LAB : 1

TCP Client Server Communication using sockets

Description:

Sockets provide the communication mechanism between two computers. A client program creates a socket on its end of the communication and attempts to connect that socket to a server.

When the connection is made, the server creates a socket object on its end of the communication. The client and server can now communicate by writing to and reading from the socket.

Notice that the client needs to know of the existence of and the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established.

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

1.Create a socket with the **socket()** system call

2.Connect the socket to the address of the server using the **connect()** system call

3.Send and receive data. There are a number of ways to do this, but the simplest is to use the **read()** and **write()** system calls.

The steps involved in establishing a socket on the server side are as follows:

1.Create a socket with the **socket()** system call

2.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.

3.Listen for connections with the **listen()** system call

4.Accept a connection with the **accept()** system call. This call typically blocks until a client connects with the server.

5.Send and receive data

In this program, client wishes to communicate with the server by sending it a message. The server acknowledges the client input by sending it a copy of the same string along with the client IP

Program (Server Side):

```
// Server program to receive a string from client and communicate with it.

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
```

```
#include <string.h>
#include <strings.h>
#include <sys/types.h>
#include <time.h>
#include <stdbool.h>

#define MAX 1024
#define PORT 25000
#define MAX_BACKLOG 10

int main(){
    // fd for server side and one fd for the client side
    int listenfd = 0, connfd = 0;

    // A socket address structure to hold in the socket
    struct sockaddr_in serv_addr = {0};

    // Strings to manipulate incoming and outgoing data
    char recvBuff[MAX] = {' '};

    // Creates a socket i.e file descriptor within the process table
    listenfd = socket(AF_INET, SOCK_STREAM, 0);

    // initialise members of the socket's address structure
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(PORT);

    // Assign protocol to a socket :=
    bind(listenfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

    // listen for incoming connections with a specified backlog queue
    listen(listenfd, MAX_BACKLOG);

    while(true){
        // accept incoming client connections to the given socket
        connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);

        // Read the message from the socket into a string
        int readChar = read(connfd, recvBuff, MAX);

        printf("\n New Client Connected with message: \n");
        for (int i = 0 ; i < readChar; i++)
            printf("%c",recvBuff[i]);
```

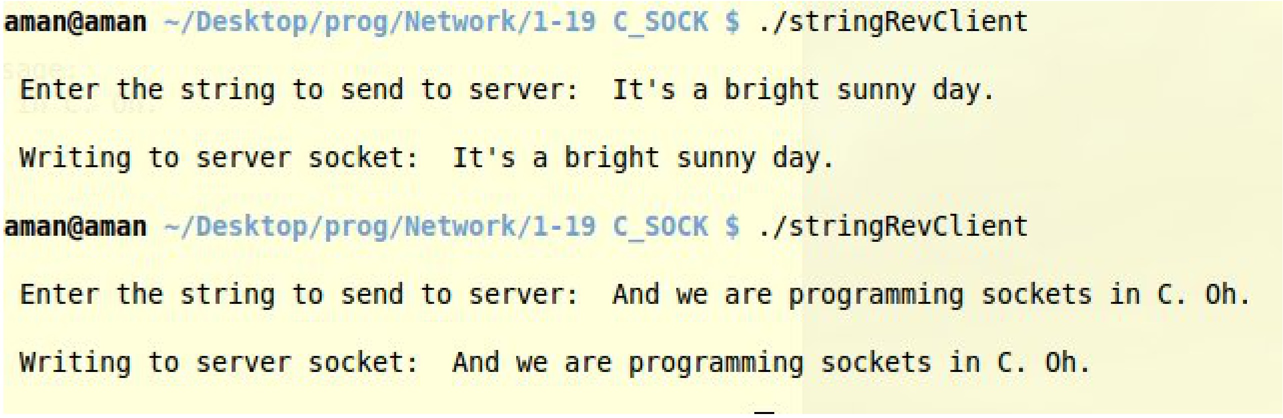
```
      close(connfd);
      sleep(1);
    }
}
```

Program (Client Side):

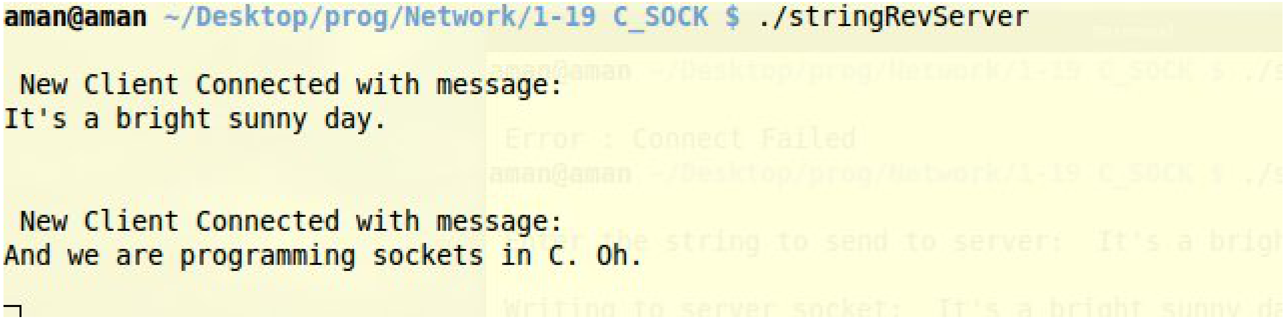
```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>

#define localhost "127.70.0.10"
#define MAX 1024
#define PORT 25000

int main(){
   // Set up client sockets, character buffers
   int sockfd = 0, n = 0;
   char sendBuff[MAX] = {' '};

   // A socket address structure to hold in the socket
   struct sockaddr_in serv_addr = {0};

   // mention domain, stream/datagram and default protocol
   sockfd = socket(AF_INET, SOCK_STREAM, 0);

   if(sockfd < 0){
      perror("\n Error : Could not create socket \n");
      return 1;
   }

   // initialise members of the socket's address structure
   serv_addr.sin_family = AF_INET;
   serv_addr.sin_port = htons(PORT);

   if(inet_pton(AF_INET, localhost, &serv_addr.sin_addr)<=0){
      printf("\n inet_pton error occured\n");
      return 1;
   }

   if( connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0){
     printf("\n Error : Connect Failed \n");
     return 1;
   }
```

```
    // Read from input and write to socket
    fprintf(stdout, "\n Enter the string to send to server:  ");
    fgets(sendBuff, MAX, stdin);
    printf("\n Writing to server socket:  %s", sendBuff);

    n = write(sockfd, sendBuff, strlen(sendBuff)) ;
    if(n < 0)
        perror("\n Write error \n");

    printf("\n");
    return 0;
}
```

OUTPUT:



```
aman@aman ~/Desktop/prog/Network/1-19 C_SOCK $ ./stringRevClient

 Enter the string to send to server:  It's a bright sunny day.

 Writing to server socket:  It's a bright sunny day.

aman@aman ~/Desktop/prog/Network/1-19 C_SOCK $ ./stringRevClient

 Enter the string to send to server:  And we are programming sockets in C. Oh.

 Writing to server socket:  And we are programming sockets in C. Oh.
```



```
aman@aman ~/Desktop/prog/Network/1-19 C_SOCK $ ./stringRevServer

 New Client Connected with message:
It's a bright sunny day.


 New Client Connected with message:
And we are programming sockets in C. Oh.
```

LAB : 2

TCP Client Server Program To Reverse A String

Description:

Sockets provide the communication mechanism between two computers. A client program creates a socket on its end of the communication and attempts to connect that socket to a server.

When the connection is made, the server creates a socket object on its end of the communication. The client and server can now communicate by writing to and reading from the socket.

The client needs to know of the existence of and the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established.

In this program, client wishes to communicate with the server by sending it a message. The server replies by sending it a reversed version of the message, following the TCP protocol.

Program (Server Side):

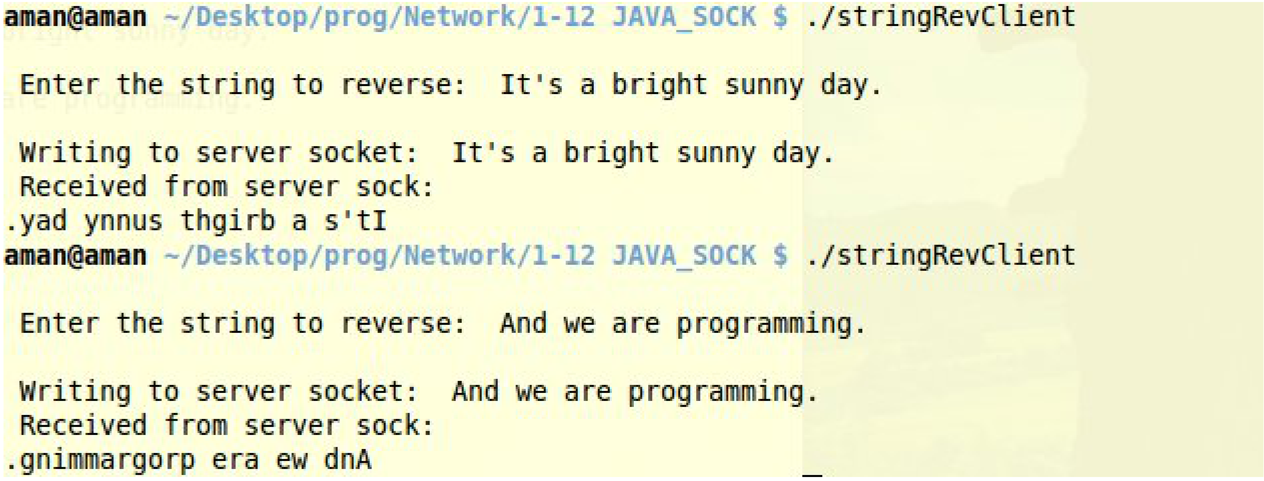
```
// Server program to receive a string from client and send it's reverse

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <strings.h>
#include <sys/types.h>
#include <time.h>

#define MAX 1024
#define PORT 25000
#define MAX_BACKLOG 10


int main(){
   int listenfd = 0, connfd = 0;
   struct sockaddr_in serv_addr = {0};

   char recvBuff[MAX] = {' '};
   char reversedStr[MAX] = {' '};
```

```
    listenfd = socket(AF_INET, SOCK_STREAM, 0);

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(PORT);

    bind(listenfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
    listen(listenfd, MAX_BACKLOG);

    while(1){
        connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);
        int readChar = read(connfd, recvBuff, MAX), count = 0;

        bzero(reversedStr, MAX);
        fprintf(stdout,"\n Received from client: %s",recvBuff );
        for (int i = readChar - 1; i >= 0; i--)
            reversedStr[count ++] = recvBuff[i];

        write(connfd, reversedStr, readChar);

        close(connfd);
     }
}
```

