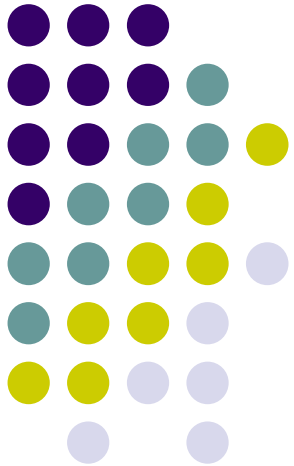
What You Need to Know for Project One

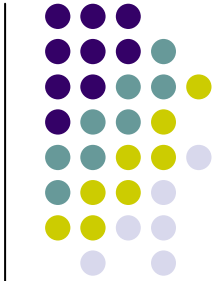
Steve Muckle

Monday, January 20 2003

15-412 Spring 2003



Overview

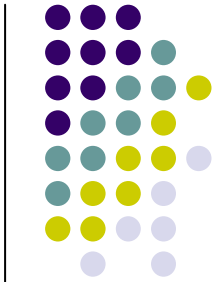


- Introduction
- Project One Motivation and Demo
- Mundane Details in x86
  PIC and hardware interrupts, software interrupts and exceptions, the IDT, privilege levels, segmentation
- Writing a Device Driver
- Installing and Using Simics

This is your life before today.

This is your life until May 2nd.

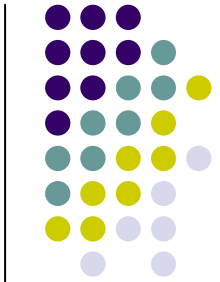
Just kidding…



- My name is Steve
- I hope to make things bearable
- Feel free to stop by outside of office hours

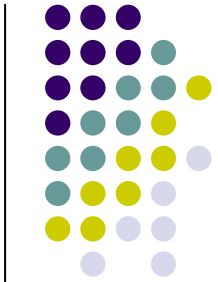
- New project set this semester!
  - should be fun, more realistic
  - improved tools

Project 1 Motivation



- What are our hopes for project 1?
  - introduction to kernel programming
  - need better understanding of the x86 arch
  - hands-on experience with hardware interrupts and device drivers
  - get acquainted with the simulator (Simics) and development tools (cons)

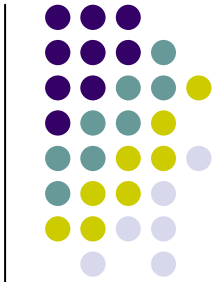
Project 1 Demo



- Project 1 consists of using the console, keyboard and timer to create a simple clock

