

CSE 451: Operating Systems
Spring 2009

Memory Management

Steve Gribble

Memory Management

- We're beginning a new multiple-lecture topic
 - goals of memory management
 - convenient abstraction for programming
 - isolation between processes
 - allocate scarce memory resources between competing processes, maximize performance (minimize overhead)
 - mechanisms
 - physical vs. virtual address spaces
 - page table management, segmentation policies
 - page replacement policies

Tools of memory management

- Address Translation
- Base and limit registers
- Swapping
- Paging
- Page Tables
- TLBs
- Segmentation (and segment tables)
- Page faults => page fault handling => virtual memory
- The policies that govern the use of these mechanisms

Virtual Memory from 10,000 feet

- The basic abstraction that the OS provides for memory management is **virtual memory** (VM)
 - VM enables programs to execute without requiring their entire address space to be resident in physical memory
 - program can also execute on machines with less RAM than it “needs”
 - many programs don’t need all of their code or data at once (or ever)
 - e.g., branches they never take, or data they never read/write
 - no need to allocate memory for it, OS should adjust amount allocated based on its **run-time** behavior
 - virtual memory **isolates** processes from each other
 - one process cannot name addresses visible to others; each process has its own isolated address space
- VM requires hardware and OS support
 - MMU’s, TLB’s, page tables, ...
 - Typically uses swapping as well

Virtual Addresses

- To make it easier to manage memory of multiple processes, make processes use **virtual addresses**
 - virtual addresses are independent of location in physical memory (RAM) that referenced data lives
 - OS determines location in physical memory
 - instructions issued by CPU reference virtual addresses
 - e.g., pointers, arguments to load/store instruction, PC, ...
 - virtual addresses are translated by hardware into physical addresses (with some help from OS)

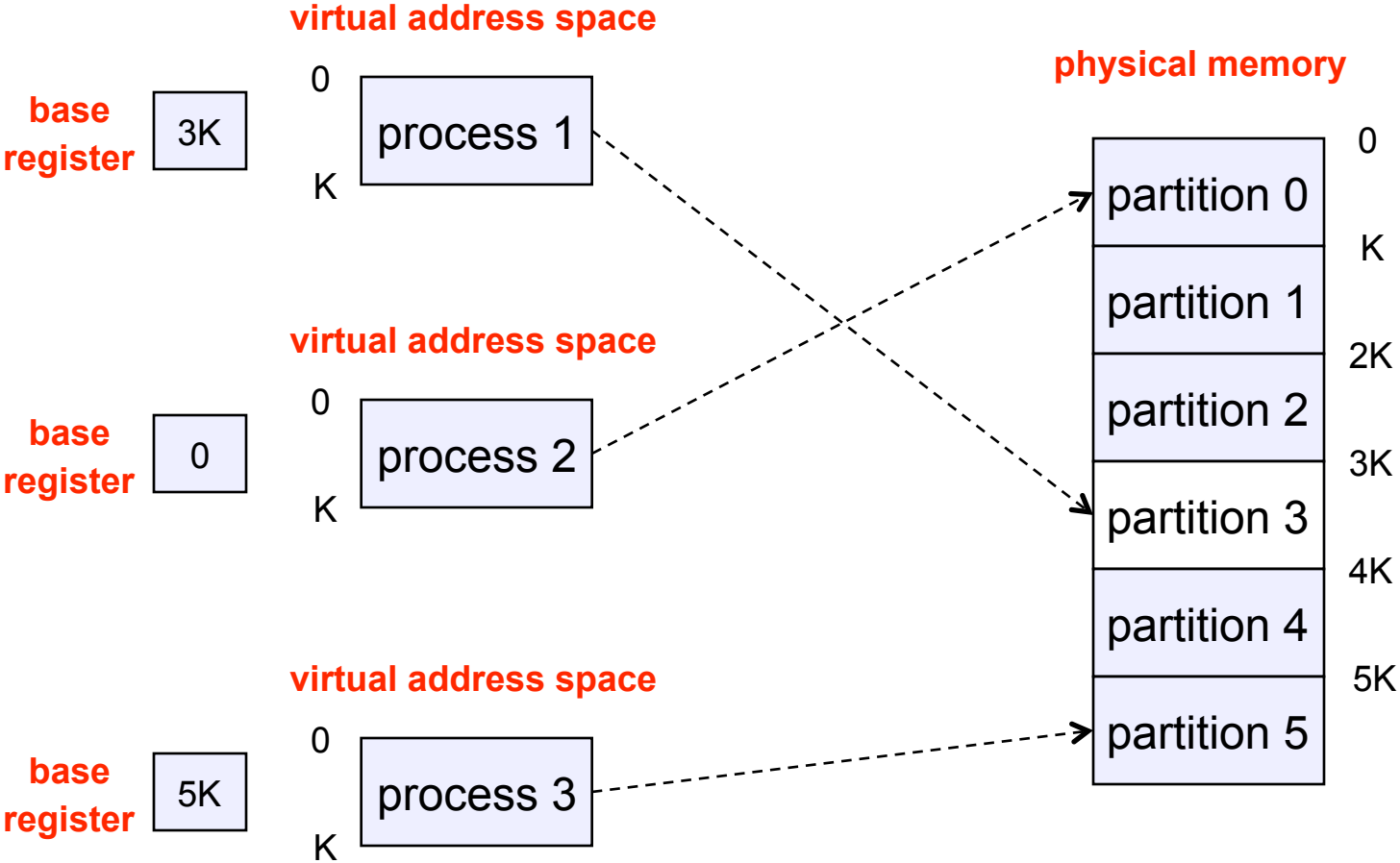
Virtual Addresses (2)

