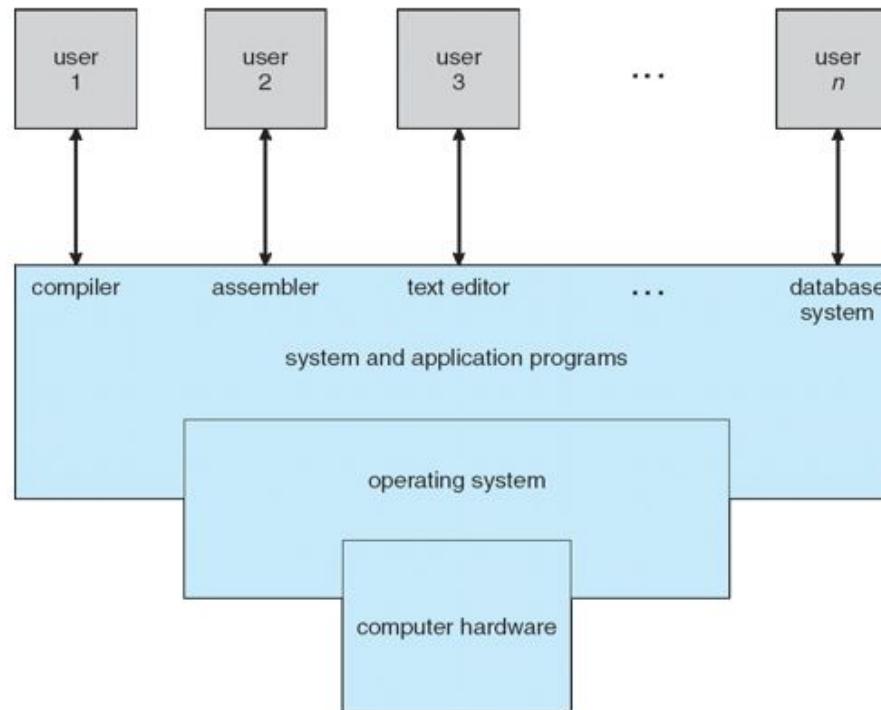


Linux OS & Systems Programming

What is an OS???

- A program that acts as an intermediary between a user of a computer and the computer hardware
- Operating system goals:
 - Execute user programs and make solving user problems easier
 - Make the computer system convenient to use
 - Use the computer hardware in an efficient manner

Four Components of an Computer System



Operating System Definition

- OS is a **resource allocator**
 - Manages all resources
 - Decides between conflicting requests for efficient and fair resource use
- OS is a **control program**
 - Controls execution of programs to prevent errors and improper use of the computer

Introduction to Linux OS

LINUX

What is LINUX?

+
x

- Linux is a family of open source OS.
- Linux source code is freely available.
- Users are free to download, modify and enhance the OS.

Linux OS components

The Linux OS components are

- Kernel
- Networking
- Security
- GUI (User interface)

- Kernel – The kernel is a computer program at the core of computer's operating system, and manages the hardware.
- Networking – Provides the feature to communication across different computers.
- Security – Is a mechanism of providing protection to computer system resources such as CPU, memory, disk and others. It provides access control for unauthorised user from accessing system resource, and hence providing security.
- GUI – The graphics version of Linux provides user interface

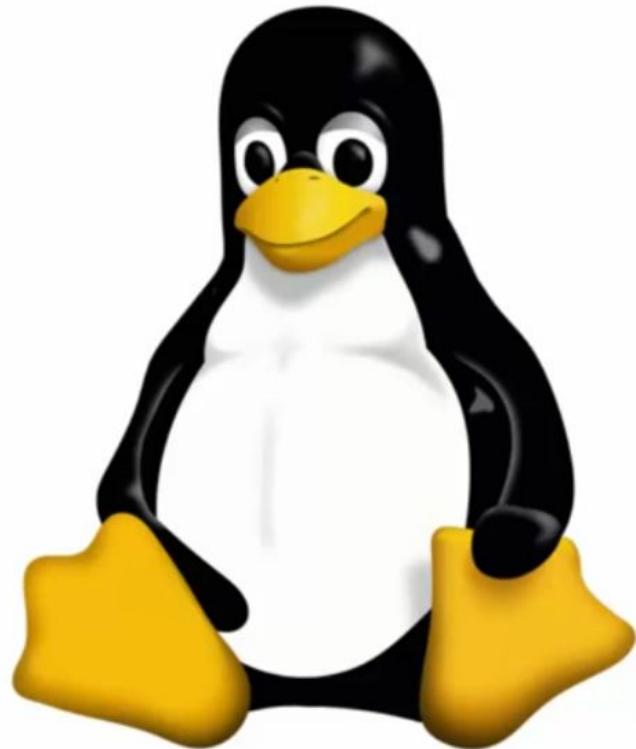
Linux Distributions

- Linux is not owned by any single organisation.
Different people and organisations across globe contribute for Linux development.
- Different Linux distributions compile the several open source features of Linux, and provide as a single Operating System.
- The different popular Linux distubutions(Distro) are Red hat, CentOS, Fedora, Ubuntu and many others.

How To Open The Terminal

```
> -
```





- How to open and close the terminal.

- Be able to open and close the terminal using graphical methods and keyboard shortcuts.

Coming Up Next ...

- Running your first terminal commands.

Our First Commands

- You will be running your **first commands**.

By The End

- You will have run your first Linux commands.
- You will be able to see a pattern in how Linux commands work.

Summary

- ✓ You have run your first Linux Commands.
- ✓ We run commands by typing them in the terminal and pressing enter.

Coming Up Next ...

- How commands are **structured**.

Terminals, Commands, and Shells (*Oh my!*)



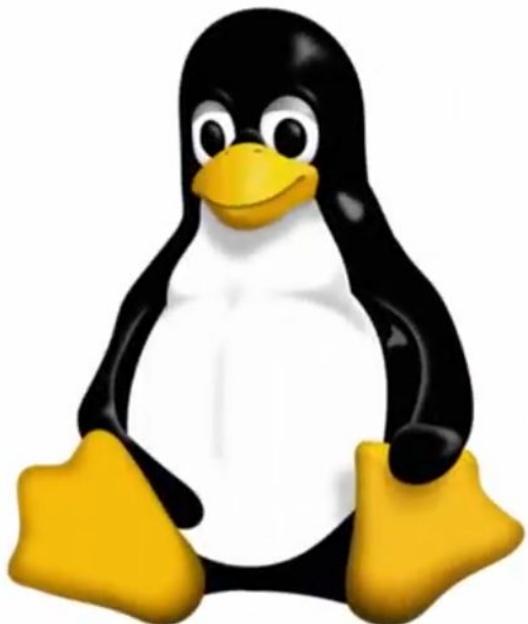
IMPORTANT



- You will learn the **difference** between the **commands**, the **terminal** and the **shell**.

By The End

- You will have a much deeper understanding of what's going on when you use the terminal.



Commands

VS



The Shell

RASPBERRIES

Commands
(Words)

RASPBERRIES



**Commands
(Words)**



The Shell

RASPBERRIES



**Commands
(Words)**



The Shell



Meaning

“Gift”

Commands
(Words)

“Gift”



**Commands
(Words)**



**English
“Shell”**



“Gift”

Commands
(Words)

“Gift”



Commands
(Words)



German
“Shell”





“Smoking”



Commands
(Words)



French
“Shell”



Meaning



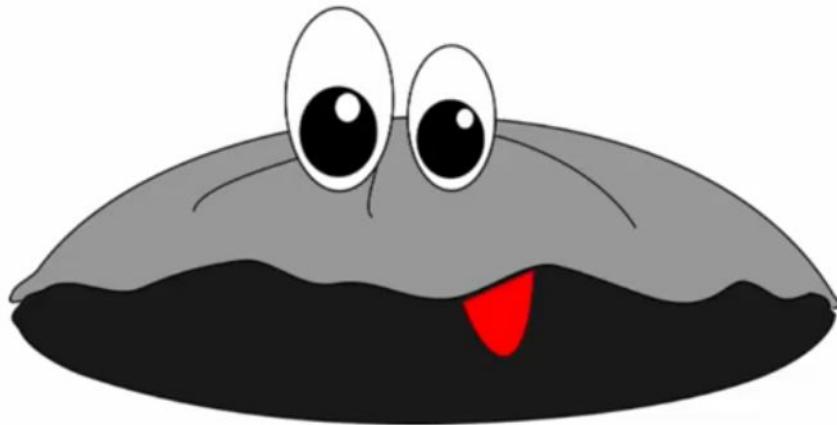
>

-



=





Summary

- ✓ Commands are just **text** you type in the terminal.
- ✓ Commands are **interpreted** by the shell.
- ✓ Different shells can interpret the **same text** in different ways.
- ✓ The **terminal** is the **window** to the shell.
- ✓ Bonus Point: Commands are **case-sensitive**.

Coming Up Next ...

- What commands **actually** are and how they are **structured**.

Understanding Command Structure

- Learn what commands **actually are** and how they are **structured**.

- Commands will begin to look like a **language** not random gibberish.

Summary

- ✓ Command = CommandName -options inputs.
- ✓ CommandNames need to be on the shell's search path.
- ✓ Commands operate on inputs.
- ✓ Options modify a command's behaviour.

Each Command Behaves Very Different

So, How Do We Know How to Use Them?

Coming Up Next ...

- How to access and read the manual (man) pages.

Using The Linux Manual

General Command Structure

- CommandName -option1 -option2 input1 input2

Each Command Behaves Very Different

So, How Do We Know How to Use Them?

Answer:
The Manual (Man) Pages!

- How the manual is **structured**.

By The End

- You will have an understanding of the manual structure and cool cheat sheet to take away with you ☺

Manual Structure

Section	Contains
1	User Commands
2	System Calls
3	C Library Functions
4	Devices and Special Files
5	File Formats and Conventions
6	Games
7	Miscellaneous
8	System Administration

Manual Structure

Section	Contains
1	User Commands
2	System Calls
3	C Library Functions
4	Devices and Special Files
5	File Formats and Conventions
6	Games
7	Miscellaneous
8	System Administration

Summary

- ✓ The Linux manual is broken up into **8 sections** (see cheat sheet).

- How to **search** the manual and **discover** new commands.
- How to **access** the man pages.
- How to actually **read** the man pages.

By The End

- You will have the independence to search for new commands and learn how they work.

Summary

- ✓ You saw how to **search** the manual pages using **man -k**.
- ✓ How to **read** the manual pages for **specific command structure** (see cheat sheet).

Command Input + Output

By The End

- You will understand the ways **data** flows **into** and **out** of a **command**.
- Be ready to **connect commands** together to build powerful **pipelines**.



Key:

—→ Standard Data Stream

→ Not a Data Stream

Summary

- ✓ There are **2 ways** to get data **into** a command **2 ways** to get data **out**.
- ✓ Standard Input, Standard Output and Standard Error are “Standard Data Streams”.
- ✓ Data streams can be **redirected** from their **default** locations to wherever you wish.
- ✓ You can **redirect** the standard **output** of one command to the standard **input** of another in a process known as “**piping**”.

Redirection

- How to perform **redirection** in Linux.

By The End

- You will be able to redirect standard data streams.

Summary

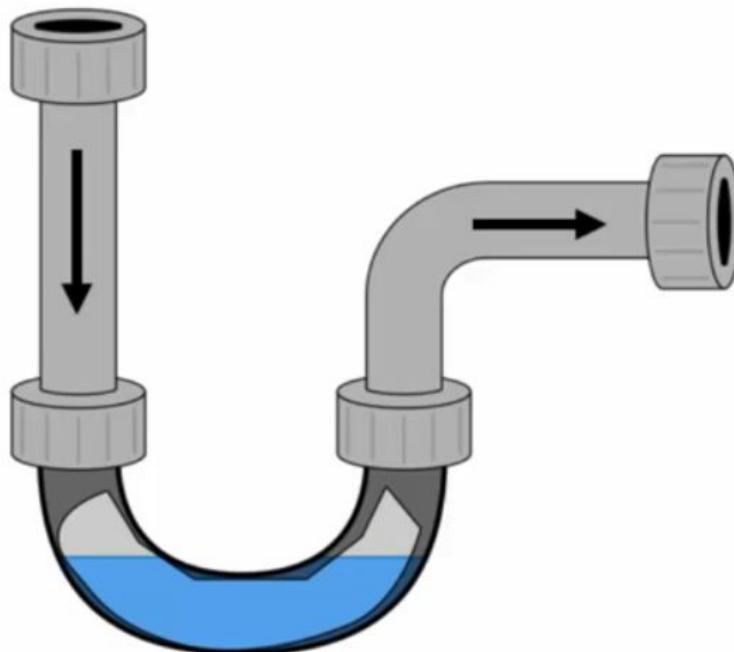
- ✓ Data Streams have numbers associated with them.
- ✓ You have seen how to redirect standard output (Data stream 1)
- ✓ Be careful of truncating your files!

Summary

- ✓ Standard **Input**, Standard **Output** and Standard **Error** are **Data Streams**.
- ✓ Using **redirection** you can **control** where those streams “**flow**”.
- ✓ Standard Input = 0, Standard Output = 1, Standard Error = 2.
- ✓ > will **overwrite** a file before writing to it.
- ✓ >> will **append** to what's already there.
- ✓ **Bonus Point:** Check the **Links in the Resources** for more information.

Piping

Command1



Command2

- You will learn how to build command pipelines.
- And then in part 2, you will learn some advanced piping techniques using the `tee` and `xargs` commands.

- You will learn some advanced piping techniques.

By The End

- You will know how to make advanced pipelines using the `tee` and `xargs` commands.

```
ls -l | tee file.txt | less
```

stdout

stdin

file.txt

- Learn about the `xargs` command.
- A `summary` of piping.

By The End

- You will be a *master* of piping.

Summary

- ✓ Piping connects **STDOUT** of one command to the **STDIN** of another.
- ✓ **Redirection** of **STDOUT** breaks pipelines.
- ✓ To save a data “**snapshot**” without breaking pipelines, use the **tee** command.
- ✓ If a command **doesn't accept STDIN**, but you want to pipe to it, use **xargs**.
- ✓ Commands you use with **xargs** can still have **their own** arguments.

Aliases

- How to make Aliases.

By The End

- Be able to **create** easy-to-remember **nicknames** for **your** pipelines.

Summary

- ✓ An alias is a custom **nickname** for a command or pipeline.
- ✓ Aliases go in a **.bash_aliases** file in your home folder.
- ✓ **Bonus Point:** A period (.) at the start of a filename makes that file “**hidden**”.
- ✓ **alias aliasName=“command1 -options args | command2 -options args ...”**
- ✓ aliases are accessible when you restart your terminal ☺
- ✓ **Bonus Point:** Make sure the **first command** can be **piped** to!

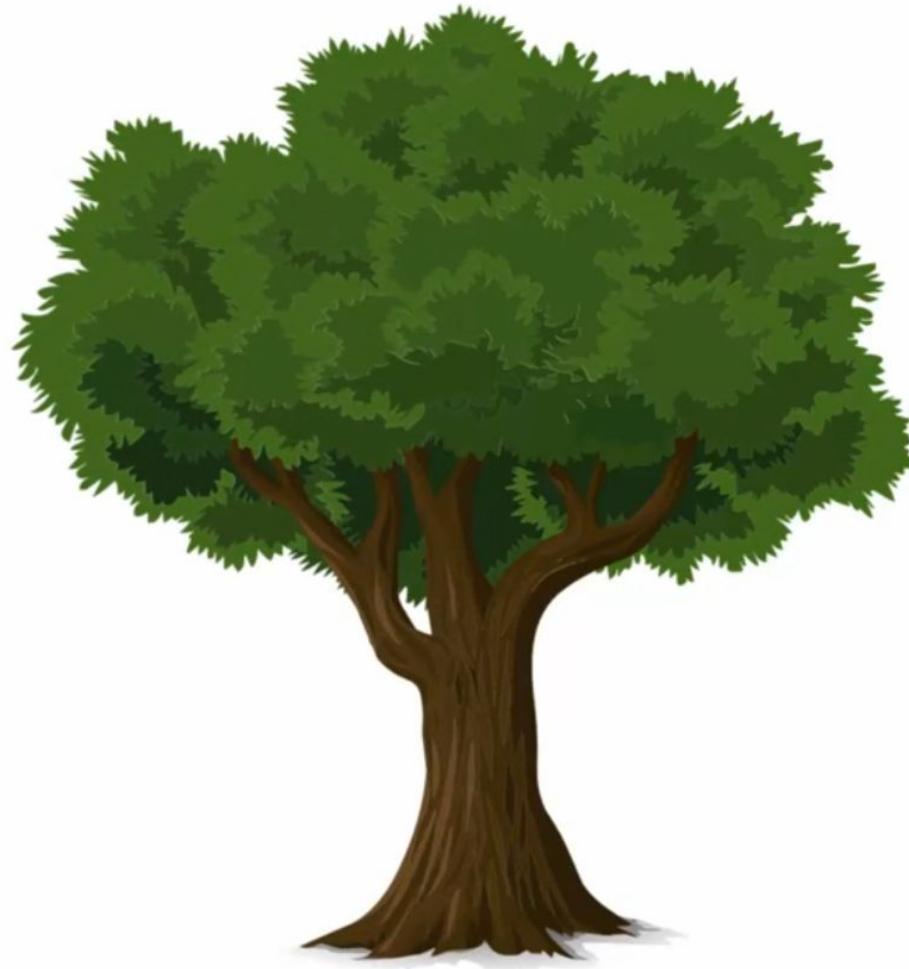
In this Assignment, you will be tasked with creating your very own command pipelines to list files on your computer :)

The Linux File System

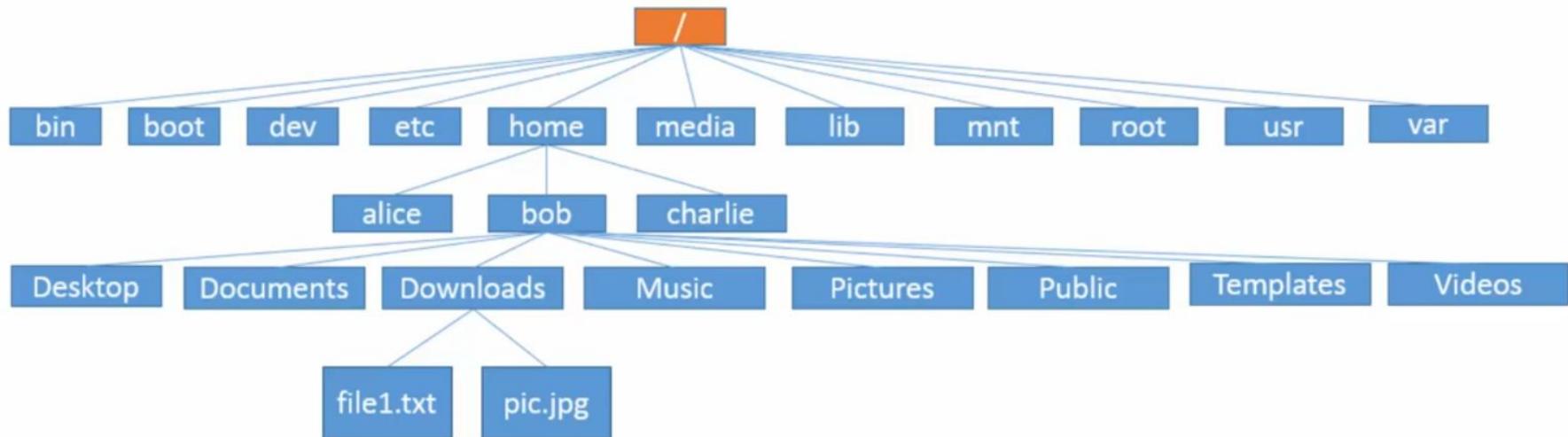
- How the Linux File System is **structured**.
- A tour of the **most important** locations.

