Virtual Memory

CSE 410, Spring 2009

Computer Systems

http://www.cs.washington.edu/410

Reading and References

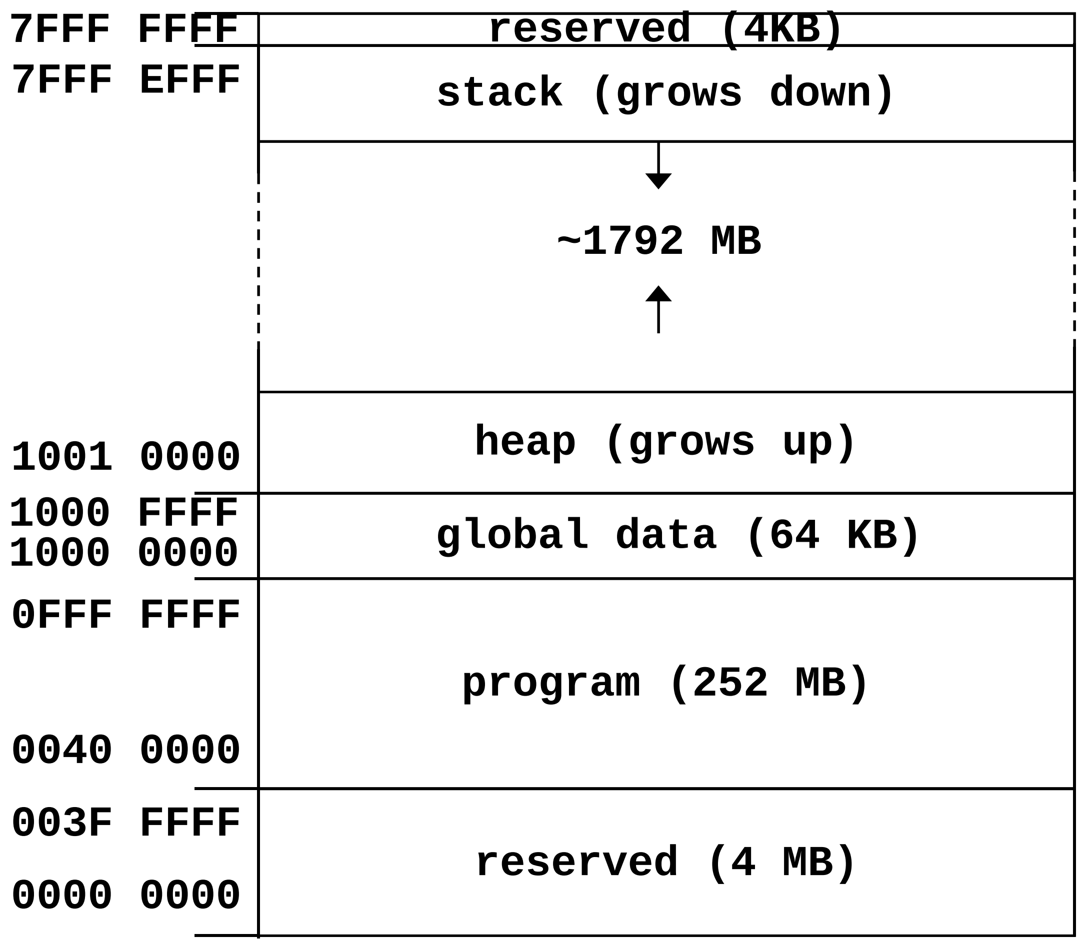
- Reading
- *Computer Organization and Design, Patterson and Hennessy*
  - » Section 5.4 Virtual Memory
  - » Section 5.5 A Common Framework for Memory Hierarchies

Memory Management Goals

We want to share main memory such that:

- Each process thinks it has a private memory of 2-4 GB (or more), even if it doesn't use it all
- Real memory is allocated efficiently to parts of process memory actually being used (locality)
- No process can interfere with or even see memory belonging to another
  - » Unless we want that to happen

