CS 110
Computer Architecture

Virtual Memory

Instructor:
Sören Schwertfeger

http://shtech.org/courses/ca/

School of Information Science and Technology SIST

ShanghaiTech University

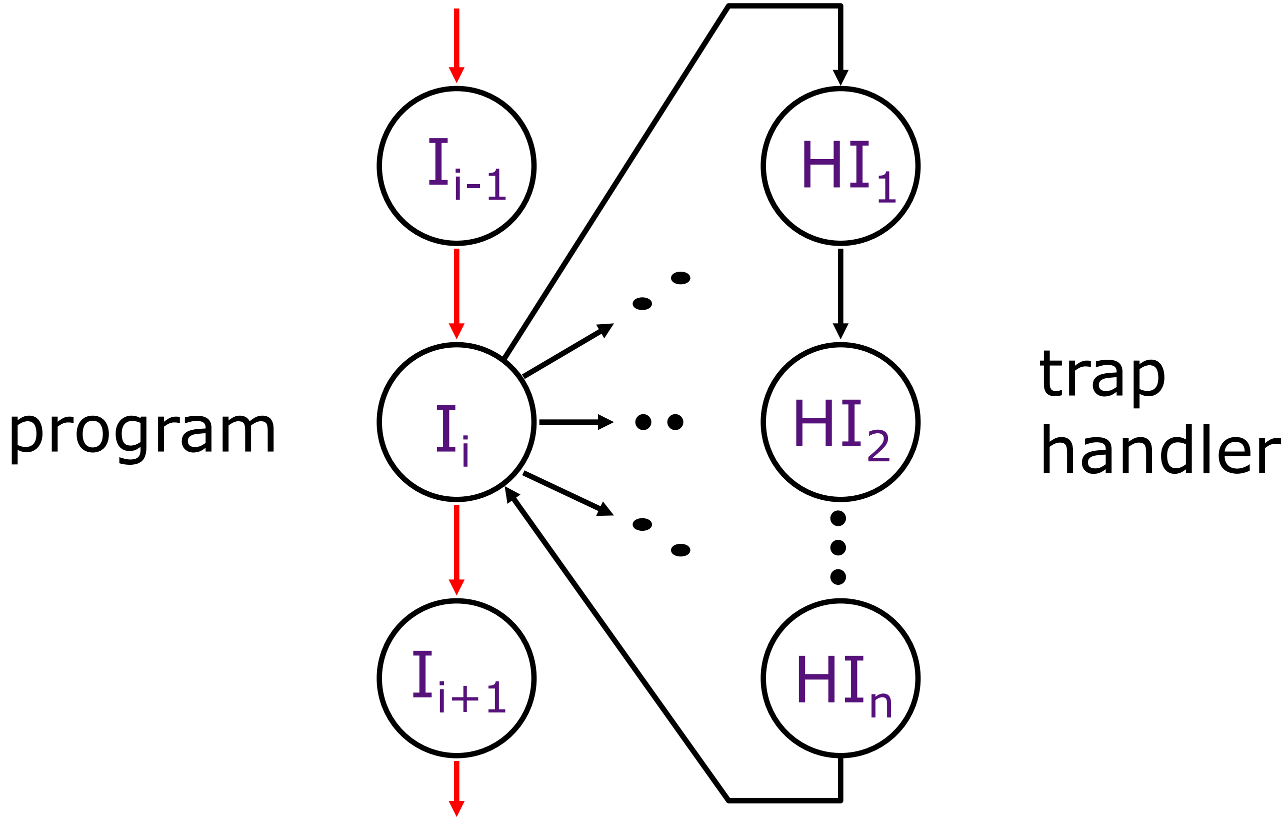
Slides based on UC Berkley's CS61C

Review

- Programmed I/O
- Polling vs. Interrupts
- Booting a Computer
  – BIOS, Bootloader, OS Boot, Init
- Supervisor Mode, Syscalls
- Base and Bounds
  – Simple, but doesn't give us everything we want
- Intro to VM

Traps/Interrupts/Exceptions:

altering the normal flow of control



An *external or internal event* that needs to be processed - by another program – the OS. The event is often unexpected from original program's point of view.

Terminology

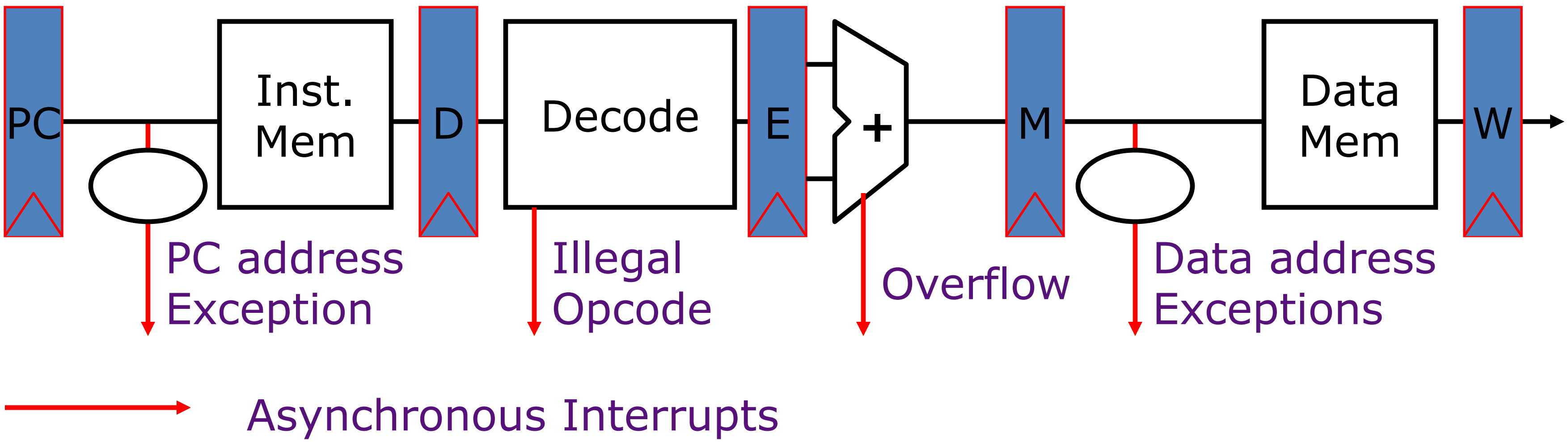
In CA (you'll see other definitions in use elsewhere):

- Interrupt – caused by an event *external* to current running program (e.g. key press, mouse activity)
  - Asynchronous to current program, can handle interrupt on any convenient instruction
- Exception – caused by some event during execution of one instruction of current running program (e.g., page fault, bus error, illegal instruction)
  - Synchronous, must handle exception on instruction that causes exception
- Trap – action of servicing interrupt or exception by hardware jump to "trap handler" code

Precise Traps

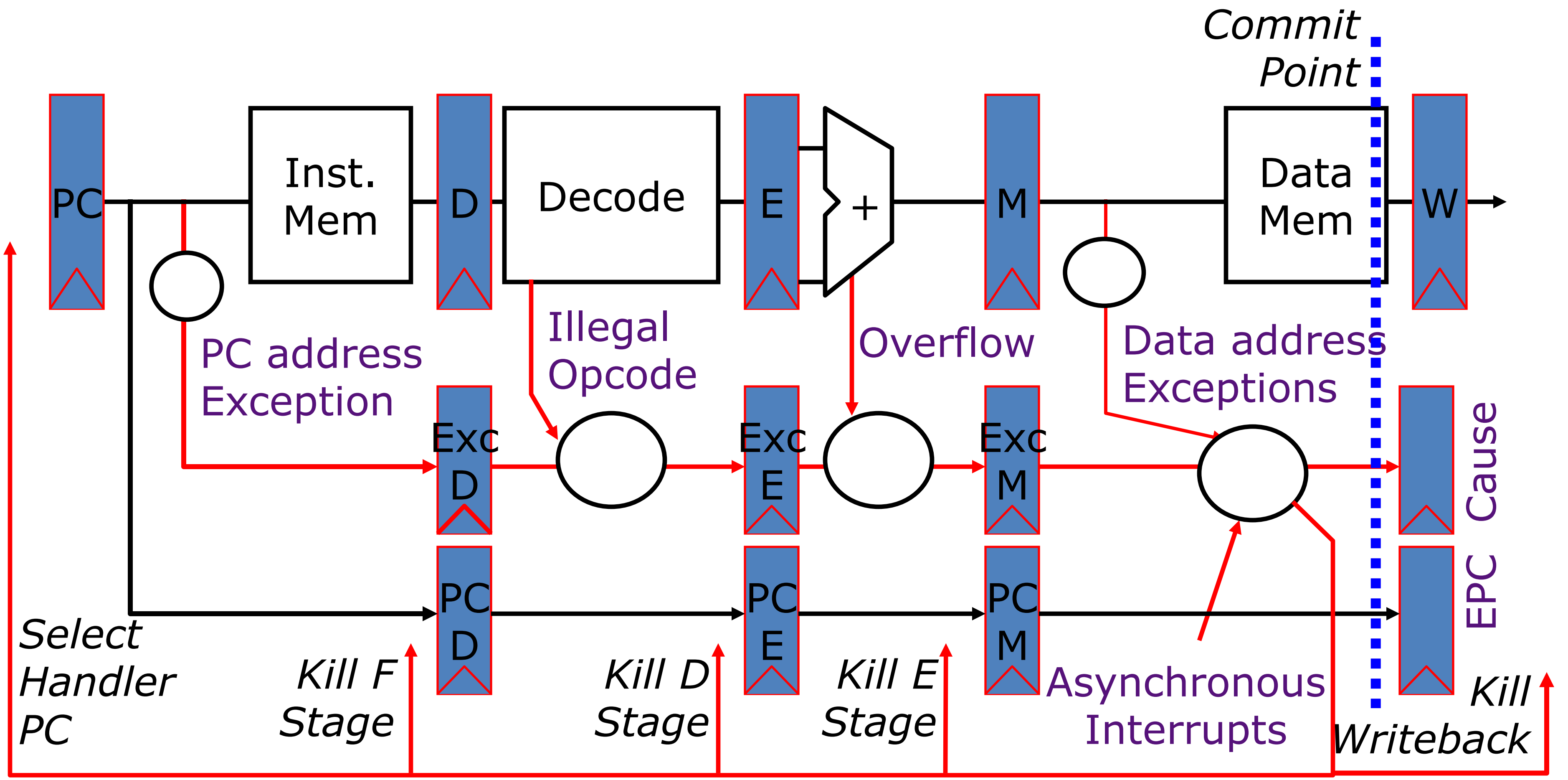
- *Trap handler's view of machine state is that every instruction prior to the trapped one has completed, and no instruction after the trap has executed.*
- Implies that handler can return from an interrupt by restoring user registers and jumping back to interrupted instruction (EPC register will hold the instruction address)
  - Interrupt handler software doesn't need to understand the pipeline of the machine, or what program was doing!
  - More complex to handle trap caused by an exception than interrupt
- Providing precise traps is tricky in a pipelined superscalar out-of-order processor!
  - But handling imprecise interrupts in software is even worse.

