Q.2 a. What are the important Distributions related to Linux? (8)

Answer:

The first version of Red Hat Enterprise Linux to bear the name originally came onto the market as "Red Hat Linux Advanced Server". In 2003 Red Hat re-branded Red Hat Linux Advanced Server to "Red Hat Enterprise Linux AS", and added two more variants, Red Hat Enterprise Linux ES and Red Hat Enterprise Linux WS.

While Red Hat uses strict trademark, rules to restrict free re-distribution of their officially supported versions of Red Hat Enterprise Linux, Red Hat freely provides the source code for the distribution's software even for software where this is not mandatory. As a result, several distributors have created re-branded and/or community-supported re-builds of Red Hat Enterprise Linux that can legally be made available, without official support from Red Hat.

The Fedora Project lists the following lineages for older Red Hat Enterprise releases:

- Red Hat Linux 6.2/7 → Red Hat Linux Enterprise Edition 6.2E
- Red Hat Linux 7.2 → Red Hat Enterprise Linux 2.1
- Red Hat Linux 10 beta 1 → Red Hat Enterprise Linux 3
- Fedora Core 3 → Red Hat Enterprise Linux 4
- Fedora Core 6 → Red Hat Enterprise Linux 5
- Fedora 12, 13 → Red Hat Enterprise Linux 6
- Fedora 19 → Red Hat Enterprise Linux 7

b. What are the Configuration facilities associated with Linux? (8)

Answer:

The main configuration file for **rsyslog** is **/etc/rsyslog.conf**. It consists of *global directives*,*rules* or comments (any empty lines or any text following a hash sign (#)). Both, global directives and rules are extensively described in the sections below.

23.1.1. Global Directives

Global directives specify configuration options that apply to the **rsyslogd** daemon. They usually specify a value for a specific pre-defined variable that affects the behavior of the **rsyslogd** daemon or a rule that follows. All of the global directives must start with a dollar sign (**$**). Only one directive can be specified per line. The following is an example of a global directive that specifies the maximum size of the syslog message queue:

```
$MainMsgQueueSize 50000
```

The default size defined for this directive (**10,000** messages) can be overridden by specifying a different value (as shown in the example above).

You may define multiple directives in your **/etc/rsyslog.conf** configuration file. A directive affects the behavior of all configuration options until another occurrence of that same directive is detected.

A comprehensive list of all available configuration directives and their detailed description can be found in **/usr/share/doc/rsyslog-<*version-number*>/rsyslog_conf_global.html**.

23.1.2. Modules

Due to its modular design, **rsyslog** offers a variety of *modules* which provide dynamic functionality. Note that modules can be written by third parties. Most modules provide additional inputs (see *Input Modules* below) or outputs (see *Output Modules* below). Other modules provide special functionality specific to each module. The modules may provide additional configuration directives that become available after a module is loaded. To load a module, use the following syntax:

```
$ModLoad <MODULE>
```

where $ModLoad is the global directive that loads the specified module and **<*MODULE*>**represents your desired module. For example, if you want to load the **Text File Input Module** (**imfile** — enables **rsyslog** to convert any standard text files into syslog messages), specify the following line in your **/etc/rsyslog.conf** configuration file:

```
$ModLoad imfile
```

rsyslog offers a number of modules which are split into these main categories:

- Input Modules — Input modules gather messages from various sources. The name of an input module always starts with the **im** prefix, such as **imfile**, **imrelp**, etc.
- Output Modules — Output modules provide a facility to store messages into various targets such as sending them across network, storing them in a database or encrypting them. The name of an output module always starts with the **om** prefix, such as **omsnmp**, **omrelp**, etc.
- Filter Modules — Filter modules provide the ability to filter messages according to specified rules. The name of a filter module always starts with the **fm** prefix.
- Parser Modules — Parser modules use the message parsers to parse message content of any received messages. The name of a parser module always starts with the **pm** prefix, such as**pmrfc5424**, **pmrfc3164**, etc.
- Message Modification Modules — Message modification modules change the content of a syslog message. The message modification modules only differ in their implementation from the output and filter modules but share the same interface.

