

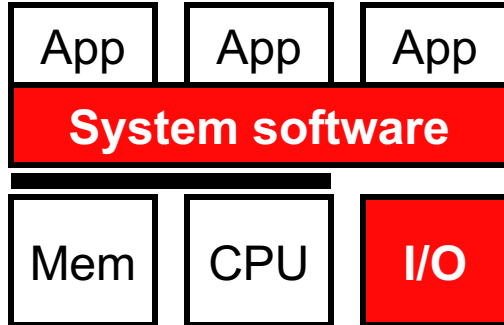
CIS 5710

Computer Organization and Design

Unit 9: Memory

Slides developed by Benedict Brown, Milo Martin, Amir Roth & C.J. Taylor
at the University of Pennsylvania
with sources that included University of Wisconsin slides
by Mark Hill, Guri Sohi, Jim Smith, and David Wood.

This Unit: Memory



- DRAM
- The operating system (OS)
 - A super-application
 - Hardware support for an OS
- Virtual memory
 - Page tables and address translation
 - TLBs and memory hierarchy issues
- Virtual Machines

Readings

- P&H
 - DRAM: Appendix C.9
 - Virtual Memory: 5.4

DRAM

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, **requires periodic refresh**

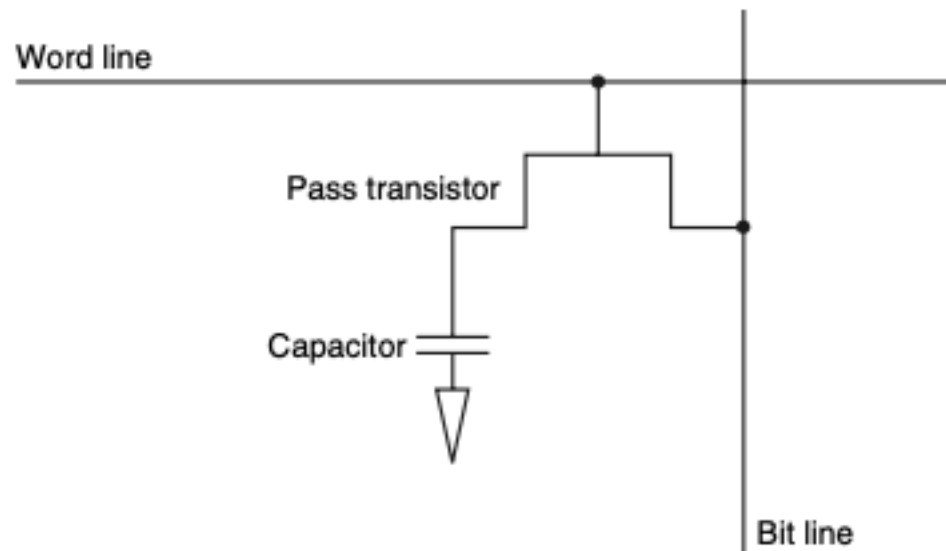
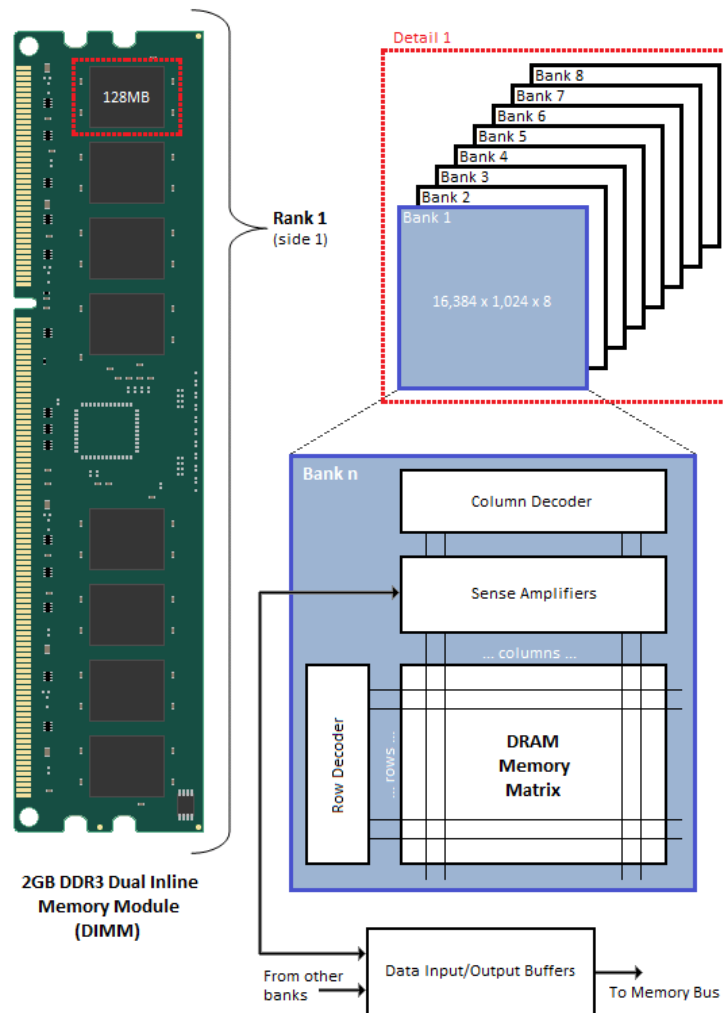


FIGURE C.9.5 A single-transistor DRAM cell contains a capacitor that stores the cell contents and a transistor used to access the cell.

c/o Patterson & Hennessy, COD, Appendix C

DRAM internal structure

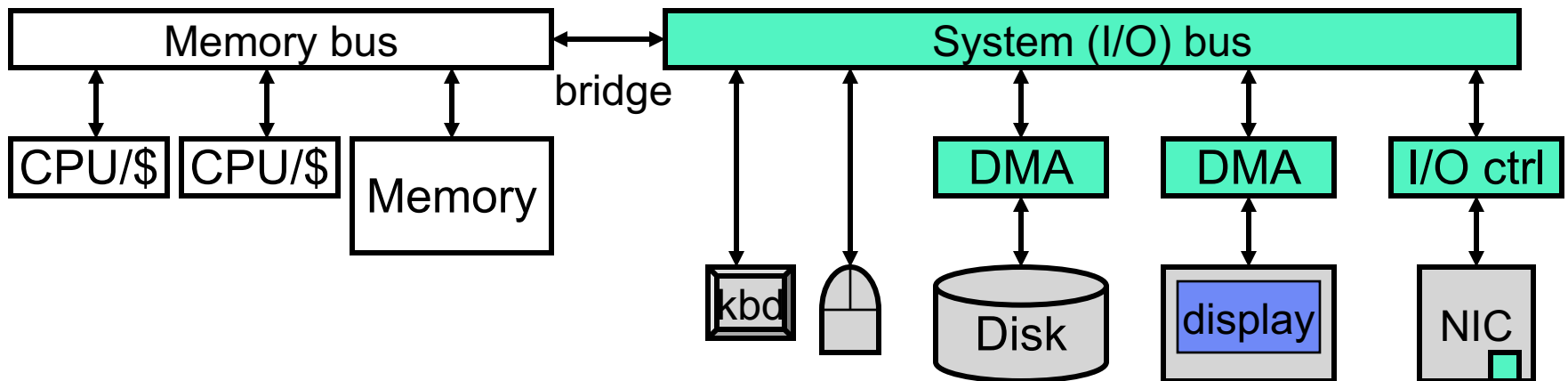
- Each **bank** composed of **cells**
- Many **banks** per **chip**
- Many **chips** per **rank**
- 1-4 **ranks** per stick/**DIMM**
- access one row at a time (~8KB) across all chips



c/o [Anandtech](#)

