#Hardware- CPU, I/O peripherals(ex-keyboard), memory, I/O controller devices (devices which are on the board ex-Usb controller).

#Operating System-a set of programs which manipulate and interact with the hardware and provide a suitable environment to run various applications.

Linux Advantage-
1.free and open source.
2.Portable on hardware machine.
3.Linux is secured. (don't need any antivirus)
4.It is scalable or modular. (In android development Linux is the core with size <1gb, whereas windows can't be scaled at this level)
5.Expected to run 24/7 without rebooting.
6.takes very short debug time (errors, defect, failures, and faults)

Booting Sequence-How Linux works internally. It has 4 steps- power on, BIOS, bootloader, kernel startup.
POWERON-CPU is executing a predefined jump instruction present in the CPU register memory. Then searches for BIOS code in predefined ROM (Read Only Memory)
BIOS - (basic I/o system) performs tests for hardware controllers i.e.; all the controllers present on the board (audio, video, ram, hard disk) and then searches the boot loader in hard disk.
BOOT LOADER- scans the hard disk for operating system and then loads the OS images to the RAM (Random Access Memory). <u>(OS is the first program to get loaded on RAM).</u>
<u>Grand Unified bootloader is the bootloader for Linux.</u>
KERNEL STARTUP- It is executed in the RAM until the system is powered off. While getting started, it creates lots of data structures for file system, memory, interrupt management, process management, devices, and threads.
These threads have some INIT process which contains the login process.

ELF 64-bit format- It is the executable - loadable file format. The compiler in Linux will process and pack the data of the program in this ELF format which is understood by Linux.
VIM EDITOR-
VIM works in 2 modes-
1.Insertion and command.
Insertion- whatever we type is displayed on the screen.
Command- we get more options to control the operation through keys or commands.
2.Most popularly used editor in Linux and Linux operating systems.
3.It is a command-oriented editor.
VIM commands-

1. “l” to move to the right side.
2. “h” to move to the left side.
3. “j” for down.
4. “k” for up.
5. “o” to get new line below the cursor also takes to the insert mode.
6. “O” to add a new line above the cursor and to put in the insertion mode.
7. “a” to insert after the cursor position.
8. “A” to insert at the end of the line.
9. “yy” to copy a line.
10. “p” to paste.
11. “x” to delete the data.
12. “u” to undo.

Redirector operator- “>” stores output of the current file in a new file so that previous output can be maintained. “./output > new file”.s

SOURCE CODE TOOLS-

C-SCOPE- used to examine the symbols (variables, functions, or macros) of the source code. Once c-scope is entered from the terminal it’ll take you to an interactive screen with multiple options to examine the source code.

C-scope –R creates the reference file “cscope.out” for c-scope to perform its functions.

HEADER FILES CONTAINS THE FUNCTION DECLARATIONS AND LIBRARIES CONTAINS THE FUNCTION DEFINITIONS. (/USR/INCLUDE, /USR/LIB)

CTAGS- creates a tags file for browsing the files in the current folder. This tags file contains all the names found in source file or the header file such as name of a variable, functions, or macros. It is a programming tool used for software development process in large applications. Used in Linux, Unix variant operators and supports many languages along with C. tags also show the type of variables, functions, or macros. It creates a file which shows the index of the file. Ctags use locators which locate where the name is used, the path name of the name, type of name, and the line of the name. This shows the complete information about all the objects or symbols which are used in source file or header files.

Commands-

1. Ctags –version.
2. Ctags --list-languages

Advantages-

1. Provides quick access across the files.
2. Provides complete function information (name, path name, use)
3. Tells whether the particular name is function or variable.

Compiling multiple files to create a single executable file-

Gcc file1.c file2.c -o objfile or include the functions declared in the other files in the file containing the main function (only be present in one file) as #include"function name."

Another option is Makefile and make tool.

Writing all the file names for multiple file compilation takes much longer in gcc command, to solve this replace gcc with Makefile and make tool that helps in compiling multiple files. It is difficult for gcc to compile the source files which are in different directories or folders. **Make and Makefile will manage the source files and the header files to be at one place.**

MAKE TOOL-

1. It is a program build development tool which makes one executable file in large application software.
2. It automatically determines which files need to be re-compiled.
3. It gets all the information to build (create a final executable file) a program from a file known as Makefile.
4. It executes shell commands.

Makefile-

1. It contains a set of commands which are the same as terminal commands that use variables names and target names to create objfile.
2. Gcc commands, which are shell commands, are executed by Makefile.
3. Makefile is a text file that contains the complete procedure for building a "build executable."

RULES-

1. M of Makefile should be capital.
2. Makefile, source file and header files all should be in the same directory.

PROCEDURE-

1. They should contain a variable and they must have some value.

Variable=value; - variable initializations.

2. Makefiles talks about margins (from where the variable should start from) which is any line number should always start from first column.
3. It also contains target (labels to your Makefile) in which we write the instructions, which are executed by Makefile.

SYNTAX FOR TARGET-

Targetname:

….tab space...write the command to be executed by the Makefile

Ex-

```
All:
        Gcc five.c -o five
```

MAKEFILE DEPENDENCIES- targets of the makefile can have optional dependencies on other files called as dependent file which is an input to create the target files.

FIND COMMAND-

It is a shell command that finds the file name that ends with particular extension.
$ find –name '*.c' gives all the file names ending with .c, this command is to be given to a variable in Makefile.
SRC=$(shell find –name '*.c') after this command all the .c files are stored in the variable named SRC.
All:
Gcc $(SRC) -o output – this replaces the SRC with all the .c files, called as de-referencing.
If the source files and header files are in different folders in the same directory, then we include them as-
SRC=source files
INC=./inc(the folder in which the header file is present)
All:
Gcc $(SRC) -I(INC) -o output "-I includes the header file"

GDB- is a powerful debugging tool used for C and C++ programs. It is a free software used with Linux and Unix OS.
1. GDB allows the user to stop the program execution in the middle of any step and allows the user to proof (check for error) during an application crash.
2.It operates with an executable file that is produced by the compiler; it never works with ".c" file.
3.It uses commands for debugging.
4. It can directly jump to any particular memory location by using x command. X is a memory command jumping to the address of the variable and accessing entire memory of that variable address.
5. It supports command line program.
If we run the command gcc with an "-g" symbol it is executed with debugging symbols.
When a file is compiled with –g option a symbol table information is added to the executable file.
SYNTAX- gdb ./obj file
COMMANDS-

1. Breakpoint or b- when we need to stop program execution in between the process or program we use this command. To apply this command, we need a break function and break line numbers where the program pauses or stops execution (mainly at main function). **We can only use break on only 1 line or function of the program.**
2. Run or r.
3. Next or n- this will take you to the next line of the source code. It does not go inside the function call; it executes the function and shows the next line outside the function.
4. Step or s- this will also take you to the next line of the source code but, **it also steps into the function call and show the next line in the function.**
5. List or l- it lists the entire program.
6. Print or p- prints the values of the variable.

p/x variable_name- decimal to hexadecimal value.
p/t - decimal to binary. p/o - decimal to oct.

7. Info locals- provide the information about local variables.
8. q for quit.
9. Set -to set a variable.
10. x variable value- to access a memory location.

Printing ASCII values at particular address- or changing a string
x/s fetches all the data or char values from the base address until null value.
Now use the command set base_address= ”change” this change should be equal to the actual string used in the program.
This will change the characters to the changed characters.
Then we can print the changed string.

Memory allocation- after compiling the source code it is divided into segments as:
Text segment, data segment and bss.
All statements of all functions are converted into instructions and get stored in text segment of executable file.
Data segments contain the initialized global and static variables.
Block started by symbol (bss) contains un-initialized global and static variables. Their default value is 0.
Memory for local variables is allocated during the run time of the source code.
During execution 2 more segments are added these are:
Stack and heap segments.
Heap segment contains memory for dynamic variables or dynamic memory allocation.
When the main is evoked a stack frame is allocated to it and for every function called in the main function another frame is created inside the main stack.
Size file_name= type of memory allocated in the program except stack and heap because these are created after the run time.
Nm also shows all the variables and their memory type.

COMMAND LINE ARGUMENTS- provides input to the program while execution like scanf() or gets. These are the parameters that are passed to the program during execution time from **command line interface.**
There are 2 ways to store the string first is the character array and the second is the character pointer.

Char array	Char pointer
1. Data can be directly modified	Data cannot be directly modified
2. All the data is stored in the same place.	Data and the pointer are stored in distinct locations.
3. The data gets stored in the stack format in the memory hence easily changed or modified.	The data is stored in the "read only" part of the data segment of the memory and therefore it cannot be modified.

SYNTAX to change or modify the main function for command line interface-
Int main (int argc , char* argv[])
Argv is a pointer array which contains the pointers for different strings stored in the same char array. It is allocated on top of the stack segment.
Argc contains the number of parameters passed in the argv.
SYNTAX to run in gdb-

1. Gdb –args ./objfile name arguments (ex- Linux kernel)

Output – Linux kernel. (According to the source code)

2. Gdb ./filename then r arguments.

EX- to find a factorial without initializing the fact variable.

When working with GDB it is found that the fact variable was uninitialized having a garbage value and that was causing the wrong result. Now set the fact to 1 with command set fact=1.

DIFFERENT VALUES OF * -
A*b = multiplication.
*ptr = de-referencing a pointer.
Scanf(%*d)- don't read the value.

ERRORS from user-

1. Dereferencing a pointer which is already free.
2. using a null pointer.
3. Using an uninitialized pointer.
4. Writing loops or use more memory than the allocated for the variable
5. When the kernel fails to map a virtual address given by the user or any invalid address to the actual physical address of the variable, the processor raises the error to the kernel and the kernel will directly terminate the code leading to segmentation fault.

ex- *ptr=malloc(sizeof(int)*4);

For(i=0;i<6;i++)
Here we want the memory for 4 int values but malloc is allocating more than that which leads to error in the output or unwanted result in the output. This is because the standard c library allocates more memory than requested. **Gdb and gcc fail to trace memory violations.**

DISADANTAGES OF GDB-

1. Not effective in errors for heap memory space (for dynamic memory allocation).

It is because in dma malloc and calloc allocates memory from the pages in form of 2^n like 32 bits or 64 bits so when we ask for memory of 10 bits it will allocate 32 bits of memory hence malloc can expand the no. Of bits asked in the program.
Separate class of tools are used for tracking the bugs when they happen in heap segment.
Heap profiling tools and memory profiling tools are used for this purpose.
Electric fence library is used for memory profiling with command-

```
gcc –g filename.c -o objfile –lefence.
```

- By adding efence library to the program we overwrite the standard c library for dma.
- It is a memory debugger also called as memory profile library.
- It has its own implementation of malloc and calloc functions for dma.
- When programmer links electric fence with the source code compilation electric fence will have dominance over standard c memory functions (malloc and calloc)
- If memory error occurs efence triggers application program crash through segmentation fault.
- Efence can identify the errors of writing beyond upper bounded region (limit) of the memory allocation (**over run error**), if we try to write before the memory allocation (before the lower limit) it is known as **under run error**
- It can identify only over run or under run errors in a single compilation. for debugging for both the errors we use envt. Variables.

To check the under-run error, we use export EF_PROTECT_BELOW=1
to stop the under-run check, we use export EF_PROTECT_BELOW=0

SEGMENTATION FAULT-

When a program is compiled and executed memory segments are allocated for the program in run, our program is supposed to access the entire region of memory segment that is allocated to it (text, data, bss, stack or heap). If the program accesses the region beyond the allocated segment this leads to segmentation fault cause to application to crash or terminate.

VALGRIND TOOL-

- It is a stand-alone debugger that finds the error caused in the heap segment with the standard c libraries.
- It is also used as a memory profile tool
- When we use debug symbol (-g) with compilation and then use valgrind it shows the lines of the program.

TYPES OF MALFUNCTIONS IN PROGRAM-

ERROR- mistakes made by the programmer in writing the code.

DEFECT- mistakes detected by the tester during the testing of functionality of the code and whether the code is meeting all its tasks which it is meant to perform.

BUG- If the defect is accepted at the developing stage of the application provided by the testing team it is called a bug.

FAILURE- inability of the system software to execute and perform task functions. EX- breakdown in operating system, System hardware failure.

FAULT- It arises due to some conditions which lead to software failure. EX- page fault (inability of kernel to map the physical address to the provided virtual address).

ARM CROSS COMPILATION USING MAKEFILE-

Compilers are of 2 types-

Native compiler- compile on the existing native CPU architecture (INTEL).

Cross compiler- compilation and execution is done on different machines.

In cross compilation we will write the code on intel architecture and execute on the ARM architecture for this cross compiler is installed in the native intel architecture. Cross compilation is a process of generating executable files for other architectures.

Command- sudo apt install gcc-arm-linux-gnueabi (executable application binary interface)

LINUX KERNEL ARCHITECTURE-

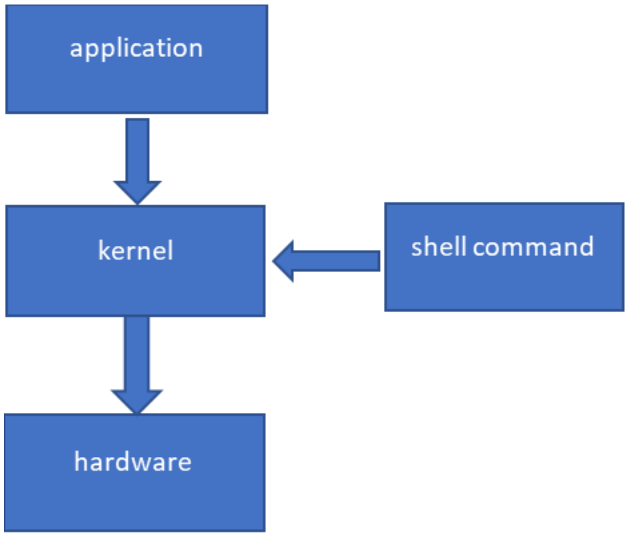
When the operating system is loaded on the RAM it gets divided into 2 parts- user space and kernel space. The RAM looks like-

User space- Applications which are ready to use by the user
Kernel space- Set of programs that interact with the hardware system

Core of the operating system is called "Kernel" which is software that communicates with the hardware which is the first program loaded in RAM and runs until the system shut down. Kernel is quiet different from hardware.

