An Introductory 4.4BSD Interprocess Communication Tutorial

Stuart Sechrest

Computer Science Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley

ABSTRACT

Berkeley UNIX† 4.4BSD offers several choices for interprocess communication. To aid the programmer in developing programs which are comprised of cooperating processes, the different choices are discussed and a series of example programs are presented. These programs demonstrate in a simple way the use of pipes, socketpairs, sockets and the use of datagram and stream communication. The intent of this document is to present a few simple example programs, not to describe the networking system in full.

1. Goals

Facilities for interprocess communication (IPC) and networking were a major addition to UNIX in the Berkeley UNIX 4.2BSD release. These facilities required major additions and some changes to the system interface. The basic idea of this interface is to make IPC similar to file I/O. In UNIX a process has a set of I/O descriptors, from which one reads and to which one writes. Descriptors may refer to normal files, to devices (including terminals), or to communication channels. The use of a descriptor has three phases: its creation, its use for reading and writing, and its destruction. By using descriptors to write files, rather than simply naming the target file in the write call, one gains a surprising amount of flexibility. Often, the program that creates a descriptor will be different from the program that uses the descriptor. For example the shell can create a descriptor for the output of the 'ls' command that will cause the listing to appear in a file rather than on a terminal. Pipes are another form of descriptor that have been used in UNIX for some time. Pipes allow one-way data transmission from one process to another; the two processes and the pipe must be set up by a common ancestor.

The use of descriptors is not the only communication interface provided by UNIX. The signal mechanism sends a tiny amount of information from one process to another. The signaled process receives only the signal type, not the identity of the sender, and the number of possible signals is small. The signal semantics limit the flexibility of the signaling mechanism as a means of interprocess communication.

The identification of IPC with I/O is quite longstanding in UNIX and has proved quite successful. At first, however, IPC was limited to processes communicating within a single machine. With Berkeley UNIX 4.2BSD this expanded to include IPC between machines. This expansion has necessitated some change in the way that descriptors are created. Additionally, new possibilities for the meaning of read and write have been admitted. Originally the meanings, or semantics, of these terms were fairly simple. When you wrote something it was delivered. When you read something, you were blocked until the data arrived. Other possibilities exist, however. One can write without full assurance of delivery if one can check later to catch occasional failures. Messages can be kept as discrete units or merged into a stream. One can ask to read, but insist on not waiting if nothing is immediately available. These new possibilities are allowed in the Berkeley UNIX IPC interface.

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

Thus Berkeley UNIX 4.4BSD offers several choices for IPC. This paper presents simple examples that illustrate some of the choices. The reader is presumed to be familiar with the C programming language [Kernighan & Ritchie 1978], but not necessarily with the system calls of the UNIX system or with processes and interprocess communication. The paper reviews the notion of a process and the types of communication that are supported by Berkeley UNIX 4.4BSD. A series of examples are presented that create processes that communicate with one another. The programs show different ways of establishing channels of communication. Finally, the calls that actually transfer data are reviewed. To clearly present how communication can take place, the example programs have been cleared of anything that might be construed as useful work. They can, therefore, serve as models for the programmer trying to construct programs which are comprised of cooperating processes.

2. Processes

A *program* is both a sequence of statements and a rough way of referring to the computation that occurs when the compiled statements are run. A *process* can be thought of as a single line of control in a program. Most programs execute some statements, go through a few loops, branch in various directions and then end. These are single process programs. Programs can also have a point where control splits into two independent lines, an action called *forking.* In UNIX these lines can never join again. A call to the system routine *fork()*, causes a process to split in this way. The result of this call is that two independent processes will be running, executing exactly the same code. Memory values will be the same for all values set before the fork, but, subsequently, each version will be able to change only the value of its own copy of each variable. Initially, the only difference between the two will be the value returned by *fork().* The parent will receive a process id for the child, the child will receive a zero. Calls to *fork(),* therefore, typically precede, or are included in, an if-statement.

A process views the rest of the system through a private table of descriptors. The descriptors can represent open files or sockets (sockets are communication objects that will be discussed below). Descriptors are referred to by their index numbers in the table. The first three descriptors are often known by special names, *stdin, stdout* and *stderr*. These are the standard input, output and error. When a process forks, its descriptor table is copied to the child. Thus, if the parent's standard input is being taken from a terminal (devices are also treated as files in UNIX), the child's input will be taken from the same terminal. Whoever reads first will get the input. If, before forking, the parent changes its standard input so that it is reading from a new file, the child will take its input from the new file. It is also possible to take input from a socket, rather than from a file.

3. Pipes

Most users of UNIX know that they can pipe the output of a program ''prog1'' to the input of another, ''prog2,'' by typing the command *''prog1 | prog2.''* This is called ''piping'' the output of one program to another because the mechanism used to transfer the output is called a pipe. When the user types a command, the command is read by the shell, which decides how to execute it. If the command is simple, for example, *''prog1,''* the shell forks a process, which executes the program, prog1, and then dies. The shell waits for this termination and then prompts for the next command. If the command is a compound command, *''prog1 | prog2,''* the shell creates two processes connected by a pipe. One process runs the program, prog1, the other runs prog2. The pipe is an I/O mechanism with two ends, or sockets. Data that is written into one socket can be read from the other.

Since a program specifies its input and output only by the descriptor table indices, which appear as variables or constants, the input source and output destination can be changed without changing the text of the program. It is in this way that the shell is able to set up pipes. Before executing prog1, the process can close whatever is at *stdout* and replace it with one end of a pipe. Similarly, the process that will execute prog2 can substitute the opposite end of the pipe for *stdin.*

Let us now examine a program that creates a pipe for communication between its child and itself (see Figure 1; all figures appear at the end of the document). A pipe is created by a parent process, which then forks. When a process forks, the parent's descriptor table is copied into the child's.

In Figure 1, the parent process makes a call to the system routine *pipe().* This routine creates a pipe and places descriptors for the sockets for the two ends of the pipe in the process's descriptor table. *Pipe()* is passed an array into which it places the index numbers of the sockets it created. The two ends are not equivalent. The socket whose index is returned in the low word of the array is opened for reading only, while the socket in the high end is opened only for writing. This corresponds to the fact that the standard input is the first descriptor of a process's descriptor table and the standard output is the second. After creating the pipe, the parent creates the child with which it will share the pipe by calling *fork().* Figure 2 illustrates the effect of a fork. The parent process's descriptor table points to both ends of the pipe. After the fork, both parent's and child's descriptor tables point to the pipe. The child can then use the pipe to send a message to the parent.

