x86 segmentation, page tables, and interrupts

3/17/08

Frans Kaashoek

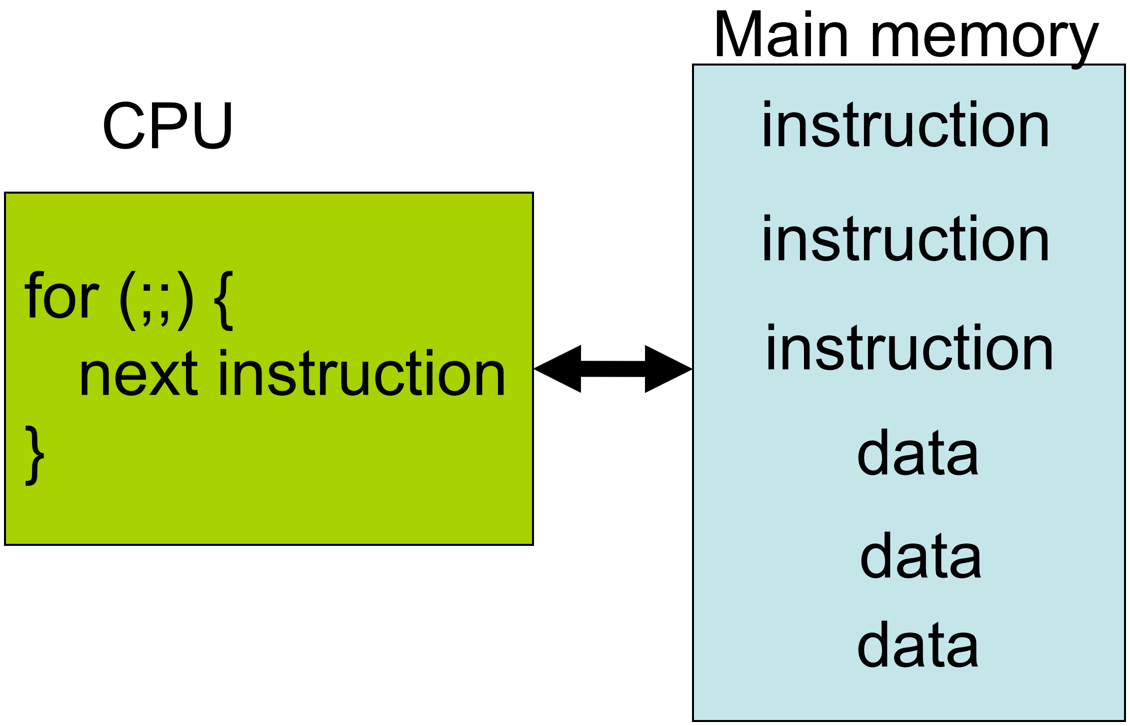
MIT

kaashoek@mit.edu

Outline

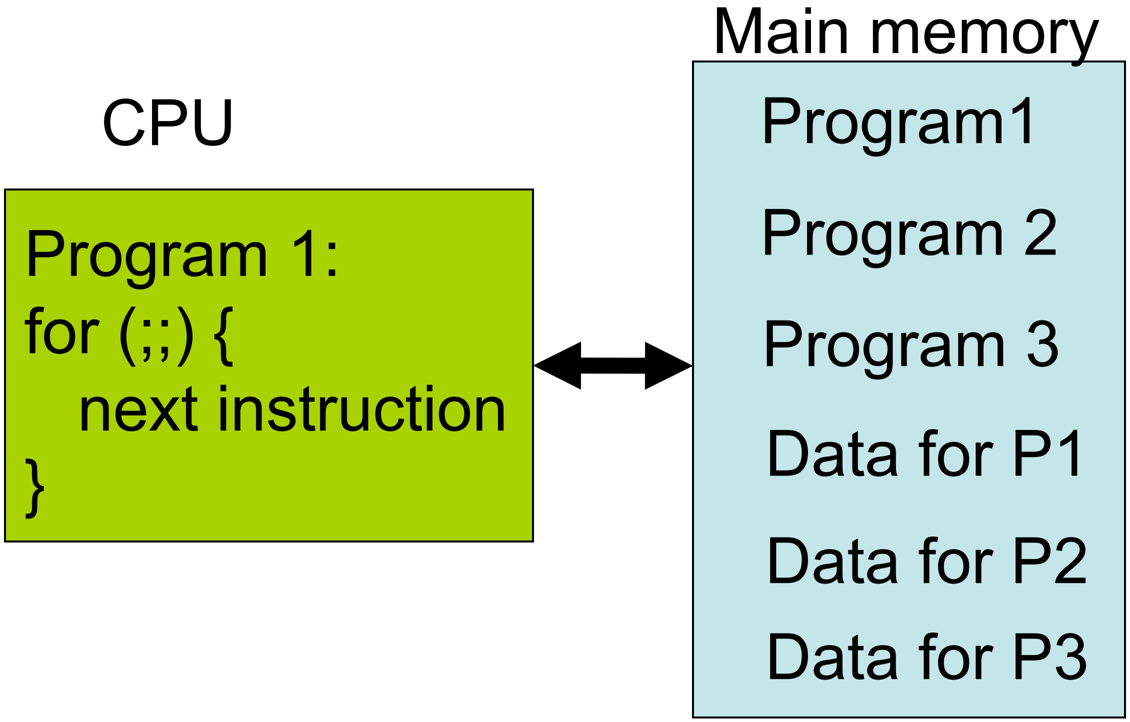
- Enforcing modularity with virtualization
  - Virtualize processor and memory
- x86 mechanism for virtualization
  - Segmentation
  - User and kernel mode
  - Page tables
  - System calls

Last lecture’s computer



- Memory holds *instructions* and *data*
- CPU *interprets* instructions

Better view



CPU

```
Program 1:
for (;;) {
   next instruction
}
```

Main memory

Program1

Program 2

Program 3

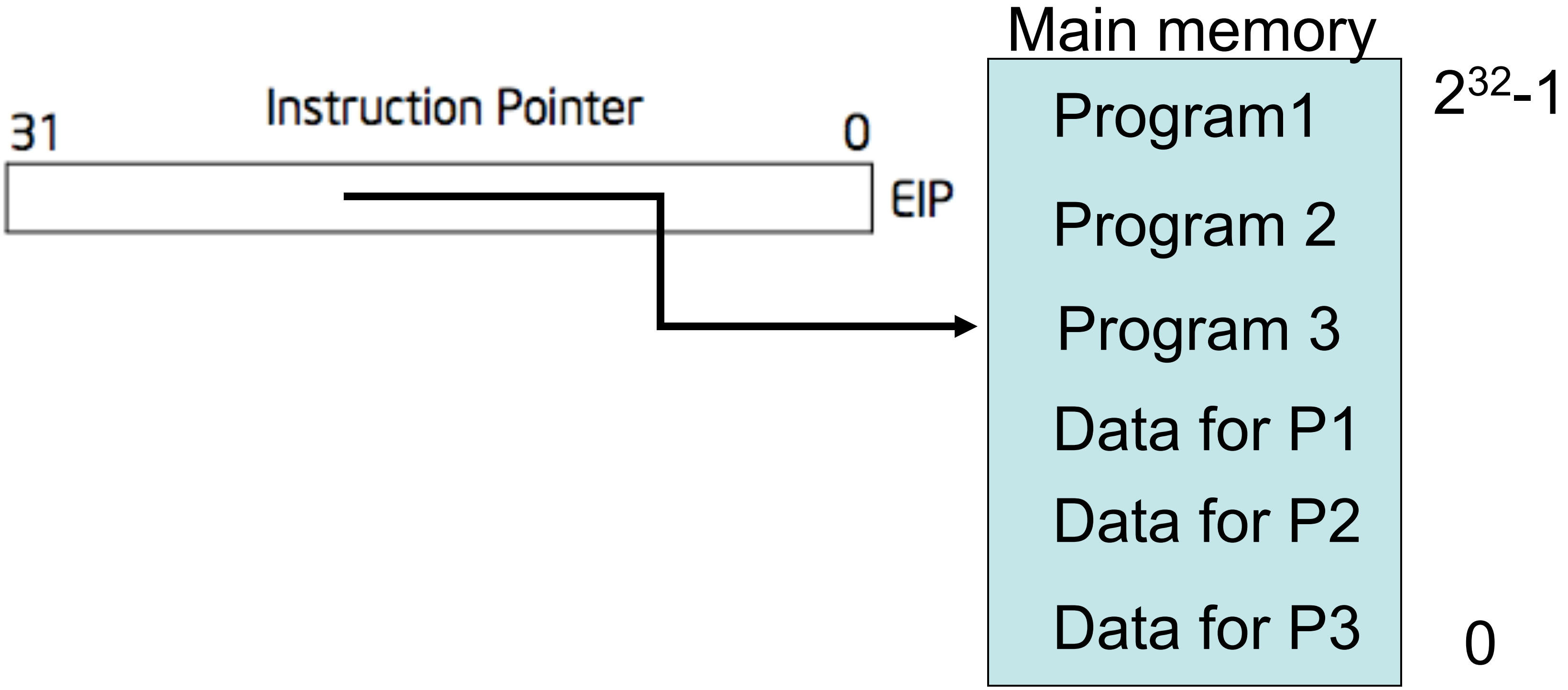
Data for P1

Data for P2

Data for P3

- For modularity reasons: many programs
- OS switches processor(s) between programs

