UNIT-IV

Introduction to Device Drivers

Introduction:- In computing, a device driver is a computer program that operates or controls a particular type of devices that is attached to a computer. A driver provides a software interface to hardware devices that enables operating system and other computer programs to access hardware functions.

A driver communicates with the device through the bus or communicates to the sub system to which the hardware connects. When a calling program invokes a routine in the driver, once the device send data back to the driver, the driver may invoke routine in the original calling program. Drivers are hardware dependent and operating system specific. They usually provide interrupt handling required for necessary asynchronous time dependent hardware interface.

Purpose of Device Drivers:-

The main purpose of device drivers is to provide abstraction by acting as a translator between hardware device and operating system. programmers can write the higher level application code independently which specifies the hardware the endeser is being using.

For example, a high level application interacts with a serial codes may have two functions.

1. Send data
2. Receive data.

Development:- Device drivers require low level access to hardware functions in order to operate, drivers typically

operate in a highly priviliged environment and can cause system operational issues if something goes wrong.

The task of writing drivers usually falls to software engineer or computer engineer who works for hardware development company. The clients can use hardware in optimum way. Typically the logical device drivers (LDD) is written by operating system vendor while physical device drivers (PDD) is implemented by the device drivers. Although this information can be learnt by reverse engineering. It is more difficult with h/w than with the s/w.

Applications:- Drivers operates in many different environment:

* Drivers interface with pointers, video adaptors, Network cards, sound cards of various slots of the modern systems.
* Low band width I/O buses of various slots
* Computer storage devices such as hard disk, CD ROM and floppy disk.
* Image scanners, digital cameras.

For hardware:-

* writing or reading from a device control register.
* Using some higher level interface.
* Using another lower level device driver.

For Software:-

* It allows the operating system in order to access hardware resources
* Implementing an interface for non-driver software.

Auto Configuration:- The kernal calls a device driver to determine what devices are available and to initialize them.

I/o operations:- The kernal calls a device driver to handle I/o operations on the device. These operations include opening the device to perform read and write and closing the device.

Special Request:- The kernal calls a device driver to handle special request to I/o calls.

Interrupt handling:- The kernal calls a device driver to handle interrupts from the devices that is capable of generating them.

Re-Initialization:- The kernal calls a device to handle to reinitialize the driver when the bus is reset.

User level request to system configuration utility:- The kernel calls a device driver to handle the request that results from the use of system configuration utility:-

* Device Driver configuration:-

It consists of the tasks necessary to incorporate device drivers into the kernel that makes them available to the system management and other utilities. There are two methods of device driver configuration.

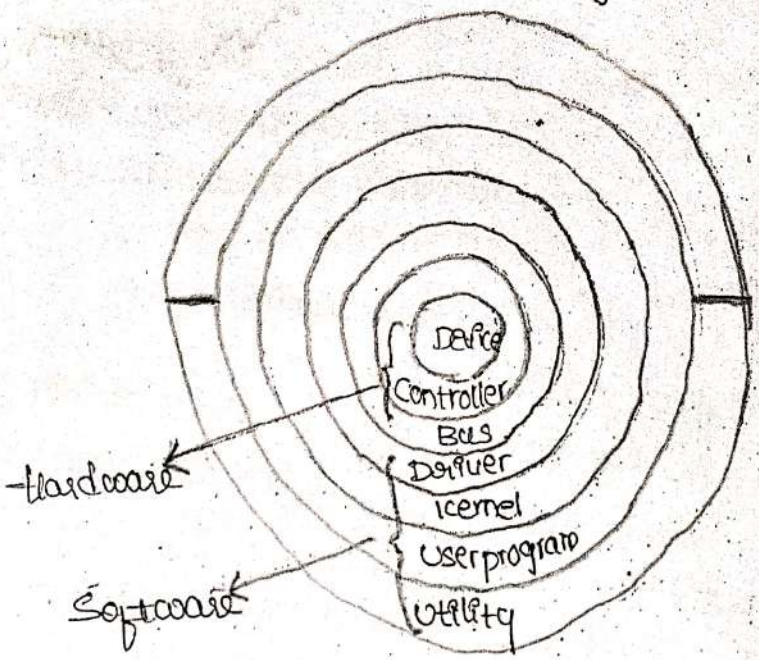
1. Static configuration:- It consists of task and tools necessary to link a device driver directly into the kernel.

