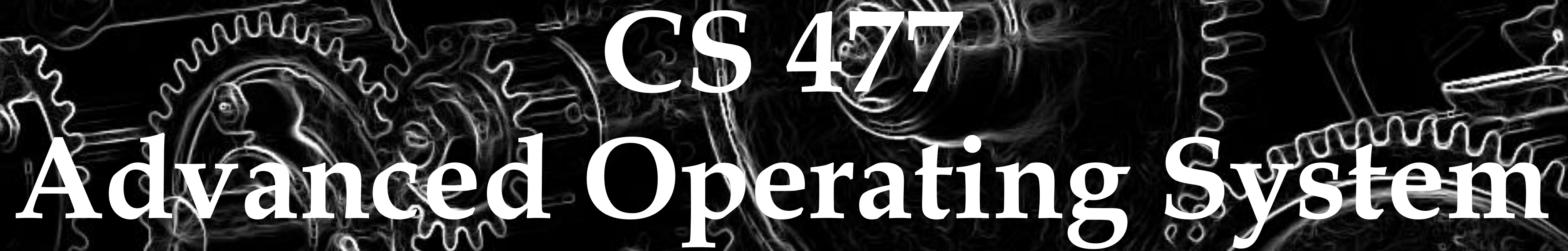


Lecture 08: Interrupts

EPFL

Today's agenda: Interrupts

- The problem: Why do we need interrupts?
- Interrupts architecture: Hardware and kernel path
- **The Top Half:** Immediate interrupt handling (ISR)
- **The Bottom Half:** Deferring work
  - Softirqs
  - Work Queues
  - Threaded IRQs
- Summary and control: choosing the right tool

The problem: CPU vs. IO

- CPUs are fast
- IO devices are slow
  - Example: A 7200 RPM Hard Disk Drive (HDD) has an average access time (seek + rotation) of ~10 msec
  - In that time, a CPU can execute *millions* of instructions
- **Polling:** The kernel periodically checks hardware status
  - Extremely inefficient and wastes CPU cycles
- **Solution: Interrupts**
  - Hardware signals the CPU when it *needs* attention
  - Kernel is free to do other work until the signal arrives

EPFL

Interrupt architecture (hardware)

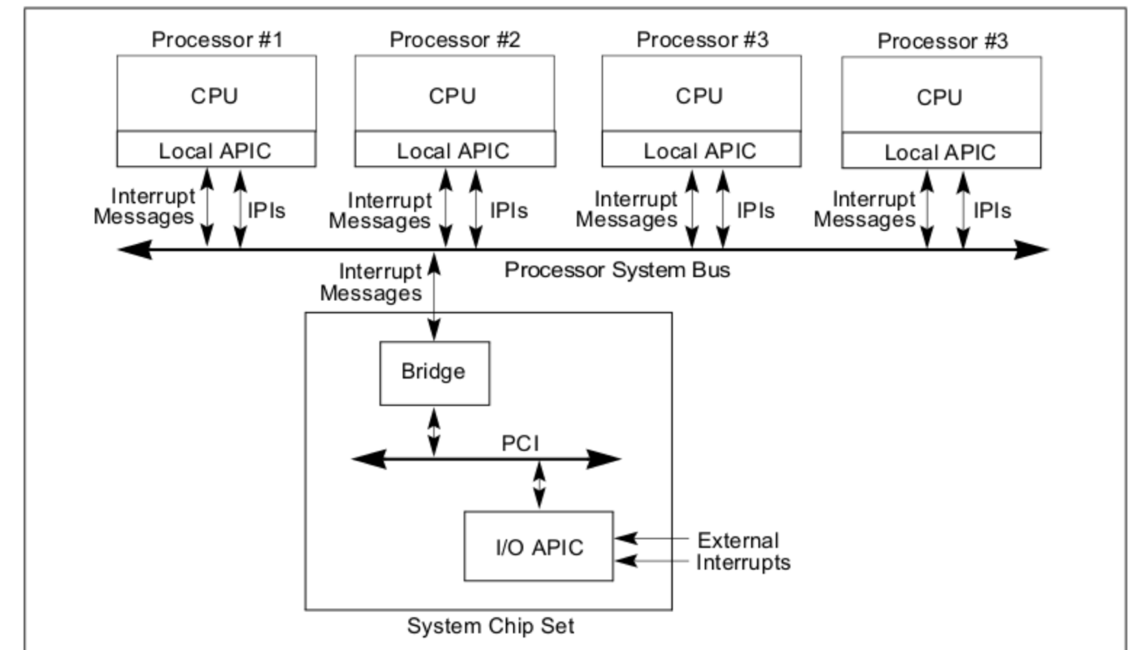
- Devices send electrical signals (interrupts) to an **Interrupt Controller**
  - Sent to a specific pin of the CPU
- Controller **multiplexes** these signals & sends a single signal to the CPU
- CPU stops its current work to execute a dedicated function → **Interrupt handler**
- Kernel/user space can be interrupted at (nearly) any time to process an interrupt

EPFL

Modern architecture (APIC)

- **IO APIC:** Part of the system chipset
  - Redistribute interrupts to local APICs