Problem: no boundaries



Instruction Pointer

31

0

EIP

Main memory

$2^{32}-1$

Program1

Program 2

Program 3

Data for P1

Data for P2

Data for P3

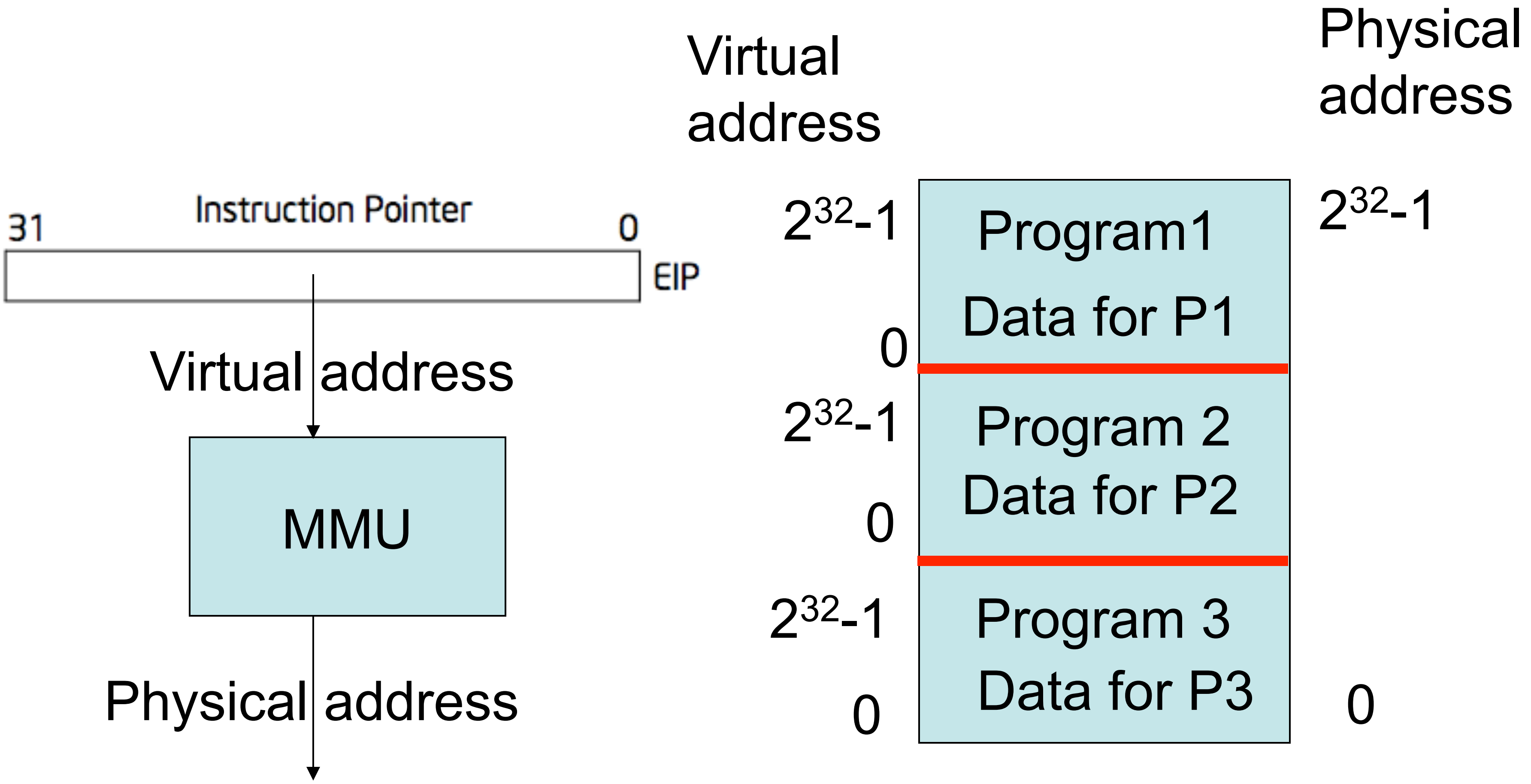
0

- A program can modify other programs data
- A program jumps into other program’s code
- A program may get into an infinite loop

Goal: enforcing modularity

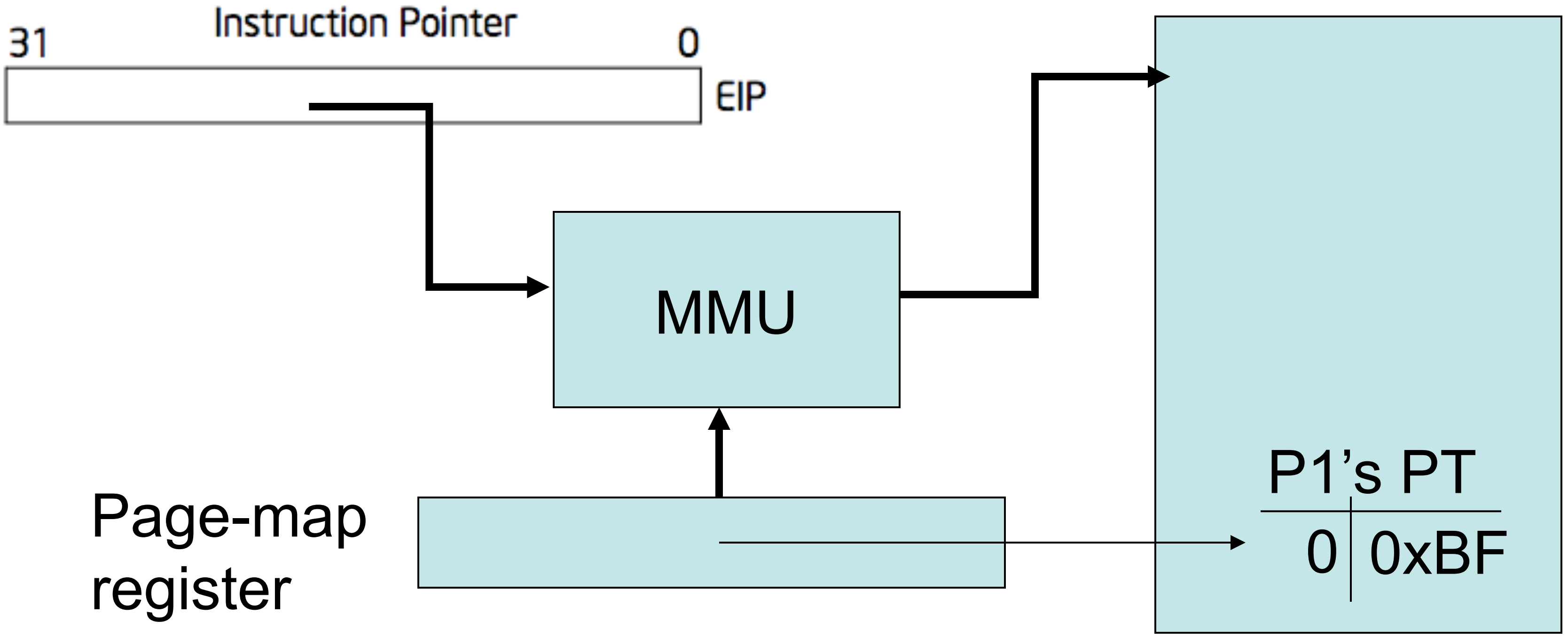
- Give each program its private memory for code, stack, and data
- Prevent one program from getting out of its memory
- Allowing sharing between programs when needed
- Force programs to share processor

