UNIX PROGRAMMING

MODULE-4

CHANGING USER IDs AND GROUP IDs

- ✓ When programs need additional privileges or need to gain access to resources that they currently aren't allowed to access, need to change their user or group ID to an ID that has the appropriate privilege or access.
- ✓ When our programs need to lower their privileges or prevent access to certain resources, they do so by changing either their user ID or group ID to an ID without the privilege or ability access to the resource.

Explain setuid and setgid functions, Explain the ways to change user ids
imp

```
#include <unistd.h>
int     setuid(uid_t uid);
int     setgid(gid_t  gid);
```

- ✓ Both return:

  0 if OK,

  -1 on error
- ✓ There are rules for who can change the IDs.
- ✓ Let's consider only the user ID for now. (Everything we describe for the user ID also applies to the group ID.)
- ✓ If the process has superuser privileges, the setuid function sets the real user ID, effective user ID, and saved set-user-ID to uid.
- ✓ If the process does not have superuser privileges, but uid equals either the real user ID or the saved set-user- ID, setuidsets only the effective user ID to uid.
- ✓ The real user ID and the saved set-user-ID are not changed. If neither of these two conditions is true, errno is set to EPERM, and 1 is returned.
- ✓ Only a superuser process can change the real user ID. Normally, the real user ID is set by the login(1) program when we log in and never changes. Because login is a superuser process, it sets all three user IDs when it calls setuid.
- ✓ The effective user ID is set by the exec functions only if the set-user-ID bit is set for the program file.

- ✓ If the set-user-ID bit is not set, the exec functions leave the effective user ID as its current value. We can call setuid at any time to set the effective user ID to either the real user ID or the saved set-user-ID.
- ✓ The saved set-user-ID is copied from the effective user ID by exec. If the file's set-user-ID bit is set, this copy is saved after execstores the effective user ID from the file's user ID.

ID	exec		setuid(uid)	
	set-user-ID bit off	set-user-ID bit on	superuser	Unprivileged user
real user ID	unchanged	unchanged	set to uid	unchanged
effective user ID	unchanged	set from user ID of program file	set to uid	set to uid
saved set-user ID	copied from effective user ID	copied from effective user ID	set to uid	unchanged

setreuid and setregid Functions

- ✓ Swapping of the real user ID and the effective user ID with the setreuid function.

```
#include <unistd.h>
int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
```

- ✓ Both return :

  0 if OK

  -1 on error
- ✓ We can supply a value of 1 for any of the arguments to indicate that the corresponding ID should remain unchanged.
- ✓ The rule is simple: an unprivileged user can always swap between the real user ID and the effective user ID.
- ✓ This allows a set-user-ID program to swap to the user's normal permissions and swap back again later for set-user- ID operations.

seteuid and setegid functions :

- ✓ POSIX.1 includes the two functions seteuid and setegid. These functions are similar to setuid and setgid, but only the effective user ID or effective group ID is changed.

```
#include <unistd.h>
int  seteuid(uid_t uid);
int setegid(gid_t gid);
```

- ✓ Both return :

  0 if OK,

  -1 on error

- ✓ An unprivileged user can set its effective user ID to either its real user ID or its saved set-user-ID. For a privileged user, only the effective user ID is set to uid.

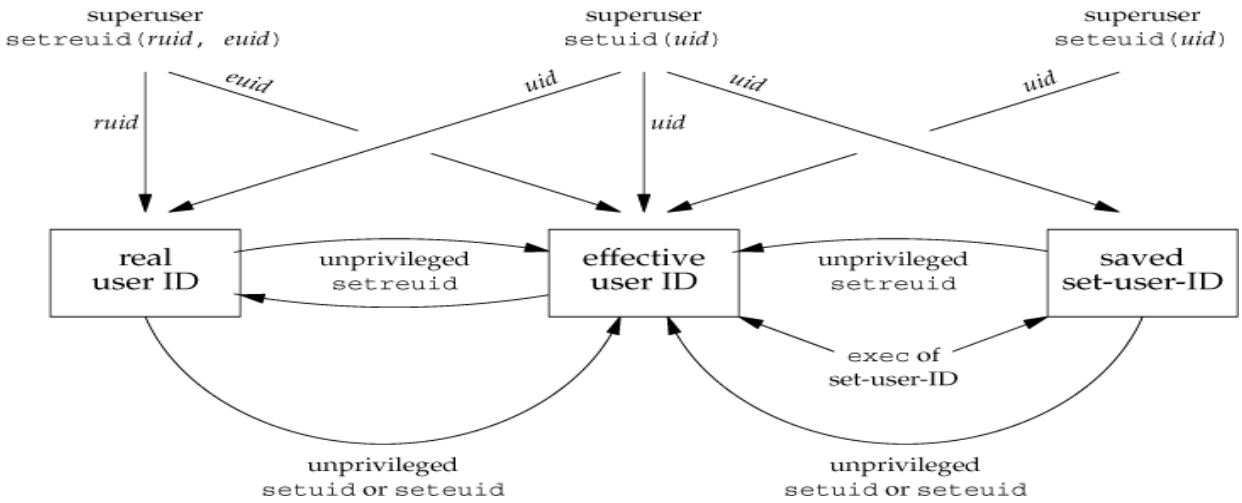


Figure: Summary of all the functions that set the various user Ids

Explain the following
(i)Interpreter files
(ii)process accounting
(iii)User identification

INTERPRETER FILES

- ✓ These files are text files that begin with a line of the form

  #! pathname [optional-argument]

- ✓ The space between the exclamation point and the pathname is optional. The most common of these interpreter files begin with the line

  #!/bin/sh

- ✓ The pathname is normally an absolute pathname, since no special operations are performed on it (i.e., PATH is not used). The recognition of these files is done within the kernel as part of processing the exec system call.

- ✓ The actual file that gets executed by the kernel is not the interpreter file, but the file specified by the pathname on the first line of the interpreter file. Be sure to differentiate between the interpreter filea text file that begins with #!and the interpreter, which is specified by the pathname on the first line of the interpreter file.

system FUNCTION

Prototype is:

```
#include <stdlib.h>
int system(const char *cmdstring);
```

- ✓ If cmdstring is a null pointer, system returns nonzero only if a command processor is available. This feature determines whether the system function is supported on a given operating system. Under the UNIX System, systemis always available. Because system is implemented by calling fork, exec, and waitpid, there are three types of return values.
- ✓ If either the forkfails or waitpidreturns an error other than EINTR, systemreturns 1 with errnoset to indicate the error.
- ✓ If the execfails, implying that the shell can't be executed, the return value is as if the shell had executed exit(127).
- ✓ Otherwise, all three functions fork, exec, and waitpid succeed, and the return value from systemis the termination status of the shell, in the format specified for waitpid

Program: The systemfunction, without signal handling

```
int system(const char *cmdstring)     /* version without signal handling */
{
        pid_t   pid;
        int     status;
        if (cmdstring == NULL)
                return(1);      /* always a command processor with UNIX */
        if ((pid = fork()) < 0)
         {
                status = -1;    /* probably out of processes */
          }
        else
                 if (pid == 0)
                {       /* child */
                        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
                        _exit(127);     /* execl error */
                }
```

```
                else
                {       /* parent */
                        while (waitpid(pid, &status, 0) < 0)
                        {
                         if (errno != EINTR)
                        {
                                status = -1; /* error other than EINTR from waitpid() */
                                break;

                        }
                        }
                }
                return(status);
        }
```

Program: Calling the system function

```
int main(void)
{
        int     status;
        if ((status = system("date")) < 0)
                err_sys("system() error");
                pr_exit(status);
        if ((status = system("nosuchcommand")) < 0)
                err_sys("system() error");
                pr_exit(status);
        if ((status = system("who; exit 44")) < 0)
                err_sys("system() error");
                pr_exit(status);
        exit(0);
}
```

PROCESS ACCOUNTING

write note on process accounting

- ✓ Most UNIX systems provide an option to do process accounting. When enabled, the kernel writes an accounting record each time a process terminates. These accounting records are typically a small amount of binary data with the name of the command, the amount of CPU time used, the user ID and group ID, the starting time, and so on.
- ✓ A superuser executes acct on with a pathname argument to enable accounting.

