2 Writing Good GNU/Linux Software

THIS CHAPTER COVERS SOME BASIC TECHNIQUES THAT MOST GNU/Linux programmers use. By following the guidelines presented, you'll be able to write programs that work well within the GNU/Linux environment and meet GNU/Linux users' expectations of how programs should operate.

2.1 Interaction With the Execution Environment

When you first studied C or C++, you learned that the special `main` function is the primary entry point for a program. When the operating system executes your program, it automatically provides certain facilities that help the program communicate with the operating system and the user. You probably learned about the two parameters to `main`, usually called `argc` and `argv`, which receive inputs to your program. You learned about the `stdout` and `stdin` (or the `cout` and `cin` streams in C++) that provide console input and output. These features are provided by the C and C++ languages, and they interact with the GNU/Linux system in certain ways. GNU/Linux provides other ways for interacting with the operating environment, too.

2.1.1 The Argument List

You run a program from a shell prompt by typing the name of the program. Optionally, you can supply additional information to the program by typing one or more words after the program name, separated by spaces. These are called *command-line arguments*. (You can also include an argument that contains a space, by enclosing the argument in quotes.) More generally, this is referred to as the program's *argument list* because it need not originate from a shell command line. In Chapter 3, "Processes," you'll see another way of invoking a program, in which a program can specify the argument list of another program directly.

When a program is invoked from the shell, the argument list contains the entire command line, including the name of the program and any command-line arguments that may have been provided. Suppose, for example, that you invoke the `ls` command in your shell to display the contents of the root directory and corresponding file sizes with this command line:

```
% ls -s /
```

The argument list that the `ls` program receives has three elements. The first one is the name of the program itself, as specified on the command line, namely `ls`. The second and third elements of the argument list are the two command-line arguments, `-s` and `/`.

The `main` function of your program can access the argument list via the `argc` and `argv` parameters to `main` (if you don't use them, you may simply omit them). The first parameter, `argc`, is an integer that is set to the number of items in the argument list. The second parameter, `argv`, is an array of character pointers. The size of the array is `argc`, and the array elements point to the elements of the argument list, as NUL-terminated character strings.

Using command-line arguments is as easy as examining the contents of `argc` and `argv`. If you're not interested in the name of the program itself, don't forget to skip the first element.

Listing 2.1 demonstrates how to use `argc` and `argv`.

Listing 2.1 **(*arglist.c*) Using *argc* and *argv***

```
#include <stdio.h>

int main (int argc, char* argv[])
{
  printf ("The name of this program is '%s'.\n", argv[0]);
  printf ("This program was invoked with %d arguments.\n", argc - 1);

  /* Were any command-line arguments specified?  */
  if (argc > 1) {
    /* Yes, print them.  */
    int i;
    printf ("The arguments are:\n");
    for (i = 1; i < argc; ++i)
```

```
      printf ("  %s\n", argv[i]);
    }

    return 0;
  }
```

2.1.2 GNU/Linux Command-Line Conventions

Almost all GNU/Linux programs obey some conventions about how command-line arguments are interpreted. The arguments that programs expect fall into two categories: *options* (or *flags*) and other arguments. Options modify how the program behaves, while other arguments provide inputs (for instance, the names of input files).

Options come in two forms:

- *Short options* consist of a single hyphen and a single character (usually a lowercase or uppercase letter). Short options are quicker to type.
- *Long options* consist of two hyphens, followed by a name made of lowercase and uppercase letters and hyphens. Long options are easier to remember and easier to read (in shell scripts, for instance).

Usually, a program provides both a short form and a long form for most options it supports, the former for brevity and the latter for clarity. For example, most programs understand the options `-h` and `--help`, and treat them identically. Normally, when a program is invoked from the shell, any desired options follow the program name immediately. Some options expect an argument immediately following. Many programs, for example, interpret the option `--output foo` to specify that output of the program should be placed in a file named `foo`. After the options, there may follow other command-line arguments, typically input files or input data.

For example, the command `ls -s /` displays the contents of the root directory. The `-s` option modifies the default behavior of `ls` by instructing it to display the size (in kilobytes) of each entry. The `/` argument tells `ls` which directory to list. The `--size` option is synonymous with `-s`, so the same command could have been invoked as `ls --size /`.

The *GNU Coding Standards* list the names of some commonly used command-line options. If you plan to provide any options similar to these, it's a good idea to use the names specified in the coding standards. Your program will behave more like other programs and will be easier for users to learn. You can view the GNU Coding Standards' guidelines for command-line options by invoking the following from a shell prompt on most GNU/Linux systems:

```
% info "(standards)User Interfaces"
```

2.1.3 Using *getopt_long*

Parsing command-line options is a tedious chore. Luckily, the GNU C library provides a function that you can use in C and C++ programs to make this job somewhat easier (although still a bit annoying). This function, `getopt_long`, understands both short and long options. If you use this function, include the header file `<getopt.h>`.

Suppose, for example, that you are writing a program that is to accept the three options shown in Table 2.1.

Table 2.1 **Example Program Options**

Short Form	**Long Form**	**Purpose**
`-h`	`--help`	Display usage summary and exit
`-o` *`filename`*	`--output` *`filename`*	Specify output filename
`-v`	`--verbose`	Print verbose messages

In addition, the program is to accept zero or more additional command-line arguments, which are the names of input files.

To use `getopt_long`, you must provide two data structures. The first is a character string containing the valid short options, each a single letter. An option that requires an argument is followed by a colon. For your program, the string `ho:v` indicates that the valid options are `-h`, `-o`, and `-v`, with the second of these options followed by an argument.

To specify the available long options, you construct an array of `struct option` elements. Each element corresponds to one long option and has four fields. In normal circumstances, the first field is the name of the long option (as a character string, without the two hyphens); the second is 1 if the option takes an argument, or 0 otherwise; the third is `NULL`; and the fourth is a character constant specifying the short option synonym for that long option. The last element of the array should be all zeros. You could construct the array like this:

```
const struct option long_options[] = {
  { "help",     0, NULL, 'h' },
  { "output",   1, NULL, 'o' },
  { "verbose",  0, NULL, 'v' },
  { NULL,       0, NULL, 0   }
};
```

You invoke the `getopt_long` function, passing it the `argc` and `argv` arguments to `main`, the character string describing short options, and the array of `struct option` elements describing the long options.

