11

A Sample GNU/Linux Application

This chapter is where it all comes together. We'll describe and implement a complete GNU/Linux program that incorporates many of the techniques described in this book. The program provides information about the system it's running on via a Web interface.

The program is a complete demonstration of some of the methods we've described for GNU/Linux programming and illustrated in shorter programs. This program is written more like "real-world" code, unlike most of the code listings that we presented in previous chapters. It can serve as a jumping-off point for your own GNU/Linux programs.

11.1 Overview

The example program is part of a system for monitoring a running GNU/Linux system. It includes these features:

- The program incorporates a minimal Web server. Local or remote clients access system information by requesting Web pages from the server via HTTP.
- The program does not serve static HTML pages. Instead, the pages are generated on the fly by modules, each of which provides a page summarizing one aspect of the system's state.

- Modules are not linked statically into the server executable. Instead, they are loaded dynamically from shared libraries. Modules can be added, removed, or replaced while the server is running.
- The server services each connection in a child process. This enables the server to remain responsive even when individual requests take a while to complete, and it shields the server from failures in modules.
- The server does not require superuser privilege to run (as long as it is not run on a privileged port). However, this limits the system information that it can collect.

We provide four sample modules that demonstrate how modules might be written. They further illustrate some of the techniques for gathering system information presented previously in this book. The `time` module demonstrates using the `gettimeofday` system call. The `issue` module demonstrates low-level I/O and the `sendfile` system call. The `diskfree` module demonstrates the use of `fork`, `exec`, and `dup2` by running a command in a child process. The `processes` module demonstrates the use of the `/proc` file system and various system calls.

11.1.1 Caveats

This program has many of the features you'd expect in an application program, such as command-line parsing and error checking. At the same time, we've made some simplifications to improve readability and to focus on the GNU/Linux-specific topics discussed in this book. Bear in mind these caveats as you examine the code.

- We don't attempt to provide a full implementation of HTTP. Instead, we implement just enough for the server to interact with Web clients. A real-world program either would provide a more complete HTTP implementation or would interface with one of the various excellent Web server implementations[1] available instead of providing HTTP services directly.
- Similarly, we don't aim for full compliance with HTML specifications (see `http://www.w3.org/MarkUp/`). We generate simple HTML output that can be handled by popular Web browsers.
- The server is not tuned for high performance or minimum resource usage. In particular, we intentionally omit some of the network configuration code that you would expect in a Web server. This topic is outside the scope of this book. See one of the many excellent references on network application development, such as *UNIX Network Programming, Volume 1: Networking APIs—Sockets and XTI,* by W. Richard Stevens (Prentice Hall, 1997), for more information.

1. The most popular open source Web server for GNU/Linux is the Apache server, available from `http://www.apache.org`.

- We make no attempt to regulate the resources (number of processes, memory use, and so on) consumed by the server or its modules. Many multiprocess Web server implementations service connections using a fixed pool of processes rather than creating a new child process for each connection.
- The server loads the shared library for a server module each time it is requested and then immediately unloads it when the request has been completed. A more efficient implementation would probably cache loaded modules.

HTTP

The *Hypertext Transport Protocol* (*HTTP*) is used for communication between Web clients and servers. The client connects to the server by establishing a connection to a well-known port (usually port 80 for Internet Web servers, but any port may be used). HTTP requests and headers are composed of plain text.

Once connected, the client sends a request to the server. A typical request is `GET /page HTTP/1.0`. The GET method indicates that the client is requesting that the server send it a Web page. The second element is the path to that page on the server. The third element is the protocol and version. Subsequent lines contain header fields, formatted similarly to email headers, which contain extra information about the client. The header ends with a blank line.

The server sends back a response indicating the result of processing the request. A typical response is `HTTP/1.0 200 OK`. The first element is the protocol version. The next two elements indicate the result; in this case, result `200` indicates that the request was processed successfully. Subsequent lines contain header fields, formatted similarly to email headers. The header ends with a blank line. The server may then send arbitrary data to satisfy the request.

Typically, the server responds to a page request by sending back HTML source for the Web page. In this case, the response headers will include `Content-type: text/html`, indicating that the result is HTML source. The HTML source follows immediately after the header.

See the HTTP specification at `http://www.w3.org/Protocols/` for more information.

11.2 Implementation

All but the very smallest programs written in C require careful organization to preserve the modularity and maintainability of the source code. This program is divided into four main source files.

Each source file exports functions or variables that may be accessed by the other parts of the program. For simplicity, all exported functions and variables are declared in a single header file, `server.h` (see Listing 11.1), which is included by the other files. Functions that are intended for use within a single compilation unit only are declared `static` and are not declared in `server.h`.

Listing 11.1 ***(server.h)* Function and Variable Declarations**

```
#ifndef SERVER_H
#define SERVER_H

#include <netinet/in.h>
#include <sys/types.h>

/*** Symbols defined in common.c  ***********************************/

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

/* If nonzero, print verbose messages.  */
extern int verbose;

/* Like malloc, except aborts the program if allocation fails.  */
extern void* xmalloc (size_t size);

/* Like realloc, except aborts the program if allocation fails.  */
extern void* xrealloc (void* ptr, size_t size);

/* Like strdup, except aborts the program if allocation fails.  */
extern char* xstrdup (const char* s);

/* Print an error message for a failed call OPERATION, using the value
   of errno, and end the program.  */
extern void system_error (const char* operation);

/* Print an error message for failure involving CAUSE, including a
   descriptive MESSAGE, and end the program.  */
extern void error (const char* cause, const char* message);

/* Return the directory containing the running program's executable.
   The return value is a memory buffer that the caller must deallocate
   using free.  This function calls abort on failure.  */
extern char* get_self_executable_directory ();


/*** Symbols defined in module.c  ***********************************/

/* An instance of a loaded server module.  */
struct server_module {
  /* The shared library handle corresponding to the loaded module.  */
  void* handle;
  /* A name describing the module.  */
  const char* name;
  /* The function that generates the HTML results for this module.  */
void (* generate_function) (int);
};
```

```
/* The directory from which modules are loaded.  */
extern char* module_dir;

/* Attempt to load a server module with the name MODULE_PATH.  If a
   server module exists with this path, loads the module and returns a
   server_module structure representing it.  Otherwise, returns NULL.  */
extern struct server_module* module_open (const char* module_path);

/* Close a server module and deallocate the MODULE object.  */
extern void module_close (struct server_module* module);


/*** Symbols defined in server.c.  ***********************************/

/* Run the server on LOCAL_ADDRESS and PORT.  */
extern void server_run (struct in_addr local_address, uint16_t port);


#endif  /* SERVER_H */
```

11.2.1 Common Functions

`common.c` (see Listing 11.2) contains functions of general utility that are used throughout the program.

