COS 318: Operating Systems

Protection and Virtual Memory

Outline

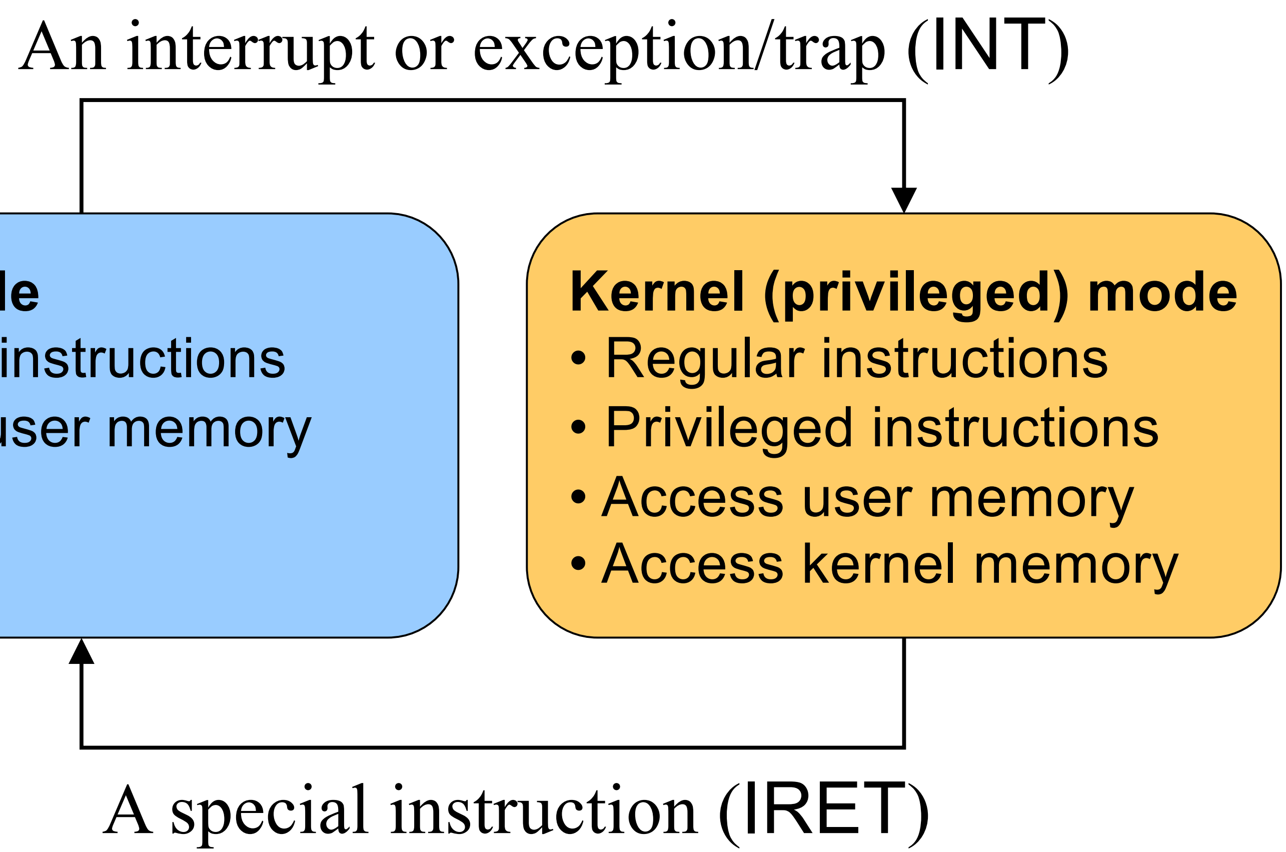
- Protection Mechanisms and OS Structures
- Virtual Memory: Protection and Address Translation

Some Protection Goals

- CPU
  - Allow kernel to take CPU away to prevent a user from using CPU forever
  - Users should not have this ability
- Memory
  - Prevent a user from accessing others' data
  - Prevent users from modifying kernel code and data structures
- I/O
  - Prevent users from performing "illegal" I/Os
- Difference between protection and security?

Architecture Support for CPU Protection

- **Privileged Mode**



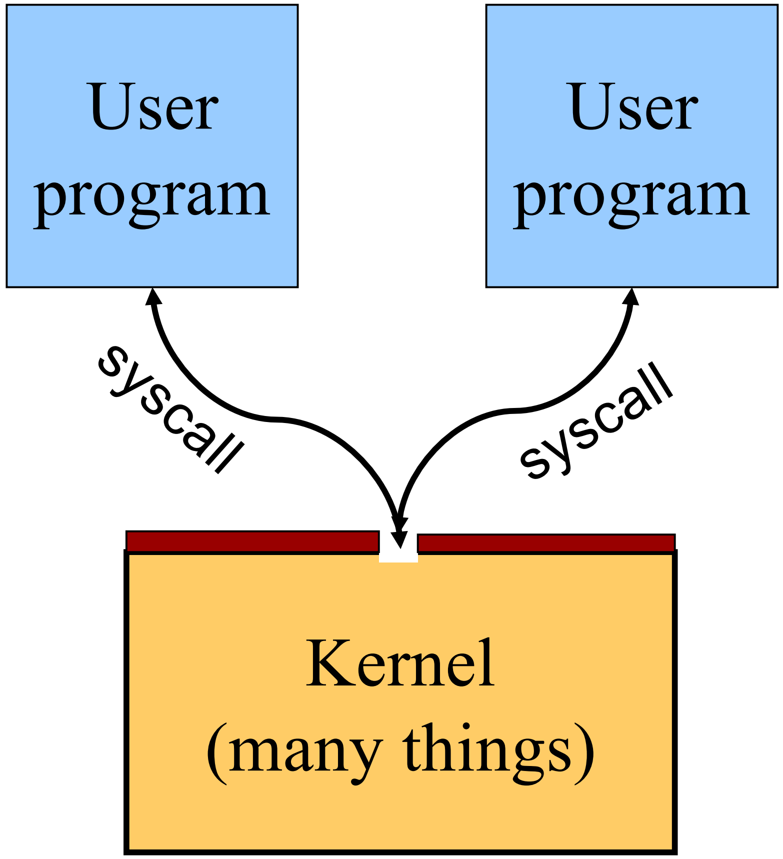
Privileged Instruction Examples

- Memory address mapping
- Flush or invalidate data cache
- Invalidate TLB entries
- Load and read system registers
- Change processor modes from kernel to user
- Change the voltage and frequency of processor
- Halt a processor
- Reset a processor
- Perform I/O operations

- Q: Other architectural support for protection in system?

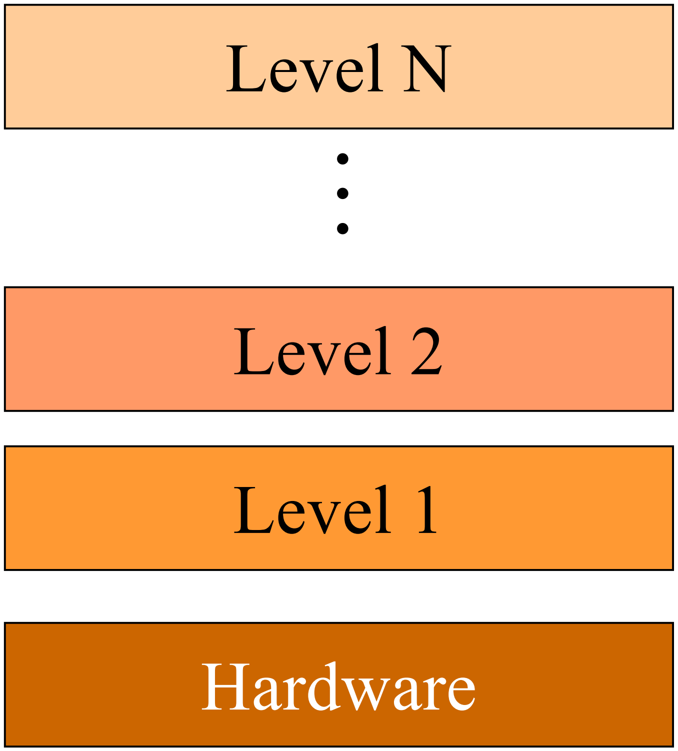
OS Structures and Protection: Monolithic

- All kernel routines are together, linked in single large executable
  - Each can call any other
  - Services and utilities
- Provides a system call API
- Examples:
  - Linux, BSD Unix, Windows, …
- Pros
  - Shared kernel space
  - Good performance
- Cons
  - Instability: crash in any procedure brings system down
  - Unweildy/difficult to maintain, extend



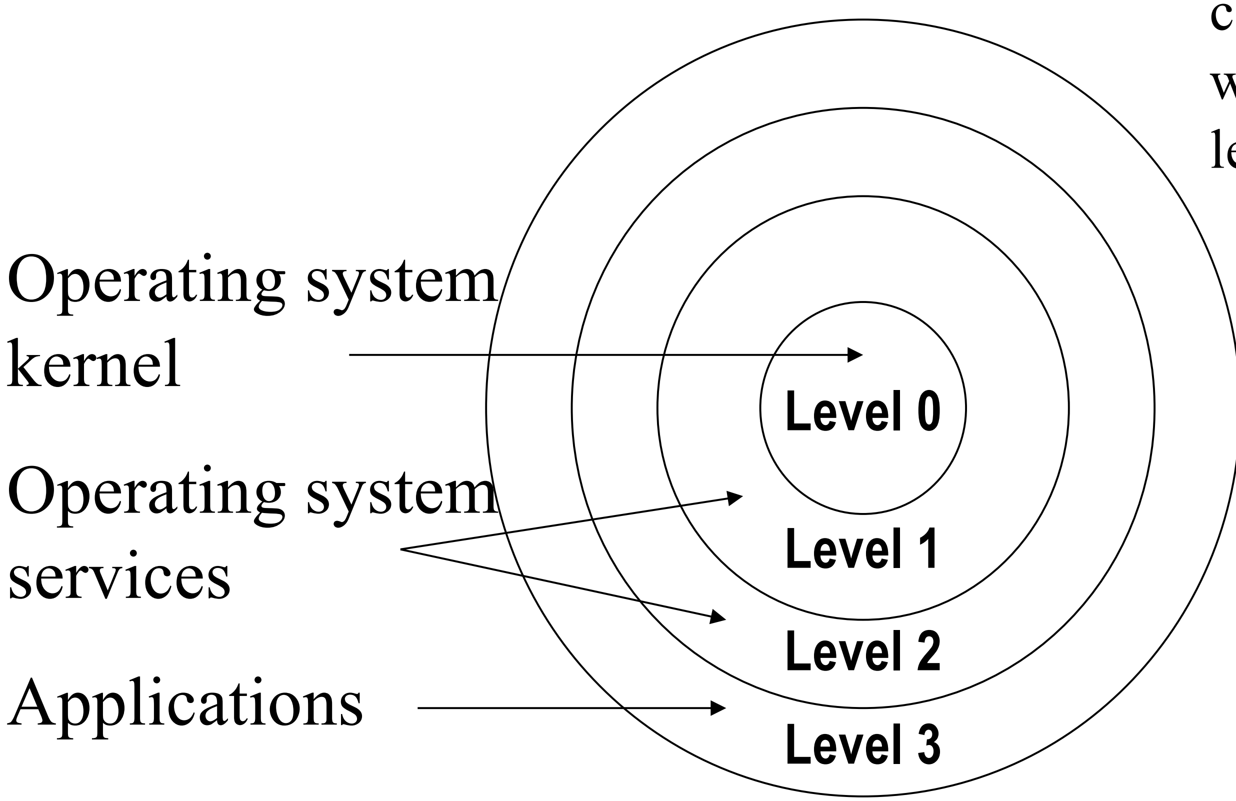
Layered Structure

- Hiding information at each layer
- Layered dependency
- Examples
  - THE (6 layers)
    - Mostly for functionality splitting
  - MS-DOS (4 layers)
- Pros
  - Layered abstraction
    - Separation of concerns, elegance
- Q: Cons?
  - Inefficiency
  - Inflexibility



