ECE 550D
Fundamentals of Computer Systems and Engineering

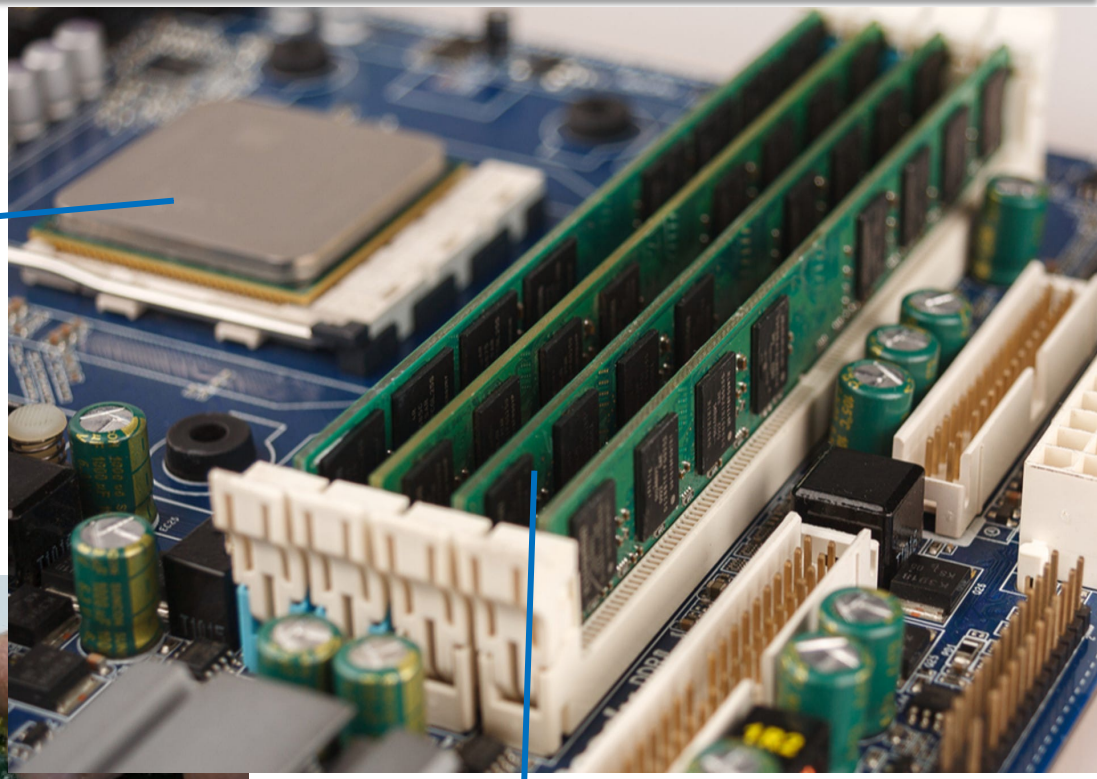
Fall 2022

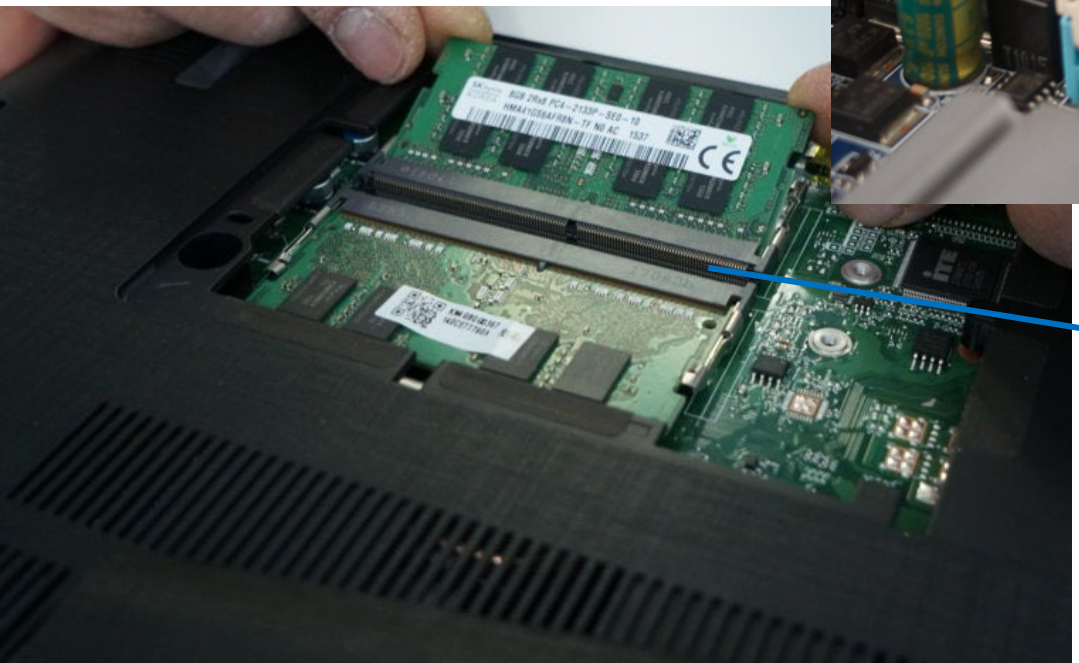
Virtual Memory

Rabih Younes
Duke University

Slides are derived from work by
Andrew Hilton and Tyler Bletsch (Duke)

Main Memory (DRAM)





Problems With Our Current Approach to Memory?

- Reasonable (main) memory: 4GB—64GB?
  - In 32-bit systems:
    - Program can address 2^32 bytes = **4GB**/program
  - In 64-bit systems:
    - Program can address 2^64 bytes = **16EB**/program!
- What if we're running many programs, not just 1?
  - Impossible using what we know for now

- → We need an approach called: **virtual memory**
  - Gives every program **the illusion** of having access to the entire address space
  - Hardware and OS (operating system) move things around behind the scenes
- How?
  - Good rule to know: when we have a functionality problem
    → we can usually solve it by adding a level of indirection

Virtual Memory

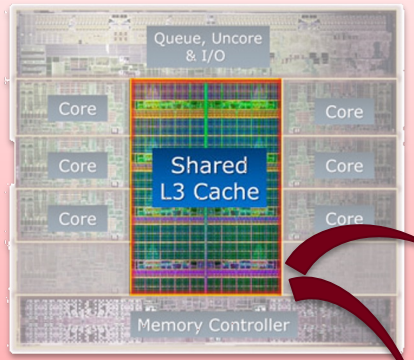
- Predates "caches" (by a little)

- Original motivation: **compatibility**
  - Ability to run the same program on machines with different main memory sizes
  - Prior to virtual memory, programmers needed to explicitly account for memory size