Solution approach: virtualization



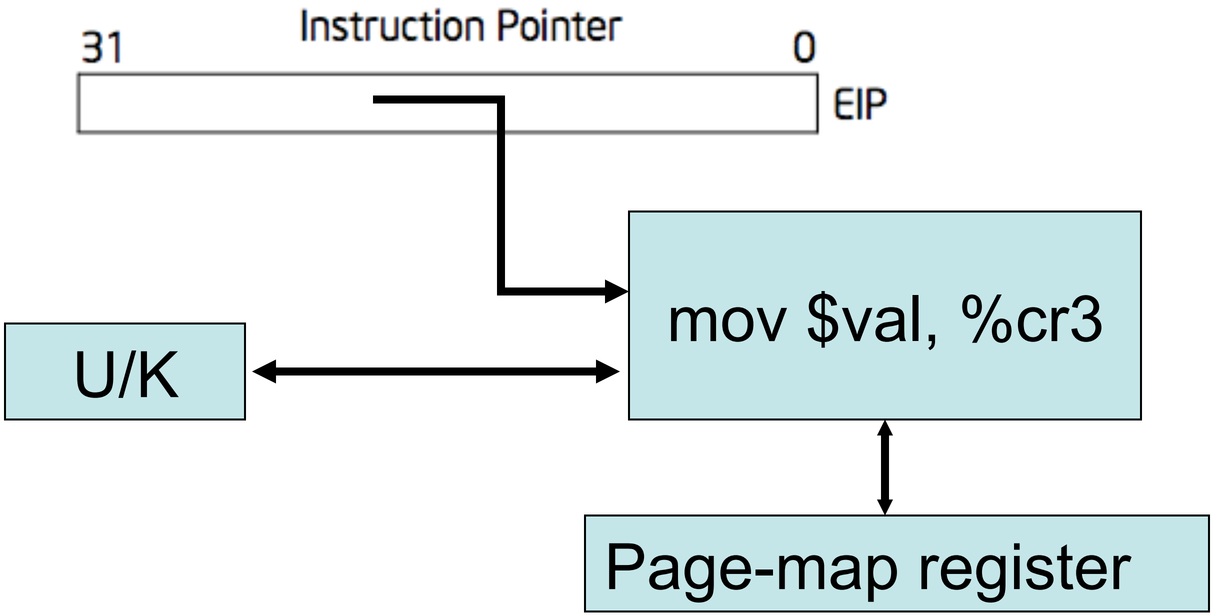
- Virtualize memory: virtual addresses
- Virtualize processor: preemptive scheduling

Page map guides translation



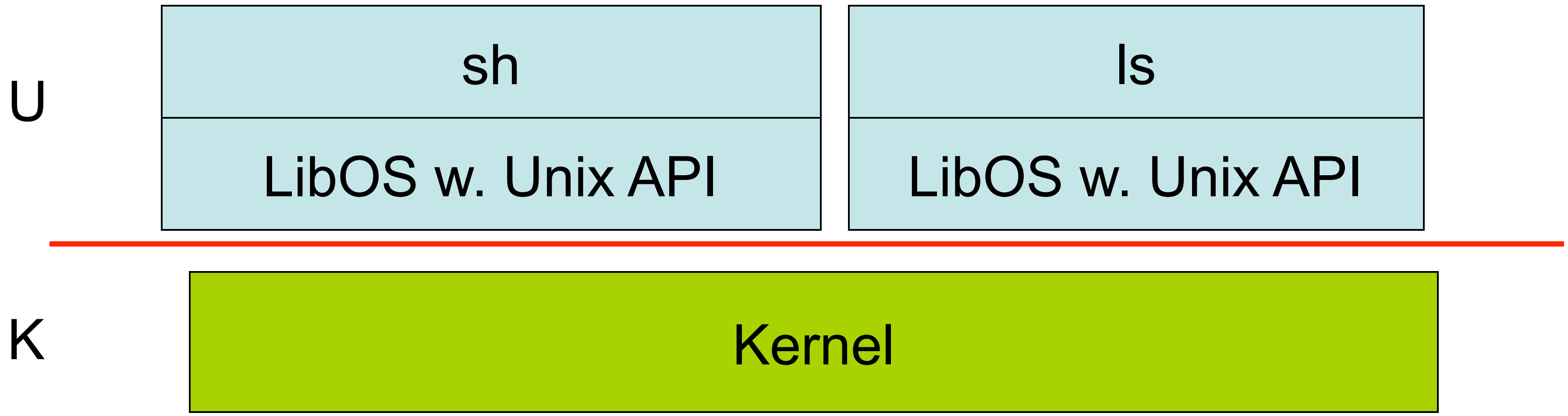
- Each program has its own page map
  - Physical memory doesn’t have to be contiguous
- When switching program, switch page map
- Page maps stored in main memory

Protecting page maps: kernel and user mode



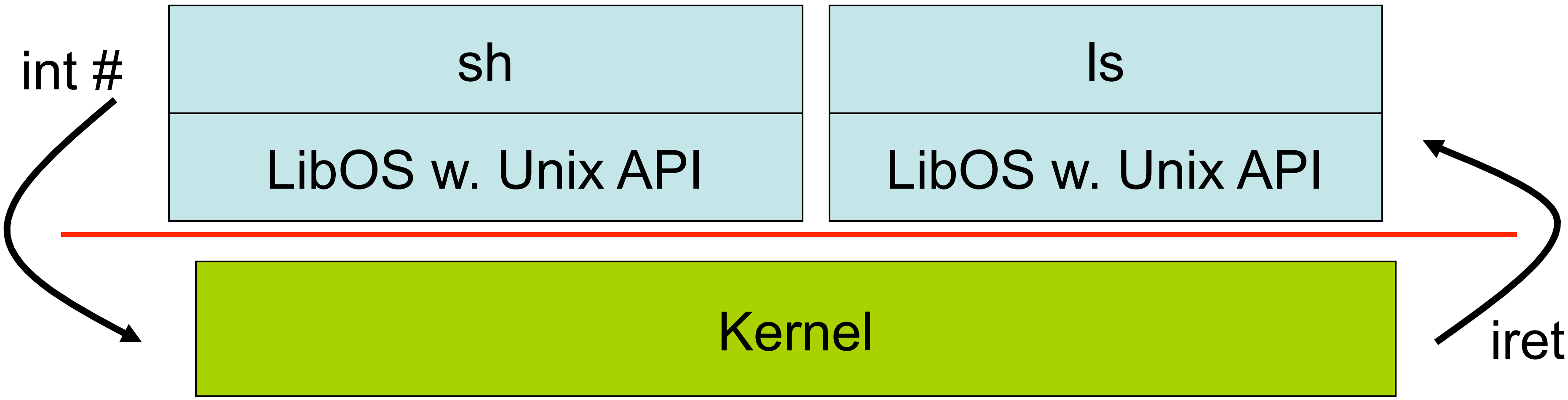
- Kernel mode: can change page-map register, U/K
- In user mode: cannot
- Processor starts in kernel mode
- On interrupts, processor switches to kernel mode

What is a kernel?



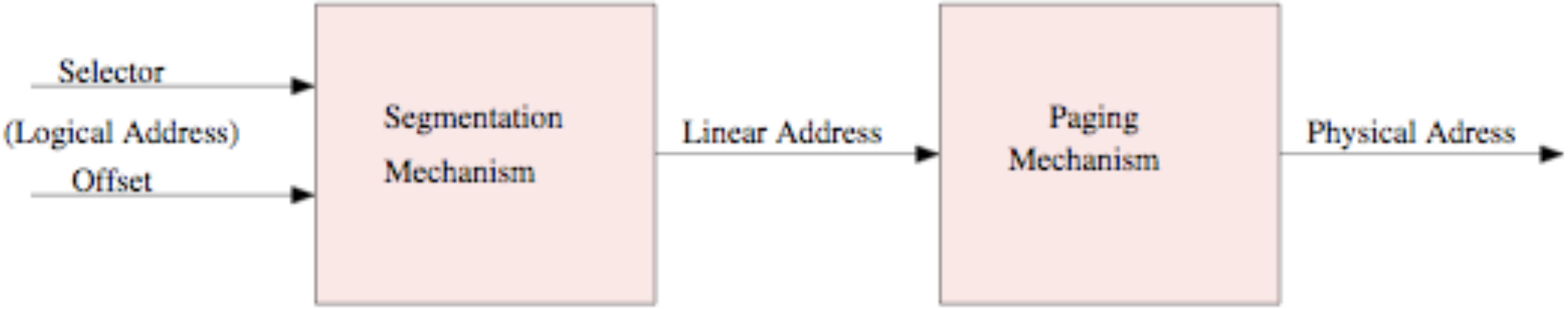
- The code running in kernel mode
  - Trusted program: e.g., sets page-map, U/K register
  - Enforces modularity