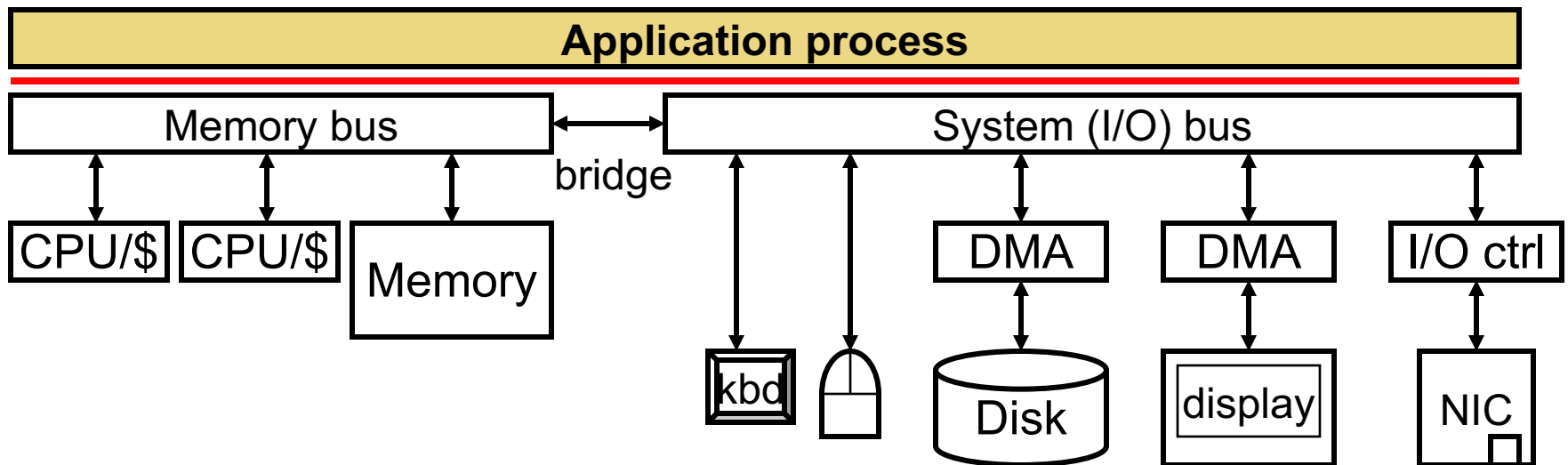
A Computer System: Hardware

- CPUs and memories
 - Connected by memory bus
- **I/O peripherals**: storage, input, display, network, ...
 - With separate or built-in DMA
 - Connected by **system bus** (which is connected to memory bus)

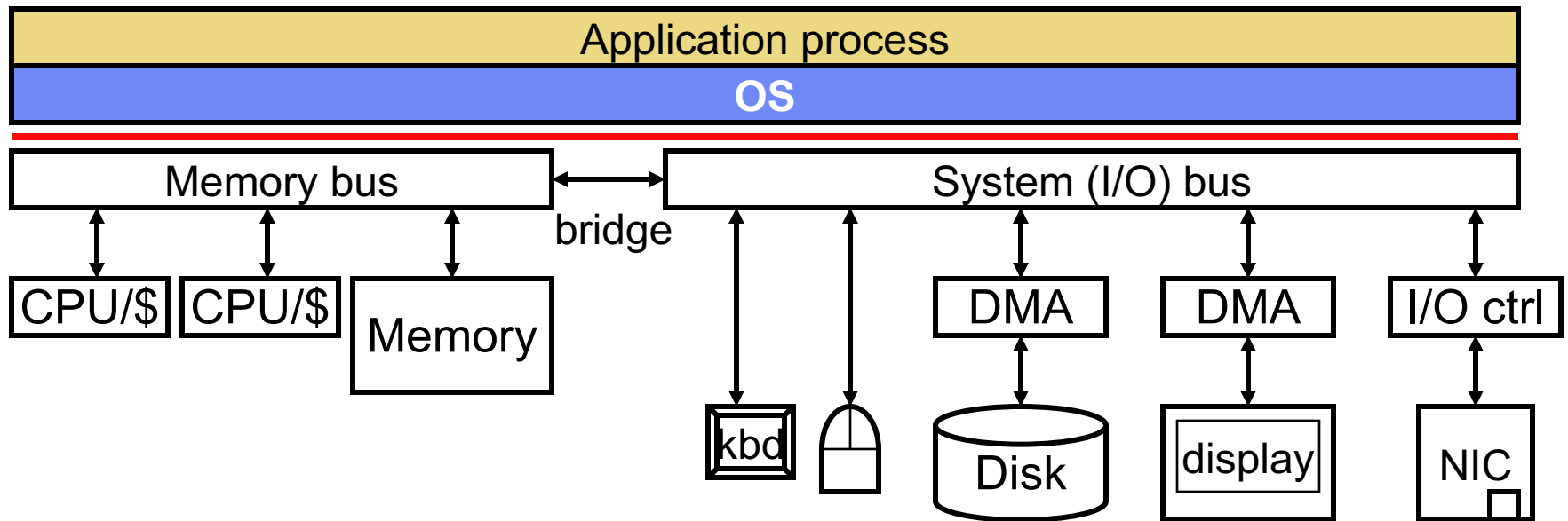


A Computer System: + App Software

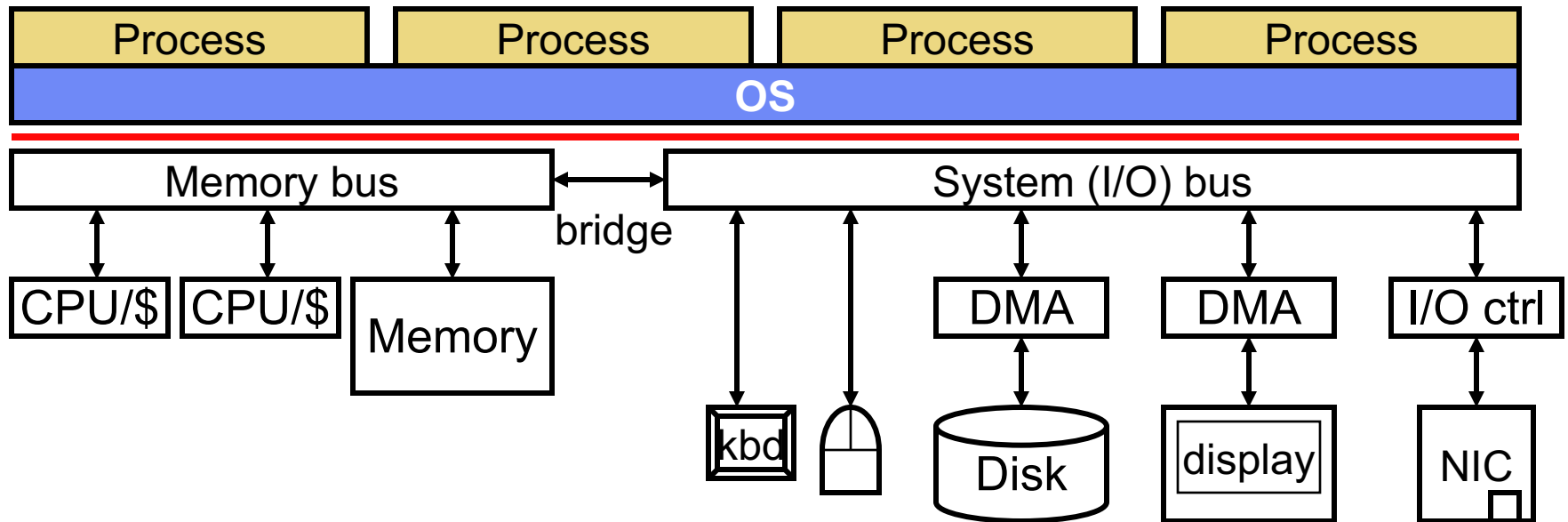
- **Application process:** the work we want to do
 - everything else is in support of this



