TABLE OF CONTENTS

A.Environment Used

GNU/Linux environment has been used here throughout for programming. GNU is a computer operating system composed entirely of free software. Its name is a recursive acronym for GNU's Not Unix; it was chosen because its design is Unix-like, but differs from Unix by being free software and containing no Unix code. A GNU System's basic components include the GNU Compiler Collection (GCC), the GNU Binary Utilities (binutils), the bash shell, the GNU C Library (glibc) and GNU Core Utilities (coreutils). GNU is in active development. Although nearly all components have been completed long ago and have been in production use for a decade or more, its official kernel, GNU Hurd, is incomplete and not all GNU components work with it. Thus, the third-party Linux kernel is most commonly used instead. The Linux Kernel was first released to the public on 17 September, 1991.

Linux distributions have mainly been used as server operating systems, and have risen to prominence in that area. Linux distributions are the cornerstone of the LAMP server-software combination (Linux, Apache, MySQL, Perl/PHP/Python) which has achieved popularity among developers, and which is one of the more common platforms for website hosting. There are many Linux Distributions available these days which consist of the Linux kernel and, usually, a set of libraries and utilities from the GNU project, with graphics support from the X Window System. One can distinguish between commercially backed distributions, such as Fedora (Red Hat), openSUSE (Novell), Ubuntu (Canonical Ltd.), and Mandriva Linux and community distributions such as Debian and Gentoo, though there are other distributions that are driven neither by a corporation nor a community; perhaps most famously, Slackware.

The user/programmer can control a Linux-based system through a command line interface (or CLI), a graphical user interface (or GUI), or through controls attached to the associated hardware (this is common for embedded systems). For desktop systems, the default mode is usually graphical user interface (or GUI). Most Linux distributions support dozens of programming languages. The most common collection of utilities for building both Linux applications and operating system programs is found within the GNU toolchain, which includes the GNU Compiler Collection (GCC) and the GNU build system. Amongst others, GCC provides compilers for Ada, C, C++, Java, and Fortran. The Linux kernel itself is written to be compiled with GCC.

Editors in Linux

Many editors are available inbuilt in Linux itself for the purpose of editing. One of those may be used for programming too. Some of the common editors are gedit, vi, vim, nano, kedit etc. All of them provide different options for editing. Here, we use vi editor for coding purpose. The commands used in vi are:

Creating / Opening an already existing file:

vi <filename>

Exiting vi:

After pressing Esc key,

:q<return> - quit vi

:wq<return> - save the current file and quit vi

There are mainly five modes in vi.

- Insert mode
- Command mode
- Replace mode
- Execute mode

- Visual mode

Command mode

To go to command mode, press Ctrl+c at any time. In command mode, some of the common operations and corresponding commands are:

- To move to the end of file : press G
- To move to beginning of file : press gg
- To move to a particular line : enter line number and then press G
- To copy a line of text : press yy at the start of the line
- To copy a group of lines : enter number of lines to be copied and then press yy

  Eg: 5yy
- To copy a word : press yw at the start of the word
- To cut a line : press cc at the start of the line
- To cut a group of lines : enter number of lines to be copied and then press cc

  Eg: 5cc
- To cut a word : press cw at the start of the word
- For paste operation : press p
- To delete a line: press dd at the start of the line
- To delete a group of lines : enter number of lines to be copied and then press dd

  Eg: 5dd
- To delete a word : press dw at the start of the word

- To search for a particular word : press / and then enter the word and press Enter key
- To go to next word – press n
- To go backwards – press N
- To undo last action press u
- To redo last action press Ctrl+r

Insert Mode

Normally when we open a file in vi, it will be in command mode. Now to enter text into the file, we need to be in Insert Mode. To switch from command to insert mode simply press Insert key or 'i'. This mode is used to manually edit the file.

Compiling Using GCC

While compiling using gcc, there are several options available. Generally, compiling a file is done by entering a command at the terminal in the following syntax:

```
gcc –o <output file> <input file>
```

To execute the compiled file: ./<output file>

What the option –o<output file> does: Place output in file <output file>. This applies regardless to whatever sort of output is being produced, whether it be an executable file, an object file, an assembler file or preprocessed C code. Since only one output file can be specified, it does not make sense to use -o when compiling more than one input file, unless you are producing an executable file as output. If -o is not specified, the default is to put an executable file in a.out. In this case the execution is done by specifying the command: . /a.out

Another method of compiling is done by: gcc -c <input file>

*What the option –c does: C*ompile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file.

To compile multiple files in gcc: Write the commands for compiling the individual files in a shell script with .sh extension and run this script file. All the files will get compiled.

For eg: to compile the files server.c, client.c, finger_clnt.c, finger_svc.c and finger_xdr.c specify the following commands inside a shell script file say, compile.sh.

```
gcc -c server.c -o ser

gcc -c client.c -o cl

gcc -c finger_clnt.c -o f_c

gcc -c finger_svc.c -o f_s

gcc -c finger_xdr.c -o f_x

gcc -o server ser f_s f_x

gcc -o client cl f_c f_x
```

To execute the compile.sh file, give the following command in command line

./compile.sh (or) sh compile.sh

B. Threads

THREADS are mechanism to allow a program to do more than one thing at a time. As with processes, threads appear to run concurrently; the Linux kernel schedules them asynchronously, interrupting each thread from time to time to give others a chance to execute. Conceptually, a thread exists within a process. Threads are a finer-grained unit of execution than processes. When you invoke a program, Linux creates a new process and in that process creates a single thread, which runs the program sequentially. That thread can create additional threads; all these threads run the same program in the same process, but each thread may be executing a different part of the program at any given time. All threads within a process share the same address space. The creating and the created thread share the same memory space, file descriptors, and other system resources as the original. If one thread changes the value of a variable, for instance, the other thread subsequently will see the modified value. Similarly, if one thread closes a file descriptor, other threads may not read from or write to that file descriptor. Because a process and all its threads can be executing only one program at a time, if any thread inside a process calls one of the exec functions, all the other threads are ended (the new program may, of course, create new threads). GNU/Linux implements the POSIX standard thread API (known as pthreads). All thread functions and data types are declared in the header file <pthread.h>.

The POSIX standard defines an API for creating and manipulating threads. Pthreads defines a set of programming C language types, functions and constants. It is implemented with a header and a thread library. Programmers can use Pthreads to create, manipulate and manage threads, as well as between threads using mutex. Mutex variables are one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur. A mutex variable acts like a "lock" protecting access to a shared data resource. The basic concept of a mutex as used in Pthreads is that only one thread can lock (or own) a mutex variable at any given time. Thus, even if several threads try to lock a mutex only one thread will be successful. No other thread can own that mutex until the owning thread unlocks that mutex. Threads must "take turns" accessing protected data.

Thread Creation

Each thread in a process is identified by a thread ID. When referring to thread IDs in C or C++ programs, use the type pthread_t. Upon creation, each thread executes a thread function. This is just an ordinary function and contains the code that the thread should run. When the function returns, the thread exits. On GNU/Linux, thread functions take a single parameter, of type void*, and have a void* return type. The program can use this parameter to pass data to a new thread. Similarly, the program can use the return value to pass data from an exiting thread back to its creator.

The pthread_create function creates a new thread. It is provided with the following:

1. A pointer to a pthread_t variable, in which the thread ID of the new thread is stored.

2. A pointer to a thread attribute object. This object controls details of how the thread interacts with the rest of the program. If you pass NULL as the thread attribute, a thread will be created with the default thread attributes.

3. A pointer to the thread function. This is an ordinary function pointer, of this type:

   void* (*) (void*)

4. A thread argument value of type void*. Whatever you pass is simply passed as the argument to the thread function when the thread begins executing.

Syntax: pthread_create (<ptr to pthread_t>, NULL, (void*) func, (void*) <thread arg>);

A call to pthread_create returns immediately, and the original thread continues executing the instructions following the call. Meanwhile, the new thread begins executing the thread function. Linux schedules both threads asynchronously, and the program must not rely on the relative order in which instructions are executed in the two threads.

A thread exits in one of two ways. One way, as illustrated previously, is by returning from the thread function. The return value from the thread function is taken to be the return value of the thread. Alternately, a thread can exit explicitly by calling pthread_exit. This function may be called from within the thread function or from some other function called directly or indirectly by the thread function. The argument to pthread_exit is the thread's return value.

Thread Joining

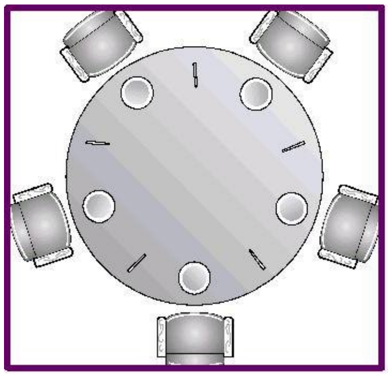
For successfully execution of all the created threads, it should be ensured that main thread never finishes executing first before the threads created by it. One solution is to force main to wait until the other threads are done. For this we need a function that waits for a thread. That function is pthread_join, which takes two arguments: the thread ID of the thread to wait for, and a pointer to a void* variable that will receive the finished thread's return value.

Syntax: pthread_join (<ptr to pthread_t>, (void*) <var>);

1. Dining Philosopher's Problem

Problem Description

Develop a program to implement the solution of the dining philosopher's problem using threads. The input to the program is the number of philosophers to be seated around the table. Output shows the various stages that each philosopher passes through within a certain time. A philosopher can be in anyone of the three stages at a time: thinking, eating or finished eating.



Data Structures and Functions

The main data structures used here are: Arrays

The arrays represent the philosophers and corresponding chopsticks for them. Each element in the philosopher's array corresponds to a thread and each element in the chopstick's array corresponds to a mutex variable.

The functions used here are:

- pthread_mutex_init (&mutex, NULL) – initialization of mutex variable
- pthread_mutex_lock (&mutex) – attempt to lock a mutex
- pthread_mutex_unlock (&mutex) – unlock a mutex
- pthread_create (ptr to thread, NULL, (void*) func, (void*) <thread arg>)
- pthread_join (ptr to thread, &msg)
- pthread_mutex_destroy (ptr to thread)
- pthread_exit(NULL)

Note: while compiling this program use the following:

```
[root@Linux philo]# gcc –o dining dining.c -lpthread
```

Algorithm

Algorithm for process:

1. Start.
2. Declare and initialize the thread variables (philosophers) as required.
3. Declare and initialize the mutex variables (chopsticks) as required.
4. Create the threads representing philosophers.
5. Wait until the threads finish execution.
6. Stop.

Algorithm for thread (philosopher) function:

1. Start.
2. Wait for left fork spoon.
3. Wait for right fork spoon.
4. Start eating.
5. Release the left fork spoon.
6. Release the right fork spoon.
7. Finish eating.
8. Stop.

Code

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

pthread_t phil[5];
pthread_mutex_t chopstick[5];
void* func(int i){
      printf("\nPhilisopher %d is thinking",i);
      pthread_mutex_lock(&chopstick[i]);
      pthread_mutex_lock(&chopstick[(i+1)%5]);
      printf("\nPhilosopher %d is eating..",i);
      sleep(2);
      pthread_mutex_unlock(&chopstick[i]);
      pthread_mutex_unlock(&chopstick[(i+1)%5]);
      printf("\nPhilosopher %d has finished eating..",i);
      pthread_exit(NULL);}
int main(){
      int i,k;
      void *msg;
      for(i=0;i<5;i++){
            k=pthread_mutex_init(&chopstick[i],NULL);
            if(k==-1){
                  printf("\nError in initializing..");
                  exit(1);}}
      for(i=0;i<5;i++){
            k=pthread_create(&phil[i],NULL,(void*)func,(int*)i);
            if(k==-1){
                  printf("\nError in creating..");
                  exit(1);}}
      for(i=0;i<5;i++){
            k=pthread_join(phil[i],&msg);
            if(k==-1){
                  printf("\nError in joining..");
                  exit(1);}}
      for(i=0;i<5;i++){
            k=pthread_mutex_destroy(&chopstick[i]);
            if(k==-1){
                  printf("\nError in destroying..");
                  exit(1);}}
      return 0;}
```

Sample Input and Output

```
philosopher 1 taken his chopsticks and started eating...
philosopher 3 taken his chopsticks and started eating...
philosopher 1 finished eating...
philosopher 3 finished eating...
philosopher 2 taken his chopsticks and started eating...
philosopher 5 taken his chopsticks and started eating...
philosopher 2 finished eating...
philosopher 5 finished eating...
philosopher 4 taken his chopsticks and started eating...
philosopher 4 finished eating...
```

2. Banker’s Algorithm

Problem Description

Develop a program to implement the banker’s algorithm used for resource allocation for a process in safe state. Input is the allocated matrix, maximum matrix, available matrix. Output is whether the system is in safe state or unsafe state. If it is in safe state, resource request is granted.

Data Structures and Functions

The main data structures used here are: Two dimensional Arrays

These arrays represent the allocated matrix, maximum matrix, available matrix and need matrix.

Algorithm

Procedure Safety

1. Let work and finish be vectors of length m & n. Initialize work = Available & Finish[i] = false for I = 0, 1,….., (n-1).
2. Find an i such that
    a. Finish [i] = false
    b. Need [i] <= work

    If no such i exists, go to step 4.
3. Work = work + allocation;

    Finish [i] = true

    Go to step 2.
4. If finish [i] = true for all I, the system is in safe state else unsafe state.

Procedure Resource Request

1. If Request [i] <= Need [i] go to step 2 otherwise raise an error since the process exceeded its maximum claim.
2. If Request [i] <=Available go to step 3, otherwise P [i] must wait since the resources are not available.
3. Available = Available – Request [i]

    Need [i] = Need [i] – Request [i]

    Allocation = Allocation + Request

Code

