CHAPTER W20

System Programming III: Interprocess Communication

Objectives

- To explain the concept of interprocess communication (IPC), important related system calls, macros, and data structures
- To describe IPC based on the client–server model
- To discuss the most primitive form of IPC between a parent process and its children processes
- To discuss IPC between related processes on a machine using Linux pipes
- To describe IPC between unrelated processes on the same machine using Linux named pipes (FIFOs)
- To discuss IPC between related or unrelated processes on the same machine or different machines on a network using sockets
- To elucidate synchronous and asynchronous IPC
- To discuss in detail the client–server models for socket-based IPC
- To describe in detail the design of client–server software and possible types of servers
- To study multiple-process-based concurrent servers
- To explain concurrent servers based on the `select()` system call
- To explain the Linux superserver, `inetd`
- To briefly discuss concurrent clients
- To cover the system calls, library calls, commands, macros, and primitives `_exit()`, `FD_CLR()`, `FD_ISSET()`, `FD_SET()`, `FD_ZERO()`, `accept()`, `bind()`, `bzero()`, `close()`, `connect()`, `exit()`, `gethostbyaddr()`, `gethostbyname()`, `getservent()`, `gettablesize()`, `htonl()`, `htons()`, `inet_addr()`, `inet_aton()`, `inet_ntoa()`, `inet_ntop()`, `inet_ntop()`, `listen()`, `memcpy()`, `memset()`, `mkfifo()`, `mknod()`, `ntohl()`, `ntohs()`, `perror()`, `pipe()`, `read()`, `recvfrom()`, `sendto()`, `select()`, `shutdown()`, `signal()`, `sizeof()`, `socket()`, `strtol()`, `system()`, `wait()`, `wait3()`, `waitpid()`, `write()`

W20.1 INTRODUCTION

Linux interprocess communication (IPC) facilities exist so that processes and threads may communicate with one another and synchronize their actions. Most, if not all, of these facilities have been adopted from UNIX. The three functional facilities into which IPC can be divided are communication, synchronization, and signals. Communication provides functionality to exchange data between processes or threads, synchronization provides for the synchronization of processes or threads, and signals are a communication technique where the signal number itself is a form of communication information. Although some of these facilities fundamentally exist for communication and synchronization, the term IPC is used to describe them all.

The Linux IPC facilities provide similar functionality because of the following reasons:

1. Similar facilities evolved on different UNIX families, and then migrated between families before they were ported to Linux. For example, First In First Out (FIFOs) were developed on System V, and (stream) sockets were developed on Berkeley Software Distribution (BSD) UNIX.
2. New facilities have been developed to make up for the deficiencies of similar earlier facilities. For example, the POSIX IPC facilities (message queues, semaphores, and shared memory) were

designed to improve on the older System V IPC facilities. In particular, POSIX IPC is thread-safe, whereas Sytem V IPC is not. Linux IPC facilities are POSIX compliant.

Some facilities, which on the surface seem similar, are actually significantly different in terms of the methods and functionality they provide. For example, stream sockets can be used to communicate over a network, while FIFOs can be used only for communication between processes on the same machine. Of all of the IPC methods, only sockets permit processes to communicate over a network. Sockets are generally used in one of two domains: that which allows communication between processes on the same system, and on the Internet, which allows communication between processes on different hosts connected via a Transmission Control Protocol/Internet Protocol (TCP/IP) network. Often, only minor changes can convert a program that uses sockets between the two domains.

Our coverage of the topic is broadly divided into three types: (a) IPC between related processes on the same computer, (b) IPC between unrelated processes on the same computer, and (c) IPC between related or unrelated processes on the same or different computers on a network, including the Internet. The relationship between processes is normally a parent–child or sibling relation. We cover in detail the system calls, library functions, macros, data structures, and header files involved in creating the requisite communication channels, as well as using them for reading and writing messages between the processes involved in communication. After covering the preliminary topics, we focus on the discussion of Linux IPC under the client–server paradigm and the fundamentals of Internet working with Linux TCP/IP. In doing so, we describe the design of various types of clients and servers, including iterative and different types of concurrent servers based on slave processes and the `select()` system call. We also explain what a superserver is and how the Linux superserver, `inetd`, works. Finally, we discuss concurrent clients. Throughout this chapter, we use the terms *computer*, *machine*, and *host* interchangeably.

W20.2 IPC: COMMUNICATION CHANNELS AND COMMUNICATION TYPES

For communication between processes, you require a communication channel and communicating process. The characteristics of communication channels are dependent on the location of processes, their relationship with each other, and the type of communication to be carried out between these processes. Processes may fall into the following types:

(a) Related processes on the same computer
(b) Unrelated processes on the same computer
(c) Unrelated processes on different computers on a network

The communication channel may be categorized based on the direction of communication and the nature of messages through the channel. The communication channel may support only *simplex* (i.e., one-way) communication or it may support *full-duplex* (i.e., two-way/bidirectional) communication. The messages may be just a *stream of bytes* or messages with clear boundaries. Messages with clear boundaries may be of fixed or variable size but with a maximum size limit and, depending on the literature you read and the network layer at which they are handled, may be called *packets*, *frames*, *datagrams*, *segments*, *chunks*, and so on.

Depending on the algorithm/protocol used by them, the communicating processes may have to establish a commection with each other before starting communication, or they may just send messages by specifying the address/name of a process. The first style of communication is called *connection oriented* and results in guaranteed, in-order, error-free delivery of messages. The second style of communication is known as *connectionless* communication and results in best-effort delivery of messages but without any guarantees. The messages used in connection-oriented communication are usually without boundaries and those used in connectionless communication are usually with boundaries.

There are several communication channels that support IPC on Linux machines, including the following:

1. Exit-wait
2. Pipe
3. Named pipe (FIFO)
4. Semaphore
5. Spin lock
6. Shared memory
7. Message queue
8. Sockets

We focus primarily on the following three commonly used channels: *pipe*, *named pipe* (also called *FIFO*), and *socket*. Which channel you should use in your application depends on the relationship between communicating processes, their location, and the nature of application from the point of view of reliability of message delivery and the integrity of message content.

W20.3 IPC: IMPORTANT SYSTEM AND LIBRARY CALLS, DATA STRUCTURES, MACROS, AND HEADER FILES

In this section, we briefly describe the most commonly used Linux channels of communication between the three types of processes already mentioned (i.e., pipe, named pipe, and socket), related data structures, system calls, library functions, and macros. We will discuss them in detail later under appropriate sections. Using the taxonomy of process types in Section W20.2, a pipe may be used for communication between type (a) processes, a named pipe (also called FIFO) for communication between type (a) processes or type (b) processes, and a socket for communication between processes belonging to any one of the three types. We describe in detail these channels as well as system calls and library functions needed for input/output (I/O) through these channels in the remainder of this chapter. Table W20.1 shows a brief summary of important system calls required for IPC using pipes, named pipes (FIFOs), and sockets.

Table W20.1 Brief Summary of Linux System Calls for IPC, Creating and Using Pipes, Named Pipes (FIFOs), and Sockets

System Call	Purpose
`pipe()`	Creates an IPC channel (two descriptors) for IPC between related processes on the same computer
`mkfifo()`	Creates an IPC channel (a file of a particular type) for IPC between related or unrelated processes on the same computer
`socket()`	Creates an endpoint (descriptor) for network-based IPC
`bind()`	Binds a local IP and protocol port number to a socket
`listen()`	Puts the socket in passive (listening) mode and sets the size of the queue where incoming connection requests may wait
`connect()`	Establishes a connection with a remote server
`accept()`	Accepts a client request for connection
`select()`	Waits for a connection request on a bit-set (flags) representing a set of descriptors for a specific period of time and flags for those descriptors that are ready for I/O
`sendto()`	Sends a datagram to a socket whose address has been prerecorded
`recvfrom()`	Receives a datagram and records the address of the sender socket
`read()`	Receives data (or datagram) from a socket on which `connect()` has been called
`write()`	Sends data (or datagram) to a socket on which `connect()` has been called
`close`	Terminates communication and deallocates a descriptor
`shutdown()`	Terminates TCP communication (I/O) in one or both directions

The list of commonly used macros and library calls in network-based IPC along with their purposes is shown in Table W20.2.

Table W20.2 Commonly Used Macros, Library Calls, and Their Purpose

Macro	Purpose
`htons()*`	Converts a 16-bit value in host byte order to network byte order
`htonl()*`	Converts a 32-bit value in host byte order to network byte order
`ntohs()*`	Converts a 16-bit value in network byte order to host byte order
`ntohl()*`	Converts a 32-bit value in network byte order to host byte order
`bzero()`	Initializes a string to null (zero) bytes
`memset()`	Initializes a string to bytes of a particular character's value
`memcpy()`	Copies the given number of bytes from one string to another
`FD_SET()**`	Includes the given descriptor in the set (i.e., sets the relevant numbered bit to 1)
`FD_CLR()**`	Exclude the given descriptor in the set (i.e., sets the relevant numbered bit to 0)

`FD_ZERO()**`	Initializes a descriptor set to null set (i.e., all zeros)
`FD_ISSET()**`	Tests if a particular descriptor in the set is 0 or 1 (i.e., if the given descriptor is ready for I/O or not)
`getdtablesize()`	Gets the size of *per-process file descriptor table* (PPFDT) (i.e., the maximum number of file descriptors that the system may use simultaneously)
`gethostbyname()`	Returns a pointer to a variable of the following structure describing an Internet host referenced by name
`gethostbyaddr()`	Returns a pointer to a variable of the following structure describing an Internet host referenced by address
`getservent()`	Returns a pointer to a variable of the `servent` structure with fields containing corresponding values in a line in the **/etc/services** file
`inet_addr()`	Converts and returns the specified string for an IP address in dotted decimal notation (DDN) to a 32-bit unsigned binary value in network byte order
`inet_aton()`	Converts the string containing an IPv4 address in DDN to an address in network byte order and stores it in an address structure
`inet_ntoa()`	Converts an IPv4 address in network byte order to a string containing the address in DDN
`inet_ntop()`	This is the newer version of the `inet_ntop()` function that works with both IPv4 and IPv6
`inet_pton()`	This is the newer version of the `inet_aton()` function that works with both IPv4 and IPv6

* These functions are normally used to convert IP addresses and port numbers returned by the `gethostbyname()` and `getservent()` library calls. On machines that have the same byte order as the network byte order, these functions are defined as null macros.

** These macros, defined in **/usr/include/select.h**, are normally used with the `select()` system call and are meant to set and test bits in a *descriptor set*, that is, a *bit mask* in which a particular number represents the state of the descriptor with that number. For example, the value in bit number 2 represents whether descriptor 2 is ready for I/O or not; the bit value 0 means "no," and the value 1 means "yes." The behavior of these macros is undefined if a descriptor value is negative or greater than the largest descriptor value on the system.

The list of important data structure in network-based IPC along with their purposes is shown in Table W20.3.

Table W20.3 Important Data Structures for Network-Based IPC and Their Purpose

Data Structure	**Purpose**
`struct sockaddr`	Most general structure for specifying the address of a socket
`struct sockaddr_un`	Structure for specifying the address of a local (i.e., UNIX domain) socket
`struct in_addr`	Structure to store the IPv4 address
`struct sockaddr_in`	Structure used to specify the address of an Internet domain (`PF_INET` or `PF_INET6`) socket
`struct hostent`	Structure to maintain information about a host on the Internet
`struct servent`	Structure to maintain information about an Internet service

Details of these system calls, library functions, macros, and data structures are described at appropriate places in subsequent sections.

The list of important header files, specifically designed for IPC through pipes, FIFOs, and sockets, and used throughout the chapter, is briefly described in Table W20.4.

Table W20.4 Brief Description of Important Header Files Included in Programs

Header File	**Purpose**
`<netdb.h>*`	Definitions of various symbolic constants, data structures, and prototypes of functions to maintain and manipulate information about hosts, networks, servers, protocols, and Internet addresses
`<time.h>`	Definitions of various symbolic constants, data structures, and prototypes of functions to maintain and manipulate information about time

<netinet/in.h>	Definitions of assigned numbers according to RFC1700 related to protocols, TCP ports, multicast addresses, and so on as symbolic constants, storage order for multibyte values, data structures (such as Internet style socket address structure), and prototypes of functions to maintain and manipulate them
<arpa/inet.h>	Definitions of various symbolic constants, data structures, and prototypes of functions to maintain and manipulate information about IP addresses for IPv4 and IPv6
<sys/errno.h>	Definitions of various symbolic constants for the errors produced by system and library calls
<sys/select.h>	Definitions of symbolic constants, data structures, macros, and prototypes for the `select()` and `pselect()` system calls
<sys/socket.h>	Definitions of symbolic constants, related data structures, and prototypes for system and library calls for socket types, address families, protocol families, message headers, addresses, and options
<sys/stat.h>	Definitions of symbolic constants for different statistics and values for files (inode number, hard link count, user ID, group ID, file size, time last accessed, time last modified, permission masks, type masks, etc.), macros to determine the type of a file, and prototypes for system calls to create and manipulate different types of files
<sys/types.h>	Definitions of symbolic constants for different types of items such as data units, inodes, file flags, disk addresses, Internet addresses, group IDs, process IDs, user IDs, thread IDs, access permissions, link counts, file offsets, resource limits, and so on; macros and function prototypes for related system calls
<sys/un.h>	Definitions of symbolic constants and data structure of socket addresses for Linux-based IPC

* <abc.h> means /usr/include/abc.h

You would include one or more of these files in your programs as and when needed. In a UNIX system, the same header file would contain the definitions of the symbolic constants related to a relevant system call and/or data structure(s). In Linux, however, which header files contain definitions for specific data structures, is an implementation level detail that you should not be concerned as a programmer. You may use the `find` command to locate all of the files relevant to each of these header files. For example, if you want to know the pathnames for all the files related to the definitions of the socket data structure and the `socket()` system call, you may use the `find /usr/include -name select.h` command, as in the following session.

```
$ find /usr/include -name socket.h
/usr/include/asm-generic/socket.h
/usr/include/x86_64-linux-gnu/sys/socket.h
/usr/include/x86_64-linux-gnu/asm/socket.h
/usr/include/x86_64-linux-gnu/bits/socket.h
/usr/include/linux/socket.h
$
```

W20.3.1 Byte Orders

Multibyte values may be stored, communicated, and manipulated in two orders: *little endian* and *big endian*. In the little endian order, the low-order byte of data value is stored in low storage location and the high-order byte in the high location. Thus, if the low-order byte is stored at memory location L, then the high-order byte is stored at location L + 1. The order is reversed in the big endian storage order; that is, the low-order byte of data value is stored in the high storage location (at address L + 1) and the high-order byte is stored in the low storage location (at address L). Figure W20.1 shows these storage orders in pictorial form.

Memory Address

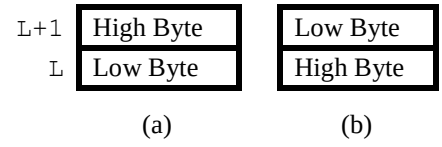


Figure W20.1 Storage orders for 16-bit data: (a) little endian (b) big endian.

Internet protocols deal with multibyte values in the big endian order. This means that network software stores and transmits multibyte data in the big endian order. Hence, the big endian order is also known as *network byte order*. Depending on the brand of central processing units (CPUs) used in the hosts on a network, they may use network byte order or little endian order. Intel processors use little endian order, whereas Sun and Motorola processors use big endian order. Thus, if a host uses an Intel CPU, it deals with multibyte values in little endian order.

The following program may be used to determine the byte order used by your computer system. Note that the compilation and sample run of the program shows that our Linux system runs on a machine that uses little endian byte order. The `uname -p` command shows that our system uses x86 family 64-bit processor.

```
$ cat byteorder.c
#include <stdio.h>

int main()
{
    int n;
    char *cp;

    n = 0x12345678;
    cp = (char*)(&n);
    if (*cp != 0x78)
       printf("Big Endian\n");
    else
       printf("Little Endian\n");
    return 0;
}
$ gcc byteorder.c -w -o byteorder
$ ./byteorder
Little Endian
$ uname -p
x86_64
$
```

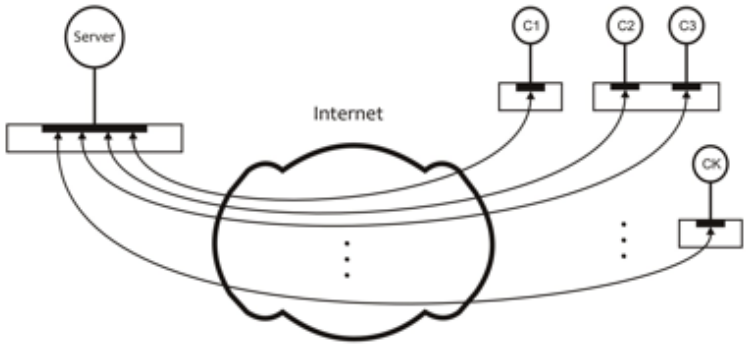
W20.4 THE CLIENT–SERVER MODEL

The design of applications to solve certain types of problems necessitates that we divide these applications into two independently running processes that communicate with each other to provide solutions for such problems. All Internet services are provided through such applications, including Web browsing, remote file transfer, remote login, remote program execution, video streaming, email, and Internet games. Today, IPC primarily deals with processes of such applications communicated by sending messages back and forth. The Internet works on the *client–server* model. Web servers, database servers, and social media all use these types of interactions among client and server processes. In this model of communication, a process, called the *server*, offers some kind of service to other processes and runs on a host whose address is known. Another process, called the *client*, runs on the same host on which the server process runs, or on a different host, and initiates communication with the server process to use its service. We describe various client–server models throughout the rest of the chapter and different types of servers in detail in Section W20.7 and subsequent sections.

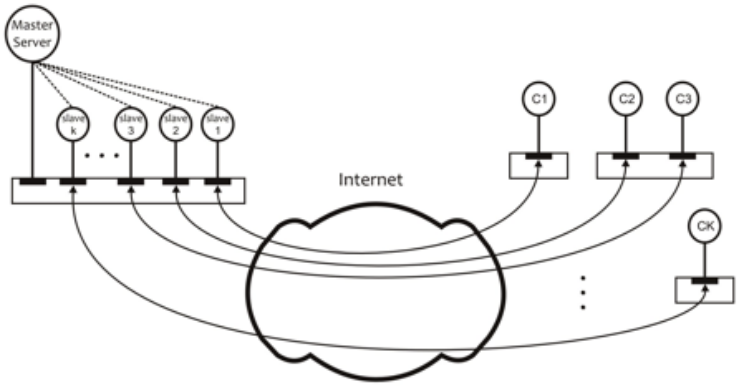
Server processes run forever and quietly wait for service requests from client processes. On receiving a request from a client process, the server process prepares a response, sends the response to the client process, and waits for the next request. Such servers are known as *iterative servers*. It is possible that, on receiving a client request, the server process creates a child process, delegates the rest of the communication with the client to that child process, and goes back to wait for another client request. Such servers are known as *concurrent servers*. In this style of client–server model, the original server process is

known as the *master server* and the processes that it creates to handle communication with client processes are known as *slave processes*. Servers that need to respond to clients with one-time, short responses are usually iterative and those that need to interact with clients in request–response sessions are concurrent. Figure W20.2 shows the conceptual models for the iterative and concurrent servers with a server process serving *k* clients. Note that clients may run on a single host, including the host that runs the server process, or on multiple hosts.

Applications may be standard—also known as *well-known applications*—or nonstandard (*unknown*). Nontechnically speaking, well-known applications are those that most users know of and use, such as Web browsing, downloading files, remote execution of programs, voice or video streaming, email, and logging on to a remote host. Technically speaking, well-known applications are those that are built around communication protocols described in Requests for Comments (RFCs) such as Hypertext Transfer Protocol (also called HTTP and WWW), File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP), and secure shell (SSH). Nonstandard applications may be sample applications written by students in a class or applications written for private use by individuals, groups of individuals, or employees of an organization or group of organizations.



(a)



(b)

Figure W20.2 Conceptual models for (a) iterative server (b) concurrent server.

W20.4.1 Simplest Form of Communication

As discussed in detail in Chapter 16, the most primitive form of communication between Linux processes is between a parent process and its children processes. This communication involves the use of the `exit()` system call (or the `exit()` library call) in a child process and the `wait()` system call (or a variant) in the parent process. Through this communication, a child communicates its exit status to the parent process. The exit status of the child is transferred from the `exit()` system call in the child to the `wait()` system call in the parent process. The Linux kernel handles this communication implicitly and no explicit communication channel is involved.

W20.4.2 Communication via Pipes

A pipe is a full-duplex (two-way) communication channel that allows two related processes to communicate with each other in terms of a stream of bytes, without message boundaries. It is normally used for communication between siblings, or parent and child. Although a pipe is a bidirectional communication channel, it is normally used for simplex (i.e., one-way) communication between two processes—a *reader process* and a *writer process*. Consequently, for two-way communication between two processes, a minimum of two pipes are usually required.

A pipe may be used to connect the standard output of a process (command) to the standard input of another process (command), as discussed in Chapter 9. The command level syntax for accomplishing this task is `cmd1 | cmd2`, as in `sort datafile | grep "John Doe"`.

From an implementation point of view, a pipe is a *fixed-size main memory circular buffer* created and maintained by the Linux kernel. In the operating system lexicon, it is also known as a *bounded buffer*. Communication using a pipe is, therefore, an implementation of the *bounded-buffer reader–writer problem*. The Linux kernel handles the synchronization required for making the reader process wait when the pipe is empty and the writer process wait when the pipe is full.

The `pipe()` and `pipe2()` system calls may be used to create a pipe. Here is a brief description of the two calls.

```
#include <unistd.h>
int pipe(int filedes[2]);
int pipe2(int filedes[2], int flags);
```

Success: 0

Failure: `-1` and kernel variable `errno` set to indicate the type of error

A call to `pipe()` or `pipe2()` returns two descriptors in the `filedes[2]` array, both allocated in the PPFDT of the caller process. Thus, creating a pipe is equivalent to opening two files. However, unlike an open file, no file pointer is associated with a pipe. Thus, each write request to a pipe appends data to the current end of the pipe. The first descriptor, `filedes[0]`, is used to read from the pipe, and the second, `filedes[1]`, is used to write to the pipe.

The `flags` argument in `pipe2()` is used to control the attributes of the pipe descriptors. Table W20.5 describes the possible flag values.

TABLE W20.5 The Flags for the `flags` Argument in the `pipe2()` System Call

Flag	Meaning
`O_CLOEXEC`	Set the "close-on-exec" flag on the pipe descriptors. This means when a process executes an `exec()` system call, it does not inherit an already open pipe
`O_NONBLOCK`	Set pipe descriptors for nonblocking I/O. This means that a `read()` system call will not block when the pipe is empty and a `write()` system call will not block when the pipe is full

A bitwise-OR of these values may also be used for the `flags` argument. When the `flags` argument has a value of 0, the `pipe2()` system call behaves like the `pipe()` system call.

The maximum amount of data that can reside in a pipe is dictated by the size of the pipe as a bounded buffer (i.e., the fixed-size array of characters). When a pipe is full and a writer process wants to put data into it, the writer process is blocked by the Linux kernel. Similarly, if a pipe is empty and a reader process tries to read data from the pipe, the reader process blocks. This is known as *blocking I/O* through a pipe.

The amount of data a writer process can write without interruption is known as the size of an *atomic write* into the pipe. It is defined as `PIPE_BUF` in the **/usr/include/linux/limits.h** file on our Linux Mint system. The value of `PIPE_BUF` is 4,096 bytes. If multiple processes write to a pipe, then requests of `PIPE_BUF` bytes or less are written atomically for each process. A request to write data greater than `PIPE_BUF` bytes may have data interleaved, on arbitrary boundaries, with writes of other processes.

