COS 318: Operating Systems

Virtual Memory and Address Translation

Prof. Margaret Martonosi
Computer Science Department
Princeton University

http://www.cs.princeton.edu/courses/archive/fall11/cos318/

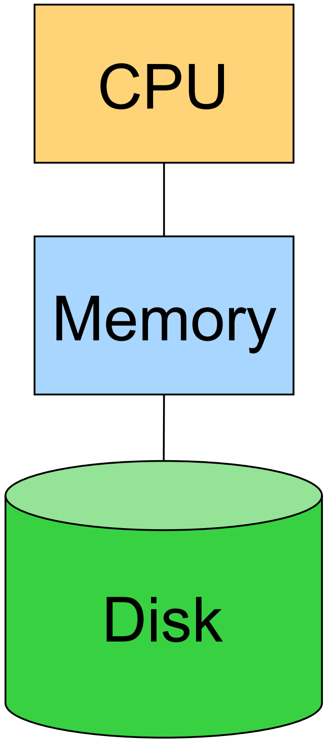
Today’s Topics

- Virtual Memory
  - Virtualization
  - Protection
- Address Translation
  - Base and bound
  - Segmentation
  - Paging
  - Translation look-ahead buffer

The Big Picture

- DRAM is fast, but relatively expensive
- Disk is ~100X cheaper, but slow

- Virtual Memory can bridge this gap.
- Furthermore, VM can help with isolation between processes, portability abstractions regarding the amount of memory in the system, etc.
- Our goals
  - Run programs as efficiently as possible
  - Make the system as safe as possible

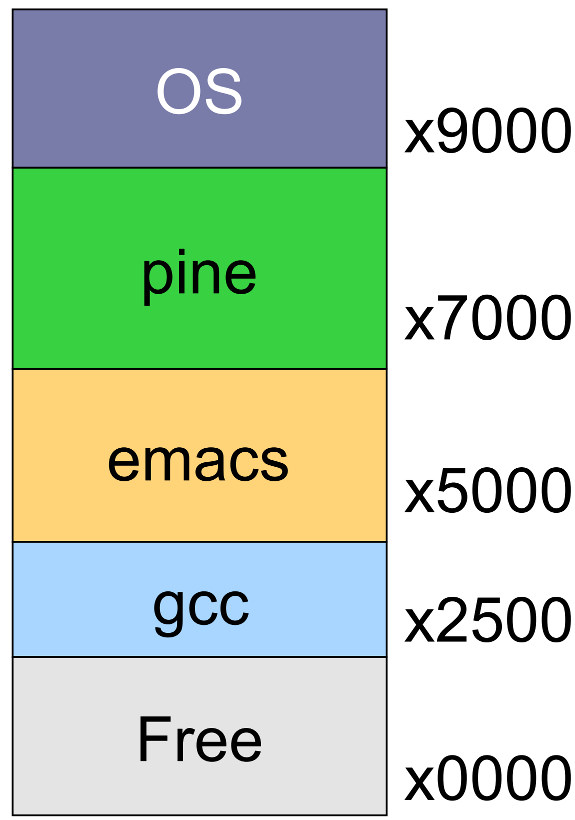


Issues

- Many processes
  - The more processes a system can handle, the better
- Address space size
  - Many small processes whose total size may exceed memory
  - Even one process may exceed the physical memory size
- Portability
  - Once I write a program, I want it to run on many platforms of the same ISA family. I don't want to need to know how much DRAM you have installed in order to compile/run it.
- Protection
  - A user process should not crash the system
  - A user process should not do bad things to other processes

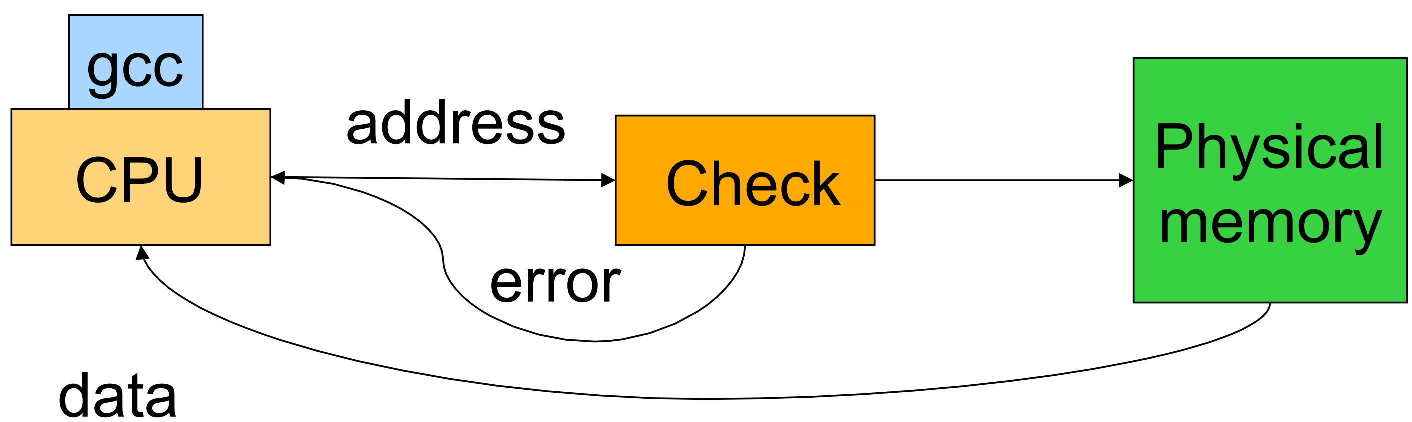
Consider A Simple System

- Only physical memory
  - Applications use physical memory directly
- Run three processes
  - emacs, pine, gcc
- What if
  - gcc has an address error?
  - emacs writes at x7050?
  - pine needs to expand?
  - emacs needs more memory than is on the machine?



