1470

REMOTE PROGRAM EXECUTION

Dissertation Submitted to
JAWAHARLAL NEHRU UNIVERSITY
in partial fulfilment of requirements
for the award of the degree of
Master of Technology
in
Computer Science & Technology

by

RAKESH KUMAR

68p

Jawaharlal Nehru University

SCHOOL OF COMPUTER & SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI - 110 067
January 1997

CERTIFICATE

This is to certify that the dissertation entitled **REMOTE PROGRAM EXECUTION** being submitted by **RAKESH KUMAR** to School of Computer and System Sciences, Jawaharlal Nehru University, New Delhi , in partial fulfilment of the requirements for the award of the degree of Master of Technology in Computer science, is a bona fide work carried by him under the guidance and supervision Prof. C.P. Katti. This work has not been submitted elsewhere for any other purpose.

Dean ,SCSS

[illegible]

(Prof.G.V.Singh)
SC&SS,J.N.U.
New Delhi 110067

C P Katti

Supervisor

6/1/97

(Prof. C.P.Katti)
SC&SS,J.N.U.
New Delhi 110067

ACKNOWLEDGEMENT

I wish to express my profound sense of gratitude to Prof. C.P.Katti, SC&SS, JNU, under whose invaluable guidance and incessant encouragement, my work has taken its present shape.

I wish to express my sincere thanks to the staff members of the computer lab, SC&SS for providing me with all the facilities required during the project.

Finally I would like to thank all my friends especially Mr Arun Kumar and Mr Gaurav Maheshwari without whose support it may have been difficult to complete the project.

Rakesh kumar

(RAKESH KUMAR)

Contents

Chapter One

INTRODUCTION

- Overview
- Objective
- Theoretical Background
- Remote Program Execution and its relevance
- Organization of The Dissertation

1.1. Overview :

The main motivation for constructing computer networks is data sharing and resource sharing among computers. The resource sharing allows a computer to use other computer's resources such as files, printers etc. Also the concept of virtual terminals come in where one computer can login on other computer and then act as a terminal to the other computer getting input and writing output for the programs running on the other computer. So this system allows to run a program in the other computer's processor and memory space and redirecting input and output to the first machine. Various protocols such as TELNET, rlogin and rsh exists for such type of resource sharing. These protocols are designed for certain environment and suits them. But in some conditions they are not very efficient. For example, when there is a small networking environment with much faster communication protocol , the use of TCP/IP based protocol is not suitable. TCP/IP based protocol are most suitable for UNIX environment in which the operating system supports them and also for larger area network where the packets transfers through a variety of network and packet transmission requires routing. Also these protocols are designed to be used for heterogeneous systems. So they do not provide the facility to access the local files. It means that the running applications in these system can not open and close local files and also we can not run local files on the remote machine without transferring it to the remote machine with the help of some other file transferring protocols. Also the

TCP/IP carries more overhead than the protocols designed for smaller area networks.

1.2. Objective :

The objective of this project is to develop a system by which programs can be executed on remote machines. These programs may be interactive as well as non interactive. By saying interactive it means that running remote programs take the input and output from the user who launched the program. The input and output required by the program is redirected to the launching terminal.

The platform for which the system has been developed uses MS DOS as the operating system on both the machines (versions 4.0 and above), NetBIOS for the communication purposes, and Ethernet as the underlying network.

As the underlying operating system will be DOS on both the machines, the system also provides an option to make the local hard disk available for the running programs. So the running programs will be able to open the local files and use them as they are present on the remote system itself. This mechanism also provides the flexibility to run the local programs i.e., programs present on the local drive on the remote machines. In this manner one can take the advantage of faster processors and better resources on the remote site for running his local programs. For this purpose the system on being invoked provides one more drive letter to represent the local drive from which the programs are being launched in the system.

1.3. Theoretical Background :

Many network systems provide computer with the ability to run programs on the remote machines. Also various methods for accessing files on remote machines has been devised. We take a look at the relevant virtual terminal protocols and file accessing protocols for distinguishing their main features.

1.3.1. TELNET :

TELNET is the ARPANET virtual terminal protocol. It allows a user at one site to establish a TCP connection to a login server at another and then it passes the keystrokes from the user's terminal directly to the remote machine as if they were typed at a terminal on the remote machine. TELNET also carries the output from the remote machine back to the user's terminal. The service is called transparent because it gives the appearance that the user's terminal is directly attached to the remote machine.

TELNET offers three basic services. First it defines a network virtual terminal that provides a standard interface to the remote systems. Client programs do not have to understand the details of all possible remote systems, they are built to use the standard interface.

Second, it has a mechanism that allows the client and server to negotiate options and it provides a set of standard options. One of these control the character set used for data communication.

Finally, TELNET treats both ends of the connection symmetrically. So instead of forcing the client side to connect to a user's terminal, TELNET allows an arbitrary program to become a client. Furthermore, either end can negotiate options.

When a user invokes TELNET, an application program on the user machine becomes the client. The client establishes a TCP connection to the server over which they will communicate. Once the connection has been established, the client accepts keystrokes from the users terminal and sends them to the server, while concurrently accepts characters that the server sends back and displays them on the user's terminal. The server must accept a TCP connection from the client and then relay data between the TCP connection and the local operating system.

TELNET allows control functions to be passed along with the normal data. But it does not guarantee the desired results. For example, a process running at the remote machine may start ill behaving without reading the input passed to it, by going into an infinite loop. So, the control function will be waiting the buffer without being interpreted. Also the buffer may become full in the local buffer of the server, and the TCP/IP will stop receiving the data from the client machine. So TELNET can not rely on the convention data stream alone to carry the control sequences between the client and the server, because a misbehaving application may block the data stream making the controls to wait infinitely in the buffer. So TELNET uses an out of band data signal by sending the SYNCH command. TCP sends the server a segment with urgent data bit set and this bypasses the data stream and reaches the server which act by rejecting all the data stream before the control signal and acts on it.

1.3.2. RLOGIN :

4 BSD UNIX system includes a remote login service rlogin which supports the trusted hosts to run program on the system. The system works as described. The terminal line discipline on the client machine is placed into the raw mode with echoing disabled, so that all keystrokes are passed to the remote system. The raw mode is required to run programs such as vi editor on the remote system. Characters that are entered on the local system are echoed by the remote system. If the remote system is in a cooked mode then echoing is done by the terminal line discipline on the remote machine. If the remote system is in a raw mode then echoing is done by the remote process itself.

So it is similar to the TELNET in working. But it provides some other facilities. It allows system administrator to choose a set of machines over which login names and file access protections are shared. It means that the user having account on two machines can access the other machine from the first one by giving only the login name only.

One variation of rlogin is rsh which invokes a remote command interpreter and passes the command line argument to the command interpreter, skipping the login step completely.

The standard input and the standard output are connected across the network to the user's terminal. rlogin protocol understands both the local and the remote computing environments, it communicate better than the general purpose remote login protocols like TELNET. So output from a remote command can be redirected to a file by giving a command a like :

```
rsh command_name > output_filename
```

where command_name is the name of the program being invokes on the remote system and the output_filename is the name of the file to which the output is redirected. ('>' is used redirect the output to a file in the UNIX system).

These protocol also understand the terminal control functions like flow control characters like CTRL+S and CTRL+Q and arranges to handle it immediately without waiting for the delay required to send them across the network to the remote host. It also exports part of user's environment to the remote machine, including information like user's terminal type. As a result, rlogin sessions appear to behave almost exactly like the local login sessions.

1.3.3. NFS :

It was developed by sun Microsystems, it provides on line shared file access that is transparent and integrated. Many TCP/IP sites use NFS to interconnect their computer file system. From a user's perspective, NFS is almost invisible. The user can execute an arbitrary application program and use arbitrary files for input and output.

NFS acts as part of operating system. When an application programs execute and call the operating system to open a file, store a file, read a file and other services, the file access mechanism accepts the request and after determining the location of file passes the request to the local file system or remote machine. After getting the result from the remote machine it returns the result to the application program.

NFS protocol itself does not provides that a program can call. Once it has been configured, program access remote file using exactly same operations as they use for local files.

1.4. Remote Program Execution and it's relevance :

In this project, a virtual terminal system has been developed which provides access to remote machine on a local area network. It is different from the TELNET and other protocols in the respect that they all provide remote execution on a non DOS platform, whereas this system provides remote execution on a DOS platform.

1.4.1. Main Advantages Of The System :

1. It is a DOS based remote program execution system and can be extended to the multi tasking system windows.

2. It is suited for smaller local area network as the overhead in NetBIOS the protocol it uses is smaller. TCP/IP bases programs are better suited in wide area network needing routing information etc.

3. This program is faster than other system as the NetBIOS is faster protocol than TCP/IP .

4. By embedding itself in the operating system the system makes itself transparent to the user and make the user feel that the terminal (keyboard and display) is directly attached to the remote machine.

5. It allows a user to take advantage of another machine with more processing power and more memory.

6. This system has the advantage that the user can access his local disks in the same manner in which he can the remote drives.

7. The system also allows the user to run the local programs on his local drive on the remote machine.

1.4.2. Main Limitations Of The System :

1. This system is not suitable for larger networks where the problem of routing comes in.

2. Ill behaving DOS programs which directly to the screen memory bypassing the operating system services and the BIOS services can not be handled by this system.

3. It can not handle windows(operating system) based input and output in its present form.

4. It can not handle the illegal file input and output operations using FILE CONTROL BLOCKS.

5. Direct sector level input output is not possible to be redirected by this system.

6. Terminate and stay resident programs can not be handled by the system.

1.5. Organization of the dissertation

The dissertation is divided into seven chapters:

1. chapter I is Introduction which gives the idea about the project and describes the problem. Also it tells about the related works and the relevance of this project.

2. Chapter II is the organization of RPE system which tells about the model being used in this system.

3. Chapter III gives the introduction to the NetBIOS system which has been used by the RPE for communication between the computers.

4. Chapter IV tells about the Advanced MS DOS system calls which are important in the execution of RPE.

5. Chapter V describes the different layers designed for communication over NetBIOS for implementing RPE.

6. Chapter VI describes the design and implementation of the main programs.

7. Chapter VII contains the result obtained, conclusion and scope of further work.

There ~~is~~ are two appendices which contain the important functions and definitions related to the project.

Chapter Two

ORGANIZATION OF RPE

- The Organization
- The User Interface Provided

2.1. Organization of The System :

RPE system allows to run programs on remote machines. The system act as the terminal (keyboard and screen) is attached to the remote machine itself. The system basically consists of two modules:

1. Remote program executioner and

2. The launcher.

It is based upon the client server model. The executioner acts as the server at the time of execution of the programs. All the programs runs in the memory of the executioner's machine and the launcher acts as the client at that time displaying the input and output of the running program. But at the time, when the executioner has to access file on the disk at the launcher's machine it becomes the client and the launcher acts as a file server processing all the requests made by the executioner.

