National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Lecture 12: Virtual Memory

CS10014 Computer Organization

Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao University

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/

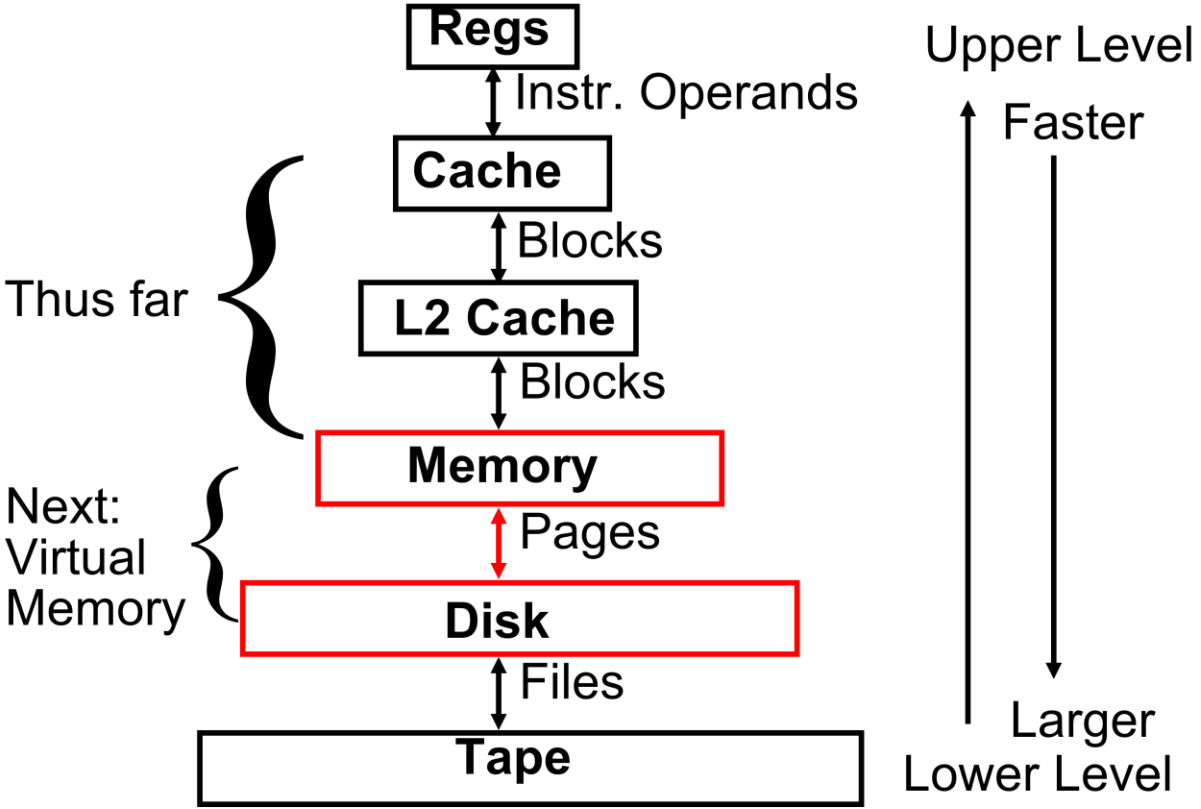
Outline

- 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



DRAM Cells

- **1 transistor and 1 capacitor per bit**
  - Contrast with SRAM 6 transistors per bit
  - Bit value is recorded as charge on the capacitor
  - Charge dissipates over time: require periodic refresh

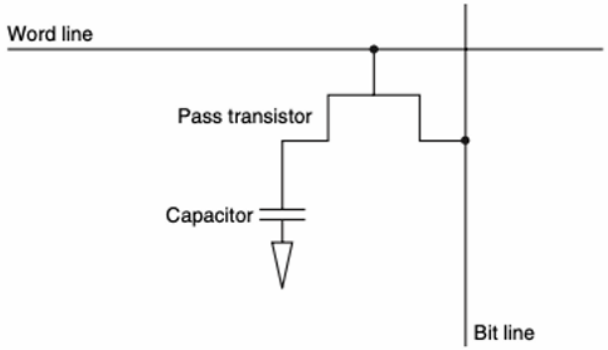
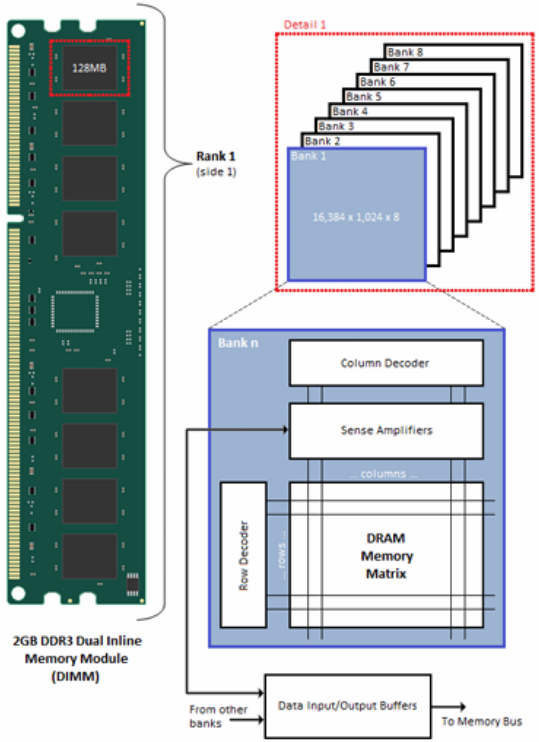


FIGURE C.9.5 A single-transistor DRAM cell contains a capacitor that stores the cell contents and a transistor used to access the cell.

c/o Patterson & Hennessy, COD, Appendix C

DRAM Internal Structure

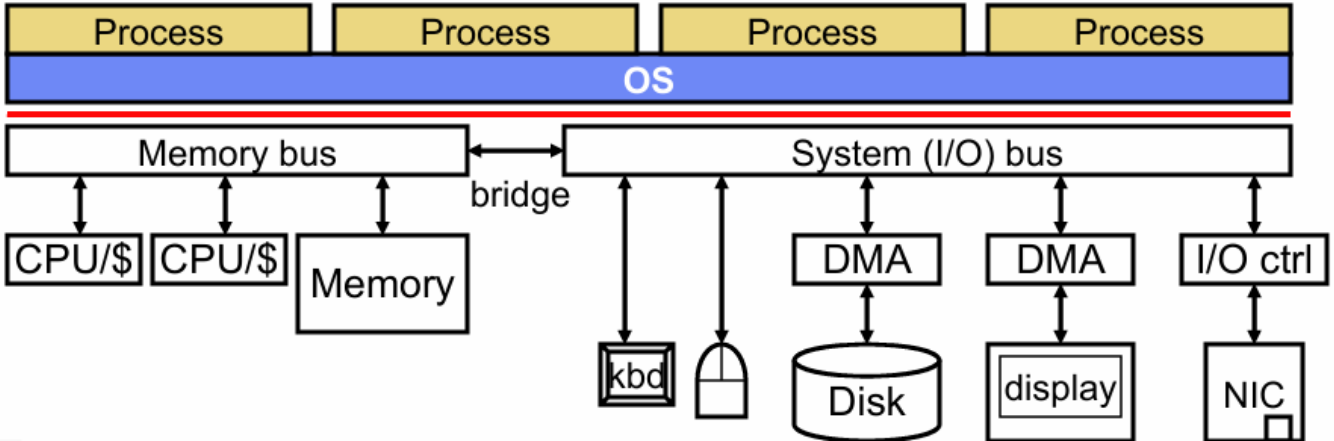
- Each **bank** composed of **cells**
- Many **banks** per **chip**
- Many **chips** per **rank**
- 1-4 **ranks** per stick/**DIMM**
- access one row at a time (~8KB) across all chips



c/o Anandtech

A Computer System + OS

- **Operating System (OS):** virtualizes hardware
  - **Abstraction**: provides **services** (e.g., threads, files, etc.)
    + Simplifies programming, raw hardware is nasty
  - **Isolation**: gives each process illusion of private CPU/mem/IO
    + Simplifies application programming model
    + Increases hardware resource utilization



Virtualizing Hardware Resources

- To **virtualize** a resource is to make a finite amount of a resource act like a very large/infinite amount.
- Easier to write programs with a virtualized interface
- Resources we can virtualize:
  - processors (via multitasking)
  - DRAM (via virtual memory)
  - entire machine+OS (via virtual machines)
- Key question: how do you manage state?

Multitasking: virtualizing a processor

- When multiple applications are run on the same core, the OS shares the computer between them
  - Multitasking dates back to the early days of computers when systems were expensive and we needed to be able to support multiple users concurrently.
- The OS needs to be able to switch between different processes. In most cases it does this so quickly that users/processes don't realize the machine is being shared.
- The act of switching between processes is referred to as a **context switch**
  - what **state** is involved?

When to perform a context switch?

