Contents

6

Berkeley Sockets

6.1 Introduction

This chapter describes the first of several *application program interfaces (APIs)* to the communication protocols. The API is the interface available to a programmer. The availability of an API depends on both the operating system being used, and the programming language. The two most prevalent communication APIs for Unix systems are Berkeley sockets and the System V Transport Layer Interface (TLI). Both of these interfaces were developed for the C language. This chapter describes the Berkeley socket interface, and the next chapter describes TLI.

First, let's compare network I/O to file I/O. For a Unix file, there are six system calls for input and output: `open`, `creat`, `close`, `read`, `write`, and `lseek`. All these system calls work with a file descriptor, as described in Section 2.3. It would be nice if the interface to the network facilities maintained the file descriptor semantics of the Unix filesystem, but network I/O involves more details and options than file I/O.† For example, the following points must be considered:

† The original TCP/IP implementation for the BSD version of Unix was developed by Bolt, Beranek, and Newman (BBN) under a DARPA contract in 1981 [Walsh and Gurwitz 1984]. It is interesting to note that this version used these six system calls for both file I/O and network I/O. The socket interface and the interprocess communication facilities, which we describe in this chapter, were then added by Berkeley with the 4.1cBSD release.

- The typical client–server relationship is not symmetrical. To initiate a network connection requires that the program know which role (client or server) it is to play.
- A network connection can be connection-oriented or connectionless. The former case is more like file I/O than the latter, since once we open a connection with another process, the network I/O on that connection is always with the same peer process. With a connectionless protocol, there is nothing like an "open" since every network I/O operation could be with a different process on a different host.
- Names are more important in networking than for file operations. For example, a program that is passed a file descriptor by a parent process can do file I/O on it without ever needing to know the original name that the file was opened under. The file descriptor is all the process needs. A networking application, however, might need to know the name of its peer process to verify that the process has authority to request the services.
- Recall our definition of an association in Chapter 4

  {protocol, local-addr, local-process, foreign-addr, foreign-process}

  There are more parameters that must be specified for a network connection, than for file I/O. Also, as we saw in Chapter 5, each of the parameters can differ from one protocol to the next. Compare, for example, the format of either the *local-addr* or the *foreign-addr* between the Internet and XNS: the Internet address is 4 bytes while the XNS address is 10 bytes.
- For some communication protocols, record boundaries have significance. The Unix I/O system is stream oriented, not message oriented, as discussed in Section 3.6.
- The network interface should support multiple communication protocols. For example, the network functions can't use a 32-bit integer to hold network addresses, since even though this is adequate for Internet addresses, it is inadequate for XNS addresses. Generic techniques must be used to handle features that can change from one protocol to another, such as addresses. Supporting multiple communication protocols is almost akin to having Unix support multiple file access techniques. Imagine the Unix I/O system if the kernel had to support Unix I/O along with three or four other techniques (DEC's RMS, IBM's VSAM, etc.).

As another comparison between network I/O and file I/O, Figure 6.1 shows some of the steps required to use sockets, TLI, System V message queues, and FIFOs, for a connection-oriented transfer. We'll cover the details in Figure 6.1 as we proceed through this chapter and the next. Our purpose now is to show the added complexity imposed by the networking routines, compared to message queues and FIFOs.

		Sockets	TLI	Messages	FIFOs
			t_alloc()		
Server:	allocate space		t_open()	msgget()	mknod()
	create endpoint	socket()			open()
	bind address	bind()	t_bind()		
	specify queue	listen()			
	wait for connection	accept()	t_listen()		
	get new fd		t_open() t_bind() t_accept()		
			t_alloc()		
Client:	allocate space		t_open()	msgget()	open()
	create endpoint	socket()	t_bind()		
	bind address	bind()	t_connect()		
	connect to server	connect()		msgrcv() msgsnd()	read() write()
	transfer data	read() write() recv() send()	read() write() t_rcv() t_snd()		
	datagrams	recvfrom() sendto()	t_rcvudata() t_sndudata()		
	terminate	close() shutdown()	t_close() t_sndrel() t_snddis()	msgctl()	close() unlink()

Figure 6.1 Comparison of sockets, TLI, message queues, and FIFOs.

In the examples that we show in this chapter we must specify the type of process (client or server) and the type of protocol (connection-oriented or connectionless). Furthermore, for the server examples we have to specify if the server is a concurrent server or an iterative server. (Usually it doesn't matter to the client whether it is communicating with a concurrent server or an iterative server.) This gives four potential combinations.

	iterative server	concurrent server
connection-oriented protocol	infrequent (Daytime)	typical
connectionless protocol	typical	infrequent (TFTP)

The Internet Daytime protocol, which we describe in Section 10.2, is an example of a connection-oriented protocol whose server is usually implemented using an iterative server. Indeed, the 4.3BSD Internet superserver which we describe in Section 6.16 provides an iterative server for this protocol. We provide an example of a concurrent server using a connectionless protocol in Chapter 12—the TFTP server.

6.2 Overview

The socket interface was first provided with the 4.1cBSD system for the VAX, circa 1982. The interface that we describe here corresponds to the original 4.3BSD VAX release from 1986. This release supported the following communication protocols:

- Unix domain (described in Section 6.3)
- Internet domain (TCP/IP)
- Xerox NS domain

Be aware that some vendors who provide 4.3BSD-based systems do not provide the XNS support that was provided by Berkeley. Nevertheless, we show examples using XNS, since it provides a good working example, in addition to TCP/IP, for some of the portability problems in network programming.

Figure 6.2 shows a time line of the typical scenario that takes place for a connection-oriented transfer—first the server is started, then sometime later a client is started that connects to the server.

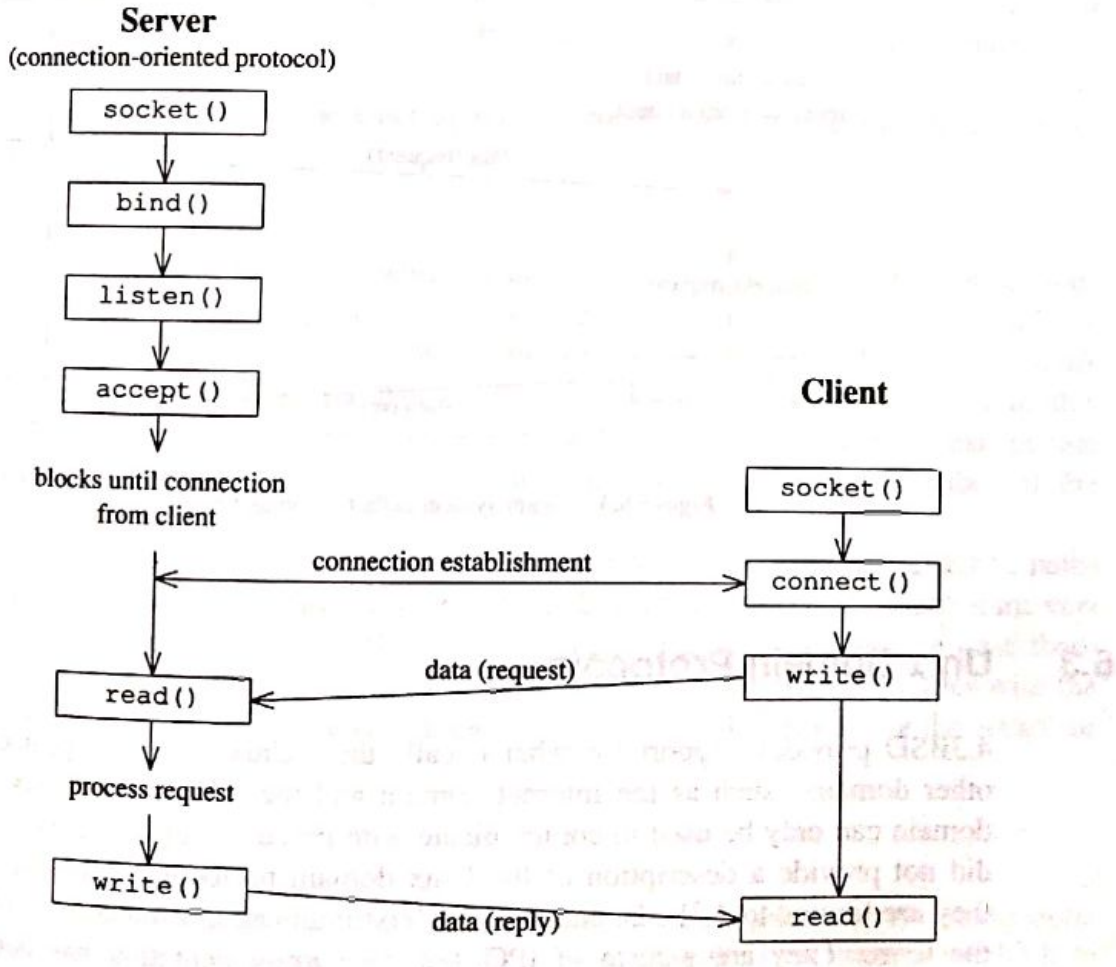

Figure 6.2 Socket system calls for connection-oriented protocol.

For a client–server using a connectionless protocol, the system calls are different. Figure 6.3 shows these system calls. The client does not establish a connection with the server. Instead, the client just sends a datagram to the server using the `sendto` system call, which requires the address of the destination (the server) as a parameter. Similarly, the server does not have to accept a connection from a client. Instead, the server just issues a `recvfrom` system call that waits until data arrives from some client. The `recvfrom` returns the network address of the client process, along with the datagram, so the server can send its response to the correct process.

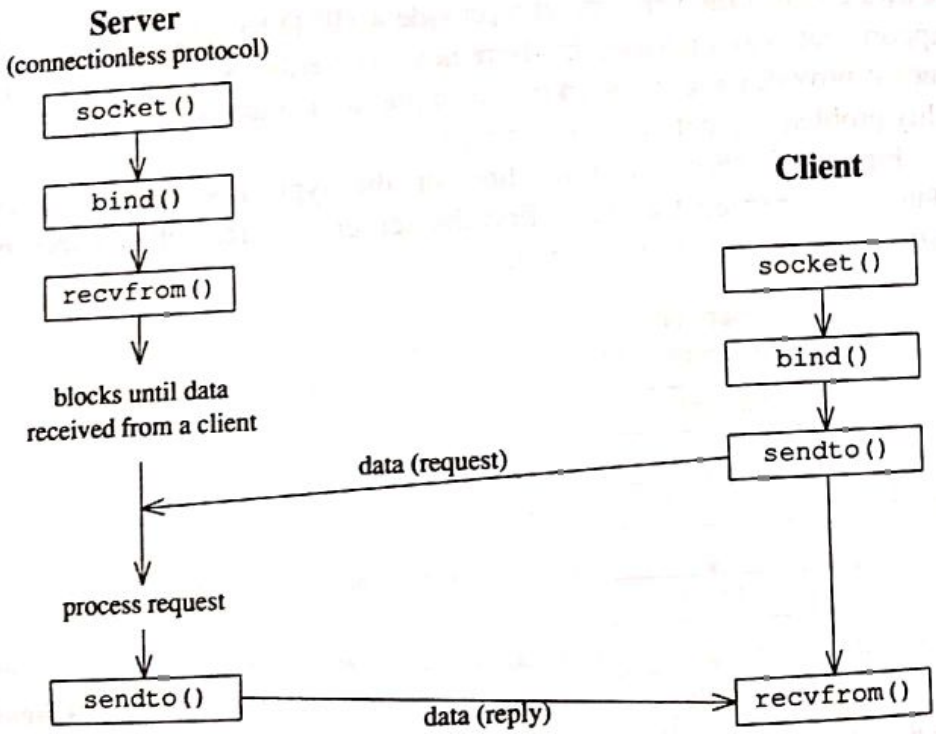

Figure 6.3 Socket system calls for connectionless protocol.

6.3 Unix Domain Protocols

4.3BSD provides support for what it calls the "Unix domain" protocols. But unlike other domains, such as the Internet domain and the XNS domain, sockets in the Unix domain can only be used to communicate with processes on the same Unix system. We did not provide a description of the Unix domain protocols in the previous chapter, as they are limited to 4.3BSD, and are not "communication protocols" in the true sense of the term. They are a form of IPC, but their implementation has been built into the 4.3BSD networking system in the same fashion as other true communication protocols.

4.3BSD provides both a connection-oriented interface and a connectionless interface to the Unix domain protocols. Both can be considered reliable, since they exist only within the kernel and are not transmitted across external facilities such as a communication line between systems. Checksums and the like are not needed. As with other connection-oriented protocols (TCP and SPP, for example), the connection-oriented Unix version provides flow control. In a similar fashion, as with other connectionless protocols (UDP and IDP, for example), the Unix domain datagram facility does *not* provide flow control. This has implications for user programs, since it is possible for a datagram client to send data so fast that buffer starvation can occur. If this happens, the sender must try to send the data repeatedly. For this reason alone, it is recommended that you use the connection-oriented Unix domain protocol.

The Unix domain protocols provide a feature that is not currently provided by any other protocol family: the ability to pass access rights from one process to another. We'll discuss this feature in more detail in Section 6.10 when we describe the passing of file descriptors between processes.

Unlike the other protocols we've covered, there is nothing like encapsulation performed on the Unix domain messages—their actual implementation is a kernel detail that need not concern us. As it turns out, 4.3BSD implements pipes using the connection-oriented Unix domain protocol.

The name space used by the Unix domain protocols consists of pathnames. A sample association could be

```
{unixstr, 0, /tmp/log.01528, 0, /dev/logfile}
```

The *protocol* here is `unixstr`, which stands for "Unix stream," the connection-oriented protocol. The *local-process* in this example is `/tmp/log.01528` and the *foreign-process* is `/dev/logfile`. We show both the *local-addr* and the *foreign-addr* as zero, since the pathnames on the local host are the only addresses used in this domain.† An association using the connectionless Unix protocol is similar, except that we use the term `unixdg` to specify the datagram protocol, the first member of the 5-tuple.