- Each time you call `getopt_long`, it parses a single option, returning the short-option letter for that option, or –1 if no more options are found.
- Typically, you'll call `getopt_long` in a loop, to process all the options the user has specified, and you'll handle the specific options in a switch statement.

- If `getopt_long` encounters an invalid option (an option that you didn't specify as a valid short or long option), it prints an error message and returns the character `?` (a question mark). Most programs will exit in response to this, possibly after displaying usage information.
- When handling an option that takes an argument, the global variable `optarg` points to the text of that argument.
- After `getopt_long` has finished parsing all the options, the global variable `optind` contains the index (into `argv`) of the first nonoption argument.

Listing 2.2 shows an example of how you might use `getopt_long` to process your arguments.

Listing 2.2 **(*getopt_long.c*) Using *getopt_long***

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

/* The name of this program.  */
const char* program_name;

/* Prints usage information for this program to STREAM (typically
   stdout or stderr), and exit the program with EXIT_CODE.  Does not
   return.  */

void print_usage (FILE* stream, int exit_code)
{
  fprintf (stream, "Usage:  %s options [ inputfile ... ]\n", program_name);
  fprintf (stream,
           "  -h  --help             Display this usage information.\n"
           "  -o  --output filename  Write output to file.\n"
           "  -v  --verbose          Print verbose messages.\n");
  exit (exit_code);
}

/* Main program entry point.  ARGC contains number of argument list
   elements; ARGV is an array of pointers to them.  */

int main (int argc, char* argv[])
{
  int next_option;

  /* A string listing valid short options letters.  */
  const char* const short_options = "ho:v";
  /* An array describing valid long options.  */
  const struct option long_options[] = {
    { "help",     0, NULL, 'h' },
    { "output",   1, NULL, 'o' },
    { "verbose",  0, NULL, 'v' },
```

continues

Listing 2.2 **Continued**

```
    { NULL,       0, NULL, 0   }   /* Required at end of array.  */
  };

  /* The name of the file to receive program output, or NULL for
     standard output.  */
  const char* output_filename = NULL;
  /* Whether to display verbose messages.  */
  int verbose = 0;

  /* Remember the name of the program, to incorporate in messages.
     The name is stored in argv[0].  */
  program_name = argv[0];

  do {
    next_option = getopt_long (argc, argv, short_options,
                               long_options, NULL);
    switch (next_option)
    {
    case 'h':   /* -h or --help */
      /* User has requested usage information.  Print it to standard
         output, and exit with exit code zero (normal termination).  */
      print_usage (stdout, 0);

    case 'o':   /* -o or --output */
      /* This option takes an argument, the name of the output file.  */
      output_filename = optarg;
      break;

    case 'v':   /* -v or  --verbose */
      verbose = 1;
      break;

    case '?':   /* The user specified an invalid option.  */
      /* Print usage information to standard error, and exit with exit
         code one (indicating abnormal termination).  */
      print_usage (stderr, 1);

    case -1:    /* Done with options.  */
      break;

    default:    /* Something else: unexpected.  */
      abort ();
    }
  }
  while (next_option != -1);

  /* Done with options.  OPTIND points to first nonoption argument.
     For demonstration purposes, print them if the verbose option was
     specified.  */
    if (verbose) {
      int i;
      for (i = optind; i < argc; ++i)
        printf ("Argument: %s\n", argv[i]);
    }

    /* The main program goes here.  */

    return 0;
  }
```

Using `getopt_long` may seem like a lot of work, but writing code to parse the command-line options yourself would take even longer. The `getopt_long` function is very sophisticated and allows great flexibility in specifying what kind of options to accept. However, it's a good idea to stay away from the more advanced features and stick with the basic option structure described.

2.1.4 Standard I/O

The standard C library provides standard input and output streams (`stdin` and `stdout`, respectively). These are used by `scanf`, `printf`, and other library functions. In the UNIX tradition, use of standard input and output is customary for GNU/Linux programs. This allows the chaining of multiple programs using shell pipes and input and output redirection. (See the man page for your shell to learn its syntax.)

The C library also provides `stderr`, the standard error stream. Programs should print warning and error messages to standard error instead of standard output. This allows users to separate normal output and error messages, for instance, by redirecting standard output to a file while allowing standard error to print on the console. The `fprintf` function can be used to print to `stderr`, for example:

```
fprintf (stderr, ("Error: ..."));
```

These three streams are also accessible with the underlying UNIX I/O commands (`read`, `write`, and so on) via file descriptors. These are file descriptors 0 for `stdin`, 1 for `stdout`, and 2 for `stderr`.

When invoking a program, it is sometimes useful to redirect both standard output and standard error to a file or pipe. The syntax for doing this varies among shells; for Bourne-style shells (including `bash`, the default shell on most GNU/Linux distributions), the syntax is this:

```
% program > output_file.txt 2>&1
% program 2>&1 | filter
```

The `2>&1` syntax indicates that file descriptor 2 (`stderr`) should be merged into file descriptor 1 (`stdout`). Note that `2>&1` must follow a file redirection (the first example) but must precede a pipe redirection (the second example).

Note that `stdout` is buffered. Data written to `stdout` is not sent to the console (or other device, if it's redirected) until the buffer fills, the program exits normally, or `stdout` is closed. You can explicitly flush the buffer by calling the following:

```
fflush (stdout);
```

In contrast, `stderr` is not buffered; data written to `stderr` goes directly to the console.[1]

This can produce some surprising results. For example, this loop does not print one period every second; instead, the periods are buffered, and a bunch of them are printed together when the buffer fills.