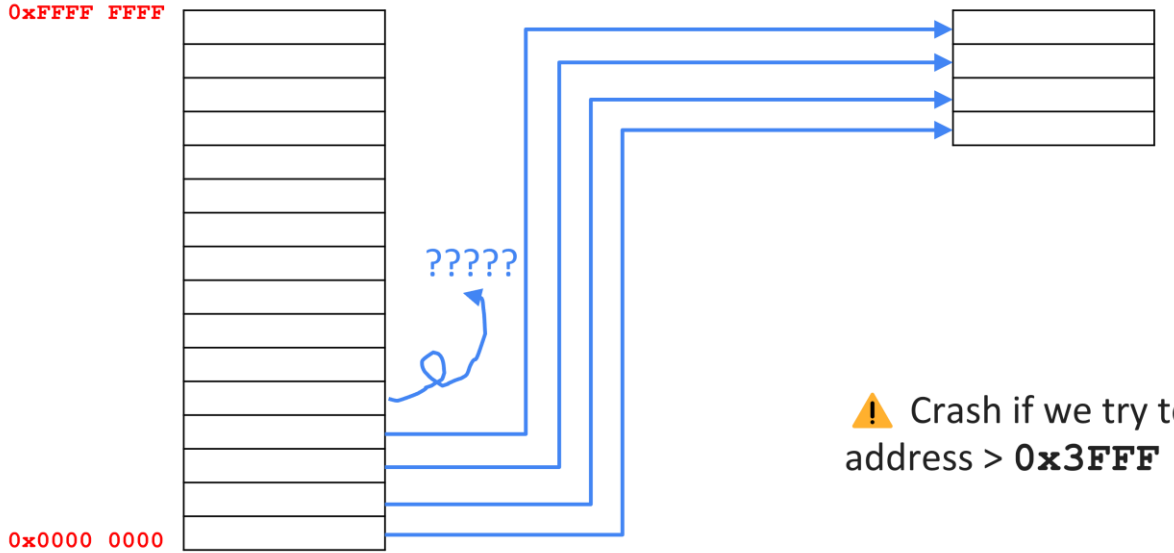
- Hardware timer ensures each process gets fair access to the CPU
- When do we switch?
  - when the timer goes off (e.g., every 2ms)
  - At system calls because they are usually slow
    - we want to do something else useful in the meantime, so switch to another process and run it instead

Virtual Memory Motivation

- 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.



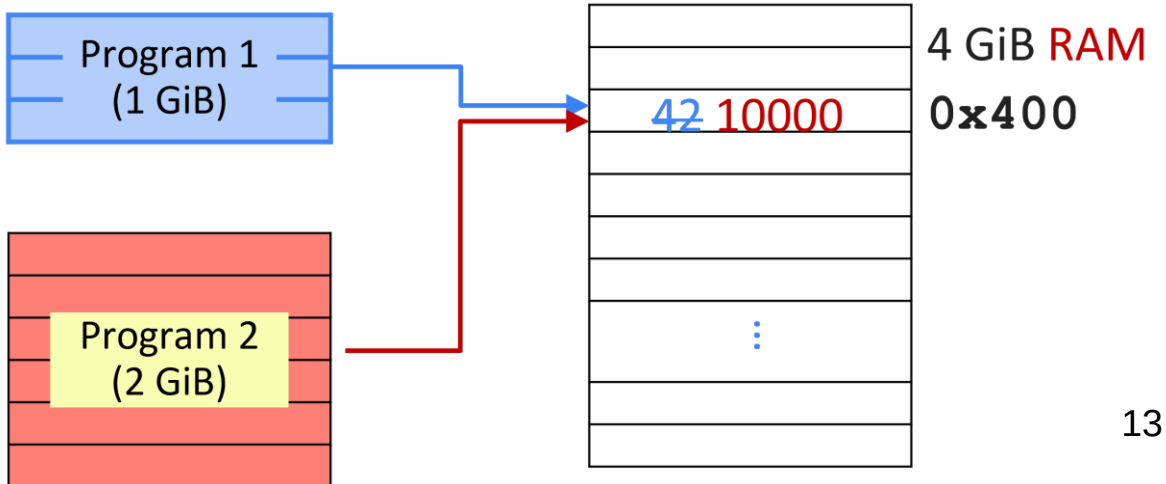
⚠ Crash if we try to access an address > **0x3FFF FFFF**!

Virtual Memory Motivation

- 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**

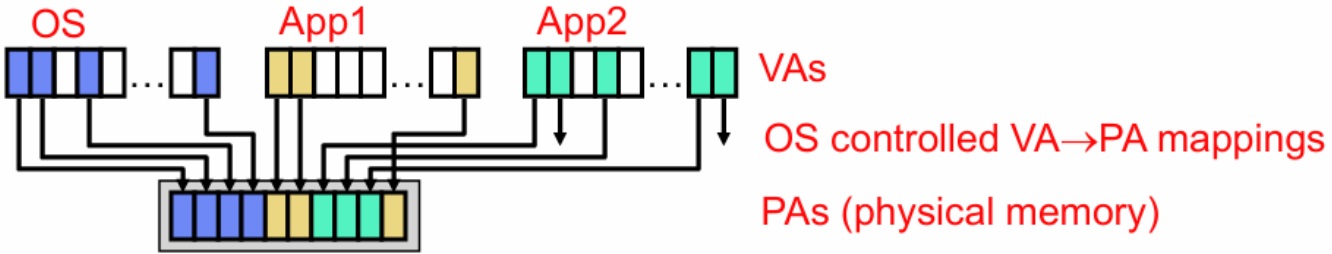


Virtualizing Main Memory

- How do we share main memory?
  - **Goal: each application thinks it owns all of memory**
- A process may want more memory than exists…
  - Process 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”
- Solution:
  - 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)

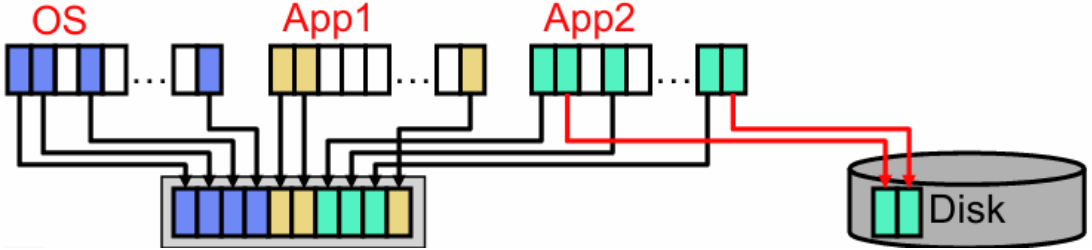
Virtualizing Memory

- **Virtual Memory (VM)**
  - Level of indirection
  - Application generated addresses are **virtual addresses (VAs)**
    - A process ***thinks*** it has its own 2^N bytes of address space
  - Memory accessed using **physical addresses (PAs)**
  - VAs translated to PAs at coarse (page) granularity
  - OS controls VA to PA mapping for itself and all processes
  - Logically: translate before every insn fetch, load, store
    - but hardware acceleration removes translation overhead

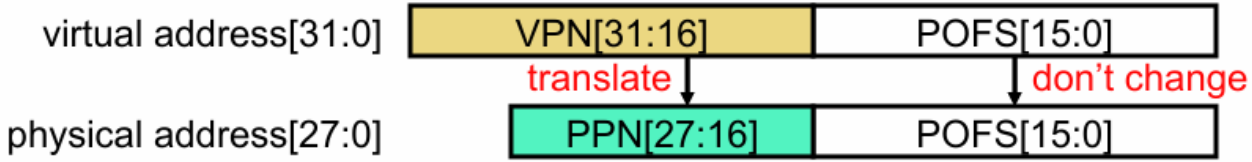


Virtualizing Memory

- Programs use **virtual addresses (VA)**
  - VA size (N) aka pointer size (these days, 64 bits)
- Memory uses **physical addresses (PA)**
  - PA size (M) typically M<N, often M=48/56 and N=64
  - 2^M is most physical memory machine supports
- 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 unaccessed)



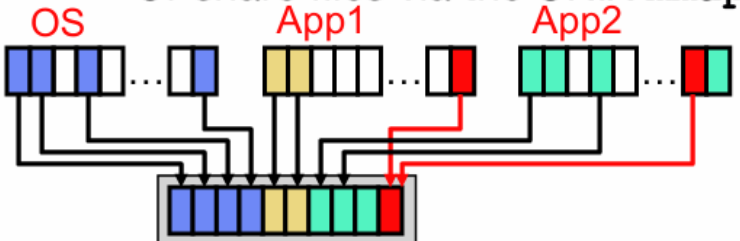
Address Translation



- 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 256MB memory → 28-bit PA → 12-bit PPN

Uses of Virtualizing Memory

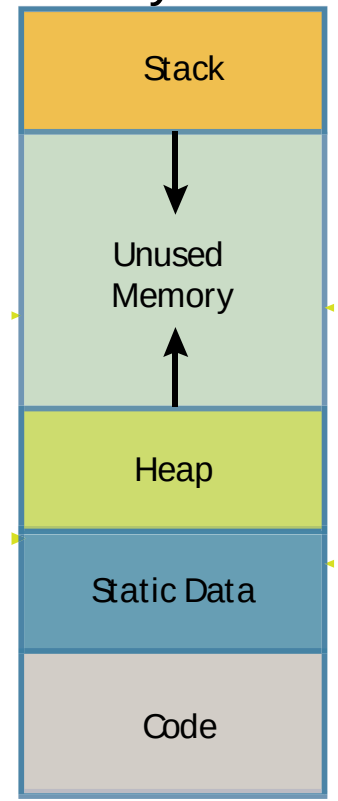
- More recently: **isolation** and **multi-programming**
  - Stack always starts at 0xFFFFFFFF for each process
  - Apps prevented from reading/writing each other's memory
    - Can't even address the other program's memory!
- **Protection**
  - Each page with a read/write/execute permission set by OS
  - Enforced by hardware
- **Inter-process communication**
  - Map same physical pages into multiple virtual address spaces
  - Or share files via the UNIX `mmap()` call



