

Interrupt Handler

Xiaoguang Wang

Summary of past lectures



Tools: building, exploring, and debugging Linux kernel

Core kernel infrastructure

- syscall, module, kernel data structures

Process management & scheduling

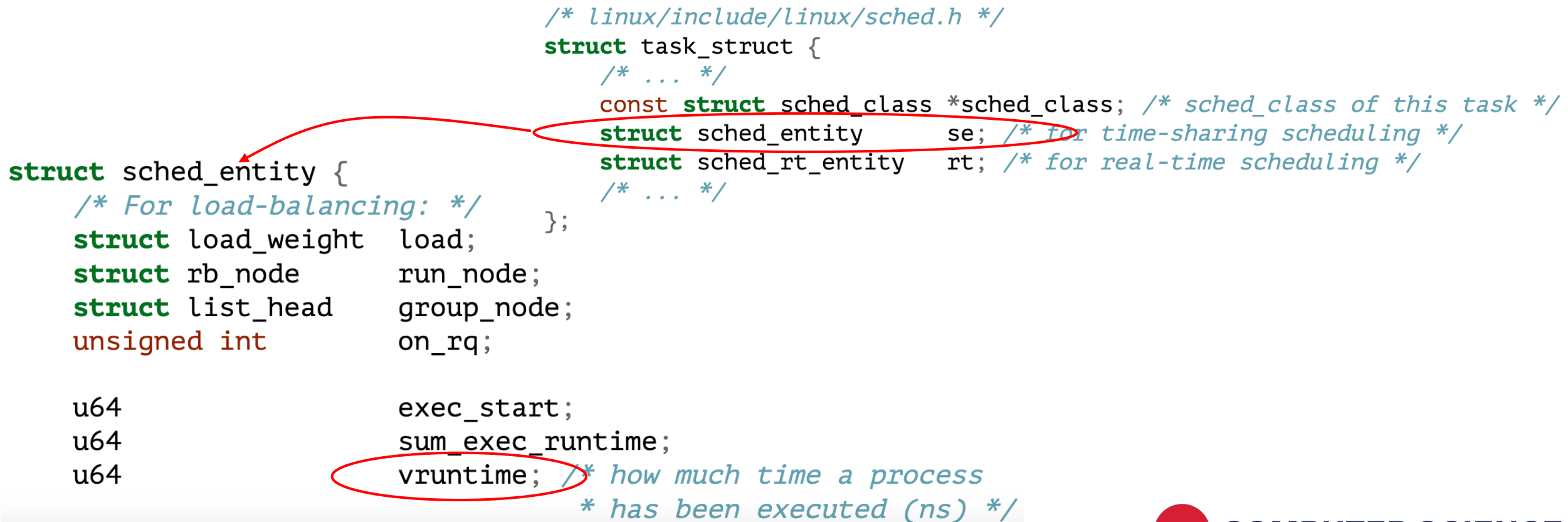
Recap: CFS implementation

Virtual runtime: how much time a process has been executed

```
/* linux/include/linux/sched.h */
struct task_struct {
    /* ... */
    const struct sched_class *sched_class; /* sched_class of this task */
    struct sched_entity se; /* for time-sharing scheduling */
    struct sched_rt_entity rt; /* for real-time scheduling */
    /* ... */
};

struct sched_entity {
    /* For load-balancing: */
    struct load_weight load;
    struct rb_node run_node;
    struct list_head group_node;
    unsigned int on_rq;

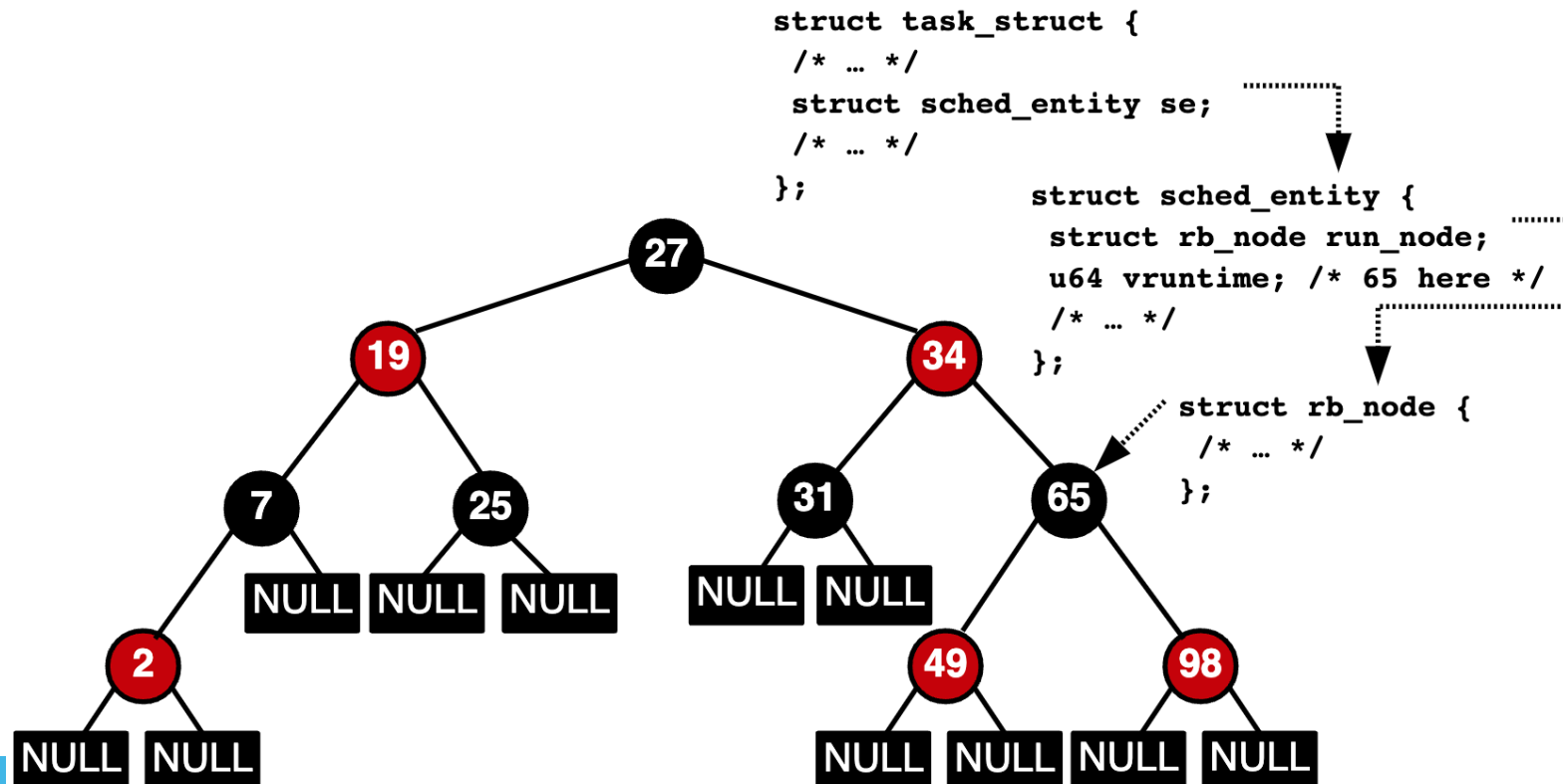
