SO2 Lecture 04 - Interrupts

View slides

Lecture objectives

- Interrupts and exceptions (x86)
- Interrupts and exceptions (Linux)
- Deferrable work
- Timers

What is an interrupt?

An interrupt is an event that alters the normal execution flow of a program and can be generated by hardware devices or even by the CPU itself. When an interrupt occurs the current flow of execution is suspended and interrupt handler runs. After the interrupt handler runs the previous execution flow is resumed.

Interrupts can be grouped into two categories based on the source of the interrupt. They can also be grouped into two other categories based on the ability to postpone or temporarily disable the interrupt:

- **synchronous**, generated by executing an instruction
- **asynchronous**, generated by an external event
- **maskable**
  - can be ignored
  - signaled via INT pin
- **non-maskable**
  - cannot be ignored
  - signaled via NMI pin

Synchronous interrupts, usually named exceptions, handle conditions detected by the processor itself in the course of executing an instruction. Divide by zero or a system call are examples of exceptions.

Asynchronous interrupts, usually named interrupts, are external events generated by I/O devices. For example a network card generates an interrupts to signal that a packet has arrived.

Most interrupts are maskable, which means we can temporarily postpone running the interrupt handler when we disable the interrupt until the time the interrupt is re-enabled. However, there are a few critical interrupts that can not be disabled/postponed.

Exceptions

There are two sources for exceptions:

- processor detected
  - **faults**
  - **traps**
  - **aborts**
- programmed
  - **int n**

Processor detected exceptions are raised when an abnormal condition is detected while executing an instruction.

A fault is a type of exception that is reported before the execution of the instruction and can be usually corrected. The saved EIP is the address of the instruction that caused the fault, so after the fault is corrected the program can re-execute the faulty instruction. (e.g page fault).

A trap is a type of exception that is reported after the execution of the instruction in which the exception was detected. The saved EIP is the address of the instruction after the instruction that caused the trap. (e.g debug trap).

Quiz: interrupt terminology

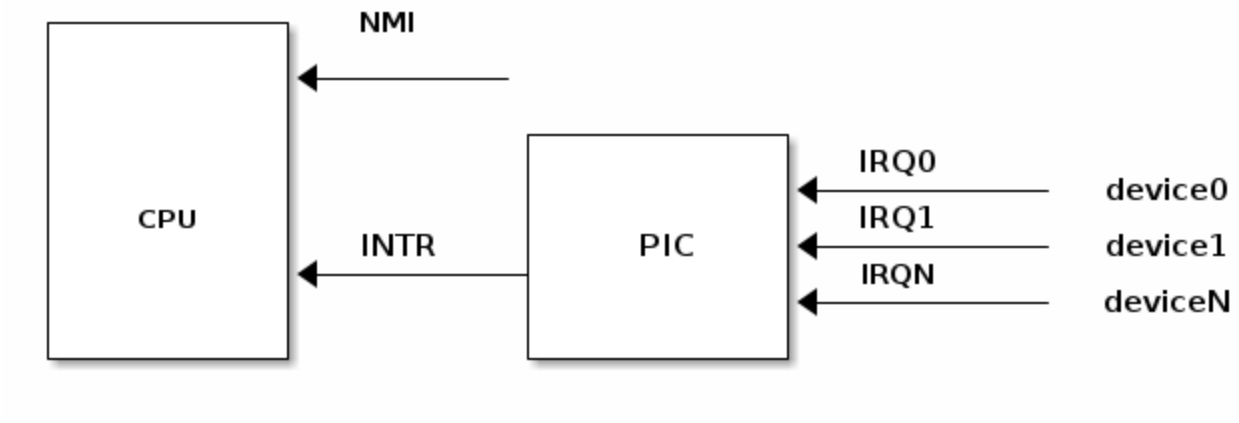
For each of the following terms on the left select all the terms from right that best describe them.

- Watchdog
- Demand paging
- Division by zero
- Timer
- System call
- Breakpoint

- Exception
- Interrupt
- Maskable
- Nonmaskable
- Trap
- Fault

Hardware Concepts

Programmable Interrupt Controller



A device supporting interrupts has an output pin used for signaling an Interrupt ReQuest. IRQ pins are connected to a device named Programmable Interrupt Controller (PIC) which is connected to CPU's INTR pin.

A PIC usually has a set of ports used to exchange information with the CPU. When a device connected to one of the PIC's IRQ lines needs CPU attention the following flow happens:

- device raises an interrupt on the corresponding IRQn pin
- PIC converts the IRQ into a vector number and writes it to a port for CPU to read
- PIC raises an interrupt on CPU INTR pin
- PIC waits for CPU to acknowledge an interrupt before raising another interrupt
- CPU acknowledges the interrupt then it starts handling the interrupt

Will see later how the CPU handles the interrupt. Notice that by design PIC won't raise another interrupt until the CPU acknowledged the current interrupt.