If the `O_NONBLOCK` flag is not set, a request to write *n* bytes of data may block a thread if there is not enough space in the pipe to write the requested data. However, on successful completion, the `write()` system call returns *n*.

If the `O_NONBLOCK` flag is set, then a `write()` system call never blocks and the thread continues its execution. When sufficient space is available in a pipe, a request to write `PIPE_BUF` bytes or less completes successfully. If sufficient space is not available, the `write()` system call returns `-1` and

errno is set to EAGAIN. If no space is available in a pipe, then a write() system call for writing more than PIPE_BUF bytes returns −1 with errno set to EAGAIN. If space is available for at least one byte, the call writes what it can and returns the number of bytes written. When the data previously written to the pipe has been read, the call writes at least PIPE_BUF bytes.

The pipe() and pipe2() calls may fail for the reasons listed in Table W20.6.

Table W20.6 Reasons for the pipe() and pipe2() System Calls to Fail

Reason for Failure	**Value of** errno
The PPFDT does not have two unused file descriptors	EMFILE
The system file table is full	EMFILE
The kernel does not have enough memory to create a pipe	ENOMEM
The flags argument in pipe2() is not valid	EINVAL

W20.4.2.1 Allocation of Pipe Descriptors

The create_pipe.c program shown next creates a pipe and displays the values of descriptors for the read and write ends of the pipe. The Linux kernel opens three standard files automatically for each process using file descriptors 0 (standard input), 1 (standard output), and 2 (standard error). Therefore, when we run this program, the kernel allocates the next two unused file descriptors, 3 and 4, to the read and write ends of the pipe, respectively. Figure W20.3 shows the pipe with its relationship to the PPFDT.

```
$ cat create_pipe.c
#include <unistd.h>

int main(void)
{
       int data_channel[2];

       if (pipe(data_channel) == -1) {
           perror("Pipe failed");
           exit(1);
       }
       printf("The pipe descriptors are: \n");
       printf("      Read end:  %d\n", data_channel[0]);
       printf("      Write end: %d\n", data_channel[1]);
       exit(0);
}
$ gcc create_pipe.c -w -o createpipe
$ ./createpipe
The pipe descriptors are:
       Read end:  3
       Write end: 4
$
```

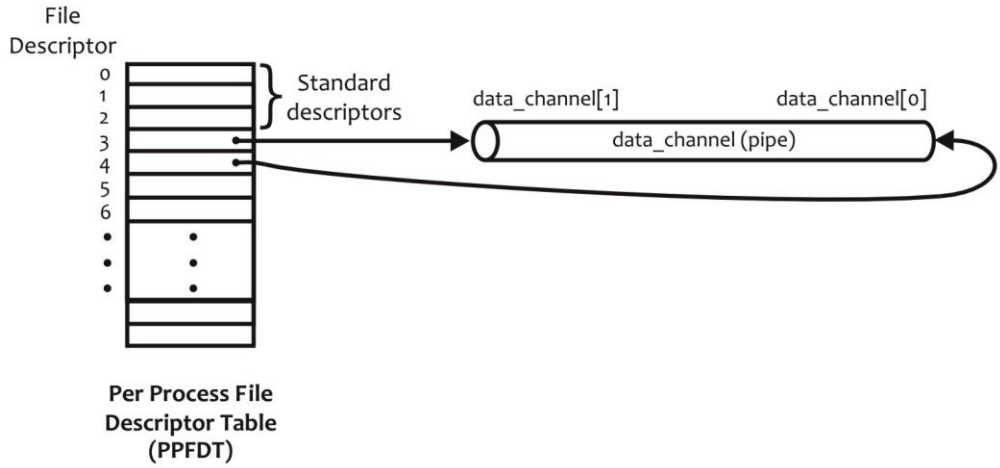


Figure W20.3 The pipe created by the create_pipe.c program and its relationship with PPFDT

W20.4.2.2 One-Way Communication

The `pipe_talk.c` program creates a pipe and spawns a child process. The child process, among other things, as discussed in Chapter 16, inherits the parent's PPFDT. The child process closes the write end of the pipe and reads data from the read end of the pipe. The child process blocks if there is nothing in the pipe. The parent process, on the other hand, closes the read end of the pipe and writes data to the child process using the write end of the pipe. The child process displays on the screen whatever it reads from the pipe. Figure W20.4 shows the setup of the program in a pictorial form.

```
$ cat pipe_talk.c
#include <unistd.h>

#define SIZE 32

const char *Child_Greeting="Hello, mom!\n";

int main(void)
{
       int data_channel[2], pid, nr, nw, nbytes;
       char buf[SIZE];

       if (pipe(data_channel) == -1) {
           perror("Pipe failed");
           exit(1);
       }
       pid = fork();
       if (pid == -1) {
           perror("Fork failed");
           exit(1);
       }
       nbytes = strlen(Child_Greeting);
       if (pid == 0) {
           close(data_channel[0]);
           nw = write(data_channel[1], Child_Greeting, nbytes);
           if (nw == -1) {
              perror("Write error");
              exit(1);
           }
           exit(0);
       }
       /* Parent process */
       close(data_channel[1]);
       nr = read(data_channel[0], buf, nbytes);
       if (nr == -1) {
           perror("Read error");
           exit(1);
       }
       nw = write(1, buf, nr);
       if (nw == -1) {
           perror("Write to stdout failed");
           exit(1);
       }
       printf("Well done, son!\n");
       exit(0);
}
$ gcc pipe_talk.c -w -o pipetalk
$ ./pipetalk
Hello, mom!
Well done, son!
$
```

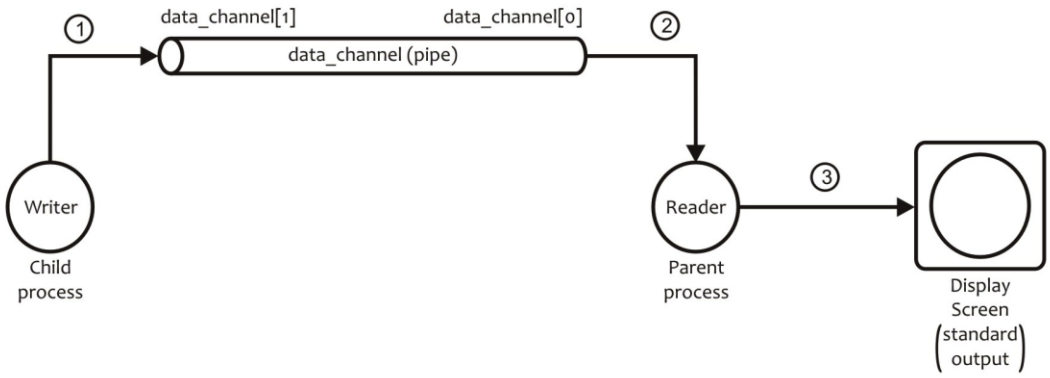


Figure W20.4 Pictorial representation of the IPC setup in pipe_talk.c

W20.4.2.3 Two-Way Communication

We now discuss a program in which the reader and writer processes carry out two-way communication using two pipes. The `pipe_2way_talk.c` program opens two pipes, `pipe1` and `pipe2`, and creates a child process. The child process sends a message to the parent process using `pipe1` and reads the parent's response from `pipe2`. The parent does it the other way round. Whatever data the parent and child processes read from their respective pipes, they throw to standard output. Figure W20.5 shows the setup in a pictorial form.

```
$ cat pipe_2way_talk.c
#include <unistd.h>

#define SIZE 32

const char *Child_Greeting="Hello, mom!\n";
const char *Parent_Greeting="Well done, son!\n";

int main(void)
{
       int pipe1[2], pipe2[2];
       int pid, nr, nw, status, sizec, sizep;
       char buf[SIZE];

       sizec = strlen(Child_Greeting);
       sizep = strlen(Parent_Greeting);
       if (pipe(pipe1) == -1) {
           perror("Pipe1 failed");
           exit(1);
       }
       if (pipe(pipe2) == -1) {
           perror("Pipe2 failed");
           exit(1);
       }
       pid = fork();
       if (pid == -1) {
           perror("Fork failed");
           exit(1);
       }
       if (pid == 0) {
           close(pipe1[0]);
           close(pipe2[1]);
           nw = write(pipe1[1], Child_Greeting, sizec);
           if (nw == -1) {
              perror("Write to pipe1 error in child");
              exit(1);
           }
           nr = read(pipe2[0], buf, sizep);
           if (nr == -1) {
              perror("Read pipe2 error in child");
              exit(1);
           }
```

```
            nw = write(1, buf, nr);
            if (nw == -1) {
               perror("Write to stdout failed in child");
               exit(1);
            }
            close(pipe1[1]);
            close(pipe2[0]);
            exit(0);
        }
        /* Parent process */
        close(pipe1[1]);
        close(pipe2[0]);
        nr = read(pipe1[0], buf, sizec);
        if (nr == -1) {
            perror("Read pipe1 error in parent");
            exit(1);
        }
        nw = write(1, buf, nr);
        if (nw == -1) {
            perror("Write to stdout failed in parent");
            exit(1);
        }
        nw = write(pipe2[1], Parent_Greeting, sizep);
        if (nw == -1) {
            perror("Write to pipe2 error in parent");
            exit(1);
        }
        close(pipe1[0]);
        close(pipe2[1]);
        wait(&status);
        exit(0);
}
$ gcc pipe_2way_talk.c -w -o p2wt
$ ./p2wt
Hello, mom!
Well done, son!
$
```

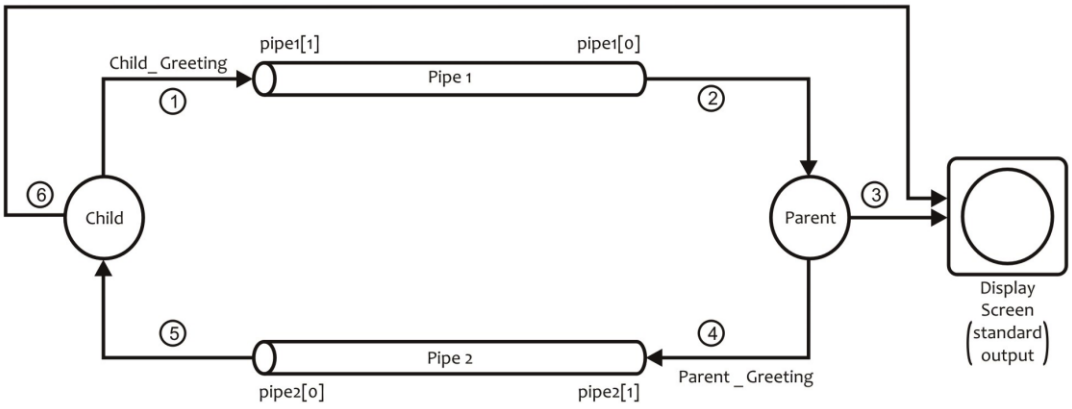


Figure W20.5 Pictorial representation of the IPC setup in pipe_2way_talk.c

W20.4.2.4 Widowed Pipe

A pipe that has one end closed is called a *widowed pipe*. When a writer process writes to a widowed pipe, the Linux kernel sends the `SIGPIPE` signal to the writer process that results in the termination of the writer process. The default action on this signal is that the kernel terminates the process and displays the "`Broken pipe`" message. When a reader process reads from a widowed pipe, it receives an end-of-file (eof) message after the reader has read all of the data in the pipe. This means that a `read()` system call on a widowed pipe returns the value 0.

In the `broken_pipe.c` program shown, we demonstrate how a widowed pipe may be created. The program creates a pipe, displays the read end and write end descriptors of the pipe, closes the read end of the pipe (i.e., no reader can read from the pipe), and writes a greeting message using the write end of the pipe. We compile the program to generate the executable code in the **brokenp** file. The execution of the

program works perfectly until it tries to write to the pipe after closing its read end. The `write()` system call generates the `SIGPIPE` signal and, due to the kernel's default action, the program terminates.

```
$ cat broken_pipe.c
#include <unistd.h>

int main(void)
{
       int data_channel[2], nw;

       if (pipe(data_channel) == -1) {
           perror("Pipe failed");
           exit(1);
       }
       printf("The pipe descriptors are: \n");
       printf("      Read end:  %d\n", data_channel[0]);
       printf("      Write end: %d\n", data_channel[1]);
       close(data_channel[0]);
       nw = write(data_channel[1], "Hello, world!\n", 14);
       printf("This and subsequent statements are never executed.\n");
       exit(0);
}
$ gcc broken_pipe.c -w -o brokenp
$ ./brokenp
The pipe descriptors are:
       Read end:  3
       Write end: 4
$
```

One some NIX systems, the "`Broken pipe`" message is displayed before the process terminates. However, on Linux, the process terminates without displaying the "`Broken pipe`" error message. You can use the `signal()` call to intercept the `SIGPIPE` signal, display the desired error message, and terminate the program. You will be asked to do this in one of the problems at the end of this chapter.

Exercise W20.1

Compile and run the `create_pipe.c`, `pipe_talk.c`, `pipe_2way_talk.c`, and `broken_pipe.c` programs in the previous session to make sure they work on your system as expected.

W20.5 COMMUNICATION BETWEEN UNRELATED PROCESSES ON THE SAME COMPUTER

Two or more related or unrelated processes on the same machine can communicate with each other using several Linux IPC channels, including a named pipe (also known as a FIFO) and a socket. We discuss FIFOs in this section and sockets in Section W20.6. Earlier, we discussed FIFOs in Section 9.15. Our treatment of the topic was focused on the command line use of FIFOs for connecting shell commands with each other to perform complex tasks that cannot be performed by existing commands. Here, we discuss the details of the underlying structure of a FIFO as an IPC channel and the Linux application programmer's interface (API) that allows creation of FIFOs and their use.

As stated earlier, a FIFO is a named pipe. It is a pipe that has a name in the file system name space, an associated file type, related kernel data structures that contain its attributes, and the main memory bounded-buffer that contains data in transition through the pipe. The pipe part of the FIFO, a main memory buffer, is created when a process opens the FIFO and is destroyed when the process closes the FIFO. Thus, unlike a pipe that is purely a main memory object and is *process persistent*, a FIFO is an amalgamation of disk and memory objects. The pipe part of it is process persistent and the name part is *file system persistent*. To sum up, when you create a FIFO, the kernel creates a pipe in main memory and connects it with a file system through a pathname in the file system name space and associated resources including an inode. This allows you to access a FIFO as a file system object.

To use a FIFO as an IPC channel, you create it with a pathname and then open it for reading, writing, or both. However, as expected, just like a pipe, a FIFO also does not have a file pointer associated with it. New data is written at the current end of the FIFO and existing data is read from the front of the FIFO.

When you create a FIFO, it does not contain anything, in the same way as an empty regular file; and its disk usage is zero. When you write data to a regular file, its disk usage depends on the amount of data

written to the file with a minimum usage of four blocks. However, the disk usage for a FIFO does not change whether it is empty or full, because its data is maintained in the memory-resident pipe.

In the following session, we create a FIFO, called **fifo1**, and an empty file, called **greeting**. Both are allocated inodes numbered 16518535 and 16518536, respectively. The output of the `ls -l fifo1 greeting` command shows that both are empty. The output of the `du fifo1 greeting` command shows that both use no disk space. We put "`Hello, world!`" (14 bytes) in the **greeting** file as well as **fifo1**. The output of the second `ls -l fifo1 greeting` command shows that **fifo1** is still empty, but **greeting** contains 14 bytes. The output of the `du fifo1 greeting` command shows that while **fifo1** still uses no disk space the **greeting** file uses four disk blocks. Even after we empty **fifo1** using the `cat fifo1` command, the disk usage of **fifo1** remains unchanged. This session shows that when we put data into a FIFO (**fifo1** in this case) the data does not go into a disk object associated with it but into the main memory buffer (pipe) associated with the FIFO.

```
$ mkfifo fifo1
$ touch greeting
$ ls -i fifo1 greeting
16518535 fifo1  16518536 greeting
$ ls -l fifo1 greeting
prw-r--r-- 1 sarwar faculty 0 Aug 25 12:08 fifo1
-rw-r--r-- 1 sarwar faculty 0 Aug 25 12:08 greeting
$ du fifo1 greeting
0       fifo1
0       greeting
$ cat > greeting
Hello, world!
<Ctrl+D>
$ cat greeting > fifo1 &
[1] 31092
$ ls -l fifo1 greeting
prw-r--r-- 1 sarwar faculty  0 Aug 25 12:08 fifo1
-rw-r--r-- 1 sarwar faculty 14 Aug 25 12:09 greeting
$ du fifo1 greeting
0       fifo1
4       greeting
$ cat fifo1
Hello, world!
[1]+  Done                    cat greeting > fifo1
$ du fifo1 greeting
0       fifo1
4       greeting
$
```

Exercise W20.2

Replicate the earlier shell session on your Linux system. Does it produce the same results? If not, list the differences between the results of our session and your session, and explore various sources on the Internet to explain the differences.

You can use any of the `mknod()`, `mknodat()`, `mkfifo()`, or `mkfifoat()` system calls to create a FIFO. However, the `mknod()` and `mknodat()` system calls require superuser privileges. The primary purpose of these system calls is to create special files, but they can also be used to create FIFOs. The `mkfifo()` and `mkfifoat()` calls do eventually invoke the `mknod()` system call. Once a FIFO has been created, you can use the `open()`, `read()`, `write()`, and `close()` system calls to perform I/O with it through multiple reader and writer processes. The last process that uses a FIFO, usually a reader process, closes the FIFO and removes it from the file system using the `unlink()` system call. Because multiple processes can write to a FIFO, the Linux kernel ensures that data up to `PIPE_BUF` bytes written by multiple processes each is written atomically and does not interleave. As stated in Section W20.4.2, `PIPE_BUF` is defined in **/usr/include/linux/limits.h** to be 4,096 bytes on Linux.

We first describe the `mkfifo()` and `mkfifoat()` system calls, listed as library calls under Solaris, and their manual pages may be viewed by using the `man -s3 mkfifo` command. We then discuss a few sample programs to describe the use of FIFOs as IPC channels. We primarily use the `mkfifo()` call in our sample programs. Here are brief descriptions of the `mkfifo()` and `mkfifoat()` system calls.

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
int mkfifoat(int fd, const char *path, mode_t mode);
```

Success: 0

Failure: −1 and kernel variable errno set to indicate the type of error

The mkfifo() creates a FIFO with the name given in the path argument having access permissions specified in the mode argument. As is the case with other types of files, access permissions are restricted by the current umask. In the mkfifoat() system call, the path argument is relative to the directory associated with the file descriptor fd and not the current working directory of the user. If the fd argument is AT_FDCWD, the behavior of the mkfifoat() call is identical to the mkfifo() call.

The mkfifo() and mkfifoat() calls may fail mostly for the same reasons that the creat() and open() system calls may fail, as discussed in Section 15.8.1 and listed in Table 15.4. A few additional reasons for the failure of these calls are listed in Table W20.7.

Table W20.7 Reasons for the mkfifo() and mkfifoat() System Calls to Fail

Reason for Failure	**Value of** errno
The directory under which the FIFO is to be created does not have write permission, or a component directory in the path prefix is not searchable	EACCES
The FIFO named in path exists	EEXIST
For the mkfifoat() call, the fd argument is neither AT_FDCWD nor a valid descriptor for searching and the path argument is not an absolute pathname	EBADF
For the mkfifoat() call, the fd argument is neither AT_FDCWD nor a valid descriptor for a directory and the path argument is not an absolute pathname	ENOTDIR

The behavior of a FIFO that is not fully opened for I/O is similar to that of a pipe under the same condition. A write to a FIFO is that no process has opened for reading results in a SIGPIPE signal to the writer process. When the last process to write to a FIFO closes it, an eof is sent to the reader process.

We now discuss a sample client–server model to illustrate the use of FIFOs for IPC between unrelated processes on the same machine. Figure W20.6 shows the algorithms for the client and server processes.

Server Process

1. Create two FIFOs, FIFO1 and FIFO2
2. Open FIFO1 for reading and FIFO2 for writing
3. Read “Hello, world!” from client through FIFO1
4. Display this message on the screen by writing it to standard output
5. Write “Hello, class!” to client through FIFO2
6. Close FIFO1 and FIFO2
7. Exit

Client Process

1. Open FIFO1 for writing and FIFO2 for reading
2. Write “Hello, world!” to server through FIFO1
3. Read FIFO2 for a message from server
4. Display this message, “Hello, class!”, on the screen by writing it to standard output
5. Close FIFO1 and FIFO2
6. Remove FIFO1 and FIFO2 from the file system
7. Exit

FIGURE W20.6 Algorithms for the client and server processes.

Here are the **fifo.h**, **client.c**, and **server.c** files that implement this client–server model.