2. Dynamic configuration:- It consists of task and tools necessary to link a device driver directly into the kernel at any point of time.

Configuration is a process associated with handling user level request to system configuration utility to

dynamically configure, unconfigure and reconfigure devices cfgmgr manager, frame work calls the drivers as a result of this utility request (place of device dd

Place of device drivers in digital UNIX:-



Device Controller:- A device controller is a hardware interface within a peripheral devices and computer. In other case a controller is an integral part of the device

Peripheral Devices:- A peripheral device is a hardware that connects to a computer system. It can be controlled by the commands by the computer and can send the data to the computer and receive the data from it.

Examples:-

* A data acquisition device
* line printer
* A disk or tape driver.

Device drivers:- A device driver communicates with a device by reading or writing through a bus through a peripheral device registers. A device driver run as part of the kernel software, manages and controls

the devices with digital UNIX, we can statically configure more device drivers into the kernel than there are physical devices in the hardware system. It makes calls on kernel, support interfaces to perform the task

3. Bus :- Bus is the data path b/w the main processor and the device controller. A BUS is a predefined set of logics, signals, timing and connectors that provides many types of device interfaces that can be easily combine with a computer system.

4. User program or utility :- User program or utility make the calls on the kernel but never directly calls a device driver that results in kernel making request of a device driver for example user can make a read system call which calls the driver read interface.

5. kernel :- the kernel makes a request to a device driver to perform operations on a particular device. Some of the request results directly from user program request for example block I/O and character I/O.

kernel calls the drivers to configure the requests such as configure, unconfigure and a query that results from a system uses the system configuration utility.

The kernel supports the tasks such as:

* Scheduling events
* Managing Buffer
* moving Initializing data
* waiting and sleeping

the devices with digital UNIX, we can statically configure more device drivers into the kernel than there are physical devices in the hardware system. It makes calls on kernel, support interfaces to perform the task

3. Bus :- Bus is the data path blw the main processor and the device controller. A BUS is a predefined set of logics, signals, timing and connectors that provides many types of device interfaces that can be easily combine with a computer system.

4. User program or Utility :- User program or utility make the calls on the kernel but never directly calls a device driver that results in kernel making request of a device driver for example user can make a read system call which calls the driver read interface.

