CONVEX Interprocess Communication (IPC) Programming Guide

Order No. DSW-143

Second Edition, Rev. 1
November 1990

CONVEX Computer Corporation
Richardson, Texas USA

CONVEX
Interprocess Communication (IPC)
Programming Guide

Order No. DSW-143

Printed in the United States of America

Revision Information for

CONVEX
Interprocess Communication (IPC)
Programming Guide

Edition	Document No.	Description
Second, Rev. 1	710-002630-202	Released with CONVEX Internet Services V9.0, November 1990. The manual was converted to the new CONVEX document format.
Second	710-002630-201	Released with CONVEX UltraNet Interface V1.0 and ConvexOS V8.0, December 1989.
First	710-002630-200	Released with Version 6.1 of the CONVEX operating system, November 1988.

Table of Contents

CONVEX Interprocess Communication (IPC) Programming Guide

Intermediate Socket Programming

Advanced Socket Programming

Appendix A: Reporting Problems

Using This Guide

Purpose and Audience

The *CONVEX Interprocess Communication (IPC) Programming Guide* provides experienced C language programmers with information needed to implement pipes, shared memory, sockets, and other Interprocess Communication (IPC) facilities. Specific prerequisites to use of this book include both familiarity with C and significant programming experience in a ConvexOS environment.

Organization

This guide is organized as follows:

- Chapter 1, "Introduction," discusses IPC programming in general terms and provides brief descriptions of each of the primary IPC facilities.
- Chapter 2, "Using Pipes for IPC Programming," explains the use of pipes, socket pairs, and named pipes.
- Chapter 3, "Using Shared Memory for Local Applications," explains the use of shared memory.
- Chapter 4, "Introductory Socket Programming," introduces socket programming and describes the basic use of network library routines.
- Chapter 5, "Intermediate Socket Programming," describes intermediate socket programming techniques, including how to create datagrams and establish connections.
- Chapter 6, "Advanced Socket Programming," discusses a variety of advanced socket programming techniques, including the use of connectionless sockets and the Internet "superserver" daemon, `inetd`.
- Appendix A, "Reporting Problems," provides instructions for reporting software or documentation problems to the CONVEX Technical Assistance Center (TAC).

Technical Assistance

If you have questions that are not answered in this book, contact the CONVEX Technical Assistance Center (TAC).

- Within the continental U.S., call 1(800)952-0379.
- From Canada, call 1(800)345-2384.
- All other locations, contact the local CONVEX office.

Associated Documents

Using this software may require information not specific to the tasks described in this document.

CONVEX Computer Corporation provides the following documents to help you with the ConvexOS operating system:

- *CONVEX UNIX Primer* (DSW-133). This book introduces new users to the ConvexOS operating system.
- *ConvexOS Programmer's Reference* (DSW-332). This book is the standard reference for the ConvexOS operating system. It contains man pages describing ConvexOS and CONVEX Internet Services software.

CONVEX Computer Corporation provides the following documents to help you with CONVEX Internet Services:

- *CONVEX Networking Concepts* (DSW-128). This book provides an overview of all CONVEX networking products.
- *CONVEX Internet Services User's Guide* (DSW-141). This guide explains how to use CONVEX Internet Services to get work done on a day-to-day basis. In particular, it describes network utilities that enable you to log in to remote systems, execute commands on different machines, and transfer files across the network.
- *CONVEX Internet Services System Manager's Guide* (DSW-142). This guide provides system managers with information needed to configure and maintain a CONVEX TCP/IP network.

Ordering Documentation

To order the current edition of this or any other CONVEX document, send requests to:

CONVEX Computer Corporation
Customer Service
P.O. Box 833851
Richardson, TX 75083-3851 USA

Include the order number or the exact title, as listed on the front cover.

Notational Conventions

This section discusses notational conventions used in this guide.

Command Syntax

Consider this example:

```
COMMAND input_file [...] {a | b} [output_file]
   ①        ②       ③     ④          ⑤
```

1. **COMMAND** must be typed as it appears.
2. *input_file* indicates a file name that must be supplied by the user.
3. The horizontal ellipsis in brackets indicates that additional input file names may be supplied.
4. Either `a` or `b` must be supplied.
5. *output_file* indicates an optional file name.

General Conventions

In general, the following conventions are used in this guide:

- **`Bold constant-width font`** identifies user input in examples.
- *Italics*
  - Designate user-supplied variables in a command-line example.
  - Introduce new and important terms.
  - Identify variables in mathematical equations.
  - Indicate titles of documents.
- `Constant-width font` is used to designate input and output, including:
  - Command names and options.
  - System calls.
  - Data structures and types.
  - Directives, program statements, display examples, printout examples, and error messages.
- Horizontal ellipsis (...) shows repetition of the preceding item(s).
- Vertical ellipses show that lines of code have been left out of an example.
- Words and abbreviations that indicate keyboard keys you press are identified in a distinctive bold type. For example, **RETURN** refers to the carriage return key. Words separated by a hyphen indicate two keys that you must press simultaneously. For example, **CTRL-X** indicates that you must press and hold down the **CTRL** key and then press the **X** key.
- The word "enter" in a phrase such as "enter `ls`" means that you type the command and then press **RETURN**.
- References to the *ConvexOS Programmer's Reference* appear in the form `adb(1)`, where the name of the man page is followed by its section number enclosed in parentheses.

❑ The ConvexOS prompt is printed as a percent sign (%).

Note

A Note highlights supplemental information or information that may be of particular interest.

Reader Response

If you have comments or questions about the contents of this book, please notify the CONVEX documentation department by using the Reader's Comment Form at the end of this document.

Introduction

This guide explains how to use pipes, sockets, and shared memory for interprocess communication. To understand how you can use these facilities, you must first understand the essentials of the CONVEX Internet Services networking environment and why you might choose to use IPC facilities. This chapter provides background information and describes briefly the following IPC facilities:

- Signals.
- Process tracing.
- Shared file descriptors.
- Pipes.
- Shared memory.
- Sockets.

Some of these facilities—signals, process tracing, and shared file descriptors—have limited usefulness for IPC programming and are not discussed beyond this introduction. The balance of this guide discusses the remaining facilities.

Note

Network File System (NFS) is an optional networking product that enables users to share file systems over a network. If you are interested in using NFS, please contact your CONVEX sales representative or refer to the *CONVEX Network File System User's Guide* for an introduction to the product.

Introduction to IPC Programming

IPC programming is the development of programs that use IPC facilities to communicate with one another. Generally speaking, IPC facilities are used for one of two reasons. First, you must use IPC facilities if you are building an application that involves more than one process. Examples of these types of applications are distributed systems and multi-user programs like `talk`. The second reason for using IPC facilities is to communicate with a system whose interface is an IPC mechanism. Examples of this type of application are piping data through a filter, or connecting to a server on a network.

The operating system facilitates and manages communication between processes. IPC programs interact with the operating system in much the same way as other programs do—through library routines and system calls. This book explains in great detail system calls and library routines needed to use a given IPC facility. Table 1-1 on the following page lists the most important system calls and library routines described in subsequent chapters.

Table 1-1
System Calls and Library Routines Used for IPC Programming

Call or Routine	Use	Chapter
`pipe, dup, dup2`	Pipes	2
`socketpair`	Bidirectional pipes	2
`mmap, munmap`	Shared memory	3
`socket, bind, connect, read, write, close, accept`	Sockets	4
`getprotoent, getservent, gethostent`	Sockets	4
`getsockopt, setsockopt`	Sockets	6

Basic IPC Facilities

The following paragraphs briefly describe basic IPC facilities. Pipes, shared memory, and sockets are discussed in greater detail in subsequent chapters.

Signals

Signals are analogous to interrupts, but unlike interrupts, they are received by user processes. Signals are sent either as a result of various exceptions or I/O events, or explicitly by other user processes. Signals are useful only for extraordinarily simple forms of IPC; for more complex applications, sockets are a better choice.

Process Tracing

Process tracing is a limited form of IPC used for debugging with `adb` and `csd`. Process tracing enables parent processes to examine and modify memory and registers owned by their children's processes. You can also use process tracing to set breakpoints, enable single-step execution of child processes, or send signals to a child process. Child processes, however, have no control over parent processes via process tracing. For more information about process tracing, refer to the `ptrace`(2) man page.

Pipes and Shared File Descriptors

In the simplest sense, a *pipe* is merely a pair of file descriptors set up so that one descriptor can read whatever is written to the other. A single process initially owns both descriptors. This process can pass the descriptors to its children, but the processes that share a pipe must have a common ancestor.

At the application level, you can use any of three types of pipes. The simplest pipes are unidirectional: data is written at one end and read from the other. *Bidirectional pipes*, or *socketpairs*, enable you to set up two-way stream communication between processes. *Named pipes*, unlike "ordinary" pipes, exist permanently in the file system with directory entries and pathnames. Because you can access these pipes by name, you can use them for a variety of applications that cannot be accomplished with ordinary pipes.

Shared Memory

Shared memory is used to map files or memory into the address space of processes. Mapped regions have any combination of read, write, and execute permissions. The use of shared memory is described in detail in Chapter 3, "Using Shared Memory for Local Applications."

Sockets

Sockets are communication endpoints supported by the operating system. Each socket has the potential to exchange information with other sockets of an appropriate type within a domain. (A *domain* is an address space shared by sockets.) ConvexOS supports two domains: the "UNIX" domain, AF_UNIX, used for communication within the local system; and the "Internet" domain, AF_INET, used for communication between hosts on the Internet.

The following three basic types of sockets exist:

- **Stream sockets**—Provide for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries.
- **Datagram sockets**—Support a bidirectional flow of data that is not promised to be sequenced, reliable, or unduplicated.
- **Raw sockets**—Provide users with access to the underlying communication protocols that support sockets.

Chapters 4, 5, and 6 describe the use of sockets in detail.

Using Pipes for IPC Programming

2

Beginning with an explanation of basic terms and concepts, this chapter describes the use of pipes, socketpairs (bidirectional pipes), and named pipes. Example programs illustrating detailed implementation are used throughout the chapter.

Simple Pipes

Most ConvexOS users know that they can send the output of one program to the input of another by entering a command similar to:

```
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 enters a command, the shell reads the command and 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 exits. When the forked process exits, the shell prompts for the next command. If the command is a compound command, for example:

```
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 written into one socket can be read from the other.

Because a program specifies its input and output only by descriptor table indices that appear as variables or constants, the input source and output destination can be changed without changing the text of the program. In this way, the shell sets up pipes. Before executing `prog1`, the process closes whatever is at `stdout` and replaces it with one end of a pipe. Similarly, the process that executes `prog2` substitutes the opposite end of the pipe for `stdin`.

Figure 2-1 on the following page contains a program that creates a pipe for communication between itself and its child process. The parent process creates a pipe, and then forks a child process. When the process forks, the parent's descriptor table is copied into the child's descriptor table.

Figure 2-1
Use of a Pipe

```
#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 communication
* device. It 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]);
        }
}
```

In this example, the parent process makes a call to the system routine, `pipe`. `pipe` then creates the pipe and places the descriptors used for the two ends of the pipe into the descriptor table. The user code passes an array to `pipe`, into which it places the index numbers of the sockets it created. In turn, `pipe` returns two file descriptors, one for each end of the pipe. The two ends are not equivalent. The first element of the array contains a descriptor for reading the pipe, whereas the second element contains a descriptor for writing the pipe.

After creating the pipe, the parent process calls `fork` to create the child process with which it will share the pipe. Figure 2-2 illustrates the effect of a fork. Before the fork, the parent process' 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 process then uses the pipe to send a message to the parent process.

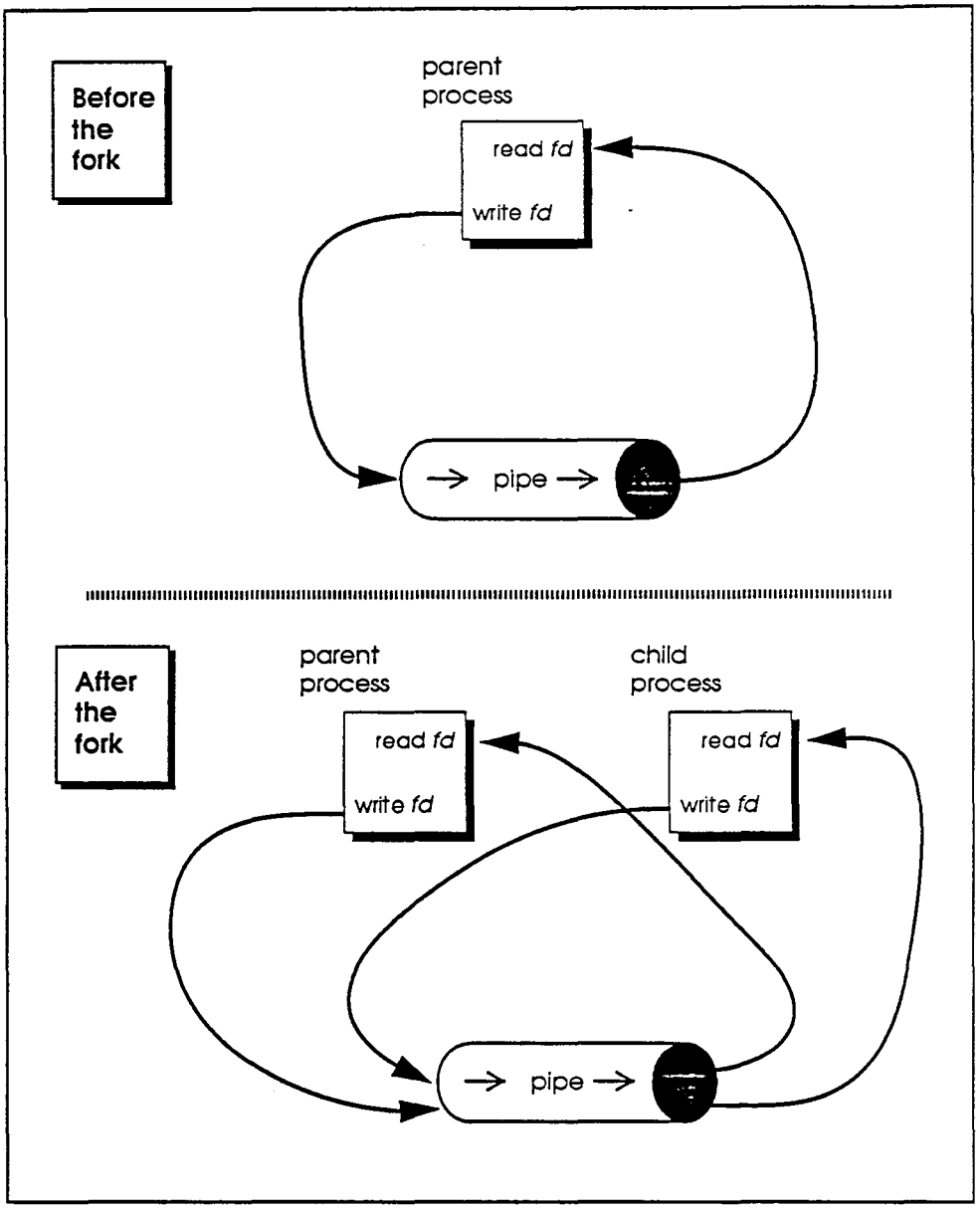


Figure 2-2
Sharing a Pipe Between Parent and Child

Remember that a pipe 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 whether 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 is possible (because both processes have references to both ends), but very complicated. Parent-child synchronization is often easier if you use two pipes, one for use in each direction.

Both parent and child in the example in Figure 2-1 close the end of the pipe that they do not use. This is a good practice to emulate. Consider these two examples. First, suppose a reader leaves the writing end of a pipe open. In this case, the pipe never returns an EOF. So, when the writer exits, the reader hangs, waiting to read data that is never written. Similarly, if a writer leaves the reading end of a pipe open, there is a strong possibility that the writer will eventually hang. If the reader exits or is killed, the writer eventually fills the pipe and hangs, because the kernel assumes the writer plans to read the data in the pipe. If, on the other hand, all processes close the read end, the writer receives a SIGPIPE signal the next time it tries to write. For these reasons, it is a good idea to close any unused descriptors, even though nothing requires you to do so.

A pipe is also a stream communication mechanism; 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, as many bytes as are available are returned, up to the amount of the request. These bytes may have come from a single call to `write` or from several calls to `write` that were concatenated.

You can establish piped I/O through the use of dup and dup2, which enable you to duplicate existing file descriptors.

dup2 takes two arguments, *oldd* and *newd*. When you call it, dup2 closes *newd*, and then creates a duplicate of *oldd*. The duplicate descriptor's index is *newd*.

dup takes just one argument, *oldd*. *newd* is chosen automatically by ConvexOS and is the lowest-numbered descriptor that is not in use. So, for example, if a program closes its standard input (descriptor 0), and then calls dup, the duplicate descriptor has index 0.

The following command, for example:

```
dup2(fdpair[1], 1)
```

makes file descriptor 1 (`stdout`) a duplicate of the existing file descriptor, `fdpair`[1]. This enables you to write to the pipe with the file descriptor `fdpair`[1].

The example in Figure 2-3 on the following page shows two different uses of dup and dup2. In the first case, with `ls_child`, dup2 is used to set up the writing end of the pipe for ls. Later, in `sort_child`, dup is used to redirect standard input from the sort.

