Berkeley UNIX† System Calls and Interprocess Communication

by Lawrence Besaw

January, 1987
Revised, September 1987, January 1991 by Marvin Solomon

The purpose of this paper is to discuss interprocess communication in the context of Berkeley UNIX. Special emphasis will be given to those system calls concerned with the creation, management, and use of sockets. There will also be a discussion of signals and selected other system calls that will be useful to those working on the network project assigned for this course. More information on all the system calls mentioned below can be found in the *UNIX Programmer's Manual*. Periodic mention will be made of other manual page entries that might be consulted. System header files are designated by enclosing angle brackets; they reside in /usr/include and its subdirectories.

1. Socket Creation

The most general mechanism for interprocess communication offered by Berkeley UNIX is the socket. A socket is an endpoint for communication. Two processes can communicate by creating sockets and sending messages between them. There are a variety of different types of sockets, differing in the way the address space of the sockets is defined and the kind of communication that is allowed between sockets. A socket type is uniquely determined by a <domain, type, protocol> triple. In order for a remote socket to be reached, it must be possible to assign a name to it. The form that this name assumes is determined by the *communication domain* or *address family* to which the socket belongs. There is also an *abstract type* or *style of communication* associated with each socket. This dictates the semantics of communication for that socket. Finally, there is a specific *protocol* that is used with the socket. A socket can be created with the *socket* system call by specifying the desired address family, socket type, and protocol.

```
socket_descriptor = socket(domain, type, protocol)
int socket_descriptor, domain, type, protocol;
```

This call returns a small positive integer called a *socket descriptor* that can be used as a parameter to reference the socket in subsequent system calls. Socket descriptors are similar to file descriptors returned by the *open* system call. Each *open* or *socket* call will return the smallest unused integer. Thus a given number denotes either an open file, a socket, or neither (but never both). Socket and file descriptors may be used interchangeably in many system calls. For example, the *close* system call is used to destroy sockets.

1.1. Domains

The communication domain or address family to which a socket belongs specifies a certain address format. All later operations on a socket will interpret the supplied address according to this specified format. The various address formats are defined as manifest constants in the file <sys/socket.h>.[1] Examples are AF_UNIX (UNIX path names), AF_INET (DARPA Internet addresses), and AF_OSI (as specified by the international standards for Open Systems Interconnection). AF_UNIX and AF_INET are the most important address families. The general form of an address is represented by the *sockaddr* structure defined in <sys/socket.h>.

```
struct sockaddr {
     short sa_family;    /* address family */
     char  sa_data[14];  /* up to 14 bytes of direct address */
}
```

†UNIX is a trademark of AT&T Bell Laboratories.

When a socket is created, it does not initially have an address associated with it. However, since it is impossible for a remote host or process to find a socket unless it has an address, it is important to bind an address to the socket. A socket does not have a name until an address is explicitly assigned to it with the *bind* system call.

```
status = bind(sock, address, addrlen)
int status;     /* status returns 0 for success, –1 otherwise */
int sock;    /* descriptor returned by socket() */
struct sockaddr *address;
int addrlen; /* size of address (in bytes) */
```

This call fails if the address is already in use, the address is not in the correct format for the address family specified, or the socket already has an address bound to it.

1.1.1. UNIX domain

In the UNIX domain, a socket is addressed by a UNIX path name that may be up to 108 characters long. The binding of a path name to a socket results in the allocation of an inode and an entry of the path name into the file system. This necessitates removing the path name from the file system (using the *unlink* system call) when the socket is closed. The created file is only used to provide a name for the socket and does not play a role in the actual transfer of data. When using sockets in the UNIX domain, it is advisable to only use path names for directories (such as /tmp) directly mounted on the local disk. The UNIX domain only allows interprocess communication for processes working on the same machine. The structure *sockaddr_un* used to define the UNIX address format can be found in <sys/un.h>.

```
struct sockaddr_un {
     short   sun_family;     /* AF_UNIX */
     char    sun_path[108-4];  /* path name */
}
```

NB: When specifying the length of UNIX domain addresses for system calls, use `sizeof(struct sockaddr_un)`. Using the size of *sockaddr* will cause the call to fail.