- String Generator Modules — String generator modules generate strings based on the message content and strongly cooperate with the template feature provided by **rsyslog**.
- Library Modules — Library modules generally provide functionality for other loadable modules. These modules are loaded automatically by **rsyslog** when needed and cannot be configured by the user.

Q.3 a. Write the steps of how to add a system call to a Kernel. (8)

Answer:

0. Must have root access(you can type sudo before each command to have super user rights. It will prompt for super user password) For example

if you want to list directory just type ”sudo ls” and it will ask for super user password .

1. Download the latest version of the Linux kernel from www.kernel.org (for example Linux-2.6.35.5).

2. Unzip and untar the kernel. (I have placed it in /usr/src/.) (untar command : tar xvjf [filename])

3. In /usr/src/Linux-2.6.35.5/, Create a NewFolder and your new file that would contain your new system call lets say “myservice.c”.

4. In /usr/src/linux-2.6.35.5/arch/x86/include/asm/unistd_32.h, define an index for your system call.
Your index should be the number after the last system call defined in the list.

#define __NR_myservice 338

5. Also, you should increment the system call count.

#define NR_syscalls 339

6. /usr/src/linux-2.6.35.5/arch/x86/kernel/syscall_table.S, you should define a pointer to hold a reference to your system call routine. It is important that your data entry placement corresponds to the index you assigned to your system call.

.long sys_myservice

7. Create a Makefile in a NewFolder (folder you created in step 3) Add your system call to the Makefile by adding the line below.

#####Makefile Start#####

obj-y += myservice.o

#####Makefile End#######

8.In /src/linux-2.6.35.5/include/linux/syscalls.h
This file contain the declarations for system calls.
Add the following line at the end of the file:
asmlinkage long sys_myservice (int arg1, char* arg2);

9. Add directory path of the NewFolder to the Makefile (/usr/src/Linux-2.6.35.5/Makefile) Add your call to core-y (Search for regex: core-y.*+=). This directory will contain the source file, header file and the Makefile for our system call.
core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ NewFolder

10.For testing our new system call you will need to recompile Kernel.

11 Create a C program and check your system call

```
?
1
2  #include
3  #include
4  #include<linux/unistd.h>
5  #include<sys/syscall.h>
6  #include
7  #include<linux/types.h>
8
9  #define _myservice_ 338
10 int main()
11 {
12 long l=syscall(_myservice_,5); //test///test
13
14 int a=0;
15 printf("test%d",l);
16 return 1;
17 }
```

b. What are the basic data structures and algorithms in the Linux? (8)

Answer:

- Linked list, doubly linked list, lock–free linked list.

A relatively simple B+Tree implementation. I have written it as a learning exercise to understand how B+Trees work. Turned out to be useful as well.

A tricks was used that is not commonly found in textbooks. The lowest values are to the right, not to the left. All used slots within a node are on the left, all unused slots contain NUL values. Most operations simply loop once over all slots and terminate on the first NUL.

- Priority sorted lists usedfor mutexes, drivers, etc.

 Red – Black trees are used for scheduling, virtual memory management, to track file descriptors and directory entries,etc.
 Interval trees

 Radix trees, are used for memory management, NFS related lookups and networking related functionality.

A common use of the radix tree is to store pointers to struct pages;

 Priority heap, which is literally, a textbook implementation, used in the control group system..

 Hash functions,

 Hash tables used to implement inodes, filesystem integrity checks etc.

 Bit arrays, , which are used for dealing with flags, interrupts, etc.

 Semaphores and spin locks

 Binary search is used for interrupt handling, register cache lookup,etc.

 Binary search with B-trees

 Depth first search and variant used in directory configuration.

Performs a modified depth-first walk of the namespace tree, starting (and ending) at the node specified by start_handle. The callback function is called whenever a node that matches the type parameter is found. If the callback function returns a non-zero value, the search is terminated immediately and this value is returned to the caller.

 Breadth first search is used to check correctness of locking at runtime.

 Merge sort on linked lists is used for garbage collection, file system management etc.

 Bubble sort is amazingly implemented too, in a driver library.

