Assignment-1

1.Why VM(Virtual Machine) and where is it?

A Linux Virtual Machine (VM) is a software emulation of a physical computer that runs the Linux operating system. It's like having a computer within a computer, where the host system allocates resources such as processing power, memory, and storage to the VM.

It can be useful in software testing where we don't have access to the required hardware. This decreases our expense and effort considerably. Apart from that, a virtual machine enables us to run multiple Linux distros simultaneously, thereby decreasing the installation and maintenance overhead.

2. What is demand paging?

Demand paging can be described as a memory management technique that is used in operating systems to improve memory usage and system performance. Demand paging is a technique used in virtual memory systems where pages enter main memory only when requested or needed by the CPU.

In demand paging, the operating system loads only the necessary pages of a program into memory at runtime, instead of loading the entire program into memory at the start. A page fault occurred when the program needed to access a page that is not currently in memory. The operating system then loads the required pages from the disk into memory and updates the page tables accordingly. This process is transparent to the running program and it continues to run as if the page had always been in memory.

3.When the page fault occurs?

The term “page miss” or “page fault” refers to a situation where a referenced page is not found in the main memory.

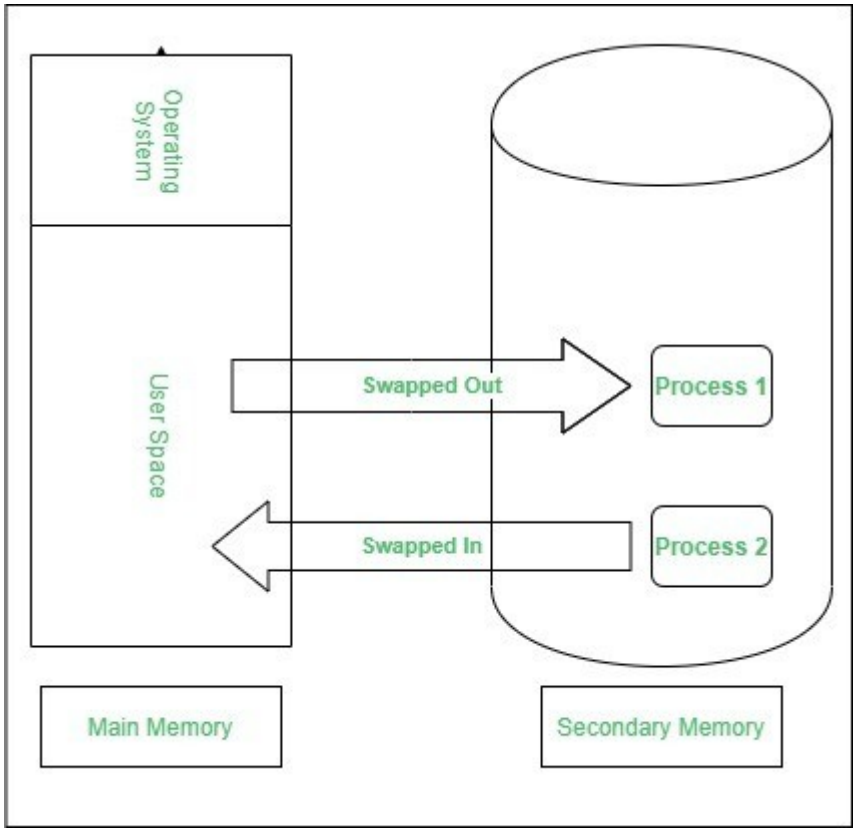
When a program tries to access a page, or fixed-size block of memory, that isn’t currently loaded in physical memory (RAM), an exception known as a page fault happens. Before enabling the program to access a page that is required, the operating system must bring it into memory from secondary storage (such a hard drive) in order to handle a page fault.

In modern operating systems, page faults are a common component of virtual memory management. By enabling programs to operate with more data than can fit in physical memory at once, they enable the efficient use of physical memory. The operating system is responsible for coordinating the transfer of data between physical memory and secondary storage as needed.

4.What is a trap,swap in, swap out ,page table?

To increase CPU utilization in multiprogramming, a memory management scheme known as swapping can be used. Swapping is the process of bringing a process into memory and then

temporarily copying it to the disc after it has run for a while. The purpose of swapping in an operating system is to access data on a hard disc and move it to RAM so that application programs can use it. It's important to remember that swapping is only used when data isn't available in RAM. Although the swapping process degrades system performance, it allows larger and multiple processes to run concurrently. Because of this, swapping is also known as memory compaction. The CPU scheduler determines which processes are swapped in and which are swapped out. Consider a multi programming environment that employs a priority-based scheduling algorithm. When a high-priority process enters the input queue, a low-priority process is swapped out so the high-priority process can be loaded and executed. When this process terminates, the low priority process is swapped back into memory to continue its execution. Below figure shows the swapping process in operating system:



Swapping has been subdivided into two concepts: swap-in and swap-out.

- Swap-out is a technique for moving a process from RAM to the hard disc.
- Swap-in is a method of transferring a program from a hard disc to main memory, or RAM.

5. Where the page table is available?

The page table is typically stored in main memory (RAM) as a data structure. It is used by the memory management unit (MMU) of the processor to translate virtual memory addresses used by the running processes into physical memory addresses where the actual data is stored in RAM.

Page tables are stored in memory

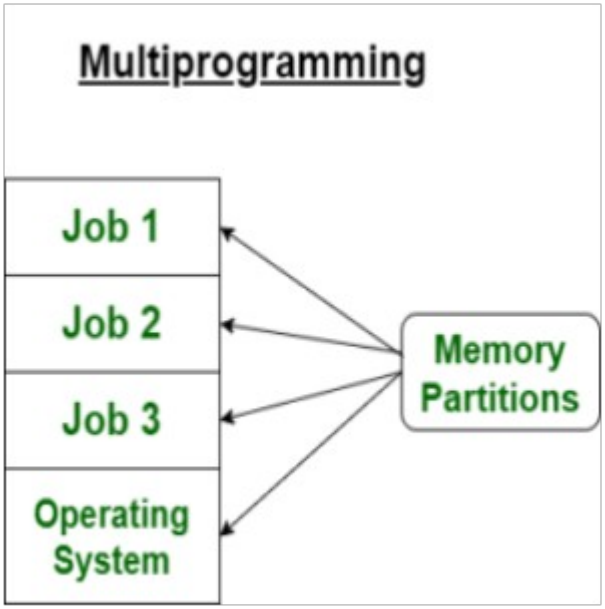
Because page tables are so big, we don't keep any special on-chip hardware in the MMU to store the page table of the currently-running process. Instead, we store the page table for each process in memory somewhere.

6. How many processes can be in the Ram in the multi programming in OS?

As the name suggests, more than one programs can be active at the same time. Before the concept of Multiprogramming, there were single tasking operating systems like MS DOS that used to allow only one program to be loaded at a time and run. These systems were not efficient as CPU was not used efficiently. For example, in a single tasking system if the current program waits for some input/output to finish, the CPU is not used. The idea of multiprogramming is to assign CPUs to other processes while the current process might not be finished. This has the below advantages.

1) User get the feeling that he/she can run multiple applications on a single CPU even if the CPU is running one process at a time.
2) CPU is utilized better

All modern operating systems like MS Windows, Linux, etc are multiprogramming operating systems,



Features of Multiprogramming

1. Need Single CPU for implementation.
2. Context switch between process.
3. Switching happens when current process undergoes waiting state.
4. CPU idle time is reduced.

5. High resource utilization.
6. High Performance.

7.What is a file, program, process, PCB?

file----A File is a collection of data stored in the secondary memory. So far data was entered into the programs through the keyboard. So Files are used for storing information that can be processed by the programs. Files are not only used for storing the data, programs are also stored in files.

Program--- Computer programming or coding is the composition of sequences of instructions, called programs, that computers can follow to perform tasks. It involves designing and implementing algorithms, step-by-step specifications of procedures, by writing code in one or more programming languages.

process---A process refers to the active execution of a program. It consists of several components, including data retrieved from files, user input, program instructions, etc.

PCB---- While creating a process, the operating system performs several operations. To identify the processes, it assigns a process identification number (PID) to each process. As the operating system supports multi-programming, it needs to keep track of all the processes. For this task, the process control block (PCB) is used to track the process's execution status. Each block of memory contains information about the process state, program counter, stack pointer, status of opened files, scheduling algorithms, etc.

