What You Need to Know for Project One

Bruce Maggs
Dave Eckhardt
Joey Echeverria
Steve Muckle

Synchronization

1. Please ***read*** the syllabus
    a) Some of your questions are answered there :-)
    b) We would rather teach than tear our hair out
2. Also the Project 1 handout
    a) Please don't post about “Unexpected interrupt 0”
3. Reflections on Project 0...

Overview

1. Introduction
2. Project One Motivation
3. Mundane Details (x86/IA-32 version)
   PICs, hardware interrupts, software interrupts and exceptions, the IDT, privilege levels, segmentation
4. Writing a Device Driver
5. Installing and Using Simics

Project 1 Motivation

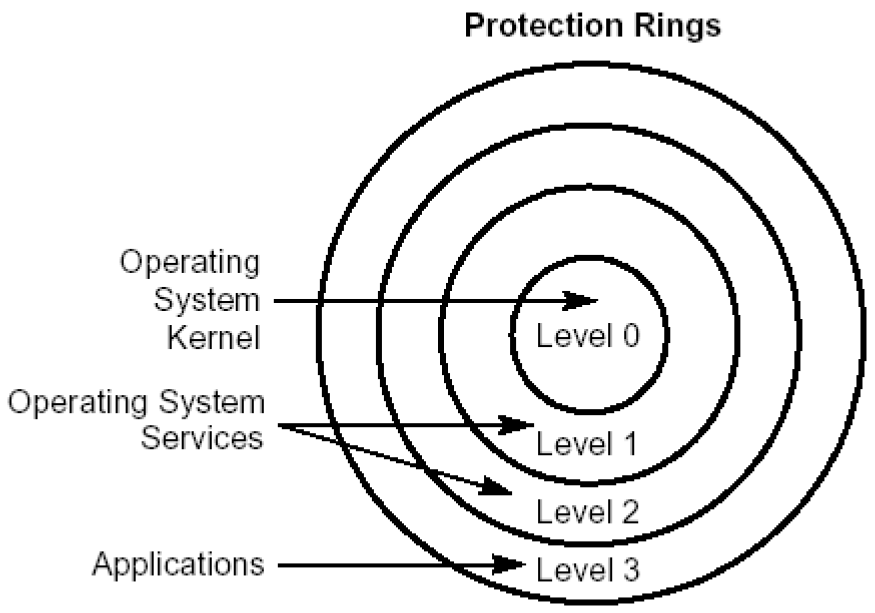
1. What are our hopes for project 1?
   a) introduction to kernel programming
   b) a better understanding of the x86 arch
   c) hands-on experience with hardware interrupts and device drivers
   d) get acquainted with the simulator (Simics) and development tools

Mundane Details in x86

1. Kernels work closely with hardware
2. This means you need to know about hardware
3. Some knowledge (registers, stack conventions) is assumed from 15-213
4. You will learn more x86 details as the semester goes on
5. Use the Intel PDF files as reference (http://www.cs.cmu.edu/~410/projects.html)

Mundane Details in x86: Privilege Levels

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



Mundane Details in x86: Privilege Levels

1. Essentially unused in Project 1
2. Projects 2 through 4
   a) PL0 is "kernel"
   b) PL3 is "user"
   c) Interrupts & exceptions usually transfer from 3 to 0
   d) Running user code means getting from 0 to 3

Memory Segmentation

1. There are different kinds of memory
2. Hardware
   a) Read-only memory (for booting)
   b) Video memory (painted onto screen)
   c) ...
3. Software
   a) Read-only memory (typically, program code)
   b) Stack (grows down), heap (grows up)
   c) ...

Memory Segmentation

1. Memory segment is a range of “the same kind”
2. Hardware
   a) Mark video memory as “don't buffer writes”
3. Software
   a) Mark all code pages read-only
4. Fancy software
   a) Process uses ***many*** separate segments
   b) Windows: each DLL is a segment

Memory Segmentation

1. x86 hardware ***loves*** segments
2. Mandatory segments
   a) Stack
   b) Code
   c) Data
3. Segments interact with privilege levels
   a) Kernel stack / user stack
   b) Kernel code / user code
   c) ...

