(A) INTRODUCTION

UNIX is a multiuser, multitasking operating system that is widely used as the master control program in workstations and especially servers. UNIX was developed in 1969 by Ken Thompson at AT&T Lab. The name was coined for a single-user version (Uno) of "multIX".

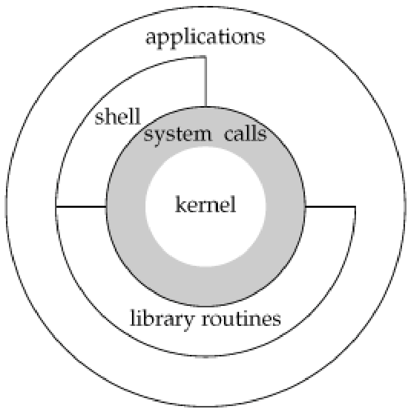
All operating systems provide services for programs they run. Typical services are,

- Executing a program.
- Opening a file.
- Reading a file.
- Allocating a region of memory.
- Getting the current time of day.

(B) OVERVIEW OF UNIX OS - UNIX Architecture

An **Operating System** can be defined as the software that controls the hardware resources of the computer and provides an environment under which programs can run. This software can also be called as **kernel**, since it is relatively small and resides at the core of the environment.

The following figure shows a diagram of the UNIX system architecture.



The interface to the kernel is a layer of software called the **system calls**. Library routines are built on top of the system call interface, but applications are free to use both.

The shell is a special application that provides an interface for running other applications.

An operating system is the kernel and all the other software such as system utilities, applications, shells and library routines that makes a computer useful and gives the computer its personality.

(i) **LOGGING IN**

Login Name:

When a user logs into a UNIX system, the user has to enter login name and password. The system then looks up our login name in its password file, usually it is in /etc/passwd.

The password file is composed of **seven colon-separated fields:**

- Login name [abc]
- Encrypted password [x]
- Numeric user ID [205]
- Numeric group ID [105]
- Comment field [Richard Stevens]
- Home directory [/home/abc]
- Shell program [/bin/ksh]

Eg:

abc:x:205:105:Richard Stevens:/home/abc:/bin/ksh

Shells:

A shell is a command-line interpreter that reads user input and executes commands. The user input to a shell is normally from the terminal **(an interactive shell)** or sometimes from a file **(called a shell script)**.

Common shells used on UNIX systems are as follows,

Name	**Path**
Bourne shell	/bin/sh
Bourne-again shell	/bin/bash
C shell	/bin/csh
Korn shell	/bin/ksh
TENEX C shell	/bin/tcsh

(i) The **Bourne shell**, developed by Steve Bourne at Bell Labs.

(ii) The **C shell**, developed by Bill Joy at Berkeley, is provided with all the BSD releases.

(iii) The **Korn shell** , developed by David Korn at Bell Labs. The Korn shell is considered a successor to the Bourne shell and was first provided with SVR4(System V Release 4).

(iv) The **Bourne-again shell** is the GNU shell provided with SVR4. Linux uses the Bourne-again shell for its default shell.

(v) The **TENEX C** shell is an enhanced version of the C shell.

(ii) FILES AND DIRECTORIES

File System:

The UNIX file system is a hierarchical arrangement of directories and files. Everything starts in the directory called **root** whose name is the single character.

Directory:

A directory is a file that contains directory entries. Each directory entry contains a file name along with a structure of information describing the attributes of the file.

The some of the attributes of a file are,

- Type of file – regular file
- Directory **-** the size of the file
- The owner of the file
- Permissions for the file.

Filename:

The names in a directory are called **filenames.** The only two characters that cannot appear in a filename are the slash character (/) and the null character.

Two filenames are automatically created whenever a new directory is created:

(i) . [called dot] – **Current Directory**
(ii) ..[called dot-dot] – **Parent Directory**

Pathname:

A sequence of one or more filenames separated by slashes and optionally starting with a slash, forms a pathname.

A pathname that begins with a slash is called an **absolute pathname**.

Relative pathnames refer to files relative to the current directory.

Working Directory:

Every process has a working directory, also called the current working directory from which all relative paths are interpreted.

A process can change its working directory with the **chdir** function.

Home Directory:

When the user logs in, the working directory is set to the user's home directory. This user's home directory is obtained from the user's entry in the password file.

(iii) INPUT AND OUTPUT

File Descriptors:

File descriptors are normally **small non-negative integers** that the kernel uses to identify the files being accessed by a particular process. Whenever the user opens an existing file or creates a new file, the kernel returns a file descriptor that the user uses to read the file or write into the file.

All shells open **three descriptors** namely,

(i) Standard input
(ii) Standard output
(iii) Standard error.

The 'ls' command provide away to redirect any or all of these three descriptors to any file.

Unbuffered I/O:

Unbuffered I/O is provided by the functions open, read, write, lseek and close. These functions all work with file descriptors.

Standard I/O:

The standard I/O functions provide a buffered interface to the unbuffered I/O functions. The fgets() reads an entire line whereas the read() reads a specified number of bytes.

(iv) PROGRAMS AND PROCESSES:

Program:

A program is an executable file residing on disk in a directory. A program is read into memory and is executed by the kernel.

Process:

An executing instance of a program. Some operating systems use the term task to refer to a program that is being executed.

Every process has a unique numeric identifier called the **process ID**. The process ID is always a non-negative integer.

Process Control:

The UNIX system provides three **primary functions** for process control. This includes,

(i) the creation of new processes – fork()
(ii) Program execution – exec()
(iii) Process termination – waitpid()

(v) USER IDENTIFICATION:

User ID

The user ID entry in the password file is a numeric value that identifies the user to the system. This user ID is assigned by the system administrator when the login name is assigned, and the user cannot change it. The userID is normally assigned to be unique for every user.

Eg:

userID 0 is referred as **root** or the **supervisor**.

Group ID

The entry in the password file also specifies the numeric group ID which is assigned by the system administrator when the login name is assigned. The password file contains multiple entries that specify the same group ID. The group file that maps group names into numeric group ID's is /etc/group.

(vi) SIGNALS:

Signals are a technique used to notify a process that some condition has occurred. Every signal has a name. These names are all defined by positive integer constants in the header<signal.h>

The signal process has three choices.

- Ignore the signal.
- Catch the signal.
- Let the default action occur.

(vii) TIME VALUES:

UNIX systems have maintained **two** different time values:

- **Calendar time**.

This value counts the number of seconds since January 1, 1970, Coordinated Universal Time(UTC). These time values are used to record the time when a file was last modified. The primitive system data type **time-t** holds these time values.

- **Process time**:

It measures the central processor resources used by a process. Process time is measured in clock ticks. Process time is also called CPU time. When we measure the execution time of a process, unix system maintains the following three values for a process.

They are,

- Clock time.
- User CPU time.
- System CPU time.

Clock time: The clock time is the amount of time the process takes to run, and its value depends on the number of other processes being run on the system.

User CPU time: The user CPU time is the CPU time attributed to user instructions.

System CPU time: The System CPU time is the CPU time attributed to the kernel when it executes on behalf of the process.

The sum of user CPU time and system CPU time is often called the **CPU time.** The **time(1) command** is used to measure the clock time, user time and system time of any process.

