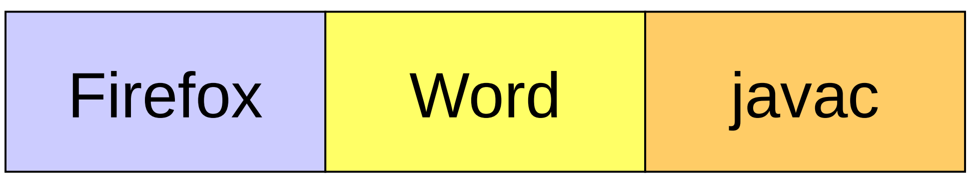
Memory Management

CS303

Operating System

Big Picture

- Up till now, we’ve focused on how multiple programs share the CPU
- Now: how do multiple programs share memory?

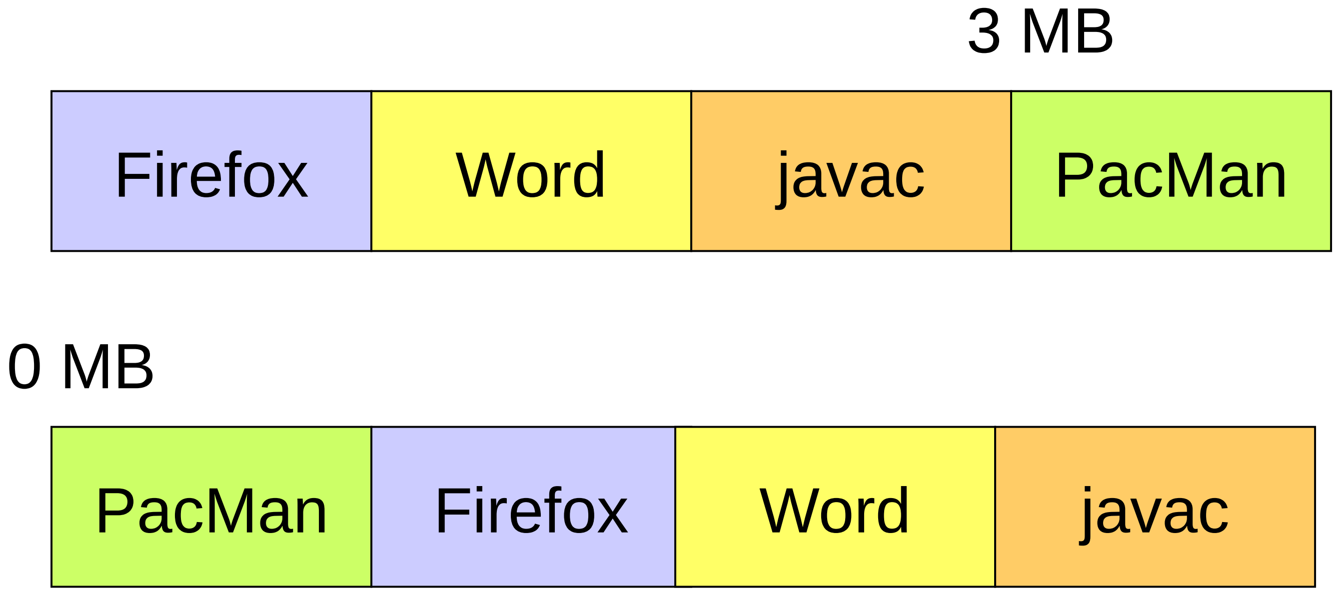


Goals of Memory Management

- Allocate scarce memory resources among competing processes
  - While maximizing memory utilization and system throughput
- Provide a convenient abstraction for programming (and for compilers, etc.)
  - Hide sharing
  - Hide scarcity
- Provide isolation between processes