```
$ cat fifo.h
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>

extern int      errno;

#define FIFO1   "/tmp/fifo.1"
#define FIFO2   "/tmp/fifo.2"
#define PERMS   0666
#define SIZE  512
```

```
static char* message1 = "Hello, world!\n";
static char* message2 = "Hello, class!\n";
$ cat client.c
#include "fifo.h"

int main(void)
{
    char buff[SIZE];
    int readfd, writefd;
    int n, size;

    /* Open FIFOs. Assume that the server
       has already created them. */
    if ((writefd = open(FIFO1, 1)) == -1) {
        perror ("client open FIFO1");
        exit (1);
    }
    if ((readfd = open(FIFO2, 0)) == -1) {
        perror ("client open FIFO2");
        exit (1);
    }

    /* client (readfd, writefd); */
    size = strlen(message1);
    if (write(writefd, message1, size) != size) {
        perror ("client write1");
        exit (1);
    }
    if ((n = read(readfd, buff, size)) == -1) {
        perror ("client read");
        exit (1);
    }
    else
        if (write(1, buff, n) != n) {
            perror ("client write2");
            exit (1);
        }
    close(readfd);
    close(writefd);

    /* Remove FIFOs now that we are done using them */
    if (unlink (FIFO1) == -1) {
        perror("client unlink FIFO1");
        exit (1);
    }
    if (unlink (FIFO2) == -1) {
        perror("client unlink FIFO2");
        exit (1);
    }
    exit (0);
}
$ cat server.c
#include "fifo.h"

int main(void)
{
    char buff[SIZE];
    int readfd, writefd;
    int n, size;

    /* Create two FIFOs and open them for
       reading and writing */
    if ((mknod (FIFO1, S_IFIFO | PERMS, 0) == -1)
                        && (errno != EEXIST)) {
        perror ("mknod FIFO1");
        exit (1);
    }
    if (mkfifo(FIFO2, PERMS) == -1) {
        unlink (FIFO1);
        perror("mknod FIFO2");
        exit (1);
    }
```

```
    if ((readfd = open(FIFO1, 0)) == -1) {
        perror ("open FIFO1");
        exit (1);
    }
    if ((writefd = open(FIFO2, 1)) == -1) {
        perror ("open FIFO2");
        exit (1);
    }

    /* server (readfd, writefd); */
    size = strlen(message1);
    if ((n = read(readfd, buff, size)) == -1) {
        perror ("server read");
        exit (1);
    }
    if (write (1, buff, n) < n) {
        perror("server write1");
        exit (1);
    }
    size = strlen(message2);
    if (write (writefd, message2, size) != size) {
        perror ("server write2");
        exit (1);
    }
    close (readfd);
    close (writefd);
}
$
```

Figure W20.7 shows the compilation and execution of the client–server model by using two terminal windows on our system. We compile the `client.c` and `server.c` programs and save the executable codes in the client and server files, respectively. We then run the server program in Window 1, followed by running the client program in Window 2. The client process sends the "`Hello, world!`" message to the server process through **FIFO1** and then waits for a message from the server process. The server process receives the client message and displays it on standard output. It then sends the "`Hello, class!`" message to the client process through **FIFO2**, closes both FIFOs, and exits. The client process receives the message and displays it on the screen. It then closes both FIFOs and removes them from the file system. Figure W20.8 shows the pictorial representation of the running of the client–server model.

Terminal Window 1

```
% gcc -w server.c -o server
% ./server
Hello, world!
%
```

Terminal Window 2

```
% gcc -w client.c -o client
% ./client
Hello, class!
%
```

FIGURE W20.7 Compilation and running of the client–server model in two terminal windows.

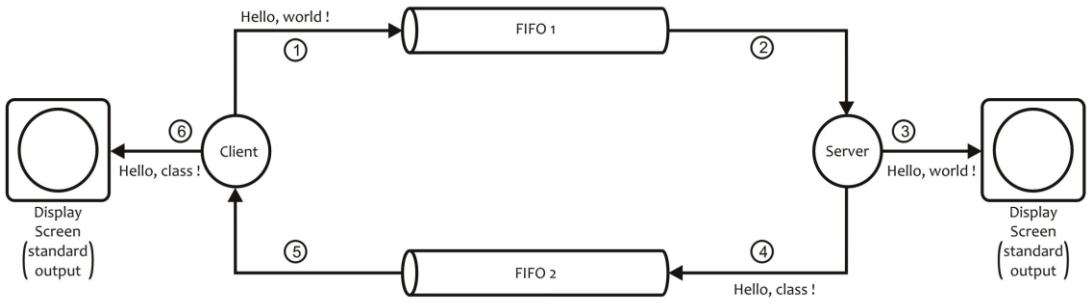


FIGURE W20.8 Pictorial representation of the execution of the client–server model using FIFOs.

The client–server model discussed is for communication between two processes running on the same machine. In this case, the communication is an exchange of one message each from client to server and

vice versa. The model may be extended to the exchange of multiple messages between the two processes. You can extend this model such that the server process may serve multiple clients. Figure W20.9 shows the general view of such a model.

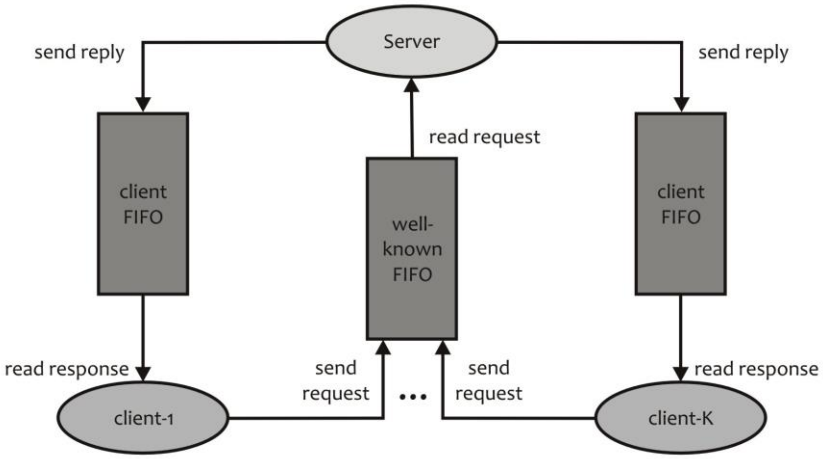


FIGURE W20.9 Client–server model with a server and multiple simultaneous clients using FIFOs.

Exercise W20.3

Compile and run this client–server model on your Linux system. Does it work as expected? If not, identify and list reasons for the issues.

Exercise W20.4

Are `mkfifo()` and `mkfifoat()` are available on your Linux system as system calls or library calls?

Exercise W20.5

What is the value of `PIPE_BUF` on your Linux system? Which file contains it? What does this value indicate?

The basic algorithms for the client and server processes for this client–server model are shown in Figure W20.10. Note that Steps 3–5 in the server code are typically implemented as an infinite loop.

Server Process

1. Create a "well-known" FIFO, i.e., create a FIFO and make its pathname well-known
2. Open the well-known FIFO for reading
3. Read a client request via the well-known FIFO; as part of its request a client sends the pathname of its FIFO to server
4. Prepare response and send it to client FIFO
5. Go to step 3

Client Process

1. Create a FIFO, called "client" FIFO, for reading server response(s)
2. Open server's well-known FIFO for writing
3. Prepare a request for the server that contains the path name of client FIFO
4. Send request to server via server's well-known FIFO
5. Receive server's response via client FIFO
6. Last response? No, go to 3.
7. Close well-known and client FIFOs
8. Remove client FIFO from the file system
9. Exit

Figure W20.10 Algorithms for the client and server processes.

W20.6 COMMUNICATION BETWEEN UNRELATED PROCESSES ON DIFFERENT COMPUTERS

Linux provides several IPC channels for communication between unrelated processes on the same or different machines on a network. The most commonly used channel is the *socket*. In this section, we discuss what a socket is and how it may be used for communication between processes. The communication between processes may be connection oriented or connectionless. We also explore the software architecture of the client–server models for the various Internet services by using the socket as the communication channel between the client and server processes. We will code some of the basic models and explain the source codes. We leave implementation of other models as end of chapter problems.

W20.6.1 Socket-Based Communication

As stated in Chapter 4, a socket is a file type in Linux. From the IPC point of view, a *socket* is a full-duplex IPC channel that may be used for communication between related or unrelated processes on the same or different machines. Both communicating processes need to create a socket to handle their side of communication: reading and writing. A socket is, therefore, called an *endpoint of communication*. It is the IPC channel of choice for network-based communication between processes under the client–server paradigm.

Like a FIFO, once a socket has been created, it is used by following the open–read–write–close paradigm used for typical file I/O. A socket remains in the system for as long as the process that creates it is up and running. Thus, a socket is *process persistent*.

W20.6.2 Creating a Socket

You can create a socket by using the `socket()` system call. Here is a brief description of the `socket()` system call.

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Success: Socket descriptor

Failure: `-1` and kernel variable `errno` set to indicate the type of error

The call returns a file descriptor from the PPFDT, known as the *socket descriptor*. Before a socket may be used, it needs to be set for a particular type of communication: connectionless or connection oriented. The `domain` argument specifies the domain (i.e., protocol family) under which the communication between a pair of sockets will take place. The value of this argument selects the protocol family that would be used for communication between sockets. These families are defined in **/usr/include/x86_64-linux-gnu/bits/socket.h**, but which header file(s) contain the various definitions is an implementation level detail; as a programmer, you simply include the **<sys/socket.h>** file in your program. A socket of a particular domain may support different types of communication under different protocols. The `type` argument specifies the type of communication for which the socket would be used. Communication may only take place between a pair of sockets of the same type. The `protocol` argument specifies the protocol to be used for the socket type within a given protocol family. However, a particular protocol needs to be specified if multiple protocols exist to support the requested type of communication under the given domain. If you set the `protocol` argument to 0, the system chooses the correct type of protocol for the given type of communication under the selected domain.

Table W20.8 lists some of the commonly used socket domains, and Table W20.9 shows the types of communication supported between a pair of sockets. The protocols for stream-oriented and datagram-oriented communication are TCP and User Datagram Protocol (UDP), respectively.

Table W20.8 Commonly Used Socket Domains

Protocol Family	**Purpose**
`PF_LOCAL` (Old `PF_UNIX`)	Host-internal protocols
`PF_INET`	IPv4 protocols
`PF_INET6`	IPv6 protocols

Table W20.9 Commonly Used Types of Communication between a Pair of Sockets

Socket Type	Communication Type
SOCK_STREAM	Stream-oriented
SOCK_DGRAM	Datagram-oriented
SOCK_RAW	Datagram-oriented (only available to superuser)

You use the PF_LOCAL domain to create a socket for communication between processes on the same host using internal protocol(s). The PF_INET and PF_NET6 domains are used for communication between processes on the Internet using the IPv4 and IPv6 protocols, respectively.

You can OR the flags given in Table W20.10 to mark a newly created socket as "close-on-exec" and/or set it for "nonblocking" I/O. When a socket is marked as "close-on-exec," it is closed when the process overwrites itself with an executable code using a call from the exec() family such as execlp(). The read() and write() system calls do not block on an empty and full socket, respectively, if the socket is marked as "nonblocking."

Table W20.10 Flags for the type Field

Flag	Purpose
SOCK_CLOEXEC	Mark the descriptor as "close-on-exec"
SOCK_NONBLOCK	Set the descriptor "nonblocking" I/O

The socket() system call may fail for various reasons. Some of the reasons for the failure of a socket() are listed in Table W20.11.

Table W20.11 A Few Reasons for socket() to Fail

Reason for Failure	Value of errno
Wrong type and/or protocol argument	EACCES
domain is not supported	EAFNOSUPPORT
PPFDT is full (i.e., no free file descriptor) or system-wide file table is full	EMFILE
Insufficient buffer space in kernel	ENOBUFS
Specified protocol not supported in the given domain	EPROTONOSUPPORT
Specified socket type not supported by the given protocol	EPROTOTYPE

W20.6.3 Domains of Socket-Based Communication

Socket-based IPC may take place in several domains, but two are most commonly used: *UNIX domain* and *Internet domain*. We briefly describe when these domains of communication are used. In the next section, we show how a socket may be created for communication under a particular domain.

W20.6.3.1 UNIX Domain Socket (PF_LOCAL or PF_UNIX)

To create a socket for communication under the UNIX domain, PF_LOCAL or PF_UNIX is used as the domain argument in the socket() system call. The IPC under this domain is between processes on the same machine with the socket address specified as the pathname in the file system, in the same way as that for a FIFO. UNIX domain protocols are not an actual protocol suite but a technique used to perform client–server communication between processes on the same machine using the socket API.

The UNIX domain sockets are preferred over FIFOs and the Internet domain sockets for IPC between processes on the same machine for the following reasons:

- UNIX domain sockets are full duplex.
- UNIX domain sockets are twice as fast as TCP sockets. For this reason, they are used in the client–server model for X Window System as well as for communication between a client and server when both are on the same host. An example of the latter case is when a printer client running on a machine sends a print job to the print server that also runs on the same machine.
- UNIX domain sockets are used for passing descriptors between processes on the same host.

W20.6.3.2 Internet Domain Socket (PF_INET or PF_INET6)

To create a socket for communication under the Internet domain, `PF_INET` or `PF_INET6` is used as the domain in the `socket()` system call. The IPC under this domain takes place between processes on the same machine or on different machines on a TCP/IP network, including the Internet. The address of an Internet domain socket is an IP address and a port number. The Internet domain protocols use the TCP/IP protocol suite for communication using the client–server paradigm.

The Internet domain sockets are the most commonly used channels for IPC and are the glue required for the construction of client–server applications for Internet services. In the next section, we discuss the most common types of communication that are carried out using sockets and the underlying protocols that support such communication.

W20.6.4 Types of Communication Using a Socket

IPC using sockets may be *connection oriented* or *connectionless*. In the connection-oriented style of communication, the two communicating processes create sockets of the required type and have them connected before starting communication. The connection between the two sockets is established using the *three-way TCP handshake*. Thus, a *virtual connection* is established between the two sockets, and the sender and receiver processes communicate using the `write()` and `read()` system calls, respectively. The `SOCK_STREAM` type of socket is used for connection-oriented, reliable, error-free, and in-sequence stream-oriented communication with no message/packet boundaries.

In the connectionless style of communication, the sender process simply sends messages to the receiver process and hopes that they will be delivered. The `SOCK_DGRAM` type socket is used for connectionless, unreliable (no guarantees), and datagram-oriented communication. A *datagram* is a small, fixed-length packet/message. Connection-oriented communication is like the guaranteed delivery service provided by FedEx, UPS, or DHL, and connectionless communication is like the normal, best-effort delivery service provided by a country's regular mail service with no guarantees.

W20.6.4.1 Stream Socket (SOCK_STREAM)

To create a socket for stream-oriented communication using IPv4, you specify `PF_INET` as the domain and `SOCK_STREAM` as the type of communication. TCP is the transport-level protocol that provides this type of communication. Thus, you can create an Internet domain socket for stream-type connection-oriented communication by using the following code snippet. Note that a value of 0 is used in the `protocol` argument to let the kernel choose the appropriate protocol for the requested style of communication.

```
<program>
int s; /* Socket descriptor */
s = socket(PF_INET, SOCK_STREAM, 0);
</program>
```

If you also want to mark this socket as nonblocking, you would use the following call:

```
<program>
int s;
s = socket(PF_INET, SOCK_STREAM | SOCK_ NONBLOCK, 0);
</program>
```

W20.6.4.2 Datagram Socket (SOCK_DGRAM)

To create an IPv4 socket for datagram-oriented communication, you would need to specify `PF_INET` as the domain and `SOCK_DGRAM` as the type of communication. UDP is the transport-level protocol that provides this type of communication. Thus, you can create an Internet domain socket for datagram-oriented, connectionless communication by using the following code snippet:

```
int s; /* Socket descriptor */
s = socket(PF_INET, SOCK_DGRAM, 0);
```

If you want to also mark this socket "close-on-exec," you would use the following call:

```
int s;
s = socket(PF_INET, SOCK_DGRAM | SOCK_CLOEXEC, 0);
```

W20.6.4.3 Compiling and Running Programs on Linux

You can compile your programs using the `gcc` command on Linux, as in

```
gcc –w source.c –o binary
```

where `source.c` is the name of the C program file that contains the source code and `binary` is the name of the file that contains the executable code for the given source code. The `–w` switch is used to suppress warnings.

The IPv4 address of our Linux machine is 202.147.169.195. Since we test run our server processes on this machine, we use this IP address as a command line argument for the client processes. We also use the local host address, 127.0.0.1, when we run the client and server processes on the same machine. When you test these programs in your environment, you can either specify the IP address of the machine that runs the server process or 12.0.0.1 if the client and server processes run on the same machine. Please note that you will not be able to run your client processes by using our Linux machine's IP address as command line argument on your system.

W20.6.4.4 First Socket-Based Program

The `sockets.c` program shown in the following session creates a socket each for stream-oriented and datagram-oriented communication between processes on the Internet and displays the descriptors allocated in the PPFDT for these sockets. The compilation and execution of this program show that the socket descriptors 3 and 4 have been assigned to the two sockets. Note that descriptors 3 and 4 are the next two descriptors available after the standard descriptors.

```
$ cat sockets.c
#include <sys/types.h>
#include <sys/socket.h>

int main(void)
{
        int s1, s2;

        if ((s1 = socket(PF_INET, SOCK_STREAM, 0)) == -1) {
            perror("SOCK_STREAM socket failed");
            exit(1);
        }
        if ((s2 = socket(PF_INET, SOCK_DGRAM, 0)) == -1) {
            perror("SOCK_DGRAM socket failed");
            exit(1);
        }
        printf("The socket descriptor for the stream socket is: %d\n", s1);
        printf("The socket descriptor for the datagram socket is: %d\n", s2);
        exit(0);
}
$ gcc sockets.c -w -o sockets
$ ./sockets
The socket descriptor for the stream socket is: 3
The socket descriptor for the datagram socket is: 4
$
```

When an Internet domain socket is created, the Linux kernel establishes a link between the socket descriptor for the newly created socket and the socket data structure allocated by the kernel when the socket is created. Figure W20.11 shows the high-level linkage between the socket descriptor and the associated socket data structure.

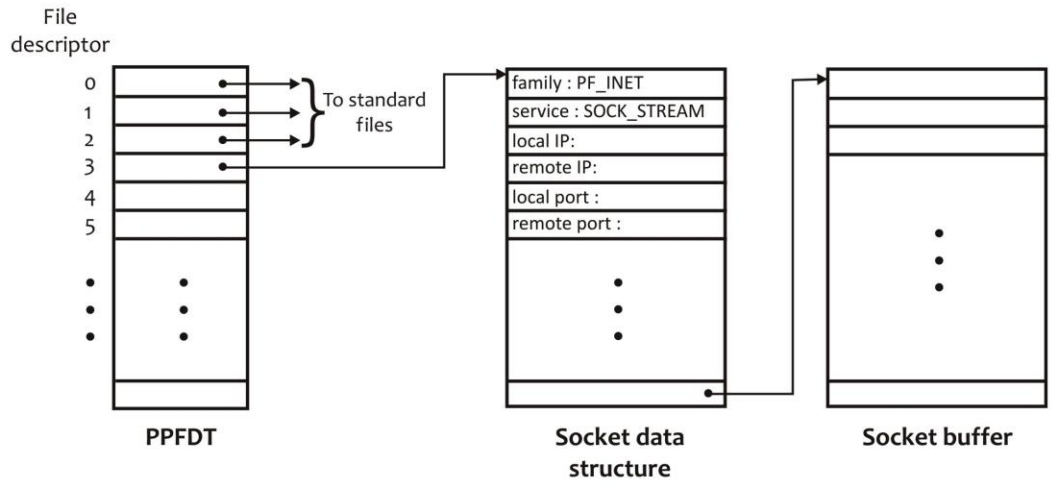


Figure W20.11 Partial description of socket descriptor, PPFDT, and socket data structure.

The socket data structure contains several pieces of information for the expected style of IPC, including domain, service type, local IP, remote IP, local port, and remote port. The Linux kernel initializes the first two fields when a socket is created. The local IP and local port are stored in the data structure explicitly by the client or server process that creates it using the `bind()` system call. The <local IP, local port> pair forms the local address of the socket. When the local address has been stored in the socket data structure, we say that the socket is *half associated*. The remote IP and remote port fields are filled out when a client process calls the `connect()` system call. The <remote IP, remote port> pair is the address of the remote socket. When both local and remote addresses have been stored in the client- and server-side sockets, we say that *full association* has been established between the two processes. When full association has been established between two sockets, we say that they have been connected and a virtual connection has been established between them. For `SOCK_STREAM` (i.e., TCP) sockets, this happens through the rendezvous of the `connect()` and `accept()` calls on the client and server sides, respectively, as discussed later in this chapter. Connection-oriented communication between two processes may take place only when full association exists between their sockets. We discuss socket addresses and their binding with sockets in more detail in the next two subsections.

W20.6.4.5 Reading Data from a Socket

When a connection has been established between the server- and client-side sockets for the `SOCK_STREAM` style of communication, they can communicate using the TCP protocol. This communication is based on a stream of bytes without any message boundaries. The sender process may send data using one or more `write()` calls. The receiver process may collect this data using single or multiple `read()` calls. Suppose the server process needs to send 128 bytes of data to the client process and it does so by sending four 32-byte chunks using four `write()` system calls. The receiver side may receive all 128 bytes of data in one `read()` call or in multiple `read()` calls. It may read 4 bytes, 15 bytes, 78 bytes, 10 bytes, and 21 bytes in five successive `read()` calls. The number of bytes returned by a `read()` call depends on several factors, including the delays caused by the underlying network, the size of the datagrams in the underlying intranet, and the buffer space available associated with the sender- and receiver-side sockets. Thus, data from a `SOCK_STREAM` socket should be read in a loop until the desired amount of data has been received. The following code snippet shows one way of reading N bytes from a `SOCK_STREAM` socket and displaying the data on the screen.

```
char buf[N];
    int n, nread, nremaining;

    for (nread=0, n=0; nread < N; nread += n) {
        nremaining = N - nread;
        n = read(s, &buf[nread], nremaining);
        if (n == -1) {
            perror("read failed");
            exit(1);
        }
```

```
}
(void) write(1, buf, N);
```

The UDP protocol is used for communication between `SOCK_DGRAM` sockets. It is a "best-effort delivery service" protocol, under which messages are transmitted and received in terms of datagrams with fixed boundaries. Thus, a reader process tries to read the entire data sent by a writer process using a single `read()` system call. The process either reads the whole datagram or, in case of an error, does not read any data. The following code may be used to read from a `SOCK_DGRAM` socket.

```
n = read(s, buf, N);
if (n == -1) {
    perror("read failed");
    exit(1);
}
(void) write(1, buf, n);
```

W20.6.5 Socket Address

The address (or name) of a socket is dependent on the socket domain. When a socket is created, it does not have any address. Until an address is assigned to a socket, it may not be referred to. For a Linux domain (`PF_LOCAL` or `PF_UNIX`) socket, the address is a pathname in the file system. For an Internet domain (`PF_INET` or `PF_INET6`) socket, the address consists of two parts: the *IP address* of the host on which the socket is created and a *port number*. As discussed in Chapter 11, an IP address is used to uniquely identify a host on the Internet. IPv4 addresses are 32 bits long and IPv6 addresses are 128 bits long. On a host with a given IP address and running multiple servers, a port number allows the operating system kernel of the host to direct an incoming client request to a particular server on the host. Thus, the <IP address, port number> pair uniquely identifies the location of a service on the Internet, offered using different transport-layer protocols such as TCP and UDP.

A *port number* is a positive integer in the range 0 to 65,535. The purpose of a port number is to distinguish different services offered on a host. The International Assigned Numbers Authority (IANA) assigns port numbers and service names. Service names are assigned on a first-come, first-served basis and port numbers are assigned according to a particular scheme described in RFC6335 (see Table W20.26). Table W20.12 describes the general scheme used by IANA for the allocation of ports.

Table W20.12 Port Numbers and Their Purpose

Port Number Range	**Purpose**
0–1,023	System/well-known Ports
1,024–49,151	User/registered Ports
49,152–65,535	Dynamic/private Ports

According to RFC6335, *system ports* and *user ports* are assigned by IANA using different processes. *Dynamic ports* are never assigned and may be used by any process randomly. System ports are used to offer well-known services and are, therefore, also known as *well-known ports*. Normally, the well-known ports are offered at the same port number regardless of the transport-level protocol used by the service (TCP, UDP, etc.). Note that only a superuser (i.e., **root**) can use ports < 1,024. A few well-known services and their respective port numbers are listed in Table W20.13.

Table W20.13 Some Well-Known Services and Their Ports

Well-Known Service	**Port**
ECHO	7
DAYTIME	13
QOTD (Quote-of-the-Day)	17
FTP-DATA	20
FTP	21
SSH	22
TELNET	23
SMTP	25

TIME	37
FINGER	79
HTTP, WWW	80
KERBEROS	88
POP3	110
SFTP	115

Typically, client processes and unknown/private servers use dynamic ports. These ports are also used to test server processes.

Exercise W20.6

Browse the IANA web site. What is the total number of services that have been assigned port numbers?

Exercise W20.7

Browse through the **/usr/include** directory. How many communication domains and service types have been defined for socket? Which file(s) contains this information?

W20.6.6 Important Data Structures and Related Function Calls

Linux provides many functions to network programmers to manipulate IP addresses. Most of these functions use data structures for storing socket names, that is, IP addresses, port numbers, address sizes, and other related information. A few are used to maintain information about a host on the Internet and an Internet service.

W20.6.6.1 Important Data Structures for IP Addresses, Hosts, and Services

A brief summary of the most important data structures for network-based IPC and network programming is given in Table W20.3. These data structures deal with socket addresses, information about hosts on the Internet, and information about Internet services. We discuss the use of these data structures and their internal detail in this section. The system calls, library calls, and macros that deal with them are discussed in the sections that follow.

`struct sockaddr`

This generic structure holds information about a socket's address. It is the basic template on which other address data structures used for storing addresses of sockets of different domains are based. Here is how the structure is defined.

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

The `sa_family` field contains the socket address family and the `sa_data` contains the actual socket address. The value of `sa_data` is interpreted based on the value of `sa_family`. The address family used for Linux domain sockets is `AF_LOCAL` (or `AF_UNIX`). For Internet domain sockets, it is `AF_INET` or `AF_INET6`. When `sa_family` is `AF_LOCAL`, the `sa_data` field is supposed to contain a pathname as the socket's address. When `sa_family` is `AF_INET`, the `sa_data` field contains both an IP address and a port number. The `sockaddr_in` structure is specifically used for this purpose.

`struct sockaddr_un`

The address structure for the address of a Linux domain socket is defined as

```
struct sockaddr_un {
    unit8_t      sun_len;
    sa_family_t  sun_family;
    char         sun_path[104];
};
```

Here, sun_len is the length of sockaddr_un including the NULL byte, sun_family is AF_LOCAL (or AF_UNIX), and sun_path[104] is the null-terminated pathname in the file system structure that refers to the socket.

struct sockaddr_in

This structure may be used to hold the address information for an Internet domain socket. It is defined as follows:

```
struct sockaddr_in {
    uint8_t sin_len;
    sa_family_t sin_family;     /* short int */
    in_port_t int sin_port;     /* unsigned short */
    struct in_addr sin_addr;
    unsigned char sin_zero[8];
};
```

The sin_family field specifies the address family; usually this is AF_INET. The sin_port and sin_addr fields contain a 16-bit port number and a 32-bit IPv4 address in network byte order, respectively. The sin_zero field is set to NULL (i.e., '0') as it is not used. The in_addr structure is defined as follows.