(C) ENVIRONMENT OF A UNIX PROCESS

(i) **Process Environment :**

main Function:

A C program starts execution with a function called main. The prototype for the main function is

```
int  main (int  argc, char  *argv [ ]);
```

Here, argc -> is the number of command-line arguments.
argv -> is an array of pointers to the arguments.

(ii)Process termination:

To terminate a process.

There are eight ways for a process to terminate. These eight ways are divided under two categories. They are

(a) Normal termination.
(b) Abnormal termination.

Normal termination occurs in five ways. They are,

1. Return from main
2. Calling exit
3. Calling_exit or _Exit
4. Return of the last thread from its start routine
5. Calling pthread_exit from the last thread

Abnormal termination occurs in three ways. They are,

6. Calling abort
7. Receipt of signal
8. Response of the last thread to a cancellation request

Return from main:

If the start-up routine were coded in ‘C’ the call to main could look like,

```
exit(main(argc, argv));
```

Returning an integer value from the main function is equivalent to calling exit with same value. Thus,

exit(0); is the same as
return(0); from the main function.

Exit Functions:

The following three functions

(a) _exit
(b) _Exit and
(c) exit terminate a program normally.

The _exit() and _Exit(), returns to the kernel immediately but exit(), performs certain cleanup processing and then returns to the kernel.

The **prototype** is as follows,

```
#include<stdlib.h>
void  exit(int  status);
void _Exit(int  status);
```

```
#include<unistd.h>
 void  _exit(int  status);
```

atexit function:

A process can register up to 32 functions that are automatically called by exit. These are called exit handlers and are registered by calling the atexit function.

Syntax:

```
#include <stdlib.h>
int  atexit (void (*func)(void));
```

Here, the user pass the address of a function as the argument to atexit .When this function is called it, is not passed any arguments and is not expected to return a value.

(D) PROCESS CONTROL

The process control provided by the UNIX system includes,

- The creation of new processes
- Program execution
- Program termination

Process Identifiers:

Every process has a unique process ID which is a non-negative integer.

There are some special processes available.

Example:

(i) Process ID 0 is usually the scheduler process and is often known as the swapper.

(ii) Process ID 1 is usually the init process and is invoked by the kernel at the end of the bootstrap procedure.

(iii) Process Id 2 is the pagedaemon responsible for supporting the paging of the virtual memory system.

The following functions return these identifiers.

```
#include<unistd.h>
pid_t  getpid(void)          Returns: process  Id of calling process
pid_t  getppid(void)         Returns: parent process ID of calling process
uid_t  getuid(void)          Returns : real  user  ID of calling process
uid_t  geteuid(void)         Returns: effective  user  ID of calling  process
gid_t  getgid(void)          Returns: real group ID of calling  process
gid_t  getegid(void)         Returns: effective  group  ID  of calling  process
```

fork function

An existing process can create a new one by calling the fork function.

Prototype:

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

Returns: 0 in child, process ID of child in parent,-1 on error.

The new process created by fork is called the child process. The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent, is the process ID of the new child. The reason the child process ID is returned to the parent is that a process can have more than one child. The reason fork return 0 to the child is that a process can have only a single parent.

The child can always call getppid to obtain the process ID of its parent.

vfork function :

The vfork function is intended to create a new process when the purpose of the new process is to exec a new program.

The vfork function creates the new process, just like fork, without copying the address space of the parent into the child; the child simply calls exec right after the vfork.

Vfork guarantees that the child runs first, until the child calls exec of exit.

exit functions:

Exit functions are used to terminate the process.

A process can terminate normally in five ways:

Normal termination occurs in five ways. They are,

1. Return from main
2. Calling exit
3. Calling_exit or _Exit
4. Return of the last thread from its start routine
5. Calling pthread_exit from the last thread

The three forms of abnormal termination are as follows:

1. Calling abort. It generates the SIGABRT signal.
2. When the process receives certain signals. The signal can be generated by the process itself, by some other process, or by the kernel.
3. The last thread responds to a cancellation request.

wait and waitpid functions:

A process that calls wait or waitpid can

1 Block, if all of its children are still running.

2 Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched.

3 Return immediately with an error, if it doesn't have any child processes.

The prototype for wait and waitpid are as follows,

```
#include<sys/wait.h>
pid_t  wait(int  *status);
pid_t waitpid(pid-t   pid,  int *status, int options);
```

The waitpid function returns the process ID of the child that terminated and stores the child's termination status in the memory location pointed by statloc.

Both return: process ID if OK, 0 or -1 error

The differences between wait and waitid functions are as follows,

1 The wait function can block the caller until a child process terminates, whereas waitpid has an option that prevents it from blocking.
2 The waitpid function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.

waitid function

The waitid function is similar to waitpid, but provides extra flexibility.

The **prototype** is,

#include<sys/wait.h>

int waitid(idtype_t idtype, id-t id, siginfo-t *infop, int options);

Returns;0 if OK, -1 on error

waitid allows a process to specify which children to wait for.

The types supported by waitid are summarized as below

Constant	Description
P-PID	Wait for a particular process.
P-PGID	Wait for any child process in a particular group
P_ALL	Wait for any child process

exec Functions:

The use of the fork function is to create a new process (the child) that then causes another program to be executed by calling one of the exec functions. When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing its main function. The process ID does not change across an exec, because a new process is not created, exec merely replaces the current process.

With fork, we can create new processes; and with the exec functions, we can initiate new programs.

There are six different exec functions as listed below.

#include<unistd.h>

int execl (const char * pathname, const char *arg0, …. /* (char *) 0 */);

int execv (const char * pathname, char * const argv[]);

```
int execle (const  char  * pathname,  const  char  * arg0, …. /* (char *) 0, char * const
                        envp[] */ );

int execve (const  char  * pathname, char  * const  argv[],  char  * const  envp[] );

int execlp (const  char  * filename,  const  char  *arg0, …. /* (char *) 0 */);

int execvp (const  char  * filename,  char  * const  argv[] );
                              All six return: -1 on error, no returns on success
```

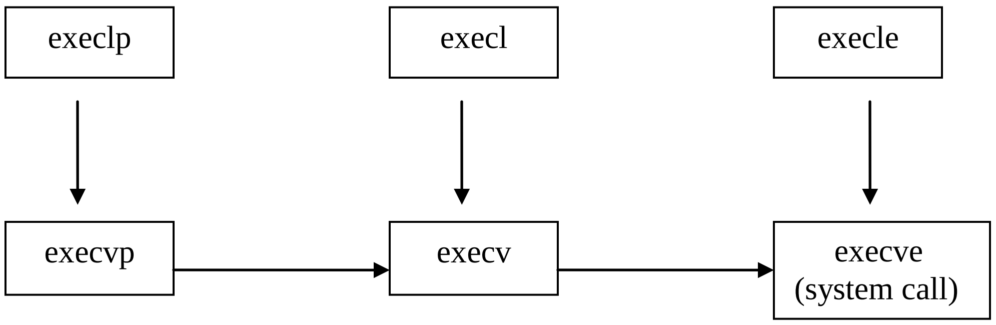
Differences between these six functions:

The first difference in these functions is that the first four take a pathname argument, whereas the last two take a filename argument.