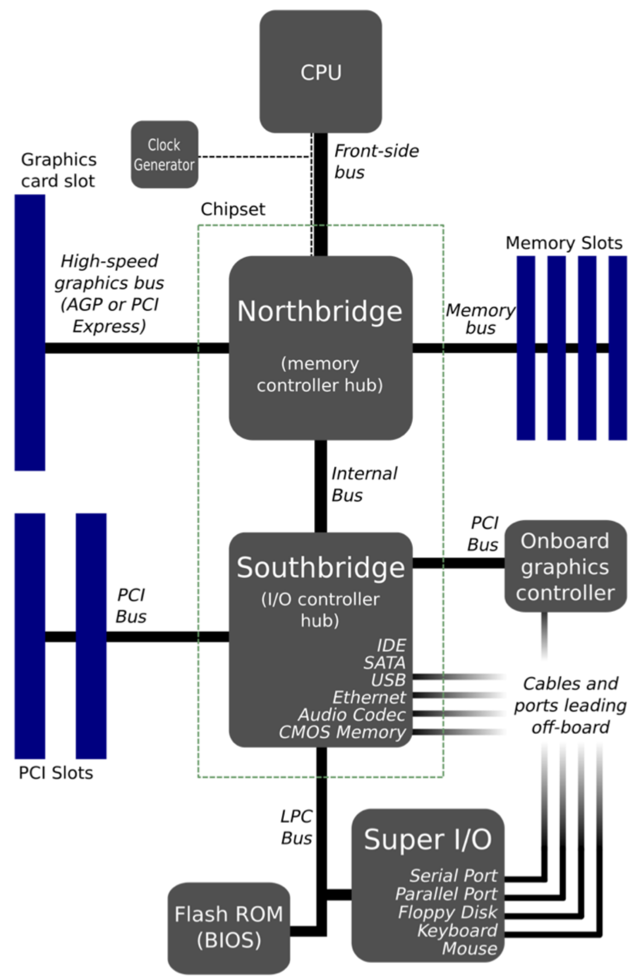
- **Local APIC (LAPIC):** Inside each CPU core
  - Receive interrupt messages from the IO APIC
  - Also receives **IPIs** (inter-process interrupts)
    - CPU-to-CPU interrupt

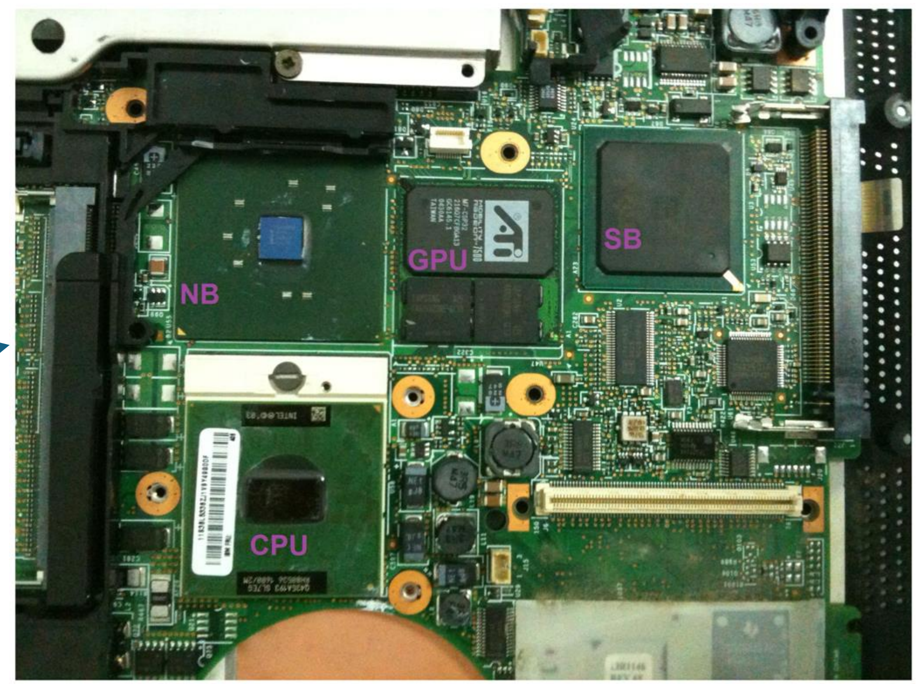


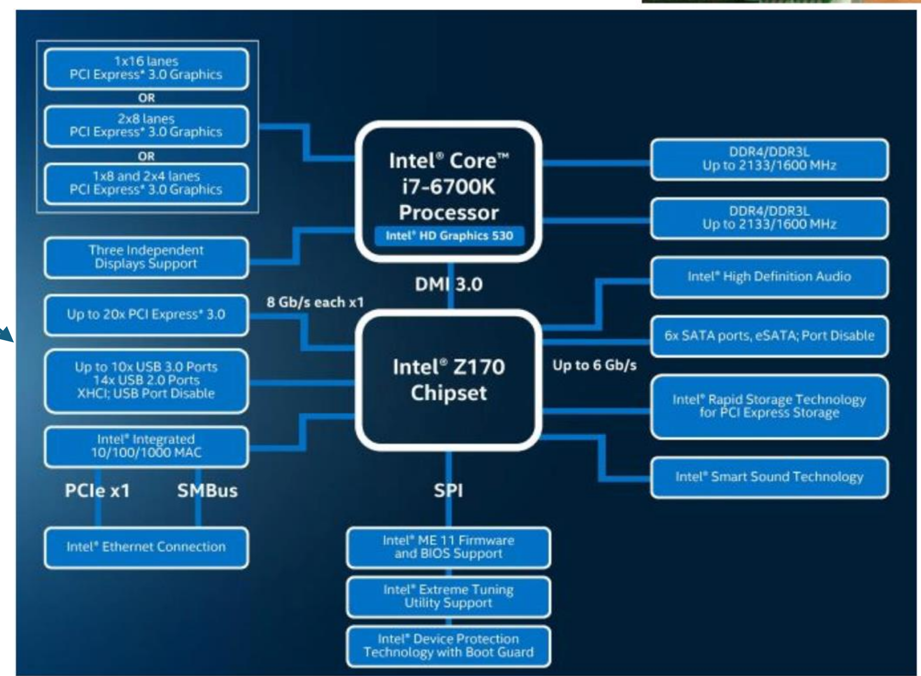
APIC: Advanced programmable interrupt controller