In the system, the executioner runs on the workstation which we want to act as the system where programs can be run from other machines i.e., where the programs can be executed remotely. The launcher resides on the machine from which the programs can be submitted to the executioner for execution. The launcher can call any of the executioner running on the network by calling the name which the executioner assigns to itself.

Both modules i.e., the executioner as well as the client uses the NetBIOS protocol to communicate with each other. As critical data pass through the network. for reducing the complexity two more layers are designed above NetBIOS. These layers are necessary as the application modules have to pass information and data to each other and the layering hides the complexity how the data is communicated so that the main system becomes easier to design and implement. Also this layering makes the system more reliable and robust. The layers designed are presentation layer and error detecting layer. Both use their own packet formats for conversation with the corresponding layer on the other side.

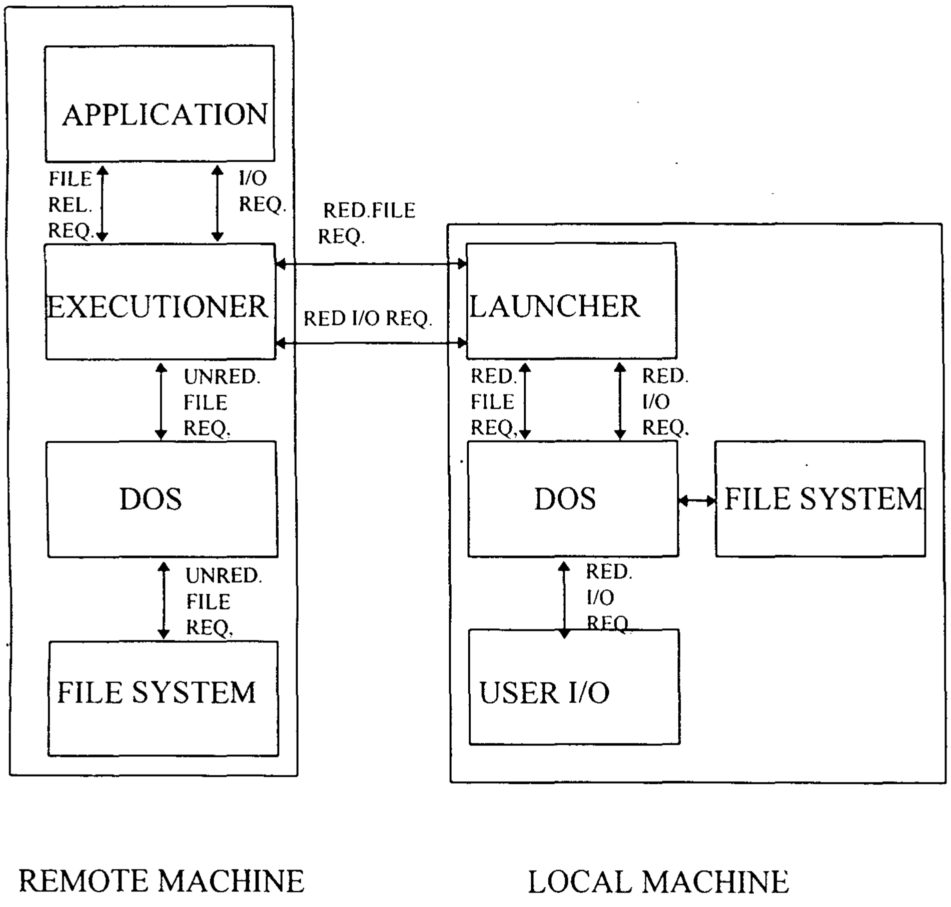


Figure 2.1 shows the model of the RPE system

The other aspects of layers and their design has been described in detail in the chapter V, the design of the layers above NetBIOS.

2.2. USER INTERFACE PROVIDED BY THE SYSTEM:

2.2.1. The Executioner :

The executioner can be run by the following command line:

```
c:\>rpe
```

After this the executioner loads itself in the memory and waits for the launcher to communicate to it. The executioner when called by the launcher to execute a program, it executes the requested program and shows on its local screen that it is busy in running an application. At other times it shows on its screen that it is free. In case the executioner is free it can be unloaded from memory by pressing the Esc key.

2.2.2. The Launcher :

The launcher can be run alone on the terminal or with some command line parameters. The command line for invoking it is:

```
c:\>launch [-t] [-name executioner_name] [-p program_name]
```

The command line options are :

-t : it says the system to run in dumb terminal mode in which the local file system is not available while running the program.

-executioner_name is the name of the executioner to which the program is to be submitted for execution.

-program_name : It is the program name which will be executed by the executioner. On giving no name, it defaults to the command.com, the DOS command interpreter on the remote system.

If the launcher is executed without any parameters on the command line it shows up a user interface which asks the user for the mode first. After this it finds all the executioners running on the network. Then it shows the user names of the executioners running and asks the user to select any of them. At the end it asks for the name of program to be executed on the executioner. The name may contain any other command line parameters as well as the pipes for redirecting standard input / output. The files used for redirection may be on the launcher's machine but they have to bear the path name indicating the same using the drive letter $:.

If the mode of execution is normal mode then before launching the program on the machine the executioner tells the user the drive letter assigned to the drive on the machine of the launcher. The user can access the drive using the drive letter assigned to it.

For launching local programs from the launcher the name of the program file to be launched must start with $:. This tells the executioner that the program is to be launched from the drive on the launcher's machine. If the command line parameters first tell the mode of execution to be dumb terminal type and gives the program name to be local then the mode of execution is set to normal mode as with out this the execution of local program will not be possible on the executioner.

After executing the program the current directory of the system is set to be the directory of the executioner. All the input and output of the executing program is redirected the launcher. If the mode of the execution is normal then the request for accessing the files on the drive of the launcher is also redirected to the launcher. In this manner the system makes the launchers computer to act as a terminal attached to the executioner's machine.

After the completion of the executed program the system determines the exit/completion code and shows it to the user. At this time the launcher exits and the executioner again starts waiting for another launcher to connect to it.

Chapter Three

INTRODUCTION TO NetBIOS

- History
- NetBIOS Services
- Network Control Block

3.1. History :

In 1984, IBM released its first LAN, the IBM PC Network. The interface card (adapter card) for the IBM PC was developed by Systek, Inc. and contained on it the first implementation of NetBIOS. The name NetBIOS is derived from the name BIOS for the 'Basic Input Output System' for the IBM PC. NetBIOS provides an interface between a program and actual interface on the network card like the BIOS provides an interface between a program on the PC and the actual hardware and contained in read only memory.

In 1985, IBM provided an implementation of NetBIOS for the token ring, as soon as it introduced its token ring LAN. The token ring version was a software module, while the previous implementation was in the read only memory on the interface card.

The third implementation of NetBIOS by IBM occurred when the IBM OS/2 system were introduced and the IBM LAN support program was available. The software package consists of device drivers and interface support for all of IBM's LAN interfaces .

NetBIOS is a software interface, not a protocol. Nevertheless the interface provided by all IBM implementations of NetBIOS are equivalent providing a consistent software interface that has become a de facto standard for the personal computers.

In addition there exist implementations of NetBIOS that use TCP and UDP as the underlying transport protocol and standards exist for this on the Internet(RFC 1001 and RFC 1002)

NetBIOS corresponds to the network layer, transport layer and session layer of the OSI Reference model. In the PC environment the application that NetBIOS is being used, is for file sharing. In this case another protocol interface is called Server Message Block (SMB).

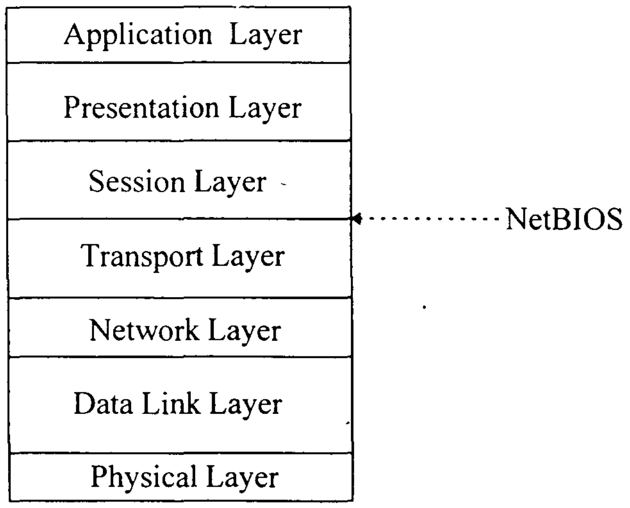


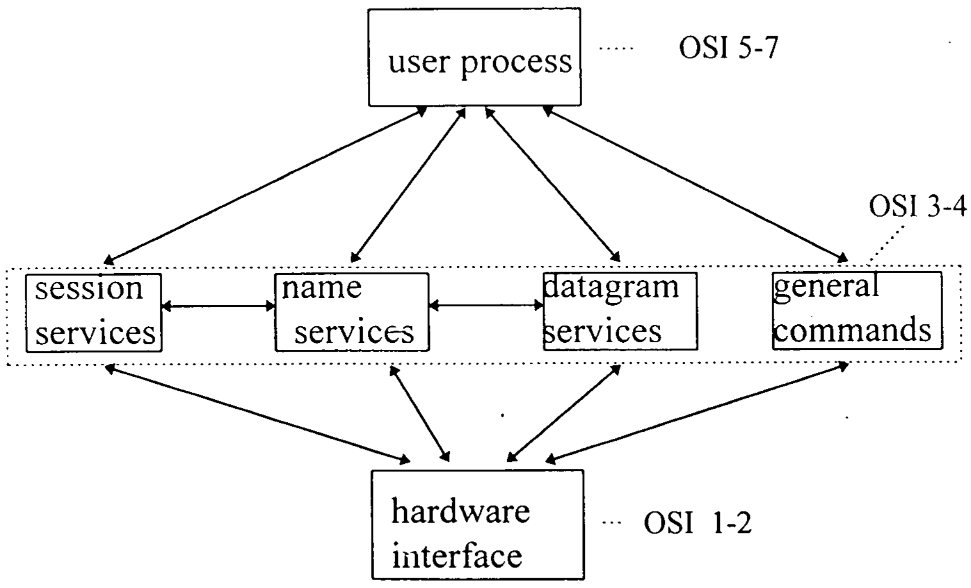
Diagram showing relationship between NetBIOS and OSI Layers

3.2. NetBIOS Services :

NetBIOS was designed for a group of personal computers, all sharing a common broadcast medium. It provides both connection oriented (virtual circuits) and connectionless (datagram) services. It supports both broadcast and multicast. Four types of services are provided by NetBIOS :

1. Name Services
2. Session Services
3. Datagram Services
4. General Commands

The general relationship among the different services of NetBIOS is shown in the next page.



