The Linux GCC HOWTO

Daniel Barlow

Linux Documentation Project

May 1999

This document covers how to set up the GNU C compiler and development libraries under Linux, and gives an overview of compiling, linking, running and debugging programs under it. Most of the material in it has been taken from Mitch D'Souza's GCC−FAQ or the ELF−HOWTO − it replaces both documents.

This is the first version to be written in DocBook instead of the old Linuxdoc format, and may contain markup errors. Please let me know if you find anything worng.

As can be determined from the long times between updates of this document, I don't actually have the time or inclination to maintain it much. If you have, can, and want to, drop me some email describing what you'd do with it and why you think you'd be good at it.

Table of Contents

Table of Contents

1. Preliminaries

1.1. ELF vs. a.out, libc 5 vs 6

Three years ago when this document was first created, I opened this section by saying "Linux development is in a state of flux right now" and going on to describe how ELF was replacing the older a.out binary format.

It still is in a state of flux. It always will be. Though that particular change is long since past, development of the Linux kernel and the surrounding system continues to happen, and things change for developers as a result. So it's a good idea to know upfront what kind of system you have in front of you.

The possible candidates, in order of age, are

- libc 4, a.out: very old systems
- libc 5, ELF: Red Hat 4.2, Debian 2.0
- libc 6 (a.k.a glibc 2), ELF: Red Hat 5 – 5.2, Debian 2.1
- libc 6.1,(a.k.a glibc 2.1) ELF: Red Hat 6

How to tell? The simplest approach is to pick a binary that you consider is typical (e.g. `/bin/ls` and run **ldd** on it. One of the listed libraries should be libc – check its version number.

```
$ ldd /bin/ls
        libc.so.6 => /lib/libc.so.6 (0x4000e000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

This document was created on a Debian 2.1 system, so no surprise there.

It's entirely possible that the system you're using may have a mix of different versions on it. What you probably want to know in that case is the version that its C development environment is set up for, so you're best off compiling "hello world" and running **ldd** on the output thus created. Note that for historical reasons, **gcc** defaults to an output file called `a.out` even on ELF systems, so don't assume anything from that.

1.2. Administrata

The copyright information and like legalese can be found at the *end* of this document, together with the statutory warnings about asking dumb questions on Usenet, revealing your ignorance of the C language by reporting bugs which aren't, and picking your nose while chewing gum.

1.3. Typography

If you're reading this in Postscipt, dvi, or html format, you get to see a little more font variation than people with the plain text version. In particular, filenames, commands, command output and source code excerpts are set in some form of `typewriter` font, whereas `variables' and random things that need emphasizing are *emphasized*.

You also get a usable index. In dvi or postscript, the numbers in the index are section numbers. In HTML they're just sequentially assigned numbers that you can click on. In the plain text version, they really are just numbers. Get an upgrade!

The Bourne (rather than C) shell syntax is used in examples. C shell users will want to use

