CS 423
Operating System Design: Interrupts 2

Professor Adam Bates
Spring 2016
Goals for Today

- **Learning Objective:**
  - Understand the inner workings of interrupt handlers
- **Announcements, etc:**
  - Poll for Special Topics is up on Piazza
  - HW1 due on 2/1 23:59 UTC-11 (Or 2/2 4:00am local)
  - Note: 3 Office Hours between now and that due date.
  - Group requests will be out Wednesday

**Reminder:** Please put away devices at the start of class
HW1 Questions

Did parent terminate on “Which of the following is true when a child process terminates and its parent never waits for it?”

“How are threads scheduled (vs. processes)?”

“Which of the following lock request sequences results in a deadlock…”

Are there quizzes, and will they be announced?
How does interrupt handling change the instruction cycle?

Fetch Stage

Execute Stage

Interrupt Stage

START → Fetch next instruction → Execute Instruction → Check for INT, init INT handler → HALT

Interrupts disabled
Interrupts Recap

- Hardware generated:
  - Different I/O devices are connected to different physical lines (pins) of an “Interrupt controller”
  - Device hardware signals the corresponding line
  - Interrupt controller signals the CPU (by signaling the Interrupt pin and passing an interrupt number)
  - CPU saves return address after next instruction and jumps to corresponding interrupt handler
Interrupts Recap

- Hardware generated:
  - Different I/O devices are connected to different physical lines (pins) of an “Interrupt controller”
  - Device hardware signals the corresponding line
  - Interrupt controller signals the CPU (by signaling the Interrupt pin and passing an interrupt number)
  - CPU saves return address after next instruction and jumps to corresponding interrupt handler
Interrupts Recap

- **Software Interrupts:**
  - Interrupts caused by the execution of a software instruction:
    - INT <interrupt_number>
  - Used by the system call `interrupt()`
- Initiated by the running (user level) process
- Cause current processing to be interrupted and transfers control to the corresponding interrupt handler in the kernel
Interrupts Recap

▪ Exceptions
  ▪ Initiated by processor hardware itself
  ▪ Example: divide by zero

▪ Like a software interrupt, they cause a transfer of control to the kernel to handle the exception
Interrupt Handlers

Designing an Interrupt handler (top half):

- request_irq (irq, handler, flags, name, dev)
- free_irq (irq, dev)

Notes:
- handler is a pointer to the interrupt handler function
- Interrupt handlers need not be re-entrant (same irq is masked until handler exits)
- IRQ lines can be shared by multiple devices. The parameter dev is a unique “cookie” to be supplied by the given device and checked by the handler
- The kernel sequentially invokes all handlers registered for a given irq
INTs, Priorities, & Blocking
Interrupts (as the name suggests) have the highest priority (compared to user and kernel threads) and therefore run first

- What are the implications on regular program execution?
• Interrupts (as the name suggests) have the highest priority (compared to user and kernel threads) and therefore run first
  • What are the implications on regular program execution?
    • Must keep interrupt code short in order not to keep other processing stopped for a long time
    • Cannot block (regular processing does not resume until interrupt returns, so if the interrupt blocks in the middle the system “hangs”)
• Can an interrupt handler use malloc()?
- Can an interrupt handler use malloc()?  
- Can an interrupt handler write data to disk?
- Can an interrupt handler use malloc()?
- Can an interrupt handler write data to disk?
- Can an interrupt handler use busy wait?
  - E.G. — while (!event) loop;
- Can an interrupt handler use `malloc()`?
- Can an interrupt handler write data to disk?
- Can an interrupt handler use busy wait?
  - E.G. — `while (!event) loop;`
- Can you call any function outside of the interrupt handler?
• Can an interrupt handler use malloc()?  
• Can an interrupt handler write data to disk?  
• Can an interrupt handler use busy wait?  
  ▪ E.G. — while (!event) loop;  
• Can you call any function outside of the interrupt handler?