Just what is a pipe? It is a one-way communication mechanism, with one end opened for reading and the other end for writing. Therefore, parent and child need to agree on which way to turn the pipe, from parent to child or the other way around. Using the same pipe for communication both from parent to child and from child to parent would be possible (since both processes have references to both ends), but very complicated. If the parent and child are to have a two-way conversation, the parent creates two pipes, one for use in each direction. (In accordance with their plans, both parent and child in the example above close the socket that they will not use. It is not required that unused descriptors be closed, but it is good practice.) A pipe is also a *stream* communication mechanism; that is, all messages sent through the pipe are placed in order and reliably delivered. When the reader asks for a certain number of bytes from this stream, he is given as many bytes as are available, up to the amount of the request. Note that these bytes may have come from the same call to *write()* or from several calls to *write()* which were concatenated.

4. Socketpairs

Berkeley UNIX 4.4BSD provides a slight generalization of pipes. A pipe is a pair of connected sockets for one-way stream communication. One may obtain a pair of connected sockets for two-way stream communication by calling the routine *socketpair().* The program in Figure 3 calls *socketpair()* to create such a connection. The program uses the link for communication in both directions. Since socketpairs are an extension of pipes, their use resembles that of pipes. Figure 4 illustrates the result of a fork following a call to *socketpair().*

Socketpair() takes as arguments a specification of a domain, a style of communication, and a protocol. These are the parameters shown in the example. Domains and protocols will be discussed in the next section. Briefly, a domain is a space of names that may be bound to sockets and implies certain other conventions. Currently, socketpairs have only been implemented for one domain, called the UNIX domain. The UNIX domain uses UNIX path names for naming sockets. It only allows communication between sockets on the same machine.

Note that the header files *<sys/socket.h>* and *<sys/types.h>*. are required in this program. The constants AF_UNIX and SOCK_STREAM are defined in *<sys/socket.h>*, which in turn requires the file *<sys/types.h>* for some of its definitions.

5. Domains and Protocols

Pipes and socketpairs are a simple solution for communicating between a parent and child or between child processes. What if we wanted to have processes that have no common ancestor with whom to set up communication? Neither standard UNIX pipes nor socketpairs are the answer here, since both mechanisms require a common ancestor to set up the communication. We would like to have two processes separately create sockets and then have messages sent between them. This is often the case when providing or using a service in the system. This is also the case when the communicating processes are on separate machines. In Berkeley UNIX 4.4BSD one can create individual sockets, give them names and send messages between them.

Sockets created by different programs use names to refer to one another; names generally must be translated into addresses for use. The space from which an address is drawn is referred to as a *domain.* There are several domains for sockets. Two that will be used in the examples here are the UNIX domain (or AF_UNIX, for Address Format UNIX) and the Internet domain (or AF_INET). UNIX domain IPC is an experimental facility in 4.2BSD and 4.3BSD. In the UNIX domain, a socket is given a path name within the file system name space. A file system node is created for the socket and other processes may then refer to the socket by giving the proper pathname. UNIX domain names, therefore, allow communication between any two processes that work in the same file system. The Internet domain is the UNIX implementation of the DARPA Internet standard protocols IP/TCP/UDP. Addresses in the Internet domain consist of a machine network address and an identifying number, called a port. Internet domain names allow communication between machines.

Communication follows some particular ‘‘style.’’ Currently, communication is either through a *stream* or by *datagram.* Stream communication implies several things. Communication takes place across a connection between two sockets. The communication is reliable, error-free, and, as in pipes, no message boundaries are kept. Reading from a stream may result in reading the data sent from one or several calls to *write()* or only part of the data from a single call, if there is not enough room for the entire message, or if not all the data from a large message has been transferred. The protocol implementing such a style will retransmit messages received with errors. It will also return error messages if one tries to send a message after the connection has been broken. Datagram communication does not use connections. Each message is addressed individually. If the address is correct, it will generally be received, although this is not guaranteed. Often datagrams are used for requests that require a response from the recipient. If no response arrives in a reasonable amount of time, the request is repeated. The individual datagrams will be kept separate when they are read, that is, message boundaries are preserved.

The difference in performance between the two styles of communication is generally less important than the difference in semantics. The performance gain that one might find in using datagrams must be weighed against the increased complexity of the program, which must now concern itself with lost or out of order messages. If lost messages may simply be ignored, the quantity of traffic may be a consideration. The expense of setting up a connection is best justified by frequent use of the connection. Since the performance of a protocol changes as it is tuned for different situations, it is best to seek the most up-to-date information when making choices for a program in which performance is crucial.

A protocol is a set of rules, data formats and conventions that regulate the transfer of data between participants in the communication. In general, there is one protocol for each socket type (stream, datagram, etc.) within each domain. The code that implements a protocol keeps track of the names that are bound to sockets, sets up connections and transfers data between sockets, perhaps sending the data across a network. This code also keeps track of the names that are bound to sockets. It is possible for several protocols, differing only in low level details, to implement the same style of communication within a particular domain. Although it is possible to select which protocol should be used, for nearly all uses it is sufficient to request the default protocol. This has been done in all of the example programs.

One specifies the domain, style and protocol of a socket when it is created. For example, in Figure 5a the call to *socket()* causes the creation of a datagram socket with the default protocol in the UNIX domain.

6. Datagrams in the UNIX Domain

Let us now look at two programs that create sockets separately. The programs in Figures 5a and 5b use datagram communication rather than a stream. The structure used to name UNIX domain sockets is defined in the file <*sys/un.h*>. The definition has also been included in the example for clarity.

Each program creates a socket with a call to *socket().* These sockets are in the UNIX domain. Once a name has been decided upon it is attached to a socket by the system call *bind().* The program in Figure 5a uses the name ‘‘socket’’, which it binds to its socket. This name will appear in the working directory of the program. The routines in Figure 5b use its socket only for sending messages. It does not create a name for the socket because no other process has to refer to it.

Names in the UNIX domain are path names. Like file path names they may be either absolute (e.g. ‘‘/dev/imaginary’’) or relative (e.g. ‘‘socket’’). Because these names are used to allow processes to rendezvous, relative path names can pose difficulties and should be used with care. When a name is bound into the name space, a file (inode) is allocated in the file system. If the inode is not deallocated, the name will continue to exist even after the bound socket is closed. This can cause subsequent runs of a program to find that a name is unavailable, and can cause directories to fill up with these objects. The names are removed by calling *unlink()* or using the *rm* (1) command. Names in the UNIX domain are only used for rendezvous. They are not used for message delivery once a connection is established. Therefore, in contrast with the Internet domain, unbound sockets need not be (and are not) automatically given addresses when they are connected.

There is no established means of communicating names to interested parties. In the example, the program in Figure 5b gets the name of the socket to which it will send its message through its command line arguments. Once a line of communication has been created, one can send the names of additional, perhaps new, sockets over the link. Facilities will have to be built that will make the distribution of names less of a problem than it now is.

