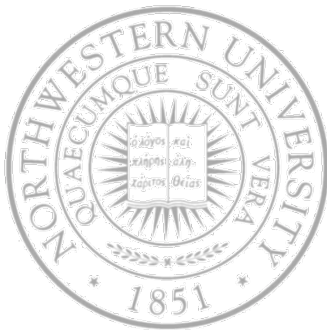


Virtual Memory



Today

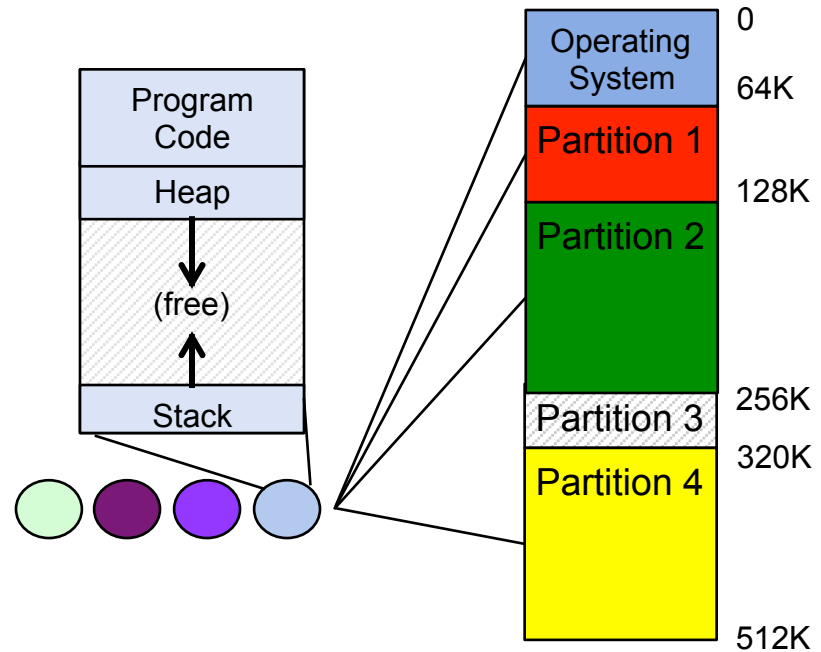
- Segmentation
- Paging
- A good, if common example

Virtual memory system – Goals

- Transparency
 - Programs should not know that memory is virtualized; the OS +HW multiplex memory among processes behind the scenes
- Efficiency
 - Both in time and space; not making processes too slow and efficiently using physical memory
- Protection
 - Isolating the address spaces of processes from each other; i.e., a process should not be able to access or affect the memory of any other process or the OS itself

Virtual memory

- IBM OS/360
 - Split memory in n parts (possible \neq sizes)
 - A process per partition



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

Virtual memory system – Goals

- Transparency
 - Programs should not know that memory is virtualized; the OS +HW multiplex memory among processes behind the scenes
 - ***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 efficiently using physical memory
- Protection
 - Isolating the address spaces of processes from each other; i.e., a process should not be able to access or affect the memory of any other process or the OS itself

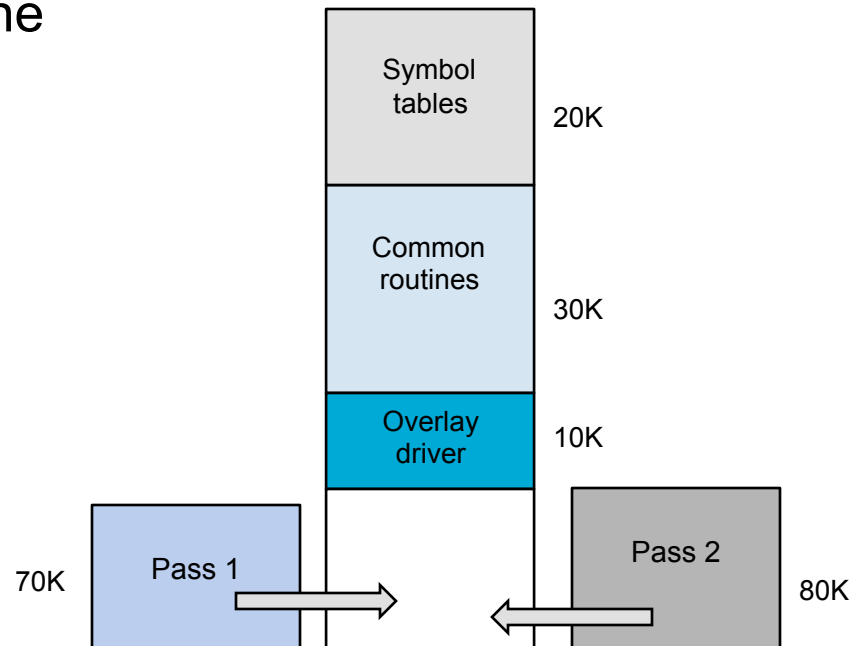
Virtual memory

- Keep in memory only what's needed
 - Don't need full address space resident in memory
 - OS uses main memory as a cache
- Overlay approach (~1960s)
 - Implemented by user
 - Easy on the OS, hard on the programmer