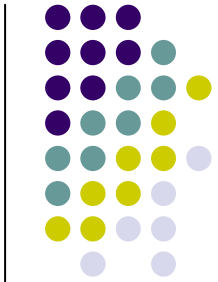
- Demo…

Mundane Details in x86

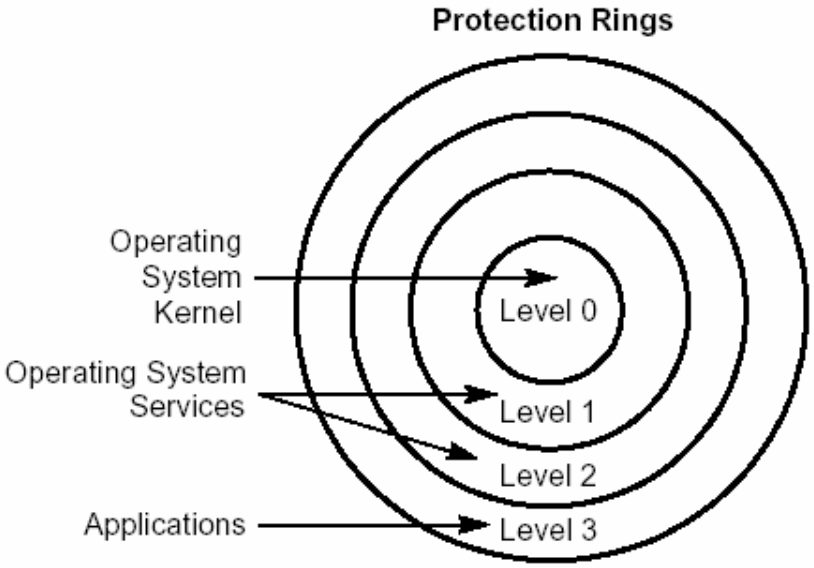


- Kernels work closely with hardware
- This means you need to know about hardware
- Some knowledge (registers, stack conventions) is assumed from 15-213
- You will learn more x86 details as the semester goes on
- Use the Intel PDF files as reference (http://www.cs.cmu.edu/~412/projects.html)

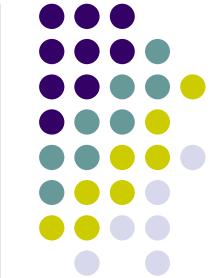
Mundane Details in x86: Privilege Levels



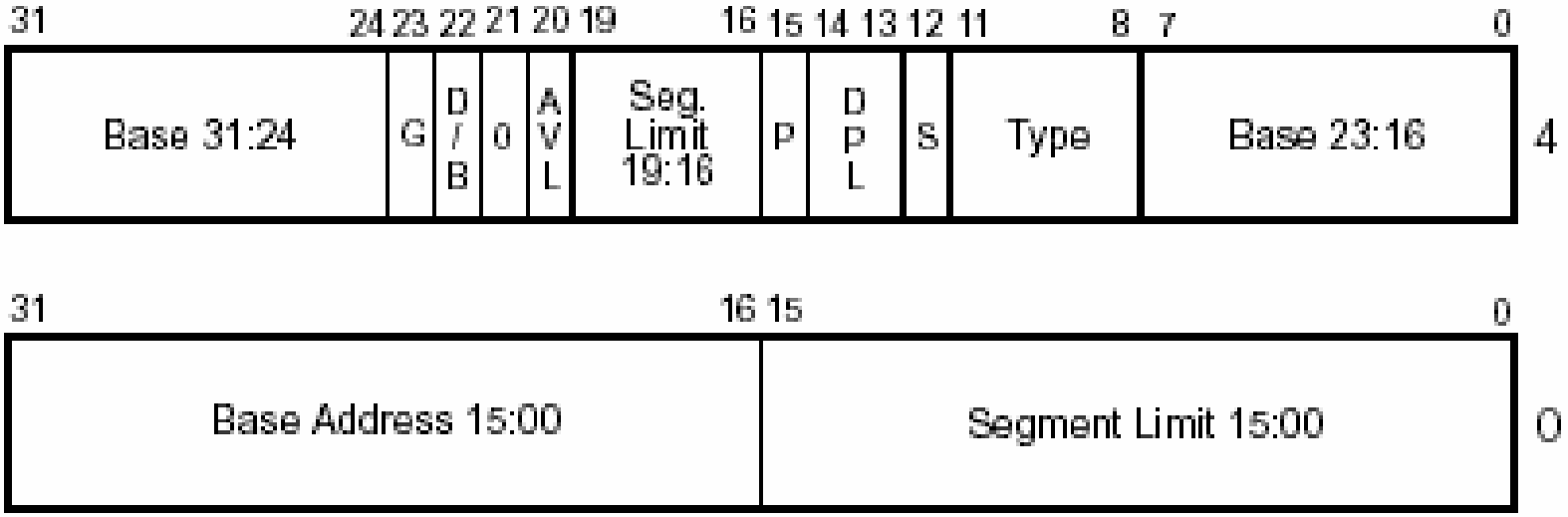
- Processor has 4 “privilege levels” (PLs)
- Zero most privileged, three least privileged
- Processor executes at one of the four PLs at any given time
- PLs protect privileged data, cause general protection faults



