Uttar Pradesh Rajarshi Tandon
Open University

Bachelor of Computer Application

BCA-E7
Network Programming

Uttar Pradesh Rajarshi Tandon
Open University

Bachelor of Computer Application

BCA-E7

Network Programming

Block

1

© **UPRTOU, Prayagraj. 2019**
ISBN : 978-93-83328-11-6

Printed and Published by Dr. Arun Kumar Gupta Registrar, Uttar Pradesh Rajarshi Tandon Open University, 2018.

[TESLA-010]
BCA-E7/4

Printed By : Chandrakala Universal Pvt. Ltd. 42/7 Jawahar Lal Neharu Road, Prayagraj.

BLOCK INTRODUCTION

Unit 1: This unit deals with introduction to network programming. It contains introduction to OSI model, UNIX standards. This unit explains TCP and UDP. This unit tells how to establish the connection and termination of connection in TCP. In this unit, you will learn about buffer sizes and its limitations and standard internet services. The protocol usages by common internet application is described in this unit.

Unit 2: This unit deals with elementary sockets. In this unit, you will learn about address structure, value-result arguments, byte ordering and manipulation functions and related functions.

Unit 3: This unit deals with elementary TCP sockets. This unit discusses about socket, connect, bind, listen, accept, fork and close functions. In this unit, you will learn about concurrent servers.

Unit 4: This unit deals with TCP client/server. This unit tells about TCP Echo server function, Normal start-up. In this unit, you will learn about signal handling server process termination, crashing and rebooting of server host and shutdown of server host.

UNIT-1 : INTRODUCTION TO NETWORK PROGRAMMING

Structure

1.0 INTRODUCTION

In this unit, the focus is to provide a basic understanding of the technical design and architecture of the Internet using two different models OSI and TCP/IP. The background of Unix standards, IEEE POSIX and The Open Group's Technical Standard designation that were later converged into The Single Unix Specification Version is discussed.

Most client/server applications use either TCP or UDP as their transport layer for which TCP connection establishment and termination are discussed in detail along with description of Ipv4, Ipv6 and their buffer size limitations.

We cover various topics in this unit that fall into this category: TCP's three-way handshake, TCP's connection termination sequence, plus TCP, and UDP buffering by the socket layer, and so on.

1.1 OBJECTIVES

After the end of this unit, you should be able to:

- Understand the OSI Model and its various communication processes
- Gain insights regarding the various UNIX standardization schemes
- Differentiate between the TCP and UDP protocols

- Know about various buffer sizes, standard internet services and popular protocols.

1.2 OSI MODEL

OSI Model is an abstract model used to understand a wide range of network architecture. It was proposed as a general approach to network models to standardize the communication functions of a telecommunication or computing system.

The OSI model has seven layers; Starting at the bottom (nearest the physical connections), the layers are: (1) Physical, (2) Data Link, (3) Network, (4) Transport, (5) Session, (6) Presentation, and (7) Application.

In the OSI model, control is passed from one layer to the next, starting at the application layer in one station, and proceeding to the bottom layer, over the channel to the next station and back up the hierarchy.

We will look at each layer in the OSI model in turn, starting with the Physical layer. Figure 1.1 shows the layer architecture of OSI model.

7	APPLICATION LAYER
6	PRESENTATION LAYER
5	SESSION LAYER
4	TRANSPORT LAYER
3	NETWORK LAYER
2	DATA LINK LAYER
1	PHYSICAL LAYER

Figure 1.1 OSI model

- **PHYSICAL LAYER**

  The OSI Physical layer deals with the physical attributes of the actual wired, wireless, fiber optic, or other connection that is used to transport data across a single link. It deals with transmission and reception of the unstructured raw bit stream- electrical impulse, light or radio signal over a physical medium.

  It provides the hardware means of sending and receiving data on a carrier, including defining cables, cards and physical aspects. It also does Bit encoding for faster data transmission. Fast Ethernet, RS232, and ATM are protocols with physical layer components.

- **DATA LINK LAYER**

  The data link layer provides error-free transfer of data frames from one node to another over the physical layer, allowing layers above it to have error-free transmission over the link. Following are some of the functions of Data link layer:

  1. Logical Link establishment between nodes.

  2. Controls Frame traffic by telling transmitting node to "back-off" when no frame buffers are available.

  3. Sequential transfer of frames by defining special sequences to indicate the beginning and end of each packet.

  4. Media access management by defining addresses to the stations communicating.

  5. Frame error checking using Checksum.

  6. Frame acknowledgment and retransmission of non - acknowledged frames and handling duplicate frame receipt.

- **NETWORK LAYER**

  The network layer governs how routers forward packets across multiple hops to get from their source to their destination. It deals with assigning global "routable" addresses to the various systems connected to the network.

  It is also responsible for subnet traffic control instructing a sending station to "throttle back" its frame transmission when the router's buffer fills up. Other functions include:

  1. Frame fragmentation and reassembly at destination station if router's Maximum transmission unit (MTU) is less than frame size.

  2. Logical-physical address mapping that involves translating logical addresses, or names, into physical addresses.

  3. Keeps track of frames forwarded by subnet intermediate systems, to produce billing information.

- **TRANSPORT LAYER**

  The transport layer ensures that messages are delivered error-free, in sequence, and with no losses or duplications. It manages packet loss and retransmission as well as flow control and window size. Services of Transport layer depend upon the services offered by the Network layer. Some of the functions of transport layer are:

  1. Session multiplexing: Multiplex several message streams, or sessions onto one logical link.

2. End to end message delivery with acknowledgements.

3. Message segmentation: accept message from the (session) layer above it, splits the message into smaller units (if not already small enough), and passes the smaller units down to the network layer.

- **SESSION LAYER**

  The OSI Session layer handles establishing connections between processes running on different stations. It allows two application processes on different machines to establish, use and terminate a connection, called a session. It also provides support to the sessions established.

- **PRESENTATION LAYER**

  The presentation layer formats the data to be presented to the application layer. Like a translator, it translates data from a format used by the application layer into a common format at the sending station, then translate the common format to a format known to the application layer at the receiving station. The key functions of the presentation layer are:

  Data compression, Data encryption/decryption, Character code translation: for example, ASCII to EBCDIC etc.

- **APPLICATION LAYER**

  The application layer allows users and application processes to access network services. The layer is responsible for functions like Electronic messaging (such as mail), Remote printer access, Remote file access, Network management, Inter-process communication, Directory services etc.

  The applications can be Client applications that initiate connection or server applications that respond to incoming connection and serve them.

1.3 UNIX STANDARDS

The most interesting Unix standardization activity was being done by The Austin Common Standards Revision Group (CSRG) that produced roughly 4,000 pages of specifications that carry both the IEEE POSIX designation as well as The Open Group's Technical Standard designation, thus leading to multiple names to same standards, for example, ISO/IEC 9945:2002, IEEE Std 1003.1 -2001, and the Single Unix Specification Version 3 are various names of same standard, The POSIX Specification.

Background on POSIX

POSIX is an acronym for Portable Operating System Interface. POSIX is not a single standard, but a set of standards being developed by the Institute for Electrical and Electronics Engineers, Inc., normally called the IEEE. The POSIX standards have also been adopted as international standards by ISO and the International Electrotechnical Commission (IEC), called ISO/IEC.

The interesting history of POSIX standards has been covered only briefly here:

- **IEEE Std 1003.1–1988** (317 pages) was the first POSIX standard. It specified the C language interface into a Unix-like kernel and covered the following areas: process primitives (***fork, exec***, signals, and timers), the environment of a process (user IDs and process groups), files and directories (all the I/O functions), terminal I/O, system databases (password file and group file), and the *tar* and *cpio* archive formats.

The first POSIX standard was a trial-use version in 1986 known as "IEEE-IX." The name "POSIX" was suggested by Richard Stallman.

- **IEEE Std 1003.1–1990** (356 pages) was next, and it was also known as ISO/IEC 9945–1: 1990. Minimal changes were made from the 1988 to the 1990 version. Appended to the title was "Part 1: System Application Program Interface (API) [C Language]," indicating that this standard was the C language API.

- **IEEE Std 1003.2–1992** came next in two volumes (about 1,300 pages). Its title contained "Part 2: Shell and Utilities." This part defined the shell (based on the System V Bourne shell) and about 100 utilities (programs normally executed from a shell, from ***awk*** and basename to ***vi*** and ***yacc***). Throughout this text, we will refer to this standard as POSIX.2.

- **IEEE Std 1003.1b–1993** (590 pages) was originally known as IEEE P1003.4. This was an update to the 1003.1–1990 standard to include the real-time extensions developed by the P1003.4 working group. The 1003.1b–1993 standard added the following items to the 1990 standard: file synchronization, asynchronous I/O, semaphores, memory management (mmap and shared memory), execution scheduling, clocks and timers, and message queues.

- **IEEE Std 1003.1, 1996** Edition [IEEE 1996] (743 pages) came next and included 1003.1–1990 (the base API), 1003.1b –1993 (real-time extensions), 1003.1c –1995 (pthreads), and 1003.1i – 1995 (technical corrections to 1003.1b). This standard was also called ISO/IEC 9945–1: 1996. Three units on threads were added, along with additional sections on thread synchronization (mutexes and condition variables), thread scheduling, and synchronization scheduling. Throughout this text, we will refer to this standard as POSIX.1.

  This standard also contains a Foreword stating that ISO/IEC 9945 consists of the following parts:

Part 1: System API (C language)

Part 2: Shell and utilities

Part 3: System administration (under development) Parts 1 and 2 are what we call POSIX.1 and POSIX.2

- **IEEE Std 1003.1g:** Protocol-independent interfaces (PII) became an approved standard in 2000. Until the introduction of The Single Unix Specification Version 3, this POSIX work was the most relevant to the topics covered in this book. This is the networking API standard and it defines two APIs, which it calls Detailed Network Interfaces (DNIs): 1. DNI/Socket, based on the 4.4BSD sockets API 2. DNI/XTI, based on the X/Open XPG4 specification Work on this standard started in the late 1980s as the P1003.12 working group (later renamed P1003.1g). Throughout this text, we will refer to this standard as POSIX.1g.

Background on The Open Group

The Open Group was formed in 1996 by the consolidation of the X/Open Company (founded in 1984) and the Open Software Foundation (OSF, founded in 1988). It is an international consortium of vendors and end-user customers from industry, government, and academia. Here is a brief background on the standards they produced:

- X/Open published the X/Open Portability Guide, Issue 3 (XPG3) in 1989.

- Issue 4 was published in 1992, followed by Issue 4, Version 2 in 1994. This latest version was also known as "Spec 1170," with the magic number 1,170 being the sum of the number of system interfaces (926), the number of headers (70), and the number of commands (174). The latest name for this set of specifications is the "X/Open Single Unix Specification," although it is also called "Unix 95."

- In March 1997, Version 2 of the Single Unix Specification was announced. Products conforming to this specification were called "Unix 98." We will refer to this specification as just "UNIX 98" throughout this text. The number of interfaces required by Unix 98 increases from 1,170 to 1,434, although for a workstation this jumps to 3,030, because it includes the Common Desktop Environment (CDE), which in turn requires the X Window System and the Motif user interface. Details are available in [Josey 1997] and at http://www.UNIX.org/version2. The networking services that are part of Unix 98 are defined for both the sockets and XTI APIs. This specification is nearly identical to POSIX.1g.

Unification of Standards

Now, Most Unix systems today conform to some version of POSIX.1 and POSIX.2; many comply with The Single Unix Specification Version 3. The focus of this book is on The Single Unix Specification Version 3, with our main focus on the sockets API.

Internet Engineering Task Force (IETF)

The Internet Engineering Task Force (IETF) is a large, open, international community of network designers, operators, vendors, and researchers concerned with the evolution of the Internet architecture and the smooth operation of the Internet. It is open to any interested individual. The Internet standards process is documented in RFC 2026 [Bradner 1996]. Internet standards normally deal with protocol issues and not with programming APIs.

Nevertheless, two RFCs (RFC 3493 [Gilligan et al. 2003] and RFC 3542 [Stevens et al. 2003]) specify the sockets API for IPv6. These are informational RFCs, not standards, and were produced to speed the deployment of portable applications by the numerous vendors working on early releases of IPv6. Although standards bodies tend to take a long time, many APIs were standardized in The Single Unix Specification Version 3.

Check your progress

1. What are the responsibilities of network layer and transport layer?
2. Explain the connection establishment phase of the TCP protocol.

1.4 TCP AND UDP & TCP CONNECTION ESTABLISHMENT AND FORMAT

This section focuses on the following transport layer protocols: TCP and UDP.

Most client/server applications use either TCP or UDP. Another protocol SCTP is a newer protocol, originally designed for transport of telephony signalling across the Internet. These transport protocols use the network-layer protocol IP, either IPv4 or Ipv6. It is possible for an application to bypass the transport layer and use IPv4 or IPv6 directly. This is called a raw socket.

UDP is a simple, unreliable datagram protocol, while TCP is a sophisticated, reliable byte stream protocol. Let us look into both the protocols in detail.

User Datagram Protocol (UDP)

UDP is a connectionless protocol, and UDP sockets are an example of datagram sockets. There is no guarantee that UDP datagrams ever reach their intended destination. The application sends message to a UDP socket, encapsulated in a UDP datagram, which is then further encapsulated as an IP datagram, which is then sent to its destination. For a UDP datagram reaching its final destination, that order will be preserved across the network, or that datagrams arrive only once is not guaranteed.

Lack of reliability is the drawback we have with network programming with UDP. If a UDP Datagram does not reach its destination or is dropped midway, there is no scope of automatic retransmission.

Each UDP datagram has a length. The length of a datagram is passed to the receiving application along with the data. Being a connectionless service, there need not be any long-term relationship between a UDP client and server. For example, a UDP client can create a socket and send a datagram to a given server and then immediately send another datagram on the same socket to a different server. Similarly, a UDP server can receive several datagrams on a single UDP socket, each from a different client.

Transmission Control Protocol (TCP)

TCP is described in RFC 793 [Postel 1981c], and updated by RFC 1323 [Jacobson, Braden, and Borman 1992], RFC 2581 [Allman, Paxson, and Stevens 1999], RFC 2988 [Paxson and Allman 2000], and RFC 3390 [Allman, Floyd, and Partridge 2002].

TCP is a connection oriented protocol and provides connections between clients and servers. A TCP client establishes a connection with a given server, exchanges data with that server across the connection, and then terminates the connection.

It provides **reliability** by using acknowledgement in return and if not received retransmitting the data and waiting for a longer duration of time. It does not provide the guarantee to deliver data at the destination. Just delivering data if it can be delivered to a notification to end user if data cannot be sent.

The waiting time for acknowledgement or Roundtrip time(RTT) between Client and server is estimated by the algorithms in TCP.

TCP also sequences the data by associating a sequence number with every byte that it sends. For example, assume an application writes 2,048 bytes to a TCP socket, causing TCP to send two segments, the first containing the data with sequence numbers 1–1,024 and the second containing the data with sequence numbers 1,025–2,048. (A segment is the unit of data that TCP passes to IP.) If the segments arrive out of order, the receiving TCP will reorder the two segments based on their sequence numbers before passing the data to the receiving application. Thus TCP can detect a duplicate data from the sequencing and can discard it.

TCP provides **flow control.** TCP has the advertised window which tells peer how many bytes of data it can accept. It guarantees that the sender cannot overflow the receiving buffer. The window changes dynamically over time: As data is received from the sender, the window size decreases, but as the receiving application reads data from the buffer, the window size increases. It is possible for the window to reach 0: when TCP's receive buffer for a socket is full and it must wait for the application to read data from the buffer before it can take any more data from the peer.

Finally, a TCP connection is **full-duplex**. This means that an application can send and receive data in both directions on a given connection at any time. This means that TCP must keep track of state information such as sequence numbers and window sizes for each direction of data flow: sending and receiving.

TCP Connection Establishment and Termination

Let us understand how TCP connections are established and terminated, and TCP's state transition diagram.

Three-Way Handshake

Figure 1.2 shows the connection establishment of TCP by three-way handshaking.

1. Host A sends a connection request to host B by setting the SYN (a *synchronize* message, used to initiate and establish a connection) bit. Host A also registers its initial sequence number to use (Seq_no fl x).

2. Host B acknowledges the request by setting the ACK (an *acknowledgment*) bit and indicating the next data byte to receive (Ack_no fl x + 1). The "plus one" is needed because the SYN bit consumes one sequence number. At the same time, host B also sends a request by setting the SYN bit and registering its initial sequence number to use (Seq_no fl y).

3. Host A acknowledges the request from B by setting the ACK bit and confirming the next data byte to receive (Ack_no fl y + 1). Note that the sequence number is set to x + 1. On receipt at B the connection is established.

   If during a connection establishment phase, one of the hosts decides to refuse a connection request, it will send a reset segment by setting the RST bit. Each SYN message can specify options such as maximum segment size, window scal- ing, and timestamps. Because TCP segments can be delayed, lost, and duplicated, the initial sequence number should be different each time a host requests a connection.

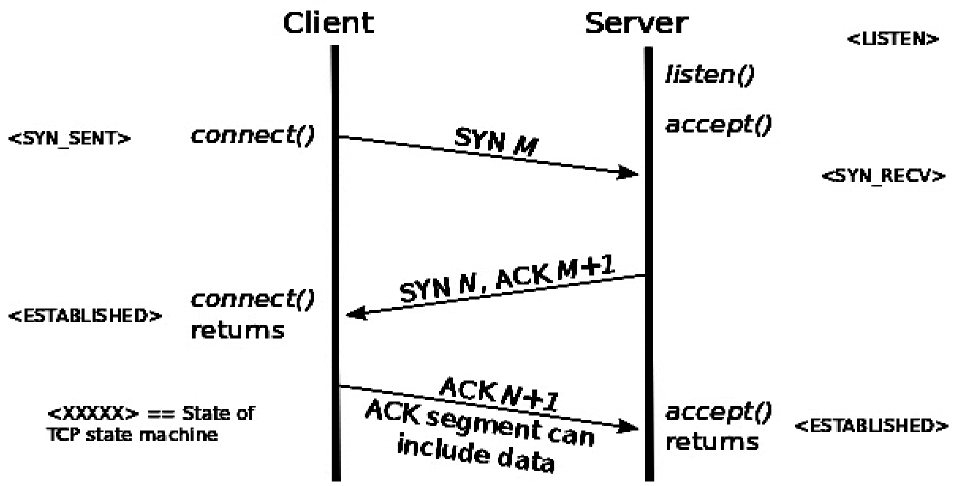


Figure 1.2: TCP Three Way Handshake

TCP Connection Termination

Figure 1.3 shows the TCP connection termination.

1. One application calls close first, and we say that this end performs the active close. This end's TCP sends a FIN segment, which means it is finished sending data.

2. The other end that receives the FIN performs the passive close. The received FIN is acknowledged by TCP. The receipt of the FIN is also passed to the application as an end-of-file (after any data that may have already been queued for the application to receive), since the receipt of the FIN means the application will not receive any additional data on the connection.

3. Sometime later, the application that received the end-of-file will close its socket. This causes its TCP to send a FIN.

4. The TCP on the system that receives this final FIN (the end that did the active close) acknowledges the FIN.

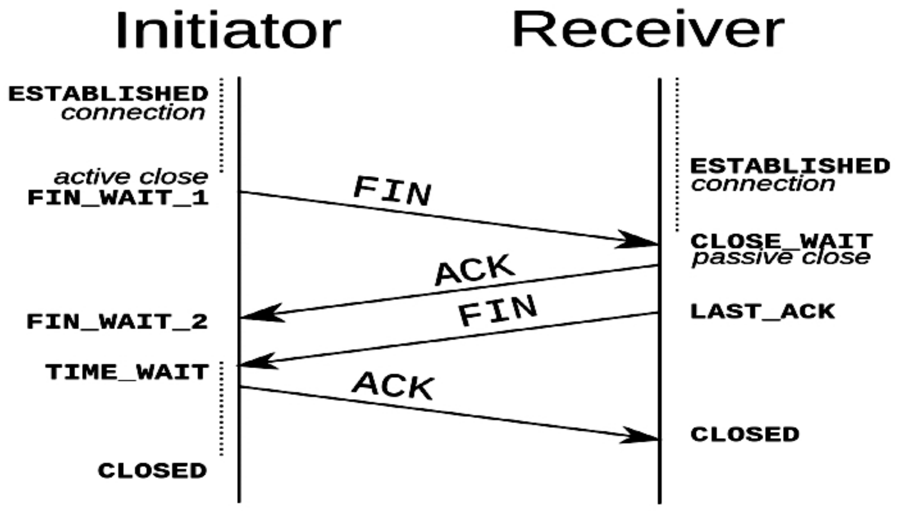


Figure 1.3: TCP connection termination

TCP State Transition Diagram

The ope ration of T CP w ith r egard t o c onnection e stablishment a nd connection termination can be specified with a state transition diagram as shown in Figure 1.4.

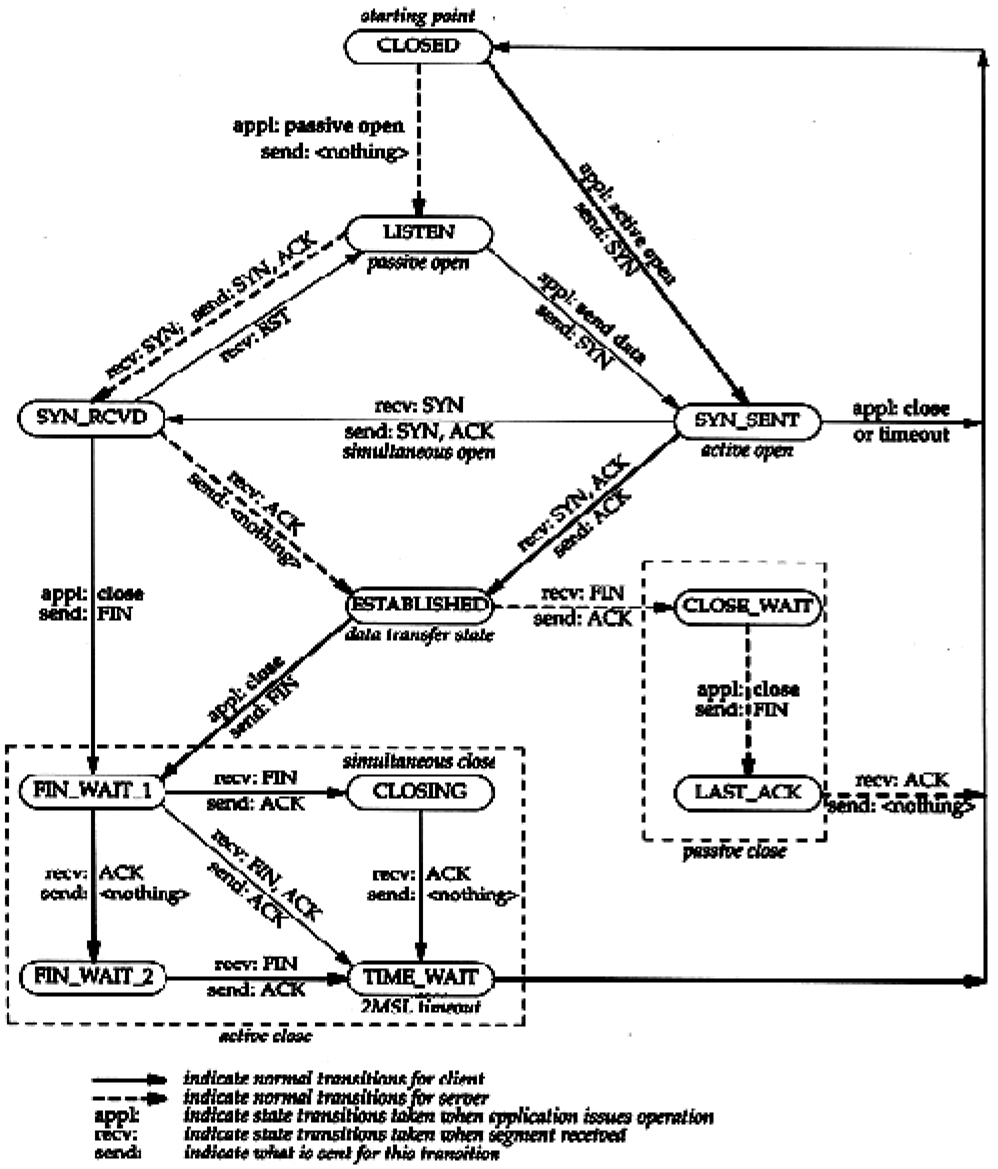


Figure 1.4: State transition diagram

A c onnection pr ogresses through a s eries of s tates dur ing i ts l ifetime a nd transition f rom s tate to state is b ased o n th at c urrent s tate a nd segment received in that state.

The s tates are: LISTEN, S YN-SENT, S YNRECEIVED, ES TABLISHED, FIN-WAIT-1, F IN-WAIT-2, C LOSE-WAIT, C LOSING, LAST-ACK, TIME-WAIT, and the fictional state CLOSED.

LISTEN represents waiting for a connection request from any remote TCP and port.

SYN-SENT represents waiting for a matching connection request after having sent a connection request. SYN-

RECEIVED represents waiting for a confirming connection request acknowledgment after having both received and sent a connection request.

ESTABLISHED represents an open connection, data received can be delivered to the user. The normal state for the data transfer phase of the connection.

FIN-WAIT-1 represents waiting for a connection termination request from the remote TCP, or an acknowledgment of the connection termination request previously sent.

FIN-WAIT-2 represents waiting for a connection termination request from the remote TCP.

CLOSE-WAIT represents waiting for a connection termination request from the local user.

CLOSING represents waiting for a connection termination request acknowledgment from the remote TCP.

LAST-ACK represents waiting for an acknowledgment of the connection termination request previously sent to the remote TCP (which includes an acknowledgment of its connection termination request).

TIME-WAIT represents waiting for enough time to pass to be sure the remote TCP received the acknowledgment of its connection termination request.

CLOSED represents no connection state at all.

1.5 BUFFER SIZES AND LIMITATION

The buffer sizes of IP Datagrams have certain limitations that affect the data an application can transmit. The limitations are as follows:

- IPv4 datagram has a maximum size of 65,535 bytes, including the IPv4 header. Its 16 bit total length field includes header size.
- IPv6 datagram has a maximum size of 65,575 bytes, including the 40-byte IPv6 header. Its 16 bit total length field does not include header size. On datalinks with a maximum transmission unit (MTU) that exceeds 65,535, IPv6 can have extended payload length field of 32 bits.
- MTU is dependent on Hardware, for example Ethernet MTU is 1500 bytes whereas Point to Point Protocol has configurable MTU. Minimum link MTU for Ipv4 is 68 bytes which means, Maximum sized header (20 bytes of fixed header, 40 bytes of options) + minimum sized fragment (8 bytes) can be passed. However, Minimum link MTU for Ipv6 is 1,280 bytes.
- The smallest MTU in the path between two hosts is called the **Path MTU**. For example, the Ethernet MTU of 1,500 bytes is the path

MTU. MTU between two hosts is different in both directions.

- IP Datagrams with size exceeding Link MTU are fragmented at the outgoing interface and reassembled at destination by both IPv4 and Ipv6. IPv4 hosts perform fragmentation on datagrams that they generate and IPv4 routers perform fragmentation on datagrams that they forward. For Ipv6, fragmentation of datagrams is performed only at Ipv6 hosts and not at Ipv6 routers with an exception of routers that generate their own datagrams instead of forwarding.

- Fragmentation fields are included in Ipv4 headers but not in Ipv6 headers. If "don't fragment" (DF) bit is set, it specifies that this datagram must not be fragmented, either by the sending host or by any router. A router that receives an IPv4 datagram with the DF bit set whose size exceeds the outgoing link's MTU generates an ICMPv4 " destination unreachable, fragmentation needed but DF bit set" error message. DF bit is implied with Ipv6 datagrams, so if a Ipv6 router receives a datagram whose size exceeds the outgoing link's MTU, it generates an ICMPv6 "packet too big" error message. This DF bit can also be used to discover the path MTU.

- TCP has a maximum segment size (MSS) that announces to the peer TCP the maximum amount of TCP data that the peer can send per segment. It tells the peer the actual value of the reassembly buffer size tries to avoid fragmentation. The MSS is often set to the interface MTU minus the fixed sizes of the IP and TCP headers.

- On an Ethernet using IPv4, MSS would be 1,460, and on an Ethernet using IPv6, this would be 1,440. (The TCP header is 20 bytes for both, but the IPv4 header is 20 bytes and the IPv6 header is 40 bytes.) The MSS value in the TCP MSS option is a 16-bit field, limiting the value to 65,535. This is fine for IPv4, since the maximum amount of TCP data in an IPv4 datagram is 65,495 (65,535 minus the 20-byte IPv4 header and minus the 20-byte TCP header).

TCP Output

When an application calls write, the kernel copies all the data from the application buffer into the TCP socket send buffer and returns only when the final byte in the application buffer has been copied into the socket send buffer. Insufficient room in the socket send buffer due to the larger application buffer size or socket send buffer already full, blocks the socket and process is put to sleep.

TCP transmits the data from buffer to peer TCP according to rules of data transmission and discards the data from the buffer only after receiving ACKs from the peer. Data is sent in MSS-sized chunks (announced by peer TCP) from TCP to IP with its header prepended to it. IP then prepends its header to the datagram, searches the appropriate routing table for destination IP and sends the datagram to proper datalink. IP performs fragmentation in case path MTU discovery (in newer implementations) not used or MSS option not used. The output queue associated with each datalink discards the packet and reports an error to TCP via IP in case it is full.

UDP Output

UDP s ocket h as a s end b uffer s ize (which w e can change w ith t he SO_SNDBUF s ocket opt ion), but t his i s s imply a n uppe r l imit on the maximum-sized U DP d atagram t hat can b e w ritten t o t he s ocket. I f an application w rites a d atagram l arger t han t he s ocket s end b uffer s ize, EMSGSIZE is returned. Since UDP is unreliable, it does not need to keep a copy of the application's data and does not need an actual send buffer.

UDP s imply pr epends i ts 8 -byte h eader an d p asses t he d atagram t o IP. IPv4 or IPv6 pr epends i ts he ader, d etermines t he out going i nterface b y performing the routing function, and then either adds the datagram to the datalink o utput q ueue (if it f its w ithin th e MTU) o r f ragments th e datagram and adds each fragment to the datalink output queue. If there is no room on the queue for the datagram or one of its fragments, ENOBUFS is often returned to the application.

1.6 STANDARD INTERNET SERVICES

Some of t he s tandard s ervices t hat a re pr ovided b y m ost implementations of TCP/IP are following:

Name	TCP Port	UDP Port	RFC	Description
Echo	7	7	862	Server returns whatever the client sends.
Discard	9	9	863	Server discards whatever the client sends.
Daytime	13	13	867	Server returns the time and date in human-readable format.
Chargen	19	19	864	TCP server sends a continual stream of characters, until the connection is terminated by the client. UDP server sends a datagram containing a random character (between 0 and 512) each time the client sends a datagram.
Time	37	37	868	Server returns the time as a 32-bit binary number. This number represents the number of seconds since midnight January 1, 1990, UTC.

Figure 1.5 Standard TCP/IP services provided by most implementations.

If we examine the port numbers for these standard services and other standard TCP/IP services (Telnet, FTP, SMTP, etc.), most are odd numbers. This is historical as these port numbers are derived from the NCP port numbers. (NCP, the Network Control Protocol, preceded TCP as a transport layer protocol for the ARPANET.) NCP was simplex, not full-duplex, so each application required two connections, and an even-odd pair of port numbers was reserved for each application. When TCP and UDP became the standard transport layers, only a single port number was needed per application, so the odd port numbers from NCP were used.

Often, *inetd* daemon provides these services on Unix hosts.

Check your progress

1. Explain some limitations of buffer sizes of IP datagrams.

2. Enlist Standard Internet Services provided by TCP/IP.

1.7 PORT NUMBERS

TCP and UDP identify applications using 16-bit port numbers called Port numbers that range between 1 and 1023. Servers are represented by their port numbers. For example, a TCP/IP implementation that provides FTP server provides that service TCP port 21, Telnet service is provided on Port 23, TFTP (the Trivial File Transfer Protocol) is on UDP port 69. The well-known ports are managed by the Internet Assigned Numbers Authority (IANA).

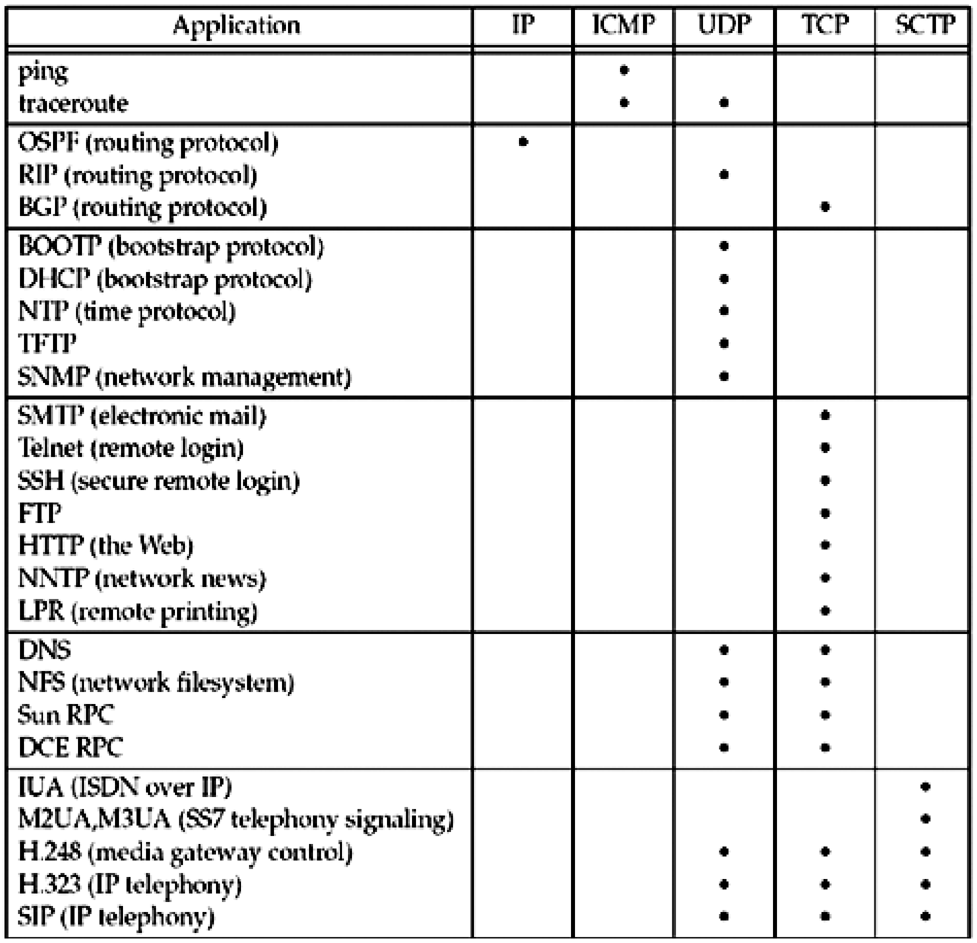
A client usually doesn't care what port number it uses on its end. All it needs to be certain of is that whatever port number it uses be unique on its host. Client port numbers are called ephemeral ports (i.e., short lived). This is because a client typically exists only as long as the user running the client needs its service, while servers typically run as long as the host is up.

The well-known port numbers are contained in the file /etc/services on most Unix systems. To find the port numbers for the Telnet server and the Domain Name System, we can execute

```
sun % grep telnet /etc/services

telnet 23/tcp                 says it uses TCP port 23

sun % grep domain /etc/services

domain 53/udp                 says it uses UDP port 53

domain 53/tcp         and TCP port 53.
```

Port numbers in the range of 1 to 1023 are reserved, and are used by some applications as part of the authentication between the client and server.

1.8 PROTOCOL USAGE BY COMMON INTERNET APPLICATIONS



Application	IP	ICMP	UDP	TCP	SCTP
ping		•			
traceroute		•	•		
OSPF (routing protocol)	•				
RIP (routing protocol)			•		
BGP (routing protocol)				•	
BOOTP (bootstrap protocol)			•		
DHCP (bootstrap protocol)			•		
NTP (time protocol)			•		
TFTP			•		
SNMP (network management)			•		
SMTP (electronic mail)				•	
Telnet (remote login)				•	
SSH (secure remote login)				•	
FTP				•	
HTTP (the Web)				•	
NNTP (network news)				•	
LPR (remote printing)				•	
DNS			•	•	
NFS (network filesystem)			•	•	
Sun RPC			•	•	
DCE RPC			•	•	
IUA (ISDN over IP)					•
M2UA,M3UA (SS7 telephony signaling)					•
H.248 (media gateway control)			•	•	•
H.323 (IP telephony)			•	•	•
SIP (IP telephony)			•	•	•

Figure 1.6: summarizes the protocol usage of various common Internet applications.

The first two applications, ping and traceroute, are diagnostic applications that use ICMP. traceroute builds its own UDP packets to send and reads ICMP replies. The three popular routing protocols demonstrate the variety of transport protocols used by routing protocols. OSPF uses IP directly, employing a raw socket, while RIP uses UDP and BGP uses TCP. The next five are UDP-based applications, followed by seven TCP applications and four that use both UDP and TCP. The final five are IP telephony applications that use SCTP exclusively or optionally UDP, TCP, or SCTP.

1.9 SUMMARY

The unit introduces many of the terms and concepts that shall be expanded on throughout the rest of the book. It also gives an overview of developing protocol-dependent programs.

TCP uses a three-way handshake for establishing connection while a connection is terminated using a four-packet exchange. When a TCP connection is established, the connection state is changed from CLOSED to ESTABLISHED, and upon termination, the state is changed to CLOSED. There are total 11 states in which a TCP connection may reside. A state transition diagram specifies the rules for switching between the states. Knowledge about the state transition diagram is necessary for understanding what happens when an application calls functions such as connect, accept, and close.

Unlike TCP, UDP doesn't establish a connection before sending data, it just sends. Because of this, UDP is called "Connectionless". UDP packets are often called "Datagrams". An example of UDP in action is the DNS service. DNS servers send and receive DNS requests using UDP.

UDP is a simple, connectionless, and unreliable protocol, while TCP is a complex, connection- oriented, and reliable. Although most of the applications on the Internet use TCP (the Web, Telnet, FTP, and email), there is a need for UDP as well. In further units, we shall discuss the reasons to choose UDP instead of TCP.

1.10 TERMINAL QUESTIONS

1. Discuss in detail the layers of OSI model.

2. Explain TCP/IP layering in detail with neat sketch?

3. Explain the TCP state transition diagram with the help of a diagram

4. TCP assumes an MSS of 536 if it does not receive an MSS option from the peer. Why is this value used?

5. UDP is a simple, connectionless, and unreliable protocol, while TCP is a complex, connection- oriented, and reliable. Explain the statement.

UNIT 2 : ELEMENTARY SOCKETS

Structure

2.0 INTRODUCTION

This unit focuses on the description of the sockets API. The socket address structures described here can be passed in two directions: from the process to the kernel, and from the kernel to the process. The latter case is an ex ample o f a v alue-result a rgument, a nd we w ill e ncounter other examples of these arguments throughout the text.

The socket address structure is created by converting a text representation of an address into the binary value using the address conversion functions. Most e xisting IPv4 c ode us es i net_addr a nd inet_ntoa, but t wo n ew functions, inet_pton and inet_ntop, handle both IPv4 and Ipv6.

2.1 OBJECTIVES

After the end of this unit, you should be able to:

- Analyse the problem and develop an algorithm for its solution;
- Represent an a lgorithm i n a n ab stract l anguage (eg. p seudo-code, Structure Diagrams);
- Represent an algorithm with the help of flowchart.
- Understand the fundamental principle of program design.

2.2 SOCKET ADDRESS STRUCTURES

Various s tructures a re us ed i n U nix S ocket P rogramming t o hol d information a bout t he address a nd por t, a nd ot her i nformation. M ost socket f unctions r equire a poi nter t o a s ocket address s tructure a s a n argument. Structures defined in this unit are related to Internet Protocol Family. The names of these structures begin with sockaddr_ and end with a unique suffix for each protocol suite.

The first structure is *sockaddr* that holds the socket information –

```
struct sockaddr {
unsigned short sa_family;
        char sa_data [14];
        };
```

Attribute	Values	Description
sa_family	AF_INET AF_UNIX AF_NS AF_IMPLINK	It represents an address family. In most of the Internet-based applications, we use AF_INET.
sa_data	Protocol-Specific Address	The content of the 14 bytes of protocol specific address are interpreted according to the type of address. For the Internet family, we will use port number IP address, which is represented by*sockaddr_in* structure defined below.

This is a generic socket address structure, which will be passed as reference in most of the socket function calls. But any socket function that takes them as pointers must be support socket address structures from any supported protocol families. This generic socket address structure is defined in <sys/socket.h> header.

The below function is an example of function taking pointer to the generic socket address structure.

```
#include <sys/socket.h>
int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
Returns: 0 if OK, –1 on error
```

Below example shows how the function is called:

```
struct sockaddr_in serv;                /*IPv4 socket address structure */
/* fill in serv{} */
bind (sockfd, (struct sockaddr *) &serv, sizeof(serv));
```

IPv4 Socket Address Structure

An IPv4 socket address structure, commonly called an "Internet socket address structure," is named *sockaddr_in* and is defined by including the

<netinet/in.h> he ader. Both t he IPv4 address a nd t he T CP o r U DP por t number a re always s tored i n t he s tructure i n network b yte or der. The internet (IPv4) s ocket a ddress s tructure: *sockaddr_in* has be en s hown below:

```
struct in_addr {
in_addr_t s_addr; /* 32-bit IPv4 address */
/* network byte ordered */
};
struct sockaddr_in {
uint8_t sin_len; /* length of structure (16) */
sa_family_t sin_family; /* AF_INET */
in_port_t sin_port; /* 16-bit TCP or UDP port number */
/* network byte ordered */
struct in_addr sin_addr; /* 32-bit IPv4 address */
/* network byte ordered */
char sin_zero [8]; /* unused */
};
```

We ne ed not s et length f ield e ven i f i t i s pr esent unl ess r outing s ockets come i nto pi cture. O nly ke rnels t hat de al w ith s ocket a ddress s tructures from various protocol families (e.g., the routing table code) use it. POSIX datatypes a re s hown f or the s_addr, s in_family, and s in_port m embers i n socket address.

The socket functions bind, connect, sendto, and sendmsg, that pass socket address s tructure a ll g o t hrough a n a dditional sockargs f untion w hich copies the structure from the process and sets its sin_member to the size of the structure being passed as the argument.

The functions a ccept, r ecvfrom, r ecvmsg, getpeername, a nd ge tsockname that pass socket address structure from the kernel to the process too set the sin_len member before returning to the process. The five socket functions that pass a s ocket address structure from the kernel to the process, accept, recvfrom, r ecvmsg, getpeername, an d getsockname, al l s et t he s in_len member before returning to the process.

The POSIX s pecification r equires o nly t hree m embers i n t he s tructure: sin_family, s in_addr, and s in_port. A P OSIX-compliant imp lementation can al so d efine ad ditional s tructure m embers, f or an Internet s ocket address structure. Almost all implementations add the sin_zero member so that all socket address structures are at least 16 bytes in size.

Datatype	Description	Header
int8_t	Signed 8-bit integer	<sys/types.h>
uint8_t	Unsigned 8-bit integer	<sys/types.h>
int16_t	Signed 16-bit integer	<sys/types.h>
uint16_t	Unsigned 16-bit integer	<sys/types.h>
int32_t	Signed 32-bit integer	<sys/types.h>
uint32_t	Unsigned 32-bit integer	<sys/types.h>
sa_family_t	Address family of socket address structure	<sys/socket.h>
socklen_t	Length of socket address structure, normally uint32_t	<sys/socket.h>
in_addr_t	IPv4 address, normally uint32_t	<netinet/in.h>
in_port_t	TCP or UDP port, normally uint16_t	<netinet/in.h>

Figure 2.1 lists these three POSIX-defined data types

While a ccessing 32 bi t Ipv4 address, i f s erv is d efined as an Internet socket a ddress s tructure, 32 -bit IPv4 a ddress i n_addr s tructure i s referenced as s erv.sin_addr, w hile s erv.sin_addr.s_addr r eferences t he same 3 2-bit IPv4 a ddress a s a n i n_addr_t (typically a n uns igned 32 -bit integer).

