5

Interprocess Communication

Chapter 3, "Processes," discussed the creation of processes and showed how one process can obtain the exit status of a child process. That's the simplest form of communication between two processes, but it's by no means the most powerful. The mechanisms of Chapter 3 don't provide any way for the parent to communicate with the child except via command-line arguments and environment variables, nor any way for the child to communicate with the parent except via the child's exit status. None of these mechanisms provides any means for communicating with the child process while it is actually running, nor do these mechanisms allow communication with a process outside the parent-child relationship.

This chapter describes means for interprocess communication that circumvent these limitations. We will present various ways for communicating between parents and children, between "unrelated" processes, and even between processes on different machines.

Interprocess communication (IPC) is the transfer of data among processes. For example, a Web browser may request a Web page from a Web server, which then sends HTML data. This transfer of data usually uses sockets in a telephone-like connection. In another example, you may want to print the filenames in a directory using a command such as `ls | lpr`. The shell creates an `ls` process and a separate `lpr` process, connecting

the two with a *pipe*, represented by the "|" symbol. A pipe permits one-way communication between two related processes. The `ls` process writes data into the pipe, and the `lpr` process reads data from the pipe.

In this chapter, we discuss five types of interprocess communication:

- Shared memory permits processes to communicate by simply reading and writing to a specified memory location.
- Mapped memory is similar to shared memory, except that it is associated with a file in the filesystem.
- Pipes permit sequential communication from one process to a related process.
- FIFOs are similar to pipes, except that unrelated processes can communicate because the pipe is given a name in the filesystem.
- Sockets support communication between unrelated processes even on different computers.

These types of IPC differ by the following criteria:

- Whether they restrict communication to related processes (processes with a common ancestor), to unrelated processes sharing the same filesystem, or to any computer connected to a network
- Whether a communicating process is limited to only write data or only read data
- The number of processes permitted to communicate
- Whether the communicating processes are synchronized by the IPC—for example, a reading process halts until data is available to read

In this chapter, we omit discussion of IPC permitting communication only a limited number of times, such as communicating via a child's exit value.

5.1 Shared Memory

One of the simplest interprocess communication methods is using shared memory. Shared memory allows two or more processes to access the same memory as if they all called `malloc` and were returned pointers to the same actual memory. When one process changes the memory, all the other processes see the modification.

5.1.1 Fast Local Communication

Shared memory is the fastest form of interprocess communication because all processes share the same piece of memory. Access to this shared memory is as fast as accessing a process's nonshared memory, and it does not require a system call or entry to the kernel. It also avoids copying data unnecessarily.

Because the kernel does not synchronize accesses to shared memory, you must provide your own synchronization. For example, a process should not read from the memory until after data is written there, and two processes must not write to the same memory location at the same time. A common strategy to avoid these race conditions is to use semaphores, which are discussed in the next section. Our illustrative programs, though, show just a single process accessing the memory, to focus on the shared memory mechanism and to avoid cluttering the sample code with synchronization logic.

5.1.2 The Memory Model

To use a shared memory segment, one process must allocate the segment. Then each process desiring to access the segment must attach the segment. After finishing its use of the segment, each process detaches the segment. At some point, one process must deallocate the segment.

Understanding the Linux memory model helps explain the allocation and attachment process. Under Linux, each process's virtual memory is split into pages. Each process maintains a mapping from its memory addresses to these virtual memory pages, which contain the actual data. Even though each process has its own addresses, multiple processes' mappings can point to the same page, permitting sharing of memory. Memory pages are discussed further in Section 8.8, "The `mlock` Family: Locking Physical Memory," of Chapter 8, "Linux System Calls."

Allocating a new shared memory segment causes virtual memory pages to be created. Because all processes desire to access the same shared segment, only one process should allocate a new shared segment. Allocating an existing segment does not create new pages, but it does return an identifier for the existing pages. To permit a process to use the shared memory segment, a process attaches it, which adds entries mapping from its virtual memory to the segment's shared pages. When finished with the segment, these mapping entries are removed. When no more processes want to access these shared memory segments, exactly one process must deallocate the virtual memory pages.

All shared memory segments are allocated as integral multiples of the system's *page size*, which is the number of bytes in a page of memory. On Linux systems, the page size is 4KB, but you should obtain this value by calling the `getpagesize` function.

5.1.3 Allocation

A process allocates a shared memory segment using `shmget` ("SHared Memory GET"). Its first parameter is an integer key that specifies which segment to create. Unrelated processes can access the same shared segment by specifying the same key value. Unfortunately, other processes may have also chosen the same fixed key, which could lead to conflict. Using the special constant `IPC_PRIVATE` as the key value guarantees that a brand new memory segment is created.

Its second parameter specifies the number of bytes in the segment. Because segments are allocated using pages, the number of actually allocated bytes is rounded up to an integral multiple of the page size.

The third parameter is the bitwise or of flag values that specify options to `shmget`. The flag values include these:

- `IPC_CREAT`—This flag indicates that a new segment should be created. This permits creating a new segment while specifying a key value.
- `IPC_EXCL`—This flag, which is always used with `IPC_CREAT`, causes `shmget` to fail if a segment key is specified that already exists. Therefore, it arranges for the calling process to have an "exclusive" segment. If this flag is not given and the key of an existing segment is used, `shmget` returns the existing segment instead of creating a new one.
- Mode flags—This value is made of 9 bits indicating permissions granted to owner, group, and world to control access to the segment. Execution bits are ignored. An easy way to specify permissions is to use the constants defined in `<sys/stat.h>` and documented in the section 2 `stat` man page.[1] For example, `S_IRUSR` and `S_IWUSR` specify read and write permissions for the owner of the shared memory segment, and `S_IROTH` and `S_IWOTH` specify read and write permissions for others.

For example, this invocation of `shmget` creates a new shared memory segment (or access to an existing one, if `shm_key` is already used) that's readable and writeable to the owner but not other users.

```
int segment_id = shmget (shm_key, getpagesize (),
                         IPC_CREAT | S_IRUSR |  S_IWUSER);
```

If the call succeeds, `shmget` returns a segment identifier. If the shared memory segment already exists, the access permissions are verified and a check is made to ensure that the segment is not marked for destruction.

5.1.4 Attachment and Detachment

To make the shared memory segment available, a process must use `shmat`, "SHared Memory ATtach." Pass it the shared memory segment identifier `SHMID` returned by `shmget`. The second argument is a pointer that specifies where in your process's address space you want to map the shared memory; if you specify NULL, Linux will choose an available address. The third argument is a flag, which can include the following:

- `SHM_RND` indicates that the address specified for the second parameter should be rounded down to a multiple of the page size. If you don't specify this flag, you must page-align the second argument to `shmat` yourself.
- `SHM_RDONLY` indicates that the segment will be only read, not written.

