# Linux Training Notes

- uname -r -> gives kernel version
- uname -a -> ubuntu version

OS is a set of programs that manipulates an interacts with the hardware and provide an environment to run various application programs.

**Advantages of Linux OS:**
- Free/open source.
- Portable on any hardware machine.
- Linux is internally secure.(We don't require any third party antivirus)
- Linux is scalable or modular.
- Expected to run 24*7 without rebooting.
- Very short debug time.(Errors/Mistakes/bugs/failures/faults)

Linux License -> GPL(GNU Public license)

**Linux Booting Sequence:**
- Power ON
  -> CPU is going to execute some predefined jump instruction from the CPU registers(hardcoded by vendor).
  -> Search for BIOS code, jumps to preferred memory location in ROM.

- BIOS
  -> Initializes all the onboard controllers in the motherboard.
  -> Search for Bootloader code from the booting device(hard disc).
  -> Generally, the first partition of the any storage device have the bootloader code.
  -> Once, bootloader code is found the BIOS job is done.

- Bootloader
  -> In Linux, bootloader name is GRUB(Grand Unified Bootloader)
  -> Bootloader from the hard disc loads OS images into the RAM.
  -> It scans the hard disc for the OS images. Once, it find the OS images, it list the users.
  -> Now Bootloader stops executing.

- Kernel Startup
  -> Now, Linux Kernel OS starts executing in the RAM.
  -> Linux Kernel is the first software program that is loaded in the RAM and executes untill we shut down the system.
  -> Creating lots of Kernel data structures(DS for devices, file system, interrupt management, priority management, memory management).
  -> Creating threads(Initializes process->login process->login page).

**Compilation process:**
- Preprocessor -> gcc -E filename.c -o filename.i
- Compiler -> gcc -S filename.i -o filename.s
- Assembler-> gcc -C filename.s -o filename.o
- Linker-> gcc filename.c -o filename.exe(Dynamic)/gcc -static -o filename.exe
  //Static linker is appending the entire C library with the object code of the source.
  //Dynamic linker is appending just the required library function.

ELF 64 – bit file format: Linux OS understands ELF(Executable loadable format file) 64 bit file format. Compiler will process and the processed code is packed in ELF 64 bit file format.

**VIM editor:**

It is the most popular editor in Linux OS. It is a command oriented editor. It has two modes:

- Insertion mode: Whatever we write on screen, it is operated in insertion mode.
- Command mode: Whenever we press some related keys, it is operated in command mode. In command mode we got more options to control.

-> w – saving
-> q – quit
-> l – move cursor to right side
-> h – move cursor to left side
-> j – move cursor down
-> k – move cursor up
-> o – new line below cursor + insertion mode
-> O – add a new line above the cursor + insertion mode
-> a – insertion happens after the cursor position
-> A – Insertion happens at the end of the line
-> yy – copy a particular line
-> p – paste
-> x – delete data
-> u – undo operation

**Source Code Browsing Tool:**

- **CScope:** It is used to examine symbols(variable, function, macros) of source code. Once cscope is entered from terminal it will take to interactive screen with multiple options to examine source code.
  cscope -R is going to create a cross reference file 'cscope.out' that is used by cscope tool.
- **Ctags:** It is a programming tool used for software development process in large applications used in Linux/Unix variant operating systems. It supports many programming languages along with C. Ctags is creating a tag file and it will contain the names(Object/symbol) found in source file or the header file.
  ctags create a file that shows the index of the objects. Ctags internally uses locators.

  Ctags >> locators >> which name(var/fun) is used
  >> path name
  >> line in which variable is used
  >> type of var/fun

**Advantages of ctags:**

- Quick access across the files.
- Provide complete function information.
- Tells particular name is variable or function.

**Repository of user header files ->** /usr/include
**Repository of user library files ->** /usr/lib or /usr/local/lib

**Compiling multiple files and generating single executible files:**

- gcc one.c two.c three.c -o output (no funtion should be repeated).

- The other way is including all files in one file using #include"".
- We can use Makefile and make tool.

Writing all the file names for multiple file compilation, much time spent only in writng gcc commands. To solve this issue, replace gcc with Makefile and make tool that helps in compiling multiple files. It is difficult for the gcc to compile the source file which are in different directories and the folders. Makefile and make tool will manage all the source file and header files to be at one place.

**make:**

- make is a programming build development tool used in Linux and Unix OS in large application software.
- make tool automatically determines, which files needed to be recompiled.
- make tool gets all the information to build a program from a file called as Makefile.
- Makefile contains set of commands that are similar to terminal commands that uses variable names and target names to create object files.
- make executes shell commands.
- gcc commands are executed by Makefile.
- Makefile is a text file that contains the complete procedure for building built executable.

**Rules:**

- Source file, header files and Makefile should be in one common directory.
- Makefile may have variables.
- Margins says where variable should start from.
- Target are labels to Makefile. Under the target we write instructions. These instructions are executed by Makefile.

Syntax : target
targetname :
<-Tab-> write the command to be executed by Makefile.

- find is a shell command that finds the filenames that ends with particular extensions.
- **find -name '*.c' ->** After this command all the files with .c extension will get printed.
- **SRC = $(shell find -name '*.c')** -> After this command all the files with .c extension are getting stored in variable SRC.
- **gcc ($SRC) -o out** -> means dereferencing the variable SRC means all the .c files are getting replaced in the current line.
- '#' is used for comments inside the Makefiles.

**GDB GNU Debugger:**

GDB is a powerful debugging tool used for C and C++ programming languages. GDB is a free software used with Linux/UNIX OS.

- GDB allows the user to stop the program execution in middle and also allows the user to probe during an application crash.
- GDB operates with the executable file that is produced by the compiler.
- GDB never works with .c file.
- When we compile the source file with -g option, a symbol table information is added to the executable file.
- GDB uses commands for debugging.
- GDB is a powerful tool that can directly jump to any particular memory location by using x command.

**GDB commands:**

- breakpoint or b: We need to stop the program execution in middle for that we use break points. To apply breakpoints we need break functions and break line numbers, where the

program pauses or stop the execution. Most of the time breakpoint is applied at the main funtion because we really don't know where the error has occured. After breakpoint is set, now execute gdb run command.

- run or r: program executes till breakpoint.
- next or n: The next command will take to the next line of the code.
- step or s: The step command will also take to the next line of the code but there is some difference. Step command will step into the function call while next command will execute the function call.
- list or l: list command is going to list the entire program.
- print or p: print command will print the values of the variables.
- quit or q: q is for quit.
- info locals: info locals will provide the local variable information.
- p/x: It gives the hexadecimal value of variable.
- p/t: It gives the binary value of variable.
- p/o: It gives the octal value of variable.
- x/d: x is memory command jumping to the address and accessing entire memory over that address.
- x/o: It is a memory command jumping to the addresss and accessing entire memory and displaying it in octal format.

**Providing inputs to the program while execution:**
-> using scanf(), gets().
-> command line arguments.

String can be stored as:

| **char *s = "linux";** | **char ch[] = "linux";** |
|---|---|
| Data can not be modified. | Data can be modified. |
| String and pointer are stored at separate location. | String is stored in same location. |

After compiling source code a.out file is divided into segments:
-> bss segment
-> data segment
-> text segement

- All statements of our functions are converted into instruction and are stored into text segment of a.out executable file.
- Data segment contains initialized global and static variable.
- BSS(Block started by symbol) contains uninitialized global and static variables and by default they are initialized with zero.
- Heap segment contains memory for DMA variables.

- The moment main is getting executed memory is allocated in stack segment.
- Heap and stack contains in executable file.
- **size filename** will show the “text data bss hex” details.

Command line arguments are parameters that are passed to the program during execution time from command line interface.
**Syntax:** int main(int argc, char argv* argv[])

```
int * ptr; // pointer declaration

* ptr; //dereferencing a pointer

scanf(“%d %*d %d”,10,20,30); // %*d says don’t read the value
```

- GDB is not so effective when bug happens to be in heap segment. So, separate class of tools are used for tracking the bugs when they happen in heap segment.
- **Memory Profiling Tools ->** Heap profiling tools are used when bugs happen to be in heap segment.
- There is one disadvantage of calloc and malloc that whatever memory we ask to assign to pointer, it always tends to assign more memory so there is wastage of memory.
- **gcc -g filename.c -o filename -lefence** //integrating application with memory profiling library.

**Electric fence:**

Most of the time if we observe real time applications debug examples bug happens to be from users point of view:

- Accidentally, dereferencing a NULL pointer.
- Accidentally, derefrencing an uninitialized pointer.
- Accidentally, dereferencing a pointer which is already freed.
- Writing after the end of the memory allocation.
- Writing before the start of the allocated memory.
- Standard c library is allocating more memory than requested so gdb and gcc failed to trace out memory violations.
- Electric fence is a memory debugger and also called as memory profile library.
- Electric fence is library has its own implementation of malloc() and calloc().
- When programmer links electric fence with the application, EF will have dominance over standard C library memory functions(malloc and calloc).
- If memory error occurs, electric fence triggers application program crash via segmentation fault.
- Electric fence can identify the bugs writing beyond upper boundary region of a memory allocation.
- Electric fence can identify if a program is writing before malloc memory allocation/underrun error.
- Export EF_PROTECT_BELOW = 1 -> electric fence environmental variable(It will check if data is inserted before the start of allocated memory).

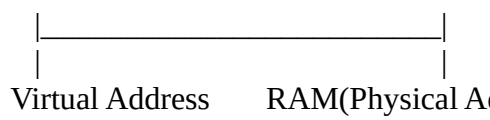
**Segmentation Fault:** When program is compiled and executed memory segments are allocated in the RAM. Our program is supposed to access the entire memory segment that is allocated that is our program can use text segment, data segment, bss segment, heap segment and stack segment. If program is trying to access region beyond this segment leads to segmentation fault causing the application to terminate.

**VALGRIND:**

Valgrind is a standalone debugger used at run time:

- To identify errors due to heap memory violations.
- Also used as memory profiling tool.

Valgrind runs without electric fence and use same standard C library functions and identify the program errors.




To the application and CPU, virtual address are written. Kernel maps the virtual address to respective physical addresses. Incase the program is using invalid addresses, Kernel fails to map physical address for invalid addresses. Processer raises an error to the Kernel. Now Kernel looks for the appropriate program i.e. making use of an invalid addresses and terminates the application or a program through segmentation fault.

**Error:** A mistake in programming code is error.

**Defect:** Before the code is released to the client, code is tested by the testing team to check whether code is meeting the requirements. If an error is detected by the tested than it is called as programming defect.

**Bug:** If the defect is accepted by respective development team then it is called as bug. Fixing and providing solution is called debugging.

**Failure:** Inability of the system software to execute and perform task function. Ex: breakdown in operating system, system hardware failure.

**Fault:** Fault arises due to some condition and leads to software failure. Ex: Page Fault(Inability of Kernel to map physical address)

**ARM Cross Compilation using Makefile:**

Compiler are of two types:

1. Native compilers -> compilation and execution on local processor(Intel).
2. Cross compilers -> Req: On Intel architecture write the program and generate executable for ARM achitecture so that the cross compilers are nedded to be installed on Intel Achitecture.