- The set of virtual addresses a process can reference is its **address space**
 - many different possible mechanisms for translating virtual addresses to physical addresses
 - we'll take a historical walk through them, ending up with our current techniques
- Note: We are not yet talking about paging, or virtual memory – only that the program issues addresses in a virtual address space, and these must be “adjusted” to reference memory (the physical address space)
 - for now, think of the program as having a contiguous virtual address space that starts at 0, and a contiguous physical address space that starts somewhere else

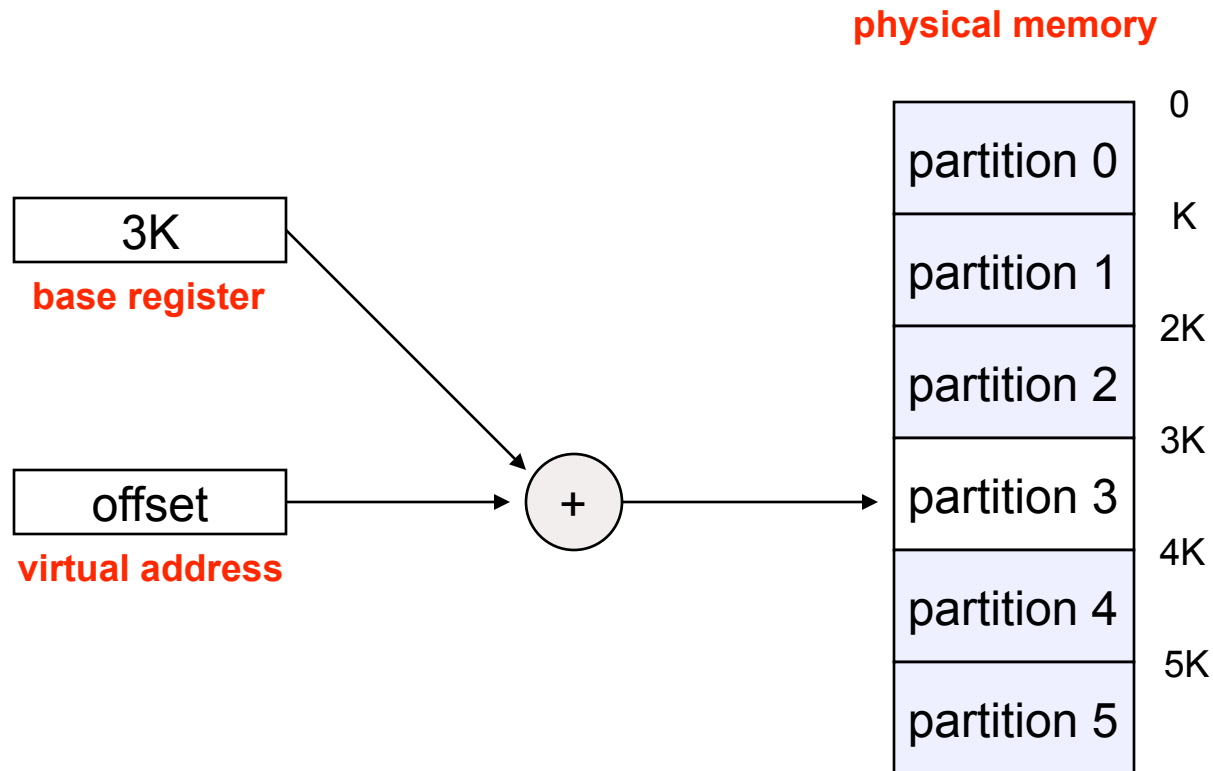