Trap Handling in 5-Stage Pipeline



- How to handle multiple simultaneous exceptions in different pipeline stages?
- How and where to handle external asynchronous interrupts?

Save Exceptions Until Commit



Handling Traps in In-Order Pipeline

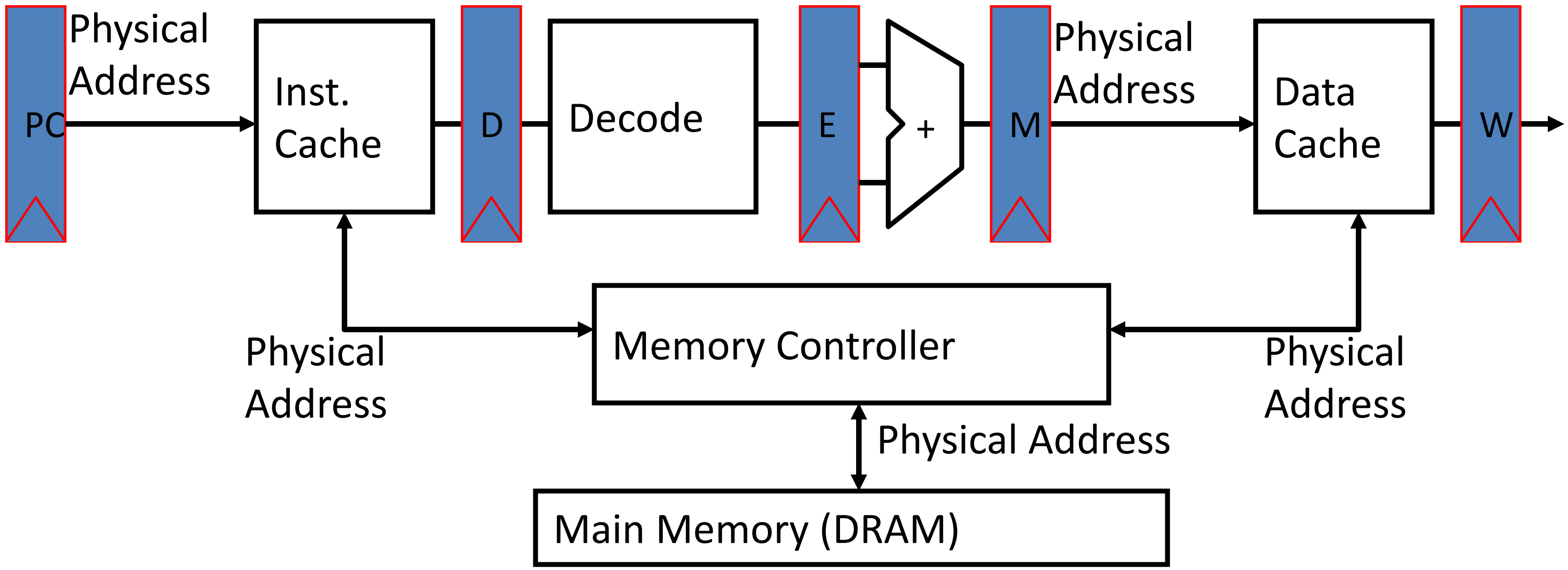
- Hold exception flags in pipeline until commit point (M stage)
- Exceptions in earlier instructions override exceptions in later instructions
- Exceptions in earlier pipe stages override later exceptions *for a given instruction*
- Inject external interrupts at commit point (override others)
- If exception/interrupt at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage

Trap Pipeline Diagram

	time								
	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) 096: ADD	IF_1	ID_1	EX_1	MA_1	-		*overflow!*		
(I_2) 100: XOR		IF_2	ID_2	EX_2	-	-			
(I_3) 104: SUB			IF_3	ID_3	-	-	-		
(I_4) 108: ADD				IF_4	-	-	-	-	
(I_5) Trap Handler code					IF_5	ID_5	EX_5	MA_5	WB_5

Virtual Memory

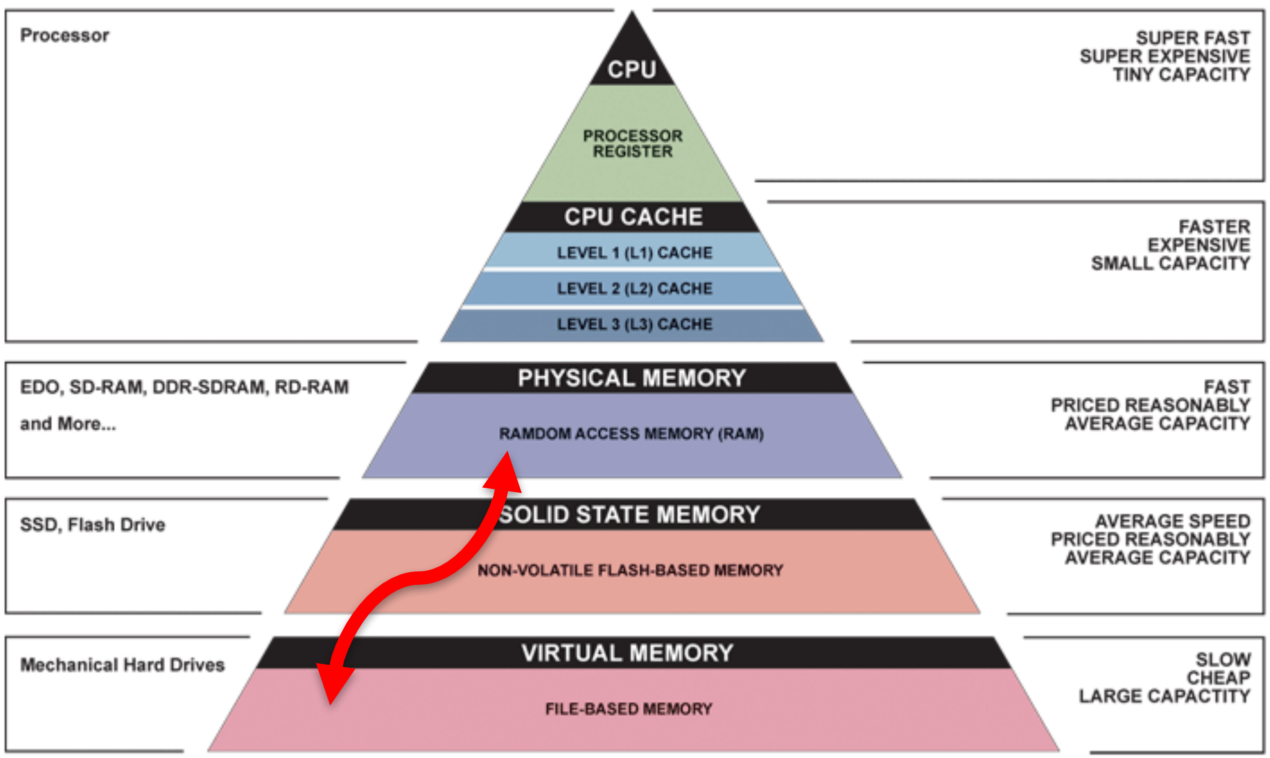
"Bare" 5-Stage Pipeline



- In a bare machine, the only kind of address is a physical address

What do we need Virtual Memory for? Reason 1: Adding Disks to Hierarchy

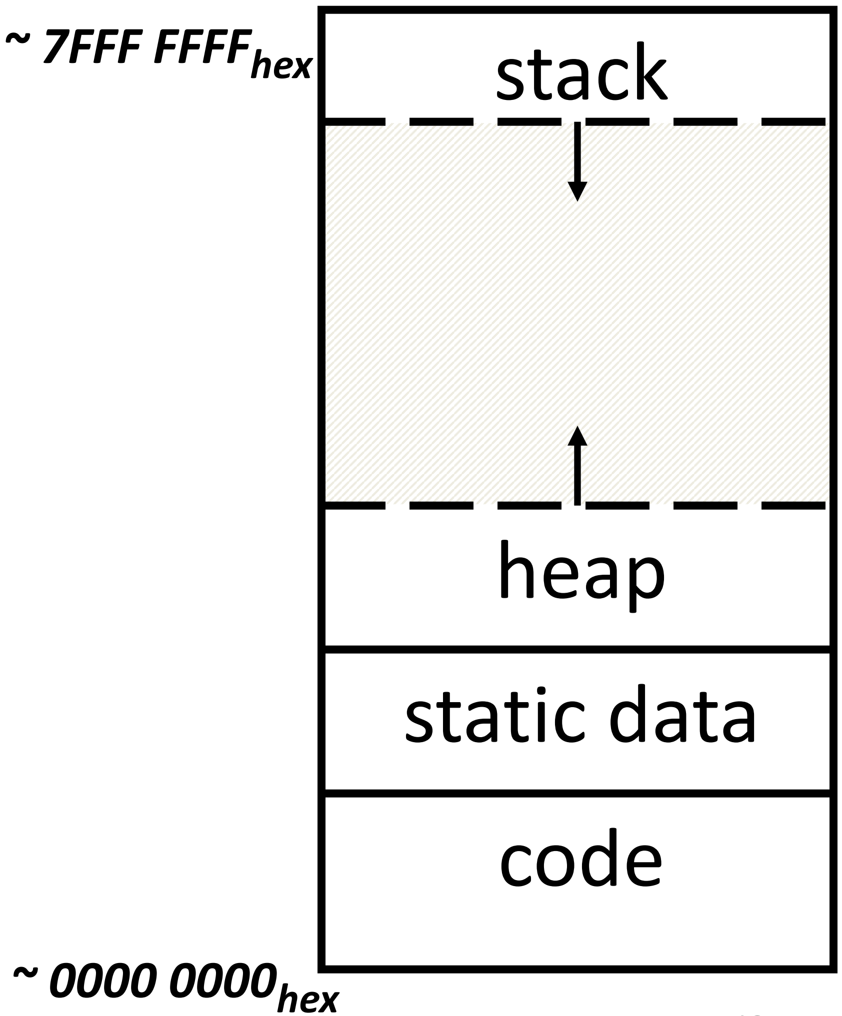
- Need to devise a mechanism to "connect" memory and disk in the memory hierarchy



▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

What do we need Virtual Memory for? Reason 2: Simplifying Memory for Apps

- Applications should see the straightforward memory layout we saw earlier ->
- User-space applications should think they own all of memory
- So we give them a **virtual** view of memory



What do we need Virtual Memory for? Reason 3: Protection Between Processes

- With a bare system, addresses issued with loads/stores are real **physical** addresses
- This means any program can issue any address, therefore can access any part of memory, even areas which it doesn't own
  - Ex: The OS data structures
- We should send all addresses through a mechanism that the OS controls, before they make it out to DRAM - **a translation mechanism**

Address Spaces

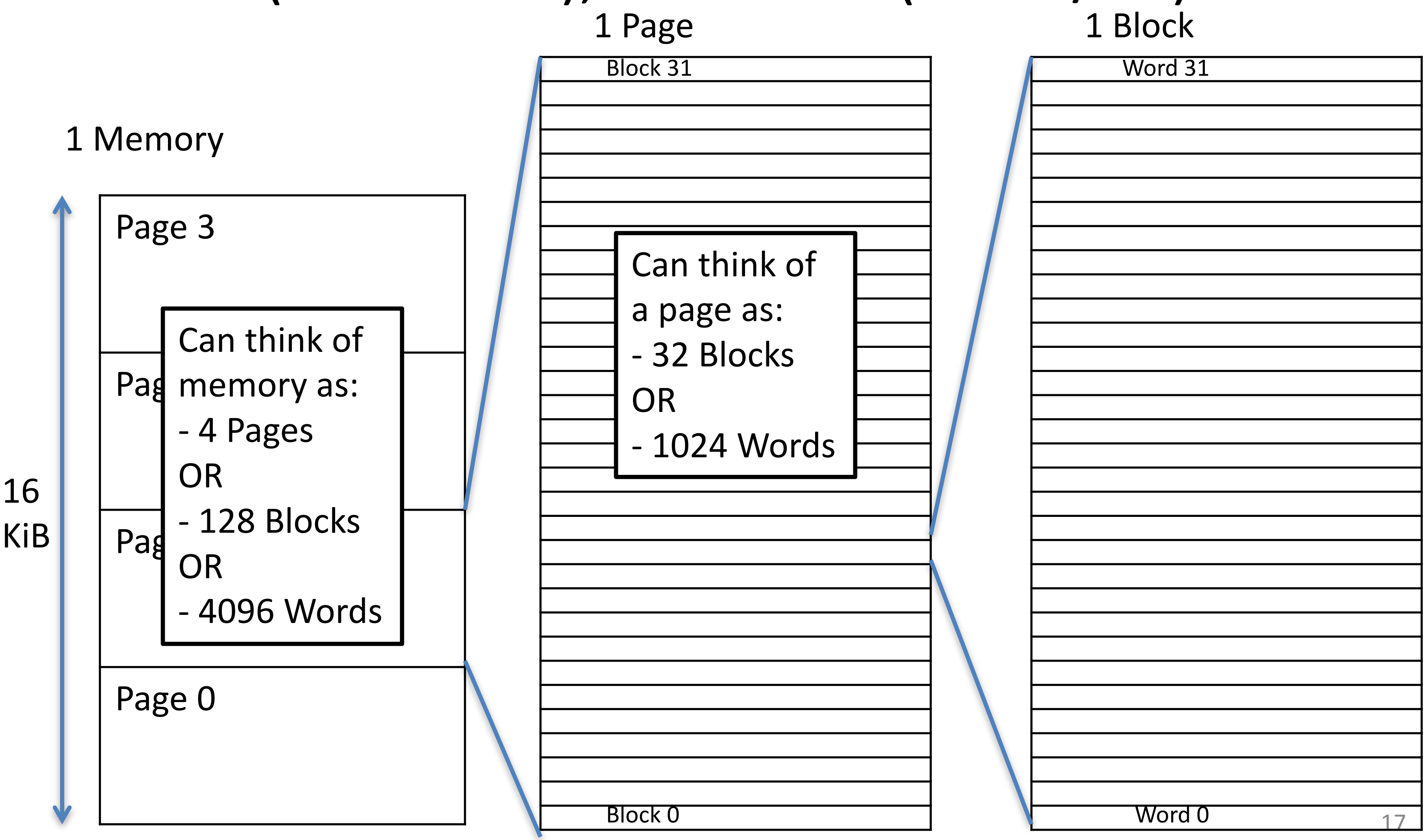
- The set of addresses labeling all of memory that we can access
- Now, 2 kinds:
  - **Virtual Address Space** - the set of addresses that the user program knows about
  - **Physical Address Space** - the set of addresses that map to actual physical cells in memory
    - Hidden from user applications
- So, we need a way to map between these two address spaces

Blocks vs. Pages

- In caches, we dealt with individual *blocks*
  - Usually ~64B on modern systems
  - We could “divide” memory into a set of blocks
- In VM, we deal with individual *pages*
  - Usually ~4 KB on modern systems
    - Larger sizes also available: 4MB, very modern 1GB!
  - Now, we’ll “divide” memory into a set of pages
- Common point of confusion: Bytes, Words, Blocks, Pages are all just different ways of looking at memory!

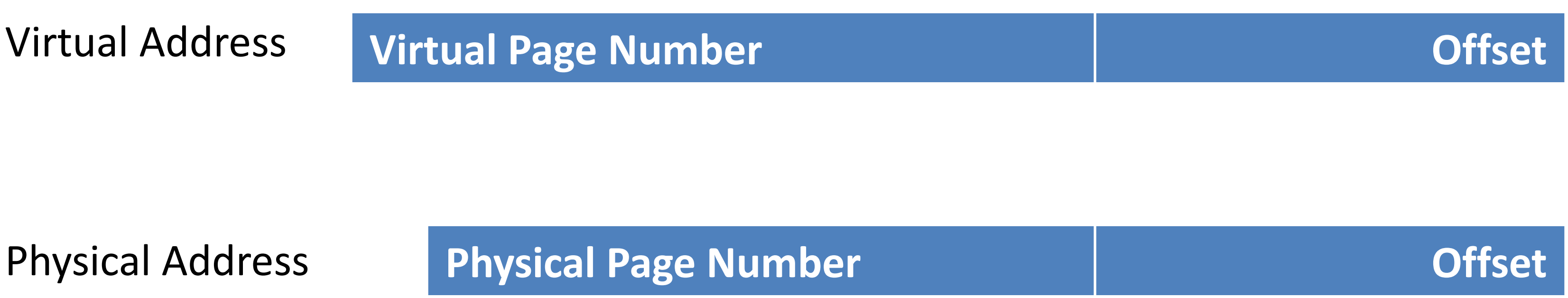
Bytes, Words, Blocks, Pages

Ex: 16 KiB DRAM, 4 KiB Pages (for VM), 128 B blocks (for caches), 4 B words (for lw/sw)

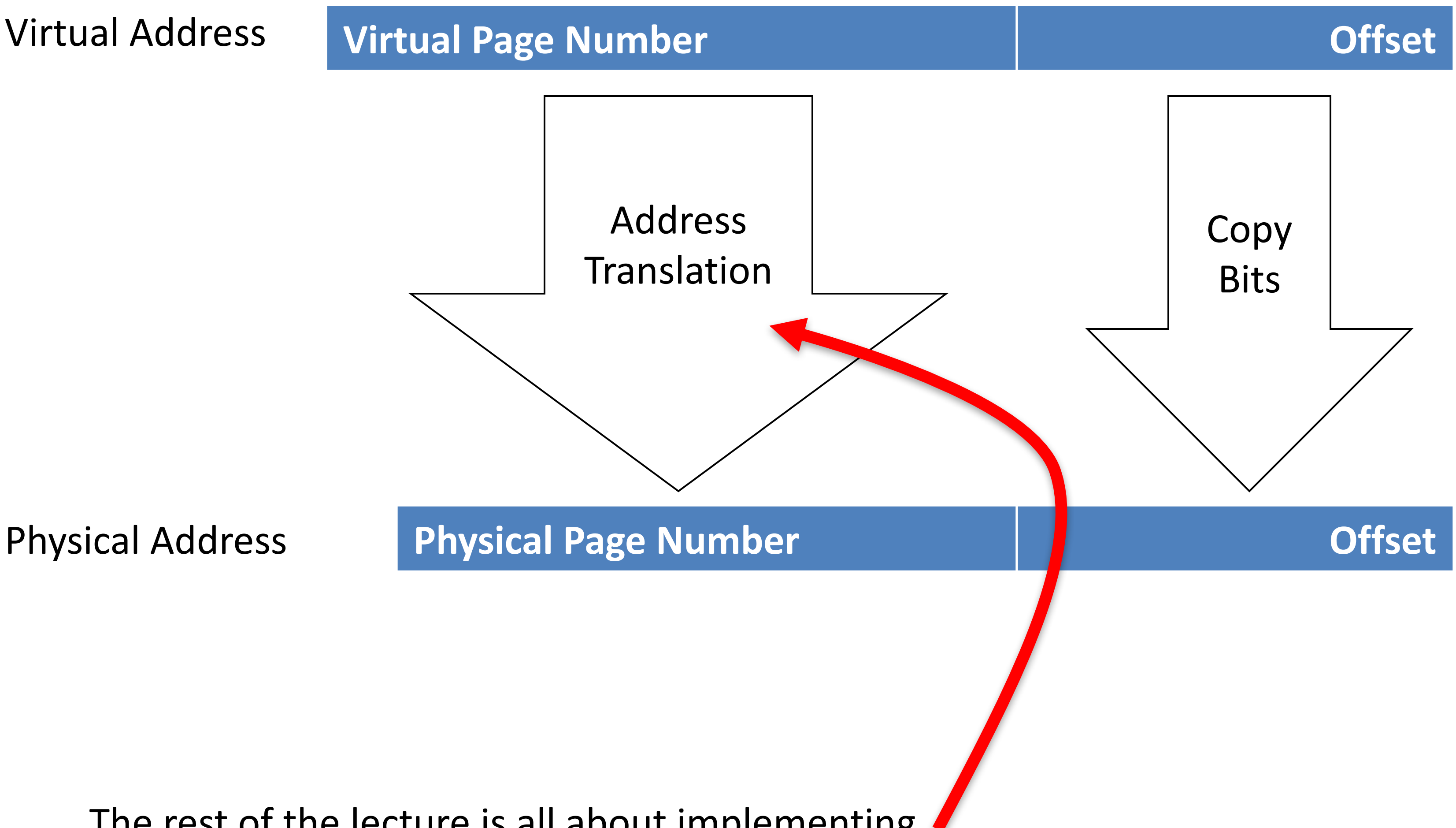


Address Translation

- So, what do we want to achieve at the hardware level?
  - Take a Virtual Address, that points to a spot in the Virtual Address Space of a particular program, and map it to a Physical Address, which points to a physical spot in DRAM of the whole machine



