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"***

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

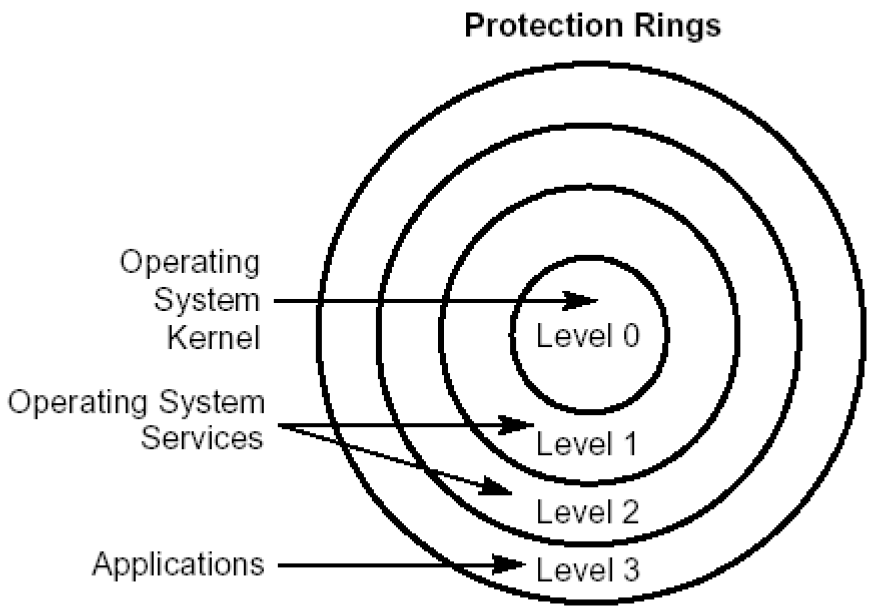
1. What are our hopes for project 1?
   a) introduction to kernel programming
   b) a better understanding of the x86 arch
   c) hands-on experience with hardware interrupts and device drivers
   d) get acquainted with the simulator (Simics) and development tools