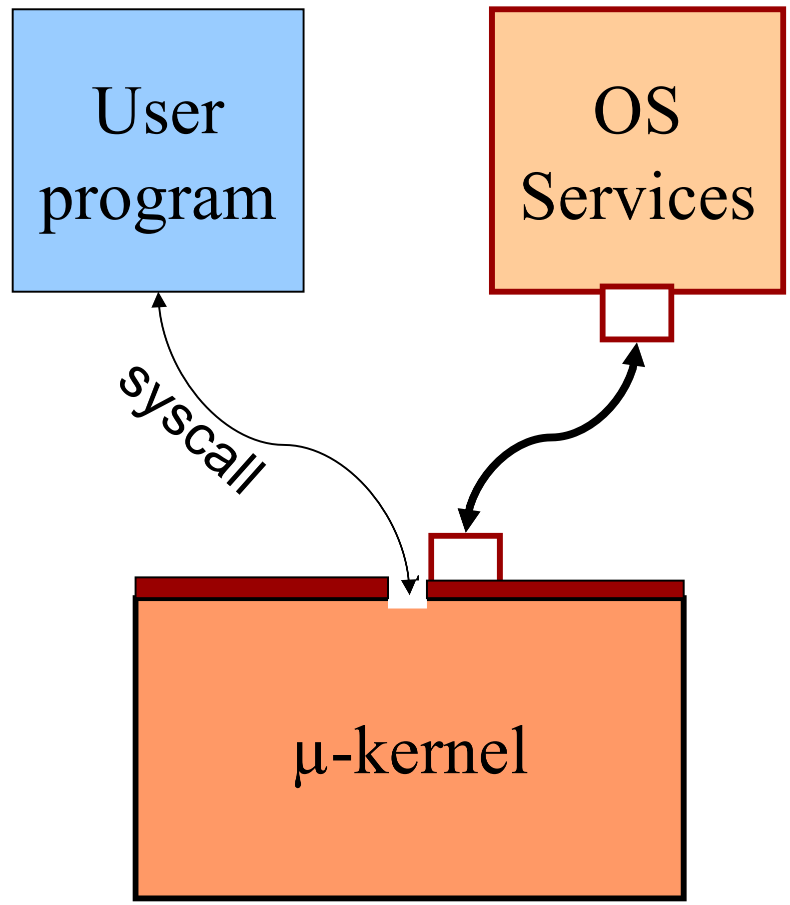
Possible Implementation: Protection Rings

Privileged instructions can be executed only when current privileged level (CPR) is 0



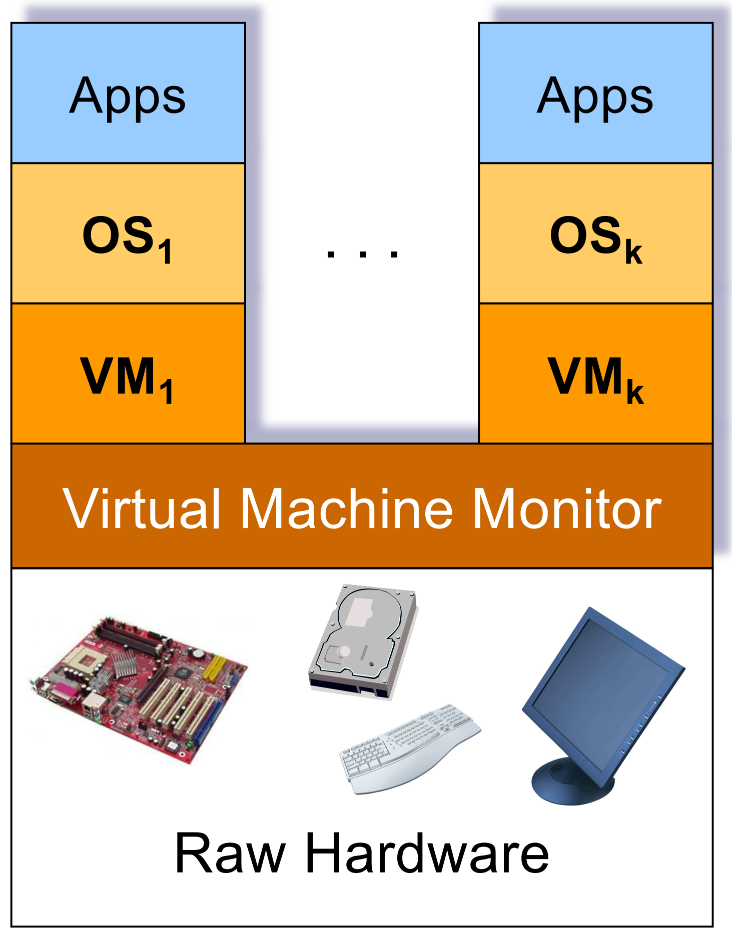
Microkernel Structure

- Services are regular processes
- Micro-kernel obtains services for users by messaging with services
- Examples:
  - Mach, Taos, L4, OS-X
- Pros?
  - Flexibility to modify services
  - Fault isolation
- Cons?
  - Inefficient (boundary crossings)
  - Inconvenient to share data between kernel and services
  - Just shifts the problem, to level with less protection? Testing?



Virtual Machine

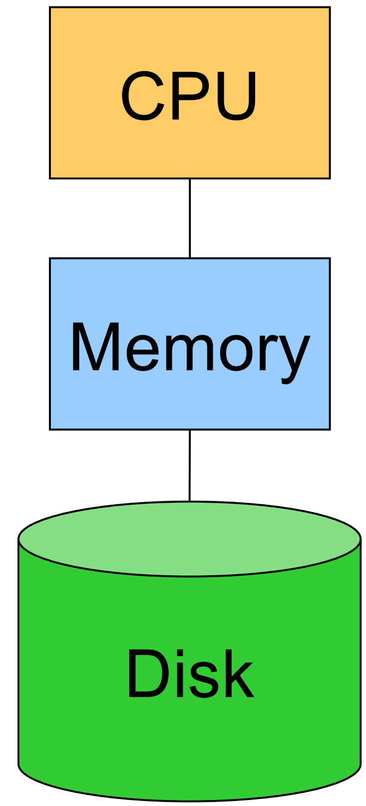
- Virtual machine monitor
  - Virtualize hardware
  - Run several OSes
  - Examples
    - IBM VM/370
    - Java VM
    - VMWare, Xen



- What would you use a virtual machine for?

Memory Management: The Big Picture

- DRAM is fast, but relatively expensive
- Disk is inexpensive, but slow
  - 100X less expensive
  - 100,000X longer latency
  - 1000X less bandwidth
- Goals
  - Make programmers not have to worry about this
  - Run programs efficiently
  - Make the system safe

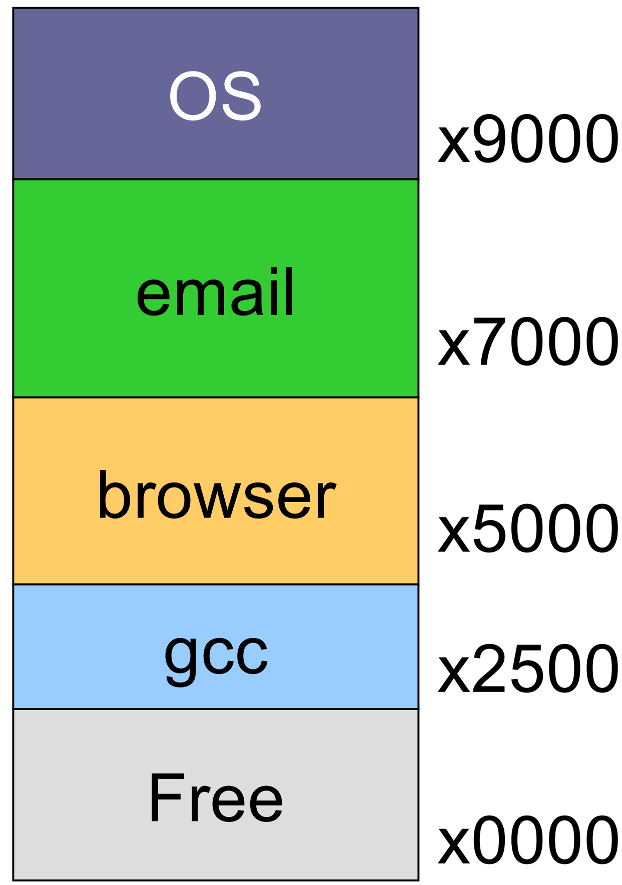


Problems

- Memory capacity
  - All my process's data don't fit in physical memory
  - There are many processes
- Locating data in memory
  - Where are my data in memory and where are yours?
- Protection
  - A user process should not do bad things to other processes: write or read their data without permission
  - A user process should not crash the system
- Scalability
  - The more processes a system can handle, the better

Consider A Simple System

- Only physical memory
  - Applications use it directly
- Run three processes
  - Email, browser, gcc
- What if
  - browser writes at x7050?
  - email needs to expand?
  - browser needs more memory than is on the machine?



