Housekeeping

- Your goal today
  - first peek outside of the “user-level” abstraction
  - the topic is simple; it is unfamiliar so it seems strange
- Notices
  - **No office hours after class today**
  - Lab 2, status check this week, due next week
  - HW2, past due
  - Midterm, next Monday
  - Handout #8: HW 3
  - Handout #9: HW 2 solutions
- Readings
  - P&H Ch 4
Interrupt Control Transfer

- Concept: an “unplanned” function call to a system routine (a.k.a., the “handler”); and later returns control to the point of interruption
- The Trick: interrupted thread cannot anticipate the control transfer or prepare for it in any way
  - must be 100% transparent
  - not enough to impose all callee-save convention
Think, how does the handler know the PC to go back to? more on this later....
- A thread should wonder why there is a routine it doesn’t know running behind its back

Use #1: Interrupts

- How to handle rare events with unpredictable arrival time and must be acted upon quickly?
  E.g., keystroke, in-bound network, disk I/O
- Option 1: write every program with periodic calls to a service routine
  - polling frequency affects worst-case response time
  - expensive for rare events needing short response time
  What if a programmer forgets to do it?
- Option 2: keep “normal” programs blissfully unaware
  - event triggers an interrupt on-demand
  - forcefully and transparently transfer control to the service routine and back
    simpler (to the normal program) and more efficient
Use #2: Exceptions

- A systematic way to handle rare exceptional conditions in a program itself, e.g., arithmetic overflow, divide-by-0 (plus page fault, TLB miss, etc.)

- Option 1: write program with explicit checks at every potential site
  
  Do you want to check for 0 before every divide?
  
  What if a programmer forgets to do it?

- Option 2: write program for common-case scenario
  
  - detect exceptional conditions in HW
  
  - “transparently” transfer control to an exception handler that knows what to do about the condition

- Better to go with Option 1 if the exceptional condition is not rare enough (Why?)

Use #3: Multitasking Preemption

- Many unrelated programs time-multiplex a processor

- Option 1: write programs to voluntarily give up processor after running for a while
  
  - how do you “give up” the processor to another program and “get it back” later?

  What if a programmer forgets to do it?

- Option 2: keep “normal” programs blissfully unaware
  
  - a timer interrupts Process A when its time is up
  
  - handler returns to an earlier interrupted Process B
  
  - a timer interrupts Process B when its time is up
  
  - handler returns to Process A

  Neither A nor B need to know anything happened!!
Terminology: Interrupt vs Exception

- Interrupt is the more general concept
- Synchronous Interrupts (a.k.a. just “exceptions”)
  - exceptional conditions tied to a particular instruction
  - a faulting instruction cannot be finished
  - must be handled immediately
- Asynchronous Interrupts (a.k.a. just “interrupts”)
  - events not tied to instruction execution
  - some flexibility on when to handle it
  - cannot postpone forever or things start to “fall on the floor”
- Trap (a.k.a. system call)
  - an instruction to trigger exception on purpose
  
  Why not just called the handler with JAL?

User-Level Abstraction: Protection and Virtualization

- Protection: each “user-level” process thinks it is alone
  - private set of user-level architectural states modifiable by user-level instruction set
  - cannot see or manipulate (directly) state outside of this abstraction
- Virtualization: UNIX user process sees a file system
  - corresponding to 0, 1, or many storage devices
  - all devices look like files; accessed through a common set of interface paradigms (character vs block)
- OS+HW support and enforce this abstraction
  - enforce protection boundaries
  - bridge between abstract and physical
  
  OS must live beyond user-level abstractions and be more “powerful” than user-level processes
Privilege Levels

- A level is a set of architectural state and instructions to manipulate them
- A more privileged level is a superset (usually) of the less privileged level
  - lowest level has basic compute state and instructions
  - each higher level has state and instructions to control virtualization and protection of lower levels
  - only highest-level sees complete “bare-metal” hardware

user level

kernel level

“hypervisor” level for virtualizing multiple OSs

Interrupt and Privilege Change

- Convenient to combine privilege level change with interrupt/exception transfer
  - switch to next higher privilege level on interrupt—handler is more privileged code
  - privilege level restored on return from interrupt
- Interrupt control transfer is only gateway to privileged mode
  - lower-level code can never escape into privileged mode
  - lower-level code don’t even need to know there is a privileged mode

You find “user-level” interrupts in 1-level systems
MIPS Interrupt Architecture

- On an interrupt transfer, hardware saves the interrupted address to a special EPC
  - can’t just leave in PC: overwritten immediately
  - can’t use r31: need to preserve user value
- In general, hardware saves any such information that cannot be callee-saved and restored in software by interrupt handler (very few such things)
- For example, GPR can be preserved by interrupt handler using callee-saved convention
  - assume proper stack usage, interrupt handler can extend below the user process’ stack
  - MIPS convention reserves r26 and r27 for the interrupt handler to allow for very short handlers
MIPS Interrupt Architecture

- Privileged system control registers; loaded automatically on interrupt transfer events
  - Exception Program Counter (EPC, CR14): which instruction location to go back to
  - Interrupt Cause Register (CR 13): what caused the interrupt
  - Interrupt Status Register (CR 12): enable and disable interrupts, set privilege modes

- Accessed by “move from/to co-processor-0” instruction: “mfc0 Ry, CRx” and “mtc0 Ry, CRx”

Where to go on an interrupt?