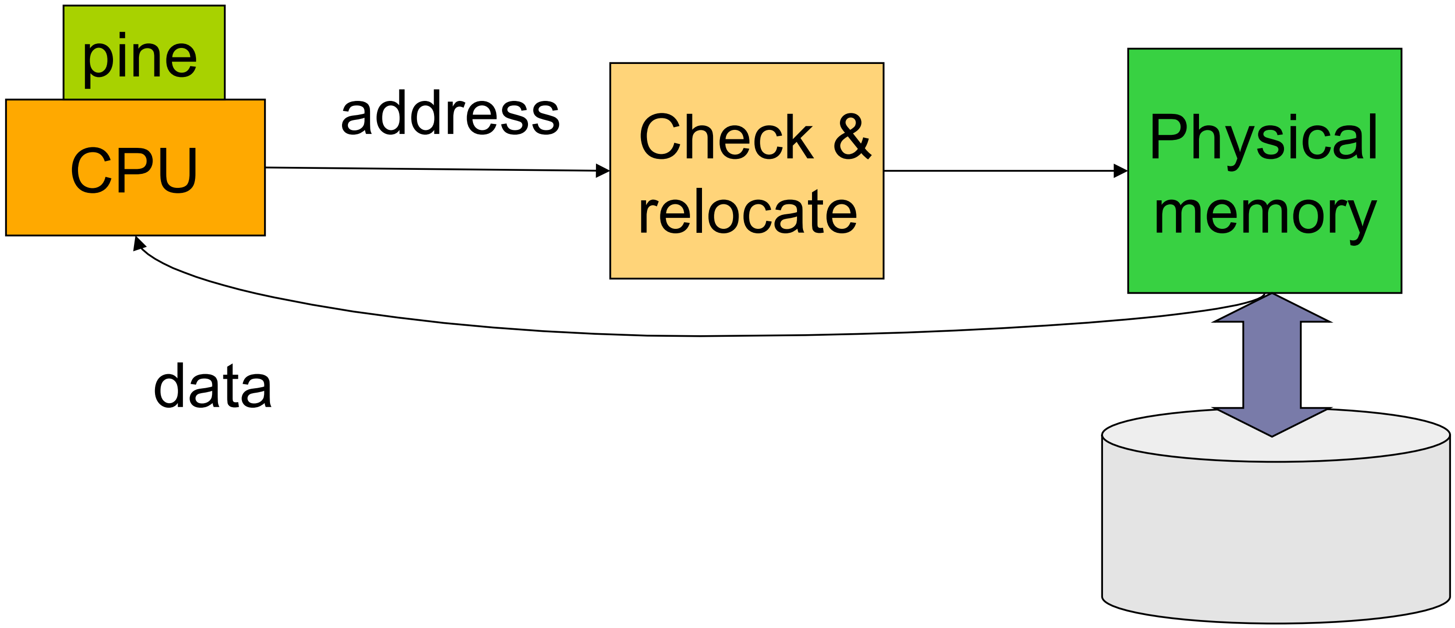
Protection Issue

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



Expansion or Transparency Issue

- A process should be able to run regardless of its physical location or the physical memory size
- Give each process a large, static "fake" address space
- As a process runs, relocate each load and store to its actual memory



Virtual Memory

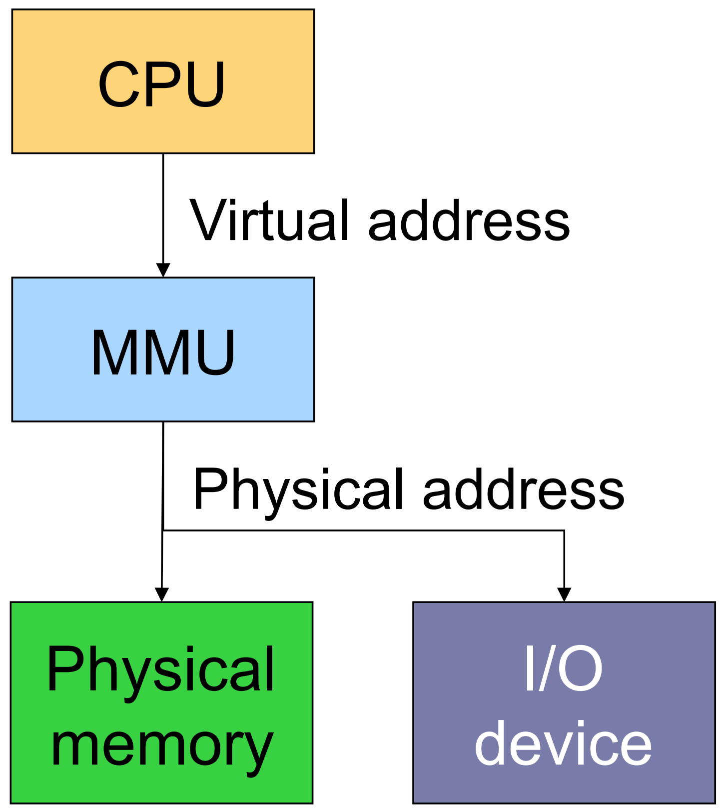
"Any problem in computer science can be solved with another layer of indirection"

- David Wheeler. (World's first PhD in CS. Worked on EDSAC; co-inventor of the subroutine aka "wheeler jump".)
- Rest of quote: "…But that usually will create another problem.

- "Logical" or "virtual" address (visible to program) is distinct from "physical" address (how you actually access DRAM)
- How to make this efficient?

Generic Address Translation

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

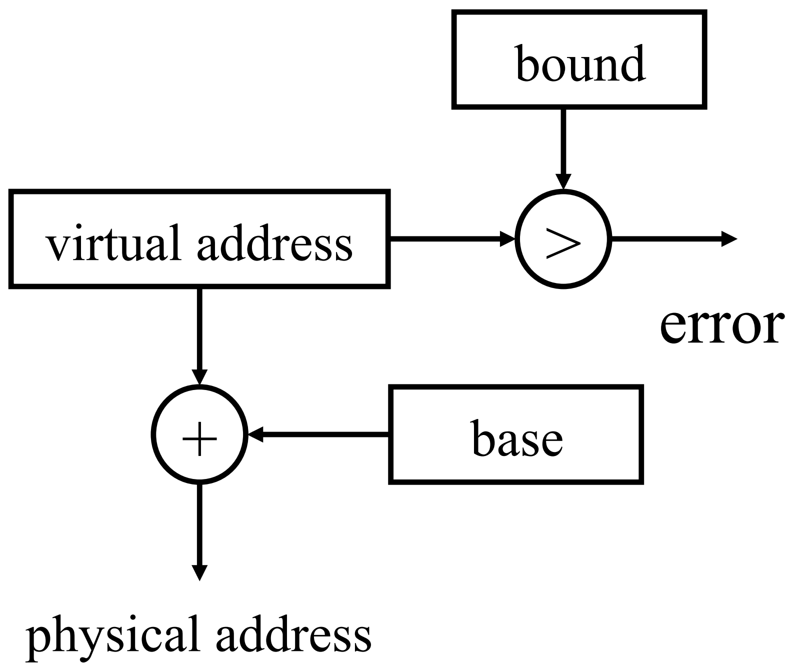


Address Mapping and Granularity

- Must have some “mapping” mechanism
  - Virtual addresses map to
    DRAM physical addresses or disk addresses
- Mapping must have some granularity
  - Granularity determines flexibility
  - Finer granularity requires more mapping information
- Extremes
  - Any byte to any byte: mapping equals program size
  - Map whole segments: larger segments problematic