**gcc-arm-linux-gnueabi ->** Executable Application Binary Interface

Cross compilation is a process of generating executable files for other architectures.

```
CC = arm-linux-gnueabi-gcc

all:
	$(CC) four.c add.c sub.c -o out
```

**Makefile Dependencies:**

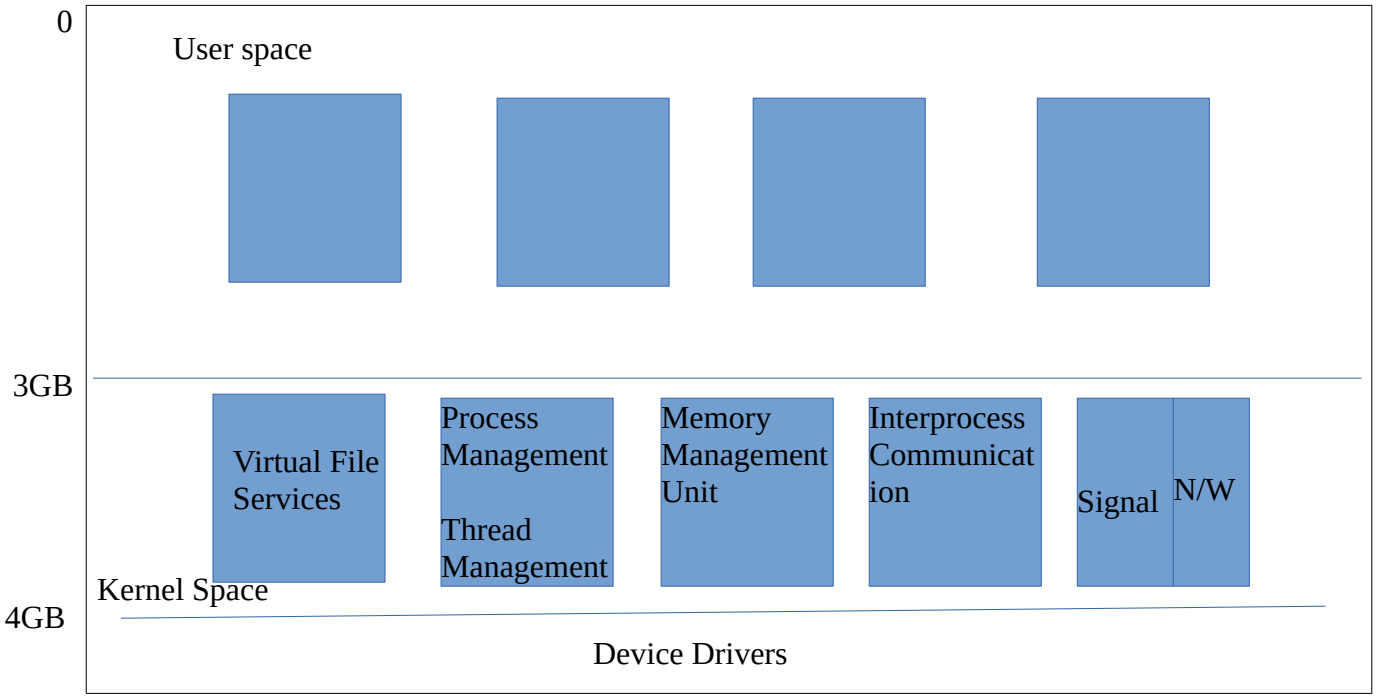
Makefile contains variables, contains targets and targets may have optional dependencies.

- Sometimes targets may depend on other files called as dependent file. So, dependent file is a input to create target files.

```
$(CC) = gcc

one: one.o add.o sub.o
	$(CC) one.o add.o sub.o -o output

one.o: one.c file.h
	$(CC) -c one.c

add.o: add.c file.h
	$(CC) -c add.c

sub.o: sub.c file.h
	$(CC) -c sub.c

clean:
	rm -rf *.o output
```

**Linux System Programming:**

**Linux Kernel Architecture:**




Application is a software which is readily available to use by end user. User space contains all the application. Kernel space contains set of programs.

**Kernel->** Core of the OS is Kernel. It is a software that talks to the hardware. Kernel is the first software program that is loaded into RAM and executes untill power off. Kernel is quite different from hardware.

**Layered Achitecture:**
Whenever application wants to communicate with the hardware. Application will make request to the kernel and Kernel on behalf of application is making a request to hardware. Applications can never interact directly with the hardware.

| Application |
|---|
| Kernel |
| Hardware |

Shell program acts as a bridge between user and kernel.

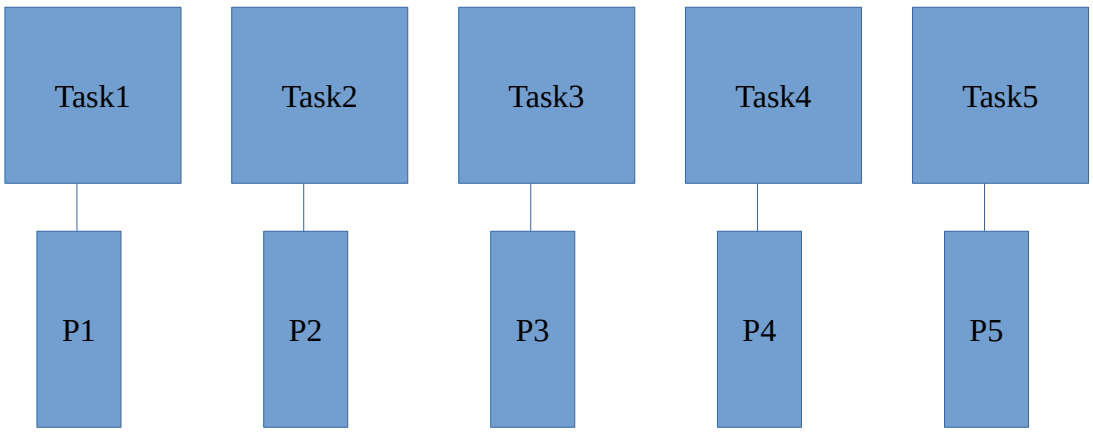
**Kernel Service:**

1. File system services
2. Input output device services
3. Multi processing services
4. Multi threading services
5. Memory mapping and memory allocation services.
6. Inter process communication services.
7. Process synchronization
8. Process scheduling
9. Signal Management
10. Network Programming
11. Drivers

**<u>Kernel File System Services:</u>** By using Kernel File sevices, a programmer can:

1. Write a program to open an exisiting file.
2. Write a program to create new files.
3. A programmer can write a program to read data sequentially from the file.
4. A programmer can write a program to write data sequentially to the file.
5. A programmer can write a program to modify file data, to change the properties of the file, to close a file.

**<u>Input Output Device Service:</u>** Linux operating system does not understand device hardware structure. Linux people implemented file for the devices and they name it as device files. A programmer can write a programs to create, open, read, write, modify, close a device file.



At design stage in order to reduce complexity of problem, the problem is divided into smaller tasks. To implement these task process management is providing processes.

- Process management will create the processes.
- Process management will schedule the processes.
- Process management will execute the processes.
- Process management will terminate the process.

**Multithreading Processes:**

Linux Kernel operating system provides various communication techniques for the processes to communicate with each other.

- Pipe
- FIFO
- Message queue
- Shared memory
- Semaphore

A process on one machine wants to communicate with the process on another machine. If you are writing program for such application we need N/W application programming for that we need IP address and Port address. To manage IP and port address we require Sokcets. Sockets require 5 parameters:

- Protocol
- Source IP address
- Destination IP address
- Source port address
- Destination port address.

Signals are software interrupts handled by the processors.

**Process Synchronization:**

Schedular:

- CFS (Complete fair scheduling)
- RR (Round Robin scheduling)

**Driver:**

Driver is a piece of software that is talking to the hardware. Driver residers in the Kernel space of RAM.

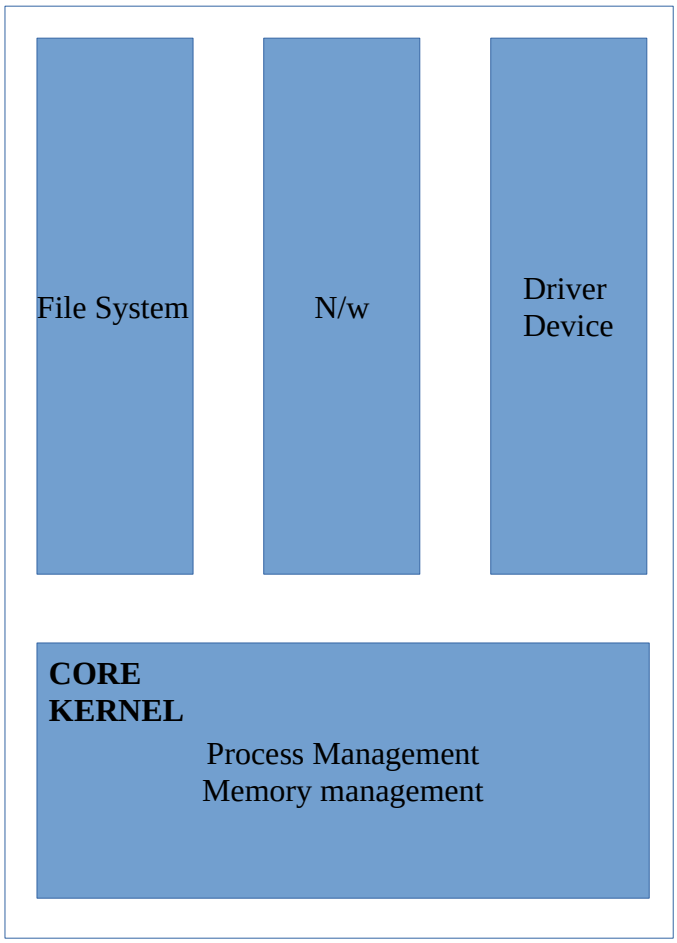
**Types of Kernel:**

**Monolithic Kernel:** All services are built into single package. It has single address location. If one of the service gets corrupted, entire OS gets corrupted. Maintanence uses more resources. Execution speed is faster. Ex: Unix/Linux/ Windows

| |
|---|
| Driver devices |
| N/w |
| File system service |

| | |
|---|---|
| **CORE KERNEL** | Process Management & Memory management |

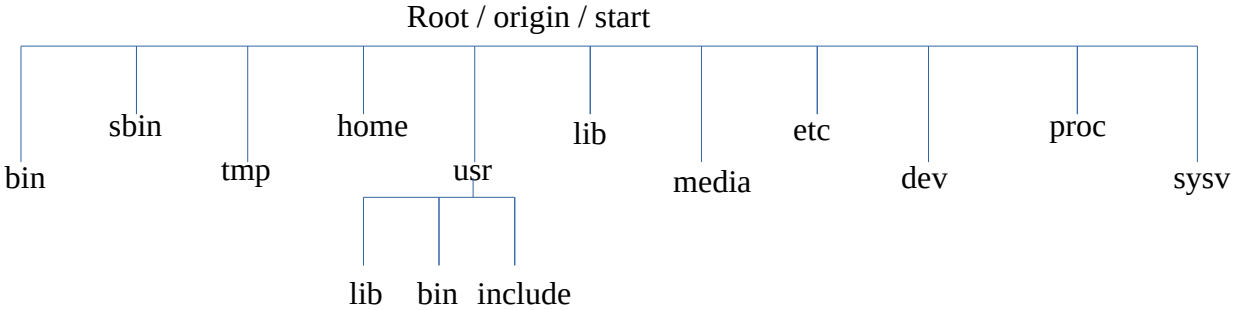
**Microlithic Kernel:** All services have separate address. If one service gets corrupted, it will not affect the working of Kernel. Maintanence uses less resources. Execution speed is low. Ex: Mac OS/ QNX/ Symbion.




**Why we need file system?**
If we upload something to microcontroller then we need to write a code in some programming language, while if we want to upload something to a pen drive(having file systems) then we do not need any programming language. Hence, if our OS can read file system then file transfer is easy. In Linux OS => there are 50+ file system.
**/lib/modules/5.4.0-84-generic/kernel/fs ->** Directory where file system are present

**Linux File System:**

- **Root File System ->** Root File system is a service that manipulates memory, uses memory more efficiently in order to organise files, directories in a proper structured manner in the memory. It is a service where all other file system service have attached during kernel bootup time. It is laid out as hierarchical tree structure format.



**bin** -> low level system utilities that contain software interfaces that interacts with the hardware.
**sbin** -> It contains super system services used by system administrator.
**tmp ->** It is a temporary folder.
**home ->** Every user has got personal file space. Home is a personal file space.
**lib ->** It contains lots of library function used for low level services. Kernel bootup time libraries are used from this folder.
**media** -> Media folder is called as extended root partition.
**/usr/lib** -> It contains lots of library function used for high level application.
**/usr/bin ->** It contains high level services/ applications.
**/usr/include** -> It contains all user header files.
**etc** -> It contains all Linux configuration information.
**dev** -> All Linux device files are in this folder.
**proc** -> It is a logical file system created during Kernel bootup time. It is available since begining of Linux OS. A proc fs creates lots of Kernel data structures shown as files to the user.
**sysv** -> The information related to all kind of devices connected to processor is given by sysv.

