BACHELOR OF COMPUTER APPLICATIONS

SEMESTER 4

DCA2203

SYSTEM SOFTWARE

Unit 9

Device Driver – I

Table of Contents

1. INTRODUCTION

We discovered the Text Editor in the preceding unit. An editor's main job is editing, and a text editor is a computer program that enables users to enter, modify, save, and typically print text. The document-editing process is an interactive user-computer dialogue designed to accomplish different tasks. You also read that an interactive debugging system provides programmers with facilities that aid in the testing and debugging programs. In this unit, you will study the device driver that acts as a component to provide Input-output (I/O) services to interact with any hardware devices that are connected to the computer system.

. Every device part of the computer system needs a device driver; otherwise, the Operating System (OS) cannot operate or cannot manage the connection between the hardware devices. The role of a device driver is to provide a certain mechanism to allow accessing data to and from the attached devices. We'll discover how the Unix/Linux operating system categorizes devices into several groups. Device drivers are necessary parts that are provided by the company that makes the device, however, there are security concerns with them. In the end we will study the design issues related to the device drivers. The focus mainly with the device drivers related to the Unix/Linux OS.

1.1 Learning Objectives:

After studying this unit, the learners should be able to:

- *Explain about device drivers*
- *Determine the role of the device driver*
- *List the classes of devices*
- *Describe the security issues concerned with device drivers*
- *Design issues associated with the device drivers.*

2. DEVICE DRIVER

The Operating System (OS) is like a manager that manages the interaction between all the devices and the underlying process between them in a computer system. It is the job of the OS to facilitate communication between all the devices connected to the computer system. However, some devices, known as *peripheral devices*, that are connected externally to the computer system may not be able to communicate because the hardware components of external devices may have been designed in a different way for communicating data to and from other devices. Hence, peripherals connected to a computer need special communication links to interfere with the CPU. The purpose of the communication link is to resolve the differences between the central computer and each peripheral.

The major differences between the central computer and each peripheral are due to the following reasons:

- Peripherals are electromagnetic and electromechanical devices, so their operations differ from the CPU and the memory.
- Data transfer rate of peripherals is different fromthe transfer rate of the CPU; hence a synchronization mechanism is needed.
- Data codes and formats in peripherals differ from the word format in the CPU and memory.
- The operating modes of peripherals are different from each other, and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

To resolve these differences, computer systems include interface units between the CPU and peripherals to supervise and synchronize all types of input and output functions. Between the CPU bus and the peripheral device are interface units. To allow the OS to recognize and access the hardware devices, certain software components in the form of software interface are necessary.

A device driver is a software interface that allows the OS to provide Input-output (I/O) services to interact with an underlying device. The device driver converts the logical requests from the user into specific commands directed to the device. Generally, the term

driver is used to mean software driver, device driver, bus driver, or other drivers related to the object.

As shown in figure 9.1, in windows OS, any application must communicate to devices through the driver.

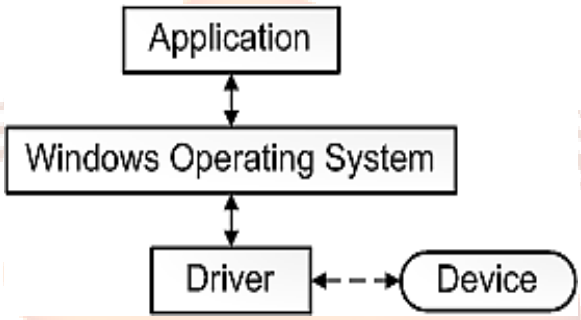


Figure 9.1: Driver between OS and devices

Although device drivers are justadd-on modules, they are an integral part of the system closely integrated with the Input/ Output Control System, which deals with I/O related system calls.

Generally, device drivers are installed for every device on the computer. Some devices drivers, such as disk drives, the processor and the chipset on the mother board are shipped with the operating system. Because they are necessary for the operating system to function.The device manufacturer generally provides device drivers if the operating system does not support the device. The drivers are written by the manufacturer of the device as it knows how the device hardware communicates to get the data. Drivers of any device are different for the different operating systems and their supporting versions.

Example: It's possible that the manufacturer of your visual card doesn't produce a driver for a specific operating system. Sometimes, not all drivers are written by manufacturers. Generally, a device is designed and made according to a published hardware standard. This means that the driver can be written by OS developing company (like Microsoft or MAC) or a group of people (for Linux) and hence the device designer may not provide a driver.

