What You Need to Know for Project One

Joey Echeverria

Friday, August 29, 2003

15-410 Fall 2003

Overview

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

A Little Bit About Myself

1. My name is Joey
2. I’m a senior in ECE (sort-of)
3. I took OS with Greg Kesden last fall
4. I, like the rest of the staff, am Here to Help™

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

Project 1 Demo

1. Project 1 consists of using the console, keyboard and timer to create a simple game

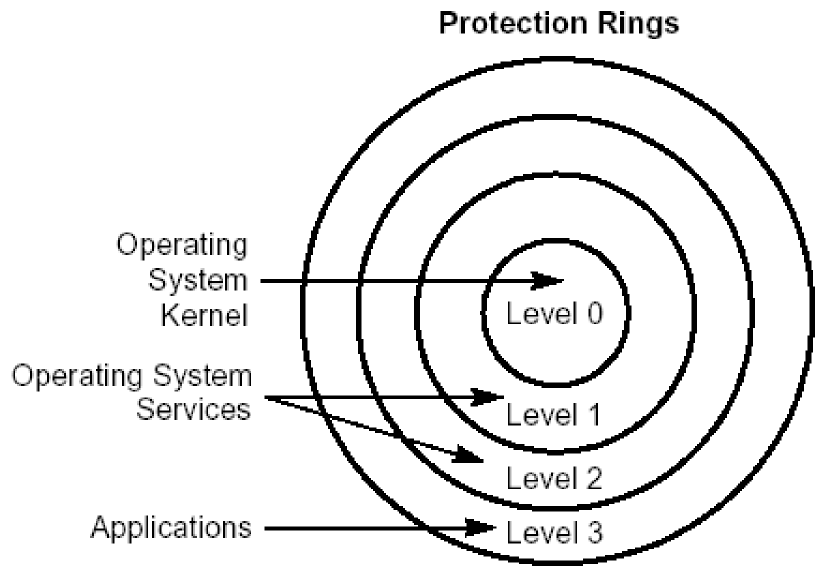
3. Demo…

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)

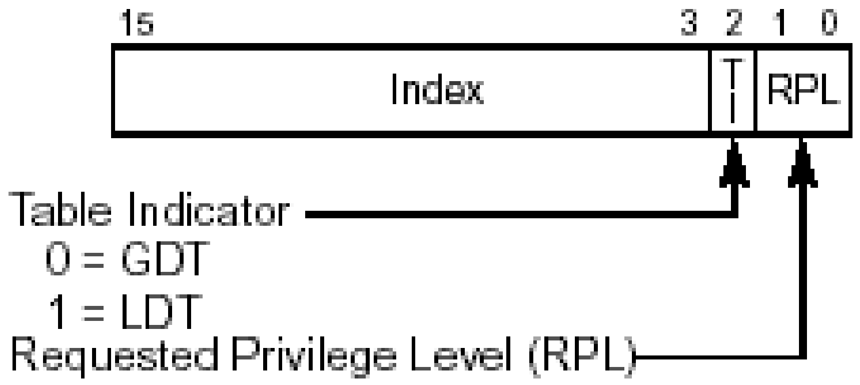
Mundane Details in x86: Privilege Levels

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: Segmentation

1. When fetching an instruction, the processor asks for an address that looks like this: CS:EIP
2. So, if %EIP is 0xbeef then this is the 48879th byte of the CS segment.
3. The CPU looks at the segment selector in %CS
4. A segment selector looks like this:

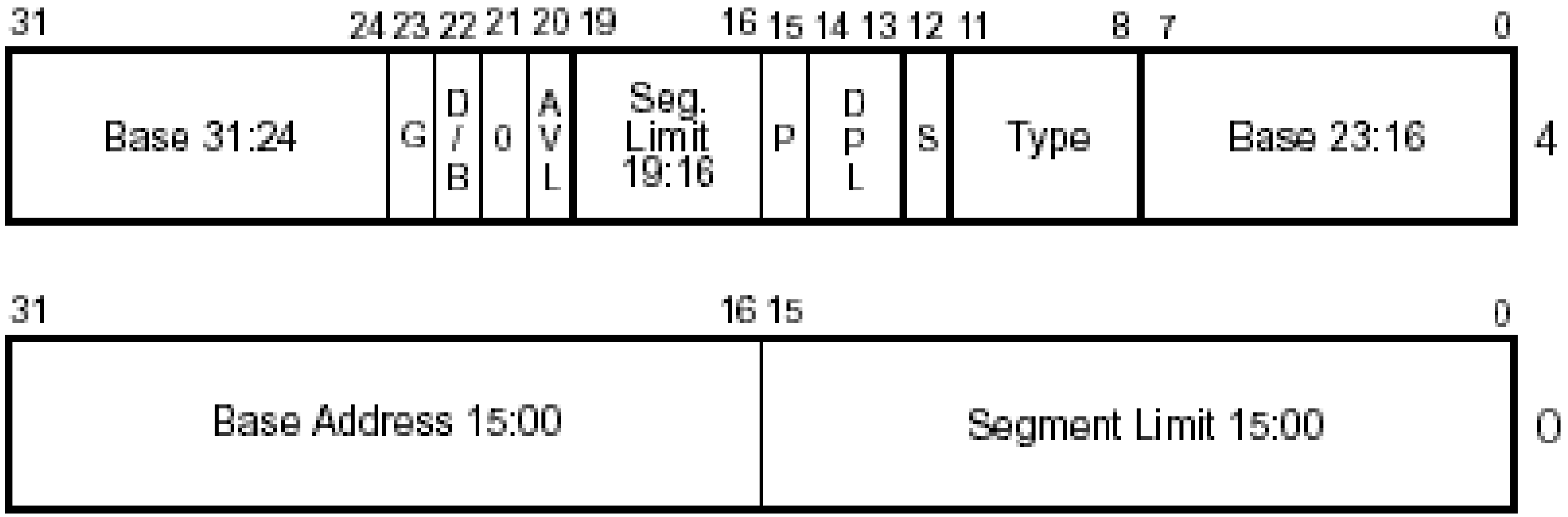


Mundane Details in x86: Segmentation

1. The segment selector has a segment number, table selector, and requested privilege level (RPL)
2. The segment number is an index into a segment descriptor table
3. The table select flag selects a descriptor table
4. There are two tables: *global descriptor table* and a *local descriptor table*
5. The RPL's meaning differs with the segment register the selector is in
6. For %CS, RPL sets the processor privilege level

Mundane Details in x86: Segmentation

1. Segments are defined areas of memory with particular access/usage constraints
2. Segments descriptors specify a base and a size
3. A segment descriptor looks like this:



Mundane Details in x86: Segmentation

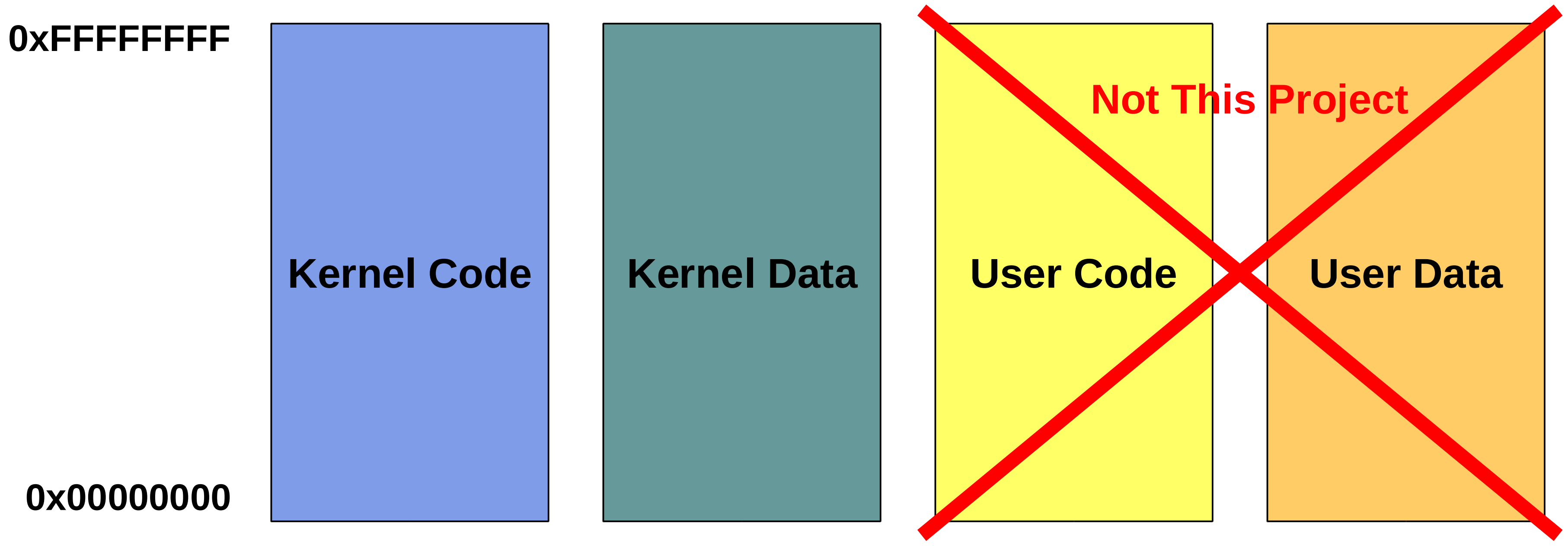
1. For our example, if the segment descriptor indexed by the segment selector in %CS specified a base address of 0xdead0000
2. Then assuming 0xbeef is smaller than the size of the segment, the address CS:EIP represents the linear virtual address 0xdeadbeef
3. This is fed into the page-directory/page-table system which will be important in project 3

Mundane Details in x86: Segmentation

1. CS is the segment register for code
2. SS is the segment register for the stack segment
3. DS is the default segment register for data read/writes, 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 these projects:



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

Mundane Details in x86: Segmentation

1. Don’t need to be concerned with the mundane details of segments in this class
2. For more information you can read the intel docs or our documentation at:

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

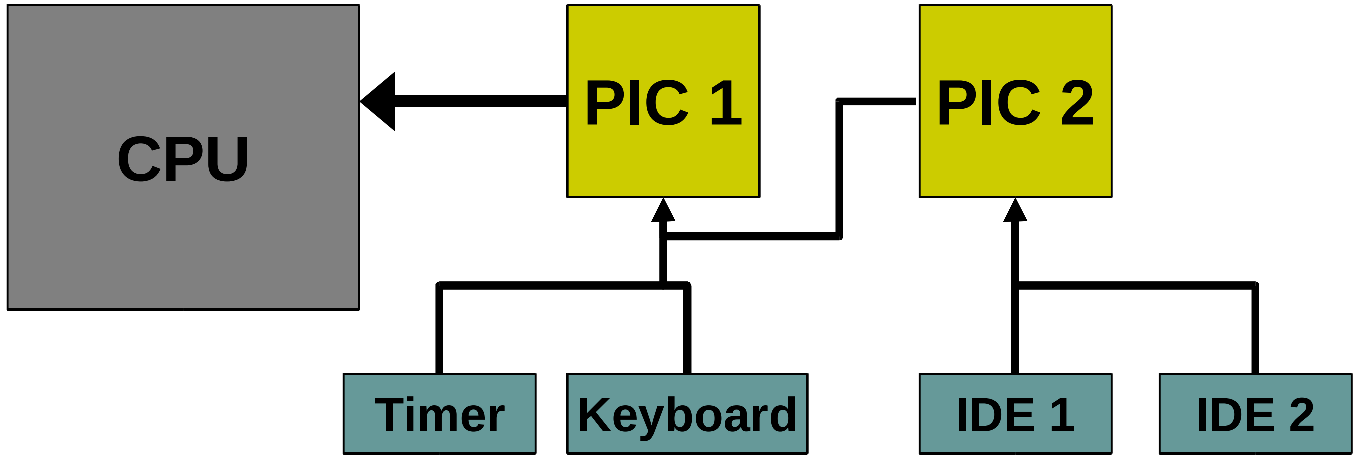
1. Sometimes user processes do stupid things
2. int gorganzola = 128/0;
3. char* idiot_ptr = NULL; *idiot_ptr = 0;
4. These cause a handler routine to be executed at PL0
5. Examples include divide by zero, general protection fault, page fault