- **Virtual memory:**
  - Treats memory like a cache for disks (or other secondary storage)
    - Disks should be able to contain all our data
  - Contents of memory would be a dynamic subset of program's address space
  - Dynamic content management of memory is transparent to program
    - Caching mechanism makes it appear as if memory is 2^N bytes regardless of how much memory there actually is

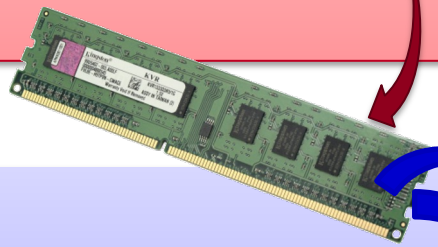
Caching vs. Virtual Memory

CACHING



Copy **block** if **popular**

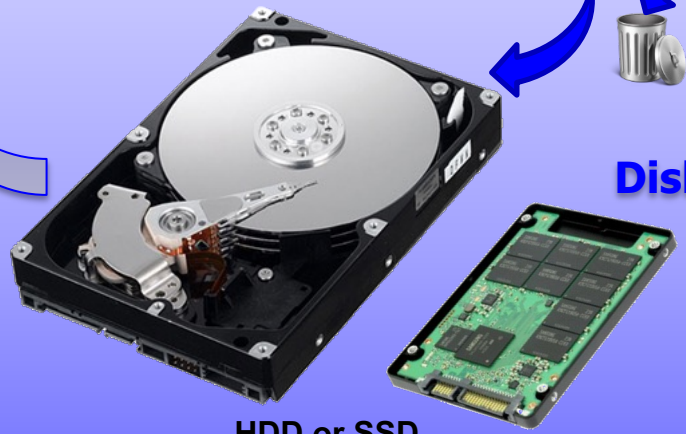
Cache
- Faster
- More expensive
- Lower capacity



RAM

VIRTUAL MEMORY

Load **page** if **needed**



HDD or SSD

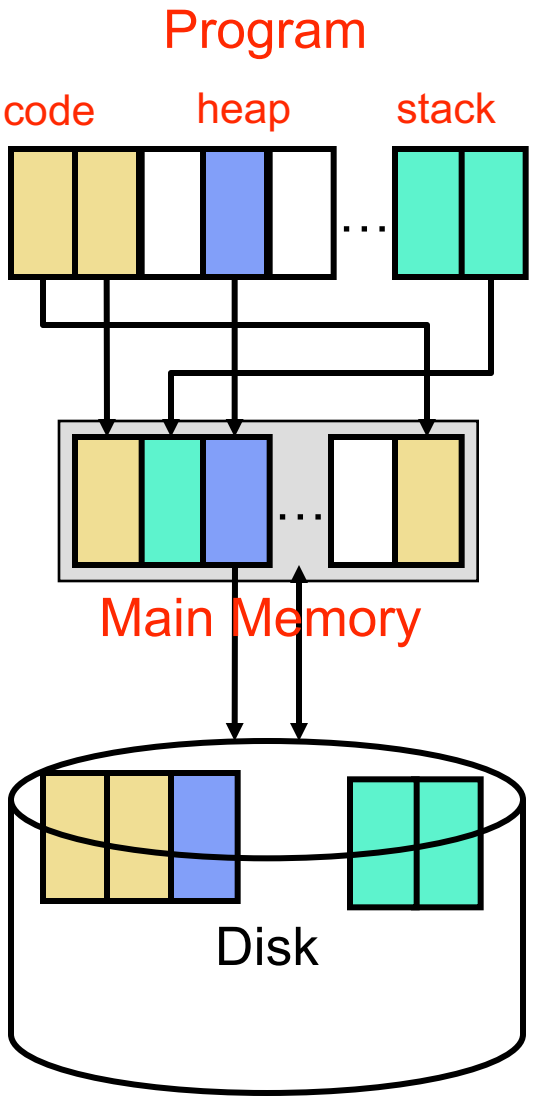
Disk
- Slower
- Cheaper
- Higher capacity

Pages

- What is swapped between disk and memory?
  - A **page** (vs. **block** in caching)
    - Using a process called "demand paging" (or "paging")

- **Page:** A small chunk (~4KB) of memory with its own record in the memory management hardware

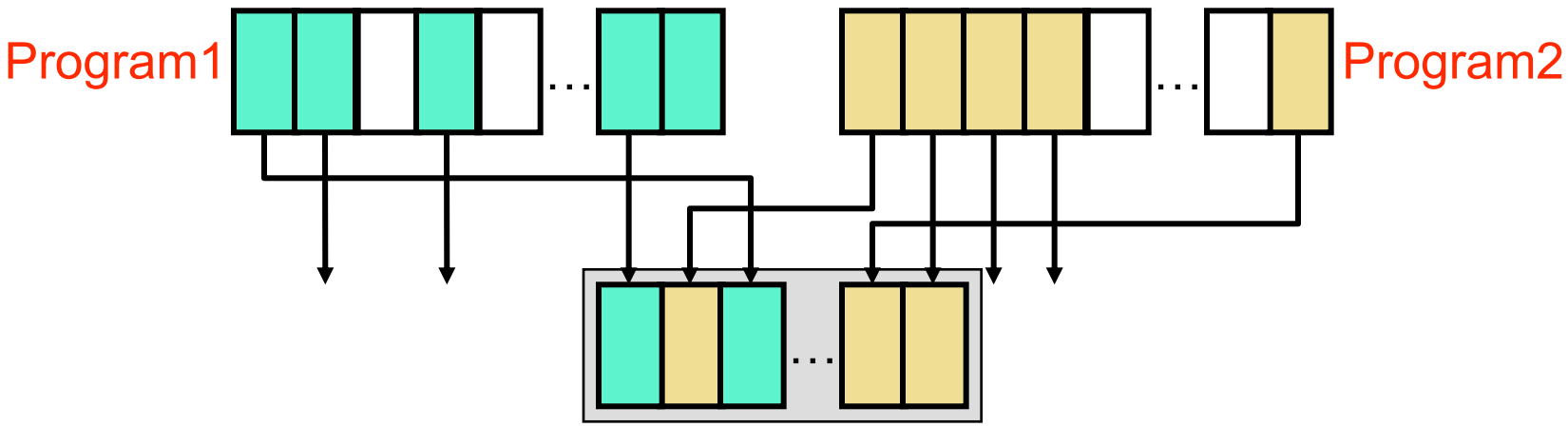
How Virtual Memory Solves Our Problem(s)