    u64 exec_start;
    u64 sum_exec_runtime;
    u64 vruntime; /* how much time a process
                  * has been executed (ns) */
};
```



Recap: process selection in CFS

CFS maintains a rbtrees of tasks indexed by vruntime (i.e., runqueue)

- Always pick a task with the smallest vruntime, the left-most node



Recap: CFS implementation

Upon every **timer interrupt**, CFS accounts the task's execution time

```
curr->sum_exec_runtime += delta_exec;  
schedstat_add(cfs_rq->exec_clock, delta_exec);  
  
/* update vruntime with delta_exec and nice value */  
curr->vruntime += calc_delta_fair(delta_exec, curr);  
update_min_vruntime(cfs_rq);
```

When a task is woken up, it is added to a runqueue

- insert a node in the rbtree

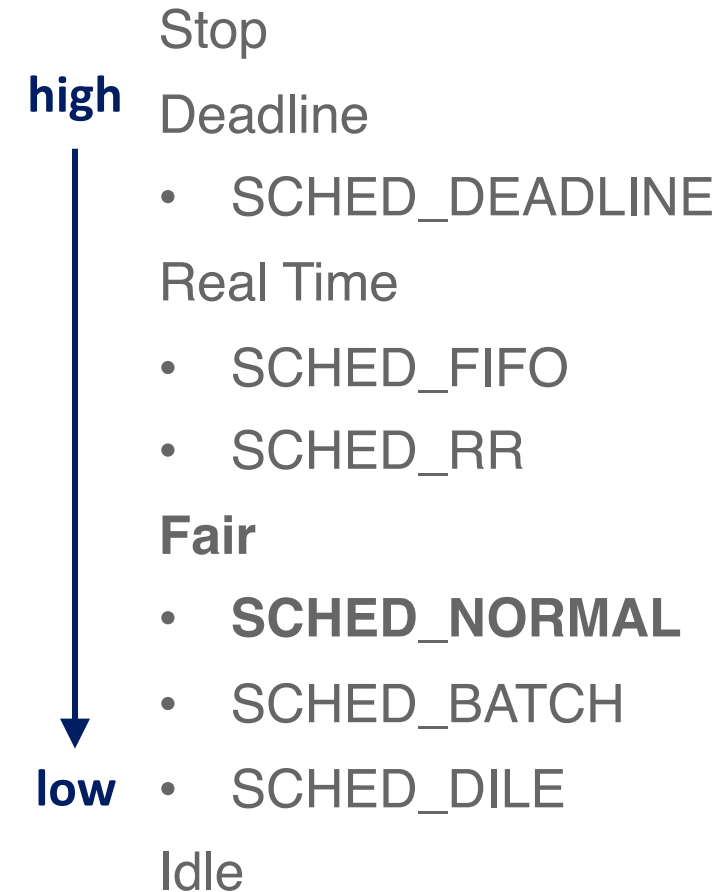
When a task goes to sleep, it is removed from a runqueue

