Tutorial 3
- Linux Interrupt Handling -

Daniel Kats
(with lots of help from Bogdan)

Today’s tutorial

- Getting started with Linux programming
- Interrupt need and interrupt types
- Hardware support for interrupts
- Interrupt handling process
- Upper and bottom halves
- Concurrency considerations
- Implementing an interrupt handler

Getting started

- Linux Kernel Development, by Robert Love
  - High-level, good starting point
- Understanding the Linux Kernel, by D.Bovet and M.Cesati
  - More advanced, lots of details
- Linux Device Drivers, A.Rubini and J.Corbet
- Cross-reference Linux sources – with hyperlinks!
  - http://lxr.linux.no
  - Really useful to understand code and data structures

Interrupts

- An event external to the currently executing process that causes a change in the normal flow of instruction execution; usually generated by hardware devices external to the CPU
- Asynchronous w.r.t current process
- External & internal devices need CPU service
- CPU must detect devices that require attention!

Alternatives

- Polling: CPU checks each device periodically
  - Too much overhead - CPU time wasted polling
  - Efficient if events arrive fast, or if not urgent (slow polling at large intervals)
- Interrupts: Each device gets an “interrupt line”
  - Device signals CPU when it needs attention, CPU handles request when it comes in
  - No overhead / wasted cycles
  - Good for events that are urgent, and/or infrequent

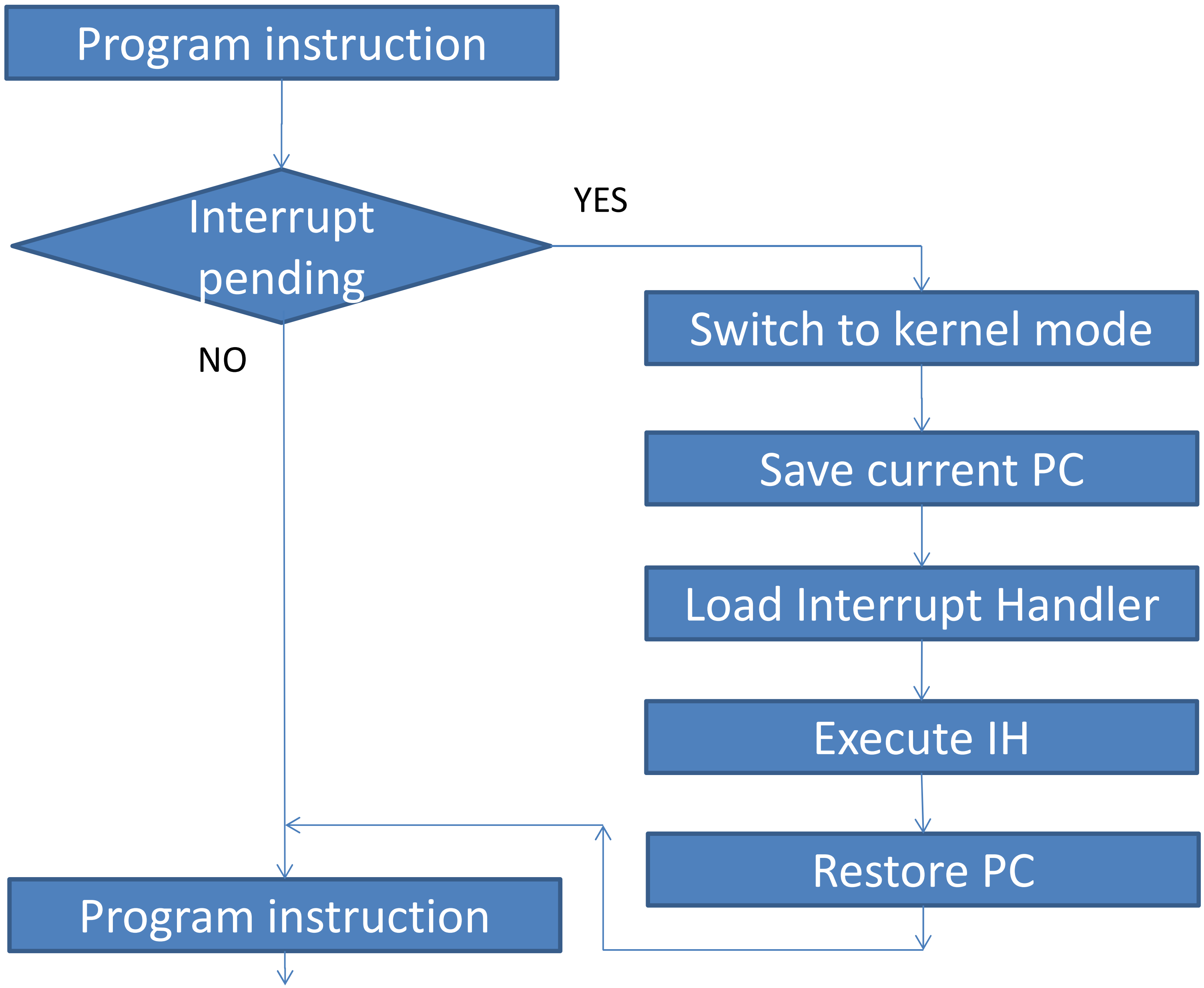
Interrupt types

- Hardware: An event/electronic signal from external device that needs CPU attention
  - Mouse moved, keyboard pressed
  - Printer ready, modem, etc
- Software:
  - exceptions (traps) in the processor: divide by zero exception, page faults, etc.
  - special software interrupt instructions (e.g., request disk reads/writes to disk controller)