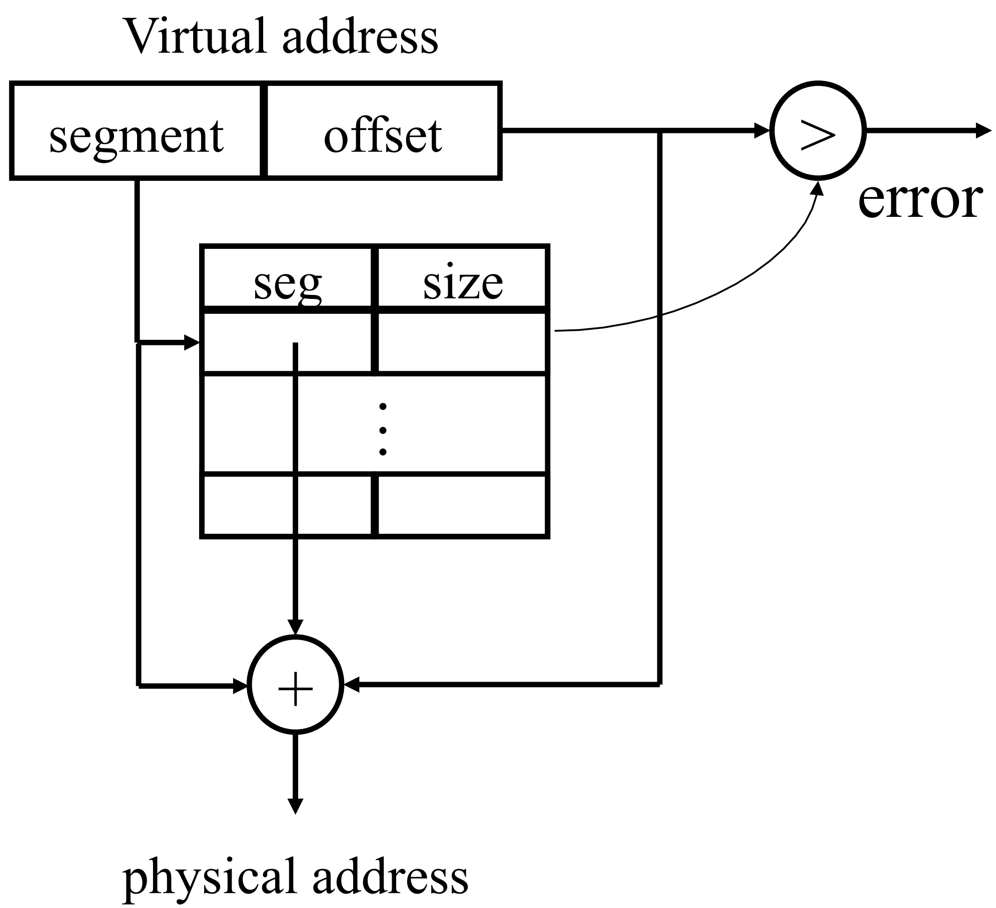
Base and Bound

- Built in Cray-1
- Each process has a pair (base, bound)
- Protection
  - A process can only access physical memory in [base, base+bound]
- On a context switch
  - Save/restore base, bound registers
- Pros
  - Simple
  - Flat and no paging
- Cons
  - Fragmentation
  - Hard to share
  - Difficult to use disks



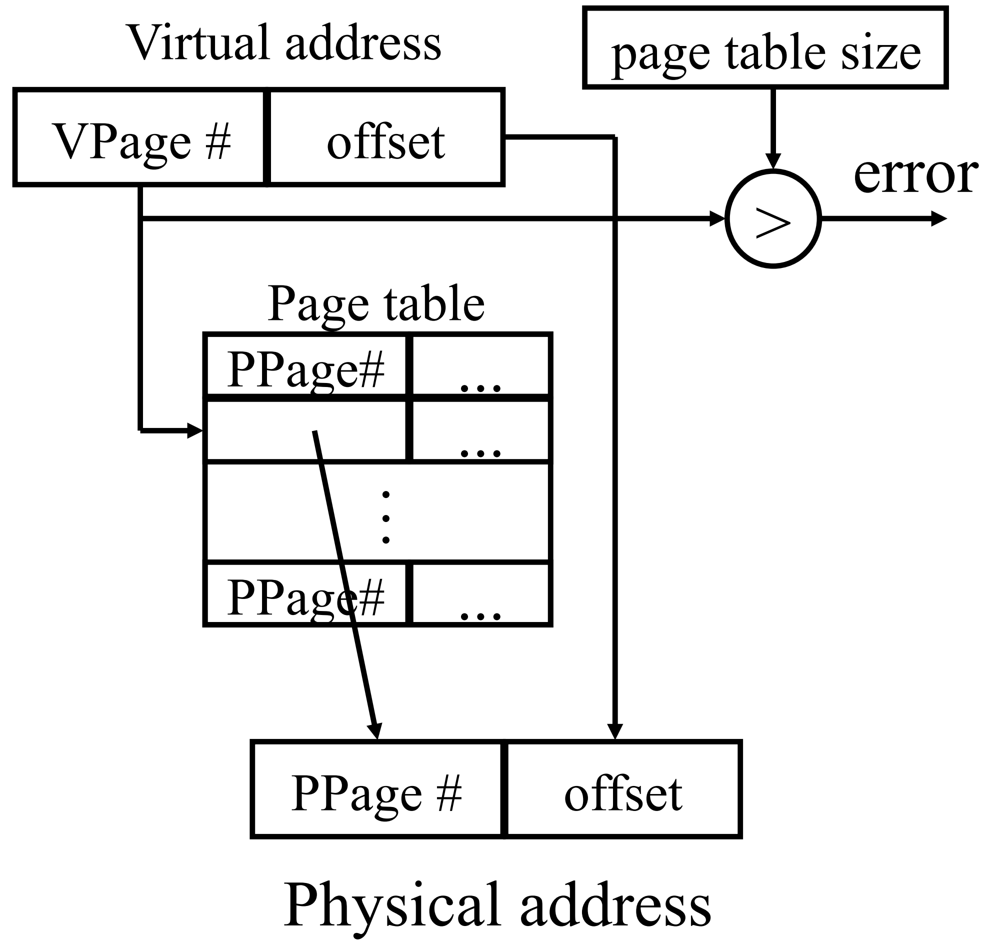
Segmentation

- Each process has a table of (seg, size)
- Treats (seg, size) has a fine-grained (base, bound)
- Protection
  - Each entry has (nil, read, write, exec)
- On a context switch
  - Save/restore the table and a pointer to the table in kernel memory
- Pros
  - Efficient
  - Easy to share
- Cons
  - Complex management
  - Fragmentation within a segment

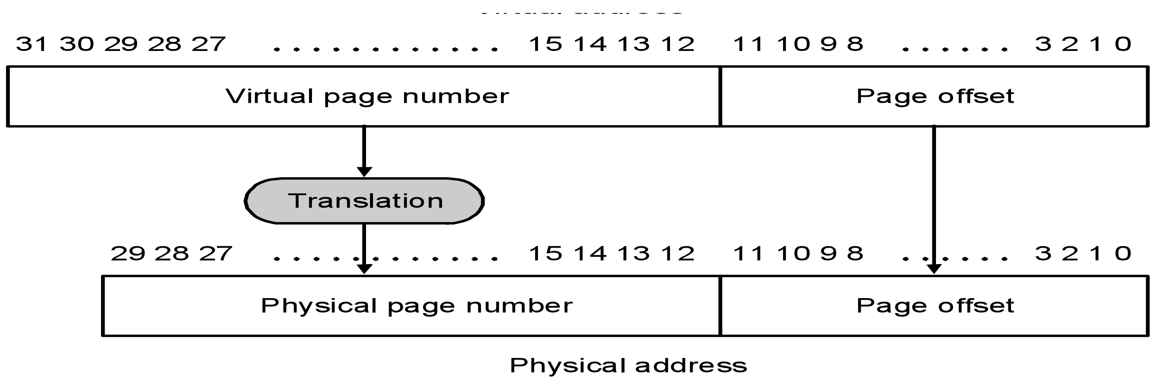


Paging

- Use a fixed size unit called page instead of segment
- Use a page table to translate
- Various bits in each entry
- Context switch
  - Similar to segmentation
- What should page size be?
- Pros
  - Simple allocation
  - Easy to share
- Cons
  - Big table
  - How to deal with holes?



