CSE 451: Operating Systems
Winter 2026

Module 11
Memory Management

Gary Kimura

Goals of memory management

- Allocate memory resources among competing processes, maximizing memory utilization and system throughput
- Provide memory isolation between processes
  - We have come to view “addressability” and “protection” as inextricably linked, even though they’re really orthogonal
- Provide a convenient abstraction of memory for programming (and for compilers, etc.)

Tools of memory management

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

- We will cover all of these soon, but first a few terms we need to loosely define…

Today's server, desktop, laptop, tablet, and phone systems

- The basic abstraction that the OS provides for memory management is virtual memory (VM)
  - Efficient use of hardware (real memory)
    - VM enables programs to execute without requiring their entire address space to be resident in physical memory
    - Many programs don't need all of their code or data at once (or ever – branches they never take, or data they never read/write)
    - No need to allocate memory for it, OS should adjust amount allocated based on run-time behavior
  - Program flexibility
    - Programs can execute on machines with less RAM than they "need"
      - On the other hand, paging is really slow, so must be minimized!
  - Protection
    - Virtual memory isolates address spaces 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 (Memory Management Unit)
- TLB's (Translation Lookaside Buffer)
- Page tables

- page fault handling, …

- Sometimes accompanied by swapping, or limited segmentation

A trip down Memory Lane …

- Why?
  - Because it's instructive
  - Because embedded processors (98% or more of all processors) typically don't have virtual memory
  - Because some aspects are pertinent to allocating portions of a virtual address space – e.g., malloc()

- First, there was one job-at-a-time batch programming
  - programs used physical addresses directly
  - OS loads job (perhaps using a relocating loader to "offset" branch addresses), runs it, unloads it
  - what if the program wouldn't fit into memory?
    - manual overlays!

- An embedded system may have only one program!

- Then came Swapping
  - save a program’s entire state (including its memory image) to disk
  - allows another program to be run
  - first program can be swapped back in and re-started right where it was

  - The first timesharing system, MIT’s “Compatible Time Sharing System” (CTSS) (circa 1961), was a uni-programmed swapping system
    - only one memory-resident user
    - upon request completion or quantum expiration, a swap took place
    - slow … but it worked!
- A later system, MULTICS (circa 1967) furthered OS development

- Then came Multiprogramming
  - multiple processes/jobs in memory at once
    - to overlap I/O and computation between processes/jobs, easing the task of the application programmer
  - memory management requirements:
    - protection: restrict which addresses processes can use, so they can't stomp on each other
    - fast translation: memory lookups must be fast, in spite of the protection scheme
    - fast context switching: when switching between jobs, updating memory hardware (protection and translation) must be quick

Virtual addresses for multiprogramming

- 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) where referenced data lives
    - OS determines location in physical memory
  - instructions issued by CPU reference virtual addresses
    - e.g., pointers, arguments to load/store instructions, PC …
  - virtual addresses are translated by hardware into physical addresses (with some setup from OS)

- 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
  - Only that the program issues addresses in a virtual address space, and these must be translated 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

Some important terms to remember

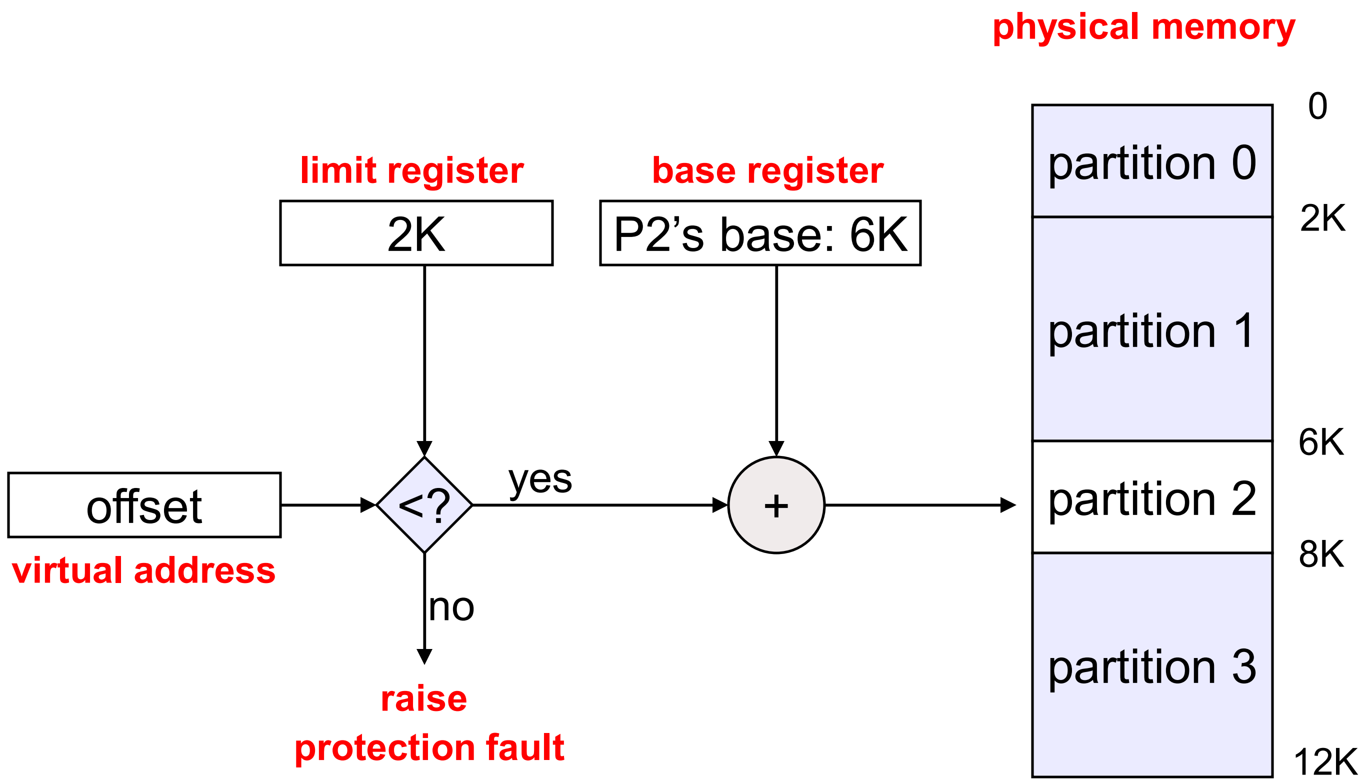
- Virtual Memory
- Virtual Address
- Address Space

