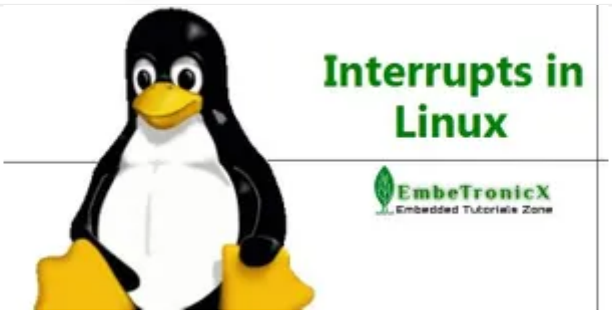
EmbeTronicX
Embedded Tutorials Zone

Sidebar

Home → Tutorials → Linux → Device Drivers → **Linux Device Driver Tutorial Part 12 – Interrupts in Linux Kernel**

Device Drivers



Linux Device Driver Tutorial Part 12 – Interrupts in Linux Kernel

This article is a continuation of the Series on Linux Device Driver, and carries on the discussion on character drivers and their implementation.This is Part 12 of the Linux device driver tutorial. In our previous tutorial, we have seen the Sysfs. Now we will see about Interrupts in the Linux kernel.

Post Contents [hide]

Interrupts in Linux kernel

This tutorial discusses interrupts and how the kernel responds to them, with special functions called interrupt handlers (ISR).

Interrupts

Here are some analogies to everyday life, suitable even for the computer-illiterate. Suppose you knew one or more guests could be arriving at the door. Polling would be like going to the door often to see if someone was there yet continuously.

That’s what the doorbell is for. The guests are coming, but you have preparations to make, or maybe something unrelated that you need to do. You only go to the door when the doorbell rings. When the doorbell rings, it’s time to check the door again. You get more done, and they get quicker responses when they ring the doorbell. This is the interrupt mechanism.

Another scenario is, Imagine that you are watching TV or doing something. Suddenly you heard someone’s voice which is like your Crush’s voice. What will happen next? That’s it, you are interrupted!! You will be very happy. Then stop your work whatever you are doing now and go outside to see him/her.

Similar to us, Linux also stops his current work and distracted because of interrupts and then it will handle them.

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

Polling vs Interrupts

POLLING	INTERRUPT
In polling the CPU keeps on checking all the hardwares of the availablilty of any request	In interrupt the CPU takes care of the hardware only when the hardware requests for some service

POLLING	INTERRUPT
The polling method is like a salesperson. The salesman goes from door to door while requesting to buy a product or service. Similarly, the controller keeps monitoring the flags or signals one by one for all devices and provides service to whichever component that needs its service.	An interrupt is like a shopkeeper. If one needs a service or product, he goes to him and apprises him of his needs. In case of interrupts, when the flags or signals are received, they notify the controller that they need to be serviced.

What will happen when the interrupt comes?

An interrupt is produced by electronic signals from hardware devices and directed into input pins on an interrupt controller (a simple chip that multiplexes multiple interrupt lines into a single line to the processor). These are the process that will be done by the kernel.

1. Upon receiving an interrupt, the interrupt controller sends a signal to the processor.
2. The processor detects this signal and interrupts its current execution to handle the interrupt.
3. The processor can then notify the operating system that an interrupt has occurred, and the operating system can handle the interrupt appropriately.

Different devices are associated with different interrupts using a unique value associated with each interrupt. This enables the operating system to differentiate between interrupts and to know which hardware device caused such an interrupt. In turn, the operating system can service each interrupt with its corresponding handler.

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

1. Hardware devices generate interrupts asynchronously (with respect to the processor clock). That means interrupts can come anytime.
2. Because interrupts can come anytime, the kernel might be handling one of them while another one (of a different type) occurs.
3. Some critical regions exist inside the kernel code where interrupts must be disabled. Such critical regions must be limited as much as

possible.

Interrupts and Exceptions

Exceptions are often discussed at the same time as interrupts. Unlike interrupts, exceptions occur synchronously with respect to the processor clock; they are often called synchronous interrupts. Exceptions are produced by the processor while executing instructions either in response to a programming error (e.g. divide by zero) or abnormal conditions that must be handled by the kernel (e.g. a page fault). Because many processor architectures handle exceptions in a similar manner to interrupts, the kernel infrastructure for handling the two is similar.

Simple definitions of the two:

Interrupts: asynchronous interrupts generated by hardware.

Exceptions: synchronous interrupts generated by the processor.

