Design and Implementation Issues

Today

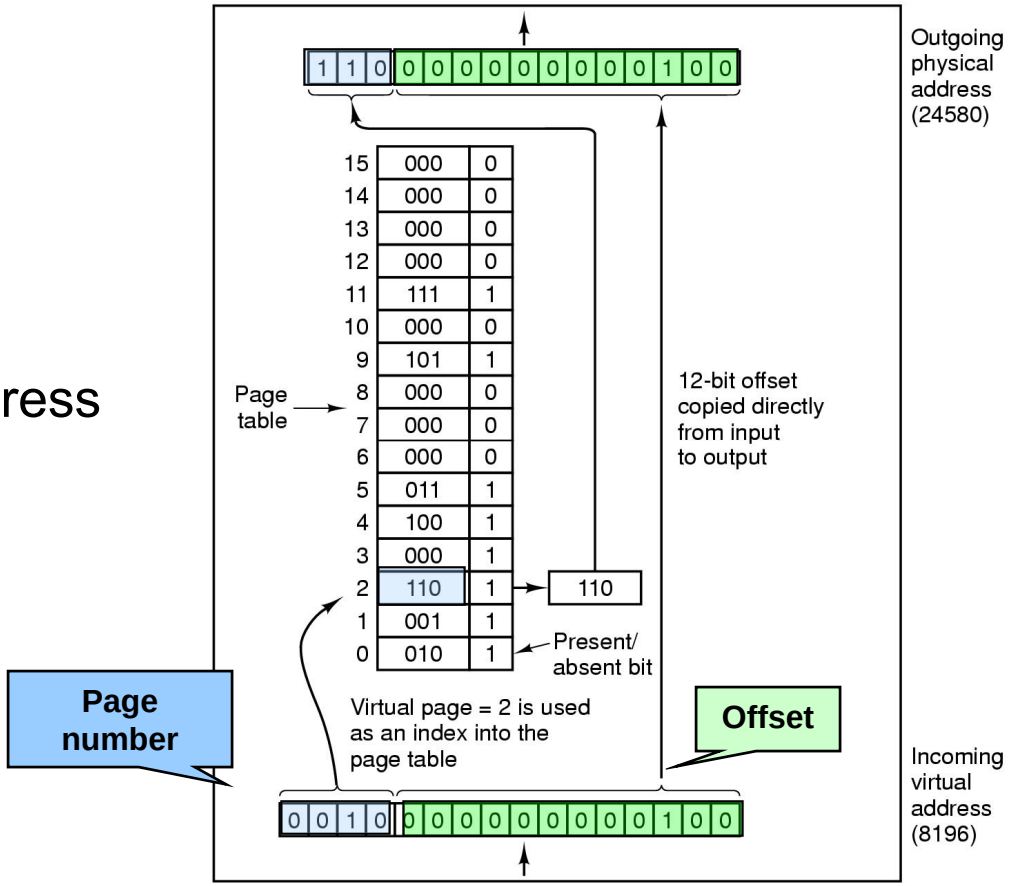
- Design issues for paging systems
- Implementation issues
- Segmentation

Next

- I/O

Details of the MMU work

- Again, 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



Considerations with page tables

Two key issues with page tables

- Mapping must be fast
  - It must be done on every memory reference, at least 1 per instruction
- With large address spaces, page tables are too big

  w/ 32 bit & 4KB page → 12 bit offset, 20 bit page # ~ 1million

  w/ 64 bit & 4KB page → 2^{12} (offset) + 2^{52} pages ~ 4.5×10^{15}!!!
- Simplest solutions
  - Page table in registers
    - Fast during execution, but potentially expensive & slow to context switch
  - Page table in memory and one register pointing to start
    - Fast to context switch and cheap, but slow during execution