**Basic IO calls:**

-> operates on all files(i.e. in virtual memory and in physical memory).
-> file categorized into:
- -> Normal files
  - -> text files(.txt, .c)
  - -> binary files(.out, .gif,.jpg)
- -> Special files
  - -> pipes
  - -> fifo
- -> Device files
  - -> Character device file
  - -> Block device file

Applications is making use of device invokes appropriate driver in the Kernel space and driver on behalf of application interacts with the respective device.

**#include<fcntl.h>**
**1) int open(const char * pathname, int flags ,mode_t mode);**
Arg1: complete path file, file name
Arg2: operation to be performed
- O_RDONLY
- O_WRONLY
- O_RDWR

- O_TRUNC
- O_APPEND
- O_CREAT
- O_NONBLOCK

Arg3: modes -------Permission

ex:
int x = open(“/Desktop/folderX/file.txt”,O_RDONLY, 0666);

**Types of blocks:**

- Boot block per storage device
- Super block -> statistic info of number of files, number of FS.
- Inode blocks
- Data blocks

Each file has one inode structure:
struct inode
{
type
size
name
perm's
* ptr to data block
* file operation structures
}

- Open >> [Kernel Space]VFS(file management) >> device driver(block hard drive) >> [user space] create inode structure >> create file object structure which stores the address of inode >> update the table with file object starting from index 3 and can go upto 1023.
- All open files in Linux OS are refered by a non negative integer number called as file descripters(FD). FD is the index value that is returned by the FD table. To maintain an opened file, kernel has to create and maintain multiple data strucutres(inode,file object, FD table)
- System calls also callled as API's (Application programming interfaces).
- Open system calls return FD on success and return -1 in failure.

#include<fcntl.h>
**2) int creat(const char * pathname, mode_t modes);**

success = fd failure = -1

**3) int close(fd);**

success = 0 failure = -1
closing the file descriptor

File descripter is used in all subsequent file operations. Ex: read, write, close

#include<unistd.h>
**ssize_t read(int fd, void * ptr, size_t nbytes);**

success = no. Of bytes read from file failure = -1
read system call is used to read data sequentially from the file.

#include<unistd.h>
**ssize_t write(int fd, void * ptr, size_t nbytes);**
write system call is used to write data sequentially to the file.

**Validate read request:**
Most of the time read system call returns less number of bytes than actually we made request for.

- If file size is 60 bytes and we are requesting for 100 bytes of data, read returns 60 bytes.

File opened with write only flag but we are making a read request will cause failure and will return -1. The second argument pointer in read syntax pointing to an address that is outside program memory segment will also cause failure and will return -1.

**Validate write request:**
If we are writing some data to any embedded devices that have very small storage so there might be possibility that it is having less storage compared to data which is going to be written.
Validate the write request call when disc space filling up make the request accordingly.

File opened with read only flag but we are making a write request will cause failure and will return -1. The second argument pointer in write syntax pointing to an address that is outside program memory segment will also cause failure and will return -1.
*** Special flags(O_TRUNC, O_CREAT, O_APPEND) must be used ORing with normal flags.

**Current file offset position:**

- Every opened file is assosiated with current file offset value, which is a non negative integer number.
- It is a account or measurement that measures from the begining of the file.
- Generally, every read and write operation begins at current file offset position.
- Bydefault, current file offset is set to 0 initially(except in case of append).
- Current file offset causes to be increased by number of bytes successfully read from a file or number of bytes successfully written to the file.

**lseek:**
lseek(int fd, off_t nbytes, int whence);

current file offset position -1

- lseek operation is used for repositioning of the current file offset.
- Interpretation of lseek operation depends on third argument called as “int whence” argument.

**Whence** -> SEEK_SET, SEEK_CURRENT, SEEK_END

**FD Table:**

- FD table contains all opened file information, at each entry of fd table.
- At each entry it has the base address of the file object and that file object points to inode structure.

| | |
|---|---|
| 0 | STD Input device |
| 1 | STD output device |
| 2 | STD error device |
| 3 | |

- When opened first file from the program, 0,1,2 are filled by OS and least avaiblable file descripter is return to the application/user.
- FD’s are used in basic IO calls.
- C library functions internally invokes basic IO calls.

**objdump -d ->** It divides the executable file into sections and list onto the screen.
**objdump -s** is also an another option.
**size filename**
**nm filename**

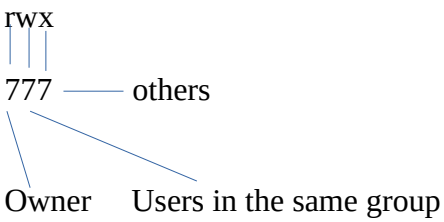
**Redirecting the output:->** printf is used to undertand the code flow. Redirecting the output can be done using (./output > filename). Also, it can be done using dup2(), system call.
**int dup(int fd):->** dup system call is going to duplicates the file descriptor. OS will allocate least available file descriptor from the fd table.
**int dup2(int fd, int fd2)**
**getdtablesize():->** It will print maximum fd table size i.e. 1024.

From the user space accessing inode object information present in Kernel space:

- ls -l
- stat() & fstat()

Each and every file weather an opened file or an existing file will have a structure called as stat structure and that structure maintains "ls -l" kind of information for a file.
**>> stat(const char * name; struct stat *buf)**
**>> fd = open()**
**fstat(int fd, struct stat * buf)**
The moment stat function is getting executed, it goes to the first argument and goes to files respective stat structure, fetching a structure of information and copying into the secong argument i.e. address pointed by the pointer supplied by the user.
fstat() system call is also doing the same as that of stat() system call but condition is file should be opened.
>> **stat -fc %s filename** -> size disc of block
>> **ls -i filename** -> File indode
>> **ls -l filename ->** file size
Linux OS provides a facility for sharing of opened files among multiple applications(processes), where a common inode object is shared between multiple applications.

**Permissions:**




Permissions is a sequence of three octal digit number where each octal digit number is talking about permissions either by owner or either by users in the same group or either by others.
Multiple user to generate multiple workstation.

>> **chmod 666 filename ->** It can change the mode of file.
>> **int fcntl(int fd, F_DUPFD, int index) ->** will work same as dup2()
fcntl is a system call is used to perform various control operation on opened file. It can be used to change file properties. fcntl system call depends on the 2nd argument >> fcntl(int fd, int cmd, optional)
cmd -: F_DUPFD, F_GETFD etc.

**LIBRARY:**

A group of precompiled object is called library.
**Need of library:** In a programming language we may require a block of code to be used again and again. In order to avoid rewriting of code several times. The code is created into a package and package is called as library.

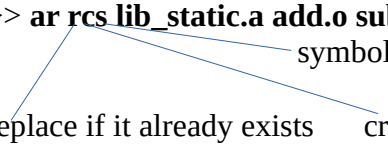
Library is not an executable file. Library can be used at ->

- run time(Dynamic Libraries)
- compile time(Static Libraries)

**Static Libraries:**

A static library is a precompiled object file that contains symbols. These symbols are used by program to execute and perform the task. Static library used by static linker at compilation time. It will link precompiled object file of the library with the target application object file and generates final executable file that is ready to get loaded and executes. Static files have extensions '.a'.

>> **ar rcs lib_static.a add.o sub.o**




>> **gcc four.c -o four -L. lib_static.a**

Static linker links the static library named as lib_static.a with application target object file and creates final executable.

**Dynamic Libraries:**

Dynamic library is a programing concept with special functionalities linked to your executable during program execution time(run time). Dynamic libraries are also called as shared libraries. Every program can use dynamic libraries inorder to avoid rewriting of code several times.

**Working of dynamic libraries->** Dynamic libraries are linked during execution of final executable. But actual linking is taking place when dynamic library is copied to the programs memory segment in the main memory.

Standard location of libraries on file system:

-> /usr/lib

-> /lib

-> /usr/local/lib

We can also store library functions in the non standard location of library function. We can create dynamic libraries and can store in non standard location of file system. For that we need to use environmental variable.

>> **gcc -c add.c sub.c -fpic**

-fpic: pic stands for creating instructions as position independent code. If a program one is executing and requires dynamic library '.so' file then addresses of library file should be shown in such a way that dynamic library '.so' file look like a part of programs main memory when library is appended to executable. Say another program running may require the same dynamic library then again library should be shown as a part of program's memory.

>> **gcc *.o -shared -o lib_dynamic.so** : -shared is used to create dynamic library, .so is extension of dynamic library.

>> **gcc four.c -o four -L. lib_dynamic.so**

**Static code analysis:** Static code analysis is the process of detecting errors and bugs in the program source code before the program is being run. Analysis is done on set of code by using some coding standards. These kind of analysis are helpful to identify loop holes and weaknesses in the program that might be harmful. Analysis and examination is done on stationary code therefore named as static code analysis.

**splint:** It is a static code analysis tool used to identify programming errors and bugs in the source code.

Programming errors are of different types like syntax error, type def error, usage of undeclared variable, usage of undefined function etc.

For a programmer its always good to have a error free program before it could be executed. splint tool is improving program performance.

**Clang:** clang is a C and C++ compiler, compiled by C++. clang is based on low level virtual machine. Low level virtual machine contains set of compiler techniques. LLVM is internally a library that is used to build binary machine code and this is used for internally compilers framework.

>> **clang -Wall filename.c -o objname**

- clang is very fast when compared to gcc.
- clang provides extremely clear diagonisitc messages and warnings.
- clang has slightly better performance when compared to gcc but still gcc is widely used.
- gcc supports many development environments.
- clang supports very few development environments.
- gcc is legacy compiler.

**Dynamic Analysis Runtime:**

Dynamic analysis is done on peice of software that is executed or in execution.

**gcov(gcc program coverage tool):** It is a open source software and gcov is used along with gcc. On analysis gcov tool will determine untested or unexecuted part of code. And also determines number of times a particular line has executed.

>> **gcc -fprofile-arcs -ftest-coverage filename.c -o filename**

- Compiling the source code with flags -fprofile-arcs and -ftest-coverage.
- -ftest-coverage flag causes gcc to add code coverage and statistics coverage information to binary executable.
- Compiling the source code with special flags will add extra information to the binary and also records the number of times the lines have been executed.
- Running the executable file -fprofile-arcs flag causes to create '.gcda' output profile file.
- >> **gcov filename.gcda**
- Gcov tool performs line coverage and function coverage.

**lcov(graphical front end tool for gcov):** lcov tool graphically represents gcc program coverage tool. lcov tool generates html file for the profiling output generated by gcov tool.

- **lcov --capture --directory . --output-file two1.info** : lcov is capturing the profile information generated by gcov from current directory and storing in '.info' file.
- **genhtml two1.info --output-directory . -o newfile**: genhtml is generation html file from '.info' file.

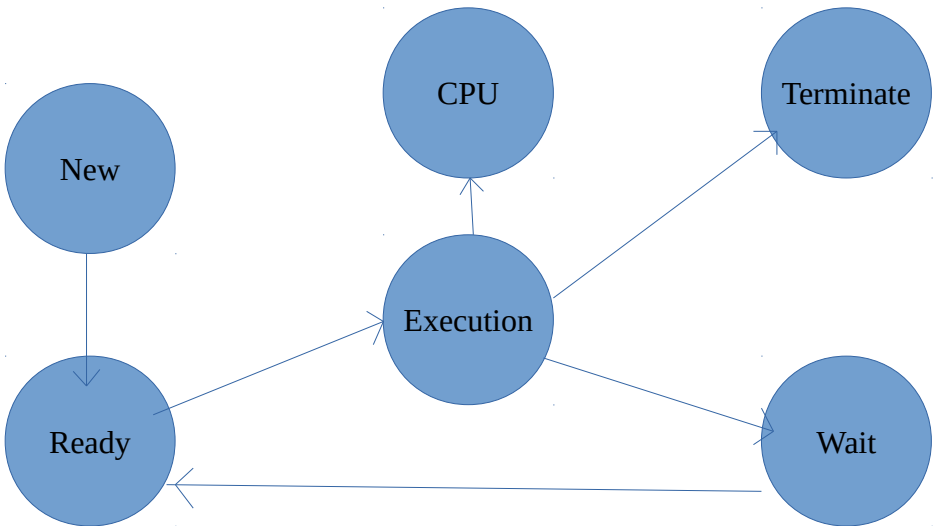
A program which is in execution is called as process. Once the program gets a CPU slice time it is becoming a process.

