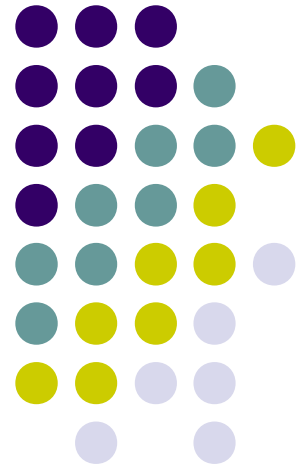


What You Need to Know for Project One

Dave Eckhardt
Caroline Buckey & Alex Crichton
Joshua Wise
Joey Echeverria
Steve Muckle

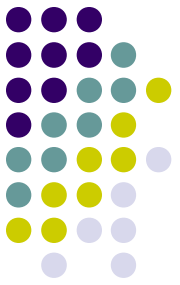


Synchronization



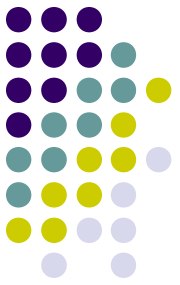
- Please read the syllabus
 - Some of your questions are answered there :-)
 - We would rather teach than tear our hair out
- Also, please read the Project 1 handout
 - Please don't post about “Why did my screen turn purple?”
- Partner registration -- please do if you can!
 - Status: 32 students registered (thanks!)
 - **The annoying e-mail will start soon!**

Overview



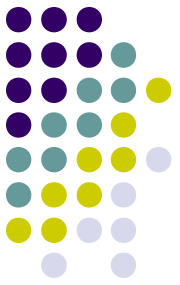
- Project 1 motivation
- Mundane details (x86/IA-32 version)
PICs, hardware interrupts, software interrupts and exceptions, the IDT, privilege levels, segmentation
- Writing a device driver
- Installing and using Simics
- Project 1 pieces

Project 1 Motivation

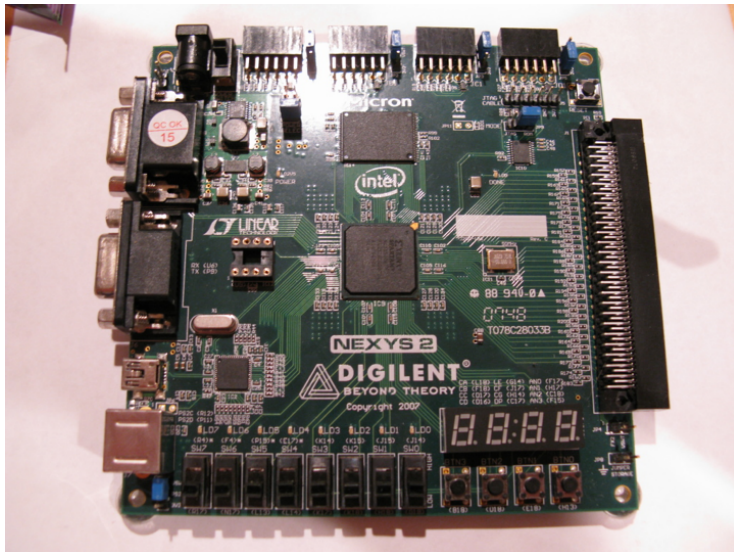


- Project 1 implements a game that runs directly on x86 hardware (no OS)
- What are our hopes for Project 1?
 - introduction to kernel programming
 - a better understanding of the x86 arch
 - hands-on experience with hardware interrupts and device drivers
 - get acquainted with the simulator (Simics) and development tools

Why do you care?

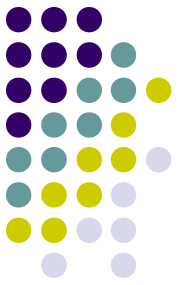


- You'll need this for Project 3
- Lots of programs run on bare hardware



Copyright 2008 HI-TECH Software

Mundane Details in x86

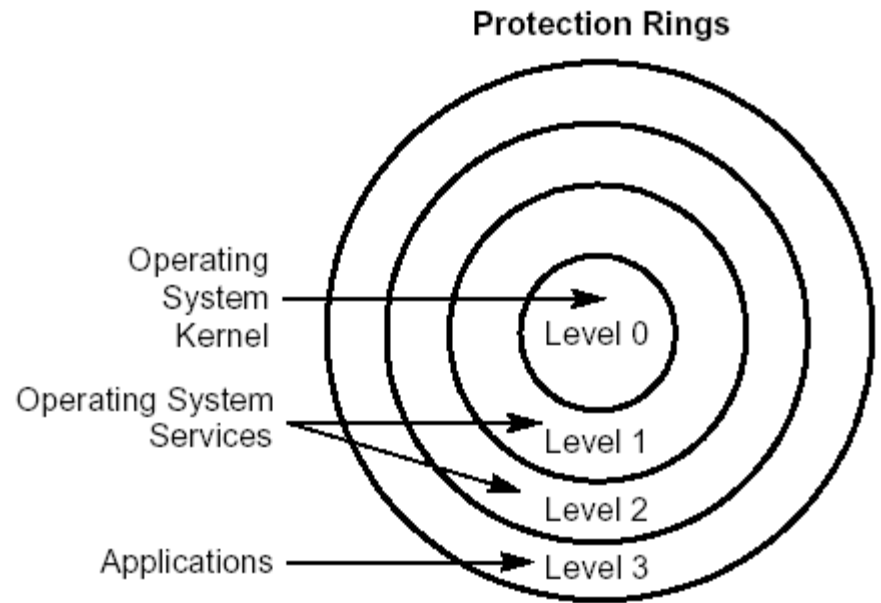


- Kernels work closely with hardware
- This means you need to know about hardware
- Some knowledge (registers, stack conventions) is assumed from 15-213
- You will learn more x86 details as the semester goes on
- Use the Intel PDF files as reference (<http://www.cs.cmu.edu/~410/projects.html>)

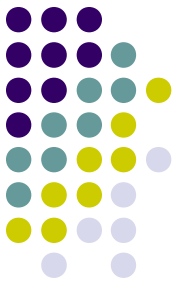
Mundane Details in x86: Privilege Levels



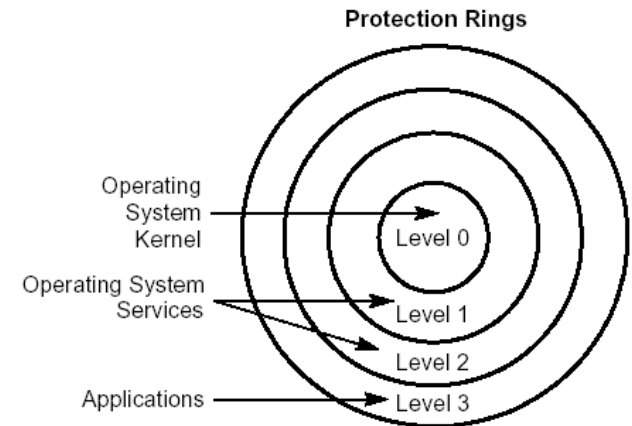
- Processor has 4 “privilege levels” (PLs)
- Zero most-privileged, three least-privileged
- Processor executes at one of the four PLs at any given time
- PLs protect privileged data, cause general protection faults



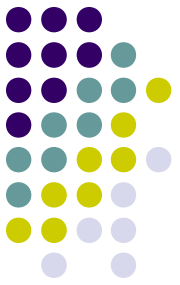
Mundane Details in x86: Privilege Levels



- Nearly unused in Project 1
- For projects 2 through 4
 - PL0 is “kernel”
 - PL3 is “user”
 - Interrupts & exceptions usually transfer from 3 to 0
 - Sometimes: from 0 to 0
 - Running user code means getting from 0 to 3



Memory Segmentation



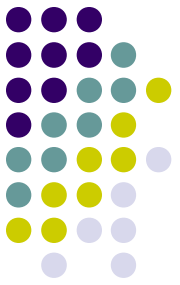
- There are different “kinds” of memory
- Hardware “kinds”
 - Read-only memory (for booting)
 - Video memory (painted onto screen)
 - ...
- Software “kinds”
 - Read-only memory (typically, program code)
 - Stack (grows down), heap (grows up)
 - ...