A Computer System: + OS

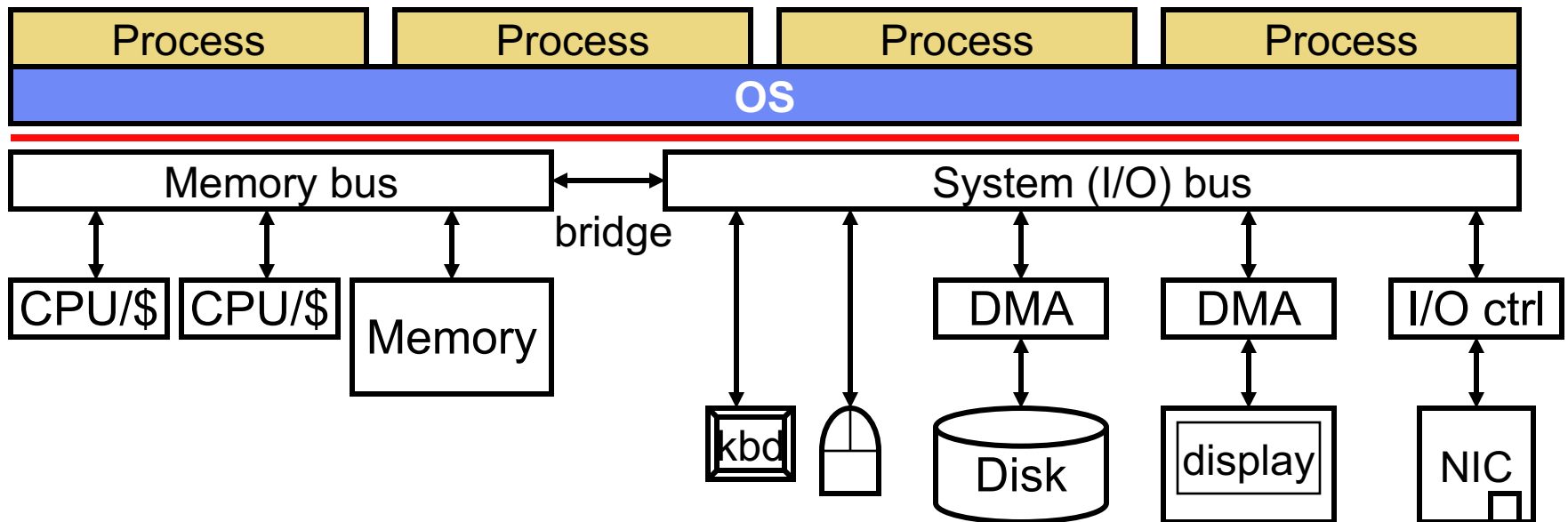


A Computer System: + OS



A Computer System: + OS

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



“Virtualizing” a resource

- 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 being 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 do we 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

Virtualizing Memory

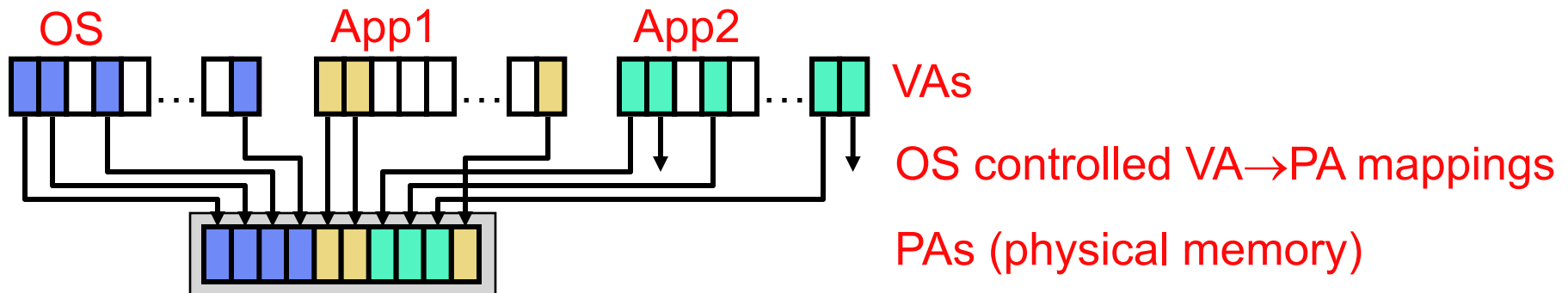
Virtualizing Main Memory

- How do multiple procs (and the OS) share main memory?
 - **Goal: each application thinks it has all of the memory**
- One process may want more memory than is in the system
 - 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”
 - No notion of “program not fitting” in registers *or* caches (why?)
- 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)

Virtual Memory (VM)

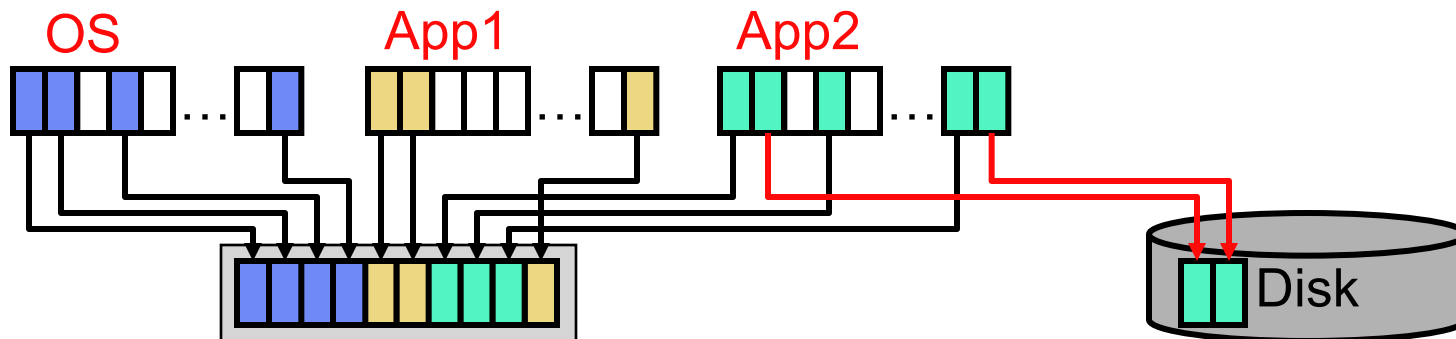
- **Virtual Memory (VM):**

- 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 coarse (page) granularity
- OS controls VA to PA mapping for itself and all other processes
- Logically: translation performed before every insn fetch, load, store
- Physically: hardware acceleration removes translation overhead



Virtual Memory (VM)

- Programs use **virtual addresses (VA)**
 - VA size (N) aka pointer size (these days, 64 bits)
- Memory uses **physical addresses (PA)**
 - PA size (M) typically $M < N$, especially if $N = 64$
 - 2^M is most physical memory machine supports
 - 48 bits in modern machines, Intel/AMD discussing 56 bits
- 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)



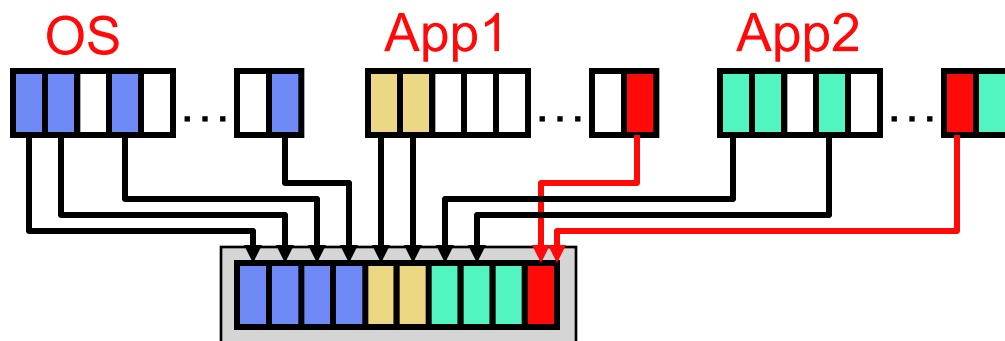
VM is an Old Idea: Older than Caches

- Original motivation: **single-program compatibility**
 - IBM System 370, in 1972
 - + Same program could run on machines with different memory sizes
 - Prior, programmers explicitly accounted for memory size
- But also: **full-associativity + software replacement**
 - Memory t_{miss} is high: extremely important to reduce $\%_{\text{miss}}$