Figure 2-3
Using dup and dup2

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

int fdpair[2];

/*
 * This program demonstrates using dup() and dup2() to do I/O
 * redirection. The program creates a pipe, then spawns two
 * children that connect the ends of the pipe to their
 * standard input and output, then invokes standard utilities
 * using execle(3). The first child invokes ls, which writes
 * on its standard output. The second child invokes sort -r,
 * which reverses the order of ls's output.
 */

main()
{
        int pid0, pid1;
        union wait status;

        if (pipe(fdpair) != 0) {
                perror("pipe");
                exit(1);
        }
        pid0 = fork();
        if (pid0 == -1) {
                perror("first fork");
                exit(1);
        }
        if (pid0 == 0)
                ls_child();
        pid1 = fork();
        if (pid1 == -1) {
                perror("second fork");
                exit(1);
        }
        if (pid1 == 0)
                sort_child();

        close(fdpair[0]);
        close(fdpair[1]);
        while (wait(&status) != -1)
                if (status.w_status != 0)
                        exit(status.w_status);
        exit(0);
}

ls_child()
{
        if (dup2(fdpair[1], 1) != 1) {          /* stdout is automatically closed */
                perror("ls_child dup2");
                exit(1);
        }
```

Figure 2-3
Using dup and dup2 (continued)

```
        close(fdpair[0]);
        close(fdpair[1]);
        execle("/bin/ls", "ls", 0);
        perror("ls_ child execve");
        exit(1);
}

sort_child()
{
        close(0);
        if (dup(fdpair[0]) != 0) {    /* 0 is lowest-numbered fd not in use */
                perror("sort_child dup");
                exit(1);
        }
        close(fdpair[0]);
        close(fdpair[1]); /* must close this; else can't detect end-of-file */
        execle("/usr/bin/sort", "sort", "-r", 0);
        perror("sort_child execve");
        exit(1);
}
```

Bidirectional Pipes (Socketpairs)

As the previous discussion indicates, a pipe is a pair of connected sockets for one-way stream communication. However, you can obtain a pair of connected sockets for two-way stream communication by calling the routine `socketpair`. The program in Figure 2-4 on the following page calls `socketpair` to create such a connection. Figure 2-5 illustrates the result of a fork following a call to `socketpair`.

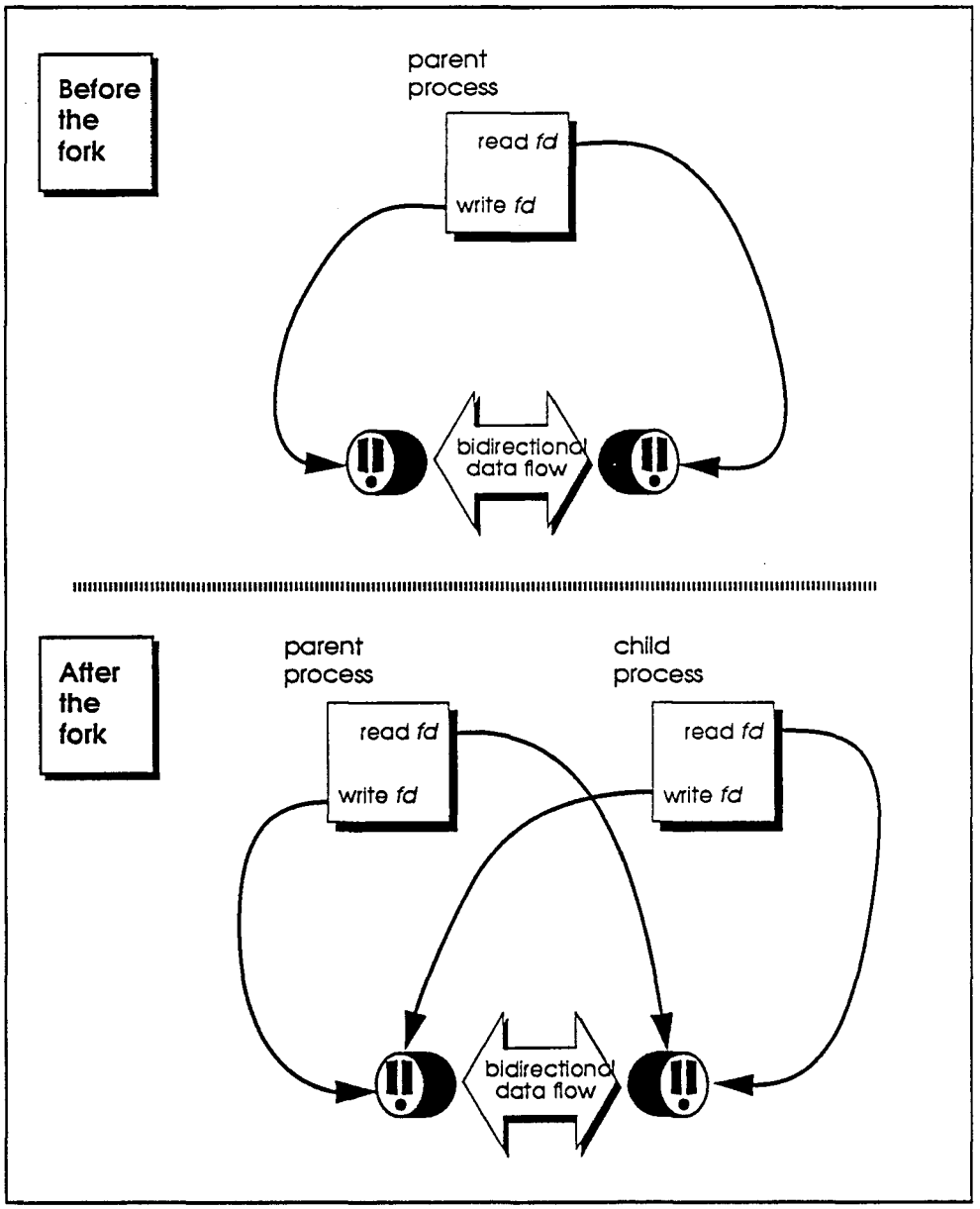
`socketpair` takes three arguments: domain specification, communication style, and protocol. These parameters are shown in Figure 2-5. (Domains and protocols are discussed in Chapter 4, "Introductory Socket Programming.") A *domain* is a space of names that may be bound to sockets and that implies certain other conventions. Currently, socketpairs have only been implemented for the UNIX domain. The UNIX domain uses ordinary file pathnames for naming sockets. It only allows communication between sockets on the same machine. Therefore, you cannot use socketpairs for networked (inter-machine) applications.

The header files <sys/socket.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.

Figure 2-4
Use of a socketpair

```
#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;
 * socketpairs, however, are two-way communication objects. Therefore it 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 2-5
Sharing a `socketpair` Between Parent and Child



Named Pipes

Software supporting named pipes is distributed to ConvexOS licensees. Named pipes are "ordinary" pipes that exist permanently in the file system with directory entries and pathnames. Because you access these pipes by name, they can be used for applications for which unnamed pipes are not suited. Typically, named pipes are used to allow a number of processes to communicate with a daemon process.

CONVEX named pipes can only be used locally. Even if the named pipe exists as an inode on a remote machine, data written to the pipe via NFS is only accessible to processes on the same client. Therefore, even though multiple clients mount the same file system, they have separate streams associated with any named pipe in that file system.

You create a named pipe using either mknod(2) or mknod(8). Once you have created the pipe, you can use open, read, and write. Figures 2-6 and 2-7 contain a simple named pipe application. For more information, refer to the mknod(2), mknod(8), and open(2) man pages.

Figure 2-6
Using Named Pipes for Writing Data

```
/*
 * writer - create a named pipe, open it for writing,
 *      then copy standard input to the pipe.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>              /* get definition of S_IFIFO */

#define PIPE_NAME "Frederick"

main()
{
        FILE *f;
        int c;

        (void) unlink(PIPE_NAME);
        if(mknod(PIPE_NAME, 0666 | S_IFIFO, 0) != 0) {
                perror("mknod");
                exit(1);
        }
        f = fopen(PIPE_NAME, "w");
        if (f == NULL) {
                perror("open");
                exit(1);
        }
        while ((c = getchar()) != EOF)
                putc(c, f);
}
```

Figure 2-7
Using Named Pipes for Reading Data

```
/*
 * reader - open a named pipe for reading, and copy
 *      data from pipe to standard output
 */

#include <stdio.h>
#include <sys/fcntl.h>                          /* get definition of O_RDONLY */

#define PIPE_NAME "Frederick"

main()
{
        FILE *f = fopen(PIPE_NAME, "r");
        char c;

        if (f == 0) {
                perror("fopen");
                exit(1);
        }
        while ((c = getc(f)) != EOF)
                putchar(c);
}
```

Using Shared Memory for Local Applications

3

Introduction

Shared memory is a mechanism that allows one or more processes to share portions of their address space with other processes in a selected group. Using shared memory offers several advantages over other IPC mechanisms. First, processes using shared memory communicate faster than processes using pipes or sockets, because the processes may share variables. Second, shared memory enables you to send I/O to or from a regular file by mapping the file directly into the process' address space; no `read` or `write` calls are required. This can make accesses to a file easier to code, or more efficient.

It is possible to obtain segments of memory that are not part of the standard text, data, or stack segments. Processes may have text, data, stack, and up to 32 shared memory segments. Because you set the protection bits for each segment, you can create segments that are executable, or have other interesting properties.

This chapter illustrates the use of shared memory through a series of increasingly complex examples. The first two examples, Figures 3-1 and 3-2, demonstrate the use of shared memory in reading and copying files. Later examples, Figures 3-3 through 3-6, demonstrate more complicated uses of shared memory, including the use of shared memory with cooperating processes and the implementation of asynchronous I/O with user processes. The chapter closes with a brief description of approaches to mapping kernel and physical memory.

Although this chapter should provide an excellent introduction to the use of shared memory, you are likely to need more detailed information as you design applications. The following pages in the *ConvexOS Programmer's Reference* should be helpful:

`fsync(2)`	`mclear(3)`	`mmap(2)`	`mset(3)`	`msleep(2)`
`msync(2)`	`munmap(2)`	`mwakeup(2)`	`tas(3)`	

Mapping Files

As Figure 3-1 shows, you can use the mmap system call to map a file into the process' address space. In this example, mmap is used to print the sum of all the bytes in the /vmunix file.

Figure 3-1
Mapping Files Into Process Address Space

```
#include <sys/file.h>                          /* include file for file I/O */
#include <sys/mman.h>                          /* include shared memory definitions */

main()
{
        char *Seg;
        int fd, i;
        long sum, len;

        /* open the file for reading */
        fd = open("/vmunix", O_RDONLY);
        if (fd < 0) {
                perror("Can't open /vmunix");
                return(2);
        }

        /* map the file into the process' address space at an o/s defined place */

        len = 0;
        Seg = mmap(0, &len, PROT_READ, MAP_FILE, fd, 0);
        if ((int) Seg == -1) {
                perror("Can't mmap");
                return(2);
        }

        /* sum the bytes in the file */

        for (sum = i = 0; i < len; i++)
                sum += Seg[i];
        printf("sum is %d\n", sum);
        return(0);
}
```

Calls to mmap require six arguments, as in the statement:

```
Seg = mmap(0, &len, PROT_READ, MAP_FILE, fd, 0);
```

The syntax of the mmap system call is as follows:

Segment = **mmap**(*addr, len, prot, share, fd, offset*)

Arguments should be supplied in the order in which they are described here. (For more information, refer to the mmap(2) man page in the *ConvexOS Programmer's Reference.*)

addr Virtual address at which the shared memory segment is mapped. If you specify an address of zero (as it is in this example), the system chooses an address.

len Address of the length of the shared segment. If you specify a length of zero, the system sets the length of the segment equal to the length of the mapped file rounded up to the next page boundary. mmap returns the actual length of the mapped file.

prot Protection to be given to each mapped page. You set *prot* to the logical OR of the three types of accesses: PROT_READ, PROT_WRITE, and PROT_EXEC, which signify read, write, and execute access, respectively.

share Options that control the sharing of memory. Refer to the mmap(2) man page for a complete list of options.

fd File descriptor that identifies the desired shared memory object. In Figure 3-1, *fd* specifies the file to be mapped into memory.

offset Byte offset in the file that corresponds to the first mapped address. Zero specifies that the map begins with the first byte of the file. (*offset* is interpreted like the seek system call, and is always referenced relative to the beginning of the file.)

mmap returns a character pointer to the beginning of the shared segment in your virtual address space. You access bytes in the shared memory segment by using this pointer, just as if malloc had returned it. If two concurrent sum programs were run, both would map the same physical pages into their address space.

If mmap detects an error or is unable to complete the mapping for some reason, it returns the integer value -1, with errno set to identify the error (refer to the mmap(2) man page). Some error numbers, such as EINVAL, may mean that any of several things is wrong. If you OR MAP_DEBUG into the share argument (e.g., MAP_FILE | MAP_DEBUG), the system prints a message to your terminal to tell you specifically which of its validity tests is failing.

Using shared memory to do sequential I/O, as in Figure 3-1, is likely to be less efficient than using a medium size buffer and `read` or `write` calls. Shared memory is less efficient because page faults are relatively expensive to process, compared with copying the data from a kernel file-system buffer. Additionally, the file system maintains a large buffer cache and anticipates sequential read operations, reading the data before the user program actually requests it. Page faults do not currently anticipate future sequential faults, although the system attempts to read or write groups of pages equal to the file-system block size. Random read accesses, on the other hand, do not benefit from file system "read-ahead," and mapped files avoid the CPU overhead of copying the data to or from the file-system buffer cache. Also, user programs that make extensive use of random access may be considerably easier to code, because of the elimination of buffer-management considerations and `seek` calls.

You are not restricted to reading files when you use shared memory; Figure 3-2 on the following page shows a simple copy program that uses mapped files to replace copying altogether. The file being copied to is mapped into the copy program's address space, and a `read` system call transfers the bytes directly from the source file to the destination file. The map address and length are explicitly specified to `mmap`.

Figure 3-2
copy Program

```
#include <sys/param.h>
#include <sys/file.h>
#include <sys/stat.h>
#include <sys/mman.h>

main(argc, argv)
char **argv;
{
        char *Seg;
        int retlen, fdfrom, fdto;
        char *from, *to;
        struct stat statbuf;

        if (argc != 3) {
                printf("usage: copy from to \n");
                return (2);
        }
        from = argv[1];
        to = argv[2];
        fdfrom = open(from, O_RDONLY);              /* open source file */
        if (fdfrom < 0) {
                perror(from);
                exit(2);
        }
        if (fstat(fdfrom, &statbuf) < 0) {          /* get length */
                perror("stat");
                exit(2);
        }
        /* open destination file (creating and truncating if necessary) */
        fdto = open(to, O_WRONLY | O_CREAT | O_TRUNC, statbuf.st_mode);
        if (fdto < 0) {
                printf("Can't open %s", to);
                exit(2);
        }

        /* map destination file into memory */
        retlen = roundup(statbuf.st_size, NBPG);
        Seg = mmap(0xc0000000, &retlen, PROT_WRITE,
                MAP_EXTEND | MAP_FILE | MAP_SEQUENTIAL | MAP_DEBUG, fdto, 0);
        if ((int) Seg == -1) {
                perror("Can't mmap");
                exit(2);
        }

        /* read, copy, and write in one operation */
        if (read(fdfrom, Seg, statbuf.st_size) != statbuf.st_size) {
                perror("read");
                exit(2);
        }
        /* truncate destination file to source file size */
        munmap(Seg, 0);
        (void) ftruncate(fdto, statbuf.st_size);
        return (0);
}
```

Because mapping only occurs in whole pages, `copy` uses `ftruncate` explicitly to truncate the destination file to the length of the source file (the length of the source file is in `statbuf.st_size`).

The `munmap` call is used to unmap shared memory segments. `munmap` has two arguments: *addr* and *size*. *addr* is any address in the shared memory segment that is to be unmapped. The system determines which shared memory segment is to be unmapped by checking this address. *size* is normally zero; if it is nonzero, it represents a request to unmap the partial segment described. Partial segment unmapping is not currently supported.

Sharing Memory

The previous examples have shown only a restricted set of applications: single processes making use of shared memory facilities. However, shared memory is also useful with multiple processes. Program `ring` in Figure 3-3 shows a set of cooperating processes that synchronize using shared memory and counting semaphores. The counting semaphores are implemented by the functions `P` and `V`. `P` acquires a resource by decrementing the count of available resources. If no resources are available (the count was zero or less), `P` sleeps until resources are available. `V` releases resources by incrementing the count.

`P` is implemented using the library functions `mset`, `mclear`, `msleep`, and `mwakeup`. These functions work on binary semaphores, or *locks*, and are supported in part by the operating system.

`mset` Attempts to lock the binary semaphore. If it is successful, i.e., the lock was not previously locked, `mset` returns a value of 1. If `mset` cannot immediately acquire the lock, and the value of its second argument is 1, `mset` waits, by suspending the process, until the lock is available, then it acquires the lock. On the other hand, if the value of the second argument passed to `mset` is 0, and it cannot acquire the lock, `mset` returns a value of 0 to indicate failure. In this case, the caller must decide when to retry the lock. `mset` does not make a system call unless lock acquisition fails and the `wait` (second) argument is 1.

`mclear` Releases a binary semaphore acquired with `mset`, awakening any callers that are suspended and awaiting acquisition of the same lock. `mclear` does not make a system call unless other processes are waiting to acquire the same lock.

`msleep` Suspends the calling process until a binary semaphore is unlocked. If the semaphore is released before the system call can be processed, `msleep` returns immediately. Otherwise, `msleep` returns when another process issues an `mwakeup` on the semaphore.

`mwakeup` Awakens all suspended processes that are awaiting the same binary semaphore.

The implementation of `P` and `V` in the `ring` program makes use of two binary semaphores: one that locks the counting semaphore structure, and one that indicates a process awaiting the value of the counting semaphore to become > 0. If the value of the counting semaphore is not > 0, `P` carefully sets the waiting semaphore while the lock semaphore is set, and then releases the lock semaphore and sleeps on the waiting semaphore. `V` awakens processes suspended on the waiting semaphore if the counting semaphore value goes from 0 to 1.

The `ring` program uses a counting semaphore to indicate to each cooperating process that it is that process' turn to handle the data. Each process awaits its own semaphore with `P`, then processes the data by checking and then setting `Seg->mynum` against its own process number. The process then tells the next process to proceed by executing a `mynum` on the next process' semaphore. Thus the processes implement a token-ring protocol.