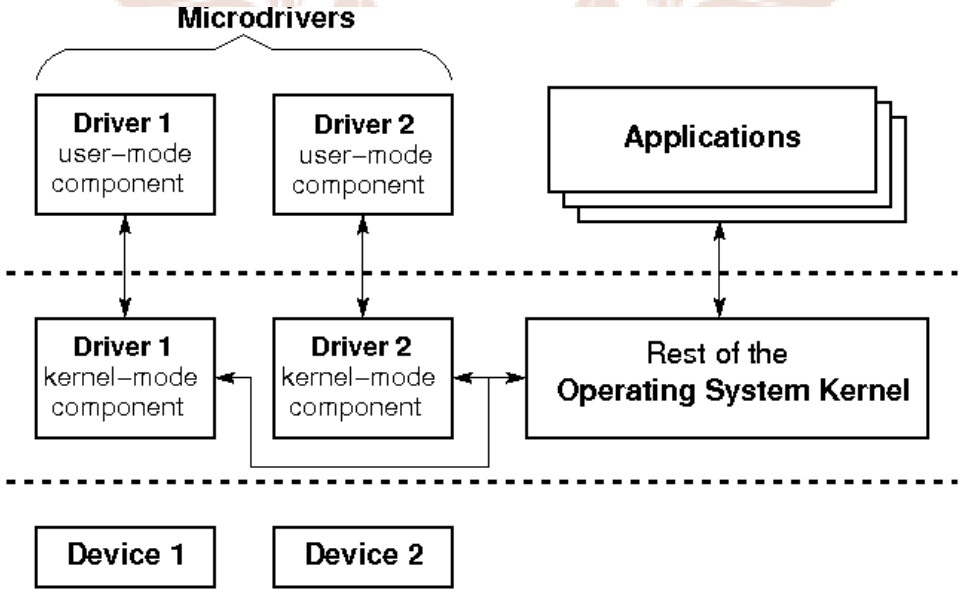
Any OS loads device drivers and calls functions in the drivers to access and carry out actions on the device. The driver functions contain the device- specific code needed to carry out actions on the device.

Example: If any application needs to read some data from a device, then the application calls a function implemented by the operating system, and the operating system calls a function implemented by the driver. After the driver gets the data from the device, it returns the data to the operating system, which returns it to the application.

Features of device drivers:

- Device responses and communications sent to the computer are managed by device drivers.
- Without system drivers, particular hardware does not function as expected or in accordance with its intended purpose.
- We can optimize the hardware thanks to it.
- It gives us the ability to speed up or slow down hardware and test its limitations.
- It helps us by informing us of the device's state.

Network Architecture for Device Driver



At the basic level, device drivers can be split into two layers, a logical layer and a physical layer.

Logical Layer:

- Logical layers process data for a class of devices.

Physical Layer:

- Specific instances of devices are communicated through physical layers.
- The physical layer will deal with the logical layer's communication needs with a specific serial port chip for them to work together.
- The device information structure and the static structure are two crucial data structures found in every device driver.
- Both the device driver installation and information sharing between entry point functions employ these structures.

Device information structure:

- The purpose of the device information structure, which is a static file supplied to the install entry point, is to send along any information needed to install a major device into the entry point, where it will be utilized to initialize the static structure.

Static structure:

- The static structure is initialized with the data stored in the information structure and is used to transmit information between the various entry points.
- The operating system uses the entry point functions,m to interact with the driver.

Micro drivers divide device driver functionality between a kernel-mode component and a user-mode component. .

A micro driver comprises a user-mode process that handles non-essential functionality and a kernel-mode component that handles crucial functionality. At function borders, device driver functionality is divided between the kernel-mode and user-mode components.

2.1 UNIX/Linux Device Drivers

Since Linux follows the UNIX model, and in UNIX, everything is a file, users communicate with device drivers through device files. Device files are supplied by the kernel for a direct User-Driver interface. Hence in UNIX/Linux OS, device drivers are in the form of files and are usually linked to the object code of the kernel (the core of the operating system). Almost every system operation eventually maps to a physical device. Nearly all device control operations are performed by a device driver specific to the device being addressed except a few of the processor, memory, and a a few other entities. Hence, in the UNIX/Linux OS, 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 drive. This means that when a new device is to be used, which was not included in the original construction of the operating system, the UNIX kernel must be re-linked with the new device driver object code. This technique has the advantages of run-time efficiency and simplicity, but the disadvantage is that adding a new device requires the regeneration of the kernel. In UNIX, entry of devices is in the /dev directory associated with a device driver that manages the communication with the related device. A list of some device names is as shown below:

Device name	**Description**
/dev/console	system console
/dev/tty01	user terminal 1
/dev/tty02	user terminal 2
/dev/lp	line printer
/dev/dsk/f03h	1.44 MB floppy drive

2.2 MS-DOS Device Drivers

In MS-DOS, device drivers are installed and loaded dynamically, i.e., they are loaded into memory when the computer is started or re-booted and accessed by the operating system as required. This technique ensures that only those drivers which are required are loaded into the main memory. The device drivers to be loaded are defined in a special file called CONFIG.SYS, which resides in the root directory. MS-DOS reads this file at start-up of the system, and its contents acted upon. A list of some device names is shown below:

Device name	**Description**
con:	keyboard/screen
com1:	serial port1
com2:	serial port2
lpt1:	printer port1
A:	first disk drive
C:	hard disk drive

2.3 Windows System Device Drivers

In the Windows system, device drivers are usually library files with .sys file name extensions and are implemented as dynamic link libraries (DLLs). This technique provides advantage that DLLs contain hence contain a shareable code; hence, one copy of the code needs to be loaded into memory. Another advantage is that a software or hardware vendor can implement a new device without modifying or affecting the Windows code. You can also configure optional drivers for devices.

