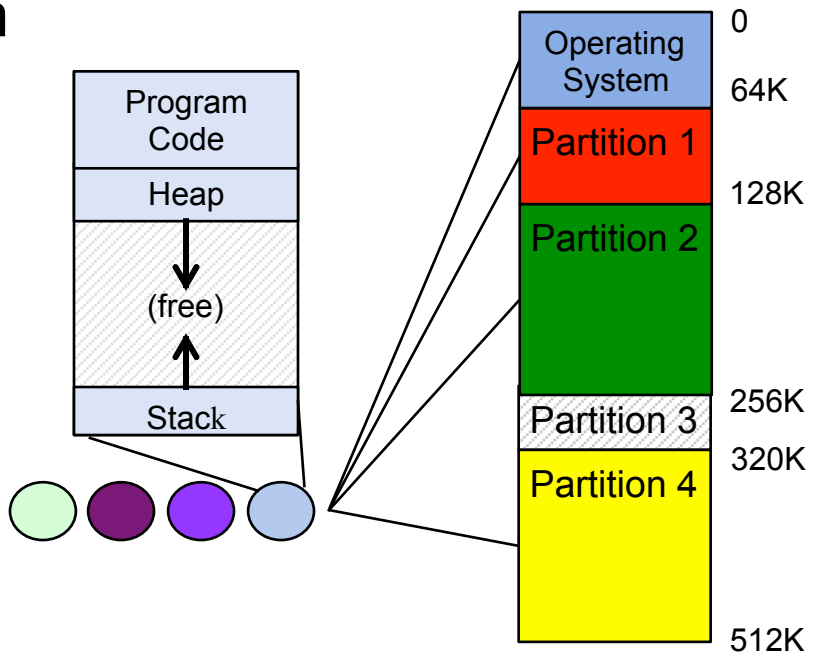
Virtual Memory 1

To do ...

- Segmentation
- Paging
- A hybrid system

Address spaces and multiple processes

- IBM OS/360
  - Split memory in *n* parts (possible != sizes)
  - A process per partition



- *How do deal with large address spaces with potentially a lot of free space?*

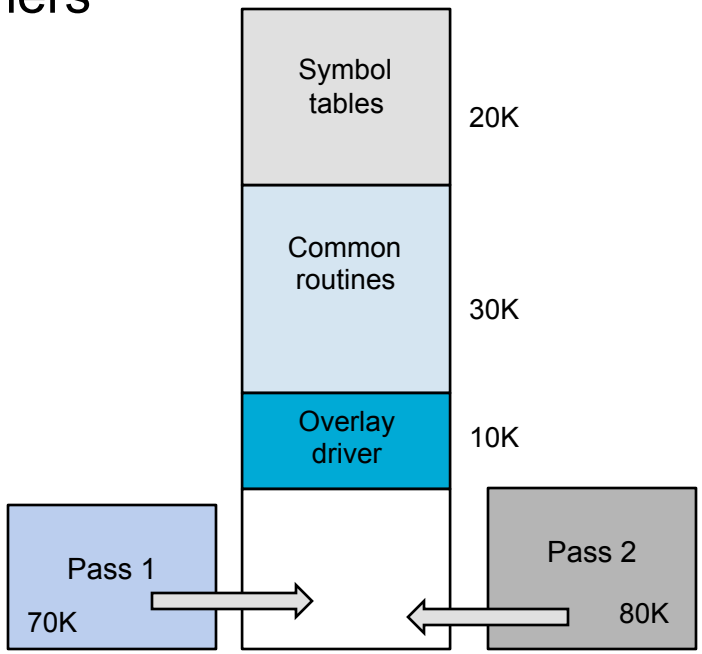
The overlay solution

- Split programs into pieces (overlays), keep in memory only what's needed
- Overlay approach (~1960s)
  - Swapping was done by the OS, but
  - Splitting was done manually by the user
  - Easy on the OS, hard on programmers

Overlay for a two-pass assembler:

Pass 1	70KB
Pass 2	80KB
Symbol Table	20KB
Common Routines	30KB
Total	200KB

Two overlays: 120 + 130KB

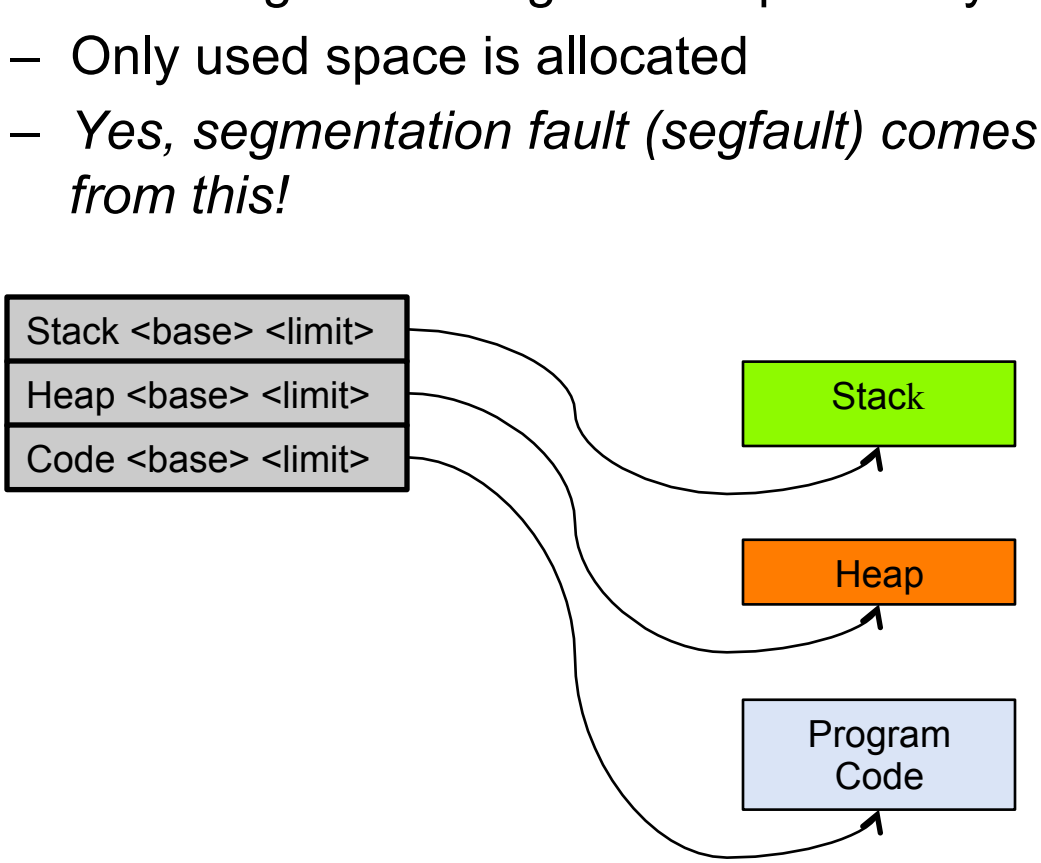


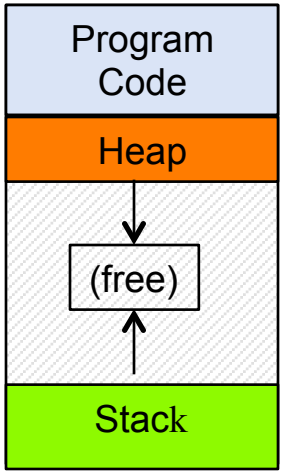
Virtual memory – Goals

- Transparency
  - Programs should not know that memory is virtualized
  - ***Program flexibility – processes can run in machines with less physical memory than they need***
- Efficiency
  - Both in time and space; not making processes too slow and using physical memory efficiently
- Protection
  - Isolating processes from each other; i.e., a process should not be able to access the memory of any other or the OS itself

Virtual memory – Segmentation

- Hide the complexity, let the OS do the job
- *How?* A related idea, to generalize base/limit
  - Each segment can grow independently
  - Only used space is allocated
  - *Yes, segmentation fault (segfault) comes from this!*

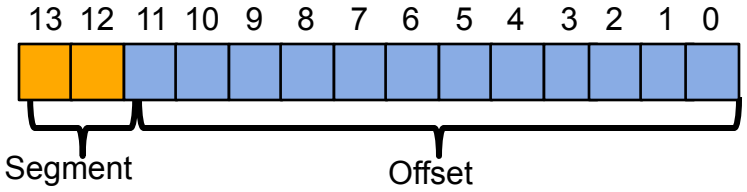




Which segment?

- How does the HW know which segment an address refers to? Explicit approach
  - With three segments, use two bits of the virtual address
  - The rest are offset into the segment

```
// Get segment number
Segment = (VirtAddr & SEG_MASK) >> SEG_SHIFT;
// Now get offset
Offset = VirtAddr & OFFSET_MASK;
if (Offset >= Limit[Segment])
        /* protection fault */
else
   PhysAddr = Base[Segment] + Offset
   ...
```



