

Lecture 13

Interrupts and Exceptions
the IDT, IRQs, bottom halves, timers

Md Mahbub E Noor

Assistant Professor, Department of CSE

University of Barishal

```
./exc (real)
7 / 0      -> FPE   trap  0
ud2        -> ILL   trap  6
hlt        -> SEGV  trap 13
read *NULL -> SEGV trap 14 err 0x4
# the CPU's own numbers, seen from C
```

What you will learn today



Classify events that stop the CPU: interrupts, traps, faults and aborts.



Explain the IDT and what the CPU saves when an exception or interrupt arrives.



Follow a disk interrupt from the device to the woken process, with real traces.



Explain top and bottom halves (softirqs, workqueues) and kernel timers.

Today's plan (60 min)

0-5

Recap of 12.2

5-12

Three ways in

12-25

Exceptions, the IDT

25-40

Device interrupts

40-50

Bottom halves

50-55

Timers

55-60

Lab, checkpoint

Lab 13: irqtop.c, a top for interrupts.

Quick recap of Lecture 12.2

Q1

Where is an ext2 superblock, whatever the block size?

At byte 1024; its magic 0xEF53 is at byte 1080 (53 ef, little-endian).

Q2

write() returned 0 errors. Is the data on the disk?

No: it is in dirty pages of the page cache. fsync waits until the disk has it.

Q3

After a crash, what does the ext4 journal replay?

Only committed transactions. lost.txt (never committed) was gone; kept.bin (fsync) came back.

PART 1

Three ways into the kernel

On purpose, by mistake, or from outside.

Every event that stops the CPU is one of four kinds

class	cause	sync?	returns to	examples (x86-64 Linux)
interrupt	a device, a timer, another CPU	async	the next instruction	disk done, key pressed, timer tick
trap	an instruction, on purpose	sync	the next instruction	syscall, int3 (breakpoint)
fault	an instruction that might be fixed	sync	the SAME instruction, again	page fault (Lecture 11.1), #GP, #UD
abort	a fatal hardware error	sync	never	machine check, double fault

A student raises a hand: to ask the teacher something: a **trap** (system call). Planned, at a known moment.

A student makes a mistake: the teacher must look at it: a **fault**. Fix it and try the same step again, or fail the student (SIGSEGV).

The phone rings: in the middle of a sentence: an **interrupt**. Finish the sentence, answer, then continue the lesson.

Book: CS:APP §8.1 (exceptions, four classes); Love, LKD Ch. 7

PART 2

Exceptions and the IDT

256 numbered events; one table of handlers.

Make the CPU raise exceptions on purpose (exc.c)

```
./exc (real; the same in the QEMU guest)

7 / 0          -> FPE  si_code 1   trap  0   err 0x0
int3           -> TRAP si_code 128 trap  3   err 0x0
ud2           -> ILL  si_code 2   trap  6   err 0x0
hlt (kernel only) -> SEGV si_code 128 trap 13   err 0x0
inb $0x60 (port I/O) -> SEGV si_code 128 trap 13   err 0x0
read *NULL     -> SEGV si_code 1   trap 14   err 0x4
write "hello" (rodata) -> SEGV si_code 2   trap 14   err 0x7
call code on the stack -> SEGV si_code 2   trap 14   err 0x15
```

```
exc.c: the handler (core lines)

void handler(int sig, siginfo_t *si,
             void *ctx) {
    ucontext_t *uc = ctx;
    ... uc->uc_mcontext
        .gregs[REG_TRAPNO]
    ... .gregs[REG_ERR]
    write(1, m, n);
    _exit(0);
}
```

trap: the CPU's exception number (vector): 0 #DE, 3 #BP, 6 #UD, 13 #GP, 14 #PF

err: an error code the CPU pushes for some exceptions; for #PF it says why (slide 11)

si_code 128: SI_KERNEL: the kernel sent the signal without details (int3, #GP)