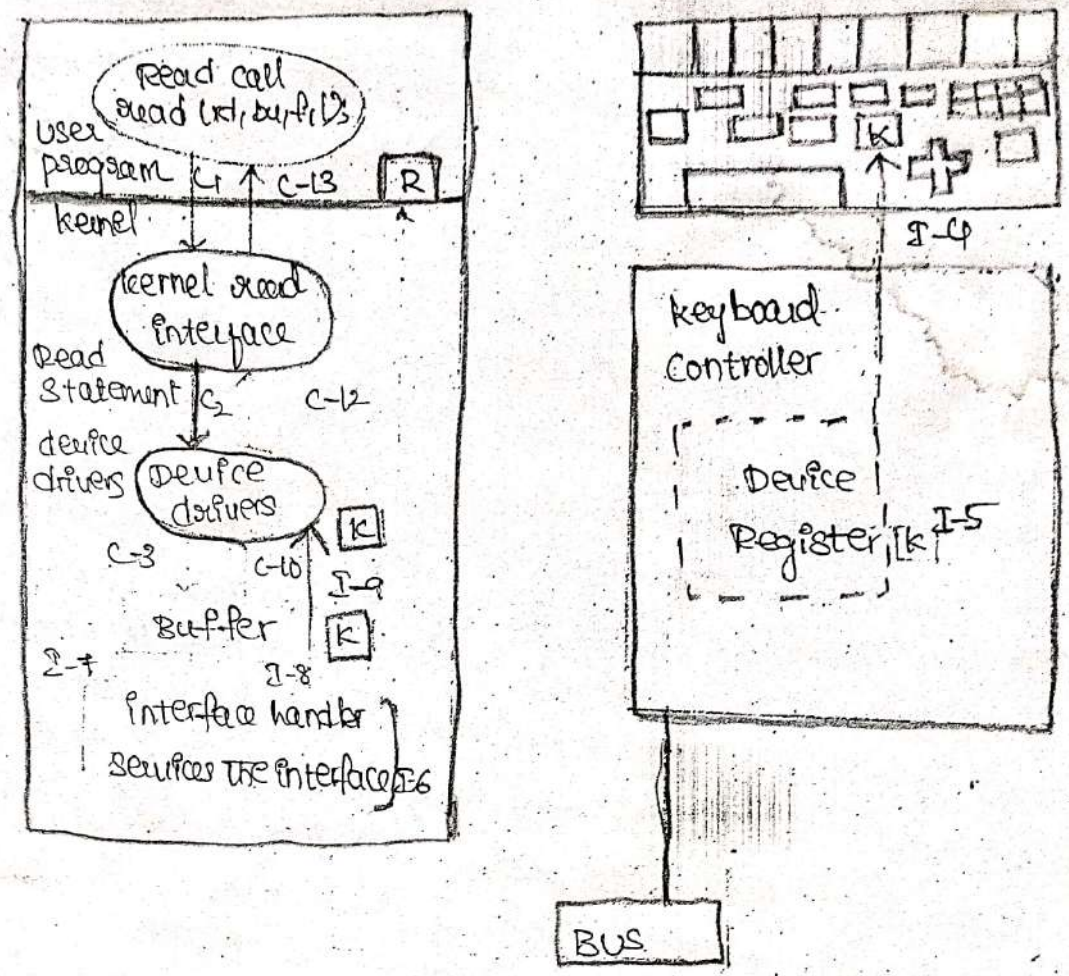
5. Kernel :- the kernel makes a request to a device driver to perform operations on a particular device. Some of the request results directly from user program request for example block I/O and character I/O.

kernel calls the drivers to configure the requests such as configure, unconfigure and a query that results from a system uses the system configuration utility.

The kernel supports the tasks such as:

* Scheduling events
* Managing Buffer
* Moving / Initializing data
* Waiting and sleeping

Example for Reading a character.



* The above diagram shows how digital UNIX processes read request of a single character. ~~character~~ processing, It shows in a raw mode it shows ~~how~~ ~~control~~ a simplified view of character processing, it shows how control can pass from a user program to the kernel to the device driver. It also shows that interrupt processing occurs asynchronously from other device drivers.

* The above diagram shows the flow of control between user program, kernel, device drivers and hardware, it shows the following sequence of events such as a read request is made to the device drivers (C1-C3).

* The character is captured by the hw (I_4-I_5)

* The interrupted is generated (I_6).

* The interrupt handler services the interrupt (I_7-I_8)

* The character is returned.

Character driver design issues → No conversion layer to character driver):- For example the program code that is presented in the device driver became a device/dev which appears on our Linux system is as follows.

Major and Minor Number:- Device driver have an major and minor number. For example /dev/devo1 & dev/null are associated with a minor number and /dev/tty0 & /dev/tty60 are associated with a driver with major number. The major number is used by the kernel to identify the correct device driver when the device is accessed. The role of minor number is device dependent and is handled internally with the driver. We can see major or minor number for each device by listing the device driver.

The device driver source code:-

The device driver source code consists of init() and exit(). However there are additional file operation function that are required for the character device

1. device_open():- It is called each time when the device is opened from the user space
2. dev_read():- It is called when the data is send from the device to user space
3. dev_write():- It is called when data is received from the device to the user space.
4. dev_release():- It is called when the data is closed by the user space

Driver have a class name and device name. In listing EBB (Exploring Beagre Box) is used as a class name and EBB char has a device name. This results in the creation of a device that appears on the file system has at /system/class/ebb/ebbchar.