The 4.3BSD implementation creates a file in the filesystem with the specified pathname, although there are comments in the 4.3BSD manuals indicating that future versions might not create these files. This is somewhat misleading, however, because these filesystem entries are not true "files." For example, we cannot open these files with the `open` system call. These files have a type of `S_IFSOCK` as reported by the `stat` or `fstat` system calls.

† We could define the *local-addr* and the *foreign-addr* to be the pathnames, with the *local-process* and *foreign-process* both being zero. Our reason for specifying the association as we did is to reiterate that a host address is not required since the association is limited to processes on the local host.

6.4 Socket Addresses

Many of the BSD networking system calls require a pointer to a socket address structure as an argument. The definition of this structure is in <sys/socket.h>:

```
struct sockaddr {
  u_short  sa_family;    /* address family: AF_xxx value */
  char     sa_data[14];  /* up to 14 bytes of protocol-specific
                            address */
};
```

The contents of the 14 bytes of protocol-specific address are interpreted according to the type of address. For the Internet family, the following structures are defined in <netinet/in.h>:

```
struct in_addr {
  u_long  s_addr;              /* 32-bit netid/hostid */
                               /* network byte ordered */
};

struct sockaddr_in {
  short           sin_family;  /* AF_INET */
  u_short         sin_port;    /* 16-bit port number */
                               /* network byte ordered */
  struct in_addr  sin_addr;    /* 32-bit netid/hostid */
                               /* network byte ordered */
  char            sin_zero[8]; /* unused */
};
```

As we'll see throughout this chapter, almost every code example has

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

at the beginning. This header file provides C definitions and data type definitions (typedefs) that are used throughout the system. We will mainly use the names defined for the four unsigned integer datatypes, which we show in Figure 6.4. Unfortunately these four names differ between 4.3BSD and System V.

C Data type	4.3BSD	System V
unsigned char	u_char	unchar
unsigned short	u_short	ushort
unsigned int	u_int	uint
unsigned long	u_long	ulong

Figure 6.4 Unsigned data types defined in <sys/types.h>.

For the Xerox NS family, the following structures are defined in <netns/ns.h>:

```
union ns_host {
  u_char        c_host[6]; /* hostid addr as six bytes */
  u_short       s_host[3]; /* hostid addr as three 16-bit shorts */
                           /* network byte ordered */
};

union ns_net {
  u_char        c_net[4];  /* netid as four bytes */
  u_short       s_net[2];  /* netid as two 16-bit shorts */
                           /* network byte ordered */
};

struct ns_addr {     /* here is the combined 12-byte XNS address */
  union ns_net   x_net;    /* 4-byte netid */
  union ns_host  x_host;   /* 6-byte hostid */
  u_short        x_port;   /* 2-byte port (XNS "socket") */
                           /* network byte ordered */
};

struct sockaddr_ns {
  u_short          sns_family;  /* AF_NS */
  struct ns_addr   sns_addr;    /* the 12-byte XNS address */
  char             sns_zero[2]; /* unused */
};
#define  sns_port  sns_addr.x_port
```

Things are more complicated with XNS, as some of the network code wants to get at the hostid and netid fields in different ways.

For the Unix domain, the following structure is defined in <sys/un.h>:

```
struct sockaddr_un {
  short  sun_family;      /* AF_UNIX */
  char   sun_path[108];   /* pathname */
};
```

(You may have noticed the discrepancies in the declarations of the first two bytes of these address structures, the `XX_family` members. The Internet and Unix domain members are declared as short integers, while the XNS member and the generic `sockaddr` member are unsigned short integers. Fortunately the values stored in these variables, the `AF_xxx` constants that we define in the next section, all have values between 1 and 20, so this discrepancy in the data types doesn't matter.)

Unlike most Unix system calls, the BSD network system calls don't assume that the Unix pathname in `sun_path` is terminated with a null byte. Notice that this protocol-specific structure is larger than the generic `sockaddr` structure, while the `sockaddr_in` and `sockaddr_ns` structures were both identical in size (16 bytes) to

the generic structure. The system interface currently supports structures up to 110 bytes, but some of the generic system network routines (the routing tables in the kernel, for example) only support the 16-byte structures. The Unix domain protocols, however, can be larger than 16 bytes since they don't use these facilities. To handle socket address structures of different sizes, the system interface always passes the size of the address structure, in addition to a pointer to the structure, as we describe in the next paragraph.

A picture of these socket address structures is shown in Figure 6.5.

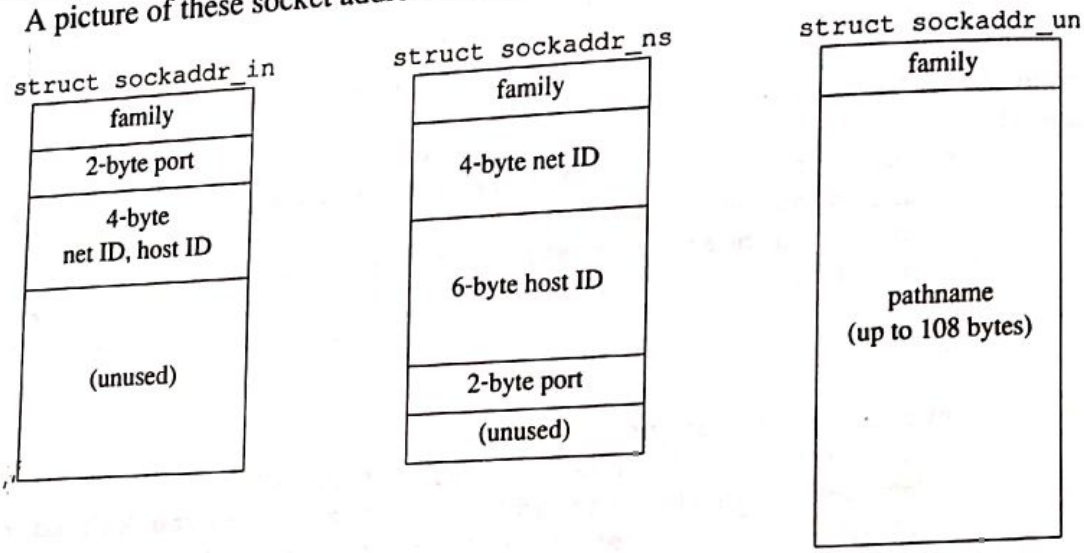

Figure 6.5 Socket address structures for Internet, XNS and Unix families.

The network system calls, such as `connect` and `bind`, work with any of the supported domains, so a technique must be used to pass any of the socket address structures shown in Figure 6.5, `sockaddr_in`, `sockaddr_ns`, or `sockaddr_un`, to these generic system calls. The network system calls take two arguments: the address of the generic `sockaddr` structure and the size of the actual protocol-specific structure. What the caller must do is provide the address of the protocol-specific structure as an argument, casting this pointer into a pointer to a generic `sockaddr` structure. For example, to invoke the `connect` system call for an Internet domain socket, we write

```
struct sockaddr_in serv_addr; /* Internet-specific addr struct */
   ...
(fill in Internet-specific information)
   ...
connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
```

The second argument is a pointer to the Internet address structure and the third argument is its size (16 bytes).

The designers of the socket interface could have also chosen to use a C union to define the socket address structure. The declaration could look like

```
struct sock_addr {
  short  sa_family;      /* AF_xxx value */
  union {
    struct sockaddr_in  sa_in;     /* Internet address */
    struct sockaddr_ns  sa_ns;     /* XNS address */
    struct sockaddr_un  sa_un;     /* Unix address */
  } sa_val;
};
```

The problem with this approach is that the size of the union is determined by the size of the largest member, which is the Unix domain address. This would cause every `sock_addr` structure to be 110 bytes in size, even though Internet and XNS addresses need only 8 bytes and 14 bytes, respectively. (Note that in this case the 8 bytes of zero at the end of an Internet `sockaddr_in` and the 2 bytes of zero at the end of an XNS `sockaddr_ns` are not needed.) One advantage of a union is that the size of the structure would not have to be an argument (and sometimes a result) to the system calls, as we'll see below. The actual interface design and the choice of a generic `sockaddr` structure along with the protocol-specific socket address structures, was a compromise.

6.5 Elementary Socket System Calls

We now describe the elementary system calls required to perform network programming. In the following section we use these system calls to develop some networking examples.

`socket` System Call

To do network I/O, the first thing a process must do is call the `socket` system call, specifying the type of communication protocol desired (Internet TCP, Internet UDP, XNS SPP, etc.).

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

int socket(int family, int type, int protocol);
```

The *family* is one of

`AF_UNIX`	Unix internal protocols
`AF_INET`	Internet protocols
`AF_NS`	Xerox NS protocols
`AF_IMPLINK`	IMP link layer

The `AF_` prefix stands for "address family." There is another set of terms that is defined, starting with a `PF_` prefix, which stands for "protocol family": `PF_UNIX`, `PF_INET`, `PF_NS`, and `PF_IMPLINK`. Either term for a given family can be used, as they are equivalent.

An IMP is an *Interface Message Processor*. This is an intelligent packet switching node that was referred to in our description of the ARPANET in the previous chapter. These nodes are connected with point-to-point links, typically leased telephone lines. The BSD system provides a raw socket interface to the IMP, which is what the `AF_IMPLINK` address family is used for. We won't concern ourselves further with this special purpose network interface.

The socket *type* is one of the following:

`SOCK_STREAM`	stream socket
`SOCK_DGRAM`	datagram socket
`SOCK_RAW`	raw socket
`SOCK_SEQPACKET`	sequenced packet socket
`SOCK_RDM`	reliably delivered message socket (not implemented yet)

Not all combinations of socket *family* and *type* are valid. Figure 6.6 shows the valid combinations, along with the actual protocol that is selected by the pair.

	`AF_UNIX`	`AF_INET`	`AF_NS`
`SOCK_STREAM`	Yes	TCP	SPP
`SOCK_DGRAM`	Yes	UDP	IDP
`SOCK_RAW`		IP	Yes
`SOCK_SEQPACKET`			SPP

Figure 6.6 Protocols corresponding to socket *family* and *type*.

The boxes marked "Yes" are valid, but don't have handy acronyms. The empty boxes are not implemented.

The *protocol* argument to the `socket` system call is typically set to 0 for most user applications. There are specialized applications, however, that specify a *protocol* value, to use a specific protocol. The valid combinations are shown in Figure 6.7.

family	*type*	*protocol*	Actual protocol
`AF_INET`	`SOCK_DGRAM`	`IPPROTO_UDP`	UDP
`AF_INET`	`SOCK_STREAM`	`IPPROTO_TCP`	TCP
`AF_INET`	`SOCK_RAW`	`IPPROTO_ICMP`	ICMP
`AF_INET`	`SOCK_RAW`	`IPPROTO_RAW`	(raw)
`AF_NS`	`SOCK_STREAM`	`NSPROTO_SPP`	SPP
`AF_NS`	`SOCK_SEQPACKET`	`NSPROTO_SPP`	SPP
`AF_NS`	`SOCK_RAW`	`NSPROTO_ERROR`	Error protocol
`AF_NS`	`SOCK_RAW`	`NSPROTO_RAW`	(raw)

Figure 6.7 Combinations of *family*, *type*, and *protocol*.

The IPPROTO_xxx constants are defined in the file <netinet/in.h> and the NSPROTO_xxx constants are defined in the file <netns/ns.h>. In Section 11.2 we'll show the Internet ping program, which uses the ICMP protocol.

The socket system call returns a small integer value, similar to a file descriptor. We'll call this a socket descriptor, or a *sockfd*. To obtain this socket descriptor, all we've specified is an address family and the socket type (stream, datagram, etc.). For an association

{protocol, local-addr, local-process, foreign-addr, foreign-process}

all the socket system call specifies is one element of this 5-tuple, the *protocol*. Before the socket descriptor is of any real use, the remaining four elements of the association must be specified. To show what a typical process does next, we'll differentiate between both client and server, and between a connection-oriented protocol and a connectionless protocol. Figure 6.8 shows the typical system calls for each case.

	protocol	*local-addr, local-process*	*foreign-addr, foreign-process*
connection-oriented server	socket()	bind()	listen(), accept()
connection-oriented client	socket()	connect()	
connectionless server	socket()	bind()	recvfrom()
connectionless client	socket()	bind()	sendto()

Figure 6.8 Socket system calls and association elements.

socketpair System Call

This system call is implemented only for the Unix domain.

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

int socketpair(int family, int type, int protocol, int sockvec[2]);
```

This returns two socket descriptors, *sockvec[0]* and *sockvec[1]*, that are unnamed and connected. This system call is similar to the pipe system call, but socketpair returns a pair of socket descriptors, not file descriptors. Additionally, the two socket descriptors returned by socketpair are bidirectional, unlike pipes, which are unidirectional. Recalling our pipe examples from Section 3.4, we had to execute the pipe system call twice, obtaining four file descriptors, to get a bidirectional flow of data between two processes. With sockets this isn't required. We'll call these bidirectional, connection-oriented, Unix domain sockets *stream pipes*. We return to these stream pipes in Section 6.9.

Since this system call is limited to the Unix domain, there are only two allowable versions of it

```
int     rc, sockfd[2];

rc = socketpair(AF_UNIX, SOCK_STREAM, 0, sockfd);
```

or

```
rc = socketpair(AF_UNIX, SOCK_DGRAM, 0, sockfd);
```

`bind` System Call

The `bind` system call assigns a name to an unnamed socket.