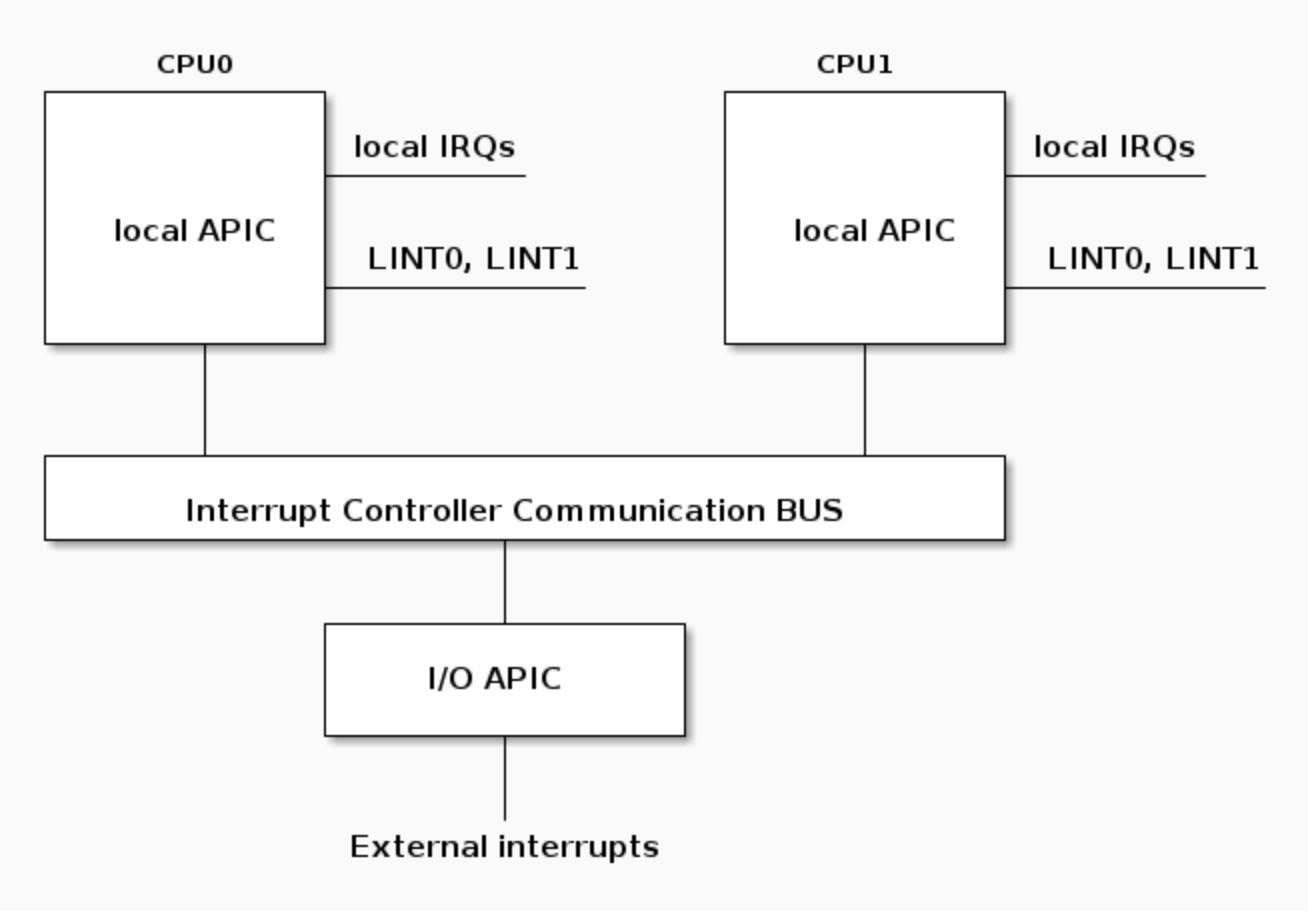
> **Note**
>
> Once the interrupt is acknowledged by the CPU the interrupt controller can request another interrupt, regardless if the CPU finished handled the previous interrupt or not. Thus, depending on how the OS controls the CPU it is possible to have nested interrupts.

The interrupt controller allows each IRQ line to be individually disabled. This allows simplifying design by making sure that interrupt handlers are always executed serially.

Interrupt controllers in SMP systems

In SMP systems we may have multiple interrupt controllers in the systems.

For example, on the x86 architecture each core has a local APIC used to process interrupts from locally connected devices like timers or thermals sensors. Then there is an I/O APIC is used to distribute IRQ from external devices to CPU cores.