By The End

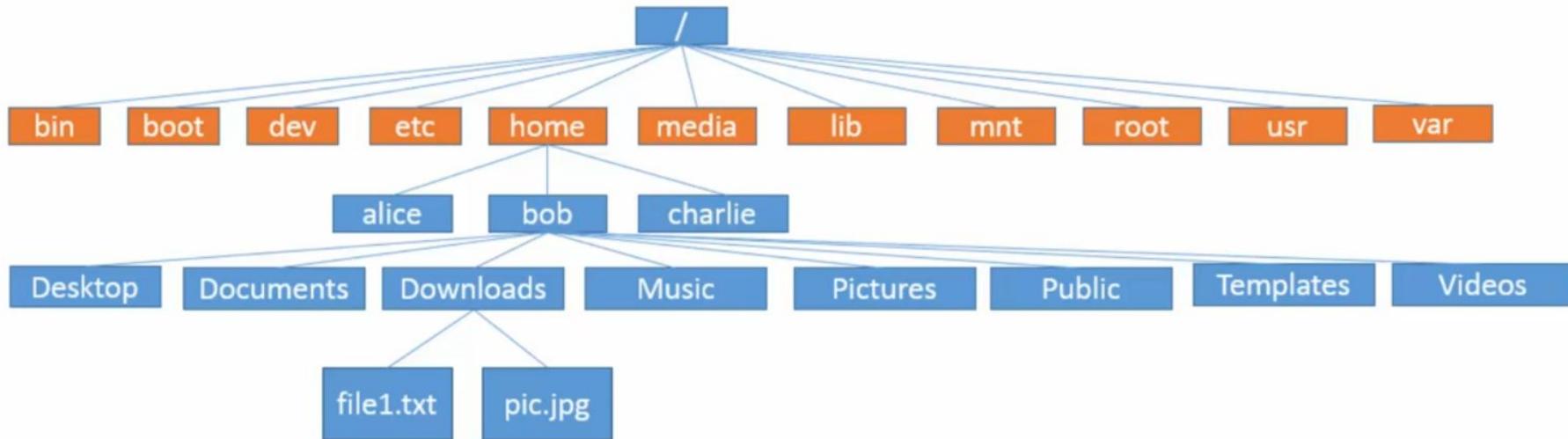
- You'll know how the Linux file system is laid out.



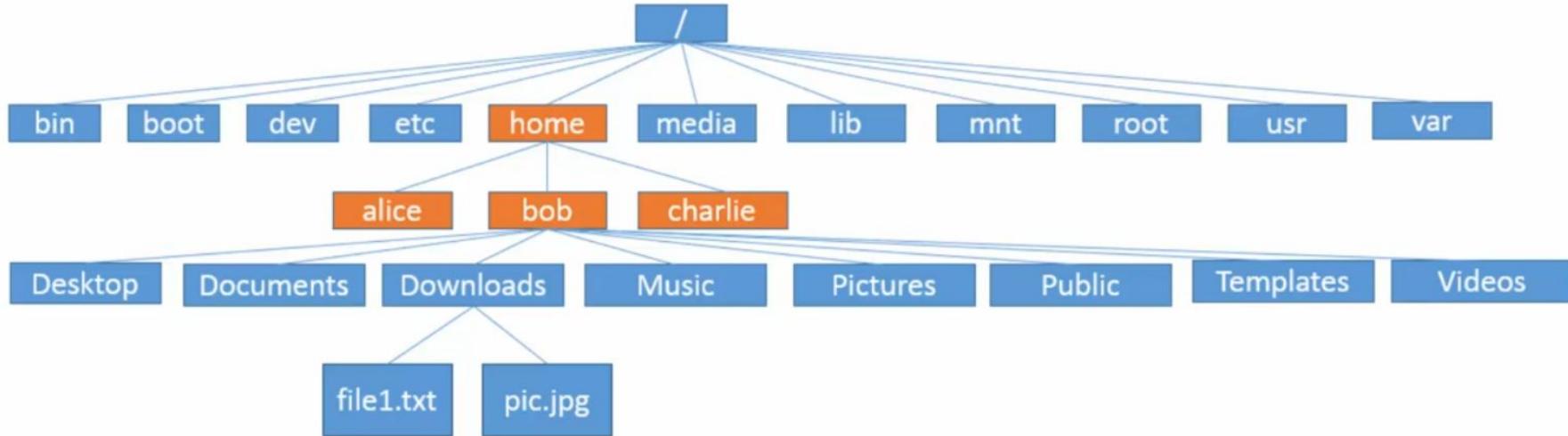
The Linux File System



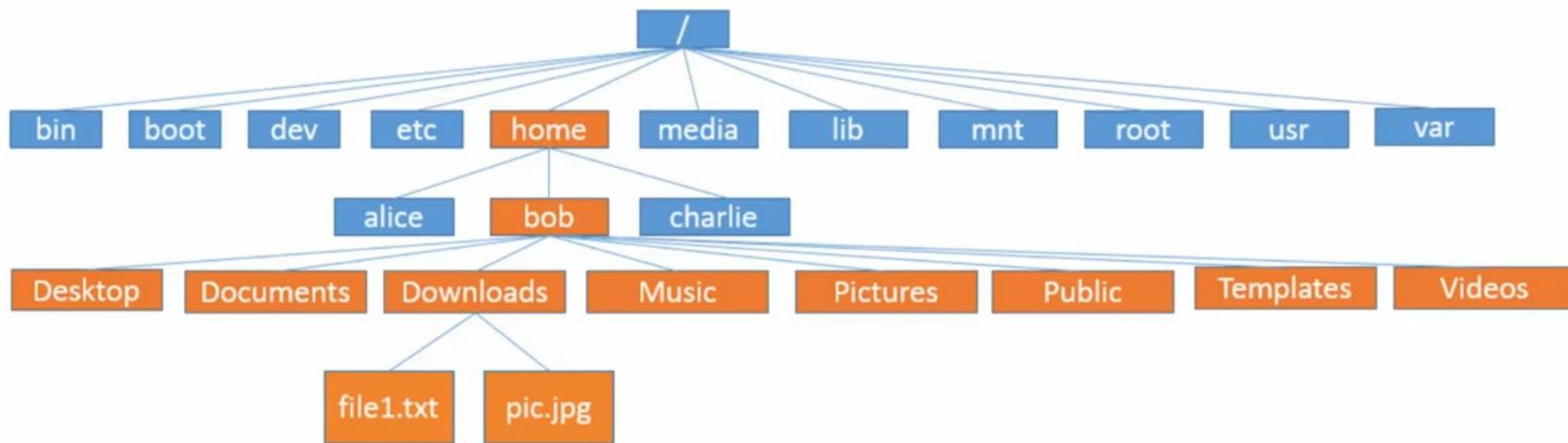
The Linux File System



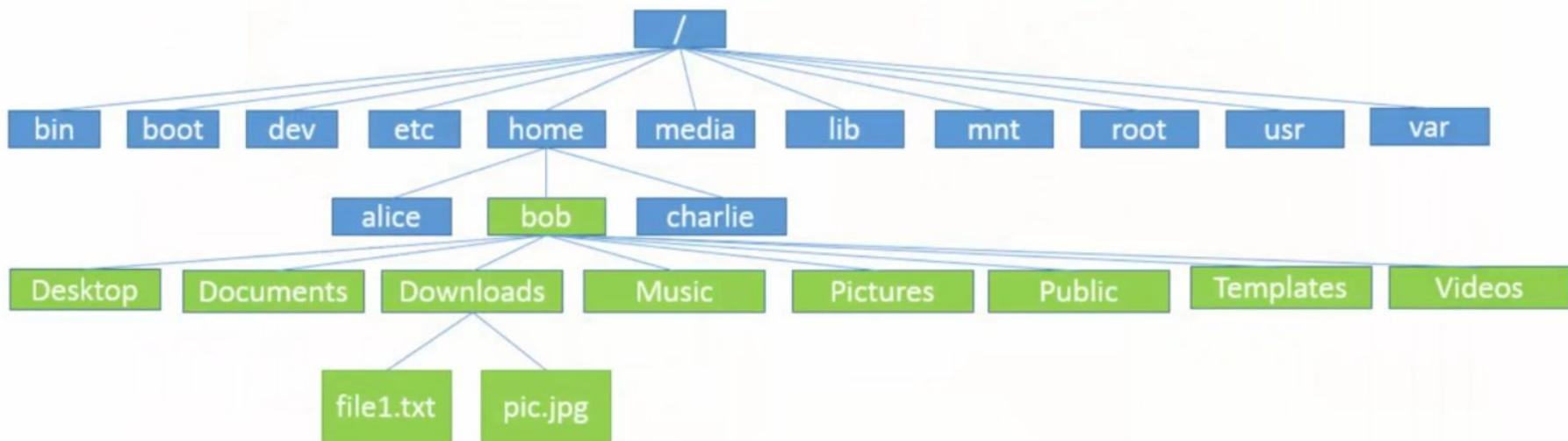
The Linux File System



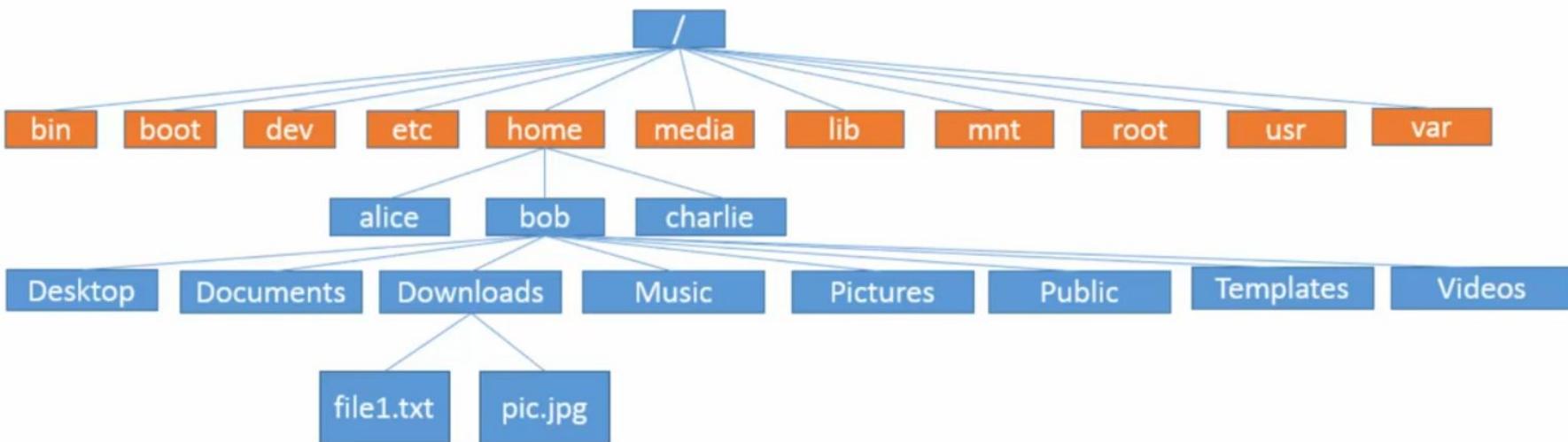
The Linux File System



The Linux File System



The Linux File System



Summary

- ✓ The Linux File system follows a tree-like structure.
- ✓ Everything can be traced back to the / directory.
- ✓ The root user has absolute power of the system.
- ✓ /home stores home directories for all regular users on the system.
- ✓ /root is the home directory for the root user.
- ✓ Had a tour of important folders (see cheat sheet).

Navigating the File System

- Learn how to use the `pwd`, and `ls` commands.

By The End

- Be able to know **where** you are on your file system and know **what's around you** by using the **command line**.

- Learn how to use the `cd` command to move around the file system.
- You will learn the difference between **absolute** and **relative** file paths.
- You will learn some **awesome** keyboard **shortcuts**.
- You will see how to put together the `pwd`, `ls` and `cd` commands to **elegantly** navigate the file system.

By The End

- You will know how to navigate the file system using the command line.

- How to use the **Tab Completion** to speed up navigating the file system.
- A **recap** of what we have learned about navigating the file system.

By The End

- You will have all the knowledge you need to navigate the file system like a pro – Just add practice!

Summary

- ✓ You use the `pwd` command to see the path to where on the file system the shell is currently operating.
- ✓ You can use the `ls` command to see what's around you.
- ✓ You use the `cd` command to move to a new location on the file system.
- ✓ **Absolute paths** start at the base (`/`) directory.
- ✓ **Relative paths** start from the current directory.

Summary

- ✓ Every directory has the `.` and `..` hidden folders.
- ✓ Tab auto completion is a really useful technique to speed up typing and avoid errors.
- ✓ Tab auto completion can be used anywhere, not just when navigating!
- ✓ Right click and select “open in terminal” to open a terminal where you are graphically.

File Extensions in Linux

- Deep Dive on File Extensions in Linux .

By The End

- Understand the **freedoms** that Linux gives you with file extension
- Know what to **be aware** of when **naming** files.
- Have learned a **new command**.

Summary

- ✓ Use the `file` command to know what `type of file` you are dealing with.
- ✓ You can name files `whatever you want` in Linux (even `.shablam`)
- ✓ Try not `confuse` third party software!

Wildcards

- The most common types of wildcards.

By The End

- Know how to **use wildcards** to make your commands more **powerful**.

Summary

- ✓ Wildcards are used to build patterns called “regular expressions”.
- ✓ Anything that matches the pattern will be passed as a command line argument to a command.
- ✓ * matches anything, regardless of length
- ✓ ? Matches anything, but just for one place
- ✓ [] Matches just one place, but allows you to specify options.

Creating Files and Directories

- How to use the `touch` and `mkdir` commands to create files and directories.

By The End

- Be able to **create** your own **files** and **folders** using the **command line**.

Summary

- ✓ `touch` is used to create files.
- ✓ `mkdir` command is used to create directories.
- ✓ `mkdir -p` can be used to create entire folder paths.
- ✓ Don't put spaces in your filenames. Use `underscores` instead.
- ✓ `Brace expansion` can be used to do very complicated things (don't worry!)

Deleting Files and Directories

- Learn how to use the `rm` command to **delete files**.

By The End

- You'll know how to delete files using the command line.

Summary

- ✓ The `rm` command is used to `remove` items from your file system.
- ✓ `Wildcards` can be used to make commands even more `powerful`.

- Learn how to use the `rm` and `rmdir` commands to **delete folders**.

By The End

- You will know how to **delete files** and **folders** using the command line.

Summary

- ✓ The `rm` command needs to `r` option to delete folders (Be careful!)
- ✓ The `i` option will allow the `rm` command to be `interactive` when deleting.
- ✓ The `rmdir` command will only delete folders that are `empty`.

Copying Files and Directories

- How to use the `cp` command to **copy** files and directories.

By The End

- You'll be able to **copy** and **paste** files and folders using the **command line**.

Summary

- ✓ The `cp` command is what deals with **copying** and pasting.
- ✓ `cp <what you want to copy> ... <Destination>`.
- ✓ The `r` option allows you to copy entire folders **recursively** too.

Moving + Renaming Files and Directories

- How to use the `mv` command to **move** and **rename** files and folders.

Summary

- ✓ The `mv` command is what deals with **moving** and **renaming**.
- ✓ `mv <what you want to move> ... <Destination>`.
- ✓ `mv <what you want to rename> <New Name + Location>`

Linux System Programming

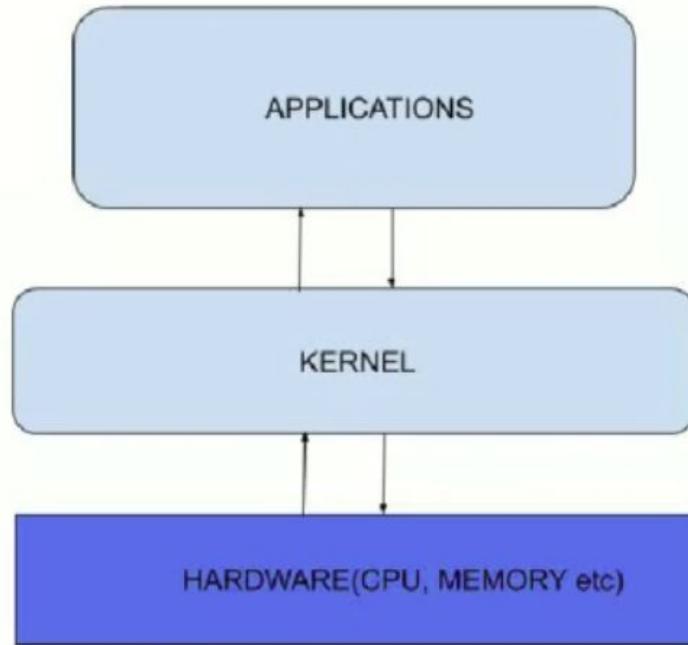
Linux Systems Programming Course Contents

- From basics to advanced Linux systems programming, with lots of hands on 'C' programming demos.
- File operation, System calls, Library functions, code compiling using GNU-GCC, Blocking and Non Blocking calls, Atomic operations, Race condition, User mode and Kernel mode.
- Process Management - Process creation, termination, Fork() system call, child-parent process, command line argument of process, Memory Layout of Process.

- Signals - signal handlers, sending signals to process, Default signal handlers.
- Memory Management - Process Virtual Memory management, Memory segments(code, data, stack, Heap)
- Posix Threads - Thread creation, thread termination, Thread ID, Joinable and detachable Threads.
- Thread Synchronisation - Mutex, Condition Variables
- Inter Process Communication (IPC)- Pipes, FIFO, Posix Message Queue, Posix semaphore, Posix shared memory.

Linux Fundamental concepts

- What is Kernel?
- The kernel is a computer program at the core of computer's operating system.
- It is the "portion of the operating system code that is always resident in memory."
- It facilitates interactions between hardware and software components.
- On most systems, it is one of the first programs loaded on startup (after the bootloader)



Kernel connects Applications and Hardware

Tasks performed by Kernel

- Process scheduling.
- Memory management
- Handling file systems
- Process management
- Device Access
- Networking

Kernel provides system calls(API) through which the user can request kernel to perform various tasks.

MultiTasking and MultiUser

The Kernel provides multi tasking, and multi user programming.

- Multi tasking – Is a feature provided by kernel in which several process can be run independently on CPU, and each process run in such a way that it cannot access other process memory.
- Multi-User – Linux kernel also provides a multi user login, that means Linux allows multiple users to use the computer without affecting each other's files. Linux provides a environment to independently perform various task.

Kernel space and user space

- The program is divided into modes/space i.e kernel space and user space.
- The program running in user space does not have access to all CPU instructions, and has limited access to resources. Hence if any error occurs, the complete system will not be affected, only the user space code will be terminated.

- When the program runs in Kernel mode, the program has access to all the underlying hardware. And no restrictions are imposed on software/program running in kernel space.
- The program in kernel space must be very much sure of its action, else the errors may lead to catastrophic results, and the entire system will be affected.

Shell

- Shell is a special program that is used to read the user's instruction through command line interface. It acts as a bridge between user's instruction and the action to be taken in kernel.

There are different shell used in linux, popular shells are

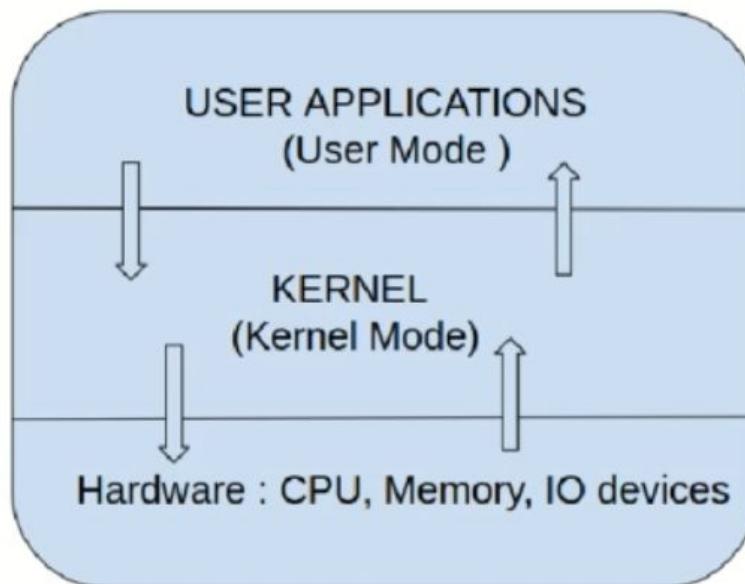
- Bourne shell(sh)
- C shell (csh)
- Korn shell (ksh)
- Bourne again shell(bash)

Files

- The files are generic word used in linux, there can be different types of files, below are the different files type in Linux.
 - 1) Regular or plain files – where the data is stored
 - 2) Pipes
 - 3) Sockets
 - 4) Directory
 - 5) Symbolic links

User mode and Kernel mode

- The main feature of Linux is to provide security/Accessiblity of the underlying hardware .