1.1.2. Internet domain

In the DARPA Internet domain, addresses consist of two parts—a host address (consisting of a network number and a host number) and a port number (also known as a *transport suffix*). This host address allows processes on different machines to communicate. The port number in turn is like a mail box that allows multiple addresses on the same host. The structure *sockaddr_in* describing an address in the Internet domain is defined in the file <netinet/in.h>.

```
struct sockaddr_in {
     short     sin_family;       /* AF_INET */
     u_short   sin_port;         /* port number */
     struct    in_addr sin_addr; /* see struct below */
     char      sin_zero[8];      /* not used */
}

struct in_addr {
      union {
            struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
            struct { u_short s_w1,s_w2; } S_un_w;
            u_long S_addr;
      } S_un;
#define     s_addr      S_un.S_addr /* can be used for most tcp & ip code */
}
```

[1] You must include <sys/types.h> before <sys/socket.h>.

It is often useful to bind a specific service to a “well-known” port, enabling remote processes to locate the required server easily. Examples of well-known ports are 79 for the “finger” service and 513 for remote login. The kernel reserves the first 1024 port numbers for its own use. There is a network services database in /etc/services that can be queried by using the system calls *getservbyname* and *getservbyport* (see *getservent*(3N) and *services*(5)). Each of these routines returns a pointer to the structure *servent* defined in <netdb.h>. If the port field in the address parameter to *bind* is specified as zero, the system will assign an unused port number. The assigned port number can be discovered by calling *getsockname*. In the Internet domain, UDP and TCP can use the same port numbers. There is no confusion in naming because ports bound to sockets with different protocols cannot communicate. Ports numbered less than 1024 are “reserved”—only processes running as superuser may bind to them.

Internet host addresses are specified in four bytes (32 bits). They are typically represented by a standard ‘**.**’ notation, a.b.c.d. The bytes of the address are represented by decimal integers, separated by periods, and ordered from high order to low order. This ordering is called *network order* and is the order in which addresses are transmitted over the network. For example, the Internet address for the host garfield.cs.wisc.edu is 128.105.1.3 which corresponds to the unsigned integer 80690103 in hex, or 2154365187 in decimal. Some hosts (such as the VAX), however, have a *host order* for these integer values that reverses the order of the bytes. When a word is transmitted from such a host (or when a C program treats a word as a sequence of bytes), the low-order byte is transmitted first (has the lowest address). Thus, it is necessary to reverse the bytes of an address, stored in an integer, before transmitting it. The system routines *htonl* and *ntohl* are provided to convert long (32-bit) integers from host to network order and *vice versa*. Similarly, *htos* and *ntohs* swap the bytes of short (16-bit) integers, such as port numbers. System calls that return or demand Internet addresses and port numbers (such as *gethostbyname* or *bind*, which is described above), deal entirely in network order, so you normally don’t have to worry about all this. However, if you want to print out one of these values or read it in, you will have to convert from/to host order.

An Internet host address can be decomposed into two parts—a network number and a local address. There are three formats for Internet addresses (see *inet*(3N)), each of which partitions the 32 bits differently. “Class A” addresses have a one-byte network number (the high order byte), and a 3-byte host number. “Class B” addresses have two bytes each of network and host number, and “class C” networks have three bytes of network number and one byte of host number. (Thus a class C network can have at most 256 hosts). The high-order bits of an address determine its class: Addresses starting with a zero are class A addresses, addresses starting with 10 are class B, and addresses starting with 110 are class C. Thus there can be at most 128 class A networks and at most 2^{14} = 16,384 class B networks.

The system calls *gethostbyaddr* and *gethostbyname* can be used to look up the host address (see *gethostent*(3N) and *host*(5)). Each of these calls returns a pointer to the structure *hostent* defined in <netdb.h>. Host addresses are returned in network order, the proper format for a call to *bind*. The system routine *memcpy* can be used to copy the host address into the *sockaddr_in* structure. If a zero host address is given (the wildcard value INADDR_ANY can be used) in the call to *bind*, the local host address will be automatically supplied. Another reason for using INADDR_ANY is that a host may have multiple Internet addresses; addresses specified this way will match any incoming messages with a valid Internet address.

1.2. Styles of Communication

The type field of the *socket* call specifies the abstract “style of communication” to be used with that socket. These types are defined as manifest constants in <sys/socket.h>. The following types are currently defined: SOCK_STREAM (stream), SOCK_DGRAM (datagram), SOCK_RAW (raw protocol interface), SOCK_RDM (reliable datagrams), and SOCK_SEQPACKET (sequenced packet stream). Each of these abstractions is supported by different protocols. We will be primarily concerned with the SOCK_STREAM and SOCK_DGRAM abstractions.

