1. Introduction

What is an Operating System?

Operating system (OS) is a software layer between the computer hardware and the application programs. OS provides the following facilities to the running programs.

- Operating system allows multiple programs to run simultaneously by allocating small time slices of CPU time for each of the programs. OS also allocates a separate virtual memory space for each program. Each programming running in this virtual memory space thinks that it is the only program running by using entire memory address space.

- Operating system provides lot of services (System calls) to the application programs. Application programs can use OS services by calling the system calls.

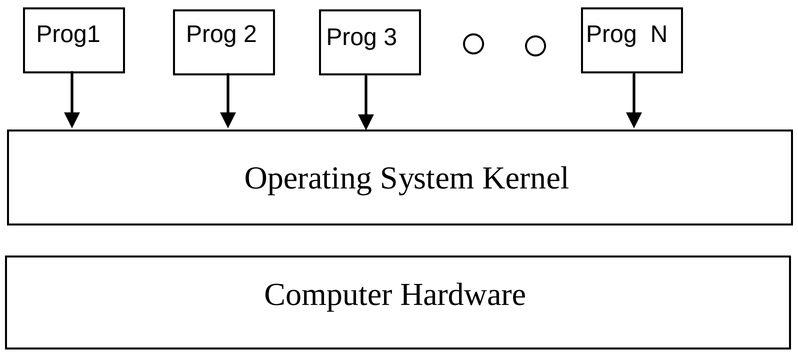
Structure of an Operating System

The core or most important part of an operating system is called Kernel. The OS, besides the kernel, consists of so many utility programs, command line interpreters (shells) and graphical user interface (X-windows). All these things put together we call an operating system. Kernels of operating systems can be implemented in two possible ways. These are:

- Monolithic Kernel
- Micro Kernel

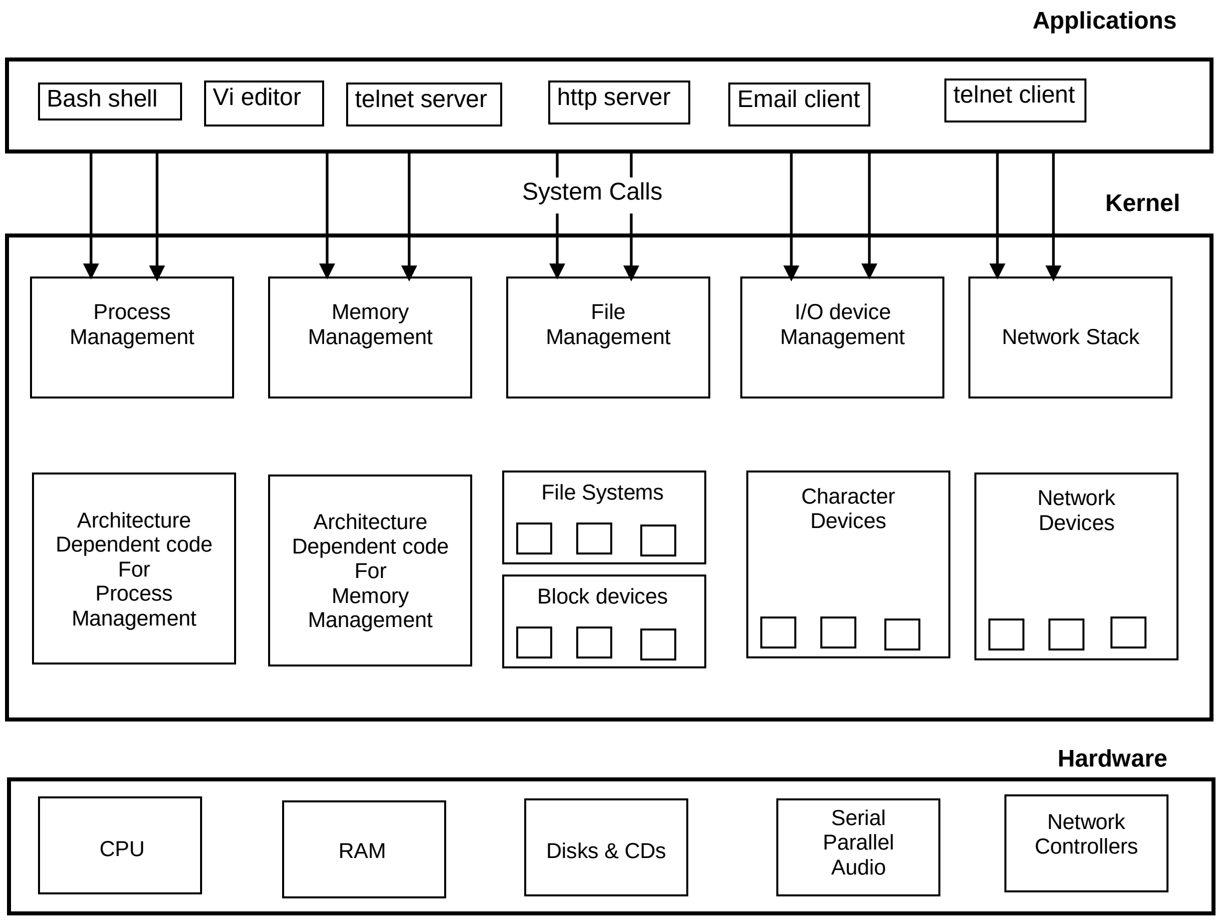
In the monolithic kernel entire kernel functionality is organized into a single block or entity. Linux kernel is an example of a monolithic kernel.

In the micro kernel based approach, kernel will be a very minimal one with process management, memory management and minimal IPC. All other parts like file systems, I/O systems and networking are implemented as separate processes or tasks.



The following diagram shows the three important layers of a computer. These are:

- Applications
- Kernel
- Hardware



Hardware

The hardware of a computer contains the following four important components:

- CPU or Microprocessor
- Memory (Non-volatile memory such as Flash and Volatile memory such as DRAM)
- I/O controllers (VGA controller, keyboard/serial controllers, IDE/Floppy controllers etc..)
- I/O devices (Display monitor, Keyboard, Mouse, Hard disk driver, Floppy driver etc..)

Kernel

Kernel is the most important component of an OS and gets loaded into the memory during boot time. Kernel initializes all the hardware components (CPU, memory, I/O controllers) and takes charge of them. Kernel allows various applications to run simultaneously by assigning the hardware resources (CPU time, memory and I/O controllers or I/O devices) to them. Following are the important components or modules of a kernel:

- Process Management
- Memory Management
- File System Management
- I/O System Management

- Inter process communication (IPC)
- Network protocol stack
- Device Drivers

Applications

The primary purpose of Kernel is to provide an environment to run various application programs simultaneously or concurrently. Kernel also provides lot of system calls for the programs. Software developers can use these system calls of kernel to develop useful application programs easily.

Programmer's view of an Operating System

As a programmer, we view the operating system as a provider of set of services. We can develop application programs by using these services (system calls) of operating system. These services help the programmer to build complex and useful applications easily. Following are the typical services provided by an operating system:

Service	Facilities offered by this service system calls
File services	Creating new files and directories, opening existing files, reading and writing into the files, closing the files
I/O services	Opening the devices, reading and writing to the devices
Memory services	Allocating memory dynamically and freeing the memory
Multiprocessing services	Creating new processes, executing new programs in the processes, waiting for the processes to complete
Multithreading services	Creating and controlling multiple threads within a process
Signal and IPC services	Controlling the processes, synchronizing the processes, mutual exclusion between processes, communication between processes
Network services	Communication between processes running on different computers connected through network
Time services	Finding time and date, alarm services, timeout services

System Calls and Library functions

Before proceeding for the study of Linux system calls and library functions, It is important to distinguish between system calls and library functions. For a programmer both system calls and library functions will appear similar. There is no difference as for as usage is concerned. But only difference is that, system calls are implemented as a part of the kernel. So whenever a system call is invoked, the program (which invoked system call) will switch from user mode to kernel mode and executes the system call and returns to the user mode once system call execution is completed. Library functions will execute completely in user mode.

But there are some functions, which will act as library functions, but occasionally calls system calls themselves. Best example for this kind of function is *malloc()*. Most of the time *malloc*() acts as a library function and allocates memory without invoking kernel call. This is because *malloc*() uses system call *sbrk*() to allocate large memory from kernel at time, and services user requests from this memory, without calling system call. Only when all the memory allocated from kernel exhausts, it calls kernel system call to allocate more memory again. Same is the case with standard I/O functions like *printf*() and *fwrite*(). Whenever user program call these functions these may put the given data in the buffer and calls the system call only when new line is received or buffer is completely filled.

GNU/Linux Operating System

Linux kernel was developed by Linus Torvalds at the University of Helsinki, with the help of Unix programmers from across the Internet. But important point to note here is that, Linux is only a kernel not a complete operating system. GNU, which is an acronym for “GNU Not Unix” is an open source movement started by Richard Stallman to develop all the software programs used in Unix as open source programs. So that any one can see the source code of these programs and can modify them.

GNU software (software development tools, shells, editors, GUI) along with Linux kernel makes Linux a full-fledged operating system just like Unix. Linux kernel similar to GNU software is released under open source license. So any one can go through the source code of entire Linux kernel and can understand it.

Open source movement is a real boon to the students, any one who got interest to learn these complex software (compilers, kernels, etc..) can go through the source code and learn.

Finally Linux is a Unix like operating system developed from scratch and made open to the public to use, understand and enhance. Linux like all other flavors of Unix follows POSIX (Portable Operating System Interface) standard. So all POSIX compliant applications will run on all flavors of Unix including Linux.

Users, Groups, Files & Processes

The Users, Groups, Files and Processes are four fundamental objects for understanding or programming Linux. Linux allows only users who have account in the system. Linux maintains the following information for each user in the /etc/passwd file.

user name	: Name of the user
user Id	: Every user of linux system is identified by unique integer Id
password	: Password in encrypted form, now a days stored in shadow file, for security
group Id	: To which group this user will belong
home directory	: Home directory of the user
shell	: Default program to run once login is successful, typically default shell

In Linux we can create various user Groups. A user can be made to belong to multiple user Groups. This user group concept allows to give common access permissions to a group of users.

Files in Linux are organized in a tree (more precisely inverted tree) structure. Top most directory is called root directory and it is represented by '/' character. Actual files may present in different devices like hard disks, SCSI disks, floppy disks, CD-ROMs. But all files present in different devices will appear under a single tree structure. This is possible because, when we mount a new device, we can specify the directory name on which this new device to be mounted. Then all the files in that device will appear under that directory. For example typically we will mount floppy disk under the directory "/mnt/floppy". Once floppy disk is mounted, the /mnt/floppy directory will contain all the files of floppy disk. This is in clear contrast to DOS or Windows in which is drive is represented with a letter (A: C: D: etc..).

All the contents of a file are stored in blocks (sectors) on block devices like floppy, hard disk and CD-ROM. But information about the file like its owner, access rights, etc.. will be maintained on a separate block called inode. So every file is associated with one inode. All these inodes are also stored on the block device. Following is the main information that will be maintained about a file in the inode structure.

- File Device Number

- Inode number
- File type and Access rights
- Owner Id
- Group Id
- File Size
- Times indicating last access, last modification and creation
- Data block numbers that contain file data
- Single, Double and Triple Indirection data block numbers

Each program under execution is called a process. There may be lot of executable programs present on the hard disk. But only when these programs are loaded into memory for execution, these will become processes. The life of a process could be from fraction of a second to as long as system is powered-on and running. For example if you give `"ls"` command, it causes ls program to run as a process and completes within a second. But if you take server programs like mail server or web server, these programs will start on power-on of machine and continue to run till machine is switched-off.

At any time there will be tens of processes running in Linux machine. You can see all the processes running in a machine at that time by giving `"ps -ax"` command. Kernel maintains lot of information about a process. This information about a process will be maintained in a kernel structure called "Process Control Block". This will be present only in the memory.

Every process is associated with a User Id and User Group ID. The access permissions for this process depends on its User Id and Group Id

2. File and I/O Services

Operating system provides various facilities or services to the application programs. One can build good and useful application programs by using these services of the OS. One of the very useful services of OS is file and I/O service. File service allows one to create, open, read and write, files and directories on block devices like hard disk and floppy disk. I/O service allows developer to read and write to I/O devices attached to serial ports, parallel ports, USB ports etc.. Most of the applications require file and I/O services.

The file management module of Linux kernel provides these services. Main job of file management is to map files and directories to sectors or blocks of a block device. File management module also links the calls to I/O devices to device drivers.

2.1 Linux File and I/O System Calls

Linux provides uniform interface to various kinds of objects through its basic I/O system calls. All these objects are called files. But these files could refer to different things like, regular file, character device, block device, named pipe (FIFO), symbolic link, or a network socket.

So we can use read() and write() system calls to:

- Read and write to files on hard disk or floppy disk
- Read and write to I/O devices like printers, modems, terminals, Audio devices
- Read and write to IPC object such as Pipes and FIFOs
- Read and write to network objects such as Sockets

Linux supports two types of I/O calls. First one is called basic I/O and it is the topic of this section. The second type of I/O is called standard I/O or buffered I/O. This standard I/O calls are part of ANSI C standard library and available on all most all operating systems. Where as basic I/O calls are specific to Linux(Unix) platform and are standardized by POSIX and X/OPEN standards. Standard I/O calls are discussed in the next section.

Following are the basic I/O functions:

- open()
- read()
- write()
- lseek()
- fcntl()
- ioctl()
- close()
- dup()
- dup2()
- unlink()

open System call

Before doing any operation on a file, one needs to create or open the file. Typically *creat()* is used to create a new file and *open()* is used to open an existing file. However we can use *open()* for both to create a new file as well as to open an existing file.

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

int open(const char *pathname, int oflag, ... /* , mode_t mode */ );
```

The *pathname* is the name of the file to open or create. While opening or creating a file we can give lot of options through *oflag* parameter. The last parameter *mode* is optional and should be given only when *open()* is used to create a new file. Following are the option flags for open call. All these options are defined in *fcntl.h* file.

O_RDONLY	Open for reading only
O_WRONLY	Open for writing only
O_RDWR	Open for reading and writing

Only any one of the above options should be specified. Any of the following options can be added to the above options by ORing with the above flags.

O_APPEND	If we open a file using this option, then all write operations to the file will happen only to the end of the file. When multiple processes(programs) are writing to a same file, this option provides atomic operation of seeking to the end of file and writing at the end.
O_CREAT	Creates the file if it doesn't exist. This option requires a third argument to the open() function, the `mode`, which specifies the action permission bits of the new file.
O_EXCL	Generates an error if O_CREAT is also specified and the file already exists. This test for whether the file already exists and the creation of the file if it doesn't exist is an atomic operation.
O_TRUNC	If the file exists, and if the file is successfully opened for either write-only or read-write, truncates its length to zero.
O_NONBLOCK	If the pathname refers to a FIFO, a block special file, or a character special-file, this option sets the non-blocking mode for both the opening of the file and for subsequent I/O.

creat System call

```
int creat(const char *pathname, mode_ t mode);
```

The *creat()* system call is used to create new file. However we can create files by using *open()* system call itself, without using *creat()* system call. In fact *open()* system call is more flexible to create files with different options. The *creat()* always uses fixed options. The *creat()* is identical to the following *open()* function.

```
open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

close System call

An opened file is closed by *close()* system call.

```
int close(int filedes);
```

When a process terminates, the kernel automatically closes all open files.

lseek System call

Every open file has an associated “current file offset”, we may call this current file position. This is a non-negative integer that measures the number of bytes from beginning of the file. Read and write operations normally start at the current file offset and cause the offset to be incremented by the number of bytes read or written. By default, this offset is initialized to 0 when a file is opened, unless the O_APPEND option specified.

The “current file offset” or file position can be explicitly modified by calling `lseek()`.

```
off_t lseek(int filedes, off_t offset, int whence);
```

The interpretation of the `offset` depends on the value of the `whence` argument.

- If `whence` is `SEEK_SET`, the file’s offset is set to the offset bytes from the beginning of the file.
- If `whence` is `SEEK_CUR`, the file’s offset is set to its current value plus the `offset`. The `offset` can be positive or negative.
- If `whence` is `SEEK_END`, the file’s offset is set to the size of the file plus the `offset`. The `offset` can be positive or negative.

read System call

```
ssize_t read(int filedes, void *buff, size_t nbytes);
```

write System call

```
ssize_t write(int filedes, const void *buff, size_t nbytes);
```

unlink System call

```
int unlink(char *pathname);
```

Sample Programs

Hexdump Program

This program takes file name as command line argument, and reads one byte at a time and prints the byte in hex form. It prints 16 bytes in a row. At the beginning of each row it prints the offset (i.e. position of byte in the file) of the first byte of the row. This hexdump program is very useful to analyse the content of any text file or binary file.

`File: hexdump.c`

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

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

int main(int argc, char *argv[])
{
  int fd;
  char ch;
  int offset = 0;

  if(argc < 2)
  {
    printf("Filename is missing\n");
    exit(1);
  }

  fd = open(argv[1],O_RDONLY);
  if(fd < 0)
  {
    printf("Could not open file\n");
    exit(2);
  }

  while(read(fd,&ch,1)==1)
  {
    if((offset % 16) == 0)
      printf("\n%08x  ", offset);
    printf("%02x ", ch);
    offset++;
  }
  printf("\n");
  close(fd);
}
```

Advanced Hexdump Program

This is program is similar to above one, but it displays every row both in hex and ASCII format. If a byte in the file is a printable ASCII code (0x20 to 0x7F), it is printing that character. If byte is not a printable ASCII code, it is printing ‘.’.

File: ahexdump.c

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

int main(int argc, char *argv[])
{
  int fd,jj,ii=0;
  unsigned char ch;
  int offset = 0;
  unsigned char buf[16];

  if(argc < 2)
  {
    printf("Filename is missing\n");
    exit(1);
  }

  fd = open(argv[1],O_RDONLY);
  if(fd < 0)
  {
    printf("Could not open file\n");
    exit(2);
  }

  while(read(fd,&ch,1)==1)
  {
    if((offset % 16) ==0)
      printf("\n%08x  ", offset);
    printf("%02x ", ch);
    offset++;

    buf[ii++] = ch;
    if(ii==16)
    {
      printf("  ");
      for(jj=0; jj<16; jj++)
      {
        if((buf[jj] >= 0x20) && (buf[jj] < 0x80))
          putchar(buf[jj]);
        else
          putchar('.');
      }
      ii=0;
    }
  }
  for(jj=ii;jj<16;jj++)
    printf("   ");

  printf("  ");
  for(jj=0; jj<ii; jj++)
  {
    if((buf[jj] >= 0x20) && (buf[jj] < 0x80))
      putchar(buf[jj]);
    else
      putchar('.');
  }
  printf("\n");
  close(fd);
}
```

Following is the partial output of above program, when ‘ahexdump.c’ is given as argument

```
$ ./a.out ahexdump.c

00000000  23 69 6e 63 6c 75 64 65 20 3c 73 74 64 69 6f 2e   #include <stdio.
00000010  68 3e 0a 23 69 6e 63 6c 75 64 65 20 3c 73 79 73   h>.#include <sys
00000020  2f 74 79 70 65 73 2e 68 3e 0a 23 69 6e 63 6c 75   /types.h>.#inclu
00000030  64 65 20 3c 73 79 73 2f 73 74 61 74 2e 68 3e 0a   de <sys/stat.h>.
00000040  23 69 6e 63 6c 75 64 65 20 3c 66 63 6e 74 6c 2e   #include <fcntl.
00000050  68 3e 0a 0a 69 6e 74 20 6d 61 69 6e 28 69 6e 74   h>..int main(int
00000060  20 61 72 67 63 2c 20 63 68 61 72 20 2a 61 72 67    argc, char *arg
00000070  76 5b 5d 29 0a 7b 0a 20 20 69 6e 74 20 66 64 2c   v[]).{.  int fd,
00000080  6a 6a 2c 69 69 3d 30 3b 0a 20 20 75 6e 73 69 67   jj,ii=0;.  unsig
00000090  6e 65 64 20 63 68 61 72 20 63 68 3b 0a 20 20 69   ned char ch;.  i
. . .
```

Text file creator program

This program reads a line of text at a time from the user, and writes that line to the file. User has to enter Control-D to quit the program.

File: txtfcrt.c

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

int main(int argc, char *argv[])
{
  int fd;
  unsigned char buf[256];

  if(argc < 2)
  {
    printf("\nFilename is missing\n");
    exit(1);
  }

  fd = open(argv[1], O_RDWR | O_CREAT | O_TRUNC, 0660);
  if(fd < 0)
  {
    printf("Could not open file\n");
    exit(2);
  }
  printf("\n\nSimple text file creator\n");
  printf("Enter few lines, When finished press control-D to quit\n");
  while(fgets(buf,256,stdin))
  {
    write(fd,buf,strlen(buf));
  }
  close(fd);
}
```

Program to read data from serial port and display

This program first opens the serial port and initializes it to use baudrate 19200. Next it reads the data from the serial port and displays it. This program can be tested in two methods. One is to connect serial port of this computer to serial port of another computer. Run terminal emulator program 'minicom' on the other computer. Next type characters at the mincom terminal. Whatever characters typed at the minicom terminal should be read and displayed by this program.

Second method is to connect DEPIK's DPK-51 microcontroller kit to serial port. Then this program will display whatever data sent by microcontroller kit.

File: serial.c

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>

int serial_init()
{
  int sfd;
  struct termios tio;

  sfd = open("/dev/ttyS0", O_RDWR | O_NOCTTY);
  if(sfd < 0)
  {
    perror("Unable to open /dev/ttyS0");
    exit(1);
  }

  tcgetattr(sfd, &tio);
  tio.c_cflag        = B9600 | CS8 | CLOCAL | CREAD;
  tio.c_iflag        = 0;
  tio.c_oflag        = 0;
  tio.c_lflag        = 0;
  tcflush(sfd, TCIFLUSH);
  tcsetattr(sfd,TCSANOW,&tio);
  return sfd;
}

int main()
{
  int sfd, n;
  char buf[128];

  sfd = serial_init();
  while(1)
  {
    n = read(sfd, buf, 128);
    if(n<0)
    {
      exit(2);
    }
    buf[n] = 0;
    printf("%s",buf);
  }
}
```

Blocking Concept and Device I/O

The concept of blocking is very important one to understand. When we run a program, and that program calls getchar(), gets(), or scanf() function; you might have observed that the program stops till you enter data. When a process (program) tries to read data from a device and no data is available with the device, the process goes blocking state or sleeping state. During this state, kernel keeps it a separate queue and never allocates any CPU time to this blocking process. So blocking process will not waste any CPU time.

