Date - 14/04/2021

Experiment No. 2

Familiarization of System Calls used for Operating system and Network Programming in Linux

Aim **:** *To familiarize and understand the use and functioning of System Calls used for System and Network Programming in Linux.*

Theory

System Call

*In computing, a **system call** is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. A system call is a way for programs to **interact with the operating system**. A computer program makes a system call when it makes a request to the operating system's kernel. System call **provides** the services of the operating system to the user programs via Application Program Interface(API). It provides an interface between a process and operating system to allow user-level processes to request services of the operating system. System calls are the only entry points into the kernel system. All programs needing resources must use system calls.*

Following are situations which need system calls in OS:

- *Reading and writing from files demand system calls.*
- *If a file system wants to create or delete files, system calls are required.*
- *System calls are used for the creation and management of new processes.*
- *Network connections need system calls for sending and receiving packets.*
- *Access to hardware devices like scanner, printer, need a system call.*

Services Provided by System Calls :

1. Process creation and management *2.Main memory management*

3. Device handling(I/O) *4.Protection* *5.Networking*

6. File Access, Directory and File system management, etc.

Types of System Calls : *There are 5 different categories of system calls –*

1. ***Process control:*** *end, abort, create, terminate, allocate and free memory.*
2. ***File management:*** *create, open, close, delete, read file etc.*
3. *Device management*
4. *Information maintenance*
5. *Communication*

Examples of Windows and Unix System Calls –

	Windows	***Unix***

Process Control	*CreateProcess()*
	ExitProcess()
	WaitForSingleObject()
		fork()
		exit()
		wait()

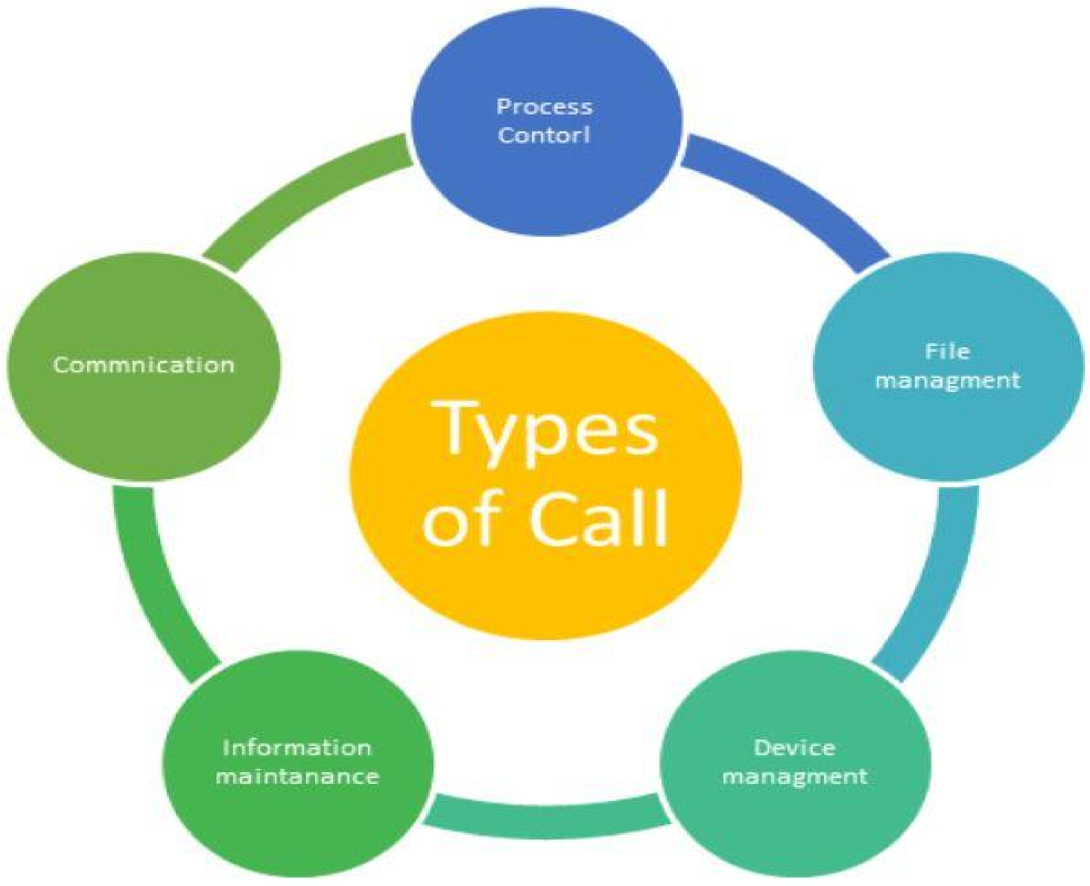
File Manipulation	*CreateFile()*
	ReadFile()
	WriteFile()
	CloseHandle()
		open()
		read()
		write()
		close()

