Unit 5: Case study on Linux

1. Explain evolution of UNIX

- UNIX development was started in 1969 at Bell Laboratories in New Jersey.
- Bell Laboratories was (1964–1968) involved on the development of a multi-user, time-sharing operating system called Multics (Multiplexed Information and Computing System). Multics was a failure. In early 1969, Bell Labs withdrew from the Multics project.
- Bell Labs researchers who had worked on Multics (Ken Thompson, Dennis Ritchie, Douglas McIlroy, Joseph Ossanna, and others) still wanted to develop an operating system for their own and Bell Labs' programming, job control, and resource usage needs.
- When Multics was withdrawn Ken Thompson and Dennis Ritchie needed to rewrite an operating system in order to play space travel on another smaller machine The result was a system called UNICS (UNiplexed Information and Computing Service)
- The first version of Unix was written in the low-level PDP-7(Programmed data process) assembler language. Later, a language called TMG was developed for the PDP-7 by R. M. McClure. Using TMG(TransMoGrifier)to develop a FORTRAN compiler, Ken Thompson instead ended up developing a compiler for a new high-level language he called B, based on the earlier BCPL (Basic Combined Programming **Language**) language developed by Martin Richard. When the PDP-11 computer arrived at Bell Labs, Dennis Ritchie built on B to create a new language called C. Unix components were later rewritten in C, and finally with the kernel itself in 1973.

- Unix V6, released in 1975 became very popular. Unix V6 was free and was distributed with its source code.
- In 1983, AT&T released Unix System V which was a commercial version.
- Meanwhile, the University of California at Berkeley started the development of its own version of Unix. Berkeley was also involved in the inclusion of Transmission Control Protocol/Internet Protocol (TCP/IP) networking protocol.
- The following were the major mile stones in UNIX history early 1980's
- AT&T was developing its System V Unix.
- Berkeley took initiative on its own Unix BSD (Berkeley Software Distribution) Unix.
- Sun Microsystems developed its own BSD-based Unix called SunOS and later was renamed to Sun Solaris.
- Microsoft and the Santa Cruz operation (SCO) were involved in another version of UNIX called XENIX.
- Hewlett-Packard developed HP-UX for its workstations.
- DEC released ULTRIX.
- In 1986, IBM developed AIX (Advanced Interactive eXecutive).

2. What is LINUX operating system?

- From smartphones to cars, supercomputers and home appliances, the Linux operating system is everywhere.

What is Linux?

Just like Windows XP, Windows 7, Windows 8, and Mac OS X, Linux is an operating system. An operating system is software that manages all of the hardware resources associated with your desktop or laptop. To put it simply – the operating system manages the communication between your software and your hardware. Without the operating system (often referred to as the “OS”), the software wouldn’t function.

The OS is comprised of a number of pieces:

- **The Bootloader:** The software that manages the boot process of your computer. For most users, this will simply be a splash screen that pops up and eventually goes away to boot into the operating system.

- **The kernel:** This is the one piece of the whole that is actually called "Linux". The kernel is the core of the system and manages the CPU, memory, and peripheral devices. The kernel is the "lowest" level of the OS.

- **Daemons:** These are background services (printing, sound, scheduling, etc) that either start up during boot, or after you log into the desktop.

- **The Shell:** You've probably heard mention of the Linux command line. This is the shell – a command process that allows you to control the computer via commands typed into a text interface. This is what, at one time, scared people away from Linux the most (assuming they had to learn a seemingly archaic command line structure to make Linux work). This is no longer the case. With modern desktop Linux, there is no need to ever touch the command line.

- **Graphical Server:** This is the sub-system that displays the graphics on your monitor. It is commonly referred to as the X server or just "X".

- **Desktop Environment:** This is the piece of the puzzle that the users actually interact with. There are many desktop environments to choose from (Unity, GNOME, Cinnamon, Enlightenment, KDE, XFCE, etc). Each desktop environment includes built-in applications (such as file managers, configuration tools, web browsers, games, etc).

- **Applications:** Desktop environments do not offer the full array of apps. Just like Windows and Mac, Linux offers thousands upon thousands of high-quality software titles that can be easily found and installed. Most modern Linux distributions (more on this in a moment) include App Store-like tools that centralize and simplify application installation. For example: Ubuntu Linux has the Ubuntu Software Center (Figure 1) which allows you to quickly search among the thousands of apps and install them from one centralized location.

Linux is also distributed under an open source license. Open source follows the following key philosophies:

- The freedom to run the program, for any purpose.

- The freedom to study how the program works, and change it to make it do what you wish.

- The freedom to redistribute copies so you can help your neighbor.
- The freedom to distribute copies of your modified versions to others.

What is a "distribution?"

Linux has a number of different versions to suit nearly any type of user. From new users to hard-core users, you'll find a "flavor" of Linux to match your needs. These versions are called distributions (or, in the short form, "distros.") Nearly every distribution of Linux can be downloaded for free, burned onto disk (or USB thumb drive), and installed (on as many machines as you like).

The most popular Linux distributions are:

- Ubuntu Linux
- Linux Mint
- Arch Linux
- Deepin
- Fedora
- Debian
- openSUSE.

And don't think the server has been left behind. For this arena, you can turn to:

- Red Hat Enterprise Linux
- Ubuntu Server
- CentOS
- SUSE Enterprise Linux.

Some of the above server distributions are free (such as Ubuntu Server and CentOS) and some have an associated price (such as Red Hat Enterprise Linux and SUSE Enterprise Linux). Those with an associated price also include support.

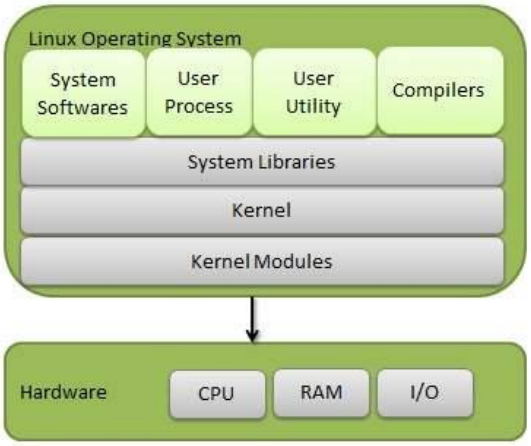
Components of Linux System

Linux Operating System has primarily three components

- **Kernel** – Kernel is the core part of Linux. It is responsible for all major activities of this operating system. It consists of various modules and it interacts directly with the underlying hardware. Kernel provides the required abstraction to hide low level hardware details to system or application programs.

- **System Library** – System libraries are special functions or programs using which application programs or system utilities accesses Kernel's features. These libraries implement most of the functionalities of the operating system and do not requires kernel module's code access rights.

- **System Utility** – System Utility programs are responsible to do specialized, individual level tasks.



Kernel Mode vs User Mode

Kernel component code executes in a special privileged mode called **kernel mode** with full access to all resources of the computer. This code represents a single process, executes in single address space and do not require any context switch and hence is very efficient and fast. Kernel runs each processes and provides system services to processes, provides protected access to hardware to processes.

Support code which is not required to run in kernel mode is in System Library. User programs and other system programs works in **User Mode** which has no access to system hardware and kernel code. User programs/ utilities use System libraries to access Kernel functions to get system's low level tasks.

Basic Features

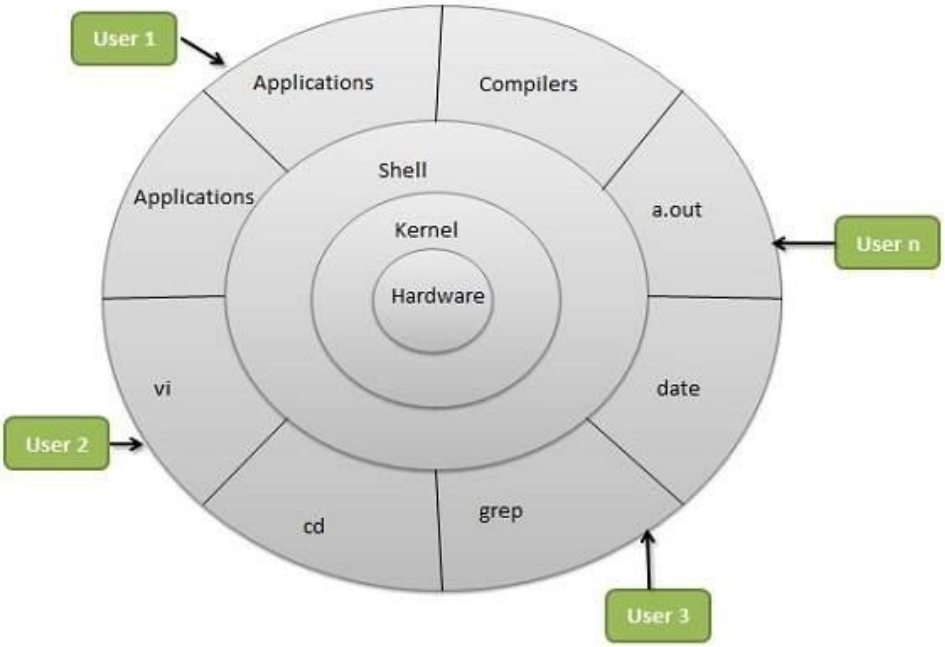
Following are some of the important features of Linux Operating System.

