KERNEL MASTERS

Linux Interrupt Handling

Authored and Compiled By: **Boddu Kishore Kumar**

Email: kishore@kernelmasters.org

Reach us online: www.kernelmasters.org

Contact: 9949062828

Important Notice

This courseware is both the product of the author and of freely available open source materials. Wherever external material has been shown, it's source and ownership have been clearly attributed. We acknowledge all copyrights and trademarks of the respective owners.

The contents of this courseware cannot be copied or reproduced in any form whatsoever without the explicit written consent of the author.

Only the programs - source code and binaries (where applicable) - that form part of this courseware, and that are present on the participant CD, are released under the GNU GPL v2 license and can therefore be used subject to terms of the afore-mentioned license. If you do use any of them, in any manner, you will also be required to clearly attribute their original source (author of this courseware and/or other copyright/trademark holders).

The duration, contents, content matter, programs, etc. contained in this courseware and companion participant CD are subject to change at any point in time without prior notice to individual participants.

Care has been taken in the preparation of this material, but there is no warranty, expressed or implied of any kind and we can assume no responsibility for any errors or omisions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Linux Interrupt Handing

Introduction:

Because of the indeterminate nature of I/O, and speed mismatches between I/O devices and the processor devices request the processor's attention by asserting certain hardware signals asynchronously .these hardware signals are called interrupts.

Each interrupting device is assigned an associated identifier called an Interrupt Request (IRQ) number .when the processor detects that an Interrupt Service Routine (ISR) registered for the corresponding IRQ .interrupt context.

1. Process Context vs Interrupt:

Process Context	Interrupt Context
Kernel code that services system calls issued by user applications runs on of the corresponding application process and it is said to execute in process context.	Interrupt handlers, run asynchronously in Interrupt context.
Kernel code running in process context is Pre-emptible.	An Interrupt Context is not Pre-emptible.

Processes Context are not tied to any interrupt context and vice versa.

Interrupt Context cannot do the following:

1. **Go to sleep or relinquish do the processor**
2. **Acquire a mutex**
3. **Perform time-consuming tasks**
4. **Access user space virtual memory**

2. Installing an Interrupt Handler:

If you want to actually "see" Interrupts being generated, writing to the hardware device isn't enough; a software handler must be configured in the system. If the Linux kernel hasn't been told to expect you're your interrupt, it simply acknowledges and ignores it.
Interrupt lines are a precious and often limited resource, particularity when there are only 15 or 16 of them. The kernel keeps a registry of interrupt lines, similar to the registry of I/O ports. A module is expected to request an interrupt channel (or IRQ, for interrupts request) before using it and to release it when finished .in many situation, modules are also expected to be able to share interrupt lines with other drivers, as we will see.

The following functions, declared in <linux/ interrupt.h>, implement the interrupt registration interface:

```
Int request_ irq (unsigned int irq,
                    Irq_handler _t(*handler)(int,void*),
                    Unsigned long irqflags ,
                    Const char *dev_name ,
                    Void*dev_id);

Void free_irq(unsigned int, void *dev_id);
```

unsigned int irq

The interrupt number being requested.

Irq_handler_t(*handler)(int, void*)

The pointer to the handling function being installed. We discuss the arguments to this function and its return value later in this chapter.

Unsigned long flags

As you might expect, a bit maskof options (described later) related to interrupt management.

const char *dev_name

The string passed to request_irq is used in/proc/interrupts to show the ower of the interrupt (see the next section).

void *dev_id

Pointer used for shared interrupt lines. It is a unique identifier that is used when the interrupt line is freed and that may also be used by the driver to point to its own private data area (to identify which device is interrupting). dev_id must be globally unique.

2.1 Interrupt Handler Flags:

The **third parameter, flags,** can be either zero or a bit of one or more of the flags defined in <linux/interrupt.h>. Among these flags the most important are:

IRQF_DISABLED_ when set, this flags instructs the kernel to disable all interrupts when executing this interrupt handler. When unset, interrupt handlers run with all interrupts except their own enabled. Most interrupt handlers do net this flag, as disabling all interrupts is bad from. Its use is reserved for performance-sensitive interrupts that execute quickly.

IRQF_TIMER_ This flag specifies that this handler processes interrupts for the system timer.

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. More information on shared handlers is provided in a following section.