- And now onto some oldish virtual memory model techniques…

Old technique #1: Fixed partitions

- Physical memory is broken up into fixed partitions
  - partitions may have different sizes, but partitioning never changes
  - hardware requirement: base register, limit register
    - physical address = virtual address + base register
    - base register loaded by OS when it switches to a process
  - how do we provide protection?
    - if (physical address > base + limit) then… ?
- Advantages
  - Simple
- Problems
  - internal fragmentation: the available partition is larger than what was requested
  - external fragmentation: two small partitions left, but one big job – what sizes should the partitions be??

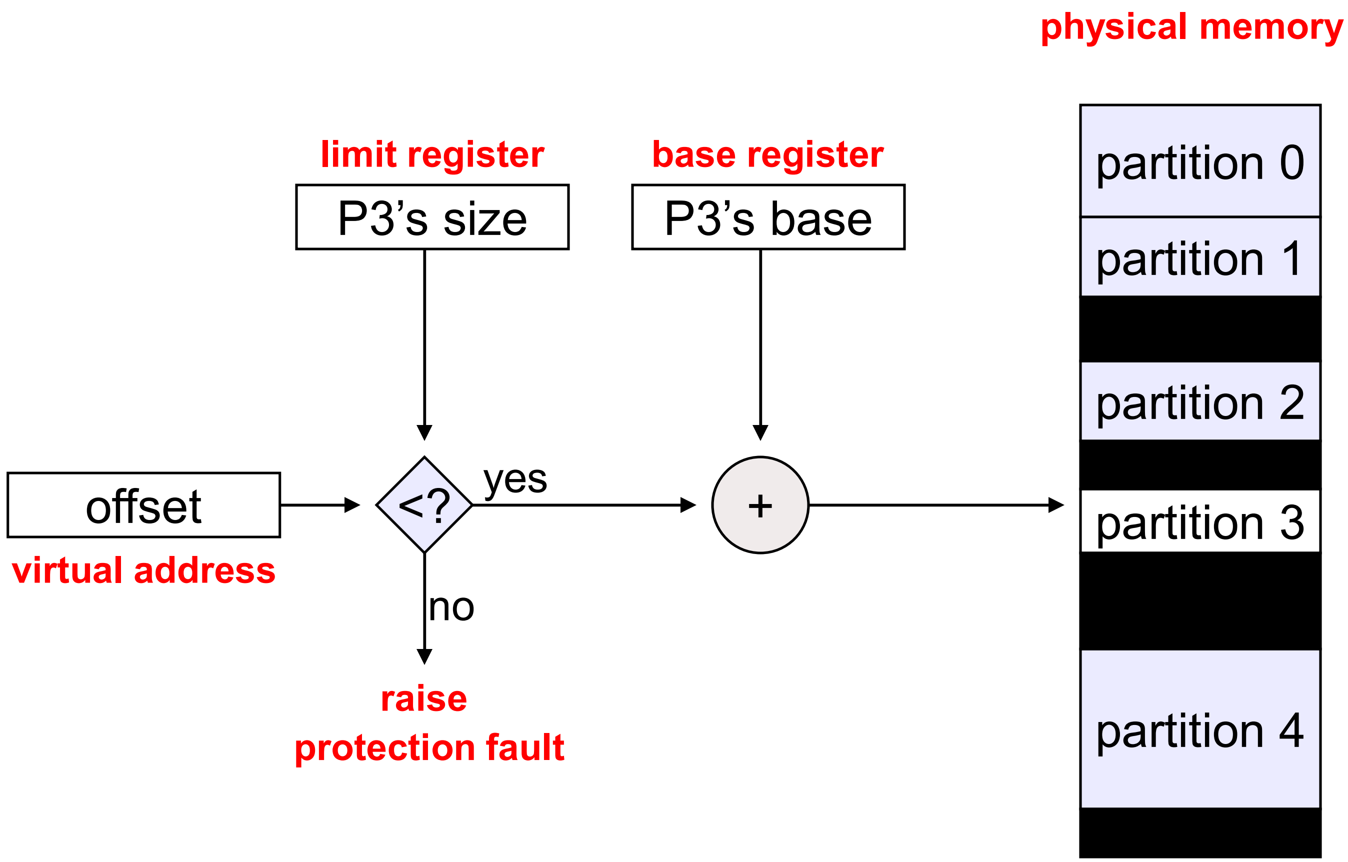
Mechanics of fixed partitions



Old technique #2: Variable partitions

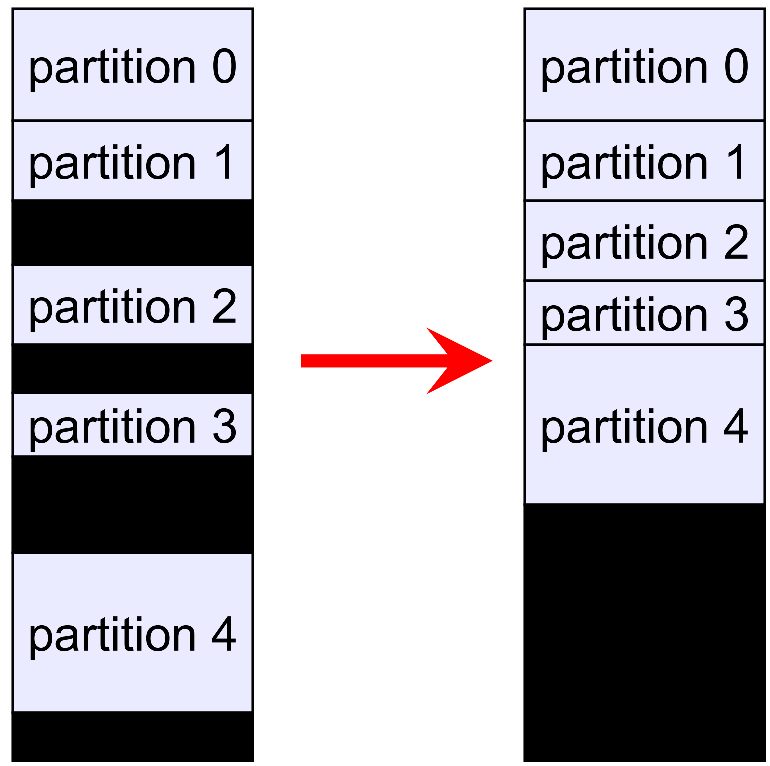
- Obvious next step: physical memory is broken up into partitions dynamically – partitions are tailored to programs
  - hardware requirements: base register, limit register
  - physical address = virtual address + base register
  - how do we provide protection?
    - if (physical address > base + limit) 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
    - slightly different than the external fragmentation for fixed partition systems

