Interrupts in Linux

COMS W4118

Prof. Kaustubh R. Joshi

krj@cs.columbia.edu

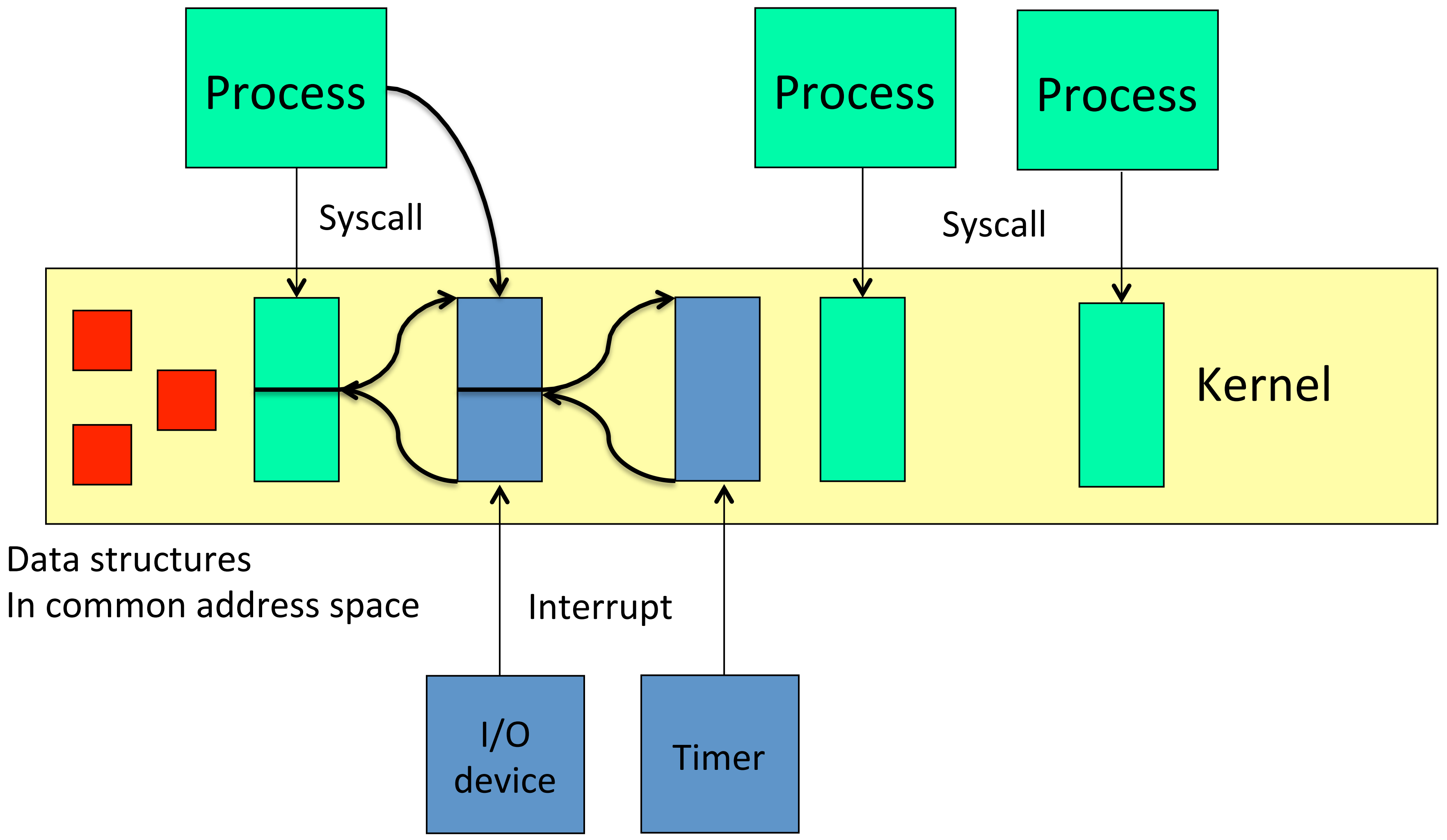
http://www.cs.columbia.edu/~krj/os

References: Operating Sys Concepts 9e, Understanding the Linux Kernel, previous W4118s

Why Interrupts?

