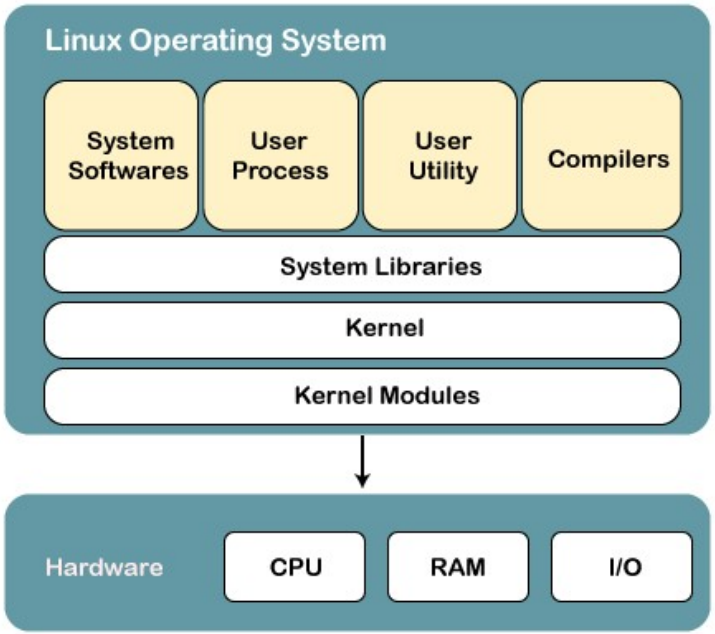
OS Lab

Palash Das @IIT Jodhpur, India

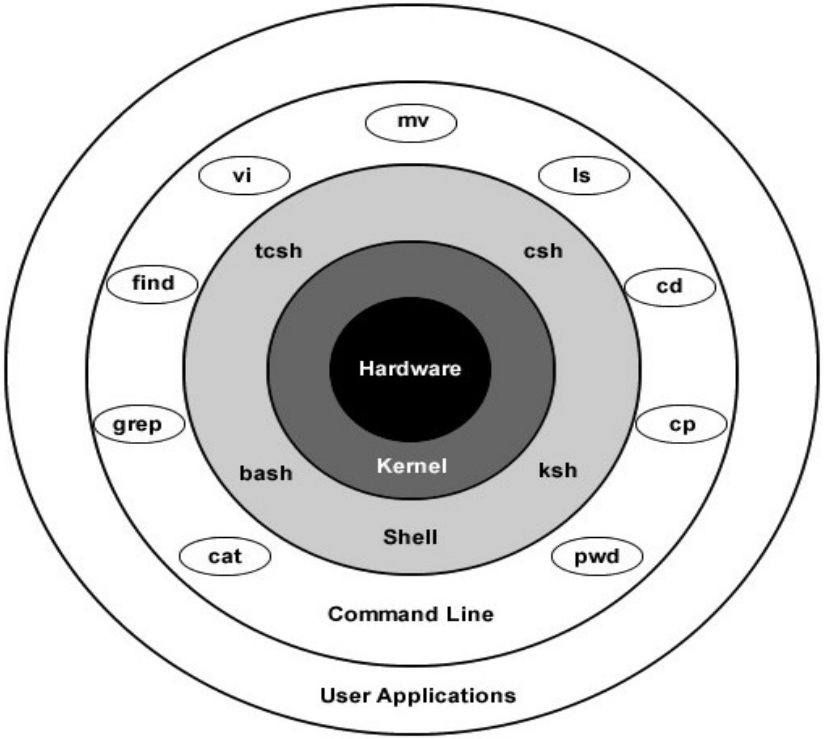
Introduction to Linux



- **Kernel**
  - Core of an OS.
  - Communication between device and software
  - Major activities: device management, memory management, process management, and handling system calls.
- **System Libraries**
  - Helps in accessing kernel features.
  - Kernel features are accessed through various system calls which are OS specific. Programmers have developed a standard library of procedures to communicate with the kernel. Each operating system supports these standards, and then these are transferred to system calls for that operating system.

Developed by **Linus Torvalds** in **1991**

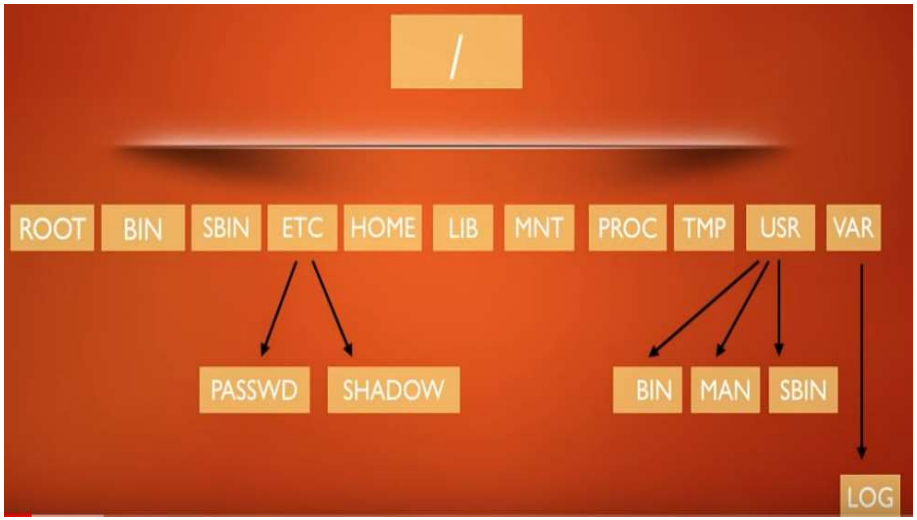
Unix Architecture



Two ways:

1. Program→ System library→ System Calls → Kernel → H/W
2. Command→ pack into shell script (optional)→ shell → Kernel→ H/W

Directory Structure of Linux



- Root directory (/) is in top of the hierarchy, contains all files and folders, similar to C drive.
- Root subdirectory → administrator, all privileges (RWE any files)
- BIN and SBIN contains binaries (cat, cp, etc.) and system binaries like fdisk, reboot (executable programs for commands).
- ETC → configuration files, account info (SHADOW file), passwords (PASSWD file) etc.
- Home → Any User's home directory.
- LIB → Common libraries (loader, linker, etc) used by the system. Contains library images needed for bootup.
- MNT →Temporary file systems like USB.
- PROC → Virtual file system stores kernel info.
- TMP → Temporary data are stored and deleted once reboot is done.
- USR → stores user's program and data.
- VAR → System logs.