Memory Segmentation



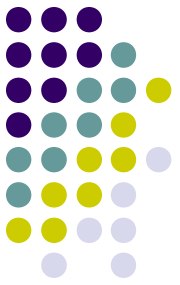
- Memory segment is a range of “the same kind”
- Hardware “kind”
 - Mark video memory as “don't buffer writes”
- Software “kind”
 - Mark all code pages read-only
- Fancy software
 - Process uses many separate segments
 - Windows: each DLL is multiple segments
 - (Well, Win16... and Win32... but not Win64...)

Memory Segmentation



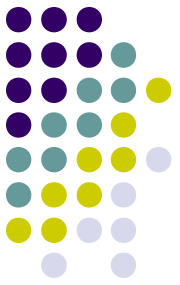
- x86 hardware *loves* segments
- Mandatory segments
 - Stack
 - Code
 - Data
- Segments interact with privilege levels
 - Kernel stack / user stack
 - Kernel code / user code
 - ...

x86 Segmentation Road Map



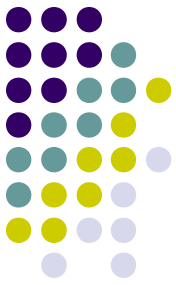
- Segment = range of “same kind of memory”
- Segment **register** = %CS, %SS, %DS, ... %GS
- Segment **selector** = contents of a segment register
 - Which segment table and index do we mean?
 - What access privilege do we have to the segment?
- Segment **descriptor** = definition of segment
 - Which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

Memory Segmentation

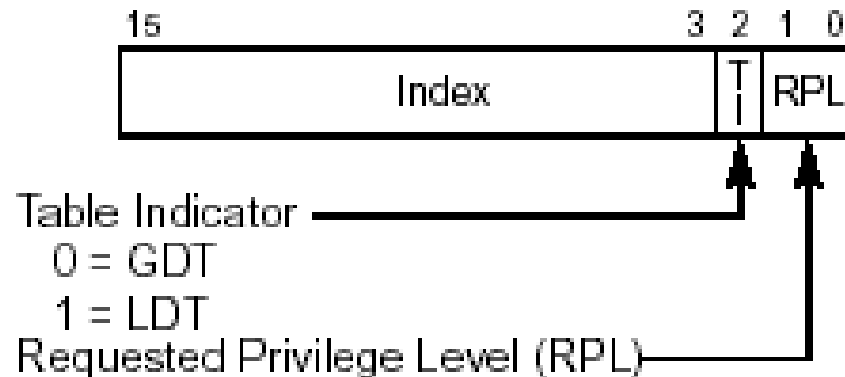


- When fetching an instruction, the processor asks for an address that looks like this: `%CS:%EIP`
- So, if `%EIP` is `0xbabe` then `%CS:%EIP` is the 47806th byte of the “code segment”.

Mundane Details in x86: Segmentation



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

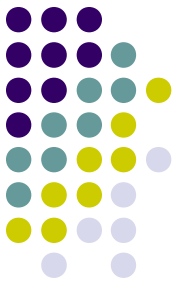


Mundane Details in x86: Segmentation



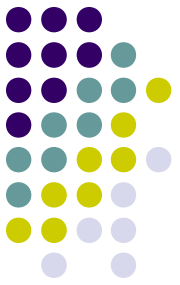
- Segment selector has a segment number, table selector, and requested privilege level (RPL)
- The table-select flag selects a descriptor table
 - *global descriptor table or local descriptor table*
- Segment number indexes into that descriptor table
 - 15-410 uses only global descriptor table (whew!)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

Mundane Details in x86: Segmentation



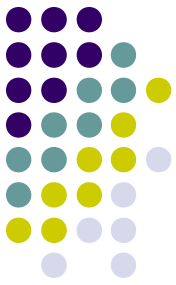
- Segment selector has a segment number, table selector, and requested privilege level (RPL)
- Table selector (done)
- Segment number/index (done)
- RPL *generally* means “what access do I have?”
- Magic special case: RPL in %CS
 - Defines *current processor privilege level*
 - Think: “user mode” vs.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

Mundane Details in x86: Segment Descriptors

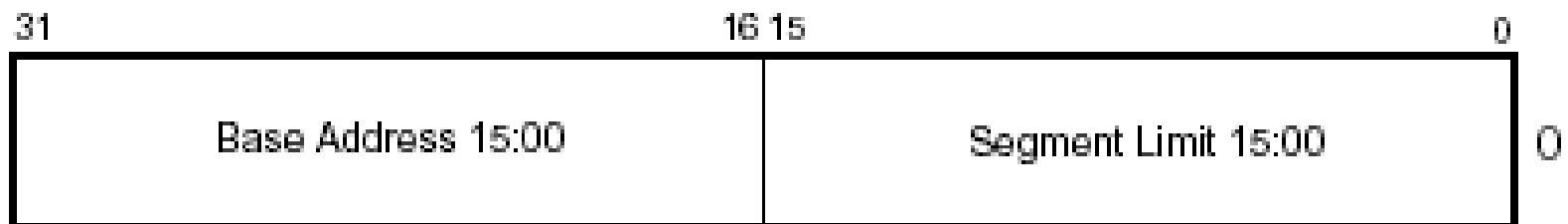
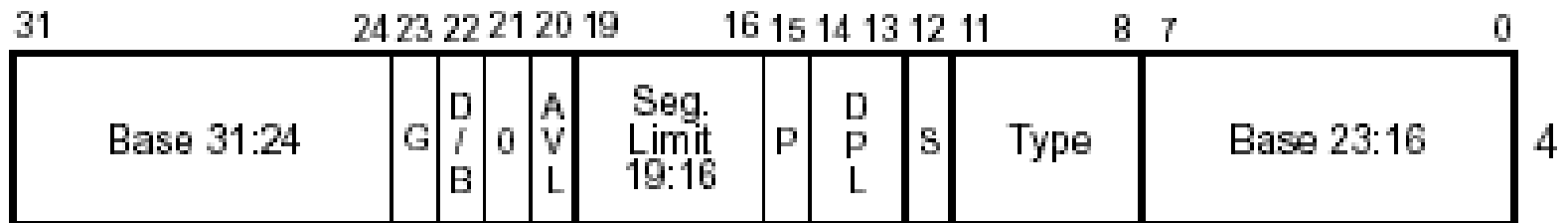


- Segment = area of memory with particular access/usage constraints
- Base, size, “stuff”
- Logically, base and size are two 32-bit numbers, “stuff” is flag/control bits

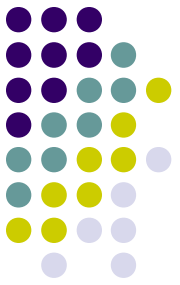
Mundane Details in x86: Segment Descriptors



- Segment = area of memory with particular access/usage constraints
- Base, size, “stuff”
- Layout:

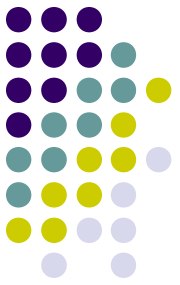


Mundane Details in x86: Segmentation



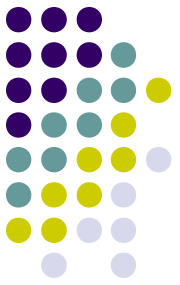
- Consider %CS segment register's segment selector's segment descriptor
 - Assume base = 0xcafe0000
 - Assume limit > 47806
- Assume %EIP contains 0xbabe
 - Then %CS:%EIP means “linear virtual address” 0xcafebabe (0xcafe0000 + 0x0000babe)
- “Linear virtual address” fed to virtual memory hardware, if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