3.2.1. Name Services :

Names are used to identify resources in NetBIOS. For example, for two processes to participate in a conversation each must have a name. The client process identifies the specific server by the server name, and server can determine the name of the client. The name space is flat(that is not hierarchical) and each name consists of from 1 to 16 alphanumeric characters. There are two types of names : Unique Names and Group Names. A Unique Name must be unique across the network. A group name does not have to be unique and all processes that have a given group name, belongs to the same group. We can use the name and the group names as the source and destination address, when we establish a session. NetBIOS assigns a name number to each name, we add. This name number is used to send datagrams. The NetBIOS commands relating to name management are :

ADD_NAME : Add unique name

ADD_GROUP_NAME : Add a group name

DELETE_NAME : Delete a name

FIND_NAME : Determine if name is registered (token ring implementation)

A table of unique names is kept inside NetBIOS. In addition to this table of names, a permanent name table is always present. The Permanent name is formed by taking the six bytes of network address from the network adapter card and prefixing them with 10 bytes of binary 0's . The permanent name is always unique in the network.

3.2.2. Session Services :

We can create a session between any two names on the network. Multiple sessions are possible between two names, and we can even create a session between two names at the same workstation. The NetBIOS session provides a connection oriented, reliable, full duplex message services to a user process. The data is organized into message and each message can be between 0 and 131,071 bytes. NetBIOS does not provide any form of out of band data.

The following commands provide session services :

CALL : Call active open

LISTEN : Listen Passive Open

SEND : Send Session Data

SEND_NO_ACK : Send Session DATA, No Acknowledgement

RECEIVE : RECEIVE Session Data

RECEIVE_ANY : Receive any session data

HANG_UP : Terminate Session

SESSION_STATUS : Retrieve Session Status

NetBIOS requires one process to be the client and another to be the server. The Server first issues a passive open with the LISTEN Command. The Client then connects with the server when the client execute CALL command.

AT the end of the dialog both workstations issue a HANG_UP Command to close the session.

3.2.3. Datagram Services :

NetBIOS supports datagram up to 512 bytes in the LAN. Datagram can be send to a specific name (either a unique name or a group name) or can be broadcast to the entire local area network. As with other datagram services, such as UDP/IP, the NetBIOS Datagrams are connectionless and unreliable.

There are four datagram commands :

SEND_DATAGRAM : Send Datagram

SEND_BROADCAST_DATAGRAM : Broadcast a datagram

RECEIVE_DATAGRAM : Receive the send Datagram

RECEIVE_BROADCAST_DATAGRAM : Receive broadcast datagram

3.2.4. General Commands :

There are four general commands :

RESET : Reset NetBIOS

CANCEL : Cancel an asynchronous command

ADAPTER_STATUS : Fetch adapter status

UNLINK : unlink from bootstrap server

The RESET command clears the NetBIOS name and session tables and also aborts any existing sessions.

The CANCEL command assumes that NetBIOS commands can be issued asynchronously by a user process, i.e. the user process starts a command but does not wait for it to complete.

The ADAPTER_STATUS command returns interface specific status associated with either a local name or remote name. Additionally it returns the NetBIOS name table for that NetBIOS node.

The UNLINK command was used with the original PC LAN interface when a diskless workstation was bootstrapped from a remote disk drive.

Most of the NetBIOS command come in both wait and no wait flavors. When we use the wait version of a command, NetBIOS completes the operation before returning to your program. If we specify no wait option we then have the option of polling (looping until an operation is complete) or giving NetBIOS the address of one our routines that NetBIOS will invoke when the command is completed. Such routine is called a POST routine. When the no wait option is used, our program must inspect two different return codes to determine whether the command has completed successfully: the first is the immediate return code (available as soon as NetBIOS return to our application) and the other is the final return code (which holds a value of 0xFF until the operation finishes, at which time NetBIOS sets the appropriate value).

3.3. Network Control Blocks :

To invoke a particular NetBIOS command, application builds a Network Control Block (NCB) and then executes an interrupt 5C(hex).The figure on the next page shows the format of NCB and following is the description of each field.

We set the 1- byte command field to tell NetBIOS which command we want to execute. If the high order bit is set, the command is executed in no wait mode. The 1 byte return code field contains the immediate error code (set by NetBIOS when it begins executing the command).

	length in bytes
Command ID	1
Immediate Return Code	1
Local Session Number	1
Network Name Number	1
Address Of Data/Message	4
Length Of Data/Message	2
Remote Computer Name	16
Our Computer Name	16
Receive Timeout	1
Send Time Out	1
Address Of Post Routine	4
Adapter Name	1
Final Return Value	1
Reserved Area	14

<u>Network Control Block Format</u>

After a listen or call command is executed, the 1 byte local session number field contains the LSN assigned to that session. For send or Receive commands, we put the session's LSN in this field.

NetBIOS returns the 1 byte name number field after an Add Name or Add Group Name command we use this name number not the name when doing any datagram related commands or Receive any command.

We put a far pointer to the data buffer associated with send or receive operation in this 4 byte (segment:offset) data buffer address field.

We set the 2 byte data buffer length field to indicate the length of the data buffer.

We set the 16 byte call name field to indicate the name of the workstation with which we want to communicate.

We set the 16 byte local name field to indicate by which of the names in the local name table we want our application to be known.

When we issue a call or listen command we set the 1 byte receive timeout field to a value that indicates how many half second intervals NetBIOS should use when waiting for a subsequent send command to be completed. A value of 0 indicates no time out.

We put into the 4 byte POST routine address a far pointer (segment : offset) to a routine that NetBIOS invokes when the command is completed. This field is meaningful only when the no wait option is in effect. If we set this field to zero we should poll the final return code to determine when the command is completed and whether it was completed successfully.

We set the 1 byte adapter number field to indicate which network adapter we want to use for (0 for primary and 1 for alternate).

The 1 byte final return code field contains 0xFF while a command is being processed; after the command is completed the field is set to show whether the command was successful.

The 14 byte Reserved area in the NCB is not used by our program.

Chapter Four

ADVANCED DOS PROGRAMMING

- Introduction to the Interrupts
- MS DOS File System

4.1. Introduction to the Interrupts :

As the remote program executioner is about redirection of input, output and redirection of accesses to files, before designing the actual application, first the ways in which an MS DOS program produces output, takes input and accesses file on the disk drive should be explored.

The MS DOS operating system uses interrupts for these tasks to be completed. The events on a machine with DOS operating system is typically driven by interrupts which can be generated by hardware as well as software. There are various interrupts that are provided by the processor itself. Others are provided by the ROM BIOS i.e., read only memory basic input and output services. Also on being loaded the MS DOS operating system sets up its own interrupts in the memory. Again we know that NetBIOS installs the interrupt 5C (hex) which is used by the programmers to invoke NetBIOS functions to communicate on the network.

The MS DOS program uses the interrupts to request the operating system for various services. Interrupts are actually signals which causes the computer's central processing unit to suspend what it is doing and transfer to a program that is called interrupt handler; takes the appropriate action and returns control to the original process that was suspended.

As mentioned above some of interrupts are generated by the hardware ports, such as completion of an input output and detection of a hardware failure etc. Some interrupts are generated by software by issuing the int instruction in the assembly language.

The processor usually has a reserved location in the memory called an interrupt vector that specifies where in the memory the interrupt handler for that interrupt type is located. This usually speeds up the processing of interrupt because the computer can transfer control directly to the appropriate routine; it does not need a central routine that wastes precious machine cycle determining the cause of interrupt.

The Kernel of MS DOS is designed to take advantage of the interrupts. It actually set the interrupts value ranging from 20 (hex) to 2F(hex) to point to its own routines. When ever an application program has to do some operation relating to the operating system like printing, giving output and requesting input etc., it generates appropriate interrupt instruction after setting the registers to the proper values. The interrupt handler then services the interrupt, acts as requested and returns to the calling program which resumes the execution.

The most important interrupt level that the MS DOS establishes is interrupt number 21 (hex) which is also called the function dispatcher of the MS DOS. This is a universal interrupt used by the programs to establish tasks. The action requested from the operating system is determined by the value in the AH register at the time when the interrupt is executed. The interrupt handler then performs the specific task and returns the result to the program. Nearly, every program uses this interrupt for various functions related to input, output and file services of MS DOS.

Other interrupts, which are important, are interrupt 10 (hex) and interrupt 16(hex) which are provided by the BIOS module. Interrupt 10 (hex) is used for screen related activities, and is a interface to the functions provided by the video driver. Interrupt 16 (hex) is used to handle the keyboard driver. The character can be read from the keyboard and different properties of the keyboard driver can be set by this interrupt.

The various interrupt 21 functions which are related to the output produced on the screen are function 02 (hex), 06 (hex),and 09 (hex). Other than these MS DOS functions the input can be taken by opening the device file CON in the reading mode when it represents the keyboard driver and reading from it. Also stdin handle which is automatically opened by operating system at the time of loading program can be used for input purpose.

4.2. MS DOS File System :

Our system also redirects the file related services to the launcher. The various file related operation that will be redirected and so are of importance are:

1. create and remove directories,
2. create, open and close files,
3. read from and write to the files,
4. rename and delete files,
5. search for files,
6. get or set the file attributes and
7. lock records.

MS DOS provides two distinct type of operating system calls for each major file and record operations. This is due to the reason that MS DOS is derived from CP/M as well as UNIX/XENIX.

The system compatible with CP/M is File Control Block functions. These rely on a data structure called *file control block (FCB)* to maintain certain book keeping information about open files. This structure resides in the application programs memory space. These allow the programmers to create, open, close, and delete files and to read or write records of any size at any record position within such files. But these do not support the hierarchical tree like file structure. So these are restricted to open file in the current subdirectory.

The structure of the FCB is shown on the next page.

The application program initializes an FCB with a drive code, a filename and extension and then passes the address of FCB to MS DOS to open or create file. If successful in opening file, DOS fills in FCB with certain fields in FCB with information from files entry in the disk directory. Reserved area is used by the operating system for its own purposes. Data is always read to or written from the current disk transfer area whose address may be set by the program. For calling any FCB related DOS call i.e., INT 21 (hex) the address of the FCB is sent to it.

DRIVE IDENTIFICATION
FILE NAME (8 CHARS)
EXTENSION (3 CHARS)
CURRENT BLOCK NO.
RECORD SIZE
FILE SIZE (4 BYTES)
DATE CREATED/UPDATED
TIME CREATED /UPDATED
RESERVED
CURRENT RECORD NUMBER
RELATIVE RECORD NUMBER

Fig. An FCB Structure

The most commonly file related Int 21 (hex)operation used with FCB are described in the appendix.

The other set of file and record functions are *handle* functions. These allow the programmer to open files by passing MS DOS a null terminated ASCIIZ string that describes the file location in the hierarchical file structure (the drive and the path), the filename and it's extension. If the open or create operation is successful, MS DOS returns a 16 bit token or handle that is saved by the application program and used to specify the file in subsequent operation.

