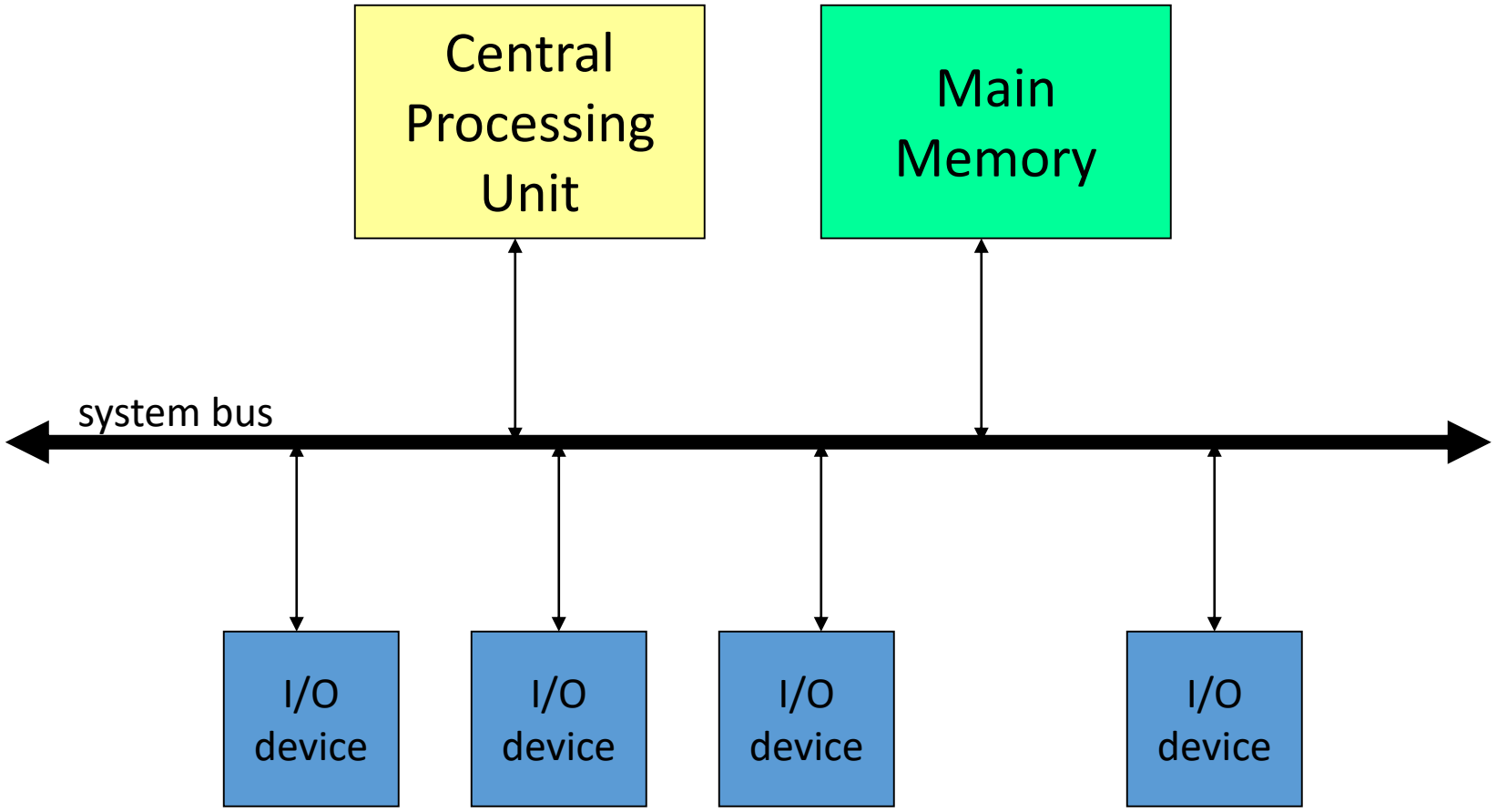
Lecture 6

Interrupts

The Nuts and Bolts of Responding to Events

- Handling interrupts require both hardware and software cooperation
- Hardware Portion
  - How x86 handles hardware interrupts
- Software Portion
  - How a kernel handles interrupts

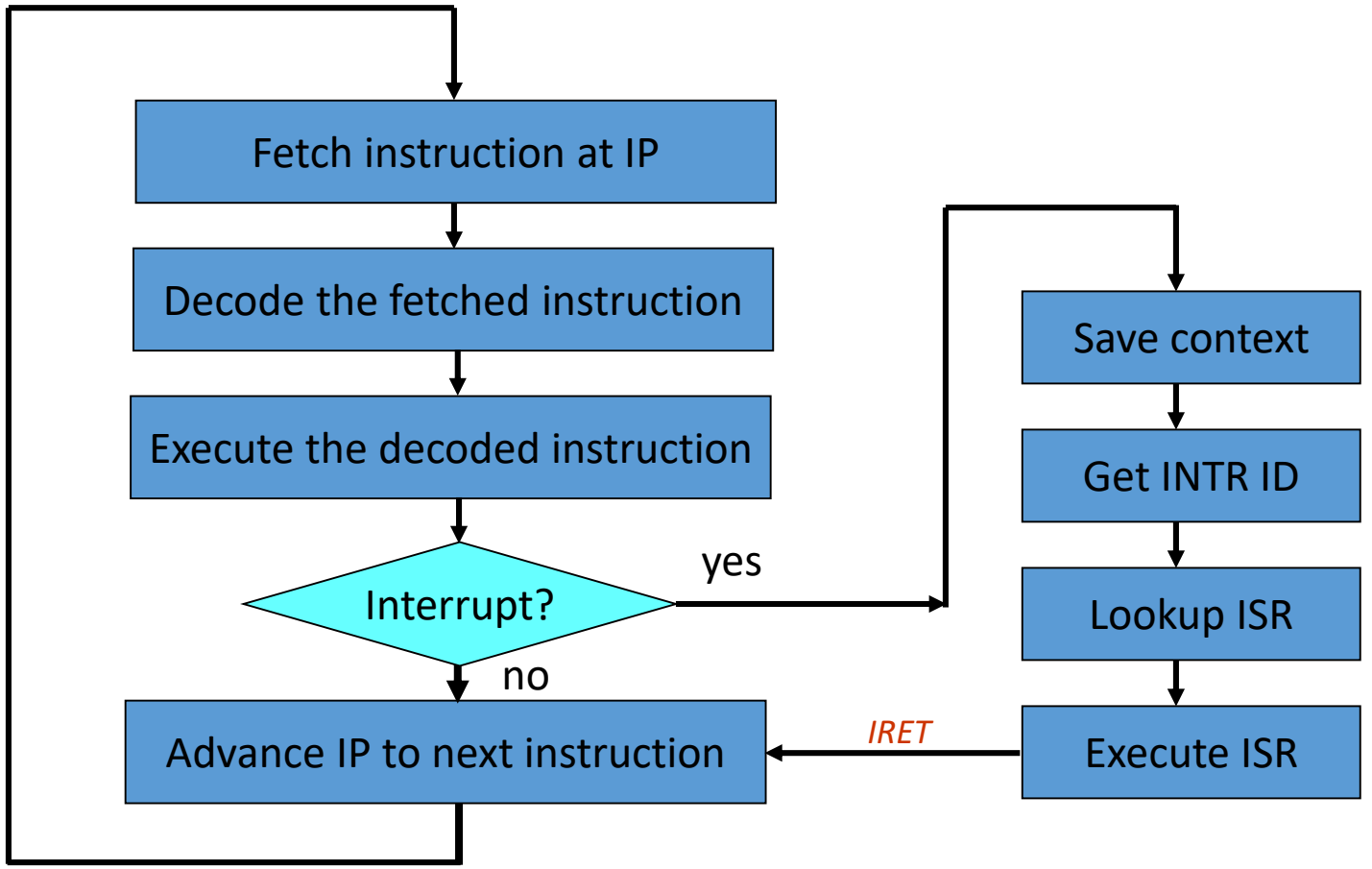
The General Computing Environment



Why interrupts?

- I/O an integral part of any computing system
  - Disk drives, keyboard or other human interface, etc.
- Apps and users expect responsiveness
- The unexpected can arise
  - Errors, faults, exceptions

Instruction Processing Cycle



What is an interrupt?