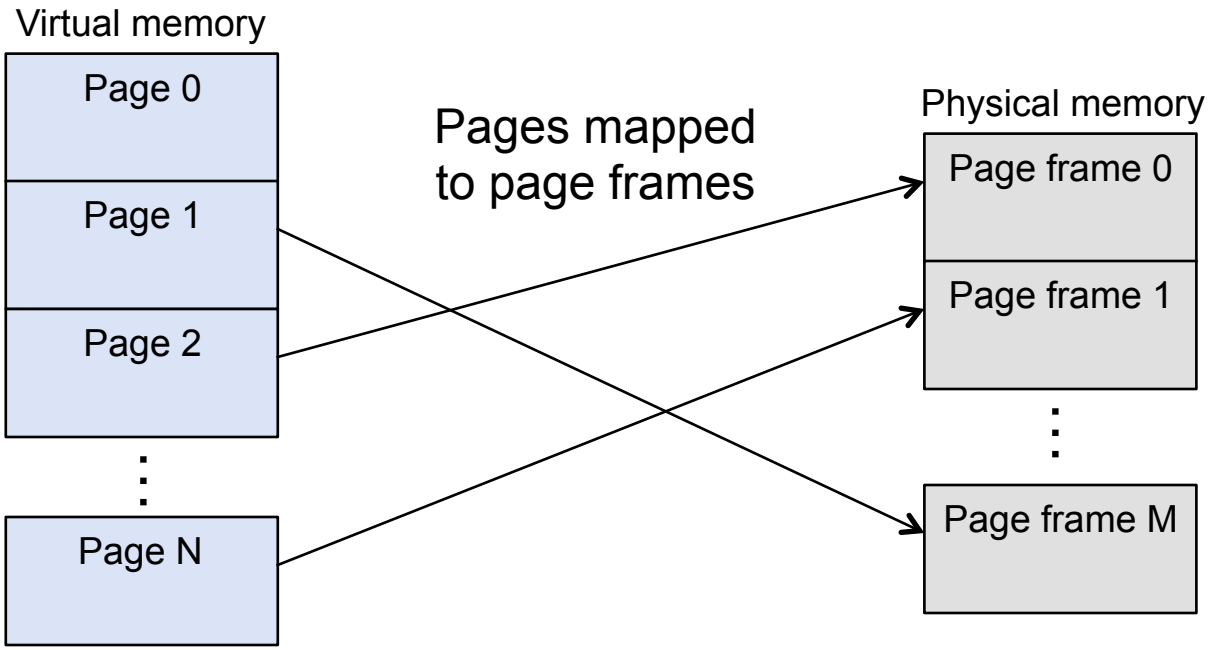
- Implicit – how was the address formed?
  - PC? Code segment; from the stack or base pointer? Stack ...

Sharing and fragmentation

- With a little bit more HW support – sharing memory segments (e.g., code)
  - Just a few bits per segments for read/write/execute rights
- Rather than code, stack, … finer grain segmentation
  - OS can keep track of segments in use and manage memory more efficiently
  - A segment table to track them
  - Need more HW support
- Many small variable-sized segments …
  - External fragmentation!
  - Compaction is expensive, free-list management may help a bit

And then came paging

- Avoid fragmentation from variable-sized blocks
  - Use fixed-sized blocks – pages
- Virtual address space split into pages
  - Each a contiguous range of addresses
- Physical memory split into page frames



Virtual memory – paging

- Pages and page frames are, typically, = size
  - Many processors support multiple page sizes (e.g., x86-64: 4KB, 2MB, 1GB)

- Pages are mapped onto frames
  - Doing the translation – OS + MMU
  - Key – not all pages have to be in at once
  - If page is in memory,

    system does the mapping

    else,

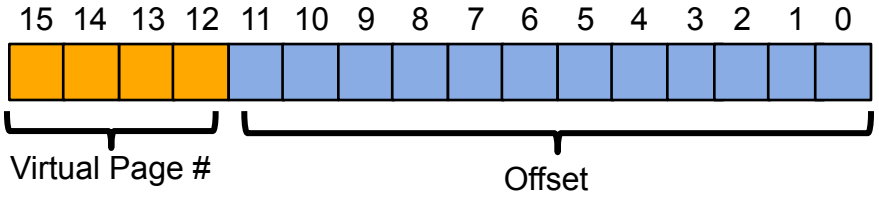
    OS is told to get the missing page and re-execute the failed instruction

Paging – Good for everyone

- For developers
  - Memory seems a contiguous address space with size independent of hardware
  - Simple and flexible – no assumptions on how memory is used
- For memory manager
  - Physical memory can be used efficiently with minimal internal (small units) & no external fragmentation (fixed size units)
- Protection is easy since processes can't access each other's memory