- ✓ The accounting records are written to the specified file, which is usually /var/account/acct. Accounting is turned off by executing acct on without any arguments.
- ✓ The data required for the accounting record, such as CPU times and number of characters transferred, is kept by the kernel in the process table and initialized whenever a new process is created, as in the child after a fork. Each accounting record is written when the process terminates.
- ✓ The accounting records correspond to processes, not programs. A new record is initialized by the kernel for the child after a fork, not when a new program is executed. The structure of the accounting records is defined in the header <sys/acct.h>and looks something like

typedefu_short comp_t; /* 3-bit base 8 exponent; 13-bit fraction */

```
struct acct
{
        char    ac_flag;        /* flag */
        char    ac_stat;        /* termination status (signal & core flag only) */
        uid_t   ac_uid;/* real user ID */
        gid_t   ac_gid;/* real group ID */
        dev_t   ac_tty;         /* controlling terminal */
        time_t ac_btime;        /* starting calendar time */
         comp_t ac_utime;       /* user CPU time (clock ticks) */
        comp_t ac_stime;        /* system CPU time (clock ticks) */
        comp_t ac_etime;        /* elapsed time (clock ticks) */
        comp_t ac_mem;          /* average memory usage */
        comp_t ac_io;           /* bytes transferred (by read and write) */
                                /* "blocks" on BSD systems */
        comp_t ac_rw;           /* blocks read or written */
                                        /* (not present on BSD systems) */
         char   ac_comm[8];  /* command name: [8] for Solaris, */
}
```

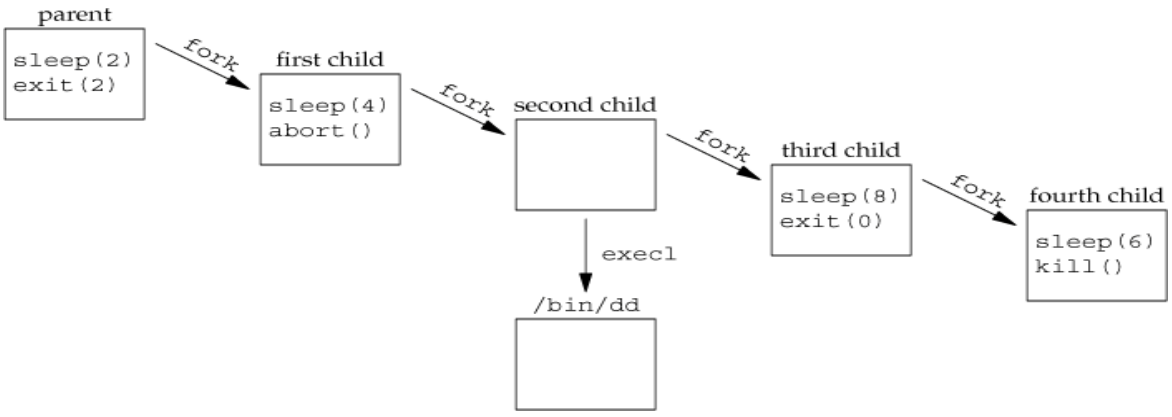
Values for ac_flagfrom accounting record

ac_flag	Description
AFORK	process is the result of fork, but never called exec
ASU	process used super user privileges
ACOMPAT	process used compatibility mode
ACORE	process dumped core
AXSIG	process was killed by a signal
AEXPND	expanded accounting entry

Program to generate accounting data

```
int main(void)
{
        pid_t    pid;
        if ((pid = fork()) < 0)
        err_sys("fork error");
        else
        if (pid != 0)
        {       /* parent */
                sleep(2);
                exit(2);/* terminate with exit status 2 */
        }
/* first child */
        if ((pid = fork()) < 0)
        err_sys("fork error");
        else
        if (pid != 0)
        {
                sleep(4);
                abort();/* terminate with core dump */
        }
/* second child */
        if ((pid = fork()) < 0)
        err_sys("fork error");
        else
        if (pid != 0)
        {
                execl("/bin/dd", "dd", "if=/etc/termcap", "of=/dev/null", NULL); exit(7);
                        /* shouldn't get here */
```

```
        }
/* third child */
        if ((pid = fork()) < 0)
                err_sys("fork error");
                else
                if (pid != 0)
                {
                        sleep(8);
                        exit(0);/* normal exit */
                }
/* fourth child */
        sleep(6);
        kill(getpid(), SIGKILL);        /* terminate w/signal, no core dump */
         exit(6);         /* shouldn't get here */
}
```



Process structure for accounting example

USER IDENTIFICATION

- ✓ Any process can find out its real and effective user ID and group ID. Sometimes, however, we want to find out the login name of the user who's running the program. We could call getpwuid(getuid()), but what if a single user has multiple login names, each with the same user ID? (A person might have multiple entries in the password file with the same user ID to have a different login shell for each entry.) The system normally keeps track of the name we log in and the getloginfunction provides a way to fetch that login name.

  #include <unistd.h>

  char *getlogin(void);

- ✓ Returns : pointer to string giving login name if OK, NULL on error
- ✓ This function can fail if the process is not attached to a terminal that a user logged in to.

PROCESS TIMES

- ✓ The three times that we can measure: wall clock time, user CPU time, and system CPU time. Any process can call the timesfunction to obtain these values for itself and any terminated children.

```
#include <sys/times.h>
clock_t times(struct tms *buf);
```

- ✓ Returns: elapsed wall clock time in clock ticks if OK, 1 on error
- ✓ This function fills in the tms structure pointed to by buf:
- ✓ This function fills in the tms structure pointed to by buf:

```
struct tms
    {
    clock_t tms_utime; /* user CPU time */
    clock_t tms_stime; /* system CPU time */
    clock_t tms_cutime; /* user CPU time, terminated children */
    clock_t tms_cstime; /* system CPU time, terminated children */
    };
```

Write a short note on System V IPC imp

INTERPROCESS COMMUNICATION

INTRODUCTION

10) Define Inter Process Communication (IPC). List the IPC types supported in UNIX system.

- ✓ IPC enables one application to control another application, and for several applications to share the same data without interfering with one another. IPC is required in all multiprocessing systems, but it is not generally supported by single-process operating systems.

The various forms of IPC that are supported on a UNIX system are as follows :

1) Pipes.
2) FIFO's
3) Message queues.
4) Shared memory.
5) Semaphores.
6) Sockets.
7) STREAMS.

1. Explain the following with example wrt IPC Method -20M
(i)Pipes-With a program to send data from parent to child
(ii)popen,pclose functions

(iii)FIFOs
(iv)Semaphores
(v)Message queue imp

- ✓ The first Five forms of IPC are usually restricted to IPC between processes on the same host.
- ✓ The final two i.e. Sockets and STREAMS are the only two that are generally supported for IPC between processes on different hosts.

PIPES

11) List the limitations of pipe IPC.