Interrupt Control

In order to synchronize access to shared data between the interrupt handler and other potential concurrent activities such as driver initialization or driver data processing, it is often required to enable and disable interrupts in a controlled fashion.

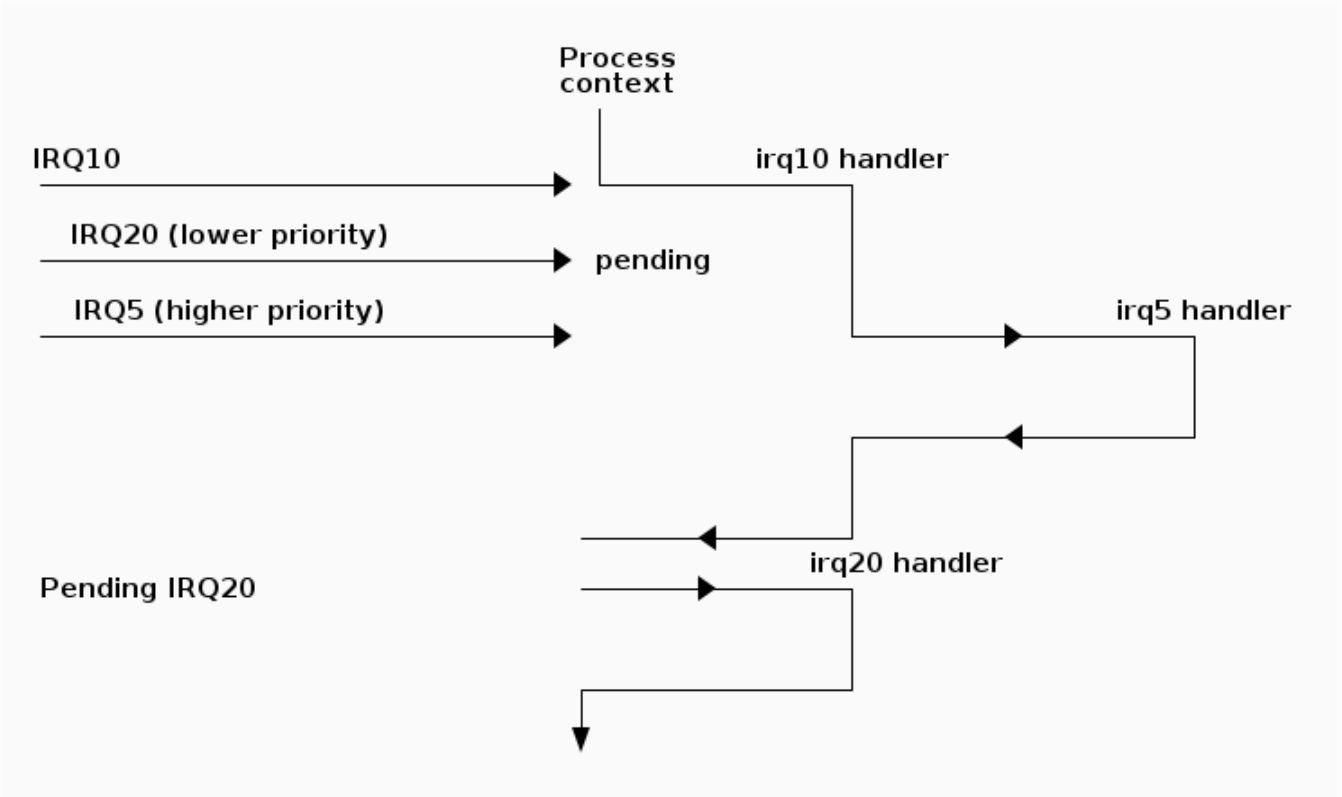
This can be accomplished at several levels:

- at the device level
  - by programming the device control registers
- at the PIC level
  - PIC can be programmed to disable a given IRQ line
- at the CPU level; for example, on x86 one can use the following instructions:

  - cli (CLear Interrupt flag)
  - sti (SeT Interrupt flag)

Interrupt priorities

Most architectures also support interrupt priorities. When this is enabled, it permits interrupt nesting only for those interrupts that have a higher priority than the current priority level.



Note

Not all architectures support interrupt priorities. It is also difficult to support defining a generic scheme for interrupt priorities for general use OSes and some kernels (Linux included) do not use interrupt priorities. On the other hand most RTOS use interrupt priorities since they are typically used in more constraint use-cases where it is easier to define interrupt priorities.

Quiz: hardware concepts

Which of the following statements are true?

- The CPU can start processing a new interrupt before the current one is finished
- Interrupts can be disabled at the device level
- Lower priority interrupts can not preempt handlers for higher priority interrupts
- Interrupts can be disabled at the interrupt controller level
- On SMP systems the same interrupt can be routed to different CPUs
- Interrupts can be disabled at the CPU level

Interrupt handling on the x86 architecture

This section will examine how interrupts are handled by the CPU on the x86 architecture.