```
% setenv FOO bar
```

where I have written

```
$ FOO=bar; export FOO
```

If the prompt shown is `#` rather than `$`, the command shown will probably only work as root. Of course, I accept no responsibility for anything that happens to your system as a result of trying these examples. Have a nice day `:-)`

---

2. Where to get things

In the three years since the first `HOWTO' version of this, useful Linux distributions have become prevalent. So, where once I'd have spent pages listing FTP sites and hours updating (failing to update) version numbers and directory names, now I will simply say – your distribution maintainer should be taking care of this for you. If you don't have, say, gcc installed, find the RPM or the deb packages that contain it, and install it. If that isn't an option because you don't have a friendly distribution, you've almost certainly been using Linux long enough that you don't need me to tell you where to find things anyway.

2.1. This document

You're reading it. You probably have it already.

This document is one of the Linux HOWTO series, so is probably already installed somewhere in `/usr/doc` if you're reading this on a linux box. Failing that, from all Linux HOWTO repositories (try Metalab) and (possibly in a slightly newer version) at my personal web site www.telent.net.

2.2. Other documentation

The official documentation for gcc is in the source distribution (see below) as texinfo files, and as `.info` files. If you have a fast network connection, a cdrom, or a reasonable amount of patience, you can just untar it and copy the relevant bits into `/usr/info`. If not, you may find them at tsx−11, but not necessarily always the latest version.

There are two source of documentation for libc. GNU libc comes with info files which describe Linux libc fairly accurately except for stdio. Also, the manpages archive are written for Linux and describe a lot of system calls (section 2) and libc functions (section 3).

2.3. GCC

There are two answers.

(a) The official Linux GCC distribution can always be found in binary (ready−compiled) form at . At the time of writing, 2.7.2 (`gcc−2.7.2.bin.tar.gz`) is the latest version.

(b) The latest source distribution of GCC from the Free Software Foundation can be had from GNU archives. This is not necessarily always the same version as above, though it is just now. The Linux GCC maintainer(s) have made it easy for you to compile the latest version available yourself −−− the `configure` script should set it all up for you. Check tsx−11 as well, for patches which you may want to apply.

To compile anything non−trivial (and quite a few trivial things also) you will also need the

2.4. C library and header files

What you want here depends on (i) whether your system is ELF or a.out, and (ii) which you want it to be. If you're upgrading from libc 4 to libc 5, you are recommended to look at the ELF−HOWTO from approximately the same place as you found this document.

These are available from tsx−11 as above:

libc−5.2.18.bin.tar.gz

−−− ELF shared library images, static libraries and include files for the C and maths libraries.

libc−5.2.18.tar.gz

−−− Source for the above. You will also need the `.bin.` package for the header files. If you are deliberating whether to compile the C library yourself or use the binaries, the right answer in nearly all cases is to use the binaries. You will however need to roll your own if you want NYS or shadow password support.

libc−4.7.5.bin.tar.gz

−−− a.out shared library images and static libraries for version 4.7.5 of the C library and friends. This is designed to coexist with the libc 5 package above, but is only really necessary if you wish to keep using/developing a.out format programs.

2.5. Associated tools (as, ld, ar, strings etc)

From tsx−11, just like everything else so far. The current version is `binutils−2.6.0.2.bin.tar.gz`.

Note that the binutils are only available in ELF, the current libc version is in ELF and the a.out libc is happiest when used in conjunction with an ELF libc. C library development is moving emphatically ELFwards, and unless you have really good reasons for needing a.out things you're encouraged to follow suit.

---

3. GCC installation and setup

3.1. GCC versions

You can find out what GCC version you're running by typing `gcc -v` at the shell prompt. This is also a fairly reliable way to find out whether you are set up for ELF or a.out. On my system it does

```
$ gcc -v
Reading specs from /usr/lib/gcc-lib/i486-box-linux/2.7.2/specs
gcc version 2.7.2
```

The key things to note here are

- `i486`. This indicates that the gcc you are using was built for a 486 processor −−− you might have 386 or 586 instead. All of these chips can run code compiled for each of the others; the difference is that the 486 code has added padding in some places so runs faster on a 486. This has no detrimental performance effect on a 386, but does make the binaries slightly larger.
- `box`. This is *not* at all important, and may say something else (such as `slackware` or `debian`) or nothing at all (so that the complete directory name is `i486-linux`). If you build your own gcc, you can set this at build time for cosmetic effect. Just like I did `:-)`
- `linux`. This may instead say `linuxelf` or `linuxaout`, and, confusingly, the meaning of each varies according to the version that you are using.
    - `linux` means ELF if the version is 2.7.0 or newer, a.out otherwise.
    - `linuxaout` means a.out. It was introduced as a target when the definition of `linux` was changed from a.out to ELF, so you won't see any `linuxaout` gcc older than 2.7.0.
    - `linuxelf` is obsolete. It is generally a version of gcc 2.6.3 set to produce ELF executables. Note that gcc 2.6.3 has known bugs when producing code for ELF −−− an upgrade is advisable.
- `2.7.2` is the version number.

So, in summary, I have gcc 2.7.2 producing ELF code. Quelle surprise.

3.2. Where did it go?

If you installed gcc without watching, or if you got it as part of a distribution, you may like to find out where it lives in the filesystem. The key bits are

- `/usr/lib/gcc-lib/`*target*`/`*version*`/` (and subdirectories) is where most of the compiler lives. This includes the executable programs that do actual compiling, and some version−specific libraries and include files.

- `/usr/bin/gcc` is the compiler driver −−− the bit that you can actually run from the command line. This can be used with multiple versions of gcc provided that you have multiple compiler directories (as above) installed. To find out the default version it will use, type `gcc −v`. To force it to another version, type `gcc −V` *version*. For example