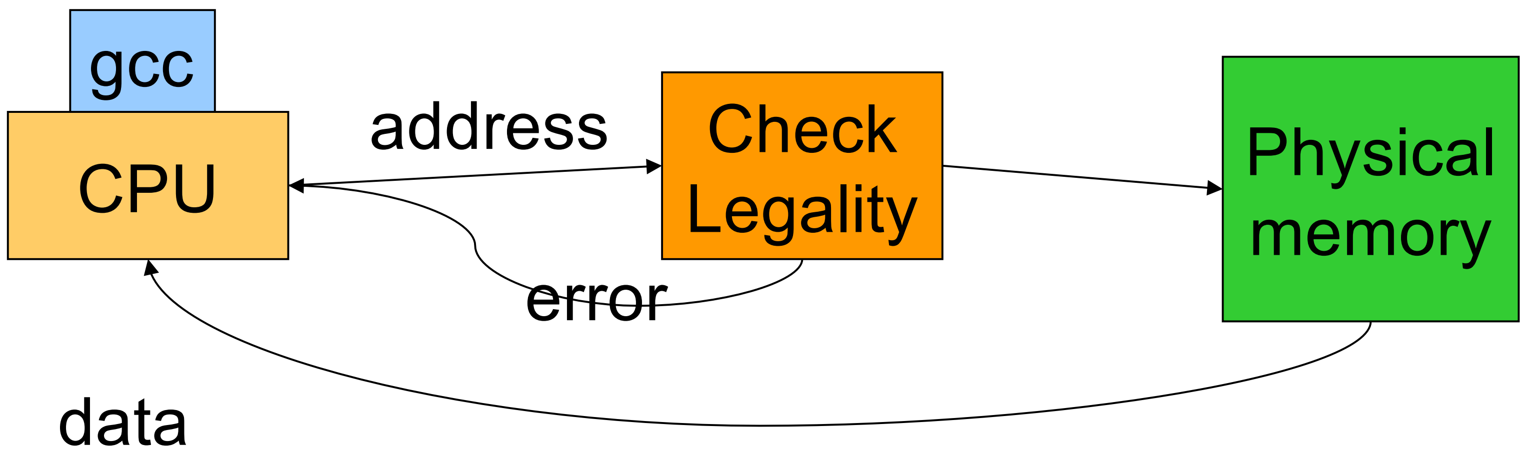
Need to Handle

- Protection
- Finiteness
  - Not having entire application/data in memory at once
  - Relocation
  - Not having programmer worry about it (too much)

Handling Protection

Check legality

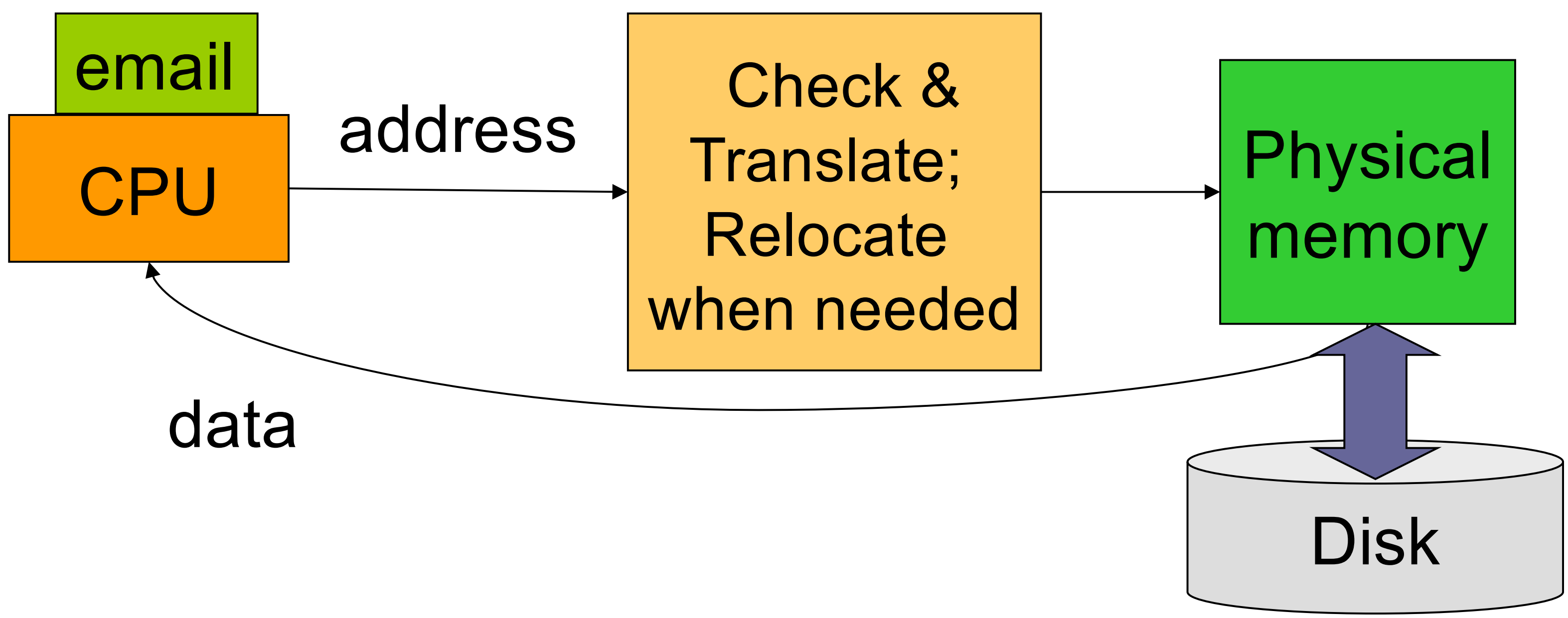
- Errors/malice in one process should not affect others
- For each process, check each load and store instruction to allow only legal memory references



Handling Finiteness

Address Translation: Mapping and Relocation

- A process should be able to run regardless of physical memory size or where its data are physically placed
- Give each process a large, static “fake” address space that is large and contiguous and entirely its own
- As process runs, translate (map) load/store to physical addresses. Relocate (change mappings) as needed



Virtual Memory

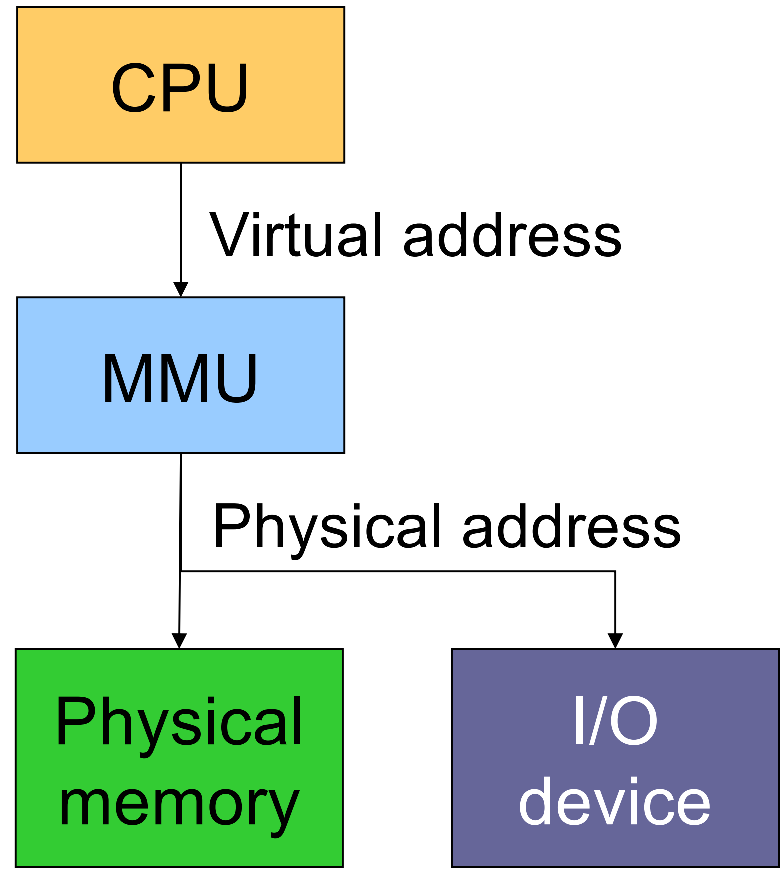
- Flexible
  - Processes (and data) can move in memory as they execute, and can be part in memory and part on disk
- Simple
  - Applications generate loads and stores to addresses in the contiguous, large, “fake” address space
- Efficient
  - 20/80 rule: 20% of memory gets 80% of references
  - Keep the 20% in physical memory (a form of caching)
- Protective
  - Protection check integrated with translation mechanism

Address Mapping

- Must have some "mapping" mechanism
  - Map virtual to physical addresses in RAM or disk
- Mapping must have some granularity
  - Finer granularity provides more flexibility
  - Finer granularity requires more mapping information

Generic Address Translation: the MMU

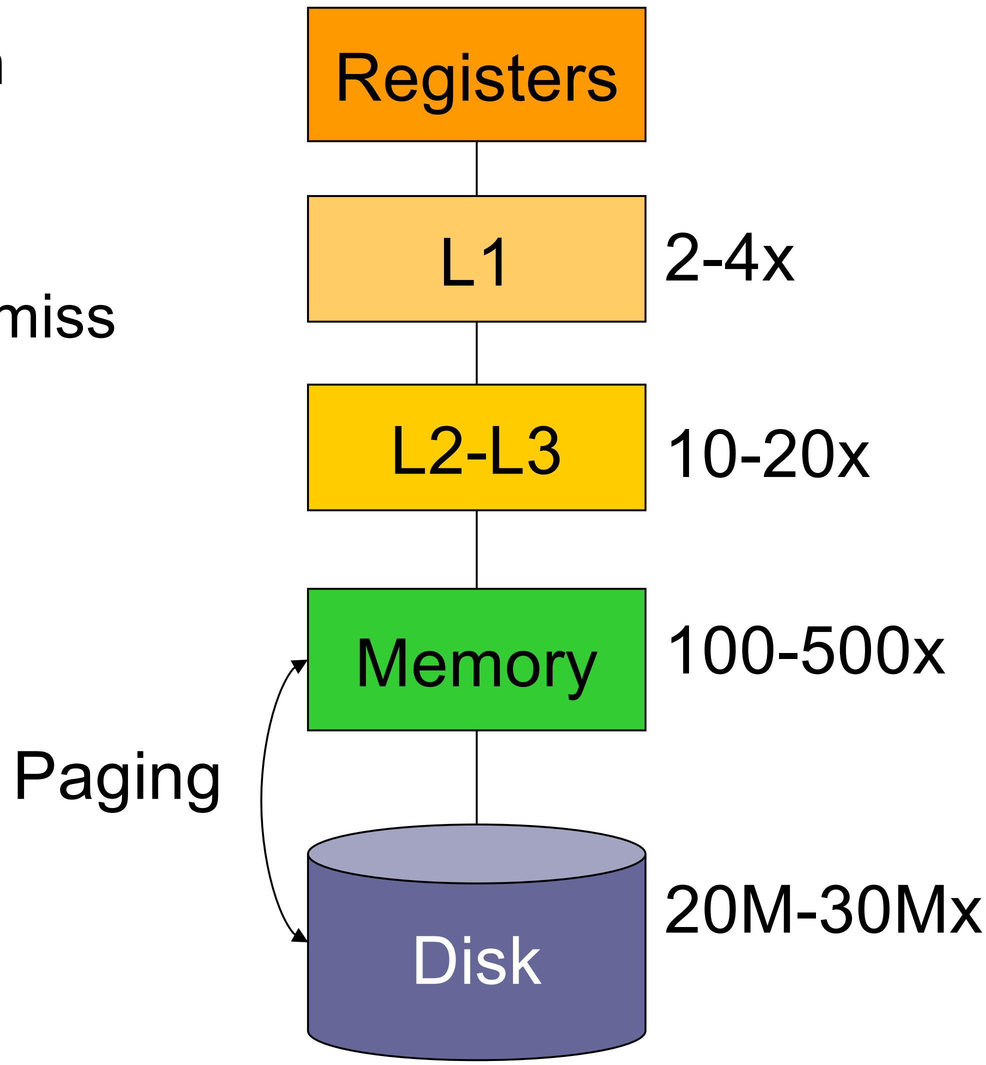
- CPU view
  - Virtual addresses
  - Each process has its own memory space [0, high] – virtual address space
- Memory or I/O device view
  - Physical addresses
  - Fragmented, changing
- Memory Management Unit (MMU) translates virtual address into physical address for each load and store
- Combination of hardware and (privileged) software controls the translation, and relocation