EPFL

CPU/chipset interaction







Devices (+ CPU) generate interrupt requests

- Interrupt line or interrupt request (IRQ)
  - Device identifier
- Eg., 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)

EPFL

CPUs also generate interrupts: Exceptions

- **Exceptions** are interrupts issued by the CPU executing some code
  - **Software interrupts,** as opposed to hardware ones (devices)
  - E.g.,
    - **Program faults**: divide-by-zero, page fault, general protection fault, etc.
    - **Voluntary exceptions**: `int` assembly instruction, for example, for syscall invocation
- *Kernel manages exceptions* in the same way as hardware interrupts

EPFL

Types of interrupts (how CPUs react to interrupts)

- **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
  - Enable/disable interrupts (**`sti`**: set interrupt; **`cli`**: clear interrupt)

- **INTA**
  - Interrupt acknowledgement
  - EOI (end of interrupt)

EPFL

Software interrupts: `int`

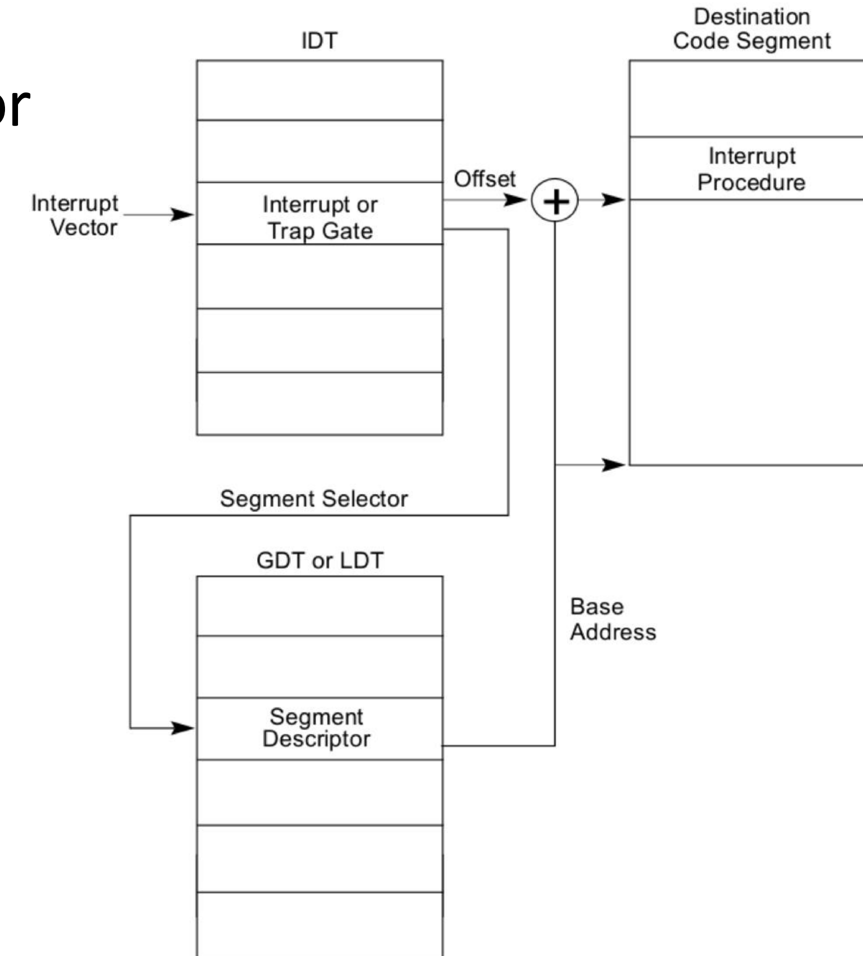
- **Intentional interrupts**
  - Invokes interrupt handler for vectors (0-255)
  - Enter: `int N`
  - Exit: `iret`
- Interrupt vector:
  - `int N`: **N**th interrupt handler (e.g., `int 0` → `vector 0`)

Q. Where is the information about all interrupts stored?

EPFL

Interrupt descriptor table: Dispatching an interrupt

1. Interrupt occurs → CPU receives interrupt vector
   - 0x20 = timer interrupt, 0x21 = keyboard interrupt
2. Vector used as an index into **`IDT`**
   - Table of 256 8-byte entries
   - **`IDTR`** stores the current **`IDT`** address
     - **`lidt`** instruction loads **`IDTR`** with the address and size of the **`IDT`**
   - **`IDT`**[vector] → contains:
     - Segment selector
     - Offset (handler address)
     - Type (interrupt/trap gate)