In the Windows system, the idea of Plug and Play device installation is required to add new devices such as a CD drive, Flash drive, etc. The objective is to make this process automatic so that the driver software is loaded when the device is attached . After that, the installation is automatic, and the settings are chosen to suit the host computer configuration.

Self-Assessment Questions - 1

1. Peripherals connected to a computer need special communication links for "_____" them with the CPU.
2. A device "_____" is a software interface that allows the OS to provide Input-output (I/O) services to interact with an underlying device.
3. Which of the following directory is associated with a device driver that manages the communication with the related device in UNIX?
   a) /dev
   b) /etc
   c) /home
   d) /lib
4. In MS-DOS, device drivers are installed and loaded "_____".
5. In the Windows system, device drivers are usually library files with "_____" filename extensions and are implemented _____ .

3. ROLE OF DEVICE DRIVERS

Before going on to discuss the role of device drivers, let us study the difference between "user space" and "kernel space." UNIX/Linux developers use the term user space and kernel space whereas Windows developers use the term user mode and kernel mode. In UNIX/Linux, processes switch between user space and kernel space using system calls. In Windows, the processor switches between the user and kernel modes depending on what type of code runs on the processor. Applications run in user mode, and core operating system components run in kernel mode. Many drivers run in kernel mode, but some run in user mode. Explanation of these different driver forms is beyond this SLM's scope.

- **Kernel space:** The kernel and its device drivers form an interface between the user/programmer and the hardware. Kernel space includes any routines or functions that are a kernel component (such as modules and device drivers).
- **User space:** Like the UNIX shell or other GUI-based applications, end-user programs are part of the user space. These applications interact with the system's hardware. However, in Linux, they interact through the kernel-supported functions.

Let us look at the types of Roles Performed by a device:

Console: You can install the configuration manager console on other computers and restrict access and limit what administrative users can find in the console by utilizing the configuration manager's role-dependent administration. With the computer, it communicates. Examples include video games, system alerts, and voice instructions.

Communication: A communication device is a hardware device capable of transmitting the digital signal of the computer system over communication interfaces. Other communication devices include NIC (network interface card), Wi-Fi, and access points.

- A computer modem is a prime example of a communication device.
- With the use of a gadget, chat, and VoIP communications are possible.
- Speaking on the phone with someone else

Multimedia: The multimedia role of the device allows a person to deal with various media while eliminating the need to have a separate device for each.

- Examples include playing or recording audio content, music, and movies, narration, and live music recording.
- A multimedia device is a piece of hardware designed to play, store, record, or display multimedia files like music, films, and images.
- . People use multimedia audio devices to record and play audio, including music, sound effects, and voice.

The examples of an input device are as follows:

- The 101 keys on keyboards are necessary for communicating.
- Pointing devices include the mouse, trackball, and space ball.
- A scanner is an input device that functions similarly to a photocopier.
- Sound is captured via a microphone and then stored digitally as input.
- An input device known as a digitizer converts analog data into digital form.
- Banks utilize the MICR input device to process a significant number of cheques each day.

Working on device drivers in UNIX/Linux OS

In UNIX/Linux, a device driver communicates with the user, kernel, and hardware, as shown in figure 9.2.

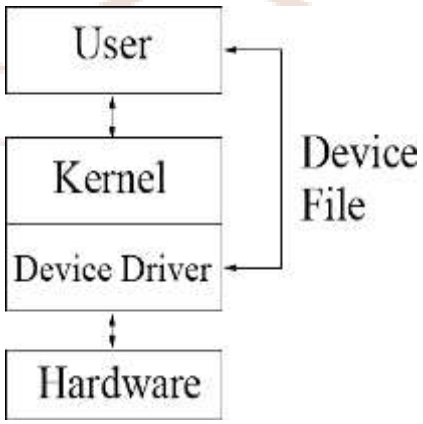


Figure 9.2: Connection of device driver with the user, Kernel, and hardware in Linux.

The kernel offers several subroutines or functions in user space, which allow the end-user application programmer to interact with the hardware. In UNIX/Linux systems, the devices are seen as files, and hence the interaction is performed through functions or subroutines to read and write files. In kernel space, UNIX/Linux offers several functions or subroutines to perform low-level interactions directly with the hardware and allow the transfer of information from the kernel to the user space. Usually, for each function that allows the use of devices or files in user space, there exists an equivalent function or subroutines in kernel space to transfer information from the kernel to the user and vice-versa.

Working on device drivers in Windows OS

Windows OS loads device drivers and calls functions in the drivers to carry out actions on the device. Each device driver exports a set of functions, and Windows calls these functions. The driver functions contain the device-specific code needed to carry out actions on the device.