Implements a linear-time string-matching algorithm due to Knuth, Morris, and Pratt [1]. Their algorithm avoids the explicit computation of the transition function DELTA altogether. Its matching time is O(n), for n being length(text), using just an auxiliary function PI[1..m], for m being length(pattern), precomputed from the pattern in time O(m). The array PI allows the transition function DELTA to be computed efficiently "on the fly" as needed. Roughly speaking, for any state "q" = 0,1,...,m and any character "a" in SIGMA, the value PI["q"] contains the information that is independent of "a" and is needed to compute DELTA("q", "a"). Since the array PI has only m entries, whereas DELTA has O(m|SIGMA|) entries, we save a factor of |SIGMA| in the preprocessing time by computing PI rather than DELTA.

Q.4 a. What is the process? How to specify a virtual address space for a process? (8)

Answer:

A *process* is an *executing* (i.e., running) instance of a *program*. Processes are also frequently referred to as *tasks*.

A program is an *executable file* that is held in *storage*. Storage refers to devices or media that can retain data for relatively long periods of time (e.g., years or even decades), such as (HDDs), optical disks and magnetic tape. This contrasts with *memory*, whose contents can be accessed (i.e., read and written to) at extremely high speeds but which are retained only temporarily (i.e., while in use or only as long as the power supply remains on).

An executable file is a *binary file* (i.e., a file at least part of which is not *plain text*) that has been *compiled* (i.e., converted using a special type of program called a *compiler*) from *source code* into machine *machine code*, which is a pattern of bytes that can be read directly by a central processing unit (CPU). Source code is the version of software as it is originally *written* (i.e., typed into a computer) by a human in plain text (i.e., human readable alphanumeric characters). A CPU is the main logic unit of a computer.

From a user perspective, the address space is a flat linear address space but predictably, the kernel's perspective is very different. The address space is split into two parts, the userspace part which potentially changes with each full context switch and the kernel address space which remains constant. The location of the split is determined by the value of PAGE_OFFSET which is at 0xC0000000 on the x86. This means that 3GiB is available for the process to use while the remaining 1GiB is always mapped by the kernel.

PGDs) is reserved at PAGE_OFFSET for loading the kernel image to run. 8MiB is simply a reasonable amount of space to reserve for the purposes of loading the kernel image. The kernel image is placed in this reserved space during kernel page tables initialization. The location of the array is usually at the 16MiB mark to avoid using ZONE_DMA but not always. With NUMA architectures, portions of the virtual mem_map will be scattered throughout this region and where they are actually located is architecture dependent.

b. What is the difference between block devices caching and paging in Linux? (8)

Answer:

From a software point of view with the 2.6 Linux kernel, swap files are just as fast as swap partitions. The kernel keeps a map of where the swap file exists, and accesses the disk directly, bypassing caching and filesystem overhead. Red Hat recommends using a swap partition.[With a swap partition one can choose where on the disk it resides and place it where the disk throughput is highest. The administrative flexibility of swap files can outweigh the other advantages of swap partitions. For example, a swap file can be placed on any drive, can be set to any desired size, and can be added or changed as needed. A swap *partition*, however, is not as flexible as a file, as it cannot be changed without using tools to resize it, generally outside the operating system that uses the swap partition.

Linux supports using a virtually unlimited number of swapping devices, each of which can be assigned a priority. When the operating system needs to swap pages out of physical memory, it

uses the highest-priority device with free space. If multiple devices are assigned the same priority, they are used in a fashion similar to level 0 RAID arrangements. This provides improved performance as long as the devices can be accessed efficiently in parallel. Therefore, care should be taken assigning the priorities. For example, swaps located on the same physical disk should not be used in parallel, but in order ranging from the fastest to the slowest (i.e. the fastest having the highest priority).

Q.5 a. What is Inter process Communication? How to create IPC with sockets? (8)

Answer:

Interprocess communication (IPC)
is the transfer of data among processes. For example,
a Web browser may request a Web page from a Web server, which then sends HTML
data.This transfer of data usually uses sockets in a telephone-like connection.

Sockets provide point-to-point, two-way communication between two processes. Sockets are very versatile and are a basic component of interprocess and intersystem communication. A socket is an endpoint of communication to which a name can be bound. It has a type and one or more associated processes.