Mundane Details in x86: Segmentation

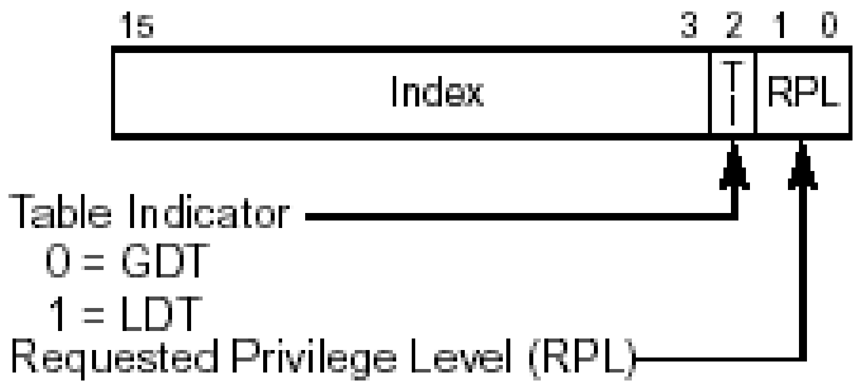
1. When fetching an instruction, the processor asks for an address that looks like this: %CS:%EIP
2. So, if %EIP is 0xbabe then %CS:%EIP is the 47806th byte of the "code segment".

x86 Segmentation Road Map

1. Segment = range of “same kind of memory”
2. Segment ***register*** = %CS, %SS, %DS, ... %GS
3. Segment ***selector*** = contents of a segment register
   a) Which segment do we mean (table, index)?
   b) What access privilege do we have to it?
4. Segment ***descriptor*** = definition of segment
   a) Which memory range?
   b) What are its properties

Mundane Details in x86: Segmentation

1. When fetching an instruction, the processor asks for an address that looks like this: %CS:%EIP
2. The CPU looks at the ***segment selector*** in %CS
3. A segment selector looks like this:



Mundane Details in x86: Segmentation

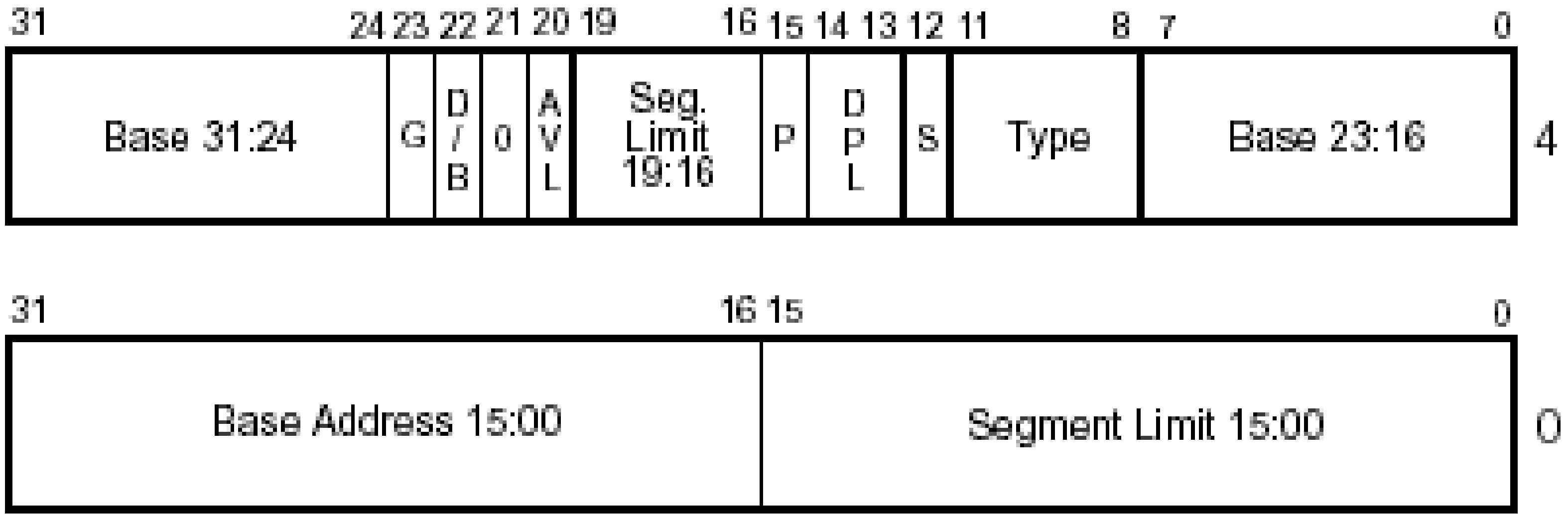
1. Segment selector has a segment number, table selector, and requested privilege level (RPL)
2. The table-select flag selects a descriptor table
   a) *global descriptor table* or *local descriptor table*
3. Segment number indexes into that descriptor table
   a) 15-410 uses only global descriptor table (whew!)
4. Descriptor tables set up by operating system
   a) 15-410 support code makes GDT for you (whew!)
5. You will still need to understand this, though...