- delete a node from the rbtree

Recap: entry point: schedule()

```
/* Pick up the highest-prio task: */
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
    const struct sched_class *class;
    struct task_struct *p;

    /* ... */
again:
    for_each_class(class) {
        /* In CFS, pick_next_task_fair() will be called.
         * pick_next_task_fair() eventually calls __pick_first_entity() */
        p = class->pick_next_task(rq, prev, rf);
        if (p) {
            if (unlikely(p == RETRY_TASK))
                goto again;
            return p;
        }
    }
    /* The idle class should always have a runnable task: */
    BUG();
}
```



Today: “interrupt”

A mechanism to implement abstraction and multiplexing

Interrupt: asking for a service to the kernel

- by software (e.g., `int`) or by hardware (e.g., keyboard)

Interrupt handling in Linux

- top half + bottom half

A hard disk drive



Roll over image to zoom in



WD Blue 500GB Mobile Hard Disk Drive - 5400 RPM SATA 6 Gb/s 7.0 MM 2.5 Inch - WD5000LPVX

[Visit the Western Digital Store](#)

4.4 ★★★★★ 3,610 ratings | [Search this page](#)

\$22⁹⁹

FREE Returns

Thank you for being an Amazon customer. Get \$50 off: Pay \$0.00 ~~\$22.99~~ upon approval for Amazon Visa.

Available at a lower price from [other sellers](#) that may not offer free Prime shipping.

Capacity: 500 GB

250 GB

320 GB

500 GB

\$22.99

Digital Storage
Capacity

500 GB

Hard Disk Interface

Serial ATA-600

Connectivity
Technology

SATA

How fast is this HDD

HDD data access time = seek time + rotational latency

Seek time

- The time to move the disk head to the track that contains data
- Average seek time: 4 ~ 10 msec

Rotational latency

- The delay for the rotation of the disk to bring the required disk sector under the read-write mechanism
- 5400 RPM: 5.56 msec

Data access time of this HDD: about 10 msec

Interrupt

Compared the the CPU, devices are slow (e.g., 10 msec)

- The kernel must be free to go and handle other work, dealing with the hardware only after that hardware has completed its work

How to know the completion of a hardware operation

- **Polling:** the kernel periodically checks the status of hardware
- **Interrupt:** the hardware signals its completion to the processor

Interrupt examples

- Completion of disk read, key press on a keyboard, network packet arrival

Interrupt controller



Interrupts are electrical signals multiplexed by the **interrupt controller**

- Sent on a specific pin of the CPU

Once an interrupt is received, a dedicated function is executed

- **Interrupt handler**

The kernel/user space can be interrupted at (nearly) any time to process an interrupt

Advanced PIC (APIC, I/O APIC)