1.2.1. Datagram Sockets

The SOCK_DGRAM type provides a datagram model of communication. A datagram service is connectionless and unreliable. Independent (and usually small) messages called *datagrams* are accepted by the transport protocol and sent to the address specified. These messages may be lost, duplicated, or received out of order. An important characteristic of datagrams is that message boundaries are maintained. This means that individual datagrams (i.e., messages sent with separate calls) will be kept separate when they are received at the other end. A *revcfrom* call on a datagram socket will only return the next datagram available. The SOCK_DGRAM type can be used in the UNIX or Internet domain. When used in the Internet domain, it is supported by the Internet transport protocol

UDP. The call

```
        sock = socket(AF_INET, SOCK_DGRAM, 0)
```

returns a UDP datagram socket.

1.2.2. Stream Sockets

The STREAM_SOCK type provides a stream model of communication. This service is reliable and connection-oriented. Data is transmitted on a full-duplex, flow-controlled byte stream. The transport protocol supporting this socket type must ensure that data is delivered in order without loss, error, or duplication. Otherwise, it must abort the connection and report an error to the user. Message boundaries are *not* preserved on a stream socket. Before data can be sent or received on a stream socket, the socket must be made the active or passive end of a connection using the system calls *listen*, *accept*, and *connect* discussed in Section 2. Again, the SOCK_STREAM abstraction can be used in both the UNIX and Internet domains. In the Internet domain, it is supported by the Internet transport protocol TCP. The call

```
        sock = socket(AF_INET, SOCK_STREAM, 0)
```

returns a TCP stream socket.

1.3. Protocols

A protocol is a set of communication conventions for regulating the exchange of information between two parties. Transport protocols that support the styles of communication described above are implemented as code in the UNIX kernel. These concrete protocols actually realize the semantics defined by the socket type. User Datagram Protocol (UDP), Transmission Control Protocol (TCP), and Internet Protocol (IP) all belong to the Internet family of protocols. Each member of this protocol family supports a different abstract type (see *tcp*(4P), *udp*(4P), and *ip*(4P)). UDP supports datagram sockets, TCP supports stream sockets, and the SOCK_RAW type of socket provides a raw interface to IP. TP, the ISO connection-based transport protocol, conceptually supports the SOCK_SEQPACKET abstraction, although it is not currently implemented in Berkeley Unix,

There is currently only one protocol supporting a given socket abstraction in a particular domain. Nothing, however, prevents future protocols from being implemented that support the same abstract type in the same domain as an existing protocol. The protocol to be used is specified in the protocol field of the *socket* call. There are three ways to specify the protocol. First, if a zero is supplied, the system provides the default protocol for that address family and socket type; this is the most common method. Second, manifest constants describing protocols, such as IPPROTO_UDP, can be found in <netinet/in.h>. Third, the network protocol database /etc/protocols can be consulted by calling *getprotobyname* and *getprotobynumber* (see *getprotoent*(3N) and *protocols*(5)).

2. Connection Establishment

Stream sockets must establish a connection before data can be transferred. A stream socket is either “active” or “passive”. A socket is initially active and only becomes passive after a *listen* call. Only active sockets can be mentioned in a *connect* call and only passive sockets can be mentioned in *accept*. These connection establishment calls are intended for use with stream sockets, but datagram sockets can call *connect* to permanently fix the destination for future *send* calls.

2.1. Active End

An active socket establishes a connection to a named passive socket by using the *connect* system call.

```
        status = connect(sock, name, namelen)
        int sock; /* socket descriptor */
        struct sockaddr *name;
        int namelen;
```

The *name* parameter is the address of the remote socket interpreted according to the communication domain in which the socket exists. If the domain is AF_UNIX, this should be a *sockaddr_un* structure; if the domain is AF_INET, it should be a *sockaddr_in* structure.