7. Datagrams in the Internet Domain

The examples in Figure 6a and 6b are very close to the previous example except that the socket is in the Internet domain. The structure of Internet domain addresses is defined in the file <*netinet/in.h*>. Internet addresses specify a host address (a 32-bit number) and a delivery slot, or port, on that machine. These ports are managed by the system routines that implement a particular protocol. Unlike UNIX domain names, Internet socket names are not entered into the file system and, therefore, they do not have to be unlinked after the socket has been closed. When a message must be sent between machines it is sent to the protocol routine on the destination machine, which interprets the address to determine to which socket the message should be delivered. Several different protocols may be active on the same machine, but, in general, they will not communicate with one another. As a result, different protocols are allowed to use the same port numbers. Thus, implicitly, an Internet address is a triple including a protocol as well as the port and machine address. An *association* is a temporary or permanent specification of a pair of communicating sockets. An association is thus identified by the tuple <*protocol, local machine address, local port, remote machine address, remote port*>. An association may be transient when using datagram sockets; the association actually exists during a *send* operation.

The protocol for a socket is chosen when the socket is created. The local machine address for a socket can be any valid network address of the machine, if it has more than one, or it can be the wildcard value INADDR_ANY. The wildcard value is used in the program in Figure 6a. If a machine has several network addresses, it is likely that messages sent to any of the addresses should be deliverable to a socket. This will be the case if the wildcard value has been chosen. Note that even if the wildcard value is chosen, a program sending messages to the named socket must specify a valid network address. One can be willing to receive from ‘‘anywhere,’’ but one cannot send a message ‘‘anywhere.’’ The program in Figure 6b is given the destination host name as a command line argument. To determine a network address to which it can send the message, it looks up the host address by the call to *gethostbyname()*. The returned structure includes the host’s network address, which is copied into the structure specifying the destination of the message.

The port number can be thought of as the number of a mailbox, into which the protocol places one’s messages. Certain daemons, offering certain advertised services, have reserved or ‘‘well-known’’ port numbers. These fall in the range from 1 to 1023. Higher numbers are available to general users. Only servers need to ask for a particular number. The system will assign an unused port number when an address is bound to a socket. This may happen when an explicit *bind* call is made with a port number of 0, or when a *connect* or *send* is performed on an unbound socket. Note that port numbers are not automatically reported back to the user. After calling *bind(),* asking for port 0, one may call *getsockname()* to discover what port was actually assigned. The routine *getsockname()* will not work for names in the UNIX domain.

The format of the socket address is specified in part by standards within the Internet domain. The specification includes the order of the bytes in the address. Because machines differ in the internal representation they ordinarily use to represent integers, printing out the port number as returned by *getsockname()* may result in a misinterpretation. To print out the number, it is necessary to use the routine *ntohs()* (for *network to host: short*) to convert the number from the network representation to the host’s representation. On some machines, such as 68000-based machines, this is a null operation. On others, such as VAXes, this results in a swapping of bytes. Another routine exists to convert a short integer from the host format to the network format, called *htons()*; similar routines exist for long integers. For further information, refer to the entry for *byteorder* in section 3 of the manual.

8. Connections

To send data between stream sockets (having communication style SOCK_STREAM), the sockets must be connected. Figures 7a and 7b show two programs that create such a connection. The program in 7a is relatively simple. To initiate a connection, this program simply creates a stream socket, then calls *connect(),* specifying the address of the socket to which it wishes its socket connected. Provided that the target socket exists and is prepared to handle a connection, connection will be complete, and the program can begin to send messages. Messages will be delivered in order without message boundaries, as with pipes. The connection is destroyed when either socket is closed (or soon thereafter). If a process persists in sending messages after the connection is closed, a SIGPIPE signal is sent to the process by the operating system. Unless explicit action is taken to handle the signal (see the manual page for *signal* or *sigvec*), the process will terminate and the shell will print the message ''broken pipe.''

Forming a connection is asymmetrical; one process, such as the program in Figure 7a, requests a connection with a particular socket, the other process accepts connection requests. Before a connection can be accepted a socket must be created and an address bound to it. This situation is illustrated in the top half of Figure 8. Process 2 has created a socket and bound a port number to it. Process 1 has created an unnamed socket. The address bound to process 2's socket is then made known to process 1 and, perhaps to several other potential communicants as well. If there are several possible communicants, this one socket might receive several requests for connections. As a result, a new socket is created for each connection. This new socket is the endpoint for communication within this process for this connection. A connection may be destroyed by closing the corresponding socket.

The program in Figure 7b is a rather trivial example of a server. It creates a socket to which it binds a name, which it then advertises. (In this case it prints out the socket number.) The program then calls *listen()* for this socket. Since several clients may attempt to connect more or less simultaneously, a queue of pending connections is maintained in the system address space. *Listen()* marks the socket as willing to accept connections and initializes the queue. When a connection is requested, it is listed in the queue. If the queue is full, an error status may be returned to the requester. The maximum length of this queue is specified by the second argument of *listen()*; the maximum length is limited by the system. Once the listen call has been completed, the program enters an infinite loop. On each pass through the loop, a new connection is accepted and removed from the queue, and, hence, a new socket for the connection is created. The bottom half of Figure 8 shows the result of Process 1 connecting with the named socket of Process 2, and Process 2 accepting the connection. After the connection is created, the service, in this case printing out the messages, is performed and the connection socket closed. The *accept()* call will take a pending connection request from the queue if one is available, or block waiting for a request. Messages are read from the connection socket. Reads from an active connection will normally block until data is available. The number of bytes read is returned. When a connection is destroyed, the read call returns immediately. The number of bytes returned will be zero.

The program in Figure 7c is a slight variation on the server in Figure 7b. It avoids blocking when there are no pending connection requests by calling *select()* to check for pending requests before calling *accept().* This strategy is useful when connections may be received on more than one socket, or when data may arrive on other connected sockets before another connection request.

The programs in Figures 9a and 9b show a program using stream communication in the UNIX domain. Streams in the UNIX domain can be used for this sort of program in exactly the same way as Internet domain streams, except for the form of the names and the restriction of the connections to a single file system. There are some differences, however, in the functionality of streams in the two domains, notably in the handling of *out-of-band* data (discussed briefly below). These differences are beyond the scope of this paper.