The second difference concerns the passing of the argument list (l stands for list and v stands for vector). The functions execl, execlp, and execle require each of the command-line arguments to the new program to be specified as separate arguments. We mark the end of the arguments with a null pointer. For the other three functions (execv, execvp, and execve), we have to build an array of pointers to the arguments, and the address of this array is the argument to these three functions.

The third difference is the passing of the environment list to the new program. The two functions whose names end in an e (execle and execve) allow us to pass a pointer to an array of pointers to the environment strings.

The relationship among these six functions as shown below.



(E) PROCESS RELATIONSHIPS

(i)Terminal Logins:

In early UNIX systems, Users logged in using dumb terminals that were connected to the host with hard wired connections.

The terminals were either

(i) local(directly connected) or

(ii) Remote (connected through a modem).

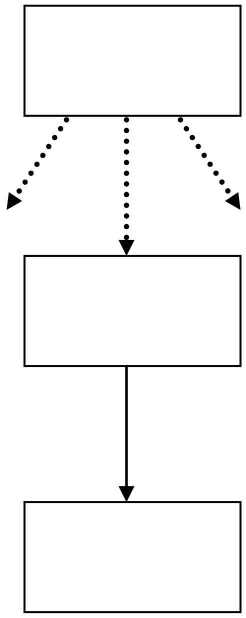
In either case, these logins came through a terminal device driver in the kernel.

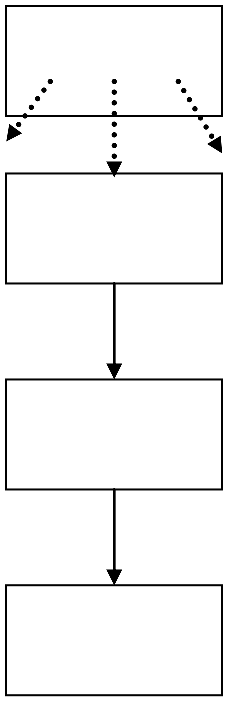
A host had a fixed number of these terminal devices. Applications were developed to create "terminal windows" to emulate character based terminal, allowing user to interact with host in familiar way (i.e, via the shell command line).

Some of the terminal logins are

BSD Terminal Logins

The system administrator creates a file, usually /etc/ttys that has one line per terminal device. Each line specifies the name of the device and other parameters are passed to the getty program. When the system is bootstrapped, the kernel creates process ID1, the init process, and it is init that brings the system up multi-user. The init process reads the file /etc/ttys and, for every terminal device that allows a login. Next the fork command is executed, followed by an exec of the program getty.





MAC OS X Terminal Logins

On Mac OS X, the terminal login process follows the same steps as in the BSD login process but with a graphical-based login screen from the start.

LINUX Terminal Logins

The Linux login procedure is very similar to the BSD procedure. The main difference between the BSD login procedure and the linux login procedure in the way the terminal configuration is specified. On Linux, /etc/inittab contains the configuration information specifying the terminal devices for which init should start a getty process. Depending on the version of getty in use, the terminal characteristics are specified either on the command line or in the file /etc/gettydefs.

SOLARIS Terminal Logins

Solaris supports two forms of terminal logins:

- getty style, and
- ttymon logins.

Normally, getty is used for the console, and ttymon is used for other terminal login. The ttymon command is part of a larger facility termed SAF (Service Access Facility).The goal of the SAF was to provide a consistency way to administer services that provide access to the system. Init is the parent of SAC (Service Access Controller) which does a fork and exec of the ttymon program when the system enters multiuser state. The ttymon program monitors all the terminal ports listed in its configuration file and does a fork when the user entered his login name. This child of ttymon does an exec of login and login prompts us for our password.

(ii) PROCESS GROUPS:

Each process also belongs to a process group.

A process group is collection of one or more processes, usually associated with the same job. Each process group has a unique process groupID. Process group IDs are positive integers and can be stored in a pid_t data type.

getpgrp(): This function returns the process group ID of the calling process.

Syntax:

```
#include<unistd.h>
pid_t getpgrp(void);
        returns: process group ID of calling process.
```

Each process group can have a process group leader. It is possible for a process group leader to create a process group, create processes in the group, and then terminate. The process group still exists, as long as atleast one process is in the group, regardless of whether the group leader terminates. This is called the process group life time—the period of time that begins when the group is created and ends when the last remaining process leaves the group.

setpgid(): A process joins an existing process group or creates a new process group by calling setpgid.

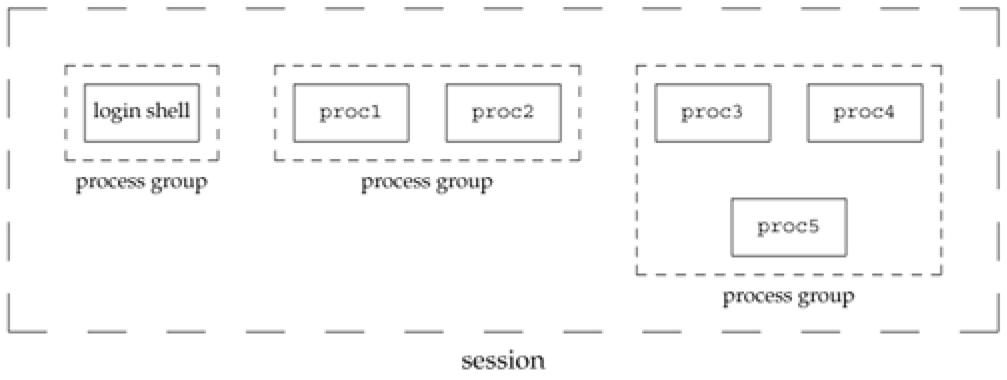
Syntax:

```
#include<unistd.h>
int setpgid(pid_t pid, pid_t pgid);
```

This function sets the process group ID to pgid in the process whose process ID equals pid. If the two arguments are equal, the process specified by pid becomes a process group leader.If pid is 0, the process ID of the caller is used. Also if pgid is 0, the process ID specified by pid is used as the process group ID.

(iii) SESSIONS:

A session is a collection of one or more process groups. The figure represents three process groups in a single session.



The processes in a group are usually placed there by a shell pipeline.

Eg: proc1| proc2 &
proc3| proc4| proc5

setsid():

A process establishes a new session by calling the setsid function.

Syntax:

```
#include<unistd.h>
  pid_t  setsid(void);
            Returns: process group ID if ok, -1 on error.
```

If the calling process is not a process group leader, this function creates a new session.

Three things happen,

- The process becomes the session leader of this new session.
- The process becomes the process group leader of a new process group.
- The process has no controlling terminal.

getsid():

The getsid function returns the process group ID of the calling process's session leader.

Syntax:

```
#include<unistd.h>
  pid_t  getsid(pid_t pid);
            returns: session leader's process group ID if ok, -1 on error.
```

(F) SIGNALS

Signals are a technique used to notify a process that some condition has occurred. Signals are software interrupts. Most nontrivial application programs need to deal with signals. Signals provide a way of handling asynchronous events.

<u>Eg:</u>

A user at a terminal typing the interrupt key to stop a program or the next program in a pipeline terminating prematurely.