LAYERED ARCHITECTURE-

When an application wants to communicate with the hardware it cannot do this directly so the kernel on behalf of the application makes the request to the hardware. Kernel protects the hardware from corruption by the application. When user wants to interact with kernel it is done through the shell commands entered by the user on the terminal. Shell program acts as a bridge between the user and the kernel. Shel passes the command entered by the user to the kernel and then the kernel follows the command and talks with the appropriate hardware machine.



KERNEL SERVICES-

1. File system service
2. Input/output device services
3. Multiprocessing services
4. Multithreading services
5. Memory mapping and allocation services.
6. Inter process communication
7. Process synchronization
8. Process scheduling
9. Signal management
10. Network programming
11. Driver's programming

KERNEL SPACE IS AGAIN DIVIDED INTO 2 PARTS FILE SERVICES AND Driver's space

FILE SERVICES
DRIVER'S SPACE

Kernel space is divided into following services in the file space inside the kernel space-

Virtual file system	Process management (includes thread management)	Memory management	Inter process communication	signals	network	schedular	synchronization

1.KERNEL FILE SYSTEM SERVICES-

By using this a programmer can write a program to perform-

1. to open an existing file.
2. to perform to create new file.
3. to read data sequentially from the file.
4. to write data sequentially to the file.
5. to modify the data.
6. To change the properties of the file.
7. To close the file.

We need file system in our OS to read the data from the file system of the devices connected to our OS. Linux supports 50+ file systems. If our OS does not support this functionality, it cannot read the data from the file system of the connected devices.
Path to check file systems supported by the Linux-
/lib/modules/kernel version(5.4.0-144-generic)/kernel/fs

2.INPUT/OUTPUT DEVICE SERVICES-

Linux operating systems do not understand device hardware structures so the developers implemented files for the devices called as device file which is understood by the Linux and perform the service.
A programmer can write a program to create, to open, to read/write, and close a device file.

3.MULTIPROCESS MANAGEMENT SERVICES-

At design stage in order to reduce the complexity of the problem the problem is divided into smaller tasks and to implement these tasks process management creates the processes, schedule the processes, executes the processes and then terminate the processes. These processes are executed or located at different memory locations.

4.MULTITHREADING SERVICES-

It is similar to the process management in this instead to processes we have multiple threads in which the tasks are divided into.

5.INTER PROCESS COMMUNICATION-

When 2 processes need to communicate with each other in the same machine for data requirements it calls for inter process communication which is of following types-

1. Pipe
2. FIFO
3. Message queue
4. Shared memory
5. Semaphore

6.NETWORK APPLICATION SERVICES-

When a process on one machine wants to communicate with the process of another machine, we write a program on over the network communication. We need IP addresses and port addresses which we manage with socket.

Socket requires-

1. Protocols
2. Source IP address
3. Destination IP address
4. Source port address
5. Destination port address

7.SIGNAL MANAGEMENT-

Signals are software interrupts handled by the processes.
Ex- when we have an infinite loop in the output we press ctrl+C which is a signal to the kernel. Kernel passes this signal to the process running which then reads the signal and performs the task to terminate the process.

8.PROCESS SYNCHRONIZATION-

When we want to perform multiple processes on the same hardware or device we need to make sure that these processes are completed in such a way that we get the expected outcome from them. To ensure this synchronization techniques are used. In this technique we have to lock the second process and when the first process is completed, we can unlock the second process to get performed.
There are following types-
Semaphore
Spin lock

9.SCHEDULER-

COMPLETE FAIR SCHEDULER
ROUND ROBIN SCHEDULER

10.DRIVER-

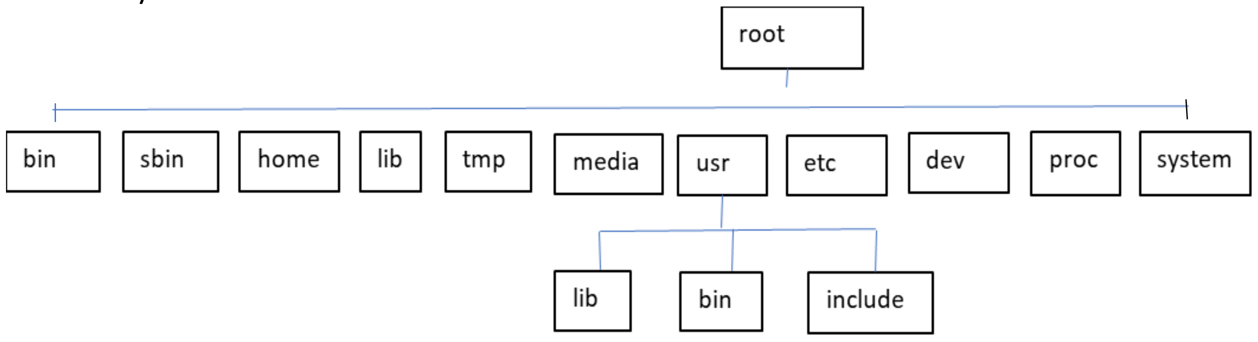
A piece of software which talks to the hardware device. It resides in the kernel space of the RAM. It has 2 interfaces, one with the application (specific to the operating system) and other with the hardware (specific to the hardware).

TYPES OF KERNEL-

MONOLITHIC	MICROLITHIC
drivers **network** **File service** **Core (process + memory management)**	**driver** | **File service** | **network** **Core (process + memory)**
single address location	Different address location of core and the services
All the services and the core of the kernel (process management + memory management) are built in the same package	Services are built separately outside the core
OS size is large	OS size is less
Debugging is difficult	Debugging is simple
OS has easy design	OS has complex design
If one of the services gets corrupted the entire Kernel will be corrupted	Corruption of one of the services will not affect the core kernel
Maintenance uses more resources	Maintenance uses less resources
Execution speed is faster	Execution speed is low.
Ex- windows, Unix, Linux.	Ex- Mac OS, Symbian OS

LINUX FILE SYSTEM-

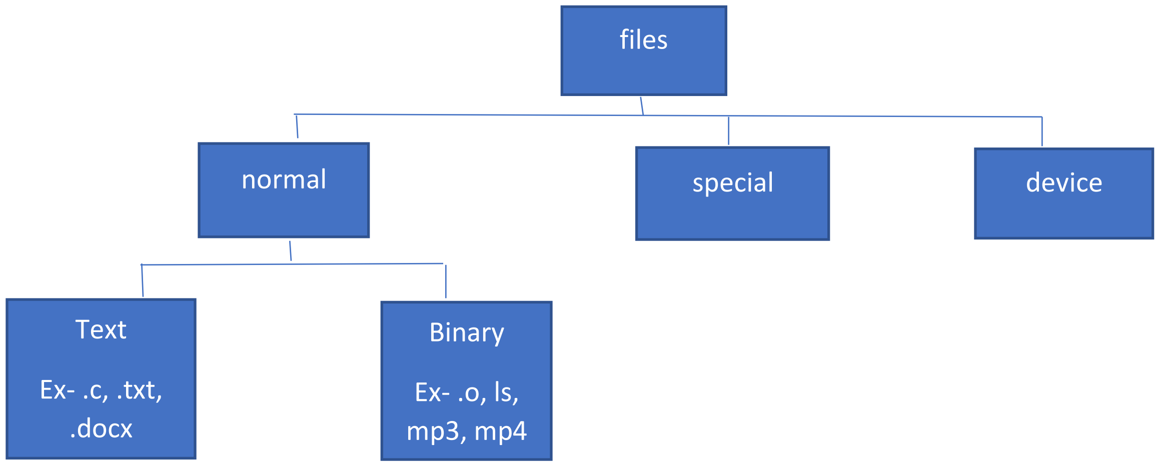
ROOT FILE SYSTEM- It manages the memory and stores the files and directory safely and directly into the memory. When the kernel is boot loaded all the 50+ file systems supported by Linux, get attached to the root file system. Only when the file systems are attached to the root file system, then they are accessible to the OS.
All the files and directories are stored in a structed and safe manner in the memory such that it does not waste any memory.
Root file system is laid out as a hierarchical tree structure format.



1. Bin contains low level system utilities that contain software interfaces that interact with hardware.
2. Sbin contains super system services used by administrator
3. tmp is a temporary folder which contains temporary files.
4. Every user has a personal file space which is known as home folder. There is a directory present in the home which contains the login info. When the user login in the system they directly get into the home folder.
5. lib contains library files for low level (hardware) services. Ex- at kernel boot up time libraries are used from this lib folder.
6. media folder is called as extended root partition.
7. lib in the user folder contains all the library functions used by the user, bin folder contains high level(applications) services, and include contains all the header files used by the user.
8. etc contains all the Linux configuration information like login id and password.
9. dev folder contains all Linux device files
10. proc is a logical file system (not permanent) which gets created at the time of kernel boot up time. It creates lots of kernel data structures shown as files to the user. Command- cat /proc/cpuinfo shows all the processor information. Cat proc/meminfo shows the memory information. During development if we want to check the interrupt information proc will show the snapshot of the interrupts by- cat /proc/interrupts. The device info is divided into 2 parts based on the bandwidth of the data transferred from the device these are- character devices (character transfer or single transfer data rate) and block devices(high transfer rate)
11. The information related to all kinds of devices connected to the processor is given by system file system.

BASIC I/O CALLS-

operates on all kinds of file categorized in following types-



Application using device files invokes the specific driver in the kernel space and driver on behalf of application communicates with the respective device.

Types of blocks in hard disk-

Boot block- contains boot loading information.

Super block- contains all the key information, statistic information (no. of directories, no. of files, no. of file systems).

Inode block- each file has 1 Inode structure. It maintains complete information of the file like-name, type, size, permission and pointers to data blocks and pointer to file operation structures.

Data blocks- contains the data of the files present in the system which is pointed by the pointer from the inode block.

Linux supports some basic I/O calls such as-

1. Open ()- takes three arguments as: open (const char* pathname, int flag, mod). Open returns some integers based on the tasks if the file is opened it returns some non-negative integer value.

- #include<fcntl.h>
- Int open (const char* pathname, int flags,mode_t modes);

Argument1- complete pathname i.e., file name

Argument2-operation to be performed.

O_RDONLY

O_WRONLY

O_RDWR

Special commands- should be used with | operator with the regular read write flag.

O_TRUNC

O_APPEND

O_CREAT

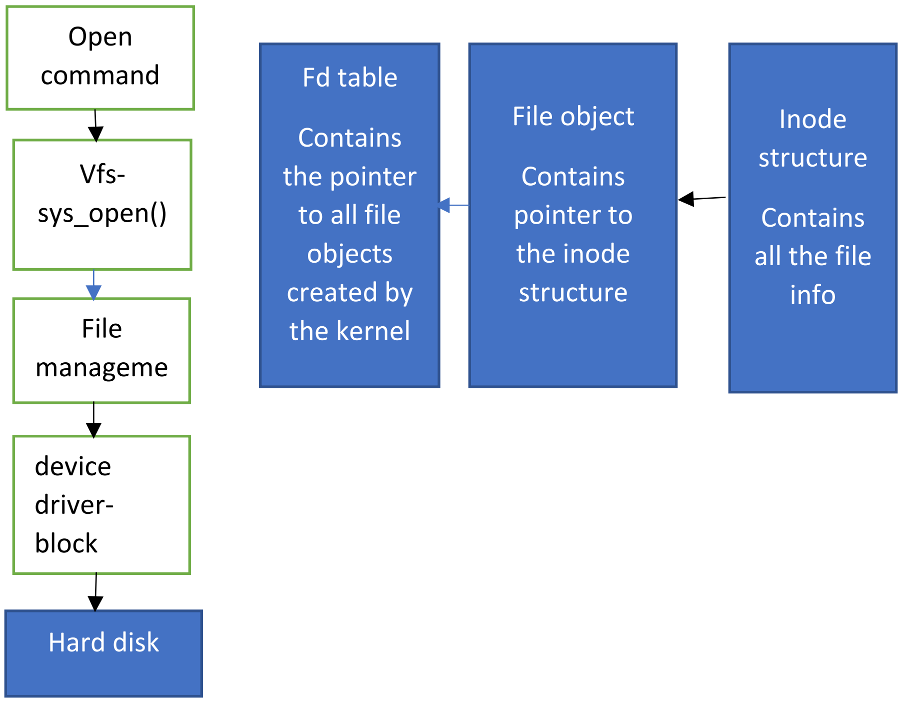
O_NONBLOCK

Argument3- modes talks about the permissions.

EX-

Int x = open("/desktop/foldername/file.txt",O_RDONLY,0666)

Linux can open 1024 files from a single open.c program.the process to open a file is-


Open command
Vfs-sys_open()
File manageme
device driver-block
Hard disk
Fd table
Contains the pointer to all file objects created by the kernel
File object
Contains pointer to the inode structure
Inode structure
Contains all the file info

When we type the command to open a file in linux, the kernel copies the argument to sys_open() function in the vfs service of the user space. Which calls the file management to look for the file. The file management looks in the device driver (block type) which contains the hard disk driver. This hard disk driver talks with the hard disk present in the kernel space of RAM. When the file is located in the hard disk the kernel copies the inode structure of the file in a new inode structure created by the kernel. The address to this inode structure is stored in the file object structure through a pointer and the address of this file object is stored in the fd table. The index of the fd table is returned by the open function therefore it has return type as an integer.