Old technique #1: Fixed Partitions

- Physical memory is broken up into fixed partitions
 - all partitions are equally sized, partitioning never changes
 - hardware requirement: **base register**
 - physical address = virtual address + base register
 - base register loaded by OS when it switches to a process
 - how can we ensure protection?
- Advantages
 - simple, ultra-fast context switch
- Problems
 - **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
 - fragmentation vs. fitting large programs in partition

Fixed Partitions (K bytes)



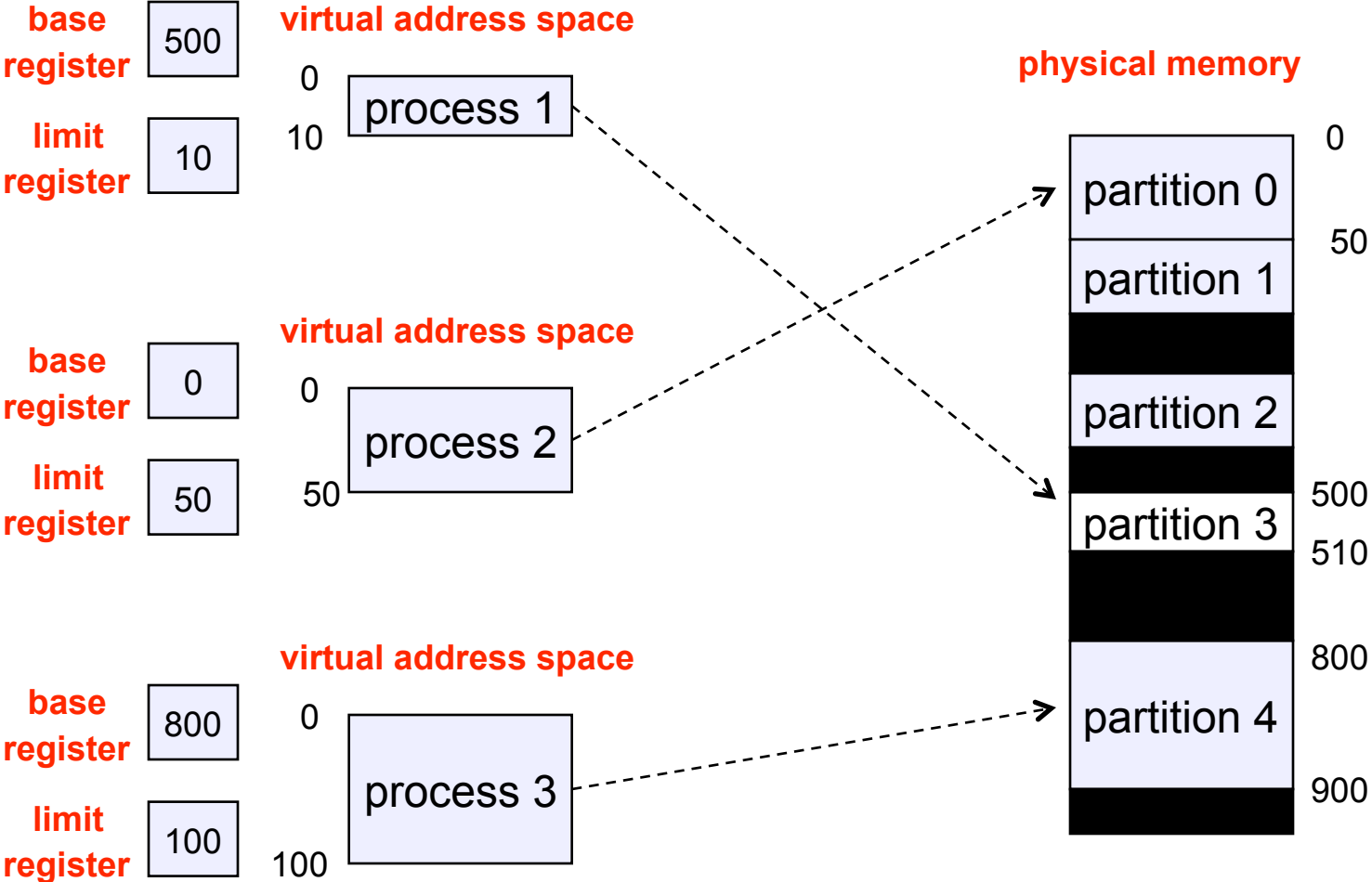
Virtual-to-physical address translation with Fixed Partitions (K bytes)