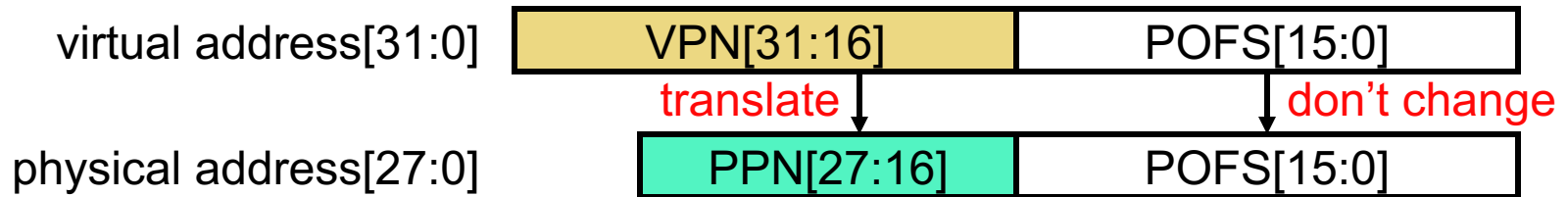
| Parameter | I\$/D\$ | L2 | Main Memory |
|-------------------|----------|-----------|---------------------|
| t_{hit} | 2ns | 10ns | 30ns |
| t_{miss} | 10ns | 30ns | 10ms (10M ns) |
| Capacity | 8–64KB | 128KB–2MB | 64MB–64GB |
| Block size | 16–32B | 32–256B | 4+KB |
| Assoc./Repl. | 1–4, LRU | 4–16, LRU | Full, “working set” |

Uses of Virtual Memory

- More recently: **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
 - 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



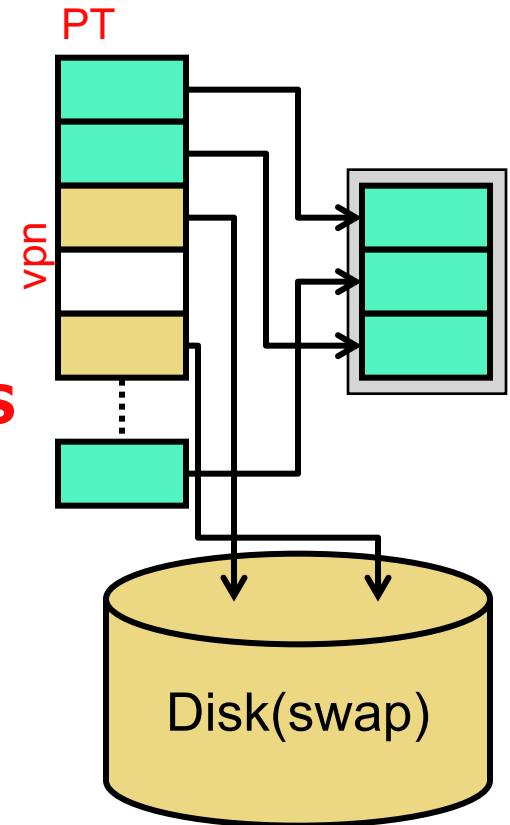
Address Translation



- 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
 - $VA \rightarrow PA = [VPN, POFS] \rightarrow [PPN, POFS]$
- 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

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

Example: Memory access at address 0xFFA8AFBA

Address of Page Table Root

0xFFFF87F8

Virtual Page Number

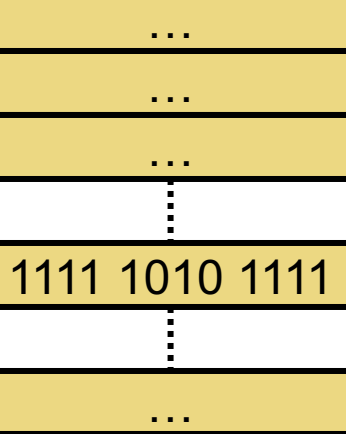
Page Offset

1111 1111 1010 1000

1010 1111 1011 1010

0

11111111 10101000



1111111111111111

Physical Address:

1111 1010 1111

1010 1111 1011 1010

Physical Page Number

Page Offset

Page Table Size

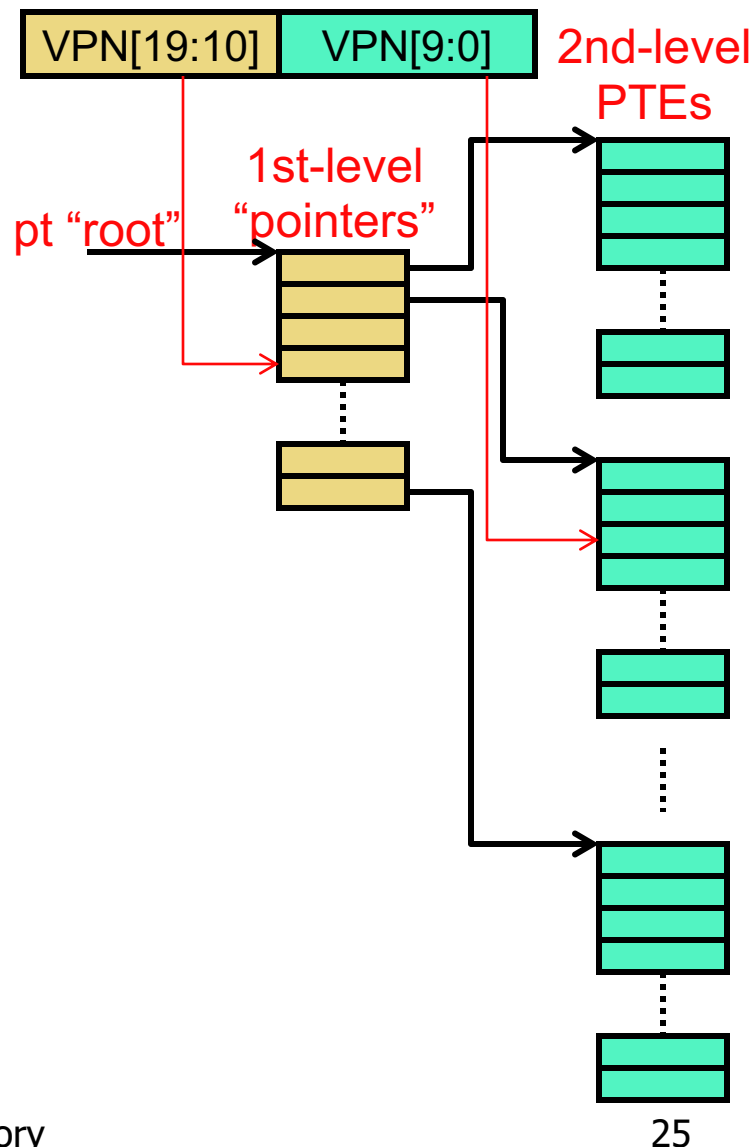
- How big is a page table on the following machine?
 - 32-bit machine
 - 4B page table entries (PTEs)
 - 4KB pages



- 32-bit machine \rightarrow 32-bit VA $\rightarrow 2^{32} = 4\text{GB}$ virtual memory
 - 4GB virtual memory / 4KB page size $\rightarrow 1\text{M}$ VPs
 - $1\text{M VPs} * 4 \text{ Bytes per PTE} \rightarrow 4\text{MB}$
- How big would the page table be with 64KB pages?
- How big would it be for a 64-bit machine?
- Page tables can get big
 - There are ways of making them smaller

Multi-Level Page Table (PT)