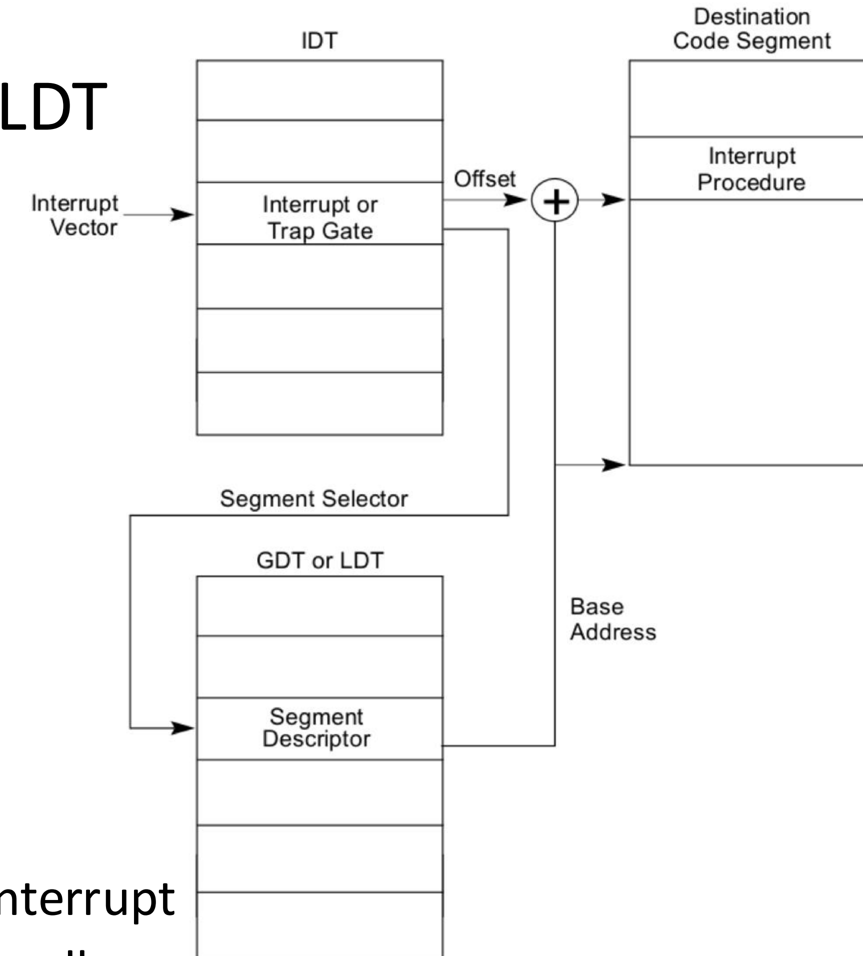


EPFL

Interrupt descriptor table: Dispatching an interrupt

3. Segment selector → locate segment in GDT/LDT
   - GDT/LDT give base address of code segment
4. Compute the actual handler address

   Handler Address = Base Address + Offset
5. CPU jumps to interrupt handler
   - Also known as **interrupt service routine**
   - Save relevant information on stack
   - Switches to kernel mode
   - Executes interrupt handler code
     - Function executed by the CPU in response to a specific interrupt
     - A C function matching a specific prototype to pass the handler information



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

How the kernel processes interrupts

Example: **`int 0x80`**

- Fetch interrupt descriptor vector 0x80 from IDT
  - 0x80th 8-byte entry starting at the physical address that IDTR points to
- Check CPL <= DPL in the descriptor
- Save ESP and SS in a CPU-internal register (but only if target segment selector’s PL < CPL)

EPFL

How the kernel processes interrupts

- 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

EPFL

Conflicting goals of interrupt service routine (ISR)

1. **Interrupt processing must be fast**
    - Interrupts other tasks (user vs. kernel)
    - Other interrupts may be disabled while processing the interrupt

2. **It may have significant work**
    - E.g., processing a network packet involves several steps

Q. What should the kernel do to handle such conflicting situations?

EPFL

Solution: Split the work

Modern OS split the interrupt processing into two parts:

1. **Top half**
    - Runs immediately upon receiving an interrupt
    - Performs only time-critical work
    - E.g., acknowledge the NIC, copy the packet to buffer, notify to receive again
2. **Bottom half**
    - Runs later with interrupts enabled
    - Performs less-critical, time-consuming work
        - Uses **softirq**, **work-queues** (similar to thread pool) for extra work
    - E.g., Process the network packet's contents, route it, etc..

Q. Does the kernel need to ensure some specific properties for ISR?

EPFL

Interrupt context

- The top half (ISR) runs in **interrupt context** (or “atomic context”)
- It is not a normal process; it has no process context
- Small stack size (4KB)

- **GOLDEN RULE:** it **CANNOT SLEEP** (or block)
  - No mutexes, no **`kmalloc(..., GFP_KERNEL)`**
  - Use **spinlocks** for concurrency and **`GFP_ATOMIC`** for memory allocation
  - No **`printk`**, use **`trace_printk`**

EPFL

Registering a top half

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

/**
 *  This call allocates interrupt resources and enables the
 *  interrupt line and IRQ handling.
 *
 *  @irq: Interrupt line to allocate
 *  @handler: Function to be called when the IRQ occurs.
 *        Primary handler for threaded interrupts
 *  @irqflags: Interrupt type flags
 *        IRQF_SHARED - allow sharing the irq among several devices
 *        IRQF_TIMER - Flag to mark this interrupt as timer interrupt
 *        IRQF_TRIGGER_* - Specify active edge(s) or level
 *  @devname: An ascii name for the claiming device
 *  @dev_id: A cookie passed back to the handler function
 *        Normally the address of the device data structure
 *        is used as the cookie.
 */

