

!

Linux Device Model

19

!

Linux kernel interrupt

21

Linux Kernel Interrupt

26

!

Linux Kernel Synchronisation

8

# Linux Kernel Interrupts and Handlers - Top and Bottom Halves

Rome was not built in one day; and so was interrupt handling in Linux!

That's why I decided to divide this discussion in two parts:

- Part I is to help you understand interrupts, exceptions, things to know about interrupt handlers; both; **top halves and bottom halves**.
- [Part II](http://linuxburps.blogspot.sg/2013/10/linux-interrupt-handling.html) [<http://linuxburps.blogspot.sg/2013/10/linux-interrupt-handling.html>] will take you through a dream ride in world of Linux interrupt handling – **a code walk through kernel code** using arm architecture as example (x86 is a cake walk, fun is going on a road less travelled, isn't it?)

In this article, we will be only looking at Part I.

## PART I – Interrupt Handlers (Top and Bottom Halves)

Imagine yourself eating cold plain bread and suddenly a cool breeze from window brings warm smell from nearby bar-be-que. You just can't resist it! You stop doing your work and peep through the window. That's it, you are interrupted!!

Similar to us, Linux also gets distracted and can't resist interrupts but it handles them with much grace than us.

In Linux, interrupt signals are the distraction which diverts processor to a new activity outside normal flow of execution. This new activity is called interrupt handler or interrupt service routine (ISR). With the help of Interrupts, hardware signals the processor.

Interrupt handling is amongst the most sensitive tasks performed by kernel and it must satisfy following:

- Interrupts can come anytime. The kernel's goal is therefore to get the interrupt out of the way as soon as possible and defer as much processing as it can.
- Because interrupts can come anytime, the kernel might be handling one of them while another one (of a different type) occurs.
- Some critical regions exist inside the kernel code where interrupts must be disabled. Such critical regions must be limited as much as possible.

Before we dwell any further, first understand the difference between interrupts and exceptions

Exception	Synchronous and produced by the CPU control unit while executing instructions either in response to a programming error or abnormal conditions that must be handled by the kernel. Synchronous because control unit issues them only after terminating the execution of an instruction.
Interrupt	Asynchronous and generated by other hardware devices at arbitrary times with respect to the CPU clock signals.

There is a further classification of interrupts and exceptions.

Interrupts	
Maskable	All Interrupt Requests (IRQs) issued by I/O devices give rise to maskable interrupts [ <a href="https://www.blogger.com/blogger.g?blogID=2918224943903629791">https://www.blogger.com/blogger.g?blogID=2918224943903629791</a> ] . A maskable interrupt can be in two states: masked or unmasked; a masked interrupt is ignored by the control unit as long as it remains masked.
Nonmaskable	Only a few critical events (such as hardware failures) give rise to nonmaskable interrupts [ <a href="https://www.blogger.com/blogger.g?blogID=2918224943903629791">https://www.blogger.com/blogger.g?blogID=2918224943903629791</a> ] . Nonmaskable interrupts are always recognized by the CPU.
Exceptions	
Falts	Like Divide by zero, Page Fault, Segmentation Fault.
Traps	Reported immediately following the execution of the trapping instruction. Like Breakpoints
Aborts	Aborts are used to report severe errors, such as hardware failures and invalid or inconsistent values in system tables.

Does that mean we need to understand the handling of all above? Don't worry, kernel developers are very nice and

kernel infrastructure for handling the two is similar.

If you want to be a kernel developer, understand and remember this: **interrupt handlers are different from other kernel functions**. Kernel invokes them in response to interrupts and they run in a special context called **interrupt context**. This special context is also called **atomic context** because code executing in this context is unable to block. **Process context** is the mode of operation the kernel is in while it is executing on behalf of a process—for example, executing a system call or running a kernel thread.

For a device to interrupt, its device driver must register an interrupt handler. Drivers can register an interrupt handler and enable a given interrupt line for handling with the function `request_irq()`, which is declared in `<linux/interrupt.h>`:

```
int request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags, const char name,
               void *dev)
```

This function registers a handler, first making sure that the requested interrupt is a valid one, and that it is not already allocated to another device unless both devices understand shared IRQs (with help of flags)

Flags can be zero or bit mask of one or more of the flags. Let's understand two important "flags" and I leave rest for the user to explore.