Hardware support for interrupts

- Devices are connected to a shared message bus, which connects to the APIC
  - Asynchronous Programmable Interrupt Controller
  - Local APIC (LAPIC) located on processor
- Limited number of IRQ lines
- After every instruction (user-mode), CPU checks for hardware interrupt signals and, if present, calls an interrupt handler (kernel routine)

Interrupt handling



Interrupt Descriptor Table

- x86 implementation of IVT for fast interrupt handling
- Reserved chunk of RAM, used by the CPU to quickly branch to a specific interrupt handler
- Mapped to kernel space at 0x0000-0x03ff (256 4-byte pointers) on 8086
- Later CPUs: flexible locations and different size
- First 32 entries (0x00-0x1F) - reserved for mapping handlers for CPU-specific exceptions (faults)
- Next entries – interrupt routines (e.g., keyboard)

Interrupt handlers

- Fast/Hard/First-Level Interrupt Handler (FLIH)
  - Quickly service an interrupt, minimal exec time
  - Schedule SLIHs if needed
- Slow/Soft/Second-Level Interrupt Handler (SLIH)
  - Long-lived interrupt processing tasks
  - Lower priority - sit in a task runqueue
  - Executed by a pool of kernel threads, when no FLIHs
- Linux:
  - FLIHs = upper halves (UH)
  - SLIHs = bottom halves (BH)
- Windows
  - Deferred Procedure Calls (DPCs)

Bottom halves (BH)

- SoftIRQs and Tasklets
  - Deferred work runs in interrupt context
  - Don’t run in process context
  - Can’t sleep
  - Tasklets somewhat easier to use
- Workqueues
  - Run in kernel threads
  - Schedulable
  - Can sleep

Concurrency

- Hardware interrupts (IRQs) can arrive while a specific interrupt handler is in execution
- Fast interrupts must run atomically => Disable all interrupts and restore them when done
- As a result, fast interrupts must run fast, and defer long-lived work to bottom halves.
- Otherwise => interrupt storm => livelocks

Interrupt enabling

- IE (Interrupt Enable) bit in the status register can be set or reset by the processor
- *cli* = clear interrupts
- *sti* = set interrupts
- Must be careful with semantics if using these directly
  - `cli` disables interrupts on ALL processors
  - If you are already handling an IRQ, `cli` only disables them on current CPU

Multiprocessors

- Linux kernel tries to divide interrupts evenly across processors to some extent
- Fast interrupts (SA_INTERRUPT) execute with all other interrupts disabled on the current processor
- Other processors can still handle interrupts, though not the same IRQ at the same time

Interrupt handling internals (x86)

- Each interrupt goes through `do_IRQ`
- A `do_IRQ` acquires spinlock on the irq#, preventing other CPUs from handling this IRQ
- Looks up handler
  - If no handler, schedule bottom halves (if any) and return
  - If handler, run `handle_IRQ_event` to invoke the handlers

Implementing an interrupt handler

- Use `request_irq()`to get interrupt handler
  - irq (IRQ number)
  - handler (func. pointer - interrupt handler)
  - flags (SA_INTERRUPT, SA_SHIRQ, etc)
  - dev_name (string used in /proc/interrupts)
  - dev_id (used for shared interrupt lines)
- Fast handler - always with SA_INTERRUPT
- From within interrupt handler, schedule BH to run (`tasklet_schedule`, `queue_work`, etc.)

Implementing an interrupt handler(2)

- A driver might need to disable/enable interrupt reporting for its own IRQ line only
- Kernel functions:
  - `disable_irq (int irq)`
  - `disable_irq_nosync (int irq)`
  - `enable_irq (int irq)`
- Enable/disable IRQ - across ALL processors
- Nosync – doesn’t wait for currently executing IH’s to complete => faster, but leaves driver open to race conditions

Useful readings

- Linux Device Drivers, 3rd edition
  - http://lwn.net/Kernel/LDD3/
  - Chapter 10 – Interrupt Handling
- The Linux Kernel Module Programming guide
  - http://www.tdlp.org/LDP/lkmpg/2.6/html/
  - Chapter 12 – Interrupt Handlers
- Understanding the Linux Kernel
  - Chapter 4 – Interrupts and Exceptions
- Consult LXR – Deep understanding of Linux source code and data structures involved