PROGRAM NO 1: Study of basic Linux commands and system calls used for network programming

Aim: To study basic networking commands in Linux

The network commands explains various tools which can be useful when networking with other computers both within the network and accross the internet, obtaining more information about other computers. This also includes information on tools for network configuration, file transfer and working with remote machines.

1. **netstat**

   Displays contents of /proc/net files. It works with the Linux Network Subsystem, it will tell you what the status of ports are ie. open, closed, waiting, masquerade connections. It will also display various other things. It has many different options.

2. **tcpdump**

   This is a sniffer, a program that captures packets off a network interface and interprets them for you. It understands all basic internet protocols, and can be used to save entire packets for later inspection.

3. **ping**

   The ping command (named after the sound of an active sonar system) sends echo requests to the host you specify on the command line, and lists the responses received their round trip time.

   You simply use ping as:

   pingip_or_host_name

   Note to stop ping (otherwise it goes forever) use **CTRL-C** (break).

   Please note: Using ping/smbmount/ssh or other UNIX system programs with a computer name rather than IP address will only work if you have the computer listed in your /etc/hosts file. Here is an example:

   192.168.1.100 new
   This line says that their is a computer called "new" with IP address 192.168.1.100. Now that it exists in the /etc/hosts file I don't have to type the IP address anymore, just the name "new".

4. **hostname**

   Tells the user the host name of the computer they are logged into. Note: may be called *host.*

5. **traceroute**

traceroute will show the route of a packet. It attempts to list the series of hosts through which your packets travel on their way to a given destination. Also have a look at *xtraceroute* (one of several graphical equivalents of this program).

Command syntax:

traceroutemachine_name_or_ip

6. **tracepath**

tracepath performs a very simlar function to *traceroute* the main difference is that *tracepath* doesn't take complicated options.

Command syntax:

tracepathmachine_name_or_ip

7. **findsmb**

findsmb is used to list info about machines that respond to SMB name queries (for example windows based machines sharing their hard disk's).

Command syntax:

findsmb

This would find all machines possible, you may need to specify a particular subnet to query those machines only...

8. **nmap**

" network exploration tool and security scanner". *nmap* is a very advanced network tool used to query machines (local or remote) as to whether they are up and what ports are open on these machines.

A simple usage example:

nmapmachine_name

This would query your own machine as to what ports it keeps open. *nmap* is a very powerful tool, documentation is available on the nmap site as well as the information in the manual page.

Network Configuration

9. **ifconfig**

This command is used to configure network interfaces, or to display their current configuration. In addition to activating and deactivating interfaces with the "up" and "down" settings, this command is necessary for setting an interface's address information if you don't have the *ifcfg* script.

Use *ifconfig* as either:

ifconfig

This will simply list all information on all network devices currently up.

ifconfig eth0 down

This will take eth0 (assuming the device exists) down, it won't be able to receive or send anything until you put the device back “up” again.

Clearly there are a lot more options for this tool, you will need to read the manual/info page to learn more about them.

PROGRAM NO 2: Process creation using system call fork ( )

Fork is an operation whereby a process creates a copy of itself. It is usually a system call, implemented in the kernel. Fork is the primary (and historically, only) method of process creation on Unix-like operating systems.

NAME

fork - create a child process

SYNOPSIS

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


pid_t fork(void);
```

DESCRIPTION

fork() creates a child process that differs from the parent process only in its PID and PPID, and in the fact that resource utilizations are set to 0. File locks and pending signals are not inherited.

Under Linux, fork() is implemented using copy-on-write pages, so the only penalty that it incurs is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child.

RETURN VALUE

On success, the PID of the child process is returned in the parent's thread of execution, and a 0 is returned in the child's thread of execution. On failure, a -1 will be returned in the parent's context, no child process will be created, and *errno* will be set appropriately.

System call **fork()** is used to create processes. It takes no arguments and returns a process ID. The purpose of **fork()** is to create a ***new*** process, which becomes the *child* process of the caller. After a new child process is created, ***both*** processes will execute the next instruction following the ***fork()*** system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork()**:

- If **fork()** returns a negative value, the creation of a child process was unsuccessful.
- **fork()** returns a zero to the newly created child process.
- **fork()** returns a positive value, the ***process ID*** of the child process, to the parent. The returned process ID is of type **pid_t** defined in **sys/types.h**. Normally, the process ID is an integer. Moreover, a process can use function **getpid()** to retrieve the process ID assigned to this process.

Therefore, after the system call to **fork()**, a simple test can tell which process is the child. **Please note that Unix will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces**.

AIM:

To write a C program to perform process creation using fork() system call

ALGORITHM:

· Start program.
· Assign fork() system call to pid.
· ifpid is equal to -1, child process not created.
· ifpid is equal to 0, child process will be created.
· Print the id of parent process and child process.
· Create another one child process in same loop.
· Print id of parent process and the child process.
· Print grand parent id.
· Stop the program.

PROGRAM SOURCE CODE

```
#include<sys/types.h>
#include<stdio.h>
#include<process.h>
int main()
{
int pid_t,pid,pid1,p,p1;
pid =fork();
if (pid ==-1)
{
printf("enter in connection");
}
else
if(pid==0)
{
printf("\n child process1 :\n\n");
p=getppid();
printf("parent process id of child1: %d\n",p);
p1=getpid();
printf("parent process id of child1: %d\n",p1);
}
else
{
pid1=fork();
if(pid==0)
{
printf("\nchild process 2:\n\n");
p=getppid();
printf("parent process id of child2: %d\n",p);
p1=grtpid();
printf("parent process id of child2: %d\n",p1);
}
```

```
else
{
printf("this is parent process \n");
p=getppid();
printf("grant parent: %d \n",p);
p1=getpid();
printf("process id of parent: %d \n",p1);
}
}
return 0;
}
```

CONCLUSION:

Thus the c program to perform process creation using fork() was written and executed successfully.

PROGRAM NO 3: Generation of prime and Fibonacci series using threads

Thread Basics:

- Thread operations include thread creation, termination, synchronization (joins,blocking), scheduling, data management and process interaction.
- A thread does not maintain a list of created threads, nor does it know the thread that created it.
- All threads within a process share the same address space.
- Threads in the same process share:
  - Process instructions
  - Most data
  - open files (descriptors)
  - signals and signal handlers
  - current working directory
  - User and group id
- Each thread has a unique:
  - Thread ID
  - set of registers, stack pointer
  - stack for local variables, return addresses
  - signal mask
  - priority
  - Return value: errno
- pthread functions return "0" if OK.

Thread Creation and Termination:

- Threads terminate by explicitly calling pthread_exit(), by letting the function return, or by a call to the function exit() which will terminate the process including any threads.
- Function call: **pthread_create** - create a new thread

```
int pthread_create(pthread_t * thread,  const pthread_attr_t * attr,   void *
(*start_routine)(void *),  void *arg);
```

Arguments:

- thread - returns the thread id. (unsigned long int defined in bits/pthreadtypes.h)
- attr - Set to NULL if default thread attributes are used. (else define members of the struct pthread_attr_t defined in bits/pthreadtypes.h) Attributes include:
  - detached state (joinable? Default: PTHREAD_CREATE_JOINABLE. Other option: PTHREAD_CREATE_DETACHED)
  - scheduling policy (real-time? PTHREAD_INHERIT_SCHED,PTHREAD_EXPLICIT_SCHED,SCHED_OTHER)
  - scheduling parameter

        - inheritsched attribute (Default: PTHREAD_EXPLICIT_SCHED Inherit from parent thread: PTHREAD_INHERIT_SCHED)
        - scope (Kernel threads: PTHREAD_SCOPE_SYSTEM User threads: PTHREAD_SCOPE_PROCESS Pick one or the other not both.)
        - guard size
        - stack address (See unistd.h and bits/posix_opt.h _POSIX_THREAD_ATTR_STACKADDR)
        - stack size (default minimum PTHREAD_STACK_SIZE set in pthread.h),
    - void * (*start_routine) - pointer to the function to be threaded. Function has a single argument: pointer to void.
    - *arg - pointer to argument of function. To pass multiple arguments, send a pointer to a structure.
- Function call: **pthread_join** - wait for termination of another thread

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

Arguments:

- th - thread suspended until the thread identified by th terminates, either by calling pthread_exit() or by being cancelled.
- thread_return - If thread_return is not NULL, the return value of th is stored in the location pointed to by thread_return.

- Function call: **pthread_exit** - terminate the calling thread

```
void pthread_exit(void *retval);
```

Arguments:

- retval - Return value of pthread_exit().

This routine kills the thread. The pthread_exit() function never returns. If the thread is not detached, the thread id and return value may be examined from another thread by using pthread_join().
Note: the return pointer *retval, must not be of local scope otherwise it would cease to exist once the thread terminates.

Thread Synchronization:

The threads library provides three synchronization mechanisms:

- mutexes - Mutual exclusion lock: Block access to variables by other threads. This enforces exclusive access by a thread to a variable or set of variables.
- joins - Make a thread wait till others are complete (terminated).
- condition variables - data type pthread_cond_t

Compile:

- C compiler: cc -pthread pthread1.c (**or** cc -lpthread pthread1.c)
  or
- C++ compiler: g++ -pthread pthread1.c (**or** g++ -lpthread pthread1.c)

The GNU compiler now has the command line option "-pthread" while older versions of the compiler specify the pthread library explicitly with "-lpthread".
Run: ./a.out

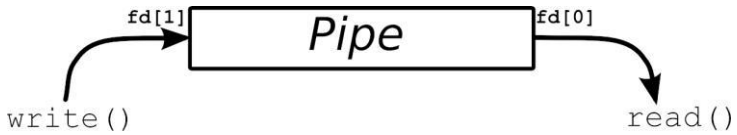
```
Fibonacci and prime numbers
===========================

#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
void fib(void *p1)
 {
  int a=0,b=1,c;
  int i,l;
  printf("\nEnter the limit\n");
  scanf("%d",&l);
  printf("\nFibnocci series is:\n");
  printf("%d\t%d",a,b);
  for(i=2;i<=l;i++)
   {
    c=a+b;
    a=b;
    b=c;
    printf("\t%d",c);
   }
 }
void prime(void *p2)
 {
  int n,i=2;
  int cl,j;
  printf("\nEnter the limit\n");
  scanf("%d",&n);
  printf("\nPrime numbers are: ");
  while(i<=n)
   {
    if(i==2)
     {
      printf("%d\t",i);
      i++;
     }
    else
     {
      cl=0;
      for(j=1;j<=i;j++)
       {
        if(i%j==0)
         {
          cl++;
         }
       }
      if(cl==2)
       {
```