Mechanics of variable partitions



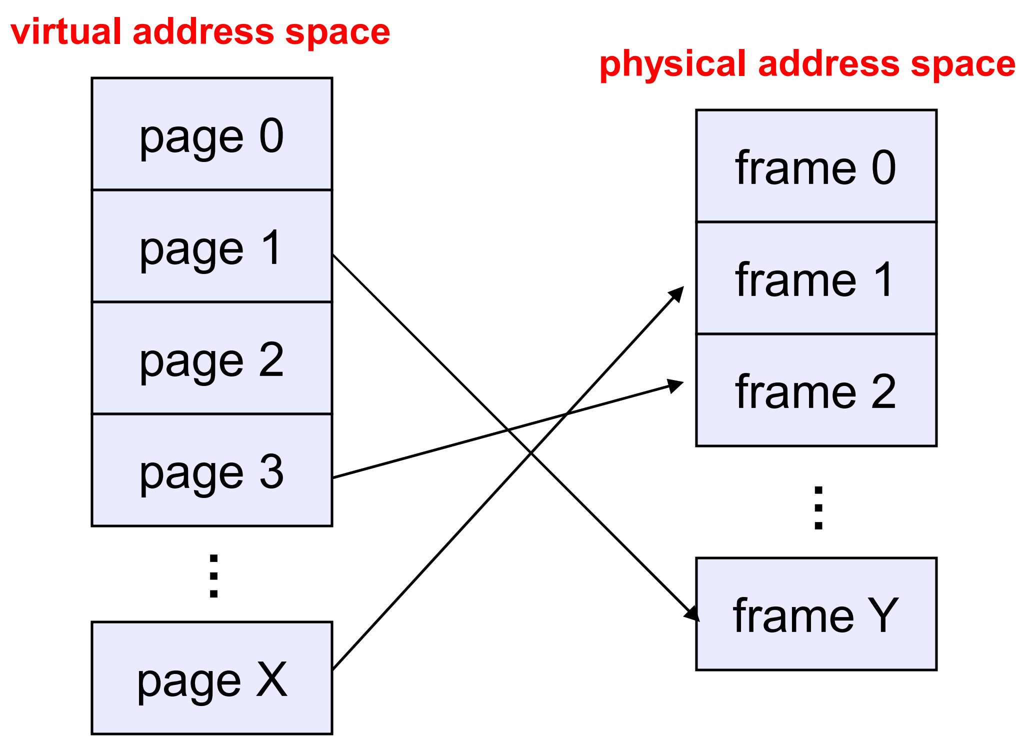
Dealing with fragmentation

- Compact memory by copying
  - 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
- Mitigate the internal fragmentation problem by making the units small



Life is easy …

- For the programmer …
  - Processes view memory as a contiguous address space from bytes 0 through N – a virtual address space
  - N is independent of the actual hardware
  - In reality, virtual pages are scattered across physical memory frames – not contiguous as earlier
    - Virtual-to-physical mapping
    - This mapping is invisible to the program
- For the memory manager …
  - Efficient use of memory, because very little internal fragmentation
  - No external fragmentation at all
    - No need to copy big chunks of memory around to coalesce free space

- For the protection system
  - One process cannot “name” another process’s memory – there is complete isolation
    - The virtual address 0xDEADBEEF maps to different physical addresses for different processes

Note: Assume for now that all pages of the address space are resident in memory – no “page faults”

- But how do we accomplish this translation from a virtual address to a physical address?