Listing 11.2 **(*common.c*) General Utility Functions**

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

#include "server.h"

const char* program_name;

int verbose;

void* xmalloc (size_t size)
{
  void* ptr = malloc (size);
  /* Abort if the allocation failed.  */
  if (ptr == NULL)
    abort ();
  else
    return ptr;
}
```

continues

Listing 11.2 **Continued**

```
void* xrealloc (void* ptr, size_t size)
{
  ptr = realloc (ptr, size);
  /* Abort if the allocation failed.  */
  if (ptr == NULL)
    abort ();
  else
    return ptr;
}

char* xstrdup (const char* s)
{
  char* copy = strdup (s);
  /* Abort if the allocation failed.  */
  if (copy == NULL)
    abort ();
  else
    return copy;
}

void system_error (const char* operation)
{
  /* Generate an error message for errno.  */
  error (operation, strerror (errno));
}

void error (const char* cause, const char* message)
{
  /* Print an error message to stderr.  */
  fprintf (stderr, "%s: error: (%s) %s\n", program_name, cause, message);
  /* End the program.  */
  exit (1);
}

char* get_self_executable_directory ()
{
  int rval;
  char link_target[1024];
  char* last_slash;
  size_t result_length;
  char* result;

  /* Read the target of the symbolic link /proc/self/exe.  */
  rval = readlink ("/proc/self/exe", link_target, sizeof (link_target));
  if (rval == -1)
    /* The call to readlink failed, so bail.  */
    abort ();
  else
```

```
    /* NUL-terminate the target.  */
    link_target[rval] = '\0';
  /* We want to trim the name of the executable file, to obtain the
     directory that contains it.  Find the rightmost slash.  */
  last_slash = strrchr (link_target, '/');
  if (last_slash == NULL || last_slash == link_target)
    /* Something strange is going on.  */
    abort ();
  /* Allocate a buffer to hold the resulting path.  */
  result_length = last_slash - link_target;
  result = (char*) xmalloc (result_length + 1);
  /* Copy the result.  */
  strncpy (result, link_target, result_length);
  result[result_length] = '\0';
  return result;
}
```

You could use these functions in other programs as well; the contents of this file might be included in a common code library that is shared among many projects:

- `xmalloc`, `xrealloc`, and `xstrdup` are error-checking versions of the C library functions `malloc`, `realloc`, and `strdup`, respectively. Unlike the standard versions, which return a null pointer if the allocation fails, these functions immediately abort the program when insufficient memory is available.

  Early detection of memory allocation failure is a good idea. Otherwise, failed allocations introduce null pointers at unexpected places into the program. Because allocation failures are not easy to reproduce, debugging such problems can be difficult. Allocation failures are usually catastrophic, so aborting the program is often an acceptable course of action.

- The `error` function is for reporting a fatal program error. It prints a message to `stderr` and ends the program. For errors caused by failed system calls or library calls, `system_error` generates part of the error message from the value of `errno` (see Section 2.2.3, "Error Codes from System Calls," in Chapter 2, "Writing Good GNU/Linux Software").

- `get_self_executable_directory` determines the directory containing the executable file being run in the current process. The directory path can be used to locate other components of the program, which are installed in the same place at runtime. This function works by examining the symbolic link `/proc/self/exe` in the `/proc` file system (see Section 7.2.1, "`/proc/self`," in Chapter 7, "The `/proc` File System").

In addition, `common.c` defines two useful global variables:

- The value of `program_name` is the name of the program being run, as specified in its argument list (see Section 2.1.1, "The Argument List," in Chapter 2). When the program is invoked from the shell, this is the path and name of the program as the user entered it.

- The variable `verbose` is nonzero if the program is running in verbose mode. In this case, various parts of the program print progress messages to `stdout`.

11.2.2 Loading Server Modules

`module.c` (see Listing 11.3) provides the implementation of dynamically loadable server modules. A loaded server module is represented by an instance of `struct server_module`, which is defined in `server.h`.

Listing 11.3 **(*module.c*) Server Module Loading and Unloading**

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

#include "server.h"

char* module_dir;

struct server_module* module_open (const char* module_name)
{
  char* module_path;
  void* handle;
  void (* module_generate) (int);
  struct server_module* module;

  /* Construct the full path of the module shared library we'll try to
     load.  */
  module_path =
    (char*) xmalloc (strlen (module_dir) + strlen (module_name) + 2);
  sprintf (module_path, "%s/%s", module_dir, module_name);

  /* Attempt to open MODULE_PATH as a shared library.  */
  handle = dlopen (module_path, RTLD_NOW);
  free (module_path);
  if (handle == NULL) {
    /* Failed; either this path doesn't exist or it isn't a shared
       library.  */
    return NULL;
  }

  /* Resolve the module_generate symbol from the shared library.  */
  module_generate = (void (*) (int)) dlsym (handle, "module_generate");
  /* Make sure the symbol was found.  */
  if (module_generate == NULL) {
```

```
    /* The symbol is missing.  While this is a shared library, it
       probably isn't a server module.  Close up and indicate failure.  */
    dlclose (handle);
    return NULL;
  }

  /* Allocate and initialize a server_module object.  */
  module = (struct server_module*) xmalloc (sizeof (struct server_module));
  module->handle = handle;
  module->name = xstrdup (module_name);
  module->generate_function = module_generate;
  /* Return it, indicating success.  */
  return module;
}

void module_close (struct server_module* module)
{
  /* Close the shared library.  */
  dlclose (module->handle);
  /* Deallocate the module name.  */
  free ((char*) module->name);
  /* Deallocate the module object.  */
  free (module);
}
```

Each module is a shared library file (see Section 2.3.2, "Shared Libraries," in Chapter 2) and must define and export a function named `module_generate`. This function generates an HTML Web page and writes it to the client socket file descriptor passed as its argument.

`module.c` contains two functions:

- `module_open` attempts to load a server module with a given name. The name normally ends with the `.so` extension because server modules are implemented as shared libraries. This function opens the shared library with `dlopen` and resolves a symbol named `module_generate` from the library with `dlsym` (see Section 2.3.6, "Dynamic Loading and Unloading," in Chapter 2). If the library can't be opened, or if `module_generate` isn't a name exported by the library, the call fails and `module_open` returns a null pointer. Otherwise, it allocates and returns a module object.
- `module_close` closes the shared library corresponding to the server module and deallocates the `struct server_module` object.

`module.c` also defines a global variable `module_dir`. This is the path of the directory in which `module_open` attempts to find shared libraries corresponding to server modules.

11.2.3 The Server

`server.c` (see Listing 11.4) is the implementation of the minimal HTTP server.

Listing 11.4 **(*server.c*) Server Implementation**

```
#include <arpa/inet.h>
#include <assert.h>
#include <errno.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <unistd.h>

#include "server.h"

/* HTTP response and header for a successful request.  */

static char* ok_response =
  "HTTP/1.0 200 OK\n"
  "Content-type: text/html\n"
  "\n";

/* HTTP response, header, and body, indicating that we didn't
   understand the request.  */

static char* bad_request_response =
  "HTTP/1.0 400 Bad Request\n"
  "Content-type: text/html\n"
  "\n"
  "<html>\n"
  " <body>\n"
  "  <h1>Bad Request</h1>\n"
  "  <p>This server did not understand your request.</p>\n"
  " </body>\n"
  "</html>\n";

/* HTTP response, header, and body template, indicating that the
   requested document was not found.  */

static char* not_found_response_template =
  "HTTP/1.0 404 Not Found\n"
  "Content-type: text/html\n"
  "\n"
  "<html>\n"
  " <body>\n"
  "  <h1>Not Found</h1>\n"
```

```
  "  <p>The requested URL %s was not found on this server.</p>\n"
  " </body>\n"
  "</html>\n";

/* HTTP response, header, and body template, indicating that the
   method was not understood.  */

static char* bad_method_response_template =
  "HTTP/1.0 501 Method Not Implemented\n"
  "Content-type: text/html\n"
  "\n"
  "<html>\n"
  " <body>\n"
  "  <h1>Method Not Implemented</h1>\n"
  "  <p>The method %s is not implemented by this server.</p>\n"
  " </body>\n"
  "</html>\n";

/* Handler for SIGCHLD, to clean up child processes that have
   terminated.  */

static void clean_up_child_process (int signal_number)
{
  int status;
  wait (&status);
}

/* Process an HTTP "GET" request for PAGE, and send the results to the
   file descriptor CONNECTION_FD.  */

static void handle_get (int connection_fd, const char* page)
{
  struct server_module* module = NULL;

  /* Make sure the requested page begins with a slash and does not
     contain any additional slashes -- we don't support any
     subdirectories.  */
  if (*page == '/' && strchr (page + 1, '/') == NULL) {
    char module_file_name[64];

    /* The page name looks OK.  Construct the module name by appending
       ".so" to the page name.  */
    snprintf (module_file_name, sizeof (module_file_name),
              "%s.so", page + 1);
    /* Try to open the module.  */
    module = module_open (module_file_name);
  }

  if (module == NULL) {
    /* Either the requested page was malformed, or we couldn't open a
       module with the indicated name.  Either way, return the HTTP
       response 404, Not Found.  */
```