- A forced change in the flow of control
- Hardware performs simple context switching
- Kernel entered at a specific predetermined point
- Normal execution resumes with a special “iret” instruction that also restores the context
- Many different types for different purposes

Types of Interrupts

- Asynchronous
  - Source is external such as I/O device
  - Not related to instruction currently being executed
- Synchronous (also called *exceptions*)
  - *Processor-detected* exceptions:
    - *Faults* — offending instruction is *retried* (eg. page faults)
    - *Traps* — instruction is *not* retried
    - *Aborts* — major error (hardware failure)
  - *Programmed* exceptions:
    - Requests for kernel intervention (eg. syscalls)

Faults

- Normal execution cannot continue
- Examples:
  - Writing to a memory segment marked ‘read-only’
  - Reading from an unavailable memory segment (on disk)
  - Executing a ‘privileged’ instruction
- The causes of ‘faults’ can often be ‘fixed’
- If the issue can be resolved, then the CPU can just restart its execution-cycle

Related: instruction replay

- Modern processors have deep pipeline and superscalar execution
- Corollary: aggressive instruction issue means that an instruction may be issued before it is known to be safe to do so
  - Problem: deep into an instruction's execution, dependency or exception arise
- Since Pentium 4, Intel has implemented a hardware replay queue
  - Faulting instructions will be placed in a queue to be replayed

Traps

- Example: A CPU might have been programmed to automatically switch control to a ‘debugger’ program at specific program point
  - `INT 3` (opcode 0xCC) used by debuggers on x86

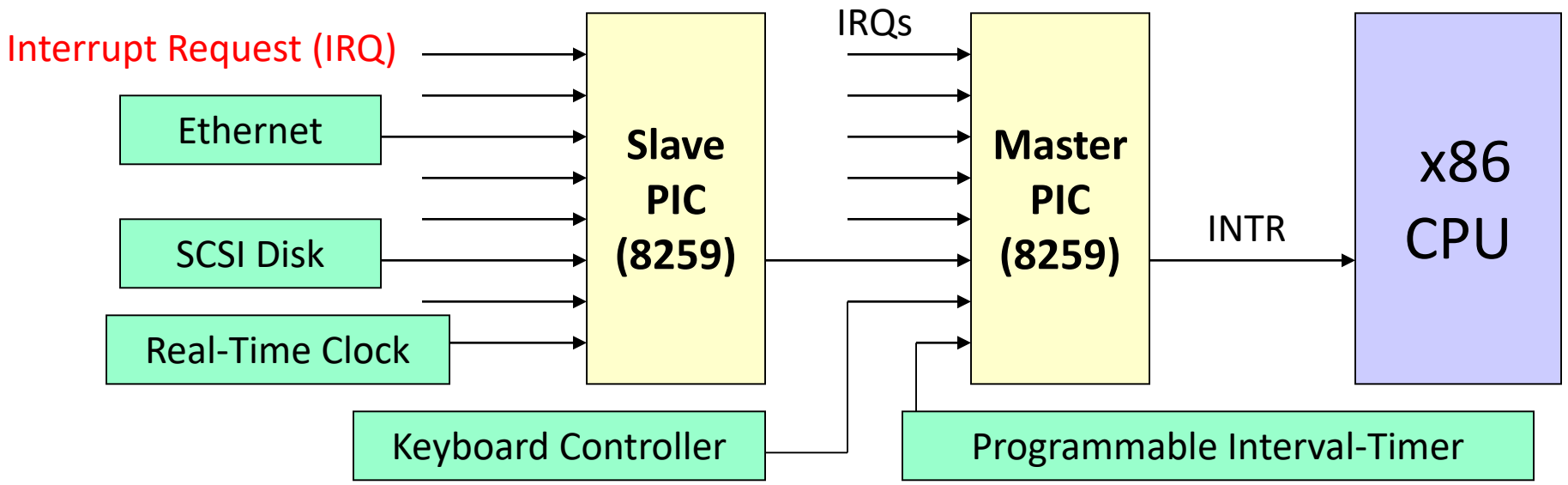
- This is known as a ‘trap’
  - Coz it is deliberately set

Error Exceptions

- Most error exceptions are translate directly into signals
  - Divide by zero
  - Privileged instruction
  - Illegal memory referencing, etc.
- The kernel's job is fairly simple: send the appropriate signal to the current process
  - `force_sig(sig_number, current);`
- Most of the time, the process by default gets killed
  - 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: The Hardware Story

In the good old days…



Legacy PC Design

Intel 8259 Programmable Interrupt Controller

- Expands on the original interrupt capability of the 8085 processor
- Can accept 8 interrupt requests, allowing them one by one to the processor INTR pin
- Programmable priorities of interrupts
- Can be cascaded to allow up to 64 interrupts

How it worked - 1

- First, it has to be initialized
- When ready, it receives interrupt through IR0 to IR7
- Checks if an interrupt is masked or not, and what is its priority
- If all ok (previous interrupt completed, current interrupt highest priority and not masked), send **`INT`** signal to **`INTR`** pin of 8085
- In response, processor sends *three* **`INTA`** signal

How it worked - 2

- Processor sends first **INTA** signal
- 8259 responds with x86 CALL opcode

- Processor sends second **INTA** signal
- 8259 responds with *low* byte of call address

- Processor sends third **INTA** signal
- 8259 responds with *high* byte of call address

- Processor saves current PC on stack, executed CALL instruction using the 16-bit address (the *interrupt servicing routine*) the 8259 sent

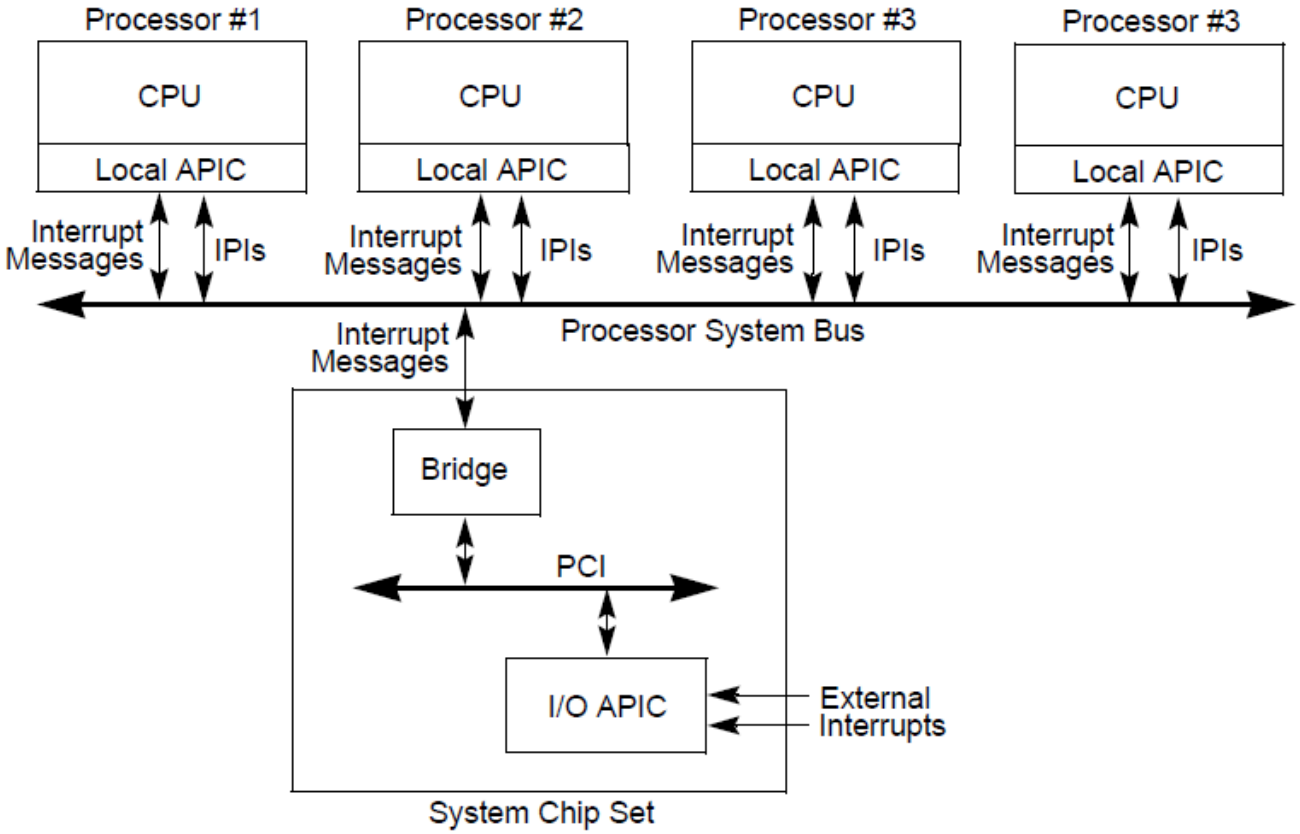
Advanced Programmable Interrupt Controller (APIC)