Layout of program memory

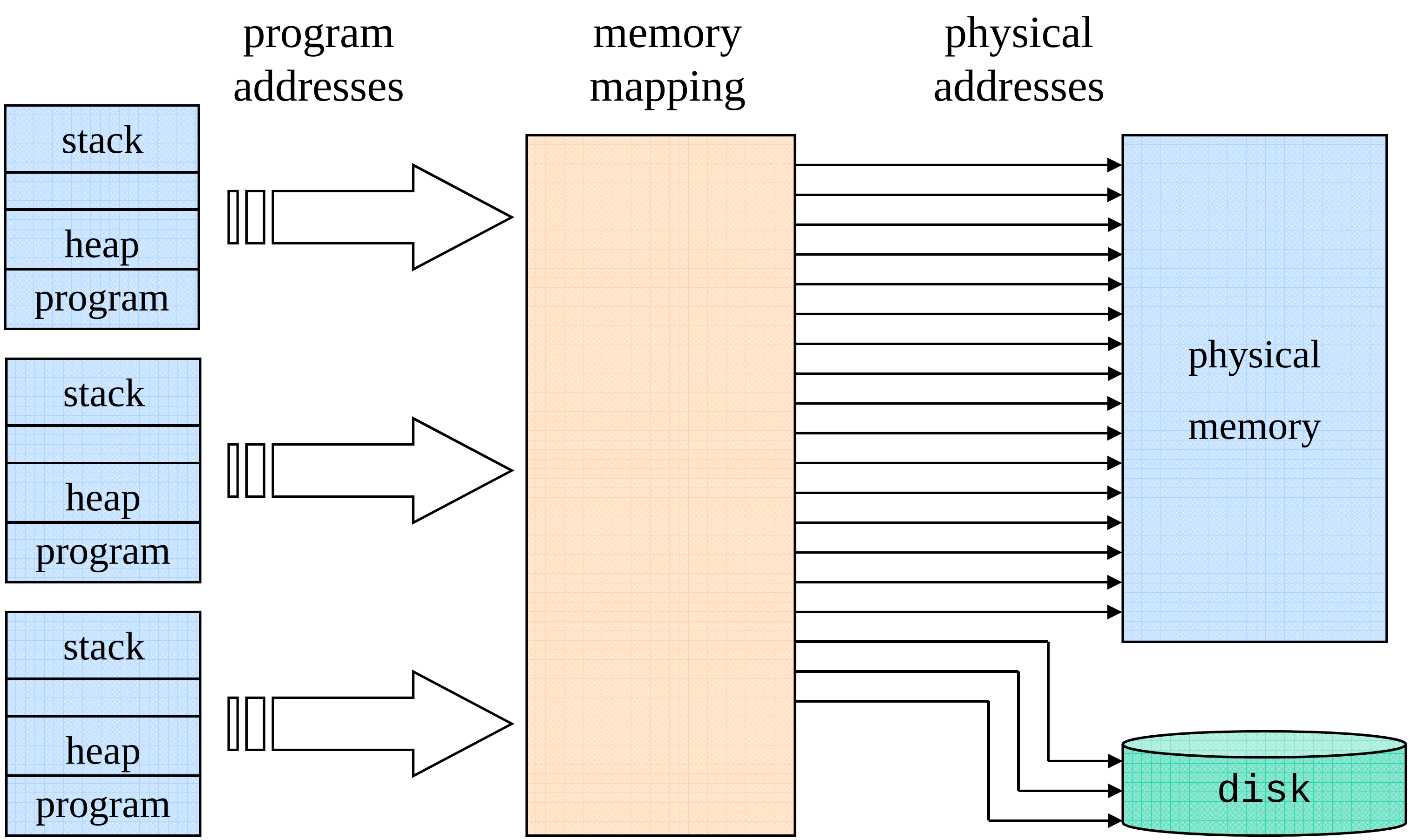


Not to Scale!

The Big Idea

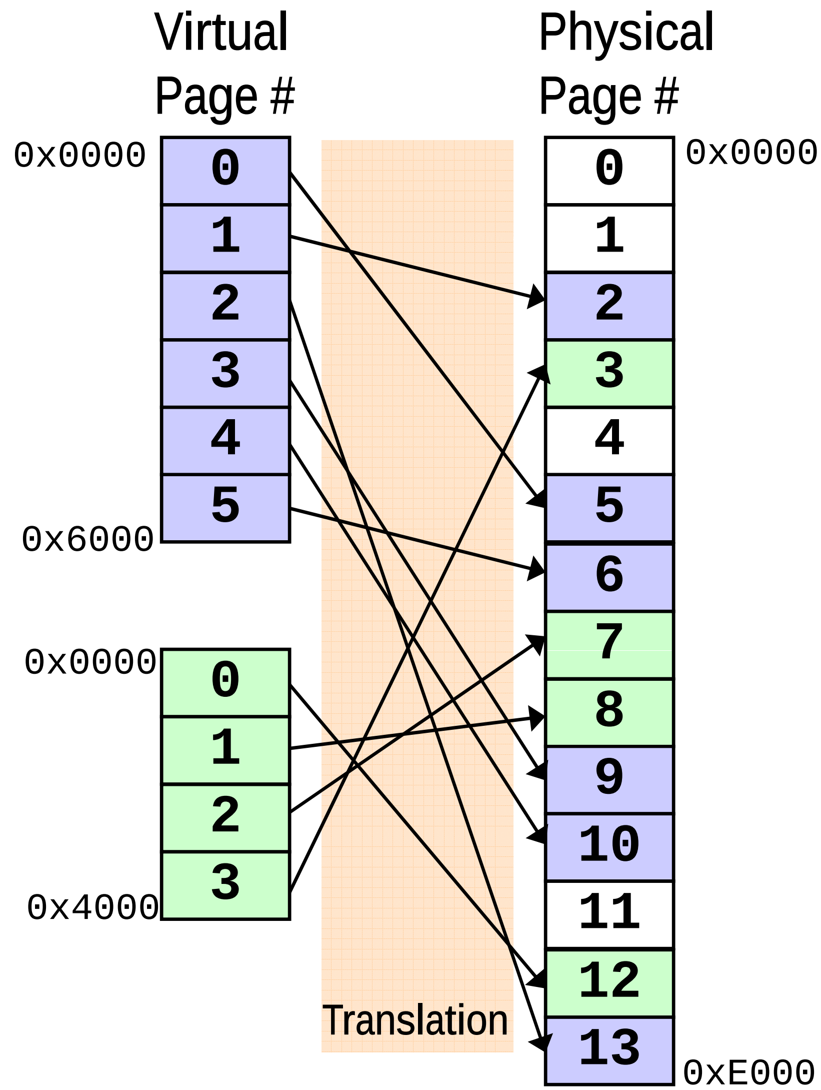
- Separate program notion of memory addresses from actual physical memory locations
  - » Program memory = virtual addresses
  - » Physical memory = real addresses
  - » Use hardware to map between the two