1. These permission bits are the same as those used for files. They are described in Section 10.3, "File System Permissions."

If the call succeeds, it returns the address of the attached shared segment. Children created by calls to `fork` inherit attached shared segments; they can detach the shared memory segments, if desired.

When you're finished with a shared memory segment, the segment should be detached using `shmdt` ("SHared Memory DeTach"). Pass it the address returned by `shmat`. If the segment has been deallocated and this was the last process using it, it is removed. Calls to `exit` and any of the `exec` family automatically detach segments.

5.1.5 Controlling and Deallocating Shared Memory

The `shmctl` ("SHared Memory ConTroL") call returns information about a shared memory segment and can modify it. The first parameter is a shared memory segment identifier.

To obtain information about a shared memory segment, pass `IPC_STAT` as the second argument and a pointer to a `struct shmid_ds`.

To remove a segment, pass `IPC_RMID` as the second argument, and pass NULL as the third argument. The segment is removed when the last process that has attached it finally detaches it.

Each shared memory segment should be explicitly deallocated using `shmctl` when you're finished with it, to avoid violating the systemwide limit on the total number of shared memory segments. Invoking `exit` and `exec` detaches memory segments but does not deallocate them.

See the `shmctl` man page for a description of other operations you can perform on shared memory segments.

5.1.6 An Example Program

The program in Listing 5.1 illustrates the use of shared memory.

Listing 5.1 **(*shm.c*) Exercise Shared Memory**

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main ()
{
  int segment_id;
  char* shared_memory;
  struct shmid_ds shmbuffer;
  int segment_size;
  const int shared_segment_size = 0x6400;

  /* Allocate a shared memory segment.  */
  segment_id = shmget (IPC_PRIVATE, shared_segment_size,
                       IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
```

continues

Listing 5.1 **Continued**

```
  /* Attach the shared memory segment.  */
  shared_memory = (char*) shmat (segment_id, 0, 0);
  printf ("shared memory attached at address %p\n", shared_memory);
  /* Determine the segment's size.  */
  shmctl (segment_id, IPC_STAT, &shmbuffer);
  segment_size = shmbuffer.shm_segsz;
  printf ("segment size: %d\n", segment_size);
  /* Write a string to the shared memory segment.  */
  sprintf (shared_memory, "Hello, world.");
  /* Detach the shared memory segment.  */
  shmdt (shared_memory);

  /* Reattach the shared memory segment, at a different address.  */
  shared_memory = (char*) shmat (segment_id, (void*) 0x5000000, 0);
  printf ("shared memory reattached at address %p\n", shared_memory);
  /* Print out the string from shared memory.  */
  printf ("%s\n", shared_memory);
  /* Detach the shared memory segment.  */
  shmdt (shared_memory);

  /* Deallocate the shared memory segment.  */
  shmctl (segment_id, IPC_RMID, 0);

  return 0;
}
```

5.1.7 Debugging

The `ipcs` command provides information on interprocess communication facilities, including shared segments. Use the `-m` flag to obtain information about shared memory. For example, this code illustrates that one shared memory segment, numbered 1627649, is in use:

```
% ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 1627649    user     640        25600      0
```

If this memory segment was erroneously left behind by a program, you can use the `ipcrm` command to remove it.

```
% ipcrm shm 1627649
```

5.1.8 Pros and Cons

Shared memory segments permit fast bidirectional communication among any number of processes. Each user can both read and write, but a program must establish and follow some protocol for preventing race conditions such as overwriting information before it is read. Unfortunately, Linux does not strictly guarantee exclusive access even if you create a new shared segment with `IPC_PRIVATE`.

Also, for multiple processes to use a shared segment, they must make arrangements to use the same key.

5.2 Processes Semaphores

As noted in the previous section, processes must coordinate access to shared memory. As we discussed in Section 4.4.5, "Semaphores for Threads," in Chapter 4, "Threads," semaphores are counters that permit synchronizing multiple threads. Linux provides a distinct alternate implementation of semaphores that can be used for synchronizing processes (called process semaphores or sometimes System V semaphores). Process semaphores are allocated, used, and deallocated like shared memory segments. Although a single semaphore is sufficient for almost all uses, process semaphores come in sets. Throughout this section, we present system calls for process semaphores, showing how to implement single binary semaphores using them.

5.2.1 Allocation and Deallocation

The calls `semget` and `semctl` allocate and deallocate semaphores, which is analogous to `shmget` and `shmctl` for shared memory. Invoke `semget` with a key specifying a semaphore set, the number of semaphores in the set, and permission flags as for `shmget`; the return value is a semaphore set identifier. You can obtain the identifier of an existing semaphore set by specifying the right key value; in this case, the number of semaphores can be zero.

Semaphores continue to exist even after all processes using them have terminated. The last process to use a semaphore set must explicitly remove it to ensure that the operating system does not run out of semaphores. To do so, invoke `semctl` with the semaphore identifier, the number of semaphores in the set, `IPC_RMID` as the third argument, and any `union semun` value as the fourth argument (which is ignored). The effective user ID of the calling process must match that of the semaphore's allocator (or the caller must be `root`). Unlike shared memory segments, removing a semaphore set causes Linux to deallocate immediately.

Listing 5.2 presents functions to allocate and deallocate a binary semaphore.

Listing 5.2 **(*sem_all_deall.c*) Allocating and Deallocating a Binary Semaphore**

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

/* We must define union semun ourselves.  */

union semun {
  int val;
  struct semid_ds *buf;
  unsigned short int *array;
  struct seminfo *__buf;
};

/* Obtain a binary semaphore's ID, allocating if necessary.  */

int binary_semaphore_allocation (key_t key, int sem_flags)
{
  return semget (key, 1, sem_flags);
}

/* Deallocate a binary semaphore.  All users must have finished their
   use.  Returns -1 on failure.  */

int binary_semaphore_deallocate (int semid)
{
  union semun ignored_argument;
  return semctl (semid, 1, IPC_RMID, ignored_argument);
}
```

5.2.2 Initializing Semaphores

Allocating and initializing semaphores are two separate operations. To initialize a semaphore, use `semctl` with zero as the second argument and `SETALL` as the third argument. For the fourth argument, you must create a `union semun` object and point its `array` field at an array of unsigned short values. Each value is used to initialize one semaphore in the set.

Listing 5.3 presents a function that initializes a binary semaphore.

Listing 5.3 **(*sem_init.c*) Initializing a Binary Semaphore**

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

```
/* We must define union semun ourselves.  */

union semun {
  int val;
  struct semid_ds *buf;
  unsigned short int *array;
  struct seminfo *__buf;
};

/* Initialize a binary semaphore with a value of 1.  */

int binary_semaphore_initialize (int semid)
{
  union semun argument;
  unsigned short values[1];
  values[0] = 1;
  argument.array = values;
  return semctl (semid, 0, SETALL, argument);
}
```

5.2.3 Wait and Post Operations

Each semaphore has a non-negative value and supports wait and post operations. The `semop` system call implements both operations. Its first parameter specifies a semaphore set identifier. Its second parameter is an array of `struct sembuf` elements, which specify the operations you want to perform. The third parameter is the length of this array.