Account Related

$ and #

sudo su or sudo su [login name]

whoami

passwd

id (uid = 0 for root)

*** Open terminal and try all commands.

Additional Commands

- `hostname` # name of this computer
- `echo "Hello, world"` # print characters to screen
- `echo $HOME` ⟷ Path of User directory # print environment variable
- `echo my login is $(whoami)` # replace $(xx) with program output
- `date` # print current time/date
- `cal` # print this month's calendar
- `shazam` # bad command

File System

cd [dir name] or cd ~
pwd
locate filename (finds files in Linux using the file name)
ls with –l or –a (long listing format, all)
mount /dev/sdb1 /mnt and umount /mnt.
(can be mounted to a mount point i.e. /mnt,
use lsblk to know disk names)
mkdir cp source des
rm, rmdir, (rm –rf * = dell all).
mv [file] [destination]
file [file] // Identify the file type

gedit filename (nano, vi, vim, etc.)
cat filename
less filename (Read one page at a time)
head –n N [file] //display first N lines
tail –n N [file] //display last N lines
tac [file] //display file(s) in reverse order
touch [file] //update modification time
| (piping) // used to combine two or more
Commands e.g. cat with less.
Echo ‘hello’ > file.txt (Overrides)
Echo ‘hi’ >> file.txt (Appends)

Text Processing

diff file1 file2// Compares two files also shows the positions where it differs

grep //search text for a pattern grep [options] pattern [files]

sed // stream editor performs operation like searching, find and replace, insertion or deletion.

sort // sort text files. SORT command is used to sort a file, arranging the records in a particular order. By default, the sort command sorts file assuming the contents are ASCII. Various options can also be used.

split // split files. Used to split large files into smaller files. It splits the files into 1000 lines per file(by default) and even allows users to change the number of lines as per requirement.

uniq // filter out the repeated lines in a file

wc // line, word, and character count (print newline, word, and byte counts)

Running a program

$PATH // PATH is an environment variable that instructs a Linux system in which directories to search for executables. The PATH variable enables the user to run a command without specifying a path.

./a.out // If the program does not exist in the path variable use ./ in front else command not found.

ps aux (aux option all process from all users including root) // List of running programs in a system e.g. ps aux | less → CTRL+Z → bg →jobs →fg 1

CTRL+C and CTRL+Z (Terminates and pauses {send the program in background} the program, respectively)

bg // Background program

jobs // Background program

fg job number // Brings background program to foreground

awk: powerful text processing

1. ps, ps | awk ‘print $1’. (you can play around a little) By default the **delimiter** is space but you can change it.
2. cat /etc/passwd→ all users of a system with all the details
    1. awk –F “:” ‘{print $1}’ /etc/passwd → Guess the output ☺
    2. awk –F “:” ‘{print $1 “\t” $3}’ /etc/passwd

3. df | awk ‘/\/dev\/loop/ {print $2 “\t” $3}’ → awk on disk free command

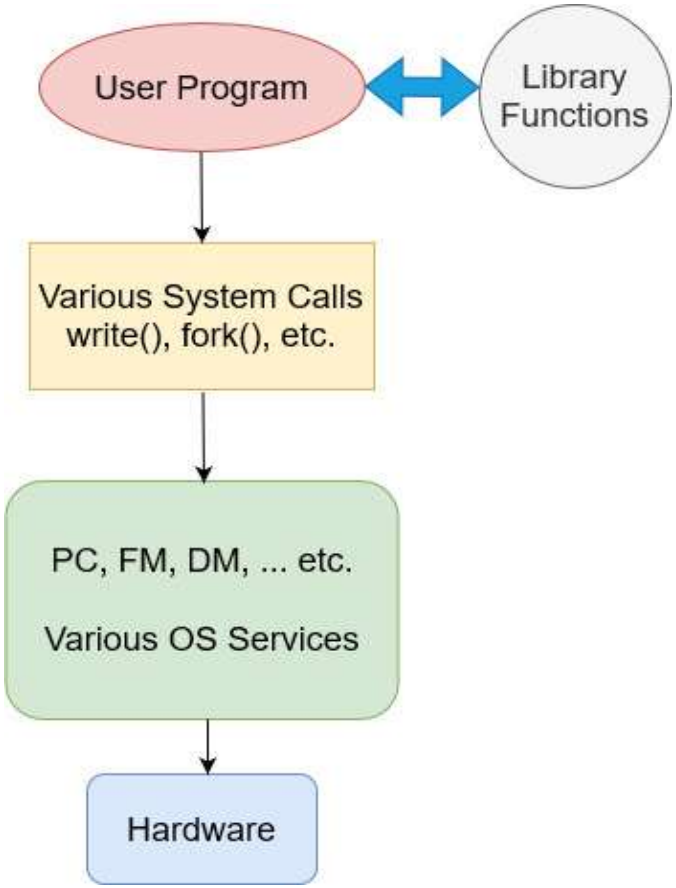
Try Yourself

SSH

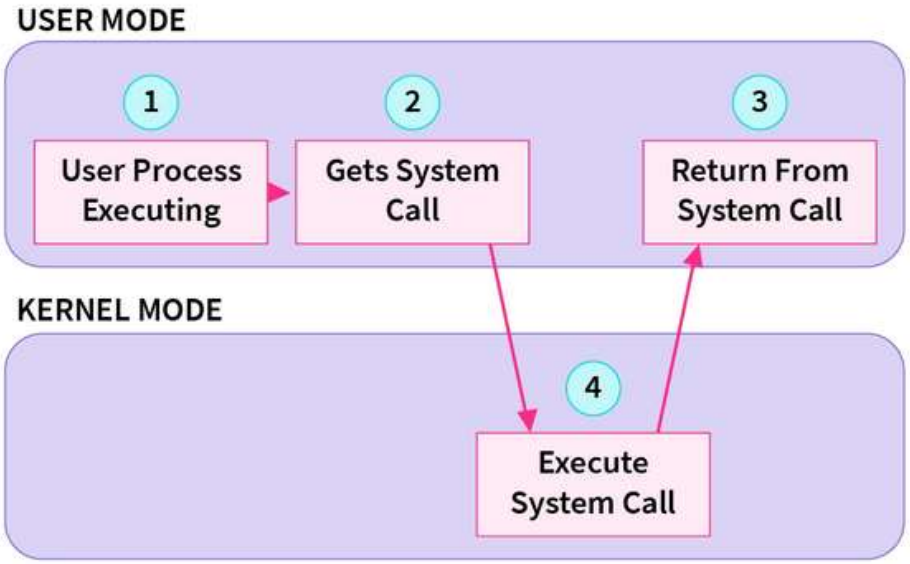
SCP

What is System call?

- ✓ It is a mechanism in which *user program requests services from the kernel* of an OS.
- ✓ It is a help for user programs to interact with OS.
- ✓ It is a kind of entry point into kernel.
- ✓ Your program needs to use some resources ----> Use System Calls.



Working of a System call



Various System Calls

There are primarily five different categories of system calls –

1.Process control (Process Creation, deletion, etc.).
2.File management (File create, del, read/write etc.).
3.Device management (Request/release of a device etc.).
4.Information maintenance (Information about system like time, data, etc.).
5.Communication (Inter-process communication).
6. Protection (Access control).

