Exceptions/Interrupts and Concurrency

Computer Science 104

Administrivia

- HW #6
- 104/16 project
- Readings: 8.1, 8.2.1, 8.2.4, 8.2.5 (skim others in 8.2)

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
- Synchronization and Atomic Sequences
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

```
<table>
<thead>
<tr>
<th>Time</th>
<th>PC</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>440</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>4</td>
<td>440</td>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>4</td>
<td>440</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>4</td>
<td>440</td>
<td>10</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
```

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
  - All of memory! (we'll come back to this)

- Maintained by operating system

- 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)
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?
  - Share a single resource
  - Improve Utilization of a resource
  - 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
  - “Trap” software trap, hardware trap
- Exception is potential problem with program
  - Condition occurs within the processor
  - Segmentation fault
  - Bus error
  - Divide by 0
  - Illegal instruction
  - 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 (iret) to different process (e.g., context switch)

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

• 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 vs. 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)

IA 32 Rings of Protection

<table>
<thead>
<tr>
<th>Ring</th>
<th>What runs</th>
<th>Privileges</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Kernel</td>
<td>Highest: Can do anything</td>
</tr>
<tr>
<td>1</td>
<td>Device driver</td>
<td>Rarely used</td>
</tr>
<tr>
<td>2</td>
<td>Device driver</td>
<td>Rarely used</td>
</tr>
<tr>
<td>3</td>
<td>applications</td>
<td>Lowest: Can’t harm anyone else</td>
</tr>
</tbody>
</table>
A System Call (syscall)

- Special Instruction to change modes and invoke service
  - read/write I/O device
  - create new process
  - Argument placed in %eax
- Invokes specific kernel service routine based on argument
- Kernel defined interface (Windows != MAC OS)
- May return from trap to different process (e.g., context switch)
- iret, instruction to return to user process
- IA32/AMD added syscall/sysenter & sysexit

How are Exceptions Handled?

1. Machine (hardware) 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 Control Detects Exceptions

- **Undefined Instruction**
  - Detect unknown opcode.
- **Arithmetic overflow**
  - Logic in the ALU to detect overflow
  - Overflow is provided as an output from the ALU.
- **Control circuit inserts a new PC**
  - \( \text{valPC} \leftarrow \text{interrupt\_handler} \)
  - \( \text{valPC} \leftarrow \text{interrupt\_table[ovflw]} \)
- **Control circuit changes mode**
  - \( \text{Mode/ring} \leftarrow \text{Kernel (supervisor)} \)

What About the Excepting Instruction?

- Some problems could occur in the way the exceptions are handled.
- Example:
  - You do not want a arithmetic overflow instruction to write to the destination register -> get wrong answer
- 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
  ➢ We have simple control
  ➢ Real systems have much more complex control
• Exceptions are the hard part of control
• Need to find convenient place to detect exceptions, store PC and invoke 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)
  ➢ Hyperthreading / Shared memory multiprocessor (multicore!)
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
  - Maintains mutual exclusion of the two threads’ execution for incrementing count
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); // begin atomic
    counter->value = counter->value + 1;
    UNLOCK(counter->lock); // end atomic
  - All updates to counter->value must be “protected” by Lock/Unlock
- test&set (x) instruction: returns previous value of x and sets x to “1”
  LOCK(x) => while (test&set(x));
  UNLOCK(x) => x = 0;

Solution: Atomic Sequence of Instructions

\[
\begin{array}{ccc}
T1 & T2 & \text{count} \\
\hline
\text{LOCK(ctr\_lock)} & \text{ld (count)} & \text{add} \\
\text{switch} \\
\text{st (count+1)} & \text{LOCK(ctr\_lock)} & \text{wait} \\
\text{UNLOCK(ctr\_lock)} & \text{switch} & \text{count+1} \\
\text{ld (count)} & \text{add} \\
\text{st (count+2)} & \text{UNLOCK(ctr\_lock)} & \text{count+2} \\
\text{LOCK(ctr\_lock)} & \text{while (test&set(x));} \\
\text{UNLOCK(x)} & \text{x = 0;} \\
\end{array}
\]
Alternative Solutions

- **Other Atomic instructions**
  - Swap/exchange: generalized test&set
  - Compare&Swap: Does a check before the swap

- **Disable interrupts**
  - No context switching
  - Only works for single threaded systems (not on hyperthreading or multicore)

- **Transactions**
  - Mark begin and end of transaction
  - If no conflict during transaction great
  - If conflict, then need to make it look like the instructions never executed and start over
  - Need support to either 1) undo operations or 2) delay updating state until transaction is guaranteed to be atomic

Transactions

<table>
<thead>
<tr>
<th>Time</th>
<th>Failure</th>
<th>Success</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T1</td>
<td>T2</td>
</tr>
<tr>
<td></td>
<td>begin trans</td>
<td>begin trans</td>
</tr>
<tr>
<td></td>
<td>ld (count)</td>
<td>ld (count)</td>
</tr>
<tr>
<td></td>
<td>add</td>
<td>add</td>
</tr>
<tr>
<td>switch</td>
<td>begin trans</td>
<td>begin trans</td>
</tr>
<tr>
<td></td>
<td>st (count+1)</td>
<td>st (count+1)</td>
</tr>
<tr>
<td></td>
<td>switch</td>
<td>switch</td>
</tr>
<tr>
<td></td>
<td>st (count+1)</td>
<td>st (count+1)</td>
</tr>
<tr>
<td></td>
<td>end trans</td>
<td>end trans</td>
</tr>
<tr>
<td></td>
<td>count+1</td>
<td>count+1</td>
</tr>
<tr>
<td></td>
<td>count</td>
<td>count+2</td>
</tr>
</tbody>
</table>

Conflict: Neither transaction can complete
Both will restart, eventually one completes without a conflicting Access (e.g., runs and completes without a context switch)
Software Synchronization Abstractions

- Operating System or libraries (e.g., pthreads)
  - provide abstractions for synchronization
  - Nearly all rely on an atomic instruction (xchg)

- Semaphores
  - Mutex & counting, like ints but not really...

- Condition Variables
  - Signal...Wait

- Monitors
  - Think method that executes w/ mutual exclusion

- Take OS Course...you're gonna love it!

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
Reminders

- Homeworks
- Up Next: Virtual Memory