Homework / Exam

• Return and Review Exam #1
• Reading
  – S&S Extracts 385-393, PIC Data Sheet
• Machine Projects
  – Start on mp3 (Due Class 19)
• Labs
  – Continue in labs with your assigned section
Interrupts

• An interrupt acts as a “hardware generated” function call – External versus Internal

• I/O device generates a signal to processor

• Processor interrupts software execution

• Processor executes appropriate interrupt service routine (ISR) for the I/O device

• ISR returns control to the software that was executing before the interrupt occurred
Adding Interrupts to the Hardware

• Add Programmable Interrupt Controller “Chip”
  – IRQ line from I/O device to PIC
  – Interrupt line from PIC to CPU
Adding Interrupts to the Hardware

• Programmable Interrupt Controller: The 8259A chip
  – Supports eight interrupt request (IRQ) lines
  – Two chips used in PC, called “master” and “slave”
  – Priority: highest to lowest order is IRQ0-1, IRQ8-15, IRQ3-7
  – Asserts INTR to CPU, responds to resulting INTA# with an 8-bit interrupt type code (“nn”) on the data bus
Adding Interrupts to the Hardware

- Interrupt Priority Encoding
- Industry Standard Architecture (ISA) Bus
Adding Interrupts to the Hardware

- **IRQ** DEVICE USED in AT, 386, 486, etc.

  0   System Timer
  1   Keyboard Controller
  2   Tied to IRQs 8-15
  3   COM 2
  4   COM 1
  5   LPT2 or Sound Card
  6   Floppy Diskette Controller
  7   LPT 1
  8   Real Time Clock
  9   Substitutes for IRQ 2
 10   Not Assigned
 11   Not Assigned
 12   PS/2 Mouse Port
 13   NPU (Numerical Processing Unit)
 14   Primary Hard Disk Controller
 15   Secondary Hard Disk Controller
Interrupt Handling

• Software that was executing “never knew what hit it” – execution is suspended until ISR ends

• Processor does the following:
  – Pushes %eflags, %cs, and %eip on stack (similar to making a function call)
  – Inhibits other interrupts (similar to cli instruction)
  – Fetches value of Interrupt Vector (IV) on bus
  – Fetches address of Interrupt Service Routine (ISR)
  – Sets %eip to entry point of the ISR
Interrupt Acknowledge Cycle

• CPU Interrupt Acknowledge (INTA#) bus cycle
  M/IO# = 0, W/R# = 0, D/C# = 0
• PIC responds with Interrupt Vector (IV) to CPU
• IV is an 8 bit number (0 – 255) equal to the IRQ number plus a constant (0x20 for Linux) passed to the processor on the data bus
• Processor uses IV as an index into the Interrupt Descriptor Table (IDT)
Interrupt Descriptor Table

- IDT is an array of 8-byte entries (gates) in memory
- Special register contains the address of the IDT
- At startup, S/W loads that register using instruction:
  ```
  lidt idt_48        # load idt with 0,0x56060
  ...
  idt_48:
    .word 0x400       # idt limit=0x400
    .word 0x6060,0x5  # idt base=0x00056060
  ```
- In our SAPC systems, the IDT is located in the Tutor memory area
Interrupt Gate Descriptor

• Contents of each Interrupt Gate Descriptor:
  
  ```c
  typedef struct desc_struct {
    unsigned short addr_lo; /* bits 0-15: handler offset lsbs */
    unsigned short selector; /* bits 16-31: selector of handler */
    unsigned char zero;     /* bits 32-39: all 0 */
    unsigned char flags;    /* bits 40-47: valid, dpl, type */
    unsigned short addr_hi; /* bits 48-63: handler offset msbs */
  } Gate_descriptor;
  ```

• An example from SAPC memory
  25 eb 10 00 00 8e 05 00
  ISR address = 0x0005eb25, CS = 0x0010, flags = 0x8e
Interrupt Service Routine

• What does an ISR do to “handle” interrupt?
• Must make I/O device turn off interrupt signal
  – If it does not † infinite loop re-executing the ISR
  – Usually accomplished by reading a status port
• Performs in or out instructions as needed
• Uses out to send End of Interrupt (EOI) to PIC
• Executes iret instruction to return
Interrupt Return