Implied Segment Registers

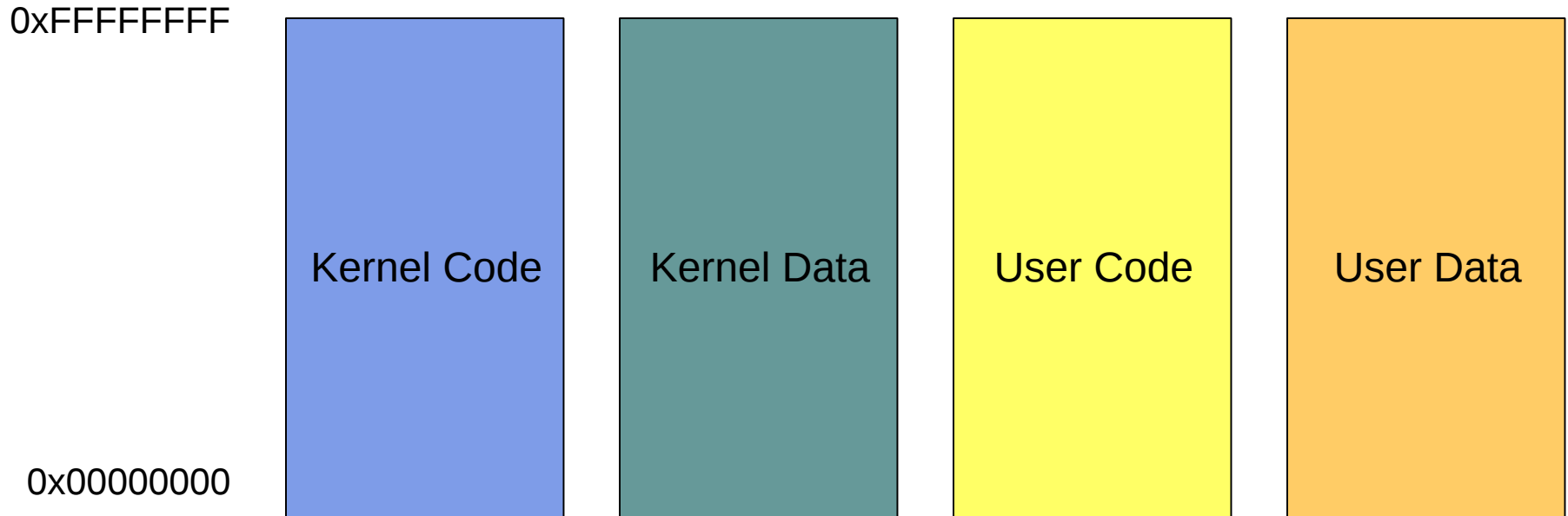


- Programmer doesn't usually *specify* segment
- Usually *implied* by “kind of memory access”
- CS is the segment register for fetching code
 - All instruction fetches are from %CS:%EIP
- SS is the segment register for the stack segment
 - PUSH, POP instructions use %SS:%ESP
- DS is the default segment register for data access
 - `MOVL (%EAX), %EBX` fetches from %DS:%EAX
 - But ES, FS, and GS can be specified instead

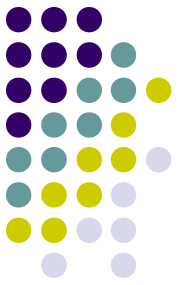
Mundane Details in x86: Segmentation



- Segments need not be fully backed by physical memory, and can overlap
- Segments defined for 15-410:

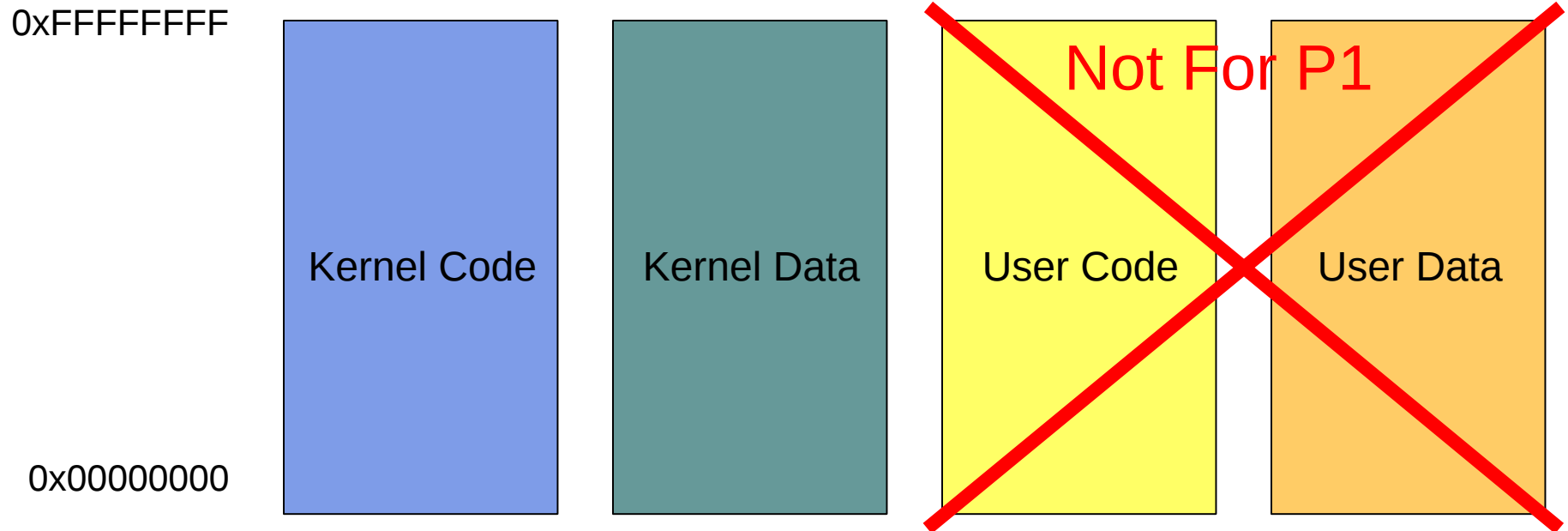


Mundane Details in x86: Segmentation

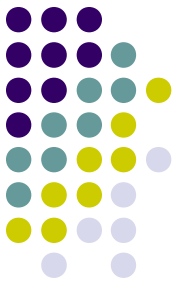


- Why so many?
- You can't specify a segment that is readable, writable *and* executable.
 - Need one for readable/executable code
 - Another for readable/writable data
- Need user and kernel segments in Project 3 for protection
- (Code, Data) X (User, Kernel) = 4

Mundane Details in x86: Segmentation

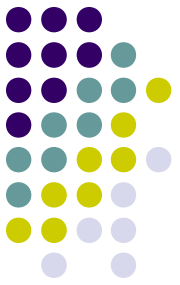


Mundane Details in x86: Segmentation



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

Execution Types



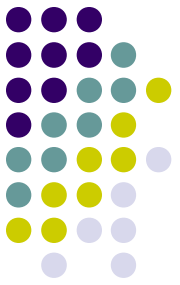
- From the processor's perspective, three kinds of instruction execution
 - Regular work – execute this one, then the next
 - Branch – execute this one, then somewhere else
 - ...?