SIGNAL CONCEPTS:

1. Every signal has a name. These names all begin with the three characters 'SIG'.

Eg:

- SIGABRT (abort the signal that is generated when the process calls the abort function).
- SIGALRM (the alarm signal that is generated when the timer set by the alarm function goes off).

2. Each version has different types of signals.
   - Version 7 has 15 different signals.
   - SVR4 & 4.4BSD both have 31 different signals.
   - FreeBSD5.2.1, Mac OS X10.3 & Linux 2.4.22 support 31 signals.
   - Solaris 9 support 38 different signals.

3. All the signals are present in header file called <signal.h>
   Numerous conditions can generate a signal.
   - The terminal-generated signals:
     It occurs when users press certain terminal keys.
     E.g.: Pressing the DELETE key on the terminal normally causes the interrupt signal (SIGINT) to be generated.
   - Hardware exceptions generate signals:
     E.g. divide by 0, invalid memory reference. These conditions are usually detected by the hardware and the kernel is notified.
     The SIGSEGV is generated for a process that execute an invalid memory reference
   - The kill (2) function:
     It allows a process to send any signal to another process or process group. To send this function, the senders have to be the owner of the process or the sender to be the superuser.
   - The kill (1) command:
     It allows the user to send signals to other processes. This command is often used to terminate a runaway background process.
   - Software conditions:
     They can generate signals when something happens about which the process should be notified.

   Examples:

   (i) SIGURG (generated when out-of-band data arrives over a network connection)

   (ii) SIGALRM (generated when an alarm clock set by the process expires)

Signals are classic examples of asynchronous events.

The process has three choices for dealing with the signal.

1. **Ignore the signal.**
   Tell kernel to ignore the signal, while process is executing**.** This works for most signals, but two signals can never be ignored: SIGKILL & SIGSTOP. If we ignore some of the signals that are generated by a hardware exception (such as illegal memory reference or divide by 0), the behavior of the process is undefined.
2. **Catch the signal.**
   Provide a function that is called when the signal occurs.
3. **Let the default action apply.**
   Every signal has a default action. But the default action for most signals is to terminate the process

signal function:

The simplest interface to the signal features of the **UNIX** System is the signal function.

Syntax:

```
#include <signal.h>
void (*signal (int signo, void (*func) (int)))(int);
```

The signo argument represents the name of the signal.

The value of func is:

a) the constant SIG_IGN.
b) the constant SIG_DFL.
c) the address of a function to be called when the signal occurs.

If we specify SIG_IGN → we are telling the system to ignore the signal.

When we specify SIG_DFL → we are setting the action associated with the signal to its default value.

When we specify the address of a function to called when the signal occurs. we are arranging to "catch" the signal. We call the function either the signal handler or the signal-catching function.

The signal function's first argument, signo, is an integer. The second argument is a pointer to a function that takes a single integer argument and returns nothing.

The return value from signal is the pointer to the previous signal handler.

Unreliable Signals:

In earlier versions of the UNIX System, signals were unreliable.

- That is signals could get lost: a signal could occur and the process would never know about it.
- Also a process had little control over a signal: a process could catch the signal or ignore it.
- Sometimes, we would like to tell the kernel to block a signal: don't ignore it, just remember if it occurs, and tell us later when we're ready.

kill and raise functions:

The kill function sends a signal to process or a group of processes.

The raise function allows a process to send a signal to itself.

Syntax:

```
#include <signal.h>
 int  kill  (pid_t   pid ,int signo ) ;
 int raise  (int signo);
```

Both return: 0 if OK, -1 on error.

There are four different conditions for the pid argument to kill.

pid > 0 This signal is sent to the processes whose process ID is pid.

pid == 0 The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send signal. .

pid < 0 The signal is sent to all processes whose process group ID equals the absolute value of pid and for which the sender has permission to send the signal.

pid ==-1 The signal is sent to all processes on the system for which the sender has permission to send the signal.

A process needs permission to send a signal to another process. The super user can send a signal to any process .

POSIX.1 defines signal number 0 as the null signal. If the signo argument is 0, then the normal error checking is performed by kill, but no signal is sent. This is often used to determine if a specific process still exists.

alarm functions:

The alarm function allows the user to set a timer that will expire at a specified time in the future. When the timer expires, the SIGALRM signal is generated .If we ignore or don’t catch this signal; its default action is to terminate the process.

Syntax:

```
# include < unistd.h >
unsigned int alarm (unsigned int seconds);
```

Returns: 0 or number of seconds until previously set alarm.

The seconds value is the number of clock seconds in the future when the signal should be generated.

Although the default action for SIGALRM is to terminate the process, most processes that use an alarm clock catch this signal.

pause functions:

The pause function suspends the calling process until a signal is caught.

Syntax:

```
# include < unistd.h >
int pause (void );
```

Returns :-1 with errno set to EINTR.

The only time pause returns is if a signal handler is executed and that handler returns. In that case, pause returns -1 with errno set to EINTR.

abort Function:

The abort function causes abnormal program termination.

Prototype:

```
# include < stdlib.h>
void abort (void);
```

This function never returns. This function sends the SIGABRT signal to the caller.

Calling abort will deliver an unsuccessful termination notification to the host environment by calling raise (SIGABRT).

Sleep function:

This function causes the calling process to be suspended until either

1. The amount of wall clock time specified by seconds has elapsed.
2. A signal is caught by the process and the signal handler returns.

In case 1, the return value is 0, when sleep returns early.

Because of some signal being caught (case 2), the return value is the number of unslept seconds.

Prototype,

```
# include <unistd.h>
unsigned int sleep (unsigned int seconds);
                    Returns: 0 or number of unslept seconds.
```

(G) INTERPROCESS COMMUNICATION (IPC):

1. Process to communicate with each other.
2. It is mechanisms which allow arbitrary processes to exchange do to and synchronize execution.
3. To exchange information is by passing open files across a fork or an exec or through file system.

Examples for IPC:

1. Pipes.
2. FIFOs.
3. Message Queues.
4. Shared Memory
5. Semaphores.

(i)Message passing (SVR4) - PIPES:

- A pipe is a one-way mechanism that allows two related processes (i.e. one is an ancestor of the other) to send a byte stream from one of them to the other one.
- Pipe is used to allow transfer of data between processes in a FIFO manner.
- Two kinds of pipes:
  1. Named pipes.
  2. Unnamed pipes.
- Pipes are the oldest form of UNIX system IPC.
- Pipes have two limitations.
  a) They have been half-duplex (i.e., data flows in only one direction).

b) Pipes can be used only between processes that have a common ancestor.

- FIFO gets around the second limitation.
- UNIX domain sockets and named STREAMS-based pipes get around both limitations.

pipe():

Pipe is created by calling the pipe function which provides a one-way flow of data. This system call takes as an argument an array of 2 integers that will be used to save the two file descriptors used to access the pipe. The first to read from the pipe, and the second to write to the pipe.