The IDT: an array of 256 handler addresses

arch/x86/include/asm/trapnr.h (real)

```
7  #define X86_TRAP_DE 0 /* Divide-by-zero */
10 #define X86_TRAP_BP 3 /* Breakpoint */
13 #define X86_TRAP_UD 6 /* Invalid Opcode */
15 #define X86_TRAP_DF 8 /* Double Fault */
20 #define X86_TRAP_GP 13 /* General Protection */
21 #define X86_TRAP_PF 14 /* Page Fault */
25 #define X86_TRAP_MC 18 /* Machine Check */
```

arch/x86/kernel/idt.c (real lines)

```
65 SYSG(X86_TRAP_BP, asm_exc_int3),
71 INTG(X86_TRAP_PF, asm_exc_page_fault),
84 static const ... struct idt_data def_idts[] = {
85     INTG(X86_TRAP_DE, asm_exc_divide_error),
88     INTG(X86_TRAP_UD, asm_exc_invalid_op),
94     INTG(X86_TRAP_GP, asm_exc_general_protection),
108    ISTG(X86_TRAP_MC, asm_exc_machine_check, ...),
```

0 – 31 exceptions, fixed by Intel

32 – 235 device interrupts (FIRST_EXTERNAL_VECTOR 0x20)

0x80 old 32-bit int \$0x80 system call (Lecture 9)

0xec – 0xff local timer 0xec, IPIs (reschedule 0xfd), spurious 0xff

C link: think of `void (*idt[256])(void);`. The CPU does `idt[vector]()` by itself. The kernel fills the table at boot; the IDTR register holds its address (lidt).

Book: Bovet & Cesati Ch. 4 (interrupts and exceptions); Love, LKD Ch. 7

What the CPU does by itself, before any kernel code runs

- 1 Finish the current instruction (interrupt) or stop it (fault)
- 2 Look up IDT[vector]; check the gate's privilege level
- 3 From user mode: switch to this thread's **kernel stack** (address from the TSS)
- 4 Push **SS, RSP, RFLAGS, CS, RIP** (and an error code for some exceptions)
- 5 Interrupt gate: clear **IF** (no more interrupts on this CPU); jump to the handler
- 6 Handler done: **iretq** pops the five values and the program continues

```
arch/x86/include/asm/ptrace.h (real)

59 struct pt_regs {
   ... r15 ... ax: saved by the kernel's entry code
84     unsigned long orig_ax; /* err code, IRQ, nr */
85     /* Return frame for iretq */
86     unsigned long ip;
87     unsigned long cs;
88     unsigned long flags;
89     unsigned long sp;
90     unsigned long ss;
```

SAME AS C

A function call, with more saved

call pushes only the return address; an exception pushes 5 values and switches stacks. `iretq` is the matching `ret`. And as with `setjmp/longjmp`, the saved registers can be changed before returning (Lecture 10.2's signal frames).

Book: Bovet & Cesati §4.2; Intel SDM Vol. 3, Ch. 6

From the CPU's #DE to your SIGFPE: the kernel's side

arch/x86/kernel/traps.c (real lines)

```
194 DEFINE_IDTENTRY(exc_divide_error)
195 {
196     do_error_trap(regs, 0, "divide error", X86_TRAP_DE,
197                 SIGFPE, FPE_INTDIV, error_get_trap_addr(regs));
198 }
199
200 do_trap(int trapnr, int signr, char *str, struct pt_regs *regs, ...)
201 {
202     if (!do_trap_no_signal(tsk, trapnr, str, regs, error_code))
203         return; /* in the kernel: fix up or die */
204     show_signal(tsk, signr, "trap ", str, regs, error_code);
205     if (!sicode)
206         force_sig(signr); /* si_code SI_KERNEL */
207     else
208         force_sig_fault(signr, sicode, addr);
209 }
```

If the divide happens inside the kernel there is no process to blame: `do_trap_no_signal` fixes it up, or the kernel prints an oops and dies.

dmesg (no handler installed; real)

```
traps: t2[392] trap divide error
       ip:557bf2e2716a sp:7ffdfb7cb4c0 error:0
       in t2[116a,557bf2e27000+1000]
```

- 1 CPU: #DE → IDT[0] → `asm_exc_divide_error`
- 2 `exc_divide_error` → `do_error_trap` → `do_trap`
- 3 In user mode: a signal, SIGFPE, `si_code` FPE_INTDIV (1)
- 4 On the way back to user mode: the handler runs, or the default kills (Lecture 10.2)