The fields of `struct sembuf` are listed here:

- `sem_num` is the semaphore number in the semaphore set on which the operation is performed.
- `sem_op` is an integer that specifies the semaphore operation.

  If `sem_op` is a positive number, that number is added to the semaphore value immediately.

  If `sem_op` is a negative number, the absolute value of that number is subtracted from the semaphore value. If this would make the semaphore value negative, the call blocks until the semaphore value becomes as large as the absolute value of `sem_op` (because some other process increments it).

  If `sem_op` is zero, the operation blocks until the semaphore value becomes zero.
- `sem_flg` is a flag value. Specify `IPC_NOWAIT` to prevent the operation from blocking; if the operation would have blocked, the call to `semop` fails instead. If you specify `SEM_UNDO`, Linux automatically undoes the operation on the semaphore when the process exits.

Listing 5.4 illustrates wait and post operations for a binary semaphore.

Listing 5.4 **(*sem_pv.c*) Wait and Post Operations for a Binary Semaphore**

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

/* Wait on a binary semaphore.  Block until the semaphore value is positive, then
   decrement it by 1.  */

int binary_semaphore_wait (int semid)
{
  struct sembuf operations[1];
  /* Use the first (and only) semaphore.  */
  operations[0].sem_num = 0;
  /* Decrement by 1.  */
  operations[0].sem_op = -1;
  /* Permit undo'ing.  */
  operations[0].sem_flg = SEM_UNDO;

  return semop (semid, operations, 1);
}

/* Post to a binary semaphore: increment its value by 1.
   This returns immediately.  */

int binary_semaphore_post (int semid)
{
  struct sembuf operations[1];
  /* Use the first (and only) semaphore.  */
  operations[0].sem_num = 0;
  /* Increment by 1.  */
  operations[0].sem_op = 1;
  /* Permit undo'ing.  */
  operations[0].sem_flg = SEM_UNDO;

  return semop (semid, operations, 1);
}
```

Specifying the `SEM_UNDO` flag permits dealing with the problem of terminating a process while it has resources allocated through a semaphore. When a process terminates, either voluntarily or involuntarily, the semaphore's values are automatically adjusted to "undo" the process's effects on the semaphore. For example, if a process that has decremented a semaphore is killed, the semaphore's value is incremented.

5.2.4 Debugging Semaphores

Use the command `ipcs -s` to display information about existing semaphore sets. Use the `ipcrm sem` command to remove a semaphore set from the command line. For example, to remove the semaphore set with identifier 5790517, use this line:

```
% ipcrm sem 5790517
```

5.3 Mapped Memory

Mapped memory permits different processes to communicate via a shared file. Although you can think of mapped memory as using a shared memory segment with a name, you should be aware that there are technical differences. Mapped memory can be used for interprocess communication or as an easy way to access the contents of a file.

Mapped memory forms an association between a file and a process's memory. Linux splits the file into page-sized chunks and then copies them into virtual memory pages so that they can be made available in a process's address space. Thus, the process can read the file's contents with ordinary memory access. It can also modify the file's contents by writing to memory. This permits fast access to files.

You can think of mapped memory as allocating a buffer to hold a file's entire contents, and then reading the file into the buffer and (if the buffer is modified) writing the buffer back out to the file afterward. Linux handles the file reading and writing operations for you.

There are uses for memory-mapped files other than interprocess communication. Some of these are discussed in Section 5.3.5, "Other Uses for `mmap`."

5.3.1 Mapping an Ordinary File

To map an ordinary file to a process's memory, use the `mmap` ("Memory MAPped," pronounced "em-map") call. The first argument is the address at which you would like Linux to map the file into your process's address space; the value NULL allows Linux to choose an available start address. The second argument is the length of the map in bytes. The third argument specifies the protection on the mapped address range. The protection consists of a bitwise "or" of `PROT_READ`, `PROT_WRITE`, and `PROT_EXEC`, corresponding to read, write, and execution permission, respectively. The fourth argument is a flag value that specifies additional options. The fifth argument is a file descriptor opened to the file to be mapped. The last argument is the offset from the beginning of the file from which to start the map. You can map all or part of the file into memory by choosing the starting offset and length appropriately.

The flag value is a bitwise "or" of these constraints:

- `MAP_FIXED`—If you specify this flag, Linux uses the address you request to map the file rather than treating it as a hint. This address must be page-aligned.
- `MAP_PRIVATE`—Writes to the memory range should not be written back to the attached file, but to a private copy of the file. No other process sees these writes. This mode may not be used with `MAP_SHARED`.

- `MAP_SHARED`—Writes are immediately reflected in the underlying file rather than buffering writes. Use this mode when using mapped memory for IPC. This mode may not be used with `MAP_PRIVATE`.

If the call succeeds, it returns a pointer to the beginning of the memory. On failure, it returns `MAP_FAILED`.

When you're finished with a memory mapping, release it by using `munmap`. Pass it the start address and length of the mapped memory region. Linux automatically unmaps mapped regions when a process terminates.

5.3.2 Example Programs

Let's look at two programs to illustrate using memory-mapped regions to read and write to files. The first program, Listing 5.5, generates a random number and writes it to a memory-mapped file. The second program, Listing 5.6, reads the number, prints it, and replaces it in the memory-mapped file with double the value. Both take a command-line argument of the file to map.

Listing 5.5 **(*mmap-write.c*) Write a Random Number to a Memory-Mapped File**

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <time.h>
#include <unistd.h>
#define FILE_LENGTH 0x100

/* Return a uniformly random number in the range [low,high].  */

int random_range (unsigned const low, unsigned const high)
{
  unsigned const range = high - low + 1;
  return low + (int) (((double) range) * rand () / (RAND_MAX + 1.0));
}

int main (int argc, char* const argv[])
{
  int fd;
  void* file_memory;

  /* Seed the random number generator.  */
  srand (time (NULL));

  /* Prepare a file large enough to hold an unsigned integer.  */
  fd = open (argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
  lseek (fd, FILE_LENGTH+1, SEEK_SET);
```

```
  write (fd, "", 1);
  lseek (fd, 0, SEEK_SET);

  /* Create the memory mapping.  */
  file_memory = mmap (0, FILE_LENGTH, PROT_WRITE, MAP_SHARED, fd, 0);
  close (fd);
  /* Write a random integer to memory-mapped area.  */
  sprintf((char*) file_memory, "%d\n", random_range (-100, 100));
  /* Release the memory (unnecessary because the program exits).  */
  munmap (file_memory, FILE_LENGTH);

  return 0;
}
```

The `mmap-write` program opens the file, creating it if it did not previously exist. The third argument to `open` specifies that the file is opened for reading and writing. Because we do not know the file's length, we use `lseek` to ensure that the file is large enough to store an integer and then move back the file position to its beginning.