Mundane Details in x86: Segmentation

1. Segment selector has a segment number, table selector, and requested privilege level (RPL)
2. Table selector (done)
3. Segment number/index (done)
4. RPL ***generally*** means “what access do I have?”
5. Magic special case: RPL in %CS
   a) Defines ***current processor privilege level***
   b) Think: “user mode” vs. “kernel mode”
   c) Remember this for Project 3!!!

Mundane Details in x86: Segmentation

1. Segments = area of memory with particular access/usage constraints
2. Base, size, “stuff”
3. Layout:



Mundane Details in x86: Segmentation

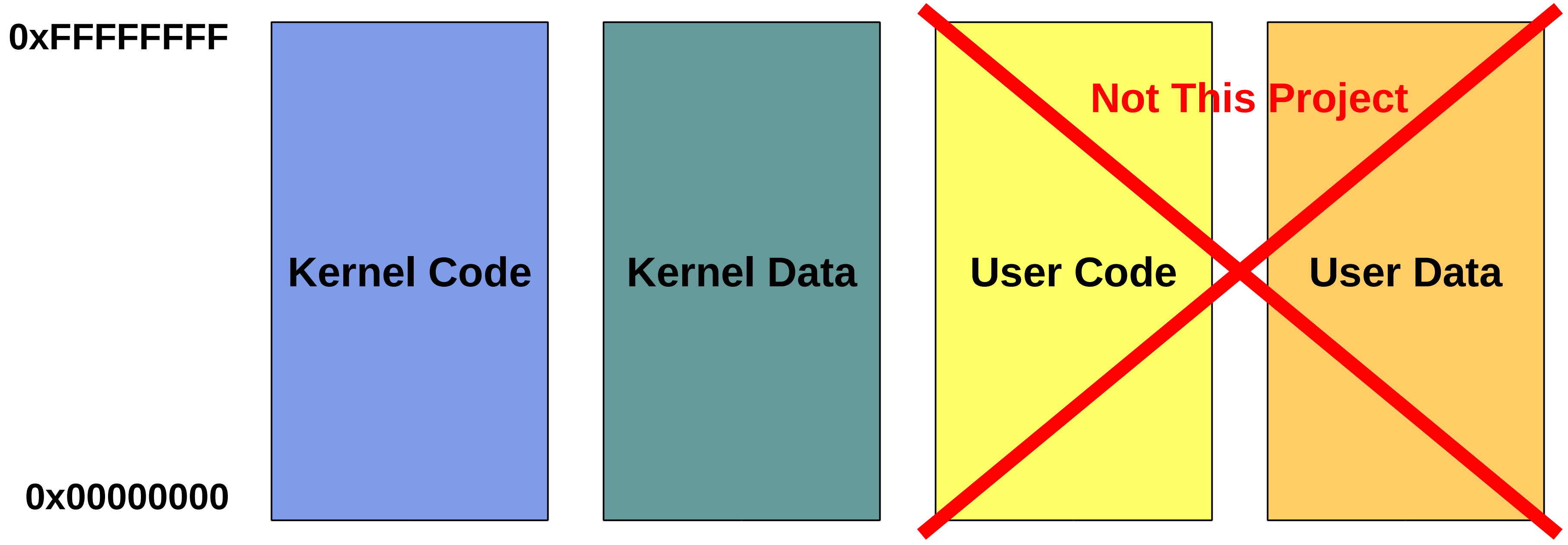
1. Consider %CS segment register's segment selector's segment descriptor
2. Assume base = 0xcafe0000
3. Assume limit > 47806
4. Then %CS:%EIP means “linear virtual address” 0xcafebabe
5. “Linear virtual address” fed to virtual memory system (Project 3)