Memory Mapping



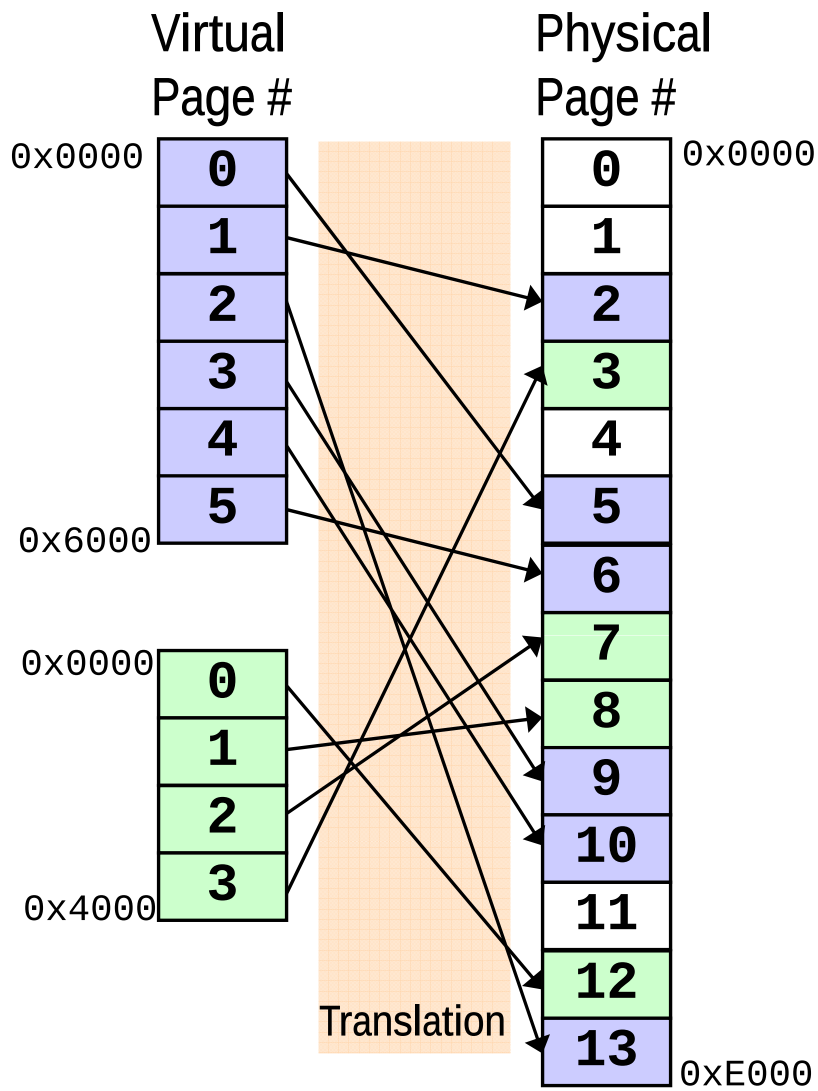
Paging

- Divide a process's virtual address space into fixed-size chunks (called **pages**)
- Divide physical memory into pages of the same size
- **Any virtual page can be located at any physical page**
- Translation box converts from virtual pages to physical pages