- 8259 cannot handle multiprocessor system
  - With more than one CPUs, send interrupt to who?
  - How to implement inter-processor interrupt (IPI)?
- PIC updated with the APIC – actually a system
- In APIC, each CPU consists of a “core” and a “local APIC” (LAPIC)
  - Contains a Local Vector Table
- In addition, there is an separate I/O APIC
  - Part of the chipset
  - If system has multi I/O subsystem, then multiple I/O APIC

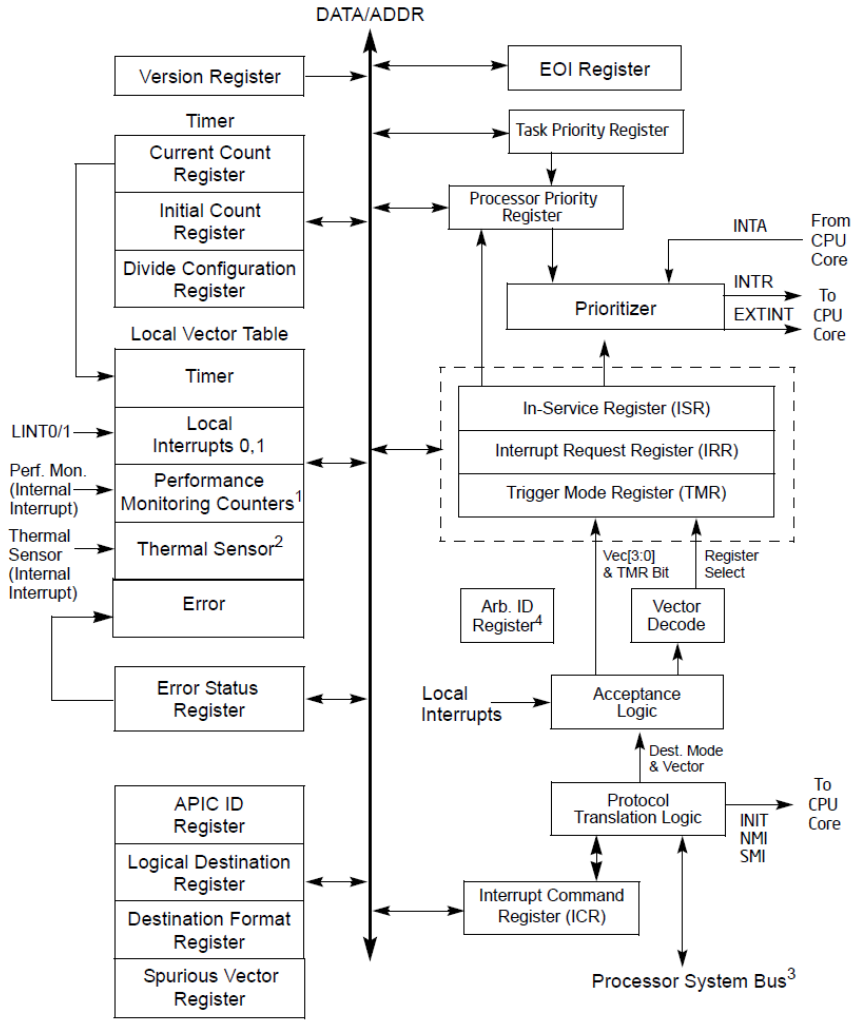
APIC, IO-APIC, LAPIC

- Advanced PIC (APIC) for multiprocessor systems
  - Used in all modern systems
  - Interrupts “routed” to CPU over system bus
  - Supports inter-processor interrupt (IPI)
- Local APIC (LAPIC) versus “frontend” IO-APIC
  - Devices connect to front-end IO-APIC (cascading)
  - 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

APIC system


Processor #1
Processor #2
Processor #3
Processor #3
CPU
CPU
CPU
CPU
Local APIC
Local APIC
Local APIC
Local APIC
Interrupt Messages
IPIs
Interrupt Messages
IPIs
Interrupt Messages
IPIs
Interrupt Messages
IPIs
Processor System Bus
Interrupt Messages
Bridge
PCI
I/O APIC
External Interrupts
System Chip Set

Local APIC



1. Introduced in P6 family processors.
2. Introduced in the Pentium 4 and Intel Xeon processors.
3. Three-wire APIC bus in P6 family and Pentium processors.
4. Not implemented in Pentium 4 and Intel Xeon processors.

Interrupts to Local APIC

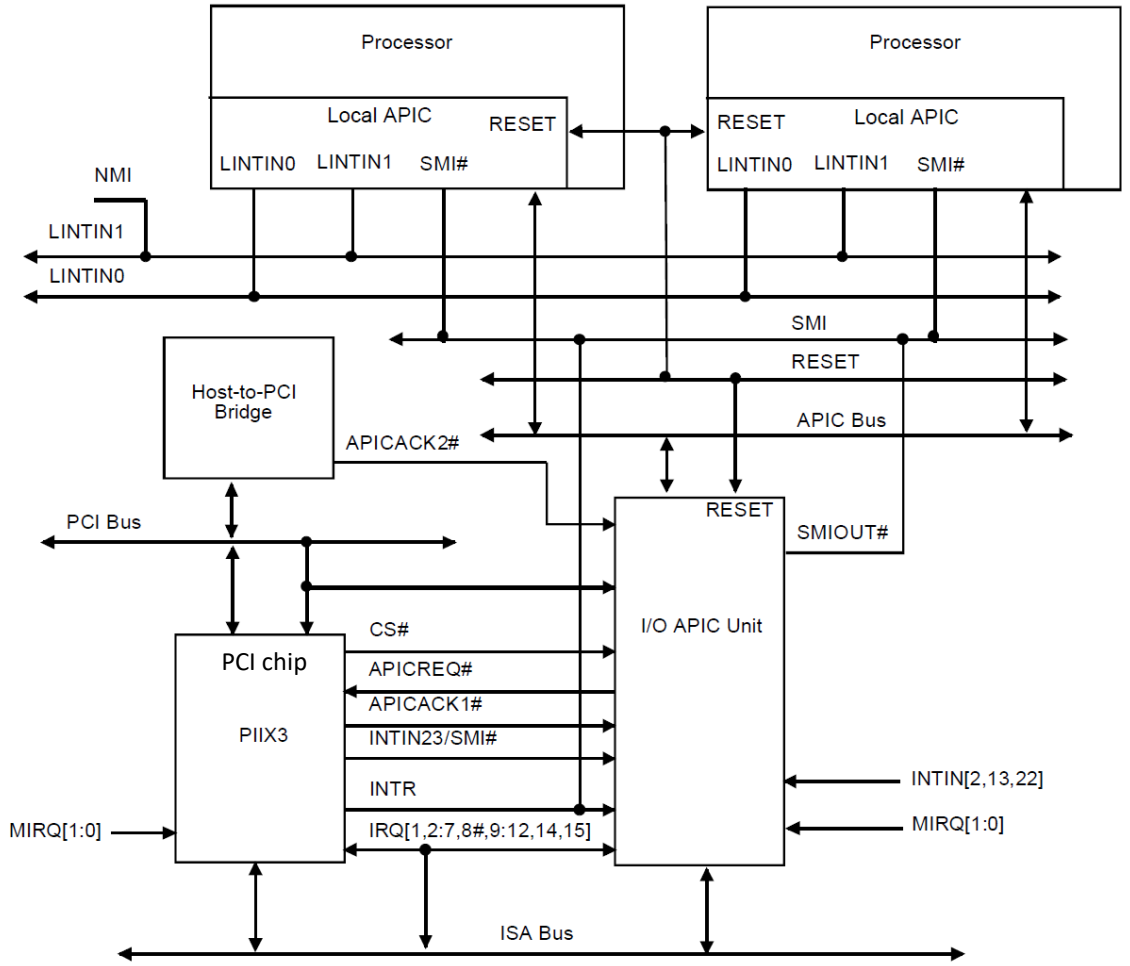
- Locally connected I/O devices — An I/O device that is connected directly to the processor's local interrupt pins (LINT0 and LINT1
- Externally connected I/O devices — Connected to the interrupt input pins of an I/O APIC
- Inter-processor interrupts (IPIs) — For software self-interrupts, interrupt forwarding, or preemptive scheduling.
- APIC timer generated interrupts
- Performance monitoring counter interrupts
- Thermal Sensor interrupts — When the internal thermal sensor has been tripped
- APIC internal error interrupts

Local Vector Table

- The processor's LINT0 and LINT1 pins, the APIC timer, the performance-monitoring counters, the thermal sensor, and the internal APIC error detector are referred to as local interrupt sources

- A (programmable) local vector table is looked up to translate these to system-wide interrupt vectors

I/O APIC



Intel 82093AA (IOAPIC)