• ISR executes iret instruction
• Processor pops %eip, %cs, and %eflags (Note: Done as a single “atomic” operation)
• Popping %eflags may have side effect of re-enabling interrupts (IF flag bit in %eflags)
• The software that was interrupted resumes its normal execution like nothing happened (It’s “context” has been preserved)
Restrictions on ISR Code

- Software that was executing never got a chance to save any registers it was using!
- ISR must save context (not use ANY registers without pushing them on stack and popping them off before returning from the interrupt)
- ISR must finish its execution promptly
- ISR design must be careful interacting with ‘background’ code via shared memory to avoid interrupt windows (critical regions)
Implementing ISR Code

• Want to write most of ISR code in “C”, but …
  – Can’t push/pop scratch registers in C code
  – Can’t make C compiler generate iret instead of ret

• Write a small assembly ISR
  – Push C compiler scratch registers
  – Call a C function for main body of ISR
  – Pop C compiler scratch registers
  – Execute iret
Interrupt Windows with ISR Code

• Note following sequence in background code:
  
  ```
  inb (%dx), %al  
  orb $0x01, %al  
  outb %al, (%dx)  
  ```

• With this sequence in ISR code:

  ```
  pushl %eax  
  pushl %edx  
  inb (%dx), %al  
  orb $0x10, %al  
  outb %al, (%dx)  
  popl %edx  
  popl %eax  
  iret  
  ```
Closing Interrupt Windows

• If a sequence of instructions in background must not be interrupted, that software must:
  “inhibit” interrupts before starting (cli instruction)
  “enable” interrupts after finishing (sti instruction)
  (sti and cli instructions set or clear IF in %eflags)
• Must not disable interrupts for very long!!
• This is commonly done in software that initializes an I/O device to operate under interrupt control – preventing an interrupt from occurring prematurely
Closing Interrupt Windows

• Corrected sequence in background code:

```
cli                      # disable interrupts
inb (%dx), %al           ISR can not execute within
orb $0x01, %al           this section of code
outb %al, (%dx)
sti                      # reenable interrupts
```

• Now does not conflict with this sequence in ISR:

```
... 
inb (%dx), %al
orb $0x10, %al
outb %al, (%dx)
... 
iret
```
Interrupt Controller Programming

- On the PC, known as the PIC which stands for “Programmable Interrupt Controller”
- Programmable means it has multiple possible behaviors selectable by software programming of its registers (via its own I/O ports)
- Devices send IRQ signals to interrupt controller
- Interrupt controller prioritizes signals, sending highest one that is not masked off to the CPU
Interrupt Controller Programming

- PIC is accessible at port addresses 0x20 and 0x21 (for master), using “initialization command words” (ICWs) and “operational command words” (OCWs)
- ICWs used to set such things as:
  - How much to add to the IRQ to produce nn (8 used for DOS, 0x20 for Linux, 0x50 for Windows)
  - We trust the (Linux) bootup code to handle this setup
- OCWs used for:
  - EOI command: Reset interrupt in PIC after accepted by ISR (outb of 0x20 to port 0x20, for master)
  - Get/Set Interrupt Mask Register (port 0x21 for master)
    - Ex: 0111 1110 = 0x7e enables IRQs 0 and 7, disables 2-6
Advantage of External Interrupts

• Processor can start an I/O device and go off to do other useful work – not wait for it
• When I/O device needs attention, the ISR for that I/O device is invoked automatically
• Example:
  – When the COM1 THRE goes to the 1 state, the COM1 port ISR is invoked to load another ASCII character into the data port and returns
Exceptions and SW Interrupts

• Internal – not based on external I/O device

• Exceptions
  – Possible side effects of executing an instruction
  – Divide by zero ‡ Execute exception “ISR”

• Software Interrupts
  – Instruction int $n$ deliberately placed in the code
  – System Call ‡ Execute system call (e.g. Linux)
  – Breakpoint ‡ Return to debugger