All the files opened with this call has their own inode structure and file object and all these file object addresses are stored one by one in a single fd table.

2. **Create ()-** also called as API (application programming interfaces) creates a new file. If fails to create it returns -1, otherwise returns int value.
   #include<fcntl.h>
   Int create(const char* pathname, mode_t modes);
3. **Read ()-** used to read data sequentially from the file. Syntax-

Ssize_t read (int fd, void* ptr, size_t nbytes);

n shows the no. of bytes to be read from the file and dump at the address pointed by the ptr pointer (it's a random address to copy the data at). **Read returns the no. of bytes successfully read by it, in case of empty file it will return 0.**

VALIDATE READ REQUEST-

Most of the time read returns less no. of bytes than the actual request. Because if the file size is less than the requested size in the read call, it will return only the bytes equal to the file size. Therefore, we have to validate the request in the call.

File opened with 'write only' flag but we are making the read call request then it will return -1 (error or failure). If the void pointer points to an address outside the program memory region (outside text, data, bss, stack or heap), it will again return failure.

4. **Write ()-** used to write data sequentially to the file. Syntax-

```
#include<unistd.h>

ssize_t write (int fd, void* ptr, size_t bytes);
```

in case of success it returns the no. of bytes written in the file, and in case of failure returns -1

VALIDATE THE WRITE REQUEST-

When the storage of the system is less than the request made to write in the system we can not write data in this context. Therefore, the request should be accordingly.

File opened in the 'read only' flag and then write request called will return failure, and if the pointer points to address outside the program memory segments this will also return failure.

5. **Close ()-** close the existing file descriptor. Returns 0 after success. Returns -1 for failure.
   Int close(fd);

   CURRENT FILE OFFSET POSITION–
   Every opened file is associated with current file offset value which is a non - negative integer number. It is a count or measurement that measures from the beginning of the file. It gives the location at which we are present in the file. Generally every read and write operation begins at current file offset position. By default, it is set at 0 initially (except- in case of append). Current file offset causes to be increased by no. of bytes successfully read from a file or successfully written to the file.
6. L SEEK- to move the current offset position or repositioning. Interpretention of lseek operation depends on 3rd argument 'whence'. Syntax-
   lseek (int fd, off_t nbytes, int whence).

Here off_t has internally int return type, which returns the current offset position.

At any given time whence argument has one of the following meaning-

SEEK_SET

SEEK_CUR

SEEK_END

STANDARD FD-

FD table contains all opened file information.

Each entry in the fd table it has the base address of the file object, and that file object points to the inode structure of the file.

These fds are used in the basic i/o calls- read, write, and lseek.

C library functions internally invokes basic I/O calls. (Printf internally calls write call and scanf calls read).

Following are the standard FDs after these file descriptors we get the FDs of the opened files.

0- Standard input device's file object that belongs to the inode object base address
1- Standard output device->base address of file object->address of inode object of output device (terminal or screen).
2- Standard error device.

Different ways to read or write without printf or scanf library function-

Write(1,"data"or buf address,size of data). Here fd= 1 because it is used for standard output file.

Objdump- divide executable file into sections and display all the sections, this helps in looking for the data stored in different memory segments (heap, stack, bss, text, data).

Command- objdump -D executable name or objdump -s objfile to see the data stored in different segments.

We can also see these sections by size or nm command

REDIRECTING THE OUTPUT-

Printf is used to see the code flow in large application program or for redirecting the output.

It is done with redirector operator or with dup2 () system call.

Redirector operator- ./outut > newfile

Dup system call- duplicates the file descriptor. Int dup (int fd).

Uses- 1. Operating system will allocate least available fd from the fd table.

2. Dup2 system call duplicates the fd at new fd value given by the user.
   Int dup2 (int fd1, int fd2)

When we call dup function the file that was already opened all of its kernel objects (fd table entry, file obj and inode), gets deleted and the kernel object of the new file gets copied in those deleted kernel objects.

From the user space accessing inode object present in kernel space-

Ls -l command will give all the information of the file.

Stat() and fstat() these system calls also shows inode information of the file.

STAT CALL-

Each and every file whether opened or an existing file will maintain a structure called as "stat structure" and that maintains 'ls -l' kind of information for a file.

stat structure contains-

- permission
- size
- inode number
- Uid
- Gid
- Time of creation
- Time of modification
- Block size
- No. of blocks used
- Device number

Syntax-

Stat(const char* pathname, struct stat *buf);

For fstat it is-

Fd= open();

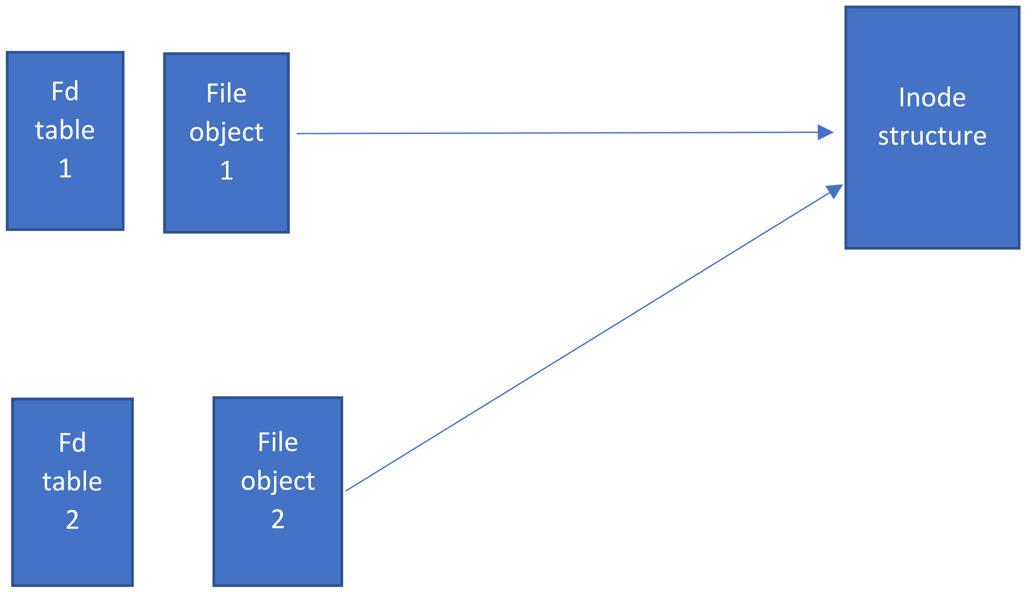
Fstat(int fd, struct stat* buf);

These calls return structure of information of the file existing or opening.

The moment stat functions gets executed it goes to the file name provided by the user and then goes to the file's respective File structure fetching a structure of information and copying in the second argument of the system call which is the address pointed by the pointer supplied by user.

Fstat is another variant of stat call by it only works for opened file.

Linux os provides a facility to share opened files among multiple programs (processes) where a common inode object is shared among multiple programs.



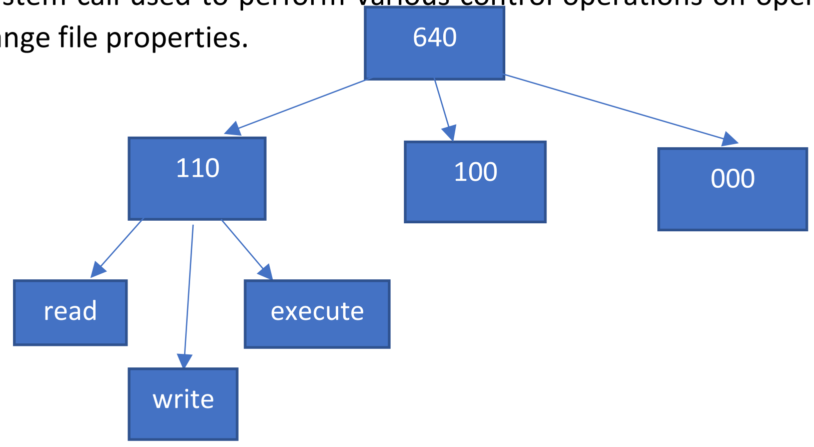
PERMISSIONS-

It is an octal digit number where each digit talks about permission either by owner, user in the group or other-

First digit	Second digit	Third digit
Owner	User in the same group	others

Every number is considered by its binary equivalent like-

fcntl is a system call used to perform various control operations on opened files which can be used to change file properties.



LIBRARY-

In a programming language we may require a block of code to used again and again, in order to avoid rewriting of the same code several times, the code is created into package which is called as library. And the application uses this package instead of codes several times. **It is not an executable file i.e. it does not get loaded anywhere. It is a group of precompiled objects.**

Dynamic libraries- which are used at the run time. These are added to the program at the time of execution.

- It is a programming concept with special functionalities linked to the executable during program execution time (run time).
- Also called as shared libraries.
- There are also known as run time libraries.
- Every program can use dynamic libraries in order to avoid re-writing of code several times.
- We can create dynamic libraries and can store in non-standard location of the file system, for that we need to use an environmental variable – LD_LIBRARY_PATH
- Working-

These are linked during the execution of final executable file

But actual linking is taking place when the dynamic library is copied to the program's main memory segment.

Steps to create library-

Gcc -c add.c fpic -> creates relocatable obj file

Gcc *.o -shared -o lib_dynamic.so

Gcc -c five.c -o five.o

Gcc five.o -o five -L. lib_dynamic.so

Export LD_LIBRARY_PATH=:

Pwd

Export LD-LIBRARY_PATH=:pathname from pwd

Standard location of libraries-

1. /usr/lib
2. /lib
3. /usr/local/lib

Fpic flag- position independent code, to create obj files directly.

Creating instructions as position independent code (add.c). If a program five.c is executing and requires dynamic library “.so” file then addresses of library files should be shown in such a way that dynamic library “.so ” file looks like a part of main memory of the program when library is appended to its executable.

For ex- another running program may require the same dynamic library then again library should be shown as a part of program’s main memory.

Static libraries-

- Which are used at the compilation time.
- At the time of linking, static libraries are added to the object file through static linking. Links the precompiled obj file of the library with a target (source) application file and generates final executable file.
- It is a precompiled obj file that contains symbols (functions, variables, macros), these symbols are used by the program to execute and perform the task.
- These are created by a tool known as archive tool (ar) with rcs flag and name of library with .a extension used for libraries where-

R= replace with existing library or update it, which is optional if library is being created for the first time.

- C= create which is mandatory
- S= symbols
- Command to add static library to the main function with static linker-
  Gcc filename.c -o filename -L. (tells the linker that our library is in the same directory as the obj file) **lib_static.a**
- Now static linker links lib_static.a library with target file and creates final executable file.

STATIC CODE ANALYSIS- is the process of detecting errors, and bugs in the program source code before the program is run. Analysis is done on set of code by using some coding standards. These kind of analysis is helpful to identify loop holes and weakness in the program that might be harmful for the program. Analysis and examination is done on stationary code therefore named as static code analysis.

Splint-

It is done with the help of “splint” tool as-

Splint code.c

This will show the warnings for our code to suggest some changes that might help in better execution or the program like is x=10 instead of x==10 or missing ‘break;’ in switch case. These warnings are given by the gcc compilation.

Splint is a static code analysis tool used to identify programming errors and bugs in source code like- syntax errors, typedef errors, usage of undeclared variables, usage of undefined functions, etc. for a programmer its always good to have an error free program before it could get executed. Splint tool is improving program performance.

CLANG-

It is a C and C++ compiler compiled (developed) with C++. It uses a low level virtual machine which contains a compiling technique (internally a library) which builds a binary machine code. This binary code is used to create the compilation framework. This framework provides the structure or foundation to create the compiler or how the compiler works.

Clang is very fast in execution as compared to gcc compiler.

It creates extremely clear diagnostic messages and warnings as compared to gcc.

Used as- clang -Wall filename.c -o objname (-Wall is a flag which includes all the warnings)

clang	Gcc

DYNAMIC ANALYSIS- It is done on piece of software that is executed or in execution

Gcov tool- gcc’s coverage tool, is a program coverage tool. It is an open source software used along with gcc. It determines how much of the program is not executed, untested and also number of times a particular line is executed.

Command-

Gcc -fprofile-arcs -ftest-coverage filename.c -o objfile

Then after ./objfile apply gcov objfile.gcda

- Compiling the source code with -fprofile-arcs and -ftest-coverage
- -testcoverage flag causes gcc to add code coverage and statistic coverage information to binary executable.
- Compiling source code with special flags adds extra information to the binary and records the number of times the lines have been executed.

- Running the executable file -fprofile-arcs flag causes to create ".gcda file" output profile file.
- Performs line coverage and function coverage.

LCOV- graphical front end tool for gcov. Lcov graphically represents gcc's program coverage tool. It generates html files for the profiling output generated by gcov tool.