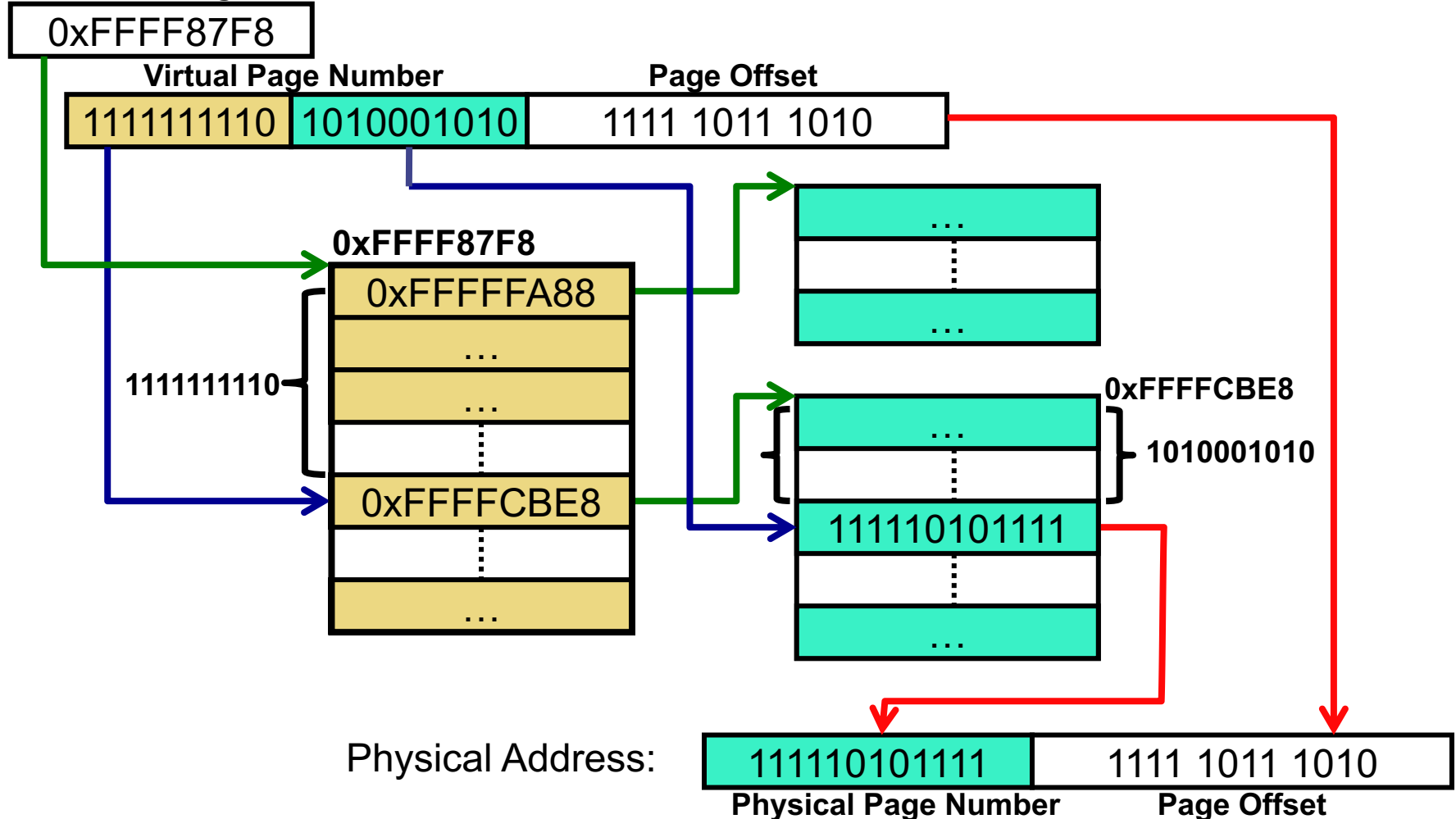
- One way:
multi-level page tables
 - Tree of page tables ("trie")
 - Lowest-level tables hold PTEs
 - Upper-level tables hold pointers to lower-level tables
 - Different parts of VPN used to index different levels
- 20-bit VPN
 - Upper 10 bits index 1st-level table
 - Lower 10 bits index 2nd-level table
 - In reality, often more than 2 levels



Multi-Level Address Translation

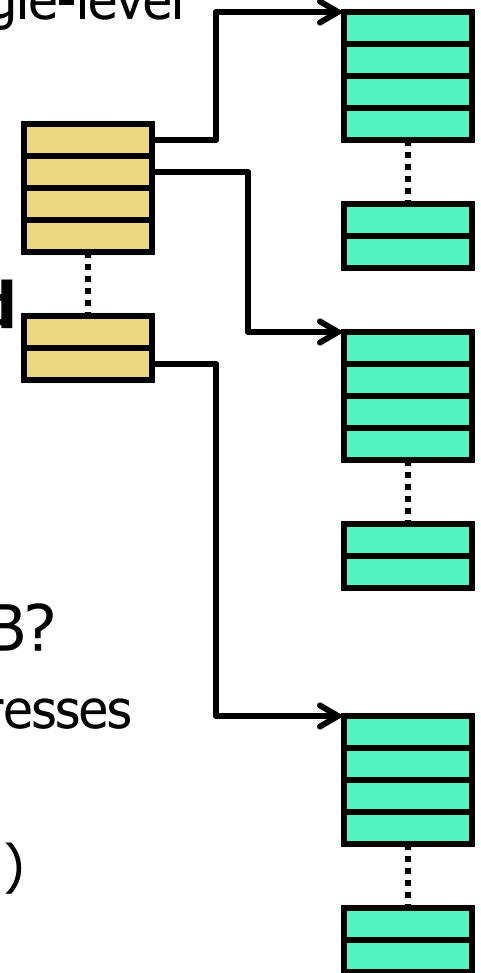
Example: Memory access at address 0xFFA8AFBA

Address of Page Table Root



Multi-Level Page Table (PT)

- 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 are **unused**
 - Corresponding 2nd-level tables unallocated
 - Corresponding 1st-level pointers are *null*
- How large for contiguous layout of 256MB?
 - Each 2nd-level table maps 4MB of virtual addresses
 - One 1st-level + 64 2nd-level pages
 - 65 total pages = 260KB (much less than 4MB!)



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

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

ARMv8-A: page table information

| VA Bits <47:39> | VA Bits <38:30> | VA Bits <29:21> | VA Bits <20:12> | VA Bits <11:0> |
|---------------------|---------------------|---------------------|----------------------------|---------------------|
| Level 1 table index | Level 2 table index | Level 3 table index | Level 4 table (page) index | Page offset address |

- 4-level lookup, 4KB translation granule, 48-bit address
 - 9 address bits per level

| VA Bits <41:29> | VA Bits <28:16> | VA Bits <15:0> |
|---------------------|----------------------------|---------------------|
| Level 1 table index | Level 2 table (page) index | Page offset address |

- 2-level lookup, 64KB page/page table size, 42-bit address
 - 13 address bits per level
 - 3 levels for 48 bits of VA – top level table is a partial table

| | | | | | | |
|------------------|--------|--------|-------------|---|-----|-------------------------------|
| 6 3 | 5 2 | 4 8 | 1 2 | 2 | 1 | 0 |
| Upper attributes | | SBZ | Address out | | SBZ | Lower attributes and validity |

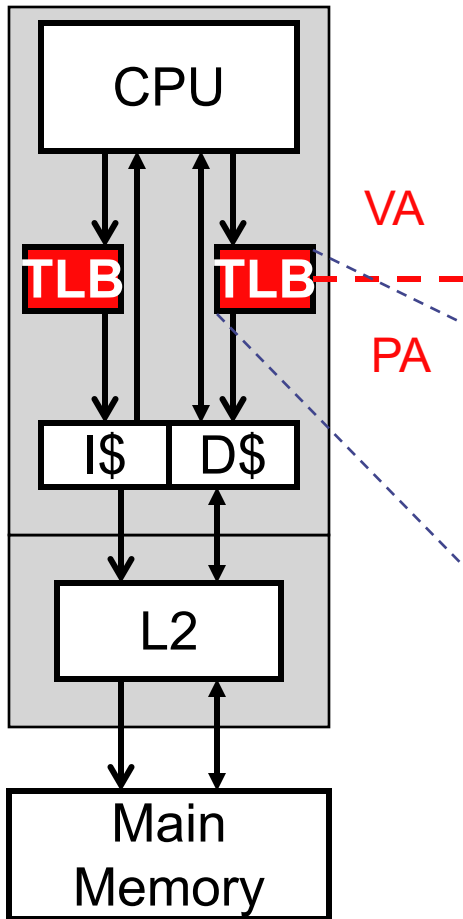
- 64-bit Translation table entry format

ARMv8 Technology Preview By Richard Grisenthwaite Lead Architect and Fellow. ARM

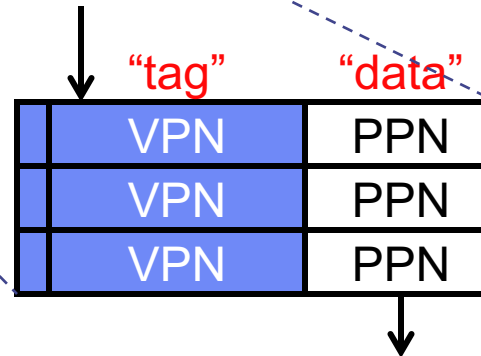
Address Translation Mechanics II

- 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)