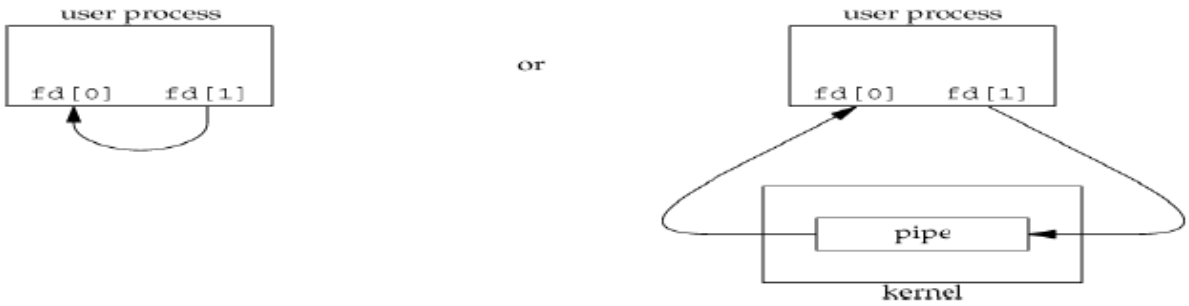
- ✓ Pipes are the oldest form of UNIX System IPC. Pipes have two limitations.
- ✓ Historically, they have been half duplex (i.e., data flows in only one direction).
- ✓ Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.
- ✓ A pipe is created by calling the pipe function.

```
#include <unistd.h>
int pipe(int filedes[2]);
```

- ✓ Returns: 0 if OK, 1 on error.

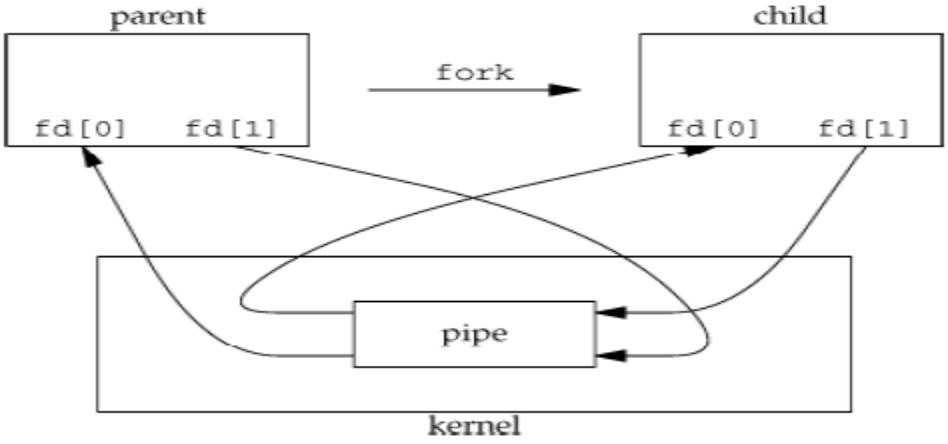
- ✓ Two file descriptors are returned through the filedes argument: filedes[0] is open for reading, and filedes[1] is open for writing. The output of filedes[1] is the input for filedes[0].
- ✓ Two ways to picture a half-duplex pipe are shown in below Figure . The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.

Fig: Two ways to view a half-duplex pipe



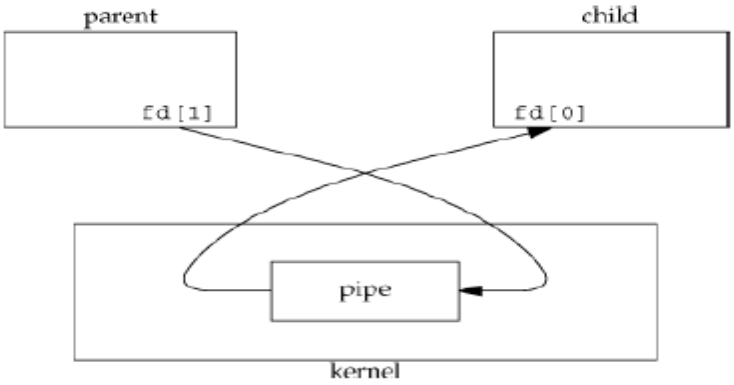
- ✓ A pipe in a single process is next to useless. Normally, the process that calls pipe then calls fork, creating an IPC channel from the parent to the child or vice versa. Below figure shows this scenario.

Fig: Half-duplex pipe after a `fork`



- ✓ What happens after the fork depends on which direction of data flow we want. For a pipe from the parent to the child, the parent closes the read end of the pipe (fd[0]), and the child closes the write end (fd[1]). Below figure shows the resulting arrangement of descriptors.

Fig: Pipe from parent to child



✓ For a pipe from the child to the parent, the parent closes fd[1], and the child closes fd[0].

When one end of a pipe is closed, the following two rules apply.

❖ If we read from a pipe whose write end has been closed, read returns 0 to indicate an end of file after all the data has been read.

❖ If we write to a pipe whose read end has been closed, the signal SIGPIPE is generated. If we either ignore the signal or catch it and return from the signal handler, write returns 1 with errno set to EPIPE.

PROGRAM 1: shows the code to create a pipe between a parent and its child and to send data down the pipe.

16) Develop a code snippet where parent sends "hello RNSIT" message to the child process through the pipe. The child on receiving this message should display it on the standard output.

```
#include "apue.h"
Int main(void)
{
        int n;
        int fd[2];
        pid_t pid;
        char line[MAXLINE];
        if (pipe(fd) < 0)
        err_sys("pipe error");
        if ((pid = fork()) < 0)
        {
                err_sys("fork error");
        }
        else
                if (pid > 0)
                {                       /* parent */
                        close(fd[0]);
```

```
                write(fd[1], "hello world\n", 12);
        }
        else
                {               /* child */
                close(fd[1]);
                n = read(fd[0], line, MAXLINE);
                write(STDOUT_FILENO, line, n);
                }
        exit(0);
}
```

PROGRAM 1: shows the code to create a pipe between a parent and its child and to send data down the pipe.

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main()
{
        int pid,n;
        char buf[20];
        int fd[2];
        pipe(fd);
        pid = fork();
        if ( pid > 0)
        {
                printf("\n Parent process");
                close(fd[0]);
                write(fd[1],"Welcome to all",15);
        }
      if(pid == 0)
        {

            printf("\n Child process");
            close(fd[1]);
            n=read(fd[0],buf,15);
            buf[n]='\0';
            printf("\n received data is %s\n",buf);
        }
        exit(0);

}
```

popen AND pclose FUNCTIONS

- ✓ Since a common operation is to create a pipe to another process, to either read its output or send it input, the standard I/O library has historically provided the popen and pclose functions.
- ✓ These two functions handle all the dirty work that we've been doing ourselves:
  - creating a pipe, forking a child, closing the unused ends of the pipe, executing a shell to run the command
  - waiting for the command to terminate.

```
#include <stdio.h>
FILE *popen(const char *cmdstring, const char *type);
```

- ✓ *Returns: file pointer if OK, NULL on error*

```
int pclose(FILE *fp);
```

- ✓ Returns: termination status of cmdstring, or 1 on error
- ✓ The function popen does a fork and exec to execute the cmdstring, and returns a standard I/O file pointer. If type is "r", the file pointer is connected to the standard output of cmdstring

Fig: Result of fp = popen(cmdstring, "r")

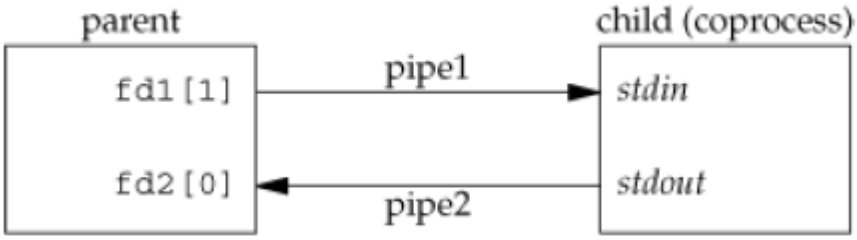
- ✓ If type is "w", the file pointer is connected to the standard input of cmdstring, as shown:

Fig: Result of fp = popen(cmdstring, "w")

COPROCESSES

- ✓ A UNIX system filter is a program that reads from standard input and writes to standard output. Filters are normally connected linearly in shell pipelines.
- ✓ A filter becomes a coprocess when the same program generates the filter's input and reads the filter's output. A coprocess normally runs in the background from a shell, and its standard input and standard output are connected to another program using a pipe.
- ✓ The process creates two pipes: one is the standard input of the coprocess, and the other is the standard output of the coprocess. Below figure shows this arrangement.

Fig: Driving a coprocess by writing its standard input and reading its standard output



Program: Simple filter to add two numbers