The **fourth parameter**, name, is an ASCII text representation of the device associated with the interrupt. For example, this value for the keyboard interrupt on a PC is keyboard. These text names are used by/proc/irq and /interrupts for communication with the user, with is discussed shortly.

The fifth parameter, dev, is used for shared interrupt lines. When an interrupt handler is freed (discussed later), dev provides a unique cookie to enable the removal of only desired interrupt handler from the interrupt line. Without this parameter, it would be impossible for the kernel to know which handler to remove on a given interrupt line. You can pass NULL here if the line is shared, but you must pass a unique cookie if your interrupt line is shared. (And unless your device is old and crusty and lives on the ISA bus, there is a good chance it must support sharing.)

3. Interrupt handler constraints

So far we've learned to register an interrupt handler but not to write one .actually, there's nothing unusual about a handler—its ordinary C code .the only peculiarity is that a handler runs at interrupt time and, therefore, suffers restriction on what it can do.

1. A handler can't transfer data to or from user space, because it doesn't execute in the context of a process, and the data transfer could generate a page fault. <<so? So, handling a page fault requires putting the current context to sleep, which means calling schedule () - which we can't allow in interrupt context). >>
2. Handler also cannot do anything that would sleep, such as calling wait _ event, allocating memory with anything other than GFP_ATOMIC, or locking a semaphore.
3. For protecting critical section inside interrupt handler you can't use mutexes because they may go to sleep. Use spinlocks instead, and use them only if you must.
4. Interrupt handlers are supposed supposed to get out of the way quickly but are expected to get the job done. Interrupt handlers usually split their work into two. The slim **top half** of the handler flags an acknowledgment claiming that it has serviced the interrupt but, in reality, offloads all the hard work to a fat **bottom half**. Execution of the bottom half is deferred to a later point in time when all interrupts are enabled. You will learn to develop bottom halves while discussing softirqs and tasklets later.
5. You need not design interrupt handlers to be reentrant. When an interrupt handler is running, the corresponding IRQ is disabled until the handler returns. So, unlike process context code, different instances of the same handler will not run simultaneously on multiple processors.
6. Interrupt handlers can be interrupted by handles associated with IRQs that have higher priority .you can prevent this nested interruption by specifically requesting the kernel to treat your interrupt handler as a fast handler .fast handlers run with all interrupts disabled on the local processor .before disabling interrupts or labeling your interrupt handler as fast, be aware that interrupt-off times are bad for system performance.

   .more the interrupt-off times more is the interrupt latency or the delay before a generated interrupt is serviced. Interrupt latency is inversely proportional to the real time responsiveness of the system.
7. Finally, handlers cannot call schedule.

Effective, it is illegal for interrupt context code to call schedule () either directly. A typical task for an interrupt, handler is awakening processor sleeping on the device if the interrupt signal the event they're waiting for, such as the arrival of new data. to stick with the frame grabber example , a process could acquire a sequence of image by continuously reading the device ; the read call block before reading each frame , while the interrupt handler awakens the process as soon as each new frame arrives .this assumes that the grabber interrupt the processor to signal successful arrival of each new frame .

A function can check the value retuned by in _ interrupt () to find out whether it's executing in interrupt context.

Unlike asynchronous interrupt generated by external hardware, there are classes of interrupt that arrive synchronously .synchronous interrupt are so called because they don't occur unexpectedly the processor itself generate them by executing an instruction .both external and synchronous interrupt are handled by the kernel using identical mechanisms . Example of synchronous interrupt includes the following:

1. Exceptions, which are used to report grave runtime errors.
2. Software interrupt such as the int 0×80 instruction used to implement system calls on the ×86 architecture.

4. Handler Arguments and return Value:

Static irqreturn_interrupt_handler (int irq, void *dev_id)

Two arguments are passed to an interrupt handler: irq and dev_id. Let's look at the role of each. The interrupt number (int irq) is useful as information you may print in your log messages, if any. The second argument, void *dev_id, is a sort of client data; a void * argument is passed to request_irq, and this same pointer is then passed back as an argument to the handler when the interrupt happens. You usually pass a pointer to your device data structure in dev_id, so a driver that manages several instances of the same device doesn't need any extra code in the interrupt handler to find out which device is in charge of the current interrupt event.