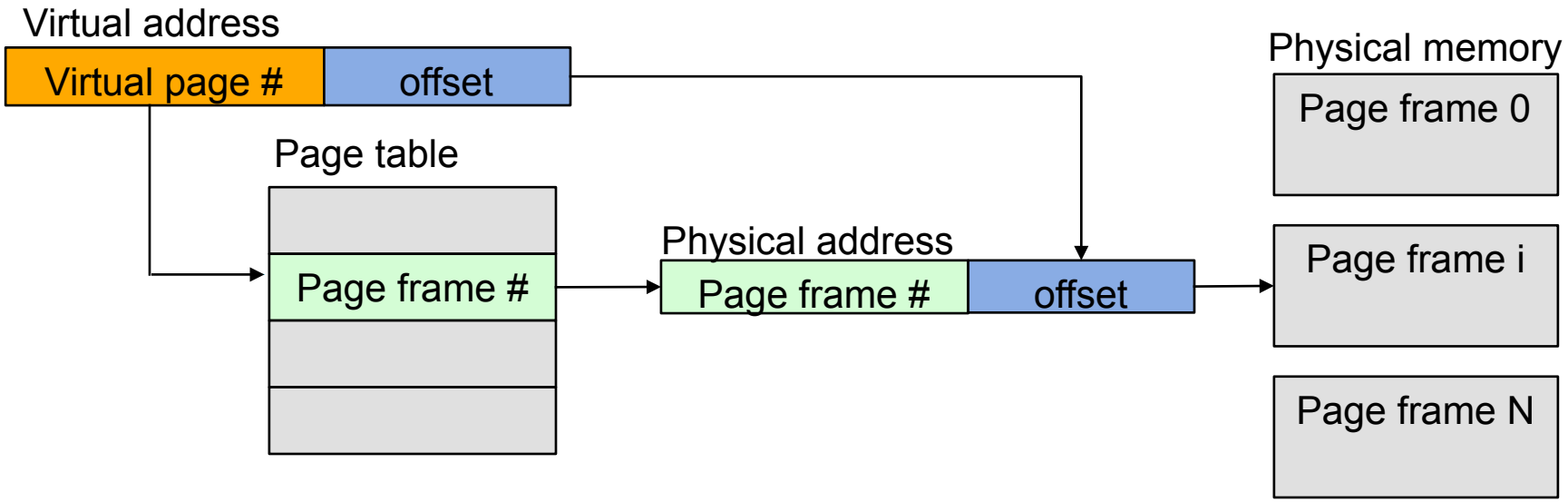
Address translation with paging

- Virtual to physical address
  - Two parts – virtual page number and offset



  - Virtual page number – index into a page table
  - Page table maps virtual pages to page frames
    - Managed by the OS
    - One entry per page in virtual address space
  - Physical address – page number and offset