```
#include "apue.h"
Int main(void)
{
        int n, int1, int2;
        char line[MAXLINE];
        while ((n = read(STDIN_FILENO, line, MAXLINE)) > 0)
        {
                line[n] = 0; /* null terminate */
                if (sscanf(line, "%d%d", &int1, &int2) == 2)
                {
                        sprintf(line, "%d\n", int1 + int2);
                        n = strlen(line);
                        if (write(STDOUT_FILENO, line, n) != n)
                        err_sys("write error");
                }
                Else
                {
                        if (write(STDOUT_FILENO, "invalid args\n", 13) != 13)
                        err_sys("write error");
                }
        }
        exit(0);
}
```

FIFOs

- ✓ FIFOs are sometimes called named pipes. Pipes can be used only between related processes when a common ancestor has created the pipe.

```
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

- ✓ Returns: 0 if OK, 1 on error
- ✓ Once we have used mkfifo to create a FIFO, we open it using open. When we open a FIFO, the nonblocking flag (O_NONBLOCK) affects what happens.
- ✓ In the normal case (O_NONBLOCK not specified), an open for read-only blocks until some other process opens the FIFO for writing. Similarly, an open for write-only blocks until some other process opens the FIFO for reading.
- ✓ If O_NONBLOCK is specified, an open for read-only returns immediately. But an open for write-only returns 1 with errno set to ENXIO if no process has the FIFO open for reading.
- ✓ There are two uses for FIFOs.
  - I. FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.
  - II. FIFOs are used as rendezvous points in client-server applications to pass data between the clients and the servers.

Example Using FIFOs to Duplicate Output Streams

- ✓ FIFOs can be used to duplicate an output stream in a series of shell commands. This prevents writing the data to an intermediate disk file. Consider a procedure that needs to process a filtered input stream twice. Below figure shows this arrangement.

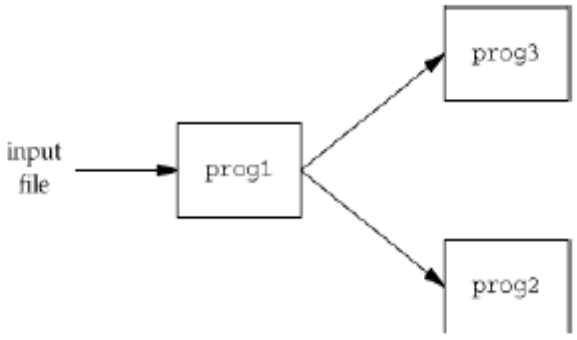


Fig: Procedure that processes a filtered input stream twice

- ✓ With a FIFO and the UNIX program tee(1), we can accomplish this procedure without using a temporary file. (The tee program copies its standard input to both its standard output and to the file named on its command line.)

mkfifo fifo1 prog3 < fifo1 & prog1 < infile | tee fifo1 | prog2

- ✓ We create the FIFO and then start prog3 in the background, reading from the FIFO. We then start prog1 and use tee to send its input to both the FIFO and prog2. Below figure shows the process arrangement.

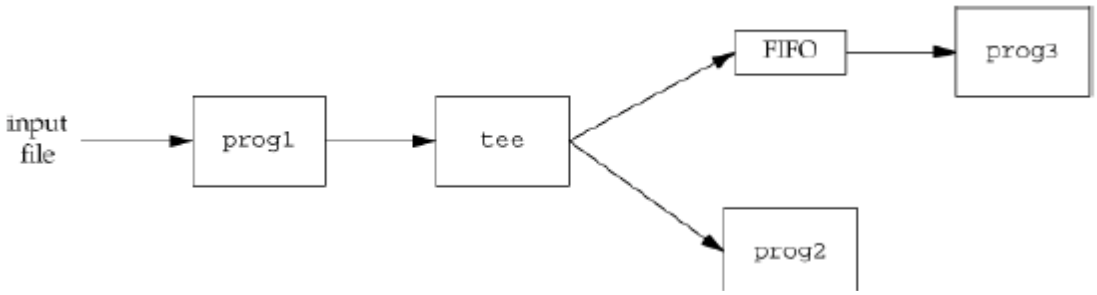


Fig: Using a FIFO and tee to send a stream to two different processes

Example Client-Server Communication Using a FIFO

- ✓ FIFO's can be used to send data between a client and a server. If we have a server that is contacted by numerous clients, each client can write its request to a well-known FIFO that the server creates. Since there are multiple writers for the FIFO, the requests sent by the clients to the server need to be less than PIPE_BUF bytes in size.
- ✓ This prevents any interleaving of the client writes. The problem in using FIFOs for this type of client server communication is how to send replies back from the server to each client.
- ✓ A single FIFO can't be used, as the clients would never know when to read their response versus responses for other clients. One solution is for each client to send its process ID with the request. The server then creates a unique FIFO for each client, using a pathname based on the client's process ID.
- ✓ For example, the server can create a FIFO with the name /vtu/ ser.XXXXX, where XXXXX is replaced with the client's process ID. This arrangement works, although it is impossible for the server to tell whether a client crashes. This causes the client-specific FIFOs to be left in the file system.

- ✓ The server also must catch SIGPIPE, since it's possible for a client to send a request and terminate before reading the response, leaving the client-specific FIFO with one writer (the server) and no reader.

Explain the following in detail
(i)Client-server Interaction using FIFO
(ii)Steam Pipes & different ways to view them
imp

