Projects: Ignore exceptions (e.g., overflow)

Homework #5

Outline

- System/Machine Organization
- Executing a program
- Exceptions and Interrupts:
  - What are exceptions and interrupts
  - How are exceptions handled
- Kernel mode execution
- Back to Exceptions - the details:
  - Additions to the MIPS ISA to support exceptions
  - Additions to the control to support exceptions.
- Concurrency
- Reading: 5.6, A-7
What Does an Operating System Do?

- **Service Provider**
  - exports commonly needed facilities with standard interfaces
  - simplifies programs
  - portability
- **Executive**
  - resource manager for greatest good
- **Custodian of the Machine**
  - monitors hardware, intervenes to resolve exceptional conditions
- **Cop**
  - protects you from others
Executing a Program

- Thread of control (program counter)
  - multiple threads/programs running
- Basic steps for program execution
  - fetch instruction from Memory[PC],
  - execute the instruction,
  - increment PC
- At boot-time, begin with PC at well-known location
  - loads the Kernel

An Execution Context

- The state of the CPU associated with a thread of control (process)
  - general purpose registers (integer and floating point)
  - status registers (e.g., condition codes, HI/LO)
  - program counter, stack pointer
- Need to be able to switch between contexts
  - timeslicing: sharing the machine among many processes
  - better utilization of machine (overlap I/O of one process with computation of another)
  - different modes (Kernel v.s. user)
- Maintained by operating system
Context Switches

• Save current execution context
  ✷ Save registers and program counter
  ✷ information about the context (e.g., ready, blocked)
• Restore other context
• Need data structures in kernel to support this
  ✷ process control block
• Why do we context switch?
  ✷ Timeslicing: HW clock tick
  ✷ I/O begin and/or end
• How do we know these events occur?
  ✷ Interrupts...

Interrupts and Exceptions

• Unnatural change in control flow
• warning: varying terminology
  ✷ “exception” sometimes refers to all cases
• Exception is potential problem with program
  ✷ condition occurs within the processor
  ✷ segmentation fault
  ✷ bus error
  ✷ divide by 0
  ✷ Don’t want my bug to crash the entire machine
  ✷ page fault (virtual memory…)
Interrupts and Exceptions

- **Interrupt** is external event
  - devices: disk, network, keyboard, etc.
  - clock for timeslicing
  - These are useful events, must do something when they occur.

- **Trap** is user-requested exception
  - operating system call (syscall)

Handling an Exception/Interrupt

- Invoke specific kernel routine based on type of interrupt
  - interrupt/exception handler
- Must determine what caused interrupt
  - could use software to examine each device
  - PC = interrupt_handler
- Vectored Interrupts
  - PC = interrupt_table[i]
- Kernel initializes table at boot time
- Clear the interrupt
- May return from interrupt (RETT) to different process (e.g., context switch)

Similar mechanism is used to handle interrupts, exceptions, traps
**Execution Mode**

- **What if interrupt occurs while in interrupt handler?**
  - *Problem:* Could lose information for one interrupt
    - clear of interrupt #1, clears both #1 and #2
  - *Solution:* disable interrupts

- **Disabling interrupts is a protected operation**
  - Only the kernel can execute it
  - user v.s. kernel mode
  - mode bit in CPU status register

- **Other protected operations**
  - installing interrupt handlers
  - manipulating CPU state (saving/restoring status registers)

- **Changing modes**
  - interrupts
  - system calls (syscall instruction)

---

**A System Call (syscall)**

- **User Program**
  - `ld`, `add`, `st`, `beq`, `ld`, `sub`, `bne`

- **Trap Handler**
  - `RETT`

- **Kernel**

- **Service Routines**

- **Special Instruction to change modes and invoke service**
  - read/write I/O device
  - create new process

- **Invokes specific kernel routine based on argument**

- **kernel defined interface**

- **May return from trap to different process (e.g., context switch)**

- **RETT, instruction to return to user process**
How are Exceptions Handled?

1. Machine must save the address of the offending instruction in the EPC (Exception Program Counter)
2. Then transfer control to the Operating System (OS) at some specified address
3. OS performs some action in response
4. OS terminates program or returns (eventually) using EPC
   - could return to different program (context switch)
   - Exceptions are like an “unscheduled procedure call” to a fixed location!

How are Exceptions Handled?