Book: Love, LKD Ch. 7

The page-fault error code: why did it fault?

```
arch/x86/include/asm/trap_pf.h (real)
18 X86_PF_PROT = 1 << 0, /* 0: not present */
19 X86_PF_WRITE = 1 << 1, /* 0: read */
20 X86_PF_USER = 1 << 2, /* 0: kernel mode */
21 X86_PF_RSVD = 1 << 3,
22 X86_PF_INSTR = 1 << 4, /* instruction fetch */
23 X86_PF_PK = 1 << 5,
```

exc.c case	err	bits	meaning
read *NULL	0x4	USER	a user read of a page that is not there
write "hello"	0x7	USER WRITE PROT	the page is there (r--), writing is not allowed
code on the stack	0x15	USER INSTR PROT	the page is there (rw-), but NX: no execute

1	0	1	0	1
INSTR (1)	RSVD (1)	USER (1)	WRITE (1)	PROT (1)

err 0x15 = 1 0 1 0 1: user, instruction fetch, page present (but not executable)

Lecture 11.1 again: SEGV_MAPERR (si_code 1) when PROT = 0, SEGV_ACCERR (si_code 2) when PROT = 1. The kernel decides from these bits and from the VMA.

The faulting address is not in the error code: the CPU puts it in register **CR2**, and `exc_page_fault` reads it first (`read_cr2()`, `fault.c` line 1512).

Book: Intel SDM Vol. 3 §4.7 (page-fault exceptions)

PART 3

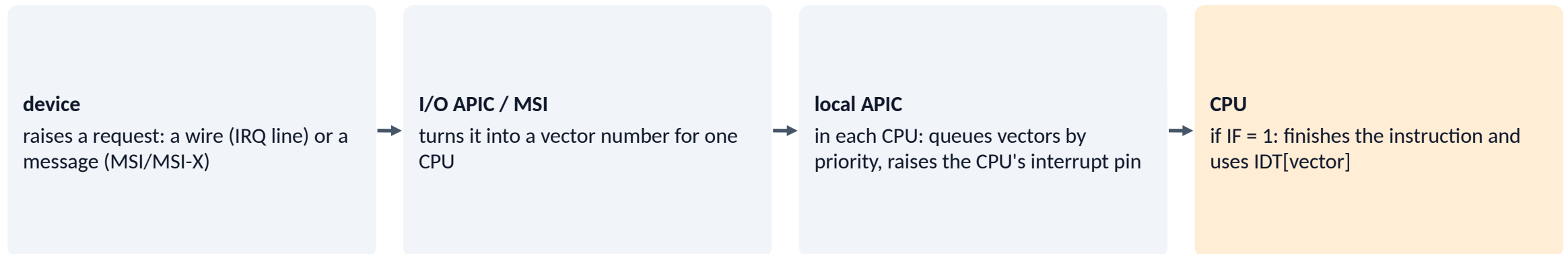
Device interrupts

From a disk's "done" to a woken process.

Polling or interrupts: who waits for whom?

	polling	interrupts
how	the CPU asks again and again: done?	the device tells the CPU when done
CPU while waiting	busy, asking	free: runs other programs, or sleeps (hlt)
delay after "done"	up to one polling interval	a few microseconds
good for	very fast devices (NVMe, 100 Gbit)	keyboard, disk, timer, USB, ...

Analogy: waiting for a parcel. Polling: you walk to the gate every five minutes to look. Interrupts: the courier rings your phone. You can study in between.



Book: Love, LKD Ch. 7; Bovet & Cesati §4.2

/proc/interrupts: every interrupt, counted per CPU

```
cat /proc/interrupts (QEMU guest, 1 CPU; real, shortened)
```

```

CPU0
 0: 579 IO-APIC 2-edge timer
 1: 9 IO-APIC 1-edge i8042
 4: 18 IO-APIC 4-edge ttyS0
 8: 1 IO-APIC 8-edge rtc0
14: 0 IO-APIC 14-edge ata_piix
25: 10 PCI-MSIX-0000:00:04.0 1-edge virtio0-req.0
NMI: 0 Non-maskable interrupts
LOC: 2328 Local timer interrupts
RES: 0 Rescheduling interrupts
CAL: 0 Function call interrupts
TLB: 0 TLB shutdowns
```

number: the IRQ number (not the vector); one column of counts per CPU

IO-APIC / PCI-MSIX who delivers it; edge or level ("fastest") triggering

the name given by the driver in request_irq: virtio0-req.0 is our disk, ttyS0 the serial console

LOC, RES, CAL, TLB CPU-local: timer ticks, and messages between CPUs (IPIs)

Our 2-CPU test machine shows the same kinds of lines: virtio1-req.0 had **1973** interrupts on CPU 1.