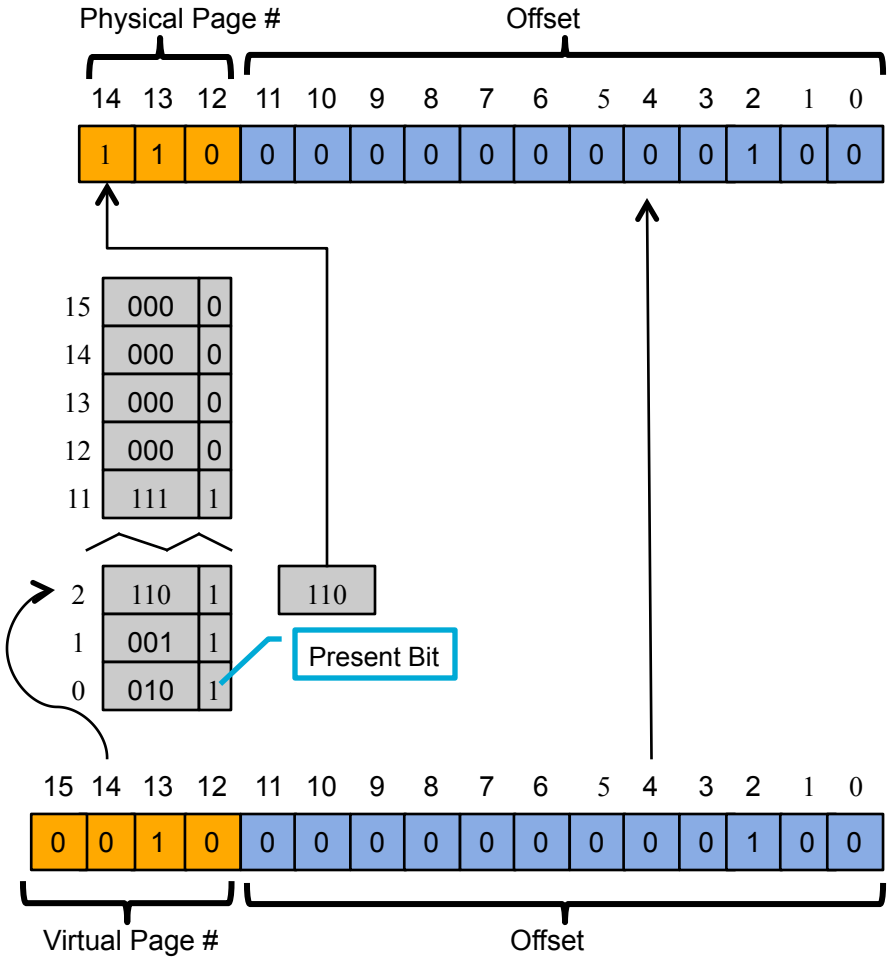
Address translation with paging



Each process has its own page table

Translation in action

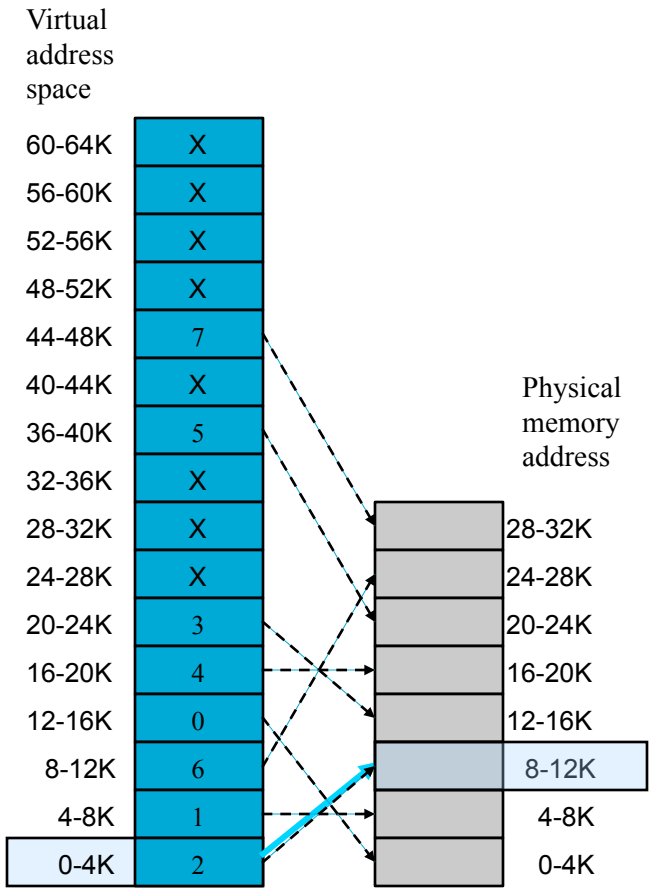
- MMU with 16 4KB pages
- Page # (first 4 bits) index into page table
- If not there
  - Page fault
- Else
  - Output register +
  - 12 bit offset →
  - 15 bit physical address



Pages, page frames and tables

A simple example with

- 64KB virtual address space
- 4KB pages
- 32KB physical address space
- 16 pages and 8 page frames



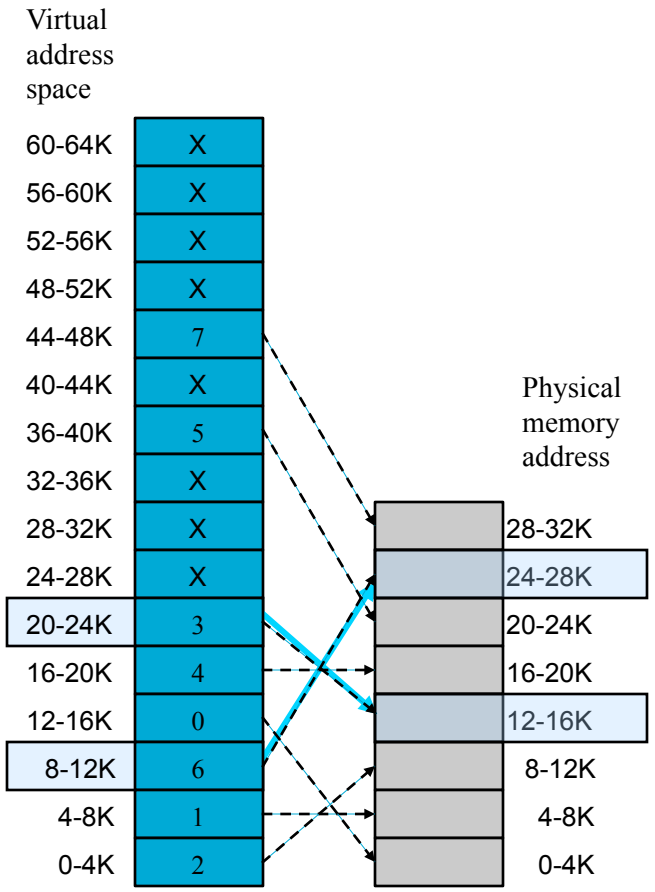
Try to access :

- **MOV REG, 0**
  Virtual address 0
  Page frame 2
  Physical address 8192

Pages, page frames and tables

A simple example with

- 64KB virtual address space
- 4KB pages
- 32KB physical address space
- 16 pages and 8 page frames



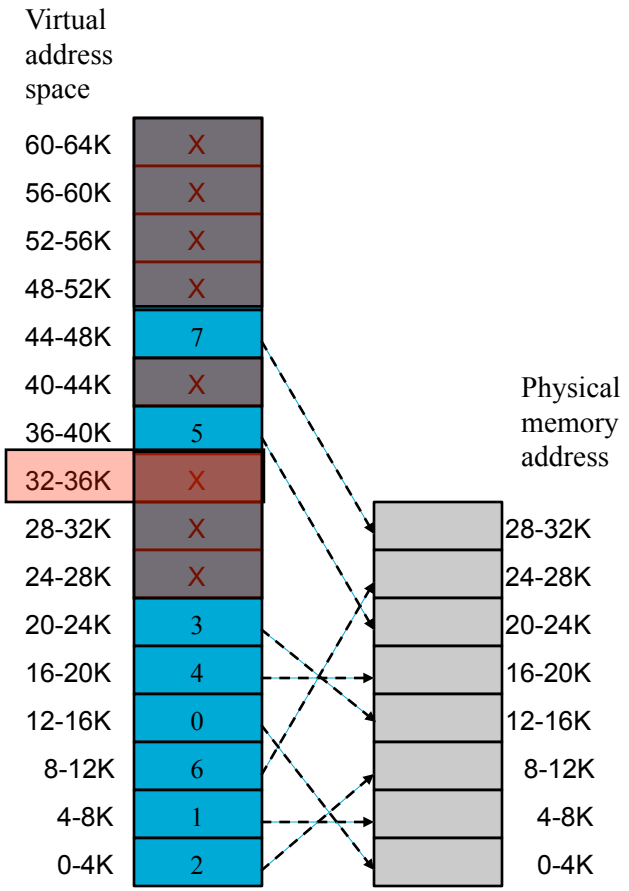
Try to access :