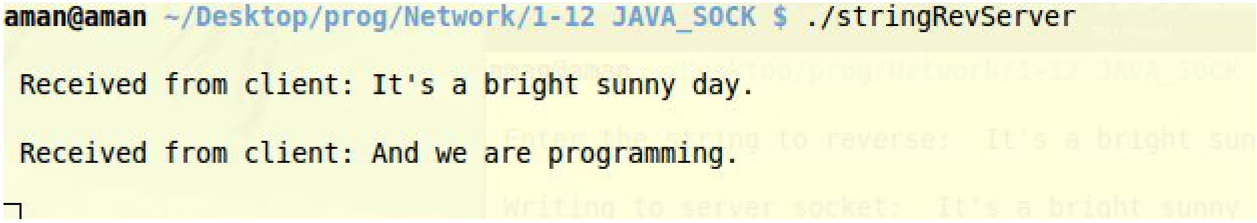
Program (Client Side):

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

#define localhost "127.0.0.10"
#define MAX 1024
#define PORT 25000

int main(){
    // Set up client sockets, buffers and socket structure
    int sockfd = 0, n = 0;
    char recvBuff[MAX] = {' '};
    char sendBuff[MAX] = {' '};
    struct sockaddr_in serv_addr = {0};

    // mention domain, stream/datagram and default protocol
    if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
        perror("\n Error : Could not create socket \n");
        return 1;
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    if( connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0){
       printf("\n Error : Connect Failed \n");
       return 1;
    }

    // Read from input and write to socket
    fprintf(stdout, "\n Enter the string to reverse:  ");
    fgets(sendBuff, MAX, stdin);
    printf("\n Writing to server socket:  %s", sendBuff);

    n = write(sockfd, sendBuff, strlen(sendBuff)) ;
    if(n < 0)
        perror("\n Write error \n");

    n = read(sockfd, recvBuff, MAX);
    printf(" Received from server sock: ");

    for (int i = 0; i < n; i++)
        printf("%c",recvBuff[i]);

    printf("\n");

    return 0;
}
```

OUTPUT:



```
aman@aman ~/Desktop/prog/Network/1-12 JAVA_SOCK $ ./stringRevClient

 Enter the string to reverse:  It's a bright sunny day.

 Writing to server socket:  It's a bright sunny day.
 Received from server sock:
.yad ynnus thgirb a s'tI
aman@aman ~/Desktop/prog/Network/1-12 JAVA_SOCK $ ./stringRevClient

 Enter the string to reverse:  And we are programming.

 Writing to server socket:  And we are programming.
 Received from server sock:
.gnimmargorp era ew dnA
```



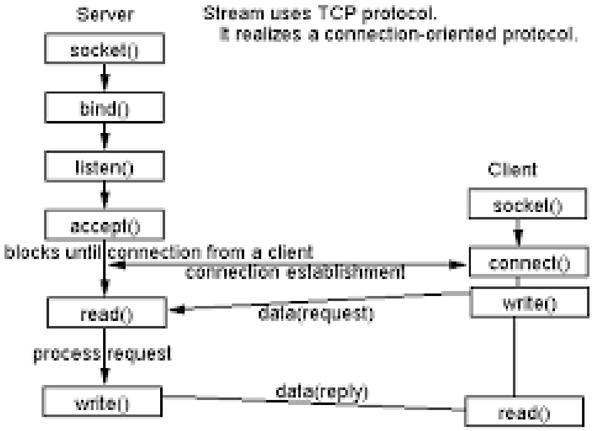
```
aman@aman ~/Desktop/prog/Network/1-12 JAVA_SOCK $ ./stringRevServer

 Received from client: It's a bright sunny day.

 Received from client: And we are programming.
```

LAB : 3

TCP Sockets - Date Time Server

Description:

The client program here, requests the current time from the server using a TCP connection. This can be done by making use of the **ctime()** defined in <time.h>. This program is equivalent to calling the Timer Server defined at port 13.



Program (Server Side):

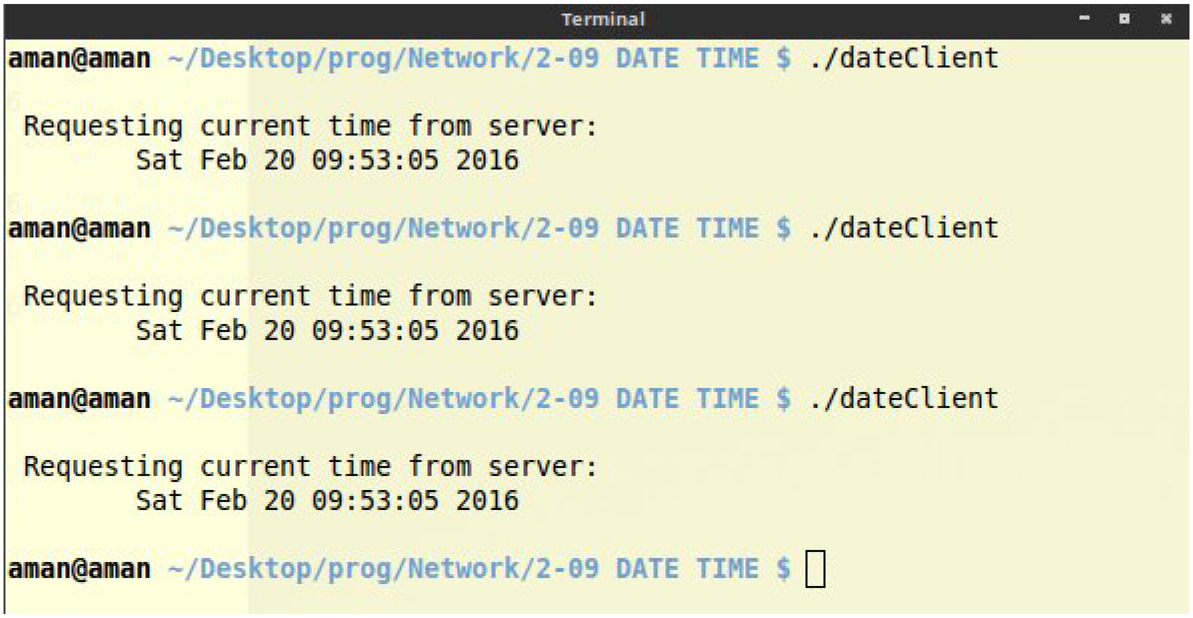
```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <strings.h>
#include <sys/types.h>
#include <time.h>
#include <stdbool.h>

#define MAX 1024
#define PORT 25000
#define MAX_BACKLOG 10


int main(){
    // fd for server side and one fd for the client side
    int listenfd = 0, connfd = 0;

    // A socket address structure to hold in the socket
    struct sockaddr_in serv_addr = {0};

    // Strings to manipulate incoming and outgoing data
    char sendBuff[MAX] = {' '};

    // Creates a socket i.e file descriptor within the process table
    listenfd = socket(AF_INET, SOCK_STREAM, 0);

    // initialise members of the socket's address structure
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(PORT);

    // Assign protocol to a socket :=
    bind(listenfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

    // listen for incoming connections with a specified backlog queue
    listen(listenfd, MAX_BACKLOG);

    time_t new_time = time(NULL);

    while(true){
        // accept incoming client connections to the given socket
        connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);

        // Read the message from the socket into a string
        snprintf(sendBuff, MAX, "%s", ctime(&new_time) );
        printf("\n Client connected. Sending Time : %s\n", sendBuff);
        write(connfd, sendBuff, strlen(sendBuff));

        close(connfd);
        sleep(1);
     }
}
```

Program (Client Side):

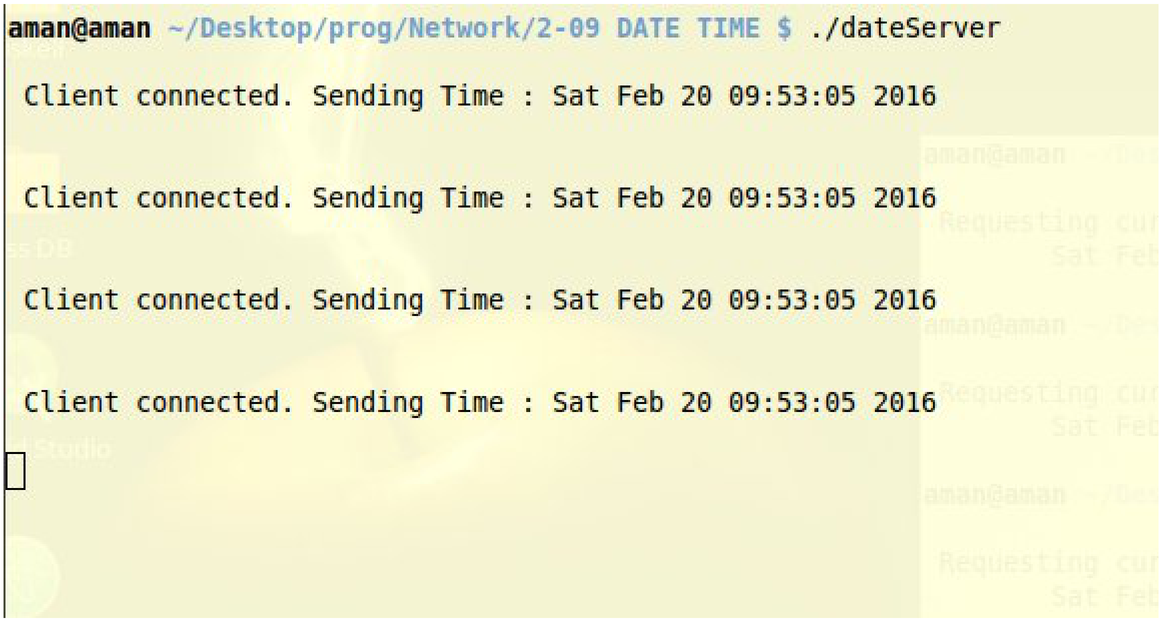
```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>
```