Mechanism and Policy

When the one design's or program a device driver, then you are free to choose an acceptable trade-off between the programming time required to develop a driver and the flexibility of the result. The term "flexible" here means that the role of a device driver is providing a mechanism, not a policy. The role of a device driver can be discussed in terms of *mechanism* and *policy*. As a programmer, one is free to design the driver in terms of mechanism and policy. When we talk of mechanism, it is"What capabilities are to be provided,". In contrast, when we talk of policy,it is about "how those capabilities can be used." Most programming problems can be split into the mechanism and the policy. If the two issues are addressed by different parts of the program, or even by different programs, it becomes much easier to develop software programs for a particular need.

For example, UNIX management of the graphic display is split between

(i) the X server and (ii) the window and session managers. The X server knows the hardware and offers a unified interface to user programs. The window and session managers implement a particular policy without knowing anything about the hardware. People can use

the same window manager on different hardware, and users can run different configurations on the same sworkstation. Even a different desktop environment, such as KDE and GNOME, can coexist on the same system. In the case of the layered structure of TCP/IP networking as another example, the operating system offers the socket abstraction, Which has no policies in place regarding the data transfers, while the services are handled by various servers (and their associated policies).

Wherever drivers are concerned, a separation of mechanism and policy can be applied. Consider that a floppy drive is a policy free as its role is only to show the diskette as a continuous array of data blocks. Higher levels of the system provide policies, such as who may access the floppy drive, whether the drive is accessed directly or via a file system, and whether users may mount file systems on the drive. Different environments usually need to use hardware differently, so it is important to be as policy free as possible.

When a programmer is developing a device driver, pay particular attention to the fundamental concept: write kernel code to access the hardware, but do not force specific particular policies on the user, since different users have different needs. The device driver is a software layer that lies between the applications and the actual device. You can design what you can enable your driver to be, i.e., can design drivers to offer different capabilities, even for the same device and design of a driver should be balanced, considering various factors. For instance, a single device may be used concurrently by different programs, and the driver programmer can determine how to handle concurrency. Consider the trade-off between the maximum options available for users and the time taken to develop the driver, along with simplicity and error-free mechanism.

Develop a policy-free driver that can have several typical characteristics. Those characteristics can be a support for both synchronous and synchronous operations, the ability to be opened multiple times, the ability to exploit the full capabilities of the hardware, and the lack of software layers to “simplify things” or provide policy-related operations. Drivers of this sort of work better for their end users and are easier to write and maintain. Hence making a policy-free driver is the main objective of software designers.

Device drivers can also be sent along with user applications to aid with configuration and access to the target device. With programs that can be simple utility or graphical apps, you can grant end-users independence.

Let us see the Advantage and Disadvantage of device drivers:

- A device driver aids in the generalization of programming by serving as an abstraction layer between a hardware device and the applications or operating systems that use it.
- Devices based on information may not be able to function for the purpose for which they were implemented if some incorrect information has been entered into the database.

How to use the manufacturer's website to get appropriate drivers in detail:

- You must first identify the hardware's model number and manufacturer before installing the driver.
- Then, access the support page for the manufacturer and log in. Find out where to download drivers.
- Now save the driver to your PC after downloading it.
- After that, extract the driver and adhere to the computer's instructions.

Let us see the steps to Download Driver from Third Party Driver Installer:

Download Driver Reviver first, then install it on your device to verify the directions displayed on the screen.

- Then click "Start PC Scan Now" to have your computer check for any outdated or corrupted drivers.
- The drivers are immediately updated when selecting "Update all."

Let us see an example of driver installation:

- Driver will begin the program installation process from the operating system library.
- The software driver for that specific printer model launches and manages the hardware component, such as a printer.

- Your computer should immediately identify the new printer after you connect the printer wire to your device.

Let us see how the driver works:

We need a driver since Operating System is software and the device is hardware. Both use different languages and do not understand each other's language.

The device driver's main job is translating commands from the operating system to the device.

The Operating System wants to exchange data with the devices for sending and receiving purposes.

Self-Assessment Questions - 2

6. Any subroutines or functions forming part of the kernel are considered part of "_____"whereas end-user programs, like the UNIX shell or other GUI-based applications are part of _____ ".
7. How does Windows OS carry out actions on the device?
   a) Windows loads device drivers and calls functions in the drivers
   b) Windows loads devices in the memory
   c) Memory connects devices to the processor
   d) Device drivers are connected to the memory and processor
8. The role of device driver design can be seen through mechanism and policy. (True/False)

4. CLASSES OF DEVICES

UNIX/Linux OS distinguishes and classifies devices attached to the computer devices into three fundamental device types. They are

(i) Character devices (ii) Block devices (iii) Network interfaces. The modules developed as device drivers are usually implemented for one of these devices. Depending upon the device types, the modules for the device driver can be classified as (i) a char module (ii) a block module (iii) a network module. Although the modules are programmed and divided into classes, it is the freedom of the programmer to either build huge modules implementing different drivers in a single chunk of code or build into the modules as classified above. Ifscalability and extension ability are required in the functionality of different modules, then it is better to create a different module for each new functionality.