Where to Keep Translation Information?

Goals of translation

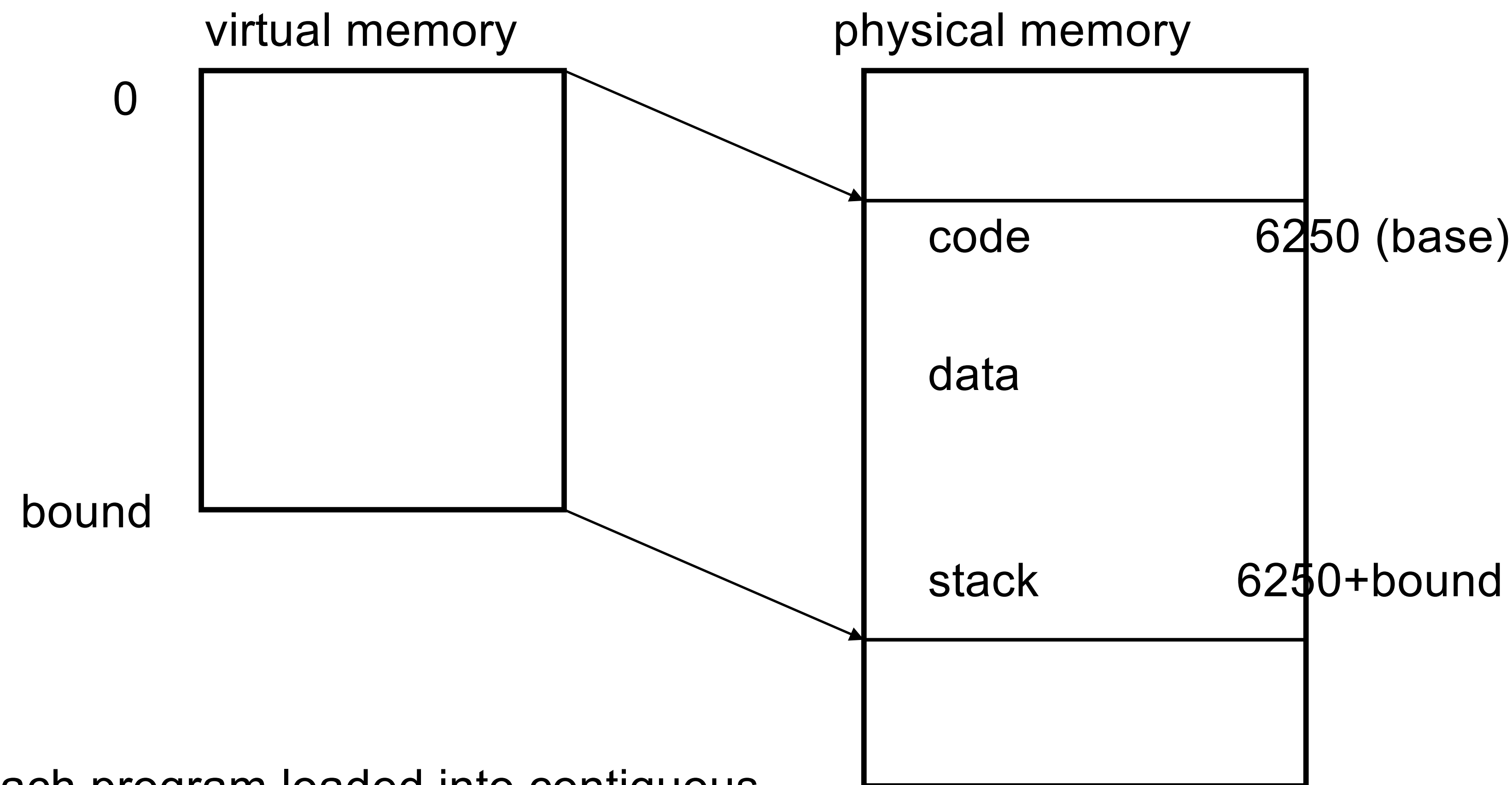
- Implicit translation for each memory reference
- A hit should be very fast
- Trigger an exception on a miss
- Protect from user's errors



Address Translation Methods

- Base and Bound
- Segmentation
- Paging
- Multilevel translation
- Inverted page tables

Base and Bound

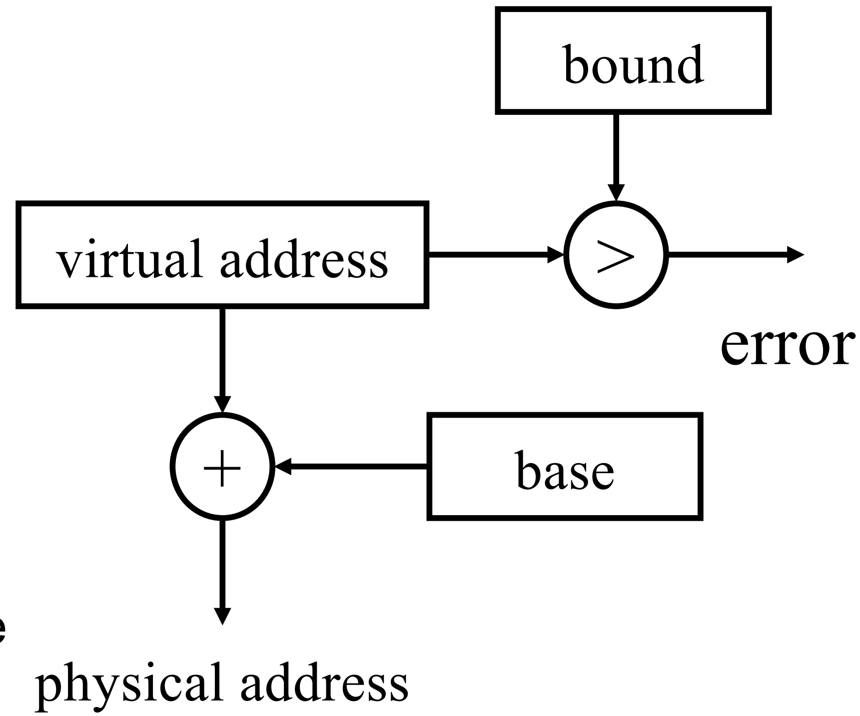


Each program loaded into contiguous regions of physical memory.

Example on next slide

Base and Bound (or Limit) Example: Cray-I

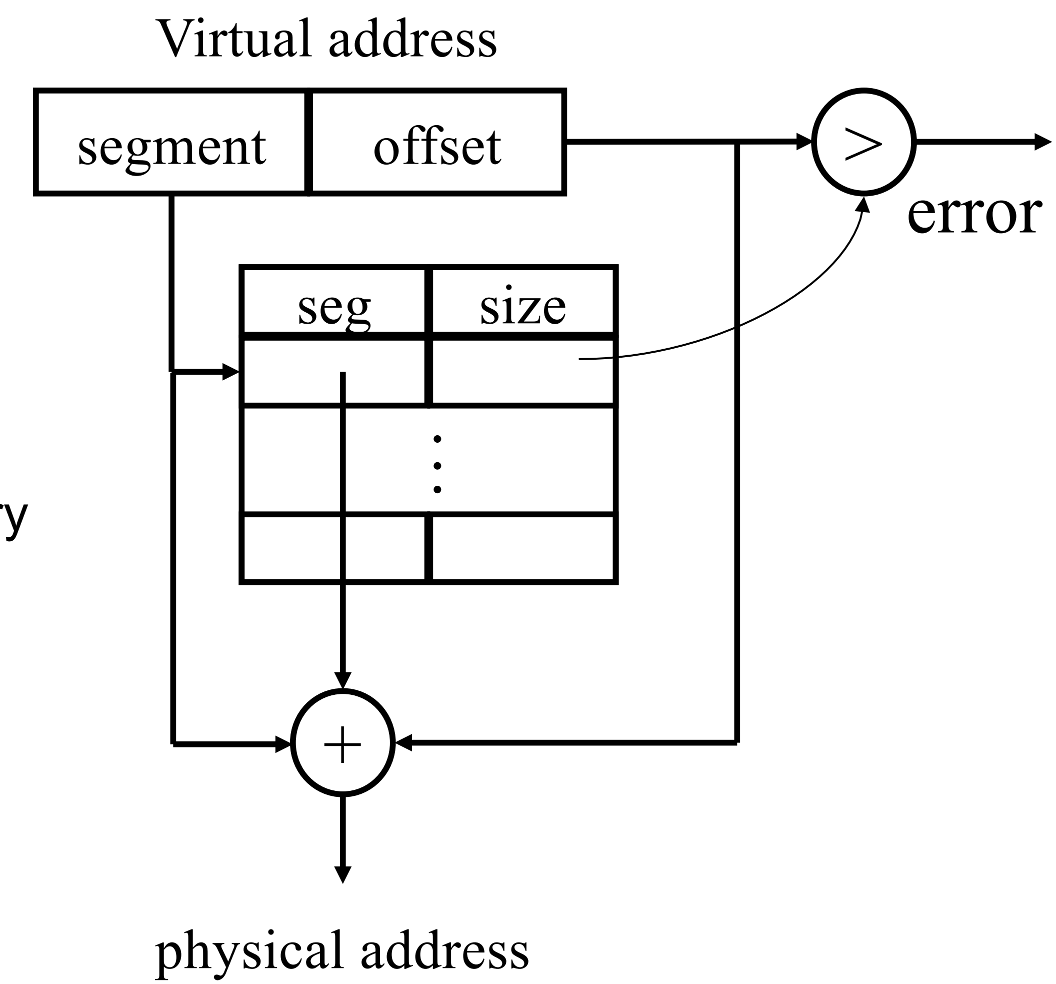
- Protection
  - A process can only access physical memory in [base, base+bound]
- On a context switch
  - Save/restore base, bound regs
- Pros
  - Simple
  - Inexpensive (Hardware cost: 2 registers, adder, comparator)
- Cons
  - Can’t fit all processes in memory, have to swap
  - Fragmentation in memory
  - Relocate processes when they grow?
  - Compare and add on every instruction
  - Very coarse grained



Why not have multiple contiguous segments for each process, and keep their base/bound data in hardware?