The operating system maintains a data structure that contains book keeping information about the file inside its own memory space, and these are not accessible to the application program. They fully support the hierarchical file structure, allowing the programmer to create, open, close and delete files in any subdirectory on any disk drive and to read and write records of any size at any byte offset within such files. These are so more powerful than the FCB related calls.

MS DOS restricts the number of handles that can be active at any time- i.e., the no of files and devices that can be opened concurrently when using the handle family of function in two ways:

a) The maximum no of concurrently opened files in the system, for all active processes in the memory combined is determined by the entry

FILES=nn

in the config.sys file in the root directory.

b) Any process can open at the maximum of 20 files after fulfilling the above condition. However this maximum restriction can be changed by calling a DOS function. But However in any case, the total no of files opened concurrently by all processes in memory can not exceed the limit given in a).

The various handle related functions calls (Int 21 (hex) functions) are described in details in the appendix.

The other interrupt 21 (hex) functions which are important to us are the exec function which is used by the programs load and execute other programs and the get exit code used to get the return code from the terminated program to notify it to the user.

Other important DOS interrupt are INT 23 which the DOS generates when ever it finds that a ctrl+C or ctrl+break is pressed at the keyboard while doing input or output and some disk operations when the BREAK flag is on and INT 24 (hex)which the DOS generates whenever any critical error is generated.

The interrupt provided by the BIOS when it get a ctrl+break is 1B (hex).Also INT 10 can used to produce output on the screen with the help video driver and the characters may be read at the keyboard using INT 16 H which is the interrupt provided by the keyboard handler.

Chapter Five

Design of Different Layers above NetBIOS

- The Design Of Error Correcting Layer
- The Design of Presentation Layer

The design of the application required design of two more protocol layers above the services provided by NetBIOS. The NetBIOS only provides session and datagram services to the user for the transfer of data and this is not sufficient for our purpose. Error free transmission is required by our application. Also different types of data is transmitted by this system from the executioner to the launcher and from the launcher to the executioner. So some type of abstraction is needed to represent the data being transferred. These services are fulfilled by these layers. These layers also make the programming more easy and less error prone. The application layer is actually used by our system to send or receive data. And this layer uses the error detection layer to transmit data. But these layers are only used for the transfer of data. The establishment of session, adding and deleting names, and tearing of connection is done directly by our application without interference of the layers.

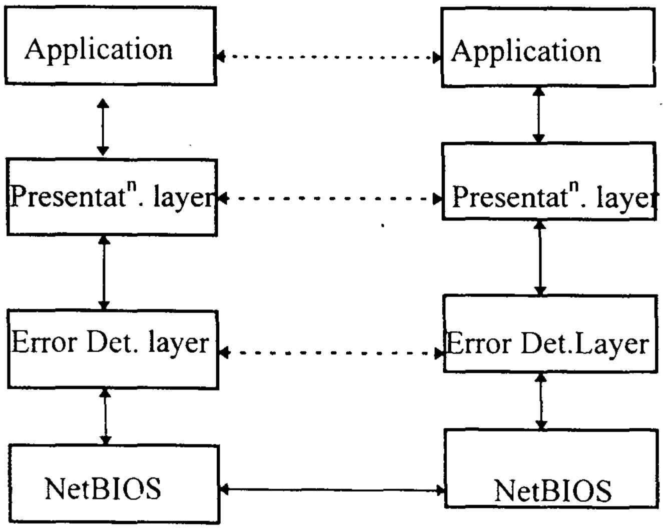


fig. The schematic diagram showing the relation between different network layers .

5.1. Design Of Error Detection Layer:

As was stated, NetBIOS does not provide checksum for the data it is carrying assuming that the underlying data links provide reliability. But it is vital that this application does not rely upon underlying hardware dependent system but develops its own method for error detection and retransmission. This becomes necessary because it can't be made sure that underlying data link will provide the adequate reliability. Also various vital information is communicated through NetBIOS by our application, a small error will prove dangerous as this may cause a different action to be taken stalling the whole system, even damaging the files on the disk.

This layer provides the data to be sent and received correctly by the applications. This layer is actually built over the session as well as datagram services of NetBIOS. The datagrams packets and the session packets are handled differently by this layer. This is because of the reason that the session packets are always delivered, and are in the same order in which they are sent but the datagram packets may be lost in the transmission and the ordering of received packets may be different from the order in which they are sent. So different policies has been adopted by the layer for them. The time-out and retransmit policy has been adopted for the datagram packets. And the Negative acknowledgement and retransmission has been adopted for the session packets.

5.1.1. Comparison of different methods of error detection :

Various methods for error detection was examined for this purpose viz., parity checking, longitudinal redundancy check, cyclic redundancy check and ISO checksum algorithm. In these CRC has the most suitable properties for error detection but was found to be relatively inefficient for the purpose of software implementation due to the number of calculation involved. So the ISO checksum algorithm which also has several desirable properties and the software implementation of this is relatively much efficient than the cyclic redundancy check.

This checksum, it can be shown assuming all bits errors to be equally likely:

- detects all the 1 bit error,

- detects all 2 bit errors,

- fails to detect only 0.000019% of all burst error of length not exceeding 16.

- fails to detect only 0.0015 % of all larger burst error.

This method originally specified by Fletcher uses 16 bit checksum which is included with the data which has to be transmitted as a field. The checksum is first calculated by the sender and placed in the outgoing data. The receiver then applies the same algorithm to the entire header including the checksum data and should get 0 if there are no errors. This method uses modulo 255 addition. The data is considered to be a sequence of 8 bit unsigned integers.

5.1.2. Generation Of Checksum:

The checksum is generating by producing two octets calculated on the octets of data to be checked. The first is the sum modulo 255 of all the octets of data in the message and the second is the sum modulo 255 of every octet weighted reversely by its position in the message. For convenience the checksum calculation is performed on the entire message including the fields that will contain two checksum octets. These fields are set to zero in start. The position of the checksum is 1 and 2 in our method and so the above calculation is performed with taking value of first and second octet to be zero. Finally the value of the first and second checksum is generated such that the result of generating checksum again with the whole data produces zero.

c1'=0=summation of all the octets + c1+c2 ....(1)

c2'=0=summation of all the octets excluding the checksum weighted reversely by it's position in the entire message + c1*L + c2 * (L-1) ....(2)

where c1' and c2' are the result of checksum calculation at the receiving station and should be zero , c1 and c2 are the checksum data that has to be sent and L is the total length of message including the checksum.

Solving equation (1) and (2) the value of the checksum data can be calculated and placed in the message.

5.1.3. Checking Of Checksum :

Upon receipt of the message the receiver performs the calculation for c1' and c2' with the received message including the checksum data and if the result is equal to zero the checksum calculations has succeeded and no errors are assumed.

5.1.4. Method used for message transfer in session :

The packet is sent by the sender and then it waits for acknowledgement from the receiving station. If it gets a Negative acknowledgement or a corrupted acknowledgement packet, it retransmits the packet again and waits for acknowledgement. If the packet is received correctly by the receiver then it sends the acknowledgement. Otherwise it sends negative acknowledgement which signifies that the sender has to send the packet again. As packets are never lost in a session the strategy for time out is not required. Again the problem of getting duplicate packets comes in.

For this the packet contains a sequence number that may be used for duplicate detection. The sequence number ranges from 0 to 255 and then rounds back to 0. On getting a duplicate packet the receiver just sends the acknowledgement of the packet and discards the packet.

5.1.5. Method used for message transfer in a datagram:

The application uses the datagram services to send small messages that contain small no of bytes. So at times there is only one packet that has to be sent by the sender. For this purpose the sender sends the datagram packet, starts a timer and waits for the acknowledgement from the receiver. If the acknowledgement does not come within stipulated time or the acknowledgement packet is corrupted then it retransmits the packet again. The receiver on receiving a datagram first checks for the errors and then sends the acknowledgement or the negative acknowledgement packet as necessary. If the receiver receives a duplicate datagram packet then it simply sends a acknowledgement and discards the message packet. For these purpose the sequence no field of the message is used. Also the data sent by the datagram packets are not very much necessary so the sender retries the no retransmission only for a limited no of times that is 10 times and then assumes that the packets has been received correctly.

5.1.6. The Format of The Packet :

The packet format used by this layer is as follows:

2 bytes	1 byte	1 byte	1 byte	1 - 535
checksum	messtype	Seq No	More pkt	Data being sent

Data Packets used by the error detection layer

2 bytes	1 byte
checksum	messtype

The Ack packet

2 bytes	1 byte	1 byte
checksum	messtype	seq no.

The NAK packet

Figure of packets used by the error detection layer.

The packet first contains the checksum data of two bytes length that is used for error detection. After that it contains the message type of size one byte that describes the packet type i.e., the packet contains the data or acknowledgement or the negative acknowledgement. If the packet contains the data then the there is a one byte field that tells the sequence number of the packet that is sent. After this there is a byte which says that if there are more packets to be received or not. This is useful in the case if the data that is being sent is larger than the maximum size that can be accommodated by the packets. And the rest of the packet is the data portion which is actually the packet that is used by the presentation layer.

5.2. The Design Of The Presentation Layer:

As our application needs different types of data to be transferred from one workstation to another, the design of this layer becomes necessary. This layer is concerned with how data is being represented that is being exchanged. The application sends requests and responses from one workstation to other. Additionally, it presents the lower session layer services to the application. So one task of the presentation layer is to convert application data into some standard format that is passed through the network to the receiving station.

The general format of a request packet is as:

1 byte	1 byte	1 byte	?	2 bytes	?
messtype	Int. No	. No of rel pars	Rel. pars	Buffer size	Buffer

The fields in the packet can be described as:

1. Message type determines the type of message the packet is carrying i.e., here it represents request.

2. The Int no tells about the interrupt no that is requested by the request packet.

3. The no of relevant parameters in the packet is after that and this is mostly 4 representing the 4 registers AX, BX, CX and DX that is being transferred.

4. The relevant parameters field which mostly contains the value of AX, BX, CX and DX respectively.

5. The buffer length contains the length of the buffer area in bytes.

6. The buffer area generally contains the data which is required in the request operation such as data to be written to a file, the name of the file to be opened etc. The buffer may contain zero or more structure. The first byte of the structure determines the type of the structure and the next two bytes gives the structure length. The rest of the structure contains the actual data.

So the open file request using handle functions request will be like:

1 byte	1 byte	1 byte	8 bytes	2 bytes	?	2 bytes
messtype	Int. No.	No of rel pars	Rel. pars.	Buffer size	filename	handle

Here message type will be request, INT NO will be the 21 and the relevant parameters will contain the values of the four registers. The buffer will contain the name of the file that has to be opened and the handle that the executioner will assign to the opened file and which will be used by the executioner to further reference the handle.