- Devices require a prompt response from the CPU when various events occur, even when the CPU is busy running a program
- Need a mechanism for a device to “gain CPU’s attention”

The Kernel as a Multithreaded Server



Overview

- Interrupts and Exceptions
- Exception Types and Handling
- Interrupt Request Lines (IRQs)
- Programmable Interrupt Controllers (PIC)
- Interrupt Descriptor Table (IDT)
- Interrupt Handling
- SoftIRQs, Tasklets
- Work Queues

Interrupts

- Forcibly change normal flow of control
- Similar to context switch (but lighter weight)
  - Hardware saves some context on stack; Includes interrupted instruction if restart needed
  - Enters kernel at a specific point; kernel then figures out which *interrupt handler* should run
  - Execution resumes with special "iret" instruction
- Many different types of interrupts

Types of Interrupts (x86 terminology)

- Asynchronous
  - From external source, such as I/O device
  - Not related to instruction being executed
- Synchronous (also called *exceptions*)
  - *Processor-detected* exceptions:
    - *Faults* — correctable; offending instruction is *retried*
    - *Traps* — often for debugging; instruction is *not* retried
    - *Aborts* — major error (hardware failure)
  - *Programmed* exceptions:
    - Requests for kernel intervention (software intr/syscalls)

Faults

- Instruction would be illegal to execute
- Examples:
  - Writing to a memory segment marked ‘read-only’
  - Reading from an unavailable memory segment (on disk)
  - Executing a ‘privileged’ instruction
- Detected *before* incrementing the IP
- The causes of ‘faults’ can often be ‘fixed’
- If a ‘problem’ can be remedied, then the CPU can just resume its execution-cycle