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

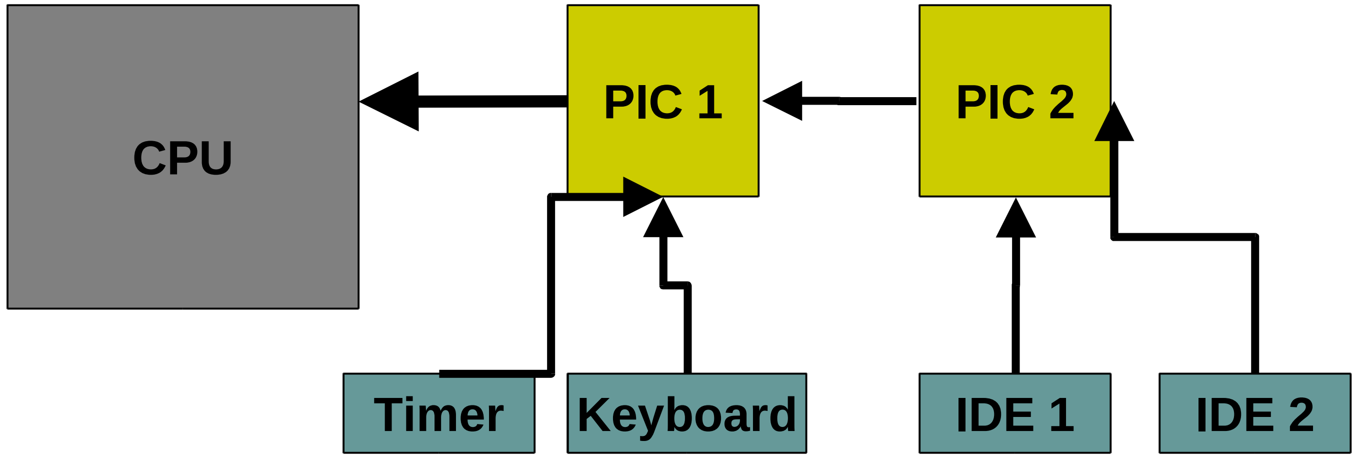


Mundane Details in x86: Privilege Levels

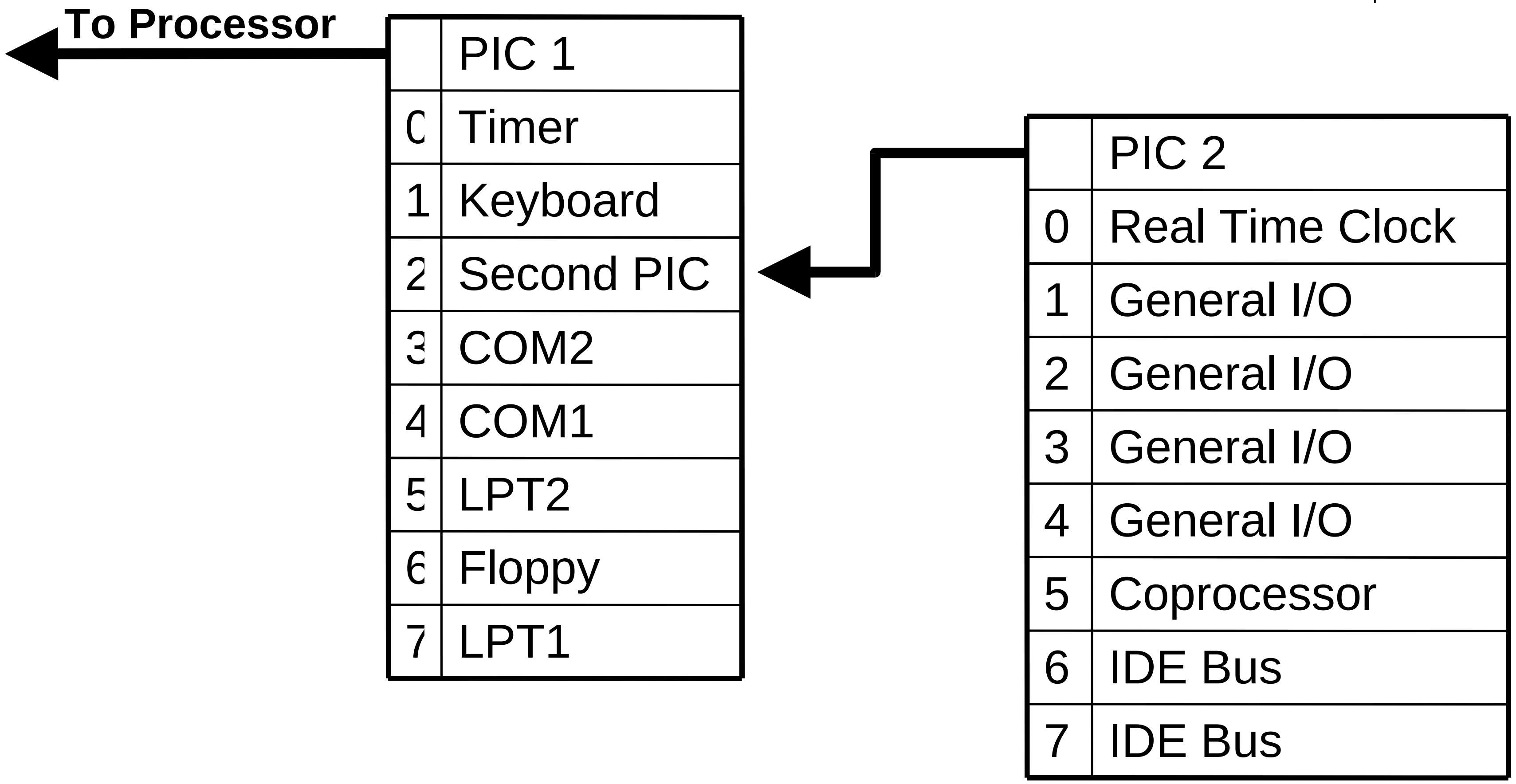
1. Essentially unused in Project 1
2. Projects 2 through 4
   a) PL0 is “kernel”
   b) PL3 is “user”
   c) Interrupts & exceptions usually transfer from 3 to 0
   d) Running user code means getting from 0 to 3