```
# gcc −v
Reading specs from /usr/lib/gcc−lib/i486−box−linux/2.7.2/specs
gcc version 2.7.2
# gcc −V 2.6.3 −v
Reading specs from /usr/lib/gcc−lib/i486−box−linux/2.6.3/specs
gcc driver version 2.7.2 executing gcc version 2.6.3
```

- `/usr/`*target*`/(bin|lib|include)/`. If you have multiple targets installed (for example, a.out and elf, or a cross−compiler of some sort, the libraries, binutils (`as`, `ld` and so on) and header files for the non−native target(s) can be found here. Even if you only have one kind of gcc installed you might find anyway that various bits for it are kept here. If not, they're in `/usr/(bin|lib|include)`.
- `/lib/`,`/usr/lib` and others are library directories for the native system. You will also need `/lib/cpp` for many applications (X makes quite a lot of use of it) −−− either copy it from `/usr/lib/gcc−lib/`*target*`/`*version*`/` or make a symlink pointing there.

3.3. Where are the header files?

Apart from whatever you install yourself under `/usr/local/include`, there are three main sources of header files in Linux:

- Most of `/usr/include/` and its subdirectories are supplied with the libc binary package from H J Lu. I say `most' because you may also have files from other sources (`curses` and `dbm` libraries, for example) in here, especially if you are using the newest libc distribution (which doesn't come with curses or dbm, unlike the older ones).
- `/usr/include/linux` and `/usr/include/asm` (for the files `<linux/*.h>` and `<asm/*.h>`) should be symbolic links to the directories `linux/include/linux` and `linux/include/asm` in the kernel source distribution. You need to install these if you plan to do *any* non−trivial development; they are not just there for compiling the kernel. You might find also that you need to do `make config` in the kernel directory after unpacking the sources. Many files depend on `<linux/autoconf.h>` which otherwise may not exist, and in some kernel versions `asm` is a symbolic link itself and only created at `make config` time. So, if you unpack your kernel sources under `/usr/src/linux`, that's

```
$ cd /usr/src/linux
$ su
# make config
[answer the questions.  Unless you're going to go on and build the kernel
it doesn't matter _too_ much what you say]
# cd /usr/include
# ln −s ../src/linux/include/linux .
# ln −s ../src/linux/include/asm .
```

- Files such as `<float.h>`, `<limits.h>`, `<varargs.h>`, `<stdarg.h>` and `<stddef.h>` vary according to the compiler version, so are found in

`/usr/lib/gcc-lib/i486-box-linux/2.7.2/include/` and places of that ilk.

3.4. Building cross compilers

3.4.1. Linux as the target platform

Assuming you have obtained the source code to gcc, usually you can just follow the instructions given in the INSTALL file for GCC. A `configure --target=i486-linux --host=XXX` on platform `XXX` followed by a `make` should do the trick. Note that you will need the Linux includes, the kernel includes, and also to build the cross assembler and cross linker from the sources in .

3.4.2. Linux as the source platform, MSDOS as the target

Ugh. Apparently this is somewhat possible by using the "emx" package or the "go" extender. Please look at .

I have not tested this and cannot vouch for its abilities.

4. Porting and Compiling

4.1. Automatically defined symbols

You can find out what symbols your version of gcc defines automatically by running it with the `-v` switch. For example, mine does:

```
$ echo 'main(){printf("hello world\n");}' | gcc -E -v -
Reading specs from /usr/lib/gcc-lib/i486-box-linux/2.7.2/specs
gcc version 2.7.2
 /usr/lib/gcc-lib/i486-box-linux/2.7.2/cpp -lang-c -v -undef
-D__GNUC__=2 -D__GNUC_MINOR__=7 -D__ELF__ -Dunix -Di386 -Dlinux
-D__ELF__ -D__unix__ -D__i386__ -D__linux__ -D__unix -D__i386
-D__linux -Asystem(unix) -Asystem(posix) -Acpu(i386)
-Amachine(i386) -D__i486__ -
```

If you are writing code that uses Linux−specific features, it is a good idea to enclose the nonportable bits in