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

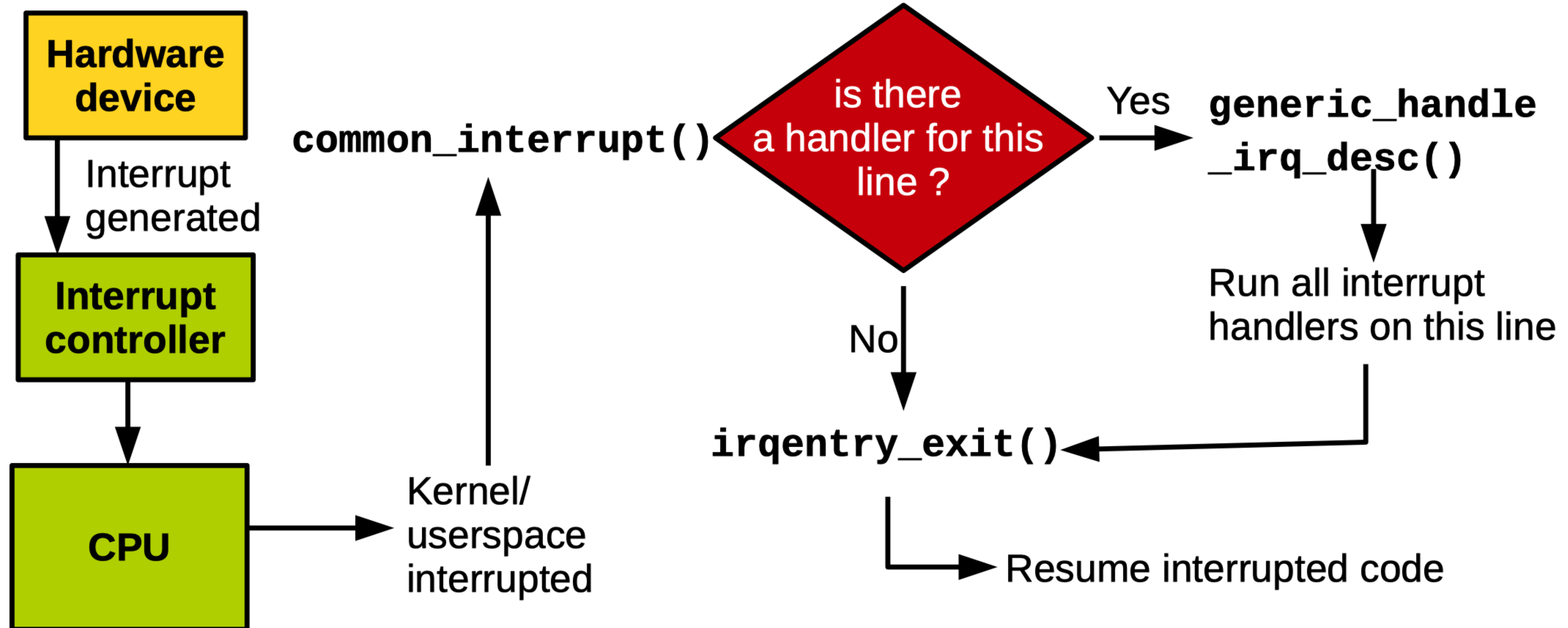
ISR handler prototype

```c
/* 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 hanlder was
 *        correctly invoked)
 *      #define IRQ_RETVAL(x)   ((x) ? IRQ_HANDLED : IRQ_NONE)
 *
 * NOTE: interrupt handlers need not be reentrant (tread-safe)
 *      - When a given interrupt handler is executing, the corresponding
 *      interrupt line is disabled on all cores while.
 */
typedef irqreturn_t (*irq_handler_t)(int irq, void *dev_id);
```

EPFL

Linux interrupt flow



The interrupt journey

1. **Hardware IRQ** → IDT lookup → assembly entry
2. common_interrupt() → IRQ resolution
3. Flow handler → event handling chain
4. **Device driver handler** executes
5. Threaded handler wakeup
6. Return via **`iretq`** → resume execution

EPFL

Stage 1: Hardware to assembly

The entry sequence

Hardware IRQ signal

CPU queries IDT

Jump to vector entry

`asm_common_interrupt` (assembly)

Save registers, clear state, prepare for C code transition

Location

arch/x86/entry/entry_64.S

IDT Setup

arch/x86/kernel/idt.c

EPFL

Stage 2: C entry point

Transition to C code

```
common_interrupt()
↓
desc = vector_irq[vector]
↓
handle_irq(desc, regs)
```

Location

arch/x86/kernel/irq.c

Direct vector-to-descriptor lookup, then dispatch to generic IRQ handling

EPFL

Stage 3: IRQ descriptor resolution

Finding the right handler

```
common_interrupt()
↓
desc = vector_irq[vector] (per-CPU)
↓
handle_irq(desc, regs)
↓
generic_handle_irq_desc(desc)
```

Location

kernel/irq/irqdesc.c

Next step

desc→handle_irq(desc)

Per-CPU vector array maps hardware interrupt to IRQ descriptor

Stage 4: Flow handlers

- High-level interrupt type managers
  - Handle different behaviors of hardware interrupt controllers

Q. Why do we need different handlers?

- **Hardware variety:** Different interrupt controllers work differently
- **Edge vs. level:** Signals behave differntly (pulse vs. sustained)
- **Ack timing:** When to acknowledge the interrupt to the chip
- **Masking policy:** Whther to disable the IRQ line during handling

Stage 4: Flow handler types

- `handle_edge_irq()`
  - Interrupts occur on signal transitions (0→1); can be lost if not handled quickly
  - E.g., PCI MSI
- `handle_level_irq()`
  - Interrupts stay active until cleared by the device
  - Must mask IRQ during handling to prevent storms
- `handle_fasteoi_irq()` **(most common)**
  - Used by APIC; sends EOI (end-of-interrupt) after handling; no masking reqd
- `handle_percpu_irq()`
  - timer-interrupts, IPIs
  - No locking required since they are CPU-local