**IRQF\_DISABLED:** When set, this flag instructs the kernel to disable all interrupts when executing this interrupt handler. When unset, interrupt handlers run with all interrupts except their own enabled. Since disabling all interrupts is bad (very bad), its use is reserved for performance-sensitive interrupts that execute quickly.

**IRQF\_SHARED:** This flag specifies that the interrupt line can be shared among multiple interrupt handlers. Each handler registered on a given line must specify this flag; otherwise, only one handler can exist per line.

You must be wondering why `IRQF_SHARED` flag is needed? This is because IRQ lines are a limited resource. And a simple way to increase the number of devices a system can host is to allow multiple devices to share a common IRQ. Normally, each driver registers its own handler to the kernel for that IRQ. Instead of having the kernel receive the interrupt notification, find the right device, and invoke its handler, the kernel simply invokes all the handlers of those devices that registered for the same shared IRQ. It is up to the handlers to filter spurious invocations, such as by reading a registry on their devices.

In order to register a handler as shared, following must be satisfied:

- The `IRQF_SHARED` flag must be set in the flags argument to `request_irq()`. However, usage of `IRQF_DISABLED` can be mixed.
- The dev argument must be unique to each registered handler. A pointer to any per-device structure is sufficient; a common choice is the device structure as it is both unique and potentially useful to the handler. You cannot pass NULL for a shared handler.
- The interrupt handler must be capable of distinguishing whether its device actually generated an interrupt.

If any one device does not share fairly, none can share the line!

When your driver unloads, you need to unregister your interrupt handler and potentially disable the interrupt line. To do this, call:

```
void free_irq(unsigned int irq, void *dev)
```

If the specified interrupt line is not shared, this function removes the handler and disables the line. If the interrupt line is shared, the handler identified via dev is removed, but the interrupt line is disabled only when the last handler is removed.

Changes (important ones) were done in 2.6 kernel and it makes perfect sense to include them in our discussion:

- An option was added to reduce the process stack size from two pages down to one, providing only a 4KB stack on 32-bit systems. To cope with the reduced stack size, interrupt handlers were given their own stack, one stack per processor, one page in size. This stack is referred to as the **interrupt stack**.
- A new friend of "`request_irq()`" was introduced: "**`request_threaded_irq()`**". It is important function to understand before we go further deep in world of bottom halves.

**Threaded interrupt handlers** were introduced to reduce the time spent in interrupt handler and deferring the rest of the work (i.e. processing) out into kernel threads. So the top half would consist of a "quick check handler" that just ensures the interrupt is from the device; if so, it simply acknowledges the interrupt to the hardware and tells the kernel to wake the interrupt handler thread.

A driver that wishes to request a threaded interrupt handler will use (defined in `kernel/irq/manage.c`):

```
int request_threaded_irq(unsigned int irq, irq_handler_t handler, irq_handler_t
                        thread_fn, unsigned long flags, const char *name, void dev)
```

handler - called when interrupt occurs

thread\_fn – Function called from irq handler thread. If NULL, no irq thread is created

**Handler is called in interrupt context and it's job is usually to quite the device and return; it cannot sleep. If it's return value is IRQ\_WAKE\_THREAD, the thread\_fn() will be called in process context; it can sleep.**

Often interrupt handlers have a large amount of work to perform. For example, consider the interrupt handler for a network device. On top of responding to the hardware, the interrupt handler needs to copy networking packets from the hardware into memory, process them, and push the packets down to the appropriate protocol stack or application.

You must be thinking why I am trying to confuse you by contradicting myself— that an interrupt handler execute quickly and perform a large amount of work. Because of these competing goals, the processing of interrupts is split into two parts, or halves.

- The interrupt handler is the **top half**. The top half is run immediately upon receipt of the interrupt and performs only the work that is time-critical, such as acknowledging receipt of the interrupt or resetting the hardware.
- Work that can be performed later is deferred until the **bottom half**. The bottom half runs in the future, at a more convenient time, with all interrupts enabled.