```
#define localhost "127.70.0.10"
#define MAX 1024
#define PORT 25000

int main(){
    // Set up client sockets, character buffers
    int sockfd = 0, n = 0;

   //Set up a string to hold the incoming response
    char recvBuff[MAX] = {' '};

    // A socket address structure to hold in the socket
    struct sockaddr_in serv_addr = {0};

    // mention domain, stream/datagram and default protocol
    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    if(sockfd < 0){
        perror("\n Error : Could not create socket \n");
        return 1;
    }

    // initialise members of the socket's address structure
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    // Read from input and write to socket
    fprintf(stdout, "\n Requesting current time from server:  \n\t");

    if(inet_pton(AF_INET, localhost, &serv_addr.sin_addr)<=0){
        printf("\n inet_pton error occured\n");
        return 1;
    }


    if( connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0){
       printf("\n Error : Connect Failed \n");
       return 1;
    }
    n = read(sockfd, recvBuff, MAX);
    for(int i = 0; i < n; i ++)
        printf("%c", recvBuff[i]);

    printf("\n");
    close(sockfd);
    return 0;
}
```

OUTPUT:



```
Terminal
aman@aman ~/Desktop/prog/Network/2-09 DATE TIME $ ./dateClient

 Requesting current time from server:
        Sat Feb 20 09:53:05 2016

aman@aman ~/Desktop/prog/Network/2-09 DATE TIME $ ./dateClient

 Requesting current time from server:
        Sat Feb 20 09:53:05 2016

aman@aman ~/Desktop/prog/Network/2-09 DATE TIME $ ./dateClient

 Requesting current time from server:
        Sat Feb 20 09:53:05 2016

aman@aman ~/Desktop/prog/Network/2-09 DATE TIME $ 
```



```
aman@aman ~/Desktop/prog/Network/2-09 DATE TIME $ ./dateServer

 Client connected. Sending Time : Sat Feb 20 09:53:05 2016


 Client connected. Sending Time : Sat Feb 20 09:53:05 2016


 Client connected. Sending Time : Sat Feb 20 09:53:05 2016


 Client connected. Sending Time : Sat Feb 20 09:53:05 2016

```

LAB : 4

TCP Client and Server Application to Transfer a file

Description:

Sockets enable us to communicate between the client and the server. The client program here, requests a file from the server using a TCP connection. Server has an input file called **"inputServer".** As soon as, a client connects, it sends the file stored in an array to the client. The client reads the information received from the socket, stores it in another array and writes that to a file called **"serverOutput"**. The program also depicts the IP addresses involved during the transfer.

Program (Server Side):

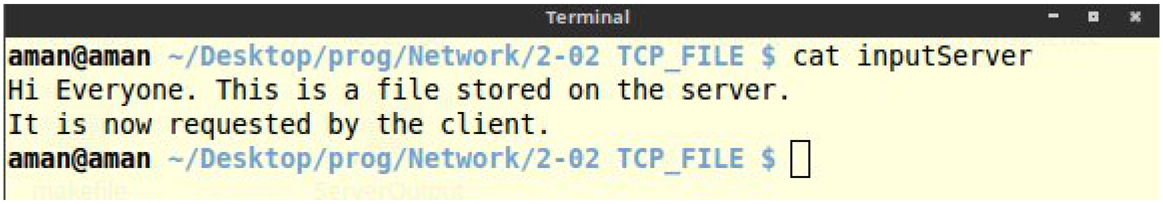
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#include <string.h>
#include <strings.h>
#include <sys/types.h>
#include <stdbool.h>
#include <assert.h>

#define MAX 1024
#define PORT 25000
#define MAX_BACKLOG 10
#define SERVER_INPUT_FILE "inputServer"

// Send a given input file to every client that wishes to connect

int main(){
    // Create a fd for server and one for client
    int sockFd = 0, clientFd = 0, readChar;

    // A socket address structure to hold in the socket
    struct sockaddr_in serv_addr = {0};

    // String to hold input file line by line for client
    char sendBuff[MAX];

    // Creates a TCP socket i.e file descriptor within the process table
    sockFd = socket(AF_INET, SOCK_STREAM, 0);

    // initialise members of the socket's address structure
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(PORT);

    // Assign protocol to a socket :=
    bind(sockFd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

    // listen for incoming connections with a specified backlog queue
    listen(sockFd, MAX_BACKLOG);

    // read input from file once
    FILE *fp = fopen(SERVER_INPUT_FILE, "r");
    assert (fp);
    while (!feof(fp))
        readChar = fread(sendBuff, 1, MAX, fp);

    fclose(fp);

    while(true){
        // Socket address structure to hold the client socket
        struct sockaddr_in addr;
        socklen_t addr_size = sizeof(struct sockaddr_in);

        // accept incoming client connections to the given socket descriptor clientFd
        clientFd = accept(sockFd, (struct sockaddr *)&addr, &addr_size);
        fprintf(stdout, "\nA new client has connected. Sending file\n");
        fprintf(stdout, "IP Address: %s\n", inet_ntoa(addr.sin_addr) );

        write(clientFd, sendBuff, readChar);
        close(clientFd);
        sleep(1);
    }

    return 0;
}
```

Program (Client Side):

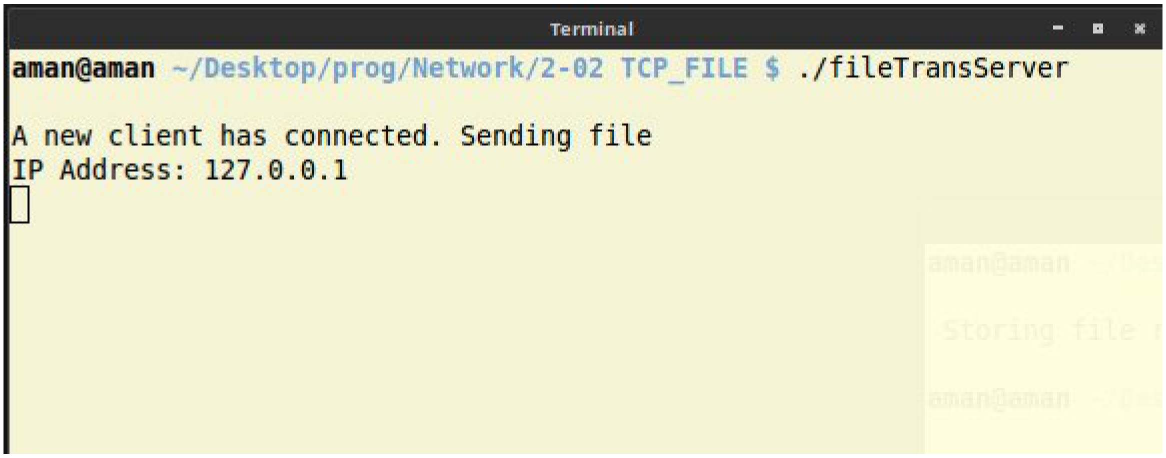
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#include <string.h>
#include <strings.h>
#include <sys/types.h>
```

```
#define localhost "127.0.0.10"
#define outputFile "ServerOutput"
#define MAX 1024
#define PORT 25000
// Connect to the server at the given location and create a file mentioning its inputs

int main(){
    // Set up client sockets, character buffers
    int sockFd = 0, n = 0;
    char recvBuff[MAX] = {' '};

    // A socket address structure to hold in the socket
    struct sockaddr_in serv_addr = {0};

    // Create TCP Socket: mention domain, stream/datagram and default protocol
    sockFd = socket(AF_INET, SOCK_STREAM, 0);

    if(sockFd < 0){
        perror("\n Error : Could not create socket \n");
        return 1;
    }

    // initialise members of the socket's address structure
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    if( connect(sockFd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0){
       printf("\n Error : Connect Failed \n");
       return 1;
    }

    // Read from input and write to socket
    fprintf(stdout, "\n Storing file received from server: %s \n\n", inet_ntoa(serv_addr.sin_addr));

    FILE * fp = fopen(outputFile, "w");
    assert(fp);

    // Read message from server
    n = read(sockFd, recvBuff, MAX);
    // Write the message received from server onto file
    fwrite(recvBuff, 1, n, fp);

    // Also display file contents
    write(0, recvBuff, n);

    fclose(fp);
    return 0;
}
```

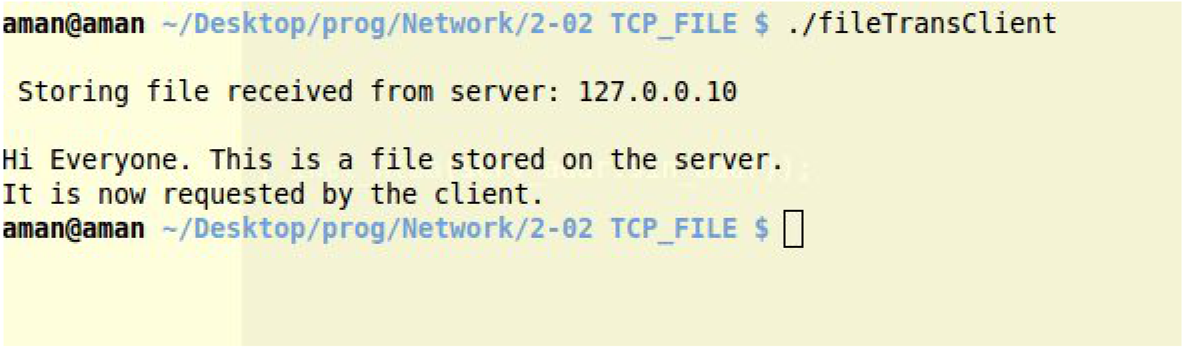
OUTPUT:



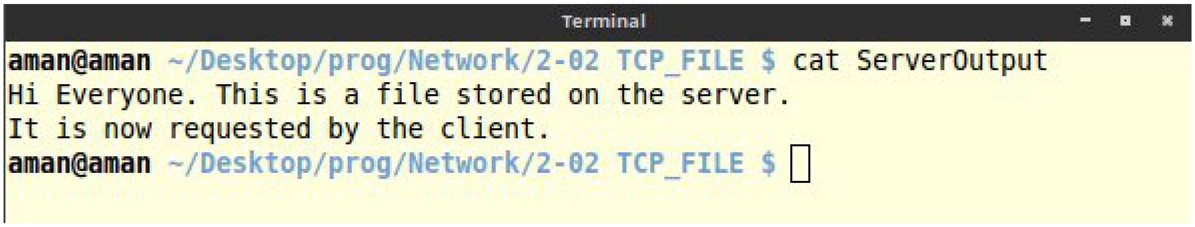
```
aman@aman ~/Desktop/prog/Network/2-02 TCP_FILE $ cat inputServer
Hi Everyone. This is a file stored on the server.
It is now requested by the client.
aman@aman ~/Desktop/prog/Network/2-02 TCP_FILE $ 
```