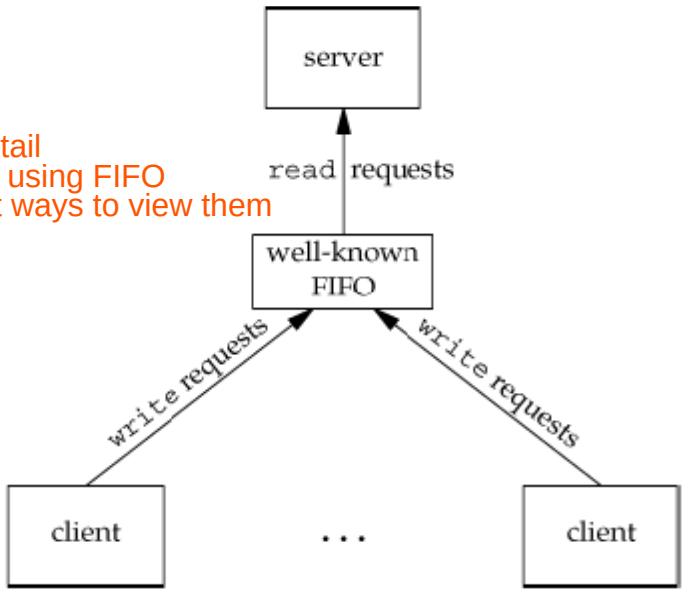


Fig: Clients sending requests to a server using a FIFO

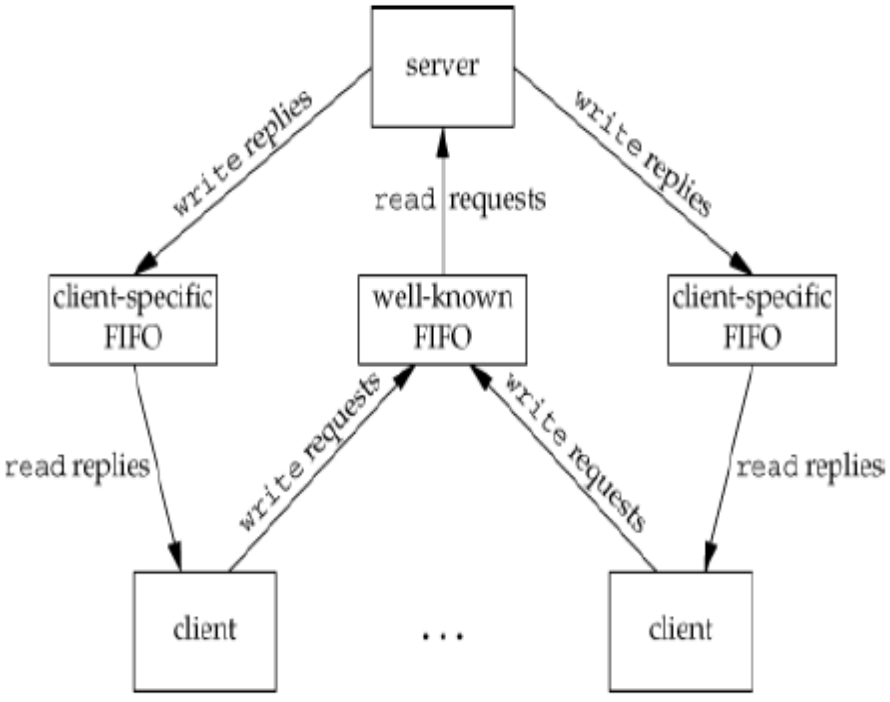


Fig: **Client-server communication using FIFOs**

SERVER PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<string.h>
#include<sys/stat.h>
#include<fcntl.h>
int main()
{
        int fd1,fd2,n;
        char buf[50];
        if (mkfifo("IPC1",S_IFIFO | 0666) < 0)
        {
                printf("\n Error in creating first Fifo");
                exit(0);
        }

        if(mkfifo("IPC2",S_IFIFO | 0666) < 0)
        {
                printf("\n Error in creating seond fifo");
                exit(0);
        }
        fd1 = open("IPC1",O_WRONLY);
        if (fd1 == -1)
        {
                printf("\n Error in opening first fifo file");
                exit(0);
        }

        write(fd1,"hello Client",15);

      fd2 = open("IPC2",O_RDONLY);
        if(fd2 == -1)
        {
                printf("\n Error in opening seconf fifo file");
                exit(0);
        }
      n=read(fd2,buf,50);
        buf[n]='\0';
        printf("\n The data received from client is %s\n",buf);

        exit(0);
}
```

CLIENT PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<string.h>
#include<sys/stat.h>
#include<fcntl.h>
int main()
{
        int fd1,fd2,n;
        char buf[50];
        fd1 = open("IPC1",O_RDONLY);
        if (fd1 == -1)
        {
                printf("\n Error in opening first fifo file");
                exit(0);
        }
      n=read(fd1,buf,20);
        buf[n]='\0';
        printf("\n The data received from server is %s\n",buf);

        fd2 = open("IPC2",O_WRONLY);
        if(fd2 == -1)
        {
                printf("\n Error in opening seconf fifo file");
                exit(0);
        }
        write(fd2,"Hello Server",15);
        exit(0);
}
```

MESSAGE QUEUES

12) Define Message Queue. Which functions are used to carry out IPC using message queue?

- ✓ A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier.
- ✓ We'll call the message queue just a queue and its identifier a queue ID.
- ✓ A new queue is created or an existing queue opened by `msgget`. New messages are added to the end of a queue by `msgsnd`. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to `msgsnd` when the message is added to a queue.
- ✓ Messages are fetched from a queue by `msgrcv`. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

Each queue has the following msqid_ds structure associated with it:

struct msqid_ds

```
{
struct ipc_perm msg_perm;

msgqnum_t msg_qnum;          /* # of messages on queue */

msglen_t msg_qbytes;              /* max # of bytes on queue */

pid_t msg_lspid;             /* pid of last msgsnd() */

pid_t msg_lrpid;             /* pid of last msgrcv() */
time_t  msg_stime;           /*last-msgsnd() time */
time_t  msg_rtime;           /* last-msgrcv() time */
time_t  msgctime;            /*last-change time */
.
.
};
```

- ✓ This structure defines the current status of the queue.
- ✓ The first function normally called is msgget to either open an existing queue or create a new queue.

```
#include <sys/msg.h>
int msgget(key_t key, int flag);
```

- ✓ Returns: message queue ID if OK, 1 on error

- ✓ When a new queue is created, the following members of the msqid_ds structure are initialized.
  - ❖ The ipc_perm structure is initialized. The mode member of this structure is set to the corresponding permission bits of flag.
  - ❖ msg_qnum, msg_lspid, msg_lrpid, msg_stime, and msg_rtime are all set to 0.
  - ❖ msg_ctime is set to the current time.
  - ❖ msg_qbytes is set to the system limit.

- ✓ On success, msgget returns the non-negative queue ID. This value is then used with the other three message queue functions.
- ✓ The msgctl function performs various operations on a queue.

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf );
```

- ✓ Returns: 0 if OK, 1 on error.
- ✓ The cmd argument specifies the command to be performed on the queue specified by msqid.

POSIX:XSI values for the cmd parameter of msgctl	
cmd	description
IPC_RMID	Remove the message queue msqid and destroy the corresponding msqid_ds
IPC_SET	Set members of the msqid_ds data structure from buf
IPC_STAT	copy members of the msqid_ds data structure into buf

- ✓ Data is placed onto a message queue by calling msgsnd.

```
#include <sys/msg.h>
int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
```

- ✓ Returns: 0 if OK, 1 on error.
- ✓ Each message is composed of a positive long integer type field, a non-negative length (nbytes), and the actual data bytes (corresponding to the length). Messages are always placed at the end of the queue.
- ✓ The ptr argument points to a long integer that contains the positive integer message type, and it is immediately followed by the message data. (There is no message data if nbytes is 0.) If the largest message we send is 512 bytes, we can define the following structure:

```
struct mymesg
{
long mtype;              /* positive message type */
char mtext[512];         /* message data, of length nbytes */
};
```

- ✓ The ptr argument is then a pointer to a mymesg structure. The message type can be used by the receiver to fetch messages in an order other than first in, first out.
- ✓ Messages are retrieved from a queue by msgrcv.

```
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
```

- ✓ Returns: size of data portion of message if OK, 1 on error.

- ✓ The type argument lets us specify which message we want.

type == 0	The first message on the queue is returned
type > 0	The first message on the queue whose message type equals type is returned
type < 0	The first message on the queue whose message type is the lowest value less than or equal to the absolute value of type is returned.

SENDER PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#define SIZE 10

struct msgbuf
{
        long type;
        char mtext[SIZE];
};

int main()
{
        int msgid,len;
        key_t key=2233;
        struct msgbuf sbuf;
        msgid = msgget(key,IPC_CREAT | 0666);
        if (msgid < 0)
        {
                printf("\n error in creating msg queue");
                exit(0);
        }
        //sbuf.type=2;
        printf("\n Read the msg to be sent\n");
      scanf("%s",sbuf.mtext);
        len = strlen(sbuf.mtext)+1;
        if(msgsnd(msgid,&sbuf,len,IPC_NOWAIT) < 0)
        {
                printf("\n error in sending message");
                exit(0);
        }
        else
                printf("\n Msg send successfully");
        exit(0);
}
```