The `runring` program shown in Figure 3-4 is used to fork an arbitrary number of `ring` processes.

Figure 3-3
`ring`: Shared Memory Semaphores

```
/*
 * ring protocol of shared memory
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/file.h>
#include <sys/mman.h>

struct counting_semaphore {
        semaphore lock, waiting;
        int value;
};
typedef struct counting_semaphore counting_semaphore;

P(s)
        counting_semaphore *s;
{
        while (mset(&s->lock, 1) && s->value <= 0) {
                s->waiting.lock = 1;
                mclear(&s->lock);
                msleep(&s->waiting);
        }
        s->value--;
        mclear(&s->lock);
}

V(s)
        counting_semaphore *s;
{
        int i;

        mset(&s->lock, 1);
        i = s->value++;
        mclear(&s->lock);
        if (i == 0)
                mclear(&s->waiting);
}

#define MAXPROC 100

struct seg {
        int mynum;
        counting_semaphore csem[MAXPROC];
} *Seg;
```

Figure 3-3
ring: Shared Memory Semaphores (continued)

```
        main(argc, argv)                    /* ring - one of the ring processes */
        char **argv;
{
        int retlen, fd;
        int i;
        int total, mynum;

        if (argc != 3) {
                printf("r total mynum \n");
                exit(2);
        }
        total = atoi(argv[1]);
        mynum = atoi(argv[2]);
        mynum = mynum % total;
        printf("total %d mynum %d\n", total, mynum);
        if (mynum >= MAXPROC) {
                printf("too many processes\n");
                exit(2);
        }

        fd = open ("xxx", O_RDWR | O_CREAT, 0666);
        if (fd < 0) {
                perror("Can't open xxx");
                exit(2);
        }
        retlen = sizeof(struct seg);
        Seg = (struct seg *) mmap(0xc0000000, &retlen, PROT_READ | PROT_WRITE,
                MAP_FILE | MAP_EXTEND | MAP_SHARED, fd, 0);
        if ((int) Seg == -1) {
                perror("Can't mmap");
                exit(2);
        }
        if (mynum == 0) {
                V(&Seg->csem[mynum]);
                Seg->mynum = total - 1;
        }
        for (i = 0; i < 10000; i++) {
                P(&Seg->csem[mynum]);
                if (((Seg->mynum+1) % total) != mynum) {
                        printf("not my turn! %d %d\n", Seg->mynum, mynum);
                        abort();
                }
                Seg->mynum = mynum;
                V(&Seg->csem[(mynum+1) % total]);
                if (i % 1000 == 999) {
                        printf(".");
                        (void) fflush(stdout);
                }
        }
        printf("%d exiting\n", getpid());
        if (mynum == 0)
                (void) unlink("xxx");
        return (0);
}
```

Figure 3-4
runring: Forking ring Processes

```
/*
 * runring -- forks ring processes
 */

char buf[100];

main(argc, argv)
        char **argv;
{
        register int i;
        int number = atoi(argv[1]);

        for (i = 0; i < number; i++) {
                if (i == (number - 1))
                        sprintf(buf, "ring %d %d", number, i);
                else
                        sprintf(buf, "ring %d %d &", number, i);
                system(buf);
        }
        return (0);
}
```

Larger Examples

The program shown in Figure 3-5 on the following page implements asynchronous I/O using user processes. Program `async` forks a child process that reads a portion of the /vmunix file into a buffer in shared memory. The parent process then sums the bytes in that buffer, while the child process fills an alternate buffer.

The two processes are synchronized using four binary semaphores, two for each of the two buffers. One of the semaphores for each buffer is used to signal that reading is complete and the buffer is full; the other is used to signal that summing is complete and the buffer is empty. This signaling mechanism is actually implemented by clearing the semaphore lock.

`async` is our first example that uses the `MAP_ANON` (map anonymous memory) domain option. Pages mapped in the anonymous domain are initially zero-filled, and are shared among all the processes that use `mmap` with an `fd` representing the same file. This mode provides a convenient way to implement shared memory when the state or contents of the memory need not be kept permanently, for example, for scratch or temporary work.

Figure 3-5
async: Asynchronous I/O Using User Processes

```
/* asynchronous I/O demonstration */

#include <sys/file.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/mman.h>

#define BUFSIZE 65536

struct seg {
        semaphore reading[2], summing[2];
        char eof[2];
        char buf1[BUFSIZE];
        char buf2[BUFSIZE];
};

int summer, reader;

main()
{
        char *buf1, *buf2;
        struct seg *Seg;
        int fd, fd1, i;
        long sum = 0;
        int len;

        /* open the file for reading */
        fd = open("/vmunix", O_RDONLY);
        if (fd < 0) {
                perror("Can't open /vmunix");
                return (2);
        }

        fd1 = open("async.tmp", O_CREAT | O_RDWR | O_TRUNC, 0666);

        /* map anonymous space into our address space at an o/s defined place */
        i = sizeof (struct seg);
        Seg = (struct seg *) mmap(0, &i, PROT_READ | PROT_WRITE,
                MAP_ANON | MAP_SEQUENTIAL | MAP_DEBUG, fd1, 0);
        if ((int) Seg == -1) {
                perror("Can't mmap");
                return (2);
        }

        buf1 = Seg->buf1;
        buf2 = Seg->buf2;
        Seg->reading[0].lock = 1;              /* put in reading state */
        Seg->summing[0].lock = 1;              /* put in reading state */
        reader = summer = 0;                   /* start with buffer 0 */
        if (fork() == 0) {                    /* if child */
        char *buf;

        buf = (reader == 0) ? buf1 : buf2;
```

Figure 3-5
async: Asynchronous I/O Using User Processes (continued)

```
while ((len = read(fd, buf, BUFSIZE)) != 0) {
                        if (len != BUFSIZE)
                                while (len < BUFSIZE)
                                        buf[len++] = 0;

                        /* indicate finished reading */
                        mclear(&Seg->reading[reader]);

                        reader ^= 1;            /* switch buffers */
                        buf = (reader == 0) ? buf1 : buf2;

                        /* wait on summing to finish */
                        mset(&Seg->summing[reader], 1);
                }
                Seg->eof[reader] = 1;
                mclear(&Seg->reading[reader]);
                return (0);
        }
        /* parent */
        while (mset(&Seg->reading[summer], 1) && !Seg->eof[summer]) {
                char *buf;

                buf = (summer == 0) ? buf1 : buf2;
                for (i = 0; i < BUFSIZE; i++)
                        sum += buf[i];
                mclear(&Seg->summing[summer]);
                summer ^= 1; /* switch buffers */
        }
        printf("sum is %d\n", sum);
        return (0);
}
```

`mapt`, the program shown in Figure 3-6 on the following page, is designed to stress the virtual memory system. It forks an arbitrary number of processes that all write and verify the contents of a shared memory segment in either domain.

Figure 3-6
mapt: Stressing the Virtual Memory System

```
/*
 * mapt -- stress the shared memory system
 */

#include <stdio.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/file.h>
#include <sys/mman.h>

#define MPROC 50 /* maximum number sharing processes */

int verbose;
int random;                /* address memory randomly */
int psize;                 /* the 'page' size */
int size;                  /* size of the mapped area */
int nproc;                 /* the number of processes in this test */
int forever, leave;        /* options */
char *filename;            /* the file name to be tested */
int pid;                   /* the process ID */
int me;                    /* my ID number */
int al;                    /* alternate ID number; my buddy to be checked */
int procid[MPROC];         /* process ID number */
int test;                  /* the type of test */
char *Map;                 /* the Map address */
int domain = MAP_FILE;
int fd;
int docore = 0;
int pat = 0x57;            /* a garbage pattern */
int Loop = 1, VLoop = 1;
int pagesize;

#define ltov(vaddr)  (Map+((vaddr)%psize)+(((((vaddr)/psize)*nproc)+me)*psize))
#define altov(vaddr) (Map+((vaddr)%psize)+(((((vaddr)/psize)*nproc)+al)*psize))

/* VARARGS1 */
error(a, b, c, d, e, f)
char *a;
{

        printf(a, b, c, d, e, f);
        if (docore)
                abort();
}

/* VARARGS1 */
fatal(a, b, c, d, e, f)
char *a;
{

        fprintf(stderr, a, b, c, d, e, f);
        exit(3);
}
```

Figure 3-6
mapt: Stressing the Virtual Memory System (continued)

```
missingarg(c)
{

        fatal("Option -%c missing argument.\n", c);
}

usage()
{
        printf("mapt [ options ]\n");
        printf("    where options are:\n");
        printf("    -f file       Use file name file\n");
        printf("    -p psize      Sets 'pagesize' to psize\n");
        printf("    -n nproc      Sets number of processes to nproc\n");
        printf("    -s size       Sets map size to size\n");
        printf("    -r            Address memory randomly\n");
        printf("    -t test       Select test number\n");
        printf("    -c            Core dump if error\n");
        printf("    -P            pattern Garbage pattern used to fill file.\n");
        printf("    -F            Repeat forever\n");
        printf("    -l            Leave output file (file domain)\n");
        printf("    -S            Use swap domain.\n");
        printf("    -v            Verbose output\n");
        printf("    -L niter      Number of times to run the test\n");
}

main(argc, argv)
        char **argv;
{

        char buf[90];
        register i;
        extern errno;
        int callsize;

        /* set up reasonable defaults */
        procid[0] = getpid();
        filename = buf;

        sprintf(buf, "xxx%d", procid[0]);
        nproc = 1;                                      /* defaults */
        size = 4 * 1024 * 1024;
        psize = 4096;
        pagesize = getpagesize();

        while (*++argv && **argv == '-') { /* while there are options */
                switch(*++*argv) {  /* switch on the option */
                case 'v':
                        verbose++;
                        break;

                case 'r':
                        random++;
                        break;
```

Figure 3-6
mapt: Stressing the Virtual Memory System (continued)

```
                case 'c':
                        docore++;
                        break;

                case 'F':
                        forever++;
                        break;

                case 'S':
                        domain = MAP_ANON;
                        break;

                case 'l':
                        leave++;
                        break;

                case 'p':
                        if (*++argv == NULL)
                                missingarg('p');
                        psize = atoi(*argv);
                        while (isdigit(**argv))
                                ++*argv;
                        if (**argv == 'k')
                                psize *= 1024;
                        if (psize < 64)
                                fatal("psize too small\n");
                        break;

                case 's':
                        if (*++argv == NULL)
                                missingarg('s');
                        size = atoi(*argv);
                        while (isdigit(**argv))
                                ++*argv;
                        if (**argv == 'k')
                                size *= 1024;
                        else if (**argv == 'm')
                                size *= 1024 * 1024;
                        break;

                case 'n':
                        if (*++argv == NULL)
                                missingarg('n');
                        nproc = atoi(*argv);
                        if (nproc > MPROC)
                                fatal("nproc too big\n");
                        break;

                case 'P':
                        if (*++argv == NULL)
                                missingarg('P');
                        pat = atoi(*argv);
                        break;
```

Figure 3-6
mapt: Stressing the Virtual Memory System (continued)

```
                case 't':
                        if (*++argv == NULL)
                                missingarg('t');
                        test = atoi(*argv);
                        break;

                case 'L':
                        if (*++argv == NULL)
                                missingarg('L');
                        Loop = atoi(*argv);
                        break;

                case 'f':
                        if (*++argv == NULL)
                                missingarg('f');
                        filename = *argv;
                        break;

                default:
                        printf("Unknown option -%c\n", **argv);
                        usage();
                        return (2);
                }
        }

        fd = open(filename, O_RDWR | O_CREAT | O_TRUNC, 0666);
        if (fd < 0)
                fatal("Can't open/create %s.\n", filename);
        callsize = size;
        Map = (char *) mmap(0, &callsize, PROT_WRITE | PROT_READ,
                domain | MAP_EXTEND, fd, 0);
        if (Map < 0 || errno)
                fatal("Can't map\n");

        for (i = 1; i < nproc; i++) {
                pid = fork();
                me = i;
                if (pid < 0)
                        fatal("Can't fork\n");
                if (pid == 0)
                        break;                  /* child */
                procid[i] = pid;                /* parent */
                me = 0;
        }
        al = (me + 1) % nproc;
        VLoop = Loop / 10;
        if (VLoop <= 0)
                VLoop = 1;

loop:
        if (verbose && ((Loop % VLoop) == 0 || Loop == 1))
                printf("Loop %d test %d size %d psize %d nproc %d me %d\n",
                        Loop, test, size, psize, nproc, me);
```

Figure 3-6
mapt: Stressing the Virtual Memory System (continued)

```
        switch(test) {
                case 0:
                        test0();
                        break;
                case 1:
                        test1();
                        break;
                case 2:
                        test2();
                        break;
                default:
                        fatal("unknown test %d\n", test);
        }
        if (forever)
                goto loop;
        if (--Loop)
                goto loop;
        munmap(Map, size);
        close(fd);
        if (!leave)
                (void) unlink(filename);
        return (0);
}

struct test0 {
        int me;             /* my process ID */
        int page;           /* my page ID */
};

wfill(a, n, d)
        register int *a;
        register int n, d;
{
        while (n--)
                *a++ = d;
}

wcheck(a, n, d)
        register int *a;
        register n, d;
{
        register int *b = a + n;

        for (; a < b; a++)
                if (*a != d)
                        error("wcheck wrote (0x%x)=0x%x read (0x%x)=0x%x\n",
                                a, d, a, *a);
}

test0()
{
        register i;
        char *addr;
```

Figure 3-6
mapt: Stressing the Virtual Memory System (continued)

```
        struct test0 *t;

        int ppp = size / psize / nproc;              /* pages per process */
        int nwords = (psize - sizeof (struct test0)) / sizeof (int);

        if (ppp < 1)
                fatal("too few pages per process\n");

        for (i = 0; i < ppp; i++) {
                addr = (char *) ltov(i * psize) + sizeof (struct test0);
                t = (struct test0 *) ltov(i * psize);
                t->me = me;
                t->page = i;
                wfill((int *) addr, nwords, pat);
        }

        for (i = 0; i < ppp; i++) {
                addr = (char *) ltov(i * psize) + sizeof (struct test0);
                t = (struct test0 *) ltov(i * psize);
                if (t->me != me || t->page != i)
                error("header: wrote %d/0x%x read %d/0x%x\n", me, i, t->me, t->page);
                wcheck((int *) addr, nwords, pat);
        }
}

#define WRITTEN 1
#define CHECKED 2

struct test1 {
        semaphore busy, wait1, wait2;
        char written;
        int me;                     /* my process ID */
        int page;                   /* my page ID */
};

test1()
{
        register i;
        char *addr;
        struct test1 *t;

        int ppp = (size - pagesize) / psize / nproc; /* pages per process */
        int nwords = (psize - sizeof (struct test1)) / sizeof (int);

        if (nproc < 2)
                fatal("test1 reqiures at least 2 processes\n");

        if (ppp < 1)
                fatal("too few pages per process\n");

        for (i = 0; i < ppp; i++) {
                addr = (char *) ltov(i * psize) + sizeof (struct test1);
                t = (struct test1 *) ltov(i * psize);
```

Figure 3-6
mapt: Stressing the Virtual Memory System (continued)

```
loop1:
                mset(&t->busy, 1);
                while (t->written) {        /* not checked but already written */
                        t->wait1.lock = 1;
                        mclear(&t->busy);
                        msleep(&t->wait1);
                        goto loop1;
                }

                t->me = me;
                t->page = i;
                wfill((int *) addr, nwords, pat);
                t->written = 1;
                mclear(&t->busy);
                mclear(&t->wait2);
        }

        for (i = 0; i < ppp; i++) {
                addr = (char *) altov(i * psize) + sizeof (struct test1);
                t = (struct test1 *) altov(i * psize);
loop2:
                mset(&t->busy, 1);
                while (t->written == 0) {       /* not written yet */
                        t->wait2.lock = 1;
                        mclear(&t->busy);
                        msleep(&t->wait2);
                        goto loop2;
                }

                if (t->me != al || t->page != i)
                        error("header: wrote %d/0x%x read %d/0x%x\n",
                                al, i, t->me, t->page);
                wcheck((int *) addr, nwords, pat);
                wfill((int *) addr, nwords, 0);
                t->written = 0;
                mclear(&t->busy);
                mclear(&t->wait1);
        }
}

struct counting_semaphore {
        semaphore lock, waiting;
        int value;
};

typedef struct counting_semaphore counting_semaphore;

P(s)

        counting_semaphore *s;
{
        while (mset(&s->lock, 1) && s->value <= 0) {
                s->waiting.lock = 1;
```

Figure 3-6
mapt: Stressing the Virtual Memory System (continued)

```
                mclear(&s->lock);
                msleep(&s->waiting);
        }
        s->value--;
        mclear(&s->lock);
}

V(s)
        counting_semaphore *s;
{
        int i;

        mset(&s->lock, 1);
        i = s->value++;
        mclear(&s->lock);
        if (i == 0)
                mclear(&s->waiting);
}

struct test2 {
        counting_semaphore written, read;
        char  initialized;
        int   me;                       /* my process ID */
        int   page;                     /* my page ID */
};

test2()
{
        register i;
        char *addr;
        struct test2 *t;

        int ppp = (size - pagesize) / psize / nproc; /* pages per process */
        int nwords = (psize - sizeof (struct test2)) / sizeof (int);

        if (nproc < 2)
                fatal("test2 requires at least 2 processes-");

        if (ppp < 1)
                fatal("too few pages per process-");

        for (i = 0; i < ppp; i++) {
                addr = (char *) ltov(i * psize) + sizeof (struct test2);
                t = (struct test2 *) ltov(i * psize);
                if (tas(&t->initialized) == 0) /* tas(3) */
                        V(&t->read);

                P(&t->read);
                t->me = me;
                t->page = i;
                wfill((int *) addr, nwords, pat);
                V(&t->written);
        }
```

Figure 3-6
mapt: Stressing the Virtual Memory System (continued)

```
        for (i = 0; i < ppp; i++) {
                addr = (char *) altov(i * psize) + sizeof (struct test2);
                t = (struct test2 *) altov(i * psize);
                P(&t->written);
                if (t->me != al || t->page != i)
                        error("header: wrote %d/0x%x read %d/0x%x\n",
                                al, i, t->me, t->page);
                wcheck((int *) addr, nwords, pat);
                V(&t->read);
        }
}
```

Mapping Kernel and Physical Memory

The program shown in Figure 3-7 displays the interrupt vectors by looking at the first few words of physical page 0. It illustrates the use of `MAP_DEVICE` on the device `/dev/mem` to look at physical memory.