Virtual Memory

- 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

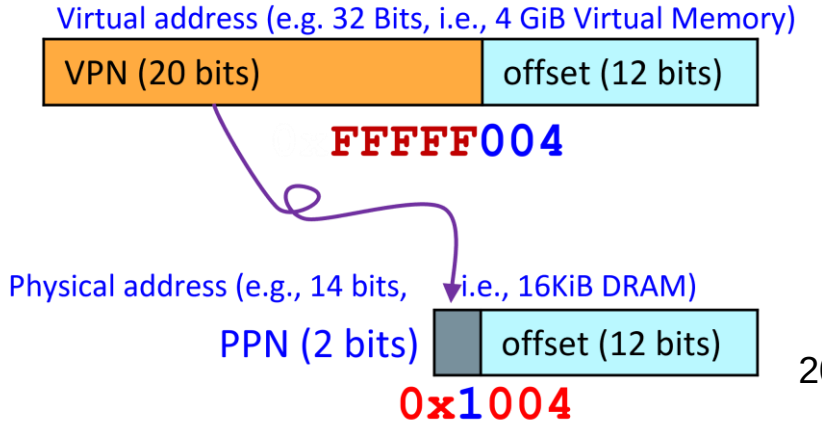
A process memory address layout



Paged Memory

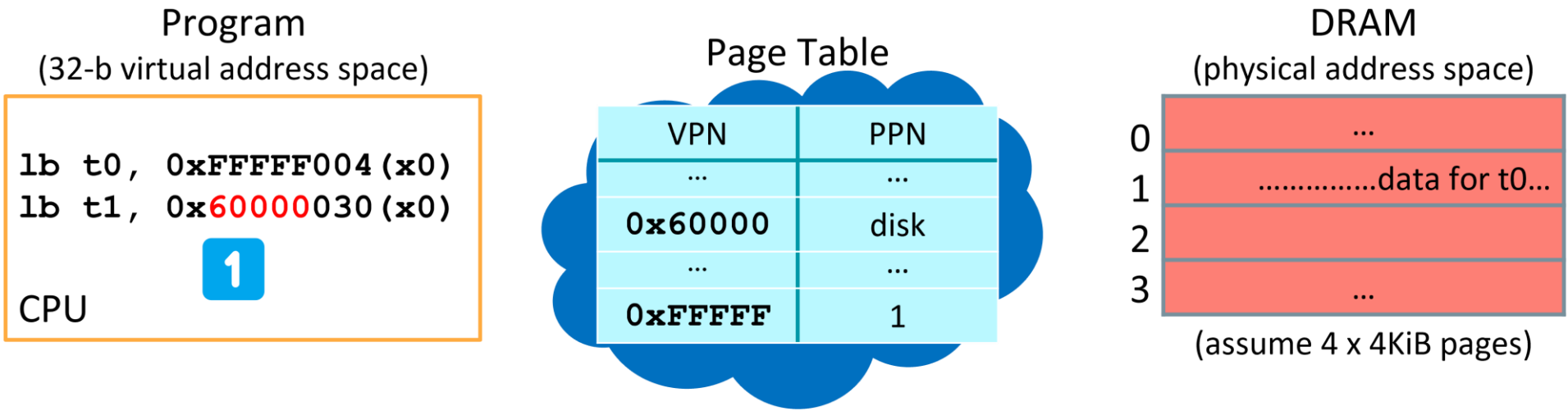
- The concept of “paged memory” dominates
  - Physical memory (DRAM) is broken into pages
  - A disk access loads an entire page into memory
  - Typical page size: 4KiB+ (on modern OSs)
    - Need 12 bits of page offset to address all 4KiB

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



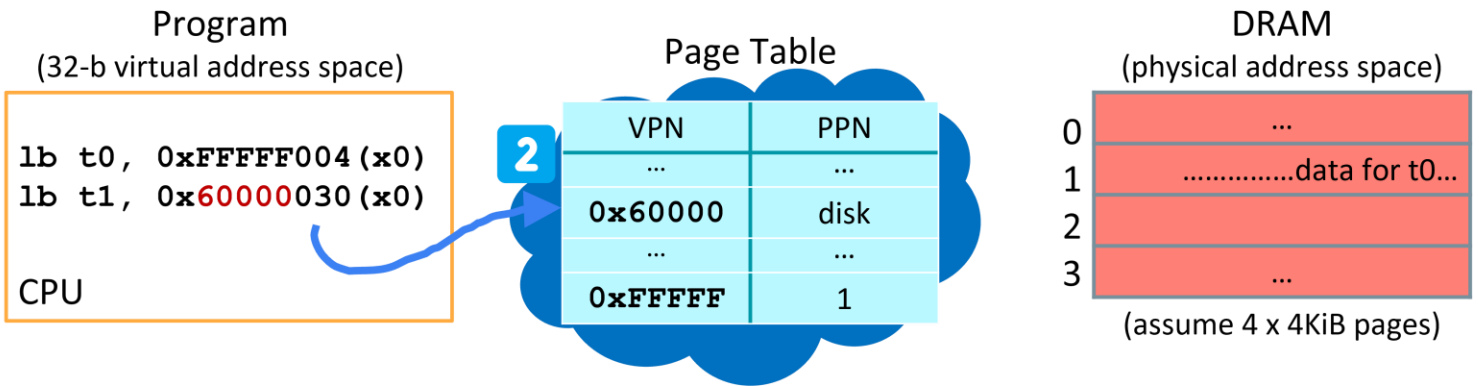
Paged Memory

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



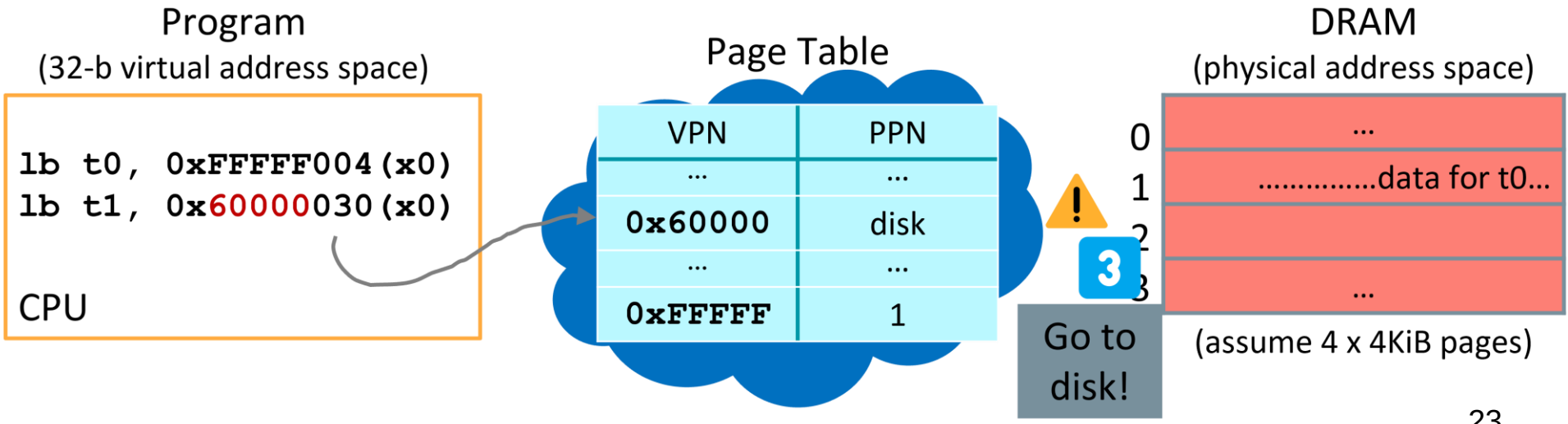
Paged Memory

- 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)