continues

Listing 11.4 **Continued**

```
    char response[1024];

    /* Generate the response message.  */
    snprintf (response, sizeof (response), not_found_response_template, page);
    /* Send it to the client.  */
    write (connection_fd, response, strlen (response));
  }
  else {
    /* The requested module was loaded successfully.  */

    /* Send the HTTP response indicating success, and the HTTP header
       for an HTML page.  */
    write (connection_fd, ok_response, strlen (ok_response));
    /* Invoke the module, which will generate HTML output and send it
       to the client file descriptor.  */
    (*module->generate_function) (connection_fd);
    /* We're done with the module.  */
    module_close (module);
  }
}

/* Handle a client connection on the file descriptor CONNECTION_FD.  */

static void handle_connection (int connection_fd)
{
  char buffer[256];
  ssize_t bytes_read;

  /* Read some data from the client.  */
  bytes_read = read (connection_fd, buffer, sizeof (buffer) - 1);
  if (bytes_read > 0) {
    char method[sizeof (buffer)];
    char url[sizeof (buffer)];
    char protocol[sizeof (buffer)];

    /* Some data was read successfully.  NUL-terminate the buffer so
       we can use string operations on it.  */
    buffer[bytes_read] = '\0';
    /* The first line the client sends is the HTTP request, which is
       composed of a method, the requested page, and the protocol
       version.  */
    sscanf (buffer, "%s %s %s", method, url, protocol);
    /* The client may send various header information following the
       request.  For this HTTP implementation, we don't care about it.
       However, we need to read any data the client tries to send.  Keep
       on reading data until we get to the end of the header, which is
       delimited by a blank line.  HTTP specifies CR/LF as the line
       delimiter.  */
    while (strstr (buffer, "\r\n\r\n") == NULL)
```

```
      bytes_read = read (connection_fd, buffer, sizeof (buffer));
    /* Make sure the last read didn't fail.  If it did, there's a
       problem with the connection, so give up.  */
    if (bytes_read == -1) {
      close (connection_fd);
      return;
    }
    /* Check the protocol field.  We understand HTTP versions 1.0 and
       1.1.  */
    if (strcmp (protocol, "HTTP/1.0") && strcmp (protocol, "HTTP/1.1")) {
      /* We don't understand this protocol.  Report a bad response.  */
      write (connection_fd, bad_request_response,
             sizeof (bad_request_response));
    }
    else if (strcmp (method, "GET")) {
      /* This server only implements the GET method.  The client
         specified some other method, so report the failure.  */
      char response[1024];

      snprintf (response, sizeof (response),
                bad_method_response_template, method);
      write (connection_fd, response, strlen (response));
    }
    else
      /* A valid request.  Process it.  */
      handle_get (connection_fd, url);
  }
  else if (bytes_read == 0)
    /* The client closed the connection before sending any data.
       Nothing to do.  */
    ;
  else
    /* The call to read failed.  */
    system_error ("read");
}


void server_run (struct in_addr local_address, uint16_t port)
{
  struct sockaddr_in socket_address;
  int rval;
  struct sigaction sigchld_action;
  int server_socket;

  /* Install a handler for SIGCHLD that cleans up child processes that
     have terminated.  */
  memset (&sigchld_action, 0, sizeof (sigchld_action));
  sigchld_action.sa_handler = &clean_up_child_process;
  sigaction (SIGCHLD, &sigchld_action, NULL);
```

continues

Listing 11.4 **Continued**

```
/* Create a TCP socket.  */
server_socket = socket (PF_INET, SOCK_STREAM, 0);
if (server_socket == -1)
  system_error ("socket");
/* Construct a socket address structure for the local address on
   which we want to listen for connections.  */
memset (&socket_address, 0, sizeof (socket_address));
socket_address.sin_family = AF_INET;
socket_address.sin_port = port;
socket_address.sin_addr = local_address;
/* Bind the socket to that address.  */
rval = bind (server_socket, &socket_address, sizeof (socket_address));
if (rval != 0)
  system_error ("bind");
/*  Instruct the socket to accept connections.  */
rval = listen (server_socket, 10);
if (rval != 0)
  system_error ("listen");

if (verbose) {
  /* In verbose mode, display the local address and port number
     we're listening on.  */
  socklen_t address_length;

  /* Find the socket's local address.  */
  address_length = sizeof (socket_address);
  rval = getsockname (server_socket, &socket_address, &address_length);
  assert (rval == 0);
  /* Print a message.  The port number needs to be converted from
     network byte order (big endian) to host byte order.  */
  printf ("server listening on %s:%d\n",
          inet_ntoa (socket_address.sin_addr),
          (int) ntohs (socket_address.sin_port));
}

/* Loop forever, handling connections.  */
while (1) {
  struct sockaddr_in remote_address;
  socklen_t address_length;
  int connection;
  pid_t child_pid;

  /* Accept a connection.  This call blocks until a connection is
     ready.  */
  address_length = sizeof (remote_address);
  connection = accept (server_socket, &remote_address, &address_length);
  if (connection == -1) {
    /* The call to accept failed.  */
    if (errno == EINTR)
```

```
      /* The call was interrupted by a signal.  Try again.  */
      continue;
    else
      /* Something else went wrong.  */
      system_error ("accept");
  }

  /* We have a connection.  Print a message if we're running in
     verbose mode.  */
  if (verbose) {
    socklen_t address_length;

    /* Get the remote address of the connection.  */
    address_length = sizeof (socket_address);
    rval = getpeername (connection, &socket_address, &address_length);
    assert (rval == 0);
    /* Print a message.  */
    printf ("connection accepted from %s\n",
            inet_ntoa (socket_address.sin_addr));
  }

  /* Fork a child process to handle the connection.  */
  child_pid = fork ();
  if (child_pid == 0) {
    /* This is the child process.  It shouldn't use stdin or stdout,
       so close them.  */
    close (STDIN_FILENO);
    close (STDOUT_FILENO);
    /* Also this child process shouldn't do anything with the
       listening socket.  */
    close (server_socket);
    /* Handle a request from the connection.  We have our own copy
       of the connected socket descriptor.  */
    handle_connection (connection);
    /* All done; close the connection socket, and end the child
       process.  */
    close (connection);
    exit (0);
  }
  else if (child_pid > 0) {
    /* This is the parent process.  The child process handles the
       connection, so we don't need our copy of the connected socket
       descriptor.  Close it.  Then continue with the loop and
       accept another connection.  */
    close (connection);
  }
  else
    /* Call to fork failed.  */
    system_error ("fork");
 }
}
```

These are the functions in `server.c`:

- `server_run` is the main entry point for running the server. This function starts the server and begins accepting connections, and does not return unless a serious error occurs. The server uses a TCP stream server socket (see Section 5.5.3, "Servers," in Chapter 5, "Interprocess Communication").