- Programs use **virtual addresses (VA)**
  - 0 to 2^N–1
  - N is the machine/system size (bus width)
    - E.g., Pentium4 is 32-bit, Core i9 is 64-bit
- Memory uses **physical addresses (PA)**
  - 0 to 2^M–1 (M<N, especially if N=64)
  - 2^M is most physical memory machine supports
- VA to PA translation at page granularity
  - → **VP to PP translation**
    (Virtual Page to Physical Page)

Other Uses of Virtual Memory

- Virtual memory is quite useful for 1 program, but is also very useful for **multiprogramming** (more than 1 program)
  - Each process thinks it has 2^N bytes of address space
  - Each thinks its stack starts at address 0xFFFFFFFF
  - "System" maps VPs from different processes to different PPs
    + Prevents processes from reading/writing each other's memory



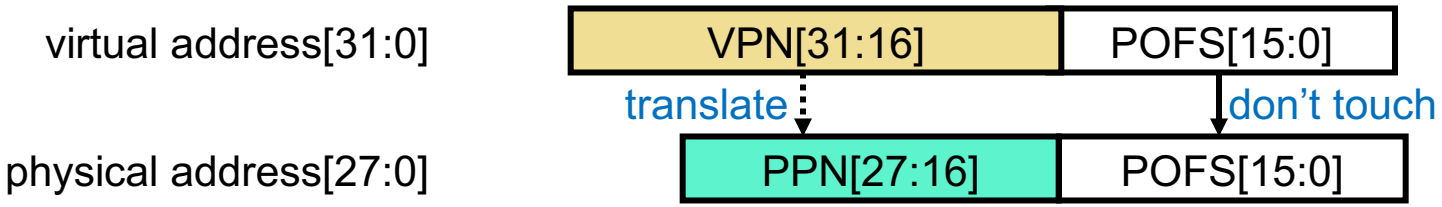
Even More Uses of Virtual Memory

- Inter-process communication
  - Map VPs in different processes to same PPs
- Direct memory access I/O
  - Think of I/O device as another process
  - Will talk more about I/O in the future
- Protection
  - Piggy-back mechanism to implement page-level protection
  - Map VP to PP … and RWX protection bits
  - Attempt to execute data, or attempt to write insn/read-only data?
    - Exception → OS terminates program

Address Translation (VA→PA or VP→PP)

Address Translation

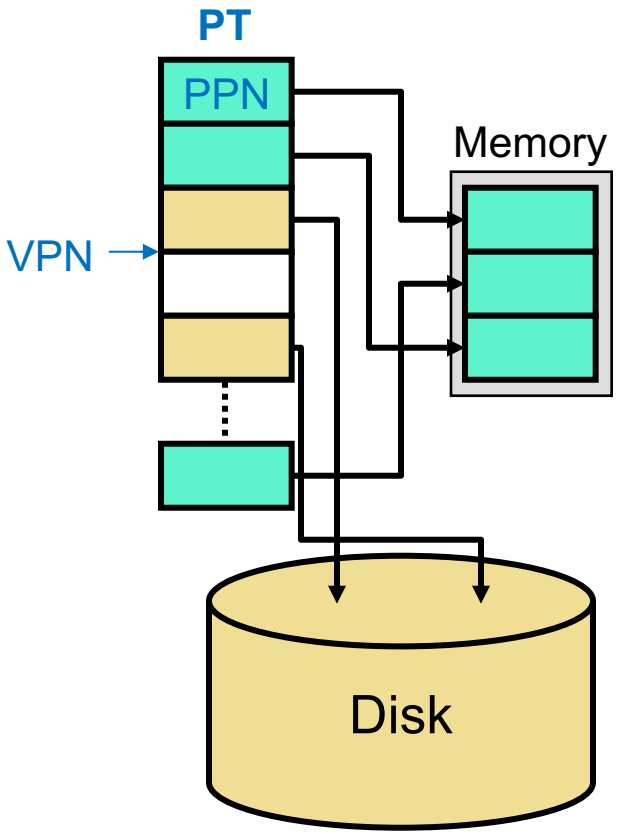
- VA→PA mapping is called **address translation**
  - Split VA into virtual page number (VPN) and page offset (POFS)
  - Translate VPN into physical page number (PPN)
  - POFS is not translated
    - Why? Because it takes us to the desired byte in a page, regardless of where that page is residing
  - VA→PA = [VPN, POFS]→[PPN, POFS]



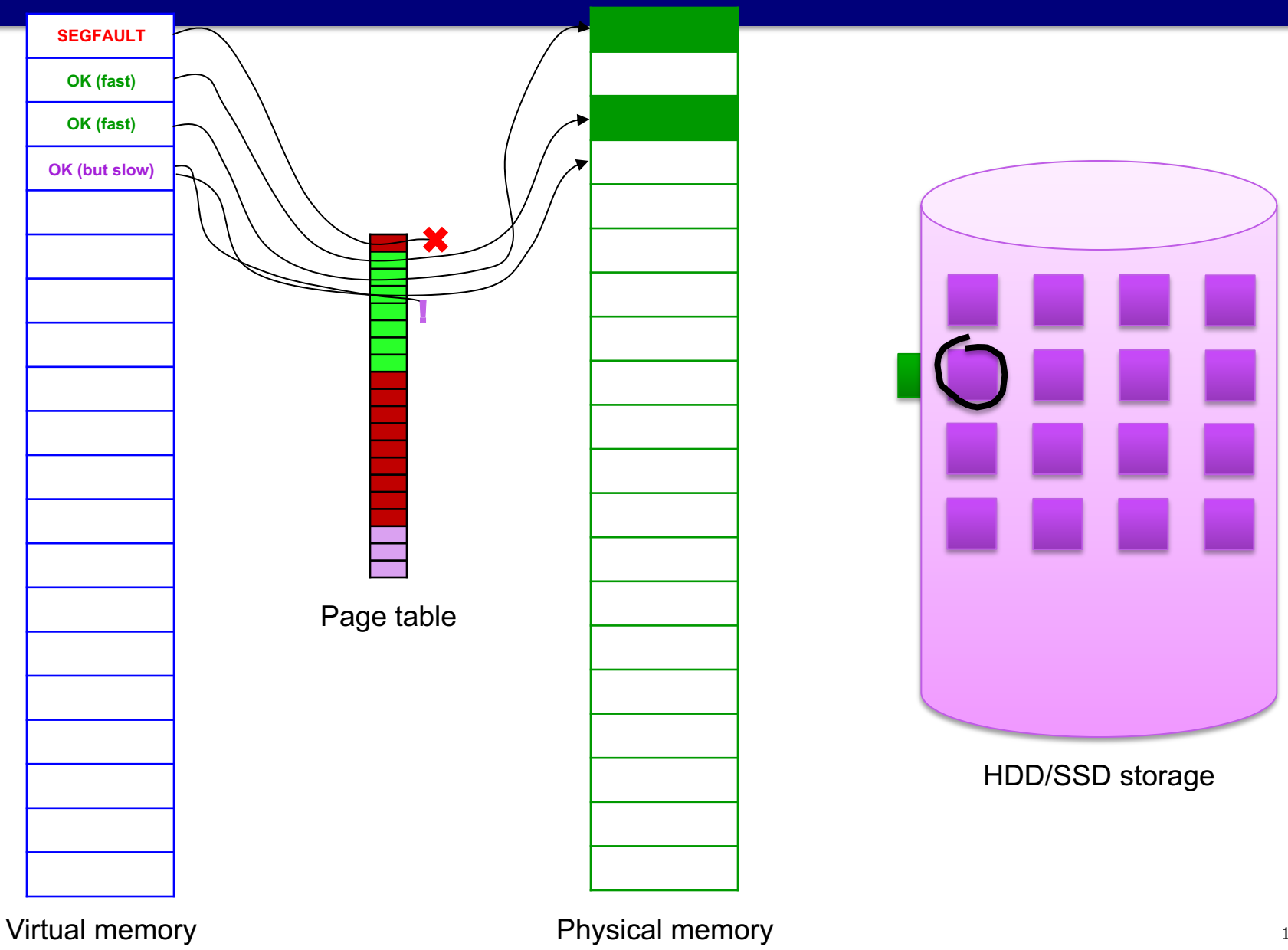
- In the example above:
  - 64KB pages? → 16-bit (= $\log_2 64K = \log_2 2^{16}$) POFS
  - 32-bit machine? → 32-bit VA → 16-bit VPN (= 32b VA – 16b POFS)
  - Maximum 256MB memory? → 28-bit PA → 12-bit PPN (= 28b – 16b)