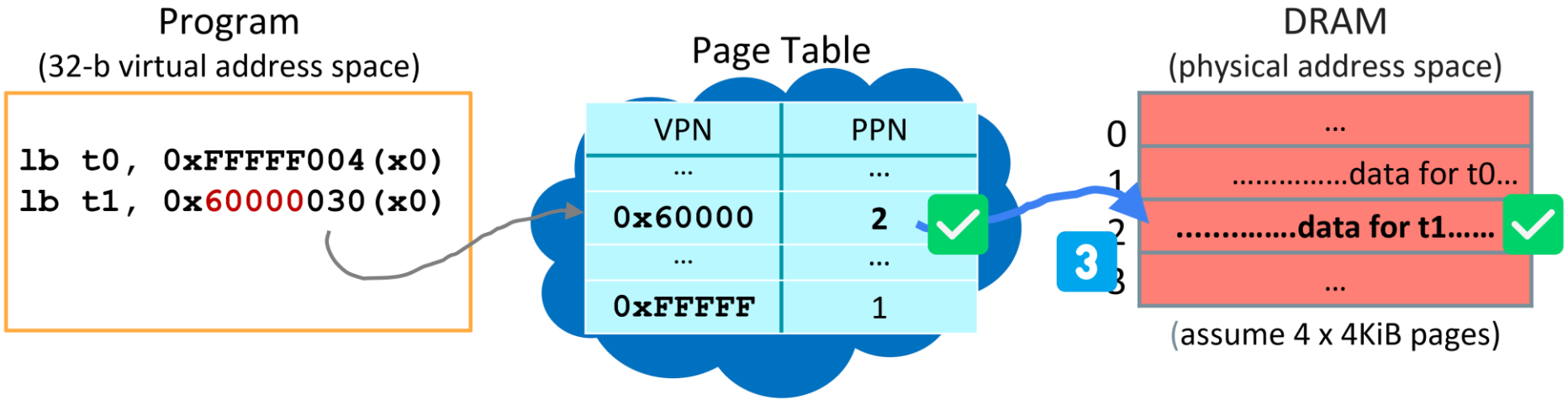
Paged Memory

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



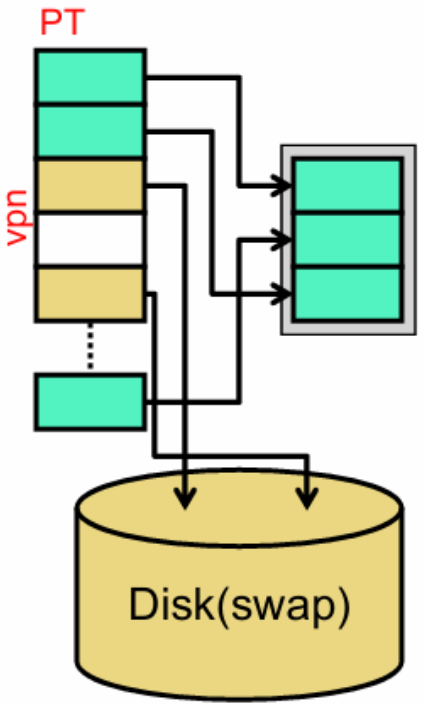
Paged Memory

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

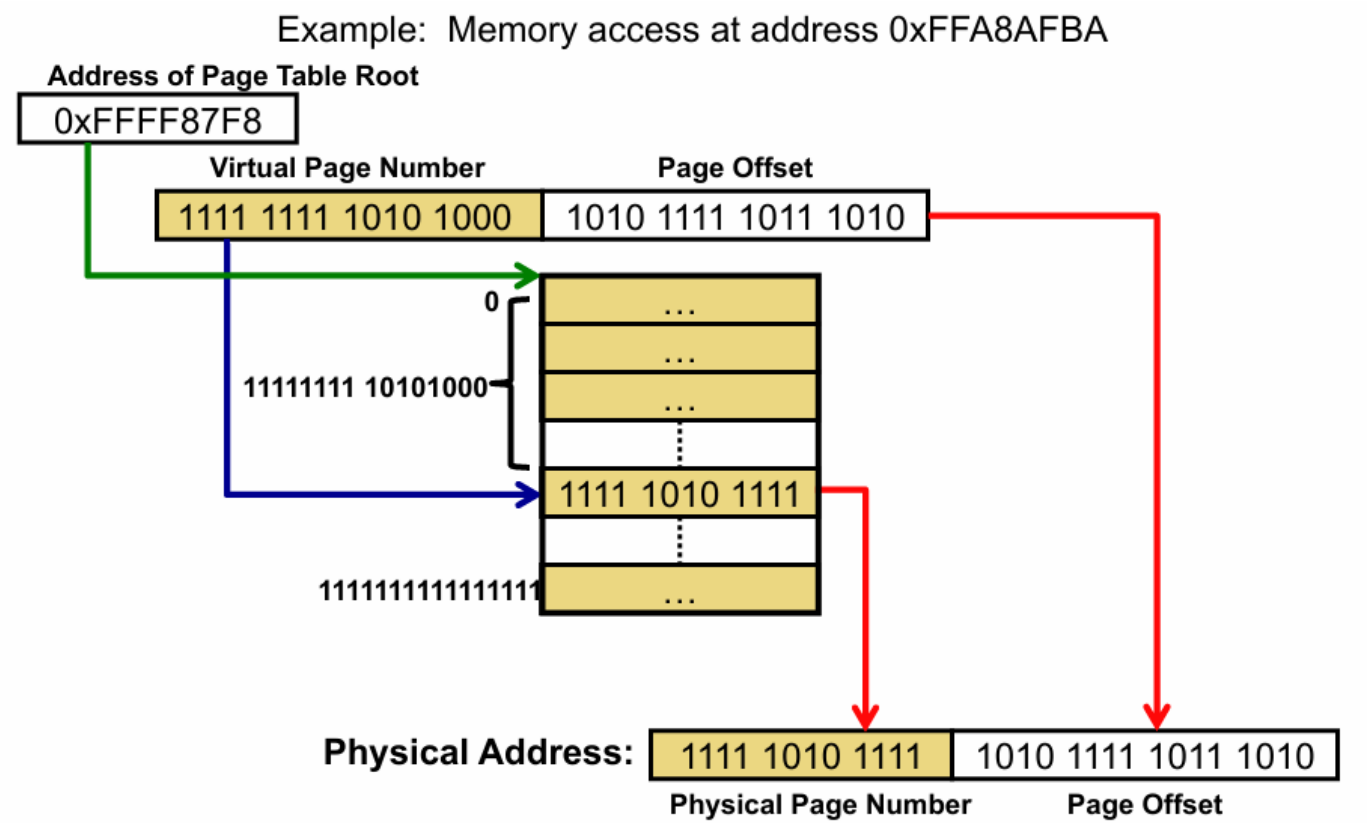


Address Translation Mechanics I

- How are addresses translated?
  - In software (for now) but with hardware acceleration (a little later)
- Each process has a **page table (PT)**
  - **Software data structure constructed by OS**
  - Maps VPs to PPs or to disk (swap) addresses
    - VP entries empty if page never referenced
  - Translation is table lookup