```
#include<stdio.h>
#include<stdlib.h>
int alloc[10][10],max[10][10],avail[3],need[10][10],flag=0,flag1=0,flag2=0;
int j,k;
int Processing(int i)
{
	flag=flag1=0;
	for(k=0;k<5;k++)
	{
		for(j=0;j<3;j++)
			if(need[k][j]!=0)
			{
				flag=1;
				break;
			}
		if(flag==1)
			break;
	}
	if(flag==0)
	{
		printf("\n\nAll resources safely allocated...");
		return 1;
	}
	if(flag2==5)
	{
		printf("\n\nUnsafe Allocation..");
		return 1;
	}
	for(j=0;j<3;j++)
		if(need[i][j]>avail[j])
		{
			flag1=1;
			break;
		}
	if(need[i][0]==0&&need[i][1]==0&&need[i][2]==0)
		flag1=1;
	if(flag1==0)
	{
		printf("\nAllocating Process P%d the needed resource..",i);
		sleep(2);
		avail[0]+=alloc[i][0];
		avail[1]+=alloc[i][1];
		avail[2]+=alloc[i][2];
		for(j=0;j<3;j++)
			need[i][j]=0;
		printf("\nProcess P%d finished execution",i);
		flag2=0;
	}
	if(i==4)
		i=-1;
	if(flag1!=0)
```

```
            flag2++;
      Processing(++i);
}
void find_need()
{
      int i;
      for(i=0;i<5;i++)
            for(j=0;j<3;j++)
                  need[i][j]=max[i][j]-alloc[i][j];
      printf("\n\nThe Need matrix is : ");
      for(i=0;i<5;i++)
      {
            printf("\nP%d : ",i);
            for(j=0;j<3;j++)
                  printf("%d ",need[i][j]);
      }
}
int main()
{
      int i;
      printf("\nInputting the allocated matrix...");
      for(i=0;i<5;i++)
            for(j=0;j<3;j++)
            {
            printf("\nEnter the Resource %c value allocated to process P%d : 
",(65+j),i);
            scanf("%d",&alloc[i][j]);
            }
      printf("\nInputting the maximum resource required matrix...");
      for(i=0;i<5;i++)
            for(j=0;j<3;j++)
            {
            printf("\nEnter the maximum value of Resource %c required to run 
process P%d : ",(65+j),i);
            scanf("%d",&max[i][j]);
            }
      for(i=0;i<3;i++)
      {
            printf("\nEnter the currently available value of resource %c : 
",(65+i));
            scanf("%d",&avail[i]);
      }
      printf("\n\nThe Allocated matrix is : ");
      for(i=0;i<5;i++)
      {
            printf("\nP%d : ",i);
            for(j=0;j<3;j++)
                  printf("%d ",alloc[i][j]);
      }
      printf("\n\nThe Maximum resource matrix is : ");
      for(i=0;i<5;i++)
        {
            printf("\nP%d : ",i);
                for(j=0;j<3;j++)
                        printf("%d ",max[i][j]);
      }
      printf("\n\nThe resource matrix is:");
      for(i=0;i<3;i++)
            printf("%d ",avail[i]);
      sleep(2);
      find_need();
      j=processing(0);
      return 0;
}
```

Sample Input and Output

```
Enter the number of processes:5
Enter the number of resources :3
 enter available vector for 3 resource types : 3 3 2
maximum demand of each resource type..
 process 1 :7 5 3
 process 2 :3 2 2
 process 3 :9 0 2
process 4 :2 2 2
 process 5 :4 3 3
 allocated resources for each process..
 process 1 :0 1 0
 process 2 :2 0 0
 process 3 :3 0 2
 process 4 :2 1 1
 process 5 :0 0 2

 need matrix
      7 4 3
      1 2 2
      6 0 0
      0 1 1
      4 3 1
System is in safe state....
```

C. Fork

Basically, a running instance of a program is called a process. Each process in a Linux system is identified by its unique process ID sometimes referred to as PID. Process IDs are 16-bit numbers that are assigned sequentially by Linux as new processes are created. Fork is a function that makes a child process that is an exact copy of its parent process. When a program calls fork, a duplicate process, called the child process, is created. The parent process continues executing the program from the point that fork was called. The child process, too, executes the same program from the same place. First, the child process is a new process and therefore has a new process ID, distinct from its parent's process ID. One way for a program to distinguish whether it's in the parent process or the child process is to call getpid. However, the fork function provides different return values to the parent and child processes—one process "goes in" to the fork call, and two processes "come out" with different return values. The return value in the parent process is the process ID of the child. The return value in the child process is zero.

A child process is created from the parent process using the command

```
child_pid = fork ()
```

D. Interprocess Communication (IPC)

Interprocess communication (IPC) is the transfer of data among processes. For example, a Web browser may request a Web page from a Web server, which then sends HTML data. This transfer of data usually uses sockets in a telephone-like connection. In another example, you may want to print the filenames in a directory using a command such as ls | lpr. The shell creates an ls process and a separate lpr process, connecting the two with a pipe, represented by the "|" symbol. A pipe permits one-way communication between two related processes. The ls process writes data into the pipe, and the lpr process reads data from the pipe. There are mainly four types of interprocess communication:

- Shared memory permits processes to communicate by simply reading and writing to a specified memory location.
- Pipes permit sequential communication from one process to a related process.
- Message queue permits interprocess communication by using a common key to communicate.
- Sockets support communication between unrelated processes even on different computers.

Shared Memory

One of the simplest interprocess communication methods is using shared memory. Shared memory allows two or more processes to access the same memory as if they all called malloc and were returned pointers to the same actual memory. When one process changes the memory, all the other processes see the modification. Shared memory is the fastest form of interprocess communication because all processes share the same piece of memory. Access to this shared memory is as fast as accessing a process's non-shared memory, and it does not require a system call or entry to the kernel. It also avoids copying data unnecessarily. Because the kernel does not synchronize accesses to shared memory, we must provide our own synchronization. For example, a process should not read from the memory until after data is written there, and two processes must not write to the same memory location at the same time. To use a shared memory segment, one process must allocate the segment. Then each process desiring to access the segment must attach the segment. After finishing its use of the segment, each process detaches the segment. At some point, one process must de-allocate the segment.

Allocation in Shared Memory

A process allocates a shared memory segment using shmget ("SHared Memory GET"). Its first parameter is an integer key that specifies which segment to create. Unrelated processes can access the same shared segment by specifying the same key value. Unfortunately, other processes may have also chosen the same fixed key, which could lead to conflict. Using the special constant IPC_PRIVATE as the key value guarantees that a brand new memory segment is created.

Its second parameter specifies the number of bytes in the segment. Because segments are allocated using pages, the number of actually allocated bytes is rounded up to an integral multiple of the page size. The third parameter is the bitwise or of flag values that specify options to shmget. The flag values include these:

- IPC_CREAT—this flag indicates that a new segment should be created. This permits creating a new segment while specifying a key value.

- IPC_EXCL—this flag, which is always used with IPC_CREAT, causes shmget to fail if a segment key is specified that already exists. Therefore, it arranges for the calling process to have an “exclusive” segment. If this flag is not given and the key of an existing segment is used, shmget returns the existing segment instead of creating a new one.

- Mode flags—this value is made of 9 bits indicating permissions granted to owner, group, and world to control access to the segment. Execution bits are ignored. An easy way to specify permissions is to use the constants defined in <sys/stat.h>. For example, S_IRUSR and S_IWUSR specify read and write permissions for the owner of the shared memory segment, and S_IROTH and S_IWOTH specify read and write permissions for others.

```
segment_id=shmget(shm_key,getpagesize(),IPC_CREAT|S_IRUSR|S_IWUSER);
```

Attachment and Detachment

To make the shared memory segment available, a process must use shmat, “Shared Memory ATtach” Pass it the shared memory segment identifier SHMID returned by shmget. The second argument is a pointer that specifies where in your process’s address space you want to map the shared memory; if you specify NULL, Linux will choose an available address .The third argument is a flag, which can include the following:

- SHM_RND indicates that the address specified for the second parameter should be rounded down to a multiple of the page size. If you don’t specify this flag, you must page-align the second argument to shmat yourself.

- SHM_RDONLY indicates that the segment will be only read, not written.

When you’re finished with a shared memory segment, the segment should be detached using shmdt (“SHared Memory DeTach”). Pass it the address returned by shmat. If the segment has been deallocated and this was the last process using it, it is removed.

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

Controlling and Deallocating Shared Memory

The shmctl (“SHared Memory ConTroL”) call returns information about a shared memory segment and can modify it. The first parameter is a shared memory segment identifier. To obtain information about a shared memory segment, pass IPC_STAT as the second argument and a pointer to a struct shmid_ds. To remove a segment, pass IPC_RMID as the second argument, and pass NULL as the third argument. The segment is removed when the last process that has attached it finally detaches it. Each shared memory segment should be explicitly deallocated using shmctl when you’re finished with it, to avoid violating the system wide limit on the total number of shared memory segments.

PIPES

A pipe is a communication device that permits unidirectional communication. Data written to the “write end” of the pipe is read back from the “read end”. Pipes are serial devices; the data is always read from the pipe in the same order it was written. Typically, a pipe is used to communicate between two threads in a single process or between parent and child processes. In a shell, the symbol | creates a pipe. For example, this shell command causes the shell to produce two child processes, one for ls and one for less:

```
% ls | less
```

The shell also creates a pipe connecting the standard output of the ls sub process with the standard input of the less process. The filenames listed by ls are sent to less in exactly the same order as if they were sent directly to the terminal. A pipe's data capacity is limited. If the writer process writes faster than the reader process consumes the data, and if the pipe cannot store more data, the writer process blocks until more capacity becomes available. If the reader tries to read but no data is available, it blocks until data becomes available. Thus, the pipe automatically synchronizes the two processes. The following are the main functions used in the implementation of PIPE for interprocess communication.

access () :

syntax: int access(const char *pathname, int mode)

The function checks whether the calling process can access the file pathname. The mode specifies the accessibility check(s) to be performed, and is either the value F_OK, or a mask consisting of the bitwise OR of one or more of R_OK, W_OK, and X_OK. F_OK tests for the existence of the file.

open () :

syntax: int open(const char *pathname, int flags);

Given a pathname for a file, open () returns a file descriptor, a small, non-negative integer for use in subsequent system calls. The argument flags must include one of the following access modes: O_RDONLY, O_WRONLY, or O_RDWR. This requests the file to open in read-only, write-only, or read / write, respectively.

read () :

syntax : read(int fd, void *buf, size_t count);

The function attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.

write () :

syntax : write(int fd, void *buf, size_t count);

The function writes up to count bytes from the buffer pointed buf to the file referred to by the file descriptor fd.

Message Queue

Message queue can be used for inter process communication between related or unrelated process on a given host. Message queues are identified by a message queue identifier. Any process with adequate privileges can place a message onto a given queue, and any process with adequate privileges can read a message from a given queue. There is no requirement that some process be waiting for a message to arrive on queue before some process writes a message to that queue.

Creating a Message Queue

A new message queue is created or an existing message queue is accessed with the msgget function.

int msgget (key_t key, int oflag);

The return value is an integer identifier that is used in the other three msg functions to refer to this queue, based on the specified key. oflag is a combination of the read-write permission values. This can be bitwise-ORed with either IPC_CREAT or IPC_CREAT|IPC_EXCL.

Sending Message to Message Queue

Once a message queue is opened by msgget, we put a message onto the queue using msgsnd.