```
aman@aman ~/Desktop/prog/Network/2-02 TCP_FILE $ ./fileTransServer

A new client has connected. Sending file
IP Address: 127.0.0.1
```



```
aman@aman ~/Desktop/prog/Network/2-02 TCP_FILE $ ./fileTransClient

 Storing file received from server: 127.0.0.10

Hi Everyone. This is a file stored on the server.
It is now requested by the client.
aman@aman ~/Desktop/prog/Network/2-02 TCP_FILE $ 
```



```
aman@aman ~/Desktop/prog/Network/2-02 TCP_FILE $ cat ServerOutput
Hi Everyone. This is a file stored on the server.
It is now requested by the client.
aman@aman ~/Desktop/prog/Network/2-02 TCP_FILE $ 
```

LAB : 5

UDP Client Server program to transfer a file

Description:

UDP is a connection-less, unreliable, datagram protocol (TCP is instead connection-oriented, reliable and stream based). There are some instances when it makes to use UDP instead of TCP. Some popular applications built around UDP are DNS, NFS, SNMP and for example, some Skype services and streaming media.

The client does not establish a connection with the server. Instead, the client just sends a datagram to the server using the **sendto()** which requires the address of the destination as a parameter. Similarly, the server does not accept a connection from a client. Instead, the server just calls the **recvfrom()** function, which waits until data arrives from some client. **recvfrom()** returns the IP address of the client, along with the datagram, so the server can send a response to the client.

•Create a socket using the socket() function;

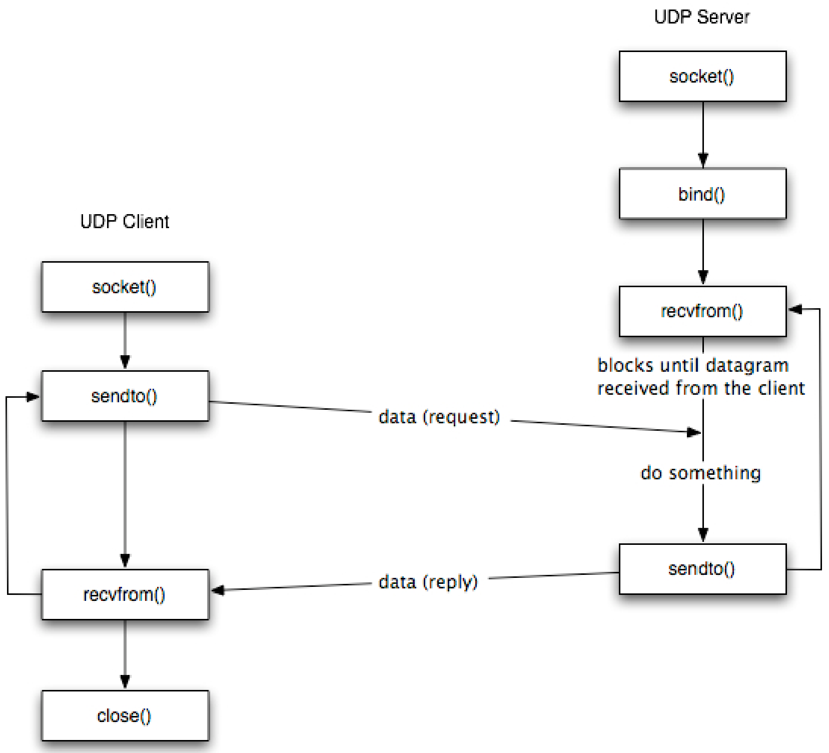
•Send and receive data by means of the **recvfrom()** and **sendto()** functions.

The steps of establishing a UDP socket communication on the server side are as follows:

•Create a socket with the socket() function;

•Bind the socket to an address using the bind() function;

•Send and receive data by means of **recvfrom()** and **sendto().**



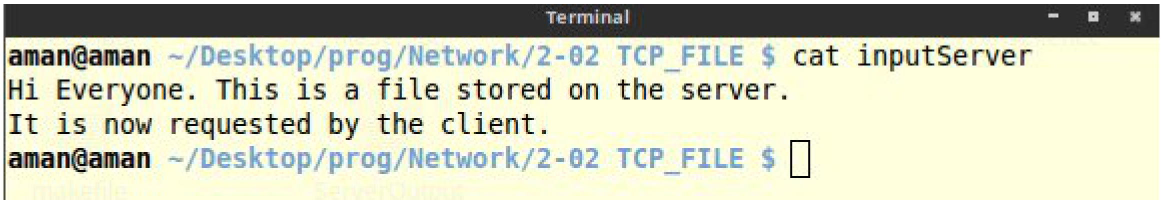
Program (Server Side):

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#include <string.h>
#include <strings.h>
#include <sys/types.h>
#include <stdbool.h>
#include <assert.h>

#define MAX 1024
#define PORT 25000
#define MAX_BACKLOG 10
#define SERVER_INPUT_FILE "inputServer"

// Send a given input file to every client that wishes to connect

int main(){
   // Create a fd for server and one for client
   int sockFd = 0, clientFd = 0;

   // A socket address structure to hold in the socket
   struct sockaddr_in serv_addr = {0};

   // String to hold input file line by line for client
   char sendBuff[MAX];

   // Creates a TCP socket i.e file descriptor within the process table
   sockFd = socket(AF_INET, SOCK_DGRAM, 0);

   // initialise members of the socket's address structure
   serv_addr.sin_family = AF_INET;
   serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
   serv_addr.sin_port = htons(PORT);

   // Assign protocol to a socket :=
    bind(sockFd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

    // listen for incoming connections with a specified backlog queue
    listen(sockFd, MAX_BACKLOG);

    while(true){
        // Socket address structure to hold the client socket
        struct sockaddr_in addr;
        socklen_t addr_size = sizeof(addr);

        fprintf(stdout, "\nA new client has connected. Sending file\n");
        fprintf(stdout, "IP Address: %s\n", inet_ntoa(addr.sin_addr) );

        FILE *fp = fopen(SERVER_INPUT_FILE, "r");
        assert (fp);
        int readChar;

        while (!feof(fp)){
            readChar = fread(sendBuff, 1, MAX, fp);
            recvfrom(clientFd, sendBuff, 0, 0, (struct sockaddr *)&addr, &addr_size);
            sendto(clientFd, sendBuff, readChar, 0, (struct sockaddr *)&addr, sizeof(addr));
        }
        fclose(fp);
        close(sockFd);
        close(clientFd);
        sleep(1);
    }
    return 0;
}
```

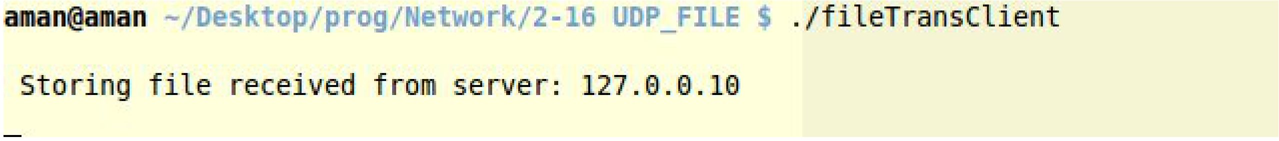
Program (Client Side):

```
#define localhost "127.0.0.10"
#define outputFile "ServerOutput"
#define MAX 1024
#define PORT 25000

// Connect to the server at the given location and create a file mentioning its inputs

int main(){
    // Set up client sockets, character buffers
    int sockFd = 0, n = 0;
    char recvBuff[MAX] = {' '};

    // A socket address structure to hold in the socket
    struct sockaddr_in serv_addr = {0};
    socklen_t serv_addr_size = sizeof(struct sockaddr_in);

    // Create TCP Socket: mention domain, stream/datagram and default protocol
    sockFd = socket(AF_INET, SOCK_DGRAM, 0);

    if(sockFd < 0){
        perror("\n Error : Could not create socket \n");
        return 1;
    }

    // initialise members of the socket's address structure
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    if(inet_pton(AF_INET, localhost, &serv_addr.sin_addr) <= 0){
        printf("\n inet_pton error occured\n");
        return 1;
    }

    if( connect(sockFd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0){
       printf("\n Error : Connect Failed \n");
       return 1;
    }

    // Read from input and write to socket
    fprintf(stdout, "\n Storing file received from server: %s \n\n", inet_ntoa(serv_addr.sin_addr));

    FILE * fp = fopen(outputFile, "w");
    assert(fp);

    // Read message from server
    n = recvfrom(sockFd, recvBuff, MAX, 0, (struct sockaddr *)&serv_addr, &serv_addr_size) ;
    // Write the message received from server onto file
    fprintf(stdout, "I Have read: %d\n", n );
    fwrite(recvBuff, 1, n, fp);

    // Close file
    close(sockFd);


   fclose(fp);
    return 0;
}
```