Mechanics of Address Translation

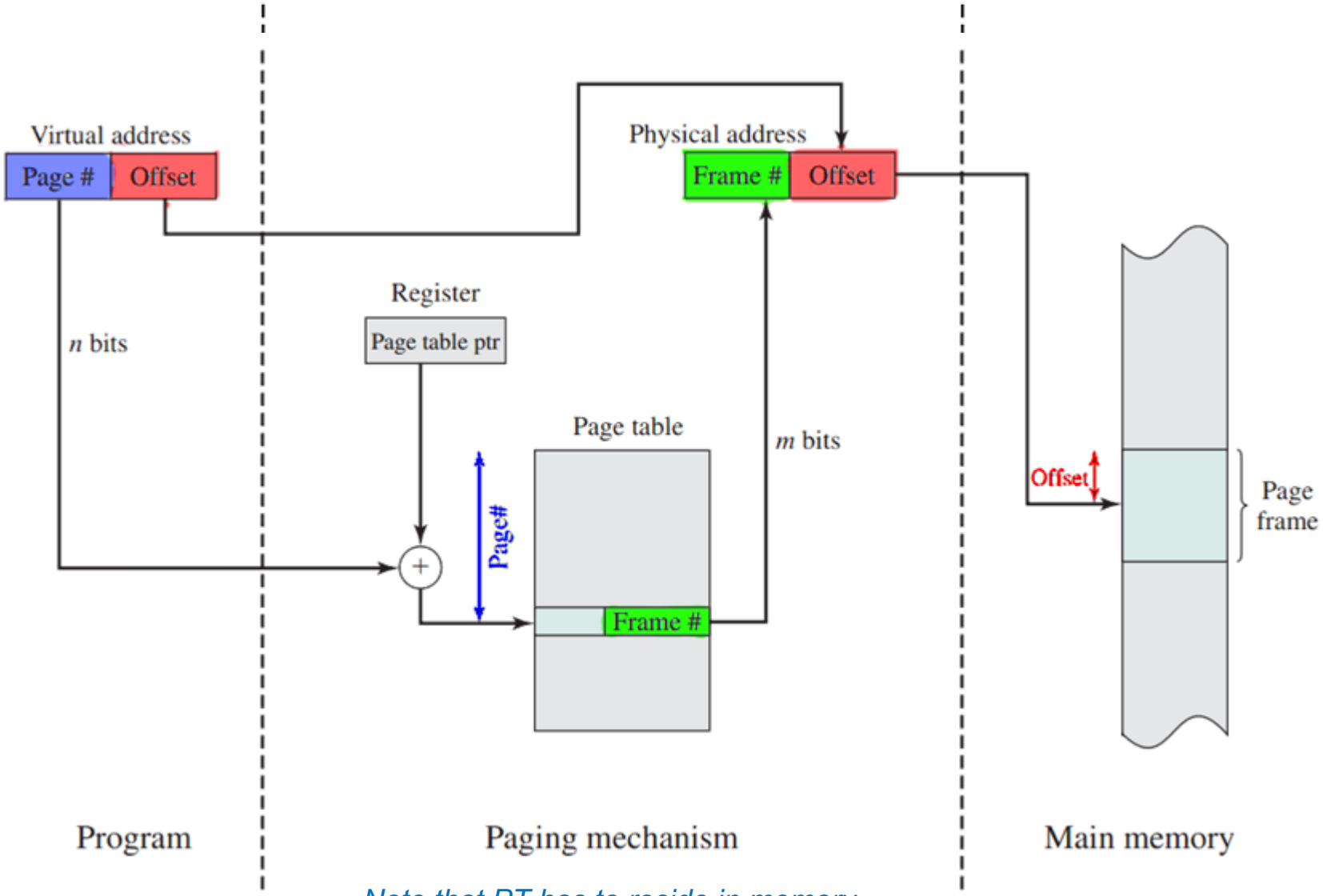
- Each process is allocated a **page table (PT)**
  - PT maps VPs to PPs or to disk addresses
    - VP entries are empty if page is never referenced
  - Translation is called **table lookup**
    - PT here is a lookup table (LUT)