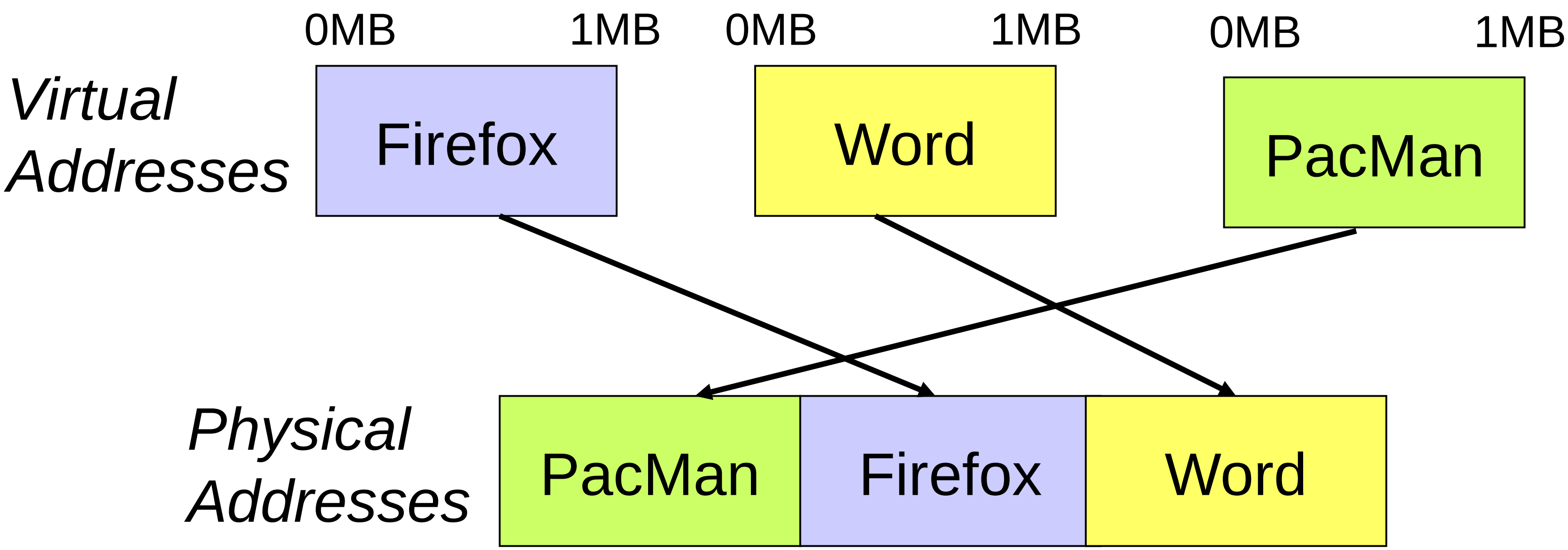
Problem: Memory Relocation

- A program “sees” different memory addresses, depending on who else is running



Virtual Addressing

- Add a layer of indirection between user addresses and hardware addresses
- Programmer sees a virtual address space, which always starts at zero



Strategy #1: Swapping

- Only one program occupies memory at once
- On a context switch:
  - Save old program's memory to disk
  - Load next program's memory from disk

- Advantage: very simple
- Disadvantage: disk is exceedingly slow!

Why Use Swapping?

- No Hardware Support
  - MIT’s CTSS operating system (1961) was built this way
    - Memory was so small that only one job would fit!
- Long-lived batch jobs
  - Job switching costs become insignificant as the quantum grows large