```
while (1) {
  printf (".");
  sleep (1);
}
```

In this loop, however, the periods do appear once a second:

```
while (1) {
  fprintf (stderr, ".");
  sleep (1);
}
```

2.1.5 Program Exit Codes

When a program ends, it indicates its status with an exit code. The exit code is a small integer; by convention, an exit code of zero denotes successful execution, while nonzero exit codes indicate that an error occurred. Some programs use different nonzero exit code values to distinguish specific errors.

With most shells, it's possible to obtain the exit code of the most recently executed program using the special `$?` variable. Here's an example in which the `ls` command is invoked twice and its exit code is printed after each invocation. In the first case, `ls` executes correctly and returns the exit code zero. In the second case, `ls` encounters an error (because the filename specified on the command line does not exist) and thus returns a nonzero exit code.

```
% ls /
bin   coda  etc   lib         misc  nfs  proc  sbin  usr
boot  dev   home  lost+found  mnt   opt  root  tmp   var
% echo $?
0
% ls bogusfile
ls: bogusfile: No such file or directory
% echo $?
1
```

1. In C++, the same distinction holds for `cout` and `cerr`, respectively. Note that the `endl` token flushes a stream in addition to printing a newline character; if you don't want to flush the stream (for performance reasons, for example), use a newline constant, `'\n'`, instead.

A C or C++ program specifies its exit code by returning that value from the `main` function. There are other methods of providing exit codes, and special exit codes are assigned to programs that terminate abnormally (by a signal). These are discussed further in Chapter 3.

2.1.6 The Environment

GNU/Linux provides each running program with an *environment*. The environment is a collection of variable/value pairs. Both environment variable names and their values are character strings. By convention, environment variable names are spelled in all capital letters.

You're probably familiar with several common environment variables already. For instance:

- `USER` contains your username.
- `HOME` contains the path to your home directory.
- `PATH` contains a colon-separated list of directories through which Linux searches for commands you invoke.
- `DISPLAY` contains the name and display number of the X Window server on which windows from graphical X Window programs will appear.

Your shell, like any other program, has an environment. Shells provide methods for examining and modifying the environment directly. To print the current environment in your shell, invoke the `printenv` program. Various shells have different built-in syntax for using environment variables; the following is the syntax for Bourne-style shells.

- The shell automatically creates a shell variable for each environment variable that it finds, so you can access environment variable values using the `$`*`varname`* syntax. For instance:

  ```
  % echo $USER
  samuel
  % echo $HOME
  /home/samuel
  ```

- You can use the `export` command to export a shell variable into the environment. For example, to set the `EDITOR` environment variable, you would use this:

  ```
  % EDITOR=emacs
  % export EDITOR
  ```

  Or, for short:

  ```
  % export EDITOR=emacs
  ```

In a program, you access an environment variable with the `getenv` function in `<stdlib.h>`. That function takes a variable name and returns the corresponding value as a character string, or `NULL` if that variable is not defined in the environment. To set or clear environment variables, use the `setenv` and `unsetenv` functions, respectively.

Enumerating all the variables in the environment is a little trickier. To do this, you must access a special global variable named `environ`, which is defined in the GNU C library. This variable, of type `char**`, is a `NULL`-terminated array of pointers to character strings. Each string contains one environment variable, in the form `VARIABLE=value`.

The program in Listing 2.3, for instance, simply prints the entire environment by looping through the `environ` array.

Listing 2.3 (***print-env.c***) **Printing the Execution Environment**

```
#include <stdio.h>

/* The ENVIRON variable contains the environment.  */
extern char** environ;

int main ()
{
  char** var;
  for (var = environ; *var != NULL; ++var)
    printf ("%s\n", *var);
  return 0;
}
```

Don't modify `environ` yourself; use the `setenv` and `unsetenv` functions instead.

Usually, when a new program is started, it inherits a copy of the environment of the program that invoked it (the shell program, if it was invoked interactively). So, for instance, programs that you run from the shell may examine the values of environment variables that you set in the shell.

Environment variables are commonly used to communicate configuration information to programs. Suppose, for example, that you are writing a program that connects to an Internet server to obtain some information. You could write the program so that the server name is specified on the command line. However, suppose that the server name is not something that users will change very often. You can use a special environment variable—say `SERVER_NAME`—to specify the server name; if that variable doesn't exist, a default value is used. Part of your program might look as shown in Listing 2.4.

Listing 2.4 (***client.c***) **Part of a Network Client Program**

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

int main ()
{
    char* server_name = getenv ("SERVER_NAME");
    if (server_name == NULL)
      /* The SERVER_NAME environment variable was not set.  Use the
         default.  */
      server_name = "server.my-company.com";

    printf ("accessing server %s\n", server_name);
    /* Access the server here...  */

    return 0;
  }
```

Suppose that this program is named `client`. Assuming that you haven't set the `SERVER_NAME` variable, the default value for the server name is used:

```
% client
accessing server server.my-company.com
```

But it's easy to specify a different server:

```
% export SERVER_NAME=backup-server.elsewhere.net
% client
accessing server backup-server.elsewhere.net
```

2.1.7 Using Temporary Files

Sometimes a program needs to make a temporary file, to store large data for a while or to pass data to another program. On GNU/Linux systems, temporary files are stored in the `/tmp` directory. When using temporary files, you should be aware of the following pitfalls:

- More than one instance of your program may be run simultaneously (by the same user or by different users). The instances should use different temporary filenames so that they don't collide.
- The file permissions of the temporary file should be set in such a way that unauthorized users cannot alter the program's execution by modifying or replacing the temporary file.
- Temporary filenames should be generated in a way that cannot be predicted externally; otherwise, an attacker can exploit the delay between testing whether a given name is already in use and opening a new temporary file.

GNU/Linux provides functions, `mkstemp` and `tmpfile`, that take care of these issues for you (in addition to several functions that don't). Which you use depends on whether you plan to hand the temporary file to another program, and whether you want to use UNIX I/O (`open`, `write`, and so on) or the C library's stream I/O functions (`fopen`, `fprintf`, and so on).