Address translation

- To go from a virtual address to a physical address, we add a level of indirection called a Page Table
- Translating virtual addresses
  - a virtual address has two parts: virtual page number & offset
  - virtual page number (VPN) is an index into a page table
  - page table entry contains page frame number (PFN)
  - physical address is PFN::offset (concatenated together)
- Page tables
  - managed by the OS
  - one page table entry (PTE) per page in virtual address space
    - i.e., one PTE per VPN
  - map virtual page number (VPN) to page frame number (PFN)
    - VPN is simply an index into the page table

Paging (K-byte pages)

process 0

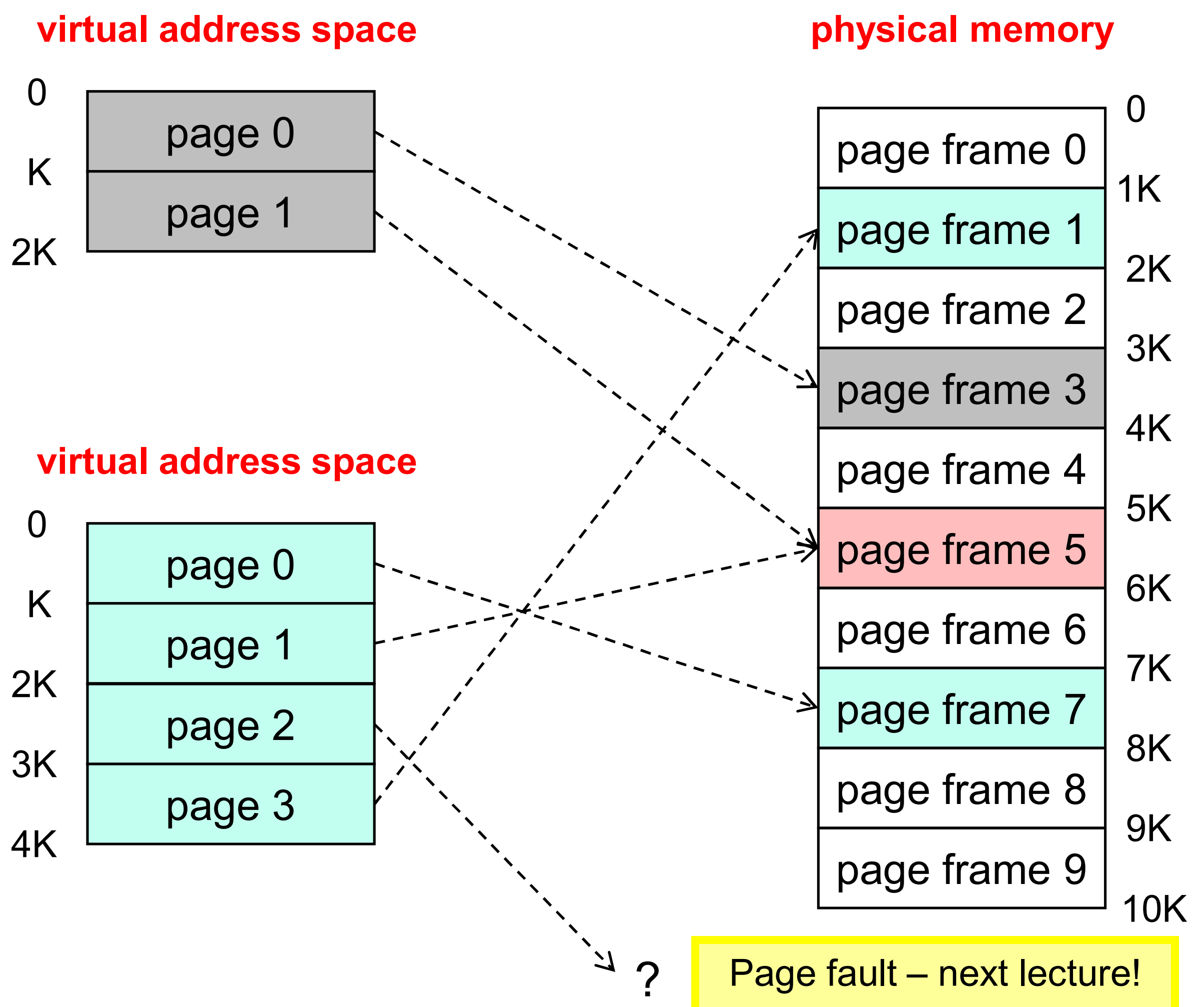
page table

page	frame
0	3
1	5

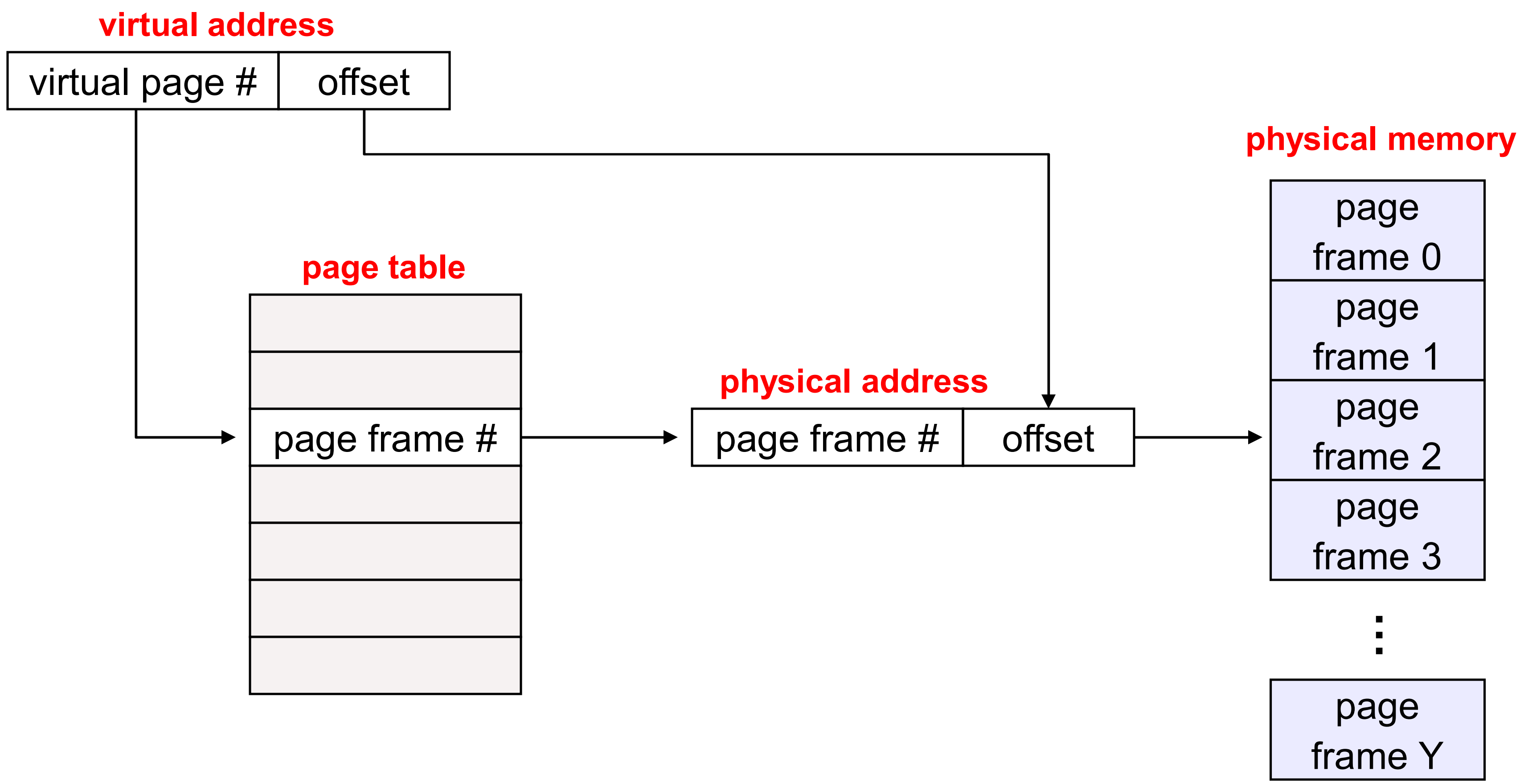
process 1

page table

page	frame
0	7
1	5
2	-
3	1



Mechanics of address translation

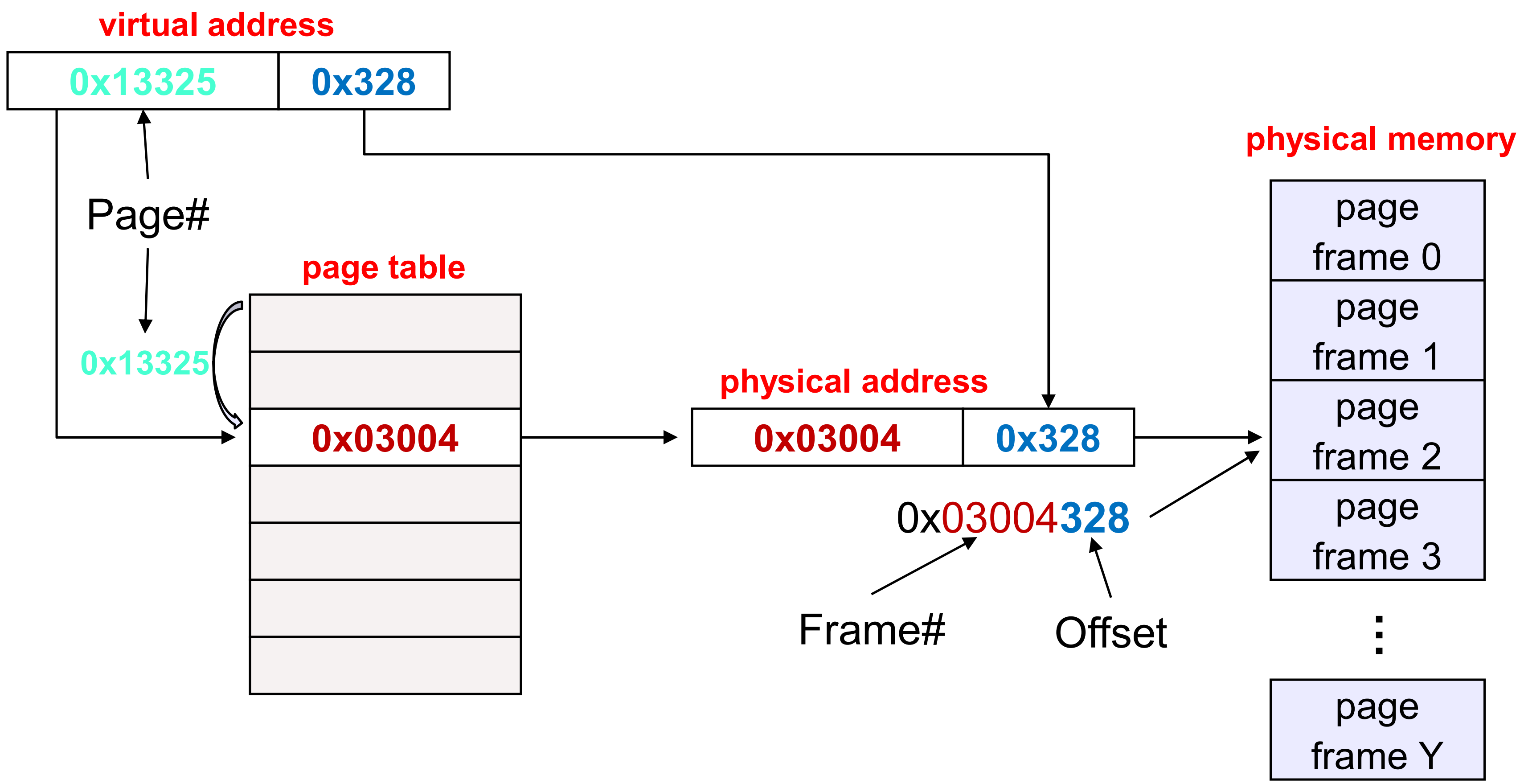


Example of address translation

- 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

Translating 0x13325328



Page Table Entries – an opportunity!

- As long as there's a PTE lookup per memory reference, we might as well add some functionality
  - We can add protection
    - A virtual page can be read-only, and result in a fault if a store to it is attempted