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

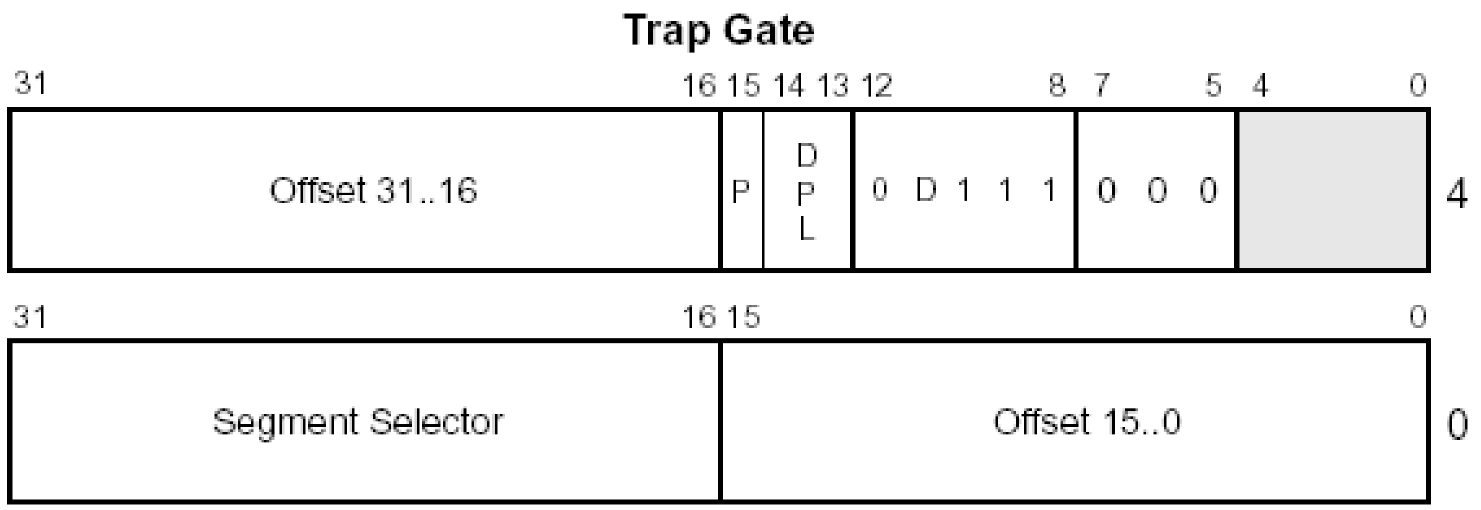


	PIC 1
0	Timer
1	Keyboard
2	Second PIC
3	COM2
4	COM1
5	LPT2
6	Floppy
7	LPT1

	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. An entry in the IDT looks like this:



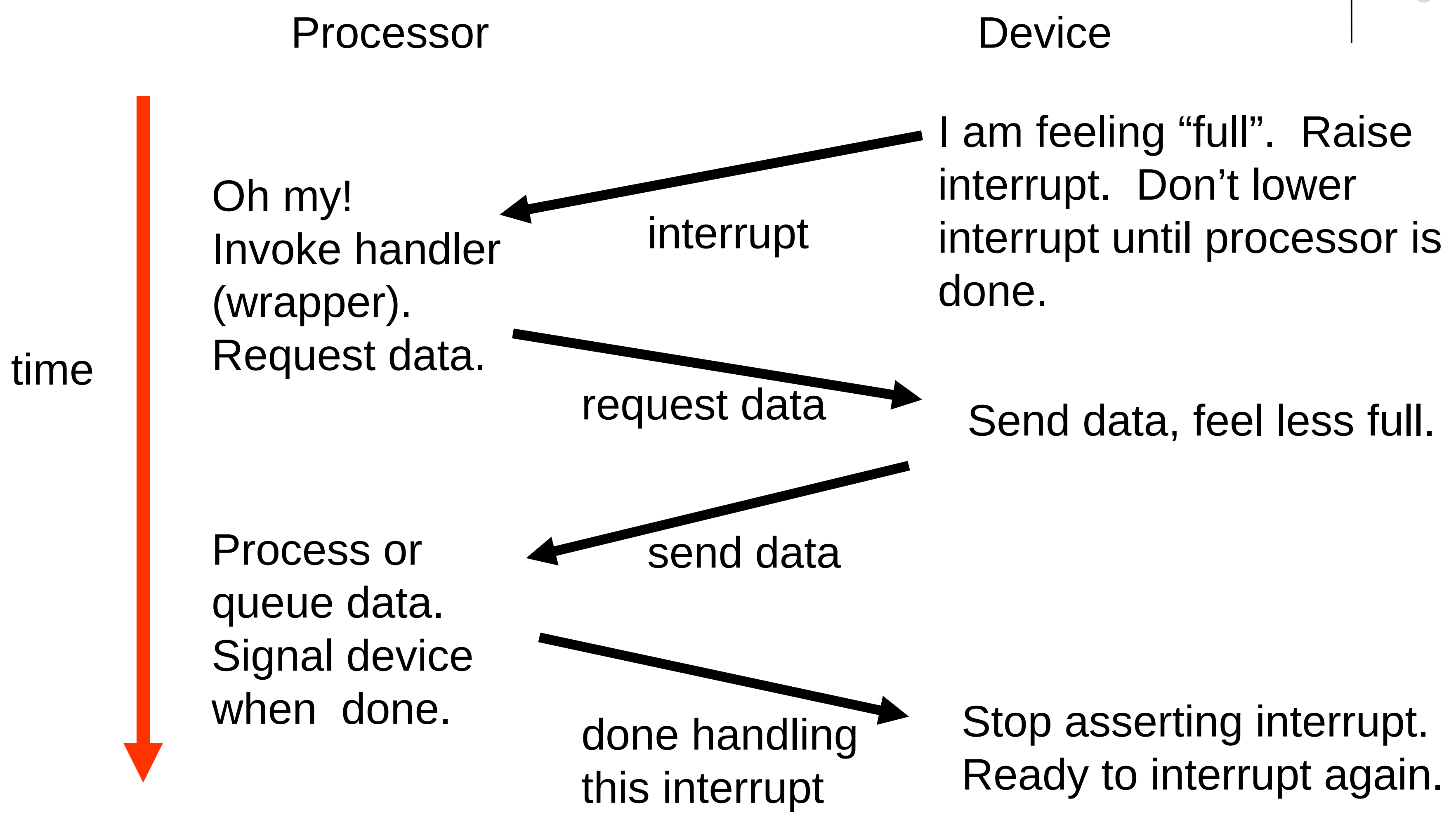
Interrupt Descriptor Table (IDT)

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

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

Typical Interrupt Handshake



Processor

Device

time

I am feeling “full”. Raise interrupt. Don’t lower interrupt until processor is done.

interrupt

Oh my!
Invoke handler (wrapper).
Request data.

request data

Send data, feel less full.

send data

Process or queue data.
Signal device when done.

done handling this interrupt

Stop asserting interrupt.
Ready to interrupt again.