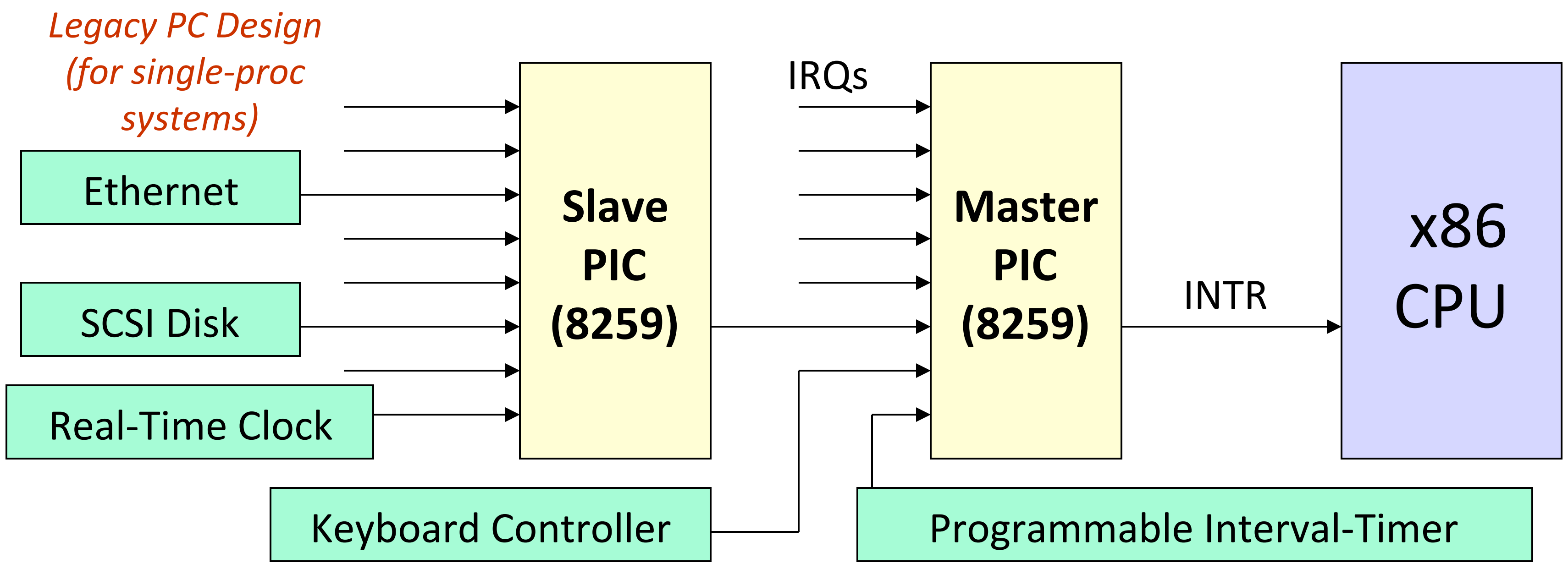
Traps

- A CPU might have been programmed to automatically switch control to a 'debugger' program after it has executed an instruction
- That type of situation is known as a 'trap'
- It is activated *after* incrementing the IP

Error Exceptions

- Most error exceptions — divide by zero, invalid operation, illegal memory reference, etc. — translate directly into signals
- This isn't a coincidence. . .
- The kernel's job is fairly simple: send the appropriate signal to the current process
  - **force_sig(sig_number, current);**
- That will probably kill the process, but that's not the concern of the exception handler
- One important exception: page fault
- An exception can (infrequently) happen in the kernel
  - **die(); // kernel oops**

Interrupt Hardware



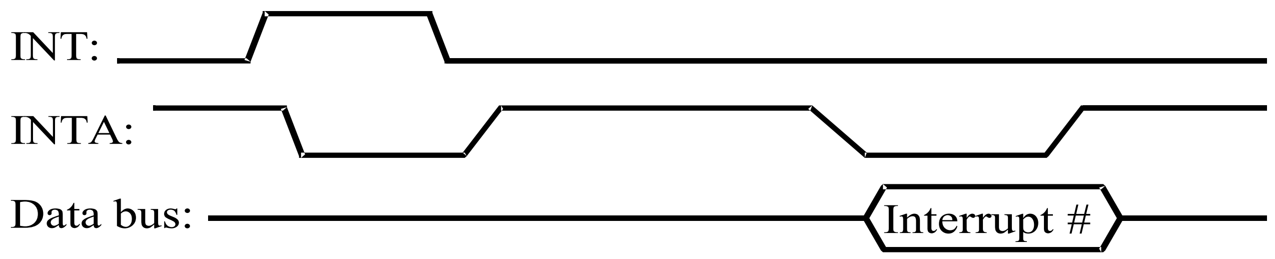
- I/O devices have (unique or shared) *Interrupt Request Lines* (IRQs)
- IRQs are mapped by special hardware to *interrupt vectors*, and passed to the CPU
- This hardware is called a *Programmable Interrupt Controller* (PIC)

The `Interrupt Controller’