Address Translation

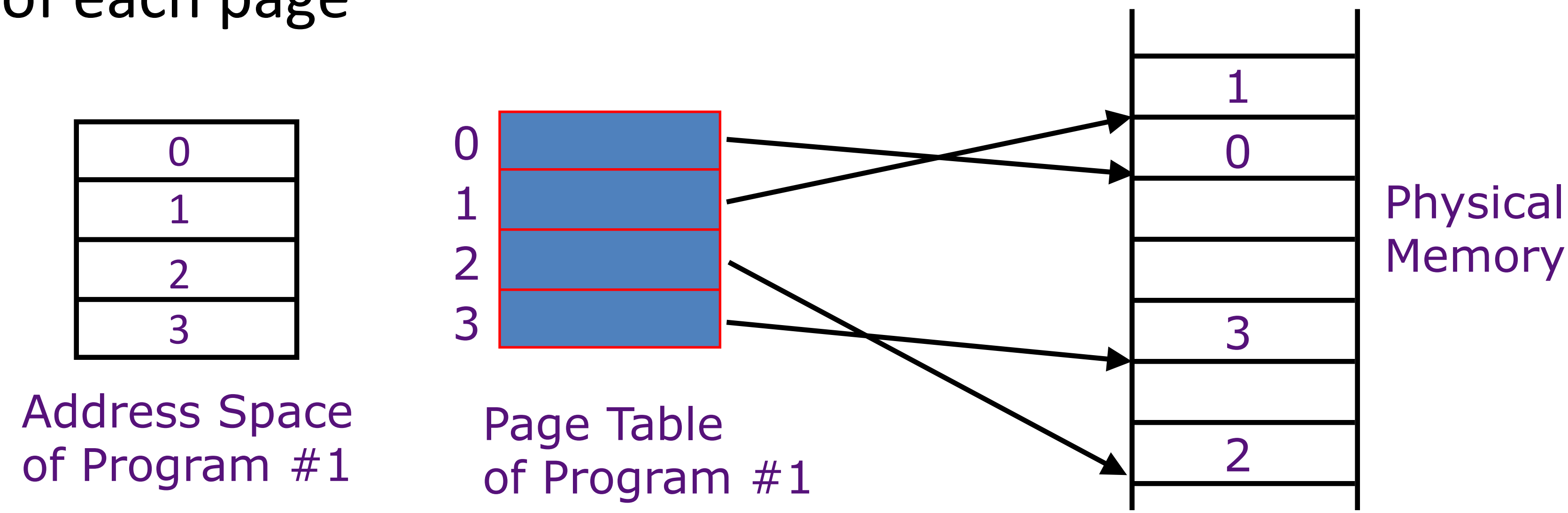


Paged Memory Systems

- Processor-generated address can be split into:

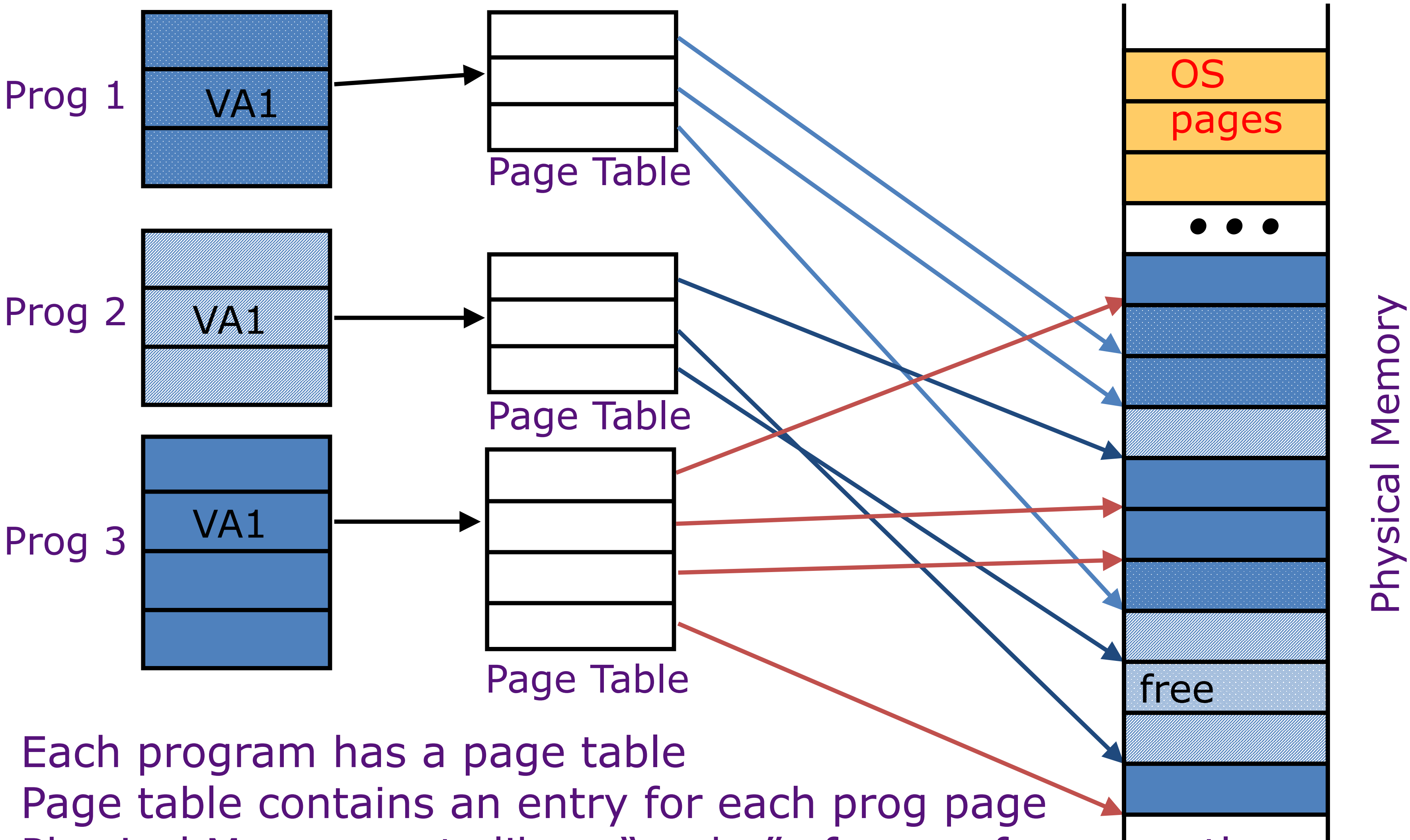
Virtual Page Number	Offset

- A *page table* contains the physical address of the base of each page



Page tables make it possible to store the pages of a program non-contiguously.

Private (Virtual) Address Space per Program



- Each program has a page table
- Page table contains an entry for each prog page
- Physical Memory acts like a "cache" of pages for currently running programs. **Not recently used pages are stored in secondary memory, e.g. disk (in "swap partition")**

Where Should Page Tables Reside?

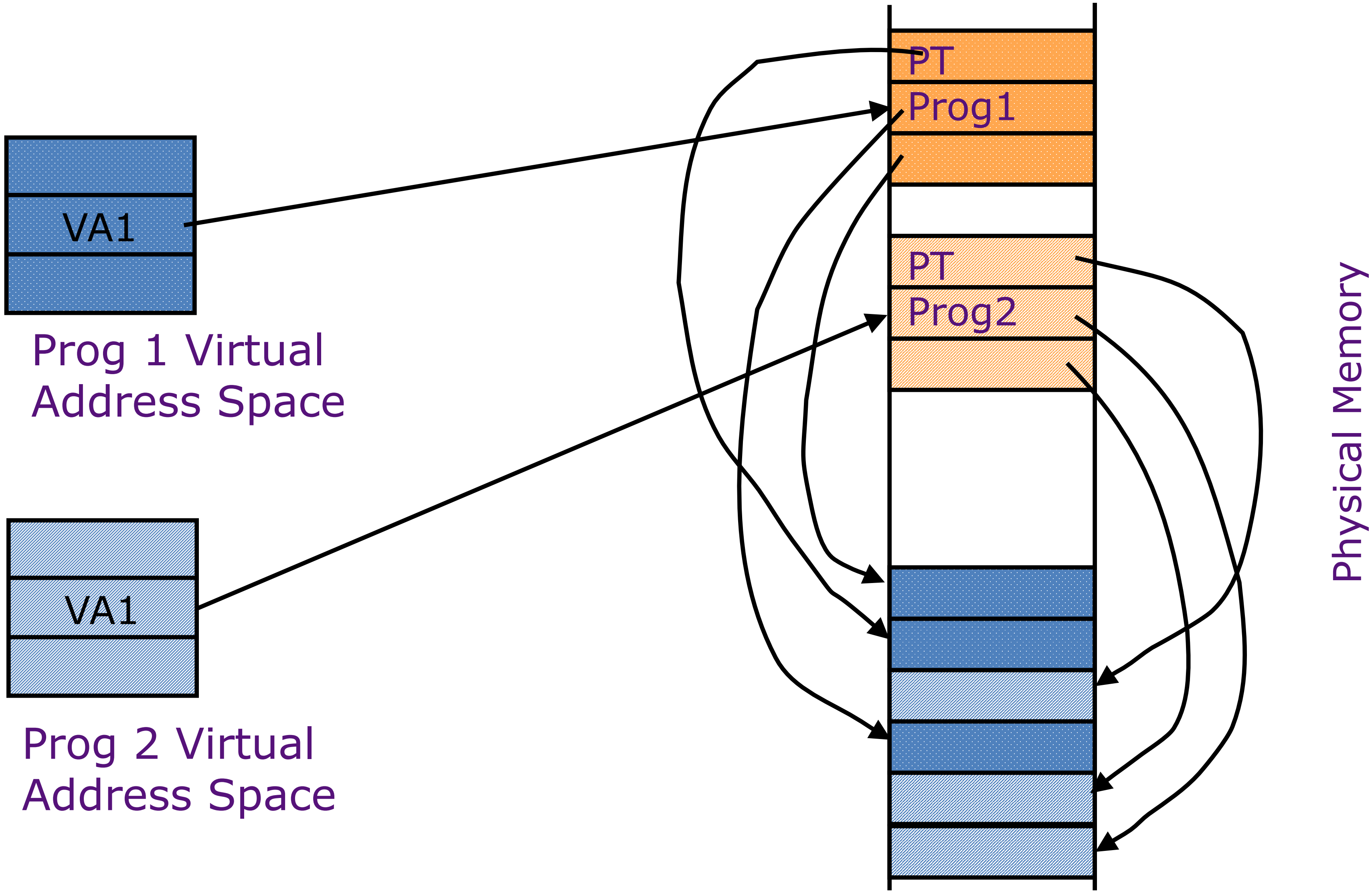
- Space required by the page tables (PT) is proportional to the address space, number of users, ...