I/O APIC

- Each IRQ associated with a 64 bit register

Field	Bits	Description
Vector	0 - 7	The Interrupt vector that will be raised on the specified CPU(s).
Delivery Mode	8 - 10	How the interrupt will be sent to the CPU(s). It can be 000 (Fixed), 001 (Lowest Priority), 010 (SMI), 100 (NMI), 101 (INIT) and 111 (ExtINT). Most of the cases you want Fixed mode, or Lowest Priority if you don't want to suspend a high priority task on some important Processor/Core/Thread.
Destination Mode	11	Specify how the Destination field shall be interpreted. 0: Physical Destination, 1: Logical Destination
Delivery Status	12	If 0, the IRQ is just relaxed and waiting for something to happen (or it has fired and already processed by Local APIC(s)). If 1, it means that the IRQ has been sent to the Local APICs but it's still waiting to be delivered.
Pin Polarity	13	0: Active high, 1: Active low. For ISA IRQs assume Active High unless otherwise specified in Interrupt Source Override descriptors of the MADT or in the MP Tables.
Remote IRR	14	**TODO**
Trigger Mode	15	0: Edge, 1: Level. For ISA IRQs assume Edge unless otherwise specified in Interrupt Source Override descriptors of the MADT or in the MP Tables.
Mask	16	Just like in the old PIC, you can temporary disable this IRQ by setting this bit, and reenable it by clearing the bit.
Destination	56 - 63	This field is interpreted according to the Destination Format bit. If Physical destination is choosen, then this field is limited to bits 56 - 59 (only 16 CPUs addressable). You put here the APIC ID of the CPU that you want to receive the interrupt. **TODO**: Logical destination format...

Intel Reserved Interrupts

Vector	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction[5]
20	#VE	Virtualization Exception	EPT violations[6]
21-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.

Intel Reserved

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
- Two devices that aren't used at the same time can share an IRQ, even if the hardware doesn't support simultaneous sharing

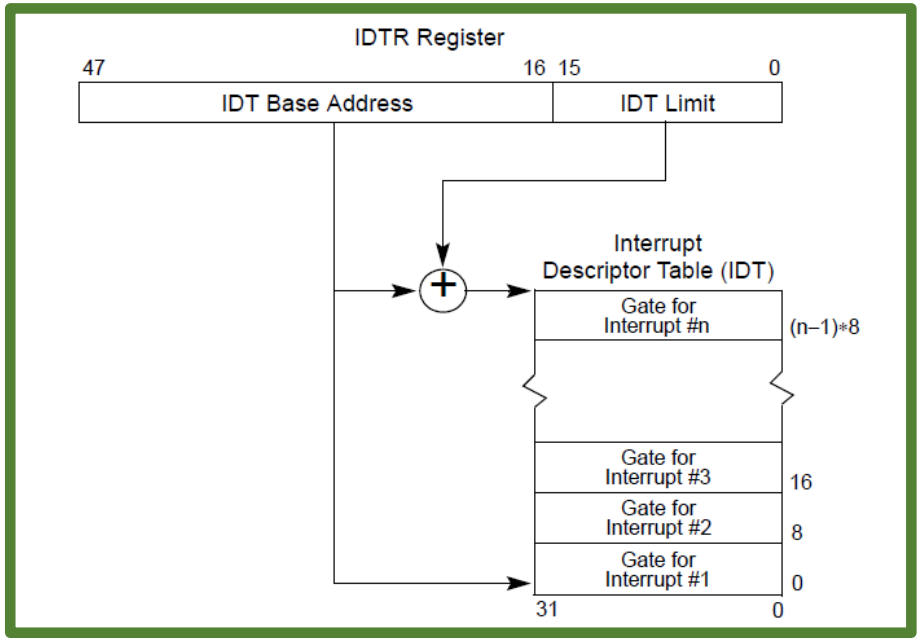
Assigning Vectors to IRQs

- The interrupt vector is an index (0-255) into the interrupt descriptor table
- Vectors usually IRQ# + 32
  - Below 32 reserved for non-maskable interrupt and exceptions
  - Maskable interrupts can be assigned as needed
  - Vector 128 used for Linux syscall
  - Vectors 251-255 used for IPI

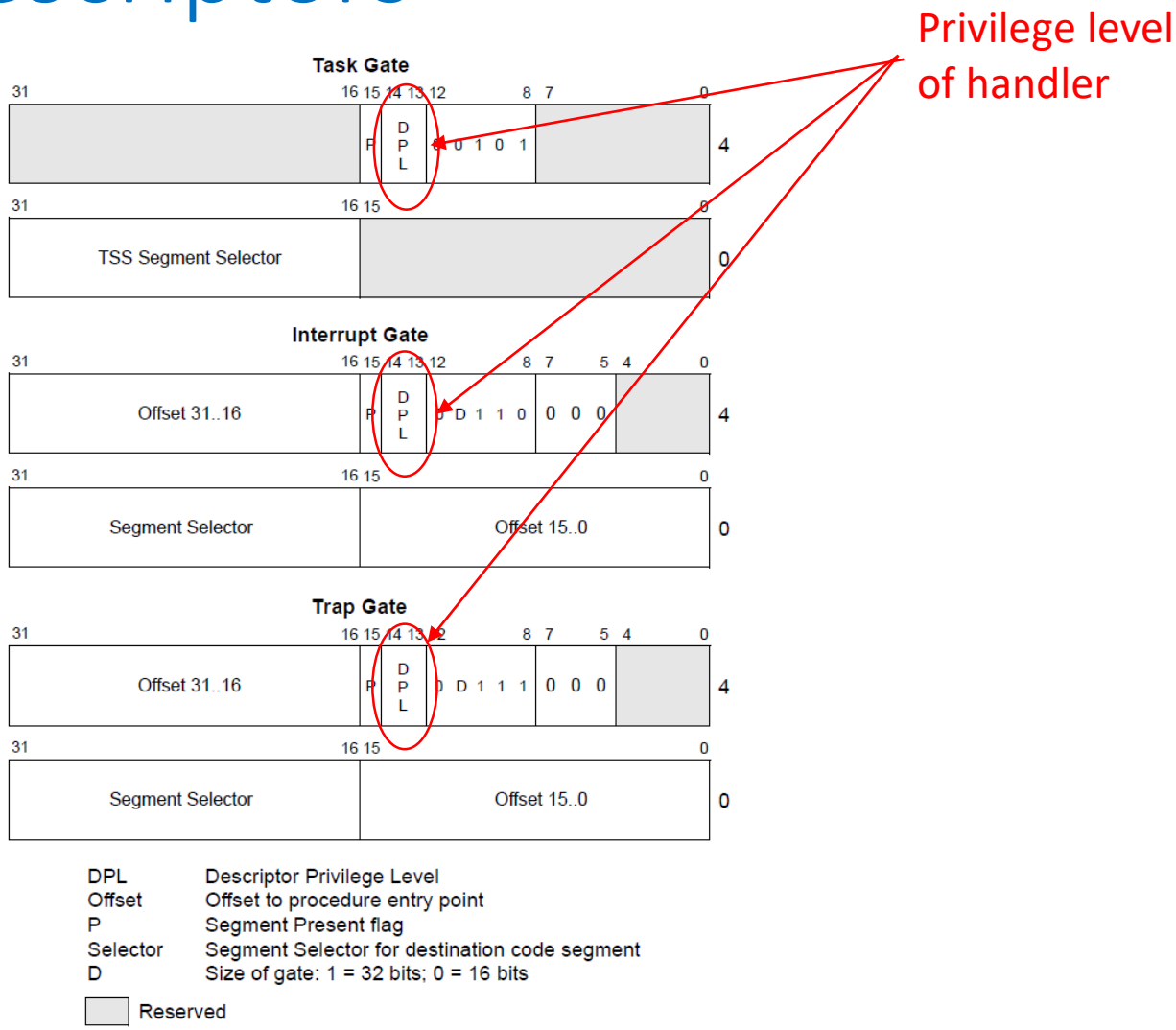
How x86 handles interrupts

Basic flow

- The Interrupt Descriptor Table Register (IDTR) points to a table (IDT) consisting of either:
  - Task-gate descriptor
  - Interrupt-gate descriptor
    - Disables further interrupts
  - Trap-gate descriptor
    - Further interrupts still allowed
- On an interrupt, the vector is used to look up the table and the corresponding actions are taken