Using *mkstemp*

The `mkstemp` function creates a unique temporary filename from a filename template, creates the file with permissions so that only the current user can access it, and opens the file for read/write. The filename template is a character string ending with "XXXXXX" (six capital X's); `mkstemp` replaces the X's with characters so that the filename is unique. The return value is a file descriptor; use the `write` family of functions to write to the temporary file.

Temporary files created with `mkstemp` are not deleted automatically. It's up to you to remove the temporary file when it's no longer needed. (Programmers should be very careful to clean up temporary files; otherwise, the `/tmp` file system will fill up eventually, rendering the system inoperable.) If the temporary file is for internal use only and won't be handed to another program, it's a good idea to call `unlink` on the temporary file immediately. The `unlink` function removes the directory entry corresponding to a file, but because files in a file system are reference-counted, the file itself is not removed until there are no open file descriptors for that file, either. This way, your program may continue to use the temporary file, and the file goes away automatically as soon as you close the file descriptor. Because Linux closes file descriptors when a program ends, the temporary file will be removed even if your program terminates abnormally.

The pair of functions in Listing 2.5 demonstrates `mkstemp`. Used together, these functions make it easy to write a memory buffer to a temporary file (so that memory can be freed or reused) and then read it back later.

Listing 2.5 **(*temp_file.c*) Using *mkstemp***

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

/* A handle for a temporary file created with write_temp_file.  In
   this implementation, it's just a file descriptor.  */
typedef int temp_file_handle;

/* Writes LENGTH bytes from BUFFER into a temporary file.  The
   temporary file is immediately unlinked.  Returns a handle to the
   temporary file.  */

temp_file_handle write_temp_file (char* buffer, size_t length)
{
  /* Create the filename and file.  The XXXXXX will be replaced with
     characters that make the filename unique.  */
  char temp_filename[] = "/tmp/temp_file.XXXXXX";
  int fd = mkstemp (temp_filename);
  /* Unlink the file immediately, so that it will be removed when the
     file descriptor is closed.  */
  unlink (temp_filename);
  /* Write the number of bytes to the file first.  */
  write (fd, &length, sizeof (length));
  /* Now write the data itself.  */
  write (fd, buffer, length);
  /* Use the file descriptor as the handle for the temporary file.  */
  return fd;
}

/* Reads the contents of a temporary file TEMP_FILE created with
   write_temp_file.  The return value is a newly allocated buffer of
   those contents, which the caller must deallocate with free.
   *LENGTH is set to the size of the contents, in bytes.  The
   temporary file is removed.  */

char* read_temp_file (temp_file_handle temp_file, size_t* length)
{
  char* buffer;
  /* The TEMP_FILE handle is a file descriptor to the temporary file.  */
  int fd = temp_file;
  /* Rewind to the beginning of the file.  */
  lseek (fd, 0, SEEK_SET);
  /* Read the size of the data in the temporary file.  */
  read (fd, length, sizeof (*length));
  /* Allocate a buffer and read the data.  */
  buffer = (char*) malloc (*length);
  read (fd, buffer, *length);
  /* Close the file descriptor, which will cause the temporary file to
     go away.  */
  close (fd);
  return buffer;
}
```

Using *tmpfile*

If you are using the C library I/O functions and don't need to pass the temporary file to another program, you can use the `tmpfile` function. This creates and opens a temporary file, and returns a file pointer to it. The temporary file is already unlinked, as in the previous example, so it is deleted automatically when the file pointer is closed (with `fclose`) or when the program terminates.

GNU/Linux provides several other functions for generating temporary files and temporary filenames, including `mktemp`, `tmpnam`, and `tempnam`. Don't use these functions, though, because they suffer from the reliability and security problems already mentioned.

2.2 Coding Defensively

Writing programs that run correctly under "normal" use is hard; writing programs that behave gracefully in failure situations is harder. This section demonstrates some coding techniques for finding bugs early and for detecting and recovering from problems in a running program.

The code samples presented later in this book deliberately skip extensive error checking and recovery code because this would obscure the basic functionality being presented. However, the final example in Chapter 11, "A Sample GNU/Linux Application," comes back to demonstrating how to use these techniques to write robust programs.

2.2.1 Using *assert*

A good objective to keep in mind when coding application programs is that bugs or unexpected errors should cause the program to fail dramatically, as early as possible. This will help you find bugs earlier in the development and testing cycles. Failures that don't exhibit themselves dramatically are often missed and don't show up until the application is in users' hands.

One of the simplest methods to check for unexpected conditions is the standard C `assert` macro. The argument to this macro is a Boolean expression. The program is terminated if the expression evaluates to false, after printing an error message containing the source file and line number and the text of the expression. The `assert` macro is very useful for a wide variety of consistency checks internal to a program. For instance, use `assert` to test the validity of function arguments, to test preconditions and postconditions of function calls (and method calls, in C++), and to test for unexpected return values.

Each use of `assert` serves not only as a runtime check of a condition, but also as documentation about the program's operation within the source code. If your program contains an `assert (`*`condition`*`)` that says to someone reading your source code that *`condition`* should always be true at that point in the program, and if *`condition`* is not true, it's probably a bug in the program.

For performance-critical code, runtime checks such as uses of `assert` can impose a significant performance penalty. In these cases, you can compile your code with the `NDEBUG` macro defined, by using the `-DNDEBUG` flag on your compiler command line. With `NDEBUG` set, appearances of the `assert` macro will be preprocessed away. It's a good idea to do this only when necessary for performance reasons, though, and only with performance-critical source files.

Because it is possible to preprocess `assert` macros away, be careful that any expression you use with `assert` has no side effects. Specifically, you shouldn't call functions inside `assert` expressions, assign variables, or use modifying operators such as `++`.

Suppose, for example, that you call a function, `do_something`, repeatedly in a loop. The `do_something` function returns zero on success and nonzero on failure, but you don't expect it ever to fail in your program. You might be tempted to write:

```
for (i = 0; i < 100; ++i)
  assert (do_something () == 0);
```

However, you might find that this runtime check imposes too large a performance penalty and decide later to recompile with `NDEBUG` defined. This will remove the `assert` call entirely, so the expression will never be evaluated and `do_something` will never be called. You should write this instead:

```
for (i = 0; i < 100; ++i) {
  int status = do_something ();
  assert (status == 0);
}
```

Another thing to bear in mind is that you should not use `assert` to test for invalid user input. Users don't like it when applications simply crash with a cryptic error message, even in response to invalid input. You should still always check for invalid input and produce sensible error messages in response input. Use `assert` for internal runtime checks only.

Some good places to use `assert` are these:

- Check against null pointers, for instance, as invalid function arguments. The error message generated by {`assert (pointer !=  NULL)`},

  ```
  Assertion 'pointer != ((void *)0)' failed.
  ```

  is more informative than the error message that would result if your program dereferenced a null pointer:

  ```
  Segmentation fault (core dumped)
  ```

- Check conditions on function parameter values. For instance, if a function should be called only with a positive value for parameter `foo`, use this at the beginning of the function body:

  ```
  assert (foo > 0);
  ```

  This will help you detect misuses of the function, and it also makes it very clear to someone reading the function's source code that there is a restriction on the parameter's value.

Don't hold back; use `assert` liberally throughout your programs.

2.2.2 System Call Failures

Most of us were originally taught how to write programs that execute to completion along a well-defined path. We divide the program into tasks and subtasks, and each function completes a task by invoking other functions to perform corresponding subtasks. Given appropriate inputs, we expect a function to produce the correct output and side effects.

The realities of computer hardware and software intrude into this idealized dream. Computers have limited resources; hardware fails; many programs execute at the same time; users and programmers make mistakes. It's often at the boundary between the application and the operating system that these realities exhibit themselves. Therefore, when using system calls to access system resources, to perform I/O, or for other purposes, it's important to understand not only what happens when the call succeeds, but also how and when the call can fail.

System calls can fail in many ways. For example:

- The system can run out of resources (or the program can exceed the resource limits enforced by the system of a single program). For example, the program might try to allocate too much memory, to write too much to a disk, or to open too many files at the same time.
- Linux may block a certain system call when a program attempts to perform an operation for which it does not have permission. For example, a program might attempt to write to a file marked read-only, to access the memory of another process, or to kill another user's program.
- The arguments to a system call might be invalid, either because the user provided invalid input or because of a program bug. For instance, the program might pass an invalid memory address or an invalid file descriptor to a system call. Or, a program might attempt to open a directory as an ordinary file, or might pass the name of an ordinary file to a system call that expects a directory.
- A system call can fail for reasons external to a program. This happens most often when a system call accesses a hardware device. The device might be faulty or might not support a particular operation, or perhaps a disk is not inserted in the drive.
- A system call can sometimes be interrupted by an external event, such as the delivery of a signal. This might not indicate outright failure, but it is the responsibility of the calling program to restart the system call, if desired.

In a well-written program that makes extensive use of system calls, it is often the case that more code is devoted to detecting and handling errors and other exceptional circumstances than to the main work of the program.

2.2.3 Error Codes from System Calls

A majority of system calls return zero if the operation succeeds, or a nonzero value if the operation fails. (Many, though, have different return value conventions; for instance, `malloc` returns a null pointer to indicate failure. Always read the man page carefully when using a system call.) Although this information may be enough to determine whether the program should continue execution as usual, it probably does not provide enough information for a sensible recovery from errors.

Most system calls use a special variable named `errno` to store additional information in case of failure.[2] When a call fails, the system sets `errno` to a value indicating what went wrong. Because all system calls use the same `errno` variable to store error information, you should copy the value into another variable immediately after the failed call. The value of `errno` will be overwritten the next time you make a system call.

Error values are integers; possible values are given by preprocessor macros, by convention named in all capitals and starting with "E"—for example, `EACCES` and `EINVAL`. Always use these macros to refer to `errno` values rather than integer values. Include the `<errno.h>` header if you use `errno` values.

GNU/Linux provides a convenient function, `strerror`, that returns a character string description of an `errno` error code, suitable for use in error messages. Include `<string.h>` if you use `strerror`.

GNU/Linux also provides `perror`, which prints the error description directly to the `stderr` stream. Pass to `perror` a character string prefix to print before the error description, which should usually include the name of the function that failed. Include `<stdio.h>` if you use `perror`.

This code fragment attempts to open a file; if the open fails, it prints an error message and exits the program. Note that the `open` call returns an open file descriptor if the open operation succeeds, or –1 if the operation fails.

```
fd = open ("inputfile.txt", O_RDONLY);
if (fd == -1) {
  /* The open failed.  Print an error message and exit.  */
  fprintf (stderr, "error opening file: %s\n", strerror (errno));
  exit (1);
}
```

Depending on your program and the nature of the system call, the appropriate action in case of failure might be to print an error message, to cancel an operation, to abort the program, to try again, or even to ignore the error. It's important, though, to include logic that handles all possible failure modes in some way or another.

2. Actually, for reasons of thread safety, `errno` is implemented as a macro, but it is used like a global variable.

One possible error code that you should be on the watch for, especially with I/O functions, is `EINTR`. Some functions, such as `read`, `select`, and `sleep`, can take significant time to execute. These are considered *blocking* functions because program execution is blocked until the call is completed. However, if the program receives a signal while blocked in one of these calls, the call will return without completing the operation. In this case, `errno` is set to `EINTR`. Usually, you'll want to retry the system call in this case.

Here's a code fragment that uses the `chown` call to change the owner of a file given by `path` to the user by `user_id`. If the call fails, the program takes action depending on the value of `errno`. Notice that when we detect what's probably a bug in the program, we exit using `abort` or `assert`, which cause a core file to be generated. This can be useful for post-mortem debugging. For other unrecoverable errors, such as out-of-memory conditions, we exit using `exit` and a nonzero exit value instead because a core file wouldn't be very useful.