- **Portable** – Portability means software can works on different types of hardware in same way. Linux kernel and application programs supports their installation on any kind of hardware platform.

- **Open Source** − Linux source code is freely available and it is community based development project. Multiple teams work in collaboration to enhance the capability of Linux operating system and it is continuously evolving.
- **Multi-User** − Linux is a multiuser system means multiple users can access system resources like memory/ ram/ application programs at same time.
- **Multiprogramming** − Linux is a multiprogramming system means multiple applications can run at same time.
- **Hierarchical File System** − Linux provides a standard file structure in which system files/ user files are arranged.
- **Shell** − Linux provides a special interpreter program which can be used to execute commands of the operating system. It can be used to do various types of operations, call application programs. etc.
- **Security** − Linux provides user security using authentication features like password protection/ controlled access to specific files/ encryption of data.

Architecture

The following illustration shows the architecture of a Linux system −



The architecture of a Linux System consists of the following layers −

- **Hardware layer** − Hardware consists of all peripheral devices (RAM/ HDD/ CPU etc).
- **Kernel** − It is the core component of Operating System, interacts directly with hardware, provides low level services to upper layer components.
- **Shell** − An interface to kernel, hiding complexity of kernel's functions from users. The shell takes commands from the user and executes kernel's functions.
- **Utilities** − Utility programs that provide the user most of the functionalities of an operating systems.

3. Explain design goals of Linux

- UNIX was always an interactive system designed to handle multiple processes and multiple users at the same time. It was designed by programmers, for programmers, to use in an environment in which the majority of the users are relatively sophisticated and are engaged in (often quite complex) software development projects. In many cases, a large number of programmers are actively cooperating to produce a single system, so UNIX has extensive facilities to allow people to work together and share information in controlled ways.

- Linux is a multi-user, multitasking system with a full set of UNIX-compatible tools..
- Its file system adheres to traditional UNIX semantics, and it fully implements the standard UNIX networking model.
- Main design goals are speed, efficiency, and standardization.
- Linux is designed to be compliant with the relevant POSIX documents; at least two Linux distributions have achieved official POSIX certification.
- The Linux programming interface adheres to the SVR4 UNIX semantics, rather than to BSD behavior.

4. Give the milestones of the original UNIX

- **As Linux turns 20, we look back on key moments for the OS that started as a school project and became a major force in technology.**
- Twenty years ago, the tech landscape looked very different from that of today. Cell phones were a luxury of the rich, and the devices themselves were pretty dumb.

Microsoft ruled the desktop landscape barely challenged by competition from IBM and Apple. The Internet was just a gleam in Al Gore's eye (*kidding!*). And a young University of Helsinki student named Linus Torvalds started work on an operating system that came to be known as Linux.

- Linux has come a long way since the early tinkerings of Torvalds in 1991. The OS has proliferated around the world and into every kind of computer, from smartphones to supercomputers. Here are 11 major milestones in the 20-year history of Linux.
- **April 1991:** From his dorm room at the University of Helsinki, college student Linus Torvalds begins working on his own operating system kernel, mostly just to see if he could do it. As he was doing his early development in a Unix clone called Minix, he posted a noteto a Minix newsgroup that said, "I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones." Torvalds was wrong in his assessment of his creation's potential.
- **May 1992:** Just over a year after Torvalds began working on his pet project, the first comprehensive distribution of Linux, Softlanding Linux System, shipped to users. SLS stood out for its incorporation of TCP/IP and X Windows.
- **July 1993:** Slackware Linux, developed by Patrick Volkerding, launches as the first commercial Linux distribution. It is currently the oldest Linux distribution still under development.
- **March 1994:** Linus Torvalds releases Linux 1.0, consisting of 176,250 lines of code.
- **April 1995:** Linux gets its own trade conference, Linux Expo, created by Donnie Barnes at North Carolina State University. Barnes went on to work for Red Hat, which later took over the expo.
- **November 1998:** In the midst of a federal antitrust lawsuit, Microsoft lawyers present a box of Red Hat Linux as evidence that Windows did not represent a monopoly on the OS market.
- **November 1999:** VA Systems launches SourceForge, which becomes a leading repository of open source projects for Linux and other platforms.
- **October 2004:** Canonical releases Ubuntu 4.1, aka "Warty Warthog," which raised the bar for community-developed Linux distributions with a six-month release cycle and a focus on user experience.
- **January 2007:** Several leading mobile technology companies, including Motorola, NEC, Samsung, NTT DoCoMo, Panasonic, and Vodafone form the LiMo Foundation to collaborate on Linux-based smartphones. This represents a major shift in the direction of Linux devices, and presages the arrival of Google Android.
- **November 2007:** The Open Handset Alliance, which includes Google, Intel, Sony, HTC, Motorola, and 78 other companies, announces its presence with a preview of Android. One week later, the OHA released a SDK to developers.

- **October 2008:** The first commercial Android phone, the T-Mobile G1, ships to consumers, marking the emergence of Linux onto mainstream consumer computing devices. On mobile phones, Android has gone on to compete mightily with Apple's iOS, putting Linux squarely in the forefront of today's hottest platform war.

5. Explain Interfaces to Linux.

- A Linux system can be regarded as a kind of pyramid, as illustrated in Fig. At the bottom is the hardware, consisting of the CPU, memory, disks, a monitor and keyboard, and other devices. Running on the bare hardware is the operating system. Its function is to control the hardware and provide a system call interface to all the programs. These system calls allow user programs to create and manage processes, files, and other resources.

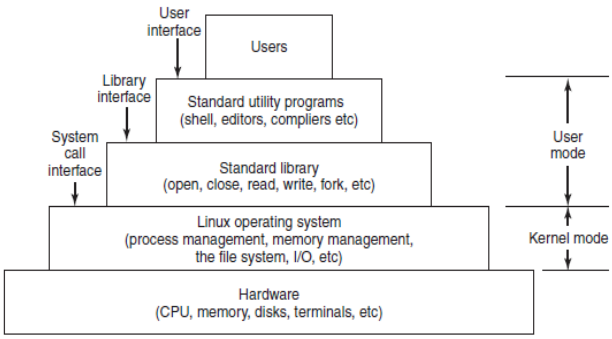


Figure 10-1. The layers in a Linux system.

A Linux operating system can be divided in to the following layers:

1) Hardware: This is the bottom most layer of a Linux system. It consists of monitor, CPU, memory, disks, terminals, keyboards, and other devices.

2) Linux operating system: Linux operating system runs on the hardware. It controls the hardware and manages memory, processes, file systems, and Input/Output. It also provides a system call interface for the programs.

3) System library: This is the standard library for calling specific procedures. It provides a library interface for the system calls. It has various library procedures like read, write, fork, etc.

4) Utility programs: A Linux system has several standard utility programs like compilers, shell, editors, file manipulation utilities, text processors, and other programs which can be called by the user. It provides a user interface for these programs.

5) Users: This is the topmost layer in the Linux operating system. It consists of the users of the Linux operating system.

6. What is mean by shell in LINUX? What is its use?

- Computer understand the language of 0's and 1's called binary language. In early days of computing, instruction are provided using binary language, which is difficult for all of us, to read and write. So in Os there is special program called Shell. Shell accepts your instruction or commands in English (mostly) and if its a valid command, it is pass to kernel.
- Shell is a user program or it's environment provided for user interaction. Shell is an command language interpreter that executes commands read from the standard input device (keyboard) or from a file.
- Shell is not part of system kernel, but uses the system kernel to execute programs, create files etc.

Several shell available with Linux including:

Shell Name	Developed by	Where	Remark
BASH (Bourne-Again Shell)	Brian Fox and Chet Ramey	Free Software Foundation	Most common shell in Linux. It's Freeware shell.
CSH (C SHell)	Bill Joy	University of California (For BSD)	The C shell's syntax and usage are very similar to the C programming language.
KSH (Korn SHell)	David Korn	AT & T Bell Labs	--
TCSH	See the man page. Type $ man tcsh	--	TCSH is an enhanced but completely compatible version of the Berkeley UNIX C shell (CSH).

- The **Bourne shell** (sh) is a **shell**, or command-line interpreter, for computer operating systems. The **Bourne shell** was the default **shell** for Unix Version 7.
- **Bash** is a Unix **shell** and command language written by Brian Fox for the GNU Project as a free software replacement for the Bourne **shell**. First released in 1989, it has been distributed widely as it is a default **shell** on the major Linux distributions and OS X.