```
int msgsnd (int msgid, const void *ptr, size_t length, int flag)
```

msgid is an identifier returned by msgget. ptr is a pointer to a structure with the following template, which is defined in <sys/msg.h>

```
struct msgbuf
{
        long mtype; /* message type must be > 0 */
        char mtext[1]; /* message data */
};
```

The message type must be greater than 0, since non positive message types are used as a special indicator to the msgrcv function.

The length argument to msgsnd specifies the length of the message in bytes. This is the length of the user-defined data that follows the long integer message type. The length can be 0.

The flag argument can be either 0 or IPC_NOWAIT. This flag makes the call to msgsnd nonblocking: the function returns immediately if no room is available for the new message.

Receiving Message from Message Queue

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

size_t msgrcv (int msgid, void *ptr, size_t length, long type, int flag)

The ptr argument specifies where the received message is to be stored.

length specifies the size of the data portion of the buffer pointed to by ptr. This is the maximum amount of data that is returned by the function. This length excludes the long integer type field.

Type specifies which message on the queue is desired:

- If type is 0, the first message on the queue is returned. Since each message queue is maintained as a FIFO list, a type of 0 specifies that the oldest message on the queue is to be returned.
- If type is greater than 0, the first message whose type equals ‘type’ is returned.
- If type is less than 0, the first message with the lowest type that is less than or equal to the absolute value of the ‘type’ argument is returned.

The flag argument specifies what to do if a message of the requested type is not on the queue. If the IPC_NOWAIT bit is set and no message is available, the msgrcv function returns immediately with an error of ENOMSG.

Controlling the Message Queue

The msgctl function provides a variety of control operations on a message queue.

int msgctl (int msgid, int cmd, struct msqid_ds *buff)

In the cmd field, we use IPC_RMID which removes the message queue specified by msgid from the system. Any messages currently on the queue are discarded. The third argument is ignored for this operation.

SOCKETS

A socket is a bidirectional communication device that can be used to communicate with another process on the same machine or with a process running on other machines. When you create a socket, you must specify three parameters: communication style, namespace, and protocol. A communication style controls how the socket treats transmitted data and specifies the number of communication partners. When data is sent through a socket, it is packaged into chunks called packets. The communication style determines how these packets are handled and how they are addressed from the sender to the receiver.

- Socket Stream styles guarantee delivery of all packets in the order they were sent. If packets are lost or reordered by problems in the network, the receiver automatically requests their retransmission from the sender. A connection-style socket is like a telephone call: The addresses of the sender and receiver are fixed at the beginning of the communication when the connection is established.

- Datagram styles do not guarantee delivery or arrival order. Packets may be lost or reordered in transit due to network errors or other conditions. Each packet must be labeled with its destination and is not guaranteed to be delivered. The system guarantees only "best effort," so packets may disappear or arrive in a different order than shipping. A datagram-style socket behaves more like postal mail. The sender specifies the receiver's address for each individual message.

3.Expression Evaluation using Fork and Shared Memory

Problem Description

Develop a program to evaluate an expression evaluation (a*b) + (c*d), where a*b is evaluated in child process and c*d in parent process and result is computed in parent process.

Data Structures and Functions

Data Structures used here are:

- pid_t: It is capable of representing a process ID.
- key_t: UNIX requires a key of type key_t defined in file sys/types.

Functions used here are:

- shmget():

  Syntax: shmid = shmget (key, size, IPC_CREAT|0666);

- shmat ():

  Syntax: shmat (int shmid, const void *shmaddr, int shmflg);

- fork ():

  Syntax: pid_t id

  id = fork();

Algorithm

1. Start
2. Declare a variable id of type pid_t and key of type key_t
3. fork() is called and an unique identifier will be returned to id.
4. If id =0, child process is executed.
5. Create a shared memory using shmget() and attach it using shmat()
6. Compute a*b
7. If id>0,parent process is executed.
8. Create a shared memory using shmget() and attach it using shmat()
9. Compute c*d
10. Computed value of c*d is obtained from memory and added it to a*b in child process.
11. Stop

Code

```
#include<stdio.h>
#include<fcntl.h>
#include<unistd.h>
#include<sys/shm.h>
#include<sys/stat.h>
int main()
{
      int identifier,a=2,b=3,c=6,d=7;
      identifier=fork();
      if(identifier==0)
      {
            int segment_id;
            char res1[2];
            char* shared_memory;
              int segment_size;
              const int shared_segment_size = 27;
              segment_id=shmget((key_t)5678,shared_segment_size,IPC_CREAT);
            res1[0]=48+(a*b);
              shared_memory = shmat(segment_id,NULL,0);
              printf("\n\nAttached at address : %p",shared_memory);
              sprintf(shared_memory,res1);
              shmdt(shared_memory);
              return 0;
      }
      else if(identifier>0)
      {
            int segment_id,res2,res;
              char* shared_memory;
              int segment_size;
              char str[20];
              const int shared_segment_size = 27;
            segment_id=shmget((key_t)5678,shared_segment_size,IPC_CREAT);
              shared_memory = shmat(segment_id,NULL,0);
            printf("\n\nShared Memory to be read attached at address :
%p",shared_memory);
              sscanf(shared_memory,"%s",str);
            res2=c*d;
            res=res2+(str[0]-48);
            printf("\n\nValue of a : %d, b : %d, c : %d, d : %d",a,b,c,d);
            printf("\n\nResult : %d",res);
              shmdt(shared_memory);
            return 0;
      }
}
```

Sample Input and Output

```
Value of a: 2, b: 3, c: 6, d: 7
Result : 48
```

4. IPC using PIPES

Problem Description

Develop a program to implement communication between two processes using pipes.

Data Structures and Functions

Data Structures used are: Arrays

Functions used here are:

- access() :

  syntax: int access(const char *pathname, int mode)

- open() :

  syntax : int open(const char *pathname, int flags);

- read() :

  syntax : read(int fd, void *buf, size_t count);

- write() :

  syntax : write(int fd, void *buf, size_t count);

Algorithm

Sender

1. Start
2. Declare writepipe and readpipe.
3. If writepipe is accessible, create pipe using mkfifo()
4. If readpipe is accessible, create pipe using mkfifo()
5. Open the writepipe in write-only mode
6. Open the readpipe in read-pipe mode
7. Write data into writepipe using write() function
8. Read data from readpipe using read() function

Client

1. Declare readpipe and writepipe in reverse as that of sender side
2. If readpipe is accessible, create pipe using mkfifo()
3. If writepipe is accessible, create pipe using mkfifo()
4. Open the readpipe in read-pipe mode
5. Open the writepipe in write-only mode
6. Read data from readpipe using read() function
7. Write data into writepipe using write() function
8. Stop

Code

Receiver

```
#include<stdio.h>
#include<stdlib.h>
#include<fcntl.h>
main()
{
        char readpipe[] = "pipe1";
        char writepipe[] = "pipe2";
        int err,len,i;
        char str1[50],str2[50];
        int f1,f2;
        if(access(readpipe,F_OK)==-1){
                err=mkfifo(readpipe,0777);
                if(err==-1)
                {
                        printf("Error !!");
                        exit(1);
                }}
      if(access(writepipe,F_OK)==-1){
                err=mkfifo(writepipe,0777);
                if(err==-1){
                        printf("Error !!");
                        exit(1);
                }}
        f1 = open(readpipe,O_RDONLY);
        if(f1==-1){
                printf("Error in opening the file");
                exit(1);}
        f2 = open(writepipe,O_WRONLY);
        if(f2==-1){
                printf("Error in opening the file");
                exit(1);}
        read(f1,str1,sizeof(str1));
        printf("Received string is: %s",str1);
        len=strlen(str1)-1;
        int k=0;
        for(i=len;i>=0;i--){
                str2[k]=str1[i];
                k++;}
        str2[k]='\0';
        write(f2,str2,sizeof(str2));
        close(f1);
        close(f2);
        return(0);
}
```

Sender

```
#include<stdio.h>
#include<fcntl.h>
#include<stdlib.h>
main()
{
        char writepipe[] = "pipe1";
        char readpipe[] = "pipe2";
        int err;
        char str1[50], str2[50];
        int f1,f2;
        if(access(writepipe,F_OK)== -1){
                err = mkfifo(writepipe,0777);
                if(err==-1){
                        printf("Error !!");
                        exit(1);}}
        if(access(readpipe,F_OK)== -1){
                err = mkfifo(readpipe,0777);
                if(err==-1){
                        printf("Error !!");
                        exit(1);}}
        f1 = open(writepipe,O_WRONLY);
        if(f1==-1)
        {
                printf("Error !!");
                exit(1);
        }
        f2 = open(readpipe,O_RDONLY);
        if(f2==-1)
        {
                printf("Error !!");
                exit(1);
        }
                printf("Enter the string.. ");
                scanf("%s",str1);
                write(f1,str1,sizeof(str1));
                read(f2,str2,sizeof(str2));
                printf("The reversed string  is: %s",str2);

        close(f1);
        close(f2);
        return(0);
}
```

Sample Input and Output

Sender

```
enter the message hello
reversed message is :olleh
enter the message hwru
reversed message is :urwh
enter the message quit
```

Receiver

```
hello
hwru
```

5. IPC using Message Queue

Problem Description

Develop a program to implement communication between two process using message queue.

Data Structures and Functions

Data Structures used here are: Arrays and Structure

```
struct name{
        long int msgtype;  /*Message Type*/
        char buf[100];   /*Message Text*/
};
```

Functions used here are:

- msgget() :

Syntax: msqid = msgget(key, (IPC_CREAT | IPC_EXCL | 0400))

- msgsnd() :

Syntax: int msgsnd(int msqid, const void *msgp, size_t msgsz,int msgflg)

- msgrcv() :

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

- msgctl() :

Syntax: k=msgctl(id,IPC_RMID,0)

Algorithm

Sender

1. Define a structure with an integer variable and a character array.
2. Create a message queue using msgget().
3. Set message type as 1.
4. Accept the message.
5. Send the message using msgsnd().
6. Receive the reversed message using msgrcv().
7. Repeat the process until “quit” is send.

Receiver

1. Define a structure with an integer variable and a character array.
2. Create a message queue using msgget().
3. Set message type as 1.
4. Receive the message using msgrcv()
5. Print the message.
6. Reverse the message and send it back to the sender.
7. Repeat the process until “quit” is received.
8. Remove the messageid using msgctl().
9. Stop

Code

Receiver

```
#include<sys/msg.h>
struct name
{
      long int msgtype;
      char buf[100];
};
int main()
{
      struct name rcv;
      int msgid,i,len,j=0;
      char temp[100][100];
      printf("The received data is : ");
      while(1)
      {
            strcpy(rcv.buf,"\0");
            strcpy(temp[j],"\0");
            msgid=msgget((key_t)2345,0666|IPC_CREAT);
            if(msgid==-1)
            {
                  printf("\nError in creating..");
            }
            rcv.msgtype=1;
            msgrcv(msgid,(void*)&rcv,100,0,0);
            if(strcmp(rcv.buf,"quit")==0)
                  break;
            printf("%s",rcv.buf);
            len=strlen(rcv.buf)-1;
            for(i=len;i>0;i--)
                  temp[j][i]=rcv.buf[i];
            j++;
      }
      for(i=0;i<j;i++)
      {
            strcpy(rcv.buf,temp[i]);
            msgsnd(msgid,(void*)&rcv,100,0);
      }
      if(i==j)
      {
            strcpy(rcv.buf,"quit");
            msgsnd(msgid,(void*)&rcv,100,0);
      }
        msgctl(msgid,IPC_RMID,0);
      return(0);
}
```

Sender

```
#include<sys/msg.h>
struct name
{
      long int msgtype;
      char buf[100];
};
int main()
{
      struct name snd;
      int msgid;
        printf("\nEnter the message to be send : ");
      while(1)
      {
            msgid=msgget((key_t)2345,0666|IPC_CREAT);
            if(msgid==-1)
            {
                  printf("\nError in creating..");
            }
            snd.msgtype=1;
            scanf("%s",snd.buf);
            msgsnd(msgid,(void*)&snd,100,0);
            if(strcmp(snd.buf,"quit")==0)
                  break;
            strcpy(snd.buf,"\0");
      }
      while(1)
      {
            msgrcv(msgid,(void*)&snd,100,0,0);
            if(strcmp(snd.buf,"quit")==0)
                  break;
            printf("%s",snd.buf);
      }
      msgctl(msgid,IPC_RMID,0);
      return(0);
}
```

Sample Input and Output

Sender

```
enter the message hello
reversed message is :olleh
enter the message hwru
reversed message is :urwh
enter the message quit
```

Receiver