9. Reads, Writes, Recvs, etc.

UNIX 4.4BSD has several system calls for reading and writing information. The simplest calls are *read()* and *write(). Write()* takes as arguments the index of a descriptor, a pointer to a buffer containing the data and the size of the data. The descriptor may indicate either a file or a connected socket. ''Connected'' can mean either a connected stream socket (as described in Section 8) or a datagram socket for which a *connect()* call has provided a default destination (see the *connect()* manual page). *Read()* also takes a descriptor that indicates either a file or a socket. *Write()* requires a connected socket since no destination is specified in the parameters of the system call. *Read()* can be used for either a connected or an unconnected socket. These calls are, therefore, quite flexible and may be used to write applications that require no assumptions about the source of their input or the destination of their output. There are variations on *read()* and *write()* that allow the source and destination of the input and output to use several separate buffers, while retaining the flexibility to handle both files and sockets. These are *readv()* and *writev(),* for read and write *vector.*

It is sometimes necessary to send high priority data over a connection that may have unread low priority data at the other end. For example, a user interface process may be interpreting commands and sending them on to another process through a stream connection. The user interface may have filled the stream with as yet unprocessed requests when the user types a command to cancel all outstanding requests. Rather than have the high priority data wait to be processed after the low priority data, it is possible to send it as *out-of-band* (OOB) data. The notification of pending OOB data results in the generation of a SIGURG signal, if this signal has been enabled (see the manual page for *signal* or *sigvec*). See [Leffler 1986] for a more complete description of the OOB mechanism. There are a pair of calls similar to *read* and *write* that allow options, including sending and receiving OOB information; these are *send()* and *recv().* These calls are used only with sockets; specifying a descriptor for a file will result in the return of an error status. These calls also allow *peeking* at data in a stream. That is, they allow a process to read data without removing the data from the stream. One use of this facility is to read ahead in a stream to determine the size of the next item to be read. When not using these options, these calls have the same functions as *read()* and *write().*

To send datagrams, one must be allowed to specify the destination. The call *sendto()* takes a destination address as an argument and is therefore used for sending datagrams. The call *recvfrom()* is often used to read datagrams, since this call returns the address of the sender, if it is available, along with the data. If the identity of the sender does not matter, one may use *read()* or *recv().*

Finally, there are a pair of calls that allow the sending and receiving of messages from multiple buffers, when the address of the recipient must be specified. These are *sendmsg()* and *recvmsg().* These calls are actually quite general and have other uses, including, in the UNIX domain, the transmission of a file descriptor from one process to another.

The various options for reading and writing are shown in Figure 10, together with their parameters. The parameters for each system call reflect the differences in function of the different calls. In the examples given in this paper, the calls *read()* and *write()* have been used whenever possible.

10. Choices

This paper has presented examples of some of the forms of communication supported by Berkeley UNIX 4.4BSD. These have been presented in an order chosen for ease of presentation. It is useful to review these options emphasizing the factors that make each attractive.

Pipes have the advantage of portability, in that they are supported in all UNIX systems. They also are relatively simple to use. Socketpairs share this simplicity and have the additional advantage of allowing bidirectional communication. The major shortcoming of these mechanisms is that they require communicating processes to be descendants of a common process. They do not allow intermachine communication.

The two communication domains, UNIX and Internet, allow processes with no common ancestor to communicate. Of the two, only the Internet domain allows communication between machines. This makes the Internet domain a necessary choice for processes running on separate machines.

The choice between datagrams and stream communication is best made by carefully considering the semantic and performance requirements of the application. Streams can be both advantageous and disadvantageous. One disadvantage is that a process is only allowed a limited number of open streams, as there are usually only 64 entries available in the open descriptor table. This can cause problems if a single server must talk with a large number of clients. Another is that for delivering a short message the stream setup and teardown time can be unnecessarily long. Weighed against this are the reliability built into the streams. This will often be the deciding factor in favor of streams.

11. What to do Next

Many of the examples presented here can serve as models for multiprocess programs and for programs distributed across several machines. In developing a new multiprocess program, it is often easiest to first write the code to create the processes and communication paths. After this code is debugged, the code specific to the application can be added.

An introduction to the UNIX system and programming using UNIX system calls can be found in [Kernighan and Pike 1984]. Further documentation of the Berkeley UNIX 4.4BSD IPC mechanisms can be found in [Leffler et al. 1986]. More detailed information about particular calls and protocols is provided in sections 2, 3 and 4 of the UNIX Programmer's Manual [CSRG 1986]. In particular the following manual pages are relevant:

creating and naming sockets	socket(2), bind(2)
establishing connections	listen(2), accept(2), connect(2)
transferring data	read(2), write(2), send(2), recv(2)
addresses	inet(4F)
protocols	tcp(4P), udp(4P).

Acknowledgements

I would like to thank Sam Leffler and Mike Karels for their help in understanding the IPC mechanisms and all the people whose comments have helped in writing and improving this report.

This work was sponsored by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4031, monitored by the Naval Electronics Systems Command under contract No. N00039-C-0235. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Government.

References

B.W. Kernighan & R. Pike, 1984,
The UNIX Programming Environment.
Englewood Cliffs, N.J.: Prentice-Hall.

B.W. Kernighan & D.M. Ritchie, 1978,
The C Programming Language,
Englewood Cliffs, N.J.: Prentice-Hall.

S.J. Leffler, R.S. Fabry, W.N. Joy, P. Lapsley, S. Miller & C. Torek, 1986,
An Advanced 4.4BSD Interprocess Communication Tutorial.
Computer Systems Research Group,
Department of Electrical Engineering and Computer Science,
University of California, Berkeley.

Computer Systems Research Group, 1986,
UNIX Programmer's Manual, 4.4 Berkeley Software Distribution.
Computer Systems Research Group,
Department of Electrical Engineering and Computer Science,
University of California, Berkeley.

```
#include <stdio.h>

#define DATA "Bright star, would I were steadfast as thou art . . ."

/*
 * This program creates a pipe, then forks.  The child communicates to the
 * parent over the pipe. Notice that a pipe is a one-way communications
 * device.  I can write to the output socket (sockets[1], the second socket
 * of the array returned by pipe()) and read from the input socket
 * (sockets[0]), but not vice versa.
 */

main()
{
      int sockets[2], child;

      /* Create a pipe */
      if (pipe(sockets) < 0) {
            perror("opening stream socket pair");
            exit(10);
      }

      if ((child = fork()) == -1)
            perror("fork");
      else if (child) {
            char buf[1024];

            /* This is still the parent.  It reads the child’s message. */
            close(sockets[1]);
            if (read(sockets[0], buf, 1024) < 0)
                  perror("reading message");
            printf("-->%s\n", buf);
            close(sockets[0]);
      } else {
            /* This is the child.  It writes a message to its parent. */
            close(sockets[0]);
            if (write(sockets[1], DATA, sizeof(DATA)) < 0)
                  perror("writing message");
            close(sockets[1]);
      }
}
```