```
struct in_addr {
    unsigned long s_addr;
};
```

struct hostent

The hostent structure may be used to hold official information about a host on the Internet. It is defined as follows:

```
struct hostent
{
    char *h_name;
    char **h_aliases;
    int h_addrtype;
    int h_length;
    char **h_addr_list
#define h_addr h_addr_list[0] /* For backward compatibility */
};
```

The meaning of each field of the structure is given in Table W20.14.

Table W20.14 Meaning of Each Field of the hostent Structure

Field	Purpose
h_name	Official name of host
h_aliases	NULL-terminated array of other names of host
h_addrtype	Address type (family) of host, usually AF_INET (currently defined as PF_INET)
h_length	Length (in bytes) of address; 4 bytes for IPv4 and 16 bytes for IPv6
h_addr_list	NULL-terminated array of network addresses for host
h_addr	Used for backward compatibility; first address in h_addr_list

struct servent

The servent structure may be used to hold official information about a host on the Internet. It is defined as follows:

```
struct servent
{
    char *s_name;
    char **s_aliases;
    int s_port;
    char *s_proto;
};
```

The meaning of each field of the structure is given in Table W20.15.

Table W20.15 Meaning of Each Field of the `servent` Structure

Field	Purpose
`h_name`	Official name of host
`h_aliases`	NULL-terminated array of other names of host
`h_addrtype`	Address type (family) of host, usually `AF_INET` (currently defined as `PF_INET`)
`h_length`	Length (in bytes) of address; 4 bytes for IPv4 and 16 bytes for IPv6
`h_addr_list`	NULL-terminated array of network addresses for host
`h_addr`	Used for backward compatibility; first address in `h_addr_list`

W20.6.6.2 Library Functions to Manipulate IP Addresses

Linux provides several library functions for manipulating IP addresses from ASCII, that is, strings in DDN to binary and vice versa. These functions are listed in Table W20.2. Here, we briefly discuss these calls and demonstrate their use with small code snippets.

A brief description of the `inet_aton()` call is given next. Note that it works for address domains `PF_INET` and `PF_INET6`.

```
#include <netinet/in.h>
#include <arpa/inet.h>
int inet_aton(const char *cp, struct in_addr *pin);
```

Success: 1
Failure: 0 if address string is invalid

This function call converts the IP address in DDN specified as string `cp` to the IP address in network byte order and stores it in the structure specified as `pin`. The following code snippet shows a sample use of this function.

```
#include <netinet/in.h>
#include <arpa/inet.h>
...
int n;
struct in_addr address;
...
memset(&address, '\0', sizeof(address));
n = inet_aton("39.59.169.110", &address);
if (n == 0) {
   /* Error handling code */
}
...
```

The `inet_addr()` call converts and returns the specified string for an IP address in DDN to a 32-bit unsigned binary value in network byte order. Here is a brief description of the call.

```
#include <sys/types.h>
#include <arpa/inet.h>
in_addr_t inet_addr(const char *cp);
```

Success: IP address in Network Byte Order
Failure: `INADDR_NONE`

The return type of the function, `in_addr_t`, is defined as a 32-bit unsigned integer. In the following code segment, we show a sample use of this call.

```
#include <sys/types.h>
#include <arpa/inet.h>
...
struct sockaddr_in address;
...
memset(&address, '\0', sizeof(address));
if ((address.sin_addr.s_addr = inet_addr("39.59.169.110")) == INADDR_NONE) {
```

```
    /* Error handling code */
}
...
```

The call returns `INADDR_NONE` on failure. This symbolic constant is defined as all 1s (i.e., `0xffffffff`). This means that this return value is `-1`.

The `inet_ntoa()` call converts and returns an IP address in network byte order binary form to the string of corresponding IP address in DDN. Here is a brief description of the call.

`#include <sys/types.h>` `#include <arpa/inet.h>`
`char * inet_ntoa(struct in_addr in);`
Success: IP address string in DDN **Failure:** `NULL`

The following piece of code shows a sample use of the call.

```
#include <arpa/inet.h>
#include <arpa/inet.h>
...
char *ip_ddn;
struct in_addr address;
...
ip_ddn = inet_ntoa(address);
printf("IP Address is: %s\n",ip_ddn);
...
```

W20.6.6.3 New Library Functions to Manipulate IP Addresses

The `inet_addr()` function received criticism and new programs used `inet_aton()` instead. The new versions of `inet_ntoa()` and `inet_aton()` that work with both IPv4 and IPv6 addresses are `inet_ntop()` and `inet_pton()`, respectively. Note that "n" stands for numeric and "p" for presentation. You should use the new calls in your code even if your system does not support IPv6.

Here are brief descriptions of the `inet_ntop()` and `inet_pton()` functions.

`#include <sys/types.h>` `#include <arpa/inet.h>`
`const char * inet_ntop(int af, const void * restrict src, char * restrict dst, socklen_t size);`
Success: IP address string in DDN **Failure:** `NULL` on system error; `-1` the specified address family not supported

`#include <sys/types.h>` `#include <arpa/inet.h>`
`int inet_pton(int af, const char * restrict src, void * restrict dst);`
Success: 1 **Failure:** `0` if input is not in a valid presentation format and `-1` on error

The first argument in both functions, `af`, stands for address family. Currently, only `AF_INET` and `AF_INET6` are supported. These functions return `-1` if the specified address family is not supported and `errno` variable is set to `EAFNOSUPPORT`. The size argument in `inet_atop()` is the size in bytes of the destination. It is used to prevent overflow of caller function's buffer. If the `size` argument is too small to store the address in the resultant presentation, the function returns `NULL` and `errno` is set to `ENOSPC`. To prevent this problem, the following constants should be used.

```
#define INET_ADDRSTRLEN     16     /* for IPv4 dotted-decimal */
#define INET6_ADDRSTRLEN    46     /* for IPv6 hex string */
```

Now, we show a few examples of the old calls and their equivalents using the new calls. The first line shows the old call and the second (and third) shows its equivalent of the new call.

```
address.sin_addr.s_addr = inet_addr("39.59.169.110");
inet_pton(AF_INET, "39.59.169.110", address.sin_addr);
```

```
inet_aton("39.59.169.110", &address);
inet_pton(AF_INET, "39.59.169.110", address.sin_addr);
```

```
ip_ddn = inet_ntoa(address);
char dest[INET_ADDRSTRLEN];
ip_ddn = inet_ntop(AF_INET, &address, dest, sizeof(dest));
```

Exercise W20.7

Browse through the header files that contain the data structures that we have discussed in this section.

Exercise W20.8

Write small programs to test the use of the various library calls and macros that we have discussed in this section. Show compilation and execution of your programs along with their outputs.

W20.6.7 Binding an Address to a Socket

When a socket is created, it belongs to a particular protocol domain but does not have any protocol address assigned to it. The protocol address of a socket is also known its *address* or *name*. If no process would refer to a socket, then it is not necessary for such a socket to have an address. For example, if a socket was created in a client process, then, most likely, no other process would refer to it. Thus, binding an address to a client-side socket is not necessary. If other processes would need to refer to a socket, it is necessary that it has an address. A socket created by a server process requires that an address be bound (i.e., assigned) to it. This address is advertised for the service that the server process offers so that a client process could contact the server process using the address of the server-side socket. Figure W20.12 shows a server process with three sockets having addresses and a client process with a socket without an address assigned to it. The sockets in the server process are of the types `PF_LOCAL`, `PF_INET` with `SOCK_STREAM` type of communication, and `PF_INET` with `SOCK_DGRAM` type of communication. The client socket does not have an address.

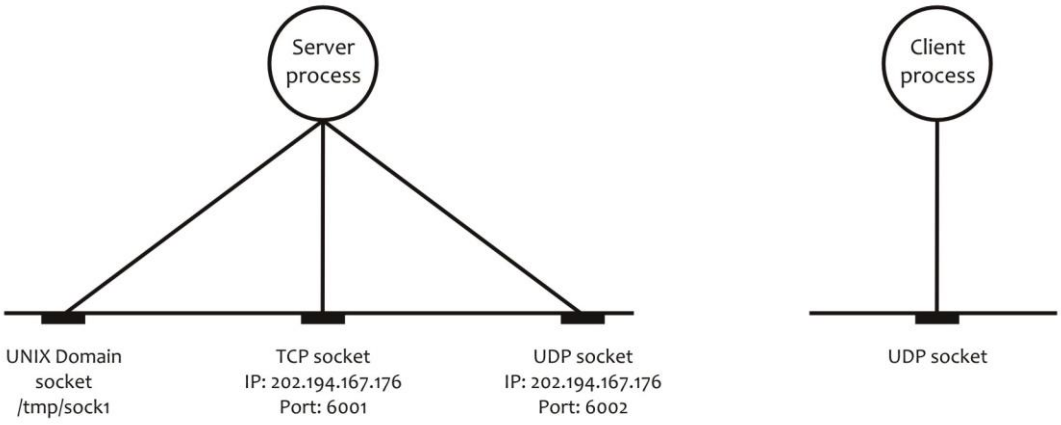


FIGURE W20.12 Sockets with and without addresses bound to them: (a) server process with three sockets with addresses; a `PF_LOCAL` socket, a `PF_INET` socket of `SOCK_STREAM` type, a `PF_INET` socket of `SOCK_DGRAM` type; and (b) client process with `PF_INET` socket of `SOCK_DGRAM` type without address.

You can use the `bind()` system call to assign an address to a socket. Here is a brief description of the `bind()` system call.

`#include <sys/types.h>` `#include <sys/socket.h>`
`int bind(int s, const struct sockaddr *addr, socklen_t addrlen);`
Success: 0 **Failure:** `-1` and kernel variable `errno` set to indicate the type of error

Here, `s` is a socket descriptor, `addr` is the address to be bound to `s`, and `addrlen` is the length of `addr`. For maximum portability, the `addr` field must be initialized to zero.

The `bind()` call may fail for various reasons. Some of the commonly occurring reasons are listed in Table W20.16.

Table W20.16 Common Reasons for the `bind()` System Call to Fail

Reason for Failure	**Value of** `errno`
Sufficient kernel resources not available to complete the request	`EAGAIN`
The `s` argument is not a valid descriptor	`EBADF`
The `s` argument represents a socket that has been shut down (or closed) or is already bound to an address	`EINVAL`
The `s` argument does not represent a socket	`ENOTSOCK`
The `addr` argument represents an address that is already in use	`EADDRINUSE`
The `addr` argument is not within the process address space	`EFAULT`

In case of the UNIX domain socket, the `bind()` system call would fail for the reasons that the `open()`, `creat()`, `mkfifo()`, and `mkfifoat()` system calls would fail, as shown in Tables 15.4 and W20.7. When a UNIX domain socket is no longer required, it must be removed from the system using the `unlink()` system call. The address of a socket cannot be read/written by anyone other than the processes with which the socket is associated.

The following code snippet shows how you can use the `bind()` system call to bind address to an Internet domain socket.

```
#define PORT  6001
...
struct sockaddr_in saddr, caddr;
...
/* Initialize socket structure */
memset(&saddr, 0, sizeof(saddr));
saddr.sin_family =  AF_INET;
saddr.sin_addr.s_addr =  INADDR_ANY;
saddr.sin_port =  htons(PORT);

/* Bind address to socket */
if (bind(s, (struct sockaddr *)&saddr, sizeof(saddr)) == -1) {
    perror("bind failed");
    exit(1);
}
```

Note that the address structure variable must be properly initialized before using it in the `bind()` call. The symbolic constant `INADDR_ANY` is a wildcard IP address that matches any of the IP addresses of the host on which the server process runs. Thus, in case of *multihomed hosts*, that is, hosts that are connected to multiple networks and, hence, have multiple IP addresses, the use of the wildcard address `INADDR_ANY` makes it possible for the server to accept connection requests from clients arriving at any of these IP addresses. The `htons(PORT)` function is used to convert the `PORT` value from the host network byte order to network byte order before assigning the port value to the `sin_port` field of the `saddr` address variable. Only a superuser (i.e., **root**) can bind to ports < 1,024.

Exercise W20.9

Write a program that creates a UNIX domain socket, assigns a name to it, and displays the socket descriptor and its address using the UNIX address variable.

W20.6.8 Enabling a Server-Side Socket to Listen for Connection Requests from Clients

Once a server-side stream-oriented socket has been assigned an address, it needs to be put in passive (listening) mode before client processes may connect to and communicate with it. You can put a socket in passive mode using the `listen()` system call. Once the `listen()` call has been called on a socket, it starts to listen for incoming connection requests from client processes. Such a socket is known as a *passive socket*. Once a socket has been placed in passive mode, a client request for connection may be accepted by it.

Here is a brief description of the `listen()` system call.

`#include <sys/types.h>` `#include <sys/socket.h>`
`int listen(int s, int backlog);`
Success: 0 **Failure:** `-1` and kernel variable `errno` set to indicate the type of error

Here, `s` is a socket descriptor and `backlog` specifies the maximum length of the queue where client connection requests may wait. According to the manual page for the `listen()` call, the real maximum queue length is 1.5 times what the programmer specifies as `backlog`. The maximum length of this queue is defined as `SOMAXCONN`. This value of `SOMAXCONN` is 128 on our Linux system. Several well-known servers use the maximum queue length with their passive sockets. You can use the `netstat -aL` command to determine the queue lengths associated with all the servers running on your system.

The `listen()` call may fail for the reasons listed in Table W20.17.

Table W20.17 Reasons for `listen()` to Fail

Reason for Failure	**Value of** `errno`
`s` is not a valid descriptor	`EBADF`
`s` is not a socket descriptor	`ENOTSOCK`
`s` is already connected or is in the process of being connected	`EINVAL`
`s` is of a type that does not support the `listen()` operation	`EOPNOTSUPP`

The following code snippet shows a typical use of the `listen()` system call. Note that the queue length associated with the passive socket is 5.

```
#define QLEN  5
...
/* Put socket in passive mode */
if (listen(s, QLEN) == -1) {
    perror("listen failed");
    exit(1);
}
...
```

W20.6.9 Sending a Connection Request to Server Process

In the connection-oriented style of communication, the server- and client-side sockets are first connected, and only then does communication start between the two processes through their respective sockets—both processes do I/O with their own sockets. Once the server-side stream-oriented socket has been assigned an address and put in passive mode, it is ready to receive and accept client requests to establish connection. A client process can send a connection request using the `connect()` system call.

Here is a brief description of the `connect()` system call.

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int s, const struct sockaddr *name, socklen_t namelen);
```

Success: 0

Failure: −1 and kernel variable `errno` set to indicate the type of error

For maximum portability, the `addr` field must be initialized to zero.

When used with the `SOCK_STREAM` type of socket, the `connect()` call performs the three-way TCP handshake to connect the client- and server-side sockets. After the call has completed its execution, the underlying data structure associated with each socket has the address of the other side and a full association has been established between the two sockets. This also means that a *virtual connection* has been established between the client and server processes. This only happens through the rendezvous of the `connect()` and `accept()` system calls. We discuss the `accept()` system call in the next section. Once a connection has been established between the client- and server-side sockets, the client and server processes can communicate with each other using the `read()` and `write()` system calls.

When used with the `SOCK_DGRAM` type of socket, the `connect()` call simply stores the address of the remote socket in the local socket's data structure. No handshake takes place between the local and remote sockets. This means that after a UDP client has made a `connect()` call on a socket, it may communicate with the other side using `read()` and `write()` instead of using `recvfrom()` and `sendto()`.

Normally, `connect()`is called only once to successfully establish full association between two `SOCK_STREAM` sockets. However, `connect()` may be called multiple times to change associations between two `SOCK_DGRAM` sockets. An association of a `SOCK_DGRAM` socket may be dissolved by using `connect()` on it with an invalid address, such as a null address.

The `connect()` call may fail for several reasons. Table W20.18 shows some of the reasons that the `connect()` call may fail for non-UNIX domain—usually `PF_INET` or `PF_INET6`—sockets.

Table W20.18 Some Reasons for `connect()` to Fail

Reason for Failure	**Value of** `errno`
`s` is not a valid descriptor	`EBADF`
`s` is not a socket descriptor	`ENOTSOCK`
Address specified in `name` is not available on the computer	`EADDRNOTAVAIL`
Address specified in `name` cannot be used with this socket	`EAFNOSUPPORT`
`s` is already connected	`EISCONN`
Host is not connected to the network	`ENETUNREACH`
Remote host is not reachable from this host	`EHOSTUNREACH`
Address specified in `name` is already in use	`EADDRINUSE`
A signal interrupted the connection establishment	`EINTR`
A previous connection request has not yet been completed	`EALREADY`
A broadcast address is specified using `INADDR_BROADCAST` or `INADDR_NONE` for a socket that does not support broadcast functionality	`EACCES`

Failure of the `connect()` call for a UNIX domain socket has mostly to do with invalid pathname issues, as discussed for system calls such as `open()` and `creat()`. Two additional reasons for failure of the `connect()` call for a UNIX domain socket are (a) the socket `s` does not exist and (b) write access for `s` is denied.

The following piece of code shows how you can use the `connect()` system call to send a connection request to a connection-oriented server process to establish a connection with it. We assume that, as is the case with all production clients, the name of the host or its IP address in DDN and the port number on which the server process runs are passed as the first and second command line arguments to the client program.

```
...
struct sockaddr_in saddr;
struct hostent *server;
...
if ((server = gethostbyname(argv[1])) == NULL) {
   printf("No such host\n");
   exit(0);
}

memset(&saddr, 0, sizeof(saddr));
saddr.sin_family = AF_INET;
saddr.sin_port = htons(atoi(argv[2]));
if (server = gethostbyname(argv[1]))
   memcpy(&saddr.sin_addr.s_addr, server->h_addr, server->h_length);
else
   if ((saddr.sin_addr.s_addr = inet_addr(argv[1])) == INADDR_NONE) {
       printf("No such host\n");
       exit(1);
   }

/* Send connection request to the server process */
if ((connect(s, &saddr, sizeof(saddr))) == -1) {
   perror("connect failed");
   exit(1);
}
...
```

Note that `gethostbyname()` returns a pointer to the `hostent` structure, which contains relevant information about the host, including its address and address length, as discussed in Section W20.6.6. We copy the address of the host to the relevant field of the address variable for the server-side socket, `saddr`, using the `bcopy()` or the newer `memcpy()` function. The `atoi()` function received criticism and the newer recommended replacement is `strtol()`. We use the `strtol()` function in the client–server software that we design and develop in this chapter. You can also replace

```
saddr.sin_addr.s_addr = inet_addr(argv[1]);
```

with the following call:

```
inet_pton(AF_INET, argv[1], address.sin_addr);
```

W20.6.10 Accepting a Client Request for Connection

Whether it is used on an *iterative server* or a *concurrent server*, the passive socket may only be used to wait for client requests for connections and not for client–server communication. This is so because, after accepting a client's request for connection and before starting communication with the client, the server process needs to go back and listen for more incoming connection requests from clients. Thus, a client-side socket does not communicate with the passive socket on the server side. Instead, a new socket is created on the server side that is connected with the client-side socket for communication between the two processes. Because it is used to interact with the client process, the newly created socket is also known as the *active socket*. As stated earlier, this new socket on the server side is created and connected to the client-side socket through the rendezvous of the `accept()` call on the server side and the `connect()` call on the client side.

The main server socket—the passive socket—remains allocated throughout the life of a server process. However, the sockets created by `accept()` have the lifespan of a connection between a client and the server process. For this reason, they are also known as *ephemeral sockets*.

A variant of the `connect()` call is `accept4()`. Here is a brief description of the `accept()` and `accept4()` system calls.

`#include <sys/types.h>` `#include <sys/socket.h>`
`int accept(int s, struct sockaddr * restrict addr, socklen_t * restrict addrlen);`
`int accept4(int s, struct sockaddr * restrict addr, socklen_t * restrict addrlen,`

int flags);
Success: Socket descriptor for the accepted socket **Failure:** -1 and kernel variable errno set to indicate the type of error

Here, s is the server-side passive socket, addr is the address variable, and addrlen is the length of the address variable in bytes. The call returns the descriptor for the newly created socket, which inherits the properties of s for nonblocking and asynchronous I/O (O_NONBLOCK and O_ASYNC), and settings of the I/O and urgent signals (SIGIO and SIGURG). For the accept4() call, the nonblocking I/O property is specified by using the SOCK_NONBLOCK flag in the flags argument. The "close-on-exec" property of the newly created socket is set using the SOCK_CLOEXEC flag in the flags argument. The signal settings and asynchronous I/O properties of the newly created socket are cleared. For maximum portability, the addr field must be initialized to zero.

The accept() and accept4() calls may fail for several reasons. Table W20.19 lists some of the reasons for these calls to fail.

Table W20.19 Some Reasons for accept() and accept4() to Fail

Reason for Failure	**Value of** errno
s is not a valid descriptor	EBADF
A signal interrupted the accept operation	EINTR
The PPFDT or the system-wide file table is full	EMFILE
s is not a socket descriptor	ENOTSOCK
s is not a passive socket, i.e., the listen() system call has not been executed on s. In the case of the accept4() call, the flags argument is invalid	EINVAL
addr is not the writable part of the address space of the caller process	EFAULT
s is nonblocking with no connection requests waiting to be accepted	EWOULDBLOCK

The following code snippet shows how you can use the accept() system call to wait, listen for, and accept connection requests from an Internet client through its socket.

```
...
int caddrlen;
struct sockaddr_in caddr;
...
/* Obtain length of client's address variable */
caddrlen = sizeof(caddr);
/* Block, listen for, and accept a connection request from a client */
if ((sock = accept(s, (struct sockaddr *)&caddr, &caddrlen)) == -1) {
    perror("accept failed");
    exit(1);
}
...
```

Figure W20.13 shows the pictorial view of the complete client–server setup for the SOCK_STREAM type of communication using a single-server process. The figure clearly shows the sequence of steps that the client and server processes have to take for the establishment of a virtual connection between the client- and server-side sockets. We discuss similar setups for multiprocess servers later in the chapter.