EPFL

Stage 4: Flow handler in action

Example: `handle_fasteoi_irq()`

```
void handle_fasteoi_irq(struct irq_desc *desc) {

 // 1. Mask if needed (rare)
 if (desc->status & IRQ_MASKED)
  mask_ack_irq(desc);
 // 2. Call device handlers
 handle_irq_event(desc);
 // 3. Send EOI to chip
 chip->irq_eoi(&desc->irq_data);
}
```

Key point
Flow handler wraps device handler call and manages chip protocol

EPFL

Stage 5: The action chain

- Action chain is a **linked list of handlers** registered for a single IRQ
  - Shared interrupts

Q. Why do we need to share IRQs?

- **Limited IRQ lines:** Old systems (ISA had only 16 IRQs)
- **PCI devices:** Multiple devices share the same interrupt line
- **Resource constraints:** More devices than available IRQs

Structure

`irq_desc → action → action → action → NULL`

Each irqaction represents one driver's handler

EPFL

Stage 5: The action chain call path

Invoking device handlers

```
handle_irq_event(desc) → handle_irq_event_percpu(desc)
↓
for_each_action_of_desc() → action→handler(irq, dev_id)
```

Iterates handler chain, supports shared IRQs

Returns **`IRQ_HANDLED, IRQ_NONE, or IRQ_WAKE_THREAD`**

Location: kernel/irq/handle.c

Stage 6: Device handler execution

Your driver code runs here

```
irqreturn_t handler(int irq, void *dev_id) {
 // Read device registers
 // Clear interrupt source
 // Process data
 return IRQ_HANDLED;
}
```

Registered via request_irq() or request_threaded_irq(). Keep handler fast and minimal.

EPFL

Stage 7: Threaded interrupts

Deferred processing model

> Primary handler returns **IRQ_WAKE_THREAD**
> ↓
> Wake kernel thread: irq/N-devicename
> ↓
> action→thread_fn(irq, dev_id) executes

Thread can sleep, block, use mutexes

Better real-time characteristics, reduces hardirq time

We will see more on this in bottom half!

EPFL

Stage 8: Return path

Unwinding the stack

Handler returns → EOI sent to chip

`generic_handle_irq_desc()` returns

`common_interrupt()` returns

Assembly: RESTORE_REGS

iretq - Resume execution

EPFL

Initializing IDT: Data structures

```
/* linux/arch/x86/include/asm/desc_defs.h */
struct gate_struct {
    u16     offset_low;
    u16     segment;
    struct idt_bits bits;
    u16     offset_middle;
#ifdef CONFIG_X86_64
    u32     offset_high;
    u32     reserved;
#endif
} __attribute__((packed));

typedef struct gate_struct gate_desc;

/* linux/arch/x86/kernel/traps.c */
DECLARE_BITMAP(system_vectors, NR_VECTORS);
```

- Hardware defined struct that CPU uses for reading IDT
- Each entry describes where to jump when an interrupt occurs

Initializing IDT: Assembly entry points setup

```
/* linux/arch/x86/include/asm/idtentry.h
/*
 * Build the entry stubs with some assembler magic.
 * We pack 1 stub into every 8-byte block.
 */
    .align 8
SYM_CODE_START(irq_entries_start)
    vector=FIRST_EXTERNAL_VECTOR
    .rept (FIRST_SYSTEM_VECTOR - FIRST_EXTERNAL_VECTOR)
    UNWIND_HINT_IRET_REGS
0 :
    .byte   0x6a, vector
    jmp asm_common_interrupt
    nop
    /* Ensure that the above is 8 bytes max */
    . = 0b + 8
    vector = vector+1
    .endr
SYM_CODE_END(irq_entries_start)
```

- Linux generates **compact array of stub functions**: one for each interrupt vector
    - Pushes its vector onto the stack
    - Jumps to the common asm_common_interrupt code

EPFL

Initializing IDT: Early boot IDT

```c
/* linux/init/main.c */

asmlinkage __visible void __init start_kernel(void)
{
    /* ... */
    early_irq_init();
    init_IRQ();
    /* ... */
}

/* linux/arch/x86/kernel/irqinit.c */
void __init init_IRQ(void)
{
    int i;
    for (i = 0; i < nr_legacy_irqs(); i++)
        per_cpu(vector_irq, 0)[ISA_IRQ_VECTOR(i)] = irq_to_desc(i);

    BUG_ON(irq_init_percpu_irqstack(smp_processor_id()));

    x86_init.irqs.intr_init();
}
```

- Minimal IDT set up to catch faults (page faults etc.)
- Legacy IRQs (0-15) and per-CPU IRQ stacks are initialized

Initializing IDT: Full IRQ initialization

```
/* linux/arch/x86/kernel/irqinit.c */
void __init native_init_IRQ(void)
{
    /* Execute any quirks before the call gates are initialised: */
    x86_init.irqs.pre_vector_init();

    idt_setup_apic_and_irq_gates();
    lapic_assign_system_vectors();

    if (!acpi_ioapic && !of_ioapic && nr_legacy_irqs())
        setup_irq(2, &irq2);
}
```

- Install all interrupt handlers
- Assign special vectors (timer, spurious, etc.)
- Setup legacy PIC if no APIC