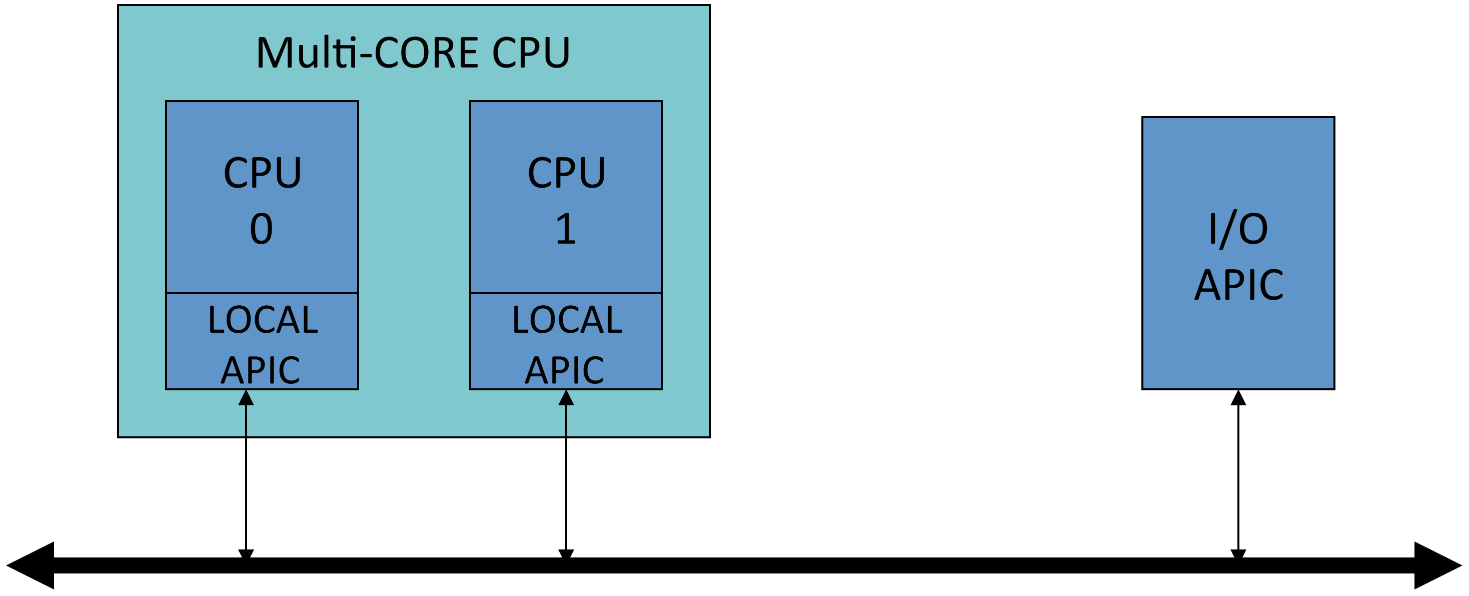
- Responsible for telling the CPU when a specific external device wishes to ‘interrupt’
  - Needs to tell the CPU *which* one among several devices is the one needing service
- PIC translates IRQ to *vector*
  - Raises interrupt to CPU
  - Vector available in register
  - Waits for ack from CPU
- Interrupts can have varying priorities
  - PIC also needs to prioritize multiple requests
- Possible to “mask” (disable) interrupts at PIC or CPU
- Early systems cascaded two 8 input chips (8259A)

Example: Interrupts on 80386

- 80386 core has one interrupt line, one interrupt acknowledge line
- Interrupt sequence:
  - Interrupt controller raises INT line
  - 80386 core pulses INTA line low, allowing INT to go low
  - 80386 core pulses INTA line low again, signaling controller to put interrupt number on data bus



Multiple Logical Processors



Advanced Programmable Interrupt Controller is needed to perform ‘routing’ of I/O requests from peripherals to CPUs

(The legacy PICs are masked when the APICs are enabled)

APIC, IO-APIC, LAPIC

- Advanced PIC (APIC) for SMP systems
  - Used in all modern systems
  - Interrupts "routed" to CPU over system bus
  - IPI: inter-processor interrupt
- Local APIC (LAPIC) versus "frontend" IO-APIC
  - Devices connect to front-end IO-APIC
  - IO-APIC communicates (over bus) with Local APIC
- Interrupt routing
  - Allows broadcast or selective routing of interrupts
  - Ability to distribute interrupt handling load
  - Routes to lowest priority process
    - Special register: Task Priority Register (TPR)
  - Arbitrates (round-robin) if equal priority