IPv6 Socket Address Structure

The IPv6 socket address is defined by including the <netinet/in.h> header file. The IPv6 family is AF_INET6, whereas the IPv4 family is AF_INET. IPv6 socket address structure sockaddr_in6 is shown below.

```
struct sockaddr_in6
{
    uint8_t          sin6_len;          // sizeof this struct - 28 bytes
    sa_family_t      sin6_family;       // AF_INET6
    in_port_t        sin6_port;
    uint32_t         sin6_flowinfo;
    struct in6_addr  sin6_addr;         // 128 bit IPv6 address
    uint32_t         sin6_scope_id;
};
struct in6_addr
{
    uint8_t     s6_addr [16];     // IPv6 addresss (16 bytes - 128 bits)
};
```

The m embers i n t his s tructure are o rdered s o t hat i f t he s ockaddr_in6 structure is 64-bit a ligned, s o i s t he 128 -bit s in6_addr m ember. O n s ome 64-bit processors, data accesses of 64-bit values are optimized if stored on a 64-bit boundary.

2.3 VALUE-RESULT ARGUMENTS

When a socket address structure is passed to any socket function, it is always passed by reference (a pointer to the structure is passed). The length of the structure is also passed as an argument.

The way in which the length is passed depends on which direction the structure is being passed:

1. From the **process to the kernel**
2. From the **kernel to the process**

From process to kernel

Bind, *connect*, and *sendto* are the functions that pass a socket address structure from the process to the kernel. Two of the Arguments to these functions are:

- The pointer to the socket address structure
- The integer size of the structure

Because of these two arguments, kernel knows how much data to copy from process to kernel.

```
struct sockaddr_in serv;
/* fill in serv{} */
connect (sockfd, (SA *) &serv, sizeof(serv));
```

The datatype for the size of a socket address structure is actually socklen_t and not int, but the POSIX specification recommends that socklen_t be defined as uint32_t.

From kernel to process

Accept, *recvfrom*, *getsockname*, and *getpeername* are the functions that pass a socket address structure from the kernel to the process.

Two of the Arguments to these functions are:

- The pointer to the socket address structure
- The pointer to an integer containing the size of the structure.

```
struct sockaddr_un  cli;   /* Unix domain */
socklen_t  len;
len = sizeof(cli);         /* len is a value */
getpeername(unixfd, (SA *) &cli, &len);
/* len may have changed */
```

Value-result argument (Figure 3.2): the size changes from an integer to be a pointer to an integer because the size is both a value when the function is called and a result when the function returns.

- As a **value**: it tells the kernel the size of the structure so that the kernel does not write past the end of the structure when filling it in
- As a **result**: it tells the process how much information the kernel actually stored in the structure

For two other functions that pass socket address structures, recvmsg and sendmsg, the length field is not a function argument but a structure member.

If the socket address structure is fixed-length, the value returned by the kernel will always be that fixed size: 16 for an Ipv4sockaddr_in and 28 for an Ipv6 sockaddr_in6. But with a variable-length socket address structure (e.g., a Unix domainsockaddr_un), the value returned can be less than the maximum size of the structure.

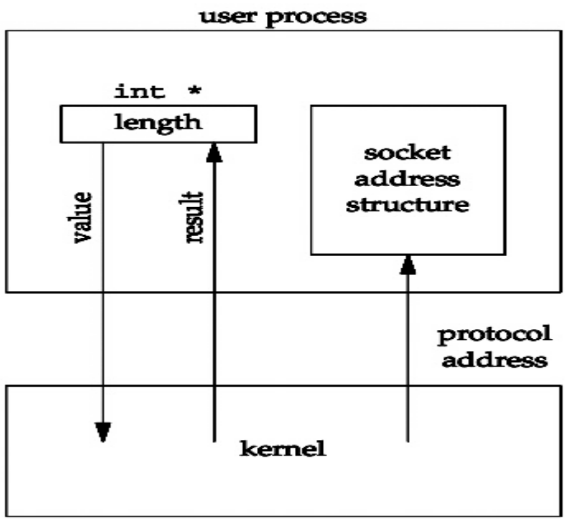


Figure 2.2: Value-result argument

CHECK YOUR PROGRESS

1. What is *sockaddr*?
2. How is the length of socket address structure sent from a process to kernel?

2.4 BYTE ORDERING FUNCTIONS

A 16-bit integer made up of 2 bytes can be stored in memory in two ways:

- **Little-endian** order: low-order byte is at the starting address.
- **Big-endian** order: high-order byte is at the starting address.

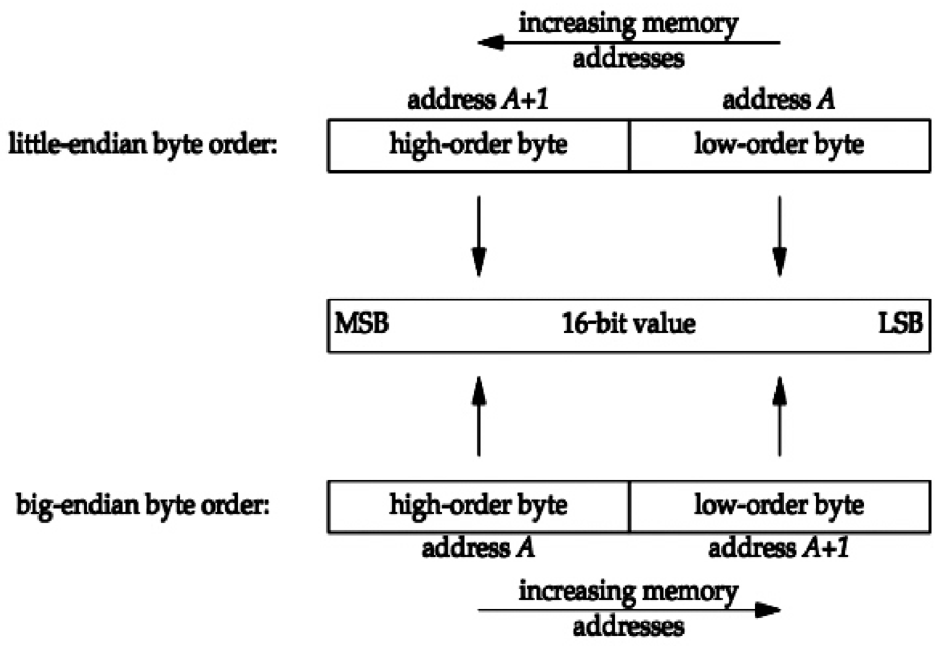


Figure 2.3: Byte Order

Figure 2.3 shows increasing memory addresses from right to left in the top and left to right in the bottom. Byte ordering used by a given system is called **host byte order.** The same applies to 32-bit integer. There are a variety of systems that can between little and big endian byte order at system reset or run time.

Since, a ll n etworking p rotocols s pecify the ne twork b yte or der w hile transferring d ata, it is imperative for a p rogrammer t o unde rstand t he ordering di fferences. F or e xample, T CP s egments transferred between nodes c ontain a 1 6-bit port num ber a nd a 32 bi t IPV4 ne twork a ddress. The r eceiving and s ending ne twork p rotocol s tacks m ust a gree on t he order in w hich th ese multibyte f ields a re transmitted. The I nternet protocols use big-endian byte ordering for these multibyte integers.

But, both history and the POSIX specification say that certain fields in the socket a ddress s tructures m ust be m aintained i n network b yte or der. We use the following four functions to convert between these two byte orders:

```
#include <netinet/in.h>

uint16_t htons(uint16_t host16bitvalue);

uint32_t ht onl(uint32_t hos t32bitvalue); /* B oth r eturn: va lue i n ne twork
byte order */

uint16_t ntohs(uint16_t net16bitvalue);

int32_t nt ohl(uint32_t n et32bitvalue); /* B oth r eturn: va lue i n hos t b yte
order */
```

- h stands for *host*
- n stands for *network*

- s stands for *short* (16-bit value, e.g. TCP or UDP port number)
- l stands for *long* (32-bit value, e.g. IPv4 address)

When using these functions, we do not care about the actual values (big-endian or little-endian) for the host byte order and the network byte order. What we must do is call the appropriate function to convert a given value between the host and network byte order. On those systems that have the same byte ordering as the Internet protocols (big-endian), these four functions are usually defined as null macros. An 8-bit entity is a "Byte" but most Internet standards use the term octet instead of Byte.

Byte Manipulation Functions

Socket address structures are manipulated using two groups of functions that operate on Multibyte fields. These functions do not interpret data and do not assume data as null-terminated C string. These functions are necessary to manipulate socket assress structures which have IP addresses that have bytes of 0 and not null-terminated C strings. The two groups of function are as follows:

❖ Ones whose names start with b (for byte) are from 4.2BSD and are still provided by almost any system that supports the socket functions. Examples for this type are bzero, bcopy and bcmp.

```
void bzero(void *dest, size_tnbytes);

void bcopy(const void *src, void *dest, size_tnbytes);

int bcmp(const void *ptr1, const void *ptr2, size_tnbytes);

                                        /*Returns: 0 if equal, nonzero if
unequal*/
```

bzero is used to initialize the socket addresses as it sets the specified number of bytes to 0 at destination, bcopy moves the specified number of bytes from the source to the destination, and bcmp compares two arbitrary byte strings.

❖ Ones whose names start with mem(for memory), are from ANSI C standard and are provided with any system that supports an ANSI C library. Examples for this type are memset, memcpy and memcmp.

```
void *memset(void *dest, intc, size_tlen);

void *memcpy(void *dest, const void *src, size_tnbytes);

int memcmp(const void *ptr1, const void *ptr2, size_tnbytes);

                /*Returns: 0 if equal, <0 or >0 if unequal*/
```

memset sets the specified number of bytes to the value c in the destination, memcpy is similar to bcopy, but the order of the two pointer arguments is swapped. bcopy correctly handles overlapping fields, while the behavior of memcpy is undefined if the source and destination overlap.

The two pointers for memcpy are written in the same left-to-right order as an a ssignment s tatement in C . A ll memxxx f unctions r equire a length argument which is th e f inal a rgument me mcmp c ompares tw o a rbitrary byte strings and returns 0 if they are identical. If not identical, the return va lue is e ither greater than 0 or less than 0, depending on w hether the first une qual b yte p ointed t o b y pt r1 i s gr eater t han or l ess t han t he corresponding b yte poi nted t o b y ptr2. The comparison is done assuming the two unequal bytes are unsigned chars.

2.5 RELATED FUNCTIONS

inet_aton, inet_addr, and inet_ntoa Functions

We w ill de scribe t wo groups o f a ddress c onversion f unctions i n t his section a nd t he ne xt. They convert Internet ad dresses b etween A SCII strings (what hum ans pr efer t o us e) a nd network b yte or dered bi nary values (values that are stored in socket address structures).

1. inet_aton, inet_ntoa, and inet_addr convert an IPv4 address from a dotted-decimal string (e.g., "206.168.112.96") to its 32-bit network byte ordered binary value. You will probably encounter these functions in lots of existing code.

2. The newer functions, inet_pton and inet_ntop, handle both IPv4 and IPv6 addresses. We describe these two functions in the next section and use them throughout the text.

```
#include <arpa/inet.h>
int inet_aton(const char *strptr, struct in_addr *addrptr);
Returns: 1 if string was valid, 0 on error
in_addr_t inet_addr(const char *strptr);
Returns:    32-bit binary      network    byte    ordered    IPv4
address;    INADDR_NONE if error
char *inet_ntoa(struct in_addr inaddr);
Returns: pointer to dotted-decimal string
```

The first of these, inet_aton, converts the C character string pointed to by strptr into its 32-bit binary network byte ordered value, which is stored through the pointer addrptr. If successful, 1 is returned; otherwise, 0 is returned.

An undocumented feature of inet_aton is that if addrptr is a null pointer, the function still performs its validation of the input string but does not store any result.

inet_addr does the same conversion, returning the 32-bit binary network byte ordered value as the return value. The problem with this function is that all 232 possible binary values are valid IP addresses (0.0.0.0 through

255.255.255.255), but the function returns the constant INADDR_NONE (typically 32 on e-bits) on an error. This means the dotted-decimal string 255.255.255.255 (the IPv4 limited broadcast address)

cannot be handled by this function since its binary value appears to indicate failure of the function.

A potential problem with inet_addr is that some man pages state that it returns 1 on an error, instead of INADDR_NONE. This can lead to problems, depending on the C compiler, when comparing the return value of the function (an unsigned value) to a negative constant.

Today, inet_addr is deprecated and any new code should use inet_aton instead. Better still is to use the newer functions described in the next section, which handle both IPv4 and IPv6.

The inet_ntoa function converts a 32-bit binary network byte ordered IPv4 address into its corresponding dotted-decimal string. The string pointed to by the return value of the function resides in static memory. This means the function is not reentrant. Finally, notice that this function takes a structure as its argument, not a pointer to a structure.

Functions that take actual structures as arguments are rare. It is more common to pass a pointer to the structure.

inet_pton and inet_ntop Functions

These two functions are new with IPv6 and work with both IPv4 and IPv6 addresses. We use these two functions throughout the text. The letters "p" and "n" stand for presentation and numeric. The presentation format for an address is often an ASCII string and the numeric format is the binary value that goes into a socket address structure.

```
#include <arpa/inet.h>

int inet_pton(int family, const char *strptr, void *addrptr);

Returns: 1 if OK, 0 if input not a valid presentation format, -1 on error

const char *inet_ntop(int family, const void *addrptr, char *strptr, size_t
len);

Returns: pointer to result if OK, NULL on error
```

The family argument for both functions is either AF_INET or AF_INET6. If family is not supported, both functions return an error with errno set to **EAFNOSUPPORT.**

The first function tries to convert the string pointed to by strptr, storing the binary result through the pointer addrptr. If successful, the return value is 1.

If the input string is not a valid presentation format for the specified family, 0 is returned.

inet_ntop does the reverse conversion, from numeric (addrptr) to presentation (strptr).

The len argument is the size of the destination, to prevent the function from overflowing the caller's buffer. To help specify this size, the following two definitions are defined by including the <netinet/in.h> header:

```
#define INET_ADDRSTRLEN 16 /* for IPv4 dotted-decimal */

#define INET6_ADDRSTRLEN 46 /* for IPv6 hex string */
```

If len is too small to hold the resulting presentation format, including the terminating null, a null pointer is returned and errno is set to ENOSPC.

The strptr argument to inet_ntop cannot be a null pointer. The caller must allocate memory for the destination and specify its size. On success, this pointer is the return value of the function.

Check your progress

1. What is host byte order?
2. What do inet_pton and inet_ntop Functions do?

2.6 SUMMARY

Sockets are an integral part of every network program. The address structures of the sockets are filled and passed as pointers to the various socket functions. When a pointer to one of these structures is passed to a socket function, it fills in the contents. These structures are always passed by reference and the size of the structure is passed as another argument. When a socket function fills the structure, the length is also passed as reference, so that the value of the length can be updated by the function. These are termed as value-result arguments.

The address structures are self-defining because they contain a field (“domain") that specifies the address family contained in the structure. Newer implementations supporting variable-length address structures also contain a length field at the beginning, indicating the length of the entire structure.

The two functions that convert IP addresses between presentation format (what we write, such as ASCII characters) and numeric format (what goes into a socket address structure) are inet_pton and inet_ntop. These two functions are, however, protocol- dependent. A better technique is to manipulate the socket address structures as opaque objects, knowing just the pointer to the structure and its size.

2.7 TERMINAL QUESTIONS

1. Define socket and list out its types.
2. Compare t he IPV4, IPV6, U nix dom ain a nd data l ink s ocket address structures. State your assumptions.
3. Describe IPV4 and IPV6 socket address structure.
4. What is byte ordering function?
5. Explain in detail about address conversion functions.
6. Explain value-result arguments.

UNIT-3 : ELEMENTARY TCP SOCKETS

Structure

3.0 INTRODUCTION

This unit describes the socket network Inter Process Communication (IPC) interface, which can be used by processes to communicate with other processes, regardless of where they are running, i.e. the same interfaces can be used for both inter and intra machine communication. The socket interface can be used to communicate using many different network protocols. However, our discussion shall be restricted to the TCP/IP protocol suite, since it is the de facto standard for communicating over the Internet.

3.1 OBJECTIVES

After the end of this unit, you should be able to:

- Understand the different elementary functions required for establish a TCP connection between a server and client
- Understand the difference between concurrent and iterative servers
- Implement a concurrent server
- Gain an insight of other functions related to socket communication

3.2 ELEMENTARY TCP SOCKETS AND SOCKET FUNCTION

The timeline of a typical scenario that takes place between a TCP client and server has been shown in Figure 3.1. First, the server is started, and then sometime later, a client is started that connects to the server. We assume that the client sends a request to the server, the server processes the request, and the server sends a reply back to the client. This continues until the client closes its end of the connection, which sends an end-of-file notification to the server. The server then closes its end of the connection and either terminates or waits for a new client connection.

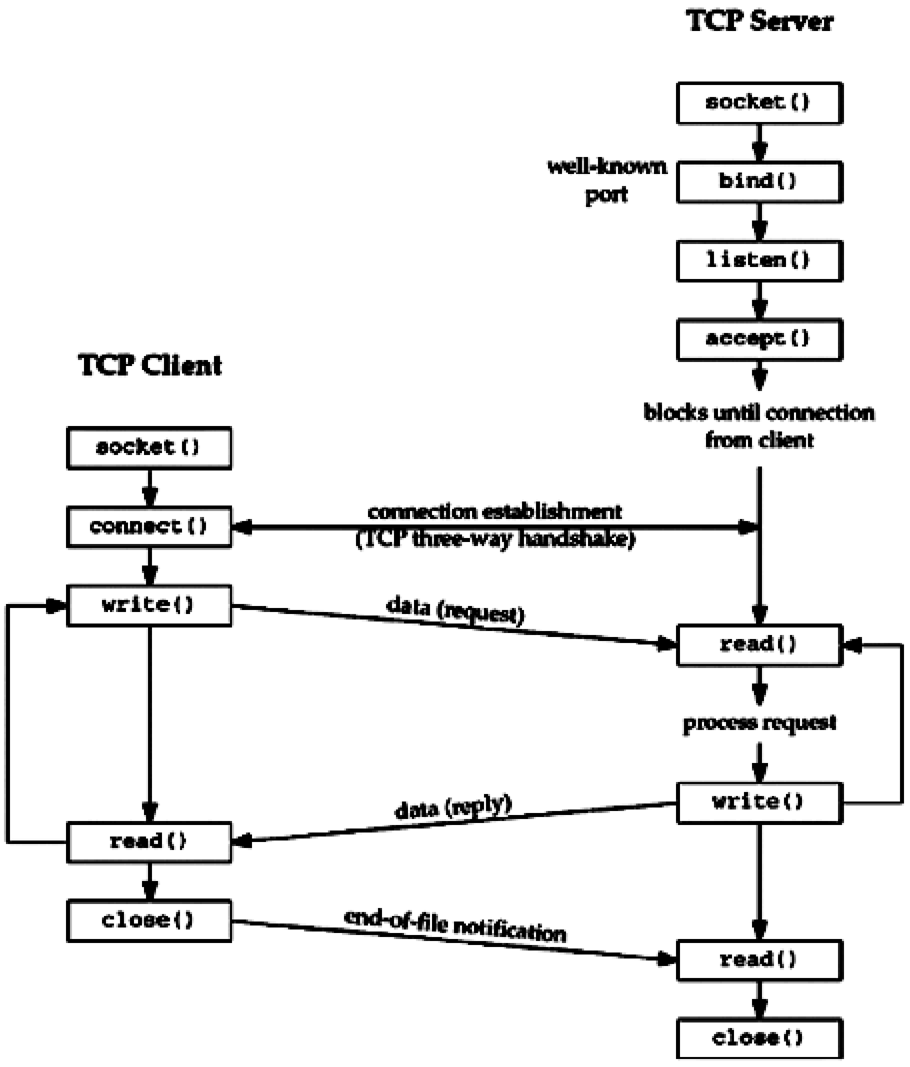


Fig. 3.1: TCP Client and Server

A socket is an abstraction of a communication endpoint. Just as they would use file d escriptors to acc ess files, ap plications u se so cket d escriptors t o ac cess sockets. S ocket d escriptors ar e implemented as f ile d escriptors i n t he U NIX

System. Indeed, many of the functions that deal with file descriptors, such as read and write, will work with a socket descriptor.

To pe rform network I/O, the first t hing a process must d o is call t he socket function, s pecifying t he t ype of c ommunication pr otocol de sired (TCP using IPv4, UDP using IPv6, Unix domain stream protocol, etc.).

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

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

Returns: file (socket) descriptor if OK, −1 on error
```

The *domain* parameter specifies the nature of the communication, as well as t he address f ormat. T able 3. 1 l ists t he dom ains s pecified b y P OSIX.1. T he constants start with AF_ (for address family) because each domain has its own representation format for an address.

Domain	**Description**
AF_INET	IPv4 Internet domain
AF_INET6	IPv6 Internet domain
AF_Local	UNIX domain
AF_ROUTE	Routing sockets
AF_KEY	Key socket
AF_UNSPEC	Unspecified

Table 3.1: Socket Communication Domains

The second parameter *type* specifies the type of the socket, which further determines th e communication characteristics. T able 3 .2 s ummarizes t he socket types defined by POSIX.1.

Type	**Description**
SOCK_DGRAM	fixed-length, connectionless, unreliable messages
SOCK_SEQPACKET	fixed-length, s equenced, r eliable, c onnection-oriented messages
SOCK_STREAM	sequenced, r eliable, bi directional, c onnection-oriented byte stream
SOCK_RAW	datagram interface to IP (optional in POSIX.1)

Table 3.2: Socket Types

The third argument *protocol* is usually zero, to select the default protocol for t he g iven dom ain a nd s ocket t ype. W hen m ultiple pr otocols a re supported for the same domain and socket type, we can use the *protocol* argument t o s elect a p articular pr otocol. T he de fault p rotocol f or a SOCK_STREAM socket in the AF_INET communication domain is TCP (Transmission C ontrol P rotocol). T he de fault pr otocol f or a SOCK_DGRAM socket in the AF_INET communication domain is UDP (User D atagram P rotocol). T able 3.3 l ists t he pr otocols de fined f or t he Internet domain sockets.

Protocol	**Description**
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IP_PROTO_SCTP	SCTP transport protocol

Table 3.3: Protocols defined for Internet domain sockets

On success, the *socket* function returns a small non-negative integer value. This is termed as socket descriptor, denoted by *sockfd*. This socket descriptor depends upon the protocol family (IPv4, IPv6, or Unix) and the type of the (stream, datagram, or raw).

These sockets support bidirectional communication. The I/O operations on a socket can be disabled by using the *shutdown* function.

```
#include <sys/socket.h>
int shutdown(int sockfd, int how);
Returns: 0 if OK, −1 on error
```

If *how* is set to SHUT_RD, then reading from the socket is disabled. If *how* is SHUT_WR, then the socket can't be used for transmitting data. The value of *how* can be set to SHUT_RDWR to disable both data transmission and reception.

If a socket can be closed then, why a shutdown is required? There can be several reasons.

- First, close will deallocate the network endpoint only when the last active reference is closed. If we duplicate the socket (with dup, for example), the socket won't be deallocated until we close the last file descriptor referring to it. The shutdown function allows us to deactivate a socket independently of the number of active file descriptors referencing it.
- Second, it is sometimes convenient to shut a socket down in one direction only. For example, we can shut a socket down for writing if we want the process we are communicating with to be able to tell when we are done transmitting data, while still allowing us to use the socket to receive data sent to us by the process.

3.3 CONNECT FUNCTION

A connection-oriented network service (SOCK_STREAM or SOCK_SEQPACKET) requires that before data is exchanged, a connection must be established between the socket of the process requesting the service (the client) and the process providing the service (the server). The connect function to create a connection.

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *addr, socklen_t len);
Returns: 0 if OK, −1 on error
```

Here, sockfd is a socket descriptor returned by the socket function. The second and third arguments are a pointer to a socket address structure and its size. The socket address structure must contain the IP address and port number of the server.

While connecting to a server, the connect request might fail for multiple reasons. For a connect request to succeed, the machine to which we are trying to connect must be up and running, the server must be bound to the address we are trying to contact, and there must be room in the server's pending connect queue.

Check your progress

1. What are the reasons of shutting down a socket?
2. Enlist some reasons of the connect request failure.

3.4 BIND FUNCTION

The bind function assigns a local protocol address to a socket. With the Internet protocols, the protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number.

```
#include <sys/socket.h>
int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
                                                Returns: 0 if OK, –1 on error
```

The second argument is a pointer to a protocol-specific address, and the third argument is the size of this address structure.

The bind function lets us specify the IP address, the port, both, or neither. Table 3.4 enlists the values to which sin_addr and sin_port, or sin6_addr and sin6_port, can be set as per the requirement.

Process Specifies		**Result**
IP address	**Port**	
Wildcard	0	Kernel chooses IP address and port
Wildcard	nonzero	Kernel chooses IP address, process specifies port
Local IP address	0	Process specifies IP address, kernel chooses port
Local IP address	nonzero	Process specifies IP address and port

Table 3.4: Result when specifying IP address and/or port number to bind

If IPv4 is being used, the wildcard address can be denoted by the constant INADDR_ANY, whose value is normally 0. This informs the kernel to choose the IP address.

```
struct sockaddr_in servaddr;

servaddr.sin_addr.s_addr = htonl (INADDR_ANY);          /* wildcard */
```

This technique cannot be used with IPv6, since the length of IPv6 address is 128-bit which can be stored in a structure. (The C language does not allow a constant structure on the right-hand side of an assignment.) The following code snippet shows the method of assigning wildcard address in case of IPv6.

```
struct sockaddr_in6 serv;

serv.sin6_addr = in6addr_any;                           /* wildcard */
```

The extern declaration of the variable in6addr_only is present in the <netinet/in.h> header file. The system shall allocate the memory and initialize the in6addr_any variable to the constant IN6ADDR_ANY_INIT.

The value of INADDR_ANY (0) shall be the same in either host or network. This eliminates the need of htonl. However, since the header <netinet/in.h> defines all the INADDR_constants in host byte order, the function htonl should be used with any of these constants.

3.5 LISTEN FUNCTION

A server announces that it is willing to accept connect requests by calling the listen function. The call to the socket function always created an active socket, i.e. a client socket that can issue a connect. The listen function converts an unconnected socket into a passive socket. This denotes that the kernel should accept incoming connection requests directed to this socket.

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

int listen(int sockfd, int backlog);

Returns: 0 if OK, −1 on error
```

The backlog argument provides a hint to the system regarding the number of outstanding connect requests that it should enqueue on behalf of the process. The actual value is determined by the system, but the upper limit is specified as SOMAXCONN in <sys/socket.h>.

To understand the backlog argument, we must realize that for a given listening socket, the kernel maintains two queues:

An incomplete connection queue, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three- way handshake. These sockets are in the SYN_RCVD state.

A completed connection queue, which contains an entry for each client with whom the TCP three-way handshake has completed. These sockets are in the ESTABLISHED state.

Figure 3.2 depicts these two queues for a given listening socket.

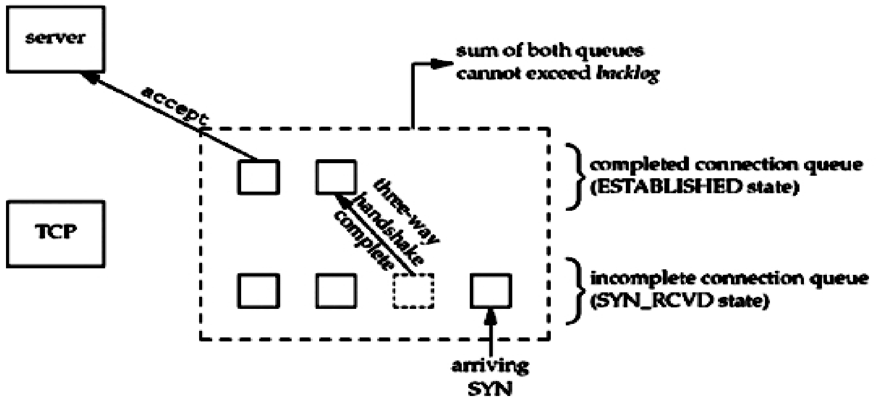

server
accept
sum of both queues cannot exceed *backlog*
completed connection queue (ESTABLISHED state)
TCP
three-way handshake complete
incomplete connection queue (SYN_RCVD state)
arriving SYN

Fig 3.2: Queues maintained by TCP for a listening socket

Whenever an entry is created in the incomplete queue, the arguments from the listen socket are copied to the newly created connection. Once the queue is full, the system will reject additional connect requests, so the backlog value must be chosen based on the expected load of the server and the amount of processing it must do to accept a connect request and start the service.

3.6 ACCEPT FUNCTION

Once a server has called listen, the socket can receive connect requests. The accept function is used to retrieve a connect request and establish a connection.

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

int accept(int sockfd, struct sockaddr *restrict addr, socklen_t *restrict len);

Returns: file (socket) descriptor if OK, −1 on error
```

If the call to accept is successful, a "new" descriptor is returned by the kernel. This new descriptor identifies the TCP connection that has been established with the client. The sockfd argument specifies the listening socket (created upon successful call to the socket function) while the return value of accept is termed as the connected socket. A server normally creates only one listening socket which exists for the lifetime of the server. A connected socket is created for each client connection that is accepted (i.e., for which the TCP three-way handshake completes). When the server is finished serving a client, the connected socket is closed.

This function returns up to three values: an integer return code that is either a new socket descriptor or an error indication, the protocol address of the client process (through the cliaddr pointer), and the size of this address (through the addrlen pointer). If the protocol address of the client is not required, both cliaddr and addrlen are set to null pointers.

3.7 FORK AND EXEC FUNCTION

The fork function is used for creating a new process in Unix.

```
#include <unistd.h>

pid_t fork(void);

            Returns: 0 in child, process ID of child in parent, −1 on error
```

The fork function is “called once but returns twice”. The process ID of the newly created child process is returned to the parent (the calling process) while the value 0 is returned to the child process.

The reason why 0 is returned to the child, instead of the process ID of the parent, is that a child has only one parent and it can always obtain the parent's process ID by calling getppid function. A parent, on the other hand, can have any number of children, and there is no way to obtain the process IDs of its children. If a parent wants to keep track of the process IDs of all its children, it must record the return values from fork.

Any descriptor opened by the parent process before calling fork shall be shared with the child process after fork executes successfully. This feature is used by the network servers where the parent first calls accept to establish a connection with the client and then calls fork. This ensures that the connected socket is shared between the parent and the child process. The child process can then read and write on the connected socket and the parent can close the same connected socket.

The fork function is generally used for the following purpose:

1. Making a copy of the process ensures that so that one copy handles one operation while the other copy performs another task. This is, generally, the case in network servers.

2. When a process wants to execute another program, it first calls fork to make a copy of itself, and then one of the copies (child) calls exec to replace itself with the new program. This is the case for programs such as shells.

The only way in which an executable program file on disk can be executed by Unix is for an existing process to call one of the six exec functions. exec replaces the current process image with the new program file, and this new program normally starts at the main function. The process ID does not change. We refer to the process that calls exec as the calling process and the newly executed program as the new program.

The differences in the six exec functions are:

a) whether the program file to execute is specified by a filename or a pathname;

b) whether the arguments to the new program are listed one by one or referenced through an array of pointers; and

c) whether the environment of the calling process is passed to the new program or whether a new environment is specified.

The six variants of the exec function have been shown below:

```
#include <unistd.h>
int execl (const char *pathname, const char *arg0, ... /* (char *) 0 */ );
int execv (const char *pathname, char *const argv[]);
int execle (const char *pathname, const char *arg0, ... /* (char *) 0,
                    char *const envp[] */ );
int execve (const char *pathname, char *const argv[], char *const envp[]);
int execlp (const char *filename, const char *arg0, ... /* (char *) 0 */ );
int execvp (const char *filename, char *const argv[]);
                              All six return: -1 on error, no return on success
```

These functions return to the caller only if an error occurs. Otherwise, control passes to the start of the new program, normally the main function.

Figure 3.3 depicts the relationship among these six variants of the exec function. It should be noted that, only execve is a system call while the remaining five are library functions that internally call execve.

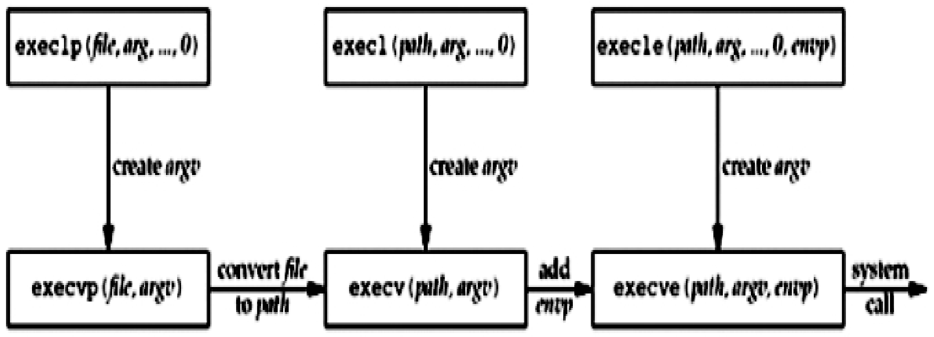


Figure 3.3: Relationship among the six exec functions

There exist following differences among these variants of exec functions:

The functions execlp, execl and execle considers each string parameter as a separate parameter to the exec function, with a null pointer terminating the v ariable n umber o f p arameters. T he f unctions e xecvp, e xecv a nd execve have an argv array, that contains pointers to the string parameters. The argv array must contain a null pointer to specify its end, since a count is not specified.

The functions ex eclp an d ex ecvp r equire a file parameter s pecifying th e filename. This i s c onverted i nto a pa thname b y using t he c urrent P ATH environment v ariable. H owever, i f t he f ilename p arameter t o ex eclp o r execvp co ntains a s lash (/) an ywhere i n the s tring, the P ATH v ariable i s not us ed. T he remaining functions r equire a f ully qua lified pa thname argument.

The functions execlp, execl, execvp, and execv do not require an explicit environment pointer. The current value of the external variable environ is used for bui lding a n environment l ist that i s pa ssed t o the ne w pr ogram. The functions ex ecle and ex ecve require an ex plicit en vironment list. The parameter envp is an array of pointers terminated by a null pointer.

Check your progress

1. What is the default value of INADDR_ANY?

2. What is the use of Fork function?

3.8 CONCURRENT SERVERS

When a c lient r equest r equires lo nger time to c omplete, it is n ot feasible t o d edicate a s ingle s erver f or o ne cl ient. T he s erver s hould b e able to s erve mu ltiple client r equests a t th e s ame time . S uch t ype o f servers is termed as concurrent servers. The simplest method to implement a concurrent server is to fork a child process for serving each client request. The following code snippet shows the implementation for a typical concurrent server.

```
pid_t pid;
int listenfd, connfd;
listenfd = Socket( ... );
/* fill in sockaddr_in{} with server's well-known port */
Bind(listenfd, ... );
Listen(listenfd, LISTENQ);
for ( ; ; ) {
        connfd = Accept (listenfd, ... ); /* probably blocks */
        if( (pid = Fork()) == 0) {
                Close(listenfd); /* child closes listening socket */
                doit(connfd); /* process the request */
                Close(connfd); /* done with this client */
                exit(0); /* child terminates */
        }
        Close(connfd); /* parent closes connected socket */
}
```

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

The function does whatever is required to service the client. When this function returns, we explicitly close the connected socket in the child. This is not required since the next statement calls exit, and part of process termination is to close all open descriptors by the kernel. Whether to include this explicit call to close or not is a matter of personal programming taste.

3.9 CLOSE FUNCTION

The close function is used to close an open socket and terminate the TCP connection.

```
#include <unistd.h>

int close (int sockfd);

Returns: 0 if OK, -1 on error
```

The default action of close function with a TCP socket is to mark the socket as closed and return to the calling process immediately. The socket descriptor shall no longer be usable by the process. It cannot be further used for either data transmission or reception. However, TCP will try to send any data that has already been queued to be sent, after which the TCP connection termination procedure takes place.

Check your progress

1. What are the functions of getsockname and getpeername methods?
2. How can you implement concurrent servers?

3.10 RELATED FUNCTIONS

These getsockname function returns the local protocol address while the function getpeername returns the foreign protocol address associated with a socket.

```
#include <unistd.h>

int getsockname(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen);

int getpeername(int sockfd, struct sockaddr *peeraddr, socklen_t *addrlen);

                                         Both return: 0 if OK, -1 on error
```

The functions return the combination of an IP address and port number associated with one of the two ends of a network connection.

These two functions are required for the following reasons:

- After connect successfully returns in a TCP client that does not call bind, getsockname returns the local IP address and local port number assigned to the connection by the kernel.
- After calling bind with a port number of 0 (telling the kernel to choose the local port number), getsockname returns the local port number that was assigned.
- getsockname can be called to obtain the address family of a socket.
- In a TCP server that binds the wildcard IP address, once a connection is established with a client (accept returns successfully), the server can call getsockname to obtain the local IP address assigned to the connection. The socket descriptor argument in this call must be that of the connected socket, and not the listening socket.
- When a server is execed by the process that calls accept, the only way the server can obtain the identity of the client is to call getpeername.

Example: Obtaining the Address Family of a Socket

The sockfd_to_family function returns the address family of a socket. The following code snippet demonstrates returning the address family of a socket.

```
#include "unp.h"
int sockfd_to_family(int sockfd)
{
        struct sockaddr_storage ss;
        socklen_t len;
        len = sizeof(ss);
        if (getsockname(sockfd, (SA *) &ss, &len) < 0)
                return (-1);
        return (ss.ss_family);
}
```

Check your progress

Write a program to exchange one hello message between server and client to demonstrate the client/server model.

3.11 SUMMARY

A call to the socket function returns a socket descriptor which can be used for inter process communication between two different machines or on the same machine. Clients wishing to establish a connection with the server call the connect function while servers call the bind, listen, and accept function to accept connections from the client. Open socket scan be closed by issuing a call to the standard close function, although there also exist a shutdown function for the similar purpose.

TCP servers should be able to serve concurrent requests from the clients. This is achieved by calling fork function for every client connection being handled by the server. However, UDP servers are, generally, iterative in nature.

3.12 TERMINAL QUESTIONS

1. Describe the procedure and sequence of function calls required for establishing a TCP connection between a client and the server.
2. In Section 3.4, we stated that the INADDR_ constants defined by the <netinet/in.h> header are in host byte order. How can we tell this?
3. An iterative server waits for the child to execute the command and exit before accepting the next connect request. Write a pseudocode for the server so that the time to service one request doesn't delay the processing of incoming connect requests.

4. Refer to code for concurrent servers in Section 3.7. Assume the child runs first after the call to fork. The child then completes the service of the client before the call to fork returns to the parent. What happens in the two calls to close?

5. Write a program to implement a chat room environment for one client and one server.

6. Write a program to implement a chat room environment for multiple client and one server.

UNIT-4 : TCP Client/Server

Structure

4.0 INTRODUCTION

A s erver pr ovides a s ervice on a given por t b y waiting f or connections from future clients. A client can connect to a service once the server i s r eady t o a ccept co nnections (accept). In or der t o m ake a connection, the client must know the IP number of the server machine and the port number of the service. If the client does not know the IP number, it ne eds t o r equest na me/number r esolution. O nce t he c onnection i s accepted b y t he s erver, each p rogram can co mmunicate v ia i nput-output channels over the sockets created at both ends.

4.1 OBJECTIVES

After the end of this unit, you should be able to:

- Understand the various functions of TCP Echo server
- Gain i nsights r egarding nor mal s tartup, t erminate a nd s ignal handling server process termination
- Differentiate between crashing and rebooting of server host
- Know about shutdown of server host

4.2 TCP ECHO SERVER FUNCTION

TCP Echo Server: Main Function

The concurrent server program has been represented through the following code:

```
#include "unp.h"
Int main(int argx, char **argv)
{
          Int listenfd, connfd;
          pid_t childpid;
          socklen_t clien;
          struct sockaddr_in cliaddr,servaddr;
          listenfd=Socket(AF_INET,SOCK_STREAM,0);
          bzero(&servaddr,sizeof(servaddr));
          servaddr.sin_family=AF_INET;
          servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
          servaddr.sin_port=htons(SERV_PORT);
          Bind(listenfd,LISTENQ);
          for( ; ;) {
          clien = sizeof(cliaddr);
          connfd = Accept(listenfd, (SA*) &cliaddr, &clilen);
          if((childpid = Fork()) == 0) { /* child process */
              Close(listenfd); /*close listening socket*/
               str_echo(connfd); /*process the request */
                exit(0);
          }
          Close(connfd); /*parent closes connected socket */
     }
  }
```

The actions processed by the code are as follows:

- **Create socket, bind server's well-known port**
  - ❖ A TCP socket is created.
  - ❖ An Internet socket address structure is filled in with the wildcard address (INADDR_ANY) and the server's well-known port (SERV_PORT, here defined as 9877 in header).

Binding th e w ildcard a ddress te lls th e s ystem th at w e w ill accept a c onnection de stined f or a ny l ocal i nterface, i n c ase the s ystem i s m ultihome. It s hould be greater t han 1023 (we do not ne ed a reserved por t), greater t han 500 0 (to a void conflict w ith th e e phemeral p orts a llocated b y many Berkeley-derived implementations), less than 49152 (to avoid conflict w ith t he " correct" r ange of ephemeral por ts), a nd i t should not conflict with any registered port.

- ❖ The socket is converted into a listening socket by listen.

- **Wait for client connection to complete**

  - ❖ The s erver b locks i n the c all to accept, w aiting f or a client connection to complete.

- **Concurrent server**

  - ❖ For each client, fork spawns a child, and the child handles the new c lient. T he c hild closes t he l istening s ocket a nd t he parent closes the connected socket.

TCP Echo Server: str_echo Function

The function s tr_echo i s r esponsible f or c onducting t he s erver pr ocessing for ea ch cl ient. It r eads d ata f rom t he cl ient an d ech oes i t b ack t o t he client.

```
#include   "unp.h"

void

str_echo(int sockfd)

{

   ssize_t    n;

   char       buf[MAXLINE];

again:

   while ( (n = read(sockfd, buf, MAXLINE)) > 0)

      Writen(sockfd, buf, n);

   if (n < 0 && errno == EINTR)

      goto again;

   else if (n < 0)

      err_sys("str_echo: read error");

}
```

The code provided above processes the following actions:

- **Read a buffer and echo the buffer**
  - ❖ Read reads data from the socket and the line is echoed back to the client by writen. If the client closes the connection (the normal scenario), the receipt of the client's FIN causes the child's read to return 0. This causes the str_echo function to return, which terminates the child.

Check your progress

1. What are the steps involved in startup of TCP client/server?

2. What are choices of disposition?

4.3 NORMAL START-UP

Normal start-up of TCP client/server includes following steps:

- Start TCP server in background on host system.

  linux % tcpserv01 &

  [1] 17870

  When the server starts, it calls *socket, bind, listen*, and *accept*, blocking in the call to *accept*. State of the server's listening socket is verified through *netstat* program. *netstat* command is used along with *-a* flag to see all listening sockets.

- Start the client on the same host, specifying the server's IP address of 127.0.0.1 (the loopback address).
- The client calls socket and connect, the latter causing TCP's three-way handshake to take place.
- When the three-way handshake completes, connect returns in the client and accept returns in the server and connection is established.
- The client calls *str_cli*, which will block in the call to *fgets*, because we have not typed a line of input yet.
- When accept returns in the server, it calls fork and the child calls *str_echo*. This function calls *readline*, which calls read, which blocks while waiting for a line to be sent from the client.
- The server parent, on the other hand, calls *accept* again, and blocks while waiting for the next client connection.

4.4 TERMINATE AND SIGNAL HANDLING SERVER PROCESS TERMINATION

Normal Termination

At this point, the connection is established and whatever we type to the client is echoed back.

```
linux % tcpcli01 127.0.0.1   # this line has been represented
earlier hello, world          # now this is typed
hello, world                  # the line is echoed
good bye
good bye
^D                   # Control-D is the terminal EOF character
```

If netstat is executed immediately, the following is received:

```
linux % netstat -a | grep 9877
tcp    0    0 *:9877            *:*              LISTEN
tcp    0    0 localhost:42758   localhost:9877   TIME_WAIT
```

The output of the netstat is piped into grep. This prints only the lines in possession of the port acquainted with the server:

- The client's side of the connection (since the local port is 42758) enters the TIME_WAIT state
- The listening server still waits for another client connection.