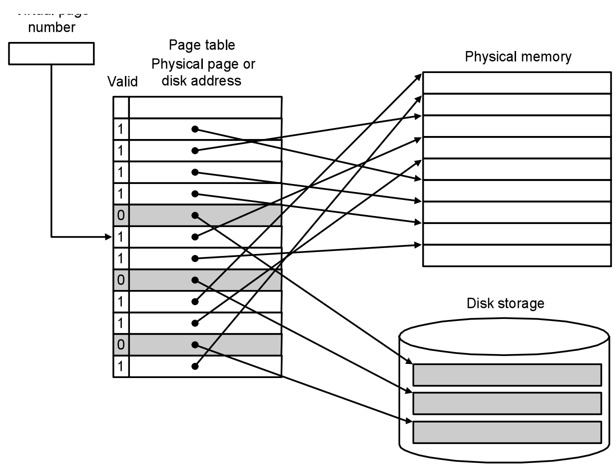
Pages: virtual memory blocks



Virtual Memory: Page Table’s role

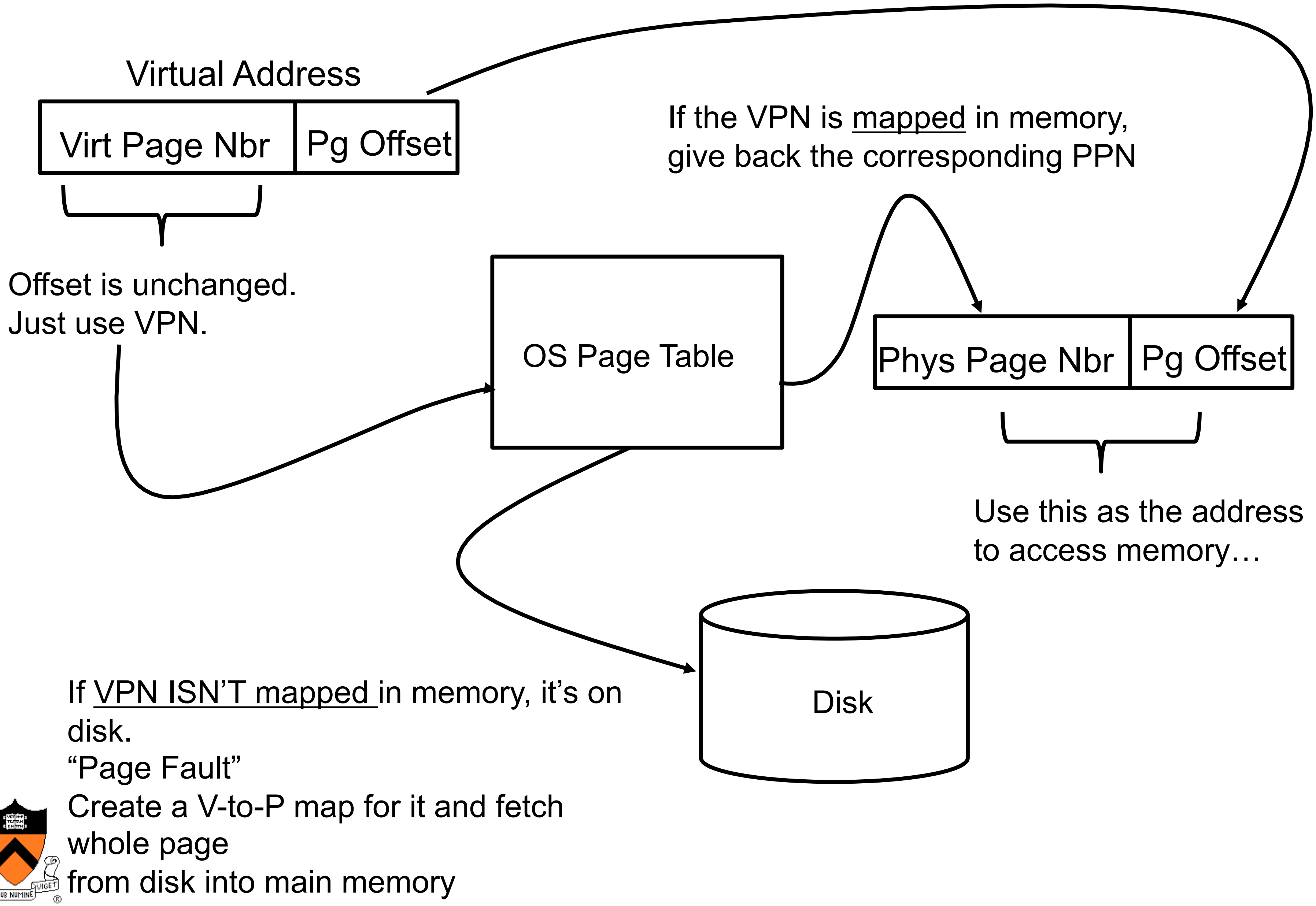
Virtual page number

Physical Memory

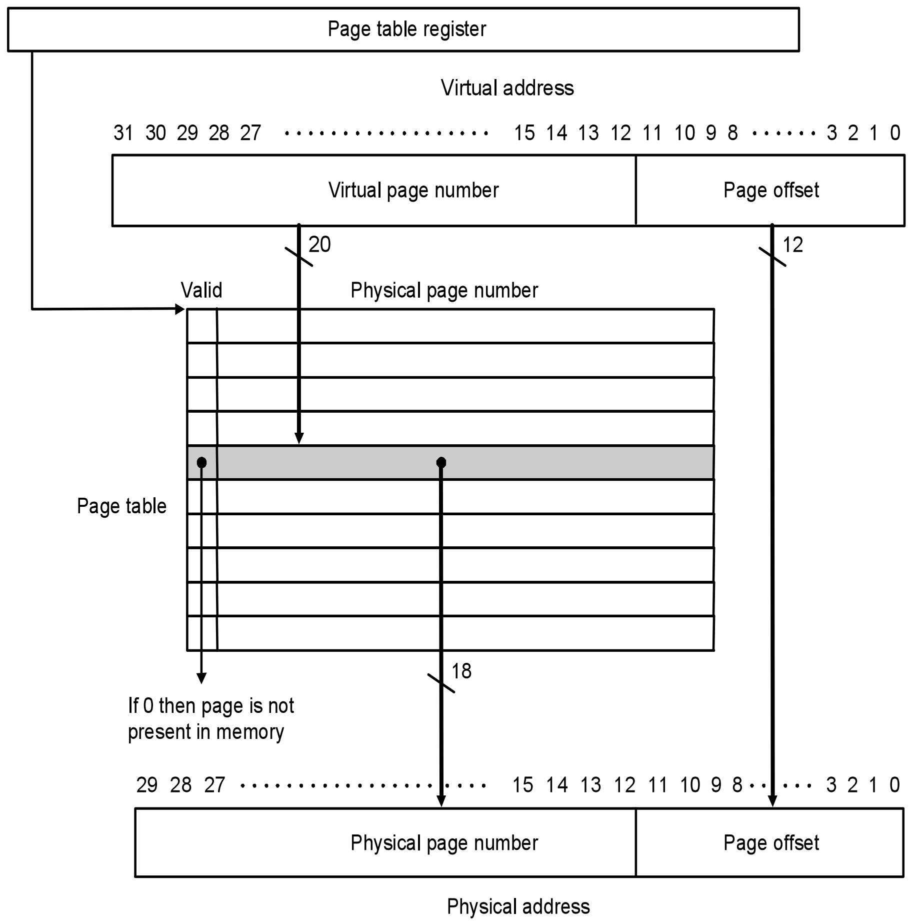


Page Table (OS)