Overlay for a two-pass assembler:

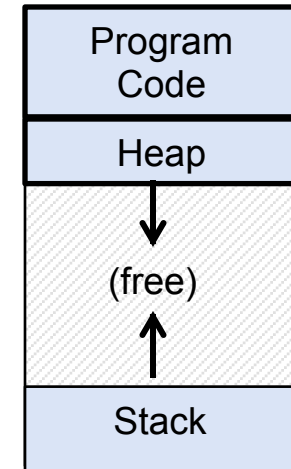
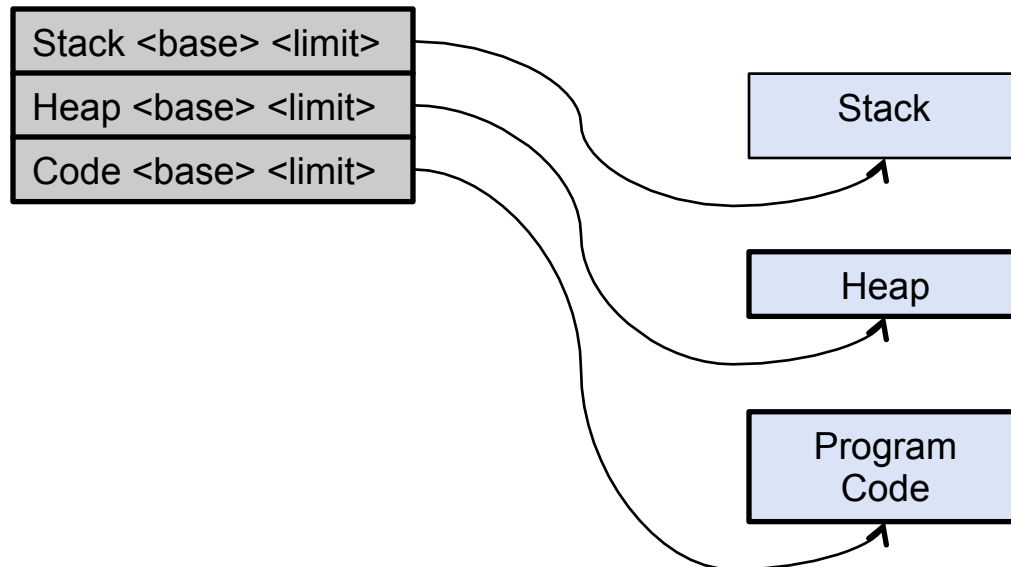
Pass 1	70KB
Pass 2	80KB
Symbol Table	20KB
Common Routines	30KB
Total	<u>200KB</u>

Two overlays: 120 + 130KB



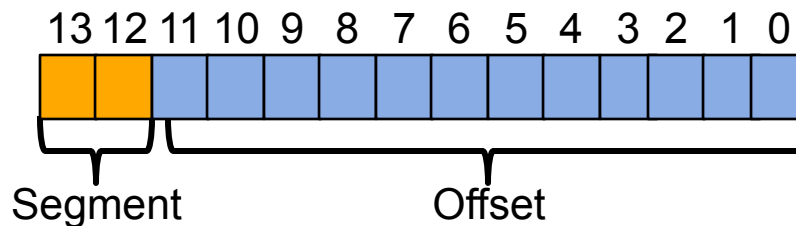
Segmentation

- Hide the complexity, let the OS do the job
- Related idea, generalize base/limit
 - Only used space is allocated
 - Each segment can grow independently
 - 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



Which segment?

- Explicit approach
 - Assuming base+limits were in an array

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

- Implicit approach – how was the address form? PC? from the stack or base pointer?