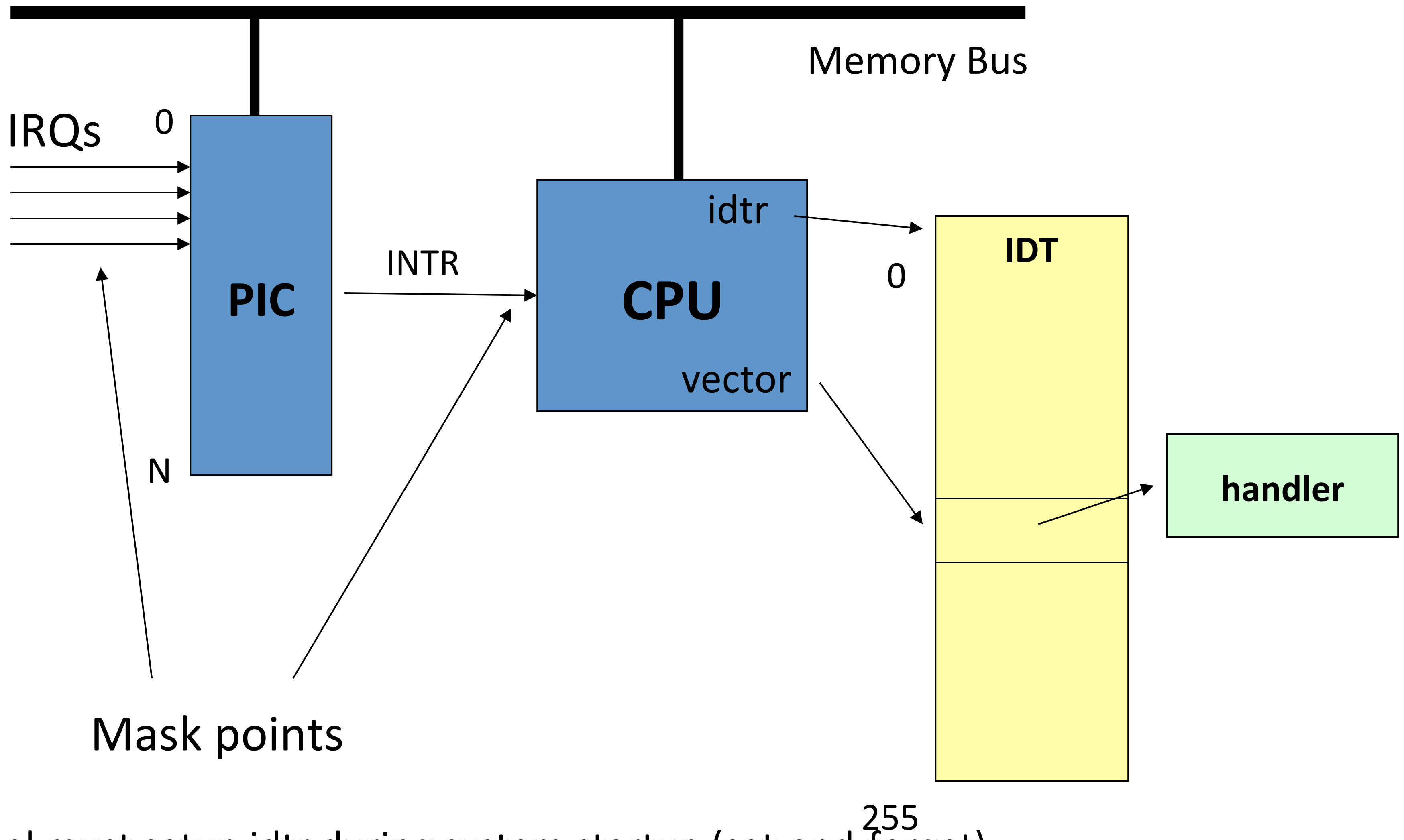
Assigning IRQs to Devices

- IRQ assignment is hardware-dependent
- Sometimes it's hardwired, sometimes it's set physically, sometimes it's programmable
- PCI bus usually assigns IRQs at boot
- Some IRQs are fixed by the architecture
  - IRQ0: Interval timer
  - IRQ2: Cascade pin for 8259A
- Linux device drivers request IRQs when the device is opened
- Note: especially useful for dynamically-loaded drivers, such as for USB or PCMCIA devices
- Two devices that aren't used at the same time can share an IRQ, even if the hardware doesn't support simultaneous sharing

Assigning Vectors to IRQs

- Vector: index (0-255) into interrupt descriptor table
- Vectors usually IRQ# + 32
  - Below 32 reserved for non-maskable intr & exceptions
  - Maskable interrupts can be assigned as needed
  - Vector 128 used for syscall
  - Vectors 251-255 used for IPI

x86 Interrupt Handling via IDT



Kernel must setup idtr during system startup (set-and-forget)
LIDT and SIDT used to set/get the pointer to this table