Mundane Details in x86: Software Interrupts

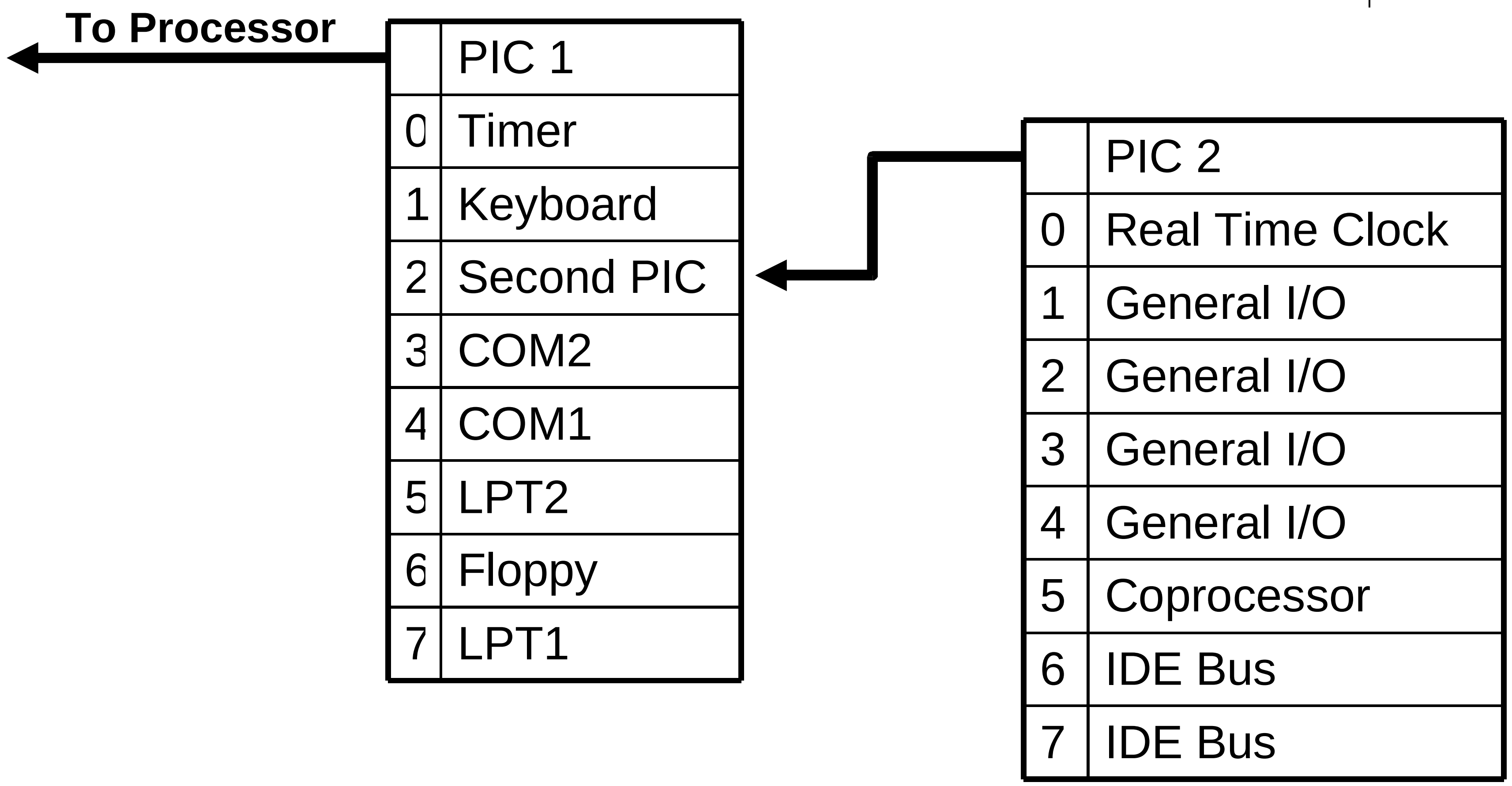
1. A device gets the kernel’s attention by raising an interrupt
2. User processes get the kernel’s attention by raising a software interrupt
3. x86 instruction **`int n`**
   (more info on page 346 of intel-isr.pdf)
4. Executes handler routine at PL0