- Option 1: Control transfers to a pre-fixed default interrupt handler address
  - this initial handler examines the source/cause and selects appropriate handler subroutine to do the work
  - this address is protected from user-level process so one cannot just jump or branch to it
  - even if user could jump to the address, the handler would fail trying since it did not gain privilege

- Option 2: Vectored Interrupt
  - a bank of registers holds separate specialized handler addresses for different interrupt source/cause
  - hardware transfer control directly to appropriate handler to reduce interrupt processing time
Examples of Causes

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Cause of exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Int</td>
<td>interrupt (hardware)</td>
</tr>
<tr>
<td>4</td>
<td>AdEL</td>
<td>address error exception (load or instruction fetch)</td>
</tr>
<tr>
<td>5</td>
<td>AdES</td>
<td>address error exception (store)</td>
</tr>
<tr>
<td>6</td>
<td>IBE</td>
<td>bus error on instruction fetch</td>
</tr>
<tr>
<td>7</td>
<td>DBE</td>
<td>bus error on data load or store</td>
</tr>
<tr>
<td>8</td>
<td>Sys</td>
<td>syscall exception</td>
</tr>
<tr>
<td>9</td>
<td>Bp</td>
<td>breakpoint exception</td>
</tr>
<tr>
<td>10</td>
<td>RI</td>
<td>reserved instruction exception</td>
</tr>
<tr>
<td>11</td>
<td>CoU</td>
<td>coprocessor unimplemented</td>
</tr>
<tr>
<td>12</td>
<td>Ov</td>
<td>arithmetic overflow exception</td>
</tr>
<tr>
<td>13</td>
<td>Tr</td>
<td>trap</td>
</tr>
<tr>
<td>15</td>
<td>FPE</td>
<td>floating point</td>
</tr>
</tbody>
</table>

Figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Handler Examples

- On asynchronous interrupts, device-specific handlers are invoked to service the I/O devices

- On exceptions, kernel handlers are invoked to either
  - correct the faulting condition and continue the program (e.g., emulate missing FP functionality, update virtual memory management), or
  - “signal” back to user process if a user-level handler function is registered, or
  - kill the process if exception cannot be corrected

- “System call” is a special kind of function call from user process to kernel-level service routines (e.g., open, close, read, write, seek on “files”)
Returning from Interrupt

- Software restores all architectural callee-state saved at the start of the interrupt handler

- MIPS32 uses a special jump instruction (ERET) to atomically
  - restore automatically saved processor states
  - restore privilege level
  - jump to address in EPC

- MIPS R2000 used a pair of instructions
  jr r26 // jump to a copy of EPC in r26
  rfe // restore from exception mode in delay slot

An Extremely Short Handler

_handler_shortest:
  # no prologue needed

  . . . short handler body . . .  # can use only r26 and r27
  # interrupt not re-enabled for
  # something really quick

  # epilogue
  mfc0 r26,epc  # get faulting PC
  jr 26  # jump to retry faulting PC
  rfe  # restore from exception mode

Note: You can find more examples in the book’s digital supplement. If you are really serious about it, take a look inside Linux source. It is not too hard to figure out once you know what to look for.
A Short Handler

```
_handler_short:
   # prologue
   addi sp, sp, -0x8       # continue on stack space (8 byte)
   sw r8, 0x0(sp)          # back-up r8 and r9 for use in body
   sw r9, 0x4(sp)          #

  . . . short handler body . . . # can use r26, r27, and r8, r9
   # interrupt not re-enabled

   # epilogue
   lw r8, 0x0(sp)          # restore r8, r9
   lw r9, 0x4(sp)          #
   addi sp, sp, 0x8        # restore stack pointer
   mfc0 r26, epc           # get EPC
   j r26                   # jump to retry EPC
   rfe                     # restore from exception mode
```

Nesting Interrupts

- On an interrupt control transfer, further asynchronous interrupts are disabled automatically
  - another interrupt would overwrite contents of EPC and Interrupt Cause and Status Registers
  - handler must be carefully written to not generate synchronous exceptions itself during this window of vulnerability