    - Some pages may not map to anything – a fault will occur if a reference is attempted
    - What other type of protection would be good? Maybe execute-only?
  - We can add some "accounting information"
    - Can't do anything fancy, since address translation must be fast
    - Can keep track of whether or not a virtual page is being used, though
      - This will help the paging algorithm, once we get to paging

Generic Page Table Entry (PTE)

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

- PTE's control mapping (a generic/stylized version)
  - 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 referenced bit says whether the page has been accessed
    - it is set when a page has been read or written to
  - 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
  - the page frame number determines the physical page
    - physical page start address = PFN

Paging advantages

- Easy to allocate physical memory
  - physical memory is allocated from free list of frames
    - to allocate a frame, just remove it from the free list
  - external fragmentation is not a problem
    - managing variable-sized allocations is a huge pain in the neck
      - “buddy system”
- Leads naturally to virtual memory
  - entire program need not be memory resident
  - take page faults using “valid” bit
  - all “chunks” are the same size (page size)
  - but paging was originally introduced to deal with external fragmentation, not to allow programs to be partially resident (here is the subtle distinction between virtual addressing and virtual memory)

Paging disadvantages

- Can still have internal fragmentation
  - Process may not use memory in exact multiples of pages
  - But minor because of small page size relative to address space size
- 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 have separate page tables per process
    - 25 processes = 100MB of page tables
  - Solution: page the page tables (!!!)
    - (ow, my brain hurts…more later)

Segmentation
(We will be back to paging soon!)

- Paging
  - mitigates various memory allocation complexities (e.g., fragmentation)
  - view an address space as a linear array of bytes
  - divide it into pages of equal size (e.g., 4KB)
  - use a page table to map virtual pages to physical page frames
    - page (*logical*) => page frame (*physical*)
  - One problem with paging is that is not very logical (i.e., programmer friendly)
- Segmentation
  - partition an address space into *logical* units
    - stack, code, heap, subroutines, …
  - a virtual address is <segment #, offset>

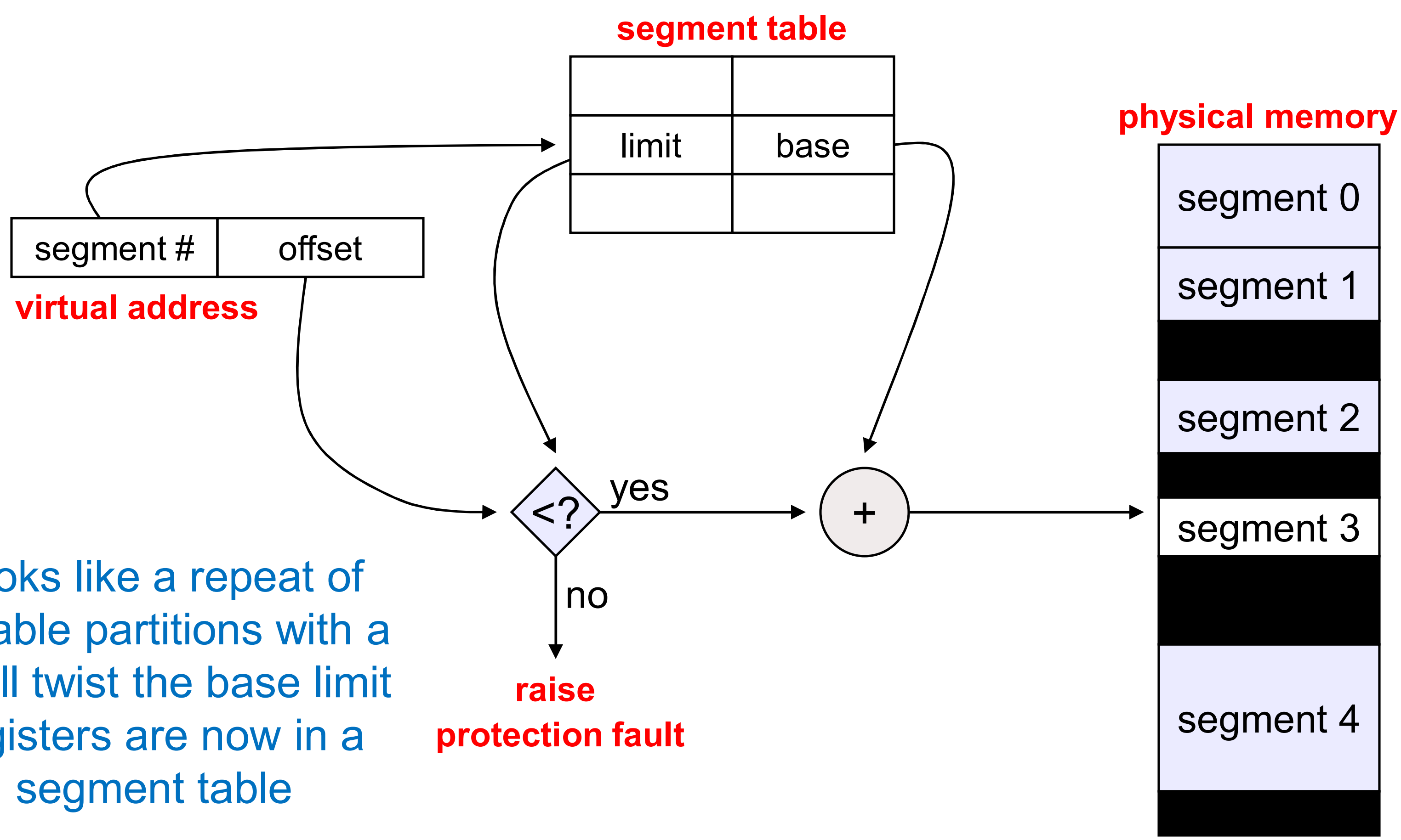
What's the point?