Sockets exist in communication domains. A socket domain is an abstraction that provides an addressing structure and a set of protocols. Sockets connect only with sockets in the same domain. Twenty three socket domains are identified (see <sys/socket.h>), of which only the UNIX and Internet domains are normally used Solaris 2.x Sockets can be used to communicate between processes on a single system, like other forms of IPC.

The UNIX domain provides a socket address space on a single system. UNIX domain sockets are named with UNIX paths. Sockets can also be used to communicate between processes on different systems. The socket address space between connected systems is called the Internet domain.

Internet domain communication uses the TCP/IP internet protocol suite.

Socket types define the communication properties visible to the application. Processes communicate only between sockets of the same type. There are five types of socket.

A stream socket

-- provides two-way, sequenced, reliable, and unduplicated flow of data with no record boundaries. A stream operates much like a telephone conversation. The socket type is SOCK_STREAM, which, in the Internet domain, uses Transmission Control Protocol (TCP).

A datagram socket

-- supports a two-way flow of messages. A on a datagram socket may receive messages in a different order from the sequence in which the messages were sent. Record boundaries in the data are preserved. Datagram sockets operate much like passing letters back and forth in the mail. The socket type is SOCK_DGRAM, which, in the Internet domain, uses User Datagram Protocol (UDP).

A sequential packet socket

-- provides a two-way, sequenced, reliable, connection, for datagrams of a fixed maximum length. The socket type is SOCK_SEQPACKET. No protocol for this type has been implemented for any protocol family.

A raw socket

provides access to the underlying communication protocols.

These sockets are usually datagram oriented, but their exact characteristics depend on the interface provided by the protocol.

Socket Creation and Naming

int socket(int domain, int type, int protocol) is called to create a socket in the specified domain and of the specified type. If a protocol is not specified, the system defaults to a protocol that supports the specified socket type. The socket handle (a descriptor) is returned. A remote process has no way to identify a socket until an address is bound to it. Communicating processes connect through addresses. In the UNIX domain, a connection is usually composed of one or two path names. In the Internet domain, a connection is composed of local and remote addresses and local and remote ports. In most domains, connections must be unique.

int bind(int s, const struct sockaddr *name, int namelen) is called to bind a path or internet address to a socket. There are three different ways to call bind(), depending on the domain of the socket.

- For UNIX domain sockets with paths containing 14, or fewer characters, you can:
- #include <sys/socket.h>
- ...
- bind (sd, (struct sockaddr *) &addr, length);
- If the path of a UNIX domain socket requires more characters, use:
- #include <sys/un.h>
- ...
- bind (sd, (struct sockaddr_un *) &addr, length);
- For Internet domain sockets, use
- #include <netinet/in.h>
- ...
- bind (sd, (struct sockaddr_in *) &addr, length);

In the UNIX domain, binding a name creates a named socket in the file system. Use unlink() or rm () to remove the socket.

b. What are pipes? Explain the usage of Pipes in Linux. (8)

Answer:

Pipe is a symbol used to provide output of one command as input to another command. The output of the command to the left of the pipe is sent as input to the command to the right of the pipe. The symbol is |.

For example:
$ cat apple.txt | wc

In the above example the output of apple.txt file will be sent as input for wc command which counts the no. of words in a file. The file for which the no. of words counts is the file apple.txt. Pipes are useful to chain up several programs, so that multiple commands can execute at once without using a shell script.

Piping in a shell helps in redirecting the output of one command as an input to the other. The main advantage of piping is that it helps in combining simple commands to achieve a particular purpose. In fact even if you know some basic commands in Linux, you can make the best use of your knowledge by using piping. One more advantage is that the total number of commands will be reduced because what can be achieved from a single new command can be achieved by combining two already present commands. In addition to this, the number of options for a particular command can be reduced because a new option is not required if we can achieve the same using piping.

Q.6 a. What are the basic principles of File System? (8)

Answer:

Everything is a file :

In Linux everything is considered as a file. Unlike windows, all devices like cd-rom, Ram, keyboard ,Monotor etc are considered as specific files. And these devices work accourding to its file configuration.

2. Configuration data of the system is stored in text usually named with extension, .conf :

All the configuration files of the system are saved as text file having an extension .conf. We need to edit these files using text editors like VI, VIM, PICO, NANO etc to change the system configuration. And system updation is done by modifying these conf files. Most of the system conf files will be located under the location /etc.