- **MOV REG, 8192**
  - **Virtual address 8192**
  - **Page frame 6**
  - **Physical address 24576**

- **MOV REG, 20500**
  - **Virtual address 20500 (20480 + 20)**
  - **Page frame 3**
  - **Physical address 20+12288**

Since virtual memory >> physical memory

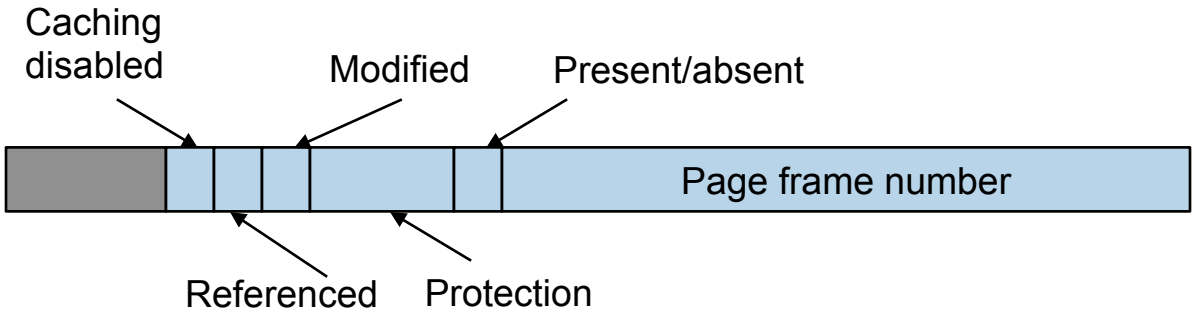
- Use a present/absent bit
- MMU checks –
  - If not there, “page fault” to the OS (trap)
  - OS picks a victim (?)
  - … sends victim to disk
  - … brings new one
  - … updates page table



MOVE REG, 32780
Virtual address 32780
Virtual page 8, byte 12 (32768+12)
Page is unmapped – page fault!

Page table entry

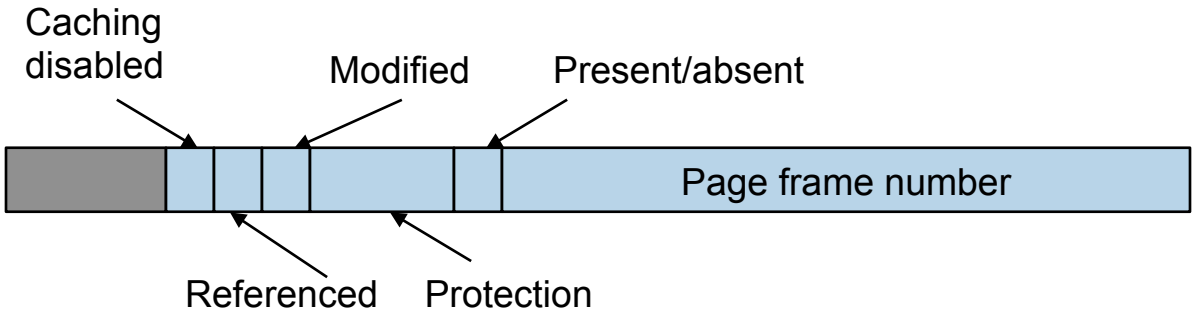
- An opportunity – there's a PTE lookup per memory ref., what else can we do with it?
- Looking at the details



  - Page frame number – the most important field
  - Protection – 1 bit for R&W or R or 3 bits for RWX

Page table entry

- Looking at the details



  - Present/absent bit
    - Says whether or not the virtual address is used
  - Modified (M): dirty bit
    - Set when a write to the page has occurred
  - Referenced (R): Has it being used?
  - To ensure we are not reading from cache (D)
    - Key for pages that map onto device registers rather than memory