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

int bind(int sockfd, struct sockaddr *myaddr, int addrlen);
```

The second argument is a pointer to a protocol-specific address and the third argument is the size of this address structure. There are three uses of `bind`.

1. Servers register their well-known address with the system. It tells the system "this is my address and any messages received for this address are to be given to me." Both connection-oriented and connectionless servers need to do this before accepting client requests.
2. A client can register a specific address for itself.
3. A connectionless client needs to assure that the system assigns it some unique address, so that the other end (the server) has a valid return address to send its responses to. This corresponds to making certain an envelope has a valid return address, if we expect to get a reply from the person we send the letter to.

The `bind` system call fills in the *local-addr* and *local-process* elements of the association 5-tuple.

`connect` System Call

A client process `connects` a socket descriptor following the `socket` system call to establish a connection with a server.

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

int connect(int sockfd, struct sockaddr *servaddr, int addrlen);
```

The *sockfd* is a socket descriptor that was returned by the `socket` system call. The second and third arguments are a pointer to a socket address, and its size, as described earlier.

For most connection-oriented protocols (TCP and SPP, for example), the `connect` system call results in the actual establishment of a connection between the local system and the foreign system. Messages are typically exchanged between the two systems and specific parameters relating to the conversation might be agreed on (buffer sizes, amount of data to exchange between acknowledgments, etc.). In these cases the `connect` system call does not return until the connection is established, or an error is returned to the process. Section 12.15 of Comer [1988] discusses the three-way handshake used by TCP to establish a connection. Section 7.4 of Xerox [1981b] discusses a similar technique used by SPP.

The client does not have to `bind` a local address before calling `connect`. The connection typically causes these four elements of the association 5-tuple to be assigned: *local-addr*, *local-process*, *foreign-addr*, and *foreign-process*. In all the connection-oriented client examples that we'll show, we'll let `connect` assign the local address. This is what we diagramed in Figure 6.2 and Figure 6.8.

A connectionless client can also use the `connect` system call, but the scenario is different from what we just described. For a connectionless protocol, all that is done by the `connect` system call is to store the *servaddr* specified by the process, so that the system knows where to send any future data that the process writes to the *sockfd* descriptor. Also, only datagrams from this address will be received by the socket. In this case the `connect` system call returns immediately and there is not an actual exchange of messages between the local system and the foreign system.

One advantage of `connect`ing a socket associated with a connectionless protocol is that we don't need to specify the destination address for every datagram that we send. We can use the `read`, `write`, `recv`, and `send` system calls. (We describe the system calls used for socket I/O later in this section.)

There is another feature provided for connectionless clients that call `connect`. If the datagram protocol supports notification for invalid addresses, then the protocol routine can inform the user process if it sends a datagram to an invalid address. For example, the Internet protocols specify that a host should generate an ICMP port unreachable message if it receives a UDP datagram specifying a UDP port for which no process is waiting to read from. The host that sent the UDP datagram and receives this ICMP message can try to identify the process that sent the datagram, and notify the process. 4.3BSD, for example, notifies the process with a "connection refused" error (`ECONNREFUSED`) on the next system call for this socket, only if the process had `connect`ed the socket to the destination address. We'll see an example of this with our Internet time client in Section 10.2.

Note that the `connect` and `bind` system calls require only the pointer to the address structure and its size as arguments, not the protocol. This is because the protocol is available from two places: first, the `AF_xxx` value is always contained in the first two

bytes of the socket address structure; second, these system calls all require a socket descriptor as an argument, and this descriptor is always associated with a single protocol family, from the `socket` system call that created the descriptor.

`listen` System Call

This system call is used by a connection-oriented server to indicate that it is willing to receive connections.

```
int listen(int sockfd, int backlog);
```

It is usually executed after both the `socket` and `bind` system calls, and immediately before the `accept` system call. The *backlog* argument specifies how many connection requests can be queued by the system while it waits for the server to execute the `accept` system call. This argument is usually specified as 5, the maximum value currently allowed.

Recall our description of a concurrent connection-oriented server in Chapter 4. In the time that it takes a server to handle the request of an `accept` (the time required for the server to `fork` a child process and then have the parent process execute another `accept`), it is possible for additional connection requests to arrive from other clients. What the *backlog* argument refers to is this queue of pending requests for connections.

`accept` System Call

After a connection-oriented server executes the `listen` system call described above, an actual connection from some client process is waited for by having the server execute the `accept` system call.

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

int accept(int sockfd, struct sockaddr *peer, int *addrlen);
```

`accept` takes the first connection request on the queue and creates another socket with the same properties as *sockfd*. If there are no connection requests pending, this call blocks the caller until one arrives. (We discuss how a socket can be specified as non-blocking in Section 6.11.)

The *peer* and *addrlen* arguments are used to return the address of the connected peer process (the client). *addrlen* is called a value–result argument: the caller sets its value before the system call, and the system call stores a result in the variable. Often these value–result arguments are integers that the caller sets to the size of a buffer, with the system call changing this value on the return to the actual amount of data stored in the buffer. For this system call the caller sets *addrlen* to the size of the `sockaddr` structure whose address is passed as the *peer* argument. On return, *addrlen* contains the actual number of bytes that the system call stores in the *peer* argument. For the Internet and XNS address structures, since their sizes are constant (16 bytes), the value that we store

in the *addrlen* argument (16) is the actual number of bytes that the system call returns in the *peer* argument. The size of the structure can differ from the actual size of the address returned with Unix domain addresses.

This system call returns up to three values: an integer return code that is either an error indication or a new socket descriptor, the address of the client process (*peer*), and the size of this address (*addrlen*).

`accept` automatically creates a new socket descriptor, assuming the server is a concurrent server. If this is the case, the typical scenario is

```
int   sockfd, newsockfd;

if ( (sockfd = socket( ... )) < 0)
        err_sys("socket error");
if (bind(sockfd, ... ) < 0)
        err_sys("bind error");
if (listen(sockfd, 5) < 0)
        err_sys("listen error");

for ( ; ; ) {
        newsockfd = accept(sockfd, ... );   /* blocks */
        if (newsockfd < 0)
                err_sys("accept error");

        if (fork() == 0) {
                close(sockfd);          /* child */
                doit(newsockfd);        /* process the request */
                exit(0);
        }

        close(newsockfd);               /* parent */
}
```

When a connection request is received and accepted, the process `forks`, with the child process servicing the connection and the parent process waiting for another connection request. The new socket descriptor returned by `accept` refers to a complete association

{protocol, local-addr, local-process, foreign-addr, foreign-process}

All five elements of the 5-tuple associated with `newsockfd` have been filled in on return from `accept`. On the other hand, the `sockfd` argument that is passed to `accept` only has three elements of the 5-tuple filled in. The *foreign-addr* and *foreign-process* are still unspecified, and remain so after `accept` returns. This allows the original process (the parent) to `accept` another connection using `sockfd`, without having to create another socket descriptor. Since most connection-oriented servers are concurrent servers, and not iterative servers, the system goes ahead and creates a new socket automatically as part of the `accept` system call. If we wanted an iterative server, the scenario would be

```
int   sockfd, newsockfd;

if ( (sockfd = socket( ... ) < 0)
        err_sys("socket error");
bind(sockfd, ... ) < 0)
        err_sys("bind error");
listen(sockfd, 5) < 0)
        err_sys("listen error");

for ( ; ; ) {
        newsockfd = accept(sockfd, ... );    /* blocks */
        if (newsockfd < 0)
                err_sys("accept error");

        doit(newsockfd);        /* process the request */
        close(newsockfd);
}
```

Here the server handles the request using the connected socket descriptor, `newsockfd`. It then terminates the connection with a `close` and waits for another connection using the original descriptor, `sockfd`, which still has its *foreign-addr* and *foreign-process* unspecified.

`send`, `sendto`, `recv` and `recvfrom` System Calls

These system calls are similar to the standard `read` and `write` system calls, but additional arguments are required.

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

int send(int sockfd, char *buff, int nbytes, int flags);

int sendto(int sockfd, char *buff, int nbytes, int flags,
           struct sockaddr *to, int addrlen);

int recv(int sockfd, char *buff, int nbytes, int flags);

int recvfrom(int sockfd, char *buff, int nbytes, int flags,
             struct sockaddr *from, int *addrlen);
```

The first three arguments, *sockfd*, *buff*, and *nbytes*, to the four system calls are similar to the first three arguments for `read` and `write`.

The *flags* argument is either zero, or is formed by or'ing one of the following constants:

MSG_OOB	send or receive out-of-band data
MSG_PEEK	peek at incoming message (recv or recvfrom)
MSG_DONTROUTE	bypass routing (send or sendto)

The MSG_PEEK flag lets the caller look at the data that's available to be read, without having the system discard the data after the recv or recvfrom returns. We'll discuss the MSG_OOB option in Section 6.14 and the MSG_DONTROUTE option in Section 6.11. The *to* argument for sendto specifies the protocol-specific address of where the data is to be sent. Since this address is protocol-specific, its length must be specified by *addrlen*. The recvfrom system call fills in the protocol-specific address of who sent the data into *from*. The length of this address is also returned to the caller in *addrlen*. Note that the final argument to sendto is an integer value, while the final argument to recvfrom is a pointer to an integer value (a value–result argument).

All four system calls return the length of the data that was written or read as the value of the function. In the typical use of recvfrom, with a connectionless protocol, the return value is the length of the datagram that was received.

close System Call

The normal Unix close system call is also used to close a socket.

```
int close(int fd);
```

If the socket being closed is associated with a protocol that promises reliable delivery (e.g., TCP or SPP), the system must assure that any data within the kernel that still has to be transmitted or acknowledged, is sent. Normally the system returns from the close immediately, but the kernel still tries to send any data already queued.

Later in this chapter we'll describe the SO_LINGER socket option. This socket option allows a process to specify that either: (a) the close should try to send any queued data, or (b) any data queued to be sent should be flushed.

Byte Ordering Routines

The following four functions handle the potential byte order differences between different computer architectures and different network protocols. We detailed some of the differences in Chapter 4.

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

u_long htonl(u_long hostlong);

u_short htons(u_short hostshort);

u_long ntohl(u_long netlong);

u_short ntohs(u_short netshort);
```

These functions were designed for the Internet protocols. Fortunately, the XNS protocols use the same byte ordering as the Internet. On those systems that have the same byte ordering as the Internet protocols (Motorola 68000-based systems, for example), these four functions are null macros. The conversions done by these functions are shown in Figure 6.9.

htonl	convert host–to–network, long integer
htons	convert host–to–network, short integer
ntohl	convert network–to–host, long integer
ntohs	convert network–to–host, short integer

Figure 6.9 Byte ordering functions.

These four functions are operate on unsigned integer values, although they work just as well on signed integers. Implicit in these functions is that a short integer occupies 16 bits and a long integer 32 bits.

Byte Operations

There are multibyte fields in the various socket address structures that need to be manipulated. Some of these fields, however, are not C integer fields, so some other technique must be used to operate on them portably.

4.3BSD defines the following three routines that operate on user-defined byte strings. By user-defined we mean they are not the standard C character strings (which are always terminated by a null byte). The user-defined byte strings can have null bytes within them and these do *not* signify the end of the string. Instead, we must specify the length of each string as an argument to the function.

```
bcopy(char *src, char *dest, int nbytes);

bzero(char *dest, int nbytes);

int bcmp(char *ptr1, char *ptr2, int nbytes);
```

`bcopy` moves the specified number of bytes from the source to the destination. Note that the order of the two pointer arguments is different from the order used by the standard I/O `strcpy` function. The `bzero` function writes the specified number of null

bytes to the specified destination. `bcmp` compare two arbitrary byte strings. The return value is zero if the two user-defined byte strings are identical, otherwise its nonzero. Note that this differs from the return value from the standard I/O `strcmp` function.

We'll use these three functions throughout the remaining chapters, usually to operate on socket address structures.

System V has three similar functions, `memcpy`, `memset`, and `memcmp`. But most socket libraries for System V supply functions with the 4.3BSD names, for compatibility. Details of the System V functions are on the *memory*(3) manual page.

Address Conversion Routines

As mentioned in Section 5.2, an Internet address is usually written in the dotted-decimal format, for example `192.43.235.1`. The following functions convert between the dotted-decimal format and an `in_addr` structure.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_addr(char *ptr);

char *inet_ntoa(struct in_addr inaddr);
```

The first of these, `inet_addr` converts a charter string in dotted-decimal notation to a 32-bit Internet address.† The `inet_ntoa` function does the reverse conversion.

Similar functions exist for handling XNS addresses, but there is not as clear a standard for representing XNS addresses in a readable form. The format of an XNS address is typically three fields, delimited by a separator character:

<network-ID><separator><host-ID><separator><port#>

For example, we write

```
123:02.07.01.00.a1.62:6001
```

using a colon as the field separator and a period as the byte separator for the host ID field. The function `ns_addr` defined below is lenient in what it accepts. First it tries to divide the character string into one to three fields, accepting either a period, colon, or pound sign as the separator. Each field is then scanned for byte separators, either a colon or a period. (Obviously you can't use the same character as the field separator and the byte separator.)

† Note that this function returns an unsigned long integer, when it should return an `in_addr` structure. Some systems that are based on 4.3BSD have corrected this.

```
#include <sys/types.h>
#include <netns/ns.h>

struct ns_addr ns_addr(char *ptr);

char *ns_ntoa(struct ns_addr ns);
```

The first of these converts a character string representation of an XNS network ID, host ID, and port number into an `ns_addr` structure. The second function does the reverse conversion. This function generates strings using a period to separate the three fields, with each field in hexadecimal.

6.6 A Simple Example

Now we'll use the elementary system calls from the previous section to provide 12 complete programs, showing examples of client–servers using each of the three protocol families available with 4.3BSD. We have 12 programs since we have a connection-oriented server, a connection-oriented client, a connectionless server and a connectionless client for each of the three protocol families (Internet, XNS, and Unix domain). The programs do the following:

1. The client reads a line from its standard input and writes the line to the server.
2. The server reads a line from its network input and echoes the line back to the client.
3. The client reads the echoed line and prints it on its standard output.