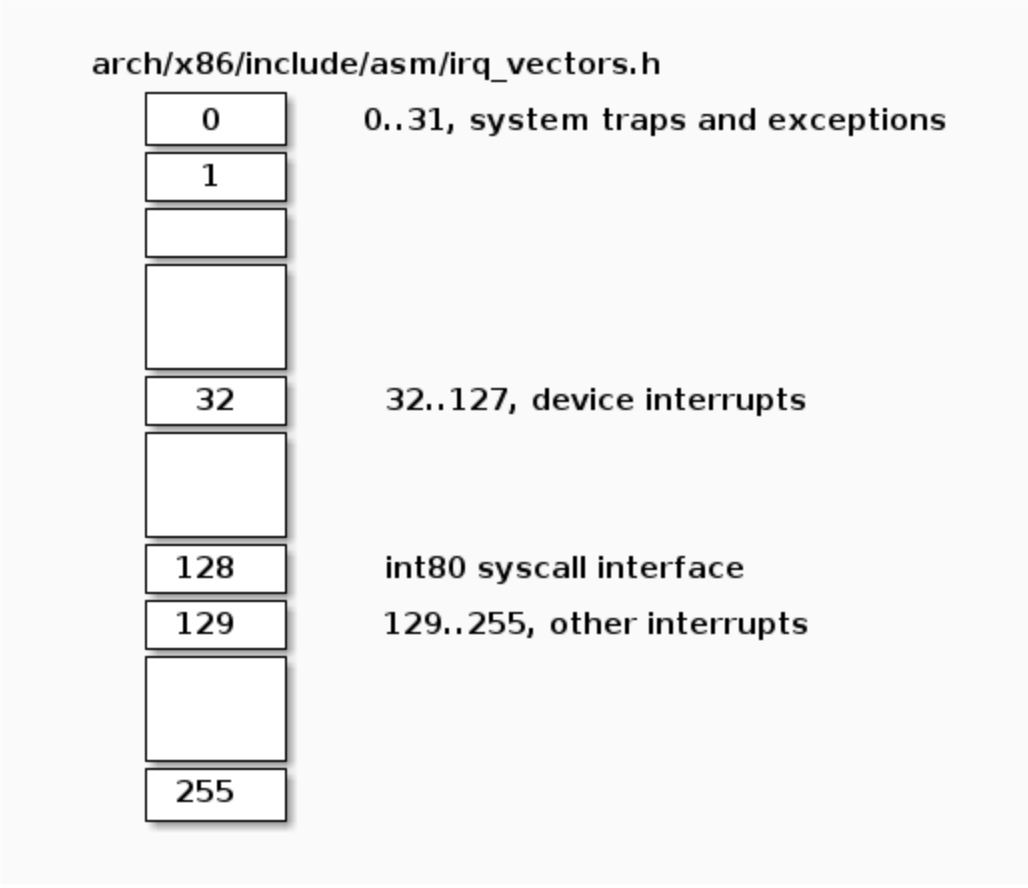
Interrupt Descriptor Table

The interrupt descriptor table (IDT) associates each interrupt or exception identifier with a descriptor for the instructions that service the associated event. We will name the identifier as vector number and the associated instructions as interrupt/exception handler.

An IDT has the following characteristics:

- it is used as a jump table by the CPU when a given vector is triggered
- it is an array of 256 x 8 bytes entries
- may reside anywhere in physical memory
- processor locates IDT by the means of IDTR

Below we can find Linux IRQ vector layout. The first 32 entries are reserved for exceptions, vector 128 is used for syscall interface and the rest are used mostly for hardware interrupts handlers.