Implied Segment Registers

1. Programmer doesn't usually ***specify*** segment
2. Usually ***implied*** by “kind of memory access”
3. CS is the segment register for fetching code
4. SS is the segment register for the stack segment
   a) Implied by PUSH and POP family
5. DS is the default segment register for data access
   a) But ES, FS, and GS can also be used

Mundane Details in x86: Segmentation

1. Segments need not be backed by physical memory and can overlap
2. Segments defined for 15-410:



Mundane Details in x86: Segmentation

1. Why so many?
2. You can’t specify a segment that is readable, writable and executable.
3. Therefore one for readable/executable code
4. Another for readable/writable data
5. Need user and kernel segments in Project 3 for protection
6. (Code, Data) X (User, Kernel)

Mundane Details in x86: Segmentation

1. Don’t need to be concerned with every detail of segments in this class
2. For more information you can read the Intel docs
3. Or our documentation at:
4. http://www.cs.cmu.edu/~410/doc/segments/segments.html

Mundane Details in x86: Getting into Kernel Mode

1. How do we get from user mode (PL3) to kernel mode (PL0)?
   a) Exception (divide by zero, etc)

   b) Software Interrupt (**`INT n`** instruction)

   c) Hardware Interrupt (keyboard, timer, etc)

Mundane Details in x86: Exceptions

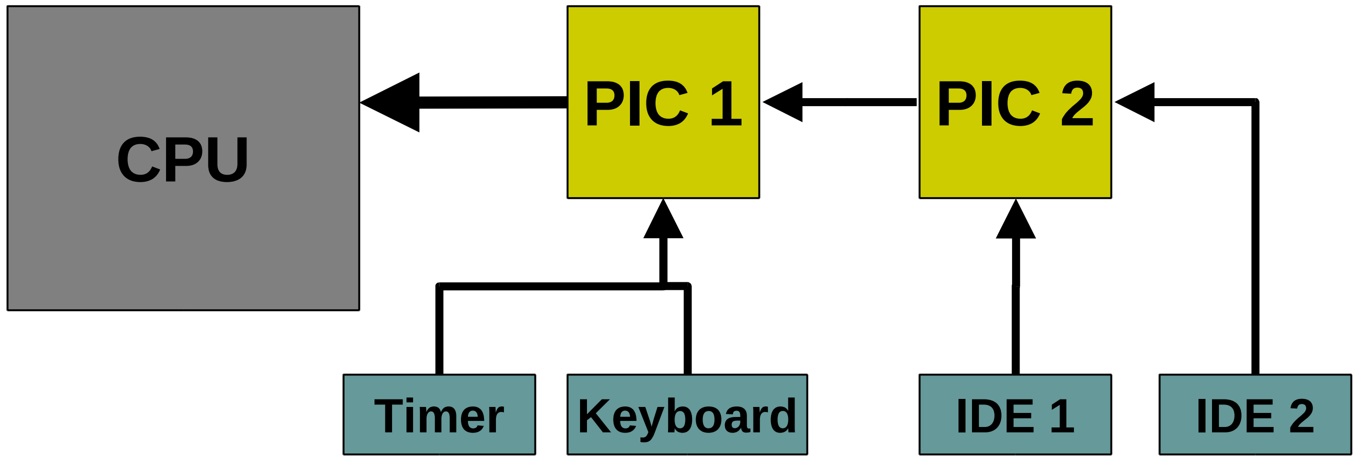
1. Sometimes user processes do stupid things
2. int gorgonzola = 128/0;
3. char* idiot_ptr = NULL; *idiot_ptr = 0;
4. These cause a handler routine to be executed
5. Examples include divide by zero, general protection fault, page fault

