

Interrupts (for CS444)

- Original is linked to www.cs.umb.edu/ulab
- What is an interrupt?
- What does an interrupt do to the “flow of control”
- Interrupts used to overlap computation & I/O
 - Examples would be console I/O, printer output, and disk accesses
- Normally handled by the OS. Thus under UNIX or Windows, rarely coded by ordinary programmers.
 - In embedded systems and real-time systems, part of the normal programming work.

1

Interrupts (Cont'd)

- Why interrupts over polling (AKA programmed i/o)? Because polling
 - Ties up the CPU in one activity
 - Uses cycles that could be used more effectively
 - Code can't be any faster than the tightest polling loop
- Bottom line: an interrupt is an asynchronous subroutine call (triggered by a hardware event) that saves both the return address and the system status

2

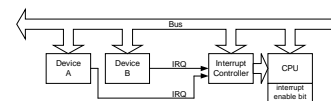
When an Interrupt Occurs

- Finish the current instruction
- Save minimal state information on stack
- Transfer to the interrupt handler, also known as the interrupt service routine (ISR)
But there is more to it than this...How do we know which device interrupted?
- And what happens if two (or more) devices request an interrupt at the same time?

3

Interrupts: Overview

- Complex hardware setup
- Needed for any multitasking OS



- Devices use IRQs to signal interrupt controller
- Like Fig. 5-5 of Tanenbaum, but note that the devices are also connected to the bus.

4

Interrupt Controller (P.I.C)

- P.I.C. stands for Programmable Interrupt Controller
- Programmable means it has multiple possible behaviors selectable by software (via its own I/O ports)
- Devices send IRQ signals to interrupt controller
- Interrupt controller prioritizes signals, sending highest to CPU, saving others for later

5

CPU Actions for Interrupt Handling

- The `sti` and `cli` instructions set and clear IF in EFLAGS to enable/disable interrupts in the CPU
- The CPU checks for interrupts between instructions if interrupts enabled (IF = 1), does its *interrupt cycle* (later slide) if there is an interrupt
- The CPU interrupt cycle executes atomically and saves certain CPU registers on the stack. It causes the interrupt handler to start executing.
- The `iret` instruction at the end of the interrupt handler restores the previously saved CPU state and resumes execution from point of interrupt (other CPUs have similar instructions).

6

Interrupt Controller Details

- Each device has an IRQ number based on its wiring to the PIC
 - Ex. COM2 uses IRQ3, timer 0 uses IRQ0
- PIC: we only consider the “master” chip
 - Supports eight interrupt request (IRQ) lines
 - Priority: highest to lowest order is IRQ0-1, IRQ8-15, IRQ3-7 (based on input wires from devices)
 - Supplies the 8-bit interrupt vector number (“nn”) on the data bus in a special bus cycle initiated by the CPU when it is first responding to an interrupt.
 - “Masks” interrupt signals as directed...

7

Interrupt Controller Programming

- PIC is accessible at i/o port addresses 0x20 and 0x21
- We depend on Linux code in Tutor for PIC initialization
- We use some operational commands in our programs:
 - EOI command: Reset interrupt in PIC by outb of 0x20 to port 0x20. SACP library call `pic_end_int()`
 - Get/Set Interrupt Mask Register (port 0x21)
 - Ex: outb of 1111 0111= 0xf7 to port 0x21 enables IRQ3 for COM2 and disables the rest.
 - SACP library call `pic_enable_irq(COM2_IRQ)` enables IRQ3 by inb of old val in port 0x21, turn off bit 3, outb back to port 0x21.

8

Interrupt Activity

- Requesting device generates a signal on IRQn
- P.I.C. checks its interrupt mask (specifically bit n) before putting out a logic high on the INTR line to the CPU.
- Between instructions, and if IF=1 in EFLAGS, the CPU sees INTR=1 on its pin and initiates its *interrupt cycle*.
- CPU uses a special bus cycle to get nn from the PIC.
- The interrupt handler for nn executes (kernel code)
- Requesting device is usually accessed in the interrupt handler and is thus notified of the completion of the event
 - Ex: UART receiver detects inb for a received char