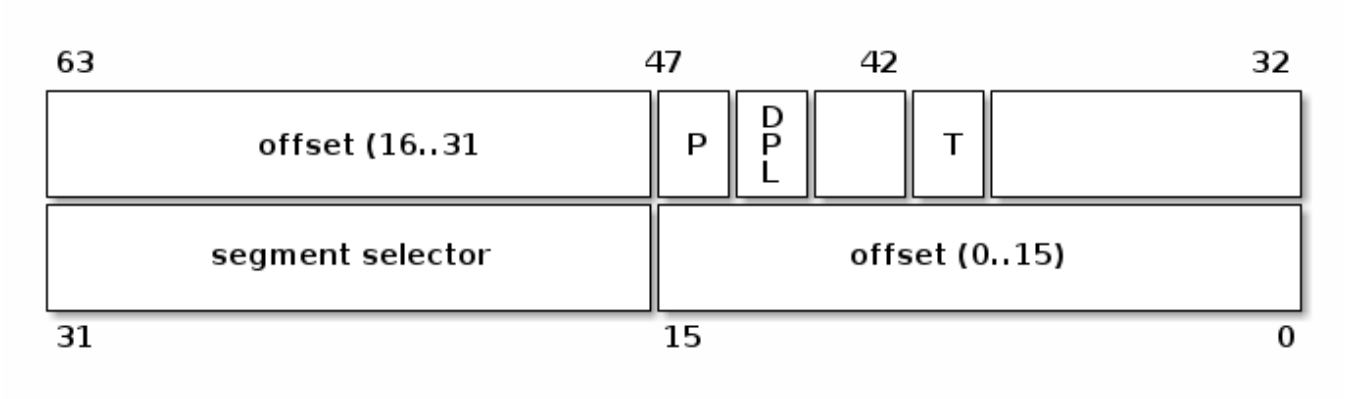


On x86 an IDT entry has 8 bytes and it is named gate. There can be 3 types of gates:

- interrupt gate, holds the address of an interrupt or exception handler. Jumping to the handler disables maskable interrupts (IF flag is cleared).
- trap gates, similar to an interrupt gate but it does not disable maskable interrupts while jumping to interrupt/exception handler.
- task gates (not used in Linux)

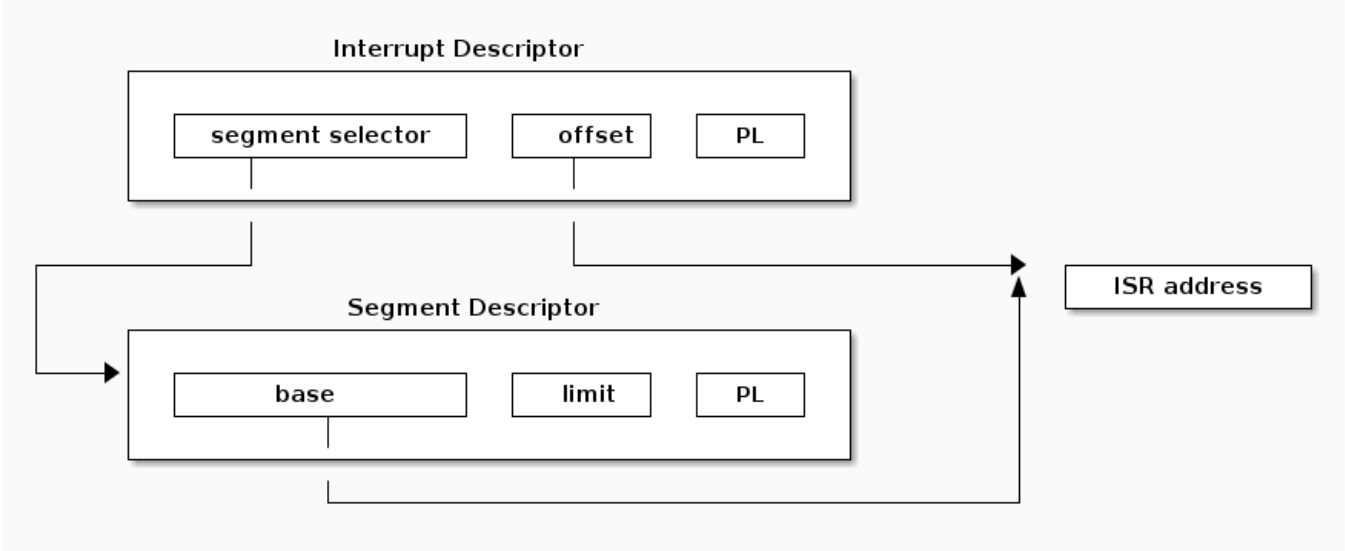
Let's have a look at several fields of an IDT entry:

- segment selector, index into GDT/LDT to find the start of the code segment where the interrupt handlers reside
- offset, offset inside the code segment
- T, represents the type of gate
- DPL, minimum privilege required for using the segments content.