- **C shell** is the UNIX **shell** (command execution program, often called a command interpreter) created by Bill Joy at the University of California at Berkeley as an alternative to UNIX's original **shell**, the Bourne **shell** . These two UNIX **shells**, along with the Korn **shell** , are the three most commonly used **shells**.

- The **Korn shell** is the UNIX **shell** (command execution program, often called a command interpreter) that was developed by David **Korn** of Bell Labs as a comprehensive combined version of other major UNIX **shells**.

- Tcsh is an enhanced, but completely compatible version of the Berkeley UNIX C shell (csh). It is a command language interpreter usable both as an interactive login shell and a shell script command processor. It includes a command-line editor, programmable word completion, spelling correction, a history mechanism, job control and a C-like syntax.

7. Give the list of Linux Utility Programs.

- The command-line (shell) user interface to Linux consists of a large number of standard utility programs. Roughly speaking, these programs can be divided into six categories, as follows:

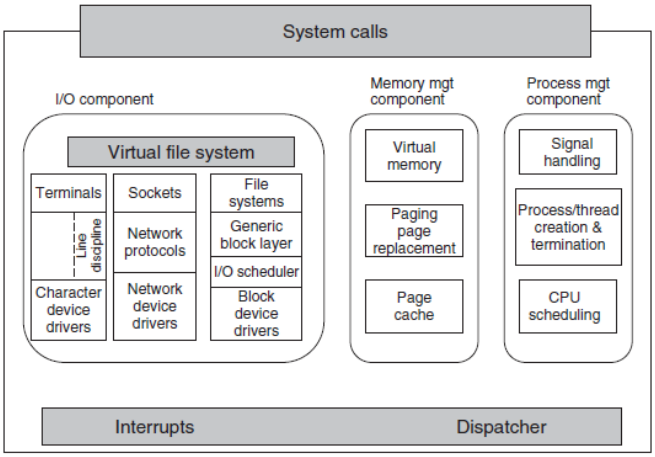
1. File and directory manipulation commands.
2. Filters.
3. Program development tools, such as editors and compilers.
4. Text processing.
5. System administration.
6. Miscellaneous.

POSIX (Portable Operating System Interface) is a set of standard operating system interfaces based on the Unix operating system.

Program	Typical use
cat	Concatenate multiple files to standard output
chmod	Change file protection mode
cp	Copy one or more files
cut	Cut columns of text from a file
grep	Search a file for some pattern
head	Extract the first lines of a file
ls	List directory
make	Compile files to build a binary
mkdir	Make a directory
od	Octal dump a file
paste	Paste columns of text into a file
pr	Format a file for printing
ps	List running processes
rm	Remove one or more files
rmdir	Remove a directory
sort	Sort a file of lines alphabetically
tail	Extract the last lines of a file
tr	Translate between character sets

8. Describe Linux kernel with appropriate diagram.

- The **Linux kernel** is a Unix-like computer operating system kernel. The Linux operating system is based on it and deployed on both traditional computer systems such as personal computers and servers, usually in the form of Linux distributions,[9] and on various embedded devices such as routers, wireless access points, PBXes, set-top boxes, FTA receivers, smart TVs, PVRs and NAS appliances. The Android operating system for tablet computers, smartphones and smartwatches is also based atop the Linux kernel.
- The Linux kernel API, the application programming interface (API) through which user programs interact with the kernel, is meant to be very stable and to not break userspace programs (some programs, such as those with GUIs, rely on other APIs as well). As part of the kernel's functionality, device drivers control the hardware; "mainlined" device drivers are also meant to be very stable. However, the interface between the kernel and loadable kernel modules (LKMs), unlike in many other kernels and operating systems, is not meant to be very stable by design.



- The kernel sits directly on the hardware and enables interactions with I/O devices and the memory management unit and controls CPU access to them. At the lowest level, as shown in Fig. it contains interrupt handlers, which are the primary way for interacting with devices, and the low-level dispatching mechanism. This dispatching occurs when an interrupt happens. The low-level code here stops the running process, saves its state in the kernel process structures, and starts the appropriate driver. Process dispatching also happens when the kernel completes some operations and it is time to start up a user process again. The dispatching code is in assembler and is quite distinct from scheduling.

- Next, we divide the various kernel subsystems into three main components. The I/O component in Fig. 10-3 contains all kernel pieces responsible for interacting with devices and performing network and storage I/O operations. At the highest level, the I/O operations are all integrated under a **VFS (Virtual File System)** layer. That is, at the top level, performing a read operation on a file, whether it is in memory or on disk, is the same as performing a read operation to retrieve a character from a terminal input. At the lowest level, all I/O operations pass through some device driver. All Linux drivers are classified as character-device drivers or block-device drivers, the main difference being that seeks and random accesses are allowed on block devices and not on character devices.

9. Explain process in Linux and PID, UID, GID in Linux.

- Processes carry out tasks within the operating system. A program is a set of machine code instructions and data stored in an executable image on disk and is, as such, a passive entity; a process can be thought of as a computer program in action.

- During the lifetime of a process it will use many system resources. It will use the CPUs in the system to run its instructions and the system's physical memory to hold it and its

data. It will open and use files within the file systems and may directly or indirectly use the physical devices in the system. Linux must keep track of the process itself and of the system resources that it has so that it can manage it and the other processes in the system fairly. It would not be fair to the other processes in the system if one process monopolized most of the system's physical memory or its CPUs.

- The most precious resource in the system is the CPU, usually there is only one. Linux is a multiprocessing operating system, its objective is to have a process running on each CPU in the system at all times, to maximize CPU utilization. If there are more processes than CPUs (and there usually are), the rest of the processes must wait before a CPU becomes free until they can be run.

- Processes are created in Linux in an especially simple manner. The fork system call creates an exact copy of the original process. The forking process is called the **parent process. The new process is called the child process. The parent and** child each have their own, private memory images. If the parent subsequently changes any of its variables, the changes are not visible to the child, and vice versa.

- Process Identifier is when each process has a unique identifier associated with it known as process id.

- User and Group Identifiers (UID and GID) are the identifiers associated with a processes of the user and group.

- The new process are created by cloning old process or current process. A new task is created by a system call i.e fork or clone. The forking process is called parent process and the new process is called as child process.

Signal	Cause
SIGABRT	Sent to abort a process and force a core dump
SIGALRM	The alarm clock has gone off
SIGFPE	A floating-point error has occurred (e.g., division by 0)
SIGHUP	The phone line the process was using has been hung up
SIGILL	The user has hit the DEL key to interrupt the process
SIGQUIT	The user has hit the key requesting a core dump
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written to a pipe which has no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purposes

Linux Processes

- **State:-** As a process executes it changes *state* according to its circumstances. Linux processes have the following states:
- **Running:-** The process is either running (it is the current process in the system) or it is ready to run (it is waiting to be assigned to one of the system's CPUs).
- **Waiting:-** The process is waiting for an event or for a resource. Linux differentiates between two types of waiting process; *interruptible* and *uninterruptible*. Interruptible waiting processes can be interrupted by signals whereas uninterruptible waiting processes are waiting directly on hardware conditions and cannot be interrupted under any circumstances.
- **Stopped:-** The process has been stopped, usually by receiving a signal. A process that is being debugged can be in a stopped state.
- **Zombie:-** This is a halted process which, for some reason, still has a data structure in the task vector. It is what it sounds like, a dead process.
- **Scheduling Information:-** The scheduler needs this information in order to fairly decide which process in the system most deserves to run,
- **Identifiers:-** Every process in the system has a process identifier. The process identifier is not an index into the task vector, it is simply a number. Each process also has User and group identifiers, these are used to control this processes access to the files and devices in the system,
- **Inter-Process Communication:-** Linux supports the classic Unix IPC mechanisms of signals, pipes and semaphores and also the System V IPC mechanisms of shared memory, semaphores and message queues.
- **Links:-** In a Linux system no process is independent of any other process. Every process in the system, except the initial process has a parent process. New processes are not created, they are copied, or rather *cloned* from previous processes
- **Times and Timers:-** The kernel keeps track of a processes creation time as well as the CPU time that it consumes during its lifetime. Each clock tick, the kernel updates the amount of time that the current process has spent in system and in user mode. Linux also supports process specific *interval* timers, processes can use system calls to set up timers to send signals to themselves when the timers expire. These timers can be single-shot or periodic timers.

- **Virtual memory:-** Most processes have some virtual memory (kernel threads and daemons do not) and the Linux kernel must track how that virtual memory is mapped onto the system's physical memory.
- **Processor Specific Context:-** A process could be thought of as the sum total of the system's current state. Whenever a process is running it is using the processor's registers, stacks and so on.
- **File system:-** Processes can open and close files as they wish and the processes contains pointers to descriptors for each open file as well as pointers to two VFS. Each VFS uniquely describes a file or directory within a file system and also provides a uniform interface to the underlying file systems.