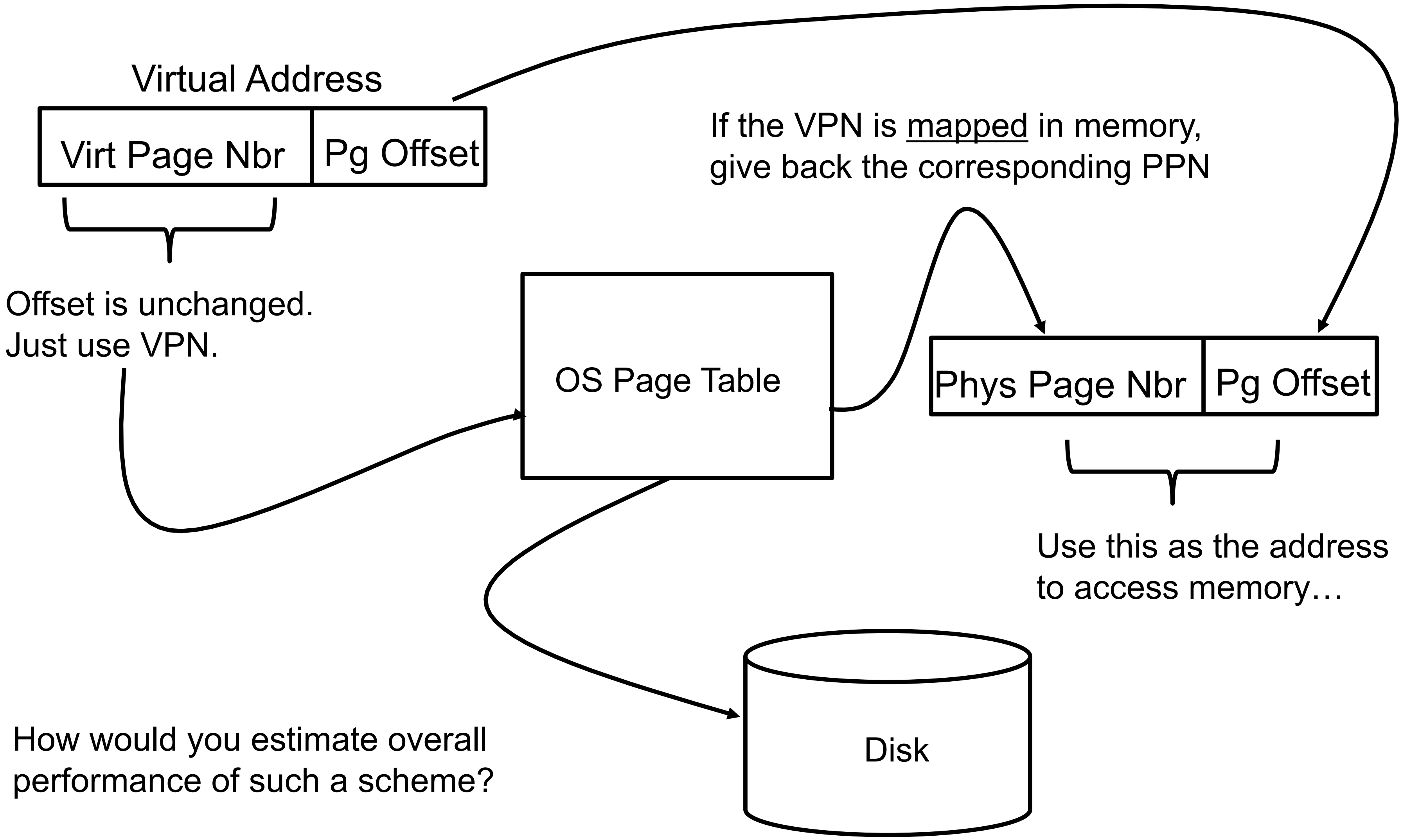
Virtual to Physical Page Translation



Page Table (OS Software)



Efficiency?



How would you estimate overall performance of such a scheme?

Page Faults are a real bummer…

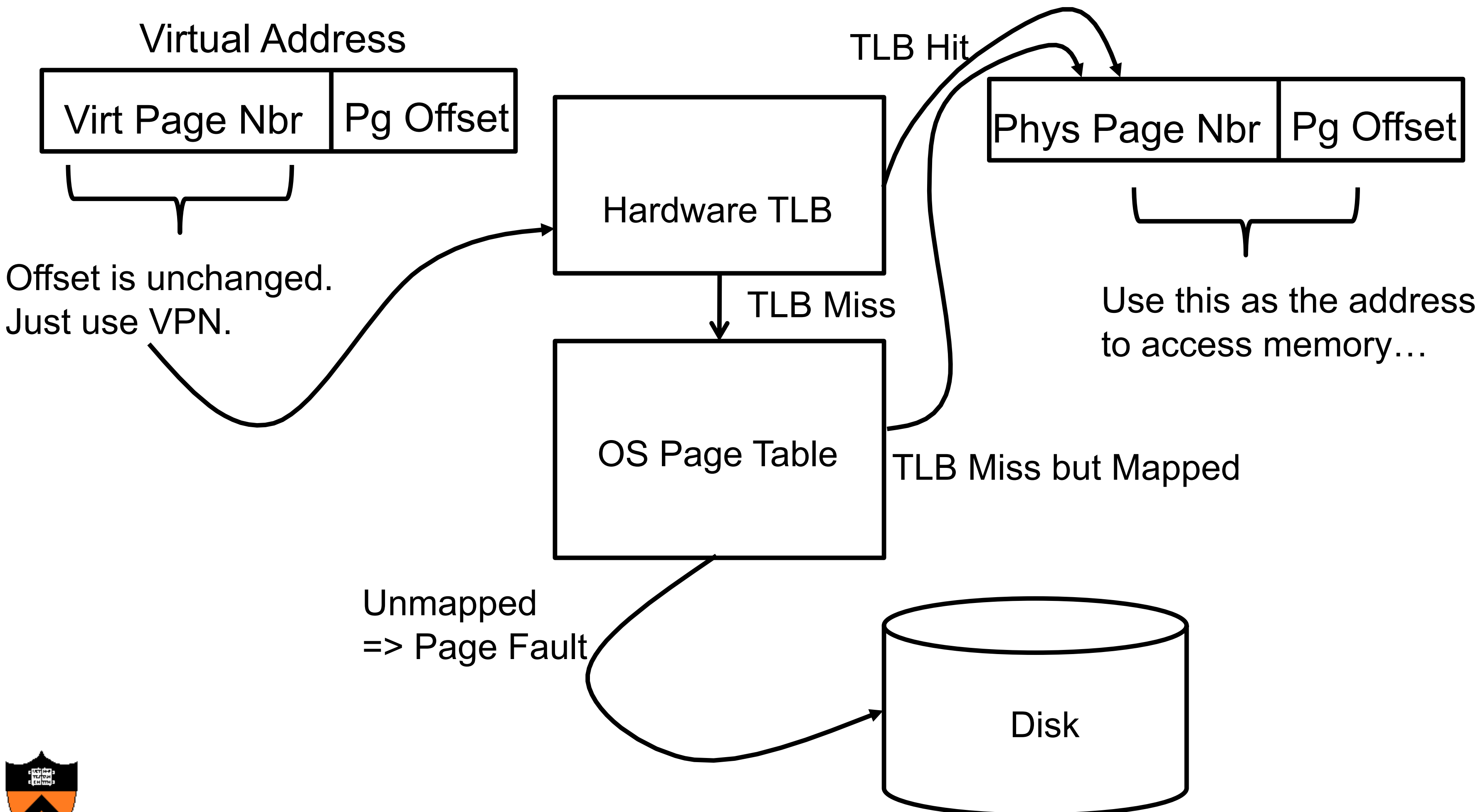
- Page faults: No Page table mapping exists for this page => the data is not in memory => retrieve it from disk
- Huge time penalty: so…
  - pages should be fairly large (e.g., 4-8KB)
  - reducing page faults is important
  - Once memory is "full", each page brought in from disk means another page currently in memory must be unmapped and sent back to disk.
    - how to decide what to evict?