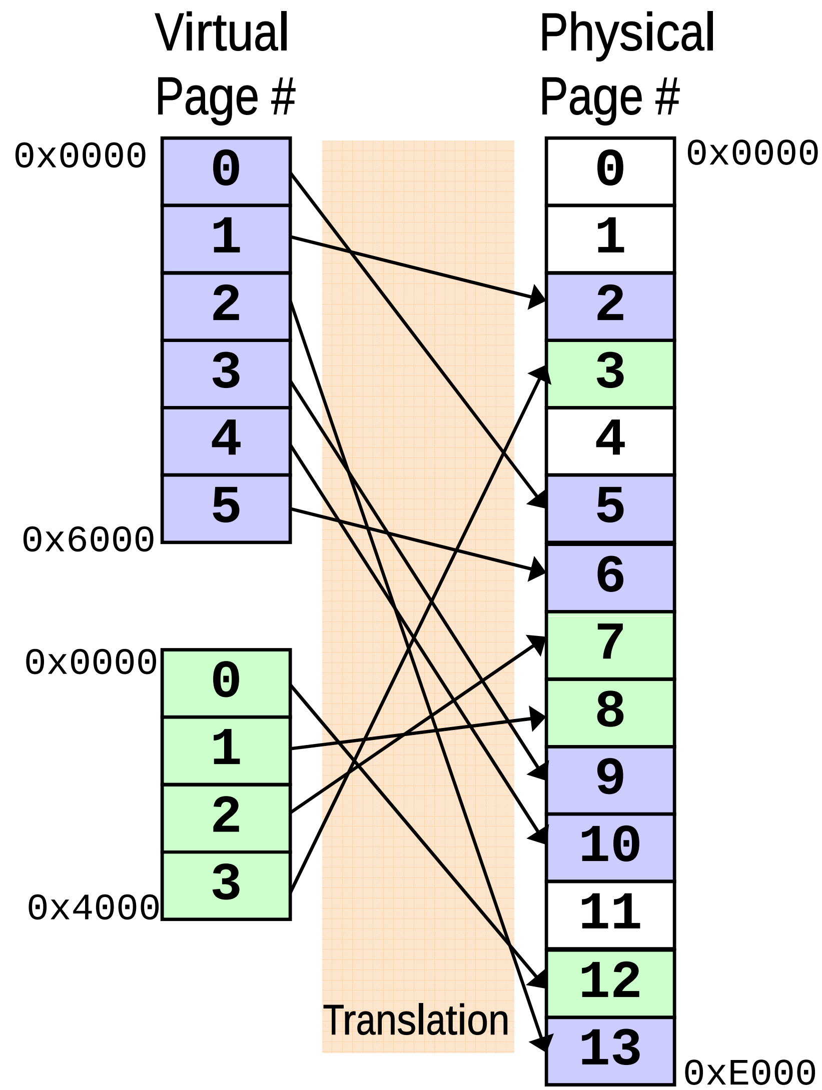
Multiple Processes Share Memory

- Each process thinks it starts at address 0x0000 and has all of memory
- A process doesn't know anything about physical addresses and doesn't care



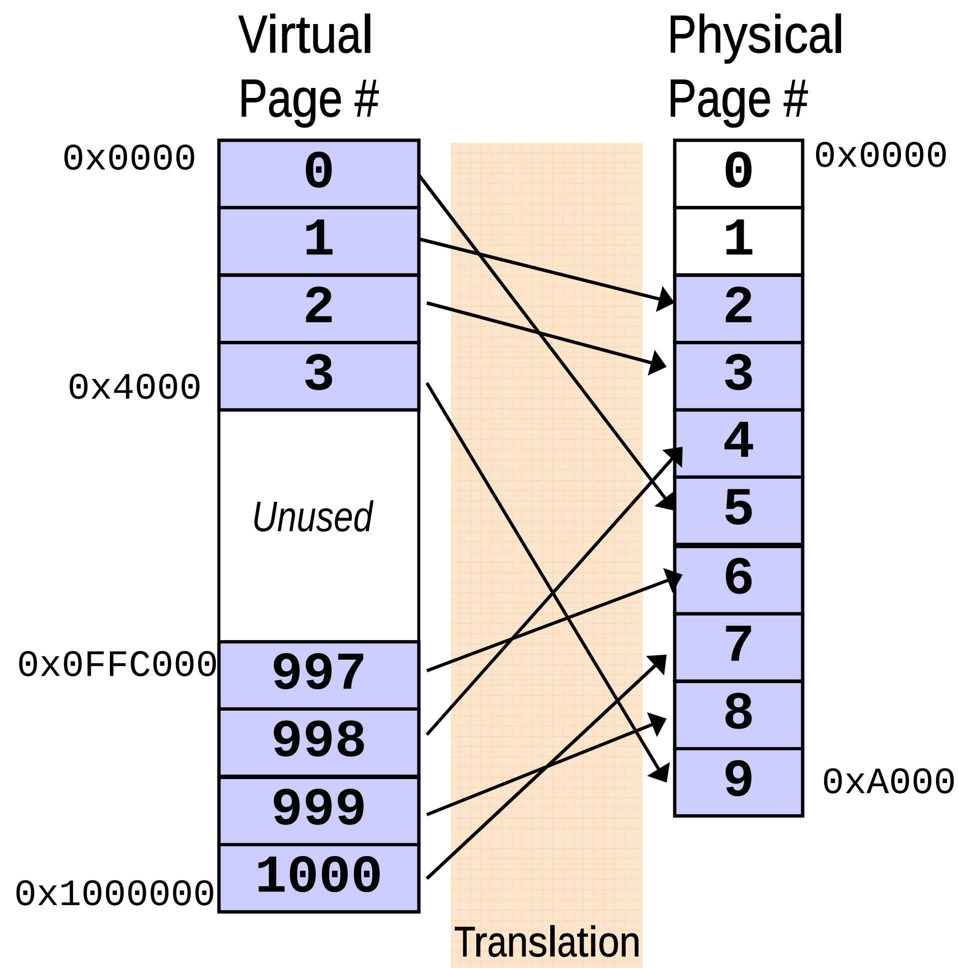
Protection

- A process can only use its own virtual addresses
- A process can't corrupt another process's memory
  - » It has no address to refer to it
- How can Blue write to Green's page 2?
  - » needs an address to refer to physical page 7, but it doesn't have one