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

System Calls on Process Control

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
int main()
{
        pid_t p;  // The pid_t data type is a signed integer type capable of representing a process ID.
        printf("Child is not created yet\n");
        p = fork();
        if (p<0)
        {
                printf ("Error!!!");
        }
        if(p==0)    /// p = 0 child
        {
                printf("Hello!!! I am child and my id is %d\n", getpid());
                printf("My parent has an id of %d\n", getppid());
        }
        else    //// p > 0 for parents
        {
                printf("My child's has an id of %d\n", p);
                printf("I am parent with an id of %d\n", getpid());
        }
        printf("This part is present is both the child and parent\n");
}
```

Fork () Return values:

- 0 to the child process.
- Process id of child to the parent.
- -1 to parent if child creation fails.

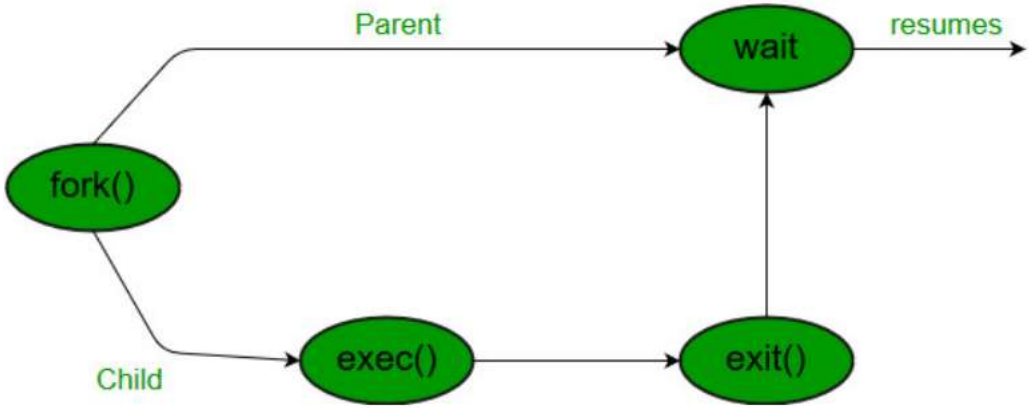
This part will be present in both the Parent and Child process.

```
int main()
{
   fork();
   fork();
   fork();
   printf("hello\n");
   return 0;
}
```

Output ???????????

wait() System Call

wait() system is used to wait for state changes in a child of the calling process and obtain information about the child whose state has changed.



State change means:

1. The child terminates.
2. The child was stopped by a signal.
3. The child was resumed by a signal.

on success, returns the process ID of the terminated child; on error, -1 is returned.

Example of wait ()

```
#include<unistd.h>
#include<sys/types.h>
#include<stdio.h>
#include<sys/wait.h>
int main()
{
        pid_t p;
        printf("before fork\n");
        p=fork();
        if(p==0)//child
        {
                sleep(10);
                printf("I am child having id %d\n", getpid());
                printf("My parent's id is %d\n", getppid());
        }
        else//parent
        {
                wait(NULL); /// waits for child to finish (can be placed down as well)
                printf("My child's id is %d\n", p);
                printf("I am parent having id %d\n", getpid());
        }
        printf("Common\n");
}
```

Output:

```
before fork
I am child having id 16154
My parent's id is 16153
Common
My child's id is 16154
I am parent having id 16153
Common
```

```
#include<unistd.h>
#include<sys/types.h>
#include<stdio.h>
#include<sys/wait.h>
int main()
{
        pid_t p;
        printf("before fork\n");
        p=fork();
        if(p==0)//child
        {
                wait(NULL); /// selfish child
                printf("I am child having id %d\n", getpid());
                printf("My parent's id is %d\n", getppid());
        }
        else//parent
        {
                sleep(2);
                printf("My child's id is %d\n", p);
                printf("I am parent having id %d\n", getpid());
        }
        printf("Common\n");
}
```

***Use **NULL** if you do not care about the state change.

Detailed Syntax of wait ()

pid_t wait(int *status); /// You want to check the state change information use parameter or make it null

pid_t waitpid(pid_t pid, int *status, int options); /// You want to wait for a specific process.

```
int main()
{
        pid_t p;
        int w1, wstatus;
        printf("before fork\n");
        p=fork();
        if(p==0)     //child
        {               printf("I am child having id %d\n", getpid());
                        printf("My parent's id is %d\n", getppid());
        }
        else         //parent
        {
                        w1 = wait(&wstatus);       /// w1 is the id of the terminated process
                        printf("Status is %d\n", WIFEXITED((wstatus))); # 1 represent terminated normally
                        printf("Terminated process %d\n", w1);
                        printf("My child's id is %d\n", p);
                        printf("I am parent having id %d\n", getpid());
        }
        printf("Common\n");
}
```

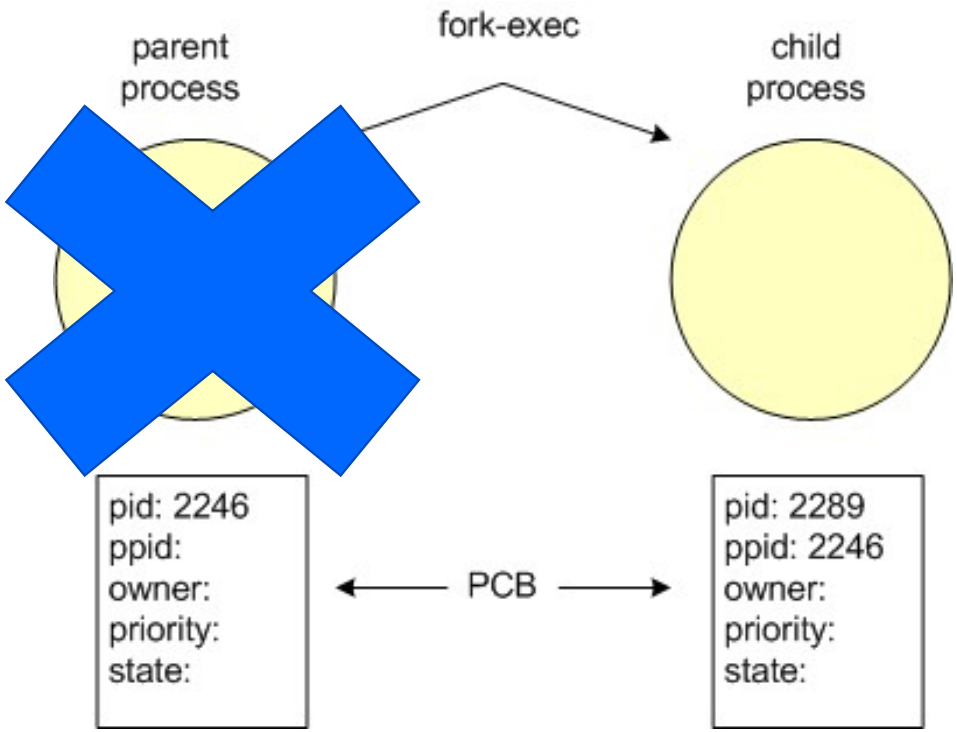
wait () for specific process

```
int main()
{
        pid_t p, q;
        printf("before fork\n");
        p=fork();
        if(p==0)     //child 1
        {              printf("I am child having id %d\n", getpid());
                       printf("My parent's id is %d\n", getppid());
        }
        else         //parent
        {

                       q = fork();
                       if (q==0)    //child 2
                       {
                                      printf("I am 2nd child having id %d\n", getpid());
                                      printf("My (2nd ) parent's id is %d\n", getppid());
                       }
                       else{
                                      waitpid (q, NULL, 0);
                                      printf("My 1st  child's id is %d\n", p);
                                      printf("My 2nd  child's id is %d\n", q);
                                      printf("I am parent having id %d\n", getpid());
                       }
        }
        printf("Common\n");
}
```

```
before fork
I am 1st child having id 18702
My (1st) parent's id is 18701
Common
I am 2nd child having id 18703
My (2nd ) parent's id is 18701
Common
My 1st  child's id is 18702
My 2nd  child's id is 18703
I am parent having id 18701
Common
```

***By default, **waitpid**() waits only for terminated children, but this behavior is modifiable via the *options* argument (3rd).

Orphan Process



Example of Orphan Process