```
            printf("%d\t",i);
            i++;
           }
          else
           {
            i++;
           }
        }
      }
 }
void main()
 {
  pthread_t p1,p2;
  pthread_create(&p1,NULL,(void*)&fib,NULL);
  pthread_join(p1,NULL);
  pthread_create(&p2,NULL,(void*)&prime,NULL);
  pthread_join(p2,NULL);
 }
```

PROGRAM NO 4: Implement inter process communication using pipe

DESCRIPTION:

There is no form of IPC that is simpler than pipes, Implemented on every flavour of UNIX.



Basically, a call to the **pipe()** function returns a pair of file descriptors. One of these descriptors is connected to the write end of the pipe, and the other is connected to the read end. Anything can be written to the pipe, and read from the other end in the order it came in. On many systems, pipes will fill up after you write about 10K to them without reading anything out.

The following example shows how a pipe is created, reading and writing from pipe.

A pipe provides a one-way flow of data.

A pipe is created by the pipe system call.

int pipe (int *filedes) ;

Two file descriptors are returned- filedes[0] which is open for reading , and filedes[1] which is open for writing.

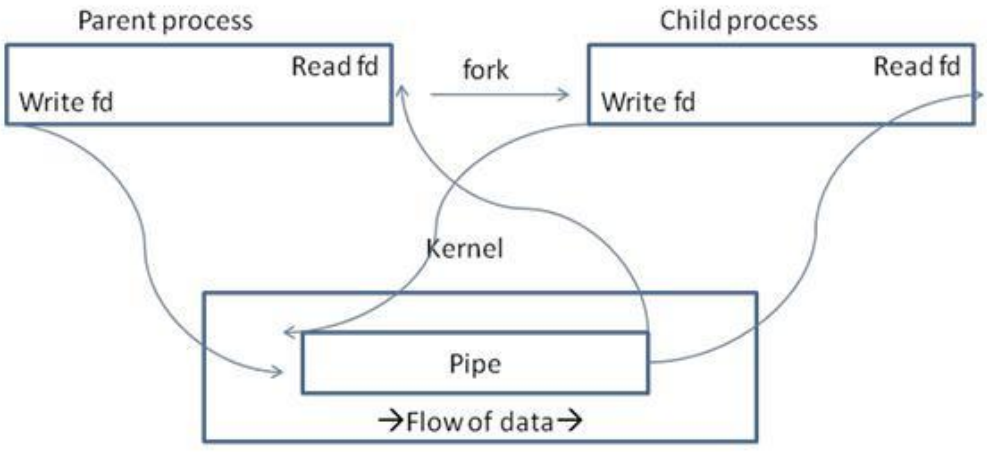


Fig: Pipe in a single process, immediately after fork

Pipes are typically used to communicate between two different processes in the following way. First, a process creates a pipe and then forks to create a copy of itself, as shown above figure. Next the parent process closes the read end of the pipe and the child process closes the write end of the pipe. The fork system call creates a copy of the process that was executing. The process that executed the fork is called the parent process and the new process is called the child process.

The fork system call is called once but it returns twice.

1) The first return value in the parent process is the process ID of the newly created child process.
2) The second return value in the child process is zero.
   If the fork system call is not successful, -1 is returned

Pseudo code:

START

Store any message in one character array (char *msg="Hello world")

Declare another character array

Create a pipe by using pipe() system call

Create another process by executing fork() system call

In parent process use system call write() to write message from one process to another process.

In child process display the message.

END

/* **CREATION OF A ONEWAY PIPE IN A SINGLE PROCESS**. */

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
main()
{
    int pipefd[2],n;
```

```
    char buff[100];
    pipe(pipefd);
    printf("\nreadfd=%d",pipefd[0]);
    write(pipefd[1],"helloworld",12);
    n=read(pipefd[1],buff,sizeof(buff));
    printf("\n size of the data%d",n);
    printf("\n data from pipe:%s",buff);
}
```

OUTPUT:

```
readfd=3
writefd=4
 size of the data-1
```

/* **CREATION OF A ONEWAY PIPE BETWEEN TWO PROCESS** */

PROGRAM

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

main()
{
        intpipefd[2],n,pid;
        char buff[100];
        pipe(pipefd);
        printf("\n readfd=%d",pipefd[0]);
        printf("\n writefd=%d",pipefd[1]);
        pid=fork();
```

```
        if(pid==0)
        {
                close(pipefd[0]);
                printf("\n CHILD PROCESS SENDING DATA\n");
                write(pipefd[1],"hello world",12);
        }


        else
        {
               close(pipefd[1]);
                printf("PARENT PROCESS RECEIVES DATA\n");
               n=read(pipefd[0],buff,sizeof(buff));
                printf("\n size of data%d",n);
                printf("\n data received from child throughpipe:%s\n",buff);
         }
}
```

OUTPUT

```
readfd=3
writefd=4
CHILD PROCESS SENDING DATA
writefd=4
PARENT PROCESS RECEIVES DATA
```

/***CREATION OF A TWOWAY PIPE BETWEEN TWO PROCESS***/

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
main()
{
        int p1[2],p2[2],n,pid;
        char buf1[25],buf2[25];
        pipe(p1);
        pipe(p2);
        printf("\n readfds=%d %d\n",p1[0],p2[0]);
        printf("\n writefds=%d %d\n",p1[1],p2[1]);
        pid=fork();
        if(pid==0)
        {
                close(p1[0]);
                printf("\n CHILD PROCESS SENDING DATA\n");
                write(p1[1],"where is CEP",25);

                close(p2[1]);
                read(p2[0],buf1,25);
                printf(" reply from parent:%s\n",buf1);
                sleep(2);
        }
        else
        {
                close(p1[1]);
                printf("\n parent process receiving data\n");
                n=read(p1[0],buf2,sizeof(buf2));
```

```
                printf("\n data received from child through pipe:%s\n",buf2);
                sleep(3);
                close(p2[0]);
                write(p2[1]," at perumon",25);
                printf("\n reply send\n");
        }
}
```

OUTPUT:

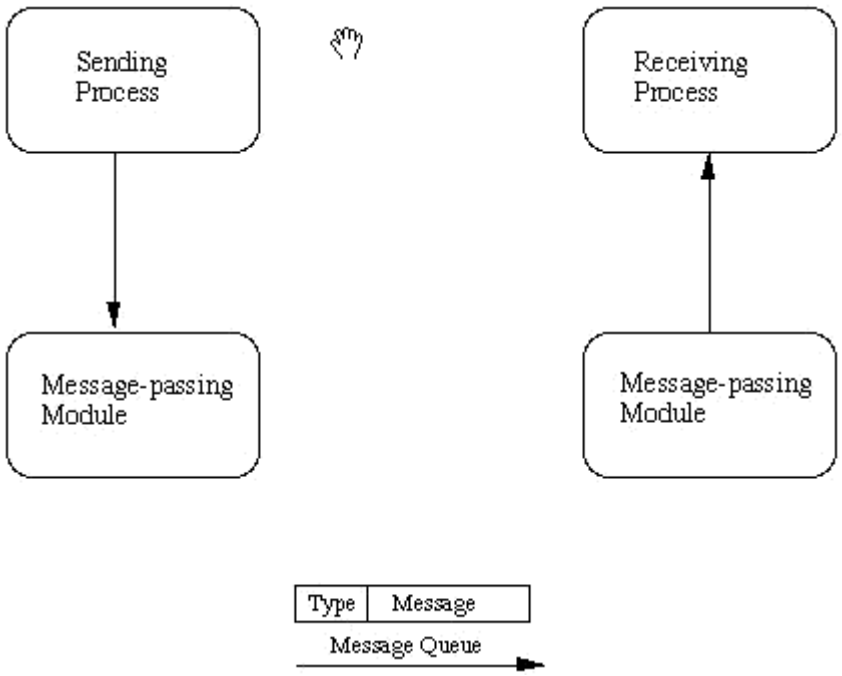
```
readfds=3 5
writefds=4 6
CHILD PROCESS SENDING DATA

parent process receiving data
data received from child through pipe: where is CEP

reply send
reply from parent: at perumon
```

PROGRAM NO 5: Implement inter process communication using message queue

IPC:Message Queues:<sys/msg.h>

The basic idea of a ***message queue*** is a simple one.

Two (or more) processes can exchange information via access to a common system message queue. The ***sending*** process places via some (OS) message-passing module a message onto a queue which can be read by another process (Figure 24.1). Each message is given an identification or type so that processes can select the appropriate message. Process must share a common key in order to gain access to the queue in the first place (subject to other permissions -- see below).



Basic Message Passing

IPC messaging lets processes send and receive messages, and queue messages for processing in an arbitrary order. Unlike the file byte-stream data flow of pipes, each IPC message has an explicit length. Messages can be assigned a specific type. Because of this, a server process can direct message traffic between clients on its queue by using the client process PID as the message type. For single-message transactions, multiple server processes can work in parallel on transactions sent to a shared message queue.

Before a process can send or receive a message, the queue must be initialized (through the msgget function see below) Operations to send and receive messages are performed by the msgsnd()and msgrcv() functions, respectively.

When a message is sent, its text is copied to the message queue. The msgsnd() and msgrcv() functions can be performed as either blocking or non-blocking

operations. Non-blocking operations allow for asynchronous message transfer -- the process is not suspended as a result of sending or receiving a message. In blocking or synchronous message passing the sending process cannot continue until the message has been transferred or has even been acknowledged by a receiver. IPC signal and other mechanisms can be employed to implement such transfer. A blocked message operation remains suspended until one of the following three conditions occurs:

- The call succeeds.
- The process receives a signal.
- The queue is removed.

Initialising the Message Queue

The msgget() function initializes a new message queue:

int msgget(key_t key, int msgflg)

It can also return the message queue ID (msqid) of the queue corresponding to the key argument. The value passed as the msgflg argument must be an octal integer with settings for the queue's permissions and control flags.

The following code illustrates the msgget() function.

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

...


key_t key; /* key to be passed to msgget() */
int msgflg /* msgflg to be passed to msgget() */
int msqid; /* return value from msgget() */

...
key = ...
msgflg = ...

if ((msqid = msgget(key, msgflg)) == &ndash;1)
  {
    perror("msgget: msgget failed");
    exit(1);
   } else
    (void) fprintf(stderr, &ldquo;msgget succeeded");
...
```

IPC Functions, Key Arguments, and Creation Flags: <sys/ipc.h>

Processes requesting access to an IPC facility must be able to identify it. To do this, functions that initialize or provide access to an IPC facility use a key_t key argument. (key_t is essentially an int type defined in <sys/types.h>

The key is an arbitrary value or one that can be derived from a common seed at run time. One way is with ftok() , which converts a filename to a key value that is unique within the system. Functions that initialize or get access to messages (also semaphores or shared memory see later) return an ID number of type int. IPC functions that perform read, write, and control operations use this ID. If the key argument is specified as IPC_PRIVATE, the call initializes a new instance of an IPC facility that is private to the creating process. When the IPC_CREAT flag is supplied in the flags argument appropriate to the call, the function tries to create the facility if it does not exist already. When called with both the IPC_CREAT and IPC_EXCL flags, the function fails if the facility already exists. This can be useful when more than one process might attempt to initialize the facility. One such case might involve several server processes having access to the same facility. If they all attempt to create the facility with IPC_EXCL in effect, only the first attempt succeeds. If neither of these flags is given and the facility already exists, the functions to get access simply return the ID of the facility. If IPC_CREAT is omitted and the facility is not already initialized, the calls fail. These control flags are combined, using logical (bitwise) OR, with the octal permission modes to form the flags argument. For example, the statement below initializes a new message queue if the queue does not exist.