I. **Character devices:** A character (char) device is classified because it can be accessed as a stream of bytes (like a file). This driver usually implements at least the open, close, read, and write system calls. For example, the text console *(/dev/console*) and the serial ports *(/dev/ttyS0*) are well represented by the stream abstraction and hence are examples of char devices. Char devices are accessed using file system nodes, such as */dev/tty1* and
/dev/lp0. Char devices are just data channels which can be accessed sequentially.
The difference between a char device and a regular file is that you can move back and forth in the regular file, whereas most char devices are data channels which can be accessed sequentially.

II. **Block devices:** A block device (e.g., a disk) that can host a filesystem. Like char devices, block devices are accessed by file system nodes in the */dev* directory. In most Unix systems, a block device can only handle I/O operations that transfer one or more whole blocks, usually 512 bytes (or a larger power of two) in length. Instead, Linux allows the application to read and write a block device like a char device – it permits the transfer of any number of bytes simultaneously . 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 file

system node, and the difference between them is transparent to the user. Block drivers have a completely different interface to the kernel than char drivers.

III. ***Network interfaces:*** Any network transaction is made through an interface, that is, a device that can exchange data with other hosts. Usually, an interface is a hardware device, but it is also a pure software device, like the loopback interface. A network interface is responsible for sending and receiving data packets, driven by the kernel's network subsystem, without knowing how individual transactions map to the actual packets being transmitted. Many network connections (especially those using TCP) are stream-oriented, but network devices are usually designed around the transmission and receipt of packets. A network driver knows nothing about individual connections; it only handles packets.

The Unix way to provide access to interfaces is to assign a unique name to them (such as eth0), but that name doesn't have a corresponding entry in the file system. Communication between the kernel and a network device driver is completely different from that used with char and block drivers. Instead of reading and writing, the kernel calls functions related to packet transmission.

There are other ways of classifying driver modules, also. In general, some drivers work with additional layers of kernel support functions for a given type of device. For example, every USB device is driven by a USB module that works with the USB subsystem. Still, the device itself shows up in the system as a char device (a USB serial port, say), a block device (a USB memory card reader), or a network device (a USB Ethernet interface).

Other classes of device drivers are added to the kernel nowadays, including FireWire drivers and I2O drivers.

In addition to device drivers, other functionalities related to hardware and software are modularized in the kernel. One common example is file systems. A file system type determines how information is organized on a block device to represent a tree of directories and files. Such an

The entity is not a device driver because there's no explicit device associated with how the information is laid down. The file system type is instead a software driver as it maps the low-level data structures to high-level data structures. that the file system determines how long a file name can be and what information about each file is stored in a directory entry. The file system module must implement the lowest level of the system calls that access directories and files by mapping filenames and paths (as well as other information, such as access modes) to data structures stored in data blocks. Such an interface is completely independent of the actual data transfer to and from the disk (or another medium), which a block device driver accomplishes.

In Unix/Linux platform, the ability to decode file system information stays at the lowest level of the kernel hierarchy. Unix/Linux supports the concept of a file system module, whose software interface declares the different operations that can be performed on a file system inode, directory, file, and superblock.

Self-Assessment Questions - 3

9. Which drivers are used to accessing character devices as a file?
   a) Block drivers
   b) char drivers
   c) network drivers
   d) stream driver
10. A block device is a device that can host a "_____ ".
11. A network driver knows nothing about individual connections; it only handles packets. (True/False)

5. SECURITY ISSUES

Security is of prime concern in modern times when any software is been developed. Any security check in the system is enforced by kernel code. There should not be security holes in the kernel due to a bad module that gets loaded into the kernel. In the official kernel distribution, only an authorized user can load modules. The system called *init* _module checks if the invoking process is authorized to load a module into the kernel. Thus, only

the superuser, or an intruder who has succeeded in becoming privileged, can exploit the power of privileged code by running an official kernel.

Driver writers should avoid encoding security policy in their code as far as possible. Security is a policy issue that is often best handled at higher levels within the kernel, under the system administrator's control. However, there are always exceptions. As a device driver writer, one should be aware of situations in which some types of device access could adversely affect the system as a whole and should provide adequate controls.

Any software that uses system resources to get an illicit profit should be prohibited. You should be aware that even the slightest software flaws become obvious to everyone and might lead to misuse. For example, device operations that affect global resources (such as setting an interrupt line), which could damage the hardware (loading firmware, for example), or that could affect other users (such as setting a default block size on a tape drive), should be available to privileged users only.

The driver writers must be careful to avoid introducing security bugs. Security bugs, as an example, can be due to buffer overrun errors in which the programmer forgets to check how much data is written to a buffer. Data ends up written beyond the end of the buffer, thus overwriting unrelated data. Such errors can compromise the entire system and must be avoided.

