Exceptions and Interrupts

- Interrupts: Notification of external events
- Exceptions: Situations caused by program, requiring OS
- Also: A bit about the OS

External Events
- Focus so far: running an application
  - Low level coding
  - C
  - Assembly
  - How to execute the instructions...
  - And store the data...
  - And give the illusion of a uniform address space...
- System software (OS) has to deal with external events
  - Which may come at unexpected times
  - Data arrives on network...
  - Disk complete read request...
  - Fixed interval timer...

First question: Finding out?
- Suppose we expect an outside event
  - E.g., requested disk drive read something...
  - It will get back to us later with data (think 10M cycles)
- How do we know when its done?
  - Option 1: Polling
    - Ask it periodically
    - "Are we there yet?" No... "Are we there yet?" No
    - Downside: can be inefficient (processor busy asking)
Interrupts

• Step 1: External device raises an interrupt
  • “Hey, processor! I need your attention!”
  • Different interrupt numbers, specifies which one it is
  • Multiple interrupts at once?
    • Interrupt controller prioritizes which one goes to processor
• Multiple interrupts at once?
  • Interrupt controller prioritizes which one goes to processor

• Step 2: CPU transfers control to OS interrupt handler
  • Stops what it's doing (drain pipeline: stall front end until empty)
  • Jumps into interrupt handler (and saves current PC)
  • Switches into privileged mode

• Step 3: OS runs interrupt handler
  • Software routine to do whatever needs to be done

• Step 4: OS returns from interrupt
  • Jumps back to application code, leaving privileged mode

Interrupt handlers

• How does processor know where to jump?
  • OS sets up interrupt vector in system startup
  • Array of PCs to jump to for interrupt routines
  • Indexed by interrupt number

• What if…?
  • Another interrupt happens while handling the first one?
  • Or an interrupt happens during interrupt vector is setup?
  • Or…
Interrupt handlers

• How does processor know where to jump?
  • OS sets up interrupt vector in system startup
  • Array of PCs to jump to for interrupt routines
    • Indexed by interrupt number
  • What if...
    • Another interrupt happens while handling the first one?
    • Or an interrupt happens during interrupt vector is setup?
    • Or...
      • OS can enable/disable interrupts (privileged instruction)
        • Allows it to prevent problematic situations
          "Look, this is important, don't bother me right now!"

Speaking of OS code... where is it?

• Where does OS code reside?
  • In memory...
  • But doesn't application think it has all of memory to itself?

Speaking of OS code... where is it?

• In memory...
  • But doesn't application think it has all of memory to itself?

• Well sort of...
  • It doesn't think anything exists past the top of the stack...
  • So the OS "lives" there
  • Same physical pages mapped into all processes' address spaces
  • Privileged bit in page table prevents access by "normal" code
    • Mapping only "valid" when in privileged mode

Timer Interrupt: Heart of multitasking

• Common interrupt: timer interrupt
  • "Ticks" at fixed interval
  • Gives OS a chance to change currently running program
  • ...and keep track of the current time
  • ...and anything else it needs to do

• This is what lets your computer run multiple programs
  • The OS switches between them quickly
  • Enabled by timer interrupt giving control to OS

Exceptions: Like interrupts, but not...

• Interrupts: external events
  • Asynchronous—don't really "belong to" any current instruction
• Exceptions: unusual circumstances for an instruction
  • Belong to one particular instruction
  • Examples:
    • Page fault: load or store missing translation
    • Divide by 0
    • Illegal instruction
    • Bits do not encode any valid instruction
    • Or, privileged instruction from user code

Interrupts vs Exceptions

• Exceptions:
  • Processor must (typically) tell OS which instruction caused it
  • OS may want to restart from same instruction
    • Example: page fault for valid address (on disk)
  • Or OS may kill program:
    • Segmentation fault: (or other fatal signal)
    • Aside: OS sends "signals" to program to kill them
      • Segfault = SIGSEGV
    • Programs can "catch" signals and not die...
    • But not in this class...
• Interrupts: no particular instruction
  • But OS will always restart program after last complete insn
  • Both require precise state: insns either done, or not
  • Division between "done and not done" in program order
Precise state

- Instructions either done or not: sounds obvious right?
  - Need to ensure no state change (reg or mem) if not done
  - Also: need "clean" division.
    - Instructions before exception, all done
    - Instructions after (and including) exception, no effect

- For interrupts:
  - Must be precise, but division can be anywhere.

Handling Exceptions

- Exceptions handled just like interrupts
  - Some ISAs just give them interrupt numbers
  - Others have separate numbering for exceptions

System calls: Exceptions on purpose

- Programs need OS to do things for them
  - Read/write IO devices (including printing, disks, network)
  - Tell "real" time of day
  - Spawn new processes/execute other programs
  - Any interaction with the "outside world"

Admiral Ackbar Explains System Calls

System calls: Kind of slow

- Bothering OS for stuff: kind of slow
  - Empty pipeline...
  - Transfer control/change privilege
  - Have OS figure out what you want...
  - Then do it...
  - Then drain pipeline again
  - Then jump back into program
- For long tasks, overhead to enter/leave is amortized
  - Reading disk (very slow)
- For short tasks, overhead is very high
  - Get current time of day

System calls: Exception on purpose

- Programs need OS to do things for them
  - Special instruction which traps into OS
  - Basically just causes exception—specifically for this purpose
  - OS gets control (in privileged mode), and does what program asked
  - Knows what program wants by arguments in registers
  - May deny request and not do it...then returns an error
Avoiding slowness

- Userspace (not OS) libraries help avoid by buffering
  - Example: malloc
  - malloc does not ask OS for more memory on every call
  - Instead, malloc asks OS for large chunks of memory
  - Then manages those chunks itself (in user space)
  - Pedantic annoyance: malloc is not a system call!

Vsycalls: a slick trick

- Linux has a slick trick: vsycalls
  - Don’t actually make a system call!
  - Example: get current time of day
    - Just needs to read an int (time in seconds)
  - OS maps vsycall page into all processes
    - Read/execute only
    - All processes map to same physical page
  - OS writes current time to fixed location on this page
    - On each timer interrupt
  - gettimeofday “system call” actually not a system call
    - Just library function which jumps onto vsycall page
    - Code there reads time and returns it

Wrap-up

- Summary:
  - Interrupts: Notification of external events
  - Exceptions: Unusual things for an instruction
  - Both handled by OS, very similar behavior
  - System calls: Ask OS to do something (also, like exception)

Going to talk about IO next
Then pipelines...

Then done!