Soon we'll be jumping on bottom halves but before we do so, there are few things worth mentioning here which you should never forget:

- **Interrupt handlers in Linux need not be reentrant.** When a given interrupt handler is executing, the corresponding interrupt line is masked out on all processors, preventing another interrupt on the same line from being received. Normally all other interrupts are enabled, so other interrupts are serviced, but the current line is always disabled. Consequently, the same interrupt handler is never invoked concurrently to service a nested interrupt. This greatly simplifies writing your interrupt handler. (Feeling any better?)
- **Interrupt context is time-critical because the interrupt handler interrupts other code. Code should be quick and simple.** Busy looping is possible, but discouraged. Always keep in mind that your interrupt handler has interrupted other code (possibly even another interrupt handler on a different line).
- Although the kernel may accept a new interrupt signal while handling a previous one, some critical regions exist inside the kernel code where interrupts must be disabled. Often, interrupts must be blocked while holding a spinlock to avoid deadlocking the system. Such critical regions must be limited as much as possible because, according to the previous requirement, the kernel, and particularly the interrupt handlers, should run most of the time with the interrupts enabled.
- Interrupt handlers do not run in process context; therefore, they cannot block.
  - A handler can't transfer data to or from user space, because it doesn't execute in the context of a process.
  - Handlers also cannot do anything that would sleep, such as calling `wait_event`, allocating memory with anything other than `GFP_ATOMIC`, or locking a semaphore.
  - Handlers cannot call `schedule`.

By now you are aware of interrupt handlers and the limitations imposed upon them.

Let's go further in our journey and understand "bottom halves".

**"The job of bottom halves is to perform any interrupt-related work not performed by the interrupt handler."**

Bottom half is a generic operating system term referring to the deferred portion of interrupt processing, so named because it represents the second, or bottom, half of the interrupt processing solution.

There is no clear guideline on how to divide the work between the top and bottom half, following useful tips can help:

- If the work is time sensitive, perform it in the interrupt handler.
- If the work is related to the hardware, perform it in the interrupt handler.
- If the work needs to ensure that another interrupt (particularly the same interrupt) does not interrupt it, perform it in the interrupt handler.
- For everything else, consider performing the work in the bottom half.

The point of a bottom half is not to do work at some specific point in the future, but **simply to defer work until any point in the future when the system is less busy and interrupts are again enabled**. Often, bottom halves run immediately after the interrupt returns. The key is that they run with all interrupts enabled.

We will not discuss about original Bottom Halves "BH" and Task queues as they were removed in 2.5, lets

understand what's available in present world.

### Softirqs:

Although they are rarely used directly; much more common form of bottom half is tasklet. But since tasklets are built on softirqs, we will discuss them first. Softirq code is in kernel/softirq.c file

**Softirqs are statically allocated at compile time**, hence, you cannot dynamically register and destroy softirqs. Softirqs are represented by *softirq\_action* structure defined in <linux/interrupt.h>

```
struct softirq_action {
    void    (*action)(struct softirq_action *);
};
```

The prototype of a softirq handler, action, looks like

```
void softirq_handler(struct softirq_action *)
```

When the kernel runs a softirq handler, it executes this action function with a pointer to the corresponding softirq\_action structure as its lone argument.

Few things to remember about softirqs:

- **A softirq never preempts another softirq. The only event that can preempt a softirq is an interrupt handler.**
- **The softirq handlers run with interrupts enabled and cannot sleep.**
- **While a handler runs, softirqs on the current processor are disabled Another softirq—even the same one—can run on another processor.**

A registered softirq must be marked before it will execute. This is called raising the softirq. In order to do so, call **raise\_softirq()** or **raise\_softirq\_irqoff()** (if interrupts are already off).

Usually, an interrupt handler marks its softirq for execution before returning. Then, at a suitable time, the softirq runs. Pending softirqs are checked for and executed in the following places:

- In the return from hardware interrupt code path.
- In the ksoftirqd kernel thread.
- In any code that explicitly checks for and executes pending softirqs, such as the networking subsystem

The softirq handler is registered at run-time via **open\_softirq()**, which takes two parameters: the softirq's index and its handler function. The kernel uses this index, which starts at zero, as a relative priority. Softirqs with the lowest numerical priority execute before those with a higher numerical priority.

Tasklet	Priority	Softirq Description
HI_SOFTIRQ	0	High-priority tasklets
TIMER_SOFTIRQ	1	Timers
NET_TX_SOFTIRQ	2	Send network packets
NET_RX_SOFTIRQ	3	Receive network packets
BLOCK_SOFTIRQ	4	Block devices
BLOCK_IOPOLL_SOFTIRQ	5	Block devices with I/O polling blocked on other CPUs
TASKLET_SOFTIRQ	6	Normal Priority tasklets
SCHED_SOFTIRQ	7	Scheduler
HRTIMER_SOFTIRQ	8	High-resolution timers
RCU_SOFTIRQ	9	RCU locking

Creating a new softirq includes adding a new entry to this enum. When adding a new softirq, you might not want to simply add your entry to the end of the list, as you would elsewhere. Instead, you need to insert the new entry depending on the priority you want to give it.