Kernel virtual memory (`/dev/kmem`) also supports mapping, but be careful, because the kernel remaps pages (such as the buffer cache). Once `mmap` does the mapping, the mapping does not change.

Figure 3-7
page0 Program

```
#include <sys/file.h>              /* include file for file I/O */
#include <sys/mman.h>              /* include file of shared mem stuff */

main()
{
        char *Seg;
        int *Pagezero, *P;
        int fd, i;

        /* open the device for reading */
        fd = open("/dev/mem", O_RDONLY);
        if (fd < 0) {
                perror("Can't open /dev/mem");
                exit(2);
        }

        /* map the file into our address space at an o/s defined place */
        i = 4096;
        Seg = mmap(0, &i, PROT_READ, MAP_DEVICE, fd, 0);
        if ((int) Seg == -1 || i != 4096) {
                perror("Can't mmap");
                exit(2);
        }

        /* display page 0 */
        Pagezero = (int *) Seg;
        for (P = Pagezero; P < Pagezero + 32; P++)
                printf("0x%x: 0x%x\n", (P - Pagezero) * sizeof(int), *P);
}
```

Introductory Socket Programming

4

Introduction

Pipes and socketpairs are a simple solution for communicating between a parent process and a child process or between child processes. But how can you communicate between processes that have no common ancestor? Neither standard pipes nor socketpairs is the answer here, because both mechanisms require a common ancestor to set up communication. Often you need to have two processes separately create sockets and then have messages sent between them as when providing or using a service in the system. This is also the case when communicating between processes on different machines. ConvexOS allows you to create individual sockets, assign them names, and send messages between them.

This chapter provides a high-level description of socket facilities provided with ConvexOS. It is designed to complement man pages for IPC primitives with examples of their use. The remainder of this chapter is organized into four parts:

- Understanding sockets, domains, and socket types.
- Creating, using, and discarding sockets.
- Using network library routines to acquire remote addresses and information needed for socket programming.
- Constructing a typical distributed application.

ConvexOS IPC facilities support two address families and allow processes to rendezvous in many ways. Processes may rendezvous through a file-system-like name space (a space where all names are pathnames) as well as through a network name space. In fact, new name spaces may be added at a future time with only minor changes visible to users. Further, communication facilities have been extended to include more than the simple byte stream provided by a pipe.

Basics

The basic mechanism for interprocess communication is the *socket*. A socket is an endpoint of communication to which a name may be associated, or *"bound."* Each socket in use has a type and one or more associated processes. (Socket types are explained in more detail subsequently.) Sockets exist within *"communication domains."* A communication domain is an abstraction introduced to bundle common properties of processes communicating through sockets. One such property is the scheme used to named sockets. For example, in the UNIX communication domain, sockets are named with path names; e.g., a socket may be named /dev/foo. Sockets normally exchange data only with sockets in the same domain. ConvexOS IPC facilities support two communication domains: the UNIX domain, for local communication; and the Internet domain, for use by processes that communicate using standard Internet communication protocols. The Internet domain can be used both locally and remotely.

Underlying communication facilities provided by these domains have a significant influence on the internal system implementation as well as the interface to socket facilities available to a user. An example of the latter is that a socket operating in the UNIX domain sees a subset of error conditions that are possible when operating in the Internet domain.

Socket Types

Sockets are typed according to communication properties visible to a user. Processes are presumed to communicate only between sockets of the same type, although there is nothing that prevents communication between sockets of different types should underlying communication protocols support this.

Three types of sockets are available to users:

- **Stream sockets**—provide for bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from bidirectionality of data flow, a pair of connected stream sockets provides an interface nearly identical to that of pipes. (In the UNIX domain, in fact, the semantics are identical and, as you might expect, pipes have been implemented internally as simply a pair of connected stream sockets.)

- **Datagram sockets**—support bidirectional flow of data that is not promised to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages lost or duplicated, and, possibly, in an order different from the order in which they were sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model facilities found in many contemporary packet- switched networks such as the Ethernet.

- **Raw sockets**—provide users access to underlying communication protocols that support socket abstractions. These sockets are normally datagram oriented, although their exact characteristics depend on the interface provided by the protocol. Raw sockets are not intended for the general user; they are provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more esoteric facilities of an existing protocol.

Using Socket System Calls

Installations often use multiple network interfaces; for example, Ethernet for normal use and UltraNet for high-speed file transfers. The CONVEX UltraNet Interface uses its own socket system calls; other CONVEX network interfaces use standard ConvexOS socket calls. To use socket system calls—*socket, bind, accept, connect, listen,* etc.—network application programs are linked with either standard socket system calls in `libc.a` or with the UltraNet Socket Compatibility Library, `libulsock.a`. By default, programs are linking with `libc.a`; for a program to use UltraNet sockets, you must specify `libulsock.a` with the `-l` option when you compile it. For example, if you enter the command:

```
cc prog.c -lulsock
```

`prog.c` is linked with the UltraNet Sockets Compatibility Library.

The CONVEX UltraNet Interface includes both a "native" UltraNet interface, and an Internet interface module that allows it to participate in a TCP/IP network. When a program is linked with UltraNet sockets, socket-related calls are routed to the appropriate network interface. CONVEX UltraNet software uses the destination host address to determine the type of socket you need (TCP/IP or UltraNet) for the path you want to take. Refer to *CONVEX Networking Concepts* for more information on the CONVEX UltraNet Interface.

Creating Sockets

To create a socket, use the `socket` system call. This call requests the system to create a socket in the specified domain and of the specified type. The syntax of the command is:

s = **socket** (*domain, type, protocol*) ;

where:

s — Descriptor, in the form of a small integer number, returned by the system for use in later system calls that operate on the socket.

domain — One of the manifest constants defined in the <sys/socket.h> file. For the UNIX domain, the constant is `AF_UNIX`; for the Internet domain, it is `AF_INET`. Domain manifest constants are named `AF_`*domain* to indicate the "address family" to use in interpreting names in that domain.

type — One of the manifest constants: `SOCK_STREAM`, `SOCK_DGRAM`, or `SOCK_RAW`.

protocol — Name of the protocol to use with the socket. Protocol names are defined in the /etc/protocols file. Use `getprotobyname` to determine the protocol number if you do not know it. If you enter a value of 0 as the *protocol* argument, the system selects an appropriate protocol from those protocols that constitute the communication domain and that may be used to support the requested socket type.

To create a stream socket in the Internet domain, use a call similar to the following:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

This call results in a stream socket being created with the TCP protocol providing the underlying communication support. To create a datagram socket for use on the local system, use a call similar to the following:

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

The default protocol used when the protocol argument to the `socket` call is 0 should be correct for most situations. It is possible, however, to specify a protocol other than the default; this is described in Chapter 5, "Intermediate Socket Programming."

There are several reasons a `socket` call may fail. Aside from rare occurrence of lack of memory (`ENOBUFS`), a socket request may fail due to a request for an unknown protocol (`EPROTONOSUPPORT`), or a request for a type of socket for which there is no supporting protocol (`EPROTOTYPE`).

Binding Names to Sockets

When you create a socket, it has no name. Until a name is bound to a socket, processes have no way to reference it and, consequently, no messages may be received on it. Communicating processes are bound by an *association*. In the Internet domain, an association is composed of local and foreign addresses, and local and foreign ports; while in the UNIX domain, an association is composed of local and foreign pathnames. The phrase "foreign pathname" means a pathname created by a foreign process, not a pathname on a foreign system. In most domains, associations must be unique. In the Internet domain, there may never be duplicates of the ordered set <protocol, local address, local port, foreign address, foreign port>. UNIX domain sockets need not always be bound to a name, but when bound, there may never be duplicate <protocol, local pathname, foreign pathname> sets. Pathnames may not refer to files already existing on the system.

The `bind` system call allows a process to specify half of an association, <local address, local port> or <local pathname>, while the `connect` and `accept` primitives are used to complete a socket's association.

In the Internet domain, binding names to sockets can be fairly complex. Fortunately, it is usually not necessary to specifically bind an address and port number to a socket, because `connect` and `send` calls automatically bind an appropriate address if they are used with an unbound socket.

The syntax of the `bind` system call is as follows:

```
bind(s, name, namelen);
```

where:

s	Descriptor returned by the `socket` system call.
name	Variable-length byte string, which is interpreted by the supporting protocol(s). Its interpretation may vary from communication domain to communication domain, as this is one of the properties that constitute the domain. In the Internet domain, names contain an Internet address and port number. In the UNIX domain, names contain a path name and a family, which is always `AF_UNIX`.
namelen	Length of the string that contains the name.

The following example binds the name, `/tmp/foo`, to a UNIX domain socket:

```
#include <sys/un.h>
    .
    .
    .
struct sockaddr_un addr;
    .
    .
    .
strcpy(addr.sun_path, "/tmp/foo");
addr.sun_family = AF_UNIX;
bind(s, (struct sockaddr *) &addr, strlen(addr.sun_path) +
      sizeof (addr.sun_family));
```

The `sockaddr_un` structure shown in this example is used in the UNIX domain to store domain names and pathnames. In the Internet domain, host address, port numbers, and domain names are carried in the `sockaddr_in` structure. You use `getservbyname` to fill in the port number field. Refer to the `getservbyname`(3N) man page for details.

In determining the size of a UNIX domain address, null bytes are not counted, which is why `strlen` is used. In the current implementation of UNIX domain IPC, the file name referred to in `addr.sun_path` is created as a socket in the system file space. The caller must, therefore, have write permission in the directory where `addr.sun_path` is to reside, and this file should be deleted by the caller when it is no longer needed.

Binding an Internet address is more complicated. The actual call is similar:

```
#include <sys/types.h>
#include <netinet/in.h>
    .
    .
    .
struct sockaddr_in sin;
    .
    .
    .
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

but the selection of what to place in address, `sin`, requires some discussion. The problem of formulating Internet addresses is described later in this chapter in the section, "Using Network Library Routines," which discusses library routines used in name resolution.

Initiating Socket Connections

Connection establishment is usually asymmetrical, with one process acting as a *client* and the other performing the role of a *server*. When willing to offer its advertised services, the server binds a socket to a well-known address associated with the service and then passively listens on its socket. It is then possible for an unrelated process to rendezvous with the server. The client requests services from the server by initiating a connection to the server's socket. On the client side, the `connect` call is used to initiate a connection. Using the UNIX domain, this system call might appear as follows:

```
struct sockaddr_un server;
    .
    .
    .
connect(s, (struct sockaddr *)&server,
      strlen(server.sun_path) + sizeof (server.sun_family));
```

In the Internet domain, the following sequence is used:

```
struct sockaddr_in server;
    .
    .
    .
connect(s, (struct sockaddr *)&server, sizeof (server));
```

In the above examples, `server` contains either the pathname or the Internet address and port number of the server to which the client process wishes to connect. If the client process' socket is unbound at the time of the `connect` call, the system automatically selects and binds a name to the socket if necessary; refer to Chapter 6, "Advanced Socket Programming." This is the usual way that local addresses are bound to a socket.

An error is returned if the connection was unsuccessful; any name automatically bound by the system, however, remains. Otherwise, the socket is associated with the server and data transfer may begin.

Some of the more common errors returned when a connection attempt fails are:

ETIMEDOUT — After failing to establish a connection for a period of time, the system decided there was no point in retrying the connection attempt. This usually occurs because the destination host is down, or because problems in the network resulted in lost transmissions.

ECONNREFUSED — The host refused service for some reason. This is usually due to a server process not being present at the requested name.

EISCONN — The socket is already connected.

ENETDOWN or EHOSTDOWN — These operational errors are returned based on status information delivered to the client host by underlying communication services.

ENETUNREACH or EHOSTUNREACH — These operational errors can occur either because the network or host is unknown (no route to the network or host is present), or because of status information returned by intermediate gateways or packet-switching nodes. Many times the status returned is not sufficient to distinguish a failed network from a failed host, in which case the system indicates that the entire network is unreachable.

The server willing to receive a client's connection must perform two steps after binding its socket:

1. Indicate a willingness to listen for incoming connection requests. For example,

```
listen(s, 5);
```

The second parameter to the `listen` call specifies the maximum number of outstanding connections that may be queued awaiting acceptance by the server process; this number is currently limited to 5. Should a connection be requested while the queue is full, the connection is not refused, but rather the individual messages that constitute the request are ignored. This gives a harried server time to make room in its pending connection queue while the client retries the connection request. Had the connection been returned with the ECONNREFUSED error, the client would be unable to tell if the server was up or not. It is still possible to get the ETIMEDOUT error back, although this is unlikely. The backlog limit of 5 avoids the problem of processes monopolizing system resources by setting an infinite backlog, then ignoring all connection requests.

2. With a socket marked as listening, a server may then `accept` a connection. For example,

```
struct sockaddr_in from;
    .
    .
    .
fromlen = sizeof (from);
newsock = accept(s, (struct sockaddr *)&from, &fromlen);
```

For the UNIX domain, `from` is declared as a `sockaddr_un` structure, but nothing different needs to be done as far as `fromlen` is concerned. In examples that follow, only Internet routines are discussed.

A new descriptor is returned on receipt of a connection, along with a new socket. If the server wishes to find out who its client is, it may supply a buffer for the client socket's name. The value-result parameter, `fromlen`, is initialized by the server to indicate how much space is associated with `from`, then modified on return to reflect the true size of the name. If the client's name is not of interest, the second parameter may be a null pointer.

`accept` normally blocks; that is, `accept` does not return until a connection is available or the system call is interrupted by a signal to the process. Furthermore, there is no way for a process to indicate it will accept connections from only a specific individual or individuals. It is up to the user process to consider who the connection is from and to close down the connection if it does not wish to speak to the process. If the server process wants to accept connections on more than one socket, or wants to avoid blocking on the `accept` call, there are alternatives; they are considered in Chapter 6, "Advanced Socket Programming."

Transferring Data

With a connection established, data may begin to flow. Several possible calls can be used to send and receive data. With the peer entity at each end of a connection anchored, a user can send or receive a message without specifying the peer. As one might expect in this case, normal `read` and `write` system calls are then usable. For example, you might use commands similar to the following:

```
write(s, buf, sizeof (buf));
read(s, buf, sizeof (buf));
```

In addition to `read` and `write`, you can use `send` and `recv`.

```
send(s, buf, sizeof (buf), flags);
recv(s, buf, sizeof (buf), flags);
```

While `send` and `recv` are virtually identical to `read` and `write`, the extra `flags` argument is important. The flags, defined in the <sys/socket.h> file, may be specified as non-zero values if one or more of the following options is required:

`MSG_OOB`	Send/receive out of band data.
`MSG_PEEK`	Look at data without reading.
`MSG_DONTROUTE`	Send data without routing packets.

Out-of-band data is a concept specific to stream sockets; it is discussed in Chapter 6, "Advanced Socket Programming." The option to have data sent without routing applied to outgoing packets is currently used only by the routing table management process, and is unlikely to be of interest to the casual user. The ability to preview data is, however, of interest. When `MSG_PEEK` is specified with a `recv` call, any data present is returned to the user, but treated as still "unread"; that is, the next `read` or `recv` call applied to the socket returns the data previously previewed.

Discarding Sockets

Once a socket is no longer needed, discard it by applying a `close` to the descriptor, as follows:

```
close(s);
```

If data is associated with a socket that promises reliable delivery (e.g., a stream socket) when a `close` takes place, the system continues to attempt to transfer data. If the data is still undelivered after a fairly long period of time, however, it is discarded. Should a user have no use for any pending data, the system may perform a `shutdown` on the socket prior to closing it. The `shutdown` system call is of the form:

```
shutdown(s, how);
```

where *how* is 0 if the user is no longer interested in reading data, 1 if no more data will be sent, or 2 if no data is to be sent or received.

Additional Information

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 processes and communication paths. After this code is debugged, code specific to the application can be added.

You can find detailed information about particular calls and protocols in the *ConvexOS Programmer's Reference*. The following man pages are particularly relevant:

Table 4-1
Relevant Man Pages

Refer to man page:	For information about:
`inet`(4F)	addresses
`socket`(2), `bind`(2)	creating and naming sockets
`listen`(2), `accept`(2), `connect`(2)	establishing connections
`gethostent`(3N)	network host entry
`getprotoent`(3N)	protocol entry
`getservent`(3N)	service entry
`read`(2), `write`(2), `send`(2), `recv`(2)	transferring data
`tcp`(4P), `udp`(4P)	protocols

Using Network Library Routines

This section considers C runtime routines provided to manipulate network addresses.

Locating a service on a remote host requires many levels of mapping before client and server may communicate. A service is assigned a name intended for human consumption; e.g., "the `login` server on host `monet`." This name and the name of the peer host must then be translated into network addresses, which are not necessarily suitable for human consumption. Finally, the address must then be used to locate a physical location and route to the service.

The specifics of these mappings are likely to vary between network architectures. For instance, it is better for a network to not require hosts to use names that reveal their physical location to the client host. Instead, underlying services in the network may discover the actual location of the host at the time a client host wishes to communicate. This ability to have hosts named in a location-independent manner may induce overhead in connection establishment, because a discovery process must take place, but it allows a host to be physically mobile without requiring it to notify its clientele of its current location.

Standard routines are provided for mapping host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers and appropriate protocol to use in communicating with the server process. The <netdb.h> file must be included when using any of these routines.

Host Names

An Internet host name-to-address mapping is represented by the `hostent` structure.