```
msqid = msgget(ftok("/tmp",
key), (IPC_CREAT | IPC_EXCL | 0400));
```

The first argument evaluates to a key based on the string ("/tmp"). The second argument evaluates to the combined permissions and control flags.

Controlling message queues

The msgctl() function alters the permissions and other characteristics of a message queue. The owner or creator of a queue can change its ownership or permissions using msgctl() Also, any process with permission to do so can use msgctl() for control operations.

The msgctl() function is prototypes as follows:

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf )
```

The msqid argument must be the ID of an existing message queue. The cmd argument is one of:

IPC_STAT

-- Place information about the status of the queue in the data structure pointed to by buf. The process must have read permission for this call to succeed.

IPC_SET

-- Set the owner's user and group ID, the permissions, and the size (in number of bytes) of the message queue. A process must have the effective user ID of the owner, creator, or superuser for this call to succeed.

IPC_RMID

-- Remove the message queue specified by the msqid argument.

The following code illustrates the msgctl() function with all its various flags:

```
#include<sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
...
if (msgctl(msqid, IPC_STAT, &buf) == -1) {
perror("msgctl: msgctl failed");
exit(1);
}
...
if (msgctl(msqid, IPC_SET, &buf) == -1) {
perror("msgctl: msgctl failed");
exit(1);
}
...
```

Sending and Receiving Messages

The msgsnd() and msgrcv() functions send and receive messages, respectively:

```
int msgsnd(int msqid, const void *msgp, size_t msgsz,
        int msgflg);

int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
        int msgflg);
```

The msqid argument **must** be the ID of an existing message queue. The msgp argument is a pointer to a structure that contains the type of the message and its text. The structure below is an example of what this user-defined buffer might look like:

```
 struct mymsg {
     long      mtype;    /* message type */
     char mtext[MSGSZ]; /* message text of length MSGSZ */
}
```

The msgsz argument specifies the length of the message in bytes.

The structure member msgtype is the received message's type as specified by the sending process.

The argument msgflg specifies the action to be taken if one or more of the following are true:

- The number of bytes already on the queue is equal to msg_qbytes.
- The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

- If (msgflg & IPC_NOWAIT) is non-zero, the message will not be sent and the calling process will return immediately.
- If (msgflg & IPC_NOWAIT) is 0, the calling process will suspend execution until one of the following occurs:
  - The condition responsible for the suspension no longer exists, in which case the message is sent.
  - The message queue identifier msqid is removed from the system; when this occurs, errno is set equal to EIDRM and -1 is returned.
  - The calling process receives a signal that is to be caught; in this case the message is not sent and the calling process resumes execution.

Upon successful completion, the following actions are taken with respect to the data structure associated with msqid:

- msg_qnum is incremented by 1.
- msg_lspid is set equal to the process ID of the calling process.
- msg_stime is set equal to the current time.

The following code illustrates msgsnd() and msgrcv():

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

...

int msgflg; /* message flags for the operation */
struct msgbuf *msgp; /* pointer to the message buffer */
int msgsz; /* message size */
long msgtyp; /* desired message type */
int msqid /* message queue ID to be used */

...

msgp = (struct msgbuf *)malloc((unsigned)(sizeof(struct msgbuf)
- sizeof msgp->mtext + maxmsgsz));

if (msgp == NULL) {
(void) fprintf(stderr, "msgop: %s %d byte messages.\n",
"could not allocate message buffer for", maxmsgsz);
exit(1);

...
```

```
msgsz = ...
msgflg = ...

if (msgsnd(msqid, msgp, msgsz, msgflg) == -1)
perror("msgop: msgsnd failed");
...
msgsz = ...
msgtyp = first_on_queue;
msgflg = ...
if (rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg) == -1)
perror("msgop: msgrcv failed");
...
```

Example: Sending messages between two processes

The following two programs should be compiled and run ***at the same time*** to illustrate basic principle of message passing:

message_send.c

-- Creates a message queue and sends one message to the queue.

message_rec.c

-- Reads the message from the queue.

message_send.c -- creating and sending to a simple message queue

The full code listing for message_send.c is as follows:

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

#define MSGSZ     128


/*
 * Declare the message structure.
 */

typedef struct msgbuf {
        long    mtype;
        char    mtext[MSGSZ];
        } message_buf;

main()
{
```

```
    int msqid;
    int msgflg = IPC_CREAT | 0666;
    key_t key;
    message_buf sbuf;
    size_t buf_length;

    /*
     * Get the message queue id for the
     * "name" 1234, which was created by
     * the server.
     */
    key = 1234;

(void) fprintf(stderr, "\nmsgget: Calling msgget(%#lx,\
%#o)\n",
key, msgflg);

    if ((msqid = msgget(key, msgflg )) < 0) {
        perror("msgget");
        exit(1);
    }
    else
     (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);


    /*
     * We'll send message type 1
     */

    sbuf.mtype = 1;

    (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);

    (void) strcpy(sbuf.mtext, "Did you get this?");

    (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);

    buf_length = strlen(sbuf.mtext) + 1 ;



    /*
     * Send a message.
     */
    if (msgsnd(msqid, &sbuf, buf_length, IPC_NOWAIT) < 0) {
       printf ("%d, %d, %s, %d\n", msqid, sbuf.mtype, sbuf.mtext, buf_length);
        perror("msgsnd");
        exit(1);
    }
```

```
  else
    printf("Message: \"%s\" Sent\n", sbuf.mtext);

   exit(0);
}
```

The essential points to note here are:

- The Message queue is created with a basic key and message flag msgflg = IPC_CREAT | 0666 -- create queue and make it read and appendable by all.
- A message of type (sbuf.mtype) 1 is sent to the queue with the message Did you get this?

message_rec.c -- receiving the above message

The full code listing for message_send.c's companion process, message_rec.c is as follows:

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

#define MSGSZ     128


/*
 * Declare the message structure.
 */

typedef struct msgbuf {
   long    mtype;
   char    mtext[MSGSZ];
} message_buf;


main()
{
   int msqid;
   key_t key;
   message_buf  rbuf;

   /*
    * Get the message queue id for the
    * "name" 1234, which was created by
    * the server.
    */
   key = 1234;

   if ((msqid = msgget(key, 0666)) < 0) {
      perror("msgget");
```

```
        exit(1);
    }


    /*
     * Receive an answer of message type 1.
     */
    if (msgrcv(msqid, &rbuf, MSGSZ, 1, 0) < 0) {
        perror("msgrcv");
        exit(1);
    }

    /*
     * Print the answer.
     */
    printf("%s\n", rbuf.mtext);
    exit(0);
}
```

The essential points to note here are:

- The Message queue is opened with msgget (message flag 0666) and the ***same*** key as message_send.c.
- A message of the ***same*** type 1 is received from the queue with the message stored in rbuf.mtext

PROGRAM

Server

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/stat.h>
#include<sys/msg.h>
#include<sys/ipc.h>
struct mymsg
{
long type;
char msg[30];
char from[10];
};// creating user defined structure
struct mymsg r;
main()
{
        key_t key;
        int mqid,d;
        char buff[30];
        key=ftok(".",1);//generating key
        mqid=msgget(key,IPC_CREAT|0666);//creating message queue
```

```
	while(1)
	{
		d=msgrcv(mqid,&r,sizeof(struct mymsg),1,0);//receiving message
		printf("Received msg:%s is from %s\n",r.msg,r.from);//printing the received
message
	}
}
```

Client

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/stat.h>
#include<sys/msg.h>
#include<sys/ipc.h>
struct mymsg
{
long type;
char msg[30];
char from[10];
};// creating user defined structure
struct mymsg m;
main()
{
	key_t key;
	int mqid,d;
	char buff[30];
	key=ftok(".",1);//generating key
	mqid=msgget(key,IPC_CREAT|0666);//creating message queue
while(1)
	{
		printf("Enter msg:");
		gets(buff);
		strcpy(m.msg,buff);//copying the message in buffer to structure variable msg
		m.type=1; // storing message type as one
		strcpy(m.from,"Client");//copying the source of message to structure variable
msg
		d=msgsnd(mqid,&m,sizeof(struct mymsg),0); //sending message
	}
}
```

OUTPUT

```
Server
Received msg:  HAI HOW R U? is from Client
Received msg: IN CEP is from Client
Received msg: KOLLAM is from Client
Client
Enter msg: HAI HOW R U?
Enter msg: IN CEP
```

Enter msg: KOLLAM
Enter msg:

Implement parent child communication using message queue

PROGRAM

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/stat.h>
#include<sys/msg.h>
#include<sys/ipc.h>
struct mymsg
{
        long type;
        char msg[30];
        char from[10];
};// creating user defined structure
struct mymsg m,r;
main()
{
        key_t key;
        pid_t pid;
        int mqid,d;
        char buff[30];
        key=ftok(".",1);//generating key
        mqid=msgget(key,IPC_CREAT|0666);//creating message queue
        pid=fork();//creating a child process
        if(pid==0)//child
{
                while(1)
                {
                        sleep(1);
                        printf("Enter msg:");
                        gets(buff);
                        strcpy(m.msg,buff);
                        m.type=1;
                        strcpy(m.from,"child");
                        d=msgsnd(mqid,&m,sizeof(struct mymsg),0); //sending message
                }
        }
        if(pid>0)       //parent
        {
                while(1)
                {
                        d=msgrcv(mqid,&r,sizeof(struct mymsg),1,0);//receiving message
                        printf("Received msg:%s is from %s\n",r.msg,r.from);//printing the
received message
```

```
                }
        }
}
```

OUTPUT

```
Enter msg:HAI
Received msg:HAI is from child
Enter msg:HOW
Received msg:HOW is from child
Enter msg:R
Received msg:R is from child
Enter msg:U?
Received msg:U? is from child
Enter msg:FROM
Received msg:FROM is from child
Enter msg:CEP
Received msg:CEP is from child
Enter msg:KOLLAM
Received msg:KOLLAM is from child
Enter msg:^C
```

Implement 2-way communication using message queue

PROGRAM

Server

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/stat.h>
#include<sys/msg.h>
#include<sys/ipc.h>
struct mymsg
{
long type;
char msg[30];
char from[10];
};// creating user defined structure
struct mymsg m,r;
main()
{
        key_t key;
        int mqid,d;
        char buff[30];
        key=ftok(".",1);//generating key
        mqid=msgget(key,IPC_CREAT|0666);//creating message queue
        while(1)
```