Figure 1 Use of a pipe

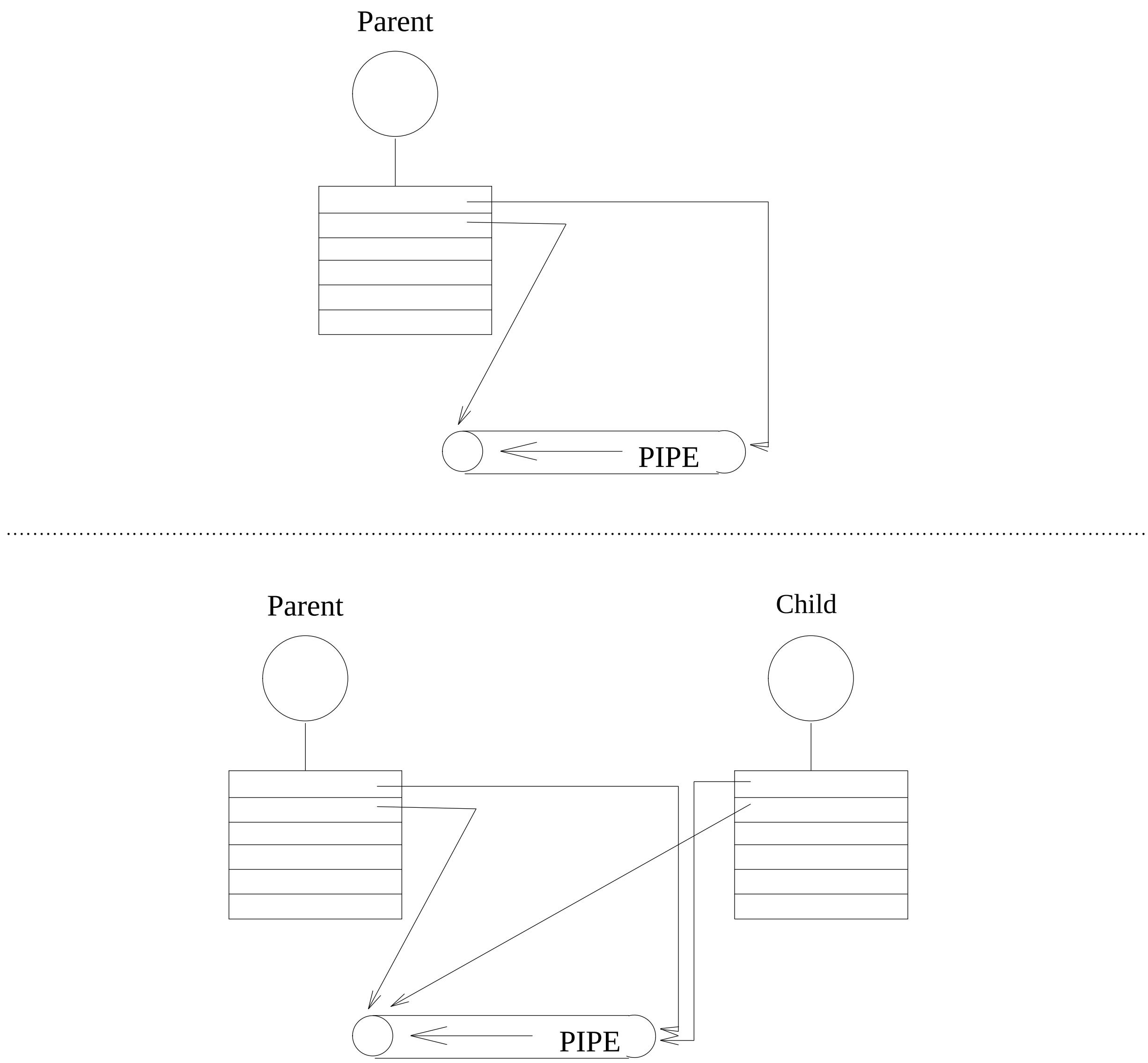

Figure 2 Sharing a pipe between parent and child

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

#define DATA1 "In Xanadu, did Kublai Khan . . ."
#define DATA2 "A stately pleasure dome decree . . ."

/*
 * This program creates a pair of connected sockets then forks and
 * communicates over them.  This is very similar to communication with pipes,
 * however, socketpairs are two-way communications objects. Therefore I can
 * send messages in both directions.
 */

main()
{
      int sockets[2], child;
      char buf[1024];

      if (socketpair(AF_UNIX, SOCK_STREAM, 0, sockets) < 0) {
            perror("opening stream socket pair");
            exit(1);
      }