The program maps the file and then closes the file descriptor because it's no longer needed. The program then writes a random integer to the mapped memory, and thus the file, and unmaps the memory. The `munmap` call is unnecessary because Linux would automatically unmap the file when the program terminates.

Listing 5.6 **(*mmap-read.c*) Read an Integer from a Memory-Mapped File, and Double It**

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#define FILE_LENGTH 0x100

int main (int argc, char* const argv[])
{
  int fd;
  void* file_memory;
  int integer;

  /* Open the file.  */
  fd = open (argv[1], O_RDWR, S_IRUSR | S_IWUSR);
  /* Create the memory mapping.  */
  file_memory = mmap (0, FILE_LENGTH, PROT_READ | PROT_WRITE,
                      MAP_SHARED, fd, 0);
  close (fd);
```

continues

Listing 5.6 **Continued**

```
  /* Read the integer, print it out, and double it.  */
  scanf (file_memory, "%d", &integer);
  printf ("value: %d\n", integer);
  sprintf ((char*) file_memory, "%d\n", 2 * integer);
  /* Release the memory (unnecessary because the program exits).  */
  munmap (file_memory, FILE_LENGTH);

  return 0;
}
```

The `mmap-read` program reads the number out of the file and then writes the doubled value to the file. First, it opens the file and maps it for reading and writing. Because we can assume that the file is large enough to store an unsigned integer, we need not use `lseek`, as in the previous program. The program reads and parses the value out of memory using `sscanf` and then formats and writes the double value using `sprintf`.

Here's an example of running these example programs. It maps the file `/tmp/integer-file`.

```
% ./mmap-write /tmp/integer-file
% cat /tmp/integer-file
42
% ./mmap-read /tmp/integer-file
value: 42
% cat /tmp/integer-file
84
```

Observe that the text `42` was written to the disk file without ever calling `write`, and was read back in again without calling `read`. Note that these sample programs write and read the integer as a string (using `sprintf` and `sscanf`) for demonstration purposes only—there's no need for the contents of a memory-mapped file to be text. You can store and retrieve arbitrary binary in a memory-mapped file.

5.3.3 Shared Access to a File

Different processes can communicate using memory-mapped regions associated with the same file. Specify the `MAP_SHARED` flag so that any writes to these regions are immediately transferred to the underlying file and made visible to other processes. If you don't specify this flag, Linux may buffer writes before transferring them to the file.

Alternatively, you can force Linux to incorporate buffered writes into the disk file by calling `msync`. Its first two parameters specify a memory-mapped region, as for `munmap`. The third parameter can take these flag values:

- `MS_ASYNC`—The update is scheduled but not necessarily run before the call returns.
- `MS_SYNC`—The update is immediate; the call to `msync` blocks until it's done. `MS_SYNC` and `MS_ASYNC` may not both be used.

- `MS_INVALIDATE`—All other file mappings are invalidated so that they can see the updated values.

For example, to flush a shared file mapped at address `mem_addr` of length `mem_length` bytes, call this:

```
msync (mem_addr, mem_length, MS_SYNC | MS_INVALIDATE);
```

As with shared memory segments, users of memory-mapped regions must establish and follow a protocol to avoid race conditions. For example, a semaphore can be used to prevent more than one process from accessing the mapped memory at one time. Alternatively, you can use `fcntl` to place a read or write lock on the file, as described in Section 8.3, "`fcntl`: Locks and Other File Operations," in Chapter 8.

5.3.4 Private Mappings

Specifying `MAP_PRIVATE` to `mmap` creates a copy-on-write region. Any write to the region is reflected only in this process's memory; other processes that map the same file won't see the changes. Instead of writing directly to a page shared by all processes, the process writes to a private copy of this page. All subsequent reading and writing by the process use this page.

5.3.5 Other Uses for *mmap*

The `mmap` call can be used for purposes other than interprocess communications. One common use is as a replacement for `read` and `write`. For example, rather than explicitly reading a file's contents into memory, a program might map the file into memory and scan it using memory reads. For some programs, this is more convenient and may also run faster than explicit file I/O operations.

One advanced and powerful technique used by some programs is to build data structures (ordinary `struct` instances, for example) in a memory-mapped file. On a subsequent invocation, the program maps that file back into memory, and the data structures are restored to their previous state. Note, though, that pointers in these data structures will be invalid unless they all point within the same mapped region of memory and unless care is taken to map the file back into the same address region that it occupied originally.

Another handy technique is to map the special `/dev/zero` file into memory. That file, which is described in Section 6.5.2, "`/dev/zero`," of Chapter 6, "Devices," behaves as if it were an infinitely long file filled with 0 bytes. A program that needs a source of 0 bytes can `mmap` the file `/dev/zero`. Writes to `/dev/zero` are discarded, so the mapped memory may be used for any purpose. Custom memory allocators often map `/dev/zero` to obtain chunks of preinitialized memory.

5.4 Pipes

A *pipe* is a communication device that permits unidirectional communication. Data written to the "write end" of the pipe is read back from the "read end." Pipes are serial devices; the data is always read from the pipe in the same order it was written. Typically, a pipe is used to communicate between two threads in a single process or between parent and child processes.

In a shell, the symbol | creates a pipe. For example, this shell command causes the shell to produce two child processes, one for `ls` and one for `less`:

```
% ls | less
```

The shell also creates a pipe connecting the standard output of the `ls` subprocess with the standard input of the `less` process. The filenames listed by `ls` are sent to `less` in exactly the same order as if they were sent directly to the terminal.

A pipe's data capacity is limited. If the writer process writes faster than the reader process consumes the data, and if the pipe cannot store more data, the writer process blocks until more capacity becomes available. If the reader tries to read but no data is available, it blocks until data becomes available. Thus, the pipe automatically synchronizes the two processes.

5.4.1 Creating Pipes

To create a pipe, invoke the `pipe` command. Supply an integer array of size 2. The call to `pipe` stores the reading file descriptor in array position 0 and the writing file descriptor in position 1. For example, consider this code:

```
int pipe_fds[2];
int read_fd;
int write_fd;

pipe (pipe_fds);
read_fd = pipe_fds[0];
write_fd = pipe_fds[1];
```

Data written to the file descriptor `read_fd` can be read back from `write_fd`.

5.4.2 Communication Between Parent and Child Processes

A call to `pipe` creates file descriptors, which are valid only within that process and its children. A process's file descriptors cannot be passed to unrelated processes; however, when the process calls `fork`, file descriptors are copied to the new child process. Thus, pipes can connect only related processes.

In the program in Listing 5.7, a `fork` spawns a child process. The child inherits the pipe file descriptors. The parent writes a string to the pipe, and the child reads it out. The sample program converts these file descriptors into `FILE*` streams using `fdopen`. Because we use streams rather than file descriptors, we can use the higher-level standard C library I/O functions such as `printf` and `fgets`.

Listing 5.7 **(*pipe.c*) Using a Pipe to Communicate with a Child Process**

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

/* Write COUNT copies of MESSAGE to STREAM, pausing for a second
   between each.  */

void writer (const char* message, int count, FILE* stream)
{
  for (; count > 0; --count) {
    /* Write the message to the stream, and send it off immediately.  */
    fprintf (stream, "%s\n", message);
    fflush (stream);
    /* Snooze a while.  */
    sleep (1);
  }
}

/* Read random strings from the stream as long as possible.  */

void reader (FILE* stream)
{
  char buffer[1024];
  /* Read until we hit the end of the stream.  fgets reads until
     either a newline or the end-of-file.  */
  while (!feof (stream)
         && !ferror (stream)
         && fgets (buffer, sizeof (buffer), stream) != NULL)
    fputs (buffer, stdout);
}

int main ()
{
  int fds[2];
  pid_t pid;

  /* Create a pipe.  File descriptors for the two ends of the pipe are
     placed in fds.  */
  pipe (fds);
  /* Fork a child process.  */
  pid = fork ();
  if (pid == (pid_t) 0) {
    FILE* stream;
    /* This is the child process.  Close our copy of the write end of
       the file descriptor.  */
    close (fds[1]);
    /* Convert the read file descriptor to a FILE object, and read
       from it.  */
    stream = fdopen (fds[0], "r");
    reader (stream);
```

continues

Listing 5.7 **Continued**

```
    close (fds[0]);
  }
  else {
    /* This is the parent process.  */
    FILE* stream;
    /* Close our copy of the read end of the file descriptor.  */
    close (fds[0]);
    /* Convert the write file descriptor to a FILE object, and write
       to it.  */
    stream = fdopen (fds[1], "w");
    writer ("Hello, world.", 5, stream);
    close (fds[1]);
  }

  return 0;
}
```

At the beginning of `main`, `fds` is declared to be an integer array with size 2. The `pipe` call creates a pipe and places the read and write file descriptors in that array. The program then forks a child process. After closing the read end of the pipe, the parent process starts writing strings to the pipe. After closing the write end of the pipe, the child reads strings from the pipe.

Note that after writing in the `writer` function, the parent flushes the pipe by calling `fflush`. Otherwise, the string may not be sent through the pipe immediately.

When you invoke the command `ls | less`, two forks occur: one for the `ls` child process and one for the `less` child process. Both of these processes inherit the pipe file descriptors so they can communicate using a pipe. To have unrelated processes communicate, use a FIFO instead, as discussed in Section 5.4.5, "FIFOs."

5.4.3 Redirecting the Standard Input, Output, and Error Streams

Frequently, you'll want to create a child process and set up one end of a pipe as its standard input or standard output. Using the `dup2` call, you can equate one file descriptor with another. For example, to redirect a process's standard input to a file descriptor `fd`, use this line:

```
dup2 (fd, STDIN_FILENO);
```

The symbolic constant `STDIN_FILENO` represents the file descriptor for the standard input, which has the value 0. The call closes standard input and then reopens it as a duplicate of `fd` so that the two may be used interchangeably. Equated file descriptors share the same file position and the same set of file status flags. Thus, characters read from `fd` are not reread from standard input.

The program in Listing 5.8 uses `dup2` to send the output from a pipe to the `sort` command.[2] After creating a pipe, the program forks. The parent process prints some strings to the pipe. The child process attaches the read file descriptor of the pipe to its standard input using `dup2`. It then executes the `sort` program.

Listing 5.8 **(*dup2.c*) Redirect Output from a Pipe with *dup2***

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main ()
{
  int fds[2];
  pid_t pid;

  /* Create a pipe.  File descriptors for the two ends of the pipe are
     placed in fds.  */
  pipe (fds);
  /* Fork a child process.  */
  pid = fork ();
  if (pid == (pid_t) 0) {
    /* This is the child process.  Close our copy of the write end of
       the file descriptor.  */
    close (fds[1]);
    /* Connect the read end of the pipe to standard input.  */
    dup2 (fds[0], STDIN_FILENO);
    /* Replace the child process with the "sort" program.  */
    execlp ("sort", "sort", 0);
  }
  else {
    /* This is the parent process.  */
    FILE* stream;
    /* Close our copy of the read end of the file descriptor.  */
    close (fds[0]);
    /* Convert the write file descriptor to a FILE object, and write
       to it.  */
    stream = fdopen (fds[1], "w");
    fprintf (stream, "This is a test.\n");
    fprintf (stream, "Hello, world.\n");
    fprintf (stream, "My dog has fleas.\n");
    fprintf (stream, "This program is great.\n");
    fprintf (stream, "One fish, two fish.\n");
    fflush (stream);
    close (fds[1]);
    /* Wait for the child process to finish.  */
    waitpid (pid, NULL, 0);
  }

  return 0;
}
```

2. `sort` reads lines of text from standard input, sorts them into alphabetical order, and prints them to standard output.

5.4.4 *popen* and *pclose*

A common use of pipes is to send data to or receive data from a program being run in a subprocess. The `popen` and `pclose` functions ease this paradigm by eliminating the need to invoke `pipe`, `fork`, `dup2`, `exec`, and `fdopen`.

Compare Listing 5.9, which uses `popen` and `pclose`, to the previous example (Listing 5.8).

Listing 5.9 **(*popen.c*) Example Using *popen***

```
#include <stdio.h>
#include <unistd.h>

int main ()
{
  FILE* stream = popen ("sort", "w");
  fprintf (stream, "This is a test.\n");
  fprintf (stream, "Hello, world.\n");
  fprintf (stream, "My dog has fleas.\n");
  fprintf (stream, "This program is great.\n");
  fprintf (stream, "One fish, two fish.\n");
  return pclose (stream);
}
```

The call to `popen` creates a child process executing the `sort` command, replacing calls to `pipe`, `fork`, `dup2`, and `execlp`. The second argument, `"w"`, indicates that this process wants to write to the child process. The return value from `popen` is one end of a pipe; the other end is connected to the child process's standard input. After the writing finishes, `pclose` closes the child process's stream, waits for the process to terminate, and returns its status value.

The first argument to `popen` is executed as a shell command in a subprocess running `/bin/sh`. The shell searches the `PATH` environment variable in the usual way to find programs to execute. If the second argument is `"r"`, the function returns the child process's standard output stream so that the parent can read the output. If the second argument is `"w"`, the function returns the child process's standard input stream so that the parent can send data. If an error occurs, `popen` returns a null pointer.

Call `pclose` to close a stream returned by `popen`. After closing the specified stream, `pclose` waits for the child process to terminate.

5.4.5 FIFOs

A *first-in, first-out (FIFO)* file is a pipe that has a name in the filesystem. Any process can open or close the FIFO; the processes on either end of the pipe need not be related to each other. FIFOs are also called *named pipes*.

You can make a FIFO using the `mkfifo` command. Specify the path to the FIFO on the command line. For example, create a FIFO in `/tmp/fifo` by invoking this:

```
% mkfifo /tmp/fifo
% ls -l /tmp/fifo
prw-rw-rw-    1 samuel   users           0 Jan 16 14:04 /tmp/fifo
```

The first character of the output from `ls` is `p`, indicating that this file is actually a FIFO (named pipe). In one window, read from the FIFO by invoking the following:

```
% cat < /tmp/fifo
```

In a second window, write to the FIFO by invoking this:

```
% cat > /tmp/fifo
```

Then type in some lines of text. Each time you press Enter, the line of text is sent through the FIFO and appears in the first window. Close the FIFO by pressing Ctrl+D in the second window. Remove the FIFO with this line:

```
% rm /tmp/fifo
```

Creating a FIFO

Create a FIFO programmatically using the `mkfifo` function. The first argument is the path at which to create the FIFO; the second parameter specifies the pipe's owner, group, and world permissions, as discussed in Chapter 10, "Security," Section 10.3, "File System Permissions." Because a pipe must have a reader and a writer, the permissions must include both read and write permissions. If the pipe cannot be created (for instance, if a file with that name already exists), `mkfifo` returns –1. Include `<sys/types.h>` and `<sys/stat.h>` if you call `mkfifo`.

Accessing a FIFO

Access a FIFO just like an ordinary file. To communicate through a FIFO, one program must open it for writing, and another program must open it for reading. Either low-level I/O functions (`open`, `write`, `read`, `close`, and so on, as listed in Appendix B, "Low-Level I/O") or C library I/O functions (`fopen`, `fprintf`, `fscanf`, `fclose`, and so on) may be used.

For example, to write a buffer of data to a FIFO using low-level I/O routines, you could use this code:

```
int fd = open (fifo_path, O_WRONLY);
write (fd, data, data_length);
close (fd);
```

To read a string from the FIFO using C library I/O functions, you could use this code:

```
FILE* fifo = fopen (fifo_path, "r");
fscanf (fifo, "%s", buffer);
fclose (fifo);
```

A FIFO can have multiple readers or multiple writers. Bytes from each writer are written atomically up to a maximum size of `PIPE_BUF` (4KB on Linux). Chunks from simultaneous writers can be interleaved. Similar rules apply to simultaneous reads.

Differences from Windows Named Pipes

Pipes in the Win32 operating systems are very similar to Linux pipes. (Refer to the Win32 library documentation for technical details about these.) The main differences concern named pipes, which, for Win32, function more like sockets. Win32 named pipes can connect processes on separate computers connected via a network. On Linux, sockets are used for this purpose. Also, Win32 allows multiple reader-writer connections on a named pipe without interleaving data, and pipes can be used for two-way communication.[3]

5.5 Sockets

A *socket* is a bidirectional communication device that can be used to communicate with another process on the same machine or with a process running on other machines. Sockets are the only interprocess communication we'll discuss in this chapter that permit communication between processes on different computers. Internet programs such as Telnet, rlogin, FTP, talk, and the World Wide Web use sockets.

For example, you can obtain the WWW page from a Web server using the Telnet program because they both use sockets for network communications.[4] To open a connection to a WWW server at `www.codesourcery.com`, use `telnet www.codesourcery.com 80`. The magic constant 80 specifies a connection to the Web server programming running `www.codesourcery.com` instead of some other process. Try typing `GET /` after the connection is established. This sends a message through the socket to the Web server, which replies by sending the home page's HTML source and then closing the connection—for example:

```
% telnet www.codesourcery.com 80
Trying 206.168.99.1...
Connected to merlin.codesourcery.com (206.168.99.1).
Escape character is '^]'.
GET /
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
...
```

3. Note that only Windows NT can create a named pipe; Windows 9*x* programs can form only client connections.

4. Usually, you'd use `telnet` to connect a Telnet server for remote logins. But you can also use `telnet` to connect to a server of a different kind and then type comments directly at it.

5.5.1 Socket Concepts

When you create a socket, you must specify three parameters: communication style, namespace, and protocol.

A communication style controls how the socket treats transmitted data and specifies the number of communication partners. When data is sent through a socket, it is packaged into chunks called *packets*. The communication style determines how these packets are handled and how they are addressed from the sender to the receiver.

- *Connection* styles guarantee delivery of all packets in the order they were sent. If packets are lost or reordered by problems in the network, the receiver automatically requests their retransmission from the sender.

  A connection-style socket is like a telephone call: The addresses of the sender and receiver are fixed at the beginning of the communication when the connection is established.

- *Datagram* styles do not guarantee delivery or arrival order. Packets may be lost or reordered in transit due to network errors or other conditions. Each packet must be labeled with its destination and is not guaranteed to be delivered. The system guarantees only "best effort," so packets may disappear or arrive in a different order than shipping.

  A datagram-style socket behaves more like postal mail. The sender specifies the receiver's address for each individual message.

A socket namespace specifies how *socket addresses* are written. A socket address identifies one end of a socket connection. For example, socket addresses in the "local namespace" are ordinary filenames. In "Internet namespace," a socket address is composed of the Internet address (also known as an *Internet Protocol address* or *IP address*) of a host attached to the network and a port number. The port number distinguishes among multiple sockets on the same host.

A protocol specifies how data is transmitted. Some protocols are TCP/IP, the primary networking protocols used by the Internet; the AppleTalk network protocol; and the UNIX local communication protocol. Not all combinations of styles, namespaces, and protocols are supported.

5.5.2 System Calls

Sockets are more flexible than previously discussed communication techniques. These are the system calls involving sockets:

`socket`—Creates a socket

`closes`—Destroys a socket

`connect`—Creates a connection between two sockets

`bind`—Labels a server socket with an address

`listen`—Configures a socket to accept conditions

`accept`—Accepts a connection and creates a new socket for the connection

Sockets are represented by file descriptors.

Creating and Destroying Sockets

The `socket` and `close` functions create and destroy sockets, respectively. When you create a socket, specify the three socket choices: namespace, communication style, and protocol. For the namespace parameter, use constants beginning with `PF_` (abbreviating "protocol families"). For example, `PF_LOCAL` or `PF_UNIX` specifies the local namespace, and `PF_INET` specifies Internet namespaces. For the communication style parameter, use constants beginning with `SOCK_`. Use `SOCK_STREAM` for a connection-style socket, or use `SOCK_DGRAM` for a datagram-style socket.

The third parameter, the protocol, specifies the low-level mechanism to transmit and receive data. Each protocol is valid for a particular namespace-style combination. Because there is usually one best protocol for each such pair, specifying 0 is usually the correct protocol. If `socket` succeeds, it returns a file descriptor for the socket. You can read from or write to the socket using `read`, `write`, and so on, as with other file descriptors. When you are finished with a socket, call `close` to remove it.

Calling *connect*

To create a connection between two sockets, the client calls `connect`, specifying the address of a server socket to connect to. A *client* is the process initiating the connection, and a *server* is the process waiting to accept connections. The client calls `connect` to initiate a connection from a local socket to the server socket specified by the second argument. The third argument is the length, in bytes, of the address structure pointed to by the second argument. Socket address formats differ according to the socket namespace.

Sending Information

Any technique to write to a file descriptor can be used to write to a socket. See Appendix B for a discussion of Linux's low-level I/O functions and some of the issues surrounding their use. The `send` function, which is specific to the socket file descriptors, provides an alternative to `write` with a few additional choices; see the man page for information.

5.5.3 Servers

A server's life cycle consists of the creation of a connection-style socket, binding an address to its socket, placing a call to `listen` that enables connections to the socket, placing calls to `accept` incoming connections, and then closing the socket. Data isn't read and written directly via the server socket; instead, each time a program accepts a new connection, Linux creates a separate socket to use in transferring data over that connection. In this section, we introduce `bind`, `listen`, and `accept`.

An address must be bound to the server's socket using `bind` if a client is to find it. Its first argument is the socket file descriptor. The second argument is a pointer to a socket address structure; the format of this depends on the socket's address family. The third argument is the length of the address structure, in bytes. When an address is bound to a connection-style socket, it must invoke `listen` to indicate that it is a server. Its first argument is the socket file descriptor. The second argument specifies how many pending connections are queued. If the queue is full, additional connections will be rejected. This does not limit the total number of connections that a server can handle; it limits just the number of clients attempting to connect that have not yet been accepted.

A server accepts a connection request from a client by invoking `accept`. The first argument is the socket file descriptor. The second argument points to a socket address structure, which is filled with the client socket's address. The third argument is the length, in bytes, of the socket address structure. The server can use the client address to determine whether it really wants to communicate with the client. The call to accept creates a new socket for communicating with the client and returns the corresponding file descriptor. The original server socket continues to accept new client connections. To read data from a socket without removing it from the input queue, use `recv`. It takes the same arguments as `read`, plus an additional `FLAGS` argument. A flag of `MSG_PEEK` causes data to be read but not removed from the input queue.

5.5.4 Local Sockets

Sockets connecting processes on the same computer can use the local namespace represented by the synonyms `PF_LOCAL` and `PF_UNIX`. These are called *local sockets* or *UNIX-domain sockets*. Their socket addresses, specified by filenames, are used only when creating connections.

The socket's name is specified in `struct sockaddr_un`. You must set the `sun_family` field to `AF_LOCAL`, indicating that this is a local namespace. The `sun_path` field specifies the filename to use and may be, at most, 108 bytes long. The actual length of `struct sockaddr_un` should be computed using the `SUN_LEN` macro. Any filename can be used, but the process must have directory write permissions, which permit adding files to the directory. To connect to a socket, a process must have read permission for the file. Even though different computers may share the same filesystem, only processes running on the same computer can communicate with local namespace sockets.

The only permissible protocol for the local namespace is 0.

Because it resides in a file system, a local socket is listed as a file. For example, notice the initial `s`:

```
% ls -l /tmp/socket
srwxrwx--x    1 user   group    0 Nov 13 19:18 /tmp/socket
```

Call `unlink` to remove a local socket when you're done with it.

5.5.5 An Example Using Local Namespace Sockets

We illustrate sockets with two programs. The server program, in Listing 5.10, creates a local namespace socket and listens for connections on it. When it receives a connection, it reads text messages from the connection and prints them until the connection closes. If one of these messages is "quit," the server program removes the socket and ends. The `socket-server` program takes the path to the socket as its command-line argument.

Listing 5.10 **(*socket-server.c*) Local Namespace Socket Server**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

/* Read text from the socket and print it out.  Continue until the
   socket closes.  Return nonzero if the client sent a "quit"
   message, zero otherwise.  */

int server (int client_socket)
{
  while (1) {
    int length;
    char* text;

    /* First, read the length of the text message from the socket.  If
       read returns zero, the client closed the connection.  */
    if (read (client_socket, &length, sizeof (length)) == 0)
      return 0;
    /* Allocate a buffer to hold the text.  */
    text = (char*) malloc (length);
    /* Read the text itself, and print it.  */

    read (client_socket, text, length);
    printf ("%s\n", text);
    /* Free the buffer.  */
    free (text);
    /* If the client sent the message "quit," we're all done.  */
    if (!strcmp (text, "quit"))
      return 1;
  }
}

int main (int argc, char* const argv[])
{
const char* const socket_name = argv[1];
```

```
  int socket_fd;
  struct sockaddr_un name;
  int client_sent_quit_message;

  /* Create the socket.  */
  socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);
  /* Indicate that this is a server.  */
  name.sun_family = AF_LOCAL;
  strcpy (name.sun_path, socket_name);
  bind (socket_fd, &name, SUN_LEN (&name));
  /* Listen for connections.  */
  listen (socket_fd, 5);

  /* Repeatedly accept connections, spinning off one server() to deal
     with each client.  Continue until a client sends a "quit" message.  */
  do {
    struct sockaddr_un client_name;
    socklen_t client_name_len;
    int client_socket_fd;

    /* Accept a connection.  */
    client_socket_fd = accept (socket_fd, &client_name, &client_name_len);
    /* Handle the connection.  */
    client_sent_quit_message = server (client_socket_fd);
    /* Close our end of the connection.  */
    close (client_socket_fd);
  }
  while (!client_sent_quit_message);

  /* Remove the socket file.  */
  close (socket_fd);
  unlink (socket_name);

  return 0;
}
```

The client program, in Listing 5.11, connects to a local namespace socket and sends a message. The name path to the socket and the message are specified on the command line.