Strategy #2: Fixed partitions

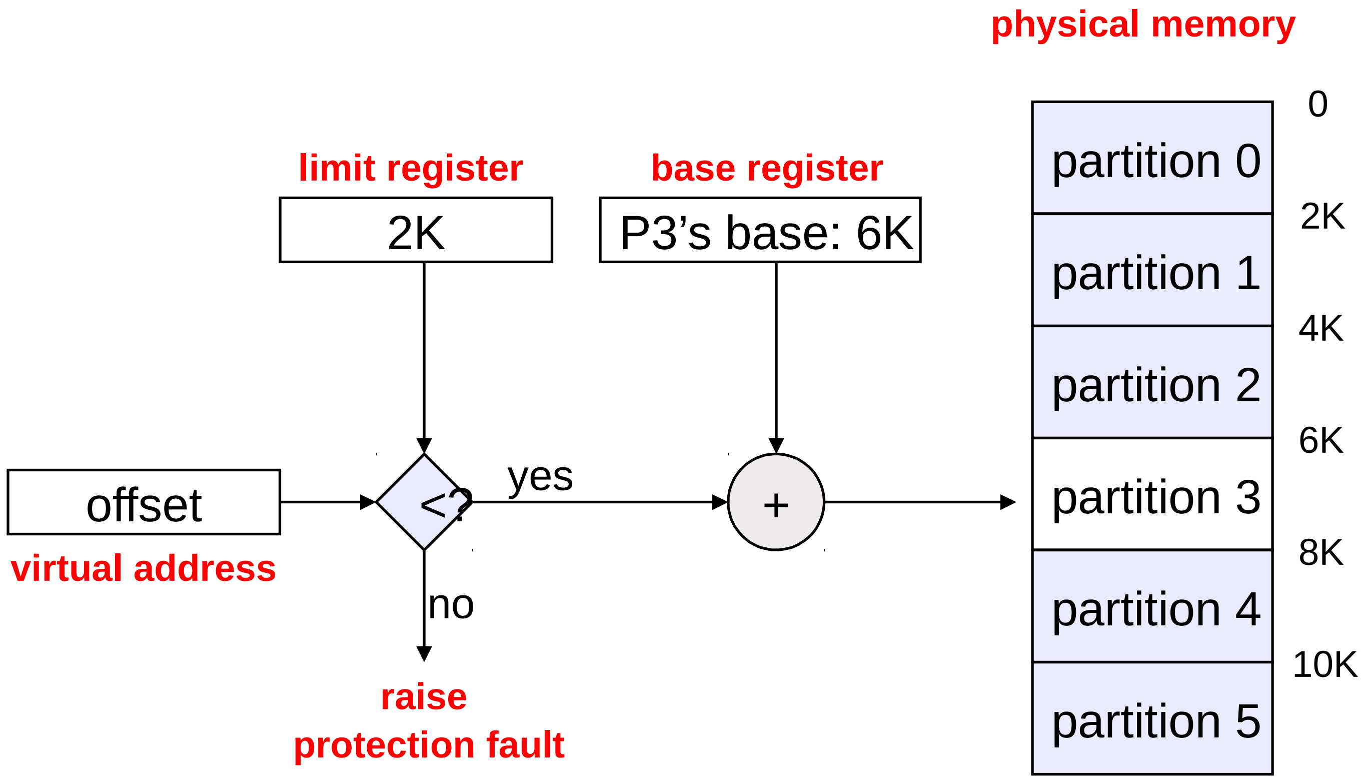
- Physical memory is broken up into fixed partitions
  - All partitions are the same size
  - Partitions never change
- Advantages
  - Simple

Disadvantages of Fixed Partitions

- Internal fragmentation: memory in a partition not used by its owning process isn’t available to other processes
- Partition size problem: no one size is appropriate for all processes
  - Tradeoff between fragmentation and accommodating large programs

Implementing Fixed Partitions

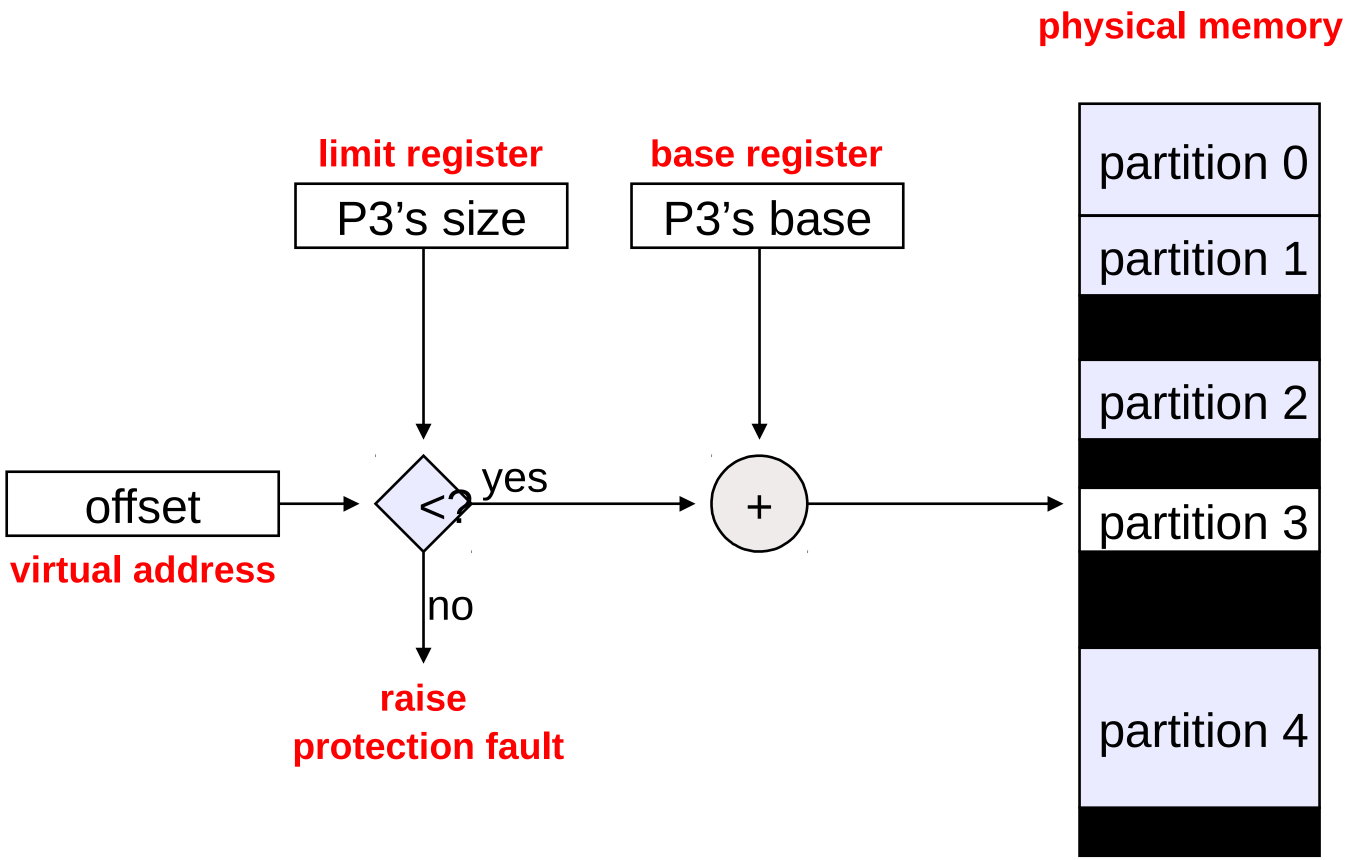
Requires hardware-supported base and limit registers