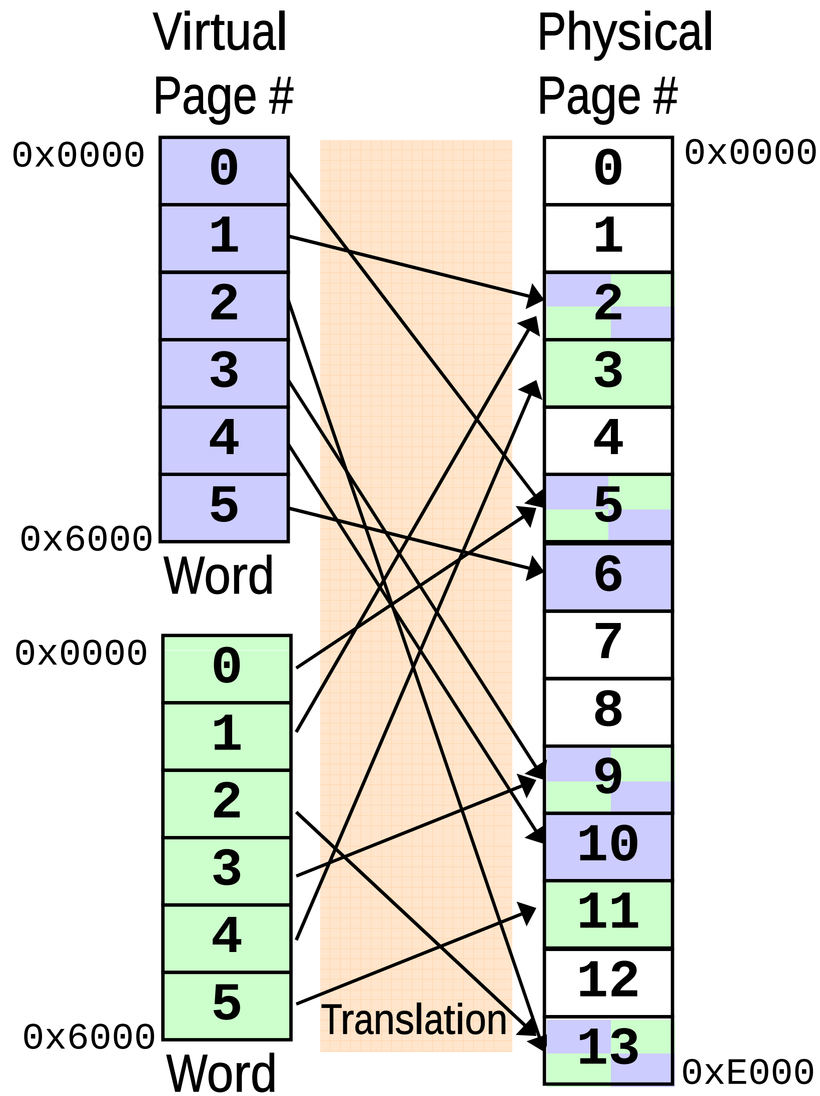
Sparse Address Spaces

- Memory addresses that aren't being used at all don't have to be assigned real addresses
  - » Code can start at a very low logical address
  - » Stack can start at a very high logical address
  - » No physical pages allocated for unused addresses in between