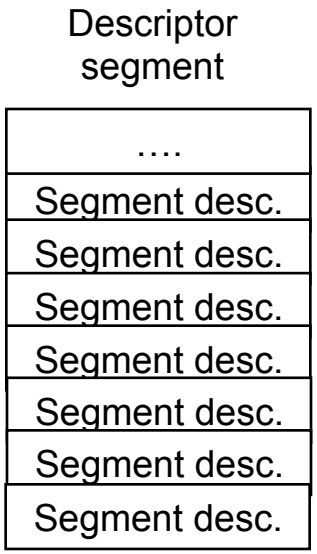
Segmentation and paging

- Segmentation pros and cons
  - ✔ It's more logical
  - ✔ Facilitates sharing and reuse
  - ✗ But all the problems of variable partitions

- Paging pros and cons
  - ✔ Easy to allocate physical memory
  - ✔ Naturally leads to virtual memory
  - ✗ Address translation time
  - ✗ Page tables can be large

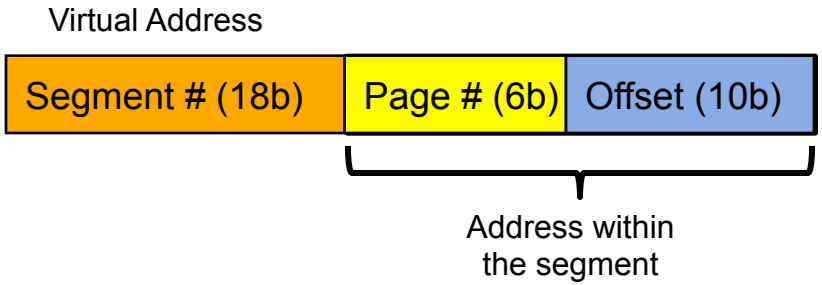
Segmentation w/ paging - MULTICS

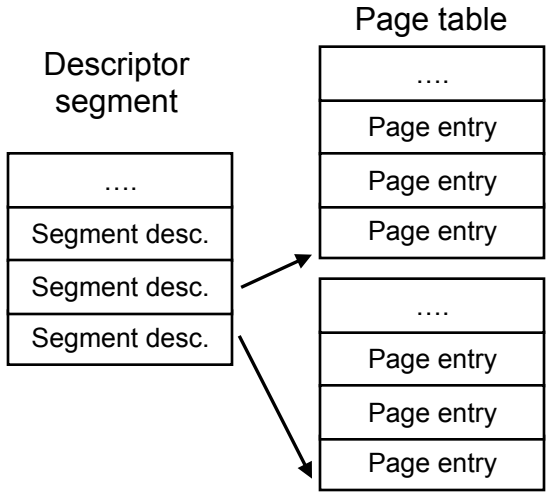
- *Large segment?* Page them
  - MULTICS, x86 and x86-64
- MULTICS: process' virtual memory – 2^{18} segments of ~64K words (36-bit)
  - Each process has a segment table (itself a paged segment)
  - Segment table address found in a Descriptor Base Register; one entry per segment



Segmentation w/ paging - MULTICS

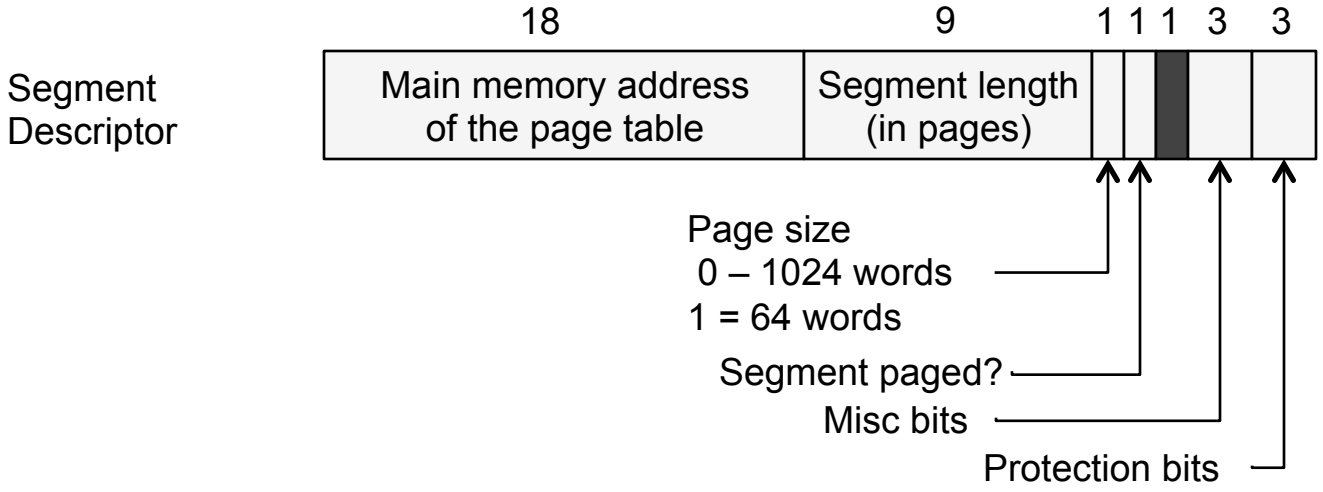
- Most (but not all) segments are paged
- Segment descriptor indicates if in memory and points to page table
  - Address of segment in secondary memory in another table
- Segment # to get segment descriptor