Hierarchical page table

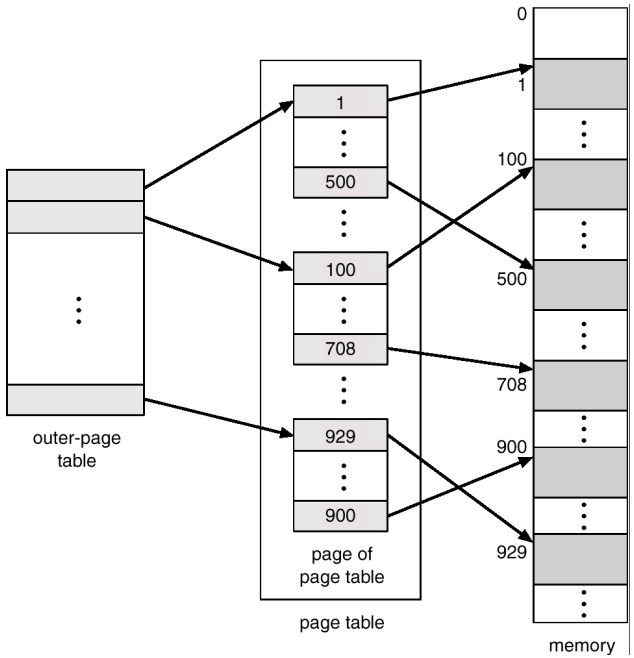
- Page the page table!
- Same argument – you don't need the full page table in memory
- Virtual address (32-bit machine, 4KB page):
  Page # (20 bits) + Offset (12 bits)
- Since page table is paged, page number is divided:
  Page number (10 bits) + Page offset in 2nd level (10 bits)