There are some additional files such as:

* This code has a fixed maximum size of 256 characters
* This code is not multiprocess safe
* Ebb char init() function is much longer.
* PTR ERR funct() is a function that is defined in the Linux/err that retrieves the error number from pointer.
* The function Sprintf(), strlen() are available in the kernel is defined linux/kernel.h
* The functions in string.h are architecture dependent.

Registering a device:- Char devices are accessed through the device files usually stored in device. The major number tells us which driver handles which device files and the minor number is used only by the driver itself.

Adding a driver to the system is registering with the kernel. We can do this by using the register_chardevice and this function is defined by linux/fs.device-file.

Eg:- int register chrdev(unsigned int major, cost char *name, struct file_operations *fops);

where unsigned int major is the major number we want to request, const char *name as it will appear in procedure devices, struct file_operations *fops is a pointer to the file operations for the device driver.

Unregistering a device:- We cannot allow kernel module to be remodeled. If the device files is opened by a process and then we remove the kernel module using the file could cause a call to a memory location where the appropriate read/write functions are used.

Normally we do not want to allow something, we return an error code (negative numbers). However there is a counter which keeps track of how many procedures are using its own modules. There are three modules in linux/module.h.

* Implementing a language of higher level.

Types of device Drivers:-

There are several kinds of device drivers each handles a different kind of I/O. Block device drivers manage devices with physically addressable storage media such as disks. All other devices are considered as character devices. Two types of character device drivers are standard character device driver and stream character device drivers.

Block device drivers:- Devices that supports a file systems are known as block devices. Drivers written for these devices are known as Block device drivers. It also provide a character driver interface that allows utility programs to bypass the file system and access the device directly.

Character devices:- These drivers normally performs I/O in a byte stream. These includes tape drivers and serial ports. It also provides additional interface not present in block drivers such as I/O control commands, memory mapping and device pooling.

Byte stream I/O device drivers:-

The main task of any device driver is to perform I/O, and the device transfers the data to and from without using a specific device address. This is in contrast to the block device drivers where a part of the file system request identifies a specific location on the device.

Memory mapped device drivers:-

For certain devices such as frame buffer, application programs have direct access to device memory is more efficient than byte stream I/O. Applications can map

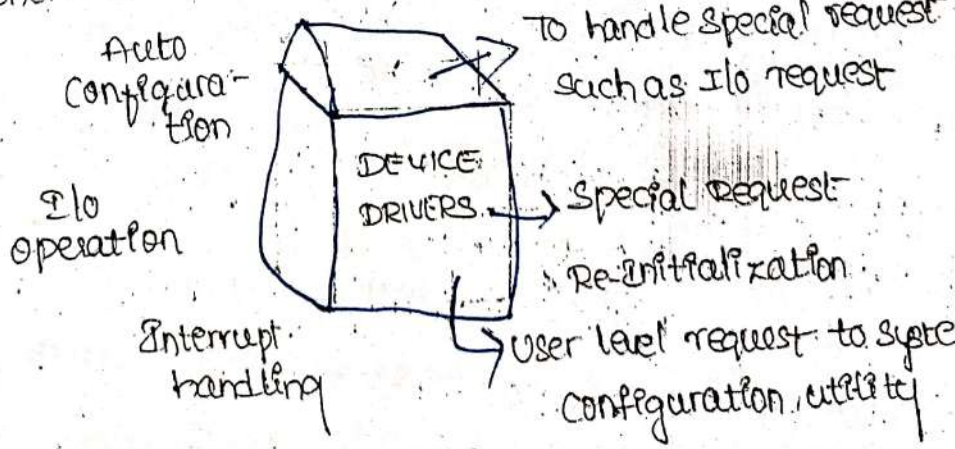
device memory into their address space using 'mmap' system call. To support memory mapping, device drivers implement 'seg' map and 'dev' map entry points usually do not define read and write entry points.

