Review
° I/O gives computers their 5 senses
° I/O speed range is million to one
° Processor speed means must synchronize with I/O devices before use
° Polling works, but expensive
  • processor repeatedly queries devices
° Interrupts works, more complex
  • devices causes an exception, causing OS to run and deal with the device
° I/O control leads to Operating Systems

Outline
° Instruction Set Support for OS
° Handling a Single Interrupt
° Prioritized Interrupts
° Re-entrant Interrupt Routine

Polling vs. Interrupt Analogy
° Imagine yourself on a long road trip with your 10-year-old younger brother...
  (You: I/O device, brother: CPU)
° Polling:
  • “Are we there yet? Are we there yet? Are we there yet? ....”
  • CPU not doing anything useful
° Interrupt:
  • Stuff him a color gameboy, “interrupt” him when arrive at destination
  • CPU does useful work while I/O busy

OS: I/O Requirements
° The OS must be able to prevent:
  • The user program from communicating with the I/O device directly
° If user programs could perform I/O directly:
  • No protection to the shared I/O resources
° 3 types of communication are required:
  • The OS must be able to give commands to the I/O devices
  • The I/O device notify OS when the I/O device has completed an operation or an error
  • Data transfers between memory and I/O device

Review of Coprocessor 0 Registers
° Coprocessor 0 Registers:
  name number usage
  BadVAddr $8 Addr of bad instr
  Status $12 Interrupt enable
  Cause $13 Exception type
  EPC $14 Instruction address
° Different registers from integer registers, just as Floating Point has another set of registers independent from integer registers
  • Floating Point called “Coprocessor 1”, has own set of registers and data transfer instructions
Instruction Set Support for OS (1/2)

- How to turn off interrupts during interrupt routine?
- Bit in Status Register determines whether or not interrupts enabled:
  - **Interrupt Enable bit (IE)** (0 ⇒ off, 1 ⇒ on)

<table>
<thead>
<tr>
<th>(described later)</th>
<th>IE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Register</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Set Support for OS (2/2)

- How to prevent user program from turning off interrupts (forever)?
- Bit in Status Register determines whether in user mode or OS (kernel) mode:
  - **Kernel/User bit (KU)** (0 ⇒ kernel, 1 ⇒ user)

<table>
<thead>
<tr>
<th>Assume Unused</th>
<th>KU</th>
<th>IE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Register</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- On exception/interrupt disable interrupts (IE=0) and go into kernel mode (KU=0)

Kernel/User Mode

- Generally restrict device access to OS
- **HOW?**
  - Add a “**mode bit**” to the machine: K/U
  - Only allow SW in “**kernel mode**” to access device registers
  - If user programs could access device directly?
    - could destroy each others data, ...
    - might break the devices, ...

Crossing the System Boundary

- System loads user program into memory and ‘gives’ it use of the processor
- **Switch back**
  - SYSCALL
    - request service
    - I/O
  - TRAP (overflow)
  - Interrupt

Syscall

- How does user invoke the OS?
  - **syscall** instruction: invoke the kernel (Go to 0x80000080, change to kernel mode)
  - By software convention, $v0 has system service requested: OS performs request

<table>
<thead>
<tr>
<th>Service Code Args Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>print int $a0 = integer</td>
</tr>
<tr>
<td>print float $f12 = float</td>
</tr>
<tr>
<td>print_double $a0 = double</td>
</tr>
<tr>
<td>print_string $a0 = string</td>
</tr>
<tr>
<td>read_int $a0 = integer (in $v0)</td>
</tr>
<tr>
<td>read_float $f12 = float (in $f0)</td>
</tr>
<tr>
<td>read_double $a0 = double</td>
</tr>
<tr>
<td>read_string $a0 = buffer, $a1 = length</td>
</tr>
<tr>
<td>sbrk $a0 = amount, address (in $v0)</td>
</tr>
</tbody>
</table>

- Note: most OS services deal with I/O
Example: User invokes OS (SPIM)

- Print "the answer = 42"
- First print "the answer =":
  ```
  .data
  str: .asciiz "the answer = 
  .text
  li $v0,4 # 4=code for print str
  la $a0,str # address of string
  syscall # print the string
  ```
- Now print 42
  ```
  li $v0,1 # 1=code for print_int
  li $a0,42 # integer to print
  syscall # print int
  ```

Handling a Single Interrupt (1/3)

- An interrupt has occurred, then what?
  - Automatically, the hardware copies PC into EPC ($14 on cop0) and puts correct code into Cause Reg ($13 on cop0)
  - Automatically, PC is set to 0x80000080, process enters kernel mode, and interrupt handler code begins execution
  - Interrupt Handler code: Checks Cause Register (bits 5 to 2 of $13 in cop0) and jumps to portion of interrupt handler which handles the current exception

Handling a Single Interrupt (2/3)

- Sample Interrupt Handler Code
  ```
  .text 0x80000080
  mfc0  $k0,$13  # $13 is Cause Reg
  sll   $k0,$k0,26  # isolate
  srl   $k0,$k0,28  # Cause bits
  ```
- Notes:
  - Don't need to save $k0 or $k1
    - MIPS software convention to provide temp registers for operating system routines
    - Application software cannot use them
  - Can only work on CPU, not on cop0

Handling a Single Interrupt (3/3)

- When the interrupt is handled, copy the value from EPC to the PC.
- Call instruction `rfe` (return from exception), which will return process to user mode and reset state to the way it was before the interrupt
- What about multiple interrupts?