10. What are Process Management System Calls in Linux?

- Processes are the most fundamental abstraction in a Linux system, after files. As object code in execution - active, alive, running programs - processes are more than just assembly language; they consist of data, resources, state, and a virtualized computer.
- Linux took an interesting path, one seldom traveled, and separated the act of reating a new process from the act of loading a new binary image. Although the two tasks are performed in tandem most of the time, the division has allowed a great deal of freedom for experimentation and evolution for each of the tasks. This road less traveled has survived to this day, and while most operating systems offer a single system call to start up a new program, Linux requires two: a fork and an exec.

Creation and termination

Syscall	Description
CLONE	Create a child process
FORK	Create a child process
VFORK	Create a child process and block parent
EXECVE	Execute program
EXECVEAT	Execute program relative to a directory file descriptor
EXIT	Terminate the calling process
EXIT_GROUP	Terminate all threads in a process
WAIT4	Wait for process to change state
WAITID	Wait for process to change state

Pocess id

Syscall	Description
GETPID	Get process ID
GETPPID	Get parent process ID
GETTID	Get thread ID

Session id

Syscall	Description
SETSID	Set session ID
GETSID	Get session ID

Process group id

Syscall	Description
SETPGID	Set process group ID
GETPGID	Get process group ID
GETPGRP	Get the process group ID of the calling process

Users and groups

Syscall	Description
SETUID	Set real user ID
GETUID	Get real user ID
SETGID	Set real group ID
GETGID	Get real group ID
SETRESUID	Set real, effective and saved user IDs
GETRESUID	Get real, effective and saved user IDs
SETRESGID	Set real, effective and saved group IDs
GETRESGID	Get real, effective and saved group IDs
SETREUID	Set real and/or effective user ID
SETREGID	Set real and/or effective group ID
SETFSUID	Set user ID used for file system checks
SETFSGID	Set group ID used for file system checks
GETEUID	Get effective user ID

GETEGID	Get effective group ID
SETGROUPS	Set list of supplementary group IDs
GETGROUPS	Get list of supplementary group IDs

11.What is mean by user space thread and kernel space thread

- The information in the process descriptor falls into a number of broad categories that can be roughly described as follows:

Types of Thread

- Threads are implemented in following two ways −
- **User Level Threads** − User managed threads.
- **Kernel Level Threads** − Operating System managed threads acting on kernel, an operating system core.

User-Space Threads

- User-space avoids the kernel and manages the tables itself. Often this is called "cooperative multitasking" where the task defines a set of routines that get "switched to" by manipulating the stack pointer. Typically each thread "gives-up" the CPU by calling an explicit switch, sending a signal or doing an operation that involves the switcher. Also, a timer signal can force switches. User threads typically can switch faster than kernel threads [however, Linux kernel threads' switching is actually pretty close in performance].
- Disadvantages. User-space threads have a problem that a single thread can monopolize the timeslice thus starving the other threads within the task. Also, it has no way of taking advantage of SMPs (Symmetric MultiProcessor systems, e.g. dual-/quad-Pentiums). Lastly, when a thread becomes I/O blocked, all other threads within the task lose the timeslice as well.
- Solutions/work arounds. Some user-thread libraries have addressed these problems with several work-arounds. First timeslice monopolization can be controlled with an external monitor that uses its own clock tick. Second, some SMPs can support user-space multithreading by firing up tasks on specified CPUs then starting the threads from there [this form of SMP threading seems tenuous, at best]. Third, some libraries solve the I/O blocking problem with special wrappers over system calls, or the task can be written for nonblocking I/O.

Kernel-Space Threads

- Kernel-space threads often are implemented in the kernel using several tables (each task gets a table of threads). In this case, the kernel schedules each thread within the timeslice of each process. There is a little more overhead with mode switching from user->kernel-> user and loading of larger contexts, but initial performance measures indicate a negligible increase in time.
- Advantages. Since the clocktick will determine the switching times, a task is less likely to hog the timeslice from the other threads within the task. Also I/O blocking is not a problem. Lastly, if properly coded, the process automatically can take advantage of SMPs and will run incrementally faster with each added CPU.

1. **Scheduling parameters.**

Process priority, amount of CPU time consumed recently, amount of time spent sleeping recently. Together, these are used to determine which process to run next.

2. **Memory image.**

Pointers to the text, data, and stack segments, or page tables. If the text segment is shared, the text pointer points to the shared text table. When the process is not in memory, information about how to find its parts on disk is here too.

3. **Signals.**

Masks showing which signals are being ignored, which are being caught, which are being temporarily blocked, and which are in the process of being delivered.

4. **Machine registers.**

When a trap to the kernel occurs, the machine registers (including the floating-point ones, if used) are saved here.

5. **System call state.**

Information about the current system call, including the parameters, and results.

6. **File descriptor table.**

When a system call involving a file descriptor is invoked, the file descriptor is used as an index into this table to locate the in-core data structure (i-node) corresponding to this file.

7. **Accounting.**

Pointer to a table that keeps track of the user and system CPU time used by the process. Some systems also maintain limits here on the amount of CPU time a process may use, the maximum size of its stack, the number of page frames it may consume, and other items.

8. **Kernel stack.**

A fixed stack for use by the kernel part of the process.

9. **Miscellaneous.**

Current process state, event being waited for, if any, time until alarm clock goes off, PID, PID of the parent process, and user and group identification.

Difference between Process and Thread

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Threads in Linux

Flag	Meaning when set	Meaning when cleared
CLONE_VM	Create a new thread	Create a new process
CLONE_FS	Share umask, root, and working dirs	Do not share them
CLONE_FILES	Share the file descriptors	Copy the file descriptors
CLONE_SIGHAND	Share the signal handler table	Copy the table
CLONE_PARENT	New thread has same parent as the caller	New thread's parent is caller

Scheduling in Linux

- Linux threads are kernel threads, so scheduling is based on threads, not processes.
- Linux distinguishes three classes of threads for scheduling purposes:

1. Real-time FIFO.

2. Real-time round robin.

3. Timesharing.

- Real-time FIFO threads are the highest priority and are not preemptable except by a newly readied real-time FIFO thread with even higher priority. Real-time round robin threads are the same as real-time FIFO threads except that they have e time quanta associated with them, and are preemptable by the clock. If multiple real-time round-robin threads are ready, each one is run for its quantum, after which it goes to the end of the list of real-time round-robin threads. Neither of these classes is actually real time in any sense. Deadlines cannot be specified and guarantees are not given. These classes are simply higher priority than threads in the standard timesharing class.
- The Completely Fair Scheduler (CFS) is a process scheduler which was merged into the Linux kernel and is the default scheduler. It handles CPU resource allocation for executing processes, and aims to maximize overall CPU utilization while also maximizing interactive performance.

12. Write a short note on Synchronization in Linux.

- You could think of the kernel as a server that answers requests; these requests can come either from a process running on a CPU or an external device issuing an interrupt request.

- We make this analogy to underscore that parts of the kernel are not run serially, but in an interleaved way. Thus, they can give rise to race conditions, which must be controlled through proper synchronization techniques.

1. **Critical Region:** A critical section is a piece of code which should be executed under mutual exclusion. Suppose that, two threads are updating the same variable which is in parent process's address space. So the code area where both thread access/update the shared variable/resource is called as a Critical Region. It is essential to protect critical region to avoid collusion in code/system.

2. **Race Condition:** So developer has to protect Critical Region such that, at one time instance there is only one thread/process which is passing under that region(accessing shared resources). If Critical Region doesn't protected with the proper mechanism, then there are chances of Race Condition. Finally, a *race condition* is a flaw that occurs when the timing or ordering of events affects a program's correctness. by using appropriate synchronization mechanism or properly protecting Critical Region we can avoid/reduce the chance of this flaw.

3. **Deadlock:** This is the other flaw which can be generated by NOT using proper synchronization mechanism. It is a situation in which two thread/process sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function.

Linux kernel provide couple of synchronization mechanism.

1. **Atomic operation:** This is the very simple approach to avoid race condition or deadlock. Atomic operators are operations, like add and subtract, which perform in one clock cycle (uninterruptible operation). **A common use of the atomic integer operations is to implement counters which is updated by multiple threads.** The kernel provides two sets of interfaces for atomic operations, one that operates on integers and another that operates on individual bits. All atomic functions are inline functions.

2. **Semaphore:** This is another kind of synchronization mechanism which will be provided by the Linux kernel. When some process is trying to access semaphore which is not available, semaphore puts process on wait queue(FIFO) and puts task on sleep. That's why semaphore is known as a sleeping lock. After this processor is free to jump to other task which is not requiring this semaphore. As soon as semaphore get available, one of task from wait queue in invoked.