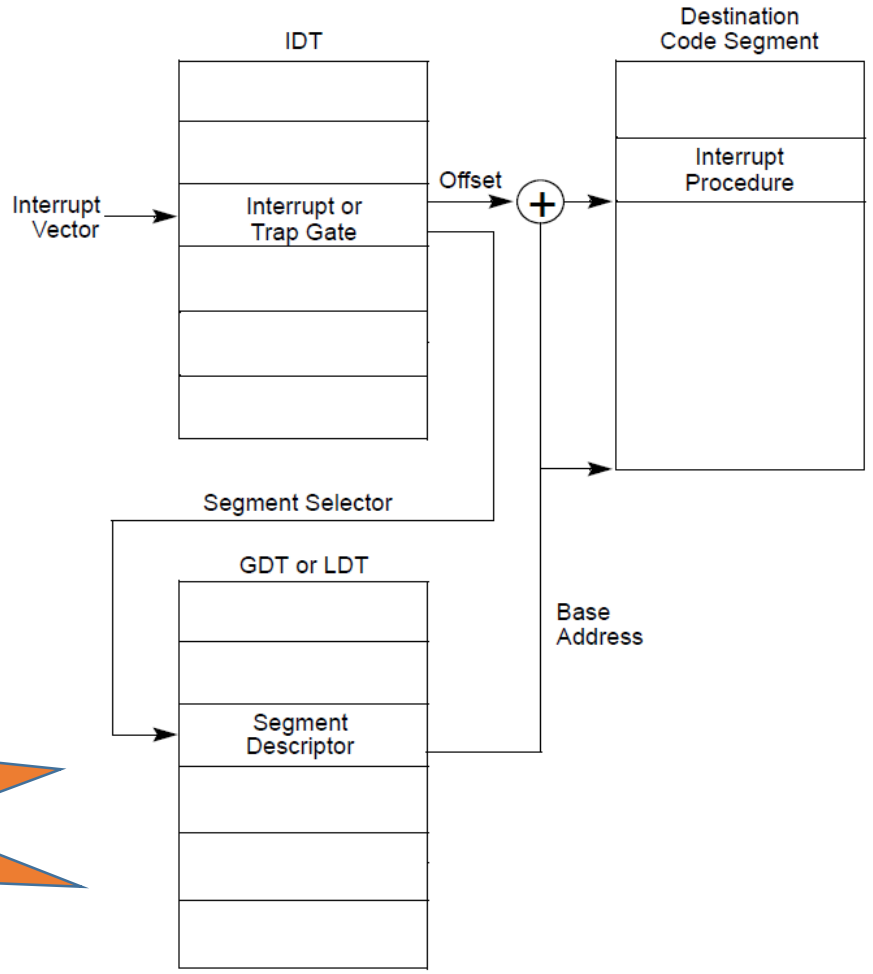
(32 bit) Gate Descriptors



Interrupt Procedure Calls

If the handler procedure is going to be executed at a numerically lower privilege level (i.e., higher privilege), a stack switch occurs. The stack specified in the current process' TSS is used.

Else, use current stack.



You see why we need a faster way to do syscall rather than through interrupts?

A digression: Task State Segment

- Originally meant for hardware task switching in 32-bit
  - Never used by Linux for process switching
- Still there in 64 bit but in 64 bit hardware task switching is not supported
- Basically, a state context with the contents of all registers
- Most important use: supply an alternative stack (kernel stack) for calling more privileged procedures
  - Only use of it in Linux
  - Pointed to by the privileged `tr` register
- IDT can contain task gates and in 32-bit system will trigger a hardware task switch

Task State Segment

Intended by Intel to be used for interrupt stacks. But Linux only use it for a few severe exceptions like double-fault.

Linux uses one TSS for each CPU

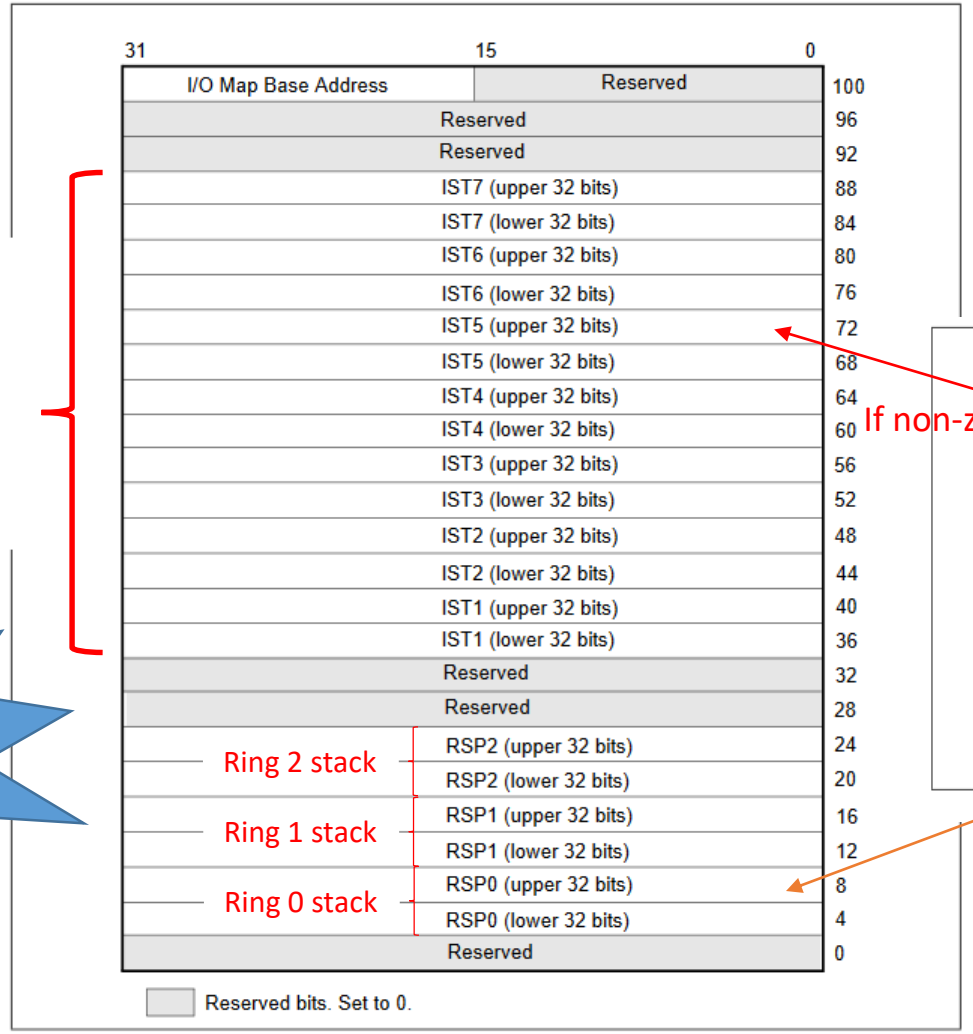


Figure 7-11. 64-Bit TSS Format

If non-zero...

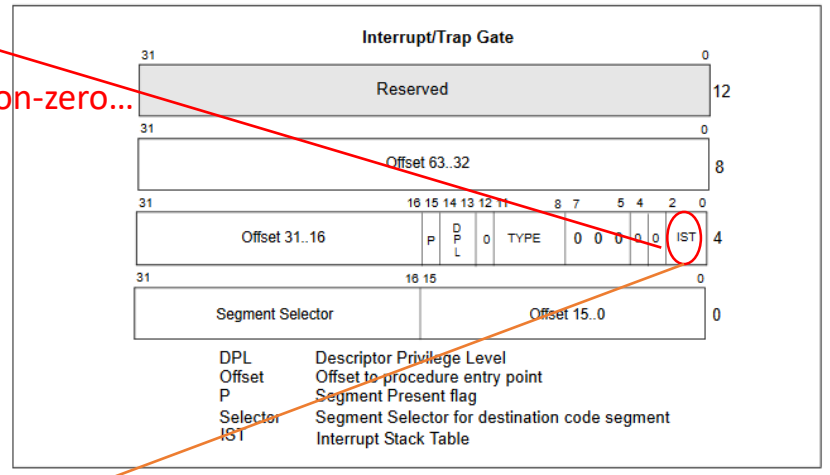


Figure 6-7. 64-Bit IDT Gate Descriptors

If zero...