Segmentation

- Every process has table of (seg, size) for its segments
- Treats (seg, size) as a finer-grained (base, bound)
- Protection
  - Every entry contains access rights
- On a context switch
  - Save/restore table in kernel memory

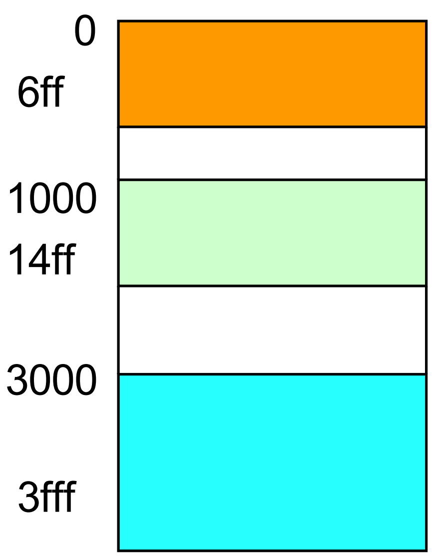


Segmentation Example

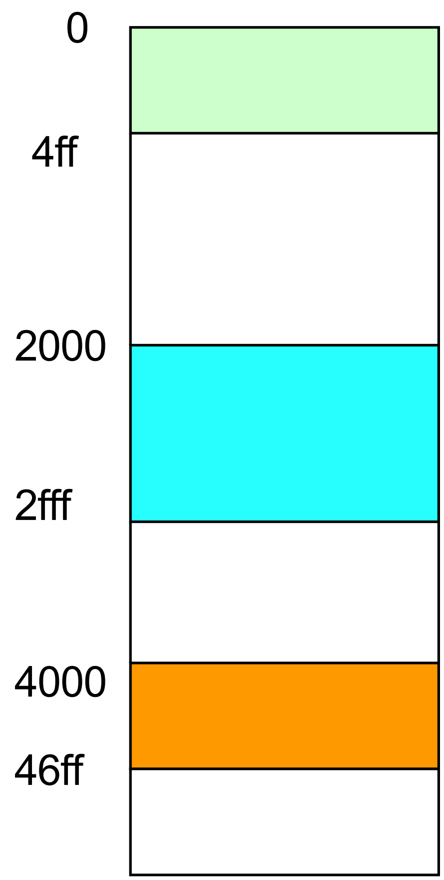
(assume 2 bit segment ID, 12 bit segment offset)

v-segment #	p-segment start	segment size
code (00)	0x4000	0x700
data (01)	0	0x500
- (10)	0	0
stack (11)	0x2000	0x1000

virtual memory



physical memory



Segmentation Example (Cont'd)

Virtual memory for `strlen(x)`	
Main: 240	store 1108, r2
244	store pc+8, r31
248	jump 360
24c	
…	
strlen: 360	loadbyte (r2), r3
…	
420	jump (r31)
…	
x: 1108	a b c \0
…	

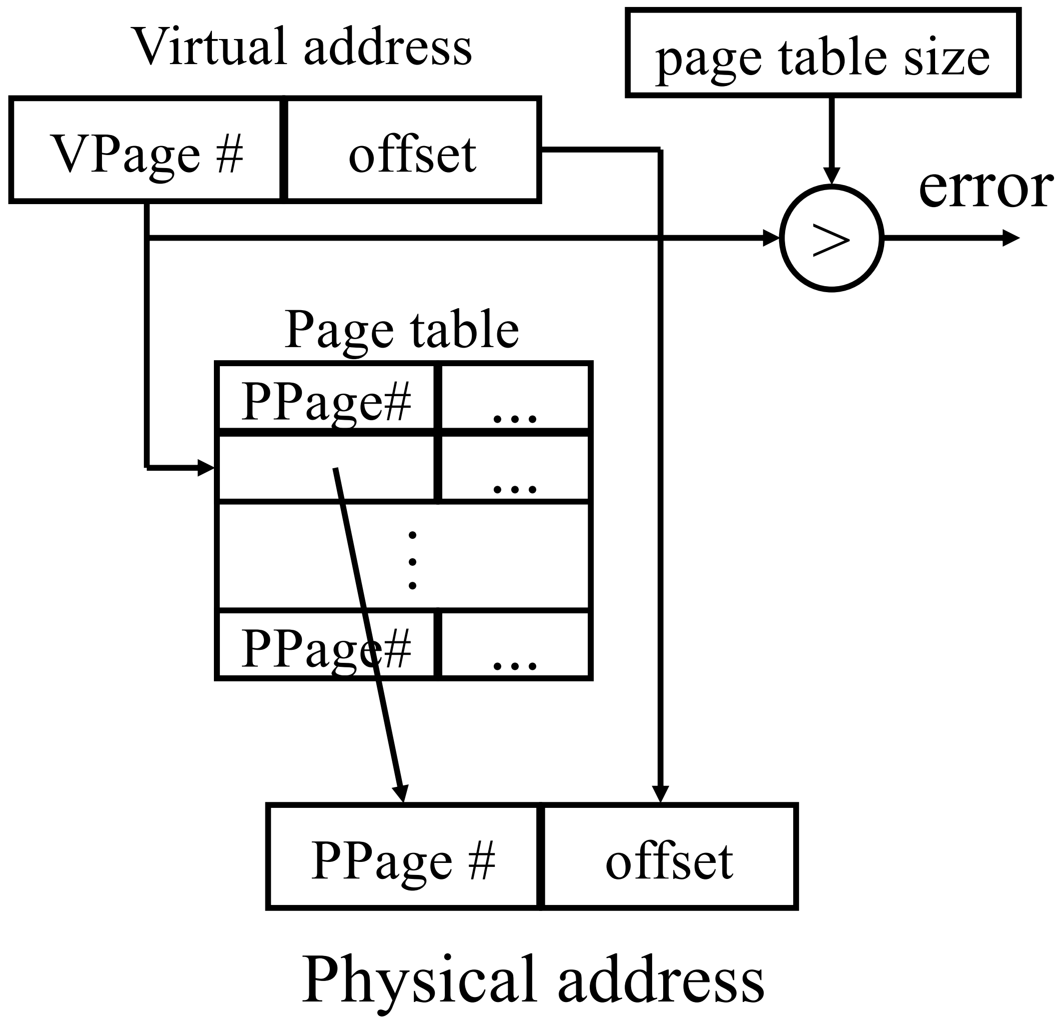
physical memory for `strlen(x)`	
x: 108	a b c \0
…	
Main: 4240	store 108, r2
4244	store pc+8, r31
4248	jump 4360
424c	
…	
strlen: 4360	loadbyte (r2), r3
…	
4420	jump (r31)
…	

Segmentation

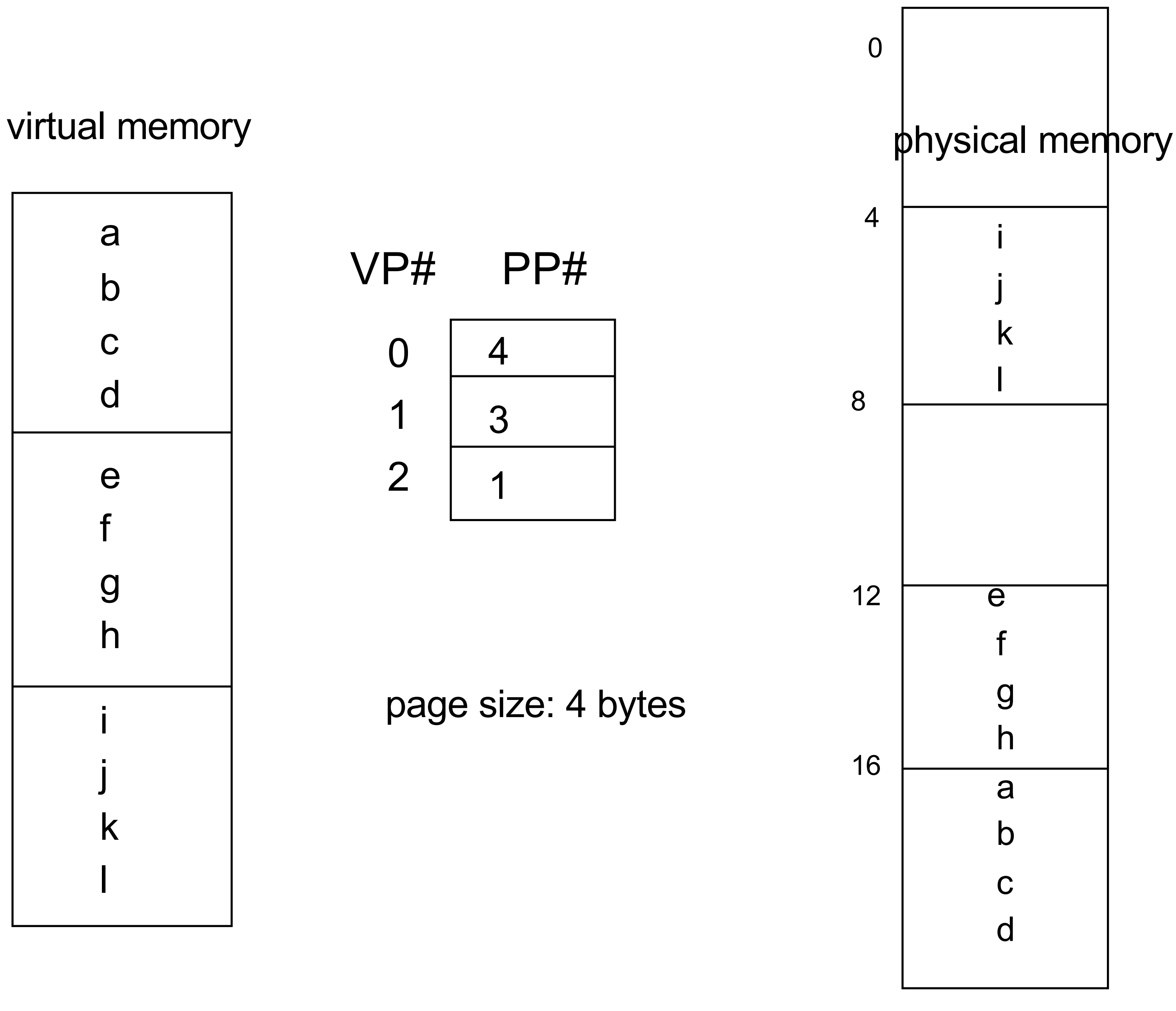
- Pros
  - Provides logical protection: programmer "knows program" and therefore how to design and manage segments
  - Therefore efficient
  - Easy to share data
- Q: Cons?

