What You Need to Know for Project One

Roger Dannenberg
Dave Eckhardt
Joey Echeverria
Steve Muckle
Joshua Wise

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 please read the Project 1 handout
   a) ***Please don't post about “Unexpected interrupt 0”***

Overview

1. Project One motivation
2. Mundane details (x86/IA-32 version)
   PICs, hardware interrupts, software interrupts and exceptions, the IDT, privilege levels, segmentation
3. Writing a device driver
4. Installing and using Simics
5. Project 1 pieces

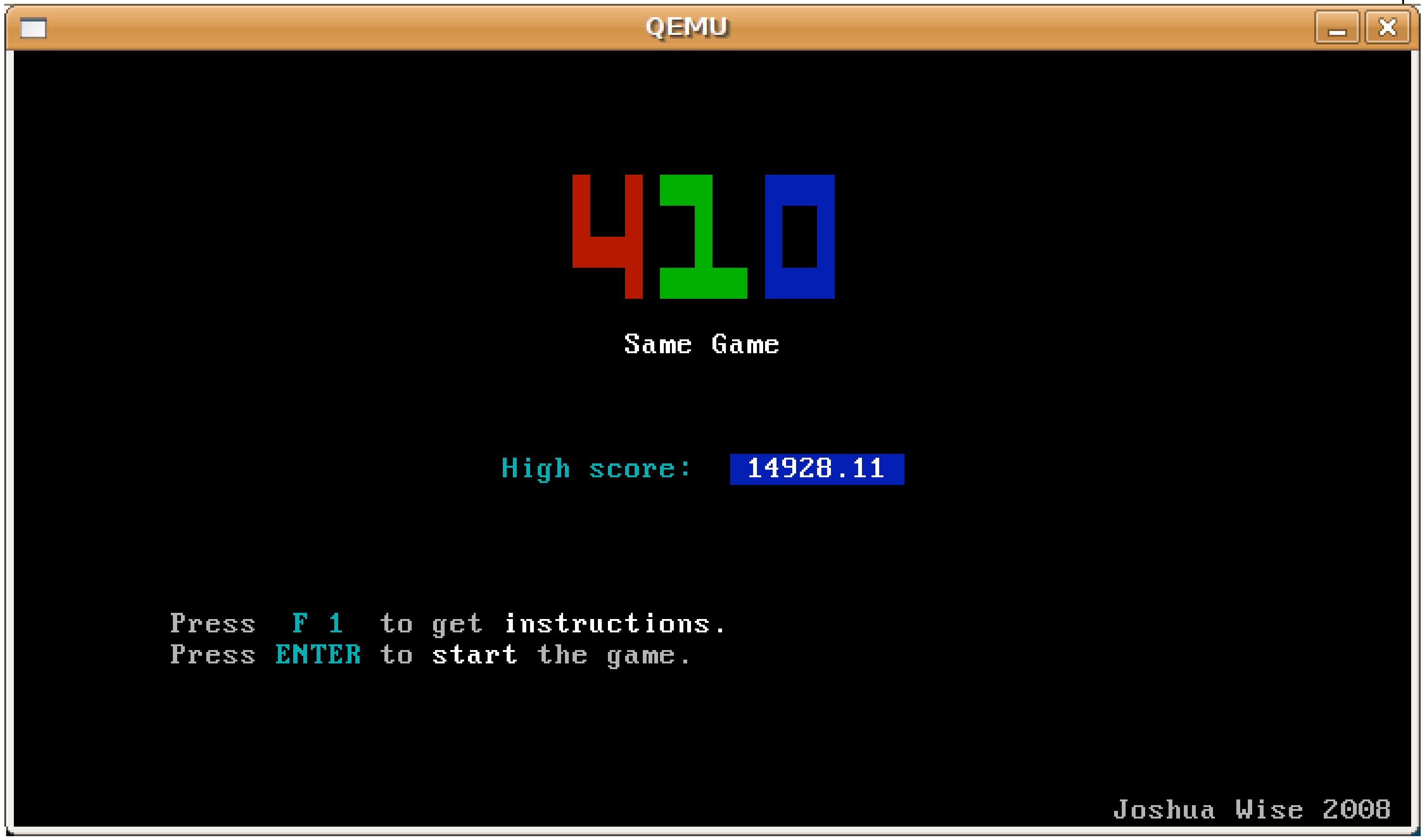
Project 1 Motivation

1. Project 1 implements a game that runs directly on x86 hardware (no OS).
2. 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

Why do you care?

1. You'll need this for Project 3
2. Lots of programs run on bare hardware

Same Game



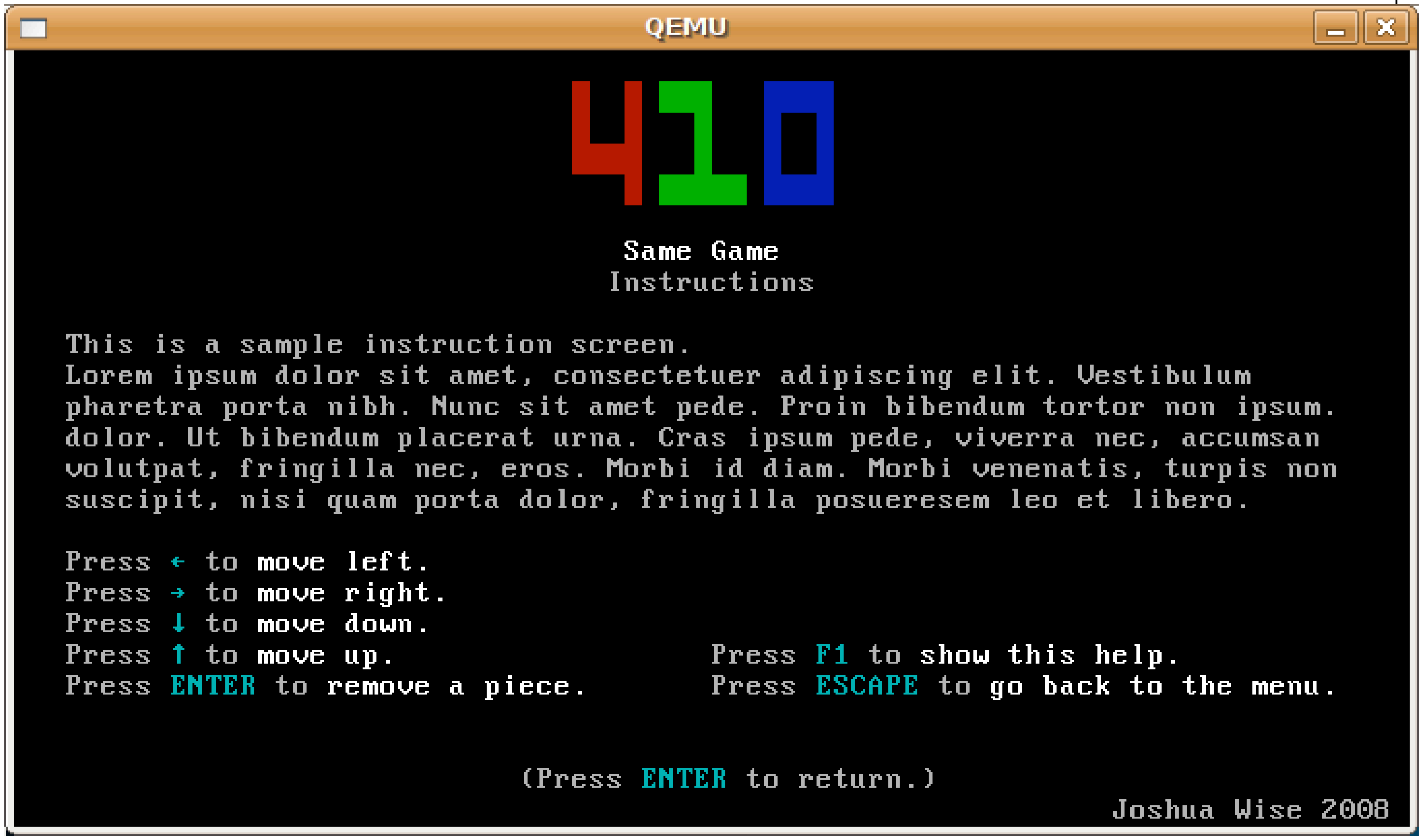
```
Same Game

High score:  14928.11

Press  F 1  to get instructions.
Press ENTER to start the game.

Joshua Wise 2008
```