This is an example of what is called an echo server. While we develop our own implementation of an echo server, most TCP/IP implementations provide such a server, using both TCP and UDP. See RFC 862 [Postel 1983a] for the official specification. Similarly, most XNS implementations also provide an echo server that is different from our implementation. We'll present a client for the standard XNS echo server in Section 11.3.

A pair of client–server programs to echo input lines is a good example of a network application. All the steps normally required to implement any client–server are illustrated by this example. All you need to do with this echo example, to expand it into your own application, is change what the server does with the input it receives from its clients.

In all these examples, we have "hard-coded" protocol-specific constants such as addresses and ports. There are two reasons for this. First, you should understand exactly what needs to be stored in the protocol-specific address structures. Second, we have not yet covered the library functions provided by 4.3BSD that make this more portable. These functions are covered in Section 8.2.

In these examples we have coded the connection-oriented Internet and Unix servers as concurrent servers, and the connection-oriented XNS server as an iterative server. This is to show examples of both types of servers.

6.7 Advanced Socket System Calls

We now proceed to the more advanced socket system calls, which we need in later sections of this chapter, and in some of the later chapters of this text.

`readv` and `writev` System Calls

4.3BSD provides what are called *scatter read* and *gather write* variants of the standard `read` and `write` system calls that we described in Section 2.3. What these two variants provide is the ability to read into or write from noncontiguous buffers.

```
#include <sys/types.h>
#include <sys/uio.h>

int writev(int fd, struct iovec iov[], int iovcount);

int readv(int fd, struct iovec iov[], int iovcount);
```

These two system calls use the following structure that is defined in `<sys/uio.h>`:

```
struct iovec {
  caddr_t  iov_base;    /* starting address of buffer */
  int      iov_len;     /* size of buffer in bytes */
};
```

The `writev` system call writes the buffers specified by *iov[0], iov[1],* through *iov[iovcount–1]*. The `readv` system call does the input equivalent. `readv` always fills one buffer (as specified by the `iov_len` value) before proceeding to the next buffer in the *iov* array. Both system calls return the total number of bytes read or written.

Consider, as an example, a hypothetical function named `write_hdr` that writes a buffer preceded by some header. The actual format of the header doesn't concern us.

```
int
write_hdr(fd, buff, nbytes)
int     fd;
char    *buff;
int     nbytes;
{
        int             n;
        struct hdr_info header;

        /* ... set up the header as required ... */

        if (write(fd, header, sizeof(header)) != sizeof(header))
                return(-1);

        if (write(fd, buff, nbytes) != nbytes)
                return(-1);

        return(nbytes);
}
```

Here two `write` system calls are used. Another way to do this is to allocate a temporary buffer, build the header in the beginning of the buffer, then copy the data into the buffer and do a single `write` of the entire buffer. A third way to do it (to avoid the extra `write` or to avoid the copy operation) is to force the caller to allocate space at the beginning of their buffer for the header. None of these three alternatives is optimal, which is why the `writev` system call (and its `readv` counterpart) were introduced.

The types of operation described here (adding information to either the beginning or end of a user's buffer) is just a form of encapsulation. The `writev` alternative for our example is

```
#include <sys/types.h>
#include <sys/uio.h>

int
write_hdr(fd, buff, nbytes)
int     fd;
char    *buff;
int     nbytes;
{
        struct hdr_info   header;
        struct iovec      iov[2];

        /* ... set up the header as required ... */

        iov[0].iov_base = (char *) &header;
        iov[0].iov_len  = sizeof(header);

        iov[1].iov_base = buff;
        iov[1].iov_len  = nbytes;

        if (writev(fd, &iovec[0], 2) != sizeof(header) + nbytes)
                return(-1);

        return(nbytes);
}
```

The `readv` and `writev` system calls can be used with any descriptor, not just sockets, although they are not portable beyond 4.3BSD. (One could write user functions named `readv` and `writev` that emulate these 4.3BSD system calls.)

The `writev` system call is an atomic operation. This is important if the descriptor refers to a record-based entity, such as a datagram socket or a magnetic tape drive. For example, a `writev` to a datagram socket produces a single datagram, whereas multiple `write`s would produce multiple datagrams. The only correct way to emulate `writev` is to copy all the data into a single buffer and then execute a single `write` system call.

`sendmsg` and `recvmsg` System Calls

These two system calls are the most general of all the read and write system calls.

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

int sendmsg(int sockfd, struct msghdr msg[], int flags);

int recvmsg(int sockfd, struct msghdr msg[], int flags);
```

These use the `msghdr` structure that is defined in `<sys/socket.h>`

```
struct msghdr {
  caddr_t        msg_name;          /* optional address */
  int            msg_namelen;       /* size of address */
  struct iovec   *msg_iov;          /* scatter/gather array */
  int            msg_iovlen;        /* # elements in msg_iov */
  caddr_t        msg_accrights;     /* access rights sent/recvd */
  int            msg_accrightslen;
};
```

The `msg_name` and `msg_namelen` fields are used when the socket is not connected, similar to the final two arguments to the `recvfrom` and `sendto` system calls. The `msg_name` field can be specified as a NULL pointer if a name is either not required or not desired. The `msg_iov` and `msg_iovlen` fields are used for scatter read and gather write operations, as described above for the `readv` and `writev` system calls. The final two elements of the structure, `msg_accrights` and `msg_accrightslen` deal with the passing and receiving of access rights between processes. We discuss this in Section 6.10 when we explain the passing of file descriptors between processes. The `flags` argument is the same as with the `send` and `recv` system calls, which we described in Section 6.5.

Figure 6.10 compares the five different read and write function groups.

System call	Any descriptor	Only socket descriptor	Single read/write buffer	Scatter/ gather read/write	Optional flags	Optional peer address	Optional access rights
`read, write`	•		•				
`readv, writev`	•			•			
`recv, send`		•	•		•		
`recvfrom, sendto`		•	•		•	•	
`recvmsg, sendmsg`		•		•	•	•	•

Figure 6.10 Read and write system call variants.

`getpeername` System Call

This system call returns the name of the peer process that is connected to a given socket.

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

int getpeername(int sockfd, struct sockaddr *peer, int *addrlen);
```

When we say that this system call returns the "name" of the peer process, we mean that it returns the *foreign-addr* and *foreign-process* elements of the 5-tuple associated with *sockfd*. Note that the final argument is a value–result argument.

The original 4.3BSD release did not support this system call for Unix domain sockets, but fixes exist to correct this limitation.

`getsockname` System Call

This system call returns the name associated with a socket.

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

int getsockname(int sockfd, struct sockaddr *peer, int *addrlen);
```

This call returns the *local-addr* and *local-process* elements of an association. As with the `getpeername` system call, the final argument is a value–result argument.

The original 4.3BSD release did not support this system call for Unix domain sockets, but fixes exist to correct this limitation.

`getsockopt` and `setsockopt` System Calls

These two system calls manipulate the options associated with a socket. We'll describe them in more detail in Section 6.11, along with two other system calls, `fcntl` and `ioctl`, that can modify the properties of a socket.

`shutdown` System Call

The normal way to terminate a network connection is to call the `close` system call. As we mentioned earlier, `close` normally attempts to deliver any data that is still to be sent. But the `shutdown` system call provides more control over a full-duplex connection.

```
int shutdown(int sockfd, int howto);
```

If the *howto* argument is 0, no more data can be received on the socket. A value of 1 causes no more output to be allowed on the socket. A value of 2 causes both sends and receives to be disallowed.

Remember that a socket is usually a *full-duplex* communication path. The data flowing in one direction is logically independent of the data going in the other direction. This is why `shutdown` allows either direction to be closed, independent of the other direction.

`select` System Call

This system call can be used when dealing with multiple descriptors. We'll discuss it in detail in Section 6.13.

6.8 Reserved Ports

There are two ways for a process to have an Internet port or an XNS port assigned to a socket.

- The process can request a specific port. This is typical for servers that need to assign a well-known port to a socket. All our server examples shown in Section 6.6 do this.
- The process can let the system automatically assign a port. For both the Internet domain and the XNS domain, specifying a port number of zero before calling `bind` requests the system to do this. Both our UDP and IDP client examples do this.

4.3BSD supports the concept of *reserved ports* in both the Internet and XNS domains. In the Internet domain, any TCP or UDP port in the range 1–1023 is reserved, and in the XNS domain ports in the range 1–2999 are reserved. A process is not allowed to `bind` a reserved port unless its effective user ID is zero (the superuser).

4.3BSD provides a library function that assigns a reserved TCP stream socket to its caller:

```
int rresvport(int *aport);
```

This function creates an Internet stream socket and `bind`s a reserved port to the socket. The socket descriptor is returned as the value of the function, unless an error occurs, in which case −1 is returned. Note that the argument to this function is the address of an integer (a value–result argument), not an integer value. The integer pointed to by *aport* is the first port number that the function attempts to `bind`. The caller typically initializes the starting port number to `IPPORT_RESERVED-1`. (The value of the constant `IPPORT_RESERVED` is defined to be 1024 in `<netinet/in.h>`.) If this `bind` fails with an `errno` of `EADDRINUSE`, then this function decrements the port number and tries again. If it finally reaches port 512 and finds it already in use, it sets `errno` to `EAGAIN`, and returns −1. If this function returns successfully, it not only returns the socket as the value of the function, but the port number is also returned in the location pointed to by *aport*.

Figure 6.11 summarizes the assignment of ports in the Internet and XNS domains.

	Internet	XNS
reserved ports	1–1023	1–2999
ports automatically assigned by system	1024–5000	3000–65535
ports assigned by `rresvport()`	512–1023	

Figure 6.11 Port assignment in the Internet and XNS domains.

Note that all ports assigned by standard Internet applications (FTP, TELNET, TFTP, SMTP, etc.) are between 1 and 255 and are therefore reserved. The servers for these applications must have superuser privileges when they create their socket and `bind` their well-known address. Also, the ports between 256 and 511 are considered reserved by 4.3BSD, but they are not currently used by any standard Internet application, and they are never allocated by the `rresvport` function.

The system doesn't automatically assign an Internet port greater than 5000. It leaves these ports for user-developed, nonprivileged servers. This provides a higher degree of certainty that a server can assign itself its well-known port, since any of the ports between 1024 and 5000 might be in use by some client.

Finally, note that any process that wants to obtain a privileged UDP, IDP, or SPP port has to do the same steps that `rresvport` does. There does not exist a standard library function similar to `rresvport` for these three protocols.

The concept of *reserved ports* as described above only handles the `bind`ing of ports to unbound sockets by the system. It is up to the application program (the server) that receives a request from a client with a reserved port, to consider the request as special or not. The server can obtain the address of the client using the `getpeername` system call. A connection-oriented server can also obtain the client's address from the *peer* argument of the `accept` system call, and a datagram server can obtain the address of the client from the *from* argument of the `recvfrom` system call. In Section 9.2 we consider reserved ports and their use by typical 4.3BSD servers for authentication.

6.11 Socket Options

There are a multitude of ways to set options that affect a socket.

- the `setsockopt` system call
- the `fcntl` system call
- the `ioctl` system call

We mentioned the `fcntl` and `ioctl` system calls in Section 2.3, and describe their effect on sockets later in this section.

`getsockopt` and `setsockopt` System Calls

These two system calls apply only to sockets.

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

int getsockopt(int sockfd, int level, int optname, char *optval, int *optlen);

int setsockopt(int sockfd, int level, int optname, char *optval, int optlen);
```

The *sockfd* argument must refer to an open socket descriptor. The *level* specifies who in the system is to interpret the option: the general socket code, the TCP/IP code, or the XNS code. Some options are protocol-specific and others are applicable to all sockets.

The *optval* argument is a pointer to a user variable from which an option value is set by `setsockopt`, or into which an option value is returned by `getsockopt`. Although this argument is type coerced into a "`char *`", none of the supported options are character values. The "data type" column in Figure 6.14 shows the data type of what the *optval* pointer must point to for each option. The *optlen* argument to `setsockopt` specifies the size of the variable. The *optlen* argument to `getsockopt` is a value–result parameter that we set to the size of the *optval* variable before the call. This size is then set by the system on return to specify the amount of data stored into the *optval* variable. This ability to handle variable-length options is only used by a single option currently: `IP_OPTIONS`. All the other options described below pass a fixed amount of data between the kernel and the user process.