>> **ps -af** will show the process that are running from current terminal shell.

>> **ps -elf :->** ps(process states).

**Process States:**

When a process is newly created it enters a new state and then process moves to ready state, where the process is preparing to load and execute. Then moves to the execution state. If encountered any sleep or delay calls, it moves into wait state. Otherwise, directly enters into termination state and process terminates.



**Ready Queues:**

- All the processes are maintained as a linked list in ready queue. Scheduler will pick the best process from the ready queue and puts into the execution mode(based on terms of priority).
- Number of ready queues maintained by the operating system depends on processor architecture. If a system is dual core there are two ready queues i.e. ready queues are per CPU.

**Wait Queues:**

- Wait queues maintains linked list of process which are suspended due to blocking(scanf and sleep(5)) or delay calls.
- OS will mantain number of wait queues based on resources. Each resouce is having its on wait queues.(Resources: Keyboard, printer, mouse etc)

**Process Identifiers(PID):**

- When a process gets created, operating system allocates PIDs to the processors.
- PIDs are unique, no two process will have same PID.
- When a process gets terminated, the same PID is reused for other processes with delay timeout.
- There is a process whose PID is zero called as core Kernel process created during Kernel bootup time.
- Core Kernel process is responsible for creating init process in the user space with PID one.
- Init process whose PID is 1 inturn creating multiple processes in user space.
  >>**pstree**

**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, process state, pointer pointing at the process queue, CPU information(register, counter value, general purpose register, fd table information, page table etc.)). This PCB is a structure maintained for every process of the user space. Processes are isolated from each other. Each and every process maintains its own address space.

**Process Creation:**

pid_t **fork()** is a system call that creates new process. Linux OS provides a system call called as fork for creation of new processes from current task:

- For implementiing concurrent application software(multi tasking operation). All modern OS supports for multi tasking.
- fork() system call creates 2^n  number of processes, where n is number of times fork is called.
- fork() system call executes once returns twice. One return value to the parent process and next returns the value to the child process.
- fork() system call to the parent process returns child process's PID.
- fork() system call returns 0 to the child process.
- As parent process should keep track of the child processes, fork() system call return child process PID to parent process and returns 0 to the child process.

**vfork()**

**process termination:**

1. main() -> 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:**

**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.

**orphan process: -**

- A parent process terminates without taking care for the child process it is called as orphan process.
- init process = 1 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 :
  When child processes are in execution parent process is BLOCKED.
  When child terminates wait() call collect exit code of child process.
  wait() call returns error when there are no child processes are executing.
- wait() itself return the pid of child process.

**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.

**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 segments 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

**waitpid:**
**>> waitpid(PID, &status, NULL);**

**Disadvantages of fork system call:**

- When a fork() is called a process is created. Kernel has to allocate and initialize lots of resources to maintain a process.

**Threads:**
A process which is creating multiple threads is called multi threaded process. Multi threaded process in multi processor environment. Each thread can run separately a different task on different processor at same time. A thread can be defined as parallel context of execution. A multi thread is multiple parallel context of execution sets of instruction.

- Threads are called as light weight processes.
- Creation of threads, maintaining threads and destroying threads is far cheaper when compared to maintaining a process in terms of Kernel resources.
- Thread maintains own stack region.
- Each thread have own scheduling policy and priority.
- Each thread have own registers and they have their own signals called as sig block/ sig mask.
- It will have its own thread specific data.

**Linux OS Thread Model:** It uses one to one thread model. Linux threads do not have their own memory segments. Linux uses memory segments of parent process. Threads uses text, data, bss and heap of parent process and each and every thread maintain their own stack memory segment. Linux uses API called as clone to create threads.

Linux>> Clone>>Thread>>Gaming Application
In windows OS the software won’t run because the software is not portable.

**POSIX(Portable Operating System Interface):** IEEE standard organisation defined set of common APIs that are implemented accross POSIX compatibility operating system(Windows and Linux OS).

**Datatypes:**
Each and every thread is identified by unique thread ID and pthread_t is a datatype that denotes thread ID also called as thread object. Each and every thread have its own properties or attributes that talks about thread behaviour; How the threads should behave once thread is created.
>> pthread_t; // attributes/properties
>> pthread_attr_t; //attributes

```
#include<pthread.h>
pthread_creat(pthread_t *ptr, pthread_attr_t *attr, void (*start_routine)(void), void *arg);
```

- Error Code -> Numeric value
- EAGAIN -> When system does not have enough resources
- EPERM -> When system do not have required permissions.

>> **pthread_join(threadID, &status);**
Pthread_join wait for termination of specified thread. First argument waits for threadID to terminate. Second argument collects exit code of thread.

**Thread Termination ways:**

- When process terminates, thread will be terminated.
- When a thread returns from task function, it will be terminated.
- API – **pthread_cancel(threadID);** //return 0 on success and return -1 on failure
- pthread_cancel is a cancellation request to other threads used to terminate, other threads of the same process. pthread_cancel function terminates, whose thread ID is mentioned as a argument. It can also terminate the calling thread.
- **pthread_self();** -> returns calling thread ID
- **pthread_exit(void * retval);** //exit from current thread

***cat /proc/sys/kernel/threads-max -> will give maximum executilbe threads

**Initialization Routine:** These are routine/ functions, that executes only for once.
**Requirement:** using pthread, Implement initialization routines.

- **pthread_once(&once, myinit)** + once control variable of type pthread_once_t data type.
- pthread_once() function must be used in combination with once control variable of type pthread_once_t.
- pthread_once_t once_control;
- once_control = PTHREAD_ONCE_INIT;

*** when pthread_exit(NULL) is called from main function or main process it will wait for execution of all the threads.

**Pthread Attribute Category:**

1. Thread Stack Management
2. Thread Synchronization
3. Thread Scheduling Policy and Priority

By default stack size(8 MB) is provided by pthread library for each thread.
An application requires 20 threads to implement a gaming software where each thread requires 15 MB of thread stack size. If without modifying the attributes if gaming application is implemented, stack overflow may occur and application may crash. Change stack size from 8 MB to 15 MB. Attributes talks about thread behavoiur once thread created how this threads should get execute.

**Procedure of changing pthread attributes:**

1. Declare a variable of type pthread_attr_t myattr;
2. Initialize the attribute variable using >> pthread_attr_init(&myattr);
3. Call approprate attribute functions.(stack, synch, scheduling)
4. Create pthread using our attribute object. >> pthread_create(&tid,&myattr,task_fun,arg);
5. Destroy the attribute object finally when task is done.>> pthread_destroy(&myattr);

***pthread rules says that each thread should have separate stack address. Hence, before the creation of second thread using same attribute destroy first thread attribute using pthread_attr_destroy(&attr);***
>> **pthread_attr_setstack(&Attr,addr,size);**
>> **PTHREAD_STACK_MIN ->** 16KB

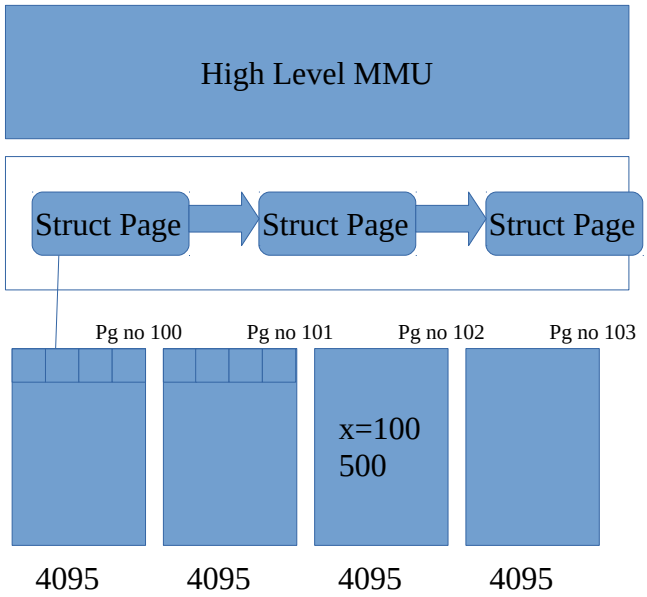
**Linux Kernel Memory Management Unit:**

Linux MMU is divided into two parts: High level and low level MMU

- High level MMU is independent of CPU architecture.
- Low level MMU is dependent on CPU hardware architecture.

Either it is Intel or ARM architecture, High level MMU will be same while Low level MMU will based on Intel and ARM architecture respectively.

Low level MMU runs during Kernel bootup time and initializes lots of Kernel memory management data structures and Low Level is also responsible for shifting the CPU with protected mode(Logical mode), where CPU sees memory as block(Where a block can also be called as Page, which is of type struct page).



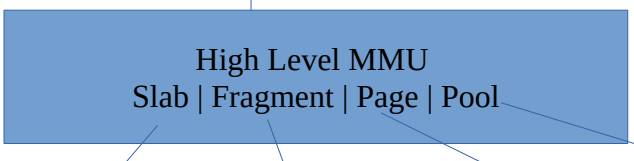

int x = 100;
printf(&x); --> pg no 102 + 500

---

User Space Process/Application

malloc()/calloc()

---




allocates small sizes memory which are continuous in memory.

allocates bulk size memory which need not to be continuous in memory.

allocates memory in terms of pages.

used by application of drivers, when they need fixed size of block of memory.

**Memory Manipulation calls:**

1. memset( >> void **memset(addr, value, size);**): memset function is used to initialize or fill a block of memory with particular given value for given size. Memset will jump to that address, starting from that address fill the given size and initializes with given values.
2. bzero( >> void **bzero(addr, size);**)

3. memchr(>> **void * memchr(addr, value, size);**): memchar function goes to address location and scan the address location for a particular value for a specified size and once found the matching value it will return the address of matching value.
4. memcmp(>> int **memcmp(addr1,addr2,size);**): memcmp function takes two addresses for comparison. It goes to those addresses and compare data byte by byte on those addresses untill it gets non matching character.
5. memmove(>> void * **memmove(dest, source, size);**): memmov function takes data from source address and copies on destination for specified size.
6. memcpy(>> void * **memcpy(dest, source, size)**); memcopy function takes same argument but memmov behaviour is guaranteed. In some cases, source address and destination address might overlap in such cases memmove provides a guaranteed behaviour. memmov function uses a temporary buffer. memmov first copies data from source to temporary buffer and from temporary buffer data is copied to destination. There is no risk even if overlapping takes place. memmov operation is slow but provides a guaranteed behaviour. In memcpy there is no temporary buffer, operation is fast but no gauranteed behaviour.

**Problem with user space and kernel space design:**

- Applications are abstracted from the hardware. Everytime applications have to go through kernel driver layer to access the hardware.
- To overcome such kind of situations, there is a concept called as user space drivers.