2.2. Passive End

A socket becomes the passive end of a connection by first doing a *listen* and then an *accept*.

status = listen(sock, queuelen) int sock, queuelen;

Listen initializes a queue for waiting connection requests. The parameter *queuelen* specifies the maximum number of queued connections that will be allowed. The maximum queue length is currently limited by the system to 5. The manifest constant SOMAXCONN from <sys/socket.h> defines this maximum. A connection can then be established using the system call *accept*.

new_socket = accept(old_sock, name, namelen) int new_socket, old_sock; /* socket descriptors */ struct sockaddr *name; /* name of peer socket on new connection */ int *namelen; /* length of name in bytes */

Accept takes the first pending connection request off the queue and returns a socket descriptor for a new, connected socket. This socket has the same properties as the old socket. The address of the socket at the active end is returned in *name*. *Namelen* is a value/result parameter that should be initialized to the size of the address structure being passed; upon return it will be set to the actual length of the address returned. The old socket remains unaffected and can be used to accept more connections. If there are no pending connection requests, *accept* blocks. If necessary, a *select* can be done first to see if there are any connection requests (see Section 4.2). A socket with pending connections will show up as being ready for reading.

3. Data Transfer

The system call pairs (*read*, *write*), (*recv*, *send*), (*recvfrom*, *sendto*), (*recvmsg*, *sendmsg*), and (*readv*, *writev*) can all be used to transfer data on sockets. The most appropriate call depends on the exact functionality required. *Send* and *recv* are typically used with connected stream sockets. They can also be used with datagram sockets if the sender has previously done a *connect* or the receiver does not care who the sender is. *Sendto* and *recvfrom* are used with datagram sockets. *Sendto* allows one to specify the destination of the datagram, while *revcfrom* returns the name of the remote socket sending the message. *Read* and *write* can be used with any connected socket. These two calls may be chosen for efficiency considerations. The remaining data transfer calls can be used for more specialized purposes. *Writev* and *readv* make it possible to scatter and gather data to/from separate buffers. *Sendmsg* and *recvmsg* allow scatter/gather capability as well as the ability to exchange access rights. The calls *read*, *write*, *readv*, and *writev* take either a socket descriptor or a file descriptor as their first argument; all the rest of the calls require a socket descriptor.

```
count = send(sock, buf, buflen, flags)
int count, sock, buflen, flags;
char *buf;

count = recv(sock, buf, buflen, flags)
int count, sock, buflen, flags;
char *buf;

count = sendto(sock, buf, buflen, flags, to, tolen)
int count, sock, buflen, flags, tolen;
char *buf;
struct sockaddr *to;

count = recvfrom(sock, buf, buflen, flags, from, fromlen)
int count, sock, buflen, flags, *fromlen;
char *buf;
struct sockaddr *from;
```

For the send calls, *count* returns the number of bytes accepted by the transport layer, or –1 if some error is detected locally. A positive return count is no indication of the success of the data transfer. NB: If made non-blocking, *send* may accept some but not all of the bytes in the data buffer (see Section 4.1). The return value should be checked so that the remaining bytes can be sent if necessary. For receive calls, *count* returns the number of bytes actually received, or –1 if some error is detected.

The first parameter for each call is a valid socket descriptor. The parameter *buf* is a pointer to the caller's data buffer. In the send calls, *buflen* is the number of bytes being sent; in the receive calls, it indicates the size of the data area and the maximum number of bytes the caller is willing to receive. The parameter *to* in the *sendto* call specifies the destination address (conforming to a particular address family) and *tolen* specifies its length. The parameter *from* in the *recvfrom* call specifies the source address of the message. *Fromlen* is a value/result parameter that initially gives the size of the structure pointed to by *from* and then is modified on return to indicate the actual length of the address.

The *flags* parameter, which is usually given zero as an argument, allows several special operations on stream sockets. It is possible to send *out-of-band* data or "peek" at the incoming message without actually reading it. The flags MSG_OOB and MSG_PEEK are defined in <sys/sockets.h>. Out-of-band data is high priority data (such as an interrupt character) that a user might want to process quickly before all the intervening data on the stream. If out-of-band data were present, a SIGURG signal could be delivered to the user. The actual semantics of out-of-band data is determined by the relevant protocol. ISO protocols treat it as expedited data, while Internet protocols treat it as urgent data.

If any of these (as well as other) system calls is interrupted by a signal, such as SIGALRM or SIGIO, the call will return −1 and the variable errno will be set to EINTR.[2] The system call will be automatically restarted. It may be advisable to reset errno to zero.