```
int main()
{
        pid_t p;
        printf("before fork\n");
        p=fork();
        if(p==0)//child
        {
                sleep(2);
                printf("I am child having id %d\n", getpid());
                printf("My parent's id is %d\n", getppid());
        }
        else//parent
        {
                printf("My child's id is %d\n", p);
                printf("I am parent having id %d\n", getpid());
        }
        printf("Common\n");
}
```

Output:

before fork
My child's id is 52260
I am parent having id 52259
Common
I am child having id 52260
My parent's id is 1
Common

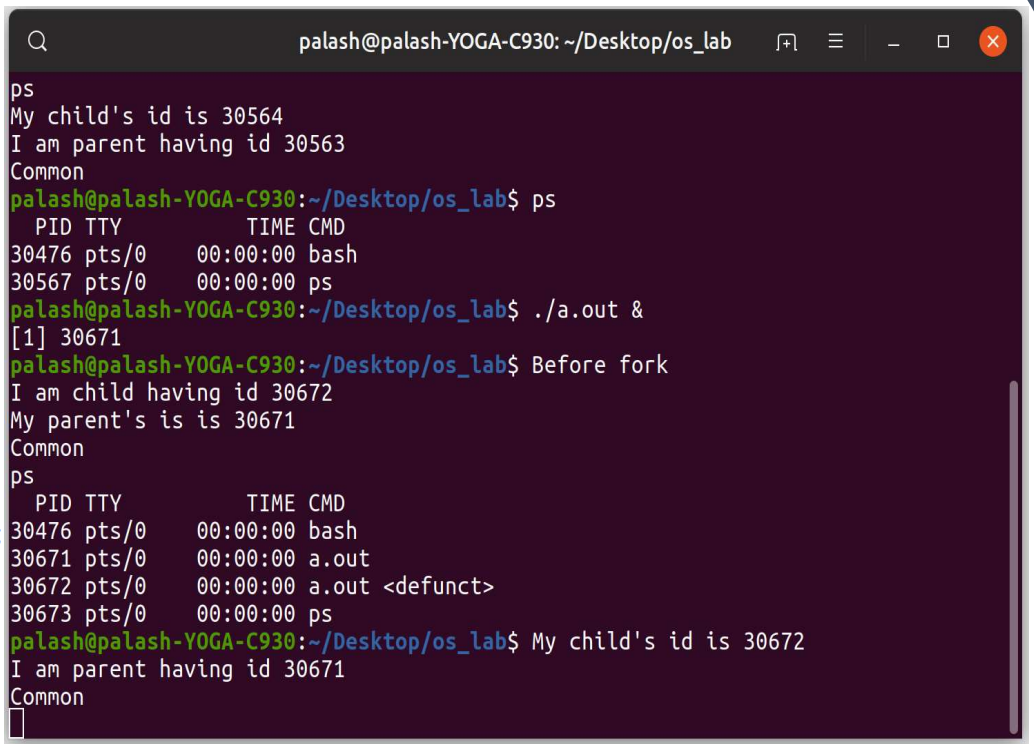
*****Parent's id is getting changed.**
Child has become an Orphan ☹

Zombie Process

- A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process.
- Parent thinks my child is still alive though it is terminated.

Example of Zombie

```
int main()
{
        pid_t p;
        printf("before fork\n");
        p=fork();
        if(p==0)//child
        {
                printf("I am child having id %d\n", getpid());
                printf("My parent's id is %d\n", getppid());
        }
        else//parent
        {
                sleep(3);
                printf("My child's id is %d\n", p);
                printf("I am parent having id %d\n", getpid());
        }
        printf("Common\n");
}
```



****Use wait() to get rid of zombies..... (Try your self and check)

execl()

Replaces the current process's image with a new one.

```
#include<stdio.h>
#include<unistd.h>
int main()
{
        printf("Before\n");
        execl("/bin/ps","ps",NULL); // 1st = Path of the process, 2nd = Name of the process, 3rd argument onwards
                                    //= arguments for the new process , when ever you want to end use NULL
        printf(After\n");
}
```

Another example of execl()

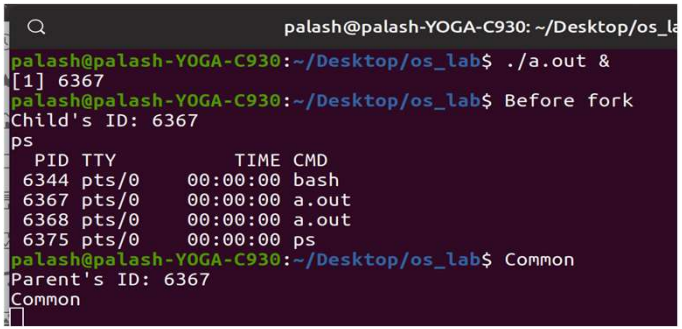
```c
#include<stdio.h>
#include<unistd.h>
int main()
{
        printf("Before\n");
        //sleep(10);
        execl("/home/palash/Desktop/os_lab/sum", "sum", "2", "3",NULL);
        printf("After\n");
}
```

```
palash@palash-YOGA-C930:~/Desktop/os_lab$ gcc sum.c -o sum
palash@palash-YOGA-C930:~/Desktop/os_lab$ gcc exe1.c
palash@palash-YOGA-C930:~/Desktop/os_lab$ ./a.out
Before
hipalash@palash-YOGA-C930:~/Desktop/os_lab$ gcc sum.c -o sum
palash@palash-YOGA-C930:~/Desktop/os_lab$ ./sum 2 3
Sum of 2, 3 is: 5
palash@palash-YOGA-C930:~/Desktop/os_lab$ gcc exe1.c
palash@palash-YOGA-C930:~/Desktop/os_lab$ ./a.out
Before
Sum of 2, 3 is: 5
palash@palash-YOGA-C930:~/Desktop/os_lab$
```

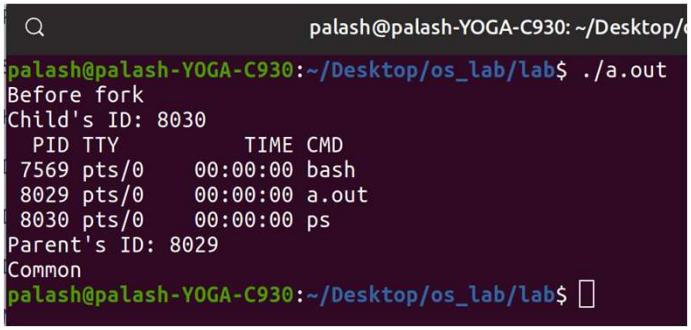
```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
        int a,b,sum;
        if(argc!=3)
        {
                printf("please use \"prg_name value1 value2 \"\n");
                return -1;
        }

        a = atoi(argv[1]);
        b = atoi(argv[2]);
        sum = a+b;

        printf("Sum of %d, %d is: %d\n",a,b,sum);

        return 0;
}
```

How about different code for child?