**User Space Driver:** User space driver means driver is still in Kernel space which expose the hardware information to the application i.e. it is showing the hardware information details to the application and allows the application to implement its logic on the hardware. To implement such kind of application there is a function called as **mmap function**.
**Anonymous memory mapping ->** mmap(); and munmap();

**mmap()** is a POSIX standard function which maps a Kernel memory or file memory or a device memory region to process address space.
>> void *** mmap(addr, size, protection, flags, fd, file offset);**

on success returns
pointer to mapped area

**addr -:** where in the processor address space we want to map.
**size -:** Size that we want to map.
**protection** -: read or write protection.
**flags** -: It will tell the nature of mapping(MAP_PRIVATE/MAP_SHARED).
**fd** -: Normal file or file descriptor.
**file offset**

***Providing addr as 0, asking the kernel to map at a free location in process address space.***

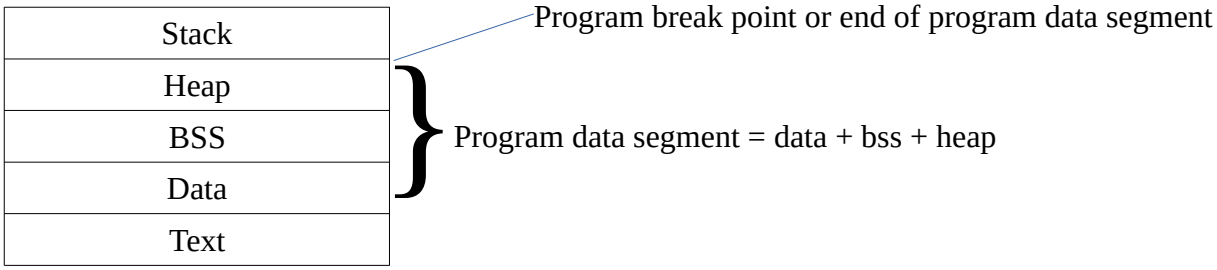
mmap();

do_mmap();

Ext3_mmap();

goes to memory convert physical memory file
into page + offset() takes data and copies into offset

**Program Break point and Program Data Segment:**




>> **void * sbrk(int value); -:** This function will increase program break point or end address of program data segment to new address(increases by 'value' bytes) and returns start of new address.
>> **int brk(void * end_data_segment); -:** On success return 0 and on failure return -1. brk function is used to change the location of program break point or end address of program data segment.

**Why can't we execute directly from hard disc?**

- Hard disc is divided into block by block, and RAM is byte by byte. CPU can understand accessing byte-by-byte. CPU can not understand block mechanism.
- Hard disc is slow campared to accessing from RAM.

Consider there are n number of process executing and RAM is fully occupied by n programs. Suppose there is one more program we want to execute then Kernel will look for segments of executing program which are not being used, then it will take that segment to the swap partition of Hard disc.

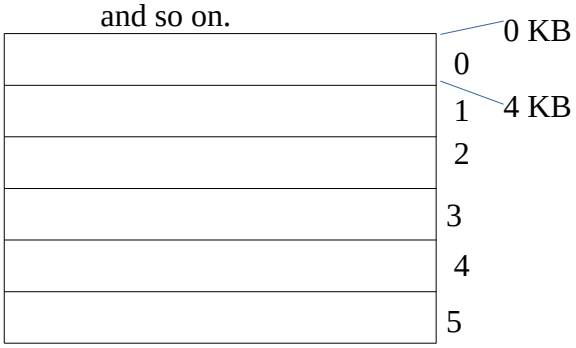
There are 2 challenges for the OS:

1. OS has to keep track which memory segments of a particular process are moved to swap partition.
2. OS should keep track of which memory segment of process are loaded into which locations of RAM.

To solve second issue, paging technique is introduced.

**Paging:**

- Process memory segments are divided into equal size parts, where each part is called as a page.
- Where each page size is 4KB, provided by low level MMU of Linux OS.
- Like process divided into equal parts, the physical memory(RAM) is also divided into equal size units where each unit is called as Physical frame or Page frame.
- The size of Physical frame should be equal to page size i.e. 4KB.
- Physical frames are numbered from 0 and goes upto RAM size.
- 0 KB is starting address $0^{th}$ physical frame and 4 KB is starting address of $1^{st}$ physical frame and so on.



- **Page table** or page frame relation table is a dynamic data structure maintained by the operating system in the PCB of the process that maintains the information of which pages of a process are mapped into which physical frames of RAM.
- **Are the number of pages of a process are fixed?:** During process execution the number of pages of a process varies:
  - When DMA calls are made.
  - When a function invocation takes place.
  - When a new thread is created.
  - When a process calls mmap function.
  - When a process uses shared memory.

- Pages of multiple processes are usually loaded completely into different physical frames of RAM.
- **Do the pages of multiple processes share same physical frame?:**
  - When a child process is created, initially uses same physical frames which are used by the parent process.
  - mmap(0,size,proctection,**MAP_SHARED**,fd,0);
  - shared memory -> IPC

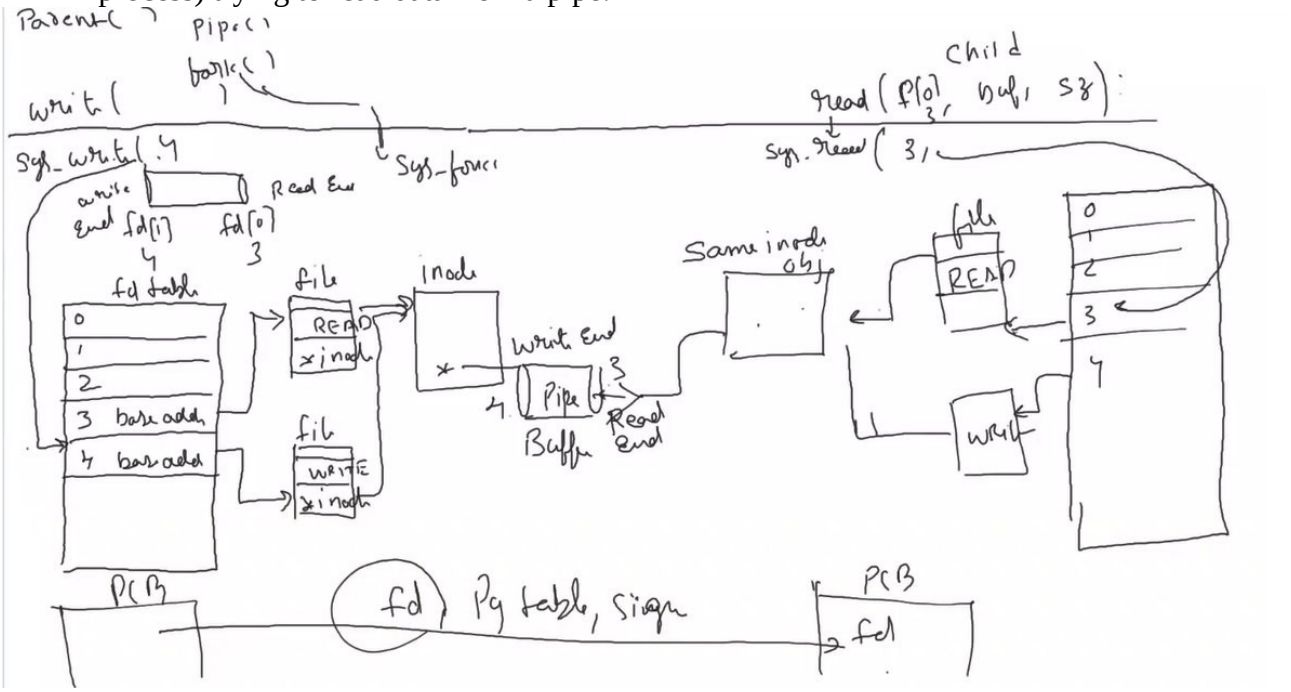
**Virtual Address Translation:**
When we use a variable in the program it is saved in the virtual address and mapped into the physical memory. When we want to assign some value to the variable the CPU takes the virtual address (page number + offset), the data given and the control signal(read or write) 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(addr, size);**: for example, we don't want operating system to swap particular pages of our process into swap partition of hard disc.
>>**mlockall();** : It is used to lock complete pages of process address space.
>>**alloca(addr, size);**:When we cannot generate memory from heap segment then we can use this function to generate memory from stack segment and free function is not required to free the memory as the function terminates stack is automatically erazed.

**<u>Inter Process Communication:</u>**
Linux OS system provides various communication techniques for process communication:
- Pipes
- FIFOs
- Message Queues
- Shared Memory
- Semaphores

**Pipes:** Pipe is a communication object that is used for communication between parent and child processes(related processes).
- Pipe is a file in the memory which has got buffer to send and receive data.
- Pipe has two ends, one is read end and one is write end.
- Data written on a write end, will be read in the same order from the read end of the file.
- Pipe is a serial communication device that permits unidirectional data transfer(Half Duplex).
- >>**ls | less :** ls writing to the pipe and less program reading from the pipe.
- From the application point of view to create a pipe, pipe system call is required.
- **int pipe(int p[2]); ->** on success gives 0.
- When a pipe system call is called successfully, a pipe gets created in the Kernel space.

- p[0] is used for read operation and p[1] is used for write operation.
- >> **cat /proc/sys/fs/pipe-max-size** -> to check new file size.
- For a normal file operation, read system call acts as a non blocking operation.
- In pipes operation, read system call may block, if there is no data and a process(child process) trying to read data from a pipe.




- Pipe system call returns -1 when there is no memory.

**Limitations of Pipe:**

- The major limitation of pipe communication is that it is used between only parent and child process communication.

**FIFOS:** They are also known as named pipes. FIFOS are used for process communication between unrelated processes.

- Command to create a FIFO is **mkfifo filename.(**It returns 0 on success and -1 on failure**)**
- FIFOs are registered as files in the file system with respective names.
- **cat > filename** will read and write data from FIFO.

**Types of communication:**

There are two types--

1. Connection oriented comunication mechanism:
   - Between sender and receiver, connection must be established before any transfer.
   - Read and write operation is possible only after connection establishment.
   - Ex: Phone calls, Video Chats etc.
2. Connectionless oriented communication mechanism:
   - No connection is established between sender and receiver.
   - Sender directly send to receiver without caring whether receiver is active or inactive.
   - Ex: Emails, mobile messages etc.

| PIPES | FIFOS |
|---|---|
| Pipes are used between parents and child. | Fifos are used between unrelated process. |
| Both are unidirectional and half duplex. | |
| Use same basic I/O calls-> read(),write(),open system calls | |
| Pipes are created as file in the Kernel space. | Fifos are created as file in the file system with names. |

| | |
|---|---|
| If a process gets terminated, all the kernel objects are also lost. | If a process terminated, Fifo still exist in the file system. |
| Pipes are created using pipe(); | FIFO are created using mkfifo(); |
| In a pipe, read() system call is a blocking call. | In fifos, both open() and read() are blocking calls. |

1. We need to write two programs, one server program and one client program.
2. We start first server program, it waits or blocks for client connection request.
3. Once client initiates a connection request to server, server has to accept the request.
4. Then client and server are ready for communication.

Server
1. Server creates server_fifo.
2. reading from server_fifo object.

Client
1. Client has to access server_fifo object (controlled by server)
2. Client writing to server object.
3. Once client write to server_fifo. Server read/receive the data and process the data.