```
struct   hostent {
         char     *h_name;       /* official name of host */
         char     **h_aliases;   /* alias list */
         int      h_addrtype;    /* host address type
         int      h_length;      /* length of address */
         char     **h_addr_list; /* addresses */
#define  h_addr   h_addr_list[0] /* backward compatible */
};
```

The `gethostbyname` routine takes an Internet host name and returns a `hostent` structure, whereas the `gethostbyaddr` routine maps Internet host addresses into a `hostent` structure.

The official name of the host and its public aliases are returned by these routines, along with the address type (family) and the address. The database for these calls is provided by the /etc/hosts file (refer to `hosts`(5)).

Network Names

Routines for mapping network names to numbers are also provided. These routines return a `netent` structure.

```
struct    netent {
          char      *n_name;      /* official name of net */
          char      **n_aliases;  /* alias list */
          int       n_addrtype;   /* net address type */
          unsigned long n_net;    /* net no., host byte order */
};
```

The `getnetbyname`, `getnetbynumber`, and `getnetent` routines are the network counterparts to the host routines described in the previous section. These routines extract their information from the /etc/networks file (refer to `networks`(5)).

Protocol Names

The `protoent` structure defines the protocol-name mapping used with the `getprotobyname`, `getprotobynumber`, and `getprotoent` routines.

```
struct    protoent {
          char      *p_name;      /* official protocol name */
          char      **p_aliases;  /* alias list */
          int       p_proto;      /* protocol number */
};
```

These routines obtain their information from the /etc/protocols file (refer to `protocols`(5)).

Service Names

Information regarding services is slightly more complicated. A service is expected to reside at a specific port and to employ a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Furthermore, a service may reside on multiple ports. If this occurs, higher-level library routines must be bypassed or extended. Available services are listed in the /etc/services file. Service mapping is described by the `servent` structure.

```
struct    servent {
          char      *s_name;      /* official service name */
          char      **s_aliases;  /* alias list */
          int       s_port;       /* port number, net byte order */
          char      *s_proto;     /* protocol to use */
};
```

The `getservbyname` routine maps service names to a `servent` structure by specifying a service name and, optionally, a qualifying protocol. Thus the call:

```
sp = getservbyname("telnet", (char *) 0);
```

returns the service specification for a `telnet` server using any protocol, whereas the call:

```
sp = getservbyname("telnet", "tcp");
```

returns only the service specification for the `telnet` server that uses the TCP protocol. The `getservbyport` and `getservent` routines are also provided. `getservbyport` has an interface similar to that provided by `getservbyname`; an optional protocol name may be specified to qualify lookups.

Miscellaneous

With the support routines described above, an Internet application program should rarely have to deal directly with addresses. This allows services to be developed in a network-independent fashion. It is clear, however, purging all network dependencies is very difficult. As long as the user is required to supply network addresses when naming services and sockets, there is always some network dependency in a program. The example client program in Figure 4-1 on the following page illustrates this point. (This example is considered in more detail subsequently.)

Figure 4-1
Remote Login Client Code

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
 .
 .
 .
main(argc, argv)
        int argc;
        char *argv[];
{
        struct sockaddr_in server;
        struct servent *sp;
        struct hostent *hp;
        int s;
         .
         .
         .
        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogin: tcp/login: unknown service\n");
                exit(1);
        }
        hp = gethostbyname(argv[1]);
        if (hp == NULL) {
                fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
                exit(2);
        }
        bzero((char *)&server, sizeof (server));
        bcopy(hp->h_addr, (char *)&server.sin_addr, hp->h_length);
        server.sin_family = hp->h_addrtype;
        server.sin_port = sp->s_port;
        s = socket(AF_INET, SOCK_STREAM, 0);
        if (s < 0) {
                perror("rlogin: socket");
                exit(3);
        }
         .
         .
         .
        /* Connect does the bind() for us */
        if (connect(s, (struct sockaddr *)&server, sizeof (server)) < 0) {
                perror("rlogin: connect");
                exit(5);
        }
         .
         .
         .
}
```

To make the remote login program independent of the Internet protocols and addressing scheme, you must add a layer of routines that mask network- dependent aspects from mainstream login code. For current facilities available in the system, this does not appear to be worthwhile.

Aside from address-related database routines, several other routines of interest are available in the runtime library. These routines are intended mostly to simplify manipulation of names and addresses. Table 4-2 summarizes routines for manipulating variable-length byte strings and handling byte-swapping of network addresses and values.

Table 4-2
Byte-Handling Routines

Call	Synopsis
`bcmp(s1, s2, n)`	compare byte-strings; 0 if same, not 0 otherwise
`bcopy(s1, s2, n)`	copy n bytes from s1 to s2
`bzero(base, n)`	zero-fill n bytes starting at base
`htonl(val)`	convert 32-bit quantity from host to network byte order
`htons(val)`	convert 16-bit quantity from host to network byte order
`ntohl(val)`	convert 32-bit quantity from network to host byte order
`ntohs(val)`	convert 16-bit quantity from network to host byte order

Byte-swapping routines are provided because ConvexOS expects addresses to be supplied in network order. (On some other architectures, host byte ordering is different from network byte ordering.) Consequently, programs are sometimes required to byte swap words. Library routines that return network addresses provide them in network order so that they may simply be copied into structures provided to the system. This implies users should encounter the byte-swapping problem only when interpreting network addresses. For example, to print an Internet port, the following code is required:

```
printf("port number%d\n", ntohs(sp->s_port));
```

Constructing Client/Server Applications

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme, client applications request services from a server process. This section looks more closely at interactions between client and server, and considers some of the problems in developing client and server applications.

The client and server require a well-known set of conventions before service may be rendered or accepted. This set of conventions constitutes a protocol that is implemented at both ends of a connection. Depending on the situation, the protocol may be symmetrical or asymmetrical. In a symmetrical protocol, either side may play the master or slave role. An example of a symmetrical protocol is the `telnet` protocol used in the Internet domain for remote terminal emulation. In an asymmetrical protocol, one side is recognized as the master, with the other as the slave. An example of an asymmetrical protocol is the Internet file transfer protocol, `ftp`. No matter whether the specific protocol used in accessing a service is symmetrical or asymmetrical, there is always a client process and a server process. Properties of server processes are described next, followed by a description of client processes.

A server process normally listens at a well-known address for service requests. That is, the server process remains dormant until a connection is requested by a client's connection to the server's address. At such a time the server process "wakes up" and services the client, performing actions the client requests of it.

Alternative schemes using a service server may eliminate server processes that clog the system while remaining dormant most of the time. For Internet servers in ConvexOS, this scheme is implemented via `inetd`, the so called "Internet super--server." The daemon, `inetd`, listens at various ports, determined at start-up by reading a configuration file. When a connection is requested to a port on which `inetd` is listening, `inetd` executes the appropriate server program to handle the client. With this method, clients are unaware that an intermediary such as `inetd` has played any part in the connection. `inetd` is described in more detail in Chapter 6, "Advanced Socket Programming." You can also refer to the `inetd`(8) man page.

Servers

In ConvexOS, most servers are accessed at well-known Internet addresses or UNIX domain names. For example, the remote login server's main loop is of the form shown in Figure 4-2 on the following page.

Figure 4-2
Remote Login Server

```
main(argc, argv)

        int argc;
        char *argv[];
{
        int f;
        struct sockaddr_in from;
        struct servent *sp;

        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogind: tcp/login: unknown service\n");
                exit(1);
        }
          .
          .
          .
#ifndef DEBUG
        /* Disassociate server from controlling terminal */
        for (i = 0; i < 3; ++i)
                close(i);

        open("/", O_RDONLY);
        dup2(0, 1);
        dup2(0, 2);

        i = open("/dev/tty", O_RDWR);
        if (i >= 0) {
                ioctl(i, TIOCNOTTY, 0);
                close(i);
        }
#endif

        sin.sin_port = sp->s_port;/* Restricted port -- refer to Chapter 6 */
          .
          .
          .
        f = socket(AF_INET, SOCK_STREAM, 0);
          .
          .
          .
        if (bind(f, (struct sockaddr *) &sin, sizeof (sin)) < 0) {
                .
                .
                .
        }
          .
          .
          .
        listen(f, 5);
```

Figure 4-2
Remote Login Server (continued)

```
        for (;;) {
                int g, len = sizeof (from);

                g = accept(f, (struct sockaddr *) &from, &len);
                if (g < 0) {
                        if (errno != EINTR)
                                syslog(LOG_DAEMON | LOG_ERR, "rlogind: accept: %m");
                        continue;
                }

                if (fork() == 0) {
                        close(f);
                        doit(g, &from);
                }
                close(g);
        }
}
```

The remainder of this chapter takes an in-depth look at the remote login server program shown in Figure 4-2.

First, the server looks up its service definition.

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogind: tcp/login: unknown service\n");
        exit(1);
}
```

The result of the `getservbyname` call is used in later portions of the code to define the Internet port at which it listens for service requests (indicated by a connection).

Second, the server disassociates from the controlling terminal of its invoker.

```
for (i = 0; i < 3; ++i)
        close(i);

open("/", O_RDONLY);
dup2(0, 1);
dup2(0, 2);

i = open("/dev/tty", O_RDWR);
if (i >= 0) {
        ioctl(i, TIOCNOTTY, 0);
        close(i);
}
```

This step is important because the server likely does not want to receive signals delivered to the process group of the controlling terminal. However, once a server has disassociated itself, it can no longer send reports of errors to a terminal, and must log errors via `syslog`.

Once a server has established a pristine environment, it creates a socket and begins accepting service requests. The `bind` call is required to ensure that the server listens at its expected location. The remote login server listens at a restricted port number, and must therefore run as `superuser`. This concept of a "restricted port number" is discussed in Chapter 6, "Advanced Socket Programming."

The main body of the loop is fairly simple:

```
for (;;) {
        int g, len = sizeof (from);

        g = accept(f, (struct sockaddr *)&from, &len);
        if (g < 0) {
                if (errno != EINTR)
                        syslog(LOG_DAEMON | LOG_ERR,
                                "rlogind: accept: %m");
                continue;
        }
        if (fork() == 0) {          /* Child */
                close(f);
                doit(g, &from);
        }
        close(g);                   /* Parent */
}
```

An `accept` call blocks the server until a client requests service. This call could return a failure status if the call is interrupted by a signal such as `SIGCHLD` (discussed in Chapter 5, "Intermediate Socket Programming"). Therefore, the return value from `accept` is checked to ensure that a connection has actually been established, and an error report is logged via syslog if an error has occurred.

With a connection in hand, the server then forks a child process and invokes the main body of the remote login protocol processing. The socket used by the parent for queuing connection requests is closed in the child, while the socket created as a result of the `accept` is closed in the parent. The address of the client is also passed to the `doit` routine, which authenticates clients.

Clients

The client side of the remote login service was shown in Figure 4-1. The separate, asymmetrical roles of the client and server are clear in the code. The server is a passive entity, listening for client connections, whereas the client process is an active entity, initiating a connection when invoked. This section describes more closely the steps taken by the client remote login process. As in the server process, the first step is to locate the service definition for a remote login.

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogin: tcp/login: unknown service\n");
        exit(1);
}
```

Next, the destination host is looked up with a `gethostbyname` call.

```
hp = gethostbyname(argv[1]);
if (hp == NULL) {
        fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
        exit(2);
}
```

With this accomplished, all that is required is to establish a connection to the server at the requested host and start the remote login protocol. The address buffer is cleared, then filled with the Internet address of the foreign host and the port number at which the login process resides on the foreign host.

```
bzero((char *)&server, sizeof (server));
bcopy(hp->h_addr, (char *) &server.sin_addr, hp->h_length);
server.sin_family = hp->h_addrtype;
server.sin_port = sp->s_port;
```

A socket is created and a connection is initiated. `connect` implicitly performs a `bind` call, because `s` is unbound.

```
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
if (s < 0) {
      perror("rlogin: socket");
      exit(3);
}
    .
    .
    .
if (connect(s, (struct sockaddr *) &server, sizeof (server)) < 0) {
      perror("rlogin: connect");
      exit(4);
}
```

Intermediate Socket Programming

5

Introduction

Having considered the basic aspects of socket programming in Chapter 4, "Introductory Socket Programming," you are now ready to explore a slightly more difficult set of concepts that enable you to develop advanced socket applications. In particular, this chapter explains in more detail the concepts of domain, communication style, and protocol. Understanding these concepts is important because they enable you to select programming options appropriate to your application.

The second part of the chapter explains in detail how to create, send, and receive datagrams in both the UNIX and Internet domains. (As Chapter 4 explains, datagrams are sockets that support a bidirectional flow of data that is not promised to be sequenced, reliable, or unduplicated.) This section contains numerous example programs that enable you to quickly learn the details of using datagrams.

The final part of this chapter explains how to develop connections for stream sockets, which provide bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Stream sockets must be connected to each other before data can be sent between them. Again, the discussion contains numerous programming examples.

Domains and Protocols

Sockets created by different programs use names to refer to one another; names generally must be translated into addresses. The space from which an address is drawn is referred to as a *domain*. ConvexOS supports two domains: the UNIX domain (or AF_UNIX, for Address Family UNIX) and the Internet domain (or AF_INET).

In the UNIX domain, a socket is given a pathname 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 an implementation of the 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 retransmits messages received with errors. It also returns error messages if you try 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, the message is generally 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. Individual datagrams are kept separate when they are read; that is, message boundaries are preserved.

The difference in performance between the two communication styles is generally less important than the difference in semantics. Performance you might gain 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. Because 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. 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 in this chapter.

You specify the domain, style, and protocol of a socket when you create it. For example, in Figure 5-1 on the following page, the call to `socket` creates a datagram socket with the default protocol in the UNIX domain.

Datagrams in the UNIX Domain

Let us now look at two programs that create sockets. Programs in Figures 5-1 and 5-2 use datagram, rather than stream, communication. The structure used to name UNIX domain sockets is defined in the <sys/un.h> file. The definition is also 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 `bind` system call. The program in Figure 5-1 uses the name, `socket`, which it binds to its socket. This name appears in the working directory of the program. The program in Figure 5-2 uses its socket only for sending messages. It does not create a name for the socket because no other process needs to refer to it.

Figure 5-1
Reading UNIX Domain Datagrams

```
#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[BUFSIZ];
        /* 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, BUFSIZ) < 0)
                perror("receiving datagram packet");
        printf("-->%s\n", buf);
        close(sock);
        unlink(NAME);
}
```

Figure 5-2
Sending a UNIX Domain Datagram

```
#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);
}
```

Names in the UNIX domain are pathnames. As with file pathnames, 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 pathnames 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 continues 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 with these objects. Names are removed by calling `unlink` or using the `rm` command. Names in the UNIX domain are used only 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. The program in Figure 5-2 gets the name of the socket to which it sends its message through its command-line arguments. Once a line of communication has been created, you can send names of additional, perhaps new, sockets over the link. Facilities must be built that make distribution of names less of a problem than it now is.

Datagrams in the Internet Domain

Examples in Figures 5-3 and 5-4 on the following pages are similar to previous examples except that the socket is in the Internet domain. The structure of Internet domain addresses is defined in the <netinet/in.h> file. Internet addresses specify a host address, as a 32-bit number, and a delivery slot, or port, on that machine. These ports are managed by 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 do not communicate with one another. As a result, different protocols may use the same port numbers. Thus, implicitly, an Internet address is an ordered set 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 ordered set <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.