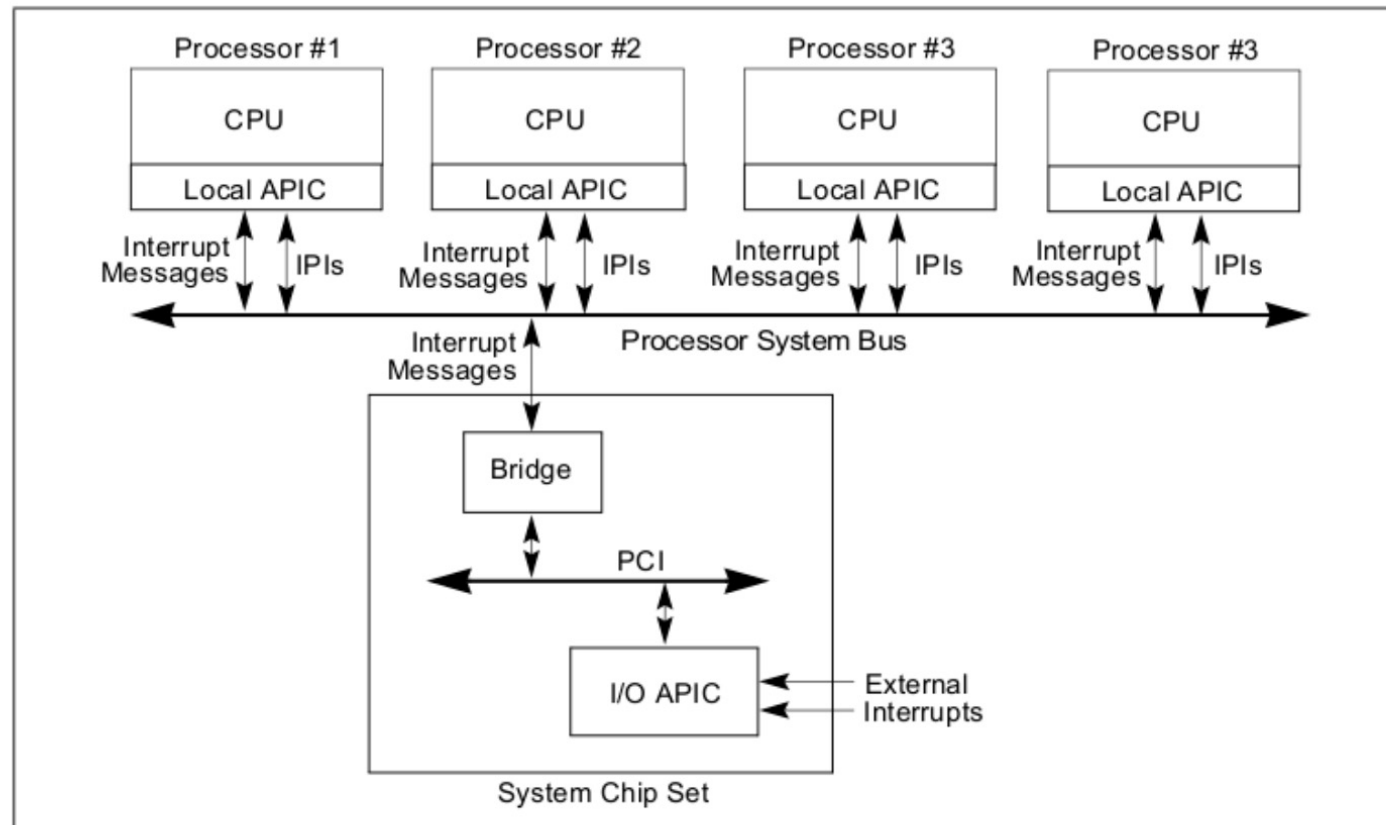


Figure 10-2. Local APICs and I/O APIC When Intel Xeon Processors Are Used in Multiple-Processor Systems

Advanced PIC (APIC, I/O APIC)