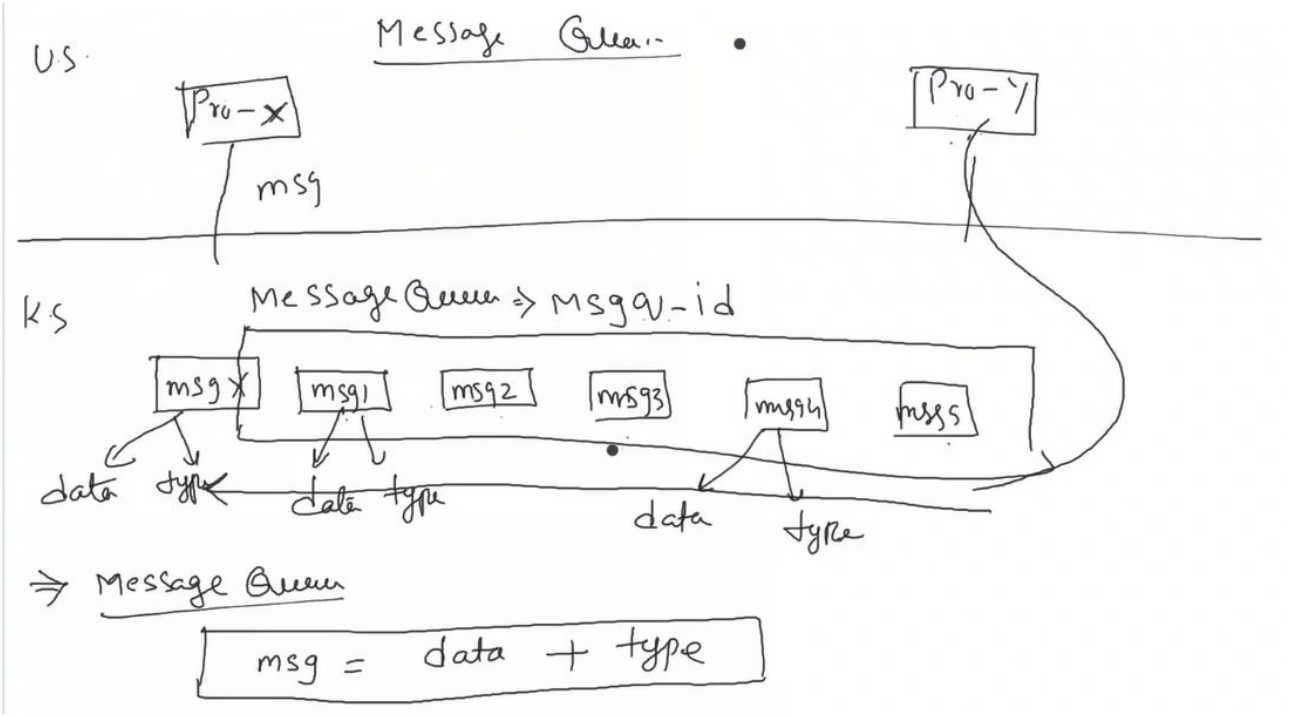
*** In real time applications implementation with respect to FIFO, first step is to check whether FIFO object is existing or not.
-> If fifo object exist, do not create a new fifo, just open and use it.
-> If fifo object do not exist, then create a new fifo using mkfifo and open then use it.

**System Five IPC Standards:**
1. Message Queues
2. Shared Memory
3. Semaphores

**Message Queues:** Issues faced during FIFO implementation is eliminated in message Queues. open() and read() block mechanism is eliminated in message queues.

| FIFO(Named Pipe) | Message Queues |
|---|---|
| Connection oriented mechanism. | Connection less oriented mechanism. |
| FIFOs are half duplex. | Message Queues is full duplex. |
| FIFOs uses basic I/O calls(open()/read()/write()). | Separate set of system calls used. |
| Stream based data transfer takes place. (Continuous flow of data) | Packet based data transfer. (Data divided into small units called packets) |

- Message queue is a linked list of message stored in the Kernel space.
- Each and every message queue is identified by unique ID called as message queue ID.
- Messages of the message queue is composed of data and type.




For each and every message queue, linux kernel maintains a structure struct msqid_ds that maintains, current status of message queue.

struct msqid_ds
{
	permission
	no. of msg's in msg Q
	no. of bytes in msg Q
	Pid of last process that has performed message send operation
	Pid of last process that has performed message receive operation
	time of last message sent
	time of last message received
}

>> **ipcs -q ->** tool or command, to check number of message queues, system wide.

**System calls:**

>> **#include <sys/msg.h>** is required to create message queues

>> **#include<sys/types.h>**

>> **int msgget(key_t key, int msgflags);** on success returns msgq id and on failure returns -1

arg1:

- key_t is a wrapper to int data type.
- Linux OS requires key, which is of type key_t data type, whenever applications are requesting for resources.
- key is like a name to the message queue.
- When process is creating new message queue, it provides a unique key.

arg2:

- int msgflags -> IPC_CREAT | 0640
- IPC_CREAT flag with msgget function will check for the given key if the message queue is existing. If it is not existing, create a new message queue in the kernel space. If exists then just use the message queue.

>> **int msgsnd(int msgid, void * ptr, size_t size, int flags);** is used to append message at the end of message queue. On success returns 0 and on failure returns -1.

>> ssize_t **msgrcv(int msq id, void * ptr, size_t size, long type, int flag);** is used to receive message from message queues. On success it will return size of the data portion and on failure returns -1.

>> **ipcrm -q msqid :** is the command to remove message queue.

>> **ipcrm -Q key**: is another command to remove message queue.

>> **msgctl(qid, IPC_STAT**(command)**, msqid_ds**(structure)**);**

**>> msgctl(qid, IPC_RMID**(command)**, NULL**(structure)**);**

*** Key is used by unrelated processes to access message queues created by other processes.

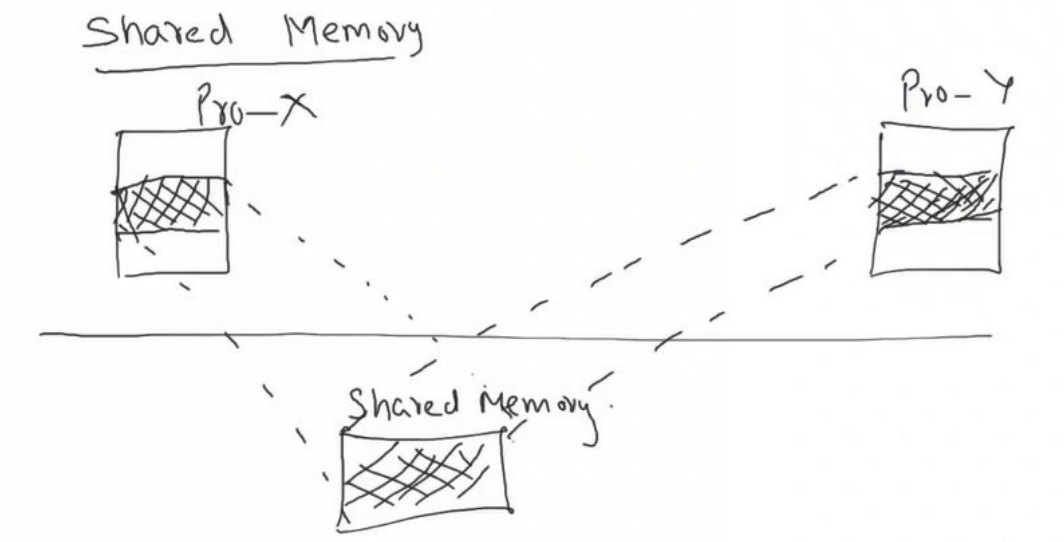
If the message queue is full and sender process wants to send more messages to the message queue, sender process by default gets blocked untill some space or a room is created because of receiver process.

Our requirement is we don't want process to block when message queue is full. Use IPC_NOWAIT flag as the fourth argument in msgsnd function.

Our requirement is we don't want receiver process to block on message queues when message with the type is not available. Use IPC_NOWAIT as fifth argument in msgrcv function.

**Fastest IPC communication technique?**

**Shared Memory:**



Shared memory segment allows two or more processes to access a given region of memory. Shared memory is fastest IPC communication technique among all. No system calls are issued, process need not to move from one memory location to another memory location for data transfer or data receive. Instead a given region of memory is attached to process memory segment.

- Shared memory is bydefault not synchronized. Programmer has to implement synchronized access to the shared memory segment.
- When server process is writting to shared memory segment, client process should not read from shared memory segment untill server process completes write operation.
- This is synchronized access to shared memory segment.
- Each and every shared memory segment is associated with a structure struct shmid_ds -> provides complete current status of the shared memory segment.

Struct shmid_ds

{

perm

shared memory segment size

last pid shm_at

last pid shm_dt

last time shm_at

last time shm_dt

ctime

.....

}

>> **ipcs -m** : command to check shared memory segments.

>> **#include<sys/shm.h> :** header files need to be included

>> int **shmget(key_t key, size_t size, int flag);** on success returns shared memory id and on failure return -1.

>> void * **shmat(int shmid, addr, flag );** Shared memory attach function takes shmid as the first argument and attaches the specified shared memory segment to the process address space. On sucess returns shared memory segment that is attached to the process and on failure returns -1.

arg1: shared memory segment to be attached.

arg2: where in process address space we have to attach shared memory segment(recommended 0).

arg3: flag SHM_RDONLY, if it is not read only then default read and write is possible on the shared memory segment.

Shared memory is created in physical memory of the RAM but shmid function attaches memory to the process address space and returns virtual address page + offset that is understood by your process.

>> int **shmdt(addr);** on success returns 0 and on failure returns -1. It is removing shared memory segments from the process address space.

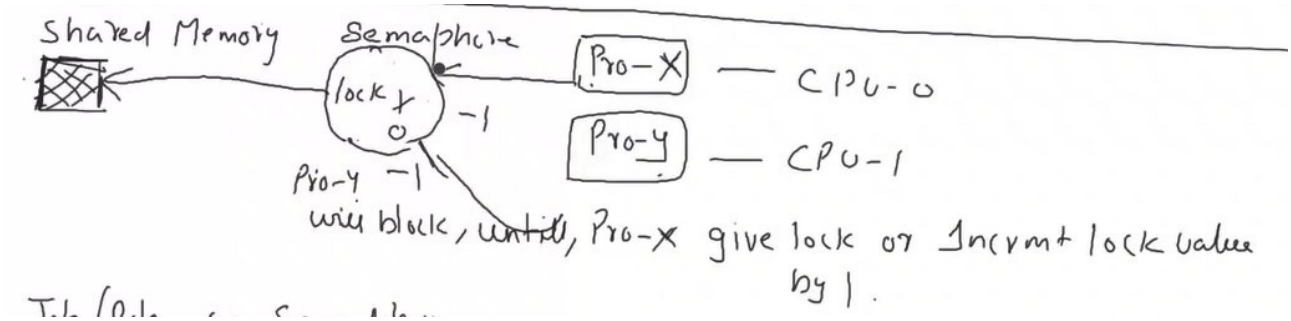
>> int **shmctl(shmid, IP_STAT**(command)**, &buf**(structure)**);** on success returns 0 and on failure returns -1.

**Semaphores:**

By default shared memory is not synchronised. To get the synchronised access to the shared memory between client and server, semaphores are used. Shared memory cannot be accessed by two process or threads at the same time. It has to be synchronized using semaphores.

**Process job/role on semaphore ->**

- Process or a thread decrement semaphore lock by one and takes the lock.
- Process uses shared resource(shared memory/global variable/global data structure/printer).
- Process or a thread increment semaphore lock by one and give back lock to semaphore.




**Application for server:**

- Creates shared memory.
- Creates semaphore.
- Server wants to read the shared memory.
- Semaphore value must be = 0.
- Server should also perform decrement operation on semaphore (-1).

**Application for client:**

- Client uses the shared memory.
- Client uses the semaphore controlled by server.
- Client then writes data to the shared memory.
- Also release the lock by incrementing the lock value by one.

>> int **semget(key_t, no. of semaphores to create, int flags);** on success gives semaphore id and on failure gives -1.