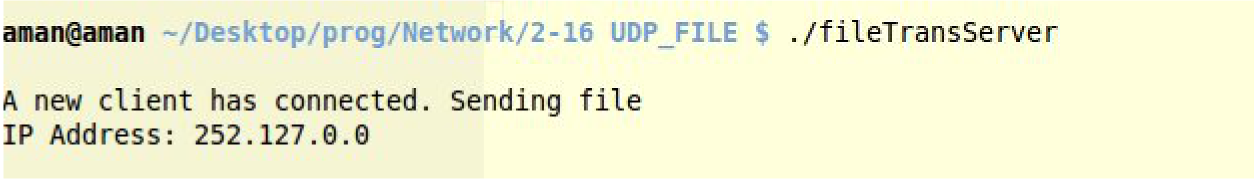
<u>OUTPUT</u>:



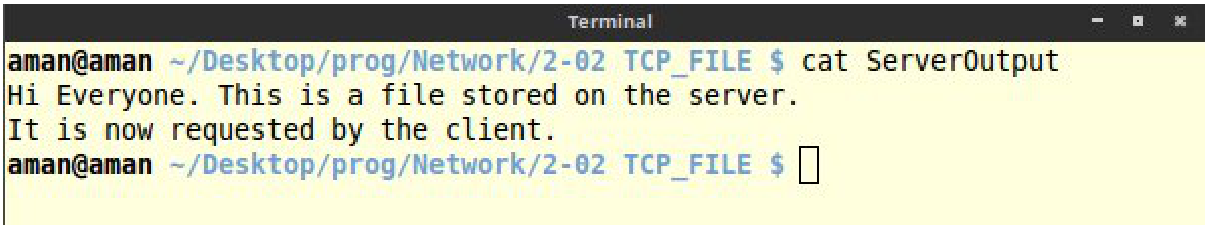
```
aman@aman ~/Desktop/prog/Network/2-02 TCP_FILE $ cat inputServer
Hi Everyone. This is a file stored on the server.
It is now requested by the client.
aman@aman ~/Desktop/prog/Network/2-02 TCP_FILE $
```



```
aman@aman ~/Desktop/prog/Network/2-16 UDP_FILE $ ./fileTransClient

 Storing file received from server: 127.0.0.10
```



```
aman@aman ~/Desktop/prog/Network/2-16 UDP_FILE $ ./fileTransServer

A new client has connected. Sending file
IP Address: 252.127.0.0
```



```
aman@aman ~/Desktop/prog/Network/2-02 TCP_FILE $ cat ServerOutput
Hi Everyone. This is a file stored on the server.
It is now requested by the client.
aman@aman ~/Desktop/prog/Network/2-02 TCP_FILE $
```

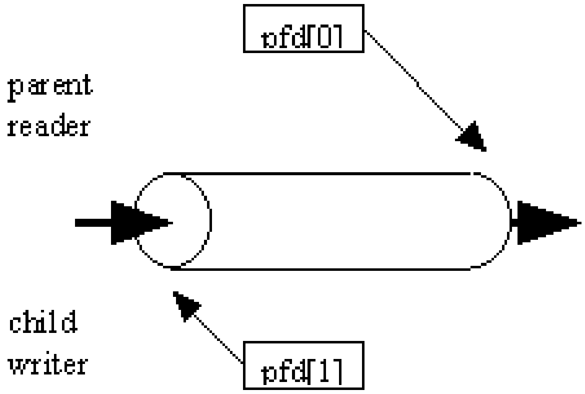
LAB : 6

One way pipe for related inter process communication

Description:

Pipes provide a medium for flow of data. A pipe can be explicitly created in Unix using the pipe system call. Two file descriptors are returned, namely **fd[0]** and **fd[1]**, and they are both open for reading and writing. A read from **fd[0]** accesses the data written to **fd[1]** on a first-in-first-out (FIFO) basis and a read from **fd[1]** accesses the data written to **fd[0]** also on a FIFO basis.

A regular pipe can only connect two related processes. It is created by a process and will vanish when the last process closes it. In this program, a single process is forked to form two related processes which then communicate using a single pipe.

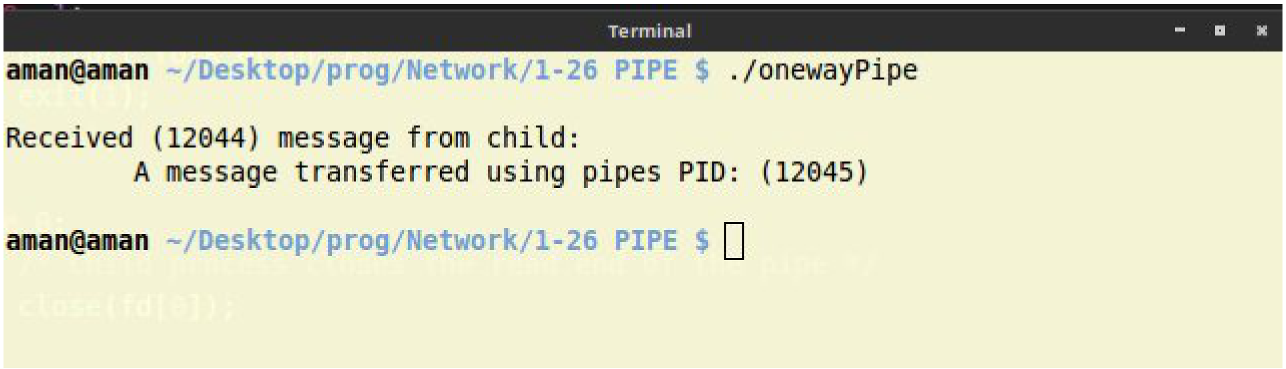


Program :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#define MAX 100

int main(void){
   int fd[2], readChar;
   pid_t childPid;
   char message[] = "A message transferred using pipes";
   char pid[100];
   char readbuffer[MAX] = {' '};

   pipe(fd);
```

```
    switch(childPid = fork()){
        case -1:
            perror("fork error");
            exit(1);
            break;

        case 0:
            /* Child process closes the read end of the pipe */
            close(fd[0]);

            /* Send message through the write end of the pipe */
            sprintf(pid, " PID: (%d) ", getpid());
            write(fd[1], strcat(message, pid), (strlen(message) + 1));
            exit(0);
            break;

        default:
            /* Parent process closes the write end of the pipe */
            close(fd[1]);

            /* Read from the pipe */
            readChar = read(fd[0], readbuffer, sizeof(readbuffer));
            printf("\nReceived (%d) message from child: \n\t%s\n\n", getpid(),  readbuffer);

    }

    return 0;
}
```

OUTPUT:



```
aman@aman ~/Desktop/prog/Network/1-26 PIPE $ ./onewayPipe

Received (12044) message from child:
        A message transferred using pipes PID: (12045)

aman@aman ~/Desktop/prog/Network/1-26 PIPE $ 
```

LAB : 7

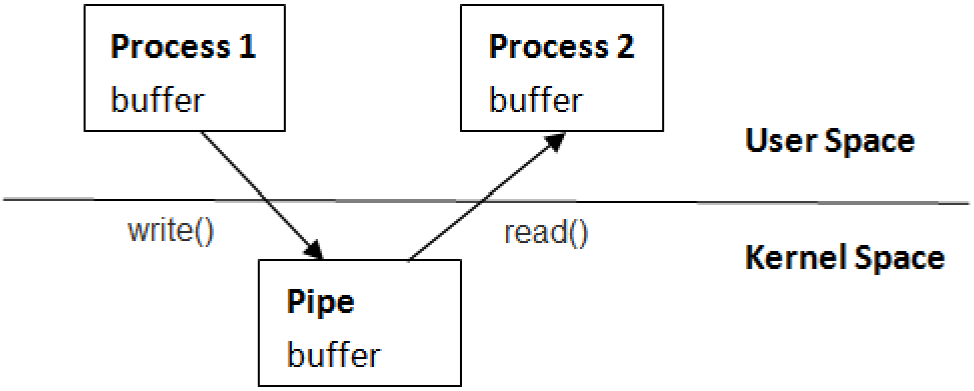
Creation of a FIFO for unrelated inter process communication

Description:

A named pipe works much like a regular pipe, but does have some noticeable differences.

- •Named pipes exist as a device special file in the file system.
- •Processes of different ancestry can share data through a named pipe.
- •When all I/O is done by sharing processes, the named pipe remains in the file system for later use.

I/O operations on a FIFO are essentially the same as for normal pipes, with once major exception. An ``**open**" system call or library function should be used to physically open up a channel to the pipe. With half-duplex pipes, this is unnecessary, since the pipe resides in the kernel and not on a physical file system.



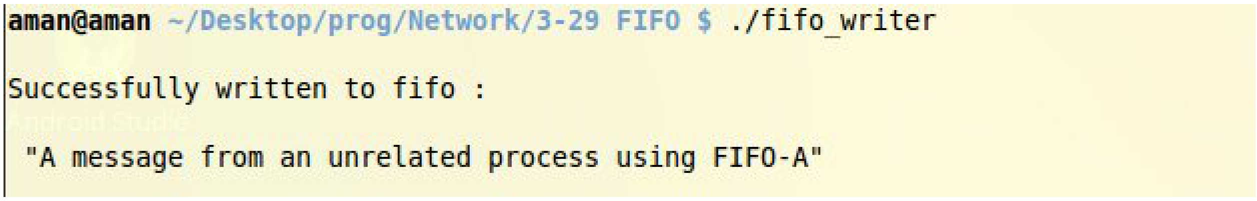
Program (Writer):

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

#define FIFO_NAME  "A_fifo_files"
#define FIFO_MODE 0666
#define MAX 100

// Two demonstrate a named pipe and message transfer between two unrelated processes
// This program creates a process A that writes to a pipe for another process C

int main(){
    //Prepare the message to send
    char MSG[MAX] = "A message from an unrelated process using FIFO-A";

    //create a named pipe
    const char * fifo = FIFO_NAME;
    mkfifo(fifo, FIFO_MODE);

    //Write to the special fifo file
    int fd = open(fifo, O_WRONLY);
    if (write(fd, MSG, sizeof(MSG)) < 0){
        perror("Writing to named pipe error");
        return -1;
    }
    printf("\nSuccessfully written to fifo : \n\n \"%s\"\n\n", MSG);

    close(fd);
    unlink(fifo);
    return 0;
}
```