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()

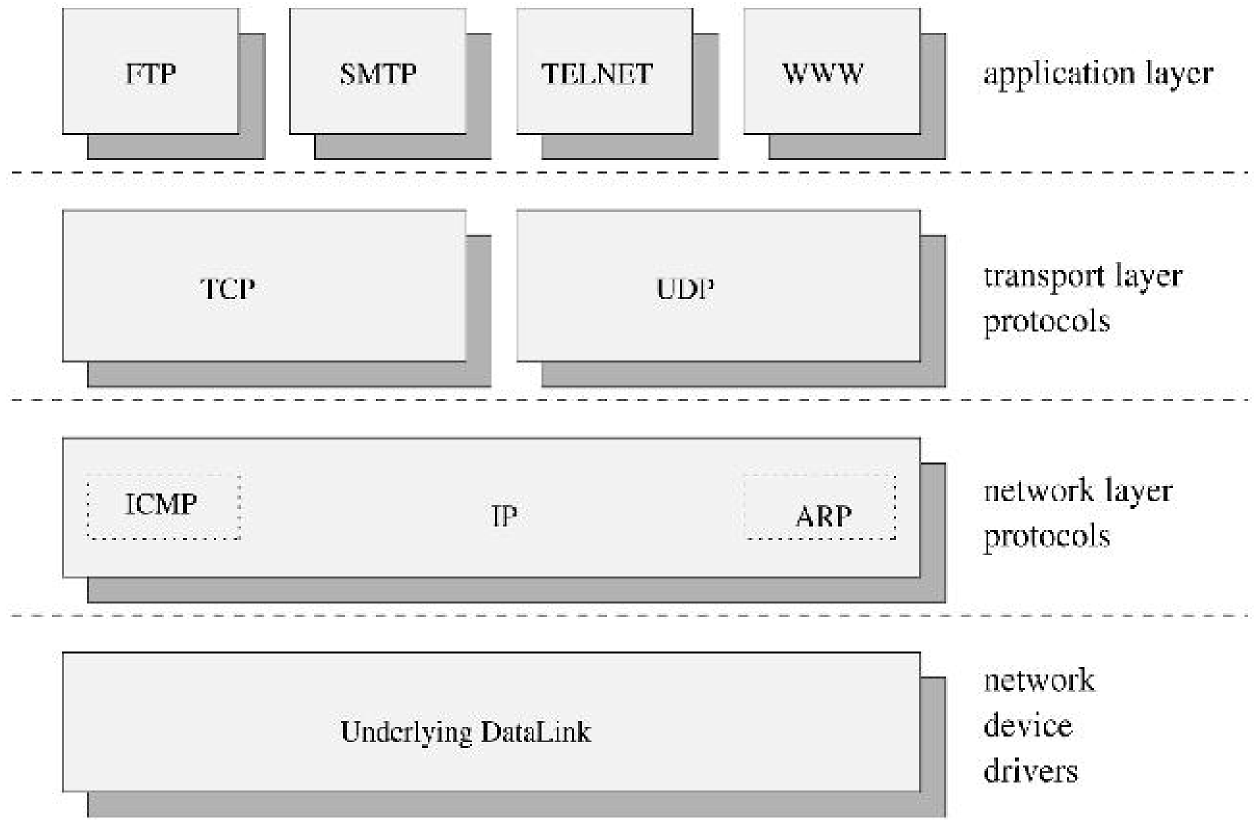


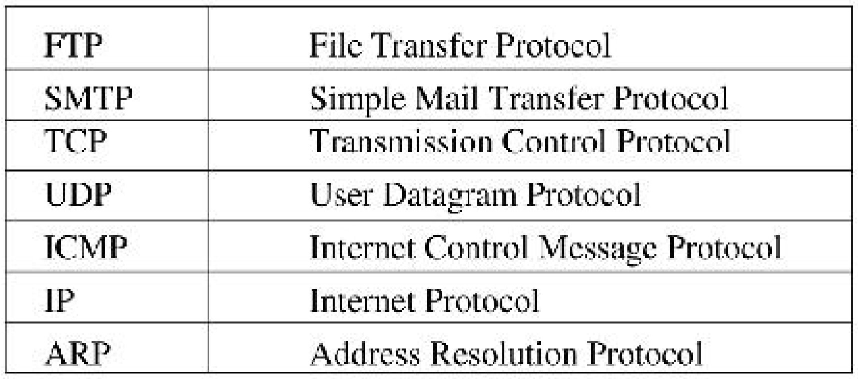
Network Programming in Linux

Like most other Unix-based operating systems, Linux supports TCP/IP as its native network transport. In this series, we will assume you are fairly familiar with C programming on Linux and with Linux topics such as signals, forking, etc.

Brief Introduction to TCP/IP

The TCP/IP suite of protocols allows two applications, running on either the same or separate computers connected by a network, to communicate. It was specifically designed to tolerate an unreliable network. TCP/IP allows two basic modes of operation—connection-oriented, reliable transmission and connectionless, unreliable transmission (TCP and UDP respectively). Figure 1 illustrates the distinct protocol layers in the TCP/IP suite stack.





FTP	File Transfer Protocol
SMTP	Simple Mail Transfer Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
ICMP	Internet Control Message Protocol
IP	Internet Protocol
ARP	Address Resolution Protocol

TCP/IP Protocol Layers

TCP provides sequenced, reliable, bi-directional, connection-based bytestreams with transparent retransmission. In English, TCP breaks your messages up into chunks (not greater in size than 64KB) and ensures that all the chunks get to the destination without error and in the correct order. Being connection-based, a virtual connection has to be set up between one network entity and the other before they can communicate. UDP provides (very fast) connectionless, unreliable transfer of messages (of a fixed maximum length).

To allow applications to communicate with each other, either on the same machine (using loopback) or across different hosts, each application must be individually addressable.

TCP/IP addresses consist of two parts—an IP address to identify the machine and a port number to identify particular applications running on that machine.