The following steps are involved in the normal termination of client and server:

- When we type our EOF character, fgets returns a null pointer and the function str_cli returns.
- str_cli r eturns to th e c lient ma in f unction, w hich te rminates b y calling exit.
- Part of process termination is the closing of all open descriptors, so the client socket is closed by the kernel. This sends a FIN to the server, to which the server TCP responds with an ACK. This is the first ha lf of t he T CP c onnection t ermination s equence. A t th is point, the server socket is in the CLOSE_WAIT state and the client socket is in the FIN_WAIT_2 state.
- When the server TCP receives the FIN, the server child is blocked in a call to read, and read then returns 0. This causes the str_echo function to return to the server child main.

- The server child terminates by calling exit.
- All open descriptors in the server child are closed.
- The closing of the connected socket by the child causes the final two segments of the TCP connection termination to take place: a FIN from the server to the client, and an ACK from the client.
- Finally, the SIGCHLD signal is sent to the parent when the server child terminates.
- This occurs in this example, but we do not catch the signal in our code, and the default action of the signal is to be ignored. Thus, the child enters the zombie state. We can verify this with the ps command.

```
linux % ps -t pts/6 -o pid,ppid,tty,stat,args,wchan
  PID  PPID TT       STAT COMMAND         WCHAN
22038 22036 pts/6    S    -bash           read_chan
17870 22038 pts/6    S    ./tcpserv01      wait_for_connect
19315 17870 pts/6    Z    [tcpserv01 <defu do_exit
```

The STAT of the child is now Z (for zombie).

Zombie Process: It is a process that has completed execution (via the exit system call) but still has an entry in the process table. This process is in the “Terminated State”. This occurs for child processes, where the entry is still needed to allow the parent process to read its child's exit status: once the exit status is read via the wait system call, the zombie's entry is removed from the process table and it is said to be "reaped". A child process always first becomes a zombie before being removed from the resource table. In most cases, under normal system operation zombies are immediately waited on by their parent and then reaped by the system – processes that stay zombies for a long time are generally an error and cause a resource leak.

We need to clean up our zombie processes and doing this requires dealing with Unix signals. The following section sheds light on signal handling.

POSIX Signal Handling

A signal is a notification to a process regarding occurrence of an event. Often, signals are regarded as software interrupts. Signals usually occur asynchronously. This means that a process is not provided any information regarding the time of occurrence of a signal before its actual occurrence. There are different types of signal as following:

Name	Description	Default action
SIGABRT	Abnormal T ermination (abort)	Terminate + core
SIGALRM	Timer expired (alarm)	terminate
SIGBUS	Hardware Fault	Terminate + core
SIGCANCEL	Threads l ibrary i nternal use	Ignore
SIGCOUNT	Continue s topped process	Continue/ignore
SIGEMT	Hardware fault	Terminate + core
SIGFPE	Arithmetic exception	Terminate +core
SIGFREEZE	Checkpoint freeze	Ignore
SIGHUP	Hang-up	terminate
SIGILL	Illegal instruction	Terminate + core
SIGINFO	Status request f rom keyword	ignore
SIGINT	Terminal in terrupt character	terminate
SIGIO	Asynchronous I/O	Terminate / Ignore
SIGIOT	Hardware fault	Terminate + core
SIGJVM1	Java v irtual m achine internal use	ignore
SIGKILL	Termination	terminate
SIGLOST	Resource lost	terminate
SIGLWP	Threads l ibrary i nternal use	Terminate/ignore
SIGPIPE	Write to p ipe w ith no readers	terminate
SIGPOLL	Pollable event (poll)	terminate
SIGVTALRM	Virtual time alarm(settimer)	terminate
SIGWAITING	threads lib rary internal use	Ignore

Table 4.1: UNIX System signal

Signals can be sent:

- By one process to another process (or to itself)
- By the kernel to a process.
    - ❖ For example, whenever a process terminates, the kernel send a SIGCHLD signal to the parent of the terminating process.

Every signal has a disposition, which is also called the action associated with the signal. The disposition of a signal is set by calling the sigaction function. Following are the three choices for the disposition:

1. Catching a signal. We can provide a function called a signal handler that is called whenever a specific signal occurs. The two signals SIGKILL and SIGSTOP cannot be caught. Our function is called with a single integer argument that is the signal number and the function returns nothing. Its function prototype is therefore:

2. void handler (int signo);

For most signals, we can call sigaction and specify the signal handler to catch it. A few signals, SIGIO, SIGPOLL, and SIGURG, all require additional actions on the part of the process to catch the signal.

3. Ignoring a signal. We can ignore a signal by setting its disposition to SIG_IGN. The two signals SIGKILL and SIGSTOP cannot be ignored.

4. Setting the default disposition for a signal. This can be done by setting its disposition to SIG_DFL. The default is normally to terminate a process on receipt of a signal, with certain signals also generating a core image of the process in its current working directory. There are a few signals whose default disposition is to be ignored: SIGCHLD and SIGURG (sent on the arrival of out-of-band data) are two that we will encounter in this text.

Signal Function

The POSIX way to establish the disposition of a signal is to call the sigaction function, which is complicated in that one argument to the function is a structure (struct sigaction) that we must allocate and fill in.

An easier way to set the disposition of a signal is to call the signal function. The first argument is the signal name and the second argument is either a pointer to a function or one of the constants SIG_IGN or SIG_DFL.

However, signal is an historical function that predates POSIX. Different implementations provide different signal semantics when it is called, providing backward compatibility, whereas POSIX explicitly spells out the semantics when sigaction is called.

The solution is to define our own function named signal that just calls the POSIX sigaction function. This provides a simple interface with the desired P OSIX s emantics. W e i nclude t his f unction i n our ow n l ibrary, along with our err_XXX functions and our wrapper functions.

```
#include    "unp.h"
Sigfunc *
signal(int signo, Sigfunc *func)
{
   struct sigaction    act, oact;
   act.sa_handler = func;
   sigemptyset(&act.sa_mask);
   act.sa_flags = 0;
   if (signo == SIGALRM) {
#ifdef  SA_INTERRUPT
      act.sa_flags |= SA_INTERRUPT;   /* SunOS 4.x */
#endif
   } else {
#ifdef  SA_RESTART
      act.sa_flags |= SA_RESTART;     /* SVR4, 44BSD */
#endif
   }
   if (sigaction(signo, &act, &oact) < 0)
      return(SIG_ERR);
   return(oact.sa_handler);
}
/* end signal */
Sigfunc *
Signal(int signo, Sigfunc *func)    /* for our signal() function */
{
   Sigfunc *sigfunc;
   if ( (sigfunc = signal(signo, func)) == SIG_ERR)
      err_sys("signal error");
   return(sigfunc);
}
```

Simplify function prototype using typedef

The nor mal f unction pr ototype f or signal is c omplicated b y th e le vel of nested parentheses.

```
void (*signal (int signo, void (*func) (int))) (int);
```

To simplify this, we define the Sigfunc type in our unp.h header as

```
typedef    void    Sigfunc(int);
```

stating that signal handlers are functions with an integer argument and the function returns nothing (void). The function prototype then becomes

```
Sigfunc *signal (int signo, Sigfunc *func);
```

A pointer to a signal handling function is the second argument to the function, as well as the return value from the function.

Set handler

The sa_handler member of the sigaction structure is set to the *func* argument.

Set signal mask for handler

POSIX allows us to specify a set of signals that will be blocked when our signal handler is called. Any signal that is blocked cannot be delivered to a process. We set the sa_mask member to the empty set, which means that no additional signals will be blocked while our signal handler is running. POSIX guarantees that the signal being caught is always blocked while its handler is executing.

Set SA_RESTART flag

SA_RESTART is an optional flag. When the flag is set, a system call interrupted by this signal will be automatically restarted by the kernel.

If the signal being caught is not SIGALRM, we specify the SA_RESTART flag, if defined. This is because the purpose of generating the SIGALRM signal is normally to place a timeout on an I/O operation, in which case, we want the blocked system call to be interrupted by the signal.

Call sigaction

We call sigaction and then return the old action for the signal as the return value of the signal function.

Throughout this text, we will use the signal function from the above definition.

Handling SIGCHLD Signals

The zombie state is to maintain information about the child for the parent to fetch later, which includes:

- Process ID of the child,
- Termination status,
- Information on the resource utilization of the child.

If a parent process of zombie children terminates, the parent process ID of all the zombie children is set to 1 (the init process), which will inherit the children and clean them up (init will wait for them, which removes the zombie).

Handling Zombies

Zombies take up space in the kernel and eventually we can run out of processes. Whenever we fork children, we must wait for them to prevent them from becoming zombies. We can establish a signal handler to catch SIGCHLD and call wait within the handler. We establish the signal handler by adding the following function call after the call to listen (in server's main function; it must be done before forking the first child and needs to be done only once.):

```
Signal (SIGCHLD, sig_chld);
```

The signal handler, the function sig_chld, is defined below:

```
#include    "unp.h"
void
sig_chld(int signo)
{
   pid_t   pid;
   int     stat;
   pid = wait(&stat);
   printf("child %d terminated\n", pid);
   return;

}
```

Note that calling standard I/O functions such as printf in a signal handler is not recommended. We call printf here as a diagnostic tool to see when the child terminates.

Compiling and running the program on Solaris

This program (tcpcliserv/tcpserv02.c) is compiled on Solaris 9 and uses the signal function from the system library.

```
solaris % tcpserv02 &              # start server in background
[2] 16939
solaris % tcpcli01 127.0.0.1         # then start client in foreground
hi there                          # we type this
hi there                          # and this is echoed
^D                               # we type our EOF character
child 16942 terminated              # output by printf in signal handler
accept error: Interrupted system call # main function aborts
```

The sequence of steps is as follows:

1. We terminate the client by typing our EOF character. The client TCP sends a FIN to the server and the server responds with an ACK.

2. The receipt of the FIN delivers an EOF to the child's pending readline. The child terminates.

3. The parent is blocked in its call to accept when the SIGCHLD signal is delivered. The sig_chld function executes (our signal handler), wait fetches the child's PID and termination status, and printf is called from the signal handler. The signal handler returns.

4. Since the signal was caught by the parent while the parent was blocked in a slow system call (accept), the kernel causes the accept to return an error of EINTR (interrupted system call). The parent does not handle this error (see server's main function), so it aborts.

From this example, we know that when writing network programs that catch signals, we must be cognizant of interrupted system calls, and we must handle them. In this example, the signal function provided in the standard C library does not cause an interrupted system call to be automatically restarted by the kernel. Some other systems automatically restart the interrupted system call. If we run the same example under BSD, using its library version of the signal function, the kernel restarts the interrupted system call and accept does not return an error. To handle this potential problem between different operating systems is one reason we define our own version of the signal function.

As part of the coding conventions used in this text, we always code an explicit return in our signal handlers, even though this is unnecessary for a function returning void. This reads as a reminder that the return may interrupt a system call.

Handling Interrupted System Calls

The term "slow system call" is used to describe any system call that can block forever, such as accept. That is, the system call need never return. Most networking functions fall into this category. Examples are:

- Accept: there is no guarantee that a server's call to accept will ever return, if there are no clients that will connect to the server.

- Read: the server's call to read in server's str_echo function will never return if the client never sends a line for the server to echo.

Other examples of slow system calls are reads and writes of pipes and terminal devices. A notable exception is disk I/O, which usually returns to the caller (assuming no catastrophic hardware failure).

When a process is blocked in a slow system call and the process catches a signal and the signal handler returns, the system call can return an error of EINT. Some kernels automatically restart some interrupted system calls. For portability, when we write a program that catches signals (most concurrent servers catch SIGCHLD), we must be prepared for slow system calls to return EINTR.

To handle an interrupted accept, we change the call to accept in server's main function, the beginning of the for loop, to the following:

```
for ( ; ; ) {
    clilen = sizeof (cliaddr);
    if ( (connfd = accept (listenfd, (SA *) &cliaddr, &clilen)) < 0) {
        if (errno == EINTR)
            continue;        /* back to for () */
        else
            err_sys ("accept error");
    }
```

Note that this accept is not our wrapper function Accept, since we must handle the failure of the function ourselves.

Restarting the interrupted system call is fine for:

- Accept
- Read
- Write
- Select
- Open

However, there is one function that we cannot restart: connect. If this function returns EINTR, we cannot call it again, as doing so will return an immediate error. When connect is interrupted by a caught signal and is not automatically restarted, we must call select to wait for the connection to complete.

wait and waitpid Functions

We can call wait function to handle the terminated child.

```
#include <sys/wait.h>
pid_t wait (int *statloc);
pid_t waitpid (pid_t pid, int *statloc, int options);


/* Both return: process ID if OK, 0 or–1 on error */
```

wait and waitpid both return two values: the return value of the function is the process ID of the terminated child, and the termination status of the child (an integer) is returned through the statloc pointer.

There are three macros that we can call that examine the termination status (see APUE):

- WIFEXITED: tells if the child terminated normally
- WIFSIGNALED: tells if the child was killed by a signal
- WIFSTOPPED: tells if the child was just stopped by job control

Additional macros let us then fetch the exit status of the child, or the value of the signal that killed the child, or the value of the job-control signal that stopped the child. We will use the WIFEXITED and WEXITSTATUS macros for this purpose.

If there are no terminated children for the process calling wait, but the process has one or more children that are still executing, then wait blocks until the first of the existing children terminates.

waitpid has more control over which process to wait for and whether or not to block:

- The *pid* argument specifies the process ID that we want to wait for. A value of -1 says to wait for the first of our children to terminate.
- The *options* argument specifies additional options. The most common option is WNOHANG, which tells the kernel not to block if there are no terminated children.

Difference between wait and waitpid

The following example illustrates the difference between the wait and waitpid functions when used to clean up terminated children.

We modify our TCP client as below, which establishes five connections with the server and then uses only the first one (sockfd[0]) in the call to str_cli. The purpose of establishing multiple connections is to spawn multiple children from the concurrent server.

```
#include    "unp.h"
int main(int argc, char **argv)
{
   int  i, sockfd[5];
   struct sockaddr_in  servaddr;

   if (argc != 2)
      err_quit("usage: tcpcli <IPaddress>");

   for (i = 0; i < 5; i++) {
      sockfd[i] = Socket(AF_INET, SOCK_STREAM, 0);

      bzero(&servaddr, sizeof(servaddr));
      servaddr.sin_family = AF_INET;
      servaddr.sin_port = htons(SERV_PORT);
      Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
      Connect(sockfd[i], (SA *) &servaddr, sizeof(servaddr));
   }
   str_cli(stdin, sockfd[0]);      /* do it all */
   exit(0);
}
```

When the client terminates, all open descriptors are closed automatically by the kernel (we do not call close, only exit), and all five connections are terminated at about the same time. This causes five FINs to be sent, one on each connection, which in turn causes all five server children to terminate at about the same time. This causes five SIGCHLD signals to be delivered to the parent at about the same time. This causes the problem under discussion.

We first run the server in the background and then our new client:

```
linux % tcpserv03 &
[1] 20419
linux % tcpcli04 127.0.0.1
hello                  # we type this
hello                  # and it is echoed
^D                     # we then type our EOF character
child 20426 terminated      # output by server
```

Only one printf is output, when we expect all five children to have terminated. If we execute ps, we see that the other four children still exist as zombies.

```
PID TTY         TIME CMD
20419 pts/6     00:00:00 tcpserv03
20421 pts/6     00:00:00 tcpserv03 <defunct>
20422 pts/6     00:00:00 tcpserv03 <defunct>
20423 pts/6     00:00:00 tcpserv03 <defunct>
```

Establishing a signal handler and calling wait from that handler are insufficient for preventing zombies. The problem is that all five signals are generated before the signal handler is executed, and the signal handler is executed only one time because Unix signals are normally not queued. This problem is nondeterministic. Dependent on the timing of the FINs arriving at the server host, the signal handler is executed two, three or even four times.

The correct solution is to call waitpid instead of wait. The code below shows the version of our sig_chld function that handles SIGCHLD correctly. This version works because we call waitpid within a loop, fetching the status of any of our children that have terminated, with the WNOHANG option, which tells waitpid not to block if there are running children that have not yet terminated. We cannot call wait in a loop, because there is no way to prevent wait from blocking if there are running children that have not yet terminated.

```
#include    "unp.h"
void
sig_chld(int signo)
{
   pid_t   pid;
   int     stat;

   while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0)
      printf("child %d terminated\n", pid);
   return;
}
```

The code below shows the final version of our server. It correctly handles a r eturn of EINTR from accept and i t es tablishes a s ignal h andler (code above) that calls waitpid for all terminated children.

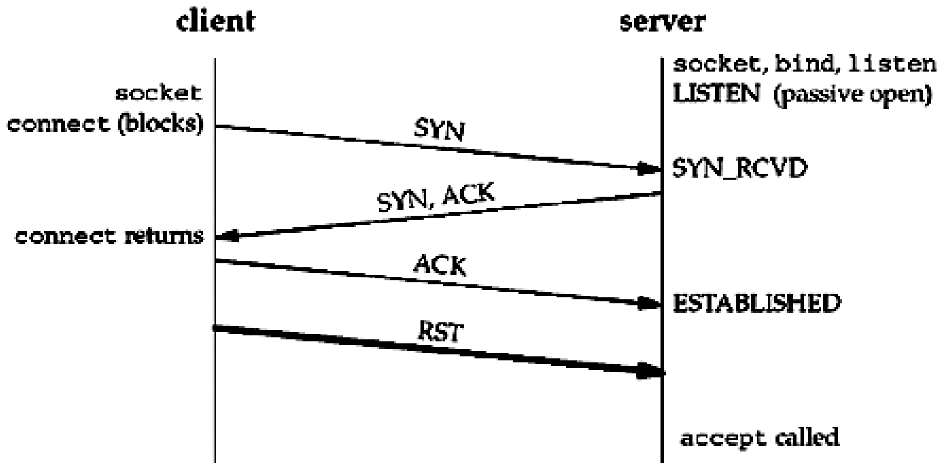
```
#include    "unp.h"
int main(int argc, char **argv)
{
   int    listenfd, connfd;
   pid_t              childpid;
   socklen_t          clilen;
   struct sockaddr_in  cliaddr, servaddr;
   void               sig_chld(int);
   listenfd = Socket(AF_INET, SOCK_STREAM, 0);
   bzero(&servaddr, sizeof(servaddr));
   servaddr.sin_family      = AF_INET;
   servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
   servaddr.sin_port        = htons(SERV_PORT);
   Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
   Listen(listenfd, LISTENQ);
   Signal(SIGCHLD, sig_chld);  /* must call waitpid() */
   for ( ; ; ) {
      clilen = sizeof(cliaddr);
      if ( (connfd = accept(listenfd, (SA *) &cliaddr, &clilen)) < 0) {
         if (errno == EINTR)
            continue;       /* back to for() */
         else
            err_sys("accept error");
      }
      if ( (childpid = Fork()) == 0) {    /* child process */
         Close(listenfd);    /* close listening socket */
         str_echo(connfd);   /* process the request */
         exit(0);
      }
      Close(connfd);          /* parent closes connected socket */
   }
}
```

The purpose of this section has been to demonstrate three scenarios that we can encounter with network programming:

- We must catch the SIGCHLD signal when forking child processes.
- We must handle interrupted system calls when we catch signals.
- A SIGCHLD handler must be coded correctly using waitpid to prevent any zombies from being left around.

Connection Abort before accept Returns

There is another condition similar to the interrupted system call that can cause accept to return a nonfatal error, in which case we should just call accept again. The sequence of packets shown below has been seen on busy servers (typically busy Web servers), where the server receives an RST for an ESTABLISHED connection before accept is called.



The three-way handshake completes, the connection is established, and then the client TCP sends an RST (reset). On the server side, the connection is queued by its TCP, waiting for the server process to call accept when the RST arrives. Sometime later, the server process calls accept.

An easy way to simulate this scenario is to start the server, have it call socket, bind, and listen, and then go to sleep for a short period of time before calling accept. While the server process is asleep, start the client and have it call socket and connect. As soon as connect returns, set the SO_LINGER socket option to generate the RST and terminate.

Termination of Server Process

We will now start our client/server and then kill the server child process, which simulates the crashing of the server process. We must be careful to distinguish between the crashing of the server *process* and the crashing of the server *host*.

The following steps take place:

1. We start the server and client and type one line to the client to verify that all is okay. That line is echoed normally by the server child.

2. We find the process ID of the server child and kill it. As part of process termination, all open descriptors in the child are closed. This causes a FIN to be sent to the client, and the client TCP responds with an ACK. This is the first half of the TCP connection termination.

3. The SIGCHLD signal is sent to the server parent and handled correctly.

4. Nothing happens at the client. The client TCP receives the FIN from the server TCP and responds with an ACK, but the problem is that the client process is blocked in the call to fgets waiting for a line from the terminal.

5. Running netstat at this point shows the state of the sockets.

```
linux % netstat -a | grep 9877
tcp    0    0 *:9877           *:*              LISTEN
tcp    0    0 localhost:9877    localhost:43604   FIN_WAIT2
tcp    1    0 localhost:43604   localhost:9877    CLOSE_WAIT
```

6. We can still type a line of input to the client. Here is what happens at the client starting from Step 1:

```
linux % tcpcli01 127.0.0.1  # start client
hello            # the first line that we type
hello              # is echoed correctly  we kill the server child on the
server host
another line       # we then type a second line to the client
str_cli : server terminated prematurely
```

When we type "another line," str_cli calls written and the client TCP sends the data to the server. This is allowed by TCP because the receipt of the FIN by the client TCP only indicates that the server process has closed its end of the connection and will not be sending any more data. The receipt of the FIN does not tell the client TCP that the server process has terminated (which in this case, it has).

When the server TCP receives the data from the client, it responds with an RST since the process that had that socket open has terminated. We can verify that the RST was sent by watching the packets with tcpdump.

7. The client process will not see the RST because it calls readline immediately after the call to writen and readline returns 0 (EOF) immediately because of the FIN that was received in Step 2. Our client is not expecting to receive an EOF at this point (str_cli) so it quits with the error message "server terminated prematurely."

8. When the client terminates (by calling err_quit in str_cli), all its open descriptors are closed.

- ❖ If the readline happens before the RST is received (as shown in this example), the result is an unexpected EOF in the client.
- ❖ If the RST arrives first, the result is an ECONNRESET ("Connection reset by peer") error return from readline.

The problem in this example is that the client is blocked in the call to fgets when the FIN arrives on the socket. The client is really working with two descriptors, the socket and the user input. Instead of blocking on input from only one of the two sources, it should block on input from either source.

SIGPIPE Signal

The rules are:

- When a process writes to a socket that has received an RST, the SIGPIPE signal is sent to the process. The default action of this signal is to terminate the process, so the process must catch the signal to avoid being involuntarily terminated.
- If the process either catches the signal and returns from the signal handler, or ignores the signal, the write operation returns EPIPE.

We can simulate this from the client by performing two writes to the server (which has sent FIN to the client) before reading anything back, with the first write eliciting the RST (causing the server to send an RST to the client). We must use two writes to obtain the signal, because the first write elicits the RST and the second write elicits the signal. It is okay to write to a socket that has received a FIN, but it is an error to write to a socket that has received an RST.

We modify our client as below:

```
#include    "unp.h"
void
str_cli(FILE *fp, int sockfd)
{
   char    sendline[MAXLINE], recvline[MAXLINE];

   while (Fgets(sendline, MAXLINE, fp) != NULL) {

       Writen(sockfd, sendline, 1);
       sleep(1);
       Writen(sockfd, sendline+1, strlen(sendline)-1);

       if (Readline(sockfd, recvline, MAXLINE) == 0)
           err_quit("str_cli: server terminated prematurely");

       Fputs(recvline, stdout);
   }
}
```

The writen is called two times. The intent is for the first writen to elicit the RST and then for the second writen to generate SIGPIPE.

Run the program on the Linux host:

```
linux % tcpclill 127.0.0.1
hi there       # we type this line
hi there       # this is echoed by the server
             # here we kill the server child
bye            # then we type this line
Broken pipe    # this is printed by the shell
```

We s tart t he cl ient, t ype i n o ne l ine, s ee t hat line ech oed co rrectly, an d then t erminate t he s erver c hild on t he s erver hos t. W e t hen t ype a nother line ("bye") and the shell tells us the process died with a SIGPIPE signal.

The r ecommended w ay t o ha ndle SIGPIPE depends on w hat t he application wants to do when this occurs:

- If there is nothing special to do, then setting the signal disposition to SIG_IGN is easy, assuming that subsequent output operations will catch the error of EPIPE and terminate.
- If special actions are needed, when the signal occurs (writing to a log file perhaps), then the signal should be caught and any desired actions can be performed in the signal handler.
- If multiple sockets are in use, the delivery of the signal will not tell us which socket encountered the error. If we need to know which write caused the error, then we must either ignore the signal or return from the signal handler and handle EPIPE from the write.

Check Your Progress

- What will this program print?

```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>

void response (int);
void response (int sig_no)
{
printf("%s is working\n",sys_siglist[sig_no]);
}
int main()
{
alarm(5);
 sleep(50);
printf("Sanfoundry\n");
 signal(SIGALRM,response);
return 0;
}
```

4.5 CRASHING AND REBOOTING OF SERVER HOST

This section discusses the case when we will establish a connection between the client and server and then assume the server host crashes and reboots. The easiest way to simulate this is to establish the connection, disconnect the server from the network, shut down the server host and then reboot it, and then reconnect the server host to the network. We do not want the client to see the server host shut down. The following steps take place:

- We start the server and then the client. We type a line to verify that the connection is established.
- The server host crashes and reboots.
- We type a line of input to the client, which is sent as a TCP data segment to the server host.
- When the server host reboots after crashing, its TCP loses all information about connections that existed before the crash. Therefore, the server TCP responds to the received data segment from the client with an RST.
- Our client is blocked in the call to *readline* when the RST is received, causing *readline* to return the error *ECONNRESET*.

Check your progress

1. How are zombies handled?
2. What happens when a system does not catch SIGTERM signal?

4.6 SHUTDOWN OF SERVER HOST

When a Unix system is shutdown, the *init* process normally sends the *SIGTERM* signal to all processes (this signal can be caught), waits some fixed amount of time (often between 5 and 20 seconds), and then sends *SIGKILL* signal (which we cannot catch) to any process still running. This gives all running processes a short amount of time to clean up and terminate.

If we do not catch *SIGTERM* and terminate, our server will be terminated by *SIGKILL* signal. When the process terminates, all the open descriptors are closed, and we then follow the same sequence of steps discussed under —termination of server process. We need to select the select or poll function in the client to have the client detect the termination of the server process as soon it occurs.

Problem: To write an algorithm for TCP echo client server

Solution: Server-

```
STEP 1: Start
STEP 2: Declare the variables for the socket
STEP 3: Specify the family, protocol, IP address and port number
STEP 4: Create a socket using socket() function
STEP 5: Bind the IP address and Port number
STEP 6: Listen and accept the client's request for the connection
STEP 7: Read the client's message
STEP 8: Display the client's message
STEP 9: Close the socket
STEP 10: Stop
```

Client-

```
STEP 1: Start
STEP 2: Declare the variables for the socket
STEP 3:  Specify the family, protocol, IP address and port number
STEP 4: Create a socket using socket() function
STEP 5: Call the connect() function
STEP 6: Read the input message
STEP 7: Send the input message to the server
STEP 8: Display the server's echo
STEP 9: Close the socket
STEP 10: Stop
```

4.7 SUMMARY

In this unit, we discuss TCP client/server mechanism with concept of TCP echo server. Client server scenario is explained using socket programming in Linux host. Various conditions that may occur during course of execution of server and client, starting from normal start up to server failure. Unit also covers various signal handling function and interrupt handling in server.

4.8 TERMINAL QUESTIONS

1. Which IP address for server is used at time of client initialization and why?

2. Write a code segment for sigaction function to establish disposition of a signal.

3. Explain the significance of wait and wait_pid along with their limitations.

4. Discuss the cases of crashing of server and server host.

5. Distinguish the behavior of server at time of server shut down and server reboot.

6. What is the output of this program?

```
#include<stdio.h>
#include<signal.h>
#include<stdlib.h>
void response (int);
void response (int sig_no)
```

```
{
   printf("%s\n",sys_siglist[sig_no]);
   printf("This is singal handler\n");
}
int main()
{
   pid_t child;
   int status;
   child = fork();
   switch (child){
      case -1 :
         perror("fork");
         exit (1);
      case 0 :
         kill(getppid(),SIGKILL);
         printf("I am an orphan process because my parent has
    been killed by me\n");
         printf("Handler failed\n");
         break;
      default :
         signal(SIGKILL,response);
         wait(&status);
         printf("The parent process is still alive\n");
         break;
   }
   return 0;
}
```

7. What is the output of this program?

```
#include<stdio.h>
#include<signal.h>
  void response (int);
  void response (int sig_no)
  {
     printf("%s\n",sys_siglist[sig_no]);
  }
```

```
int main()
{
    pid_t child;
    int status;
    child = fork();
    switch(child){
        case -1:
            perror("fork");
        case 0:
            break;
        default :
            signal(SIGCHLD,response);
            wait(&status);
            break;
    }
}
```

Uttar Pradesh Rajarshi Tandon
Open University

Bachelor of Computer Application

BCA-E7
Network Programming

Block

2

Course Preparation Committee

Dr. Prabhat Kumar **Author (Block 1,2)**
Assistant Professor, Department of IT
NIT Patna

Dr. Prabhat Ranjan **Author (Block 3,4)**
Assistant Professor, Department of Computer Science
Central University of South Bihar

Dr. Rajiv Mishra **Editor**
Associate Professor, Department of CSE
IIT Patna

Dr. Ashutosh Gupta (Director in Charge)
School of Computer & Information Sciences,
UPRTOU Prayagraj

Mr. Manoj Kumar Balwant **Coordinator**
Assistant Professor, (Computer Science)
School of sciences UPRTOU Prayagraj

© **UPRTOU, Prayagraj. 2019**
ISBN : 978-93-83328-11-6

Printed and Published by Dr. Arun Kumar Gupta Registrar, Uttar Pradesh Rajarshi Tandon Open University, 2019.

Printed By : Chandrakala Universal Pvt. Ltd. 42/7 Jawahar Lal Neharu Road, Prayagraj.

BLOCK INTRODUCTION

Unit 5: This unit deals with I/O multiplexing wherein the different I/O models are discussed. It also explains the select function, Batch Input and Buffering, Shutdown Function and Poll Function.

Unit 6: The various socket options are discussed in this unit. You will learn about the socket states, Generic socket options, IPV6 socket options, ICMP6 socket options along with TCP socket options.

Unit 7: The el ementary U DP s ockets are b riefed i n t his u nit. E cho server f unctions, l ost da tagram, l ack o f f low c ontrol i n U DP a nd determining out going i nterface w ith U DP a re a lso t he s ub-topics t o be focused.

Unit 8: This unit covers the DNS protocol of application layer that is responsible f or na me a nd a ddress c onversion. The g ethost b y N ame function, Resolver options, Functions and IPV6 support, Uname function and other networking information is also discussed.

UNIT-5 : I/O MULTIPLEXING

Structure

5.0 INTRODUCTION

I/O mu ltiplexing is th e c apability o f h andling mu ltiple I/O conditions (i.e., input is ready to be read or the descriptor is ready to take more output). There are various situations where I/O multiplexing is being required.

- When a client is handling multiple descriptors.
- When a client is to handle multiple sockets at the same time.
- If a TCP s erver h andles both a l istening s ocket a nd i ts c onnected sockets.
- If a server handles both TCP and UDP.
- If a s erver ha ndles multiple s ervices a nd pe rhaps m ultiple protocols.

5.1 OBJECTIVE

This unit imparts the basic knowledge of I/O multiplexing.

- The different kinds of I/O model and select function is discussed.
- Buffering and Batch Input are explained.
- Use and working of Shutdown and Poll functions is discussed.

5.2 I/O MODELS

There are five basic I/O models that are available in Unix.

- Blocking I/O
- Nonblocking I/O
- I/O multiplexing (select and poll)
- Signal driven I/O (SIGIO)
- Asynchronous I/O (the POSIX aio_ functions)

There are normally two distinct phases for an input operation:

1. Waiting for the data to be ready:

This involves waiting for data to arrive on the network. When it arrives, it is copied into a buffer within the kernel.

2. Copying the data from the kernel to the process.

This means copying the (ready) data from the kernel's buffer into our application buffer.

Blocking I/O Model

The most prevalent model for I/O is the blocking I/O model. By default, all sockets are blocking. The scenario is shown in the figure below:

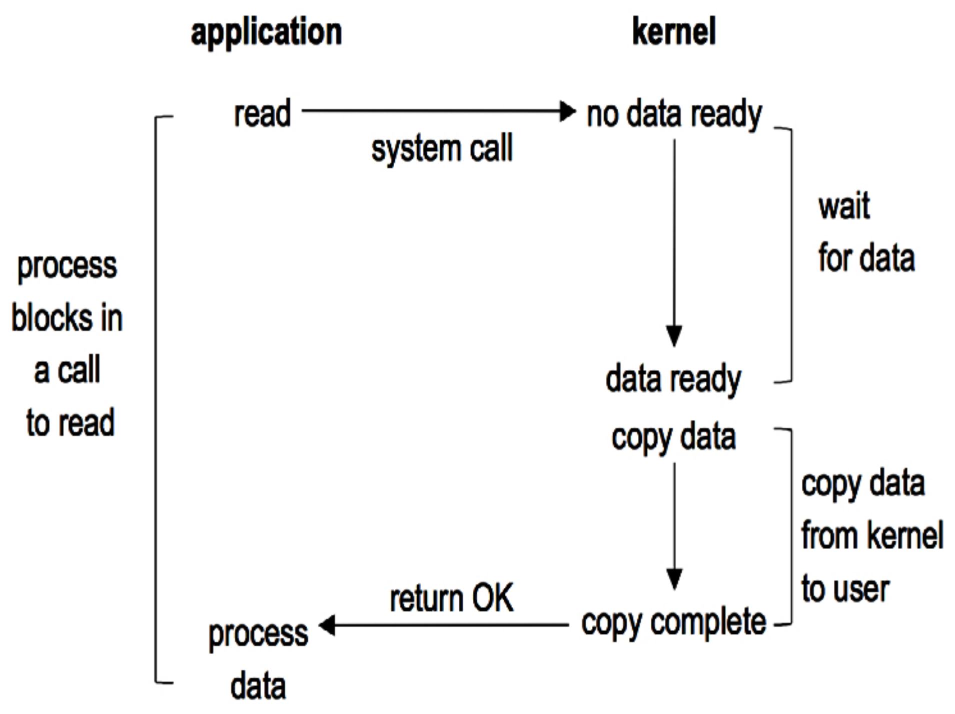


Figure 5.1: Blocking I/O Model

We use UDP for this example instead of TCP because with UDP, the concept of data being "ready" to read is simple, either an entire datagram has been received or it has not. With TCP it gets more complicated, as additional variables such as the socket's low-water mark come into play.

We also refer to recvfrom as a system call to differentiate between our application and the kernel, regardless of how recvfrom is implemented.

In the figure above, the process calls recvfrom and the system call does not return until the datagram arrives and is copied into our application buffer, or an error occurs. We say that the process is blocked the entire time from when it calls recvfrom until it returns. When recvfrom returns successfully, our application processes the datagram.

Nonblocking I/O Model

When a socket is set to be nonblocking, we are telling the kernel "when an I/O operation that I request cannot be completed without putting the process to sleep, do not put the process to sleep, but return an error instead". The figure is below:

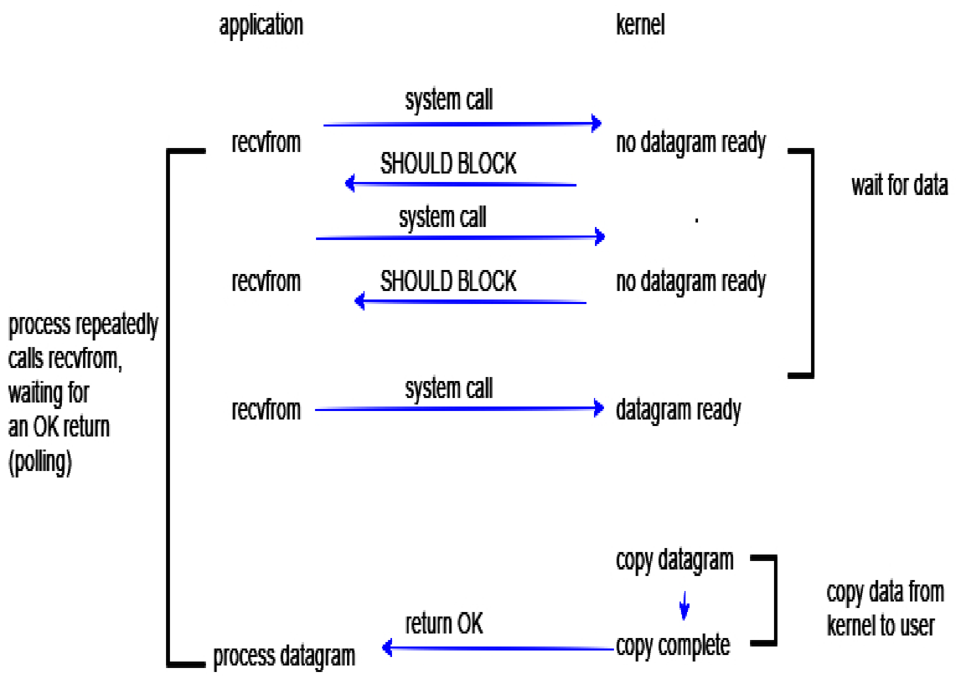


Figure 5.2: Nonblocking I/O Model

- For the first three recvfrom, there is no data to return and the kernel immediately returns an error of EWOULDBLOCK.
- For the fourth time we call recvfrom, a datagram is ready, it is copied into our application buffer, and recvfrom returns successfully. We then process the data.

When an application sits in a loop calling recvfrom on a nonblocking descriptor like this, it is called **polling**. The application is continually polling the kernel to see if some operation is ready. This is often a waste of CPU time, but this model is occasionally encountered, normally on systems dedicated to one function.

I/O Multiplexing Model

With **I/O multiplexing**, we call select or poll and block in one of these two system calls, instead of blocking in the actual I/O system call. The figure is a summary of the I/O multiplexing model:

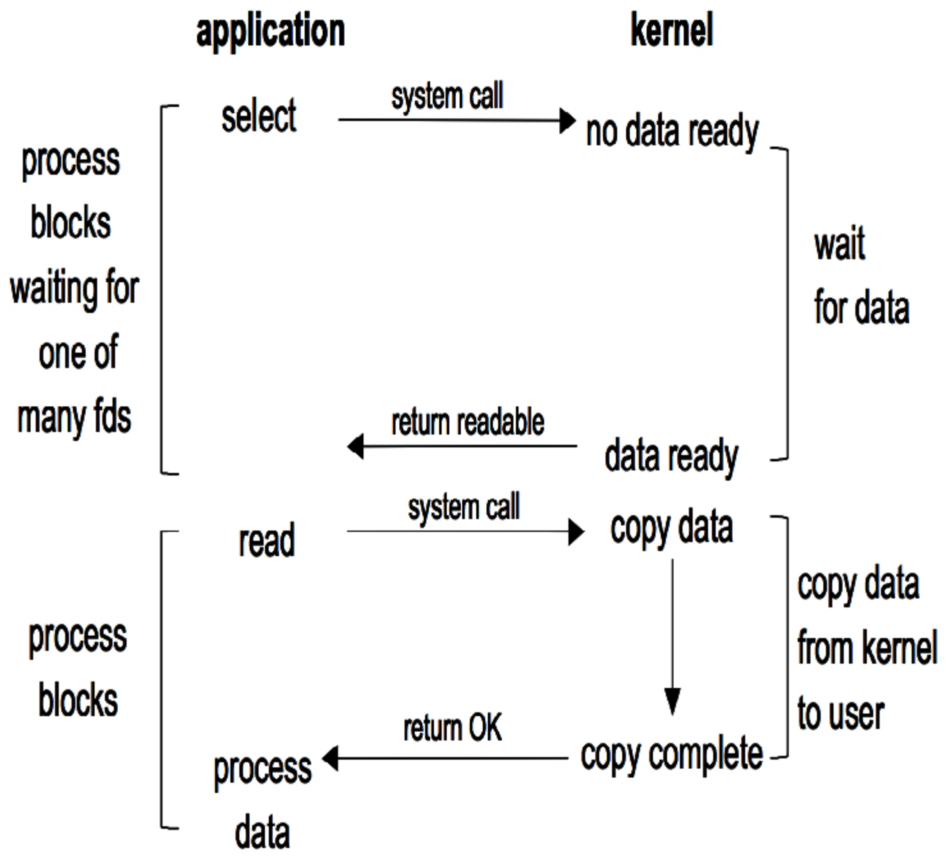


Figure 5.3: I/O Multiplexing Model

We block in a call to select, waiting for the datagram socket to be readable. When select returns that the socket is readable, we then call recvfrom to copy the datagram into our application buffer.

Multithreading with blocking I/O

Another closely related I/O model is to use multithreading with blocking I/O. That model very closely resembles the model described above, except that instead of using select to block on multiple file descriptors, the program uses multiple threads (one per file descriptor), and each thread is then free to call blocking system calls like recvfrom.

Signal-Driven I/O Model

The **signal-driven I/O model** uses signals, telling the kernel to notify us with the SIGIO signal when the descriptor is ready. The figure is below:

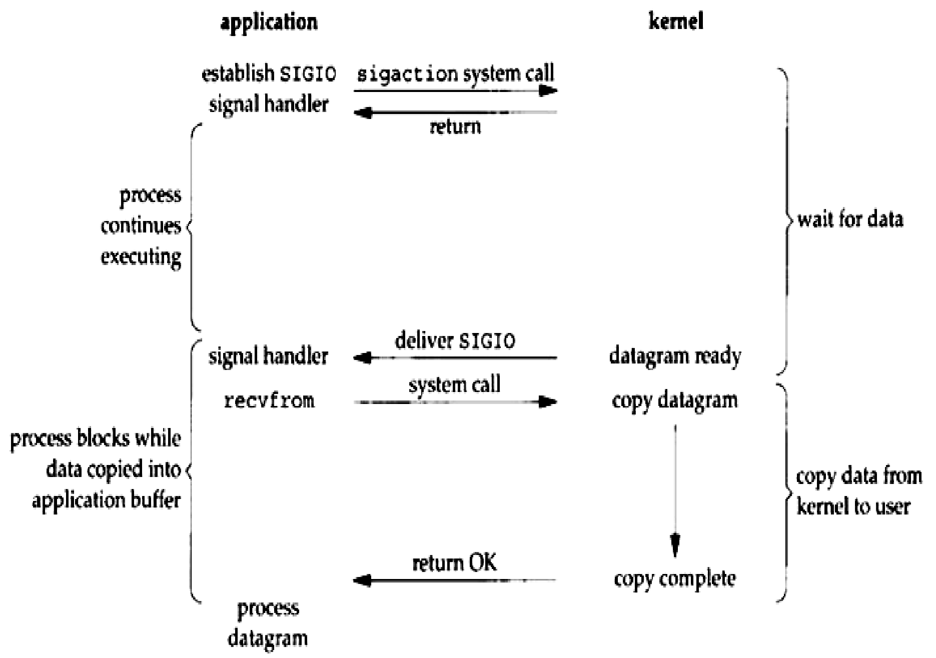


Figure 5.4: Signal-Driven I/O Model

- We first enable the socket for signal-driven I/O and install a signal handler us ing t he s igaction system c all. T he r eturn f rom th is system c all is imme diate a nd o ur p rocess c ontinues; it is n ot blocked.
- When t he d atagram i s r eady t o b e r ead, t he SIGIO s ignal i s generated for our process. We can either:
    - ❖ Read t he d atagram f rom t he s ignal h andler by calling recvfrom a nd t hen not ify the m ain l oop t hat t he da ta i s ready to be processed
    - ❖ Notify the main loop and let it read the datagram.

The advantage to this model is that we are not blocked while waiting for the datagram to arrive. The main loop can continue executing and just wait to be notified by the signal handler that either the data is ready to process or the datagram is ready to be read.

Asynchronous I/O Model

Asynchronous I /O is de fined b y t he P OSIX s pecification, a nd va rious differences i n t he *real-time* functions t hat a ppeared i n t he v arious standards which came together to form the current POSIX specification have been reconciled.