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

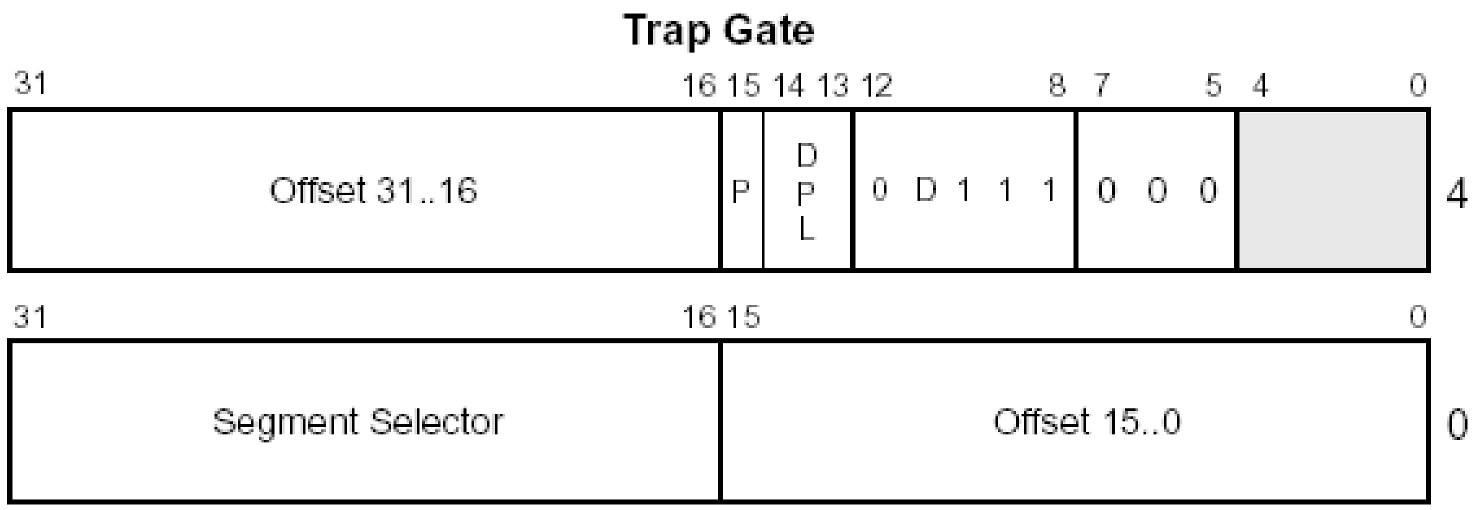


	PIC 1
0	Timer
1	Keyboard
2	Second PIC
3	COM2
4	COM1
5	LPT2
6	Floppy
7	LPT1

	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

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`**, **`outb`**
   b) use separate address space
2. Memory-Mapped I/O
   a) magic areas of memory tied to devices
   b) console is one of them

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 enqueue scan codes
3. Top half:
   a) In readchar(), read 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) If you want to run from Solaris, we can try and accommodate, but the tarball only has linux support

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 exec simics in your project dir
4. The proj1.tar.gz includes:
   a) Only simics-linux.sh right now, can provide simics-solaris.sh if you are really interested

Installing and Using Simics: Running on AFS

1. We give you some volume space
2. It is located here:
   /afs/cs.cmu.edu/academic/class/15410-f03/usr/<user-id>
3. You should use this volume
4. This way, if you have a problem the staff will have permission to look at your code
5. This will help speed up answers to your questions

Installing and Using Simics: Running on Personal PC

1. Runs under Linux or Solaris (we will try to support this)
2. As of now you need a 128.2.*.* IP or a node locked license (takes a day or two to get this)
3. Download simics-linux.tar.gz (real soon™)
4. Install mtools RPM (pointer on course www)
5. 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. Demo