- For long-running handlers, interrupt must be re-enabled to not missed additional interrupts
  - handler “caller-save” contents of EPC/Cause/Status to memory (stack) before re-enabling interrupt
  - once interrupt is re-enabled, handler cannot rely on EPC/Cause/Status register contents anymore
Interrupt Priority

- Asynchronous interrupt sources are ordered by priorities
  - higher-priority interrupts are more timing critical
  - if multiple interrupts are triggered, handler handles highest-priority interrupt first
- Interrupts from different priorities can be selectively disabled by setting mask in Status register
- When servicing a particular priority interrupt, handler only re-enables higher-priority interrupts
  - ensure higher-priority interrupts not delayed
  - re-enabling same/lower-priority interrupts could cause infinite loop if a device interrupts repeatedly

Nestable Handler

```assembly
_handler_nest:
  # prologue
  addi sp, sp, -0x8   # allocate stack space for EPC
  mfc0 r26, epc      # get EPC
  sw r26, 0x00(sp)   # store EPC onto stack
  sw r8, 0x04(sp)    # allocate a register for use later
  addi r26, r0, 0x405 # set interrupt enable bit
  mtc0 r26, status   # write into status reg

  # epilogue
  addi r8, r0, 0x404 # clear interrupt enable bit
  mtc0 r8, status    # write into status reg
  ld r26, 0x00(sp)   # get EPC back from stack
  ld r8, 0x4(sp)     # restore r8
  addi sp, sp, 0x8   # restore stack pointer
  j r26              # jump to retry EPC
  rfe                # restore from exception mode
```

... interruptible
longer handler body...

# could free-up more registers
# to stack if needed
Implementing Interrupt in a Pipeline

Precise Interrupt/Exception

Sequential Code Semantics

1. \( i_1 \)
2. \( i_2 \)
3. \( i_3 \)

Overlapped Execution

\( i_1' \)
\( i_2' \)
\( i_3' \)

A precise interrupt appears (to the handler) to have taken place exactly between two instructions:

- older instructions finished completely
- younger instructions as if never happened
- on synchronous interrupts, execution stops just before the faulting instruction
“Flushing” a Pipeline

<table>
<thead>
<tr>
<th></th>
<th>t₀</th>
<th>t₁</th>
<th>t₂</th>
<th>t₃</th>
<th>t₄</th>
<th>t₅</th>
<th>t₆</th>
<th>t₇</th>
<th>t₈</th>
<th>t₉</th>
<th>t₁₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>l₀</td>
<td>l₁</td>
<td>l₂</td>
<td>l₃</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>l₉</td>
<td>l₉⁺₁</td>
<td>l₉⁺₂</td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>l₀</td>
<td>l₁</td>
<td>l₂</td>
<td>l₃</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>l₉</td>
<td>l₉⁺₁</td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>l₀</td>
<td>l₁</td>
<td>l₂</td>
<td>l₃</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>l₉</td>
<td>l₉⁺₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>l₀</td>
<td>l₁</td>
<td>l₂</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
</tr>
<tr>
<td>WB</td>
<td>l₀</td>
<td>l₁</td>
<td>l₂</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
<td>bub</td>
</tr>
</tbody>
</table>

- Kill faulting and younger instructions; drain older instructions
- Don’t start handler until faulting inst. is oldest inst.
- Better yet, don’t start handler until pipeline is empty for 100% bullet-proof privilege change

Better to be safe than to be fast

Exception Sources in Different Stages

- **IF**
  - instruction memory address/protection fault
- **ID**
  - illegal opcode
  - trap to SW emulation of unimplemented instructions
  - system call instruction (a SW requested exception)
- **EX**
  - invalid results: overflow, divide by zero, etc.
- **MEM**
  - data memory address/protection fault
- **WB**
  - nothing can stop an instruction now...

Okay to associate async interrupts (I/O) with any instruction/stage we like
Pipeline Flush for Exceptions

Diagram showing pipeline stages:
- IF (Instruction Fetch)
- ID (Instruction Decode)
- EX (Execution)
- MEM (Memory Access)
- WB (Write Back)

Instruction carries PC value throughout pipeline.

Delay Slot and Precise Exception

- Non-atomic instructions leave arch. state imprecise
- What if faulting inst. is in a branch delay slot?
  - EPC does not have enough information to restart from if branch was taken
  - MIPS set BD bit in Cause register and saves PC–4 in EPC if faulting instruction is in a branch delay slot (and don’t ever put a branch in a branch delay slot)
  
  Execute preceding branch instruction twice, okay?
  How about “JALR r31”?

- What if faulting inst is in a load delay slot?
  - delayed load value becomes visible to delay slot inst. after returning from exception
  - MIPS says don’t ever use load’s destination register as input operands in load delay slot