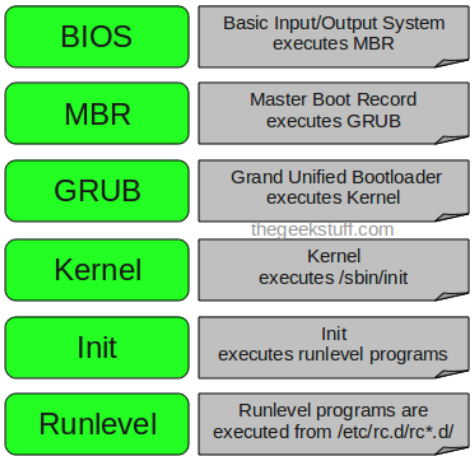
- There two flavors of semaphore is present.
- Basic semaphore

- Reader-Writter Semaphore

3. **Spin-lock:** This is special type of synchronization mechanism which is preferable to use in multi-processor(SMP) system. Basically its a busy-wait locking mechanism until the lock is available. In case of unavailability of lock, it keeps thread in light loop and keep checking the availability of lock. Spin-lock is not recommended to use in single processor system. If some procesq_1 has acquired a lock and other process_2 is trying to acquire lock, in this case process 2 will spins around and keep processor core busy until it acquires lock. process_2 will create a deadlock, it dosent allow any other process to execute because CPU core is busy in light loop by semaphore.

4. **Sequence Lock:** This is very useful synchronization mechanism to provide a lightweight and scalable lock for the scenario where many readers and a few writers are present. Sequence lock maintains a counter for sequence.

13.Explain the Booting process in Linux.



- 1.BIOS(Basic Input/Output System)
- 2.MBR(Master Boot Record)
- 3.LILO or GRUB
- LILO:-LInux LOader
- GRUB:-GRand Unified Bootloader
- 4.Kernel
- 5.init
- 6.Run Levels

1. **BIOS:**

i. When we power on BIOS performs a **P**ower-**O**n **S**elf-**T**est (**POST**) for all of the different hardware components in the system to make sure everything is working properly

ii. Also it checks for whether the computer is being started from an off position (cold boot) or from a restart (warm boot) is stored at this location.

iii. Retrieves information from **CMOS** (**C**omplementary **M**etal-**O**xide **S**emiconductor) a battery operated memory chip on the motherboard that stores time, date, and critical system information.

iv. Once BIOS sees everything is fine it will begin searching for an operating system Boot Sector on a valid master boot sector on all available drives like hard disks, CD-ROM drive etc.

v. Once BIOS finds a valid MBR it will give the instructions to boot and executes the first 512-byte boot sector that is the first sector ("Sector 0") of a partitioned data storage device such as hard disk or CD-ROM etc. .

2. **MBR**

I. Normally we use multi-level boot loader. Here MBR means I am referencing to DOS MBR.

II. After BIOS executes a valid DOS MBR, the DOS MBR will search for a valid primary partition marked as bootable on the hard disk.

III. If MBR finds a valid bootable primary partition then it executes the first 512-bytes of that partition which is second level MBR.

iv. In Linux we have two types of the above mentioned second level MBR known as LILO and GRUB.

3. LILO

i. LILO is a Linux boot loader which is too big to fit into single sector of 512-bytes.
ii. So it is divided into two parts :an installer and a runtime module.
iii. The installer module places the runtime module on MBR. The runtime module has the info about all operating systems installed.
iv. When the runtime module is executed it selects the operating system to load and transfers the control to kernel.
v. LILO does not understand filesystems and boot images to be loaded and treats them as raw disk offsets

4.GRUB

i. GRUB MBR consists of 446 bytes of primary bootloader code and 64 bytes of the partition table.

ii. GRUB locates all the operating systems installed and gives a GUI to select the operating system need to be loaded.

iii. Once user selects the operating system GRUB will pass control to the kernel of that operating system. see below what is the difference between LILO and GRUB

4.Kernel

i. Once GRUB or LILO transfers the control to Kernel,the Kernels does the following tasks

- Initializes devices and loads initrd module
- mounts root filesystem

5.Init

i. The kernel, once it is loaded, finds init in sbin(/sbin/init) and executes it.

ii. Hence the first process which is started in linux is init process.

iii. This init process reads /etc/inittab file and sets the path, starts swapping, checks the file systems, and so on.

iv. It runs all the boot scripts(/etc/rc.d/*,/etc/rc.boot/*)

v. starts the system on specified run level in the file /etc/inittab

6.Runlevel

i. There are 7 run levels in which the linux OS runs and different run levels serves for different purpose. The descriptions are given below.

- 0 – halt
- 1 – Single user mode
- 2 – Multiuser, without NFS (The same as 3, if you don't have networking)
- 3 – Full multiuser mode
- 4 – unused
- 5 – X11
- 6 – Reboot

ii. We can set in which runlevel we want to run our operating system by defining it on /etc/inittab file. Now as per our setting in /etc/inittab the Operating System the operating system boots up and finishes the bootup process.

Below are given some few important differences about LILO and GRUB

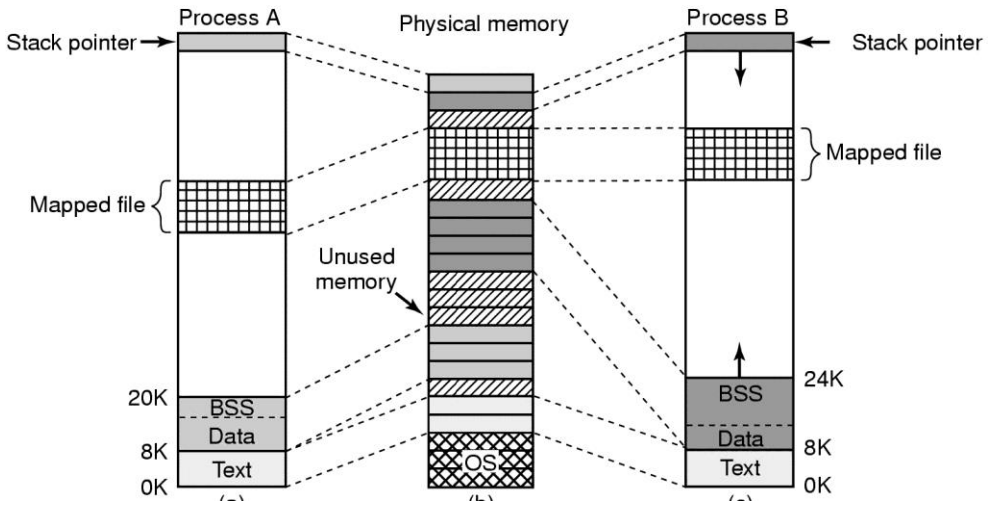
LILO	**GRUB**
LILO has no interactive command interface	GRUB has interactive command interface
LILO does not support booting from a network	GRUB does support booting from a network
If you change your LILO config file, you	GRUB automatically detects any change in

have to rewrite the LILO stage one boot loader to the MBR	config file and auto loads the OS
LILO supports only linux operating system	GRUB supports large number of OS

14. Write a short note on memory management in Linux.

- Memory management is one of the most complex activity done by Linux kernel. It has various concepts/issues associated with it.
- The memory management subsystem is one of the most important parts of the operating system. Since the early days of computing, there has been a need for more memory than exists physically in a system. Strategies have been developed to overcome this limitation and the most successful of these is virtual memory. Virtual memory makes the system appear to have more memory than it actually has by sharing it between competing processes as they need it.
- Virtual memory does more than just make your computer's memory go further. The memory management subsystem provides:
- **Large Address Spaces:-** The operating system makes the system appear as if it has a larger amount of memory than it actually has. The virtual memory can be many times larger than the physical memory in the system,
- **Protection:-** Each process in the system has its own virtual address space. These virtual address spaces are completely separate from each other and so a process running one application cannot affect another. Also, the hardware virtual memory mechanisms allow areas of memory to be protected against writing. This protects code and data from being overwritten by rogue applications.
- **Memory Mapping:-** Memory mapping is used to map image and data files into a processes address space. In memory mapping, the contents of a file are linked directly into the virtual address space of a process.
- **Fair Physical Memory Allocation:-** The memory management subsystem allows each running process in the system a fair share of the physical memory of the system,
- **Shared Virtual Memory:-** Although virtual memory allows processes to have separate (virtual) address spaces, there are times when you need processes to share memory. For example there could be several processes in the system running the bash command shell. Rather than have several copies of bash, one in each processes virtual address

space, it is better to have only one copy in physical memory and all of the processes running bash share it. Dynamic libraries are another common example of executing code shared between several processes. Shared memory can also be used as an Inter Process Communication (IPC) mechanism, with two or more processes exchanging information via memory common to all of them. Linux supports the Unix ™ System V shared memory IPC.