Figure 5-3
Reading Internet 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 5-4
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 "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 host, 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);
}
```

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 for 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 5-3. 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 is the case if the wildcard value has been chosen. Even if the wildcard value is chosen, a program sending messages to the named socket must specify a valid network address. Programs can be willing to receive from "anywhere," but cannot send a message "anywhere." The program in Figure 5-4 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 calling `gethostbyname`. The returned structure includes the host's network address, which is copied into the structure to specify the destination of the message.

The port number can be thought of as the street number on a mailbox, into which the protocol places messages. Daemons offering certain advertised services have reserved or "well-known" port numbers. These port numbers are from 1 to 1023. Higher numbers are available to general users. Only servers need to ask for a particular number. The system assigns 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. Port numbers are not automatically reported back to the user. After calling `bind`, asking for port 0, you may call `getsockname` to discover what port was actually assigned. The `getsockname` routine does not support 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 bytes in the address. Because machines differ in the internal representation of integers, printing the port number as returned by `getsockname` may result in a misinterpretation. To print the number, it is necessary to use the `ntohs` routine (for network to host: short) to convert the number from the network representation to the host's representation. On CONVEX machines, this is a null operation. On other computers, 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. It is called `htons`; similar routines exist for long integers. For further information, refer to the entry for `byteorder`(3) in the *ConvexOS Programmer's Reference.*

Connections

To send data between stream sockets, sockets must be connected. Figures 5-5 and 5-6 on the following pages show two programs that create such a connection. The program in Figure 5-5 is relatively simple. To initiate a connection, this program simply creates a stream socket, and 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, the connection is completed, and the program can begin to send messages. Messages are 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, the operating system sends a `SIGPIPE` signal to the process. Unless explicit action is taken to handle the signal, the process terminates and the shell prints the message, "`broken pipe`." Refer to the `signal`(3C) or `sigvec`(2) man pages for more information.

Figure 5-5
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 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 host, 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 5-6
Accepting an Internet Domain Stream Connection

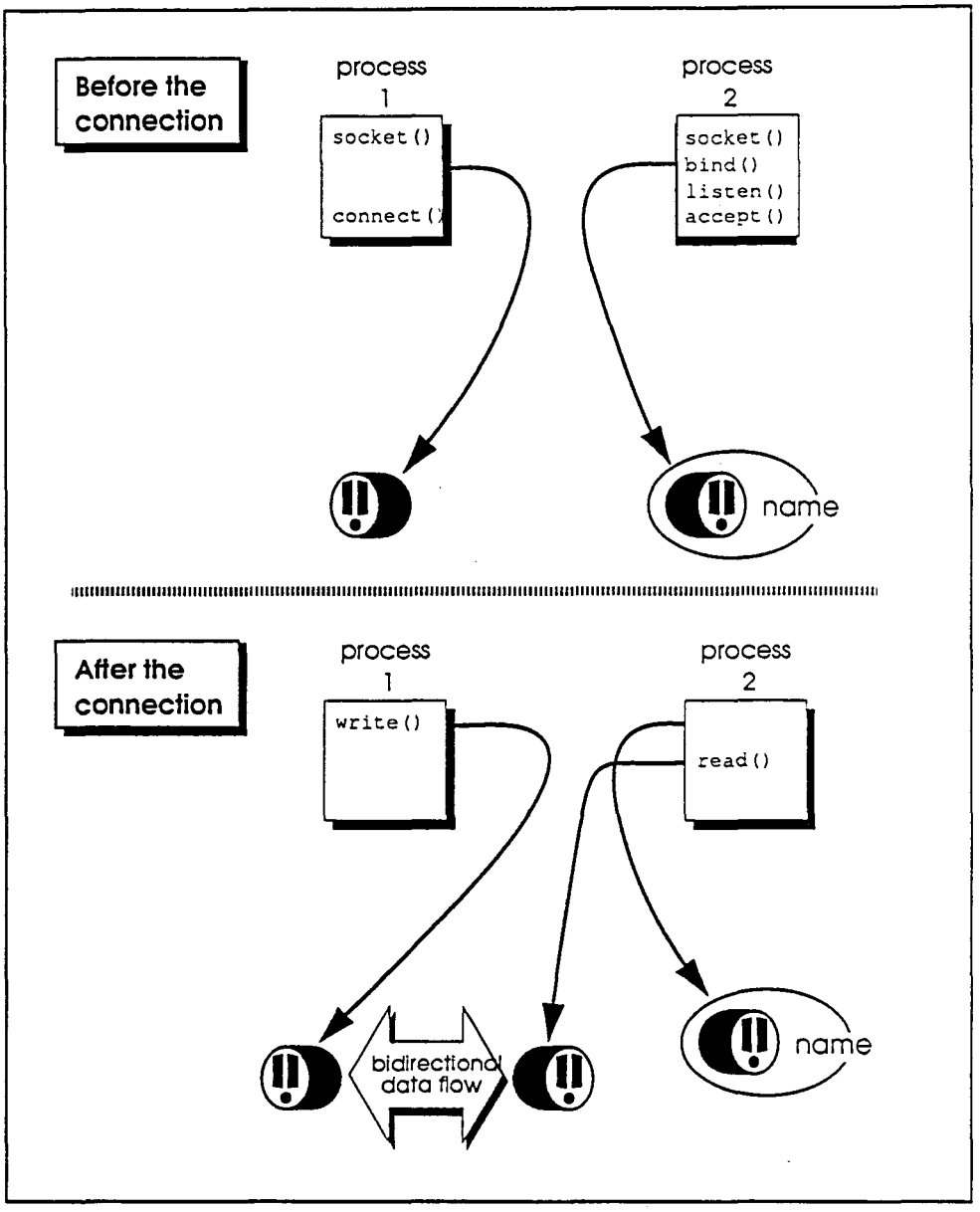
```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#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 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);
```

Figure 5-6
Accepting an Internet Domain Stream Connection (continued)

```
        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 are closed
         * automatically when a process is killed or terminates normally.
         */
}
```

Forming a connection is asymmetrical. One process, such as the program in Figure 5-5, 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 5-7 on the following page.

Figure 5-7
Establishing a Stream Connection



In Figure 5-7, `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 is destroyed by closing the corresponding socket.

The program in Figure 5-6 is a simple example of a server. It creates a socket to which it binds a name, which it then advertises; in this case, it prints the socket number. The program then calls `listen` for this socket. Because 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 passed to `listen`; the maximum length is limited by the system.

When the `listen` call has 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 5-7 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 the one printing the messages, is performed and the connection socket closed. The `accept` call takes a pending connection request from the queue if one is available, or blocks waiting for a request. Messages are read from the connection socket. Read operations from an active connection normally block until data is available. The number of bytes read is returned. When a connection is destroyed, the `read` call returns immediately, with the number of bytes set to zero.

The program in Figure 5-8 is a slight variation on the server in Figure 5-6. It avoids blocking when there are no pending connection requests by calling `select` to check for pending requests before calling `accept`. `select` takes five arguments: *nfds*, *readfds*, *writefds*, *exceptfds*, and *timeout*. *readfds*, *writefds*, and *exceptfds* are each `fd_sets`. An `fd_set` is a bitmap of file descriptor indexes. When you want to use `select` to learn when a socket is ready to have a connection accepted, use the `FD_SET` macro to set the bit corresponding to the socket's descriptor in the *readfds* `fd_set`. Do not set the bit in the *writefds* `fd_set` or the *exceptfds* `fd_set`. 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.

Figure 5-8
Using `select` to Check for Pending Connections

```
#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];
```

Figure 5-8
Using `select` to Check for Pending Connections (continued)

```
        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 */
        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);
}
```

Figures 5-9 and 5-10 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 machine. There are some differences, however, in the functionality of streams in the two domains, notably in the handling of out-of-band data. (Out-of-band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out-of--band data is discussed further in Chapter 6, "Advanced Socket Programming.") Describing these differences is beyond the scope of this manual.

Figure 5-9
Initiating a UNIX Domain 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 5-10
Accepting 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 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);
```

Figure 5-10
Accepting a UNIX Domain Stream Connection (continued)

```
        /*
         * 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 must kill this
         * program, it is necessary to remove the name by a command from
         * the shell.
         */

        close(sock);
        unlink(NAME);
}
```

Advanced Socket Programming

6

Introduction

For many users, socket facilities discussed in Chapters 4 and 5 are sufficient for construction of the application required. Some users, however, need to consider some or all of the advanced topics discussed here.

Topics discussed in this chapter include the following:

- Addressing datagrams.
- Using connectionless servers.
- Using `select` to multiplex I/O requests.
- Handling out-of-band data.
- Using non-blocking sockets.
- Using interrupt-driven socket I/O.
- Using signals and process groups.
- Using pseudoterminals.
- Using `inetd`, the Internet "super-server" daemon.
- Sending broadcast packets.
- Using `setsockopt` and `getsockopt` to set and retrieve socket options.
- Passing file descriptors.

Datagram Addressing

ConvexOS supports connectionless interactions typical of datagram facilities found in contemporary packet-switched networks. A datagram socket provides a symmetrical interface to data exchange. While processes are still likely to be client and server, there is no requirement to establish a connection. Instead, each message includes the destination address.

To use connectionless sockets, create datagrams just as you did with the connection-oriented model. If a particular local address is needed, the `bind` operation must precede the first data transmission. Otherwise, the system sets the local address and port when data is first sent. To send data, the `sendto` system call is used. The syntax of the `sendto` call is as follows:

```
sendto(s, buf, buflen, flags, (struct sockaddr *)&to, tolen);
```

s, *buf*, *buflen*, and *flags* parameters are used as before. *to* and *tolen* values indicate the address of the intended recipient of the message. When using an unreliable datagram interface, it is unlikely that any errors are reported to the sender. When information is present locally to recognize a message that can not be delivered (for instance, when a network is unreachable), the call returns 1, and the global value `errno` contains an error number.

To receive messages on an unconnected datagram socket, the `recvfrom` system call is provided. Its syntax is as follows:

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *)&from, &fromlen);
```

Once again, the *fromlen* parameter is handled in a value-result fashion, initially containing the size of the *from* buffer, and modified on return to indicate the actual size of the address from which the datagram was received.

In addition to the two calls mentioned above, datagram sockets may also use the `connect` call to associate a socket with a specific destination address. In this case, any data sent on the socket automatically addresses the connected peer, and only data received from that peer is delivered to the user. Only one connected address is permitted for each socket at one time; a second `connect` changes the destination address, and a `connect` to a null address (family `AF_UNSPEC`) disconnects. Connect requests on datagram sockets return immediately, as this simply results in the system recording the peer's address (as compared to a stream socket, where a connect request initiates establishment of an end-to-end connection). `accept` and `listen` are not used with datagram sockets.

While a datagram socket is connected, errors from recent `send` calls may be returned asynchronously. These errors may be reported on subsequent operations on the socket, or a special socket option used with `getsockopt`, `SO_ERROR`, may be used to interrogate the error status. A `select` for reading or writing returns true when an error indication is received. The next operation returns the error, and the error status is cleared.

Connectionless Servers

This section describes the use of a connectionless datagram-based broadcast using a server. The specific example used is the `rwho` service, which provides users with status information for hosts connected to a local area network. This service, while predicated on the ability to broadcast information to all hosts connected to a particular network, is of interest as an example usage of datagram sockets.

A user on any machine running the `rwho` server may find out the current status of a machine with the `ruptime` program. Typical output is shown in Figure 6-1.

Figure 6-1
Example `ruptime` Output

```
arpa    up    9:45,      5 users, load   1.15,   1.39,   1.31
cad     up    2+12:04,   8 users, load   4.67,   5.13,   4.59
calder  up    10:10      0 users, load   0.27,   0.15,   0.14
dali    up    2+06:28,   9 users, load   1.04,   1.20,   1.65
degas   up    25+09:48,  0 users, load   1.49,   1.43,   1.41
```

Status information for each host is periodically broadcast by `rwho` server processes on each machine. The same server process also receives status information and uses it to update a database. This database is then interpreted to generate the status information for each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

The use of broadcast for such a task is fairly inefficient, because all hosts must process each message, whether or not they use an `rwho` server. Unless such a service is sufficiently universal and is frequently used, the expense of periodic broadcasts outweighs the simplicity.

A simplified form of the `rwho` server, `rwhod`, is shown in Figure 6-2. The server performs two separate tasks:

1. `rwhod` acts as a receiver of status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the `rwho` port are interrogated to ensure they have been sent by another `rwho` server process. Packets are then time-stamped with their arrival time and used to update a file indicating the status of the host. When a host has not been heard from for an extended period of time, database interpretation routines assume the host is down and indicate such on status reports.
2. `rwhod` supplies information regarding the status of its host. This involves periodically acquiring system status information, packaging it in a message, and broadcasting it on the local network for other `rwho` servers to receive. The supply function is triggered by a timer and runs off a signal.

Locating system status information is somewhat involved, but uninteresting. Deciding where to transmit the resultant packet is somewhat problematical, however.

Status information must be broadcast on the local network. For networks that do not support broadcasting, another scheme must be used to simulate or replace broadcasting. One possibility is to enumerate known neighbors based on status messages received from other `rwho` servers. Unfortunately, this requires some bootstrapping information, because a server has no idea what machines are its neighbors until it receives status messages from them. Therefore, if all machines on a network are freshly booted, no machine has any known neighbors and thus never receives, or sends, any status information.

This is the identical problem faced by the routing table management process in propagating routing status information. Unsatisfactory as it may be, the standard solution is to inform one or more servers of known neighbors and request that they always communicate with these neighbors. If each server has at least one neighbor supplied to it, status information may then propagate through a neighbor to hosts that possibly are not directly neighbors. If the server is able to support networks that provide a broadcast capability, as well as those that do not, then networks with an arbitrary topology may share status information. However, you must be concerned about loops. That is, if a host is connected to multiple networks, it receives status information from itself. This can lead to an endless, wasteful exchange of information.

It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This requires a separate copy of the server at each host and makes maintenance a severe headache. ConvexOS attempts to isolate host-specific information from applications by providing system calls that return the necessary information. An example of such a system call is `gethostname`, which returns the host's official name.

A mechanism exists, in the form of an `ioctl` call, for finding the collection of networks to which a host is directly connected. Furthermore, a local network broadcasting mechanism is implemented at the socket level. Combining these two features allows a process to broadcast in a site-independent manner on any directly-connected local network that supports broadcasting. This allows ConvexOS to solve the problem of deciding how to propagate status information in the case of `rwho`, or more generally in broadcasting. Such status information is broadcast to connected networks at the socket level, where the connected networks have been obtained via appropriate `ioctl` calls. Specifics of such broadcasting are complex, however, and are covered subsequently.

Figure 6-2
rwho Server

```
main()
{
        .
        .
        .
    sp = getservbyname("who", "udp");
    np = getnetbyname("localnet");
    sin.sin_addr = inet_makeaddr(np->n_net, INADDR_ANY);
    sin.sin_port = sp->s_port;
        .
        .
        .
    s = socket(AF_INET, SOCK_DGRAM, 0);
        .
        .
        .
    on = 1;

    if (setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on)) < 0) {
            syslog(LOG_DAEMON | LOG_ERR, "setsockopt SO_BROADCAST: %m");
            exit(1);
    }

    bind(s, (struct sockaddr *) &sin, sizeof (sin));
        .
        .
        .
    signal(SIGALRM, onalrm);
    onalrm();

    for (;;) {
            struct whod wd;
            int cc, whod, len = sizeof (from);
            cc = recvfrom(s, (char *)&wd, sizeof (struct whod), 0,
                    (struct sockaddr *)&from, &len);
            if (cc <= 0) {
                    if (cc < 0 && errno != EINTR)
                            syslog(LOG_DAEMON | LOG_ERR, " rwho d: recv: %m");
                    continue;
            }

            if (from.sin_port != sp->s_port) {
                    syslog(LOG_DAEMON | LOG_ERR, " rwhod : %d: bad from port",
                            ntohs(from.sin_port));
                    continue;
            }
               .
               .
               .
```

Figure 6-2
rwho Server (continued)

```
                if (!verify(wd.wd_hostname)) {
                        syslog(LOG_DAEMON | LOG_ERR, " rwho d: malformed host name from %x",
                                ntohl(from.sin_addr.s_addr));
                        continue;
                }
                (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
                whod = open(path, O_WRONLY | O_CREAT | O_TRUNC, 0666);
                    .
                    .
                    .
                (void) time(&wd.wd_recvtime);
                (void) write(whod, (char *)&wd, cc);
                (void) close(whod);
        }
}
```

Using select to Multiplex I/O Requests

Another facility for developing applications is the use of I/O requests multiplexed among multiple sockets and/or files. This is done using the `select` call, as follows:

```
#include <sys/time.h>
#include <sys/types.h>
    .
    .
    .
fd_set readmask, writemask, exceptmask;
struct timeval timeout;
    .
    .
    .
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

`select` takes as arguments pointers to three descriptor sets: one for the set of descriptors for files from which the caller wishes to read data, one for those descriptors to which the caller wishes to write data, and one for which exceptional conditions are pending; out-of-band data is the only exceptional condition currently implemented for sockets. (Out-of-band data is a logically independent transmission channel associated with each pair of connected stream sockets. It is discussed later in this chapter.) If the user is not interested in certain conditions, e.g., read, write, or exceptions, the corresponding argument to the `select` should be a null pointer.

Each set is actually a structure containing an array of long integer bit masks; the size of the array is set by the definition `FD_SETSIZE`. The array is long enough to hold one bit for each of `FD_SETSIZE` file descriptors.

Macros `FD_SET(fd, &mask)` and `FD_CLR(fd, &mask)` are provided for adding and removing file descriptor `fd` in the set mask. The set should be zeroed before use, and the macro `FD_ZERO(&mask)` is provided to clear the set mask. The parameter `nfds` in the `select` call specifies the range of file descriptors to be examined in a set, i.e., one plus the value of the largest descriptor.

Both the `tv_sec` and `tv_usec` fields of the `timeval` structure should be initialized prior to calling `select`.

A `timeout` value may be specified if the selection is not to last more than a predetermined period of time. If the fields in `timeout` are set to 0, the selection takes the form of a poll, returning immediately. If `timeout` is a null pointer, the selection blocks indefinitely; a return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the system call.

`select` normally returns the number of file descriptors selected; if the `select` call returns due to the `timeout` expiring, then the value 0 is returned. If the `select` terminates because of an error or interruption, a 1 is returned with the error number in `errno`, and with the file descriptor masks unchanged.

Assuming a successful return, the three sets indicate which file descriptors are ready to be read from, written to, or have exceptional conditions pending. The status of a file descriptor in a `select` mask may be tested with the `FD_ISSET(fd, &mask)` macro, which returns a non-zero value if `fd` is a member of the set mask, and 0 if it is not.

To determine if connections are waiting on a socket to be used with an `accept` call, `select` can be used, followed by a `FD_ISSET(fd, &mask)` macro to check for read readiness on the appropriate socket. If `FD_ISSET` returns a non-zero value, indicating permission to read, then a connection is pending on the socket.

As an example, the code shown in Figure 6-3 on the following page might be used to read data from two sockets, `s1` and `s2`, as it is available from each and with a one-second `timeout`.

Figure 6-3
Reading Data From Two Sockets

```
#include <sys/time.h>
#include <sys/types.h>
    .
    .
    .
fd_set read_template;
struct timeval wait;
    .
    .
    .
for (;;) {
        wait.tv_sec = 1;        /* one second */
        wait.tv_usec = 0;

        FD_ZERO(&read_template);

        FD_SET(s1, &read_template);
        FD_SET(s2, &read_template);

        nb = select(FD_SETSIZE, &read_template, (fd_set *) 0, (fd_set *) 0, &wait);
        if (nb <= 0) {
                /* An error occurred during the select, or
                the select timed out */

           .
           .
           .

        }
        if (FD_ISSET(s1, &read_template)) {
                /* Socket #1 is ready to be read from. */

           .
           .
           .

        }
        if (FD_ISSET(s2, &read_template)) {
                /* Socket #2 is ready to be read from. */
        }

           .
           .
           .

}
```

Note

In previous versions of the CONVEX operating system, arguments to `select` were pointers to integers instead of pointers to `fd_sets`. This type of call still works as long as the number of file descriptors being examined is fewer than the number of bits in an integer; however, methods illustrated above should be used in programs running V6.0 or a later version of the CONVEX operating system.