On a lab PC: ahci or nvme, xhci_hcd (USB), i915 ...

Book: Love, LKD Ch. 7 (/proc/interrupts)

How many per second? Watch them change

QEMU guest: `./irqtop while dd reads 64 MiB (real)`

```
IRQ  per second  name
25      493  PCI-MSIX-0000:00:04.0 1-edge virtio0-req.0
LOC      351  Local timer interrupts
```

test machine (2 CPUs): `./irqtop, idle (real)`

```
LOC      92  Local timer interrupts
CAL      25  Function call interrupts
49        3  ... virtio7-input.0
```

QEMU guest, 5 seconds	LOC (timer) interrupts
idle (sleep 5)	74
busy (yes > /dev/null)	4926 ≈ 985 per second

Tickless (NO_HZ): a busy CPU gets a timer tick 1000 times a second (HZ = 1000, next part). An idle CPU programs its timer only for the next real event and sleeps: 15 ticks a second instead of 1000. That saves power.

1 Disk reading: about 500 disk interrupts a second, one for each finished batch of requests (read-ahead makes them bigger).

2 In the lab: run `irqtop`, then type, move the mouse, copy a file: watch which line wakes up.

Inside the kernel: from vector to the driver's handler

```
● ● ● __common_interrupt, function_graph (guest; real)

__common_interrupt() {
  handle_edge_irq() {
    irq_chip_ack_parent() { apic_ack_edge(); }
    handle_irq_event() {
      __handle_irq_event_percpu() {
        vring_interrupt() {
          virtblk_done() {
            virtqueue_get_buf();
            blk_mq_complete_request(); 421 us
            virtqueue_enable_cb();
          }
        }
      }
    }
    add_interrupt_randomness(); /* feeds /dev/random */
  }
} 1560 us
```

```
● ● ● arch/x86/kernel/irq.c (real lines)

247 DEFINE_IDENTRY_IRQ(common_interrupt)
255     desc = __this_cpu_read(vector_irq[vector]);
256     if (likely(!IS_ERR_OR_NULL(desc))) {
257         handle_irq(desc, regs);
```

```
● ● ● kernel/irq/handle.c (real line)

139 irqreturn_t __handle_irq_event_percpu(...)
158     res = action->handler(irq, action->dev_id);
```

1 vector → **irq_desc** (one per IRQ number)

2 flow handler (edge/level) → the **driver's handler**

Book: Love, LKD Ch. 7 (interrupt handlers)

The whole story in one trace: request, interrupt, wake-up

● ● ● ftrace events during readone /mnt/hello.txt, cold (QEMU guest; real, shortened)

```
readone [000] ..... 20.465358: block_rq_issue: 253,0 RM 4096 () 304 + 8 [readone]
<idle> [000] d.h1. 20.465556: irq_handler_entry: irq=25 name=virtio0-req.0
<idle> [000] d.h2. 20.465571: block_rq_complete: 253,0 RA () 296 + 8 [0]
<idle> [000] d.h2. 20.465600: block_rq_complete: 253,0 RM () 304 + 8 [0]
<idle> [000] dNh6. 20.465627: sched_wakeup: comm=readone pid=131
<idle> [000] dNh1. 20.465648: irq_handler_exit: irq=25 ret=handled
readone [000] ..... 20.466359: block_rq_issue: 253,0 RA 4096 () 164216 + 8 [readone]
... the same pattern for the data block, then read() returns 20 bytes
```

<idle>

the CPU was idle (hlt) when the disk interrupted: nobody waited actively

d

interrupts disabled on this CPU (IF = 0)