It's important to remember that any input from user processes shouldn't be used until it can be verified. Use uninitialized memory with caution. Any memory obtained from the kernel should be zeroed or otherwise initialized before being made available to a user process or device. Otherwise, information (disclosure of data, passwords, etc.) can be leaked. If device

interprets data sent to it, be sure the user cannot send anything that can harm the system. Finally, think about the possible effect of device operations. Suppose there are specific operations (e.g., reloading the firmware on an adapter board or formatting a disk) that could affect the system. In that case, those operations should be restricted to privileged users only.

Be careful of software from third parties, especially when the kernel is concerned, because if everybody has access to the source code, everybody can break and recompile the codes. The pre-compiled kernels found in your distribution are quite reliable, one should avoid running kernels compiled by an untrusted source. For example, a maliciously modified kernel could allow anyone to load a module via the init_ module. Note that the Linux kernel can be compiled to have no module support, so no security holes are introduced. All needed drivers must be built directly into the kernel itself to achieve this. . It is also possible, with version 2.2 and later kernels, to disable the loading of kernel modules after system boot via the capability mechanism.

Importance of Addressing Security and Design Issues:

- It is crucial to provide users of the software with adequate security since we must make sure that the data fed into the system are secure and kept private from the system itself.
- Any shortcomings in the software's design could cause it to perform incorrectly, which would be unsatisfactory to its users.

The bugs and broken authentication security issues:

- Bugs: The most frequent cause of software security problems are bugs. Almost all software has many types of defects.
- Broken Authentication: Authentication describes the procedure used to confirm that users are who they claim to be. Security problems develop when authentication-related functions behave improperly.

Let us see about Modularization Techniques:

A method of dividing a software system into numerous distinct modules is called modularization.

- Each module works independently.
- Depending on the situation, a module may be utilized multiple times. No need to use recursive writing.

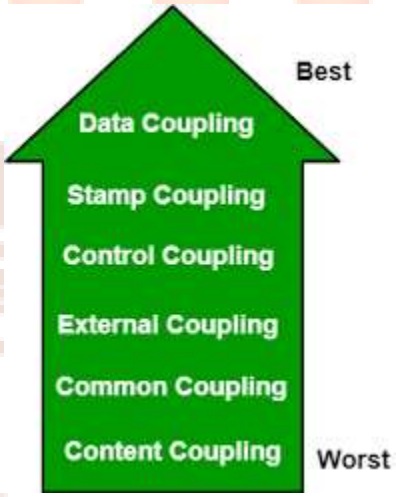
Coupling:

Dependency between the modules should be as low as possible.

The degree of interdependence between the software modules is measured by coupling.

If a software is having a low coupling rate, then it is said to be an excellent software.

Types of coupling:



Types of coupling is shown here:

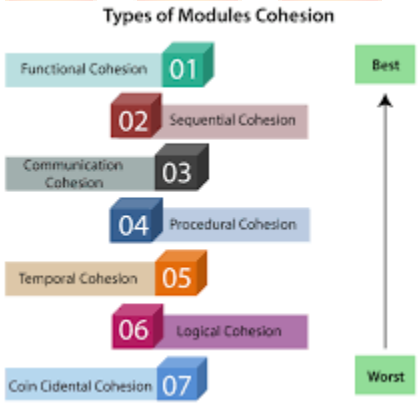
- Data Coupling: Data coupling is the transfer of data from one module to another.
- Stamp coupling: If two modules communicate via composite data items like structure, objects, etc., they are said to be stamp linked.
- Control Coupling: The manner or degree by which the execution of one software component is influenced by another software component.

- External Coupling: When one or more modules share an enforced format, interface, or communication protocol, this is known as external coupling.
- Common Coupling: Two or more functions sharing global data is known as common coupling, also known as global coupling.
- Content coupling: When one module alters or depends on the internal operations of another module, this is known as content coupling.

Next let us learn about Cohesion:

The greater the cohesion, the better is the program design.

In terms of cohesion, a module's degree of intra-dependability means that components that belong together should be kept together.



Functional Cohesion:

All the components inside the module are unconcerned with each other except for the task's execution that is linked to the issue.

Functional cohesiveness serves a singular, determined, and concentrated objective.

The module's components all carry out only the tasks that are required.

Sequential Cohesion:

The elements are involved in such a way that the data that is the outcome of one activity is used as the input data for the subsequent action.

Sequential cohesion offers a good connection and is simple to maintain.

Because the activities are useless even when employed simultaneously, it isn't easy to utilize sequential cohesion.

Communication Cohesion:

The pieces inside the module contribute to the actions that use the same input data or output data.

We cannot focus on all the activities at once; communicational cohesiveness is not adaptable in this way.

The links that produce interactions between the activities make up communication cohesion.

Communicational cohesiveness separates the functionally cohesive ones.

Procedure Cohesion:

If the module's elements are connected by sequence, then the activities are related; otherwise, they are not.

Sequential and procedural cohesion are similar concepts, but procedural cohesion's module pieces do not share any connections.

At the top of the hierarchy, like the main program's module, is procedural cohesion.

Temporal Cohesion:

The actions that are related in time include module-related components.

The startup and termination modules exhibit temporal coherence.