Strategy #3: Variable Partitions

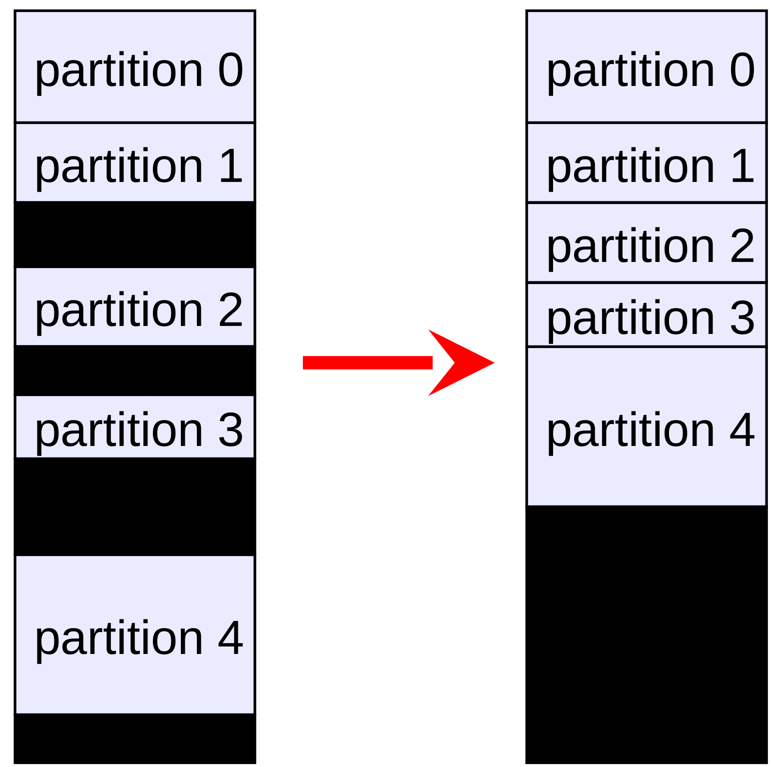
- Obvious next step: physical memory is broken up into variable-sized partitions
- Advantages
  - No internal fragmentation
    - Simply allocate partition size to be just big enough for process
- Problems
  - We must know in advance the program size
  - External fragmentation
    - As we load and unload jobs, holes are left scattered throughout physical memory