Linux Layered Architecture overview

Library Functions and System Calls

NOTE :

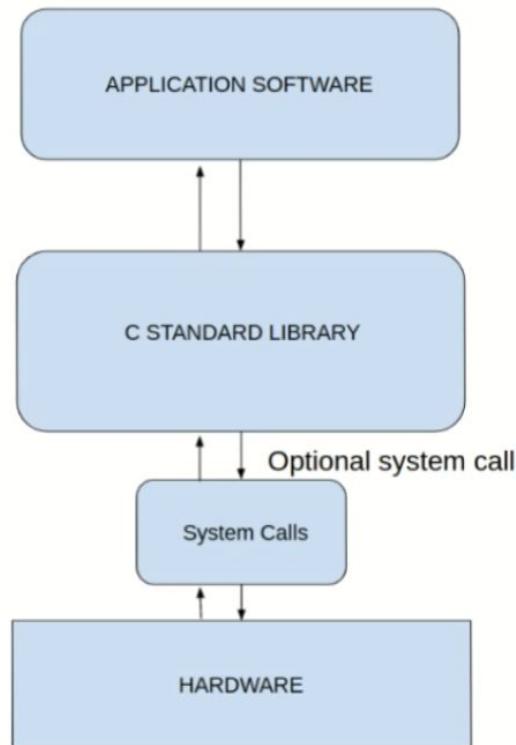
- All programs explained during this course are in C.
- Ubuntu PC is used to run the C program(User can use any Linux distribution).
- GCC compiler is used to compile the code.

- The Linux library functions are the C standard library.
- The C standard library is also called as libc.
- The C standard library provides header files that needs to be included in user's application program, in order to use the standard library function calls.

Library function

- A library is a collection of pre-compiled code.
- The library contains several functions together and they form a package or library. The library help to reuse the code by avoiding writing same code again and again.
- For example, the standard C library provide ‘stdio.h’ header file, and this header file has to be included in user program inorder to use the library function ‘printf’.
- Some Library functions are designed over the system calls, inorder to provide more additional functionality in the library functions(We will discuss this after learning system calls)

Library function Interface



Demo Program and C code Compilation

- The open source compiler we use to compile the code is GCC.
- We will look into a First demo 'C' program to write a simple program, and make use of standard Library and call library functions.
- The Terminal is used to compile the code using gcc
- The command used to compile code is
eg: If the source code is 'lib_ex.c' and required out file is 'lib_ex.out' then below command is used to compile the C code.

```
gcc lib_ex.c -o lib_ex
```

-o is used to create out file by name lib_ex.out

- For more details on usage on gcc type 'man gcc' on terminal

System Calls

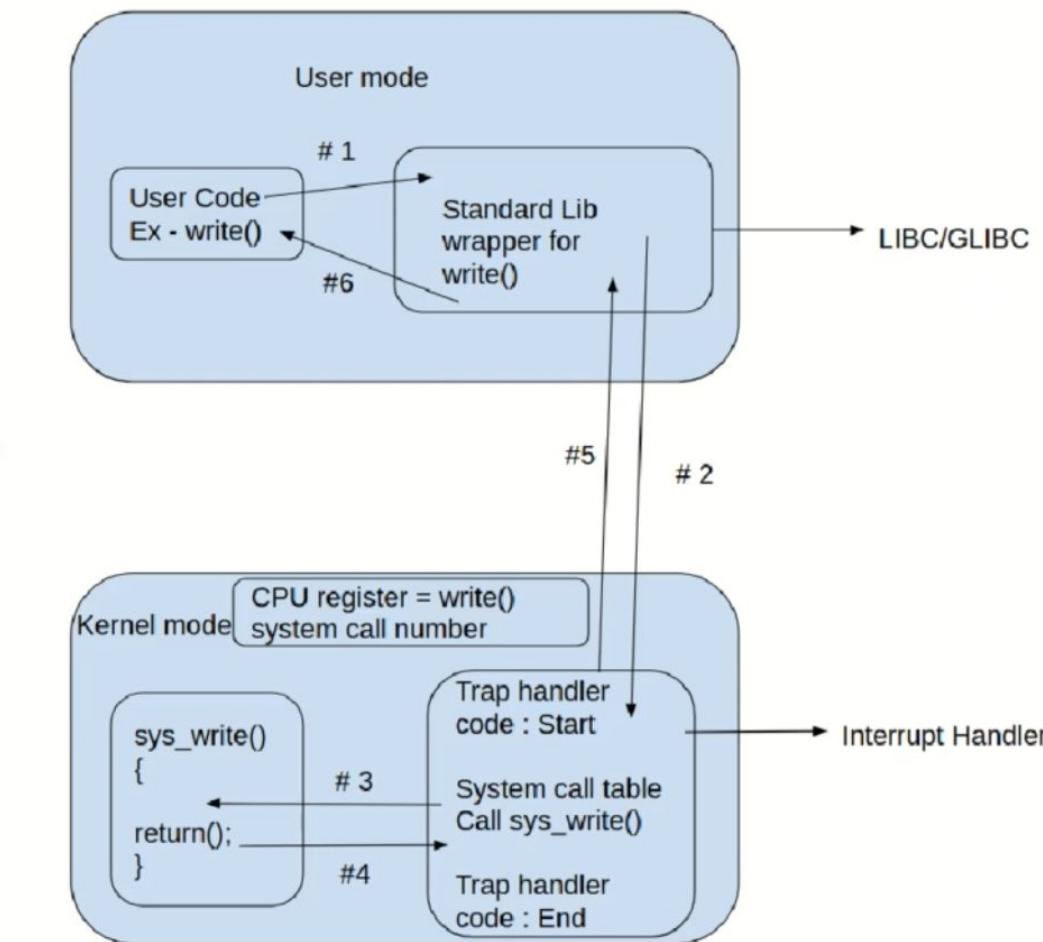
- A system call can be considered as an interface through which the User application code enters Kernel mode.
- Thus the system runs in kernel mode on behalf of the user program during system call execution.

System Calls

- System calls can be invoked from user application in 2 ways.
 - A) Directly calling the system call
 - B) The user code calls the library function, and this inturn calls system calls when required.

- Few examples of system calls are `open()`, `close()`, `read()`, `write()` etc.
- The system calls called through library functions are example `printf()`, inturn calls the `write()` system call

System Call working



Preliminary of Files in Linux

- Each file have below file permissions.
Read(r),write(w),execute(x)
 - Read has '4' units(2 power 2)
 - Write has '2' units (2 power 1)
 - Execute has '1' units (2 power 0)
- Each file can be accessed by different users as follows
 - User, group, others
- Each user type can have r/w/x access for file

Types of Files

- - : regular file (data files).
- d : directory.
- c : character device file.
- b : block device file.
- s : local socket file.
- p : named pipe.
- l : symbolic link.

File Input Output

The focus of this topic is exploring the system calls used for performing file input and output operations
Below concepts will be covered with examples.

1. File descriptors
2. System calls involved in file operations
3. Open()
4. Read()
5. Write()
6. Close()
7. Lseek()

File descriptor

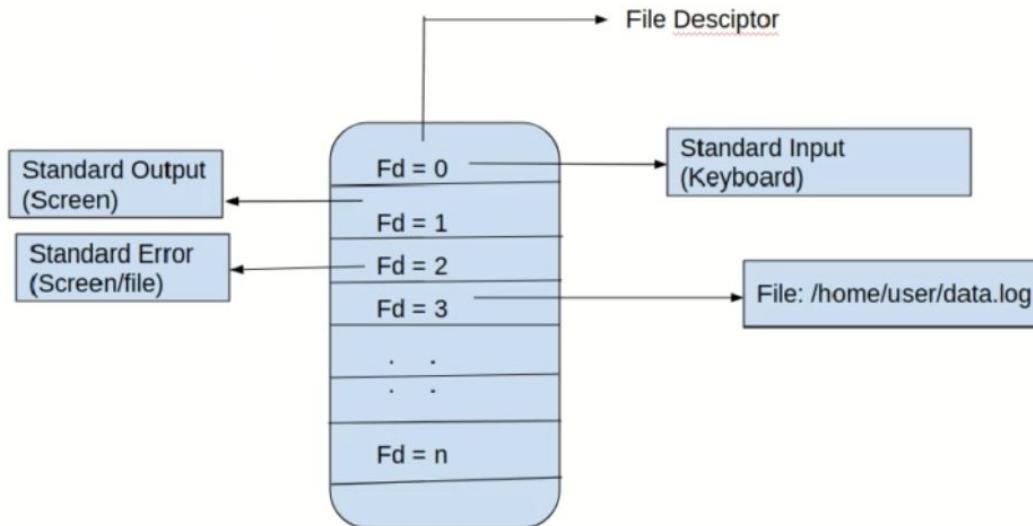
- All system calls for performing I/O refer to open files using a file descriptor, a (usually small) nonnegative integer.
- All file related operations are performed via file descriptor(fd)

- There are 3 standard file descriptor by default available in Linux.

File Descriptor	Description	Posix name	Stdio stream
0	Standard input	STDIN_FILENO	stdin
1	Standard output	STDOUT_FILENO	stdout
2	Standard error	STDERR_FILENO	stderr

Open() - System call

- In order to perform any operation like reading/writing a file, we need to open the file and provide a handle to the kernel. Any further operations on this file, will be done using this handle.



Open()

- Syntax of open() in C language

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (const char* Path, int flags , int mode );
int open (const char* Path, int flags );
```

Or

```
int open(const char * pathname , int flags , ... /* mode_t mode */); //  
generic description
```

Parameters of Open()

- › Path : path to file which you want to open the file.
- › Flags :
 1. The Flag should contain atleast one of `O_RDONLY`,
`O_WRONLY` , `O_RDWR` along with other flags.
`O_RDONLY` : read only,
`O_WRONLY` : write only,
`O_RDWR` : read and write.
 2. `O_CREAT`: create file if it doesn't exist.
When `open()` is used to create a new file.
If the `open()` call doesn't specify `O_CREAT` , mode can be omitted.
- › Mode: Specifies the file creation with access permissions.

Open() in C

Open() - return value

- If an error occurs, open() returns –1.
- errno is set accordingly

The errors that occur which are most common are

EACCES, EEXIST, EISDIR

Fd returned by open()

- If open() succeeds, it returns least non zero value. This value is called file descriptor.
- All operations(read,write) on this file opened is made via this descriptor.

- Lets us open the 'Linux Manual' page for open().

open(2) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [CONFORMING TO](#) | [NOTES](#) | [BUGS](#) |
[SEE ALSO](#) | [COLOPHON](#)

[OPEN\(2\)](#)

Linux Programmer's Manual

[OPEN\(2\)](#)

NAME

[top](#)

`open, openat, creat - open and possibly create a file`



SYNOPSIS

[top](#)

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

int open(const char * pathname, int flags);
int open(const char * pathname, int flags, mode_t mode);

int creat(const char * pathname, mode_t mode);

int openat(int dirfd, const char * pathname, int flags);
int openat(int dirfd, const char * pathname, int flags, mode_t mode);

/* Documented separately, in openat2(2): */
int openat2(int dirfd, const char * pathname,
           const struct open_how * how, size_t size);
```

Feature Test Macro Requirements for glibc (see `feature_test_macros(7)`):

```
openat():
  Since glibc 2.10:
    _POSIX_C_SOURCE >= 200809L
  Before glibc 2.10:
    _ATFILE_SOURCE
```

Read() - system call

- The read() system call reads data from the open file referred to by the descriptor fd.

```
#include <unistd.h>
ssize_t read(int fd , void * buffer , size_t count );
(size_t - unsigned integer)
```

- The buffer argument supplies the address of the memory buffer into which the input data is to be placed. This buffer must be at least 'count' bytes long

- Note : Read() does not allocate any memory, user must allocate memory and pass to read().
- A successful call to read() returns the number of bytes actually read, or 0 if end-of-file is encountered. On error, the usual –1 is returned.
Refer to C program on read() func call
- Note : read() system calls are applied on files like regular files, PIPES, sockets, FIFO

Write() - system call

- The write() system call writes data to an opened file.

```
#include <unistd.h>
```

```
ssize_t write(int fd, void * buffer , size_t count );
```

Returns number of bytes written, or –1 on error

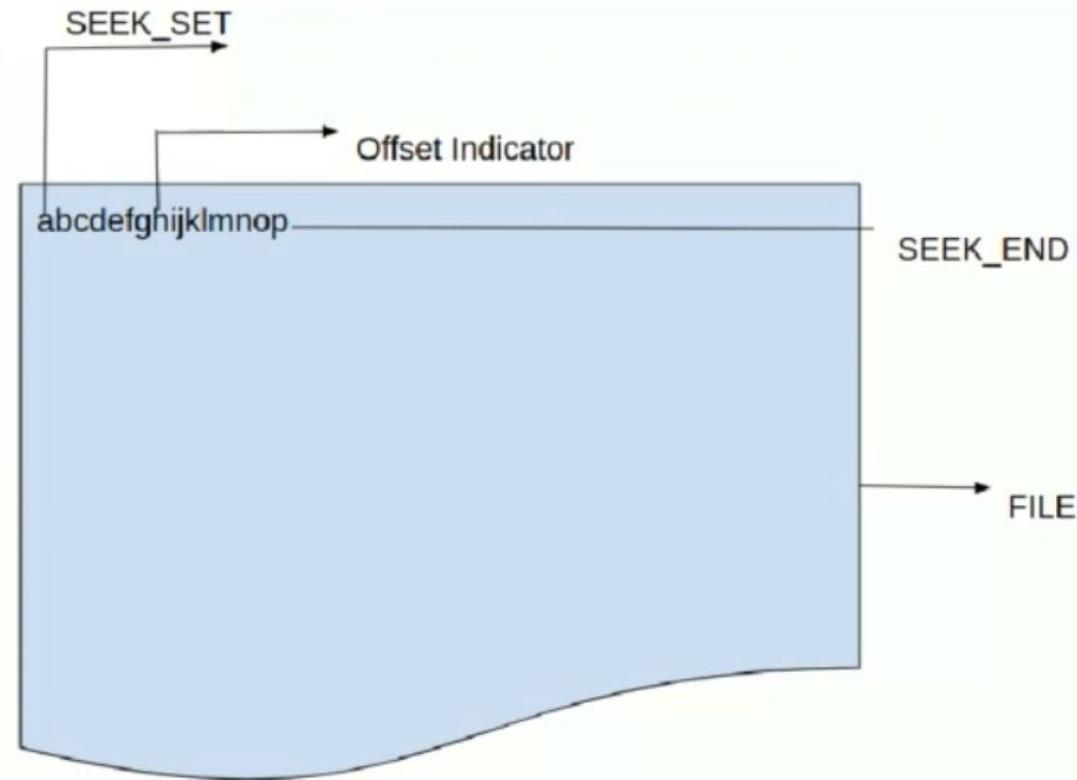
- The arguments to write() are similar to those for read(): buffer is the address of the data to be written on File.
- count is the number of bytes to write to file from buffer; and fd is a file descriptor referring to the file to which data is to be written.

- The buffer argument supplies the address of the memory buffer containing input data. This data is written to file specified by 'fd'. This buffer must be at least 'count' bytes long.
- A successful call to write() returns the number of bytes actually written to file, On error -1 is returned.

Note: The actual bytes written to file may also be lesser than 'count' bytes specified in 3rd parameter of write() system call.

Lseek() - system call

- A file can be considered as a continuous set of bytes.
 - There is an internal indicator present, which points to the offset byte of the file. This offset is used to read / write the next set of bytes/data from file. This indicator is updated when we do any file operation like read() or write().
 - Lseek is a system call that is used to change the location of the read/write pointer of a file descriptor. The location can be set either in absolute or relative terms.
-
- ```
➤ #include <unistd.h>
off_t lseek(int fd , off_t offset , int whence);
```
- Returns new file offset if successful, or -1 on error



# Lseek()

- fd : The file descriptor of the file.
- off\_t offset : The offset of the pointer (measured in bytes).
- int whence : The method in which the offset is to be interpreted
  - SEEK\_SET - The offset is set to offset bytes.
  - SEEK\_CUR - The offset is set to its current location plus offset bytes.
  - SEEK\_END - The offset is set to the size of the file plus offset bytes.
- return value : Returns the offset of the pointer (in bytes) from the beginning of the file. If the return value is -1, then there was an error moving the pointer.

# Advanced – FILE I/O

Advanced File Input output operations

What is Atomicity

What is Race condition

Fcntl() - system call

What is Blocking and non-Blocking I/O calls

File descriptor Table and Open file Table

Inode of a file.

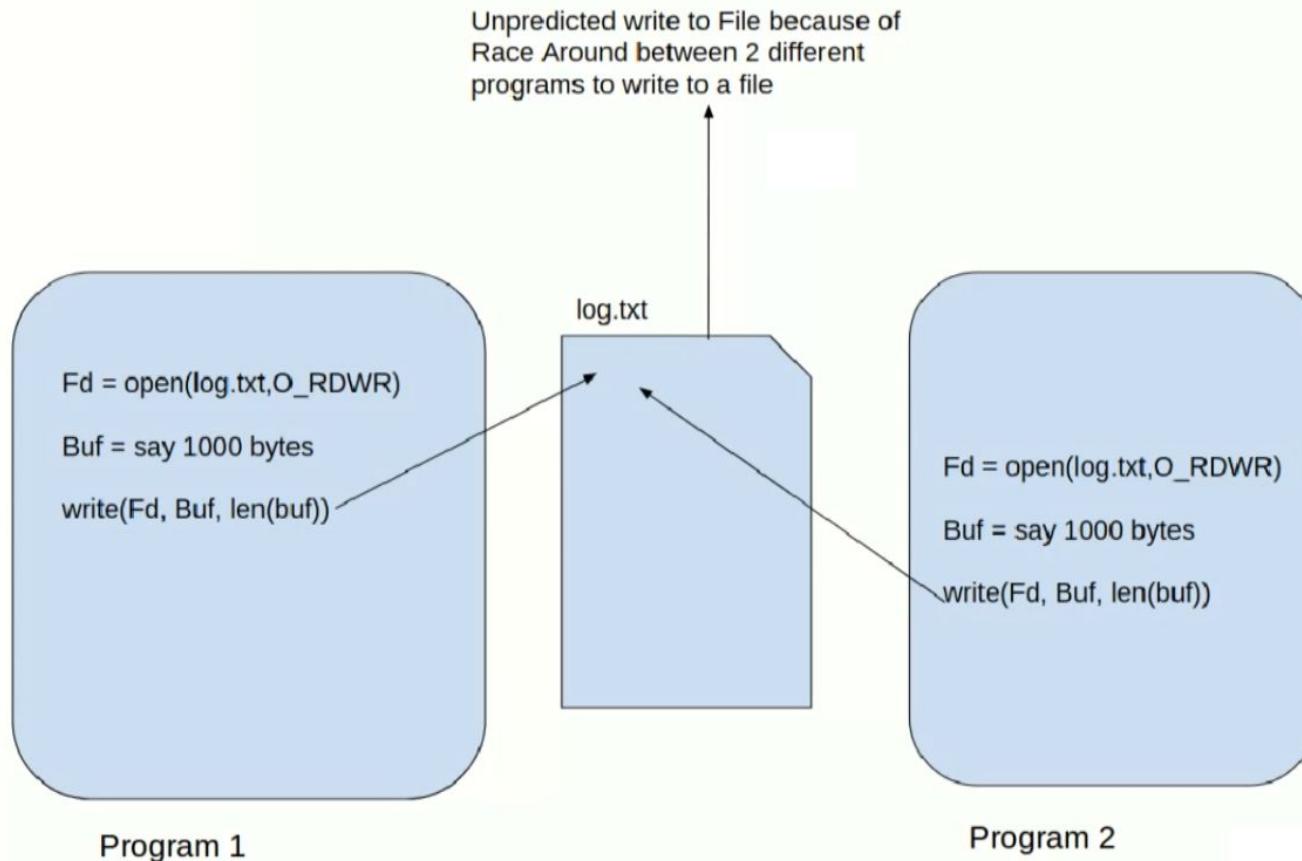
# Multi Processing/Threading

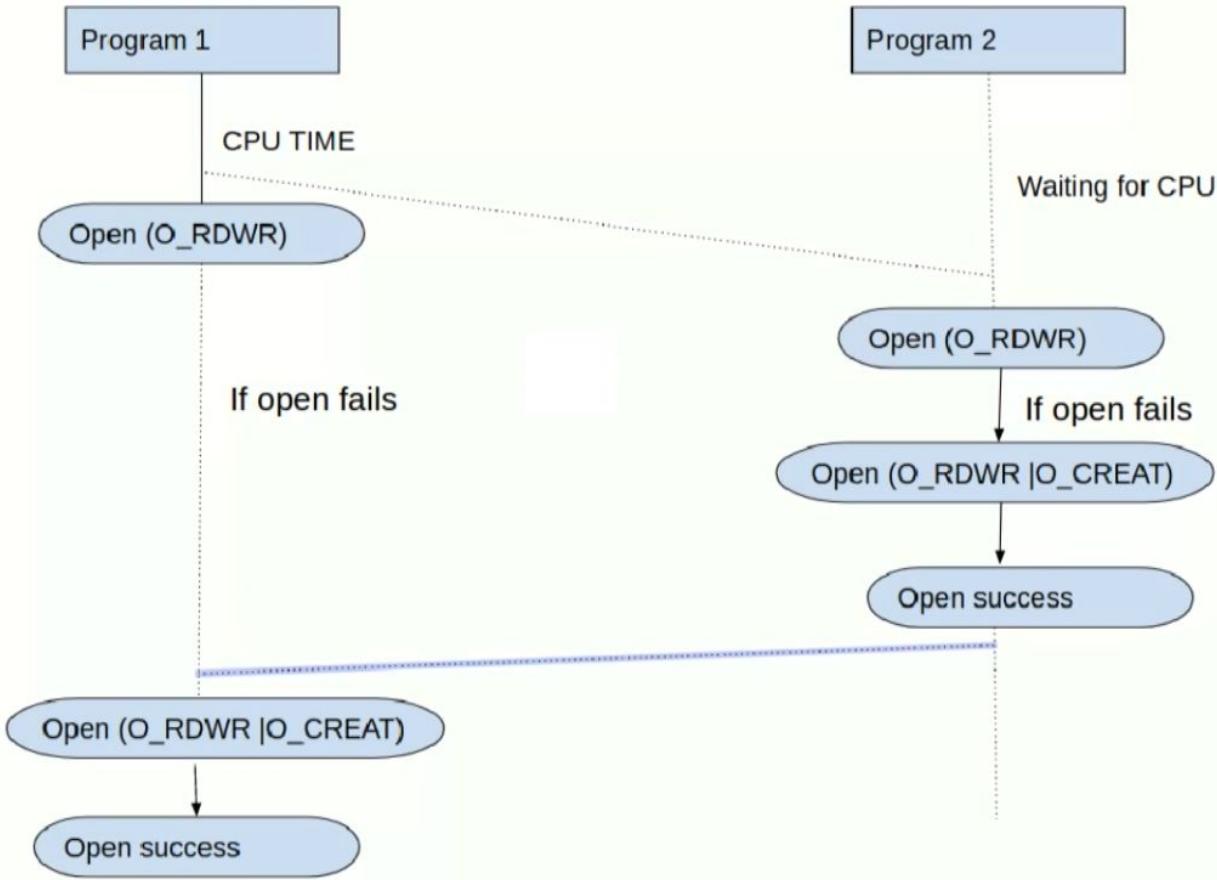
- Linux Provides a multi processing environment.
- This means there are several process running parallelly in linux system.
- Each Process based on its priority, The Scheduler gives every process a definite time on CPU to execute.
- Hence giving the user a virtual feeling as if all process are executing at same time and in parallel.