Softirqs are reserved for the most timing-critical and important bottom-half processing on the system. Normally you don't need to add a new softirq, if you do, think why using a tasklet is insufficient. Nonetheless, for timing-critical applications that can do their own locking in an efficient way, softirqs might be the correct solution.

### Tasklets:

Tasklets are implemented on top of softirqs, they are softirqs. They are represented by two softirqs: HI\_SOFTIRQ

and TASKLET\_SOFTIRQ. The only difference in these types is that the HI\_SOFTIRQ-based tasklets run prior to the TASKLET\_SOFTIRQ based tasklets.

Although they are implemented on top of softirqs, but following differentiates them:

- **Tasklets can be created statically and dynamically.**
- **Two different tasklets can run concurrently on different processors, but two of the same type of tasklet cannot run simultaneously.**

Tasklets are represented by the *tasklet\_struct* structure. Each structure represents a unique tasklet. The structure is declared in <linux/interrupt.h>:

```
struct tasklet_struct {
    struct tasklet_struct *next;    /* next tasklet in the list */
    unsigned long state;           /* state of the tasklet */
    atomic_t count;                /* reference counter */
    void (*func)(unsigned long);    /* tasklet handler function */
    unsigned long data;             /* argument to the tasklet function */
};
```

The func member is the tasklet handler (the equivalent of action to a softirq) and receives data as its sole argument.

Tasklets are scheduled (similar to raising the softirq) via the **tasklet\_schedule()** and **tasklet\_hi\_schedule()** functions.

To statically create the tasklet (and thus have a direct reference to it), use one of two macros in <linux/interrupt.h>:

```
DECLARE_TASKLET(name, func, data)
DECLARE_TASKLET_DISABLED(name, func, data);
```

To initialize a tasklet given an indirect reference (a pointer) to a dynamically created struct tasklet\_struct, t, call:

```
tasklet_init(t, tasklet_handler, dev);
```

### Work Queues:

They are quite different from softirqs, tasklets as:

- **work queues defer work into a kernel thread. They always runs in process context.** Thus, code deferred to a work queue has all the usual benefits of process context.
- **work queues are schedulable and can therefore sleep.**

Work queues let your driver create a kernel thread, called as **worked thread**, to handle deferred work.

Many drivers in the kernel defer their bottom-half work to the default thread called **events/n** (where n is the processor number). Unless a driver or subsystem has a strong requirement for creating its own thread, the default thread is preferred.

Work threads are represented by *workqueue\_struct* structure defined in kernel/workqueue.c

To create the structure statically at runtime, use DECLARE\_WORK:

```
DECLARE_WORK(name, void (*func)(void *), void *data);
```

This statically creates a work\_struct structure named name with handler function func and argument data. Alternatively, you can create work at runtime via a pointer:

```
INIT_WORK(struct work_struct *work, void (*func)(void *), void *data);
```

This dynamically initializes the work queue pointed to by work with handler function func and argument data.

To queue a given work's handler function with the default events worker threads, simply call

```
schedule_work(&work);
schedule_delayed_work(&work, delay);
```

With first call, the work is scheduled immediately and is run as soon as the events worker thread on the current processor wakes up.

While with second call, the work\_struct represented by &work will not execute for at least delay timer ticks into the future.

In order to flush a work queue i.e. ensure given batch of work is completed, use:

```
void flush_scheduled_work(void);
```

This function waits until all entries in the queue are executed before returning. While waiting for any pending work to execute, the function sleeps. Therefore, you can call it only from process context. Note that this function does not cancel any delayed work.

You can create a new work queue and corresponding worker threads if default queue is insufficient via a simple functions:

```
struct workqueue_struct *create_workqueue(const char *name);
```

Because this creates one worker thread per processor, you should create unique work queues only if your code needs the performance of a unique set of threads.

In order to understand the need for synchronization between thread/atomic and atomic/atomic context along with different primitives which allow you to do so, refer to [linux kernel synchronization primitives](http://linuxburps.blogspot.in/2013/09/linux-kernel-synchronization-primitives.html) [http://linuxburps.blogspot.in/2013/09/linux-kernel-synchronization-primitives.html] .

