B Low-Level I/O

C PROGRAMMERS ON GNU/LINUX HAVE TWO SETS OF INPUT/OUTPUT functions at their disposal. The standard C library provides I/O functions: `printf`, `fopen`, and so on.[1] The Linux kernel itself provides another set of I/O operations that operate at a lower level than the C library functions.

Because this book is for people who already know the C language, we'll assume that you have encountered and know how to use the C library I/O functions.

Often there are good reasons to use Linux's low-level I/O functions. Many of these are kernel system calls[2] and provide the most direct access to underlying system capabilities that is available to application programs. In fact, the standard C library I/O routines are implemented on top of the Linux low-level I/O system calls. Using the latter is usually the most efficient way to perform input and output operations—and is sometimes more convenient, too.

1. The C++ standard library provides *iostreams* with similar functionality. The standard C library is also available in the C++ language.

2. See Chapter 8, "Linux System Calls," for an explanation of the difference between a system call and an ordinary function call.

Throughout this book, we assume that you're familiar with the calls described in this appendix. You may already be familiar with them because they're nearly the same as those provided on other UNIX and UNIX-like operating systems (and on the Win32 platform as well). If you're not familiar with them, however, read on; you'll find the rest of the book much easier to understand if you familiarize yourself with this material first.

B.1 Reading and Writing Data

The first I/O function you likely encountered when you first learned the C language was `printf`. This formats a text string and then prints it to standard output. The generalized version, `fprintf`, can print the text to a stream other than standard output. A stream is represented by a `FILE*` pointer. You obtain a `FILE*` pointer by opening a file with `fopen`. When you're done, you can close it with `fclose`. In addition to `fprintf`, you can use such functions as `fputc`, `fputs`, and `fwrite` to write data to the stream, or `fscanf`, `fgetc`, `fgets`, and `fread` to read data.

With the Linux low-level I/O operations, you use a handle called a *file descriptor* instead of a `FILE*` pointer. A file descriptor is an integer value that refers to a particular instance of an open file in a single process. It can be open for reading, for writing, or for both reading and writing. A file descriptor doesn't have to refer to an open file; it can represent a connection with another system component that is capable of sending or receiving data. For example, a connection to a hardware device is represented by a file descriptor (see Chapter 6, "Devices"), as is an open socket (see Chapter 5, "Interprocess Communication," Section 5.5, "Sockets") or one end of a pipe (see Section 5.4, "Pipes").

Include the header files `<fcntl.h>`, `<sys/types.h>`, `<sys/stat.h>`, and `<unistd.h>` if you use any of the low-level I/O functions described here.

B.1.1 Opening a File

To open a file and produce a file descriptor that can access that file, use the `open` call. It takes as arguments the path name of the file to open, as a character string, and flags specifying how to open it. You can use `open` to create a new file; if you do, pass a third argument that specifies the access permissions to set for the new file.

If the second argument is `O_RDONLY`, the file is opened for reading only; an error will result if you subsequently try to write to the resulting file descriptor. Similarly, `O_WRONLY` causes the file descriptor to be write-only. Specifying `O_RDWR` produces a file descriptor that can be used both for reading and for writing. Note that not all files may be opened in all three modes. For instance, the permissions on a file might forbid a particular process from opening it for reading or for writing; a file on a read-only device such as a CD-ROM drive may not be opened for writing.

You can specify additional options by using the bitwise or of this value with one or more flags. These are the most commonly used values:

- Specify `O_TRUNC` to truncate the opened file, if it previously existed. Data written to the file descriptor will replace previous contents of the file.
- Specify `O_APPEND` to append to an existing file. Data written to the file descriptor will be added to the end of the file.
- Specify `O_CREAT` to create a new file. If the filename that you provide to `open` does not exist, a new file will be created, provided that the directory containing it exists and that the process has permission to create files in that directory. If the file already exists, it is opened instead.
- Specify `O_EXCL` with `O_CREAT` to force creation of a new file. If the file already exists, the `open` call will fail.