Figure 6.14 provides a summary of all options that can be queried by `getsockopt` or set by `setsockopt`. There are two types of options: binary options that enable or disable a certain feature (flags), and options that fetch and return specific values that we can either set or examine (values). The column labeled "flag" specifies if the option is a flag option. When calling `getsockopt` for these flag options, *optval* is an integer. The value returned in *optval* is zero if the option is disabled, or nonzero if the option is enabled. Similarly, `setsockopt` requires a nonzero *optval* to turn the option on, and a zero value to turn the option off. If the "flag" column does not contain a "•" then the option is used to pass a value of the specified data type between the user process and the system.

level	optname	get	set	Description	flag	Data type
IPPROTO_IP	IP_OPTIONS	•	•	options in IP header		
IPPROTO_TCP	TCP_MAXSEG	•		get TCP maximum segment size		int
	TCP_NODELAY	•	•	don't delay send to coalesce packets	•	int
NSPROTO_SPP	SO_HEADERS_ON_INPUT	•	•	pass SPP header on input	•	int
	SO_HEADERS_ON_OUTPUT	•	•	pass SPP header on output	•	int
	SO_DEFAULT_HEADERS	•	•	set default SPP header for output		struct sphdr
	SO_LAST_HEADER	•		fetch last SPP header		struct sphdr
	SO_MTU	•	•	SPP MTU value		u_short
NSPROTO_PE	SO_SEQNO	•		generate unique PEX ID		long
NSPROTO_RAW	SO_HEADERS_ON_INPUT	•	•	pass IDP header on input	•	int
	SO_HEADERS_ON_OUTPUT	•	•	pass IDP header on output	•	int
	SO_DEFAULT_HEADERS	•	•	set default IDP header for output		struct idp
	SO_ALL_PACKETS	•	•	pass all IDP packets to user	•	int
	SO_NSIP_ROUTE		•	specify IP route for XNS		struct nsip_req
SOL_SOCKET	SO_BROADCAST	•	•	permit sending of broadcast msgs	•	int
	SO_DEBUG	•	•	turn on debugging info recording	•	int
	SO_DONTROUTE	•	•	just use interface addresses	•	int
	SO_ERROR	•		get error status and clear		int
	SO_KEEPALIVE	•	•	keep connections alive	•	int
	SO_LINGER	•	•	linger on close if data present		struct linger
	SO_OOBINLINE	•	•	leave received OOB data in-line	•	int
	SO_RCVBUF	•	•	receive buffer size		int
	SO_SNDBUF	•	•	send buffer size		int
	SO_RCVLOWAT	•	•	receive low-water mark		int
	SO_SNDLOWAT	•	•	send low-water mark		int
	SO_RCVTIMEO	•	•	receive timeout		int
	SO_SNDTIMEO	•	•	send timeout		int
	SO_REUSEADDR	•	•	allow local address reuse	•	int
	SO_TYPE	•		get socket type		int
	SO_USELOOPBACK	•	•	bypass hardware when possible	•	int

Figure 6.14 Socket options for `getsockopt` and `setsockopt`.

Let's first show a simple program that both fetches an option and sets an option.

```
/*
 * Example of getsockopt() and setsockopt().
 */

#include        <sys/types.h>
#include        <sys/socket.h>          /* for SOL_SOCKET and SO_xx values */
#include        <netinet/in.h>          /* for IPPROTO_TCP value */
#include        <netinet/tcp.h>         /* for TCP_MAXSEG value */

main()
{
        int     sockfd, maxseg, sendbuff, optlen;

        if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
```

```
            err_sys("can't create socket");

        /*
         * Fetch and print the TCP maximum segment size.
         */

        optlen = sizeof(maxseg);
        if (getsockopt(sockfd, IPPROTO_TCP, TCP_MAXSEG, (char *) &maxseg,
                       &optlen) < 0)
            err_sys("TCP_MAXSEG getsockopt error");
        printf("TCP maxseg = %d\n", maxseg);

        /*
         * Set the send buffer size, then fetch it and print its value.
         */

        sendbuff = 16384;       /* just some number for example purposes */
        if (setsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, (char *) &sendbuff,
                       sizeof(sendbuff)) < 0)
            err_sys("SO_SNDBUF setsockopt error");

        optlen = sizeof(sendbuff);
        if (getsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, (char *) &sendbuff,
                       &optlen) < 0)
            err_sys("SO_SNDBUF getsockopt error");
        printf("send buffer size = %d\n", sendbuff);
}
```

If we compile and execute this program, its output is

```
TCP maxseg = 512
send buffer size = 16384
```

The most confusing thing about the `getsockopt` and `setsockopt` system calls is the handling of the final argument.

The following list gives additional details on the options that affect a socket. When we say *allows us to specify* we are referring to the `setsockopt` system call. We say *when requested* to refer to the `getsockopt` system call. When a given option can only be used with a specific type of socket, the socket type (such as `SOCK_DGRAM`) is listed in parentheses first. For example, the `TCP_MAXSEG` option only applies to an Internet `SOCK_STREAM` socket.

`IP_OPTIONS` (Internet `SOCK_STREAM` or `SOCK_DGRAM`.) Allows us to set specific options in the IP header. Requires intimate knowledge of the IP header. See Section 7.8 of Comer [1988] for a description of the possible IP options. We'll use this option in our remote command server in Section 14.3 and our remote login server in Section 15.12. Both of these servers check for client connections that specify IP options, and if encountered, record the fact in the system log and disable the options.

TCP_MAXSEG (Internet SOCK_STREAM.) Returns the maximum segment size in use for the socket. The typical value for a 4.3BSD socket using an Ethernet is 1024 bytes. Note that the value for this option can only be examined, it cannot be set by us, since the system decides what size to use.

Note in the example program shown above that the value printed for this option was 512 and not 1024. This is because the socket was not yet connected, so TCP does not know the type of interface being used. It defaults the size to 512 until the socket is connected when it decides on the size to use.

TCP_NODELAY (Internet SOCK_STREAM.) When TCP is being used for a remote login (which we discuss in Chapter 15) there will be many small data packets sent from the client's system to the server. Each packet can contain a single character that the client enters, which is sent to the server for echoing and processing. On a fast LAN these small packets are not a problem, since the capacity of the network (its bandwidth) is usually adequate to handle all the packets. On a slower WAN, however, it is desirable to reduce the number of these small packets, to reduce the traffic on the network. The scheme used by 4.3BSD is to allow only a single small packet to be outstanding on a given TCP connection at any time. (For additional details on the 4.3BSD TCP output algorithms, see Section 12.7 of Leffler et al. [1989].) On a fast LAN this isn't a problem since the round-trip time for a 1-character packet is usually less than the time between a user entering successive characters on a terminal. On a slower WAN, what happens is the client's TCP buffers input characters until the previous small packet is acknowledged. One problem, however, is if the client is sending small packets that are not keystrokes being entered on a terminal to be echoed by the server. This can happen if the client is sending input from a mouse, for example, on a windowed terminal. The client's input can consist of small packets that are not echoed by the server. The TCP_NODELAY option is used for these clients, to defeat the buffering algorithm described above, to allow the client's TCP to send small packets as soon as possible.

SO_HEADERS_ON_INPUT (XNS SOCK_DGRAM.) When set, the first 30 bytes of the data returned by the read and receive system calls is the IDP header.

(XNS SOCK_STREAM or SOCK_SEQPACKET.) When set, the first 12 bytes of the data returned by the read and receive system calls is the SPP header.

SO_HEADERS_ON_OUTPUT

(XNS SOCK_DGRAM.) When set, the first 30 bytes of the data passed to the send and write system calls must be the IDP header.

(XNS SOCK_STREAM or SOCK_SEQPACKET.) When set, the first 12 bytes of the data passed to the send and write system calls must be the SPP header.

SO_DEFAULT_HEADERS

This option provides a way for a user process to set and examine the XNS bridge fields that we described in Section 5.3.

(XNS SOCK_DGRAM.) Allows us to specify a default IDP header for outgoing packets. The actual variable pointed to by the *optval* argument must be a `struct idp` (the structure defining an IDP header). The only value used by the system from this header is the packet type field. When requested, the system returns its default IDP header, with the following fields filled in: packet type, local address, and foreign address. We'll show an example using this option in Section 11.3.

(XNS SOCK_STREAM or SOCK_SEQPACKET.) Allows us to specify a default SPP header for outgoing packets. The actual variable pointed to by the *optval* argument must be a `struct sphdr` (the structure defining an SPP header). The only values used by the system from this header are the End-of-Message bit and the Datastream Type. When requested, the system returns its default SPP header.

SO_LAST_HEADER

(XNS SOCK_STREAM or SOCK_SEQPACKET.) When requested, returns the most recent SPP header received. The actual variable pointed to by the *optval* argument must be a `struct sphdr` (the structure defining an SPP header).

SO_MTU

(XNS SOCK_STREAM or SOCK_SEQPACKET.) By default the SPP MTU is 534 bytes, the Xerox standard. This allows a maximum IDP packet size of 576 bytes, allowing 30 bytes for the IDP header and 12 bytes for the SPP header. Note that any value other than this default is frowned on by an actual Xerox system. This option applies only to SPP sockets, since the BSD implementation of XNS allows IDP packets up to the maximum size of the underlying interface. For an Ethernet, this maximum IDP packet size is 1500 bytes. Again, however, these larger than normal IDP packets can create problems with true Xerox systems.

SO_SEQNO (XNS SOCK_DGRAM.) The XNS Packet Exchange protocol requires a unique 32-bit ID field, called the transaction ID, with every packet. This ID should be unique among different processes on the same system. This socket option allows the caller to obtain a unique value. The kernel allocates these IDs to any requesting process, incrementing the ID each time a process needs one. Furthermore, the kernel initializes its starting ID whenever the system is rebooted, from a random source (usually the time-of-day clock). This way individual processes can obtain unique IDs and, if the system is taken down and rebooted quickly (while a packet is possibly being retransmitted by some other host on the network), the IDs provided after rebooting are different from those used earlier. Note that the value for this option can only be examined, it cannot be set by us.

SO_ALL_PACKETS (XNS SOCK_DGRAM.) When set, prevents the system from processing Error Protocol packets and SPP packets. Instead, these packets are passed to us.

SO_NSIP_ROUTE (XNS SOCK_DGRAM.) NSIP is an option that can be enabled when the 4.3BSD kernel is configured. If enabled, it allows XNS packets to be sent to another 4.3BSD system, encapsulated in Internet IP datagrams. This allows us, for example, to share a single data link on a WAN between Internet applications and XNS applications. Using this feature any two cooperating systems that are connected on an Internet using the TCP/IP protocols can also exchange XNS packets. This socket option allows us to specify an Internet address to be associated with an XNS address. We won't discuss this option any further in this text.

SO_BROADCAST (Internet or XNS SOCK_DGRAM.) Enables or disables the ability of the process to send broadcast messages. Broadcasting is only provided for datagram sockets and only on networks that support the concept of a broadcast message (the kernel does not provide a simulation of broadcasting through software).

SO_DEBUG Enables or disables low-level debugging within the kernel. This option allows the kernel to maintain a history of the recent packets that have been received or sent. With adequate knowledge of the kernel, this history can either be examined (by looking at kernel memory with a debugger) or printed on the console.

SO_DONTROUTE Specifies that outgoing messages are to bypass the normal routing mechanisms of the underlying protocol. Instead, the message is directed to the appropriate network interface, as specified by the network portion of the destination address. The equivalent of this

option can also be applied to individual datagrams using the MSG_DONTROUTE flag with the send, sendto, or sendmsg system calls.

SO_ERROR Returns to the caller the current contents of the variable so_error, which is defined in <sys/socketvar.h>. This variable holds the standard Unix error numbers (the same values found in the Unix errno variable) for the socket. This error variable is then cleared by the kernel.

SO_KEEPALIVE (Internet or XNS SOCK_STREAM.) Enables periodic transmissions on a connected socket, when no other data is being exchanged. If the other end does not respond to these messages, the connection is considered broken and the so_error variable is set to ETIMEDOUT.

SO_LINGER (Internet SOCK_STREAM.) This option determines what to do when unsent messages exist for a socket when the process executes a close on the socket. By default, the close returns immediately and the system attempts to deliver any unsent data. But if the linger option is set for the socket, the action taken depends on the linger time specified by the user. This option requires the following structure to be passed between the user process and the kernel. It is defined in <sys/socket.h>.

```
struct linger {
  int   l_onoff;   /* zero=off, nonzero=on */
  int   l_linger;  /* linger time, in seconds */
};
```

If the linger time is specified as zero, any remaining data to be sent is discarded when the socket is closed. If the linger time is nonzero, the system attempts to deliver any unsent data. Currently the actual value of a nonzero linger time is ignored, despite the comment in the structure definition.

SO_OOBINLINE (Internet SOCK_STREAM.) When set, specifies that out-of-band data also be placed in the normal input queue (i.e., in-line). When the out-of-band data is placed in-line, the MSG_OOB flag to the receive system calls is not needed to read the out-of-band data. This option applies only to TCP sockets, since the XNS SPP protocol specifies that out-of-band data also be placed in the normal input queue by default. We discuss out-of-band data in more detail in Section 6.14.

SO_RCVBUF and SO_SNDBUF
Specifies the size of the receive queue buffer or the send queue buffer for the socket. These options are only needed when we would like more buffer space than is provided by default. These options are not required for these cases, but they can improve

performance. For example, the BSD `rdump` command, which writes 32768-byte buffers using TCP, sets its `SO_SNDBUF` option value to 32768, overriding the default TCP send buffer of 4096 bytes. Similarly, the `/etc/rmt` daemon, which receives these buffers and writes them to tape, sets its `SO_RCVBUF` option to 32768 also. With 4.3BSD there is an upper limit of around 52,000 bytes for either of these buffer sizes. We'll see one use of these options when we look at the 4.3BSD remote tape server in Chapter 16.

`SO_RCVLOWAT` and `SO_SNDLOWAT`
: These two options specify the sizes of the receive and send low-water marks. These values are currently unused.

`SO_RCVTIMEO` and `SO_SNDTIMEO`
: These two options specify the receive and send timeout values. These values are not currently used for anything. Note that this does not imply that the reliable protocols do not use timers for detecting lost packets and lost connections. Timers are indeed used by TCP and SPP, for example, but these two socket options don't currently affect these timers.

`SO_REUSEADDR`
: (Internet `SOCK_STREAM` or `SOCK_DGRAM`.) Instructs the system to allow local addresses, the *local-process* portion of an association 5-tuple, to be reused. Normally the system does not allow a local address to be reused. When `connect` is called for the socket, the system still requires that the complete association be unique, as discussed earlier. We mentioned this requirement in Section 5.2 when discussing the use of TCP port numbers by FTP.

`SO_TYPE`
: Returns the socket type. The integer value returned is a value such as `SOCK_STREAM` or `SOCK_DGRAM`. This option is typically used by a process that inherits a process when it is started.

`SO_USELOOPBACK`
: This option is unused.