Sharing and fragmentation

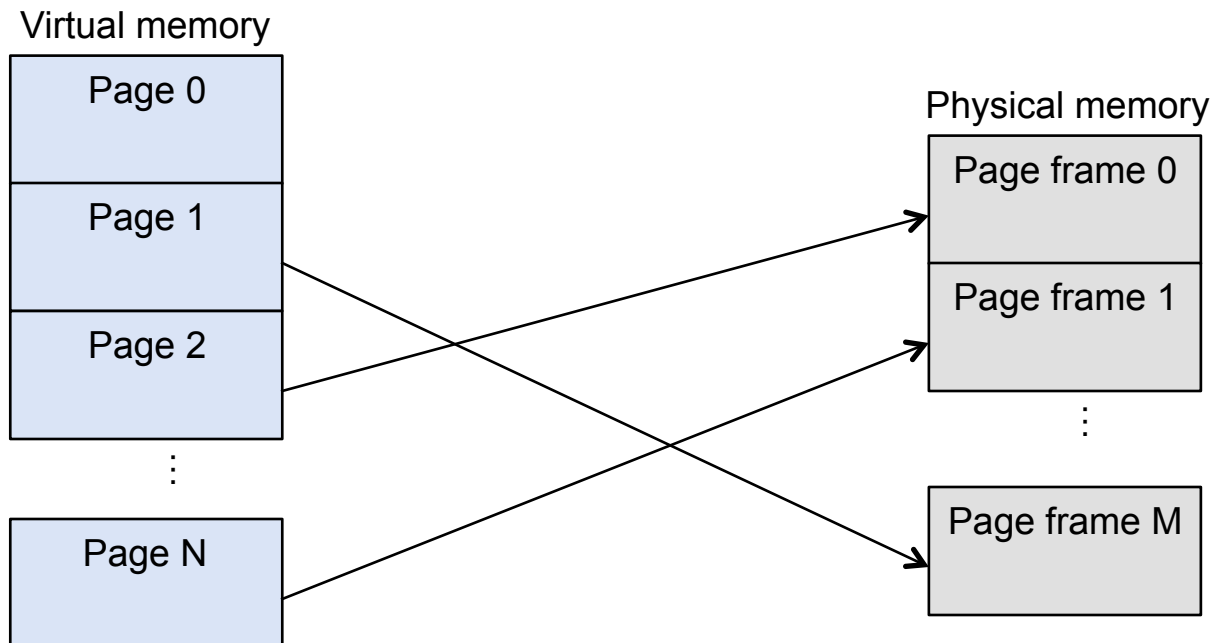
- With a little bit of hardware support – sharing memory segments (e.g., code)
 - Just a few bits per segments for read/write/execute rights
- Finer grain segmentation (e.g., Multics, Burroughs B5000)
 - Need a segment table to track them
 - The OS could keep track of segments in use and manage memory more efficiently

Sharing and fragmentation

- ...
- Many small variable-sized segments ...
 - External fragmentation!
 - Compaction is expensive
 - Free-list management algorithms (e.g., best fit, worst fit, ...) may help but not avoid it

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

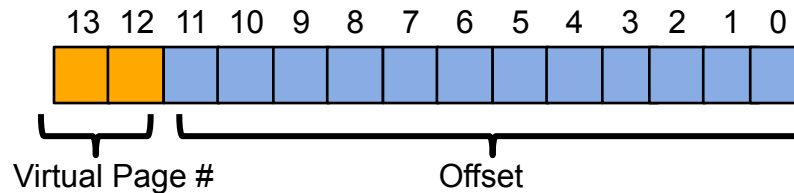
- Pages and page frames are generally = size
 - Many processors support multiple page sizes (e.g., x86-64: 4KB, 2MB, 1GB)
- Pages are mapped onto frames
 - Doing the translation – OS + MMU
 - Key – not all pages have to be in at once
 - If page is in memory,
 - system does the mapping
 - else,
 - OS is told to get the missing page and re-execute the failed instruction

Virtual memory – paging

- Good for everyone
 - Developers – memory seems a contiguous address space with size independent of hardware
 - Simple and flexible – no assumptions on how memory is used
 - Mem manager can efficiently use mem with minimal internal (small units) & no external fragmentation (fixed size units)
 - Protection since processes can't access each other's memory