All this information is required and must be saved when the process is switched from one state to another. When the process makes a transition from one state to another, the operating system must update information in the process's PCB. A process control block (PCB) contains information about the process, i.e. registers, quantum, priority, etc. The process table is an array of PCBs, that means logically contains a PCB for all of the current processes in the system.

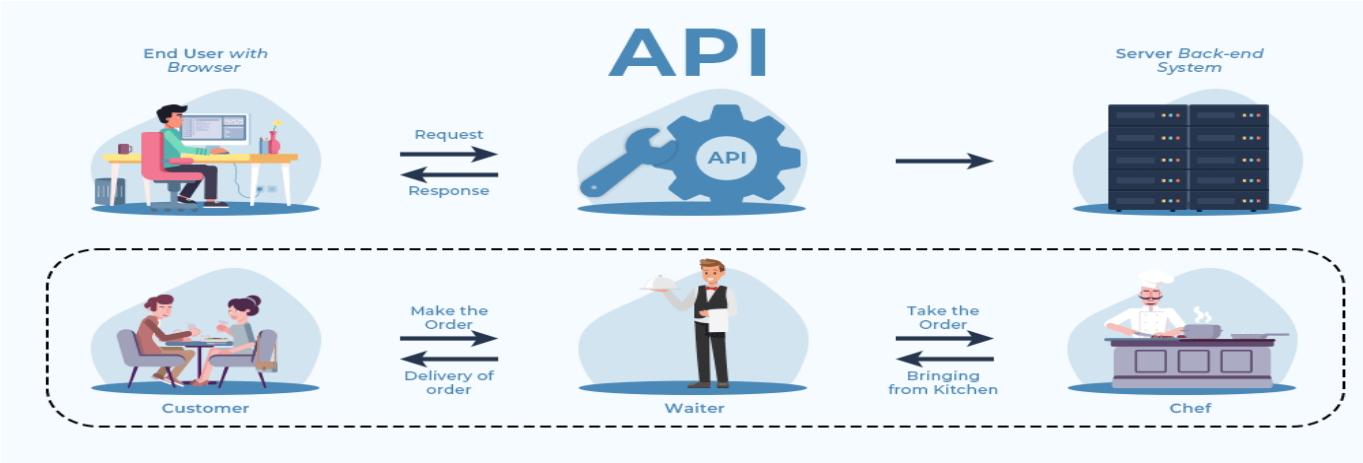
Assignment -2

1. What is a Application program?

A type of software that lets you perform a task. Apps include everything from desktop tools and programming languages to multiuser business suites. Most Linux distributions offer a central database to search for and download additional apps.

2. What is a Application program Interface?

API – (Application Programming Interface)?



To make you clear with the diagram of what is API, let's take a real-life example of an API, you can think of an API as a waiter in a restaurant who listens to your order request, goes to the chef, takes the food items ordered and gets back to you with the order. Also, if you want to look for the working of an API with the example, here's one. *You're searching for a course(let's say DSA-Self Paced) on the XYZ website, you send a request(product search requested) through an API, and the database searches for the course and checks if it's available, the API is responsible here to send your request to the database (in search of the course) and responds with the output(best DSA courses).*

What is an API?

API full form is an **Application Programming Interface** that is a collection of communication protocols and subroutines used by various programs to communicate between them. A programmer can make use of various API tools to make their program easier and simpler. Also, an API facilitates programmers with an efficient way to develop their software programs. Thus **api meaning** is when an API helps two programs or applications to communicate with each other by providing them with the necessary tools and functions. It takes the request from the user and sends it to the service provider and then again sends the result generated from the service provider to the desired user.

A developer extensively uses APIs in his software to implement various features by using an API call without writing complex codes for the same. We can create an API for an **operating system**, **database system**, **hardware system, JavaScript file**, or similar object-oriented files. Also, an API is similar to a GUI(Graphical User Interface) with one major difference. Unlike GUIs, an **application program interface** helps software developers to access web tools while a GUI helps to make a program easier to understand for users.

3.What is a system software?

https://www.geeksforgeeks.org/system-software/

4.What is user space and kernel space ?

Various layers within Linux, also showing separation between the userland and kernel space

User mode	**User applications**	*bash*, *LibreOffice*, *GIMP*, *Blender*, *0 A.D.*, *Mozilla Firefox*, ...				
	System components	**init daemon**: *OpenRC*, *runit*, *systemd*...	**System daemons**: *polkitd*, *smbd*, *sshd*, *udevd*...	**Window manager**: *X11*, *Wayland*, *SurfaceFlinger* (*Android*)	**Graphics**: *Mesa*, *AMD Catalyst*, ...	**Other libraries:** *GTK*, *Qt*, *EFL*, *SDL*, *SFML*, *FLTK*, *GNUstep*, ...
	C standard library	`fopen`, `execv`, `malloc`, `memcpy`, `localtime`, `pthread_create`... (up to 2000 subroutines) *glibc* aims to be fast, *musl* aims to be lightweight, *uClibc* targets embedded systems, *bionic* was written for Android, etc. All aim to be POSIX/SUS-compatible.				
		`stat`, `splice`, `dup`, `read`, `open`, `ioctl`, `write`, `mmap`, `close`, `exit`, etc. (about 380 system calls) The Linux kernel System Call Interface (SCI), aims to be POSIX/SUS-compatible[2]				
Kernel mode	**Linux kernel**	Process scheduling subsystem	IPC subsystem	Memory management subsystem	Virtual files subsystem	Networking subsystem
		Other components: ALSA, DRI, evdev, klibc, LVM, device mapper, Linux Network Scheduler, Netfilter Linux Security Modules: *SELinux*, *TOMOYO*, *AppArmor*, *Smack*				
Hardware (CPU, main memory, data storage devices, etc.)						

Kernel space

The memory space where the core of the operating system (kernel) executes and provides its services is known as **kernel space**. It's reserved for running device drivers, OS kernel, and all other kernel extensions.

User space

The **user space** is also known as userland and is the memory space where all user applications or application software executes. Everything other than OS cores and kernel runs here.

One of the roles of the kernel is to manage all user processes or applications within user space and to prevent them from interfering.

5. What is a shell?

The shell is the Linux command line interpreter. It provides an interface between the user and the kernel and executes programs called commands. For example, if a user enters ls then the shell

executes the ls command. The shell can also execute other programs such as applications, scripts, and user programs (e.g., written in c or the shell programming language).

6. Internal command and external command?

The UNIX system is command-based *i.e* things happen because of the commands that you key in. All UNIX commands are seldom more than four characters long.
They are grouped into two categories:

- **Internal Commands :** Commands which are built into the shell. For all the shell built-in commands, execution of the same is fast in the sense that the shell doesn't have to search the given path for them in the PATH variable, and also no process needs to be spawned for executing it.
  Examples: source, cd, fg, etc.
- **External Commands :** Commands which aren't built into the shell. When an external command has to be executed, the shell looks for its path given in the PATH variable, and also a new process has to be spawned and the command gets executed. They are usually located in /bin or /usr/bin. For example, when you execute the "cat" command, which usually is at /usr/bin, the executable /usr/bin/cat gets executed.
  Examples: ls, cat etc.

7.What is context switching?

An operating system is a program loaded into a system or computer. and manage all the other program which is running on that OS Program, it manages the all other application programs. or in other words, we can say that the OS is an interface between the user and computer hardware.

So in this article, we will learn about what is Context switching in an Operating System and see how it works also understand the triggers of context switching and an overview of the Operating System.

What is Context Switching in an Operating System?

Context switching in an operating system involves saving the context or state of a running process so that it can be restored later, and then loading the context or state of another. process and run it.

Context Switching refers to the process/method used by the system to change the process from one state to another using the CPUs present in the system to perform its job.**Example of Context Switching**

Suppose in the OS there (N) numbers of processes are stored in a Process Control Block(PCB). like The process is running using the CPU to do its job. While a process is running, other processes with the highest priority queue up to use the CPU to complete their job.