`fcntl` System Call

This system call affects an open file, referenced by the *fd* argument.

```
#include <fcntl.h>

int fcntl(int fd, int cmd, int arg);
```

As mentioned in Section 2.3, the *cmd* argument specifies the operation to be performed. Regarding sockets we are interested in the *cmd* values of `F_GETOWN`, `F_SETOWN`, `F_GETFL`, and `F_SETFL`.

The F_SETOWN command sets either the process ID or the process group ID to receive the SIGIO or SIGURG signals for the socket associated with *fd*. Every socket has an associated process group number, similar to the terminal process group number that we discussed in Chapter 2. For a socket, its process group number is initialized to zero and can be set by calling fcntl with the F_SETOWN command. (It can also be set by both the FIOSETOWN and SIOCSPGRP ioctls. As we said, there are many ways to set options that affect a socket.) The *arg* value for the F_SETOWN command can be either a positive integer, specifying a process ID, or a negative integer, specifying a process group ID. The F_GETOWN command returns, as the return value from the fcntl system call, either the process ID (a positive return value) or the process group ID (a negative value other than −1) associated with the socket. The difference between specifying a process or a process group to receive the signal is that the former causes only a single process to receive the signal, while the latter causes all processes in the process group (perhaps more than one) to receive the signal. This fcntl command is only available for terminals and sockets. We show an example of the F_SETOWN command in Section 6.12.

A file's flag bits are set and examined with the F_SETFL and F_GETFL commands. Figure 6.15 shows the *arg* values that can be used with the F_GETFL and F_SETFL commands.

arg	Meaning
FAPPEND	append on each write
FASYNC	signal process group when data ready
FCREAT	create if nonexistent
FEXCL	error if already created
FNDELAY	nonblocking I/O
FTRUNC	truncate to zero length

Figure 6.15 fcntl options for F_GETFL and F_SETFL commands.

Of these five values, only the following two are of interest to us now.

FNDELAY This option designates the socket as "nonblocking." An I/O request that cannot complete on a nonblocking socket is not done. Instead, return is made to the caller immediately and the global errno is set to EWOULDBLOCK.

This option affects the following system calls: accept, connect, read, readv, recv, recvfrom, recvmsg, send, sendto, sendmsg, write, and writev. Note that a connect on a connectionless socket cannot block, since all the system does is record the peer address that is to be used for future output. On a connection-oriented socket, however, the connect can take a while to execute, since it usually involves the exchange of actual information with the peer system. In this case a nonblocking socket returns immediately from the connect system call, but the errno value is set to EINPROGRESS instead of EWOULDBLOCK.

Note also that the output system calls (the five starting with `send` in the list above) do partial writes on a nonblocking socket when it makes sense (e.g., if the socket is a stream socket). When the socket has record boundaries associated with it (e.g., a datagram socket), if the entire record cannot be written, nothing is written.

`FASYNC` This option allows the receipt of asynchronous I/O signals. The `SIGIO` signal is sent to the process group of the socket when data is available to be read. We'll cover asynchronous I/O in the next section.

`ioctl` System Call

This system call affects an open file, referenced by the *fd* argument. The intent is for `ioctl`s to manipulate device options.

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

int ioctl(int fd, unsigned long request, char *arg);
```

System V defines the second argument to be an `int` instead of an `unsigned long`, but that is not a problem, since the header file `<sys/ioctl.h>` defines the constants that should be used for this argument.

We can divide the *requests* into four categories.

- file operations
- socket operations
- routing operations
- interface operations

Figure 6.16 lists the *requests* provided by 4.3BSD, along with the data type of what the *arg* address must point to. (Notice that none of the requests use the character-pointer data type specified in the function prototype.) We now give a brief description of these options.

`FIOCLEX` Sets the close-on-`exec` flag for the file descriptor. Similar to the `F_SETFD` command of `fcntl` (with an *arg* of one).

`FIONCLEX` Clears the close-on-`exec` flag for the file descriptor. Similar to the `F_SETFD` command of `fcntl` (with an *arg* of zero).

`FIONBIO` Set or clear the nonblocking I/O flag for the file. This flag accomplishes the same affect as the `FNDELAY` argument for the `F_SETFL` command to the `fcntl` system call. Note one difference between the two system calls, however: with the `F_SETFL` command of `fcntl` we must specify all the flag values for the file each time we want to

Category	*request*	Description	Data type
file	FIOCLEX	set exclusive use on fd	
	FIONCLEX	clear exclusive use	
	FIONBIO	set/clear nonblocking i/o	int
	FIOASYNC	set/clear asynchronous i/o	int
	FIONREAD	get # bytes to read	int
	FIOSETOWN	set owner	int
	FIOGETOWN	get owner	int
socket	SIOCSHIWAT	set high-water mark	int
	SIOCGHIWAT	get high-water mark	int
	SIOCSLOWAT	set low-water mark	int
	SIOCGLOWAT	get low-water mark	int
	SIOCATMARK	at out-of-band mark ?	int
	SIOCSPGRP	set process group	int
	SIOCGPGRP	get process group	int
routing	SIOCADDRT	add route	struct rtentry
	SIOCDELRT	delete route	struct rtentry
interface	SIOCSIFADDR	set ifnet address	struct ifreq
	SIOCGIFADDR	get ifnet address	struct ifreq
	SIOCSIFFLAGS	set ifnet flags	struct ifreq
	SIOCGIFFLAGS	get ifnet flags	struct ifreq
	SIOCGIFCONF	get ifnet list	struct ifconf
	SIOCSIFDSTADDR	set point-to-point address	struct ifreq
	SIOCGIFDSTADDR	get point-to-point address	struct ifreq
	SIOCGIFBRDADDR	get broadcast addr	struct ifreq
	SIOCSIFBRDADDR	set broadcast addr	struct ifreq
	SIOCGIFNETMASK	get net addr mask	struct ifreq
	SIOCSIFNETMASK	set net addr mask	struct ifreq
	SIOCGIFMETRIC	get IF metric	struct ifreq
	SIOCSIFMETRIC	set IF metric	struct ifreq
	SIOCSARP	set ARP entry	struct arpreq
	SIOCGARP	get ARP entry	struct arpreq
	SIOCDARP	delete ARP entry	struct arpreq

Figure 6.16 `ioctl` options for networking.

turn a specific flag on or off. With this `ioctl` call, however, we can turn a flag on or off, by specifying a zero or nonzero value for *arg*.

`FIOASYNC` Set or clear the flag that allows the receipt of asynchronous I/O signals (`SIGIO`). This flag accomplishes the same affect as the `FASYNC` argument for the `F_SETFL` command to the `fcntl` system call.

`FIONREAD` Returns in *arg* the number of bytes available to read from the file descriptor. This feature works for files, pipes, terminals and sockets.

`FIOSETOWN` Set either the process ID or the process group ID to receive the `SIGIO` and `SIGURG` signals. This *request* works only for terminals and sockets. This command is identical to an `fcntl` of `F_SETOWN`.

`FIOGETOWN` Get either the process ID or the process group ID that is set to receive the `SIGIO` and `SIGURG` signals. This *request* works only for terminals and sockets. This command is identical to an `fcntl` of `F_GETOWN`.

`SIOCSHIWAT` Set the high-water mark. This command is not currently implemented.

`SIOCGHIWAT` Get the high-water mark. This command is not currently implemented.

`SIOCSLOWAT` Set the low-water mark. This command is not currently implemented.

`SIOCGLOWAT` Get the low-water mark. This command is not currently implemented.

`SIOCATMARK` Return a zero or nonzero value, depending whether the specified socket's read pointer is currently at the out-of-band mark. We describe out-of-band data in more detail in Section 6.14.

`SIOCSPGRP` Equivalent to `FIOSETOWN` for a socket.

`SIOCGPGRP` Equivalent to `FIOGETOWN` for a socket.

The following *requests* are for lower level operations, usually relating to the actual network interface being used. We briefly mention their purpose. The interested reader should consult both of the references by Leffler et al. [1986a, 1986b] for additional information. Several of these commands are used by the network configuration program, `ifconfig`, which is described in Section 8 of the 4.3BSD Programmer's Manual. This program is usually invoked on system startup to initialize all the network interfaces on the system. Some of the commands related to routing are used by the BSD routing daemon, `routed`, which is also described in Section 8 of the BSD manual.

`SIOCADDRT` Add an entry to the interface routing table.

`SIOCDELRT` Delete an entry from the interface routing table.

`SIOCSIFADDR`
Set the interface address. Additionally the initialization function for the interface is also called.

`SIOCGIFADDR`
Get the interface address.

`SIOCSIFFLAGS`
Set the interface flags.

`SIOCGIFFLAGS`
Get the interface flags. The flags indicate, for example, if the interface is a point-to-point interface, if the interface supports broadcast addressing, if the interface is running, and so on.

`SIOCGIFCONF`
Get the interface configuration list. This command returns a list containing one `ifreq` structure for every interface currently is use by the

system. This command allows a user program to determine at run time, the interfaces on the system. A program can then go through this list and determine which interfaces it is interested in (all interfaces that are point-to-point Internet interfaces, for example).

`SIOCSIFDSTADDR`
Set the point-to-point interface address.

`SIOCGIFDSTADDR`
Get the point-to-point interface address.

`SIOCSIFBRDADDR`
Set the broadcast address for the interface.

`SIOCGIFBRDADDR`
Get the broadcast address for the interface.

`SIOCSIFNETMASK`
Set the mask for the network portion of the interface address. For Internet addresses the network mask defines the netid portion of the 32-bit address. If this mask contains more bits that would normally be indicated by the class of Internet address (class A, B, or C) then subnets are being used. Refer to Section 5.2 for additional details on Internet addresses.

`SIOCGIFNETMASK`
Get the interface network mask for an Internet address.

`SIOCGIFMETRIC`
Get the interface routing metric. The routing metric is stored in the kernel for each interface, but is used by the routing protocol, the `routed` daemon.

`SIOCSIFMETRIC`
Set the interface routing metric. The 4.3BSD kernel does not make policy decisions for routing. Instead, this is left to a user process, which then uses this `ioctl` to modify the kernel's routing tables. Refer to Section 11.5 of Leffler et al. [1989] for additional details.

`SIOCSARP` Set an entry in the Internet ARP (Address Resolution Protocol) table. The structure `arpreq` is defined in `<net/if_arp.h>`. Refer to the entry for `arp`(4P) in Section 4 of the BSD manual for additional information.

`SIOCGARP` Get an entry from the Internet ARP entry. The caller specifies an Internet address and this system call returns the corresponding Ethernet address.

`SIOCDARP` Delete an entry from the Internet ARP table. The caller specifies the Internet address for the entry to be deleted.

6.12 Asynchronous I/O

Asynchronous I/O allows the process to tell the kernel to notify it when a specified descriptor is ready for I/O. It is also called signal-driven I/O. The notification from the kernel to the user process takes place with a signal, the `SIGIO` signal.

To do asynchronous I/O, a process must perform the following three steps:

1. The process must establish a handler for the `SIGIO` signal. This is done by calling the `signal` system call, as described in Section 2.4.
2. The process must set the process ID or the process group ID to receive the `SIGIO` signals. This is done with the `fcntl` system call, with the `F_SETOWN` command, as described in Section 6.11.
3. The process must enable asynchronous I/O using the `fcntl` system call, with the `F_SETFL` command and the `FASYNC` argument.

To show an example of asynchronous I/O, let's first show a simple program that copies standard input to standard output.

```
/*
 * Copy standard input to standard output.
 */

#define BUFFSIZE        4096

main()
{
        int             n;
        char            buff[BUFFSIZE];

        while ( (n = read(0, buff, BUFFSIZE)) > 0)
                if (write(1, buff, n) != n)
                        err_sys("write error");

        if (n < 0)
                err_sys("read error");

        exit(0);
}
```

We now change this program to do the three steps listed, using asynchronous I/O.

```
/*
 * Copy standard input to standard output, using asynchronous I/O.
 */

#include        <signal.h>
#include        <fcntl.h>

#define BUFFSIZE        4096
```

```
int     sigflag;

main()
{
        int             n;
        char            buff[BUFFSIZE];
        int             sigio_func();

        signal(SIGIO, sigio_func);

        if (fcntl(0, F_SETOWN, getpid()) < 0)
                err_sys("F_SETOWN error");

        if (fcntl(0, F_SETFL, FASYNC) < 0)
                err_sys("F_SETFL FASYNC error");

        for ( ; ; ) {
                sigblock(sigmask(SIGIO));
                while (sigflag == 0)
                        sigpause(0);            /* wait for a signal */

                /*
                 * We're here if (sigflag != 0).  Also, we know that the
                 * SIGIO signal is currently blocked.
                 */

                if ( (n = read(0, buff, BUFFSIZE)) > 0) {
                        if (write(1, buff, n) != n)
                                err_sys("write error");
                } else if (n < 0)
                        err_sys("read error");
                else if (n == 0)
                        exit(0);                /* EOF */

                sigflag = 0;                    /* turn off our flag */

                sigsetmask(0);                  /* and reenable signals */
        }
}

int
sigio_func()
{
        sigflag = 1;            /* just set flag and return */