Same Game



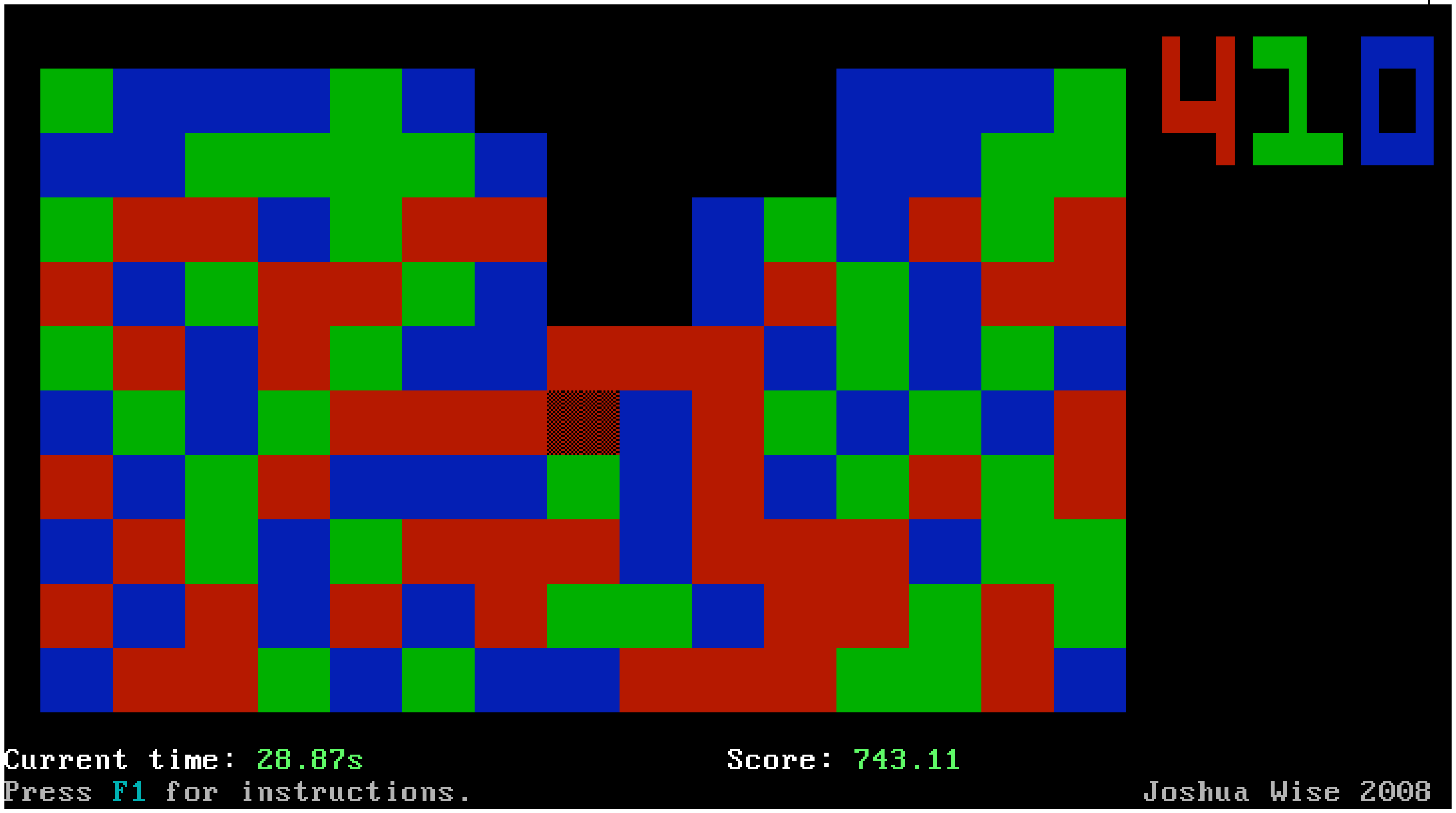
```
                    Same Game
                   Instructions

This is a sample instruction screen.
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Vestibulum
pharetra porta nibh. Nunc sit amet pede. Proin bibendum tortor non ipsum.
dolor. Ut bibendum placerat urna. Cras ipsum pede, viverra nec, accumsan
volutpat, fringilla nec, eros. Morbi id diam. Morbi venenatis, turpis non
suscipit, nisi quam porta dolor, fringilla posueresem leo et libero.

Press ← to move left.
Press → to move right.
Press ↓ to move down.
Press ↑ to move up.                  Press F1 to show this help.
Press ENTER to remove a piece.       Press ESCAPE to go back to the menu.


                (Press ENTER to return.)
                                                         Joshua Wise 2008
```

Same Game



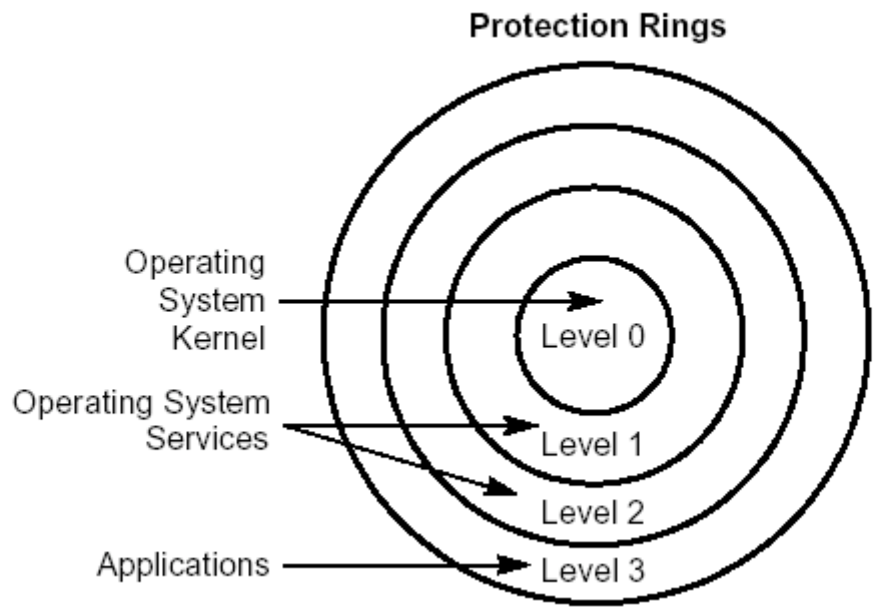
Current time: 28.87s Score: 743.11
Press F1 for instructions. Joshua Wise 2008