If you read from a file (assume that file's offset is reached to 'end of file') and there is no further data to read from file, then read() system call returns zero. Similarly if you try to read from a device (such as serial port or standard input device), and there is no data with the device, then the read() system call puts the process in blocking state. It comes out of blocking state to ready state when data is available with the device.

Note that, blocking of process happens till at least one byte of data is available, it will not block till all the data (length specified to read system call). So read system call returns even if one byte of data is available.

It is easy to understand that, read() is a blocking call, as program blocks every time you try to read from input device. But note that, write() is also a blocking call, when we write to an I/O device. When we write() to an I/O device, the data is placed in the kernel buffer associated with the device, and write() call returns immediately. Kernel will ensure that, data from the device buffer will be sent to the device as and when device is ready to take data. But if you write lot of data quickly, the device buffer will become full and kernel puts the writing process to blocking state. Once sufficient space is available in the device buffer, kernel makes the process ready. Now ready process writes all the data to device buffer and write() function returns.

File System Kernel Data Structures

For every running program (i.e. process), kernel will maintain a very important data structure called Process Control Block (PCB). PCB also referred as Process descriptor. The PCB contains lot of information about a process in the form of various structures. One of the important strucure is 'file descriptor table' or simply 'FD Table'. This table holds array of entries called 'file descriptors'.

When we call an open() system call to open a file, the control will switch to the kernel. The kernel searches the block device (hard disk) for the file's 'inode'. For every file, there exist an 'inode' information in the block device. Open call, creates inode structure in the memory and copies inode content from block device to inode structure in memory. Next it creates a new structure called 'File object'. This file object structure contains 'file status flags', 'file offset' and pointer to inode structure created in the above step. Next open call, finds a free entry in the FD table and fills that FD table entry with a pointer to the 'File object' structure. Finally open call returns index of ths FD table entry to the application. This is illustrated in the following figure.

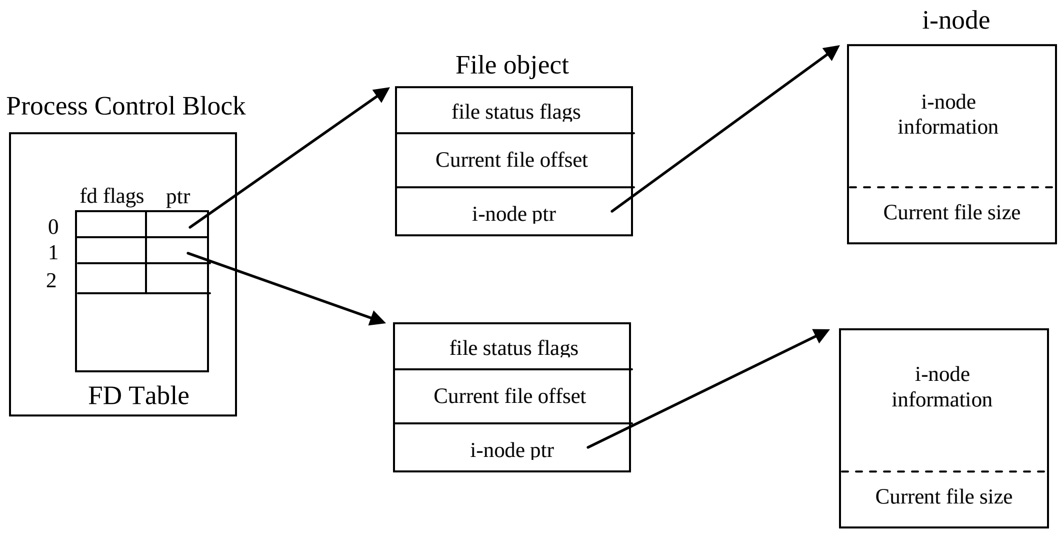


Figure: Kernel data structures for open files

For every opened file of a process, there exist one entry in the FD table. The index of this entry only returned to the application by the open() system call. However the first three entries (i.e fd numbers 0, 1 and 2) of FD table are already opened for every process. These three enties are pointed to the standard input device, standard output device and standard error device. For every process there will be a terminal device (or window), which acts as standard input, output and error device to the process. So one can use fd numbers 0, 1 and 2 directly as shown in the following program.

File: stdfds.c

```
int main()
{
  char buf[80];
  int n;

  write(1, "This is written to fd number 1\n",31);
  write(2, "This is written to fd number 2\n",31);
  write(1, "Enter a line of text\n",21);
  n = read(0, buf, 80);
  write(1, "Following is the line i read\n",29);
  write(1,buf,n);
}
```

Opening of a file with multiple processes

Multiple processes can open a single file. Then each process will have a separate file table entry. But all these file table entries point to the same v-node entry. As each process is getting a separate file table entry, file offset is maintained separately for each process. The following figure shows opening of same file with two independent processes.

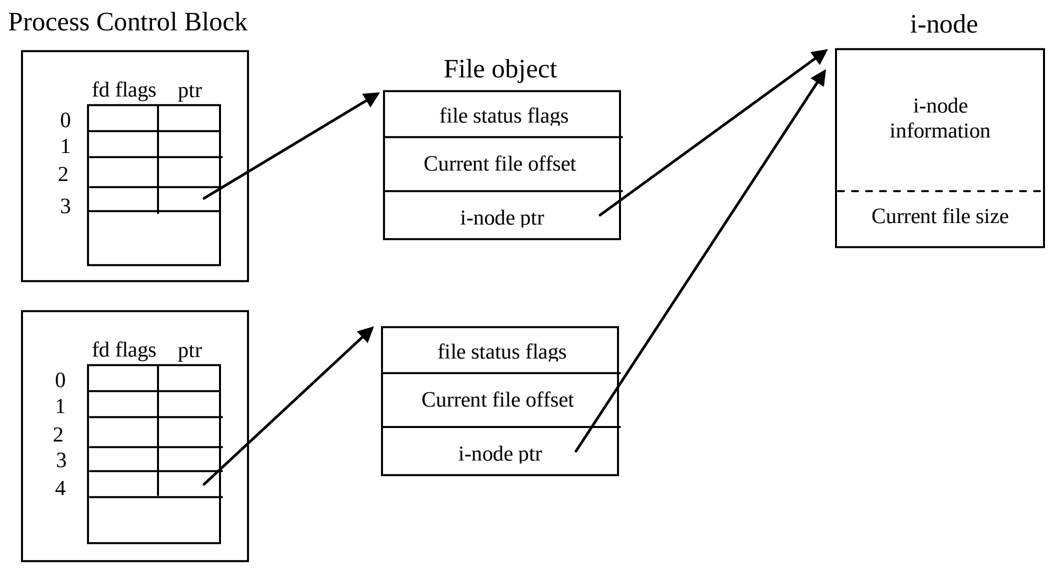


Figure: Two independent processes with same file open

Duplicating file descriptors

An existing file descriptor can be duplicated by using either of the following functions.

```
#include <unistd.h>
int dup(int filedes);
int dup2(int filedes, int filedes2);
```

```
                  Both return : new file descriptor if OK, -1 on error
```

The *dup()* system call duplicates the given descriptor by copying its contents to new descriptor entry and returns index of new descriptor. The new file descriptor returned by *dup()* is guaranteed to be the lowest numbered available descriptor. With *dup2()* we specify the value of the new descriptor with *filedes2* argument. If *filedes2* is already open, it is closed first. The new file descriptor that is returned as value of the functions shares the same file table entry as the *filedes* argument.

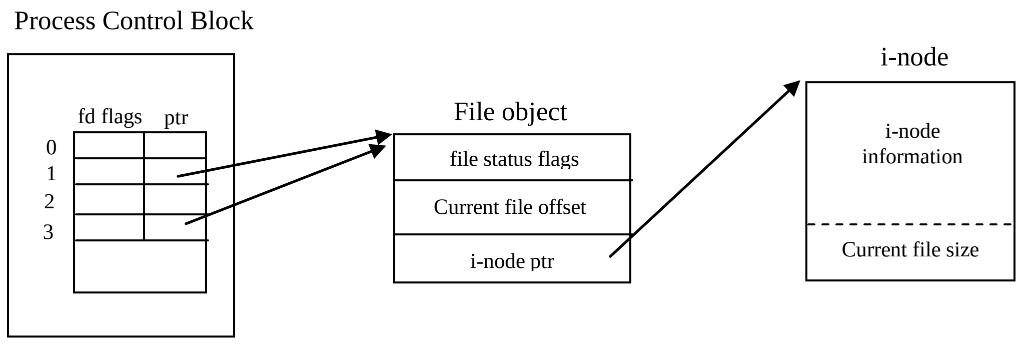


Figure: Kernel data structures after dup(1)

Use of dup() system call

The *dup()* system call is very useful to redirect the standard input or output to some other file, device or communication object. If you open a file and dup the file descriptor of this file onto a standard output, then whatever process writes to standard output using *printf()* or *puts()* functions will go to this file. When you run the following program, the output of printf() statements will not get displayed on the terminal. Instead, output will got ‘dupdemo.txt’ file.

File: dup.c

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

int main()
{
  int fd;

  fd = open("dupdemo.txt", O_RDWR | O_CREAT | O_TRUNC,0660);
  if(fd < 0)
  {
    printf("Could not open file\n");
    exit(2);
  }
  dup2(fd,1);
  printf("Sample string one\n");
  printf("Sample string two\n");
  puts("Sample string three\n");
}
```

Information about Files

When you give `'ls -l'` command, it displays lot of information about each file. To get such file information from a program we can use the following system calls.

```
#include <sys/types.h>
#include <sys/stat.h>
int stat (const char *pathname, struct stat *buf);
int fstat (int filedes, struct stat *buf);
int lstat (const char *pathname, struct stat *buf);
```

All the above three functions gives the information about a given file in the '*stat*' structure. The first function *stat()* takes the name of the file as input, where as *fstat()* function takes the file descriptor as input. The *lstat()* function is similar to stat() function, it only differs when given filename refers to a symbolic link. When symbolic link file is given as input, the *stat()* function gives the information about the file to which this symbol link is pointing to. But *lstat()* function gives the information about this symbol link itself.

Following is the structure declaration for 'stat' structure.

```
struct stat
{
  mode_t  st_mode;   // file type and mode (permissions)
  ino_t   st_ino;    // I-node number
  dev_t   st_dev;    // device number (file system)
  dev_t   st_rdev;   // device number for special files
  nlink_t st_nlink;  // number of links
  uid_t   st_uid;    // user ID of owner
  gid_t   st_gid;    // group ID of owner
  off_t   st_size;   // size in bytes, for regular files
  time_t  st_atime;  // time of last access
  time_t  st_mtime;  // time of last modification
  time_t  st_ctime;  // time of last file status change
  long    st_blksize // best I/o block size
  long    st_blocks  // number of 512-byte blocks allocated
}
```

File types

When you use `'ls'` command to list the files present in directory, it displays all the files present in that directory. Each of these files may refer to different types files. In Linux, a file could be any of the following types:

Regular File	What we normally refer as files are these regular files
Directory File	Every directory in Linux is a file containing file names and other info
Character Device File	Character devices such as serial ports and parallel are represented with these files
Block Device File	Block devices such as floppies, hard disks and CD-ROMs are represented
FIFO	This type of file is used for inter-process communication. This is also called named pipe.
Socket	This file is used for network communication between processes running on different machines or on the same machine
Symbolic Link	This is file, which contains a pointer (name) to another file. This allows a single file to have multiple names

Files, Processes, Users and Groups

In Linux, there exist associations or links between files, processes, users and groups. Each user of Linux will have unique user ID. This user ID is maintained both as a number and name. The "`passwd`" file in `/etc` directory will list all user names and their ID numbers. Similarly all group names and their ID numbers are maintained in "`group`" file of `/etc` directory.

As seen in the above section, every file is associated with user ID (owner's user id) and group Id (owner's group Id). In a similar way, every process running in a system will associate with the following Ids.

- Real user ID
- Real Group ID
- Effective user ID
- Effective group ID
- Supplementary group Ids

Standard I/O or Buffered I/O

The standard I/O functions are library functions developed over Unix basic I/O system calls. The ANSI C standard specifies this library and this library is available on most of the operating systems that supports C compilers. Because of this reason these are called standard I/O library functions. These are also called buffered I/O calls, because these functions buffers the data before calling system calls to transfer data. The reason for buffering is to improve the efficiency of I/O transfers when a small amount of data is used to transfer.

Applications normally require to write one character at a time or one line at a time to files. If applications use I/O system calls for writing characters or line, there will be more overhead. Each system call requires switching to kernel mode and executing the system call and switching back to user mode. So one should avoid making many numbers of system calls to improve the efficiency and standard I/O will just to that.

For example, an application may be writing only one character at time to the file. The buffered I/O writes these characters into its internal buffer first and writes to the file only when the buffer is full. So this reduces the number of system calls made and improves the performance. In the same way when application reads a single character from a file, the buffered read API reads a buffer from the file and returns a single character to the application. When application reads next character, the read API returns it from its internal buffer itself.

But note that, when a large size of data transfer is involved, buffering may be inefficient as it involves multiple copying.

The basic I/O calls are centered around a concept of file descriptor. When file is opened with basic I/O calls, a file descriptor is returned, and that file descriptor is used in all other basic I/O calls to do operations on that file.

The standard I/O calls are centered around a concept stream. When we open a file with standard I/O calls, what we receive is a pointer to a stream. This stream is represented using a FILE structure or FILE object. This FILE object internally maintains file descriptor associated with the file, the buffer to hold stream data, size of buffer and count of bytes presently in the buffer etc..

The same way how first three descriptors (0,1 and 2) are automatically available to a process, the streams associated with these three file descriptors are also available to the process with the following file pointer variables:

- stdin
- stdout
- stderr

Buffering

The goal of the buffering provided by the standard I/O library is to use the minimum number of `read()` and `write()` calls. There are three type of buffering provided. These are:

1. Fully buffered. In this case actual I/O takes place when the standard I/O buffer is filled. Files that reside on disk are normally fully buffered by the standard I/O library.
2. Line buffered. In this case the standard I/O library performs I/O when a new line character is encountered on input or output.
3. Unbuffered. The standard I/O library does not buffer the characters.

By default standard input and output are fully buffered, if and only if they do not refer to an interactive device. Standard error is never fully buffered. But we can change this default-buffering scheme by using following system calls.

```
void setbuf(FILE *fp, char *buf);
int setvbuf(FILE *fp, char *buf, int mode, size_t size);
```

With `setbuf()` we can turn buffering on or off. To enable buffering, `buf` must point to a buffer of length BUFSIZ. But this buffering could be fully buffered or line buffered based on type of device associated with stream. To disable buffering, we set buf to NULL.

With `setvbuf()` we specify exactly which type of buffering we want. This is done with the `mode` argument.

_IOFBF	fully buffered
_IOLBF	line buffered
_IONBF	unbuffered

If we give mode as unbuffered, the `buf` and `size` arguments are ignored. If we specify mode as fully buffered or line buffered, buf and size can optionally specify a buffer and its size.

Types of Regular Files

Files (more precisely regular files) are useful for storing data. This data could be text information, database records, documents, audio data, video data, image data, executable instructions, etc.. But all these files could be divided into just two categories like text files and binary files. The clear understanding of differences between Text files and Binary files is very essential for using the files in applications. This difference is identical to difference between Text messages and Binary messages studied in advanced C module.

All files both binary and text files can be treated as an array of bytes. The size of this array is same as the size of the file.

Text Files

In text files, each byte will be a valid ASCII code. Valid ASCII codes ranges from 0 to 127. Out of these 127 codes, first 32 (0 to 31) codes represents controller characters or non-printable characters. Typically text files contains only printable characters. Only few control characters like carriage return, line feed and TAB will be used in text files. Following are the typical text files.

Program and script source files, text files, mail files, html files, XML files, database files in text format.

All text files can be viewed or modified using text editors such as `‘vi’.`

Binary Files

In binary files, each byte of file can take any value from 0 to 255. Most of the files are binary files. Each binary file will have some format. One should know the format of a file to interpret the content of binary files. For example the format of a bitmap file contains the header structure and an array of elements. The header structure describes the number of rows and columns in the bitmap and also size of each pixel. Header will be followed by pixel data.

Object files, executable files, audio/music files, bitmap files, video files and database files are some examples for binary files.

Using Standard I/O functions

All the standard I/O functions works more or less similar to system calls covered previously. While system calls operate directly on file descriptors of kernel, these standard I/O calls operate on streams (* FILE) in user space. For every system call studied, we have a corresponding standard I/O function. The parameters to these system calls and standard I/O functions differs slightly. One should note this carefully.

Opening and Closing a Stream

```
FILE *fopen(const char *pathname, const char *type);
int fclose(FILE *fp);
```

Positioning a Stream

Whenever a stream is opened, its file position is set to zero. As we read or write data this file position will be moved automatically. But we can modify the file position to read or write to anywhere within the file by using following standard I/O functions.

```
long ftell(FILE *fp);
int fseek(FILE *fp, long offset, int whence);
void rewind(FILE *fp);
```

The *`ftell()`* function returns the current position of the stream. The *`fseek()`* function changes the file position to the given `offset` value. This `offset` value is specified with reference to file beginning or file end or current file position. The reference is specified with whence argument. The `rewind()` function sets the file position to zero. The *`fseek()`* function is identical to the *lseek()* system call studied previously.

Binary I/O

Reading and writing structures to files

When we want to store structures like student record, or book record in a file, we could do that in two ways. One is to store the record as it is from memory to the file. This is called storing the structure in binary form. Second is to convert each field of string into string and store all the strings as a single line in the file. You may put some delimiting character between each filed. This way of storing is called storing in text form.

Storing in binary is more efficient. But only programs can read these files. We can't view or edit them through standard text editors. Also the integers will be stored in little endian or big endian format depending on the processor. So same file will not give correct data if read on a different endian processor.

If you store in text form, one can view the files easily using text editors. But storing in text form may require more memory space. Also conversion from binary to text and text binary is required whenever file is written or read. With text files there will not be any endian issues.

In ascii format
In Binary format

If we want to delete a record from the file, there is no simple way to do this. Only way is to read all the other records (except record to be deleted) and write them into a different file. Delete the first file and rename the new file with old name. As this involves lot of steps, typically to delete a record, simple way is to modify one field of record to an invalid value. So when records are read from the file, the records with invalid field will not be used. And when a new record is added, instead of adding to the end of the file, search first for the deleted record and overwrite on it.

If a record present in the file need to be modified, first read the record into memory, modify it in the memory and overwrite it on the original record of the file.

Following standard I/O calls are used for reading and writing binary files.

```
size_t  fread(void *ptr, size_t size, size_t nobj, FILE *fp);
size_t  fwrite(const void *ptr, size_t size, size_t nobj, FILE *fp);
```

I/O on Text files

Following are the typical operations on text files.

- Reading or writing one character at a time
- Reading or writing one line at a time
- Formatted I/O to read and write binary values to text files after appropriate conversion

The following functions are useful read one character at a time.

```
int getc(FILE *fp);
int fgetc(FILE *fp);
int getchar(void);
```

The following functions are useful write one character at a time.

```
int putc(int c, FILE *fp);
int fputc(int c, FILE *fp);
int putchar(int c);
```

The following functions are used to read one line at a time. The first function `fgets()` is a generic function to read from any given stream. Where as second function `gets()` reads always from standard input stream. But using of gets() function is discouraged because with this function we can specify the size of the buffer. So we can use fgets() function only to read from standard input stream.

```
char *fgets(char *buf, int n, FILE *fp);
char *gets(char *buf);
```

The following functions are used to write one line at a time. The first function `fputs()` is a generic function to write to any given stream. Where as second function `puts()` write given string always to a standard output stream.

```
int fputs(const char *str, FILE *fp);
int puts(const char *str);
```

Formatted I/O

All the variables in memory are stored in binary format. Computer will provide operations only on binary numbers. But when we want to display these numbers we need to convert each number into a string of decimal ASCII codes. And these ASCII codes can be written on to a console or printer. In the same way, when user enters a number from keyboard, what he enters is a string of ASCII decimal characters representing the number. So to store this number into a variable, this string needs to be converted into a binary integer format.

The following functions support the first type of conversion that is from integer to ASCII.

```
int printf(const chat *format, ...)
int fprintf(FILE *fp, const chat *format, ...)
int sprintf(char *buf, const chat *format, ...)
```

All the above functions returns number characters output. The `printf`() function outputs the converted decimal numbers on to a standard output. The second function `fprintf`() outputs on to a given file pointed by file pointer. And the last function `sprintf`() stores the characters into a given buffer. This last function is very useful for doing conversions, without outputting.

Following are the formatted input functions for converting from ASCII to binary.

```
int scanf(const chat *format, ...)
int fscanf(FILE *fp, const chat *format, ...)
int sscanf(char *buf, const chat *format, ...)
```

Text file reader program

This program reads a line at a time from the given file and prints that line on the terminal.

File: txtfrdr.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  FILE *fp;
  char buf[128];

  if(argc < 2)
  {
    printf("Filename is missing\n");
    exit(1);
  }

  fp = fopen(argv[1], "r");
  if(fp==0)
  {
    printf("Could not open file\n");
    exit(2);
  }

  while(fgets(buf,128,fp))
  {
    printf("%s",buf);
  }
  fclose(fp);
}
```

Program to write numbers to file in binary form

This program reads one integer at a time from the user, and writes it to file in the binary form. It exits the program if number zero is entered.