Entering the kernel: system calls



- Special instructions
  - Switches U/K bit
- Enter kernel at kernel-specified addresses

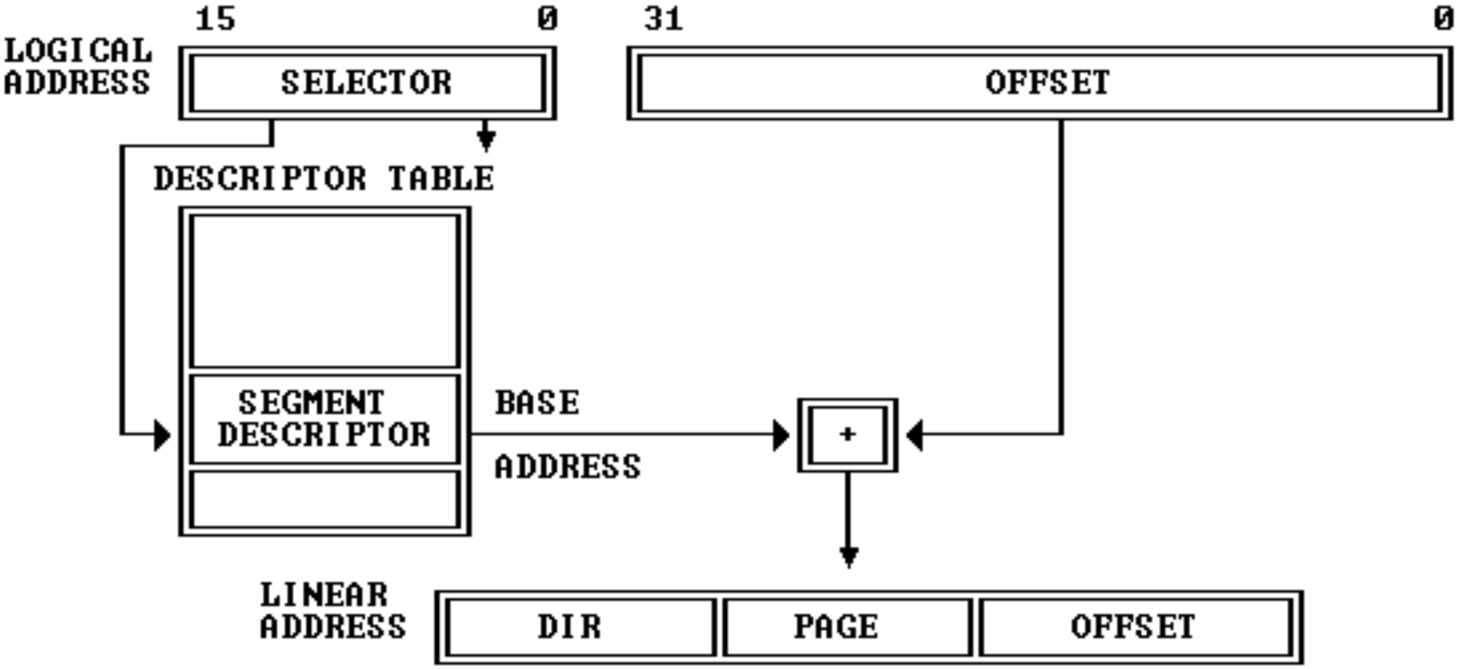
x86 virtual addresses



Protected–mode address translation

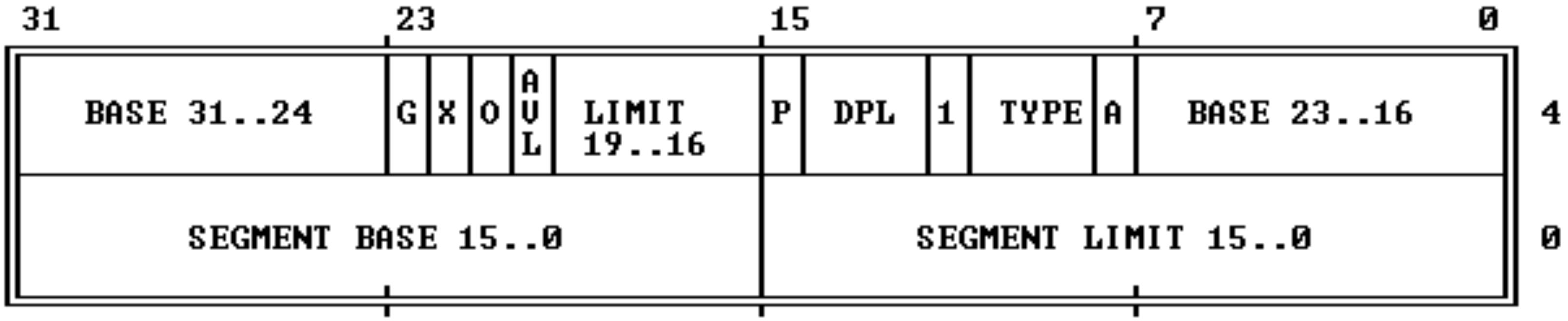
- x86 starts in real mode (no protection)
  - – segment registers (cs, ss, ds, es)
  - – segment * 16 + offset ⇨physical address
- OS can switch to protected mode
  - – Segmentation and paging

Translation with segments



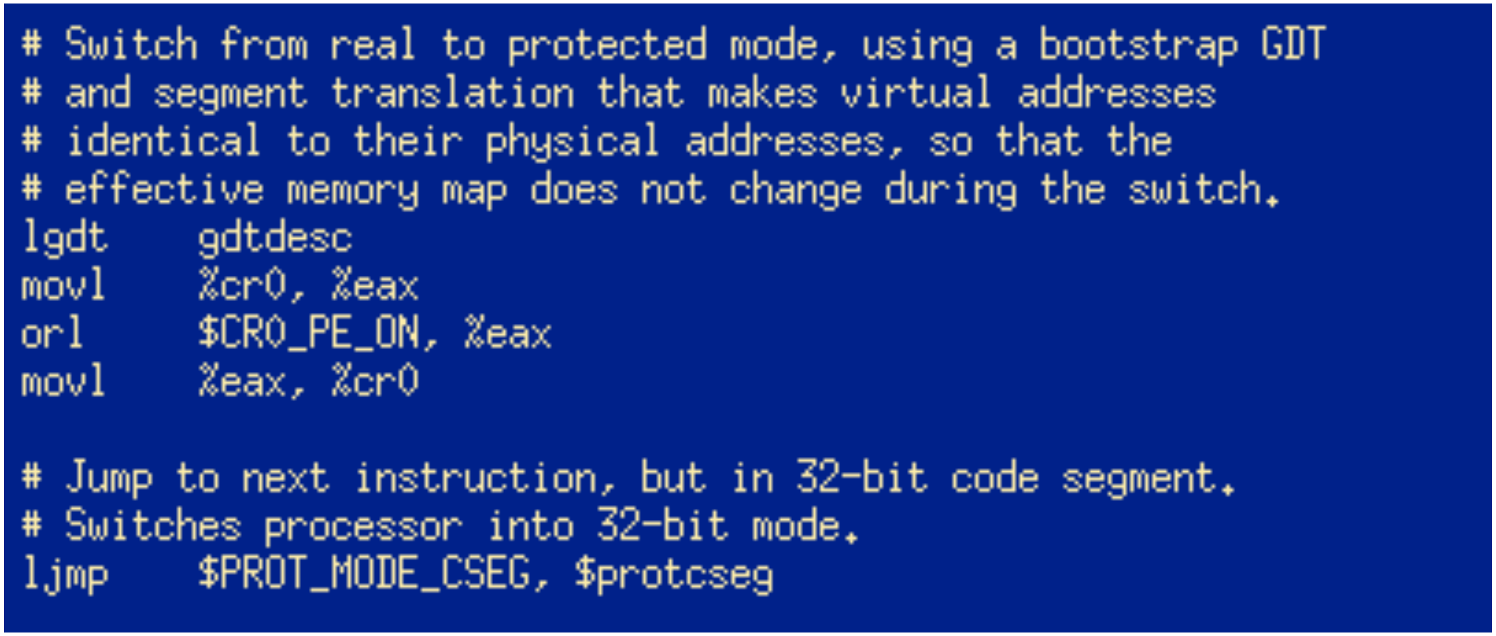
- LDGT loads CPU’s GDT
- PE bit in CR0 register enables protected mode
- Segments registers contain *index*