Interrupt Descriptor Table

- The entry-point to the interrupt-handler is located via the Interrupt Descriptor Table (IDT)
- IDT: gate descriptors, one per vector
  - Address of handler
  - Current Privilege Level (CPL)
  - Descriptor Privilege Level (DPL)
  - Gates (slightly different ways of entering kernel)
    - **Task gate**: includes TSS to transfer to (not used by Linux)
    - **Interrupt gate**: disables further interrupts
    - **Trap gate**: further interrupts still allowed

IDT Initialization

- Initialized once by BIOS in real mode
  - Linux re-initializes during kernel init
- Must not expose kernel to user mode access
  - start by zeroing all descriptors
- Linux lingo:
  - Interrupt gate (same as Intel; no user access)
    - Not accessible from user mode
  - System gate (Intel trap gate; user access)
    - Used for int, int3, into, bounds
  - Trap gate (same as Intel; no user access)
    - Used for exceptions

Dispatching Interrupts

- On entry hardware:
  - Checks which vector?
  - Get corresponding descriptor in IDT
  - Find specified descriptor in GDT (for handler)
  - Check privilege levels (CPL, DPL)
    - If entering kernel mode, set kernel stack
  - Save eflags, cs, (original) eip on stack
- Jump to appropriate handler
  - Assembly code prepares C stack, calls handler
- On return (i.e. iret):
  - Restore registers from stack
  - If returning to user mode, restore user stack
  - Clear segment registers (if privileged selectors)

Interrupt Masking

- Two different types: global and per-IRQ
- Global — delays all interrupts
- Selective — individual IRQs can be masked selectively
- Selective masking is usually what's needed — interference most common from two interrupts of the same type

Nested Interrupts

- What if a second interrupt occurs while an interrupt routine is excuting?
- Generally a good thing to permit that — is it possible?
- And why is it a good thing?

Maximizing Parallelism

- You want to keep all I/O devices as busy as possible
- In general, an I/O interrupt represents the end of an operation; another request should be issued as soon as possible
- Most devices don’t interfere with each others’ data structures; there’s no reason to block out other devices

Handling Nested Interrupts

- As soon as possible, unmask the global interrupt
- As soon as reasonable, re-enable interrupts from that IRQ
- But that isn't always a great idea, since it could cause re-entry to the same handler
- IRQ-specific mask is not enabled during interrupt-handling

Nested Execution

- Interrupts can be interrupted
  - By different interrupts; handlers need not be reentrant
  - No notion of priority in Linux
  - Small portions execute with interrupts disabled
  - Interrupts remain pending until acked by CPU
- Exceptions can be interrupted
  - By interrupts (devices needing service)
- Exceptions can nest two levels deep
  - Exceptions indicate coding error
  - Exception code (kernel code) shouldn't have bugs
  - Page fault is possible (trying to touch user data)

First-Level Interrupt Handler

- Often in assembler
- Perform minimal, common functions: saving registers, unmasking other interrupts
- Eventually, undoes that: restores registers, returns to previous context
- Most important: call proper second-level interrupt handler (C program)

Interrupt Handling

- Do as little as possible in the interrupt handler
- Defer non-critical actions till later
- Three types of actions:
  - Critical: Top-half (interrupts disabled – briefly!)
    - Example: acknowledge interrupt
  - Non-critical: Top-half (interrupts enabled)
    - Example: read key scan code, add to buffer
  - Non-critical deferrable: Bottom half, do it "later" (interrupts enabled)
    - Example: copy keyboard buffer to terminal handler process
    - Softirqs, tasklets
- **No process context available**

No Process Context

- Interrupts (as opposed to exceptions) are not associated with particular instructions
- They' re also not associated with a given process
- The currently-running process, at the time of the interrupt, as no relationship whatsoever to that interrupt
- Interrupt handlers cannot refer to **current**
- Interrupt handlers cannot sleep!