4. Synchronization

By default, all the reading and writing calls are blocking: Read calls do not return until at least one byte of data is available for reading and write calls block until there is enough buffer space to accept some or all of the data being sent. Some applications need to service several network connections simultaneously, performing operations on connections as they become enabled. There are three techniques available to support such applications: non-blocking sockets, asynchronous notifications, and the *select* system call. The *select* system call is by far the most commonly used; non-blocking sockets are less common, and asynchronous notifications are rarely used.

4.1. Socket Options

The system calls *getsockopt* and *setsockopt* can be used to set and inspect special options associated with sockets. These might be general options for all sockets or implementation-specific options. Sample options taken from <sys/sockets.h> are SO_DEBUG and SO_REUSEADDR (allow the reuse of local addresses).

Fcntl and *ioctl* are system calls that make it possible to control files, sockets, and devices in a variety of ways.

```
status = fcntl(sock, command, argument)
int status, sock, command, argument;

status = ioctl(sock, request, buffer)
int status, sock, request;
char *buffer;
```

The *command* and *argument* parameters for *fcntl* can be supplied with manifest constants found in <fcntl.h>. The manifest constants for *ioctl*'s *request* parameter are located in <sys/ioctl.h>. This parameter also specifies how the buffer argument of the call is to be used.

Either one of these system calls can be used to enable asynchronous notifications on a socket. Whenever data arrives on such a socket a SIGIO signal will be delivered to the process. The process should have already declared a signal handler for this signal (see Section 6.1). This signal handler can then read the data from the socket. Program execution continues at the point of interruption. The calling sequence

```
fcntl(sock, F_SETOWN, getpid());
fcntl(sock, F_SETFL, FASYNC);
```

enables asynchronous i/o on the given socket. The first call is necessary to specify the process to be signalled. The second call sets the descriptor status flags to enable SIGIO.

[2] Error numbers are defined in the file <errno.h>.

The same calling sequence can be used to make a socket non-blocking; the only change is that the flag FNDELAY is used in the second call instead of FASYNC. In this case, if a read or write operation on the socket would normally block, –1 is returned and the external system variable errno is set to EWOULDBLOCK. This error number can be checked and appropriate action taken.

4.2. Multiplexing File Descriptors

The system call *select* makes possible synchronous multiplexing of socket and file descriptors. It can be used to determine when there is data to read or when it is possible to send more data.

```
nfound = select(numdes, readmask, writemask, exceptmask, timeout)
int nfound, numdes;
fd_set *readmask, *writemask, *exceptmask;
struct timeval *timeout;
```

The masks in this call are value/result parameters through which file and socket descriptors are indicated. The possibility of a particular operation—reading, writing, or the presence of an exceptional condition—is investigated by setting the bit for that socket in the corresponding mask. A zero pointer can used if a given condition is of no interest. For example, if writemask is zero, sockets will not be checked for writing.

The type *fd_set* is defined in <sys/types.h> to be a struct containing a single field, which is an array of integers. The array is interpreted as a bitmask, with one bit for each possible file or socket descriptor. (Representing the mask as struct rather than a simple array allows fd_set values to be assigned to each other, without necessitating a call to *memcpy*). Four macros are defined in <sys/types.h> for setting, clearing, and testing individual bits in the mask:

```
FD_SET(n, p)             /* set bit n */
FD_CLR(n, p)             /* clear bit n */
result = FD_ISSET(n, p)  /* test bit n */
FD_ZERO(p)               /* clear all bits */
int n;
fd_set *p;
```

Multiple bits can be set in a given mask, but each must correspond to a valid descriptor.

The parameter *numdes* indicates that bits 0 through *numdes* – 1 should be examined. The manifest constant FD_SETSIZE, defined in <sys/types.h>, indicates the maximum number of descriptors that can be represented by an *fd_set*. Thus setting numdes = FD_SETSIZE will ensure that no descriptor is overlooked. The *timeout* parameter is a pointer to a *timeval* structure defined in <sys/time.h>. It is used to specify the maximum time interval that the call will wait before returning. If *timeout* is zero (a null pointer), then *select* blocks indefinitely. If *timeout* points to a zero *timeval* structure, then the call returns immediately, even if no descriptor is ready. *Select* returns when a condition has been discovered for one or more of the sockets or the specified time interval has elapsed. The return value *nfound* indicates the number of conditions that were satisfied. The masks are modified to indicate those sockets for which the respective conditions hold.