But the PT lookups need to be fast also…

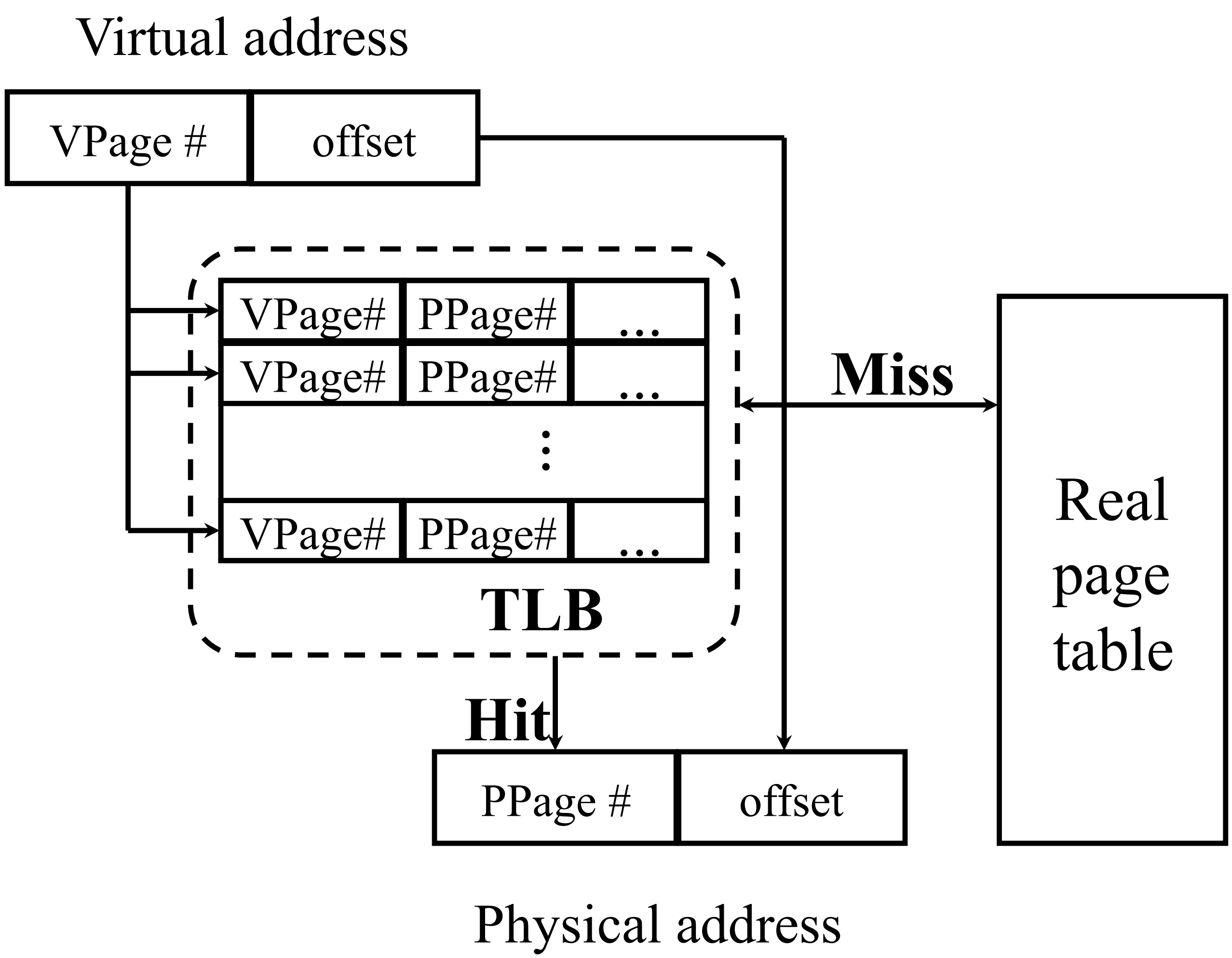
- Even if we DON"T have a page fault, just reading a plain software page table on every reference would be a huge time penalty
  - We need a wait to make the common case (V-to-P mapping is present) as fast as possible.
  - Hardware: Translation Lookaside Buffer
  - HW/SW: Efficient Page Table designs and support to "walk" them fast

Translation Lookaside Buffer (hardware)

- Store most common V->P mappings in hardware table
- Typical size: 100's – 1000's of entries.



Translation Look-aside Buffer (TLB)