Mundane Details in x86: Software Interrupts

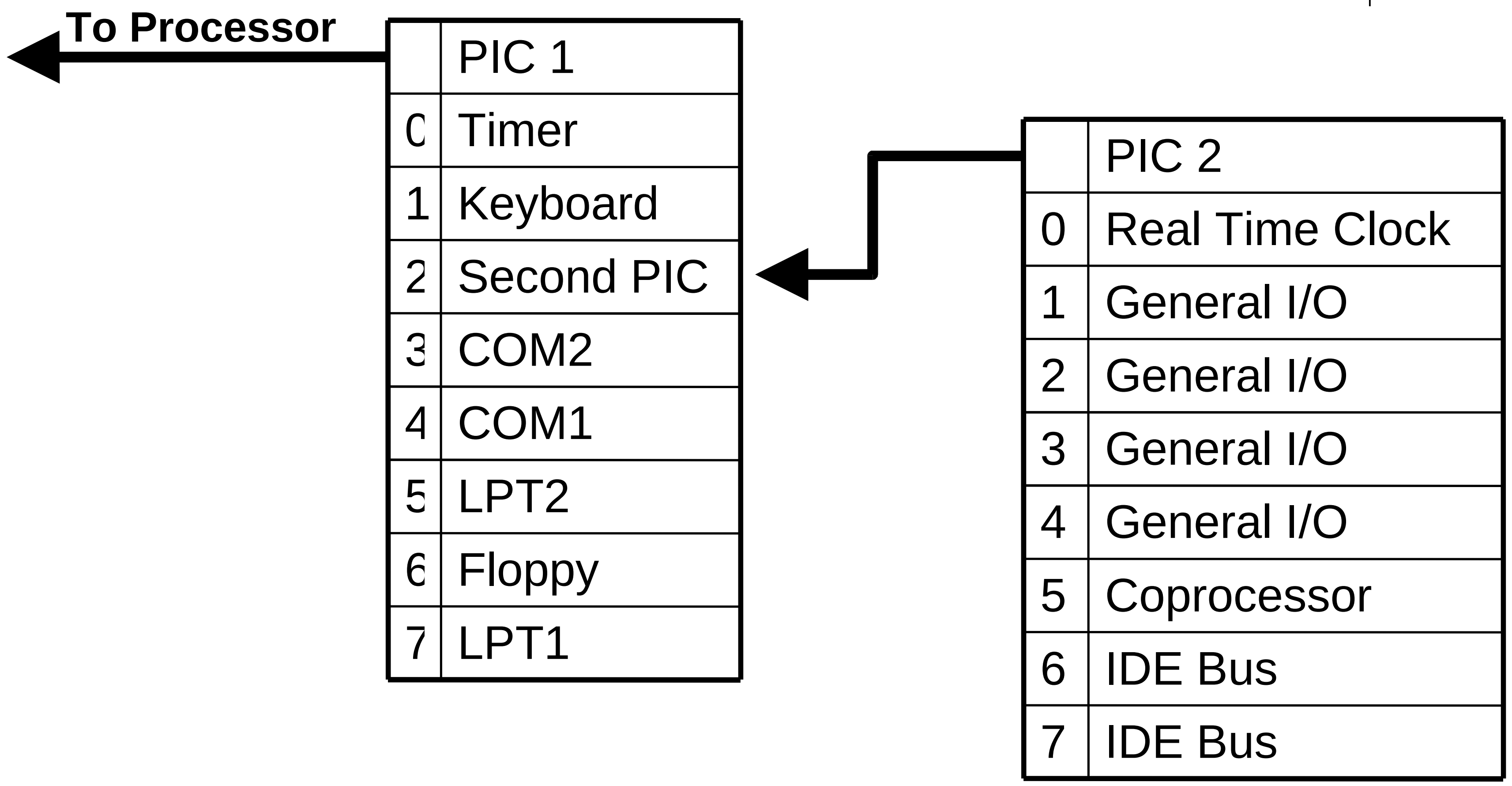
1. A device gets the kernel's attention by raising a (hardware) interrupt
2. User processes get the kernel's attention by raising a software interrupt
3. x86 instruction **`int n`**
   (more info on page 346 of intel-isr.pdf)
4. Invokes handler routine