Execution Types - Surprises



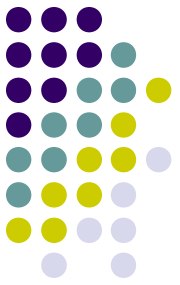
- From the processor's perspective, three kinds of instruction execution
 - Regular work – execute this one, then the next
 - Branch – execute this one, then somewhere else
 - “Surprise” – suddenly we must run a different body of code!
- Surprises
 - Exception/fault – this instruction can't be executed
 - Trap – voluntary transfer to different code
 - Interrupt – involuntary, unpredictable transfer to different code

Surprises



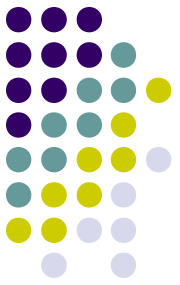
- Exception: a *particular instruction* broke
 - SIGSEGV, page fault, zero divide, illegal instruction
 - We may fix the conditions and re-run the instruction
 - We may kill the program
- Trap: a *particular instruction* asks for help
 - System call: “please invoke the kernel to ...”
 - We later resume at the instruction after the trap
- Interrupt: an I/O device needs attention
 - A *random instruction* is deferred while we run driver
 - We later resume the deferred instruction

Mundane Details in x86: Faults



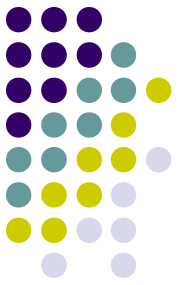
- Sometimes code does stupid things
 - `int gorgonzola = 128/0;`
 - `char* idiot_ptr = NULL; *idiot_ptr = 0;`
 - Executing bytes which don't encode an instruction
- Exceptions cause a handler routine to be run
 - Record information about which instruction broke
 - Record information about why it broke
 - Locate “exception handler”
 - Exception handler decides: fix/kill/crash

Mundane Details in x86: “Software Interrupts”

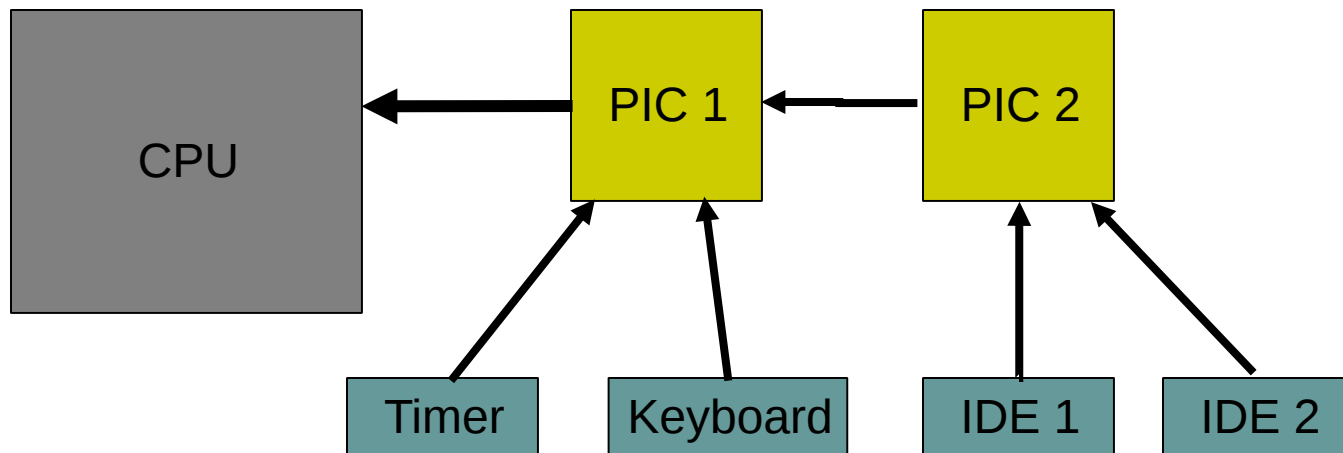


- A device gets the kernel’s attention by raising a (hardware) interrupt
- User processes get the kernel’s attention by raising a “software interrupt”
 - Which is not an interrupt even if Intel calls it one!
- x86 instruction `INT n`
(more info on page 346 of [intel-isr.pdf](#))
- Invokes handler routine: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

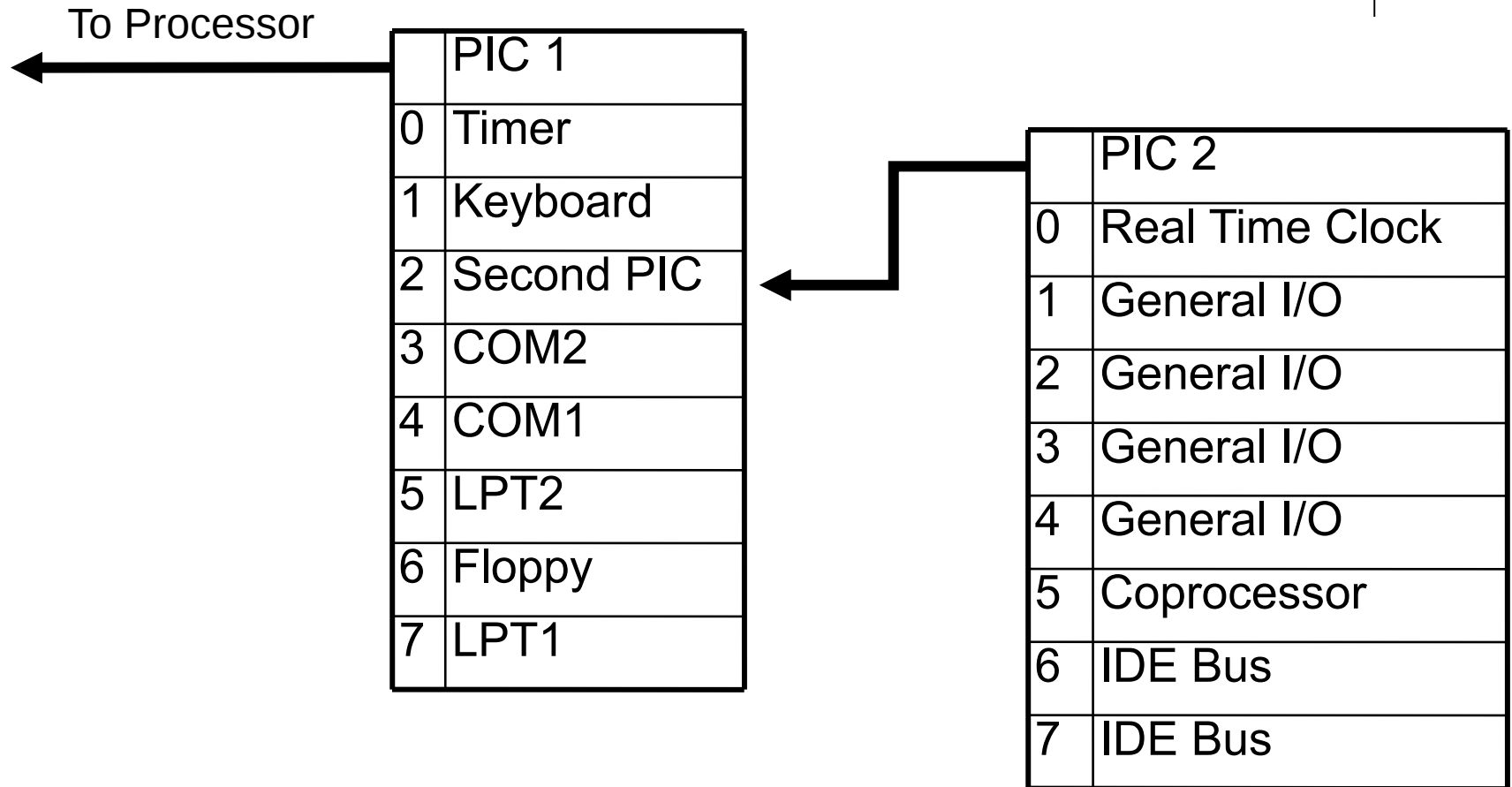
Mundane Details in x86: Interrupts and the PIC