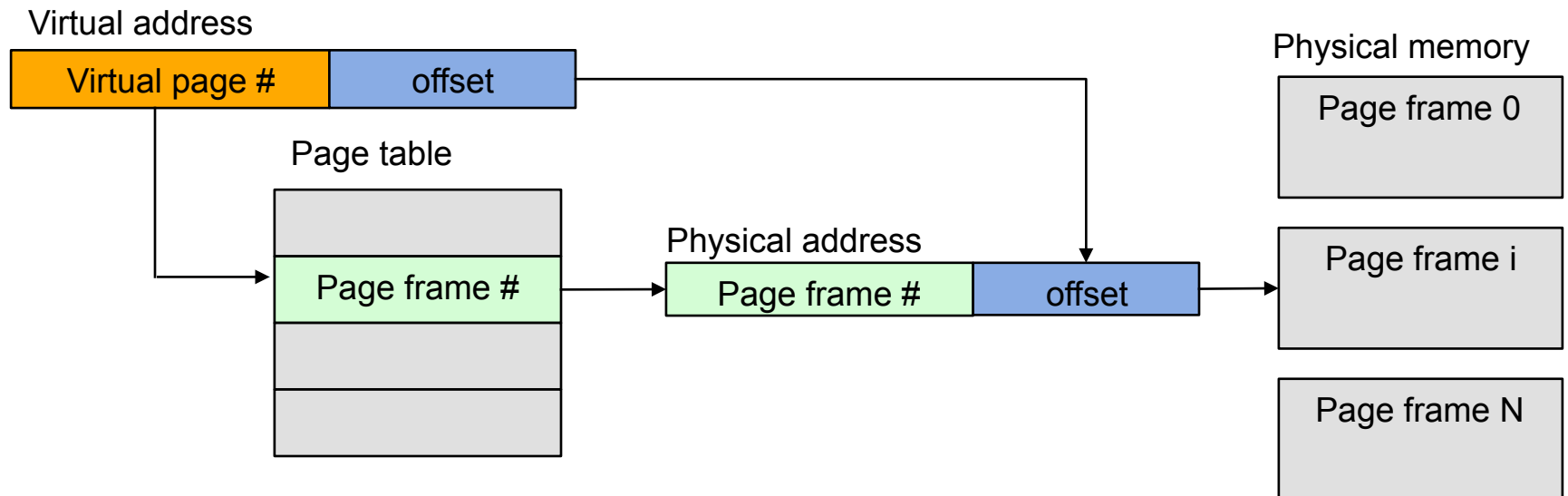
Address translation with paging

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



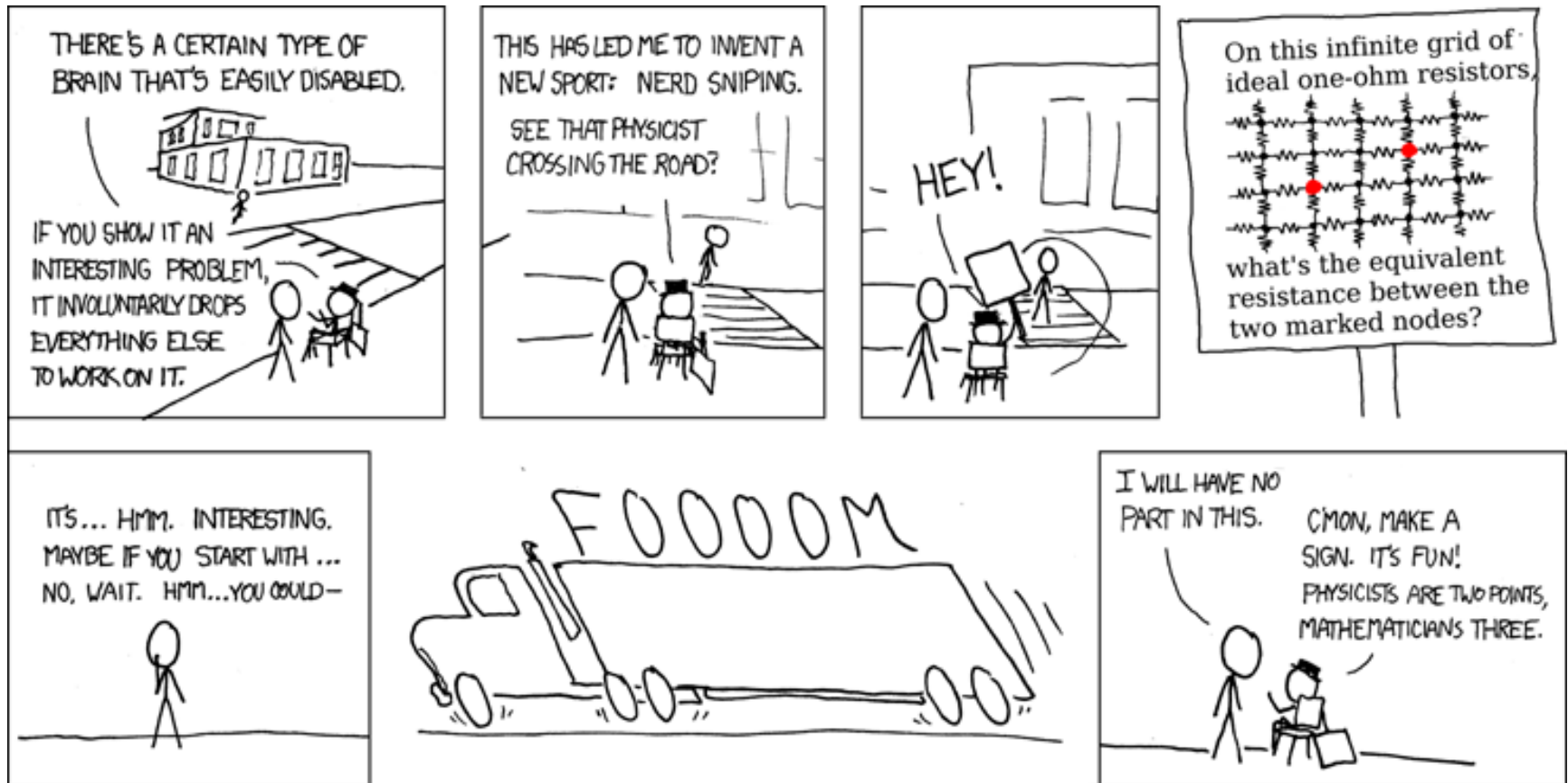
- Virtual page number – index into a page table
- Page table maps virtual pages to page frames
 - Managed by the OS
 - One entry per page in virtual address space
- Physical address – page number and offset

Address translation with paging



Each process has its own page table

And now a short break ...