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead because the system does no

useful work while switching. Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). A typical speed is a few milliseconds. Context-switch times are highly dependent on hardware support. For instance, some processors (such as the Sun UltraSPARC) provide multiple sets of registers. A context switch here simply requires changing the pointer to the current register set. Of course, if there are more active processes than there are register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the operating system, the greater the amount of work that must be done during a context switch

Need of Context Switching

Context switching enables all processes to share a single CPU to finish their execution and store the status of the system's tasks. The execution of the process begins at the same place where there is a conflict when the process is reloaded into the system.

The operating system's need for context switching is explained by the reasons listed below.

- One process does not directly switch to another within the system. Context switching makes it easier for the operating system to use the CPU's resources to carry out its tasks and store its context while switching between multiple processes.
- Context switching enables all processes to share a single CPU to finish their execution and store the status of the system's tasks. The execution of the process begins at the same place where there is a conflict when the process is reloaded into the system.
- Context switching only allows a single CPU to handle multiple processes requests parallelly without the need for any additional processors.

Context Switching Triggers

The three different categories of context-switching triggers are as follows.

- Interrupts
- Multitasking
- User/Kernel switch

Interrupts: When a CPU requests that data be read from a disc, if any interruptions occur, context switching automatically switches to a component of the hardware that can handle the interruptions more quickly.

Multitasking: The ability for a process to be switched from the CPU so that another process can run is known as context switching. When a process is switched, the previous state is retained so that the process can continue running at the same spot in the system.

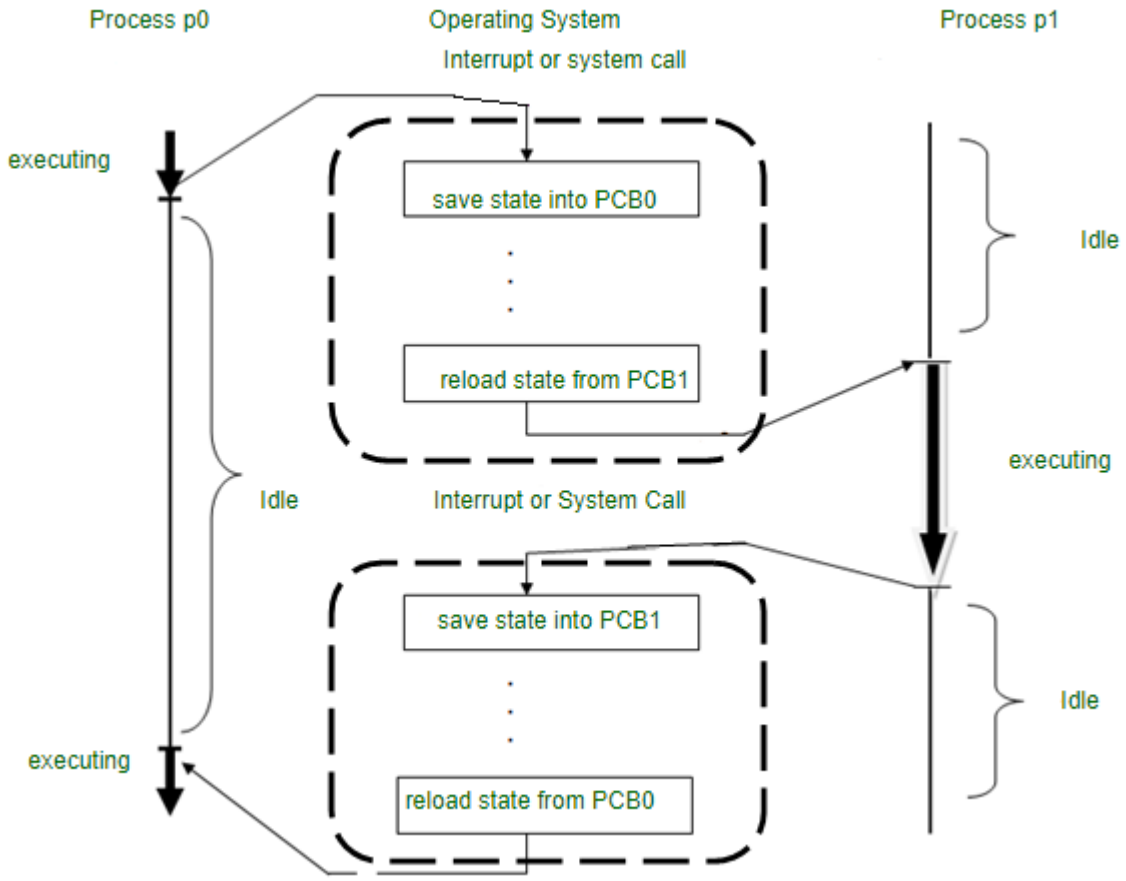
Kernel/User Switch: This trigger is used when the OS needed to switch between the user mode and kernel mode.

When switching between user mode and kernel/user mode is necessary, operating systems use the kernel/user switch.

What is Process Control Block(PCB)?

So, The Process Control block(PCB) is also known as a Task Control Block. it represents a process in the Operating System. A process control block (PCB) is a data structure used by a computer to store all information about a process. It is also called the descriptive process. When a process is created (started or installed), the operating system creates a process manager.

State Diagram of Context Switching



Working Process Context Switching

So the context switching of two processes, the priority-based process occurs in the ready queue of the process control block. These are the following steps.

- The state of the current process must be saved for rescheduling.
- The process state contains records, credentials, and operating system-specific information stored on the PCB or switch.
- The PCB can be stored in a single layer in kernel memory or in a custom OS file.
- A handle has been added to the PCB to have the system ready to run.
- The operating system aborts the execution of the current process and selects a process from the waiting list by tuning its PCB.
- Load the PCB's program counter and continue execution in the selected process.
- Process/thread values can affect which processes are selected from the queue, this can be important.

8. Why virtual memory?

Virtual Memory is a storage allocation scheme in which secondary memory can be addressed as though it were part of the main memory. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites and program-generated addresses are translated automatically to the corresponding machine addresses.

A memory hierarchy, consisting of a computer system's memory and a disk, that enables a process to operate with only some portions of its address space in memory. A virtual memory is what its name indicates- it is an illusion of a memory that is larger than the real memory. We refer to the software component of virtual memory as a virtual memory manager. The basis of virtual memory is the noncontiguous memory allocation model. The virtual memory manager removes some components from memory to make room for other components.

The size of virtual storage is limited by the addressing scheme of the computer system and the amount of secondary memory available not by the actual number of main storage locations.

It is a technique that is implemented using both hardware and software. It maps memory addresses used by a program, called virtual addresses, into physical addresses in computer memory.

1. All memory references within a process are logical addresses that are dynamically translated into physical addresses at run time. This means that a process can be swapped in and out of the main memory such that it occupies different places in the main memory at different times during the course of execution.
2. A process may be broken into a number of pieces and these pieces need not be continuously located in the main memory during execution. The combination of dynamic run-time address translation and the use of a page or segment table permits this.

If these characteristics are present then, it is not necessary that all the pages or segments are present in the main memory during execution. This means that the required pages need to be loaded into memory whenever required. Virtual memory is implemented using Demand Paging or Demand Segmentation.

Assignment -3

1. What is kernel? Which version of Linux you are using(you have used)? Hint: use "uname -a" "uname -v".

Linux Kernel is the heart of Linux operating systems. It is an open-source (source code that can be used by anyone freely) software that is most popular and widely used in the industry as well as on a personal use basis. Who created Linux and why? Linux was created by Linus Torvalds in 1991 as a hobby project. Since then, many of the users have contributed to its growth and development of it.

Before Jumping directly to the main topic “Linux Kernel” one must know a few concepts (prerequisites) to better understand the Linux Kernel.

Version of linux I am using is **5.15.0-92-generic**

2. Explain booting steps of your PC(desktop/laptop)

The boot process is a fundamental aspect of any operating system. It is the process by which the operating system loads into the computer's memory, initializes its components, and prepares to execute user applications.

The Linux boot process consists of several stages that are critical to the functionality of the operating system. Let us go through the various stages of the booting process in Linux.

Stages of Linux Boot Process

- **BIOS :** The BIOS (Basic Input/Output System) is the firmware responsible for initiating the computer's hardware components. The BIOS is the first step in any operating system’s boot process and is independent of the operating system that is to be loaded. The BIOS runs from the ROM of the computer and handles a number of important tasks such as POST or the Power On Self Test. During this stage, the firmware checks the system hardware components such as the memory, CPU, and storage devices.