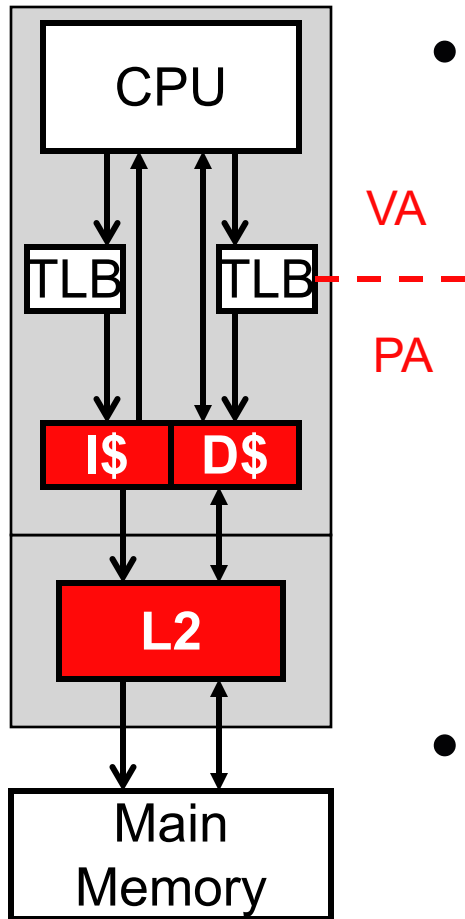
Translation Lookaside Buffer



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



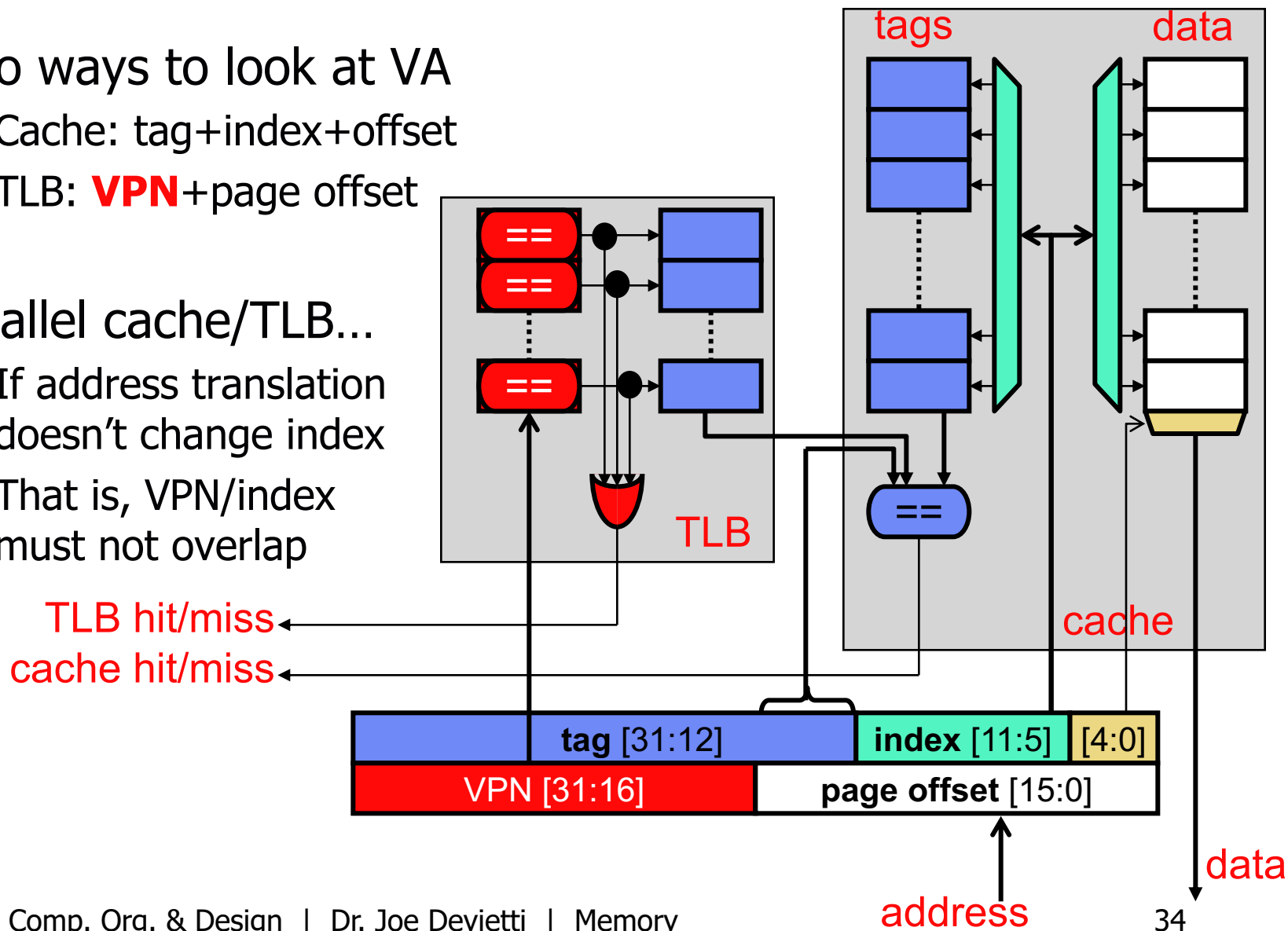
Serial TLB & Cache Access



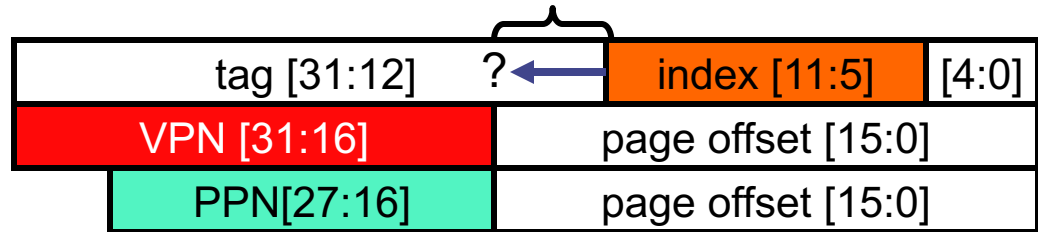
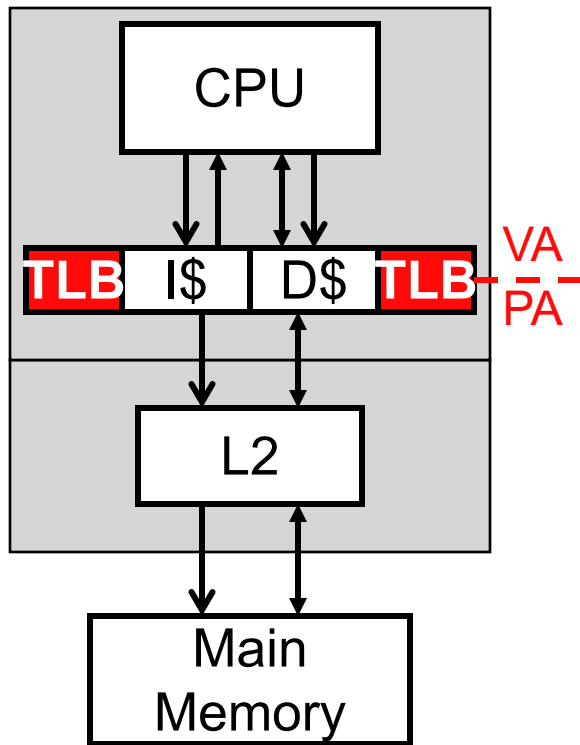
- **“Physical” caches**
 - Indexed and tagged by **physical addresses**
 - + Natural, “lazy” sharing of caches between apps/OS
 - VM ensures isolation (via **physical addresses**)
 - No need to do anything on context switches
 - Multi-threading works too
 - + Cached inter-process communication works
 - Single copy indexed by physical address
 - Slow: adds at least one cycle to t_{hit}
- Note: **TLBs are by definition “virtual”**
 - Indexed and tagged by **virtual addresses**
 - Flush across context switches
 - Or extend with process identifier tags (x86)