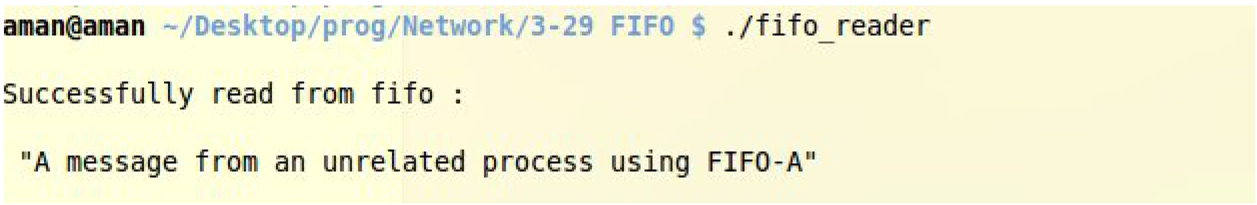
Program (Reader):

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

#define FIFO_NAME "A_fifo_files"
#define FIFO_MODE 0666
#define MAX 100

// Two demonstrate a named pipe and message transfer between two unrelated processes
// This program creates a process B that reads from a named pipe written by another process A

int main(){
    //Prepare the message to send
    char RCV[MAX] = {0};

    //open a named named pipe
    const char * fifo = FIFO_NAME;

    //Read from the special fifo file
    int fd = open(fifo, O_RDONLY);
    while (1) {
        if (read(fd, RCV, MAX) > 0){
            printf("\nSuccessfully read from fifo : \n\n \"%s\"\n\n", RCV);
            break;
        }
        else {
            perror("Read error from fifo");
            return -1;
        }
    }

    close(fd);

    return 0;
}
```

OUTPUT:



```
aman@aman ~/Desktop/prog/Network/3-29 FIFO $ ./fifo_writer

Successfully written to fifo :

 "A message from an unrelated process using FIFO-A"
```



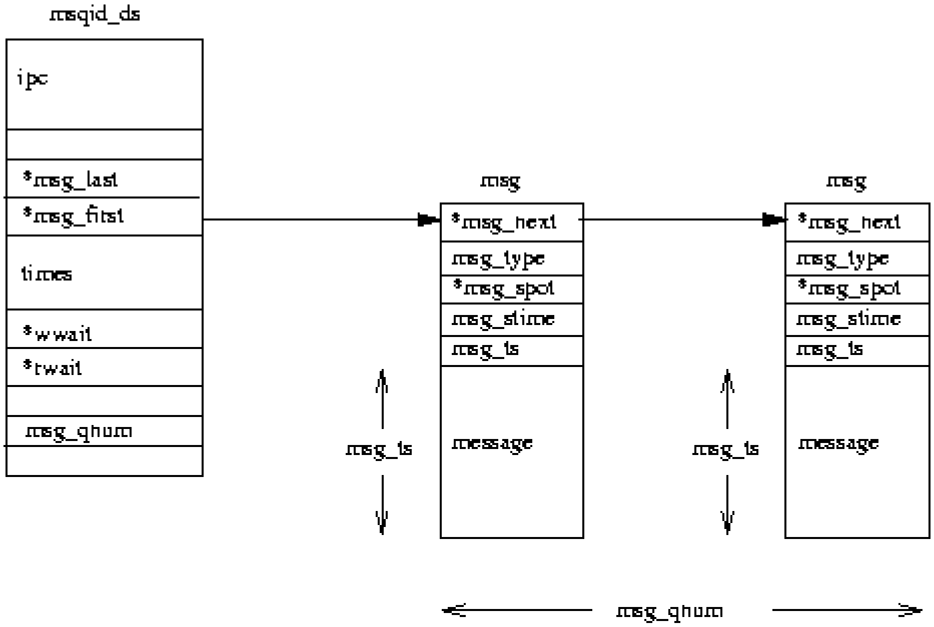
```
aman@aman ~/Desktop/prog/Network/3-29 FIFO $ ./fifo_reader

Successfully read from fifo :

 "A message from an unrelated process using FIFO-A"
```

LAB : 8

Program to implement IPC Message Queues (C)

Description:

Message queues allow one or more processes to write messages that will be read by one or more reading processes. Linux maintains a list of message queues, the **msgque** vector: each element of which points to a **msqid_ds** data structure that fully describes the message queue. When message queues are created, a new **msqid_ds** data structure is allocated from system memory and inserted into the vector.



Each **msqid_ds** data structure contains an **ipc_perm** data structure and pointers to the messages entered onto this queue. In addition, Linux keeps queue modification times such as the last time that this queue was written to and so on. The **msqid_ds** also contains two wait queues: one for the writers to the queue and one for the readers of the queue.

Each time a process attempts to write a message to the write queue, its effective user and group identifiers are compared with the mode in this queue's ipc_perm data structure. If the process can write to the queue then the message may be copied from the process' address space into a msg data structure and put at the end of this message queue. Each message is tagged with an application specific type, agreed between the cooperating processes. However, there may be no room for the message as Linux restricts the number and length of messages that can be written. In this case the process will be added to this message queue's write wait queue and the scheduler will be called to select a new process to run. It will be awakened when one or more messages have been read from this message queue.

Reading from the queue is similar. Again, the process' access rights to the write queue are checked. A reading process may choose to either get the first message in the queue regardless of its type or select messages with particular types. If no messages match this criteria the reading process will be added to the message queue's read wait queue and the scheduler run. When a new message is written to the queue this process will be awakened and run again.

Program (Sender);

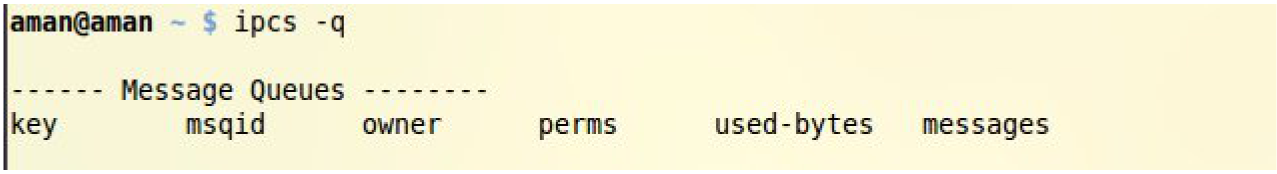
```
#include <stdio.h>
#include <string.h>
#include <sys/msg.h>

#define MSG_QUE_KEY 31
#define MAX_MSG_LEN 100
#define MSG_FLAG  IPC_CREAT | 0666
// A program to demo the working of message queues in IPC (sender)

typedef struct msgbuf{
   long mtype;
   char mtext[MAX_MSG_LEN];
}newMessage;

int main(){
   newMessage sendMsg;
   int msgQue;

   printf("\nEnter a message to send: ");
   scanf("%[^\n]",sendMsg.mtext);

   // Create a new message queue exclusively for the process
   if ((msgQue = msgget(MSG_QUE_KEY, MSG_FLAG)) < 0){
      perror("Message Queue Error");
      return -1;
   }

   printf("\nIn Parent Sending : %s => %ld\n", sendMsg.mtext, strlen(sendMsg.mtext) +1 );

   if (msgsnd(msgQue, &sendMsg, strlen(sendMsg.mtext) +1, IPC_NOWAIT) < 0)
      perror("\nParent message queue error");

   else
      printf("\nParent Message Sent\n\n");

   return 0;
}
```

Program (Receiver);

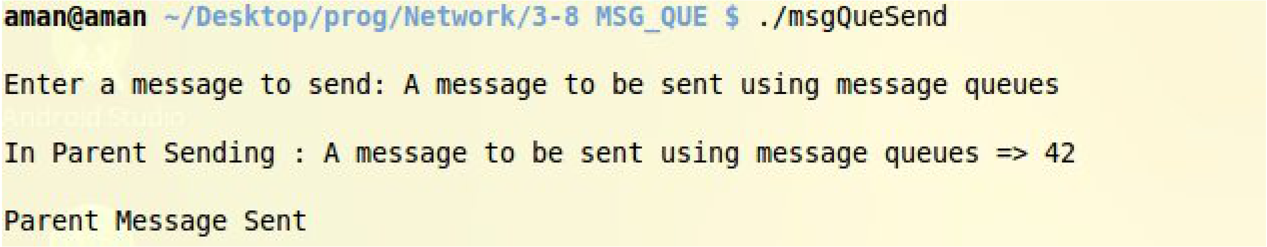
```
#include <stdio.h>
#include <sys/msg.h>

#define MSG_QUE_KEY 31
#define MAX_MSG_LEN 100
#define MSG_FLAG  IPC_CREAT | 0666
// A program to demo the working of message queues in IPC (receiver)

typedef struct msgbuf{
   long mtype;
   char mtext[MAX_MSG_LEN];
}newMessage;

int main(){
   newMessage recvMsg;
   struct msqid_ds wholeQueue;
   int msgQue;

   if ((msgQue = msgget(MSG_QUE_KEY, MSG_FLAG))< 0)
      perror("\nMessage Queue doesn't exist\n");

   else{//queue exists
      if (msgrcv(msgQue, &recvMsg, MAX_MSG_LEN, 0, IPC_NOWAIT) < 0)
      perror("\nMessage Reading failed\n\n");