Multiple Interrupts

- Problem: What if we're handling an Overflow interrupt and an I/O interrupt (printer ready, for example) comes in?
- Options:
  - drop any conflicting interrupts: unrealisitic, they may be important
  - simultaneously handle multiple interrupts: unrealistic, may not be able to synchronize them (such as with multiple I/O interrupts)
  - queue them for later handling: sounds good

Administrivia

- Midterm will be Wed Oct 25 5-8 P.M.
  - 1 Pimintel
  - Midterm conflicts? Talk to TA about taking early midterm ("beta tester")
  - 2 sides of paper with handwritten notes; no calculators
  - Sample midterm will be online soon (Monday?)
  - Old midterms will be online soon
- Rest of homework assignments are online: 6, 7, 8
Prioritized Interrupts (1/3)

° Question: Suppose we’re dealing with a computer running a nuclear facility. What if we’re handling an Overflow interrupt and a Nuclear Meltdown Imminent interrupt comes in?

° Answer: We need to categorize and prioritize interrupts so we can handle them in order of urgency: emergency vs. luxury.
Interrupt Levels in MIPS?

- What are they?
- It depends what the MIPS chip is inside of: differ by app Casio PalmPC, Sony Playstation, HP LaserJet printer
- MIPS architecture enables priorities for different I/O events

Interrupt Levels in MIPS Architecture

- Conventionally, from highest level to lowest level exception/interrupt levels:
  - Bus error
  - Illegal Instruction/Address trap
  - High priority I/O Interrupt (fast response)
  - Low priority I/O Interrupt (slow response)
  - (later in course, will talk about other events with other levels)

Improving Data Transfer Performance

- Thus far: OS give commands to I/O, I/O device notify OS when the I/O device completed operation or an error
- What about data transfer to I/O device?
  - Processor busy doing loads/stores between memory and I/O Data Register
- Ideal: specify the block of memory to be transferred, be notified on completion?
  - Direct Memory Access (DMA): a simple computer transfers a block of data to/from memory and I/O, interrupting upon done

Example: code in DMA controller

- DMA code from Disk Device to Memory
  .data
  Count: .word 4096
  Start: .space 4096
  .text
  Initial: lw $s0, Count # No. chars
  la $s1, Start # @next char
  Wait: lw $s2, DiskControl
  andi $s2,$s2,1 # select Ready
  beq $s2,$0,Wait # spinwait
  lb $t0, DiskData # get byte
  sb $t0, 0($s1) # transfer
  addiu $s0,$s0,-1 # Count--
  addiu $s1,$s1,1 # Start++
  bne $s0,$0,Wait # next char

Details not covered

- MIPS has a field to record all pending interrupts so that none are lost while interrupts are off; in Cause register
- The Interrupt Priority Level that the CPU is running at is set in memory
- MIPS has a field in that can mask interrupts of different priorities to implement priority levels; in Status register
- MIPS has limited nesting of saving KU,IE bits to recall in case higher priority interrupts; in Status Register

Interrupts while serving interrupts?

- Suppose there was an interrupt while the interrupt enable or mask bit is off: what should you do? (cannot ignore)

- Cause register has field--Pending Interrupts [PI]-- 5 bits wide (bits15:11) for each of the 5 HW interrupt levels
  - Bit becomes 1 when an interrupt at its level has occurred but not yet serviced
  - Interrupt routine checks pending interrupts ANDed with interrupt mask to decide what to service

<table>
<thead>
<tr>
<th>Cause</th>
<th>ExcCode</th>
<th>PI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
support for OS: User => System mode

- Bit in Status Register determines whether in user mode or OS (kernel) mode:
  - Kernel/User bit (KU) (0 => kernel, 1 => user)
  - (described later)

- On exception/interrupt disable interrupts (IE=0) and go into kernel mode (KU=0)

- How remember old KU, IE bits?
  - Hardware copies Current KU and IE bits (0-1) into Previous KU and IE bits (2-3)

support for OS: System => user mode

- OS saves user registers, performs its task and restores user registers.
  - can JR back to value saved in EPC
  - how to get back to user mode?
  - use Return from Exception (rfe)

Prioritizing Interrupts

- How implement interrupt levels?
  - Allow selective interruption via Interrupt Mask(IM) in Status Register: 5 for HW interrupts
    - Interrupt only if IE=1 AND Mask bit = 1 (bits 15:11 of SR) for that interrupt level
    - Set Mask Bits above your level to 1
  - To support interrupts of interrupts, have 3 deep stack in Status for IE,K/U bits:
    - Current (1:0), Previous (3:2), Old (5:4)

Revised Interrupt Routine 2/2

- Jump to appropriate interrupt routine
- On Return, disable interrupts using Current IE bit of Status Register
- Then restore saved registers, previous KU,IE bits of Status (Via rfe) and return to instruction determined by old EPC

Re-entrant Interrupt Routine?

- How allow interrupt of interrupts and safely save registers?
  - Stack?
    - Resources consumed by each exception, so cannot tolerate arbitrary deep nesting of exceptions/interrupts
  - With priority level system only interrupted by higher priority interrupt, so cannot be recursive
  - Only need one save area ("exception frame") per priority level

Things to Remember

- Kernel Mode v. User Mode: OS can provide security and fairness
- Syscall: provides a way for a programmer to avoid having to know details of each I/O device
- To be acceptable, interrupt handler must:
  - service all interrupts (no drops)
  - service by priority
  - make all users believe that no interrupt has occurred