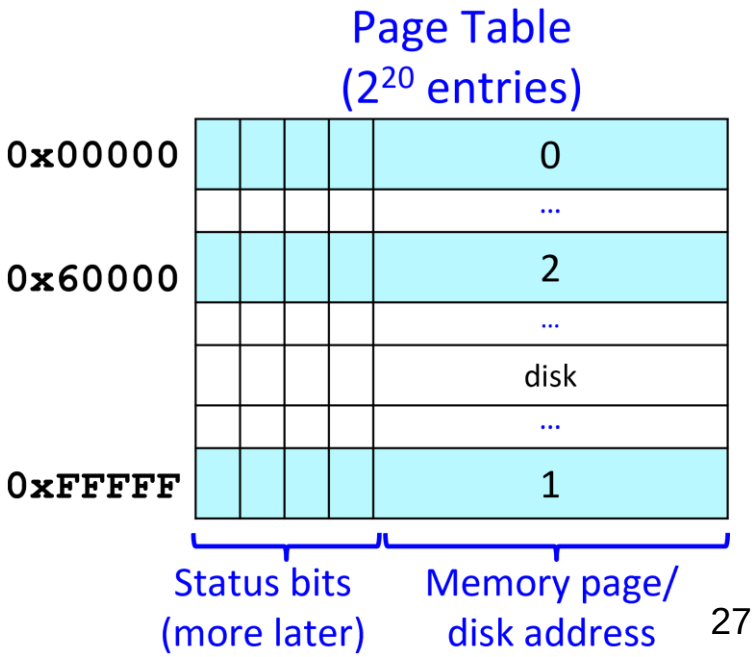


Page Table Example



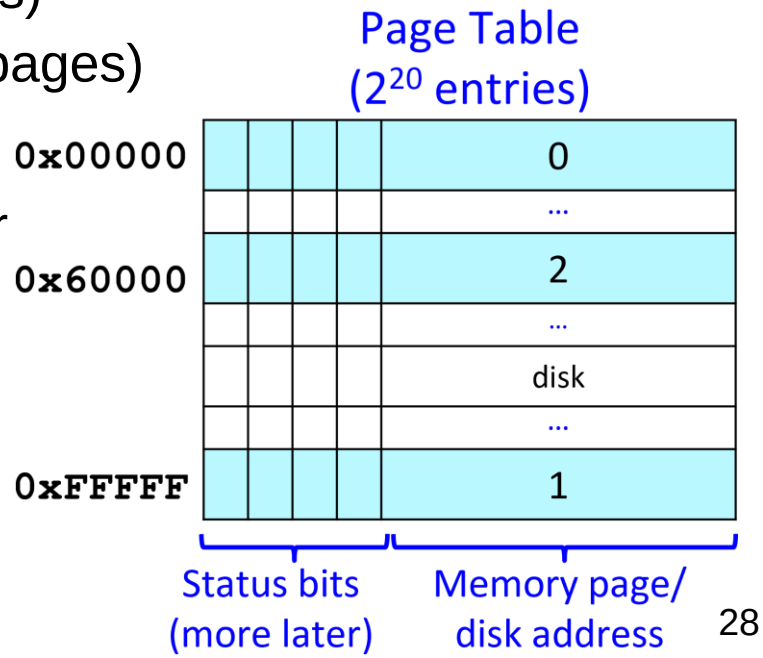
Paged Table

- 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



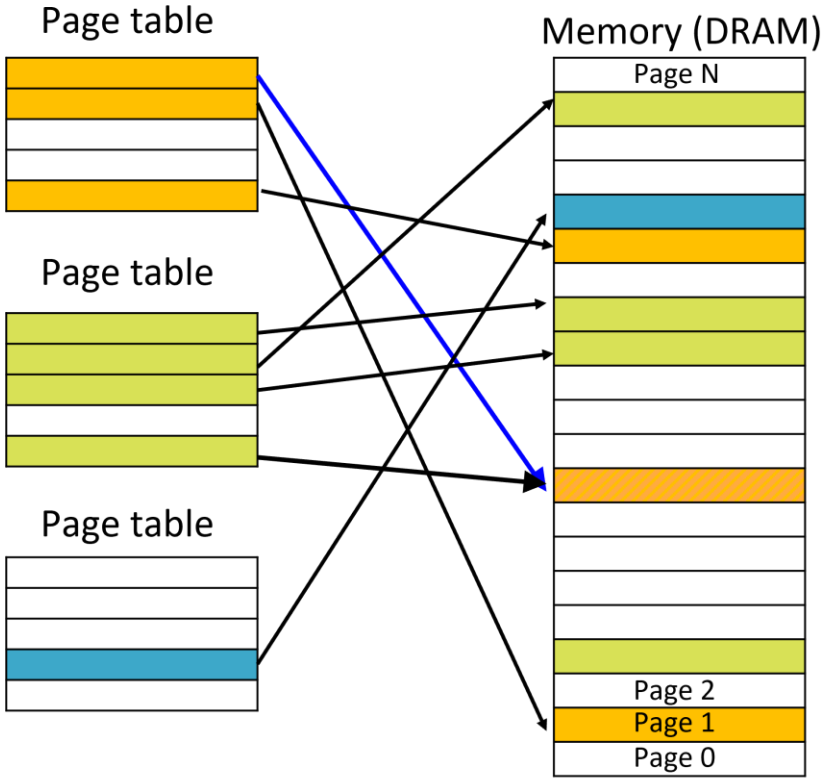
Paged Table

- 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

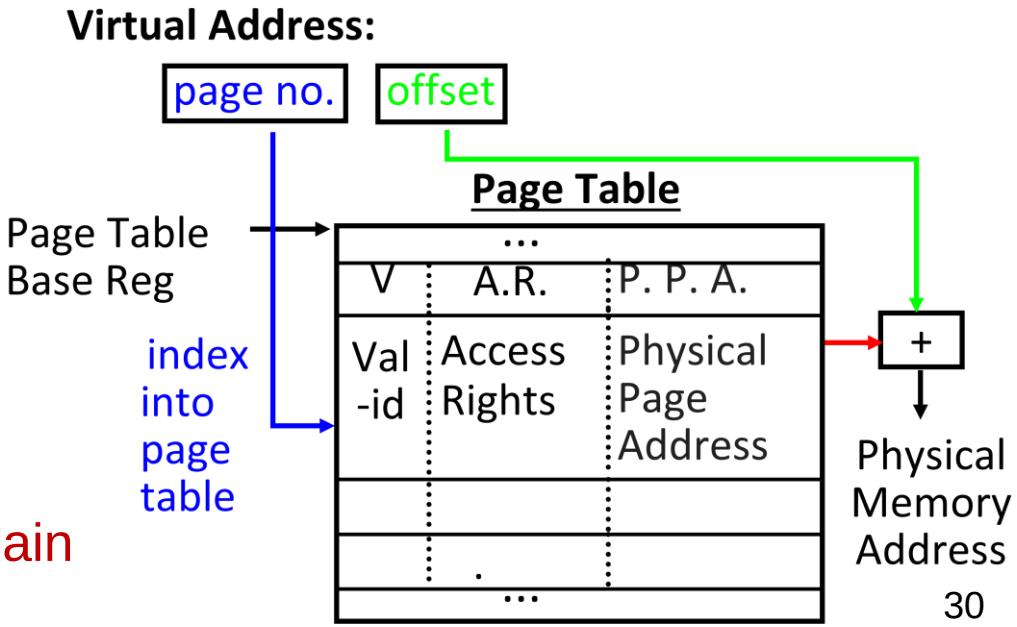
- Each process has a dedicated page table
  - OS keeps track of which process is active
- **Isolation**: Assign processes different pages in DRAM
  - Prevent accessing other processes' memory
  - OS may assign some physical page to several processes (e.g. system data), sharing is also possible



Paged Table

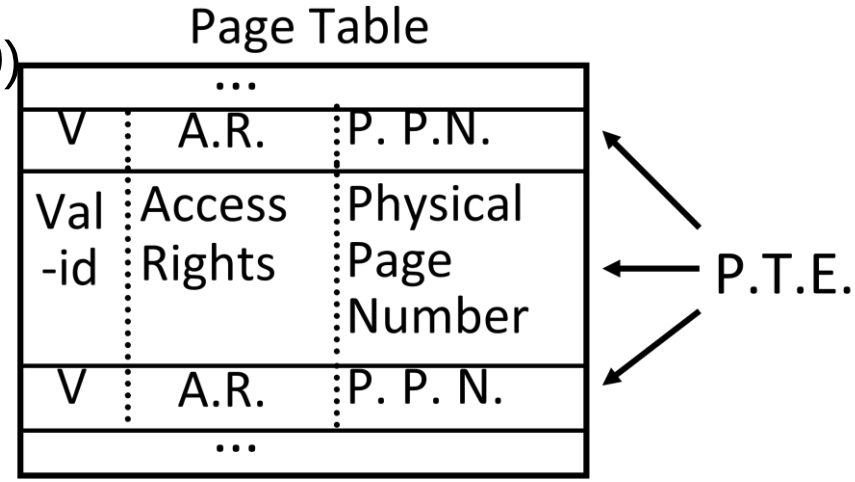
- A page table contains the mapping of virtual address to physical locations
  - Each process has its own page table
  - OS changes page tables by changing contents of Page Table Base Register

Page tables are stored in main memory



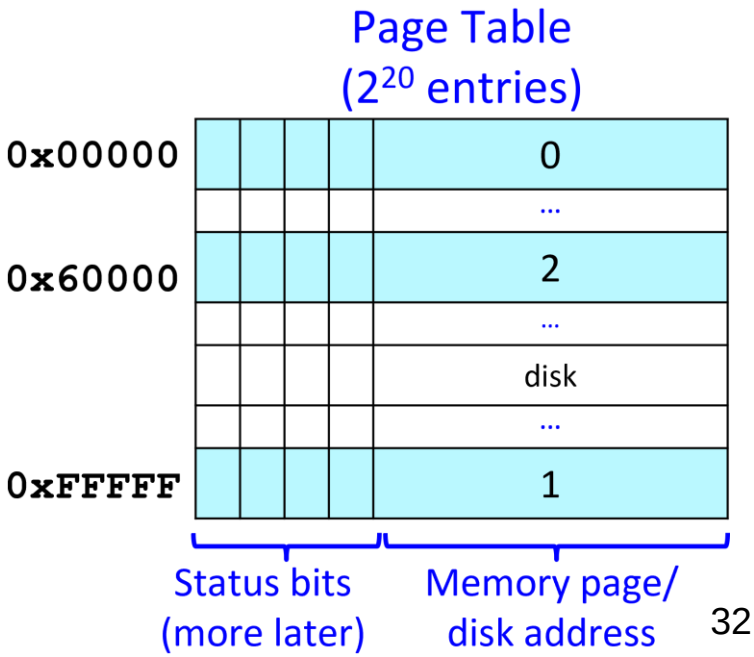
Paged Table

- **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

- **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

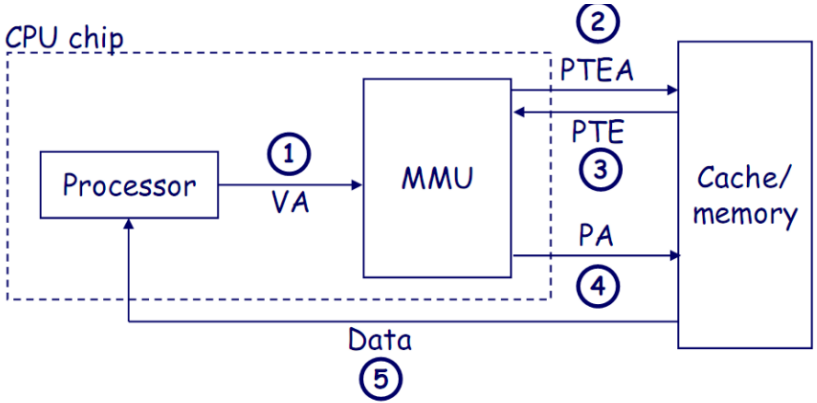


Paged Table Size

- 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

Paged Table

- **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

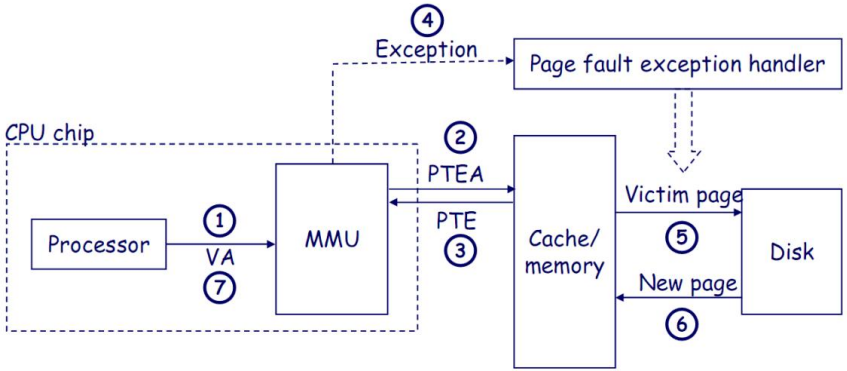
- **Page faults**
  - Page table entries store status to indicate if the page is in memory (DRAM) or only on disk
    - One each memory access, check the page table entry “valid” status bit
  - Valid -> in DRAM
    - Read/write data in DRAM
  - Not valid -> on disk
    - Trigger a page fault; OS intervenes 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

Page Fault

- **Page fault**: PTE not in TLB or page table
  - page is not in memory
  - If no valid mapping for this page → segmentation fault
  - Starts out as a TLB miss, detected by OS/hardware handler
- OS chooses a page to replace
  - **"Working set"**: refined LRU, tracks active page usage
  - If dirty, write to disk
  - Read missing page from disk
    - Takes so long (~10ms), OS schedules another task
  - Whose page are we evicting?
    - **frame map** maps physical pages to <process,virtual page> pairs
  - Update page tables, flush TLBs, retry memory access