Interrupt handler address

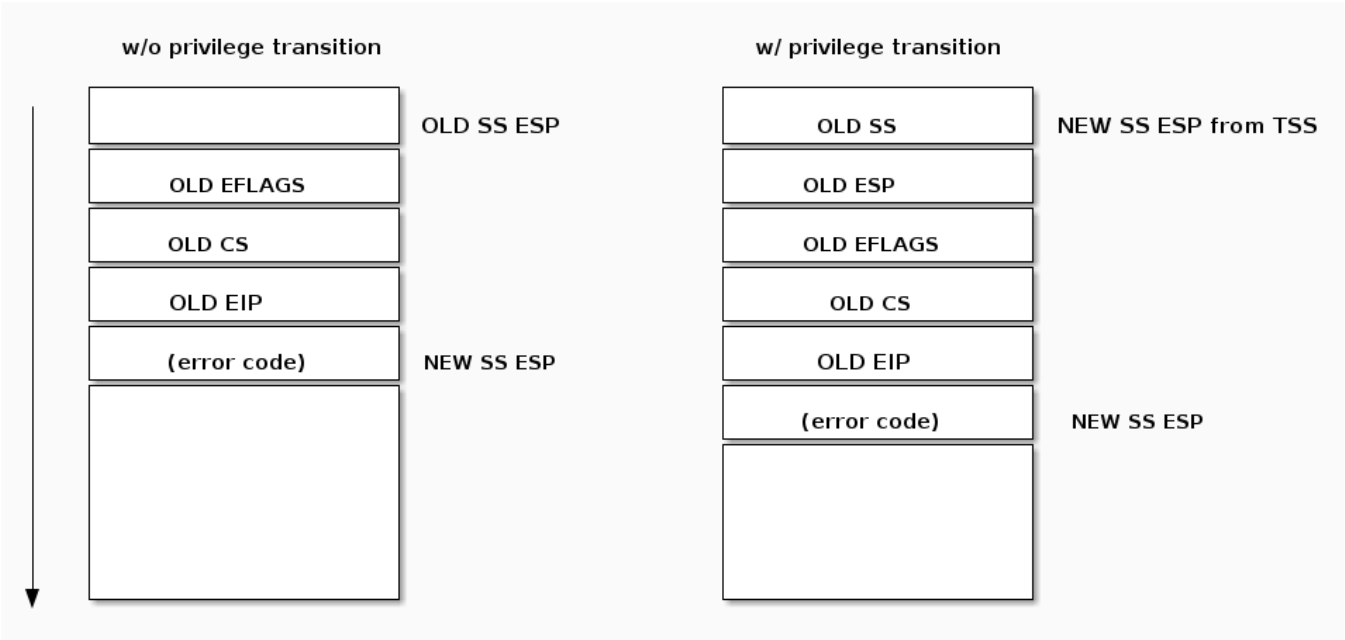
In order to find the interrupt handler address we first need to find the start address of the code segment where interrupt handler resides. For this we use the segment selector to index into GDT/LDT where we can find the corresponding segment descriptor. This will provide the start address kept in the 'base' field. Using base address and the offset we can now go to the start of the interrupt handler.



Stack of interrupt handler

Similar to control transfer to a normal function, a control transfer to an interrupt or exception handler uses the stack to store the information needed for returning to the interrupted code.

As can be seen in the figure below, an interrupt pushes the EFLAGS register before saving the address of the interrupted instruction. Certain types of exceptions also cause an error code to be pushed on the stack to help debug the exception.



Handling an interrupt request

After an interrupt request has been generated the processor runs a sequence of events that eventually end up with running the kernel interrupt handler:

- CPU checks the current privilege level
- if need to change privilege level
    - change stack with the one associated with new privilege
    - save old stack information on the new stack
- save EFLAGS, CS, EIP on stack
- save error code on stack in case of an abort
- execute the kernel interrupt handler

Returning from an interrupt handler

Most architectures offer special instructions to clean up the stack and resume the execution after the interrupt handler has been executed. On x86 IRET is used to return from an interrupt handler. IRET is similar to RET except that IRET increments ESP by extra four bytes (because of the flags on stack) and moves the saved flags into EFLAGS register.

To resume the execution after an interrupt the following sequence is used (x86):

- pop the error code (in case of an abort)
- call IRET
    - pops values from the stack and restore the following register: CS, EIP, EFLAGS
    - if privilege level changed returns to the old stack and old privilege level

Inspecting the x86 interrupt handling

Quiz: x86 interrupt handling

The following gdb commands are used to determine the handler for the int80 based system call exception. Select and arrange the commands or output of the commands in the correct order.

```
(void *) 0xc15de780 <entry_SYSENTER_32>

set $idtr_addr=($idtr_entry>>48<<16)|($idtr_entry&0xffff)

print (void*)$idtr_addr

set $idtr = 0xff800000

(void *) 0xc15de874 <entry_INT80_32>

set $idtr = 0xff801000

set $idtr_entry = *(uint64_t*)($idtr + 8 * 128)

monitor info registers
```

Interrupt handling in Linux

In Linux the interrupt handling is done in three phases: critical, immediate and deferred.