If you call `open` with `O_CREAT`, provide an additional third argument specifying the permissions for the new file. See Chapter 10, "Security," Section 10.3, "File System Permissions," for a description of permission bits and how to use them.

For example, the program in Listing B.1 creates a new file with the filename specified on the command line. It uses the `O_EXCL` flag with `open`, so if the file already exists, an error occurs. The new file is given read and write permissions for the owner and owning group, and read permissions only for others. (If your umask is set to a nonzero value, the actual permissions may be more restrictive.)

Umasks

When you create a new file with `open`, some permission bits that you specify may be turned off. This is because your umask is set to a nonzero value. A process's umask specifies bits that are masked out of all newly created files' permissions. The actual permissions used are the bitwise and of the permissions you specify to `open` and the bitwise complement of the umask.

To change your umask from the shell, use the `umask` command, and specify the numerical value of the mask, in octal notation. To change the umask for a running process, use the `umask` call, passing it the desired mask value to use for subsequent `open` calls.

For example, calling this line

```
umask (S_IRWXO | S_IWGRP);
```

in a program, or invoking this command

```
% umask 027
```

specifies that write permissions for group members and read, write, and execute permissions for others will always be masked out of a new file's permissions.

Listing B.1 **(*create-file.c*) Create a New File**

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

int main (int argc, char* argv[])
{
  /* The path at which to create the new file.  */
  char* path = argv[1];
  /* The permissions for the new file.  */
  mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH;

  /* Create the file.  */
  int fd = open (path, O_WRONLY | O_EXCL | O_CREAT, mode);
  if (fd == -1) {
    /* An error occurred.  Print an error message and bail.  */
    perror ("open");
    return 1;
  }

  return 0;
}
```

Here's the program in action:

```
% ./create-file testfile
% ls -l testfile
-rw-rw-r--    1 samuel   users          0 Feb  1 22:47 testfile
% ./create-file testfile
open: File exists
```

Note that the length of the new file is 0 because the program didn't write any data to it.

B.1.2 Closing File Descriptors

When you're done with a file descriptor, close it with `close`. In some cases, such as the program in Listing B.1, it's not necessary to call `close` explicitly because Linux closes all open file descriptors when a process terminates (that is, when the program ends). Of course, once you close a file descriptor, you should no longer use it.

Closing a file descriptor may cause Linux to take a particular action, depending on the nature of the file descriptor. For example, when you close a file descriptor for a network socket, Linux closes the network connection between the two computers communicating through the socket.

Linux limits the number of open file descriptors that a process may have open at a time. Open file descriptors use kernel resources, so it's good to close file descriptors when you're done with them. A typical limit is 1,024 file descriptors per process. You can adjust this limit with the `setrlimit` system call; see Section 8.5, "`getrlimit` and `setrlimit`: Resource Limits," for more information.

B.1.3 Writing Data

Write data to a file descriptor using the `write` call. Provide the file descriptor, a pointer to a buffer of data, and the number of bytes to write. The file descriptor must be open for writing. The data written to the file need not be a character string; `write` copies arbitrary bytes from the buffer to the file descriptor.

The program in Listing B.2 appends the current time to the file specified on the command line. If the file doesn't exist, it is created. This program also uses the `time`, `localtime`, and `asctime` functions to obtain and format the current time; see their respective man pages for more information.

Listing B.2 **(*timestamp.c*) Append a Timestamp to a File**

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>

/* Return a character string representing the current date and time.  */

char* get_timestamp ()
{
  time_t now = time (NULL);
  return asctime (localtime (&now));
}

int main (int argc, char* argv[])
{
  /* The file to which to append the timestamp.  */
  char* filename = argv[1];
  /* Get the current timestamp.  */
  char* timestamp = get_timestamp ();
  /* Open the file for writing.  If it exists, append to it;
     otherwise, create a new file.  */
  int fd = open (filename, O_WRONLY | O_CREAT | O_APPEND, 0666);
  /* Compute the length of the timestamp string.  */
  size_t length = strlen (timestamp);
  /* Write the timestamp to the file.  */
  write (fd, timestamp, length);
  /* All done.  */
  close (fd);
  return 0;
}
```

Here's how the `timestamp` program works:

```
% ./timestamp tsfile
% cat tsfile
Thu Feb  1 23:25:20 2001
% ./timestamp tsfile
% cat tsfile
Thu Feb  1 23:25:20 2001
Thu Feb  1 23:25:47 2001
```

Note that the first time we invoke `timestamp`, it creates the file `tsfile`, while the second time it appends to it.