Command-

Lcov --capture --directory . --output-file filename.info this will capture all the information generated by gcov from current directory and generate a ".info" file. Then we use genhtml filename.info –capture-directory . -o objfile to generate the html file from the .info file

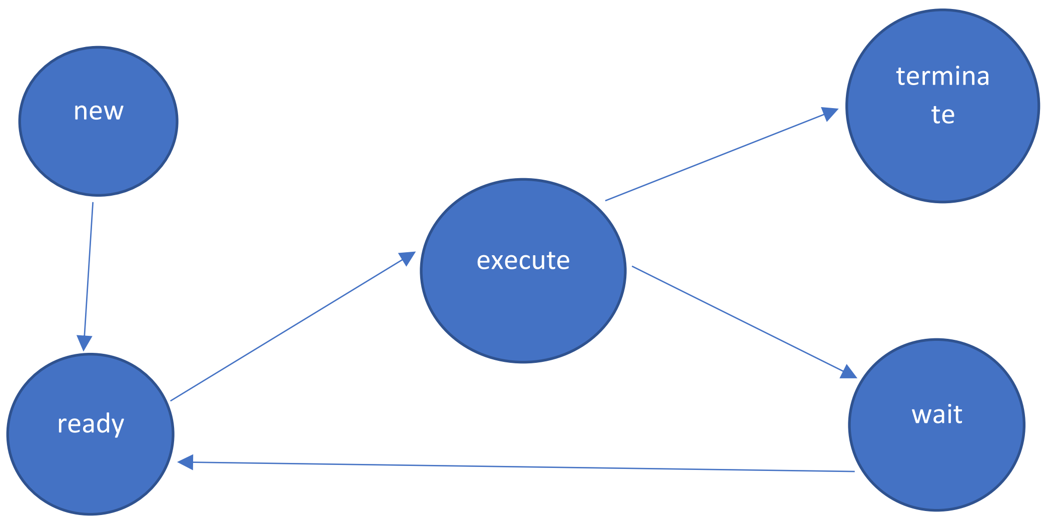
PROCESS MANAGEMENT-

A program which is in execution is called as process. Once the program gets cpu slice time it becomes a process.

Commands to see the process-

Ps -af -> shows the processes executed from the terminals. Here ps= process state.

Process states are as-



When a process is created it goes into the new state. After creation it goes into ready where it is prepared for the execution, then it moves to the execution state. If encountered any sleep or delay calls (like in scanf the process is in wait state until data is entered) moves into wait state.

Otherwise directly moves into termination state. If a process is in wait state, after the delay time it again goes to ready state looks for the memory and then gets executed.

PROCESS QUEUES-

Ready queue- it has a linked list of all the processes that are ready for execution. These queues are maintained as per the cores or number of CPU (dual, quad or octa) in the system. The process to be executed by the CPU from the ready queue is decided on the basis of priority by the schedular. Schedular picks the best (priority) process and puts it into execution mode.

Wait queue- it is a linked list of processes which are suspended due to blocking or delay calls (scanf, sleep of file). Number of wait queues are per resources (memory, keyboard, mouse, printer etc) each resource has its own wait queue.

PROCESS IDENTIFIER (pid)- when a process is created OS allocates pids to the processes. These are unique, no two processes will have same pid. When a process gets terminated the same pid is used for other process with delay time out but not immediately to avoid confusion. All the process are stored in the user space except the core kernel process which gets started at the time of kernel boot up with pid=0. Core kernel process is responsible for creating init process in the user space with pid=1. This init process contains or starts all the other processes in the user space and acts as the parent to all the other processes.

We can see all the running processes with ps -elf command.

Pstree shows the processes in tree format which shows the process and its parent process.

PROCESS CONTROL BLOCK-

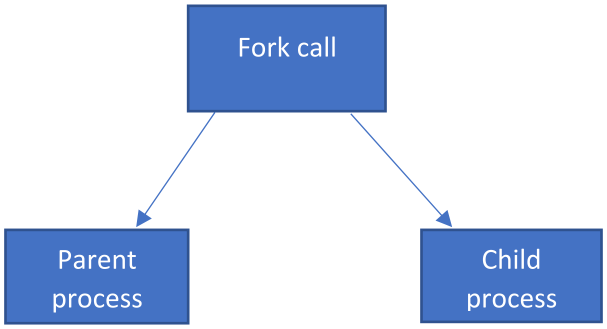
For each process a process control block is created in the kernel space which contains all the information about the process in the running (pid, ppid (parent pid), process state, pointer pointing at the process queue (ready or wait), CPU information (register, counter value, general purpose registers, fd table information, page table, etc.)). This PCB is a structure maintained for every process of the user space. All the memory segments are stored in the process address space in user space which is linked to the pcb in kernel space. Each process has its own isolated memory.

PROCESS CREATION-

Pid_t fork()- this system call creates new process.

Linux OS provides a system call called as fork() for creation of new processes from current task. For implementing concurrent applications (multitasking) software. All modern OS supports multitasking in various ways. When fork is called it creates a new process and executes the code written after it, and when this process is-

completed the original process gets executed.



- The pid for new process (process after fork call called as child process) will be different and has the parent pid = pid of original process (parent process).
- Fork system call creates 2^n no. of processes where n is number of times fork is called.
- Fork system call gets executed once but returns twice, one time for the parent process and next time for the child process.
- It returns child process's pid to the parent process and returns 0 to the child process.
- As the parent process should be able to track its child process therefore it returns child process's pid to the parent process and there's nothing to keep track for the child process therefore it returns 0.

Three states of file:

1. Modified
2. Staged
3. Committed

Modified	staged	committed
• Files are in the working directory.	• By using command add file will be in staging area. • As work progress files are in stage area. • Files are marked to go into commit.	• Commit will move the files from staging are permanently moved to local repository. • Local repository will maintain all the commit history as linked list structure. • The last commit repository can also be referenced by the repository.

Process management:
pid_t fork();
system will create a process.
main()

```
{
pf(“one”);
fork();
pf(“two”);
}
```
output:
one two two
note: fork called once
requirement: we want the parent process to execute one block of code and child to execute another block of code. for that we need two thing return values of fork calls and use conditional statements.
code:

parent process	child process
main()	main()
{	{
int pid;	int pid;
pid=fork();	pid=fork();
if(pid) //fork will be return child’s pid to parent	if(pid) //fork will be return 0 to child
{ //therefore if block will be executed	{
pf(“1”);	pf(“1”);
pf(“2”);	pf(“2”);
}	}
else	else //therefore else part will be executed
{	{
pf(“3”);	pf(“3”);
pf(“4”);	pf(“4”);
}	}
}	}
ouput: 1 2	output: 3 4

note: schedular will decide which process to execute first depending on the load on the system.
vfork()

process termination:

1. main() à return 0; //normal
2. exit(0); //normal
3. we can send signal and terminates a process //abnormal
4. ctrl + c //abnormal

- Whenever a process terminates, the kernel sends a exit status of terminated process to parent process.
- parent process has to fetch read the exit status of terminated child process.
- when process doesn’t terminate properly and its resources removed from user space that is called zombie process.

- if parent process is busy in doing some other job and failed to fetch the exit code of the child process, then child process entered into the state called as zombie and is called as zombie process.
- signals are used to terminate the zombie process.
- a parent process terminates without waiting for the child process execution it is called as orphan process.
- init process whose pid=1 will act as a parent process to orphan process.

types of processes:

1. **zombie process: -**
   - when process doesn't terminate properly and its resources removed from user space that is called zombie process.
   - if parent process is busy in doing some other job and failed to fetch the exit code of the child process, then child process entered into the state called as zombie and is called as zombie process.
2. **orphan process: -**
   - a parent process terminates without taking care for the child process it is called as orphan process.
   - init process will act as a parent process to orphan process.

wait() system call: -

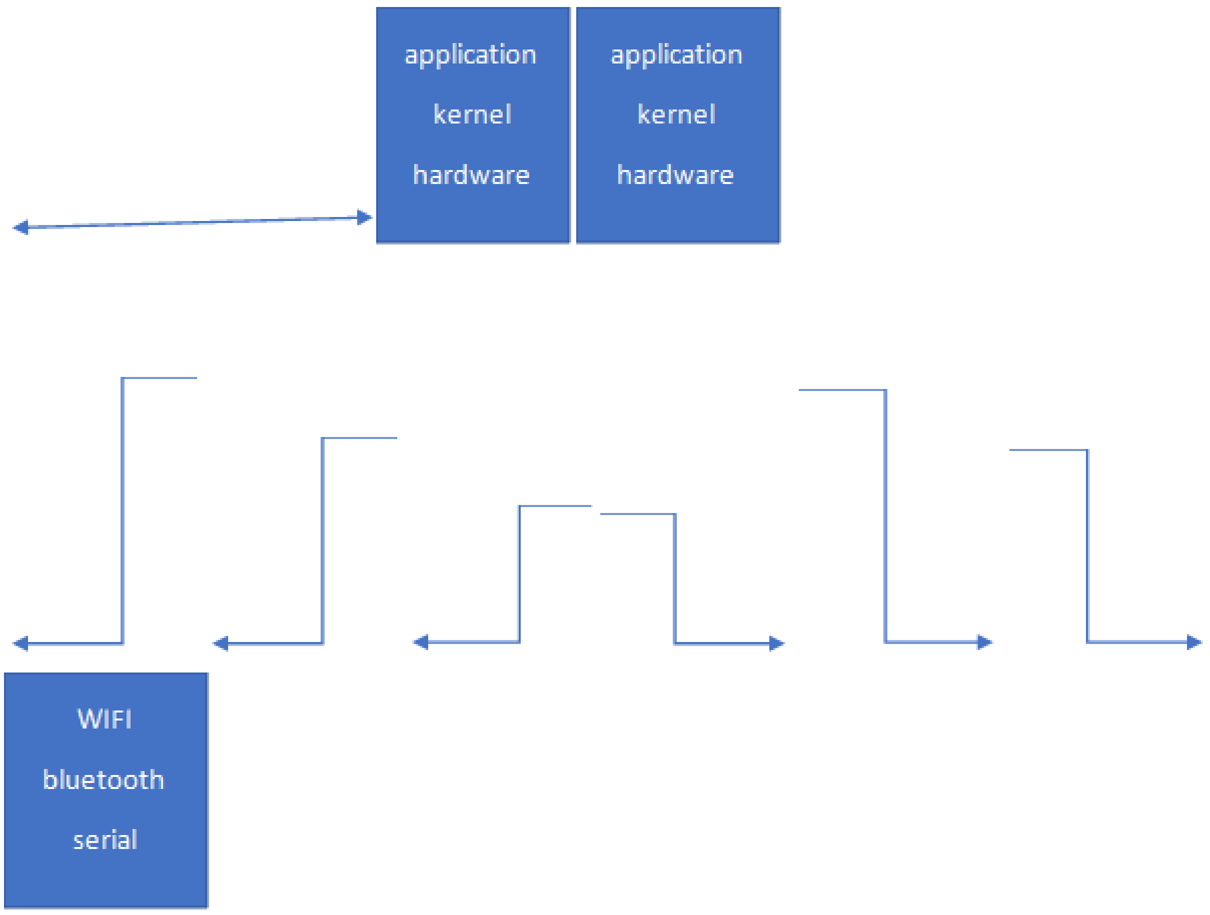
- wait system call will block the parent process until the termination of child process. once child process terminates parent process resumes.
- here wait(&stat) will return the exit code to 2nd byte of stat and WEXITSTATUS(stat) will fetch the that byte.
- wait() has 3 behaviors :
  1. When child processes are in execution parent process is BLOCKED.
  2. When child terminates wait() call collect exit code of child process.
  3. wait() call returns error when there are no child processes are executing.
- wait() itself return the pid of child process.
- Another variant of this is waitpid(pid of the child process we want to terminate first, &status, null flag(0));

uses of fork() system call:

1. fork system call is widely used in implementing concurrent application software.
2. fork system call is used to implement terminal applications.
   - terminal application is internally a shell program.
     - bash
     - ssh (secure shell)
     - c shell
3. fork system call used to implement client and server application.

Disadvantages-

When a fork is called a new process is created and for each process a new process control block is created which has many resources and information initialized and maintained by kernel. This is very heavy process for the kernel.



terminal or shell Application:

- To implement terminal or shell application we need fork() + Exec() system calls. execl("name of function","ls","-l",NULL)
- The moment process calls exec system call the current programs memory segments are replaces by a brand-new program's memory segmnets from the hard disk.
- no new process generated or no new pid generated, the only logic is current program is replaces by new program and also executing the new program.
- exec() returns value -1 only on failure.
- the program terminates from the program that is loaded by exec.

The program with exec() call have the memory segments but when exec executes the memory segments which is holding the program that will be replaced by the program that is provided in exec and its arguments.

execl("name of function","ls","-l",NULL) //takes arguments as command line arguments here all are considered as command line arguments for the new program. in case of ls -l NULL will be considered as arguments and their addressed will be stored in argv[]. NULL character is mandatory in exec function.

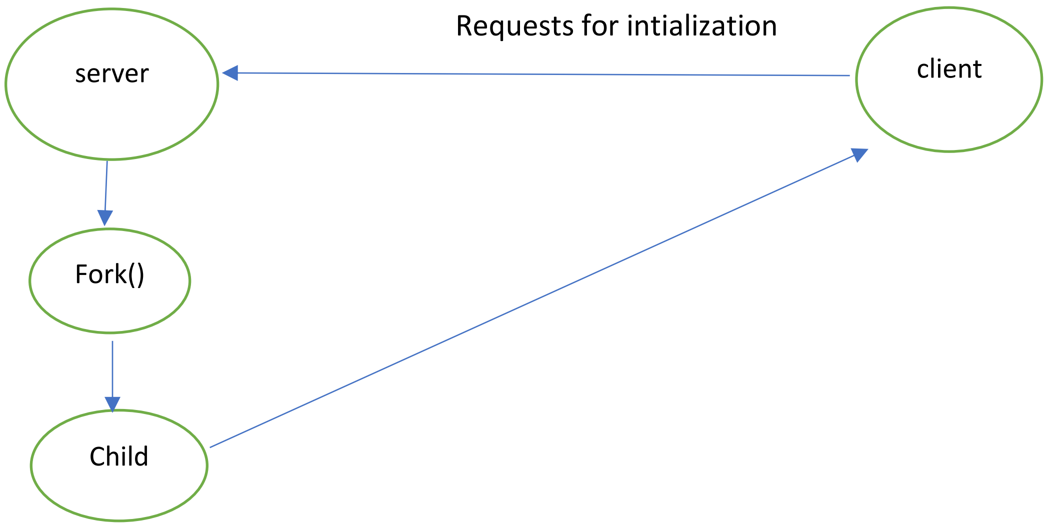
execv:

execv("/bin/ls",args); //takes only two arguments

execlp("name"," parent name",0)- it does not require whole pathname, it automatically searches for the path in which the file or program is located and loads it into the parent process.