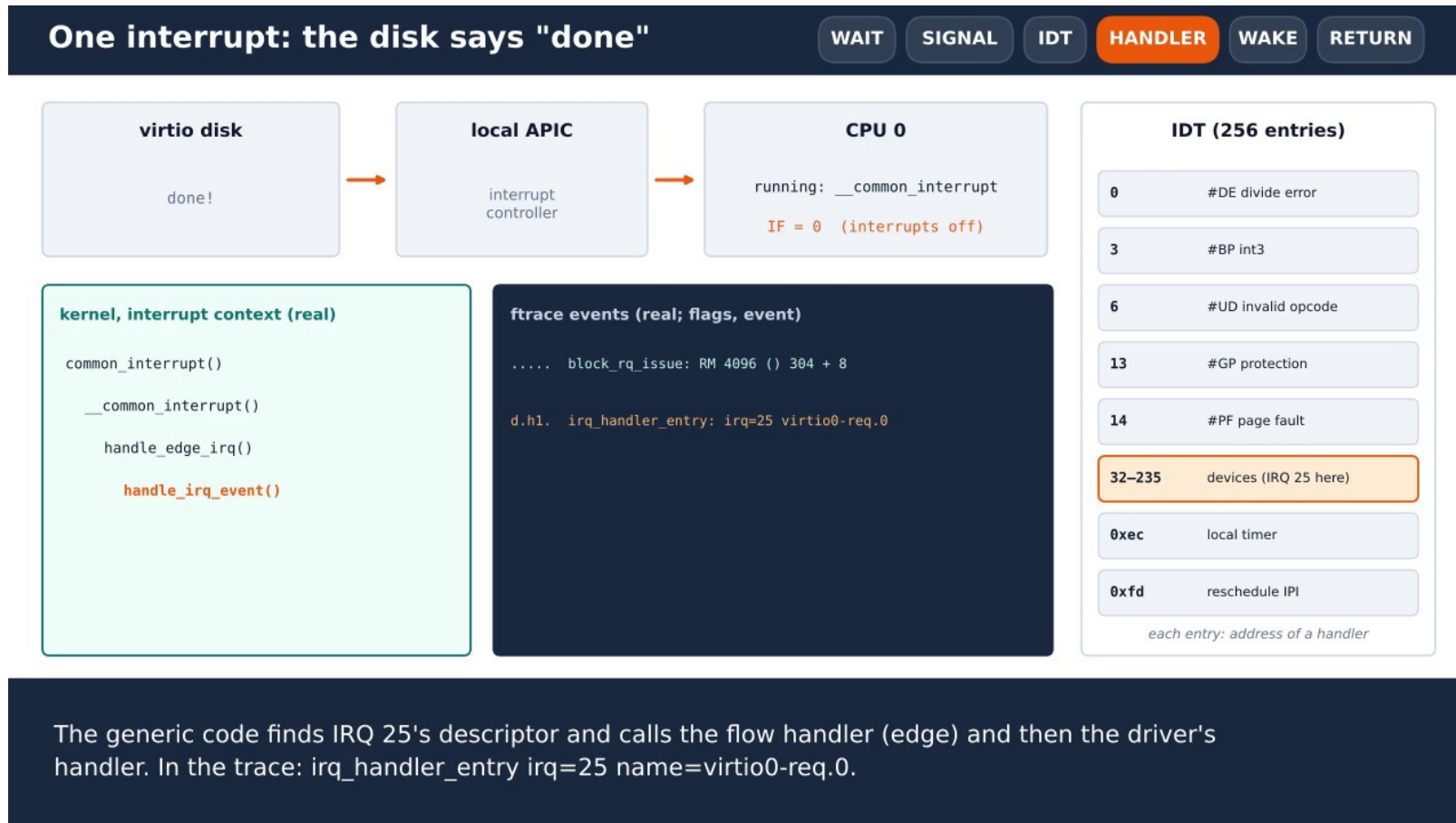
h

running in hard-interrupt context

N

"need resched": a better task (readone) is now runnable

Watch one interrupt, from the disk to the woken process



Click the video to play or pause. About 1.4 minutes, no sound.