EPFL

Initializing IDT: Populate IDT from table

```
/* linux/arch/x86/kernel/idt.c */
void __init idt_setup_apic_and_irq_gates(void)
{
    int i = FIRST_EXTERNAL_VECTOR;
    void *entry;

    idt_setup_from_table(idt_table, apic_idts, ARRAY_SIZE(apic_idts), true);

    for_each_clear_bit_from(i, system_vectors, FIRST_SYSTEM_VECTOR) {
        entry = irq_entries_start + 8 * (i - FIRST_EXTERNAL_VECTOR);
        set_intr_gate(i, entry);
    }
}
```

- Iterate through vectors 32-255 and skip already assigned vectors
- Point each IDT entry to its stub in the **`irq_entries_start`**

EPFL

Initializing IDT: Set individual gates

```
/* linux/arch/x86/kernel/idt.c */
static void set_intr_gate(unsigned int n, const void *addr)
{
	struct idt_data data;

	BUG_ON(n > 0xFF);

	memset(&data, 0, sizeof(data));
	data.vector = n;
	data.addr   = addr;
	data.segment    = __KERNEL_CS;
	data.bits.type  = GATE_INTERRUPT;
	data.bits.p = 1;

	idt_setup_from_table(idt_table, &data, 1, false);
}
```

Sets up only for 224 interrupts? Why not 255?

0-31 are CPU exceptions (Intel reserved)
They do not use the device stub

- Takes a vector number and handler address
- Validates inputs (CS/SS, vector number)
- Formats data into hardware gate_desc structure
- Writes to the IDT
- Marks it as an **interrupt gate** (auto-disables interrupts)

EPFL

/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-fasteoi   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-fasteoi   virtio0
 21:       3142        533   IO-APIC  21-fasteoi   ahci[0000:00:0d.0],
        snd_intel8x0
 22:         27          0   IO-APIC  22-fasteoi   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

- **Goal:** Ensure **atomic execution** of an interrupt handler code
- **Guarantee:** Disable interrupts to avoid preemption of normal code
- **Side effects:** Also disables **kernel preemption** on the local core
  - Scheduler will not switch tasks
- **Critical limitation:** No protection against concurrent access from other CPUs
- **Full solution:** use **locking** for mutual exclusion in conjunction with **disabling interrupts**

EPFL

How to disable interrupts *correctly!*

Flawed method: `local_irq_disable/enable` are not nest safe

- If an inner function calls `local_irq_enable()`, it prematurely re-enables interrupts, breaking an outer function's critical section

Correct method: `local_irq_save()/local_irq_restore()`

```c
unsigned long flags;
local_irq_save(flags);    /* disables interrupts if needed */
/* ... */
local_irq_restore(flags); /* restore interrupt status to the previous */

/* nesting is okay */
unsigned long flags;
local_irq_save(flags);
{
  unsigned long flags;
  local_irq_save(flags);
  /* ... */
  local_irq_restore(flags);
}
local_irq_restore(flags);
```

- `local_irq_save(flags)`: **Saves** the current state (on *or* off) to flags, then disables
- `local_irq_restore(flags)`: **Restores** whatever state was saved in flags
- **Nest-safe** because if interrupts were already off, `local_irq_restore()` leaves them off

Bottom half: Running deferred work

Bottom half runs when system is less busy and interrupts are enabled

1. **Interrupt context**
   - Cannot sleep or block
   - Runs with high priority, can preempt process-context code
   - Mechanism: **softirq**
2. **Process context (thread)**
   - **Can sleep** (acquires mutexes, `kmalloc(GFP_KERNEL)`, etc.)
   - Runs as a normal kernel thread, subject to the scheduler
   - Mechanisms: **work queues**, **threaded IRQs**

History of bottom halves

- “Top half” and “bottom half” are generic terms, not specific to Linux
- Old “bottom half” (BH) mechanism
  - A statistically created list of 32 bottom halves
  - Globally synchronized
  - Easy-to-use yet inflexible and a performance bottleneck
- Task queues: queue of function pointers
  - Still too inflexible
  - Not lightweight enough for performance-critical subsystems (e.g., networking)

EPFL

Softirq (the high-performance way)

- A set of 32 statically defined, high-priority deferred tasks
  - `TIMER_SOFTIRQ` (Timers)
  - `NET_TX_SOFTIRQ / NET_RX_SOFTIRQ` (Networking)
  - `BLOCK_SOFTIRQ` (Block I/O)
  - `RCU_SOFTIRQ` (RCU locking)
- **Context:** Runs in **Interrupt Context** (cannot sleep)
- **Concurrency:** *Same* softirq **can run concurrently on multiple CPUs**
  - Requires mututal exclusion using spinlocks or use per-CPU data structures
- Almost never add a new softirq; use an existing one!

Using softirqs: assigning an index

```
/* linux/include/linux/interrupt.h */
enum {
    HI_SOFTIRQ=0,       /* [highest priority] high-priority tasklet */
    TIMER_SOFTIRQ,      /* timer */
    NET_TX_SOFTIRQ,     /* send network packets */
    NET_RX_SOFTIRQ,     /* receive network packets */
    BLOCK_SOFTIRQ,      /* block devices */
    IRQ_POLL_SOFTIRQ,   /* interrupt-poll handling for block device */
    TASKLET_SOFTIRQ,    /* normal priority tasklet */
    SCHED_SOFTIRQ,      /* scheduler */
    HRTIMER_SOFTIRQ,    /* unused */
    RCU_SOFTIRQ,        /* [lowest priority] RCU locking */

    YOUR_NEW_SOFTIRQ,   /* TODO: add your new softirq index */

    NR_SOFTIRQS         /* the number of defined softirq (< 32) */
};
```