Because the module's components are unrelated to one another, they cannot be reused.

The optimal approach for maintaining temporal coherence is to initiate as late and end as soon as possible.

Logical Cohesion:

The module's components contribute to activities of the same kind or broad category.

Examples of tasks of the same kind or broad category that the module's components contribute to include report modules, display modules, and input-output modules.

Cohesion by Coincidence:

The module's components contribute to the nonsensical interactions between the activities.

The only difference between coincidental cohesion and logical cohesion is the types of actions involved.

Activities like rojak are a blend of coincidental coherence.

Self-Assessment Questions - 4

12. Security is an _____ issue that is often best handled at higher levels within the kernel, under the control of the_____ administrator.
13. In the official kernel distribution, the system calls _____ checks if the invoking process is authorized to load a module into the kernel.

6. DESIGN ISSUES

Let's now discuss the design considerations for creating device drivers that can function across various CPU and bus architectures. The issues that influence the device driver design can broadly be of two types:

1. CPU issues that influence device driver design
2. Bus issues that influence device driver design

6.1 CPU Issues That Influence Device Driver Design

A device driver should be designed to accommodate peripheral devices to operate on more than one CPU architecture. The following issues must be considered to make your drivers portable across CPU architectures:

i. Control status register (CSR) access
ii. I/O copy operation
iii. Direct memory access (DMA) operation
iv. Memory mapping
v. 64-bit versus 32-bit
vi. Memory barriers

i. ***Control status register (CSR) access:*** Many device drivers based on the UNIX operating system access a device's control status register (CSR) addresses directly through a device register structure. Some CPU architectures do not allow access to the device CSR addresses directly. Write one device driver with the appropriate conditional compilation statements so that the device driver can operate on both types of CPU architectures.

ii. ***I/O copy operation issues***: I/O copy operations can differ from one device driver to another due to the differences in CPU architectures. If you use techniques other than the generic kernel interfaces that digital technology provides for performing I/O copy operations, you may need help to write one device driver to operate on more than one CPU architecture or more than one CPU type within the same architecture.

iii. ***Direct memory access (DMA) operation issues***: Direct memory access (DMA) operations can differ from one device driver to another because of the different types of buses used and its DMA hardware support features.

iv. ***Memory mapping issues:*** Some CPU architectures do not support an application's use of a memory map section. You should design the device driver suitable for different types of memory map sections deployed to handle applications in different types of OS and for the one that do not use memory map sections, also

v. ***64-bit versus 32-bit:*** Consider while declaring data types for 32-bit and 64-bit CPU architectures. Pay careful attention to data types to make your device drivers work on 32-bit and 64-bit systems.

vi. ***Memory barriers issues:*** In certain architecture or a model of a computer system, a CPU may perform the memory operations in any order it likes, provided program causality appears to be maintained. Independent memory operations or read and write operations can be performed in random order without any problem, but this can be a problem for CPU-CPU interaction and I/O. So, some way of intervening to instruct the CPU to restrict the order is required. Memory barriers are interventions that impose a perceived partial ordering over the memory operations. Memory barrier (mb) acts as an interface that guarantees the ordering of operations. The mb interface is derived from the MB instruction. The MB instruction must be used in any system to guarantee correctly ordered access to I/O registers or memory that can be accessed via off-board DMA.

6.2 Bus Issues That Influence Device Driver Design

A device driver should be designed to accommodate peripheral devices that can operate on more than one bus architecture. Portability across bus architectures can be achieved when the bus architectures have common features and attributes. Write one device driver for a device that operates on the Industry Standard Architecture (ISA) and Extended Industry Standard Architecture (EISA) buses because their architectures have common features and attributes. It may not be feasible to write one device driver for multiple bus architectures if they exhibit dissimilar features and attributes.

Generally, the following issues must be considered to make the drivers portable across bus architectures. The following issues may apply to many bus architectures in general.

i. Bus-specific header file issues: Each bus implemented on different architecture has a specific header file. To write portable device drivers across multiple bus architectures, you need to consider how to include bus-specific header files in the include files section of your device driver.
ii. Bus-specific constant name issues: You must be aware of and define the bus names for the buses the driver will operate in order to develop portable device drivers that work across multiple bus architectures.
iii. Bus-specific issues related to the /etc/sysconfigtab database: The "sysconfigtab" file fragment contains device special file information, bus option data information, and physically contiguous memory usage information for statically and dynamically configured drivers. You have to consider the way the sysconfigtab file fragment gets appended to the /etc/sysconfigtab database.
iv. Bus-specific issues related to implementing the probe interface: The first argument associated with a driver's probe interface is bus specific. The second argument for a driver's probe interface is always a pointer to a controller structure. Controller structure pointer can be used to determine the bus to which the device controller is connected. To write portable device drivers across multiple bus architectures, you need to know the first argument associated with the probe interface for that bus.
v. Bus-specific issues related to implementing the slave interface: The first argument for a driver's slave interface is always a pointer to a device structure. The second argument associated with a driver's slave interface is bus specific. To write portable device drivers across multiple bus architectures, y The second argument connected to the slave interface for that bus is something you need to be aware of.
vi. Bus-specific issues for implementing the configure interface: A device driver's configure interface is called indirectly by the cfgmgr framework. You should know how this cfgmgr framework calls all single binary modules for registration and integration into the kernel.