p1| p2 | offset

p1 - index into the outer page table
p2 - displacement within outer page

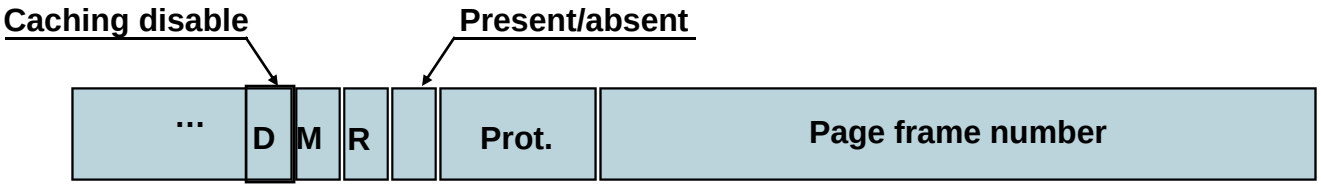
Example
Virtual address: 0x00403004

0000 0000 01	00 0000 0011	0000 0000 0100
P1 = 1	P2 = 3	Offset = 4



Page table entry

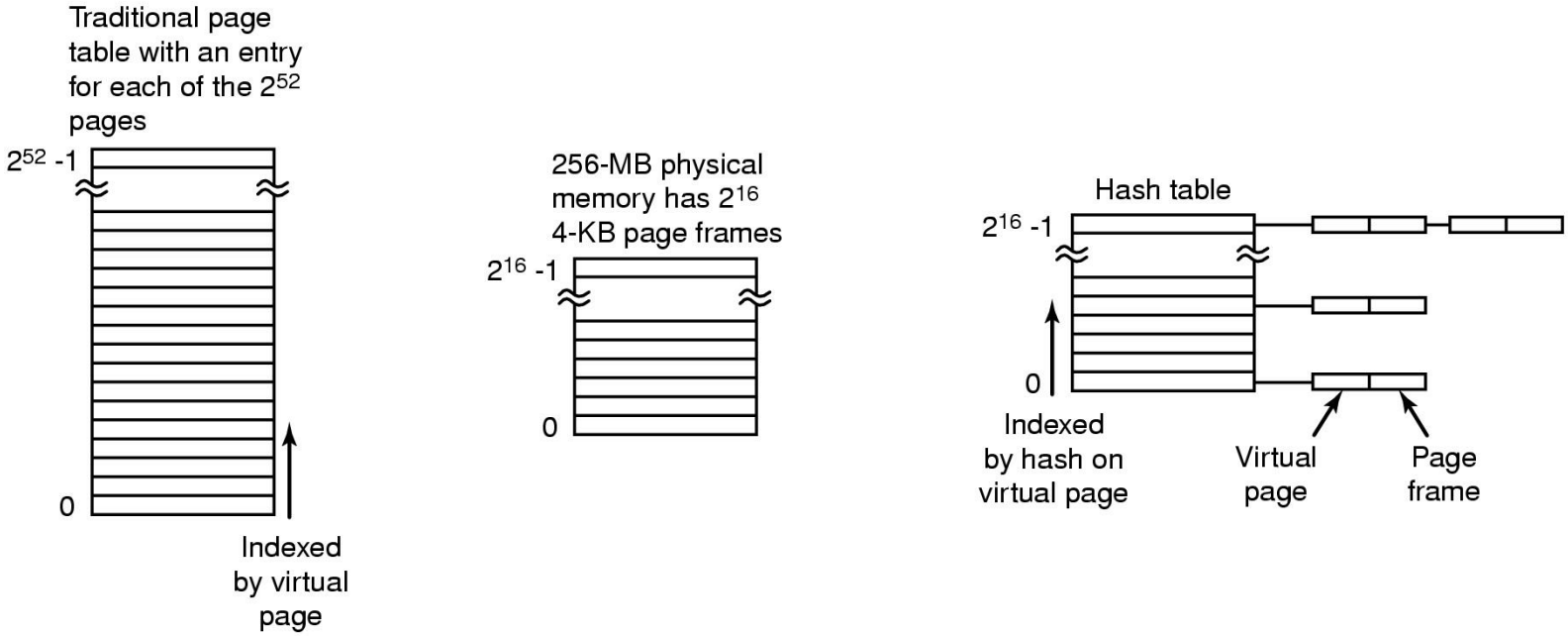
- Looking at the details



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

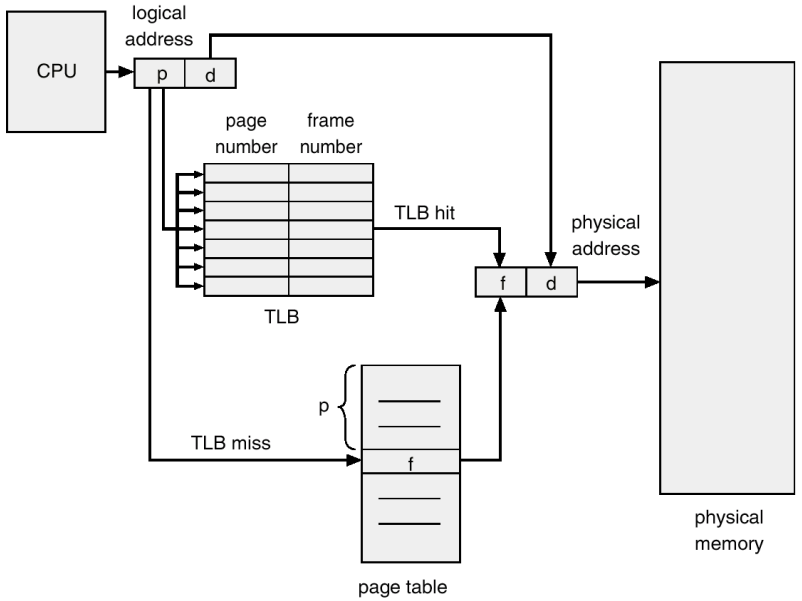
Inverted and hashed page tables

- Another way to save space – inverted page tables
  - Page tables are index by virtual page #, thus their size
  - Inverted page tables – one entry per page frame
    - Problem – too slow mapping!
  - Hash tables may help
  - Also, Translation Lookaside Buffer (TLB) …



Translation lookaside buffer (TLB)

- Small # entries (~64) – cache for page table entry
- Associative memory – parallel search
  - If hit – page frame is taken from there
  - Else – page table lookup + TLB replacement
- In many new architectures, TLB mgmnt. is done in software (simpler MMUs)
  - Entries loaded by OS
  - OS responsible for handle miss too



Paging Hardware with TLB

Effective access time

- Associative Lookup = ε time units
- Hit ratio - α - percentage of times that a page number is found in the associative registers (ratio related to TLB size)

Effective Memory Access Time (EAT)