Listing 5.11 **(*socket-client.c*) Local Namespace Socket Client**

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

continues

Listing 5.11 **Continued**

```
/* Write TEXT to the socket given by file descriptor SOCKET_FD.  */

void write_text (int socket_fd, const char* text)
{
  /* Write the number of bytes in the string, including
     NUL-termination.  */
  int length = strlen (text) + 1;
  write (socket_fd, &length, sizeof (length));
  /* Write the string.  */
  write (socket_fd, text, length);
}

int main (int argc, char* const argv[])
{
  const char* const socket_name = argv[1];
  const char* const message = argv[2];
  int socket_fd;
  struct sockaddr_un name;

  /* Create the socket.  */
  socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);
  /* Store the server's name in the socket address.  */
  name.sun_family = AF_LOCAL;
  strcpy (name.sun_path, socket_name);
  /* Connect the socket.  */
  connect (socket_fd, &name, SUN_LEN (&name));
  /* Write the text on the command line to the socket.  */
  write_text (socket_fd, message);
  close (socket_fd);
  return 0;
}
```

Before the client sends the message text, it sends the length of that text by sending the bytes of the integer variable `length`. Likewise, the server reads the length of the text by reading from the socket into an integer variable. This allows the server to allocate an appropriately sized buffer to hold the message text before reading it from the socket.

To try this example, start the server program in one window. Specify a path to a socket—for example, `/tmp/socket`.

```
% ./socket-server /tmp/socket
```

In another window, run the client a few times, specifying the same socket path plus messages to send to the client:

```
% ./socket-client /tmp/socket "Hello, world."
% ./socket-client /tmp/socket "This is a test."
```

The server program receives and prints these messages. To close the server, send the message "quit" from a client:

```
% ./socket-client /tmp/socket "quit"
```

The server program terminates.

5.5.6 Internet-Domain Sockets

UNIX-domain sockets can be used only for communication between two processes on the same computer. *Internet-domain sockets*, on the other hand, may be used to connect processes on different machines connected by a network.

Sockets connecting processes through the Internet use the Internet namespace represented by `PF_INET`. The most common protocols are TCP/IP. The *Internet Protocol (IP)*, a low-level protocol, moves packets through the Internet, splitting and rejoining the packets, if necessary. It guarantees only "best-effort" delivery, so packets may vanish or be reordered during transport. Every participating computer is specified using a unique IP number. The *Transmission Control Protocol (TCP)*, layered on top of IP, provides reliable connection-ordered transport. It permits telephone-like connections to be established between computers and ensures that data is delivered reliably and in order.

> **DNS Names**
>
> Because it is easier to remember names than numbers, the *Domain Name Service (DNS)* associates names such as `www.codesourcery.com` with computers' unique IP numbers. DNS is implemented by a worldwide hierarchy of name servers, but you don't need to understand DNS protocols to use Internet host names in your programs.

Internet socket addresses contain two parts: a machine and a port number. This information is stored in a `struct sockaddr_in` variable. Set the `sin_family` field to `AF_INET` to indicate that this is an Internet namespace address. The `sin_addr` field stores the Internet address of the desired machine as a 32-bit integer IP number. A *port number* distinguishes a given machine's different sockets. Because different machines store multibyte values in different byte orders, use `htons` to convert the port number to *network byte order*. See the man page for `ip` for more information.

To convert human-readable hostnames, either numbers in standard dot notation (such as 10.0.0.1) or DNS names (such as `www.codesourcery.com`) into 32-bit IP numbers, you can use `gethostbyname`. This returns a pointer to the `struct hostent` structure; the `h_addr` field contains the host's IP number. See the sample program in Listing 5.12.

Listing 5.12 illustrates the use of Internet-domain sockets. The program obtains the home page from the Web server whose hostname is specified on the command line.

Listing 5.12 **(*socket-inet.c*) Read from a WWW Server**

```
#include <stdlib.h>
#include <stdio.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#include <unistd.h>
#include <string.h>

/* Print the contents of the home page for the server's socket.
   Return an indication of success.  */

void get_home_page (int socket_fd)
{
  char buffer[10000];
  ssize_t number_characters_read;

  /* Send the HTTP GET command for the home page.  */
  sprintf (buffer, "GET /\n");
  write (socket_fd, buffer, strlen (buffer));
  /* Read from the socket.  The call to read may not
  return all the data at one time, so keep
  trying until we run out.  */
  while (1) {
    number_characters_read = read (socket_fd, buffer, 10000);
    if (number_characters_read == 0)
      return;
    /* Write the data to standard output.  */
    fwrite (buffer, sizeof (char), number_characters_read, stdout);
  }
}

int main (int argc, char* const argv[])
{
  int socket_fd;
  struct sockaddr_in name;
  struct hostent* hostinfo;

  /* Create the socket.  */
  socket_fd = socket (PF_INET, SOCK_STREAM, 0);
  /* Store the server's name in the socket address.  */
  name.sin_family = AF_INET;
  /* Convert from strings to numbers.  */
  hostinfo = gethostbyname (argv[1]);
  if (hostinfo == NULL)
    return 1;
  else
    name.sin_addr = *((struct in_addr *) hostinfo->h_addr);
  /* Web servers use port 80.  */
  name.sin_port = htons (80);
```

```
  /* Connect to the Web server  */
  if (connect (socket_fd, &name, sizeof (struct sockaddr_in)) == -1) {
    perror ("connect");
    return 1;
  }
  /* Retrieve the server's home page.  */
  get_home_page (socket_fd);

  return 0;
}
```

This program takes the hostname of the Web server on the command line (not a URL—that is, without the "http://"). It calls `gethostbyname` to translate the hostname into a numerical IP address and then connects a stream (TCP) socket to port 80 on that host. Web servers speak the *Hypertext Transport Protocol (HTTP)*, so the program issues the HTTP `GET` command and the server responds by sending the text of the home page.

> **Standard Port Numbers**
>
> By convention, Web servers listen for connections on port 80. Most Internet network services are associated with a standard port number. For example, secure Web servers that use SSL listen for connections on port 443, and mail servers (which speak SMTP) use port 25.
>
> On GNU/Linux systems, the associations between protocol/service names and standard port numbers are listed in the file `/etc/services`. The first column is the protocol or service name. The second column lists the port number and the connection type: `tcp` for connection-oriented, or `udp` for datagram.
>
> If you implement custom network services using Internet-domain sockets, use port numbers greater than 1024.

For example, to retrieve the home page from the Web site `www.codesourcery.com`, invoke this:

```
% ./socket-inet www.codesourcery.com
<html>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">

...
```

5.5.7 Socket Pairs

As we saw previously, the `pipe` function creates two file descriptors for the beginning and end of a pipe. Pipes are limited because the file descriptors must be used by related processes and because communication is unidirectional. The `socketpair` function creates two file descriptors for two connected sockets on the same computer. These file descriptors permit two-way communication between related processes.

Its first three parameters are the same as those of the `socket` call: They specify the domain, connection style, and protocol. The last parameter is a two-integer array, which is filled with the file descriptions of the two sockets, similar to `pipe`. When you call `socketpair`, you must specify `PF_LOCAL` as the domain.