The `write` call returns the number of bytes that were actually written, or -1 if an error occurred. For certain kinds of file descriptors, the number of bytes actually written may be less than the number of bytes requested. In this case, it's up to you to call `write` again to write the rest of the data. The function in Listing B.3 demonstrates how you might do this. Note that for some applications, you may have to check for special conditions in the middle of the writing operation. For example, if you're writing to a network socket, you'll have to augment this function to detect whether the network connection was closed in the middle of the write operation, and if it has, to react appropriately.

Listing B.3 **(*write-all.c*) Write All of a Buffer of Data**

```c
/* Write all of COUNT bytes from BUFFER to file descriptor FD.
   Returns -1 on error, or the number of bytes written.  */

ssize_t write_all (int fd, const void* buffer, size_t count)
{
  size_t left_to_write = count;
  while (left_to_write > 0) {
    size_t written = write (fd, buffer, count);
    if (written == -1)
      /* An error occurred; bail.  */
      return -1;
    else
      /* Keep count of how much more we need to write.  */
      left_to_write -= written;
  }
  /* We should have written no more than COUNT bytes!  */
  assert (left_to_write == 0);
  /* The number of bytes written is exactly COUNT.  */
  return count;
}
```

B.1.4 Reading Data

The corresponding call for reading data is `read`. Like `write`, it takes a file descriptor, a pointer to a buffer, and a count. The count specifies how many bytes are read from the file descriptor into the buffer. The call to `read` returns -1 on error or the number of bytes actually read. This may be smaller than the number of bytes requested, for example, if there aren't enough bytes left in the file.

Reading DOS/Windows Text Files

After reading this book, we're positive that you'll choose to write all your programs for GNU/Linux. However, your programs may occasionally need to read text files generated by DOS or Windows programs. It's important to anticipate an important difference in how text files are structured between these two platforms.

In GNU/Linux text files, each line is separated from the next with a newline character. A newline is represented by the character constant `'\n'`, which has ASCII code 10. On Windows, however, lines are separated by a two-character combination: a carriage return character (the character `'\r,'` which has ASCII code 13), followed by a newline character.

Some GNU/Linux text editors display `^M` at the end of each line when showing a Windows text file—this is the carriage return character. Emacs displays Windows text files properly but indicates them by showing `(DOS)` in the mode line at the bottom of the buffer. Some Windows editors, such as Notepad, display all the text in a GNU/Linux text file on a single line because they expect a carriage return at the end of each line. Other programs for both GNU/Linux and Windows that process text files may report mysterious errors when given as input a text file in the wrong format.

If your program reads text files generated by Windows programs, you'll probably want to replace the sequence `'\r\n'` with a single newline. Similarly, if your program writes text files that must be read by Windows programs, replace lone newline characters with `'\r\n'` combinations. You must do this whether you use the low-level I/O calls presented in this appendix or the standard C library I/O functions.

Listing B.4 provides a simple demonstration of `read`. The program prints a hexadecimal dump of the contents of the file specified on the command line. Each line displays the offset in the file and the next 16 bytes.