  The primary function of BIOS is to check the hardware components and ensure that they are functioning correctly before passing control to the bootloader. Then, BIOS checks its list of devices for the MBR, and once found, it attempts to boot the operating system.

- **MBR :** The MBR or Master Boot Record is a boot sector that is located in the very first sector of the drive and only occupies 512 bytes. It contains the partition table and the boot loader code, which is responsible for loading the operating system into memory and starting its execution. The MBR is an essential component of the boot process, as it provides the necessary information for the boot loader to locate and load the operating system.

- **Bootloader :** The bootloader is a small piece of software responsible for loading the kernel into memory. The bootloader's primary function is to provide the user with a choice of operating systems to boot into.

  There are two main boot loaders used in Linux systems, GRUB (Grand Unified Bootloader) and **LILO (Linux Loader)**. The configuration file for GRUB is located in the /boot/grub directory and is named grub.cfg. The configuration file for LILO is located in the /etc/lilo.conf directory. Both configuration files contain instructions on how to boot the kernel, such as the location of the kernel, the root file system, and the kernel parameters.

  LILO can only be used to boot Linux-based operating systems though GRUB can be used to boot multiple operating systems such as Linux or Windows. Most modern devices these days use GRUB or GRUB 2, which is a more modern implementation of GRUB.

- **Kernel Initialization :** The kernel is the core component of the Linux operating system. Its primary function is to manage system resources such as memory, CPU, and I/O devices.

After the boot loader loads the kernel into memory, the kernel begins the process of initializing the system. During this stage, the kernel identifies and loads the necessary drivers and modules required for the system to function.

The kernel also mounts the root file system, which contains the essential files needed to start the system. Another critical aspect of kernel initialization is the initiation of the initial RAM disk (initrd) or initial RAM file system (initramfs). The initrd or initramfs contains a minimal file system and necessary kernel modules that are loaded into memory before the root file system is mounted. The initrd or initramfs allows the kernel to have access to the necessary files and drivers to mount the root file system.

- **Init System:** After the kernel completes its initialization process, it passes control to the Initialization system or Init system. A crucial step in the Linux boot process, the Init system initializes and starts all the system functions and processes necessary for the operating system to run. The Init procedure begins by deciding which run level should be used to start the system, which determines the system services and processes that are to be started. Then all the system services required for run level are initialized, such as the network stack, logging, and authentication. The user environment is then set up, and all daemon processes necessary for the selected run level are then initiated.

  There are three primary initialization systems in Linux. These are SysVinit, Upstart, and Systemd. These init systems are responsible for controlling the startup and shutdown of the system.

- **Runlevel Programs:** The Linux operating system has different run levels that determine which system services and processes are started during booting. Runlevels are numbered from 0 to 6, and each run level corresponds to a specific set of system services and processes.

  For example, **runlevel 0** is the shutdown state, while runlevel 6 is the reboot state. Runlevel 1 is also known as the single-user mode, which is a minimal state where only essential services are started. Runlevel 5 is typically the default runlevel for graphical user interface (GUI) login.

3. What is shell in lunux? Name few commonly used shells?

A shell is a program that acts as an interface between a user and the kernel. It allows a user to give commands to the kernel and receive responses from it. Through a shell, we can execute programs and utilities on the kernel. Hence, at its core, a shell is a program used to execute other programs on our system.

Different Types of Shells in Linux

If you now understand what a kernel is, what a shell is, and why a shell is so important for Linux systems, let’s move on to learning about the different types of shells that are available.

Each of these shells has properties that make them highly efficient for a specific type of use over other shells. So let us discuss the different types of shells in Linux along with their properties and features.

1. The Bourne Shell (sh)

Developed at AT&T Bell Labs by Steve Bourne, the Bourne shell is regarded as the first UNIX shell ever. It is denoted as sh. It gained popularity due to its compact nature and high speeds of operation.

This is what made it the default shell for Solaris OS. It is also used as the default shell for all Solaris system administration scripts. Start reading about shell scripting here.

However, the Bourne shell has some major drawbacks.

- It doesn't have in-built functionality to handle logical and arithmetic operations.
- Also, unlike most different types of shells in Linux, the Bourne shell cannot recall previously used commands.
- It also lacks comprehensive features to offer a proper interactive use.

The complete path-name for the Bourne shell is /bin/sh and /sbin/sh. By default, it uses the prompt # for the root user and *$* for the non-root users.

2. The GNU Bourne-Again Shell (bash)

More popularly known as the Bash shell, the GNU Bourne-Again shell was designed to be compatible with the Bourne shell. It incorporates useful features from different types of shells in Linux such as Korn shell and C shell.

It allows us to automatically recall previously used commands and edit them with help of arrow keys, unlike the Bourne shell.

The complete path-name for the GNU Bourne-Again shell is /bin/bash. By default, it uses the prompt *bash-VersionNumber#* for the root user and *bash-VersionNumber$* for the non-root users.

3. The C Shell (csh)

The C shell was created at the University of California by Bill Joy. It is denoted as csh. It was developed to include useful programming features like in-built support for arithmetic operations and a syntax similar to the C programming language.

Further, it incorporated command history which was missing in different types of shells in Linux like the Bourne shell. Another prominent feature of a C shell is "aliases".

The complete path-name for the C shell is /bin/csh. By default, it uses the prompt *hostname#* for the root user and *hostname%* for the non-root users.

4. The Korn Shell (ksh)

The Korn shell was developed at AT&T Bell Labs by David Korn, to improve the Bourne shell. It is denoted as ksh. The Korn shell is essentially a superset of the Bourne shell.

Besides supporting everything that would be supported by the Bourne shell, it provides users with new functionalities. It allows in-built support for arithmetic operations while offereing interactive features which are similar to the C shell.

The Korn shell runs scripts made for the Bourne shell, while offering string, array and function manipulation similar to the C programming language. It also supports scripts which were written for the C shell. Further, it is faster than most different types of shells in Linux, including the C shell.

The complete path-name for the Korn shell is /bin/ksh. By default, it uses the prompt # for the root user and *$* for the non-root users.

5. The Z Shell (zsh)

The Z Shell or zsh is a sh shell extension with tons of improvements for customization. If you want a modern shell that has all the features a much more, the zsh shell is what you're looking for.

Some noteworthy features of the z shell include:

- Generate filenames based on given conditions
- Plugins and theming support
- Index of built-in functions
- Command completion
- and many more…

Let us summarise the different shells in Linux which we discussed in this tutorial in the table below.

Shell	Complete path-name	Prompt for root user	Prompt for non root user
Bourne shell (sh)	/bin/sh and /sbin/sh	#	$
GNU Bourne-Again shell (bash)	/bin/bash	bash-VersionNumber#	bash-VersionNumber$
C shell (csh)	/bin/csh	#	%
Korn shell (ksh)	/bin/ksh	#	$
Z Shell (zsh)	/bin/zsh	<hostname>#	<hostname>%

4. Explain internal and external command. How to recognize the command is internal or external?

Internal commands are commands that are already loaded in the system. They can be executed any time and are independent. On the other hand, external commands are loaded when the user requests for them. Internal commands don’t require a separate process to execute them. External commands will have an individual process. Internal commands are a part of the shell while external commands require a Path. If the files for the command are not present in the path, the external command won’t execute.

What is the difference between internal and external commands?

- The commands that are directly executed by the shell are known as internal commands. No separate process is there to run these commands.
- The commands that are executed by the kernel are knows as external commands. Each command has its unique process id.

5. What is ps and top command?

Syntax of `ps` Command in Linux

The `ps` command provides a snapshot of the current processes on your system. The basic syntax is as follows:

```
ps [options]
```

Without any options, **`ps`** displays information about the processes associated with the current terminal session. However, to harness the full potential of the **`ps`** command, various options can be used to customize the output.

Options for `ps` Command to List Running Processes in Linux

Some commonly used options

Options	Description
a	List all ruining processes for all users.
-A, -e	Lists all processes on the entire system, offering a complete overview of running tasks and programs.
-a	List all processes except session leaders (instances where the process ID is the same as the session ID) and processes not associated with a terminal.
-d	Lists all processes except session leaders, providing a filtered view of processes running on the system.
--deselect, -N	Lists all processes except those that meet specific user-defined conditions.