SHELL PROCESS/APPLICATION- the bash internally does the following processes to execute-

1. Calls fork and starts a new process
2. Calls execl and executes new program
3. Wait for the termination of child and parent process



Each time a client requests the server for connection, server invokes fork and create a child process to handle the client. For every fork call a new pcb structure with lots of information is created which takes a lot of resources from the kernel. Therefore, threads are used which have less information or uses less resources.

MULTITHREADED PROCESS-

It is a process which creates multiple threads. Each thread can run separately a different task on different processors at same time.

THREAD- it is a parallel context of execution (set of instruction that is executing).

- Threads are called as light weight processes. Each OS has its own design of thread modelling. Linux uses 1 to 1 thread model.
- Creation of threads maintain threads and destroying threads is less resourceful than process management.

  No inter process communication is needed as the communication between all threads and their process is by default.

- Linux threads do not occupy their own memory segments, they share the process memory (data, text, bss and heap not the stack). Each thread maintains its own stack.
- Linux uses an API called as clone() which creates the threads. This clone is native to only Linux OS. These threads created will generate an executable file which will only work on Linux and not on other OS as it is native to only Linux and the executable is not portable.
- To make other OS understand and make the executable portable we can use another API called as portable OS interface (POSIX). IEEE standard organization defined set of common APIs that are implemented across POSIX compatible OS (windows, Linux etc.) to provide portable software.

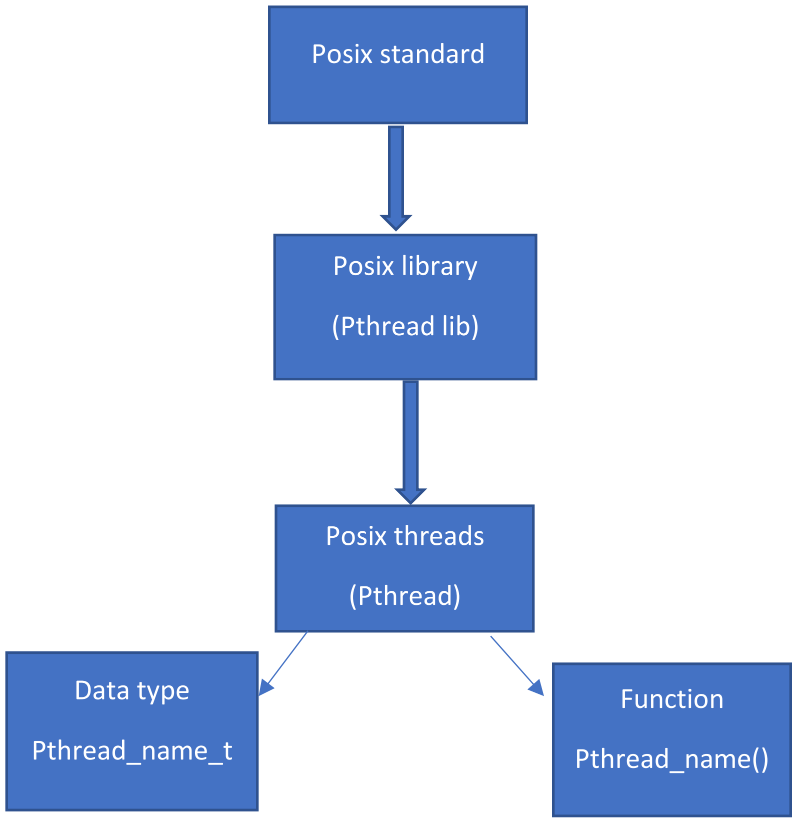
Thread has following information-

1. Maintains own stack region.
2. Own scheduling policy and scheduling priority.
3. Own register
4. Own signals (signal block/signal mask).
5. Own thread specific data.

Threads do not have their own memory segments, they share the process memory segment and creates their own stack region on top of process stack region hence they are considered as light weighted and uses less resources from the kernel.

Since the memory segments are shared therefore the global data saved in the data segment can be accessed and changed by any thread and this change will be reflected in the process and all the threads sharing the data segment.

POSIX-



Posix standard contains a set of functions and instructions in the posix library which creates the posix thread also known as pthread. This pthreads contains a data type always starts with pthread, and functions also starting from pthread. Each pthread is identified by a unique id and pthread_t is the data type that denotes the thread id. It is also called as thread object. Each pthread contains its own attributes or properties which contain how a thread should behave once created. This is denoted by one more data type that is- pthread_attr_t.

THREAD CREATION-

Include the header file for pthread #include<pthread.h>

Int pthread_create(pthread_t *ptr1, pthread_attr_t *ptr2, void (*start_routine)(void), void *arg);

For compilation include gcc name.c -o name -lpthread

The first argument gives address to the thread created by the function (id of thread), the second arguments points to the attributes of the thread, the third argument is a function pointer which takes no argument and return nothings and points to a function which contains the task for the thread and the fourth argument is a pointer which contains the argument passed to the function that is to be performed by the third argument.

If a thread is successfully created it return 0 else returns error codes such as- EAGAIN, EPERM.

When we don't have enough resources and wants to create a thread then it returns EAGAIN stating that the system doesn't have enough resources. When system doesn't have required permissions it returns EPERM.

Pthread_join(tid,NULL)- joins the thread passed as the argument to the main function and wait for the termination of the specified thread.

THREAD TERMINATION:

- when a thread returns from task function.
- When the process terminates, thread also terminates.

pthread_cancel()

- API called as pthread_cancel(thread id) that terminates the thread.
- pthread_cancel() is a cancelation request to other threads used to terminate other threads of the same process.
- pthread_cancel terminates whose thread id is metioned as argument.
- can also terminate calling thread.
- on success returns 0 and on failure -1.

pthread_self()

1. returns calling thread id.

pthread_exit()

pthread_exit(exit code) will have exit status.

pthread join waiting for tid to terminate.

when it is called from a process pthread_exit going to wait for execution of all the threads.

and another one collects the exit code of the thread.

/proc/sys/kernel/threads-max will show the maximum number of threads

INITIALIZATION ROUTINE:

These are routines that execute only once.

requirement: using pthread, implement initialization routines.

pthread_once()

This function must be used in combination with once control variable of type pthread_once_t data type.

1. pthread_once_t once_control;
2. there is constant macro called as PTHREAD_ONCE_INIT and initialized the variable with this macro.

syntax:

pthread_once(&once_control, <address of init function>,);

After execution of this function value of once_control value will be change.

THREAD ATTRIBUTES:

pthread attribute category:

1. thread stack management: attr function related to stack
2. thread synchronization:
3. thread scheduling policy and priority requirement: gaming software with 20 threads. with stack size of 15 MB.

- by default, pthread library allocate 8 MB to each thread for stack.

- if we want 15 MB stack size for each thread.
- if without modifying the attributes if gaming application is implemented stack overflow may occur and application may crash.
- change stack size 8 MB to 15 MB.

attributes talk about behavior once thread is created that how thread should be executed.**procedure:**

1. declare a variable of type pthread_attr_t myattr;
2. initialize attribute variable using pthread_attr_init(<addgress of myattr> &myattr);
    1. syntax: pthread_attr_init(pthread_attr_t *ptr);
3. call appropriate attribute functions
    1. stack
    2. synchronization
    3. policy and priorities
4. create pthread using our attribute obj.
    1. pthread_create(&tid,&myattr,<task fun>,argument)
5. destroy the attribute object after task is done. pthread_atr_destroy(&myattr)

2 threads are pointing to same stack address but pthread rule says each thread should have separate stack address. before a second thread created using the same attribute object destroy attribute object using pthread_attr_destroy().

Thread attributes also talks about attachable and detachable property. By default, threads created are joinable. Only joinable threads can be joined with the main process by using pthread_join() function. Detachable threads cannot be joined using this function. When a datable thread is terminated its resources are immediately freed from the kernel space on its own, parent process does not wait for termination of detachable thread. The problem with these is that it is not joined from the parent process therefore, it is not guaranteed that the parent process will always successfully wait for the detachable process to execute. There are 2 ways to create them-

1. Pthread_detach();
2. Using pthread attributes.

MEMORY MANAGEMENT UNIT-

In the kernel MMU is a subsystem which is divided into 2 parts-

1.high level MMU-

It is totally independent of CPU architecture; it has nothing to do with the hardware. Even the architecture is of intel or ARM high level MMU will work the same.

If an application or any kernel services requesting more memory like malloc or calloc they calls the high level MMU which creates memory of the specified size.

High level contains 4 types of algorithms to serve this memory allocation-

1. **slab allocation**- allocates continuous small size memory.
2. **fragment allocation**-allocates non continuous large size memory. It identifies all the empty spaces in every page and join these fragments virtually to give larger memory to the application.
3. **page allocation**-allocates memory in terms of page.
4. **pool allocation-** generally used by applications (programs of driver). When we need a fixed size of memory frequently high level uses this algorithm.

2.low level MMU-

It is dependent on the CPU architecture. For different architectures like Intel or ARM low level MMU will be different on each architecture.

It is loaded to the RAM with the kernel at the boot loading time which initializes lots of memory management data structures and creates pages for the memory. It is also responsible for shifting CPU is protected or logical mode where CPU sees the memory as blocks (pages) which is a structure containing information about the memory and each page is of 4kb. This page stores and gives the virtual address of any variable used in the application or process.

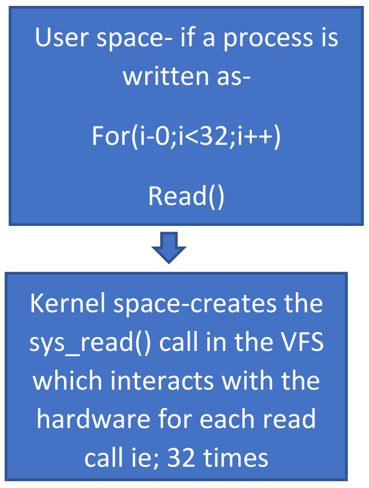
Virtual address= page no. + offset (address at which the variable is stored in the page)

User space- app1 app2 app3 …………………………………………………..
Kernel space-
High level MMU- allocates memory according to the need of application
Low level MMU- contains linked list of struct page which gives the virtual address.

MEMORY MANIPULATION CALLS-

1. memset- void * memset(address, value, size); ex- memset(address,'#',100); this will write # in the given address for 100 bytes. This returns pointer to the given address. It is used to initialize of fill the memory with a particular value for given size.
2. Bzero- bzero(address, size); it changes the values or initializes the memory at the given address to 0 for given size. Ex- when we call calloc() it assigns the memory and initializes with 0 using bzero.
3. Memchr- void * memchr(address, value, size); it scans the given address location and finds the value given to it and returns the address at which the value is found using the internal pointer. Using this address, we can modify or manipulate the data from that address. If the value is not present, then it gives segmentation fault.
4. Memcmp- int memcmp(address1, adress2, size); it goes to the given address and compare data byte by byte on the given addresses until it gets different characters in the addresses. It returns the ascii difference if it finds the characters on the given addresses. If the returned value is + it means address 1 is > address 2, if – it means address2>address1 and it returns 0 it means both are equal.
5. Memmove- void * memmove(destination, source, size); takes data from the source address and copies it to the destination address for the specifies size. On success returns pointer to destination and on failure returns -1. There is no risk of overlapping of source and destination (mixing of data or addresses) as it takes the data from source and first copies it to a temp buffer then copies it to the destination. It is a slow operation but provides a guaranteed operation.
6. Memcpy = void * memcpy(destination, source, size); does same operation as memmove, but there is no temp buffer hence source and destination overlapping is at risk. Operation is fast, but no guarantee of data in case of overlapping.

PROBLEM WITH USER SPACE AND KERNEL SPACE OF THE RAM-applications are abstracted from the hardware, every time applications have to go through kernel driver layer to access the hardware. To overcome such situations we provide a user driver space in the user space which directly reads from the driver without going to the kernel space.


User space- if a process is written as-
For(i-0;i<32;i++)
Read()
Kernel space-creates the sys_read() call in the VFS which interacts with the hardware for each read call ie; 32 times

USER SPACE DRIVER- means driver is still in the kernel space, which expose the hardware information to the application. It is showing hardware details to the application and allows the application to implement its logic on the hardware. To implement such operation Linux provides mmap() function.

MEMORY ANNONYMOS ALLOCATION-

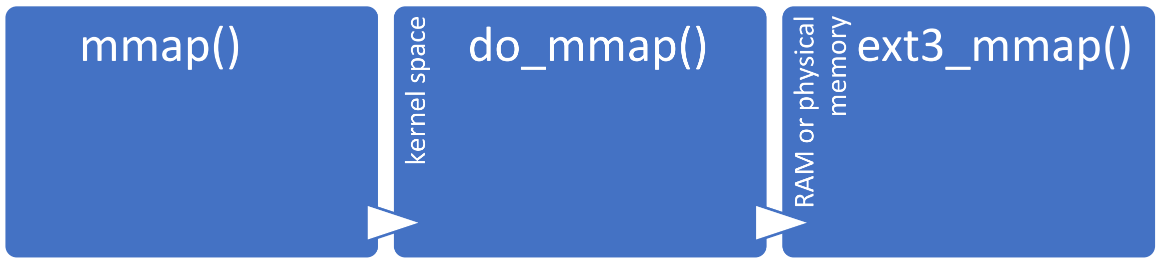
It contains some functions for memory allocation such as – mmap and munmop.