Mechanics of Variable Partitions



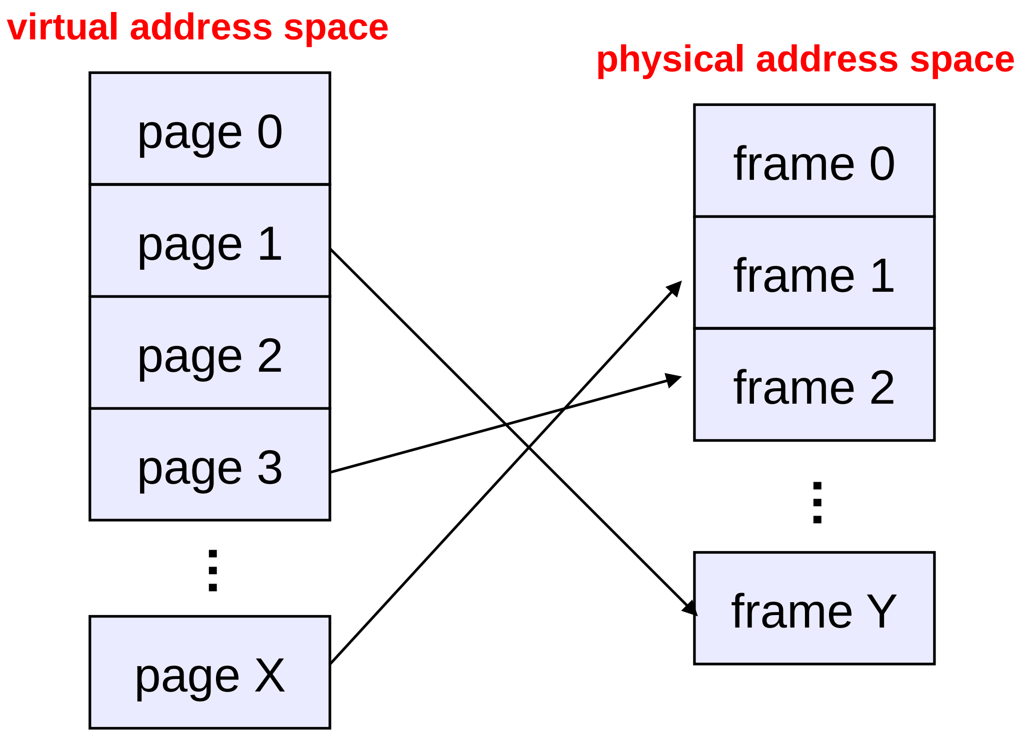
Dealing with External Fragmentation

- Swap a program out
- Re-load it, adjacent to another
- Adjust its base register
- "Lather, rinse, repeat"
- Ugh



Strategy #4: Paging

- Use fixed-size units of memory (called pages) for both virtual and physical memory



Paging Advantages

- Paging reduces internal fragmentation
  - How?
- Paging eliminates external fragmentation
  - How?

- Disadvantages?