Page Fault

- **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

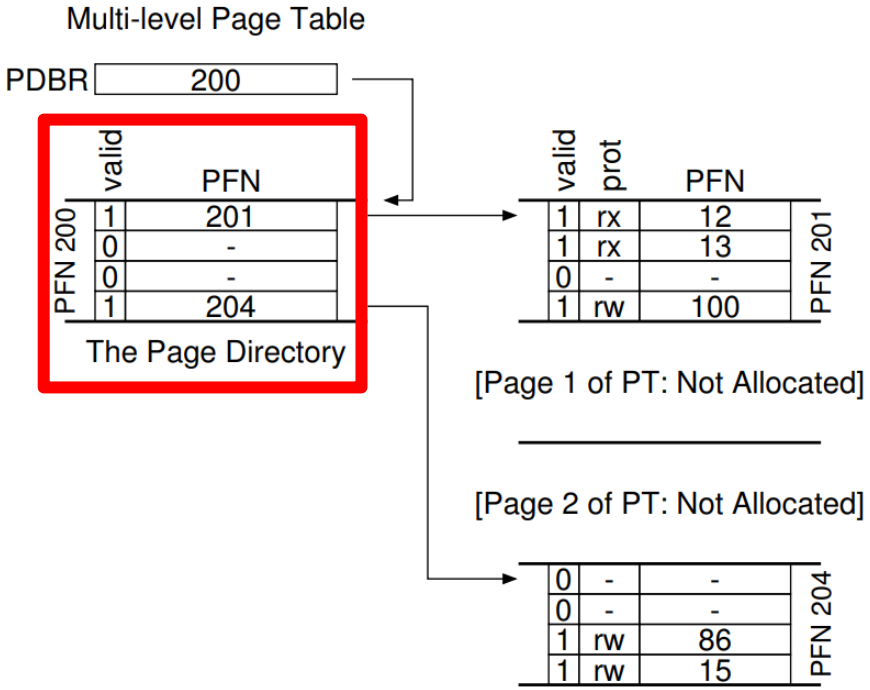


Paged Table

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

Multi-Level Page Table

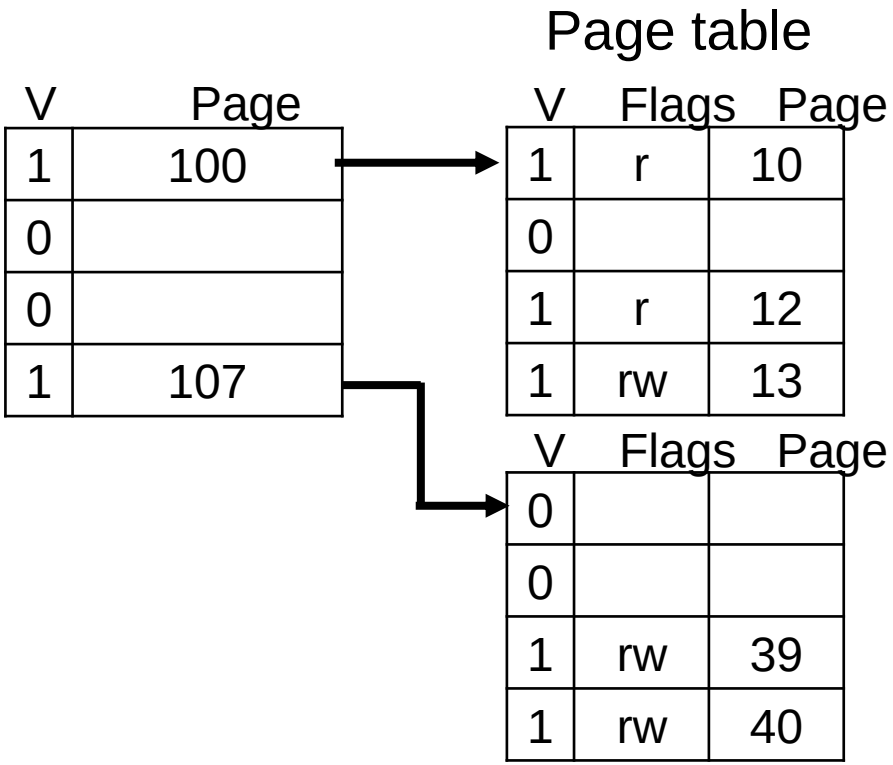
- **Multi-level page table**
  - Chop up the page table into page-sized units
  - **Page directory** tells where a page of the page table is
    - A number of **page directory entries (PDE)**
    - A **page frame number (PFN),** and a valid bit



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

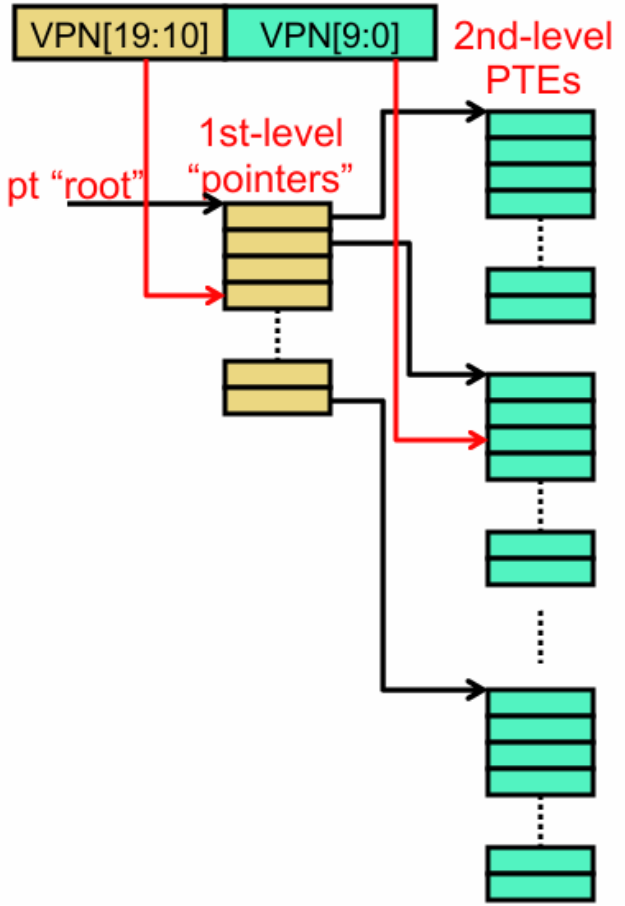
Multi-Level Page Table

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

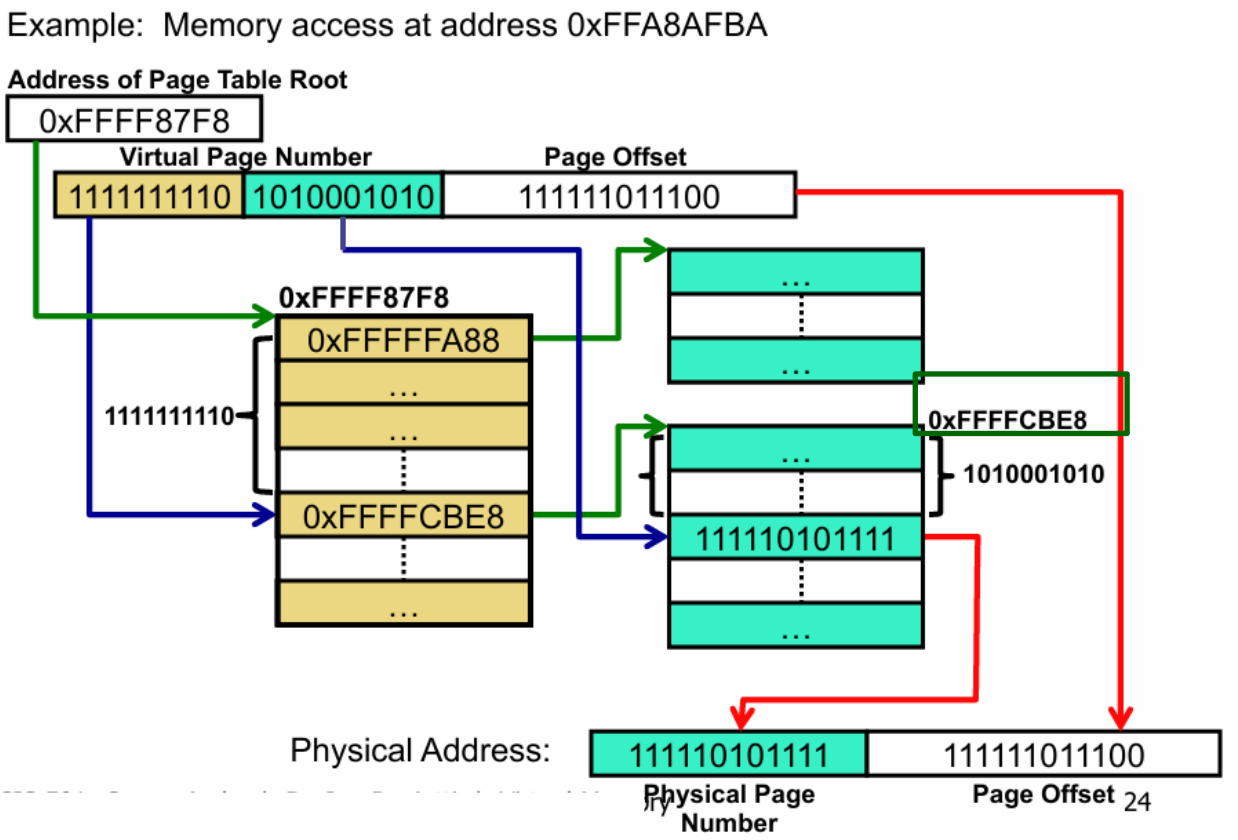


Multi-Level Page Table

- **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



Multi-Level Page Table

- **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

- 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 Bytes
- Initially
  - Page size = 8 KB = 2^{13} Bytes
  - Virtual address space size = 2^{46} Bytes
  - PTE = 4 Bytes = 2^{2} Bytes
  - Number of pages or number of entries in page table = 2^{46} Bytes / 2^{13} Bytes = 2^{33}
  - Size of page table = 2^{33} x 2^{2} Bytes = 2^{35} Bytes

Multi-Level Page Table

- 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} Bytes / 2^{13} Bytes = 2^{22}
  - Size of page table [second last level]
  - 2^{22} x 2^{2} Bytes = 2^{24} Bytes

Multi-Level Page Table

- 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} Bytes / 2^{13} Bytes = 2^{11}
  - Size of page table [third last level]=
    = 2^{11} x 2^{2} Bytes = 2^{13} Bytes = page size

Multi-Level Page Table

- 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} Bytes / 2^{13} Bytes = 2^{11}
  - Size of page table [third last level]=
    = 2^{11} x 2^{2} Bytes = 2^{13} Bytes = page size

Page Sizes

- More ISAs support multiple page sizes
  - x86: 4KB, 2MB, 1GB
- larger pages have pros and cons
  - \+ reduce page table size
    - fewer entries needed to map a given amount of address space
  - \+ page table can be shallower
    - makes page table lookups faster
  - – “internal fragmentation” that wastes physical memory
    - allocate 2MB page but use only 5KB of it
  - – complex implementation
    - OS looks for opportunities to use large pages

RV32 Page Sizes

- 32-bit virt addrs
- 4KB pages
- RV64
  - supports 4KB, 2MB, 1GB, 512GB pages

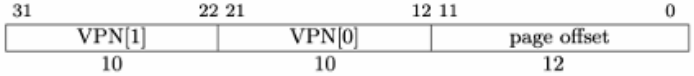


Figure 4.16: Sv32 virtual address.