RECEIVER PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#define SIZE 10

struct msgbuf
{
        long mtype;
        char mtext[SIZE];
};

int main()
{
        int msgid,len;
        key_t key=2233;
        struct msgbuf rbuf;
        msgid = msgget(key,0666);
        if (msgid < 0)
        {
                printf("\n error in creating msg queue");
                exit(0);
        }

        if(msgrcv(msgid,&rbuf,SIZE,0,0) < 0)
        {
                printf("\n error in receving message");
                exit(0);
        }
        else
                printf("\n Msg received is %s\n",rbuf.mtext);
        exit(0);
}
```

SEMAPHORES

- ✓ A semaphore is a counter used to provide access to a shared data object for multiple processes.
- ✓ To obtain a shared resource, a process needs to do the following:
  - i. Test the semaphore that controls the resource.
  - ii. If the value of the semaphore is positive, the process can use the resource. In this case, the process decrements the semaphore value by 1, indicating that it has used one unit of the resource.
  - iii. Otherwise, if the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0. When the process wakes up, it returns to step 1.

- ✓ When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1. If any other processes are asleep, waiting for the semaphore, they are awakened.
- ✓ A common form of semaphore is called a ***binary semaphore***. It controls a single resource, and its value is initialized to 1. In general, however, a semaphore can be initialized to any positive value, with the value indicating how many units of the shared resource are available for sharing.
- ✓ XSI semaphores are, unfortunately, more complicated than this. Three features contribute to this unnecessary complication.
  - i. A semaphore is not simply a single non-negative value. Instead, we have to define a semaphore as a set of one or more semaphore values. When we create a semaphore, we specify the number of values in the set.
  - ii. The creation of a semaphore (semget) is independent of its initialization (semctl). This is a fatal flaw, since we cannot atomically create a new semaphore set and initialize all the values in the set.
  - iii. Since all forms of XSI IPC remain in existence even when no process is using them, we have to worry about a program that terminates without releasing the semaphores it has been allocated. The undo feature that we describe later is supposed to handle this.
- ✓ The kernel maintains a semid_ds structure for each semaphore set:

```
struct semid_ds {
struct ipc_perm sem_perm;
 unsigned short sem_nsems;          /* # of semaphores in set */
time_t sem_otime;                    /* last-semop() time */
time_t sem_ctime;                    /* last-change time */
.
.
};
```

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

```
struct {
unsigned short semval;                  /* semaphore value, always >= 0 */
pid_t sempid;                        /* pid for last operation */
unsigned short semncnt;              /* # processes awaiting semval>curval */
```

```
unsigned short semzcnt;          /* # processes awaiting semval==0 */
.
.
};
```

- ✓ The first function to call is semget to obtain a semaphore ID.

```
#include <sys/sem.h>
int semget(key_t key, int nsems, int flag);
```

- ✓ Returns: semaphore ID if OK, 1 on error
- ✓ When a new set is created, the following members of the semid_ds structure are initialized. The ipc_perm structure is initialized. The mode member of this structure is set to the corresponding
  - ❖ permission bits of flag.
  - ❖ sem_otime is set to 0.
  - ❖ sem_ctime is set to the current time.
  - ❖ sem_nsems is set to nsems.

- ✓ The number of semaphores in the set is nsems. If a new set is being created (typically in the server), we must specify nsems. If we are referencing an existing set (a client), we can specify nsems as 0.
- ✓ The semctl function is the catchall for various semaphore operations

```
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd,... /* union semun arg */);
```

- ✓ The fourth argument is optional, depending on the command requested, and if present, is of type semun, a union of various command-specific arguments:

```
union semun
{
int val;                    /* for SETVAL */
struct semid_ds *buf; /* for IPC_STAT and IPC_SET */
unsigned short *array; /* for GETALL and SETALL */
};
```

POSIX:XSI values for the command parameter of semctl	
Cmd	description
GETALL	Return values of the semaphore set in arg.array
GETVAL	Return values of the specific semaphore element
GETPID	Return process ID of last process to manipulate element
GETNCNT	Return number of process waiting for element to increment
GETZCNT	Return number of process waiting for element to become 0
IPC_RMID	Remove semaphore set identified by semid
IPC_SET	Set permission of the semaphore set from arg.buf
IPC_STAT	Copy members to semid_ds of semaphore set semid into arg.buf
SETALL	Set values of semaphore set from arg.array
SETVAL	Set values of a specific semaphore element to arg.val

- ✓ The *cmd* argument specifies one of the above ten commands to be performed on the set specified by semid. The function semop atomically performs an array of operations on a semaphore set.

```
#include <sys/sem.h>
int semop(int semid, struct sembuf semoparray[], size_t nops);
```

- ✓ Returns: 0 if OK, 1 on error.
- ✓ The semoparray argument is a pointer to an array of semaphore operations, represented by sembuf structures:

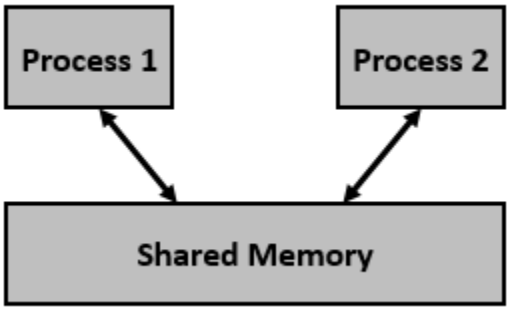
```
struct sembuf {
unsigned short sem_num /* member # in set (0, 1, ..., nsems-1) */
short sem_op;          /* operation (negative, 0, or positive) */
short sem_flg;                 /* IPC_NOWAIT, SEM_UNDO */
};
```

- ✓ The nops argument specifies the number of operations (elements) in the array.
- ✓ The sem_op element operations are values specifying the amount by which the semaphore value is to be changed.
  - ❖ If sem_op is an integer ***greater than zero***, semop adds the value to the corresponding semaphore element value and awakens all processes that are waiting for the element to increase.
  - ❖ If sem_op is ***0*** and the semaphore element value is not 0, semop blocks the calling process (waiting for 0) and increments the count of processes waiting for a zero value of that element.
  - ❖ If sem_op is a ***negative*** number, semop adds the sem_op value to the corresponding semaphore element value provided that the result would not be negative. If the operation would make the element value negative, semop blocks the process on the event that the semaphore element value increases. If the resulting value is 0, semop wakes the processes waiting for 0.

SHARED MEMORY

these are also called as system V Ipc

- ✓ Shared memory is a memory shared between two or more processes. However, why do we need to share memory or some other means of communication?
- ✓ To reiterate, each process has its own address space, if any process wants to communicate with some information from its own address space to other processes, then it is only possible with IPC (inter process communication) techniques.
- ✓ Inter-related process communication is performed using Pipes or Named Pipes. Unrelated processes (say one process running in one terminal and another process in another terminal) communication can be performed using Named Pipes or through popular IPC techniques of Shared Memory and Message Queues.



- ✓ Communicate between two or more processes, we use shared memory but before using the shared memory the following steps are required.
  - ❖ Create the shared memory segment or use an already created shared memory segment (shmget())
  - ❖ Attach the process to the already created shared memory segment (shmat())
  - ❖ Detach the process from the already attached shared memory segment (shmdt())
  - ❖ Control operations on the shared memory segment (shmctl())

Let us look at a few details of the system calls related to shared memory.

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

#include <sys/shm.h>

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

- ✓ The above system call creates or allocates a System V shared memory segment. The arguments that need to be passed are as follows −
- ✓ The **first argument, key,** recognizes the shared memory segment. The key can be either an arbitrary value or one that can be derived from the library function ftok(). The key can also be IPC_PRIVATE, means, running processes as server and client (parent and child relationship) i.e., inter-related process communication.
- ✓ If the client wants to use shared memory with this key, then it must be a child process of the server. Also, the child process needs to be created after the parent has obtained a shared memory.
- ✓ The **second argument, size,** is the size of the shared memory segment rounded to multiple of PAGE_SIZE.
- ✓ The **third argument, shmflg,** specifies the required shared memory flag/s such as IPC_CREAT (creating new segment) or IPC_EXCL (Used with IPC_CREAT to create new segment and the call fails, if the segment already exists). Need to pass the permissions as well.
- ✓ This call would return a valid shared memory identifier (used for further calls of shared memory) on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

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

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

- ✓ The above system call performs shared memory operation for System V shared memory segment i.e., attaching a shared memory segment to the address space of the calling process. The arguments that need to be passed are as follows −
- ✓ **The first argument, shmid,** is the identifier of the shared memory segment. This id is the shared memory identifier, which is the return value of shmget() system call.
- ✓ **The second argument, shmaddr,** is to specify the attaching address. If shmaddr is NULL, the system by default chooses the suitable address to attach the segment.
- ✓ If shmaddr is not NULL and SHM_RND is specified in shmflg, the attach is equal to the address of the nearest multiple of SHMLBA (Lower Boundary Address). Otherwise, shmaddr must be a page aligned address at which the shared memory attachment occurs/starts.
- ✓ **The third argument, shmflg,** specifies the required shared memory flag/s such as SHM_RND (rounding off address to SHMLBA) or SHM_EXEC (allows the contents of segment to be executed) or SHM_RDONLY (attaches the segment for read-only purpose, by default it is read-write) or SHM_REMAP (replaces the existing mapping in the range specified by shmaddr and continuing till the end of segment).
- ✓ This call would return the address of attached shared memory segment on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

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

int shmdt(const void *shmaddr)
```