- The virtual and physical memory is divided into fixed length chunks known as pages.
- Each entry in the theoretical page table contains the following information:
- Valid flag. This indicates if this page table entry is valid,
- The physical page frame number that this entry is describing,
- Access control information. This describes how the page may be used. Can it be written to? Does it contain executable code?

Memory Management System Calls in Linux

System call	Description
s = brk(addr)	Change data segment size
a = mmap(addr, len, prot, flags, fd, offset)	Map a file in
s = unmap(addr, len)	Unmap a file

Physical Memory Management

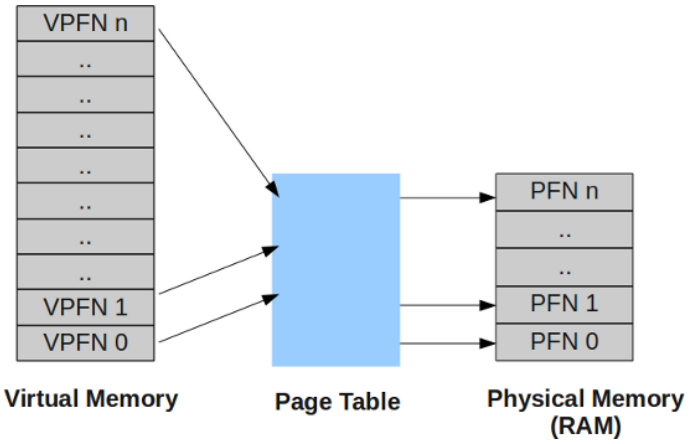
- Linux distinguishes between three memory zones:
- ZONE_DMA - pages that can be used for DMA operations.
- ZONE_NORMAL - normal, regularly mapped pages.
- ZONE_HIGHMEM - pages with high-memory addresses, which are not permanently mapped.

15.What is the use of paging in Linux?

- Only the required memory pages are moved to main memory from the swap device for execution. Process size does not matter. Gives the concept of the virtual memory.
- When a process starts in Unix, not all its memory pages are read in from the disk at once. Instead, the kernel loads into RAM only a few pages at a time. After the CPU digests these, the next page is requested. If it is not found in RAM, a **page fault** occurs, signaling the kernel to load the next few pages from disk into RAM. This is called **demand paging** and is a perfectly normal system activity in Unix.
- As there is much less physical memory than virtual memory the operating system must be careful that it does not use the physical memory inefficiently. One way to save physical memory is to only load virtual pages that are currently being used by the executing program. For example, a database program may be run to query a database. In this case not all of the database needs to be loaded into memory, just those data records that are being examined. If the database query is a search query then it does not make sense to load the code from the database program that deals with adding new records. This technique of only loading virtual pages into memory as they are accessed is known as demand paging.

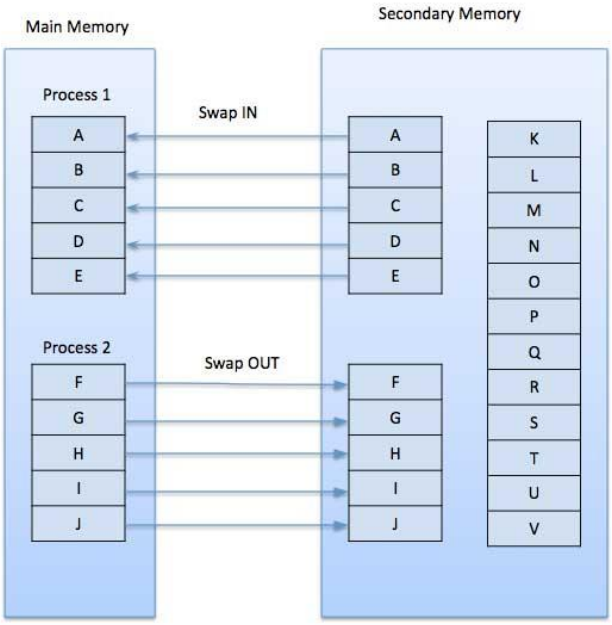
There are three kernel variables which control the paging operation

- *minfree* - the absolute minimum of free RAM needed. If free memory falls below this limit, the memory management system does its best to get back above it. It does so by **page stealing** from other, running processes, if practical.
- *desfree* - the amount of RAM the kernel wants to have free at all times. If free memory is less than *desfree*, the *pageout* syscall is called every clock cycle.
- *lotsfree* - the amount of memory necessary before the kernel stops calling *pageout*. Between *desfree* and *lotsfree*, *pageout* is called 4 times a second.



Demand Paging

A demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance. When a context switch occurs, the operating system does not copy any of the old program's pages out to the disk or any of the new program's pages into the main memory Instead, it just begins executing the new program after loading the first page and fetches that program's pages as they are referenced.



While executing a program, if the program references a page which is not available in the main memory because it was swapped out a little ago, the processor treats this invalid memory reference as a **page fault** and transfers control from the program to the operating system to demand the page back into the memory.

Advantages

Following are the advantages of Demand Paging −

- Large virtual memory.
- More efficient use of memory.
- There is no limit on degree of multiprogramming.

Disadvantages

- Number of tables and the amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.

Swapping

- If a process needs to bring a virtual page into physical memory and there are no free physical pages available, the operating system must make room for this page by discarding another page from physical memory.
- If the page to be discarded from physical memory came from an image or data file and has not been written to then the page does not need to be saved. Instead it can be discarded and if the process needs that page again it can be brought back into memory from the image or data file.

Shared Virtual Memory

- Virtual memory makes it easy for several processes to share memory. All memory access are made via page tables and each process has its own separate page table. For two processes sharing a physical page of memory, its physical page frame number must appear in a page table entry in both of their page tables.

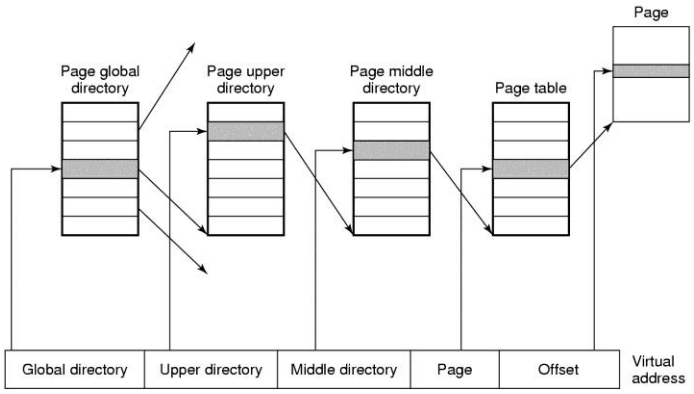
Physical and Virtual Addressing Modes

- It does not make much sense for the operating system itself to run in virtual memory. This would be a nightmare situation where the operating system must maintain page tables for itself. Most multi-purpose processors support the notion of a physical address mode as well as a virtual address mode. Physical addressing mode requires no page tables and the processor does not attempt to perform any address translations in this mode. The Linux kernel is linked to run in physical address space.

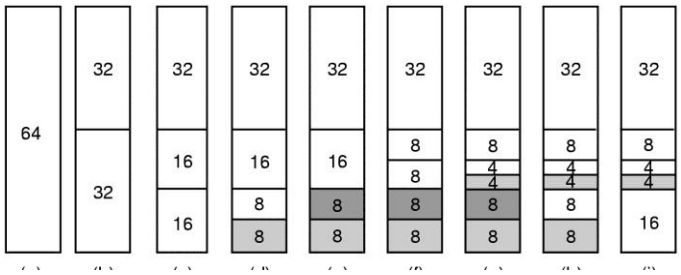
Access Control

- The page table entries also contain access control information. As the processor is already using the page table entry to map a processes virtual address to a physical one, it can easily use the access control information to check that the process is not accessing memory in a way that it should not.

Physical Memory Management



Memory Allocation Mechanisms



16.Explain the terms Directories, Special files, Links, sockets, Named pipes.

Directories:

Linux stores data and programs in **files**. These are organized in directories. In a simple way, a directory is just a file that contains other files (or directories).

The part of the hard disk where you are authorised to save data is called your **home directory**. Normally all the data you want will be saved in files and directories in your home directory. To find your home directory (if you need), type:

```
echo $HOME
```

The symbol **~** can also be used for your home directory.

There is a general directory called **/tmp** where every user can write files. But files in /tmp usually get removed (erased) when the system boots or periodically, so you should not store in /tmp data that you want to keep permanently.

Special Files:

Operating system like MS-DOS and UNIX have the following types of files −

Ordinary files

- These are the files that contain user information.
- These may have text, databases or executable program.
- The user can apply various operations on such files like add, modify, delete or even remove the entire file.

Directory files

- These files contain list of file names and other information related to these files.