Watch for

1. The CPU **sleeps** in hlt; no polling.
2. APIC → **vector** → IDT entry.
3. **IF = 0** while entering.
4. The driver's handler only **completes and wakes**.
5. **iret** or a switch to readone.

The driver's side: request_irq registers a handler

```
● ● ● include/linux/interrupt.h (real)

100 typedef irqreturn_t (*irq_handler_t)(int, void *);
164 static inline int __must_check
165 request_irq(unsigned int irq, irq_handler_t handler,
166             unsigned long flags, const char *name, void *dev)
167 {
168     return request_threaded_irq(irq, handler, NULL, ...);
}
```

```
● ● ● a handler, the usual shape (sketch)

static irqreturn_t mydev_irq(int irq, void *dev)
{
    struct mydev *d = dev;
    if (!(readl(d->regs + STATUS) & DONE))
        return IRQ_NONE; /* not mine: shared line */
    writel(DONE, d->regs + STATUS); /* acknowledge */
    wake_up(&d->waitq);
    return IRQ_HANDLED;
}
```

```
● ● ● include/linux/irqreturn.h (real)

12  IRQ_NONE      = (0 << 0),
13  IRQ_HANDLED   = (1 << 0),
14  IRQ_WAKE_THREAD = (1 << 1),
```

```
● ● ● QEMU guest trace (real)

irq_handler_entry: irq=4 name=ttyS0
irq_handler_exit: irq=4 ret=unhandled
```

SAME AS C

A callback, like `signal()`

You give the kernel a function pointer and a void * to pass back, as with `qsort`'s `compare` or `pthread_create`. Lecture 14 writes kernel modules; Lecture 15 a driver that uses this.

Book: Love, LKD Ch. 7 (registering an interrupt handler)

PART 4

Top halves and bottom halves

Do the urgent part now, with interrupts off; do the rest a little later.

Interrupt context: be fast, never sleep

in a hard interrupt handler	why
no sleeping: no mutex_lock, no msleep	"current" is just whoever was interrupted
no copy_to_user / copy_from_user	they can page-fault and sleep; and whose memory would it be?
kmalloc only with GFP_ATOMIC	GFP_KERNEL may wait for memory to be freed
keep it short (microseconds)	IRQs are off on this CPU: other devices must wait

Analogy: a restaurant. The waiter takes your order at once and writes it on a slip (top half: fast, must not be missed). The kitchen cooks it a few minutes later (bottom half: the long work). If the waiter cooked every dish himself, the next customers would wait at the door.

1 Top half (the handler): acknowledge the device, take the data or the status, schedule the rest. Interrupts off.

2 Bottom half: everything else, later, with interrupts on: softirqs, tasklets, workqueues, threaded IRQs.

Same as Lecture 10.2: a signal handler only sets a flag; an interrupt handler does only the minimum. Same reason: it interrupts code at an unknown point.

Book: Love, LKD Ch. 7 (interrupt context), Ch. 8 (bottom halves)

Softirqs: raised in the handler, run right after it

● ● ● include/linux/interrupt.h (real)

```
550 HI_SOFTIRQ=0,  
551 TIMER_SOFTIRQ,  
552 NET_TX_SOFTIRQ,  
553 NET_RX_SOFTIRQ,  
554 BLOCK_SOFTIRQ,  
555 IRQ_POLL_SOFTIRQ,  
556 TASKLET_SOFTIRQ,  
557 SCHED_SOFTIRQ,  
558 HRTIMER_SOFTIRQ,  
559 RCU_SOFTIRQ,
```

● ● ● QEMU guest, 2 CPUs: events softirq_* and timer (real)

```
<idle> [000] d.h1. softirq_raise: vec=1 [action=TIMER]  
<idle> [000] d.h1. softirq_raise: vec=7 [action=SCHED]  
<idle> [000] ..s1. softirq_entry: vec=1 [action=TIMER]  
<idle> [000] ..s1. timer_expire_entry: ...  
                function=blk_stat_timer_fn  
<idle> [000] ..s1. softirq_exit: vec=1 [action=TIMER]  
<idle> [001] ..s1. timer_expire_entry: ...  
                function=process_timeout
```

raise: in the hard handler (d.h): just set a bit "TIMER pending"

run: when the hard handler returns: interrupts back ON (..s), still not a process

overload: if softirqs keep coming, the rest moves to a thread per CPU: ksoftirqd/0, ksoftirqd/1

/proc/softirqs counts them per CPU. process_timeout = a sleeping process's timer expired: the softirq wakes it.