Interrupt Stack Table

- **`ESTACK_DF`**
  - INT8 - Double Fault Exception
  - Invoked when an exception occurs while handling another exception
- **`ESTACK_NMI`**
  - INT2 - Non-maskable interrupt
- **`ESTACK_DB`**
  - Hardware debug interrupts (INT1) and for software debug interrupts (INT3)
- **`ESTACK_MCE`**
  - INT18 – Machine Check Exception
- Each stack is of size 4K (1 page)

Stack Switching

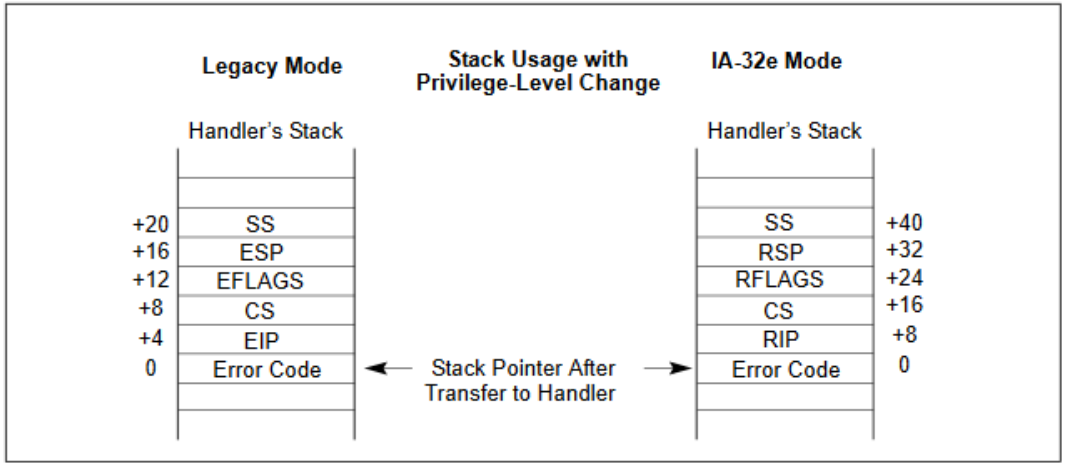


Figure 6-8. IA-32e Mode Stack Usage After Privilege Level Change

- In 32 bit, rely on hardware task switching to switch stack
- In 64 bit, done in **`entry_SYSCALL_64`** of **`arch/x86/entry/entry_64.S`**

Linux and IDT

- BIOS will initialize and use the IDT
- When Linux boots, it will overwrite introduce a new table **`idt_table`**
- In **`arch/x86/kernel/traps.c/trap_init()`** meaningful values will be written in
- Later on, drivers will request for IRQ from kernel

How Linux initialize the IDT

- All Linux interrupt handlers are activated via interrupt gates, and all restricted to kernel mode; no user mode access
  - Linux calls these "interrupt gates"
- Interrupt 4 (overflow), 5 (bounds check), and 128 (int 0x80) activates interrupt gates that are at Ring 3 (user mode)
  - Linux calls these "system gates"
- Interrupt 3 (breakpoint) is implemented via an interrupt gate at user mode privilege
  - Linux calls this "system interrupt gate"
- Most Linux trap handlers are activated via trap gates, all restricted to kernel mode
  - Linux calls these "trap gates"
- The sole use of a task gate in Linux is for the "double fault" exception. It is in Ring 0.

Multiple IDTs

- Immediately after kernel boots:
  - **`early_idts[]`** in **`arch/x86/kernel/idt.c:57`**
- Pre-IST stack version:
  - **`def_idts[]`** in **`arch/x86/kernel/idt.c:76`**
- Post-IST stack version:
  - **`ist_idts[]`** in **`arch/x86/kernel/idt.c:228`**

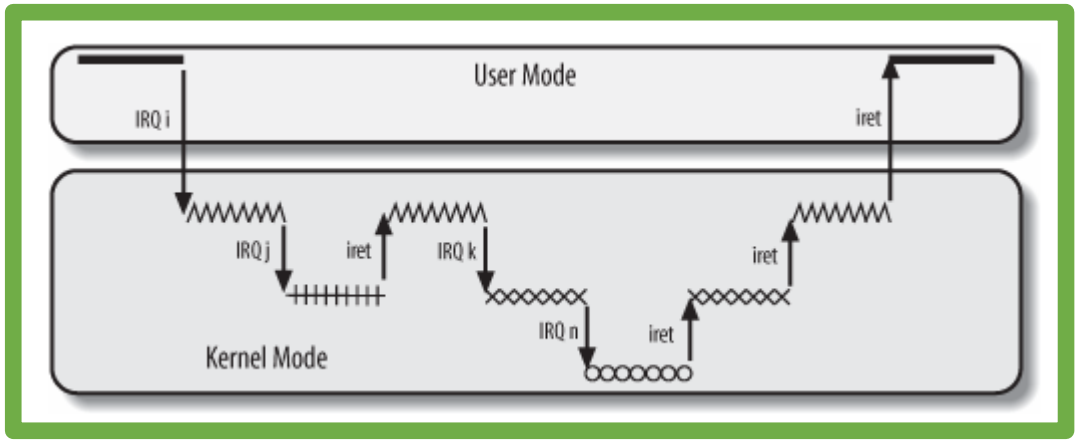
Setting up interrupt handling

- There are a number of varieties of IDT entries and their handlers, and macros are used to generate the "entry stubs"
- From interrupt 32 (known by a constant **`FIRST_EXTERNAL_VECTOR`**) onwards, the interrupts are handled in the same way, with a small exception.
  - If local APIC is turned on, then all interrupts above number 236 are "spurious" and treated as error.
  - Otherwise, all handled the same way.
- All will use **`sp0`** entry stack

Nested Interrupts

- A second (or even more) interrupt may come while the one is being serviced
- A kernel control path is the sequence of instructions executed by a kernel to handle a system call, an interrupt or an exception.
  - The kernel control path may be nested – while executing in the kernel, you may get a page fault!
- On strategy: mask out all interrupts during servicing
- Bad idea coz:
  - You have many cores – you want parallelism
  - Very likely nested interrupts have nothing to do with each other
  - You want to keep I/O devices fully utilized

Nested kernel control path



How to handle nested interrupts

- Unmask all interrupts (except the current IRQ) as soon as possible

- Leave current IRQ masked coz most interrupt servicing routines are *not* re-entrant
  - A piece of code is reentrant if it can be interrupted in the middle of its execution, and then be safely called again ("re-entered") before its previous invocations complete execution.
  - Not re-entrant – coz use the same stack

Nuts and bolts

- On entry to an interrupt handling, check if there was a privilege change
  - Done by check CS register
    - Constant __USER_CS if in user mode
    - Constant __KERNEL_CS if in kernel
- If a privilege level change, do the same entry steps as SYSCALL
  - **`swapgs`** to bring in per CPU data structure
  - **`SWITCH_TO_KERNEL_CR3`** to bring in full kernel page tables
  - Change stack to per task kernel stack
- If no privilege level change, do not do these!
  - In addition, check **`irq_count`** to see if we are interrupted inside an interrupt servicing routine

The OS aspect

The Linux Story

The overall strategy

- Divide interrupt handling into two parts
- Top half – immediate handler
  - Do the absolute minimal to handle the interrupt and return as quickly as possible
    - Enables near immediate response to new interrupts
  - Push the bulk of the non-critical processing to Part 2
- Bottom half – deferrable functions
  - Not time critical work
  - Mechanisms: tasklet, softirq, work queue, kernel thread

Top half

- Do the bare minimum amount of work to safely
  - Save registers
  - Unmask all other interrupts
- Exceptions are handled simply by sending a signal to the process that caused the exception
- I/O interrupt has to be handled using the current process

Limitations of an interrupt service routine

- Cannot sleep or call something that *might* sleep
- Cannot refer to **`current`** (pointer to the current process)
- Cannot do **`kmalloc(…, GFP_KERNEL)`** (which can sleep)
  - Must use **`kmalloc(…, GFP_ATOMIC)`** (which can fail)
- Cannot call **`schedule()`**
- Cannot do a **`down()`** semaphore call
  - But *can* do an **`up()`**
- Cannot transfer data to/from user space

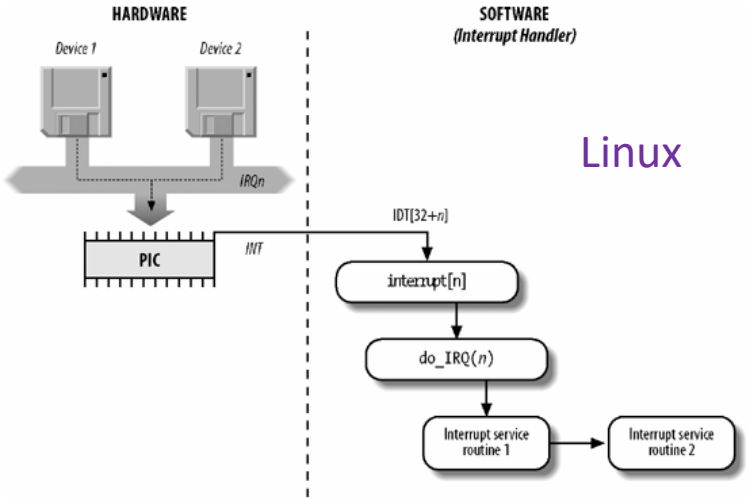
The Interrupt Stacks

- Interrupt service routines mostly written in C – hence need stack
- Exceptions use the kernel-level exception stack of the *current* process
  - Kernel stack is the stack in the kernel space of the current process
    - One per process whose address is located in the process data structure
  - The reason why cannot switch user context or sleep
- Interrupts uses a hard IRQ stack
  - One per processor, only one page frame in size
- SoftIRQ uses soft IRQ stack
  - One per processor, only one page frame in size
- The latter two configured in the IDT and TSS at boot time