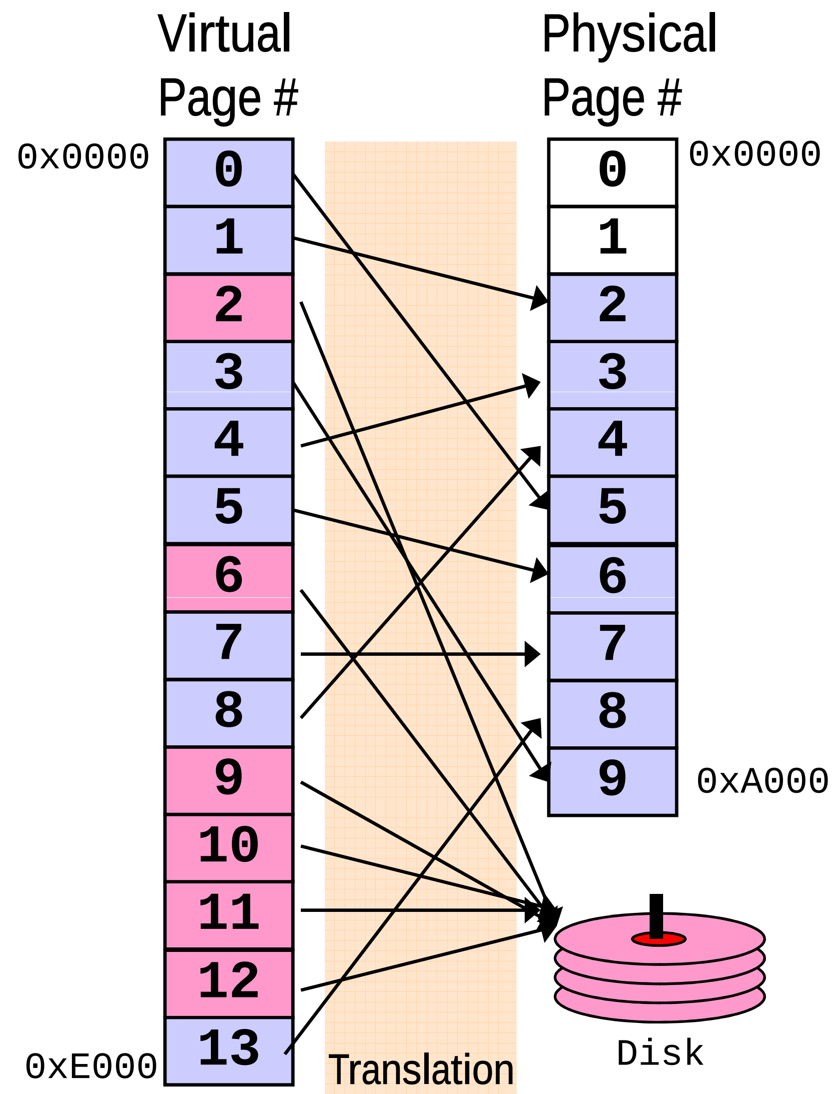
Sharing Memory

- Two processes can share memory by mapping two virtual pages to the same physical page
- The code for Word can be shared for two Word processes
  - » code pages are read only
- Each process has its own data pages
  - » possible to share data pages too, but less common



Store Memory on Disk

- Memory that isn't being used can be saved on disk
  - » swapped back in when it is referenced via page fault
- **Programs can address more memory than is physically available**
- This is one important reason for virtual memory
  - » too hard for programs to do this on their own



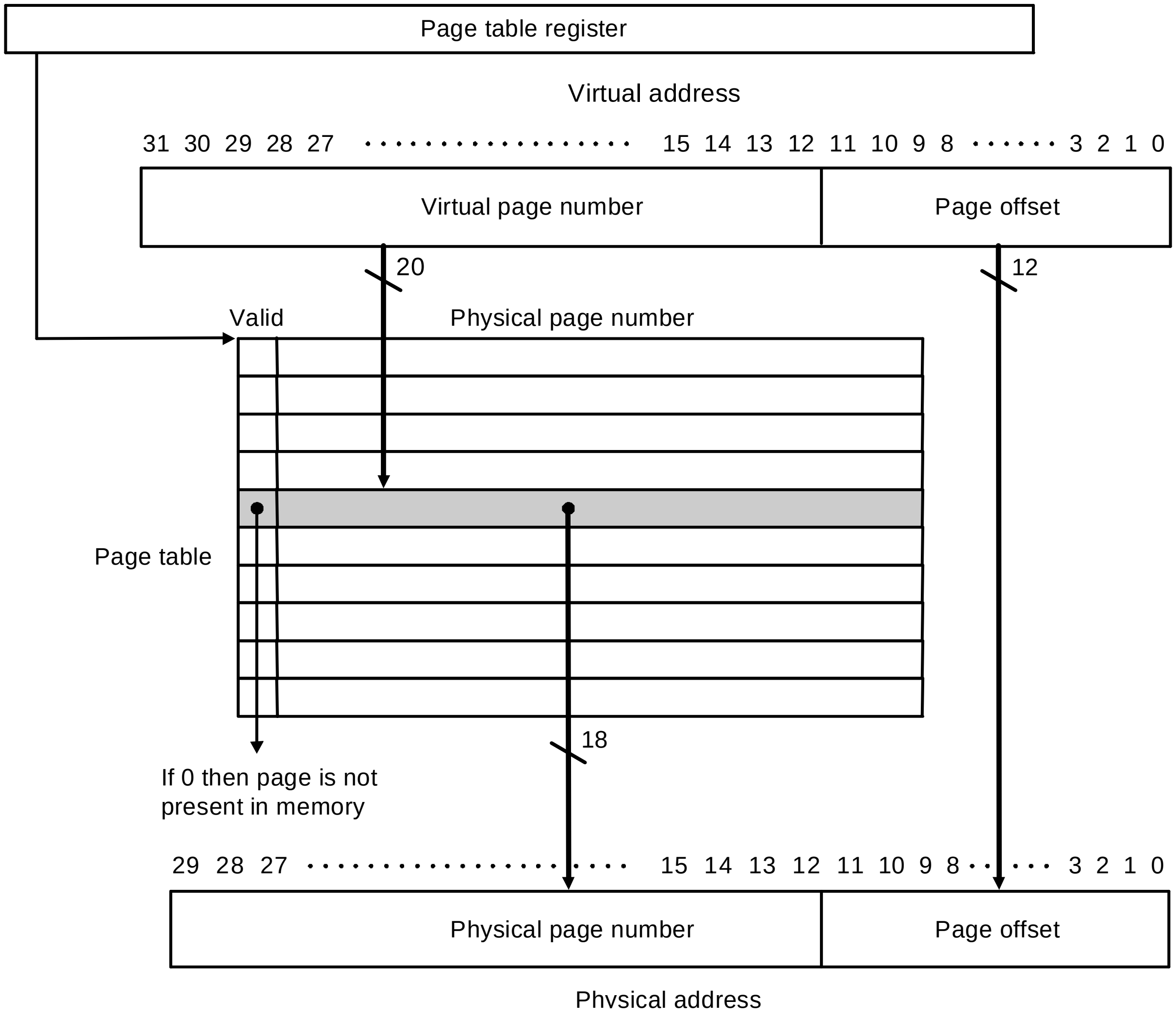
Memory Hierarchy Revisited

- Once the translation hardware is there we have a caching problem again
  - » Want size ≈ disk, performance ≈ memory
- Key issue: disk latency is 100,000 times memory, so design motivation is to avoid accessing disk
- Minimizing miss rate (“page faults”):
  - » VM “pages” are much larger than cache blocks = size of disk blocks, usually 4K or 8K or more
  - » Use fully associative lookup with approximate LRU
  - » Question: should it be write-back or write-through?