      if ((child = fork()) == -1)
            perror("fork");
      else if (child) { /* This is the parent. */
            close(sockets[0]);
            if (read(sockets[1], buf, 1024, 0) < 0)
                  perror("reading stream message");
            printf("-->%s\n", buf);
            if (write(sockets[1], DATA2, sizeof(DATA2)) < 0)
                  perror("writing stream message");
            close(sockets[1]);
      } else {          /* This is the child. */
            close(sockets[1]);
            if (write(sockets[0], DATA1, sizeof(DATA1)) < 0)
                  perror("writing stream message");
            if (read(sockets[0], buf, 1024, 0) < 0)
                  perror("reading stream message");
            printf("-->%s\n", buf);
            close(sockets[0]);
      }
}
```

Figure 3 Use of a socketpair

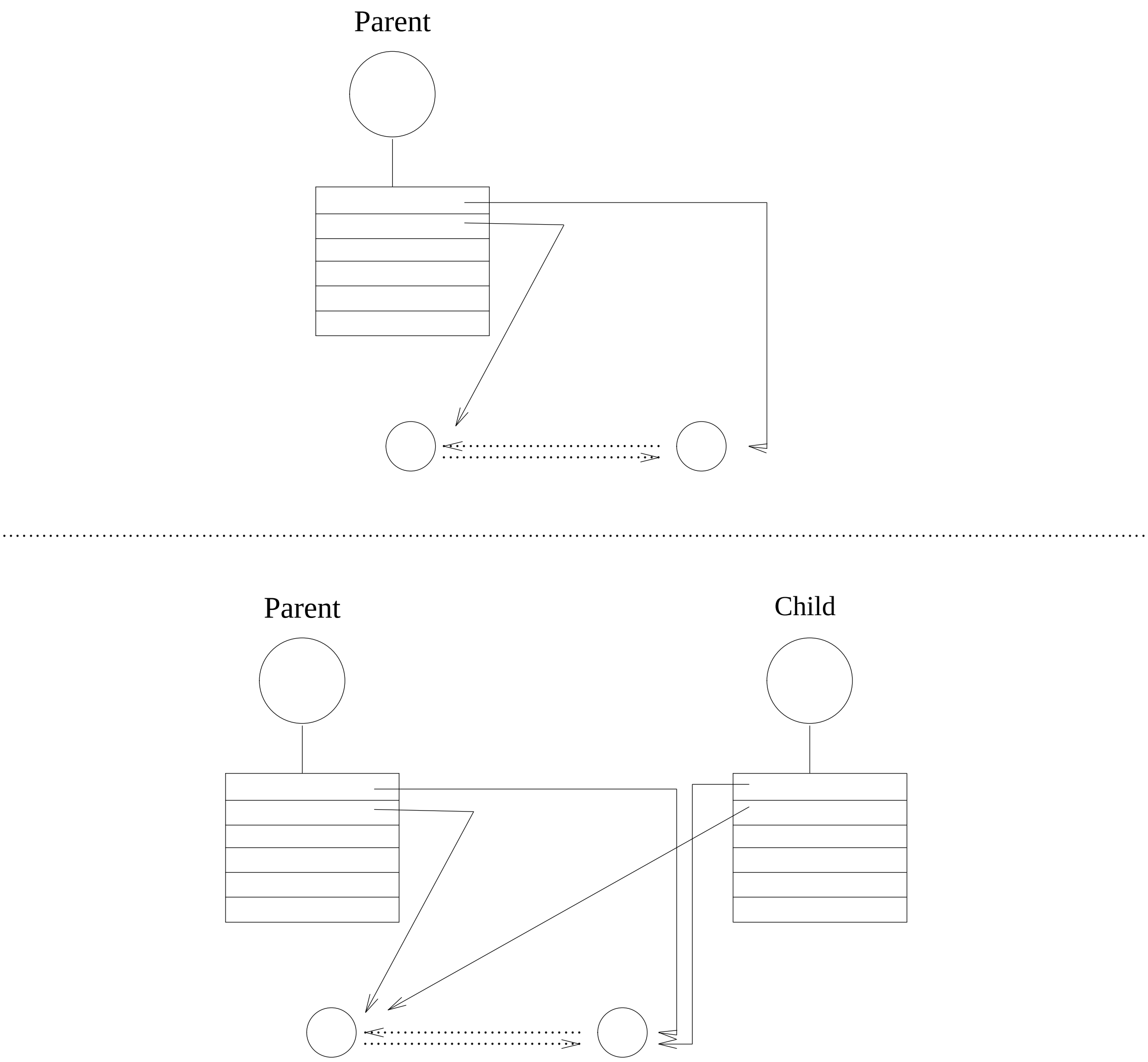

Figure 4 Sharing a socketpair between parent and child

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

/*
 * In the included file <sys/un.h> a sockaddr_un is defined as follows
 * struct sockaddr_un {
 *    short sun_family;
 *    char  sun_path[108];
 * };
 */

#include <stdio.h>

#define NAME "socket"

/*
 * This program creates a UNIX domain datagram socket, binds a name to it,
 * then reads from the socket.
 */
main()
{
      int sock, length;
      struct sockaddr_un name;
      char buf[1024];

      /* Create socket from which to read. */
      sock = socket(AF_UNIX, SOCK_DGRAM, 0);
      if (sock < 0) {
            perror("opening datagram socket");
            exit(1);
      }
      /* Create name. */
      name.sun_family = AF_UNIX;
      strcpy(name.sun_path, NAME);
      if (bind(sock, &name, sizeof(struct sockaddr_un))) {
            perror("binding name to datagram socket");
            exit(1);
      }
      printf("socket -->%s\n", NAME);
      /* Read from the socket */
      if (read(sock, buf, 1024) < 0)
            perror("receiving datagram packet");
      printf("-->%s\n", buf);
      close(sock);
      unlink(NAME);
}
```

Figure 5a Reading UNIX domain datagrams

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

#define DATA "The sea is calm tonight, the tide is full . . ."

/*
 * Here I send a datagram to a receiver whose name I get from the command
 * line arguments.  The form of the command line is udgramsend pathname
 */

main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_un name;

      /* Create socket on which to send. */
      sock = socket(AF_UNIX, SOCK_DGRAM, 0);
      if (sock < 0) {
            perror("opening datagram socket");
            exit(1);
      }
      /* Construct name of socket to send to. */
      name.sun_family = AF_UNIX;
      strcpy(name.sun_path, argv[1]);
      /* Send message. */
      if (sendto(sock, DATA, sizeof(DATA), 0,
          &name, sizeof(struct sockaddr_un)) < 0) {
            perror("sending datagram message");
      }
      close(sock);
}
```

Figure 5b Sending a UNIX domain datagrams

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

/*
 * In the included file <netinet/in.h> a sockaddr_in is defined as follows:
 * struct sockaddr_in {
 *    short sin_family;
 *    u_short     sin_port;
 *    struct in_addr sin_addr;
 *    char  sin_zero[8];
 * };
 *
 * This program creates a datagram socket, binds a name to it, then reads
 * from the socket.
 */
main()
{
      int sock, length;
      struct sockaddr_in name;
      char buf[1024];

      /* Create socket from which to read. */
      sock = socket(AF_INET, SOCK_DGRAM, 0);
      if (sock < 0) {
            perror("opening datagram socket");
            exit(1);
      }
      /* Create name with wildcards. */
      name.sin_family = AF_INET;
      name.sin_addr.s_addr = INADDR_ANY;
      name.sin_port = 0;
      if (bind(sock, &name, sizeof(name))) {
            perror("binding datagram socket");
            exit(1);
      }
      /* Find assigned port value and print it out. */
      length = sizeof(name);
      if (getsockname(sock, &name, &length)) {
            perror("getting socket name");
            exit(1);
      }
      printf("Socket has port #%d\n", ntohs(name.sin_port));
      /* Read from the socket */
      if (read(sock, buf, 1024) < 0)
            perror("receiving datagram packet");
      printf("-->%s\n", buf);
      close(sock);
}
```

Figure 6a Reading Internet domain datagrams

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

#define DATA "The sea is calm tonight, the tide is full . . ."