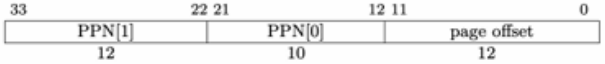


Figure 4.17: Sv32 physical address.

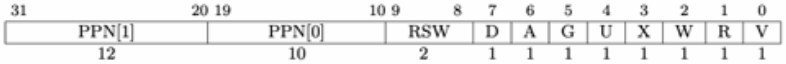


Figure 4.18: Sv32 page table entry.

X	W	R	Meaning
0	0	0	Pointer to next level of page table.
0	0	1	Read-only page.
0	1	0	*Reserved for future use.*
0	1	1	Read-write page.
1	0	0	Execute-only page.
1	0	1	Read-execute page.
1	1	0	*Reserved for future use.*
1	1	1	Read-write-execute page.

Table 4.5: Encoding of PTE R/W/X fields.

Page Protections

- Piggy-back on page-table mechanism
- Map VPN to PPN + Read/Write/Execute permission bits
- If you attempt to execute data, to write read-only data, or to read unmapped data…
  - Exception → OS terminates program
  - this is what a **segmentation fault** is
  - helps protect against bugs and security vulnerabilities
- Useful for processes, and even for OS itself

Address Translation Mechanics II

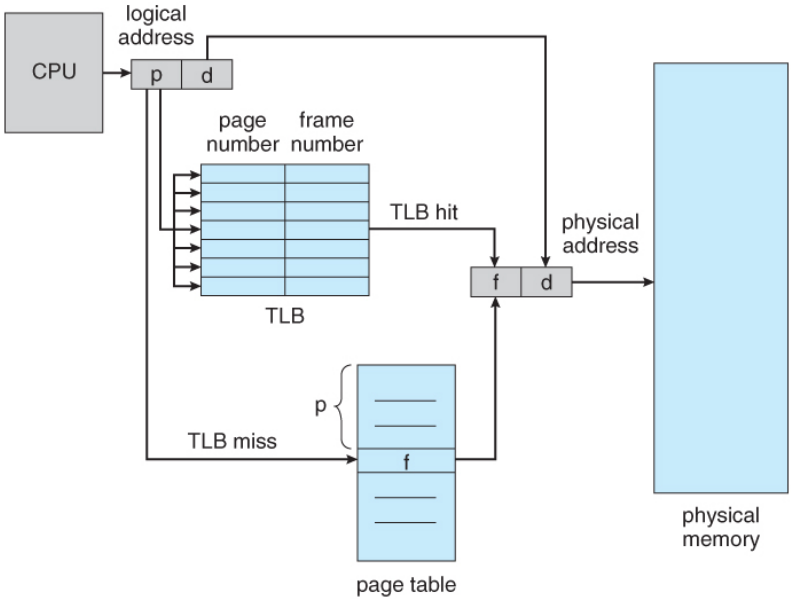
- Conceptually
  - Translate VA to PA **before every cache access**
  - Walk the page table before every load/store/insn-fetch
  - – Would be terribly inefficient (even in hardware)
- In reality
  - **Translation Lookaside Buffer (TLB)**: cache translations
  - Only walk page table on TLB miss
- Computer system design truisms
  - Functionality problem? Add indirection (e.g., VM)
  - Performance problem? Add cache (e.g., TLB)

Translation Lookaside Buffer (TLB)

- 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**

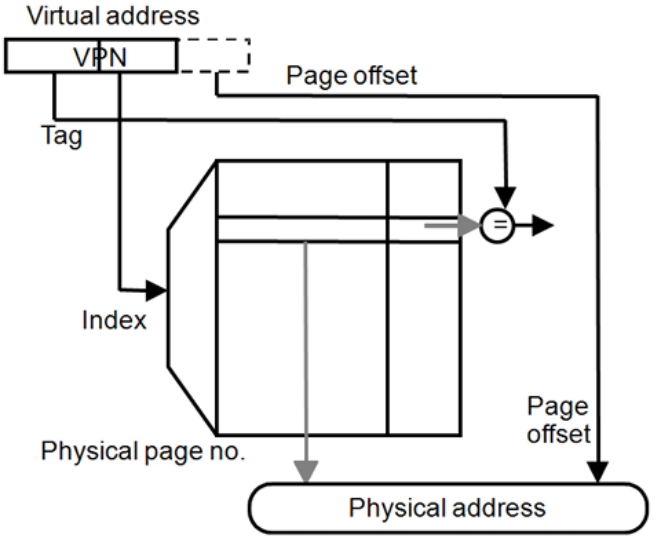
Translation Lookaside Buffer (TLB)

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

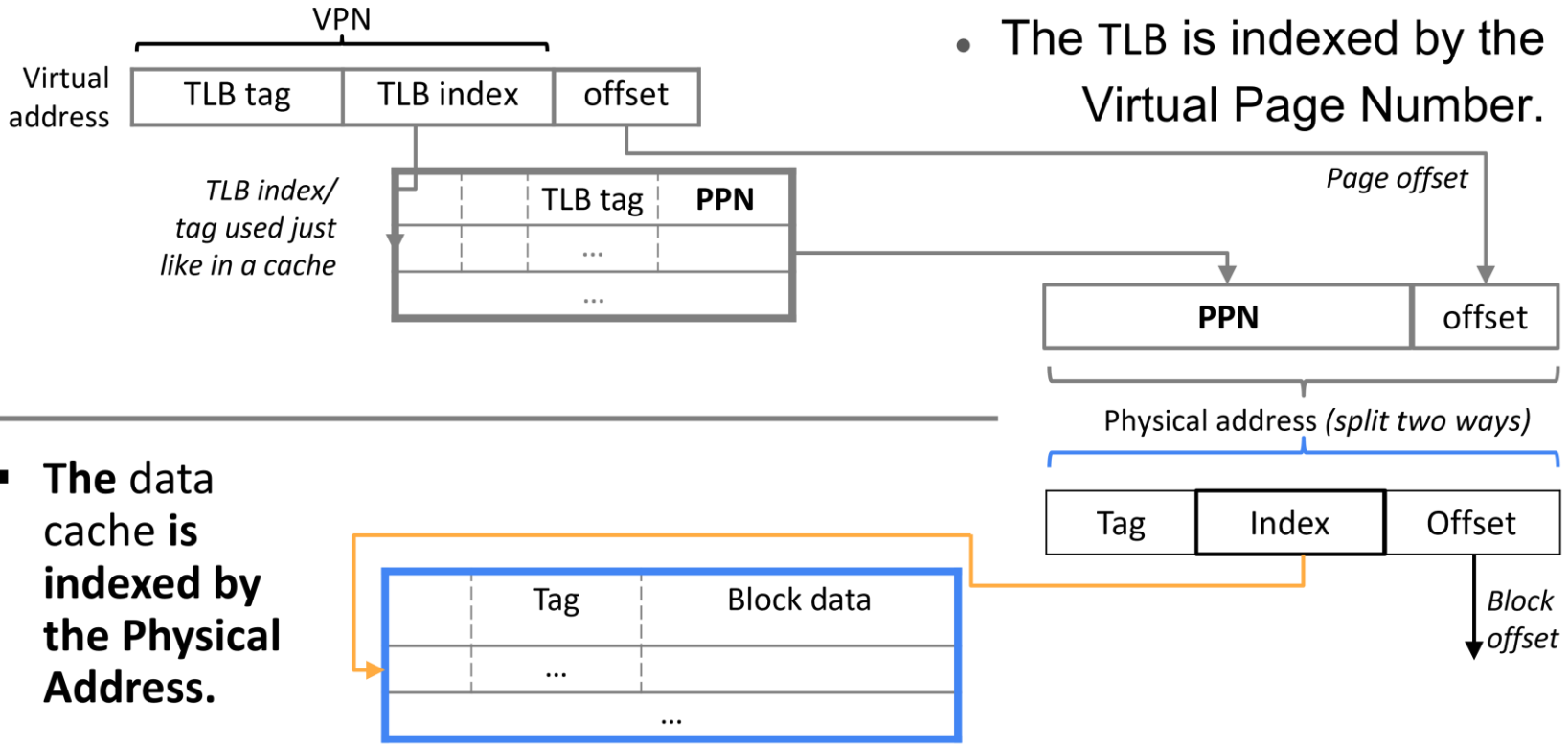


Translation Lookaside Buffer (TLB)

- 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)



- The TLB is indexed by the Virtual Page Number.