  => Too large to keep in registers inside CPU

- Idea: Keep page tables in the main memory
  - Needs one reference to retrieve the page base address and another to access the data word

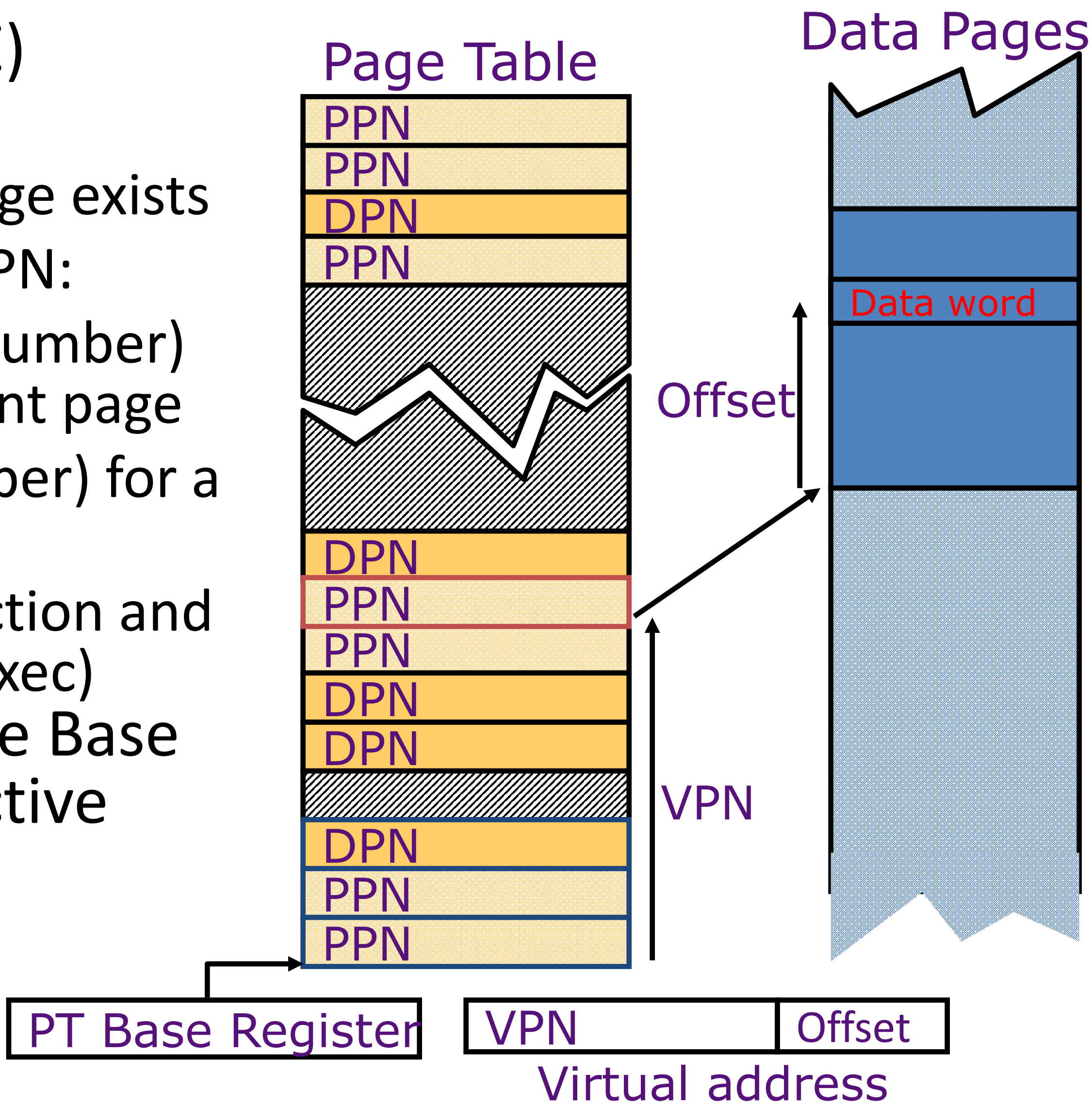
    => doubles the number of memory references! (but we can fix this using something we already know about…)

Page Tables in Physical Memory



Linear (simple) Page Table

- Page Table Entry (PTE) contains:
  - 1 bit to indicate if page exists
  - And either PPN or DPN:
    - PPN (physical page number) for a memory-resident page
    - DPN (disk page number) for a page on the disk
  - Status bits for protection and usage (read, write, exec)
- OS sets the Page Table Base Register whenever active user process changes



Suppose an instruction references a memory page that isn't in DRAM?

- We get a exception of type "page fault"
- Page fault handler does the following:
  - If virtual page doesn't yet exist, assign an unused page in DRAM, or if page exists …
  - Initiate transfer of the page we're requesting from disk to DRAM, assigning to an unused page
  - If no unused page is left, a *page currently in DRAM is selected to be replaced* (based on usage)
  - The replaced page is written (back) to disk, page table entry that maps that VPN->PPN is marked as invalid/DPN
  - Page table entry of the page we're requesting is updated with a (now) valid PPN

Size of Linear Page Table

With 32-bit memory addresses, 4-KB pages:

=> $2^{32} / 2^{12} = 2^{20}$ virtual pages per user, assuming 4-Byte PTEs,

=> 2^{20} PTEs, i.e, 4 MB page table per process!

Larger pages?

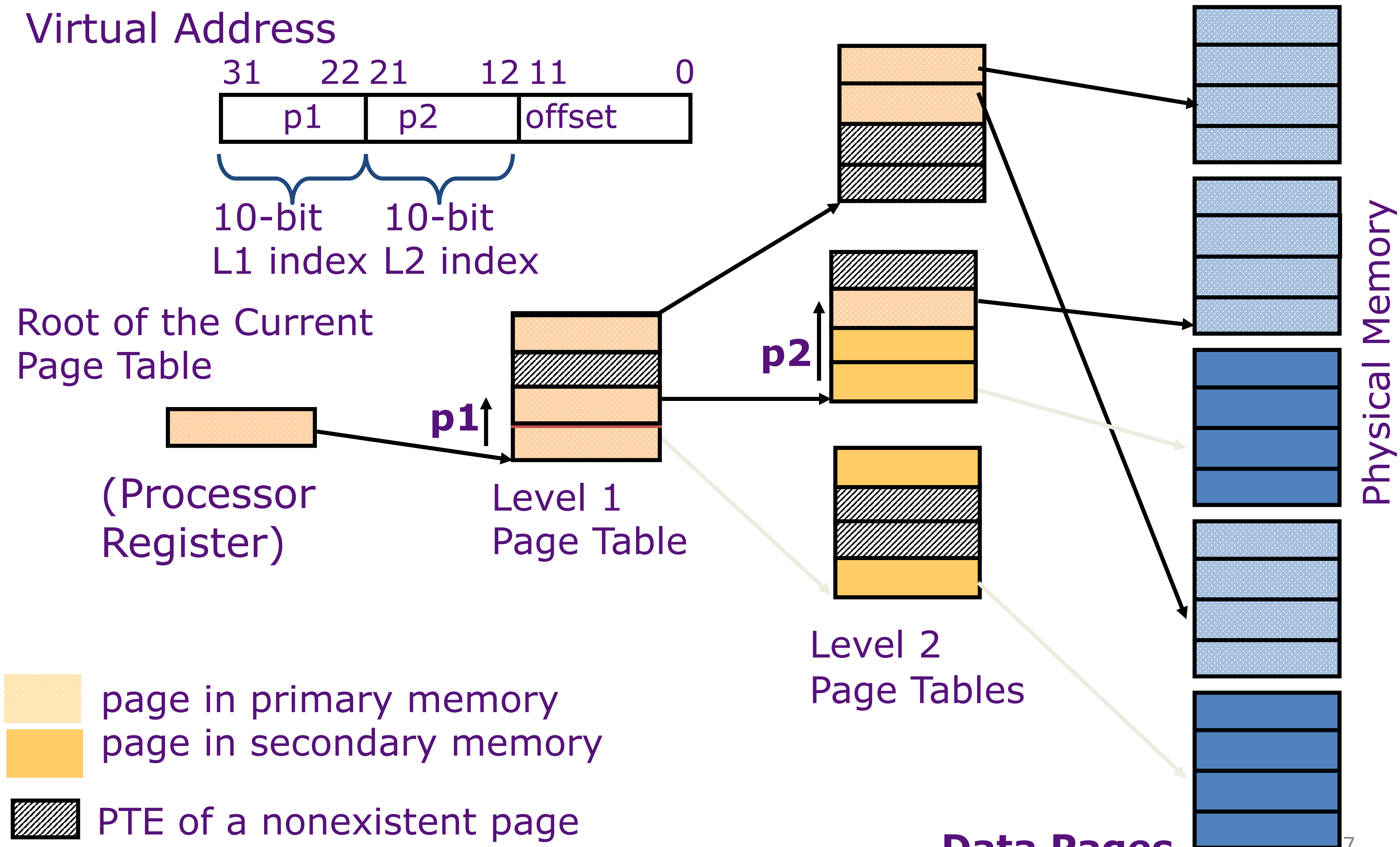
- Internal fragmentation (Not all memory in page gets used)
- Larger page fault penalty (more time to read from disk)

What about 64-bit virtual address space???