File: wbinnums.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  FILE *fp;
  int num;

  if(argc < 2)
  {
    printf("Filename is missing\n");
    exit(1);
  }

  fp = fopen(argv[1], "w");
  if(fp==0)
  {
    printf("Could not open file\n");
    exit(2);
  }
  printf("\nEnter integers to store in file\n");
  printf("Enter 0 to exit\n\n");
  while(1)
  {
    scanf("%d",&num);
    if(num==0)
      break;
    fwrite(&num, sizeof(int), 1, fp);
  }
  fclose(fp);
}
```

Program to read numbers in binary form

This program reads one integer at a time from the file, and prints it.

rbinnums.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  FILE *fp;
  int num;

  if(argc < 2)
  {
    printf("Filename is missing\n");
    exit(1);
  }

  fp = fopen(argv[1], "r");
  if(fp==0)
  {
    printf("Could not open file\n");
    exit(2);
  }
  printf("\nFollowing are the numbers stored in file in binary form\n\n");
  while(fread(&num,sizeof(int), 1, fp)==1)
  {
    printf("%d\n", num);
  }
  printf("\n");
  fclose(fp);
}
```

Program to write numbers to file in text form

This program reads one integer at a time from the user, and writes it to file in the text form.
It exits the program if number zero is entered.

wtxtnums.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  FILE *fp;
  int num;

  if(argc < 2)
  {
    printf("Filename is missing\n");
    exit(1);
  }

  fp = fopen(argv[1], "w");
  if(fp==0)
  {
    printf("Could not open file\n");
    exit(2);
  }
  printf("\nEnter integers to store in file\n");
  printf("Enter 0 to exit\n\n");
  while(1)
  {
    scanf("%d",&num);
    if(num==0)
      break;
    fprintf(fp,"%d\n",num);
  }
  fclose(fp);
}
```

Program to read numbers from file in text form

This program reads one integer at a time from the file, and prints it. The numbers in the file are in text format. The fscanf() function is responsible for converting numbers in text form to binary form.

`rtxtnums.c`

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  FILE *fp;
  int num;

  if(argc < 2)
  {
    printf("Filename is missing\n");
    exit(1);
  }

  fp = fopen(argv[1], "r");
  if(fp==0)
  {
    printf("Could not open file\n");
    exit(2);
  }
  printf("\nFollowing are the numbers stored in file in text form\n\n");
  while(fscanf(fp,"%d",&num)==1)
  {
    printf("%d\n",num);
  }
  printf("\n");
  fclose(fp);
}
```

Demo Application Programs

The following sample program shows how to maintain structures in a file. The first program “*bfphonedb.c*” maintains telephone records in binary file. This application supports, display, addition, deletion, modification and search of telephone records present in the binary file.

The second program “*tfphonedb.c*” maintains the records in the text file.
Use these programs just as a reference. Implement all the assignments in your own way.
Whenever have a doubt refer the following code. Never copy this program as it is.

File: bfphonedb.c

```
#include <stdio.h>
#include <fcntl.h>

/********************* MACROS *********************/
```

```
#define SUCCESS                0
#define DATA_BASE_FULL_ERR     1
#define ID_NOT_FOUND_ERR       2

#define MAX_PHONE_RECS 100

/************** Structure Type definitions **********/
typedef struct
{
  int   id;
  char  name[40];
  int   phone;
}phrec_t;

#define REC_SIZE  sizeof(phrec_t)

/************ Function Prototypes **************/
void disp_help();
void disp_recs();
int  add_rec();
int  del_rec();
int  mod_rec();
void search_rec();

/*********** Global data definitions *************/
int    fd;

/************* Function Definitions **************/
int main()
{
  int option;

  printf("\nTelephone directory Program, Ver 1\n\n");
  fd = open("db.dat", O_RDWR);
  if(fd<0)
  {
    printf("Creating file...\n");
    fd = open("db.dat", O_RDWR | O_CREAT, 0600);
    if(fd <0)
    {
      printf("Could not create file\n");
      exit(1);
    }
  }
  disp_help();

  while(1)
  {
    printf("\nEnter option: ");
    scanf("%d",&option);
    switch(option)
    {
      case 1:
        disp_recs();
       break;

      case 2:
        if(add_rec()!=SUCCESS)
         printf("Record add operation failed\n");
       break;

      case 3:
         if(del_rec()!=SUCCESS)
          printf("Record delete operation failed\n");
       break;

      case 4:
        mod_rec();
       break;

      case 5:
        search_rec();
       break;

      case 6:
       printf("Good bye\n");
        return SUCCESS;

      default:
       printf("Invalid option. Use the following\n");
        disp_help();
       break;
    }
  }
  return SUCCESS;
}

void disp_help()
{
  printf("Enter 1 to display all records\n");
  printf("      2 to add a record\n");
  printf("      3 to delete a record\n");
  printf("      4 to modify a record\n");
  printf("      5 search records with matching name\n");
  printf("      6 to exit\n");
}

void disp_recs()
{
  int len;
  phrec_t rec;

  lseek(fd,0,SEEK_SET);
  len = read(fd,&rec,REC_SIZE);
  if(len != REC_SIZE)
  {
    printf("No records to print\n");
    return;
  }

  printf(" ID          Name             Phone \n");
  printf("---- --------------------   --------\n");
  while(len == REC_SIZE)
  {
    if(rec.id != -1)
      printf("%3d  %20s   %8d\n",rec.id,rec.name,rec.phone);
    len = read(fd,&rec,REC_SIZE);
  }
}

void strip_newline(char *name)
{
  int len;
  len = strlen(name);
  if(name[len-1] == '\n')
    name[len-1]= 0;
}

int add_rec()
{
  int len;
  phrec_t rec,trec;

  printf("Enter   id: ");
  scanf("%d",&rec.id);

  printf("Enter Name: ");
  __fpurge(stdin);
  fgets(rec.name,40,stdin);
  strip_newline(rec.name);

  printf("Enter Phone Number: ");
  scanf("%d",&rec.phone);

  lseek(fd,0,SEEK_SET);
  while(1)
  {
    len = read(fd,&trec,REC_SIZE);
    if(len != REC_SIZE)
      break;
    if(trec.id == -1)
    {
      lseek(fd,REC_SIZE * -1,SEEK_CUR);
      write(fd,&rec,REC_SIZE);
      return SUCCESS;
    }
  }
  write(fd,&rec,REC_SIZE);
  return SUCCESS;
}

int del_rec()
{
  int id,len;
  phrec_t rec;

  printf("Enter id of record to delete: ");
  scanf("%d",&id);
  lseek(fd,0,SEEK_SET);
  while(1)
  {
    len = read(fd,&rec,REC_SIZE);
    if(len != REC_SIZE)
      break;
    if(rec.id == id)
    {
      lseek(fd,REC_SIZE * -1,SEEK_CUR);
      rec.id = -1;
      write(fd,&rec,REC_SIZE);
      return SUCCESS;
    }
  }
  return ID_NOT_FOUND_ERR;
}
```

```
int mod_rec()
{
  int len, id, phone;
  phrec_t rec;

  printf("Enter id of record to modify: ");
  scanf("%d",&id);

  lseek(fd,0,SEEK_SET);
  while(1)
  {
    len = read(fd,&rec,REC_SIZE);
    if(len != REC_SIZE)
      break;
    if(rec.id == id)
    {
      lseek(fd,REC_SIZE * -1,SEEK_CUR);
      printf("Enter new phone number: ");
      scanf("%d",&phone);
      rec.phone = phone;
      write(fd,&rec,REC_SIZE);
      return SUCCESS;
    }
  }
  return ID_NOT_FOUND_ERR;
}

void search_rec()
{
  phrec_t rec;
  int len, recs=0;
  char str[40];

  printf("Enter name string to match ");
  scanf("%s",str);

  lseek(fd,0,SEEK_SET);
  while(1)
  {
    len = read(fd,&rec,REC_SIZE);
    if(len != REC_SIZE)
      break;
    if(strstr(rec.name,str))
    {
      printf("%3d  %20s   %8d\n",rec.id,rec.name,rec.phone);
      recs++;
    }
  }

  if(!recs)
    printf("No matching records found\n");
}
```

File: tfphonedb.c

```
#include <stdio.h>
#include <fcntl.h>
#include <string.h>

/************* MACROS ******************/
#define SUCCESS                0
#define INVALID_CMD_ERR        1
```

```
#define DATA_BASE_FULL_ERR      2
#define ID_NOT_FOUND_ERR        3

#define REC_SIZE  45

/************** Structure Type definitions **********/
typedef struct
{
  int   id;
  char  name[40];
  int   phone;
}phrec_t;

typedef struct
{
  char cmdstr[32];
  int  (*cmdfp)();
}cmd_t;

/***** Function Prototypes *************/
int  disp_help();
int  disp_recs();
int  add_rec();
int  del_rec();
int  mod_rec();
int  xit_cmd();
void line2rec(char *lbuf, phrec_t *pr);

/********* Global data definitions **************/
FILE   *fp;
cmd_t cmds[] =
{
       {"add", add_rec},
       {"disp", disp_recs},
       {"del", del_rec},
       {"mod", mod_rec},
       {"help", disp_help},
       {"exit", xit_cmd}
};
int no_of_cmds = sizeof(cmds)/sizeof(cmd_t);


int main()
{
  char cmd[100];
  int ii;
  int stat;

  printf("\nTelephone directory Program, Ver 1\n\n");
  fp = fopen("db.txt", "r+");
  if(!fp)
  {
    fp = fopen("db.txt", "w+");
    if(!fp)
    {
      printf("Could not create db.txt file\n");
      exit(1);
    }
  }
  disp_help();

  while(1)
  {
    printf("\nEnter_cmd> ");
    scanf("%s",cmd);
    stat = INVALID_CMD_ERR;
    for(ii=0; ii<no_of_cmds; ii++)
    {
      if(strcmp(cmds[ii].cmdstr,cmd)==0)
      {
        stat = (*cmds[ii].cmdfp)();
       break;
      }
    }
    if(stat == INVALID_CMD_ERR)
    {
      printf("Invalid command.\nEnter 'help' to display valid commands\n");
    }
    else if(stat != SUCCESS)
      printf("Command failed with error %d\n",stat);
  }
  return SUCCESS;
}

int disp_help()
{
  printf("disp : Displays all the records\n");
  printf("add  : Add a new record\n");
  printf("del  : Deletes a given record\n");
  printf("mod  : Modifies a given record\n");
  printf("help : Display these help strings\n");
  printf("exit : Exits the program\n");

  return SUCCESS;
}

int  xit_cmd()
{
  fclose(fp);
  exit(0);
}

void line2rec(char *lbuf, phrec_t *pr)
{
  char *str;

  str = strtok(lbuf,",\n");
  sscanf(str,"%d",&pr->id);
  str = strtok(0,",\n");
  strcpy(pr->name,str);
  str = strtok(0,",\n");
  sscanf(str,"%d",&pr->phone);
}

int disp_recs()
{
  int     len;
  phrec_t rec;
  char    lbuf[100];

  rewind(fp);
  len = fread(lbuf,1,REC_SIZE,fp);
  if(len != REC_SIZE)
  {
    printf("No records to print\n");
    return;
  }

  while(len == REC_SIZE)
  {
    line2rec(lbuf,&rec);
    if(rec.id != -1)
      printf("%3d  %20s   %8d\n",rec.id,rec.name,rec.phone);
    len = fread(lbuf,1,REC_SIZE,fp);
  }
  return SUCCESS;
}

void strip_newline(char *name)
{
  int len;

  len = strlen(name);
  if(name[len-1] == '\n')
    name[len-1]= 0;
}

int add_rec()
{
  int     len;
  phrec_t rec,trec;
  char    lbuf[100];


  printf("Enter   id: ");
  scanf("%d",&rec.id);

  printf("Enter Name: ");
  __fpurge(stdin);
  fgets(rec.name,40,stdin);
  strip_newline(rec.name);

  printf("Enter Phone Number: ");
  scanf("%d",&rec.phone);
  rewind(fp);
  while(1)
  {
    len = fread(lbuf,1,REC_SIZE,fp);
    if(len != REC_SIZE)
      break;
    line2rec(lbuf,&trec);
    if(trec.id == -1)
    {
      fseek(fp,REC_SIZE * -1,SEEK_CUR);
      fprintf(fp,"%4d,%30s,%8d\n",rec.id,rec.name,rec.phone);
      return SUCCESS;
    }
  }
  fprintf(fp,"%4d,%30s,%8d\n",rec.id,rec.name,rec.phone);
  return SUCCESS;
}

int del_rec()
{
  int      id,len;
  phrec_t  rec;
  char     lbuf[100];

  printf("Enter id of record to delete: ");
  scanf("%d",&id);
  rewind(fp);
  while(1)
  {
    len = fread(lbuf,1,REC_SIZE,fp);
    if(len != REC_SIZE)
      break;
    line2rec(lbuf,&rec);
    if(rec.id == id)
    {
      rec.id = -1;
      fseek(fp,REC_SIZE * -1,SEEK_CUR);
      fprintf(fp,"%4d,%30s,%8d\n",rec.id,rec.name,rec.phone);
      return SUCCESS;
    }
  }
  return ID_NOT_FOUND_ERR;
}

int mod_rec()
{
  int      len, id, phone;
  phrec_t  rec;
  char     lbuf[100];

  printf("Enter id of record to modify: ");
  scanf("%d",&id);
  rewind(fp);
  while(1)
  {
    len = fread(lbuf,1,REC_SIZE,fp);
    if(len != REC_SIZE)
      break;
    line2rec(lbuf,&rec);
    if(rec.id == id)
    {
      printf("Enter new phone number: ");
      scanf("%d",&phone);
      rec.phone = phone;
      fseek(fp,REC_SIZE * -1,SEEK_CUR);
      fprintf(fp,"%4d,%30s,%8d\n",rec.id,rec.name,rec.phone);
      return SUCCESS;
    }
  }
  return ID_NOT_FOUND_ERR;
}
```

3. Threads

When we execute a program, first program will be loaded into virtual memory. Next the execution starts with *startup()* function. This *startup()* function calls *main()* function. Each statement in the *main()* function will get executed. If any statement in the *main()* function, contains a function call statement, control will go to that function. When that function returns, control will come back to *main()* function. Finally when *main()* function returns, it will returns to *startup()* function. The *startup()* function finally calls *exit()* system call to terminate the program. This execution path of a program is called '*thread of execution'* or simply '*thread'*.

So far whatever program we have written, contains only a single thread of execution. This thread as we discussed above, starts with *startup()* function and also ends with *startup()* function. It is possible to write programs that contain more than one thread. Each thread executes independently, within the same virtual memory by sharing, Text, Data, BSS and Heap areas with other threads. But each thread will have a separate stack area within the virtual address space. The programs with multiple threads of execution are called multi-threaded applications.

The threads are not part of original Unix OS. But because of advantages of threads, every OS started offering multi-threading system calls. Each flavor of Unix also started implementing its own system calls for multi-threading. Later POSIX has standardized the system calls for threads. These are referred as POSIX Threads or pthreads. Now Linux supports POSIX compliant pthreads.

Multiple threads are useful in concurrent applications. Best example for concurrent applications are network server applications. These server applications have to process messages coming from multiple clients. A separate thread is created to processes the messages coming from each client.

When an application has to respond to multiple events, then also multi-threading is useful. For example a GUI, network application program has to respond to events coming from the user as well as events coming from network. In this case two separate threads are created. While one thread handling the GUI events, the other will handle network events.

These threads are very much similar to Tasks in Real-Time Operating Systems (RTOS). We are gong to learn RTOS in the next module. Similar to threads, all tasks in RTOS will share the same address space. In the case of threads, all threads share virtual memory space of a process. Where as in the case of RTOS, all tasks share physical address space for RTOS. Normally RTOSes will not use virtual memory for efficiency and simplicity.

```
#include <pthread.h>

int pthread_create(pthread_t *pthread, pthread_attr_t *attr,
                     void *(*start_routine)(void *), void *arg);

void pthread_exit(void *retval);

int pthread_join(pthread_t th, void **thread_return);

pthread_t pthread_self();

void pthread_detach(pthread_t *pth);
```

File: pthread1.c

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

```
#include <stdio.h>

int cnt;
pthread_t th;

void * mythfun(void *arg);

int main()
{
  char ch;
  int exitstat;
  void *retptr;

  printf(“I am main thread, going to create new thread\n”);

  pthread_create(&th, NULL, mythfun, (void *)10);

  while(1)
  {
    printf("I am main thread\n");
    ch = getchar();
    if(ch=='c')
        printf("cnt = %d\n",cnt);
    if(ch=='x')
        exit(2);
    if(ch=='r')
    {
      printf("I am going wait till my child thread is terminated\n");
      pthread_join(th, &retptr);
      printf("my child terminated, the pointer returned is %d\n", retptr);
    }
  }
}

void *mythfun(void *arg)
{
  int sleepTime = (int)arg;
  while(1)
  {
    printf("I am sleeping %d seconds\n", sleepTime);
    sleep(sleepTime);
    cnt++;
    if(cnt == 10)
      return (void *)25;
  }
}
```

Simple Terminal Emulator Program with Threads

Terminals are input/output devices with a RS-232 serial interface. These are used extensively in earlier days for connecting to multi-user mini-computers and mainframes. Multiple users are able to work on a single computer, by using individual terminals connected to the computer. Terminals are also useful to interface with embedded devices, which got only RS-232 port for user interface. As terminals have become absolated, we are using terminal emulator programs. These programs make PC to act as a terminal.

The following is the implementation of simple terminal emulator program with the help of threads. In this program main thread reads from standard input and writes to the serial device. The new thread reads from the serial device and writes to the standard output. The main thread changes the terminal attributes to disable echo and canonical mode.

File: pthread2.c

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <termios.h>
#include <fcntl.h>

void * thFun(void *);
int serfd;

int main()
{
  int        res;
  pthread_t th1;
  char       ch;
  struct termios  cuset, newset;

  serfd = open("/dev/ttyS0",O_RDWR);
  if(serfd<0)
  {
    printf("Could not open serial device\n");
    exit(1);
  }
  /*** Disable echo and canonical mode of processing ***/
  tcgetattr(0,&cuset);
  newset = cuset;
  newset.c_lflag &= ~ICANON;
  newset.c_lflag &= ~ECHO;
  tcsetattr(0,TCSANOW,&newset);
  /*** create a new thread ***/
  res = pthread_create(&th1,NULL,thFun,"Thread");
  if(res != 0)
  {
    printf("Thread creation failed\n");
    exit(1);
  }
  while(1)
  {
    /*** Read from standard input ***/
    read(0,&ch,1);

    /*** Write to serial device ***/
    write(serfd,&ch,1);
  }
}

void * thFun(void *arg)
{
  char ch;
  while(1)
  {
    /*** Read from serial device ***/
    read(serfd,&ch,1);
    /*** Write to standard output ***/
    write(1,&ch,1);
  }
}
```

Thread Synchronization

Most of the times, the threads in a single process need to co-operate with each other to achieve the application's purpose. This co-operation between threads could be of three types.

- Communication between threads
- Synchronization between threads
- Mutual exclusion between threads

Communication between threads is trivial, as all the threads share the same global address space. So threads can communicate with each other through global buffers. But just global buffers alone are not sufficient for efficient communication between threads. We need synchronization. Synchronization allows a thread to wait efficiently till data is available in global buffer.

Semaphores are useful to achieve the synchronization between threads. Mutual exclusion between threads is also a kind of synchronization, with little difference. We can use semaphores for both synchronization and mutual exclusion between threads. However there exist another objects called 'Mutexes' to serve the needs of mutual exclusion. We will discuss Mutexes in the next section.

We can think, semaphore as a object, that holds tokens inside. Semaphore object just maintain a integer variable to maintain the token count. While creating a semphore, we specify the initial count of tokens. The basic operations on semaphore are very simple, one is to post a token to the semaphore. When we post a token, the semaphore token count will get incremented. The second is to wait for a token from the semaphore. When we wait for a token, if token is available (i.e token count is greater than zero), count will get decremented and wait will be over immediately. If token count is zero, the wait will block the thread till token is available.

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_destroy(sem_t *sem);
```

Following sample program illustrates the communication between threads using global buffers and POSIX semaphores for synchronization.

`File: pthread3.c`

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define MAX_MSG_LEN 256

sem_t  sem1;
char   msg1[MAX_MSG_LEN];
char   msg2[MAX_MSG_LEN];
sem_t  sem2;

void * thrdFun1(void *arg);
void toggleCase(char *buf);
```

```
int main()
{
  pthread_t thrd1;
  char      argmsg1[] = "Thread1: ";
  int       res;
  int       thNum;

  res = sem_init(&sem1,0,0);
  res = sem_init(&sem2,0,0);
  res = pthread_create(&thrd1, NULL, thrdFun1,  argmsg1);

  while(1)
  {
    printf("Enter message to send to thread\n");

    fgets(msg1,MAX_MSG_LEN,stdin);
    sem_post(&sem1);

    /** wait for response **/
    sem_wait(&sem2);
    printf("resp msg : %s\n",msg2);
  }
}

void * thrdFun1(void *arg)
{
  printf("I am %s\n",arg);
  while(1)
  {
    sem_wait(&sem1);
    strcpy(msg2,msg1);
    toggleCase(msg2);
    sem_post(&sem2);
  }
}

void toggleCase(char *str)
{
  while(*str)
  {
    if(isupper(*str))
      *str = tolower(*str);

    else if(islower(*str))
      *str = toupper(*str);
    str++;
  }
}
```

Mutual exclusion using Mutexes

Mutexes are a kind of semaphores, which holds a single token when created. Mutexes are associated with a resource. Any thread that wants to use the resouce, first should take the token from the mutex. This is called lock operation. Any other thread that wants to use resouce, will not get token as it is already taken by first thread. So second thread will goto blocking state. When first thread done with the resource, it returns the token back. Returning a token back is called unlock operation. When first thread unlocks (returns token), the second thread which is blocking, will get the token and becomes ready (unblocks).

```
#include <pthread.h>
int pthread_mutex_init( pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

In the following program, there are two threads, each trying to read two strings from the user. With the help of mutex, following program ensures that, whichever thread got the mutex will read two strings from the user. If no mutex is used, it is possible that, second string may read by other thread. Here keyboard is the resource. With mutex we are sharing the keyboard effectively.