TLB hit

TLB miss

EAT = α * (ε + memory-access)

α = 80% ε = 20 nsec memory-access = 100 nsec

EAT = 0.8 * (20 + 100) + 0.2 * (20 + 2 * 100) = 140 nsec

Design issues – global vs. local policy

- When you need a page frame
  - Pick a victim among your own resident pages? Local
  - Or among all pages? Global
- Local algorithms
  - Basically every process gets a fixed % of memory
- Global algorithms
  - Dynamically allocate frames among processes
  - Better, especially if working set size changes during execution
  - How many page frames per process?
    - Start with a basic set and react to Page Fault Frequency (PFF)
- Except for working set based algorithms, all the page replacement algorithms we've seen work both ways

  Why not working set based algorithms?

Load control

- Despite good designs, system may still thrash
  - Sum of working sets > physical memory
- Page Fault Frequency (PFF) indicates that
  - Some processes need more memory
  - but no process needs less

Page size

- OS can pick a page size *(how?)* - small or large?

  Small

  – Less internal fragmentation

  – Better fit for various data structures, code sections

  – Less unused program in memory,

  but …

  – More I/O time, getting page from disk … most of the time goes into seek and rotational delay!

  – Larger page tables

Average process size *s*
Page size *p*
Page entry size *e*

overhead = se / p + p/2

Page table space

Internal fragmentation

Taking first derivative respect to p and equating it to zero

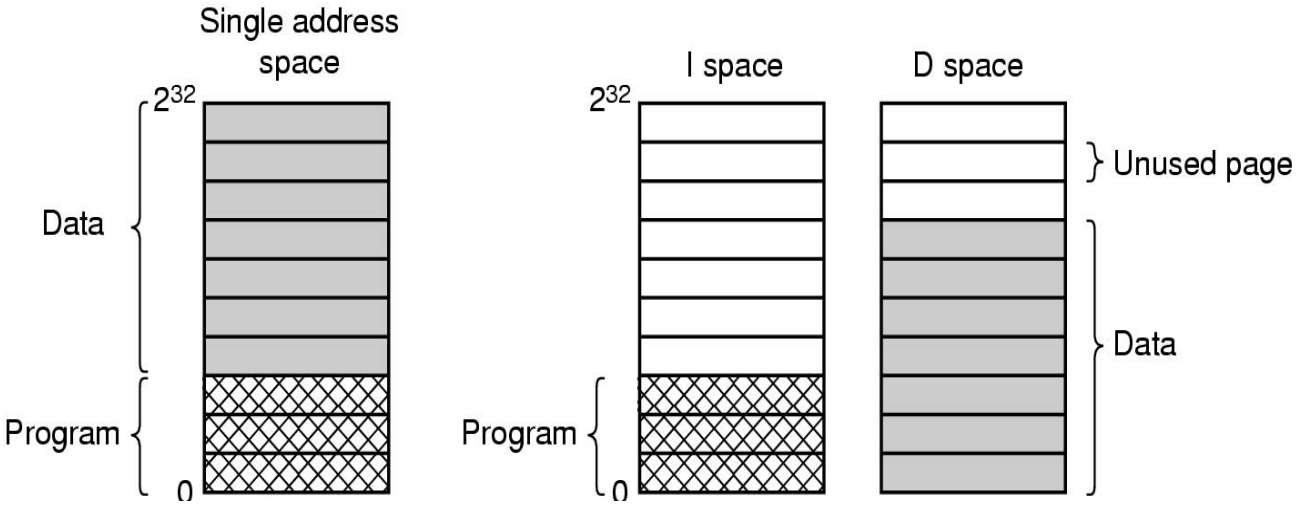
$-se / p^2 + 1/2 = 0$

$p = \sqrt{2se}$

s = 1MB
e = 8 bytes
Optimal p = 4KB

Separate instruction & data spaces

- One address space – size limit
- Pioneered by PDP-11: 2 address spaces, Instruction and Data spaces
  - Double the space
  - Each with its own page table & paging algorithm