```
rval = chown (path, user_id, -1);
if (rval != 0) {
  /* Save errno because it's clobbered by the next system call.  */
  int error_code = errno;
  /* The operation didn't succeed; chown should return -1 on error.  */
  assert (rval == -1);
  /* Check the value of errno, and take appropriate action.  */
  switch (error_code) {
  case EPERM:         /* Permission denied.  */
  case EROFS:         /* PATH is on a read-only file system.  */
  case ENAMETOOLONG:  /* PATH is too long.  */
  case ENOENT:        /* PATH does not exit.  */
  case ENOTDIR:       /* A component of PATH is not a directory.  */
  case EACCES:        /* A component of PATH is not accessible.  */
    /* Something's wrong with the file.  Print an error message.  */
    fprintf (stderr, "error changing ownership of %s: %s\n",
             path, strerror (error_code));
    /* Don't end the program; perhaps give the user a chance to
       choose another file...  */
    break;

  case EFAULT:
    /* PATH contains an invalid memory address.  This is probably a bug.  */
    abort ();

  case ENOMEM:
    /* Ran out of kernel memory.  */
    fprintf (stderr, "%s\n", strerror (error_code));
    exit (1);

  default:
    /* Some other, unexpected, error code.  We've tried to handle all
       possible error codes; if we've missed one, that's a bug!  */
    abort ();
  };
}
```

You could simply have used this code, which behaves the same way if the call succeeds:

```
rval = chown (path, user_id, -1);
assert (rval == 0);
```

But if the call fails, this alternative makes no effort to report, handle, or recover from errors.

Whether you use the first form, the second form, or something in between depends on the error detection and recovery requirements for your program.

2.2.4 Errors and Resource Allocation

Often, when a system call fails, it's appropriate to cancel the current operation but not to terminate the program because it may be possible to recover from the error. One way to do this is to return from the current function, passing a return code to the caller indicating the error.

If you decide to return from the middle of a function, it's important to make sure that any resources successfully allocated previously in the function are first deallocated. These resources can include memory, file descriptors, file pointers, temporary files, synchronization objects, and so on. Otherwise, if your program continues running, the resources allocated before the failure occurred will be leaked.

Consider, for example, a function that reads from a file into a buffer. The function might follow these steps:

1. Allocate the buffer.
2. Open the file.
3. Read from the file into the buffer.
4. Close the file.
5. Return the buffer.

If the file doesn't exist, Step 2 will fail. An appropriate course of action might be to return `NULL` from the function. However, if the buffer has already been allocated in Step 1, there is a risk of leaking that memory. You must remember to deallocate the buffer somewhere along any flow of control from which you don't return. If Step 3 fails, not only must you deallocate the buffer before returning, but you also must close the file.

Listing 2.6 shows an example of how you might write this function.

Listing 2.6 **(*readfile.c*) Freeing Resources During Abnormal Conditions**

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

continues

Listing 2.6 **Continued**

```
char* read_from_file (const char* filename, size_t length)
{
  char* buffer;
  int fd;
  ssize_t bytes_read;

  /* Allocate the buffer.  */
  buffer = (char*) malloc (length);
  if (buffer == NULL)
    return NULL;
  /* Open the file.  */
  fd = open (filename, O_RDONLY);
  if (fd == -1) {
    /* open failed.  Deallocate buffer before returning.  */
    free (buffer);
    return NULL;
  }
  /* Read the data.  */
  bytes_read = read (fd, buffer, length);
  if (bytes_read != length) {
    /* read failed.  Deallocate buffer and close fd before returning.  */
    free (buffer);
    close (fd);
    return NULL;
  }
  /* Everything's fine.  Close the file and return the buffer.  */
  close (fd);
  return buffer;
}
```

Linux cleans up allocated memory, open files, and most other resources when a program terminates, so it's not necessary to deallocate buffers and close files before calling `exit`. You might need to manually free other shared resources, however, such as temporary files and shared memory, which can potentially outlive a program.

2.3 Writing and Using Libraries

Virtually all programs are linked against one or more libraries. Any program that uses a C function (such as `printf` or `malloc`) will be linked against the C runtime library. If your program has a graphical user interface (GUI), it will be linked against windowing libraries. If your program uses a database, the database provider will give you libraries that you can use to access the database conveniently.

In each of these cases, you must decide whether to link the library *statically* or *dynamically*. If you choose to link statically, your programs will be bigger and harder to upgrade, but probably easier to deploy. If you link dynamically, your programs will be smaller, easier to upgrade, but harder to deploy. This section explains how to link both statically and dynamically, examines the trade-offs in more detail, and gives some "rules of thumb" for deciding which kind of linking is better for you.

2.3.1 Archives

An *archive* (or static library) is simply a collection of object files stored as a single file. (An archive is roughly the equivalent of a Windows `.LIB` file.) When you provide an archive to the linker, the linker searches the archive for the object files it needs, extracts them, and links them into your program much as if you had provided those object files directly.

You can create an archive using the `ar` command. Archive files traditionally use a `.a` extension rather than the `.o` extension used by ordinary object files. Here's how you would combine `test1.o` and `test2.o` into a single `libtest.a` archive:

```
% ar cr libtest.a test1.o test2.o
```

The `cr` flags tell `ar` to create the archive.[3] Now you can link with this archive using the `-ltest` option with `gcc` or `g++`, as described in Section 1.2.2, "Linking Object Files," in Chapter 1, "Getting Started."

When the linker encounters an archive on the command line, it searches the archive for all definitions of symbols (functions or variables) that are referenced from the object files that it has already processed but not yet defined. The object files that define those symbols are extracted from the archive and included in the final executable. Because the linker searches the archive when it is encountered on the command line, it usually makes sense to put archives at the end of the command line. For example, suppose that `test.c` contains the code in Listing 2.7 and `app.c` contains the code in Listing 2.8.

Listing 2.7 **(*test.c*) Library Contents**

```
int f ()
{
  return 3;
}
```

Listing 2.8 **(*app.c*) A Program That Uses Library Functions**