```
	{
		d=msgrcv(mqid,&r,sizeof(struct mymsg),1,0);//receiving message
		printf("Received msg:%s is from %s\n",r.msg,r.from);
		sleep(1);
		printf("Enter msg:");
		gets(buff);
		strcpy(m.msg,buff);//copying the message in buffer to structure variable msg
		m.type=1;
		strcpy(m.from,"Client");
		d=msgsnd(mqid,&m,sizeof(struct mymsg),0); //sending message
	}
}
```

Client

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/stat.h>
#include<sys/msg.h>
#include<sys/ipc.h>
struct mymsg
{
long type;
char msg[30];
char from[10];
};// creating user defined structure
struct mymsg m,r;
main()
{
	key_t key;
	int mqid,d;
	char buff[30];
	key=ftok(".",1);//generating key
	mqid=msgget(key,IPC_CREAT|0666);//creating message queue
	while(1)
	{
		sleep(1);
		printf("Enter msg:");
		gets(buff);
		strcpy(m.msg,buff);//copying the message in buffer to structure variable msg
		m.type=1;
		strcpy(m.from,"Client");
		d=msgsnd(mqid,&m,sizeof(struct mymsg),0); //sending message
		sleep(1);
		d=msgrcv(mqid,&r,sizeof(struct mymsg),1,0);//receiving message
		printf("Received msg:%s is from %s\n",r.msg,r.from);
	}
}
```

OUTPUT

Server
enter data:HAI
received data:SURYA
Client
received data:HAI
enter data:SURYA

PROGRAM NO 6: Implement inter process communication using shared memory

Shared Memory is an efficient means of passing data between programs. One program will create a memory portion, which other processes (if permitted) can access. A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory. One process must explicitly ask for an area, using a key, to be shared by other processes. This process will be called the server. All other processes, the clients that know the shared area can access it. A shared memory segment is identified by a unique integer, the shared memory ID. Here we have to write two programs one server program and one client program. Save these two programs in two separate files.->Server.c and Client.c Start a new terminal and compile and run the server program first. Then start other terminal and compile and run the client program

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

#include <sys/ipc.h>

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

For a server, it should be started before any client. The server should perform the following tasks:

1. This is performed by system call shmget( ).

2. Attach this shared memory to the server's address space with system call shmat( ).

3. Initialize the shared memory, if necessary.

4. Do something and wait for all clients' completion.

5. Detach the shared memory with system call shmdt( ).

6. Remove the shared memory with system call shmctl( ).

For the client part,

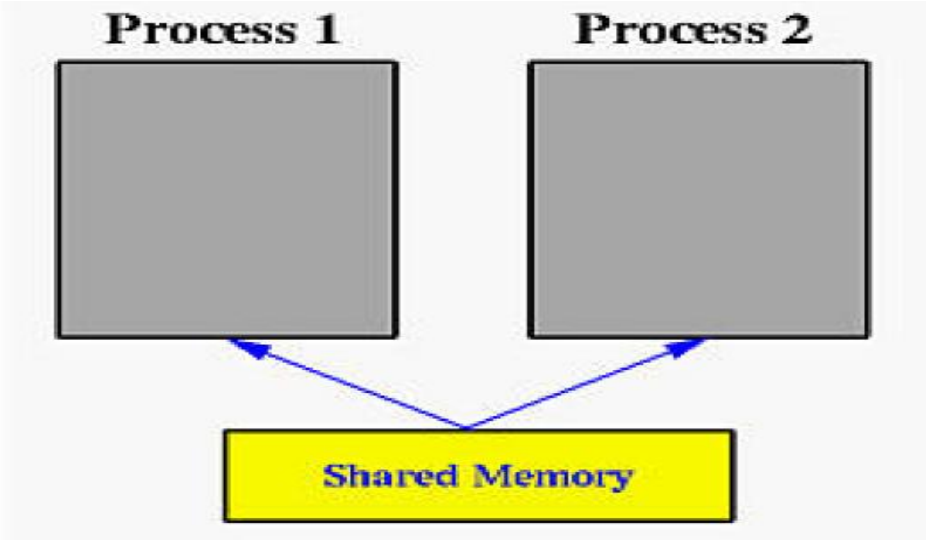
1. Ask for a shared memory with the same memory key and memorize the returned shared memory ID.

2. Attach this shared memory to the client's address space.

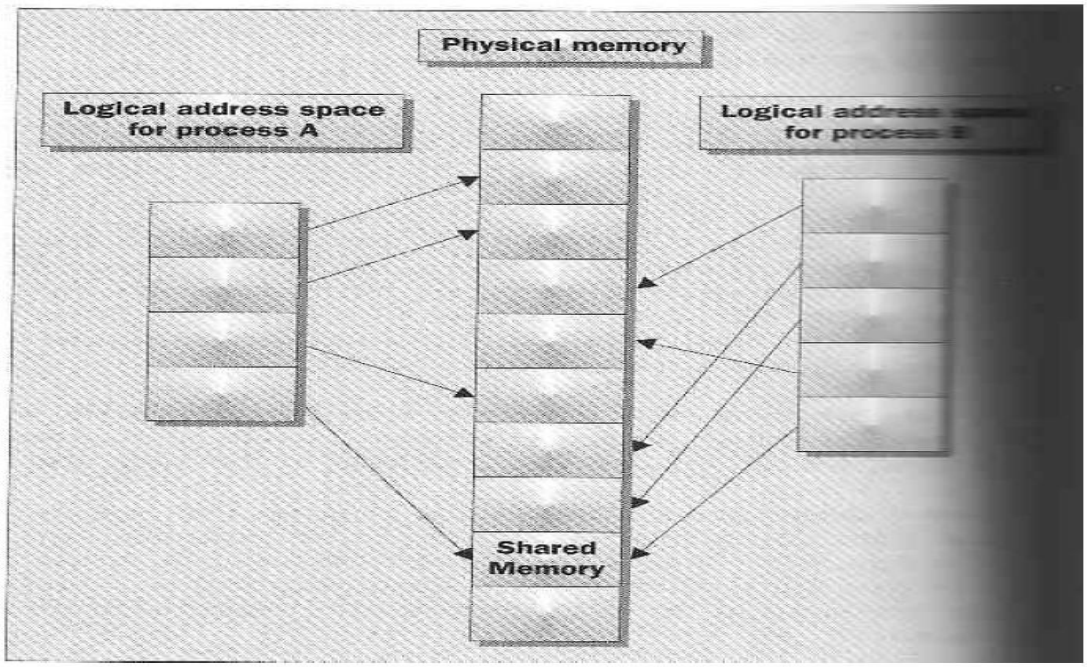
3. Use the memory.

4. Detach all shared memory segments, if necessary.

5. Exit.



The above figure shows two processes and their address spaces. The yellow rectangle is a shared memory attached to both address spaces and both process 1 and process 2 can have access to this shared memory as if the shared memory is part of its own address space.



Shmget System call

The system call that requests a shared memory segment.

shm_id = shmget (key_t k, int size, int flag);

k is the key for the segment . It is the numeric key to be assigned to the returned shared memory segment size is the size of the requested shared memory flag create/use flag . The purpose of flag is to specify the way that the shared memory will be used.

IPC_CREAT | 0666 for a server (i.e., creating and granting read and write access to the server)

0666 for any client (i.e., granting read and write access to the client)

If shmget( ) can successfully get the requested shared memory, its function value is a non-negative integer the shared memory ID. otherwise, the function value is negative.

A key is simply an integer of type key_t

There are three different ways of using keys, namely:

1. a specific integer value (e.g., 123456)

```
key_t SomeKey;

SomeKey = 1234;
```

2. a key generated with function ftok( )

```
key_t ftok ( const char *path, int id);
```

Function ftok( ) takes a character string that identifies a path and an integer (usually a character) value, and generates an integer of type key_t based on the first argument with the value of id in the most significant position.

3. a uniquely generated key using IPC_PRIVATE (i.e., a private key).

When the key field accepts a special parameter, IPC_PRIVATE. In this case, the system will generate a unique key and guarantee that no other process will have the same key.

```
int shm_id;

key_t mem_key;

mem_key = ftok(".", 'a');

shm_id = shmget(mem_key, 4*sizeof(int), IPC_CREAT | 0666);
```

Shmat System call

After a shared memory ID is returned by shmget() system call, the next step is to attach it to the address space of a process. This is done with system call shmat( ). process 1- a server uses shmget( ) to request a shared memory segment successfully. That shared memory segment

exists somewhere in the memory, but is not yet part of the address space of process 1. process 2 –Client requests the same shared memory segment with the same key value, process 2 will be granted the right to use the shared memory segment; but it is not yet part of the address space of process 2. To make a requested shared memory segment part of the address space of a process, use shmat()

shm_ptr = shmat (int shm_id, char *ptr, int flag);

shm_id is shared memory ID returned by shmget()

ptr is a character pointer. usually it is NULL

The returned value is a pointer of type (void *) to the attached shared memory. Thus, type casting is usually necessary.

If this call is unsuccessful, the return value is -1.

shm_ptr = (int *) shmat(shm_id, NULL, 0);

Shmdt System call

System call shmdt( ) is used to detach a shared memory. After a shared memory is detached, it cannot be used.

Syntax: shmdt (shm_ptr);

where shm_ptr is the pointer to the shared memory. This pointer is returned by shmat( ) when the shared memory is attached. If the detach operation fails, the returned function value is non-zero.

Shmctl System call

To remove a shared memory, use shmctl( ). The only argument to shmdt( ) is the shared memory address returned by shmat( ). To remove a shared memory segment, use the following code:

shmctl (shm_id, IPC_RMID, NULL); IPC_RMID indicates this is a remove operation

There are two programs server and client. The server will write alphabets a to z into a shared memory, which is shared between the client and the server. And the server waits until client has read the alphabets from the shared memory. The client will read alphabets from shared memory and prints it. After that it set first location of shared memory to '*' indicating it has read the contents in the shared memory.