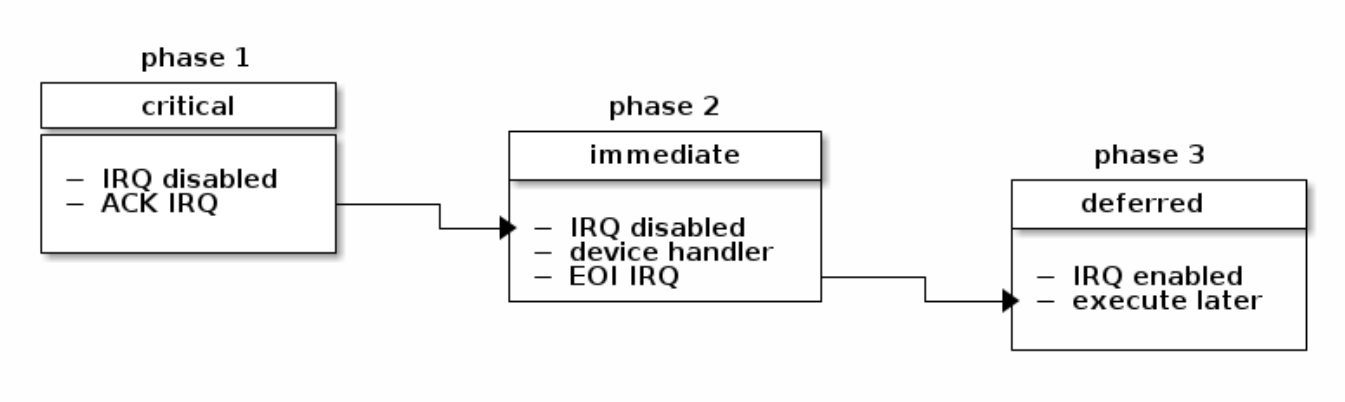
In the first phase the kernel will run the generic interrupt handler that determines the interrupt number, the interrupt handler for this particular interrupt and the interrupt controller. At this point any timing critical actions will also be performed (e.g. acknowledge the interrupt at the interrupt controller level).

Local processor interrupts are disabled for the duration of this phase and continue to be disabled in the next phase.

In the second phase, all of the device driver's handlers associated with this interrupt will be executed. At the end of this phase, the interrupt controller's "end of interrupt" method is called to allow the interrupt controller to reassert this interrupt. The local processor interrupts are enabled at this point.

> **Note**
>
> It is possible that one interrupt is associated with multiple devices and in this case it is said that the interrupt is shared. Usually, when using shared interrupts it is the responsibility of the device driver to determine if the interrupt is target to its device or not.

Finally, in the last phase of interrupt handling interrupt context deferrable actions will be run. These are also sometimes known as "bottom half" of the interrupt (the upper half being the part of the interrupt handling that runs with interrupts disabled). At this point, interrupts are enabled on the local processor.



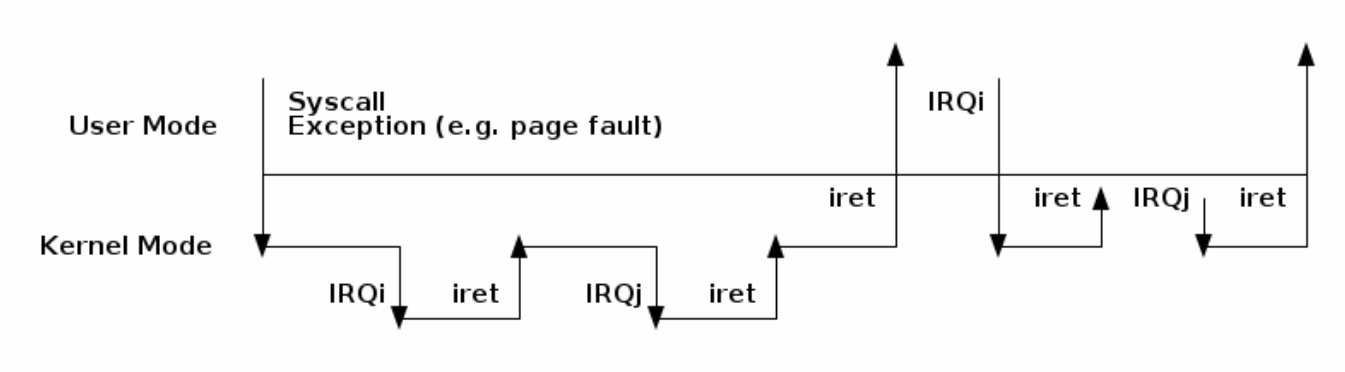
Nested interrupts and exceptions

Linux used to support nested interrupts but this was removed some time ago in order to avoid increasingly complex solutions to stack overflows issues - allow just one level of nesting, allow multiple levels of nesting up to a certain kernel stack depth, etc.

However, it is still possible to have nesting between exceptions and interrupts but the rules are fairly restrictive:

- an exception (e.g. page fault, system call) can not preempt an interrupt; if that occurs it is considered a bug
- an interrupt can preempt an exception
- an interrupt can not preempt another interrupt (it used to be possible)

The diagram below shows the possible nesting scenarios:



Interrupt context

While an interrupt is handled (from the time the CPU jumps to the interrupt handler until the interrupt handler returns - e.g. IRET is issued) it is said that code runs in "interrupt context".