```
int main ()
{
        pid_t q;
        printf("Before fork\n");

        q = fork();
        if(q==0)
        {
                printf("Child's ID: %d\n", getppid());
                sleep(10);
        }
        else
        {
                wait(NULL);
                printf("Parent's ID: %d\n", getpid());
        }
        printf("Common\n");
}
```




```
int main ()
{
        pid_t q;
        printf("Before fork\n");
        q = fork();
        if(q==0)
        {
                printf("Child's ID: %d\n", getppid());
                execl("/bin/ps", "ps", NULL);
                sleep(10);
        }
        else
        {
                wait(NULL);
                printf("Parent's ID: %d\n", getpid());
        }
        printf("Common\n");
}
```



Inter-Process Communication popen()

- IPC means: One process will send the data while other will receive the data.
- The popen() function **executes the command specified by the string command**. It creates a pipe between the calling program and the executed command and returns a pointer to a stream that can be used to either read from or write to the pipe.
  - popen() opens a pipe stream to or from a process. You can either send or receive the data between processes.
  - Popen() is unidirectional.
- A pipe simply refers to a temporary software connection between two programs or commands.

Syntax:

#include<stdio.h>

File * popen (const char * command, const char * type) -----> Process for communication, Send or receive data

When you open a pipe, you also should close it. Use int pclose (FILE *stream).

Example1: popen()

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>

int main()
{
        FILE *rd;
        char buffer[50];
        sprintf(buffer, “Hello world");
        rd = popen("wc -c", "w"); // wc -c -> is the process which counts the number of characters passed.  2nd parameter is "w" which means pipe is opened in writing mode
        fwrite (buffer, sizeof(char), strlen(buffer), rd); // to write the data into the pipe. (data, sending data one char at a time, length of the data, write data to the appropriate pipe)
        pclose(rd);
}
```

Example2: popen()

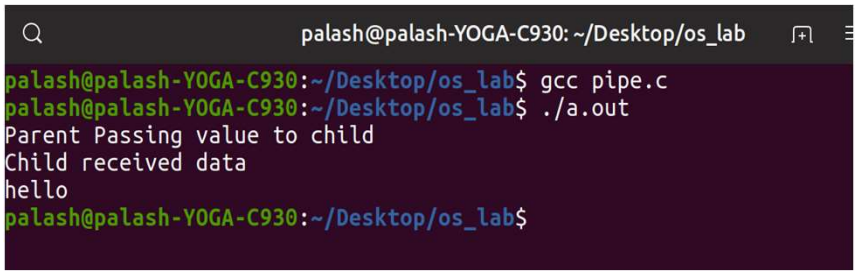
```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
int main()
{
        FILE *rd;
        char buffer[50];
        rd=popen("ls","r"); //pipe opened in reading mode
        fread(buffer, 1, 40, rd); //(Where to store the received data, how am I receiving, how much data, source)
        printf("%s\n", buffer); // or write(1, buffer, 40) // screen, source, amount of data
        pclose(rd);
}
```

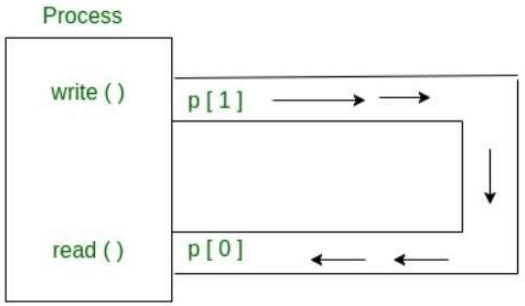
IPC: pipe()

```c
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
int main()
{
        int fd[2],n;
        char buffer[100];
        pid_t p;
        pipe(fd); //creates a unidirectional pipe with two end fd[0] and fd[1]
        p=fork();
        if(p>0) //parent
        {
                printf("Parent Passing value to child\n");
                write(fd[1],"hello\n",6); //fd[1] is the write end of the pipe
        }
        else // child
        {
                printf("Child received data\n");
                n=read(fd[0],buffer,100); //fd[0] is the read end of the pipe
                write(1,buffer,n);
        }
}
```

Properties:

- Pipe is one-way communication
- One process write to the pipe, and the other process reads from the pipe.
- Pipe is an area of main memory that is treated as a ***"virtual file"***.
- One process can write to this "virtual file" or pipe and another related process can read from it.
- If a process tries to read before something is written to the pipe, the process is suspended until something is written.
- Return 0 on success, -1 on error.
- Pipe behave like a **queue** data structure.

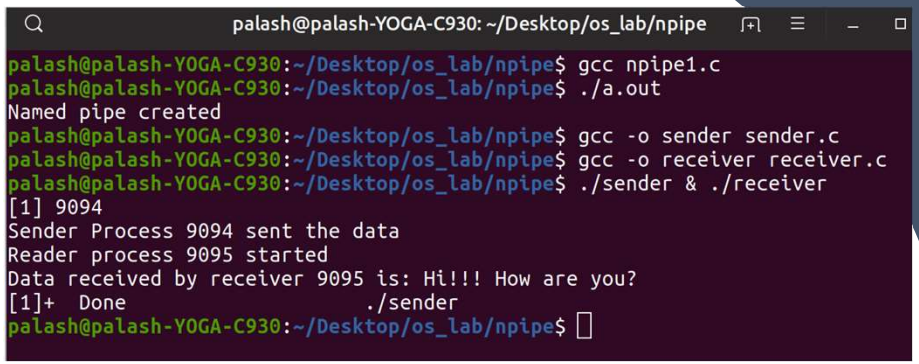




IPC using named pipe()

Named Pipe by mkfifo()

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
int main()
    {
        int res;
        res = mkfifo("npipe", 0777); //creates a named pipe with the name npipe, all permissions are given to user, group, and others.
        printf("Named pipe created\n");
    }
```



Sender Process

```
#include<unistd.h>
#include<stdio.h>
#include<fcntl.h>
    int main()
    {
            int res,n;
            res=open("npipe",O_WRONLY);
            write(res,"Hi!!! How are you?",18);
            printf("Sender Process %d sent the data\n",getpid());
    }
```

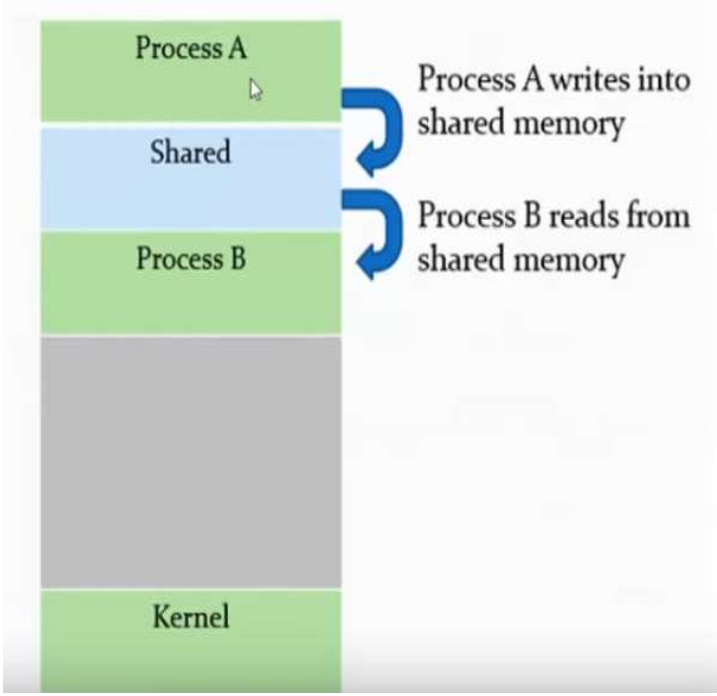
Receiver Process