Options	Description
f	Displays the hierarchy of processes in a visual ASCII art format, illustrating parent-child relationships.
-j	Presents the output in the jobs format, providing detailed information such as process ID, session ID, and command.
T	Lists all processes associated with the current terminal, aiding in focusing on tasks related to a specific terminal.
r	Only lists running processes, useful for monitoring system performance.
u	Expands the output to include additional information like CPU and memory usage.
-u	Specifies a username, listing processes associated with that user.
x	Includes processes without a TTY, showing background processes not tied to a specific terminal session.

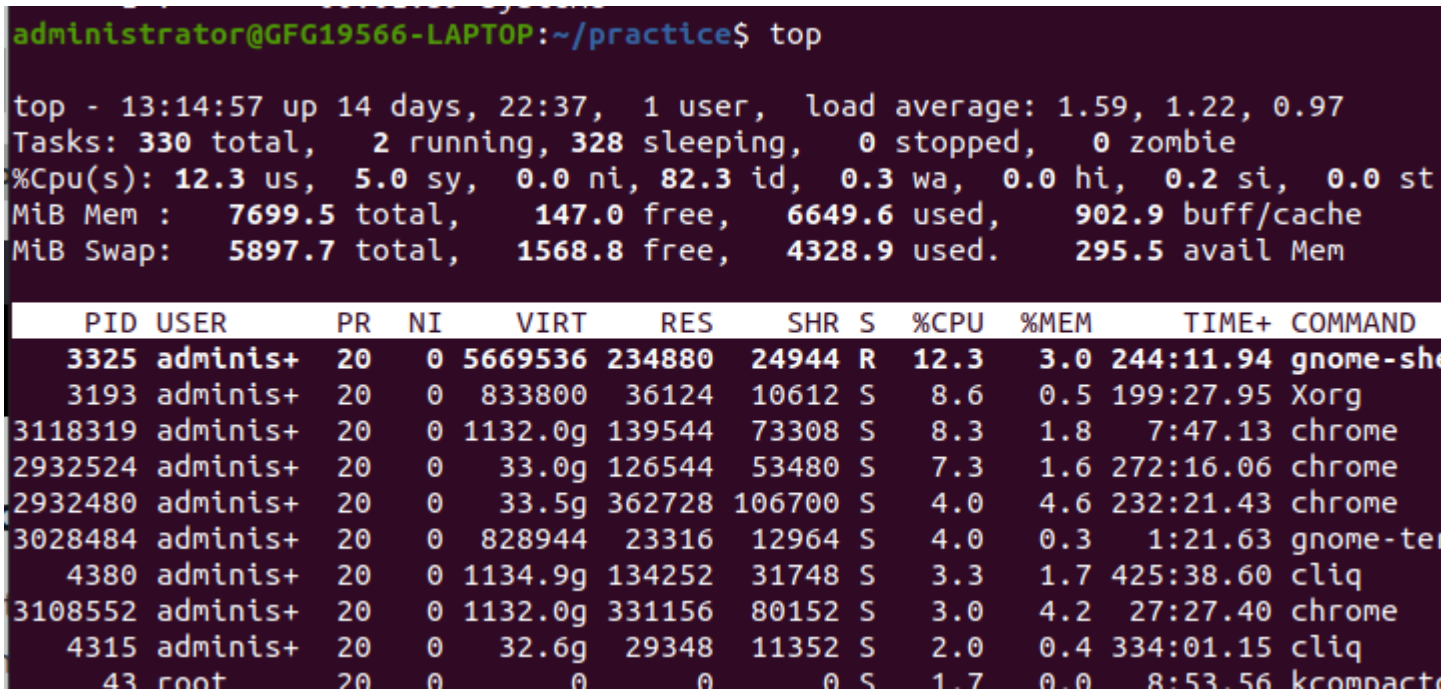
The `top` Command to List Running Processes in Linux

In Linux, the `top` command is a dynamic and interactive tool that provides real-time information about system processes. It offers a comprehensive view of running processes, system resource utilization, and other critical system metrics. This article explores how to effectively use the `top` command to monitor and manage processes.

Launching `top`

To launch the `top` command, open a terminal and simply type:

```
top
```



list all processes running in Linux in top

Process-related information including:

- **PID**: Process ID
- **USER**: Owner of the process
- **PR**: Priority
- **NI**: Nice value
- **VIRT**: Virtual memory usage
- **RES**: Resident set size (non-swapped physical memory used)
- **SHR**: Shared memory
- **S**: Process status (S: Sleeping, R: Running, I: Idle)
- **%CPU**: Percentage of CPU usage
- **%MEM**: Percentage of memory usage
- **TIME+**: Total CPU time
- **COMMAND**: Command or process name

6. Explain

i. ps-e | wc -I

ii. ps-I

iii.ps-e | grep pts.?

-e	Lists all processes on the entire system, offering a complete overview of running tasks and programs.

ps -e

```
administrator@GFG19566-LAPTOP:~/practice$ ps -e
    PID TTY          TIME CMD
      1 ?        00:01:37 systemd
      2 ?        00:00:00 kthreadd
      3 ?        00:00:00 rcu_gp
      4 ?        00:00:00 rcu_par_gp
      5 ?        00:00:00 slub_flushwq
```

ps -e option to view all running processes in linux

ps -l

Display BSD long format.
Long format. The **-y** option is often useful with this.

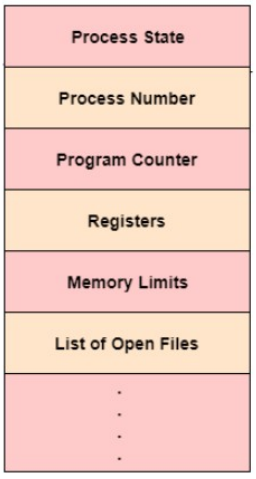
7. What is PCB ? List members of it.

Process Control Block is a data structure that contains information of the process related to it. The process control block is also known as a task control block, entry of the process table, etc.

It is very important for process management as the data structuring for processes is done in terms of the PCB. It also defines the current state of the operating system.

Structure of the Process Control Block

The process control stores many data items that are needed for efficient process management. Some of these data items are explained with the help of the given diagram −



Process Control Block (PCB)

The following are the data items −

Process State

This specifies the process state i.e. new, ready, running, waiting or terminated.

Process Number

This shows the number of the particular process.

Program Counter

This contains the address of the next instruction that needs to be executed in the process.

Registers

This specifies the registers that are used by the process. They may include accumulators, index registers, stack pointers, general purpose registers etc.

List of Open Files

These are the different files that are associated with the process

CPU Scheduling Information

The process priority, pointers to scheduling queues etc. is the CPU scheduling information that is contained in the PCB. This may also include any other scheduling parameters.

Memory Management Information

The memory management information includes the page tables or the segment tables depending on the memory system used. It also contains the value of the base registers, limit registers etc.

I/O Status Information

This information includes the list of I/O devices used by the process, the list of files etc.

Accounting information

The time limits, account numbers, amount of CPU used, process numbers etc. are all a part of the PCB accounting information.