Listing B.4 (*hexdump.c*) **Print a Hexadecimal Dump of a File**

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

int main (int argc, char* argv[])
{
  unsigned char buffer[16];
  size_t offset = 0;
  size_t bytes_read;
```

continues

Listing B.4 **Continued**

```
  int i;

  /* Open the file for reading.  */
  int fd = open (argv[1], O_RDONLY);

  /* Read from the file, one chunk at a time.  Continue until read
     "comes up short", that is, reads less than we asked for.
     This indicates that we've hit the end of the file.  */
  do {
    /* Read the next line's worth of bytes.  */
    bytes_read = read (fd, buffer, sizeof (buffer));
    /* Print the offset in the file, followed by the bytes themselves.  */
    printf ("0x%06x : ", offset);
    for (i = 0; i < bytes_read; ++i)
      printf ("%02x ", buffer[i]);
    printf ("\n");
    /* Keep count of our position in the file.  */
    offset += bytes_read;
  }
  while (bytes_read == sizeof (buffer));

  /* All done.  */
  close (fd);
  return 0;
}
```

Here's `hexdump` in action. It's shown printing out a dump of its own executable file:

```
% ./hexdump hexdump
0x000000 : 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
0x000010 : 02 00 03 00 01 00 00 00 c0 83 04 08 34 00 00 00
0x000020 : e8 23 00 00 00 00 00 00 34 00 20 00 06 00 28 00
0x000030 : 1d 00 1a 00 06 00 00 00 34 00 00 00 34 80 04 08
...
```

Your output may be different, depending on the compiler you used to compile `hexdump` and the compilation flags you specified.

B.1.5 Moving Around a File

A file descriptor remembers its position in a file. As you read from or write to the file descriptor, its position advances corresponding to the number of bytes you read or write. Sometimes, however, you'll need to move around a file without reading or writing data. For instance, you might want to write over the middle of a file without modifying the beginning, or you might want to jump back to the beginning of a file and reread it without reopening it.

The `lseek` call enables you to reposition a file descriptor in a file. Pass it the file descriptor and two additional arguments specifying the new position.

- If the third argument is `SEEK_SET`, `lseek` interprets the second argument as a position, in bytes, from the start of the file.
- If the third argument is `SEEK_CUR`, `lseek` interprets the second argument as an offset, which may be positive or negative, from the current position.
- If the third argument is `SEEK_END`, `lseek` interprets the second argument as an offset from the end of the file. A positive value indicates a position beyond the end of the file.

The call to `lseek` returns the new position, as an offset from the beginning of the file. The type of the offset is `off_t`. If an error occurs, `lseek` returns -1. You can't use `lseek` with some types of file descriptors, such as socket file descriptors.

If you want to find the position of a file descriptor in a file without changing it, specify a 0 offset from the current position—for example:

```
off_t position = lseek (file_descriptor, 0, SEEK_CUR);
```

Linux enables you to use `lseek` to position a file descriptor beyond the end of the file. Normally, if a file descriptor is positioned at the end of a file and you write to the file descriptor, Linux automatically expands the file to make room for the new data. If you position a file descriptor beyond the end of a file and then write to it, Linux first expands the file to accommodate the "gap" that you created with the `lseek` operation and then writes to the end of it. This gap, however, does not actually occupy space on the disk; instead, Linux just makes a note of how long it is. If you later try to read from the file, it appears to your program that the gap is filled with 0 bytes.

Using this behavior of `lseek`, it's possible to create extremely large files that occupy almost no disk space. The program `lseek-huge` in Listing B.5 does this. It takes as command-line arguments a filename and a target file size, in megabytes. The program opens a new file, advances past the end of the file using `lseek`, and then writes a single 0 byte before closing the file.

Listing B.5 ***(lseek-huge.c)* Create Large Files with *lseek***

```
#include <fcntl.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  int zero = 0;
  const int megabyte = 1024 * 1024;

  char* filename = argv[1];
```

continues

Listing B.5 **Continued**

```
    size_t length = (size_t) atoi (argv[2]) * megabyte;

    /* Open a new file.  */
    int fd = open (filename, O_WRONLY | O_CREAT | O_EXCL, 0666);
    /* Jump to 1 byte short of where we want the file to end.  */
    lseek (fd, length - 1, SEEK_SET);
    /* Write a single 0 byte.  */
    write (fd, &zero, 1);
    /* All done.  */
    close (fd);

    return 0;
  }
```

Using `lseek-huge`, we'll make a 1GB (1024MB) file. Note the free space on the drive before and after the operation.