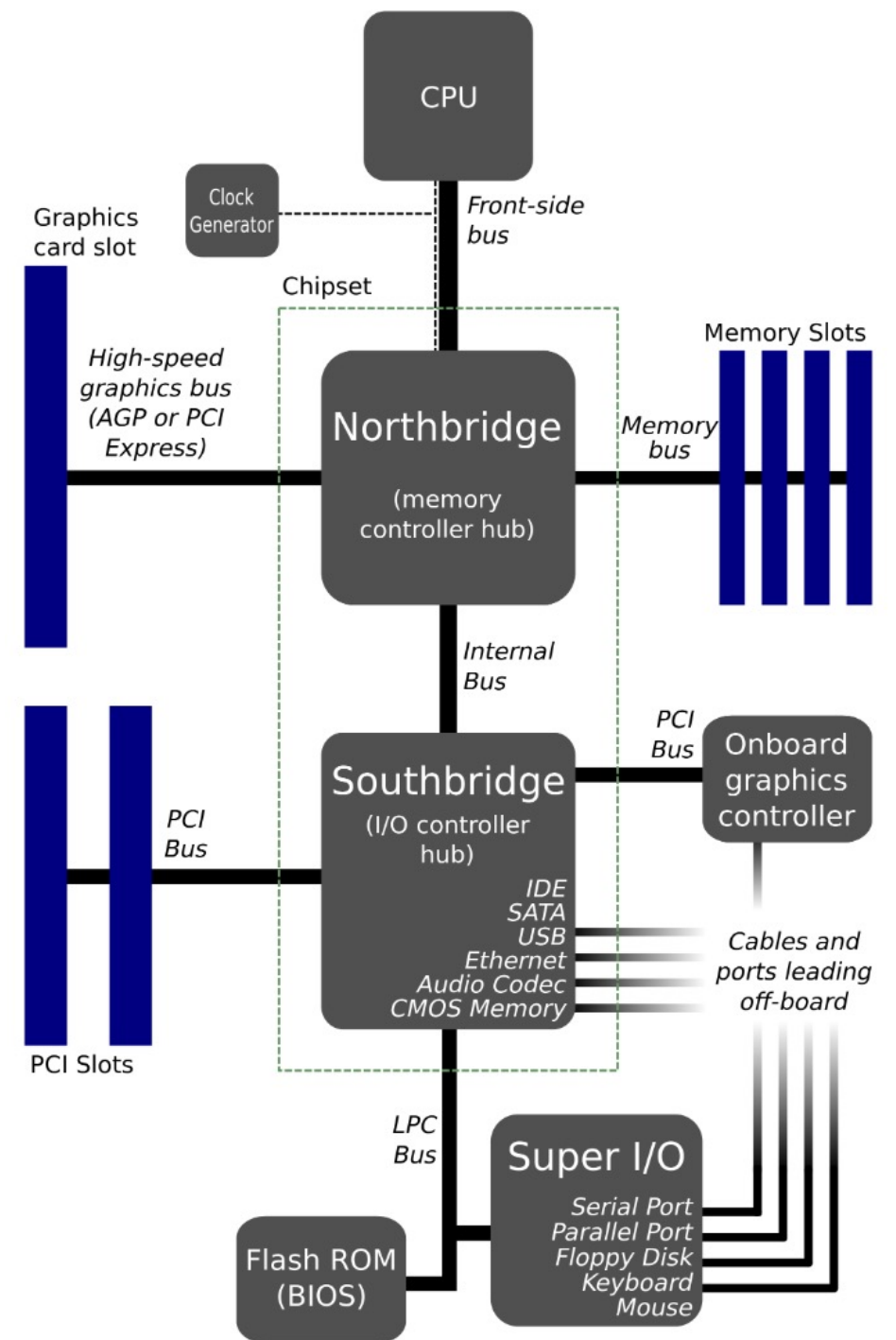
I/O APIC

- system chipset (or south bridge)
- redistribute interrupts to local APICs

Local APIC

- inside a processor chip
- has a timer, which raises **timer interrupt**
- issues an IPI (inter-process interrupt)

A bigger picture



Interrupt request (IRQ)

Interrupt line or interrupt request (IRQ)

- device identifier
- e.g., intel 8259A interrupt lines
 - IRQ 0: system timer, IRQ 1: keyboard controller
 - IRQ 3, 4: serial port, IRQ 5: terminal

Some interrupt lines can be shared among several devices

- True for most modern devices (PCIe)

Exception

Exception are interrupt issued by the CPU executing some **code**

- **Software interrupts**, as opposed to hardware ones (devices)
- Examples:
 - **Program faults**: divide-by-zero, page fault, general protection fault, etc
 - **Voluntary exceptions**: `int` assembly instruction, for example for `syscall` invocation (e.g., `int 0x80`)

Exceptions are managed by the kernel the same as hardware interrupts

Interface to hardware interrupt

Non-Maskable Interrupt (NMI)

- Never ignored, e.g., power failure, memory error
- In x86, vector 2, prevents other interrupts from executing.

Maskable interrupt

- Ignored when IF in EFLAGS is 0
- Enabling/disabling: - sti: set interrupt - cli: clear interrupt

“Software” interrupt: INT

Intentionally interrupts

- x86 provides the INT instruction
- Invokes the interrupt handler for the vector (0-255)