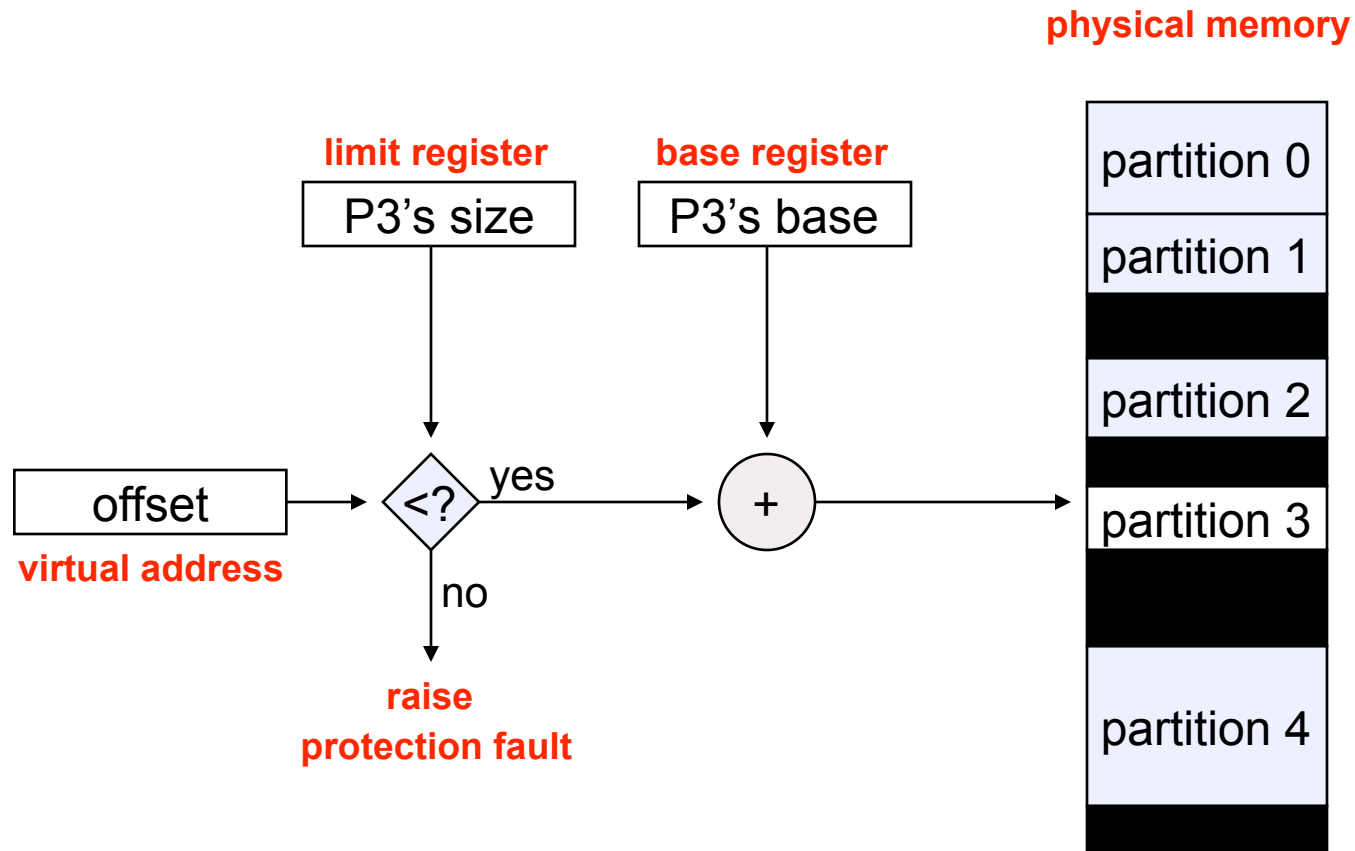
Old technique #2: Variable Partitions

- Obvious next step: physical memory is broken up into variable-sized partitions
 - hardware requirements: **base register**, **limit register**
 - physical address = virtual address + base register
 - how do we provide protection?
 - if (physical address > base + limit) then... ?
- Advantages
 - no internal fragmentation
 - simply allocate partition size to be just big enough for process
 - (assuming we know what that is!)
- Problems
 - **external fragmentation**
 - as we load and unload jobs, holes are left scattered throughout physical memory