PROGRAM

Server Program

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

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

```
#include <sys/shm.h>
#include <stdio.h>
#define SHMSIZE 27
main()
{
        char c;
        int shmid;
        key_t key;
        char *shm, *s;
        /* * We'll name our shared memory segment * "5678". */
        key = 5678;
        /* * create the segment.* */
        shmid = shmget(key, SHMSIZE, IPC_CREAT | 0666);
        /** Now we attach the segment to our data space.*/
        shm = shmat(shmid, NULL, 0);

        /** Now put some things into the memory for the other process to read. */
        s = shm;
        for (c = 'a'; c <= 'z'; c++)
                *s++ = c;
        *s = NULL;
        /** Finally, we wait until the other process
        * Changes the first character of our memory
        * to '*', indicating that it has read what
        * we put there.
        */
        while (*shm != '*')
                sleep(1);
```

```
}
```

Client Program

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSIZE 27
main()
{
        int shmid;
        key_t key;
        char *shm, *s;

        /*
        * We need to get the segment named
        * "5678", created by the server.
        */
        key = 5678;
        /*
        * Locate the segment.
        */
        shmid = shmget(key, SHMSIZE, 0666);

        /*
        * Now we attach the segment to our data space.
        */
        shm = shmat(shmid, NULL, 0);
```

```
/*
* Now read what the server put in the memory.
*/
for (s = shm; *s != NULL; s++)
putchar(*s);
putchar('\n');
/*
* Finally, change the first character of the
* segment to '*', indicating we have read
* the segment.
*/
*shm = '*';
printf ("\nIts done from client.\n\n\n");

}
```

Implement IPC using shared memory in which server sends a set of numbers to client and client print even numbers and then odd numbers.

There are two programs server and client. The server will write limit and numbers in an array into a shared memory, which is shared between the client and the server. And the server waits until client has read the contents from the shared memory. The client will read limit and array from shared memory into a new array and prints even number then odd numbers. After that it set first location of shared memory to '*' indicating it has read the contents in shared memory.

PROGRAM

Server Program

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

```
#define SHMSIZE 27
main()
{
        int i,a;
        int shmid,n;
        key_t key;
        int *shm, *s;
        /* * We'll name our shared memory segment * "5678". */
        key = 5678;
        /* * create the segment.* */
        shmid = shmget(key, SHMSIZE, IPC_CREAT | 0666);
        /** Now we attach the segment to our data space.*/
        shm = shmat(shmid, NULL, 0);

        /** Now put some things into the memory for the other process to read. */
        printf("Enter the limit");
        scanf("%d",&n);
        s = shm;
        // store limit in the first location of shared memory
        *s=n;
        // store numbers from second location of shared memory
        s=s+1;
        printf("\nEnter numbers");
        for (i=0;i<n;i++)
        {
        scanf("%d",&a);
        *s++ =a;
        }
        *s = NULL;
```

```
        /** Finally, we wait until the other process
        * Changes the first character of our memory
        * to '*', indicating that it has read what
        * we put there.
        */
        while (*shm != '*')
                sleep(1);

}
```

Client Program

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSIZE 27
main()
{
        int shmid,n,a[20],i;
        key_t key;
        int *shm, *s;
        /*
        * We need to get the segment named
        * "5678", created by the server.
        */
        key = 5678;
        /*
        * Locate the segment.
```

```
*/
shmid = shmget(key, SHMSIZE, 0666);
/*
* Now we attach the segment to our data space.
*/
shm = shmat(shmid, NULL, 0);

/*
* Now read what the server put in the memory.
*/
s=shm;// Starting location of shared memory
n=*s;
s=s+1;
for (i=0;i<n;i++)
{
a[i]=*s;// store numbers send from server in an array
s=s+1; // Increment the addresss to access next number
}
printf("\nEven Numbers are:");
for (i=0;i<n;i++)
{
if(a[i]%2==0)
        printf("%d ",a[i]);
}
printf("\nOdd Numbers are:");
for (i=0;i<n;i++)
{
if(a[i]%2==1)
        printf("%d ",a[i]);
```

```
        }
        /*
        * Finally, change the first character of the
        * segment to '*', indicating we have read
        * the segment.
        */
        *shm = '*';
        printf ("\nIts done from client.\n\n\n");

}
```

PROGRAM NO 7: Implement Readers Writers problem

The simplest reader writer problem which uses only two semaphores and doesn't need an array of readers to read the data in buffer.

Please notice that this solution gets simpler than the general case because it is made equivalent to the Bounded buffer problem, and therefore only N readers are allowed to enter in parallel, N being the size of the buffer.

Reader

```
do {
        wait(read)
        .....
        reading data
        .....
        signal(write)
} while(TRUE);
```

Writer

```
do {
        wait(write)
        .....
        writing data
        .....
        signal(read)
} while(TRUE);
```

Algorithm

1. Reader will run after Writer because of read semaphore.
2. Writer will stop writing when the write semaphore has reached 0.
3. Reader will stop reading when the read semaphore has reached 0.

In writer, the value of write semaphore is given to read semaphore and in reader, the value of read is given to write on completion of the loop.

Readers Writer Problem

Readers writer problem is another example of a classic synchronization problem. There are many variants of this problem, one of which is examined below.

Problem Statement:

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer**is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non zero number of readers accessing the resource.

Solution:

From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.

Here, we use one mutex **m** and a semaphore **w**. An integer variable **read_count** is used to maintain the number of readers currently accessing the resource. The variable **read_count** is initialized to 0. A value of 1 is given initially to **m** and **w**.

Instead of having the process to acquire lock on the shared resource, we use the mutex **m** to make the process to acquire and release lock whenever it is updating the **read_count** variable.

The code for the writer process looks like this:

```
while(TRUE) {
wait(w);
   /*perform the
write operation */
signal(w);
}
```

The code for the reader process looks like this:

```
while(TRUE) {
wait(m);   //acquire lock
read_count++;
if(read_count == 1)
wait(w);
signal(m);  //release lock
   /* perform the
reading operation */
```

```
wait(m);   // acquire lock
read_count--;
if(read_count == 0)
signal(w);
signal(m);  // release lock
}
```

Code Explained:

- As seen above in the code for the writer, the writer just waits on the **w** semaphore until it gets a chance to write to the resource.
- After performing the write operation, it increments **w** so that the next writer can access the resource.
- On the other hand, in the code for the reader, the lock is acquired whenever the **read_count** is updated by a process.
- When a reader wants to access the resource, first it increments the **read_count** value, then accesses the resource and then decrements the **read_count** value.
- The semaphore **w** is used by the first reader which enters the critical section and the last reader which exits the critical section.
- The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.
- Similarly, when the last reader exits the critical section, it signals the writer using the **w** semaphore because there are zero readers now and a writer can have the chance to access the resource.

Program:

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

sem_tmutex,writeblock;
int data = 0,rcount = 0;
```

```
void *reader(void *arg)
{
  int f;
  f = ((int)arg);
  sem_wait(&mutex);
  rcount = rcount + 1;
  if(rcount==1)
   sem_wait(&writeblock);
  sem_post(&mutex);
  printf("Data read by the reader%d is %d\n",f,data);
  sleep(1);
  sem_wait(&mutex);
  rcount = rcount - 1;
  if(rcount==0)
   sem_post(&writeblock);
  sem_post(&mutex);
}

void *writer(void *arg)
{
  int f;
  f = ((int) arg);
  sem_wait(&writeblock);
  data++;
  printf("Data writen by the writer%d is %d\n",f,data);
  sleep(1);
  sem_post(&writeblock);
}

main()
{
  inti,b;
  pthread_trtid[5],wtid[5];
  sem_init(&mutex,0,1);
  sem_init(&writeblock,0,1);
  for(i=0;i<=2;i++)
  {
    pthread_create(&wtid[i],NULL,writer,(void *)i);
    pthread_create(&rtid[i],NULL,reader,(void *)i);
  }
  for(i=0;i<=2;i++)
  {
    pthread_join(wtid[i],NULL);
    pthread_join(rtid[i],NULL);
  }
}
```

PROGRAM NO 8: Implement distance vector routing protocol

In computer communication theory relating to packet-switched networks, a **distance-vector routing protocol** is one of the two major classes of routing protocols, the other major class being the link-state protocol. A distance-vector routing protocol uses the Bellman-Ford algorithm to calculate paths.

A distance-vector routing protocol requires that a router informs its neighbors of topology changes periodically and, in some cases, when a change is detected in the topology of a network. Compared to link-state protocols, which require a router to inform all the nodes in a network of topology changes, distance-vector routing protocols have less computational complexity and message overhead.

Distance Vector means that Routers are advertised as vector of distance and direction. 'Direction' is represented by next hop address and exit interface, whereas 'Distance' uses metrics such as hop count.

Routers using distance vector protocol do not have knowledge of the entire path to a destination. Instead DV uses two methods:

1. Direction in which or interface to which a packet should be forwarded.
2. Distance from its destination.

Examples of distance-vector routing protocols include Routing Information Protocol Version 1 & 2, RIPv1 and RIPv2 and IGRP. EGP and BGP are not pure distance-vector routing protocols because a distance-vector protocol calculates routes based only on link costs whereas in BGP, for example, the local route preference value takes priority over the link cost.

```
#include<stdio.h>

struct node
{
   unsigned dist[20];
   unsigned from[20];
}rt[10];

int main()
{
   int dmat[20][20];
   int n,i,j,k,count=0;
   printf("\nEnter the number of nodes : ");
   scanf("%d",&n);
   printf("\nEnter the cost matrix :\n");
   for(i=0;i<n;i++)
      for(j=0;j<n;j++)
      {
         scanf("%d",&dmat[i][j]);
         dmat[i][i]=0;
         rt[i].dist[j]=dmat[i][j];
         rt[i].from[j]=j;
```

```
        }
        do
        {
           count=0;
           for(i=0;i<n;i++)
           for(j=0;j<n;j++)
           for(k=0;k<n;k++)
              if(rt[i].dist[j]>dmat[i][k]+rt[k].dist[j])
              {
                 rt[i].dist[j]=rt[i].dist[k]+rt[k].dist[j];
                 rt[i].from[j]=k;
                 count++;
              }
        }while(count!=0);
        for(i=0;i<n;i++)
        {
           printf("\n\nState value for router %d is \n",i+1);
           for(j=0;j<n;j++)
           {
              printf("\t\nnode %d via %d Distance%d",j+1,rt[i].from[j]+1,rt[i].dist[j]);
           }
        }
     printf("\n\n");
}


a.out:distanceVector.c
     gcc -ggdb distanceVector.c
PHONY:clean
clean:
     rm a.out *~
```

output

Enter the number of nodes : 4

Enter the cost matrix :
0 3 5 99
3 0 99 1
5 4 0 2
99 1 2 0

State value for router 1 is

node 1 via 1 Distance0
node 2 via 2 Distance3
node 3 via 3 Distance5
node 4 via 2 Distance4

State value for router 2 is

node 1 via 1 Distance3
node 2 via 2 Distance0
node 3 via 4 Distance3
node 4 via 4 Distance1

State value for router 3 is

node 1 via 1 Distance5
node 2 via 4 Distance3
node 3 via 3 Distance0
node 4 via 4 Distance2

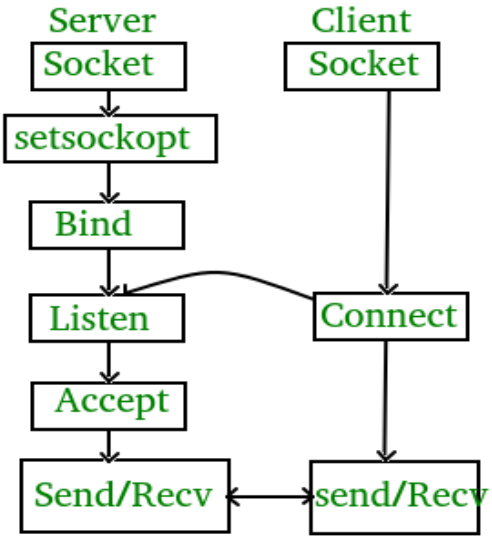
State value for router 4 is

node 1 via 2 Distance4
node 2 via 2 Distance1
node 3 via 3 Distance2
node 4 via 4 Distance0

SOCKET PROGRAMMING

Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while other socket reaches out to the other to form a connection. Server forms the listener socket while client reaches out to the server.

State diagram for server and client model



Stages for server

- **Socket creation:**

