

# Project 4 Roadmap

Finally, the user stack will have some breathing room

# x86 Paging Overview

- <ftp://download.intel.com/design/Pentium4/manuals/25366820.pdf>
- figures on pages 3-2, 3-21

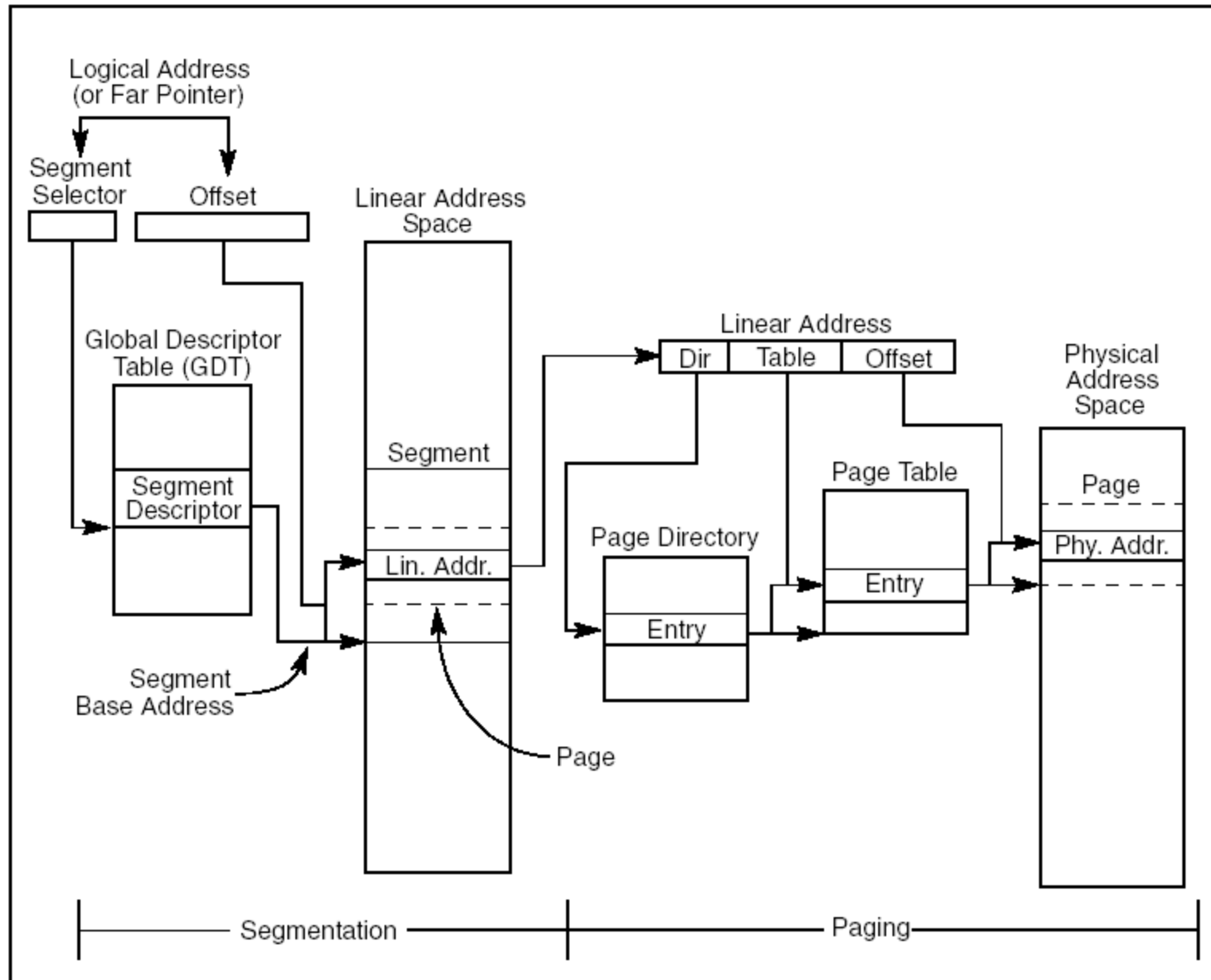
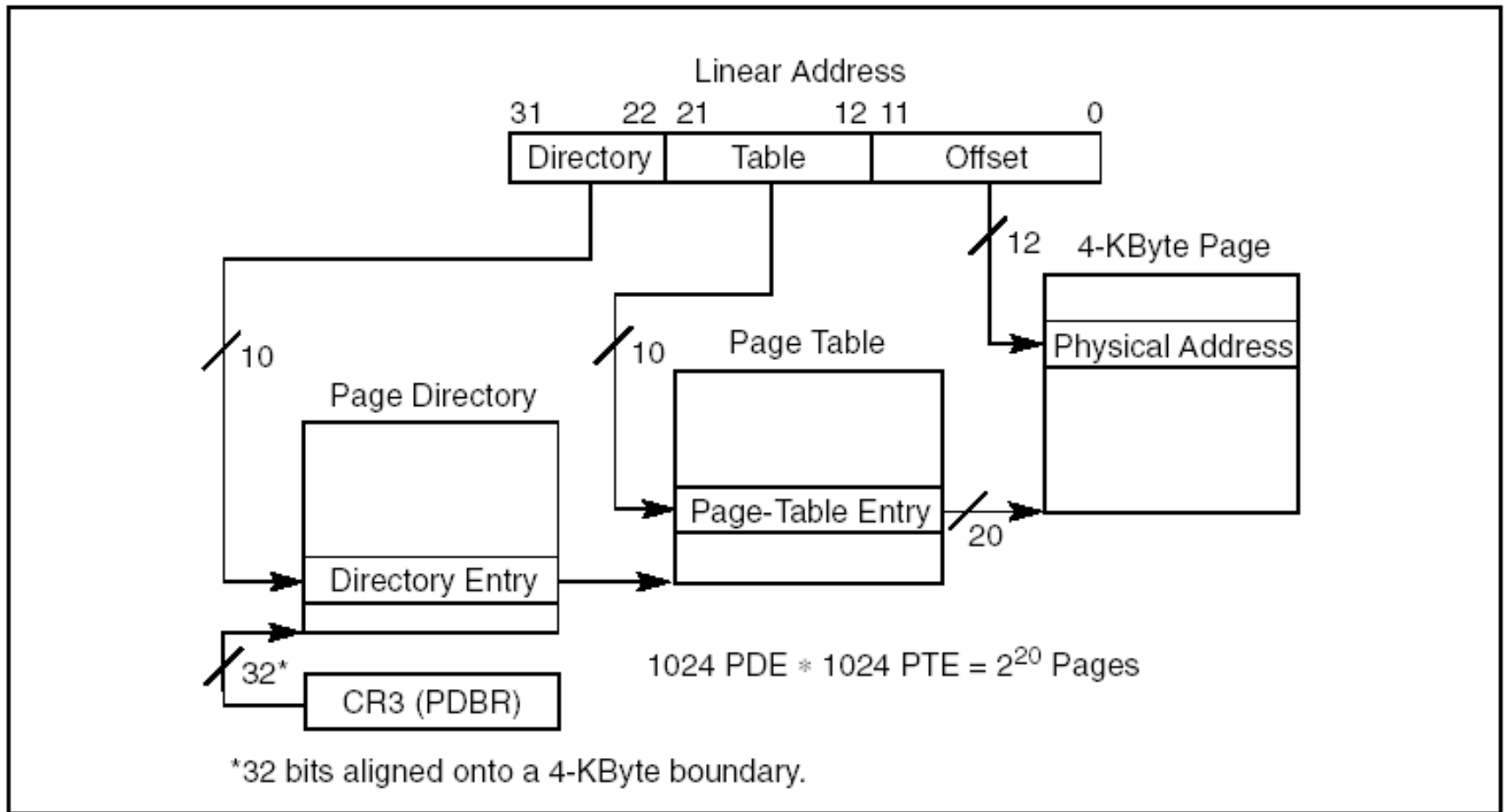


Figure 3-1. Segmentation and Paging



**Figure 3-12. Linear Address Translation (4-KByte Pages)**

# Mapping kernel memory (theory)

- Premise: for the kernel, linear to physical mapping is *one-to-one*
- GeekOS has 8MB of physical memory
  - how many page directories will be needed?
  - how many page tables will be needed?
- Kernel is mapped from 0-2GB, user from 2GB-4GB
  - What does the paging infrastructure look like?