The close file request will be like :

1 byte	1 byte	1 byte	8 bytes	2 bytes
messtype	Int. No.	No of rel pars	Rel. pars.	Buffer size

Message type will be request, Int no 21 and the relevant parameters will again contain the four registers and the buffer length will be zero as the handle no is already there in the registers.

The read file packet will be like :

1 byte	1 byte	1 byte	8 bytes	2 bytes
messtype	Int. No.	No of rel pars	Rel. pars.	Buffer size

All the parameters here are similar to the parameters in the close file request. The buffer again contains nothing as the relevant parameters are present in the registers.

The write file request is like :

1 byte	1 byte	1 byte	?	2 bytes	?
messtype	Int. No.	No of rel pars	Rel. pars.	Buffer size	Write Data

The buffer contains the data to be written to the file or device.

The input and output redirection packet are similar to the above packets.

There is a response packet corresponding to every request packet. The response packet format is like :

1 byte	1 byte	2 bytes	1 bytes	1 bytes	1 byte	?	2 bytes	?
messtype	Int. No.	Return error Class	Critical error Class	Return Code	Par Cnt	Pars	Buff length	Buff

Like for open file request packet if request successful the response packet will be like :

1 byte	1 byte	2 bytes	1 byte	1 byte	1byte	?	2 bytes
messtype	Int. No.	Return error Class	Critical error Class	Return Code	Par Cnt	Pars	Buff length

The message type field contains the value response. The interrupt number field tells about the interrupt that invoked the request, the relevant parameter's field is similar to that of request packet. The error classes and the return code specifies the error that occurred while the request was being serviced.

The buffer data field is similar to the request packet and is similar in structure i.e.,

```
struct buffer_data{
        byte type_of_data;
        byte length_of_data[2];
        struct actual_data;
        }
```

Other type of packets that has been used are:

1. A broadcast datagram request which is used to find all the executioners on the network which are not busy. It's format is like :

messtype	own_name

the own name field contains the name of the launcher from which this request originated.

2. The response for above datagram which is of the format :

messtype	own_name of executioner

The own name field contains the name of the executioner sending the response.

3. The start session packet which is used by the launcher to submit the name of the program to be run on the executioner and other parameters.

messtype	filename len	Filename	Mode of Connection

4. The packet sent by the executioner before starting execution which also contains the drive no assigned to the drive on the launcher. If the drive no is greater than 26 it is assumed that no drive letter has been assigned to the local drive on the launcher.

messtype	Drive no. assigned

5. The packet sent by the executioner at the end of the execution of the submitted program which contains the exit code of the program.

messtype	exit_code

5.1. Message Type, Error Codes and the buffer data type

1. Message Type field is an essential part of the packets send by the presentation layer and tells about the message being carried out by the packets.

The message type codes which are used by the system are:

1. RPE_REQUEST_PACKET	0	The packet contains a request for execution of some program
2. RPE_RESPONSE_PACKET	1	The packet contains a response to the previous request.

3. RPE_START_SESSION	2	sent by the launcher to submit the name of the program to be executed.
4. RPE_END_SESSION	3	sent by the executioner at the end of the execution.
5. RPE_STARTING_EXECUTION	4	sent by the executioner at the time of starting execution of the loaded program.
6. RPE_FINDING_EXECUTIONER	5	sent by the launcher to find the address of executioner in the network.
7. RPE_RESPONSE_DGM	6	sent by the executioner when it gets the packet no 6 .

2. The error codes which are returned in the response packet describes the errors occurred while the previous request was being serviced. These are used to set the extended error information or calling the critical error handler on the executioner. The error codes are provided by the system running the launcher and it provides some description of the error. The launcher must choose the code that best describes the error.

The commonly used error codes are:

RPE_NOERROR	0	No error
RPE_FNOTFOUND	1	File not found
RPE_EACCESS	2	File is accessed incorrectly
RPE_EMAXOPEN	3	Maximum limit of opened file reached.
RPE_EEOF	4	EOF of file reached
RPE_EFILENOPEN	5	File not opened
RPE_EFRDONLY	6	Writing to read only file
RPE_PATHNOTFOUND	7	Specified path not found
RPE_REMCURDIR	8	Attempt to remove current directory
RPE_CTRLCPRESSED	9	Control break or Control c pressed.

The critical error codes represent the critical errors that occurred while performing the request on the launcher. This is usually caused by a hardware error. The different error codes are :

RPE_WRITEPRCT	0	Write protect error
RPE_DRVNOTRDY	1	drive not ready
RPE_DATAERROR	2	data error
RPE_SEEK_ERROR	3	seek error
RPE_SECNF	4	sector not found
RPE_RFAULT	5	read fault
RPE_WFAULT	6	write fault
RPE_GENFAIL	7	general failure

The return code tells about the completion status of the command requested. It is set to 0 if successful and set to 1 if the command could not be completed due to some error, the error may be determined using the error codes described previously.

The buffer data type is used to identify the data which the structure is carrying:

1. RPE_HANDLE	1	the data is a handle
2. RPE_WRITEDATA	2	the data is to be written to some device or file.
3. RPE_READDATA	3	the data is the response to the read request.
4. RPE_MDDESB	4	the data is the media descriptor byte returned by some function calls.
5. RPE_FCB	5	the data is a copy of FCB
6. RPE_XFCB		the data is a copy of extended FCB
7. RPE_DIRNAME	7	the data contains the path of a directory
8. RPE_FILENAME	8	the data contains the name of a file
9. RPE_DTA	9	the data contains the offset of the DTA Being used.
10.RPE_FCB_ADD	10	the data contains the address of the FCB

Chapter Six

DESIGN AND IMPLEMENTATION OF THE MAIN PROGRAM

Both the executioner and launcher use the services provided by the presentation layer, which has been discussed in the previous chapter, to transfer request and other messages. But the connection and the termination of a session is done directly using the NetBIOS services.

In this chapter the design and implementation of both executioner and launcher is described. First we take a look at some design considerations that is required before designing the executioner and the launcher.

6.1. Design Considerations:

6.1.1. Comparison of Session Services and Datagram services

NetBIOS provides two types of services for transmission of data, session and datagram. The most important difference between datagrams and the messages that are sent and received during the session is that datagrams get lost sometimes in transmission and the program is not notified of this. Messages sent within the framework of a session are guaranteed to be delivered. If multiple packets are outstanding, session control also guarantees that they are delivered in the same sequence in which they are sent. But these services makes the delivery of the message in a session somewhat slower due to the overhead required. The delivery of datagram may be ascertained using some acknowledgement techniques.

So the session services are good for point to point communication in which two workstations pass back and forth several related data packets. Datagrams are good design choice in application whose dialogs consist of unrelated messages, only a few messages or a series of messages that do not always have to be a complete set.

For our purposes, when the system is operating, related messages are passed between two workstations most of the time, except at the time when the network is searched for executioners by the launcher. So it becomes reasonable to use datagram services at the time of searching and session services at the other times when point to point communication takes place between the launcher and the executioner.

6.1.2. Redirection of Input Output and File related requests

The policy adopted to redirect the input, output and file related requests to the launcher is based upon the methods that are used for placing the requests. As has been mentioned, the MS DOS programs use interrupts for these purpose. So the method used is to change the interrupt handler for the relevant interrupts with our routines. These routines then check the cause of the interrupt and redirect the request to the launcher if necessary or pass the request to the original handler.

6.1.3. Assigning File Handles For files opened on the remote Machine:

There can be two methods that can be used to assign handles for the files opened on the launcher's drive.

The first one is to assign the handles (which are actually integers) by its own. But this value must not conflict with the handles supplied by original operating system. For this purpose, handles of value greater than 255 may be assigned (the maximum value assigned by the operating system to a handle is 254).

Second one opens a local dummy file and returns the handle to the program and remembers it, so that when ever the program uses this handle the request is transferred to the launcher.

The first method has lesser overhead at start but it has some complications. First one is that if a local handle is redirected to a remote handle then the local handle must be remembered by the executioner that it should redirect all the requests with this handle to the remote machine. Also if a remote handle is redirected to a local handle then the executioner has to remember this to redirect all the requests to the local system. Again if the remote files are opened and closed several times, the executioner must maintain the list of unassigned and assigned handles.

The second method has more overhead but the management of the handles becomes easy. Only one array structure may serve the purpose if the handle that has been assigned is remote or not. Also there is no problem in redirecting the handles etc. So this method has been used in our application.

Now after this discussion we consider the design and implementation of the launcher which is invoked by the user to submit and execute a program on this system.

6.2. The Design Of Launcher :

The design of launcher is less complex than the executioner as most of the processing is done by the executioner. The launcher first connects to the executioner and submit the program for execution. After that it waits for any output or input request and also the file related request for the launcher's local drive made by the executioner, executes the particular request made and then sends the result back to the executioner. Also it maintains different tables that are necessary to map the file related calls to the local file related functions.

6.2.1. The Different Data Structures Used By The Launcher :

1. The file handle table: It contains the local file handles that is used by the launcher and the corresponding file handles that is assigned by the executioner to the running program. This is used by the launcher at the time

when some file handle related request comes from the executioner that has to be handled.

2. The current DTA address on the executioner's machine that is used by the FCB related file function calls. This is used by the launcher to determine some type of errors that may result due to file read and write operations.

3. Different FCB structures that is used for FCB related file operations on the launcher.

4. A FCB address table which is used by the launcher to correlate the requested FCB with the FCB that is used on the launcher.

6.2.2. Broad Algorithm Used By The Launcher:

1. It first checks the version of the operating system (MS DOS). Then it checks whether share.exe is installed or not. Then it checks the presence of NetBIOS.

2. It adds its name to the network name table of NetBIOS.

3. It checks the command line parameters and tries to contact the appropriate executioner on the network. Also it gets other specifications from the user.

4. Connects to the executioner and establishes a session with it and submits the program that is to be run by the executioner.

5. Gets the drive letter for the local drive on the launcher workstation and informs it to the user.

6. After that it waits for any request made by the executioner and services the request and updates the tables as necessary.

7. Then it sends back the result to the executioner. It also indicates the critical errors that occurred while servicing the request on the launcher's machine.

8. It loops till a final exit call is not received from the executioner and after that it notifies to the user the exit code of the running application.

9. It then tears down the session, deletes its own name from the NetBIOS name table and finally exits.

6.2.3. Detailed Description Of Algorithm Of The Launcher:

6.2.3.1. Initial preparation by the launcher:

On being called first the launcher determines the version of the operating system that is running and confirms that it is not less than the minimum requirement (4.0).

```
if( os_major < 4)
{
        printf("\n Early version (<4) of DOS not supported");
        exit(0);
}
```

Then it checks whether share.exe is installed using the DOS interrupt 2F that is the multiplex interrupt. After that it finds whether NetBIOS is installed on the computer or not. It is done in two steps. First it is tested whether Int 5C entry in the vector table contains a valid pointer (Non NULL pointer). After that a call to NetBIOS is performed using an invalid command and the return code is checked. If it corresponds to invalid command then it may be safely assumed that the NetBIOS is active.