Stream drivers:- Streams is a separate programming model for writing a character driver. Devices that receives the data asynchronously such as network devices and terminal devices are suited to a stream implementation. It must support the loading and provides auto configuration.

Character driver 1 and device issues:-

A character device typically transperse the data to and from a user application, they behave like pipes or serial ports which is used to read or write the byte data into a character by character stream that provides frame work for many typical drivers that are required for interfacing to serial communication, video capture and audio devices. The main alternative of character device is block device driver. It allows a buffered array of cashed data to be viewed or manipulated with operations such as read and write.

When the kernel calls a device driver



with increase, decrease and display the counter.

MOD_INC_USE_COUNT: Increment the use count

MOD_DEC_USE_COUNT: Decrement the use count

MOD_IN_USE: Display the use count

RAM Disk Driver:-

The RAM disk driver does not require any hardware. It uses functions of malloc (memory allocation) to allocate the space to store files. The device configuration specifies the maximum size of the RAM disk. But the driver will only allocate the space as they are used thus even if we request a RAM disk of 256 MB, for example we write files with only 4MB size to the RAM disk, only this 4MB will be allocated. When files are deleted, the space previously allocated to the file is deallocated with the function free().

RTF Device.Driver data:-

The RAM disk driver provides a structure of type RAM data for RTF Device.Driver data with the following layout.

```
typedef struct
{
  ...
}
RTF_DRV_RAM_DATA;
```

RTF DRV RAM DATA specifies the maximum size of RAM disk in the center of 512 bytes MB, if set to zero, a RAM disk with a maximum size of 4MB is setup

RTF Device.Driver Type:-

The RTF Device.Driver Type may not set either RTF Device floppy or RTF device disk.

RTF Device.Device Number:-

This field is ignored and set to zero even if several RAM disks are used.

Extended RAM disk driver:- Extended RAM disk driver uses RT-Target 32 extended RAM management API to allocate a disk image. Extended RAM disk format themselves automatically acts as FAT-16, FAT-32 volume when the disk is mounted unless it was formatted already.

The Extended RAM disk driver contains device flag RTF_dev_DEVICE-NEW-LOCK that is supported for each RAM disk is as follows:

* RTF Device.Driver data
* RTF Device.Device Number
* RTF Device.Device flag

Linear flash Driver:-

This implements block device that directly addressable Linear flash chips. This driver contains of 2 layers

* MTDL(Memory technology Driver)
* flash chip independent flash driver.

which are responsible for handling the flash memory for example erasing, programming, mapping etc, The Linear flash disk driver 512 MB and the flash blocks. Each block has a header which is used for data sectors. It consists of the following characteristics.

1. Each erase unit has a size of atleast four bytes.
2. At least two blocks are available to implement a disk device.
3. All flash memory is always directly accessable.

RTF device.device number:- ①

1. The device is ignored and set to zero.

RTF Device.Device Type:- It may be set to either RTF device or 1. RTF device floppy. once the flash disk has been formatted, its value must be changed. This field must point a unique structure of Dow flash data is as follows.

```
type def struct
{
RTF_MTD* MTD driver;
void *MTD Data;
- - - - - -
}
RTF dow flash data;
```

USB Disk Driver:- The USB Disk driver requires RTSUB-32 and supports all USB Disk which adhere to the USB mass storage device specification class codes, subclass 2, protocal 0, protocal 1.

Before the USB DISK driver used RTSUB-32, call back the USB disk that must be registered using RTSUB-32 API functions.

RTF Device.Device Type:- It must set to be RTF Device 1. Disk for USB hard disk, flash Disk, Memory ~~stick~~ disk etc, for USB floppy disks drives it must be a RTF device floppy.

RTF Device.Device Number:- This field is the logical drive unit and on the USB dos device to access most USB mass storage devices contains just one disk with a logical disk unit of zero so this value should be usually zero.

RTF Device.Driver Data:- The USB Disk driver requires a pointer to a structure of type RTF Dow, USB data, the structure should be left uninitialized or initialized to zero.