Address translation

- A virtual address has two parts: virtual page number and offset

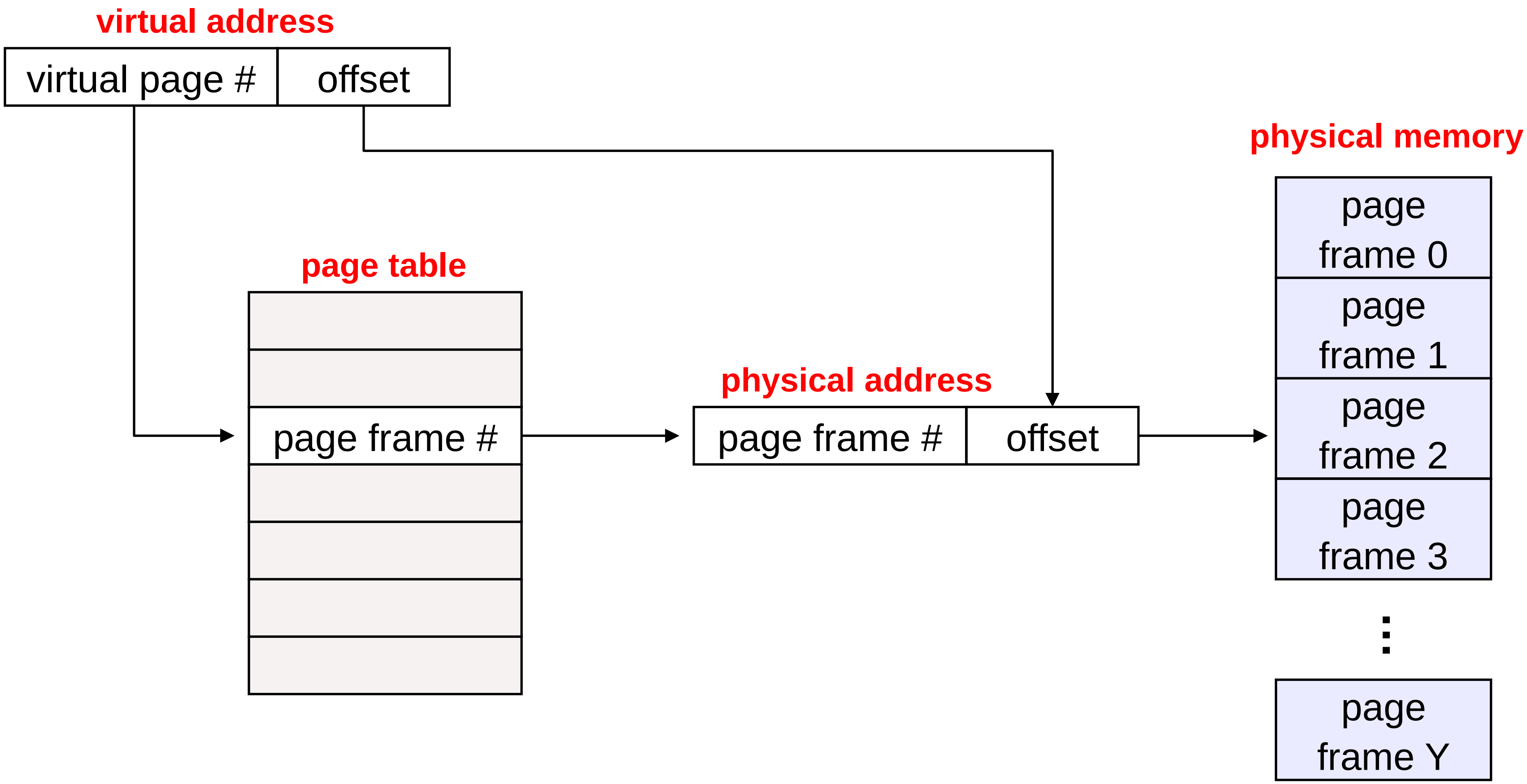
virtual page #	offset

- Address translation only applies to the virtual page number
  - Why?
- Virtual page number (VPN) is index into a page table
  - Page table entry contains page frame number (PFN)
  - Physical address is PFN::offset

Page Tables

- Managed by the OS
- Map a virtual page number (VPN) to a page frame number (PFN)
  - VPN is simply an index into the page table
- One page table entry (PTE) per page in virtual address space
  - i.e., one PTE per VPN

Mechanics of address translation



Example of address translation

- Assume 32 bit addresses
  - assume page size is 4KB (4096 bytes, or 2^{12} bytes)
  - VPN is 20 bits long (2^{20} VPNs), offset is 12 bits long

- Let's translate virtual address 0x13325328
  - VPN is 0x13325, and offset is 0x328
  - assume page table entry 0x13325 contains value 0x03004
    - page frame number is 0x03004
    - VPN 0x13325 maps to PFN 0x03004
  - physical address = PFN::offset = 0x03004328

Page Table Entries (PTEs)