6.2.3.2. Adding it's name to the NetBIOS name table:

For this purpose it opens the file launch.dat in the default directory of the launcher and reads the name from it that it will use to identify itself for communication with the executioner. The file launch.dat may be used by the administrator to provide each of the launcher a unique name. After that it calls the ADD_NAME command of NetBIOS to add its name to the NetBIOS name table. Unique name must be provided to the launcher on each workstation so that a conflict of name does not arise. If any error occurs then the launcher is terminated after displaying the cause of error.

```
memset(&add_name_ncb,0,sizeof(NCB));
add_name_ncb.NCB_COMMAND=ADD_NAME; /* command is
                                        ADD_NAME */
strcpy(add_name_ncb,localname);/* localname of the launcher*/
NetBIOS(&add_name_ncb); /* transfer to NetBIOS function */
```

6.2.3.3. Contacting the appropriate Executioner:

The launcher then retrieves the command line parameters to find the name of the executioner that has to be contacted and the name of the program file that it has to submit to the executioner and the mode of the execution.

If the command line does not contain any of the parameters, it asks the user for the same. For example if no command line parameter has been given, it asks the user to give the mode of execution. Then the launcher finds out all the executioner present on the network. This is done by sending broadcast datagram with the help of presentation layer (which issues a RPE_FINDINDING_EXECUTIONER packet). The launcher then waits for the response from different executioners. After that it shows all the names of the executioner that have responded to the request and asks from the user to select one of them. Then the launcher takes from the user the name of the program to be launched with any redirection parameter such as filters that redirects the input and output.

6.2.3.4.Connecting to the particular executioner:

Then the launcher issues a NetBIOS CALL command for the executioner selected and connection is established between the executioner and the launcher. If the connection is not established, it is assumed that the executioner is busy or is not present on the network and user is informed of the situation.

```
memset(&call_ncb,0,sizeof(NCB));
call_ncb.NCB_COMMAND=CALL;
memcpy(call_ncb.NCB_NAME,localname,16);
memcpy(call_ncb.NCB_CALLNAME,remotename,16);
```

```
call_ncb.NCB_STO=sto;
call_ncb.NCB_RTO=0;
NetBIOS(&call_ncb);
local_session_number=call_ncb.NCB_LSN;
```

After that it saves the session number returned by CALL and uses it to further send or receive data. The launcher then submits the program and other specification to the executioner using the start session packet of the presentation layer. After that it receives the executing packet from the executioner which also contains the name of the drive assigned to the local drive. It saves this value in a variable for further use and also shows it to the user. The drive number represents 0= A, 1=B etc.

Now the launcher set the int 23 and int 24 that are the ctrl+c handler and the critical error handler to point to its own routine that on being called set the particular flag that represents the particular error or the state of ctrl+c.

6.2.3.5. Waiting for the requests made by the executioner

The launcher then issues a receive request command and waits for any request sent by the executioner. On getting the request it first examines the interrupt no requested and the function that has to be executed. If the interrupt no is either 10 or 16 it is serviced and appropriate response is sent to the executioner. Also various data that is needed by the interrupt 10 and interrupt 16 request is sent back to the executioner.

If the request is for the interrupt 21, the function is determined using AH, the upper part of AX register. If it is a request for input or output, the interrupt 21 is executed with the same parameters and the result is sent back to the executioner. If the requested function is related to file functions related with handles then these actions are taken :

1. If the request is to open the file, the file is opened using the same function. The handle returned is saved with the handle sent by the executioner in the handle table. The handle table is actually a linked iist that is

used for determining which local handles corresponds to the handle on the executioner's machine.

2. If the request is for closing a file then the appropriate local handle is determined from the local handle table and the file is closed using the close file function and the entry for the handle is deleted from the handle table.

3. If any other handle related read or write operation is required by the request then the corresponding local handle is used for the required operation and the required data is placed in the buffer.

4. If there is a duplicate handle call then the same command is executed on the local machine and the entry is added in the local handle table with the duplicate handle on the executioner's machine (that will be assigned to the running application) which is also sent with the request.

5. If the call is for redirection of handle then the handles are examined for taking the appropriate action. If both handles belong to the machine on the launcher (it is found out by looking into the file handle table) then the directly forced redirection is applied for the handles. If the handle which has to be redirected is on the executioner's machine and the other handle is on the launcher's handle table, then one dummy file is opened and it's handle is associated with the handle to be redirected and placed in the file handle table. After that the handle is redirected to the original handle. In the other case when the handle to be redirected is on the local machine and the other is on the executioner's machine, in that case the entry in the local handle table is deleted and the second handle is redirected to a null device.

For FCB related function the process is somewhat complicated. The launcher maintains a variable which always has the offset value of the current DTA on the executioner. It is necessary to find out an error that is called segment wraparound error which occurs if the length of the data to written to the DTA or to be read from the DTA exceeds the boundary of the segment of the DTA and then corruption of memory may occur.

Whenever there is an open file call using FCB the launcher copies the FCB into its own memory space, then calls the open file function in the current directory. Also before performing the operation the launcher changes the disk letter in the FCB to represent its own disk letter (not the redirected letter). The address of the original FCB is saved in a table of FCB addresses with the local FCB address to later identify the local FCB related to the remote FCB on the executioner's machine in subsequent FCB related calls. The appropriate result is sent to the executioner in the response packet.

If the call is for file close function then the file is closed and the result is sent to the executioner. The entry in the FCB address table corresponding to this FCB is deleted.

If any file read or write command is requested, the launcher determines the address of corresponding local FCB using the FCB table and copies required data into the local FCB from the FCB data in the packet and performs the call. The response is then sent to the executioner. The segment wrap is also determined using the length of data to be read or written and the offset address of the DTA and on any conflict the particular error is set in the particular register.

Any other directory and disk related requests are serviced in the normal fashion. And the result is sent to the executioner in the response packet.

If any error occurs at call of any of these functions or ctrl+c is pressed at any time then the particular flag is set. Before sending each and every response the launcher sees if the critical error or ctrl+c flag is set. If this is the case then the error codes in the response packet is set to represent the condition. After sending the response that the particular error flag is reset.

6.2.3.7. Getting the exit call from the executioner

If the launcher gets an exit call by the executioner then it checks the exit code and if it is a normal exit, it closes all the open files. Then it notifies to the user the exit status of the program. After that it tears down the session using the hangup command and deletes it's name using the delete name command of the NetBIOS. Then it restores the address of original interrupt 23 and interrupt 24 and then exits.

```
//setting the interrupt 23 and interrupt 24 to its own handlers
old_23=getvect(0x23);                //getting the address of
old_24=getvect(0x24);               //original int handlers
setvect(0x24,own_critical_error_handler);//setting our own handler
setvect(0x23,own_ctrlc_handler);           //for the interrupts
                :
                :
                :

setvect(0x23,old_23); //setting the original interrupt handlers
setvect(0x24,old_24); //for interrupt 23 and interrupt 24
exit(0);                       //exiting
```

6.3. Design Of The Executioner :

The design of the executioner is more complicated than the launcher. This is because the launcher has only to process the redirected requests and send the result back to the executioner. But the executioner has the burden to execute the program, determine which of the requests are for input and, output and for files on the launcher's drive and then transfer the request to the launcher. And then after getting the result from the launcher, it sets the registers to appropriate values and copies any data requested to the buffer of the requesting program. This process is complex because it should not corrupt any register's content. Also the executioner should use very less memory to load itself. For this purpose, only those parts are loaded into memory that are necessary at a time. This makes the design of the executioner more complicated.

6.3.1. The data structures used by the executioner :

1. It uses a table of the FCB pointer's that are on the disk of the launcher. A linked list is used for the creation of this table.

2. It also uses an array is_remote[MAX_HAND] that is used to determine the drive is remote or on the disk of the executioner. If the value is_remote[i] is equal to 1 then it signifies that the handle i is located on the launcher's dictionary. The MAX_HAND is the maximum no of handle that can be opened on the machine.

3. It also uses an internal stack that is necessary to prevent overflow of stack. The stack is defined like

```
byte stack[1000]
our_stack_seg=_ds and stack pointer= 998;
```

6.3.2. The algorithm used by the executioner :

1. On being invoked it first checks the minimum requirements which is the version of the operating system and the presence of share and NetBIOS.

2. Then it adds its own name to the NetBIOS name table.

3. It then waits for setting up of a connection by a launcher for a request to run programs.

4. On getting a request to execute a program it prepares the executioner's machine to redirect the requests. Also different tables are initialized.

5. Then it loads and executes the program requested by the launcher from the local disk or from the disk of the launcher's site.

6. It redirects all the input and output requests to the launcher. Also it determines all the requests for the drive at the launcher and redirects the request to the launcher's machine.

7. It then gets the response from the launcher and passes it to the application program. On completion of the application it sends the final exit code to the launcher.

8. Then it restores the machines state to the normal and waits for any launcher for another request to execute an application.

6.3.3. description of the algorithm:

6.3.3.1. initial preparation:

The algorithm for checking out the NetBIOS presence and other operating system constraints is similar to the launcher.

After that the executioner uses the file exec.dat to get the name it has to use to describe itself. The method for adding name to the NetBIOS name table is similar to that of launcher's. After this step the executioner determines the drive letter that can be assigned to the launcher's drive when the normal mode of execution is required.

```
for(drive letter=3; drive_letter<=26;drive_letter++)
{
        regs.h.ah=0x36;
        regs.h.dl=(unsigned char)drive_letter;
        int86(0x21,&regs,&regs);
        if((regs.x.dx & 0xFFFF)==0xFFFF)
        {
                assigned_drive_letter=drive_letter;
                break;
        }
}
```

The determination of the name of the drive is done only at this time, and the letter is saved in a variable for further use.

6.3.3.2. Listening for any Launcher that may contact the executioner

The executioner then executes an listen command and a receive broadcast datagram command in nowait mode. After that it waits for any request that may come. If the user presses Esc key at this time then the executioner exits.

```
write on the executioner's screen that it is not processing any request.
Loop while no call from any launcher
{
        if user on the executioner terminal presses Esc key
        delete it's name from the NetBIOS name table and exit;
}
```

The executioner then finds out if it received a broadcast datagram or a call has been issued by any launcher.

If the a broadcast datagram has been received, it looks into the request packet. If it is the RPE_FINDING_EXECUTIONER, it sends back the response_for_search packet which tells the launcher it's own name by which it can be contacted. After this the executioner again issues a receive broadcast datagram request.

If the listen command returns, the executioner cancels the outstanding receive broadcast datagram command and cancels any other activity that it is doing like sending response datagram packet. The executioner then issues a receive command and waits for any packet from the launcher. If it gets a start_of_session packet it carries on. Otherwise it simply closes the connection and goes back to the waiting mode.