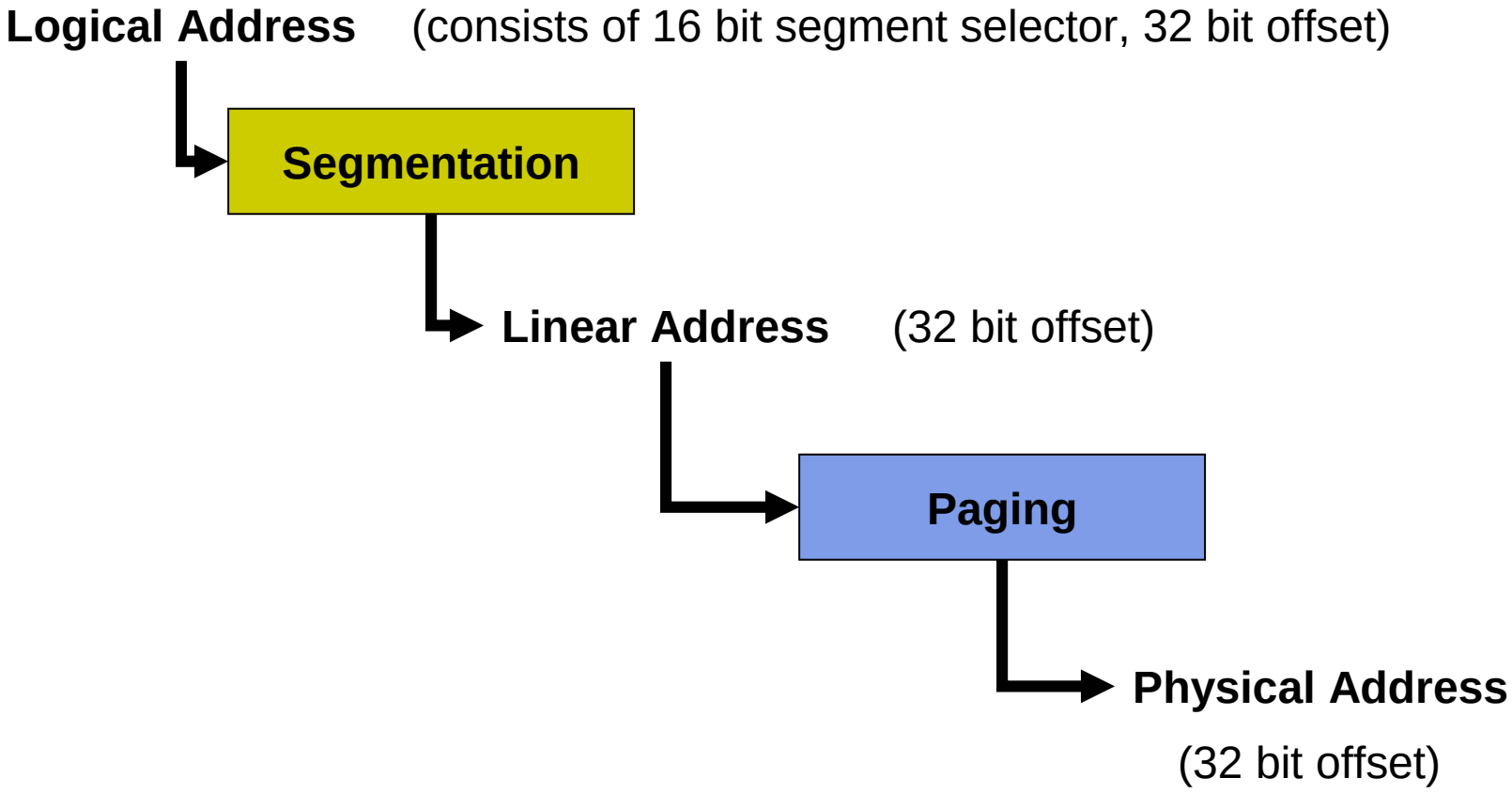
Mundane Details in x86: Segmentation



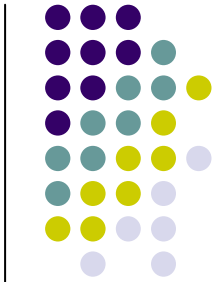
- One way to use PLs: segmentation
- Segments are defined areas of memory with particular access/usage constraints
- A segment descriptor looks like this:



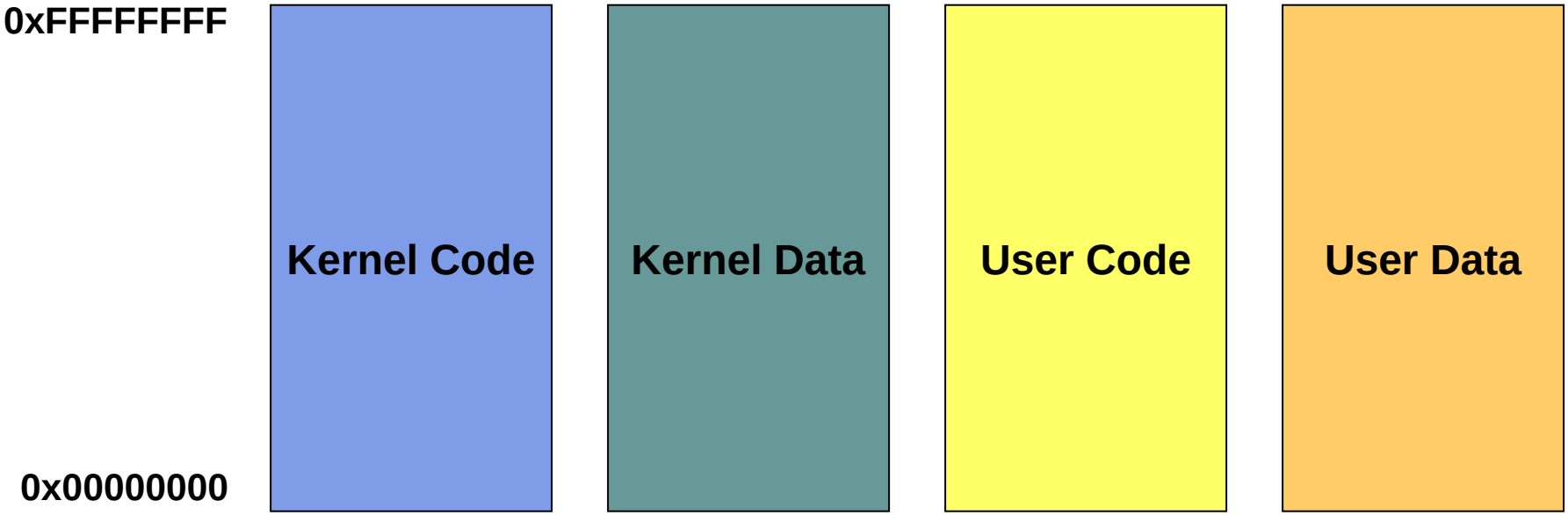
Mundane Details in x86: Segmentation



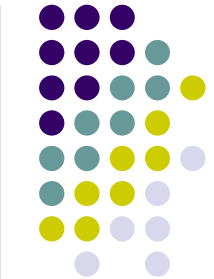
Mundane Details in x86: Segmentation



- Segments need not be backed by physical memory and can overlap
- Segments defined for these projects:



Mundane Details in x86: Getting into Kernel Mode



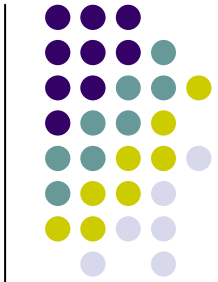
- How do we get from user mode (PL3) to kernel mode (PL0)?