9

CPU’s interrupt cycle

- CPU detects INTR between instructions with IF=1
- CPU signals the PIC using a special bus cycle.
- P.I.C. responds by expressing the 8-bit interrupt code, *nn*, on data lines
- CPU reads *nn* and executes `int nn` instruction:
 - Machine state saved on stack (cs:eip and eflags)
 - IF set to zero, enter kernel mode (SACP: already there)
 - Access IDT[nn] to obtain interrupt handler address
 - Interrupt handler address is loaded in CPU register EIP
 - Causing the interrupt handler to execute next

10

Interrupt Handler Details

- An interrupt handler must
 - Save all registers used
 - Issue EOI command (end-of-interrupt) to P.I.C. (outb to port 0x21, SACP library `pic_end_int()`)
 - Service the device, i.e., do whatever processing is needed for the event the device was signaling
 - Ex. Read (inb) the received character, for UART receiver interrupts
 - Restore registers
 - Finish with `iret` instruction.

11

UART Interrupts

- The UART is a real I/O device, more typical of interrupt sources than timer 0
- The UART has four ways to interrupt, we’ll cover receiver interrupts here.
- No interrupts are enabled in the UART until we command the UART to enable them, via the UART’s register 1, the IER (i.e outb to port 0x3f8+1 or port 0x2f8+1)

12

UART Receiver Interrupts

- The receiver interrupts each time it receives a char, and remembers the interrupt-in-progress
- The COM1 UART is connected to pin IR4 on the PIC, so its IRQ is 4 and its vector number `nn = 0x24`. Similarly COM2's is 3, and its `nn = 0x23`.
- The interrupt handler code must read in (with `inb`) the received char to satisfy the UART, even if no one wants the char. It also must send an EOI command to the PIC (with `outb`).
- The UART's receiver detects the `inb` for the char, and this completes the interrupt-in-progress for the UART.

13

UART Interrupts (COM1)

Initialization in kernel code

- Disallow interrupts in CPU (`cli`)
- Enable interrupts in the UART (`outb` to port `0x3f9`, IER)
- Allow COM1 interrupts to pass through the PIC by clearing the IRQ4 bit (`inb`, then `outb` to port `0x21`)
- Set up interrupt handler address in IDT [`0x24`]
- Allow interrupts (`sti`)

Shutdown in kernel code

- Disallow interrupts (`cli`)
- Disable interrupts in the UART (`outb` to port `0x3f9`, IER)
- Disallow COM1 interrupts in the PIC by setting the IRQ4 bit (`inb`, then `outb` to port `0x21`)
- Allow interrupts (`sti`)

14

UART (COM1) Interrupts: two parts of the interrupt handler

- `irq4inhand` - the outer assembly language interrupt handler
 - Save registers
 - Call C function `irq4inhandc`
 - Restore registers
 - `iret`
- `irq4inhandc` - the C interrupt handler
 - Issue the EOI command to the PIC (`outb` to port `0x20`)
 - Input the char, and whatever else is wanted

15

Timer 0 Device

- Simplest device: always is interrupting, every time it downcounts to zero.
- Can't disable interrupts in this device! Can mask them off in the P.I.C.
- We can control how often it interrupts
- Timer doesn't keep track of interrupts in progress—just keeps sending them in
- So we don't need to interact with it in the interrupt handler (but we do need to send an EOI to the PIC)

16

Timer Interrupt Software

- Initialization
 - Disallow interrupts in CPU (`cli`)
 - Unmask IRQ0 in the PIC by ensuring bit 0 is 0 in the Interrupt Mask Register accessible via port `0x21`. (SAPC library: `pic_enable_int(TIMER0_IRQ)`)
 - Set up interrupt gate descriptor in IDT, using `irq0inhand` (SAPC library `set_intr_gate(...)`)
 - Set up timer downcount to determine tick interval
 - Allow interrupts (`sti`)
- Shutdown
 - Disallow interrupts (`cli`)
 - Disallow timer interrupts by masking IRQ0 in the PIC. by making bit 0 be 1 in the Mask Register (port `0x21`) (SAPC library: `pic_disable_int(TIMER0_IRQ)`)
 - Allow interrupts (`sti`)

17

Timer Interrupts: two parts of the interrupt handler

- `irq0inhand` - the outer assembly language interrupt handler
 - Save registers
 - Call C function `irq0inhandc`
 - Restore registers
 - `iret`
- `irq0inhandc` - the C interrupt handler
 - Issue EOI (SAPC lib: `pic_end_int()`)
 - Increase the tick count, or whatever is wanted

18