Return Value:

Interrupt handlers should return a value indicating whether there was actually an interrupt to handle. If the handler found that its device did, indeed attention**, it should return IRQ_HANDLED;** otherwise the return value should be **IRQ_NONE.**

Where handled is nonzero if you were able to handle the interrupt. The return value is used by the kernel to detect and suppress spurious interrupts. If your device gives you no way to tell whether it really interrupted, you should return **IRQ_HANDLED.**

5. Interrupt Control methods:

5.1. Disabling and enabling interrupts:

To disable interrupts locally for the current processor (and only the current processor) and then later reenable them, do the following:

Local_irq_disable();
/*interrupts are disabled ..*/
Local_irq_enable();

The local_irq_disable() routine is dangerous if interrupts were already disabled prior to its invocation. The corresponding call to local_irq_enable() unconditionally enables interrupts, despite the fact that they were off to begin with.

Instead, a mechanism is needed to (save and subsequently) restore interrupts to a previous state. This is a common concern because a given code path in the kernel can be both with and without interrupts enabled, depending on the call chain.

Unsigned long flags;
Local _irq_save(flags);
/* interrupts are now disabled */
/* …. */
Local_irq_restore (flags); /* interrupts are restored to their
Previous state */

All the previous functions can be called from both interrupt and process context.

5.2 Status of the interrupt System:

It is often useful to know the state of the interrupt system (for example, whether interrupts are enabled or disabled) or whether you are currently executing in interrupt context.

The macro irqs_disabled(), defined in <asm/system.h>, returns nonzero if the interrupt system on the local processor is disabled. Otherwise, it returns zero.

Two macros, defined in <linux/hardirq.h>, provide an interface to check the kernel's current context. They are

In_interrupt()

In_irq()

The most useful is the first: It returns nonzero if the kernel is performing any type of interrupt handling. This includes either executing an interrupt handler or a bottom half handler. The macro in_irq() returns nonzero only if the kernel if is specifically executing an interrupt handler.

Local_irq_disable()	Disables local interrupt delivery
Local_irq_enable()	Enable local interrupt delivery
Local_irq_save()	Saves the current state of local interrupt delivery and then disables it.
Local_irq_restore()	Restores local interrupt delivery to the given state.
Disable_irq()	Disables the given interrupt line.
Disable_irq_nosync()	Disables the given interrupt line.
Enable_irq()	Enables the given interrupt line.
Irqs_disabled()	Returns nonzero if local interrupt delivery is disabled; otherwise returns zero.
In_interrupt()	Returns nonzero if in interrupt context and zero if in process context.
In_irq()	Returns nonzero if currently executing an interrupt handler and zero otherwise.

6. Top-Half and Bottom-Half Processing:

Technically speaking the top half is the interrupt handler. Atypical top half:

1. Checks to make sure the interrupt was generated by the right hardware this is necessary for interrupt sharing.
2. Clears an interrupt pending bit on the interface board.
3. Does what needs to be done immediately (usually read or write something to/from the device.) The data is usually written to read from a device specific buffer, which has been previously allocated.
4. Schedules handling the new information later (in the bottom half).

- One of the main problems with interrupt handling is how to perform longish tasks within a handler. Often a substantial amount of work must be done in response to a device interrupt, but interrupt handlers need to finish up quickly and not keep interrupts blocked for long. **These two needs (work and speed) conflict** with each other, leaving the driver writer in a bit of a bind.
- Linux (along with many other systems) resolves this problem by **splitting the interrupt handler into two halves.** The so-called **top half** is the routine that **actually responds to the interrupt—**the one you register with request_irq.
- The **bottom half** is a routine **that is scheduled by the top half to be executed later, at a safer time.**
- But what is a bottom half useful for?
  The big difference between the top_half handler and the bottom half is that **all interrupts are enabled during execution of the bottom half—**that's why it runs at a safer time.
- In the typical scenario, the top half saves device data to a device-specific buffer, schedules its bottom half, and exits: this is very fast. The bottom half then performs whatever other work is required, such as awakening processor, starting up another I/O operation, and so on. This setup permits the top half to service a new interrupt while the bottom half is still working.
- Every serious interrupt handler is split this way. For instance, when a network interface reports the arrival of a new packet, the handler just retrieves the data and pushes it up to the protocol layer; actual processing of the packet is performed in a bottom half.
- One thing to keep in mind with bottom-half processing is that all of the restrictions that apply to interrupt handlers also apply to bottom halves .thus, bottom halves cannot sleep, cannot access user space and cannot invoke the scheduler.