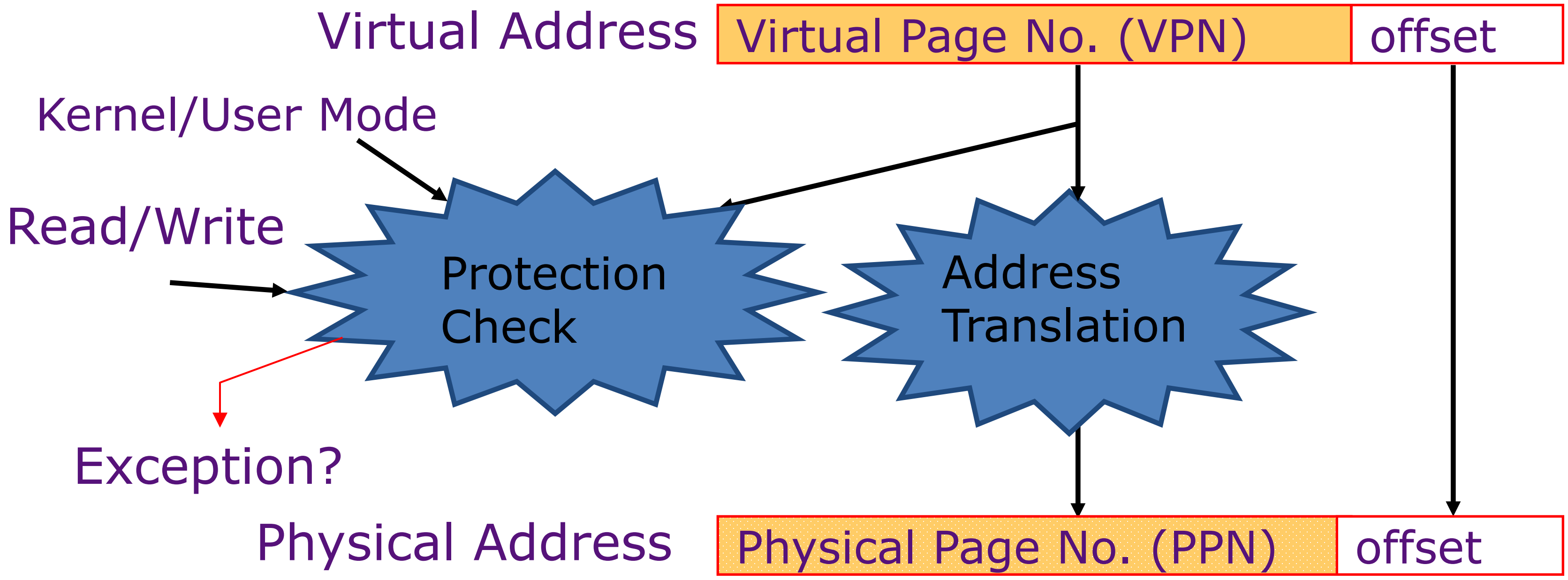
- Even 1MB pages would require 2^{44} 8-Byte PTEs (35 TB!)

What is the “saving grace” ? Most processes only use a set of high address (stack), and a set of low address (instructions, heap)

Hierarchical Page Table – exploits sparcity of virtual address space use



Address Translation & Protection



- Every instruction and data access needs address translation and protection checks

A good VM design needs to be fast (~ one cycle) and space efficient

Translation Lookaside Buffers (TLB)

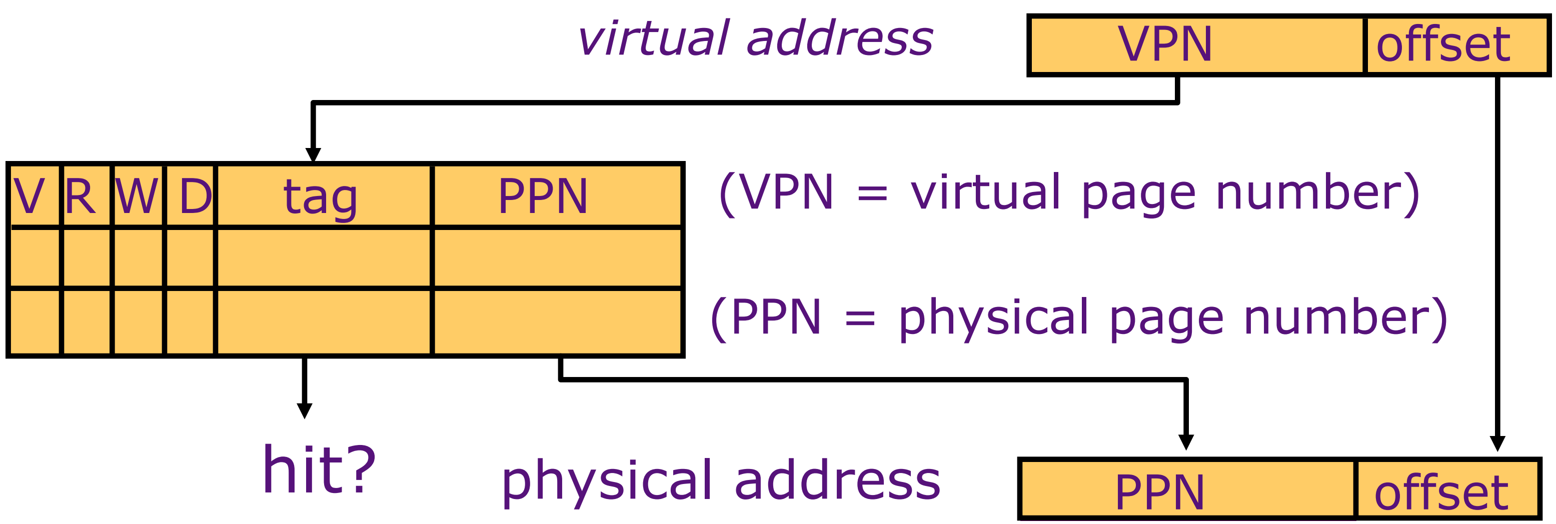
Address translation is very expensive!

In a two-level page table, each reference becomes several memory accesses

Solution: *Cache some translations in TLB*

TLB hit => *Single-Cycle Translation*

TLB miss => *Page-Table Walk to refill*



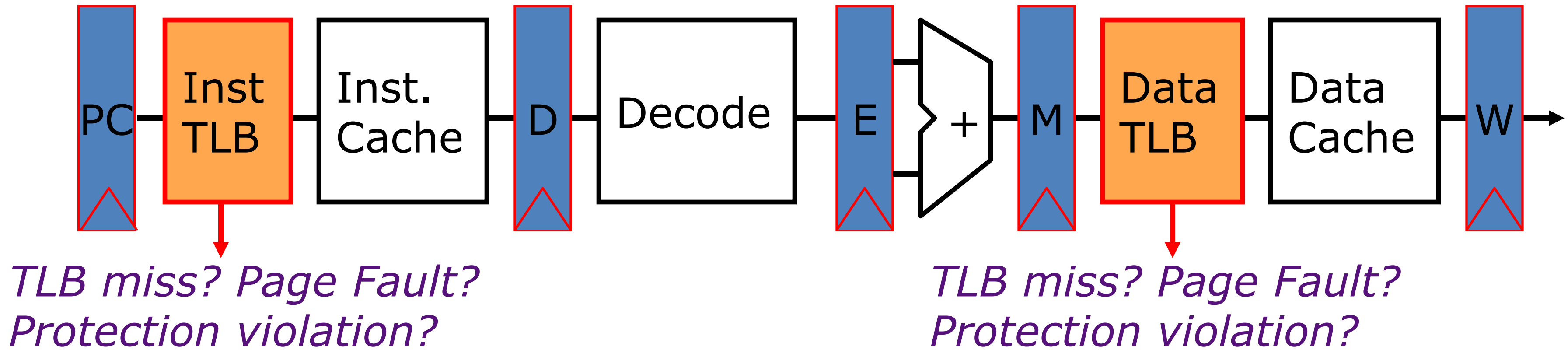
TLB Designs

- Typically 32-128 entries, usually fully associative
  - Each entry maps a large page, hence less spatial locality across pages => more likely that two entries conflict
  - Sometimes larger TLBs (256-512 entries) are 4-8 way set-associative
  - Larger systems sometimes have multi-level (L1 and L2) TLBs
- Random or FIFO replacement policy
- Upon context switch? New VM space! Flush TLB …
- "TLB Reach": Size of largest virtual address space that can be simultaneously mapped by TLB