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. Nearly 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 “kinds”
   a) Read-only memory (for booting)
   b) Video memory (painted onto screen)
   c) ...
3. Software “kinds”
   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 multiple segments

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) ...

x86 Segmentation Road Map

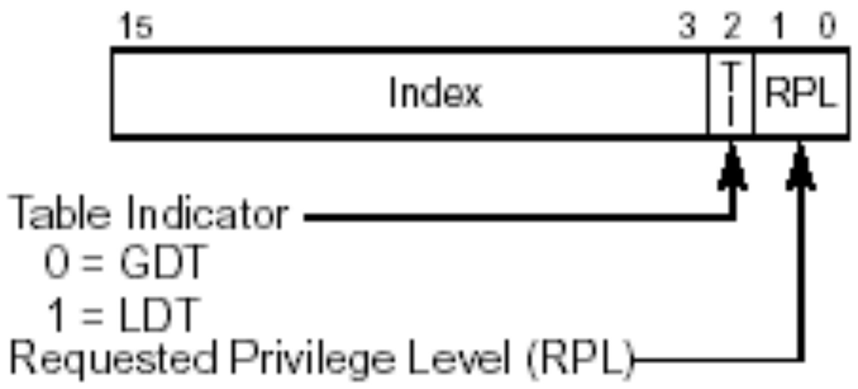
1. Segment = range of “same kind of memory”
2. Segment ***register*** = %CS, %SS, %DS, ... %GS
3. Segment ***selector*** = contents of a segment register
   a) Which segment table and index do we mean?
   b) What access privilege do we have to the segment?
4. Segment ***descriptor*** = definition of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

Memory 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".

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 the %CS ***segment register***
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 builds GDT for you (whew!)
5. You will still need to understand this, though...

Mundane Details in x86: Segmentation

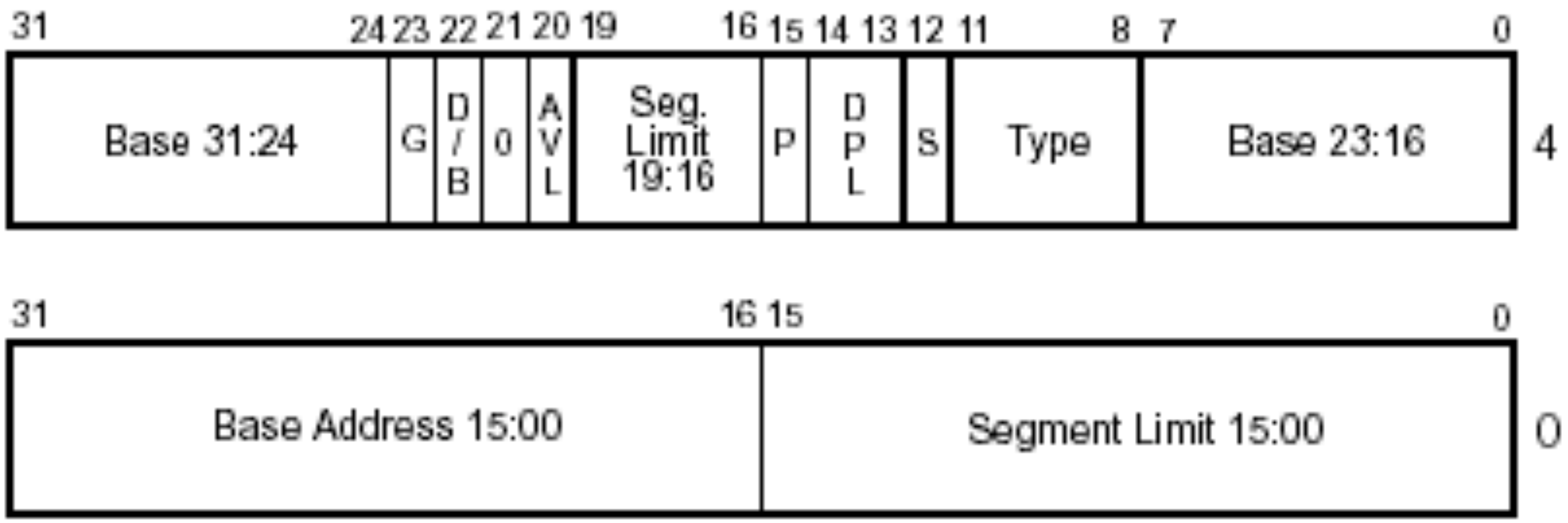
1. Segment selector has a segment number, table selector, and requested privilege level (RPL)
2. Table selector (done)
3. Segment number/index (done)
4. RPL ***generally*** means "what access do I have?"
5. Magic special case: RPL in %CS
   a) Defines ***current processor privilege level***
   b) Think: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

Mundane Details in x86: Segment Descriptors

1. Segments = area of memory with particular access/usage constraints
2. Base, size, “stuff”
3. Logically, base and size are two 32-bit numbers, “stuff” is flag/control bits

Mundane Details in x86: Segment Descriptors

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



Mundane Details in x86: Segmentation

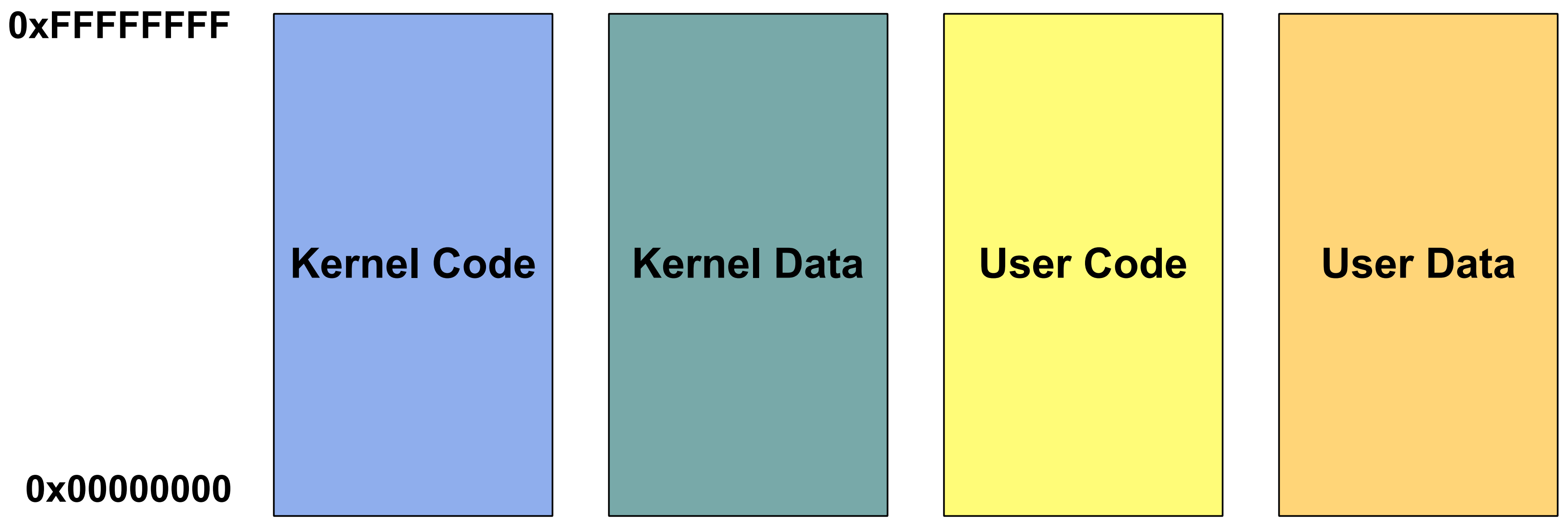
1. Consider %CS segment register's segment selector's segment descriptor
   a) Assume base = 0xcafe0000
   b) Assume limit > 47806
2. Assume %EIP contains 0xbabe
   a) Then %CS:%EIP means “linear virtual address” 0xcafebabe (0xcafe0000 + 0x0000babe)
3. “Linear virtual address” fed to virtual memory hardware, if it's turned on (Project 3, not Project 1)