```
#include<unistd.h>
#include<stdio.h>
#include<fcntl.h>
    int main()
    {
            int res,n;
            char buffer[100];
            res=open("npipe",O_RDONLY);
            n=read(res,buffer,100);
            printf("Reader process %d started\n",getpid());
            printf("Data received by receiver %d is: %s\n",getpid(), buffer);
    }
```

IPC using Shared Memory

- Shared Segment is not a part of Process A or B.
- Sender will create a shared segment.
- Sender will attached the shared segment to its address space.
- After attachment, the sender can write the date.

- Receiver will first attach itself with shared segment.
- Receiver then can read data from the region.



Two important functions

shmget(): It is used to create the shared memory segment.

shmat(): It is used to attach the shared segment with the address space of the process.

*** If we do not attach the shared segment with a process, it remains unusable.***

Syntax

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (key_t key, size_t size, int shmflg);
```

Identifies the shared segment

Size of the shared Segment (Byte)

Permissions on the shared segment

On success, returns a valid Identifier which will be used by shmat to attach

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

void *shmat (int shmid, const void *shmaddr, int shmflg);
```

Where to attach
If we do not know the address, we put “NULL”

0 if 2nd param is NULL

***If shmaddr is a NULL pointer, the segment is attached at the first available address as selected by the system.

Two Programs

Program 1 – Sender

- Create the shared segment.
- Attach to it.
- Write some content into in.

Program 2 – Receiver

- Attach itself to the shared segment.
- Read the value written by program 1

Sender

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
int i, n;
void *shared_memory;
char buff[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666|IPC_CREAT); //creates shared memory segment with key 2345, having size 1024 bytes. IPC_CREAT is used to
                                                 create the shared segment if it does not exist. 0666 are the permisions on the shared segment
printf("Key of shared memory is %d\n",shmid);
shared_memory=shmat(shmid,NULL,0); //process attached to shared memory segment
printf("Process attached at %p\n",shared_memory); //this prints the address where the segment is attached with this process
printf("Enter some data to write to shared memory\n");
n=read(0,buff,100); //get some input from user
buff[n] = '\0';
strcpy(shared_memory,buff); //data written to shared memory
printf("You wrote : %s\n",(char *)shared_memory);
}
```

Receiver

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100], input[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666);
printf("Key of shared memory is %d\n",shmid);
shared_memory=shmat(shmid,NULL,0); //process attached to shared memory segment
printf("Process attached at %p\n",shared_memory);
printf("Data read from shared memory is : %s\n",(char *)shared_memory);
}
```

System Calls from File Management

Write system call is used to write to a file descriptor. In other words, write() can be used to write to any file (all hardware are also referred as file in Linux) in the system but rather than specifying the file name, you need to specify its file descriptor. (File descriptor is integer that uniquely identifies an open file of the process.)

Predefined value of fd = 0 (stdin), 1 (stdout/screen), 2 (stderr/screen).

Syntax

```
#include <unistd.h>

ssize_t write (int fd, const void *buf, size_t count)---> number of written data write (where to write, source/data, amount)
```

***Use man 2 system_call's_name (ex: man 2 write) on your terminal to get the details.

Example on write

```
#include<stdio.h>
#include<unistd.h>
int main()
{
        int count;
        count=write(1,"hello\n",6);
        printf("Total bytes written: %d\n", count); ## Fun is printf is also calling write system call internally.
}
```

Output: (Increase/Decrease the 3rd parameter and check the outputs)

```
hello
Total bytes written: 6
```

read () System call

The use of read() system call is to read from a file descriptor. The working is same as write(), the only difference is read() will read the data from file pointed to by file descriptor.

Syntax

```
#include <unistd.h>

ssize_t read (int fd, void *buf, size_t count)---> number of read data read (Read from, temp store, amount)
```

```
#include<unistd.h>
int main()
{
        int nread;
        char buff[20];
        nread=read(0,buff,10); //read 10 bytes from standard input device(keyboard), store in buffer(buff)
        write(1,buff,nread); //print 10 bytes from the buffer on the screen
}
```

open() System call: for user created file

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

int open(const char *pathname, int flags); -------> Used when the file is already created.

int open(const char *pathname, int flags, mode_t mode);----> Used when the file has to be created.

Return value:

- On success, it returns the file descriptor. The value of the file descriptor will always be a positive number (greater than 2).
- -1 is returned in the case of failure.

Example on open system call

The file is already created.

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
int main()
{
        int n, fd;
        char buff[50];
        fd=open("test.txt", O_RDONLY);        //opens test.txt in read mode and the file descriptor is saved in integer fd.
        printf("The file descriptor of the file is: %d\n, fd);  // the value of the file descriptor is printed
        n=read(fd, buff,10);          //read 10 characters from the file pointed to by file descriptor fd and save them in buffer (buff)
        write(1, buff, n);            //write on the screen from the buffer
}
```

Example on open system call

The file does not exist in the directory.

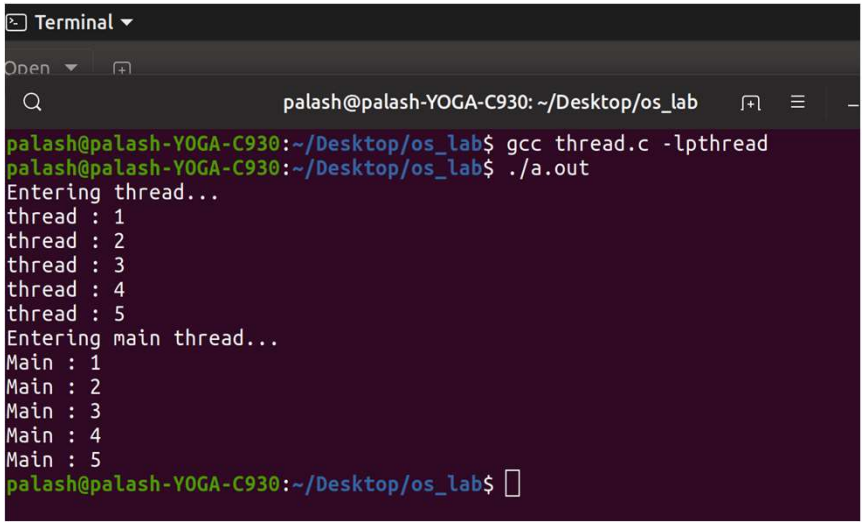
```
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
int main()
{
        int n, fd, fd1;
        char buff[50];
        fd = open("test.txt", O_RDONLY);
        n = read(fd, buff, 10);
        fd1=open("towrite.txt", O_WRONLY|O_CREAT, 0642);  //use the pipe symbol (|) to separate O_WRONLY and O_CREAT
        write(fd1, buff, n);
}
```

0 represents octal number in c. Other 3 digits are permission for user, group, and others. This permission exists after the Execution of the process.

Permission during the process's execution.