```
intsockfd = socket(domain, type, protocol)
```

sockfd: socket descriptor, an integer (like a file-handle)
domain: integer, communication domain e.g., AF_INET (IPv4 protocol) , AF_INET6 (IPv6 protocol)
type: communication type
SOCK_STREAM: TCP(reliable, connection oriented)
SOCK_DGRAM: UDP(unreliable, connectionless)
protocol: Protocol value for Internet Protocol(IP), which is 0. This is the same number which appears on protocol field in the IP header of a packet.(man protocols for more details)

- **Setsockopt:**

```
intsetsockopt(intsockfd, int level, intoptname,
```

```
const void *optval, socklen_toptlen);
```

This helps in manipulating options for the socket referred by the file descriptor sockfd. This is completely optional, but it helps in reuse of address and port. Prevents error such as: “address already in use”.

- **Bind:**

- int bind(intsockfd, conststructsockaddr *addr,

```
socklen_taddrlen);
```

After creation of the socket, bind function binds the socket to the address and port number specified in addr(custom data structure). In the example code, we bind the server to the localhost, hence we use INADDR_ANY to specify the IP address.

- **Listen:**

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

It puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection. The backlog, defines the maximum length to which the queue of pending connections for sockfd may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED.

- **Accept:**

```
intnew_socket= accept(intsockfd, structsockaddr *addr, socklen_t *addrlen);
```

It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket. At this point, connection is established between client and server, and they are ready to transfer data.

Stages for Client

- **Socket connection:** Exactly same as that of server's socket creation
- **Connect:**

- int connect(intsockfd, conststructsockaddr *addr,

```
socklen_taddrlen);
```

The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. Server's address and port is specified in addr.

PROGRAM NO 9: Implement client server communication using TCP Socket

Tcp Server

```
# include<stdio.h>
# include<string.h>
# include<sys/socket.h>
# include<stdlib.h>
# include<netdb.h>
int main()
{
  int serversocket,clientsocket,port;
  struct sockaddr_in serveraddr,clientaddr;
  socklen_t len;
  char message[50];
  struct serveraddr;
  serversocket=socket(AF_INET,SOCK_STREAM,0);
  bzero((char*)&serveraddr,sizeof(serveraddr));
  serveraddr.sin_family=AF_INET;
  printf("Enter the port number ");
  scanf("%d",&port);
  serveraddr.sin_port=htons(port);
  serveraddr.sin_addr.s_addr=INADDR_ANY;
  bind(serversocket,(struct sockaddr*)&serveraddr,sizeof(serveraddr));
  bzero((char*)&clientaddr,sizeof(clientaddr));
  len=sizeof(clientaddr);
  listen(serversocket,5);
  printf("\nWaiting for client connection\n");printf("\nhai:");
  clientsocket=accept(serversocket,(struct sockaddr*)&clientaddr,&len);
  printf("\nClient connectivity received.\n");
  printf("\nReading message from the client.\n");
```

```
  read(clientsocket,message,sizeof(message));
  printf("\nThe client has sent.%s",message);
  printf("\nSending message to the client.\n");
  write(clientsocket,"YOUR MESSAGE RECEIVED.",sizeof("YOUR MESSAGE
RECEIVED."));
  close(clientsocket);
  close(serversocket);
}
```

Tcp Client

```
#include<stdio.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<netdb.h>
#include<strings.h>
int main()
{
  int clientsocket,port;
  struct sockaddr_in serveraddr;
  socklen_t len;
  char message[50];
  clientsocket=socket(AF_INET,SOCK_STREAM,0);
  bzero((char*)&serveraddr,sizeof(serveraddr));
  len=sizeof(serveraddr);
  serveraddr.sin_family=AF_INET;
  printf("Enter the port number ");
  scanf("%d",&port);
  serveraddr.sin_port=htons(port);
  printf("\nTrying to connect to the server.\n");
  connect(clientsocket,(struct sockaddr*)&serveraddr,sizeof(serveraddr));
```

```
  printf("\nConnected to the server.\n");
  printf("\nSending message for server connection");
  send(clientsocket,"HI,IAM CLIENT...",sizeof("HI,IAM CLIENT..."),0);
  printf("\nReceiving message from server.\n");
  recv(clientsocket,message,sizeof(message),0);
  printf("\nMessage received.\t%s\n",message);
  close(clientsocket);
}
```

OUTPUT

SERVER

```
gcc tcpnews.c -o  server
./server
Enter the port number 5016
Waiting for client connection

hai:
Client connectivity received.
Reading message from the client.
The client has sent. HI,I AM CLIENT...
Sending message to the client.
```

CLIENT

```
gcc tcpnews.c -o client
./client
Enter the port number 5016
Trying to connect to the server.
Connected to the server.
Sending message for server connection
Receiving message from server.
```

PROGRAM NO 10: Implement client server communication using UDP Socket

UDP CLIENT

```
#include<netinet/in.h>
#include<netdb.h>
#include<strings.h>
int main()
{
//variable to store the socket_id.
 int clientsocket,port;
//variable to store the address.
 struct sockaddr_in serveraddr;
//variableto store the address length.
 socklen_t len;
//variable to store the network byte order address.
 struct hostent *server;
 char message[50];
//socket creation.
 clientsocket=socket(AF_INET,SOCK_DGRAM,0);
//steps involved in the server address creation.
 bzero((char*)&serveraddr,sizeof(serveraddr));
 len=sizeof(serveraddr);
 serveraddr.sin_family=AF_INET;
printf("Enter the port number ");
          scanf("%d",&port);
  serveraddr.sin_port=htons(port);
 //server=gethostbyname("127.0.0.1");
 //bcopy((char*)server->h_addr,(char*)&serveraddr.sin_addr.s_addr,sizeof(server->h_addr));
 fgets(message,2,stdin);
 printf("\nSending message for server connection\n");
//sending message.
```

```
 sendto(clientsocket,"HI I AM CLIENT...",sizeof("HI I AM CLIENT...."),0,(struct
sockaddr*)&serveraddr,sizeof(serveraddr));
 printf("\nReceiving message from server.\n");
//receiving messages.
recvfrom(clientsocket,message,sizeof(message),0,(struct sockaddr*)&serveraddr,&len);
 printf("\nMessage received:\t%s\n",message);
 close(clientsocket);
}
```

UDP SERVER

```
#include<sys/types.h>
#include<netinet/in.h>
#include<netdb.h>
int main()
{
        //variable to store the socket_id.
 int serversocket,port;
//variable to store the network addresses.
 struct sockaddr_in serveraddr,clientaddr;
        //variable to store the address length.
 socklen_t len;
 char message[50];
//socket creation.
 serversocket=socket(AF_INET,SOCK_DGRAM,0);
//steps involved in defining the serveraddress.
 bzero((char*)&serveraddr,sizeof(serveraddr));
 serveraddr.sin_family=AF_INET;
printf("Enter the port number ");
        scanf("%d",&port);
  serveraddr.sin_port=htons(port);
 serveraddr.sin_addr.s_addr=INADDR_ANY;
```

```
//binding the socket to the operating system.
 bind(serversocket,(struct sockaddr*)&serveraddr,sizeof(serveraddr));
 printf("\nWaiting for the client connection\n");
 bzero((char*)&clientaddr,sizeof(clientaddr));
 len=sizeof(clientaddr);
//receiving message from the client.
 recvfrom(serversocket,message,sizeof(message),0,(struct sockaddr*)&clientaddr,&len);
 printf("\nConnection received from client.\n");
 printf("\nThe client has send:\t%s\n",message);
 printf("\nSending message to the client.\n");
//sending message to the client.
 sendto(serversocket,"YOUR MESSAGE RECEIVED.",sizeof("YOUR MESSAGE
RECEIVED."),0,( struct sockaddr*)&clientaddr,sizeof(clientaddr));
 close(serversocket);
}
```

SERVER OUTPUT

```
gcc udpserver.c -o server
./server
Enter the port number 5012
Waiting for the client connection
Connection received from client.

The client has send:    HI I AM CLIENT...

Sending message to the client.
```

CLIENT OUTPUT

gcc udpclient.c -o client

./client

Enter the port number 5012

Sending message for server connection

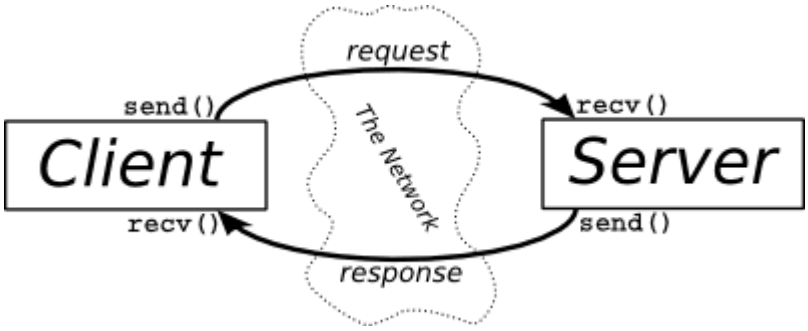
Receiving message from server.

Message received: YOUR MESSAGE RECEIVED.

PROGRAM NO 11: Implement multi user/ client chat server using TCP as transport layer protocol

In a multi client chat server, N clients are connected to a server and send messages. In this program, one of the clients send messages to the server and it will send back the messages to all other clients. I implemented it using TCP.

A simple Client-Server Interaction



Server:

In the server program, first is the establishment of connection to a port. we get the socket file descriptor ,

int socket(int domain, int type, int protocol);

'setsockopt' is used for losing the pesky "Address already in use" error message. Once we have a socket, we might have to associate that socket with a port on our local machine. This can be done by using the 'bind'.

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);

We want to wait for incoming connections, we use 'listen' in this situation.

int listen(int sockfd, int backlog);

sockfd is the usual socket file descriptor from the socket() system call. backlog is the number of connections allowed on the incoming queue .

In the case of a server, it wants to listen for incoming connections as well as keep reading from the connections it already have. select() gives the power to monitor several sockets at the same time. It'll tell you which ones are ready for reading, which are ready for writing, and which sockets have raised exceptions.

int select(int numfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);

If we want to see if we can read from standard input and some socket descriptor, sockfd, just add the file descriptors 0 and sockfd to the set readfds. The parameter numfds should be set to

the values of the highest file descriptor plus one. When select() returns, readfds will be modified to reflect which of the file descriptors we

selected which is ready for reading. After 'select', it run through the existing connections looking for data to read. If we got one, new connections are handled and the new file descriptor is added to the master set by keeping track of the maximum file descriptor. If there is no need to handle new connection, handle data from a client. If there is any data in the recv_buf, it can be received by using recv().Or , the data is send to all other clients by using the function send().

Client:

In the client program, first is the establishment of connection to the server and running on the localhost. Connection is established by using connect(). Then select() is used for either reading or writing as in the server program. It sends message to the server from the keyboard input using stdin. If there is data in the recv_buf, it receives data using recv().

TCPserver