# Race condition

- A race condition is a situation where the result produced by two processes (or threads) operating on shared resources depends in an unexpected way on the relative order in which the processes gain access to the CPU(s).

# Race Round - Condition

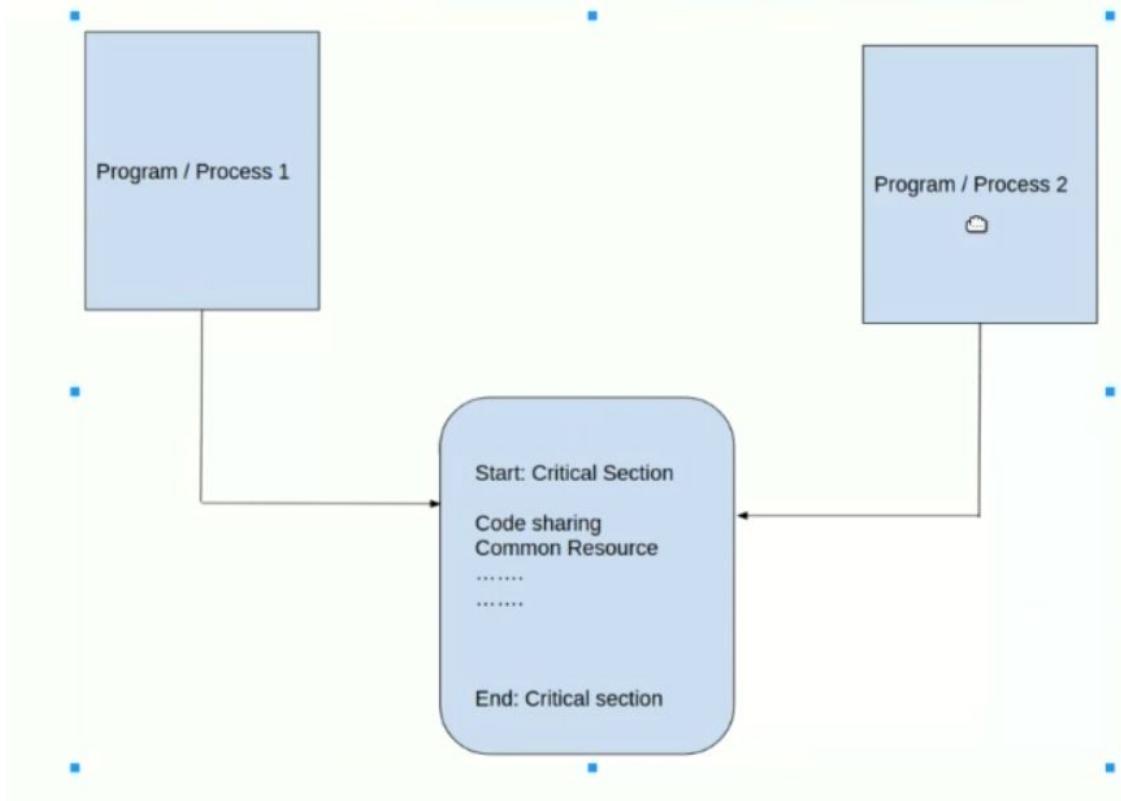




# Atomicity

- Atomicity is a condition in which the code is written in such a way that code cannot be run by other process/thread 'AT SAME TIME' while it is being executed/running by first process/thread.
- All system calls in Linux are executed atomically. By this, we mean that the kernel guarantees that all of the steps in a system call are completed as a single operation. Also in Linux the system calls either return pass or fail, without any intermediate results.
- The Atomicity is very much important to maintain in multi-processing environment like a linux system, else 'Race Round' condition can occur.

# Atomicity



# Pre-Emptive Scheduling

# Pre-Emptive Scheduler

- Pre-emptive is a behaviour when the running process (say Process A) time slice is ended, the scheduler schedules new process (say Process B) the CPU time, and again this continues for different Process, and now the Process A will be scheduled back(given CPU time).
- The Linux Kernel post 2.6 is a pre-emptive scheduling, can pre-empt the system calls running in kernel mode when it is in a safe to pre-empt state.
- Before versions of linux kernel performed non preemptive system call scheduling. Which means when a process was running in kernel mode, it was not able for scheduler to schedule other process, until the running process voluntarily releases CPU, or returns back to user mode.

# File descriptor table, File table and Inode table

# file descriptor table(per process)

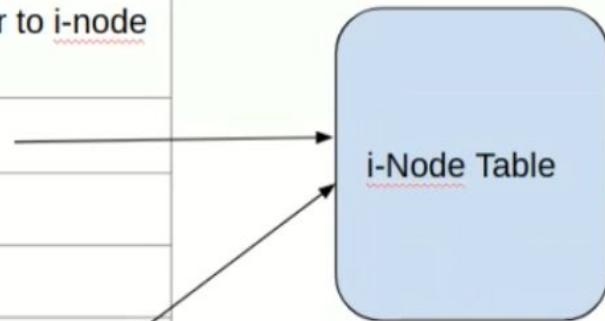
- File descriptor table(per process)
- Each process will have its own file descriptor table
- This table holds all open file of that process.

File descriptor table (Per process)

| Fd | Pointer to open file table |
|----|----------------------------|
| 0  |                            |
| 1  |                            |
| 2  |                            |
|    |                            |
|    |                            |

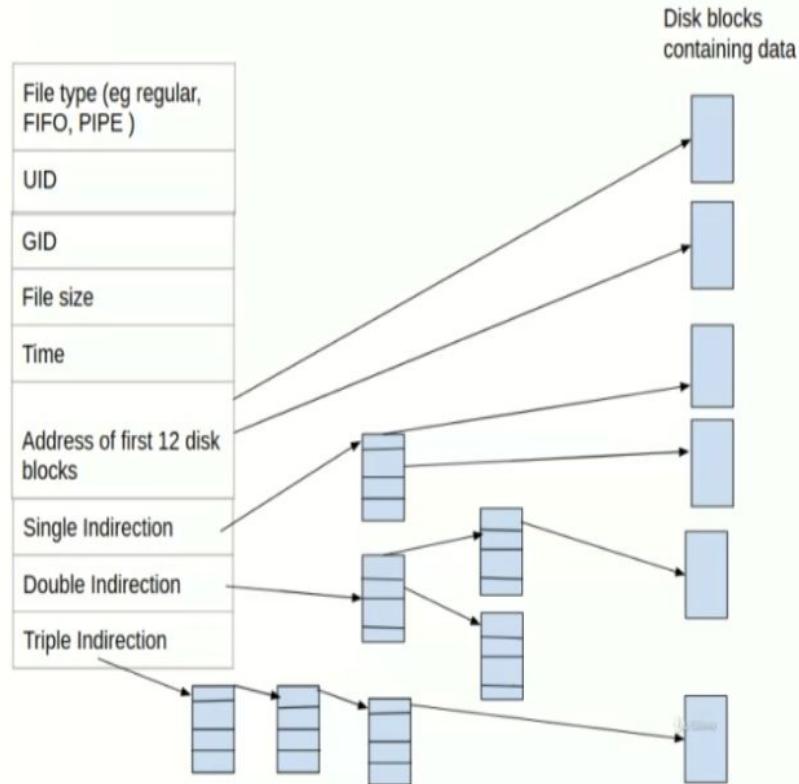
# Open file table (System wide)

- 1



# Inode - Table

I-Node Table



- Disk block size may be 4K Bytes, 8K Bytes, and so on
- Generally Block size in Linux is 4KBytes
- ‘stat filename’ – will give status information of inode and file

- File Type:

This keeps information about two things, one is the permission information, the other is the type of inode, for example an inode can be of a file, directory or a block device etc.
- Owner Info: Access details like owner of the file(UID), group of the file (GID) etc.
- Size: This location store the size of the file in terms of bytes.
- Time Stamps: it stores the inode creation time, modification time, etc.
- Block Size: Whenever a partition is formatted with a file system. It normally gets formatted with a default block size. Now block size is the size of chunks in which data will be spread.

# I-node – data mapping

- Address of first 12 block of memory

Total memory it can map is  $(12 \times 4\text{k}) = 48\text{KB}$

Note : We are assuming block size is 4KiloByte.

- Single Redirection

Total memory mapped =  $4\text{KB} \times 1\text{KB} = 4\text{MB}$

because in single redirection the first block acts as a pointer to other blocks

since the pointer block has 4KB, and every pointer needs 4 bytes in memory, we have total  $4\text{K}/4 = 1\text{K}$  pointers.

# I-node – data mapping

- Double Redirection

Total memory mapped =  $4\text{KB} \times 1\text{KB} \times 1\text{KB} = 4\text{GB}$

because in double redirection the first level and second level block acts as a pointer.

- Triple Redirection

Total memory mapped =  $4\text{KB} \times 1\text{KB} \times 1\text{KB} \times 1\text{KB} = 4\text{TB}$

because in triple redirection the first level and second level block acts as a pointer.

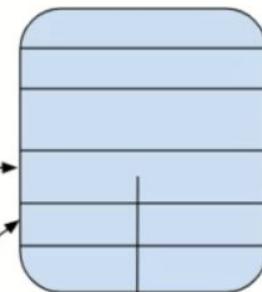
File descriptor table (Per process)

| Fd | Pointer to open file table |
|----|----------------------------|
| 0  |                            |
| 1  |                            |
| 2  |                            |
| 3  |                            |

Open file table(system Wide)

| File offset | File status flags | Pointer to i-node |
|-------------|-------------------|-------------------|
|             |                   |                   |
| 6           | O_RDWR            |                   |
|             |                   |                   |
|             |                   |                   |
|             |                   |                   |
|             |                   |                   |

I-Node Table  
(system wide)



Fd = open("./log.txt", O\_RDWR);  
// 3 will be assigned to fd

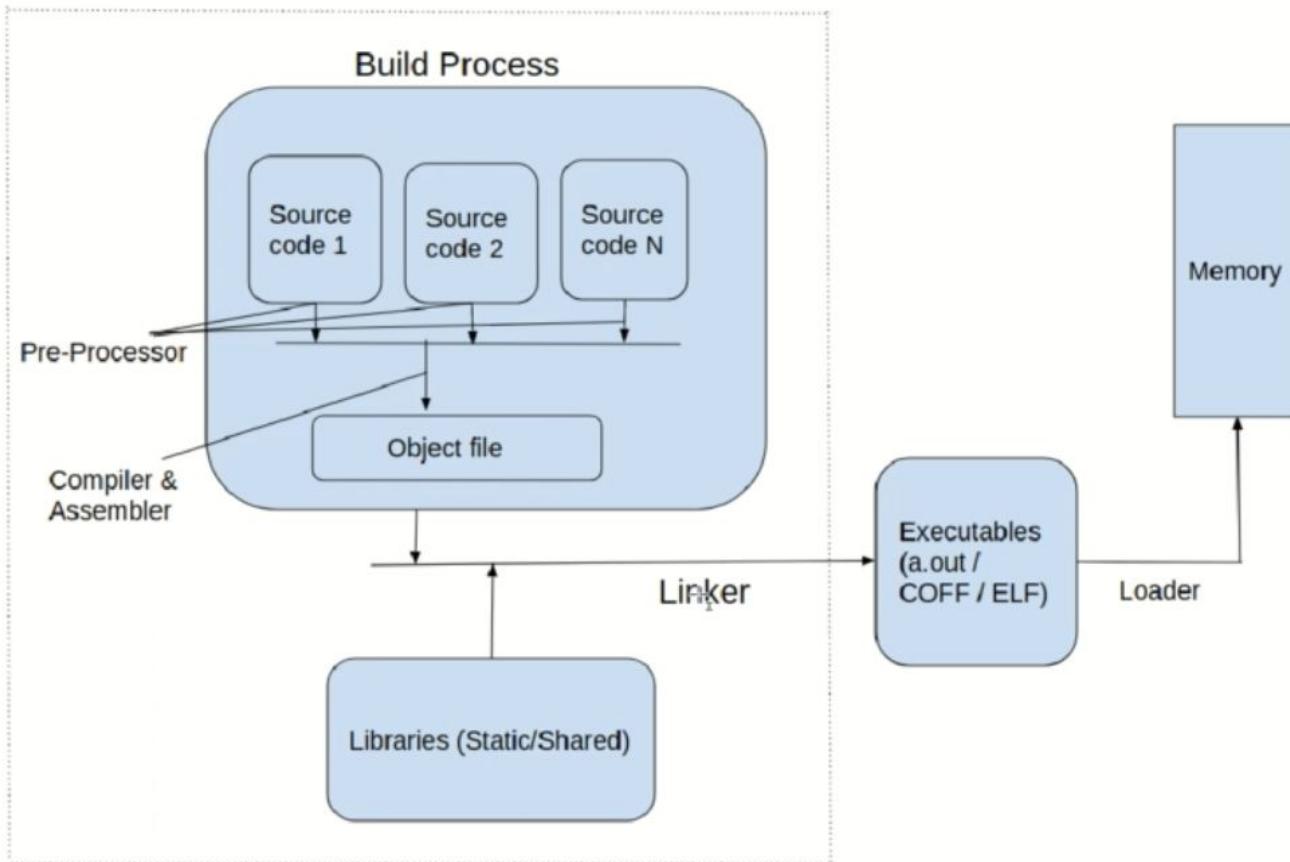
log.txt

abcdef

# PROCESS

- Processes and Programs
- Process ID and Parent Process ID
- Memory Layout of a Process.
- Examples of different memory section of Process using C Code.

# Program and Process



# Processes and Programs

- A process is an instance of an executing program.
- A program is a file containing a range of information that describes how to construct a process at run time.

# Child and Parent Process

- In Linux Processes have a Parent-child relation.
- A Process is created by its parent
- In Linux startup sequence

Process '0' is first process – called 'swapper' process

Process '1' is 'Init' Process – Init process plays an important role. It creates and monitors set of other process.

- Init process becomes the parent of any 'Orphan' Process.
- Man 8 init – for manual page on 'init' process.

# Process ID

```
#include <unistd.h>
```

- pid\_t getpid(void);  
Always successfully returns process ID of caller
- The Linux kernel limits process IDs to being less than or equal to 32,767(default on 32 bit).
- /proc/sys/kernel/pid\_max – contains 1 greater than actual max number of process allowed in system.
- ‘Ps -ef’ – get running process and PID

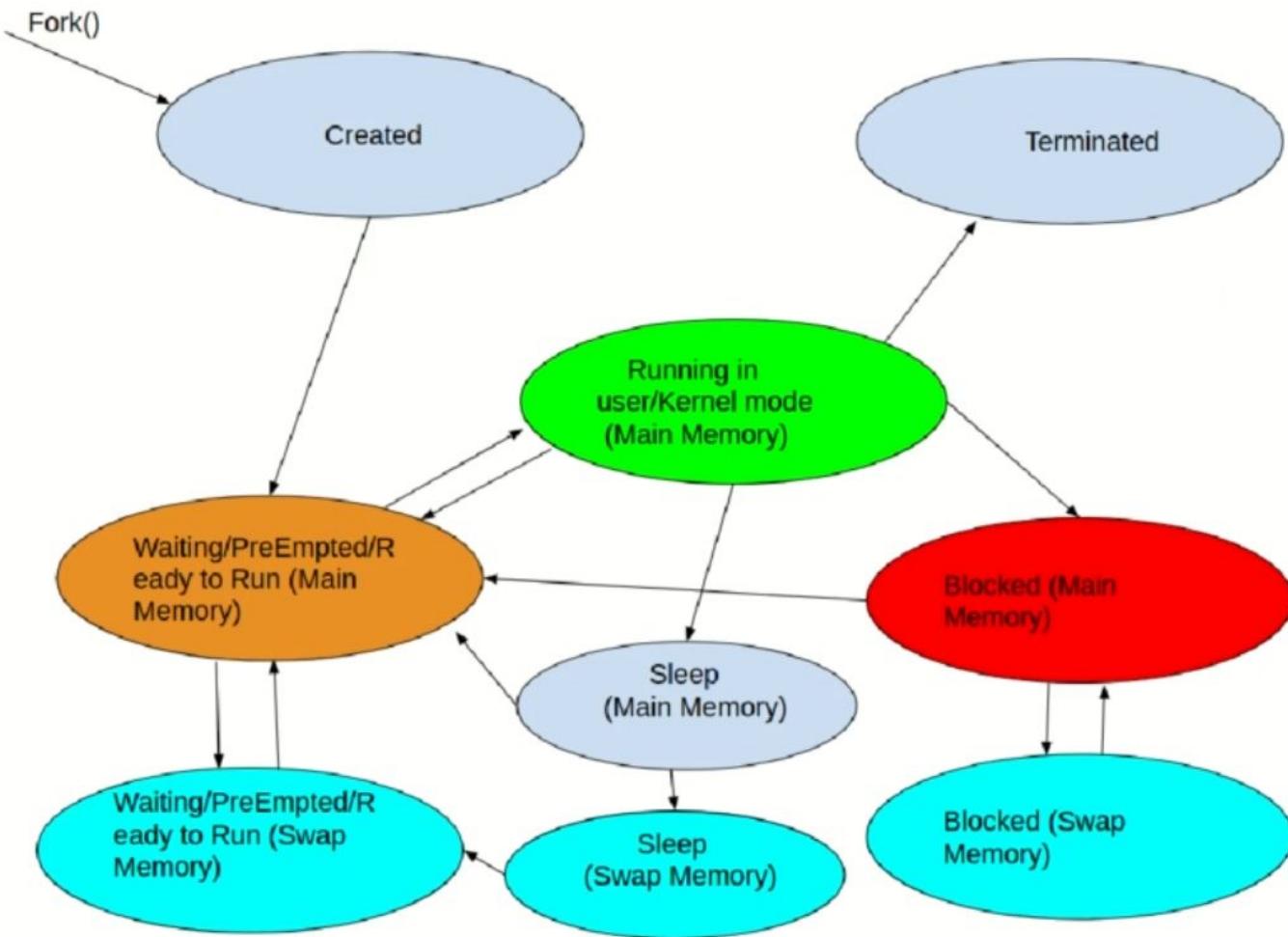
# Parent Process ID

```
#include <unistd.h>
pid_t getppid(void);
```

Always successfully returns process ID of parent of caller process.