Segmentation w/ paging - MULTICS

- If segment in mem, segment's page table is in memory
- Protection violation? Fault
- Look at the page table's entry - is page in memory? Page fault
- Add offset to page origin to get word location
- … to speed things up – cache (TLB, first system)



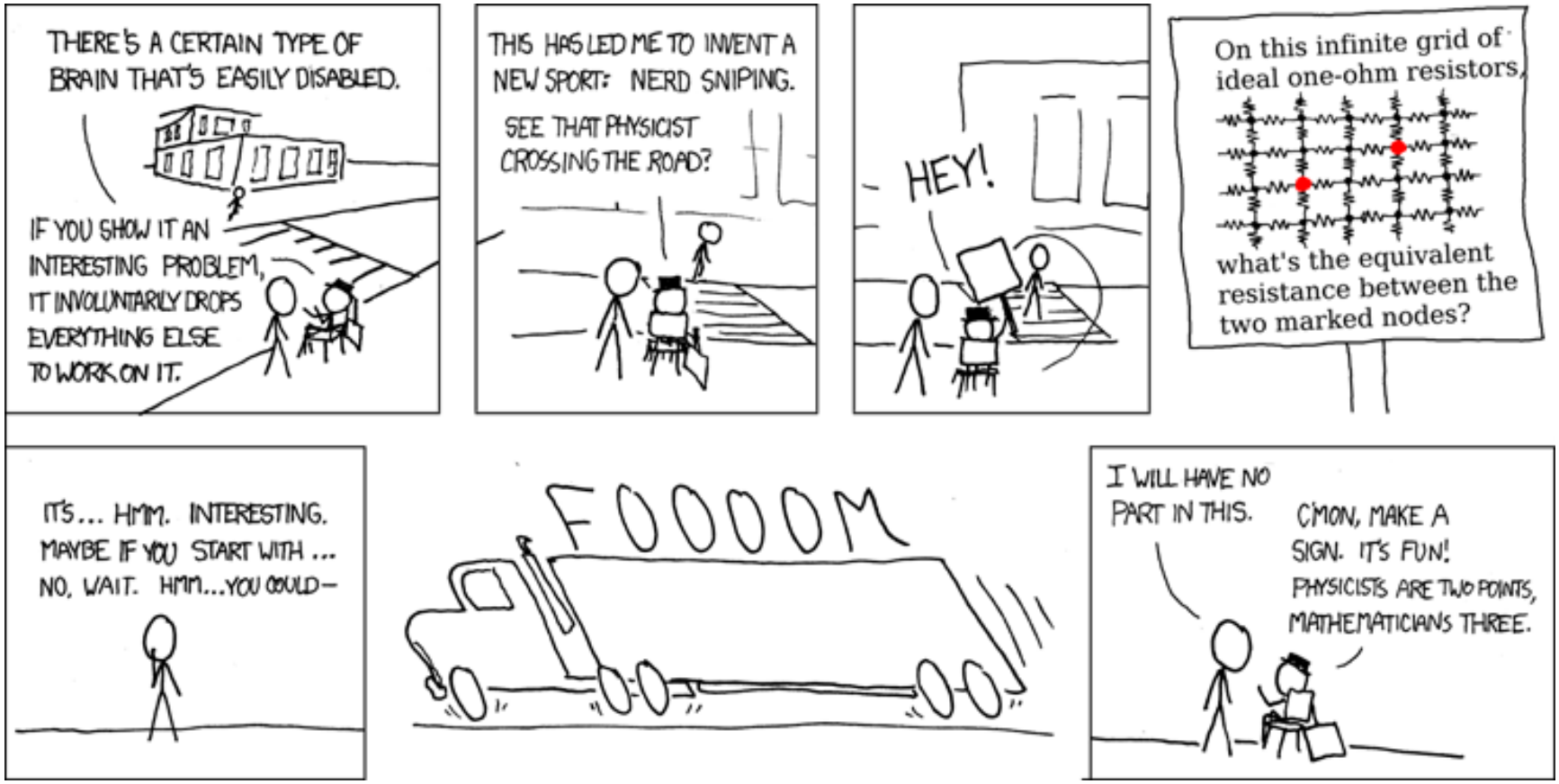
Next … considerations with page tables

Two key issues with page tables

- Mapping must be fast
  - Done on every memory reference, at least 1 per instruction
- With large address spaces, page tables will be large
  - 32b & 4KB page → 12 bit offset, 20 bit page # ~ 1million PTE
  - 64b & 4KB page → 2^{12} (offset) + 2^{52} pages ~ 4.5×10^{15}!!!

- Simplest solutions
  - Page table in registers
    - Fast during execution, $$$ & slow to context switch
  - Page table in memory & Page Table Base Register (PTBR)
    - Fast to context switch & cheap, but slow during execution

And now a short break …



xkcd