  - Exception (divide by zero, etc)

  - Software Interrupt (*int n* instruction)

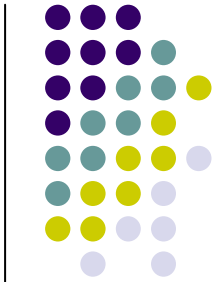
  - Hardware Interrupt (keyboard, timer, etc)

Mundane Details in x86: Exceptions



- Sometimes user processes do stupid things
- int gorganzola = 128/0;
- char* idiot_ptr = NULL; *idiot_ptr = 0;
- These cause a handler routine to be executed at PL0
- Examples include divide by zero, general protection fault, page fault

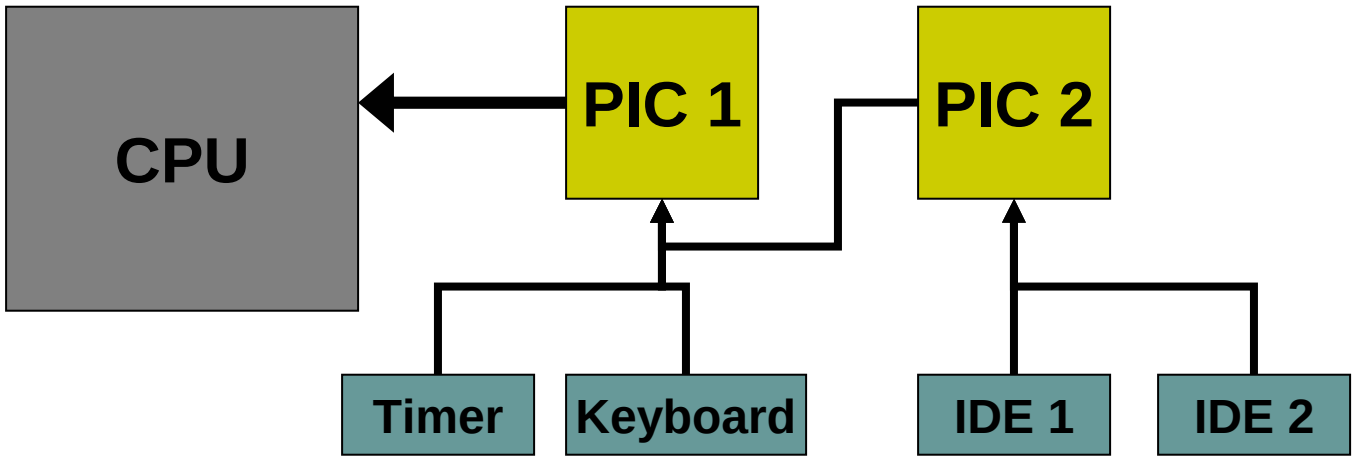
Mundane Details in x86: Software Interrupts