These functions work by telling the kernel to start the operation and to notify us when the entire operation (including the copy of the data from the kernel to our buffer) is complete. The main difference between this model and the signal-driven I/O model is that with signal-driven I/O, the kernel tells us when an I/O operation can be initiated, but with asynchronous I/O, the kernel tells us when an I/O operation is complete. See the figure below for example:

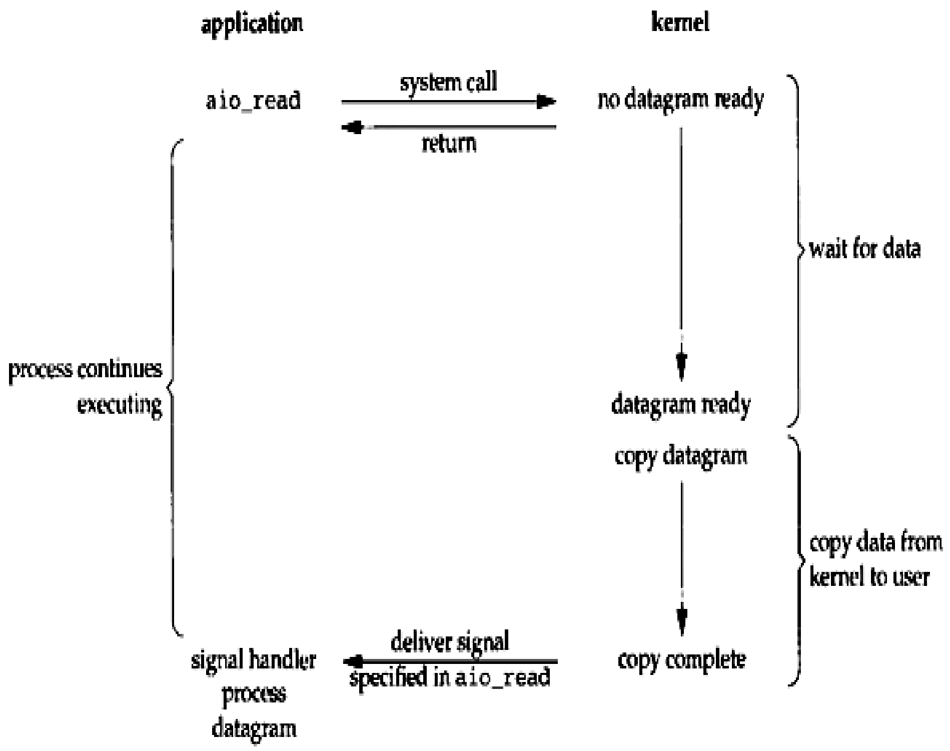

application
kernel
aio_read
system call
return
no datagram ready
wait for data
process continues executing
datagram ready
copy datagram
copy data from kernel to user
deliver signal specified in aio_read
copy complete
signal handler process datagram

Figure 5.5: Asynchronous I/O Model

- We call aio_read (the POSIX asynchronous I/O functions begin with aio_ or lio_) and pass the kernel the following:
  - ❖ Descriptor, buffer pointer, buffer size (the same three arguments for read),
  - ❖ File offset (similar to lseek),
  - ❖ Method to notify us when the entire operation is complete.

  This system call returns immediately and our process is not blocked while waiting for the I/O to complete.

- We assume in this example that we ask the kernel to generate some signal when the operation is complete. This signal is not generated until the data has been copied into our application buffer, which is different from the signal-driven I/O model.

Comparison of the I/O Models

The main difference between the first four models is the first phase, as the second phase in the first four models is the same: the process is blocked in a call to recvfrom while the data is copied from the kernel to the caller's buffer. Asynchronous I/O, however, handles both phases and is different from the first four. The figure below is a comparison of the five different I/O models.

Synchronous I/O versus Asynchronous I/O

POSIX defines these two terms as follows:

- A synchronous I/O operation causes the requesting process to be blocked until that I/O operation completes.
- An asynchronous I/O operation does not cause the requesting process to be blocked.

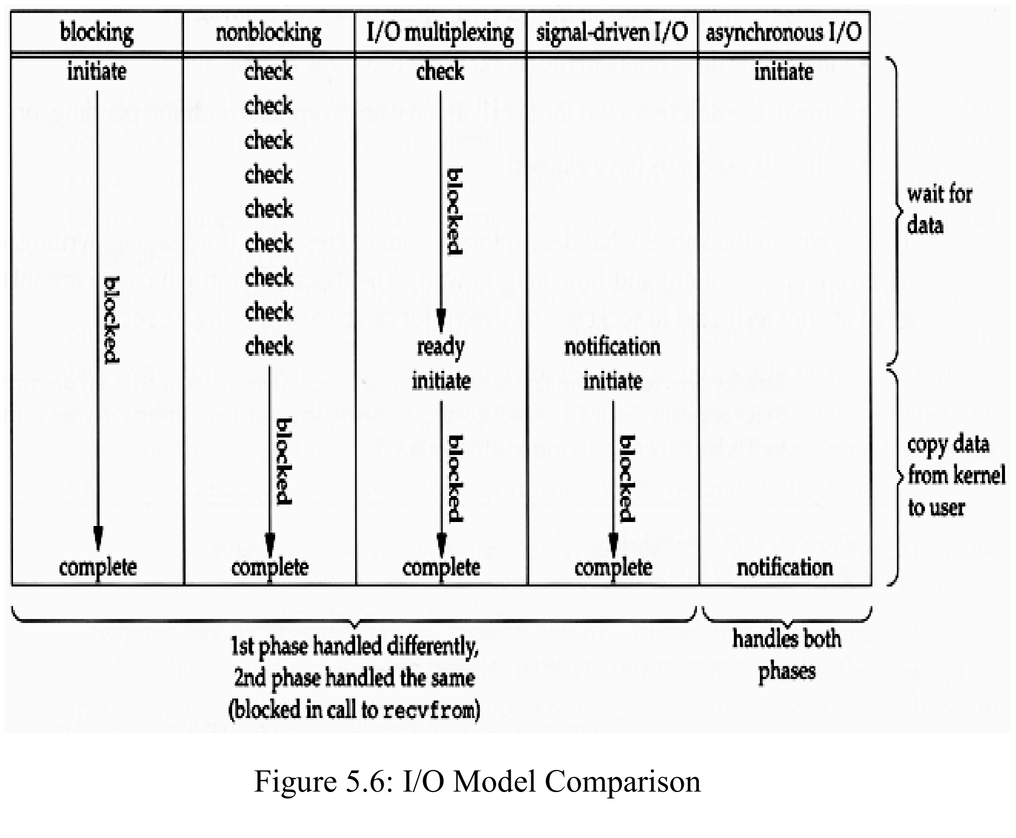


Figure 5.6: I/O Model Comparison

Using these definitions, the first four I/O models (blocking, nonblocking, I/O multiplexing, and signal-driven I/O) are all synchronous because the actual I/O operation (recvfrom) blocks the process. Only the asynchronous I/O model matches the asynchronous I/O definition.

5.3 SELECT FUNCTION

The select function allows the process to instruct the kernel to either:

- Wait for any one of multiple events to occur and to wake up the process only when one or more of these events occurs, or
- When a specified amount of time has passed.

This means that we tell the kernel what descriptors we are interested in (for reading, writing, or an exception condition) and how long to wait. The descriptors in which we are interested are not restricted to sockets; any descriptor can be tested using select.

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

#include <sys/time.h>

int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,

        const struct timeval *timeout);

/* Returns: positive count of ready descriptors, 0 on timeout, –1 on error */
```

The *timeout* argument

The *timeout* argument tells the kernel how long to wait for one of the specified descriptors to become ready. A timeval structure specifies the number of seconds and microseconds.

```
struct timeval  {

 long   tv_sec;        /* seconds */

 long   tv_usec;       /* microseconds */

};
```

There are three possibilities for the *timeout*:

1. **Wait forever** (*timeout* is specified as a null pointer). Return only when one of the specified descriptors is ready for I/O.

2. **Wait up to a fixed amount of time** (*timeout* points to a timeval structure). Return when one of the specified descriptors is ready for I/O, but do not wait beyond the number of seconds and microseconds specified in the timeval structure.

3. **Do not wait at all** (*timeout* points to a timeval structure and the timer value is 0, i.e. the number of seconds and microseconds specified by the structure are 0). Return immediately after checking the descriptors. This is called polling.

Note:

- The wait in the first two scenarios is normally interrupted if the process catches a signal and returns from the signal handler. For portability, we must be prepared for select to return an error of EINTRif we are catching signals. Berkeley-derived kernels never automatically restart select.
- Although the timeval structure has a microsecond field tv_usec, the actual resolution supported by the kernel is often more coarse. Many Unix kernels round the timeout value up to a multiple of 10 ms. There is also a scheduling latency involved, meaning it takes some time after the timer expires before the kernel schedules this process to run.
- On some systems, the timeval structure can represent values that are not supported by select; it will fail with EINVAL if the tv_sec field in the timeout is over 100 million seconds.
- The const qualifier on the *timeout* argument means it is not modified by select on return.

The descriptor sets arguments *

The three middle arguments, *readset*, *writeset*, and *exceptset*, specify the descriptors that we want the kernel to test for reading, writing, and exception conditions. There are only two exception conditions currently supported:

- The arrival of out-of-band data for a socket.
- The presence of control status information to be read from the master side of a pseudo-terminal that has been put into packet mode. (Not covered in UNP)

select uses descriptor sets, typically an array of integers, with each bit in each integer corresponding to a descriptor. For example, using 32-bit integers, the first element of the array corresponds to descriptors 0 through 31, the second element of the array corresponds to descriptors 32 through 63, and so on. All the implementation details are irrelevant to the application and are hidden in the fd_set datatype and the following four macros:

```
void FD_ZERO(fd_set *fdset);        /* clear all bits in fdset */

void FD_SET(int fd, fd_set *fdset);  /* turn on the bit for fd in fdset */

void FD_CLR(int fd, fd_set *fdset);  /* turn off the bit for fd in fdset */

int FD_ISSET(int fd, fd_set *fdset); /* is the bit for fd on in fdset ? */
```

We allocate a descriptor set of the fd_set datatype, we set and test the bits in the set using these macros, and we can also assign it to another descriptor set across an equals sign (=) in C.

An array of integers using one bit per descriptor, is just one possible way to implement select. Nevertheless, it is common to refer to the individual descriptors within a descriptor set as bits, as in "turn on the bit for the listening descriptor in the read set."

The following example defines a variable of type fd_set and then turns on the bits for descriptors 1, 4, and 5:

```
fd_set rset;
FD_ZERO(&rset);         /* initialize the set: all bits off */
FD_SET(1, &rset);       /* turn on bit for fd 1 */
FD_SET(4, &rset);       /* turn on bit for fd 4 */
FD_SET(5, &rset);       /* turn on bit for fd 5 */
```

It is important to initialize the set, since unpredictable results can occur if the set is allocated as an automatic variable and not initialized.

Any of the middle three arguments to select, *readset*, *writeset*, or *exceptset*, can be specified as a null pointer if we are not interested in that condition. Indeed, if all three pointers are null, then we have a higher precision timer than the normal Unix sleep function. The poll function provides similar functionality.

The *maxfdp1* argument

The *maxfdp1* argument specifies the number of descriptors to be tested. Its value is the maximum descriptor to be tested plus one. The descriptors 0, 1, 2, up through and including *maxfdp1*–1 are tested.

The constant FD_SETSIZE, defined by including <sys/select.h>, is the number of descriptors in thefd_set datatype. Its value is often 1024, but few programs use that many descriptors.

The reason the *maxfdp1* argument exists, along with the burden of calculating its value, is for efficiency. Although each fd_set has room for many descriptors, typically 1,024, this is much more than the number used by a typical process. The kernel gains efficiency by not copying unneeded portions of the descriptor set between the process and the kernel, and by not testing bits that are always 0.

readset, *writeset*, and *exceptset* as value-result arguments

select modifies the descriptor sets pointed to by the *readset*, *writeset*, and *exceptset* pointers. These three arguments are value-result arguments. When we call the function, we specify the values of the descriptors that we are interested in, and on return, the result indicates which descriptors are ready. We use theFD_ISSET macro on return to test a specific descriptor in an fd_set structure. Any descriptor that is not ready on return will have its corresponding bit cleared in the descriptor set. To handle this, we turn on all the bits in which we are interested in all the descriptor sets each time we call select.

Return value of select

The return value from this function indicates the total number of bits that are ready across all the descriptor sets. If the timer value expires before any of the descriptors are ready, a value of 0 is returned. A return value of –1 indicates an error (which can happen, for example, if the function is interrupted by a caught signal).

Check your progress

1. Explain nonblocking I/O model.
2. What are the possibilities for timeouts?

5.4 BATCH INPUT AND BUFFERING

If we consider the network between the client and server as a full-duplex pipe, with requests going from the client to the server and replies in the reverse direction, then the following figure shows our stop-and-wait mode:

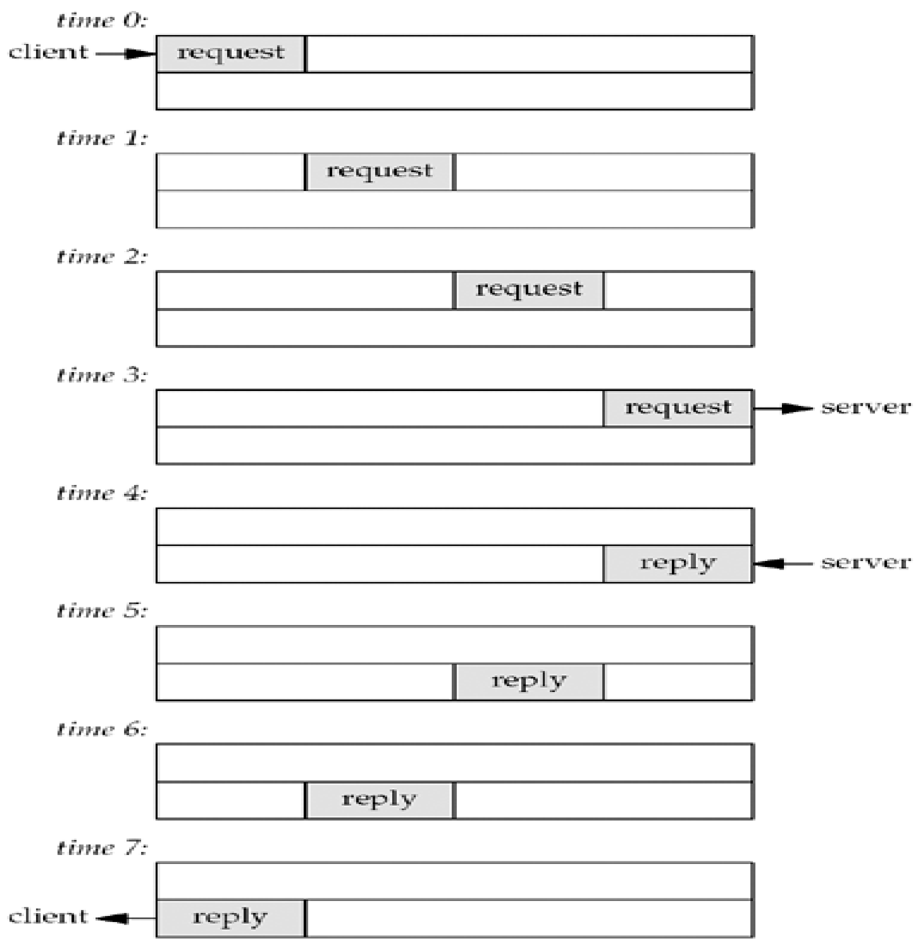


Figure 5.7: Stop and Wait Mode

Note that this figure:

- Assumes that there is no server processing time and that the size of the request is the same as the reply
- Shows show only the data packets, ignoring the TCP acknowledgments that are also going across the network

A request is sent by the client at time 0 and we assume an RTT of 8 units of time. The reply sent at time 4 is received at time 7.

This stop-and-wait mode is fine for interactive input. The problem is: if we run our client in a batch mode, when we redirect the input and output, however, the resulting output file is always smaller than the input file (and they should be identical for an echo server).

Batch mode

To see what's happening, realize that in a batch mode, we can keep sending requests as fast as the network can accept them. The server processes them and sends back the replies at the same rate. This leads to the full pipe at time 7, as shown below:

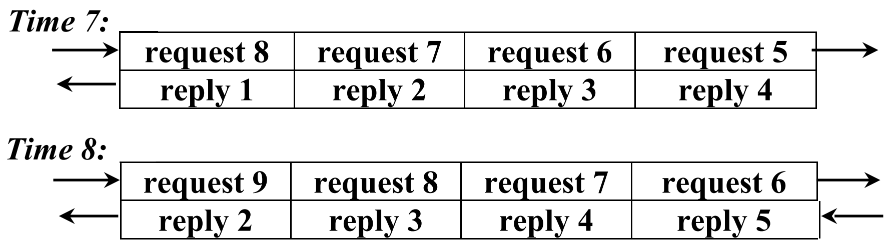


Figure 5.8: Batch Mode

We assume:

- After sending the first request, we immediately send another, and then another
- We can keep sending requests as fast as the network can accept them, along with processing replies as fast as the network supplies them.

Assume that the input file contains only nine lines. The last line is sent at time 8, as shown in the above figure. But we cannot close the connection after writing this request because there are still other requests and replies in the pipe. The cause of the problem is our handling of an EOF on input: The function returns to the main function, which then terminates. But in a batch mode, an EOF on input does not imply that we have finished reading from the socket; there might still be requests on the way to the server, or replies on the way back from the server.

The solution is to close one-half of the TCP connection by sending a FIN to the server, telling it we have finished sending data, but leave the socket descriptor op en f or r eading. T his i s don e w ith t he shutdown f unction, described in the next section.

Buffering concerns

When s everal l ines of i nputs a re a vailable f rom t he s tandard i nput select will cause the code to read the input using fgets which will read the available lines into a buffer used by stdio. But, fgets only returns a single line a nd le aves a ny remaining d ata s itting in th e s tdio buf fer. T he following code writes that single line to the server and then select is called again to wait for more work, even if there are additional lines to consume in the stdio buffer. The reason is that select knows nothing of the buffers used b y s tdio;it w ill only s how r eadability from th e v iewpoint o f the read system c all, n ot c alls lik e fgets. T hus, mixing s tdio a nd select is considered very error-prone and should only be done with great care.

The s ame p roblem e xists w ith readline in th is e xample (str_cli function). Instead of da ta be ing h idden f rom select in a s tdio buf fer, i t i s hi dden in readline's b uffer. A function th at g ives v isibility in to readline's buf fer, so one possible solution is to modify our code to use that function before calling select to s ee i f da ta ha s a lready b een read but not c onsumed. But again, the complexity grows out of hand quickly when we have to handle the case where the readline buffer contains a partial line (meaning we still need to read more) as well as when it contains one or more complete lines (which we can consume).

pselect Function

The pselect function w as i nvented b y P OSIX a nd i s now s upported b y many of the Unix variants.

```
#include <sys/select.h>
#include <signal.h>
#include <time.h>
int pselect (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
        const struct timespec *timeout, const sigset_t *sigmask);
/* Returns: count of ready descriptors, 0 on timeout, –1 on error */
```

pselect contains two changes from the normal select function:

1. pselect uses t he timespec structure (another P OSIX i nvention) instead of the timeval structure. The tv_nsec member of the newer structure s pecifies n anoseconds, w hereas t he tv_usec member o f the older structure specifies microseconds.

```
struct timespec {
 time_t tv_sec;      /* seconds */
 long   tv_nsec;     /* nanoseconds */
};
```

2. pselect adds a sixth argument: a pointer to a signal mask. This allows the program to disable the delivery of certain signals, test some global variables that are set by the handlers for these now-disabled signals, and then call pselect, telling it to reset the signal mask.

With regard to the second point, consider the following example:

Our program's signal handler for SIGINT just sets the global intr_flag and returns. If our process is blocked in a call to select, the return from the signal handler causes the function to return with errno set to EINTR. But when select is called, the code looks like the following:

```
if (intr_flag)
   handle_intr();      /* handle the signal */
/* signals occurring in here are lost */
if ( (nready = select( ... )) < 0) {
   if (errno == EINTR) {
      if (intr_flag)
         handle_intr();
   }
   ...
}
```

The problem is that between the test of intr_flag and the call to select, if the signal occurs, it will be lost if select blocks forever.

With pselect, we can now code this example reliably as:

```
sigset_t newmask, oldmask, zeromask;
sigemptyset(&zeromask);
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);
sigprocmask(SIG_BLOCK, &newmask, &oldmask); /* block SIGINT */
if (intr_flag)
   handle_intr();     /* handle the signal */
if ( (nready = pselect ( ... , &zeromask)) < 0) {
   if (errno == EINTR)  {
      if (intr_flag)
```

```
        handle_intr ();
    }
    ...
}
```

Before testing the intr_flag variable, we block SIGINT. When pselect is called, it replaces the signal mask of the process with an empty set (i.e., zeromask) and then checks the descriptors, possibly going to sleep. But when pselect returns, the signal mask of the process is reset to its value before pselect was called (i.e., SIGINT is blocked).

5.5 SHUTDOWN FUNCTION

The normal way to terminate a network connection is to call the close function. But, there are two limitations with close that can be avoided with shutdown:

1. close decrements the descriptor's reference count and closes the socket only if the count reaches 0. With shutdown, we can initiate TCP's normal connection termination sequence (the four segments beginning with a FIN in), regardless of the reference count.

2. close terminates both directions of data transfer, reading and writing. Since a TCP connection is full-duplex, there are times when we want to tell the other end that we have finished sending, even though that end might have more data to send us. This is the scenario we encountered in the previous section with batch input to our str_cli function. The figure below shows the typical function calls in this scenario.

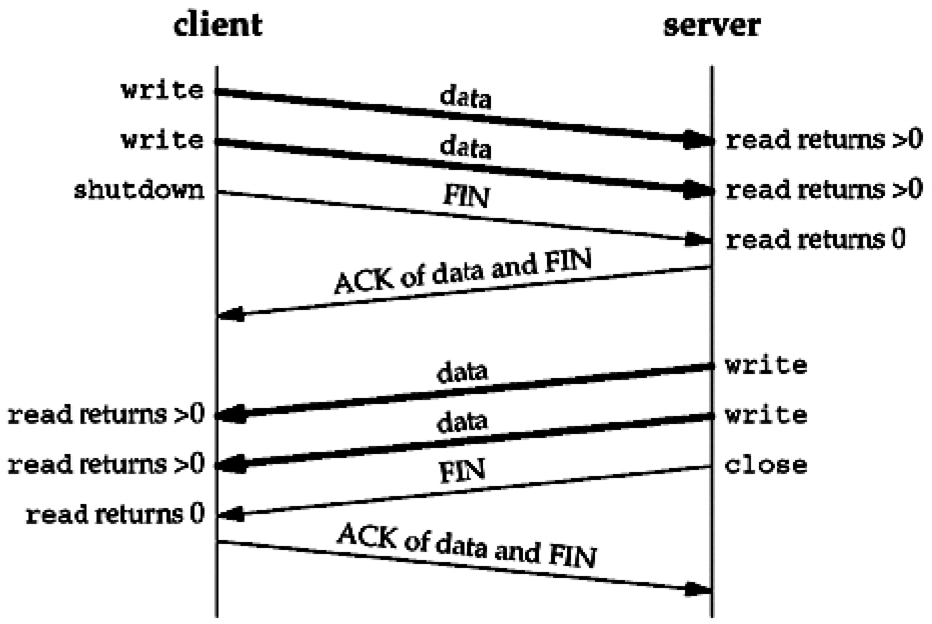


Figure 5.9: Network Termination using shutdown Function

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

int shutdown(int sockfd, int howto);

/* Returns: 0 if OK, –1 on error */
```

The action of the function depends on the value of the *howto* argument:

- SHUT_RD: **The read half of the connection is closed.** No more data can be received on the socket and any data currently in the socket receive buffer is discarded. The process can no longer issue any of the read functions on the socket. Any data received after this call for a TCP socket is acknowledged and then silently discarded.
- SHUT_WR: **The write half of the connection is closed.** In the case of TCP, this is called a **half-close**. Any data currently in the socket send buffer will be sent, followed by TCP's normal connection termination sequence. As we mentioned earlier, this closing of the write half is done regardless of whether or not the socket descriptor's reference count is currently greater than 0. The process can no longer issue any of the write functions on the socket.
- SHUT_RDWR: **The read half and the write half of the connection are both closed.** This is equivalent to calling shutdown twice: first with SHUT_RD and then with SHUT_WR.

The three SHUT_*xxx* names are defined by the POSIX specification. Typical values for the howto argument that you will encounter will be 0 (close the read half), 1 (close the write half), and 2 (close the read half and the write half).

Check your progress

1. What are the possible values of *howto* argument?
2. Explain *pselect* function?

5.6 POLL FUNCTION

Poll provides functionality that is similar to select, but poll provides additional information when dealing with STREAMS devices.

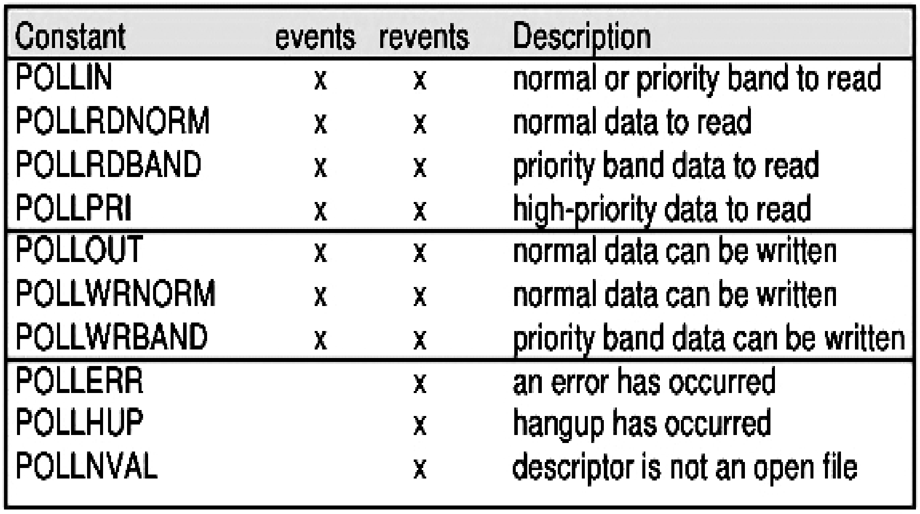
```
#include <poll.h>
int poll (struct pollfd *fdarray, unsigned long nfds, int timeout);
/* Returns: count of ready descriptors, 0 on timeout, –1 on error */
```

Arguments:

The first argument (*fdarray*) is a pointer to the first element of an array of structures. Each element is apollfd structure that specifies the conditions to be tested for a given descriptor, fd.

```
struct pollfd {
  int     fd;      /* descriptor to check */
  short   events;  /* events of interest on fd */
  short   revents; /* events that occurred on fd */
};
```

The c onditions t o be t ested a re s pecified b y t he events member, and t he function r eturns t he s tatus f or t hat de scriptor i n t he corresponding revents member. T his da ta s tructure (having t wo v ariables per descriptor, one a value and one a result) avoids value-result arguments (the middle three arguments forselect are value-result). Each of these two members is composed of one or more bits that specify a certain condition. The f ollowing f igure s hows t he c onstants us ed t o s pecify t he events flag and to test the revents flag.



Constant	events	revents	Description
POLLIN	x	x	normal or priority band to read
POLLRDNORM	x	x	normal data to read
POLLRDBAND	x	x	priority band data to read
POLLPRI	x	x	high-priority data to read
POLLOUT	x	x	normal data can be written
POLLWRNORM	x	x	normal data can be written
POLLWRBAND	x	x	priority band data can be written
POLLERR		x	an error has occurred
POLLHUP		x	hangup has occurred
POLLNVAL		x	descriptor is not an open file

Figure 5.10: Summary of Constants specifying events and revents flags

The first four constants deal with input, the next three deals with output, and the f inal t hree d eals w ith er rors. T he f inal t hree cannot b e s et in events, b ut ar e al ways r eturned i n revents when t he corresponding condition exists.

With r egard t o T CP a nd U DP s ockets, t he f ollowing c onditions cause poll to r eturn t he s pecified revent. Unfortunately, P OSIX l eaves many hol es (optional w ays t o r eturn t he s ame c ondition) i n i ts de finition ofpoll.

- All regular TCP data and all UDP data is considered normal.

- TCP's out-of-band data is considered priority band.
- When the read half of a TCP connection is closed (e.g., a FIN is received), this is also considered normal data and a subsequent read operation will return 0.
- The presence of an error for a TCP connection can be considered either normal data or an error (POLLERR). In either case, a subsequent read will return –1 with errno set to the appropriate value. This handles conditions such as the receipt of an RST or a timeout.
- The availability of a new connection on a listening socket can be considered either normal data or priority data. Most implementations consider this normal data.
- The completion of a nonblocking connect is considered to make a socket writable.

The number of elements in the array of structures is specified by the *nfds* argument.

The *timeout* argument specifies how long the function is to wait before returning. A positive value specifies the number of milliseconds to wait. The constant INFTIM (wait forever) is defined to be a negative value.

Return values from poll:

- –1 if an error occurred
- 0 if no descriptors are ready before the timer expires
- Otherwise, it is the number of descriptors that have a nonzero revents member.

If we are no longer interested in a particular descriptor, we just set the fd member of the pollfd structure to a negative value. Then the events member is ignored and the revents member is set to 0 on return.

This section is discusses the TCP echo server from using poll instead of select.

In the select version we allocate a client array along with a descriptor set named rset. With poll, we must allocate an array of pollfd structures to maintain the client information instead of allocating another array. We handle the fd member of this array the same way we handled the client array in the selection version: a value of –1 means the entry is not in use; otherwise, it is the descriptor value. Any entry in the array of pollfd structures passed to poll with a negative value for the fd member is just ignored.

```
/* include fig01 */
#include    "unp.h"
#include    <limits.h>      /* for OPEN_MAX */
Int main(int argc, char **argv)
{
   int              i, maxi, listenfd, connfd, sockfd;
   int              nready;
   ssize_t           n;
   char              buf[MAXLINE];
   socklen_t          clilen;
   struct pollfd      client[OPEN_MAX];
   struct sockaddr_in  cliaddr, servaddr;
   listenfd = Socket(AF_INET, SOCK_STREAM, 0);
   bzero(&servaddr, sizeof(servaddr));
   servaddr.sin_family      = AF_INET;
   servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
   servaddr.sin_port       = htons(SERV_PORT);
   Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
   Listen(listenfd, LISTENQ);
   client[0].fd = listenfd;
   client[0].events = POLLRDNORM;
   for (i = 1; i < OPEN_MAX; i++)
      client[i].fd = -1;      /* -1 indicates available entry */
   maxi = 0;                  /* max index into client[] array */
/* end fig01 */
/* include fig02 */
   for ( ; ; ) {
      nready = Poll(client, maxi+1, INFTIM);
      if (client[0].revents & POLLRDNORM) {   /* new client connection */
         clilen = sizeof(cliaddr);
         connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
#ifdef  NOTDEF
         printf("new client: %s\n", Sock_ntop((SA *) &cliaddr, clilen));
```

```
#endif
        for (i = 1; i < OPEN_MAX; i++)
          if (client[i].fd < 0) {
            client[i].fd = connfd;  /* save descriptor */
            break;
          }
        if (i == OPEN_MAX)
          err_quit("too many clients");
        client[i].events = POLLRDNORM;
        if (i > maxi)
          maxi = i;            /* max index in client[] array */
        if (--nready <= 0)
          continue;            /* no more readable descriptors */
      }
      for (i = 1; i <= maxi; i++) {   /* check all clients for data */
        if ( (sockfd = client[i].fd) < 0)
          continue;
        if (client[i].revents & (POLLRDNORM | POLLERR)) {
          if ( (n = read(sockfd, buf, MAXLINE)) < 0) {
            if (errno == ECONNRESET) {
                /* connection reset by client */
#ifdef  NOTDEF
              printf("client[%d] aborted connection\n", i);
#endif
              Close(sockfd);
              client[i].fd = -1;
            } else
              err_sys("read error");
          } else if (n == 0) {
              /* connection closed by client */
#ifdef  NOTDEF
            printf("client[%d] closed connection\n", i);
#endif
            Close(sockfd);
            client[i].fd = -1;
            } else
                Writen(sockfd, buf, n);
            if (--nready <= 0)
                break;          /* no more readable descriptors */
        }
    }
  }
}
```

This code does the following:

- **Allocate array of pollfd structures.**
  We declare OPEN_MAX elements in our array of pollfd structures. Determining the maximum number of descriptors that a process can have opened at any one time is difficult. One way is to call the POSIX sysconf function with an argument of _SC_OPEN_MAX (as described in APUE) and then dynamically allocate an array of the appropriate size.
- **Initialize.** We u se th e f irst e ntry in th e client array fo r t he listening socket and set the descriptor for the remaining entries to –1. We also set the POLLRDNORM event for this descriptor, to be notified b ypoll when a ne w c onnection i s r eady t o b e a ccepted. The v ariable maxi contains t he l argest i ndex o f t he client array currently in use.
- **Call poll, check for new connection.** We cal l poll to w ait for either a new connection or data on existing connection.
  - ❖ When a ne w c onnection is accep ted, w e f ind t he f irst available entry in the client array by looking for the first one with a negative descriptor.
  - ❖ We start the search with the index of 1, since client[0] is used for the listening socket.
  - ❖ When an available entry is found, we save the descriptor and set the POLLRDNORM event.
- **Check for data on an existing connection.** The two return events that we check for are POLLRDNORM and POLLERR. We did not s et POLLERR in t he ev ents m ember b ecause i t i s al ways returned w hen t he c ondition i s true. T he r eason w e ch eck for POLLERR is be cause s ome i mplementations r eturn t his e vent

when an RST is received for a connection, while others just return POLLRDNORM. In either case, we call read and if an error has occurred, it will return an error. When an existing connection is terminated by the client, we just set the fd member to –1.

Check your progress

1. Explain with figure stop and wait mode for request reply messages between client and server in full duplex network.
2. What are the possible return values of the timeout argument and what do they mean?

5.7 SUMMARY

In this unit, we study about the five different models in Unix for I/O:

- Non-blocking
- Blocking
- I/O multiplexing
- Signal-driven I/O
- Asynchronous I/O

We see that the Blocking I/O is the prevalently applied default. It is also observed that the POSIX specification is widely used for defining true asynchronous I/O. The select function is used for I/O multiplexing. The descriptors, the maximum waiting time along with the maximum descriptor number incremented by one are provided to the select function. Readability is specified by the calls to select. It is also observed that arrival of out of band data is the only exception that arises during socket processes. The select function dictates a limit on the length of time for which a block in a function persists. This salient feature can be applied to administer the time limit length for input operations. Similar functionality is also provided by the poll function. It also describes information related to STREAM devices. Though, the select function as well as the poll function is necessary for POSIX but the select function is preferably used in most cases.

5.8 TERMINAL QUESTIONS

1. Define I/O multiplexing. Under which circumstances is it used?
2. Compare the different I/O multiplexing models.

3. What a re t he f ive f unctions us ed t o pe rform f ile I/O on a U nix System? Elaborate each function with example.

4. What is timeout argument? Explain the *timeval* structure.

5. Describe t he s teps i nvolved i n C l anguage while as signing a descriptor s et t o a nother one a cross t he e quals s ign w hen t he descriptor set is an array of integers.

6. Differentiate between select and poll functions.

7. What i s t he c onsequence w hen t he s econd a rgument pr ovided t o shutdown is SHUT_RD?

8. Describe the conditions under which an application calls shutdown using t he a rgument of S HUT_RDWR a s a n a lternative t o s imply calling close.

UNIT-6 : SOCKET OPTIONS

Structure

6.0 INTRODUCTION

A socket is an endpoint of a connection across a computer network which is responsible to deliver data packet to appropriate process or thread. It is a combination of IP address and port number. Sockets are communication points on the same or different computers to exchange data. These are supported by Unix, Windows, Mac, and many other operating systems. To be more precise, it's a way to talk to other computers using standard Unix file descriptors. In Unix, every I/O action is done by writing or reading a file descriptor. A file descriptor is just an integer associated with an open file and it can be a network connection, a text file, a terminal, or something else. To a programmer, a socket looks and behaves much like a low-level file descriptor. This is because commands such as read() and write() work with sockets in the same way they do with files and pipes.

6.1 OBJECTIVE

To create awareness about the different socket options available for the development of application.

- Firstly getsockop and setsockopt functions are discussed, then the different socket states and generic socket options are explained further.

- This unit also sheds light on socket options for IPV6 and ICMP6. Lastly TCP socket options are specified and the chapter terminates with summary.

6.2 GETSOCKOPT AND SETSOCKOPT FUNCTION

There are various ways to control a socket:

getsockopt() is used to retrieve options associated with the socket. If an option is to be interpreted by the TCP protocol, protocolLevel is set to the TCP protocol number. The parameters optionValuePtr and optionLengthPtr is used to identify a buffer in which the value(s) for the requested option(s) are to be returned. The socket in use may require the process to have appropriate privileges to use the getsockopt() function. The option_name argument specifies a single option to be retrieved.

It can be one of the following values which have been defined in <*sys/socket.h*>:

Socket Level Options

The following options are recognized at the socket level:

Protocol Level **Options**	**Data Type**	**Description**
SO_BINDTODEVICE	string	The device name, as set with tfAddInterface(), will be stored as a null-terminated string in the buffer pointed to by optionValuePtr.
SO_DONTROUTE	int	Enable/disable routing bypass for outgoing messages. **Default 0.**
SO_ERROR	int	Retrieve the socket error. ***This option is for getsockopt() only!***
SO_KEEPALIVE	int	Enable/disable keep connections alive. **Default 0 (disable).**
SO_LINGER	linger	Linger on close if data is present. **Default ON with a linger time of 60 seconds.**
SO_OOBINLINE	int	Enable/disable reception of out-of-band data in band. **Default is 0.**

SO_RCVBUF	unsigned long	The buffer size for input. **Default is 8192 bytes.**
SO_RCVLOWAT	unsigned long	The l ow water m ark f or receiving in bytes. **Default value is 1.**
SO_REUSEADDR	int	Enable this socket option to bind the same port number to multiple sockets us ing di fferent local IP addresses. **Default 0 (disable).**
SO_REUSEPORT	int	Enable this socket option to bind the s ame l ocal IP ad dress an d port t o m ultiple s ockets. If multiple U DP s ockets ha ve t he SO_REUSEPORT option s et, then t hose s ockets c an bi nd t o the s ame l ocal IP ad dress, an d local UDP port. **Default 0 (disable).**
SO_SNDBUF	unsigned long	The buffer size for output. **Default is 8192 bytes.**
SO_SNDLOWAT	unsigned long	The low water mark for sending in bytes. **Default value is 2048.**
TM_SO_RCVCOPY	unsigned int	**TCP socket:** fraction u se o f a receive b uffer b elow w hich w e try and a ppend t o a previous receive b uffer i n t he s ocket receive queue. **UDP socket:** fraction u se o f a receive b uffer b elow w hich w e try an d co py t o a n ew r eceive buffer, if there is already at least a buffer in the receive queue. **Default value is 4 (25%).**
TM_SO_SNDAPPEND	unsigned int	**TCP socket only**. T hreshold i n bytes of s end buf fer be low, which w e try and append to the previous send buffer in the TCP send que ue. O nly us ed w ith send(), not w ith tfZeroCopySend(). **Default value is 128 bytes.**

TM_SO_SND_DGRAMS	unsigned int	The num ber o f n on-TCP datagrams that can be queued for send on a socket. **Default is 8 datagrams.**
TM_SO_RCV_DGRAMS	unsigned int	The number of non-TCP datagrams that can be queued for receive on a socket. **Default is 8 datagrams.**
SO_UNPACKEDDATA	int	**TI C3x and C5x DSP platforms only**. If this option is enabled, all socket data will be sent and received in byte unpacked format. If this option is disabled, all socket data will be sent in a byte packed format, as received from the network. **Default 0 (disable)**

Table 6.1: Socket level options

IP Level Options

The following options are recognized at the IP level:

protocolLevel **Options**	**Data Type**	**Description**
IPO_HDRINCL	int	This is a toggle option used on raw s ockets onl y. If t he va lue is non -zero, it in structs th e Treck s tack t hat t he u ser i s including t he IP he ader w hen sending data. **Default 0**
IPO_RCV_TOS	unsigned char	Received IP type of service on the connection (from the last IP datagram arrived on t he connection.)
IPO_TOS	unsigned char	IP type of service. **Default 0**
IPO_TTL	unsigned char	IP Time To Live in seconds. **Default 64**
IPO_SRCADDR	ttUserIpAddress	Set t he IP s ource a ddress f or the connection. **Default: The first multi-home IP address on the outgoing**

		interface
IPO_MULTICAST_TTL	unsigned char	Change the default IP TTL for outgoing multicast datagrams.
IPO_MULTICAST_IF	in_addr	Specify a configured IP address that will uniquely identify the outgoing interface for multicast datagrams sent on this socket. A zero IP address parameter indicates that we want to reset a previously set outgoing interface for multicast packets sent on that socket.
IPO_ADD_MEMBERSHIP	ip_mreq	Add group multicast IP address to given interface (see struct ip_mreq data type).
IPO_DROP_MEMBERSHIP	ip_mreq	Delete group multicast IP address to given interface.
IP_BLOCK_SOURCE	ip_mreq_source	Block data from a given source to a given multicast group (mute).
IP_UNBLOCK_SOURCE	ip_mreq_source	Unblock data from a given source to a given multicast group (un-mute).
IP_ADD_SOURCE_MEMBERSHIP	ip_mreq_source	Join a source-specific group.
IP_DROP_SOURCE_MEMBERSHIP	ip_mreq_source	Leave a source-specific group.
MCAST_JOIN_GROUP	group_req	Add group multicast IP address to given interface. This option also supports IPPROTO_IPV6.
MCAST_LEAVE_GROUP	group_req	Delete group multicast IP address to given interface. This option also supports IPPROTO_IPV6.
MCAST_BLOCK_SOURCE	group_source_req	Block data from a given source to a given multicast group (mute). This option also supports IPPROTO_IPV6.
MCAST_UNBLOCK_SOURCE	group_source_req	Unblock data from a given source to a given multicast

		group (un-mute). T his opt ion also supports IPPROTO_IPV6.
MCAST_JOIN_SOURCE_GROUP	group_source_req	Join a s ource-specific g roup. This opt ion a lso s upports IPPROTO_IPV6.
MCAST_LEAVE_SOURCE_GROUP	group_source_req	Leave a source-specific group. This option also supports IPPROTO_IPV6.
IP_RCV_TOS	unsigned char	Retrieve the IP header TOS from a packet on a TCP connection, after the TCP connection has been established.

Table 6.2: IP level options

6.3 SOCKET STATES

The following socket options are inherited by a connected TCP socket from the listening socket:

- SO_DEBUG: It i s a boolean option w hich r eports w hether debugging information is being recorded.
- SO_ACCEPTCONN: It is a bool ean opt ion w hich r eports s ocket listening has been enabled.
- SO_BROADCAST: It is boolean option to report that transmission of broadcast messages is being supported by the protocol.
- SO_REUSEADDR: It i s a bool ean opt ion w hich r eports w hether the r ules us ed i n va lidating a ddresses s upplied t o bind() should allow reuse of local addresses.
- SO_KEEPALIVE: It r eports w hether connections ar e k ept active with pe riodic t ransmission of m essages. If t he connected s ocket fails to r espond to these m essages, the c onnection s hall be broken and threads writing to that socket shall be notified with a SIGPIPE signal. This option shall store a n int value. This i s a lso a bool ean option.
- SO_LINGER: It r eports w hether t he s ocket l ingers on close() if data is p resent. If S O_LINGER is s et, th e s ystem s hall b lock th e calling thread dur ing close() until it c an tr ansmit th e d ata o r u ntil the e nd of t he i nterval i ndicated b y t he *l_linger* member, whichever comes first. If SO_LINGER is not specified, and close() is i ssued, t he s ystem h andles t he c all in a w ay that a llows th e calling thread to continue as quickly as possible. This option shall store a linger structure.

- SO_OOBINLINE: Reports whether the socket leaves received out-of-band data (data marked urgent) inline. This option shall store an int value. This is a Boolean option.
- SO_SNDBUF: Reports send buffer size information. This option shall store an int value.
- SO_RCVBUF: Reports receive buffer size information. This option shall store an int value.
- SO_ERROR: Reports information about error status and clears it. This option shall store an int value.
- SO_TYPE: Reports the socket type. This option shall store an int value.
- SO_DONTROUTE: Reports whether outgoing messages bypass the standard routing facilities. The destination shall be on a directly-connected network, and messages are directed to the appropriate network interface according to the destination address. The effect, if any, of this option depends on what protocol is in use. This option shall store an int value. This is a Boolean option.
- SO_RCVLOWAT: Reports the minimum number of bytes to process for socket input operations. The default value for SO_RCVLOWAT is 1. If SO_RCVLOWAT is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. (They may return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different from that returned; for example, out-of-band data.) This option shall store an int value. Note that not all implementations allow this option to be retrieved.
- SO_RCVTIMEO: Reports the timeout value for input operations. This option shall store a timeval structure with the number of seconds and microseconds specifying the limit on how long to wait for an input operation to complete. If a receive operation has blocked for this much time without receiving additional data, it shall return with a partial count or *errno* set to [EAGAIN] or [EWOULDBLOCK] if no data was received. The default for this option is zero, which indicates that a receive operation shall not time out. Note that not all implementations allow this option to be retrieved.