```
int main ()
{
  return f ();
}
```

3. You can use other flags to remove a file from an archive or to perform other operations on the archive. These operations are rarely used but are documented on the `ar` man page.

Now suppose that `test.o` is combined with some other object files to produce the `libtest.a` archive. The following command line will not work:

```
% gcc -o app -L. -ltest app.o
app.o: In function 'main':
app.o(.text+0x4): undefined reference to 'f'
collect2: ld returned 1 exit status
```

The error message indicates that even though `libtest.a` contains a definition of `f`, the linker did not find it. That's because `libtest.a` was searched when it was first encountered, and at that point the linker hadn't seen any references to `f`.

On the other hand, if we use this line, no error messages are issued:

```
% gcc -o app app.o -L. -ltest
```

The reason is that the reference to `f` in `app.o` causes the linker to include the `test.o` object file from the `libtest.a` archive.

2.3.2 Shared Libraries

A *shared library* (also known as a shared object, or as a dynamically linked library) is similar to a archive in that it is a grouping of object files. However, there are many important differences. The most fundamental difference is that when a shared library is linked into a program, the final executable does not actually contain the code that is present in the shared library. Instead, the executable merely contains a reference to the shared library. If several programs on the system are linked against the same shared library, they will all reference the library, but none will actually be included. Thus, the library is "shared" among all the programs that link with it.

A second important difference is that a shared library is not merely a collection of object files, out of which the linker chooses those that are needed to satisfy undefined references. Instead, the object files that compose the shared library are combined into a single object file so that a program that links against a shared library always includes all of the code in the library, rather than just those portions that are needed.

To create a shared library, you must compile the objects that will make up the library using the `-fPIC` option to the compiler, like this:

```
% gcc -c -fPIC test1.c
```

The `-fPIC` option tells the compiler that you are going to be using `test.o` as part of a shared object.

Position-Independent Code (PIC)

PIC stands for position-independent code. The functions in a shared library may be loaded at different addresses in different programs, so the code in the shared object must not depend on the address (or position) at which it is loaded. This consideration has no impact on you, as the programmer, except that you must remember to use the `-fPIC` flag when compiling code that will be used in a shared library.

Then you combine the object files into a shared library, like this:

```
% gcc -shared -fPIC -o libtest.so test1.o test2.o
```

The `-shared` option tells the linker to produce a shared library rather than an ordinary executable. Shared libraries use the extension `.so`, which stands for shared object. Like static archives, the name always begins with `lib` to indicate that the file is a library.

Linking with a shared library is just like linking with a static archive. For example, the following line will link with `libtest.so` if it is in the current directory, or one of the standard library search directories on the system:

```
% gcc -o app app.o -L. –ltest
```

Suppose that both `libtest.a` and `libtest.so` are available. Then the linker must choose one of the libraries and not the other. The linker searches each directory (first those specified with `-L` options, and then those in the standard directories). When the linker finds a directory that contains either `libtest.a` or `libtest.so`, the linker stops search directories. If only one of the two variants is present in the directory, the linker chooses that variant. Otherwise, the linker chooses the shared library version, unless you explicitly instruct it otherwise. You can use the `-static` option to demand static archives. For example, the following line will use the `libtest.a` archive, even if the `libtest.so` shared library is also available:

```
% gcc -static -o app app.o -L. –ltest
```

The `ldd` command displays the shared libraries that are linked into an executable. These libraries need to be available when the executable is run. Note that `ldd` will list an additional library called `ld-linux.so`, which is a part of GNU/Linux's dynamic linking mechanism.

Using *LD_LIBRARY_PATH*

When you link a program with a shared library, the linker does not put the full path to the shared library in the resulting executable. Instead, it places only the name of the shared library. When the program is actually run, the system searches for the shared library and loads it. The system searches only `/lib` and `/usr/lib`, by default. If a shared library that is linked into your program is installed outside those directories, it will not be found, and the system will refuse to run the program.

One solution to this problem is to use the `-Wl,-rpath` option when linking the program. Suppose that you use this:

```
% gcc -o app app.o -L. -ltest -Wl,-rpath,/usr/local/lib
```

Then, when `app` is run, the system will search `/usr/local/lib` for any required shared libraries.

Another solution to this problem is to set the `LD_LIBRARY_PATH` environment variable when running the program. Like the `PATH` environment variable, `LD_LIBRARY_PATH` is a colon-separated list of directories. For example, if `LD_LIBRARY_PATH` is `/usr/local/lib:/opt/lib`, then `/usr/local/lib` and `/opt/lib` will be searched before the standard `/lib` and `/usr/lib` directories. You should also note that if you have `LD_LIBRARY_PATH`, the linker will search the directories given there in addition to the directories given with the `-L` option when it is building an executable.[4]

2.3.3 Standard Libraries

Even if you didn't specify any libraries when you linked your program, it almost certainly uses a shared library. That's because GCC automatically links in the standard C library, `libc`, for you. The standard C library math functions are not included in `libc`; instead, they're in a separate library, `libm`, which you need to specify explicitly. For example, to compile and link a program `compute.c` which uses trigonometric functions such as `sin` and `cos`, you must invoke this code:

```
% gcc -o compute compute.c -lm
```

If you write a C++ program and link it using the `c++` or `g++` commands, you'll also get the standard C++ library, `libstdc++`, automatically.

2.3.4 Library Dependencies

One library will often depend on another library. For example, many GNU/Linux systems include `libtiff`, a library that contains functions for reading and writing image files in the TIFF format. This library, in turn, uses the libraries `libjpeg` (JPEG image routines) and `libz` (compression routines).

Listing 2.9 shows a very small program that uses `libtiff` to open a TIFF image file.

Listing 2.9 (*tifftest.c*) Using *libtiff*

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

int main (int argc, char** argv)
{
  TIFF* tiff;
  tiff = TIFFOpen (argv[1], "r");
  TIFFClose (tiff);
  return 0;
}
```

4. You might see a reference to `LD_RUN_PATH` in some online documentation. Don't believe what you read; this variable does not actually do anything under GNU/Linux.

