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
  - Each Chip supports eight interrupt request (IRQ) lines
  - Asserts INTR to CPU, responds to resulting INTA# with an 8-bit interrupt vector (“0xdd”) on the data bus

- Industry Standard Architecture (ISA) Bus
  - Priority: highest to lowest order is IRQ0-1, IRQ8-15, IRQ3-7
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 but it saves more context)
  - Inhibits other interrupts (similar to cli instruction)
  - Initiates an Interrupt Acknowledge Cycle to get IV
  - Uses IV to get address of Interrupt Service Routine
  - 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) to get entry point for ISR
Interrupt Descriptor Table

- IDT is an array of 8-byte entries (gates) in memory
- A special register contains the address of the IDT
- In our Tutor VM systems, the IDT is located in the Tutor memory area
- At startup, S/W loads that register using instruction:
  ```
  lidt idt_48                       # load idt with 0,0x5604c
  ...
  
  idt_48:
    .word 0x400 # idt limit=0x400
    .word 0x604c,0x5 # idt base=0x0005604c
  ```
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 Tutor VM 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?
• Saves any additional registers it uses
• Must make I/O device turn off interrupt signal
  – If it does not → infinite loop re-executing the ISR
  – Usually accomplished by reading/writing a port
• Performs in or out instructions as needed
• Uses out to send End of Interrupt (EOI) to PIC
• Restores any additional saved registers
• Executes iret instruction to return
Interrupt Return

• When the 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 (The IF flag bit is in %eflags register)

• The software that was interrupted resumes its
  normal execution like nothing happened (It’s
  “context” has been preserved)
Implementing ISR Code

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

• Write a small assembly ISR or use inline Assy
  – Push System registers / C compiler scratch registers
  – Call C function for body of ISR code
  – Pop System Registers / C compiler scratch registers
  – Execute iret
Assembly Language ISR Wrapper

.text  # Example for MP3 IRQ0 timer chip interrupt
.globl _irq0inhand

KERNEL_DS = 0x18  # Linux kernel data segment value

_irq0inhand:
  cld  # D bit gets restored to old val by iret
  push %es  # in case user code changes data segments
  push %ds
  pushl %eax  # save C scratch regs
  pushl %edx
  pushl %ecx
  movl $KERNEL_DS, %edx
  mov %dx, %ds  # make sure our data segments are in use now
  mov %dx, %es
Assembly Language ISR Wrapper

call _irq0inthandc    # call C interrupt handler
popl %ecx            # restore C scratch registers
popl %edx
popl %eax
pop %ds              # restore data segment registers
pop %es
iret                # return to interrupted background code

/* C interrupt Handler Code for MP3 IRQ0 timer chip interrupt */
void irq0inthandc() {
    pic_end_int();       /* notify PIC that its part is done */
    tickcount++;         /* count the tick in global variable */
}
In-line Assembly ISR

- Allows us to write MP3 IRQ0 ISR without need for an assembly language wrapper function
- Store the address of irq0inhandc ISR directly in the IDT instead of address of irq0inhandc()
- Compare to the existing IRQ0 ISR code in mp3 and previous Assembly Language ISR Wrapper
- Since our exercises do not change the Direction Flag and the CS or ES registers, we don’t need to save and restore that part of the context
In-line Assembly Code

• Modified MP3 C interrupt handler code for IRQ0

```c
void irq0inthandc(void)
{
    /* hardware pushes 3 registers before entry here */
    /* compiler generated code adds stack frame */
    asm("pushl %eax"); /* save scratch registers */
    asm("pushl %ecx");
    asm("pushl %edx");
    pic_end_int(); /* notify PIC that its part is done */
    tickcount++; /* count the tick in global var */
}
```
In-Line Assembly Code

• State of the Stack during irq0inintandc execution:

* * *

Push for call to pic_end_int

Pushed by HW interrupt

Pushed by compiler

Stack frame code

%esp

%ebp

Pushed by in-line assembly code

%edx %ecx %eax %ebp %eip %cs %eflags

Background Stack State

Pop from ret in pic_end_int

Popped by in-line assembly code

Popped by in-line assembly iret
In-line Assembly Code

asm("popl %edx");  /* restore scratch registers */
asm("popl %ecx");
asm("popl %eax");
asm("movl %ebp, %esp");  /* undo stack frame */
asm("popl %ebp");
asm("iret");  /* return from interrupt */
/* compiler generated ret instruction is never reached */
}
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 $\rightarrow$ Execute an exception “ISR”
• Software Interrupts
  – Instruction int $n$ deliberately placed in the code
  – System Call $\rightarrow$ Execute system call (e.g. Linux OS)
  – Tutor Breakpoint (int $3$) $\rightarrow$ Return to debugger