Segment descriptor



- Linear address = logical address + base
  - assert: logical address < limit
- Segment restricts what memory an application can reference

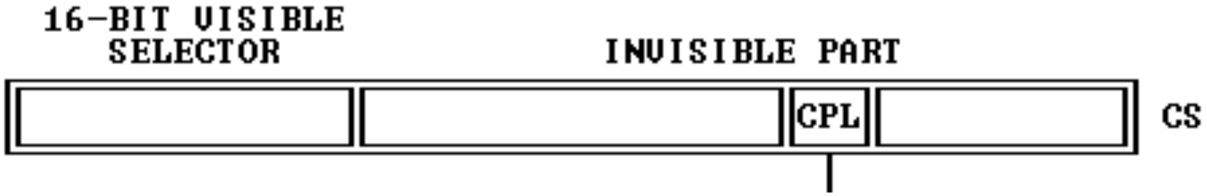
JOS code



```
# Switch from real to protected mode, using a bootstrap GDT
# and segment translation that makes virtual addresses
# identical to their physical addresses, so that the
# effective memory map does not change during the switch.
lgdt    gdtdesc
movl    %cr0, %eax
orl     $CR0_PE_ON, %eax
movl    %eax, %cr0

# Jump to next instruction, but in 32-bit code segment.
# Switches processor into 32-bit mode.
ljmp    $PROT_MODE_CSEG, $protcseg
```

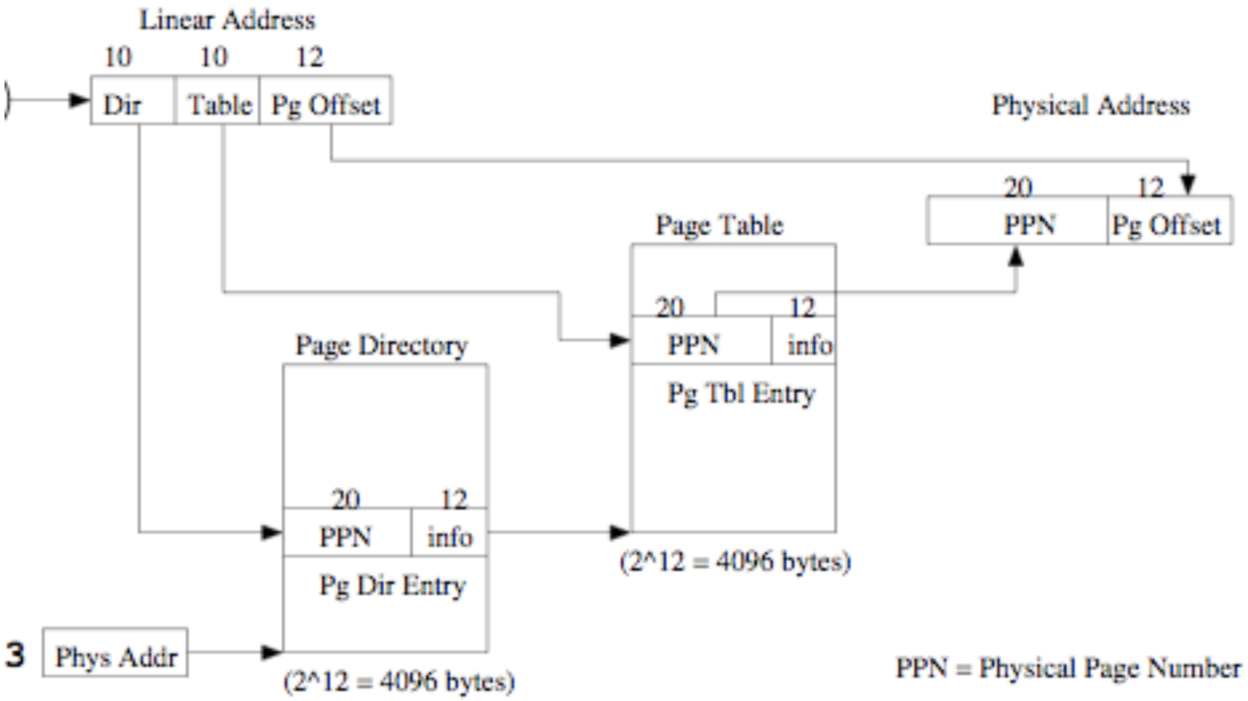
- Why does EIP contain the address of “ljmp” instruction after “movl %eax, %cr0”?

Enforcing modularity in x86



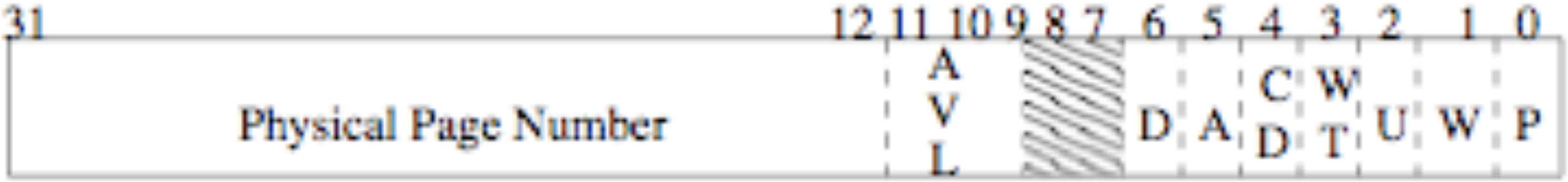
- CPL: current privilege level
  - 0: privileged (kernel mode)
  - 3: user mode
- User programs can set segment selector
- Kernel can load value in CPL and GDT, but user programs cannot

x86 two-level page table



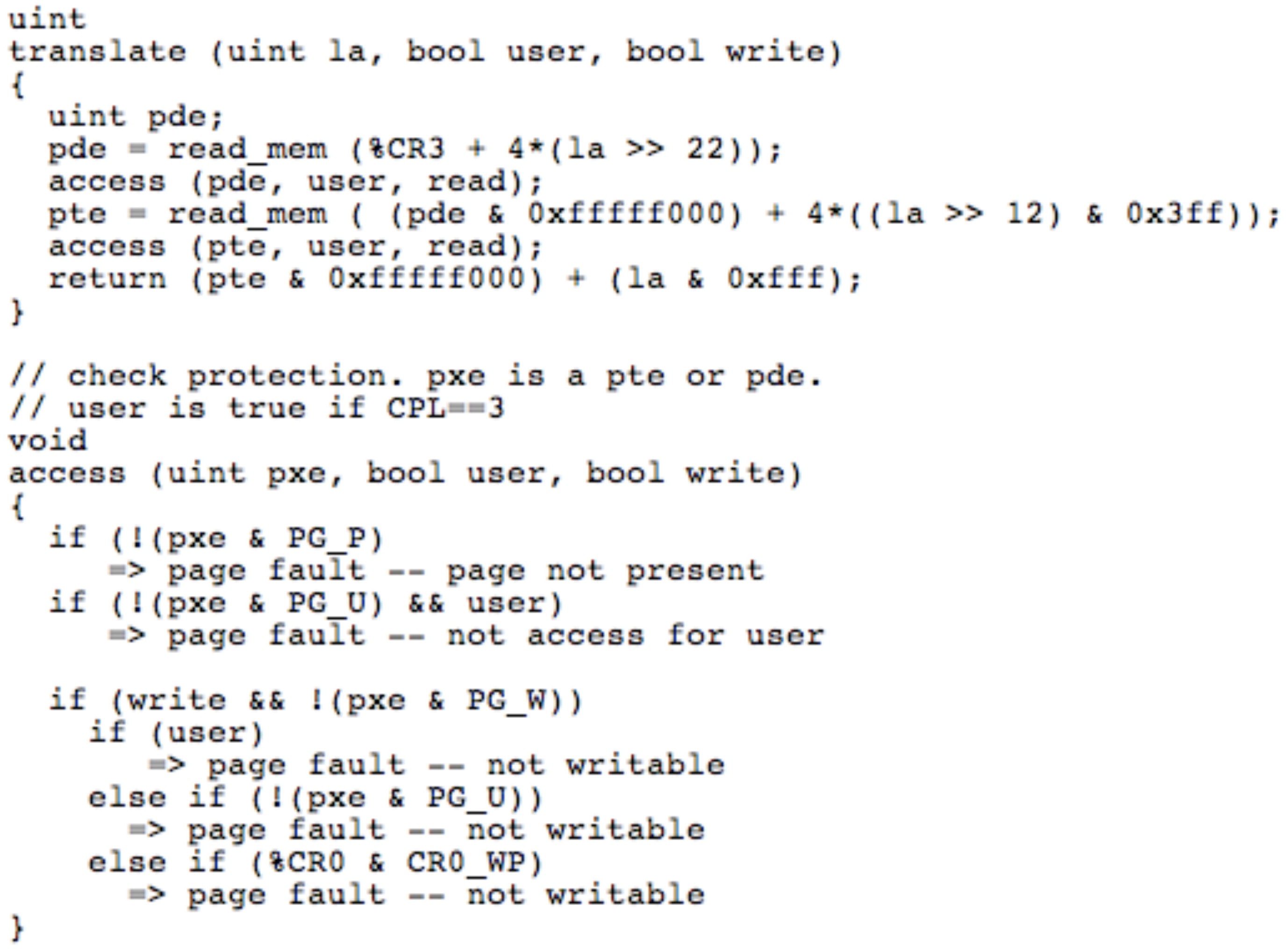
- Page size is 4,096 bytes
  - 1,048,576 pages in 2^{32}
  - Two-level structure to translate