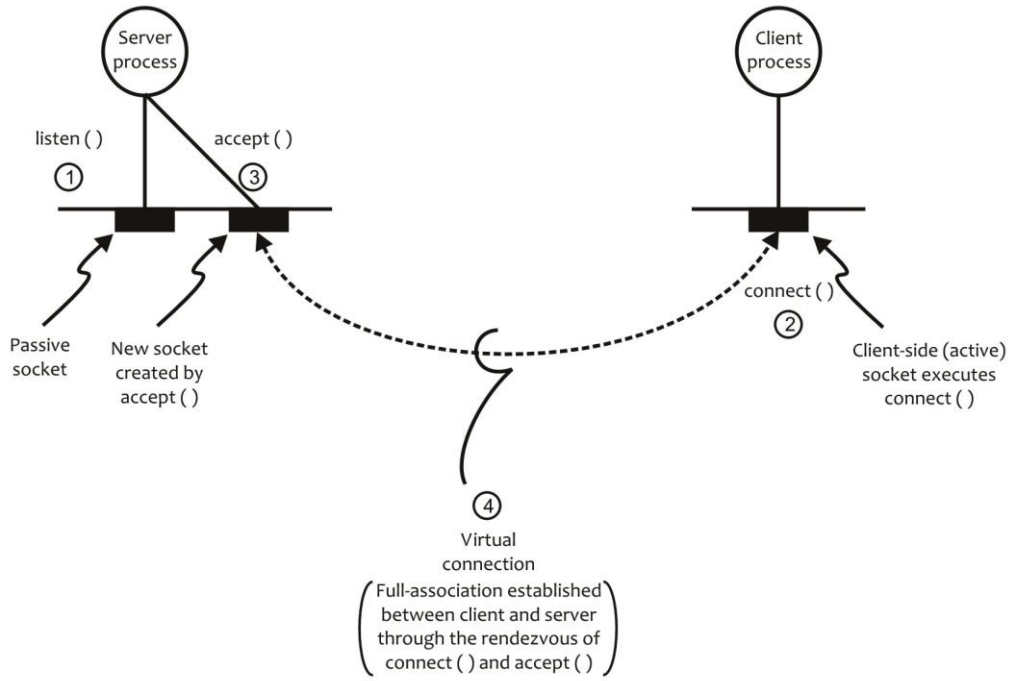


Figure W20.13 View of the passive socket and virtual connection between client and server sides.

W20.6.11 Closing a Socket

When a socket is no longer needed, it should be closed so that its descriptor and associated kernel resources may be reused. The `close()` or `shutdown()` system calls may be used to close a socket. The `close()` system call closes the socket completely, whereas the `shutdown()` call may be used to close a socket for input (read) only, output (write) only, or both input and output.

Here is a brief description of the `shutdown()` system call.

`#include <sys/types.h>` `#include <sys/socket.h>`
`int shutdown(int s, int how);`
Success: 0 **Failure:** `-1` and kernel variable `errno` set to indicate the type of error

Table W20.20 shows the different values for the `how` parameter and its effect on the socket `s`.

Table W20.20 Possible Values of the `how` Parameter and Its Effect on Socket `s`

Value of `how`	Effect on Socket `s`
`SHUT_RD`	No input (i.e., read) operation may be performed on socket `s`
`SHUT_WR`	No output (i.e., write) operation maybe performed socket `s`
`SHUT_RDWR`	No I/O may be performed on socket `s`

The `shutdown()` call is used when a process has completed either input or output, but not both. For example, a client process can close its socket for output after sending its last request to the server process. The server process may close its socket for both input and output after it has sent its response to the last client request it has received. Finally, the client process closes the socket for input after receiving the response to its last request.

W20.6.12 Putting It All Together: A Simple Connection-Oriented Client–Server Software

We now put together the code snippets shown for the various system calls and build simple connection-oriented client–server software. This service is similar to the well-known ECHO service, except that the server process does not run at the well-known port 7, and terminates after serving one client request.

W20.6.12.1 Design of a Server Process

In the design of our code for the client and server processes, we write some generic functions that may be used in the TCP and UDP client and server processes. The server processes we discuss in this section run with two command line arguments, the transport-level protocol for which the service is offered (`tcp`, `udp`, etc.) and the protocol port at which the service is offered. Thus, the syntax for the execution of a server process is shown next.

```
server-name transport-protocol protocol-port
```

If `transport-protocol` or `protocol-port` is invalid, the program displays an error message and terminates.

The first function that we will design is `CreatePassiveSock()`. It takes three arguments: a transport protocol, a protocol port, and the queue length for the passive socket. The protocol and port are passed to the server process as command line arguments. The server supports only TCP and UDP as transport-level protocols. It creates a socket for the type of communication for the given protocol, binds a name to the socket, and puts the socket in passive mode if it is a TCP socket. Finally, it returns the descriptor for the socket as its return value.

As discussed earlier in Section W20.6.7, the wild card IP address `INADDR_ANY` makes it possible for the server to accept connection requests from clients at any of the IP addresses of a multihomed host. Note that we use the `strtol()` library call to convert a port number string to an integer and make sure to exit if a string of nondigits is passed as the port number at the command line. The criticized `atoi()` function would not work properly because it converts some character strings of nondigits into valid port numbers.

The second function, `EchoServerTCP()`, is specific to the service to be offered by the server process. It takes an active socket created by the `accept()` system call as an argument, reads data from the client process, and writes it back to the client process. Normally, we would read data from a TCP socket in a loop. However, to keep things simple, we read client data using a single `read()` call. If you want to make it a production server, you must read and write data in a loop as discussed in Section W20.6.4.

Here is the code for the server process saved in the **echo_server_TCP.c** file.

```
$ cat echo_server_TCP.c
/*
Usage: Server-name Protocol Port
Here, Protocol is the transport level protocol
and Port is the procotol port number where the
service is to be offered.
*/

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

#define BUFF  256
#define QLEN  16

int main(int argc, char *argv[])
{
    int psock, asock, caddrlen;
    struct sockaddr_in caddr;

    /* Exit if program is not run with two command line arguments. */
    if (argc != 3) {
       printf("Usage: server protocol port\n");
       exit(1);
    }
```

```
    /* Create a TCP socket, bind name to it, and put it in passive mode */
    psock = CreatePassiveSock(argv[1], argv[2], QLEN);

    caddrlen = sizeof(caddr);
    while (1) {
       /* Accept actual connection from the client */
       if ((asock = accept(psock, (struct sockaddr *)&caddr, &caddrlen)) == -1) {
           perror("accept failed");
           exit(1);
       }

       /* TCP echo service code */
       EchoServerTCP(asock);

       /* Close active socket */
       close(asock);
    } /* while */
}

/* Create a TCP or UDP socket, bind a name to it, and put it in    */
/* passive mode if it is a TCP socket.                             */
/* 'protocol' is transport layer protocol ("tcp", "udp", etc).     */
/* 'portptr' is pointer to port number as a character string.      */
/* 'qlen' is the queue length associated with the passive socket.  */
int CreatePassiveSock(char *protocol, char *portstr, int qlen)
{
    int s, port, type, saddrlen;
    char *endptr;
    struct sockaddr_in saddr;

    /* Convert portstr to port number as integer. Display  */
    /* error message and exit if portstr is not a number.  */
    port = (int) strtol(portstr, &endptr, 10);
    if (*endptr) {
        printf("\nPlease specify a positive integer for port.\n");
        exit(1);
    }

    /* Initialize socket structure */
    saddrlen = sizeof(saddr);
    memset(&saddr, 0, saddrlen);
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = INADDR_ANY;
    saddr.sin_port = htons(port);

    if (strcmp("tcp", protocol) == 0)
       type = SOCK_STREAM;
    else if (strcmp("udp", protocol) == 0)
       type = SOCK_DGRAM;
    else {
       printf("Unsupported protocol\n");
       exit(1);
    }

    /* Create a TCP or UDP socket for IPv4 */
    if ((s = socket(PF_INET, type, 0)) == -1) {
        perror("socket call failed");
        exit(1);
    }

    /* Bind address to socket */
    if (bind(s, (struct sockaddr *)&saddr, saddrlen) == -1) {
        perror("bind failed");
        exit(1);
    }

    /* If it is a TCP socket, put it in passive mode, */
    /* i.e., ready to listen for incoming connection  */
    if (type == SOCK_STREAM) {
       if (listen(s, qlen) == -1) {
            perror("listen failed");
            exit(1);
```

```
        }
    }

    /* Return the TCP passive or UDP socket with a   */
    /* name bound to it.                             */
    return s;
}

/* Provide echo service to a client. 'sock' is the     */
/* active socket connected to the client-side socket.  */
void EchoServerTCP(int sock)
{
    int nr, nw;
    char buf[BUFF];

    /* Communicate with client: read and write back */
    memset(buf, 0, BUFF);
    if ((nr = read(sock, buf, BUFF-1)) == -1) {
       perror ("socket read error");
       exit(1);
    }

    /* Write back (echo) the same data to client */
    if ((nw =  write(sock, buf, nr)) == -1) {
        perror("socket write error");
        exit(1);
    }
}
$
```

First, the program makes sure that it has been run with the correct number of arguments. Second, it verifies that the second argument, `port_number`, is a number. It then calls the `CreatePassiveSock()` function to create a passive socket in the case of the TCP protocol. Finally, it starts an infinite loop and blocks on the `accept()` call. As soon a client request arrives and has been accepted, an active socket is created with its descriptor in `asock`. The program then calls `EchoServerTCP()` to service the client request, passing it the active socket's descriptor as an argument. This function serves the client and returns. On return from this function, the program closes the active socket and blocks on `accept()` again, waiting for the next connection request.

Here is the compilation and a sample run of the client–server model. Note that, in our sample run, we offer the TCP ECHO service at port 6001. The output of the `netstat -ltnp | grep -w '6001'` command shows our server running on port 6001. The last fieled of the output of the command shows the process ID (PID) and the name of the process. Since we have not yet written the code for the client process corresponding to this server, we test it with a **telnet** client. The **telnet** client sends the text entered by the user from the keyboard (shown in boldface) to the server process, the server process receives it, sends it back to the **telnet** client, deallocates the active socket, and goes back to wait for another client's connection request. The **telnet** client receives the text from the server process, displays it on the screen, and terminates. You can use the loopback address (127.0.0.1) with the `telnet` command instead of using the IP address of the host (202.147.169.195), because the client and server processes run on the same host. The last `ps` command is used to show that, after serving a client, the server continues to run, as expected. The `kill` command is used to terminate the server process.

```
$ gcc echo_server_TCP.c -w -o tcpechos
$ tcpechos tcp 6001 &
[1] 32080
$ ps
UID        PID  PPID  C STIME TTY          TIME CMD
sarwar   31759 31758  0 13:36 pts/2    00:00:00 -bash
sarwar   32080 31759  0 14:11 pts/2    00:00:00 tcpechos tcp 6001
sarwar   32130 31759  0 14:15 pts/2    00:00:00 ps -f
$ netstat -ltnp | grep -w '6001'
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
tcp     0    0 0.0.0.0:6001       0.0.0.0:*           LISTEN        32080/tcpechos
$ telnet 202.147.169.195 6001
Trying 202.147.169.195...
Connected to 202.147.169.195.
```

```
Escape character is '^]'.
Hello, world!
Hello, world!
Connection closed by foreign host.
$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
sarwar   31759 31758  0 13:36 pts/2    00:00:00 -bash
sarwar   32080 31759  0 14:11 pts/2    00:00:00 tcpechos tcp 6001
sarwar   32130 31759  0 14:15 pts/2    00:00:00 ps -f
$ kill -9 32080
$
```

Exercise W20.10

Repeat this session on your Linux system to verify that it works as expected.

W20.6.12.2 Design of a Client Process

The client processes that we discuss in this chapter run with three command line arguments, the IP address in DDN or domain name of the server process, the protocol port at which the service is offered, and the transport-level protocol for which the service is offered (`tcp`, `udp`, etc.). Thus, the syntax for the execution of a client process is shown next.

```
client-name ip-address (or domain name) protocol-port transport-protocol
```

If any of the command line arguments is invalid, the program displays an error message and terminates.

The first function that we design for the client software is `CreateConnectedSock()`, which takes the three arguments that you pass to the client process as command line arguments. It converts the port number from a string to an integer using the `strtol()` library call and initializes the address variable for the server process. It then creates a TCP or UDP socket, depending on the value of the transport-level protocol. Next, it establishes a connection with the server-side socket using the `connect()` system call and returns the socket descriptor. As has been discussed earlier, in the case of the UDP client–server software, no connection is established between the client- and server-side processes. However, the server's address is stored in the client-side socket's data structure. This allows the client to communicate with the server process using the `read()` and `write()` system calls, instead of the `sendto()` and `recvfrom()` system calls.

The second function, `EchoClientTCP()`, handles the client side of the ECHO service. It takes a connected socket descriptor at the client side as an argument, prompts the user for input from the keyboard, writes the user input to the server process, reads the server's response (which is the client-side data sent back), displays it on the screen, and returns. As stated earlier, normally, we would read and write data from and to a TCP socket in a loop. However, to keep things simple for this rather trivial service, we do I/O with the socket using single `read()` and `write()` calls. If you want to make it part of a production client–server software, you must read and write data in loops as discussed in Section W20.6.4. Here is the code for the function.

Here is the code for the server process saved in the **echo_client_TCP.c** file. After making sure that the client program is run with the requisite number of command line arguments, it calls the `CreateConnectedSock()` function, which returns the descriptor for the appropriate socket connected to the server-side socket. It then calls the `EchoClientTCP()` function to perform the client-side functionality of the ECHO service. At the end, it deallocates the connected socket and returns.

```
$ cat echo_client_TCP.c
#include <netdb.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define BUFF    256

int main(int argc, char *argv[])
{
    int csock; /* Descriptor for connected socket */

    if (argc != 4) {
```

```
        printf("Usage: %s hostname port protocol\n", argv[0]);
        exit(0);
    }

    csock = CreateConnectedSock(argv[1], argv[2], argv[3]);

    /* Perform client-side echo using csock, the connected socket */
    EchoClientTCP(csock);

    /* Deallocate socket and return */
    close(csock);
    return 0;
}

/* Create a TCP or UDP socket, based on the third command line   */
/* argument. Connect the socket to the server whose name (IP     */
/* address and port number) is passed as command line arguments. */
/* Return the descriptor of the connected socket. In case of the */
/* UDP socket is not connected but the address of the remote     */
/* socket is stored in the local socket's data structure. This   */
/* allows us to communicate with the server using the read()     */
/* and write() system calls, instead of recvfrom() and sendto(). */
int CreateConnectedSock(char *ip, char *portstr, char *protocol)
{
    int s, port, type, saddrlen;
    char *endptr;
    struct sockaddr_in saddr;
    struct hostent *server;

    /* Convert portstr to port number as integer. Display  */
    /* error message and exit if portstr is not a number.  */
    port = (int) strtol(portstr, &endptr, 10);
    if (*endptr) {
        printf("\nPlease specify a positive integer for port.\n");
        exit(1);
    }

    saddrlen = sizeof(saddr);
    memset(&saddr, 0, saddrlen);
    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(port);
    if (server = gethostbyname(ip))
        memcpy(&saddr.sin_addr.s_addr, server->h_addr, server->h_length);
    else
        if ( !(inet_pton(AF_INET, ip, saddr.sin_addr)) ) {
            printf("No such host\n");
            exit(1);
        }

    if (strcmp("tcp", protocol) == 0)
       type = SOCK_STREAM;
    else if (strcmp("udp", protocol) == 0)
       type = SOCK_DGRAM;
    else {
       printf("Unsupported protocol\n");
       exit(1);
    }

    /* Create a TCP or UDP socket for IPv4 */
    if ((s = socket(PF_INET, type, 0)) == -1) {
        perror("socket call failed");
        exit(1);
    }

    /* Send connection request to the server process */
    if (connect(s, &saddr, saddrlen) == -1) {
        perror("connect failed");
        exit(1);
    }
    return s;
}
```

```
/* Client side of the echo service. 'sock' is the  */
/* socket connected to the server-side socket.     */
void EchoClientTCP(int sock)
{
    int n;
    char buf[BUFF];

    /* Get input from the user */
    memset(buf, 0, BUFF);
    printf("Enter text for server : ");
    fgets(buf, BUFF-1, stdin);

    /* Send message to server */
    if ((n =  write(sock, buf, strlen(buf))) == -1) {
        perror("socket write error");
        exit(1);
    }

    /* Rread server's response */
    memset(buf, 0, BUFF);
    if ((n = read(sock, buf, BUFF-1)) == -1) {
        perror ("socket read failed");
        exit(1);
    }

    /* Display server's response */
    printf("Text from server : %s", buf);
}
$
```

In the following session, we show the compilation of the client software and the running of the executable codes for the server and client processes. The client–server software runs as expected. Again, you can use the loopback address (127.0.0.1) with the client command instead of using the IP address of the host (202.147.169.195), because the client and server processes run on the same host.

```
$ gcc echo_server_TCP.c -w -o tcpechos
$ tcpechos tcp 6001 &
[1] 2663
$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
sarwar    2312  2310  0 16:24 pts/1    00:00:00 -bash
sarwar    2663  2312  0 16:46 pts/1    00:00:00 tcpechos tcp 6001
sarwar    2665  2312  0 16:46 pts/1    00:00:00 ps -f
$ tcpechoc 202.147.169.195 6001 tcp
Enter text for server : Hello, world!
Text from server : Hello, world!
$ kill -9 2663
$
```

Exercise W20.11

Repeat this session on your Linux system to verify that it works as expected.

Exercise W20.12

Run the client and server processes in different windows on the same machine and on different machines on the Internet.

Exercise W20.13

Design and code the client–server model for the ECHO service using UNIX domain sockets. Show the compilation and execution of the model on your machine.

W20.7 TYPES OF SOCKET-BASED SERVERS

The type of service to be offered dictates the design of a client–server application. The core of the client-side and server-side software is dependent on the communication between them, as outlined in the application protocol to be implemented. Such communication, barring a few services, is always based on a client sending a request to a server and the server sending a response to the client request. For some applications, such interaction is limited to one request and one response. However, for several well-known services, this request–response session continues until the client sends some sort of "quit" request to the server. When the server process has served a client, it is ready to accept a request from another client. If a single server process handles a client's requests, it is known as an *interactive server*.

Interactive servers can be connection oriented or connectionless. A connection-oriented service may be triggered simply by the presence of an incoming connection, without an explicit request from the client. Such a service is known as a *connection-triggered service* and the corresponding server as a *connection-triggered server*. For example, the well-known DAYTIME service is a connection-triggered service. A connection-oriented service may be based on serving a single request from a client. The server for such a service is known as a *one-shot, connection-oriented server*. TIME and ECHO are examples of such well-known services.

If a connection-oriented server has to respond to several requests from a client before moving to the next client, the iterative version of such a server will cause unnecessarily long delays at the client side. This would result in a long average waiting time for clients and, possibly, lost connection requests from clients if the queue associated with the server-side passive socket overflows. This necessitates the design of servers that can handle multiple clients simultaneously. Such servers are known as *connection-oriented, concurrent servers*. These servers use slave processes to handle multiple clients simultaneously. Several well-known services, including HTTP (WWW), FTP, SSH, and TELNET are offered through connection-oriented, concurrent servers.

This discussion leads us to the following types of servers:

1. Iterative connectionless
2. Iterative connection-oriented
    a. Connection-triggered
    b. Interaction based
3. Concurrent connectionless
4. Concurrent connection-oriented
    a. Master–slave-based
    b. Master–slave, multiservice

In case of the connection-oriented, concurrent, master–slave model, the main server process is known as the *master server process* and a process created to serve a client is known as the *slave process*. The master server accepts a client request, creates a new socket, forks the slave process, and the slave process services the client by communicating with it using the newly created socket. The slave process may also overwrite itself with the executable for the service using a call in the `exec()` family. The master–slave model may be scaled to handle multiple clients simultaneously.

A server may offer a service using both stream (`SOCK_STREAM`) and datagram (`SOCK_DGRAM`) styles of communication. Similarly, a server may offer multiple services. Such a server is known as a *multiservice server*. Lastly, a multiservice server may offer all of the services that it offers using the stream and datagram styles of communication.

W20.8 ALGORITHMS AND EXAMPLES FOR SOCKET-BASED CLIENT–SERVER SOFTWARE

We now discuss the algorithms, system call graphs, client–server interaction sequences, and example source code for the client–server models based on the types of servers discussed in the previous section. Note that the call graphs show the sequence of system calls for socket-based I/O in the client and server processes. Library and/or system calls for performing I/O with standard devices and for performing other ancillary operations, including conversion from host to network byte order and vice versa, ASCII to integer conversion, and translation of an IP address from the DDN to binary are not included in these call graphs. The interaction sequences show how system calls between the client and server processes rendezvous. The call graphs for the client and server processes are shown as solid lines, and interaction sequences are displayed as dotted lines between the two processes.

W20.8.1 Iterative Connectionless Client–Server Model

In this model, the client and server processes communicate using the connectionless style of communication based on the UDP protocol. The server process waits for a client request, forms a reply after receiving the request, sends the response to the client process, and goes back to wait for the next request from the same or another client. Figures W20.14 and W20.15 show the algorithms, system call graphs, and interaction sequences for this client–server model.

Server Process	**Client Process**
1. Create a socket for `SOCK_DGRAM` style of communication 2. Bind an address to the socket 3. Receive a request from a client using the `recvfrom()` system call 4. Prepare a response and send it to the client using the `sendto()` system call 5. Go to step 3	1. Create a socket for `SOCK_DGRAM` style of communication 2. Send a request to the server using the `sendto()` system call 3. Receive server's response using the `recvfrom()` system call and process it according to the protocol for the service 4. Close the socket 5. Exit

Figure W20.14 Algorithms for the client and server processes for the iterative, connectionless client–server model

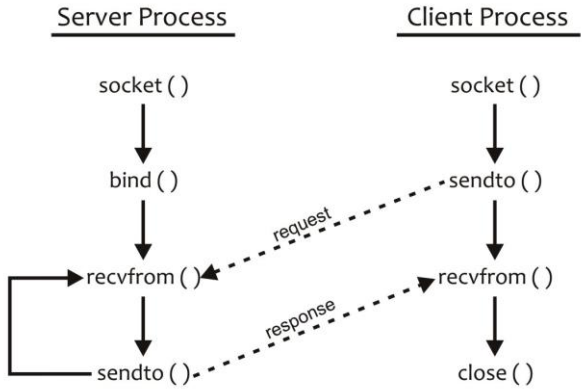


Figure W20.15 System call graphs and interaction sequences for the iterative, connectionless client–server model

We now discuss the example of an iterative connectionless client–server model to deliver the current time in human-readable form. In this example, we implement the well-known TIME service, except that it is offered at an arbitrary port and not at the well-known port 37. We discuss the source code for both the client and server processes for the application.

Before discussing the protocol for the TIME service and the relevant source code, we need to understand how UNIX and the Internet maintain time. UNIX maintains time in terms of the number of seconds since the *UNIX epoch*; that is, midnight, January 1, 1970. The Internet, on the other hand, maintains time in terms of the number of seconds since midnight, January 1, 1900. The number of seconds between these two baselines is 2,208,988,800 seconds. In other words, the UNIX epoch is 2,208,988,800 seconds away from the Internet baseline. Thus, if you use a function on a UNIX machine that returns time and you want to convert it to the Internet time, you need to add 2,208,988,800 to it. Conversely, if you receive time from the Internet and want to process it on a UNIX machine, you need to subtract 2,208,988,800 from it. Same scheme of things works for Linux too.

In the client–server model for the TIME service, the client process sends an arbitrary request to the server process. The server process, without even deciphering the client request, uses a function to get the current time with respect to the UNIX epoch, adds 2,208,988,800 to it, converts the resultant value (i.e., time in the Internet domain) to the network byte order, and sends it to the client process. The client

process receives it, converts it from the network byte order to the host byte order, subtracts 2,208,988,800 from it, uses a function to convert it into human-readable form and displays the time, closes its socket, and quits. In the following and subsequent code examples, we define the symbolic constant UNIXEPOCH as 2,208,988,800.