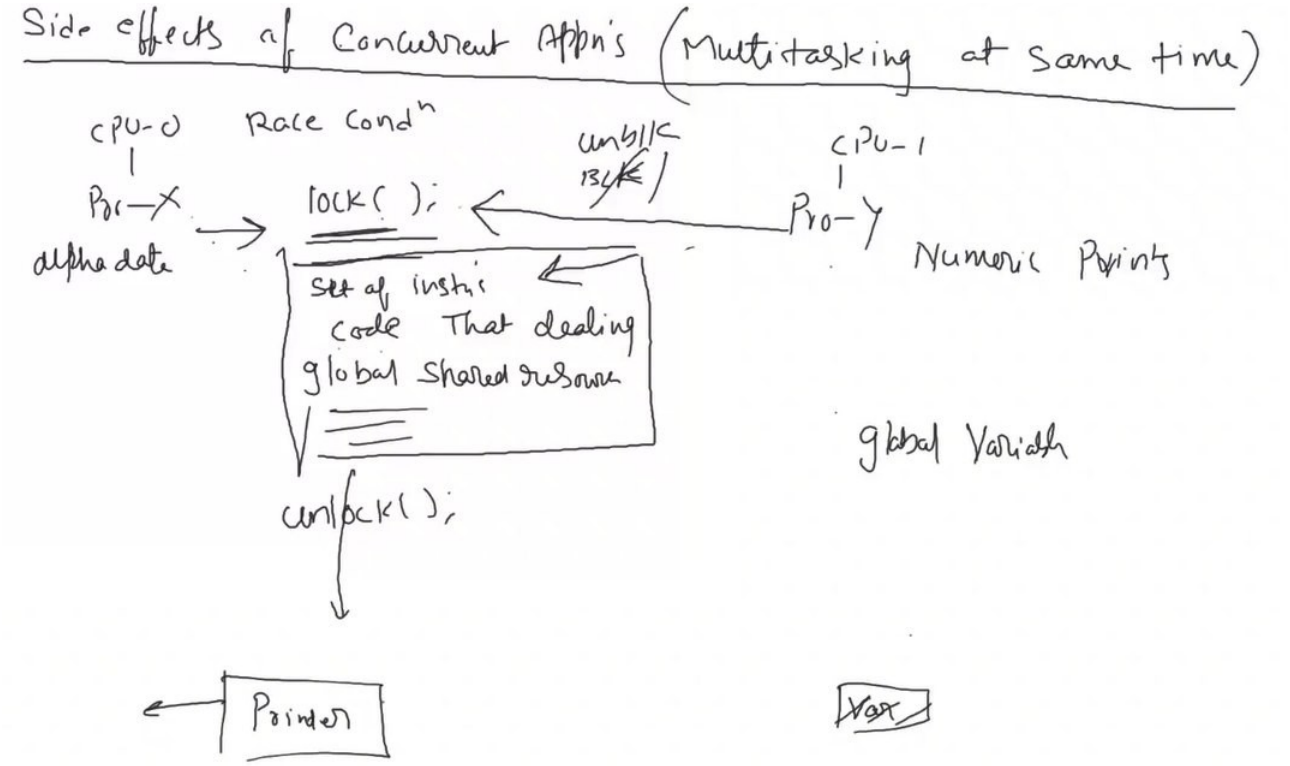
>> int **semctl(semid, semaphore number, command, value);** on success gives 0 and on failure -1.

Semaphore opeation can perform increment or decrement operation on semaphores lock.

>> **semop(semid, struct sem_buf *ptr, no. of semaphores);**

**Side Effects of Concurrent Application Programming(Multitasking at same time):-**

When two process or threads tries to access same global shared resource leads to race condition.




Critical Section: Set of instruction or code that is dealing with accessing of global shared resource(Shared memory, global variable, global data structure, a resource like printer).

To avoid race conditions over the critical section synchronization techniques are implemented.

Synchronization Techniques:

1. Semaphore
2. Spin lock
3. Mutex

Linux OS provides above three locking techniques.

Semaphore:

- Semaphore are optimised for non contension cases(No competition).
- Semaphore can not be used with interrupt context or interrupt handlers.
- No issue of process priorities.

Pthread Semaphore:

- Declare or create a semaphore object/variable.

sem_t mysem; -> mes_t is a data type, to declare semaphore variables.

- Initial semaphore obj/var.

  Sem_init(&mysem,0,1);

Initializing semaphore object with 1

All the threads of the process

can access semaphore variable

If value is 1, semaphore object shared between processes.

- Applying a lock

  sem_wait(&mysem);

- Unlocking

  sem_post(&mysem);

**Binary Semaphore:** Process on a semaphore -> Take lock and decrement it by 1, use resource, give lock and increment it by 1. A semaphore whose changes between 0 and 1 is a binary semaphore.

* A semaphore value can never be negative.

*** A semaphore whose value is greater than 1 is called as a **counting semaphore** used for protecting multiple global shared resources.

**Spinlock:**

- Contension 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.

   **pthread_spinlock_t my_spin;**

2. Initialization

   **pthread_spin_init(&my_spin**(address)**, PTHREAD_PROCESS_PRIVATE**(command)**);**

3. **pthread_spin_lock(&my_spin);**
4. **pthread_spin_unlock(&my_spin);**
5. **pthread_spin_destroy(&my_spin);**

**Mutex(Mutual Exclusion):**

- Mutex are synchronization locking techniques like semaphores with a usage count of 1.
- Mutex are similar to binary semaphore.
- One process or a thread can hold one mutex at a time.
- A process or a thread blocking mutex must unlock the mutex.
- A process or a thread can not terminate while holding a mutex object.
- Mutex can not be integrated with interrupt context.
- Semaphore are bydefault not initialized, we have to initialize semaphore lock value.
- But mutex lock variables when created by default they are initialized by 1.

Mutex Procedure:

1. Declare mutex variable.

   **pthread_mutex_t** my_mutex**;**

2. Initialize mutex variable (Dynamic and Static)

   **pthread_mutex_init(&my_mutex, &mutex_attr);** **pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZATION;**

3. Locking

   **pthread_mutex_lock(&my_mutex);** or **pthread_mutex_trylock(&my_mutex);**

4. Unlock

   **pthread_mutex_unlock(&my_mutex);**

Thread attribute talks about joinable or detachable property.

- Bydefault, the threads created using pthread are in joinable.
- Only joinable threads can be joined to the main process.
- Datachable threads cannot be joined using pthread join.
- When thread is terminated, resources and the memory are immediately removed from kernel space on its own.
- Parent process is not waiting for the termination of detachable thread.

Detachable threads can be created using two ways:

1. **pthread_detach();**
2. **pthread attributes**

**Context Switching:**

Switching of the CPU from one process or a thread to another. Generally, context switching occurs when kernel transfers the control of CPU from one executing process or a thread to another process or a thread.

- For this kernel has to save the process context(Context is process data + CPU registers value).
- Then kernel loads new process and executes.
- The process which was taken off the CPU, will execute when it gets next CPU slice time.
- Execution begin from where the CPU was taken off.(with help of context saved information and instruction IP register value)

**Scheduling Policies:**

- Round Robin Scheduling
- Complete Fair Scheduling

**Round Robin Scheduling:**

Round Robin scheduling algorithm used by computer system where each process will be assigned with equal proportions of CPU slice time in circular order. Each process is allowed to use CPU for a given amount of time. If job is not done within given time slice, CPU is given to next process in the queue for same amount of time. All the process will have same priority. Round Robin implements starvation free scheduling.

**Complete Fair Scheduling:**

Linux CFS does not assign CPU slice time directly to processes. Instead proportion of the CPU slice time depends on a factor called as NICE value. NICE value can increase or decrease the weightage of processes with respect to CPU slice time. NICE value will priorities the processes.

| -20 | 0 | +19 |
|---|---|---|
| Highest Priority | default | Lowest Priority |

Even a lowest priority process will get a CPU slice time therefore it is called as fair scheduling.

=> **taskset** by default comes with Linux distribution. taskset is a command or a tool used to set CPU core to particular processes for execution.

**nice(N.V.); =>** it is function used by Linux OS to change process priority.

=> & is a special background symbol that runs the operation in the backround.

=> policy = **sched_getschedular(0);**

**=>** ret = **sched_setscheduler(current process**(0),**Macro with which we want to change, structure);**

arg1: 0..setting policy for current calling process

arg2: SCHED_RR

arg3: struct sched_param

contains args/parameters that can used to change policy of the process

struct sched_param sp = {.sched_priority = 1};}

**Structure sched_param contains parameters or arguments that are used to change the policy of the process.

**SIGNALS:**

Signals are the events generated by operating system on some conditions. These generated signals are send to processes. Process on receiving signal has to perform some action. Action purely depends on signal it received.

=> **kill -l** will give all the signals can be used in Linux OS.

```
1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP      6) SIGABRT
7) SIGBUS       8) SIGFPE       9) SIGKILL      10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT
19) SIGSTOP     20) SIGTSTP     21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13
52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9
56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6  59) SIGRTMAX-5
60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2  63) SIGRTMAX-1  64) SIGRTMAX
```

- Out of these 64, 1-31 are some traditional signal.
- And 34 to 64 are realtime signal.

Different condition in which signals are generated:

1. Terminal specified key combination like ctrl+c, ctrl+z
2. When exception occurs.
   - Executing illegal instruction: x/0
   - Executing invalid address location.

   Execption conditions are stopping current process execution. Exception handler is a special function that deals with exception. Exception handler will find out the pid of the process that performed illegal instruction execution or invalid address memory execution. Exception handler then terminate the process by sending a signal called as SIGKILL.
3. Kernel can also send signal to process. For example: Kernel sends a signal SIGCHLD signal to parent process when parent process receives signal SIGCHLD. Parent understood that child terminated.
4. **kill signo pid :** A user can send signal to the process, with the help of command called as kill.

5. Using a system call we can send signal to process. **kill(pid,sig_number);**

**Delivery of a Signal:**

PCB

| |
|---|
| 1. Signal Disposition table |
| 2. Signal mask(Blocked signal Information) |
| 3. Pending signal information |

**1. Signal Disposition table :** It talks about signal behaviour. It is a table of 64 enteries. SDT maintains information, how to handle each signal.

=> Each entry will have either of following information.

| SIGDFL | SIGIGN | Signal Handler function |
|---|---|---|
| Executing the default behaviour | Simply Ignore no impact | Signal handler is a speed function defined by user to deal with signal |

>> **kill -sigNo. Pid**: kill command invoking kill system call.

>> **kill(Pid, -sigNo.);** : kill system call will invoke sys_kill inside kernel space;

>> **sys_kill(Pid, -sigNo.)**: sys_kill will look for PCB with matching PID.

Once after getting access to the PCB. Then it will access Signal Disposition Table and next uses signal number as the index to SDT table and execute the value in the respective entry.

For most of the signals default behaviour is to terminate. SIGIGN is no action performed by the process. There are two signals, signal no. 9 and signal no. 19 can never be ignored. When a program is executing illegal instruction like x/0 and a program is doing invalid memory access(segmentation fault). To stop those application or a process, Kernel has to terminate these processes by using SIGKILL(no. 9) and SIGSTOP(no. 19).

**Can a function can be invoked without using main function? -> Yes, Using signal disposition table and signal handler.

**Installing/Modifying a signal handler to signal disposition table:**

There are two functions to modify SDT:

1. void **signal(sig_number, signalHandler);**

   signal no. for which we want to modify the behaviour.

   It is a function that is to be used when a signal is generated.

2. int **sigaction(sig_number, struct sigaction *ptr, NULL);** On success 0 else -1.

Next step after installing a signal handler is to execute the application continuously. Only then we can check the behaviour of signal.

>> **structure sigset_t** -> deals with group of signals/multiple. Purpose of this structure is to maintain multiple signals information.

**System Calls:**

1. **sigaction();**

2. **sigprocmask();**

These two system calls allows the programmer to specify group of signals that are to be blocked for current calling process.

3. **sigpending();** It will give information of pending signals information for the current calling process.

>> **sigemptyset(sigset_t *set);** This function will make signal set to empty set.

>> **sigfillset(sigset_t *set);** This function will initialize signal set all signal to active.

>> **sigaddset(sigset_t *set, sig_num);** Individual signal added to group of signals.

>> **sigdelset(segset_t *set, sig_num);** Inividual signal delelted from group of signals.

>> **sigismember(sigset_t, sig_num);** Check particular signal member.

**2. Signal mask(Blocked signal Information):** Kernel inside a PCB maintains a variable signal mask. It is a 64 bit variable maintained by Kernel inside PCB. It will talk about group of signals that are actually blocked for current calling process. If sig_num is blocked, again sending same/blocked signal to process, delivery of signal to process is delayed[Pending Signal]. Untill, signal is removed from Blocking state(signal mask).

>> **sigprocmask(cmd, sigset_t *new_set, sigset_t *old_set);** This function is going to change process signal mask. It can also be used to retrieve blocked signal information.

If a signal is sent to a process and the signal is blocked and again same signal is sent to the process, the delivery of the signal gets delayed and enters into the pending state untill the signal is removed from the block state.