Three types of interrupts

- I/O interrupts – I/O device requires attention
  - Three possible types of actions: critical, non-critical, non-critical deferrable
- Timer interrupts
- Interprocessor interrupts

I/O interrupts



Linux

- IRQ are few and hence a precious resource

- IRQ sharing
  - More than one device may share the same IRQ
  - Every service routine registered for the same IRQ must be tried coz you don't know which is the right one

- IRQ allocation
  - Device driver requests for IRQ assignment at the last possible moment just before use

A complication in SMP: lost IRQ

- IRQ a comes, CPU x handles
- IRQ b comes, CPU y handles
- Because of interrupt masking at beginning of process, before, say, CPU x can acknowledge, CPU y masks out all interrupts
- CPU x mistakenly thinks IRQ a disabled, returns without processing
- IRQ a is “lost” (unprocessed)
- Needs additional checking for this case

Bottom Half – deferrable functions

- SoftIRQ – “software” IRQ
- Tasklet
- Work queue
- Kernel thread

SoftIRQ

- In many ways, behaves like a hardware IRQ but done entirely in software
- Re-entrant code
  - Different processors may execute the same softIRQ handlers at the same time
  - Must protect data structures using spinlocks
- Can be interrupted
  - All hardware interrupts enabled
- Data structure: `softirq_vec[]` – an array of handlers

SoftIRQ vectors

include/linux/interrupt.h

```
enum
{
        HI_SOFTIRQ=0,
        TIMER_SOFTIRQ,
        NET_TX_SOFTIRQ,
        NET_RX_SOFTIRQ,
        BLOCK_SOFTIRQ,
        IRQ_POLL_SOFTIRQ,
        TASKLET_SOFTIRQ,
        SCHED_SOFTIRQ,
        HRTIMER_SOFTIRQ, /* Unused, but kept as tools rely on the
                            numbering. Sigh! */
        RCU_SOFTIRQ,    /* Preferable RCU should always be the last softirq */

        NR_SOFTIRQS
};
```

SoftIRQ processing

- Initialization – done by `open_softirq()`
- Activation – `raise_softirq()`
- Masking
  - Needed when a processor wants to change the softIRQ data structure
- Execution

Executing softIRQ

- Run at various points by the kernel:
  - After system calls
  - After exceptions
  - After interrupts (top halves/IRQs, including the timer intr)
  - When the scheduler runs `ksoftirqd`
- Softirq routines can be executed simultaneously on multiple CPUs:
  - Code must be re-entrant
  - Code must do its own locking as needed

ksoftirqd

- One daemon per processor
- Infinite loop check for and servicing softIRQ on that particular processor

```
for(;;) {
    set_current_state(TASK_INTERRUPTIBLE );
    schedule( );
    /* now in TASK_RUNNING state */
    while (local_softirq_pending( )) {
        preempt_disable();
        do_softirq( );
        preempt_enable();
        cond_resched( );
    }
}
```

Tasklets

- Implemented on top of softIRQ using **`HI_SOFTIRQ`** and **`TASKLET_SOFTIRQ`**
- Only real difference: **`HI_SOFTIRQ`**-based tasklets run prior to the **`TASKLET_SOFTIRQ`** tasklets
- Invoked either the **`tasklet_schedule( )`** function or the **`tasklet_hi_schedule( )`**

Tasklets vs softIRQ

- Tasklets can be statically or dynamically allocated
  - SoftIRQ is fixed

- Tasklets are not re-entrant
  - Tasklets of the same type runs serially in the entire system
  - Two tasklets of different types may run concurrently

Work Queues

- SoftIRQ and tasklets run in the interrupt context, work queue functions run in the process context
- Work queue functions allowed to sleep
- All cannot access user space
- Work queues can be created dynamically
- Or use the predefined (per CPU) "`events`" work queue
- A worker thread waits for work in the work queue

Kernel Threads

- Always operate in kernel mode
  - Again, no user context

- 2.6.30 introduced the notion of *threaded interrupt handlers*
  - Imported from the realtime tree
  - `request_threaded_irq()`
  - Now each bottom half has its own context, unlike work queues
  - Idea is to eventually replace tasklets and work queues

request_threaded_irq()

- Supplies a IRQ number, a handler and a threaded function
- Spawns a new kernel thread using the threaded function
- Handler is the hard IRQ handler
  - Determine interrupt cause
  - Returns **IRQ_HANDLED** or **IRQ_WAKE_THREAD**
- If **IRQ_WAKE_THREAD**, schedule kernel thread created during setup for execution
- As part of thread scheduling, threaded function will eventually be executed

A comparison

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

*Within limits, can be run by ksoftirqd

Signals

Unix Signals

- A signal is a notification of an event
- User application is stopped immediately
- A user-supplied signal handler will execute to completion
  - This handler is an userspace procedure
- User application will resume where it left off
- If no handler supplied, default action is taken
  - Usually a core dump followed by process termination

POSIX.1-1990 Signals

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at terminal
SIGTTIN	21,21,26	Stop	Terminal input for background process
SIGTTOU	22,22,27	Stop	Terminal output for background process

The signals **SIGKILL** and **SIGSTOP** cannot be caught, blocked, or ignored.

POSIX.1-2001 Signals

Signal	Value	Action	Comment
SIGBUS	10,7,10	Core	Bus error (bad memory access)
SIGPOLL		Term	Pollable event (Sys V). Synonym for **SIGIO**
SIGPROF	27,27,29	Term	Profiling timer expired
SIGSYS	12,31,12	Core	Bad argument to routine (SVr4)
SIGTRAP	5	Core	Trace/breakpoint trap
SIGURG	16,23,21	Ign	Urgent condition on socket (4.2BSD)
SIGVTALRM	26,26,28	Term	Virtual alarm clock (4.2BSD)
SIGXCPU	24,24,30	Core	CPU time limit exceeded (4.2BSD)
SIGXFSZ	25,25,31	Core	File size limit exceeded (4.2BSD)

Other signals

Signal	Value	Action	Comment
SIGIOT	6	Core	IOT trap. A synonym for **SIGABRT**
SIGEMT	7,-,7	Term	
SIGSTKFLT	-,16,-	Term	Stack fault on coprocessor (unused)
SIGIO	23,29,22	Term	I/O now possible (4.2BSD)
SIGCLD	-,-,18	Ign	A synonym for **SIGCHLD**
SIGPWR	29,30,19	Term	Power failure (System V)
SIGINFO	29,-,-		A synonym for **SIGPWR**
SIGLOST	-,-,-	Term	File lock lost (unused)
SIGWINCH	28,28,20	Ign	Window resize signal (4.3BSD, Sun)
SIGUNUSED	-,31,-	Core	Synonymous with **SIGSYS**

Examples from the keyboard

- Ctrl-c → 2/SIGINT signal
  - Default handler exits process
- Ctrl-z → 20/SIGTSTP signal
  - Default handler suspends process
- Ctrl-\ → 3/SIGQUIT signal
  - Default handler exits process

Sending signals

- You can do it at command line using the **`kill`** command
  - **`kill -signal pid`**
    - Send a signal of type **`signal`** to the process with id **`pid`**
    - Can specify either signal type name (-SIGINT) or number (-2)
  - If no signal type name or number specified, SIGTERM (15) will be sent
    - Default SIGTERM handler exits process
- Inside a program:
  - Use **`raise()`** system call to send to self
  - Use **`kill()`** system call to send to another process

Example

```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>

void sig_handler(int signo)
{
  if (signo == SIGINT)
    printf("received SIGINT\n");
}

int main(void)
{
  if (signal(SIGINT, sig_handler) == SIG_ERR)
  printf("\ncan't catch SIGINT\n");
  // A long long wait so that we can easily issue a signal to this process
  while(1)
    sleep(1);
  return 0;
}
```