The most common reason for including file descriptors in the masks passed to *select* is to respond to interactive terminal activity. For example,

```
FD_SET(fileno(stdin), readmask)
```

may be used to check whether anything has been typed on the terminal.

5. Connection Dissolution

A connection can be dissolved by simply using the system call *close* to close one of the sockets involved.

```
status = close(sock)
int status, sock;
```

The precise semantics of the connection close are determined by the responsible protocol. It may be a “graceful close” or an abort that loses data in transit.

The *shutdown* system call can be used to selectively close a full-duplex socket.

```
        status = shutdown(sock, how)
        int status, sock, how;
```

The *how* parameter indicates either that data will no longer be sent on the socket (0), that data will no longer be received on the socket to (1), or that the socket should be completely closed (2).

In the case of sockets created in the UNIX domain, the system call *unlink* should be used to remove the path name to which the socket was bound from the file system.

```
        status = unlink(pathname)
        char *pathname;
```

These path names are not automatically removed when the socket is closed.

6. Signals

Berkeley UNIX provides a set of signals that may be delivered to a process for various reasons, such as an interrupt key being typed or a bus error occurring. These signals—examples of which are SIGIO and SIGALRM—are defined in the file <signal.h>. Typically, the default action is that the delivery of the signal causes a process to terminate. This default can be changed so that a signal is caught or ignored. If the signal is caught, a signal handler is declared as the location where control is transferred at the time of interrupt. The arrival of a signal is thus similar to a hardware interrupt. When a signal is delivered to a process, the program state is saved, the delivered signal is blocked from further occurrence, and program control is transferred to the designated handler. If the handler returns normally, the signal is once again enabled and program execution resumes from the point where it was interrupted. If a signal that is currently blocked arrives, it is queued for later delivery.

6.1. Signal Handlers

A signal handler can be declared with either the *signal* or *sigvec* system calls. *Signal* is a simplified version of the more general *sigvec* call.

```
        oldhandler =  signal(sig, handler)
        int sig;
        int (*handler)(), (*oldhandler)();
```

The *sig* parameter is a manifest constant found in <signal.h> describing the signal. *Handler* is the name of the routine that will be called when the signal is delivered. SIG_DFL (default action) and SIG_IGN (ignore) can also be specified as arguments to this parameter. The value returned is the previous handler (if any). The handler routine itself has the form

```
        sighandler(sig, code, scp)
        int sig, code;
        struct sigcontext *scp;
```

In the signal handler for SIGIO, it is advisable to remove all the data possible from the socket before exiting the routine. This is eliminates the possibility of accidentally losing data because of a missed signal. If more than one event for a given signal occurs while that signal is blocked, only one signal is saved to deliver to the process.

6.2. Blocking Signals

There is a global mask that specifies which signals are currently blocked. The system call *sigblock* can be used to block signals, while the system calls *sigsetmask* and *sigpause* can be used to unblock signals by restoring the original mask. In the file <signal.h>, there is a macro *sigmask* that makes it convenient to set the signal mask for the call to *sigblock*. (It is defined as

```
        #define sigmask(m) (1 << ((m)-1))
        oldmask = sigblock(mask)
        int oldmask, mask;

        oldmask = sigsetblock(mask)
        int oldmask, mask;

        result = sigpause(mask)
        int result, mask; /* call always returns EINTR */
```

For *sigblock*, *mask* specifies those signals to be blocked. *Sigsetmask* and *sigpause* set *mask* as the current signal mask. The call *sigsetmask* returns immediately, while *sigpause* waits for a signal to arrive. For critical sections where it is necessary to block a signal such as SIGIO, the following sequence can be used.

```
        newmask = sigmask(SIGIO);
        oldmask = sigblock(newmask);
         ...
         ...
        sigsetmask(oldmask);
```

7. Timers

There are two system calls that can be used to deliver a SIGALRM signal to a calling process, *alarm* and *setitimer*. *Alarm*, which causes a single SIGALRM to be sent to a process when the specified time expires, provides no resolution finer than a second. *Setitimer*, on the other hand, provides periodic clock interrupts (via SIGALRM) at regular intervals and has a time resolution as small as 10 milliseconds. This is ideal for updating program timers.

```
        status = setitimer(which, value, oldvalue)
        int status, which;
        struct itimerval *value, *oldvalue;
```