- ✓ The above system call performs shared memory operation for System V shared memory segment of detaching the shared memory segment from the address space of the calling process. The argument that needs to be passed is −
- ✓ The argument, shmaddr, is the address of shared memory segment to be detached. The to-be-detached segment must be the address returned by the shmat() system call.
- ✓ This call would return 0 on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

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

int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

- ✓ The above system call performs control operation for a System V shared memory segment. The following arguments needs to be passed −
- ✓ The first argument, shmid, is the identifier of the shared memory segment. This id is the shared memory identifier, which is the return value of shmget() system call.
- ✓ The second argument, cmd, is the command to perform the required control operation on the shared memory segment.
- ✓ Valid values for cmd are −
  - ❖ **IPC_STAT** − Copies the information of the current values of each member of struct shmid_ds to the passed structure pointed by buf. This command requires read permission to the shared memory segment.
  - ❖ **IPC_SET** − Sets the user ID, group ID of the owner, permissions, etc. pointed to by structure buf.
  - ❖ **IPC_RMID** − Marks the segment to be destroyed. The segment is destroyed only after the last process has detached it.
  - ❖ **IPC_INFO** − Returns the information about the shared memory limits and parameters in the structure pointed by buf.
  - ❖ **SHM_INFO** − Returns a shm_info structure containing information about the consumed system resources by the shared memory.
- ✓ The third argument, buf, is a pointer to the shared memory structure named struct shmid_ds. The values of this structure would be used for either set or get as per cmd.
- ✓ This call returns the value depending upon the passed command. Upon success of IPC_INFO and SHM_INFO or SHM_STAT returns the index or identifier of the shared memory segment or 0 for other operations and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

SERVER PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#define SIZE 25
int main()
{
        int shmid;
        char *sm;
        key_t key = 3344;
        shmid = shmget(key,SIZE,IPC_CREAT | 0666);
        if(shmid < 0)
        {
                printf("\n error in creating shared memory");
```

```
                exit(0);
        }
        sm = shmat(shmid,NULL,0);
        strcpy(sm,"Welcome to shared memory1");
        while(*sm != '$')
                sleep(2);
        puts(sm);
        exit(0);
}
```

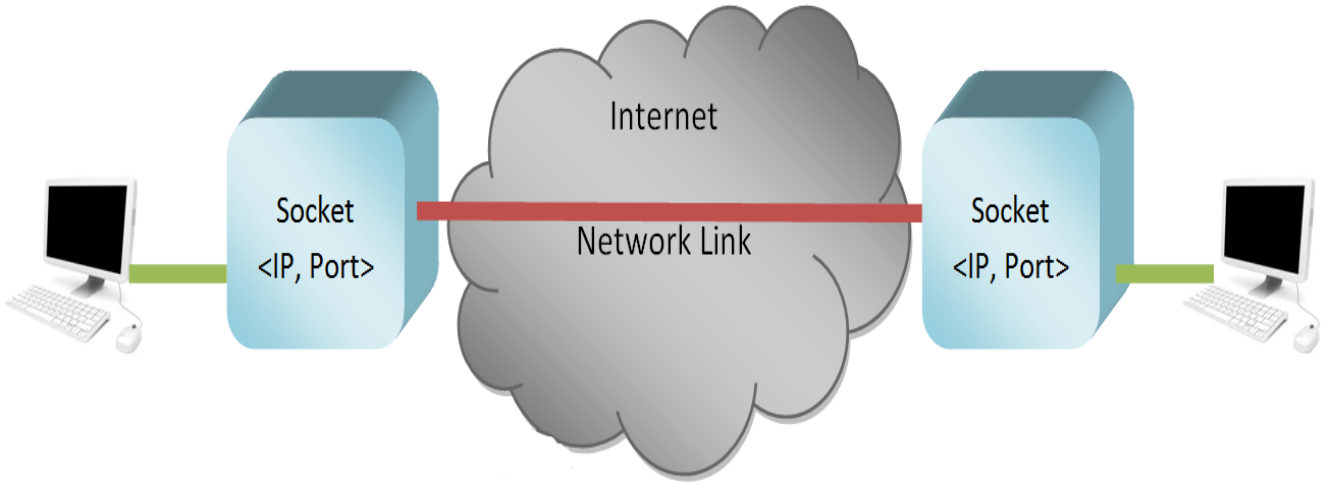
CLIENT PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#define SIZE 25
int main()
{
        int shmid;
        char *sm;
        key_t key = 3344;
        shmid = shmget(key,SIZE, 0666);
        if(shmid < 0)
        {
                printf("\n error in creating shared memory");
                exit(0);
        }
        sm = shmat(shmid,NULL,0);
      puts(sm);
        *sm = '$';
        exit(0);
}
```

SOCKETS

explain socket function

- ✓ Is a file type in UNIX
- ✓ End point for communication
- ✓ sockets are identified with IP address and Port Number
- ✓ Sockets allow communication between two different processes on the same or different machines. To be more precise, it's a way to talk to other computers using standard Unix file descriptors. In Unix, every I/O action is done by writing or reading a file descriptor.



Socket Types

- ✓ There are four types of sockets available to the users. The first two are most commonly used and the last two are rarely used.
- ✓ Processes are presumed to communicate only between sockets of the same type but there is no restriction that prevents communication between sockets of different types.
  - ❖ **Stream Sockets** – Delivery in a networked environment is guaranteed. If you send through the stream socket three items "A, B, C", they will arrive in the same order – "A, B, C". These sockets use TCP (Transmission Control Protocol) for data transmission. If delivery is impossible, the sender receives an error indicator. Data records do not have any boundaries.
  - ❖ **Datagram Sockets** – Delivery in a networked environment is not guaranteed. They're connectionless because you don't need to have an open connection as in

Stream Sockets − you build a packet with the destination information and send it out. They use UDP (User Datagram Protocol).

- **Raw Sockets** − These provide users access to the underlying communication protocols, which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more cryptic facilities of an existing protocol.
- **Sequenced Packet Sockets** − They are similar to a stream socket, with the exception that record boundaries are preserved. This interface is provided only as a part of the Network Systems (NS) socket abstraction, and is very important in most serious NS applications. Sequenced-packet sockets allow the user to manipulate the Sequence Packet Protocol (SPP) or Internet Datagram Protocol (IDP) headers on a packet or a group of packets, either by writing a prototype header along with whatever data is to be sent, or by specifying a default header to be used with all outgoing data, and allows the user to receive the headers on incoming packets.