Now you are aware of the weapons in the arsenal. But before you go out and use them in war, understand them clearly or else, they will back fire.


Bottom Half	Context	Inherit Serialization
Softirq	Interrupt	None
Tasklet	Interrupt	Against the same tasklet
Work Queue	Process	None (scheduled as process context)

Once you are ready with the basic concepts and understanding discussed here, in the next part we will look inside the kernel interrupt handling code.

Posted 15th September 2013 by [Ankur Tyagi](#)


26 View comments


26 comments




Add a comment as aleem shariff

Top comments

**Ankur Tyagi** 2 years ago - [Linux \(Training/Tutorials/HOWTO\)](#)  
About understanding Linux kernel interrupts and handlers - top and bottom halves  
+5 1

**Mikeit P.** via Google+ 2 years ago - Shared publicly  
Linux Kernel Interrupts and Handlers Part I  
1 · Reply

**Ankur Tyagi** 2 years ago - [nerdha \(kernel\)](#)  
About understanding Linux kernel interrupts and handlers - top and bottom halves

+1 1



**Ankur Tyagi** 2 years ago - [Linux II \(Others OS and wine-likes\)](#)  
About understanding Linux kernel interrupts and handlers - top and bottom halves

1



**Ankur Tyagi** 2 years ago - [I Love Linux \(Tutorials\)](#)  
About understanding Linux kernel interrupts and handlers - top and bottom halves

1



**Ankur Tyagi** 2 years ago - [Linux \(Discussion\)](#)  
About understanding Linux kernel interrupts and handlers - top and bottom halves

1



**Ankur Tyagi** 2 years ago - [Linux Tutorials \(Tutorial\)](#)  
About understanding Linux kernel interrupts and handlers - top and bottom halves

+1 1



**Bjørn J** 2 years ago  
Good stuff, thanks for sharing.



**Ankur Tyagi** 2 years ago - [Linux Software \(Discussion\(\\*\)\)](#)  
About understanding Linux kernel interrupts and handlers - top and bottom halves

1



**Ankur Tyagi** via Google+ 2 years ago - Shared publicly  
About understanding Linux kernel interrupts and handlers - top and bottom halves

+1 1 · Reply



**Ankur Tyagi** 2 years ago - [Linux \(Kernel\)](#)  
About understanding Linux kernel interrupts and handlers - top and bottom halves

1



**Ankur Tyagi** 2 years ago - [Linux Expert \(Discussions\)](#)  
About understanding Linux kernel interrupts and handlers - top and bottom halves

+1 1



**Ankur Tyagi** 2 years ago - [Linux bloggers \(Discussion\)](#)  
About understanding Linux kernel interrupts and handlers - top and bottom halves

1



**Ankur Tyagi** 2 years ago - [Linux India \(Guides/Instructions\)](#)  
About understanding Linux kernel interrupts and handlers - top and bottom halves

1



**Ankur Tyagi** 2 years ago - [linux Tech blogs \(tutorial blogs\)](#)  
About understanding Linux kernel interrupts and handlers - top and bottom halves

1

**Ankur Tyagi** 2 years ago - [Linux \(Linux Tips\)](#)

About understanding Linux kernel interrupts and handlers - top and bottom halves

1

**Georgiy Shapovalov** via Google+ 3 months ago - Shared publicly

Great summary...

1 · Reply

**Prasad Lakshman** 2 years ago - Shared publicly

Hi your articles are very useful for better understanding ,some concepts are new to me and some are reinforcing my knowledge thanks a lot for such articles/posts . I was searching for some answers I found your article provided the answers to my questions. I followed all four posts by you on Linux internals I found all of them very useful.

1 · Reply

**Ankur Tyagi** 9 months ago

Thanks Prasad

**Senthil Veppur** 9 months ago - Shared publicly

Good content, but the images are not visible at all so the understanding is some what incomplete as you have referenced some explanation in figures.

1 · Reply

**Ankur Tyagi** 9 months ago

sorry for the lost images but if I get time, I'll try to put them (as I made everything on my own, it was time consuming)

**Senthil Veppur** via Google+ 9 months ago - Shared publicly

Very good explanation...the best i have read so far

+1 1 · Reply

**Dipak Patel** 1 month ago

How are u? Congratulation new life.

[Show more](#)