  The first argument to `server_run` specifies the local address at which connections are accepted. A GNU/Linux computer may have multiple network addresses, and each address may be bound to a different network interface.[2] To restrict the server to accept connections from a particular interface, specify the corresponding network address. Specify the local address `INADDR_ANY` to accept connections for any local address.

  The second argument to `server_run` is the port number on which to accept connections. If the port number is already in use, or if it corresponds to a privileged port and the server is not being run with superuser privilege, the server fails. The special value 0 instructs Linux to select an unused port automatically. See the `inet` man page for more information about Internet-domain addresses and port numbers.

  The server handles each client connection in a child process created with `fork` (see Section 3.2.2, "Using `fork` and `exec`," in Chapter 3, "Processes"). The main (parent) process continues accepting new connections while existing ones are being serviced. The child process invokes `handle_connection` and then closes the connection socket and exits.

- `handle_connection` processes a single client connection, using the socket file descriptor passed as its argument. This function reads data from the socket and attempts to interpret this as an HTTP page request.

  The server processes only HTTP version 1.0 and version 1.1 requests. When faced with a different protocol or version, it responds by sending the HTTP result code 400 and the message `bad_request_response`. The server understands only the HTTP GET method. If the client requests any other method, the server responds by sending the HTTP result code 501 and the message `bad_method_response_template`.

- If the client sends a well-formed GET request, `handle_connection` calls `handle_get` to service it. This function attempts to load a server module with a name generated from the requested page. For example, if the client requests the page named information, it attempts to load a server module named `information.so`. If the module can't be loaded, `handle_get` sends the client the HTTP result code 404 and the message `not_found_response_template`.

2. Your computer might be configured to include such interfaces as `eth0`, an Ethernet card; `lo`, the local (loopback) network; or `ppp0`, a dial-up network connection.

If the client sends a page request that corresponds to a server module, `handle_get` sends a result code 200 header to the client, which indicates that the request was processed successfully and invokes the module's `module_generate` function. This function generates the HTML source for a Web page and sends it to the Web client.

- `server_run` installs `clean_up_child_process` as the signal handler for `SIGCHLD`. This function simply cleans up terminated child processes (see Section 3.4.4, "Cleaning Up Children Asynchronously," in Chapter 3).

11.2.4 The Main Program

`main.c` (see Listing 11.5) provides the `main` function for the server program. Its responsibility is to parse command-line options, detect and report command-line errors, and configure and run the server.

Listing 11.5 **(*main.c*) Main Server Program and Command-Line Parsing**

```
#include <assert.h>
#include <getopt.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>

#include "server.h"

/* Description of long options for getopt_long.  */

static const struct option long_options[] = {
  { "address",          1, NULL, 'a' },
  { "help",             0, NULL, 'h' },
  { "module-dir",       1, NULL, 'm' },
  { "port",             1, NULL, 'p' },
  { "verbose",          0, NULL, 'v' },
};

/* Description of short options for getopt_long.  */

static const char* const short_options = "a:hm:p:v";

/* Usage summary text.  */

static const char* const usage_template =
  "Usage: %s [ options ]\n"
  "  -a, --address ADDR        Bind to local address (by default, bind\n"
  "                              to all local addresses).\n"
```

continues

Listing 11.5 **Continued**

```
    "  -h, --help                Print this information.\n"
    "  -m, --module-dir DIR      Load modules from specified directory\n"
    "                              (by default, use executable directory).\n"
    "  -p, --port PORT           Bind to specified port.\n"
    "  -v, --verbose             Print verbose messages.\n";

/* Print usage information and exit.  If IS_ERROR is nonzero, write to
   stderr and use an error exit code.  Otherwise, write to stdout and
   use a non-error termination code.  Does not return.  */

static void print_usage (int is_error)
{
  fprintf (is_error ? stderr : stdout, usage_template, program_name);
  exit (is_error ? 1 : 0);
}

int main (int argc, char* const argv[])
{
  struct in_addr local_address;
  uint16_t port;
  int next_option;

  /* Store the program name, which we'll use in error messages.  */
  program_name = argv[0];

  /* Set defaults for options.  Bind the server to all local addresses,
     and assign an unused port automatically.  */
  local_address.s_addr = INADDR_ANY;
  port = 0;
  /* Don't print verbose messages.  */
  verbose = 0;
  /* Load modules from the directory containing this executable.  */
  module_dir = get_self_executable_directory ();
  assert (module_dir != NULL);

  /* Parse options.  */
  do {
    next_option =
      getopt_long (argc, argv, short_options, long_options, NULL);
    switch (next_option) {
    case 'a':
      /* User specified -a or --address.  */
      {
        struct hostent* local_host_name;

        /* Look up the hostname the user specified.  */
        local_host_name = gethostbyname (optarg);
        if (local_host_name == NULL || local_host_name->h_length == 0)
```

```
      /* Could not resolve the name.  */
      error (optarg, "invalid host name");
    else
      /* Hostname is OK, so use it.  */
      local_address.s_addr =
        *((int*) (local_host_name->h_addr_list[0]));
  }
  break;

case 'h':
  /* User specified -h or --help.  */
  print_usage (0);

case 'm':
  /* User specified -m or --module-dir.  */
  {
    struct stat dir_info;

    /* Check that it exists.  */
    if (access (optarg, F_OK) != 0)
      error (optarg, "module directory does not exist");
    /* Check that it is accessible.  */
    if (access (optarg, R_OK | X_OK) != 0)
      error (optarg, "module directory is not accessible");
    /* Make sure that it is a directory.  */
    if (stat (optarg, &dir_info) != 0 || !S_ISDIR (dir_info.st_mode))
      error (optarg, "not a directory");
    /* It looks OK, so use it.  */
    module_dir = strdup (optarg);
  }
  break;

case 'p':
  /* User specified -p or --port.  */
  {
    long value;
    char* end;

    value = strtol (optarg, &end, 10);
    if (*end != '\0')
      /* The user specified nondigits in the port number.  */
      print_usage (1);
    /* The port number needs to be converted to network (big endian)
       byte order.  */
    port = (uint16_t) htons (value);
  }
  break;

case 'v':
  /* User specified -v or --verbose.  */
  verbose = 1;
  break;
```

continues

Listing 11.5 **Continued**

```
      case '?':
        /* User specified an unrecognized option.  */
        print_usage (1);

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

      default:
        abort ();
      }
    } while (next_option != -1);

    /* This program takes no additional arguments.  Issue an error if the
       user specified any.  */
    if (optind != argc)
      print_usage (1);

    /* Print the module directory, if we're running verbose.  */
    if (verbose)
      printf ("modules will be loaded from %s\n", module_dir);

    /* Run the server.  */
    server_run (local_address, port);

    return 0;
  }
```

`main.c` contains these functions:

- `main` invokes `getopt_long` (see Section 2.1.3, "Using `getopt_long`," in Chapter 2) to parse command-line options. It provides both long and short option forms, the former in the `long_options` array and the latter in the `short_options` string.