Example: 64 TLB entries, 4KB pages, one page per entry

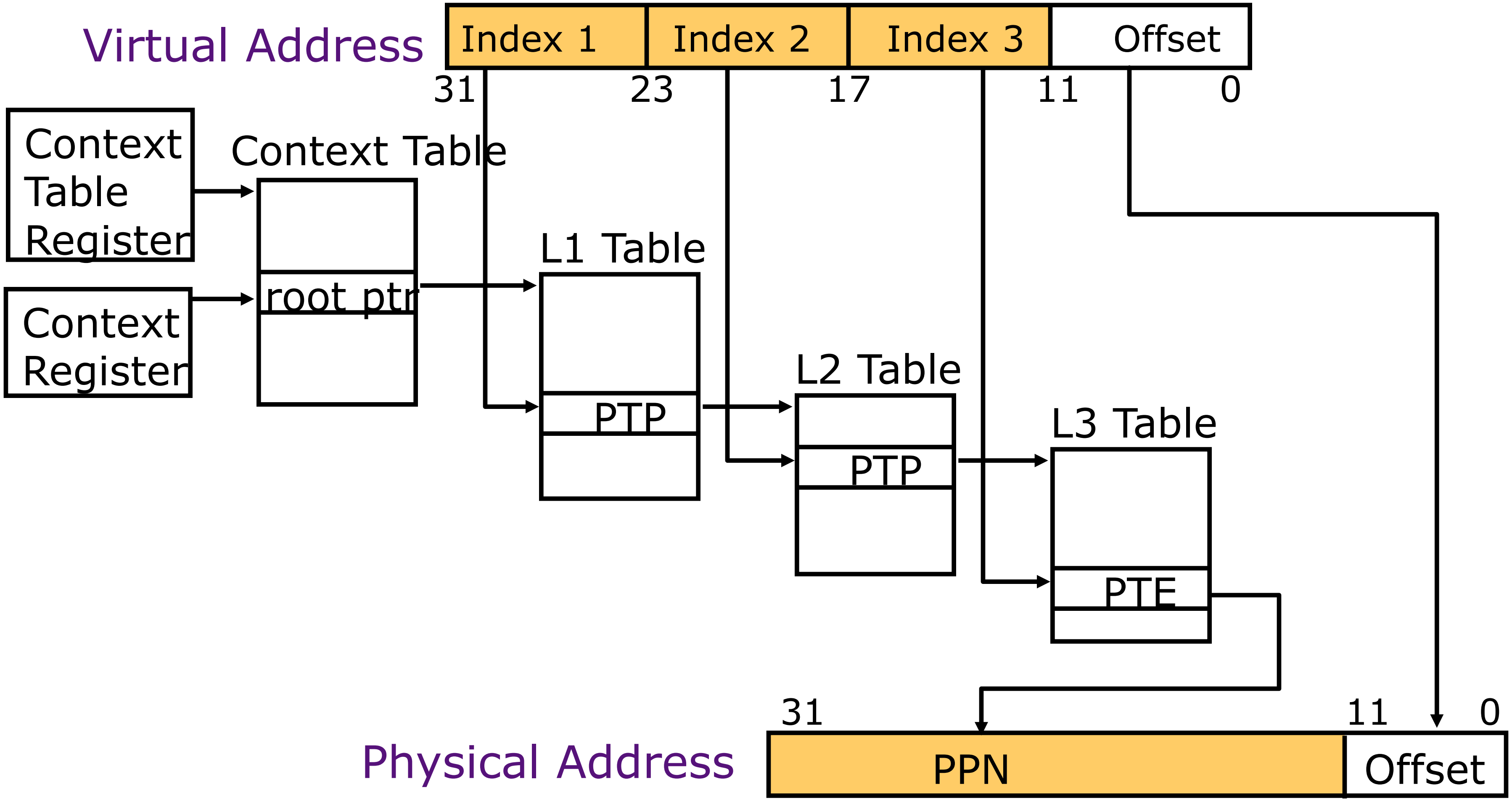
TLB Reach = ____*?*

VM-related events in pipeline



- Handling a TLB miss needs a hardware or software mechanism to refill TLB
  - usually done in hardware now
- Handling a page fault (e.g., page is on disk) needs a *precise* trap so software handler can easily resume after retrieving page
- Handling protection violation may abort process

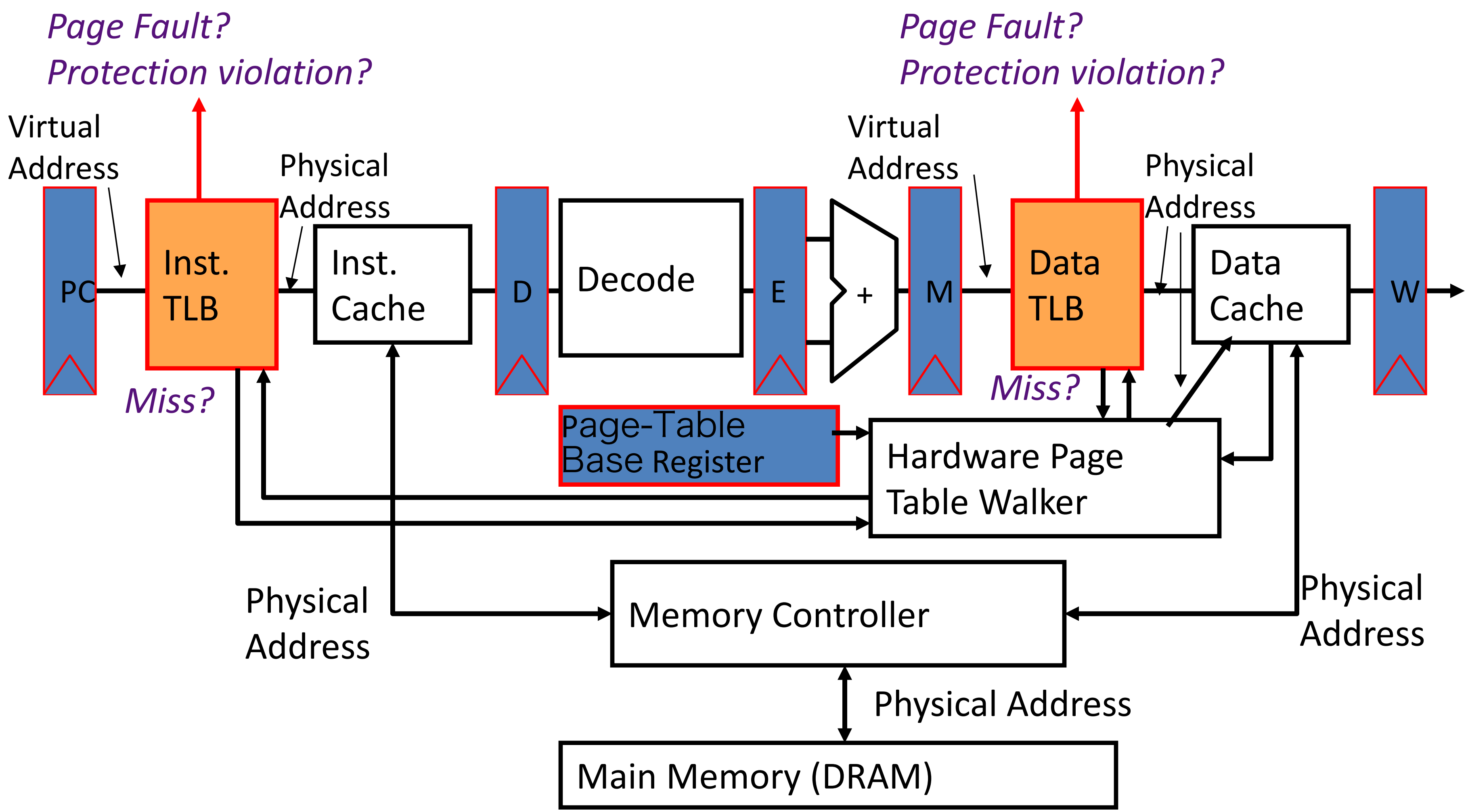
Hierarchical Page Table Walk: SPARC v8



MMU does this table walk in hardware on a TLB miss

Page-Based Virtual-Memory Machine

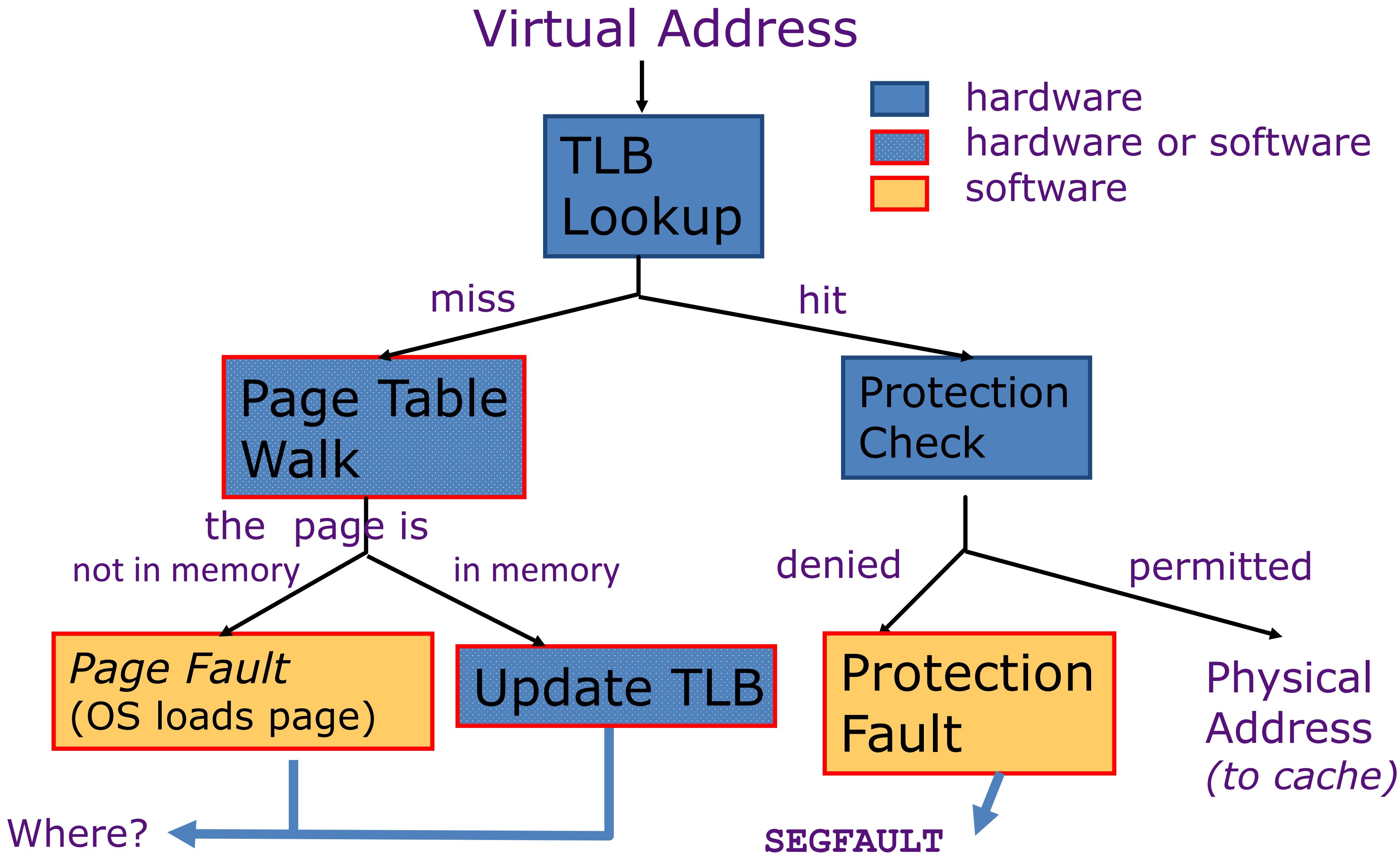
(Hardware Page-Table Walk)