x86 page table entry



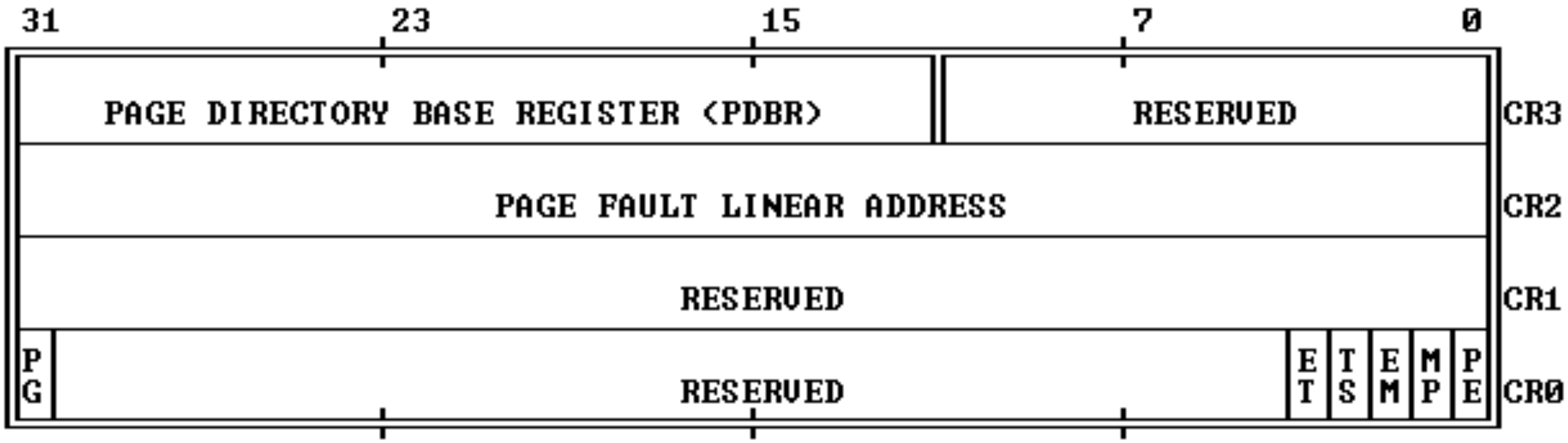
- W: writable?
  - Page fault when W = 0 and writing
- U: user mode references allowed?
  - Page fault when U = 0 and user references address
- P: present?
  - Page fault when P = 0

what does the x86 do exactly?



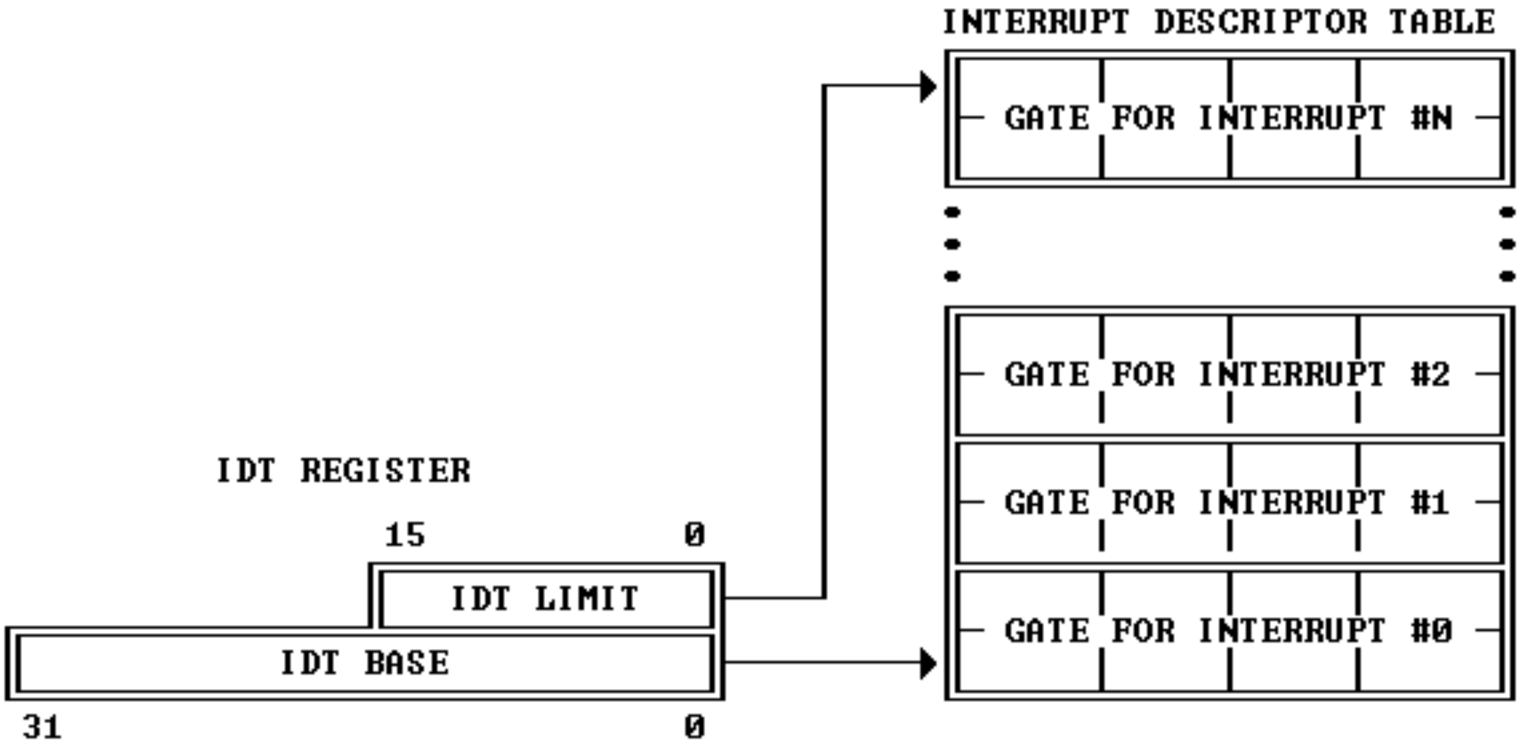
```
uint
translate (uint la, bool user, bool write)
{
  uint pde;
  pde = read_mem (%CR3 + 4*(la >> 22));
  access (pde, user, read);
  pte = read_mem ( (pde & 0xfffff000) + 4*((la >> 12) & 0x3ff));
  access (pte, user, read);
  return (pte & 0xfffff000) + (la & 0xfff);
}

// check protection. pxe is a pte or pde.
// user is true if CPL==3
void
access (uint pxe, bool user, bool write)
{
  if (!(pxe & PG_P)
     => page fault -- page not present
  if (!(pxe & PG_U) && user)
     => page fault -- not access for user

  if (write && !(pxe & PG_W))
    if (user)
       => page fault -- not writable
    else if (!(pxe & PG_U))
      => page fault -- not writable
    else if (%CR0 & CR0_WP)
      => page fault -- not writable
}
```

When does page table take effect?