1	1	1	2	20
V	R	M	prot	page frame number

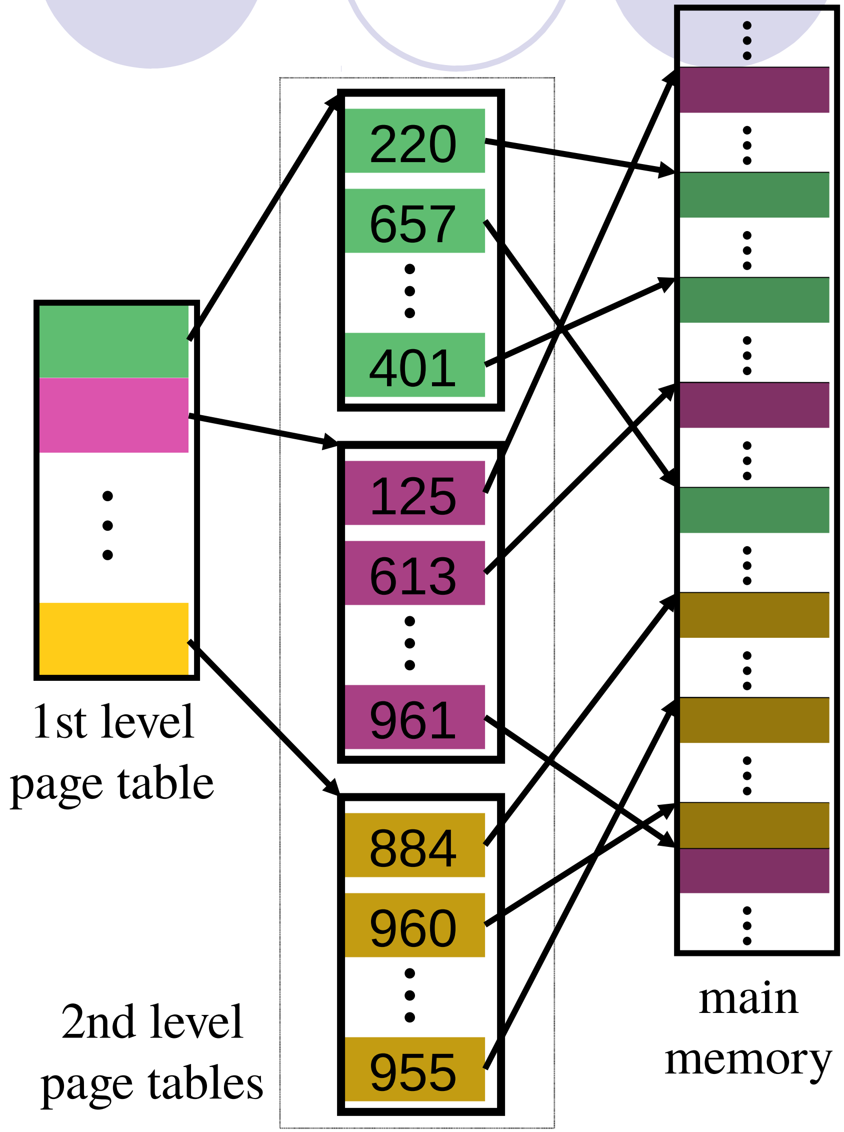
- PTE's control address mappings
  - The page frame number indicates the physical page
  - The valid bit says whether or not the PTE can be used
    - says whether or not a virtual address is valid
  - The referenced bit says whether the page has been accessed recently
  - The modified bit says whether or not the page is dirty
    - it is set when a write to the page has occurred
  - The protection bits control which operations are allowed
    - read, write, execute

Paging Issues (Stay Tuned…)

- How to make it fast?
  - Accessing the page table on each memory reference is not workable
- How to deal with memory scarcity
  - Virtual memory
- How do we control the memory overhead of page tables?
  - Need one PTE per page in virtual address space
    - 32 bit AS with 4KB pages = 2^{20} PTEs = 1,048,576 PTEs
    - 4 bytes/PTE = 4MB per page table
  - OS's typically have separate page tables per process
    - 25 processes = 100MB of page tables

Two-level page tables

- Problem: page tables can be too large
  - 2^{32} bytes in 4KB pages need 1 million PTEs
- Solution: use multi-level page tables
  - “Page size” in first page table is large (megabytes)
  - PTE marked invalid in first page table needs no 2nd level page table
- 1st level page table has pointers to 2nd level page tables
- 2nd level page table has actual physical page numbers in it