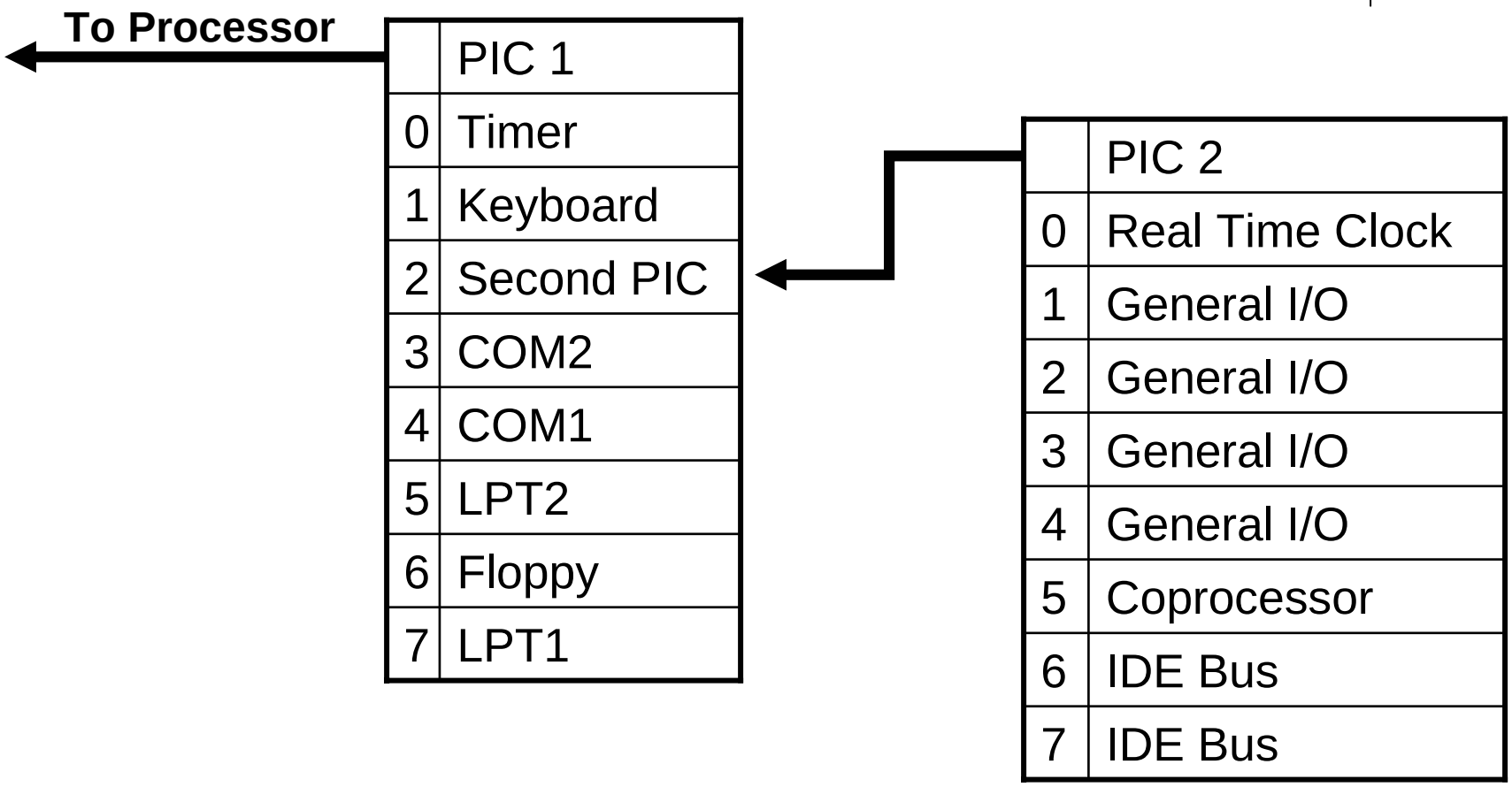
- A device gets the kernel's attention by raising an interrupt
- User processes get the kernel's attention by raising a software interrupt
- x86 instruction *int n*
  (more info on page 346 of intel-isr.pdf)
- Executes handler routine at PL0

Mundane Details in x86: Interrupts and the PIC

- Devices raise interrupts through the Programmable Interrupt Controller (PIC)
- The PIC serializes interrupts, delivers them
- There are actually two daisy-chained PICs