```
hello
hwru
```

E. Socket Programming

A socket is a communications connection point (endpoint) that you can name and address in a network. The processes that use a socket can reside on the same system or on different systems on different networks. Sockets are useful for both stand-alone and network applications. A socket allow you to exchange information between processes on the same machine or across a network, distribute work to most efficient machine and allows access to centralized data easily.

The connection that a socket provides can be connection-oriented or connectionless.

Connection-oriented communication implies that a connection is established, and a dialog between the programs will follow. The program that provides the service (the server program) establishes the available socket which is enabled to accept incoming connection requests. Optionally, the server can assign a name to the service that it supplies which allows clients to identify where to obtain and how to connect to that service. The client of the service (the client program) must request the service of the server program. The client does this by connecting to the distinct name or to the attributes associated with the distinct name that the server program has designated. It is similar to dialing a telephone number (an identifier) and making a connection with another party that is offering a service (for example, a plumber). When the receiver of the call (the server, in this example, a plumber) answers the telephone, the connection is established. The plumber verifies that you have reached the correct party, and the connection remains active as long as both parties require it. Connectionless communication implies that no connection is established over which a dialog or data transfer can take place. Instead, the server program designates a name that identifies where to reach it (much like a post office box).

SOCKET TYPE

Stream (SOCK_STREAM):

This type of socket is connection-oriented. Establish an end-to-end connection by using the bind (), listen (), accept (), and connect () functions. SOCK_STREAM sends data without errors or duplication, and receives the data in the sending order. SOCK_STREAM builds flow control to avoid data overruns. It does not impose record boundaries on the data. SOCK_STREAM considers the data to be a stream of bytes.

Datagram (SOCK_DGRAM):

In Internet Protocol terminology, the basic unit of data transfer is a datagram. The datagram socket is connectionless. It establishes no end-to-end connection with the transport provider (protocol). The socket sends datagrams as independent packets with no guarantee of delivery. Datagrams can arrive out of order. For some transport providers, each datagram can use a different route through the network.

Creating a connection-oriented socket:

A connection-oriented server uses the following sequence of function calls:

socket() bind() connect() listen() accept() send() recv() close()

A connection-orientated client uses the following sequence of function calls:

socket () gethostbyname() connect() send() recv() close()

Creating a connectionless socket:

A connectionless client illustrates the socket APIs that are written for User Datagram Protocol (UDP).

The server uses the following sequence of function calls:

socket () bind() sendto() recvfrom() close()

The client example uses the following sequence of function calls:

socket () gethostbyname() sendto() recvfrom() close().

Functions and their parameters:

- socket ()

  This function gives a socket descriptor that can be used in later system calls.

  syntax: int socket (int domain, int type, int protocol)

  domain-> AF_INET or AF_UNIX

  type-> the type of socket needed (Stream or Datagram).

  SOCK_STREAM for Stream socket

  SOCK_DGRAM for Datagram Socket

  protocol-> 0

- bind ()

  This function associates a socket with a port.

  syntax: int bind (int fd, struct sockaddr *my_addr, int addrlen)

  fd-> socket descriptor

  my_addr-> ptr to structure sockaddr

  addrlen-> sizeof(struct sockaddr)

- connect ()

  This function is used to connect to an IP address on a defined port.

  syntax: int connect (int fd, struct sockaddr* serv_addr, int addrlen)

  fd-> socket file descriptor

  addrlen -> sizeof (struct sockaddr)

- listen ()

  This function is used to wait for incoming connection. Before calling listen(), bind() is to be called. After calling listen (), accept () is to be called in order to accept incoming connection.

  syntax: int listen (int fd, int backlog)

  fd-> socket file descriptor

  backlog-> number of allowed connections

- accept ()

  This function is used to accept an incoming connection.

  syntax: int accept (int fd, void* addr, int *addrlen)

  fd-> socket file descriptor

  addr-> ptr to struct sockaddr

  addrlen-> sizeof (struct sockaddr)

  - send (): This function is used to send data over stream sockets. It returns the number of bytes sent out.

syntax: int send (int fd, const void *msg, int len, int flags)

fd-> socket descriptor

msg-> ptr to the data to be send

len-> length of the data to be send

flags-> 0

- recv ()

  This function receives the data send over the stream sockets. It returns the number of bytes read into the buffer.

  syntax: int recv (int fd, void *buf, int len,unsigned int flags)

  fd-> socket file descriptor

  buf-> buffer to read the information into

  len-> maximum length of the buffer

  flags-> set to 0

- sendto ()

  This function serves the same purpose as send () function except that it is used for datagram sockets. It also returns the number of bytes sent out.

  syntax: int sendto (int fd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen)

  fd-> socket file descriptor

  msg-> ptr to the data to be send

len-> length of the data to be send

flags-> 0

to-> ptr to a struct sockaddr

tolen-> sizeof (struct sockaddr)

- recvfrom ()

  This function serves the same purpose as recv () function except that it is used for datagram sockets. It also returns the number of bytes received.

  syntax: int sendto (int fd, const void *buf, int len, unsigned int flags, const struct sockaddr *from, int fromlen)

  fd-> socket file descriptor

  buf-> buffer to read the information into

  len-> maximum length of the buffer

  flags-> set to 0

  from-> ptr to struct sockaddr

  fromlen-> sizeof (ptr to an int that should be initialized to struct sockaddr)

- close()

  This function is used to close the connection on your socket descriptor.

  syntax: close (fd);

  fd-> socket file descriptor

6. Client-Server Communication using TCP

Problem Description

Develop a program to implement interprocess communication using stream sockets with the help of interfaces provided by the standard library.

Data Structures and Functions

Data Structures used here are:

struct sockaddr:

This structure holds socket address information for many types of sockets:

```
struct sockaddr
        {
                unsigned short    sa_family;    // address family, AF_xxx
                char sa_data[14];  // 14 bytes of protocol address
        };
```

sa_family can be a variety of things, but it'll be AF_INET for everything we do in this document. sa_data contains a destination address and port number for the socket.

To deal with struct sockaddr, programmers created a parallel structure:

struct sockaddr_in ("in" for "Internet".)

```
struct sockaddr_in
{
        short int          sin_family;  // Address family
         unsigned short int sin_port;    // Port number
         struct in_addr     sin_addr;    // Internet address
         unsigned char      sin_zero[8]; // Same size as struct sockaddr
};
```

This structure makes it easy to reference elements of the socket address. Note that sin_zero (which is included to pad the structure to the length of a struct sockaddr) should be set to all zeros with the function memset. Finally, the sin_port and sin_addr must be in Network Byte Order!

Functions used here are:

System calls that allow to access the network functionality of a Unix box are as given below. When you call one of these functions, the kernel takes over and does all the work for you automatically.

Server Side:

socket () bind() connect() listen() accept() send() recv() close()

Client Side:

socket () gethostbyname() connect() send() recv() close()

Algorithm

TCP Server

1. Create a socket

2. Bind it to the operating system.

3. Listen over it.

4. Accept connections.

5. Receive data from client and reverse it, send it back to client.

6. Close the socket.

TCP Client

1. Create a datagram socket.

2. Receive / send message to the server.

3. Close the socket.

Code

Client

```
#include"sys/socket.h"
#include"netinet/in.h"
#include"stdio.h"
#include"string.h"
#include"stdlib.h"
int main(){
  char buf[100];
  int k;
  int sock_desc;
  struct sockaddr_in client;
  memset(&client,0,sizeof(client));
  sock_desc=socket(AF_INET,SOCK_STREAM,0);
  if(sock_desc==-1){
    printf("Error in socket creation");
    exit(1);
  }
  client.sin_family=AF_INET;
  client.sin_addr.s_addr=INADDR_ANY;
  client.sin_port=3001;
  k=connect(sock_desc,(struct sockaddr*)&client,sizeof(client));
  if(k==-1){
    printf("Error in connecting to server");
    exit(1);
  }
  printf("\nEnter data to be send : ");
  gets(buf);
  k=send(sock_desc,buf,100,0);
  if(k==-1){
    printf("Error in sending");
    exit(1);
  }
  close(sock_desc);
  exit(0);
  return 0;
}
```

Server

```
#include"sys/socket.h"
#include"netinet/in.h"
#include"stdio.h"
#include"string.h"
#include"stdlib.h"
int main(){
  char buf[100];
  int k;
  socklen_t len;
  int sock_desc,temp_sock_desc;
  struct sockaddr_in server,client;
  memset(&server,0,sizeof(server));
  memset(&client,0,sizeof(client));

  sock_desc=socket(AF_INET,SOCK_STREAM,0);
  if(sock_desc==-1){
    printf("Error in socket creation");
    exit(1);
  }
  server.sin_family=AF_INET;
  server.sin_addr.s_addr=inet_addr("127.0.0.1");
  server.sin_port=3001;
  k=bind(sock_desc,(struct sockaddr*)&server,sizeof(server));
  if(k==-1){
    printf("Error in binding");
    exit(1);
  }
  k=listen(sock_desc,20);
  if(k==-1){
    printf("Error in listening");
    exit(1);
  }
  len=sizeof(client);
  temp_sock_desc=accept(sock_desc,(struct sockaddr*)&client,&len);
  if(temp_sock_desc==-1){
    printf("Error in temporary socket creation");
    exit(1);
    }
  k=recv(temp_sock_desc,buf,100,0);
  if(k==-1){
    printf("Error in receiving");
    exit(1);
  }
  printf("Message got from client is : %s",buf);
  close(temp_sock_desc);
  exit(0);
  return 0;
}
```

Sample Input and Output

Client

```
      Enter the data:how are you
      received: uoy era who
```

Server

```
      Received:how are you
```

7. Client-Server Communication using UDP

Problem Description

Develop a program to implement interprocess communication using datagram sockets with the help of interfaces provided by the standard library.

Data Structures and Functions

Data structures used are:

struct sockaddr:

This structure holds socket address information for many types of sockets:

```
struct sockaddr
{
        unsigned short    sa_family;    // address family, AF_xxx
        char sa_data[14];  // 14 bytes of protocol address
};
```

sa_family can be a variety of things, but it'll be AF_INET for everything we do in this document. sa_data contains a destination address and port number for the socket. To deal with struct sockaddr, programmers created a parallel structure:

struct sockaddr_in (“in” for “Internet”.)

```
struct sockaddr_in
{
        short int         sin_family;  // Address family
        unsigned short int sin_port;    // Port number
        struct in_addr    sin_addr;    // Internet address
        unsigned char     sin_zero[8]; // Same size as struct sockaddr
};
```

This structure makes it easy to reference elements of the socket address. Note that sin_zero (which is included to pad the structure to the length of a struct sockaddr) should be set to all zeros with the function memset. Finally, the sin_port and sin_addr must be in Network Byte Order!

Functions used here are:

System calls that allow to access the network functionality of a Unix box are as given below. When you call one of these functions, the kernel takes over and does all the work for you automatically.

The server uses the following sequence of function calls:

socket () bind() sendto() recvfrom() close()

The client example uses the following sequence of function calls:

socket () gethostbyname() sendto() recvfrom() close().

Algorithm

UDP Server

1. Create the internal host *server and initialize

2. Initialize the members of sin_addr structure and port address

3. Create the socket so with parameters (AF_INET, SOCK_DGRAM, 0) for datagram communication

4. Bind the socket to its address

5. Receive message sent from client, reverse it and send it back.

6. Display message

7. Close the socket

UDP Client

1. Initialize port address corresponding to the server

2. Initialize the members of sin_addr structure

3. Create the socket so with parameters (AF_INET, SOCK_DGRAM, 0) for datagram communication

4. Send message to server

Code

Client

```
#include"sys/socket.h"
#include"sys/types.h"
#include"arpa/inet.h"
#include"netdb.h"
#include"unistd.h"
#include"stdio.h"
#include"string.h"
#include"stdlib.h"

int main(){
  socklen_t len;
  char buf[100];
  int sock_desc;
  int k;
  struct sockaddr_in client;

  memset(&client,0,sizeof(client));

  sock_desc=socket(AF_INET,SOCK_DGRAM,0);
  if(sock_desc==-1){
    printf("Error in creating socket");
    exit(1);
  }

  client.sin_family=AF_INET;
  client.sin_addr.s_addr=INADDR_ANY;
  client.sin_port=3000;

  k=bind(sock_desc,(struct sockaddr*)&client,sizeof(client));
  if(k==-1){
    printf("\nError in binding socket");
    exit(1);
    }

  while(1){
    len=sizeof(client);
    k=recvfrom(sock_desc,buf,100,0,(struct sockadddr*)&client,&len);
    if(k==-1){
      printf("\nError in receiving pkt");
      exit(1);
    }
    if((strcmp(buf,"end")==0)){
      printf("End of transmission from server");
      break;
    }
    printf("\n%s\n",buf);
  }
  close(sock_desc);
  exit(0);
  return 0;
}
```

Server

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

int main(){
  char buf[100];
  int i=1,k;
  int sock_desc;
  struct sockaddr_in server;
  memset(&server,0,sizeof(server));
  sock_desc=socket(AF_INET,SOCK_DGRAM,0);
  if(sock_desc==-1){
    printf("Error in creating socket");
    exit(1);}
  server.sin_family=AF_INET;
  server.sin_addr.s_addr=inet_addr("127.0.0.1");
  server.sin_port=3000;
  while(1){
    sprintf(buf,"Data Pkt %d",i);
    k=sendto(sock_desc,buf,100,0,(struct sockaddr *)&server,sizeof(server));
    if(k==-1){
      printf("\nError in sending data pkts");
      exit(1);}
    sleep(1);
    if(i>=5){
      strcpy(buf,"end");
      k=sendto(sock_desc,buf,100,0,(struct sockaddr
*)&server,sizeof(server));
      break;
    }
    i++;
  }
  close(sock_desc);
  exit(0);
  return 0;
}
```

Sample Input and Output

Client

```
      Enter the data: how are you
      Received: uoy era who
```

Server

```
      Received: how are you
```

F. MAC Protocols

The Media Access Control (MAC) data communication protocol sub-layer , also known as the medium Access Control, is a part of the data link layer specified in the seven layer OSI model (layer 2) . It provides addressing and channel access control mechanisms that make it possible for several terminal or network nodes to communicate within a multipoint network (LAN) or Metropolitan Area Network (MAN). AMAC protocol is not required in full-duplex point –to- point communication. In single channel point –to- point communications full-duplex can be emulated. This emulation can be considered a MAC layer.

The MAC sub-layer acts as an interface between the Logical Link Control Sublayer and the network’s physical layer. The MAC layer provides an addressing mechanism called physical address or MAC address. This is a unique serial number assigned to each network adapter, making it possible to deliver data packets to a destination within a subnetwork, i.e a physical network without routers, for example an Ethernet network. Media access control is often used as a synonym to multiple access protocol, since the MAC Sublayer provides the protocol and control mechanisms that are required for a certain channel access method. This makes it possible for several stations connected to the same physical medium to share it. Example of shared physical medium are bus networks, ring networks, hub networks, wireless networks and half duplex point-to–point links.

SLIDING WINDOW PROTOCOL

In the simplex, stop and wait protocols, data frames were transmitted in one direction only. In most practical solutions, there is a need for transmitting data in both directions. One way of achieving full-duplex data transmission is to have to separate communication channels and use one for simplex data traffic. If this is done, we have two separate physical circuits, each with a “forward: channel (for data) and a “reverse” channel (for acknowledgments). In both case the bandwidth of the reverse channel is almost wasted. A better idea is to use same circuit for data in both directions. In this model data frames from A to B are intermixed with the acknowledgment frames A to B. By looking at the field in the header of and incoming frame, the receiver can tell whether the frame is data or acknowledgment. When a data from arrives, instead of immediately sending a separate control frame, the receiver retains itself and waits until the network layer passes it the next packet. The acknowledgment is attached to the outgoing data frame. The technique of temporarily delaying outgoing acknowledgments so that they can be hooked onto the next outgoing data frame is known as piggybacking. The principal advantage of using piggybacking over having distinct acknowledgment frames is a better use of the available channel bandwidth. Bidirectional protocols that belong to a class called sliding window protocols. They differ among themselves in terms of efficiency, complexity, and buffer requirements. All sliding window protocols, each outbound frame contains a sequence number ranging from 0 up to some maximum. The essence of all sliding window protocols is that at any instant of time, the sender maintains a set of sequence numbers corresponding frames it is permitted to send. These frames are said to fall within the sending window, the receiver also maintains a receiving window corresponding to the set of frames it is permitted to accept.