File: pthread4.c

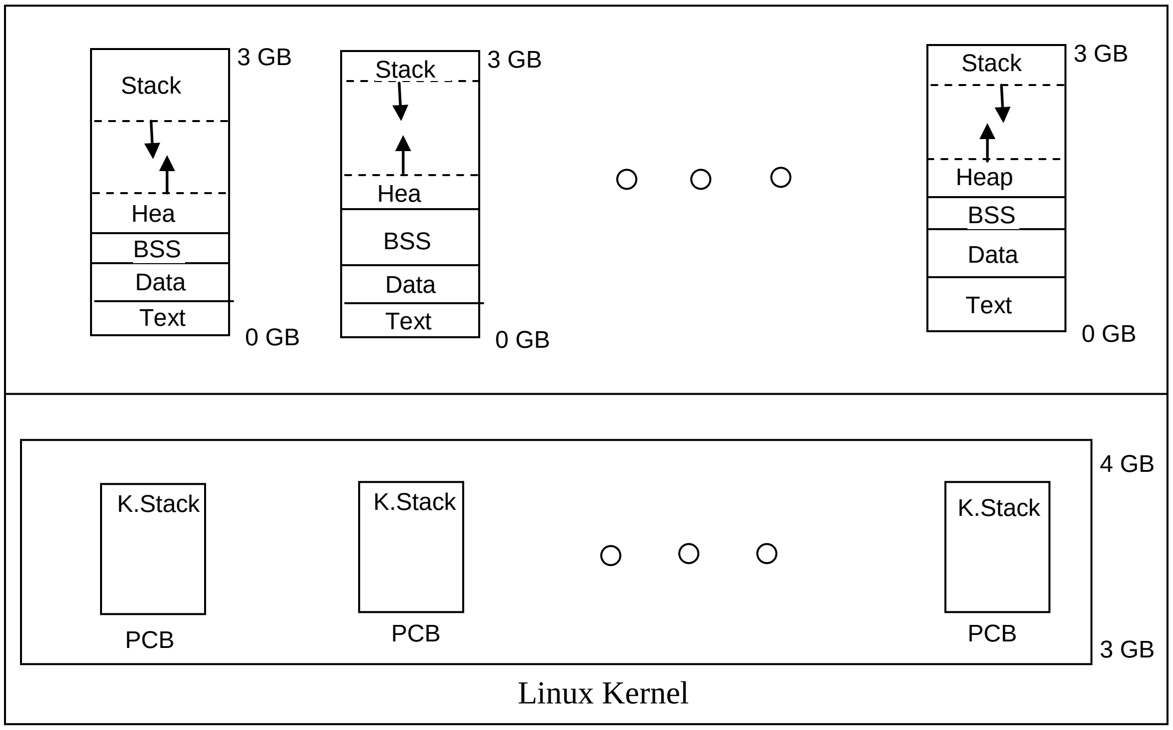
```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t  mlock;
pthread_t        th1;
void * th1fun(void * arg);

int main()
{
  char str1[80];
  char str2[40];

  if(pthread_mutex_init(&mlock, NULL) != 0)
  {
    printf("Mutex creation failed\n");
    exit(1);
  }
  pthread_create(&th1, NULL, th1fun, NULL);
  while(1)
  {
    pthread_mutex_lock(&mlock);
    printf("MainThread: Enter Two strings\n");
    fgets(str1,40,stdin);
    fgets(str2,40,stdin);
    strcat(str1, str2);
    printf("Combined string: %s\n",str1);
    pthread_mutex_unlock(&mlock);
  }
}

void * th1fun(void * arg)
{
  char str1[80];
  chat str2[40];

  while(1)
  {
    pthread_mutex_lock(&mlock);
    fgets(str1,40,stdin);
    fgets(str2,40,stdin);
    strcat(str1, str2);
    printf("Combined string: %s\n",str1);
    pthread_mutex_unlock(&mlock);
  }
}
```

4. Processes

Introduction

A running program is called a process. Linux supports multi-processing, so kernel allows multiple programs to load and run concurrently. But note that, this concurrent execution is only a psudo concurrency. When computer hardware contains only a single CPU, only one program runs at a time. However because of fast switching between programs, we feel that all programs are running concurrently. But when we have multi-core CPU, then multiple programs (as many as cores) run concurrently.

Every process gets a separate virtual memory in the user space. The executable file of the program is loaded into this virtual memory. For every process kernel creates and maintains a process control block (PCB) structure in the kernel memory space. Kernel keeps all the required information about a process in this PCB.



Following are the important fields in process control block.

state	RUNNING(R), INTERRUPTIBLE SLEEP (S), UNINTERRUPTIBLE SLEEP (D), STOPPED(T), ZOMBIE (Z)
flags	
need resched	
counter	
nice	Priority
Link pointers	Self-referential pointers PCB structures. Used to maintain various process lists and queues such as Process list, Run queue, PID queue, Wait queue. Pointers to parent, child
Pid, Ppid, UId	Process ID and Parent process ID, user ID (owner of process)
tty (tty_struct)	Terminal associated with the process
thread	Context of all the registers

fs (fs_struct)	Current directory and Root directory
files (files_struct)	File descriptor table
mm (mm_struct)	Memory area descriptors describing the virtual memory of process
sigmask_lock	Signals that are masked
signal signal_struct	Signal disposition table
Kernel Stack	While process is executing in the kernel mode, it uses the kernel stack

Linux maintains all the processes (PCBs) in various lists and queues. All the processes are maintained in a list called ‘process list’. All the processes those are ready to run are maintained in a separate queue called ‘run queue’. All the processes those are waiting on an event are kept in event specific queues called wait queues. To efficiently search for a process based on Process ID, all processes are also placed in a hash list.

All processes in Linux are related to each other with parent-child relationships. To maintain these relations, there are four pointers in the PCB. The first pointer points to its parent process. Second pointer points to its child process (if any exist). Third pointer points to next sibling (younger brother/sister) and last pointer points previous sibling (elder sister/brother). l the processes are linked to maintain their parent-child relation ship.

Process ID and Parent Process ID

Kernel assigns every process, a unique ID called process ID or PID. This PID of a process and its parent’s PID (PPID) are stored in the PCB of a process. Linux has got very useful command `‘ps’`, which displays information about all the processes in Linux system. This command takes lot of options. The followingis the output of `‘ps’` command with `‘-ef’` option. You may try `‘ps’` command with other options like `‘-ax’, ‘-aux’, ‘-elf’`. You may look into man page for other options and lot more additional information.

UID	User ID or Owner ID of the process. User ID is integer, but here it is displaying user name associated with the user ID. The file ‘/etc/passwd’ maps user Ids to user names
PID	Process ID of the process
PPID	Parent’s Process ID
C	Percentage of CPU time consumed by this process
STIME	Start time in HH:MM format. Time at which this process is started
TTY	Name of the standard input/output device (Terminal device) for this process. ? means no terminal is associated with this process.
TIME	Amount of CPU time consumed by this process in HH:MM:SS
CMD	Program or executable file name, which is loaded into the process virtual memory, [xxxx] indicates that, this process is a kernel process/thread that runs completely in the kernel mode. No virtual memory is allocated for this kernel process.

```
$ ps -ef

UID         PID  PPID  C STIME TTY          TIME CMD
root          1     0  0 21:59 ?        00:00:00 init [3]
root          2     1  0 21:59 ?        00:00:00 [ksoftirqd/0]
root         29     1  0 21:59 ?        00:00:00 [khubd]
root         40     1  0 21:59 ?        00:00:00 [kswapd0]
root        113     1  0 21:59 ?        00:00:00 [kseriod]
root        187     1  0 21:59 ?        00:00:07 [kjournald]
root       1087     1  0 21:59 ?        00:00:00 udevd
root       1399     1  0 21:59 ?        00:00:01 [kjournald]
root       1400     1  0 21:59 ?        00:00:02 [kjournald]
root       1679     1  0 21:59 ?        00:00:00 syslogd -m 0
root       1683     1  0 21:59 ?        00:00:00 klogd -x
rpc        1709     1  0 21:59 ?        00:00:00 portmap
rpcuser    1728     1  0 21:59 ?        00:00:00 rpc.statd
root       1759     1  0 22:00 ?        00:00:00 rpc.idmapd
root       1797     1  0 22:00 ?        00:00:00 ypserv
root      1831     1  0 22:00 ?        00:00:00 nifd -n
nobody    1860     1  0 22:00 ?        00:00:00 mDNSResponder
root      1871     1  0 22:00 ?        00:00:00 /usr/sbin/smartd
root      1880     1  0 22:00 ?        00:00:00 /usr/sbin/acpid
root      1945     1  0 22:00 ?        00:00:00 /usr/sbin/sshd
root      1955     1  0 22:00 ?        00:00:00 xinetd -stayalive -pidfile
/var/run/xinetd.pid
root      2005     1  0 22:00 ?        00:00:00 rpc.rquotad
root      2009     1  0 22:00 ?        00:00:00 [nfsd]
root      2013     1  0 22:00 ?        00:00:00 [nfsd]
root      2014     1  0 22:00 ?        00:00:00 [nfsd]
root      2015     1  0 22:00 ?        00:00:00 [nfsd]
root      2016     1  0 22:00 ?        00:00:00 [lockd]
root      2017     1  0 22:00 ?        00:00:00 [rpciod]
root      2018     1  0 22:00 ?        00:00:00 [nfsd]
root      2022     1  0 22:00 ?        00:00:00 rpc.mountd
root      2048     1  0 22:00 ?        00:00:00 /usr/sbin/dhcpd
root      2057     1  0 22:00 ?        00:00:00 gpm -m /dev/input/mice -t imps2
root      2066     1  0 22:00 ?        00:00:00 crond
xfs       2086     1  0 22:00 ?        00:00:00 xfs -droppriv -daemon
daemon    2103     1  0 22:00 ?        00:00:00 /usr/sbin/atd
dbus      2112     1  0 22:00 ?        00:00:00 dbus-daemon-1 --system
root      2123     1  0 22:00 ?        00:00:00 hald
root      2131     1  0 22:00 tty2     00:00:00 /sbin/mingetty tty2
root      2132     1  0 22:00 tty3     00:00:00 /sbin/mingetty tty3
root      2133     1  0 22:00 tty4     00:00:00 /sbin/mingetty tty4
root      2134     1  0 22:00 tty5     00:00:00 /sbin/mingetty tty5
root      2135     1  0 22:00 tty6     00:00:00 /sbin/mingetty tty6
root      2777     1  0 22:06 tty1     00:00:00 /sbin/mingetty tty1
root      2805  1945  0 22:47 ?        00:00:00 sshd: karasala [priv]
karasala  2807  2805  0 22:47 ?        00:00:00 sshd: karasala@pts/0
karasala  2808  2807  0 22:47 pts/0    00:00:00 -bash
karasala  2852  2808  0 22:54 pts/0    00:00:00 ps -ef
```

From a program we can get the PID and PPID of a process by using following two system calls.

```
 #include <unistd.h>

pid_t getpid();
pid_t getppid();
```

The following sample program prints the PID and PPID of a process.

File: pid1.c

```
int main()
{
  printf("\nI am a process, my process id is %d\n", getpid());
  printf("My parent's process id is       %d\n\n", getppid());
}
```

Creating a new process

It is possible for a process (a running program) to create another process by calling a system call **'**`fork()`**'.** In fact this is how all the processes (except first process with PID 1) in Linux have got created. The first process (process with PID 1, and runs 'init' program) is created by the Linux kernel during booting time. And this process creates most of the other processes. These other processes in turn creates some other processes. If you observe the list of processes displayed for 'ps –ef' command you will observe that parent process for most of the processes is 1.

fork() System Call

The fork() system call creates a new process. When a process calls fork() system call, control goes to the kernel, and fork() system call inside the kernel will execute. The system call creates a new PCB in the kernel and new virtual memory in the user space. Next it copies all the virtual memory of original process (parent process) into this new process (child process) virtual memory. So after fork, the virtual memory contents of parent and child process are identical. That is both parent and child processes will have identical Text, Data, BSS, Heap and Stack sections. Next kernel copies most of the content of parent process PCB into child process PCB. Only few fields in the child process PCB will be different form the parent process PCB.

Now parent and child processes are two independent processes. And both are going to execute the same program after fork() statement onwards. While entering into fork() system call, only parent process is present, but fork() returns twice. Once in parent process context and next in child process context. The return value of fork() will be different in these two contexts. For the parent the return value of fork() will be PID of child process. For child process, the return value of fork is zero.

```
#include <unistd.h>

pid_t fork(void);
```

When you execute the following program, you will find that, the statement after fork will execute twice. From the output of this statement, you can makeout that this statement is executed once by parent and once by child process.

File: fork1.c

```
int main()
{
  printf("\nI am a process my process id is %d\n", getpid());
  printf("My parent's process id is       %d\n", getppid());
  printf("Now i am going to create a child process by calling fork\n\n");
  fork();
  printf("My PID : %d, My parent PID : %d\n",getpid(), getppid());
}
```

Both parent and chid processes executing same code is not much meaningful. We normally create a child process to do some other work and parent to do something else. But both parts of code are in the same program. We make parent to execute some portion of the program and child to execute some other portion of the program. The followng program just does that. We are using the return value of fork() to distinguish between parent process and child process. For child process the return value of fork will be zero, and for parent process the return value of fork will be process Id of the child. So let us rewrite the above program.

File: fork2.c

```
int main()
{
  int id;
  int cnt = 0;
  printf(" I am going to create a new process\n");
  id = fork();
  if(id)
  {
    while(cnt < 10)
    {
      cnt++;
      printf("I am Parent, my child process is is %d\n", id);
      sleep(1);
    }
  }
  else
  {
    while(cnt < 20)
    {
      cnt++;
      printf("I am child process I got id value as %d\n", id);
      sleep(1);
    }
  }
}
```

The above program when executed produces the following output:

```
I am going to create a new process
I am Parent, my child process is 2991
I am child process I got id value as 0
```

Now we are able to make both parent and child processes to execute different portions of the same executable file based on the return value of a fork.

Termination of a Process

A process terminates whenever main function returns or whenever exit() system call is invoked. A process also terminates when it receives certain signals (we will cover signals in next session).

Exit status of a Process

Whenever a process terminates it should give the exit status to its parent process. Till its parent process reads the exit status, the child process will not be removed completely. The processes, which are terminated, but still waiting for their parents to read their exit status, are in Zombie State.

Waiting for a exist status of a child process

A parent process can wait for the completion of its child processes. So a process can create one or more child processes and can wait for their completion. Shell program is the best example for this. Shell creates a child process or multiple child processes to run multiple interconnected (pipe) commands and waits for their completion. Following are the system calls for waiting for the child and as well as to get the exit status of child process.

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

pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

Both wait() and waitpid() returns process ID if OK, -1 on error.

The wait() blocks till any child process terminates. By the time wait() is called, if any child is already terminated, then wait() returns immediately with terminated child status.

The waitpid() system call provides better options than wait() system call. First waitpid() provides non blocking option. When non-blocking option is used, waitpid() returns when no child is terminated. Other option is that, waitpid() can be used to block for a particular child or a group of children. The argument pid to waitpid()

`pid == -1`	Waits for any child process. For this option `waitpid` is equivalent to `wait`.
`pid > 0`	Waits for the child whose process ID equals `pid`.
`pid == 0`	Waits for any child whose process group ID equals that of the calling process.
`pid < -1`	Waits for any child whose process group ID equals the absolute value `pid`.

Orphan process

Typically parent waits for the termination of a child process by using wait() or waitpid() calls described above. However if parent terminates without waiting for the termination of a child, then the child process will become orphan process. All orphan processes are attached to the 'init' process, which is the root of all the processes. So init process will become new parent for these orphan processes.

Zombie state

When a child process terminates and parent process does not call wait() or waitpid() functions, then child process enters into a state called zombie state. The child process frees almost all the resources and simple waits in this state till parent calls wait() functions. If parent terminates without calling wait() function, then this zombie state child is attached to Init process. Init process is always waits for termination of its children, so it calls wait() function and zombie child will terminate fully.

The following program, shows application of fork() and wait() system calls. In this program parent process creates two child processes and makes each child process to do some work. Next parent process waits till both the children have done their work.

`File: fork3.c`

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

int main()
{
  pid_t chpid, child1, child2;
  int ii,stat;

  printf("\nI am a process my process id is %d\n", getpid());
  printf("I am creating two child proceses & make them run some code\n");

  child1 = fork();
  if(child1==0)
  {
    printf("I am first child, my pid is %d\n",getpid());
    printf("I first child going to execute following loop for 6 times\n");
    for(ii=0; ii<6; ii++)
    {
      printf("\nI am child 1, this is iteration %d of loop \n",ii+1);
      printf("I am tired, sleeping for 10 seconds\n\n");
      sleep(10);
    }
    exit(2);
  }

  child2 = fork();
  if(child2==0)
  {
    printf("I am second child, my pid is %d\n",getpid());
    printf("I second child going to execute following loop for 8 times\n");
    for(ii=0; ii<8; ii++)
    {
      printf("\nI am child 2, this is iteration %d of loop \n",ii+1);
      printf("I am tired, sleeping for 5 seconds\n\n");
      sleep(5);
    }
    exit(3);
  }

  printf("I am parent, my children are working & sleeping, i wait for them\n");
  chpid = wait(&stat);
  if(chpid == child1)
    printf("My first child terminated with status %d\n",WEXITSTATUS(stat));
  if(chpid == child2)
    printf("My second child terminated with status %d\n", WEXITSTATUS(stat));

  chpid = wait(&stat);
  if(chpid == child1)
    printf("My first child terminated with status %d\n", WEXITSTATUS(stat));
  if(chpid == child2)
    printf("My second child terminated with status %d\n", WEXITSTATUS(stat));

  printf("\nBoth children got terminated, now i will do the same\n\n");
}
```

The following program is very useful to do some experiments. Following are some experiments.

Printing the variable in parent process, which is incremented by the child process.
Observing the PPID of child process when parent of child process is terminated.
Allow child process to terminate first, and observe the state of terminated child process.
Now make parent to call wait() call so that terminated child process is fully terminated.

File: fork4.c

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

int cnt;

int main()
{
  char ch;
  int exitstat;
  int retval;

  printf("I am parent,my PID is %d\n",getpid());

  if(fork()==0)
  {
    while(1)
    {
      printf("I am child: my PID and PPID are %d and %d\n",getpid(),getppid());
      sleep(10);
      cnt++;
      if(cnt == 10)
        exit(25);
    }
  }
  else
  {
    while(1)
    {
      ch = getchar();
      if(ch=='c')
        printf("cnt = %d\n",cnt);
      if(ch=='x')
        exit(2);
      if(ch=='r')
      {
        retval = wait(&exitstat);
        printf("exit status = %d\n", WEXITSTATUS(exitstat));
        printf("retval, that is PID of child = %d\n", retval);
      }
    }
  }
}
```

exec System call

The exec() system call is useful to load and execute a new program by a process. A running program (process) can call exec() system call to execute a new executable file by the same process. The process ID remains same. The executable file is loaded into the virtual memory of current process and starts executing from the beginning. Once exec() is successful, no instructions after exec() statement will be executed. Because this program is over written with a new program.

Following are the different flavors of exec system calls.

```
int execl(const char *pathname, const char *arg0, ... /* (char *)0 */);
int execv(const char *pathname, char *const argv[]);
int execlp(const char *filename, const char *arg0, ... /*(char *) 0*/);
int execvp(const char *filename, char *const argv[]);
```

The following program shows the basic usage of execl() system call. The execl() system call takes executable file name as first parameter. The other parameters to execl() are command line argument strings. The last string should be a NULL pointer indicating last string.

File: exec1.c

```
int main()
{
  printf("I am going to exec an 'ls' program\n");
  execl("/bin/ls","ls",0);
  printf("I executed 'ls' program;
}
```

The execv() system call is similar to execl(), only difference is in the passing of command line argument strings. The execl() takes each command argument string as a separate parameter. Where as execv() takes command argument strings array, as single parameter. Following program shows the usage of execv() system call. Note that the last command string in the array should be NULL pointer, indicating no more arguments in the array.

File: exec2.c

```
int main()
{
  char *args[2];


  printf("I am going to exec an 'ls' program in current process\n");

  args[0] = "ls";
  args[1] = 0;
  execv("/bin/ls", args);

  printf("I execed 'ls' program\n");
}
```

The execlp() is identical to execl(). But for execl() we must given full pathname of executable file, where as for execlp() only executable file name is sufficient. Observe this difference in the following program, which is using execlp().

exec3.c

```
int main()
{
  printf("I am going to exec an 'ls' program in current process\n");
  execlp("ls","ls",0);
  printf("I execed 'ls' program\n");
}
```

So far we are running, execxx() function in the same process. But one very common and useful way to run execxx() is in a child process. A parent process, whenever it wants to run another program, it creates a child process and in the child process it execs the new executable file. The followng program shows this usage of execv() sytem call. Parent process waits till child completes the execution.

File: exec4.c

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

int main()
{
  pid_t pid;
  int option, stat;

  while(1)
  {
    printf("\nEnter 1 to exec 'ls' program in child process and 0 to exit\n");
    scanf("%d",&option);
    if(!option)
      exit(0);

    printf("\n");
    if(fork()==0)
      execl("/bin/ls", "ls", 0);

    pid = wait(&stat);
    printf("\nChild with pid %d is terminated with exit status %d\n",pid,
                                                       WEXITSTATUS(stat));
  }
}
```

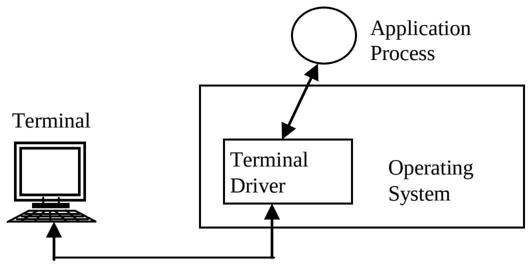
The following program shows the application of fork(), execv() and wait() system calls in implementing our own shell program. This shell is not allowing any command line arguments. You can extent this shell program to take command line arguments also.

File: exec5.c

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

int main()
{
  pid_t pid;
  int option, stat;
  char cmdbuf[80];
  char *cmdargs[4];

  while(1)
  {
    printf("myshell> ");
    gets(cmdbuf);
    cmdargs[0] = cmdbuf;
    cmdargs[1] = 0;

    if(strncmp(cmdbuf,"ver",3)==0)
    {
      printf("Simple shell version 1.00, Sep 25, 2008\n");
      continue;
    }
    if(strncmp(cmdbuf,"quit",4)==0)
      exit(0);

    if(fork()==0)
    {
      execvp(cmdbuf, cmdargs);
      exit(0);
    }

    pid = wait(&stat);
  }
}
```

5. Terminals

To work on any Unix or Linux system, user needs input/output device called terminal. The terminal contains a keyboard as input device and a screen (on monitor device) as output device. The terminal driver in operating system is responsible for interacting with the terminal device.



