Outline for today

- Objectives
  - Dynamic Voltage Scheduling
  - System calls, interrupts, and kernel code

- Announcements
  - Added slides will be posted
  - Other slides come from link to author presentation

Crossing Protection Boundaries

- For a user to do something "privileged", it must invoke an OS procedure providing that service. How?

- System Calls
  - special trap instruction that causes an exception which vectors to a kernel handler
  - parameters indicate which system routine called
User / Kernel Modes

- Hardware support to differentiate between what we'll allow user code to do by itself (user mode) and what we'll have the OS do (kernel mode).
- Mode indicated by status bit in protected processor register.
- Privileged instructions can only be executed in kernel mode (I/O instructions).
- Protected memory space

Interrupts and Exceptions

- Unnatural change in control flow
- Interrupt is external event
  - devices: disk, network, keyboard, etc.
  - clock for timeslicing
  - These are useful events, must do something when they occur.
- Exception is potential problem with program
  - segmentation fault
  - bus error
  - divide by 0
  - Don’t want my bug to crash the entire machine
  - page fault (virtual memory…)
- System calls leverage this mechanism
An Execution Context

• The state of the CPU associated with a thread of control (process context)
  – general purpose registers (integer and floating point)
  – status registers (e.g., condition codes)
  – program counter, stack pointer

• Kernel executes in process context during system calls
  – Preemptible, kernel capable of sleeping
  – Linux system calls must be reentrant

• Need to be able to switch between contexts
  – better utilization of machine (overlap I/O of one process with computation of another)
  – timeslicing: sharing the machine among many processes

A System Call

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

• Invokes specific kernel routine based on argument
  – Syscall # in eax register

• kernel defined interface
  – Arguments passed in registers

• May return from trap to different process -- schedule()

• instruction to return to user process
Usual Path to Invoking System Call

Role of Interrupts in I/O

So, the program needs to access an I/O device...

- Start an I/O operation (special instructions or memory-mapped I/O)
- Device controller performs the operation asynchronously (in parallel with) CPU processing (between controller’s buffer & device).
- If DMA, data transferred between controller's buffer and memory without CPU involvement.
- Interrupt signals I/O completion when device is done.
CPU handles Interrupt

- Device raises interrupt line, CPU detects this, CPU stops current operation, disables interrupts, enters kernel mode, saves current program counter, jumps to predefined location, saves other processor state needed to continue at interrupted instruction.

- For each interrupt number, jumps to address of appropriate interrupt service routine. [do_IRQ()]
  - Interrupt context
  - Kernel stack of whatever was interrupted
  - Can not sleep

- Handlers on this line do what needs to be done. [handle_IRQ_event()]
  - Unless SA_INTERRUPT specified when registered, re-enable interrupts during handler execution
  - If line is shared, loop through all handlers

- Restores saved state at interrupted instruction [ret_from_intr()], returns to user mode.

Interrupt Control

- local_irq_disable() and local_irq_enable() -- affecting all interrupts for this processor
- local_irq_save(...) and local_irq_restore(...) – save and disable interrupts on this processor and restore previous interrupt state
- disable_irq(irq), disable_irq_nosynch(irq), enable_irq(irq) – affecting particular interrupt line

- Informational:
  - irqs_disabled() – local interrupts disabled?
  - in_interrupt() – in interrupt context or process context?
  - In_irq() – executing an interrupt handler?
Bottom Half Processing

- Deferring work that is too heavyweight for interrupt handling
- Mechanisms:
  - Softirqs
    “soft interrupts”, statically defined (32 max.) action functions that can run concurrently on SMP pending when bit set in 32-bitmask (usually set in associated interrupt handler [raise_softirq()]) run with interrupts enabled, proper locking required
  - Tasklets
    dynamically created functions that are built upon softirqs, two of the same type can not run concurrently lists of tasklet_struct hooked to 2 of the softirqs
  - Workqueue
    implemented as kernel-based “worker” threads with process context of their own -- thus allowed to sleep