Notes and caveats on signal

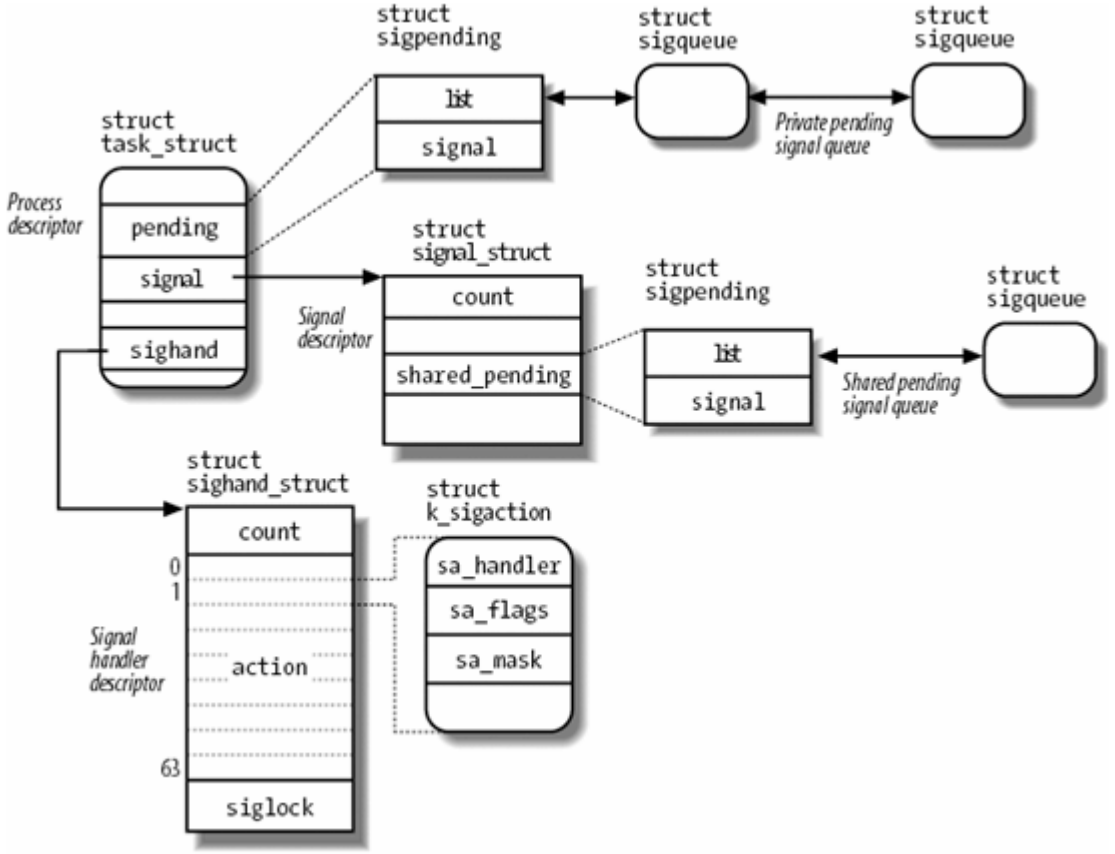
- Use **`sigprocmask()`** (and a host of related functions) to check or manipulate the signal mask
  - Read the man pages for caveats

- If you use the above, watch for race conditions
  - Normally, other signals are not raised in a handler as by default blocked

Kernel handling of user-level signals

- Kernel is involved in user-level signal handling because
  - Need to interrupt the user process and later resume it
  - Signal may arise from exceptions triggered from hardware
  - Signals may be sent from one process to another requiring OS checking and intervention
- Each process has its own set of signal handlers
- Each process can also have a list of pending signals
- In multicore systems, possible race conditions
  - Need to guard data structures with locks

Main data structure for signal handling



The two phases

- Signal generation
  - Kernel can send signal to user process
  - Another process may send signal to a user process
  - See functions in **`kernel/signal.c`**
  - Update the relevant process(es) data structures
    - Updates the pending signal list
    - If sending another process, and target is sleeping, "kick" that process so that signal will be delivered
    - If sending another process, and target is not sleeping, do an interprocessor interrupt
- Signal delivery
  - Set up invocation of the user level signal handler

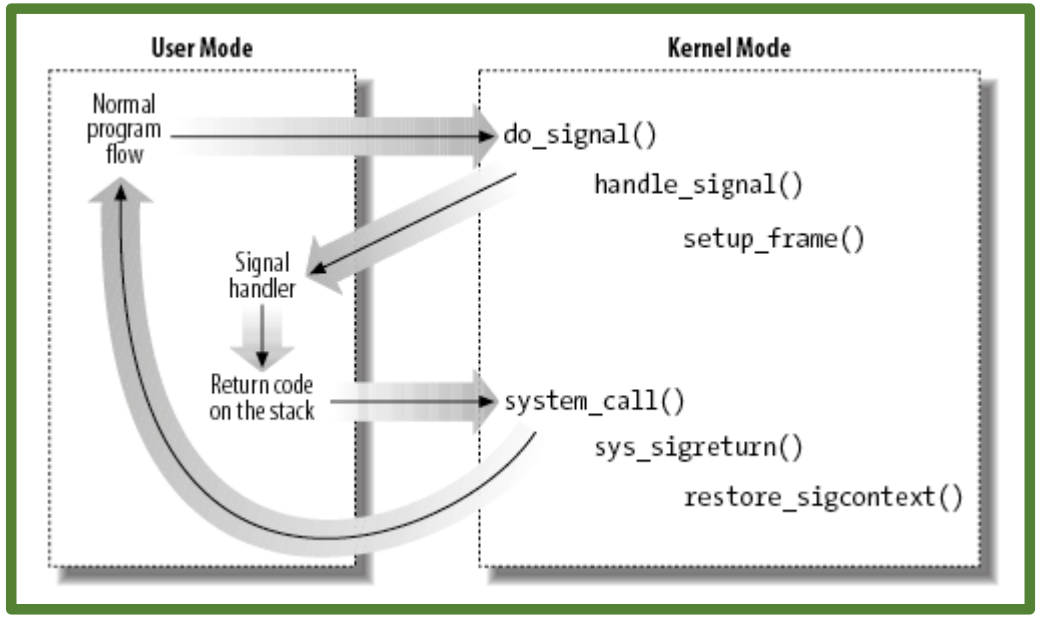
Signal delivery – `do_signal()`

- Before an interrupt routine (in kernel mode) returns to user mode, pending signal is checked
- If condition is right, `do_signal()` is called
- `do_signal()` is given the current process' context and signal mask
- If not handler, `do_signal()` performs the default action; a core dump (usually)
- If there is a handler, `do_signal()` will call `handle_signal()`

`handle_signal()`

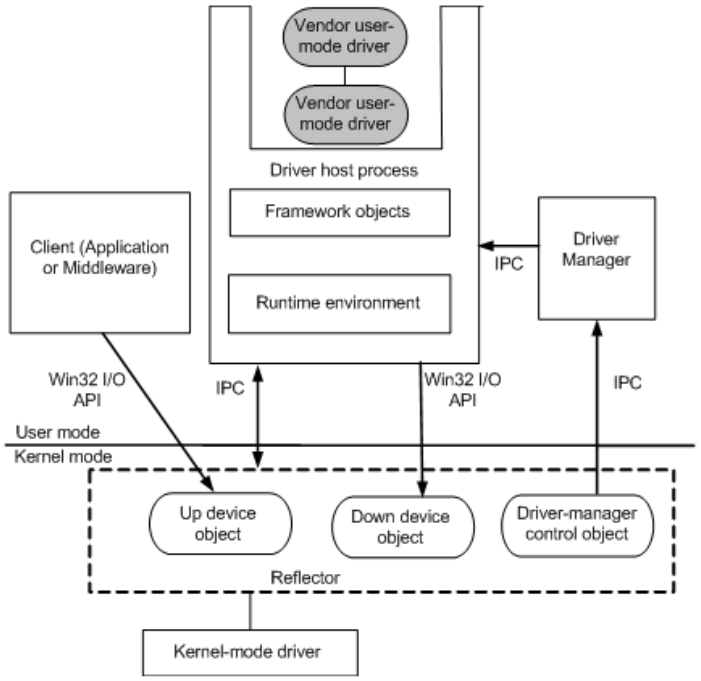
- Copies process context from kernel stack into user space stack
- Force EIP to point to first instruction of handler
- Manipulate user stack such that the handler will return onto the stack which will make a system call to sigreturn()
  - Cannot just return simply coz handler may make other system calls – including ones using "`int 80h`" – an interrupt!
- After all these setup, do a true return to user mode

Linux signal delivery



A Quick Word about Windows

- User-Mode Driver Framework (UMDF)
- UMDF drivers can handle hardware interrupt via the framework
- Interrupt objects are created and registered with the framework
- Same two-half philosophy:
  - `OnInterruptIsr` – do bare minimum
  - `OnWorkItem` – deferred processing for non-critical part of interrupt processing



End