Mundane Details in x86: Interrupts and the PIC

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



Mundane Details in x86: Interrupts and the PIC

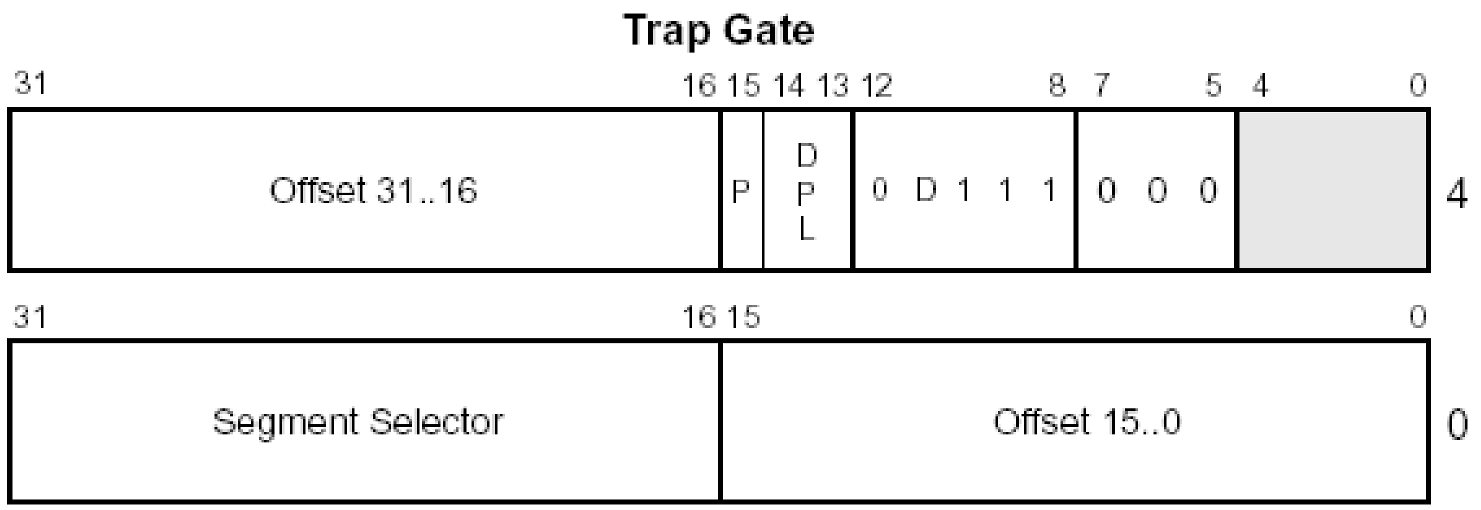


	PIC 1
0	Timer
1	Keyboard
2	Second PIC
3	COM2
4	COM1
5	LPT2
6	Floppy
7	LPT1

	PIC 2
0	Real Time Clock
1	General I/O
2	General I/O
3	General I/O
4	General I/O
5	Coprocessor
6	IDE Bus
7	IDE Bus