Creating a Thread ...

```
#include<stdio.h>
#include<unistd.h>
#include<pthread.h>
void *func_for_thread (void *arg);
int i,n,j;
int main () {
        pthread_t t1;
        pthread_create(&t1, NULL, func_for_thread, NULL);
        pthread_join(t1, NULL);
        printf("Entering main thread...\n");
        for (j = 0; j<5; j++){
                printf("Main : %d\n",j+1);
                sleep(1);
        }
}
void *func_for_thread (void *arg) {

        printf("Entering thread...\n");
        for (i = 0; i<5; i++){

                printf("thread : %d\n", i+1);
                sleep(1);
        }
}
```



pthread_create:

- 1st Parameter: Id of the thread
- 2nd Parameter: Attributes of the thread (NULL= default attributes by system)
  - SCOPE, PRIORITY, schedpolicy, ...ETC. (Default is sufficient).
  - An attribute object is opaque and cannot be directly modified by assignments.

  A set of functions is provided to initialize, configure, and destroy each object type.
- 3rd Parameter: Operations for the thread
- 4th Parameter: Input to the function (3rd Parameter)

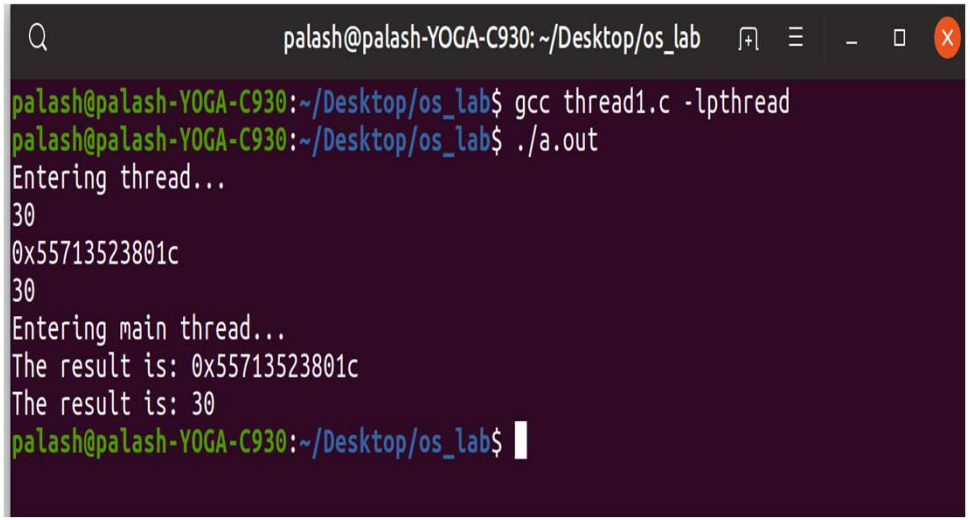
Creating a Thread ...

```
#include<stdio.h>
#include<unistd.h>
#include<pthread.h>
void *func_for_thread (void *arg);
int i,n,j;
int main () {
        pthread_t t1;
        pthread_create(&t1, NULL, func_for_thread, NULL);
        //pthread_join(t1, NULL);
        printf("Entering main thread...\n");
        for (j = 0; j<5; j++){
                printf("Main : %d\n",j+1);
                sleep(1);
        }
}
void *func_for_thread (void *arg) {

        printf("Entering thread...\n");
        for (i = 0; i<5; i++){

                printf("thread : %d\n", i+1);
                sleep(1);
        }
}
```

```
palash@palash-YOGA-C930:~/Desktop/os_lab$ gcc thread.c -lpthread
palash@palash-YOGA-C930:~/Desktop/os_lab$ ./a.out
Entering main thread...
Main : 1
Entering thread...
thread : 1
Main : 2
thread : 2
Main : 3
thread : 3
Main : 4
thread : 4
Main : 5
thread : 5
```

Passing values to a thread

```
void *func_for_thread(void *arg);
int data[2] = {6, 5};
int mul = 0;
int main() {

        pthread_t t1;
        void *result;
        pthread_create(&t1, NULL, func_for_thread, (void *)data);
        pthread_join(t1, &result);
        printf("Entering main thread...\n");
        //printf("The result is: %s\n", (char *)result);
        printf("The result is: %p\n", (int *) result);
        printf("The result is: %d\n", *((int *) result));

}

void *func_for_thread(void *arg) {

        printf("Entering thread...\n");
        int *x = arg;

        mul = x[0] * x[1];
        printf("%d\n", mul);
        printf("%p\n", &mul);
        printf("%d\n", *(&mul));
        pthread_exit(&mul);
}
```



Race Condition may Occur...

```c
#include<pthread.h>
#include<stdio.h>
#include<unistd.h>
void *fun1();
void *fun2();
int shared=1; //shared variable
int main()
 {
 pthread_t thread1, thread2;
 pthread_create(&thread1, NULL, fun1, NULL);
 pthread_create(&thread2, NULL, fun2, NULL);
 pthread_join(thread1, NULL);
 pthread_join(thread2,NULL);
 printf("Final value of shared is %d\n",shared); //prints the last updated value of shared variable
 }
 void *fun1()
 {
   int x;
   x=shared;//thread one reads value of shared variable
   printf("Thread1 reads the value of shared variable as %d\n",x);
   x++;  //thread one increments its value
   printf("Local updation by Thread1: %d\n",x);
   shared=x; //thread one updates the value of shared variable
   printf("Value of shared variable updated by Thread1 is: %d\n",shared);
 }
void *fun2()
 {
    int y;
    y=shared;//thread two reads value of shared variable
    printf("Thread2 reads the value as %d\n",y);
    y--;  //thread two increments its value
    printf("Local updation by Thread2: %d\n",y);
    shared=y; //thread one updates the value of shared variable
    printf("Value of shared variable updated by Thread2 is: %d\n",shared);
 }
```

- This guarantees that the main thread will Execute at the last.
- There is no guarantee that the thread2 will execute after the thread1.
- Interleaved execution of the threads can lead to **RACE CONDITION.**

```
palash@palash-YOGA-C930: ~/os_lab
palash@palash-YOGA-C930:~/os_lab$ gcc thread2.c -lpthread
palash@palash-YOGA-C930:~/os_lab$ ./a.out
Thread2 reads the value as 1
Local updation by Thread2: 0
Value of shared variable updated by Thread2 is: 0
Thread1 reads the value of shared variable as 1
Local updation by Thread1: 2
Value of shared variable updated by Thread1 is: 2
Final value of shared is 2
palash@palash-YOGA-C930:~/os_lab$ ./a.out
Thread1 reads the value of shared variable as 1
Local updation by Thread1: 2
Value of shared variable updated by Thread1 is: 2
Thread2 reads the value as 1
Local updation by Thread2: 0
Value of shared variable updated by Thread2 is: 0
Final value of shared is 0
palash@palash-YOGA-C930:~/os_lab$
```