These terminal devices can be connected to Linux system in three possible ways.

- As a console (connected directly to a system through graphic controller/adapter)
- Dumb terminal through RS-232 serial cables
- As a network terminal (pseudo terminal) (telnet session)

Directly attached console

In the case of console, both keyboard and Display monitor are directly connected to the system. Typically only one console can be connected to one system. However each console can support multiple logical terminals called virtual terminals. These virtual terminals are useful to run multiple login sessions simultaneously. Typically all Linux supports six virtual terminals on each console device. You can switch from one virtual terminal to other easily, but you can use only one at a time.

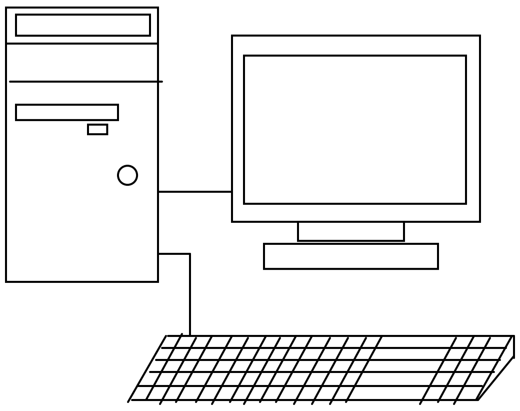


Figure: Console connected directly to Linux System

Dumb terminal through serial port

Traditional Unix systems around 15 or 20 years' back used to have only system unit. This system unit is connected to 8 or 16 dumb terminals through RS-232 serial ports. This dumb terminal contains Video Display Unit (VDU) and keyboard. The dumb terminal just acts as an input output device. What ever typed on the keyboard is sent on the serial port. And what ever received on the serial port is displayed on the VDU.

Dumb terminals are still useful to connect to embedded systems or Linux/Unix systems whose hardware does not contain any support for directly attached monitors are keyboards. For example some embedded systems may not have any attached keyboard or display monitor. But still they are capable of running Linux. In such cases Linux will use serial port and dumb terminal. We can use computer itself as dumb terminal by executing a terminal emulator software. In Linux 'minicom' is the terminal emulator software. In windows hyper-terminal is the terminal emulator software and it comes with windows by default.

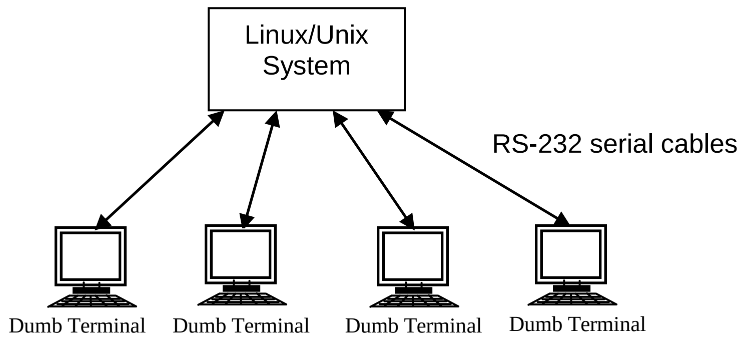
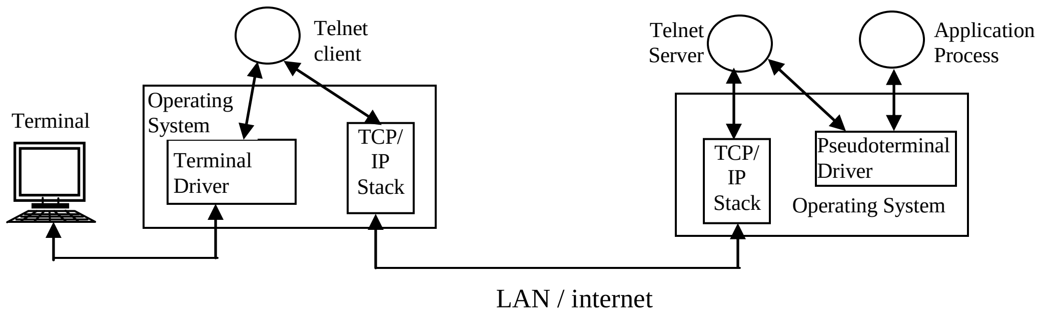


Figure: Dumb terminal connected to Linux System through serial cables

Telnet session through network

It is also possible to connect to a Linux system through a network terminal. When we connect to a Linux through telnet session it is a network terminal. Telnet client simulates a terminal. This terminal is connected to the Linux through network connection. For a Linux system, users logged in through network (telnet) appear as connected to pseudo terminals. The pseudo terminal simulates a real terminal by establishing connection with the telnet client.

It is also possible that Linux can simultaneously supports user logins from console, dumb terminals connected through serial ports and network terminals.



Linux booting sequence

When Linux kernel is booted, it creates process with process ID 1, and it execs 'init' program. This is called `init` process. This process brings the system into multiuser and starts all the server application (called daemons in unix/linux). This init process first reads the file `'/etc/inittab'`. This file contains the list of terminals connected to the system. In linux typically there are six virtual terminals specified in this file. For every terminal specified, `init` forks a new process and execs `mingetty` program.

The mingetty program opens the terminal and displays '`login:'`. This is the prompt what you see on each virtual terminal on power on. When we type our username, the mingetty program execs the 'login' program. This login program asks the password. Once you entered password, the login program verifies the password. If passworkd is correct it forks a new process and execs shell program.

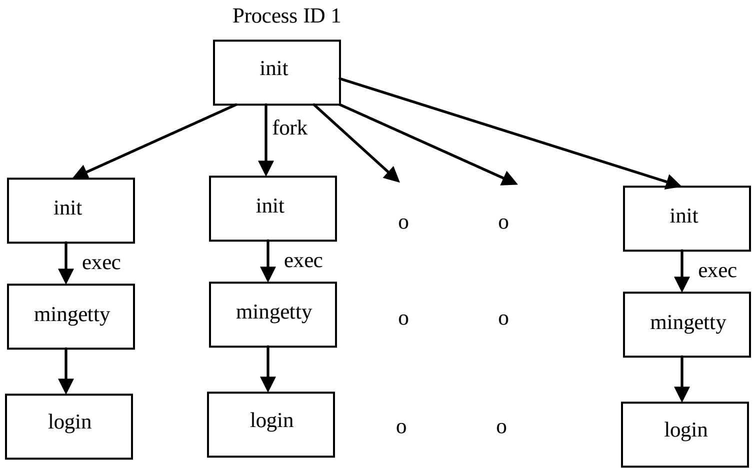


Figure: State of processes after login has been invoked

Process Groups

In addition to having a process ID, each process also belongs to a process group. A process group is a collection of one or more processes. Each process group has a unique process group ID. Process group Ids are similar to process IDs – they are positive integers and they can be stored in a pid_t data type. The following system call returns the process group ID of the calling process.

```
pid_t getpgrp(void);
```

Each process group can have a process group leader. The leader is identified by having its process group ID equals its process ID.

Sessions

A session is a collection of one or more processes. A process establishes a new session by calling the `setsid` system call.

```
pid_t setsid(void);
```

If the calling process is not a process group leader, this function creates a new session. When new session is created following three things will happen:

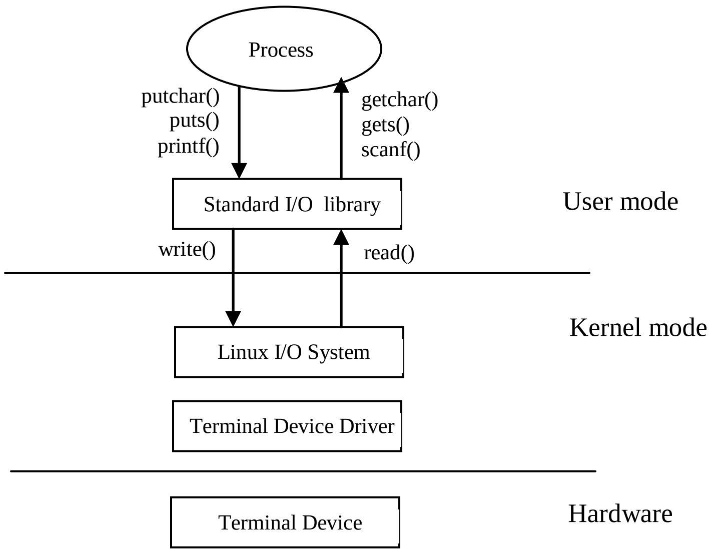
1. The process becomes the session leader of this new session. (A session leader is the process that creates a session). The process is the only process in this new session.

2. The process becomes the process group leader of new process group. The new process group ID is the process ID of the calling process.

3. The process has no controlling terminal. If the process had a controlling terminal before calling setsid, that association is broken. (A session can have a single controlling terminal).

Terminal I/O

Terminal is the interface between user and a process on the Linux. In the previous section we covered different types of terminals and how they are connected to Linux. When we read or write to terminals it goes through various layers as shown below.



When user program reads or writes data to terminal, it calls standard I/O functions like `putchar()` or `getchar()`. These library functions in turn calls system calls like `read()` and `write()`. These system calls are implemented as part of kernel's I/O system. These `read()` and `write()` in turn calls the terminal device driver calls.

This terminal device driver implements lot of line editing functionality, which is useful for many application programs. For example if an application wants read a line of text it uses gets() or fgets() function. As user enters the line of text, it get displayed on the screen, but it will not go to the application till 'Enter' character is pressed. This allows user to edit or modify the line of text being given. If terminal driver does not provide this functionality, then every application has to provide line-editing facilities in their programs.

But for some application programs, we must not have this driver functionality. For example if our application is accepting a password from the user, then password should not get displayed on the screen. If we are developing terminal emulator program, then as soon as character is entered it should be received by the application. With terminal driver, we will not receive individual characters till ‘Enter’ character is given. If we are developing editor program like ‘vi’ then all editing characters should be passed on to the editor program without being consumed by terminal driver.

So there is a clear need to modify the functionality of terminal driver for some applications. Linux provides following system calls to modify the terminal characteristics or terminal attributes.

```
#include <termios.h>

int tcgetattr(int filedes, struct termios *termptr);
int tcsetattr(int filedes, int opt, const struct termios *termptr);
```

The struct termios is defined as below in termios.h file.

```
struct termios
{
  tcflag_t  c_iflag;     // input flags
  tcflag_t  c_oflag;     // output flags
  tcflag_t  c_cflag;     // control flags
  tcflag_t  c_lflag;     // local flags
  cc_t      c_cc[NCCS]; // control characters
};
```

The input flags control the input of characters by the terminal device driver (strip eighth bit on input, enable input parity checking etc.). The output flags control the driver output(map newline to CR/LF, expand tabs to spaces etc.). The control flags affect the RS-232 serial lines. The local flags affect the interface between the driver and the user (echo on or off, canonical or non-canonical mode, etc.).

Even though there are only five fields in the termios structure, each field holds lot of option flags. For example input flags contain nearly 12 options, output flags contain 16 options and so on. Refer to Figure 11.3 [1].

So it is not easy to set all the options of this termios structure. So the simple way of changing terminal attributes is to read the current attributes of the terminal and modify only the options we would like to change and write this modified attributes to the terminal.

The `tcgetattr()` is used to get the current attributes of a terminal. When this function is called, it fills the termios structure pointed by the `termptr` argument.

The `tcsetattr()` is used to set the new attributes to a terminal. The `termptr` argument points to the `termios` structure that contains the new attributes to set.

The argument opt allows us to specify when the new terminal attributes to take effect. Opt is specified as one of the following constants:

`TCSANOW`	The change occurs immediately
`TCSADRAIN`	The change occurs after all output has been transmitted. This option should be used if we are changing the output parameters.
`TCSAFLUSH`	The change occurs after all output has been transmitted. Further more when the change takes place, all input data that has not been read is discarded (flushed).

The following program shows the usage of `tcsetattr()` and `tcgetattr()` functions for accepting password in an invisible way.

File: password.c

```
#include <stdio.h>
#include <termios.h>
#include <fcntl.h>

int getPasswd(char *passwd)
{
  int ii=0;
  struct termios  cuset, newset;
  char ch;

  /*** Disable echo and canonical mode of processing ***/
  tcgetattr(0,&cuset);
  newset = cuset;
  newset.c_lflag &= ~ICANON;
  newset.c_lflag &= ~ECHO;
  tcsetattr(0,TCSANOW,&newset);

  setbuf(stdin,NULL);
  setbuf(stdout,NULL);

  while((ch=getchar()) != '\n')
  {
    passwd[ii++] = ch;
    putchar('*');
  }
  passwd[ii++] = 0;
  putchar('\n');
  tcsetattr(0,TCSANOW,&cuset);
}

int main()
{
  char pw[20];

  printf("Enter password : ");
  getPasswd(pw);
  printf("Your password is %s\n", pw);
}
```

6. **Signals**

What are signals?

Signals are the events to a process that are sent by a kernel.

When kernel sends signal to a process?

Kernel sends the signals (events) to processes because of following reasons.

- When a process performance some illegal operation
- When user enters some special key at the terminal or when terminal disconnects
- When another process requests the kernel to send a signal
- When some events which are important to a process occurs

Let us discuss each of these reasons:

Process while executing a program it may perform some illegal operations like: dividing by zero, accessing a memory location outside process address space, accessing an integer at unaligned address, executing an invalid instruction, executing a privileged instruction etc. Whenever process performs these illegal operations, CPU will generate an exception or trap. This causes CPU to execute exception handler functions of kernel. In these function kernel identify the process that caused exception and sends a corresponding signal to that process.

When user enters some special keys like Control-C or Control-Z, the terminal driver (kernel) detects these special keys and generates signal to the process.

A process can request the kernel to send a signal to other process (If it got permission) by using kill() function. Then kernel will send the specified signal to the specified process.

Kernel also sends signal to a process on some software conditions like, child process terminating, expiration of alarm, writing to a pipe when reader has terminated etc..

How process handles signals?

Every process control block (PCB) maintains information on how to handle each of the signals. This is called disposition of signals. So each process can have its own signal disposition table. The signal disposition can specify one of the following three things:

Ignore the signal	This works for most signal, but there are two signals that can never be ignored. These are SIGKILL and SIGSTOP. The reason these two signals can't be ignored is to provide the super user with a sure way of either killing or stopping a process.
Execute a signal handler	For this we need to give the address of a signal handler function. When signal occurs, process will execute this handler function. Installing a signal handler function is called *'catching a signal'.*
Take default Action	Every signal has default action. Default action for most of the action is to terminate the process. For some signals default action could be ignoring a signal.

By default, all signal dispositions of a process are set to take default action.

There are 31 different types of signals. These 31 signals are predefined, each signal is having a name indicating its purpose and each signal is also associated with a number. Every signal name starts with three letters 'SIG'. Following are the names of some important signals. Refer section 9.2 [1] for complete list of signals.

Name	Description	Default action
SIGABRT	This signal is generated by calling the `abort()` function. The process terminates abnormally.	terminate w/core
SIGALRM	This signal is generated when a timer that we've set with the `alarm()` function expires.	terminate
SIGCHLD	Whenever a process terminates or stops, the SIGCHLD signal is sent to the parent.	ignore
SIGFPE	This signals an arithmetic exception, such as divide-by-0, floating point overflow, and so on.	terminate w/core
SIGINT	This signal is generated by the terminal driver when we type the interrupt key (Control-C).	terminate
SIGKILL	This signal is one of the two that can't be caught	terminate
SIGPIPE	If we write to a pipe but the reader has terminated, this signal is generated.	terminate
SIGTERM	This is the termination signal sent by the kill(1) command by default.	terminate
SIGUSR1	This is a user-defined signal, for use in application programs.	terminate
SIGUSR2	This is a user-defined signal, for use in application programs.	terminate

Changing disposition of a signal

So far in all the programs written, we never bothered about handling signals. But every non-trivial application needs to handle signals. How a process will respond (what action process takes) when a signal is received depends on how its signal dispositions are set. We can change disposition of a signal using `signal()` system call.

`signal` Function

```
void (* signal (int signo, void (*func) (int))) (int);

typedef void sigFunc(int);
sigFunc *signal (int, sigFunc *);
```

Following is the sample program that shows the handling of a SIGINT (Control-c) signal.

`File: sig1.c`

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

void mysigHandler(int signo)
{
  printf("I caught the signal %d\n",signo);
}
```

```
int main()
{
  signal(SIGINT, mysigHandler);
  while(1)
  {
    sleep(15);
    printf("I slept for 15 seconds\n");
  }
}
```

`kill and raise` Functions

The *`kill`* function sends a signal to a process or group of processes. The *`raise`* function allows a process to send a signal to itself.

```
int kill (pid_t pid, int signo);
```

Following are the different conditions for the *`pid`* argument to *`kill.`*

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

The following *`raise`* function allows the process to send a signal to itself.

```
int raise (int signo);
```

`alarm and pause` Functions

The *`alarm`* function allows us 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.

```
unsigned int alarm (unsigned int seconds);
```

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

```
int pause(void);
```

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.

`File: sig2.c`

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

```
int sgint, sgusr1, sgusr2, sgalrm, sgothers;

void sigIntHandler(int signo)
{
  sgint++;
}

void genericHandler(int signo)
{
  switch(signo)
  {
    case SIGUSR1:
      sgusr1++;
      break;
    case SIGUSR2:
      sgusr2++;
      break;
    default:
      sgothers++;
      break;
  }
}

void sigAlarmHandler(int signo)
{
  sgalrm++;
  alarm(5);
}

int main()
{

  signal(SIGINT, sigIntHandler);
  signal(SIGUSR1, genericHandler);
  signal(SIGUSR2, genericHandler);
  signal(SIGALRM, sigAlarmHandler);

  alarm(5);
  while(1)
  {
    pause();
    printf("INT, USR1, USR2, ALARM, OTHER : %d, %d, %d, %d, %d\n",
            sgint,sgusr1,sgusr2,sgalrm,sgothers);
  }
}
```

Signal Sets and `sigset_t` data type

We need a data type to represent multiple signals – `a signal set`. We need to pass a signal set to various functions. A data type sigset_t is available to define a signal set and the following five functions to manipulate signal sets.

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset (sigset_t *set);
int sigaddset  (sigset_t *set, int signo);
int sigdelset  (sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
```

Blocking Signals

Every process is associated with a signal mask. The signal mask of a process is the set of signals currently blocked from delivery to that process. We can think of this mask as having one bit for each possible signal. If the bit is on for a given signal, that signal is currently blocked. A process can examine or change (or both) its signal mask by calling the following function.

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

If *`oset`* parameter is non-null pointer, then function returns the current signal mask through *`oset`*. If *`set`* is non-null pointer, then *`how`* argument indicates how the current signal mask is modified. Following are the values of how and their meaning.

SIG_BLOCK	The new signal mask for the process is OR of its current signal mask and the signal set pointed by *`set`*. That is *`set`* contains the additional signals that we want to block.
SIG_UNBLOCK	The new signal mask for the process is the AND of its current signal mask and the complement of the signal set pointed to by set. That is *`set`* contains the signals that we want to unblock.
SIG_SETMASK	The new signal mask for the process is the value pointed by set.

Pending Signals

The sigpending() returns the set of signals that are blocked from delivery and currently pending for the calling process. The set of signals is returned through the set argument.

```
int sigpending(sigset_t *set);
```

Signals and Blocking system calls

The 'System calls', which may put a process in sleep or blocking state, are called blocking system calls. Best examples of blocking system calls are read() and write() system calls. The read() call to an I/O device will put a process in sleep state when no data is available with the device. Similarly write() call will put a process in sleep state, when I/O device is not accepting any more data.

If a signal is received by a process while it is in sleep state because of a blocking system call, what will happen?

Even though the process is in sleep state, still the signal handler will get executed. Next there are two possibilities. One is the system call still continues in the blocking state, as if nothing happened. Second is the system call return with an error. Depending on the application requirement we can chose one of these behaviors.

```
#include <signal.h>
int sigaction(int signo, const struct sigaction *act,
                               struct sigaction *oact);
```

The argument *signo* is the signal number whose action we are examining or modifying. The *act* is pointer to the sigaction structure, which contains the new disposition for the signal. The oact is pointer to the sigaction structure, in which the old disposition of the signal is returned. Following is the sigaction structure.

```
struct sigaction
{
  void      (*sa_handler) ()   /* addr of signal handler, or SIG_IGN, or SIG_DFL */
  sigset_t  sa_mask;           /* additional signals to block */
  int       sa_flags;          /* signal options */
};
```

When changing the action for a signal, if the sa_handler points to a signal catching function then the sa_mask field specifies a set of signals that are added to the signal mask of the process before the signal-catching function is called. When the signal-catching function returns, the signal mask of the process is reset to its previous value. This we are able to block certain signals whenever a signal handler is invoked.

This new signal mask that is installed by the system when the signal handler is invoked automatically includes the signal being delivered. Hence, we are guaranteed that whenever we are processing a given signal, another occurrence of that same signal is blocked until we are finished processing the first occurrence.

Signal delivery mechanism

Review Questions

What are the various reasons for a kernel to send a signal to a process?
When kernel sends a signal to a process, what are the various possibilities that could happen?
What are pending signals to a process?
Is it possible to read the pending signals of a process, if so how?
What could be the one reason for a signal to be pending for a long time?
Could you explain how alarm signal can be used to implement a timeout while taking user input.
What are the various macros used to read or write to sigset_t structure?
How do you mask or block certain signals?
Is it possible to find the signals currently blocked by a process?
What are the advantages of using sigaction() system call over signal() system call
How exactly a signal is delivered to a process by the kernel?
What is blocking system call?
What is the behavior of a blocking system call when a signal is received?

7. Inter Process Communication (IPC)

Inter process communication is all about exchanging messages or data between processes. Most of the large application programs are developed as multi-process or multi-threaded (we will discuss threads in later sections) applications. All these processes (or threads) of the application should co-ordinate among themselves. This co-ordination among processes involves following three things:

- Communication
- Synchronization
- Mutual exclusion

First we will study communication between processes. The synchronization and mutual exclusion between processes will be covered while studying semaphores.

Characteristics of Inter process communication

In linux we have different methods for doing inter process communication. Before studying them, let us see common characteristics or features of communications. Understanding of these characteristics allows us the compare various types of IPC.