Here is the source code for both the client and server software, their compilation, and a sample run. Note that we use the `connect()` system call in the client process. However, because the socket descriptor specified in the call is a UDP socket, no network traffic is therefore generated and no three-way handshake takes place between the client- and server-side sockets. Nonetheless, the address of the remote socket is stored in the local socket's data structure. This allows us to use the `read()` and `write()` system calls instead of `recvfrom()` and `sendto()` calls.

```
$ more time_server_UDP.c
/*
Usage: Server-name Protocol Port
Here, Protocol is the transport level protocol
and Port is the procotol port number where the
service is to be offered.
*/
#include <time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define UNIXEPOCH 2208988800
#define QLEN    0
#define BUFF    256

int main(int argc, char *argv[])
{
    int s;

    /* Exit if program is not run with two command line arguments. */
    if (argc != 3) {
        printf("Usage: server protocol port\n");
        exit(1);
    }

    /* Create a TCP socket, bind name to it, and put it in passive mode */
    s = CreatePassiveSock(argv[1], argv[2], QLEN);

    /* Read client request and send current time to client */
    while (1) {
       /* UDP time service code */
       TimeServerUDP(s);
    }
}

int CreatePassiveSock(char *protocol, char *portstr, int qlen)
{
    /* Insert code for the function */
}

/* Provide TIME service to a client. 'sock' is the  */
/* server-side socket.  */
void TimeServerUDP(int sock)
{
    int n, saddrlen;
    char buf[BUFF];
    time_t current_time;
    struct sockaddr_in saddr;

    saddrlen = sizeof(saddr);
    /* Read client request and send current time to client */
    n = recvfrom(sock, buf, sizeof(buf), 0,
                        (struct scokaddr *) &saddr, &saddrlen);
    if (n == -1) {
       perror("recvfom failed");
       exit(1);
    }
```

```
    (void) time(&current_time);
    current_time = htonl((u_long) (current_time + UNIXEPOCH));
    (void) sendto(sock, (char *) &current_time, sizeof(current_time), 0,
                        (struct sockaddr *)&saddr, saddrlen);
}
$ more time_client_UDP.c
#include <time.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define UNIXEPOCH 2208988800
#define Message "Message to time sever\n"

int main(int argc, char *argv[])
{
    int csock; /* Descriptor for connected socket */

    if (argc != 4) {
        printf("Usage: %s hostname port protocol\n", argv[0]);
        exit(0);
    }

    csock = CreateConnectedSock(argv[1], argv[2], argv[3]);

    /* Perform client-side echo using csock, the connected socket */
    TimeClientUDP(csock);

    /* Deallocate socket and return */
    close(csock);
    return 0;
}

int CreateConnectedSock(char *ip, char *portstr, char *protocol)
{
    /* Insert code for the function */
}

/* Client side of the TIME service. 'sock' is the  */
/* socket "connected" to the server-side socket.   */
void TimeClientUDP(int sock)
{
    int n;
    time_t current_time;

    (void) write(sock, Message, strlen(Message));

    n = read(sock, (char *)&current_time, sizeof(current_time));
    if (n < 0) {
        perror("read failed");
        exit(1);
    }
    current_time = ntohl((u_long) current_time);
    current_time = current_time - UNIXEPOCH;
    printf("%s", ctime(&current_time));
}
$
```

Here are the compilation of the server and client software and a few sample runs. As expected, execution of the client process with the TCP protocol failed.

```
$ gcc time_server_UDP.c -w -o udptimes
$ gcc time_client_UDP.c -w -o udptimec
$ udptimes udp 6001 &
[1] 4618
$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
sarwar    4388  4387  0 20:20 pts/1    00:00:00 -bash
sarwar    4618  4388  0 20:49 pts/1    00:00:00 udptimes udp 6001
sarwar    4621  4388  0 20:49 pts/1    00:00:00 ps -f
$ udptimec 127.0.0.1 6001 udp
```

```
Sat Aug 25 20:49:22 2018
$ udptimec 127.0.0.1 6001 tcp
connect failed: Connection refused
$ kill -9 4618
$
```

Exercise W20.14

Repeat this session on your Linux system to verify that it works as expected.

Exercise W20.15

Run the client and server processes on different windows on the same machine and on different machines on the Internet.

W20.8.2 Iterative Connection-Triggered Client–Server Model

In this model, the client and server processes communicate using the connection-oriented style of communication based on the TCP protocol. The server process waits for the connection request from a client process. As soon as it receives a connection request, it forms a reply without receiving an explicit request from the client process, sends the response to the client process, and goes back to wait for a connection request from another client. Figures W20.16 and W20.17 show the algorithms, system call graphs, and interaction sequences for this client–server model.

Server Process	**Client Process**
1. Create a socket for `SOCK_STREAM` style of communication 2. Bind an address to the socket using the `bind()` system call 3. Put the socket in passive mode using the `listen()` system call 4. Receive a connection request from a client using the `accept()` system call and communicate with the client using the newly created active socket 5. Prepare a response and send it to the client using the `write()` system call and close the active socket 6. Go to step 4	1. Create a socket for `SOCK_STREAM` style of communication 2. Send a connection request to the server process using the `connect()` system call 3. Receive server's response using the `read()` system call and process it according to the protocol 4. Close the socket 5. Exit

Figure W20.16 Algorithms and system call sequences for the iterative, connection-triggered client–server model

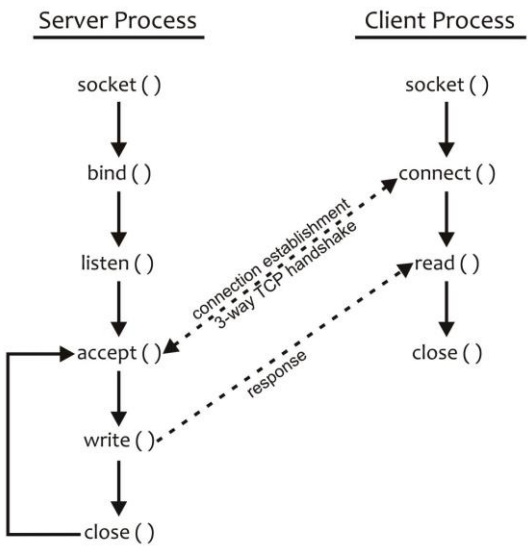


Figure W20.17 System call graphs and interaction sequences for the iterative, connection-triggered client–server model

We now discuss the example of an iterative connection-triggered client–server model to deliver the current time in human-readable form. In this example, we implement the well-known DAYTIME service, except that our service is offered at an arbitrary port and not at the well-known port 13. We discuss the source code for the client and server processes for the application.

The protocol for the client–server model for the TCP connection-triggered DAYTIME service is similar to the TCP TIME service, except for two differences. First, the client process does not send any message to the server process after successfully establishing a connection with the server process. Second, whereas in the case of the TIME service, the current clock-tick count is converted into a human-readable form of time on the client side, in the case of the DAYTIME service, the conversion is carried out on the server side. The general structure of the client and server software is similar to that of the TIME service, except, of course, for the functions to handle the current time on both sides.

Here is the source code for the client and server software:

```
$ cat daytime_server_TCP.c
#include <stdio.h>
#include <time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define BUFF  256
#define QLEN  5

int main(int argc, char *argv[])
{
    int psock, asock, caddrlen;
    struct sockaddr_in caddr;

    /* Exit if program is not run with two command line arguments. */
    if (argc != 3) {
        printf("Usage: server protocol port\n");
        exit(1);
    }

    /* Create a TCP socket, bind name to it, and put it in passive mode */
    psock = CreatePassiveSock(argv[1], argv[2], QLEN);

    caddrlen = sizeof(caddr);
    while (1) {
       /* Accept actual connection from the client */
       if ((asock = accept(psock, (struct sockaddr *)&caddr, &caddrlen)) == -1) {
           perror("accept failed");
           exit(1);
       }

       /* The Daytime service: Send current time as string to client */
       (void) TCPdaytime(asock);

       /* Deallocate the active socket and accept next connection */
       close(asock);
    } /* while */
}

int CreatePassiveSock(char *protocol, char *portstr, int qlen)
{
    /* Insert code for the function */
}

/* TCPdaytime(active socket descriptor) */
int TCPdaytime(int s)
{
    int       n;
```

```
    time_t    current_time;  /* Current time in ticks   */
    char      *str;          /* Pointer to time string  */
    char      *ctime();

    /* Get the current time in terms of clock ticks from   */
    /* UNIX Epoch, i.e., midnight January 1, 1970 and      */
    /* save it in current_time.                            */
    (void) time(&current_time);

    /* Convert current time into a humanly readable string */
    /* and return pointer to this string, saved in ptr     */
    str = ctime(&current_time);

    /* Send humanly readable time string to client process */
    if ((n = write(s, str, strlen(str))) == -1) {
       perror("write call failed");
       exit(1);
    }
    return 0;
}
$ cat daytime_client_TCP.c
#include <netdb.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define BUFF  256

int main(int argc, char *argv[])
{
    int csock; /* Descriptor for connected socket */

    if (argc != 4) {
       printf("Usage: %s hostname port protocol\n", argv[0]);
       exit(0);
    }

    csock = CreateConnectedSock(argv[1], argv[2], argv[3]);

    /* Perform client-side of the DAYTIME service using csock,   */
    /* the client socket connected the server-side active socket */
    DaytimeClientTCP(csock);

    /* Deallocate socket and return */
    close(csock);
    return 0;
}

int CreateConnectedSock(char *ip, char *portstr, char *protocol)
{
    /* Insert code for the function */
}

/* Client side of the DAYTIME service. 'sock' is the  */
/* client socket connected to serve's active socket.  */
void DaytimeClientTCP(int sock)
{
    int n;
    char buf[BUFF];

    /* Read time string from server and display on screen */
    while ((n = read(sock, buf, BUFF)) > 0)
       write (1, buf, n);
}
$
```

Here is the compilation and a sample run of the DAYTIME client–server software.

```
$ gcc daytime_server_TCP.c -w -o tcpdaytimes
$ gcc daytime_client_TCP.c -w -o tcpdaytimec
$ tcpdaytimes tcp 6001 &
[1] 5012
$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
sarwar    4388  4387  0 20:20 pts/1    00:00:00 -bash
sarwar    5012  4388  0 21:07 pts/1    00:00:00 tcpdaytimes tcp 6001
sarwar    5013  4388  0 21:07 pts/1    00:00:00 ps -f
$ tcpdaytimec 127.0.0.1 6001 tcp
Sat Aug 25 21:07:35 2018
$ kill -9 5012
$
```

Exercise W20.16

Repeat this session on your Linux system to verify that it works as expected.

Exercise W20.17

Run the client and server processes in different windows on the same machine and on different machines on the Internet.

We have tried to structure our software from the three applications that we have discussed so far. However, we still see *code clones* in the client and server software, particularly in the `CreateConnectedSock()` and `CreatePassiveSock()` functions. This means that we can further improve the design of our client–server software by using *refactoring* and making functions out of these clones. We can further improve our design by using additional abstractions. These functions can be archived in a library on top of the existing libraries for network programming and system calls. We leave this work as an exercise for the reader.

W20.8.3 Iterative One-Shot Connection-Oriented Client–Server Model

In this model, the client and server processes communicate using the connection-oriented style of communication based on the TCP protocol. The server process waits for the connection request from a client process, accepts a connection request from the client, forms a response after receiving an explicit request from the client process, sends the response to the client process, and goes back to wait for a connection request from another client. Figures W20.18 and W20.19 show the algorithms, system call graphs, and interaction sequences for this client–server model.

Server Process	**Client Process**
1. Create a socket for `SOCK_STREAM` style of communication 2. Bind an address to the socket using the `bind()` system call 3. Put the socket in passive mode using the `listen()` system call 4. Receive a connection request from a client using the `accept()` system call and communicate with the client using the newly created active socket 5. Receive an explicit request from a client process 6. Prepare a response and send it to the client using the `write()` system call and close the active socket 7. Go to step 4	1. Create a socket for `SOCK_STREAM` style of communication 2. Send a connection request to the server process using the `connect()` system call 3. Send a request to the server process using the `write()` system call 4. Receive server's response using the `read()` system call and process it according to the protocol 5. Close the socket 6. Exit

Figure W20.18 Algorithms and system call sequences for the iterative, one-shot, connection-oriented client–server model

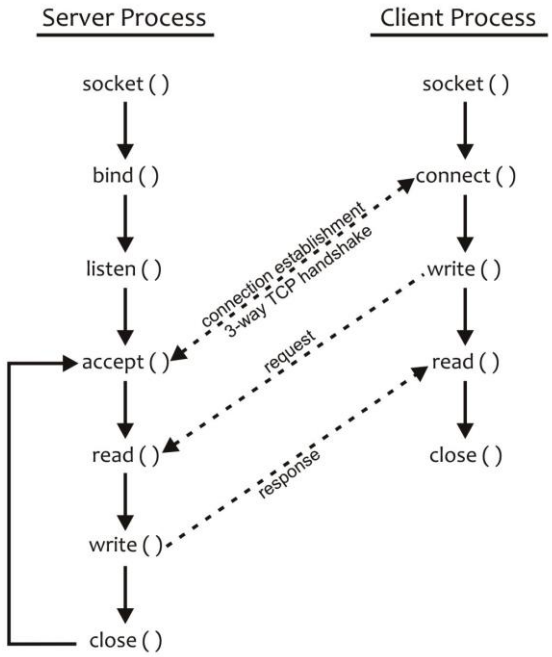


Figure W20.19 System call graphs and interaction sequences for the iterative, one-shot, connection-triggered client–server model

We now discuss the example of an iterative one-shot connection-oriented client–server model for the ECHO service. Our model implements the well-known ECHO service, except that it is offered at an arbitrary port and not at the well-known port 7. In this model for the ECHO service, the client process sends a connection request to the server process. The server process accepts the connection request, receives some text from the client process, sends back the same text to the client process, closes the active socket, and goes back to accept another client request. The client process receives the text from the server process, displays it on the screen, closes its socket, and quits.

Exercise W20.18

Write the code for the iterative one-shot connection-oriented client–server model for the ECHO service. Compile and run the implementation on your Linux system to verify that it works as expected. Show compiler commands for generating the executable codes for the client and server programs on your computer.

Exercise W20.19

Run the client and server processes in different windows on the same machine and on different machines on the Internet.

W20.8.4 Iterative Connection-Oriented Client–Server Model

In this model, the client and server processes communicate using the connection-oriented style of communication based on the TCP protocol. The server process waits for the connection request from a client process, accepts the connection request from a client process, and waits for a service request from the client. After receiving a request from the client, the server process forms a response, sends it to the client process, and waits for another request from the same client. The client process receives the response, processes it according to the underlying algorithm, and sends the next request to the server process. This interaction between the client and server processes continues until the client sends some sort of "quit" request to the server process. On receiving this request, the server and client processes disconnect gracefully by closing their respective sockets in an orderly manner. Figures W20.20 and W20.21 show the algorithms, system call graphs, and interactions sequences for this client–server model.

Server Process	Client Process
1. Create a socket for `SOCK_STREAM` style of communication 2. Bind an address to the socket using the `bind()` system call 3. Put the socket in passive mode using the `listen()` system call 4. Receive a connection request from a client using the `accept()` system call and communicate with the client using the newly created active socket 5. Receive an explicit request from the client process using the `read()` system call 6. If the received request is some kind of "quit," then go to step 9 7. Prepare a response and send it to the client process using the `write()` system call 8. Go to step 5 9. Close the active socket created by the `accept()` system call and go to step 4	1. Create a socket for `SOCK_STREAM` style of communication 2. Send a connection request to the server process using the `connect()` system call 3. Send an explicit request to the server process using the `write()` system call 4. Receive server's response using the `read()` system call and process it according to the service protocol 5. If the request sent was not some sort of "quit," then go to step 3 6. Close the socket 7. Exit

Figure W20.20 Algorithms and system call sequences for the iterative, connection-orientd client–server model

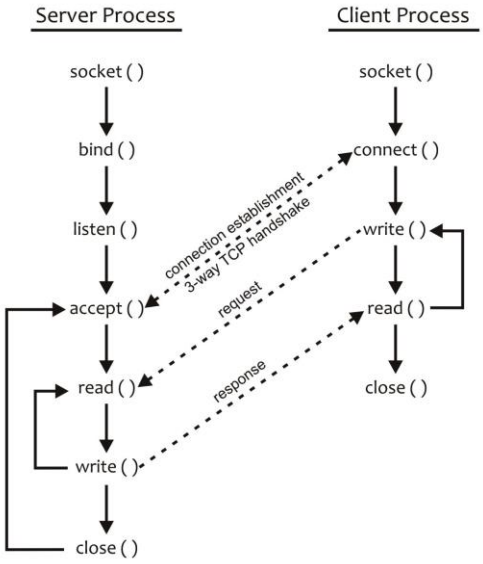


Figure W20.21 System call sequences and interaction sequences for the general iterative, connection-oriented client–server model

We now discuss an example of an iterative connection-oriented client–server model that deals with three requests from a client: "echo," "daytime," and "quit." A client process may repeat each of these requests multiple times. The client takes requests from the user, accepts a request only if it is one of these three, sends it to the server process, and waits for the server response. For any other request, the client process displays an error message on the screen and prompts the user for another request. If the user input is "echo," the model implements the well-known ECHO service. If the user input is "daytime," the model implements the DAYTIME service. If the user input is "quit," the client and server processes disconnect gracefully. The service is offered at an arbitrary port.

Exercise W20.20

Write the code for the iterative connection-oriented client–server model for the ECHO service.

Compile and run the implementation on your Linux system to verify that it works as expected.

Exercise W20.21

Run the client and server processes in different windows on the same machine and on different machines on the Internet.

20.8.5 Concurrent Connectionless Client–Server Model

In this model, the client and server processes communicate using the connectionless style of communication based on the UDP protocol. The master server process waits for a request from a client process, creates a slave process, hands over the client request and socket to the slave process, and goes back to receive another client request. The slave process forms a response according to the application protocol, sends it to the client process, and exits. Figures W20.22 and W20.23 show the algorithms, system call graphs, and interaction sequences for this client–server model. Note that the algorithm for the client process is the same as for an iterative connectionless server discussed in Section W20.7.1.

Server Process

Master Process

1. Create a socket for `SOCK_DGRAM` style of communication
2. Bind an address to the socket using the `bind()` system call
3. Receive a request from a client using the `recvfrom()` system call and create a slave process using the `fork()` system call to handle the client request
4. Go to step 3

Slave Process

1. Receive the client request as well as access to the socket
2. Prepare a response according to the application protocol and send it to the client process using the `sendto()` system call
3. Exit after serving a request

Figure W20.22 Algorithm for the concurrent, connectionless server process

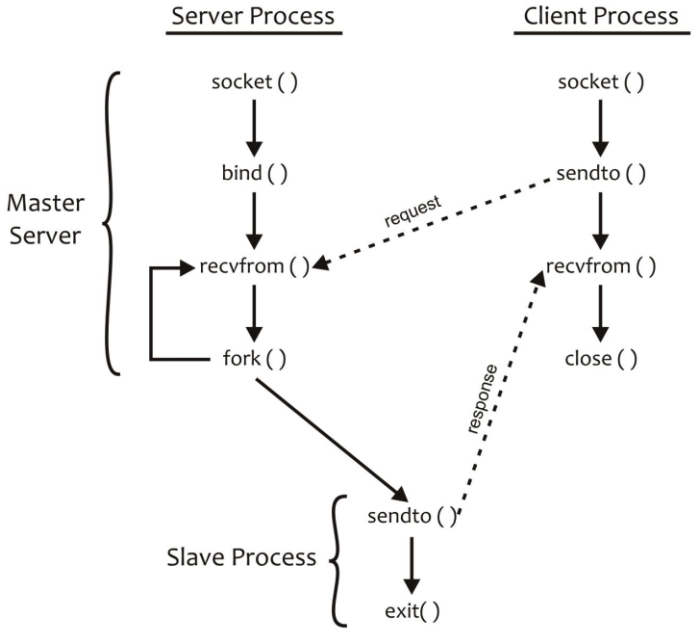


Figure W20.23 System call graphs and interaction sequences for the concurrent, connectionless client–server model

W20.8.6 Concurrent Connection-Oriented Client–Server Model

In this model, the client and server processes communicate using the connection-oriented style of communication based on the TCP protocol. The server process waits for the connection request from a client process, accepts the connection request, creates a slave process, hands over the connection and socket to the slave process, and goes back to accept another connection request. The slave process receives an explicit request form the connected client process, forms a response according to the application protocol, sends the response to the client process, and waits for the next request from the client process. This request–response session continues until the client sends a "quit" request of some sort. On receiving the "quit" request, the slave and client processes disconnect gracefully and close their sockets. Figures W20.24 and W20.25 show the algorithms, system call graphs, and interaction sequences for this client–server model. Note that the algorithm for the client process is the same as for an iterative connectionless server as discussed in Sections W20.6.12 and W20.8.1.

Server Process

Master Process

1. Create a socket for `SOCK_STREAM` style of communication
2. Bind an address to the socket using the `bind()` system call
3. Put the socket in passive mode using the `listen()` system call
4. Receive a connection request from a client using the `accept()` system call and create a slave process using the `fork()` system call to handle the request.
5. Go to step 4

Slave Process

1. Receive the socket that was created by the `accept()` system call and is connected to the client-side socket
2. Receive a request from the client using the `read()` system call
3. If the received request is some kind of "quit," then go to step 5
4. Prepare a response according to the application protocol, send it to the client using the `write()` system call, and go to step 2
5. Close the active socket created by the `accept()` system call, disconnect with the client process gracefully, and exit

Figure W20.24 Algorithm for the concurrent, connection-oriented server process

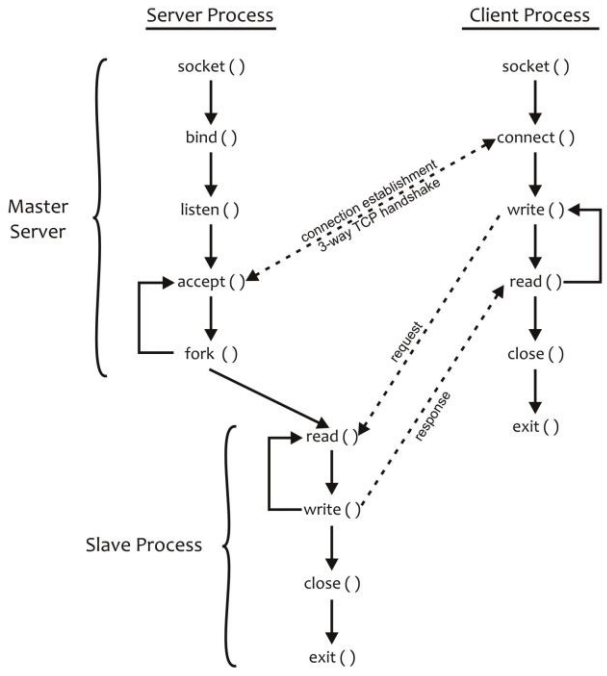


Figure W20.24 System call graphs and interaction sequences for the concurrent, connection-oriented client–server model using slave processes

Figure W20.26 shows the pictorial view of the concurrent connection-oriented client–server model with *k* clients being handled by *k* slave processes and the master process waiting to accept a connection request from another client process.