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
   All instruction fetches are from %CS:%EIP
4. SS is the segment register for the stack segment
   PUSH, POP instructions use %SS:%ESP
5. DS is the default segment register for data access
   `MOVL (%EAX),%EBX` fetches from %DS:%EAX
   But ES, FS, and GS can be specified instead

Mundane Details in x86: Segmentation

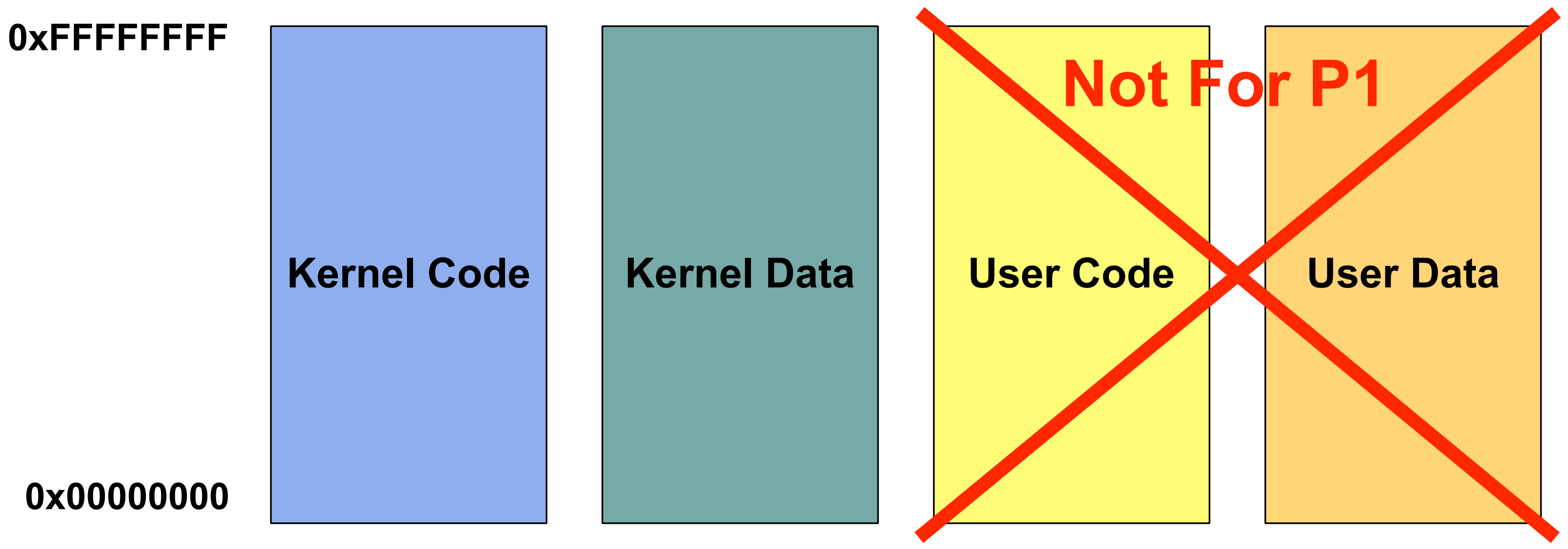
1. Segments need not be backed by physical memory and can overlap
2. Segments defined for 15-410:



Mundane Details in x86: Segmentation

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

Mundane Details in x86: Segmentation



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:

   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

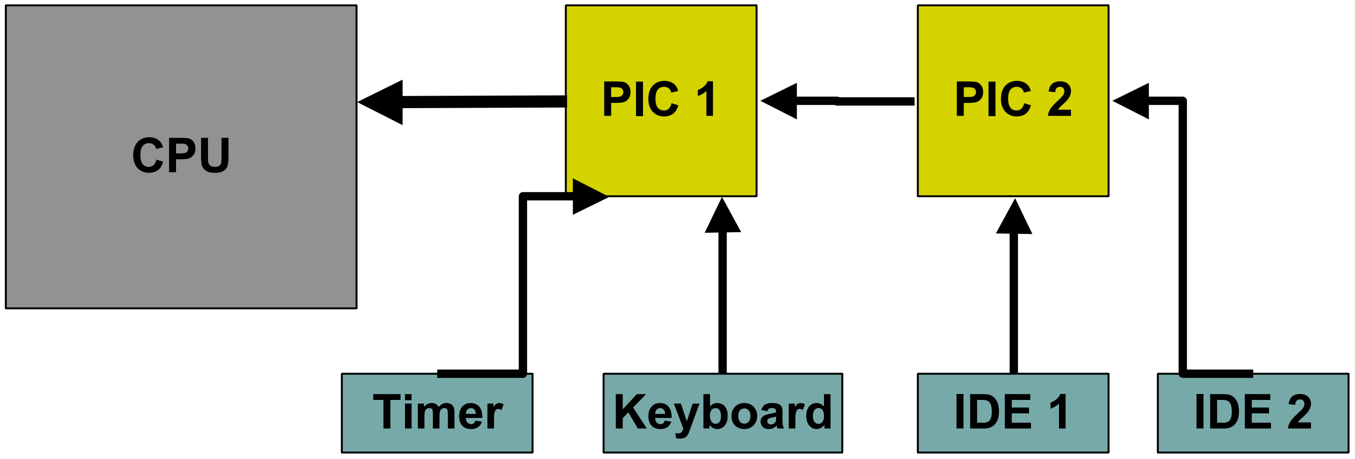
1. Sometimes user processes do stupid things
2. int gorgonzola = 128/0;
3. char* idiot_ptr = NULL; *idiot_ptr = 0;
4. These exceptions cause a handler routine to be executed
5. Examples include divide by zero, general protection fault, page fault

Mundane Details in x86: “Software Interrupts”