# Process States

- A process is “created” state using fork() system call.
- ‘Running’ state – Process is running in ‘main memory’.
- ‘Ready to Run’ in ‘Main Memory’
- ‘Ready to Run’ in ‘Swap Memory’
- ‘Sleep’ state in ‘Main Memory’
- ‘Sleep’ state in ‘Swap memory’
- ‘Blocked’ state in ‘Main Memory’
- ‘Blocked’ state in ‘Swap Memory’
- ‘Terminated’ state



# Memory Layout of a Process

- The different segments of a process are..
  1. Text segment – The code resides here
  2. Data Segment – for data variables during compile time.
    - A. Initialised data segment
    - B. Un-Initialised data segment (BSS)
  3. Stack segment – for local variables
  4. Heap segment – for dynamic memory data's.

# Text Segment of a Process

- The ‘text segment’ contains the code of the program that is ran. This segment of memory cannot be written, so that the code is not altered by any pointers. This means ‘Text segment’ is Read only.

# ‘Initialized data segment’ of a Process

- The ‘initialized data segment’ contains global and static variables that are ‘explicitly initialized’ with value in the code.

# Memory Layout of a Process

- The uninitialized data segment contains global and static variables that are not explicitly initialized in the code. The system initializes all value's of this segment to '0'.

# 'Stack' segment of a Process

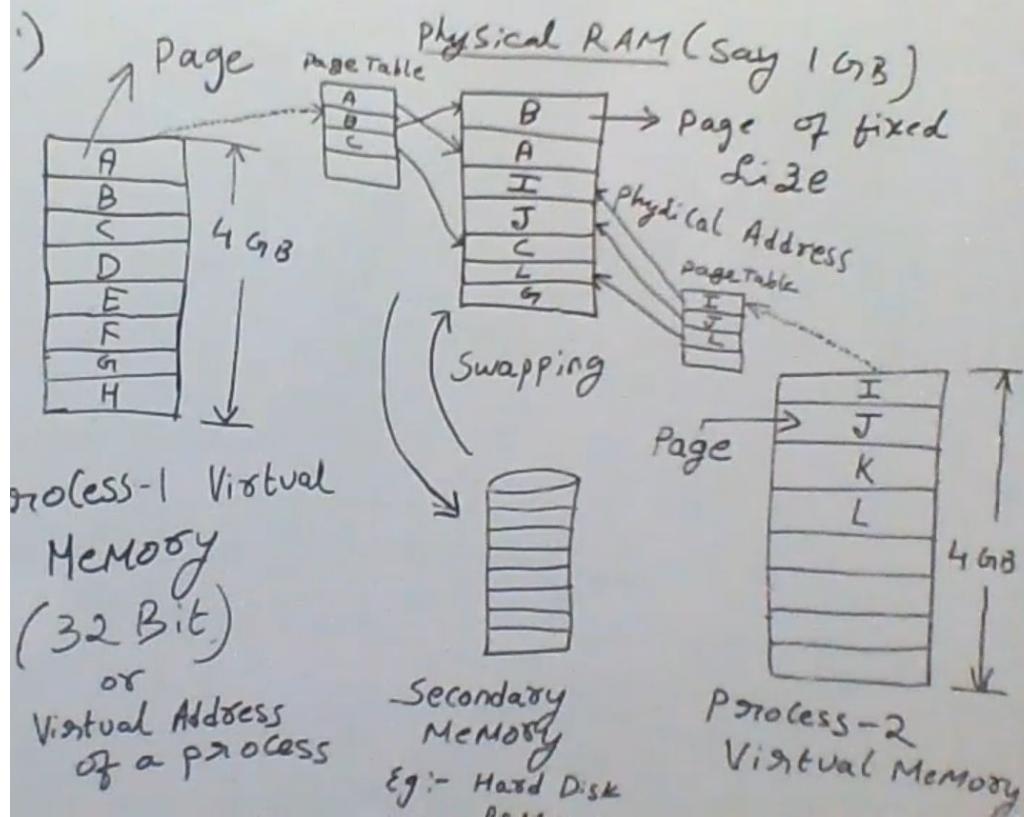
- The stack is a dynamically growing and shrinking segment containing local data's(stack frames) of functions. One stack frame is allocated for each function that are called. A frame stores the function's local variables, function arguments.

# 'Heap' segment of a Process

- The heap is an area from which memory (for variables) can be dynamically allocated at run time.

32 BIT  
 $2^{32} = 4GB$

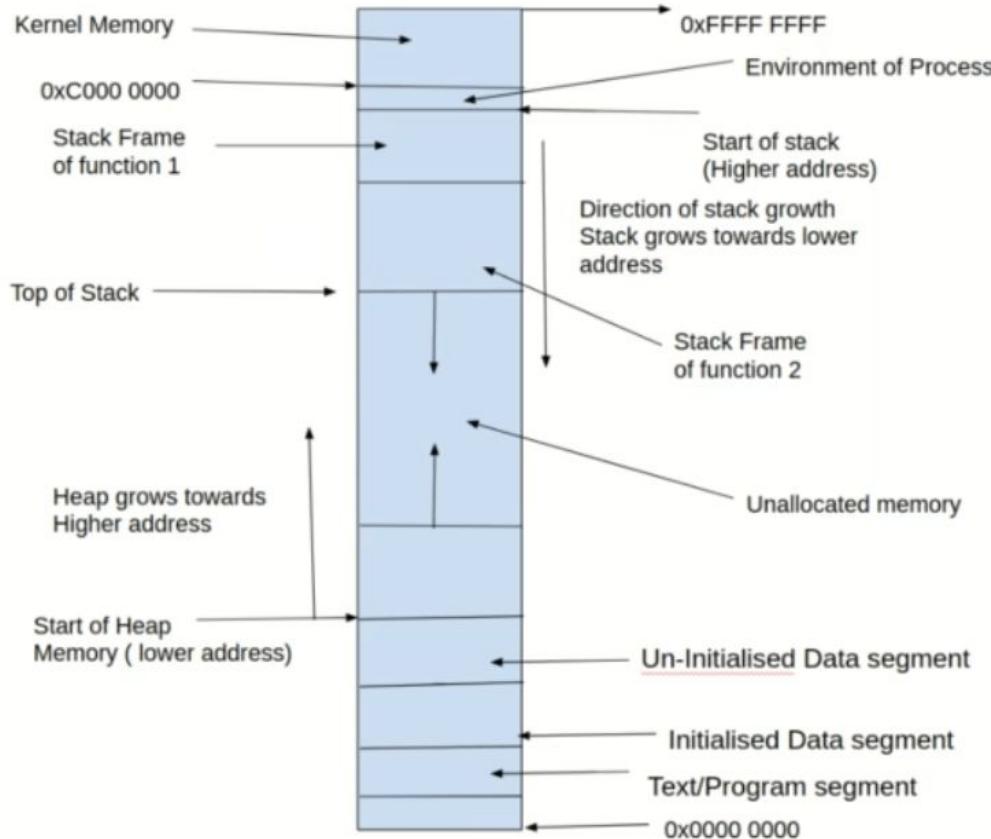
# Virtual Memory



# Virtual Memory Management

- Similar to other modern kernels, Linux has a technique known as virtual memory management. The aim of this technique is to make efficient use of both the CPU and RAM (physical memory/Main Memory)
- Each Process has its own memory space(4GB) on a 32 bit system.
- Each process has a private user space memory. This means one process cannot access memory of other process directly.
- Each process has below separate Memory segments(the size of each memory segment is configurable)
  - a. Text segment
  - b. Data segment(Initialized and uninitialised)
  - c. Stack segment
  - d. Heap segment
- The Process virtual memory has a user space, and a Kernel space. This memory region is configurable, but it is usually 3GB user space and 1GB Kernel space.

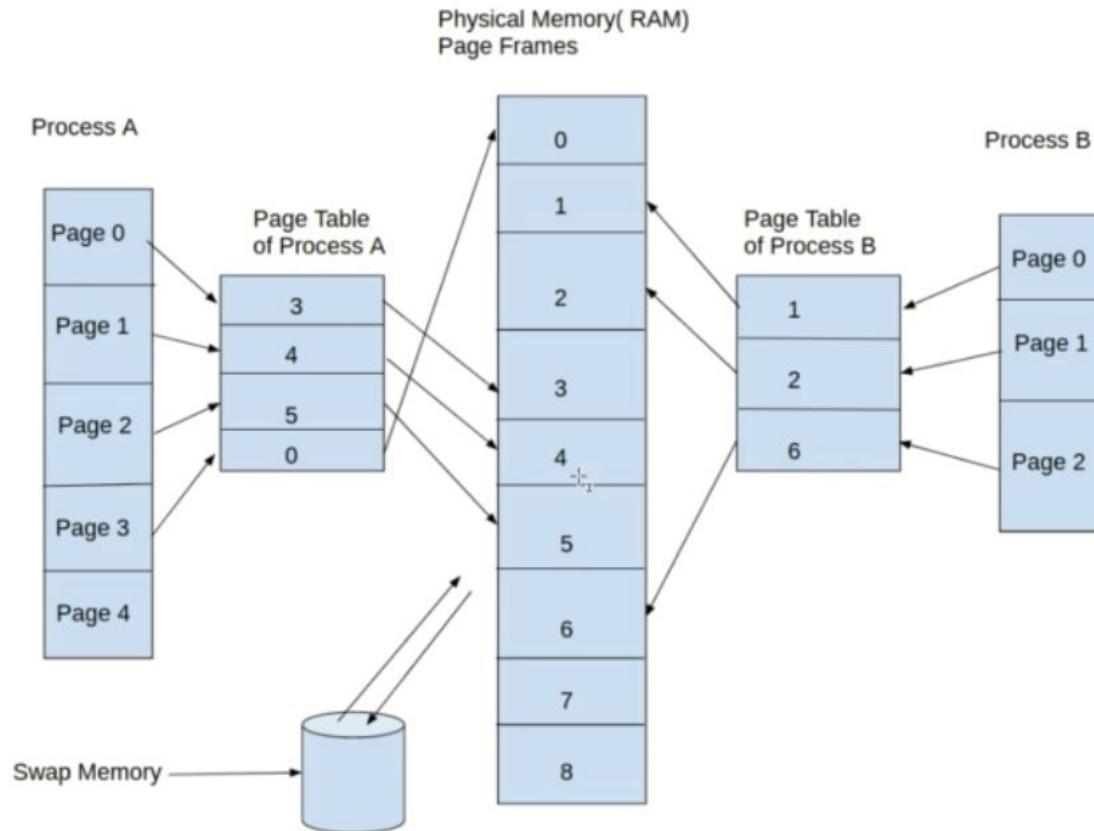
# Virtual Memory of process



# Page Frame

- A virtual memory of each process, is split into small, fixed size units called 'pages' (default page size usually is 4096 Bytes, but configurable to different size).
- Similarly, RAM(Physical memory) is divided into a series of page frames of the same size.
- At any given point of time, only some of the 'pages' of a process are present in physical memory (RAM), these pages are called 'Resident Set'.
- Copies of the unused pages (whose entry is not present in page table) of a process are maintained in the swap area (usually disk space) and loaded into physical memory, when required.
- Frame size varies – generally 4K, 8K or 16K and is configurable.

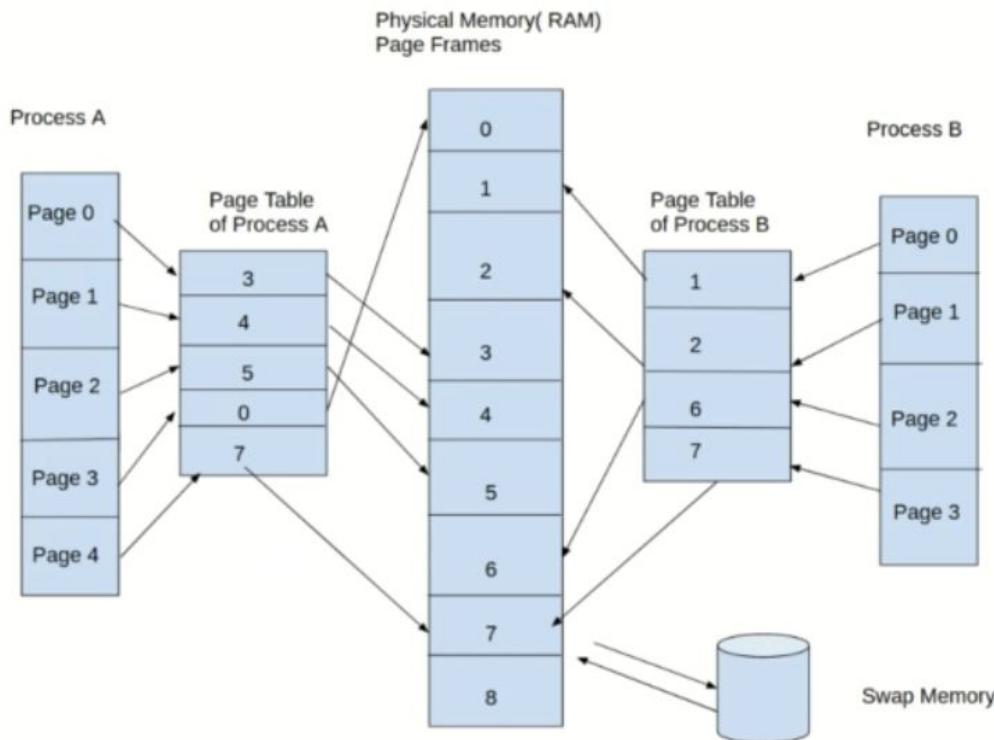
# Page Table



# Page Table

- Two or more processes can share memory. This is possible by having page-table entries of different processes refer to the same pages of RAM. Memory sharing occurs below circumstances:
  - a. Multiple processes executing the same program can share a single (read-only) copy of the program code.
  - b. Processes can use the shared memory with other processes to exchange data's.

# Memory Sharing between Process



# Page Fault

- When a process tries to access a page that is not currently present in physical memory (and page table), a page fault occurs, at this point the kernel suspends execution of the process while the page is loaded from swap memory into Main memory (RAM).

# Environment of a Process

- Each process has a set / list of strings called the environment . Each of these strings are in form ‘name=value’.
- Thus, the environment represents a set of ‘name-value’ pairs that can be used to hold information. The names in the list<sup>I</sup> are referred to as environment variables.

# Memory Allocations

In the session of Memeory Allocation, we will discuss the below topics

- a. Malloc()
- b. Calloc()
- c. Realloc()
- d. Alloca() - Stack level memory allocation.
- e. Brk()
- f. Sbrk()
- g. Free()

Note – brk() and sbrk() are the 'system call' for memory allocation which are used in implementation of 'library functions' malloc(), calloc() and realloc().

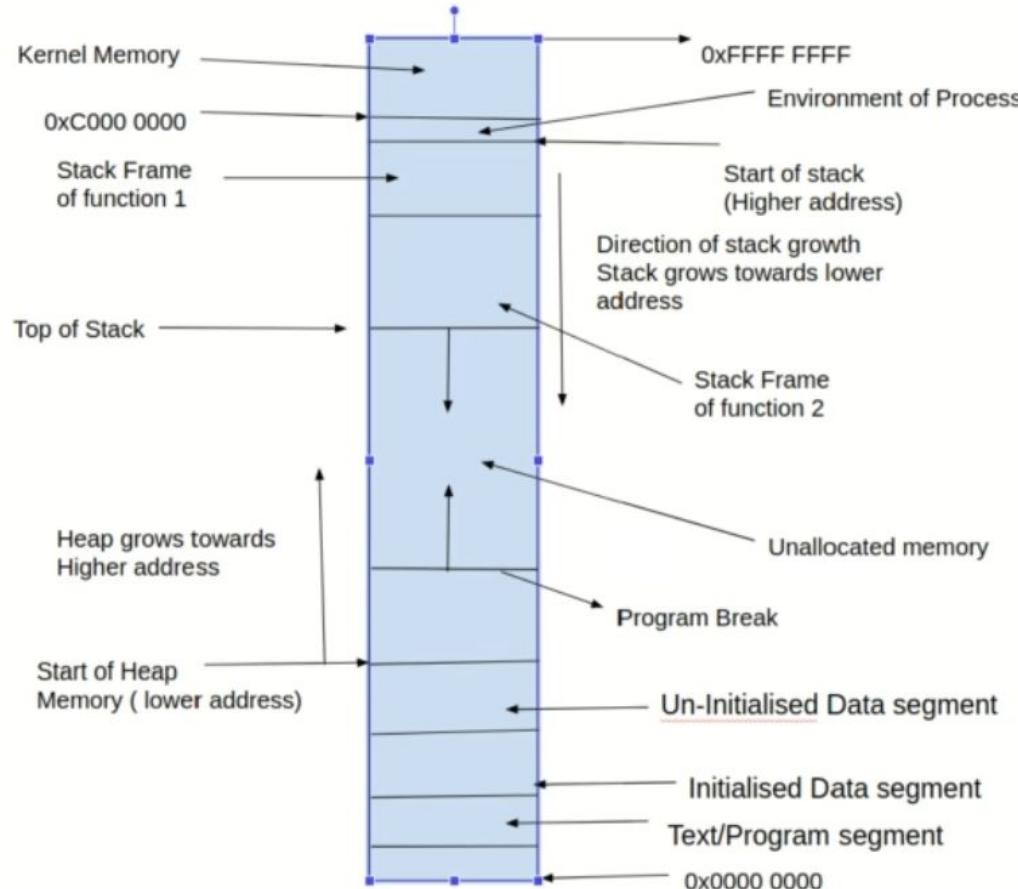
# Memory allocation

- A process can allocate dynamic memory by increasing the size of the heap, memory allocated is contiguous virtual memory that begins after the uninitialized data segment of a process in a virtual memory map. It grows and shrinks as memory is allocated and freed.

# Program Break

- The current limit of the heap is referred to as the 'program break'.
- When dynamic memory is allocated 'Program Break' increases.
- After the program break is increased, the process may access any address in the newly allocated area.

# Program Break



# Brk() and sbrk() - System Call

- brk()

```
#include <unistd.h>
```

```
int brk(void * mem_loc);
```

brk() on success, sets 'Program Break' to new memory location specified by brk().

success: Returns 0 on success,

Fail: -1 on error

- sbrk()

```
#include <unistd.h>
```

```
void *sbrk(intptr_t increment);
```

success: Returns 'program break' which is a virtual memory location on success,

Fail : (void \*) -1 on error, and 'errno' is set to ENOMEM (Out of memory).

intptr\_t : integer data type

# Allocating Memory on the Heap: malloc()

- In general, C programs use the malloc family of functions to allocate and deallocate memory on the heap. These functions are built over brk() and sbrk().

```
#include <stdlib.h>
```

- `void *malloc(size_t size );`  
Returns pointer to allocated memory on success, or NULL on error

# calloc()

- When process terminates, by default the memory of a process is freed. By good practise when the memory is utilised, it must be explicitly freed by programmer.
- The calloc() function allocates memory for an array of identical items

```
#include <stdlib.h>
void *calloc(size_t numitems , size_t size);
```

- Returns pointer to allocated memory on success, or NULL on error

# realloc()

```
#include <stdlib.h>
void *realloc(void * ptr , size_t size);
```

- Returns pointer to allocated memory on success, or NULL on error
- When we are increasing the size of the block of memory, realloc() attempts to form a single block.
- If the block lies at the end of the heap, then realloc() expands the heap.
- If the block of memory lies in the middle of the heap, and there is insufficient free space immediately following it, realloc() allocates a new block of memory and copies all existing data from the old block to the new block.
- Care must be taken in above case as the old memory location and newly allocated memory location may not both be same, hence storing memory address before realloc() and later using same memory after realloc for memory access may give wrong results.