A one – bit sliding window protocol

A sliding window protocol has a maximum window size of 1. Such a protocol use a stop – and – wait since the sender transmits a frame and waits from its acknowledgment before sending the next one. The starting machine fetches the first packet from its network layer, builds a frame from it , and sends it. The acknowledgment field contains the number of last frame received without error. If this number agrees with the sequence number of the frame the sender trying to send, the sender knows it is done with the frame stored in buffer and can fetch the next packet from its network layer. If the sequence number disagrees, it must continue trying to send the same dame frame whenever a frame is received, a frame also sent back.

A Protocol Using Go back N

Two basic approaches are available for dealing with errors in the presence of pipelining. One way, called go back N , is for the receiver simply to discard all subsequent frames, sending no acknowledgments for the discarded frames. In other words, the data link layer refuses to accept any frame except the next one it must give to the network layer.

A Protocol Using Selective Repeat

In this protocol, both the sender and receiver maintain a window of acceptable sequence numbers. The sender's window size starts out at 0 and grows to some predefined maximum. The receiver's window, in contrast, is always fixed in size and equal to maximum. The receiver has a buffer reserved for each sequence number within its fixed window. Whenever a frame arrives, its sequence number is checked by the function between to see if falls within the window. If so and if it has not already been received , it is accepted and stored. This action is taken without regard to whether or not it contains the next packet expected by the network layer.

8. 1-BIT SLIDING WINDOW PROTOCOL

Problem Description

Develop a program to simulate 1-bit sliding window protocol. The sender sends frame to the receiver. Since the size of the sliding window is only one bit, the sender sends one frame and waits for the acknowledgment from the receiver for it. Unless the acknowledgment is received, the sender cannot send more frames. For simulation purpose, any one packet is supposed to be lost for the first time. Then client sends a RETRANSMIT request along with the current packet number. On receiving this, the server retransmits the lost packet.

Data Structures and Functions

Client

Sockfd and newSockFd are integer socket descriptors.

currentPacket is an integer that denotes the packet that is sent last.

Data is a string that is used to store the message.

Server

Sockfd and newSockFd are integer socket descriptors.

currentPacket is an integer that denotes the packet that is sent last.

Data is a string that is used to store the message.

Algorithm

Server

1. Start.

2. Establish connection with the client.

3. Accept the window size from the client.

4. Accept the packet from the network layer.

5. Combine the packets to form frames / window. (depending on window size).

6. Initialize the transmit buffer.

7. Send the frame and wait for the acknowledgment.

8. If a negative acknowledgment is received, repeat the transmission of the previous frame else increment the frame to be transmitted.

9. Repeat steps 5 to 8 until all packets are transmitted successfully.

10. Close the connection.

11. Stop.

Client

1. Start.

2. Establish a connection with the server. (recommended UDP).

3. Send the window size on server request.

4. Accept the details from the server (total packets, total frames).

5. Initialize the receive buffer with the expected frame and packets.

6. Accept the frame and check for its validity.

7. Send the positive or negative acknowledgment as required.

8. Repeat steps 5 to 7 until all packets are received.

9. Close the connection with server.

10. Stop.

Code

Client

```
#include"sys/socket.h"
#include"netinet/in.h"
#include"stdio.h"
#include"string.h"
#include"stdlib.h"
int main(int argc, char* argv[])
{
      char buf[100],id[2];
      int k,i=1,flag=0,num,pid;
      int sock_desc;
      struct sockaddr_in client;
      memset(&client,0,sizeof(client));
      sock_desc=socket(AF_INET,SOCK_STREAM,0);
      if(sock_desc==-1)
      {
            printf("\nError in socket creation");
            exit(1);
      }
      client.sin_family=AF_INET;
      client.sin_addr.s_addr=inet_addr("127.0.0.1");
      client.sin_port=atoi(argv[1]);
      k=connect(sock_desc,(struct sockaddr*)&client,sizeof(client));
      if(k==-1)
      {
            printf("\nError in connecting to server");
            exit(1);
      }
      while(1)
      {
            if(flag==0)
            {
                  printf("\nEnter the next packet number : ");
                  scanf("%d",&i);
            }
            else
                  i++;
            if(i>9)
                  break;
            printf("\nEnter data packet %d to be send : ",i);
            id[0]=i;
            id[1]='\0';
            scanf("%s",buf);
            k=send(sock_desc,id,sizeof(id),0);
            if(k==-1)
            {
                  printf("\nError in sending..");
                  exit(1);
            }
            k=recv(sock_desc,id,sizeof(id),0);
                if(k==-1)
                {
                        printf("\nError in receiving");
                        exit(1);
                }
            if(id[0]=='~')
            {
                  k=send(sock_desc,buf,sizeof(buf),0);
                  if(k==-1)
                  {
                        printf("\nError in sending");
                        exit(1);
                  }
                  sleep(2);
                  strcpy(buf,"\0");
                  k=recv(sock_desc,buf,sizeof(buf),0);
                  if(k==-1)
                  {
                        printf("\nError in receiving");
                        exit(1);
                  }
                  printf("%s by the server\n\t",buf);
            }
            else
            {
                  num=(int)id[0];
                  k=recv(sock_desc,id,sizeof(id),0);
                        if(k==-1)
                        {
                                printf("\nError in receiving");
                                exit(1);
                        }
                  pid=(int)id[0];
                  k=recv(sock_desc,buf,sizeof(buf),0);
                        if(k==-1)
                        {
                                printf("\nError in receiving");
                                exit(1);
```

```
                        }
                  printf("\n%s\n\t",buf);
                  i=pid-1;
                  flag=1;
            }
      }
      if(i==10)
      {
            strcpy(buf,"END");
            k=send(sock_desc,buf,sizeof(buf),0);
            if(k==-1)
            {
                  printf("\nError in sending");
                  exit(1);
            }
      }
      close(sock_desc);
      return 0;
}
```

Server

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

int main(int argc, char* argv[])
{
      char buf[100],c[2],id[2];
      int k,i=0,num=0,old;
      socklen_t len;
      int sock_desc,temp_sock_desc;
      struct sockaddr_in server,client;

      memset(&server,0,sizeof(server));
      memset(&client,0,sizeof(client));

      sock_desc=socket(AF_INET,SOCK_STREAM,0);
      if(sock_desc==-1)
      {
            printf("Error in socket creation");
            exit(1);
      }

      server.sin_family=AF_INET;
      server.sin_addr.s_addr=INADDR_ANY;
      server.sin_port=atoi(argv[1]);
      k=bind(sock_desc,(struct sockaddr*)&server,sizeof(server));
      if(k==-1)
      {
            printf("Error in binding");
```

```
		exit(1);
	}

	k=listen(sock_desc,5);
	if(k==-1)
	{
		printf("Error in listening");
		exit(1);
	}
	len=sizeof(client);
	temp_sock_desc=accept(sock_desc,(struct sockaddr*)&client,&len);
	if(temp_sock_desc==-1)
	{
		printf("\nError in temporary socket creation");
		exit(1);
	}
	printf("\n\nMessage got from client is : \n\t");
	while(1)
	{
		k=recv(temp_sock_desc,id,sizeof(id),0);
		if(k==-1)
		{
			printf("\nError in receiving");
			exit(1);
		}
		old = num;
		num = (int)id[0];
		if((num-old)==1)
		{
			id[0]='~';
				id[1]='\0';
				k=send(temp_sock_desc,id,sizeof(id),0);
				if(k==-1)
				{
					printf("Error in sending\n");
					exit(1);
				}
			k=recv(temp_sock_desc,buf,sizeof(buf),0);
			if(k==-1)
			{
				printf("\nError in receiving");
				exit(1);
			}
			if(strcmp(buf,"END")==0)
				break;
			id[0]=num+48;
			id[1]='\0';
			printf("Data packet %d: %s\n\t",num,buf);
			strcpy(buf,"\0");
			strcpy(buf,"Data packet ");
			strcat(buf,id);
			strcat(buf," acknowledged");
			printf("Acknowledging reception of data packet %s to the 
client..\n\n\t",id);
			k=send(temp_sock_desc,buf,sizeof(buf),0);
```

```
                        if(k==-1)
                        {
                                    printf("Error in sending\n");
                                    exit(1);
                        }
                  }
                  else
                  {
                        strcpy(buf,"\0");
                        id[0]=(num-old-1);
                        id[1]='\0';
                        k=send(temp_sock_desc,id,sizeof(id),0);
                              if(k==-1)
                              {
                                      printf("Error in sending\n");
                                      exit(1);
                              }
                        id[0]=(num-old-1)+48;
                        id[1]='\0';
                        strcpy(buf,"Didn't receive ");
                        strcat(buf,id);
                        if((num-old-1)==1)
                              strcat(buf," data packet starting from packet ");
                        else
                              strcat(buf," data packets starting from packet ");
                        id[0]=(old+1);
                        id[1]='\0';
                        k=send(temp_sock_desc,id,sizeof(id),0);
                              if(k==-1)
                              {
                                      printf("Error in sending\n");
                                      exit(1);
                              }
                        id[0]=(old+1)+48;
                        id[1]='\0';
                        strcat(buf,id);
                        k=send(temp_sock_desc,buf,sizeof(buf),0);
                        if(k==-1)
                        {
                                    printf("Error in sending\n");
                                    exit(1);
                        }
                        num=old;
                  }
                  sleep(1);
            }
            close(temp_sock_desc);
            return 0;
      }
```

Sample Input and Output

server
```
Socket created successfully...
Starting up...
Binding completed successfully. Waiting for connection..
Recieved a request from client. Sending packets one by one...
Packet Sent: 1
Packet Sent: 2
Packet Sent: 3
** Received a RETRANSMIT packet. Resending last packet...
Packet Sent: 3
Packet Sent: 4
Packet Sent: 5
Sending Complete. Sockets closed. Exiting...
```

client
```
Socket created successfully...
Starting up...
Binding completed successfully. Waiting for connection..
Recieved a request from client. Sending packets one by one...
Packet Sent: 1
Packet Sent: 2
Packet Sent: 3
** Received a RETRANSMIT packet. Resending last packet...
Packet Sent: 3
Packet Sent: 4
Packet Sent: 5
Sending Complete. Sockets closed. Exiting...
```

9. Go-Back-N Sliding Window Protocol

Problem Description

This program is a simulation of Go Back N Sliding window protocol. After establishing connection, the server sends 10 packets to the client. For this non blocking send is used. i.e., the packets are sent continuously without waiting for the ACK. If the packet is received successfully, an acknowledgement message is sent back by the client. For simulation purpose, packet 3 is supposed to be lost for the first time. Then client sends a RETRANSMIT request along with the current packet number. On receiving this, the server moves the window Start back to 3. Then it retransmits packets from 3 onwards.

Data Structures and Functions

Client

Sockfd and newSockFd are integer socket descriptors.

currentPacket is an integer that denotes the packet that is sent last.

Digit is used to store the packet no for ACK or Retransmit message.

Server

Sockfd and newSockFd are integer socket descriptors.

WindowStart and WindowEnd are used to store the starting and ending of window.

Data is a string that is used to store the message.

Algorithm

1. Start
2. Establish connection
3. Accept the window size from the client
4. Accept the packet from the network layer.
5. Calculate the total frames / windows required.
6. Send the details to the client (total packets , total frames .)
7. Initialize the transmit buffer.
8. Built the frame / window depending on the window size.
9. Transmit the frame.
10. Wait the acknowledgment frame.
11. Check for the acknowledgment of each packet and repeat the process from the packet for which the first negative acknowledgment is received.

Else continue as usual.

12. Increment the frame count and repeat steps 7 to 12 until all packets are transmitted.
13. Close the connection.
14. Stop.

Client

1. Start.
2. Establish a connection. (Recommended UDP)
3. Send the windowsize on server request.
4. Accept the details from the server (total packets, total frames)
5. Initialize the receive buffer with the expected packets.
6. Accept the frame / window from the server.
7. Check the validity of the packet and construct the acknowledgment frame depending on the validity. (Here the acknowledgment is accepted from the users)
8. Depending on the acknowledgment frame readjust the process.
9. Increment the frame count and repeat steps 5-9 until all packets are received.
10. Close the connection
11. Stop.

Code