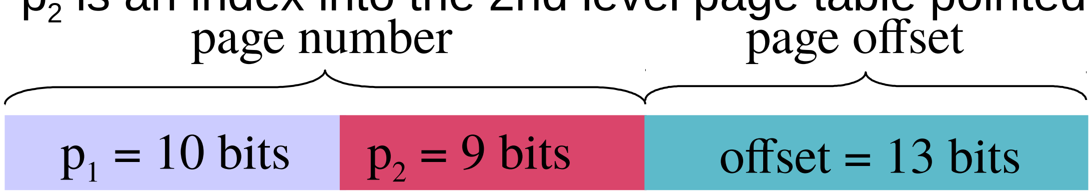


More on two-level page tables

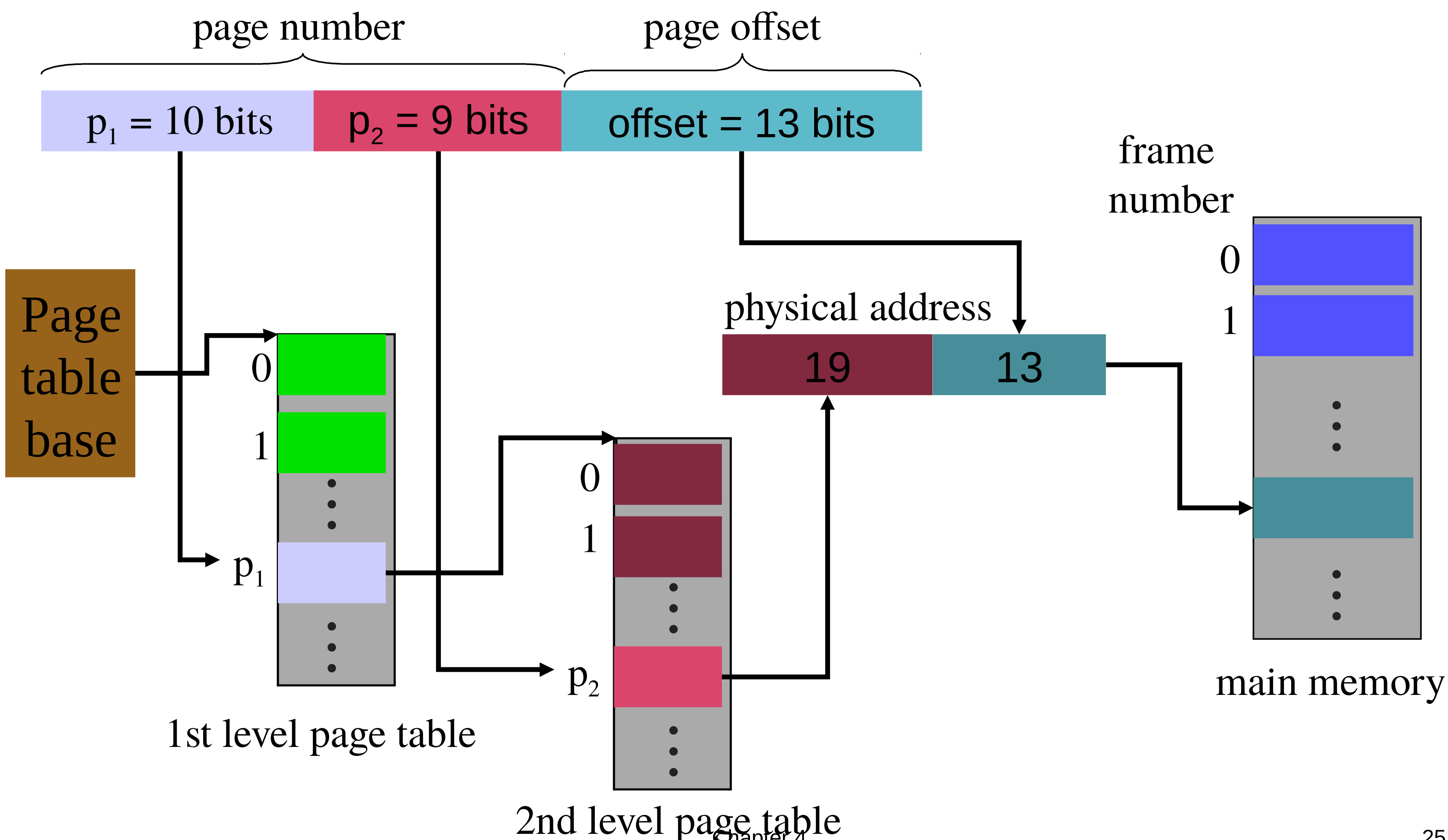
- Tradeoffs between 1st and 2nd level page table sizes
  - Total number of bits indexing 1st and 2nd level table is constant for a given page size and logical address length
  - Tradeoff between number of bits indexing 1st and number indexing 2nd level tables
    - More bits in 1st level: fine granularity at 2nd level
    - Fewer bits in 1st level: maybe less wasted space?

Two-level paging: example

- System characteristics
  - 8 KB pages
  - 32-bit logical address divided into 13 bit page offset, 19 bit page number
- Page number divided into:
  - 10 bit page number
  - 9 bit page offset
- Logical address looks like this:
  - p_1 is an index into the 1st level page table
  - p_2 is an index into the 2nd level page table pointed to by p_1



2-level address translation example



Strategy #5: Segmentation

- Instead of a flat address space, programmers see a collection of segments

heap	stack	main()	Shared library

- Virtual address = segment #, offset

segment #	offset

Why Segments?

- Facilitates sharing and re-use
  - Unlike a page, a segment is a **logical** entity
    - Segments can have arbitrary size