Special files

- These files are also known as device files.
- These files represent physical device like disks, terminals, printers, networks, tape drive etc.

These files are of two types –

- **Character special files** − data is handled character by character as in case of terminals or printers.
- **Block special files** − data is handled in blocks as in the case of disks and tapes.

Links:

Two types of links

There are two types of links

- **symbolic links**: Refer to a symbolic path indicating the abstract location of another file
- **hard links** : Refer to the specific location of physical data.

Hard link vs. Soft link in Linux or UNIX

- Hard links cannot link directories.
- Cannot cross file system boundaries.

Soft or symbolic links are just like hard links. It allows to associate multiple filenames with a single file. However, symbolic links allows:

- To create links between directories.
- Can cross file system boundaries.

These links behave differently when the source of the link is moved or removed.

- Symbolic links are not updated.
- Hard links always refer to the source, even if moved or removed.

Sockets:

A **socket** is just a logical endpoint for communication. They exist on the transport layer. You can send and receive things on a **socket**, you can bind and listen to a**socket**. A **socket** is specific to a protocol, machine, and port, and is addressed as such in the header of a packet.

Named pipes:

In computing, a **named pipe** (also known as a FIFO for its behavior) is an extension to the traditional **pipe** concept on Unix and Unix-like systems, and is one of the methods of inter-process communication (IPC). The concept is also found in OS/2 and Microsoft Windows, although the semantics differ substantially.

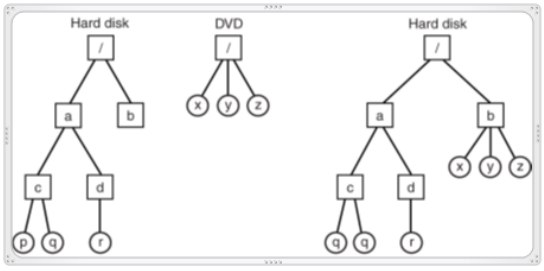
17. Explain any five File-System Calls in Linux.

- The file is the most basic and fundamental abstraction in Linux. Linux follows the everything-is-a-file philosophy. Consequently, much interaction transpires via filesystem system calls such as reading of and writing to files, even when the object in question is not what you would consider your everyday file.
- In order to be accessed, a file must first be opened. Files can be opened for reading, writing, or both. An open file is referenced via a unique descriptor, a mapping from the metadata associated with the open file back to the specific file itself. Inside the Linux kernel, this descriptor is handled by an integer (of the C type int) called the file descriptor, abbreviated fd. File descriptors are shared with user space, and are used

directly by user programs to access files. A large part of Linux system programming consists of opening, manipulating, closing, and otherwise using file descriptors.

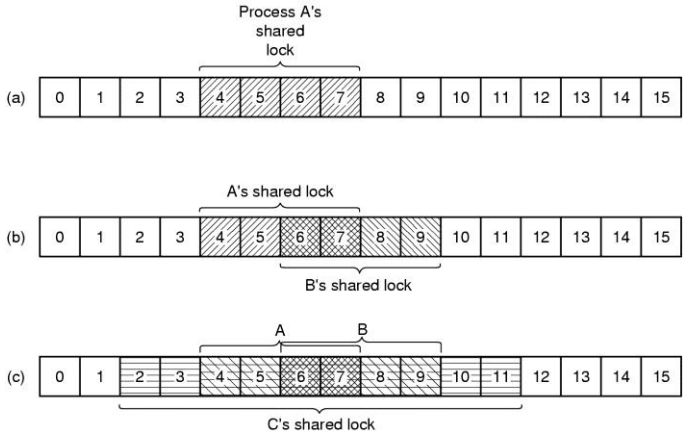
Directory	Contents
bin	Binary (executable) programs
dev	Special files for I/O devices
etc	Miscellaneous system files
lib	Libraries
usr	User directories

Function call	Description
s = cfsetospeed(&termios, speed)	Set the output speed
s = cfsetispeed(&termios, speed)	Set the input speed
s = cfgetospeed(&termios, speed)	Get the output speed
s = cfgetispeed(&termios, speed)	Get the input speed
s = tcsetattr(fd, opt, &termios)	Set the attributes
s = tcgetattr(fd, &termios)	Get the attributes



Separate file systems. (b) After mounting.

The Linux File System



(a) A file with one lock. (b) Addition of a second lock. (c) A third lock.

System call	Description
s = mkdir(path, mode)	Create a new directory
s = rmdir(path)	Remove a directory
s = link(oldpath, newpath)	Create a link to an existing file
s = unlink(path)	Unlink a file
s = chdir(path)	Change the working directory
dir = opendir(path)	Open a directory for reading
s = closedir(dir)	Close a directory
dirent = readdir(dir)	Read one directory entry
rewinddir(dir)	Rewind a directory so it can be reread

System call	Description
fd = creat(name, mode)	One way to create a new file
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information
s = fstat(fd, &buf)	Get a file's status information
s = pipe(&fd[0])	Create a pipe
s = fcntl(fd, cmd, ...)	File locking and other operations

18.Explain NFS (Network File System) calls in Linux.

The Network File System (NFS) is a way of mounting Linux discs/directories over a network. An NFS server can export one or more directories that can then be mounted on a remote Linux machine. Note, that if you need to mount a Linux filesystem on a Windows machine

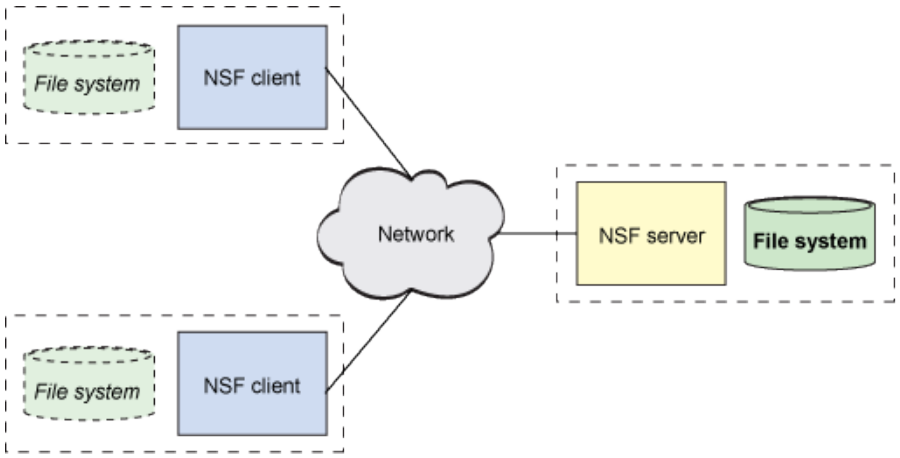
Why use the Network File System (NFS)?

The main use of NFS in the home context, is to share out data on a central server (-for example, your music collection) to all the PCs in the house. This way, you have a single copy of data (-hopefully, well backed up) accessible from a central location.

A *network file system* is a network abstraction over a file system that allows a remote client to access it over a network in a similar way to a local file system. Although not the first such system, NFS has grown and evolved into the most powerful and widely used network file system in UNIX. NFS permits sharing of a common file system among a multitude of users and provides the benefit of centralizing data to minimize needed storage.

The NFS architecture

NFS follows the client-server model of computing (see Figure 2). The server implements the shared file system and storage to which clients attach. The clients implement the user interface to the shared file system, mounted within the client's local file space. Figure 2. The client-server architecture of NFS



Within Linux, the virtual file system switch (VFS) provides the means to support multiple file systems concurrently on a host (such as International Organization for Standardization [ISO] 9660 on a CD-ROM and ext3fs on the local hard disk). The VFS determines which storage a request is intended for, then which file system must be used to satisfy the request. For this reason, NFS is a pluggable file system just like any other. The only difference with NFS is that input/output (I/O) requests may not be satisfied locally, instead having to traverse the network for completion.

Once a request is found to be destined for NFS, VFS passes it to the NFS instance within the kernel. NFS interprets the I/O request and translates it into an NFS procedure (OPEN, ACCESS, CREATE, READ, CLOSE, REMOVE, and so on).

19.How to implement NFS in Linux?

NFS (Network File System) is basically developed for sharing of files and folders between Linux/Unix systems by Sun Microsystems in 1980. It allows you to mount your local file systems over a network and remote hosts to interact with them as they

are mounted locally on the same system. With the help of NFS, we can set up file sharing between Unix to Linux system and Linux to Unix system.