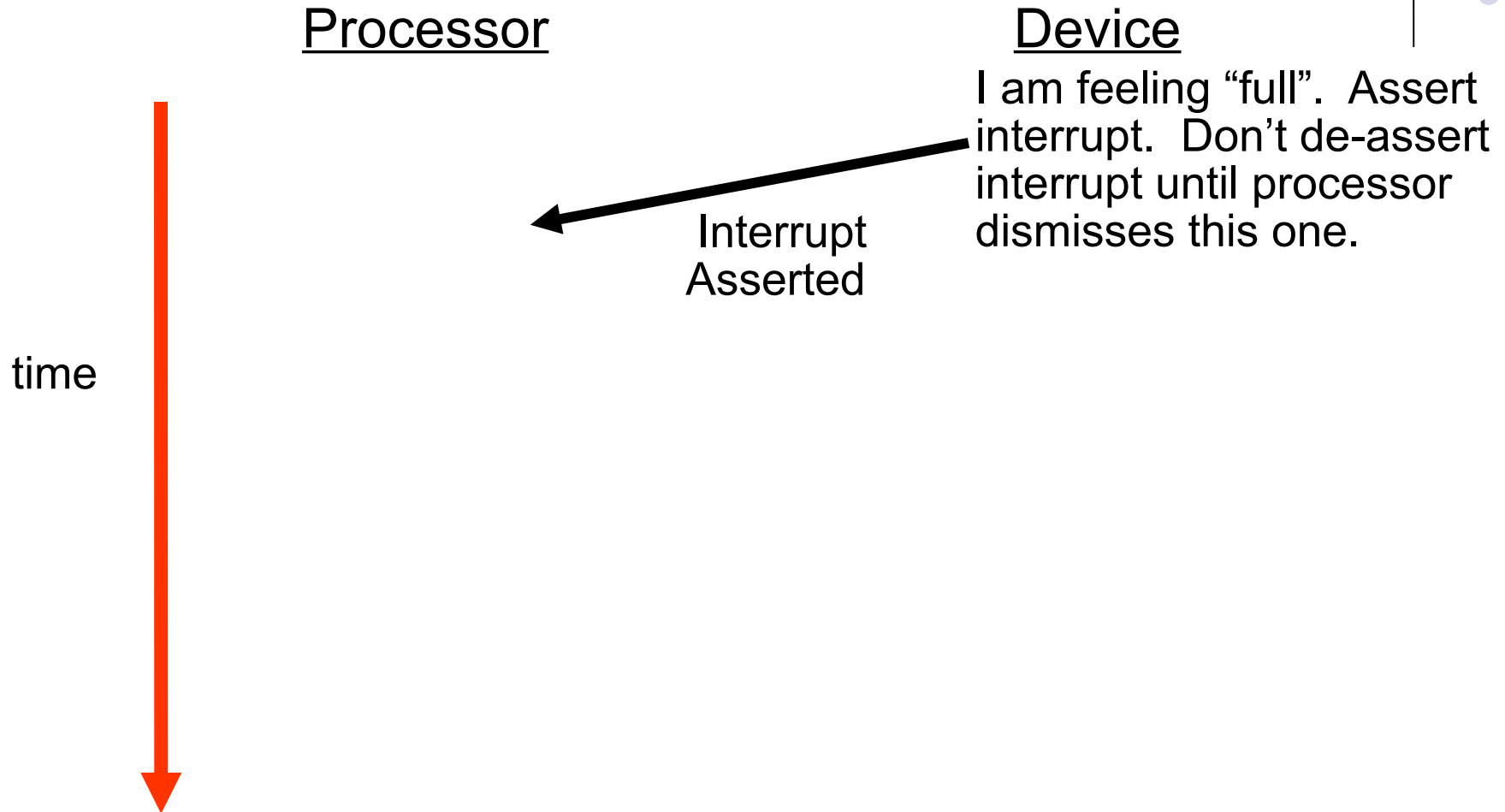
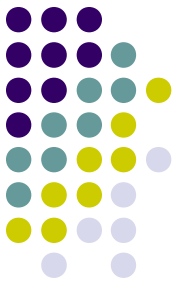
- Devices raise interrupts through the Programmable Interrupt Controller (PIC)
- The PIC serializes interrupts, delivers them
- There are actually two daisy-chained PICs



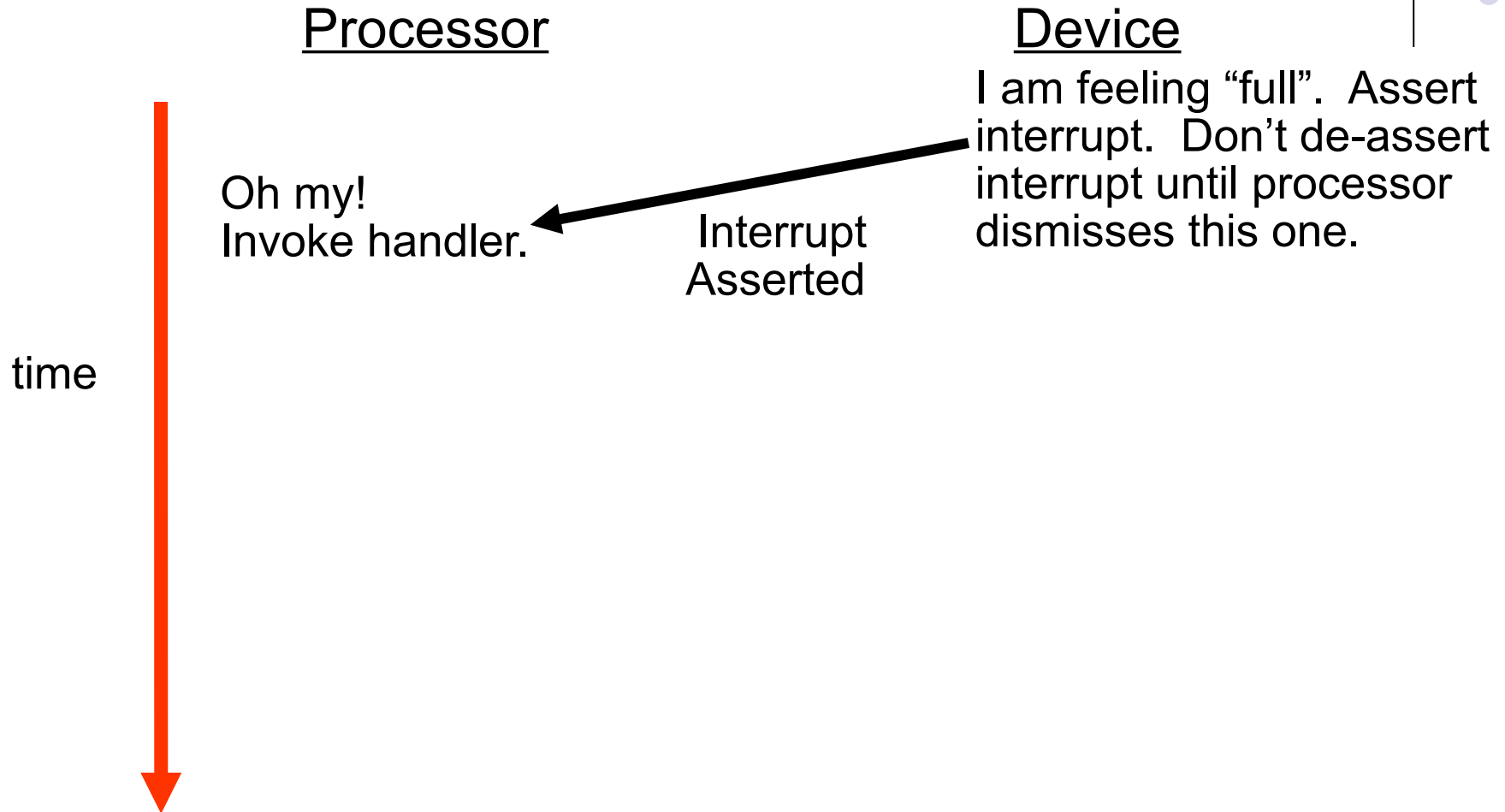
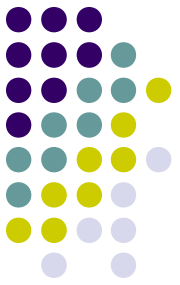
Mundane Details in x86: Interrupts and the PIC