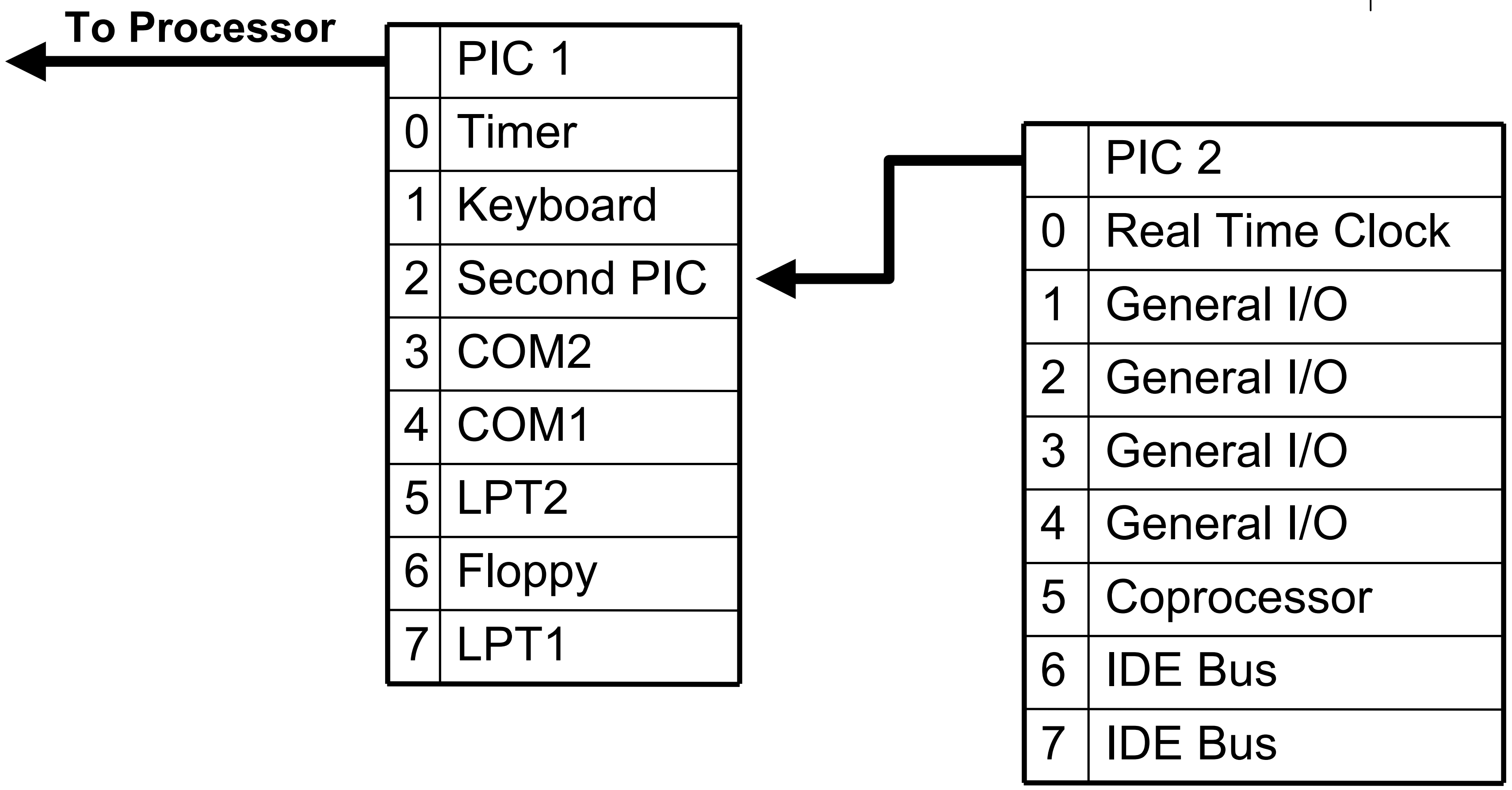
1. A device gets the kernel’s attention by raising a (hardware) interrupt
2. User processes get the kernel’s attention by raising a “software interrupt”
   a) Which is not an interrupt even if Intel calls it one!
3. x86 instruction **`INT n`**
   (more info on page 346 of intel-isr.pdf)
4. Invokes handler 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

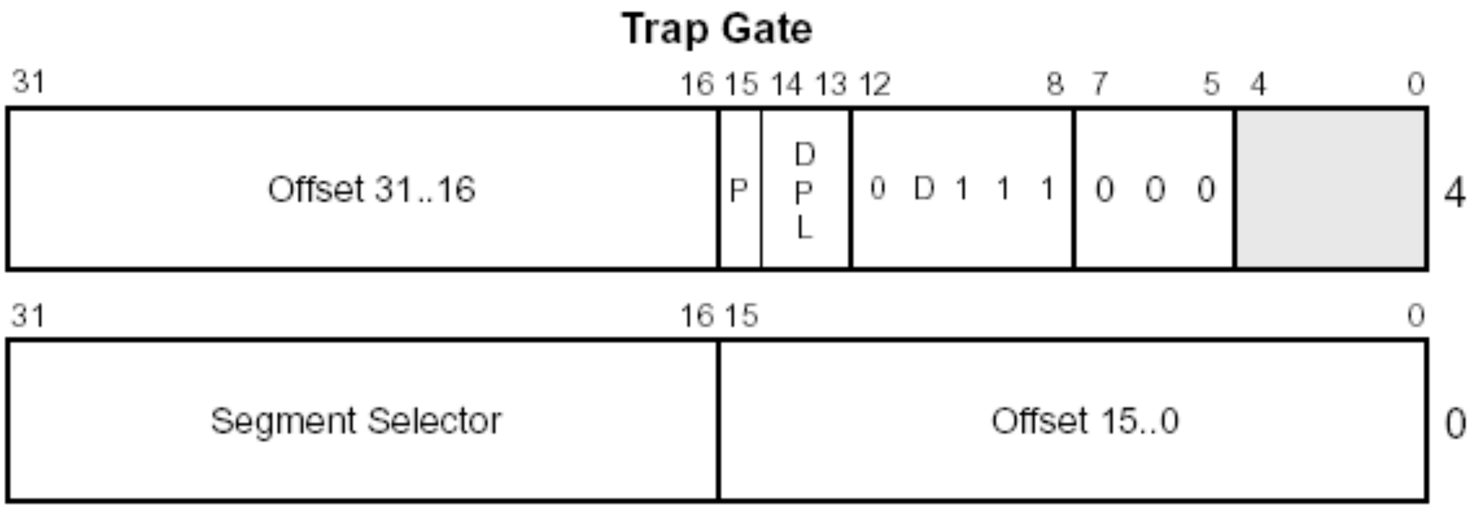
	PIC 2
0	Real Time Clock
1	General I/O
2	General I/O
3	General I/O
4	General I/O
5	Coprocessor
6	IDE Bus
7	IDE Bus

Interrupt Descriptor Table – IDT

1. Processor needs info on which handler to run when
2. Processor reads appropriate IDT entry depending on the interrupt, exception *or* **`INT n`** instruction
3. Logically, an IDT entry contains a function pointer and some flags