xkcd

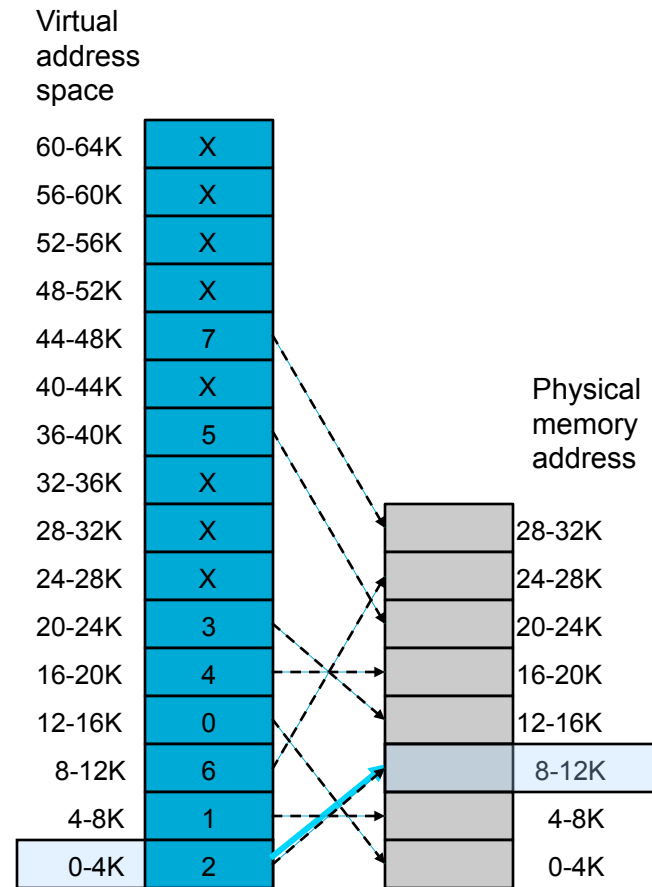
Pages, page frames and tables

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

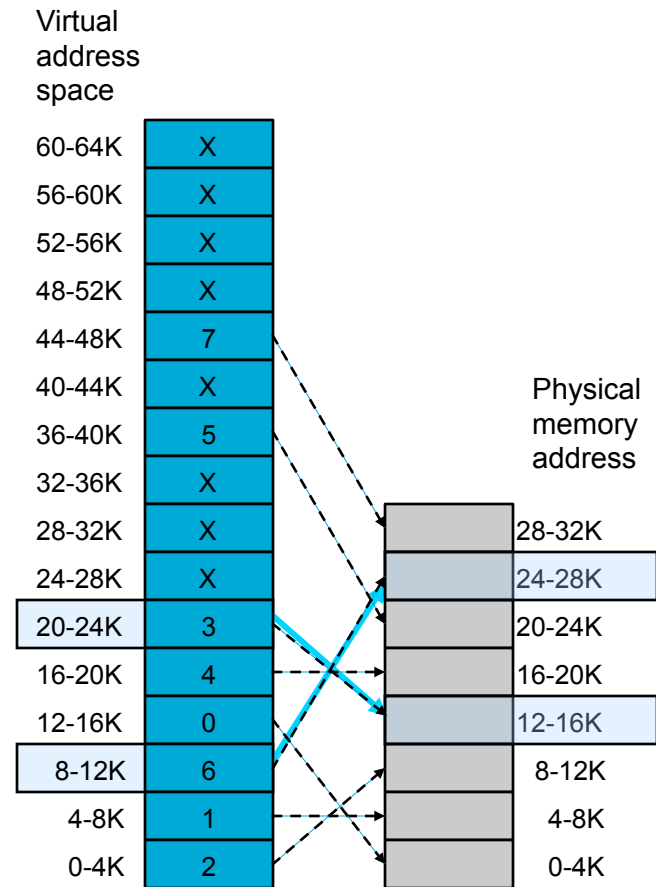
Physical address 24576

- **MOV REG, 20500**

Virtual address 20500 ($20480 + 20$)

Page frame 3

Physical address $20 + 12288$



Since virtual memory >> physical memory

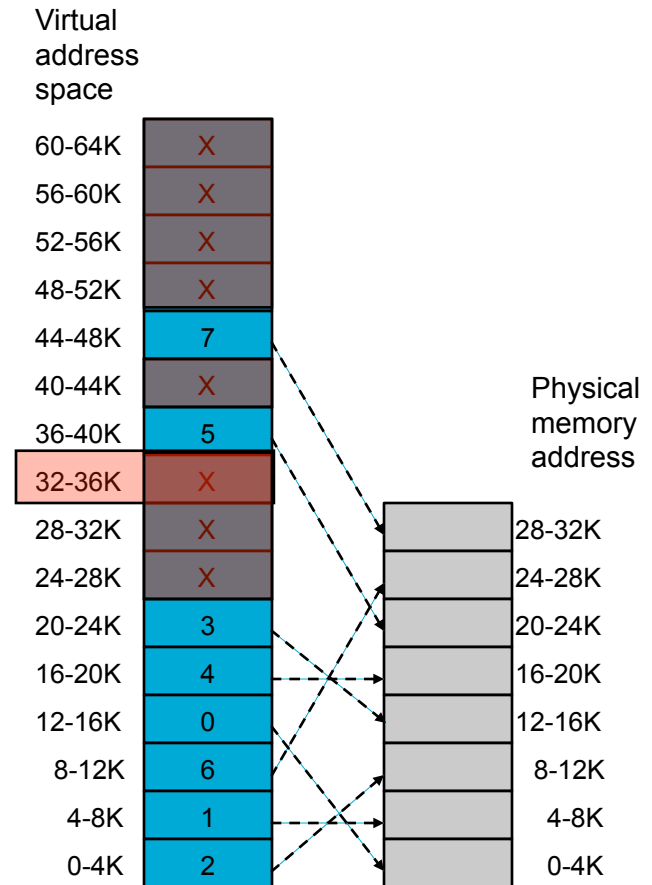
- Use a present/absent bit
- MMU checks –
 - If not there, “page fault” to the OS (trap)
 - OS picks a victim (?)
 - ... sends victim to disk
 - ... brings new one
 - ... updates page table