High-Level Operation

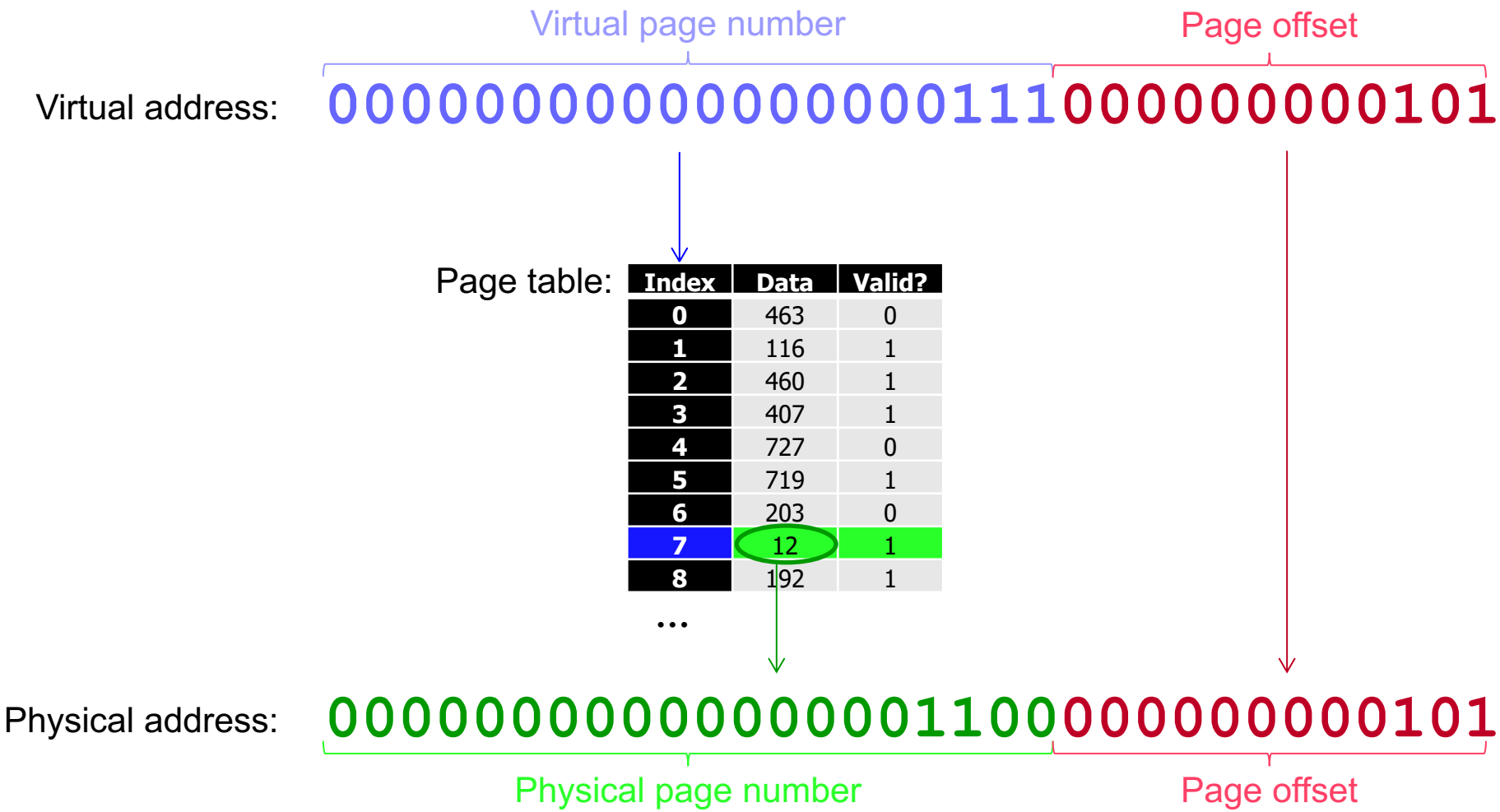


Address Translation



Note that PT has to reside in memory

Address Translation



Structure of the Page Table

Page Table Size

- How big is a page table on the following machine?
  - 4B page table entries (PTEs)
  - 32-bit machine
  - 4KB pages

- How big would the page table be with 64KB pages?
- How big would it be for a 64-bit machine with 8B PTEs?

Page Table Size

- How big is a page table on the following machine?
  - 4B page table entries (PTEs)
  - 32-bit machine
  - 4KB pages
- Solution
  - 32-bit machine → 32-bit VA → 2^{32}B virtual memory (VM) = 4GB VM
  - 4GB/VM / 4KB/page = 1M pages/VM → 1M PTEs/program's PT
  - 1M PTEs/PT * 4B/PTE → 4MB/PT (or "4MB PTs")

- How big would the page table be with 64KB pages? 256KB PTs
- How big would it be for a 64-bit machine with 8B PTEs? 35PB PTs!

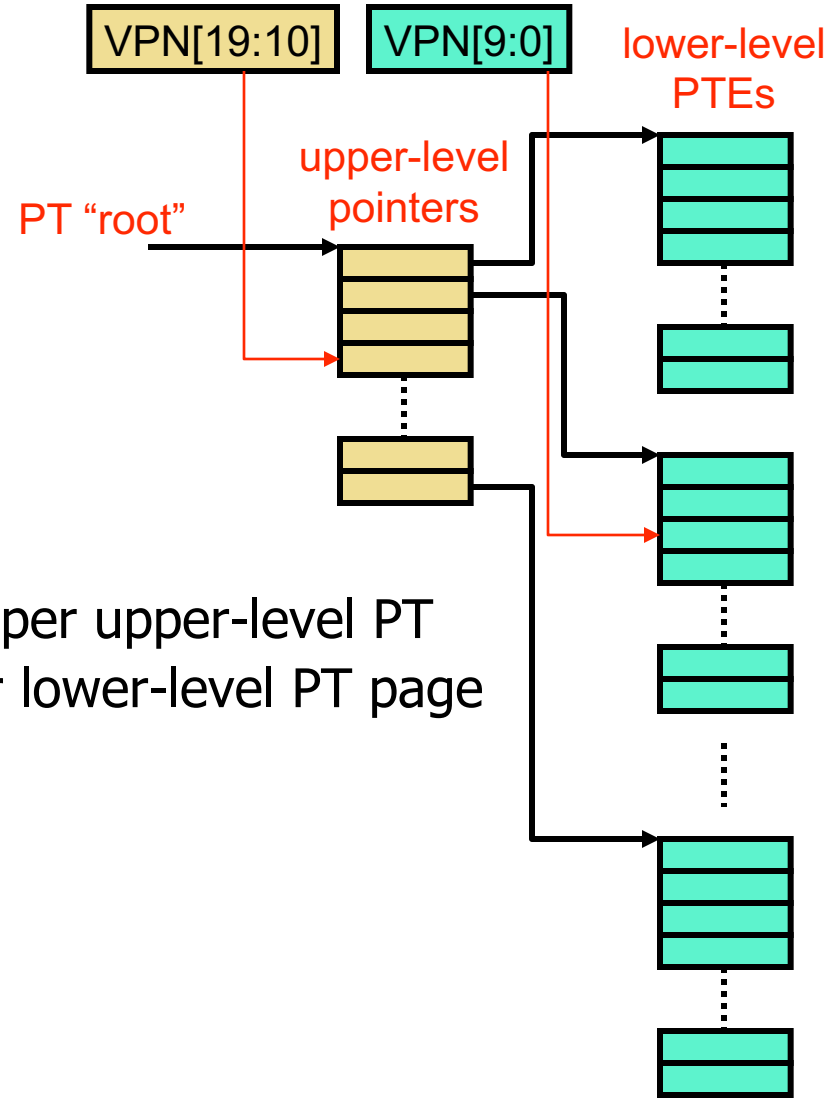
- **PTs can get enormous!**
  - **There are ways to make them smaller**