It is only necessary to call *setitimer* once and it will continue to send SIGALRM signals at regular intervals. There should be a handler declared to catch this signal, otherwise it will terminate the process. The *which* parameter specifies which interval timer to use. Two possibilities defined in <sys/time.h> are ITIMER_REAL and ITIMER_VIRTUAL. ITIMER_REAL causes the timer to decrement in real time, while an ITIMER_VIRTUAL timer only runs while the process is active. The other two parameters are pointers to a structure defined in <sys/time.h>.

```
        struct itimerval {
             struct timeval it_interval;  /* timer interval */
             struct timeval it_value;     /* current value */
        }

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

Setitimer sets the specified interval timer to the time values specified by *value* and returns the previous value in *oldvalue*. A SIGALRM signal (SIGVTALRM for ITIMER_VIRTUAL) will be sent to the process when the time value specified by *value->it_value* becomes zero. The interval timer will then be loaded with the time value designated by *value->it_interval*, and so on repeatedly. Thus, in order to provide a periodic clock interrupt, it is only necessary to call *setitimer* once with *value->it_interval* and *value->it_value* initialized to the same desired time value. If *it_interval* is zero, the timer will only fire once. A zero value for *it_value* turns off the timer. There are three macros defined in <sys/time.h> that are quite useful in manipulating *timeval* structures. They are *timerisset*, *timerclear*, and *timercmp*.

8. Example Programs

The following two programs demonstrate the use of TCP STREAM sockets. The first program, **client.c**, establishes a connection to a port and host specified as command-line arguments and then sits in a loop sending everything typed at the terminal to the connection and prints everything that comes back from the connection. It terminates when end-of-file is encountered on *stdin* (control-D typed on the terminal). The second program, **server.c**, is a primitive echo server. It listens for TCP connections on a port specified as a command-line argument and reads data from them. Data is printed and then sent back over the connection from which it came. Both programs use *select* with a read mask: **client.c** uses it to choose between keyboard and network input, while **server.c** uses it to select among exsting connections and requests for new connections. They both do unconditionally blocking output. A client that sends a large amount of data to the server and then fails to read the echo can cause the server to hang. In fact, because a client may be blocked sending to the server and thus unable to receive the echo, deadlock is possible. These problems could be solved (at the cost of additional complexity in the programs) by buffering data internally and using write masks in the *select* statements.

8.1. client.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <errno.h>
#include <ctype.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>

main(argc, argv)
int  argc;
char *argv[];
{
    struct hostent *hostp;
    struct servent *servp;
    struct sockaddr_in server;
    int sock;
    static struct timeval timeout = { 5, 0 }; /* five seconds */
    fd_set rmask, xmask, mask;
    char buf[BUFSIZ];
    int nfound, bytesread;

    if (argc != 3) {
        (void) fprintf(stderr,"usage: %s service host\n",argv[0]);
        exit(1);
    }
    if ((sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
        perror("socket");
        exit(1);
    }
    if (isdigit(argv[1][0])) {
        static struct servent s;
        servp = &s;
        s.s_port = htons((u_short)atoi(argv[1]));
    } else if ((servp = getservbyname(argv[1], "tcp")) == 0) {
        fprintf(stderr,"%s: unknown service\n",argv[1]);
        exit(1);
    }
    if ((hostp = gethostbyname(argv[2])) == 0) {
        fprintf(stderr,"%s: unknown host\n",argv[2]);
        exit(1);
    }
    memset((void *) &server, 0, sizeof server);
    server.sin_family = AF_INET;
    memcpy((void *) &server.sin_addr, hostp->h_addr, hostp->h_length);
    server.sin_port = servp->s_port;
    if (connect(sock, (struct sockaddr *)&server, sizeof server) < 0) {
        (void) close(sock);
        perror("connect");
        exit(1);
    }
    FD_ZERO(&mask);
    FD_SET(sock, &mask);
    FD_SET(fileno(stdin), &mask);
    for (;;) {
        rmask = mask;
        nfound = select(FD_SETSIZE, &rmask, (fd_set *)0, (fd_set *)0, &timeout);
        if (nfound < 0) {
            if (errno == EINTR) {
                printf("interrupted system call\n");
                continue;
            }
            /* something is very wrong! */
            perror("select");
            exit(1);
        }
        if (nfound == 0) {
            /* timer expired */
            printf("Please type something!\n");
            continue;
        }
        if (FD_ISSET(fileno(stdin), &rmask)) {
            /* data from keyboard */
            if (!fgets(buf, sizeof buf, stdin)) {
                if (ferror(stdin)) {
                    perror("stdin");
                    exit(1);
                }
                exit(0);
            }
            if (write(sock, buf, strlen(buf)) < 0) {
                perror("write");
                exit(1);
            }
        }
        if (FD_ISSET(sock,&rmask)) {
            /* data from network */
            bytesread = read(sock, buf, sizeof buf);
            buf[bytesread] = ’\0’;
            printf("%s: got %d bytes: %s\n", argv[0], bytesread, buf);
        }
    }
} /* main - client.c */
```