Finding the Right Page (frame)

- If fully associative, how do we find the right page without scanning all of memory?
- Answer: index is called the page table
  - » Each process has a separate page table
    - Processor “page table register” points to active one – part of process state
  - » Page table indexed with virtual page number (VPN)
    - The bits that aren’t part of the page offset
  - » Each entry contains a valid bit and a physical page number (PPN)
    - PPN is concatenated with page offset to get physical address
  - » No index tag needed – full VPN is index

Page Table picture

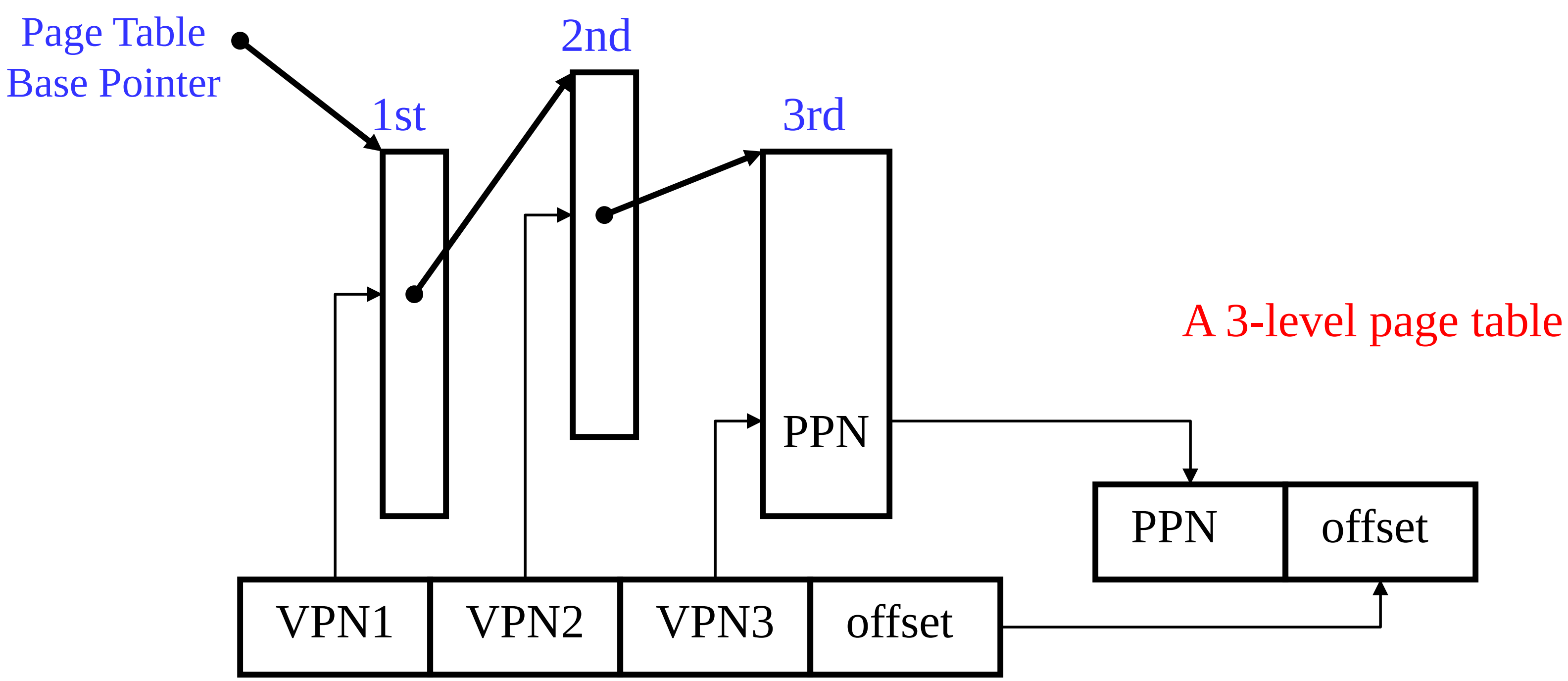


How big is the page table?

- From the previous slide:
  - » Virtual page number is 20 bits.
  - » Physical page number is 18 bits + valid bit -> round up to 32 bits.
    - Or 20 bits + valid bit if 32-bit physical addressing

Dealing with large page tables

- Multi-level page tables
  - » "Any problem in CS can be solved by adding a level of indirection"
    or two…



- Since most processes don't use the whole address space, you don't allocate the tables that aren't needed
  - » Also, the 2nd and 3rd level page tables can be "paged" to disk.

Waitaminute!