Book: Love, LKD Ch. 8 (softirqs)

Four ways to do the rest later

mechanism	runs in	may sleep?	used for
softirq	interrupt context, right after the handler	no	network packets, timers, block completion: fixed list, very fast
tasklet	a softirq (TASKLET_SOFTIRQ)	no	older drivers; one function, never on two CPUs at once
workqueue	a kernel thread (kworker/...)	yes	most driver work: can take mutexes, allocate with GFP_KERNEL
threaded IRQ	its own kernel thread (irq/N-name)	yes	slow buses (I2C sensors), real-time kernels

● ● ● ps in the QEMU guest (real, shortened)

```
16 0 [ksoftirqd/0]
24 0 [ksoftirqd/1]
41 0 [irq/9-acpi]    # a threaded IRQ
9 0 [kworker/0:1-pm] # a workqueue worker
```

Rule of thumb for driver writers: need to sleep → workqueue (or a threaded IRQ). Must be fast and frequent → softirq or tasklet. The square brackets in ps mean kernel threads: no user memory at all.

Book: Love, LKD Ch. 8 (bottom halves and deferring work)

PART 5

Timers

The timer interrupt: ticks, jiffies, and sleeping exactly (almost).

The timer interrupt: the kernel's heartbeat

● ● ● arch/x86/kernel/apic/apic.c (real lines)

```
1076 DEFINE_IDTENTRY_SYSVEC(sysvec_apic_timer_interrupt)
1080     apic_eoi();
1081     trace_local_timer_entry(LOCAL_TIMER_VECTOR);
1082     local_apic_timer_interrupt();
```

● ● ● function_graph, QEMU guest (real, shortened)

```
__sysvec_apic_timer_interrupt() {
    hrtimer_interrupt() {
        ktime_get_update_offsets_now();
        __hrtimer_run_queues() {
            tick_nohz_highres_handler();    87.6 us
        }
        tick_program_event();    /* set the next one */
    } 186 us
```

● ● ● QEMU guest: measuring HZ (real)

```
jiffies 4294670395 -> 4294675437
uptime  3.59 s      -> 8.62 s
5042 ticks / 5.03 s = 1002 per second: HZ = 1000
```

The tick does: update jiffies and the time of day, charge CPU time to the running process (Lecture 8.1's scheduler), run expired timers (TIMER softirq).

jiffies: a counter of ticks since boot; kernel timeouts are in jiffies. It starts near $2^{32} - 300$ 000 on purpose, so wrap-around bugs show up in the first 5 minutes.

hrtimers: high-resolution timers in nanoseconds: nanosleep, timerfd, and the tick itself are hrtimers.

Book: Love, LKD Ch. 11 (timers and time management)

Sleep 1 ms: how long does it really take?

● ● ● slack.c (core lines)

```
struct timespec ms = { 0, 1000000 };      /* 1 ms */
for (i = 0; i < 1000; i++) {
    t = now_us(); nanosleep(&ms, NULL);
    late = now_us() - t - 1000; ... }
prctl(PR_SET_TIMERSLACK, 1, 0, 0, 0);     /* 1 ns */
```

● ● ● ./slack (test machine; real, two runs)

```
timer slack: 50000 ns
default slack (50 us) late by 99.3 us on average, worst 703.8 us
slack 1 ns           late by 52.1 us on average, worst 2143.9 us
(run 2: 106.5 us and 51.8 us)
```

1 nanosleep sets an **hrtimer**; the process sleeps

2 The kernel may fire it up to **timer slack** (50 μ s) late, to group wake-ups and let the CPU sleep longer

3 Then the timer interrupt, the wake-up and the scheduler add tens of μ s more

4 **Worst** cases are ms: other work, or the host of our virtual machine

C link: sleep(1) and nanosleep promise a minimum time, never an exact one.