6.4 GENERIC SOCKET OPTIONS

Protocol-independent code (or not by any existing protocol module) within the kernel are used to handle these generic socket options.

Some options are socket type specific. For example, the SO_BROADCAST socket option is called "generic," which is used only for datagram sockets.

SO_BROADCAST Socket Option

This option controls the ability of the process to send broadcast messages. Only datagram sockets support broadcasting and networks that support the concept of a broadcast message (e.g., Ethernet, token ring, etc.).

Applications that doesn't support broadcast mechanism are not allowed to do so because applications have to set this socket option before initializing broadcast. For example, a UDP application might take the destination IP address as a command-line argument, but the application never intended for a user to type in a broadcast address. Rather than forcing the application to try to determine if a given address is a broadcast address or not, the test is in the kernel: If the destination address is a broadcast address and this socket option is not set, EACCES is returned.

SO_DEBUG Socket Option

This option is supported only by TCP. When enabled for a TCP socket, the kernel keeps track of detailed information about all the packets sent or received by TCP for the socket. These are kept in a circular buffer_within the kernel that can be examined with the trpt program.

SO_DONTROUTE Socket Option

This option specifies that outgoing packets are to bypass the normal routing mechanisms of the underlying protocol. The destination must be on a directly-connected network, and messages are directed to the appropriate network interface according to the destination address. For example, in case of IPv4 packets are routed through unique local interfaces and if the interface is not found, ENETUNREACH is returned.

The equivalent of this option can also be applied to individual datagrams using the MSG_DONTROUTE flag with the send, sendto or sendmsg functions. This option is often used by routing daemons (e.g., routed and gated) to bypass the routing table and force a packet to be sent out a particular interface.

SO_ERROR Socket Option

This option is one that can be fetched but cannot be set. When an error occurs on a socket, the protocol module in a Berkeley-derived kernel sets a variable namedso_error for that socket to one of the standard Unix Exxx values. This is called the *pending error* for the socket. The process can be immediately notified of the error in one of two ways:

1. If the process is blocked in a call to select on the socket, for either readability or writability, select returns with either or both conditions set.

2. If the process is using signal-driven I/O, the SIGIO signal is generated for either the process or the process group.

The process can then obtain the value of so_error by fetching the SO_ERROR socket option. The integer value returned by getsockopt is the pending error for the socket. The value of so_error is then reset to 0 by the kernel.

- If so_error is nonzero when the process calls read and there is no data to return, read returns –1 with errno set to the value of so_error. The value of so_error is then reset to 0. If there is data queued for the socket, that data is returned by read instead of the error condition.

- If so_error is nonzero when the process calls write, –1 is returned with errno set to the value ofso_error and so_error is reset to 0.

SO_KEEPALIVE Socket Option

When the keep-alive option is set for a TCP socket and no data has been exchanged across the socket in either direction for two hours, TCP automatically sends a keep-alive probe to the peer. This probe is a TCP segment to which the peer must respond. One of three scenarios results:

1. The peer responds with the expected ACK. The application is not notified (since everything is okay). For further two hour of inactivity TCP will send a probe.

2. Peer host's crash or reboot is reported via RST to the local TCP. Sockets remaining errors are set to ECONNRESET and the socket is closed.

3. If peer doesn't response to keep-alive probe, Berkeley-derived TCPs send 8 additional probes with gap period of 75 seconds. After 11 minutes and 15 seconds of inactivity, TCP will give up.

SO_LINGER Socket Option

This option specifies how the close function operates for a connection-oriented protocol (for TCP, but not for UDP). By default, close returns immediately, but if there is any data still remaining in the socket send buffer, the system will try to deliver the data to the peer.

The SO_LINGER socket option can change this default. This option requires the following structure to be passed (as the *optval argument) between the user process and the kernel. It is defined by including<sys/socket.h>.

```
struct linger {
 int   l_onoff;       /* 0=off, nonzero=on */
  int   l_linger;      /* linger time, POSIX specifies units as seconds
*/ };
```

Calling setsockopt leads to one of the following three scenarios, depending on the values of the two structure members:

1. **1.** If l_onoff is 0, the option is turned off. The value of l_linger is ignored and the previously discussed TCP default applies: close returns immediately.

2. **2.** If l_onoff is nonzero and l_linger is zero, TCP aborts the connection when it is closed.

    - ❖ In this case, TCP discards any data still remaining in the socket send buffer and sends an RST to the peer, not the normal four-packet connection termination sequence.
    - ❖ This scenario avoids TCP's TIME_WAIT state, but leaves open the possibility of another incarnation of this connection being created within 2MSL seconds and having old duplicate segments from the just-terminated connection being incorrectly delivered to the new incarnation.
    - ❖ Occasional USENET postings advocate the use of this feature just to avoid the TIME_WAIT state and to be able to restart a listening server even if connections are still in use with the server's well-known port. This should NOT be done and could lead to data corruption, as detailed in RFC1337. Instead, the SO_REUSEADDR socket option should always be used in the server before the call to bind. We should make use of the TIME_WAIT state to let old duplicate segments expire in the network rather than trying to avoid it.
    - ❖ There are certain circumstances which warrant using this feature to send an abortive close. One example is an RS-232 terminal server, which might hang forever in CLOSE_WAIT trying to deliver data to a stuck terminal port, but would properly reset the stuck port if it got an RST to discard the pending data.

3. **3.** If l_onoff is nonzero and l_linger is nonzero, then the kernel will linger when the socket is closed.

    - ❖ In this scenario, if there is any data still remaining in the socket send buffer, the process is put to sleep until either:
        1. **1.** All the data is sent and acknowledged by the peer TCP, or
        2. **2.** The linger time expires.
    - ❖ If the socket has been set to nonblocking, it will not wait for the close to complete, even if the linger time is nonzero. When using this feature of the SO_LINGER option, it is important for the application to check the return value from close, because if the linger time expires before the remaining data is sent and

acknowledged, close returns EWOULDBLOCK and any remaining data in the send buffer is discarded.

Given the above three scenarios, consider the situations when a close on a socket returns. Assume that the client writes data to the socket and then calls close.

Default operation of close: it returns immediately *

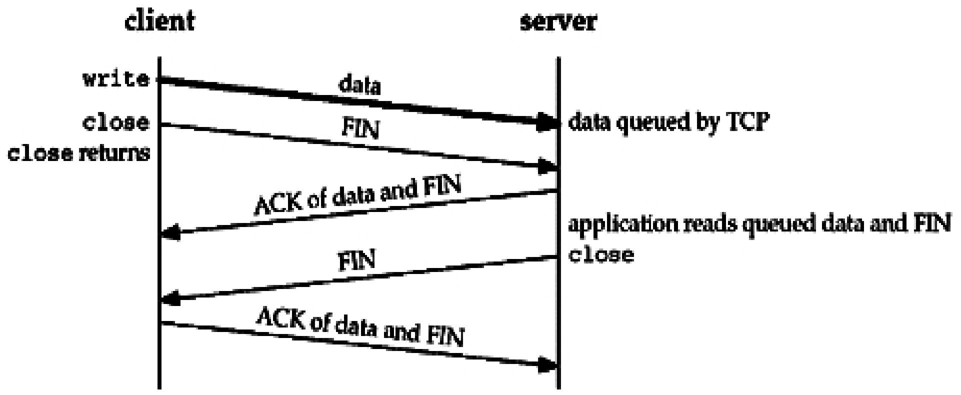


Figure 6.1: Default operation of close

Assume that when the client's data arrives, the server is temporarily busy, so the data is added to the socket receive buffer by its TCP. Similarly, the next segment, the client's FIN, is also added to the socket receive buffer. But by default, the client's close returns immediately. In the scenario shown above, the client's close can return before the server reads the remaining data in its socket receive buffer. Therefore, it is possible for the server host to crash before the server application reads this remaining data, and the client application will never know.

Close with SO_LINGER socket option set and l_linger a positive value *

The client can set the SO_LINGER socket option, specifying some positive linger time. When this occurs, the client's close does not return until all the client's data and its FIN have been acknowledged by the server TCP, as shown in the figure below.

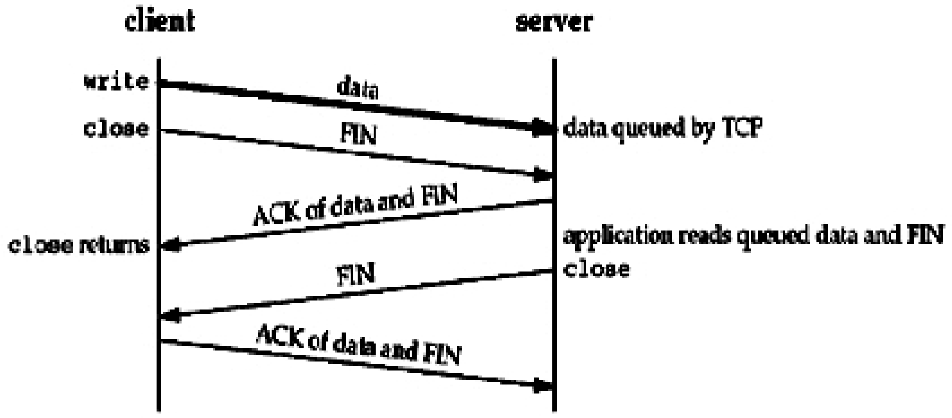


Figure 6.2: close with SO_LINGER option

But this still has the same problem as: The server host can crash before the server application reads its remaining data, and the client application will never know. Worse, the following figure shows what can happen when the SO_LINGER option is set to a value that is too low.

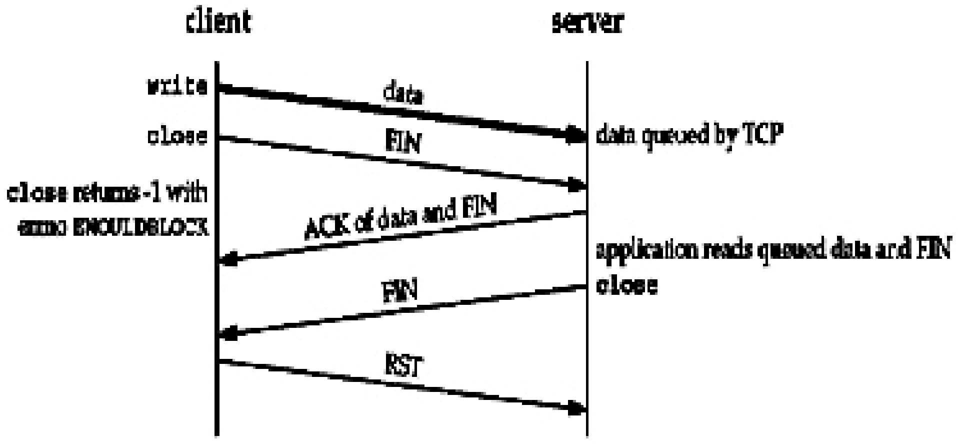


Figure 6.3: close with SO_LINGER option set to very low value

It is important to know that a successful return from close, with the SO_LINGER socket option set, only tells us that the data we sent (and our FIN) have been acknowledged by the peer TCP. This does not tell us whether the peer application has read the data. If we do not set the SO_LINGER socket option, we do not know whether the peer TCP has acknowledged the data.

6.4 IPV6 Socket Options

The following options are recognized at the IPv6 level:

protocolLevel **Options**	**Data Type**	**Description**
IPV6_V6ONLY	int	Force the socket to be IPv6-only. Normally, when running with both TM_USE_IPV4 and TM_USE_IPV6 defined, a socket created with AF_INET6 is able to communicate via both IPv4 and IPv6. Setting this socket option forces the socket to communicate via IPv6 only.
IPV6_JOIN_GROUP	ipv6_mreq	Join an IPv6 multicast group.
IPV6_LEAVE_GROUP	ipv6_mreq	Leave an IPv6 multicast group.

MCAST_JOIN_GROUP	group_req	Join an IPv6 multicast group. This option also supports IPPROTO_IP.
MCAST_LEAVE_GROUP	group_req	Leave an IPv6 multicast group. This option also supports IPPROTO_IP.
MCAST_BLOCK_SOURCE	group_source_req	Block data from a given source to a given IPv6 multicast group (mute). This option also supports IPPROTO_IP.
MCAST_UNBLOCK_SOURCE	group_source_req	Unblock data from a given source to a given IPv6 multicast group (un-mute). This option also supports IPPROTO_IP.
MCAST_JOIN_SOURCE_GROUP	group_source_req	Join a source-specific IPv6 group. This option also supports IPPROTO_IP.
MCAST_LEAVE_SOURCE_GROUP	group_source_req	Leave a source-specific IPv6 group. This option also supports IPPROTO_IP.
IPV6_MULTICAST_HOPS	unsigned int	This option allows the user to set the hop limit field in the IPv6 header for multicast packets sent via this socket. **Default 1**
IPV6_MULTICAST_IF	int	Specify the interface index of the outgoing interface for multicast datagrams sent on this socket. An interface index of 0 indicates that we want to reset a previously set outgoing interface for multicast packets sent on this socket.
IPV6_UNITCAST_HOPS	int	This option allows the user to set the hop limit field in the IPv6 header for unicast packets sent via this socket.

Table 6.3: IPv6 socket options

Check your progress

1. Describe SO_REUSEPORT option of sockets.
2. Enlist the generic socket options.

6.5 ICMP6 SOCKET OPTIONS

The ICMP6_FILTER socket option can be used by a RAW application to filter out ICMPv6 message types that it does not need to receive. ICMPv6 provides function comparable to ICMPv4 plus IGMPv4 and ARPv4 functionality. An application might be interested in receiving only a subset of the messages received for ICMPv6.

This option is enabled or disabled with a setsockopt(). The option value provides a 256-bit array of message types that should be filtered. To disable the option, the setsockopt() should be issued with an option length of 0. This causes the TCP/IP protocol stack's default filter to be in effect.

A getsockopt() with this option returns the value set by a setsockopt(). If a setsockopt() has not been done, the TCP/IP protocol stack's default filter is returned.

6.6 TCP SOCKET OPTIONS

The following options are recognized at the TCP level:

***Protocol Level* Options**	**Data Type**	**Description**
TCP_KEEPALIVE	int	Sets the idle time in seconds for a TCP connection before it starts sending keep alive probes. Note that keep alive probes will be sent only if the SO_KEEPALIVE socket option is enabled. **Default 7,200 seconds.**
TCP_MAXRT	int	Sets the amount of time in seconds before the connection is broken once TCP starts retransmitting, or probing a zero window when the peer does not respond. A TCP_MAXRT value of 0 means the system default, and -1 means retransmit forever. If a positive value is specified, it may be rounded up to the connection next retransmission time. Note that unless the TCP_MAXRT value is -1 (transmit forever), the connection can also be broken if the number of

		maximum retransmission TM_TCP_MAX_REXMIT has been reached. See TM_TCP_MAX_REXMIT below. **Default 0.** Meaning: use the system default of TM_TCP_MAX_REXMIT times network computed round trip time for an established connection. For a non-established connection, since there is no computed round trip time yet, the connection can be broken when either 75 seconds or when TM_TCP_MAX_REXMIT times default network round trip time have elapsed, whichever occurs first).
TCP_MAXSEG	int	Sets the maximum TCP segment size sent on the network. Note that the TCP_MAXSEG value is the maximum amount of data (including TCP options, but not the TCP header) that can be sent per segment to the peer. This means that the amount of user data sent per segment is the value given by the TCP_MAXSEG option minus any enabled TCP option (for example 12 bytes for a TCP time stamp option). The TCP_MAXSEG value can be decreased or increased prior to a connection establishment, but it is not recommended to set it to a value higher than the IP MTU minus 40 bytes (for example 1460 bytes on Ethernet), since this would cause fragmentation of TCP segments. Note: setting the TCP_MAXSEG option will inhibit the automatic computation of that value by the system based on the IP MTU (which avoids fragmentation), and will also inhibit Path MTU Discovery. After the connection has started, this value cannot be changed. Note also that the TCP_MAXSEG value cannot be set below 64 bytes. Default value is IP MTU minus 40 bytes. **Default is IP MTU minus 40 bytes.**
TCP_NODELAY	int	Set this option value to a non-zero value, to disable the Nagle algorithm that buffers the sent data inside the TCP. Useful to allow client's TCP to send small packets as soon as possible (like mouse clicks). **Default 0.**

TCP_NOPUSH	int	Set this option value to a non-zero value, to force TCP to delay sending any TCP data until a full sized segment is buffered in the TCP buffers. Useful for applications that send continuous big chunks of data like FTP, and know that more data is coming. (Normally the TCP code sends a non full-sized segment, only if it empties the TCP buffer). **Default 0.**
TCP_STDURG	int	Set this option value to a zero value if the peer is a Berkeley system since Berkeley systems set the urgent data pointer to point to last byte of urgent data+1. **Default 1.**
TM_TCP_2MSLTIME	int	Sets the maximum amount of time TCP will wait in the TIME WAIT state, once it has initiated a close of the connection. **Default 60 seconds.**
TM_TCP_DELAY_ACK	int	Sets the TCP delay ack time in milliseconds. **Default 200 milliseconds.**
TM_TCP_FINWT2TIME	int	Sets the maximum amount of time TCP will wait for the remote side to close, after it initiated a close. **Default 600 seconds.**
TM_TCP_KEEPALIVE_CNT	int	Sets the maximum numbers of keep alive probes without any response from the remote, before TCP gives up and aborts the connection. **Default 8.**
TM_TCP_KEEPALIVE_INTV	int	Sets the interval between Keep Alive probes in seconds. See TM_TCP_KEEPALIVE_CNT. This value can not be changed after a connection is established, and cannot be bigger than 120 seconds. **Default 75 seconds.**
TM_TCP_MAX_REXMIT	int	Sets the maximum number of retransmissions without any response from the remote, before TCP gives up and aborts the connection. **Default 12.**
TM_TCP_PACKET	int	Set this option value to a non-zero value to make TCP behave like a message-oriented protocol (i.e. respect packet boundaries) at the application level in

		both send and receive directions of data transfer. Note that for the receive direction to respect packet boundaries, the TCP peer which is sending must also implement similar functionality in its send direction. This is useful as a reliable alternative to UDP. Note that preserving packet boundaries with TCP will not work correctly if you use out-of-band data. TM_USE_TCP_PACKET must be defined in < trsystem.h> to use the TM_TCP_PACKET option. **Default 0.**
TM_TCP_PEND_ACCEPT_RECV_WND	unsigned long	Specify the size (in bytes) of the listening socket's receive window. This size will override the default size or the size specified by setsockopt() with the SO_RCVBUF flag. Once accept() is called on the listening socket, the window size will return to the size specified by SO_RCVBUF (or the default). Note: This size may not be larger than the default window size to avoid shrinking of the receive window.
TM_TCP_PROBE_MAX	unsigned long	Sets the maximum window probe timeout interval in milliseconds. The network computed window probe timeout is bound by TM_TCP_PROBE_MIN and TM_TCP_PROBE_MAX. **Default 60,000 milliseconds.**
TM_TCP_PROBE_MIN	unsigned long	Sets the minimum window probe timeout interval in milliseconds. The network computed window probe timeout is bound by TM_TCP_PROBE_MIN and TM_TCP_PROBE_MAX. **Default 500 milliseconds.**
TM_TCP_PURE_ACK_SEGS	int	Option only available if TM_USE_TCP_PURE_ACK is deined. Sets the number of outstanding un-ACKed segments, before a pure ACK is sent (even if the recv window has not changed.) Default value is zero, in which case the stack will behave as if TM_USE_TCP_PURE_ACK had not been defined, and will only ACK every other segment provided that it is combined with a window update, or will ACK when the delay ACK timer expires regardless of the window update. **Default 0.**

TM_TCP_REXMIT_CONTROL	int	Dynamically modify the behavior of the TCP retransmission timer for the specified socket. Valid values are 1 (Pause), 2 (Resume), and 3 (Reset). <u>TM_USE_TCP_REXMIT_CONTROL</u> must be defined in <trsystem.h> to make this option available.
TM_TCP_RTO_DEF	unsigned long	Sets the TCP default retransmission timeout value in milliseconds, used when no network round trip time has been computed yet. **Default 3,000 milliseconds.**
TM_TCP_RTO_MAX	unsigned long	Sets the maximum retransmission timeout in milliseconds. The network computed retransmission timeout is bound by TM_TCP_RTO_MIN and TM_RTO_MAX. **Default 64,000 milliseconds.**
TM_TCP_RTO_MIN	unsigned long	Sets the minimum retransmission timeout in milliseconds. The network computed retransmission timeout is bound by TM_TCP_RTO_MIN and TM_TCP_RTO_MAX. **Default 100 milliseconds.**
TM_TCP_SEL_ACK	int	Set this option value to a non-zero value to enable sending the TCP selective Acknowledgment option. **Note**: This option can only be changed prior to establishing a TCP connection. **Default 1.**
TM_TCP_SLOW_START	int	Set this option value to zero, to disable the TCP slow start algorithm. **Default 1.**
TM_TCP_SSL_CLIENT	int	Set this option to enable SSL client negotiation on this socket, optionLength must be sizeof(int), any non-zero value will enable SSL client.
TM_TCP_SSL_SERVER	int	Set this option to enable SSL server negotiation on this socket, optionLength must be sizeof(int), any non-zero value will enable SSL server. Note that, if you set this option for a listening socket, all accepted sockets inherit this option value, you don't have to set this option again on an accepted socket.
TM_TCP_SSLSESSION	int	Set the SSL session Id for this socket. The option length must be sizeof(int).

		Note that, if you set this option for a listening socket, all accepted sockets inherit this option value, you don't have to set this option again on an accepted socket
TM_TCP_SSL_SE ND_MIN_SIZE	int	Set the SSL send minimum size. If user's send data is less than this value, user data will be queued. Option length must be sizeof(int), and option value can not greater than 0xffff. Don't set this value too big. **Default value is defined as macro TM_SSL_SEND_DATA_MIN_SIZE (0)**
TM_TCP_SSL_SE ND_MAX_SIZE	int	Set the SSL record maximum size. Each record will at most have that much user data encapsulated. User data bigger than this size limit will be cut into two records, Option length must be sizeof(int), and option value can not greater than 0x4000 to enable reasonable encapsulate. Don't set this value too small. (<100 value will be rejected) **Default value is defined as macro TM_SSL_SEND_DATA_MAX_SIZE (8000).**
TM_TCP_TS	int	Set this option value to a non-zero value to enable sending the Time stamp option. **Note**: This option can only be changed prior to establishing a TCP connection. **Default 1.**
TM_TCP_WND_S CALE	int	Set this option value to a non-zero value to enable sending the TCP window scale option. **Note**: This option can only be changed prior to establishing a TCP connection. **Default 1.**
TM_TCP_STATE	int	Get the state of the TCP vector associated with the socket. **Note**: Read only value.
TM_TCP_USER_P ARAM	ttUserGeneric Union	Use this option to set/get user data for a specific TCP socket. To enable this feature, uncomment the TM_USE_USER_PARAM macro definition in your <trsystem.h>.
TM_TCP_CA_HY BLA	int	Set this option value to 1, to switch to the TCP Hybla Congestion Avoidance Algorithm. The TCP Hybla algorithm

		yields better performance for TCP connections with a long round trip time (such as on a high-latency terrestrial or satellite radio link). Set this option value to 0, to switch back to the TCP Reno Congestion Avoidance Algorithm. **Default 0.**
TM_TCP_PACING	int	Set this option value to 1, to turn on TCP Pacing. With TCP Pacing turned on, the stack will attempt to send TCP segments within the congestion window and peer receive window over the Round Trip Time, instead of sending them all at once. For better performance, this option should be turned on, if the TCP HYBLA algorithm is switched on. Set this option value to 0, to turn off TCP Pacing. **Default 0.**
TM_TCP_CA_WE STOOD	int	Set this option value to 1, to switch to the TCP Westwood+ Congestion Avoidance Algorithm. The TCP Westwood+ algorithm yields better performance on TCP connections over wireless lossy links. Set this option value to 0, to switch back to the TCP Reno Congestion Avoidance Algorithm. **Default 0.**

Table 6.4: TCP socket options

Check your progress

1. Explain IPV6_MULTICAST_IF option of IPV6 Socket.
2. What is the use of ICMP6_FILTER socket option?

6.8 SUMMARY

This unit details the socket concept and its programming tools. The two main socket functions are discussed with brief description of its option parameter which are used to set specific requirement. Listening socket retrieve some parameters from connected TCP to get the connection status knowledge. Idea of generic sockets is also included in the unit. Socket options description at IPv6, ICMP6 and TCP level are included with their syntactic representation.

6.9 TERMINAL QUESTIONS

1. Describe the use of socket options with example.
2. Write a code segment to retrieve output buffer size and set it to 1024 bytes.
3. How default of close function is modified and why it is needed?
4. Explain what and why of generic options.

UNIT-7 : ELEMENTARY UDP SOCKETS

Structure

7.0 INTRODUCTION

The **User Datagram Protocol** (**UDP**) is one of the core members of the internet protocol suite. UDP is the simpler of the two standard TCP/IP transport protocols where prior communications are not required to set up transmission channels or data paths. It is a process-to-process protocol that adds only port number for addressing, checksum for data integrity and length information of data from the upper layer. With UDP, computer applications can send messages, to other hosts on an internet protocol (IP) network. Although this is an "unreliable" protocol due to no handshaking but unlike TCP, it does not include mechanisms for retrying on transmission failures or data corruption and also it has restrictions on message length (a little under 65536 bytes). It is mostly needed for applications that use broadcasting or multicasting and may play performance-intensive roles such as multimedia. UDP is suitable for purposes where error checking and correction is either not necessary and it also avoids the overhead of such processing at the network interface level. Time-sensitive applications often use UDP because dropping packets is preferable to waiting for delayed packets, which may not be an option in a real-time system.

7.1 OBJECTIVE

To understand the usage, properties and implementation of UDP. After this unit you will come to know about:

- Tasks performed by echo server function and drawbacks of UDP
- What happens when a datagram is lost.

- Lack of support for flow control in UDP
- How to determine an outgoing interface with UDP

7.2 ECHO SERVER FUNCTION

UDP is a "connectionless" protocol which enables a program to use a single UDP socket to communicate with more than one host and port. UDP port numbers are entirely independent of TCP port numbers, though the IANA tries to register the same port number for both UDP and TCP when a given service is offered through both protocols. In fact, one of the most important practical differences between TCP and UDP is that there are no message boundaries in a TCP stream, whereas in UDP, every packet (datagram) is effectively a self-contained message. For applications where reliability is not a concern and where all messages are known to fit within the limited size of datagram, this can occasionally make UDP more convenient to use than TCP. UDP server socket is created in much the same way as a TCP server socket. The communication between client and server through UDP protocol is implemented through UDP echo server.

An echo server is an application which is used to test the connection between client and server. This server sends back whatever text the client sent. However, client server is an environment where server process the request sent by client.

In the UDP Echo server, we create a socket and bind to an advertised port number. Then an infinite loop is started to process the client requests for connections. Figure 7.1 shows the working of UDP Echo server.

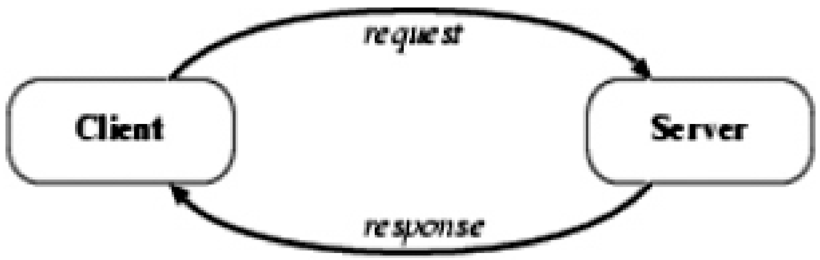


Figure 7.1: Working of UDP Echo server

The process receives data from the client using recvfrom() function and echoes the same data using the sendto() function. It handles multiple clients automatically as UDP is a datagram based protocol hence no exclusive connection is required to a client in this case.

Drawbacks of UDP:

TCP has emerged as the dominant protocol used for the bulk of internet connectivity owing to services for breaking large data sets into individual packets, checking for and resending lost packets and reassembling packets into the correct sequence. But these additional services come at a cost in terms of additional data overhead, and delays called latency.

7.3 LOST DATAGRAMS

UDP sends the packets over lower bandwidth overhead and latency. But packets can be lost or received out of order between sender and receiver. UDP is an ideal protocol for network applications in which perceived latency is critical such as gaming, voice and video communications, which can suffer some data loss without adversely affecting perceived quality. In some cases, forward error correction techniques are used to improve audio and video quality in spite of some loss.

UDP can also be used in applications that require lossless data transmission when the application is configured to manage the process of retransmitting lost packets and correctly arranging received packets. This approach can help to improve the data transfer rate of large files compared with TCP.

UDP client/server is not reliable. If a client datagram is lost, the client will block forever in its call to recvfrom in the function dg_cli. It may wait for a server reply that will never arrive. Similarly, if the client datagram arrives at the server but the server's reply is lost, the client will again block forever in its call to recvfrom. However, just placing a time out on the recvfrom cannot be the solution. For example, if we do time out, we cannot tell whether our datagram never made it to the server, or if the server's reply never made it back.

7.4 LACK OF FLOW CONTROL WITH UDP

We now examine the effect of UDP not having any flow control. First, we modify our dg_cli function to send a fixed number of datagrams. It no longer reads from standard input. Figure 1 shows the new version. This function writes 2,000 1,400-byte UDP datagrams to the server.

We next modify the server to receive datagrams and count the number received. This server no longer echoes datagrams back to the client. Figure 2 shows the new dg_echo function. When we terminate the server with our terminal interrupt key (SIGINT), it prints the number of received datagrams and terminates.

udpcliserv/dgcliloop1.c

```
#include    "unp.h"

#define NDG     2000        /* datagrams to send */

#define DGLEN   1400        /* length of each datagram */

void
```

```
dg_cli(FILE *fp, intsockfd, const SA *pservaddr, socklen_tservlen)
{
    int     i;
    char    sendline[DGLEN];
    for (i = 0; i < NDG; i++) {
        Sendto(sockfd, sendline, DGLEN, 0, pservaddr, servlen);
    }
}
```

udpcliserv/dgecholoop1.c

```
#include     "unp.h"
static void recvfrom_int(int);
static intcount;
void
dg_echo(intsockfd, SA *pcliaddr, socklen_tclilen)
{
    socklen_tlen;
    char    mesg[MAXLINE];
    Signal(SIGINT, recvfrom_int);
    for ( ; ; ) {
        len = clilen;
        Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);
        count++;
    }
}
static void
recvfrom_int(intsigno)
{
    printf("\nreceived %d datagrams\n", count);
    exit(0);
}
```

We now run the server on the host freebsd, a slow SPARCStation. We run the client on the RS/6000 system aix, connected directly with 100Mbps

Ethernet. Additionally, we run netstat -s on the server, both before and after, as the statistics that are output tell us how many datagrams were lost.

Check your progress

1. What is an echo server?
2. How does UDP improve the data transfer rate of large files compared with TCP?

7.5 DETERMINING OUTGOING INTERFACE WITH UDP

A connected UDP socket can also be used to determine the outgoing interface that will be used to a particular destination. This is because of a side effect of the connect function when applied to a UDP socket. The kernel chooses the local IP address (assuming the process has not already called bind to explicitly assign this). This local IP address is chosen by searching the routing table for the destination IP address, and then using the primary IP address for the resulting interface.

udpcliserv/udpcli09.c

```
#include     "unp.h"
int
main(intargc, char **argv)
{
   intsockfd;
   socklen_tlen;
   structsockaddr_incliaddr, servaddr;
   if (argc != 2)
      err_quit("usage: udpcli<IPaddress>");
   sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
   bzero(&servaddr, sizeof(servaddr));
   servaddr.sin_family = AF_INET;
   servaddr.sin_port = htons(SERV_PORT);
   Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
   connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
   len = sizeof(cliaddr);
   Getsockname(sockfd, (SA *) &cliaddr, &len);
printf("local address %s\n", Sock_ntop((SA *) &cliaddr, len));
exit(0);
}
```

If we run the program on the multihomed host freebsd, we have the following output:

```
freebsd % udpcli09 206.168.112.96

local address 12.106.32.254:52329

freebsd % udpcli09 192.168.42.2

local address 192.168.42.1:52330

freebsd % udpcli09 127.0.0.1

local address 127.0.0.1:52331
```

The first time we run the program, the command-line argument is an IP address that follows the default route. The kernel assigns the local IP address to the primary address of the interface to which the default route points. The second time, the argument is the IP address of a system connected to a second Ethernet interface, so the kernel assigns the local IP address to the primary address of this second interface. Calling connect on a UDP socket does not send anything to that host; it is entirely a local operation that saves the peer's IP address and port. We also see that calling connect on an unbound UDP socket also assigns an ephemeral port to the socket.

7.6 SOLVED EXAMPLES

Ques: What is the largest length that we can pass to *sendto* for a UDP/IPv4 socket, that is, what is the largest amount of data that can fit into a UDP/IPv4 datagram?

Solution:

The largest IPv4 datagram is 65,535 bytes, limited by the 16-bit total length field. The IP header requires 20 bytes and the UDP header requires 8 bytes, leaving a maximum of 65,507 bytes for user data. With IPv6 without jumbogram support, the size of the IPv6 header is 40 bytes, leaving a maximum of 65,487 bytes for user data. The new version of dg_cli has been used. If you forget to set the send buffer size, Berkeley-derived kernels return an error of EMSGSIZE from sendto, since the size of the socket send buffer is normally less than required for a maximum-sized UDP datagram. But if we set the client's socket buffer sizes and run the client program, nothing is returned by the server. We can verify that the client's datagram is sent to the server by running tcpdump, but if we put a printf in the server, its call to recvfrom does not return the datagram. The problem is that the server's UDP socket receive buffer is smaller than the datagram we are sending, so the datagram is discarded and not delivered to the socket. On a FreeBSD system, we can verify this by running netstat -s and looking at the "dropped due to full socket buffers" counter before and after our big datagram is received. The final solution is to modify the server, setting its socket send and receive buffer sizes.

7.7 SUMMARY

UDP is an unreliable transport layer protocol. It serves processes where error checking and correction is not necessary and processes that are time sensitive, that is, real time system. Being connectionless, it enables to use single socket to communicate with more than one host and port. Every packet is a self-contained message in UDP. An echo server is an application used to test the connection between client and server. If a client datagram is lost it is blocked forever. UDP does not support flow control, but is can be used for determining outgoing interface. This protocol is ideal for network applications like gaming, voice and video communication that can suffer some data loss without adversely affecting the quality.

7.8 TERMINAL QUESTIONS

1. Explain with examples the drawbacks of UDP.
2. "If a client datagram is lost, the client will block forever in its call to recvfrom in the function dg_cli". Explain.
3. State some real life examples of where UDP is used.
4. Write a program to implement Echo server function.

UNIT-8 : NAME AND ADDRESS CONVERSION

Structure

8.0 INTRODUCTION

There is some IP address attached with corresponding domain name server (DNS). DNS lookup, NSLOOKUP or IP lookup are the process to find the IP address by searching the DNS until a match found. The Domain Name System also specifies the technical functionality of the database service that is at its core. A DNS name server is a server that stores the DNS records for a domain; a DNS name server responds with answers to queries against its database. In a nutshell, you tell it what the human readable address is for a site and it will give you the IP address. There are some special IP addresses such as 127.0.0.1 which is default IP address of every computer. No matter which computer you use, it will always have an IP address of 127.0.0.1 and a name 'localhost'. In addition, a computer can have more than one IP address. In order to connect to other computers, it will have an IP address that is known to other computers.

8.1 OBJECTIVE

At the end of this unit, we will able to know the working of DNS.

- The importance and working of gethost by name function is explained.
- The different resolver options are discussed.

- Functions of IPV6 and its support is mentioned.
- Use of Uname function and other important networking information is explained.

8.2 DNS

DNS is hierarchical naming convention which contains information about services or any other resource connected to the network. It defines the DNS protocol, a detailed specification of the data structures and data communication exchanges. The Internet maintains two principal namespaces, the domain name hierarchy[and the Internet Protocol address spaces.

Most importantly, it translates more readily used domain names to the numerical IP addresses needed for the purpose of locating and identifying that resource. It provides worldwide directory service created in 1983 by Paul Mockapetris. The Domain Name System delegates the responsibility of assigning domain names and mapping those names to Internet resources by designating authoritative name servers for each domain.

There is often confusion about a host name and a domain name. A domain name is the name that is purchased from a registrar. It will be something like hc idata.com or hc idata.co.uk. Note that there is no "www" at the beginning of a domain name. A domain name can be subdivided into sub-domains - for example www.hcidata.com. Once you own a domain, there is no reasonable limit to the number of the sub-domains you can create. In fact many sub-domains can be allocated to the same host machine. Any requests for a sub-domain (e.g. www.hcidata.com) are converted to an IP address by DNS and the IP address is used to route the request through the network until it reaches the host machine.

In the early years of the internet, each sub-domain would have a unique IP address so it was common for a host machine to have only one sub domain name. Network administrators may delegate authority over sub-domains of their allocated name space to other name servers. This mechanism provides distributed and fault tolerant service and was designed to avoid a single large central database.

Nowadays, the common practice is to have many sub-domains with the same IP address. It is also common for the domain name to get converted into the IP address of the host machine that runs the www sub domain. For example, a host machine that converts host names to IP address using DNS may be called dns.hcidata.com and a host machine that is a web server may be called www.hcidata.com.

IP address to Country

IP addresses are allocated by regional organizations. Therefore, it is relatively easy to work out the country in which an IP is likely to reside. When an IP is allocated to a company they are expected to be used in the country in which the organization resides. But, there is nothing to stop a company allocating an IP to a machine in another country. A company is allocated a range of IP addresses X.Y.Z.0 to X.Y.Z.255 for use in England. This company has a private network with a branch office in New York. So, it uses most of the IP address in England but uses some of them in the United States. So, we cannot guarantee that the country is 100% correct when converting an IP address, but we would expect it to be correct at least 90% of the time.

8.3 GETHOST BY NAME FUNCTION

The gethostbyname function retrieves host information corresponding to a host name. This function has been deprecated by the introduction of the getaddrinfo function. Developers creating Windows Sockets 2 applications are advised to use the getaddrinfo function instead of gethostbyname.

struct hostent* FAR gethostbyname(_In_ const char *name);

Return value

If no error occurs, gethostbyname returns a pointer to the hostent structure described above. Otherwise, it returns a null pointer and a specific error number. The gethostbyname function does not check the size of the *name* parameter before passing the buffer which may result heap corruption.

8.4 RESOLVER OPTION

The OptionsResolver component helps you configure objects with option arrays. It supports default values, option constraints and lazy options. The *resolver* is a set of routines in the C library that provide access to the Internet Domain Name System (DNS). The OptionsResolver component helps you configure objects with option arrays. It supports default values, option constraints and lazy options. The resolver configuration file is designed to be human readable format which contains a list of keywords with values that provide various type of resolver option.

The different configuration options are:

nameserver- Name server IP address

Internet address of a name server that the resolver should query

Resolver query IP address from the name server. If there are multiple servers, the resolver queries them in order. If no nameserver entries are present, the default is the name server on the local machine.

domain -Local domain name

Short names are used relative to the local domain. If no domain entry is present, the domain is determined form the local hostname returned by gethostname(). The domain part is taken to be everything after the first '.'.

The root domain is assumed if the hostname does not contain a domain part.

search -Search list for host-name lookup

The search list is normally determined from the local domain name. However, by default, it contains only the local domain name. This may be changed by listing the desired domain search path following the search keyword with space and tabs separating the names. This process may be slow and may generate network traffic if the servers for the listed domains are not local. Queries will time out if no server is available for one of the domains.The search list is currently limited to six domains with a total of 265 characters.

shortlist

Sorted address are returned by gethostbyname() through this option. A shortlist is specified by IP-address-netmask pairs. The IP address and optional network pairs are separated by slashes.

options

It allows certain internal resolver variables to be modified. The syntax is options option where option is as follows:

debug

It sets RES_DEBUG.

ndots: n

It sets a threshold for the number of dots which must appear in a name given to res_query before an initial absolute query will be made. The default value for n is 1. It implies that if there are any dots in a name, the name will be tried first as an absolute name before any search list elements are appended to it.

timeout: n

It sets the amount of time the resolver will wait for a response from a remote name server. It is measured in seconds.

attempts: n

It sets the number of times the resolver will send a query to its name server before giving up.

rotate

It makes round robin selection of name servers by spreading the query load among all listed servers.

no-check-names

It disables the modern BIND checking of incoming hostnames and mail names for invalid characters such as underscore, non-ASCII or control characters.

int6

This has the effect of trying a AAAA query before an A query inside gethostname() function. It maps IPv4 responses in IPv6 "tunneled form".

ip6-bytestring

It causes reverse IPv6 lookups to be made using the bit-label format described in RFC 2673.

ip6-dotint/no-ip6-dotint

When this option is clear, reverse IPv6 lookups are made in the ip6.int zone. When this option is set, reverse IPv6 lookups are made in the ip6.arpa zone. reverse IPv6 lookups are made in the *ip6.arpa* zone by default. This option is set by default.

ends0

It enables support for the DNS extension described in RFC 2671.

single-request

Sometime DNS server cannot handle these queries properly and make a requests time out. This option disables the behaviour and makes glibc perform the IPv6 and IPv4 requests sequentially.

single-request-reopen

The resolver uses the same socket for A and AAAA requests. Some hardware does mistake to send back only one reply. The client sits and waits for second reply. By turning this option ON, it closes the socket and opens a new one before sending the second request.

Check your progress

1. Explain how DNS can be used in recursive way?
2. What are the return values returned by gethostbyname function?

8.5 FUNCTION AND IPV6 SUPPORT

Internet Protocol Version 6 (IPv6) is a network layer protocol that enables data communications over a packet switched network. Packet switching involves the sending and receiving of data in packets between two nodes in a network. The working standard for the IPv6 protocol was published by the Internet Engineering Task Force (IETF) in 1998.Japan and Korea were acknowledged as having the first public deployments of IPv6

IPv6 and IPv4 share a similar architecture. The majority of transport layer protocols that function with IPv4 will also function with the IPv6 protocol. Most application layer protocols are expected to be interoperable with IPv6 as well. A main advantage of IPv6 is increased address space. The 128-bit length of IPv6 addresses is a significant gain over the 32-bit length of IPv4 addresses, allowing for an almost limitless number of unique IP addresses. The size of the IPv6 address space makes it less vulnerable to malicious activities such as IP scanning. IPv6 packets can support a larger payload than IPv4 packets resulting in increased throughput and transport efficiency. Notable exception of File Transfer Protocol (FTP).

IPv6 functions

IBM is implementing IPv6 on i5/OS® over several software releases. IPv6 functions are transparent to existing TCP/IP applications and coexist with IPv4 functions.

These are the main i5/OS features that are affected by IPv6:

If you configure IPv6, you are sending IPv6 packets over an IPv6 network. Creating an IPv6 local area network for a scenario that describes a situation in which you configure IPv6 on your network.

The Start and Stop menu items on the TCP/IP Configuration folder are removed. IPv6 can be started and stopped in the same way as IPv4, with STRTCP and ENDTCP commands. IPv6 cannot be started or stopped independent of IPv4.

The Configure IPv6 wizard is removed from iSeries Navigator. The line configuration options in the wizard are replaced by actions on individual lines in the **Lines** folder. Similarly, you can use a new wizard to create IPv6 interfaces.

8.6 UNAME FUNCTION

This function is used to get name and information about current kernel. This is a system call, and the operating system presumably knows its name, release and version. It also knows what hardware it runs on. So, four of the fields of the struct are meaningful. On the other hand, the field *nodename* is meaningless: it gives the name of the present machine in some undefined network, but typically machines are in more than one network and have several names. Moreover, the kernel has no way of knowing about such things, so it has to be told what to answer here. The same holds for the additional *domainname* field.

It returns system information in the structure pointed to by *buf*.