- More "logical"
  - absent segmentation, a linker takes a bunch of independent modules that call each other and linearizes them
  - they are really independent; segmentation treats them as such
- Facilitates sharing and reuse
  - a segment is a natural unit of sharing – a subroutine or function
- A natural extension of variable-sized partitions
  - variable-sized partition = 1 segment/process
  - segmentation = many segments/process

Hardware support

- Segment table
  - multiple base/limit pairs, one per segment
  - segments named by segment #, used as index into table
    - a virtual address is <segment #, offset>
  - offset of virtual address added to base address of segment to yield physical address

Segment lookups



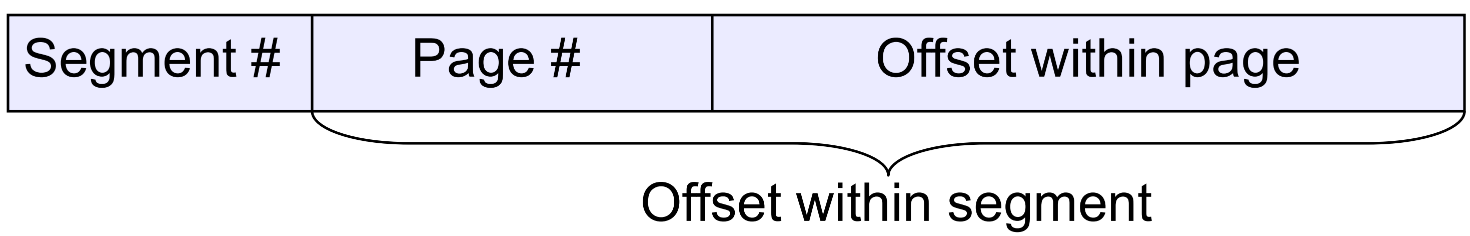
Looks like a repeat of variable partitions with a small twist the base limit registers are now in a segment table

Pros and cons

- Yes, it's "logical" and it facilitates sharing and reuse
- But it has all the horror of a variable partition system
  - except that linking is simpler, and the "chunks" that must be allocated are smaller than a "typical" linear address space
- What to do?

Combining segmentation and paging

- Can combine these techniques
  - modern architectures support both segments and paging
- Use segments to manage logical units
  - segments vary in size, but are typically large (multiple pages)
- Use pages to partition segments into fixed-size chunks
  - each segment has its own page table
    - there is a page table per segment, rather than per user address space
  - memory allocation becomes easy once again
    - no contiguous allocation, no external fragmentation