• 3 types of exceptions in our current implementation
  - undefined instruction
  - arithmetic overflow
  - unaligned access

• Which Event caused Exception?
  - Option 1 (used by MIPS): Cause register contains reason
  - Option 2 Vectored interrupts: cause is part of the address.
    » addresses separated by 32 instructions
    » E.g.,

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>Exception Vector Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined instruction</td>
<td>C0 00 00 00</td>
</tr>
<tr>
<td>Arithmetic overflow</td>
<td>C0 00 00 20</td>
</tr>
<tr>
<td>Unaligned Access</td>
<td>C0 00 00 40</td>
</tr>
</tbody>
</table>

» Jump to appropriate address
Additions to MIPS ISA

- **EPC**: a 32-bit register used to hold the address of the affected instruction
- **Cause**: a register used to record the cause of the exception.
  - In the MIPS architecture this register is 32 bits, though some bits are currently unused.
- **BadVAddr**: register contains memory address at which memory reference occurred
  - when memory access exception occurs
- **Status**: interrupt mask and enable bits

Additions to MIPS ISA

- Control signals to write EPC, Cause, BadVAddr, and Status
- Need to write exception address into PC
  - increase mux to add as input interrupt handler address
- May have to undo PC = PC + 4
  - if EPC points to offending instruction (not its successor)
  - PC = PC - 4
Details of Status register

<table>
<thead>
<tr>
<th>Status</th>
<th>15</th>
<th>8</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mask</td>
<td>k</td>
<td>e</td>
<td>k</td>
<td>e</td>
<td>k</td>
<td>e</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>old</td>
<td>prev</td>
<td>current</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Mask** = 1 bit for each of 5 hardware and 3 software interrupt levels
  - 1 enables interrupts, 0 disables interrupts
- **k** = kernel/user
  - 0 => was in the kernel when interrupt occurred
  - 1 => was running user mode
- **e** = interrupt enable
  - 0 means disabled, 1 means enabled
- **Interrupt handler runs in kernel mode with interrupts disabled**
  - When interrupt occurs, 6 LSB shifted left 2 bits, setting 2 LSB to 0

Details of Cause register

<table>
<thead>
<tr>
<th>Cause</th>
<th>15</th>
<th>10</th>
<th>5</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pending</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Pending interrupt** 5 hardware levels: bit set if interrupt occurs but not yet serviced
  - handles cases when more than one interrupt occurs at the same time, or records interrupt requests when interrupts disabled
- **Exception Code** encodes reasons for interrupt
  - 0 (INT) => external interrupt
  - 4 (ADDR) => address error exception (load or instr fetch)
  - 5 (ADDR) => address error exception (store)
  - 6 (IBUS) => bus error on instruction fetch
  - 7 (DBUS) => bus error on data fetch
  - 8 (Syscall) => Syscall exception
  - 9 (BKPT) => Breakpoint exception
  - 10 (RI) => Reserved Instruction exception
  - 12 (Ovf) => Arithmetic overflow exception
How Control Detects Exceptions (Multicycle)

• Undefined Instruction
  
  Detected when no next state is defined from state 1 for the op value.
  We handle this exception by defining the next state value for all op values other than lw, sw, 0 (R-type), jump, beq, and ori as new state 12.
  Shown symbolically using “other” to indicate that the op field does not match any of the opcodes that label arcs out of state 1.

• Arithmetic overflow
  
  Logic in the ALU to detect overflow.
  Overflow is provided as an output from the ALU. This signal is used in the modified finite state machine to specify an additional possible next state for state 7 (state 13)

© Alvin R. Lebeck
CPS 104
19
Extra States to Handle Exceptions

What About the Excepting Instruction?

- Some problems could occur in the way the exceptions are handled.
- Example:
  - In the case of arithmetic overflow, the instruction causing the overflow completes writing its result, because the overflow branch is in the state when the write completes.
  - However, the architecture may define the instruction as having no effect if the instruction causes an exception; MIPS specifies this.
- When we get to virtual memory we will see that certain classes of exceptions must prevent the instruction from changing the machine state.
- This gets complex and potentially limits performance => this is why exceptions are hard
### Exception Summary

- Control is hard part of computer design
  - Large Finite-state machine
- Exceptions are the hard part of control
- Need to find convenient place to detect exceptions and to branch to state that saves PC and invokes the operating system
- Things get more difficult with pipelining and the notion of **precise exceptions**
  - No modifications to machine state from this or any following instructions

### Concurrency

- Multiple things happening simultaneously
  - Logically or physically
- Causes
  - Interrupts
  - Voluntary context switch (system call/trap)
  - Shared memory multiprocessor
The Trouble with Concurrency

- Two threads (T1, T2) in one address space or two processes in the kernel
- One counter (profile `bcopy`)

Solution: Atomic Sequence of Instructions

- **Atomic Sequence**
  - Appears to execute to completion without any intervening operations
HW Support for Atomic Operations

• Could provide direct support in HW
  ✓ Atomic increment
  ✓ Insert node into sorted list??

• Just provide low level primitives to construct atomic sequences
  ✓ called synchronization primitives
  LOCK(counter->lock);
  counter->value = counter->value + 1;
  UNLOCK(counter->lock);

• test&set (x) instruction: returns previous value of x and sets x to “1”
  LOCK(x) => while (test&set(x));
  UNLOCK(x) => x = 0;

Summary

• Operating System
• Threads of Control
  ✓ Execution context
  ✓ context switches
• Kernel vs. User Mode
  ✓ Mode bit; privileged instructions
• Interrupts and Exceptions
  ✓ what are they?
  ✓ handling exceptions
  ✓ extending the architecture
  ✓ implementing control
• Concurrency
• Synchronization and Atomic Sequences
Next Time

• Projects
• Homework #5
• Virtual Memory