Variable Partitions

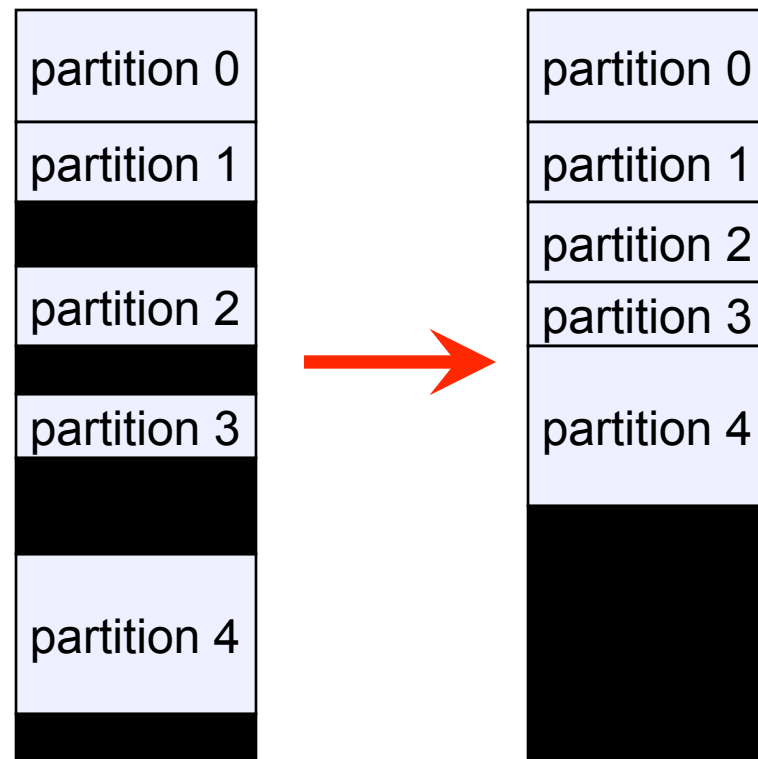


Address translation with Variable Partitions



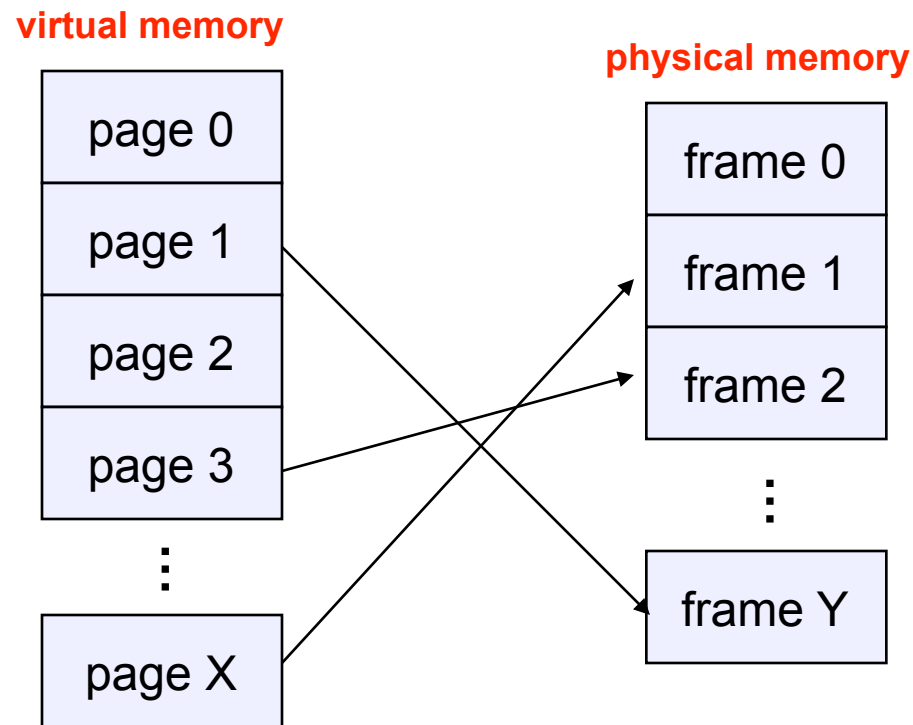
Dealing with fragmentation

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



Modern technique: Paging

- Solve the external fragmentation problem by using fixed sized units in both physical and virtual memory



