Address Types

- Three kinds of addresses
  - Logical addresses
    - X = relative user addresses
    - Process P accesses memory address X
  - Linear addresses
    - L = base address of P + X
    - Mapped via segmentation
  - Physical addresses
    - P = f(L) where f is a 1-1 function
    - P in kernel-land is where data is!
    - Mapped via paging

Addresses in GeekOS

- Currently in GeekOS
  - Logical address → Linear address = Physical address
  - Downsides:
    - Need space allocated for each process
    - Limited by physical memory of the system (GeekOS has 8MB of physical memory)
    - Less flexible

IA-32 Memory Management

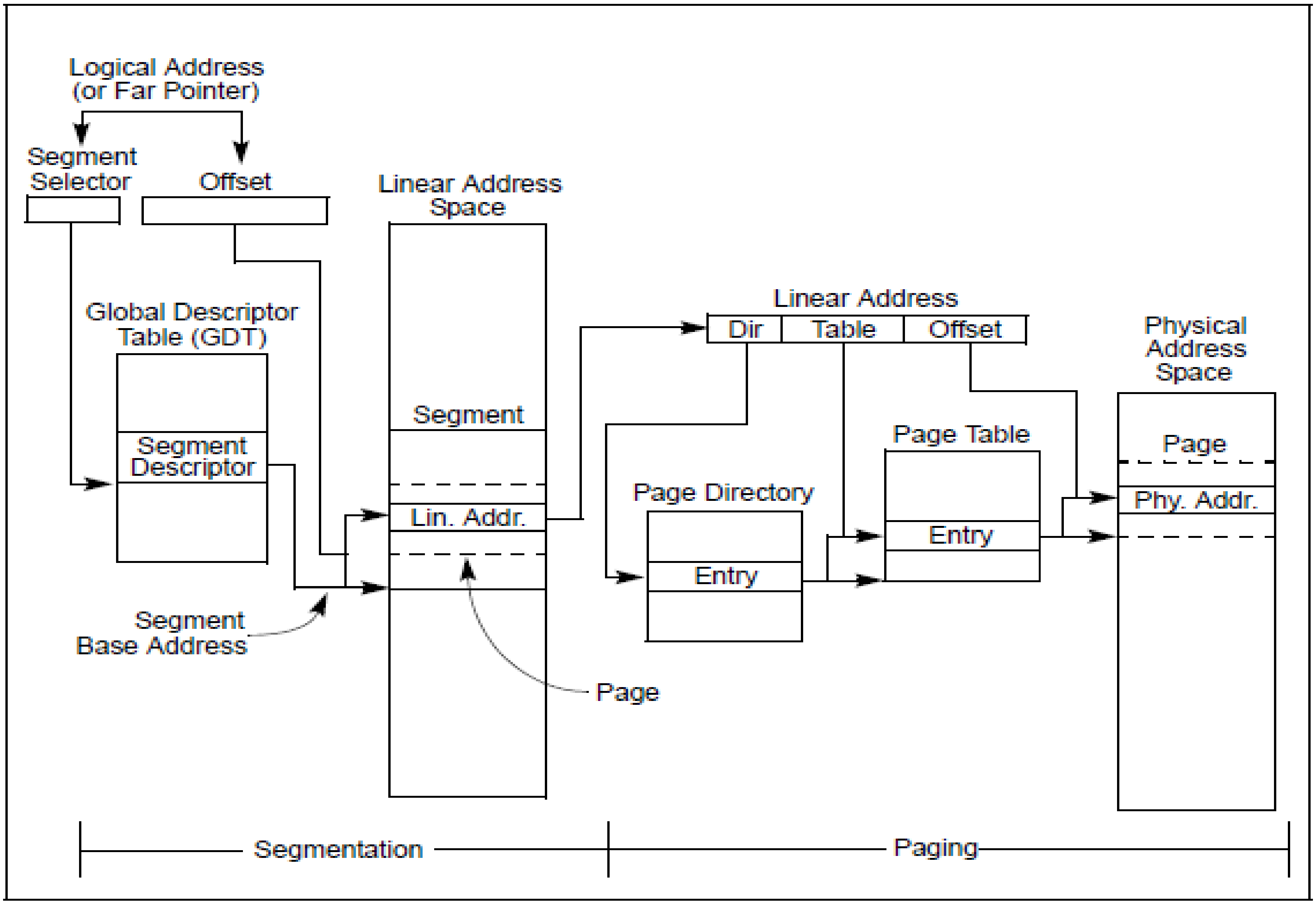
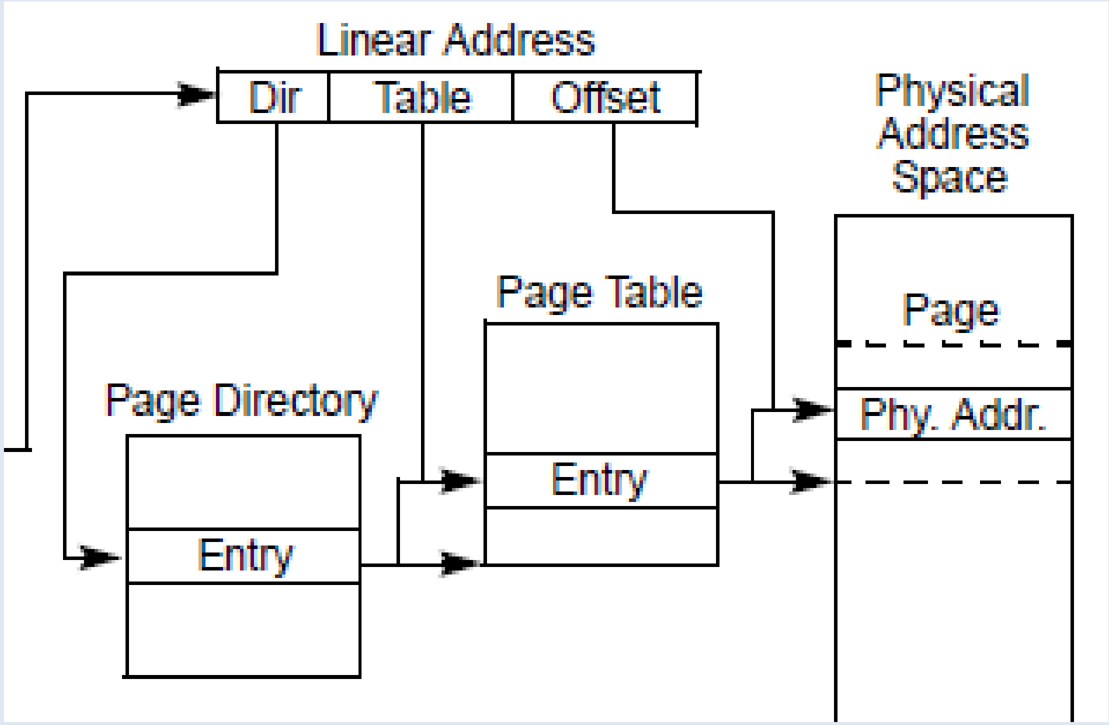