- Connection Oriented or Connection less
- Message oriented or Stream oriented
- Half duplex or Full duplex
- Name or identifier of communication object

In the case of connection less, message oriented service, a process can send data without caring for the connection between two processes. So sender can send message, even if no receiver is present. This we call connection less mechanism.

In the message oriented communication data transfers between processes will happen in units of messages only. Each write operation involves sending one message and each read operation involved receiving one message. It is not possible to read half message at a time or more than one message at a time. Even if we read half of message, the remaining half of the message will be lost. So in message oriented communication, message boundaries are maintained.

In the case of connection oriented service, writing or reading is possible only after connection is established between two processes. That means both processes should open the connection before reading or writing.

Traditional Unix is to offer only two forms of IPC. These are:

- Pipes
- Named pipes OR FIFOs

Later AT&T Unix implemented following three new types of IPC mechanism in their System V version of Unix. That’s why these are called System V IPC. Now all most all Unix including Linux will support System V IPC.

- Message Queues
- Semaphores
- Shared Memory

The Posix standard requires a new set of IPC objects called Posix message queues, Posix shared memory and Posix semaphores. We will study Posix semaphores along with Posix threads in next section.

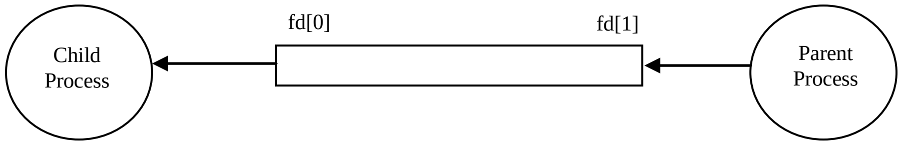
Pipes

Pipe is a communication object, that can be created using `pipe()` system call. When pipe is created it gives us two file descriptors. The first descriptor is opened for reading and second descriptor is opened for writing. Now this process (who created the pipe) and another process that got access to this file descriptors can communicate to each other through this pipe.

Only child processes can access the pipe created by parent, this is because only child processes can access file descriptors created by parent. So one important limitation of pipe is that, it can only be used between related processes (like parent and child). Another limitation of pipe is that it is half-duplex or uni-directional. If we need bi-directional communication then we need to create two pipes.

Pipe offers stream type of communication facilities. Pipe does not associate with any global name, so only related processes can use it.

```
#include <unistd.h>

int pipe(int filedes[2]);
```

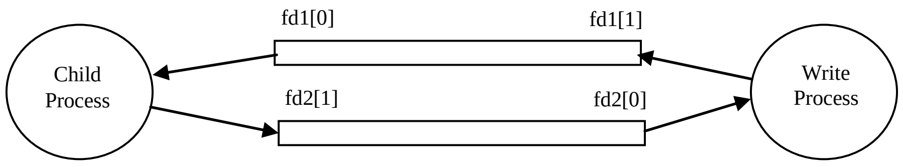


File pipe1.c

```
#include <stdio.h>

int main()
{
  int fds[2],ii,n;
  char readbuf[100];
  char writebuf[100];

  if(pipe(fds) < 0)
  {
    perror("pipe creation failed");
    exit(1);
  }

  if(fork()==0) // child process
  {
    close(fds[1]);
    while(1)
    {
      n = read(fds[0],readbuf,100);
      for(ii=0; ii<n; ii++)
        readbuf[ii] = toupper(readbuf[ii]);
      puts(readbuf);
    }
  }

  // Parent process
  close(fds[0]);
  while(1)
  {
    fgets(writebuf,100,stdin);
    write(fds[1],writebuf,strlen(writebuf)+1);
  }
 }
```

When parent process creates a pipe, two descriptors fd[0] and fd[1] are created. Now parent process forks a child process. Now child process also contains fd[0] and fd[1] in its image. The parent process closes fd[0] and writes to fd[1]. The child process closes fd[1] and reads from fd[0]. This is illustrated in the above figure and program.

If we need full duplex communications then parent needs to create two pipes, so it gets four descriptors fd1[0], fd1[1], fd2[0] and fd2[1]. This is illustrated below.



File pipe2.c

```
#include <stdio.h>

int main()
{
  int pfds1[2],ii,n;
  int pfds2[2];
  char readbuf[100];
  char writebuf[100];

  if(pipe(pfds1) < 0)
  {
    perror("pipe creation failed");
    exit(1);
  }

  if(pipe(pfds2) < 0)
  {
    perror("pipe creation failed");
    exit(1);
  }

  if(fork()==0) // child process
  {
    close(pfds1[1]);
    close(pfds2[0]);
    while(1)
    {
      n = read(pfds1[0],readbuf,100);
      for(ii=0; ii<n; ii++)
      {
        if(isupper(readbuf[ii]))
          readbuf[ii] = tolower(readbuf[ii]);
        else
          readbuf[ii] = toupper(readbuf[ii]);
      }
      write(pfds2[1],readbuf,n);
    }
  }

  // Parent process
  close(pfds1[0]);
  close(pfds2[1]);
  while(1)
  {
    fgets(writebuf,100,stdin);
    write(pfds1[1],writebuf,strlen(writebuf)+1);
    read(pfds2[0],writebuf,100);
    puts(writebuf);
  }
}
```

FIFOs or Named Pipes

One major limitation of Pipes is that, they can be used only between related processes like parent and children. But FIFOs (also called named pipes) eliminate this major limitation of pipes. FIFO can be accessed by any process, because it got a name and appears as a file in the file system. So any process can open a FIFO if it got access permissions.

Similar to Pipe, FIFO also provides stream-based communication. We can consider FIFO as a connection oriented mechanism. This is because, opening of FIFO for read or write from one side will not be successful till other end is opened for write or read respectively.

Fifo is created using following system call. Once created it can be opened similar to file using open() system call. Next read() and write() system calls can be used similar file or pipe.

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

int mkfifo(const char *pathname, mode_t mode);
```

Client – Server program using FIFO

The following program illustrates the use of FIFO between client and server processes. Two separate FIFOs are used for this purpose. The server program opens (creates if required) a server FIFO in reading mode. Once opened, it reads the data from FIFO and toggles the case of all characters in the read data. Finally it writes the toggled data to a client FIFO after opening client FIFO in write mode.

The client program opens the server FIFO for write mode and takes a message from the user and writes that message to server FIFO. Next reads the response message from client FIFO and displays the message to the user.

File fifosrv.c

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


#define     BUF_LEN          256
void   toggleCase(char *buf, int cnt);

int main()
{
  int  srvFd;
  int  cliFd;
  char buf[BUF_LEN];
  int  cnt;

  while(1)
  {
    printf("Waiting for connecting to client\n");
    srvFd = open("srvfifo",O_RDONLY);
    if(srvFd < 0)
    {
      if( mkfifo("srvfifo",0600) < 0)
      {
        printf("Error in creating FIFO\n");
        return(1);
      }
      else
      {
        printf("Created a FIFO\n");
        srvFd = open("srvfifo",O_RDONLY);
      }
    }

    printf("Connected to client through FIFO\n");

    while(1)
    {
      cnt = read(srvFd,buf,BUF_LEN);
      if(cnt == 0)
        break;

      printf("Received message\n");
      toggleCase(buf,cnt);
      cliFd = open("clififo",O_WRONLY);
      if(cliFd)
      {
        printf("Sent response message\n");
        write(cliFd,buf,cnt);
        close(cliFd);
      }
      else
      {
        printf("Could not open client fifo\n");
      }
    }
    close(srvFd);
  }
}


void toggleCase(char *buf, int cnt)
{
  int ii;

  for(ii=0; ii<cnt; ii++)
  {
    if((buf[ii] >= 'A')  && (buf[ii] <= 'Z'))
      buf[ii] += 0x20;
    else if((buf[ii] >= 'a')  && (buf[ii] <= 'z'))
      buf[ii] -= 0x20;
  }
}
```

File: fifocli.c

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


#define     BUF_LEN          256


int main()
{
  int  srvFd;
  int  cliFd;
  char txmsg[BUF_LEN];
  char rxmsg[BUF_LEN];
  int  cnt;

  srvFd = open("srvfifo",O_WRONLY);
  if(!srvFd)
  {
    printf("Error in opening server FIFO\n");
    return(1);
  }
  printf("Connected to server\n");
  printf("Enter some message to send to server\n");
  fgets(txmsg,BUF_LEN,stdin);

  write(srvFd,txmsg,strlen(txmsg)+1);
  printf("Sent given message to server\n");

  cliFd = open("clififo",O_RDONLY);
  if(cliFd < 0)
  {
    if( mkfifo("clififo",0600) < 0)
    {
      printf("Error in creating client FIFO\n");
      return(1);
    }
    else
    {
      cliFd = open("clififo",O_RDONLY);
      printf("Created client FIFO\n");
    }
  }
  printf("Waiting for response message from server\n");
  cnt = read(cliFd,rxmsg,BUF_LEN);
  puts(rxmsg);
  close(srvFd);
  close(cliFd);
}
```

System V IPC

The system V IPC objects are not integrated with I/O system. So we can not use standard read() and write() system calls on these System V IPC objects. System V IPC contains the following objects:

- Message queues
- Semaphores
- Shared memory

We need to create these objects first before using them. When we create we will get an identifier. This identifier can be used to perform operations on that object. If any other process knows this identifier, then it can also perform operations on this object.

But to allow easy access by all processes, each System V objects can be associated with some 'key' value. So that all other processes that try to access the same object, can open that object by using that key.

Message Queues

Message queues offer message oriented connection less communication between processes. Basic unit of exchange through message queue is a message. We can send messages to message queue even if no other process is waiting for messages.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int flag);
int msgctl (int msqid, int cmd, struct msqid_ds *buf);
int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
int msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
```

The `msgget()` is to create a new message queue or to open an existing message queue.

```
int msqId = msgget(12345, IPC_CREAT | 0666);

int msqId = msgget(12345, 0);
```

The first one creates new message queue with key as 12345 and permission as 0666. The second one opens the existing message queue with key 12345. Both functions return the identifier for the message queue.

Client – Server program using message queues

The following program illustrates the use of Message Queues for IPC between client and server processes. The server program creates a message queue. And reads the message from this queue using msgrcv() system call. The server specifies the type of the message as 1. Once it receives the message it toggles the case of characters in the message. The toggled message written back to the message queue with message type as given by the client.

The client program opens the same message queue by using the same KEY as used by the server program. The client program takes the message from the user and writes the message to the message queue. Client specifies the message type as 1, so that it could be read by the server. In the data portion of the message, client places its process ID. This process ID of client is used by the server as message type to write the response message. Next client reads the same message queue with message type as its own process ID.

File: mqsrv.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>



#define  MY_KEY            19920809
#define  MSG_LEN           256
#define  SRV_MSG_TYPE      1

void   toggleCase(char *buf, int cnt);


int main()
{
  int mqId;
  int msgLen;
  char rxmsg[MSG_LEN];
  char txmsg[MSG_LEN];

  mqId = msgget(MY_KEY, 0660 | IPC_CREAT);

  if(mqId < 0)
  {
    printf("Could not create message queue\n");
    return(1);
  }
  else
   printf("Opened message queue. Id is %d\n",mqId);


  while(1)
  {
    printf("Server: waiting for message\n");
    msgLen = msgrcv(mqId, rxmsg, MSG_LEN, SRV_MSG_TYPE, 0);
    if(msgLen == -1)
    {
      perror("msgrcv");
      return 1;
    }
    printf("Received message of size %d from client\n",msgLen);
    toggleCase(rxmsg+8, msgLen - 4);
    memcpy(txmsg, rxmsg+4, msgLen);
    msgsnd(mqId,txmsg,msgLen-4,0);
    printf("Sent processed message to client\n");
  }
}
```

File: mqcli.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define  MY_KEY            19920809
#define  MSG_LEN           256
#define  SRV_MSG_TYPE      1

int main()
{
  int mqId;
  int msgLen, cliMsgType;
  char rxmsg[MSG_LEN];
  char txmsg[MSG_LEN];
  int  *msgHdr = (int *)txmsg;

  mqId = msgget(MY_KEY, 0);
  if(mqId < 0)
  {
    printf("Could not open the message queue\n");
    return(1);
  }

  printf("Client: Enter some request message to send to server\n");
  fgets(txmsg+8,MSG_LEN,stdin);
  msgHdr[0] = SRV_MSG_TYPE;
  /** Use process Id as client message type **/
  cliMsgType = getpid();
  msgHdr[1]  = cliMsgType;
  msgsnd(mqId,txmsg,strlen(txmsg+8)+5,0);

  printf("Client: sent given request message to server\n");
  msgLen = msgrcv(mqId, rxmsg, MSG_LEN, cliMsgType, 0);
  printf("Client: Following is the response message from server\n");
  puts(rxmsg+4);
}
```

Semaphores

The System V semaphores are made unnecessarily as complex objects. POSIX semaphores which are available in Unix and Linux, also semaphores available in RTOS are very simple objects. So before studying System V semaphores let us study simple semaphore concept.

We can think semaphore as an object, which holds tokens. Semaphore can hold any number of tokens starting from 0 (no tokens) to some maximum limit. Normally when semaphore object is created, the initial token count will be specified.

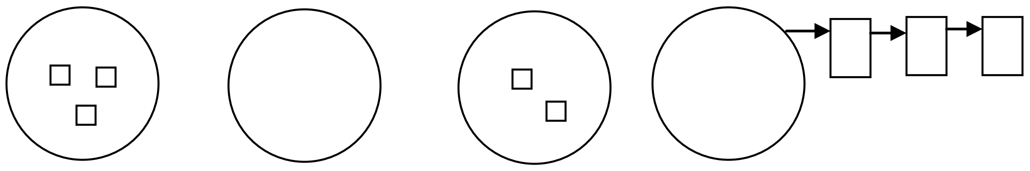


Figure: Three Semaphore objects with different number of tokens in it

The above figure shows four different semaphores. First one holding three tokens, second one zero tokens and third one, two tokens. But on the fourth semaphore there are no tokens and also three Processes are waiting on that semaphore for a token.

A process can perform two types of operations on the semaphores. These are:

- Taking a token from the semaphore
- Giving a token to the semaphore

When a process calls a system call for taking a token from the semaphore, if there are no tokens in the semaphore, the process will block (go to pending state) till it gets a token. If tokens are present in the semaphore, the system call for taking a token will return successfully and token count in the semaphore will be decrement by one.

When a process calls a system call for giving a token, the system call returns successfully and token count in the semaphore will increment by one. If already some process is waiting for a token from that semaphore, then that process will get a token.

Use of semaphores

This simple concept of semaphore can be used for synchronization and mutual exclusion between processes.

Mutual exclusion

Let us consider the usage of semaphore for mutual exclusion. Printer is a resource, which can be used by only process at a time. The other processes can use the printer only when first process finishes its use. This mutual exclusion can be achieved very easily with the execution of following code by all the processes that want to use printer.

During boot time create a printer semaphore with one token in it.

```
Take  token from the semaphore.
Use the printer.
Give token to the semaphore.
```

Every process can call the above code. But only first process will get a token. All the other processes will go to block/pending state. Only when first process gives the token back then only another process will get a token and can use the printer.

Synchronization

The other use of semaphores is for synchronization. Synchronization is typically required between a producer process and consumer process. Initially the semaphore is created with zero tokens in it. The consumer process will block on the semaphore for a token. The producer process produces some data (like request message, page of data to print etc.) and sends a token to the semaphore. This token wakes up the consumer process. The consumer process now consumes the data supplied by the producer process.

Complexity of System V semaphores

Even though semaphores are very simple objects to understand, the System V semaphore are made unnecessarily complex for the following reasons.

- Sysem V semaphore calls allow to create just not one semaphore but set of semaphores and other semaphore system calls also operate on multiple semaphores at a time.
- System V semaphores also support taking and giving multiple tokens with single call.
- System V semaphore, provides a mechanism of waiting for token count to become zero

But POSIX semaphores and semaphores in RTOS, will allow only one semaphore creation at a time and also all system calls will act only on one semaphore. Taking and giving multiple tokens at a time are not allowed. There is no concept of waiting for token count to become zero.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int flag);

int semctl(int semid, int semnum, int cmd, union semun arg);

union semun
{
  int              val;
  struct semid_ds  *buf;
  ushort           *array;
};
int semop(int semid, struct sembuf semoparray[], size_t nops);

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

One reason for the complexity of system V semaphores is its complex interface, which offers more features. If we not want all these features, we can develop simple POSIX semaphore like wrapper functions over these complex system calls. The following client server programs illustrate these wrapper functions.

File: semsrv.c

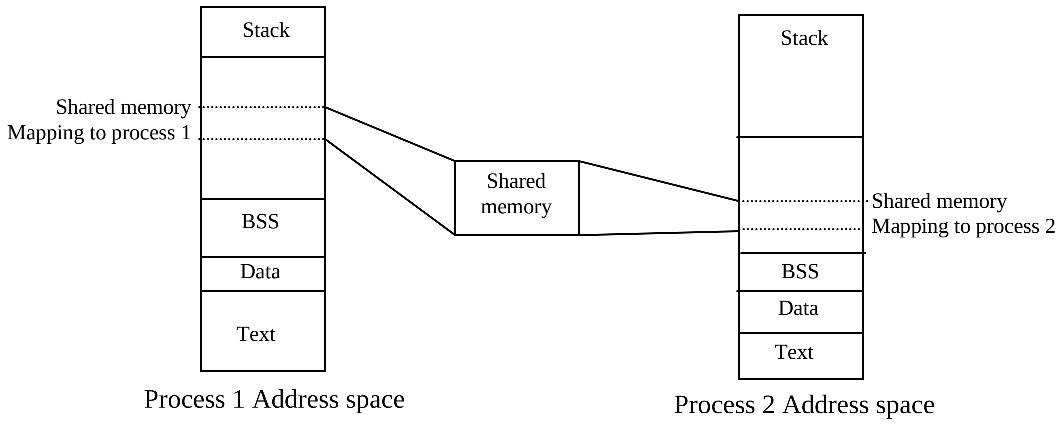
```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

//Simple POSIX semaphore like wrapper functions for system V semaphores
int mysem_init(int key)
{
  return semget(key,1,IPC_CREAT | 0660);
}

int mysem_open(int key)
{
  return semget(key,1,0);
}

int mysem_post(int semid)
{
  struct sembuf sb;

  sb.sem_num = 0;
  sb.sem_op  = 1;
  sb.sem_flg = 0;

  return semop(semid,&sb,1);
}

int mysem_wait(int semid)
{
  struct sembuf sb;

  sb.sem_num = 0;
  sb.sem_op  = -1;
  sb.sem_flg = 0;

  return semop(semid,&sb,1);
}

int mysem_destroy(int semid)
{
  semctl(semid,0,IPC_RMID,0);
}

int main()
{
  int semid;

  semid = mysem_init(1234);
  if(semid < 0)
  {
    perror("Sem open failed");
    exit(1);
  }
  printf("semid = %d\n", semid);

  while(1)
  {
    mysem_wait(semid);
    printf("I got semaphore token\n");
  }
}
```

File: semcli.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

//Copy wrapper functions of semsrv.c file here.
int main()
{
  int semid;

  semid = mysem_open(1234);
  if(semid < 0)
  {
    perror("Sem open failed");
    exit(1);
  }
  printf("semid = %d\n", semid);
  while(1)
  {
    printf("Press ENTER key to send semaphore\n");
    getchar();
    mysem_post(semid);
  }
}
```

Shared Memory

Every process has its own address space. One process can not access the memory belong to other processes. But shared memory allows some memory to be shared by multiple processes. For this we need to create some memory segment first, and then attach that memory to all the required processes. With attach, this memory is mapped into the address space of all the processes that attached it. If any process writes to this shared memory then it is visible to all other processes.



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

int shmget(key_t key, int size, int flag);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);

void * shmat(int shmid, void *addr, int flag);

int shmdt (void *addr);
```

The shared memory alone can not be used for inter process communication. When a process writes to a shared memory, we need a way to inform the second process that it can go and read the data from shared memory. Semaphores are ideally suited for this purpose. These two processes (one writing to shared memory, other reading from shared memory) can be considered as producer and consumer processes. We can use semaphores to synchronize these two processes.

The following sample program illustrates the use of shared memory and semaphores for inter process communication.

Shared memory and Semaphores base Client Sever programs

File: shmsrv.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>

#define  MY_KEY            19920809
#define  SHM_SIZE          0x1000

void   toggleCase(char *buf, int cnt);

int main()
{
  int semId,shmId;
  char *pShm;
  struct sembuf smop;

  /** Create a semaphore set, containing two semaphores **/
  semId = semget(MY_KEY, 2, 0660 | IPC_CREAT);
  if(semId < 0)
  {
    printf("Could not create semaphore\n");
    return(1);
  }
  else
   printf("Opened a semaphore. Id is %d\n",semId);

  /** Set initial token count of both semaphores to zeros **/
  semctl(semId,0,SETVAL,0);
  semctl(semId,1,SETVAL,0);

  /** Create shared memory segment **/
  shmId = shmget(MY_KEY, SHM_SIZE, 0660 | IPC_CREAT);
  if(shmId < 0)
  {
    printf("Could not create shared memory segment\n");
    return(2);
  }

  /*** Attach shared memory segment to process address space ***/
  pShm = shmat(shmId, NULL, 0);
  if(!pShm)
  {
    printf("Could not attach shared memory segment\n");
    return(3);
  }

  while(1)
  {
    /** wait for a token from semaphore 0 **/
    smop.sem_num = 0;
    smop.sem_op  = -1;
    smop.sem_flg = 0;
    semop(semId, &smop, 1);

    /** Process the message available in shared memory **/
    printf("Got the semaphore\n");
    strcpy(pShm+256, pShm);
    toggleCase(pShm+256, strlen(pShm+256));
    printf("Processed the request message, and placed response\n");

    /** Send token to semaphore 1 **/
    smop.sem_num = 1;
    smop.sem_op  = 1;
    smop.sem_flg = 0;
    semop(semId, &smop, 1);
  }
}
```

File: shmcli.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>

#define  MY_KEY            19920809
#define  SHM_SIZE          0x1000
#define  MSG_LEN           256
#define  RESP_MSG_START    256

int main()
{
  int semId,shmId;
  char *pShm;
  struct sembuf smop;

  semId = semget(MY_KEY, 2, 0);

  if(semId < 0)
  {
    printf("Could not create semaphore\n");
    return(1);
  }
  else
   printf("Opened a semaphore. Id is %d\n",semId);

  shmId = shmget(MY_KEY, SHM_SIZE, 0);

  if(shmId < 0)
  {
    printf("Could not create shared memory segment\n");
    return(2);
  }

  pShm = shmat(shmId, NULL, 0);
  if(!pShm)
  {
    printf("Could not attach shared memory segment\n");
    return(3);
  }

  printf("Client: Enter some request message to send to server\n");
  fgets(pShm,MSG_LEN,stdin);
  smop.sem_num = 0;
  smop.sem_op  = 1;
  smop.sem_flg = 0;
  semop(semId, &smop, 1);
  smop.sem_num = 1;
  smop.sem_op  = -1;
  smop.sem_flg = 0;
  semop(semId, &smop, 1);
  puts(pShm+RESP_MSG_START);
}
```

Listing System V IPC objects

All the IPC objects reside in the kernel memory. By using *ipcs* command we can display all of them. Following is the output of *ipcs* command. According to this command output there is only one semaphore with key 0x4d2 and semid 0 with owner as karasala.

```
$ ipcs

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x000004d2 0          karasala  660        1

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
```

8. Network Programming

Network programming is all about writing network application programs. Network application programs are the ones, which uses network API and communicates with the other programs running on some other machines. Typical network applications are client server programs and distributed computing programs.