#include <sys/utsname.h>

int uname(struct utsname **buf*);

The *utsname* struct is defined in <*sys/utsname.h*>:

```
struct utsname {
char sysname[];    /* Operating system name*/
char nodename[];   /* Name within "some implementation-defined
                                    network" */
  char release[];    /* Operating system release*/
  char version[];    /* Operating system version */
  char machine[];    /* Hardware identifier */
#ifdef _GNU_SOURCE
  char domainname[]; /* NIS or YP domain name */
  #endif
  };
```

On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

The length of the fields in the struct varies. Some operating systems or libraries use a hardcoded 9 or 33 or 65 or 257. Other systems use SYS_NMLN or _SYS_NMLN or UTSLEN or _UTSNAME_LENGTH. Clearly, it is a bad idea to use any of these constants; just use sizeof(...). Often 257 is chosen in order to have room for an internet hostname.

8.7 Other Networking Information

When looking at networking basics, understanding the way a network operates is the first step to understanding routing and switching. The network operates by connecting computers and peripherals using two pieces of equipment; switches and routers. Switches and routers, essential networking basics, enable the devices that are connected to your network to communicate

Networking Basics: Switches

- Switches are used to connect multiple devices on the same network within a building or campus. For example, a switch can connect your computers, printers and servers, creating a network of shared resources. The switch, one aspect of your networking basics, would serve as a controller, allowing the various devices to share information and talk to each other. Through information sharing and resource allocation, switches save you money and increase productivity.

There are two basic types of switches to choose from as part of your networking basics: managed and unmanaged.

- ❖ An unmanaged switch works out of the box and does not allow you to make changes. Home-networking equipment typically offers unmanaged switches.
- ❖ A managed switch allows you access to program it. This provides greater flexibility to your networking basics because the switch can be monitored and adjusted locally or remotely to give you control over network traffic, and who has access to your network.

- **Routers**, the second valuable component of your networking basics, are used to tie multiple networks together. For example, you would use a router to connect your networked computers to the Internet and thereby share an Internet connection among many users. The router will act as a dispatcher, choosing the best route for your information to travel so that you receive it quickly. Routers analyse the data being sent over a network, change how it is packaged, and send it to another network, or over a different type of network. They connect your business to the outside world, protect your information from security threats, and can even decide which computers get priority over others. Depending on your business and your networking plans, you can choose from routers that include different capabilities. These can include networking basics such as:
    - ❖ **Firewall:** Specialized software that examines incoming data and protects your business network against attacks.
    - ❖ **Virtual Private Network (VPN):** A way to allow remote employees to safely access your network remotely.
    - ❖ **IP Phone network:** Combine your company's computer and telephone network, using voice and conferencing technology, to simplify and unify your communications.

Check your progress

1. How does IPV6 improve throughput and transport efficiency of a network?

2. What is the use of uname function?

8.8 SOLVED EXAMPLE

Q. Modify following program to call getnameinfo instead of sock_ntop. What flags should you pass to getnameinfo?

names/daytimetcpcli1.c

```
#include "unp.h"
int
main (int argc, char **argv)
{
int sockfd, n;
char recvline [MAXLINE + 1];
struct sockaddr_in servaddr;
struct in_addr **pptr;
struct in_addr *inetaddrp [2];
struct in_addr inetaddr;
struct hostent *hp;
struct servent *sp;
if (argc ! = 3)
err_quit ("usage: daytimetcpclil ");
if ( (hp = gethostbyname (argv [1]) ) == NULL) {
if (inet_aton (argv [1], &inetaddr) == 0) {
err_quit ("hostname error for %s: %s", argv [1],
hstrerror (h_errno) );
} else {
inetaddrp [0] = &inetaddr;
inetaddrp [1] = NULL;
pptr = inetaddrp;
}
} else {
pptr = (struct in_addr **) hp->h_addr_list;
}
if ( (sp = getservbyname (argv [2], "tcp") ) == NULL)
err_quit ("getservbyname error for %s", argv [2] );
for ( ; *pptr != NULL; pptr++) {
sockfd = Socket (AF_INET, SOCK_STREAM, 0) ;
bzero (&servaddr, sizeof (servaddr) ) ;
servaddr.sin_family = AF_INET;
servaddr.sin_port = sp->s_port;
memcpy (&servaddr.sin_addr, *pptr, sizeof (struct in_addr) ) ;
printf ("trying %s\n", Sock_ntop ( (SA *) &servaddr, sizeof (servaddr) ) ) ;
if (connect (sockfd, (SA *) &servaddr, sizeof (servaddr) ) == 0)
break; /* success */
err_ret ("connect error");
close (sockfd) ;
}
if (*pptr == NULL)
err_quit ("unable to connect");
while ( (n = Read (sockfd, recvline, MAXLINE) ) > 0) {
recvline [n] = 0; /* null terminate */
Fputs (recvline, stdout);
}
exit (0);
}
```

Solution:

Following modifications are made in the code given above.

1. We first allocate arrays to hold the hostname and service name as follows:
   char host[NI_MAXHOST], serv[NI_MAXSERV];
2. After accept returns, we call getnameinfo instead of sock_ntop as follows:

   if (getnameinfo(cliaddr, len, host, NI_MAXHOST, serv, NI_MAXSERV, NI_NUMERICHOST | NI_NUMERICSERV) == 0)

   printf("connection from %s.%s\n", host, serv);

3. Since this is a server, we specify the NI_NUMERICHOST and NI_NUMERICSERV flags to avoid a DNS query and a lookup of /etc/services.

8.9 SUMMARY

The Domain Name System specifies the technical functionality of the database service that is at its core. A DNS name server is a server that stores the DNS records for a domain; a DNS name server responds with answers to queries against its database. The Internet maintains two principal namespaces, the domain name hierarchy[and the Internet Protocol address spaces. The *resolver* is a set of routines in the C library that provide access to the Internet Domain Name System (DNS). Internet Protocol Version 6 (IPv6) is a network layer protocol that enables data communications over a packet switched network. The 128-bit length of IPv6 addresses is a significant gain over the 32-bit length of IPv4 addresses, allowing for an almost limitless number of unique IP addresses. The Uname function is used to get name and information about current kernel. It also knows what hardware it runs on. Switches and routers enable the devices that are connected to your network to communicate. Switches are used to connect multiple devices on the same network providing information sharing and resource allocation that in turn saves your money and increases productivity. Routers are used to tie multiple networks together, choosing the best route for your information, connect your business to the outside world, protect your information from security threats, and can even decide which computers get priority over others.

8.10 TERMINAL QUESTIONS

1. State the similarities and differences between IPv4 and IPv6.
2. Write a program for choosing the best route for some hypothetical network.
3. How load balancing is achieved using DNS?

Uttar Pradesh Rajarshi Tandon
Open University

Bachelor of Computer Application

BCA-E7
Network Programming

Block

3

DAEMON PROCESSES, ADVANCE I/O FUNCTIONS AND UNIX DOMAIN PROTOCOLS

© **UPRTOU, Prayagraj. 2019**
ISBN : 978-93-83328-11-6

Printed and Published by Dr. A run Kumar G upta R egistrar, Uttar P radesh Rajarshi Tandon Open University, 2019.

Printed By : Chandrakala U niversal P vt. Ltd. 42/7 Jawahar L al N eharu R oad, Prayagraj.

BLOCK INTRODUCTION

The obj ective of t his course i s t o i ntroduce t he basic c oncept about t he network pr ogramming as w ell as provides a mix o f p ractical ex perience and a d epth of unde rstanding. The network pr ogramming course address today's m ost crucial s tandards, implementations a nd t echniques. The ai m is to provide an extensive variety of topics on this subject with appropriate examples. The course is organized into following blocks:

Block 3 describes the daemon processes, advance I/O functions and UNIX domain protocols.

UNIT-9 : DAEMON PROCESSES

Structure

9.1 INTRODUCTION

In this unit, we will learn about daemons and characteristics of daemon processes. Further on we will look on how to log messages using *syslog* facility. Then daemon providing internet services is discussed. Then we will also have a look on *daemon_init* function, *inetd* daemon and *daemon_inetd* functions.

9.2 OBJECTIVES

At the end of this unit we will have knowledge about: -

- Daemons and their characteristics.
- Ways to start a daemon.
- *syslogd* Daemon and *syslog* function.
- *daemon_init* function, *inetd* daemon and *daemon_inetd* functions.

9.3 DAEMON

A daemon is a process that runs in the background as a background process instead of being under the direct control of an interactive user. In other words, it is not associated with controlling terminal or login shell. Unix systems typically have many processes that are daemons, running in the background, performing different administrative tasks.

For example: -

- A line printer has a daemon process that is waiting for a request to print a file on a line printer.
- A remote login program has a daemon process that waits for a request to come across the network for someone to login.

Generally, in UNIX system, the name of the daemon process end with the letter *d*. As for example, *syslogd* daemon, *inetd* daemon, *sshd* daemon, etc.

System daemons have the following characteristics: -

- Started once when the system is initialized.
- Runs until the system is shut down.
- During the service time, spends most of their time waiting for some event to occur.
- Frequently spawn other processes to handle service requests.

Ways to start a daemon: -

a. Many daemons are started by the system initialization scripts. These scripts are mainly in the directory /etc. or in a directory whose name begins with /etc/rc.

b. Many network servers are started by *inetd* superserver.

c. The execution of programs on a regular basis is performed by the *cron* daemon, and programs that it invokes run as daemons.

d. The execution of a program at one time in the future is specified by the at command. The cron daemon normally initiates these programs when their time arrives, so these programs run as daemons.

e. Daemons can be started from user terminals, either in the foreground or in the background.

Check Your Progress:

1. Can you define the daemon process?

2. Give the two examples for daemon process.

9.4 *syslogd* DAEMON

Many versions of UNIX provide a general-purpose logging facility called *syslog*. Individual programs that need to have information logged send the information to *syslog*. In order to handle these logs status *syslogd* daemon comes into play. Purpose of *syslogd* daemon is to log system messages. It reads the log message and does what the configuration file (normally */etc. / syslog.conf*) s pecifies t o do w ith that message. If t he daemon receives the SIGHUP signal, it rereads its configuration file.

Berkeley-derived imp lementations o f s yslogd perform t he f ollowing actions on startup:

1. The c onfiguration f ile, nor mally / etc/syslog.conf, i s r ead, specifying w hat t o do with e ach t ype o f l og message t hat t he daemon can r eceive. T hese m essages c an b e ap pended t o a f ile written to a s pecific user, or forwarded to the syslogd daemon on another host.

2. A U nix dom ain s ocket i s c reated a nd bound to t he pa thname /var/run/log.

3. A UDP s ocket i s created a nd bound t o por t 514 (the s yslog service).

4. The pa thname / dev/klog i s ope ned. A ny error m essages f rom within the kernel appear as input on this device.

9.5 *syslog* FUNCTION

Since d aemon d oesn't h ave a controlling te rminal, it n eeds s ome way t o out put m essages when s omething ha ppens l ike nor mal informational messages or emergency messages that need to be handled by an administrator. So, there comes the role of *syslog* function.

Structure of syslog function:

```
#include <syslog.h>

void syslog (int priority, const char *message, …);
```

Here, priority is combination of *level* (0 to 7) and *facility* (to identify the type of process sending the message). Log messages have a level between 0 and 7, which shown in table 9.1. These are ordered values. If no level is specified by the sender, LOG_NOTICE is the default**.**

Level	**Value**	**Description**
LOG_EMERG	0	System is unusable (highest priority)
LOG_ALERT	1	Action must be taken immediately
LOG_CRIT	2	Critical conditions
LOG_ERR	3	Error conditions
LOG_WARNING	4	Warning conditions
LOG_NOTICE	5	Normal but significant condition(default)
LOG_INFO	6	Informational
LOG_DEBUG	7	Debug-level messages (lowest priority)

Table 9.1: Level of log messages

Log m essages al so contain a f acility t o i dentify the t ype of pr ocesses sending t he m essages. We s how t he di fferent values i n table 9 .2. I f no facility is specified, LOG_USER is the default.

Facility	Description
LOG_AUTH	Security/authorization messages
LOG_AUTHPRIV	Security/authorization messages(private)
LOG_CRON	Cron daemon
LOG_DAEMON	System daemons
LOG_FTP	FTP daemon
LOG_KERN	Kernel messages
LOG_LOCAL0	Local use
LOG_LOCAL1	Local use
LOG_LOCAL2	Local use
LOG_LOCAL3	Local use
LOG_LOCAL4	Local use
LOG_LOCAL5	Local use
LOG_LOCAL6	Local use
LOG_LOCAL7	Local use
LOG_LPR	Line printer system
LOG_MAIL	Mail system
LOG_NEWS	Network news system
LOG_STSLOG	Messages g enerated internally b y syslogd
LOG_USER	Random user-level messages(default)
LOG_UUCP	UUCP system

Table 9.2: Facility of log messages

The p urpose o f f acility and l evel i s t o al low al l m essages f rom a given facility to b e h andled th e s ame in th e /e tc/syslog.conf file o r to a llow a ll messages of a given level to be handled the same.

Check Your Progress

1. *Can you define the steps performed by syslogd Daemon?*

2. *Can you define the different levels of log message?*

9.6 *DAEMON_INIT* FUNCTION

daemon_init function is used to demonize a process i.e. to start an arbitrary program and run it as a daemon. This function should be suitable for use on all variants of UNIX but some offer C library function called daemon that provides similar feature.

The program below shows a function named daemon_init that can call to daemonize the process.

```
#include "unp.h"
#include     <syslog.h>
#define   MAXFD   64
extern int daemon_proc;
int
daemon_init(const char *pname, int facility)
{
    int  i;
    pid_t  pid;
    if ((pid=Fork ()) <0)
      return (-1);
   else if (pid)
         _exit (0); /*
  /* child 1 continues…*/
 if(setsid()   < 0)
    return (-1);
 signal (SIGHUP,SIG_IGN);
if ((pid =Fork ()) < 0)
   return (-1);
else if (pid)
```

```
    _exit (0);
/* child 2 continues…*/
daemon_proc = 1;
chdir("/")
/*close off file descriptors */
for (i=0; i<MAXFD; i++)
  close(i);
/* redirect stdin, stdout and stderr   to /dev/null */
open ("/dev/null", O_RDONLY);
open ("/dev/null", O_RDWR);
open ("/dev/null", O_RDWR);
openlog (pname,  LOG_PID, facility);
return (0);
}
```

In the program, the daemon_init function first call fork and then the parent terminates, and child continues. If the process starts as a shell command in the foreground, when the parent terminates, the shell thinks the command is done. This automatically runs the child process in the background. Also, the child inherits the process group ID from the parent but gets its own process ID. This guarantees that the child is not a process group leader, which is required for the next call to setsid.

The setsid is a POSIX function that creates a new session. The process becomes the session leader of the new session. The process becomes group leader of a new process group and has no controlling terminal. Ignore SIGHUP and call fork again. When this function returns, the parent is the first child and it terminates, leaving the second child running. The purpose of this second fork is to guarantee that the daemon cannot automatically acquire a controlling terminal should it open a terminal device in the future. The calling fork in a second time, guarantee that the second child is no longer a session leader, so it cannot acquire a controlling terminal.

Then set flag for error functions. Set the global daemon_proc to nonzero. Then change the working directory to the root directory, although some daemons might have a reason to change to some other directory. After that close, any open descriptors that are inherited from the process that executed the daemon. After that redirect stdin, stdout, and stderr to /dev/null for standard input, standard output, and standard error. Then open log is called. The first argument is from the caller and is normally the name of the program (e.g., argv[0]). The process ID should be added to each log message. This facility is also specified by the caller.

9.7 *INETD* DAEMON

inetd r efers t o i nternet s ervice da emon. i netd daemon i s a superserver (service d ispatcher) d aemon o n m any UNIX systems th at provide i nternet s ervices. This d aemon i s u sed b y servers t hat u se ei ther TCP or UDP.

This *inetd* process establishes itself as a d aemon using the techniques that we de scribed w ith our d aemon_init f unction. It t hen r eads and pr ocesses its c onfiguration f ile, typically / etc/inetc.conf. This f ile s pecifies t he services that the s uper server is to handle, and what to do w hen a service request arrives. The table 9.3 shows the fields in inetd.conf file.

Field	Description
service-name	Must be in /etc/services
socket-type	Stream(TCP) or dgram (UDP)
Protocol	Must be in /etc/protocols either tcp or udp
wait-flag	Typically, nowait for TCP or wait for UDP
login-name	From /etc/passwd: typically root
server-program	Full pathname to exec
server-program-arguments	Arguments for exec

Table 9.3: Fields in inetd.conf file

When a TCP packet o r UDP packet ar rives w ith a p articular d estination port number, *inetd* launches the appropriate s erver pr ogram t o h andle t he connection.

The steps performed by inetd shown in figure 9.1. The steps are as follows: -

- On s tartup, i t r eads t he */etc/inetd.conf* and creates a s ocket o f appropriate type for all the services specified in the file.
- *bind* is c alled for the s ocket to specify port and IP a ddress for the server.
- *listen* is called for TCP (not needed for datagram sockets).
- After cr eation o f s ocket *select* is c alled to w ait f or a ny o f th e socket to become readable.
- When the *select* returns that a socket is readable, *accept* is called to accept the new connection (only for TCP connection).

- The *inetd* daemon *forks* and the child process handle the service request.

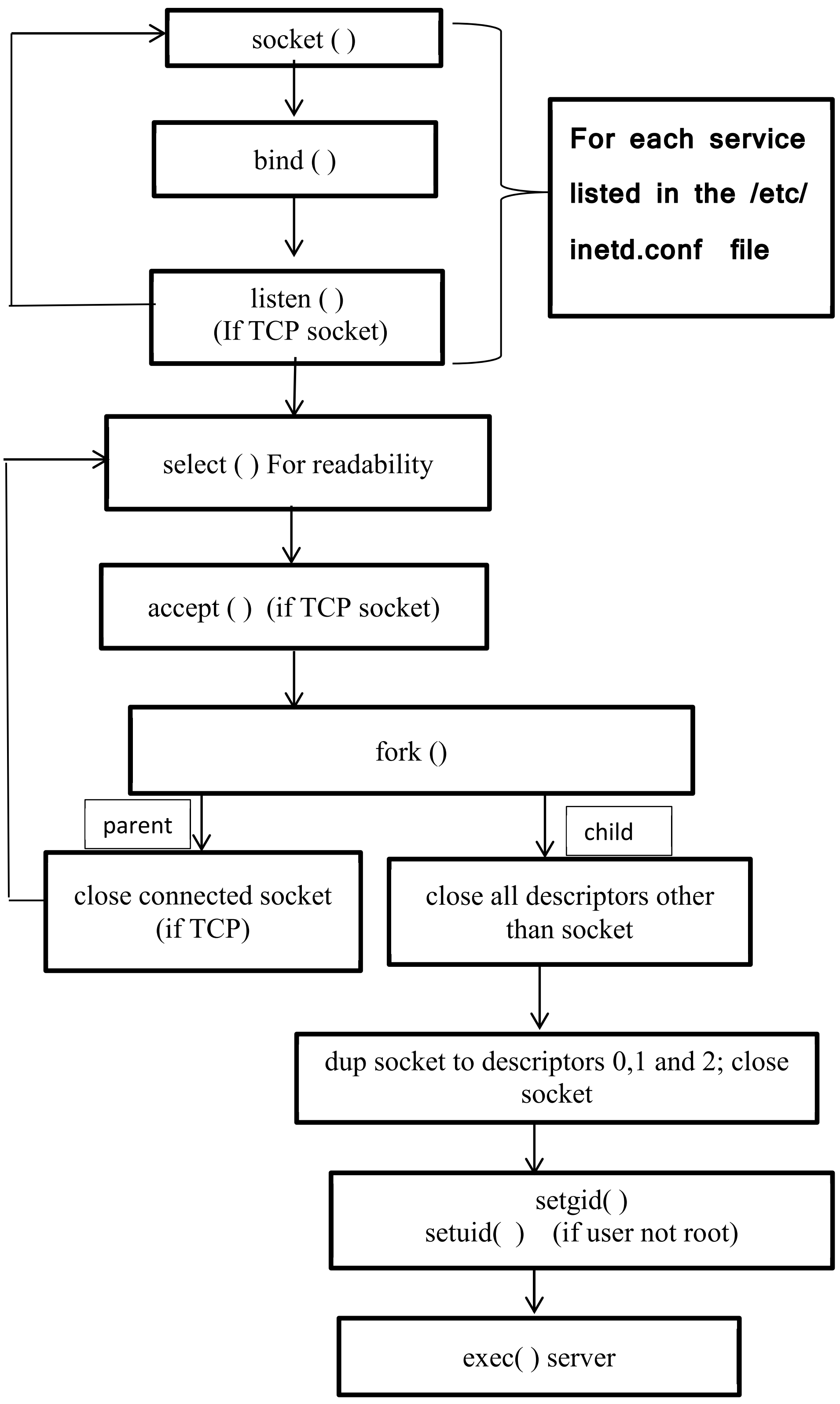

socket ( )
bind ( )
listen ( )
(If TCP socket)
For each service listed in the /etc/inetd.conf file
select ( ) For readability
accept ( ) (if TCP socket)
fork ()
parent
child
close connected socket (if TCP)
close all descriptors other than socket
dup socket to descriptors 0,1 and 2; close socket
setgid( )
setuid( ) (if user not root)
exec( ) server

Figure 9.1: Steps performed by inetd

9.8 *DAEMON_INETD* FUNCTION

It demonizes process run b y *inetd.* This function is tr ivial compared t han da emon_init be cause a ll of t he daemonization s teps a re performed by inetd when it starts.

The program below shows a function named daemon_inetd

```
#include "unp.h"

#include    <syslog.h>

extern int daemon_proc;

void

daemon_inetd(const char *pname,int facility)

{

   daemon_proc =1;

  openlog (pname, LOG_PID,facility)

}
```

The daemonization a re performed b y i netd w hen i t s tarts. T he daemon_proc f lag f or e rror f unctions a nd o pen log is c alled. T he f irst argument is from the caller and is normally the name of the program. The process ID s hould b e a dded t o ea ch l og m essage. T his facility is a lso specified by the caller.

9.9 SUMMARY

Daemons a re t he pr ocesses r unning i n ba ckground and a re independent of control from all terminals. All outputs from a daemon are normally s ent t o *syslog* daemon b y c alling t he *syslog* function. Start o f daemon r equires a few steps. *daemon_init* handles d etails o f s tarting a daemon. Many UNIX servers that provide internet services are started by the *inetd* daemon.

9.10 TERMINAL QUESTIONS

1. Define daemon and its characteristics.
2. Write ways to start a daemon.
3. Define *syslogd* daemon and *syslog* function.
4. Explain function of *inetd* daemon.
5. Explain steps of *inetd* daemon.
6. Explain *daemon_init* function with an example.

UNIT-10 : ADVANCE I/O FUNCTIONS

Structure

10.1 INTRODUCTION

In this unit, we will cover a variety of functions and techniques that is categorized as "Advance I/O". First, we will see three ways of setting a timeout on I/O operation involved in socket. Next comes three variations on the *read* and *write* functions. We will study about ancillary data. Then we will also have a look on how to determine the amount of data in the socket receive buffer and how to use the C standard I/O library with sockets.

10.2 OBJECTIVE

At the end of this unit we will get to know: -

- Three ways to place a timeout on I/O operation involving a socket.
- Format of variations of *read* and *write.*
- About ancillary data
- How to determine the amount of data in the socket receive buffer?
- How to use the C standard I/O library with sockets?

10.3 SOCKET TIMEOUTS

Sockets involve some I/O operation (like *read, write, etc.*). So, a timeout can be placed on these I/O operations.

Following are three ways to place the timeout: -

- Call *alarm*, which generates the SIGALRM signal when the specified time has expired.
- Block waiting for I/O in *select*, which has a time limit built in instead of blocking in a call to read or write.
- Use the newer *SO_RCVTIMEO* and *SO_SNDTIMEO* socket options.

10.4 *RECV* AND *SEND* FUNCTIONS

recv function is used to receive messages from a socket and may be used to receive data on a socket. It is normally used only on a connected socket.

send function is used to transmit a message to another socket. It is normally used only on a connected socket.

These two functions are similar to the standard *read* and *write* functions, but one additional argument is required. Header file for these operations is <sys/socket.h>. Format of *read* and *write* functions are given below:

#include <sys/socket.h>

ssize_t recv(int *sockfd*, void **buff*, size_t *nbytes*, int *flags*);

ssize_t send (int *sockfd*, void const **buff*, size_t *nbytes*, int *flags*);

Both return: number of bytes read or write if OK, -1 on error

Here, first three arguments are same as the first three arguments to *read* and *write. recv* read *nbytes* bytes from socket file descriptor *sockfd* into buffer *buff. send* function sends *nbytes* to the socket file descriptor *sockfd* from buffer *buff* starting at *buff.*

The *flags* argument is either 0 or is formed by logically OR'ing one or more of constant shown in table 10.1 below: -

Flags	**Description**	**recv**	**Send**
MSG_DONTROUTE	Bypass routing table lookup		*
MSG_DONTWAIT	Only this operation is nonblocking	*	*
MSG_OOB	Send or receive out-of-band data	*	*
MSG_PEEK	Peek at incoming message	*	
MSG_WAITALL	Wait for all the data	*	

Table 10.1: flags for I/O function

10.5 *READV* AND *WRITEV* FUNCTIONS

These two functions are similar to read and write but, *readv and writev* let us read into or write from one or more buffers with a single function call. These operations are called *scatter read* and *gather write.*

readv function *iovcnt* blocks from the file associated with the file descriptor *fields* into the multiple buffers described by *iov*.

writev function writes at most *iovcnt* blocks described by the *iovec* to the file associated with the file descriptor *filedes*.

Format of readv and writev functions are given below:

#include <sys/uio.h>

ssize_t readv(int *filedes*, const struct *iovec *iov*, int *iovcnt*);

ssize_t writev(int *filedes*, const struct *iovec *iov*, int *iovcnt*);

Both return: number of bytes read or write if OK, -1 on error

Here, iovec is a structure defined to denote buffer starting address and its size.

```
struct iovec{
        void *iov_base;         /*starting address of buffer*/
        size_t iov_len;/*size of buffer*/
};
```

Header file for these operations is <sys/uio.h>.

Check Your Progress

1. *Explain the recv( ) and send( ) function with syntax.*
2. *Write the difference between readv( ) and writev( ) function.*

10.6 *RECVMSG* AND *SENDMSG* FUNCTIONS

These two functions are the most general of all the I/O functions. We could replace all calls to *read, readv, recv, and recvfrom* with calls to *recvmsg*. Similarly, all calls to the various output functions could be replaced with calls to *sendmsg*. Header file to be included is <sys/socket.h>

Format of recvmsg and sendmsg functions are given below:

#include <sys/socket.h>

ssize_t recvmsg(int *sockfd*, struct msghdr **msg*, int *flags*);

ssize_t sendmsg(int *sockfd*, struct msghdr **msg*, int *flags*);

msghdr structure is defined as: -

```
struct msghdr{
        void    *msg_name;              /*protocol address*/
        socklen_t    msg_namelen; /*size of protocol address*/
        struct  iovec   *msg_iov;       /*scatter/gather
array*/
        int     msg_iovlen;             /*# e    lements in
msg_iov*/
        void    *msg_control;     /*ancillary d  ata  (cmsghdr
struct) */
        socklen_t    msg_controllen;    /*length o f a ncillary
data*/
        int     msg_flags;              /*flags r  eturned b  y
recvmsg ()*/
        };
```

Here, *msg_name* and *msg_namelen* members are used when the socket is not c onnected. *msg_iov* and *msg_iovlen* members s pecify t he array of input or out put buf fers (the a rray of *iovec* structures). *msg_control* and *msg_controllen* members s pecify t he l ocation a nd s ize of t he opt ional ancillary data. *msg_flags* member is used only by *recvmsg* while ignored by *sendmsg*.

Summary of the flags that are examined by the kernel for the input and output f unctions, a s w ell a s t he *msg_flags* that might be r eturned b y *recvmsg* is shown in table 10.2 below:

Flag	**Examined by: send *flags* sendto *flags* sendmsg *flags***	**Examined by: recv *flags* recvfrom *flags* recvmsg *flags***	**Returned by: recvmsg msg_flags**
MSG_DONTROUTE	*		
MSG_DONTWAIT	*	*	
MSG_PEEK		*	
MSG_WAITALL		*	
MSG_EOR	*		*
MSG_OOB	*	*	*
MSG_BCAST			*
MSG_MCAST			*
MSG_TRUNC			*
MSG_CTRUNC			*
MSG_NOTIFICATION			*

Table 10.2: Summary of input and output flags by various I/O functions

The first four flags are only examined and never returned; the next two are both examined and returned; and the last four are only returned.

10.7 ANCILLARY DATA

Control messages or control information is also called as ancillary data. Ancillary Data can be sent and received using the *msg_control* and *msg_controllen* members of the *msghdr* structure with the *sendmsg* and *recvmsg* functions. Summary of the various uses of ancillary data is shown in table 10.3 below.

Protocol	**cmag_level**	**cmag_type**	**Description**
IPv4	IPPROTO_IP	IP_RECVDSTADDR	Receive destination address with UDP datagram
		IP_RECVIF	Receives interface index with UDP datagram
IPv6	IPPROTO_IPv6	IPv6_DSTOPTS	Specify destination options
		IPv6_HOPLIMIT	Specify hop limit
		IPv6_HOPOPTS	Specify hop-by-hop options
		IPv6_NEXTHOP	Specify next-hop address
		IPv6_PKTINFO	Specify packet information
		IPv6_RTHDR	Specify routing header
		IPv6_TCLASS	Specify traffic class
Unix domain	SQL_SOCKET	SCM_RIGHTS	Send/receive descriptors
		SCM_CREDS	Send/receive user credentials

Table 10.3: Summary of uses of ancillary data

Ancillary data consists of one or more *ancillary data objects*, each one beginning with a *cmsghdr* structure, defined by including<*sys/socket.h*>.

```
struct cmsghdr{
        socklen_t      cmsg_len;      /*length in bytes, including this structure*/
        int     cmsg_level;           /*originating protocol*/
        int     cmsg_type;            /*protocol-specific type*/
                       /*followed by unsigned char cmsg_data[]*/
};
```

The following five macros are defined by including the <sys/socket.h> header to simplify processing of the ancillary data: -

#include <sys/socket.h>

#include < sys/param.h>/* f or A LIGN m acro on m any implementations */

struct cmsghdr *CMSG_FIRSTHDR(struct msghdr *mhdrptr) ;

Returns: poi nter t o first c msghdr s tructure o r N ULL i f no ancillary data

struct cmsghdr *CMSG_NXTHDR(struct msghdr *mhdrptr, struct cmsghdr *cmsgptr) ;

Returns: poi nter t o ne xt cmsghdr s tructure or N ULL i f no more ancillary data objects

unsigned char *CMSG_DATA(struct cmsghdr *cmsgptr) ;

Returns: p ointer to f irst b yte o f d ata a ssociated w ith cmsghdr structure

unsigned int CMSG_LEN(unsigned int length) ;

Returns: va lue t o s tore in c msg_len g iven t he a mount of data

unsigned int CMSG_SPACE(unsigned int length) ;

Returns: to tal s ize o f an a ncillary data o bject given th e amount of data

Check Your Progress

1. *Write a brief note on flags that are examined by the kernel for the input and output functions.*

2. *What are the various uses of ancillary data?*

10.8 HOW MUCH DATA IS QUEUED?

There ar e times when we want to see how much data is queued to be r ead on a s ocket, without r eading t he data. There a re f ollowing techniques has to covered.

- If the goal is not to block in the kernel because we have something else t o do w hen nothing i s r eady t o be read, non blocking I/O can be used.
- If we w ant to e xamine th e d ata b ut s till le ave i t o n th e receive queue f or s ome ot her pa rt of our pr ocess t o r ead, w e c an us e the *MSG_PEEK* flag.
- Some implementations support the *FIONREAD* command of *ioctl.*

10.9 SOCKETS AND STANDARD I/O

One of the methods of performing I/O is the *standard I/O library*. It is specified by the ANSI C standard. The standard I/O library handles some of the details such as automatically buffering the input and output streams.

The standard I/O library can be used with sockets, but there are a few things to consider: -

- A standard I/O stream can be created from any descriptor by calling the *fdopen* function. Similarly, given a standard I/O stream, we can obtain the corresponding descriptor by calling *fileno*.
- TCP and UDP sockets are full-duplex. Standard I/O streams can also be full-duplex.
- The easiest way to handle this read-write problem is to open two standard I/O streams for a given socket: one for reading and one for writing.

  Standard I/O uses three types of buffering: -
- *Fully buffered*: I/O takes place only when the buffer is full, or the process calls *fflush* or *exit*.
- *Line buffered*: I/O takes place only when a new line is encountered, or the process calls *fflush* or *exit.*
- *Unbuffered*: I/O takes place each time a standard I/O output function is called.

Most UNIX implementations of the standard I/O library use the following rules:

- Standard error is always unbuffered.
- Standard input and standard output are fully buffered, unless they refer to a terminal device, in which case, they are line buffered.
- All other streams are fully buffered unless they refer to a terminal device, in which case, they are line buffered.

10.10 SUMMARY

There are three main ways to set a time limit on a socket operation:

(i) Use the *alarm* function and the *SIGALRM* signal,

(ii) Use the time limit that is provided by *select* and

(iii) Use the newer *SO_RCVTIMEO* and *SO_SNDTIMEO* socket options.

recvmsg and *sendmsg* are the most general of all the I/O functions provided. We know the various uses of ancillary data.

10.11 TERMINAL QUESTIONS

1. Write the three ways of performing I/O operations involving sockets.
2. Write syntax/format of (i) readv and writev function, (ii) recvmsg and sendmsg function.
3. What are the three types of buffering used by standard I/O?
4. What are the different methods to check queued data?
5. What are the different rules used by most UNIX implementations of the standard I/O library?
6. What are different macros are defined by including the <sys/socket.h> header to simplify processing of the ancillary data?

UNIT-11 : UNIX DOMAIN PROTOCOLS

Structure

11.1 INTRODUCTION

The UNIX domain protocols are not an actual protocol suite, but a way of performing client/server communication on a single host using the same API that is used for clients and servers on different hosts. The UNIX domain pr otocols a re an a lternative t o t he interprocess c ommunication (IPC) when the client and server are on the same host. It is important to note that the protocol addresses used to identify clients and servers in the UNIX domain are pathnames within the normal file system.

Two types of sockets provided in UNIX domain: -

- Stream sockets (similar to TCP).
- Datagram sockets (similar to UDP).

Reason for using UNIX domain sockets: -

- UNIX domain sockets are often twice as fast as a TCP socket when both peers are on same host.
- UNIX domain sockets are used when passing descriptors between processes on the same host.
- Newer i mplementations of U NIX dom ain s ockets pr ovide t he client's c redentials (user I D a nd g roup I Ds) to the server, w hich can provide additional security checking.

Also, we will learn about UNIX domain socket address structure, socket pair function and socket function are discussed to provide the insight view. We will see programs of UNIX domain stream/datagram client/server. Then there is discussion on topics descriptors passing and receiving sender credentials.

11.2 OBJECTIVES

At the end of this unit we get to know about: -

- What are UNIX domain protocols?
- Unix domain socket address structure
- Format of socket pair and socket functions
- UNIX domain stream client/server and UNIX domain datagram client/server
- Descriptors passing and receiving sender credentials

11.3 UNIX DOMAIN SOCKET ADDRESS STRUCTURE

UNIX domain socket address structure is defined by including the <*sys/un.h*>header.

```
          struct sockaddr_un{
                    sa_family_t          sun_family;
     /*AF_LOCAL*/
                    char          sun_path[104];          /*null-
terminated pathname*/
          };
```

(Unix domain socket address structure: socketaddr_un)

The pathname stored in the sun_path array must be null-terminated. The macro SUN_LEN is provided and it takes a pointer to a sockaddr_un structure and returns the length of the structure, including the number of non-null bytes in the pathname.

The below program creates a unix domain socket, binds a pathname to it and then calls get sockname and prints the bound pathname.

```
#include "unp.h"

int

main (int argc, char **argv)

{
 int sockfd;
socklen_t len;
struct sockaddr_un addr1, addr2;
if (argc != 2)
err_quit("usage: unixbind <pathname>");
sockfd = Socket(AF_LOCAL, SOCK_STREAM, 0);
unlink(argv[1]); /* OK if this fails */
bzero(&addr1, sizeof(addr1));
addr1.sun_family = AF_LOCAL;
strncpy(addr1.sun_path, argv[1], sizeof(addr1.sun_path) - 1);
Bind(sockfd, (SA *) &addr1, SUN_LEN(&addr1));
len = sizeof(addr2);
Getsockname(sockfd, (SA *) &addr2, &len);
printf("bound name = %s, returned len = %d\n", addr2.sun_path,
len);
exit(0);
 }
```

In the program, the pathname that bind to the socket is the command-line argument. B ut th e b ind w ill f ail if th e p athname a lready exists in th e filesystem. T herefore, it calls unlink t o de lete t he pa thname, i n cas e i t already exists. If i t do es not e xist, unl ink r eturns a n e rror, which i gnore bind a nd t hen ge tsockname. C opy t he command-line a rgument u sing strncpy, t o avoid ove rflowing t he s tructure i f t he pa thname i s t oo l ong. Since initialize the structure to zero and then subtract one from the size of the s un_path a rray, t he pathname i s nul l-terminated. A fter th at b ind is called and use the macro SUN_LEN to calculate the length argument for the function. Then call getsockname to fetch the name that was just bound and print the result.

11.4 *SOCKETPAIR* FUNCTION

This function creates two sockets that are then connected together. This f unction a pplies onl y t o UNIX domain s ockets. It includes header *<sys/socket.h>*

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

int socketpair(int *family*, int *type*, int *protocol*, int *sockfd[2]*);

Here, family - AF_LOCAL

protocol- 0

type - SOCK_STREAM or SOCK_DGRAM

and two socket descriptor that are returned are sockfd[0] and sockfd[1].

The result of socketpair with a type of SOCK_STREAM is called a stream pipe. It is similar to a regular Unix pipe, but a stream pipe is full-duplex; that is both descriptors can be read and written.

Check Your Progress:

1. *Write the difference between Unix pipe and stream pipe.*

2. *Can you create a function for socket pair?*

11.5 SOCKET FUNCTIONS

While using UNIX domain sockets there exists several differences and restrictions in the socket functions. Below is the list of POSIX requirements when applicable and note that not all implementations are currently at this level.

- The default file access permissions for a pathname created by bind should be 0777, modified by the current unmask value.
- The pathname associated with a UNIX domain socket should be an absolute pathname, not a relative pathname.
- The pathname specified in a call to connect must be a pathname that is currently bound to an open UNIX domain socket of the same type (stream or datagram).
- The permission testing associated with the *connect* of a UNIX domain socket is the same as if *open* had been called for write-only access to the pathname.
- UNIX domain stream sockets provide a byte stream interface to the process with no record boundaries.
- If socket's queue is full then ECONNREFUSED is returned in response to a call to *connect* for UNIX domain stream socket.
- UNIX domain datagram sockets are similar to UDP sockets.
- Unlike UDP sockets, sending a datagram on an unbound UNIX domain datagram socket does not bind a pathname to the socket.

11.6 UNIX DOMAIN STREAM CLIENT/SERVER

UNIX domain stream client/server uses stream socket. Most of the steps are similar to TCP echo client/server. But some modifications have been dow n l ike associating p athname t o UNIX. The s teps f or cr eating UNIX domain stream server are following: -

- Call s ocket ( ) - A c all to socket ( ) with t he p roper a rguments creates the UNIX socket. Here, we will pass SOCK_STREAM as second argument to create a stream socket.
- We first unlink the pathname, in case it exists from an earlier run of the server, and then initialize the socket address structure before calling bind (). Rest of steps is same as of TCP echo client/server.
- Call bind ( )- We get a socket descriptor from the call to socket(), now bind that to an address in the UNIX domain.
- Call listen ( ) - This i nstructs th e s ocket to lis ten f or in coming connections from client programs.
- Call accept ( ) - This will accept a connection from a client.
- Close the connection.

In o rder t o cr eate UNIX domain s tream c lient s ome mo difications h ave been made to TCP client/socket. The steps are given below: -

- The socket address structure to contain the server's address is now a *sockaddr_un* structure.
- Call s ocket ( ) -The f irst a rgument t o s ocket i s A F_LOCAL a nd second one is SOCK_STREAM.
- Call connect ( ) – To connect with server.

The program below shows the unix domain stream protocol echo server.

```
#include    "unp.h"
 int
 main (int argc, char **argv)
 {
 int listenfd, connfd;
 pid_t childpid;
 socklen_t clilen;
 struct sockaddr_un cliaddr, servaddr;
 void sig_chld(int);
 listenfd = Socket(AF_LOCAL, SOCK_STREAM, 0);
 unlink(UNIXSTR_PATH);
 bzero(&servaddr, sizeof(servaddr));
servaddr.sun_family=AF_LOCAL;
strcpy(servaddr.sun_path, UNIXSTR_PATH);
 Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
 Listen(listenfd, LISTENQ);
 Signal(SIGCHLD, sig_chld);
 for ( ; ; ) {
 clilen = sizeof(cliaddr);
 if ( (connfd = accept(listenfd, (SA *) &cliaddr, &clilen)) < 0) {
 if (errno == EINTR)
 continue; /* back to for() */
 else
 err_sys("accept error");
 }
 if ( (childpid = Fork()) == 0) { /* child process */
 Close(listenfd); /* close listening socket */
 str_echo(connfd); /* process request */
 exit(0);
 }
 Close(connfd);
}
}
```

The program of the server is use the Unix domain stream protocol instead of T CP. The d atatype of t he t wo s ocket a ddress s tructures i s no w sockaddr_un. The f irst argument t o s ocket i s AF_LOCAL, t o cr eate a Unix domain stream socket. The constant UNIXSTR_PATH is defined in unp.h to be /tmp/unix.str. First unlink the pathname, in case it exists from an earlier run of the server and then initialize the socket address structure before cal ling bind. A n e rror fro m unlink is acc eptable. The s tream protocol echo server program in bind call it specify the size of the socket address structure (the third argument) as the total size of the sockaddr_un structure, not j ust t he num ber of b ytes oc cupied b y t he pa thname. B oth lengths are valid since the pathname must be null-terminated.

The program below shows the Unix domain stream protocol echo client

```
#include "unp.h"
 int
 main(int argc, char **argv)
 {
 int sockfd;
 struct sockaddr_un servaddr;
 sockfd = Socket(AF_LOCAL, SOCK_STREAM, 0);
 bzero(&servaddr, sizeof(servaddr));
 servaddr.sun_family = AF_LOCAL;
 strcpy(servaddr.sun_path, UNIXSTR_PATH);
 Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
Str_cli(atdin,sockfd);
Exit(0);
}
```

The s ocket a ddress s tructure t o c ontain t he s erver's a ddress i s a sockaddr_un structure. The first argument to socket is AF_LOCAL. The code to fill in the socket address structure is identical to the code shown in the pr evious pr ogram for the server: Initialize th e structure to 0, set th e family to AF_LOCAL, and copy the pathname into the sun_path member.

11.7 UNIX DOMAIN DATAGRAM CLIENT /SERVER

UNIX domain d atagram c lient/server requires mo dification to UDP e cho c lient/server. Important poi nts i n creating UNIX domain datagram echo server are following: -

- The da tatype of t he two s ocket a ddress s tructures i s now *sockaddr_un*.
- The f irst ar gument t o s ocket i s A F_LOCAL, t o cr eate a UNIX domain datagram socket.
- We first unlink the pathname, in case it exists from an earlier run of the server, and then initialize the socket address structure before calling bind ().
- Others are same as UDP echo server.

Similarly, in UNIX domain datagram echo client, some modifications have been done.

- The socket address structure to contain the server's address is now a *sockaddr_un* structure. We also allocate one of these structures to contain the client's address.
- The first argument to socket is AF_LOCAL.
- Unlike our UDP client, when using the UNIX domain datagram protocol, we must explicitly bind a pathname to our socket so that the server has a pathname to which it can send its reply. Other is same as UDP echo client.

The program below shows the Unix domain datagram protocol echo server.

```
#include "unp.h"
int
main(int argc, char **argv)
{
int sockfd;
struct sockaddr_un servaddr, cliaddr;
sockfd = Socket(AF_LOCAL, SOCK_DGRAM, 0);
unlink(UNIXDG_PATH);
bzero(&servaddr, sizeof(servaddr));
servaddr.sun_family = AF_LOCAL;
strcpy(servaddr.sun_path, UNIXDG_PATH);
Bind(sockfd, (SA *) &servaddr, sizeof(servaddr));
dg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
}
```

In the program, the datatype of the two socket address structures is now sockaddr_un. The first argument to socket is AF_LOCAL, to create a Unix domain datagram socket. The constant UNIXDG_PATH is defined in unp.h to be /tmp/unix.dg. First unlink the pathname, in case it exists from an earlier run of the server, and then initialize the socket address structure before calling bind. An error from unlink is acceptable. The dg_echo function is used.