Note to beginning kernel hackers: This also means that any function that is (assumed to be) interrupt-safe function in the kernel can’t be extended to do these things. > _ <
Interrupt Context
Interrupt Context

- Interrupt handlers execute in the kernel. They are historically “trusted” (at least in Linux)
- Problem?
Interrupt Context

- Interrupt handlers execute in the kernel. They are historically “trusted” (at least in Linux)
- Problem?
  - A bad handler (e.g., a buggy device driver that does not correctly handle interrupts from its I/O device) can crash the kernel or cause inefficiencies and poor performance
Where does an interrupt’s stack come from?
Interrupt Context

Where does an interrupt’s stack come from?

- Option 1: Use the stack of the interrupted process
- Option 2: Share a common single kernel stack
- Option 3: Use own stack
A Note on Multicore
How are interrupts handled on multicore machines?
A Note on Multicore

- How are interrupts handled on multicore machines?
  - On x86 systems each CPU gets its own local Advanced Programmable Interrupt Controller (APIC). They are wired in a way that allows routing device interrupts to any selected local APIC.
  - The OS can program the APICs to determine which interrupts get routed to which CPUs.
  - The default (unless OS states otherwise) is to route all interrupts to processor 0
Designing an Interrupt Handler (Bottom Half):

- Since the interrupt handler must be minimal, all other processing related to the event that caused the interrupt must be deferred
  - Example:
    - Network interrupt causes packet to be copied from network card
    - Other processing on the packet should be deferred until its time comes
  - The deferred portion of interrupt processing is called the “Bottom Half”
soft_irq's

- 32 handlers that must be statically defined in the Linux kernel.
- A hardware interrupt (before returning) uses `raise_softirq()` to mark that a given soft_irq must execute the bottom half
- At a later time, when scheduling permits, the marked soft_irq handler is executed
  - When a hardware interrupt is finished
  - When a process makes a system call
  - When a new process is scheduled
soft_irq types (Linux)

- HI_SOFTIRQ
- TIMER_SOFTIRQ
- NET_TX_SOFTIRQ
- NET_RX_SOFTIRQ
- BLOCK_SOFTIRQ
- TASKLET_SOFTIRQ
- SCHED_SOFTIRQ
- ...

CS 423: Operating Systems Design
soft_irq types

- HI_SOFTIRQ
- TIMER_SOFTIRQ
- NET_TX_SOFTIRQ
- NET_RX_SOFTIRQ
- BLOCK_SOFTIRQ
- TASKLET_SOFTIRQ
- SCHED_SOFTIRQ
- ...

CS 423: Operating Systems Design
Task let

- Bottom halves multiplexed on top of soft_irq’s
- Scheduled using
  - tasklet_schedule()
  - tasklet_hi_schedule()
- Same tasklet invocations are serialized
- Tasklets can be created or removed dynamically
- Cannot sleep (cannot save their context)
Work Queues

- Work deferred to its own thread
- Can be scheduled together with other threads according to priorities set by a scheduling policy
- Associated with its thread control block and hence can block (and save context)
  - DECLARE_WORK(name, void (*func)(void *), void *data);
  - INIT_WORK(struct work_struct *work, void (*func)(void *), void *data);
  - schedule_work(&work);
Work Queue Examples

- Allocate the work queue data structures:
  - struct workqueue_struct *create_workqueue(const char *name);
  - struct workqueue_struct *create_singlethread_workqueue(const char *name);

- Create new work:
  - INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);

- Submit the work to the workqueue:
  - int queue_work(struct workqueue_struct *queue, struct work_struct *work);

For more info see
http://www.makelinux.net/ldd3/chp-7-sect-6
What is the size of the stack in the interrupt context??
What is the size of the stack in the interrupt context??

- Answer: It is a configuration option.
- Typically 2 page sizes (shared across all interrupts)
  - 8 KB on 16 bit architectures
  - 16 KB on 32 bit architectures
- In Linux 2.6 and beyond
  - 4 KB per interrupt handler