- We've just replaced every memory access MEM[addr] with:

  MEM[MEM[MEM[MEM[PTBR + VPN1<<2] + VPN2<<2] + VPN3<<2] + offset]

  » *i.e.*, 4 memory accesses

- And we haven't talked about the bad case yet (*i.e.*, page faults)…

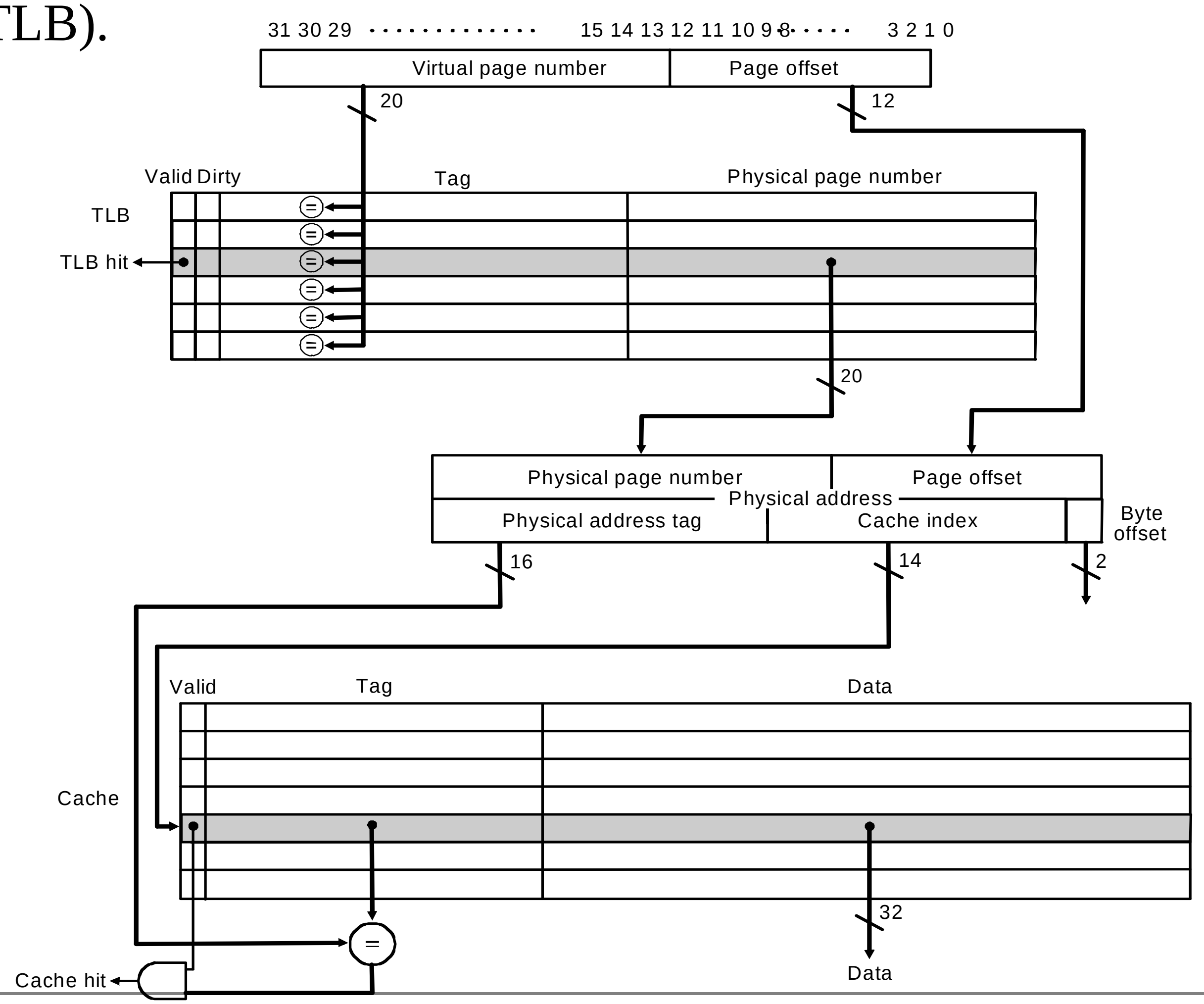
"Any problem in CS can be solved by adding a level of indirection"

» except too many levels of indirection…

- How do we deal with too many levels of indirection?

Caching Translations

- Virtual to Physical translations are cached in a Translation Lookaside Buffer (TLB).



What about a TLB miss?

- If we miss in the TLB, we need to “walk the page table”
  - » In MIPS, an exception is raised and software fills the TLB
  - » In x86, a “hardware page table walker” fills the TLB
- What if the page is not in memory?
  - » This situation is called a **page fault**.
  - » The operating system will have to read the page from disk.
  - » It will need to select a page to replace.
    - The O/S tries to approximate LRU (coming next)
  - » The replaced page will need to be written back if dirty.

Summary

- Virtual memory is great:
  - » It means that we don't have to manage our own memory.
  - » It allows different programs to use the same physical memory.
  - » It provides protect between different processes.
  - » It allows controlled sharing between processes (albeit somewhat inflexibly).
- The key technique is **indirection**:
  - » Yet another classic CS trick you've seen in this class.
  - » Many problems can be solved with indirection.
- Caching made a few appearances, too:
  - » Virtual memory enables using physical memory as a cache for disk.
  - » We used caching (in the form of the Translation Lookaside Buffer) to make Virtual Memory's indirection fast.