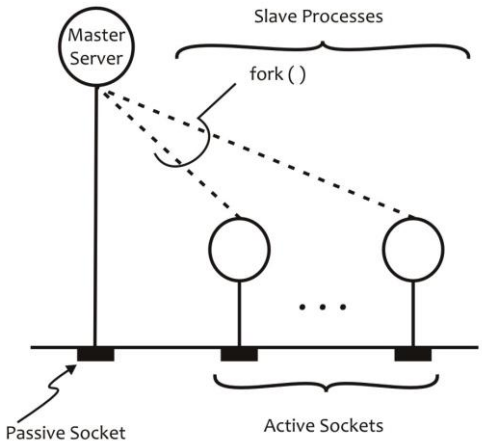


Figure W20.26 Connection-oriented, concurrent server

Here is the source code for the concurrent connection-oriented server software based on slave processes for the DAYTIME service. We have replaced the call to the `TCPdaytime()` function in the `while()` loop to a piece of code for creating a child (slave) process that closes the passive socket and calls the `TCPdaytime()` function to handle the client request. The parent process closes the active socket and goes back to accept the next connection from a client process.

```
% cat daytime_concurrent_server_TCP.c
#include <stdio.h>
#include <time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define BUFF  256
#define PORT  6001
#define QLEN  5

int main(int argc, char *argv[])
{
    int asock, psock, pid, caddrlen;
    struct sockaddr_in caddr;

    /* Exit if program is not run with two command line arguments. */
    if (argc != 3) {
        printf("Usage: server protocol port\n");
        exit(1);
    }

    /* Create a TCP socket, bind name to it, and put it in passive mode */
    psock = CreatePassiveSock(argv[1], argv[2], QLEN);

    while (1) {
       /* Accept actual connection from the client */
       if ((asock = accept(psock, (struct sockaddr *)&caddr, &caddrlen)) == -1) {
           perror("accept failed");
           exit(1);
       }
       /* Create a slave process and have it server the client   */
       /* The parent process closes the newly created active     */
```

```
        /* socket and goes back to accept the next client request */
        pid = fork();
        if (pid == -1) {
            perror("fork failed");
            exit(1);
        }
        if (pid == 0) { /* Child process */
            close(psock); /* Deallocate passive socket */
            /* The Daytime service: Send current time as string to */
            /* client using active socket                          */
            (void) TCPdaytime(asock);
            exit(0);
        }
        else /* Parent process: Deallocate the active socket       */
            close(asock);
    } /* while */
}

int CreatePassiveSock(char *protocol, char *portstr, int qlen)
{
    /* Insert code for the function */
}

int TCPdaytime(int s)
{
    /* Insert code for the function */
}
```

Here is the compilation of the source, the running of the resultant executable code, and its testing with the DAYTIME client developed in Section W20.8.2. The outputs of the various commands are self-explanatory. The session shows that you can use the loopback address with the client command instead of explicitly specifying the IP address of the host on which the client and server processes run.

```
% gcc daytime_concurrent_server_TCP.c -w -o tcpdaytimecons
% ./tcpdaytimecons tcp 6001 &
[1] 18738
% ./tcpdaytimec 202.147.169.196 6001 tcp
Sat Aug 15 22:56:46 2015
% ps
  PID TT  STAT    TIME COMMAND
 1291  1  Is+  0:00.73 -csh (csh)
16351  2  Ss   0:00.46 -csh (csh)
18738  2  S    0:00.00 ./tcpdaytimecons tcp 6001
18822  2  Z    0:00.00 <defunct>
18827  2  R+   0:00.01 ps
16380  3  Is+  0:00.16 -csh (csh)
% ./tcpdaytimec 127.0.0.1 6001 tcp
Sat Aug 15 22:56:55 2015
%
  PID TT  STAT    TIME COMMAND
 1291  1  Is+  0:00.73 -csh (csh)
16351  2  Ss   0:00.47 -csh (csh)
18738  2  S    0:00.00 ./tcpdaytimecons tcp 6001
18822  2  Z    0:00.00 <defunct>
18840  2  Z    0:00.00 <defunct>
18849  2  R+   0:00.01 ps
16380  3  Is+  0:00.16 -csh (csh)
% kill -9 18738 18822 18840
%
```

Exercise W20.22

Repeat this session on your Linux system to verify that it works as expected. Show the compiler commands to generate the executable codes for the client and server programs on a Solaris machine.

Exercise W20.23

Run the client and server processes in different windows on the same machine and on different machines on the Internet. Verify working of the concurrent server with multiple simultaneous clients.

Note that the outputs of the `ps` commands show that the child/slave processes become *zombies* when they terminate. This happens because its parent is not waiting when it terminates. It seems that an obvious solution for this problem is to use the `wait()` (or a variant of this call) before the `close(sock)` statement in the parent's code. However, this will make the parent process wait until the child (slave) process terminates, thereby making the model iterative. The real solution for the problem is to let the child become a zombie and then immediately remove it from the system. This can be achieved via the use of signal handling, as discussed in Section W20.5. Recall that when a child process terminates, the Linux kernel generates a `SIGCHLD` signal. You can insert a few lines of code in the server process to intercept this signal and run the `wait3()` system call to remove the relevant zombie from the system. We discuss this issue in detail in Section W21.12. For now, you can either remove zombies manually by using the `kill` command or use the code segments given in Section W21.12 to clean up a zombie process as it is created. For now, we use the `kill` command to remove zombie processes from the system.

The structure of the server process in this model may be extended by having each slave process overwrite itself with another executable file using a call from the `exec()` family. The executable file corresponds to the service to be provided for a given client request. A pictorial view of such a server process is shown in Figure W20.27.

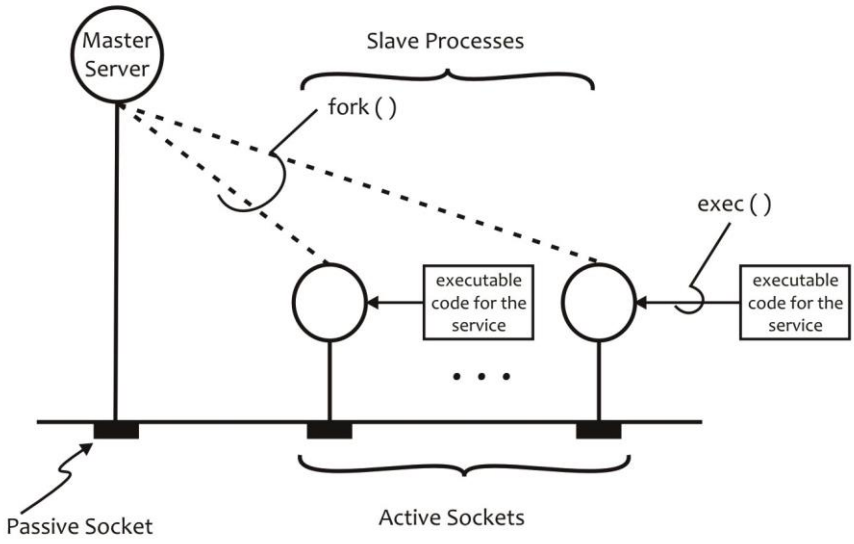


Figure W20.27 Concurrent connection-oriented server process using `fork()` and `exec()`

W20.9 SYNCHRONOUS VERSUS ASYNCHRONOUS I/O: THE `select()` SYSTEM CALL

Linux system calls such as `read()`, `write()`, `recvfrom()`, `sendto()`, and `accept()` block if the socket descriptors associated with them are not ready to perform input, output, or connection establishment [in the case of `accept()`]. As soon as the socket descriptors associated with any of these calls are ready for I/O or connection establishment, the calls unblock and perform their designated operation.

The I/O based on signals also works in a similar manner. As soon as a signal occurs, the associated signal-handling code executes. The I/O based on blocking calls and signals is known as *synchronous I/O*. Both work similar to interrupt handling in a computer system.

There are times when you need to perform nonblocking, asynchronous I/O and connection establishment. You may perform *asynchronous I/O* using the `select()` system call. Before we discuss

the algorithm for the connection-oriented concurrent server that works based on the principles of asynchronous I/O, we need to discuss the `select()` system call. It is a powerful system call that, depending on the value of one of its parameters, allows you to perform synchronous as well as asynchronous I/O.

The `select()` system call deals with descriptor sets. A *descriptor set* is a bit mask in which a bit represents a descriptor and its value indicates the state of the corresponding descriptor. The size of the bit mask is defined as `FD_SETSIZE`. This value is usually at least as large as the number of descriptors in the PPFDT on the system. On our Linux system, it is defined as unsigned 1,024. Descriptor numbers and bit numbers start with 0. Thus, bit *k* in a descriptor set represents descriptor number *k*, as shown in Figure W20.28.

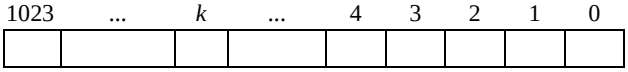


Figure W20.28 Pictorial view of a descriptor set.

The `select()` call takes three descriptor sets as arguments, one each for reading, writing, and exceptions. This call is normally used for socket I/O. Thus, the descriptors in the sets are usually socket descriptors. The call monitors the descriptors in these sets for input (reading), output (writing), and exceptions. If a bit in a descriptor set has a value of 0 (i.e., not set), it means that the corresponding descriptor is not ready for reading, writing, or exception, depending on which descriptor set in the `select()` call we are referring to. If a bit has a value of 1 (i.e., it is set), this means that the corresponding descriptor is ready for the purpose for which it is intended—input, output, or exception.

Here is a brief description of the `select()` system call.

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

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

Success: Number of ready descriptors out of those specified in the descriptor sets; 0 if time limit expires and no descriptor is ready

Failure: `-1` and kernel variable `errno` set to indicate the type of error; descriptor sets remain unmodified

Here, `nfds` is the number of file descriptors in the descriptor sets that need to be monitored, `readfds` is the read descriptor set, `writefds` is the set of descriptors on which the write operation is possible, `exceptfds` is the set of descriptors on which exceptions are possible, and `timeout` is the time for which the `select()` call waits for the selection process to complete. If you are not interested in one or more items in the descriptor set, you may specify them as null pointers. The `fd_set` type is defined as `unsigned long`. Thus, descriptor sets are stored as bit fields in arrays of unsigned long. As stated in Table W20.2, the following macros are used to manipulate descriptor sets: `FD_ZERO()`, `FD_SET()`, `FD_CLR()`, and `FD_ISSET()`. Table W20.21 gives the syntax and semantics of these macros. The behavior of these macros is undefined if `fd` is greater than `FD_SETSIZE` or less than 0.

Table W20.21 Macros for Manipulating Descriptor Sets

Macro	**Purpose**
`FD_ZERO(&fdset)`	Initializes the `fdset` descriptor set to the null set (i.e., all zeros)
`FD_SET(fd, &fdset)`	Includes the given descriptor in the set (i.e., sets the relevant numbered bit to 1)
`FD_CLR(fd, &fdset)`	Excludes the given descriptor from the set (i.e., sets the relevant numbered bit to 0)
`FD_ISSET(fd, &fdset)`	Tests if `fd` in `fdset` has a value of 0 or 1. Returns 0 if `fd` is not set in `fdset`, nonzero otherwise. It is usually used after the `select()` call has returned to see if the given descriptor is ready for I/O or exception

The amount of time that `select()` monitors the descriptor sets is dependent on the value of the `timeout` argument, as shown in Table W20.22.

Table W20.22 The Value of the `timeout` Argument and Its Effect

Value of `timeout`	**Effect**
Not a null pointer	The value in the variable of `struct timeval` type specifies the maximum time for the selection process to complete
A null pointer	The `select()` call blocks indefinitely until a descriptor in a descriptor set is ready for the activity it is designated for (input, output, or exception)
A pointer to a zero-valued `timeval` structure	The `select()` call continuously polls the descriptors in the descriptor sets to determine if any is ready

The `timeval` structure is defined as shown next. Thus, a variable of the `timeval` structure may be used to specify time with microsecond granularity.

```
struct timeval {
    time_t       tv_sec;   /* seconds      */
    suseconds_t  tv_usec;  /* microseconds */
};
```

The `select()` call may fail for several reasons. Table W20.23 lists some of the reasons why the `accept()` call may fail.

Table W20.23 Reasons for the `select()` Call to Fail

Reason for Failure	**Value of** `errno`
One of the descriptor sets contains an invalid descriptor	`EBADF`
One (or more) of the following arguments points to an illegal address: `readfds`, `writefds`, `exceptfds`, or `timeout`	`EFAULT`
A signal occurred before the time limit expired and before any of the descriptors became ready	`EINTR`
`timeout` is invalid (i.e., one of the fields of this argument is too big or negative)	`EINVAL`
`nfds` is invalid	`EINVAL`

The following program illustrates the semantics of the `select()` system call.

Figure W20.29 shows the algorithm for a single-process connection-oriented concurrent server using the `select()` system call. Note that this server serves one client request at a time and then moves to serve the request from the next client.

Server Process

1. Create a socket for `SOCK_STREAM` style of communication.
2. Bind an address to the socket using the `bind()` system call.
3. Put the socket in passive mode using the `listen()` system call.
4. Add the socket descriptor to the set of those sockets on which I/O or exception is possible—to keep the discussion simple, we will not deal with sockets meant to monitor exception conditions.
5. Use the `select()` system call to monitor and select those descriptors that are ready for I/O.
6. See which of the sockets are ready by checking the descriptors in the descriptor sets. If the passive socket is ready, use the `accept()` system call to accept the connection request from a client and add the newly created socket to the set of those on which I/O is possible.
7. If a socket other than the passive socket is ready, receive the next request from an already connected client using the `read()` system call, form a response, and send it to the client using the `write()` system call.
8. If the client request is to close a connection, close the connection gracefully, set the corresponding descriptor value to 0 in the original descriptor set on which I/O was possible.
9. Set the "readfds," "writefds," and "exceptfds" descriptor sets to the original set of descriptors on which I/O is possible (after accommodating any changes outlined in step 8)
10. Go to step 5.

Figure W20.29 Algorithm for the single-process connection-oriented, concurrent server process using the `select()` system call

We now discuss the single-process connectionless concurrent server for the TIME and ECHO services using the `select()` system call. The following session contains the code for the server, its compilation, execution, and testing using the TIME and ECHO client discussed in Section W20.8.2.

```
% more select_server.c
#include <time.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/select.h>

#define UNIXEPOCH 2208988800
#define TIME      "8001"
#define ECHO      "8002"
#define BUFF      256
#define QLEN      5

int main(int argc, char *argv[])
{
    int s1, s2, nfds;
    fd_set rfds;

    /* Create UDP sockets for the TIME and ECHO services at   */
    /* ports 8001 and 8002, respectively, bind names to them, */
    /* and return their descriptors in s1 and s2.             */
    s1 = CreatePassiveSock("udp", TIME, QLEN);
    s2 = CreatePassiveSock("udp", ECHO, QLEN);

    /* Set number of descriptors to be monitored by select   */
    nfds = s2+1;

    /* Set the read descriptor set to zero                   */
    FD_ZERO(&rfds);

    /* Set bits for TIME and ECHO in the read descriptor set */
    FD_SET(s1, &rfds);
    FD_SET(s2, &rfds);

    while (1) {
       if (select(nfds, &rfds, NULL, NULL, NULL) == -1) {
           perror("select failed");
           exit(1);
       }
       if (FD_ISSET(s1, &rfds)) {
           /* UDP time service code */
           TimeServerUDP(s1);
       }
       if (FD_ISSET(s2, &rfds)) {
           /* UDP time service code */
           EchoServerUDP(s2);
       }
       /* Reset descriptors */
       FD_SET(s1, &rfds);
       FD_SET(s2, &rfds);
    }
}

int CreatePassiveSock(char *protocol, char *portstr, int qlen)
{
    /* Insert code for the function */
}

void TimeServerUDP(int sock)
{
    /* Insert code for the function */
```

```
}

/* Provide echo service to a UDP client.                    */
void EchoServerUDP(int sock)
{
    int n, caddrlen;
    char buf[BUFF];
    struct sockaddr_in caddr;

    /* Communicate with client: read and write back         */
    memset(buf, 0, BUFF);
    n = recvfrom(sock, buf, sizeof(buf), 0,
                        (struct sockaddr *)&caddr, &caddrlen);
    if (n == -1) {
       perror("recvfom failed");
       exit(1);
    }
    (void) sendto(sock, buf, sizeof(buf), 0,
                        (struct sockaddr *)&caddr, caddrlen);
}
%
```

The program for the server process is fairly straightforward and similar to the code previously written in this chapter, except for the code related to the `select()` system call. We create two UDP sockets, one each for the two services, and bind addresses to them using the `CreatePassiveSock()` function. We then initialize the `nfds` variable to the number of bits to be monitored in the read descriptor set, `rfds`, initialize the read file descriptor set to 0, and the set bits in `rfds` corresponding to the two socket descriptors. This is followed by the `select()` system call with 0 (i.e., `NULL`) values for the write descriptor set, exception descriptor set, and `timeval` structure. The value of 0 for the `timeval` structure means that the `select()` call blocks indefinitely until a descriptor becomes ready. When a client request arrives at either or both sockets, then `select()` returns. Depending on the bit(s) in `rfds` that are set, the corresponding client is served. After serving a client request, we reset the bits in `rfds`. Note that we have written a new function, `EchoServerUDP()`, to serve UDP clients for the ECHO service.

Here is a sample run of our client–server model. We test the two services by using the UDP TIME client and the TCP ECHO client. The TCP ECHO client works for the UDP service also, because we run a connect on the UDP socket in our code. We use the `udpechoc` and `udptimec` clients developed in Section W20.6.12 and W20.8.1, respectively.

```
% gcc select_server.c -w -o selects
% ./selects &
[1] 67008
% ps
  PID TT  STAT    TIME COMMAND
59200  3  Is+  0:00.81 -csh (csh)
59284  4  Ss   0:00.97 -csh (csh)
67008  4  S    0:00.00 ./selects
67009  4  R+   0:00.00 ps
% ./udptimec 202.147.169.196 8001 udp
Wed Aug 19 00:11:09 2015
% ./udptimec 127.0.0.1 8001 udp
Wed Aug 19 00:11:13 2015
% ./udpechoc 202.147.169.196 8002 udp
Enter text for server : That's all folks!
Text from server : That's all folks!
% kill -9 67008
%
```

Exercise W20.24

Repeat this session on your Linux system to verify that it works as expected.

Exercise W20.25

At which port is each service offered?

Exercise W20.26

Run the client and server processes in different windows on the same machine and on different machines on the Internet. Verify working of the concurrent server with multiple simultaneous clients.

W20.10 THE LINUX SUPERSERVER: `inetd`

When you offer a network service on a system, the corresponding daemon runs on the system. This means that the main memory and several kernel data structures allocated to the daemon are used. If some services are sparingly used, you can offer them through a single server that monitors the client requests on the sockets associated with the services and runs the corresponding server daemons only on demand. This scheme makes efficient use of the main memory and kernel data structures. A server that offers multiple network services is called a *superserver*.

In Linux, `inetd` offers several basic Internet services and is known as the *Linux superserver*. It is *dynamically configurable*; that is, it can reconfigure itself while it runs. It executes as a single master process, creates sockets for each of the services that it is to offer according to the type of communication (`SOCK_STREAM`, `SOCK_DGRAM`, etc.) and relevant protocol to be used (TCP, UDP, etc.), binds a name to each socket, and monitors client requests on these sockets using the `select()` system call. When a client request arrives on a socket, the master server process forks a slave (child) process, hands over the request to it, and goes back to wait for new client requests. The slave process uses a call in the `exec()` family to overlay itself with the executable code for the corresponding service. The **/etc/inetd.conf** file specifies complete information about all the services that `inetd` offers, one line per service, as discussed in the next subsection. Figure W20.30 shows the basic setup for `inetd` as it starts running. Figure W20.31 shows the state of `inetd` while it handles one UDP client for the TELNET service, two requests from TCP clients for the ECHO service, and one TCP client each for the TIME, TELNET, and FINGER services.

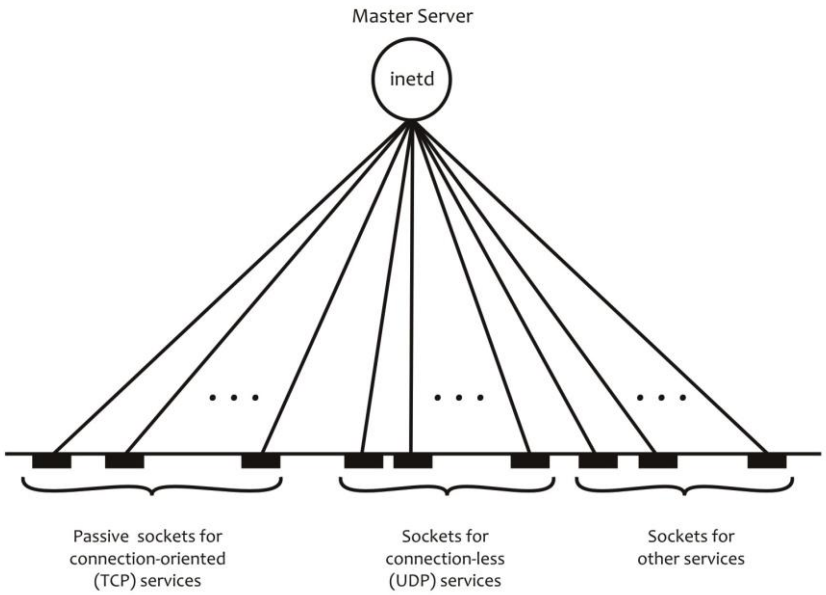


Figure W20.30 Basic setup of `inetd` as it starts running for offering TCP-based services

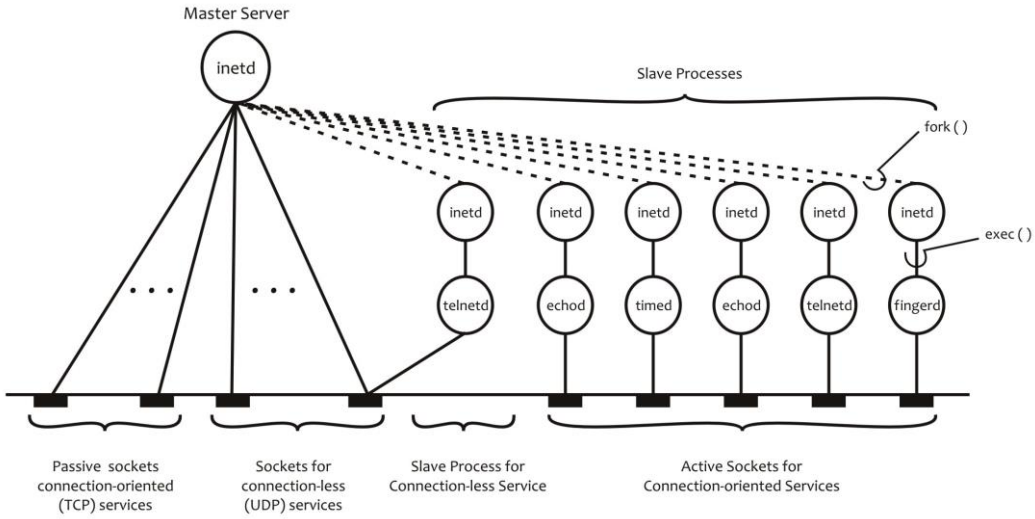


Figure W20.31 The setup of `inetd` as it serves two connection-oriented ECHO clients, one connectionless TELNET client, and one connection-oriented client each for TIME, TELNET, and FINGER

As an administrator (i.e., superuser), you should offer only lightly loaded services through this server. For example, you should offer SSH as a stand-alone service and the TELNET service through `inetd`, because most users today use the SSH service and TELNET is rarely used. When you offer a service through `inetd` and it becomes heavily loaded, then the performance of other services under `inetd` deteriorates. This is so because, with a higher number of slave processes serving the clients for a particular service, `inetd` has to multiplex a greater number of sockets associated with the slave processes. This means longer average waiting times for other services. Under these scenarios, you should remove such services from `inetd`.

We discuss briefly how `inetd` works, the kind of services it offers, and how you can change the set of services that it offers. On some Linux systems, the *tcpwrapper* package is used to enhance the security of `inetd`, and on other Linux systems this functionality is built into `inetd`.

W20.10.1 Managing `inetd` on Linux

We now briefly describe how you can manage inetd on your Linux system.

W20.10.1.1 Configuring `inetd` on Linux

The **/etc/inetd.conf** file contains the list of services offered by `inetd` on a Linux system. It is a text file that contains one seven-field line per service, having the following format:

```
service-name socket-type protocol wait/nowait user:group server-program arguments
```

Each of the services listed in **/etc/inetd.conf** must also be recorded in the **/etc/services** file. Table W20.24 contains the purpose of each field.