3. Connection pipes for standard I/O to chain programs :

The pipe operation connects the standard O/P of one program to the standard I/P of an another. And such a chain of programs connected is called a *pipeline.*

b. Explain the representation of File Systems in the kernel with the help of diagram. (8)

Answer:

The Virtual File System (VFS)

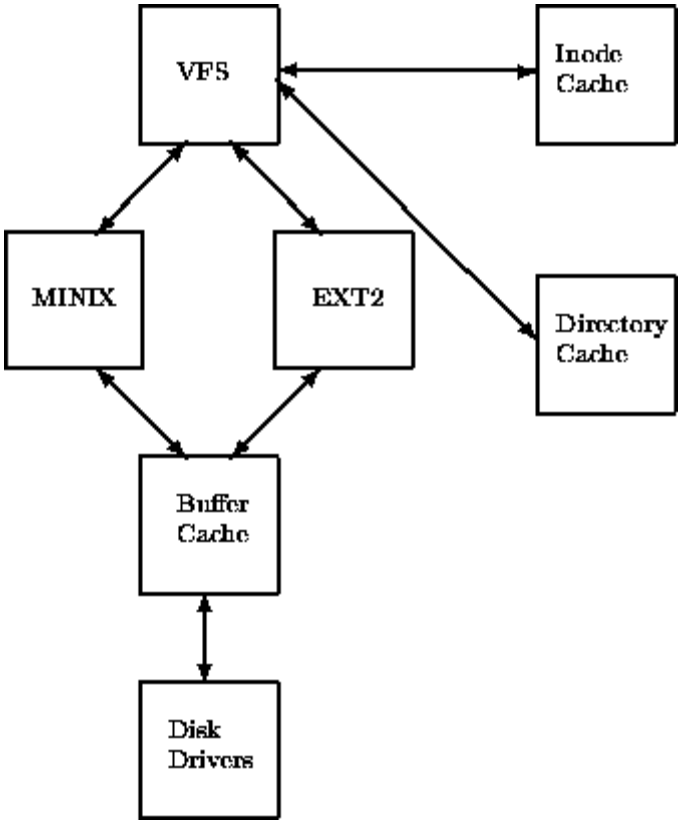

Figure: A Logical Diagram of the Virtual File System

Figure ❑ shows the relationship between the Linux kernel's Virtual File System and it's real file systems. The virtual file system must manage all of the different file systems that are mounted at any given time. To do this it maintains data structures that describe the whole (virtual) file system and the real, mounted, file systems.

Rather confusingly, the VFS describes the system's files in terms of superblocks and inodes in much the same way as the EXT2 file system uses superblocks and inodes. Like the EXT2 inodes, the VFS inodes describe files and directories within the system; the contents and topology of the Virtual File System. From now on, to avoid confusion, I will write about VFS inodes and VFS superblocks to distinquish them from EXT2 inodes and superblocks.

As each file system is initialised, it registers itself with the VFS. This happens as the operating system initialises itself at system boot time. The real file systems are either built into the kernel itself or are built as loadable modules. File System modules are loaded as the system needs them, so, for example, if the VFAT file system is implemented as a kernel module then it is only loaded when a VFAT file system is mounted. When a block device based file system is mounted, and this includes the root file system, the VFS must read its superblock. Each file system type's superblock read routine must work out the file system's topology and map that information onto a VFS superblock data structure. The VFS keeps a list of the mounted file systems in the system together with their VFS superblocks. Each VFS superblock contains information and pointers to routines that perform particular functions. So, for example, the superblock representing a

mounted EXT2 file system contains a pointer to the EXT2 specific inode reading routine. This EXT2 inode read routine, like all of the file system specific inode read routines, fills out the fields in a VFS inode. Each VFS superblock contains a pointer to the first VFS inode on the file system. For the root file system, this is the inode that represents the ``/'' directory. This mapping of information is very efficient for the EXT2 file system but moderately less so for other file systems.

As the system's processes access directories and files system routines are called which traverse the VFS inodes in the system.