Shared pages

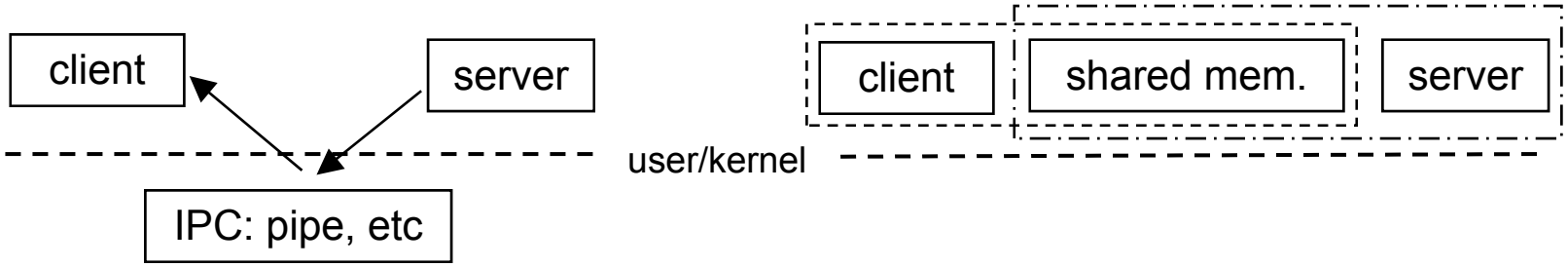
- In large multiprogramming systems – multiple users running same program - share pages?
- Some details
  - Not all is shareable
  - With I-space and D-space, sharing would be easier
  - What do you do if you swap one of the sharing process out?
- Sharing data is slightly trickier than sharing code
  - Fork in Unix
  - Sharing both data and program bet/ parent and child; each with its own page table but pages marked as READ ONLY
  - Copy On Write

Cleaning policy

- To avoid having to write pages out when needed – paging daemon
  - Periodically inspects state of memory
  - Keep enough pages free
  - If we need the page before it's overwritten – reclaim it!
- Two hands for better performance (BSD)
  - First one clears R, second checks it
  - If hands are kept close, only heavily used pages have a chance
  - If back is just ahead of front hand (359 degrees), original clock
  - When? If there are less than *x* free pages

Virtual memory interface

- So far, transparent virtual memory
- Some control for expert use
  - For shared memory – fast IPC



  - For distributed shared memory

    Going to disk may be slower than going to somebody else's memory!

Implementation issues

Operating System involvement w/ paging:

- Process creation
  - Determine program size, allocate space for page table, for swap, bring stuff into swap, record info into PCB
- Process execution
  - Reset MMU for new process, flush TLB, make new page table current, pre-page?
- Page fault time
  - Find out which virtual address cause the fault, find page in disk, get page frame, load page, reset PC, …
- Process termination time
  - Release page table, pages, swap space, careful with shared pages

Page fault handling

- Hardware traps to kernel
- General registers saved by assembler routine, OS called
- OS find which virtual page cause the fault
- OS checks address is valid, seeks page frame
- If selected frame is dirty, write it to disk *(CS)*
- Get new page *(CS)*, update page table
- Back up instruction where interrupted
- Schedule faulting process
- Routine load registers & other state and return to user space

Instruction backup

- As we've seen, when a program causes a page fault, the current instruction is stopped part way through …
- Harder than you think!
  - Consider instruction: MOV.L #6(A1), 2(A0)
  - Which one caused the page fault?
  - It can even get worse – auto-decrement and auto-increment?
- Some CPU designers have included hidden registers to store
  - Beginning of instruction
  - Indicate autodecr./autoincr. and amount