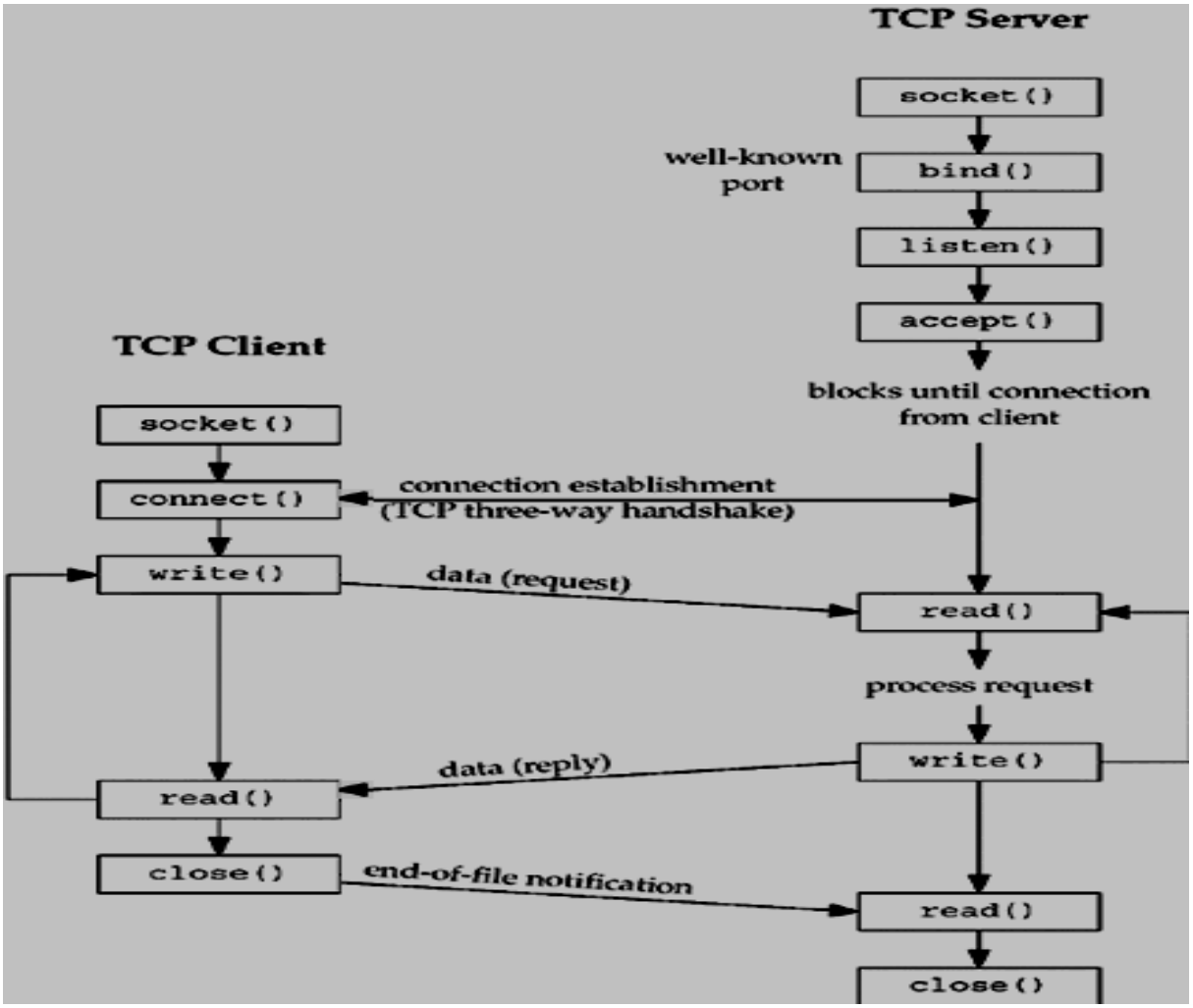
✓ The system calls for establishing a connection are somewhat different for the client and the server, but both involve the basic construct of a socket. Both the processes establish their own sockets.

✓ The steps involved in establishing a socket on the client side are as follows −

- Create a socket with the **socket()** system call.
- Connect the socket to the address of the server using the **connect()** system call.
- Send and receive data. There are a number of ways to do this, but the simplest way is to use the **read()** and **write()** system calls.

Client and Server Interaction

Following is the diagram showing the complete Client and Server interaction



- ✓ The steps involved in establishing a socket on the server side are as follows −
  - ❖ Create a socket with the **socket()** system call.
  - ❖ Bind the socket to an address using the **bind()** system call. For a server socket on the Internet, an address consists of a port number on the host machine.
  - ❖ Listen for connections with the **listen()** system call.
  - ❖ Accept a connection with the **accept()** system call. This call typically blocks the connection until a client connects with the server.
  - ❖ Send and receive data using the **read()** and **write()** system calls.

Example program to communicate Client and server Using Socket

Client Side:

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<netdb.h>
int main(int argc,char *argv[])
{
        int sockfd,newsockfd,portno,len,n;
        char buffer[256],c[20000];
        struct sockaddr_in serv,cli;
        FILE *fd;
        if(argc<2)
        {
                printf("Err:no port no.\nusage:\n./client portno\n ex:./client 7777\n");
                exit(1);
        }
                sockfd=socket(AF_INET,SOCK_STREAM,0);
                bzero((char *)&serv,sizeof(serv));
                portno=atoi(argv[1]);
                serv.sin_family=AF_INET;
                serv.sin_port=htons(portno);
                if(connect(sockfd,(struct sockaddr *)&serv,sizeof(serv))<0)
                {
                        printf("server not responding..\n\n\n\ti am to terminate\n");
                        exit(1);
                }
        printf("Enter the file with complete path\n");
        scanf("%s",&buffer);
        if(write(sockfd,buffer,strlen(buffer))<0)
        printf("Err writing to socket..\n");
        bzero(c,2000);
        printf("Reading..\n..\n");
        if(read(sockfd,c,1999)<0)
        printf("error: read error\n");
        printf("client: display content of %s\n..\n",buffer);
        fputs(c,stdout);
        printf("\n..\n");
        return 0;
}
```

Server Side:

```
#include<stdio.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<sys/socket.h>
#include<netdb.h>
int main(int argc,char *argv[])
{
        int sockfd,newsockfd,portno,len,n;
        char buffer[256],c[2000],cc[20000];
        struct sockaddr_in serv,cli;
        FILE *fd;
        if(argc<2)
        {
                printf("erroe:no port no\n usage:\n/server port no\n");
                exit(1);
        }
                sockfd=socket(AF_INET,SOCK_STREAM,0);
                portno=atoi(argv[1]);
                serv.sin_family=AF_INET;
                serv.sin_addr.s_addr=INADDR_ANY;
                serv.sin_port=htons(portno);
                bind(sockfd,(struct sockaddr *)&serv,sizeof(serv));
                listen(sockfd,10);
                len=sizeof(cli);
                printf("serve:\nwaiting for connection\n");
                newsockfd=accept(sockfd,(struct sockaddr *)&cli,&len);
                bzero(buffer,255);
                n=read(newsockfd,buffer,255);
                printf("\nserver recv:%s\n",buffer);
        if((fd=fopen(buffer,"r"))!=NULL)
                {
                        printf("server:%s found\n opening and reading..\n",buffer);
                        printf("reading..\n..reading complete");
                        fgets(cc,2000,fd);
                        while(!feof(fd))
                        {
                                fgets(c,2000,fd);
                                strcat(cc,c);
                        }
                n=write(newsockfd,cc,strlen(cc));
                if(n<0)
                printf("error writing to socket");
                printf("\ntransfer complete\n");
```

```
		}
			else
			{
				printf("server:file not found\n");
				n=write(newsockfd,"file not foung",15);
				if(n<0)
				printf("error: writing to socket..\n");
			}
		return 0;
}
```