  The default value for the server port is 0 and for a local address is `INADDR_ANY`. These can be overridden by the `--port` (`-p`) and `--address` (`-a`) options, respectively. If the user specifies an address, `main` calls the library function `gethostbyname` to convert it to a numerical Internet address.[3]

  The default value for the directory from which to load server modules is the directory containing the server executable, as determined by `get_self_executable_directory`. The user may override this with the `--module-dir` (`-m`) option; `main` makes sure that the specified directory is accessible.

  By default, verbose messages are not printed. The user may enable them by specifying the `--verbose` (`-v`) option.

3. `gethostbyname` performs name resolution using DNS, if necessary.

- If the user specifies the `--help` (`-h`) option or specifies invalid options, `main` invokes `print_usage`, which prints a usage summary and exits.

11.3 Modules

We provide four modules to demonstrate the kind of functionality you could implement using this server implementation. Implementing your own server module is as simple as defining a `module_generate` function to return the appropriate HTML text.

11.3.1 Show Wall-Clock Time

The `time.so` module (see Listing 11.6) generates a simple page containing the server's local wall-clock time. This module's `module_generate` calls `gettimeofday` to obtain the current time (see Section 8.7, "`gettimeofday`: Wall-Clock Time," in Chapter 8, "Linux System Calls") and uses `localtime` and `strftime` to generate a text representation of it. This representation is embedded in the HTML template `page_template`.

Listing 11.6 **(*time.c*) Server Module to Show Wall-Clock Time**

```
#include <assert.h>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>

#include "server.h"

/* A template for the HTML page this module generates.  */

static char* page_template =
  "<html>\n"
  " <head>\n"
  "  <meta http-equiv=\"refresh\" content=\"5\">\n"
  " </head>\n"
  " <body>\n"
  "  <p>The current time is %s.</p>\n"
  " </body>\n"
  "</html>\n";

void module_generate (int fd)
{
  struct timeval tv;
  struct tm* ptm;
  char time_string[40];
  FILE* fp;

  /* Obtain the time of day, and convert it to a tm struct.  */
  gettimeofday (&tv, NULL);
  ptm = localtime (&tv.tv_sec);
```

continues

Listing 11.6 **Continued**

```
  /* Format the date and time, down to a single second.  */
  strftime (time_string, sizeof (time_string), "%H:%M:%S", ptm);

  /* Create a stream corresponding to the client socket file
     descriptor.  */
  fp = fdopen (fd, "w");
  assert (fp != NULL);
  /* Generate the HTML output.  */
  fprintf (fp, page_template, time_string);
  /* All done; flush the stream.  */
  fflush (fp);
}
```

This module uses standard C library I/O routines for convenience. The `fdopen` call generates a stream pointer (`FILE*`) corresponding to the client socket file descriptor (see Section B.4, "Relation to Standard C Library I/O Functions," in Appendix B, "Low-Level I/O"). The module writes to it using `fprintf` and flushes it using `fflush` to prevent the loss of buffered data when the socket is closed.

The HTML page returned by the `time.so` module includes a `<meta>` element in the page header that instructs clients to reload the page every 5 seconds. This way the client displays the current time.

11.3.2 Show the GNU/Linux Distribution

The `issue.so` module (see Listing 11.7) displays information about the GNU/Linux distribution running on the server. This information is traditionally stored in the file `/etc/issue`. This module sends the contents of this file, wrapped in a `<pre>` element of an HTML page.

Listing 11.7 **(*issue.c*) Server Module to Display GNU/Linux Distribution Information**

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

#include "server.h"

/* HTML source for the start of the page we generate.  */

static char* page_start =
  "<html>\n"
  " <body>\n"
```

```
  "  <pre>\n";

/* HTML source for the end of the page we generate.  */

static char* page_end =
  "  </pre>\n"
  " </body>\n"
  "</html>\n";

/* HTML source for the page indicating there was a problem opening
   /proc/issue.  */

static char* error_page =
  "<html>\n"
  " <body>\n"
  "  <p>Error: Could not open /proc/issue.</p>\n"
  " </body>\n"
  "</html>\n";

/* HTML source indicating an error.  */

static char* error_message = "Error reading /proc/issue.";

void module_generate (int fd)
{
  int input_fd;
  struct stat file_info;
  int rval;

  /* Open /etc/issue.  */
  input_fd = open ("/etc/issue", O_RDONLY);
  if (input_fd == -1)
    system_error ("open");
  /* Obtain file information about it.  */
  rval = fstat (input_fd, &file_info);

  if (rval == -1)
    /* Either we couldn't open the file or we couldn't read from it.  */
    write (fd, error_page, strlen (error_page));
  else {
    int rval;
    off_t offset = 0;

    /* Write the start of the page.  */
    write (fd, page_start, strlen (page_start));
    /* Copy from /proc/issue to the client socket.  */
    rval = sendfile (fd, input_fd, &offset, file_info.st_size);
    if (rval == -1)
      /* Something went wrong sending the contents of /proc/issue.
         Write an error message.  */
      write (fd, error_message, strlen (error_message));
```

continues

Listing 11.7 **Continued**

```
    /* End the page.  */
    write (fd, page_end, strlen (page_end));
  }

  close (input_fd);
}
```

The module first tries to open `/etc/issue`. If that file can't be opened, the module sends an error page to the client. Otherwise, the module sends the start of the HTML page, contained in `page_start`. Then it sends the contents of `/etc/issue` using `sendfile` (see Section 8.12, "`sendfile`: Fast Data Transfers," in Chapter 8). Finally, it sends the end of the HTML page, contained in `page_end`.

You can easily adapt this module to send the contents of another file. If the file contains a complete HTML page, simply omit the code that sends the contents of `page_start` and `page_end`. You could also adapt the main server implementation to serve static files, in the manner of a traditional Web server. Using `sendfile` provides an extra degree of efficiency.

11.3.3 Show Free Disk Space

The `diskfree.so` module (see Listing 11.8) generates a page displaying information about free disk space on the file systems mounted on the server computer. This generated information is simply the output of invoking the `df -h` command. Like `issue.so`, this module wraps the output in a `<pre>` element of an HTML page.

Listing 11.8 **(*diskfree.c*) Server Module to Display Information About Free Disk Space**

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

#include "server.h"

/* HTML source for the start of the page we generate.  */

static char* page_start =
  "<html>\n"
  " <body>\n"
  "  <pre>\n";
```

```
/* HTML source for the end of the page we generate.  */

static char* page_end =
  "  </pre>\n"
  " </body>\n"
  "</html>\n";

void module_generate (int fd)
{
  pid_t child_pid;
  int rval;

  /* Write the start of the page.  */
  write (fd, page_start, strlen (page_start));
  /* Fork a child process.  */
  child_pid = fork ();
  if (child_pid == 0) {
    /* This is the child process.  */
    /* Set up an argument list for the invocation of df.  */
    char* argv[] = { "/bin/df", "-h", NULL };

    /* Duplicate stdout and stderr to send data to the client socket.  */
    rval = dup2 (fd, STDOUT_FILENO);
    if (rval == -1)
      system_error ("dup2");
    rval = dup2 (fd, STDERR_FILENO);
    if (rval == -1)
      system_error ("dup2");
    /* Run df to show the free space on mounted file systems.  */
    execv (argv[0], argv);
    /* A call to execv does not return unless an error occurred.  */
    system_error ("execv");
  }
  else if (child_pid > 0) {
    /* This is the parent process.  Wait for the child process to
       finish.  */
    rval = waitpid (child_pid, NULL, 0);
    if (rval == -1)
      system_error ("waitpid");
  }
  else
    /* The call to fork failed.  */
    system_error ("fork");
  /* Write the end of the page.  */
  write (fd, page_end, strlen (page_end));
}
```