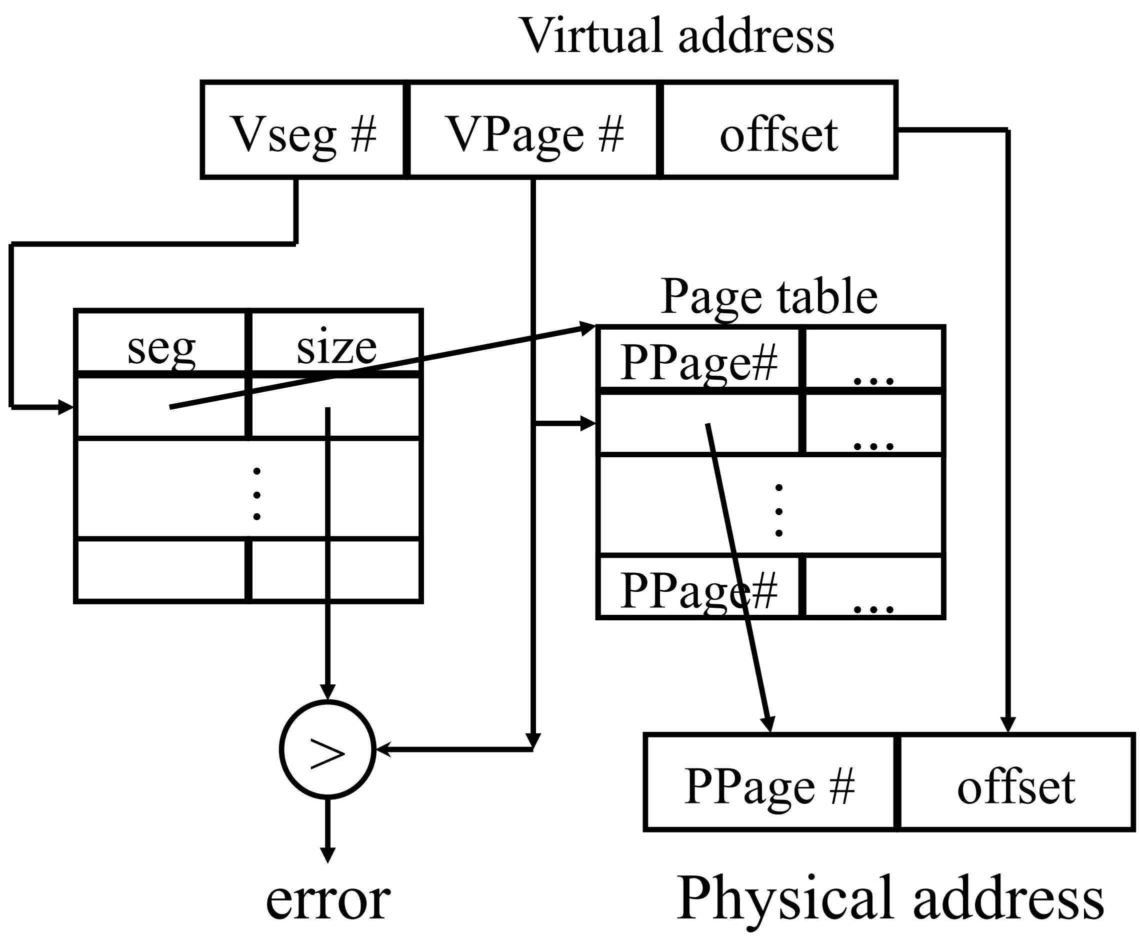
Bits in a TLB Entry

- Common (necessary) bits
  - Virtual page number: match with the virtual address
  - Physical page number: translated address
  - Valid
  - Access bits: kernel and user (nil, read, write)
- Optional (useful) bits
  - Process tag
  - Reference
  - Modify
  - Cacheable

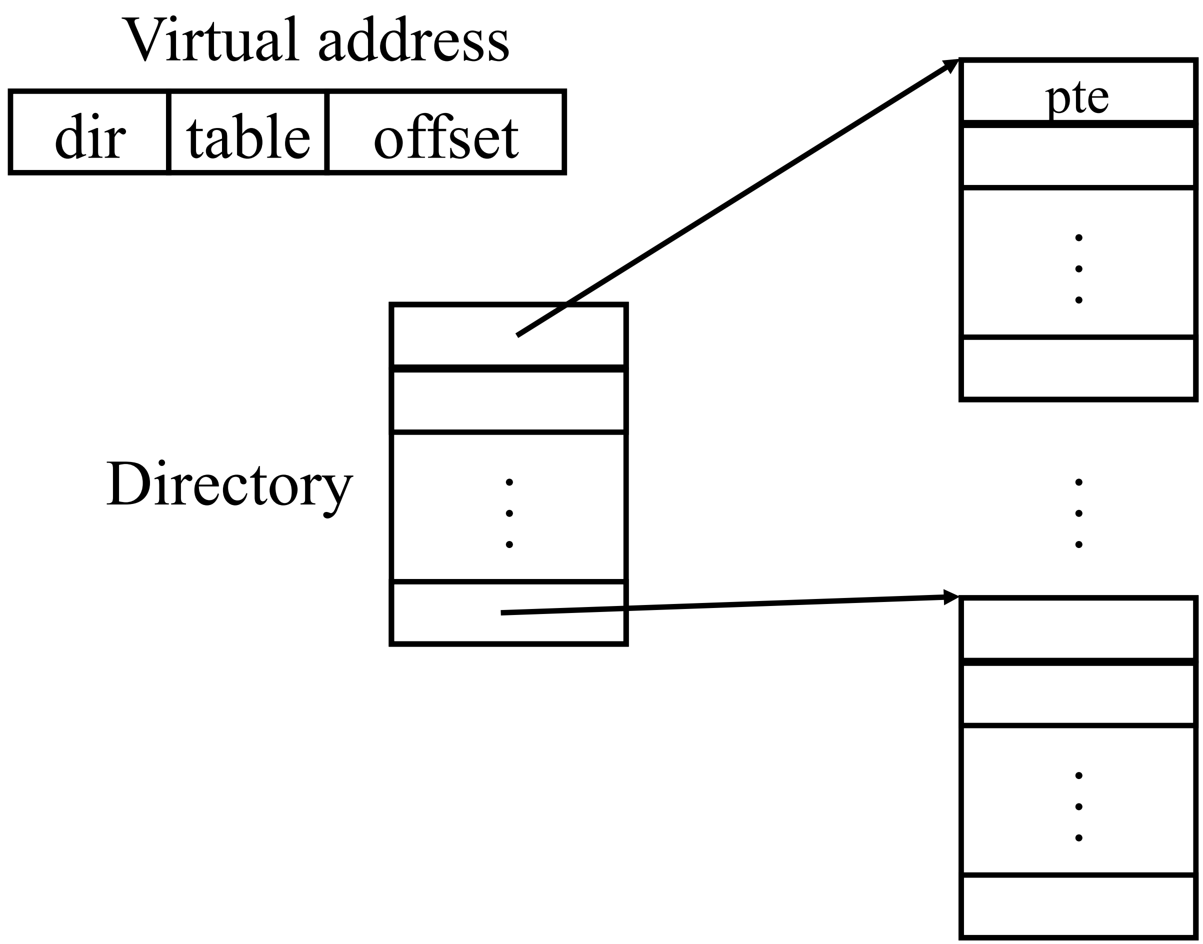
How Many PTEs Do We Need?

- Assume 4KB page
  - Equals "low order" 12 bits
- Worst case for 32-bit address machine
  - # of processes × 2^{20}
  - 2^{20} PTEs per page table (~4Mbytes), but there might be 10K processes. They won't fit in memory together
- What about 64-bit address machine?
  - # of processes × 2^{52}
  - A page table cannot fit in a disk (2^{52} PTEs = 16PBytes)!