Entering: `int N`

Exiting: `iret`

Interrupt descriptor table

IDT

- Table of 256 16-byte entries on x86-64 (similar to GDT)
- Located in memory

IDTR register

- Stores current IDT

lidt instruction

- Loads IDTR with address and size of the IDT
- Takes in a linear address

Interrupt descriptor table

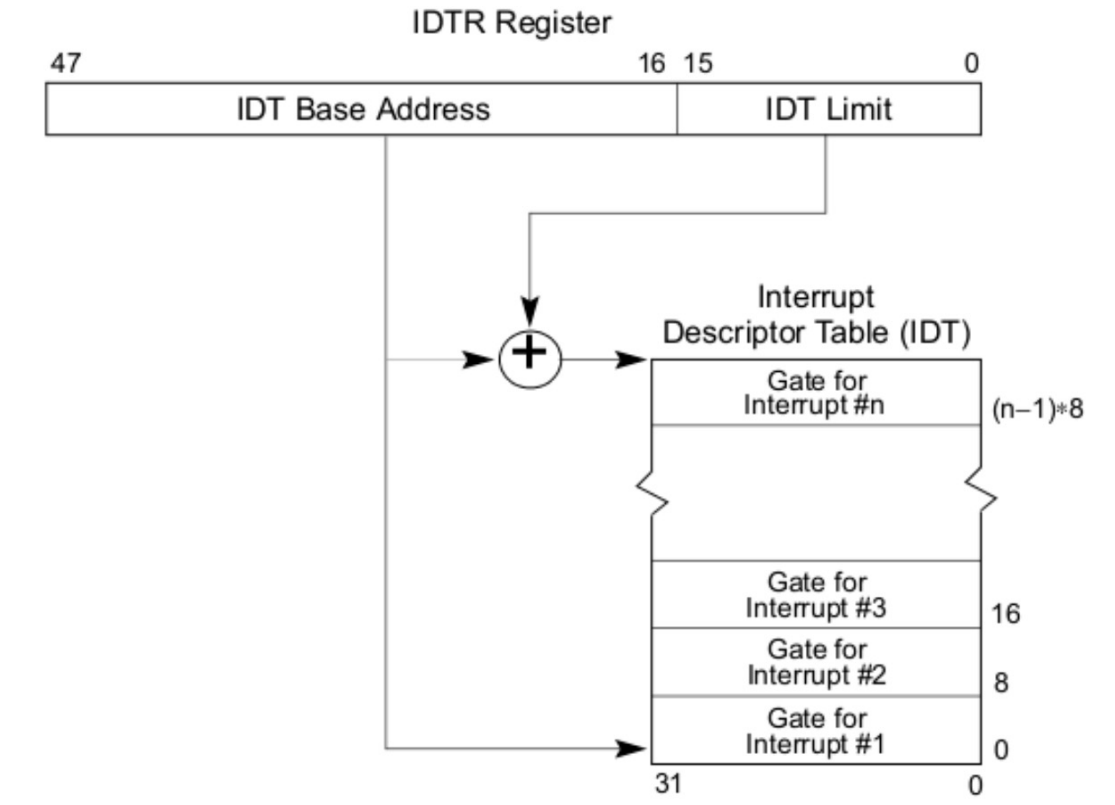


Figure 6-1. Relationship of the IDTR and IDT

Interrupt descriptor entry

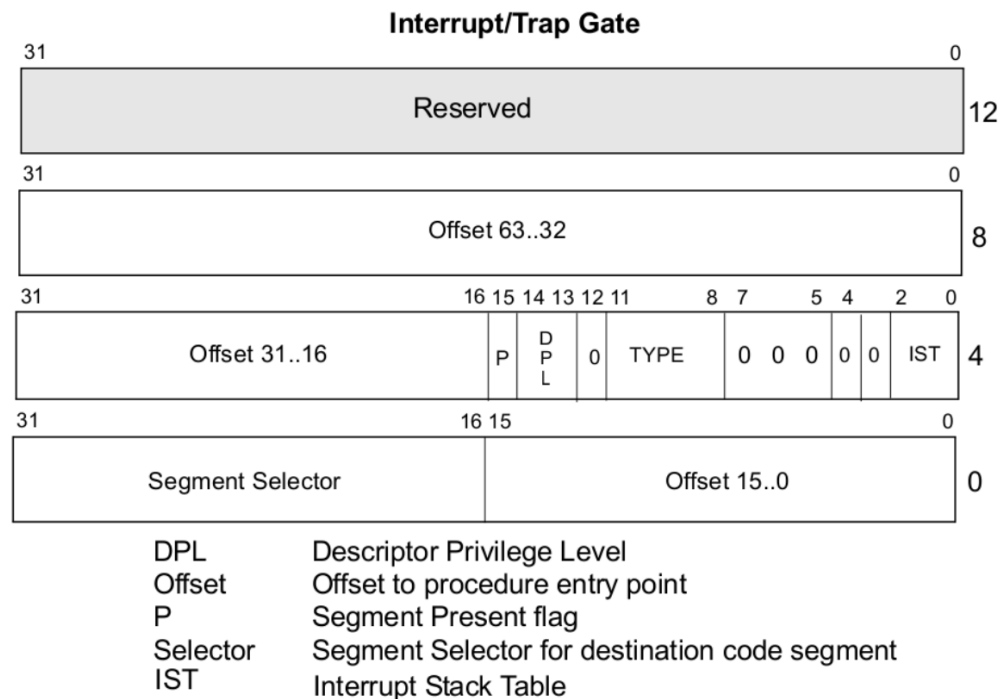


Figure 6-7. 64-Bit IDT Gate Descriptors

Offset is 64-bit, pointing to the dest IP

- split in multiple places

Segment selector points to the destination CS in the kernel

Present flag indicates that this is a valid entry

Predefined interrupt vectors

0: Divide Error

1: Debug Exception

2: Non-Maskable Interrupt

3: Breakpoint Exception (e.g., `int 3`)

4: Invalid Opcode

13: General Protection Fault

14: Page Fault

18: Machine Check (abort)

32-255: User Defined Interrupts

The INT instruction (1/2)

Decide the vector number, in this case it's the 0x80 in `int 0x80`

Fetch the interrupt descriptor for vector 0x80 from the IDT

- The CPU finds it by taking the 0x80'th 16-byte entry starting at the physical address that the IDTR CPU register points to

Check that $CPL \leq DPL$ in the

Save ESP and SS in a CPU-internal register

The INT instruction (2/2)

Load SS and ESP from TSS (Task State Segment)

Push user SS

Push user ESP

Push user EFLAGS

Push user CS

Push user EIP

Clear some EFLAGS bits

Set CS and EIP from IDT descriptor's segment selector and offset

Interrupt service routine (ISR)

Interrupt handler or Interrupt Service Routine (ISR)