Parallel TLB & Cache Access

- Two ways to look at VA
 - Cache: tag+index+offset
 - TLB: **VPN**+page offset
- Parallel cache/TLB...
 - If address translation doesn't change index
 - That is, VPN/index must not overlap



Parallel TLB & Cache Access



- What about parallel access?
 - Only if...
 - $(\text{cache size}) / (\text{associativity}) \leq \text{page size}$**
 - Index bits same in VA and PA!
- Access TLB in parallel with cache
 - Cache access needs tag only at very end
 - + Fast: no additional t_{hit} cycles
 - + No context-switching/aliasing problems
 - Dominant organization used today
- Index bits constrain capacity
 - but associativity allows bigger caches

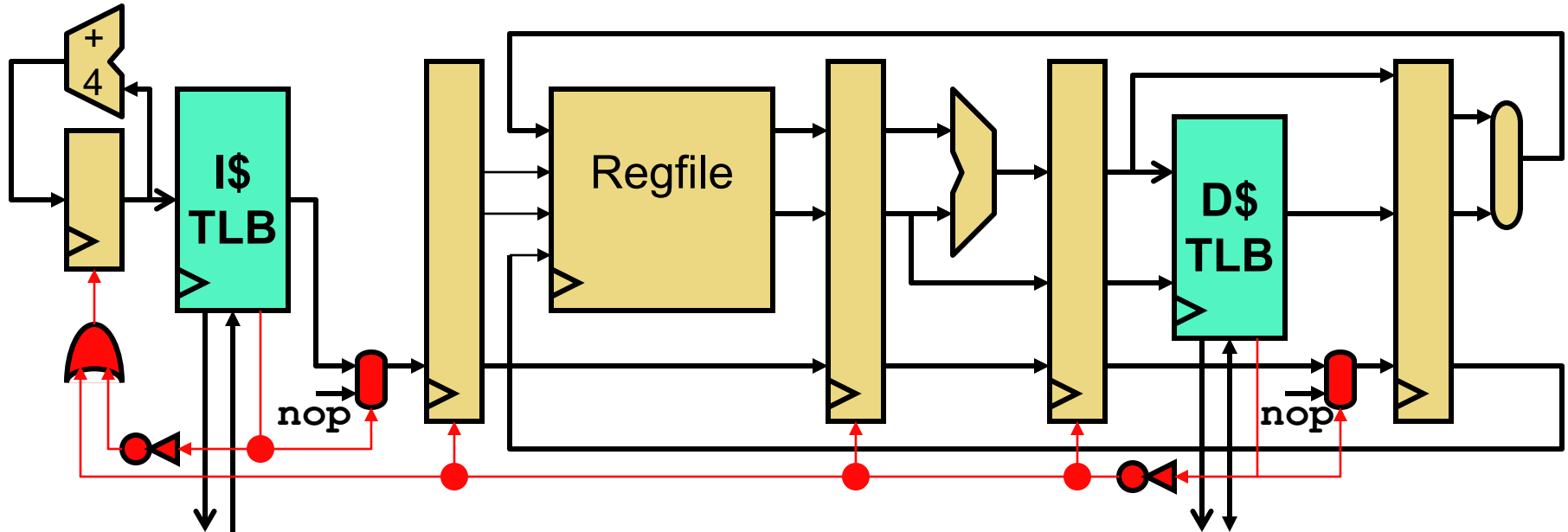
TLB Organization

- **Like caches:** TLBs also have ABCs
 - Capacity
 - Associativity (At least 4-way associative, fully-associative 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 fully-assoc. TLBs, 512-entry 4-way L2 TLB (insn & data)
 - 4KB pages, 48-bit virtual addresses, four-level page table
- **Rule of thumb:** TLB should “cover” size of on-chip caches
 - In other words: $(\text{\#PTEs in TLB}) * \text{page size} \geq \text{cache size}$
 - 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:** e.g., x86, recent SPARC, ARM
 - 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:** e.g., Alpha, MIPS
 - Short (~10 insn) OS routine walks page table, updates TLB
 - + Keeps page table format flexible
 - Latency: one or two memory accesses + OS call (pipeline flush)
- hardware TLB miss handler is popular today