MOVE REG, 32780

Virtual address 32780

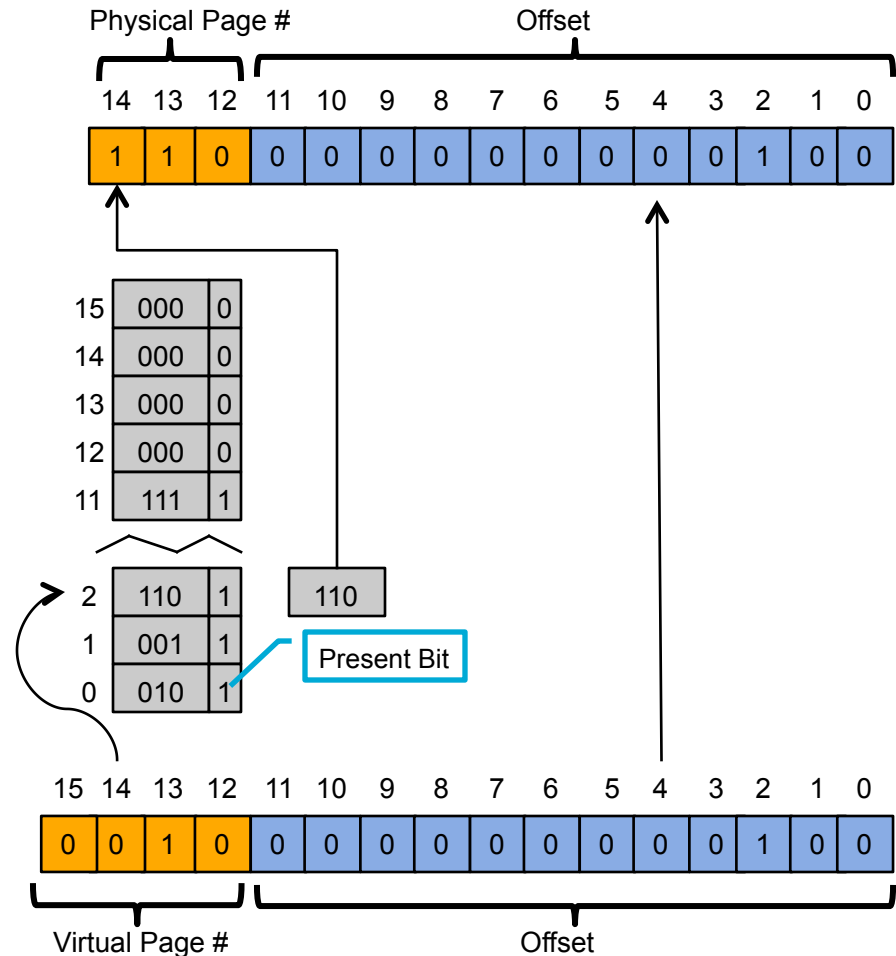
Virtual page 8, byte 12 (32768+12)

Page is unmapped – page fault!