/*
 * Here I send a datagram to a receiver whose name I get from the command
 * line arguments.  The form of the command line is dgramsend hostname
 * portnumber
 */

main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_in name;
      struct hostent *hp, *gethostbyname();

      /* Create socket on which to send. */
      sock = socket(AF_INET, SOCK_DGRAM, 0);
      if (sock < 0) {
            perror("opening datagram socket");
            exit(1);
      }
      /*
       * Construct name, with no wildcards, of the socket to send to.
       * Getnostbyname() returns a structure including the network address
       * of the specified host.  The port number is taken from the command
       * line.
       */
      hp = gethostbyname(argv[1]);
      if (hp == 0) {
            fprintf(stderr, "%s: unknown host0, argv[1]);
            exit(2);
      }
      bcopy(hp->h_addr, &name.sin_addr, hp->h_length);
      name.sin_family = AF_INET;
      name.sin_port = htons(atoi(argv[2]));
      /* Send message. */
      if (sendto(sock, DATA, sizeof(DATA), 0, &name, sizeof(name)) < 0)
            perror("sending datagram message");
      close(sock);
}
```

Figure 6b Sending an Internet domain datagram

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

#define DATA "Half a league, half a league . . ."

/*
 * This program creates a socket and initiates a connection with the socket
 * given in the command line.  One message is sent over the connection and
 * then the socket is closed, ending the connection. The form of the command
 * line is streamwrite hostname portnumber
 */

main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_in server;
      struct hostent *hp, *gethostbyname();
      char buf[1024];

      /* Create socket */
      sock = socket(AF_INET, SOCK_STREAM, 0);
      if (sock < 0) {
            perror("opening stream socket");
            exit(1);
      }
      /* Connect socket using name specified by command line. */
      server.sin_family = AF_INET;
      hp = gethostbyname(argv[1]);
      if (hp == 0) {
            fprintf(stderr, "%s: unknown host0, argv[1]);
            exit(2);
      }
      bcopy(hp->h_addr, &server.sin_addr, hp->h_length);
      server.sin_port = htons(atoi(argv[2]));

      if (connect(sock, &server, sizeof(server)) < 0) {
            perror("connecting stream socket");
            exit(1);
      }
      if (write(sock, DATA, sizeof(DATA)) < 0)
            perror("writing on stream socket");
      close(sock);
}
```

Figure 7a Initiating an Internet domain stream connection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1

/*
 * This program creates a socket and then begins an infinite loop. Each time
 * through the loop it accepts a connection and prints out messages from it.
 * When the connection breaks, or a termination message comes through, the
 * program accepts a new connection.
 */

main()
{
	int sock, length;
	struct sockaddr_in server;
	int msgsock;
	char buf[1024];
	int rval;
	int i;

	/* Create socket */
	sock = socket(AF_INET, SOCK_STREAM, 0);
	if (sock < 0) {
		perror("opening stream socket");
		exit(1);
	}
	/* Name socket using wildcards */
	server.sin_family = AF_INET;
	server.sin_addr.s_addr = INADDR_ANY;
	server.sin_port = 0;
	if (bind(sock, &server, sizeof(server))) {
		perror("binding stream socket");
		exit(1);
	}
	/* Find out assigned port number and print it out */
	length = sizeof(server);
	if (getsockname(sock, &server, &length)) {
		perror("getting socket name");
		exit(1);
	}
	printf("Socket has port #%d\n", ntohs(server.sin_port));

	/* Start accepting connections */
	listen(sock, 5);
	do {
		msgsock = accept(sock, 0, 0);
		if (msgsock == -1)
			perror("accept");
		else do {
			bzero(buf, sizeof(buf));
			if ((rval = read(msgsock, buf, 1024)) < 0)
				perror("reading stream message");
			i = 0;
			if (rval == 0)
```

```
                        printf("Ending connection\n");
                else
                        printf("-->%s\n", buf);
            } while (rval != 0);
            close(msgsock);
      } while (TRUE);
      /*
       * Since this program has an infinite loop, the socket "sock" is
       * never explicitly closed.  However, all sockets will be closed
       * automatically when a process is killed or terminates normally.
       */
}
```

Figure 7b Accepting an Internet domain stream connection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1

/*
 * This program uses select() to check that someone is trying to connect
 * before calling accept().
 */

main()
{
      int sock, length;
      struct sockaddr_in server;
      int msgsock;
      char buf[1024];
      int rval;
      fd_set ready;
      struct timeval to;

      /* Create socket */
      sock = socket(AF_INET, SOCK_STREAM, 0);
      if (sock < 0) {
            perror("opening stream socket");
            exit(1);
      }
      /* Name socket using wildcards */
      server.sin_family = AF_INET;
      server.sin_addr.s_addr = INADDR_ANY;
      server.sin_port = 0;
      if (bind(sock, &server, sizeof(server))) {
            perror("binding stream socket");
            exit(1);
      }
      /* Find out assigned port number and print it out */
      length = sizeof(server);
      if (getsockname(sock, &server, &length)) {
            perror("getting socket name");
            exit(1);
```

```
	}
	printf("Socket has port #%d\n", ntohs(server.sin_port));

	/* Start accepting connections */
	listen(sock, 5);
	do {
		FD_ZERO(&ready);
		FD_SET(sock, &ready);
		to.tv_sec = 5;
		if (select(sock + 1, &ready, 0, 0, &to) < 0) {
			perror("select");
			continue;
		}
		if (FD_ISSET(sock, &ready)) {
			msgsock = accept(sock, (struct sockaddr *)0, (int *)0);
			if (msgsock == -1)
				perror("accept");
			else do {
				bzero(buf, sizeof(buf));
				if ((rval = read(msgsock, buf, 1024)) < 0)
					perror("reading stream message");
				else if (rval == 0)
					printf("Ending connection\n");
				else
					printf("-->%s\n", buf);
			} while (rval > 0);
			close(msgsock);
		} else
			printf("Do something else\n");
	} while (TRUE);
}
```