The program below shows the Unix domain datagram protocol echo client

```
#include "unp.h"
```

```
 int
 main(int argc, char **argv)
 {
 int sockfd;
 struct sockaddr_un cliaddr, servaddr;
 sockfd = Socket(AF_LOCAL, SOCK_DGRAM, 0);
bzero(&cliaddr,sizeof(cliaddr));
cliaddr.sun_family=AF_LOCAL;
strcpy(cliaddr.sun_path,tmpnam(NULL));
Bind(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
bzero(&servaddr, sizeof(servaddr));
servaddr.sun_family=AF_LOCAL;
strcpy(servaddr.sun_path,UNIXDG_PATH);
dg_cli(stdin,sockfd,(SA*) &servaddr,sizeof(servaddr));
exit(0);
}
```

In the program, the socket address structure to contain the server's address is now a sockaddr_un structure. Also allocate one of these structures to contain the client's address. The first argument to socket is AF_LOCAL. Unlike our UDP client, when using the Unix domain datagram protocol, then must explicitly bind a pathname to socket so that the server has a pathname to which it can send its reply. The code to fill in the socket address structure with the server's well-known pathname is identical to the code shown earlier for the server. The function dg_cli is the used.

Check Your Progress

1. *Write the difference between Unix domain datagram and Unix domain stream client/server.*

11.8 PASSING DESCRIPTORS

Steps involved in passing a descriptor between two processes are as follows: -

- Create UNIX domain sockets either a stream socket or a datagram socket and connect them for communication between a server and a client.
- One process opens a descriptor. Any type of descriptor can be exchanged.

- Sender builds an *msghdr* structure containing the descriptor to be passed, and calls *sendmsg* with the structure across one of the UNIX domain sockets.
- Reciever calls *recvmsg* to receive the descriptor from the other UNIX domain socket.

Client and server must have an application protocol so they know when the descriptor is to be passed.

11.9 RECEIVING SENDER CREDENTIALS

When a client and server communicate using UNIX domain protocols, the server often needs a way to know exactly who the client is, to validate that the client has permission to ask for the service being requested.

FreeBSD passes credentials in a *cmsgcred* structure, which is defined by including the<*sys/socket.h*>header.

```
structcmsgcred {
        pid_t cmcred_pid;          /* PID of sending process */
        uid_t cmcred_uid;          /* real UID of sending process */
        uid_t cmcred_euid;         /* effective UID of sending process */
        gid_t cmcred_gid;          /* real GID of sending process */
        short cmcred_ngroups;      /* number of groups */
        gid_t cmcred_groups[CMGROUP_MAX];
                /* groups */
};
```

***The program below shows the* read_cred function that reads and returns sender's credentials.**

```
 #include "unp.h"
#defineCONTROL_LEN(sizeof(structcmsghdr)+sizeof(struct cmsgcred))
ssize_t
 read_cred(int fd, void *ptr, size_t nbytes, struct cmsgcred *cmsgcredptr)
{
```

```
struct msghdr msg;
struct iovec iov[1];
char control[CONTROL_LEN];
int n;
msg.msg_name = NULL;
msg.msg_namelen = 0;
iov[0].iov_base = ptr;
iov[0].iov_len = nbytes;
msg.msg_iov = iov;
msg.msg_iovlen = 1;
msg.msg_control = control;
msg.msg_controllen = sizeof(control);
msg.msg_flags = 0;
if ( (n = recvmsg(fd, &msg, 0)) < 0)
return (n);
cmsgcredptr->cmcred_ngroups  = 0;  / * i ndicates no c  redentials
returned */
if (cmsgcredptr && msg.msg_controllen > 0) {
struct cmsghdr *cmptr = (struct cmsghdr *) control;
if (cmptr->cmsg_len < CONTROL_LEN)
err_quit("control length = %d", cmptr->cmsg_len);
if (cmptr->cmsg_level != SOL_SOCKET)
err_quit("control level != SOL_SOCKET");
if (cmptr->cmsg_type != SCM_CREDS)
err_quit("control type != SCM_CREDS");
memcpy(cmsgcredptr, C      MSG_DATA(cmptr), s      izeof(struct
cmsgcred));
}
return (n);
}
```

In t he program, the f irst th ree a rguments a re id entical to r ead, w ith th e fourth argument being a pointer to a cmsgcred structure that will be filled in. If credentials were returned, the length, level, and type of the ancillary data are verified, and the resulting structure is copied back to the caller. If no credentials were returned, then set the structure to 0. Since the number of groups (cmcred_ngroups) is always 1 or more, the value of 0 indicates to the caller that no credentials were returned by the kernel. The main function for echo server, str_echo function is called by the child after the parent has accepted a new client connection and called fork. If credentials were returned, they are printed. The further code reads buffers from the client and writes them back to the client.

The program below shows the str_echo function that asks for client credentials

```
#include "unp.h"

ssize_t read_cred(int, void *, size_t, struct cmsgcred *);

 void

 str_echo(int sockfd)

 {

 ssize_t n;

 int i;

 char buf[MAXLINE];

 struct cmsgcred cred;

 again:

while ( (n = read_cred(sockfd, buf, MAXLINE, &cred))>0) {

 if (cred.cmcred_ngroups == 0) {

printf("(no credentials returned)\n");

 } else {

 printf("PID of sender = %d\n", cred.cmcred_pid);

 printf("real user ID = %d\n", cred.cmcred_uid);

 printf("real group ID = %d\n", cred.cmcred_gid);

 printf("effective user ID = %d\n", cred.cmcred_euid);

 printf("%d groups:", cred.cmcred_ngroups - 1);

 for (i = 1; i < cred.cmcred_ngroups; i++)

 printf(" %d", cred.cmcred_groups[i]);

printf("\n");

 }

writen(sockfd, buf, n);
```

```
    }
    if (n < 0 && errno == EINTR)
    goto again
    else if(n<0)
      err_sys("str_echo:read error");
    }
```

In the program, the main function for echo server is str_echo function. This function is called by the child after the parent has accepted a new client connection and called fork. If credentials were returned, they are printed. Further code reads buffers from the client and writes them back to the client. Here client is to pass an empty cmsgcred structure that will be filled in when it calls sendmsg.

11.10 SUMMARY

UNIX domain sockets are an alternative to IPC when the client and server are on the same host. The advantage in using UNIX domain sockets over some form of IPC is that the API is nearly identical to a networked client/server. We modified our TCP and UDP echo clients and servers to use the UNIX domain protocols and the only major difference had to bind a pathname to the UDP client's socket, so that the UDP server had somewhere to send the replies. Descriptor passing is a powerful technique between clients and servers on the same host and it takes place across a UNIX domain socket.

11.11 TERMINAL QUESTIONS

1. Explain UNIX domain protocol.
2. Write two types of socket provided in UNIX domain socket.
3. Write structure of UNIX domain socket address.
4. What is use of Bind ( ) system call?
5. Write a program to show read_cred function that reads and returns sender's credentials.
6. Write a program to show the str_echo function that asks for client credentials.

Uttar Pradesh Rajarshi Tandon
Open University

Bachelor of Computer Application

BCA-E7
Network Programming

Block

4

Broadcast, Multicast and Inter Process Communication

© **UPRTOU, Prayagraj. 2019**
ISBN : 978-93-83328-11-6

Printed and Published by Dr. Arun Kumar Gupta Registrar, Uttar Pradesh Rajarshi Tandon Open University, 2019.

Printed By : Chandrakala Universal Pvt. Ltd. 42/7 Jawahar Lal Neharu Road, Prayagraj.

BLOCK INTRODUCTION

The objective of this course is to introduce the basic concept about the network programming as well as provides a mix of practical experience and a depth of understanding. The network programming course address today's most crucial standards, implementations and techniques. The aim is to provide an extensive variety of topics on this subject with appropriate examples. The course is organized into following blocks:

Block 4 covers broadcasting, multicast, inter process communication and remote login.

UNIT 12 : BROADCASTING

Structure

12.1 INTRODUCTION

In th is unit, we w ill l earn about; br oadcasting and i ts us es, broadcast a ddress, di fference be tween uni cast a nd br oadcast addressing, *dg_cli* function using broadcasting and race conditions.

There ar e f our types of a ddressing; U nicast, Anycast, M ulticast a nd Broadcast. Unicasting i s pr ocessing t alking to e xactly one another process, f or e xample T CP. A ny casting i s a dded i n IPv6 a ddressing architecture. M ulticasting s upport i s opt ional i n IPv4 but m andatory i n IPv6. Broadcasting i s not a vailable in I Pv6. A ny I Pv6 a pplication th at uses br oad c asting m ust be r ecorded IPv6 t o use m ulticasting. B road casting an d m ulticasting require d ata gram t ransport s uch as U DP o r raw IP, they cannot work with TCP.

12.2 OBJECTIVES

At the end of this unit, you should be able to: -

- Know what is broadcasting, its uses.
- How to write broadcast address.
- Able to differentiate between unicast and broadcast address.
- Able to write dg_cli function that broadcasts.
- Know race conditions.

12.3 BROADCAST ADDRESSES

Broadcasting refers to transferring a message to all the recipients. Broadcasting require datagram transport like UDP or raw IP, it cannot work with TCP.

Uses

- To locate a server on the local subnet when the server is assumed to be on the local subnet but its unicast IP address is not known. This is sometimes called *resource discovery*.
- To minimize the network traffic on a LAN when there are multiple clients communicating with a single server.

If we denote an IPv4 address as { *subnetid*, *hostid*}, where *subnetid* represents the bits that are covered by the network mask (or the CIDR prefix) and *hostid* represents the bits that are not covered, then we have **two types of broadcast addresses**. We denote a field containing all one bits as –1.

a) *Subnet-directed broadcast address - <subnetid, -1>*: This addresses all the interfaces on the specified subnet. For example, if we have the subnet 192.168.42/24, then 192.168.42.255 would be the subnet-directed broadcast address for all interfaces on the192.168.42/24 subnet.

b) *Limited broadcast address* -<-1, -1, -1> or 255.255.255.255: Datagrams destined to this address must never be forwarded by a router.

Check Your Progress

1. *Can you explain the subnetid and hostid?*

2. *Can you explain types of broadcast addresses?*

12.4 UNICAST VERSUS BROADCAST

Unicast is the term used to describe communication where a piece of information is sent from one point to another point. In this case there is just one sender, and one receiver. Unicast uses IP delivery methods such as Transmission Control Protocol (TCP) and User Datagram Protocol (UDP), which are session-based protocols. When a Windows Media Player client connects using unicast to a Windows Media server, that client has a direct relationship to the server. Each unicast client that connects to the server takes up additional bandwidth. For example, if you have 10 clients all playing 100-kilobits per second (Kbps) streams, those clients as a group are taking up 1,000 K bps. If you have only one client playing the 100 Kbps stream, only 100 Kbps is being used.

Broadcast is the term used to describe communication where a piece of information is sent from one point to all other point. In this case there is just one sender, but the information is sent to all connected receivers.

12.5 DG_CLI FUNCTION USING BROADCASTING

The *dg_cli* function is used to perform most of the client processing in UDP echo client. In order to broadcast to the standard UDP daytime server and printing all replies we make some modifications to *dg_cli function*. In *main ( )* function we change the destination port number to 13.

```
servaddr.sin_port = htons(13);
```

The working of *dg_cli* function is as follows: -

- Allocate room for server's address, set socket option.
- Read line; send to socket, read all replies.
- Print each received reply.

The dg_cli function that broadcasts as shown below:

```
#include "unp.h"

static void recvfrom_alarm(int);

void

dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)

{

        int n;

        const int on = 1;

        char sendline[MAXLINE], recvline[MAXLINE + 1];

  socklen_t len;

        struct sockaddr *preply_addr;

        preply_addr = Malloc(servlen);

        Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));

        Signal(SIGALRM, recvfrom_alarm);

        while (Fgets(sendline, MAXLINE, fp) != NULL) {

                Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
                alarm(5);
                for ( ; ; ) {
                        len = servlen;
                        n = recvfrom(sockfd, recvline, MAXLINE, 0,
                        preply_addr, &len);
                        if (n < 0) {
                                if (errno == EINTR)
                                        break; /* waited long enough
for replies */
                                else
                                        err_sys("recvfrom error");
                        } else {
                                recvline[n] = 0; /* null terminate */
                                printf("from %s: %s",
                                Sock_ntop_host  (preply_addr,  len),
recvline);
                        }
                }
        }
        free(preply_addr);
}
static void
recvfrom_alarm(int signo)
{
        return; /* just interrupt the recvfrom() */
}
```

The dg_cli function sets the SO_BROADCAST socket option and prints all the replies received within five seconds. In the program, the malloc allocates room for the server's address to be returned by recvfrom. The SO_BROADCAST socket option is set and a signal handler is installed for SIGALRM. The next two steps, fgets and sendto of this function are sending a broadcast datagram, receive multiple replies, call recvfrom in a loop and print all the replies received within five seconds. After five seconds, SIGALRM is generated, signal handler is called, and recvfrom returns the error EINTR. For each reply received, in the program call sock_ntop_host, which in the case of IPv4 returns a string containing the dotted-decimal IP address of t he s erver. T his i s pr inted a long w ith th e server's reply.

Check Your Progress

1. *Can you explain the major steps of dg_cli function?*

A race condition is a situation which occurs usually when multiple processes are a ccessing data t hat i s s hared among t hem, b ut t he co rrect outcome depends on the execution order of the processes.

Race conditions are always a co ncern with threads programming since so much data is shared among all the threads (e.g., all the global variables). Race conditions of a different type often exist when dealing with signals. The pr oblem oc curs because a s ignal can n ormally b e d elivered at an y time while our program is executing. POSIX allows us to block a signal from being delivered, but this is often of little use while we are performing I/O operations.

A race condition exists in above program (*dg_cli* function using broadcast) if force the condition to occur as follows: Change the argument to alarm from 5 to 1, and add sleep (1) immediately before the printf. When make these changes to the function and then type the first line of input, the line is sent as a broadcast and set the alarm for one second in the future. The block in the call to recvfrom, and the first reply then arrives for our socket, probably within a few milliseconds. The reply is returned by recvfrom, but we then go to sleep for one second. Additional replies are received, and they are placed into our sockets receive buffer. But while we are asleep, the alarm t imer ex pires an d t he S IGALRM s ignal i s g enerated: s ignal handler is called, and it just returns and interrupts the sleep in which we are b locked. Then l oop around a nd read the que ued r eplies w ith a one-second p ause e ach t ime w e p rint a r eply. W hen w e h ave read al l the replies, w e b lock a gain in th e c all to r ecvfrom, b ut th e timer is n ot running. T hus, w e w ill block f orever i n r ecvfrom. T he f undamental problem is that our intent is for our signal handler to interrupt a blocked recvfrom, but t he s ignal c an b e de livered at a ny time, and w e c an be executing anywhere in the infinite for loop when the signal is delivered.

12.7 SUMMARY

In this unit, we have covered about broadcast addresses, difference between uni cast a nd br oadcast, *dg_cli* function u sing b roadcasting an d also studied race conditions.

Broadcasting sends datagram that all hosts on the attached subnet receive. While unicasting sends datagram to a single intended host.

There are two ways to write broadcast address

(i) subnet-directed broadcast address and

(ii) limited broadcast address.

12.8 TERMINAL QUESTIONS

1. Explain broadcast address and its uses.

2. Explain difference between unicast and broadcast?

3. Explain dg_cli function using broadcasting.

4. What is race condition?

5. Write a dg_cli function with race condition.

UNIT 13 : MULTICASTING

Structure

13.1 INTRODUCTION

In this unit, we will learn details about multicasting. We will study multicasting on LAN and its difference with broadcasting. Then we get knowledge about multicasting on WAN. Then we see difficulties related to multicasting on WAN and discuss solutions to it in terms of source specific multicast. Then we will have a look over multicast socket options and *mcast_join* and related functions. We will see *dg_cli* function using multicasting. Then there will be discussion on topics Receiving IP Multicast Infrastructure Session Announcements, Sending and Receiving, and Simple Network Time Protocol (SNTP).

Multicast address identifies a set of IP interfaces. A multicast datagram should be received by only those interfaces interested in the datagram, that is, by the interfaces on the host running applications wishing to participate in the multicast group. Multicasting is used on a LAN or across a WAN. Indeed, applications multicast across a subset of internet on a daily basis.

13.2 OBJECTIVES

At the end of this unit, you should be able to know about: -

- Multicast and multicast address.
- Difference between Multicasting versus Broadcasting on a LAN
- Multicasting on a WAN and Source-Specific multicast
- Multicast Socket Options
- mcast_join and related functions and dg_cli Function using multicasting
- Receiving IP Multicast Infrastructure Session Announcements
- Sending and Receiving multicast datagrams
- Simple Network Time Protocol (SNTP)

13.3 MULTICAST ADDRESSES

A multicast address is a logical identifier for a group of hosts in a computer network that are available to process datagrams or frames intended to be multicast for a designated network service. Multicast addresses identify a set of IP interfaces. IP multicast address ranges and uses are shown below in table 13.1.

Range Start Address	Range End Address	Description
224.0.0.0	**224.0.0.255**	Reserved for special "well-known" multicast addresses.
224.0.1.0	**238.255.255.255**	Globally-scoped (Internet-wide) multicast addresses.
239.0.0.0	**239.255.255.255**	Administratively-scoped (local) multicast addresses.

Table 13.1 : IP Multicast Address Ranges and Uses

IPv4 Multicast addresses

In IPv4, class D addresses ranging from 224.0.0.0 to 239.255.255.255 are the multicast addresses. The lower order 28 bits of class D address form the multicast *groupID* and the 32-bit address is called the *group address.*

Mapping IPv4 multicast address to Ethernet address involves copy of low-order 23 bits of multicast address to low-order 23 -bits of the Ethernet address. The high order 24 bits of the Ethernet address are always *01:00:5e a*nd the next bit is always 0.

IPv6 Multicast Addresses

The high-order byte of an IPv6 multicast address has the value *ff.*

The mapping from a 16-byte IPv6 multicast address into a 6-byte Ethernet address involves copy of low-order 32 bits of the group address into the low-order 32 bits of the Ethernet address. The high-order 2 bytes of the Ethernet address are 33:33.

The table 13.2 and figure 13.1 shown the IPv6 Multicast Address Format.

Field Name	**Size (bits)**	**Description**
(Indicator)	8	The first eight bits are always " 1111 1111" to indicate a multicast address.
Flags	4	Four bits are reserved for flags that can be used to indicate the nature of certain multicast addresses. At the present time, the first three of these are unused and set to zero. The fourth is the "*T*" (*Transient*) flag. If left as zero, this marks the multicast address as a permanently-assigned, "well-known" multicast address. If set to one, this means this is a *transient* multicast address, meaning that it is not permanently assigned.
Scope ID	4	Four bits are used to define the scope of the multicast address; 16 different values from 0 to 15 are possible. **Scope ID Value** **Multicast Address Scope** 0 Reserved 1 Node-Local Scope 2 Link-Local Scope 5 Site-Local Scope 8 Organization-Local Scope 14 Global Scope 15 Reserved
Group ID	112	***Group ID:*** Defines a particular group within each scope level.

Table 13.2 : IPv6 Multicast Address Format

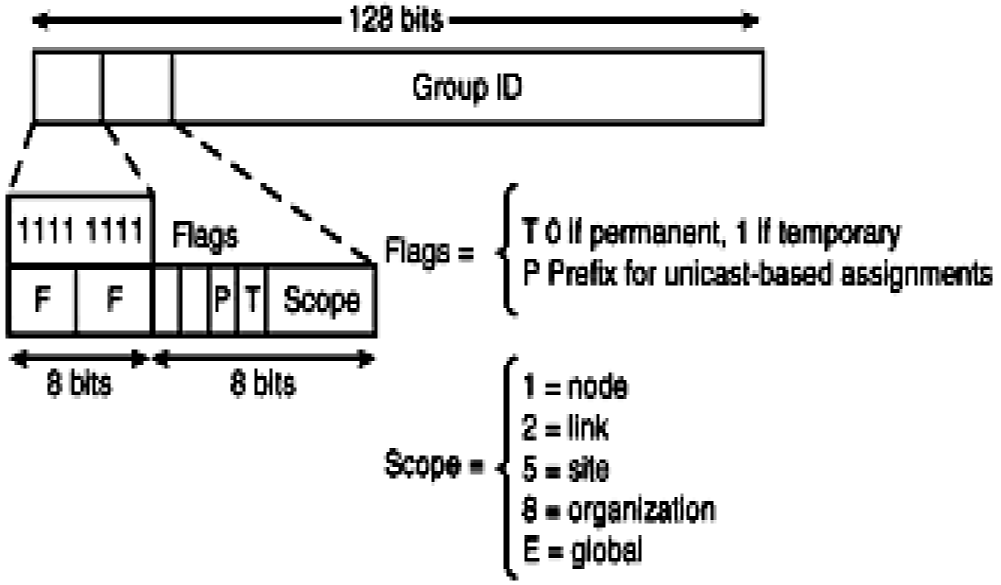


Figure 13.1 : IPv6 Multicast Address Format

13.4 MULTICASTING VERSUS BROADCASTING ON A LAN

Broadcast is a term used to describe communication where a piece of information is sent from one point to all other points. In network case, there i s j ust one s ender, but the i nformation i s s ent t o a ll c onnected receivers. Broadcast i s mostly u sed i n l ocal sub ne tworks. In or der t o transmit b roadcast p acket, t he d estination M AC a ddress i s s et t o FF:FF:FF:FF:FF:FF and all such packets will be received by other NICs.

Multicast i s a t erm u sed t o d escribe communication w here a p iece o f information is sent from a source host to a group of destination hosts. The notion of gr oup i s e ssential t o t he c oncept of multicasting. A m ulticast group a ddress i s d efined. A ll t he hos ts t hat ha ve j oined t his group w ill receive messages sent to this multicast group address.

13.5 MULTICASTING ON A WAN

We know W AN pr ovides l ong di stance t ransmission of da ta, image, a udio a nd vi deo i nformation ove r l arge g eographical areas t hat may comprise a country, a c ontinent, or even the w hole w orld. WAN i .e wide area network is a combination of LANs connected through routers.

In order to have multicasting on a WAN we need to have multicast routers for connecting LAN. M ulticast r outers c ommunicate e ach ot her us ing multicast r outing pr otocol (MRP). G roup of hos ts be longing t o di fferent LANs ma y f orm a mu lticast group. A n ew h ost c an jo in th e mu lticast group b y s ending a n IGMP t o a ny a ttached multicast router which t hen exchanges this information with other multicast routers using MRP.

Suppose a host on a LAN want to send a message to a multicast group on a WAN. It will multicast the message to its LAN. Other hosts on this LAN belonging to the required multicast group will receive this message. Multicast router attached to this LAN will also receive the message. This multicast router will then send the message to another multicast router attached to it. All multicast router will then multicast on its respective LAN and multicast routers attached to it. Intended recipient on LAN will then receive message from multicast.

Multicasting on a WAN has been difficult to deploy for several reasons.

- The biggest problem is that the MRP needs to get the data from all the senders, which may be located anywhere in the network, to all receivers, which may similarly be located anywhere.
- Another large problem is multicast address allocation. There are not enough IPv4 multicast addresses to statically assign them to everyone who wants one, as is done with unicast addresses.

13.6 SOURCE-SPECIFIC MULTICAST

Source-specific multicast (SSM) is a method of delivering multicast packets in which the only packets that are delivered to a receiver are those originating from a specific source address requested by the receiver.

SSM combines the group address with a system's source address, which solves the multicasting problems in WAN by the following ways:

- The receivers supply the sender's source address to the routers as part of joining the group.
- It redefines the identifier from simply being a multicast group address to being a combination of a unicast source and multicast destination.

IP version 4 (IPv4) addresses in the 232/8 (232.0.0.0 to 232.255.255.255) range are designated as source-specific multicast (SSM) destination addresses and are reserved for use by source- specific applications and protocols. For IP version 6 (IPv6), the address prefix FF3x: :/32 is reserved for source-specific multicast use. This document defines an extension to the Internet network service that applies to datagrams sent to SSM addresses and defines the host and router requirements to support this extension.

Check Your Progress

1. *Can you explain the difference between multicast and broadcast?*
2. *What are the advantages of SSM ?*

13.7 MULTICAST SOCKET OPTION

The API support for traditional multicasting requires only five new socket options. Source-filtering support, which is required for SSM, adds four more. The following are the multicast socket options: -

- IP_MULTICAST_IF – Specify default interface for outgoing multicasts.
- IP_MULTICAST_TTL – Specify TTL for outgoing multicasts.
- IP_MULTICAST_LOOP – Enable or disable loopback of outgoing multicasts.
- IPV6_MULTICAST_IF – Specify default interface for out going multicasts.
- IPV6_MULTICAST_HOPS – Specify hop limit for out going multicasts.
- IPV6_MULTICAST_LOOP – Enable or disable loopback of outgoing multicasts.

Working with multicast sockets and UNIX (FreeBSD) as follows:

1. Sending socket: - In general, there's nothing special you need to do on the sending end. The key is simply to send to a multicast IP (group) address. Tips:
   - ❖ Use socket ( ) with AF _INET and SOCK_DGRAM arguments as normal.
   - ❖ Use bind () to associate this socket with a local address and port.
   - ❖ Do not attempt to associate the socket with a multicast destination address using connect ().
   - ❖ Use sendto () for sending data.
2. Receiving socket: - Receiving is nearly the same, but with one additional system call setsockopt ().
   - ❖ Use socket ( ) with AF _INET and SOCK_DGRAM arguments as normal.
   - ❖ Use setsockopt () with the IP_ADD_MEMBERSHIP option. This tells the system to receive packets on the network whose destination is the group address (but not its own).

13.8 *mcast_join* and RELATED FUNCTIONS

The multicast socket options for IPv4 are similar to the multicast socket options for IPv6, there are enough differences that protocol-independent c ode using mu lticasting b ecomes complicated w ith lo ts o f #ifdefs. A b etter s olution is to h ide th e d ifferences w ithin th e f ollowing eight functions:

#include "unp.h"
int mc ast_join(int *sockfd*, c onst s truct s ockaddr * *grp*, s ocklen_t *grplen*, const char **ifname*, u_int *ifindex*);
int mcast_leave(int *sockfd*, const struct sockaddr **grp*, socklen_t *grplen*);
int m cast_block_source(int *sockfd*, const s truct s ockaddr * *src*, socklen_t *srclen*, const struct sockaddr **grp*, socklen_t *grplen*);
int m cast_unblock_source(int *sockfd*, c onst struct s ockaddr * *src*, socklen_t *srclen*, const struct sockaddr **grp*, socklen_t *grplen*);
int m cast_join_source_group(int *sockfd*, c onst s truct s ockaddr * *src*, socklen_t *srclen*, c onst s truct s ockaddr **grp*, s ocklen_t *grplen*, c onst c har **ifname*, u_int *ifindex*);
int m cast_leave_source_group(int *sockfd*, c onst s truct s ockaddr * *src*, socklen_t *srclen*, const struct sockaddr **grp*, socklen_t *grplen*);
int mcast_set_if(int *sockfd*, const char **ifname*, u_int *ifindex*);
int mcast_set_loop(int *sockfd*, int *flag*);
int mcast_set_ttl(int *sockfd*, int *ttl*);
All above return: 0 if OK, –1 on error
int mcast_get_if(int *sockfd*);
Returns: non-negative interface index if OK, –1 on error
int mcast_get_loop(int *sockfd*);
Returns: current loopback flag if OK, –1 on error
int mcast_get_ttl(int *sockfd*);
Returns: current TTL or hop limit if OK, –1 on error

- *mcast_join* joins the any-source multicast group whose IP address is within the socket address structure pointed to by *grp,* and whose length is specified by *grplen.*
- *mcast_leave* leaves the multicast group whose IP address is contained within the socket address structure pointed to by *grp.*
- *mcast_block_source* blocks reception on the given socket of the source and group whose IP address are contained within the socket address structures pointed to by *src and grp,* respectively, and whose lengths are specified by *srclen* and *grplen.*
- *mcast_unblock_source* unblocks reception of traffic from the given source to the given group.
- *mcast_join_source_group* joins the source-specific group where the source and group IP addresses are contained within the socket address structures pointed to by *src and grp,* respectively, and whose lengths are specified by *srclen* and *grplen.*
- *mcast_leave_source_group* leaves the source-specific group whose source and group IP addresses are contained within the socket address structures pointed to by *src and grp,* respectively, and whose lengths are specified by *srclen* and *grplen.*
- *mcast_set_if* sets the default interface index for outgoing multicast datagrams.
- *mcast_set_loop* sets the loopback option to either 0 or 1, and *mcast_set_ttl* sets either the IPv4 TTL or the IPv6 hop limit.

The program below shows the first third of mcast_join function. The program shows how straightforward the protocol-independent API can be.

```
#include    "unp.h"
#include    <net/if.h>
int
mcast_join(int sockfd, const SA *grp, socklen_t grplen,
        const char *ifname, u_int ifindex)
{
#ifdef MCAST_JOIN_GROUP
   struct group_req req;
   if (ifindex > 0) {
        req.gr_interface = ifindex;
    } else if (ifname != NULL) {
        if ( (req.gr_interface = if_nametoindex(ifname)) == 0) {
            errno = ENXIO;      /* i/f name not found */
            return (-1);
        }
    } else
        req.gr_interface = 0;
    if (grplen > sizeof(req.gr_group)) {
        errno = EINVAL;
        return -1;
    }
    memcpy(&req.gr_group, grp, grplen);
    return (setsockopt(sockfd, family_to_level(grp->sa_family),
                    MCAST_JOIN_GROUP, &req, sizeof(req)));
 #else
```

In the program, the caller is supplied an index, and then use it directly. Otherwise, if the caller supplied an interface name, the index is obtained by calling if_nametoindex. Otherwise, the interface is set to 0, telling the kernel to choose the interface. The caller's socket address is copied directly into the request's group field. Recall that the group field is a sockaddr_storage, so it is big enough to handle any socket address type the system supports. However, to guard against buffer overruns caused by sloppy coding, check the sockaddr size and return EINVAL if it is too large. The setsockopt performs the join. The level argument to setsockopt is determined using the family of the group address and family_to_level function. Some systems support a mismatch between level and the socket's address family, for instance using IPPROTO_IP with MCAST_JOIN_GROUP, even with an AF_INET6 socket, but not all do, so it turns the address family into the appropriate level.

The program below shows the second third of mcast_join, which handles IPv4 sockets.

```
switch (grp->sa_family) {
case AF_INET:{
        struct ip_mreq mreq;
        struct ifreq ifreq;
        memcpy(&mreq.imr_multiaddr,
        &((const struct sockaddr_in *) grp)->sin_addr,
        sizeof(struct in_addr));
        if (ifindex > 0) {
        if (if_indextoname(ifindex, ifreq.ifr_name) == NULL) {
        errno = ENXIO; /* i/f index not found */
        return (-1);
}
goto doioctl;
} else if (ifname != NULL) {
        strncpy(ifreq.ifr_name, ifname, IFNAMSIZ);
        doioctl:
        if (ioctl(sockfd, SIOCGIFADDR, &ifreq) < 0)
        return (-1);
        memcpy(&mreq.imr_interface,
        &((struct sockaddr_in *) &ifreq.ifr_addr)->sin_addr,
        sizeof(struct in_addr));
} else
mreq.imr_interface.s_addr = htonl(INADDR_ANY);
return (setsockopt(sockfd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
&mreq, sizeof(mreq)));
```

In the program, the IPv4 multicast address in the socket address structure is c opied i nto a n i p_mreq structure. If a n i ndex is specified, if_indextoname i s c alled, s toring t he na me into i freq structure. If t his succeeds then branch a head is to is sue th e io ctl. The c aller's n ame i s copied into an ifreq structure, and an ioctl of SIOCGIFADDR returns the unicast address associated with this name. Upon success, the IPv4 address is c opied i nto t he imr_interface member of t he ip_mreq s tructure. If a n index is not specified and a name is not specified, the interface is set to the wildcard address, telling the kernel to choose the interface. The setsockopt performs the join.

The final portion of the function, which handles IPv6 sockets, is shown below.

```
#ifdef IPV6
case AF_INET6:{
        struct ipv6_mreq mreq6;
        memcpy(&mreq6.ipv6mr_multiaddr,
        &((const struct sockaddr_in6 *) grp) ->sin6_addr,
        sizeof(struct in6_addr));
        if (ifindex > 0) {
        mreq6.ipv6mr_interface = ifindex;
        } else if (ifname != NULL) {
        if ( (mreq6.ipv6mr_interface = if_nametoindex(ifname)) == 0) {
        errno = ENXIO; /* i/f name not found */
        return (-1);
}
} else
        mreq6.ipv6mr_interface = 0;
        return          (setsockopt(sockfd,             IPPROTO_IPV6,
       IPV6_JOIN_GROUP,
        &mreq6, sizeof(mreq6)));
}
#endif
        default:
        errno = EAFNOSUPPORT;
        return (-1);
}
#endif
}
```

In the program, first the IPv6 multicast address is copied from the socket address structure into the ipv6_mreq structure. If an index was specified, it is stored in the ipv6mr_interface member; if a name was specified, the index is obtained by calling if_nametoindex; otherwise, the interface index is set to 0 for setsockopt, telling the kernel to choose the interface. The group is joined.

Check Your Progress

1. *Can you explain the use of mcast_join and mcast_leave.*

2. *Can you explain the use of mcast_join_source_group*

13.9 dg_cli FUNCTION USING MULTICASTING

Modify dg_*cli* function by removing the call to *setsockopt.* Run a modified UDP echo server that joins the all-hosts group, and then run our program specifying the all hosts group as the destination address. We get a response from both the system on the subnet. They are each running the multicast echo server. Each reply is unicast because the source address of the request which is used by each server as the destination address of the reply is a unicast address.

13.10 RECEIVING IP MULTICAST INFRASTRUCTURE SESSION ANNOUNCEMENTS

The IP multicast infrastructure is the portion of the Internet with inter-domain multicast enabled. Multicast is not enabled on the entire Internet.

In order to receive a multimedia conference on the IP multicast infrastructure, a site needs to know only the multicast address of the conference and the UDP ports for the conference's data streams. The *Session Announcement Protocol (SAP),* describes the way this is done (the packet headers and frequency with which these announcements are multicast to the IP multicast infrastructure) and the *Session Description Protocol (SDP)*, describes the contents of these announcements (how the multicast addresses and UDP port numbers are specified). A site wishing to announce a session on the IP multicast infrastructure periodically sends a multicast packet containing a description of the session to a well-known multicast group and UDP port. Sites on the IP multicast infrastructure run a program named *sdr* to receive these announcements.

The below shows the main program to receive SAP/SDP announcements

```
#include "unp.h"
#define SAP_NAME "sap.mcast.net" /* default group name and port */
#define SAP_PORT  "9875"
void loop(int, socklen_t);
 int
main(int argc, char **argv)
 {
         int sockfd;
        const int on = 1;
        socklen_t salen;
        struct sockaddr *sa;
         if (argc == 1)
         sockfd = Udp_client(SAP_NAME, SAP_PORT, (void **) &sa,
        &salen);
         else if (argc == 4)
```

```
    sockfd = Udp_client(argv[1], argv[2], (void **) &sa, &salen);
    else
    err_quit("usage: m    ysdr <m    cast-addr> <p    ort#> <i    nterface-
   name>");
    Setsockopt(sockfd, S   OL_SOCKET, S   O_REUSEADDR, &  on,
   sizeof(on));
    Bind (sockfd, sa, salen);
    Mcast_join(sockfd, sa, salen, (argc == 4) ? argv[3] : NULL, 0);
    loop (sockfd, salen); /* receive and print */
    exit (0);
}
```

In the program, the multicast address assigned for SAP announcements is 224.2.127.254 and its name is sap.mcast.net. All the well-known multicast appears in the DNS under the mcast.net hierarchy. The well-known UDP port is 9875. In the program udp_ client function is call to look up the name and port, and it fills in the appropriate socket address structure. In the program set the SO_REUSEADDR socket option to allow multiple instances of this program to run on a host, and bind the port to the socket. By binding the multicast address to the socket, prevent the socket from receiving any other UDP datagrams that may be received for the port. After that mcast_join function is call to join the group. If the interface name is specified as a command-line argument, it is passed to function; otherwise, the kernel chooses the interface on which the group is joined. Lastly call loop function to read and print all the announcements.

13.11 SENDING AND RECEIVING

The program that sends and receives multicast datagrams consists of two parts. The first part sends a multicast datagram to a specific group every five seconds and the datagram contains the sender's hostname and process ID. The second part is an infinite loop that joins the multicast group to which the first part is sending and prints every received datagram (containing the hostname and process ID of the sender). This allows us to start the program on multiple hosts on a LAN and easily see which host is receiving datagrams from which sender.

The program below shows the main function of the program.

```
#include "unp.h"
void recv_all(int, socklen_t);
void send_all(int, SA *, socklen_t);
 int
 main (int argc, char **argv)
 {
 int sendfd, recvfd;
 const int on = 1;
 socklen_t salen;
 struct sockaddr *sasend, *sarecv;
 if (argc != 3)
 err_quit("usage: sendrecv <IP-multicast-address> <port#>");
 sendfd = Udp_client(argv[1], argv[2], (void **) &sasend, &salen);
 recvfd = Socket(sasend->sa_family, SOCK_DGRAM, 0);
 Setsockopt(recvfd, S      OL_SOCKET, S      O_REUSEADDR, &      on,
sizeof(on));
 sarecv = Malloc(salen);
 memcpy(sarecv, sasend, salen);
 Bind (recvfd, sarecv, salen);
 Mcast_join(recvfd, sasend, salen, NULL, 0);
 Mcast_set_loop(sendfd, 0);
 if (Fork () == 0)
 recv_all (recvfd, salen); /* child -> receives */
 send_all (sendfd, sasend, salen); /* parent -> sends */
```

In t he p rogram t wo s ockets i s created, on e f or s ending and on e f or receiving. T he receiving s ocket i s t o bi nd t he m ulticast gr oup and por t. Then t he receiving s ocket i s t o j oin t he m ulticast group. T he s ending socket will send datagrams to this same multicast address and port. But if we try to use a single socket for sending and receiving, the source protocol address is 239.255.1.2:8888 from the bind (using netstat notation) and the destination pr otocol a ddress f or t he s endto i s a lso 239.255.1.2: 8888. However, now t he s ource pr otocol a ddress t hat i s bound t o t he s ocket becomes t he s ource IP address o f t he UDP da tagram, a nd R FC 1122 forbids an IP datagram from having a source IP address that is a multicast address or a br oadcast a ddress. T he udp_c lient f unction c reates t he sending s ocket, pr ocessing the two c ommand-line a rguments th at specify the multicast address and port number. This function also returns a socket address structure that is ready for calls to sendto along with the length of this s ocket ad dress s tructure. T hen cr eate t he r eceiving s ocket u sing t he same a ddress f amily t hat w as us ed f or t he s ending s ocket. T hen s et t he SO_REUSEADDR s ocket o ption to a llow mu ltiple in stances o f th is program to run at the same time on a host. Then allocate room for a socket address structure for this socket, copy its contents from the sending socket address structure, and bind the multicast address and port to the receiving socket. After that call mcast_join function to join the multicast group on the receiving socket and mcast_set_loop function to disable the loopback feature on the sending socket. For the join, specify the interface name as a null pointer and the interface index as 0, telling the kernel to choose the interface. Lastly the fork and then the child is the receive loop and the parent is the send loop.

The program below shows the send a multicast datagram every five seconds.

```
 #include "unp.h"
 #include <sys/utsname.h>
 #define SENDRATE 5 /* send one datagram every five seconds */
 void
 send_all(int sendfd, SA *sadest, socklen_t salen)
 {
          char line[MAXLINE]; /* hostname and process ID */
          struct utsname myname;
         if (uname(&myname) < 0)
          err_sys("uname error");;
          snprintf(line, sizeof(line), " %s, %d \n", m yname.nodename,
         getpid());
          for ( ; ; ) {
          Sendto(sendfd, line, strlen(line), 0, sadest, salen);
          sleep(SENDRATE);
}
}
```

In t he p rogram send_all f unction, w hich s ends one m ulticast d atagram every f ive s econds. The m ain f unction pa sses a s a rguments t he s ocket descriptor, a pointer to a socket address structure containing the multicast destination and port, and the structure's length. In the program obtain the hostname from the uname function and build the output line containing it and t he pr ocess ID. S end da tagram, t hen go t o s leep. Then s end a datagram and then sleep for five seconds.

The program below shows that receive all multicast datagrams for a group we have joined.

```
#include "unp.h"
void
recv_all(int recvfd, socklen_t salen)
{
        int n;
        char line[MAXLINE + 1];
        socklen_t len;
        struct sockaddr *safrom;
        safrom = Malloc(salen);
         for ( ; ; ) {
         len = salen;
         n = Recvfrom(recvfd, line, MAXLINE, 0, safrom, &len);
         line[n] = 0; /* null terminate */
         printf("from %s: %s", Sock_ntop(safrom, len), line);
}
}
```

In t he program recv_all f unction, w hich i s t he infinite r eceive l oop. A socket ad dress s tructure i s al located t o r eceive t he s ender's p rotocol address f or ea ch c all t o r ecvfrom. E ach d atagram i s r ead b y r ecvfrom, null-terminated, and printed.

Check Your Progress

1. *Write a program to send and receive the multicast datagram with two sockets.*

2. *What happens if we create one socket for both sending and receiving.*

13.12 SIMPLE NETWORK TIME PROTOCOL (SNTP)

Simple N etwork T ime P rotocol (SNTP) is a simplified v ersion o f Network Time Protocol (NTP) that is used to synchronize computer clocks on a network. This simplified version of NTP is generally used when full implementation of NTP is not needed

SNTP is a simplified access strategy for servers and clients using NTP. SNTP synchronizes a computer's system time with a server that has already been synchronized by a source such as a radio, satellite receiver or modem.

SNTP supports unicast, multicast and anycast operating modes. In unicast mode, the client sends a request to a dedicated server by referencing its unicast address. Once a reply is received from the server, the client determines the time, roundtrip delay and local clock offset in reference to the server. In multicast mode, the server sends an unsolicited message to a dedicated IPv4 or IPv6 local broadcast address. Generally, a multicast client does not send any requests to the service because of the service disruption caused by unknown and untrusted multicast servers. The disruption can be avoided through an access control mechanism that allows a client to select a designated server he or she knows and trusts.

NTP is a sophisticated protocol for synchronizing clocks across a WAN or a LAN, and can often achieve millisecond accuracy. SNTP, a simplified but protocol-compatible version intended for hosts that do not need the complexity of a complete NTP implementation. It is common for a few hosts on a LAN to synchronize their clocks across the Internet to other NTP hosts and then redistribute this time on the LAN using either broadcasting or multicasting.

The below program shows the NTP packet format and definitions

```
#define JAN_1970              2208988800UL /* 1970 - 1900 in seconds */

struct l_fixedpt { /* 64-bit fixed-point */

                  uint32_t int_part;

                  uint32_t fraction;

                  };

struct s_fixedpt { /* 32-bit fixed-point */

         uint16_t int_part;

          uint16_t fraction;

          };

struct ntpdata { /* NTP header */

         u_char status;

          u_char stratum;

          u_char ppoll;

          int precision:8;
```

```
            struct s_fixedpt distance;
            struct s_fixedpt dispersion;
            uint32_t refid;
            struct l_fixedpt reftime;
            struct l_fixedpt org;
            struct l_fixedpt rec;
            struct l_fixedpt xmt;
            };
 #define VERSION_MASK 0x38
 #define MODE_MASK 0x07
 #define MODE_CLIENT 3
#define MODE_SERVER 4
#define MODE_BROADCAST 5
```

In the program, the 1_fixedpt defines the 64-bit fixed-point values used by NTP f or t imestamps and s _fixedpt de fines t he 32-bit f ixed-point va lues that are also used by NTP. The ntpdata structure is the 48-byte NTP packet format.

13.13 SUMMARY

A mu lticast a pplication s tarts b y jo ining th e mu lticast g roup assigned to the application. This tells the IP layer to join the group, which in turns tells the datalink layer to receive multicast frames that are sent to the corresponding hardware layer multicast address.

Multicasting on a WAN requires multicast-capable routers and a multicast routing protocol. Until all the routers on the Internet are multicast-capable, multicast is o nly a vailable to a subset of Internet us ers. The te rm "IP multicast infrastructure" is use to describe the set of all multicast-capable systems on the Internet.