Extended IPC

We can treat network communication between processes as an extension to IPC mechanisms studied in Linux Programming. Only difference is that, IPC is restricted to processes running on a same machine. Where as network communication can be established between processes running on different machines, provided these machines are connected through network. These machines could be present on a LAN. Or these machines could be sitting across a globe, connected by internet.

Choice of API

If we want to do network programming in C language, we have two choices. One is Berkeley socket APIs and other is System V Transport Layer Interface (TLI) APIs. Socket API is more popular than TLI and we will study these APIs. When other languages like C++ and Java are used for network programming, they will have their own APIs for network programming.

But as you guessed, lot of things are going on under these simple APIs we are using to establish network communication. TCP/IP protocol is responsible for providing this network communication across the LAN or across the internet.

Socket Concept

Socket represents end-point of network communication link. Communication is typically established between two end points. That is between two sockets. Each socket is associated with the following five parameters.

- Protocol used (datagram(UDP) / stream(TCP))
- Source IP address
- Destination IP address
- Source port address
- Destination port address

The first step for any network program is to create a socket. Both server and client network programs will create socket first. Then the server network program will bind the socket to a well-known port address. The well-known port address allows the client network programs to connect to that server socket. These server side sockets, which are not yet connected to client side socket, are said to be half bind. This is because these sockets do not have destination IP address and destination port address.

Once client connects to a server socket, then both sockets (client side socket and server side socket) are fully bind.

Client Server Concept

Most of the network applications follows client server model. In this model, server application programs runs forever, and waits for request messages from clients. When a request message is

received, server processes it, and sends response to it. So server applications always run in the background as daemon processes (Daemon processes are described later).
User typically starts client applications. Client applications connect to the server and exchange messages or files with the server and closes the connection. For example user runs POP client program to download mails from the remote mail server (POP server). Similarly user runs web browser (Netscape or Internet Explorer) which is a HTTP client program, to browse HTML files present on the web server (HTTP server).

Distributed computing

Another major area of networking applications is distributed computing. In this model large application is spilt into various small applications and each application is made to run on different computers, which are connected together. In this model there is no clear distinction between server and client. Every node acts as both client and server. We may call this as peer to peer networking.

Socket System calls

One note before studying Socket system calls is that, these socket system calls are designed to support many network communication protocols. TCP/IP is one of the network protocols supported by sockets. Because of this reason the parameters to most of Socket system calls appears to be generic. For example as you see later, every where we pass generic sockaddr structure, even though we use sockaddr_in structure. Also we pass always size of socket address size, which is required to support variable length addresses as required by other network protocols.

Many of socket system calls require a pointer to a socket address structure as an argument. The definition of this structure is in <sys/socket.h>

```
struct sockaddr
{
  u_short sa_familty;  /* address family: AF_xxx value */
  char    sa_data[14]; /* upto 14 bytes of protocol specific addrs*/
};
```

The content of the 14 bytes of protocol-specific address is interpreted according to the type of address. For internet family (AF_INET), the following structures are defined in <netinet/in.h>:

```
struct in_addr
{
  u_long s_addr; /* 32 bit IP address in network byte order */
};

struct sockaddr_in
{
  short          sin_family;  /* AF_INET */
  u_short        sin_port;    /*16-bit port number in network order */
  struct in_addr sin_addr;    /* 32bit IP address in network order */
  unsigned char  sin_zero[8]; /* to make size as total 14 bytes    */
};
```

Socket system call

This *socket()* system call is used both by client and server to create a socket. This is first step of network communication. The value return by socket is a small integer, similar to file descriptor. This return value of a socket is typically called sockfd.

```
int socket(int family, int type, int protocol);
```

The `family` specifies the type of network protocol to be used, we always set this family as AF_INET, which stands for Internet protocol or TCP/IP protocol.

The `type` specifies characteristics of the communication we want. Following are possible types:

SOCK_STREAM
SOCK_DGRAM

The SOCK_STREAM provides connection oriented, reliable, stream based service. The SOCK_DGRAM provides connection-less, message oriented service.

The `protocol` argument is typically set to 0 for most user applications.

The socket system call returns a small integer value, similar to a file descriptor. Let us call this as socket descriptor or sockfd.

bind System call

This *bind()* system call is typically used by server to bind the socket to a known source port address.

```
int bind(int sockfd, struct sockaddr *myaddr, int addrlen);
```

The first parameter is the socket FD created using socket() system call. The second parameter is pointer to sockaddr structure. But what we use in program is sockaddr_in structure. We pass address of sockaddr_in structure by type casting it as sockaddr structure.

listen System call

This *listen()* system call is only used by TCP server to specify the queue length for the connection requests from clients. TCP stack will queue this specified number of connections, for the server to accept once server is ready to accept a new connection.

```
int listen(int sockfd, int backlog);
```

accept System call

This *accept()* system call is only used by TCP server program. This is a blocking call. Till client connects to this socket, the server program will be in blocking state, waiting for the connection.

```
int accept(int sockfd, struct sockaddr *cliaddr, int *addrlen);
```

connect System call

This *connect()* system call is typically used by a client program. This system call initiates the connection to a server socket.

```
int connect(int sockfd, struct sockaddr *servaddr, int addrlen);
```

Data transfer system calls

All the above calls are used to create sockets, to set address to a socket, to request for connections or to wait for connections. But to transfer data, we can use standard read() and write() calls on the sockets. Besides standard read() and write() calls, we can also following calls to read and write data to sockets.

```
int send(int sockfd, char *buff, int nbytes, int flags);

int sendto(int sockfd, char *buff, int nbytes, int flags,
           struct sockaddr *toaddr, int addrlen);

int recv(int sockfd, char *buff, int nbytes, int flags);

int recvfrom(int sockfd, char *buff, int nbytes, int flags,
           struct sockaddr *fromaddr, int *addrlen);
```

close System call

Once data transfer is completed, we need to close the socket. Standard close() system call can be used for this.

```
int close(int sockfd);
```

Byte ordering APIs

In a network there can exist different types computers using different types of CPUs or Microprocessors. Some CPUs may follow big endian format (most significant byte of integer is stored first in memory) for storing integers in memory, others may follow little endian format. So TCP/IP specifies that all the fields in protocol headers of network packets should be in big endian format only. Every machine on network should be aware of this and should convert from host byte order to network byte order when sending messages on the network. Same way fields in the incoming messages should be converted from network byte order to host byte order.

Application uses the following APIs to convert from host order to network byte order and vice versa. These APIs are implemented to suite the requirements of CPU. If CPU is already big endian, then these functions will not change the byte order. So once these APIs are used, the source code works for both type of CPUs.

```
u_long htonl(u_long hostlong);
u_short htons (u_short hostshort);
u_long ntohl(u_long netlong);
u_short ntohs(u_short netshort);
```

Address conversion APIs

```
unsigned long inet_addr(char *ptr);
```

This API converts a character string in dotted-decimal notation to a 32-bit internet address in network byte order.

```
char * inet_ntoa(struct in_addr inaddr);
```

This API does the reverse of above API. That is, it takes IP address in binary format and returns the character string representing IP address in dotted decimal format.

UDP client server program

UDP Server

File: udpsrv.c

```
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/socket.h>
#include    <netinet/in.h>
#include    <arpa/inet.h>

#define     SRV_UDP_PORT  8000
#define     MAX_MSG       100
void errExit(char *str)
{
  puts(str);
  exit(0);
}

int main()
{
  int                 sockFd;
  struct sockaddr_in  srvAdr, cliAdr;
  int                 cliLen,n;
  char                mesg[MAX_MSG];

  if( (sockFd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    errExit("Can't open datagram socket\n");

  memset(&srvAdr,0,sizeof(srvAdr));
  srvAdr.sin_family      = AF_INET;
  srvAdr.sin_addr.s_addr = htonl(INADDR_ANY);
  srvAdr.sin_port        = htons(SRV_UDP_PORT);

  if(bind(sockFd,(struct sockaddr*)&srvAdr, sizeof(srvAdr)) < 0)
    errExit("Can't bind local address\n");

  printf("Server waiting for messages\n");
  while(1)
  {
    cliLen = sizeof(cliAdr);
    n = recvfrom(sockFd, mesg, MAX_MSG, 0,(struct sockaddr*)&cliAdr, &cliLen);
    if(n < 0)
      errExit("recvfrom error\n");

    if(sendto(sockFd,mesg,n,0,(struct sockaddr*)&cliAdr,cliLen) != n)
      errExit("sendto error\n");

    printf("Received following message from client %s\n%s\n",
            inet_ntoa(cliAdr.sin_addr),mesg);
  }
}
```

UDP Client

File : udpcli.c

```
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/socket.h>
#include    <netinet/in.h>
#include    <arpa/inet.h>

#define     SRV_IP_ADRS  "127.0.0.1"
#define     SRV_UDP_PORT  8000
#define     MAX_MSG       100

void errExit(char *str)
{
  puts(str);
  exit(0);
}

int main()
{
  int                 sockFd;
  struct sockaddr_in  srvAdr;
  char                txmsg[MAX_MSG];
  char                rxmsg[MAX_MSG];
  int                 n;

  if( (sockFd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    errExit("Can't open datagram socket\n");

  memset(&srvAdr,0,sizeof(srvAdr));
  srvAdr.sin_family      = AF_INET;
  srvAdr.sin_addr.s_addr = inet_addr(SRV_IP_ADRS);
  srvAdr.sin_port        = htons(SRV_UDP_PORT);

  printf("Enter message to send:\n");
  fgets(txmsg,MAX_MSG,stdin);

  n = strlen(txmsg) + 1;
  if(sendto(sockFd,txmsg,n,0,(struct sockaddr*)&srvAdr,sizeof(srvAdr)) != n)
    errExit("sendto error\n");

  n = recv(sockFd, rxmsg, MAX_MSG, 0);
  if(n < 0)
    errExit("recv error\n");

  printf("Received following message:\n%s\n",rxmsg);
}
```

TCP client server program

TCP Server (Iterative server)

File: tcpsrv.c

```
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/socket.h>
#include    <netinet/in.h>
#include    <arpa/inet.h>

#define     SRV_TCP_PORT  8000
#define     MAX_MSG       100

void errExit(char *str)
{
  puts(str);
  exit(0);
}

int main()
{
  int                   sockFd,newSockFd;
  struct sockaddr_in  srvAdr, cliAdr;
  int                   cliLen,n;
  char                  mesg[MAX_MSG];

  if( (sockFd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    errExit("Can't open stream socket\n");

  memset(&srvAdr,0,sizeof(srvAdr));
  srvAdr.sin_family      = AF_INET;
  srvAdr.sin_addr.s_addr = htonl(INADDR_ANY);
  srvAdr.sin_port        = htons(SRV_TCP_PORT);

  if(bind(sockFd,(struct sockaddr*)&srvAdr, sizeof(srvAdr)) < 0)
    errExit("Can't bind local address\n");

  listen(sockFd,5);

  while(1)
  {
    printf("Server waiting for new connection:\n");
    cliLen = sizeof(cliAdr);
    newSockFd = accept(sockFd, (struct sockaddr *) &cliAdr, &cliLen);
    if(newSockFd < 0)
      errExit("accept error\n");
    printf("Connected to client: %s\n", inet_ntoa(cliAdr.sin_addr));

    while(1)
    {
      n = recv(newSockFd, mesg, MAX_MSG, 0);
      if(n < 0)
        errExit("recv error\n");
      if(n==0)
      {
        close(newSockFd);
        break;
      }

      mesg[n] = 0;
      if(send(newSockFd,mesg,n,0) != n)
        errExit("send error\n");

      printf("Received and sent following message:\n%s\n", mesg);
    }
  }
}
```

TCP Client

File: tcpcli.c

```
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/socket.h>
#include    <netinet/in.h>
#include    <arpa/inet.h>

#define     SRV_IP_ADRS  "127.0.0.1"
#define     SRV_TCP_PORT  8000
#define     MAX_MSG       100

void errExit(char *str)
{
  puts(str);
  exit(0);
}

int main()
{
  int                 sockFd;
  struct sockaddr_in  srvAdr;
  char                txmsg[MAX_MSG];
  char                rxmsg[MAX_MSG];
  int                 n;

  if( (sockFd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    errExit("Can't open stream socket\n");

  memset(&srvAdr,0,sizeof(srvAdr));
  srvAdr.sin_family      = AF_INET;
  srvAdr.sin_addr.s_addr = inet_addr(SRV_IP_ADRS);
  srvAdr.sin_port        = htons(SRV_TCP_PORT);

  if(connect(sockFd,(struct sockaddr *)&srvAdr,sizeof(srvAdr)) < 0)
    errExit("can't connect to server\n");

  while(1)
  {
    printf("Enter message to send, Enter # to exit:\n");
    fgets(txmsg,MAX_MSG,stdin);
    if(txmsg[0] == '#')
      break;
    n = strlen(txmsg) + 1;
    if(send(sockFd,txmsg,n,0) != n)
      errExit("send error\n");

    n = recv(sockFd, rxmsg, MAX_MSG, 0);
    if(n < 0)
      errExit("recv error\n");

    printf("Received following message:\n%s\n",rxmsg);
  }
  close(sockFd);
}
```

Network communication Vs IPC

If you look at the TCP and UDP communication described above, you can find lot of similarities with Inter Process Communication(IPC) studied in Linux programming. We can compare TCP connection to FIFO IPC and UDP communication to message queues IPC.

FIFO and TCP

FIFO and TCP both are connection oriented. If you try to open a FIFO for reading (server), the open() system call will block till other end of the FIFO is opened by another process (client) for writing. In this way data transfer between client and server starts only after both parties agreed to connect to each other (by using open() system call). In TCP also same is the case, server blocks at accept() system call and client uses connect() system call to connect. Then only connection will establish. Data transfer takes place only after connection is established.

Another similarity between FIFO and TCP is that, both are stream oriented. That means no boundaries are maintained in the data flow. Client can write data to FIFO or TCP by using 10 write calls. But server can read all the data in a single call if it provides sufficient buffer. Same way client can write large data in single write() call. But server can read in 10 times, each time reading only part of the data.

Message Queues and UDP

Both message queues and UDP communication are message oriented (NOT stream oriented). Message boundaries are maintained. If client writes 10 times to message queue or UDP, then the server has to read exactly 10 times, to read all the 10 messages. Each read at server will fetch only one message from message queue or UDP socket. Server can provide big buffer to read multiple messages in a single call. But still it receives only one message.

Both UDP and message queues provides connection less data transfers. UDP can write to a socket or message queue, independent of any other process is reading from the other side or not. But one difference between UDP and message queues is regarding reliability. We call UDP as unreliable communication, as packets could be dropped and sender will not know about it. But message queues being run on same system, are reliable.

Server application design

We can classify server applications into two classes. One class is iterative servers and second is concurrent servers..

Iterative servers

Iterative server is the one, which serves only one connection (client) at a time. Only after closing connection with one client, the server will serve the second client. The sample TCP echo server program studied in previous session belongs to iterative server class. In this program, once accept() system call returns a client socket, it enters into a forever loop and serves the client requests (here serving is just echoing whatever received). The program comes out of the loop only when client closes the connection (receiving 0 bytes from recv() call is indication for client closing the connection). Once client closes the connection, the server program again blocks at accept() system call for new client connection.

The iterative server is simple to implement, and will require less resources from OS. If each client transaction is only of small duration, iterative server is good. For example client just sends one request, waits for response and closes the connection once response is received. On the other hand if each client does not close connection immediately, then iterative server cannot serve other client requests immediately.

Concurrent servers

Concurrent servers, as name implies serve multiple clients simultaneously. So client need not wait till all previous clients in the queue are finished. Concurrency in serving multiple clients can be achieved in three different ways:

- Forking a new process to serve each client connection
- Using a separate thread for each client connection
- Using select() system call to serve multiple clients with single process

Concurrent server by forking a new process

`File: tcpCsrvF.c`

```
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/socket.h>
#include    <netinet/in.h>
#include    <arpa/inet.h>

#define     SRV_TCP_PORT  8000
#define     MAX_MSG       100

void errExit(char *str)
{
  puts(str);
  exit(0);
}

int main()
{
  int                 sockFd,newSockFd;
  struct sockaddr_in  srvAdr, cliAdr;
  int                 cliLen,n;
  char                mesg[MAX_MSG];
  int                 pid;

  if( (sockFd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    errExit("Can't open stream socket\n");

  memset(&srvAdr,0,sizeof(srvAdr));
  srvAdr.sin_family      = AF_INET;
  srvAdr.sin_addr.s_addr = htonl(INADDR_ANY);
  srvAdr.sin_port        = htons(SRV_TCP_PORT);

  if(bind(sockFd,(struct sockaddr*)&srvAdr, sizeof(srvAdr)) < 0)
    errExit("Can't bind local address\n");

  listen(sockFd,5);

  while(1)
  {
    printf("Server waiting for new connection:\n");
    cliLen = sizeof(cliAdr);
    newSockFd = accept(sockFd, (struct sockaddr *) &cliAdr, &cliLen);
    if(newSockFd < 0)
      errExit("accept error\n");
    printf("Connected to client: %s\n", inet_ntoa(cliAdr.sin_addr));

    pid = fork();
    if(pid == 0) /*** Child process ***/
    {
      while(1)
      {
        n = read(newSockFd, mesg, MAX_MSG);
        if(n < 0)
          errExit("recv error\n");
        if(n==0)
          break;
        mesg[n] = 0;
        if(write(newSockFd,mesg,n) != n)
          errExit("send error\n");

        printf("Received and sent following message:\n%s\n", mesg);
      }
      exit(0);
    }
    else  /*** Parent process ***/
    {
      close(newSockFd);
    }
  }
}
```

Multi-threaded concurrent server

File: tcpCsrvT.c

```
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/socket.h>
#include    <netinet/in.h>
#include    <arpa/inet.h>
#include    <pthread.h>

#define     SRV_TCP_PORT  8000
#define     MAX_MSG       100

void * servClient(void *arg);

void errExit(char *str)
```

```
{
  puts(str);
  exit(0);
}

int main()
{
  int                 sockFd,newSockFd;
  struct sockaddr_in  srvAdr, cliAdr;
  int                 cliLen;
  pthread_t           srvThread;

  if( (sockFd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    errExit("Can't open stream socket\n");

  memset(&srvAdr,0,sizeof(srvAdr));
  srvAdr.sin_family      = AF_INET;
  srvAdr.sin_addr.s_addr = htonl(INADDR_ANY);
  srvAdr.sin_port        = htons(SRV_TCP_PORT);

  if(bind(sockFd,(struct sockaddr*)&srvAdr, sizeof(srvAdr)) < 0)
    errExit("Can't bind local address\n");

  listen(sockFd,5);

  while(1)
  {
    printf("Server waiting for new connection:\n");
    cliLen = sizeof(cliAdr);
    newSockFd = accept(sockFd, (struct sockaddr *) &cliAdr, &cliLen);
    if(newSockFd < 0)
      errExit("accept error\n");
    printf("Connected to client: %s\n", inet_ntoa(cliAdr.sin_addr));
    pthread_create(&srvThread,NULL,servClient,(void*)newSockFd);
  }
}

void * servClient(void *arg)
{
  int   n;
  int   sockfd = (int) arg;
  char  mesg[MAX_MSG];

  pthread_detach(pthread_self());
  while(1)
  {
    n = recv(sockfd, mesg, MAX_MSG, 0);
    if(n < 0)
      return;
    if(n==0)
      break;
    mesg[n] = 0;
    if(send(sockfd,mesg,n,0) != n)
      errExit("send error\n");
    printf("Received and sent following message:\n%s\n", mesg);
  }
  close(sockfd);
}
```

Select System call

When we are developing terminal emulator program, we found that single thread will not work. This is because, we want wait on two inputs simultaneously. That means we want to read both from keyboard and from serial port. But process can read only from one file descriptor at a time. If data is not present with the file descriptor, then read() system call will block and we can not read from second descriptor even if data is available.

One way of solving this problem is polling. In this case will open both the descriptors (keyboard and serial port) in non-blocking mode. So the read() system call will return immediately when no data is present. So program can keep on reading both the descriptors one after another till data is available. But polling is very inefficient technique; it wastes CPU time significantly even when no data is present with descriptors.

So we solved this problem in threads section by having two threads. So that one thread can read the keyboard(standard input) and other thread can read and block on the serial port.
But Linux has got a mechanism to block for read and write operations on multiple file descriptors simultaneously. The select() system will provide this facility.