Paging

- Use a fixed size unit called page instead of segment
- Use page table to translate
- Various bits in each entry
- Context switch
  - Similar to segmentation
- What should page size be?



Paging example



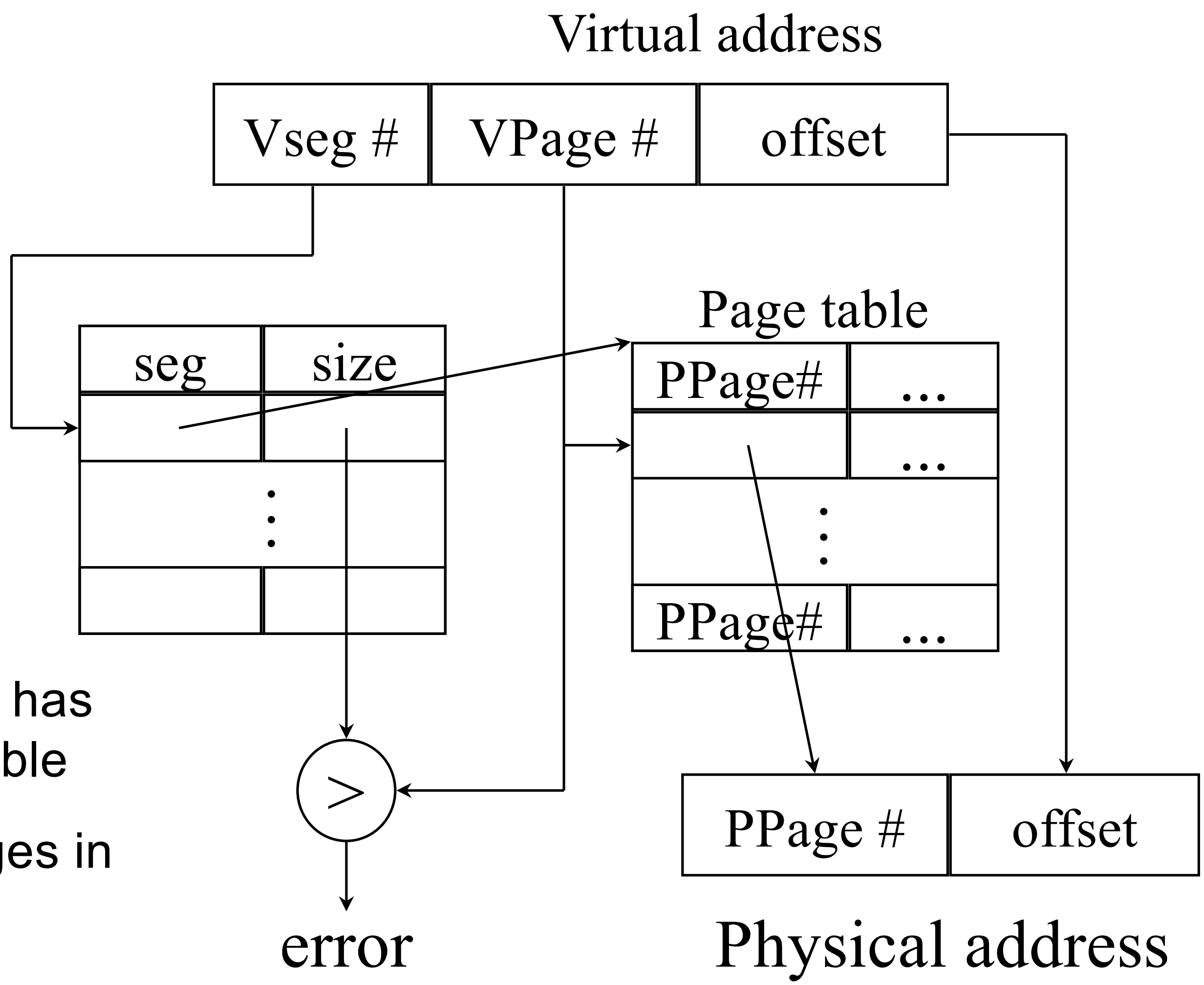
How Many PTEs Do We Need?

- Assume 4KB page
  - Needs "low order" 12 bits to address byte within page
- Worst case for 32-bit address machine
  - 20 bits for virtual page no., so 2^{20} PTEs for a process
  - # of processes $\times$ 2^{20}
  - 2^{20} PTEs per page table (~4Mbytes), but there might be 10K processes. They won't even fit in memory together
- What about 64-bit address machine?
  - # of processes $\times$ 2^{52}
  - A page table cannot fit in a disk (2^{52} PTEs = 16PBytes)!

Paging

- Pros
  - Simple allocation
  - Easy to share
  - Hardware likes fixed sizes (in fact, powers of two)
- Cons
  - Big table
  - PTEs even for big holes in memory