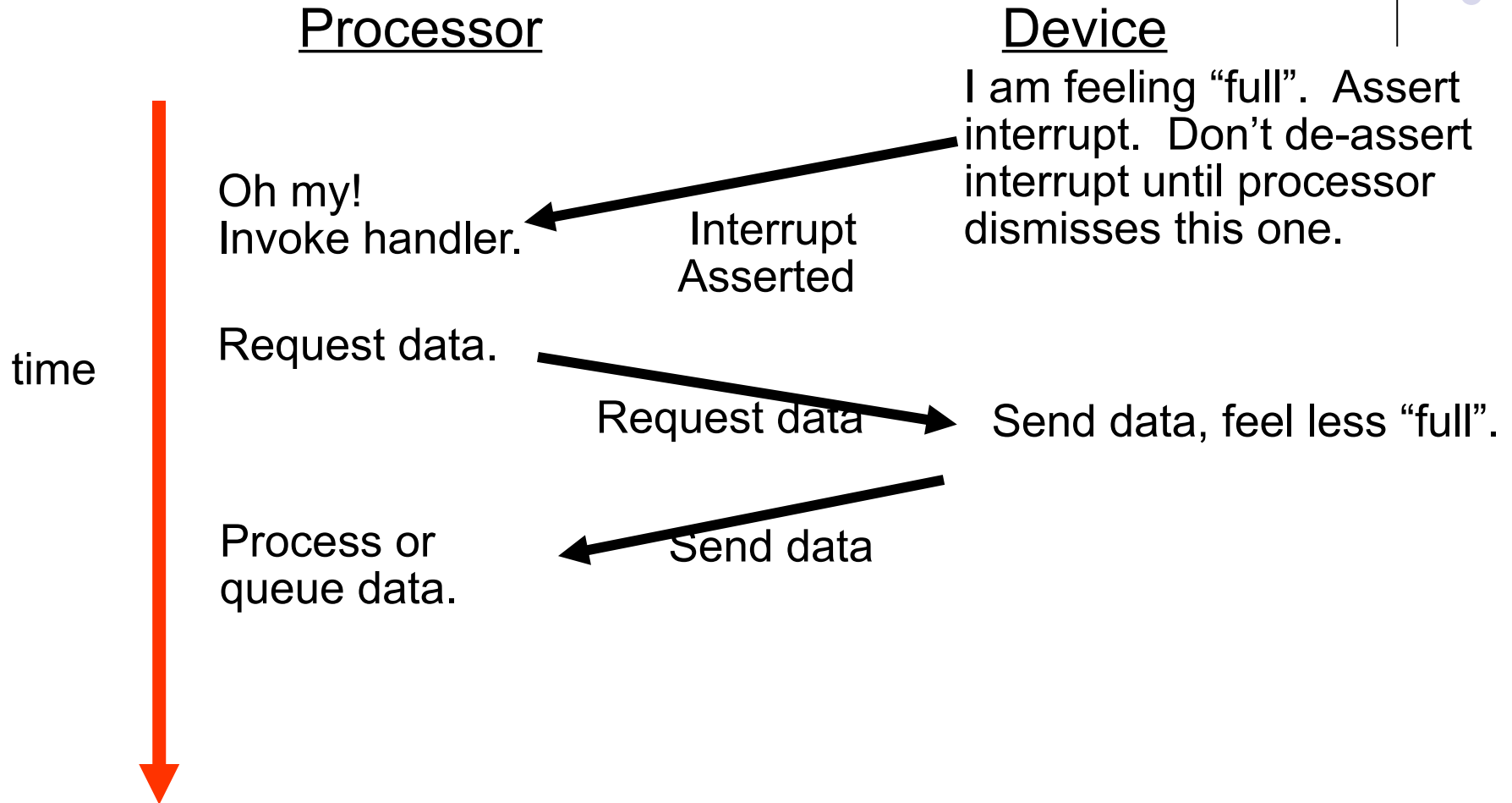
Typical Interrupt Handshake



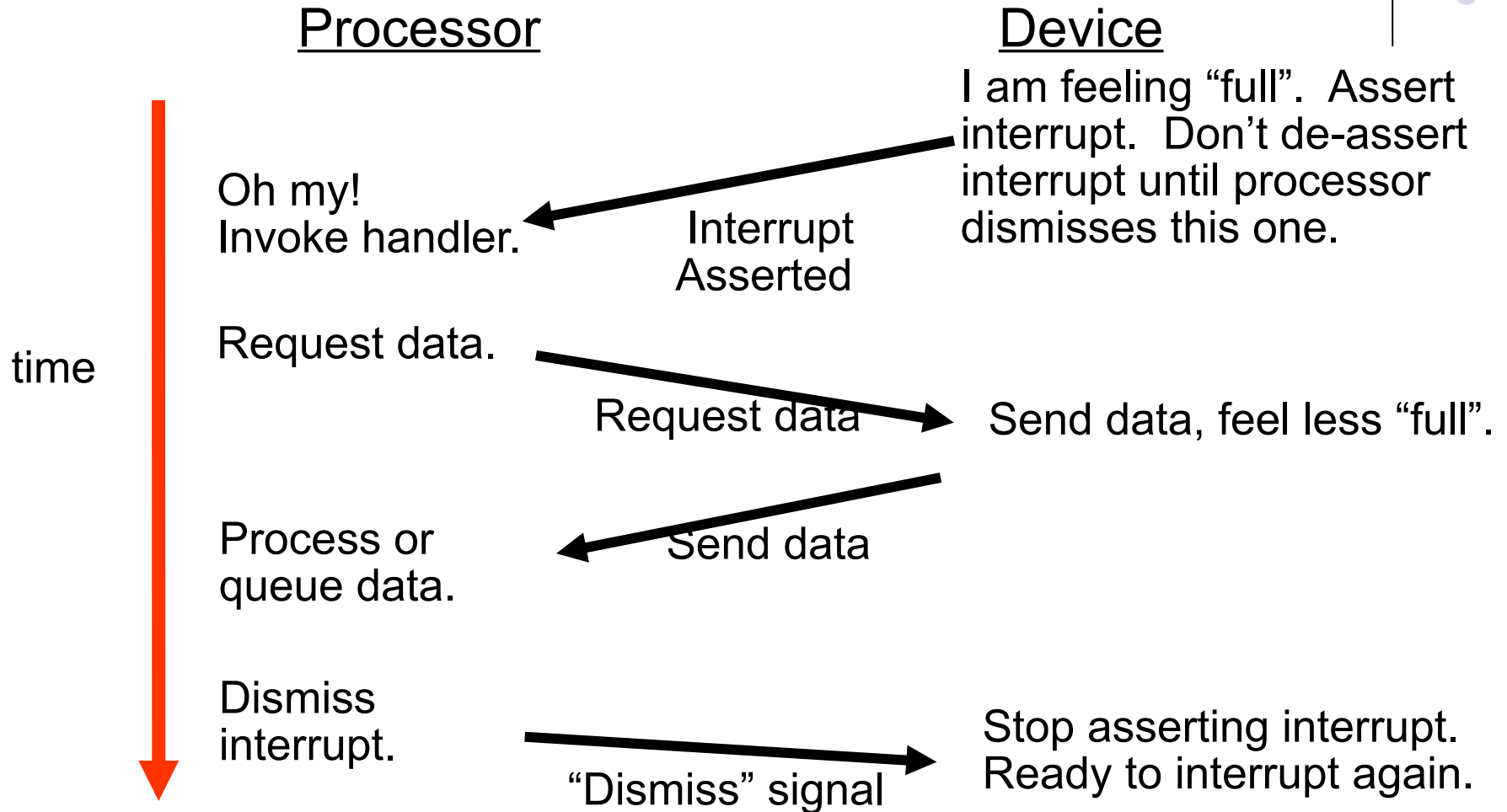
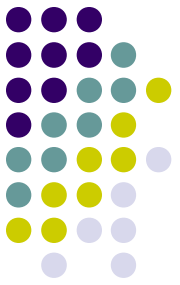
Typical Interrupt Handshake