```
#include"stdio.h"
#include"stdlib.h"
#include"sys/types.h"
#include"sys/socket.h"
#include"string.h"
#include"netinet/in.h"

#define PORT 4444
#define BUF_SIZE 2000
#define CLADDR_LEN 100

void main() {
 struct sockaddr_in addr, cl_addr;
 int sockfd, len, ret, newsockfd;
 char buffer[BUF_SIZE];
 pid_t childpid;
 char clientAddr[CLADDR_LEN];
```

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0) {
 printf("Error creating socket!\n");
 exit(1);
}
printf("Socket created...\n");

memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY;
addr.sin_port = PORT;

ret = bind(sockfd, (struct sockaddr *) &addr, sizeof(addr));
if (ret < 0) {
 printf("Error binding!\n");
 exit(1);
}
printf("Binding done...\n");

printf("Waiting for a connection...\n");
listen(sockfd, 5);

for (;;) { //infinite loop
 len = sizeof(cl_addr);
 newsockfd = accept(sockfd, (struct sockaddr *) &cl_addr, &len);
 if (newsockfd < 0) {
  printf("Error accepting connection!\n");
  exit(1);
 }
 printf("Connection accepted...\n");

 inet_ntop(AF_INET, &(cl_addr.sin_addr), clientAddr, CLADDR_LEN);
```

```
  if ((childpid = fork()) == 0) { //creating a child process


   close(sockfd);
//stop listening for new connections by the main process.
//the child will continue to listen.
//the main process now handles the connected client.


   for (;;) {
    memset(buffer, 0, BUF_SIZE);
    ret = recvfrom(newsockfd, buffer, BUF_SIZE, 0, (struct sockaddr *) &cl_addr, &len);
    if(ret < 0) {
     printf("Error receiving data!\n");
     exit(1);
    }
    printf("Received data from %s: %s\n", clientAddr, buffer);

    ret = sendto(newsockfd, buffer, BUF_SIZE, 0, (struct sockaddr *) &cl_addr, len);
    if (ret < 0) {
     printf("Error sending data!\n");
     exit(1);
    }
    printf("Sent data to %s: %s\n", clientAddr, buffer);
   }
  }
  close(newsockfd);
 }
}
```

TCPclient

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<string.h>
#include<netinet/in.h>
#include<netdb.h>

#define PORT 4444
#define BUF_SIZE 2000

int main(int argc, char**argv) {
 struct sockaddr_in addr, cl_addr;
 int sockfd, ret;
 char buffer[BUF_SIZE];
 struct hostent * server;
 char * serverAddr;

 if (argc < 2) {
  printf("usage: client < ip address >\n");
  exit(1);
 }
 serverAddr = argv[1];

 sockfd = socket(AF_INET, SOCK_STREAM, 0);
 if (sockfd < 0) {
  printf("Error creating socket!\n");
  exit(1);
 }
 printf("Socket created...\n");
```

```
 memset(&addr, 0, sizeof(addr));
 addr.sin_family = AF_INET;
 addr.sin_addr.s_addr = inet_addr(serverAddr);
 addr.sin_port = PORT;

 ret = connect(sockfd, (struct sockaddr *) &addr, sizeof(addr));
 if (ret < 0) {
  printf("Error connecting to the server!\n");
  exit(1);
 }
 printf("Connected to the server...\n");

 memset(buffer, 0, BUF_SIZE);
 printf("Enter your message(s): ");

 while (fgets(buffer, BUF_SIZE, stdin) != NULL) {
  ret = sendto(sockfd, buffer, BUF_SIZE, 0, (struct sockaddr *) &addr, sizeof(addr));
  if (ret < 0) {
   printf("Error sending data!\n\t-%s", buffer);
  }
  ret = recvfrom(sockfd, buffer, BUF_SIZE, 0, NULL, NULL);
  if (ret < 0) {
   printf("Error receiving data!\n");
  } else {
   printf("Received: ");
   fputs(buffer, stdout);
   printf("\n");
  }
 }

 return 0;
}
```

OUTPUT

Terminal 1

gcc tcpserver.c

./a.out 192.168.1.159 // use ip address of the system

Socket created...

Binding done...

Waiting for a connection...

Connection accepted...

Received data from 192.168.1.159: hi

Sent data to 192.168.1.159: hi

Connection accepted...

Received data from 192.168.1.161: hello

Sent data to 192.168.1.161: hello

Terminal 2

gcc tcpclient.c

./a.out 192.168.1.160 // use ip address of the system

Socket created...

Connected to the server...

Enter your message(s): hi

Received: hi

Terminal 3

./a.out 192.168.1.160 // use ip address of the system

Socket created...

Connected to the server...

Enter your message(s): hello

Received: hello

CONCURRENT SERVERS

There are two main classes of servers, iterative and concurrent. An *iterative* server iterates through each client, handling it one at a time. A *concurrent* server handles multiple clients at the same time. The simplest technique for a concurrent server is to call the fork function, creating one child process for each client. An alternative technique is to use *threads* instead (i.e., light-weight processes).

The fork() function

The fork() function is the only way in Unix to create a new process. It is defined as follows:

```
#include <unist.h>

pid_t fork(void);
```

The function returns 0 if in child and the process ID of the child in parent; otherwise, -1 on error.

In fact, the function fork() is called once but returns *twice*. It returns once in the calling process (called the parent) with the process ID of the newly created process (its child). It also returns in the child, with a return value of 0. The return value tells whether the current process is the parent or the child.

Example

A typical concurrent server has the following structure:

```
pid_t pid;
int listenfd, connfd;
listenfd = socket(...);

/***fill the socket address with server's well known port***/

bind(listenfd, ...);
listen(listenfd, ...);

for ( ; ; ) {

  connfd = accept(listenfd, ...); /* blocking call */

  if ( (pid = fork()) == 0 ) {
```

```
    close(listenfd); /* child closes listening socket */

    /***process the request doing something using connfd ***/
    /* ..... */

    close(connfd);
    exit(0);  /* child terminates
   }
   close(connfd);  /*parent closes connected socket*/
}
}
```

When a connection is established, accept returns, the server calls fork, and the child process services the client (on the connected socket connfd). The parent process waits for another connection (on the listening socket listenfd. The parent closes the connected socket since the child handles the new client. The interactions among client and server are presented in Figure 4.

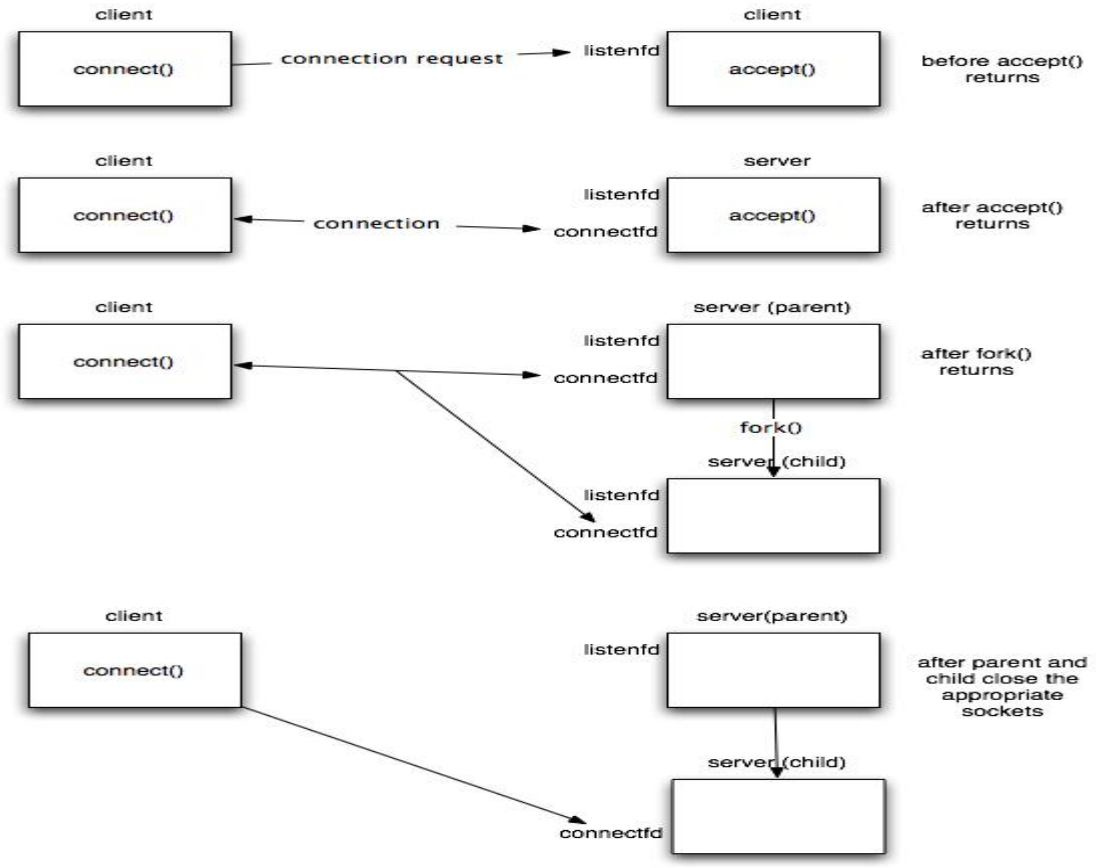


Figure 4: Example of interaction among a client and a concurrent server.

TCP Client/Server Examples

We now present a complete example of the implementation of a TCP based echo server to summarize the concepts presented above. We present an iterative and a concurrent implementation of the server.

We recommend that you run the client and server on different machines so there is a TCP connection over the Internet. However, you can also use a local TCP connection bewteen the client and server processes using the IP address **127.0.0.1** as the address given to the client. The **localhost** (meaning "this computer") is the standard hostname given to the address of the loopback network interface.

Please note that socket programming regularly resolve names of machines such as wildcat.cs.dartmouth.edu to a 32 bit IP address needed to make a connect(). In class we have interacted directly with the DNS (domain name server) using the host command:

```
$# you can use localhost or 127.0.0.1 for testing the client and server on the same machine

$ host localhost
localhost has address 127.0.0.1

$# find the name of the machine you are logged into

$ hostname
bear.cs.dartmouth.edu

$# find the IP address of the machine

$ host bear
bear.cs.dartmouth.edu has address 129.170.213.32
bear.cs.dartmouth.edu mail is handled by 0 mail.cs.dartmouth.edu.