Multi-Level Page Table

- One way to make PTs smaller: **multi-level page tables**
  - Tree of page tables
  - Lowest-level tables hold PTEs
  - Upper-level tables hold pointers to pages in lower-level tables
  - Different parts of VPN used to index different levels
  - *Another (better) way is using Inverted Page Tables (see ECE 650)*

- Example: two-level PT for the 32-bit machine on previous slide
  - Compute number of pages needed to address pages in lower-level PT (the one holding PTEs)
    - 4MB/PT / 4KB/pages = 1K pages/PT
  - Compute size of upper-level PT
    - 1K lower-level PT pages → 1K pointers needed in upper-level PT
    - 1K pointers * 4B/pointer = 4KB/upper-level PT

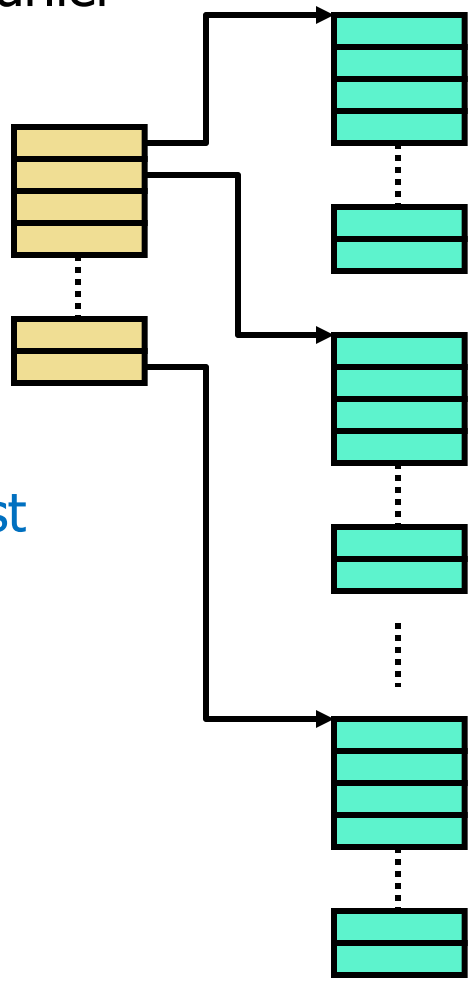
Multi-Level Page Table



- 20-bit VPN:
  - Upper 10 bits index 2^{10} pointers per upper-level PT
  - Lower 10 bits index 2^{10} PTEs per lower-level PT page

Multi-Level Page Table

- Have we saved any space?
  - Isn't total size of lower-level PT the same as earlier (i.e., 4MB)?
  - Not exactly…
- Lower-level PT can be offloaded to disk!
  - Its pages paged to memory as needed
- Large virtual address regions unused
  - Corresponding lower-level pages need not exist
  - Corresponding upper-level pointers are null



Address Translation Mechanics

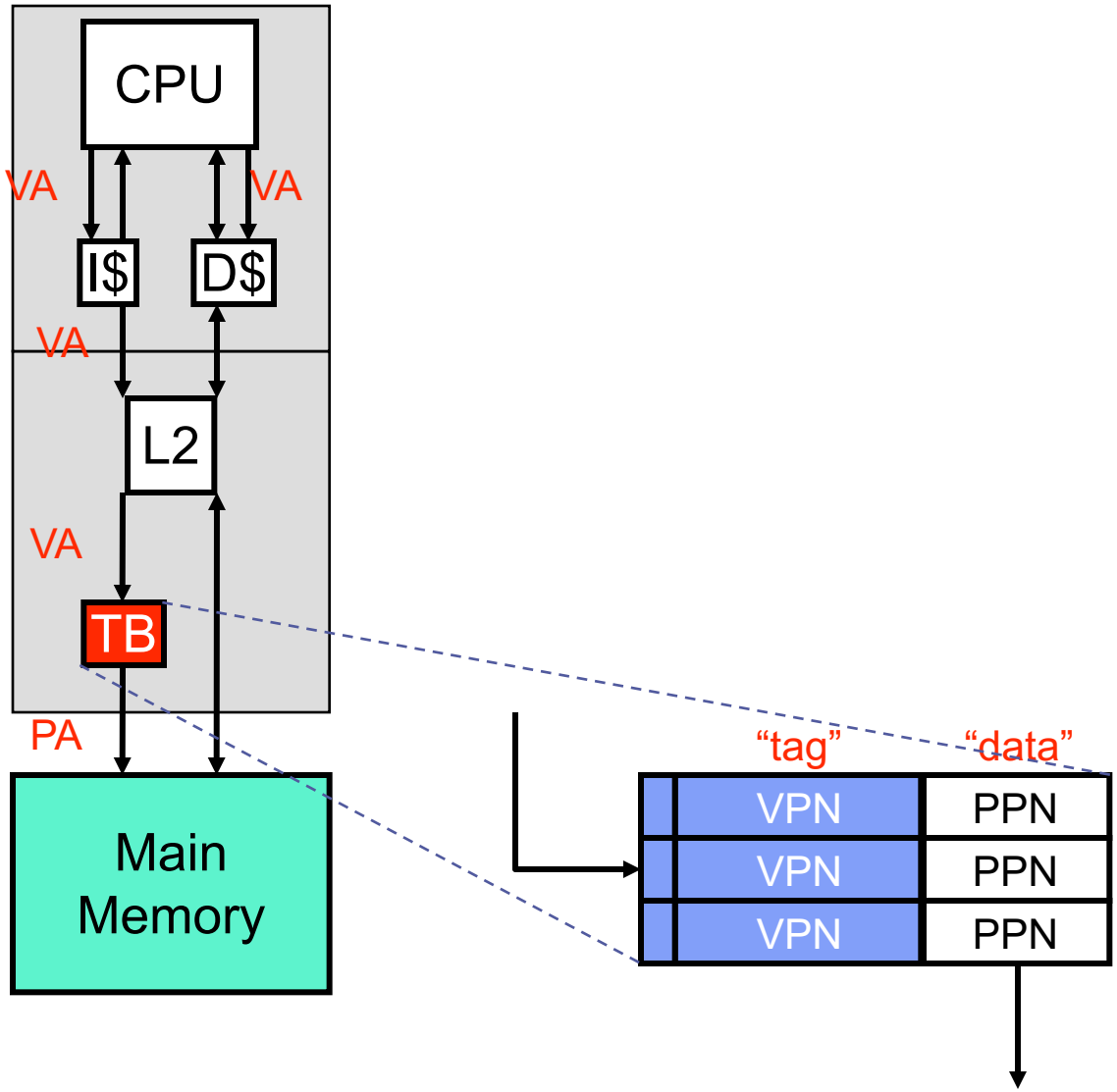
- The six questions:
  - What? Address translation
  - Why? Compatibility, multi-programming, protection
  - How? Page table
  - Where does page table reside? In memory (top-level PT at least)
  - **Who performs it? Operating system (OS)**
    - Cannot be handled by each process alone because we wouldn't be able to manage different processes using the same main memory
    - → We have to let something else handle it that knows about all processes → The OS
  - **When?**