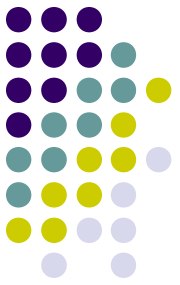
Typical Interrupt Handshake



Typical Interrupt Handshake

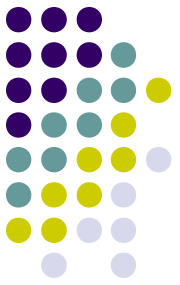


Enabling / Disabling Interrupts



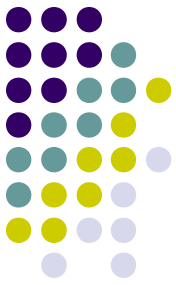
- PIC automatically defers new interrupts from a device until old one dismissed by processor.
- We also provide `disable_interrupts()`, which “disables” interrupts from ALL devices. Think of this as deferring interrupts. They are still out there, waiting to happen.
- We provide `enable_interrupts()`, which re-enables interrupts.
- Finer-grained control is also possible.

Interrupt Descriptor Table – IDT

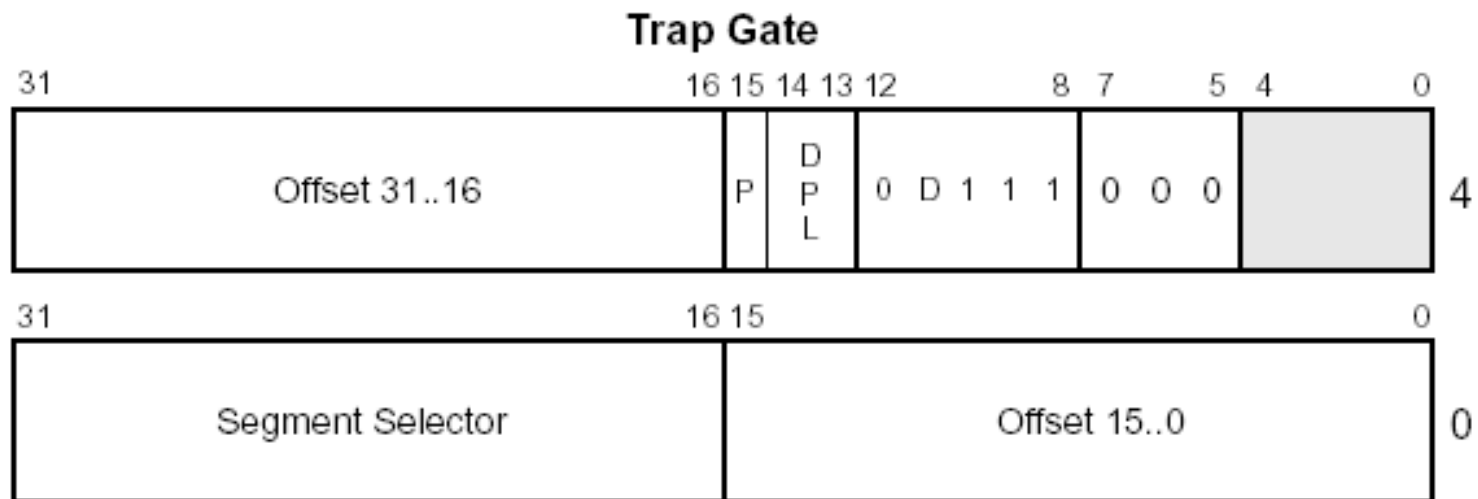


- Processor needs info on which handler to run when
- Processor reads appropriate IDT entry depending on the interrupt, exception *or* INT n instruction
- Logically, an IDT entry contains a function pointer and some flags

Interrupt Descriptor Table – IDT



- Processor needs info on which handler to run when
- Processor reads appropriate IDT entry depending on the interrupt, exception *or* INT_n instruction
- An entry in the IDT looks like this:



Interrupt Descriptor Table – IDT

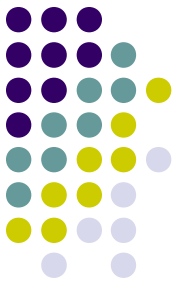


- The first 32 entries in the IDT correspond to processor exceptions. 31-255 correspond to hardware/software interrupts
- Some interesting entries:

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

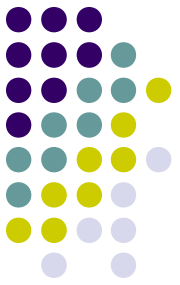
- More information in section 5.12 of intel-sys.pdf.
- Note: One “IDT” table is used for faults, traps, and interrupts

Classifying Surprises



- Asynchronous or synchronous?
 - Asynchronous – happens at a random time
 - Can be deferred (“blocked”) until a convenient time
 - Synchronous – a particular instruction is to blame
 - **Cannot** be deferred – happen when instruction happens
- What happens afterward?
 - **Retry** the surprising instruction (exception)
 - **Kill** program (exception)
 - **Run** the next instruction (trap, interrupt)

Mundane Details in x86: Communicating with Devices



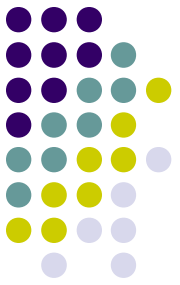
- I/O Ports
 - Use instructions like `inb(port)`, `outb(port, data)`
 - *Are not memory!*
- Memory-Mapped I/O
 - Magic areas of memory tied to devices
- PC video hardware uses *both*
 - **Cursor is controlled by I/O ports**
 - **Characters are painted from memory**

x86 Device Perversity



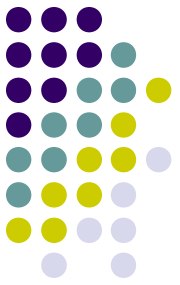
- Influence of ancient history
 - **IA-32 is fundamentally an 8-bit processor!**
 - **Primeval I/O devices had 8-bit ports**
- I/O devices have multiple “registers”
 - **Timer: waveform type, counter value**
 - **Screen: resolution, color depth, *cursor position***
- You must get the right value in the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

x86 Device Perversity



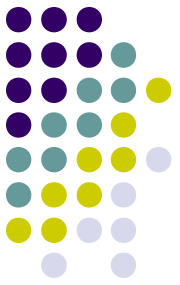
- Value/bus mismatch
 - **Counter value, cursor position are 16 bits**
 - **Primeval I/O devices *still* have 8-bit ports**
- Typical control flow
 - “I am about to tell you half of register 12”
 - “32”
 - “I am about to tell you the other half of register 12”
 - “0”

x86 Device Perversity



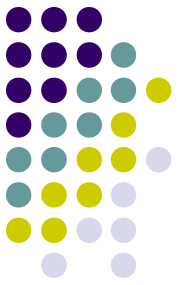
- Sample interaction
 - `outb(command_port, SELECT_R12_LOWER);`
 - `outb(data_port, 32);`
 - `outb(command_port, SELECT_R12_UPPER);`
 - `outb(data_port, 0);`
- This is not intuitive (for software people).
 - Why can't we just “`*R12 = 0x00000032`”?
- But you can't get anywhere on P1 without understanding it.