Forcing Race Condition to occur

```
#include<pthread.h>
#include<stdio.h>
#include<unistd.h>
void *fun1();
void *fun2();
int shared=1; //shared variable
int main()
 {
 pthread_t thread1, thread2;
 pthread_create(&thread1, NULL, fun1, NULL);
 pthread_create(&thread2, NULL, fun2, NULL);
 pthread_join(thread1, NULL);
 pthread_join(thread2,NULL);
 printf("Final value of shared is %d\n",shared); //prints the last updated value of shared variable
 }
 void *fun1()
{
   int x;
   x=shared;//thread one reads value of shared variable
   printf("Thread1 reads the value of shared variable as %d\n",x);
   x++;  //thread one increments its value
   printf("Local updation by Thread1: %d\n",x);
   sleep(1);  //thread one is preempted by thread 2
   shared=x; //thread one updates the value of shared variable
   printf("Value of shared variable updated by Thread1 is: %d\n",shared);
}
void *fun2()
 {
    int y;
    y=shared;//thread two reads value of shared variable
    printf("Thread2 reads the value as %d\n",y);
    y--;  //thread two increments its value
    printf("Local updation by Thread2: %d\n",y);
    sleep(1); //thread two is preempted by thread 1
    shared=y; //thread one updates the value of shared variable
    printf("Value of shared variable updated by Thread2 is: %d\n",shared);
 }
```

```
palash@palash-YOGA-C930: ~/os_lab
Thread1 reads the value of shared variable as 1
Local updation by Thread1: 2
Thread2 reads the value as 1
Local updation by Thread2: 0
Value of shared variable updated by Thread1 is: 2
Value of shared variable updated by Thread2 is: 0
Final value of shared is 0
palash@palash-YOGA-C930:~/os_lab$ ./a.out
Thread1 reads the value of shared variable as 1
Local updation by Thread1: 2
Thread2 reads the value as 1
Local updation by Thread2: 0
Value of shared variable updated by Thread1 is: 2
Value of shared variable updated by Thread2 is: 0
Final value of shared is 0
palash@palash-YOGA-C930:~/os_lab$ ./a.out
Thread2 reads the value as 1
Local updation by Thread2: 0
Thread1 reads the value of shared variable as 1
Local updation by Thread1: 2
Value of shared variable updated by Thread2 is: 0
Value of shared variable updated by Thread1 is: 2
Final value of shared is 2
palash@palash-YOGA-C930:~/os_lab$
```

One solution: Binary Semaphore

```
#include<pthread.h>
#include<stdio.h>
#include<semaphore.h>
#include<unistd.h>
void *fun1();
void *fun2();
int shared=1; //shared variable
sem_t s;
int main()
{
 sem_init(&s,0,1);
 pthread_t thread1, thread2;
 pthread_create(&thread1, NULL, fun1, NULL);
 pthread_create(&thread2, NULL, fun2, NULL);
 pthread_join(thread1, NULL);
 pthread_join(thread2,NULL);
 printf("Final value of shared is %d\n",shared); //prints the last updated value of shared variable
}
```

```
void *fun1()
{
   int x;
   sem_wait(&s);
   x=shared;//thread one reads value of shared variable
   printf("Thread1 reads the value of shared variable as %d\n",x);
   x++;  //thread one increments its value
   printf("Local updation by Thread1: %d\n",x);
   sleep(1);  //thread one is preempted by thread 2
   shared=x; //thread one updates the value of shared variable
   printf("Value of shared variable updated by Thread1 is: %d\n",shared);
   sem_post(&s);
}
```

```
void *fun2()
{
   int y;
   sem_wait(&s);
   y=shared;//thread two reads value of shared variable
   printf("Thread2 reads the value as %d\n",y);
   y--;  //thread two increments its value
   printf("Local updation by Thread2: %d\n",y);
   sleep(1); //thread two is preempted by thread 1
   shared=y; //thread one updates the value of shared variable
   printf("Value of shared variable updated by Thread2 is: %d\n",shared);
   sem_post(&s);
}
```

```
Thread1 reads the value of shared variable as 1
Local updation by Thread1: 2
Value of shared variable updated by Thread1 is: 2
Thread2 reads the value as 2
Local updation by Thread2: 1
Value of shared variable updated by Thread2 is: 1
Final value of shared is 1
palash@palash-YOGA-C930:~/Desktop/os_lab$ ./a.out
Thread1 reads the value of shared variable as 1
Local updation by Thread1: 2
Value of shared variable updated by Thread1 is: 2
Thread2 reads the value as 2
Local updation by Thread2: 1
Value of shared variable updated by Thread2 is: 1
Final value of shared is 1
palash@palash-YOGA-C930:~/Desktop/os_lab$ ./a.out
Thread1 reads the value of shared variable as 1
Local updation by Thread1: 2
Value of shared variable updated by Thread1 is: 2
Thread2 reads the value as 2
Local updation by Thread2: 1
Value of shared variable updated by Thread2 is: 1
Final value of shared is 1
palash@palash-YOGA-C930:~/Desktop/os_lab$
```

```
wait(S) {
        while S <= 0; //no-operation
        S--;
}
```

```
signal(S) {
        S++;
}
```

int sem_init(sem_t *sem, int pshared, unsigned int value);

- Address of semaphore variable
- 0 – shared between threads; Non Zero – shared between processes
- Initial value

Another Solution: Mutex

```
#include<pthread.h>
#include<stdio.h>
#include<unistd.h>
void *fun1();
void *fun2();
 int shared=1; //shared variable
 pthread_mutex_t l; //mutex lock
 int main()
 {
 pthread_mutex_init(&l, NULL); //initializing mutex locks (address, attribute. NULL = Default)
 pthread_t thread1, thread2;
 pthread_create(&thread1, NULL, fun1, NULL);
 pthread_create(&thread2, NULL, fun2, NULL);
 pthread_join(thread1, NULL);
 pthread_join(thread2,NULL);
 printf("Final value of shared is %d\n",shared); //prints the last updated value of shared variable
 }
     void *fun1()
      {
         int x;
         printf("Thread1 trying to acquire lock\n");
         pthread_mutex_lock(&l); //thread one acquires the lock. Now thread 2 will not be able to acquire the lock
     //until it is unlocked by thread 1
        printf("Thread1 acquired lock\n");
         x=shared;//thread one reads value of shared variable
         printf("Thread1 reads the value of shared variable as %d\n",x);
         x++;  //thread one increments its value
         printf("Local updation by Thread1: %d\n",x);
         sleep(1);  //thread one is preempted by thread 2
         shared=x; //thread one updates the value of shared variable
         printf("Value of shared variable updated by Thread1 is: %d\n",shared);
         pthread_mutex_unlock(&l);
         printf("Thread1 released the lock\n");
      }
```

```
void *fun2()
 {
    int y;
    printf("Thread2 trying to acquire lock\n");
    pthread_mutex_lock(&l);
    printf("Thread2 acquired lock\n");
    y=shared;//thread two reads value of shared variable
    printf("Thread2 reads the value as %d\n",y);
    y--;  //thread two increments its value
    printf("Local updation by Thread2: %d\n",y);
    sleep(1); //thread two is preempted by thread 1
    shared=y; //thread one updates the value of shared variable
    printf("Value of shared variable updated by Thread2 is: %d\n",shared);
    pthread_mutex_unlock(&l);
    printf("Thread2 released the lock\n");
 }
```

```
Thread1 trying to acquire lock
Thread1 acquired lock
Thread1 reads the value of shared variable as 1
Local updation by Thread1: 2
Thread2 trying to acquire lock
Value of shared variable updated by Thread1 is: 2
Thread1 released the lock
Thread2 acquired lock
Thread2 reads the value as 2
Local updation by Thread2: 1
Value of shared variable updated by Thread2 is: 1
Thread2 released the lock
Final value of shared is 1


...Program finished with exit code 0
Press ENTER to exit console.
```