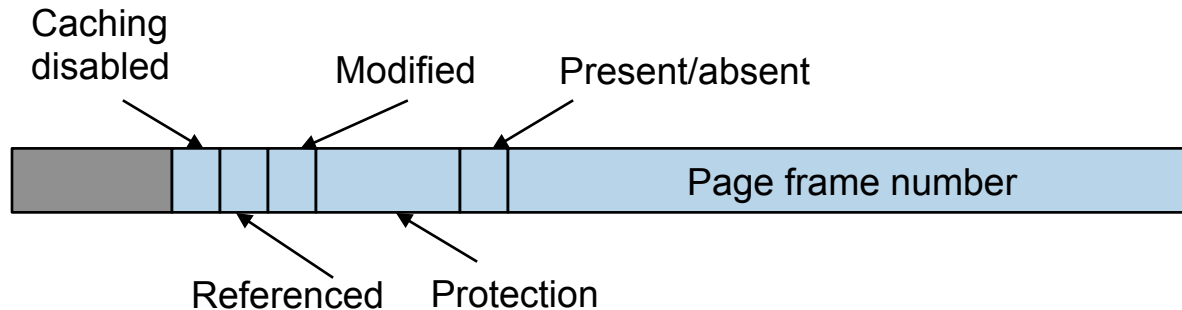
Translation in action

- 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



Page table entry

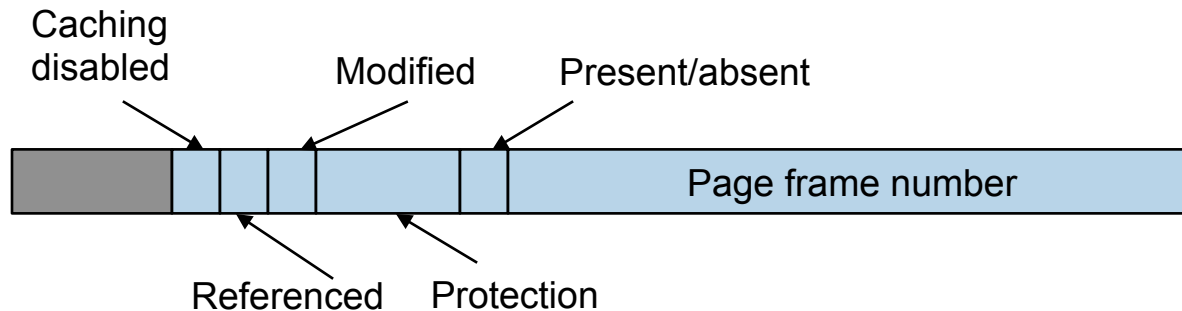
- An opportunity – there's a PTE lookup per memory ref., what else can we do with it?
- Looking at the details



- 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



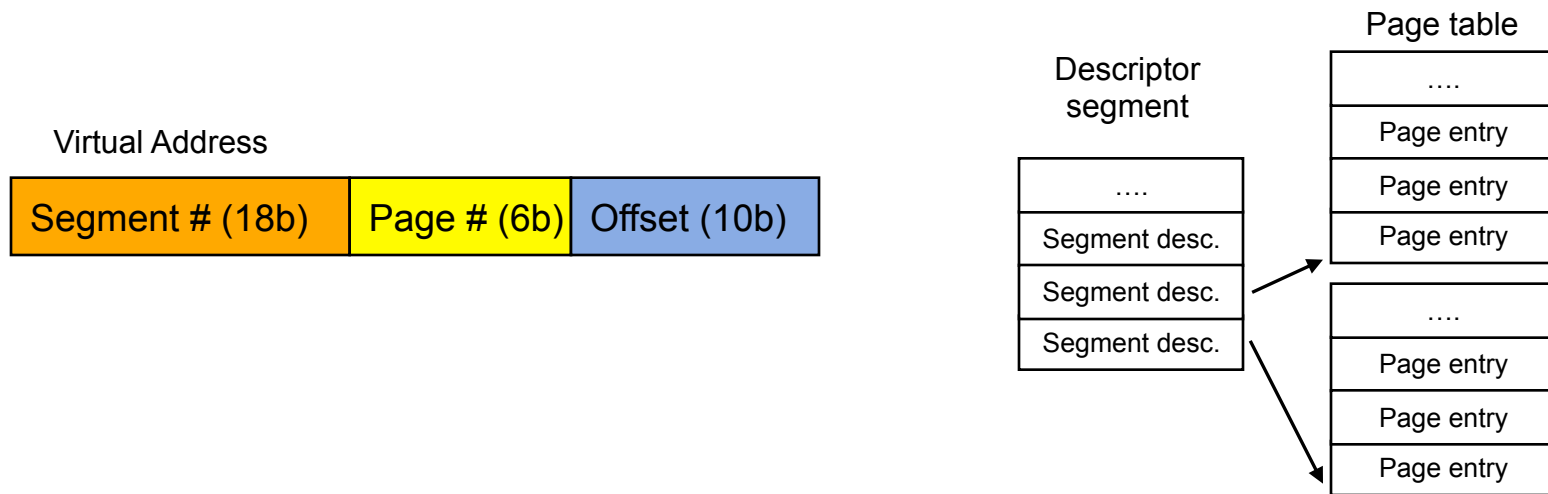
- Present/absent bit
 - Says whether or not the virtual address is used
- Modified (M): dirty bit
 - Set when a write to the page has occurred
- Referenced (R): Has it being used?
- To ensure we are not reading from cache (D)
 - Key for pages that map onto device registers rather than memory

Segmentation and paging

- Paging pros and cons
 - ✓ – Easy to allocate physical memory
 - ✓ – Naturally leads to virtual memory
 - ✗ – Address translation time
 - ✗ – Page tables can be large
- Segmentation pros and cons
 - ✓ – It's more logical
 - ✓ – Facilitates sharing and reuse
 - ✗ – But all the problems of variable partitions