Mmap- is a Posix standard function. It maps kernel, file or device memory region to process address space. The kernel decides at which segment in the process address space the data from the file region (in kernel space which will be copied to the process address space) will be copied.

Void * mmap(**address** (at which segment in the process address we want to copy the file segment or map the file), **size**, **protection** (to protect the file information of the kernel space copied in the user space), **flag** (nature of mapping like MAP_PRIVATE (mapped region only used by the current process) or MAP_SHARED (mapped region shared by some other processes.)), **fd** (of the file from which we want to map, could be a normal file or device file) **or -1** (some random kernel memory), **file offset** (position from which we want to read or write));

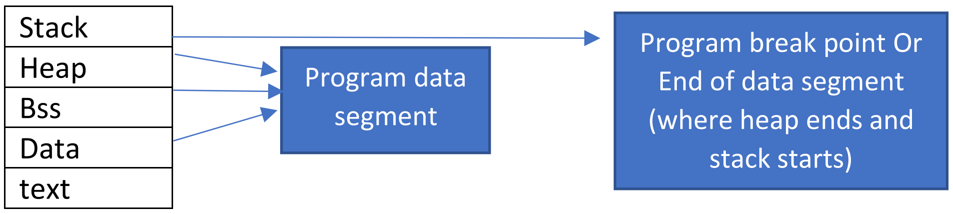
On success return pointer to mapped region. When we provide address argument as 0 we are asking the kernel to map a file region or a device region at a free location in process address space that can be any segment (text, data, bss, heap, stack).

WORKING-



When we call mmap in the user space it calls its kernel version do_mmap in the kernel space. Then to get the physical address of our file/device memory do_mmap calls ext3_mmap. This call converts the physical address of the file into a virtual address and copies the data from the file on this virtual address. Now this data and address is copied to the process address space of the process in the user space.

PROGRAM BREAK POINT AND PROGRAM DATA SEGMENT-


Stack
Heap
Bss
Data
text
Program data segment
Program break point Or End of data segment (where heap ends and stack starts)

We can use some calls to manipulate this program break point to increase or decrease the stack and heap segments.

1. Void * sbrk(value)- if it takes some +ve value and increase the program break point by that size. It returns start point of new address of the program break point. It increases the end of data segment by the given size. If it takes -ve value, it will decrease the size of data segment and if it is 0 it gives the exact location of the current program break point. On success returns new address and on failure returns -1.
   Sbrk(10)- increases the data segment by 10 bytes.
   Sbrk(-10)- decreases the data segment by 10 bytes.
   Sbrk(0)- gives the current position of the program break.

2. Int brk(void * end_data_segment)- used to change the location of program break point. it takes the address as an argument and changes the location as per that given address. On success returns 0 and on failure returns -1. It does not give any address therefore we have to use sbrk with it to display the address.

Why do we transfer the file from hard disk and execute in RAM-

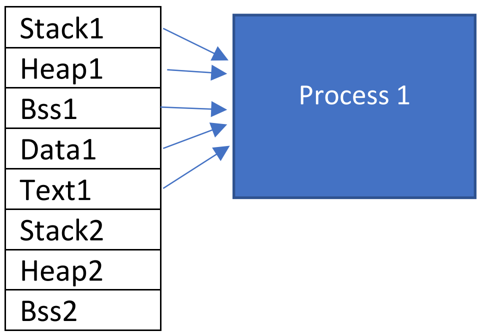
1. Hard disk access is slow and RAM access is faster because it is nearer to the CPU.
2. Hard disk is divided into blocks of data which is not understandable by the CPU whereas RAM is divided into bytes where each byte has its own address which is understood by the CPU.

1. **SWAP PARTITION**- when lots of processes are carried out in the system, each process memory is shifted to the RAM for execution. If the RAM is full and we want to execute more process then the OS will first wait for a process to terminate on its own but if this is not possible then, the OS will find which part of the memory segments of the processes are not in use and the time of its process execution and transfer it to a special part in the hard disk known as “SWAP partition”. It is generally made twice the size of RAM. Now there is space for new process in the RAM. When we want to use the transferred memory segment in its process the OS will swap the segment from hard disk to the RAM.

Diificulties-

2. To keep track which segment of which process is loaded into the hard disk.
3. To keep track where the segments of the new process are located in the segment of which process's emptied segment.

   RAM



PAGING TECHNIQUE- to solve the issue of which segment is loaded in which part of RAM we use this technique.

Stack	4
Heap	3
Bss	2
Data	1
Text	0

- Each part of the process memory segment is divided equally and called as "page".
- These pages are created by low level MMU at the time of kernel boot up.
- Each page size is 4kb. Like this the physical memory (RAM)is also divided into equal size units, called as physical frame or page frame.
- the size of page frame should be equal to the page size (4kb).
- Physical frames are numbered from 0-n (n=size of RAM).
- 0kb is the starting address of the 0^{th} page frame and 4kb is the start address of 1^{st} page frame. Each page frame starts after 4kb size.
- A page table is maintained which contains the information which page number is mapped in which page frame. Also called as page frame relation table.
- Number of pages assigned to a process are not fixed. It depends on the data in the process, ex- if we do dynamic memory allocation, we increase the number of pages in heap or-

1. when we use threads the page numbers of the stack segment increases.
2. when a function invocation takes place, it also increases the stack segment page number.

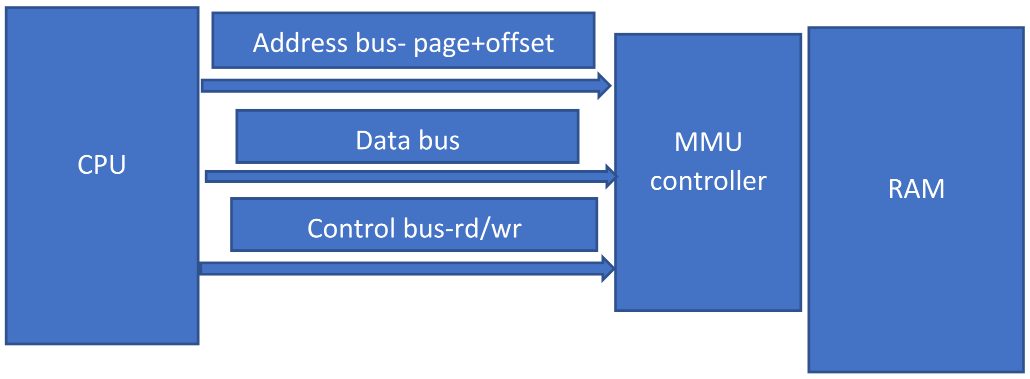
3. when the process calls mmap function the memory outside the process is mapped in the process memory hence increasing the page number.
4. when a process uses shared memory.

- It is a dynamic data structure maintained by the OS in the process control block (pcb).
- Generally pages of multiple processes are loaded to different page frames but there are some scenarios where these pages may share the same page frame. These are-
    1. When a child process is created it initially uses the same page frames used by the parent process.
    2. When we use map_shared flag in mmap() function.

VIRTUAL ADDRESS TRANSLATION-

When we use a variable in the program it is saved in the virtual address and mapped into the physical memory. When we want to assign some value to the variable the CPU takes the virtual address (page number + offset), the data given and the control signal(read or write) and looks for the variable in the physical memory(RAM). Before RAM this information is given to the hardware MMU controller which converts the virtual address given by the CPU into the physical address by-

Physical address= frame number (maintained in pcb) * page size (4kb) + offset value.



MEMORY LOCKING FUNCTION-

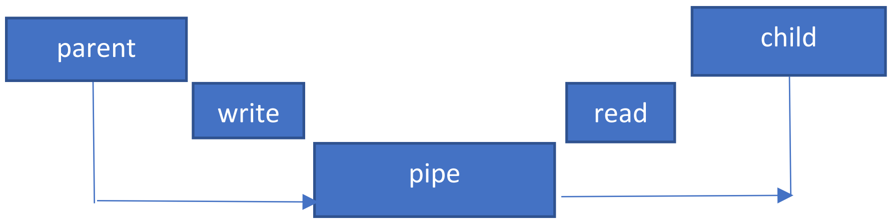
Int mlock(address,size)- it goes to the given address and locks the size of the address given to the function. This locked address can not be further used by the OS. It is used to lock the page from swap partition. When we want to lock all the pages of the program, we use mlockall().

If we want to allocate memory specifically from stack we use alloca(size). When heap memory is full and the program requires dynamic memory allocation we can allocate the stack memory of the process using this function. calling a free function is not required in this case because when the process terminates the stack is automatically freed.

INTERPROCESS COMMUNICATION-

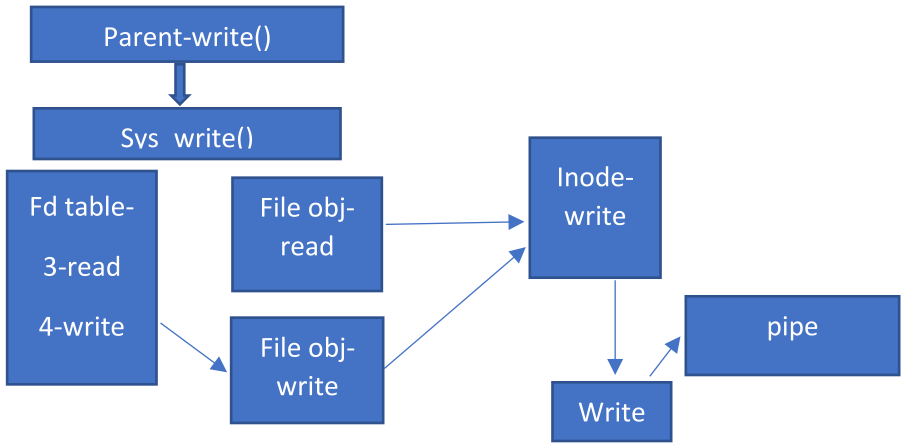
Linus OS provides various communication techniques for process communication, such as-

1. **Pipe**- it is a communication object used between related processes (parent and child). It is internally a file which has memory and buffer to receive and send data. When a pipe is created in the kernel space, a buffer is created which is used for read and write data. It is a special file which has 2 file descriptors, at write end and at read end. Data written on write end is read in the same sequence as written. It is a half-duplex (unidirectional) serial communication process.
   From application pov, to create a pipe we need to use a function pipe() called as pipe system call. Pipe takes an array of 2 int elements as an argument. Its return type is int.
   Int pipe(int p[2])- returns 0 on success and on failure returns -1. After succession a pipe (special file) gets created in the kernel space. It has 2 fd, f[0]- associated with the read end of the pipe and f[1] with write end. According to the fd table rule least available index will be filled in p[0] and the successive index in the p[1] (from the array in the argument of call).
   To see the max size of pipe- cat /proc/sys/fs/pipe-max-size

   For normal files read call is non blocking call (it will return some values and doesn't get blocked by the schedular), but in pipe read can get blocked by the schedular if there's no data in the pipe and the process (child) tries to read data from that empty pipe. For read and write operation 2 separate file objects are created because they have separate fds. In the file object of pipe along with the pointer to the inode structures it also contains flags to define the operation for the pipe(read or write). The inode structure actually

contains another pointer for the buffer of pipe (the actual place in the kernel where the pipe is created).
FOR PARENT PROCESS TO WRITE DATA- the write call will call the sys_write in the kernel which goes to the fd table and chooses the index given to the write call. This fd will point to 2 file objects (file obj for read and file obj for write) and chooses the file object with write flag in it which will point to an inode structure for the write end of the pipe.
FOR CHILD PROCESS TO READ DATA-
THE READ CALL WILL CALL THE SYS_READ CALL AND GOES TO THE FD TABLE AND CHOOSES THE INDEX FOR THE READ FILE OBJECT, WHICH POINTS TO THE INODE STRUCTURE FOR THE READ END OF THE PIPE.



The major limitation is that it is only used for the parent and child communication.

2. **FIFO**- also known as “named pipes”. used for communication between unrelated processes. Command to create a fifo is mkfifo on the terminal as- mkfifo filename.
These are registered as files in the file system with respective names. Returns 0 on success and -1 to failure. In the type of communication one process is a writer or sender and other process is a reader or receiver. Ex- if we create a fifo on 1 terminal and use command cat > fifoname to write on the terminal we can read the same data on another terminal by cat < fifoname. Here the two terminals are acting like 2 different processes, and we have created a fifo. between them to communicate as writer and reader. In real time applications first step is to check whether the fifo object exists or not. If it is existing, then it should just open it and use it. If it is not existing, then create a new fifo with mkfifo and then open and use it. To check the existence of fifo we use open call if it returns the fd then the fifo exists if does not it returns -1.
TYPES OF COMMUNICATION (FOR NETWORKING)-
   1. CONNECTION ORRIENTED- connection should be established before the transfer of the data between the sender and receiver. Read and write are only possible after the connection is made. Ex- <u>in fifo until the receiver opens the file the data from writer is not send to the reader.</u>

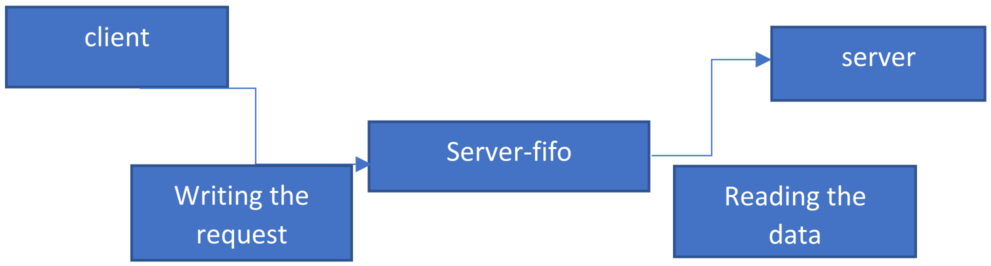
2. CONNECTION LESS ORIENTED- sender can send the data without establishing the connection prior to the transfer of data with the receiver without caring receiver is active or inactive. Ex- messages in phones.