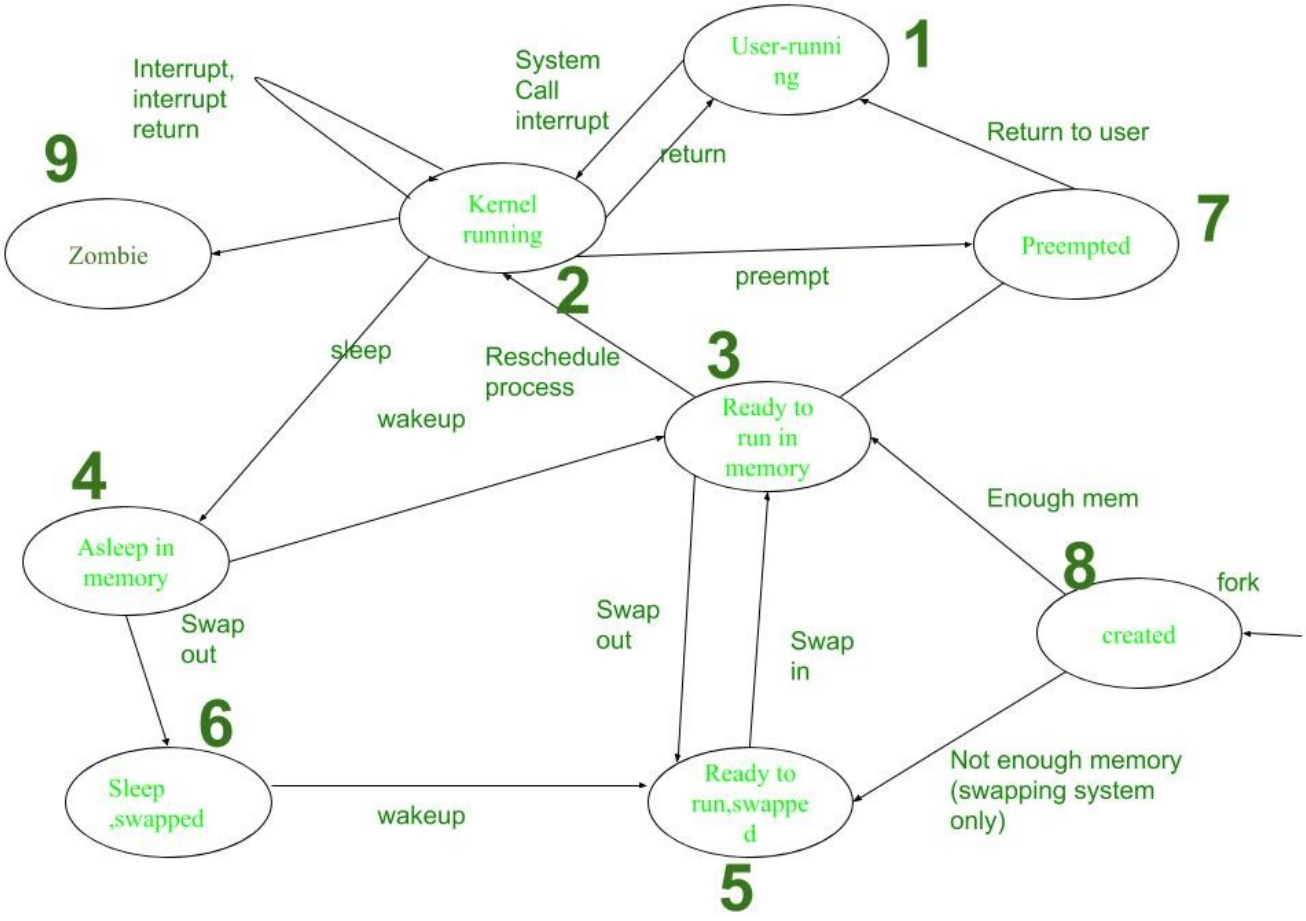
Location of the Process Control Block

The process control block is kept in a memory area that is protected from the normal user access. This is done because it contains important process information. Some of the operating systems place the PCB at the beginning of the kernel stack for the process as it is a safe location.

8. Explain the states of a process and show the state-transition -diagram?

s has been swapped to secondary storage and is at a blocked state.

- **Asleep in memory-** Process is in memory(not swapped to secondary storage) but is in blocked state.



The numbers indicate the steps that are followed.

Process Transitions

The working of Process is explained in following steps:

1. **User-running:** Process is in user-running.
2. **Kernel-running:** Process is allocated to kernel and hence, is in kernel mode.
3. **Ready to run in memory:** Further, after processing in main memory process is rescheduled to the Kernel.i.e.The process is not executing but is ready to run as soon as the kernel schedules it.
4. **Asleep in memory:** Process is sleeping but resides in main memory. It is waiting for the task to begin.
5. **Ready to run, swapped:** Process is ready to run and be swapped by the processor into main memory, thereby allowing kernel to schedule it for execution.
6. **Sleep, Swapped:** Process is in sleep state in secondary memory, making space for execution of other processes in main memory. It may resume once the task is fulfilled.
7. **Pre-empted:** Kernel preempts an on-going process for allocation of another process, while the first process is moving from kernel to user mode.
8. **Created:** Process is newly created but not running. This is the start state for all processes.

9. **Zombie:** Process has been executed thoroughly and exit call has been enabled. The process, thereby, no longer exists. But, it stores a statistical record for the process. This is the final state of all processes.

9. How can we get/modify the stack limit of a process? Write a program to make the stack of a process double the current value?
Hint: use getrlimit and setrlimit.

The Old limits values may vary depending upon the system.

Now, If you try to open a new file, it will show run time error, because maximum 3 files can be opened and that are already being opened by the system(STDIN, STDOUT, STDERR).

```
// C program to demonstrate error when a
// process tries to access resources beyond
// limit.
#include <stdio.h>
#include <sys/resource.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main() {

struct rlimit old_lim, lim, new_lim;

    // Get old limits
    if( getrlimit(RLIMIT_NOFILE, &old_lim) == 0)
        printf("Old limits -> soft limit= %ld \t"
          " hard limit= %ld \n", old_lim.rlim_cur,
                                 old_lim.rlim_max);
    else
        fprintf(stderr, "%s\n", strerror(errno));

    // Set new value
    lim.rlim_cur = 3;
    lim.rlim_max = 1024;


    // Set limits
    if(setrlimit(RLIMIT_NOFILE, &lim) == -1)
        fprintf(stderr, "%s\n", strerror(errno));

    // Get new limits
    if( getrlimit(RLIMIT_NOFILE, &new_lim) == 0)
        printf("New limits -> soft limit= %ld \t"
          " hard limit= %ld \n", new_lim.rlim_cur,
```

```
                                new_lim.rlim_max);
    else
        fprintf(stderr, "%s\n", strerror(errno));

    // Try to open a new file
    if(open("foo.txt", O_WRONLY | O_CREAT, 0) == -1)
        fprintf(stderr, "%s\n", strerror(errno));
    else
            printf("Opened successfully\n");

    return 0;
}
```

Output:

```
Old limits -> soft limit= 1048576      hard limit= 1048576

New limits -> soft limit= 3                     hard limit= 1024

Too many open files
```

There is another system call **prlimit()** that combines both the system calls.
For more details, check manual by typing

```
man 2 prlimit
```

File operations in linux

1. Theoretical Background

The following article presents the way to use the most common system calls in order to make input-output operations on files, as well as operations to handle files and directories in the Linux operating system.

2. File descriptors

The operating system assigns internally to each opened file a descriptor or an identifier (usually this is a positive integer). When opening or creating a new file the system returns a file descriptor to the process that executed the call. Each application has its own file descriptors. By convention, the first three file descriptors are opened at the beginning of each process. The 0 file descriptor identifies the standard input, 1 identifies the standard output and 2 the standard output for errors. The rest of the descriptors are used by the processes when opening an ordinary, pipe or special file, or directories. There are five system calls that generate file descriptors: *create*, *open*, *fcntl*, *dup* and *pipe*.

3. System calls when working with files

3.1. System call OPEN

Opening or creating a file can be done using the system call open. The syntax is:

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int open(**const char** *path,

```
int flags,... /* mode_t mod */);
```

This function returns the file descriptor or in case of an error -1. The number of arguments that this function can have is two or three. The third argument is used only when creating a new file. When we want to open an existing file only two arguments are used. The function returns the smallest available file descriptor. This can be used in the following system calls: *read*, *write*, *lseek* and *close*. The effective UID or the effective GID of the process that executes the call has to have read/write rights, based on the value of the argument *flags*. The file pointer is places on the first byte in the file. The argument *flags* is formed by a bitwise OR operation made on the constants defined in the *fcntl.h* header.

O_RDONLY

Opens the file for reading.

O_WRONLY

Opens the file for writing.

O_RDWR

The file is opened for reading and writing.

O_APPEND

It writes successively to the end of the file.

O_CREAT

The file is created in case it didn\92t already exist.

O_EXCL

If the file exists and O_CREAT is positioned, calling *open* will fail.

O_NONBLOCK

In the case of pipes and special files, this causes the open system call and any other future I/O operations to never block.

O_TRUNC

If the file exists all of its content will be deleted.

O_SYNC

It forces to write on the disk with function *write*. Though it slows down all the system, it can be useful in critical situations.

The third argument, *mod*, is a bitwise OR made between a combination of two from the following list:

S_IRUSR, S_IWUSR, S_IXUSR

Owner: *read*, *write*, *execute*.

S_IRGRP, S_IWGRP, S_IXGRP

Group: *read*, *write*, *execute*.

S_IROTH, S_IWOTH, S_IXOTH

Others: *read*, *write*, *execute*.