Table W20.24 Fields of a Line in **/etc/inetd.conf** and Their Purpose

Field	**Purpose**
Service-name	A port number in decimal or name corresponding to the service as listed in the **/etc/services** file
Socket-type	The type of socket that the service is offered for, usually, `stream` for TCP style communication or `dgram` for UDP style communication. It could also be `raw`, `rdm`, or `seqpacket`
Protocol	For TCP, this field is `tcp` or `tcp6` for IPv4 and IPv6, respectively. For UDP, this field is `udp` or `udp6` for IPv4 and IPv6, respectively. For protocols based on remote procedure call (RPC), this field may be `rpc/tcp` or `rpc/udp`

Wait/nowait	This field specifies if `inetd` should wait until the service program terminates (`wait`) or continue (`nowait`). For `stream`-type sockets, this field is normally `nowait`
User:group	The user name and optional group name that the respective service process runs as. Most of the services run under root ownership
Server program	Absolute pathname of the program that is executed for the offered service
Arguments	This field contains the name of the program and the arguments with which it runs

Here are the contents of the **/etc/inetd.conf** file on our Linux system.

```
$ cat /etc/inetd.conf
#
# Internet superserver configuration database
#
#
# Lines starting with "#:LABEL:" or "#<off>#" should not
# be changed unless you know what you are doing!
#
# If you want to disable an entry so it isn't touched during
# package updates just comment it out with a single '#' character.
#
# Packages should modify this file by using update-inetd(8)
#
# <service_name> <sock_type> <proto> <flags> <user> <server_path> <args>
#
#:INTERNAL: Internal services
discard           stream tcp    nowait root   internal
discard           dgram  udp    wait   root   internal
daytime           stream tcp    nowait root   internal
time        stream tcp    nowait root   internal

#:STANDARD: These are standard services.

#:BSD: Shell, login, exec and talk are BSD protocols.
talk        dgram  udp    wait   nobody.tty    /usr/sbin/in.talkd   in.talkd
ntalk             dgram  udp    wait   nobody.tty    /usr/sbin/in.ntalkd
    in.ntalkd

#:MAIL: Mail, news and uucp services.

#:INFO: Info services

#:BOOT: TFTP service is provided primarily for booting.  Most sites
#       run this only on machines acting as "boot servers."

#:RPC: RPC based services

#:HAM-RADIO: amateur-radio services

#:OTHER: Other services
$
```

Note that the service name in the first field is different from the program name given in the last field, which is the name of the program (daemon), along with command line arguments, that executes when `inetd` receives a client request for this service. Any user may display the contents of this file.

Some of the services offered by `inetd` are labeled as “internal.” The following session shows such services on our Linux system.

```
$ grep "internal"$ /etc/inetd.conf
discard           stream tcp    nowait root   internal
discard           dgram  udp    wait   root   internal
daytime           stream tcp    nowait root   internal
time        stream tcp    nowait root   internal
$
```

These services are quite trivial and are handled directly by `inetd` without running any external program (daemon). They are useful for testing purposes but are prone to "denial of service" attacks. Accordingly, you should disable them by commenting them out. You can list the services that are enabled by using the `grep -v "^#" /etc/inetd.conf` command.

W20.10.1.2 Locating `inetd` Services on Linux

The **/etc/services** file is the database that contains the name of a service, the port number for the service, and the transport-layer protocol it uses for communication (usually `tcp` or `udp`). Any service that `inetd` offers must have an entry in the **/etc/services** file. It has one line per service in the following format:

```
service-name port-number/protocol-name [aliases]
```

Here, `service-name` is the name of the service such as `smtp`, `port-number` is the IANA assigned port number, such as `25` for `smtp`, and `protocol-name` is the transport-layer protocol, such as `tcp`, `udp`, and so on. Here is an example line from the **/etc/services** file:

```
pop3        110/tcp    #Post Office Protocol - Version 3
```

This line shows that `pop3` is a TCP (i.e., connection-oriented) service, offered at port 110.

The following session shows that there are three entries for the SSH service in the **/etc/services** file on our Linux system. This means that the SSH service is offered on port 22 under the `tcp`, `udp`, and `sctp` protocols. Stream Control Transmission Protocol (`SCTP`) has been designed to transmit multiple streams of data between two connected sockets. It is also known as the "next-generation TCP" or "TCPng." The SSHELL service on port 614 is for secure socket layer shell (SSLshell).

```
$ grep ^ssh /etc/services
ssh             22/tcp                      # SSH Remote Login Protocol
ssh             22/udp
$
```

W20.10.1.3 Locating `inetd` Protocols on Linux

Another file that `inetd` reads when it runs is **/etc/services**. This file is the database of protocol names and contains the name of a protocol, the IANA assigned number for the protocol, and aliases for the protocol. Any protocol that `inetd` uses for a service must have an entry in the **/etc/protocols** file. It has one line per protocol in the following format:

```
protocol-name number [aliases]
```

Here, `protocol-name` is the name of the protocol such as `tcp`, `number` is the IANA assigned number for the protocol such as `6` for `tcp`, and `aliases` are alternative names for the protocols, such as `TCP` for `tcp`. Here is the line for the `tcp` protocol in the **/etc/protocols** file:

```
$ grep ^tcp /etc/protocols
tcp     6       TCP             # transmission control protocol
$
```

W20.10.1.4 Adding or Deleting `inetd` Services on Linux

As an administrator, you can delete a service that `inetd` offers by simply removing the line for the relevant service from the **/etc/inetd.conf** file. Similarly, you can add a new service in `inetd` by adding a line for the corresponding service in the **/etc/inetd.conf** file.

After making the requisite changes in the **/etc/inetd.conf** file, you can reconfigure `inetd` dynamically so that it offers the updated set of services without having to terminate and restart it. You can do so by sending `SIGHUP` (no hang-up signal) to `inetd` using the `kill` or `killall` command. Table W20.25 shows four ways of performing this task. Note that the `killall` command is used with a process name and the `kill` with a PID. Note the use of *grave accents* (`) for the substitution of the `cat /var/run/inetd.pid` command, with the resultant execution of the command.

Table W20.25 Ways of Restarting `inetd`

1	`killall -HUP inetd`
2	`killall -1 inetd`
3	`kill -HUP `cat /var/run/inetd.pid``

4	`kill -1 `cat /var/run/inetd.pid``

Exercise W20.27

Display the lines for all “internal” services listed in the **/etc/inet.conf** file on your system.

Exercise W20.28

Display the line in the **/etc/protocols** file for the UDP protocol. What numerical value is associated with the UDP protocol? How many protocols are defined in this file?

W20.11 CONCURRENT CLIENTS

It seems obvious to think of concurrent servers to serve multiple clients simultaneously in an efficient manner, thereby reducing the average response time experienced by a client. However, the need for concurrent clients is not so obvious. There are several real-life examples of concurrent clients that are necessitated due to the application protocol. Many of the well-known services require concurrent clients, including SSH, TELNET, FTP, and HTTP. A client for each of these services has to deal with two descriptors: a standard input descriptor (keyboard) to read user input and a socket descriptor used by the client to communicate with the server process. Either descriptor may become ready for I/O asynchronously; a user may type the next command for the client using the keyboard any time that he/she desires to do so and the server response for a previous client request may arrive at the client-side socket at any instant of time, depending on the load on the server process and network traffic.

We explain the need for a concurrent client by using the example of a connection-oriented (TCP) SSH client. A TCP client for SSH expects input from two descriptors, one for the keyboard and the other for the socket that is connected to the SSH server-side socket. The client needs to receive user input (i.e., the next shell command to be executed on the remote host) entered through the keyboard and the server’s response through the connected socket, as shown in Figure W20.32. Since the two descriptors become ready asynchronously, you may implement such concurrent clients by using the `select()` system call.

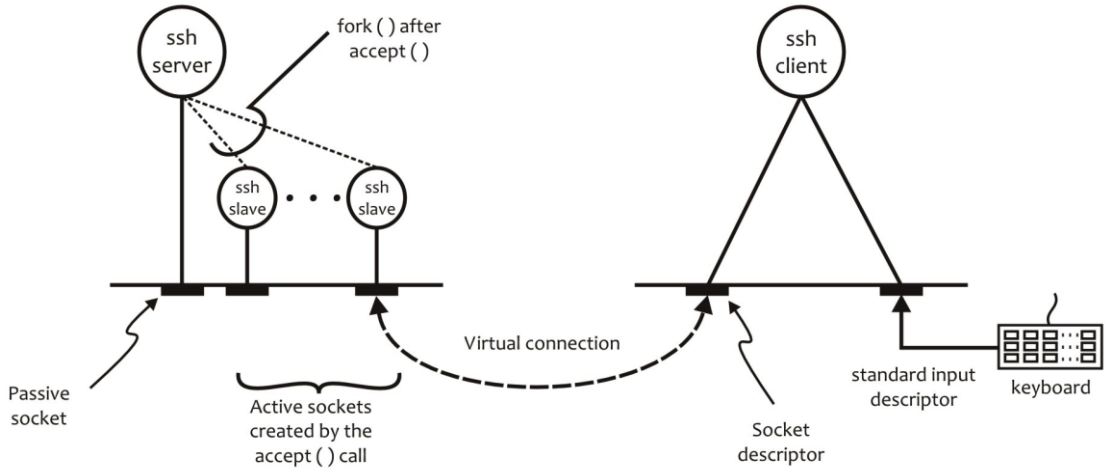


Figure W20.32 An SSH client deals with two descriptors

Another scenario under which a concurrent client becomes necessary arises because BSD UNIX does not allow independent processes to share main memory. The X Window System allows multiple clients to paint (i.e., redraw) a bitmapped display, so that the windows of the respective clients may be updated when required. Since the X display is memory mapped, the X server puts the information it receives from the various clients into one contiguous display buffer in the main memory. Depending on its location on the screen, each client window occupies a particular region of the buffer. The single-process server then uses the `select()` system call to handle asynchronous input from the client-side sockets and paint the screen accordingly.

You may also like to write the client for a particular service that simultaneously connects with multiple servers for a given service. Such concurrent clients are written using the master–slave model, where a

slave process is created to handle a particular server. The domain names and/or IP addresses of the servers are passed to the client as command line arguments. One reason for writing such concurrent clients is to measure the response time for different servers.

Exercise W20.29

Write the names of five well-known Internet services that use concurrent clients. Explain why these services use concurrent clients.

W20.12 WEB RESOURCES

Table 20.26 lists useful Web sites for Linux IPC and related topics.

Table W20.26 Web Resources for the Linux IPC and Related Topics

http://www.iana.org/	Webpage for IANA: responsible for the global coordination of the DNS Root, IP addressing, transport protocol port numbers, and other Internet protocol resources
http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml	Service Name and Transport Protocol Port Number Registry
http://tools.ietf.org/html/rfc6335	RFC6335: IANA Procedures for the Management of the Service Name and Transport Protocol Port Number Registry; forums, user groups, etc.
http://www.iana.org/assignments/protocol-numbers	Official names and numbers of the protocols
https://www.freebsd.org/	Home page for FreeBSD. Contains a lot of useful material, including FreeBSD source, manual pages, support, SVN repository, forums, user groups, etc.
http://www.t1shopper.com/tools/port-number/N/ http://www.adminsub.net/tcp-udp-port-finder/N	Reports IANA-assigned service on port "N" (e.g., for N = 22 the page reports the service that is assigned port 22)
http://www.tutorialspoint.com/	Excellent site for tutorials on all kinds of programming languages, including UNIX/Linux system programming in C
http://stackoverflow.com/	Free question and answer site for rookie and professional programmers

SUMMARY

We discussed a number of important topics, primarily related to Linux IPC, between related processes on the same machine, using related or unrelated processes on the same machine and related or unrelated processes on the same or different machines on a network. In doing so, we covered the use of pipes, named pipes (FIFOs), and sockets as communication channels and the related data structures, system calls, macros, and library functions. These calls and data structures deal with the creation of a communication channel, preparing it for communication where required, and using it for reading and writing messages between processes.

We discussed the issue of client–server design in detail, both for connectionless and connection-

oriented communication between client and server processes. Our focus in this regard was the transport-level protocols, UDP and TCP. We described the need for different types of servers, including iterative and concurrent servers.

We explained the working of the various types of client–server models by designing and implementing application-level services similar to the well-known Internet services ECHO, TIME, and DAYTIME. We also discussed the design and implementation of a concurrent server using slave processes. Throughout the chapter, our discussions included algorithms for the client and server processes, call graphs, and interaction sequences. The call graphs describe the sequence of system calls for implementing client and server software individually, and interaction sequences describe the interaction between the client and server processes to implement the application protocol at hand.

We then covered the design of servers that offers multiple services using the `select()` system call. We built a simple concurrent server using the `select()` system call to implement the TIME and ECHO services for the UDP protocol. We also described how the Linux superserver, `inetd`, works. Finally, we discussed the need for concurrent clients and how they work.

QUESTIONS AND PROBLEMS

1. What are little endian and big endian byte orders? Are these for data storage, data transmission, or both? What is network byte order? Give one example each of the processors that use these byte orders.
2. Show the contents of memory locations for the 32-bit hex number F9327CA5 using the little endian and big endian byte orders. Assume that the main memory is byte oriented.
3. What byte orders are used by the following CPUs: AMD64, Intel i7, Sun SPARC, Motorola 68000, IBM, PDP, and VAX? How did you obtain your answers?
4. What shell command can you use to display the name of the CPU used by your computer system? What byte order does this CPU use? How did you find out?
5. Write a C program that displays the size of the PPFDT and the largest descriptor value on your Linux system. Show your program and a sample run of the program.
6. What happens to a reader process when it reads from a pipe that has no data in it? Explain the reason for your answer.
7. What happens to a writer process when it writes to a full pipe? Explain the reason for your answer.
8. What is the effect of using the following flags in the `pipe2()` system call?
   a. `O_CLOEXEC`
   b. `O_NONBLOCK`
9. How should the `pipe2()` system call be used so that it behaves like the `pipe()` system call?
10. What is the synchronization issue in the bounded-buffer reader–writer problem? Who handles synchronization when two Linux processes communicate with each other using pipes: the reader process, the writer process, or someone else (specify)?
11. Write a program that demonstrates how the reader and writer processes behave while communicating through a widowed pipe.
12. Write a C program and save it in **widowed.c** that creates a widowed pipe and demonstrates that when a process reads from a pipe that cannot be written to, it results in the reader process receiving the eof message.
13. When you compile and run the program in the **broken_pipe.c** file in Section W20.4.2.4, it terminates without generating the "`Broken pipe`" error message. How can you verify that the pipe was actually widowed and the program terminated when the `write()` system call was executed?
14. Use signal handling to generate the "`Broken pipe`" error message by the program in **broken_pipe.c** when you compile and run it on your Linux machine. Write two versions of the program. The first version should terminate the program after displaying the "`Broken pipe`" message, and the second version should continue after displaying the message and execute the statement after the `write()` system call that causes `SIGPIPE`. Show the modified programs, their compilation, and sample runs.

15. Write a C program to demonstrate that a widowed pipe with its write end closed sends the eof message to the reader process. Show compilation and execution of the program.
16. Write a program that takes two command line arguments: the name of a text file that contains single-digit integer data and an integer to be searched from the sorted version of the data in the file. The program creates two children processes. The first child process reads the text file passed as the first argument, sorts the numbers (any sorting algorithm is allowed), and communicates the sorted numbers to the second child process via a pipe and exits. The second child process searches for the "majority number" in the sorted list of numbers, displays the majority number and sends the sorted numbers to the parent process via another pipe, and exits. The parent process searches the sorted data that it receives from the child process and displays the number to be searched for, or a message in case the number is not found. All printing, input, reading (from files or the console) should be done by system calls. You cannot use any library functions.
17. Are FIFOs process persistent or file system persistent? Explain your answer.
18. When a pipe is created, two file descriptors are used in the PPFDT. How many descriptors are used when a FIFO is created? How many are used when a FIFO is opened?
19. What is the amount (size) of data that can be written into a FIFO atomically on BSD and Solaris? Where did you find answer to the question?
20. Compile and run the client–server model shown next. Make sure to run the server process first because it creates the FIFO used for communication between the client and server processes. What does the model do? Does the model work as expected? If not, what is wrong with it? Clearly identify the issues and their remedy.

```
% more fifo.h
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>

extern int      errno;

#define FIFO   "/tmp/fifo"
#define PERMS   0666
#define SIZE    512

static char* message1 = "Hello, world!\n";
static char* message2 = "Hello, class!\n";
% more client.c
#include "fifo.h"

int main(void)
{
    char buff[SIZE];
    int fd;
    int n, size;

    /* Open FIFO. Assume that the server
       has already created them. */
    if ((fd = open(FIFO, 2)) == -1) {
        perror ("client open FIFO");
        exit (1);
    }

    /* client (fd); */
    size = strlen(message1);
    if (write(fd, message1, size) != size) {
        perror ("client write fd");
        exit (1);
    }
    if ((n = read(fd, buff, size)) == -1) {
        perror ("client read");
        exit (1);
    }
    else
        if (write(1, buff, n) != n) {
            perror ("client write stdout");
            exit (1);
        }
```

```
    close(fd);

    /* Remove FIFO now that we are done using it */
    if (unlink (FIFO) == -1) {
        perror("client unlink FIFO");
        exit (1);
    }
    exit (0);
}
% more server.c
#include "fifo.h"

int main(void)
{
    char buff[SIZE];
    int fd;
    int n, size;

    /* Create two FIFOs and open them for
       reading and writing */
    if ((mknod (FIFO, S_IFIFO | PERMS, 0) == -1)
                        && (errno != EEXIST)) {
        perror ("mknod FIFO");
        exit (1);
    }

    if ((fd = open(FIFO, 2)) == -1) {
        perror ("open FIFO");
        exit (1);
    }

    /* server (fd); */
    size = strlen(message1);
    if ((n = read(fd, buff, size)) == -1) {
        perror ("server read");
        exit (1);
    }
    if (write (1, buff, n) < n) {
        perror("server write stdout");
        exit (1);
    }
    size = strlen(message2);
    if (write (fd, message2, size) != size) {
        perror ("server write fd");
        exit (1);
    }
    close (fd);
}
%
```

21. Implement the client–server model given in Figure W20.9. The server offers the ECHO service. Test the model by running three concurrent clients through three terminal windows on your systems.
22. What system call would you use to create an IPv4 socket for stream-oriented communication that closes when the process executes a system call (or library call) in the `exec()` family?
23. What system call would you use to create an IPv6 socket for nonblocking stream-oriented communication that closes when the process executes a system call (or library call) in the `exec()` family?
24. What are the values of the following symbolic constants:
    a. `PF_INET`
    b. `PF_INET6`
    c. `SOCK_STREAM`
    d. `SOCK_NONBLOCK`
    e. `AF_UNIX`
    f. `PF_UNIX`
    g. `AF_INET`

25. How many domains of communication are supported by your system? How did you obtain your answer? If a symbolic constant is defined for this purpose, show this definition.
26. Write the code snippet to create a socket for the stream-oriented style of communication using the TCP protocol under IPv4. Mark the socket "close-on-exec" and "nonblocking."
27. When is a socket half associated? When is it fully associated?
28. What feature of the TCP/IP protocol allows multiple distinct servers to run simultaneously on a host on the Internet and establish multiple concurrent communication sessions between the client and server programs running on the hosts on the Internet?
29. What type of socket-based communication requires that the sockets of the two communicating processes are fully associated?
30. What are the new versions of the `inet_aton()` and `inet_ntoa()` calls? Why would you prefer to use the new calls as opposed to the old calls?
31. Design and implement a client–server model for the ECHO service using the UNIX domain sockets.
32. What is the maximum length of the queue associated with the `listen()` system call on your system? What flavor of Linux are you using? How did you obtain your answer?
33. Log on to a machine that runs at least one Internet server, such as SSH. Use the `netstat` command with appropriate options to determine the size of the queue associated with the passive socket of the server. Show the output of the command that you used for this purpose.
34. The `shutdown()` system call is used to close a socket. Why is it needed when you can use the `close()` system call to close a descriptor?
35. What type of network traffic is generated by the `connect()` system call for `SOCK_STREAM` and `SOCK_DGRAM` types of sockets under the `PF_INET` and `PF_LOCAL` domains?
36. What are the differences between the `accept()` and `accept4()` system calls?
37. Why must a process read data from a `SOCK_STREAM` socket in a loop? Why is this not required in the case of a `SOCK_DGRAM` socket?
38. What is the difference between active and passive sockets? How are they created? Why is the socket on which a server process waits for connection requests known as a passive socket?
39. Why are active sockets also known as ephemeral sockets?
40. Give two reasons each for the following system calls to fail:
    a. `socket()`
    b. `mkfifo()`
    c. `mkfifoat()`
    d. `bind()`
    e. `listen()`
    f. `connect()`
    g. `accept()`
    h. `select()`
    i. `shutdown()`
41. Suppose *S* is a connection-oriented concurrent server. If it is currently serving *K* clients, how many sockets and slave processes is the server using? What would these numbers be if *S* were a connectionless concurrent server? Explain your answers.
42. Write the code for the concurrent connectionless server for the ECHO service and test it with multiple clients accessing it simultaneously.
43. Write the code for the concurrent connection-oriented server for the ECHO service and test it with multiple clients accessing it simultaneously.
44. Design, code, and test the connection-oriented iterative client–server model discussed in Section W20.8.4.
45. What are the queue lengths associated with the passive sockets for all the servers running on your Linux system? How did you obtain your answer? Show your work.
46. Some of the library functions for network programming, such as `inet_ntop()`, have a size (or length) argument to specify the length in bytes of the destination address variable. Why is it needed?
47. What happens to the descriptor sets when the `select()` system call fails and returns `-1`?
48. What is returned by the `select()` system call when it returns due to the expiration of the timer?
49. When is it necessary to have concurrent clients? Give a few examples and give the structure of the code for an `HTTP` client.

50. Write the code for the single-process TCP ECHO server using the `select()` system call and test it using the ECHO client discussed in this chapter.
51. Write a thread-safe implementation of the `sleep()` system call that uses the `select()` system call. Explain why your implementation is thread safe.
52. Add the TCP DAYTIME service to `select_server.c` in Section W20.9.
53. Modify `select_server.c` so that it returns after 10.5 s instead of blocking until a descriptor becomes ready.
54. Why is there a need for concurrent clients? Write the concurrent client for the ECHO service that connects with multiple ECHO servers and displays the response time for each server. Show the source code for the client, its compilation, and a few sample runs.
55. What is the format of a line in the configuration file for `inetd`, the Linux superserver? What is the meaning of each field?
56. What types of services are offered through the Linux `inetd`? Explain your answer.
57. What would happen if a heavily loaded service was offered through `inetd`? Explain your answer. What should the system administrator do under such circumstances?
58. Write down the code snippet that makes `inetd` dynamically reconfigurable by using the `SIGHUP` signal. Show only signal-handling code and the structure of the code that actually reconfigures.
59. How many server and client processes run when `inetd` is serving two TIME clients and one client each for the ECHO, TELNET, and FTP services? How many sockets are used on the server side under this setting? Explain your answer.
60. What is the PID of `inetd` on your system? Write down the command that you used for this purpose.
61. What is a concurrent client? Why are concurrent clients needed? List three reasons. Also, list three well-known Internet services that require the use of concurrent clients.

ADVANCED QUESTIONS AND PROBLEMS

62. Locate the header files that contain the following data structures and symbolic constants. What command(s) did you use for this purpose. Show all your work.
    a. `struct sockaddr`
    b. `struct sockaddr_un`
    c. `struct sockaddr_in`
    d. `struct hostent`
    e. `struct servent`
    f. `struct timeval`
    g. `in_addr_t`
    h. `INADDR_NONE`
    i. `INADDR_ANY`
    j. `INET_ADDRSTRLEN`
    k. `INET6_ADDRSTRLEN`
    l. `SOMAXCONN`
    m. `FD_SETSIZE`
    n. `fd_set`
63. What command would you use to reinitialize and restart `inetd` on your Linux system?