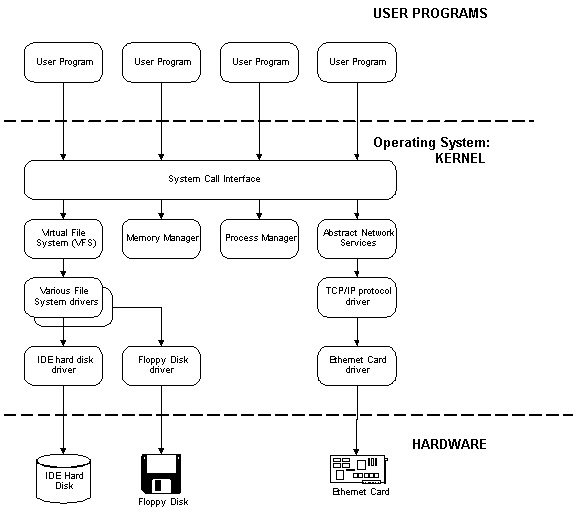
## INTRODUCTION TO DEVICE DRIVERS

### Introduction to Linux:

Linus B. Torvalds wrote the first Linux kernel in 1991 and made its source code freely available on internet. Linux kernel is coded in GNU C. Linux gained in popularity because it has always been distributed as free software unlike windows. Since the source code is readily available, users can freely change the kernel to suit their needs. However, it is important to understand how the Linux kernel has evolved and how it currently works. Linux supports nearly twenty hardware platforms including popular x86, AMD, SUN, SPARC, and Motorola.

### What is a Device Driver?

Device driver is a software which gives a device its behavior. Device driver is a piece of software which gives intelligence to hardware to make work as it should.

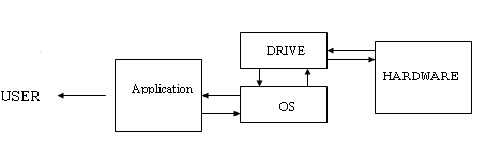


A device driver, often called a driver for short, is a computer program that enables another program, typically, an operating system (OS) (e.g., Windows, Linux, FreeBSD) to interact with a hardware device. A driver is essentially a set of instructions that provides the operating system with the information on how to control and communicate with a particular piece of hardware. In layman's terms, a driver is an important, vital piece to a program application; the main ingredients of the system

Device drivers serve several different purposes. In their purest form, they are the link between software and hardware.

For example, applications generally do not care where on a storage device a file resides; instead, they submit generic calls to the operating system to open, access, and close a file, and it is the task of the operating system and the device drivers to cooperatively locate the file on the storage device and read from or write to the correct physical locations of the device.

The fig depicts the control flow of a user request to the hardware device graphically. The advantage of this architecture is that no change needs to be made to the application if the hardware is changed.



DRIVERR

**Fig:1.1**

When writing drivers, a programmer should pay particular attention to this fundamental concept: write kernel code to access the hardware, but don’t force particular policies on the user, since different users have different needs.

Device drivers take on a special role in the Linux kernel. They make a particular piece of hardware respond to a well-defined internal programming interface; they hide completely the details of how the device works. User activities are performed by means of a set of standardized calls that are independent of the specific driver; mapping those calls to device-specific operations that act on real hardware is then the role of the device driver.

This programming interface is such that drivers can be built separately from the rest of the kernel, and "plugged in" at runtime when needed. This modularity makes Linux drivers easy to write, to the point that there are now hundreds of them available.

A device driver is a “C’ program that controls a device. The device can be a physical device (such as a disk) or a virtual device (such as a RAM disk). However, unlike a C program, you do not link a device driver into an executable program as it does not have a main( ) function - meaning that a device driver does not have a single entry point.

Each driver is different; as a driver writer, you will need to understand your specific device well. But most of the principles and basic techniques are the same for all drivers.

When writing drivers, a programmer should pay particular attention access the hardware, but don’t force particular policies on the user, since different users have different needs. The driver should deal with making the hardware available, leaving all the issues about *how* to use the hardware to the applications.

### Why to write device drivers in linux?

There are a number of reasons to be interested in the writing of Linux device drivers.The rate at which new hardware becomes available (and obsolete!) alone guarantees that driver writers will be busy for the foreseeable future.

Hardware vendors, by making a Linux driver available for their products, can add the large and growing Linux user base to their potential markets. And the open source nature of the Linux system means that if the driver writer wishes, the source to a driver can be quickly disseminated to millions of users.

As you learn to write drivers, you find out a lot about the Linux kernel in general; this may help you understand how your machine works and why things aren’t always as fast as you expect or don’t do quite what you want. We introduce new ideas gradually, starting off with very simple drivers and building on them; every new concept is accompanied by sample code that doesn’t need special hardware to be tested.