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

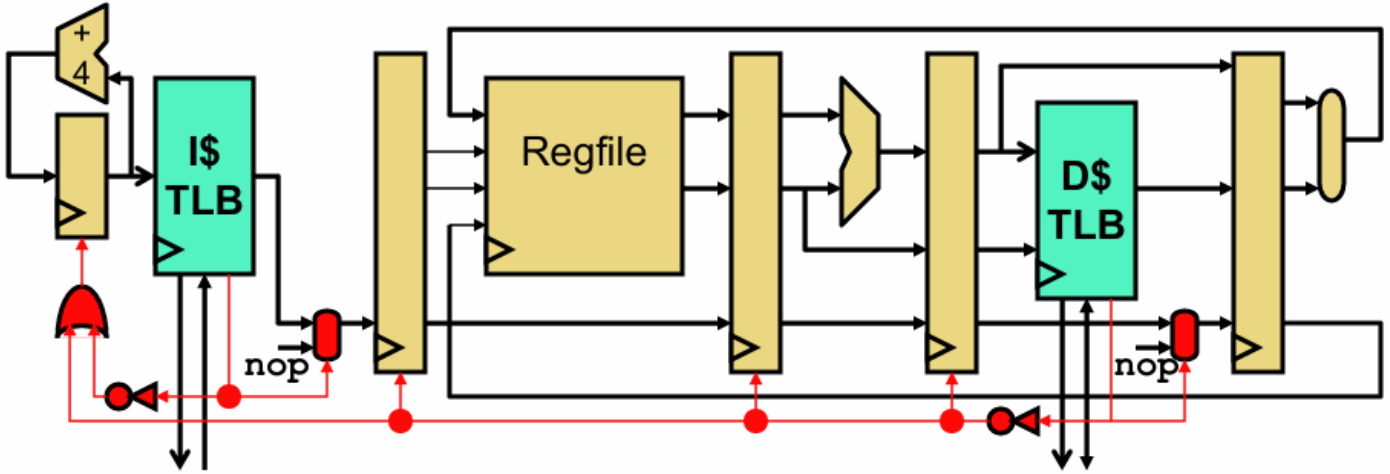
TLB Organization

- **Like caches**: TLBs also have ABCs
  - **Capacity**
  - **Associativity** (≥4-way associative, full-assoc common)
  - What does it mean for a TLB to have a **block size** of two?
    - Two VPs can a single tag
    - VPs must be aligned and consecutive
  - **Like caches**: there are second-level TLBs
- Example: AMD Opteron
  - 32-entry FA TLBs, 512-entry 4-way L2 TLB (insn & data)
  - 4KB pages, 48-bit virtual addresses, four-level page table
- TLB should **"cover"** size of on-chip caches
  - (#PTEs in TLB) * page_size ≥ cache_size
  - Why? Consider relative miss latency in each…

TLB Misses

- **TLB miss:** translation not in TLB, but in page table
  - Two ways to "fill" it, both relatively fast
- **Hardware-managed TLB**: x86, ARM, RV
  - Page table root in hardware register, hardware "walks" table
  + Latency: saves cost of OS call (avoids pipeline flush)
  – Page table format is hard-coded
- **Software-managed TLB**: Alpha, MIPS
  - Short (~10 insn) OS routine walks page table, updates TLB
  + Keeps page table format flexible
  – Latency: 1-2 memory accesses + OS call (pipeline flush)

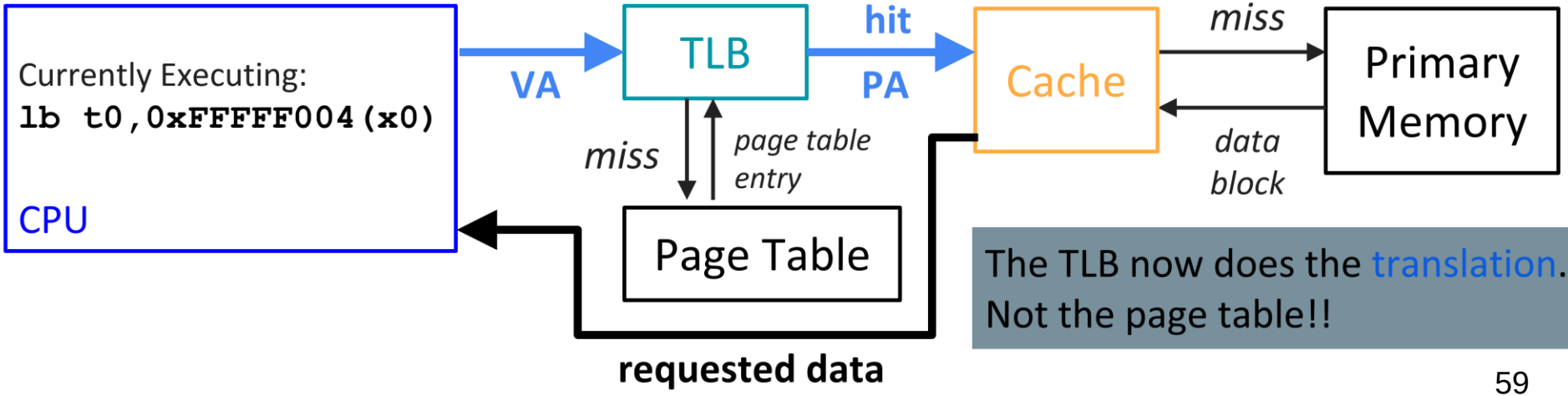
TLB Misses and Pipeline Stalls



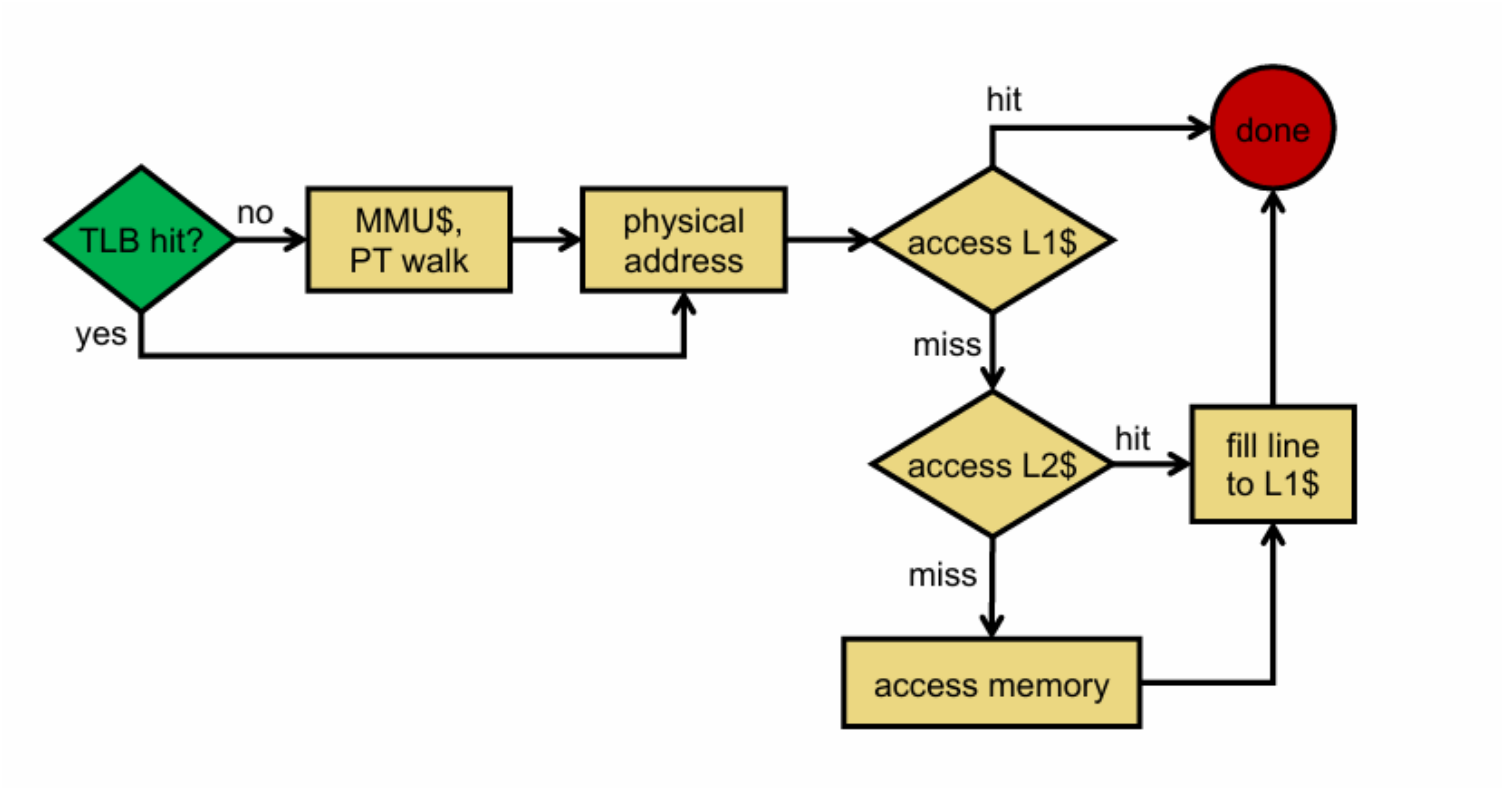
- TLB misses stall pipeline just like data hazards...
  - ...if TLB is hardware-managed
- If TLB is software-managed...
  - ...must generate an interrupt
  - Hardware will not handle TLB miss

Memory Access

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



The Life of Virtual Memory Access



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