Enabling / Disabling Interrupts

1. PIC automatically disables interrupts from same device until acknowledged 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 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).***
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. Runs on both x86 Linux and Solaris
   a) We haven't run it on Solaris in “a while”
   b) Cluster PCs ran faster than cluster SunBlades
   c) Cluster SunBlades then became gone

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 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

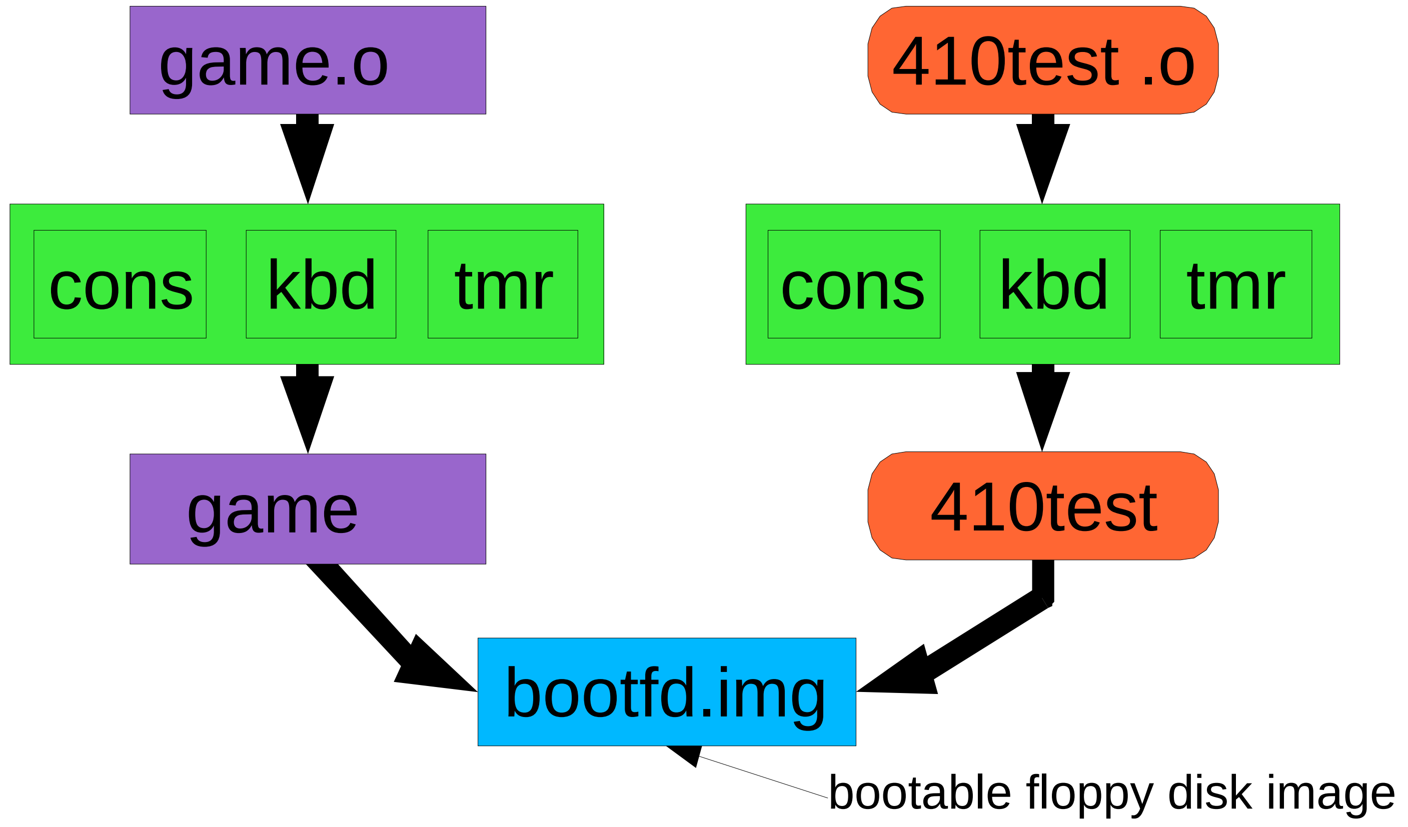
1. Similar jobs: symbolic debugging
2. Random differences
   a) Details of commands and syntax
3. Notable differences
   a) Simics knows ***everything*** about PC hardware – all magic registers, TLB contents, interrupt masks, etc.
   b) Simics is scriptable in Python

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

1. ***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.***
2. ***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