### Device Drivers can be classified into

**Statically linked driver:** whose object code is linked with the kernel. The code of such device driver is physically contained in the kernel and therefore loaded in memory when the system boots.

**Dynamically linked driver**: whose object code is NOT linked with the kernel. The code of such device driver is NOT contained in the kernel, and the device driver is loaded and unloaded as and when required

### Advantages of Dynamic Loading:

Linux device drivers can be integrated into the kernel in two different ways : either by compiling them into the kernel so that they are always available, or by compiling them into an object format that the kernel can load whenever access to a specific device is required. Kernel code that can be automatically loaded into the kernel is referred to as a loadable kernel module. When configuring the Linux kernel, each kernel configuration editor displays a description of available kernel configuration variables and enables you to specify whether each should be deactivated, compiled into the kernel, or compiled as a loadable kernel module.

Compiling device drivers into the kernel has the advantage that they are always instantly available, but each increases the size of the kernel that you are running. Compiling device drivers as loadable kernel modules implies some slight overhead when you search for and initially load the module in order to access the associated device, plus some small runtime overhead, but these are negligiblecompared to thesavings in size and associated memory requirements. Writing device drivers as loadable kernel modules also provides significant advantages during development. As you develop and debug your device driver, you can dynamically unload the previous version and load the new version each time you want to test the new version. If your device driver is compiled into the kernel, you have to recompile the kernel and reboot each time that you want to test a set of iterative changes. Similarly, developing and deploying device drivers as loadable kernel modules simplifies maintaining them in the field, since the device driver can be updated as a separate system component without requiring a kernel update.

Configuring your kernel for support for loadable modules is done in the Loadable module support section of your kernel configuration editor. The Automatic kernel module loading option determines whether the kernel will automatically try to locate and load modules as they are needed by new devices or subsystems.

The Module versioning support option (marked as experimental in the current 2.6 kernel source) adds extra versioning information to compiled modules at build-time. This information is designed to help increase module portability to kernels other than the one that they were compiled against. The Module unloading option is new to 2.6 kernel support for loadable kernel modules. You must enable this option if you want your kernel to be able to unload modules when they are no longer needed. This is especially important in resource-constrained and power-sensitive environments such as embedded systems. If you activate this option, you can also activate the Forced module unloading option, which enables you to forcibly unload modules regardless of whether the kernel believes that they are in use.

**Linux Kernel Versions:**

1. **Stable kernels** : these are production level releases suitable for widespread

deployment

**Ex: 3.10.18**

3 is Major version no.

10 is minor version no.

Changed only when there drastic changes to kernel i.e from 3.9 to 3.10

18 is first release

These are releases with bugfixes.

1. **Development kernels:** these undergo rapid change where anything goes, often

drastic changes to the kernel are made

**Ex: 3.9.18**

### What Is Kernel?

Kernel is core part of operating system which is always running.

It includes:

* Process control subsystem
* File subsystem
* Memory management subsystem
* IPC
* Scheduler
* Device Drivers

Device drivers are part of the kernel which deal with yhe hardware. All the requests from

user are ultimately handeled by device drivers.

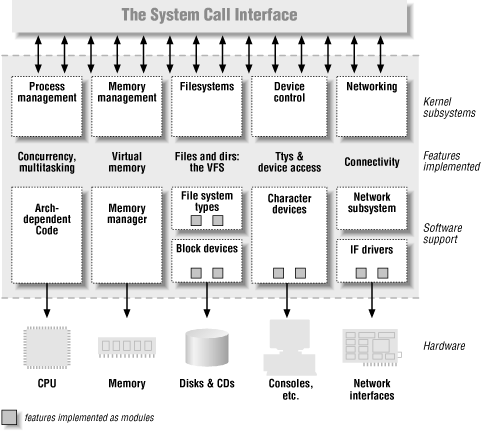
### Features of linux kernel:

1. Does not have access to C library.
2. Kernel stack is fixed
3. Kernel cannot easily use floating point
4. No memory protection in kernel space.

### Splitting the Kernel

In a UNIX system, several concurrent *processes* attend to different tasks. Each process asks for system resources, be it computing power, memory, network connectivity, or some other resource.

The *kernel* is the big chunk of executable code in charge of handling all such requests. Though the distinction between the different kernel tasks isn't always clearly marked, the kernel's role can be split, as shown in Figure 1-1, into the following parts:



**Figure 1-2. A split view of the kernel**

### Process management

The kernel is in charge of creating and destroying processes and handling their connection to the outside world (input and output). Communication among different processes (through signals, pipes, or interprocess communication primitives) is basic to the overall system functionality and is also handled by the kernel.

In addition, the scheduler, which controls how processes share the CPU, is part of process management. More generally, the kernel's process management activity implements the abstraction of several processes on top of a single CPU or a few of them.