Interrupt Stack

- When an interrupt occurs, what stack is used?
  - Exceptions: The *kernel stack* of the current process, whatever it is, is used (There's always some process running — the "idle" process, if nothing else)
  - Interrupts: hard IRQ stack (1 per processor)
  - SoftIRQs: soft IRQ stack (1 per processor)
- These stacks are configured in the IDT and TSS at boot time by the kernel

Finding the Proper Handler

- On modern hardware, multiple I/O devices can share a single IRQ and hence interrupt vector
- First differentiator is the interrupt *vector*
- Multiple *interrupt service routines* (ISR) can be associated with a vector
- Each device’s ISR for that IRQ is called; the determination of whether or not that device has interrupted is device-dependent

Deferrable Work

- We don't want to do too much in regular interrupt handlers:
  - Interrupts are masked
  - We don't want the kernel stack to grow too much
- Instead, interrupt handlers schedule work to be performed later
- Three deferred work mechanisms: *softirqs*, *tasklets*, and *work queues*
- Tasklets are built on top of softirqs
- For all of these, requests are queued

Softirqs

- Statically allocated: specified at kernel compile time
- Limited number:

Priority	*Type*
0	High-priority tasklets
1	Timer interrupts
2	Network transmission
3	Network reception
4	SCSI disks
5	Regular tasklets

Running Softirqs

- Run at various points by the kernel
- Most important: after handling IRQs and after timer interrupts
- Softirq routines can be executed simultaneously on multiple CPUs:
  - Code must be re-entrant
  - Code must do its own locking as needed

Rescheduling Softirqs

- A softirq routine can reschedule itself
- This could starve user-level processes
- Softirq scheduler only runs a limited number of requests at a time
- The rest are executed by a kernel thread, `ksoftirqd`, which competes with user processes for CPU time

Tasklets

- Similar to softirqs
- Created and destroyed dynamically
- Individual tasklets are locked during execution; no problem about re-entrancy, and no need for locking by the code
- Only one instance of tasklet can run, even with multiple CPUs
- Preferred mechanism for most deferred activity

Work Queues

- Always run by kernel threads
- Softirqs and tasklets run in an interrupt context; work queues have a process context
- Because they have a process context, they can sleep
- However, they' re kernel-only; there is no user mode associated with it

SoftIRQs vs. Tasklet vs. WQ

	ISR	SoftIRQ	Tasklet	WorkQueue	
Will disable all interrupts?	Briefly	No	No	No	
Will disable other instances of self?	Yes	Yes	No	No	
Higher priority than regular scheduled tasks?	Yes	Yes*	Yes*	No	
Will be run on same processor as ISR?	N/A	Yes	Maybe	Maybe	
More than one run can on same CPU?	No	No	No	Yes	
Same one can run on multiple CPUs?	Yes	Yes	No	Yes	
Full context switch?	No	No	No	Yes	
Can sleep? (Has own kernel stack)	No	No	No	Yes	
Can access user space?	No	No	No	No	

*Within limits, can be run by ksoftirqd

Return Code Path

- Interleaved assembly entry points:
  - ret_from_exception()
  - ret_from_intr()
  - ret_from_sys_call()
  - ret_from_fork()
- Things that happen:
  - Run scheduler if necessary
  - Return to user mode if no nested handlers
    - Restore context, user-stack, switch mode
    - Re-enable interrupts if necessary
  - Deliver pending signals

Demo: /proc/interrupts

```
$ cat /proc/interrupts
         CPU0
  0:  865119901        IO-APIC-edge   timer
  1:         4         IO-APIC-edge   keyboard
  2:         0         XT-PIC         cascade
  8:         1         IO-APIC-edge   rtc
 12:        20         IO-APIC-edge   PS/2 Mouse
 14:   6532494         IO-APIC-edge   ide0
 15:        34         IO-APIC-edge   ide1
 16:         0         IO-APIC-level  usb-uhci
 19:         0         IO-APIC-level  usb-uhci
 23:         0         IO-APIC-level  ehci-hcd
 32:        40         IO-APIC-level  ioc0
 33:        40         IO-APIC-level  ioc1
 48:  273306628        IO-APIC-level  eth0
NMI:         0
ERR:         0
```

- Columns: IRQ, count, interrupt controller, devices