Nine socket options provide the API for multicasting:

- Join an any-source multicast group on an interface
- Leave a multicast group
- Block a source from a joined group
- Unblock a blocked source
- Join a source-specific multicast group on an interface
- Leave a source-specific multicast group

- Set the default interface for outgoing multicasts
- Set the TTL or hop limit for outgoing multicasts
- Enable or disable loopback of multicasts

The first six are for receiving, and the last three are for sending.

13.14 TERMINAL QUESTIONS

1. Explain multicast address.

2. Write a short note on multicast socket option.

3. Explain the use of mcast_block_source.and mcast_unblock_source

4. Explain source specific multicast.

5. Write about SNTP in terms of multicast.

6. Write a program to show NTP packet format and definitions

UNIT-14 : INTER PROCESS COMMUNICATION

Structure

14.1 INTRODUCTION

In this unit, the different methods of IPC will be discussed. In this unit, we will learn about; File and record locking, Pipes, FIFOs streams and messages, name spaces, system IPC, message queues and semaphores.

A process is an active operating system entity which executes programs. Normally, a process, like a specialist, does one particular job (well). In real life, there are complex workflows and we, often have multiple processes collaborating to accomplish certain objectives. In order to work together, processes need to exchange data. So, we have various inter process communication (IPC) mechanisms.

The figure 14.1 shows the IPC between two processes on a single system. The information between the two processes going through the kernel. The figure 14.2 shows the IPC between two processes on different system.

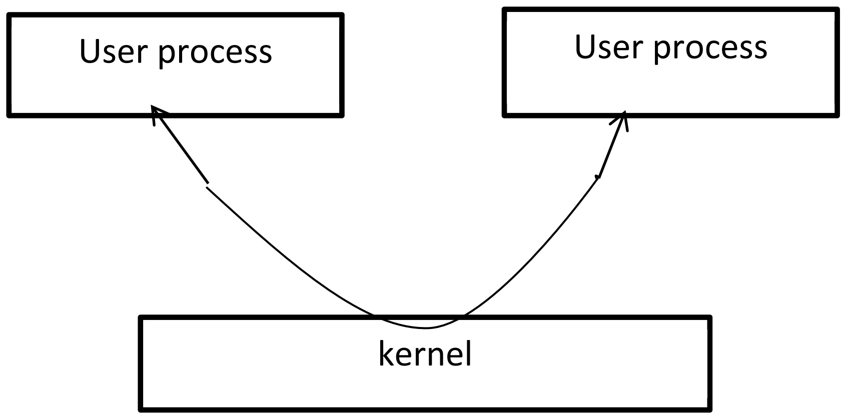


Figure 14.1: IPC between two processes on a single system

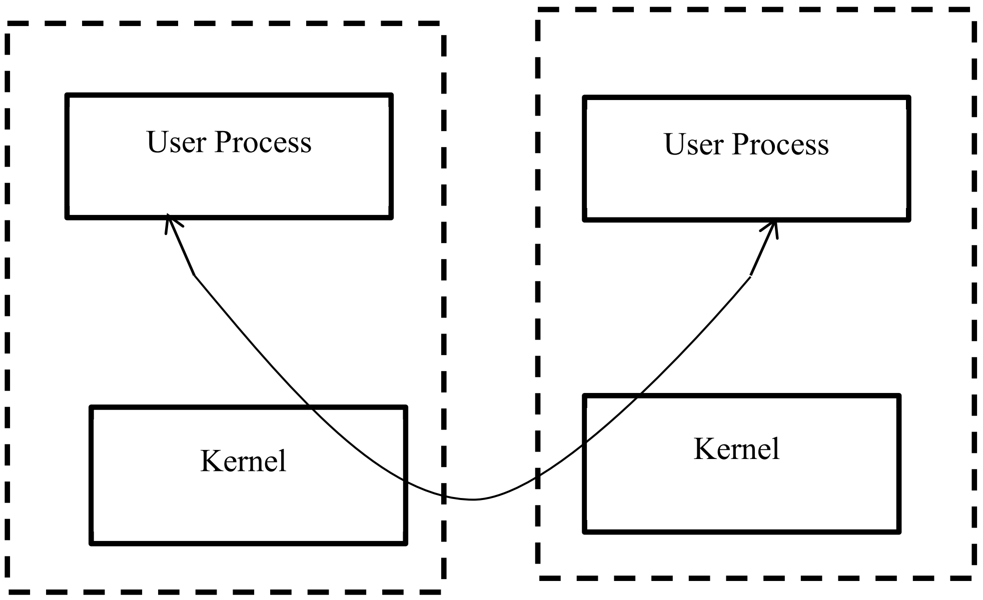


Figure 14.2: IPC between two processes on different system

14.2 OBJECTIVE

At the end of this unit, you should be able to know about: -

- Purpose of File and record locking
- How to use Pipes and FIFOs.
- Streams and messages, Message queues
- Name spaces, system IPC
- What is the use of Semaphores?

14.3 FILE AND RECORD LOCKING

When multiple process wants to share resource, it is essential that some form of mutual exclusion be provide so that only one process at a time can access the resource. The example is line printer daemon. The process that places a job on the print queue (to be printed at a later time by another process) has to assign a unique sequence number to each print job. Each process that needs to assign a sequence number goes through three steps:

- It reads the sequence number file
- It uses the number
- It increments the number and writes it back

The problem is that in the time it takes a single process can perform the same three steps; another process can perform the same three steps. The need is for a process to be locked so no other process can access the same file until the first process is done.

In file locking locks an entire file, while record locking allows a process to lock a specified portion of a file. Used to ensure that a process has exclusive access to a file before using it

#include int lockf (int fd, int function, long size);

fd---file descripter (not a file pointer)

size--- define the record size or lock area: [offset, offset + size]. size=0 means the rest

of the file. Use lseek() to move the current offset. When the offset position is

set to the beginning and size=0 then lock the whole file.

Function:

F_ULOCK---unlock a previous lock

F_LOCK ---lock a region(blocking)

F_TLOCK ---Test and lock a region(nonblocking)

F_TEST ---Test a region to see if it is locked.

Example:

Use F_TLOCK instead of F_TEST and F_LOCK.

If (lockf(fd, F_TEST, size)= =0) /* If the region is locked, -1 is returned and the

process is in sleep state*/

Re= lockf(fd, F _LOCK, size); / *a small chance that another process locks between

TEST and LOCK*/

```
rc=lockf(fd, F_TLOCK, size) /* Test + lock done as an atomic operation, If
                                   unsuccessful, lockf() returns -1 and the calling
                                   process continues to do other things*/
```

The following are the two types of Linux file locking:

1. Advisory locking
2. Mandatory locking

1. Advisory Locking

Advisory locking requires cooperation from the participating processes. Suppose process "A" acquires a WRITE lock, and it started writing in to the file, and process "B", without trying to acquire a lock, it can open the file and write into it. Here process "B" is the non-cooperating process. If process "B", tries to acquire a lock, then it means this process is co-operating to ensure the "serialization". Advisory locking will work, only if the participating processes are cooperative. Advisory locking sometimes also called as "unenforced" locking.

Posix record locking is called advisory locking. This means the kernel maintains correct knowledge of all files that have been locked by each process, but it does not prevent a process from writing to a file that is read-locked by another process. Similarly, the kernel does not prevent a process from reading from a file that is write-locked by another process. A process can ignore an advisory lock and write to a file that is read-locked, or read from a file that is write-locked, assuming the process has adequate permissions to read or write the file. Advisory locks are fine for cooperating processes.

2. Mandatory Locking

Mandatory locking doesn't require cooperation from the participating processes. Mandatory locking causes the kernel to check every open, read, and write to verify that the calling process isn't violating a lock on the given file.

Check Your Progress

1. *Write the different between advisory locking with mandatory locking.*
2. *Can you test that whether a region is locked or not.*

14.4 PIPES

A pi pe pr ovides a on e-way f low o f da ta. Two p rocesses can be joined b y t he pi pe s ymbol (|) on t he s hell c ommand l ine. T he s tandard output of t he f irst pr ocess be comes t he s tandard i nput f or t he s econd process. For example,

```
$ ls -ls | more
```

Example a pipe provides a one-way flow of data.

```
int pipe (int * filedes);
```

```
int pipefd[2]; /* pipefd[0] is opened for reading; pipefd[1] is opened for
writing */
```

The program below shows how to create and use a pipe:

```
main ()
{
int pipefd[2], n;
 char buff[100];
if (pipe(pipefd) < 0 ) err_sys("pipe error");
printf("read fd = %d, write fd = %d\n", pipefd[0], pipefd[1]);
if (write(pipefd[1], "hello world\n", 12) != 12) err_sys("write
error");
if  ((n=read(pipefd[0], buf f, s izeof(buff)))  < =  0)  err_sys("read
error");
write (1, buff, n); /*fd=1=stdout*/
}
```

Result:
```
hello world
read fd=3, write df =4
```

Properties of Pipe:

- Pipes do not have a name. For this reason, the processes must share a pa rent pr ocess. T his i s t he m ain dr awback t o pi pes. H owever, pipes are treated as file descriptors, so the pipes remain open even after fork and exec.
- Pipes do not distinguish between messages; they just read a fixed number of bytes. Newline (\n) can be used to separate messages. A structure w ith a le ngth field c an be us ed f or m essage c ontaining binary data.
- Pipes c an a lso be us ed t o g et t he out put of a c ommand or t o provide input to a command

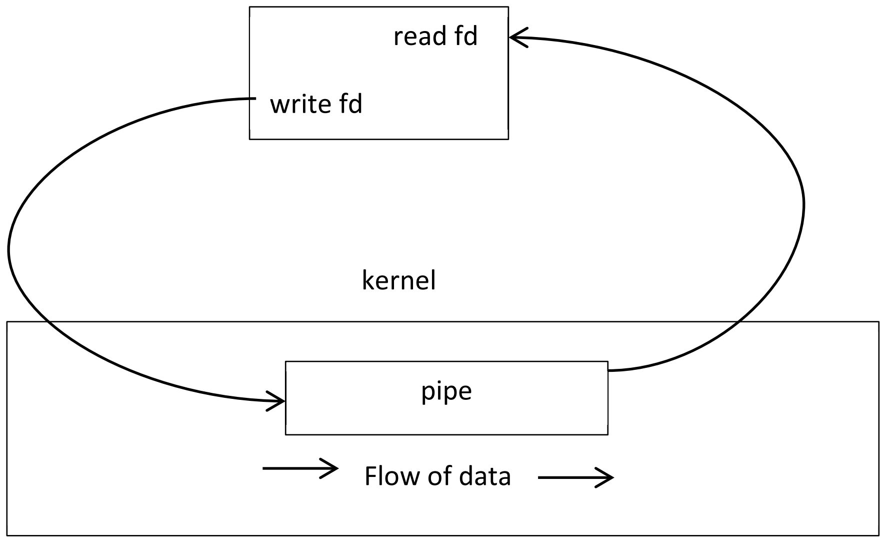


Figure 14.3: Pipe in a single process

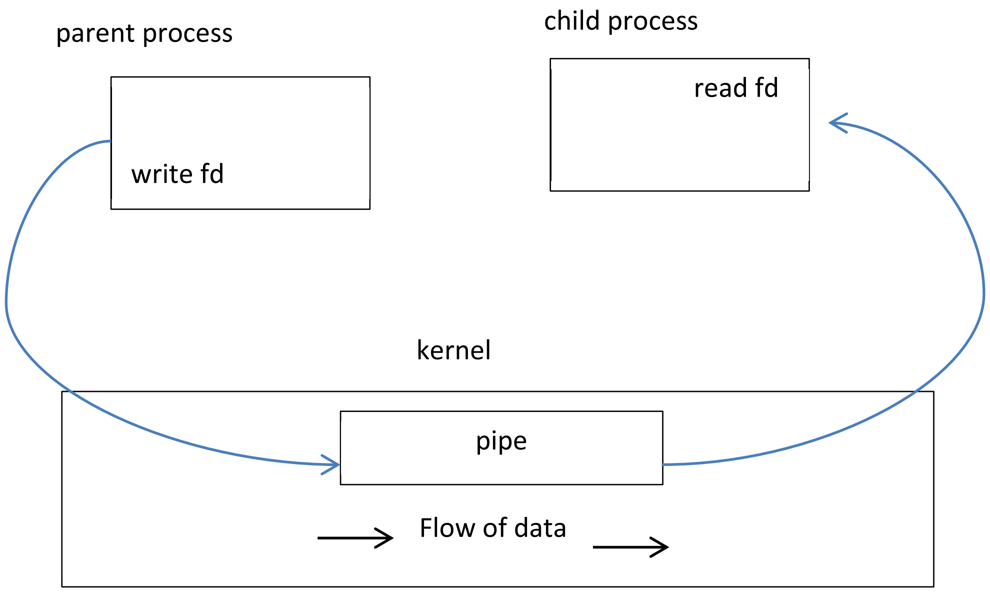


Figure 14.4: Pipe between two processes

The figure 14.3 shows the pipe in a single process and figure 14.4 shows the pipe between two processes.

One major feature of pipe is that the data flowing through the communication medium is transient, that is, data once read from the read descriptor cannot be read again. Also, if we write data continuously into the write descriptor, then we will be able to read the data only in the order in which the data was written. One can experiment with that by doing successive writes or reads to the respective descriptors.

14.5 FIFO

A FIFO is similar to a pipe. A FIFO (First in First Out) is a one-way flow of data. FIFOs have a name, so unrelated processes can share the FIFO. FIFO is a named pipe. This is the main difference between pipes and FIFOs. Another major difference between FIFOs and pipes is that FIFOs last throughout the life-cycle of the system, while pipes last only during the life-cycle of the process in which they were created. To make it more clearly, FIFOs exist beyond the life of the process. Since they are identified by the file system, they remain in the hierarchy until explicitly removed using unlink, but pipes are inherited only by related processes, that is, processes which are descendants of a single process.

Create: A FIFO is created by the mkfifo function:

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

pathname – a UNIX pathname (path and filename).

mode – the file permission bits.

FIFO can also be created by the mknod system call, e.g., mknod(“fifo1”, S_IFIFO|0666, 0) is same as mkfifo(“fifo1”, 0666).

Open: mkfifo tries to create a new FIFO. If the FIFO already exists, then an EEXIST error is returned. To open an existing FIFO, use open (), fopen () or freopen ()

Close: to close an open FIFO, use close (). To delete a created FIFO, use unlink ().

The table 14.1 shows the Effect of O_NDELAY flag on pipes and FIFOs. A pipe or FIFO follows these rules for reading and writing:

- A read requesting less data than is in the pipe or FIFO returns only the requested amount of data.
- If a process asks to read more data than is currently available in the pipe FIFO, Only the data available is returned. The process must be prepared to handle a return value from read that is less than the requested amount.
- If there is no data in the pipe or FIFO, and if no processes have it open for writing, a read return zero, signifying the end of file. If the reader has specified O_NDELAY, it can not tell if a return value of zero means there is no data currently available or if there are no writers left.

Condition	Normal	O_NDELAY set
Open FIFO, read-only with no process having the FIFO open for writing	for Wait until a process opens the FIFO writing	Return immediately, no error
Open FIFO, write-only with no process having the FIFO open for reading	Wait until a process opens the FIFO for reading	Return an error immediately, errno set to ENXIO
read pipe or FIFO, no data	Wait until there is data in the pipe or FIFO, or until no processes have it open for writing; return a value of zero if no processes have it open for writing, otherwise return the count of data	Return immediately, return value of zero
Write, pipe or FIFO is full	Wait until space is available, then write data	return immediately, return value of zero

Table 14.1: Effect of O_NDELAY flag on pipes and FIFOs.

- If a process writes less than the capacity of a pipe (which is at least 4096 bytes) the write is guaranteed to be atomic. This means that if two processes each write to a pipe or FIFO at about the same time, either all the data from the first process is written, followed by all the data from the second process, or vice versa. The system does not mix the data from the two processes-i.e., part of the data from one process, followed by part of the data from the other process. If, however, the write specifies more data than the pipe can hold, there is no guarantee that the write operation is atomic.

- If a process writes to a pipe or FIFO, but there are no processes in existence that have it open for reading, the SIGPIPE signal is generated, and the write returns zero with errno set to EPIPE. If the process has not called signal to handle the SIGPIPE notification, the default action is to terminate the SIGPIPE signal, or if it handles the signal and returns from its signal handler.

14.6 STREAMS AND MESSAGES

A STREAM is a general, flexible programming model for UNIX system communication services. STREAMS define standard interfaces for character input/output (I/O) within the kernel, and between the kernel and the rest of the UNIX system. The mechanism consists of a set of system calls, kernel resources, and kernel routines.

A STREAM enables you to create modules to provide standard data communications services and then manipulate the modules on a stream. From the application level, modules can be dynamically selected and interconnected. No kernel programming, compiling, and link editing are required to create the interconnection.

A STREAM provides an effective environment for kernel services and drivers requiring modularity. STREAMS parallel the layering model found in networking protocols. For example, STREAMS are suitable for:

- Implementing network protocols
- Developing character device drivers
- Developing network controllers (for example, for an Ethernet card)
- I/O terminal services

In STREAMS, all information is exchanged via messages i.e., both data and control messages of various priorities. A multi-component message structure is used to reduce the overhead of

1. Memory-to-memory copying i.e., via reference counting
2. Encapsulation/de-encapsulation i.e., via composite messages.

Messages may be queued at STREAM modules. Many Unix processes that need to impose a message structure on top of a stream based IPC facility. More structured message can also be built, and this is what the Unix message queue form of IPC does. We can also add more structure to either a pipe or FIFO. We define a message in mesg.h header file as

```
/*
    *Definition of "our" message.
```

```
* You may have to change the 4096 to a smaller value, if message
*queues on   your s ystem w ere  configured w ith “m sgmax” l ess
*than 4096.
 */
# define MAXMESGDATA      (4096-16)
                                                /* w   e don’  t    want
sizeof(Mesg) > 4096 */
#define MESGHDRSIZE     (sizeof(Mesg) – MAXMESGDATA)
                                                  /* l      ength of
mesg_len and mesg_type*/
typedef struct {
int mesg_len;  /*#bytesin mesg_data, can be 0 or > 0 */
long mesg_type; /* message type, must be > 0 */
char mesg_data [MAXMESGDATA];
} Mesg;
```

Check Your Progress:

1. *Can you write a program that create FIFO in which it writes first then read?*
2. *How stream and message are useful in Unix?*

14.7 NAME SPACES

The set of possible names for a given type of IPC is called its name space. The name space is important because all forms of IPC other than plain pipes, the name is how the client and server connected to exchange message. The table 14.2 shows the list of available name space below.

IPC type	**Name Space**	**Identification**
pipe	No name	File descriptor
fifo	Path name	File descriptor
message queue	Key_t key	identifier
shared memory	Key_t key	identifier
semaphore	Key_t key	identifier
socket-unix domain	Path name	File descriptor
socket-other domains	Domain depndent	File descriptor

Table 14.2 : List of available name space

14.8 SYSTEM IPC

The three types of IPC

- Message queues
- Semaphores
- Shared memory

These are collectively referred as "system V IPC"

Linux supports three types of interprocess communication mechanisms that first appeared in Unix System V (1983). These are message queues, semaphores and shared memory. These System V IPC mechanisms all share common authentication methods. Processes may access these resources only by passing a unique reference identifier to the kernel via system calls. Access to these System V IPC objects is checked using access permissions; much like accesses to files are checked. The access rights to the System V IPC object is set by the creator of the object via system calls. The object's reference identifier is used by each mechanism as an index into a table of resources. It is not a straight forward index but requires some manipulation to generate the index. All Linux data structures representing System V IPC objects in the system include an ipc_perm structure which contains the owner and creator process's user and group identifiers. The access mode for this object (owner, group and other) and the IPC object's key. The key is used as a way of locating the System V IPC object's reference identifier. Two sets of keys are supported: public and private. If the key is public then any process in the system, subject to rights checking, can find the reference identifier for the System V IPC object. System V IPC objects can never be referenced with a key, only by their reference identifier.

A summary of their system calls is shown in table 14.3.

	Message queue	**Semaphore**	**Shared memory**
Include file	<sys/msg.h>	<sys/sem.h>	<sys/shm.h>
System call to create or open	msgget	semget	shmget
System call for control operations	msgctl	semctl	shmct1
System calls for IPC operations	msgsnd	semop	shmat
	msgrcv		shmdt

Table 14.3: Summary of system V IPC system calls

The value returned by msgget is the message queue identifier, msqid, or -1 if an error occurred.

14.9 MESSAGE QUEUES

Message q ueues al low o ne o r m ore p rocesses t o w rite m essages, which will be read by one or more reading processes. Linux maintains a list o f m essage q ueues, in the m sgque vector; each el ement o f w hich points t o an msqid_ds da ta s tructure t hat f ully de scribes t he m essage queue. When message queues are created a new msqid_ds data structure is allocated f rom s ystem memory and in serted in to th e v ector. For e very message query in the system, the kernel maintains the following structure of information

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

#include<sys/ipc.h>

struct msqid_ds{

        struct ipc_perm msg_perm;

        struct msg     *msg_first;

        struct msg     *msg_last;

        ushort    msg_cbytes;

        ushort    msg_qnum;

        ushort    msg_qbytes;

        ushort    msg_lspid;

        ushort     msg_lrpid;

        time_t     msg_stime;

        time_t    msg_rtime;

        time_t    msg_ctime;

};
```

A new message query is created or an existing message queue is accessed with the msgget system call

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

#include    <sys/ipc.h>

#include    <sys/msg.h>

int msgget (key_t key,int msgflag);
```

The msgflag value is a combination of constants shown in table 14.4.

Numeric	Symbolic	Description
0400	MSG_R	Read by owner
0200	MSG_W	Write by owner
0040	MSG_R >> 3	Read by group
0020	MSG_W >>3	Write by group
0004	MSG_R >>6	Read by world
0002	MSG_W>>6	Write by world
	IPC_CREAT	Create new entry
	IPC_EXCL	Create new entry

Table 14.4: msgflag values for msgget system call.

Each msqid_ds data structure contains an ipc_perm data structure and pointers to the messages entered onto this queue. In addition, Linux keeps queue modification times such as the last time that this queue was written to and so on. The msqid_ds also contain two wait queues; one for the writers to the queue and one for the readers of the message queue. Each time a process attempts to write a message to the write queue its effective user and group identifiers are compared with the mode in this queue's ipc_perm data structure. If the process can write to the queue then the message may be copied from the process's address space into an msg data structure and put at the end of this message queue. Each message is tagged with an application specific type, agreed between the co operating processes.

However, there may be no room for the message as Linux restricts the number and length of messages that can be written. In this case the process will be added to this message queue's write wait queue and the scheduler will be called to select a new process to run. It will be woken up when one or more messages have been read from this message queue. Reading from the queue is a similar process. Again, the processes access rights to the write queue are checked. A reading process may choose to either get the first message in the queue regardless of its type or select messages with particular types. If no messages match these criteria the reading process will be added to the message queue's read wait queue and the scheduler run. When a new message is written to the queue this process will be woken up and run again.

14.10 SEMAPHORES

Semaphores are synchronization primitive. The main use of semaphores is to synchronize the access to shared memory segments. In its simplest form, a semaphore is a location in memory whose value can be tested and set by more than one process. Semaphores can be used to implement critical regions, areas of critical code that only one process at a time should be executing.

The following information is related to semaphore:

1. The semaphore is stored in the kernel:

   a. Allows atomic operations on the semaphore.

   b. Processes are prevented from indirectly modifying the value.

2. A process acquires the semaphore if it has a value of zero. The value of the semaphore is then incremented to 1. When a process releases the semaphore, the value of the semaphore is decremented.

3. If the semaphore has non-zero value when a process tries to acquire it, that process blocks.

4. In 2 and 3, the semaphore acts as a customer counter. In most cases, it is a resource counter.

5. When a process waits for a semaphore, the kernel puts the process "to sleep" until the semaphore is available. This is better (more efficient) than busy waiting such as TEST & SET.

6. The kernel maintains information on each semaphore internally, using a data structure struct semis_ds that keeps track of permission, number of semaphores, etc.

7. Apparently, a semaphore in Unix is not a single binary value, but a set of nonnegative integer values.

8. There are 3 (logical) types of semaphores:

- Binary semaphore – have a value of 0 or 1. Similar to a mutex lock. 0 means locked; 1 means unlocked.
- Counting semaphore – has a value ≥ 0. Used for counting resources, like the producer-consumer example. Note that value =0 is similar to a lock (resource not available).
- Set of counting semaphores – one or more semaphores, each of which is a counting semaphore.

9. There are 2 basic operations performed with semaphores:

- Wait – waits until the semaphore is > 0, then decrements it.
- Post – increments the semaphore, which wakes waiting processes.

Operations on a semaphore are performed using:

int semop(int *semid*, struct sembuf **opsptr*, unsigned int *nops*)

semid — value returned by semget.

nops — # of operations to perform, or the number of elements in the *opsptr* array.

opsptr — points to an array of one or more operations. Each operation is defined as:

```
struct sembuf { ushort sem_num; /* semaphore #, numbered from 0, 1, 2 … */
        short sem_op; /* semaphore operation */
        short sem_flg; /*operations flags, such as 0, IPC_NOWAIT for nonblocking call,
        or SEM_UNDO to have the semaphore automatically released when the process is terminated prematurely. */
        };
```

sem_op = 0 – wait until the semaphore is 0. IPC_NOWAIT causes an error if semval≠0.

sem_op > 0 – increment the semaphore value: semval + sem_op, (acquire)

sem_op < 0 – wait until the semaphore value≥|sem_op| and decrement the semaphore value: semval - |sem_op|, (release)

Example: How to write lock/unlock (somewhat like P/V operations)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#define SEMKEY 123456L /* key value for semget() */
#define PERMS 0666
static struct sembuf op_lock[2]= { 0, 0, 0, /* wait for sem #0 to become 0 */
                        0, 1, SEM_UNDO /* then increment sem #0 by 1 */ };
static struct sembuf op_unlock[1]= { 0,    -1, ( IPC_NOWAIT | SEM_UNDO)
                        /* decrement sem #0 by 1 (sets it to 0) */ };
int semid = -1; /* semaphore id . Only the first time will create a semaphore. */
my_lock( )
{       if (semid <0) {
        if ( ( semid=semget(SEMKEY, 1, IPC_CREAT | PERMS )) < 0 ) printf("semget error"); }
        if (semop(semid, &op_lock[0], 2) < 0) printf("semop lock error");
}
my_unlock( )
{
        if ( semop(semid, & op_unlock[0], 1) < 0) pr intf("semop unl ock
error");
}
```

The semaphore has used for synchronization. The binary semaphore has created, a single semaphore value that is either zero or one. For locking the semaphore call *semop( )* to do operations automatically. First, wait for the semaphore value to become zero, and then increment the value to one. This is an example where multiple semaphore operation must be done atomically by the kernel. If it took two system calls to do this, one to test the value and wait for it to become zero, and another to increment the value, the operations would not work. For unlocking the resource, s*emop( )* will call to decrement the semaphore value. Since we have lock on the resource, we know that the semaphore value is one before the call, so the call cannot wait.

14.11 SUMMARY

IPC has traditionally been a massive area in UNIX. In this unit, we have covered record locking and file locking since the sharing of single file between multiple processes is a common occurrence. Various IPC techniques like PIPES, FIFO, Message queues, semaphores and shared memory are covered.

14.12 TERMINAL QUESTIONS

1. What is a signal generated for the writer of a pipe of FIFO when the other end disappears, and for the reader of a PIPE or FIFO when its writer disappears?

2. What happens with the client server example using message queues if the file to be copied is a binary file?

3. What happens to the version that uses the popen function if the file is a binary file?

4. What is the use of semaphore?

5. Can you design a message in mesg.h header file?

6. List few of the available name spaces.

7. Write a program to lock and unlock a semaphore.

UNIT-15 : REMOTE LOGIN

Structure

15.1 INTRODUCTION

In this unit, we will learn details about rlogin (remote login). We will study Terminal line disciplines. Then we get knowledge about Pseudo- Terminal. Then we see the terminal modes and control terminal. Lastly, we will discuss the transparent issues in RPC.

rlogin (remote login) is a UNIX command that allows an authorized user to login to other UNIX machines (hosts) on a network and to interact as if the user were physically at the host computer. Once logged in to the host, the user can do anything that the host has given permission for, such as read, edit, or delete files.

15.2 OBJECTIVES

In this unit, we will understand the following:

- Terminal line disciplines
- Pseudo- Terminal and Terminal modes
- Control Terminal
- rlogin overview
- RPC Transparency Issues

15.3 TERMINAL LINE DISCIPLINES

Terminal drivers are complicated by the line discipline associated with their Terminal. It is assumed to be a full duplex device so that the input path and output path are separate. The line discipline is within the kernel, somewhere between the actual device driver and the user process. The terminal line discipline is just a module that is pushed onto a stream on top of the actual terminal device driver. Figure 15.1 shows a normal interactive shell showing terminal line discipline.

There are several functions that can be done by a line discipline mode.

- Echo the characters entered into lines
- Assemble the characters entered into lines, so that a process reading from the terminal receives complete lines.
- Edit the lines that are input. UNIX allows you to erase the preceding character and also to kill the entire line being input and start over with a new line.
- Generate signals when certain terminal keys are entered. The SIGINT and SIGQUIT signals can be generated this way, for example.
- Process flow control characters. For Example, when you press the control –S key, the output to the terminal is stopped. The restart the output, the Control-Q key is entered.
- Allow you to enter an end –of-file character.
- Do character conversions

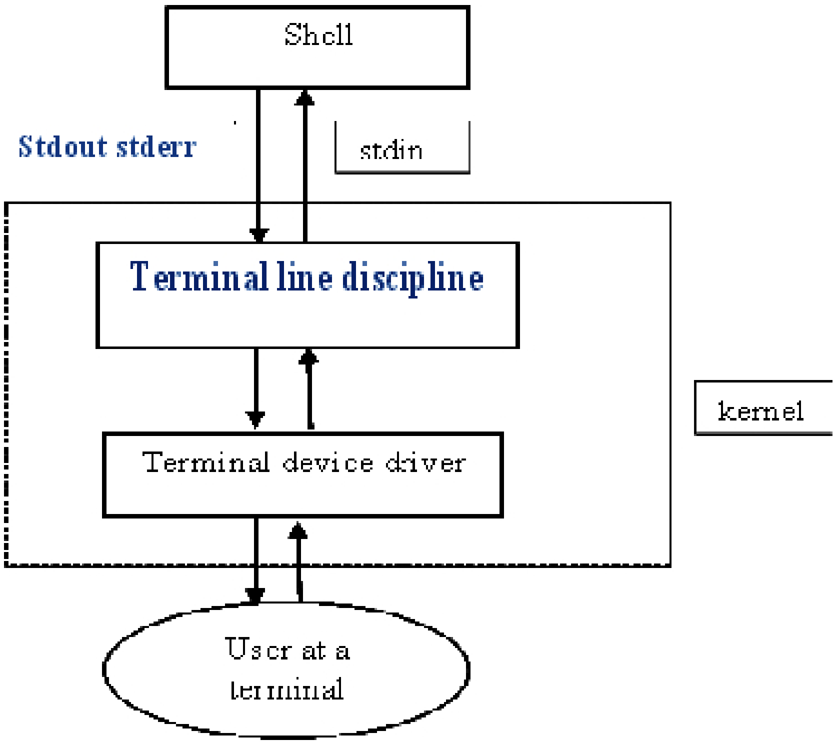


Figure 15.1: Normal interactive shell showing terminal line discipline

There are many versions of the line discipline modules. For example, BSD supplies five modules.

- The "old d iscipline" th at is s imilar to th e v ersions 7 U NIX terminal handler.
- The new discipline is a superset of the old discipline.
- It provides the features needed for job control along with enhanced editing capabilities.
- The Berknet line discipline.
- The s erial Line Internet P rotocol c an be us ed t o t ransfer IP datagrams across serial lines.

Check Your Progress

1. *Can you explain the different function in terminal line discipline mode?*

15.4 PSEUDO- TERMINAL

A pseudo-terminal is pair of devices. One half is called the master and the other half is called the slave. A process opens a p air o f p seudo-terminal de vices and gets t wo f ile de scriptors. T he s lave po rtion of pseudo-terminal devices gets two file descriptors. The slave portion of a pseudo-terminal presents an interface to the user process that looks like a terminal device.

A ps eudo-terminal is mainly u sed to ma ke a p rocess b elieve th at it interacts with a terminal although it a ctually interacts with one or more processes. The figure 15.2 s hows the Pseudo-terminals as they ar e u sed by script.

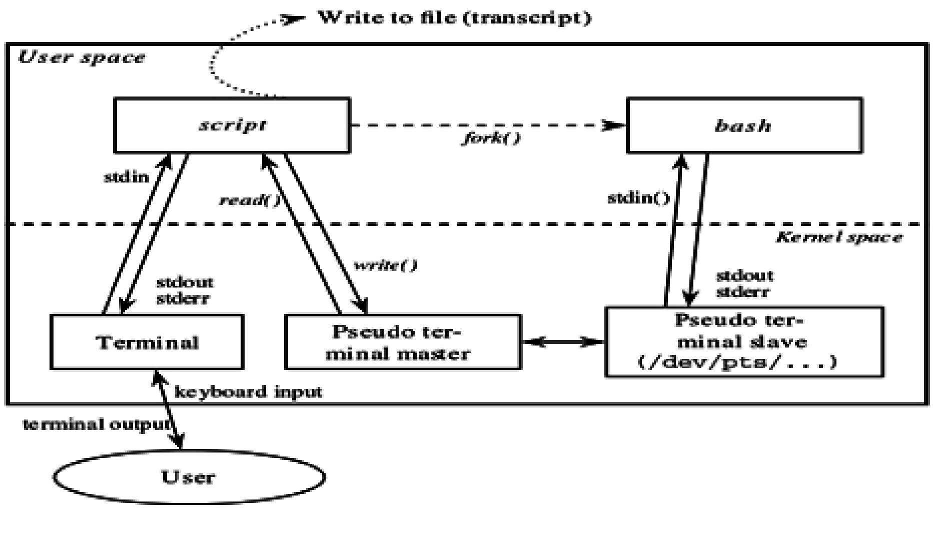


Figure 15.2 : Pseudo-terminals

15.5 TERMINAL MODES

In terminal models we are considering only standard terminal line discipline modules such as old line discipline and the new line discipline modules supported by 4.3BSD.

4.3BSD considers a terminal device in one of three modes.

- **Cooked mode** provides a ll t he pr ocessing s teps. T he i nput i s collected i nto l ines an d all s pecial ch aracter p rocessing i s d one. This is normal mode for interactive use.
- **Raw mode** lets t he p rocess r eceive every charter as i n i s i nput, with no i nterpretation do ne b y t he s ystem. Raw mode i s us ed for example by full screen editors such as vi and also by programs that use a serial line something other than interactive use.
- **Cbreak mode** is somewhere between cooked mode and raw mode. The cbreak mode provides character at a time input to the process reading from the terminal, instead of collecting the input into lines. The signal generating keys are still processed; however the editing features are disabled.

15.6 CONTROL TERMINAL

In 4.3BSD we have the child process from the fork dissociate from its c ontrol te rminal b efore it o pens th e p seudo-terminal s lave d evice. When the slave is opened it becomes the control terminal. Since we only want t he n ew s hell p rocess t hat t he ch ild p rocess ex ecs t o d isassociate from its c ontrol te rminal- we do not w ant t he r ecording pr ocess t hat i s reading f rom your actual te rminal to d o th is- we m ust do t his i n c hild process and not i n t he parent. T his i s pr ecisely w hy t he ope ning o f a pseudo-terminal pair into pieces. We do not want to open the slave device until we are in the child process.

15.7 RLOGIN OVERVIEW

The terminal line discipline on the local system is placed into the raw m ode w ith e choing disabled b y t he r login c lient pr ocess, s o t hat a ll keystrokes are passed to the remote system. The raw mode is required to run pr ograms s uch as t he vi e ditor on t he r emote s ystem. In t he nor mal UNIX fashion ch aracters t hat ar e en tered o n t he local ar e e choed b y t he remote system. If the remote system is in a cooked mode then the echoing is done by the terminal line disciplines on the remote system. If the remote system is in a raw mode then the echoing is done by that remote process itself. The figure 15.3 shows the 4.3BSD rlogin processes.

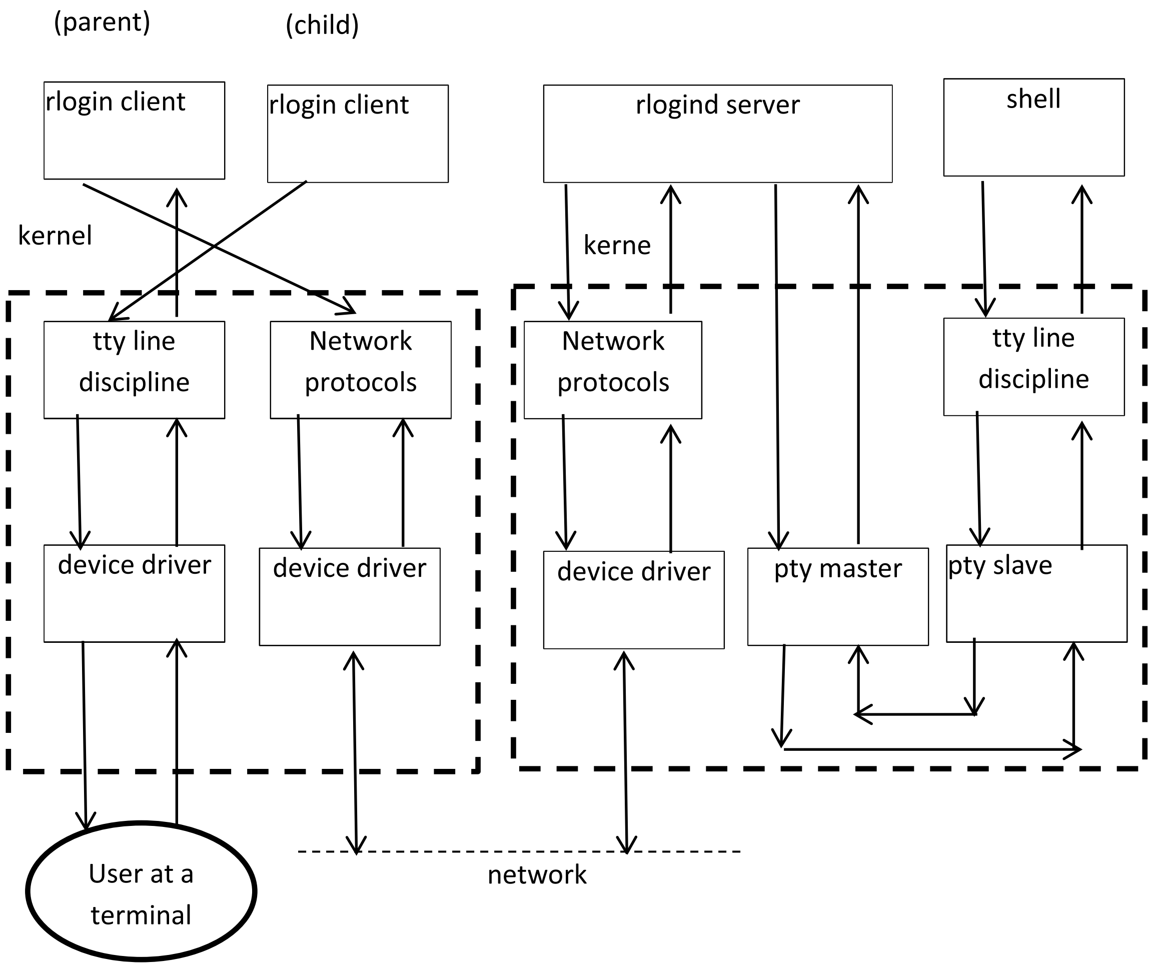


Figure 15.3 : 4.3BSD rlogin processes

The rlogin facility provides a remote-echoed, locally flow-controlled virtual terminal with proper flushing of output. It is widely used between UNIX hosts because it provides transport of more of the UNIX terminal environment semantics than does the Telnet protocol, and because on many UNIX hosts it can be configured not to require user entry of passwords when connections originate from trusted hosts.

Apart from this, rlogin suffers most of the same security disadvantages as telnet, such as the fact that all communication, including passwords, is transmitted in clear-text. The trusted hosts feature bypasses password authentication when an rlogin/rhosts-file is specified. This poses a great security risk as the files themselves are not very well secured, and in many cases, can be found on the host's NFS share. Because of these problems, rlogin is not in much use today and has mostly been replaced by the superior SSH protocol.

Check Your Progress

1. *What is the role of pseudo terminal?*
2. *Can you explain the major steps for 4.3BSD rlogin process?*

15.8 RPC TRANSPARENCY ISSUES

The system needs to provide a transparent interface for the client, so that there is no distinction between making a remote procedure call and making a local function call.

The client and server stubs hide the network code, but there are other issues that need to be addressed:

Parameter Passing – can't pass parameters by reference, since the subroutine and the calling program don't share the same address space. Sun RPC allows only a single argument and a single result. A structure is required for multiple values.

Binding – the client needs some way to determine which host is a server. Choices are to require that the client knows which host to contact, or uses super server that keeps track of the addresses of each server, or use a centralized database where each host indicates which servers it is willing to run. Sun RPC takes the following approach. The port mapper on the remote host is contacted for the port number. The port mapper also accepts the broadcast requests. If a matching server is found, the request is passed to the server. The port number is then returned with the results, so the client can be connected directly to the server on future calls.

Transport Protocol – Sun RPC supports TCP and UDP. TCP is a byte-stream protocol, so there are no message delimiters. To solve this, a 32-bit integer giving the number of bytes is placed at the beginning of each record. With UDP on older systems, at most 8192 bytes can be sent for the arguments or results of one call. The maximum can never exceed the size of a UDP datagram, which is 64 K – headers.

Exception Handling – not only could the typical errors, such as segmentation fault, occur in the remote procedure, but also network problems are also possible. A timeout is usually used to detect server crashes.

The client might also wish to terminate the server. With Sun RPC, the client cannot send an interrupt to the server. Both UDP and TCP handle timeouts and retransmissions. UDP will terminate after some number of unsuccessful attempts.

Call Semantics – because of network problems, the request to start a remote procedure might be sent multiple times. Procedures that can be executed multiple times without a problem are called idempotent. Examples include computing a square root or checking an account balance.

There are three different forms of RPC semantics

1. Exactly once-Means that the remote procedure was executed one time period. This type of operation is hard to achieve, owning to the possibility of server crashes.
2. At most once –Means that the remote procedure was either not executed at all or it was executed one time at most. If a normal return is made to the caller, we know the remote procedure was executed one time. But if an error return is made, we are not certain if remote procedure was executed once or not at all.
3. At least once-Means that the remote procedure was executed at least one time, but perhaps more. This is typically for idempotent procedures-the client keeps transmitting its request until it receives a valid response. But if the client has to send its request more than once to receive a valid response, there is a possibility that the remote procedure was executed more than once.

Data Representation – Need a standard representation, so the client and server can execute on different architectures. Sun RPC uses the XDR data representation standard.

Performance – there can be considerable overhead for calling an RPC. For example, the overhead might be 100 times the overhead of a local procedure call. Sun RPC uses several mechanisms, such as passing pointers, to minimize coping data. The purpose of RPC is to simplify network programming, not just to replace LPC with RPC.

Security – May need to restrict who can execute a program on the server. In the local case the caller of a function can be sure that it is calling an authorised provider of the service, and the procedure can be sure it is called by an authorised user, because they are linked together at compile time. With a remote call, neither party can be sure.

To assure clients and server that they are talking to authorised servers and clients, Sun RPC includes an authentication mechanism. The client sends its credentials and a verifier to the server with its RPC call, the server returns its own verifier with the results. The standard authentication methods provided by the library are Null, UNIX, Short and DES, but it is easy to add new methods. Using the authentication mechanisms is not transparent; it requires some extra programming on the client and server sides.

15.9 SUMMARY

Remote Login is comparatively complicated networking example, which we have discussed. The most complicated part of remote login is terminal handling. Also, Users want remote login to be as simple as local login. In this chapter 4.3BSD rlogin client and server was described in a step wise mode. First a recording process was developed to understand the terminal line disciples and pseudo terminals.

15.10 TERMINAL QUESTIONS

1. Why RPC not pass parameters by reference?
2. Explain how Sun RPC maintains at-most-once semantics?
3. What are different terminal modes?
4. What are the different transparency issues with RPC?
5. Write short note on (a) Pseudo-Terminal (b) rlogin
6. What are the different forms of RPC semantics?