```
% df -h .
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda5             2.9G  2.1G  655M  76% /
% ./lseek-huge bigfile 1024
% ls -l bigfile
-rw-r-----    1 samuel   samuel   1073741824 Feb  5 16:29 bigfile
% df -h .
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda5             2.9G  2.1G  655M  76% /
```

No appreciable disk space is consumed, despite the enormous size of `bigfile`. Still, if we open `bigfile` and read from it, it appears to be filled with 1GB worth of 0s. For instance, we can examine its contents with the `hexdump` program of Listing B.4.

```
% ./hexdump bigfile | head -10
0x000000 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000010 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000020 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000030 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000040 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000050 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
...
```

If you run this yourself, you'll probably want to kill it with Ctrl+C, rather than watching it print out 2^{30} 0 bytes.

Note that these magic gaps in files are a special feature of the `ext2` file system that's typically used for GNU/Linux disks. If you try to use `lseek-huge` to create a file on some other type of file system, such as the `fat` or `vfat` file systems used to mount DOS and Windows partitions, you'll find that the resulting file does actually occupy the full amount of disk space.

Linux does not permit you to rewind before the start of a file with `lseek`.

B.2 *stat*

Using `open` and `read`, you can extract the contents of a file. But how about other information? For instance, invoking `ls -l` displays, for the files in the current directory, such information as the file size, the last modification time, permissions, and the owner.

The `stat` call obtains this information about a file. Call `stat` with the path to the file you're interested in and a pointer to a variable of type `struct stat`. If the call to `stat` is successful, it returns 0 and fills in the fields of the structure with information about that file; otherwise, it returns -1.

These are the most useful fields in `struct stat`:

- `st_mode` contains the file's access permissions. File permissions are explained in Section 10.3, "File System Permissions."
- In addition to the access permissions, the `st_mode` field encodes the type of the file in higher-order bits. See the text immediately following this bulleted list for instructions on decoding this information.
- `st_uid` and `st_gid` contain the IDs of the user and group, respectively, to which the file belongs. User and group IDs are described in Section 10.1, "Users and Groups."
- `st_size` contains the file size, in bytes.
- `st_atime` contains the time when this file was last accessed (read or written).
- `st_mtime` contains the time when this file was last modified.

These macros check the value of the `st_mode` field value to figure out what kind of file you've invoked `stat` on. A macro evaluates to true if the file is of that type.

`S_ISBLK (`*mode*`)`	block device
`S_ISCHR (`*mode*`)`	character device
`S_ISDIR (`*mode*`)`	directory
`S_ISFIFO (`*mode*`)`	fifo (named pipe)
`S_ISLNK (`*mode*`)`	symbolic link
`S_ISREG (`*mode*`)`	regular file
`S_ISSOCK (`*mode*`)`	socket

The `st_dev` field contains the major and minor device number of the hardware device on which this file resides. Device numbers are discussed in Chapter 6. The major device number is shifted left 8 bits; the minor device number occupies the least significant 8 bits. The `st_ino` field contains the *inode number* of this file. This locates the file in the file system.

If you call `stat` on a symbolic link, `stat` follows the link and you can obtain the information about the file that the link points to, not about the symbolic link itself. This implies that `S_ISLNK` will never be true for the result of `stat`. Use the `lstat` function if you don't want to follow symbolic links; this function obtains information about the link itself rather than the link's target. If you call `lstat` on a file that isn't a symbolic link, it is equivalent to `stat`. Calling `stat` on a broken link (a link that points to a nonexistent or inaccessible target) results in an error, while calling `lstat` on such a link does not.

If you already have a file open for reading or writing, call `fstat` instead of `stat`. This takes a file descriptor as its first argument instead of a path.

Listing B.6 presents a function that allocates a buffer large enough to hold the contents of a file and then reads the file into the buffer. The function uses `fstat` to determine the size of the buffer that it needs to allocate and also to check that the file is indeed a regular file.

Listing B.6 **(*read-file.c*) Read a File into a Buffer**

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

/* Read the contents of FILENAME into a newly allocated buffer.  The
   size of the buffer is stored in *LENGTH.  Returns the buffer, which
   the caller must free.  If FILENAME doesn't correspond to a regular
   file, returns NULL.  */

char* read_file (const char* filename, size_t* length)
{
  int fd;
  struct stat file_info;
  char* buffer;

  /* Open the file.  */
  fd = open (filename, O_RDONLY);

  /* Get information about the file.  */
  fstat (fd, &file_info);
  *length = file_info.st_size;
  /* Make sure the file is an ordinary file.  */
  if (!S_ISREG (file_info.st_mode)) {
    /* It's not, so give up.  */
    close (fd);
    return NULL;
  }
```

```
    /* Allocate a buffer large enough to hold the file's contents.  */
    buffer = (char*) malloc (*length);
    /* Read the file into the buffer.  */
    read (fd, buffer, *length);

    /* Finish up.  */
    close (fd);
    return buffer;
  }
```