        /* the 4.3BSD signal facilities leave this handler enabled
           for any further SIGIO signals. */
}
```

This interrupt driven example works only for terminals (and would work for a socket too, if the appropriate code were added to create a socket) because of the restrictions of the 4.3BSD `F_SETOWN` command.

This example also shows the use of the reliable signals provided by 4.3BSD, which we described in Section 2.4.

Since a process can have only one signal handler for a given signal, if asynchronous I/O is enabled for more than one descriptor, the process doesn't know, when the signal occurs, which descriptor is ready for I/O. To do this, the `select` system call is used, which we describe in the next section.

6.13 Input/Output Multiplexing

Consider a process that reads input from more than one source. An actual example is the 4.3BSD line printer spooler that we cover in Section 13.2. It opens two sockets, a Unix stream socket to receive print requests from processes on the same host, and a TCP socket to receive print requests from processes on other hosts. Since it doesn't know when requests will arrive on the two sockets, it can't start a read on one socket, as it could block while waiting for a request on that socket, and in the mean time a request can arrive on the other socket. There are a few different techniques available to handle this multiplexing of different I/O channels.

1. It can set both sockets to nonblocking, using either the `FNDELAY` flag to `fcntl` or the `FIONBIO` request to `ioctl`. But the process has to execute a loop that reads from each socket, and if nothing is available to read, wait some amount of time before trying the reads again. This is called *polling*. The software polls each socket at some interval (every second, perhaps) and if no data is available to read, the process waits by calling the `sleep` function in the standard C library. Polling can waste computer resources, since most of the time there is no work to be done.

2. The process can `fork` one child process to handle each I/O channel. In this example it spawns two processes, one to read from the Unix socket and one to read from the TCP socket. Each child process calls `read` and blocks until data is available. When a child returns from its `read` it passes the data to the parent process using some form of IPC (that the parent must set up before spawning its children). Any of the techniques that we discussed in Chapter 3 can be used: pipes, FIFOs, message queues, shared memory with a semaphore, or another socket can also be used.

3. Asynchronous I/O can be used. The problem with this method is that signals are expensive to catch [Leffler et al. 1989]. Also, if more than one descriptor has been enabled for asynchronous I/O, the occurrence of the `SIGIO` signal doesn't tell us which descriptor is ready for I/O. Finally, as mentioned in Section 6.12, this technique is only available for terminals and sockets under 4.3BSD.

4. Another technique is provided by 4.3BSD with its `select` system call, which we describe below. This system call allows the user process to instruct the kernel to wait for any one of multiple events to occur and to wake up the process only when one of these events occurs. In this example the kernel can be instructed to notify the process only when data is available to be read from either of the two sockets that we're interested in.

The prototype for the system call is

```
#include <sys/types.h>
#include <sys/time.h>

int select(int maxfdp1, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);

FD_ZERO(fd_set *fdset);                 /* clear all bits in fdset */
FD_SET(int fd, fd_set *fdset);          /* turn the bit for fd on in fdset */
FD_CLR(int fd, fd_set *fdset);          /* turn the bit for fd off in fdset */
FD_ISSET(int fd, fd_set *fdset);        /* test the bit for fd in fdset */
```

The structure pointed to by the *timeout* argument is defined in `<sys/time.h>` as

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

The C `typedef` for the `fd_set` structure, and the definitions of the `FD_xxx` macros are in the `<sys/types.h>` include file. The definition of this system call makes it look more complicated than it is.

A request to `select` could be: tell us if any of the file descriptors in the set {1, 4, 5} are ready for reading, or if any of the file descriptors in the set {2, 7} are ready for writing, or if any of the file descriptors in the set {1, 4} have an exceptional condition pending. Additionally we can tell the kernel to

- Return immediately after checking the descriptors. This is a poll. For this, the *timeout* argument must point to a `timeval` structure, and the timer value (the number of seconds and microseconds specified by the structure) must be zero.
- Return when one of the specified descriptors is ready for I/O, but don't wait beyond a fixed amount of time. For this, the *timeout* argument must point to a `timeval` structure, and its value (the number of seconds and microseconds specified by the structure members) must be the nonzero amount of time to wait.
- Return only when one of the specified descriptors is ready for I/O—wait indefinitely. For this, the *timeout* argument must be NULL. This wait can also be interrupted by a signal.

Before describing how to specify the file descriptors to be checked, let's first show an example where we are not interested in any file descriptors. We'll use `select` as a higher precision timer than is provided by the `sleep` function. (`sleep` provides a resolution of seconds, whereas `select` allows us to specify a resolution in microseconds.)

```
#include        <sys/types.h>
#include        <sys/time.h>

main(argc, argv)
int     argc;
char    *argv[];
{
        long                            atol();
        static struct timeval   timeout;

        if (argc != 3)
                err_quit("usage: timer <#seconds> <#microseconds>");
        timeout.tv_sec  = atol(argv[1]);
        timeout.tv_usec = atol(argv[2]);

        if (select(0, (fd_set *) 0, (fd_set *) 0, (fd_set *) 0, &timeout) < 0)
                err_sys("select error");
        exit(0);
}
```

Here we have specified the second, third and fourth arguments to `select` as NULL.

The *readfds*, *writefds*, and *exceptfds* arguments specify which file descriptors we're interested in checking for each of the conditions—descriptor ready for reading, descriptor ready for writing and exceptional condition on descriptor, respectively. There are only two exceptional conditions currently supported.

1. The arrival of out-of-band data for a socket. We describe this in more detail in the next section.
2. The presence of control status information to be read from the master side of a pseudo-terminal that has been put into packet mode. We cover this in Section 15.10.

The problem the designers of this system call had was how to specify one or more descriptor values for each of these three arguments. The decision was made to represent the set of all possible descriptors using an array of integers, where each bit corresponds to a descriptor. For example, using 32-bit integers, the first element of the array corresponds to descriptors 0 through 31, the second element of the array corresponds to descriptors 32 through 63, and so on. All this implementation detail is hidden through the `typedef` of the `fd_set` structure and the `FD_xxx` macros.

For example, to define a variable of type `fd_set` and then turn on the indicators for descriptors 1, 4, and 5

```
fd_set  fdvar;

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

It is important to initialize the set, since unpredictable results can occur if the set is allocated as an automatic variable and not initialized. Any of the three arguments to `select`, *readfds*, *writefds*, or *exceptfds*, can be specified as NULL pointers, if we're not interested in that condition.

The *maxfdp1* argument specifies the number of file descriptors to be tested. Its value is the maximum file descriptor to be tested, plus one. The descriptors 0, 1, 2, up through and including *maxfdp1*−*1* are tested. The system allows for a potentially huge number of descriptors to be tested (currently 256), although most BSD kernels don't allow any single process to have that many open files. The *maxfdp1* argument lets us tell the system the largest descriptor that we're interested in, which is usually much less than 256. For example, given the example code above that turns on the indicators for descriptors 1, 4, and 5, a *maxfdp1* value of 6 can be specified. The reason it is 6 and not 5 is that we're specifying the number of descriptors, not the largest value, and descriptors start at zero.

This system call modifies the three arguments *readfds*, *writefds*, and *exceptfds* to indicate which descriptors are ready for the specified condition. These three arguments are value–result arguments. The caller should use the `FD_ISSET` macro to test a specific bit in an `fd_set` structure. The return value from this system call indicates the total number of descriptors that are ready. If the timer value expired before any of the descriptors were ready, a value of zero is returned. As usual, a return value of −1 indicates an error.

We have been talking about waiting for a file descriptor to become ready for I/O (reading or writing) or to have an exceptional condition pending on it. For sockets, this assumes that the process wants to do I/O on the socket. The `select` system call can also be used to await a connection on a socket. For example, if a process is awaiting a `connect` on more than one socket, if it doesn't want to execute the `accept` system call and potentially block, it can issue a `select` instead. If a `connect` is received by the system for a socket that is being waited for, the socket is then considered ready for reading, and the `select` returns.

6.14 Out-of-Band Data

We described the notion of out-of-band data in Chapter 4, and also mentioned it in Chapter 5 when we discussed the TCP and SPP protocols. Of all the protocols we've discussed so far, only TCP and SPP support out-of-band data. (Out-of-band data is only defined for stream sockets, and the Unix domain stream sockets don't support it.) Unfortunately, TCP and SPP treat out-of-band data differently. The designers of the socket interface tried to support out-of-band data in a generic way, but there are many programs that have been written using the TCP out-of-band data features that would be hard to port to SPP. The BSD socket abstraction calls for the protocol to support at least one outstanding out-of-band message at any time, and the out-of-band message must contain at least one byte of data.

To send an out-of-band message, the `MSG_OOB` flag must be specified for the `send`, `sendto`, or `sendmsg` system calls. SPP only allows a single byte of out-of-band data to be sent, while TCP has no restriction.

SPP Out-of-Band Data

When out-of-band data is received by the other end, several possibilities exist about how it is handled. Let's first consider how SPP handles it, as it is less complicated than the TCP implementation.

- The out-of-band byte is received along with a special flag specifying that the byte contains out-of-band data. (This flag is the Attention bit in the SPP header.)
- If the socket has a process group, the `SIGURG` signal is generated for the process group of the socket.
- The socket option `SO_OOBINLINE` has no effect, since the Xerox protocol specifies that the out-of-band byte is to be delivered again to the receiving process, in its normal position in the data stream.
- The process can read the out-of-band byte by calling any one of the three receive system calls, specifying the `MSG_OOB` flag. Only the single out-of-band byte is returned. If there is no out-of-band data when the `MSG_OOB` flag is specified, an error of `EINVAL` is returned by these three system calls instead.
- Regardless whether the process reads the out-of-band byte or not, the position of the byte in the normal data stream is remembered.
- The process continues reading data from the socket. Since the system remembers the position of the out-of-band data byte, it does not read right past it in a single read request. That is, if there are 20 bytes from the current read position until the out-of-band byte, and if we execute a read or receive system call specifying a length of 30 bytes, the system only returns 20 bytes. This forced stopping at the

out-of-band mark is to allow us to execute the SIOCATMARK ioctl to determine when we're at the mark. This allows us to ignore all the data up to the out-of-band byte. When this ioctl indicates that we're at the mark, the next byte we read is the out-of-band byte.

TCP Out-of-Band Data

All this gets more complicated with TCP because it can send the notification that out-of-band data exists (termed "urgent data" by TCP) *before* it sends the out-of-band data. In this case, if the process executes one of the three receive system calls, an error of EWOULDBLOCK is returned if the out-of-band data has not arrived. As with the SPP example above, an error of EINVAL is returned by these three system calls if the MSG_OOB flag is specified when there isn't any out-of-band data.

Still another option is provided with the TCP implementation, to allow for multiple bytes of out-of-band data. By default, only a single byte of out-of-band data is provided, and unlike SPP, this byte of data is not stored in the normal data stream. This data byte can only be read by specifying the MSG_OOB flag to the receive system calls. But if we set the SO_OOBINLINE option for the socket, using the setsockopt system call described in Section 6.11, the out-of-band data is left in the normal data stream and is read without specifying the MSG_OOB flag. If this socket option is enabled, we must use the SIOCATMARK ioctl to determine where in the data stream the out-of-band data occurs. In this case, if multiple occurrences of out-of-band data are received, all the data is left in-line (i.e., none of the data can be lost) but the mark returned by the SIOCATMARK ioctl corresponds to the final sequence of out-of-band data that was received. If the out-of-band data is not received in-line, it is possible to lose some intermediate out-of-band data when the out-of-band data arrives faster than it is processed by the user.

The remote login application in Chapter 15 uses TCP out-of-band data along with the SIOCATMARK ioctl.

6.15 Sockets and Signals

We have mentioned signals in relation to sockets a few times in this chapter, and its worth taking a moment to summarize the conditions under which signals are generated for a socket and all the related nuances. There are three signals that can be generated for a socket

SIGIO — This signal indicates that a socket is ready for asynchronous I/O. The signal is sent to the process group of the signal. This process group is established by calling ioctl with a command of either FIOSETOWN or SIOCSPGRP, or by calling fcntl with a command of F_SETOWN. This

signal is sent to the process group only if the process has enabled asynchronous I/O on the socket by calling `ioctl` with a command of `FIOASYNC` or by calling `fcntl` with a command of `F_SETFL` and an argument of `FASYNC`.

`SIGURG` This signal indicates that an urgent condition is present on a socket. An urgent condition is either the arrival of out-of-band data on the socket or the presence of control status information to be read from the master side of a pseudo-terminal that has been put into packet mode. (We discuss pseudo-terminal packet mode in Section 15.10.) The signal is sent to the process group of the signal. This process group is established by calling `ioctl` with a command of either `FIOSETOWN` or `SIOCSPGRP`, or by calling `fcntl` with a command of `F_SETOWN`.

`SIGPIPE` This signal indicates that we can no longer write to a socket, pipe, or FIFO. Nothing special is required by a process to receive this signal, but unless the process arranges to catch the signal, the default action is to terminate the process. This signal is sent only to the process associated with the socket; the process group of the socket is not used for this signal.

We mentioned in Section 2.4 that certain system calls (termed the "slow" system calls) can be interrupted when the process handles a signal. This always has to be considered when handling signals. Although 4.3BSD tries automatically to restart certain system calls that are interrupted, the `accept` and `recvfrom` system calls are never restarted automatically by the kernel. Since both of these system calls can typically block, if you are using them and handling signals, be prepared to restart the system call if they're interrupted.

6.16 Internet Superserver

On a typical 4.3BSD system there can be many daemons in existence, just waiting for a request to arrive. For example, there could be an Internet FTP daemon, an Internet TELNET daemon, and Internet TFTP daemon, a remote login daemon, a remote shell daemon, and so on. With systems before 4.3BSD, each of these services had a process associated with it. This process was started at boot time from the `/etc/rc` startup file, and each process did nearly identical startup tasks: create a socket, `bind` the server's well-known address to the socket, wait for a connection, then `fork`. The child process performed the service while the parent waited for another request.

The 4.3BSD release simplified this by providing an Internet superserver: the `inetd` process. This daemon can be used by a server that uses either TCP or UDP. It does not handle either the XNS or Unix domain protocols. What this daemon provides is two features:

- It allows a single process (`inetd`) to be waiting to service multiple connection requests, instead of one process for each potential service. This reduces the total number of processes in the system.
- It simplifies the writing of the daemon processes to handle the requests, since many of the start-up details are handled by `inetd`. This obviates the need for the actual service process to call the `daemon_start` function that we developed in Section 2.6. All the details that we said had to be handled by a typical daemon, are done by `inetd`, before the actual server is invoked.