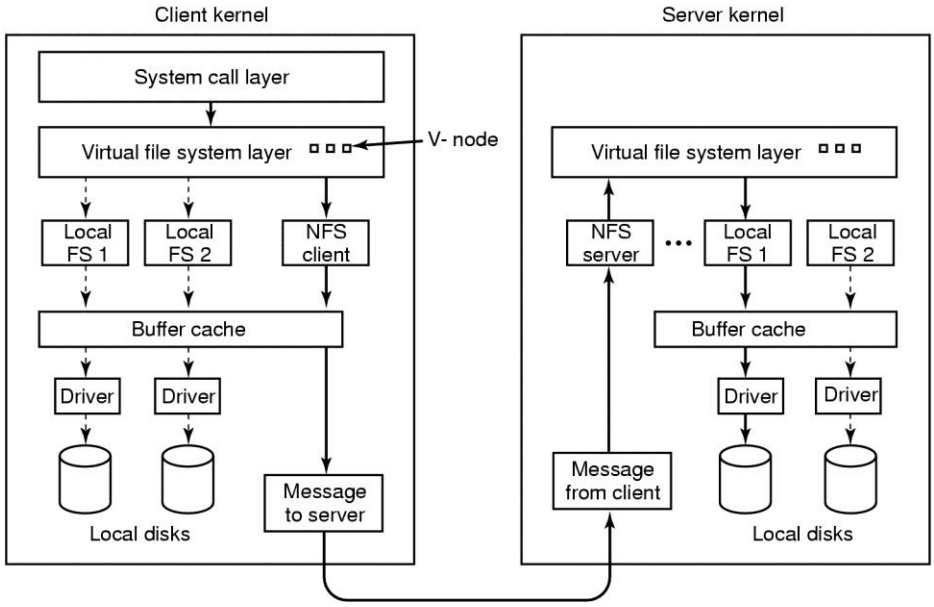
Benefits of NFS

1. NFS allows local access to remote files.
2. It uses standard client/server architecture for file sharing between all *nix based machines.
3. With NFS it is not necessary that both machines run on the same OS.
4. With the help of NFS we can configure centralized storage solutions.
5. Users get their data irrespective of physical location.
6. No manual refresh needed for new files.
7. Newer version of NFS also supports acl, pseudo root mounts.
8. Can be secured with Firewalls and Kerberos.

Important commands for NFS

Some more important commands for NFS.

1. showmount -e : Shows the available shares on your local machine
2. showmount -e <server-ip or hostname>: Lists the available shares at the remote server
3. showmount -d : Lists all the sub directories
4. exportfs -v : Displays a list of shares files and options on a server
5. exportfs -a : Exports all shares listed in /etc/exports, or given name
6. exportfs -u : Unexports all shares listed in /etc/exports, or given name
7. exportfs -r : Refresh the server's list after modifying /etc/exports



NFS Architecture and Main Components

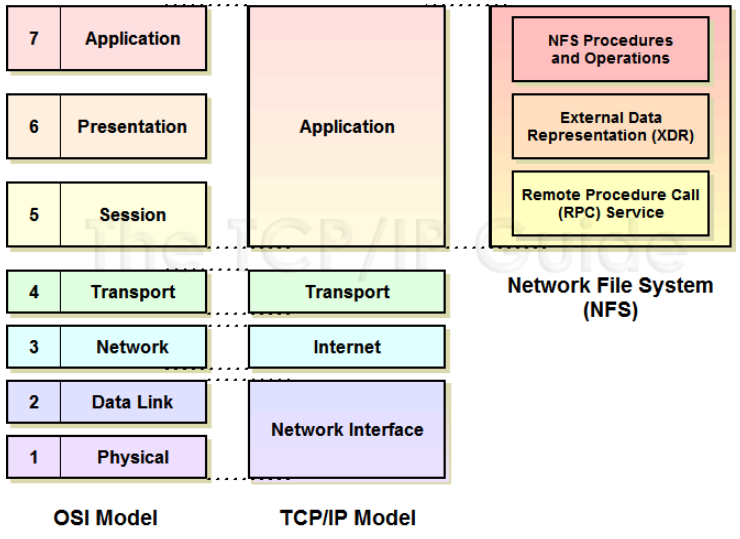
The operation of NFS is defined in the form of three main components that can be viewed as logically residing at each of the three OSI model layers corresponding to the TCP/IP application layer . These components are:

- **Remote Procedure Call (RPC):** RPC is a generic session layer service used to implement client/server internetworking functionality. It extends the notion of a program calling a local procedure on a particular host computer, to the calling of a procedure on a remote device across a network.

- **External Data Representation (XDR):** XDR is a descriptive language that allows data types to be defined in a consistent manner. XDR conceptually resides at the presentation layer; its universal representations allow data to be exchanged using NFS between computers that may use very different internal methods of storing data.

- **NFS Procedures and Operations:** The actual functionality of NFS is implemented in the form of procedures and operations that conceptually function at layer seven of the OSI model. These procedures specify particular tasks to be carried out on files over the network, using XDR to represent data and RPC to carry the commands across an internetwork.

These three key "subprotocols" if you will, comprise the bulk of the NFS protocol. Each is described in more detail in a separate topic of this section on NFS.



Architectural Components

NFS resides architecturally at the TCP/IP application layer. Even though in the TCP/IP model no clear distinction is made generally between the functions of layers five through seven of the OSI Reference Model, NFS's three subprotocols correspond well to those three layers as shown.

20.Explain File Security Model.

- The user community for a Linux system consists of some number of registered users, each of whom has a unique **UID** (**User ID**). A UID is an integer between 0 and 65,535. Files (but also processes and other resources) are marked with the UID of their owner. By default, the owner of a file is the person who created the file, although there is a way to change ownership.
- Users can be organized into groups, which are also numbered with 16-bit integers called **GIDs** (**Group IDs**). Assigning users to groups is done manually (by the system administrator) and consists of making entries in a system database telling which user is in which group. A user could be in one or more groups at the same time. For simplicity, we will not discuss this feature further.
- The basic security mechanism in Linux is simple. Each process carries the UID and GID of its owner. When a file is created, it gets the UID and GID of the creating process.
- The file also gets a set of permissions determined by the creating process. These permissions specify what access the owner, the other members of the owner's group, and the rest of the users have to the file. For each of these three categories, potential accesses are read, write, and execute, designated by the letters *r*,
- *w*, and *x*, respectively. The ability to execute a file makes sense only if that file is an executable binary program, of course.
- An attempt to execute a file that has execute permission but which is not executable (i.e., does not start with a valid header) will fail with an error. Since there are three categories of users and 3 bits per category, 9 bits are sufficient to represent the access rights. Some examples of these 9-bit numbers and their meanings are given in Fig

Binary	Symbolic	Allowed file accesses
111000000	rwx–––––––	Owner can read, write, and execute
111111000	rwxrwx–––	Owner and group can read, write, and execute
110100000	rw–r–––––	Owner can read and write; group can read
110100100	rw–r––r––	Owner can read and write; all others can read
111101101	rwxr–xr–x	Owner can do everything, rest can read and execute
000000000	–––––––––	Nobody has any access
000000111	––––––rwx	Only outsiders have access (strange, but legal)

Some example file protection modes.

The first two entries in Fig. 10-37 allow the owner and the owner's group full access, respectively. The next one allows the owner's group to read the file but not to change it, and prevents outsiders from any access. The fourth entry is common for a data file the owner wants to make public. Similarly, the fifth entry is the usual one for a publicly available program. The sixth entry denies all access to all users. This mode is sometimes used for dummy files used for mutual exclusion because an attempt to create such a file will fail if one already exists. Thus if multiple processes simultaneously attempt to create such a file as a lock, only one of them will succeed. The last example is strange indeed, since it gives the rest of the world more access than the owner. However, its existence follows from the protection rules.

21. Explain any five Security System Calls in Linux.

There are only a small number of system calls relating to security. The most important ones are listed in Fig. 10-38. The most heavily used security system call is chmod. It is used to change the protection mode. For example,

s = chmod("/usr/ast/newgame", 0755);

sets *newgame* to *rwxr–xr–x* so that everyone can run it (note that 0755 is an octal constant, which is convenient, since the protection bits come in groups of 3 bits). Only the owner of a file and the superuser can change its protection bits.

System call	Description
s = chmod(path, mode)	Change a file's protection mode
s = access(path, mode)	Check access using the real UID and GID
uid = getuid( )	Get the real UID
uid = geteuid( )	Get the effective UID
gid = getgid( )	Get the real GID
gid = getegid( )	Get the effective GID
s = chown(path, owner, group)	Change owner and group
s = setuid(uid)	Set the UID
s = setgid(gid)	Set the GID

System calls relating to security.

The access call tests to see if a particular access would be allowed using the real UID and GID. This system call is needed to avoid security breaches in programs that are SETUID and owned by the root. Such a program can do anything, and it is sometimes needed for the program to figure out if the user is allowed to perform a certain access. The program cannot just try it, because the access will always succeed. With the access call the program can find out if the access is allowed by the real UID and real GID. The next four system calls return the real and effective UIDs and GIDs. The last three are allowed only for the super user. They change a file's owner, and a process' UID and GID.