For example, typing *ls* for a directory or *cat* for a file cause the the Virtual File System to search through the VFS inodes which represent the file system. As every file and directory on the system is represented by a VFS inode then a number of inodes will be being repeatedly accessed. These inodes are kept in the inode cache which makes access to them quicker. If an inode is not in the inode cache, then a file system specific routine must be called in order to read the appropriate inode. The action of reading the inode causes it to be put into the inode cache and further accesses to the inode keep it in the cache. the less used VFS inodes get removed from the cache.

Q.7 a. What is the difference between Polling and Interrupts? (8)

Answer:

Hardware is slow. That is, in the time it takes to get information from your average device, the CPU could be off doing something far more useful than waiting for a busy but slow device. So to keep from having to **busy-wait** all the time, **interrupts** are provided which can interrupt whatever is happening so that the operating system can do some task and return to what it was doing without losing information. In an ideal world, all devices would probably work by using interrupts. However, on a PC or clone, there are only a few interrupts available for use by your peripherals, so some drivers have to poll the hardware: ask the hardware if it is ready to transfer data yet. This unfortunately wastes time, but it sometimes needs to be done.

Also, some hardware (like memory-mapped displays) is as fast as the rest of the machine, and does not generate output asyncronously, so an interrupt-driven driver would be rather silly, even if interrupts were provided.

In Linux, many of the drivers are interrupt-driven, but some are not, and at least one can be either, and can be switched back and forth at runtime. For instance, the lp device (the parallel port driver) normally polls the printer to see if the printer is ready to accept output, and if the printer stays in a not ready phase for too long, the driver will sleep for a while, and try again later. This improves system performance. However, if you have a parallel card that supplies an interrupt, the driver will utilize that, which will usually make performance even better.

There are some important programming differences between interrupt-driven drivers and polling drivers. To understand this difference, you have to understand a little bit of how system calls work under . The kernel is not a separate task under . Rather, it is as if each process has a copy of the kernel. When a process executes a system call, it does not transfer control to another process, but rather, the process changes execution modes, and is said to be ``in kernel mode.'' In this mode, it executes kernel code which is trusted to be safe

b. Explain in detail DMA operation with the help of example. (8)

Answer:

Direct memory access (**DMA**) is a feature of modern computers that allows certain hardware subsystems within the computer to access system memory independently of the central processing unit (CPU).

Without DMA, when the CPU is using programmed input/output, it is typically fully occupied for the entire duration of the read or write operation, and is thus unavailable to perform other work. With DMA, the CPU initiates the transfer, does other operations while the transfer is in progress, and receives an interrupt from the DMA controller when the operation is done. This feature is useful any time the CPU cannot keep up with the rate of data transfer, or where the CPU needs to perform useful work while waiting for a relatively slow I/O data transfer. Many hardware systems use DMA, including disk drive controllers, graphics cards, network cards and sound cards. DMA is also used for intra-chip data transfer in multi-core processors. Computers that have DMA channels can transfer data to and from devices with much less CPU overhead than computers without DMA channels. Similarly, a processing element inside a multi-core processor can transfer data to and from its local memory without occupying its processor time, allowing computation and data transfer to proceed in parallel.

For example, a sound card may need to access data stored in the computer's RAM, but since it can process the data itself, it may use DMA to bypass the CPU. Video cards that support DMA can also access the system memory and process graphics without needing the CPU. Ultra DMA hard drives use DMA to transfer data faster than previous hard drives that required the data to first be run through the CPU.

Q.8 a. Write short notes on: (8)

(i) IP

(ii) UDP

(iii) TCP

Answer:

1. IP

The Internet **Protocol** (**IP**) is the principal communications protocol in the Internet protocol suite for relaying datagrams across network boundaries. Its routing function enables internetworking, and essentially establishes the Internet.

IP, as the primary protocol in the Internet layer of the Internet protocol suite, has the task of delivering packets from the source host to the destination host solely based on the IP addresses in the packet headers. For this purpose, IP defines packet structures thatencapsulate the data to be delivered. It also defines addressing methods that are used to label the datagram with source and destination information.

Historically, IP was the connectionless datagram service in the original *Transmission Control Program* introduced by Vint Cerf and Bob Kahn in 1974; the other being the connection-

oriented Transmission Control Protocol (TCP). The Internet protocol suite is therefore often referred to as TCP/IP.