B.3 Vector Reads and Writes

The `write` call takes as arguments a pointer to the start of a buffer of data and the length of that buffer. It writes a contiguous region of memory to the file descriptor. However, a program often will need to write several items of data, each residing at a different part of memory. To use `write`, the program either will have to copy the items into a single memory region, which obviously makes inefficient use of CPU cycles and memory, or will have to make multiple calls to `write`.

For some applications, multiple calls to `write` are inefficient or undesirable. For example, when writing to a network socket, two calls to `write` may cause two packets to be sent across the network, whereas the same data could be sent in a single packet if a single call to `write` were possible.

The `writev` call enables you to write multiple discontiguous regions of memory to a file descriptor in a single operation. This is called a *vector write*. The cost of using `writev` is that you must set up a data structure specifying the start and length of each region of memory. This data structure is an array of `struct iovec` elements. Each element specifies one region of memory to write; the fields `iov_base` and `iov_len` specify the address of the start of the region and the length of the region, respectively. If you know ahead of time how many regions you'll need, you can simply declare a `struct iovec` array variable; if the number of regions can vary, you must allocate the array dynamically.

Call `writev` passing a file descriptor to write to, the `struct iovec` array, and the number of elements in the array. The return value is the total number of bytes written.

The program in Listing B.7 writes its command-line arguments to a file using a single `writev` call. The first argument is the name of the file; the second and subsequent arguments are written to the file of that name, one on each line. The program allocates an array of `struct iovec` elements that is twice as long as the number of arguments it is writing—for each argument it writes the text of the argument itself as well as a new line character. Because we don't know the number of arguments in advance, the array is allocated using `malloc`.

Listing B.7 (*write-args.c*) **Write the Argument List to a File with writev**

```
#include <fcntl.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  int fd;
  struct iovec* vec;
  struct iovec* vec_next;
  int i;
  /* We'll need a "buffer" containing a newline character.  Use an
     ordinary char variable for this.  */
  char newline = '\n';
  /* The first command-line argument is the output filename.  */
  char* filename = argv[1];
  /* Skip past the first two elements of the argument list.  Element
     0 is the name of this program, and element 1 is the output
     filename.  */
  argc -= 2;
  argv += 2;

  /* Allocate an array of iovec elements.  We'll need two for each
     element of the argument list, one for the text itself, and one for
     a newline.  */
  vec = (struct iovec*) malloc (2 * argc * sizeof (struct iovec));

  /* Loop over the argument list, building the iovec entries.  */
  vec_next = vec;
  for (i = 0; i < argc; ++i) {
    /* The first element is the text of the argument itself.  */
    vec_next->iov_base = argv[i];
    vec_next->iov_len = strlen (argv[i]);
    ++vec_next;
    /* The second element is a single newline character.  It's okay for
       multiple elements of the struct iovec array to point to the
       same region of memory.  */
    vec_next->iov_base = &newline;
    vec_next->iov_len = 1;
    ++vec_next;
  }

  /* Write the arguments to a file.  */
  fd = open (filename, O_WRONLY | O_CREAT);
  writev (fd, vec, 2 * argc);
  close (fd);

  free (vec);
  return 0;
}
```

Here's an example of running `write-args`.

```
% ./write-args outputfile "first arg" "second arg" "third arg"
% cat outputfile
first arg
second arg
third arg
```

Linux provides a corresponding function `readv` that reads in a single operation into multiple discontiguous regions of memory. Similar to `writev`, an array of `struct iovec` elements specifies the memory regions into which the data will be read from the file descriptor.