8.2. server.c

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

```
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <ctype.h>
#include <errno.h>
#include <stdio.h>

main(argc, argv)
int  argc;
char *argv[];
{
    struct servent *servp;
    struct sockaddr_in server, remote;
    int request_sock, new_sock;
    int nfound, fd, maxfd, bytesread, addrlen;
    fd_set rmask, mask;
    static struct timeval timeout = { 0, 500000 }; /* one half second */
    char buf[BUFSIZ];

    if (argc != 2) {
        (void) fprintf(stderr,"usage: %s service\n",argv[0]);
        exit(1);
    }
    if ((request_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
        perror("socket");
        exit(1);
    }
    if (isdigit(argv[1][0])) {
        static struct servent s;
        servp = &s;
        s.s_port = htons((u_short)atoi(argv[1]));
    } else if ((servp = getservbyname(argv[1], "tcp")) == 0) {
        fprintf(stderr,"%s: unknown service\n");
        exit(1);
    }
    memset((void *) &server, sizeof server);
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = servp->s_port;
    if (bind(request_sock, (struct sockaddr *)&server, sizeof server) < 0) {
        perror("bind");
        exit(1);
    }
    if (listen(request_sock, SOMAXCONN) < 0) {
        perror("listen");
        exit(1);
    }
    FD_ZERO(&mask);
    FD_SET(request_sock, &mask);
    maxfd = request_sock;
    for (;;) {
        rmask = mask;
        nfound = select(maxfd+1, &rmask, (fd_set *)0, (fd_set *)0, &timeout);
        if (nfound < 0) {
            if (errno == EINTR) {
                printf("interrupted system call\n");
                continue;
            }
            /* something is very wrong! */
            perror("select");
            exit(1);
        }
        if (nfound == 0) {
            /* timeout */
            printf("."); fflush(stdout);
            continue;
        }
        if (FD_ISSET(request_sock, &rmask)) {
            /* a new connection is available on the connetion socket */
            addrlen = sizeof(remote);
            new_sock = accept(request_sock,
                (struct sockaddr *)&remote, &addrlen);
            if (new_sock < 0) {
                perror("accept");
                exit(1);
            }
            printf("connection from host %s, port %d, socket %d\n",
                inet_ntoa(remote.sin_addr), ntohs(remote.sin_port),
                new_sock);
            FD_SET(new_sock, &mask);
            if (new_sock > maxfd)
                maxfd = new_sock;
            FD_CLR(request_sock, &rmask);
        }
        for (fd=0; fd <= maxfd ; fd++) {
            /* look for other sockets that have data available */
            if (FD_ISSET(fd, &rmask)) {
                /* process the data */
                bytesread = read(fd, buf, sizeof buf - 1);

                if (bytesread<0) {
                    perror("read");
                    /* fall through */
                }
                if (bytesread<=0) {
                    printf("server: end of file on %d\n",fd);
                    FD_CLR(fd, &mask);
                    if (close(fd)) perror("close");
                    continue;
                }
                buf[bytesread] = ’\0’;
                printf("%s: %d bytes from %d: %s\n",
                    argv[0], bytesread, fd, buf);
                /* echo it back */
                if (write(fd, buf, bytesread)!=bytesread)
                    perror("echo");
            }
        }
    }
} /* main - server.c */
```

9. References

[1] N. Hall, "The IPC Interface Under Berkeley Unit", CS838 Handout #3, UW−Madison Computer Science Department, April 1986.

[2] B.W. Kernighan and R. Pike, *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[3] S.J Leffler, W.N. Joy, and M.K. McKusick, *UNIX Programmer's Manual*, Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1983.

[4] S. Sechrest, "Tutorial Examples of Interprocess Communication in Berkeley UNIX 4.2bsd", Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1984.