Interrupt Descriptor Table – IDT

1. Processor needs info on which 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:



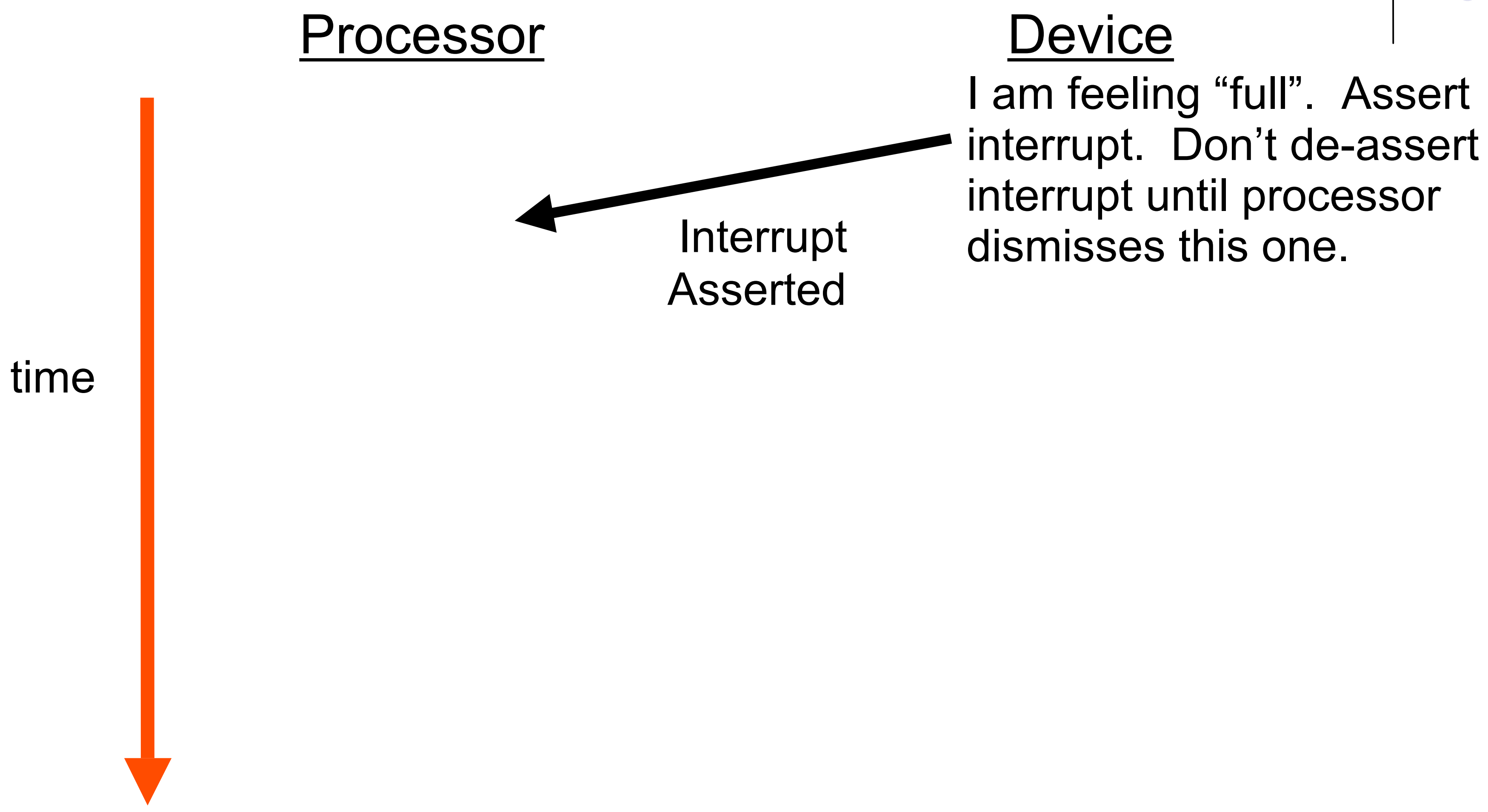
Interrupt Descriptor Table (IDT)

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 interrupt

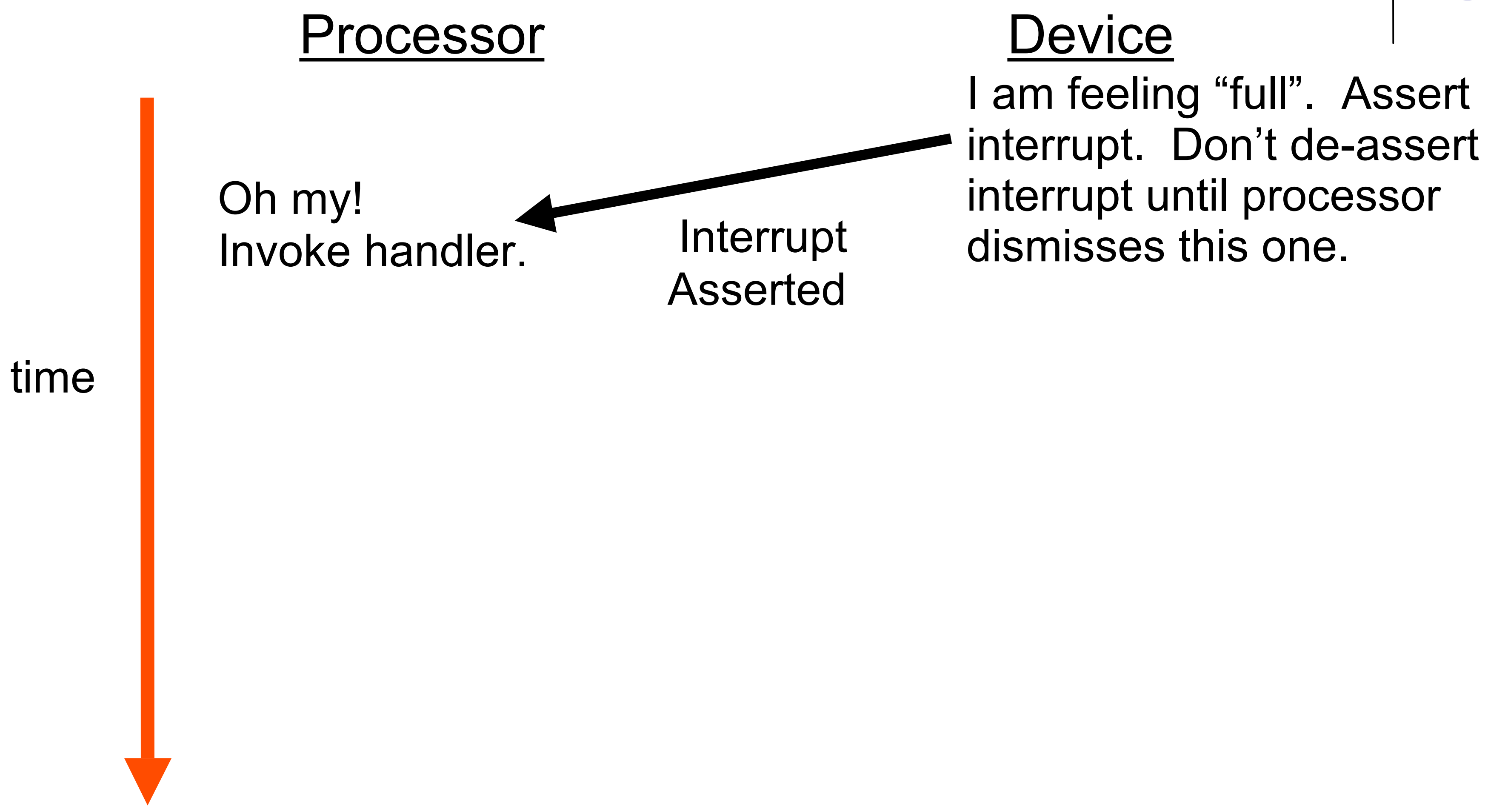
More information in section 5.12 of intel-sys.pdf.