Code that runs in interrupt context has the following properties:

- it runs as a result of an IRQ (not of an exception)
- there is no well defined process context associated
- not allowed to trigger a context switch (no sleep, schedule, or user memory access)

Deferrable actions

Deferrable actions are used to run callback functions at a later time. If deferrable actions scheduled from an interrupt handler, the associated callback function will run after the interrupt handler has completed.

There are two large categories of deferrable actions: those that run in interrupt context and those that run in process context.

The purpose of interrupt context deferrable actions is to avoid doing too much work in the interrupt handler function. Running for too long with interrupts disabled can have undesired effects such as increased latency or poor system performance due to missing other interrupts (e.g. dropping network packets because the CPU did not react in time to dequeue packets from the network interface and the network card buffer is full).

Deferrable actions have APIs to: **initialize** an instance, **activate** or **schedule** the action and **mask/disable** and **unmask/enable** the execution of the callback function. The latter is used for synchronization purposes between the callback function and other contexts.

Typically the device driver will initialize the deferrable action structure during the device instance initialization and will activate / schedule the deferrable action from the interrupt handler.

Soft IRQs

Soft IRQs is the term used for the low-level mechanism that implements deferring work from interrupt handlers but that still runs in interrupt context.

Soft IRQ APIs:

- initialize: `open_softirq()`
- activation: `raise_softirq()`
- masking: `local_bh_disable()`, `local_bh_enable()`

Once activated, the callback function `do_softirq()` runs either:

- after an interrupt handler or
- from the ksoftirqd kernel thread

Since softirqs can reschedule themselves or other interrupts can occur that reschedules them, they can potentially lead to (temporary) process starvation if checks are not put into place. Currently, the Linux kernel does not allow running soft irqs for more than `MAX_SOFTIRQ_TIME` or rescheduling for more than `MAX_SOFTIRQ_RESTART` consecutive times.

Once these limits are reached a special kernel thread, **ksoftirqd** is woken up and all of the rest of pending soft irqs will be run from the context of this kernel thread.

Soft irqs usage is restricted, they are use by a handful of subsystems that have low latency requirements and high frequency:

```
/* PLEASE, avoid to allocate new softirqs, if you need not _really_ high
   frequency threaded job scheduling. For almost all the purposes
   tasklets are more than enough. F.e. all serial device BHs et
   al. should be converted to tasklets, not to softirqs.
*/

enum
{
  HI_SOFTIRQ=0,
  TIMER_SOFTIRQ,
  NET_TX_SOFTIRQ,
  NET_RX_SOFTIRQ,
  BLOCK_SOFTIRQ,
  IRQ_POLL_SOFTIRQ,
  TASKLET_SOFTIRQ,
  SCHED_SOFTIRQ,
  HRTIMER_SOFTIRQ,
  RCU_SOFTIRQ,    /* Preferable RCU should always be the last softirq */

  NR_SOFTIRQS
};
```

Packet flood example

The following screencast will look at what happens when we flood the system with a large number of packets. Since at least a part of the packet processing is happening in softirq we should expect the CPU to spend most of the time running softirqs but the majority of that should be in the context of the *ksoftirqd* thread.

Tasklets

Tasklets are a dynamic type (not limited to a fixed number) of deferred work running in interrupt context.

Tasklets API:

- initialization: `tasklet_init()`
- activation: `tasklet_schedule()`
- masking: `tasklet_disable()`, `tasklet_enable()`

Tasklets are implemented on top of two dedicated softirqs: `TASKLET_SOFITIRQ` and `HI_SOFTIRQ`

Tasklets are also serialized, i.e. the same tasklet can only execute on one processor.

Workqueues

> Workqueues are a type of deferred work that runs in process context.
>
> They are implemented on top of kernel threads.
>
> Workqueues API:
>
> - init: `INIT_WORK`
> - activation: `schedule_work()`

Timers

Timers are implemented on top of the `TIMER_SOFTIRQ`

Timer API:

- initialization: `setup_timer()`
- activation: `mod_timer()`

Deferrable actions summary

Here is a cheat sheet which summarizes Linux deferrable actions:

- softIRQ
  - runs in interrupt context
  - statically allocated
  - same handler may run in parallel on multiple cores
- tasklet
  - runs in interrupt context
  - can be dynamically allocated
  - same handler runs are serialized
- workqueues
  - run in process context

Quiz: Linux interrupt handling

Which of the following phases of interrupt handling runs with interrupts disabled at the CPU level?

- Critical
- Immediate
- Deferred