Interrupt Descriptor Table – IDT

1. Processor needs info on what handler to run when
2. Processor reads appropriate IDT entry depending on the interrupt, exception *or* **`int n`** instruction
3. An entry in the IDT looks like this:



Interrupt Descriptor Table (IDT)

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

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

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

Mundane Details in x86: Communicating with Devices

1. I/O Ports
   a) Use instructions like **`inb(port)`**, **`outb(port,data)`**
   b) ***Are not memory!***
2. Memory-Mapped I/O
   a) Magic areas of memory tied to devices
3. P1-P4 screen hardware uses ***both***
   a) Cursor controlled by I/O ports
   b) Characters painted from memory

x86 Device Perversity

1. Influence of ancient history
   a) IA-32 is fundamentally an 8-bit processor!
   b) Primeval I/O devices had 8-bit ports
2. I/O devices have multiple “registers”
   a) Timer: waveform type, counter value
   b) Screen: resolution, color depth, ***cursor position***
3. You must get the right value in the right register

x86 Device Perversity

1. Value/bus mismatch
   a) Counter value, cursor position are 16 bits
   b) Primeval I/O devices ***still*** have 8-bit ports
2. Typical control flow
   a) “I am about to tell you half of register 12”
   b) “32”
   c) “I am about to tell you the other half of register 12”
   d) “0”

x86 Device Perversity

1. Sample interaction
   a) `outb(command_port, SELECT_R12_LOWER);`
   b) `outb(data_port, 32);`
   c) `outb(command_port, SELECT_R12_UPPER);`
   d) `outb(data_port, 0);`
2. Now you are an expert device hacker!

Writing a Device Driver

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

Writing a Device Driver

1. For this project, your keyboard driver will likely have a top and bottom half
2. Bottom half
   a) Responds to keyboard interrupts and queues scan codes
3. Top half
   a) In readchar(), reads from the queue and processes scan codes into characters

Installing and Using Simics

1. Simics is an instruction set simulator
2. Makes testing kernels MUCH easier
3. Runs on both x86 and Solaris
    a) We haven't run it on Solaris in a year
    b) Because the cluster PCs ran faster than cluster SunBlades

Installing and Using Simics: Running on AFS

1. We use mtools to copy to disk image files
2. Proj1 Makefile sets up config file for you
3. You must run simics in your project dir
4. The proj1.tar.gz includes what you need

Installing and Using Simics: Running on AFS

1. Your 15-410 AFS space has p1, scratch
2. If you work in this space, we can read your files
3. Answering questions can be much faster

Installing and Using Simics: Running on Personal PC

1. Not a “supported configuration”
2. 128.2.*.* IP addresses can use campus license
3. You can apply for a personal single-machine Simics license (“Software Setup Guide” page)
4. Download simics-linux.tar.gz
5. Install mtools RPM
6. Tweak Makefile

Installing and Using Simics: Debugging

1. Run simulation with r, stop with ctl-c
2. Magic instruction
   a) **`xchg %bx,%bx`** (wrapper in interrupts.h)
3. Memory access breakpoints
   a) break 0x2000 –x *OR* break (sym init_timer)
4. Symbolic debugging
   a) psym foo *OR* print (sym foo)
5. See our local Simics hints (on Project page)

Summary

1. Project 1 runs on ***bare hardware***
   a) Not a machine-invisible language like ML or Java
   b) Not a machine-portable language like C
   c) Budget time for understanding this environment
2. Project 1 runs on ***simulated*** bare hardware
   a) You probably need more than printf() for debugging
   b) Not gdb
   c) Budget time to learn the Simics debugger

Summary

1. Project 1 runs on bare ***PC*** hardware
    a) As hardware goes, it's pretty irrational
    b) ***Almost nothing*** works “how you would do it”
    c) Those pesky diagrams do matter