31
23
15
7
0
PAGE DIRECTORY BASE REGISTER (PDBR)
RESERVED
CR3
PAGE FAULT LINEAR ADDRESS
CR2
RESERVED
CR1
PG
RESERVED
ET
TS
EM
MP
PE
CR0

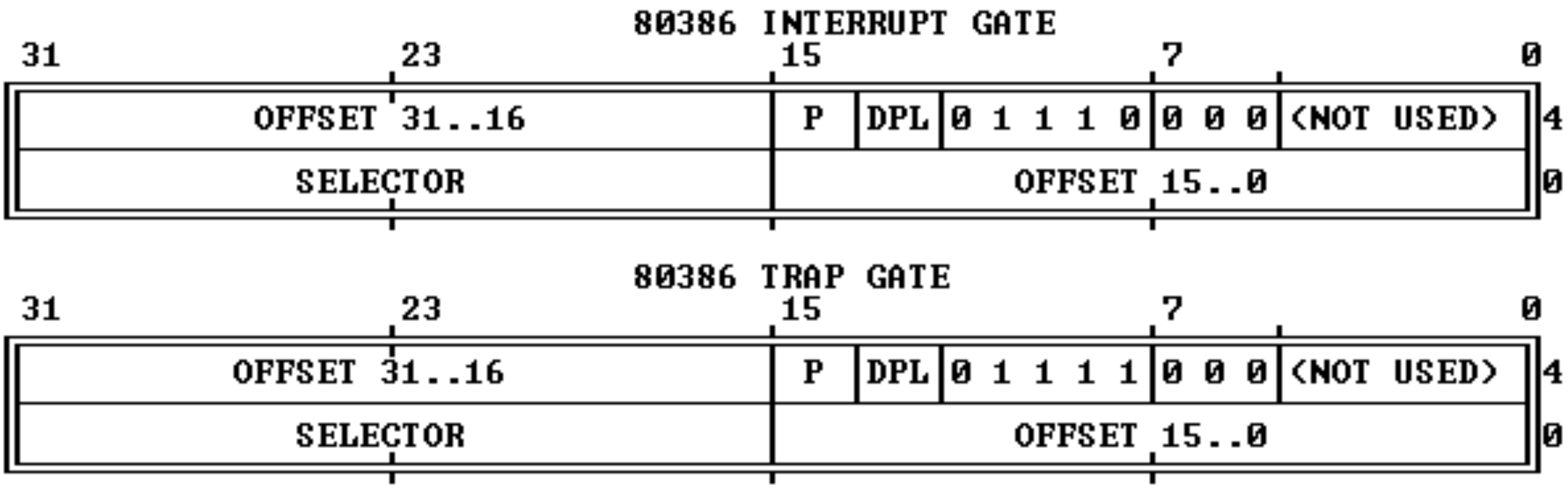
- PG enables page-based translation
- CR3 contains address of page table
  - Where does the next instruction come from?
- When changing PDE or PTE, you must flush TLB
  - Reload CR3

User mode to kernel mode



- Instruction: INT *n,* or interrupt
- *n* indexes into interrupt descriptor table (IDT)
- IDTR contains physical address of IDT

IDT descriptor



80386 INTERRUPT GATE

31 23	15			7	0	
OFFSET 31..16	P	DPL	0 1 1 1 0	0 0 0	<NOT USED>	4
SELECTOR	OFFSET 15..0					0

80386 TRAP GATE

31 23	15			7	0	
OFFSET 31..16	P	DPL	0 1 1 1 1	0 0 0	<NOT USED>	4
SELECTOR	OFFSET 15..0					0

- Three ways to get into kernel:
  - User asks (trap)
  - Page fault (trap)
  - Interrupts

What happens on trap/interrupt?

1. CPU uses vector n to index into IDT
2. Checks that CPL ≤ DPL
3. Saves ESP and SS in internal register
4. Loads ESP and SS from TSS
5. Push user SS
6. Push user ESP
7. Push user EFLAGS
8. Push user CS
9. Push user EIP
10. Clear some EFLAGS bits
11. Set CS and EIP from IDT descriptor