```
/********************************/
/* SLIDING WINDOW GO-BACK CLIENT */
/********************************/

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

#define PORT 3599

int main() {
  int sockfd, newSockFd, size, firstTime = 1, currentPacket, wait = 3;
  char data[100], digit[2];
  struct sockaddr_in client;
  memset(&client, 0, sizeof(client));
  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if(sockfd == -1) {
    printf("Error in socket creation...");
  }
  else {
    printf("\nSocket Created...");
  }
  client.sin_family = AF_INET;
  client.sin_port = PORT;
  client.sin_addr.s_addr = inet_addr("127.0.0.1");
  printf("\nStarting up...");
  size = sizeof(client);
  printf("\nEstablishing Connection...");
  if(connect(sockfd, (struct sockaddr *)&client, size) == -1) {
    printf("\nError in connecting to server...");
    exit(1);
  } else {
    printf("\nConnection Established!");
  }
  memset(&data, 0, sizeof(data));
  sprintf(data, "REQUEST");
  if(send(sockfd, data, strlen(data), 0) == -1) {
    printf("Error in sending request for data...");
    exit(1);
}
  do {
    memset(&data, 0, sizeof(data));
    recv(sockfd, data, 100, 0);
    currentPacket = atoi(data);
    printf("\nGot packet: %d", currentPacket);
    if(currentPacket == 3 && firstTime) {
      printf("\n*** Simulation: Packet data corrupted or
incomplete.");
      printf("\n*** Sending RETRANSMIT for packet 1.");
      memset(&data, 0, sizeof(data));
      sprintf(data, "R1");
      if(send(sockfd, data, strlen(data), 0) == -1) {
        printf("\nError in sending RETRANSMIT...");
        exit(1);
      }
      firstTime = 0;
    }
    else {
      wait--;
      if(!wait) {
        printf("\n*** Packet Accepted -> Sending ACK");
        wait = 3;
        memset(&data, 0, sizeof(data));
        sprintf(data, "A");
        digit[0] = (char)(currentPacket + 48);
        digit[1] = '\0';
        strcat(data, digit);
        send(sockfd, data, strlen(data), 0);
      }
    }
  } while(currentPacket != 9);
  printf("\nAll packets recieved... Exiting.");
  close(sockfd);
  return(0);
}

/*********************************/
/* SLIDING WINDOW GO-BACK SERVER */
/*********************************/

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

#define PORT 3599

void itoa(int number, char numberString[]) {
  numberString[0] = (char)(number + 48);
  numberString[1] = '\0';
}

int main()
{
  int sockfd, newSockFd, size, windowStart = 1, windowCurrent = 1,
windowEnd = 4, oldWindowStart, flag;
  char buffer[100];
  socklen_t len;
  struct sockaddr_in server, client;
  memset(&server, 0, sizeof(server));
  memset(&client, 0, sizeof(client));

  if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    printf("\nError in socket creation...");
    exit(1);
  }
  else {
    printf("\nSocket created successfully...");
  }
  server.sin_family = AF_INET;
  server.sin_port = PORT;
  server.sin_addr.s_addr = INADDR_ANY;
  printf("\nStarting up...");
  if(bind(sockfd, (struct sockaddr *)&server, sizeof(server)) == -1) {
    printf("\nBinding Error...");
    exit(1);
  }
  else {
    printf("\nBinding completed successfully. Waiting for
connection..");
  }
  len = sizeof(client);
  if(listen(sockfd, 20) != -1) {
    if((newSockFd = accept(sockfd, (struct sockaddr *)&client,
&len)) == -1) {
      printf("Error in accepting connection...");
      exit(1);
    }
    memset(&buffer, 0, sizeof(buffer));
    if(recv(newSockFd, buffer, 100, 0) == -1) {
      printf("\n Recieve Error! Exiting...");
      exit(1);
    }
    fcntl(newSockFd, F_SETFL, O_NONBLOCK);
    printf("\nRecieved a request from client. Sending packets one
by one...");
    do {
      if(windowCurrent != windowEnd) {
      memset(&buffer, 0, sizeof(buffer));
      itoa(windowCurrent, buffer);
      send(newSockFd, buffer, 100, 0);
      printf("\nPacket Sent: %d", windowCurrent);
      windowCurrent++;
      }
      /*DEBUG*/ printf("\n**%d||%d**", windowCurrent,
windowEnd);
      memset(&buffer, '\0', sizeof(buffer));
      if(recv(newSockFd, buffer, 100, 0) != -1) {
        if(buffer[0] == 'R') {
          printf("\n** Received a RETRANSMIT packet.
Resending packet no. %c...", buffer[1]);
          itoa((atoi(&buffer[1])), buffer);
          send(newSockFd, buffer, 100, 0);
          windowCurrent = atoi(&buffer[0]);
          windowCurrent++;
        }
        else if(buffer[0] == 'A') {
          oldWindowStart = windowStart;
          windowStart = atoi(&buffer[1]) + 1;
          windowEnd += (windowStart - oldWindowStart);
          printf("\n** Recieved ACK %c. Moving window
boundary.", buffer[1]);
            }
      sleep(1);
    } while(windowCurrent != 10);
  }
  else {
    printf("\nError in listening...");
    exit(1); }
  close(sockfd);
  close(newSockFd);
  printf("\nSending Complete. Sockets closed. Exiting...\n");
  return(0);
}
```

Sample Input and Output

```
[root@Linux smtp]#gcc –o gobacknc gobacknc.c

[root@Linux smtp]#gcc –o gobackns gobackns.c

[root@Linux smtp]#./selrejs

[root@Linux smtp]#./selrejc
```

Client

```
Socket Created...
Starting up...
Establishing Connection...
Connection Established!
Got packet: 1
Got packet: 2
Got packet: 3
*** Simulation: Packet data corrupted or incomplete.
*** Sending RETRANSMIT for packet 1.
Got packet: 1
*** Packet Accepted -> Sending ACK
Got packet: 2
Got packet: 3
Got packet: 4
*** Packet Accepted -> Sending ACK
Got packet: 5
Got packet: 6
Got packet: 7
*** Packet Accepted -> Sending ACK
Got packet: 8
Got packet: 9
All packets recieved... Exiting.
```

Server

```
Socket created successfully...
Starting up...
Binding completed successfully. Waiting for connection..
Recieved a request from client. Sending packets one by one...
Packet Sent: 1
**2||4**
Packet Sent: 2
**3||4**
Packet Sent: 3
**4||4**
**4||4**
** Received a RETRANSMIT packet. Resending packet no. 1...
Packet Sent: 2
**3||4**
** Recieved ACK 1. Moving window boundary.
Packet Sent: 3
**4||5**
Packet Sent: 4
**5||5**
**5||5**
** Recieved ACK 4. Moving window boundary.
Packet Sent: 5
**6||8**
Packet Sent: 6
**7||8**
Packet Sent: 7
**8||8**
**8||8**
** Recieved ACK 7. Moving window boundary.
Packet Sent: 8
**9||11**
Packet Sent: 9
**10||11**
Sending Complete. Sockets closed. Exiting...
```

10. Selective Repeat

Problem Description

This program is a simulation of Selective Repeat Sliding window protocol. After establishing connection, the server sends 10 packets to the client. For this non blocking send is used i.e., the packets are sent continuously without waiting for the ACK. If the packet is received successfully, an acknowledgement message is sent back by the client. For simulation purpose, packet 3 is supposed to be lost for the first time. Then client sends a RETRANSMIT request along with the current packet number. On receiving this, the server retransmits the 3rd packet. After that it resumes sending packets from where it stopped.

Data Structures and Functions

Client

Sockfd and newSockFd are integer socket descriptors.

currentPacket is an integer that denotes the packet that is sent last.

Digit is used to store the packet no for ACK or Retransmit message.

Server

Sockfd and newSockFd are integer socket descriptors.

WindowStart and WindowEnd are used to store the starting and ending of window.

Data is a string that is used to store the message.

Algorithm

Server

1. Start.
2. Establish connection (recommended UDP)
3. Accept the window size from the client (should be <= 40)
4. Accept the packet from the network layer.
5. Calculate the total frames / windows required.
6. Send the details to the client (total packets, total frames.)
7. Initialize the transmit buffer.
8. Built the frame / window depending on the window size.
9. Transmit the frame.
10. Wait the acknowledgment frame.
11. Check for the acknowledgment of each packet and repeat the process from the packet for which the first negative acknowledgment is received.

    Else continue as usual.
12. Increment the frame count and repeat steps 7 to 12 until all packets are transmitted.
13. Close the connection.
14. Stop.

Client

1. Start.

2. Establish a connection. (Recommended UDP).

3. Send the windowsize on server request.

4. Accept the details from the server (total packets, total frames)

5. Initialize the receive buffer with the expected packets.

6. Accept the frame / window from the server.

7. Check the validity of the packet and construct the acknowledgment frame depending on the validity. (Here the acknowledgment is accepted from the users)

8. Depending on the acknowledgment frame readjust the process.

9. Increment the frame count and repeat steps 5-9 until all packets are received.

10. Close the connection

11. Stop.

Code

```
/***************************/
/* SELECTIVE REJECT CLIENT */
/***************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#define PORT 3599
int main()
{
      int sockfd, newSockFd, size, firstTime = 1, currentPacket, wait = 3;
      char data[100], digit[2];
      struct sockaddr_in client;
      memset(&client, 0, sizeof(client));
      sockfd = socket(AF_INET, SOCK_STREAM, 0);
      if(sockfd == -1)
      {
            printf("Error in socket creation...");
      }
      else
      {
            printf("\nSocket Created...");
      }
      client.sin_family = AF_INET;
      client.sin_port = PORT;
      client.sin_addr.s_addr = inet_addr("127.0.0.1");
      printf("\nStarting up...");
      size = sizeof(client);
      printf("\nEstablishing Connection...");
      if(connect(sockfd, (struct sockaddr *)&client, size) == -1)
```

```
{
      printf("\nError in connecting to server...");
      exit(1);
}
else
{
      printf("\nConnection Established!");
}
memset(&data, 0, sizeof(data));
sprintf(data, "REQUEST");
if(send(sockfd, data, strlen(data), 0) == -1)
{
      printf("Error in sending request for data...");
      exit(1);
}
do
{
      memset(&data, 0, sizeof(data));
      recv(sockfd, data, 100, 0);
      currentPacket = atoi(data);
      printf("\nGot packet: %d", currentPacket);
      if(currentPacket == 3 && firstTime)
      {
            printf("\n*** Simulation: Packet data corrupted or
            incomplete.");
            printf("\n*** Sending RETRANSMIT.");
            memset(&data, 0, sizeof(data));
            sprintf(data, "R3");
            if(send(sockfd, data, strlen(data), 0) == -1)
            {
                  printf("\nError in sending RETRANSMIT...");
                  exit(1);
            }
            firstTime = 0;
      }
      else
```

```
            {
                  wait--;
                  if(!wait)
                  {
                        printf("\n*** Packet Accepted -> Sending ACK");
                        wait = 3;
                        memset(&data, 0, sizeof(data));
                        sprintf(data, "A");
                        digit[0] = (char)(currentPacket + 48);
                        digit[1] = '\0';
                        strcat(data, digit);
                        send(sockfd, data, strlen(data), 0);
                  }
            }
      } while(currentPacket != 9);
      printf("\nAll packets recieved... Exiting.");
      close(sockfd);
      return(0);
}


/****************************/
/*  SELECTIVE REJECT SERVER */
/****************************/

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

#define PORT 3599
```

```
void itoa(int number, char numberString[]) {
  numberString[0] = (char)(number + 48);
  numberString[1] = '\0';
}

int main() {

  int sockfd, newSockFd, size, windowStart = 1, windowCurrent = 1,
windowEnd = 4, oldWindowStart, flag;
  char buffer[100];
  socklen_t len;

  struct sockaddr_in server, client;

  memset(&server, 0, sizeof(server));
  memset(&client, 0, sizeof(client));

  if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    printf("\nError in socket creation...");
    exit(1);
  }
  else {
    printf("\nSocket created successfully...");
  }

  server.sin_family = AF_INET;
  server.sin_port = PORT;
  server.sin_addr.s_addr = INADDR_ANY;

  printf("\nStarting up...");

  if(bind(sockfd, (struct sockaddr *)&server, sizeof(server)) == -1) {
    printf("\nBinding Error...");
    exit(1);
  }
  else {
    printf("\nBinding completed successfully. Waiting for
connection..");
  }

  len = sizeof(client);

  if(listen(sockfd, 20) != -1) {

    if((newSockFd = accept(sockfd, (struct sockaddr *)&client,
&len)) == -1) {
      printf("Error in accepting connection...");
      exit(1);
    }



    memset(&buffer, 0, sizeof(buffer));
    if(recv(newSockFd, buffer, 100, 0) == -1) {
      printf("\n Recieve Error! Exiting...");
      exit(1);
    }

    fcntl(newSockFd, F_SETFL, O_NONBLOCK);

    printf("\nRecieved a request from client. Sending packets one
by one...");
    do {
      if(windowCurrent != windowEnd) {
      memset(&buffer, 0, sizeof(buffer));
      itoa(windowCurrent, buffer);
      send(newSockFd, buffer, 100, 0);
      printf("\nPacket Sent: %d", windowCurrent);
      windowCurrent++;
      }

      /*DEBUG*/ printf("\n**%d||%d**", windowCurrent,
windowEnd);
```

```
      memset(&buffer, '\0', sizeof(buffer));
      if(recv(newSockFd, buffer, 100, 0) != -1) {
        if(buffer[0] == 'R') {
          printf("\n** Received a RETRANSMIT packet.
Resending packet no. %c...", buffer[1]);
          itoa((atoi(&buffer[1])), buffer);
          send(newSockFd, buffer, 100, 0);
        }
        else if(buffer[0] == 'A') {
          oldWindowStart = windowStart;
          windowStart = atoi(&buffer[1]) + 1;
          windowEnd += (windowStart - oldWindowStart);
          printf("\n** Recieved ACK %c. Moving window
boundary.", buffer[1]);
        }
      }

      sleep(1);
    } while(windowCurrent != 10);
  }
  else {
    printf("\nError in listening...");
    exit(1);
  }

  close(sockfd);
  close(newSockFd);
  printf("\nSending Complete. Sockets closed. Exiting...\n");

  return(0);
}
```

Sample Input and Output

```
[root@Linux smtp]#gcc –o selrejc selrejc.c

[root@Linux smtp]#gcc –o selrejs selrejs.c

[root@Linux smtp]#./selrejs

[root@Linux smtp]#./selrejc
```

Server

```
Socket created successfully...
Starting up...
Binding completed successfully. Waiting for connection..
Recieved a request from client. Sending packets one by one...
Packet Sent: 1
**2||4**
Packet Sent: 2
**3||4**
Packet Sent: 3
**4||4**
**4||4**
** Received a RETRANSMIT packet. Resending packet no. 3...
**4||4**
** Recieved ACK 3. Moving window boundary.
Packet Sent: 4
**5||7**
Packet Sent: 5
**6||7**
Packet Sent: 6
**7||7**
**7||7**
** Recieved ACK 6. Moving window boundary.
Packet Sent: 7
**8||10**
Packet Sent: 8
**9||10**
Packet Sent: 9
```