- function executed by the CPU in response to a specific interrupt

In Linux, a normal C function matching a specific prototype to pass the handler information

Runs in interrupt context (or atomic context)

- Opposite to process context (system call)
- A task cannot sleep in an ISR because an interrupt context is not a schedulable entity

Two conflicting goals of ISR

1. Interrupt processing must be fast
 - We are indeed interrupting user processes executing (user/kernel space)
 - In addition, other interrupts may need to be **disabled** during an interrupt processing
2. It sometimes involves performing significant amount of work
 - so, it will take time
 - e.g., processing a network packet from the network card

Top half vs. bottom half

In many modern OSes, including Linux, interrupt processing is split into two parts:

Top-half: run immediately upon receipt of the interrupt

- Performs only the time-critical operations
 - e.g., acknowledging receipt of the interrupt
 - resetting the hardware

Bottom-half: less critical & time-consuming work

- Run later with other interrupts enabled

Example: network packet processing

Top-half: interrupt service routine

- Acknowledges the hardware
- Copies the new network packets into the main memory
- Readies the network card for more packets
- **Time critical** because the packet buffer on the network card is limited in size → packet drop

Bottom-half: processing the copied packets

- Softirq, tasklet, work queue
- Similar to thread pool in user-space

Register/free an interrupt handler

```
/* include/linux/interrupt.h */
```

```
/* This call allocates interrupt resources and enables the interrupt line and IRQ handling. */
```

```
int request_irq(unsigned int irq, irq_handler_t handler,  
               unsigned long irqflags, const char *devname, void *dev_id);
```

```
/* Free an interrupt allocated with request_irq*/
```

```
const void *free_irq(unsigned int irq, void *dev_id);
```

Writing an interrupt handler

```
/* linux/include/linux/interrupt.h */
```

```
/* Interrupt handler prototype
```

```
* @irq: the interrupt line number that the handler is serving
```

```
* @dev_id: a generic pointer that was given to request_irq() when the interrupt handler is registered
```

```
* Return value:
```

```
* IRQ_NONE: the interrupt is not handled (i.e., the expected
```

```
* device was not the source of the interrupt)
```

```
* IRQ_HANDLED: the interrupt is handled (i.e., the handler was
```

```
* correctly invoked) * #define IRQ_RETVAL(x) ((x) ? IRQ_HANDLED : IRQ_NONE) */
```

```
typedef irqreturn_t (*irq_handler_t)(int irq, void *dev_id);
```

Shared handlers

The `IRQF_SHARED` flag must be set in the flags argument to `request_irq()`

The `dev_id` argument must be unique to each registered handler.

- A pointer to any per-device structure is sufficient (e.g., `struct device`)

When the kernel receives an interrupt, it invokes sequentially each registered handler on the line.

- Therefore, it is important that the handler be capable of distinguishing whether it generated a given interrupt.

Interrupt context

Process context: normal task execution, syscall, and exception

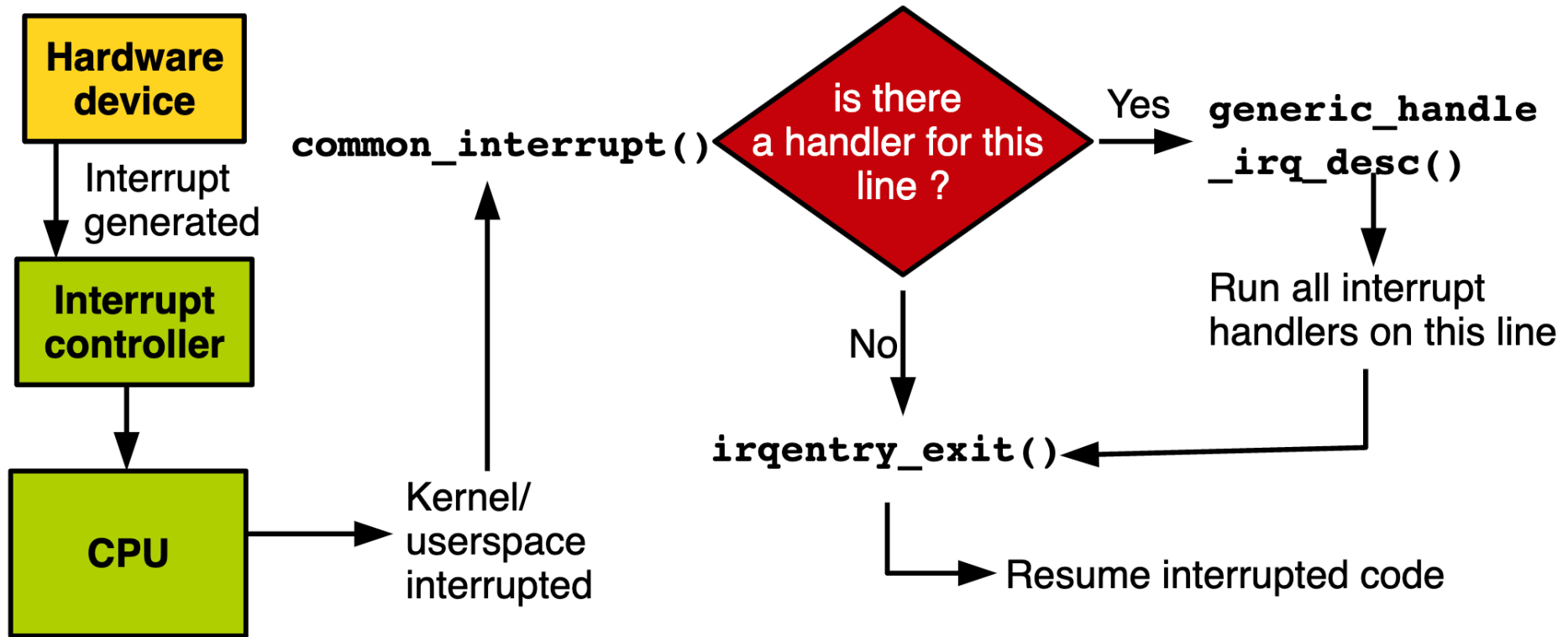
Interrupt context: interrupt service routine (ISR)