Caching Address Translation

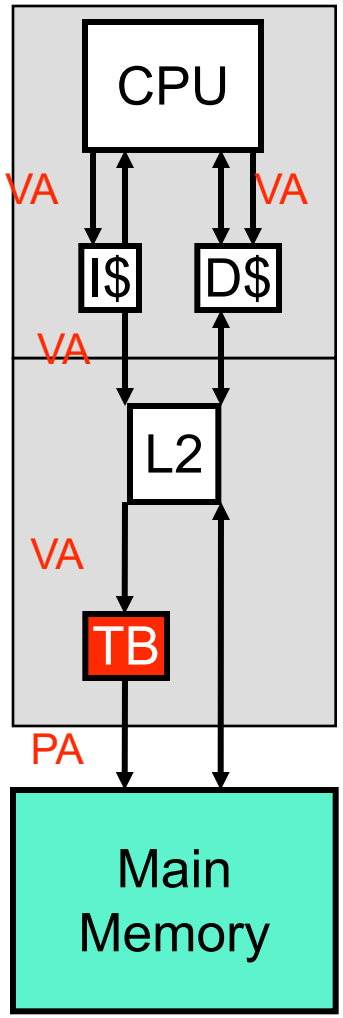
There's Still a Problem…

- OS handling address translation at every L2 miss is too slow!
- We need to accelerate the translation of VA to PA
- Answer: **translation buffer**
  - A special **cache** for page tables
  - A typical example of the way we solve performance problems in computing by adding caches
    - **Performance problem → Add cache!**

Translation Buffer

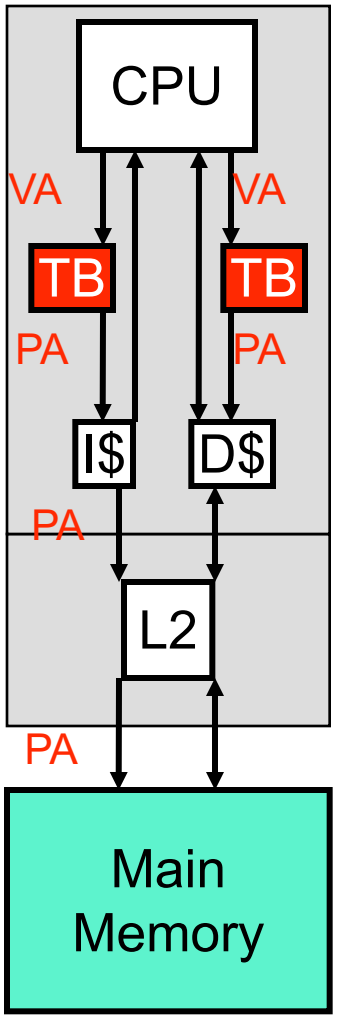


Virtual Caches



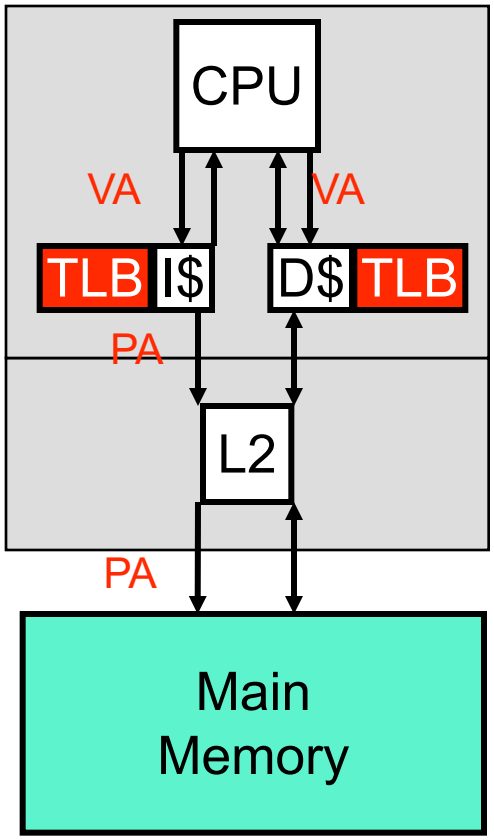
- Memory hierarchy so far: **virtual caches**
  - Indexed and tagged by VAs
  - Translate to PAs only to access memory
  - \+ Fast: avoids translation latency in common case
- What to do on **process switches**?
  - Flush caches? Slow
  - Add process IDs to cache tags?
- Does inter-process communication work?
  - **Aliasing**: multiple VAs map to same PA
    - How are multiple cache copies kept in sync?
    - Also, a problem for I/O (later in the course)
  - Disallow caching of shared memory? Slow

Physical Caches



- Alternatively: **physical caches**
  - Indexed and tagged by PAs
  - Translate to PA at the outset
  - \+ No need to flush caches on process switches
    - Processes do not share PAs
  - \+ Cached inter-process communication works
    - Single copy indexed by PA
  - – Slow: adds 1 cycle to t_{hit}

Virtual-Physical Caches



- Compromise: **virtual-physical caches**
  - Indexed by VAs
  - Tagged by PAs
  - Cache access and address translation in parallel
  - \+ No context-switching/aliasing problems
  - \+ Fast: no additional t_{hit} cycles
- → A TB that acts in parallel with a cache:
  - The **Translation Lookaside Buffer (TLB)**
  - A Common organization in processors today