```
**10||10**
Sending Complete. Sockets closed. Exiting...
```

Client

```
Socket Created...
Starting up...
Establishing Connection...
Connection Established!
Got packet: 1
Got packet: 2
Got packet: 3
*** Simulation: Packet data corrupted or incomplete.
*** Sending RETRANSMIT.
Got packet: 3
*** Packet Accepted -> Sending ACK
Got packet: 4
Got packet: 5
Got packet: 6
*** Packet Accepted -> Sending ACK
Got packet: 7
Got packet: 8
Got packet: 9
*** Packet Accepted -> Sending ACK
All packets recieved... Exiting.
```

11. SMTP Using UDP

Problem Description

Develop a program to simulate SMTP using UDP connection. The input to the program from the client side is the sender's and receiver's email id, which is of the form user@url.com. The content of the mail is also accepted from the user. The server validates the domain and the URL from which the mail is sent and displays the content of the mail.

Data Structures and Functions

The data structures used are:

- A structure named sockaddr_in which can store the required information about a client or a server.
- A character array for storing the message to be communicated.

The functions used are:

- socket () – used to create a socket for communication.
- sendto () – used to send data.
- recvfrom () – used to receive data.
- bind () – used to bind the socket with the sockaddr_in object.

Algorithm

Client Side

1. Start
2. Create Socket
3. Fill up Socket Address
4. Get ‘Helo’ from user and send to server via the socket
5. Get other details such as destination email address, from email address and message body from the client and send to server via the socket
6. Receive acknowledgment from client
7. Stop

Server Side

1. Start
2. Create Socket
3. Fill up the socket address
4. Bind socket to port
5. Wait for ‘Helo’ from client and acknowledge it back
6. Receive other details from the client using recvfrom () function
7. Acknowledge back to server using sendto () function
8. Stop

Code

Smtpclient

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

int main()
{
      char buf[100];
      int k;
      int sock_desc;
      struct sockaddr_in client;

      memset(&client,0,sizeof(client));

      sock_desc=socket(AF_INET,SOCK_DGRAM,0);
      if(sock_desc==-1)
      {
            printf("\nError in socket creation");
            exit(1);
      }

      client.sin_family=AF_INET;
      client.sin_addr.s_addr=inet_addr("127.0.0.1");
      client.sin_port=3500;

      printf("\nMAIL TO : ");
      gets(buf);
      k=sendto(sock_desc,buf,sizeof(buf),0,(struct
sockaddr*)&client,sizeof(client));
      if(k==-1)
      {
            printf("\nError in sending");
            exit(1);
      }
      strcpy(buf,"\0");
      printf("\nFROM : ");
      gets(buf);
      k=sendto(sock_desc,buf,sizeof(buf),0,(struct
sockaddr*)&client,sizeof(client));
      if(k==-1)
      {
            printf("\nError in sending");
            exit(1);
      }

      strcpy(buf,"\0");
      printf("\nSUBJECT : ");
      gets(buf);
      k=sendto(sock_desc,buf,sizeof(buf),0,(struct
sockaddr*)&client,sizeof(client));
      if(k==-1)
      {
            printf("\nError in sending");
            exit(1);
      }
      strcpy(buf,"\0");
      printf("\nMSG BODY : ");
      while(strcmp(buf,".")!=0)
      {
            strcpy(buf,"\0");
            gets(buf);
            k=sendto(sock_desc,buf,sizeof(buf),0,(struct
sockaddr*)&client,sizeof(client));
            if(k==-1)
            {
                  printf("\nError in sending");
                  exit(1);
            }     }
      close(sock_desc);
      return 0;
}
```

Smtpserver

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

int main()
{
      char buf[100],domain[20],snd[25];
      int k,i=0,j,m=0;
      socklen_t len;
      int sock_desc,temp_sock_desc;
      struct sockaddr_in server,client;

      memset(&server,0,sizeof(server));
      memset(&client,0,sizeof(client));

      sock_desc=socket(AF_INET,SOCK_DGRAM,0);
      if(sock_desc==-1)
      {
            printf("\nError in socket creation");
            exit(1);
      }

      server.sin_family=AF_INET;
      server.sin_addr.s_addr=INADDR_ANY;
      server.sin_port=3500;


      k=bind(sock_desc,(struct sockaddr*)&server,sizeof(server));
      if(k==-1)
      {
            printf("\nError in binding");
```

```
		exit(1);
	}



	len=sizeof(server);
	  k=recvfrom(sock_desc,buf,sizeof(buf),0,(struct 
sockaddr*)&server,&len);
	if(k==-1)
	{
		printf("\nError in receiving");
		exit(1);
	}
	strcpy(snd,buf);
	while(i<(strlen(buf)))
	{
		if(buf[i]=='@')
		{
			for(j=i+1;j<strlen(buf);j++)
			domain[m++]=buf[j];
			break;
		}
		i++;
	}
	printf("Receiving Mail...");
	printf("\nDomain verified << %s >>",domain);

	len=sizeof(server);
	  k=recvfrom(sock_desc,buf,sizeof(buf),0,(struct 
sockaddr*)&server,&len);
	  if(k==-1)
	  {
		  printf("\nError in receiving");
		  exit(1);
	  }
	  printf("\nFROM: %s\n",buf);
```

```
      len=sizeof(server);
        k=recvfrom(sock_desc,buf,sizeof(buf),0,(struct
sockaddr*)&server,&len);
        if(k==-1)
        {
                printf("\nError in receiving");
                exit(1);
        }
        printf("\nSUBJECT: %s\n",buf);

      printf("\nMSG BODY: \n\t");
      len=sizeof(server);
        k=recvfrom(sock_desc,buf,sizeof(buf),0,(struct
sockaddr*)&server,&len);
        if(k==-1)
        {
                printf("\nError in receiving");
                exit(1);
        }
      while(strcmp(buf,".")!=0)
      {
            printf("%s\n\t",buf);
            len=sizeof(server);
                k=recvfrom(sock_desc,buf,sizeof(buf),0,(struct
sockaddr*)&server,&len);
              if(k==-1)
            {
                  printf("\nError in receiving");
                      exit(1);
            }
        }
      printf("\nMail received successfully from %s\n",snd);
      close(temp_sock_desc);
      exit(0);
      return 0;
}
```

Sample Input and Output

Client

```
[root@Linux smtp]# gcc –o smtpc smtpc.c

[root@Linux smtp]# ./smtpc

Helo

Server acknowledged: 250 – Request command completed

MAIL FROM: <mail@example.com>

RCPT TO: <mail@linux.com>

MSGBODY: This is a sample email.

Sending the mail…

QUIT

Server Closed Successfully
```

Server

```
[root@Linux smtp]# gcc –o smtps smtps.c

[root@Linux smtp]# ./smtps

Client Send a Helo

Sending Helo Reply

MAIL FROM: mail@example.com

RCPT TO: mail@linux.com <<Domain Verified>>

MSGBODY: This is a sample email.

Received QUIT message from client.

Server Shutting Down
```

12. FTP using TCP

Problem Description

Creation of a simple simulator for performing FTP operations of LIST, LOAD, STORE where the LIST operation gets the list of files present in the server and displays it in the client .The LOAD operation downloads a particular file from the server to the client and the STORE operation uploads a file from the client to the server.

Data Structures and Functions

Data Structures used here are:

A structure called stat which contains the status of various elements.

struct stat obj;   obj st_size;

A header file <sys/stat.h> is required for this structure.

Functions used here are:

- sendfile (): This function is used to send a file between server and client.

  syntax: sendfile (temp_descr, filehandle, NULL, obj.st_size)

  A header file <sys/sendfile.h> is required for this function.

- system ()

  This function is used to initiate terminal command from inside a program.

  eg: system(“cat list.txt”)

Algorithm

Client

For LIST operation:

1. Open a file ‘list2.txt ‘in the read / write or create mode

2. Create a socket and notify the server on which operation is to be performed by using the send command

3. From the server the size of the file as well as the file is received in the variables ‘length’ and ‘fil’ respectively.

4. Write the received details into the filehandle which is the descriptor for the opened file list2.txt

5. Now write the contents into list2.txt

For LOAD operation:

1. Open a file ‘list3.txt ‘in the read / write or create mode

2. Create a socket and notify the server that LOAD is to be performed by using the send command. Along with it send the name of the file to be downloaded.

3. From the server the size of the file as well as the file is received in the variables ‘length’ and ‘fil’ respectively

4. Write the received details into the filehandle which is the descriptor for the opened file list3.txt.

5. Now write the contents into list3.txt

Server

For LIST operation:

1. Create a socket and bind to the port which is given as input from the user

2. Listen and accept the request from the client.

3. For the LIST operation using the command system("ls -al>list.txt") get the details of all the files along with its attributes into the file named list.txt.

4. Get the size of the file in blocks using the commands:

```
stat ("list.txt",&obj);

sprintf (length,"%d",(int)obj.st_size);
```

5. Send the size to the client.

6. Using the function sendfile () send the contents of list.txt to the client

For LOAD operation:

1. For the LOAD operation the filename to be downloaded is specified by the client.

2. Get the size of the file in blocks using the commands:

```
stat (filename,&obj);

sprintf (length,"%d",(int)obj.st_size);
```

3. Send the size to the client.

4. Using the function sendfile () send the contents to the client

Code

Server

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include<sys/stat.h>
#include<sys/sendfile.h>
#include <fcntl.h>

int main(int argc,char *argv[])
{
      struct sockaddr_in address1,address2;
      struct stat obj;
      int sock_descr,temp_descr;
      socklen_t len;
      char buf[100],choice[10],length[100],filename[100];
      int k,i;
      int filehandle;

      memset(&address1,0,sizeof(address1));
      memset(&address2,0,sizeof(address2));

      sock_descr=socket(AF_INET,SOCK_STREAM,0);
      if(sock_descr==-1)
      {
            printf("socket creation failed");
            exit(1);
      }

      address1.sin_family=AF_INET;
      address1.sin_port=atoi(argv[1]);
      address1.sin_addr.s_addr=INADDR_ANY;//inet_addr(argv[1]);

      k=bind(sock_descr,(struct sockaddr*)&address1,sizeof(address1));
      if(k==-1)
      {
            printf("binding error");
            exit(1);
      }

      k=listen(sock_descr,5);
      if(k==-1)
      {
            printf("listen failed");
            exit(1);
      }

      len=sizeof(address2);
      temp_descr=accept(sock_descr,(struct sockaddr*)&address2,&len);
```

```
if(temp_descr==-1)
{
      printf("temp: socket creation failed");
      exit(1);
}

while(1)
{
      k=recv(temp_descr,buf,100,0);
      if(k==-1)
      {
            printf("receive failed");
            exit(1);
      }

      for(i=0;i<4;i++)
      choice[i]=buf[i];
      choice[4]='\0';
      printf("\n%s",choice);
      if(strcmp(choice,"LIST")==0)
      {
            system("ls -al>list.txt");
            filehandle=open("list.txt",O_RDONLY);//
            stat("list.txt",&obj);//
            sprintf(length,"%d",(int)obj.st_size);
            k=send(temp_descr,length,strlen(length),0);
            if(k==-1)
            {
                  printf("send failed");
                  exit(1);
            }
            k=sendfile(temp_descr,filehandle,NULL,obj.st_size);
            if(k==-1)
            {
                  printf("file sendingfailed");
                  exit(1);
            }
      }
      if(strcmp(choice,"LOAD")==0)
      {
            strcpy(filename,buf+4);
            stat(filename,&obj);
            filehandle=open(filename,O_RDONLY);
            if(filehandle==-1)
            {
                  printf("NO SUCH FILE\n");
                  exit(1);
            }
            sprintf(length,"%d",(int)obj.st_size);
            printf("\n%s\n",length);
            k=send(temp_descr,length,strlen(length),0);
            if(k==-1)
            {
                  printf("send failed");
                  exit(1);
                  }
                  sendfile(temp_descr,filehandle,NULL,obj.st_size);
                  if(k==-1)
                  {
                        printf("file sendingfailed");
                        exit(1);
                  }
            }
      }
return 0;
}
```