Syntax:

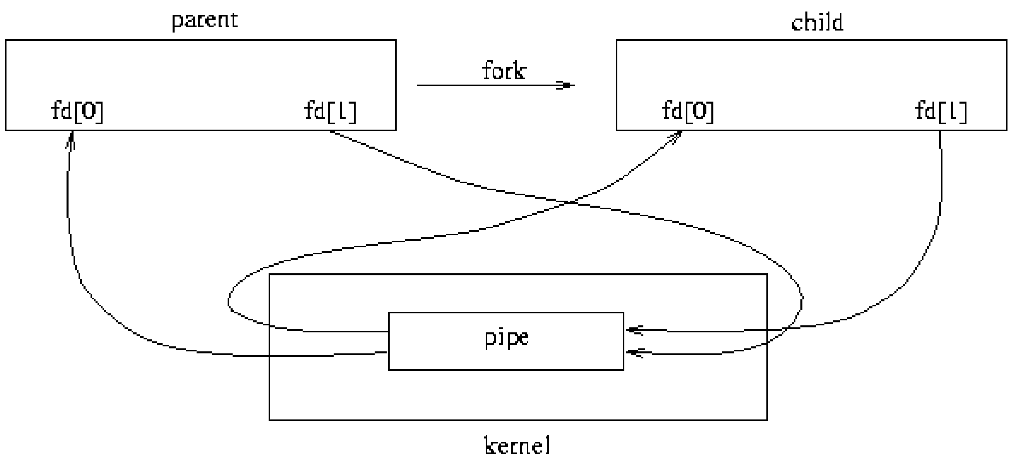
```
#include < unistd .h>
  int pipe ( int filedes[2]);
                                    returns:0 if OK, -1 on error.
```

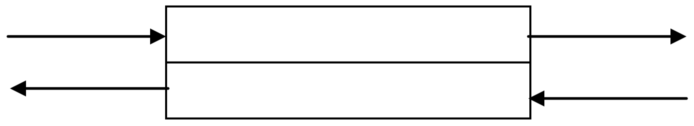
Two file descriptors are returned through the filedes argument:
filedes[0] is open for reading, and
filedes[1] is open for writing.
The output of filedes[1] is the input for filedes[0].

Disadvantage of pipe:

One limitation of anonymous pipes is that only processes 'related' to the process that created the pipe (i.e. siblings of that process.) may communicate using them.

The following figure depicts Half Duplex and Full Duplex Pipes.





(ii) Message Queues:

A message queue is a queue onto which messages can be placed. A message is composed of a message type (which is a number), and message data.

Messages queues are identified by a message queue identifier. For every message queue in the system, the kernel maintains the following structure.

```
struct msgid_ds
{
  struct ipc_prem    msg_perm;  // read, write permissions
  msgqnum_t          msg_qnum;   // no. of messages on queue
  msglen_t           msg_qbytes; // no. bytes on the queue
  pid_t              msg_lspid;    // pid of last msgsnd()
  pid_t              msg_lrpid;    // pid of last msgrcv()
  time_t             msg_stime     // last-msgsnd() time
  time _t            msg_rtime       // last-msgrcv() time
  time_t             msg_ctime       // last-change time
.
.
};
```

(a) msgget function:

A new message queue is created or an existing message queue is opened with the msgget function. This system call accepts two parameters - a queue key, and flags.

The **prototype** is as follows,

```
#include<sys/msg.h>
int msgget(key_t key, int flag);
                returns: message queue ID if OK, -1 on error.
```

The first parameter key may be one of:

IPC_PRIVATE - used to create a private message queue.

a positive integer - used to create (or access) a publicly-accessible message queue.

The second parameter contains flags that control how the system call is to be processed. It may contain flags like IPC_CREAT or IPC_EXCL.

When a new message queue is created, the following members of the msqid_ds structure are initialized.

- The ipc_perm structure
- msg_qnum, msg_lspid, msg_lrpid, msg_stime and msg_rtime are all set to 0.
- msg_ctime is set to the current time.
- msg_qbytes is set the system limit.

(b) msgsnd function:

Data(message) is placed onto a message queue by calling msgsnd. This system call copies the user message structure and places that as the last message on the queue.

The **prototype** is as follows:

```
#include<sys/msg.h>
int msgsnd(int msqid, const void *ptr, size_t nbytes,int flags);
                Returns: 0 if OK,-1 on error.
```

Parameters:

1. `int msqid` - id of message queue, as returned from the `msgget()` call.
2. *ptr - message frame
3. size_t nbytes - the size of the data part of the message, in bytes.
4. `int flags` - either 0 or IPC_NOWAIT

(c) msgrcv function:

A Message is read from a message queue using msgrcv function.

The **prototype** is as follows:

```
#include<sys/msg.h>
  ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag)
      Returns: size of data portion of message if OK, -1 on error.
```

Parameters:

1. int msqid - id of the queue, as returned from msgget().
2. void *ptr – specifies where the received message is to be stored.
3. size_t nbytes - size of the data portion of the buffer pointed to by a ptr
4. long type - Type of message we wish to read. may be one of:
    - 0 - The first message on the queue will be returned.

  - a positive integer - the first message on the queue whose type (mtype) equals this integer (unless a certain flag is set in msgflg, see below).
  - a negative integer - the first message on the queue whose type is less than or equal to the absolute value of this integer.

5. int flag - a logical 'or' combination of any of the following flags:
   - IPC_NOWAIT - if there is no message on the queue matching what we want to read, return '-1', and set errno to ENOMSG..
   - MSG_NOERROR - If a message with a text part larger than 'msgsz' matches what we want to read, then truncate the text when copying the message to our msgbuf structure. If this flag is not set and the message text is too large, the system call returns '-1', and errno is set to E2BIG.

When msgrcv succeeds, the kernel updates the msqid_ds structure associated with the message queue.

(d)msgctl function:

The msgctl function performs various operations on a queue.

```
#include<sys/msg.h>
  int msgctl(int msqid, int cmd, struct msqid_ds *buf);
                        Returns: 0 if OK,-1 on error
```

The cmd argument specifies the command to be performed on the queue specified by msqid.

- IPC_STAT:
  Fetch the msqid_ds structure for this queue, storing it in the structure pointed to by buf.
- IPC_SET:
  Copy the following fields from structure pointed to buf to the msqid_ds structure associated with this queue: msg_perm.uid, msg_perm.gid, msg_perm.mode and the msg_qbytes.
- IPC_RMID:
  Remove the message queue from the system and any data still on the queue.

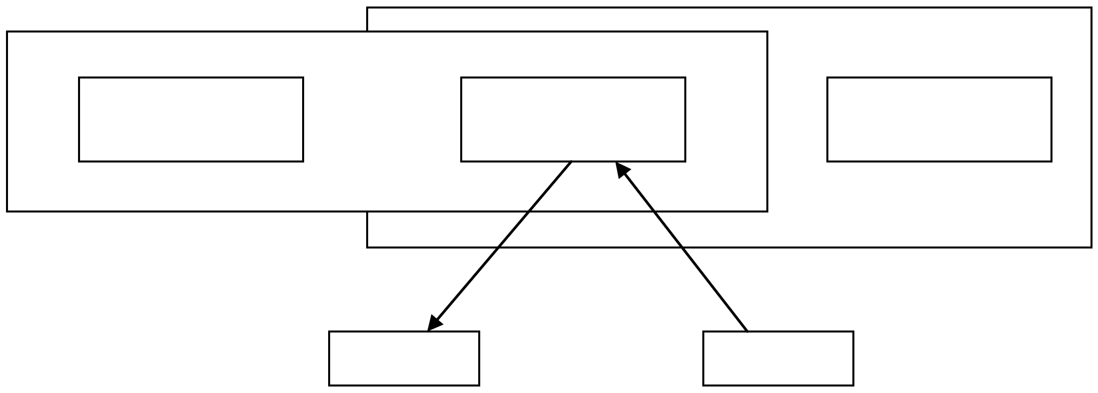
(iii) Shared Memory:

Shared memory allows two or more processes to share a given region of memory. With shared memory, we declare a given section in the memory as one that will be used simultaneously by several processes.