DIFFERENCE BETWEEN PIPES AND FIFO-

PIPE	FIFO
1.USED for related processes	Used for unrelated processes
2.it is unidirectional (half duplex)	Also half duplex and unidirectional
3.uses same basic I/O calls	Also uses the same calls (read, write, open)
4.Created in the kernel space as files	Created in the file system with names
5.If a process is terminated all the kernel objects are lost (pipe gets terminated)	If process is terminated fifos as a file still exist in the file system.
6.Pipes are created using pipe system call	Created by the command and the same call mkfifo.
7.read acts as a blocking call	Both read and open are blocking calls.

We need to write 2 programs- one for server and one for client.

We start with the server program where the server waits/blocks for the client connection request. Once the connection request is sent by the client to the server, it accepts the request then client and server are ready for communication. To establish this communication the server creates a fifo object and gets ready to read from the fifo. The client access this fifo and writes the requests for the server in that already existing fifo. Once the request is read from the server it starts to process the data. To perform the requested task.



3. **Message queue**- it comes under system file IPC system. Issues of blocking (open and read) faced by fifo is eliminated in this technique.

Difference between fifo and message queues-

FIFO	MESSAGE QUEUE
Connection oriented mechanism	Connection less mechanism
These are unidirectional, half duplex	These are bidirectional, full duplex
Uses basic i/o system calls (open, read)	Uses separate I/O calls
Stream based data transfer (continues and consistent data flow)	Packet based data transfer (data divided into small packets or units)

These are linked list of messages stored in kernel space used for process communication in user space. These messages are sent in a queue form. Each queue is linked with unique id called as queue id. When a process wants to communicate with another process , the data is stored in the form of message and passed to the message queue, the other process will scan for the data and follow a process to select the particular message. Messages are composed of data and type.
Message = data + type. While executing if the type field is 0 it will fetch the data in first in first out manner. If the type field is > 0 if will fetch the specified msg. if the type field is repeated it will return the first occurred msg.

For each message queue Linux kernel maintains a structure called as struct msqid_ds {};
That maintains the current status of the MQ. This has-

- Number of messages in MQ ;
- Permissions;
- number of bytes in MQ;
- process id of last process that has performed last message send operation;
- process id of last process that has performed message received operation;
- time of last message sent;
- time of last received message;
- current time;
- first msg pointer;

Tool to check number of message queue in the system- ipcs -q

SYSTEM CALLS-

- Sys_msg.h header file is required to create message queue.
  Msgget(key_t, msg flags);- on success returns msg q id on failure returns -1. Key_t is internally an integer type. Whenever a process requests to uses the system resources like mq, shared memory or semaphores this key is needed by the Linux OS. Key is like a name to the msg q. when a process creates a msgq it returns a unique key. Message flags are also used with some permissions to the msg q. ex- IPC_CREAT | 0640.IPC_CREATE flag with msgget function will check for the given key if that msg q is existing, if not existing it creates a new msg q in the kernel space, if exists it uses that msg q. after this call members of the struct msgq_id will be set to some values like permissions will be set to the permission provided in the msgget function, no. of msgs and bytes will be set to system

limit and other members are set to 0 except current time which is set according to the system time. key is used by unrelated processes to access msg queues created by other processes. key is used by unrelated processes to access msg queues created by other processes.

- Msgsnd()- used to append msg at the end of the msg q. on success returns 0 and on failure returns -1. takes 4 arguments-

  Int msg id- tells the msg q at which we want to send the msg.

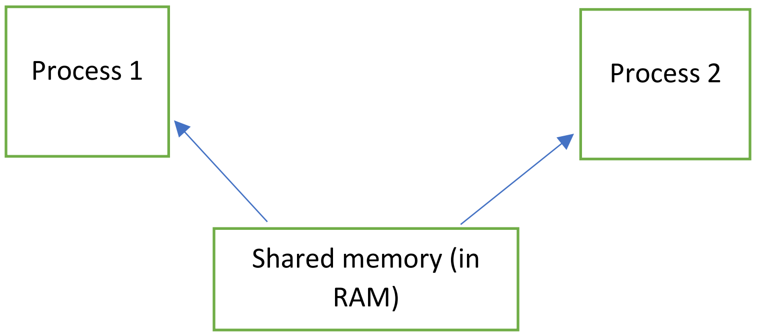
  Void* ptr-pointing to the msg we want to send.

  Size_t- number of bytes we want to write.

  Int flags- if the msg q is full and the sender process wants to send more msgs to the msg q then, sender process by default will be blocked until the msg q creates some space because receiver process will receive some of the msgs from the msg q. these blocking is prevented by using flags like IPC_NOWAIT. This can happen with the receiver side as the type given to the receiver is not present in the msg q, again this flag can be used to prevent the wait or block condition.

- Msgrcv()- is used to receive a msg from msg q. takes same arguments as msgsnd() with 1 additional argument of long type. on success returns size of data portion on failure returns -1. While executing if the type field is 0 it will fetch the data in first in first out manner. If the type field is > 0 if will fetch the specified msg. flag used to prevent the blocking is IPC_NOWAIT
- Ipcrm -q id or ipcrm -Q keynumber- to remove the msg q with the provided id, or key number.
- Msgctl(msg id, command, &variable with same structure as msqid_ds)- used to control the message queue. Command can be IPC_STAT which copies all the info of queue struct and copies to the provided buf structure.

4. **Shared memory-** fastest communication technique. In this technique a part of physical memory is mapped into the processes that want to communicate with each other.



No system calls are used in this process. The processes are not moved from 1 memory location(user) to another location (kernel) for data transfer. Instead a given region of memory is attached or mapped to the process memory segment, therefore it is the fastest way of communication technique. This shared memory segment in both processes should

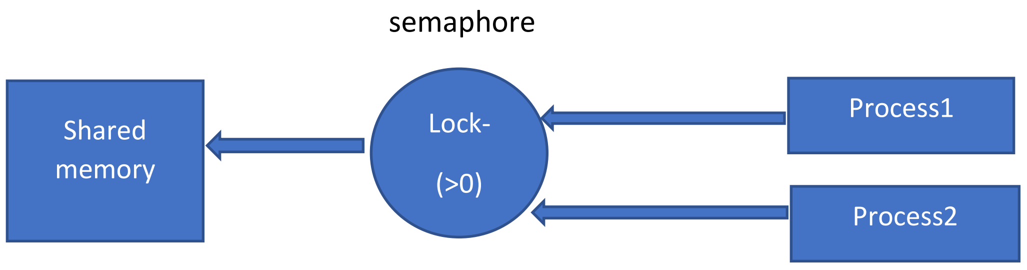
be secured as both the processes are in the race condition to access that shared memory. To avoid this race condition we use synchronized access (which is not present in the default condition) to make sure that when 1 process is using the memory and after the task completion (read or write) ,2nd process can access that memory to do its task (to read or write). Each shared memory segment has its own structure which gives the current status of the shared memory. It is –
Struct shmid_ds {};- provides complete current status. Members are-

1. Permission
2. Shared memory segment size
3. Last pid of attached memory
4. Last pid of the detached process
5. Last time share memory attached.
6. Last time shared memory detached.
7. Current time

SYSTEM CALLS-

1. Command to check shared memory segment- ipcs -m
2. Include the header file sys/shm.h
3. Shmget()- used to create new shared memory segment. Takes 3 arguments. Key_t, size_t, int flag. On success returns pointer to the shared memory segment and on failure return -1. Has int type return value. After the call a shared memory segment is allocated with the defined structure for it and all the members are updated according to the current status of the memory. Permissions provided in the shmget function will be given in the permission member of the structure, same with the provided size. Other members will be 0.
4. Shmat(int shmid, address , flag)- return type is a void pointer. It takes the shmid and attaches the specified segment to the process address space. Shared memory is created in the physical memory in the RAM, but shmat function attaches shared memory to the process address space and returns virtual address (page+offset) that is understood by the process. On success returns pointer to the mapped address attached to the process, failure -1. Address argument in the function tells where in the process address space we have to attach the shared memory. Generally we take it as 0 so the kernel will find free space in the process and attach the segment there. Argument 3 is the flag which is given as SHM_RDONLY, its default value it can do both read and write.
5. Shmdt (address given by shmat)- removes the shared memory segment from the process address space.
6. Shmctl(shmid, command, buffer)- on success returns 0 and failure returns -1.used to give specific command or access the structure of the shared memory.

5.Semaphores- By default shared memory is not synchronized, to synchronize the shared memory semaphores are used between clients and servers. Shared memory can't be accessed by 2 processes at the same time. It has to be synchronized using semaphores.



Process job on semaphore-

1. Decrement semaphore value by 1 and takes the lock.
2. Process uses the shared resource. It can be shared memory, a global variable, a global data structure, or It could be some share output devices.
3. Process then increments the semaphore lock value by 1 and give back this token to the semaphore.
4. Taking the lock- -1 to the value, Giving back the lock- +1 to the value.
5. The lock value cant be less than 0, therefore when 1st process access the lock and make it 0 (if the initial lock value is set to 1, when the 1st process access the lock it will decrement it to 0), the other process accessing the lock will be blocked until the lock value (with increment)is given back to the lock by the first process.

APPLICATION FOR SERVER-

Creates the shared resource (in this ex-memory), creates the semaphore and wants to read the data in the memory. When there's no data in the memory the server should wait for the client to write the data. This waiting condition is introduced by the developer by making the lock value in semaphore as 0, so that when the server access the memory and make the lock value as -1 it will get blocked until the client makes the lock value back to 0 by writing the data and incrementing the lock value from -1. This sends a signal to the server to start the process and read the data.

SYSTEM CALLS-

1. Semget (key_t, no. of semaphores, flags);- returns semid on success and -1 on failure. Return type is integer. Flags are same as shared memory.
2. Semctl (semid, semaphore number, command, value);- semaphore number in this function is the index number of semaphore that is if there is 1 semaphore created by semget() call the argument will be 0 in this call as it's the first index of the semaphore array, next semaphore numbers will be 1,2,3…n. the value argument

in the function is the initial value given to the lock of semaphore, or the semaphore value.

3. Semop(semid, struct sem_buf *ptr, no. of semaphore)- performs increment and decrement function on semaphore locks.
   Sem buf structure-
   Sem_num;
   Sem_op;
   Sem_flags;
4. 2 types of semaphores- system Y (complex design with more options) and POSIX(simple design but few options, used for threads)

DISADVANTAGES OF MULTITASKING-

1. Different process or threads tries to access same global- shared data or resource it leads to race condition. In this condition to prevent the data from corruption we apply lock to the critical section of the (the part at which the race condition may happen or when 2 processes share the same resources). This is to synchronize the program and prevent the data and resource corruption.
   Critical section- set of instructions or code that is dealing with accessing of global shared resources (variable, shared memory, output device like printer or data structure).to avoid this or prevent the race condition we synchronize the processes with locking and unlocking in different techniques.
   SYNCHRONIZATION TECHNIQUES- Linux provides these locking techniques.
   - **Semaphore-** these are optimized for non-contention cases (no competition, when 1 process uses the resource the other process is simply in the wait state which belongs to the semaphore).
     Cannot be used with interrupt contexts or interrupt handlers (because it increases the interrupt latency or time to execute the interrupt).
     No issue of process priorities (does not execute the processes according to the priority).
     Counting semaphore- a semaphore whose value is >1 is called as counting semaphore used for protecting multiple global shared resources.
     Binary semaphore-
     Process will take the lock decrement it and after completion returns the lock with increment. A binary semaphore changes between 0 and 1. Semaphores cant be negative therefore if one process had accessed the lock and made the lock 0 from 1 the process cant access the lock by making it -1.
     Pthread semaphore-
     1. Declare or create a semaphore object/variable.
        Sem_t mysem- is a data type, to declare the semaphore variable.

2. Initialize the object/variable-
   Sem_init(&mysem,0,1)- initialize the semaphore with object 1, 0 shows that all the threads of the process can access semaphore variable. If it is non zero it is sharing between different processes.
3. Applying a lock-
   Sem_wait(&mysem)- here we have the critical section.
   Sem_post(&mysem)- to unlock the semaphore.

- **Spin lock-** contention is very expensive in terms of kernel resources.
  interrupts can be integrated with spin locks. In this technique when a process access the lock other processes does not go into block state, instead they go into loop condition on top of their respective processes and keep checking for the availability of the lock. This is called polling or spinning of processes. Therefore, we can not afford to keep all the processes busy for long time in this spinning condition. To prevent this no delay calls are added in the critical section of the code hence interrupts can be integrated with spin lock and is faster in use.
  Spinlock procedure-
  1. Declare spinlock variable-
     Phtread_spinlock_t myspin;
  2. Initialization-
     Phtread_spin_init(&my_spin,PTHREAD_PROCESS_PRIVATE); here argument 1 is address of spin lock variable, second is a macro which tells if the variable is shared between the threads of the same process(PRIVATE) or shared between different processes (SHARED) when processes are in shared memory.
  3. Locking –
     Pthread_spin_lock(&myspin);
     Pthread_spin_unlock(&myspin);
  4. Termination-
     Phtread_spin_destroy(&myspin);

- **Mutex- Mutual Exclusion is** similar to semaphores (binary) with a usage count of one. 1 process or a thread holds 1 mutex at a time. A process locking mutex must unlock the mutex. It cannot terminate while holding the mutex.
  Cannot be integrated with interrupts. Semaphores by default are not initialized, we have to initialize the semaphore lock value. There's no need to initialize mutex lock variable by default they are initialized by 1.
  1. Declare mutex variable-
     Pthread_mutex_t mumutex;

2.initialize mutex- 2 types-
Dynamic- pthread_mutex_init(&mymutex,&mutex_attr(generally null));
dynamic is more flexible as the attributes can be changed during program execution time.
Static- pthread_mutex_t my_mutex=PTHREAD_MUTEX_INITIALIZATION
for this program attributes are fixed throughout the program.
3.locking-
Pthread_mutex_lock(&mymutex) or pthread_mutex_trylock(&mymutex);
Pthread_mutex_unlock(&mymutex);

CONTEXT SWITCHING-

Switching of the CPU from 1 process or a thread to another. When kernel transfers the processor from 1 execution to another (generally after the time has expired for the process). For this kernel has to save the process context (process data + CPU register value) in the PCB of the process in the kernel space. Then kernel will load the new process and execute it. The process terminated from the processor will be executed when it will get next slice time from the CPU. Execution begins from where the CPU was taken off with the help of context saved information and instruction register value.