Locking pages in memory

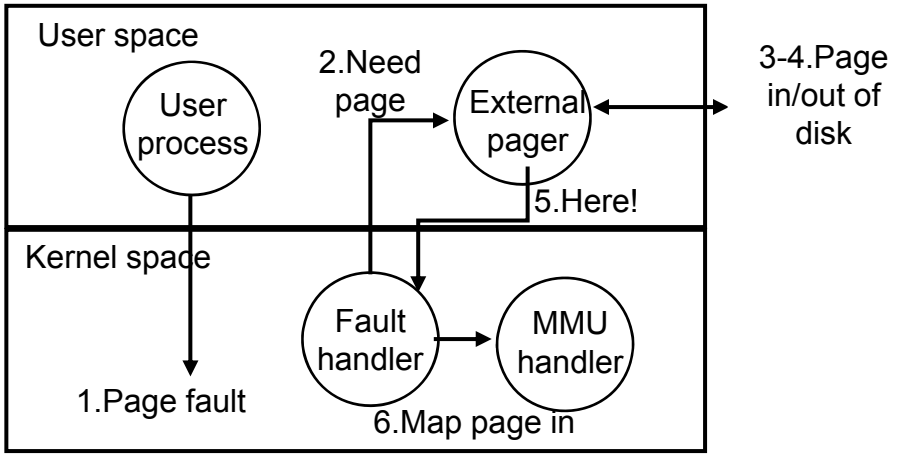
- Virtual memory and I/O occasionally interact
- Process issues call for read from device
  - While waiting for I/O, another processes starts up
  - Second process has a page fault
  - Buffer for the first process may be chosen to be paged out!
- Solutions:
  - Pinning down pages in memory
  - Do all I/O to kernel buffers and copy later

Backing store

- How do we manage swap area?
  - Allocate space to process when started
  - Keep offset to process swap area in PCB
  - Process can be brought entirely when started or as needed
- Some problems
  - Size – process can grow … split text/data/stack segments in swap area
  - Do not allocate anything … you may need extra memory to keep track of pages in swap!

Separation of policy & mechanism

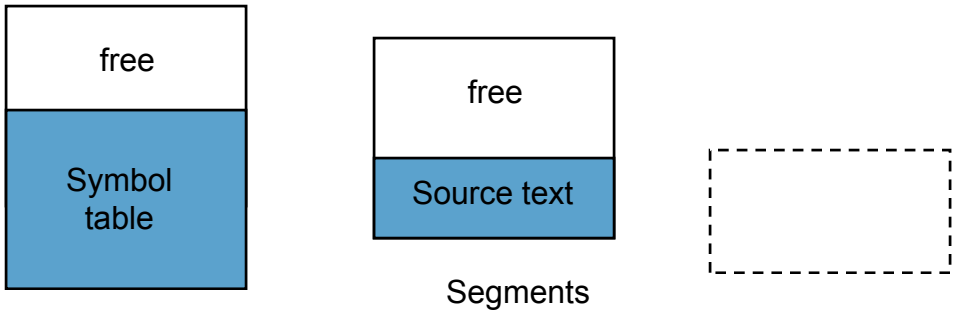
- How to structure the memory management system for easy separation? Mach:
  1. Low-level MMU handler – machine dependent
  2. Page-fault handler in kernel – machine independent, most of paging mechanism
  3. External pager in user space – user-level process
- Where do you put the page replacement algorithm?
- Pros and cons

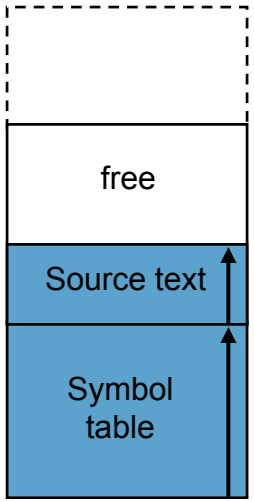


Segmentation

- So far - one-dimensional address spaces
- For many problems, having multiple AS is better

  e.g. compiler with various tables that grow dynamically
- Multiple AS → segments
  - A logical entity – programmer knows
  - Different segments of different sizes
  - Each one growing independently
  - Address now includes segment # + offset
  - Protection per segment can be different