There is a price to pay for this, however, in that the `inetd` daemon has to execute both a `fork` and an `exec` to invoke the actual server process, while a self-contained daemon that did everything itself only has to execute a `fork` to handle each request. This additional overhead, however, is worth the simplification of the actual servers and the reduction in the total number of processes in the system.

The `inetd` process establishes itself as a daemon using many of the techniques that we described in Section 2.6. It then reads the file `/etc/inetd.conf` to initialize itself. This text file specifies the services that the superserver is to listen for, and what to do when a service request arrives. Each line contains the fields shown in Figure 6.17.

Field	Description
service-name	must be in `/etc/services`
socket-type	`stream` or `dgram`
protocol	must be in `/etc/protocols`: either `tcp` or `udp`
wait-flag	`wait` or `nowait`
login-name	from `/etc/passwd`: typically `root`
server-program	full pathname to `exec`
server-program-arguments	maximum of 5 arguments

Figure 6.17 Fields in `inetd` configuration file.

Some sample lines are

```
ftp     stream  tcp     nowait  root    /etc/ftpd       ftpd
telnet  stream  tcp     nowait  root    /etc/telnetd    telnetd
login   stream  tcp     nowait  root    /etc/rlogind    rlogind
tftp    dgram   udp     wait    nobody  /etc/tftpd      tftpd
```

The actual name of the server is always passed as the first argument to a program when it is `exec`ed.

A picture of what the daemon does is shown in Figure 6.18. Let's go through a typical scenario for this daemon.

1. On startup it reads the `/etc/inetd.conf` file and creates a socket of the appropriate type (stream or datagram) for all the services specified in the file. Realize that

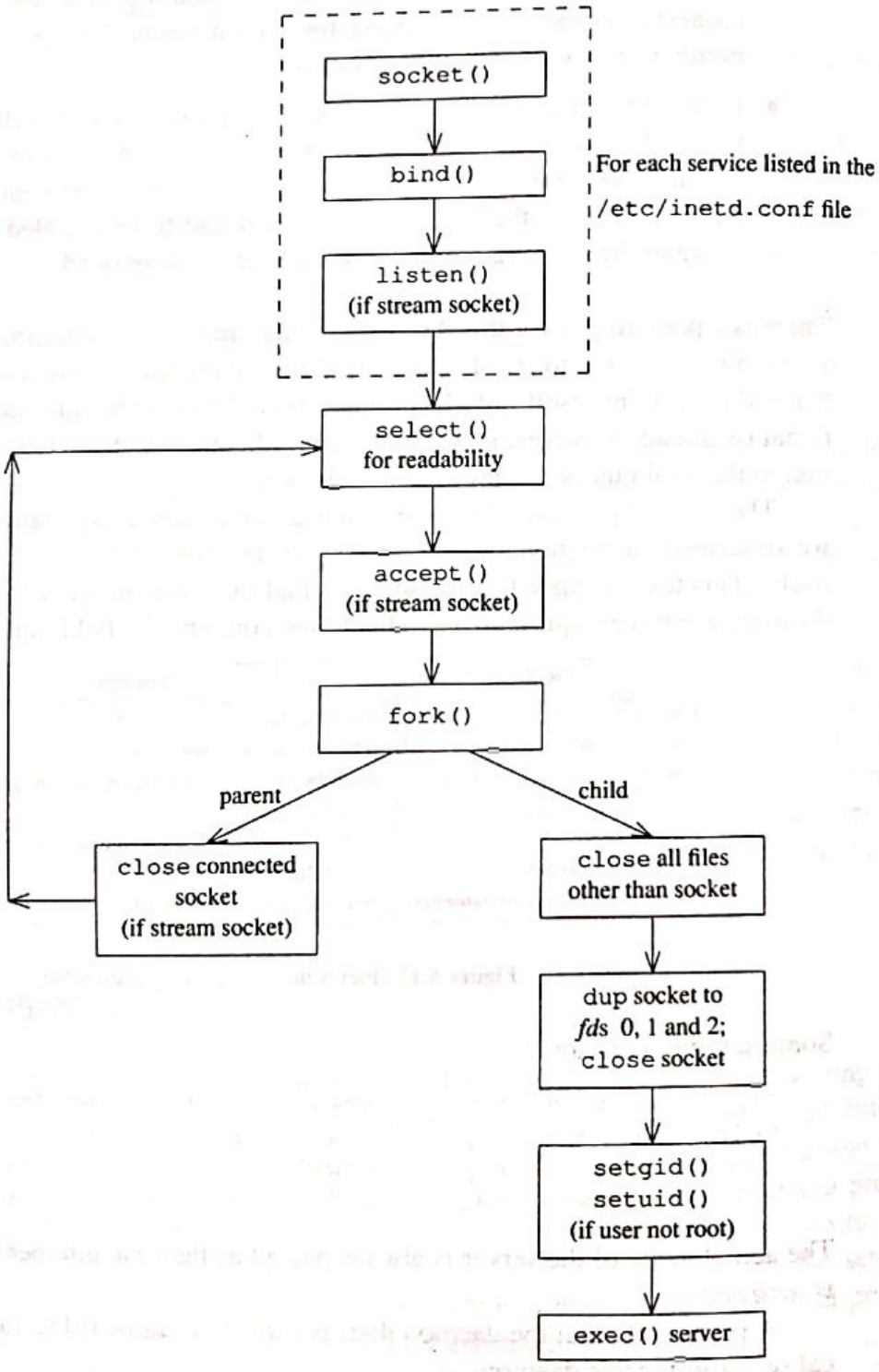

Figure 6.18 Steps performed by `inetd`.

there is a limit to the number of servers that `inetd` can be waiting for, as the total number of file descriptors used by `inetd` can't exceed the system's per-process limit. On most BSD systems this limit is 64, but it can be changed by reconfiguring the kernel.

2. As each socket is created, a `bind` is executed for every socket, specifying the well-known address for the server. This TCP or UDP port number is obtained by looking up the *service-name* field from the configuration file in the `/etc/services` file. Some typical lines in this file, corresponding to the example lines in the configuration file that we showed above, are

```
ftp         21/tcp
telnet      23/tcp
tftp        69/udp
login       513/tcp
```

   Both the *service-name* (such as `telnet`) and the *protocol* from the `inetd` configuration file are passed as arguments to the library function `getservbyname`, to locate the correct port number for the `bind`. (We describe the `getservbyname` function in Section 8.2.)

3. For stream sockets, a `listen` is executed, specifying a willingness to receive connections on the socket and the queue length for incoming connections. This step is not done for datagram sockets.

4. A `select` is then executed, to wait for the first socket to become ready for reading. Recall from our description of this system call in Section 6.13, that a stream socket is considered ready for reading when a connection request arrives for that socket. A datagram socket is ready for reading when a datagram arrives. At this point the `inetd` daemon just waits for the `select` system call to return.

5. When a socket is ready for reading, if it is a stream socket, an `accept` system call is executed to accept the connection.

6. The `inetd` daemon `forks` and the child process handles the service request. The child closes all the file descriptors other than the socket descriptor that it is handling and then calls `dup2` to cause the socket to be duplicated on file descriptors 0, 1, and 2. The original socket descriptor is then closed. Doing this, the only file descriptors that are open in the child are 0, 1, and 2. It then calls `getpwnam` to get the password file entry for the *login-name* that is specified in the `/etc/inetd.conf` file. If this entry does not have a user ID of zero (the superuser) then the child becomes the specified user by executing the `setgid` and `setuid` system calls. (Since the `inetd` process is executing with a user ID of zero, the child process inherits this user ID across the `fork`, so it is able to become any user that it chooses.) The child process now does an `exec` to execute the appropriate *server-program* to handle the request, passing the arguments that are specified in the configuration file.

7. If the socket is a stream socket, the parent process must close the connected socket. The parent goes back and executes the `select` system call again, waiting for the next socket to become ready for reading.

The scenario we've described above handles the case where the configuration file specifies `nowait` for the server. This is typical for all TCP services. If another connection request arrives for the same server, it is returned to the parent process as soon as it executes the `select`. The steps listed above are executed again, and another child process handles this new service request.

Specifying the `wait` flag for a datagram service changes the steps done by the parent process. First, after the `fork` the parent saves the process ID of the child, so it can tell later when that specific child process terminates, by looking at the value returned by the `wait` system call. Second, the parent disables the current socket from future `select`s by using the `FD_CLR` macro to turn off the bit in the `fd_set` structure that it uses for the `select`. This means that the child process takes over the socket until it terminates. Once the child process terminates, the parent process is notified by a `SIGCLD` signal, and the parent's signal handler obtains the process ID of the terminating child and reenables the `select` for the corresponding socket by using the `FD_SET` macro. When the child process terminates, the parent is probably waiting for its `select` system call to return. As mentioned in Section 2.4, when a signal handler is invoked while a process is executing a "slow" system call, when the signal handler returns, the system call (the `select` in this case) returns with an error indication and an `errno` value of `EINTR`. The `inetd` process recognizes this and executes the `select` system call again. But by executing the system call again, it can now wait for the socket corresponding to the child process that terminated. If the occurrence of the signal did not interrupt the `select` system call, the parent would not be able to wait for the socket corresponding to the terminating child process. When we go through the TFTP example in Chapter 12 we'll see how a datagram server uses the `wait` mode.

It is worth going through a time sequence of the steps involved in the previous paragraph, making certain we understand the interaction of the parent and child processes, the `SIGCLD` signal, and how the `select` system call can be interrupted.

- The datagram request arrives on socket *N*, the `select` returns to the `inetd` process.
- A child process is `fork`ed and `exec`ed to handle the request.
- `inetd` disables socket descriptor *N* from its `fd_set` structure for the `select`. The child process takes over the socket.
- The child handles this request and `inetd` handles requests for other services.
- Eventually `inetd` calls `select` and blocks.
- The child terminates, the `SIGCLD` signal is generated for the `inetd` process.

- `inetd` handles the signal and obtains the process ID of the terminating child process from the `wait` system call. It figures out which socket descriptor corresponded to this child process and turns on the appropriate bit in its `fd_set` structure.
- When the signal handler returns, the `select` returns to the `inetd` process, with an `errno` of `EINTR`.
- `inetd` calls `select` again, this time with an `fd_set` structure that enables socket descriptor *N*.

While the `inetd` daemon is set up to handle both a TCP socket with the `wait` flag and a UDP socket with the `nowait` flag, there are no examples of this in the distributed 4.3BSD system.

The `inetd` daemon has several servers that are handled by the daemon itself. These internal servers handle some of the standard Internet services.

- Echo service—RFC 862 [Postel 1983a]
- Discard service—RFC 863 [Postel 1983b]
- Character generator service—RFC 864 [Postel 1983c]
- Daytime (human readable) service—RFC 867 [Postel 1983d]
- Machine time (binary) service—RFC 868 [Postel and Harrenstein 1983]

Each of these services can be contacted using either TCP or UDP. The TCP servers for these internal functions are handled as iterative servers if the amount of time to handle the request is fixed (the daytime and machine time servers), or as concurrent servers if the amount of time depends on the request (echo, discard and character generator).

Since `inetd` is the process that does the `accept` of a stream connection, the server that is invoked by `inetd` has to execute the `getpeername` system call to obtain the address of the client. This is done, for example, by servers that want to verify that the client is using a reserved port. For datagram servers, the address of the client is returned to the server when it executes one of the receive system calls.

6.17 Socket Implementation

Sockets, as implemented by 4.3BSD, are implemented within the Unix kernel. All the system calls that we discussed in this chapter are entry points into the kernel. All the algorithms and code to support the communication protocols (TCP, UDP, IDP, etc.) are completely within the kernel. Adding a new protocol (such as the OSI-related standards) requires changing the kernel. The overall networking design gets more modular with each release (the addition of XNS support in 4.3BSD required changes, as documented in O'Toole, Torek, and Weiser [1985], and the addition of OSI protocols in 4.4BSD will require additional changes that weren't anticipated in earlier releases), but it still requires kernel changes.

The actual kernel implementation separates the network system into three layers, as shown in Figure 6.19. See Leffler et al. [1989] for additional details.

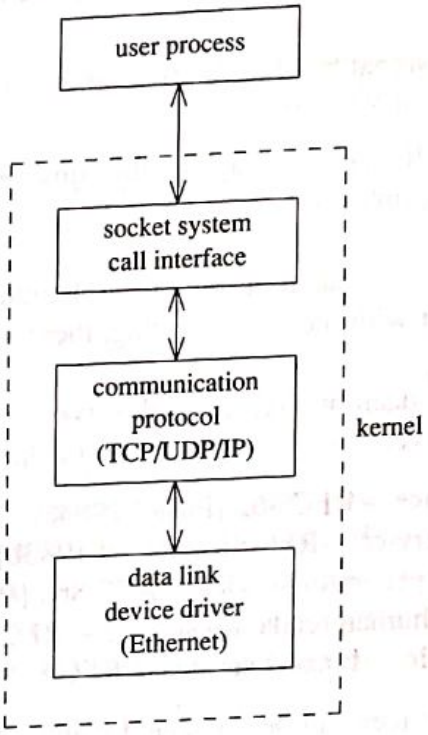

Figure 6.19 Networking implementation in 4.3BSD.

6.18 Summary

This has been a long chapter with many details. Our presentation was divided into four parts.

- Elementary socket system calls
- Examples using only the elementary socket system calls
- Advanced socket system calls
- Advanced socket features and options

One key point is that the simple examples presented in Section 6.6 can form the basis for any networking application. Just modify the client and server to do whatever you desire.

We'll present additional features for these examples in Chapter 8. For example, in Section 8.2 we show a better way of obtaining the address of a remote system (instead of hard coding it in a header file). Section 8.3 develops some functions that simplify the standard operations that most client applications execute when opening a socket. Section 8.4 shows how to add reliability to a datagram application.