While `issue.so` sends the contents of a file using `sendfile`, this module must invoke a command and redirect its output to the client. To do this, the module follows these steps:

1. First, the module creates a child process using `fork` (see Section 3.2.2, "Using `fork` and `exec`," in Chapter 3).
2. The child process copies the client socket file descriptor to file descriptors `STDOUT_FILENO` and `STDERR_FILENO`, which correspond to standard output and standard error (see Section 2.1.4, "Standard I/O," in Chapter 2). The file descriptors are copied using the `dup2` call (see Section 5.4.3, "Redirecting the Standard Input, Output, and Error Streams," in Chapter 5). All further output from the process to either of these streams is sent to the client socket.
3. The child process invokes the `df` command with the `-h` option by calling `execv` (see Section 3.2.2, "Using `fork` and `exec`," in Chapter 3).
4. The parent process waits for the child process to exit by calling `waitpid` (see Section 3.4.2, "The `wait` System Calls," in Chapter 3).

You could easily adapt this module to invoke a different command and redirect its output to the client.

11.3.4 Summarize Running Processes

The `processes.so` module (see Listing 11.9) is a more extensive server module implementation. It generates a page containing a table that summarizes the processes currently running on the server system. Each process is represented by a row in the table that lists the PID, the executable program name, the owning user and group names, and the resident set size.

Listing 11.9 **(*processes.c*) Server Module to Summarize Processes**

```
#include <assert.h>
#include <dirent.h>
#include <fcntl.h>
#include <grp.h>
#include <pwd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>

#include "server.h"

/* Set *UID and *GID to the owning user ID and group ID, respectively,
   of process PID.  Return 0 on success, nonzero on failure.  */
```

```
static int get_uid_gid (pid_t pid, uid_t* uid, gid_t* gid)
{
  char dir_name[64];
  struct stat dir_info;
  int rval;

  /* Generate the name of the process's directory in /proc.  */
  snprintf (dir_name, sizeof (dir_name), "/proc/%d", (int) pid);
  /* Obtain information about the directory.  */
  rval = stat (dir_name, &dir_info);
  if (rval != 0)
    /* Couldn't find it; perhaps this process no longer exists.  */
    return 1;
  /* Make sure it's a directory; anything else is unexpected.  */
  assert (S_ISDIR (dir_info.st_mode));

  /* Extract the IDs we want.  */
  *uid = dir_info.st_uid;
  *gid = dir_info.st_gid;
  return 0;
}

/* Return the name of user UID.  The return value is a buffer that the
   caller must allocate with free.  UID must be a valid user ID.  */

static char* get_user_name (uid_t uid)
{
  struct passwd* entry;

  entry = getpwuid (uid);
  if (entry == NULL)
    system_error ("getpwuid");
  return xstrdup (entry->pw_name);
}

/* Return the name of group GID.  The return value is a buffer that the
   caller must allocate with free.  GID must be a valid group ID.  */

static char* get_group_name (gid_t gid)
{
  struct group* entry;

  entry = getgrgid (gid);
  if (entry == NULL)
    system_error ("getgrgid");
  return xstrdup (entry->gr_name);
}
```

continues

Listing 11.9 **Continued**

```
/* Return the name of the program running in process PID, or NULL on
   error.  The return value is a newly allocated buffer which the caller
   must deallocate with free.  */

static char* get_program_name (pid_t pid)
{
  char file_name[64];
  char status_info[256];
  int fd;
  int rval;
  char* open_paren;
  char* close_paren;
  char* result;

  /* Generate the name of the "stat" file in the process's /proc
     directory, and open it.  */
  snprintf (file_name, sizeof (file_name), "/proc/%d/stat", (int) pid);
  fd = open (file_name, O_RDONLY);
  if (fd == -1)
    /* Couldn't open the stat file for this process.  Perhaps the
       process no longer exists.  */
    return NULL;
  /* Read the contents.  */
  rval = read (fd, status_info, sizeof (status_info) - 1);
  close (fd);
  if (rval <= 0)
    /* Couldn't read, for some reason; bail.  */
    return NULL;
  /* NUL-terminate the file contents.  */
  status_info[rval] = '\0';

  /* The program name is the second element of the file contents and is
     surrounded by parentheses.  Find the positions of the parentheses
     in the file contents.  */
  open_paren = strchr (status_info, '(');
  close_paren = strchr (status_info, ')');
  if (open_paren == NULL
      || close_paren == NULL
      || close_paren < open_paren)
    /* Couldn't find them; bail.  */
    return NULL;
  /* Allocate memory for the result.  */
  result = (char*) xmalloc (close_paren - open_paren);
  /* Copy the program name into the result.  */
  strncpy (result, open_paren + 1, close_paren - open_paren - 1);
  /* strncpy doesn't NUL-terminate the result, so do it here.  */
  result[close_paren - open_paren - 1] = '\0';
  /* All done.  */
  return result;
}
```

```
/* Return the resident set size (RSS), in kilobytes, of process PID.
   Return -1 on failure.  */

static int get_rss (pid_t pid)
{
  char file_name[64];
  int fd;
  char mem_info[128];
  int rval;
  int rss;

  /* Generate the name of the process's "statm" entry in its /proc
     directory.  */
  snprintf (file_name, sizeof (file_name), "/proc/%d/statm", (int) pid);
  /* Open it.  */
  fd = open (file_name, O_RDONLY);
  if (fd == -1)
    /* Couldn't open it; perhaps this process no longer exists.  */
    return -1;
  /* Read the file's contents.  */
  rval = read (fd, mem_info, sizeof (mem_info) - 1);
  close (fd);
  if (rval <= 0)
    /* Couldn't read the contents; bail.  */
    return -1;
  /* NUL-terminate the contents.  */
  mem_info[rval] = '\0';
  /* Extract the RSS.  It's the second item.  */
  rval = sscanf (mem_info, "%*d %d", &rss);
  if (rval != 1)
    /* The contents of statm are formatted in a way we don't understand.  */
    return -1;

  /* The values in statm are in units of the system's page size.
     Convert the RSS to kilobytes.  */
  return rss * getpagesize () / 1024;
}

/* Generate an HTML table row for process PID.  The return value is a
   pointer to a buffer that the caller must deallocate with free, or
   NULL if an error occurs.  */

static char* format_process_info (pid_t pid)
{
  int rval;
  uid_t uid;
  gid_t gid;
  char* user_name;
  char* group_name;
  int rss;
  char* program_name;
```