`select` provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of `SIGIO` and `SIGURG` signals described in the next section.

Out-of-Band Data

The stream socket abstraction includes the notion of out-of-band data. *Out-of-band data* is a logically independent transmission channel associated with each pair of connected stream sockets. Out-of-band data is delivered to the user independently of normal data. The abstraction defines that out-of-band data facilities must support reliable delivery of at least one out-of-band message at a time. This message may contain at least one byte of data, and at least one message may be pending delivery to the user at any one time. For communication protocols that support only in-band signaling (i.e., urgent data is delivered in sequence with normal data), the system normally extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving urgent data in order and receiving it out of sequence without having to buffer all intervening data.

It is possible to "peek," via `MSG_PEEK`, at out-of-band data. If the socket has a process group, a `SIGURG` signal is generated when the protocol is notified of its existence. A process can set the process group or process id to be informed by the `SIGURG` signal via the appropriate `fcntl` call, as described below for `SIGIO`. If multiple sockets may have out-of-band data awaiting delivery, a `select` call for exceptional conditions may be used to determine those sockets with such data pending. Neither the signal nor the `select` indicates the actual arrival of out-of-band data, but only notification that it is pending.

In addition to information passed, a logical mark is placed in the data stream to indicate the point at which out-of-band data was sent. The remote login application uses this facility to propagate signals between client and server processes. When a signal flushes any pending output from the remote process or processes, all data up to the mark in the data stream is discarded.

To send an out-of-band message, the `MSG_OOB` option is supplied to a `send` or `sendto` call, while to receive out-of-band data, `MSG_OOB` should be indicated when performing a `recvfrom` or `recv` call. To find out if the read pointer is currently pointing at the mark in the data stream, the `SIOCATMARK ioctl` is provided.

```
ioctl(s, SIOCATMARK, &yes);
```

If *yes* is a 1 on return, the next `read` returns data after the mark. Otherwise (assuming out-of-band data has arrived), the next `read` provides data sent by the client prior to transmission of the out-of-band signal. The routine used in the remote login process to flush output on receipt of an interrupt or quit signal is shown in Figure 6-4 on the following page. It reads normal data up to the mark (to discard it), and then reads the out-of-band byte.

Figure 6-4
Flushing Terminal I/O on Receipt of Out-of-Band Data

```
#include <sys/ioctl.h>
#include <sys/file.h>
    .
    .
    .
oob()
{
        int out = FWRITE;
        char waste[BUFSIZ], mark;

        /* flush local terminal output */
        ioctl(1, TIOCFLUSH, (char *)&out);
        for (;;) {
                if (ioctl(rem, SIOCATMARK, &mark) < 0) {
                        perror("ioctl");
                        break;
                }
                if (mark)
                        break;
                (void) read(rem, waste, sizeof (waste));
        }
        if (recv(rem, &mark, 1, MSG_OOB) < 0) {
                perror("recv");
                   .
                   .
                   .
        }
           .
           .
           .
}
```

A process may also read or peek at out-of-band data without first reading up to the mark. This is more difficult when the underlying protocol delivers urgent data in-band with normal data, and only sends notification of its presence ahead of time, as with the TCP protocol used to implement streams in the Internet domain. With such protocols, the out-of-band byte may not yet have arrived when a `recv` is done with the `MSG_OOB` flag. In that case, the call returns an error of `EWOULDBLOCK`. Worse, there may be enough in-band data in the input buffer that normal flow control prevents the peer from sending urgent data until the buffer is cleared. The process must then read enough of the queued data that the urgent data may be delivered.

Certain programs that use multiple bytes of urgent data and must handle multiple urgent signals (e.g., `telnet`) need to retain the position of urgent data within the stream. This treatment is available as a socket-level option, `SO_OOBINLINE`; refer to the `setsockopt`(2) man page for usage. With this option, the position of the urgent data (the mark) is retained, but the urgent data immediately follows the mark within the normal data stream returned without the `MSG_OOB` flag. Reception of multiple urgent indications causes the mark to move, but no out-of-band data is lost.

Non-Blocking Sockets

It is occasionally convenient to make use of sockets that do not block; that is, I/O requests that cannot complete immediately and that would therefore cause the process to be suspended awaiting completion are not executed, and an error code is returned. Once a socket has been created via the `socket` call, it may be marked as non-blocking by using `fcntl` as follows:

```
#include <fcntl.h>
    .
    .
    .
int   s;
    .
    .
    .
s = socket(AF_INET, SOCK_STREAM, 0);
    .
    .
    .
if (fcntl(s, F_SETFL, FNDELAY) < 0)
      perror("fcntl F_SETFL, FNDELAY");
      exit(1);
}
```

When performing non-blocking I/O on sockets, you must be careful to check for the error `EWOULDBLOCK` (stored in the global variable `errno`), which occurs when an operation would normally block, but the socket it was performed on is marked as non-blocking. In particular, `accept`, `connect`, `send`, `recv`, `read`, and `write` can all return `EWOULDBLOCK`, and processes should be prepared to deal with such return codes. If an operation such as a `send` cannot be done in its entirety, but partial writes are sensible (for example, when using a stream socket), the data that can be sent immediately is processed, and the return value indicates the amount actually sent.

Interrupt-Driven Socket I/O

The SIGIO signal allows a process to be notified via a signal when a socket (or more generally, a file descriptor) has data waiting to be read. Using the SIGIO facility requires three steps:

1. The process must set up a SIGIO signal handler by use of the `signal` or `sigvec` calls.
2. It must set the process id or process group id that is to receive notification of pending input to its own process id, or the process group id of its process group (the default process group of a socket is group zero). This is accomplished by use of an `fcntl` call.

It must enable asynchronous notification of pending I/O requests with another `fcntl` call. Sample code to allow a given process to receive information on pending I/O requests as they occur for a socket s is given in Figure 6-5. With the addition of a handler for SIGURG, this code can also be used to prepare for receipt of SIGURG signals.

Figure 6-5
Using Asynchronous Notification of I/O Requests

```
#include <fcntl.h>
    .
    .
    .

int     io_handler();
    .
    .
    .

signal(SIGIO, io_handler);

/* Set the process receiving SIGIO/SIGURG signals to us */

if (fcntl(s, F_SETOWN, getpid()) < 0) {
        perror("fcntl F_SETOWN");
        exit(1);
}

/* Allow receipt of asynchronous I/O signals */

if (fcntl(s, F_SETFL, FASYNC) < 0) {
        perror("fcntl F_SETFL, FASYNC");
        exit(1);
}
```

Signals and Process Groups

Because of the existence of SIGURG and SIGIO signals, each socket has an associated process number (as do terminals). This value is initialized to zero, but may be redefined at a later time with the F_SETOWN fcntl, as was done in the code above for SIGIO. To set the socket's process id for signals, positive arguments should be given to the fcntl call. To set the socket's process group for signals, negative arguments should be passed to fcntl. The process number indicates either the associated process id or the associated process group; it is impossible to specify both at the same time. A similar fcntl, F_GETOWN, is available for determining the current process or process group number of a socket.

Another useful signal when constructing server processes is SIGCHLD. This signal is delivered to a process when any child process has changed state. Normally servers use the signal to reap child processes that have exited without explicitly awaiting their termination or periodic polling for exit status. For example, the remote login server loop shown in Chapter 4, "Introductory Socket Programming," may be augmented as shown in Figure 6-6.

Figure 6-6
Using the SIGCHLD Signal

```
int reaper();
    .
    .
    .
signal(SIGCHLD, reaper);
listen(f, 5);
for (;;) {
        int g, len = sizeof (from);

        g = accept(f, (struct sockaddr *)&from, &len,);
        if (g < 0) {
                if (errno != EINTR)
                        syslog(LOG_DAEMON | LOG_ERR, "rlogind: accept: %m");
                continue;
        }
           .
           .
           .
}
    .
    .
    .
#include <wait.h>
reaper()
{
        union wait status;

        while (wait3(&status, WNOHANG, 0) > 0)
                ;
}
```

Note

If the parent server process fails to reap its child processes, a large number of zombie processes may be created.

Pseudoterminals

Many programs do not function properly without a terminal for standard input and output. Because sockets do not provide the semantics of terminals, it is often necessary to have a process communicating over the network do so through a pseudoterminal. A *pseudoterminal* is actually a pair of devices, master and slave, that allow a process to serve as an active agent in communication between processes and users. Data written on the slave side of a pseudoterminal is supplied as input to a process reading from the master side, whereas data written on the master side is processed as terminal input for the slave. In this way, the process manipulating the master side of the pseudoterminal controls information read and written on the slave side as if it were manipulating the keyboard and reading the screen on an actual terminal. This abstraction preserves terminal semantics over a network connection—that is, the slave side appears as a normal terminal to any process reading from or writing to it.

For example, the remote login server uses pseudoterminals for remote login sessions. A user logging in to a machine across the network is provided a shell with a slave pseudoterminal as standard input, output, and error. The server process then handles communication between the programs invoked by the remote shell and the user's local client process. When a user sends a character that generates an interrupt and flushes terminal output on the remote machine, the pseudoterminal generates a control message for the server process. The server then sends an out-of-band message to the client process to signal a flush of data at the real terminal and on the intervening data buffered in the network.

Under ConvexOS, the name of the slave side of a pseudoterminal is of the form `/dev/tty`*xy*, where *x* is a single letter from the set "pqrstonmlkjihgfe," and *y* is a hexadecimal digit, i.e., a single character in the range 0 through 9 or "a" through "f." The master side of a pseudoterminal is `/dev/pty`*xy*, where *x* and *y* correspond to the slave side of the pseudoterminal.

In general, the method of obtaining a pair of pseudoterminals is to find an unused pseudoterminal. The master half of a pseudoterminal is a single-open device; thus, you could open each master in turn until an open succeeds, or you could use `getpty` as described below. Once you have found an unused pseudoterminal for the master side of the pair, the slave side of the pseudoterminal is then opened, and set to the proper terminal modes if necessary. The process then forks; the child closes the master side of the pseudoterminal, and `exec`s the appropriate program. Meanwhile, the parent closes the slave side of the pseudoterminal and begins reading and writing from the master side.

Using `getpty` is preferable to using successive `open` calls because `getpty` functions without regard to the number of pseudoterminals configured into your system, and independently of `pty` naming conventions described above. Be aware when using `getpty` that the name it returns is advisory only; it does not guarantee that a call to `open` will succeed.

Sample code making use of pseudoterminals is shown in Figure 6-7 on the following page; this code assumes that a connection on socket `s` exists, connected to a peer who wants a service of some kind, and that the process has disassociated itself from any previous controlling terminal.

Figure 6-7
Creating and Using a Pseudoterminal

```
/*
 * open master side.
 */

for (;;) {
        if ((line = getpty()) == NULL) {
                syslog(LOG_DAEMON | LOG_ERR, "All network ports in use");
                exit(1);
        }
        if ((master_fd = open (line, 2)) >= 0) {
                /*
                 * open slave side.
                 */
                line[5] = 't'; /* /dev/ptyXY ==> /dev/ttyXY */
                if ((slave_fd = open (line, 2)) >= 0)
                        break;
                else
                        close (master_fd);
        }
}

line[sizeof("/dev/")-1] = 't';
slave = open(line, O_RDWR);             /* slave is now slave side */
if (slave < 0) {
        syslog(LOG_DAEMON | LOG_ERR, "Cannot open slave pty %s", line);
        exit(1);
}

ioctl(slave, TIOCGETP, &b);             /* Set slave tty modes */
b.sg_flags = CRMOD|XTABS|ANYP;
ioctl(slave, TIOCSETP, &b);

i = fork();
if (i < 0) {
        syslog(LOG_DAEMON | LOG_ERR, "fork: %m");
        exit(1);
} else if (i) {                         /* Parent */
        close(slave);
           .
           .
           .
} else {                                /* Child */
        (void) close(s);
        (void) close(master);
        dup2(slave, 0);
        dup2(slave, 1);
        dup2(slave, 2);
        if (slave > 2)
                (void) close(slave);
           .
           .
           .
}
```

Using inetd

One of the daemons provided with ConvexOS is `inetd`, the so-called "Internet super-server." Invoked at boot time, `inetd` determines from the /etc/inetd.conf file the servers for which it is to listen. Once this information has been read and a pristine environment created, `inetd` creates one socket for each service it is to listen for, binding the appropriate port number to each socket. `inetd` is useful because it eliminates system overhead associated with multiple server processes waiting to accept connections. The format of the /etc/inetd.conf file is described in the `inetd`(8) man pages.

`inetd` then performs a `select` on all its sockets for read availability, waiting for somebody wishing a connection to the service corresponding to that socket. `inetd` then performs an `accept` on the socket in question, `fork`s, `dup`s the new socket to file descriptors 0 and 1 (`stdin` and `stdout`), closes other open file descriptors, and `exec`s the appropriate server.

Servers making use of `inetd` are considerably simplified, because `inetd` takes care of the majority of the IPC work required in establishing a connection. The server invoked by `inetd` expects the socket connected to its client on file descriptors 0 and 1, and may immediately perform any operations such as `read`, `write`, `send`, or `recv`. Indeed, servers may use buffered I/O as provided by `stdio` conventions, as long as they remember to use `fflush` when appropriate.

One call that may be of interest when writing servers under `inetd` is the `getpeername` call, which returns the address of the peer process connected on the other end of the socket. For example, to log the Internet address in dot notation (e.g., 128.32.0.4) of a client connected to a server under `inetd`, the following code might be used:

```
struct sockaddr_in name;
int namelen = sizeof (name);
    .
    .
    .
if (getpeername(0, (struct sockaddr *)&name, &namelen) < 0) {
      syslog(LOG_DAEMON | LOG_ERR, "getpeername: %m");
      exit(1);
} else
      syslog(LOG_DAEMON | LOG_INFO, "Connection from %s",
              inet_ntoa(name.sin_addr));
    .
    .
    .
```

While the `getpeername` call is especially useful when writing programs to run with `inetd`, it can be used under other circumstances. Be warned, however, that `getpeername` fails on UNIX domain sockets.

Broadcasting and Determining Network Configuration

By using a datagram socket, it is possible to send broadcast packets on many networks supported by the system. The network itself must support broadcast; the system provides no simulation of broadcast in software. Broadcast messages can place a high load on a network because they force every host on the network to service them. Consequently, the ability to send broadcast packets is limited to sockets that are explicitly marked as allowing broadcasting. Broadcast is typically used for one of two reasons: it is desired to find a resource on a local network without prior knowledge of its address, or important functions such as routing require that information be sent to all accessible neighbors.

To send a broadcast message, create a datagram socket, as follows:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

The socket is marked as allowing broadcasting,

```
int       on = 1;
setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof (on));
```

and at least a port number should be bound to the socket.

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

The destination address of the broadcast message depends on the network(s) on which the message is to be broadcast. The Internet domain supports a shorthand notation for broadcast on the local network, the address `INADDR_BROADCAST`, defined in the <netinet/in.h> file. To determine the list of addresses for all reachable neighbors requires knowledge of the networks to which the host is connected. Because this information should be obtained in a host-independent fashion and may be impossible to derive, ConvexOS provides a method of retrieving this information from system data structures. The `SIOCGIFCONF ioctl` call returns the interface configuration of a host in the form of a single `ifconf` structure; this structure contains a "data area" made up of an array of `ifreq` structures, one for each network interface to which the host is connected. These structures are defined in the <net/if.h> file as shown on the following page:

```
struct ifconf {
      int     ifc_len;          /* size of associated buffer */
      union {
              caddr_t ifcu_buf;
              struct ifreq *ifcu_req;
      } ifc_ifcu;
};

#define ifc_buf ifc_ifcu.ifcu_buf/* buffer address */
#define ifc_req ifc_ifcu.ifcu_req /* array of structures returned */

#define IFNAMSIZ  16

struct ifreq {
      char    ifr_name[IFNAMSIZ]; /* if name, e.g. "en0" */
      union {
              struct sockaddr ifru_addr;
              struct sockaddr ifru_dstaddr;
              struct sockaddr ifru_broadaddr;
              short ifru_flags;
              caddr_t ifru_data;
      } ifr_ifru;
};

#define ifr_addr ifr_ifru.ifru_addr
                      /* address */
#define ifr_dstaddr ifr_ifru.ifru_dstaddr
                      /* other end of p-to-p link */
#define ifr_broadaddr ifr_ifru.ifru_broadaddr
                      /* broadcast address */
#define ifr_flags ifr_ifru.ifru_flags
                      /* flags */
#define ifr_data ifr_ifru.ifru_data
                      /* for use by interface */
```

The actual call that obtains the interface configuration is as follows:

```
struct ifconf ifc;
char buf[BUFSIZ];

ifc.ifc_len = sizeof (buf);
ifc.ifc_buf = buf;
if (ioctl(s, SIOCGIFCONF, (char *) &ifc) < 0) {
    .
    .
    .
}
```

After this call, `buf` contains one `ifreq` structure for each network to which the host is connected, and `ifc.ifc_len` has been modified to reflect the number of bytes used by the `ifreq` structures.