- Assumes page tables held in untranslated physical memory

Address Translation:

putting it all together



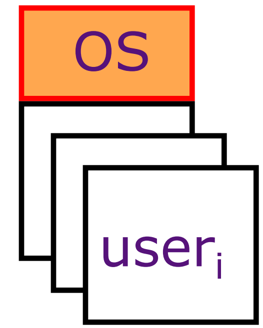
Modern Virtual Memory Systems

Illusion of a large, private, uniform store

Protection & Privacy

several users, each with their private address space and one or more shared address spaces

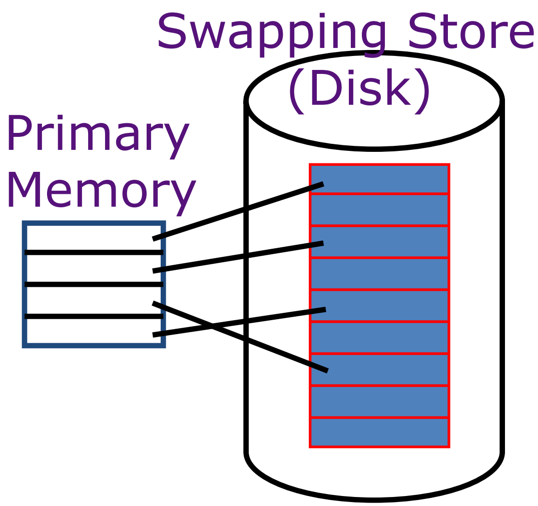
page table ≡ name space



Demand Paging

Provides the ability to run programs larger than the primary memory

Hides differences in machine configurations



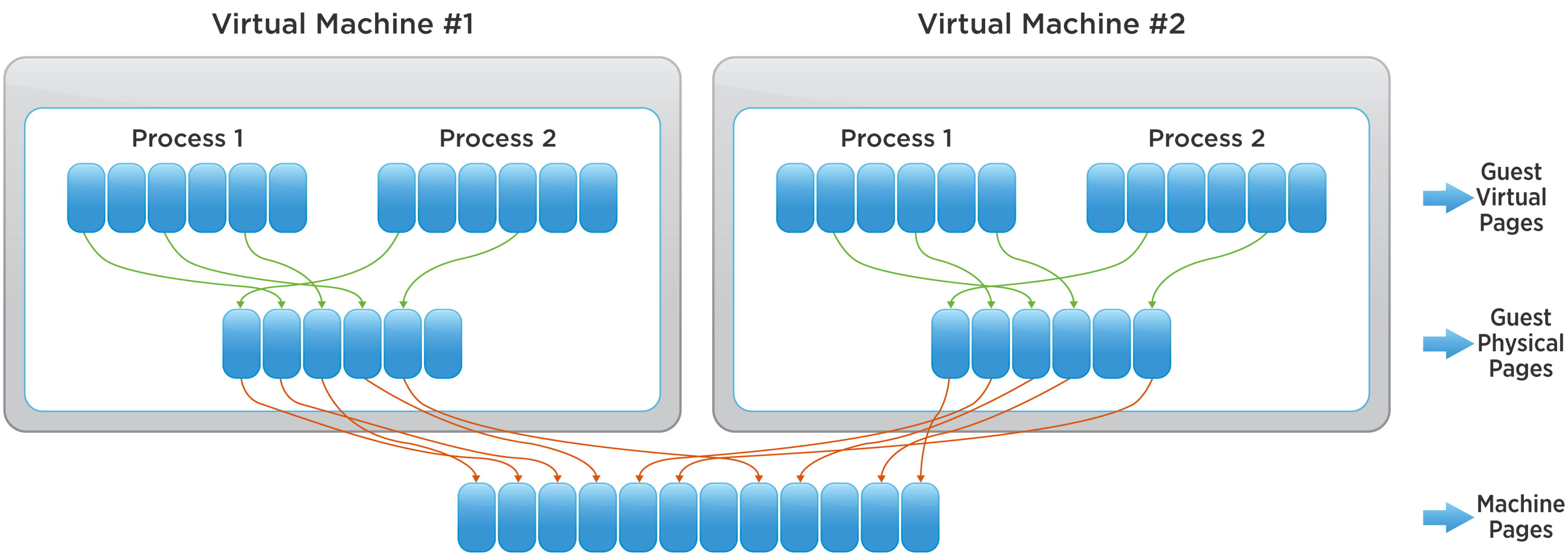
The price is address translation on each memory reference

Virtual Machine

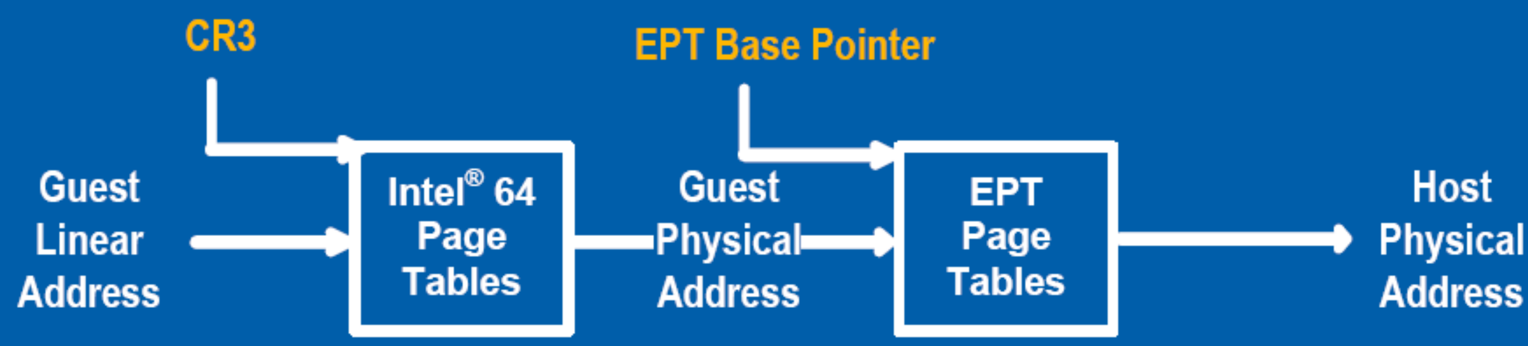
- Virtual Memory (VM) != Virtual Machine (VM) (e.g. Virtual Box)
- Emulation: Run a complete virtual CPU & Memory & … - a complete virtual machine in software (e.g. MARS)
- Virtual Machine: Run as many instructions as possible directly on CPU, only simulate some parts of the machine.
- Last lecture: Supervisor Mode & Use Mode;
  now also: Virtual Machine Mode
  - Host OS activates virtual execution mode for guest OS =>
  - Guest OS thinks it runs in supervisor mode, but in fact it doesn't have access to physical memory! (among other limitations)
- CPUs support it (AMD-V, Intel VT-x), e.g. new Intel instructions: VMPTRLD, VMPTRST, VMCLEAR, VMREAD, VMWRITE, VMCALL, VMLAUNCH, VMRESUME, VMXOFF, and VMXON

What about the memory in Virtual Machines?

- Need to translate Guest Virtual Address to Guest Physical address to Machine (Host) Physical address: Earlier the Guest part was done (transparently) in software by the Virtual Machine … now in hardware!



EPT: Overview



- Intel® 64 page tables
  - Map guest-linear to guest-physical (translated again)
  - Can be read and written by guest
- New EPT page tables under VMM control
  - Map guest-physical to host-physical (accesses memory)
  - Referenced by new EPT base pointer
- No VM exits due to page faults, INVLPG, or CR3 accesses

Intel® VT Roadmap: Overview

Vector 3: I/O Focus

PCI-SIG

Standards for I/O-device sharing:
- Natively sharable I/O devices
- Endpoint DMA-translation caching

Vector 2: Platform Focus

VT-d

Infrastructure for I/O-device virtualization:
- DMA protection and remapping
- Interrupt filtering and remapping

Vector 1: Processor Focus

VT-x

VT-i

Establish foundation for virtualization in the Intel® 64 and Itanium® architectures…

… followed by on going evolution of support:
- Microarchitectural (e.g., lower VM entry/exit costs)
- Architectural (e.g., extended page tables – EPT)

VMM Software Evolution

Software-only VMMs
- Binary translation
- Paravirtualization
- Device Emulation

Simpler and more **Secure** VMMs through foundation of virtualizable ISAs

Improved CPU and I/O virtualization **Performance** and **Functionality** as VMMs exploit infrastructure provided by VT-x, VT-i, VT-d

Past

No Hardware Support

Today

VMM software evolution over time with hardware support

Conclusion: VM features track historical uses

- **Bare machine, only physical addresses**
  - One program owned entire machine
- **Batch-style multiprogramming**
  - Several programs sharing CPU while waiting for I/O
  - Base & bound: translation and protection between programs (not virtual memory)
  - Problem with external fragmentation (holes in memory), needed occasional memory defragmentation as new jobs arrived
- **Time sharing**
  - More interactive programs, waiting for user. Also, more jobs/second.
  - Motivated move to fixed-size page translation and protection, no external fragmentation (but now internal fragmentation, wasted bytes in page)
  - Motivated adoption of virtual memory to allow more jobs to share limited physical memory resources while holding working set in memory
- **Virtual Machine Monitors**
  - Run multiple operating systems on one machine
  - Idea from 1970s IBM mainframes, now common on laptops
    - e.g., run Windows on top of Mac OS X
  - Hardware support for two levels of translation/protection
    - Guest OS virtual -> Guest OS physical -> Host machine physical
  - Also basis of Cloud Computing
    - Virtual machine instances on EC2