continues

Listing 11.9 **Continued**

```
  size_t result_length;
  char* result;

  /* Obtain the process's user and group IDs.  */
  rval = get_uid_gid (pid, &uid, &gid);
  if (rval != 0)
    return NULL;
  /* Obtain the process's RSS.  */
  rss = get_rss (pid);
  if (rss == -1)
    return NULL;
  /* Obtain the process's program name.  */
  program_name = get_program_name (pid);
  if (program_name == NULL)
    return NULL;
  /* Convert user and group IDs to corresponding names.  */
  user_name = get_user_name (uid);
  group_name = get_group_name (gid);

  /* Compute the length of the string we'll need to hold the result, and
     allocate memory to hold it.  */
  result_length = strlen (program_name)
    + strlen (user_name) + strlen (group_name) + 128;
  result = (char*) xmalloc (result_length);
  /* Format the result.  */
  snprintf (result, result_length,
            "<tr><td align=\"right\">%d</td><td><tt>%s</tt></td><td>%s</td>"
            "<td>%s</td><td align=\"right\">%d</td></tr>\n",
            (int) pid, program_name, user_name, group_name, rss);
  /* Clean up.  */
  free (program_name);
  free (user_name);
  free (group_name);
  /* All done.  */
  return result;
}

/* HTML source for the start of the process listing page.  */

static char* page_start =
  "<html>\n"
  " <body>\n"
  "  <table cellpadding=\"4\" cellspacing=\"0\" border=\"1\">\n"
  "   <thead>\n"
  "    <tr>\n"
  "     <th>PID</th>\n"
  "     <th>Program</th>\n"
  "     <th>User</th>\n"
  "     <th>Group</th>\n"
```

```
  "     <th>RSS (KB)</th>\n"
  "    </tr>\n"
  "   </thead>\n"
  "   <tbody>\n";

/* HTML source for the end of the process listing page.  */

static char* page_end =
  "   </tbody>\n"
  "  </table>\n"
  " </body>\n"
  "</html>\n";

void module_generate (int fd)
{
  size_t i;
  DIR* proc_listing;

  /* Set up an iovec array.  We'll fill this with buffers that'll be
     part of our output, growing it dynamically as necessary.  */

  /* The number of elements in the array that we've used.  */
  size_t vec_length = 0;
  /* The allocated size of the array.  */
  size_t vec_size = 16;
  /* The array of iovcec elements.  */
  struct iovec* vec =
    (struct iovec*) xmalloc (vec_size * sizeof (struct iovec));

  /* The first buffer is the HTML source for the start of the page.  */
  vec[vec_length].iov_base = page_start;
  vec[vec_length].iov_len = strlen (page_start);
  ++vec_length;

  /* Start a directory listing for /proc.  */
  proc_listing = opendir ("/proc");
  if (proc_listing == NULL)
    system_error ("opendir");

  /* Loop over directory entries in /proc.  */
  while (1) {
    struct dirent* proc_entry;
    const char* name;
    pid_t pid;
    char* process_info;

    /* Get the next entry in /proc.  */
    proc_entry  = readdir (proc_listing);
    if (proc_entry == NULL)
      /* We've hit the end of the listing.  */
      break;
```

continues

Listing 11.9 **Continued**

```
    /* If this entry is not composed purely of digits, it's not a
       process directory, so skip it.  */
    name = proc_entry->d_name;
    if (strspn (name, "0123456789") != strlen (name))
      continue;
    /* The name of the entry is the process ID.  */
    pid = (pid_t) atoi (name);
    /* Generate HTML for a table row describing this process.  */
    process_info = format_process_info (pid);
    if (process_info == NULL)
      /* Something went wrong.  The process may have vanished while we
         were looking at it.  Use a placeholder row instead.  */
      process_info = "<tr><td colspan=\"5\">ERROR</td></tr>";

    /* Make sure the iovec array is long enough to hold this buffer
       (plus one more because we'll add an extra element when we're done
       listing processes).  If not, grow it to twice its current size.  */
    if (vec_length == vec_size - 1) {
      vec_size *= 2;
      vec = xrealloc (vec, vec_size * sizeof (struct iovec));
    }
    /* Store this buffer as the next element of the array.  */
    vec[vec_length].iov_base = process_info;
    vec[vec_length].iov_len = strlen (process_info);
    ++vec_length;
  }

  /* End the directory listing operation.  */
  closedir (proc_listing);

  /* Add one last buffer with HTML that ends the page.  */
  vec[vec_length].iov_base = page_end;
  vec[vec_length].iov_len = strlen (page_end);
  ++vec_length;

  /* Output the entire page to the client file descriptor all at once.  */
  writev (fd, vec, vec_length);

  /* Deallocate the buffers we created.  The first and last are static
     and should not be deallocated.  */
  for (i = 1; i < vec_length - 1; ++i)
    free (vec[i].iov_base);
  /* Deallocate the iovec array.  */
  free (vec);
}
```

Gathering process data and formatting it as an HTML table is broken down into several simpler operations:

- `get_uid_gid` extracts the IDs of the owning user and group of a process. To do this, the function invokes `stat` (see Section B.2, "`stat`," in Appendix B) on the process's subdirectory in `/proc` (see Section 7.2, "Process Entries," in Chapter 7). The user and group that own this directory are identical to the process's owning user and group.
- `get_user_name` returns the username corresponding to a UID. This function simply calls the C library function `getpwuid`, which consults the system's `/etc/passwd` file and returns a copy of the result. `get_group_name` returns the group name corresponding to a GID. It uses the `getgrgid` call.
- `get_program_name` returns the name of the program running in a specified process. This information is extracted from the `stat` entry in the process's directory under `/proc` (see Section 7.2, "Process Entries," in Chapter 7). We use this entry rather than examining the `exe` symbolic link (see Section 7.2.4, "Process Executable," in Chapter 7) or `cmdline` entry (see Section 7.2.2, "Process Argument List," in Chapter 7) because the latter two are inaccessible if the process running the server isn't owned by the same user as the process being examined. Also, reading from `stat` doesn't force Linux to page the process under examination back into memory, if it happens to be swapped out.
- `get_rss` returns the resident set size of a process. This information is available as the second element in the contents of the process's `statm` entry (see Section 7.2.6, "Process Memory Statistics," in Chapter 7) in its `/proc` subdirectory.
- `format_process_info` generates a string containing HTML elements for a single table row, representing a single process. After calling the functions listed previously to obtain this information, it allocates a buffer and generates HTML using `snprintf`.
- `module_generate` generates the entire HTML page, including the table. The output consists of one string containing the start of the page and the table (in `page_start`), one string for each table row (generated by `format_process_info`), and one string containing the end of the table and the page (in `page_end`).

  `module_generate` determines the PIDs of the processes running on the system by examining the contents of `/proc`. It obtains a listing of this directory using `opendir` and `readdir` (see Section B.6, "Reading Directory Contents," in Appendix B). It scans the contents, looking for entries whose names are composed entirely of digits; these are taken to be process entries.

  Potentially a large number of strings must be written to the client socket—one each for the page start and end, plus one for each process. If we were to write each string to the client socket file descriptor with a separate call to `write`, this would generate unnecessary network traffic because each string may be sent in a separate network packet.

To optimize packing of data into packets, we use a single call to `writev` instead (see Section B.3, "Vector Reads and Writes," in Appendix B). To do this, we must construct an array of `struct iovec` objects, `vec`. However, because we do not know the number of processes beforehand, we must start with a small array and expand it as new processes are added. The variable `vec_length` contains the number of elements of `vec` that are used, while `vec_size` contains the allocated size of `vec`. When `vec_length` is about to exceed `vec_size`, we expand `vec` to twice its size by calling `xrealloc`. When we're done with the vector write, we must deallocate all of the dynamically allocated strings pointed to by `vec`, and then `vec` itself.