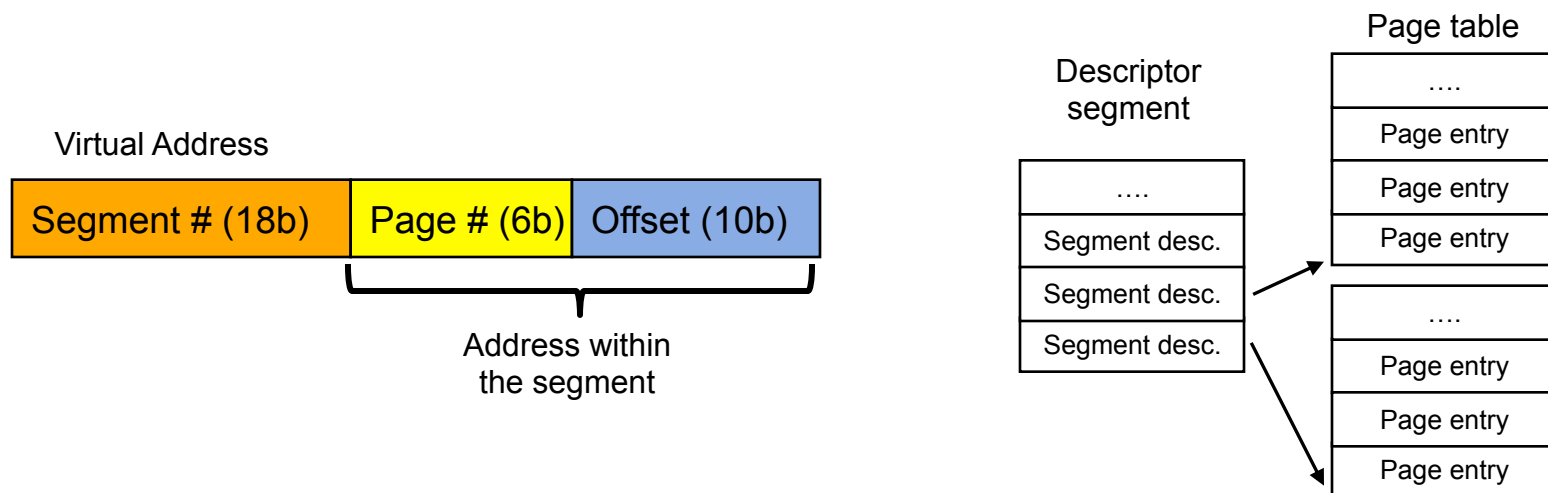
Segmentation w/ paging - MULTICS

- Large segment? Page them e.g MULTICS, x86 and x86-64
- Process: 2^{18} segments of ~64K words (36-bit)
 - Process has a segment table (itself a paged segment)
- Most (but not all) segments are paged



Segmentation w/ paging - MULTICS

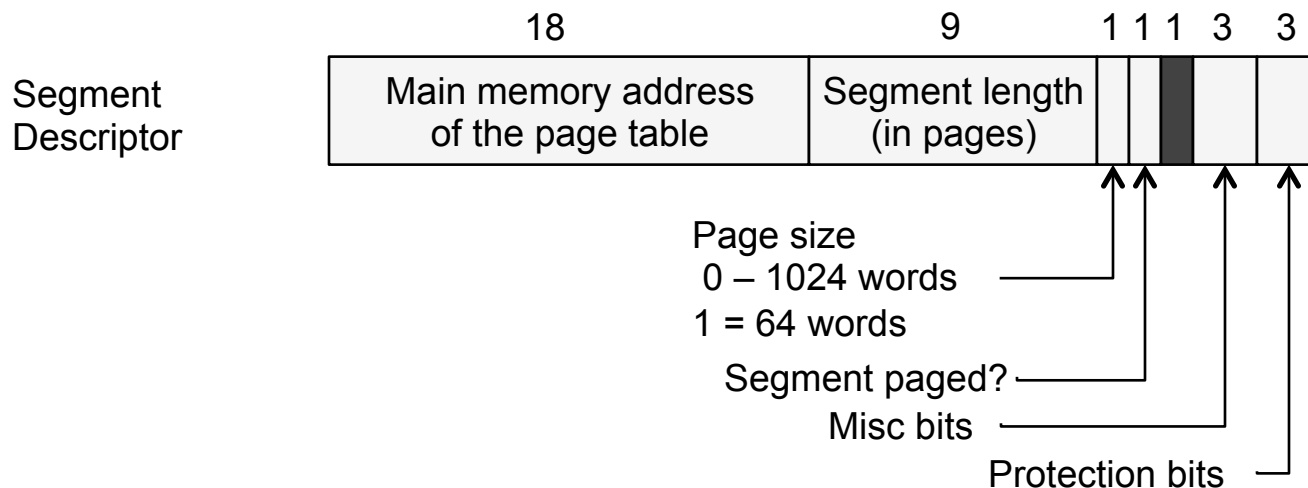
- To and from the segment table
 - Segment table address found in a Descriptor Base Register
 - One entry per segment
 - Segment descriptor indicates if in memory and points to page table
 - Address of segment in secondary memory in another table



Segmentation w/ paging - MULTICS

With memory references

- Segment # to get segment descriptor
- If segment in mem, segment's page table is in memory
- Protection violation?
- Look at the page table's entry - is page in memory?
- Add offset to page origin to get word location
- ... to speed things up – cache (TLB)



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 \rightarrow 12 bit offset, 20 bit page # \sim 1million PTE
 - 64b & 4KB page $\rightarrow 2^{12}$ (offset) + 2^{52} pages $\sim 4.5 \times 10^{15}$!!!
- 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

Next time

- Bigger and faster