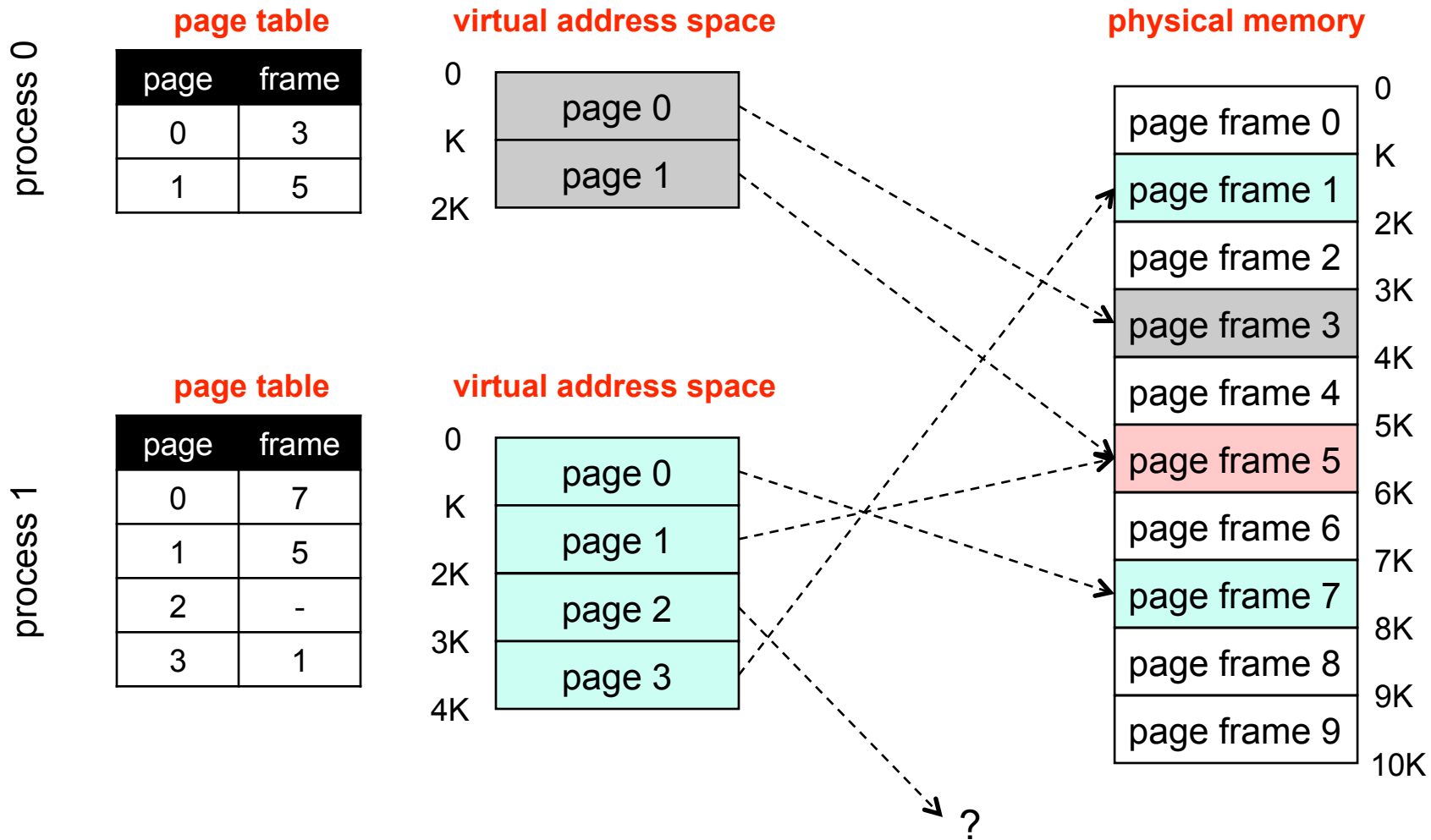
User's Perspective

- Processes view memory as a contiguous address space from bytes 0 through N
 - virtual address space (VAS)
- In reality, virtual pages are scattered across physical memory frames
 - virtual-to-physical mapping
 - this mapping is **invisible** to the program
- Protection is provided because a program cannot reference memory outside of its VAS
 - the virtual address 0xDEADBEEF maps to different physical addresses for different processes