B.4 Relation to Standard C Library I/O Functions

We mentioned earlier that the standard C library I/O functions are implemented on top of these low-level I/O functions. Sometimes, though, it's handy to use standard library functions with file descriptors, or to use low-level I/O functions on a standard library `FILE*` stream. GNU/Linux enables you to do both.

If you've opened a file using `fopen`, you can obtain the underlying file descriptor using the `fileno` function. This takes a `FILE*` argument and returns the file descriptor. For example, to open a file with the standard library `fopen` call but write to it with `writev`, you could use this code:

```
FILE* stream = fopen (filename, "w");
int file_descriptor = fileno (stream);
writev (file_descriptor, vector, vector_length);
```

Note that `stream` and `file_descriptor` correspond to the same opened file. If you call this line, you may no longer write to `file_descriptor`:

```
fclose (stream);
```

Similarly, if you call this line, you may no longer write to `stream`:

```
close (file_descriptor);
```

To go the other way, from a file descriptor to a stream, use the `fdopen` function. This constructs a `FILE*` stream pointer corresponding to a file descriptor. The `fdopen` function takes a file descriptor argument and a string argument specifying the mode in which to create the stream. The syntax of the mode argument is the same as that of the second argument to `fopen`, and it must be compatible with the file descriptor. For example, specify a mode of `r` for a read file descriptor or `w` for a write file descriptor. As with `fileno`, the stream and file descriptor refer to the same open file, so if you close one, you may not subsequently use the other.

B.5 Other File Operations

A few other operations on files and directories come in handy:

- `getcwd` obtains the current working directory. It takes two arguments, a `char` buffer and the length of the buffer. It copies the path of the current working directory into the buffer.
- `chdir` changes the current working directory to the path provided as its argument.
- `mkdir` creates a new directory. Its first argument is the path of the new directory. Its second argument is the access permissions to use for the new file. The interpretation of the permissions are the same as that of the third argument to `open` and are modified by the process's umask.
- `rmdir` deletes a directory. Its argument is the directory's path.
- `unlink` deletes a file. Its argument is the path to the file. This call can also be used to delete other file system objects, such as named pipes (see Section 5.4.5, "FIFOs") or devices (see Chapter 6).

  Actually, `unlink` doesn't necessarily delete the file's contents. As its name implies, it unlinks the file from the directory containing it. The file is no longer listed in that directory, but if any process holds an open file descriptor to the file, the file's contents are not removed from the disk. Only when no process has an open file descriptor are the file's contents deleted. So, if one process opens a file for reading or writing and then a second process unlinks the file and creates a new file with the same name, the first process sees the old contents of the file rather than the new contents (unless it closes the file and reopens it).
- `rename` renames or moves a file. Its two arguments are the old path and the new path for the file. If the paths are in different directories, `rename` moves the file, as long as both are on the same file system. You can use `rename` to move directories or other file system objects as well.

B.6 Reading Directory Contents

GNU/Linux provides functions for reading the contents of directories. Although these aren't directly related to the low-level I/O functions described in this appendix, we present them here anyway because they're often useful in application programs.

To read the contents of a directory, follow these steps:

1. Call `opendir`, passing the path of the directory that you want to examine. The call to `opendir` returns a `DIR*` handle, which you'll use to access the directory contents. If an error occurs, the call returns NULL.
2. Call `readdir` repeatedly, passing the `DIR*` handle that you obtained from `opendir`. Each time you call `readdir`, it returns a pointer to a `struct dirent` instance corresponding to the next directory entry. When you reach the end of the directory's contents, `readdir` returns NULL.

   The `struct dirent` that you get back from `readdir` has a field `d_name`, which contains the name of the directory entry.
3. Call `closedir`, passing the `DIR*` handle, to end the directory listing operation.