### Memory management

The computer's memory is a major resource, and the policy used to deal with it is a critical one for system performance. The kernel builds up a virtual addressing space for any and all processes on top of the limited available resources. The different parts of the kernel interact with the memory-management subsystem through a set of function calls, ranging from the simple *malloc*/*free* pair to much more exotic functionalities.

### Filesystems

Unix is heavily based on the filesystem concept; almost everything in Unix can be treated as a file. The kernel builds a structured filesystem on top of unstructured hardware, and the resulting file abstraction is heavily used throughout the whole system. In addition, Linux supports multiple filesystem types, that is, different ways of organizing data on the physical medium. For example, diskettes may be formatted with either the Linux-standard ext2 filesystem or with the commonly used FAT filesystem.

### Device control

Almost every system operation eventually maps to a physical device. With the exception of the processor, memory, and a very few other entities, any and all device control operations are performed by code that is specific to the device being addressed. That code is called a *device driver*. The kernel must have embedded in it a device driver for every peripheral present on a system, from the hard drive to the keyboard and the tape streamer. This aspect of the kernel's functions is our primary interest in this book.

### Networking

Networking must be managed by the operating system because most network operations are not specific to a process: incoming packets are asynchronous events. The packets must be collected, identified, and dispatched before a process takes care of them. The system is in charge of delivering data packets across program and network interfaces, and it must control the execution of programs according to their network activity. Additionally, all the routing and address resolution issues are implemented within the kernel.

### Classes of Devices and Modules

The UNIX way of looking at devices distinguishes between three device types. Each module usually implements one of these types, and thus is classifiable as a *char* module, a block module, or a network module. The classes are the following:

* Character devices
* Block devices
* Network devices
* Pseudo device drivers

### Character devices

A character (char) device is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior. Such a driver usually implements at least the *open*, *close*, *read*, and *write* system calls. The text console (*/dev/console*) and the serial ports (*/dev/ttyS0* and friends) are examples of char devices, as they are well represented by the stream abstraction. Char devices are accessed by means of filesystem nodes, such as */dev/tty1* and */dev/lp0*. The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially. There exist, nonetheless, char devices that look like data areas, and you can move back and forth in them; for instance, this usually applies to frame grabbers, where the applications can access the whole acquired image using *mmap* or *lseek*.

### Block devices

Like char devices, block devices are accessed by filesystem nodes in the */dev* directory. A block device is something that can host a filesystem, such as a disk. In most Unix systems, a block device can be accessed only as multiples of a block, where a block is usually one kilobyte of data or another power of 2. Linux allows the application to read and write a block device like a char device -- it permits the transfer of any number of bytes at a time. As a result, block and char devices differ only in the way data is managed internally by the kernel, and thus in the kernel/driver software interface. Like a char device, each block device is accessed through a filesystem node and the difference between them is transparent to the user.

### Network interfaces

Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts. Usually, an interface is a hardware device, but it might also be a pure software device, like the loopback interface. A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Though both Telnet and FTP connections are stream oriented, they transmit using the same device; the device doesn't see the individual streams, but only the data packets.

Each piece of code that can be added to the kernel at runtime is called a module. The Linux kernel offers support for quite a few different types (or classes) of modules, including, but not limited to, device drivers. Each module is made up of object code (not linked into a complete executable) that can be dynamically linked to the running kernel by the insmod program and can be unlinked by the rmmod program

### Pseudo device driver

Not all device drivers control physical hardware. Such device drivers are called “Pseudo device drivers” or just drivers. Like block & character device drivers, pseudo device drivers make use of device drivers interfaces. Unlike block & character device drivers, pseudo device drivers do not operate on the bus. One example of pseudo device drivers is the pseudo terminal or pty terminal driver, which simulates a terminal device. The pty terminal driver is a character device driver typically used for remote logins.

### When is a device driver called ?

A device driver is called during:

* Autoconfiguration

The kernel calls a device driver at autoconfiguration time to determine what devices are available & to initialize them.

* I/O operations

The kernel calls a device driver to perform I/O operations on the device. These operations include opening the device to perform reads & writes & closing the device.

* Interrupt handling

The kernel calls a device driver to handle interrupts from devices capable of generating them.

* The kernel calls a device driver to handle special requests through ioctl calls
* Reinitialisation

The kernel calls the device driver to reinitialize the driver, the device, or both when the bus is reset.

Some of these requests, such as input or output, result directly or indirectly from corresponding system calls in a user program. Other requests, such as the calls at autoconfiguration time, do not result from system calls but from activities that occur at boot time.

### Security Issues:

A kernel module can do anything, A module is just as powerful as a superuser shell, any security check in the system is enforced by the kernel code, if the kernel code has security holes, then the system has holes**,** The system call *create\_module* checks if the invoking process is authorized to load a module into the kernel, thus, when running an official kernel, only the superuser can load a module into the kernel.