Using softirqs: registering a handler

```c
/* linux/kernel/softirq.c */
/* register a softirq handler for nr */
void open_softirq(int nr, void (*action)(struct softirq_action *))
{
    softirq_vec[nr].action = action;
}

/* linux/net/core/dev.c */
static int __init net_dev_init(void)
{
    /* ... */
    /* register softirq handler to send messages */
    open_softirq(NET_TX_SOFTIRQ, net_tx_action);

    /* register softirq handler to receive messages */
    open_softirq(NET_RX_SOFTIRQ, net_rx_action);
    /* ... */
}

static void net_tx_action(struct softirq_action *h)
{
    /* ... */
}
```

Using softirqs: raising a softirq

```
/* linux/include/linux/interrupt.h */
/* Disable interrupt and raise a softirq */
extern void raise_softirq(unsigned int nr);

/* Raise a softirq. Interrupt must already be off. */
extern void raise_softirq_irqoff(unsigned int nr);

/* linux/net/core/dev.c */
raise_softirq(NET_TX_SOFTIRQ);
raise_softirq_irqoff(NET_TX_SOFTIRQ);
```

- Softirqs are raised within interrupt handlers (top halves)
  - Interrupt handler performs basic hardware-related work, raises the softirq and then exits

Q. What if softirqs are raised faster than they can be processed?

EPFL

The `ksoftirqd` solution

System can live-lock, spending 100% of time processing softirqs and starving user-space tasks

Solution: `ksoftirqd`

- Per-processor kernel thread (`ksoftirqd/0`, `ksoftirqd/1` ...)
- Kernel detects being "overwhelmed" by softirqs, it wakes up ksoftirqd
- ksoftirqd runs pending softirqs in a normal **process context** with normal priority (nice 0)
  - Avoids starving user space tasks

```
└─$ ps ax -eo pid,nice,stat,cmd | grep ksoftirqd
    17    0 S     [ksoftirqd/0]
    27    0 S     [ksoftirqd/1]
    33    0 S     [ksoftirqd/2]
    39    0 S     [ksoftirqd/3]
```

Q. What is the main issue with softirqs?

EPFL

The modern approach: Process context

Problem: atomic context, latency, complexity of softirqs

- **Goal:** move deferred work into the process context
  - Code is *vastly* simpler to write
  - Handler **can sleep**, use mutexes, and perform blocking I/O
  - Handler is managed by the scheduler, respecting task priorities

- Mechanisms:
  1. **Work queues** (for general-purpose deferred work)
  2. **Threaded IRQs** (for work tightly coupled to an interrupt)

EPFL

Work queues

- Most common and flexible bottom-half mechanism
- Defers work into a pool of generic kernel threads (e.g., `kworker/n`)
  - Can create additional per-CPU workers, if needed
- Runs in **process context**
- **Can sleep**

EPFL

Work queues: defining work

- Define the work structure

```c
/* linux/include/workqueue.h */
struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;
};

typedef void (*work_func_t)(struct work_struct *work);
```

- Initialize the work

```c
/* Statically creating a work */
DECLARE_WORK(work, handler_func);

/* Dynamically creating a work at runtime */
INIT_WORK(work_ptr, handler_func);
```

EPFL

Using work queues: scheduling work

- From top half (or any context), call **`schedule_work()`**

```
/* Put work task in global workqueue (kworker/n) */
bool schedule_work(struct work_struct *work);
bool schedule_work_on(int cpu,
         struct work_struct *work); /* on the specified CPU */
```

- Queue work on already existing work queue

```
/* Queue work on a specified workqueue */
bool queue_work(struct workqueue_struct *wq, struct work_struct *work);
bool queue_work_on(int cpu, struct workqueue_struct *wq,
         struct work_struct *work); /* on the specified CPU */
```

EPFL

Threaded IRQs

- Tightly couples a dedicated kernel thread to a *specific IRQ line*
  - Replacement for the "ISR schedules work" pattern
  - Exposes request_threaded_irq() function
- Two handlers:
  1. **`handler`** (The "Top Half"):
     - Runs in **hard-IRQ context** (cannot sleep)
     - Should be *tiny* (e.g., acknowledge hardware, check status)
     - Returns IRQ_WAKE_THREAD to wake the bottom-half
  2. **`thread_fn`** (The "Bottom Half")
     - Kernel immediately wakes the dedicated IRQ thread
     - This function runs in **process context** and **can sleep**
- Simplifies driver logic by co-locating the top and bottom halves in one registration call

EPFL

Summary of bottom halves: Choosing the right tool

Mechanism	Context	Can sleep?	Concurrency	Modern use case
Softirq	**Interrupt**	**No**	**Yes** (Runs on multiple CPUs)	Core kernel (networking, timers). Avoid in drivers.
Work Queue	**Process**	**Yes!**	**Yes** (Normal thread)	**Default choice.** For any complex, non-urgent, or blocking work.
Threaded IRQ	**Process**	**Yes!**	**Yes** (Dedicated thread)	**Preferred IRQ handler.** For all IRQ-related work that can sleep.

EPFL

Disabling bottom halves

```
/* Disable softirq and tasklet processing on the local processor */
void local_bh_disable();

/* Eanble softirq and tasklet processing on the local processor */
void local_bh_enable();
```

- Disable Softirq processing on the local CPU
- These calls can be nested
- They do not disable work queues or threaded IRQs
  - Workqueues and threaded IRQs are just normal kernel threads, managed by the scheduler

EPFL

Further reading

- 0xAX: Interrupts and Interrupt Handling
- Moving interrupts to threads