      else {
         printf("\nMessage Received: %s\n", recvMsg.mtext );
         if (msgctl(msgQue, IPC_STAT, &wholeQueue) == 0){
            printf("\nNumber of messages in Queue: %ld", wholeQueue.msg_qnum);
            printf("\nLast PID that wrote: %d", wholeQueue.msg_lspid);
            printf("\nTime of last write: %ld", wholeQueue.msg_stime);
            printf("\nLast PID that read: %d", wholeQueue.msg_lrpid);
            printf("\nTime of last read: %ld\n\n", wholeQueue.msg_rtime);
            msgctl(msgQue, IPC_RMID, NULL); //Remove message
         }
      }
   }
   return 0;
```

}

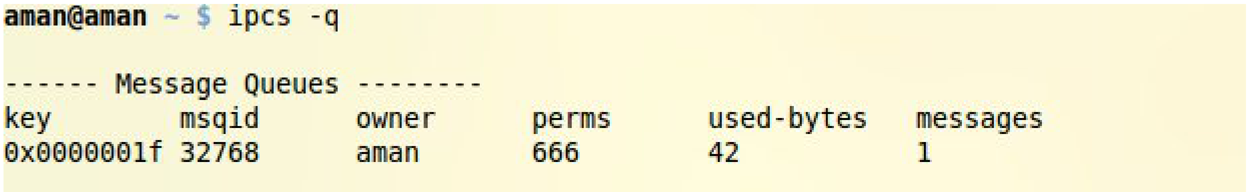
OUTPUT:



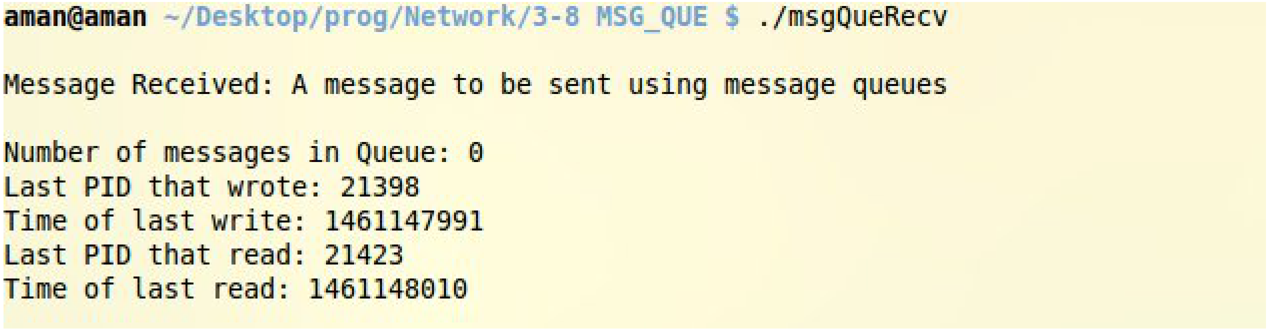
```
aman@aman ~ $ ipcs -q

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



```
aman@aman ~/Desktop/prog/Network/3-8 MSG_QUE $ ./msgQueSend

Enter a message to send: A message to be sent using message queues

In Parent Sending : A message to be sent using message queues => 42

Parent Message Sent
```



```
aman@aman ~ $ ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x0000001f 32768      aman       666        42           1
```



```
aman@aman ~/Desktop/prog/Network/3-8 MSG_QUE $ ./msgQueRecv

Message Received: A message to be sent using message queues

Number of messages in Queue: 0
Last PID that wrote: 21398
Time of last write: 1461147991
Last PID that read: 21423
Time of last read: 1461148010
```

LAB : 9

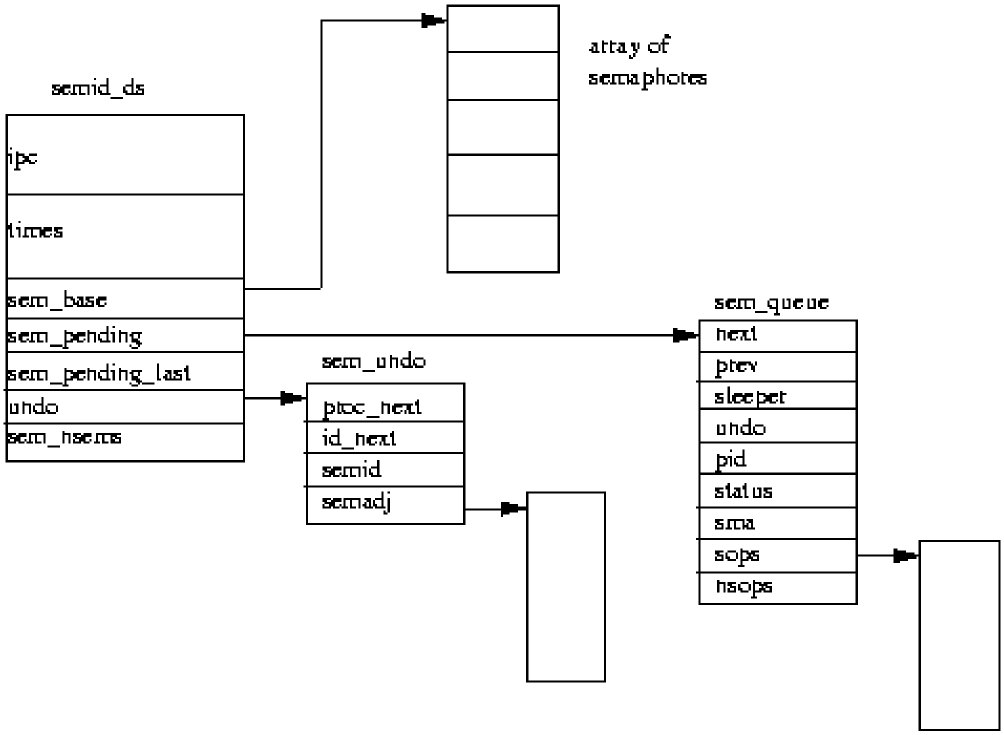
Program to implement semaphore operations

Description:

Semaphores are a programming construct designed by E. W. Dijkstra in the late 1960s. A semaphore appears to be a simple integer. A process (or a thread) waits for permission to proceed by waiting for the integer to become 0. The signal if it proceeds signals that this by performing incrementing the integer by 1. When it is finished, the process changes the semaphore's value by subtracting one from it. Semaphores let processes query or alter status information. They are often used to monitor and control the availability of system resources such as shared memory segments.

Semaphores can be operated on as individual units or as elements in a set. A semaphore set consists of a control structure and an array of individual semaphores. A set of semaphores can contain up to 25 elements.

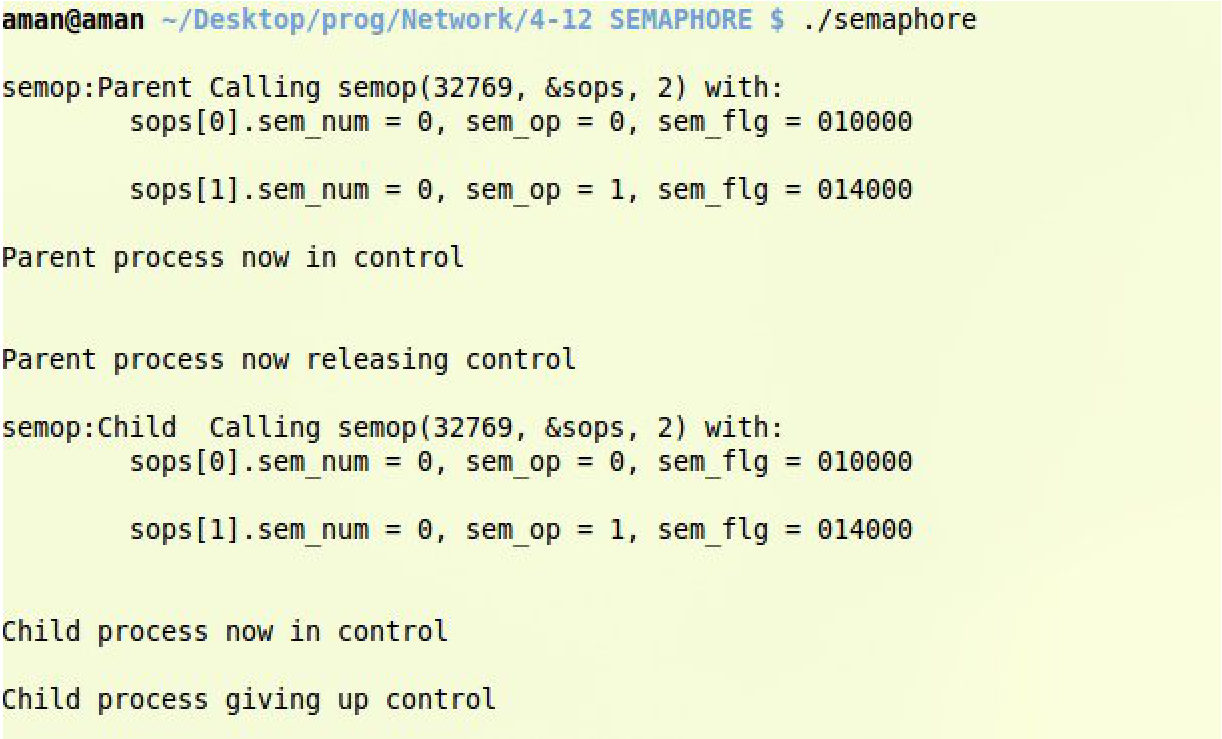
In a similar fashion to message queues, the semaphore set must be initialized using **semget()**; the semaphore creator can change its ownership or permissions using **semctl()**; and semaphore operations are performed via the **semop()** function. These are now discussed below:



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

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

#define SEMFLAG IPC_CREAT | 0666
#define NUMSEMS 1

void error(char *msg){
  perror(msg);
  exit(1);
}

int main(){
    int j;
    pid_t pid;
    int semid; /* semid of semaphore set */
    key_t key ; /* key to pass to semget() */
    int nsops; /* number of operations to do */

    struct sembuf *sops = (struct sembuf *) malloc(2 * sizeof(struct sembuf));
    /* ptr to operations to perform */

    //generate key
    if ((key = ftok("semaphore.c", 'Q')) == -1)
       error("ftok");

    /* set up semaphore */
    if ((semid = semget(key, NUMSEMS, SEMFLAG)) == -1)
        error("semget: semget failed");

    if ((pid = fork()) < 0)
        error("fork");

    if (pid == 0){ //child
        nsops = 2;

        /* wait for semaphore to reach zero */
        sops[0].sem_num = 0;
        sops[0].sem_op = 0; /* wait for semaphore flag to become zero */
        sops[0].sem_flg = SEM_UNDO; /* take off semaphore asynchronous  */

        sops[1].sem_num = 0;
        sops[1].sem_op = 1; /* increment semaphore*/
        sops[1].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore */

        printf("\nsemop:Child  Calling semop(%d, &sops, %d) with:", semid, nsops);
        for (j = 0; j < nsops; j++){
             printf("\n\tsops[%d].sem_num = %d, ", j, sops[j].sem_num);
             printf("sem_op = %d, ", sops[j].sem_op);
             printf("sem_flg = %#o\n", sops[j].sem_flg);
        }