# Free()

- Free() - function is used to free the dynamically allocated memory from the heap. This needs to be called only when memory is allocated using malloc() or calloc or realloc(). The syntax is.

```
#include <stdlib.h>
```

```
void free(void * ptr);
```

# free()

- free() doesn't lower the program break, it adds the freed block of memory to a list of free blocks that can be used by future calls to malloc().
- The block of memory being freed can be in the middle of the heap, rather than at the end of heap, so that lowering the program break is not done during free().

# Allocating Memory on the Stack: alloca()

- Like the functions in the malloc, alloca() allocates memory dynamically, but this memory is not allocated from heap, alloca() obtains memory from the stack by increasing the size of the stack frame of current executing function.

```
#include <alloca.h>

void *alloca(size_t size);
```

Returns pointer to allocated block of memory which is in stack virtual address.

# Fork() - system call

- Overview of fork()
- Parent and child process.
- Memory segment of parent and child process

# Man Pages for memory Allocation

- Please refer to below ‘MAN’ pages for further details of discussed functions

man 2 brk

man 2 sbrk

man malloc

man calloc

man realloc

man alloca

# Process creation

- Creating new process: fork()
- In many applications, creating multiple processes can be a useful way of dividing up a task.
- Dividing tasks up in this way often makes application design simpler. It also permits greater concurrency (i.e., more tasks or requests can be handled simultaneously).
- The fork() system call creates a new process(the child), which is an almost exact duplicate of the calling process (parent).

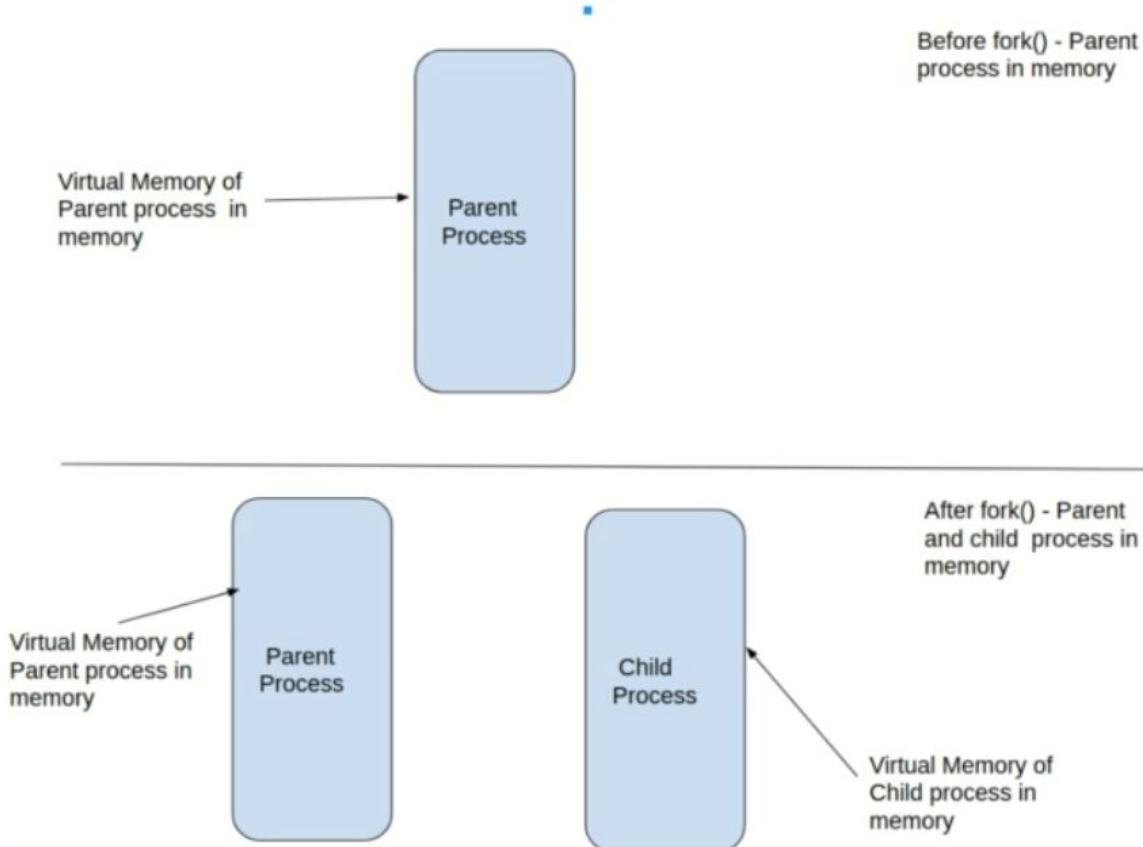
## Fork() - syntax

```
#include <unistd.h>
pid_t fork(void);
```

Note: Unlike other function calls, fork() return twice(on success), one in parent process, and other in child process.

- a. In parent: returns process ID of child  
on success, or -1 on error;
- b. In successfully created child: always returns 0

# Before and After Fork()



# Fork()

- An important point wrt fork() is - After fork() is executed successfully, two processes exist in memory, and each process, execution continues from the point where fork() returns.
- The child process has a “seperate copy” of all the memory segment like, text, data, stack and heap, hence Child process executes the same program as parent process.
- The child’s memory contents are initially exact duplicates of the corresponding parts the parent’s memory.
- After the fork(), each process can modify the variables in its stack, data, and heap segments without affecting the other process.

# Wait() and Exit()

- Wait() - system call – used to synchronise parent and child process and get exit status of child process.
- Exit() - system call – to terminate the process.

## Wait()

- Each process has a entry in process table.
- When a process ends, all of the memory and is deallocated(but a entry of exited process is still maintained in process table). The parent can read the child's exit status by executing the wait system call.
- The parent process receives signal SIGCHLD when child process dies.

# wait()

- Wait() - system call also gives information on how the child process exited.

```
#include <sys/wait.h>
pid_t wait(int * status);
```

# waitpid()

- Waitpid() - system call also gives information on how the child process exited.

```
#include <sys/wait.h>
```

```
pid_t waitpid(pid_t pid, int *status, int options);
```

# Execve() - Running New program

- Execve() - ‘system call’ loads a new program into a process’s memory, During this operation, the old program is discarded, and its virtual memory is replaced by new program’s virtual memory.
- The process’s stack, data, and heap are replaced by those of the new program.
- The new program starts its execution from main() function.

# execve()

```
#include <unistd.h>
int execve(const char * pathname , char *const
argv [], char *const envp []);
```

Never returns on success; returns -1 on error.

- Refer to “man 2 execve”

# The exec() family Library Functions

- execl()
- execlp()
- execle()
- execv()
- execvp()
- execvpe()
- Refer to “man exec” for syntax on manual pages.
- None of the above returns on success; all return -1 on error  
NOTE: execve() is a system call, whereas exec's shown in this section are library functions built over ‘execve()’.
- Examples of execl and execv.

# File Sharing Between Parent and Child

- When a fork() is performed, the child receives duplicates of all of the parent's file descriptors.
- This means that corresponding file descriptors in the parent and the child refer to the same file description.
- The open file description contains the current file offset (as modified by read(), write(), and lseek())
  - As a result, these attributes of an open file are shared between the parent and child.

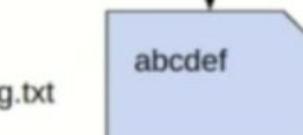
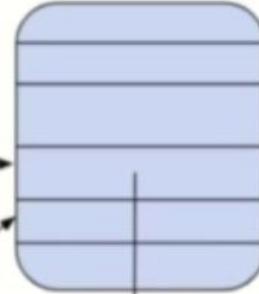
File descriptor table (Per process)

| Fd | Pointer to open file table |
|----|----------------------------|
| 0  |                            |
| 1  |                            |
| 2  |                            |
| 3  |                            |

Open file table(system Wide)

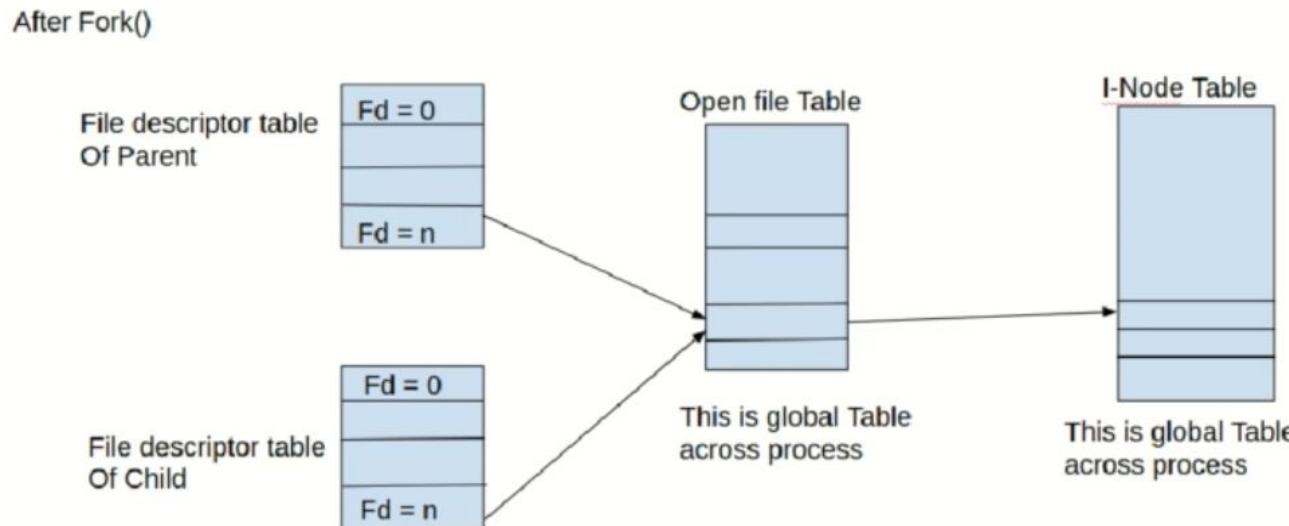
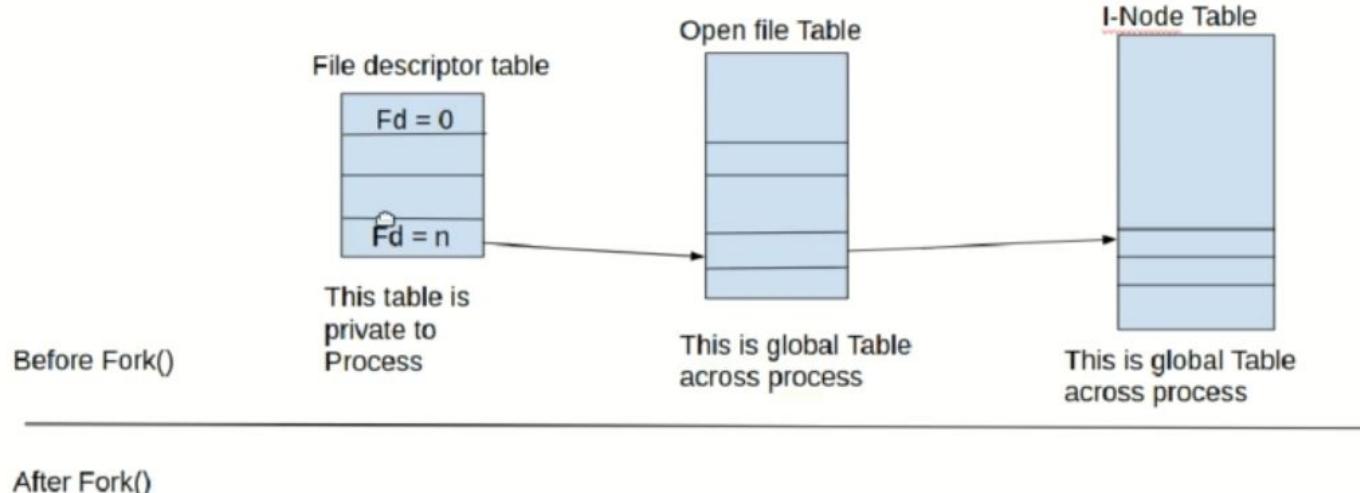
| File offset | File status flags | Pointer to i-node |
|-------------|-------------------|-------------------|
|             |                   |                   |
| 6           | O_RDWR            |                   |
|             |                   |                   |
|             |                   |                   |
|             |                   |                   |
|             |                   |                   |

I-Node Table  
(system wide)

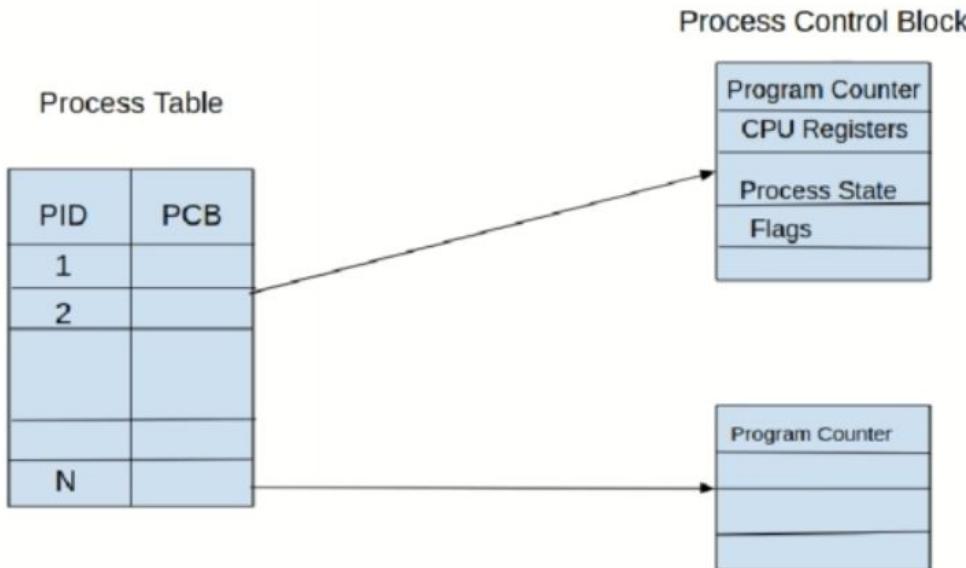


```
Fd = open("./log.txt", O_RDWR);
// 3 will be assigned to fd
```

log.txt



# Process table and Process Control Block



# PCB – Data structure of each process

There are many data members of PCB few of them are

- Program Counter
- Process number
- Process State
- CPU Registers
- Process Priority
- Memory management information
- List of open files of the process
- And many more.....

Note: the struct 'task\_struct' represents process control block in Linux

# Signals

- Signals are software interrupts that provide a mechanism for handling asynchronous events.
- These events can originate from outside the system, such as when the user generates the interrupt character by pressing Ctrl-C, or from activities within the system.
- In Linux there are different signals, and have a unique numbers, by which they are identified.

# signals

- Signals have a very precise lifecycle. First, a signal is raised, The kernel then stores the signal until it is able to deliver it. Finally the signals are delivered to corresponding process.

- The kernel can perform one of three actions, depending on what the process asked it to do
- “Ignore the signal” - No action is taken.

There are two signals that cannot be ignored: SIGKILL and SIGSTOP.

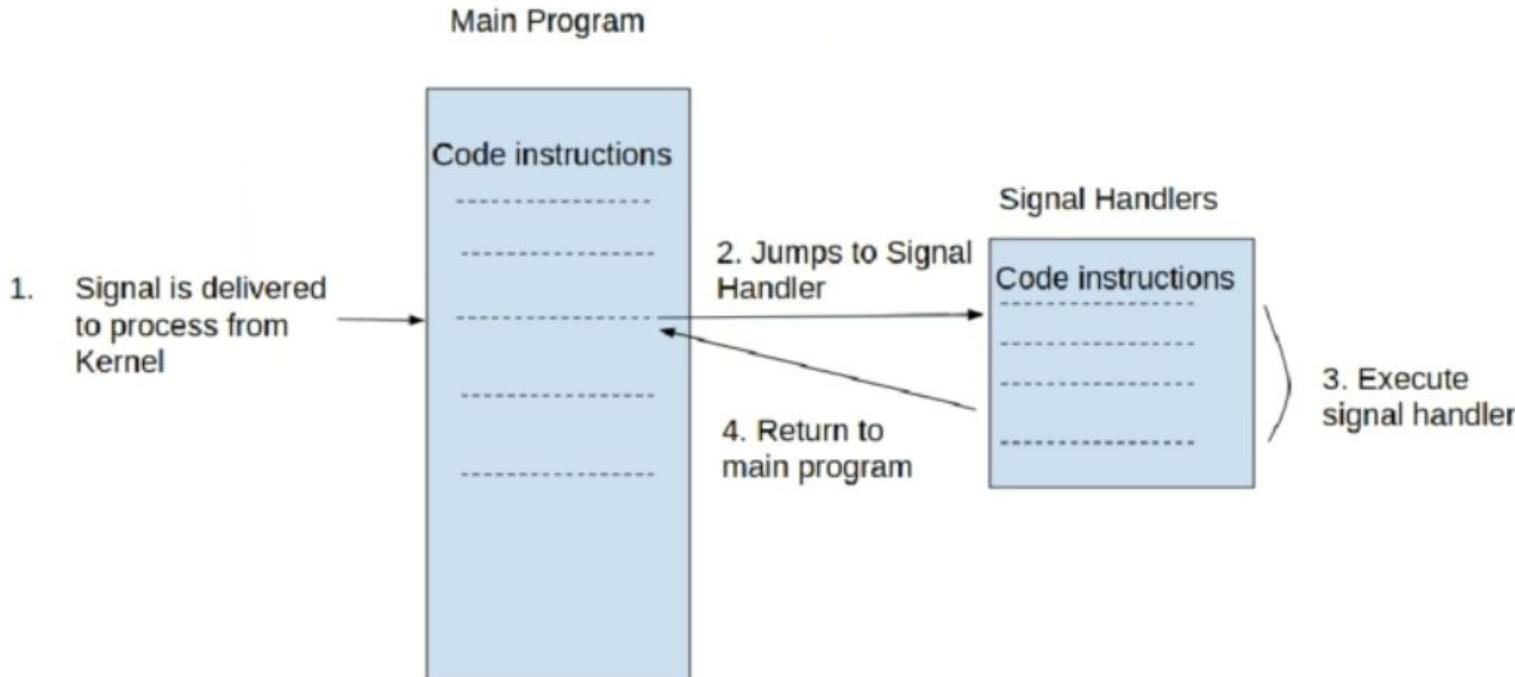
- “Catch and handle the signal” - The kernel will suspend execution of the process’s current code path and jump to a previously registered function. The process will then execute this function. Once the process returns from this function, it will jump back to wherever it was when it caught the signal.

# Signals

- “Perform the default action” - This action depends on the signal being sent. The default action is often to terminate the process.

Every signal will have its default behaviour incase the process has not handled it.

# How Signal handler work



- Signal Identifiers - Every signal has a symbolic name that starts with the prefix 'SIG'. For example, SIGINT is the signal sent when the user presses Ctrl-C.

SIGKILL is the signal sent when a process is forcefully terminated.

- These signals are all defined in a header file included from <signal.h>

# Signals list

- Signals are defined in signal.h header file

```
#define SIGHUP 1
#define SIGINT 2
#define SIGQUIT 3
#define SIGILL 4
#define SIGTRAP 5
#define SIGABRT 6
#define SIGIOT 6
#define SIGBUS 7
#define SIGFPE 8
#define SIGKILL 9
#define SIGUSR1 10
#define SIGSEGV 11
#define SIGUSR2 12
#define SIGPIPE 13
#define SIGALRM 14
#define SIGTERM 15
#define SIGSTKFLT 16
#define SIGCHLD 17
#define SIGCONT 18
#define SIGSTOP 19
#define SIGTSTP 20
#define SIGTTIN 21
#define SIGTTOU 22
#define SIGURG 23
#define SIGXCPU 24
#define SIGXFSZ 25
#define SIGVTALRM 26
#define SIGPROF 27
#define SIGWINCH 28
#define SIGIO 29
#define SIGPOLL SIGIO
/*
#define SIGLOST 29
*/
#define SIGPWR 30
```

# Default actions of signals

| POSIX signal      | default action |
|-------------------|----------------|
| SIGHUP            | terminate      |
| SIGINT            | terminate      |
| SIGQUIT           | coredump       |
| SIGILL            | coredump       |
| SIGTRAP           | coredump       |
| SIGABRT/SIGIOT    | coredump       |
| SIGBUS            | coredump       |
| SIGFPE            | coredump       |
| SIGKILL           | terminate(+)   |
| SIGUSR1           | terminate      |
| SIGSEGV           | coredump       |
| SIGUSR2           | terminate      |
| SIGPIPE           | terminate      |
| SIGALRM           | terminate      |
| SIGTERM           | terminate      |
| SIGCHLD           | ignore         |
| SIGCONT           | ignore(*)      |
| SIGSTOP           | stop(*)(+)     |
| SIGTSTP           | stop(*)        |
| SIGTTIN           | stop(*)        |
| SIGTTOU           | stop(*)        |
| SIGURG            | ignore         |
| SIGXCPU           | coredump       |
| SIGXFSZ           | coredump       |
| SIGVTALRM         | terminate      |
| SIGPROF           | terminate      |
| SIGPOLL/SIGIO     | terminate      |
| SIGSYS/SIGUNUSED  | coredump       |
| SIGSTKFLT         | terminate      |
| SIGWINCH          | ignore         |
| SIGPWR            | terminate      |
| SIGRTMIN-SIGRTMAX | terminate      |

# Signal explanations

- SIGABRT - The abort() function sends this signal to the process that invokes it. The process then terminates and generates a core dump file.
- SIGALRM - The alarm() and setitimer(), functions send this signal to the process that invoked them when an alarm expires

- **SIGBUS** – (Bus Error) This signal is raised when process has a memory access error .
- **SIGCHLD** - Whenever a process terminates or stops, the kernel sends this signal to the process's parent.