For each structure, a set of "interface flags" tells whether the network corresponding to that interface is up or down, point-to-point or broadcast, etc. The `SIOCGIFFLAGS ioctl` retrieves these flags for an interface specified by an `ifreq` structure as follows:

```
struct ifreq *ifr;

ifr = ifc.ifc_req;

for (n = ifc.ifc_len / sizeof (struct ifreq); --n >= 0; ifr++) {
      if (ifr->ifr_addr.sa_family != AF_INET)
            continue;
      if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
         .
         .
         .
      }
      /*
       * Skip boring cases.
       */
      if ((ifr->ifr_flags & IFF_UP) == 0 ||
          (ifr->ifr_flags & IFF_LOOPBACK) ||
          (ifr->ifr_flags & (IFF_BROADCAST | IFF_POINTTOPOINT)) == 0)
            continue;
```

After the flags have been obtained, the broadcast address must be obtained. In the case of broadcast networks, this is done via the `SIOCGIFBRDADDR ioctl`, whereas for point-to-point networks, the address of the destination host is obtained with `SIOCGIFDSTADDR`, as follows:

```
struct sockaddr dst;

if (ifr->ifr_flags & IFF_POINTTOPOINT) {
      if (ioctl(s, SIOCGIFDSTADDR, (char *) ifr) < 0) {
         .
         .
         .
      }
      bcopy((char *) ifr->ifr_dstaddr, (char *) &dst,
              sizeof (ifr->ifr_dstaddr));
} else if (ifr->ifr_flags & IFF_BROADCAST) {
      if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
         .
         .
         .
      }
      bcopy((char *) ifr->ifr_broadaddr, (char *) &dst,
              sizeof (ifr->ifr_broadaddr));
}
```

After appropriate `ioctl` calls have obtained the broadcast or destination address (now in `dst`), the `sendto` call may be used, as follows:

```
          sendto(s, buf, buflen, 0, (struct sockaddr *)&dst,
                    sizeof (dst));

}
```

In the above loop, one `sendto` occurs for every interface to which the host is connected that supports the notion of broadcast or point-to-point addressing. If a process wished only to send broadcast messages on a given network, code similar to that outlined above is used, but the loop needs to find the correct destination address.

Received broadcast messages contain the sender's address and port, because datagram sockets are bound before a message is allowed to go out.

Socket Options

It is possible to set and get a number of options on sockets via the `setsockopt` and `getsockopt` system calls. These options include such things as whether or not to mark a socket for broadcasting, whether or not to route, whether or not to linger on close, etc. The general forms of the calls are:

```
setsockopt(s, level, optname, optval, optlen);
```

and

```
getsockopt(s, level, optname, optval, optlen);
```

where:

s Socket on which to apply the option.

level Protocol layer on which to apply the option; in most cases, this is the "socket level," indicated by the symbolic constant `SOL_SOCKET`, defined in the <sys/socket.h> file.

optname Actual option in the form of a symbolic constant also defined in the <sys/socket.h> file.

For `getsockopt`, *optval* and *optlen* point to the value of the option (in most cases, whether the option is to be turned on or off), and the length of the value of the option, respectively. For `getsockopt`, *optlen* is a value-result parameter, initially set to the size of the storage area pointed to by *optval*, and modified on return to indicate the actual amount of storage used.

For `setsockopt`, neither *optlen* nor *optval* is a pointer. *optlen* is the length of the value of the option, not a pointer to that length. Similarly, *optval* contains the option value; it does not point to a buffer containing it.

An example helps clarify things. It is sometimes useful to determine the type, e.g., stream, datagram, etc., of an existing socket; programs under control of `inetd` (described below) may need to perform this task. This can be accomplished as follows via the `SO_TYPE` socket option and the `getsockopt` call.

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

int type, size;
size = sizeof (int);

if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &type, &size) < 0){
    .
    .
    .
}
```

After the `getsockopt` call, `type` is set to the value of the socket type, as defined in the <sys/socket.h> file. For example, if the socket were a datagram socket, `type` would have the value corresponding to `SOCK_DGRAM`.

Options available for a socket depend upon its communication domain and socket type. The UNIX domain supports `SO_SNDBUF`, `SO_RCVBUF`, `SO_ERROR`, and `SO_TYPE` options. In the Internet domain, all socket options are supported when you are using a stream socket; datagram sockets support only `SO_DONTROUTE`, `SO_BROADCAST`, `SO_SNDBUF`, `SO_RCVBUF`, `SO_REUSEADDR`, `SO_ERROR`, and `SO_TYPE` options.

The SO_OOBINLINE option is only supported for stream sockets in the Internet domain. When it is enabled, the socket can only receive out-of-band data as regular data; however, both the select for exceptional conditions and the SIGURG signal still apply. Out-of-band data is returned as one octet.

A socket's type and communication domain also influence actions taken when the socket receives a message larger than its receive buffer (as specified with the SO_RCVBUF option). Because stream data is reassembled at the socket layer, a stream socket could receive messages larger than its receive buffer. In the UNIX domain, if a socket receives a datagram larger than its receive buffer, an ENOBUFS error will occur. However, in the Internet domain, receiving a datagram larger than a socket's receive buffer will not result in an error, but the socket will not receive the datagram.

The SO_REUSEADDR option is useful in that it allows you to bind a socket to an address bound to another socket that has been closed, but its process control block still exists. This may be necessary because sometimes a small window exists between closing a socket and the time the address bound to it is released.

When the SO_KEEPALIVE option is enabled, the underlying protocol transmits periodic messages at 45-second intervals. If no response is received after transmission of 8 such messages, the connection is considered broken and processes using the socket are notified via the SIGPIPE signal.

Passing File Descriptors

Within the UNIX domain, sockets can be used to pass access rights. Access rights refer to file descriptors a process "owns" or has access to. These ownership rights can be passed via sockets (using sendmsg and recvmsg) between processes. In this way, the ability to access needed files can be shared between processes.

The ability to pass file descriptors allows processes to access files they would otherwise be restricted from accessing. A client program can contact a server and request open descriptors for files to which the server, and not the client, has access.

Figures 6-8 and 6-9 on the following pages illustrate how access rights can be transferred between client and server.

Figure 6-8
client.c: Transmitting Access Rights

```
/*
 * acc_client -- opens a file, then transmits its descriptor
 *    through the access rights of a datagram type
 *    unix domain socket.
 */

#include <sys/types.h>
#include <sys/file.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <sys/uio.h>

#define SOCKET_NAME "Delilah"

main(argc, argv)
int argc;
char *argv[];
{
        int i, fd, s;
        struct msghdr msg;
        struct sockaddr_un s_un;
        struct iovec iov;
        char buf[1], strcpy();
        if ((s = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0) {
                perror("socket");
                exit(1);
        }

        s_un.sun_family = AF_UNIX;
        (void) strcpy(s_un.sun_path, SOCKET_NAME);
        for (i = 1; i < argc; i++)
                if ((fd = open(argv[i], O_RDONLY, 0)) < 0)
                        perror(argv[i]);
                else {
                        msg.msg_accrights = (caddr_t) &fd;
                        msg.msg_accrightslen = sizeof fd;
                        msg.msg_name = (caddr_t) &s_un;
                        msg.msg_namelen = sizeof s_un;
                        iov.iov_base = buf;
                        iov.iov_len = 0;
                        msg.msg_iov = &iov;
                        msg.msg_iovlen = 1;
                        if (sendmsg(s, &msg, 0) < 0) {
                                perror("sendmsg");
                                exit(1);
                        }
                        (void) close(fd);
                }
        return 0;
}
```

Figure 6-9
acc.server: Receiving Access Rights

```
/*
 * acc_server -- receives file descriptors in the access rights
 *     of unix domain datagrams, then copies from the
 *     file descriptors to standard output.
 */
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <sys/uio.h>
#define SOCKET_NAME "Delilah"

main()
{
        int newfd, s, n;
        struct sockaddr_un s_un;
        struct msghdr msg;
        char buf[1024];
        struct iovec iov;
        extern char *strcpy();
        if ((s = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0) {
                perror("socket");
                exit(1);
        }
        s_un.sun_family = AF_UNIX;
        (void) strcpy(s_un.sun_path, SOCKET_NAME);
        (void) unlink(SOCKET_NAME);
        if (bind(s, (struct sockaddr *) &s_un, sizeof s_un)) {
                perror("bind");
                exit(1);
        }
        while (1) {
                msg.msg_accrights = (caddr_t) &newfd;
                msg.msg_accrightslen = sizeof newfd;
                msg.msg_name = 0;
                msg.msg_namelen = 0;
                iov.iov_base = buf;
                iov.iov_len = sizeof buf;
                msg.msg_iov = &iov;
                msg.msg_iovlen = 1;
                if (recvmsg(s, &msg, 0) < 0) {
                        perror("recvmsg");
                        exit(1);
                }
                while ((n = read(newfd, buf, sizeof buf)) > 0)
                        (void) write(1, buf, n);
                if (n < 0) {
                        perror("read");
                        exit(1);
                }
                if (close(newfd)) {
                        perror("close");
                        exit(1);
                }
        }
}
```

Reporting Problems

This appendix introduces the CONVEX Technical Assistance Center (TAC) and the `contact` utility.

The `contact` utility is an online system for reporting problems to the TAC. To use it, enter **`contact`** at the system prompt and answer the questions as they appear on the screen.

This appendix describes:

- Prerequisites for using `contact`.
- Tips for using `contact`.
- The step-by-step process `contact` takes you through.

Technical Assistance Center

The CONVEX Technical Assistance Center (TAC) is staffed by technical specialists who can address diverse questions and problems that arise in a supercomputing environment. If you have a hardware, software, or documentation question, contact the TAC. This group stands ready to solve such problems.

The contact Utility

The TAC recommends using the `contact` utility to report a hardware, software, or documentation problem. The `contact` utility is an interactive program that helps the TAC track reports and route them to the CONVEX personnel most qualified to fix a problem.

After you invoke `contact`, it prompts you for information about the problem. When you finish your report, `contact` electronically mails it to the TAC. The TAC notifies you within 48 hours that your report has been received.

Use of the `contact` utility requires:

- UNIX-to-UNIX Communication Protocol (UUCP) connection to the TAC.
- Full path name of the program or utility in question.
- Version number of the program or utility in question.

UUCP Connection

Before using `contact`, ask your system administrator if your site has a UUCP connection to the TAC. A UUCP connection allows files to be copied from one UNIX-based system to another. The `uucp` (UNIX-to-UNIX copy) command relies on either a dial-up or hard-wired UUCP communication line.

Finding the Program Path Name

To determine the full path name of the program or utility in question, use the `which` command. Figure A-1 illustrates use of the `which` command to find the full path name of the loader utility.

Figure A-1
Using the `which` Command

```
> which ld
/bin/ld
>
```

In this example, the full path name of the loader is `/bin/ld`.

If you use the C shell (`csh`), you can also use the `whence` command to find the program path name. The `whence` command works like `which`, but faster.

For more information on the `which` command, refer to the `which`(1) man page. You can also use the `info` online information system by entering **`info which`** at the system prompt.

Finding the Program Version Number

To determine the version number of the program or utility in question, use the `vers` command. Figure A-7 illustrates use of the `vers` command to find the version number of the loader (`ld`) utility. Enter **`vers`**, then the path name of the program or utility.

Figure A-2
Using the `vers` Command

```
> vers /bin/ld
/bin/ld: 7.0
>
```

In this example, the loader version number is 7.0.

For more information on the `vers` command, refer to the `vers`(1) man page. You can also use the `info` online information system by entering **`info vers`** at the system prompt.

Using contact

The `contact` utility prompts for the following information:

- ❑ Your name, title, phone number, and corporate name.
- ❑ Name and version of the product.
- ❑ One-line summary of the problem.
- ❑ Detailed description of the problem.
- ❑ Priority of the problem.
- ❑ Instructions on how to reproduce the problem.
- ❑ Comments about the problem.
- ❑ Comments about the documentation relating to the problem.
- ❑ Files to include in the `contact` report.

Following is a step-by-step discussion of these prompts.

Step 1a

To invoke the `contact` utility, enter **`contact`** at the system prompt. The system responds with a welcome message and a series of questions regarding your hardware, software, or documentation question. Figure A-3 illustrates use of the `contact` command.

Figure A-3
Beginning a `contact` Session

```
> contact
Welcome to contact version 0.11 ()

Enter your name, title, phone number, and corporate name (^D to terminate)
>
```

Step 1b

If there is a .contact file in your home directory, `contact` skips the first prompt. (Refer to "Using a .contact File" on page 6 for more information.) Figure A-4 illustrates the `contact` command and the system response when you have a .contact file in your home directory.

Figure A-4
Beginning a `contact`
Session with a .contact File

```
> contact
Welcome to contact version 0.11 ()

Enter the name of the product involved
>
```

Step 2

The `contact` utility prompts for the version number of the product. If you do not know the version number, press **CTRL-Z** to suspend the session.

Use the `which` (or `whence` if you use `csh`) and `vers` commands to find the version number of the product. Use the `fg` command to return to the session, and enter the version number in the form X.X or X.X.X.X.

Step 3

The `contact` utility prompts for a one-line summary of the problem. This summary is the subject header in any further correspondence regarding the problem. Please make this summary as descriptive as possible in one line.

Step 4

The `contact` utility prompts for a detailed description of the problem. Please make this description as complete as possible. Include source code and a stack backtrace when possible. (Refer to the `adb`(1) or `csd`(1) man page for information on obtaining a stack backtrace.) The more information you provide, the quicker the TAC can isolate and solve the problem.

Step 5

The `contact` utility prompts for the priority of the problem. Figure A-5 illustrates this prompt and priority levels from which to choose.

Figure A-5
Specifying Priority
of a Problem

```
Enter a problem priority, based on the following:
1) Critical    - work cannot proceed until the problem is resolved.
2) Serious     - work can proceed around the problem, with difficulty.
3) Necessary   - problem has to be fixed.
4) Annoying    - problem is bothersome.
5) Enhancement - requested enhancement.
6) Informative - for informational purposes only.
>
```

Step 6

The `contact` utility prompts for an explanation of how to reproduce the problem. Please include the command syntax, the options you used, and anything else you did to run the program.

Step 7

The `contact` utility prompts for any other pertinent comments. Please include all relevant information.

Step 8

The `contact` utility prompts for suggestions regarding documentation supporting the product. Indicate whether the documentation could be revised to address the problem.

Step 9

The `contact` utility prompts for names of files necessary to reproduce the problem. Figure A-6 illustrates this prompt and sample user response.

Figure A-6
Including Files in a `contact` Report

```
Are there any files that should be included in this report (yes | no)?
> yes
Please enter the names of the files, one to a line (^D to terminate)
> test.f
> ~/subroutines/sub.f
>
```

Note

"Tilde-Escape Sequences" on page 7 are not recognized in responses to this prompt. In `contact`, a tilde in this section indicates your home directory. This convention is based on use of the tilde for expanding file names in `csh`.

If you specify small text files, they are automatically included in the `contact` report. If the files are too large to be included in this report, `contact` gives further instructions on how to submit these files.

To specify a directory, combine directory files into a single file using the `tar` command (refer to the `tar`(1) man page for further information) or enter each file name in the directory on a single line in the `contact` report.

Step 10

The `contact` utility prompts you to review, edit, submit, or abort the report. Figure A-7 illustrates this prompt.

Figure A-7
Prompt to Review, Edit, Submit, or Abort Report

```
Please select one of the following options:
1) Review the problem report.
2) Edit the problem report.
3) Submit the problem report.
4) Abort the problem report.
>
```

Choose the number of the option you want to select. These options let you do the following:

Review Review the text of the `contact` report. You are then prompted again to select an option.

Edit Edit the text of the `contact` report. If you choose to edit the report, `contact` opens your default text editor.

Submit Send the report to the CONVEX TAC. The TAC notifies you within 48 hours that your report has been received. Choosing this option exits the `contact` utility and returns you to the shell.

Abort Save the text of the report in a file named ~/dead.report. Choosing this option exits `contact` and returns you to the shell.

Tips for Using contact

The `contact` utility is interactive and easy to use. This section lists tips to help you use it efficiently. In particular, this section explains how to:

- Use a `.contact` file.
- Abort a `contact` session.
- Resubmit an aborted report.
- Suspend a `contact` session.
- Move within `contact` from one prompt to another.
- Use tilde-escape sequences in the `contact` utility.

Using a .contact File

When you invoke `contact`, it first prompts for your name, title, phone number, and company name. You can, however, create a .contact file to skip this first prompt.

Follow these steps to create a .contact file:

1. Create a .contact file in your home directory.
2. Enter your name, job title, phone number, and company name, each on a new line.

When you invoke `contact`, it automatically includes the .contact file as input for the first prompt and proceeds to the next prompt.

Aborting the Report

To abort a `contact` report, either press the interrupt key (usually **CTRL-C**) or choose the abort option when prompted by the `contact` utility. Using **CTRL-C** to abort does not save the contents of the report. Using the `abort` option saves the contents of the report in a file named ~/dead.report.

Submitting the dead.report File

After you abort a `contact` session, the `contact` utility saves the report in a file named ~/dead.report. Using the `contact` command with the `-r` option automatically merges the contents of the ~/dead.report file into the new `contact` session. Enter

```
contact -r
```

and `contact` finds the ~/dead.report file and merges it into the `contact` report. You can then edit the report. When you end the editing session, `contact` resumes at the final prompt, which asks you to review, edit, submit, or abort the report.

Suspending a Report

Sometimes it is necessary to stop in the middle of a `contact` report and return to the shell (for instance, to suspend the `contact` session to find the program path name or version number). To suspend the `contact` session, press **CTRL-Z**.

To return to the `contact` session, type **`fg`**. Using **CTRL-Z** and the `fg` (foreground) command, you can switch between the `contact` utility and the shell. You cannot, however, use **CTRL-Z** and `fg` to switch back and forth in the Bourne shell (`sh`).

Ending a Response

The `contact` utility prompts for information pertinent to your hardware, software, or documentation question. Some prompts require one-line responses; to move to the next prompt, press **RETURN**. Other prompts require more than a one-line response; to move to the next prompt, press **CTRL-D**.

Tilde-Escape Sequences

The `contact` utility treats input beginning with a tilde (~) as a special sequence. The character following the tilde is considered a request for a special function. You can use the following tilde sequences within `contact`:

`~e` Start the text editor (defined in the `EDITOR` environment variable).

`~h` Display a list of available tilde-escape sequences.

`~p` Print the `contact` report to the terminal screen.

`~r` *filename* Read the contents of *filename* as a response to the current prompt. Some prompts require only a one-line response. This tilde-escape sequence works only for prompts that allow more than a one-line response.

`~~` Insert a single tilde as the first character in the line.

Index

D

E

F

R

S

T

U

V

W