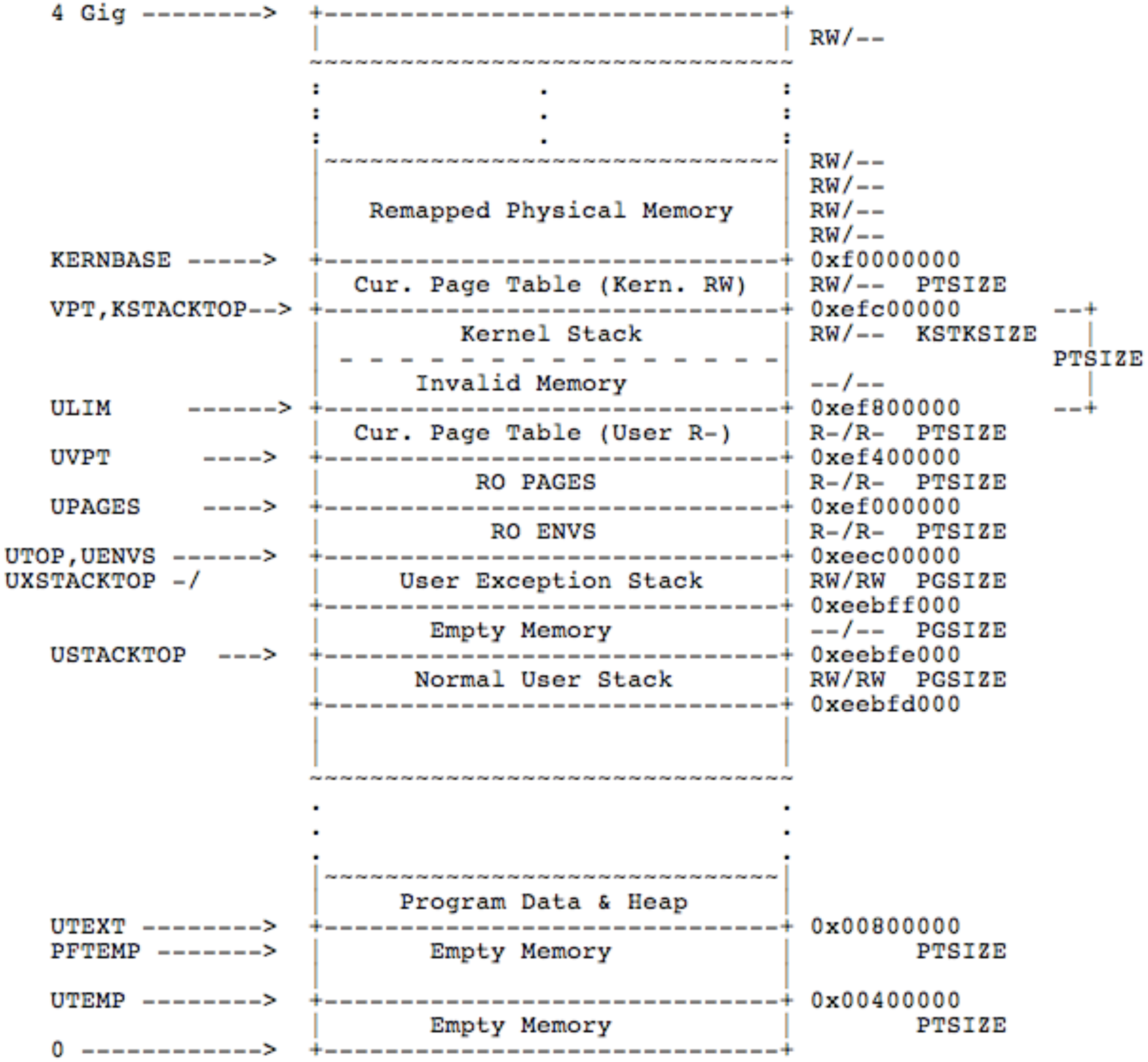
From kernel to user

- IRET instruction
  - Reverse of INT

Labs

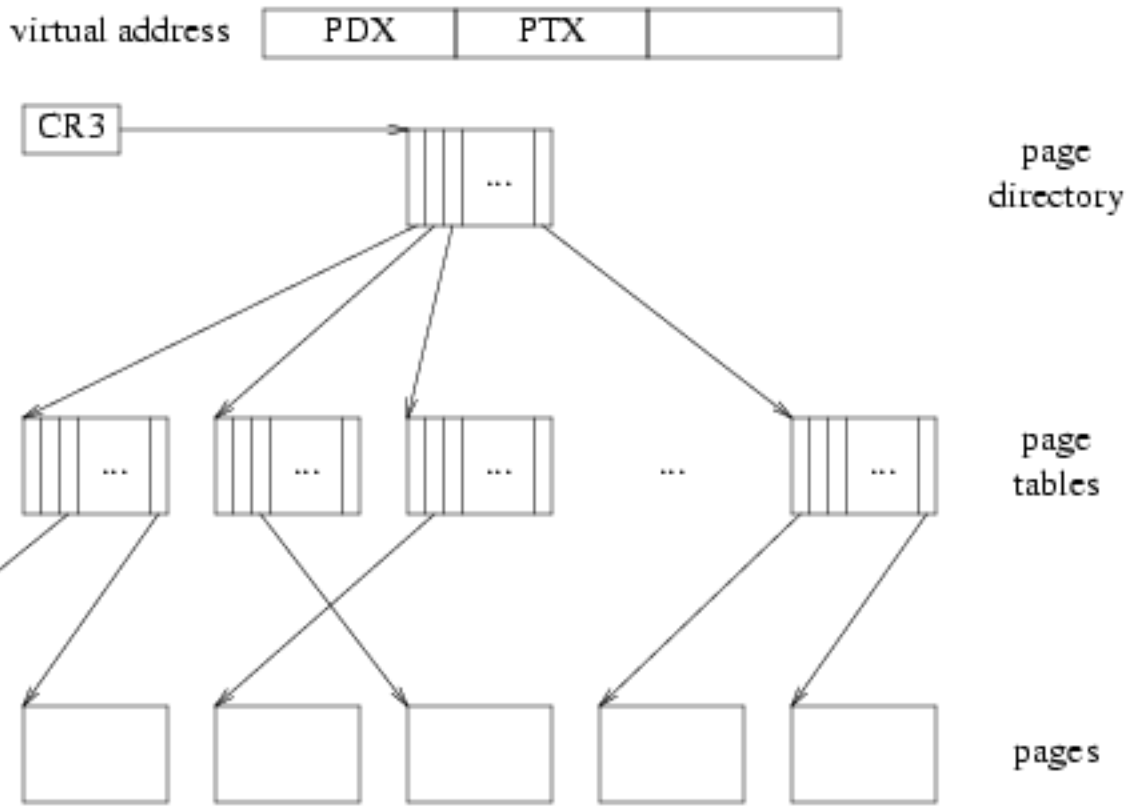
- Lab 1: start kernel
  - setup and use segmentation
- Lab 2: kernel
  - Set up kernel address space
- Lab 3: user/kernel
  - Set up user address space
  - Set up IDT
  - System calls and page faults
- Lab 4: many user programs
  - Preemptive scheduling

JOS



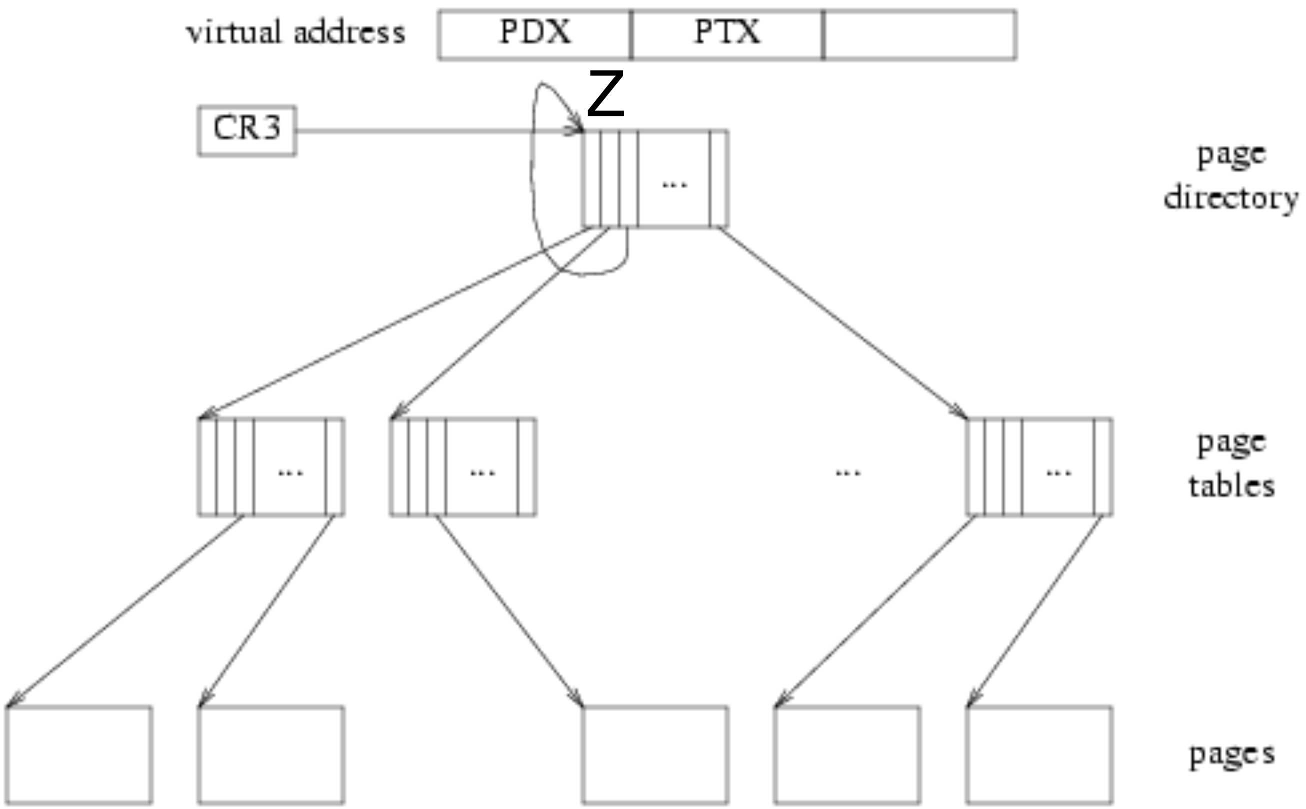
```
    4 Gig -------->  +------------------------------+
                     |                              | RW/--
                     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                     :              .               :
                     :              .               :
                     :              .               :
                     |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~| RW/--
                     |                              | RW/--
                     |   Remapped Physical Memory   | RW/--
                     |                              | RW/--
    KERNBASE ----->  +------------------------------+ 0xf0000000
                     |  Cur. Page Table (Kern. RW)  | RW/--  PTSIZE
    VPT,KSTACKTOP--> +------------------------------+ 0xefc00000      --+
                     |         Kernel Stack         | RW/--  KSTKSIZE   |
                     | - - - - - - - - - - - - - - -|                 PTSIZE
                     |        Invalid Memory        | --/--             |
    ULIM     ------> +------------------------------+ 0xef800000      --+
                     |  Cur. Page Table (User R-)   | R-/R-  PTSIZE
    UVPT      ---->  +------------------------------+ 0xef400000
                     |          RO PAGES            | R-/R-  PTSIZE
    UPAGES    ---->  +------------------------------+ 0xef000000
                     |           RO ENVS            | R-/R-  PTSIZE
 UTOP,UENVS ------>  +------------------------------+ 0xeec00000
 UXSTACKTOP -/       |     User Exception Stack     | RW/RW  PGSIZE
                     +------------------------------+ 0xeebff000
                     |         Empty Memory         | --/--  PGSIZE
    USTACKTOP  --->  +------------------------------+ 0xeebfe000
                     |      Normal User Stack       | RW/RW  PGSIZE
                     +------------------------------+ 0xeebfd000
                     |                              |
                     |                              |
                     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                     .                              .
                     .                              .
                     .                              .
                     |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|
                     |     Program Data & Heap      |
    UTEXT -------->  +------------------------------+ 0x00800000
    PFTEMP ------->  |         Empty Memory         |        PTSIZE
                     |                              |
    UTEMP -------->  +------------------------------+ 0x00400000
                     |         Empty Memory         |        PTSIZE
    0 ------------>  +------------------------------+
```

Recall x86 page table



- To find P for V OS can walk PT manually

VPT: Mapping the page table



- Z|Z maps to the page directory
- Z|V maps to V’s page table entry

Summary

- Kernel enforcing modularity
  - By switching processor between programs
  - By giving each program its own virtual memory
- x86 support for enforcing modularity
  - Segments
  - User and kernel mode
  - Page tables
  - Interrupts and traps
- JOS