# 1 Writing Good GNU/Linux Software

## 1.1 Interaction With the Execution Environment

### 1.1.1 GNU/Linux Command-Line Conventions

- 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).
- the command **ls -s /** displays the contents of the root directory.
- The −−size option is synonymous with -s, so the same command could have been invoked as **ls −−size /**.

### 1.1.2 Using getopt_long

- The GNU C library provides a 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 1. In addition, the program is to

Table 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 |

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; one for short options, each a single letter and one for long options, you construct an array of struct option elements. getopt_long.c

```
#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' },
    { 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 non-option 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;
}
```

### 1.1.3 Standard I/O

- The standard C library provides standard input and output streams (**stdin** and **stdout**, respectively).
- The C library also provides **stderr**, the standard error stream. Programs should print warning and error messages to standard error instead of standard output.
- The **fprintf** function can be used to print to **stderr**, for example:

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

- 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.

- 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);
  }
  ```

### 1.1.4 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.

```
$ ls /
$ echo $?
0
$ ls bogusfile
ls: bogusfile: No such file or directory
$ echo $?
1
```

### 1.1.5 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.
  - 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.
  - 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
$ echo $HOME
```

– 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:

```
$ export EDITOR=emacs
```

- In a program, you access an environment variable with the **getenv** function in $< stdlib.h >$.
- To set or clear environment variables, use the **setenv** and **unsetenv** functions, respectively. print_env.c

```
#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. client.c

```
#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;
}

$ client
accessing server server.my-company.com
$ export SERVER_NAME=backup-server.elsewhere.net
$ client
accessing server backup-server.elsewhere.net
```

### 1.1.6 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
- ***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. temp_file.c

```
#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;
}
```

## 1.2 Coding Defensively

### 1.2.1 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.
- 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. Locate the buffer.
  2. Open the file.
  3. Read from the file into the buffer.
  4. Close the file.
  5. Return the buffer.

  readfile.c

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

```
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.

## 1.3 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.
- 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. If you link dynamically, your programs will be smaller, easier to upgrade.

### 1.3.1 Archives

- An **archive** (or static library) is simply a collection of object files stored as a single 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.

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

The *cr* flags tell ar to create the archive. Now you can link with this archive using the **-ltest** option with gcc or g++

- 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. test.c

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

app.c

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

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
```

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.

### 1.3.2 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.
- 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 contains a reference to the shared library.
- Thus, the library is "shared" among all the programs that link with it.
- A second important difference is that a shared library is not a collection of object files. 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
  ```

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

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

- 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.

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

- Suppose that both **libtest.a** and **libtest.so** are available. 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.

```
$ gcc -static -o app app.o -L. -ltest
```

- ***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.
  - 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.

### 1.3.3 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).

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

int main (int argc, char** argv)
{
  TIFF* tiff;
  tiff = TIFFOpen (argv[1], "r");
  TIFFClose (tiff);
  return 0;
}
$ gcc -o tifftest tifftest.c -ltiff
$ 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
```

### 1.3.4 Pros and Cons

- One major advantage of a shared library is that it saves space on the system.
- A related advantage to shared libraries is that users can upgrade the libraries without upgrading all the programs that depend on them.
- 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.
- If you're not going to be able to install your libraries in /lib or /usr/lib.