The first major version of IP, Internet Protocol Version 4 (IPv4), is the dominant protocol of the Internet. Its successor is Internet Protocol Version 6 (IPv6).

2. UDP

The **User Datagram Protocol** (**UDP**) is one of the core members of the Internet protocol suite (the set of network protocols used for the Internet). With UDP, computer applications can send messages, in this case referred to as *datagrams*, to other hosts on an Internet Protocol (IP) network without prior communications to set up special transmission channels or data paths. The protocol was designed by David P. Reed in 1980 and formally defined in RFC 768.

UDP uses a simple transmission model with a minimum of protocol mechanism.[1] It has no handshaking dialogues, and thus exposes any unreliability of the underlying network protocol to the user's program. As this is normally IP over unreliable media, there is no guarantee of delivery, ordering, or duplicate protection. UDP provides checksums for data integrity, and port numbers for addressing different functions at the source and destination of the datagram.

UDP is suitable for purposes where error checking and correction is either not necessary or performed in the application, avoiding the overhead of such processing at the network interface level. Time-sensitive applications often use UDP because dropping packets is preferable to waiting for delayed packets, which may not be an option in a real-time system.[2] If error correction facilities are needed at the network interface level, an application may use the Transmission Control Protocol (TCP) or Stream Control Transmission Protocol (SCTP) which are designed for this purpose.

3. TCP

The **Transmission Control Protocol** (**TCP**) is one of the core protocols of the Internet protocol suite (IP), and is so common that the entire suite is often called *TCP/IP*. TCP provides reliable, ordered, error-checked delivery of a stream of octets between programs running on computers connected to a local area network, intranet or the public Internet. It resides at the transport layer.

Web browsers use TCP when they connect to servers on the World Wide Web, and it is used to deliver email and transfer files from one location to another. HTTP, HTTPS, SMTP, POP3, IMAP, SSH, FTP, Telnet and a variety of other protocols are typically encapsulated in TCP.

Applications that do not require the reliability of a TCP connection may instead use theconnectionless User Datagram Protocol (UDP), which emphasizes low-overhead operation and reduced latency rather than error checking and delivery validation.

b. With the help of diagram explain ARP. What are the network devices associated with Linux? (8)

Answer:

ARP, the Address Resolution Protocol, is a member of the TCP/IP protocol suite that is used to translate between logical IP addresses, and physical MAC addresses. It accomplishes this task by building a correspondence table of IP and MAC addresses, using specialized pack ets, broadcast on the local network

Physical Network Interfaces

eth0, eth8, radio0, wlan19, .. always represent an actual network hardware device such as a NIC, WNIC or some other kind of Modem. As soon as the device driver is loaded into the Kernel a corresponding physical network interface becomes present and available.

Any physical network interface is a named software representation by the operating system to the user to enable him to configure the hardware network device and also to integrate it into programs and scripts.

Virtual Network Interfaces

lo, eth0:1, eth0.1, vlan2, br0, pppoe-dsl, gre0, sit0 tun0, imq0, teql0, .. are virtual network interfaces that do NOT represent an existent hardware device but are linked to one (otherwise they would be useless). Virtual network interfaces were invented to give the system administrator maximum flexibility when configuring a Linux-based operating system. A virtual network interface is generally associated with a physical network interface (eth6) or another virtual interface (eth6.9) or be stand alone such as the loopback interface.

Q.9 a. What is Kernel Daemon? Explain the term Debugging. (8)

Answer:

A kernel module is a code that can be loaded into the kernel image at will, without requiring users to rebuild the kernel or reboot their computer. Modular design ensures that you do not have to make a monolithic kernel that contains all code necessary for hardware and situations.
Common kernel modules are device drivers, which directly access computer and peripheral hardware.
The debugger needs al least a translation for kernel addresses. This should be
created by the user when configuring TRACE32 for Linux Debugging. Moreover, the debugger can scan the kernel MMU tables and held a local copy of them. However, this is not needed if TableWalk is enabled. The debugger MMU translation tables are maintained and used only inside the debugger software,
T Training Linux Debugging