Segmentation with Paging

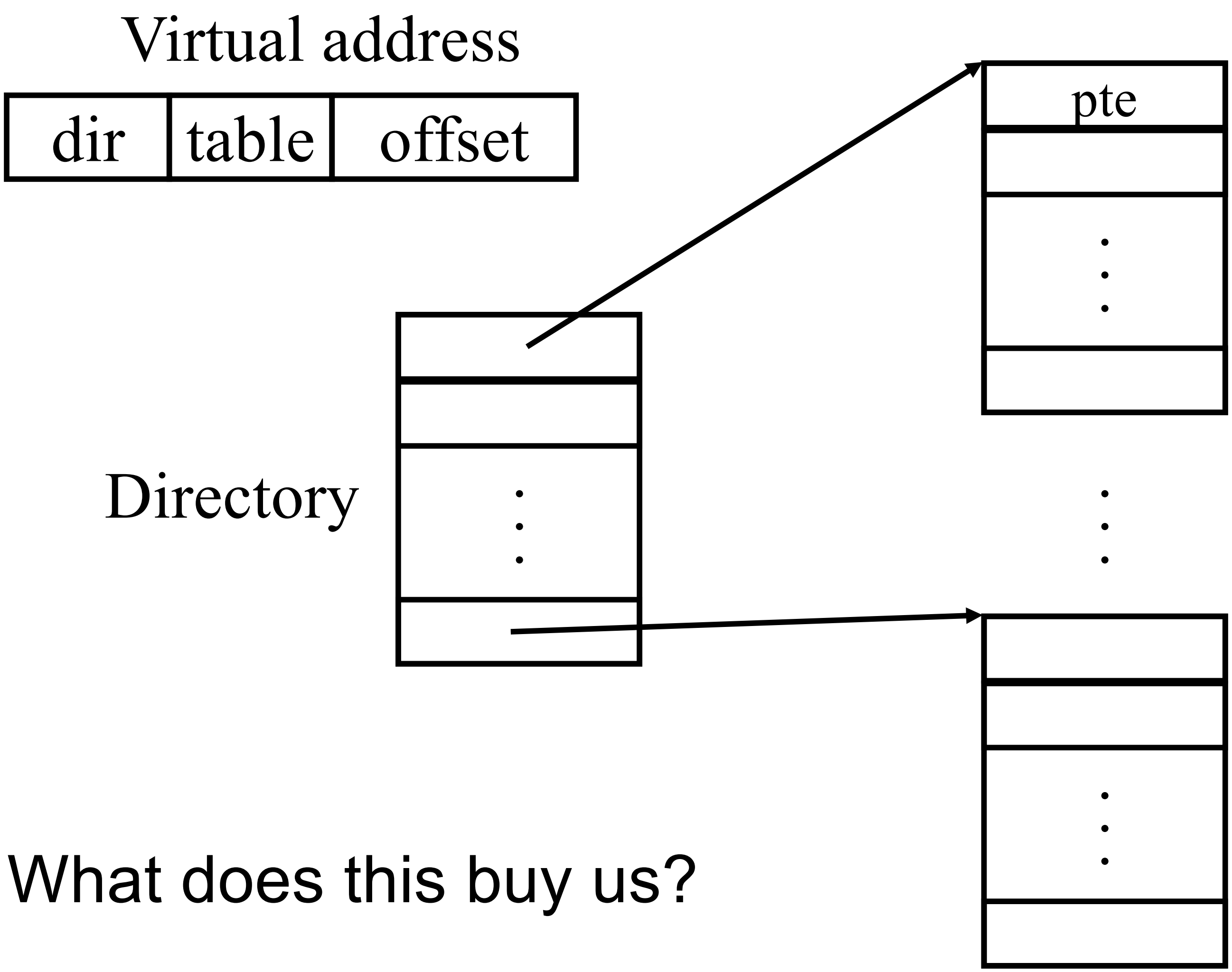


Every segment has its own page table

Size is # of pages in that segment

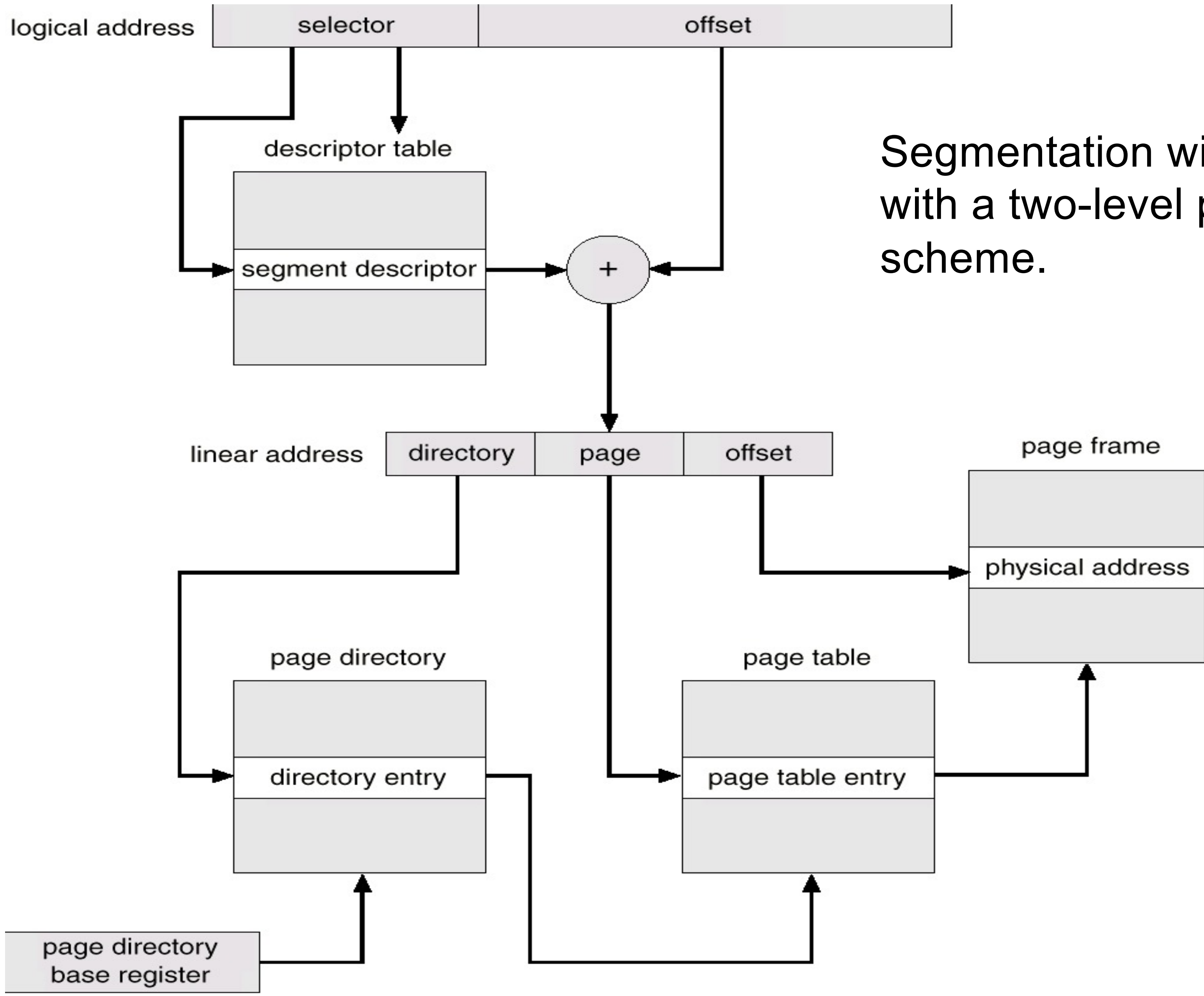
Benefit?

Multiple-Level Page Tables



What does this buy us?

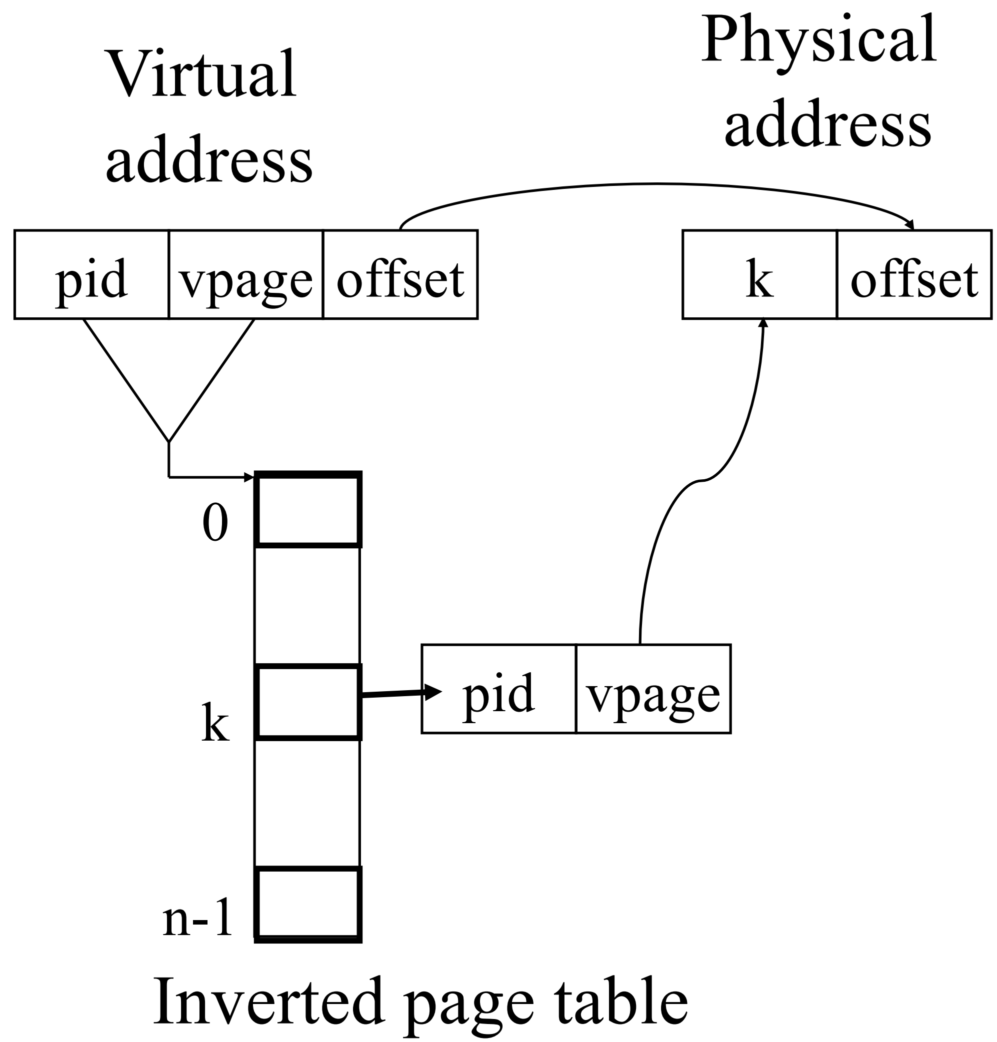
Segmentation with 2-level Paging (30386)



Segmentation with paging. with a two-level paging scheme.

Inverted Page Tables

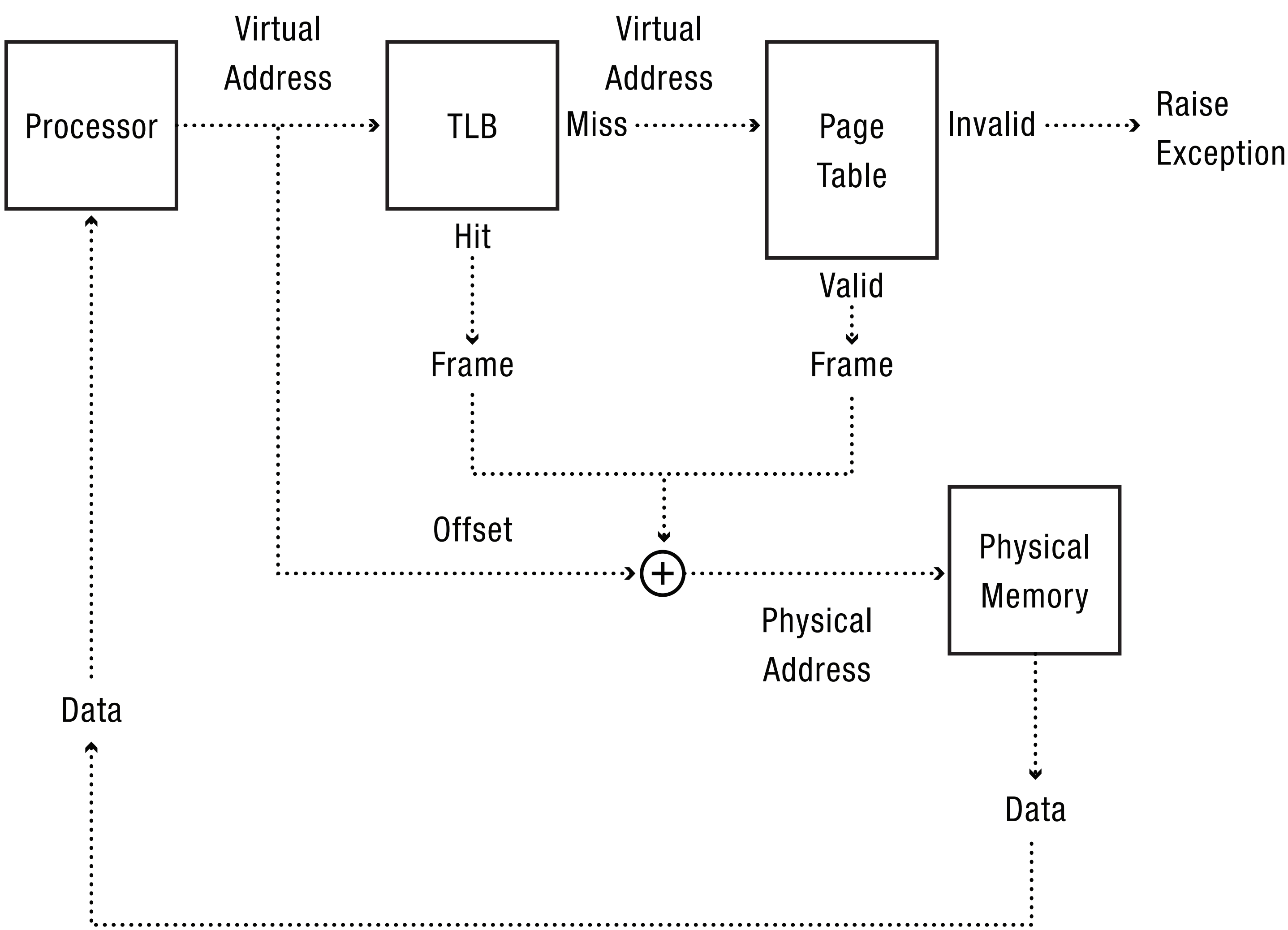
- Main idea
  - One PTE for each physical page frame
  - Hash (Vpage, pid) to Ppage#
- Pros
  - Small page table for large address space
- Cons
  - Lookup is difficult
  - Overhead of managing hash table, etc