6.3.3.3. Making the connection and execution of submitted program:

On getting a start of session command packet the executioner finds out the mode of execution. If the mode of execution is normal then the executioner sends the drive letter determined earlier to the launcher in the starting execution packet. If the mode of execution is dumb terminal mode then the launcher simply sends a value 27 in the starting execution packet which signifies that no drive letter has been assigned to the launcher's drive.

Now the executioner examines the name of the program that has to be loaded and determines the location of the program that has to be run. Then the launcher prepares the machine so that the output, input and file requests can be redirected to the launcher's machine. First the executioner loads the required routines into the memory that are required to handle the redirection requests. The alternate interrupt 21 handler is not loaded if the mode of execution is dumb terminal. Then it first replaces the interrupt 10 handler and interrupt 16 handler with it's own handlers and saves the old handler's addresses. If the mode of execution is normal then the interrupt 21 handler is also replaced by its own handler and the old address is saved in a variable. The executioner shrinks it's allocated memory to the minimum, it requires for the proper execution. Clearly the dumb terminal mode of execution requires lesser memory for the executioner to be loaded in memory.

Then the executioner prepares for executing the required program. It first sees if the required program to be run is on the executioner's machine or is on the launcher's machine. The executioner first creates two FCBs from the cōmmand line parameter of the program to be run. Also it looks for any redirection of standard input or output, which is determined by the presence of character '<' and '>' input in the command line parameter. If the command line parameter contains any redirection then the redirection is effected by first opening the file on the required machine and redirecting the handle for standard input or output using the redirect handle function of interrupt 21. The redirection of handle is described below.

After that the actual loading of program is done. If the program is on the executioner's disk then the program is loaded using the interrupt 21 function 4B. If the program is on the launcher's machine the executioner loads the program through the network. It first creates a PSP in the memory using the interrupt 21 function. If the program has an extension of com then the image of the program is directly loaded into the memory at the PSP:0100 offset and then the control is transferred to the address. Before transferring the control to the program the executioner loads the different registers with appropriate values. If the program has an extension .EXE then it is loaded after the relocation factors has been calculated using the header of the exe file and then the program is loaded into the memory at appropriate address. After that the control is transferred to the program after setting the registers to appropriate values.

Then the required program starts running on the executioner's machine. Now all the input and output requests which are made by the running program calls our routine instead of the usual routine. If there is a call to interrupt 16 for input purposes it is simply redirected to the launcher's machine and the launcher's machine uses the same interrupt to get the input from the user and sends the result to the executioner's machine. Similarly the interrupt 10 is redirected to the launcher. A particular handler for interrupt 16 or interrupt 10 looks like:

```
void interrupt new_10(void)
{
        store the current stack segment and stack pointer
        set the stack segment and stack pointer to our own internal
        stack
        find out the function and subfunction and make a request packet
        send the request packet to the launcher's machine
        wait for the result and then on getting response packet set the
        registers to appropriate values and place any data required in
        the requested buffer address
        restore the current stack segment to the previous value
        return
}
```

If the mode of execution is normal then the interrupt 21 request is also processed by our routine. The interrupt 21 has many functions. The handler first determines the function requested by the value of AH register and branches to the appropriate routine to handle the request. This handler is particularly a complex one as this has many functions involved and this also is used for input, output as well as file and directory request.

If the function call relates to the input and output to the standard input or output, first it is found if the input or output has been redirected to a local file or not using the is_remote array values. (The value for is_remote[0] is checked for standard input and is_remote[1] is checked for standard output.) If the value are found to be 1 for the corresponding stdin or stdout then the call is redirected to the launcher. All this is done by sending the proper request packet to the launcher. On the other hand if these are found to be not redirected then the control is passed to the original interrupt handler which handles the request. If at the time of loading the standard input or output has not been redirected, the executioner automatically set the value of is_remote[0],is_remote[1] and is_remote[2] to 1 to redirect all the input or output request to the launcher's drive.

If the function relates to creating or opening file using handle then the function finds out if the file to be opened is on the launcher's drive by looking the first two letters in the name or if the drive has not been mentioned then by the current disk drive in effect. Also the file name con which is name for console device is always supposed to be on the launcher's machine. If these point to the launcher's drive then first a dummy file is opened on the executioner by using the open file call to the original interrupt handler. If the dummy file is not opened due to some reason, the same result is reflected to the executing program. On successful opening of the local file, the request is transferred to the launcher with its handle. Then the response for the request is checked and if it is a success then the is_remote[handle]is set to 1 and the local handle value is returned to the requesting program. If there is a failure in opening file, then the local dummy file is closed and the particular flags are set by the executioner for reflecting the failure to the requesting program. In case of the file being on the local drive the request is passed to the original interrupt handling function.

If a close file request with using handle comes then the executioner then it first checks if the handle is remote and if it is, sends the request to the launcher. On getting a success the local handle is also closed and is_remote array is modified to reflect the status.

If the request is for any read or write file then it is checked if the handle is on the launcher's drive then the request is simply transferred to the launcher. In other cases the request is transferred to original int 21 handler.

If the request is for redirect handle then four case may arise :

1. both handles are local on the machine, and then the request is transferred to the original interrupt handler which services the request.

2. If both are on the remote machine then the request is transferred to the launcher which services it and makes appropriate changes.

3. If the handle to be redirected on the local machine and other handle refers to the file on the launcher's drive then the is_remote for the local

handle is changed to 1 to make it reflect that it represents the file on the launcher's drive.

4. If the handle to be redirected refers to a file on the launcher's drive and the other handle is local in that case the request is sent to the launcher as well as to the local interrupt handler and the is_remote for the remote handle is changed to represent local. This makes the handle to be redirected to the true file.

If the request is for duplicating remote handle then the local handle is duplicated first with the help of original int 21 handler and then the request is transferred to the launcher with the new handle received by duplication.

If the function call is an open call then the drive byte of the FCB is checked to find if the request for the launcher's drive. If it is not then the request is passed to the original handler other wise redirected to the launcher. The address of the FCB is saved in the FCB table maintained by the executioner. The request packet contains in its buffer the FCB structure and also the address of the FCB. The response packet contains the information to be filled in the FCB structure and the executioner fills them into the FCB.

If their is an request for closing the FCB file then the FCB address is compared with the addresses in the table and if it is found in the table the request is transmitted to the launcher's machine and after getting the result the address of FCB is deleted from the FCB address table.

If the request is to read or write in file on the launcher's machine then the data (if the request is to write) and the FCB is sent to the launcher. After getting the response the executioner fills the particular fields in the FCB to represent the current state of the file and fills the data read in the current DTA.

Similar action is taken when an extended FCB is used. Only the bytes positions required to be read in the FCB are increased by 7 bytes.

Other critical functions to be serviced are find first and find next matching files using FCB which are serviced by simply sending the address of FCB and the content of the FCB at the time of find first matching file function. Then on subsequent find next matching file function only the address of FCB is sent to the launcher. The similar action is taken for find first matching file and find next matching file function using the handles. In this case the drive letter in the path of the file is found out or in absence of the drive name the current drive is used to determine if the request has to be transferred to the launcher. The executioner in this case sends the whole structure to the launcher at each request of find first or next matching file to the launcher. The response carries the modified structure that is copied back into the same place. And the result is sent back in the registers.

One more function related to the FCB is set dta address function which is saved by the executioner and also sent to the launcher at that time and sent to the original handler . This address is set by default to psp:0080 (hex) of the running program in start the address which represents the default FCB used by the program.

The functions related to. the disk and directory are simple to handle. If the function is set disk and the demanded disk is the redirected_disk, the variable is_remote_disk_default is set to 1 and the success is returned in the appropriate register. Otherwise the original function is called and if that is a success then the variable is_remote_disk_default is set to 0. If the function is get current disk and the is_remote_disk_default is 1, the drive returned is that of the redirected drive. Other functions are similarly detected to find the disk on which it has to be operated and then either the call is redirected to the launcher or to original handler. The response from the launcher is copied to the appropriate buffer in every call.

6.3.3.4. Submitting the exit code of program and again going into waiting mode:

If the executing programs exits then the launcher gets back the control. It then determines the exit status of the running program and send it to the launcher in the ending execution packet.

After this step the executioner tears down the session by issuing a NetBIOS hangup command, restores the address of all the interrupt handlers and again starts waiting for other request from any of the launchers.

Chapter Seven

Results and Future Extensions

7.1. Results:

After the implementation of the program it was tested for the following situations:

1. The launcher was loaded and was submitted a program which was on the executioner's drive. The program run correctly and the output and the input of the program was redirected to the launcher.

2. The launcher submitted a program which was on the launcher's drive and this also run in usual fashion.

3. The launched program tried to open files on the launcher's disk and it was successful.

4. The command interpreter was loaded on the system and all the copy delete operations were tried for both the drive on executioner as well as the drive on the launcher's machine and this was a success.

5. A program was loaded on the system and the control+c was used to terminate the program successfully.

6. A program with its output redirected to the file on the launcher's drive was opened and it was a success.

7. A terminate and stay resident program was loaded and this made the system on the executioner and the launcher to hang.

7.2. Future Scope

The future extension of this work may be done in the following direction.

1. The first obvious extension to the system will be to port the system to windows.

2. Again the communication is done using different protocol layers the presentation layer may be changed to make it compatible with the UNIX system in which the layer may be used to map system call of one platform to other's. This may be used to run a DOS or windows session from UNIX system (UNIX system also has the capability to support NetBIOS protocol)

3. The system may be modified to support sector level input output for the remote drive.

4. Higher level memory may be used by the system to conserve the conventional memory in which the DOS programs are run.

5. This system may be extended to run NetPC which connected to a powerful server machine having a multitasking operating system like windows or windows 95 and running programs on behalf of the NetPC.