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

Writing a Device Driver



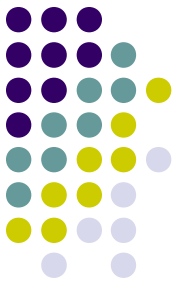
- For this project, your keyboard driver will likely have a top and bottom half
- Bottom half
 - Responds to keyboard interrupts and queues scan codes
- Top half
 - In `readchar()`, reads from the queue and processes scan codes into characters

Installing and Using Simics



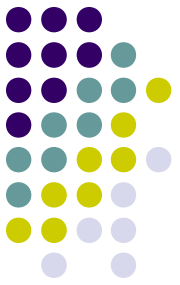
- Simics is an instruction set simulator
- Makes testing kernels much easier
- Project 1 Makefile builds floppy-disk images
- Simics boots and runs them
 - Launch `simics4` in your build directory
- Your 15-410 AFS space has `p1/`, `scratch/`
- If you work in `scratch/`, we can read your files, and answering questions can be much faster.

Installing and Using Simics: Running on Personal PC



- Not a “supported configuration”
 - Campus IP addresses can use campus license
 - Andrew's VPN should work for off-campus users (details in previous lecture)
- Download `simics-linux.tar.gz`
- Install `mtools` package
- Tweak Makefile

Installing and Using Simics: Overview of usage



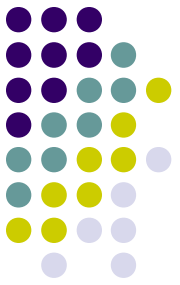
- Run simulation with `r`, stop with `^C`
- Magic instruction
 - `xchg %bx, %bx` (wrapper in `interrupts.h`)
 - This may change -- use the macros!
- Memory access breakpoints
 - `break 0x2000 -x OR break (sym init_timer)`
- Symbolic debugging
 - `psym foo OR print (sym foo)`
- See our local Simics hints! (on Project page)

Simics vs. gdb



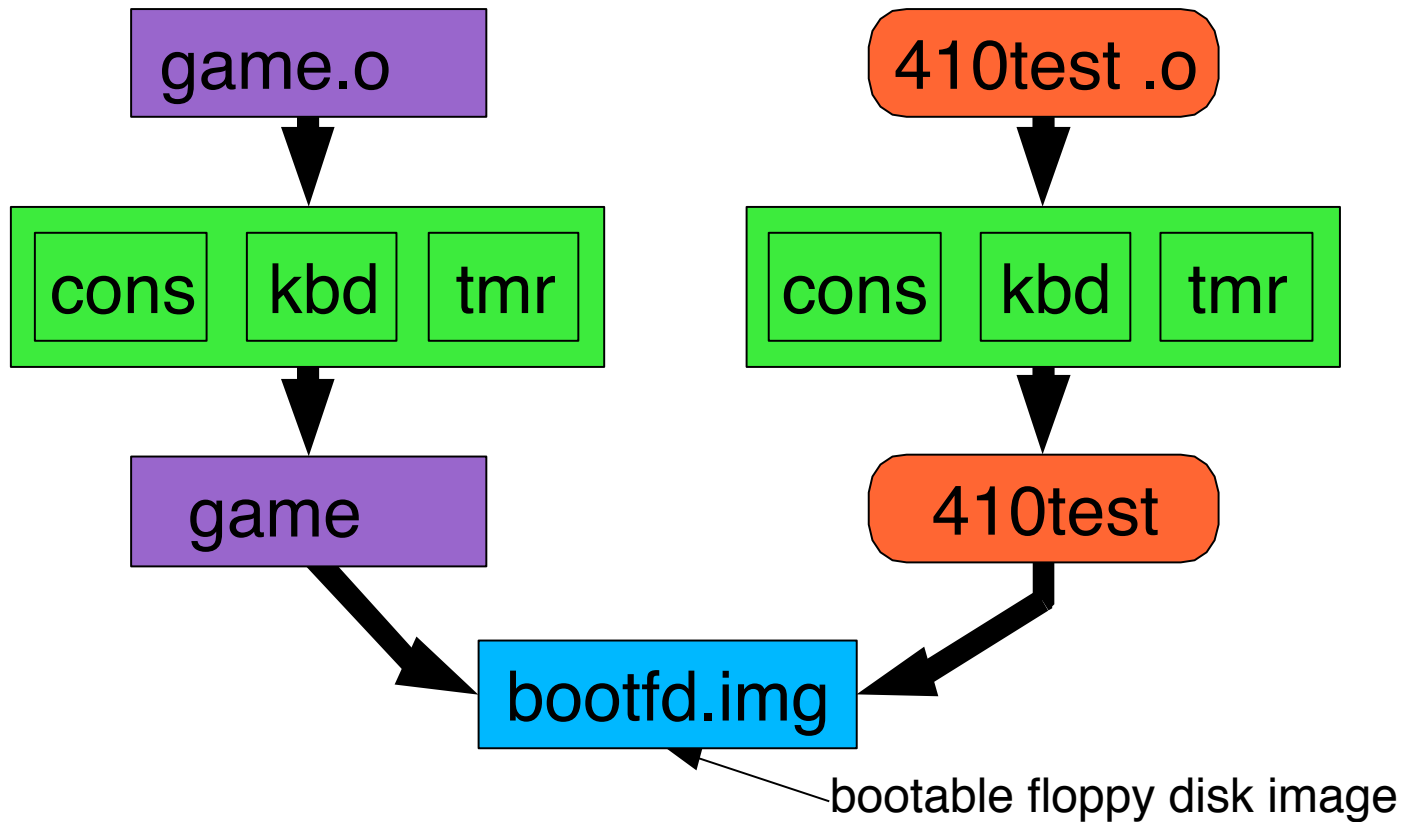
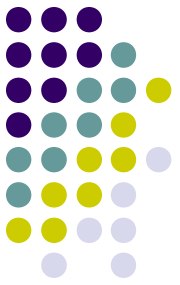
- Similar jobs: symbolic debugging
- Random differences
 - Details of commands and syntax
- Notable differences
 - Simics knows *everything* about PC hardware – all magic registers, TLB contents, interrupt masks, etc.
 - Simics is scriptable in Python

Project 1 Pieces

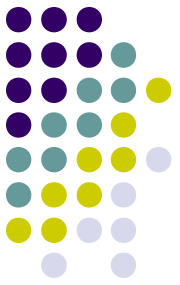


- You will build
 - A device-driver library
 - “console” (screen) driver
 - keyboard driver
 - timer driver
 - A simple game application using your driver library
- We will provide
 - underlying setup/utility code
 - A simple device-driver test program

Project 1 Pieces

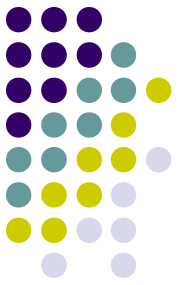


Summary



- Project 1 runs on *bare hardware*
 - Not a machine-invisible language like ML or Java
 - Not a machine-portable language like C
 - Budget time for understanding this environment
- Project 1 runs on *simulated* bare hardware
 - You probably need more than printf() for debugging
 - Simics is not (exactly) gdb
 - Invest time to learn more than bare minimum

Summary



- Project 1 runs on bare *PC* hardware
 - As hardware goes, it's pretty irrational
 - *Almost nothing* works “how you would expect”
 - Those pesky bit-field diagrams do matter
 - Getting started is tough, so please don't delay.
- This isn't throwaway code
 - We will read it
 - You will use it for Project 3
 - So spend extra time to make it really great code!