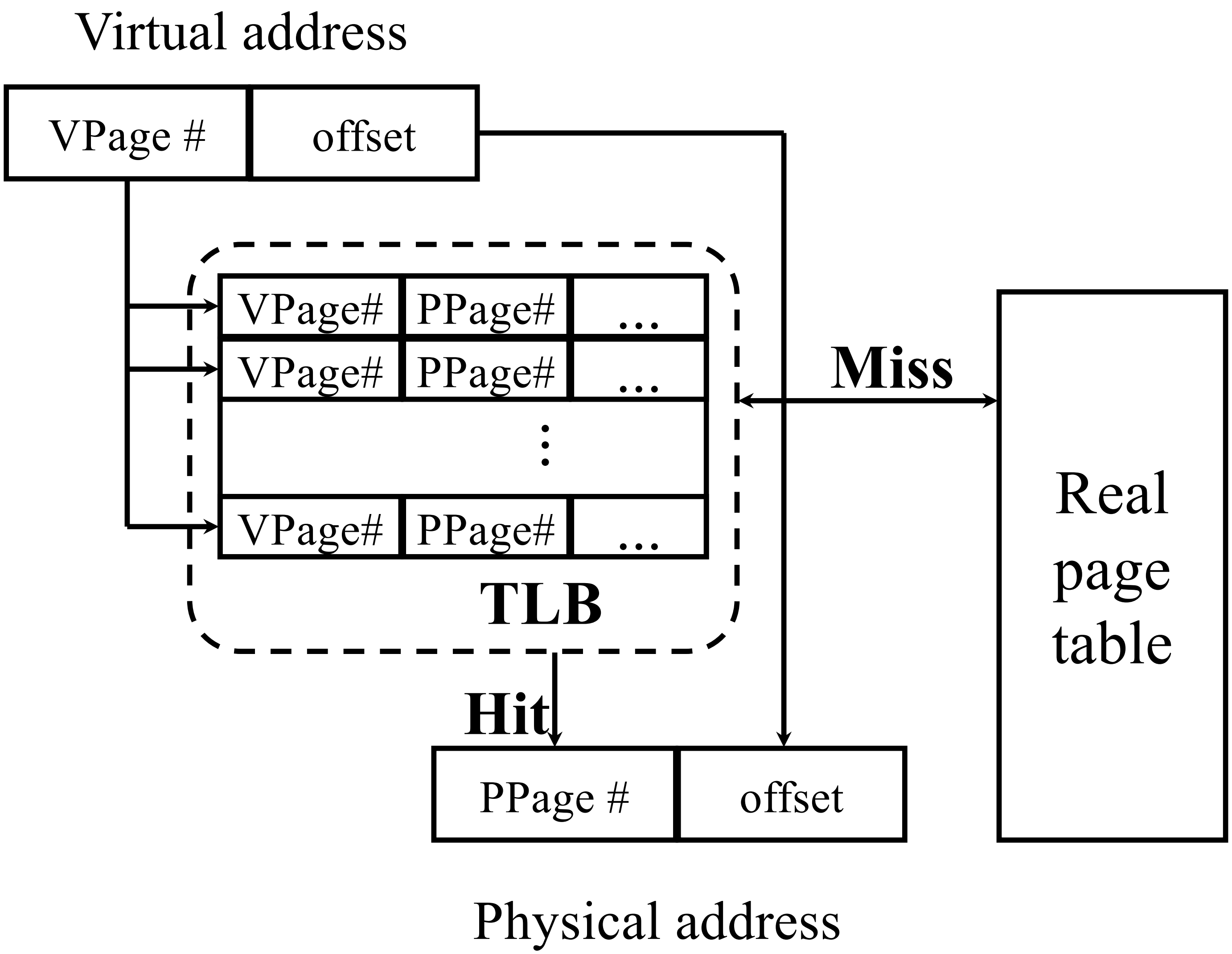
Making Translation Lookups Faster: TLBs

- Programs only know virtual addresses
  - Every program or process starts from 0 to high address
- Every virtual address must be translated
  - May involve walking through a hierarchical page table
  - Since page table is in memory, a program memory access may require several actual memory accesses
- Solution
  - Cache recent virtual to physical translations, i.e. "active" part of page table, in a very fast memory
  - If virtual address hits in TLB, use cached translation
  - Typically fully associative cache, match against entries

TLB and Page Table Translation



What's in the TLB?



Bits in a TLB Entry

- Common (necessary) bits
  - Virtual page number
  - Physical page number: translated address
  - Valid bit
  - Access bits: kernel and user (none, read, write)
- Optional (useful) bits
  - Process tag
  - Reference bit
  - Modify bit
  - Cacheable bit

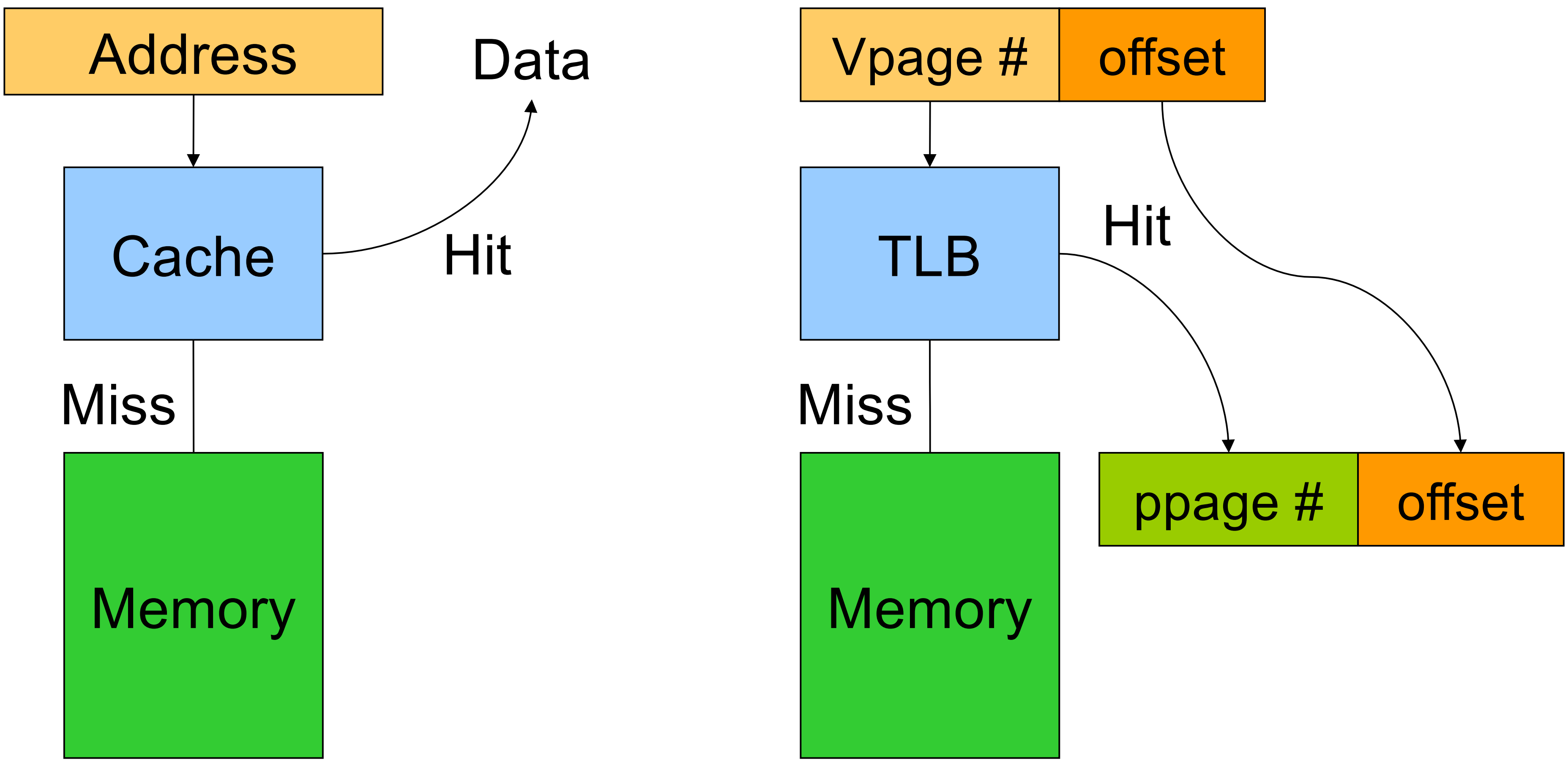
Hardware-Controlled TLB

- On a TLB hit, hardware checks the valid bit
  - If valid, pointer to page frame in memory
  - If invalid, the hardware generates a page fault
    - Perform page fault handling
    - Restart the faulting instruction
- On a TLB miss
  - HW checks if page containing the PTE is valid (in memory), and if so loads the PTE into the TLB
    - Write back and replace a TLB entry if there is no free entry
  - If the page containing the PTE is invalid, or if there is a protection fault, generate a fault
  - VM software performs fault handling
  - Restart the CPU

Software-Controlled TLB

- On TLB hit, same as in hardware-controlled TLB
- On a miss in TLB, software is invoked
  - Write back if there is no free entry
  - Check if the page containing the PTE is in memory
  - If not, perform page fault handling
  - Load the PTE into the TLB
  - Restart the faulting instruction

Hardware Cache vs TLB



- Similarities
  - Cache a portion of memory
  - Write back on a miss
- Differences
  - Associativity
  - Consistency

TLB Related Issues

- What TLB entry to replace?
  - Random
  - Pseudo LRU
- What happens on a context switch?
  - Process tag: invalidate appropriate TLB entries
  - No process tag: Invalidate the entire TLB contents
- What happens when changing a page table entry?
  - Change the entry in memory
  - Invalidate the TLB entry

Consistency Issues

- "Snoopy" cache protocols (hardware)
  - Maintain consistency with DRAM, even when DMA happens
- Consistency between DRAM and TLBs (software)
  - You need to flush related TLBs whenever changing a page table entry in memory
- Consistency across processors in multiprocessor
  - Q: What happens when a processor changes a PTE?

Summary: Virtual Memory

- Virtual Memory
  - Virtualization makes software development easier and enables memory resource utilization better
  - Separate address spaces provide protection and isolate faults
- Address Translation
  - Translate every memory operation using table (page table, segment table).
  - Speed: cache frequently used translations
- Result
  - Every process has a private address space
  - Programs run independently of actual physical memory addresses used, and actual memory size
  - Protection: processes only access memory they are allowed to