>> **sigpending(sigset_t *ptr);** This will list pending signal information for the current process.

struct sigaction is predefined in <signal.h>

```
{
        void * (*sa_handler)(int);
        int sa_flags;
        sigset_t sa_mask;
}
```

**COMPUTER NETWORKS:**

A set of computers integrated together is computer network. It can be of two types : WAN and LAN.

WAN:

1. Covers large geographical area.
2. WAN consists of number of interconncected nodes.
3. Transmission of data from any device will be routed through this interconnected nodes to specified destination.
4. Node can be a server, router, switch. Router and switch are networking deivces that connects one or more computer to other network.

WAN can be implemented in two ways:

| Circuit Switch Network | Packet Switch Network |
|---|---|
| A dedicated path is established between sender and receiver through nodes. | It follows networking protocol, where messages are divided into small packets called as network packets. |
| The path is a connection of sequence of physical links between nodes. | These packets are transmitted through digital network. |
| Data transmitted from source point, routed through this physical links to destination device. | Data transmitted via radio waves and electrical signal. |
| Ex: Telephone. | Ex: WiFi, ethernet devices. |

LAN:

It is a collection of computer network devices that are connected together in one physical location, such as office, school or home. Internal data transfer rates of LAN are faster than WAN.

**Internet Protocol Address:**

I.P. address are used to identify a node in the network and also used for communicating with particular nodes.

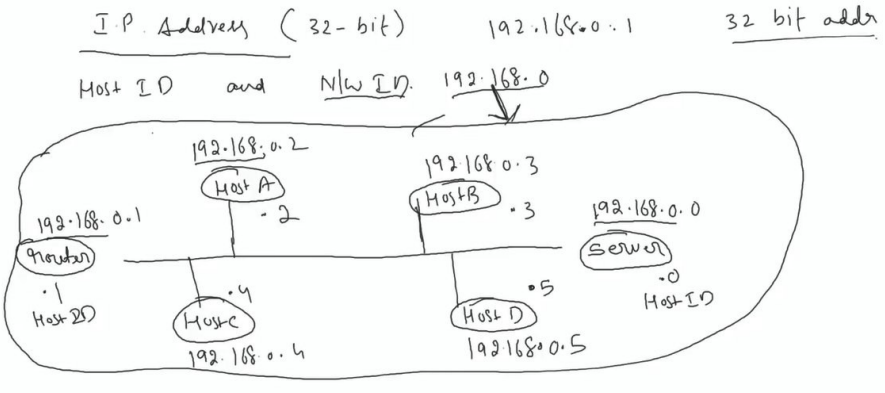
I.P. versioning:

**IPv6** **IPv4**

128 bit hexadecimal address 32 bit hexadecimal address

Better performance and more feature

Provides large address scope.



**MAC Address:**

These are physically attached to the devices. These are provided by the vendor or manufacturer. These are also called as hardware address.

| MAC Address | IP Address |
|---|---|
| MAC address are used to identify devices locally. | IP address are used to identify device globally. |
| 12 – digit number | 32 – bit or 128 – bit hexadecimal |
| They can't be changed | They cab be changed at any time. |
| These are also called as physical addresses. | These are called as logical address. |

When a data or a packet is sent over the network both these address are used.

**Network Protocol Layer:**

| |
|---|
| Application |
| Presentation layer |
| Session layer |
| Transport layer |
| Network layer |
| Data Link layer |
| Physical layer |

Network protocol layers are organised as series of layers. Reason for developing protocol as layers is to reduce the complexity of design. Each layer is providing functionality to other layer.

<u>Application layer:</u>

- Application layer provides an interface through which we can interact to any application to start data transmission.
- All the application protocols are present in this layer. For ex: HTTP, HTTPS, FTP etc.

<u>Presentation layer:</u>

- After application layer data is sent to presentation layer.
- It will check in which format data is to be transmitted to receiver.
- It will also provide encryption and decryption for data security.
- Data compression to reduce the size

<u>Session layer:</u>

- After presentation layer data is sent to session layer.
- It establishes a connection session between sender and receiver untill receiver receives all the data.

Transport layer:

- From session layer data is sent to transport layer.
- TCP(Transport control protocol) is a connection oriented and UDP(User datagram protocol) is connection less.

Network layer:

- Data is now transmitted to network layer.
- It divides the data into smaller packets and these are called as network packets.
- It will hold IP address of source and destination.
- A network layer is also responsible for routing operation.

Data Link layer:

- Data link layer will look for errors in the data and removes the errors and send an error free data or transfer error free data.
- It also maitains data flow transfer speed between sender and receiver.
- If sender is sending at 10 mbps and receiver is receiving at 5mbps then less data may be received, data may be lost, traffic may increase etc.
- Also doing physical addressing job.

Physical layer:

- Physical layer getting data from data link layer and converting into bits 0's and 1's.
- This will decide whether the data is to be sent via wired or wireless medium.\

**Socket System Calls:**

Socket system calls supports many network communication protocols : TCP, UDP, FTP, POP, HTTP

Socket system calls are generic in nature as they are going to support many communication protocols. Size is used as a key argument in socket system calls. The first step of implementing client-server model application is to create sockets on the both the sides. Sockets are end addresses or final end point in a communicating programs to which the connection is established. Sockets are internally special type of file. Each and every socket have these parameters:

1. Protocol used.
2. Source IP address 32 bit(N/W byte order).
3. Destination IP address 32 bit(N/W byte order).
4. Source port address 16 bit(N/W byte order).
5. Destination port address 16 bit(N/W byte order).

Destination IP address will decide the client IP address and Destination port address will identify to which destination application the packets is to be sent.

**Server:**

>> **socket(int family, int type, int protocol);**

family => Ipv6/Ipv4, AF_INET -> Address format internet protocol.

type => characterisitics of communication -> connection oriented (SOCK_STREAM) or connection less(SOCK_DGRAM)

On success, it returns a positive integer value and on failure returns -1.

**socket address structure:**

sockaddr_in

{

sin_family; //AF_INET

sin_port; //16-bit N/W byte order

struct in_addr sin_addr; // 32-bit IP addr in N/W byte order

}

>> **bind(int sockfd, struct sockaddr *anyaddr, int sockaddrlen);** is to assign the server side socket with well known IP address and source address. Server side socket with the well known IP address and port number after bind system call server socket has protocol + source IP + source port number therefore server socket is said to be half binded.

>> **listen(int sockfd, int len);** is used by the TCP servers which specifies number of client connection request going to serve. If the server is busy length may increase.

>> **accept(int sockfd, struct sockaddr *cliaddr, int lenofcli_sockaddr *len);** is waiting for client connection request and fetches the information of first client connected information. And then creates socket of same type of same family and assign a file descriptor to it and called as client socket or connected socket, whcih is used for data transmission.

** Accept is a blocking call that puts your server to wait untill client makes a connection request.

Client:

>> **socket(family, type, protocol);** these parameters will be same as that of server.

>> **connect(int sockfd, struct sockaddr *serv, int lenofser_sockaddr);** client uses connect system call to establish connection with server side socket. Connect system uses TCP system calls to make a connection request to the server.

CPU -> Big Endian (Motorola/Power PC CPU's)

Little Endian(Intel/AMD)

-> computer store data as bits in memory

-> Endianess to how integer data gets stored in the memory.

Suppose data = 0x12345678

**Big Endian /**

| | | | |
|---|---|---|---|
| 0x12 | 0x34 | 0x56 | 0x78 |

**Little Endian**

| | | | |
|---|---|---|---|
| 0x78 | 0x56 | 0x34 | 0x12 |

In network there are different machine, with different CPU's few could be few CPU's will follow little endian and few follow big endian formats.

```
bzero(&servaddr, sizeof(servaddr));


servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr =
inet_addr("127.0.0.1");//htonl(INADDR_ANY);
servaddr.sin_port = htons(8000);


port num 16

2 pow 16          (0 - 65535)

8000 port addr

8080    testing purpose....
-------------------------------------------------


htons(16 port num);  // convert 16 bit  host order to 16
network byte i.e big endian format addr.....

htonl(32 IP ADDR);  //// convert 32 bit  host order to 32
network byte i.e big endian format addr.....
```

>> **ntohs();**
**>> ntohl();**
This function converts string dotted decimal address into network byte order.

<u>127.0.0.1:</u>
Loop back internet protocol address, called local host used to establish, IP connection on the same machine. Every device will have this address 127.0.01
>> **ifconfig** used to check IP address
-> In reality packet will never reach the network. Packet is in the loop in NIC(Notwork Interface Card). Used for testing and development to TCP internet path. Loop back address allows the device to send and receive its own packet.

| **TCP** | **UDP** |
|---|---|
| TCP -> transport layer | UDP -> transport layer |
| Establishes connection oriented communication | Establishes connection less oriented communication |
| Reliable service. | Not reliable |
| Retransmission(incase of data lost) | No retransmission |

| | |
|---|---|
| ACK, sends acknowledgement | No acknowledgement |
| TCP is slower | UDP is faster |
| TCP header size is 20-60 bytes | UDP header size is 8 bytes |
| TCP for connection establishment uses 3-WAY HANDSHAKING. And for connection closing uses 4-WAY HANDSHAKING | No Handshaking. |
| Byte streaming data transfer. | Unit packet data transfer. |
| Application of TCP: HTTP, HTTPS, FTP, POP | Application of UDP: DHCP, DNS, TFTP. |

**TCP 3-way hand shaking:**
First step is SYN packet to server. SYN is used to initiate a connection with server. Second step is, server has to open the port and should receive the client initiation request and responds with return confirmation (ACK). Third step is, Client inturn sends one more acknowledgement to server for accepting client request.

**TCP 4-way hand shaking:**
Firstly, Client will send FIN request to server and waits for ACK from server. Secondly, Server acknowledges client that it received connection close request. Thirdly, server sends a FIN connection close to client. Fourthly, client will sends acknowledgement for closing the connection.

**System call in UDP:**
>> **recvfrom(sockfd, char *buff, int len, flags, struct sockaddr * addr, int *addrlen);**
arg1: sockfd
arg2: ptr to buf, that rcv's data
arg3: length of buff
arg4: flags
arg5: addr is a pointer to sockaddr structure from which data is received
arg6: length of (sockaddr) structure

>> **sendto(sockfd, char *buff, int leng, flag, struct sockaddr *dest, int len dest);**
arg2: data to be transmitted
arg3: contains length of the buffer
arg4: flags
arg5: pointer points to destination address
arg6: length of the destination address

**Implementing UDP Client-Server Application connectionless oriented communication:**
In UDP client never forms connection with server. Client just create a socket and send just datagram to the server. Server program just creates the socket and bind the socket with the well known port address. As there is no connection from the client side program, server need not to call listen and accept system calls. After binding server will wait for a packet from client side. Client send a packet that contains the sender address, once the datagram packet is received by the server, server process the data and send the packet to the correct client.

**WireShark:**
Wireshark iis the open source network protocol analyzer tool is used to capture incoming and outgoing packets in the network in real time. This information is used for troubleshooting and communication protocol, used in software development process and also used for packet analysis. The purpose of the wireshark is to see what is happening around the network. It also provides the detailed information of the packets which are in movement around the network. It can inspect

hundreds of packet at a time. Wireshark is used to capture live packets and can examine the packets later. Wireshark tool automatically gets updated to the new protocol released in the market.
>> **cat /sys/class/net/*/address** It will give mac address.
Each Packet in WS interface:

1. Frame number -> Frame number is the packet number, it is order in which packets are received.
2. Ethernet -> MAC address of source and destination.
3. Internet Protocol version -> IP address of source and destination
4. TCP -> Port number of source and destination.

**Promiscuous mode:** Promiscuous mode will capture traffic going through all the devices connected to a network.
With wireshark we have two types of filter:

1. Capture filter: Before start of wireshark tool, we specify what traffic or packets we want to capture. Once set the capture filter then wireshark will ignore all other packets and just capture the packets according to our requirements.
2. Display filter