Book: Kerrisk, TLPI Ch. 23 (timers and sleeping)

LAB 13

irqtop.c: a top for interrupts

Parse /proc/interrupts twice, one second apart, and sort the differences.

Lab 13 tasks, step by step

1 **TODO 1** in irqtop_starter.c: read the header line; count "CPU" with strstr → ncpu

2 **TODO 2**: for each line: the id before ':', the sum of up to ncpu numbers (strtoull), the name

3 **TODO 3**: match lines of the two snapshots by id (strcmp); count = after – before

4 **TODO 4**: qsort by count, biggest first; print the top 10

5 Run it idle; then type, copy a big file, run `yes > /dev/null`: which lines change?

Careful: the lines ERR and MIS have only ONE number, not one per CPU. strtoull tells you when it found no number: `end == p`.

```
● ● ● reference, real
$ ./irqtop           # test machine, idle
IRQ per second name
LOC          92 Local timer interrupts
CAL          25 Function call interrupts
              49          3 ... virtio7-input.0
$ ./irqtop 6         # while reading files
IRQ per second name
 37          138 ... virtio1-req.0
LOC          92 Local timer interrupts
CAL          26 Function call interrupts
$ ./irqtop           # QEMU guest, dd
 25          493 ... virtio0-req.0
LOC          351 Local timer interrupts
```

Checkpoint: test yourself

Q1 A page fault returns to which instruction? A system call? → the same one (fault, retried) / the next one (trap)

Q2 What does the CPU push on the kernel stack for an interrupt from user mode? → SS, RSP, RFLAGS, CS, RIP (+ error code for some exceptions)

Q3 exc printed trap 13 for hlt. Which exception, and which signal? → #GP, general protection → SIGSEGV

Q4 Why may an interrupt handler not call mutex_lock? → it may sleep, and there is no process to put to sleep

Q5 An idle CPU got 74 timer interrupts in 5 s with HZ = 1000. Why? → tickless idle (NO_HZ): no periodic tick when idle

Key ideas from today



Interrupts are async (devices, timers); traps, faults and aborts come from the instruction itself.



The IDT maps 256 vectors to handlers; the CPU pushes SS, RSP, RFLAGS, CS, RIP and switches stacks.



Device → APIC → vector → irq_desc → the driver's handler, which completes work and wakes processes.



Top half fast with IRQs off; softirqs, workqueues, threads later. The timer tick: HZ, tickless, hrtimers.

Homework 13 (due before Lecture 14.1)

1. Error codes

pferr.c makes 4 page faults: write *NULL, call address 0, write into main's code, read a kernel address. Predict each err value with trap_pf.h; run it; explain any surprise (fault.c, sanitize_error_code).

2. Ticks

On a lab PC, run irqtop idle and while one `yes > /dev/null` runs per CPU. Compare LOC with $\text{HZ} \times \text{CPUs}$ (grep CONFIG_HZ=/boot/config-\$(uname -r)). Explain.

3. Timer slack

Run slack.c on a lab PC. Why is the average about 50 μs smaller with slack 1 ns? Why does the kernel add slack at all?

4. irqtop, live

Make irqtop refresh every second until Ctrl-C, like top: clear the screen with "\033[H\033[J", and stop cleanly with a SIGINT handler that only sets a flag.

5. Read the source

Follow #DE from the IDT to your SIGFPE: idt.c, traps.c (exc_divide_error, do_error_trap, do_trap). Why does int3 give si_code 128 but ud2 gives 2?

Next lecture and reading

NEXT: LECTURE 14.1

Loadable kernel modules

Your first code inside the kernel: `module_init` and `module_exit`, `printk` and `dmesg`, `insmod`, `lsmod` and `rmmod`, module parameters, and why we do it inside a virtual machine.

Main text

R. Love, **LKD**, Ch. 7–8 and 11 today; Ch. 17 (devices and modules) for 14.1.

Driver book

Corbet, Rubini & Kroah-Hartman, **Linux Device Drivers**, 3rd ed. (free online), Ch. 2 (building and running modules).

CS:APP

§8.1 (exceptions) today; the kernel module chapters build on Lectures 7–13.

Lab files

`lab13_files.zip`: `irqtop_starter.c`, `exc.c`, `slack.c`, `pferr.c`.