$# If you have the dot IP address form you can find the name

$ host 129.170.213.32
32.213.170.129.in-addr.arpa domain name pointer bear.cs.dartmouth.edu.
```

Host allows us to get the host IP address by name or get the host name given the IP address.

Luckly you don't have to call "host" from your code. There are two commands that you can use:

```
struct hostent *gethostbyname(const char *name);

struct hostent *gethostbyaddr(const char *addr,  int len, int type);
```

PROGRAM NO 12: Implement concurrent time server program using UDP

UDP Server

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <time.h>
#include <sys/wait.h>
#include <signal.h>
#define MYPORT 1025 // the port users will be connecting to
#define BACKLOG 10 // how many pending connections queue will hold
void sigchld_handler(int s)
{
while(wait(NULL) > 0);
}
int main(void)
{
time_t timer;
int sockfd, new_fd; // listen on sock_fd, new connection on new_fd
struct sockaddr_in my_addr; // my address information
struct sockaddr_in their_addr; // connector.s address information
int sin_size;
struct sigaction sa;
int yes=1;
```

```
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
perror("socket");
exit(1);
}
if (setsockopt(sockfd,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1)
{
perror("setsockopt");
exit(1);
}
my_addr.sin_family = AF_INET; // host byte order
my_addr.sin_port = htons(MYPORT); // short, network byte order
my_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
memset(&(my_addr.sin_zero), 0, 8); // zero the rest of the struct
if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) == -1)
{
perror("bind");
exit(1);
}
if (listen(sockfd, BACKLOG) == -1)
{
perror("listen");
exit(1);
}
sa.sa_handler = sigchld_handler; // reap all dead processes
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;
if (sigaction(SIGCHLD, &sa, NULL) == -1)
{
perror("sigaction");
exit(1);
}
```

```
while(1) // main accept() loop
{
sin_size = sizeof(struct sockaddr_in);
if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr,&sin_size)) == -1)
{
//perror("accept");
continue;
}
printf("Received request from Client: %s:%d\n",
inet_ntoa(their_addr.sin_addr),MYPORT);
if (!fork())
{ // this is the child process
close(sockfd); // child doesn.t need the listener
timer = time(NULL);
if (send(new_fd, ctime(&timer), 30, 0) == -1)
perror("send");
close(new_fd);
exit(0);
}
close(new_fd); // parent doesn.t need this
}
return 0;
}
```

UDP Client

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

```
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#define PORT 1025 /*This is the port for the client server connection*/
#define MAXDATASIZE 500 // max number of bytes we can get at once
int main(int argc, char *argv[])
{
int sockfd, numbytes;
char buf[MAXDATASIZE];
struct hostent *he;
struct sockaddr_in their_addr; // connector.s address information
//**************************block of code finds the localhost IP
char hostn[400]; //placeholder for the hostname
char ipadd[400]; //placeholder for my IP address
struct hostent *hostIP; //placeholder for the IP address
//if the gethostname returns a name then the program will get the ip address using
gethostbyname
if((gethostname(hostn, sizeof(hostn))) == 0)
{
hostIP = gethostbyname(hostn); //the netdb.h function gethostbyname
}
else
{
printf("ERROR:FC4539 - IP Address not found."); //error if the hostname is not found
}
//*************************************************************
if (argc != 2)
{
fprintf(stderr,"usage: client hostname\n");
exit(1);
}
if ((he=gethostbyname(argv[1])) == NULL)
```

```
{ // get the host info
perror("gethostbyname");
exit(1);
}
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
perror("socket");
exit(1);
}
their_addr.sin_family = AF_INET; // host byte order
their_addr.sin_port = htons(PORT); // short, network byte order
their_addr.sin_addr = *((struct in_addr *)he->h_addr);
memset(&(their_addr.sin_zero), 0, 8); // zero the rest of the
if (connect(sockfd, (struct sockaddr *)&their_addr, sizeof(struct sockaddr)) == -1)
{
perror("connect");
exit(1);
}
if ((numbytes=recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1)
{
perror("recv");
exit(1);
}
buf[numbytes] = 0;
printf("\n\nLocalhost: %s\n", inet_ntoa(*(struct in_addr *)hostIP->h_addr));
printf("Local Port: %d\n", PORT);
printf("Remote Host: %s\n", inet_ntoa(their_addr.sin_addr));
printf("Received daytime data: %s\n",buf);
close(sockfd);
return 0;
}
```

OUTPUT

Terminal 1

gcc udpserver.c -o server

./server

Received request from Client: 192.168.1.161:1025

Terminal 2

./client 192.168.1.160

Localhost: 192.168.1.160

Local Port: 1025

Remote Host: 192.168.1.160

Received daytime data: Mon Jan 29 15:15:59 2018

PROGRAM NO 13: Implement concurrent file server

Concurrent Server handling Multiple Clients Code in C in Linux Platform

```
//TCP SERVER Concurrent: multiple clients can be handled by this server
//Now second client no need to wait for first client to close by finishing its task.
//This server now handles 5 clients simultaneously. i.e. Now up to 5 clients simultaneously can send and receive messages from this server.
//Here concurrency is achieved  by creating a new child process which process e each new client while parent continues to accepting new connections.
#include<stdio.h>
#include<sys/types.h>//socket
#include<sys/socket.h>//socket
#include<string.h>//memset
#include<stdlib.h>//sizeof
#include<netinet/in.h>//INADDR_ANY

#define PORT 8000
#define MAXSZ 100
int main()
{
intsockfd;//to create socket
intnewsockfd;//to accept connection

structsockaddr_inserverAddress;//server receive on this address
structsockaddr_inclientAddress;//server sends to client on this address

int n;
char msg[MAXSZ];
intclientAddressLength;
intpid;

//create socket
sockfd=socket(AF_INET,SOCK_STREAM,0);
//initialize the socket addresses
memset(&serverAddress,0,sizeof(serverAddress));
serverAddress.sin_family=AF_INET;
serverAddress.sin_addr.s_addr=htonl(INADDR_ANY);
serverAddress.sin_port=htons(PORT);

//bind the socket with the server address and port
bind(sockfd,(structsockaddr *)&serverAddress, sizeof(serverAddress));

//listen for connection from client
listen(sockfd,5);

while(1)
{
```

```
//parent process waiting to accept a new connection
printf("\n*****server waiting for new client connection:*****\n");
clientAddressLength=sizeof(clientAddress);
newsockfd=accept(sockfd,(structsockaddr*)&clientAddress,&clientAddressLength);
printf("connected to client: %s\n",inet_ntoa(clientAddress.sin_addr));

//child process is created for serving each new clients
pid=fork();
if(pid==0)//child process rec and send
{
//rceive from client
while(1)
{
n=recv(newsockfd,msg,MAXSZ,0);
if(n==0)
{
close(newsockfd);
break;
}
msg[n]=0;
send(newsockfd,msg,n,0);

printf("Receive and set:%s\n",msg);
}//close interior while
exit(0);
}
else
{
close(newsockfd);//sock is closed BY PARENT
}
}//close exterior while

return 0;
}
/*
*****server waiting for new client connection:*****
connected to client: 127.0.0.1

*****server waiting for new client connection:*****
Receive and set:India

Receive and set:is

Receive and set:Great

connected to client: 127.0.0.1
```

```
*****server waiting for new client connection:*****
Receive and set:The

Receive and set:USA
 */

/************Client Code***********/
//CLIENT ITERATIVE

#include<stdio.h>
#include<sys/types.h>//socket
#include<sys/socket.h>//socket
#include<string.h>//memset
#include<stdlib.h>//sizeof
#include<netinet/in.h>//INADDR_ANY

#define PORT 8000
#define SERVER_IP "127.0.0.1"
#define MAXSZ 100
int main()
{
intsockfd;//to create socket

structsockaddr_inserverAddress;//client will connect on this

int n;
char msg1[MAXSZ];
char msg2[MAXSZ];

//create socket
sockfd=socket(AF_INET,SOCK_STREAM,0);
//initialize the socket addresses
memset(&serverAddress,0,sizeof(serverAddress));
serverAddress.sin_family=AF_INET;
serverAddress.sin_addr.s_addr=inet_addr(SERVER_IP);
serverAddress.sin_port=htons(PORT);

//client  connect to server on port
connect(sockfd,(structsockaddr *)&serverAddress,sizeof(serverAddress));
//send to sever and receive from server
while(1)
{
printf("\nEnter message to send to server:\n");
fgets(msg1,MAXSZ,stdin);
if(msg1[0]=='#')
```

```
break;

n=strlen(msg1)+1;
send(sockfd,msg1,n,0);

n=recv(sockfd,msg2,MAXSZ,0);

printf("Receive message from  server::%s\n",msg2);
}

return 0;
}
/*
client:

ms@ubuntu:~/ipc_socket$ gcctcpclient.c -o c
ms@ubuntu:~/ipc_socket$ ./c

Enter message to send to server:
India
Receive message from  server::India


Enter message to send to server:
IS
Receive message from  server::IS


Enter message to send to server:
Great
Receive message from  server::Great


Enter message to send to server:
*/
```

PROGRAM NO 14: Configure the following services in the network- TELNET, SSH, FTP server, File server.

TELNET

1. Install package *sudo apt-get install telnetd*
2. Change the telnet port number by using command *sudo gedit etc/services*. Change the constant TELNET port number 23 to any random number such as 22222
3. sudo gedit etc/inetd.conf
4. Restart the server using command *sudo service inetd restart* to remove any error
5. If we know the IP address and password of another machine, we can connect to it and send message using command *telnet ipaddress*
6. *sudo service inetd stop*

SSH

1. Install package *sudo apt-get install openssh-server*
2. *sudo gedit etc/ssh/sshd_config*
3. open the conf file change port number 22 to 22222 and password to no
4. Restart the server using command *sudo service inetd restart* to remove any error
5. *sudo systemctl status ssh* to know the current status of the connection
6. If no connection then use command *sudo apt-get install*
7. *ssh user@ip_address –p22* where user is the user name and ip_address is IP of system and 22 is port number. For eg: ssh user@172.20.35.125 –p22
8. Connected

FTP

1. *sudo apt-get update*
2. *sudo apt-get install vsftpd*
3. add a test user: *sudo adduser mili*
4. *sudo mkdir /home/mili/ftp*
5. *sudo chown nobody:nogroup /home/mili/ftp*
6. *sudo ls –la/home/mili/ftp*
7. create another directory file *sudo mkdir /home/mili/ftp/sample*
8. change permission *sudo chown mili:mili /home/mili/ftp/sample*
9. *echo "WELCOME"|sudo tee /home/mili/ftp/sample/sample.txt*

10. configuring ftp file *sudo gedit /etc/vsftpd.conf* and edit contents
11. *echo "mili"|sudo tee –a /etc/vsftpd.chroot_list* for adding to list
12. *cat /etc/vsftpd.chroot_list*
13. *sudo systemctl restart vsftpd*
14. Now from a client machine: *ftp –p ipaddress*
15. Enter username and password
16. *ftp>cd sample*
17. *ftp>get sample.txt*
18. the file will be downloaded in our client machine

FILE SERVER

1. *sudo apt install samba*
2. *sudo gedit /etc/samba/smb.conf*
3. Add a directory for sharing

   [share] //in guest

   comment=ubuntufileserver

   path= /srv/samba/share

   browseable=yes

   guest ok=yes

   readonly=no

   createmask=0777
4. *sudo mkdir -p /srv/samba/share*
5. *sudo chown nobody:nogroup /srv/ samba/share*
6. *sudo systemctl restart smbd.service.nmtd.service*
7. Check in network