Figure 7c Using select() to check for pending connections

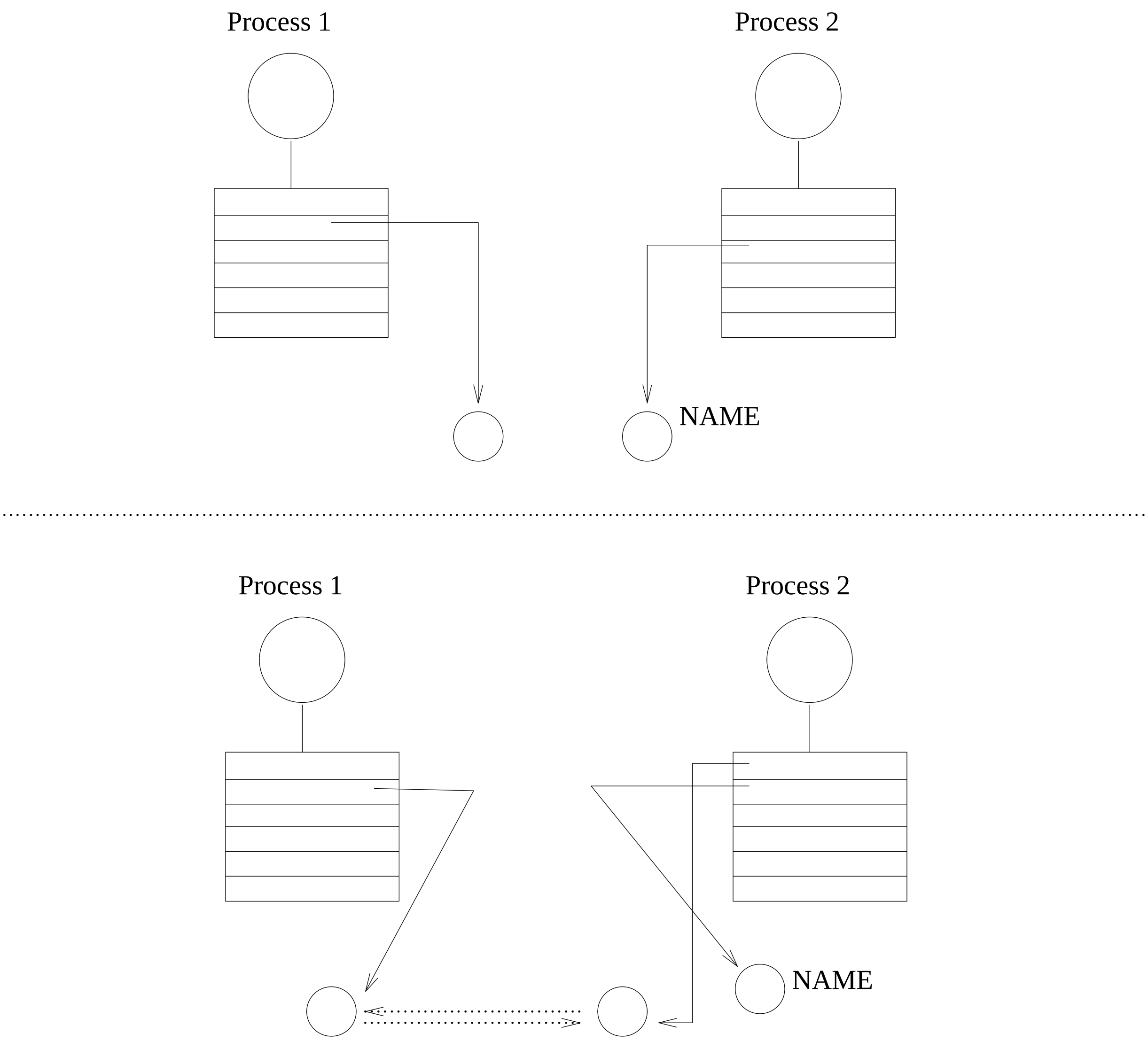

Figure 8 Establishing a stream connection

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

#define DATA "Half a league, half a league . . ."

/*
 * This program connects to the socket named in the command line and sends a
 * one line message to that socket. The form of the command line is
 * ustreamwrite pathname
 */
main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_un server;
      char buf[1024];

      /* Create socket */
      sock = socket(AF_UNIX, SOCK_STREAM, 0);
      if (sock < 0) {
            perror("opening stream socket");
            exit(1);
      }
      /* Connect socket using name specified by command line. */
      server.sun_family = AF_UNIX;
      strcpy(server.sun_path, argv[1]);

      if (connect(sock, &server, sizeof(struct sockaddr_un)) < 0) {
            close(sock);
            perror("connecting stream socket");
            exit(1);
      }
      if (write(sock, DATA, sizeof(DATA)) < 0)
            perror("writing on stream socket");
}
```

Figure 9a Initiating a UNIX domain stream connection

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

#define NAME "socket"

/*
 * This program creates a socket in the UNIX domain and binds a name to it.
 * After printing the socket’s name it begins a loop. Each time through the
 * loop it accepts a connection and prints out messages from it.  When the
 * connection breaks, or a termination message comes through, the program
 * accepts a new connection.
 */
main()
{
```

```
	int sock, msgsock, rval;
	struct sockaddr_un server;
	char buf[1024];

	/* Create socket */
	sock = socket(AF_UNIX, SOCK_STREAM, 0);
	if (sock < 0) {
		perror("opening stream socket");
		exit(1);
	}
	/* Name socket using file system name */
	server.sun_family = AF_UNIX;
	strcpy(server.sun_path, NAME);
	if (bind(sock, &server, sizeof(struct sockaddr_un))) {
		perror("binding stream socket");
		exit(1);
	}
	printf("Socket has name %s\n", server.sun_path);
	/* Start accepting connections */
	listen(sock, 5);
	for (;;) {
		msgsock = accept(sock, 0, 0);
		if (msgsock == -1)
			perror("accept");
		else do {
			bzero(buf, sizeof(buf));
			if ((rval = read(msgsock, buf, 1024)) < 0)
				perror("reading stream message");
			else if (rval == 0)
				printf("Ending connection\n");
			else
				printf("-->%s\n", buf);
		} while (rval > 0);
		close(msgsock);
	}
	/*
	 * The following statements are not executed, because they follow an
	 * infinite loop.  However, most ordinary programs will not run
	 * forever.  In the UNIX domain it is necessary to tell the file
	 * system that one is through using NAME.  In most programs one uses
	 * the call unlink() as below. Since the user will have to kill this
	 * program, it will be necessary to remove the name by a command from
	 * the shell.
	 */
	close(sock);
	unlink(NAME);
}
```

Figure 9b Accepting a UNIX domain stream connection

```
/*
 * The variable descriptor may be the descriptor of either a file
 * or of a socket.
 */
cc = read(descriptor, buf, nbytes)
int cc, descriptor; char *buf; int nbytes;

/*
 * An iovec can include several source buffers.
 */
cc = readv(descriptor, iov, iovcnt)
int cc, descriptor; struct iovec *iov; int iovcnt;

cc = write(descriptor, buf, nbytes)
int cc, descriptor; char *buf; int nbytes;

cc = writev(descriptor, iovec, ioveclen)
int cc, descriptor; struct iovec *iovec; int ioveclen;

/*
 * The variable ‘‘sock’’ must be the descriptor of a socket.
 * Flags may include MSG_OOB and MSG_PEEK.
 */
cc = send(sock, msg, len, flags)
int cc, sock; char *msg; int len, flags;

cc = sendto(sock, msg, len, flags, to, tolen)
int cc, sock; char *msg; int len, flags;
struct sockaddr *to; int tolen;

cc = sendmsg(sock, msg, flags)
int cc, sock; struct msghdr msg[]; int flags;

cc = recv(sock, buf, len, flags)
int cc, sock; char *buf; int len, flags;

cc = recvfrom(sock, buf, len, flags, from, fromlen)
int cc, sock; char *buf; int len, flags;
struct sockaddr *from; int *fromlen;

cc = recvmsg(sock, msg, flags)
int cc, socket; struct msghdr msg[]; int flags;
```

Figure 10 Varieties of read and write commands