The above define the access rights for a file and they are defined in the *sys/stat.h* header.

3.2. System call CREAT

A new file can be created by:

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

```
int creat(const char *path, mode_t mod);
```

The function returns the file descriptor or in case of an error it returns the value -1. This call is equivalent with:

```
open(path, O_WRONLY | O_CREAT | O_TRUNC, mod);
```

The argument *path* specifies the name of the file, while *mod* defines the access rights. If the created file doesn\92t exist, a new i-node is allocated and a link is made to this file from the directory it was created in. The owner of the process that executes the call - given by the effective UID and the effective GUID - must have writing permission in the directory. The open file will have the access rights that were specified in the second argument (see *umask*, too). The call returns the smallest file descriptor available. The file is opened for writing and its initial size is 0. The access time and the modification time are updated in the i-node. If the file exists (permission to search the directory is needed), it looses its contents and it will be opened for writing. The ownership and the access permissions won\92t be modified. The second argument is ignored.

3.3. System call READ

When we want to read a certain number of bytes starting from the current position in a file, we use the *read* call. The syntax is:

#include <unistd.h>

ssize_t read(**int** fd, **void*** buf, **size_t** noct);

The function returns the number of bytes read, 0 for end of file (EOF) and -1 in case an error occurred. It reads *noct* bytes from the open file referred by the *fd* descriptor and it puts it into a buffer *buf*. The pointer (current position) is incremented automatically after a reading that certain amount of bytes. The process that executes a read operation waits until the system puts the data from the disk into the buffer.

3.4. System call WRITE

For writing a certain number of bytes into a file starting from the current position we use the *write* call. Its syntax is:

#include <unistd.h>

```
ssize_t write(int fd, const void* buf, size_t noct);
```

The function returns the number of bytes written and the value -1 in case of an error. It writes *noct* bytes from the buffer *buf* into the file that has as its descriptor *fd*. It is interesting to note that the actual writing onto the disk is delayed. This is done at the initiative of the root, without informing the user when it is done. If the process that did the call or an other process reads the data that haven\ 92t been written on the disk yet, the system reads all this data out from the cache buffers. The delayed writing is faster, but it has three disadvantages:

a) a disk error or a system error may cause loosing all the data

b) a process that had the initiative of a write operation cannot be informed in case a writing error occurred

c) the physical order of the write operations cannot be controlled.

To eliminate these disadvantages, in some cases the O_SYNC is used. But as this slows down the system and considering the reliability of today\92s systems it is better to use the mechanism which includes using cache buffers.

3.5. System call CLOSE

For closing a file and thus eliminating the assigned descriptor we use the system call *close*.

#include <unistd.h>

```
int close(int fd);
```

The function returns 0 in case of success and -1 in case of an error. At the termination of a process an open file is closed anyway.

3.6. System call LSEEK

To position a pointer (that points to the current position) in an absolute or relative way can be done by calling the *lseek* function. Read and write operations are done relative to the current position in the file. The syntax for *lseek* is:

#include <sys/types.h>

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int ref);
```

The function returns the displacement of the new current position from the beginning of the file or -1 in case of an error. There isn\92t done any I/O operation and the function doesn\92t send any commands to the disk controller. It *ref* is set to SEEK_SET the positioning is done relative to the beginning of the file (the first byte in the file is at position 0). If *ref* is SEEK_CUR the positioning is done relative to the current position. If *ref* is SEEK_END then the positioning is done relative to the end of the file. The system calls *open*, *creat*, *write* and *read* execute an *lseek* by default. If a file was opened using the symbolic constant O_APPEND then an *lseek* call is made to the end of the file before a write operation.

3.7. System call LINK

To link an existing file to another directory (or to the same directory) link can be used. To make such a link in fact means to set a new name or a path to an existing file. The *link* system call creates a hard link. Creating symbolic links can be done using *symlink* system call. The syntax of link is:

#include <unistd.h>

int link(**const char*** oldpath, **const char*** newpath);

int symlink(**const char*** oldpath, **const char*** newpath);

The function returns 0 in case of success and -1 in case of an error. The argument *oldpath* has to be a path to an existing file. Only the root has the right to set a link to a directory.

3.8. System call UNLINK

To delete a link (a path) in a directory we can use the *unlink* system call. Its syntax is:

#include <unistd.h>

int unlink(**const char*** path);

The function returns 0 in case of success and -1 otherwise. The function decrements the hard link counter in the i-node and deletes the appropriate directory entry for the file whose link was deleted. If the number of links of a file becomes 0 then the space occupied by the file and its i-node will be freed. Only the root can delete a directory.

3.9. System calls STAT, LSTAT and FSTAT

In order to obtain more details about a file the following system calls can be used: *stat*, *lstat* or *fstat*.

#include <sys/types.h>

#include <sys/stat.h>

int stat(**const char*** path, **struct stat*** buf);

int lstat(**const char*** path, **struct stat*** buf);

int fstat(**int** df, **struct stat*** buf);

These three functions return 0 in case of success and -1 in case of an error. The first two gets as input parameter a name of a file and completes the structure of the buffer with additional information read from its i-node. The *fstat* function is similar, but it works for files that were

already opened and for which the file descriptor is known. The difference between *stat* and *lstat* is that in case of a symbolic link, function *stat* returns information about the linked (refered) file, while *lstat* returns information about the symbolic link file. The *struct stat* structure is described in the *sys/stat.h* header and has the following fields:

```
struct stat {
        mode_t st_mode;   /* file type & rights */
        ino_t st_ino;     /* i-node */
        dev_t st_dev;     /* număr de dispozitiv (SF) */
        nlink_t st_nlink; /* nr of links */
        uid_t st_uid;     /* owner ID */
        gid_t st_gid;     /* group ID */
        off_t st_size;    /* ordinary file size */
        time_t st_atime; /* last time it was accessed */
        time_t st_mtime; /* last time it was modified */
        time_t st_ctime; /* last time settings were changed */
                dev_t st_rdev;   /* nr. dispozitiv */
                _________/* pt. fişiere speciale /
        long st_blksize; /* optimal size of the I/O block */
        long st_blocks;   /* nr of 512 byte blocks allocated */
        };
```

The Linux command that the most frequently uses this function is *ls*. Type declarations for the members of this structure can be found in the *sys/stat.h* header. The type and access rights for the file are encrypted in the *st_mode* field and can be determined using the following macros:

Table 1. Macros for obtaining the type of a file

Macro	**Meaning**
`S_ISREG(st_mode)`	Regular file.
`S_ISDIR(st_mode)`	Directory file.
`S_ISCHR(st_mode)`	Special device of type character.
`S_ISBLK(st_mode)`	Special device of type block.

`S_ISFIFO(st_mode)`	Pipe file or FIFO.
`S_ISLNK(st_mode)`	Symbolic link.

Decrypting the information contained in the *st_mode* field can be done by testing the result of a bitwise AND made between the *st_mode* field and one of the constants (bit mask): S_IFIFO, S_IFCHR, S_IFBLK, S_IFDIR, S_IFREG, S_IFLNK, S_ISUID (*suid* bit set), S_ISGID (*sgid* bit set), S_ISVTX (*sticky* bit set), S_IRUSR (read right for the owner), S_IWUSR (write right for the owner), S_IWUSR (execution right for the owner), etc.

3.10. System call ACCESS

When opening a file with system call *open* the root verifies the access rights in function of the UID and the effective GID. There are some cases though when a process verifies these rights based upon the real UID and GID. A situation when this can be useful is when a process is executed with other access right using the *suid* or *sgid* bit. Even though a process may have root rights during execution, sometimes it is necessary to test whether the real user can or cannot access the file. For this we can use *access* which allows verifying the access rights of a file based on the real UID or GID. The syntax for this system call is:

#include <unistd.h>

int access(**const char*** path, **int** mod);

The function returns 0 if the access right exists and -1 otherwise. The argument mod is a bitwise AND between R_OK (permission to read), W_OK (permission to write), X_OK (execution right), F_OK (the file exists).

3.11. System call UMASK

To enhance the security in case of operations regarding the creation of files, the Linux operating system offers a default mask to reset some access rights. Encrypting this mask is made in a similar way to the encrypting of the access rights in the i-node of a file. When creating a file those bits that are set to 1 in the mask invalidate the corresponding bits in the argument that specify the access rights. The mask doesnot affect the system call *chmod*, so the processes can explicitly set their access rights independently form the *umask* mask. The syntax for the call is:

#include <sys/types.h>

#include <sys/stat.h>

mode_t umask(**mode_t** mask);

The function returns the value of the previous mask. The effect of the call is shown below:

```
    main() /* test umask */
{
        int fd;
        umask(022);
```

```
    if ((fd=creat("temp", 0666))==-1)
                perror("creat");
    system("ls -l temp");
}
```

The result will be of the following form:

```
    -rw-r--r-- temp
```

Note that the write permission for the group and other users beside the owner was automatically reset.

3.12. System call CHMOD

To modify the access rights for an existing file we use:

```
#include <sys/types.h>
#include <sys/stat.h>
```

int chmod(**const char*** path, **mode_t** mod);

The function returns 0 in case of a success and -1 otherwise. The *chmod* call modifies the access rights of the file specified by the *path* depending on the access rights specified by the *mod* argument. To be able to modify the access rights the effective UID of the process has to be identical to the owner of the file or the process must have root rights.

The *mod* argument can be specified by one of the symbolic constants defined in the *sys/stat.h* header. Their effect can be obtained by making a bitwise OR operation on them:

Table 2. Bit masks for testing the access rights of a file

Mode	**Description**
`S_ISUID`	Sets the suid bit.
`S_ISGID`	Sets the sgid bit.
`S_ISVTX`	Sets the sticky bit.
`S_IRWXU`	Read, write, execute rights for the owner obtained from: `S_IRUSR | S_IWUSR | S_IXUSR`
`S_IRWXG`	Read, write, execute rights for the group obtained from: `S_IRGRP | S_IWGRP | S_IXGRP`
`S_IRWXO`	Read, write, execute rights for others obtained from: `S_IROTH | S_IWOTH | S_IXOTH`

3.13. System call CHOWN

This system call is used to modify the owner (UID) and the group (GID) that a certain file belongs to. The syntax of the function is:

#include <sys/types.h>

#include <unistd.h>

int chown(**const char*** path, **uid_t** owner, **gid_t** grp);

The function returns 0 in case of success and -1 in case of an error. Calling this function will change the owner and the group of the file specified by the argument *path* to the values specified by the arguments *owner* and *grp*. None of the users can change the owner of any file (even of his/her own files), except the root user, but they can change the GID for their own files to that of any group they belong to.

3.14. System call UTIME

There are three members of the structure *stat* that refer to time. They are presented in the following table:

Table 3. Timing information associated with a file

Field	**Description**	**Operation**
`st_atime`	Last time the data in the file was accessed	Read
`st_mtime`	Last time the data in the file was modified	Write
`st_ctime`	Changing the settings for the i-node	chmod, chown

The difference between the time the file was last modified and the change in the setting of the i-node is that the first one refers to the time when the contents of the file were modified while the second one refers to the time when the information in the i-node was last modified. This is due to the fact that the information in the i-node is kept separately from the contents of the file. System calls that change the i-node are those ones which modify the access rights of a file, change the UID, change the number of links, etc. The system doesnot keep the time when the i-node was last accessed. This is why neither of the system calls *access* or *stat* do not change these times.

The access time and last modification time of any kind of files can be changed by calling one of the system call presented below:

#include <sys/time.h>

int utimes(**const char*** path,
const struct timeval* times);

int lutimes(**const char*** path,
const struct timeval* times);

int futimes(**int** fd, **const struct timeval*** times);

The functions return 0 in case of success and -1 otherwise. Only the owner of a file or the root can change the times associated with a file. The parameter *times* represents the address (pointer) of a list of two *timeval* structures, corresponding to the access and modification time. The fields of the *timeval* structure are:

struct timeval {

long tv_sec; /* seconds passed since 1.01.1970 */

suseconds_t tv_usec; /* microseconds */

}

To obtain the current time in the form it is required by the *timeval* structure, we can use the *gettimeofday* function. For different conversions between the normal format of a data and hour and the format specific to the *timeval* structure the function *ctime* can be used or any other functions belonging to the same family (for more details see the textbook).

4. Functions for working with directories

A directory can be read as a file by anyone whoever has reading permissions for it. Writing a directory as a file can only be done by the kernel. The structure of the directory appears to the user as a succession of structures named directory entries. A directory entry contains, among other information, the name of the file and the i-node of this. For reading the directory entries one after the other we can use the following functions:

#include <sys/types.h>

#include <dirent.h>

DIR* opendir(**const char*** pathname);

struct dirent* readdir(**DIR*** dp);

void rewinddir(**DIR*** dp);

```
int closedir(DIR* dp);
```

The *opendir* function opens a directory. It returns a valid pointer if the opening was successful and NULL otherwise.

The *readdir* function, at every call, reads another directory entry from the current directory. The first *readdir* will read the first directory entry; the second call will read the next entry and so on. In case of a successful reading the function will return a valid pointer to a structure of type *dirent* and NULL otherwise (in case it reached the end of the directory, for example).

The *rewinddir* function repositions the file pointer to the first directory entry (the beginning of the directory).

The *closedir* function closes a previously opened directory. In case of an error it returns the value -1.

The structure *dirent* is defined in the *dirent.h* file. It contains at least two elements:

struct dirent {

```
ino_t d_fileno;              // i-node nr.
char d_name[MAXNAMLEN + 1];  // file name
}
```

5. Examples

5.1. Example 1

Write a program that creates a file with a 4K bytes free space. Such files are called files with holes.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
char buf1[]="LAB ";
char buf2[]="OS Linux";
int main( void)
{
        int fd;
        if ((fd=creat("file.gol", 0666)) < 0) {
                        perror("Creation error");
             exit (1);
        }
        if (write(fd, buf1, sizeof(buf1)) < 0)
        perror("Writing error");
        exit(2);
        }
        if (lseek(fd, 4096, SEEK_SET) < 0)
        perror("Positioning error");
        exit(3);
        }
        if (write(fd, buf2, sizeof(buf2)) < 0)
        perror("Writing error");
        exit(2);
        }
```

}

Trace the execution of the program with the help of the following commands:

ls -l

```
stat file.gol
od -c file.gol
```

5.2. Example 2

Write a program that copies the contents of an existing file into another file. The names of the two file should be read as an input from the command line. You may presume that any of the commands *read* or *write* may cause errors.

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
#include <fcntl.h>
#define BUFSIZE 512
int main (int argc, char** argv)
{
        int from, to, nr, nw, n;
        char buf[BUFSIZE];

        if ((from=open(argv[1], O_RDONLY)) < 0) {
        perror(Error opening source file);
        exit(1);
    }

    if ((to=creat(argv[2], 0666)) < 0) {
        perror("Error creating destination file");
        exit(2);
    }
        while((nr=read(from, buf, sizeof( buf))) != 0) {
     if (nr < 0) {
       perror("Error reading source file");
                    exit(3);
```

```
      }
      nw=0;
      do {
        if ((n=write(to, &buf[nw], nr-nw)) < 0) {
          perror("Error writing destination file");
          exit(4);
        }
        nw += n;
      } while (nw < nr);
    }
    close(from); close(to);
}
```

5.3. Example 3

Write a program that displays the contents of a directory, specifying the type for each of its files. The name for the directory should be an input parameter.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
void listDir(char *dirName)
{
  DIR* dir;
  struct dirent *dirEntry;
  struct stat inode;
  char name[1000];
  dir = opendir(dirName);
     if (dir == 0) {
    perror ("Eroare deschidere fisier");
    exit(1);
     }
     while ((dirEntry=readdir(dir)) != 0) {
    sprintf(name,"%s/%s",dirName,dirEntry->d_name);
    lstat (name, &inode);
```

```
      // test the type of file
     if (S_ISDIR(inode.st_mode))
        printf("dir ");
     else if (S_ISREG(inode.st_mode))
          printf ("fis ");
         else
           if (S_ISLNK(inode.st_mode))
             printf ("lnk ");
          else;
     printf(" %s\n", dirEntry->d_name);
   }
}
int main(int argc, char **argv)
{
   if (argc != 2) {
     printf ("UTILIZARE: %s nume_dir\n", argv[0]);
     exit(0);
   }
   printf(\94Continutul directorului este:\n\94);
   listDir(argv[1]);
}
```