Client

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include<sys/stat.h>
#include<sys/sendfile.h>
#include <fcntl.h>
int main(int arc,char *argv[])
{
      struct sockaddr_in address;
      int sock_descr;
      int k,choice,len,l,c=0,filehandle;
      char buf[100],length[100],fil[100],filename[100];
      memset(&address,0,sizeof(address));
      sock_descr=socket(AF_INET,SOCK_STREAM,0);
      if(sock_descr==-1)
      {
            printf("socket creation failed");
            exit(1);
      }
      address.sin_family=AF_INET;
      address.sin_port=atoi(argv[1]);
      address.sin_addr.s_addr=INADDR_ANY;
      k=connect(sock_descr,(struct sockaddr*)&address,sizeof(address));
      if(k==-1)
      {
            printf("connecting failed");
            exit(1);
      }
      while(1)
      {
            printf("\n1:LIST\n2:LOAD\n3:Exit : ");
            scanf("%d",&choice);
            switch(choice)
            {
                  case 1:
                        filehandle=open("list2.txt",O_RDWR|O_CREAT,0700);
                        strcpy(buf,"LIST");
                        k=send(sock_descr,buf,strlen(buf),0);
                        if(k==-1)
                        {
                              printf("send failed");
                              exit(1);
                        }
                        k=recv(sock_descr,length,100,0);
                        if(k==-1)
                        {
                              printf("receive failed");
                              exit(1);
                        }
                        len=atoi(length);
                        while(c<len)
                        {
                              l=read(sock_descr,fil,100);
                              if(l==0)
                                    break;
                              write(filehandle,fil,100);
                              c+=l;
                        }
                        system("cat list2.txt");
                        close(filehandle);
                        break;
                  case 2:
                        filehandle=open("list3.txt",O_RDWR|O_CREAT,0700);
                        strcpy(buf,"LOAD");
                        printf("Enter filename : ");
                        scanf("%s",filename);
                        strcat(buf,filename);
                        k=send(sock_descr,buf,strlen(buf),0);
                        if(k==-1)
                        {
                              printf("send failed");
                              exit(1);
                        }
                        k=recv(sock_descr,length,100,0);
                        if(k==-1)
                        {
                              printf("receive failed");
                              exit(1);
                        }
                        len=atoi(length);
                        c=0;
                        while(c<len)
                        {
                              l=read(sock_descr,fil,strlen(fil));
                              if(l==0)
                                    break;
                              write(filehandle,fil,strlen(fil));
                              c+=l;
                        }
                        system("cat list3.txt");
                        break;
                  case 3:
```

```
                        exit(0);
                  default:
                        break;
            }
      }
      return 0;
}
```

Sample Input and Output

```
[root@Linux smtp]#gcc –o ftpclient ftpclient.c
[root@Linux smtp]#gcc –o ftpserver ftpserver.c
[root@Linux smtp]#./ftpserver 4000
[root@Linux smtp]#./ftpclient 4000
LIST
-rwx------ 1 sample sample 6874 2009-05-09 17:50 1bitclient
-rwx------ 1 sample sample 2396 2009-05-09 17:50 1bitclient.c
-rwx------ 1 sample sample 6833 2009-05-09 17:50 1bitserver
-rwx------ 1 sample sample 2653 2009-05-09 17:50 1bitserver.c
-rwx------ 1 sample sample 6874 2009-05-09 17:50 1c
-rwx------ 1 sample sample 6833 2009-05-09 17:50 1s
-rwxr-xr-x 1 sample sample 9417 2009-05-09 17:52 a.out
-rwx------ 1 sample sample 7804 2009-05-09 17:50 ban
-rwx------ 1 sample sample 7804 2009-05-09 17:50 bank
-rwx------ 1 sample sample 1533 2009-05-09 17:50 bankers.c
-rwx------ 1 sample sample 7804 2009-05-09 17:50 bankexit
-rwxr-xr-x 1 sample sample 9117 2009-05-09 17:52 c
LOAD bankers.c
The chosen file has been downloaded.
STORE example.txt
The chosen file has been uploaded
```

G. Virtual File System

The main data item in any Unix-like system is the ``file'', and a unique pathname identifies each file within a running system. Every file appears like any other file in the way is accessed and modified: the same system calls and the same user commands apply to every file. This applies independently of both the physical medium that holds information and the way information is laid out on the medium. Abstraction from the physical storage of information is accomplished by dispatching data transfer to different device drivers; abstraction from the information layout is obtained in Linux through the VFS implementation.

The Unix way

Linux looks at its file-system in the way UNIX does: it adopts the concepts of super-block, inode, directory and file in the way Unix uses them. The tree of files that can be accessed at any time is determined by how the different parts are assembled together, each part being a partition of the hard drive or another physical storage device that is ``mounted'' to the system.

- The super-block owes its name to its historical heritage, from when the first data block of a disk or partition was used to hold meta-information about the partition itself. The super-block is now detached from the concept of data block, but still is the data structure that holds information about each mounted file-system. The actual data structure in Linux is called struct super_block and hosts various housekeeping information, like mount flags, mount time and device blocksize. The 2.0 kernel keeps a static array of such structures to handle up to 64 mounted file-systems.
- An inode is associated to each file. Such an ``index node'' encloses all the information about a named file except its name and its actual data. The

owner, group, permissions and access times for a file are stored in its inode, as well as the size of the data it holds, the number of links and other information. The idea of detaching file information from filename and data is what allows to implement hard-links -- and to use the `dot' and `dot-dot' notations for directories without any need to treat them specially. An inode is described in the kernel by a struct inode.

- The directory is a file that associates inodes to filenames. The kernel has no special data structure to represent a directory, which is treated like a normal file in most situations. Functions specific to each filesystem-type are used to read and modify the contents of a directory independently of the actual layout of its data.
- The file itself is something that is associated to an inode. Usually files are data areas, but they can also be directories, devices, FIFO's or sockets. An ``open file'' is described in the Linux kernel by a struct file item; the structure encloses a pointer to the inode representing the file. File structures are created by system calls like open, pipe and socket, and are shared by father and child across fork.

13. Disk Status Usage Report

Problem Description

Develop a program to find the status information of a file system

Data Structures and Functions

Data Structures used here are:

A structure called statvfs: contains detailed file system information.

The implementation makes use of the following fields in statvfs

f_bsize: File system block size.

f_blocks: Total number of blocks on the file system

f_bfree: Total number of free blocks.

Functions used here are:

- statvfs ()

This function obtains information about the file system containing the file referred to by the specified path name.

syntax: int statvfs(const char *path, struct statvfs *buf);

Algorithm

1. Define an object Data of the structure statvfs.
2. The path of the filename whose status is to be found is stored in variable Path.
3. Using the function statvfs() status of the file system is obtained in the object data.
4. Now the required status info can be retrieved using Data.f_bsize, Data.f_blocks etc.
5. Display the block size, free blocks size and used blocks in MB.
6. Stop

Code

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

int main( int argc, char* argv[] )
{
        struct statvfs Data;
        char Path[128];
        int i;

        if( argc < 2 ) {
                printf("Usage, <executable> DEVICE0 ..... DEVICEX\n");
                return(2);
        }

        for( i = 1 ; i<argc; i++ ) {
                strcpy(Path, argv[i]);
                if((statvfs(Path,&Data)) < 0 ) {
                        printf("Failed to stat %s:\n", Path);
                } else {
                        printf("Disk %s: \n", Path);
                        printf("\tblock size: %u\n", Data.f_bsize);
                        printf("\ttotal no blocks: %i\n", Data.f_blocks);
                        printf("\tfree blocks: %i\n", Data.f_bfree);
                }
        }
}
```

Sample Input and Output

```
[root@Linux smtp]#gcc –o du du.c

[root@Linux smtp]#./du /root

Disk /root:

      block size: 4096

      total no blocks: 998153

      free blocks: 420875
```

H. Remote Procedure Call

What Is RPC

RPC is a powerful technique for constructing distributed, client-server based applications. It is based on extending the notion of conventional, or local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them. By using RPC, programmers of distributed applications avoid the details of the interface with the network. The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports. RPC makes the client/server model of computing more powerful and easier to program. When combined with the ONC RPCGEN protocol compiler, clients transparently make remote calls through a local procedure interface.

How RPC Works

An RPC is analogous to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure. Figure shows the flow of activity that takes place during an RPC call between two networked systems. The client makes a procedure call that sends a request to the server and waits. The thread is blocked from processing until either a reply is received, or it times out. When the request arrives, the server calls a dispatch routine that performs the requested service, and sends the reply to the client. After the RPC call is completed, the client program continues. RPC specifically supports network applications.

A remote procedure is uniquely identified by the triple: (program number, version number, procedure number) The program number identifies a group of related remote procedures, each of which has a unique procedure number. A program may consist of one or more versions. Each version consists of a collection of procedures which are available to be called remotely.

Defining the Protocol

The easiest way to define and generate the protocol is to use a protocol complier such as rpcgen. For the protocol you must identify the name of the service procedures, and data types of parameters and return arguments.The protocol compiler reads a definition and automatically generates client and server stubs. rpcgen uses its own language (RPC language or RPCL) which looks very similar to preprocessor directives. rpcgen exists as a standalone executable compiler that reads special files denoted by

a .x prefix.

So to compile a RPCL file you simply do rpcgen rpcprog.x

This will generate possibly four files:

- rpcprog_clnt.c -- the client stub
- rpcprog_svc.c -- the server stub
- rpcprog_xdr.c -- If necessary XDR (external data representation) filters
- rpcprog.h -- the header file needed for any XDR filters.

The external data representation (XDR) is an data abstraction needed for machine independent communication. The client and server need not be machines of the same type.

External Data Representation

XDR is a standard for the description and encoding of data. It is useful for transferring data between different computer architectures, and has been used to communicate data between such diverse machines as the SUN WORKSTATION*, VAX*, IBM-PC*, and Cray*.

XDR uses a language to describe data formats. The language can only be used only to describe data; it is not a programming language. This language allows one to describe intricate data formats in a concise manner. The alternative of using graphical representations (itself an informal language) quickly becomes incomprehensible when faced with complexity. The XDR language itself is similar to the C language, just as Courier [4] is similar to Mesa. Protocols such as ONC RPC (Remote Procedure Call) and the NFS* (Network File System) use XDR to describe the format of their data.

The XDR standard makes the following assumption: that bytes (or octets) are portable, where a byte is defined to be 8 bits of data. A given hardware device should encode the bytes onto the various media in such a way that other hardware devices may decode the bytes without loss of meaning. For example, the Ethernet* standard suggests that bytes be encoded in "little-endian" style [2], or least significant bit first.

What is rpcgen

The rpcgen tool generates remote program interface modules. It compiles source code written in the RPC Language. RPC Language is similar in syntax and structure to C. rpcgen produces one or more C language source modules, which are then compiled by a C compiler. The default output of rpcgen is:

- A header file of definitions common to the server and the client

- A set of XDR routines that translate each data type defined in the header file
- A stub program for the server
- A stub program for the client

rpcgen can optionally generate:

- Various transports
- A time-out for servers
- Server stubs that are MT safe
- Server stubs that are not main programs
- C-style arguments passing ANSI C-compliant code
- An RPC dispatch table that checks authorizations and invokes service routines.

rpcgen significantly reduces the development time that would otherwise be spent developing low-level routines. Handwritten routines link easily with the rpcgen output. The “finger.x” file is passed to rpcgen which is given by:

```
struct finger_out{
  char message[1024];
};
program FINGER
{
        version FINGER_VERSION
        {
                finger_out MyFinger() = 1;
        } = 1;
} = 0x21230000;
```

14. Finger Utility using RPC

Problem Description

Develop a program to implement the finger utility using Remote Procedure Call (RPC).

Data Structures and Functions

RPCGEN

Rpcgen-an RPC protocol compiler

Rpcgen is a tool that generates C code to implement an RPC protocol. The input to rpcgen is a language similar to C known as, rpc language.

Algorithm

1. Start
2. Client makes a TCP connection to the server (default port 79) and transfers parameters.
3. If input parameter is {c} only, print list of all users online and some additional information like
   - terminal location
   - home location
   - idle time [no. of minutes since last typed input or since last job activity]
4. If input parameter is {u}{c}, only print above mentioned information of a specified user {u} along with the following information:
   - login information
   - amount of time after being logged in
   - new mail information
5. If input is {u}{H}{c} where {h} is host H, then transfer the finger request to the computer on behalf of this host.
6. Stop

Code

```
struct finger_out{
  char message[1024];
};
program FINGER{
  version FINGER_VERSION{
    finger_out MyFinger() = 1;
  } = 1;
} = 0x21230000;
```

Client

```
#include <rpc/rpc.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "finger.h"
void err_quit(char *msg)
{
  printf("%s\n", msg);
  exit(1);
}
int main(int argc, char* argv[])
{
  CLIENT *c1;
  finger_out *outp;
  if(argc!=2)
    err_quit("usage: client <hostname>");
  c1 = clnt_create(argv[1], FINGER, FINGER_VERSION, "tcp");
  if( (outp=myfinger_1(NULL, c1))==NULL )
    err_quit(clnt_sperror(c1, argv[1]));
  printf("result: %s\n", outp->message);
  exit(0);
}
```

Server

```
#include <rpc/rpc.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include "finger.h"
finger_out* myfinger_1_svc(void *dummy, struct svc_req *rqstp)
{
  static finger_out fo;
  char buffer[1024];
  system("finger > result.txt");
  FILE *fp = fopen("result.txt", "r");
  int i=0;
  while( !feof(fp) ){
    buffer[i++] = fgetc(fp);
  }
  buffer[i] = '\0';
  strcpy(fo.message, buffer);
  system("rm -f result.txt");
  return &fo;
}
```

SAMPLE INPUT/OUTPUT

```
      Open the executable server file
      Open the executable client file with arguments
                              ./client 127.0.0.1 5000
root @example:~/Desktop/RPC$ ./client 127.0.0.1 5000
Login      Name        Tty       Idle  Login Time   Office     Office Phone
root  root   tty7          May  9 15:52 (:0)
root  root   pts/0         May  9 16:06 (:0.0)
root @example:~/Desktop/RPC$
```