How Intel combines segments and pages

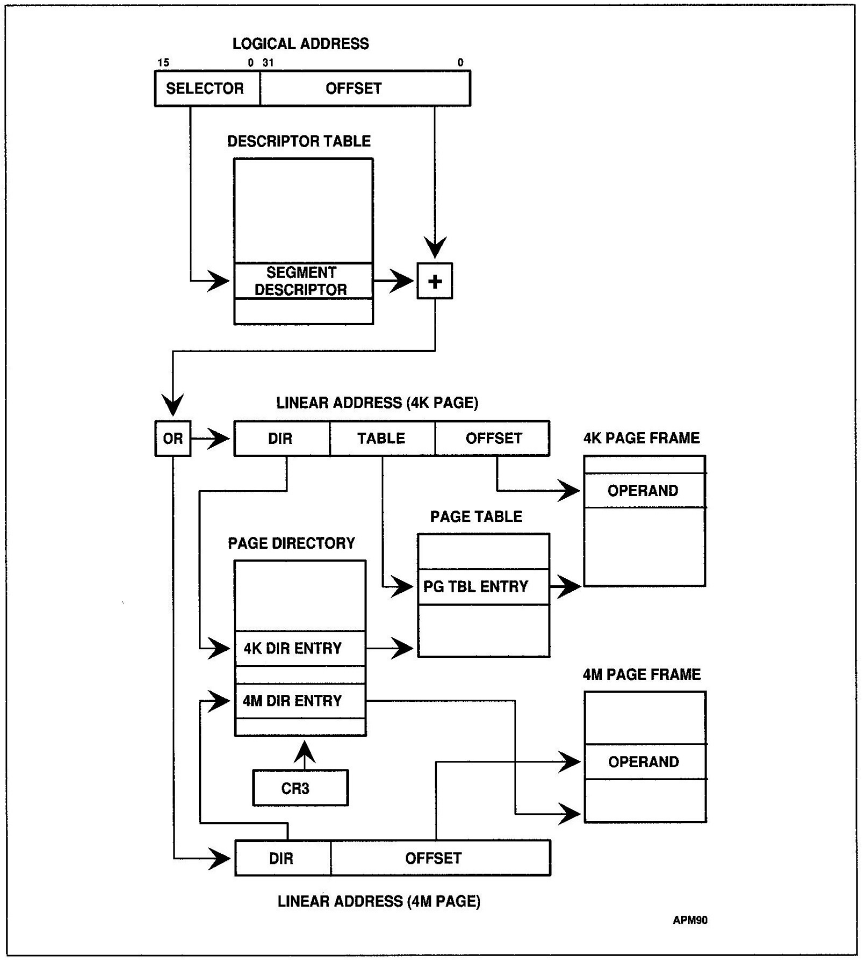


Figure 11-16. Combined Segment and Page Address Translation

Mixing segments and pages

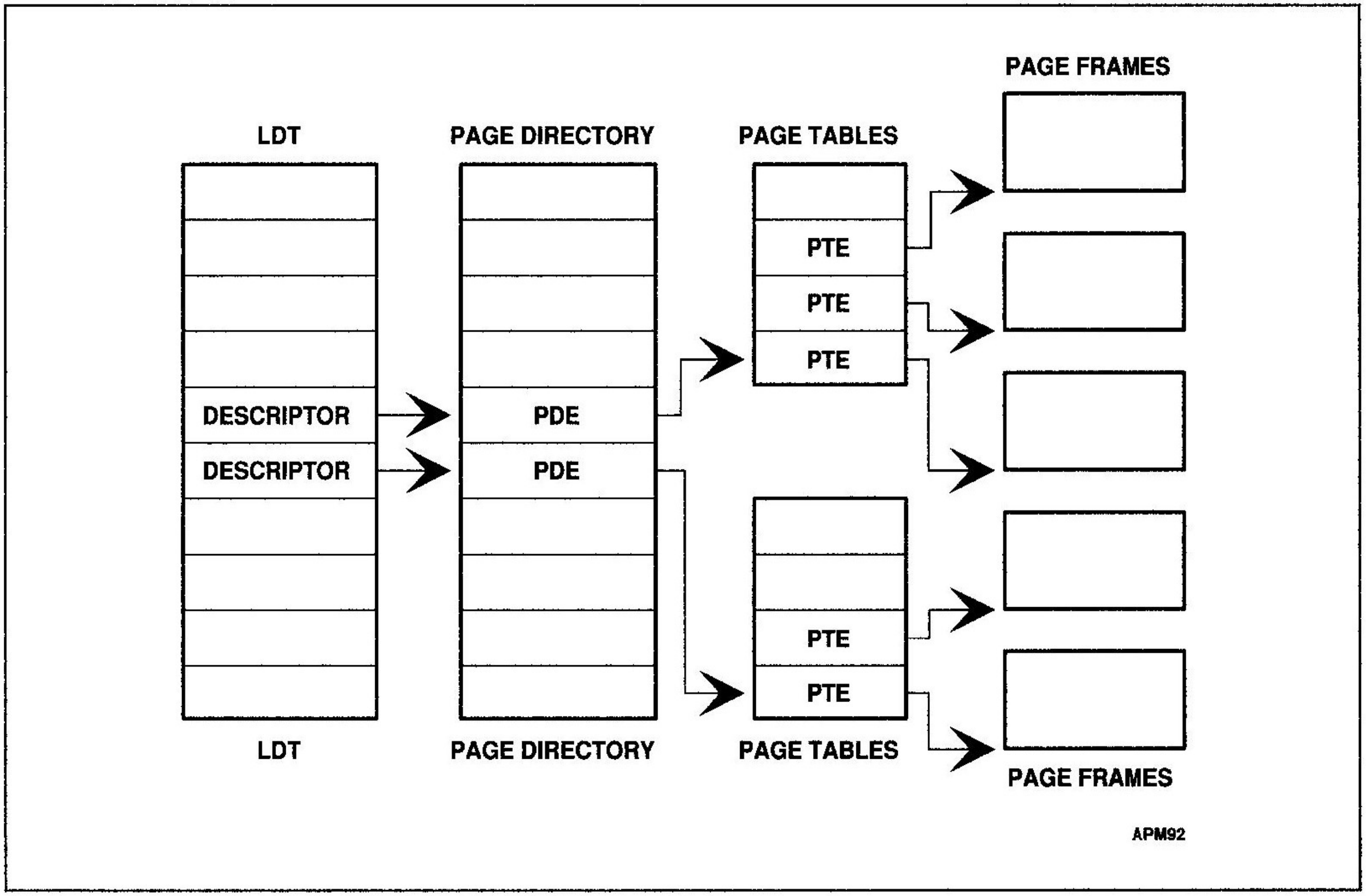


Figure 11-17. Each Segment Can Have Its Own Page Table

- Linux:
  - 1 kernel code segment, 1 kernel data segment
  - 1 user code segment, 1 user data segment
  - all of these segments are paged
- Note: this is a very limited/boring use of segments!