6.1. Task lets:

- The Linux kernel has two different mechanisms that may be used to implement bottom-half processing. tasklet are often the preferred mechanism for bottom-half processing ;they are very fast ,but all tasklets code must be atomic.
- Tasklets, bottom-halves and softirqs are all example of the "deferred function" mechanism of the linux kernel .bottom halves are built upon tasklets: tasklets are built on softriqs.
- Tasklets are the preferred way to implement deferrable function in I/O devices. Tasklets are built over two softirqs -HI_SOFTIRQ and TASKLETS _SOFTIRQ. Several tasklets may be associated with the same IRQ, each tasklets having its own function.
- Tasklets are special function that may be schedule to run, in interrupt context, at a system-determined safe time. However, if your driver has multiple tasklets, they must employ some sort of locking (spinlocks) to avoid conflicting with each other.
- Tasklets are also guaranteed to run on the same CPU as the function that first schedules them .an interrupt handler has completed .however ,another interrupt can certainly be delivered while the tasklets is running , so locking between the tasklets and the interrupt handler may still be required .

Tasklets must be declared and initialized with the tasklets _init() function:

Tasklet_init (struct tasklet _struct *t, Void (*func)(unsigned long), unsigned long data);

“t” is the new tasklet structure which you must allocate memory for (it will be initialized internally by this function), “func” is the function that is called to execute the tasklet (it takes one unsigned long argument and returns void), and “data” is an unsigned long value to be passed to the tasklet function.

The function tasklet_schedule (or tasklet_hi_schedule) is used to activate a tasklet for running.
Void tassklet_schedule (struct tasklet_struvt *t);

6.2 Softirqs:

Softirqs are deferred processing mechanisms that usually run in an interrupt context. After handling a hardware interrupt, the kernel code path checks for pending softirqs.

do_IRQ()→ ….→ Irq_exit () →invoke_softirq()→do_softirq()

Linux uses several kinds of software IRQs:

Softirq Type	**Priority**	**Comment**
HI_SOFTIRQ	0	Handles high priority tasklets
TIMER_SOFTIRQ	1	Timers
NET_TX_SOFTIRQ	2	Transmits packets to network cards.
NET_RX_SOFTIRQ	3	Retrieves packets from network cards.
BLOCK_SOFTIRQ	4	Block devices
TASKLET_ SOFTIRQ	5	Handles regular tasklets
SCHED_SOFTIRQ	6	Scheduler
HRTIMER_ SOFTIRQ	7	HRT (High resolution timers)
RCU_ SOFTIRQ	8	RCU locking

6.3 Comparing Softirqs, Tasklets, and work Queues:

You have seen the differences between interrupt handlers and bottom halves, but there are a few similarities, too. Interrupt handlers and tasklets are both not reentrant. And neither of them can go to sleep. Also, interrupt handlers, tasklets, and softirqs cannot be preempted.

Work queues are a third way to defer work from interrupt handlers. They execute in process context and are allowed to sleep, so they can use drowsy functions such as mutexes.

	Softirqs	**Tasklets**	**Work Queues**
Execution context	Deferred work runs in interrupt context.	Deferred work runs in interrupt context.	Deferred work runs in process context.
Sleep semanption	Cannot go to sleep.	Cannot go to sleep.	May go to sleep.
Preemption	Cannot be preempted/scheduled.	Cannot be preempted/scheduled.	May be preempted/scheduled.
Ease of us	Not easy to use.	Easy to use.	Easy to use.
When to use	If deferred work go to sleep and if you have crucial scalability or speed requirement.	If deferred work will not go to sleep.	If deferred work may go to sleep.
SMP Behaviour	Same ones can be run on different CPUs Simultaneously	Can be run on Simultaneously on different CPUS, but not if they use same **tasklet_struct**.	Can be run on any CPU.

Lab Experiments:

Shared Interrupts and Bottom Halves

Write a module that shares its IRQ with your keyboard interrupt. You can generate a keyboard interrupts by press any key in keyboard.

- When ever read method invokes driver going to blocking stage.
- Make it use a top half and a bottom half.
- Check /proc/interrupts while it is loaded.
- Have the module keep track of the number of times the interrupt's halves are called.
- Implement the bottom half using tasklets & work queues