Typical Interrupt Handshake



I am feeling “full”. Assert interrupt. Don’t de-assert interrupt until processor dismisses this one.

Typical Interrupt Handshake



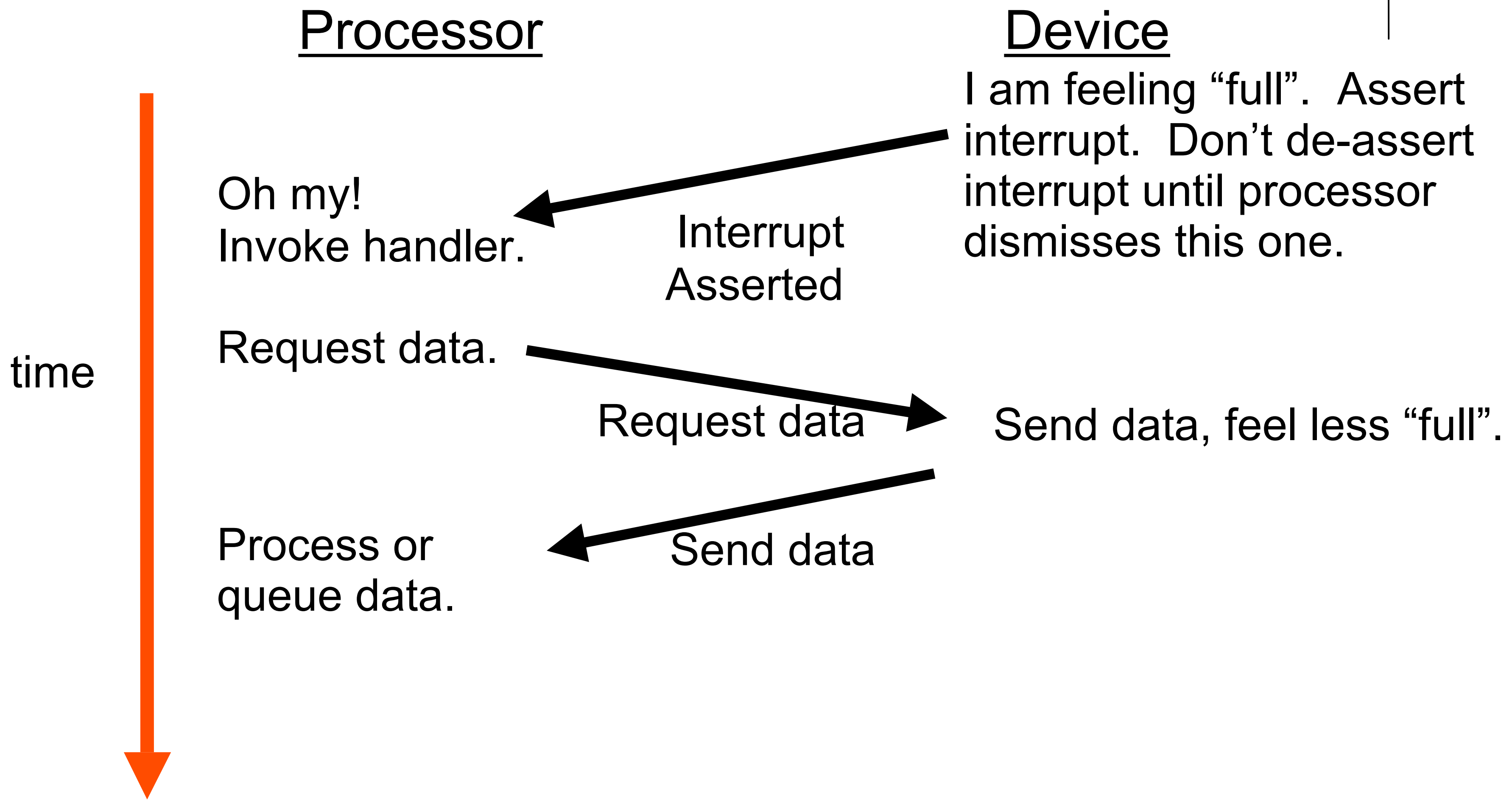
Processor

Device

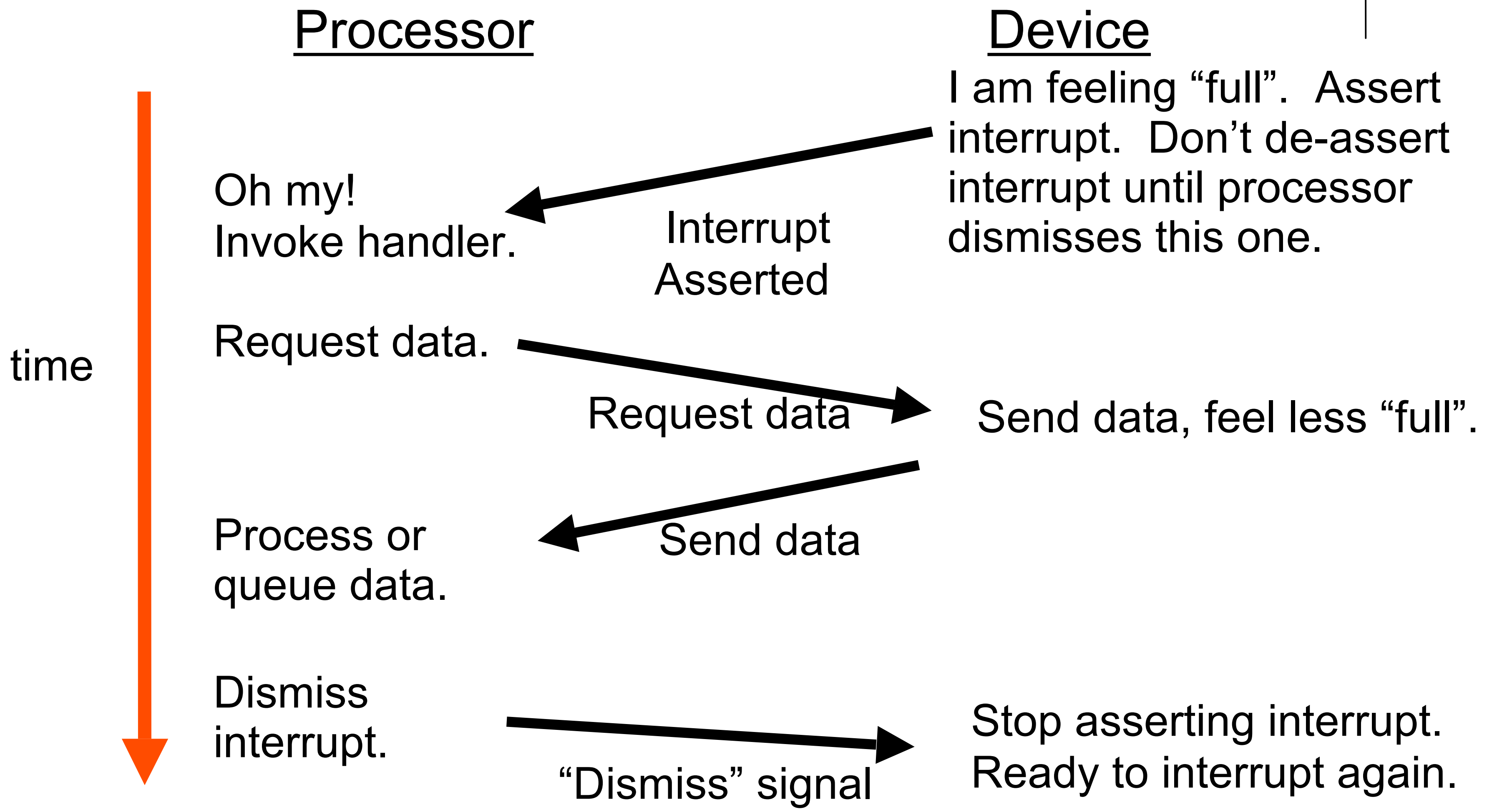
I am feeling “full”. Assert interrupt. Don’t de-assert interrupt until processor dismisses this one.