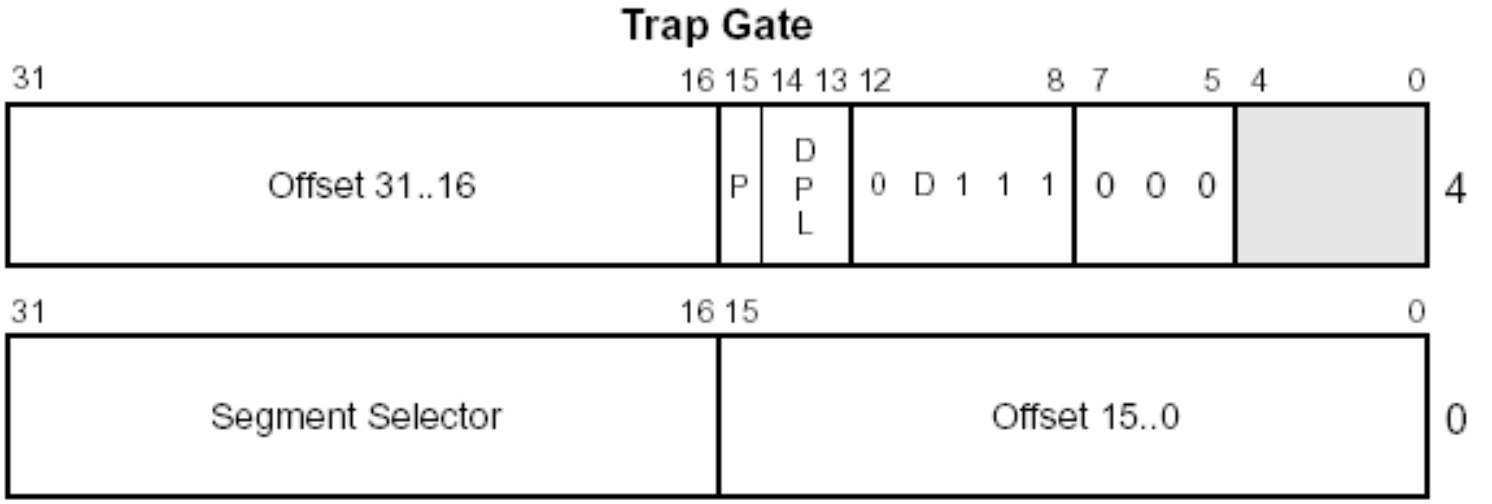


Mundane Details in x86: Interrupts and the PIC



Mundane Details in x86: Interrupt Descriptor Table (IDT)

- Processor needs info on what handler to run when
- Processor reads appropriate IDT entry depending on the interrupt *OR* exception *OR int n* instruction
- An entry in the IDT looks like this:



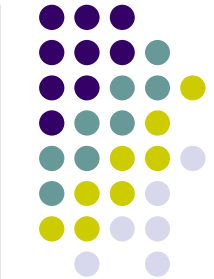
Mundane Details in x86: Interrupt Descriptor Table (IDT)

- The first 32 entries in the IDT correspond to processor exceptions. 32-255 correspond to hardware/software interrupts.
- Some interesting entries:

IDT Entry	Interrupt
0	Divide by zero
14	Page fault
32	Keyboard

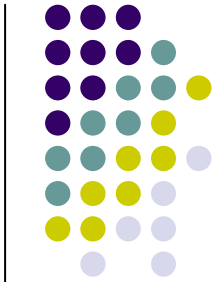
More information in section 5.12 of intel-sys.pdf.

Mundane Details in x86: Communicating with Devices



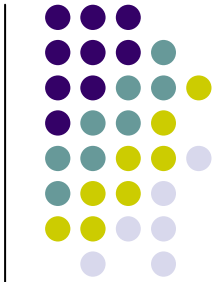
- I/O Ports
  - use instructions like inb, outb
  - use separate address space
- Memory-Mapped I/O
  - magic areas of memory tied to devices
  - console is one of them

Writing a Device Driver



- Traditionally consist of two separate halves
  - named “top” and “bottom” halves
  - BSD and Linux use these names differently
- One half is interrupt driven, executes quickly, queues work
- The other half processes queued work at a more convenient time

Installing and Using Simics

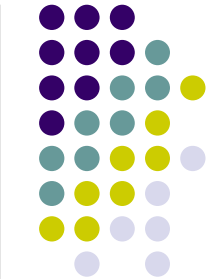


- Simics is an instruction set simulator
- Makes testing kernels MUCH easier
- Runs on both x86 and Solaris

Installing and Using Simics: Running on AFS

- We use mtools to copy to disk image files
- Proj1 Makefile sets up config file for you
- You must exec simics in your project dir
- The proj1.tar.gz includes:
  - simics-linux.sh
  - simics-solaris.sh

Installing and Using Simics: Running on Personal PC



- Runs under Linux, Solaris
- As of now you need a 128.2.*.* IP
- Download simics-linux.tar.gz from /afs/andrew.cmu.edu/scs/cs/15-412b/
- Install mtools RPM (pointer on course www)
- Install OSkit libs (directions on course www)
- Tweak Makefile

Installing and Using Simics: Debugging

- Run simulation with r, stop with ctl-c
- Magic instruction
  - xchg %bx,%bx (wrapper in interrupts.h)
- Memory access breakpoints
  - break 0x2000 –x *OR* break (sym init_timer)
- Symbolic debugging
  - psym foo *OR* print (sym foo)
- Demo