- SIGCONT - The kernel sends this signal to a process when the process needs to be resumed which is in a stopped state.
- SIGFPE - This signal represents any arithmetic exception.
- SIGILL - The kernel sends this signal when a process attempts to execute an illegal machine instruction.
- SIGINT - This is a interrupt signal. This occurs when users presses Ctrl-C from keyboard.
- SIGKILL - This signal is sent from the kill() system call.

- SIGPIPE - If a process writes to a pipe but the reader has terminated, the kernel raises this signal.
- SIGSEGV - This signal, whose name derives from segmentation violation, is sent to a process when it attempts an invalid memory access.
- SIGUSR1 and SIGUSR2 - These signals are available for user-defined purposes; the kernel never raises them.

These are few most used signals, there are few other signals also

# Basic Signal Management

```
#include <signal.h>
```

```
sighandler_t signal (int signo, sighandler_t my_handler);
```

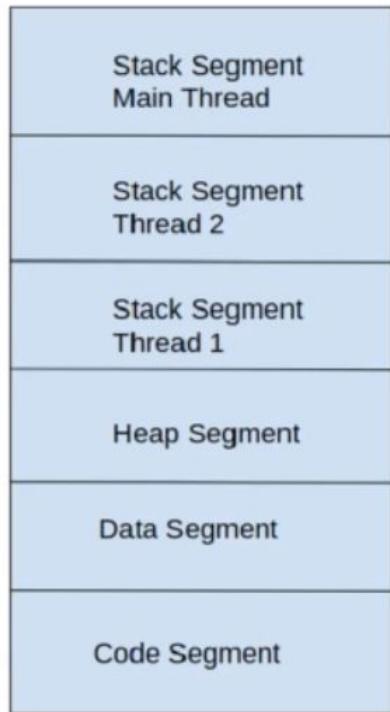
- void my\_handler (int signo);
- When a signal is delivered to a process, using signal(), the signal can either perform default action or ignore the particular signal.
- SIG\_DFL - Set the behavior of the signal given by signo to its default. For example, in the case of SIGPIPE , the process will terminate.
- SIG\_IGN - Ignore the signal given by signo
- Refer to examples.

# Sending a Signal

- The `kill()` system call, the basis of the common `kill` utility, sends a signal from one process to another:
- `#include <sys/types.h>`
- `#include <signal.h>`
- `int kill (pid_t pid, int signo);`

# Thread Overview

Process with Threads



# Threads in Linux

- Introduction to Threads(POSIX).
- Creating Thread.
- Thread ID
- Terminating Thread.

# pthread\_create

- When a program is started, the resulting process consists of a single thread, called the initial or main thread.
- The pthread\_create() function creates a new thread.

```
#include <pthread.h>
```

- `int pthread_create(pthread_t * thread , const  
pthread_attr_t * attr , void *(* start )(void *), void * arg );`
- Returns 0 on success, or a positive error number on error

# pthread creation

```
pthread_t *thread;
int s; +
s = pthread_create(&thread, NULL, func, &arg);
if (s != 0){
 Error creating thread.
}
```

# Compiling Pthreads programs

- The program is linked with the `libpthread` library (the equivalent of `-lpthread`).
- Eg: `gcc *.c -lpthread`

# Thread IDs

- Each thread within a process is uniquely identified by a thread ID.
- This ID is returned to the caller of `pthread_create()`.
- A thread can obtain its own ID using `pthread_self()`.

```
#include <pthread.h>
pthread_t pthread_self(void);
```

- A thread can get its own thread ID by calling `pthread_self()`.
- Applications of thread ID will be discussed next.

# Thread Termination

The execution of thread terminates in one of the below ways:

+<sub>x</sub>

- The thread's start function calls a return with return value for the thread.
- The thread calls `pthread_exit()`
- The thread is canceled using `pthread_cancel()`.
- Main thread performs a return from `main()` function, this causes all threads in the process to terminate immediately.

# pthread\_exit()

- The pthread\_exit() function terminates the calling thread, and specifies a return value, this value can be obtained in main thread by calling pthread\_join().

```
include <pthread.h>
```

```
void pthread_exit(void * retval);
```

- The retval argument specifies the return value for the thread.
- If the main thread calls pthread\_exit() instead of calling exit() or return() , then the other threads continue to execute.

# pthread\_join()

- The pthread\_join() function when called from main thread, waits for the specified thread to terminate.
- If that thread has already terminated, pthread\_join() returns immediately.

- Syntax:

```
int pthread_join(pthread_t thread , void ** retval);
```

Returns 0 on success, or a positive error number on error

## pthread\_join()

- The task that pthread\_join() performs for threads is similar to that performed by waitpid() for processes.
- The main use case of using pthread\_join is that the main thread should wait for all the threads it has created, and then only should terminate itself in the end.

# Detaching a Thread

- By default, a thread is joinable, this means that when it terminates, another thread can obtain its return value using `pthread_join()`.
- If a joinable thread is not joined by calling `pthread_join()`, then the terminated thread becomes a Zombie thread, consuming system memory resource.
- `pthread_detach()` - The system automatically cleans up terminated thread, no need of calling `pthread_join`.

```
#include <pthread.h>
```

- `int pthread_detach(pthread_t thread );`
- Returns 0 on success, or a positive error number on error

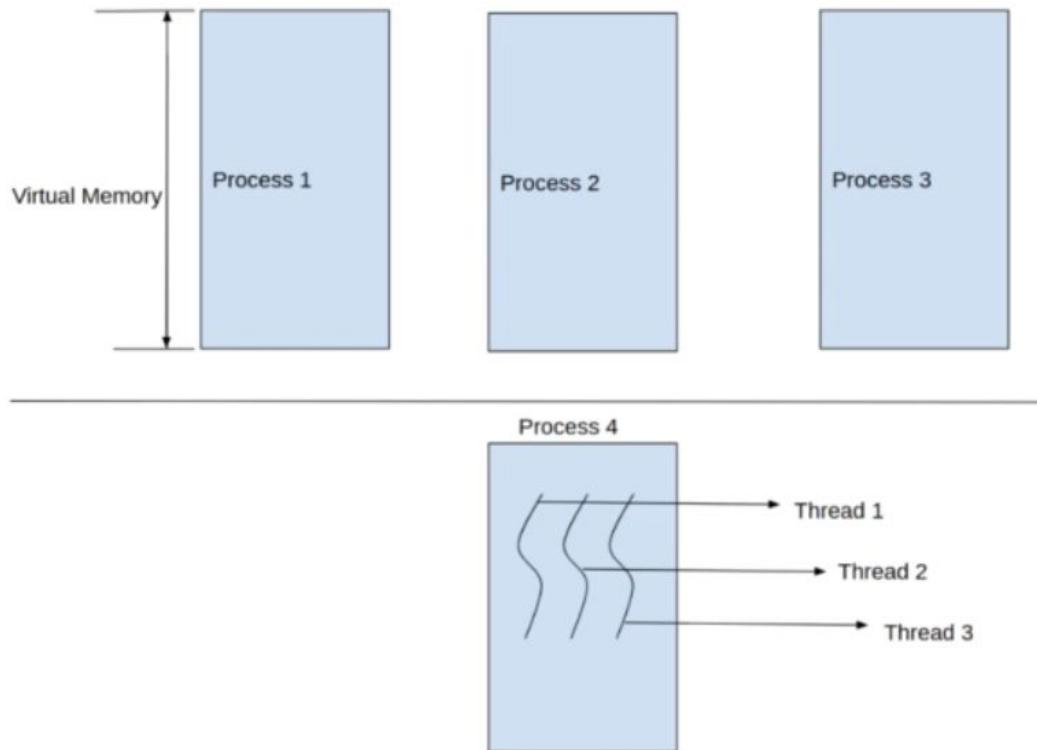
## Pthread\_cancel

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

Returns 0 on success, or a positive error number on error

- Used to cancel/terminate a specified thread.

# Threads vs Process



# Thread synchronisation

- MUTEX (Mutual Exclusion)
- Condition variables

# Overview

- One of the main advantages of threads are that they can share information through global variables.(data segment, heap segment)
- Programmer has to take care that multi-threads do not attempt to modify/read the same global variable at the same time.
- “critical section” is used to refer to a section of code that accesses a shared resource and whose execution should be atomic, else synchronisation issues arise.
- It means that the shared resources have to be used in protected manner, to overcome synchronisation issues, we will discuss different methods to overcome synchronisation issues.

# Incorrect method

- Lets see a Example program of incorrect way of multi-thread programming, which causes synchronisation issue while accessing global resource.

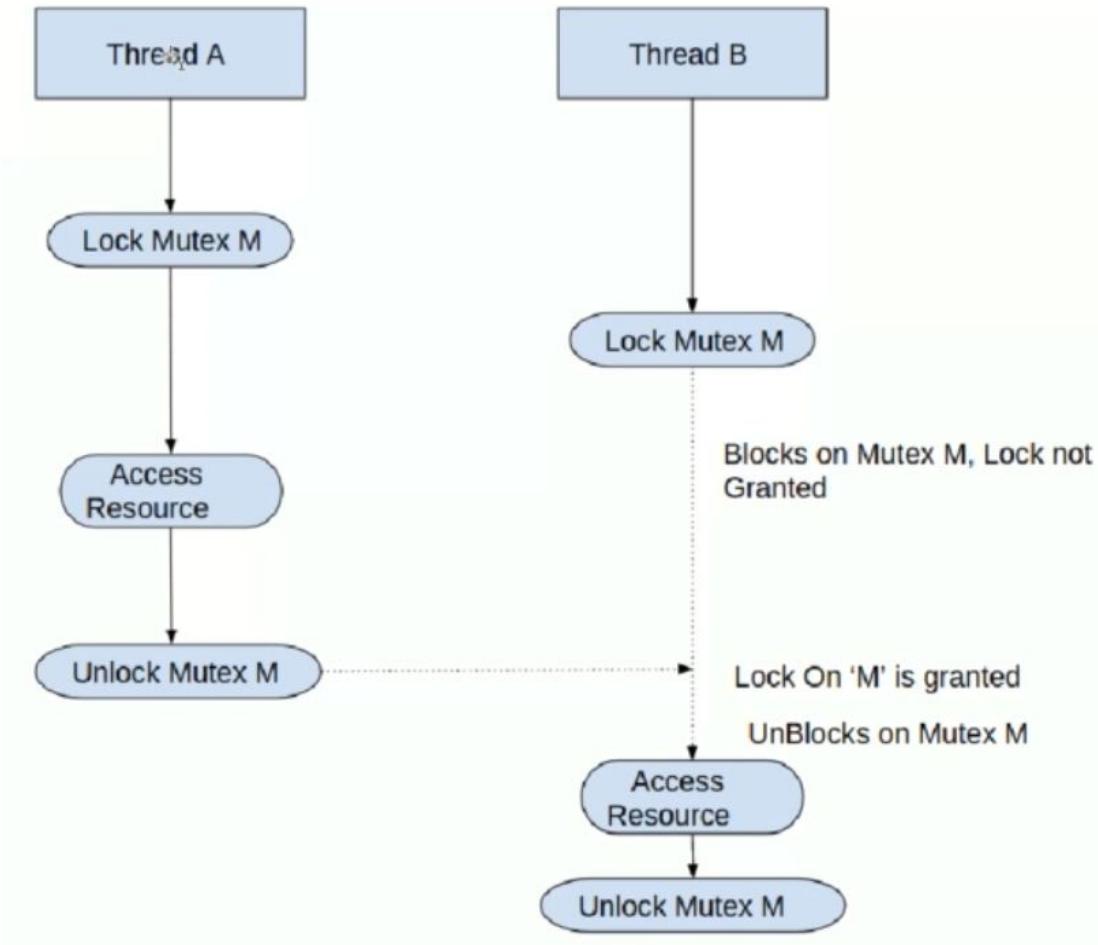
# Mutex

- Mutex is a type of lock(its data type is `pthread_mutex_t`).
- To avoid the problems that can occur when multiple threads try to access/modify a shared variable, we must use a mutex (short for mutual exclusion)
- More generally, mutexes can be used to ensure atomic access to any shared resource.
- A mutex has two states: locked and unlocked.
- At any moment, at most one thread may hold the lock on a mutex, and hence only that thread can execute the critical section.
- When a thread locks a mutex, it becomes the owner of that mutex. Only the mutex owner can unlock the mutex.

# Mutex operation

## Steps of Mutex operations

- Lock the mutex for the shared resource;
- Access the shared resource, perform operations on shared resource as required, and
- Unlock the mutex.



# Statically Allocated Mutexes

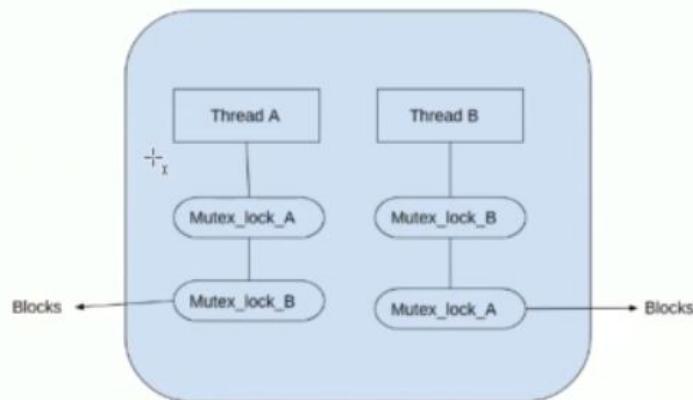
- A mutex can either be allocated as a static variable or be created dynamically at run time.
- `pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;`

# Mutex syntax

- `#include <pthread.h>`
- `int pthread_mutex_lock(pthread_mutex_t * mutex );`
- `int pthread_mutex_unlock(pthread_mutex_t * mutex );`
- Both return 0 on success, or a positive error number on error

# Dead lock

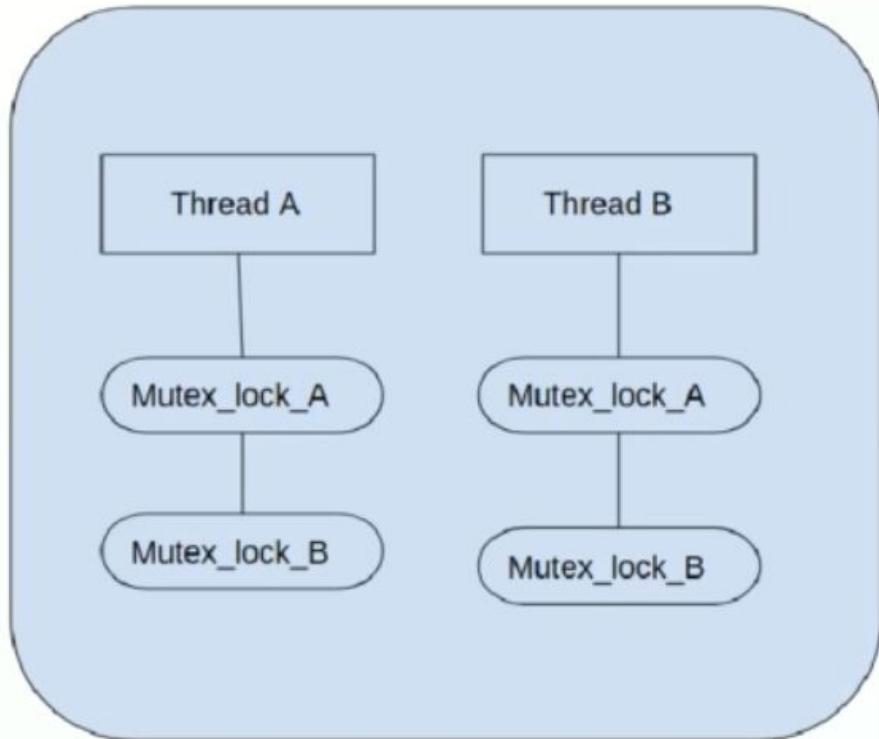
- Sometimes, a thread needs to simultaneously access two or more different shared resources, each of which is governed by a separate mutex



- Dead lock can also be caused, if a thread has already locked a mutex, and same thread is trying to lock the same mutex again.

# Avoiding Dead locks

- The simplest way to avoid such deadlocks is to define a mutex hierarchy
- Threads can lock the same set of mutexes, they should always lock them in the same order.
- For example, in before scenario, the deadlock could be avoided if the two threads always lock the mutexes in the order mutex1 followed by mutex2.



## `pthread_mutex_trylock` and `pthread_mutex_timedlock`

- The Pthreads API provides two variants of the `pthread_mutex_lock()` function:
- `pthread_mutex_trylock()` and `pthread_mutex_timedlock()`.
- The `pthread_mutex_trylock()` function is the same as `pthread_mutex_lock()`, except that if the mutex is currently locked, `pthread_mutex_trylock()` fails, returning the error `EBUSY`.

## `pthread_mutex_trylock` and `pthread_mutex_timedlock`

- The `pthread_mutex_timedlock()` function is the same as `pthread_mutex_lock()`
- Except that the caller can specify an additional argument, `abstime`, that places a limit on the time that the thread will sleep while waiting to acquire the mutex.
- If the time interval specified by its `abstime` argument expires without the caller becoming the owner of the mutex, `pthread_mutex_timedlock()` returns the error `ETIMEDOUT`.

# Signaling Changes of State: Condition Variables

- A condition variable allows signalling from one thread to other thread, about changes in the state of a shared variable.
- Condition variables help to define the sequence of thread execution, for eg: In case of producer and consumer application, first the producer thread runs, and then the producer thread signals the consumer thread for further execution.

# Condition variable

- In condition variable the thread waiting for shared resource will be made to sleep, and as soon as it gets signal from other thread, the thread wakes up and executes(like accessing/modifying shared resource).
- A condition variable is always used in conjunction with a mutex.

# Statically Allocated Condition Variables

- condition variables can be allocated statically or dynamically.
- `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`

# Signaling and Waiting on Condition Variables

- The principal of condition variable is ‘signal and wait’.
- The ‘signal’ operation is a notification to one or more waiting threads that a shared variable’s state has changed.
- The ‘wait’ operation is the method of blocking until such a notification is received from other Thread.

# Syntax

- The `pthread_cond_signal()` functions signal's the condition variable specified by cond.
  - The `pthread_cond_wait()` function blocks thread until the condition variable cond is signaled.
- 
- `#include <pthread.h>`
  - `int pthread_cond_signal(pthread_cond_t * cond );`
  - `int pthread_cond_wait(pthread_cond_t * cond ,  
pthread_mutex_t * mutex );`
- All return 0 on success, or a positive error number on error

# Working principle of Condition Variable

```
pthread_cond_wait(pthread_cond_t * cond ,
pthread_mutex_t * M)
```

- Step1 – The thread calling pthread\_cond\_wait unlocks Mutex – M.
- Step2 – Blocks on the condition variable ‘cond’ to receive the signal from other thread
- Step3 – As soon as it receives required signal, it further locks Mutex – M.

Note: Step1 & Step 2 is executed as a atomic operation.

# Inter Process Communication

- IPC are used to send/receive data between Process.
- IPC are also used to synchronise between process.
- IPC are usually Communication based or Synchronisation Based

# Communication based IPC

- Communication based IPC are of 2 types
  - 1) data transfer based
    - example – PIPE, FIFO, message queue and socket
  - 2) memory sharing based
    - example – Shared Memory
- Synchronisation based
  - 1) semaphore
  - 2) Mutex (used for threads)
  - 3) Condition variables(used for threads).