System calls (one type of exception) on the x86 architecture are implemented by the issuance of a software interrupt, which traps into the kernel and causes execution of a special system call handler. Interrupts work in a similar way, except hardware (not software) issues interrupts.

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 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. 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.

For a device's each interrupt, its device driver must register an interrupt handler.

Interrupt handler

An interrupt handler or interrupt service routine (ISR) is the function that the kernel runs in response to a specific interrupt:

1. Each device that generates interrupts has an associated interrupt handler.
2. The interrupt handler for a device is part of the device's driver (the kernel code that manages the device).

In Linux, interrupt handlers are normal C functions, which match a specific prototype and thus enables the kernel to pass the handler information in a standard way. What differentiates interrupt handlers from other kernel functions is that the kernel invokes them in response to interrupts and that they run in a special context called interrupt context. This special context is occasionally called atomic context because code executing in this context is unable to block.

Because an interrupt can occur at any time, an interrupt handler can be executed at any time. It is imperative that the handler runs quickly, to resume execution of the interrupted code as soon as possible. It is important that

1. To the hardware: the operating system services the interrupt without delay.
2. To the rest of the system: the interrupt handler executes in as short a period as possible.

At the very least, an interrupt handler's job is to acknowledge the interrupt's receipt to the hardware. However, interrupt handlers can often have a large amount of work to perform.

Process Context and Interrupt Context

The kernel accomplishes useful work using a combination of process contexts and interrupt contexts. Kernel code that services system calls issued by user applications runs on behalf of the corresponding application processes and is said to execute in process context. Interrupt handlers, on the other hand, run asynchronously in interrupt context. Processes contexts are not tied to any interrupt context and vice versa.

Kernel code running in process context is preemptible. An interrupt context, however, always runs to completion and is not preemptible. Because of this, there are restrictions on what can be done from interrupt context. Code executing from interrupt context cannot do the following:

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

Based on our idea, ISR or Interrupt Handler should be executed very quickly and it should not run for more time (it should not perform time-consuming tasks). What if, I want to do a huge amount of work upon receiving interrupts? So it is a problem right? If we do like that this will happen.

1. While ISR run, it doesn't let other interrupts to run (interrupts with higher priority will run).
2. Interrupts with the same type will be missed.

To eliminate that problem, the processing of interrupts is split into two parts, or halves:

1. Top halves
2. Bottom halves

Top halves and Bottom halves

Top half

The interrupt handler is the top half. The top half will 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.

Bottom half

The bottom half is used to process data, letting the top half to deal with new incoming interrupts. Interrupts are enabled when a bottom half runs. The interrupt can be disabled if necessary, but generally, this should be avoided as this goes against the basic purpose of having a bottom half – processing data while listening to new interrupts. The bottom half runs in the future, at a more convenient time, with all interrupts enabled.

For example, using the network card:

- When network cards receive packets from the network, the network cards immediately issue an interrupt. This optimizes network throughput and latency and avoids timeouts.
- The kernel responds by executing the network card's registered interrupt.
- The interrupt runs, acknowledges the hardware, copies the new networking packets into main memory, and readies the network card for more packets. These jobs are important, time-critical, and hardware-specific work.
  - The kernel generally needs to quickly copy the networking packet into the main memory because the network data buffer on the networking card is fixed and miniscule in size, particularly compared to the main memory. Delays in copying the packets can result in a buffer overrun, with incoming packets overwhelming the networking card's buffer and thus packets being dropped.
  - After the networking data is safely in the main memory, the interrupt's job is done, and it can return control of the system to whatever code was interrupted when the interrupt was generated.

- The rest of the processing and handling of the packets occurs later, in the bottom half.

If the interrupt handler function could process and acknowledge interrupts within few microseconds consistently, then absolutely there is no need for top half/bottom half delegation.

There are 4 bottom half mechanisms are available in Linux:

1. Workqueue
2. Threaded IRQs
3. Softirq
4. Tasklets

You can see the Bottom Half tutorials Here. In our next tutorial, you can see the programming of Interrupts with ISR.

Related

Linux Device Driver

Linux Device Driver

Linux Device Driver

Tutorial Part 14 – Workqueue in Linux Kernel Part 1
In "Device Drivers"

Tutorial Part 13 – Interrupts Example Program in Linux Kernel
In "Device Drivers"

Tutorial Part 25 – Sending Signal from Linux Device Driver to User Space
In "Device Drivers"

Subscribe ▾ Connect with | Login

Be the First to Comment!

B I U S {} [+]

This site uses Akismet to reduce spam. Learn how your comment data is processed.

0 COMMENTS

report this ad