Oh my!
Invoke handler.

Typical Interrupt Handshake


Processor
Device
I am feeling "full". Assert interrupt. Don't de-assert interrupt until processor dismisses this one.
Oh my! Invoke handler.
Interrupt Asserted
Request data.
time
Request data
Send data, feel less "full".
Process or queue data.
Send data

Typical Interrupt Handshake



Processor

Oh my!
Invoke handler.

Request data.

Process or
queue data.

Dismiss
interrupt.

Device

I am feeling “full”. Assert interrupt. Don’t de-assert interrupt until processor dismisses this one.

Send data, feel less “full”.

Stop asserting interrupt.
Ready to interrupt again.

Enabling / Disabling Interrupts

1. PIC automatically disables interrupts from a device until one dismissed by processor.
2. We also provide **disable_interrupts()**, which disables interrupts from ALL devices. Think of this as deferring interrupts. They are still out there, waiting to happen.
3. We provide **enable_interrupts()**, which re-enables interrupts.
4. Finer-grain control is also possible.

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. PC video hardware uses ***both***
   a) ***Cursor is controlled by I/O ports***
   b) ***Characters are 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 device 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. This is not intuitive (for software people).
   a) Why can't we just "`*R12 = 0x00000032`"?
3. But you can't get anywhere on P1 without understanding it.

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. Project 1 Makefile builds floppy-disk images
4. Simics boots and runs them
   a) Launch simics-linux.sh in your build directory
5. Your 15-410 AFS space has p1/, scratch/
6. If you work in scratch/, we can read your files, and 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)

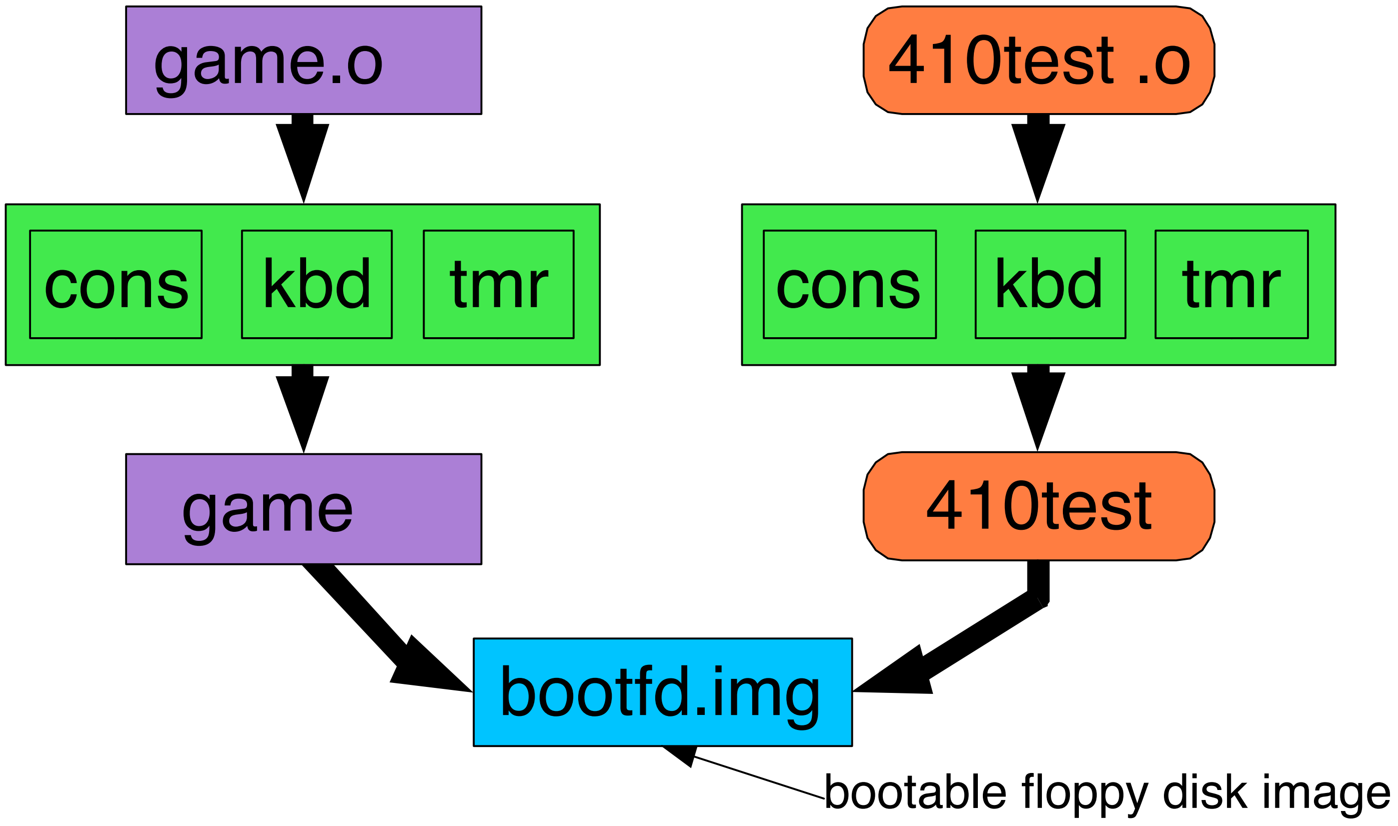
Simics vs. gdb

1. Similar jobs: symbolic debugging
2. Random differences
   a) Details of commands and syntax
3. Notable differences
   a) Simics knows ***everything*** about PC hardware – all magic registers, TLB contents, interrupt masks, etc.
   b) Simics is scriptable in Python

Project 1 Pieces

1. You will build
   a) A device-driver library
      - "console" (screen) driver
      - keyboard driver
      - timer driver
   b) A simple game application using your driver library
2. We will provide
   a) underlying setup/utility code
   b) A simple device-driver test program

Project 1 Pieces



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) Simics is not (exactly) gdb
   c) Invest time to learn more than bare minimum

Summary

3. Project 1 runs on bare *PC* hardware
   a) As hardware goes, it's pretty irrational
   b) *Almost nothing* works “how you would expect”
   c) Those pesky bit-field diagrams do matter
   d) Getting started is tough, so please don't delay.
4. This isn't throwaway code
   a) We will read it
   b) You will use it for Project 3

   So spend extra time to make it really great code