6. The system may be modified to run a terminate and stay resident program also.

APPENDIX -A

LISTING OF HEADER FILE NETBIOS.H

```
/* Netbios.h    */

#ifndef _NETBIOS__H
#define _NETBIOS__H

typedef unsigned char byte;
typedef unsigned int word;

typedef struct
{
        byte NCB_COMMAND;
        byte NCB_RETCODE;
        byte NCB_LSN;
        byte NCB_NUM;
        void far *NCB_BUFFER_PTR;
        word NCB_LENGTH;
        byte NCB_CALLNAME[16];
        byte NCB_NAME[16];
        byte NCB_RTO;
        byte NCB_STO;
        void interrupt(*POST_FUNC)(void);
        byte NCB_LANA_NUM;
        byte NCB_CMD_CPLT;
        byte NCB_RESERVE[14];
}NCB;

char  *net_error_message[]={
        "success",                  /*    00    */
        "invalid buffer length",    /*    01    */
        "ret code 02",              /*    02    */
        "invalid command",          /*    03    */
        "ret code 04",              /*    04    */
        "timed out",                /*    05    */
        "buffer too small",         /*    06    */
        "ret code 07",              /*    07    */
        "invalid session no",       /*    08    */
        "no resource",              /*    09    */
        "session closed",           /*    0A    */
        "command cancelled",        /*    0B    */
        "ret code 0C",              /*    0C    */
        "dupl. local name",         /*    0D    */
        "name table full ",         /*    0E    */
        "active session",          /*   0F   */
        "ret code 10",             /*   10   */
        "session table full",      /*   11   */
        "no one listening",        /*   12   */
        "invalid name number",     /*   13   */
        "no answer",               /*   14   */
        "no local name",           /*   15   */
        "name in use",             /*   16   */
        "name is deleted",         /*   17   */
        "abnormal end",            /*   18   */
        "name conflict",           /*   19   */
        "ret code 1A",             /*   1A   */
        "ret code 1B",             /*   1B   */
        "ret code 1C",             /*   1C   */
        "ret code 1D",             /*   1D   */
        "ret code 1E",             /*   1E   */
        "ret code 1F",             /*   1F   */
        "ret code 20",             /*   20   */
        "card busy",               /*   21   */
        "too many cmds",           /*   22   */
        "invalid card num",        /*   23   */
        "cancel done",             /*   24   */
        "ret code 25",             /*   25   */
        "cannot cancel"            /*   26   */
}
#define         RESET                         0x32
#define         CANCEL                        0x35
#define         STATUS                        0xb3
#define         STATUS_WAIT                   0x33
#define         TRACE                         0xf9
#define         TRACE_WAIT                    0x79
#define         UNLINK                        0x70
#define         ADD_NAME                      0xb0
#define         ADD_NAME_WAIT                 0x30
#define         ADD_GROUP_NAME                0xb6
#define         ADD_GROUP_NAME_WAIT           0x36
#define         DELETE_NAME                   0xb1
#define         DELETE_NAME_WAIT              0x31
#define         CALL                          0x90
#define         CALL_WAIT                     0x10
#define         LISTEN                        0x91
#define         LISTEN_WAIT                   0x11
#define         HANG_UP                       0x92
#define         HANG_UP_WAIT                  0x12
#define         SEND                          0x94
#define         SEND_WAIT                     0x14
#define         SEND_NO_ACK                   0xf1
#define         SEND_NO_ACK_WAIT              0x71
```

```
#define         CHAIN_SEND                    0x97
#define         CHAIN_SEND_WAIT               0x17
#define         CHAIN_SEND_NO_ACK             0xf2
#define         CHAIN_SEND_NO_ACK_WAIT        0x72
#define         RECEIVE                       0x95
#define         RECEIVE_WAIT                  0x15
#define         RECEIVE_ANY                   0x96
#define         RECEIVE_ANY_WAIT              0x16
#define         SESSION_STATUS                0xb4
#define         SESSION_STATUS_WAIT           0x34
#define         SEND_DATAGRAM                 0xa0
#define         SEND_DATAGRAM_WAIT            0x20
#define         SEND_BCST_DATAGRAM            0xa2
#define         SEND_BCST_DATAGRAM_WAIT       0x22
#define         RECEIVE_DATAGRAM              0xa1
#define         RECEIVE_DATAGRAM_WAIT         0x21
#define         RECEIVE_BCST_DATAGRAM         0xa3
#define         RECEIVE_BCST_DATAGRAM_WAIT 0x23

#endif
```

APPENDIX B

MS DOS INTERRUPT 21 reference

- Input Output related functions
- FCB related Int21 functions
- Handle related Int21 functions
- Directory related calls

B.1. INPUT OUTPUT Related Functions:

The various interrupt 21 functions which are related to the output produced on the screen are (the function no is the number to be set in the AH register while executing the interrupt):

1. function 02 (hex) : It is used to print a character at the current cursor position on the screen. The cursor position is advanced after printing the character.

2. function 06 (hex) : It is used to write all possible characters and control codes without the interference from the operating system. The character to be printed must be between 0 (hex) and 0FE (hex).

3. function 09 (hex) : It is used to display string on the screen or the standard output. The string must be terminated with the character '$'. It means that this function prints all the characters in the buffer till the character '$' and so it can not print the character '$' itself. A control-C at the keyboard while using this function calls control break handler which is actually int 23 (hex) set by the operating system.

The various INT 21 (hex) functions which are used for input by a program are:

1. function 01 (hex) : It is used to input a character from the key board. The function echoes the character to the display. When no character is ready in the keyboard buffer, it waits for one to be available.

2. function 06 (hex) : It is used for direct console input. The function represent an input when the value in DL register is equal to 0FF (hex).

3. function 07 (hex) : It reads a character from standard input device without echoing it to the standard output device. It waits if no character is available at the keyboard.

4. function 08 (hex) : It is similar to the function 07.

5. function 0A (hex) : It reads a string of bytes from the standard input device up to and including an ASCII carriage return and places the character read in a user designated buffer. The characters are echoed to the standard output device.

6. function 0B (hex) : It checks whether a character is available from the standard input device.

7. function 0C (hex) : It clears the standard input buffer and then invokes one of the character input function which is specified. It is used to flush the MS DOS type ahead buffer.

In these functions, 01 (hex), 08(hex), 0A (hex),0B (hex) and 0C (hex) are control-C sensitive. It means that if control+C is pressed at the time these active, then the control+break handler of the DOS is invoked. The 07 (hex) and 08 (hex) although similar in other respect differ in this property only.

B.2. FCB Related Int 21 Calls

The most commonly file related Int 21 (hex)operation used with FCB are:

1. function 0F (hex) : opens a file and makes it available for subsequent read and write operations.

2. function 10 (hex) : closes a file and flushes all the internal MS DOS disk buffers associated with the file to the disk and updates the directory entry if the file has been modified.

3. function 11 (hex) : It searches the current directory on the designated drive for a matching filename.

4. function 12 (hex) : Given that a previous call to INT 21 function 11 (hex) has been successful, returns the next matching filename (if any). It should be used with the above call only.

5. function 13 (hex) : It delete all the matching files from the current directory on the default or specified disk drive.

6. function 14 (hex) : It reads the next sequential block of data from a file and then increments the file pointer appropriately.

7. function 15 (hex) : It is used for writing next sequential block of data into a file. It then increments the file pointer appropriately.

8. function 16 (hex) : It creates a new directory entry in the current directory or truncates any existing file with the same name to zero length and opens it for subsequent read/write operations.

9. function 17 (hex) : It is used to rename file in the current directory on the disk in the specified drive.

10. function 1A (hex):. sets the address of the DTA (disk transfer area) for subsequent FCB related function calls.

11. function 21 (hex): used for reading a selected record in a file into memory.

12. function 22 (hex): writes data from memory into a selected record in the disk.

13. function 23 (hex): It is used for getting the file size.

14. function 24 (hex): Sets the relative record number field of a FCB to correspond to the current file position as recorded in the opened FCB.

15. function 27 (hex): reads one or more sequential records from a file into memory, starting at a designated file location.

16. function 28 (hex): writes one or more sequential records from memory to a file, starting at a designated file location.

17. function 29 (hex): parses a text string into various fields of an FCB to be used afterwards.

18. function 2F (hex): It is used to get the current DTA address.

B.3. Handle related Int 21 calls

The various handle related functions calls (Int 21 (hex) functions) are:

1. function 3C (hex) : Creates a new file in the designated or default directory in the designated or default drive. If the specified file exists then it is truncated to zero length. In any case the file is opened and a handle is returned.

2. function 3D (hex) : It opens the specified file and returns a handle to it.

3. function 3E (hex) : It closes the file associated with the handle. All the data in the MS DOS disk buffer related to the file are flushed to the disk and the directory entry is updated.

4. function 3F (hex) : Given a valid file handle from a previous open or create operation, this transfers specified bytes of data from the file at the current file pointer position to the designated buffer whose address is passed to the function. It also updates the file pointer position.

5. function 40 (hex) : It is used to transfer data from the specified buffer into the file. It also updates the file pointer position.

6. function 41 (hex) : It is used to delete a file.

7. function 42 (hex) : set file pointer relative to start of file, end of file or the current file pointer position.

8. function 43 (hex) : get or alters the attributes of a file or directory.

9. function 44 (hex) : provides input/output control (IOCTL) for devices from the application program.

10. function 45 (hex): duplicates handle for a currently open file or device.

11. function 46 (hex): Given two handles, it redirects a handle to point to the same device or file at the same position as the second handle.

12. function 4E (hex): Used for searching a matching file in a specified or default directory in the specified or default disk.

13. function 4F (hex): Used with the above function to find next matching file.

14. function 56 (hex): used for renaming file.

15. function 57 (hex): used for getting or setting file data and time.

16. function 5A (hex): used for creating a new file with a unique name in the specified or default directory on the specified or default disk.

17. function 5B (hex): creates a new file. fails if file exists.

18. function 5C (hex): locks or unlocks the specified region of a file.

19. function 67 (hex): used to set the maximum handle count for the application.

20. function 68 (hex): commits file i.e., forces all the internal buffer of the operating system to be physically written to the file or device.

21. function 6C (hex): opens, creates or replaces file and returns a handle that can be used. It combines the capabilities of function 3CH, 3DH and 5BH.

B.4. Directory And Disk Related Calls

The functions related to the directory tree are :

1. function 0E (hex) : used to select the current drive.

2. function 19 (hex) : get current drive.

3. function 39 (hex) : create a directory.

4. function 3A (hex) : remove directory.

5. function 3B (hex) : select current directory.

6. function 47 (hex) : get current directory.

7. function 0D (hex) : used to reset disk by flushing all the file buffers.

8. function 1B (hex) : obtains selected information about the current disk drive.

9. function 1C (hex) : similar to 1 BH but can be used for other disk than default.

10. function 36 (hex): obtains selected information about a disk drive, from which drive capacities and the remaining free space can be calculated.

BIBLIOGRAPHY

BOOKS AND JOURNALS

1. Tanenbaum, Andrew S., *Computer Networks* , Prentice Hall Of India, New Delhi, 1993.

2. Stevens, W. Richard., *UNIX NETWORK PROGRAMMING,* Prentice Hall Of India, New Delhi, 1996.

3. Comer, Douglas E., *Internetworking with TCP/IP, Volume 1, Principles, Protocols and Architecture,* Prentice Hall Of India, New Delhi, 1996.

4. Stallings, William., *DATA AND COMPUTER COMMUNICATIONS,* Prentice Hall Of India, New Delhi, 1996.

5. Abel, Peter., *IBM PC ASSEMBLY LANGUAGE AND PROGRAMMING,* Prentice Hall Of India, New Delhi, 1993.

6. Duncan, R., *ADVANCED MS DOS PROGRAMMING,* Microsoft Press, 1994.

7. Nance, Barry., *NETWORK PROGRAMMING IN C,* Prentice Hall Of India, New Delhi, 1996.

8. Schildt, H., *TURBO C/C++, A COMPLETE REFERENCE,* Osborne McGraw-Hill Publication, 1994.