Include `<sys/types.h>` and `<dirent.h>` if you use these functions in your program.

Note that if you need the contents of the directory arranged in a particular order, you'll have to sort them yourself.

The program in Listing B.8 prints out the contents of a directory. The directory may be specified on the command line, but if it is not specified, the program uses the current working directory. For each entry in the directory, it displays the type of the entry and its path. The `get_file_type` function uses `lstat` to determine the type of a file system entry.

Listing B.8 ***(listdir.c)* Print a Directory Listing**

```
#include <assert.h>
#include <dirent.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

/* Return a string that describes the type of the file system entry PATH.  */

const char* get_file_type (const char* path)
{
  struct stat st;
  lstat (path, &st);
  if (S_ISLNK (st.st_mode))
    return "symbolic link";
  else if (S_ISDIR (st.st_mode))
    return "directory";
  else if (S_ISCHR (st.st_mode))
    return "character device";
  else if (S_ISBLK (st.st_mode))
    return "block device";
```

continues

Listing B.8 **Continued**

```
  else if (S_ISFIFO (st.st_mode))
    return "fifo";
  else if (S_ISSOCK (st.st_mode))
    return "socket";
  else if (S_ISREG (st.st_mode))
    return "regular file";
  else
    /* Unexpected.  Each entry should be one of the types above.  */
    assert (0);
}

int main (int argc, char* argv[])
{
  char* dir_path;
  DIR* dir;
  struct dirent* entry;
  char entry_path[PATH_MAX + 1];
  size_t path_len;

  if (argc >= 2)
    /* If a directory was specified on the command line, use it.  */
    dir_path = argv[1];
  else
    /* Otherwise, use the current directory.  */
    dir_path = ".";
  /* Copy the directory path into entry_path.  */
  strncpy (entry_path, dir_path, sizeof (entry_path));
  path_len = strlen (dir_path);
  /* If the directory path doesn't end with a slash, append a slash.  */
  if (entry_path[path_len - 1] != '/') {
    entry_path[path_len] = '/';
    entry_path[path_len + 1] = '\0';
    ++path_len;
  }

  /* Start the listing operation of the directory specified on the
     command line.  */
  dir = opendir (dir_path);
  /* Loop over all directory entries.  */
  while ((entry = readdir (dir)) != NULL) {
    const char* type;
    /* Build the path to the directory entry by appending the entry
       name to the path name.  */
    strncpy (entry_path + path_len, entry->d_name,
             sizeof (entry_path) - path_len);
    /* Determine the type of the entry.  */
    type = get_file_type (entry_path);
    /* Print the type and path of the entry.  */
    printf ("%-18s: %s\n", type, entry_path);
  }
```

```
    /* All done.  */
    closedir (dir);
    return 0;
  }
```

Here are the first few lines of output from listing the `/dev` directory. (Your output might differ somewhat.)

```
% ./listdir /dev
directory         : /dev/.
directory         : /dev/..
socket            : /dev/log
character device  : /dev/null
regular file      : /dev/MAKEDEV
fifo              : /dev/initctl
character device  : /dev/agpgart
...
```

To verify this, you can use the `ls` command on the same directory. Specify the `-U` flag to instruct `ls` not to sort the entries, and specify the `-a` flag to cause the current directory (`.`) and the parent directory (`..`) to be included.

```
% ls -lUa /dev
total 124
drwxr-xr-x    7 root     root        36864 Feb  1 15:14 .
drwxr-xr-x   22 root     root         4096 Oct 11 16:39 ..
srw-rw-rw-    1 root     root            0 Dec 18 01:31 log
crw-rw-rw-    1 root     root       1,   3 May  5  1998 null
-rwxr-xr-x    1 root     root        26689 Mar  2  2000 MAKEDEV
prw-------    1 root     root            0 Dec 11 18:37 initctl
crw-rw-r--    1 root     root      10, 175 Feb  3  2000 agpgart
...
```

The first character of each line in the output of `ls` indicates the type of the entry.