For every shared memory segment, the kernel maintains the following structure.

```
struct shmid_ds
{
  struct ipc_prem    shm_perm;  // read, write permissions
  size_t             shm_segsz; // size of segment in bytes
  shmatt_t      shm_nattch;   // no. of current attaches
  pid_t         shm_lpid;    // pid of last shmop()
  pid_t         shm_cpid;    // pid of creator()
  time_t        shm_atime     // last-attach time
```

```
    time _t        shm_dtime      // last-detach time
    time_t         shm_ctime      // last-change time
.
.    };
```



(a) shmget()

A shared memory segment first needs to be allocated (created), using the `hmget()` system call. This function returns shared memory identifier.

The **prototype** is as follows,

```
#include<sys/shm.h>
    int shmget(key_t key, size_t size, int flag);
            returns: shared memory ID if OK, -1 on error.
```

This call gets a key for the segment (like the keys used in `msgget()` and `semget()`), the desired segment size, and flags to denote access permissions and whether to create this page if it does not exist yet.

When a new segment is created, the following members of the shmid_ds structure are initialized.

- The ipc_perm structure
- shm_lpid, shm_nattach, shm_atime and shm_dtime are all set to 0.
- shm_ctime is set to the current time.
- shm_segsz is set the size requested.

(b)shmctl()

The shmctl function performs various shared memory operations.

The **prototype** is as follows,

```
#include<sys/shm.h>
    int shmctl(int shmid, int cmd, struct shmid_ds *buf);
         returns: 0 if OK,-1 on error
```

The cmd argument specifies one of the following five commands to be performed on the segment specified by shmid.

- IPC_STAT: Fetch the shmid_ds structure for this segment, storing it in the structure pointed to by buf.
- IPC_SET: Copy the following fields from structure pointed to buf to the shmid_ds structure associated with this shared memory segment: shm_perm.uid, shm_perm.gid and msg_perm.mode.
- IPC_RMID: Remove the shared memory segment from the system.
- SHM_LOCK: Lock the shared memory segment in memory.
- SHM_UNLOCK : Unlock the shared memory segment in memory.

(c)shmat( )

After the user allocated a memory segment, the user need to add it to the address space of the process. This is done using the `shmat()` (shared-memory attach) system call.

Syntax:

```
        #include<sys/shm.h>
                void *shmat (int shmid, const void *addr, int flag);
```

The value returned by shmat is the address at which the segment is attached, or -1 if an error is occurred. If shmat is successful then the kernel will increment the shm_nattch.

(d) shmdt( )

To destroy the shared memory segment. (i.e.) detach it.

Syntax:

```
  #include<sys/shm.h>
       int shmdt ( void *addr);
                  returns: 0 if OK, -1 on error.
```

(iv) Semaphores

A semaphore is a counter used to provide access to a shared data object for multiple processes. (i.e.) a semaphore is a resource that contains an integer value, and allows processes to synchronize by testing and setting this value in a single atomic operation

To obtain the shared resource, a process needs to do the following:

- Test the semaphore that controls the resource.
- If the value of the semaphore is positive, the process can use the resource. Here, the process decrements the semaphore value by 1, which means it has used one unit of the resource.
- If the value of the semaphore is 0, the process goes to sleep.

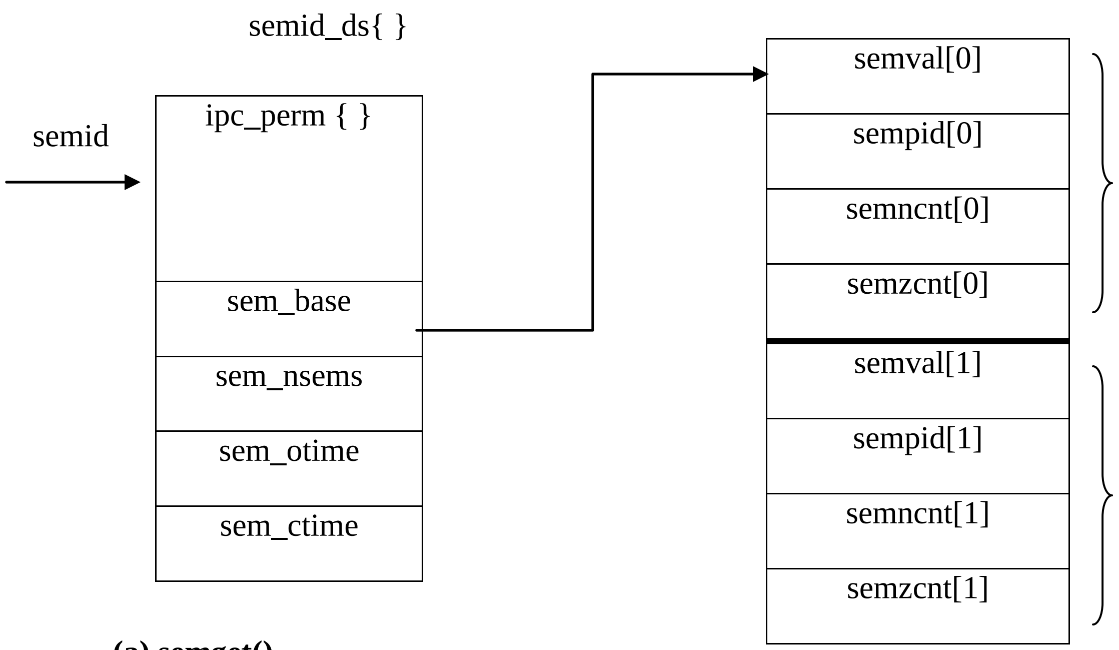
A binary semaphore is one which controls a single resource, and its value is initialized to one. In general, however, a semaphore can be initialized to any positive value, with the value indicating how many units of the shared resource are available for sharing.

A semaphore set is a structure that stores a group of semaphores together, and possibly allows the process to commit a transaction on part or all of the semaphores in the set together. The kernel maintains a semid_ds structure for each semaphore set:

```
struct semid_ds{
        struct ipc_perm sem_perm;     /* defines the permission and owner */
        unsigned short sem_nsems;    /* no. of semaphores in set */
        time_t sem_otime;            /* last-semop() time */
        time_t sem_ctime;            /* last-change time */
.
.
};
```

Each semaphore is represented by an anonymous structure containing at least the following members:

```
struct{
        unsigned short semval;  /* semaphore value */
        pid_t sempid;           /* pid for last operation*/
        unsigned short semncnt; /* no. of processes awaiting semval>curval */
        unsigned short semzcnt; /* no. of processes awaiting semval == 0 */
                .
                .
        };
```



(a) semget()

This function is used to get a semaphore ID.