The addresses are normally given in either the “dotted-quad” notation (i.e., **127.0.0.1**) or as a host name (**foobar.bundy.org**). The system can use either the /etc/hosts file or the *Domain Name Service* (DNS) (if available) to translate host names to host addresses.

Port numbers range from 1 upwards. Ports between 1 and IPPORT_RESERVED (defined in /usr/include/netinet/in.h—typically 1024) are reserved for system use (i.e., you must be root to create a server to bind to these ports).

The simplest network applications follow the *client-server* model. A server process waits for a client process to connect to it. When the connection is established, the server performs some task on behalf of the client and then usually the connection is broken.

Using the BSD Socket Interface

The most popular method of TCP/IP programming is to use the *BSD socket interface*. With this, network endpoints (IP address and port number) are represented as *sockets*.

The socket interprocess communication (IPC) facilities (introduced with 4.2BSD) were designed to allow network-based applications to be constructed independently of the underlying communication facilities.

Creating a Server Application

To create a server application using the BSD interface, you must follow these steps:

1. Create a new socket by typing: **socket()**.
2. *bind* an address (IP address and port number) to the socket by typing: **bind**. This step identifies the server so that the client knows where to go.
3. *listen* for new connection requests on the socket by typing: **listen()**.
4. *accept* new connections by typing: **accept()**.

Often, the servicing of a request on behalf of a client may take a considerable length of time. It would be more efficient in such a case to accept and deal with new connections while a request is being processed. The most common way of doing this is for the server to *fork* a new copy of itself after accepting the new connection.

***RESULT* :**

The usage and functioning of system calls in an operating system and network programming in linux has been studied and familiarized successfully.