SCHEDULING POLICIES- 2 types- round robin and complete fair scheduling. Deals with the CPU slice time assigned to each process.

ROUND ROBIN – each process gets equal amount of CPU slice time and is executed in circular order. Each process is allowed to use the CPU for a given amount of time. If task is not completed in assigned time, then the next process will start their task to execute, and the incomplete task will be resumed after all the time slices are completed. All the processes have the same priority. It is implementing a starvation free scheduling.

COMPLETE FAIR SCHEDULING- Linux uses this policy. It does not assign slice time directly to processes, instead proportions of slice time depend on a factor called as "nice value". This factor can increase or decrease the weightage of the process with respect to CPU slice time. Nice value will prioritize the processes.

Nice value	Priority	CPU time (total 100ms)
-20	Highest	More CPU time (88ms)
0	Default	10ms
19	Lowest	2ms

Since every process, even the lowest priority process will get a CPU slice time, therefore it is called as fair scheduling. We use a tool or a command called taskset tool which is used to assign CPU core to particular processes for execution.

NICE- it is a function used by Linux OS to change process priority.

When we call this function it invokes the same function in the kernel space which invokes a particular process's priority inside the PCB and set the new priority.

"&" is a special background symbol that runs the process in the background.

After top command- will show the process update every sec.

- Enter f key.
- Select P by using space bar.
- Esc.

Schedular calls-

1. Struct sched_param sp= {.sched_priority=1}; this structure contains parameters that can be used to change the policy of the process.
2. Sched_getshceduler(0); to set the policy.
3. Ret = sched_setscheduler(0, SCHED_RR, &sp); to change the policy.
   Arg1- setting policy for the current process.
   Arg2- type of policy
   Arg3- structure parameter.

SIGNALS- signals are the events generated by the OS on some conditions which are sent to processes, on receiving signals has to perform the action based on the signal conditions. Out of 64 signals supported by Linux 1-31 traditional signals are 32-64 are real time signals. Kill -l will show the signals in the OS.

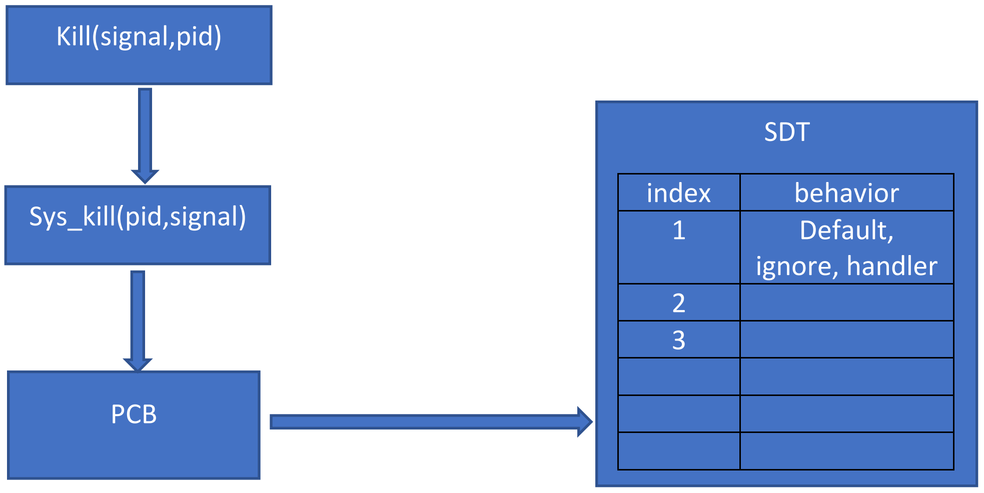
CONDITIONS TO GENERATE SIGNALS-

1. **Terminal special key combination**- ctrl+c, gives signal to the process to terminate itself.
2. **When an exception occurs**- variable divided by 0 is an exception because it is an invalid function. It will stop the current process. When these exceptions happen exception handlers are started which are special function to handle the exception. This handler searches for the pid at which the exception (illegal instruction) is occurring and then sends the signal to that pid process to be terminated with generating kill signal called as SIGKILL.
3. Executing invalid address- using the freed pointer, using the pointer without initialization.
4. **Kernel can also send signals to the process**- kernel sending a signal SIGCHLD to the parent process to tell it that the child process has been terminated.
5. **Using a system call we can send signal to process**-
   Kill(pid,sig_number(1-64 signals supported by the OS));
   We call also use kill command from the shell to terminate the process.

DELIVERY OF A SIGNAL- PCB of a process is also called as task_struct structure which contains some additional information regarding to the signals given to the process as-

1. **Signal disposition table**- contains signal behavior. This contains how to handle all the 64 signals available in the OS. It is a table of 64 entries. Each entry will have either of the following information-
    - Signals Default- SIGDEF. It executes the default behavior of the signal defined by the OS in signal.h library.

    SIGNAL GENERATION- when we give signal to a process by kill -signal no. pid. The kill command is actually an executable file which internally contains kill(pid, signal no.) function which invokes sys_kill function in the kernel space. This sys function looks for the pcb with given pid and in the pcb and access the signal disposition table and uses the signal number as the index for the signal table and executes the value in the respective entry (default, ignore or handler). For most of the signals default behavior is to terminate the process.



- Signal ignore- SIGIGN ignore the signal, process will not have any impact of the signal. Signal number 9 and 19 can never be ignored. When a program is doing illegal instruction such as -number/0, or invalid memory access (executing uninitialized pointer or using a freed pointer), to stop these processes kernel has to terminate these processes by using SIGKILL (signal no. 9) or SIGSTOP(signal 19) therefore, kernel can never ignore these signal numbers.
- Signal handler- user defined function. Special function defined by the user to deal with signals. <u>Using these types of signals, we can access or execute a function in a process without writing it in the main function.</u>

INSTALLING A SIGNAL HANDLER IN SDT- also called as modifying the SDT. There are 2 methods-

1. Signal(signal number,signal handler()); 1st argument is the signal number for which we want to modify the behavior in SDT. 2nd argument is the function that is to be executed when that signal is generated.

After the installation of handler next step is to execute the process continuously, onl the we can check the changes in the SDT.

2.sighandler()- it is to be used when a signal is generated.

To maintain multiple signal information, we use a structure called as struct sigset_t. It deals with multiple or group of signals. There are some system calls that deal with this group of signals. These signals are used to set some signals which will be blocked or delayed for the current process until those signals are removed from the blocking list.

- Sigaction(sig_num,struct sigaction *ptr,NULL);- return type is int (success-0,failure- -1). Argument1- signal no. to modify, argument2- pointer of type struct sigaction predefined in signal.h header file.
- Sigprocmask(cmd, sigset_t,sigset _t *oldset);- this function can change process signal mask. Blocking, unblocking and all the information of the signal mask is given by this function. Can also retrieve the blocked signal information.
- Sigpending()- pending system call will give information for the blocked or pending signal of the current process.
- Sigemptyset(sigset_t *set)- this function will make sigset to empty set.
- Sigfillset(sigset_t *set)- will initialize signal set, all signal to active. It fills all the set with all the signals
- Sigaddset(sigset_t *set, signum)- individual signal will be added to sig set.
- Sigdelset(sigset_t *set, signum)- deletes the particular signal from the set.
- Sigismember(sigset_t *set,signum)- used to check the particular member in the group.

2. Signal mask- also known as blocked signal information. Kernel inside the PCB maintains a variable called as signal mask which is of 64bit for 64 signals in the OS. It talks about group of signals that are blocked from the current process. If a signal number is blocked but it is used in the process or called by the user, delivery of the signal to the process will be delayed (pending signals) until those are removed from the blocking set.
3. Pending signal information- If a signal is sent to a process and is blocked, again the same signal is sent then this delivery gets delayed and enter into the pending state until the signal is removed from the blocked state.
   Sigpending(sigset*ptr)- this will list the pending signal information for the current process.

NETWORK COMMUNICATIONS IN LINUX- a set of computers integrated together is called as computer network. Can be of 2 types-

1. **WAN –**

- covers a larger area.
- consists of number of interconnected nodes.
- transmission of data from any device will be routed through these inter-connected nodes to specified destination.
- Nodes are any devices which are connected over network like server and router.
- A router is a networking device that connects one or more computer to each other over network.
- WAN is implemented in 2 ways-
  Circuit switch network- a dedicated path is established between the sender and receiver through the nodes. The path is a connection of sequence if physical links between the nodes. Data transmitted from 1 source device routed through this physical links to the destination device. Ex- telephones.
  Packet switch network- follows networking protocols where the messages are divided into small packets called as network packets. These packets are transmitted through digital network.

2. **LAN-**
   - Is a collection of computer network devices that are connected together in 1 physical location such as an office, or school, or at home.
   - It may have small or large number of computers.
   - Internal data transfer rates are faster than WAN

INTERNET PROTOCOL ADDRESS- are used to identify a node in the network, also for communication between the nodes. It has 2 versions.

1. IPV4- supports 32-bit address.
2. IPV6- supports 128 hexadecimal addresses. has better performance and more features. Provides large address scope.

IP address consists of 2 parts- network id and host id. Network id (first 3 numbers with.) will be common for every device in the network and each host will have its own unique id (after the last .). Ex- 192.168.0.1 – network id- 192.168.0, host- .1. IP address is a logical address which changes for every network, we change the network, the address changes. They are used to identify the device in the network globally.

MAC ADDRESSES- also called hardware addresses, these are provided by the manufacturer of the hardware and attached to the hardware. Hardware address is used to identify the device locally. These do not change with changing the network. It is a 12 digit number.

NOTE-When a data or packet is sent over the network it uses both these addresses to identify the network and the device for the communication.

NETWORK PROTOCOL LAYERS- these layers are organized as series of layers. The reason for developing the protocols as layers is to reduce the complexity of the design. Each layer is providing functionality to the other layer.

OSI model – every data transmission follows these layers in each device in the network for communication. The transmitted data will follow these layers in the transmitting device and in the receiving device.

Application
Presentation
Session
Transport
Network
Data link
Physical

1. Application layer- provides an interface through which we can interact with the application. All the application protocols are present in this layer. Ex- http, https, ftp, pop, etc. if we are using google chrome or any other application the protocols for those are present in this layer.
2. Presentation- checks in which format the data is to be transmitted to the receiver, so that it can transfer the data in the fastest way possible and the receiver can understand the packet or data. It also provides encryption and decryption of data for data security. It also provides data compression.
3. Session- establishes connection sessions between sender and receiver until receiver gets all the data.
4. Transport- uses either TCP or UDP protocols to transfer the data. TCP (transfer control protocol) is connection-oriented protocol and UDP (user defined) is connectionless protocol. Through this layer data packets are sent to the network layer.
5. Network- divides the data into smaller packets of data and these are called as network packets. It holds the IP address of the source and destination. It performs all the routing of the data to determine the best path for the data transmission.
6. Data link- will look for errors in the data and remove the error, then transfer error free data. It maintains the data flow transfer speed between the sender and receiver. If the flow rate is not same or synchronized in transmitter and receiver data loss, traffic load or less data can occur. Therefore, data link layer maintains uniform speed between sender and receiver. It also identifies the hardware using the MAC address.

7. Physical- converts the data from data link into binary data. Decides if the data is sent through void connection (ethernet) or through wireless connection (internet).

SOCKET SYSTEM CALLS- supports many network communication protocols (tcp, udp, ftp, http, pop, etc.). These are generic in nature because they support many protocols with the same format. They only differ in size. Size is used as a key argument in socket system calls.

The first step of implementing the client-server application model is creating sockets on both sides. Sockets are end addresses or final end points in a communication program to which the connection is established. Internally, sockets are special types of files. Each socket has 5 parameters-

1. Protocol used.
2. Source IP address (32-bit network bite order)- server IP address.
3. Destination IP address (32- bit network bite order) – client IP address.
4. Source port address (16-bit network bite order). The port number determines the application from which the data is sent.
5. Destination port address (16-bit)- The port number determines the application to which the data is sent.

SYSTEM CALLS- for server-

1. Socket(family (AF_INET), type (SOCK_STREAM), NULL)- creates the socket. Argument1- which version IPV4 or IPV6, 2nd argument- type of connection (TCP or UDP), 3rd argument-
2. Bind()- used to assign the server side socket with well known IP address and source port number. after this call server socket has protocol + source IP + source port, therefore server socket is called “half binded”. Because the client is not connected it is called as half binded.
3. Listen()- announce that it can wait for a defined number of clients.
4. Accept()- waiting for client connection request.

For client-

1. Socket()
2. Connect()- for connecting to the server side socket. This request is accepted by the server accept() call and then the connection is fully established.