# Mapping kernel memory (practice)

- **Crucial!** cannot get credit for any part of the project if this doesn't work
- Basic Idea: for the kernel, linear to physical mapping is one-to-one
- effectively
  - for all linear pages: map linear pages  $i$  to physical page  $i$ )
- early deadline for this, see webpage

# Mapping kernel memory: steps

*Remember: for the kernel, linear to physical mapping is one-to-one*

- determine the amount of physical memory (`bootInfo->memSizeKB`)
- allocate page directory
- write functions for allocating page directory entries/page table entries
  - handy `PAGE_DIRECTORY_INDEX/PAGE_TABLE_INDEX` are defined for you
- **for (i=0; i< allPhysical Pages;i++) do**
  - register page (i.e. linear page `i` maps to physical page `i`)
  - use `Get_Page(addr)` from `mem.h` to get the `struct Page` associated with a physical page
  - flags to (`VM_WRITE | VM_READ | VM_USER`) for `pde_t/pte_t`
- turn on paging (`Enable_Paging`)
- register page fault handler (`Install_Interrupt_Handler`)
- test here!

# User Memory Mapping

- `uservm.c`, but can copy-paste massively from `userseg.c`
- `Load_User_Program/Create_User_Context`
  - allocate page directory; save it in `userContext->pageDir`
  - copy kernel's page directory entries
  - allocate pages for data/text; copy from image
    - don't leave space for stack
  - allocate two more for stack/args
  - linear memory space is identical for all processes now
  - start address is `0x80000000`, size is `0x80000000`
  - make sure you get `userContext's`  
`memory/size/stackPointerAddr/argBlockAddr` right

# Demand Paging

- Page fault handler (paging.c)
  - register handler w/interrupt 14 in Init\_VM()
- Demand paging implementation
  - only a user program may fault
  - case 1 – “page in” request
  - case 2 – stack growth request
- Test: use rec.c to trigger a fault (memory pressure by stack expansion)

# Virtual Memory: Physical Page Allocation

- `Alloc_Pageable_Page()` vs `Alloc_Page()`  
(`mem.c`)
  - use `Alloc_Page()` for directories/page tables
  - use `Alloc_Pageable_Page()` for everything else
    - returned page is `PAGE_PAGEABLE`, hence possibly swap out

# VM: Page Replacement

- LRU in theory: see textbook 9.4.5
- Ours - “pseudo” LRU
  - add hook in `Page_Fault_Handler()`
  - walk thru all physical pages
  - if page subject to paging and `accessed==1` then increment clock, set `accessed=0`  
(see `struct Page`, `struct pte_t`)
  - HW sets the `accessed` to 1 automatically upon read/write in that page; but you have to set it to 0 manually when you update the clocks
  - `Find_Page_To_Page_Out` finds page with lowest clock

# VM: Swapping

- Page out
  - when ?
  - which page ? (`Find_Page_To_Page_Out`, see previous slide!)
  - where ? (`Find_Space_On_Paging_File()`)
  - how ? (you'll do `Write_To_Paging_File (void *paddr, ulong_t virtual, int pageFileIndex)`)
- Page in
  - when ?
  - how ? (you'll do `Read_From_Paging_File (void *paddr, ulong_t virtual, int pageFileIndex)`)
- Housekeeping
  - `pageTable->kernelInfo = KINFO_PAGE_ON_DISK/0`
  - `pageTable->pageBaseAddr = <block on disk>`
  - disk page management – have to do it yourself