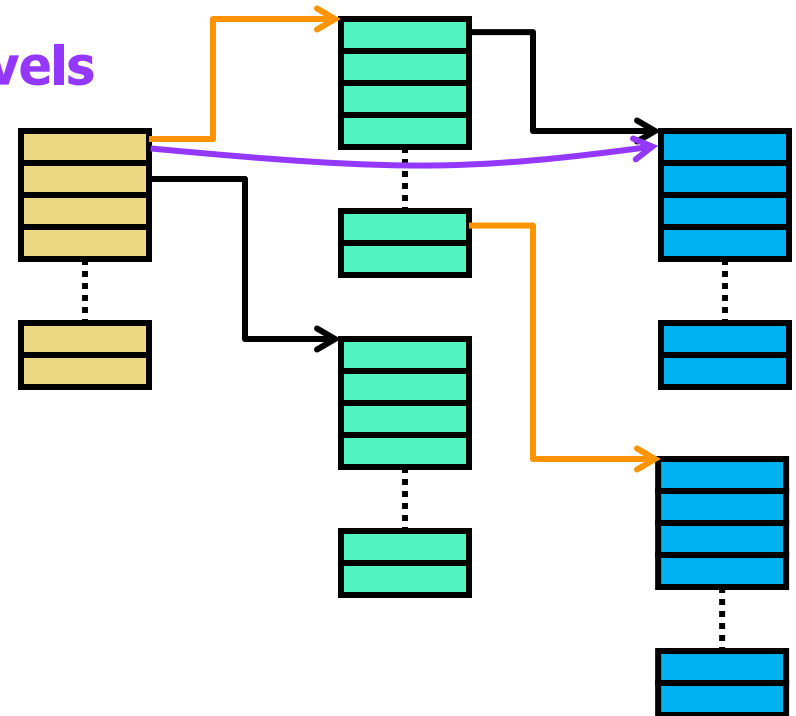
TLB Misses and Pipeline Stalls



- 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

TLB Misses on x86

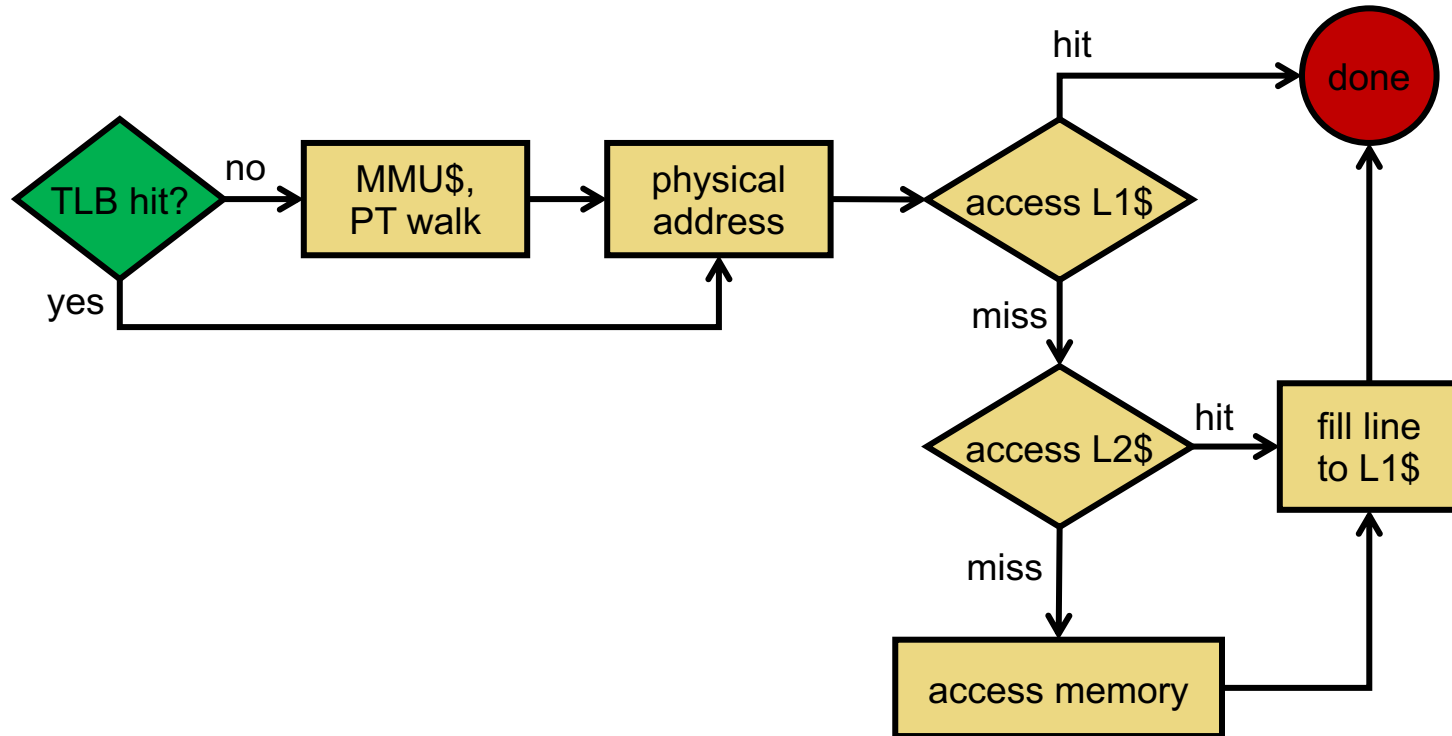
- After a TLB miss, hardware page table (PT) walker starts
 - root of PT for current process in register CR3
 - hardware walks trie
- **MMU cache** caches **parts** of the page table traversal
 - AMD: \$ entry holds **one level**
 - Intel: \$ entry holds **multiple levels**
 - MMU\$ miss means accessing the page table in memory



Page Faults

- **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 software routine:**
 - Choose a physical 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?
 - Requires yet another data structure: **frame map**
 - Maps physical pages to <process,virtual page> pairs
 - Update page tables, flush TLBs, retry memory access

The life of a virtual memory access

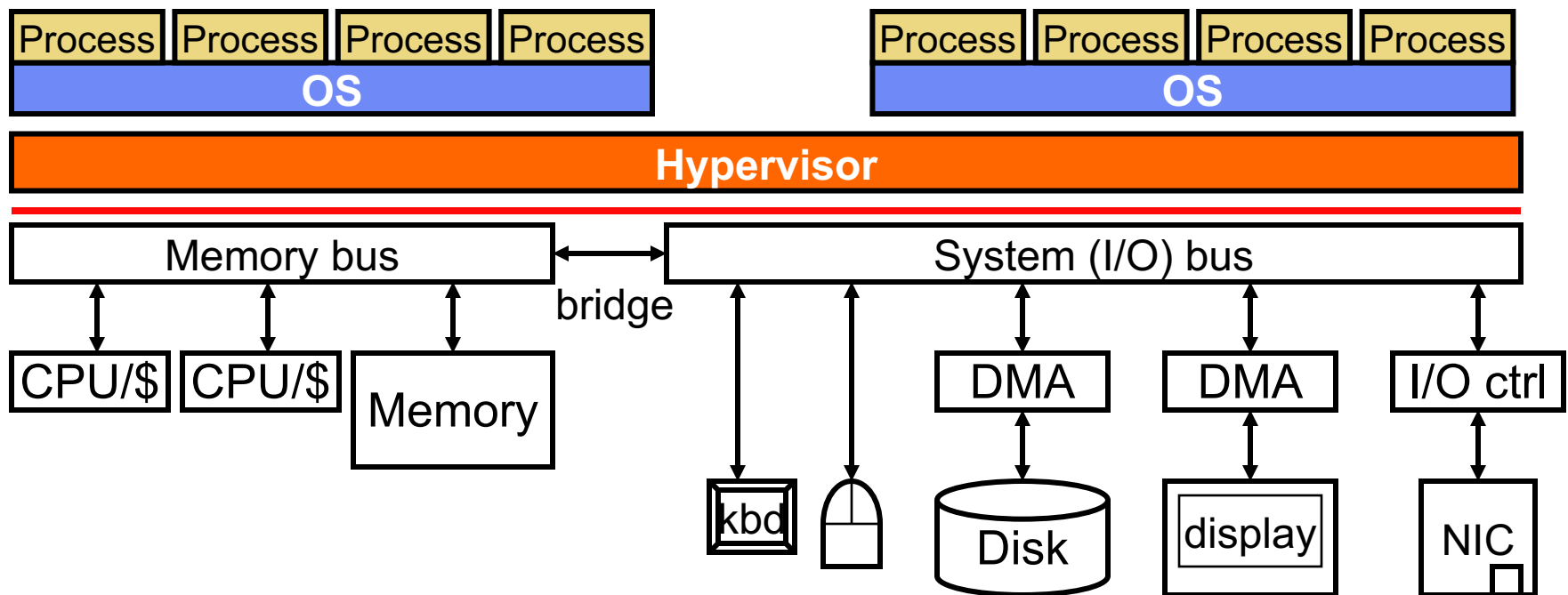


What happens on a Context Switch

- Operating system responsible for handling context switching
 - Hardware support is just a timer interrupt
- Each process has an associated data structure which is used to record relevant state such as:
 - **Architected state**
 - PC, registers, Page table pointer
 - Saved to, restored from physical memory
 - Memory state? Use virtual memory
 - **Non-architected state**: caches, predictor tables, etc.
 - **Ignore** or flush
 - ignoring is key aspect of Spectre/Meltdown vulnerabilities!
- OS swaps out values for old process, swaps in values for new process, and lets new process run

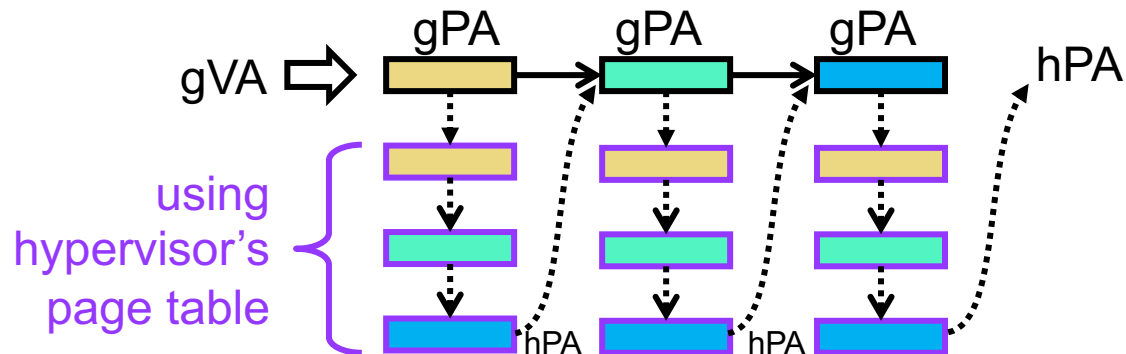
Virtual Machines

- Share physical machine among many “virtual machines”
 - each has an OS which thinks it totally controls the hardware
 - control actually resides w/ **hypervisor** or **VMM (Virtual Machine Manager)**



Virtual Machines

- Need to “context switch” between OSes
 - need to virtualize virtual memory using **nested page tables**
 - add a level of indirection
 - from VA, PA to **guest VA, guest PA, host PA**
 - an n-level PT traversal is now n^2
 - a big reason why VMs can be slow



Summary

- OS virtualizes memory and I/O devices
- Virtual memory
 - “infinite” memory, isolation, protection, inter-process communication
 - Page tables
 - Translation buffers
 - Parallel vs serial access, interaction with caching
 - Page faults