11.4 Using the Server

If we were planning to distribute this program in source form, maintain it on an ongoing basis, or port it to other platforms, we probably would want to package it using GNU Automake and GNU Autoconf, or a similar configuration automation system. Such tools are outside the scope of this book; for more information about them, consult *GNU Autoconf, Automake, and Libtool* (by Vaughan, Elliston, Tromey, and Taylor, published by New Riders, 2000).

11.4.1 The *Makefile*

Instead of using Autoconf or a similar tool, we provide a simple `Makefile` compatible with GNU Make[4] so that it's easy to compile and link the server and its modules. The `Makefile` is shown in Listing 11.10. See the info page for GNU Make for details of the file's syntax.

Listing 11.10 **(*Makefile*) GNU Make Configuration File for Server Example**

```
### Configuration.  #####################################################

# Default C compiler options.
CFLAGS                = -Wall -g
# C source files for the server.
SOURCES               = server.c module.c common.c main.c
# Corresponding object files.
OBJECTS               = $(SOURCES:.c=.o)
# Server module shared library files.
MODULES               = diskfree.so issue.so processes.so time.so

### Rules.  ############################################################

# Phony targets don't correspond to files that are built; they're names
# for conceptual build targets.
.PHONY:         all clean
```

4. GNU Make comes installed on GNU/Linux systems.

```
# Default target: build everything.
all:            server $(MODULES)

# Clean up build products.
clean:
        rm -f $(OBJECTS) $(MODULES) server

# The main server program.  Link with -Wl,-export-dyanamic so
# dynamically loaded modules can bind symbols in the program.  Link in
# libdl, which contains calls for dynamic loading.
server:         $(OBJECTS)
        $(CC) $(CFLAGS) -Wl,-export-dynamic -o $@ $^ -ldl

# All object files in the server depend on server.h.  But use the
# default rule for building object files from source files.
$(OBJECTS):     server.h

# Rule for building module shared libraries from the corresponding
# source files.   Compile -fPIC and generate a shared object file.
$(MODULES): \
%.so:           %.c server.h
        $(CC) $(CFLAGS) -fPIC -shared -o $@ $<
```

The `Makefile` provides these targets:

- `all` (the default if you invoke `make` without arguments because it's the first target in the `Makefile`) includes the `server` executable and all the modules. The modules are listed in the variable `MODULES`.
- `clean` deletes any build products that are produced by the `Makefile`.
- `server` links the server executable. The source files listed in the variable `SOURCES` are compiled and linked in.
- The last rule is a generic pattern for compiling shared object files for server modules from the corresponding source files.

Note that source files for server modules are compiled with the `-fPIC` option because they are linked into shared libraries (see Section 2.3.2, "Shared Libraries," in Chapter 2).

Also observe that the `server` executable is linked with the `-Wl,-export-dynamic` compiler option. With this option, GCC passes the `-export-dynamic` option to the linker, which creates an executable file that also exports its external symbols as a shared library. This allows modules, which are dynamically loaded as shared libraries, to reference functions from `common.c` that are linked statically into the `server` executable.

11.4.2 Building the Server

Building the program is easy. From the directory containing the sources, simply invoke `make`:

```
% make
cc -Wall -g   -c -o server.o server.c
cc -Wall -g   -c -o module.o module.c
cc -Wall -g   -c -o common.o common.c
cc -Wall -g   -c -o main.o main.c
cc -Wall -g -Wl,-export-dynamic -o server server.o module.o common.o main.o -ldl
cc -Wall -g -fPIC -shared -o diskfree.so diskfree.c
cc -Wall -g -fPIC -shared -o issue.so issue.c
cc -Wall -g -fPIC -shared -o processes.so processes.c
cc -Wall -g -fPIC -shared -o time.so time.c
```

This builds the `server` program and the server module shared libraries.

```
% ls -l server *.so
-rwxr-xr-x    1 samuel   samuel      25769 Mar 11 01:15 diskfree.so
-rwxr-xr-x    1 samuel   samuel      31184 Mar 11 01:15 issue.so
-rwxr-xr-x    1 samuel   samuel      41579 Mar 11 01:15 processes.so
-rwxr-xr-x    1 samuel   samuel      71758 Mar 11 01:15 server
-rwxr-xr-x    1 samuel   samuel      13980 Mar 11 01:15 time.so
```

11.4.3 Running the Server

To run the server, simply invoke the `server` executable.

If you do not specify the server port number with the `--port` (`-p`) option, Linux will choose one for you; in this case, specify `--verbose` (`-v`) to make the server print out the port number in use.

If you do not specify an address with `--address` (`-a`), the server runs on all your computer's network addresses. If your computer is attached to a network, that means that others will be capable of accessing the server, provided that they know the correct port number to use and page to request. For security reasons, it's a good idea to specify the `localhost` address until you're confident that the server works correctly and is not releasing any information that you prefer to not make public. Binding to the `localhost` causes the server to bind to the local network device (designated "lo")—only programs running on the same computer can connect to it. If you specify a different address, it must be an address that corresponds to your computer:

```
% ./server --address localhost --port 4000
```

The server is now running. Open a browser window, and attempt to contact the server at this port number. Request a page whose name matches one of the modules. For instance, to invoke the `diskfree.so` module, use this URL:

```
http://localhost:4000/diskfree
```

Instead of `4000`, enter the port number you specified (or the port number that Linux chose for you). Press Ctrl+C to kill the server when you're done.

If you didn't specify `localhost` as the server address, you can also connect to the server with a Web browser running on another computer by using your computer's hostname in the URL—for example:

```
http://host.domain.com:4000/diskfree
```

If you specify the `--verbose` (`-v`) option, the server prints some information at startup and displays the numerical Internet address of each client that connects to it. If you connect via the `localhost` address, the client address will always be 127.0.0.1.

If you experiment with writing your own server modules, you may place them in a different directory than the one containing the `server` module. In this case, specify that directory with the `--module-dir` (`-m`) option. The server will look in this directory for server modules instead.

If you forget the syntax of the command-line options, invoke `server` with the `--help` (`-h`) option.

```
% ./server --help
Usage: ./server [ options ]
  -a, --address ADDR        Bind to local address (by default, bind
                              to all local addresses).
  -h, --help                Print this information.
  -m, --module-dir DIR      Load modules from specified directory
                              (by default, use executable directory).
  -p, --port PORT           Bind to specified port.
  -v, --verbose             Print verbose messages.
```

11.5 Finishing Up

If you were really planning on releasing this program for general use, you'd need to write documentation for it as well. Many people don't realize that writing good documentation is just as difficult and time-consuming—and just as important—as writing good software. However, software documentation is a subject for another book, so we'll leave you with a few references of where to learn more about documenting GNU/Linux software.

You'd probably want to write a man page for the `server` program, for instance. This is the first place many users will look for information about a program. Man pages are formatted using a classic UNIX formatting system `troff`. To view the man page for `troff`, which describes the format of troff files, invoke the following:

```
% man troff
```

To learn about how GNU/Linux locates man pages, consult the man page for the `man` command itself by invoking this:

```
% man man
```

You might also want to write info pages, using the GNU Info system, for the server and its modules. Naturally, documentation about the info system comes in info format; to view it, invoke this line:

```
% info info
```

Many GNU/Linux programs come with documentation in plain text or HTML formats as well.

Happy GNU/Linux programming!

III

Appendixes