        /* Make the semop() call and report the results. */
        if ((j = semop(semid, sops, nsops)) == -1){
            perror("semop: semop failed");
        }
        else{
            printf("\n\nChild process now in control\n");
            nsops = 1;

            /* wait for semaphore to reach zero */
            sops[0].sem_num = 0;
            sops[0].sem_op = -1; /* Give UP COntrol of track */
            sops[0].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore, asynchronous  */

            if ((j = semop(semid, sops, nsops)) == -1)
                perror("semop: semop failed");
            else
                printf("\nChild process giving up control\n\n");
            sleep(1); /* halt process to allow parent to catch semaphore change first */
        }

    }
    else{ /* parent */
        nsops = 2;
        /* wait for semaphore to reach zero */
        sops[0].sem_num = 0;
        sops[0].sem_op = 0; /* wait for semaphore flag to become zero */
        sops[0].sem_flg = SEM_UNDO; /* take off semaphore asynchronous  */

        sops[1].sem_num = 0;
        sops[1].sem_op = 1; /* increment semaphore */
        sops[1].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore */

        printf("\nsemop:Parent Calling semop(%d, &sops, %d) with:", semid, nsops);
        for (j = 0; j < nsops; j++){
            printf("\n\tsops[%d].sem_num = %d, ", j, sops[j].sem_num);
            printf("sem_op = %d, ", sops[j].sem_op);
            printf("sem_flg = %#o\n", sops[j].sem_flg);
        }
        /* Make the semop() call and report the results. */
        if ((j = semop(semid, sops, nsops)) == -1)
            perror("semop: semop failed");
        else{
            printf("\nParent process now in control\n");
            nsops = 1;

            /* wait for semaphore to reach zero */
            sops[0].sem_num = 0;
            sops[0].sem_op = -1; /* Give UP Control of track */
            sops[0].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore, asynchronous  */

            if ((j = semop(semid, sops, nsops)) == -1)
                perror("semop: semop failed");
            else
                printf("\nParent process now releasing control\n\n");
            sleep(1); /* halt process to allow child to catch semaphore change first */
        }
    }
    return 0;
}
```

OUTPUT:



```
aman@aman ~/Desktop/prog/Network/4-12 SEMAPHORE $ ./semaphore

semop:Parent Calling semop(32769, &sops, 2) with:
        sops[0].sem_num = 0, sem_op = 0, sem_flg = 010000

        sops[1].sem_num = 0, sem_op = 1, sem_flg = 014000

Parent process now in control


Parent process now releasing control

semop:Child  Calling semop(32769, &sops, 2) with:
        sops[0].sem_num = 0, sem_op = 0, sem_flg = 010000

        sops[1].sem_num = 0, sem_op = 1, sem_flg = 014000


Child process now in control

Child process giving up control
```

LAB : 10

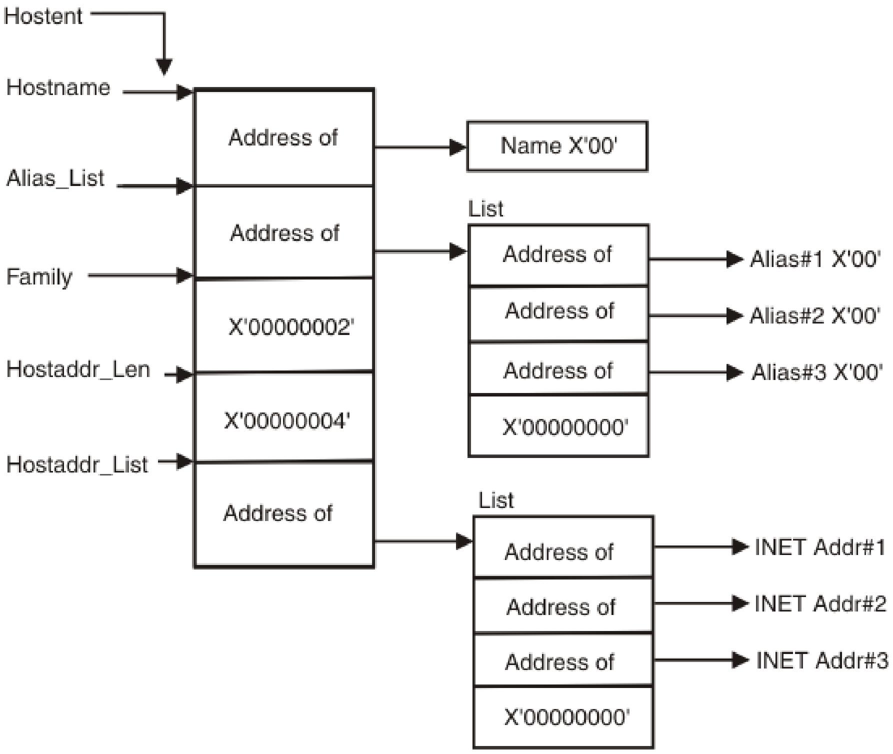
Program to resolve host name using DNS Server

Description:

The Domain Name System (DNS) is a hierarchical decentralized naming system for computers, services, or any resource connected to the Internet or a private network. It associates various information with domain names assigned to each of the participating entities. Most prominently, it translates more readily memorized domain names to the numerical IP addresses needed for the purpose of locating and identifying computer services and devices with the underlying network protocols. By providing a worldwide, distributed directory service, the Domain Name System is an essential component of the functionality of the Internet.

In this program, we receive a URL from the user. If the URL is resolved by the DNS server, the corresponding IP is returned. Otherwise an error is shown.

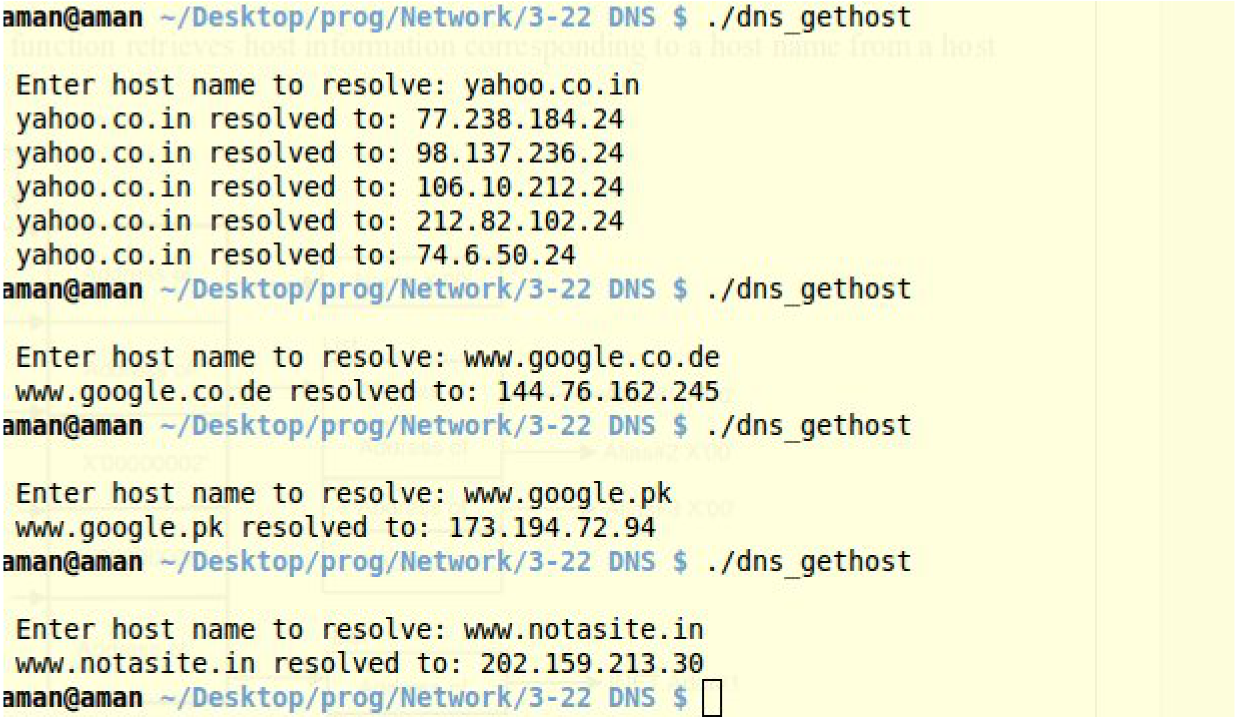
The **gethostbyname()** function retrieves host information corresponding to a host name from a host database.



Program:

```
#include <stdio.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <netinet/in.h>

void resolveName(char *host_name){
    // Given a url in human readable format, convert it into byte order IP
    struct hostent *host;
    struct in_addr **h_addr;

    if ((host = gethostbyname(host_name)) == NULL){
        perror(" Couldn't resolve hostname");
        return;
    }
    h_addr = (struct in_addr **) host -> h_addr_list;
    for (int i = 0; h_addr[i] != NULL; i++)
        printf(" %s resolved to: %s\n\n", host_name, inet_ntoa(*h_addr[i]));
}

int main(){
    char host_name[MAX];
    printf("\n Enter host name to resolve: ");
    scanf("%s", host_name);

    resolveName(host_name);
    return 0;
}
```

<u>OUTPUT:</u>



```
aman@aman ~/Desktop/prog/Network/3-22 DNS $ ./dns_gethost

 Enter host name to resolve: yahoo.co.in
 yahoo.co.in resolved to: 77.238.184.24
 yahoo.co.in resolved to: 98.137.236.24
 yahoo.co.in resolved to: 106.10.212.24
 yahoo.co.in resolved to: 212.82.102.24
 yahoo.co.in resolved to: 74.6.50.24
aman@aman ~/Desktop/prog/Network/3-22 DNS $ ./dns_gethost

 Enter host name to resolve: www.google.co.de
 www.google.co.de resolved to: 144.76.162.245
aman@aman ~/Desktop/prog/Network/3-22 DNS $ ./dns_gethost

 Enter host name to resolve: www.google.pk
 www.google.pk resolved to: 173.194.72.94
aman@aman ~/Desktop/prog/Network/3-22 DNS $ ./dns_gethost

 Enter host name to resolve: www.notasite.in
 www.notasite.in resolved to: 202.159.213.30
aman@aman ~/Desktop/prog/Network/3-22 DNS $
```