Self-Assessment Questions - 5

14. Many device drivers based on the UNIX operating system access a device's "_____ "addresses directly through a device register structure.
15. Some CPU architectures do not support an application's use of a memory map section. (True/False)
16. The first argument associated with a driver's probe interface is "_____" specific.

7. SUMMARY

Let us recapitulate the important concepts discussed in this unit:

- Peripherals connected to a computer need special communication links for interfacing them with the CPU. Communication links resolve the differences between the central computer and each peripheral.
- A device driver is a software interface that allows the OS to provide Input-output (I/O) services to interact with an underlying device.
- Although device drivers are in fact add-on modules, they are an integral part of the system closely integrated with the Input/ Output Control System, which deals with I/O related system calls.
- In UNIX/Linux OS, device drivers are in the form of files and are usually linked onto the object code of the kernel.
- Applications run in user mode, and core operating system components run in kernel mode.
- The role of a device driver can be discussed in terms of mechanism and policy. As a programmer, you are free to design your driver in terms of mechanism and policy. Mechanisms refer to "what capabilities are to be provided," whereas policy refers to "how those capabilities can be used".
- UNIX/Linux OS distinguishes and classifies devices that are attached to the computer devices into (i) Character devices (ii) Block devices (iii) Network interfaces.
- Depending upon the device types, the modules for the device driver can be classified as (i) a char module (ii) a block module (iii) a network module.
- Security is a policy issue, and there should not be security holes in the kernel due to a bad module that gets loaded into the kernel.
- The issues that influence the device driver design can be due to (i) CPU issues that influence the device driver design (ii) Bus issues that influence the device driver design

8. GLOSSARY

Buffer. A device or storage area [memory] used to store data temporarily to compensate for differences in rates of data flow, time of occurrence of events, or amounts of data that can be handled by the devices or processes involved in the transfer or use of the data.

Bus: A common pathway along which data and control signals travel between hardware devices within a computer system.

Driver: A program that links a peripheral device or internal function to the operating system and provides for activation of all device functions.

Device Driver: Device Driver is to provide certain mechanisms to allow accessing data to and from the attached devices

Interface: A peripheral device that permits two or more devices to communicate

Peripheral Device: Equipment that is directly connected to a computer. A peripheral device can be used to input data; e.g., keypad, bar code reader, transducer, laboratory test equipment; or to output data; e.g., printer, disk drive, video system, tape drive, valve controller, motor controller

9. TERMINAL QUESTIONS

Short Answer Questions

1. What is a device driver? Explain.
2. Explain the role of device drivers in terms of mechanism and policy.
3. How do UNIX/Linux OS distinguish and classify devices that are attached to the computer devices? Explain.
4. What are the issues related to CPU architecture to make drivers portable? Explain briefly.
5. What are the issues to be considered related to bus architecture to make drivers portable? Explain briefly.

10. ANSWERS

Self-Assessment Questions

1. interfacing
2. driver
3. a) /dev
4. dynamically
5. .sys, dynamic link libraries (DLLs)
6. kernel space, user space
7. a) Windows loads device drivers and calls functions in the driver
8. True
9. b) char driver
10. filesystem
11. True
12. policy, system
13. init_module
14. control status register (CSR)
15. True
16. bus

Short Answer Questions

1. A device driver is a software interface that allows the OS to provide Input-output (I/O) services to interact with an underlying device. (Refer to section 9.2 for more details)
2. The role of device drivers is to provide their capabilities and how those capabilities can be used. Mechanism refers to capabilities, and policy refers to the use of capabilities. (Refer to section 9.3 for more details)
3. UNIX/Linux OS distinguishes and classifies devices that are attached to the computer devices into (i) Character devices (ii) Block devices (iii) Network interfaces. (Refer to section 9.4 for more details)

4. The following are to be considered to make drivers portable across CPU architectures: (i) Control status register (CSR) access issues. (ii) I/O copy operation. (iii) Direct memory access (DMA) operation etc. (Refer to section 9.6.1 for more details)
5. The issues to be considered related to bus architecture are (i) Bus- specific header file issues (ii) Bus-specific constant name issues, etc. (Refer to section 9.6.2 for more details)

11. SUGGESTED BOOKS

- Corbet, J., Kroah-Hartman, G., & Rubini, A. (2005). Linux Device Drivers. O'Reilly Media.
- *memory-barriers*. (n.d.). Retrieved September 12, 2012, from LINUX KERNEL MEMORY BARRIERS: http://www.kernel.org/doc/Documentation/memory-barriers.txt
- Microsoft. (n.d.). *What is a driver?* Retrieved December 18, 2012, from http://msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/windows/hardware/ff554678(v=vs.85).aspx