Syntax:

```
#include<sys/sem.h>
int semget(key_t key, int nsems, int flags);
        Returns:semaphore ID if OK, -1 on error.
```

When a new set is created, the following members of the semid_ds structure are initialized.

- The ipc_perm structure is initialized.
- sem_otime is set to 0.
- sem_ctime is set to the current time.
- sem_nsems is set to nsems. [nsems refers the number of semaphores in the set.]

(b) semctl()

This function is the catchall for various semaphore operations. One operation is to initialize the value of the semaphores in the set.

Syntax:

```
  #include<sys/sem.h>
   int semctl(int semid, int semnum, int cmd, ….. /*union semun arg */);
      union semun
      {
            int val;
            struct semid_ds *buf;
            unsigned short *array;
      };
```

The cmd argument specifies one of the following **ten** commands to be performed on the set specified by semid.

IPC_STAT : Obtains status information about the semaphore set.
IPC_SET : Change certain attributes of the semaphore set. For example, set the sem_perm.uid, sem_perm.gid …
IPC_RMID : Remove the semaphore set from the system and any data still on the queue.
GETVAL : Return the value of semaphore value for the member semnum. (i.e) To fetch a specific semaphore value.
SETVAL : To set a specific semaphore value.
GETPID : Returns the number of processes waiting for a notify on a specific semaphore within the set.
GETZCNT : Waiting for zero condition.
GETALL : Fetch all the semaphore values in the set.
SETALL : Set all the semaphore values in the set to the values.

(c) semop function:

The function **semop** automatically performs an array of operations on a semaphore set.

The **prototype** is,

```
#include<sys/sem.h>
int semop( int semid,struct sembuf semoparray[],size_t nops);
                    Returns : 0 if OK,-1 on error.
```

nops: this argument specifies the number of operations(elements) in the array**.**

Semoparray []:

Each element in this array specifies an operation for one particular semaphore value in the set. The semoparray argument is a pointer to an array of semaphore operations, represented by sembuf structures:

```
struct sembuf
{
        unsigned short  sem_num;
        short           sem_op;
        short           sem_flg;
};
```

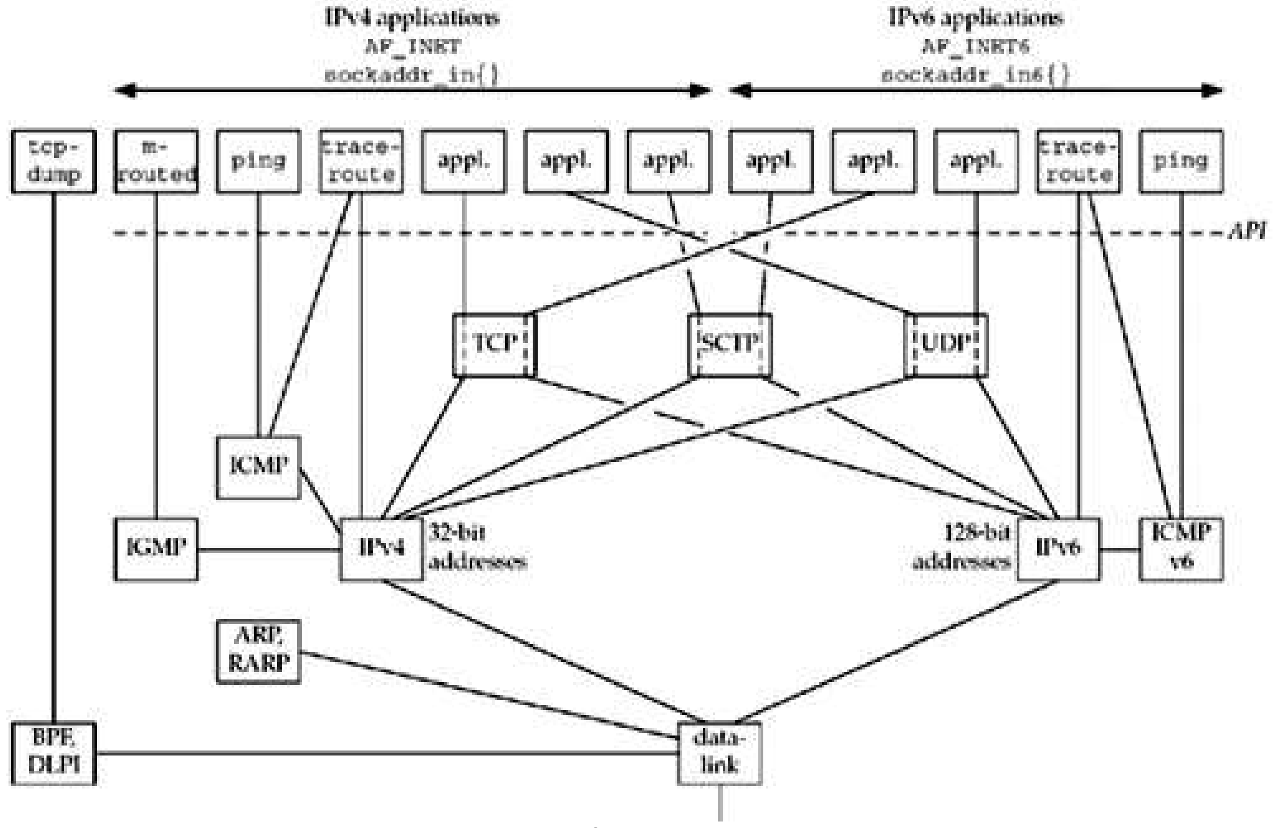
The operation on each member of the set is specified by the corresponding sem_op value. This value can be positive, negative, or 0.

1. If sem_op is positive, means releasing or returning of resources by the process. The value of sem_op is added to the semaphore's value.
2. If sem_op is negative, the caller wants to wait until, i.e.) the caller want to obtain resources that the semaphore controls.
3. If sem_op is 0, this means that the calling process wants to wait until the semaphore's value becomes 0.
   If the semaphore's value is currently 0, the function returns immediately.
   If the semaphore's value is nonzero, the following conditions apply.

   a. If **IPC_NOWAIT** is specified, return is made with an error of EAGAIN.
   b. If **IPC_NOWAIT** is not specified, the **semzcnt** value for this semaphore is incremented, and the calling process is suspended until one of the following occurs.

      i. The semaphore's value becomes 0. The value of semzcnt for this semaphore is decremented.
      ii. The semaphore is removed from the system. In this case, the function returns an error of EIDRM.
      iii. A signal is caught by the process, and the signal handler returns.

(H) OVERVIEW OF TCP/IP PROTOCOLS



IPv4 Internet Protocol version 4. IPv4, often denoted as IP, has been the workhorse protocol of the IP suite since the early 1980s. It uses 32-bit addresses. IPv4 provides packet delivery service for TCP, UDP, SCTP, ICMP and IGMP.