Other Performance Issues

The Table of Time

Event	Picoseconds	≈	Hardware/target	Source
Average instruction time*	30	30 ps	Intel Core i7 4770k (Haswell), 3.9GHz	https://en.wikipedia.org/wiki/Instructions_per_second
Time for light to traverse CPU core (~13mm)	44	40 ps	Intel Core i7 4770k (Haswell), 3.9GHz	http://www.anandtech.com/show/7003/the-haswell-review-intel-core-i74770k-i54560k-tested/5
Clock cycle (3.9GHz)	256	300 ps	Intel Core i7 4770k (Haswell), 3.9GHz	Math
Memory read: L1 hit	1,212	1 ns	Intel i3-2120 (Sandy Bridge), 3.3 GHz	http://www.7-cpu.com/cpu/SandyBridge.html
Memory read: L2 hit	3,636	4 ns	Intel i3-2120 (Sandy Bridge), 3.3 GHz	http://www.7-cpu.com/cpu/SandyBridge.html
Memory read: L3 hit	8,439	8 ns	Intel i3-2120 (Sandy Bridge), 3.3 GHz	http://www.7-cpu.com/cpu/SandyBridge.html
Memory read: DRAM	64,485	60 ns	Intel i3-2120 (Sandy Bridge), 3.3 GHz	http://www.7-cpu.com/cpu/SandyBridge.html
Process context switch or system call	3,000,000	3 us	Intel E5-2620 (Sandy Bridge), 2GHz	http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html
Storage sequential read, 4kB (SSD)**	7,233,796	7 us	SSD: Samsung 840 500GB	http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html
Storage sequential read, 4kB (HDD)**	65,104,167	70 us	HDD: 2.5" 500GB 7200RPM	http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html
Storage random read, 4kB (SSD)	100,000,000	100 us	SSD: Samsung 840 500GB	http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html
Storage random read, 4kB (HDD)	10,000,000,000	10 ms	HDD: 2.5" 500GB 7200RPM	http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html
Internet latency, Raleigh home to NCSU (3 mi)	21,000,000,000	20 ms	courses.ncsu.edu	Ping
Internet latency, Raleigh home to Chicago ISP (639 mi)	48,000,000,000	50 ms	dls.net	Ping
Internet latency, Raleigh home to Luxembourg ISP (4182 mi)	108,000,000,000	100 ms	eurodns.com	Ping
Time for light to travel to the moon (average)	1,348,333,333,333	1 s	The moon	http://www.wolframalpha.com/input/?i=distance+to+the+moon

* Based on Dhrystone, single core only, average time per instruction
** Based on sequential throughput, average time per block

Thrashing

- If we frequently have to go to disk, the memory latency becomes like disk latency (1000x worse!)
- Need the vast majority of memory accesses to be in memory.
- Remember when our **working set** didn't fit in **cache**?
- It's like that, but now our **working set** isn't fitting into **RAM**!
- This is called **thrashing**
  - When a process is busy swapping pages between disk and memory
  - Makes CPU appear under-utilized, as most time is spent waiting on disk

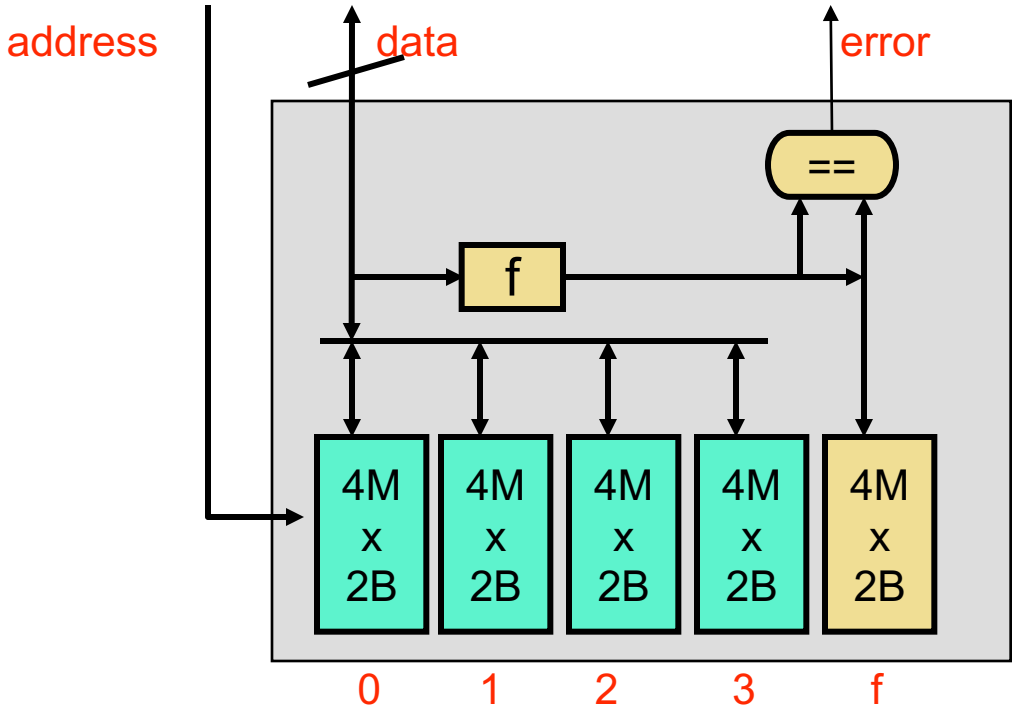
Error Correction in DRAM

Error Detection and Correction

- One last thing about DRAM technology: **errors**
  - DRAM fails at a higher rate than SRAM or CPU logic
    - Capacitor wear
    - Bit flips from energetic α-particle strikes
    - Many more bits
  - Modern DRAM systems have built-in error detection/correction

- **Solution: redundancy**
  - Main DRAM chips store data, additional chips store f(data)
    - #bits |f(data)| < #bits |data|
  - On read: re-compute f(data), compare to stored f(data)
    - Different ? Error…
  - Option I (**detect error**): kill program when error detected
  - Option II (**correct error**): enough information to fix error? fix and go on

Error Detection and Correction



- Error detection/correction schemes are distinguished by:
  - How many (simultaneous) errors they can detect
  - How many (simultaneous) errors they can correct

Error Detection Example: Parity

- **Parity**: simplest error detection scheme
  - $f(data_{N-1...0}) = XOR(data_{N-1}, ..., data_1, data_0)$
  + Can detects a single bit flip (this is usually enough)
    - Will miss two simultaneous bit flips
    - But the odds of that happening are usually very low (depends on many factors technology)
  – Cannot correct error: no way to tell which bit flipped
- Many other schemes exist for detecting/correcting errors
  - Take ECE 554 (Fault Tolerant Computing) for more info
  - Also, some of this in ECE 650