14 Basic terms on Embedded Linux
4.) Run-Mode vs. Stop-Mode Debugging

There are two principle alternatives of debugging a Linux target: hardware based and software based. This
chapter gives a small introduction about the differences to help you understand the operation theory of the
hardware based debugger TRACE32 in conjunction with a Linux target.
a.) Hardware Based Debuggers
TRACE32 is a hardware based debugger, i.e. it uses special hardware to access target, processor and memory (e.g. by using the JTAG interface). No active
target software is required, there are no software requirements at all at the target. This allows debugging of bootstraps (right from the reset vector), interrupts, and any other software. Even if the target application runs into a complete system crash, you
are still able to access the memory contents (post mortem debugging).
Because the debugging execution engine is part of the target program, all software restrictions apply to the debugger, too. In the case of a gdbserver for example, which is a user application, the debugger can only access the resources of the currently debugged processes. In this case, it is not possible to access the
kernel and other processes.

b. What are the problems associated with Multi-processor systems? (8)

Answer:

- Scheduling multiprocessor tasks on parallel processors available in time windows.

- Scheduling multiprocessor tasks on parallel processors available in time
- Scheduling divisible tasks on other architectures.
- Scheduling multiple divisible tasks in a system with many originators.

1. Cache coherence

The effects of synchronization variables can have a major impact on cache performance, since they can
have a high degree of read/write sharing and often induce large amounts of false sharing.

2. Cache miss latency In addition to the greater frequency of cache misses due to sharing, SMMP programmers are faced with
the problem that cache misses cost more in multiprocessors regardless of their cause.

3. the problem that cache misses cost more in multiprocessors regardless of their cause. The latency increase
to complete an operation, resulting in higher overhead even if there is no contention for the locks.

Reentrancy

A routine is reentrant if the same copy in memory can be shared by more than one user. Reentrant routines do not maintain static data between calls; all data is provided by the caller of the function. Any caller-specific data that the routine maintains must be stored in an area specific to that call.

Most standard driver routines are designed to be reentrant. For example, in a driver's I/O dispatch routines, call-specific data is maintained in each individual IRP and passed from one driver to the next. Each driver must either complete the IRP or mark the IRP pending before passing it to the next driver on the stack. Only the **DriverEntry** and *Unload* routines are not reentrant.

Concurrency

Two routines that can run at the same time are said to be *concurrent*. For a driver, concurrency generally means that the operating system (usually the I/O or PnP manager) might call one routine before a previously called routine has completed. For example, the system could call a driver's *Cancel* routine while its *DispatchRead* routine is running.

When two routines can run concurrently, you must ensure that any shared, writable data is accessed by only one routine at a time unless all such accesses are read-only. The data might be in a shared memory buffer, in the device extension, or in a global variable. This means you must use locks, interlocked routines, or some other synchronization technique to prevent conflicts.

Some drivers manage more than one device; others can perform I/O on more than one file at a time. Windows typically calls the standard driver routines to perform a specific task on behalf of a particular driver object, device object, or file object. Whether Windows calls two routines concurrently thus depends on the driver, device, or file objects that each such routine uses. For example, the system might call a driver's *DispatchPnp* routine to handle an IRP_MN_REMOVE_DEVICE request for one device while the same routine is handling an IRP_MN_START_DEVICE request on behalf of another device, because the two requests affect different device objects. However, the system would not call these routines concurrently if the two requests were targeted at the same device.

Tables 2, 3, and 4 in the following sections list the concurrency of standard driver routines with respect to driver objects, device objects, and file objects. Refer to the tables to find out whether the operating system calls two routines concurrently with the same object. In the tables:

Yes means that the system might call the routine listed at the top while the routine at the left is running.

No means that the system does not call the routine listed at the top while the routine at the left is running.

Maybe means that whether the routines can run concurrently depends on how the driver is implemented. For example, whether a particular worker thread routine can be called concurrently depends on what the worker thread does.

For example, in a driver that manages two or more devices, the *StartIo* routine for one device can run concurrently with the *DispatchRead* routine for another device. However, if the two requests target the same device, these two routines cannot run concurrently. Thus, the *StartIo* and *DispatchRead* routines are concurrent with respect to the driver object, but not with respect to the device object.

TEXT BOOK

I. Linux Kernel Internals, M. Beck, H. Bome, et al, Pearson Education, Second Edition, 2001.