Figure 3-1. Segmentation and Paging

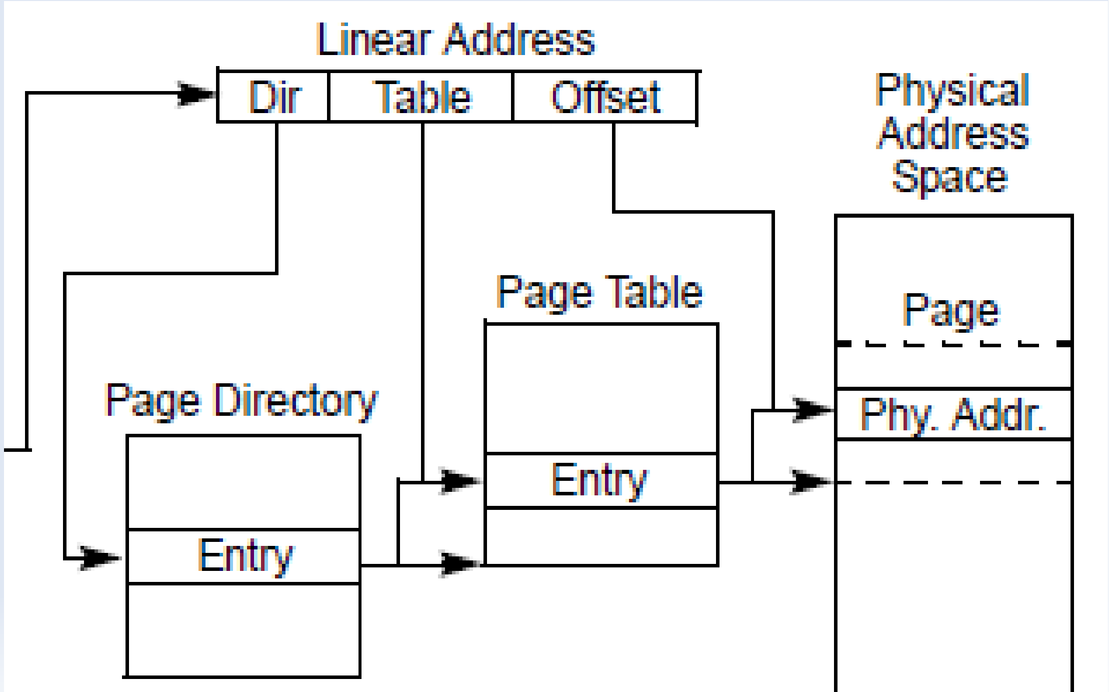
Paging Schemes

- Two-level paging scheme – directory and tables
  - Why use this instead of a giant page table?



GeekOS Paging

- Given a linear address, how to get page?
- Take linear address (32 bits)
- First 10 bits to get directory entry → page table
  - 10 bits = 1024 entries per directory
- Next 10 bits to get table entry → page
  - 10 bits = 1024 entries per table
- Last 12 bits to get byte in page
  - 12 bits = 4096 bytes per page
- Therefore, memory is split up into ”chunks” of size 4KB called pages



Mapping Kernel Memory (Part I)

- For the kernel, linear addresses = physical addresses
- Therefore, for all linear pages, map linear address X to physical address X
  - Example!
- GeekOS should still work exactly the same, except you've added a transparent paging system
- Deadline April 8

GeekOS Memory – Linear addresses

- 0x0000 0000 (0GB) - Kernel Memory starts
- 0x8000 0000 (2GB)- User Memory data/text start (base address)
- 0xFFFF E000 - User Memory - initial stack at top of this page
- 0xFFFF F000 - User Memory - args in this page
- 0xFFFF FFFF (4GB) - Memory space ends here

User Memory Mapping

- Implement in uservm.c, but copy-paste massively from userseg.c
- Copy kernel page directory (bottom 2GB) so that kernel can access memory when handling interrupts
- Allocate pages for text/data/stack for the upper 2GB, but only pages the program needs!

Demand Paging

- Errors with paging trigger interrupt 14
  - Register a page fault handler to handle this
  - Default one provided kills user program
  - Only user programs can fault
    - Kernel accessing wrong address = boom
- If a user program accesses right above its current stack limit, grow the stack!
  - Page doesn't exist, so will trigger handler

Paging to Disk

- Before: only had 8MB physical memory
- What if a program wanted to use 20MB?
- Solution: write ”unused” memory to disk!
- First, we allocate swappable pages using Alloc_Pageable_Page instead of Alloc_Page
- If our memory is full, call Find_Page_To_Page_Out
  - ”pseudo” LRU: Maintain ptr to a page, check accessed bit
  - if zero, reclaim page, otherwise, set to zero

Paging to Disk

- Take page, write it to disk (Find_Space_On_Paging_File, Write_To_Paging_File)
- Overwrite that page with whatever new data
- Later on... you try to access that page again, and it's not there! It will trigger a page fault, so you need to modify the interrupt handler to swap it back in.
  - Get index of block on disk from page table entry pageBaseAddr
  - Read_From_Paging_File