Save this source file as `tifftest.c`. To compile this program and link with `libtiff`, specify `-ltiff` on your link line:

```
% gcc -o tifftest tifftest.c -ltiff
```

By default, this will pick up the shared-library version of `libtiff`, found at `/usr/lib/libtiff.so`. Because `libtiff` uses `libjpeg` and `libz`, the shared-library versions of these two are also drawn in (a shared library can point to other shared libraries that it depends on). To verify this, use the `ldd` command:

```
% ldd tifftest
        libtiff.so.3 => /usr/lib/libtiff.so.3 (0x4001d000)
        libc.so.6 => /lib/libc.so.6 (0x40060000)
        libjpeg.so.62 => /usr/lib/libjpeg.so.62 (0x40155000)
        libz.so.1 => /usr/lib/libz.so.1 (0x40174000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Static libraries, on the other hand, cannot point to other libraries. If decide to link with the static version of `libtiff` by specifying `-static` on your command line, you will encounter unresolved symbols:

```
% gcc -static -o tifftest tifftest.c -ltiff
/usr/bin/../lib/libtiff.a(tif_jpeg.o): In function 'TIFFjpeg_error_exit':
tif_jpeg.o(.text+0x2a): undefined reference to 'jpeg_abort'
/usr/bin/../lib/libtiff.a(tif_jpeg.o): In function 'TIFFjpeg_create_compress':
tif_jpeg.o(.text+0x8d): undefined reference to 'jpeg_std_error'
tif_jpeg.o(.text+0xcf): undefined reference to 'jpeg_CreateCompress'
...
```

To link this program statically, you must specify the other two libraries yourself:

```
% gcc -static -o tifftest tifftest.c -ltiff -ljpeg -lz
```

Occasionally, two libraries will be mutually dependent. In other words, the first archive will reference symbols defined in the second archive, and vice versa. This situation generally arises out of poor design, but it does occasionally arise. In this case, you can provide a single library multiple times on the command line. The linker will research the library each time it occurs. For example, this line will cause `libfoo.a` to be searched multiple times:

```
% gcc -o app app.o -lfoo -lbar -lfoo
```

So, even if `libfoo.a` references symbols in `libbar.a`, and vice versa, the program will link successfully.

2.3.5 Pros and Cons

Now that you know all about static archives and shared libraries, you're probably wondering which to use. There are a few major considerations to keep in mind.

One major advantage of a shared library is that it saves space on the system where the program is installed. If you are installing 10 programs, and they all make use of the same shared library, then you save a lot of space by using a shared library. If you used a static archive instead, the archive is included in all 10 programs. So, using shared libraries saves disk space. It also reduces download times if your program is being downloaded from the Web.

A related advantage to shared libraries is that users can upgrade the libraries without upgrading all the programs that depend on them. For example, suppose that you produce a shared library that manages HTTP connections. Many programs might depend on this library. If you find a bug in this library, you can upgrade the library. Instantly, all the programs that depend on the library will be fixed; you don't have to relink all the programs the way you do with a static archive.

Those advantages might make you think that you should always use shared libraries. However, substantial reasons exist to use static archives instead. The fact that an upgrade to a shared library affects all programs that depend on it can be a disadvantage. For example, if you're developing mission-critical software, you might rather link to a static archive so that an upgrade to shared libraries on the system won't affect your program. (Otherwise, users might upgrade the shared library, thereby breaking your program, and then call your customer support line, blaming you!)

If you're not going to be able to install your libraries in `/lib` or `/usr/lib`, you should definitely think twice about using a shared library. (You won't be able to install your libraries in those directories if you expect users to install your software without administrator privileges.) In particular, the `-Wl,-rpath` trick won't work if you don't know where the libraries are going to end up. And asking your users to set `LD_LIBRARY_PATH` means an extra step for them. Because each user has to do this individually, this is a substantial additional burden.

You'll have to weigh these advantages and disadvantages for every program you distribute.

2.3.6 Dynamic Loading and Unloading

Sometimes you might want to load some code at run time without explicitly linking in that code. For example, consider an application that supports "plug-in" modules, such as a Web browser. The browser allows third-party developers to create plug-ins to provide additional functionality. The third-party developers create shared libraries and place them in a known location. The Web browser then automatically loads the code in these libraries.

This functionality is available under Linux by using the `dlopen` function. You could open a shared library named `libtest.so` by calling `dlopen` like this:

```
dlopen ("libtest.so", RTLD_LAZY)
```

(The second parameter is a flag that indicates how to bind symbols in the shared library. You can consult the online man pages for dlopen if you want more information, but RTLD_LAZY is usually the setting that you want.) To use dynamic loading functions, include the <dlfcn.h> header file and link with the -ldl option to pick up the libdl library.

The return value from this function is a void * that is used as a handle for the shared library. You can pass this value to the dlsym function to obtain the address of a function that has been loaded with the shared library. For example, if libtest.so defines a function named my_function, you could call it like this:

```
void* handle = dlopen ("libtest.so", RTLD_LAZY);
void (*test)() = dlsym (handle, "my_function");
(*test)();
dlclose (handle);
```

The dlsym system call can also be used to obtain a pointer to a static variable in the shared library.

Both dlopen and dlsym return NULL if they do not succeed. In that event, you can call dlerror (with no parameters) to obtain a human-readable error message describing the problem.

The dlclose function unloads the shared library. Technically, dlopen actually loads the library only if it is not already loaded. If the library has already been loaded, dlopen simply increments the library reference count. Similarly, dlclose decrements the reference count and then unloads the library only if the reference count has reached zero.

If you're writing the code in your shared library in C++, you will probably want to declare those functions and variables that you plan to access elsewhere with the extern "C" linkage specifier. For instance, if the C++ function my_function is in a shared library and you want to access it with dlsym, you should declare it like this:

```
extern "C" void foo ();
```

This prevents the C++ compiler from mangling the function name, which would change the function's name from foo to a different, funny-looking name that encodes extra information about the function. A C compiler will not mangle names; it will use whichever name you give to your function or variable.