Fundamental concept for select() system call is FD Set. FD set is a data structure, which represent a set of file descriptors. We can create FD Set data structure as below. Next we can zero (reset) all the FDs in the set, or we can set any one of the FD in the set to one, or we can check whether any FD in the set is zero or one.

```
fd_set      myFds;  /* myFds is the fd_set data structure */
```

Following macros are used to perform above mentioned operations on the FD set.

```
FD_ZERO(fd_set *fdset);
FD_SET(int fd, fd_set *fdset);
FD_CLR(int fd, fd_set *fdset);
FD_ISSET(int fd, fd_set *fdset);
```

Following is the prototype for select() system call:

```
int select(int nfds, fd_set *readFds, fd_set *writeFds,
           fd_set *errorFds, struct timeval *timeout);
```

The select() system call will take pointers to three FD Sets. First FD set represents the file descriptors, which we want to test for the availability of data for reading those FDs. The second set is for testing the readiness of the FDs for taking the data, that is to write to data. The last FD set is for testing if any FDs in this group have any errors pending. If we want to pass only one or two FD sets, we can set other FD set pointers as null pointers.

The first parameter to the select() funcation, 'nfds', will tell the select() system call, the number of maximum FDs that need to be checked for each FD set. For example each FD set may represent 256 FDs. But if application is using only few FDs, and the maximum value of FD is 10, then 'nfds' could be specified as 11. That is only FDs 0 to 10 need to be tested.

The last parameter to select system call provides optional timeout feature to the select() function. When this pointer to timeval structure is non-zero, it represents the timeout period for the select() function. If this timeout is specified then select() will return when timeout occurs, even if no FD set is ready for data transfer.

File: termEmulSelect.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <termios.h>
#include <fcntl.h>

int serial_init()
{
  int sfd;
  struct termios tio;

  sfd = open("/dev/ttyS0", O_RDWR | O_NOCTTY);
  if(sfd < 0)
  {
    perror("Unable to open /dev/ttyS0");
    exit(1);
  }
  tcgetattr(sfd, &tio);
  tio.c_cflag        = B9600 | CS8 | CLOCAL | CREAD;
  tio.c_iflag        = 0;
  tio.c_oflag        = 0;
  tio.c_lflag        = 0;
  tcflush(sfd, TCIFLUSH);
  tcsetattr(sfd,TCSANOW,&tio);
  return sfd;
}

int main()
{
  char            ch;
  struct termios  cuset, newset;
  fd_set          readfds, testfds;
  int             serfd,stat;

  serfd = serial_init();

  /*** Disable echo and canonical mode of processing ***/
  tcgetattr(0,&cuset);
  newset = cuset;
  newset.c_lflag &= ~ICANON;
  newset.c_lflag &= ~ECHO;
  tcsetattr(0,TCSANOW,&newset);

  FD_ZERO(&readfds);
  FD_SET(0,&readfds);
  FD_SET(serfd,&readfds);

  while(1)
  {
    testfds = readfds;
    stat = select(serfd+1, &testfds, NULL, NULL, NULL);

    /*** If standard input FD is set ***/
    if(FD_ISSET(0,&testfds))
    {
      /*** Read from standard input ***/
      read(0,&ch,1);

      /*** Write to serial device ***/
      write(serfd,&ch,1);
    }

    /*** If Serial device FD is set ***/
    if(FD_ISSET(serfd,&testfds))
    {
      /*** Read from standard input ***/
      read(serfd,&ch,1);

      /*** Write to standard output ***/
      write(1,&ch,1);
    }
  }
}
```

File: tmoutSelect.c

```
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/socket.h>
#include    <netinet/in.h>
#include    <arpa/inet.h>

#define     SRV_UDP_PORT  8000
#define     MAX_MSG       100

void errExit(char *str)
{
  puts(str);
  exit(0);
}

int main()
{
  struct timeval      tv;
  int                 sockFd,stat;
  struct sockaddr_in  srvAdr, cliAdr;
  int                 cliLen,n;
  char                mesg[MAX_MSG];
  fd_set              readfds, testfds;

  if( (sockFd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    errExit("Can't open datagram socket\n");

  memset(&srvAdr,0,sizeof(srvAdr));
  srvAdr.sin_family      = AF_INET;
  srvAdr.sin_addr.s_addr = htonl(INADDR_ANY);
  srvAdr.sin_port        = htons(SRV_UDP_PORT);

  if(bind(sockFd,(struct sockaddr*)&srvAdr, sizeof(srvAdr)) < 0)
    errExit("Can't bind local address\n");

  printf("Server waiting for messages\n");

  FD_ZERO(&readfds);
  FD_SET(sockFd,&readfds);

  while(1)
  {
    testfds = readfds;
    tv.tv_sec = 5;
    tv.tv_usec = 0;

    stat    = select(sockFd+1, &testfds, (fd_set *)0, (fd_set *)0, &tv);
    if(stat == 0)
    {
      printf("Time out error\n");
      continue;
    }

    cliLen = sizeof(cliAdr);
    n = recvfrom(sockFd, mesg, MAX_MSG, 0,(struct sockaddr*)&cliAdr, &cliLen);

    if(sendto(sockFd,mesg,n,0,(struct sockaddr*)&cliAdr,cliLen) != n)
      errExit("sendto error\n");

    printf("Received following message from client %s\n%s\n",
            inet_ntoa(cliAdr.sin_addr),mesg);
  }
}
```

Concurrent server with select() system call

`File: tcpCsrvS.c`

```
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/socket.h>
#include    <netinet/in.h>
#include    <arpa/inet.h>

#define     SRV_TCP_PORT  8000
#define     MAX_MSG       100

void errExit(char *str)
{
  puts(str);
  exit(0);
}

int main()
{
  int                 srvSockFd,newSockFd,fd;
  struct sockaddr_in  srvAdr, cliAdr;
  int                 cliLen,n;
  char                mesg[MAX_MSG];
  fd_set              readfds, testfds;
  int                 stat;

  if( (srvSockFd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    errExit("Can't open stream socket\n");

  memset(&srvAdr,0,sizeof(srvAdr));
  srvAdr.sin_family      = AF_INET;
  srvAdr.sin_addr.s_addr = htonl(INADDR_ANY);
  srvAdr.sin_port        = htons(SRV_TCP_PORT);

  if(bind(srvSockFd,(struct sockaddr*)&srvAdr, sizeof(srvAdr)) < 0)
    errExit("Can't bind local address\n");

  listen(srvSockFd,5);
  FD_ZERO(&readfds);
  FD_SET(srvSockFd,&readfds);


  printf("Server waiting for new connection:\n");
  while(1)
  {
    testfds = readfds;
    stat    = select(FD_SETSIZE, &testfds, (fd_set *)0, (fd_set *)0,
                   (struct timeval *)0);
    if(stat < 1)
      errExit("select error\n");

    for(fd=0; fd<FD_SETSIZE; fd++)
    {
      if(FD_ISSET(fd,&testfds))
      {
        if(fd == srvSockFd)
        {
          newSockFd = accept(fd, (struct sockaddr *)0 , NULL);
          FD_SET(newSockFd,&readfds);
          printf("Adding client fd: %d to readFdSet\n",newSockFd);
        }
        else
        {
          n = read(fd, mesg, MAX_MSG);
          if(n < 0)
            errExit("recv error\n");
          else if(n==0)
          {
            close(fd);
            FD_CLR(fd, &readfds);
            printf("Removing client fd: %d from readFdSet\n",fd);
          }
          else
          {
            if(write(fd,mesg,n) != n)
              errExit("send error\n");
          }
        }
      }
    }
  }
}
```

File: tmoutSockopt.c

```
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/socket.h>
#include    <netinet/in.h>
#include    <arpa/inet.h>

#define     SRV_UDP_PORT  8000
#define     MAX_MSG       100

void errExit(char *str)
{
  puts(str);
  exit(0);
}

int main()
{
  struct timeval      tv;
  int                 sockFd;
  struct sockaddr_in  srvAdr, cliAdr;
  int                 cliLen,n;
  char                mesg[MAX_MSG];

  if( (sockFd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    errExit("Can't open datagram socket\n");

  memset(&srvAdr,0,sizeof(srvAdr));
  srvAdr.sin_family      = AF_INET;
  srvAdr.sin_addr.s_addr = htonl(INADDR_ANY);
  srvAdr.sin_port        = htons(SRV_UDP_PORT);

  if(bind(sockFd,(struct sockaddr*)&srvAdr, sizeof(srvAdr)) < 0)
    errExit("Can't bind local address\n");

  printf("Server waiting for messages\n");

  tv.tv_sec = 5;
  tv.tv_usec = 0;
  setsockopt(sockFd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));

  while(1)
  {
    cliLen = sizeof(cliAdr);
    n = recvfrom(sockFd, mesg, MAX_MSG, 0,(struct sockaddr*)&cliAdr, &cliLen);
    if(n < 0)
    {
      printf("Time out error\n");
      continue;
    }

    if(sendto(sockFd,mesg,n,0,(struct sockaddr*)&cliAdr,cliLen) != n)
      errExit("sendto error\n");

    printf("Received following message from client %s\n%s\n",
            inet_ntoa(cliAdr.sin_addr),mesg);
  }
}
```

Daemon Processes

Most of the server applications will run as daemon processes. Daemon processes will run in background mode, they will not have any controlling terminal. Daemon processes are different from the applications you run in background mode by giving ‘&’ at the end of the command. When you run applications in background mode, these background applications are still associated with controlling terminal and if you logout from terminal, all the background processes belonging to this terminal also get killed.

File: udpdmn.c

```
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/socket.h>
#include    <netinet/in.h>
#include    <arpa/inet.h>

#define     SRV_UDP_PORT  8000
#define     MAX_MSG       100

void errExit(char *str)
{
  puts(str);
  exit(0);
}

int daemonInit(void)
{
  pid_t pid;

  if( (pid = fork()) < 0)
    return(-1);
  else if (pid != 0)
    exit(0);              /** Parent exits **/

  /** Child continues **/
  setsid();
  chdir("/");
  umask(0);
  return(0);
}

int main()
{
  int                 sockFd;
  struct sockaddr_in  srvAdr, cliAdr;
  int                 cliLen,n;
  char                mesg[MAX_MSG];


  if( (sockFd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    errExit("Can't open datagram socket\n");

  memset(&srvAdr,0,sizeof(srvAdr));
  srvAdr.sin_family      = AF_INET;
  srvAdr.sin_addr.s_addr = htonl(INADDR_ANY);
  srvAdr.sin_port        = htons(SRV_UDP_PORT);

  if(bind(sockFd,(struct sockaddr*)&srvAdr, sizeof(srvAdr)) < 0)
    errExit("Can't bind local address\n");

  printf("I am (UDP Server) becoming daemon and waiting”
         “for messages\n");
  daemonInit();

  while(1)
  {
    cliLen = sizeof(cliAdr);
    n = recvfrom(sockFd, mesg, MAX_MSG, 0,(struct sockaddr*)&cliAdr, &cliLen);
    if(n < 0)
      exit(1);
    if(sendto(sockFd,mesg,n,0,(struct sockaddr*)&cliAdr,cliLen) != n)
      exit(1);
  }
}
```

Telnet program as a generic TCP client program

We are using telnet client program to login to a remote server. Here telnet client program is talking to a telnet daemon program on the server. The telnet daemon in turn talks to a shell process on the server.

Though this is a standard use of telnet client, telnet client can be used to connect to any other TCP servers including our own TCP servers. For this we need to specify the port number also as shown below.

```
$ telnet 192.168.0.1 5555
```

The telnet client connects to the server whose port number is 5555. What ever typed on the telnet client is accumulated till ‘Enter’ character is pressed. Once enter character is pressed, the telnet client sends the entire line to the server. Telnet client also displays the response messages sent by the server. So telnet client can be used to test TCP server programs.

netstat Command

The ‘netstat’ command is useful to monitor the sockets opened by applications. The following commands list all the UDP sockets. The ‘n’ option used in second command displays numbers instead of names.

```
# netstat -ua
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
udp        0      0 *:bootps                *:*
udp        0      0 *:tftp                  *:*

# netstat -uan
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
udp        0      0 0.0.0.0:67              0.0.0.0:*
udp        0      0 0.0.0.0:69              0.0.0.0:*

# netstat -ta
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 *:www                   *:*                     LISTEN
tcp        0      0 *:x11                   *:*                     LISTEN
tcp        0      0 *:ftp                   *:*                     LISTEN

# netstat -tan
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:6000            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:21              0.0.0.0:*               LISTEN
```

9. TCP and UDP Applications

So far we have studied how to implement client server programs using socket system calls. In these programs we are exchanging our own messages or packets between client and servers. But there are so many standard client-server applications for different purposes. All these applications are defined through documents called ‘Request For Comments’ (RFCs). These RFC specifies the protocol (message formats exchanged) between clients and servers. Following are some standard client-server applications.

Protocol	**Purpose**
File Transfer Protocol (FTP) Trivial File Transfer Protocol (TFTP)	For transferring files between computers
Telnet and Rlogin	For logging into a remote server
Simple Mail Transfer Protocol (SMTP) Post Office Protocol - 3(POP-3)	For exchanging mails between computers
Simple Network Management Protocol (SNMP)	For monitoring and controlling the network equipment
Hyper Text Transfer Protocol (HTTP)	Web browsers and Web servers
Boot Protocol (BOOTP) Dynamic Host Control Protocol (DHCP)	For assigning IP addresses to hosts
Domain Name Service (DNS)	For managing host names

One should study as many standard client server applications as possible to understand the network application protocols. This knowledge will be useful to design and implement our own proprietary protocols.

Some protocols are based on UDP and other are based on TCP. Following gives this break-up.

TCP Application Protocols

FTP, Telnet, Rlogin, SMTP, POP-3, HTTP

UDP Application Protocols

TFTP, BOOTP, DHCP, SNMP, DNS

Trivial File Transfer Protocol (TFTP)

The TFTP is simplified protocol for transfer of files. This protocol runs over UDP sockets. TFTP protocol is designed for boot loading the operating systems or for boot loading embedded systems.

There are only 5 types of messages defined for the TFTP. All these messages are binary messages. These messages are

- Read request (RRQ)
- Write Requet (WRQ)
- Data (DATA)
- Acknowledgement (ACK)
- Error (ERROR)

Following are the format for these messages.

WRQ/ RRQ packet

Both WRQ and RRQ packets have the same format as shown in the figure. The opcode is 1 for RRQ and 2 for WRQ messages. Both messages contain null terminated filename string and null terminated mode string. Both strings are of variable length. The most common mode string is “octect”. Other are modes are “netascii” and “mail”.

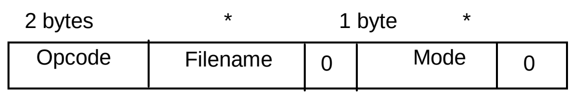


Figure: RRQ/WRQ packet

Data Packet

The opcode for DATA packet is 3. Every data packet contains a block number. This block number is useful to identify missed data packets or duplicated data packets. The number of data bytes in the message varies form 0 to maximum of 512 bytes. If number of data bytes is 512, then it is not the last data block. But if number of data bytes or 0 to 511, then it indicates that this is the last data block of the file and signals end of file transfer.

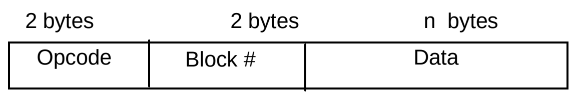


Figure: DATA packet

ACK Packet

All packets other than duplicate ACK’s and those used for termination are acknowledged unless a timeout occurs. Sending a DATA packet is an acknowledgement for the first ACK packet of the previous DATA packet. The WRQ and DATA packets are acknowledged by ACK or ERROR packets. While RRQ and ACK packets are acknowledged by DATA or ERROR packets. The following figure shows the format of ACK packet. The opcode for ACK command is 4.

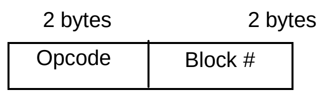


Figure: ACK packet

The block number in the ACK echoes the block number of the DATA packet being acknowledged. A WRQ is acknowledged with an ACK packet having a block number of zero.

ERROR Packet

The ERROR packet is shown below. An ERROR packet can be the acknowledgement of any other type of packet. The error code is an integer indicating the nature of the error. The error message is intended to by used by the user.

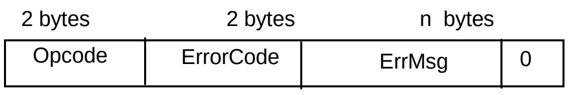


Figure: ERROR packet

Writing a file to TFTP Server

The TFTP client sends WRQ packet to the TFTP server. The TFTP server acknowledges WRQ packet by sending ACK with 0 block number. Now client will send first data packet with block number as 1. Server acknowledges with ACK packet with block number as 1. This continues till client sends a data packet with number of bytes less than 512. Once TFTP server receives packet with less than 512 bytes it assumes end of file transfer.

Reading a file from TFTP Server

The TFTP client sends RRQ packet to the TFTP server. The TFTP server sends DATA packet with block number as 1. Client acknowledges DATA packet by sending ACK with block number as 1. Server sends next data packet and client acknowledges ack packets. Once client receives data packet with less than 512 bytes, client detects this as last packet and acknowledges it.

Post Office Protocol – 3 (POP3)

POP3 server allows downloading of e-mails from central server (on which POP3 server is running) to the local machine. POP3 client running on the local machine connects to POP3 server and downloads mails.

When we take an Internet connection from Internet Service Provider (ISP) we will get an e-mail account. All the mails that come to our email account will be placed on the mail server of ISP. A POP-3 server will run on the ISP machine. Our computers at home will run POP-3 client program and downloads all the mails present on the server to our home machine.

POP-3 uses TCP connection. The port number of POP-3 server is 110. The messages exchanged between client and server are text messages only.

Following are the list of command messages that are sent by client to the server. Every command message is terminated by a carriage return and line feed (CR-LF) pair of characters. All command messages are ASCII (text) messages.

```
USER <username>  CR-LF
PASS <password>  CR-LF
STAT CR-LF
LIST    [msgNo] CR-LF
RETR <msgNo> CR-LF
DELE <msgNo> CR-LF

RSET CR-LF
QUIT CR-LF
```

The responses messages sent by server depends on the type of command. But all response messages contains status at the beginning of the message. The status part could be '+OK' or '-ERR'. We can classify response messages as two types. First one is single line response message and second type is multi line response messages. For single line responses, client can easily identify the end of response message because it contains only single line. For multi line response messages client should identify the last line of response. POP3 server sends last line of message as ".CR-LF". That means if last line contains just '.', then it is a last line.

```
$ telnet 192.168.0.20 110
Trying 192.168.0.20...
Connected to 192.168.0.20.
Escape character is '^]'.

S:   +OK POP3 depik20.depik.com v2003.83rh server ready
C:   user guest
S:   +OK User name accepted, password please
C:   pass guest123
S:   +OK Mailbox open, 3 messages
C:   stat
S:   +OK 3 1477
C:   list
S:   +OK Mailbox scan listing follows
S:   1 422
S:   2 637
S:   3 418
S:   .
C:   retr 1
S:   +OK 422 octets
S:   Return-Path: <guest@depik29.depik.com>
S:   Received: from depik29 (depik29.depik.com [192.168.0.29])
S:     by depik20.depik.com (8.12.10/8.12.10) with SMTP id m0P4h3XC004210
S:     for guest; Fri, 25 Jan 2008 10:16:47 +0530
S:   Date: Fri, 25 Jan 2008 10:13:03 +0530
S:   From: guest@depik29.depik.com
S:   Message-Id: <200801250446.m0P4h3XC004210@depik20.depik.com>
S:   Status: RO
S:
S:   Hi guest,
S:   this is sample
S:   message from guest.
S:   Thanks,
S:   guest
S:   .
C: retr 3
S: +OK 418 octets
S: Return-Path: <guest@depik20.depik.com>
S: Received: from depik20 (depik20.depik.com [192.168.0.20])
S:     by depik20.depik.com (8.12.10/8.12.10) with SMTP id m0P6gX73003172
S:     for guest; Fri, 25 Jan 2008 12:14:43 +0530
S: Date: Fri, 25 Jan 2008 12:12:33 +0530
S: From: Guest User <guest@depik20.depik.com>
S: Message-Id: <200801250644.m0P6gX73003172@depik20.depik.com>
S: Status: RO
S:
S: this is the test mail
S: sent by NV Rao & team
S: .
C: quit
S: +OK Sayonara
   Connection closed by foreign host.
```

Simple Mail Transfer Protocol (SMTP)

SMTP server is responsible for receiving the mails from the SMTP client programs. Similar to POP-3 server, SMTP server also accepts text (ASCII) messages.

```
$ telnet 192.168.0.20 25
Trying 192.168.0.20...
Connected to 192.168.0.20.
Escape character is '^]'.

S: 220 depik20.depik.com ESMTP Sendmail 8.12.10/8.12.10; Sat, 26 Jan 2008
09:50:09 +0530
C: helo box
S: 250 depik20.depik.com Hello [192.168.0.197], pleased to meet you
C: mail from: karasala@box
S: 250 2.1.0 karasala@box... Sender ok
C: rcpt to: guest
S: 250 2.1.5 guest... Recipient ok
C: data
S: 354 Enter mail, end with "." on a line by itself
C: Hi guest,
C:
C: I am sending this mail from
C: telnet program to show how to talk to
C: SMTP server through the telnet program.
C:
C: thanks,
C: karasala
C: .
S: 250 2.0.0 m0Q4K9E5003639 Message accepted for delivery
C: quit
S: 221 2.0.0 depik20.depik.com closing connection
Connection closed by foreign host.

$ telnet 192.168.0.20 110
Trying 192.168.0.20...
Connected to 192.168.0.20.
Escape character is '^]'.
S: +OK POP3 depik20.depik.com v2003.83rh server ready
C: user guest
S: +OK User name accepted, password please
C: pass guest123
S: +OK Mailbox open, 4 messages
C: list
S: +OK Mailbox scan listing follows
S: 1 422
S: 2 637
S: 3 418
S: 4 456
S: .
C: stat
S: +OK 4 1933
C: retr 4
S: +OK 456 octets
S: Return-Path: <karasala@box>
S: Received: from box ([192.168.0.197])
S:     by depik20.depik.com (8.12.10/8.12.10) with SMTP id m0Q4K9E5003639
S:     for guest; Sat, 26 Jan 2008 09:54:06 +0530
S: Date: Sat, 26 Jan 2008 09:50:09 +0530
```

```
S: From: karasala@box
S: Message-Id: <200801260424.m0Q4K9E5003639@depik20.depik.com>
S: Status:
S:
S: Hi guest,
S: I am sending this mail from
S: telnet program to show how to talk to
S: SMTP server through the telnet program.
S:
S: thanks,
S: karasala
S: .
C: quit
S: +OK Sayonara
Connection closed by foreign host.
```