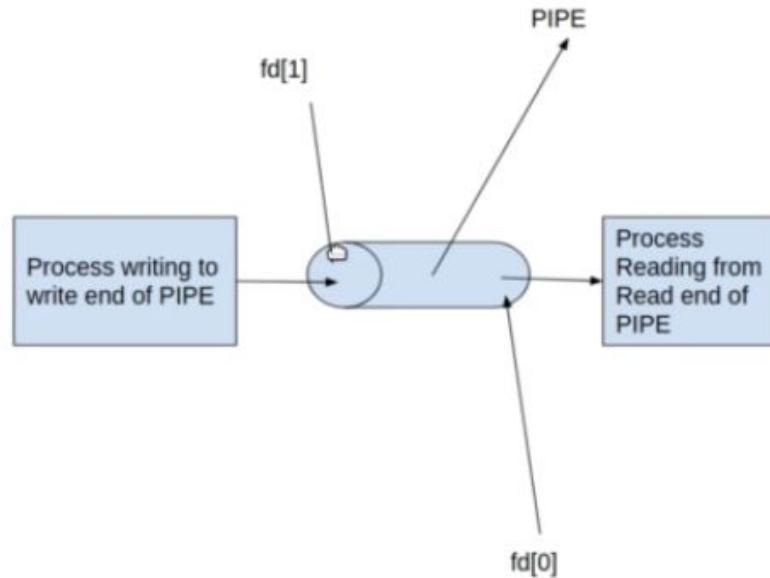
# PIPE and FIFO for IPC

# PIPES

- A PIPE is a byte stream used for IPC.
- A PIPE has 2 ends.
- A read end.
- A write end.
- Refer to diagram below.

# Pipes

- 

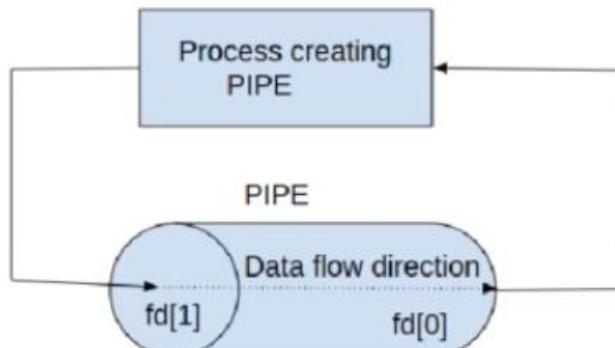


# Features of PIPES

- Pipes are uni-directional.
- Pipes have limited capacity(usually 64 K bytes). A pipe is simply a buffer maintained in kernel memory.
- When a PIPE is full with data, further writes to PIPE will be blocked, until the receive end process removes the data from PIPE.
- Writing to PIPE puts data, whereas reading data from PIPE removes the data from PIPE.

# Pipe creation

- `#include <unistd.h>`
- `int pipe(int fd[2]);`
- `Pipe()` open ‘two file descriptor’, one in `fd[0]`, other in `fd[1]`
- Returns 0 on success, or –1 on error



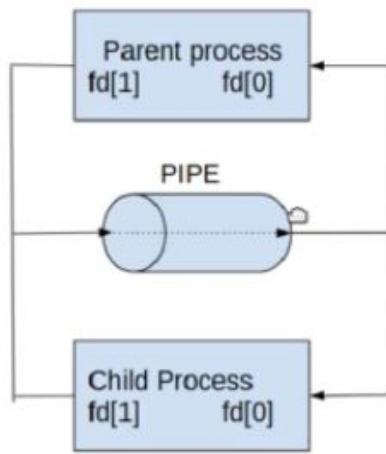
# Application of Pipes

- Pipes are used to communicate across related process, ex – Parent , child process.
- After fork() - the child process inherits copies of its parent's file descriptors.

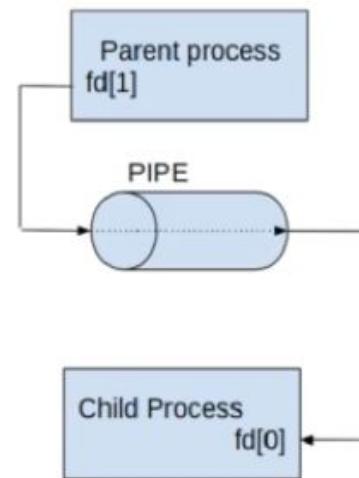
# Pipe after fork()

- 

After fork()



After closing  
unused descriptor



# SIGPIPE signal in Pipes

- SIGPIPE signal(broken pipe signal) is sent to write end process when read end is closed.
- Example Programs of PIPES.

# FIFO - First In First Out.

- A FIFO is similar to a pipe.
- Also called 'name Pipes'.
- The principal difference between PIPES and FIFO
  - A) FIFO has a name within the file system and is opened in the same way as a regular file, whereas PIPES does not have a name in file system.
  - B) FIFO is used for communication between 'unrelated processes' (e.g: a client and server), but PIPES are used to send data only between related processes (e.g: parent-child process)

# FIFO

- Once a FIFO has been opened, we use the same I/O system calls as are used with pipes and other files (i.e., read(), write(), and close()).
- Just as with pipes, a FIFO has a write end and a read end, and data is read from the FIFO in the same order as it is written (This means the first data written to write end of fifo, is the first read out from reading end of fifo)

## Create a fifo

- #include <sys/stat.h>
- int mkfifo(const char \* pathname , mode\_t mode );
- Returns 0 on success, or -1 on error.
- Example programs of FIFO.

# POSIX MESSAGE QUEUE

# Posix – Message Queue

- Message queues can be used to pass messages between processes.
- Unlike Pipes and Fifo which are ‘Byte’ oriented IPC, whereas Message Queue is ‘Message’ oriented IPC.
- Readers and writers communicate each other in units of messages.
- POSIX message queues permit each message to be assigned a priority.
- Priority allows high-priority messages to be queued ahead of low-priority messages.
- Message Queue entry is present in file system in /dev/mqueue.

# Message Queue – system calls

The below are the system calls related to Posix Message queue

- mq\_open()
- mq\_close()
- mq\_unlink()
- mq\_send(),mq\_receive()
- mq\_setattr(), mq\_getattr()

# Message Queue operations

- The `mq_open()` - function creates a new message queue or opens an existing queue, returning a message queue descriptor for use in later calls.
- The `mq_send()` - function writes a message to a queue.
- The `mq_receive()` - function reads a message from a queue.
- The `mq_close()` - function closes a message queue that the process previously opened.
- The `mq_unlink()` function removes a message queue name and marks the queue for deletion when all processes have closed it.

# Opening a message queue

```
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>
```

- `mqd_t mq_open(const char *name, int oflag);`
- `mqd_t mq_open(const char *name, int oflag, mode_t mode,  
struct mq_attr *attr);`
- Returns a message queue descriptor on success, or `(mqd_t) -1` on error

Note: The 'name' should always start with '/'

EG: /my\_queue

# mq\_open()

Oflag – Specify the different options in which the message queue can be opened. One of the below flag has to be specified.

- O\_RDONLY
- O\_WRONLY
- O\_RDWR

Below one or more flags can be Ored

- O\_CREAT
- O\_EXCL
- O\_NONBLOCK

Refer to below manual pages for more details .

man 3 mq\_open

- Mode – If O\_CREAT is specified in 'oflag', then mode refers to the permission in which the message queue is created(Read /write)

Note: oflag and mode used here are similar to parameters used in open() system calls used to open regular files .

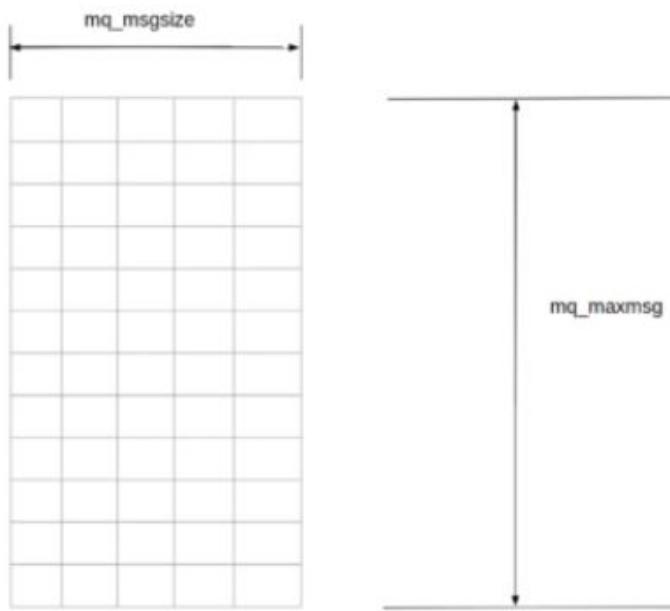
# Attributes of message Queue

```
struct mq_attr {
 long mq_flags;
 long mq_maxmsg;
 long mq_msgsize;
 long mq_curmsgs; // Number of messages currently in queue
}
```

- mq\_maxmsg - Maximum number of messages in queue.
- mq\_msgsize - Maximum size of a message (in bytes).
- mq\_curmsgs - Total Number of messages currently in queue.

# Analogous of Message Queue

- 



## mq\_send() and mq\_receive()

```
#include <mqueue.h>
int mq_send(mqd_t mqdes , const char * msg_ptr , size_t msg_len ,
unsigned int msg_prio);
```

- Returns 0 on success, or -1 on error

```
#include <mqueue.h>
ssize_t mq_receive(mqd_t mqdes , char * msg_ptr , size_t msg_len ,
unsigned int * msg_prio);
```

- Returns number of bytes in received message on success, or -1 on error

# Closing Message queue

- Closing a message queue
- The `mq_close()` function closes the message queue descriptor mqdes.

# POSIX SEMAPHORE

# Semaphore

- POSIX semaphores allows processes and threads to synchronize access to shared resources.
- Semaphores are used to execute the critical section(resource share by different process/Threads) in an Atomic manner.
- Named semaphores - This semaphore has a name as specified in `sem_open()`.  
(on linux present in `/dev/shm`)
- Unnamed semaphores: This type of semaphore doesn't have a name, it resides at an location in memory.

# Named Semaphore

List of functions used by Named semaphores

- `sem_open()`
- `sem_post()`
- `sem_wait()`
- `sem_getvalue()`
- `sem_close()`
- `sem_unlink()`

# Named Semaphore

- The `sem_open()` function opens or creates a semaphore, initializes the semaphore and returns a handle for use in later calls.
- The `sem_post()` and `sem_wait()` functions respectively increment and decrement a semaphore's value.

# Named Semaphore

- The `sem_getvalue()` function retrieves a semaphore's current value.
- The `sem_close()` function removes the calling process's association with a semaphore that it previously opened.
- The `sem_unlink()` function removes a semaphore name and marks the semaphore for deletion when all processes have closed it.

# Open a named semaphore

- Refer to 'man sem\_open' for more details

```
#include <fcntl.h> /* For O_* constants */
#include <sys/stat.h> /* For mode constants */
#include <semaphore.h>
```

```
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,
 mode_t mode, unsigned int value);
```

- Link the program with -lpthread.

## sem\_open()

- If O\_CREAT is specified in oflag, then a new semaphore is created if one with the given name doesn't already exist.
- If O\_CREAT is specified in flags, then two further arguments are required: mode and value
- Mode – similar as file open mode - O\_RDONLY  
O\_WRONLY , and O\_RDWR.
- [Note] – when a program is opening an existing semaphore, we need to take care of the 'oflag', as most of the times to perform semaphore operations, both 'read' and 'write' permission is required.<sup>+</sup>

## sem\_open()

- Value – Is an unsigned integer that specifies the initial value to be assigned to the new semaphore.
- When a child is created via fork(), it inherits references to all of the named semaphores that are open in its parent. After the fork(), the parent and child can use these semaphores to synchronize their actions.

# Closing a Semaphore

```
#include <semaphore.h>
int sem_close(sem_t * sem);
```

- Returns 0 on success, or -1 on error

# Removing a Named Semaphore

- The `sem_unlink()` function removes the semaphore identified by name and marks the semaphore to be destroyed once all processes stops using it.

```
#include <semaphore.h>
int sem_unlink(const char * name);
```

Returns 0 on success, or -1 on error

# Semaphore Operations

- POSIX semaphore is an integer that the system never allows to go below 0.
- The `sem_post()` and `sem_wait()` functions increment and decrement a semaphore's value by exactly one.

## `sem_wait()`

- If the semaphore currently has a value greater than 0, `sem_wait()` returns immediately. If the value of the semaphore is currently 0, `sem_wait()` blocks until the semaphore value rises above 0, at that time, the semaphore is decremented and `sem_wait()` returns.

# Retrieving the Current Value of a Semaphore

- The `sem_getvalue()` function returns the current value of the semaphore referred to by 'sem' in the int pointed to by 'sval'.

```
#include <semaphore.h>
int sem_getvalue(sem_t * sem , int * sval);
```

Returns 0 on success, or -1 on error

# Unnamed Semaphores

- Unnamed semaphores (also known as memory-based semaphores) are variables of type `sem_t` that are stored in memory allocated by the application.
- The semaphore is made available to the processes or threads that use it by placing it in an area of memory that they share.

Note: Named semaphore was present in file system similar to a regular file, whereas a un-named semaphore does not exist on file system, rather on Volatile memory like RAM.

# Unnamed Semaphores

- Operations on unnamed semaphores use the same functions (`sem_wait()`, `Sem_post()`, `sem_getvalue()`, and so on) that are used to operate on named semaphores. In addition, two further functions are required.
- The `sem_init()` - function initializes a semaphore and informs the system of whether the semaphore will be shared between processes or between the threads of a single process.
- The `sem_destroy(sem)` - function destroys a semaphore.

These functions(`sem_init()` and `sem_destroy()`) should not be used with named semaphores.

# Initializing an Unnamed Semaphore

```
#include <semaphore.h>
int sem_init(sem_t * sem , int pshared , unsigned int value);
```

Returns 0 on success, or -1 on error

- The pshared argument indicates whether the semaphore is to be shared between threads or between processes.
- If pshared is 0, then the semaphore is to be shared between the threads of the calling process.
- If pshared is nonzero, then the semaphore is to be shared between processes.

# Destroying an Unnamed Semaphore

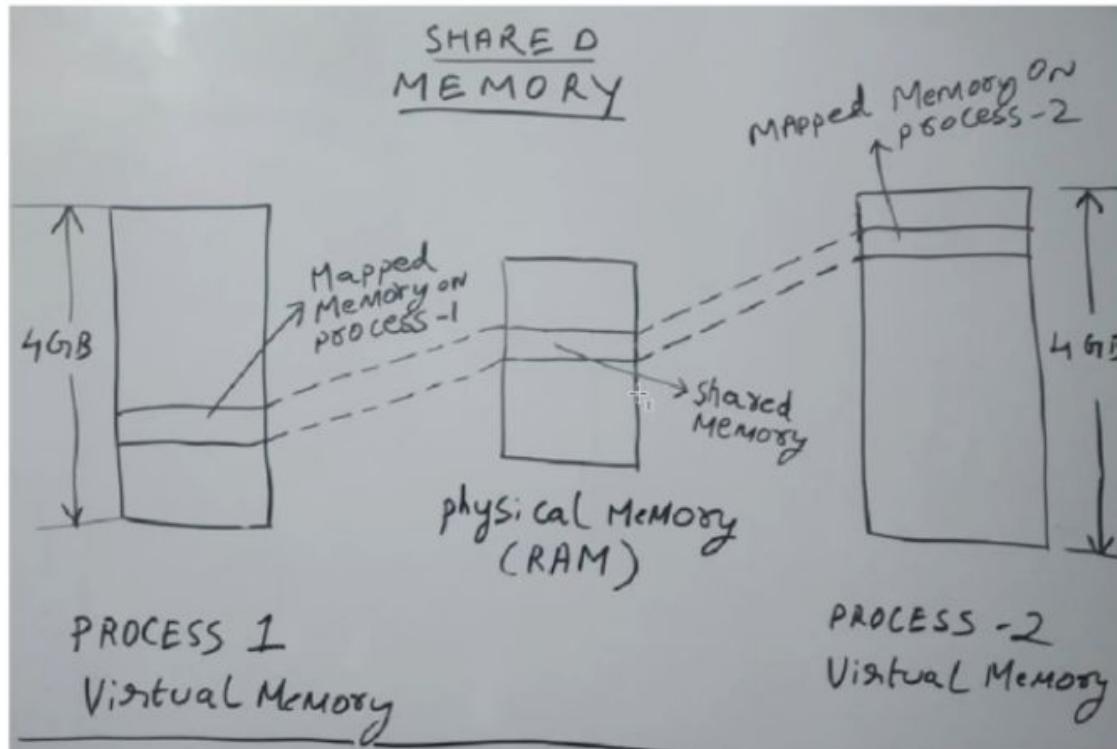
- The `sem_destroy()` function destroys the semaphore `sem`, which is an unnamed semaphore that was previously initialized using `sem_init()`. It is good practice to destroy a semaphore only if no processes or threads are waiting on it.

```
#include <semaphore.h>
int sem_destroy(sem_t * sem);
```

Returns 0 on success, or -1 on error

Posix Shared memory

## SHARED MEMORY



Step 1 - Create shared Memory

Step 2 - Define size of shared Memory

Step 3 - Map shared Memory on individual process

# Shared Memory

- POSIX shared memory allows to share a mapped memory region between unrelated processes.
- Shared memory entries are present in '/dev/shm'.
- shared memory the fastest IPC mechanism, as no extra kernel data structures are involved.

In other IPC, if data needs to be sent from say process-1 to process-2, then data from process-1 user space to kernel space needs to be copied, then again from kernel space to user space of process-2 needs to be performed.

# Shared Memory

To use a POSIX shared memory object, we perform two steps:

Step 1.

- Use the `shm_open()` function to open an object with a specified name.
- The `shm_open()` function is analogous to the `open()` system call.
- It either creates a new shared memory object or opens an existing one.
- `shm_open()` returns a file descriptor referring to the shared memory.

Note: The shared memory created newly is of length '0' bytes.

# Shared Memory

Step 2.

- Define the length of shared memory.

Step 3

- File descriptor obtained in the previous step is referenced in `mmap()`.
- This maps the shared memory object into the process's virtual address space.

Note: Any read/write to process's Virtual memory, will be actually read/written from shared memory(Part of Physical RAM memory)

# Creating Shared Memory Objects

- The `shm_open()` function creates and opens a new shared memory object or opens an existing shared memory. The arguments to `shm_open()` are analogous to those for `open()`.

```
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>
```

```
int shm_open(const char * name , int oflag , mode_t mode);
```

- Returns file descriptor on success, or `-1` on error

# shm\_open()

- Oflag- Can take below one or more OR'ed members
    - O\_RDONLY – open existing shared memory for read only purpose.
    - O\_RDWR - open existing shared memory for read /write purpose.
    - O\_TRUNC – To truncate shared memory to zero length.  
If shared memory needs to be created newly then  
'O\_EXCL' is used.
- Note – The 'shm\_open()' is similar to 'open()' for regular files.

# Setting Shared memory size

- When a shared memory is newly created, its size is zero.
- User needs to assign size required for the shared memory, this is similar to creating a buffer with given size.

```
#include <unistd.h>
int ftruncate(int fd , off_t length);
```

- Fd is the file descriptor obtained from `shm_open()`, and length is the required length in bytes of shared memory.

# Mapping shared memory to process virtual Memory

- `mmap()` is used to map the shared memory created to the process virtual map.

```
#include <sys/mman.h>
```

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

Input values  
+  
addr

'addr' – Address of process Virtual memory to which the shared memory is mapped, If 'addr' is NULL, then the kernel chooses the address at which to create the mapping.

Length – length of the created shared memory.

Prot – describes the memory protection of mapping(PROT\_READ for read and PROT\_WRITE for write)

flags - for shared memory flags use value 'MAP\_SHARED' describing that the memory can be shared, i.e updates to this mapped memory is visible among Process.

Fd – file descriptor obtained from `shm_open()`.

# Mapping shared memory to process virtual Memory

Returns Values of `mmap()`

On success - The virtual memory of process, to which the shared memory is mapped.

On error - The value `MAP_FAILED` (that is, `(void *) -1`) is returned, and `errno` is set to indicate the cause of the error.

Note : Refer to below command on terminal for more information.

`man 2 mmap`

Thank You!!