- **Sleeping/blocking is not possible** because the ISR is not a schedule entity
- No `kmalloc(size, GFP_KERNEL)`: use `GFP_ATOMIC`
- No blocking locking (e.g., mutex): use spinlock
- No `printk`: use `trace_printk`

Small stack size

- Interrupt stack: one page (4KB)

Interrupt handling in Linux



Interrupt handling in Linux

Specific entry point for each interrupt line

- Saves the interrupt number and the current registers
- Calls `common_interrupt()`

```
void common_interrupt(struct pt_regs *regs, u32 vector)
```

- Acknowledge interrupt reception and disable the line
- Calls architecture specific functions

```
236 /*
237  * common_interrupt() handles all normal device IRQ's (the special SMP
238  * cross-CPU interrupts have their own entry points).
239  */
240 DEFINE_IDENTRY_IRQ(common_interrupt)
241 {
242     struct pt_regs *old_regs = set_irq_regs(regs);
243     struct irq_desc *desc;
```

/proc/interrupts

```
$ cat /proc/interrupts
```

```
# Int line
```

```
# /      Num of occurrence per CPU
```

```
# /      /      Int controller
```

```
# /      /      /      Edge/level
```

```
# /      /      /      /      Device name
```

```
# /      /      /      /      /
```

	CPU0	CPU1			
0:	34	0	IO-APIC	2-edge	timer
1:	26	8	IO-APIC	1-edge	i8042
8:	0	0	IO-APIC	8-edge	rtc0
9:	0	0	IO-APIC	9-fastestoi	acpi
12:	156	0	IO-APIC	12-edge	i8042
14:	0	0	IO-APIC	14-edge	ata_piix
15:	116	24	IO-APIC	15-edge	ata_piix
19:	5	68	IO-APIC	19-fastestoi	virtio0
21:	3142	533	IO-APIC	21-fastestoi	ahci[0000:00:0d.0]
22:	27	0	IO-APIC	22-fastestoi	ohci_hcd:usb1
NMI:	0	0	Non-maskable interrupts		
LOC:	5031	4373	Local timer interrupts		
PMI:	0	0	Performance monitoring interrupts		
TLB:	27	89	TLB shootdowns		

Interrupt control



Kernel code sometimes needs to disable interrupts to ensure **atomic execution of a section of code**

- By disabling interrupts, you can guarantee that an interrupt handler will not preempt your current code.
- Moreover, disabling interrupts also disables kernel preemption.

Note that disabling interrupts does not protect against concurrent access from other cores

- Need locking, often used in conjunction with interrupts disabling

The kernel provides an API to disable/enable interrupts

Disabling ints on the local core

Disable and enable IRQ

```
local_irq_disable();  
/* interrupts are disable ... */  
local_irq_enable();
```

What happened if `local_irq_disable()` is called twice?

```
local_irq_disable(); /* interrupt is disabled */  
local_irq_disable();  
/* ... */  
local_irq_enable(); /* interrupt is enabled! */  
local_irq_enable();
```

Disabling a specific interrupt line

/ disable_irq - disable an irq and wait for completion*

** @irq: Interrupt to disable*/*

```
void disable_irq(unsigned int irq);
```

/ disable_irq_nosync - disable an irq without waiting */*

```
void disable_irq_nosync(unsigned int irq);
```

/ enable_irq - enable handling of an irq */*

```
void enable_irq(unsigned int irq);
```

Further readings



[LWN: Debugging the kernel using Ftrace - part 1](#)

[0xAX: Interrupts and Interrupt Handling](#)

Next week



Interrupt handler: bottom half

Kernel synchronization