IPv6 Internet Protocol version 6. IPv6 was designed in the mid-1990s as a replacement for IPv4. The major change is a larger address comprising 128 bits, to deal with the explosive growth of the Internet in the 1990s. IPv6 provides packet delivery service for TCP, UDP, SCTP, and ICMPv6.

TCP Transmission Control Protocol. TCP is a connection-oriented protocol that provides a reliable, full-duplex byte stream to its users. TCP sockets are an example of stream sockets. TCP takes care of details such as acknowledgments, timeouts, retransmissions, and the like. Most Internet application programs use TCP.

UDP User Datagram Protocol. UDP is a connectionless protocol, and UDP sockets are an example of datagram sockets. There is no guarantee that UDP datagrams ever reach their intended destination. As with TCP, UDP can use either IPv4 or IPv6.

SCTP Stream Control Transmission Protocol. SCTP is a connection-oriented protocol that provides a reliable full-duplex association. The word "association" is used when referring to a connection in SCTP because SCTP is multihomed, involving a set of IP addresses and a single port for each side of an association.

	SCTP provides a message service, which maintains record boundaries. As with TCP and UDP, SCTP can use either IPv4 or IPv6, but it can also use both IPv4 and IPv6 simultaneously on the same association.
ICMP	Internet Control Message Protocol. ICMP handles error and control information between routers and hosts. These messages are normally generated by and processed by the TCP/IP networking software itself, not user processes.
IGMP	Internet Group Management Protocol. IGMP is used with multicasting, which is optional with IPv4.
ARP	Address Resolution Protocol. ARP maps an IPv4 address into a hardware address (such as an Ethernet address). ARP is normally used on broadcast networks such as Ethernet, token ring, and FDDI, and is not needed on point-to-point networks.
RARP	Reverse Address Resolution Protocol. RARP maps a hardware address into an IPv4 address. It is sometimes used when a diskless node is booting.
ICMPv6	Internet Control Message Protocol version 6. ICMPv6 combines the functionality of ICMPv4, IGMP, and ARP.
BPF	BSD packet filter. This interface provides access to the datalink layer. It is normally found on Berkeley-derived kernels.
DLPI	Datalink provider interface. This interface also provides access to the datalink layer. It is normally provided with SVR4.

Each Internet protocol is defined by one or more documents called a Request for Comments (RFC), which are their formal specifications.

Characteristics of TCP

- TCP also provides reliability.
- TCP contains algorithms to estimate the round-trip time (RTT) between a client and server dynamically so that it knows how long to wait for an acknowledgment.
- TCP also sequences the data by associating a sequence number with every byte that it sends.
- TCP provides flow control. TCP always tells its peer exactly how many bytes of data it is willing to accept from the peer at any one time. This is called the advertised window.
- TCP connection is full-duplex. This means that an application can send and receive data in both directions on a given connection at any time.

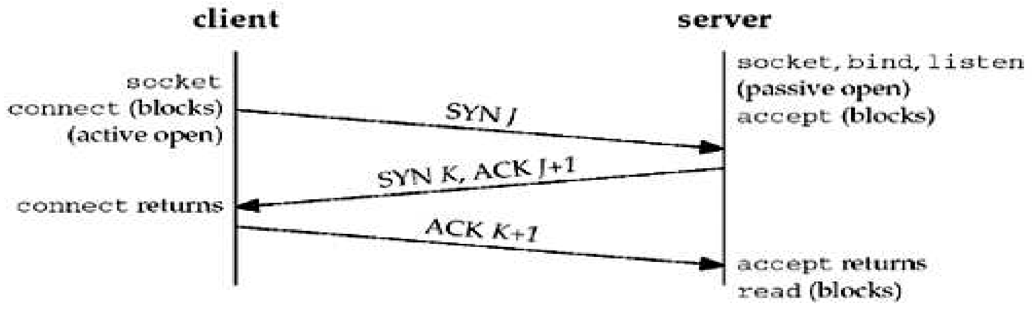
TCP Connection Establishment and Termination

Three-Way Handshake

The following scenario occurs when a TCP connection is established:

1. The server must be prepared to accept an incoming connection. This is normally done by calling `socket`, `bind`, and `listen` and is called a passive open.
2. The client issues an active open by calling `connect`. This causes the client TCP to send a "synchronize" (SYN) segment, which tells the server the client's initial sequence number for the data that the client will send on the connection. Normally, there is no data sent with the SYN; it just contains an IP header, a TCP header, and possible TCP options.
3. The server must acknowledge (ACK) the client's SYN and the server must also send its own SYN containing the initial sequence number for the data that the server will send on the connection. The server sends its SYN and the ACK of the client's SYN in a single segment.
4. The client must acknowledge the server's SYN.

   The minimum number of packets required for this exchange is three; hence, this is called TCP's three-way handshake.



TCP Three-way Handshake

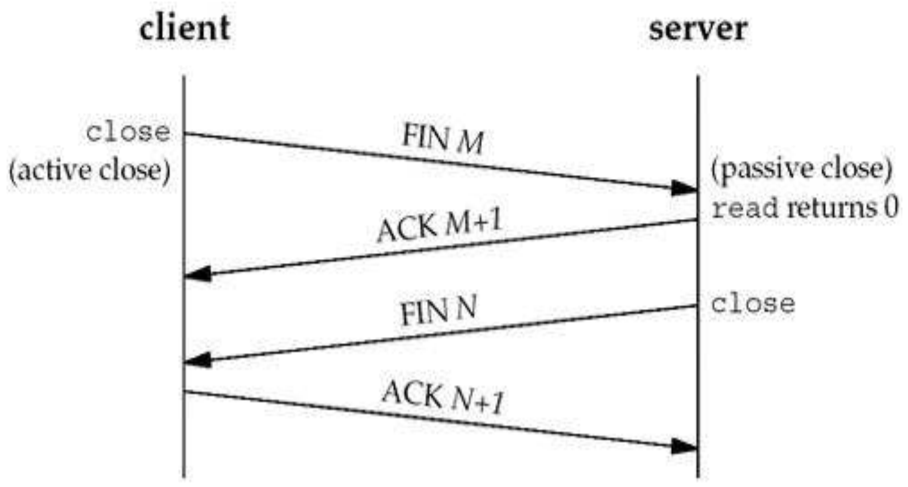
TCP Connection Termination

While it takes three segments to establish a connection, it takes four to terminate a connection.

1. One application calls `close` first, and we say that this end performs the active close. This end's TCP sends a FIN segment, which means it is finished sending data.
2. The other end that receives the FIN performs the passive close. The received FIN is acknowledged by TCP. The receipt of the FIN is also passed to the application as an end-of-file (after any data that may have already been queued for the application to receive), since the receipt of the FIN means the application will not receive any additional data on the connection.

3. Sometime later, the application that received the end-of-file will `close` its socket. This causes its TCP to send a FIN.
4. The TCP on the system that receives this final FIN (the end that did the active close) acknowledges the FIN.

Since a FIN and an ACK are required in each direction, four segments are normally required. We use the qualifier "normally" because in some scenarios, the FIN in Step 1 is sent with data. Also, the segments in Steps 2 and 3 are both from the end performing the passive close and could be combined into one segment.



Packets exchanged when a TCP connection is closed