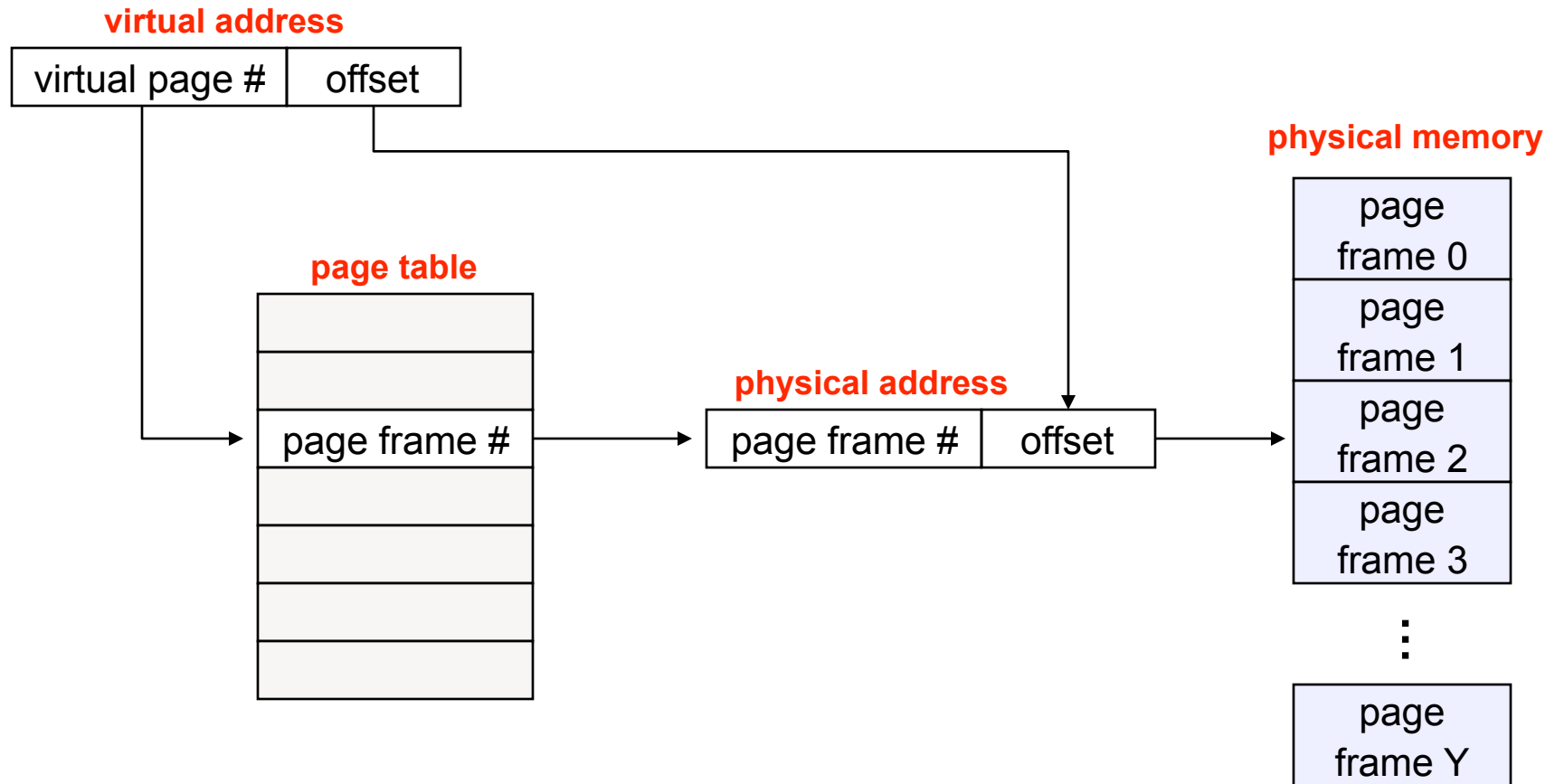
Paging

- Translating virtual addresses
 - a virtual address has two parts: **virtual page number** & **offset**
 - 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 virtual page number (VPN) to 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

Paging (K byte pages)



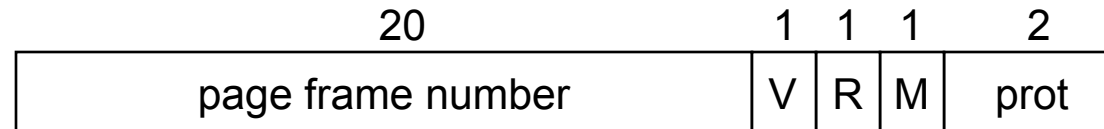
Address translation with Paging



Paging example

- 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)



- PTE's control mapping
 - the **valid bit** says whether or not the PTE can be used
 - says whether or not a virtual address is valid
 - it is checked each time a virtual address is used
 - the **reference bit** says whether the page has been accessed
 - it is set when a page has been read or written to
 - the **modify 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; user vs. supervisor
 - the **page frame number** determines the physical page
 - physical page start address = PFN << (#bits/page)

Paging Advantages

- Easy to allocate physical memory
 - physical memory is allocated from free list of frames
 - to allocate a frame, just remove it from its free list
 - external fragmentation is not a problem!
 - complication for kernel contiguous physical memory allocation
 - many lists, each keeps track of free regions of particular size
 - regions' sizes are multiples of page sizes
 - “buddy algorithm”
- Easy to “page out” chunks of programs
 - all chunks are the same size (page size)
 - use valid bit to detect references to “paged-out” pages
 - also, page sizes are usually chosen to be convenient multiples of disk block sizes

Paging Disadvantages

- Can still have internal fragmentation
 - process may not use memory in exact multiples of pages
- Memory reference overhead
 - 2 references per address lookup (page table, then memory)
 - solution: use a hardware cache to absorb page table lookups
 - translation lookaside buffer (TLB) – next class
- Memory required to hold page tables can be large
 - 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
 - one solution: page the page tables (!!!)
 - another solution: multi-level page tables