BLOCK DRIVER

Ablock driver provides access to devices that transfer randomly accessible data in fixed-size blocks—disk drives, primarily. The Linux kernel sees block devices as being fundamentally different from char devices; as a result, block drivers have a distinct interface and their own particular challenges.

Efficient block drivers are critical for performance—and not just for explicit reads and writes in user applications. Modern systems with virtual memory work by shifting (hopefully) unneeded data to secondary storage, which is usually a disk drive. Block drivers are the conduit between core memory and secondary storage; therefore, they can be seen as making up part of the virtual memory subsystem.

Much of the design of the block layer is centered on performance. Many char devices can run below their maximum speed, and the performance of the system as a whole is not affected. The system cannot run well, however, if its block I/O subsystem is not well-tuned. The Linux block driver interface allows you to get the most out of a block device but imposes, necessarily, a degree of complexity that you must deal with. Happily, the 2.6 block interface is much improved over what was found in older kernels.

A block is a fixed-size chunk of data, the size being determined by the kernel. Blocks are often 4096 bytes, but that value can vary depending on the architecture and the exact filesystem being used. A sector, in contrast, is a small block whose size is usually determined by the underlying hardware. The kernel expects to be dealing with devices that implement 512-byte sectors. If your device uses a different size, the kernel adapts and avoids generating I/O requests that the hardware cannot handle. It is worth keeping in mind, however, that any time the kernel presents you with a sector number, it is working in a world of 512-byte sectors.

Eg: Hard disk drivers, CDROM drivers, etc.

Block driver Registration

Registration Block drivers, like char drivers, must use a set of registration interfaces to make their devices available to the kernel. The first step taken by most block drivers is to register themselves with the kernel. The function for this task is register_blkdev (which is declared in): int register_blkdev(unsigned int major, const char *name); The arguments are the major number that your device will be using and the associated name (which the kernel will display in /proc/devices). If major is passed as 0, the kernel allocates a new major number and returns it to the caller. As always, a negative return value from register_blkdev indicates that an error has occurred.

The corresponding function for canceling a block driver registration is: int unregister_blkdev(unsigned int major, const char *name); Here, the arguments must match those passed to register_blkdev, or the function returns -EINVAL and not unregister anything.

Disk Registration:

While register_blkdev can be used to obtain a major number, it does not make any disk drives available to the system. There is a separate registration interface that you must use to manage individual drives. Using this interface requires familiarity with a pair of new structures, so that is where we start.

Block device operations:

A set of function pointers:

open() and release(), called when a device handled by the driver is opened and closed

ioctl() for driver specific operations. unlocked_ioctl() is the non

BKL variant, and compat_ioctl() for 32 bits processes running on a 64 bits kernel

direct_access() required for XIP support

media_changed() and revalidate() required for removable media support

getgeo(), to provide geometry informations to userspace

Char devices make their operations available to the system by way of the file_ operations structure. Asimilar structure is used with block devices; it is struct block_device_operations, which is declared in . The following is a brief overview of the fields found in this structure; we revisit them in more detail when we get into the details of the sbull driver:

int (*open)(struct inode *inode, struct file *filp);

int (*release)(struct inode *inode, struct file *filp); Functions that work just like their char driver equivalents; they are called whenever the device is opened and closed. Ablock driver might respond to an open call by spinning up the device, locking the door (for removable media), etc. If you lock media into the device, you should certainly unlock it in the release method.

int (*ioctl)(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg); Method that implements the ioctl system call. The block layer first intercepts a large number of standard requests, however; so most block driver ioctl methods are fairly short.

int (*media_changed) (struct gendisk *gd); Method called by the kernel to check whether the user has changed the media in the drive, returning a nonzero value if so. Obviously, this method is only applicable to drives that support removable media (and that are smart enough to make a "media changed" flag available to the driver); it can be omitted in other cases.

The struct gendisk argument is how the kernel represents a single disk;

int (*revalidate_disk) (struct gendisk *gd);

The revalidate_disk method is called in response to a media change; it gives the driver a chance to perform whatever work is required to make the new media ready for use. The function returns an int value, but that value is ignored by the kernel.