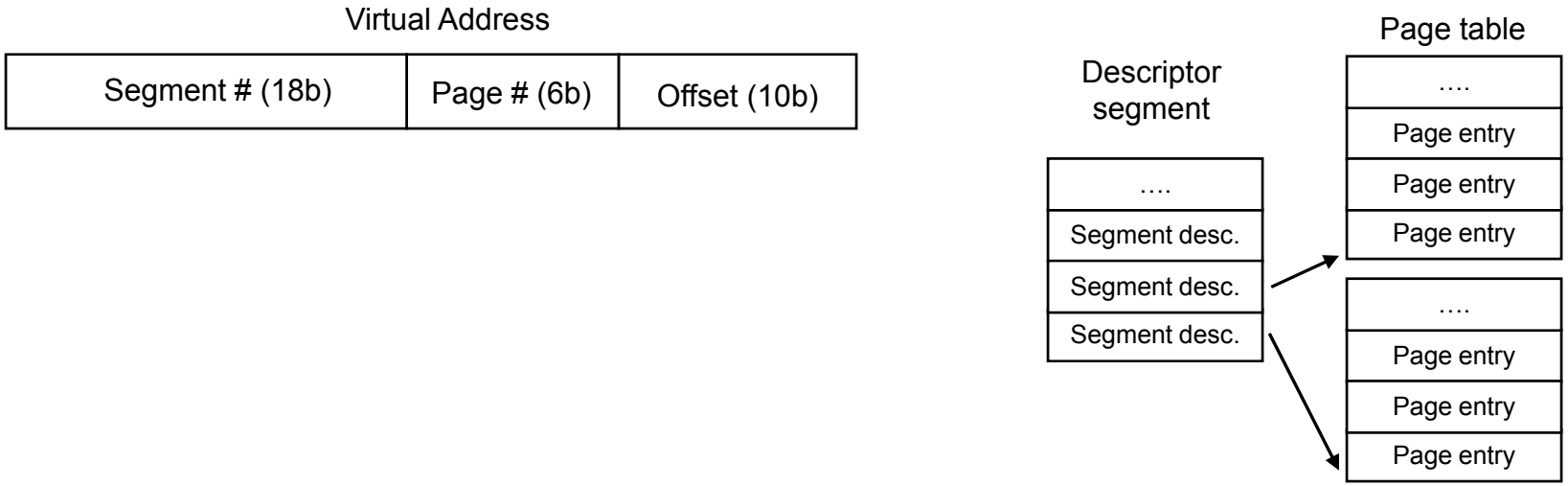


Paging vs. segmentation

Consideration	Paging	Segmentation
Need the programmer be aware?	No	Yes
# Linear address spaces	1	Many
Can procedure & data be distinguished & separately protected?	No	Yes
Is sharing procedures bet/ processes facilitated?	No	Yes
Why was the technique invented?	Get a large virtual space w/o more physical memory?	Allow programs & data to be broken into logically independent address spaces Aid sharing & protection

Segmentation w/ paging - MULTICS

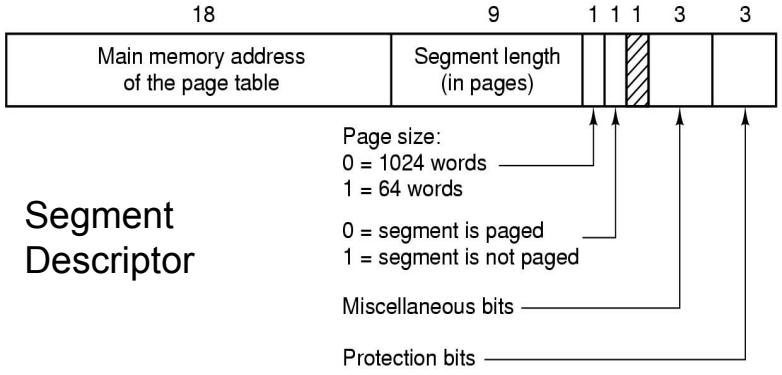
- Large segment? Page them e.g **MULTICS** & Pentium
- Process: 2^{18} segments of ~64K words (36-bit)
- Most segments are paged
- Process has a segment table (itself a paged segment)
- Segment descriptor indicates if in memory
- Segment descriptor points to page table
- Address of segment in secondary memory in another table



Segmentation w/ paging - MULTICS

With memory references

- Segment # to get segment descriptor
- If segment in memory, 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 - TLB



Next time

- Principles of I/O hardware and software
- Disks and disk arrays
- … file systems