Segmentation with Paging



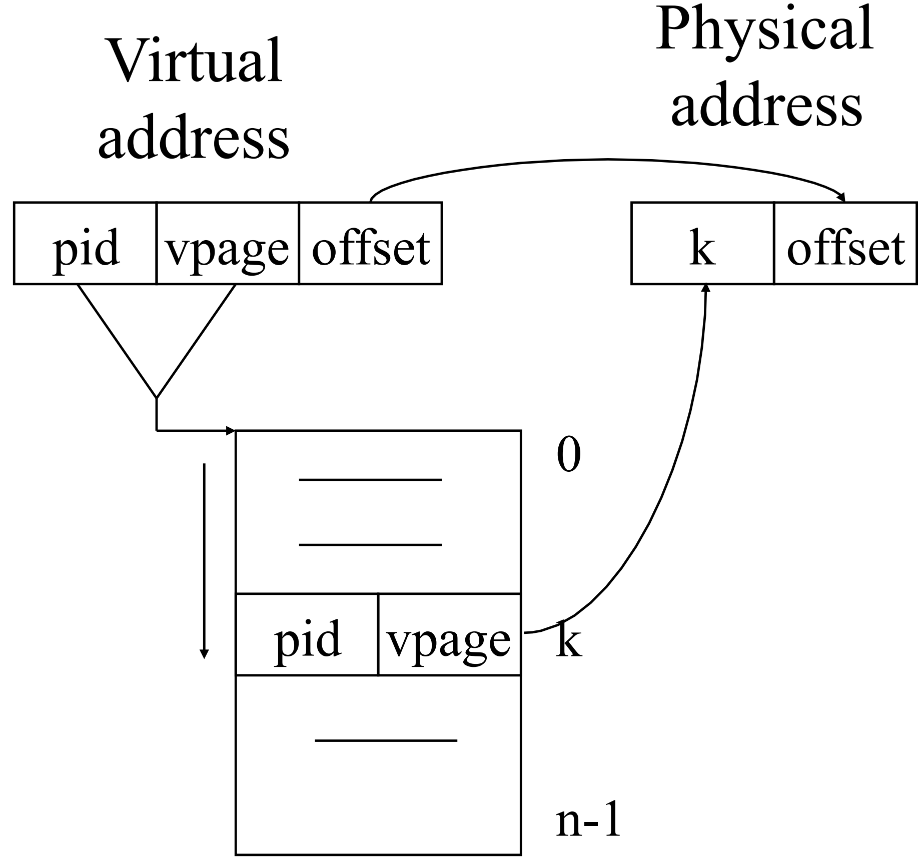
Multiple-Level Page Tables



What does this buy us?

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 chains, etc



Inverted page table

Hardware-Controlled TLB

- On a TLB miss
  - Hardware loads the PTE into the TLB
    - Write back and replace an entry if there is no free entry
  - Generate a fault if the page containing the PTE is invalid
  - VM software performs fault handling
  - Restart the CPU
- On a TLB hit, hardware checks the valid bit
  - If valid, pointer to page frame in memory
  - If invalid, treat as TLB miss

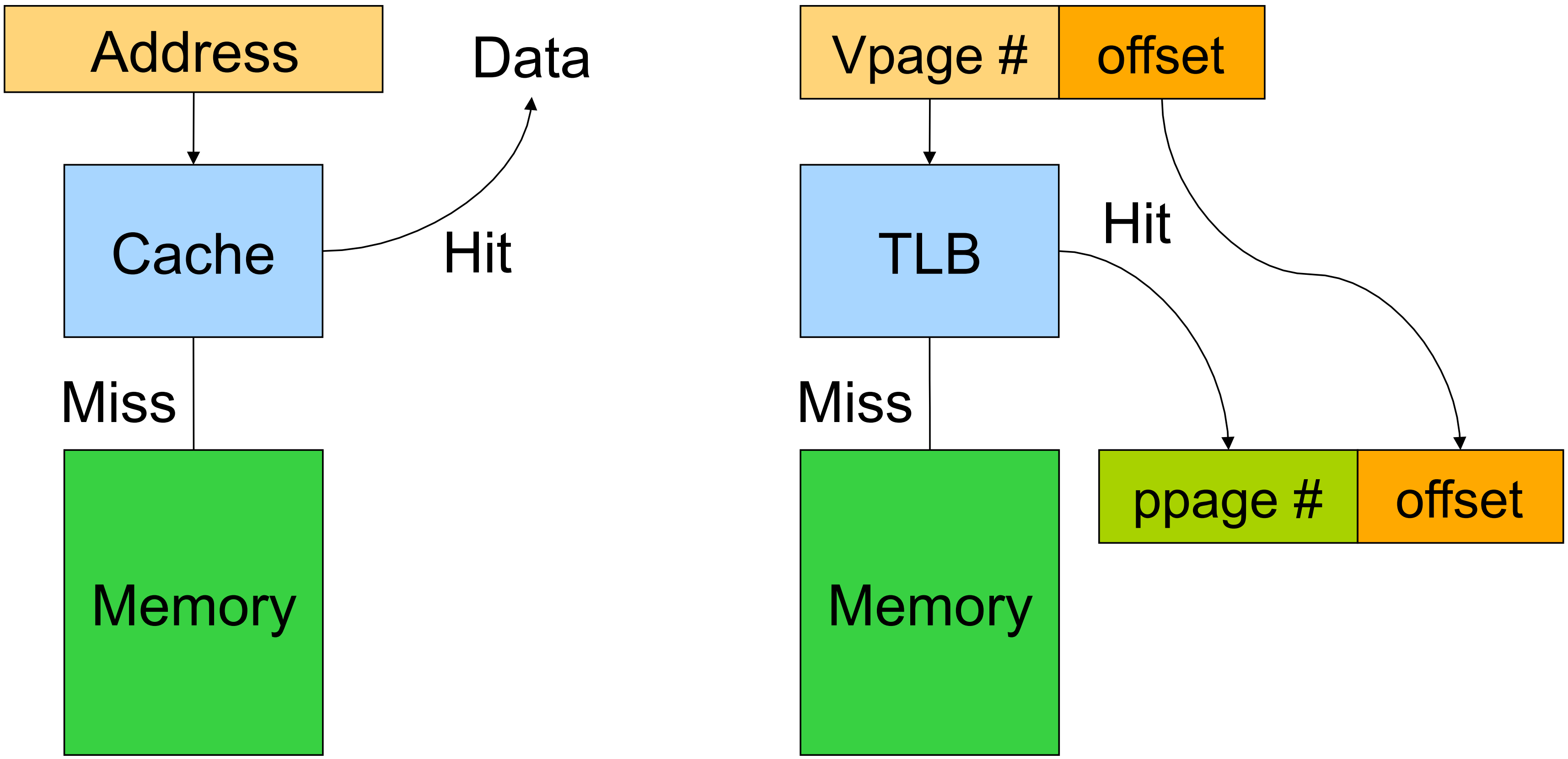
Software-Controlled TLB

- On a miss in TLB
  - 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
- On a hit in TLB, the hardware checks valid bit
  - If valid, pointer to page frame in memory
  - If invalid, treat as TLB miss

Hardware vs. Software Controlled

- Hardware approach
  - Efficient
  - Inflexible
- Software approach
  - Flexible
  - Software can do mappings by hashing
    - PP# → (Pid, VP#)
    - (Pid, VP#) → PP#
  - Can deal with large virtual address space

Cache vs. TLB



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

TLB Related Issues

- What TLB entry to be replaced?
  - Random
  - Pseudo LRU
- What happens on a context switch?
  - Process tag: change TLB registers and process register
  - 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
- TLB "shoot-down"
  - On multiprocessors, when you modify a page table entry, you need to flush all related TLB entries on all processors

Summary

- Virtual Memory
  - Virtualization makes software development easier and enables memory resource utilization better
  - Separate address spaces provide protection and isolate faults
- Address translation
  - Base and bound: very simple but limited
  - Segmentation: useful but complex
- Paging
  - TLB: fast translation for paging
  - VM needs to take care of TLB consistency issues

Midterm Grading

- Problems 1-2: Srinivas Narayan
- Problem 3: Xianmin Chen
- Problem 4: MRM
- Problem 5: Vivek Pai
- Problem 6: Mark Browning
- Problem 7: Vivek Pai

- Suggested solution online