```
#ifdef __linux__
/* ... funky stuff ... */
#endif /* linux */
```

Use `__linux__` for this purpose, *not* `linux`. Although the latter is defined, it is not POSIX compliant.

4.2. Compiler invocation

The documentation for compiler switches is the gcc info page (in Emacs, use `C-h i` then select the `gcc' option). Your distributor may not have packed this with your system, or you may have an old version; the best thing to do in this case is to download the gcc source archive from or one of its mirrors, and copy them out of it.

The gcc manual page (`gcc.1`) is, generally speaking, out of date. It will warn you of this when you try to look at it.

4.2.1. Compiler flags

gcc can be made to optimize its output code by adding `-O`*n* to its command line, where *n* is an optional small integer. Meaningful values of *n*, and their exact effect, vary according to the exact version, but typically it ranges from 0 (no optimization) to 2 (lots) or 3 (lots and lots).

Internally, gcc translates these to a series of `-f` and `-m` options. You can see exactly which `-O` levels map to which options by running gcc with the `-v` flag and the (undocumented) `-Q` flag. For example, for `-O2`, mine says

```
enabled: -fdefer-pop -fcse-follow-jumps -fcse-skip-blocks
-fexpensive-optimizations
         -fthread-jumps -fpeephole -fforce-mem -ffunction-cse -finline
         -fcaller-saves -fpcc-struct-return -frerun-cse-after-loop
         -fcommon -fgnu-linker -m80387 -mhard-float -mno-soft-float
         -mno-386 -m486 -mieee-fp -mfp-ret-in-387
```

Using an optimization level higher than your compiler supports (e.g. `-O6`) will have exactly the same effect as using the highest level that it *does* support. Distributing code which is set to compile this way is a poor idea though --- if further optimisations are incorporated into future versions, you (or your users) may find that they break your code.

Users of gcc 2.7.0 thru 2.7.2 should note that there is a bug in `-O2` on these. Specifically, strength reduction doesn't work. A patch can be had to fix this if you feel like recompiling gcc, otherwise make sure that you always compile with `-fno-strength-reduce`

4.2.1.1. Processor-specific

There are other `-m` flags which aren't turned on by any variety of `-O` but are nevertheless useful. Chief among these are `-m386` and `-m486`, which tell gcc to favour the 386 or 486 respectively. Code compiled with one of these will still work on the other; 486 code is bigger, but otherwise not slower on the 386.

There is currently no `-mpentium` or `-m586`. Linus suggests using `-m486 -malign-loops=2 -malign-jumps=2 -malign-functions=2`, to get 486 code optimisations but without the big gaps for alignment (which the pentium doesn't need). Michael Meissner (of Cygnus) says

"My hunch is that `-mno-strength-reduce` also results in faster code on the x86 (note, I'm not talking about the strength reduction bug, which is another issue). This is because the x86 is rather register starved (and GCC's method of grouping registers into spill registers vs. other registers doesn't help either). Strength reduction typically results in using additional registers to replace multiplications with addition. I also suspect `-fcaller-saves` may also be a loss." "Another hunch is that `-fomit-frame-pointer` might or might not be a win. On the one hand, it can mean that another register is available for allocation. On the other hand, the way the x86 encodes its instruction set, means that stack relative addresses take more space instead of frame relative addresses, which means slightly less Icache availble to the program. Also, `-fomit-frame-pointer`, means that the compiler has to constantly adjust the stack pointer after calls, while with a frame, it can let the stack accumulate for a few calls."

The final word on this subject is from Linus again:

"Note that if you want to get optimal performance, don't believe me: test. There are lots of gcc compiler switches, and it may be that a particular set gives the best optimizations for you. "

4.2.2. `Internal compiler error: cc1 got fatal signal 11`

Signal 11 is SIGSEGV, or `segmentation violation'. Usually it means that the program got its pointers confused and tried to write to memory it didn't own. So, it could be a gcc bug.

gcc is however, a well tested and reliable piece of software, for the most part. It also uses a large number of complex data structures, and an awful lot of pointers. In short, it's the pickiest RAM tester commonly available. If you *can't duplicate the bug* −−− if it doesn't stop in the same place when you restart the compilation −−− it's almost certainly a problem with your hardware (CPU, memory, motherboard or cache). *Don't* claim it as a bug because your computer passes the power−on checks or runs Windows ok or whatever; these `tests' are commonly and rightly held to be worthless. And don't claim it's a bug because a kernel compile always stops during ``make zImage`' −−− of course it will! ``make zImage`' is probably compiling over 200 files; we're looking for a slightly *smaller* place than that.

If you can duplicate the bug, and (better) can produce a short program that exhibits it, you can submit it as a bug report to the FSF, or to the linux−gcc mailing list. See the gcc documentation for details of exactly what information they need.

4.3. Portability

It has been said that, these days, if something hasn't been ported to Linux then it is not worth having :−)

Seriously though, in general only minor changes are needed to the sources to get over Linux's 100% POSIX compliance. It is also worthwhile passing back any changes to authors of the code such that in the future only `make' need be called to provide a working executable.

4.3.1. BSDisms (including `bsd_ioctl`, `daemon` and `<sgtty.h>`)

You can compile your program with `−I/usr/include/bsd` and link it with `−lbsd` (i.e. add `−I/usr/include/bsd` to `CFLAGS` and `−lbsd` to the `LDFLAGS` line in your Makefile). There is *no* need to add `−D__USE_BSD_SIGNAL` any more if you want BSD type signal behavior, as you get this automatically when you have `−I/usr/include/bsd` and include `<signal.h>`.

4.3.2. `Missing' signals (`SIGBUS`, `SIGEMT`, `SIGIOT`, `SIGTRAP`, `SIGSYS` etc)

Linux is POSIX compliant. These are not POSIX−defined signals −−− ISO/IEC 9945−1:1990 (IEEE Std 1003.1−1990), paragraph B.3.3.1.1 sez:

"``The signals SIGBUS, SIGEMT, SIGIOT, SIGTRAP, and SIGSYS were omitted from POSIX.1 because their behavior is implementation dependent and could not be adequately categorized. Conforming implementations may deliver these signals, but must document the circumstances under which they are delivered and note any restrictions concerning their delivery.''"

The cheap and cheesy way to fix this is to redefine these signals to `SIGUNUSED`. The *correct* way is to bracket the code that handles them with appropriate `#ifdefs`:

```
#ifdef SIGSYS
/* ... non-posix SIGSYS code here .... */
#endif
```

4.3.3. K & R Code

GCC is an ANSI compiler; much existing code is not ANSI. There's really not much that can be done about this, except to add `-traditional` to the compiler flags. There is a certain amount of finer-grained control over which varieties of brain damage to emulate; consult the gcc info page.

Note that `-traditional` has effects beyond just changing the language that gcc accepts. For example, it turns on `-fwritable-strings`, which moves string constants into data space (from text space, where they cannot be written to). This increases the memory footprint of the program.

4.3.4. Preprocessor symbols conflict with prototypes in the code

One of the most frequent problems is that some common functions are defined as macros in Linux's header files and the preprocessor will refuse to parse similar prototype definitions in the code. Common ones are `atoi()` and `atol()`.

4.3.5. `sprintf()`

Something to be aware of, especially when porting from SunOS, is that `sprintf(string, fmt, ...)` returns a pointer to `string` on many unices, whereas Linux (following ANSI) returns the number of characters which were put into the string.

4.3.6. `fcntl` and friends. Where are the definitions of `FD_*` stuff ?

In `<sys/time.h>`. If you are using `fcntl` you probably want to include `<unistd.h>` too, for the actual prototype.

Generally speaking, the manual page for a function lists the necessary `#includes` in its SYNOPSIS section.

4.3.7. The `select()` timeout. Programs start busy-waiting.

The BSD manual page for select(2) used to say "select() should probably return the time remaining from the original timeout, if any, by modifying the time value in place. This may be implemented in future versions of the system. Thus, it is unwise to assume that the timeout pointer will be unmodified by the select() call."

Some versions of Linux do perform this modification. Some don't. It is incredibly unwise to assume one behaviour or the other.

To fix, put the timeout value into that structure every time you call `select()`. Change code like

```
      struct timeval timeout;
      timeout.tv_sec = 1; timeout.tv_usec = 0;
      while (some_condition)
            select(n,readfds,writefds,exceptfds,38;timeout);
```

to, say,

```
      struct timeval timeout;
      while (some_condition) {
            timeout.tv_sec = 1; timeout.tv_usec = 0;
            select(n,readfds,writefds,exceptfds,38;timeout);
      }
```

Some versions of Mosaic were at one time notable for this problem. The speed of the spinning globe animation was inversely related to the speed that the data was coming in from the network at!

4.3.8. Interrupted system calls.

4.3.8.1. Symptom:

When a program is stopped using Ctrl−Z and then restarted − or in other situations that generate signals: Ctrl−C interruption, termination of a child process etc. − it complains about "interrupted system call" or "write: unknown error" or things like that.

4.3.8.2. Problem:

POSIX systems check for signals a bit more often than some older unices. Linux may execute signal handlers −−−

* asynchronously (at a timer tick)
* on return from any system call
* during the execution of the following system calls: `select()`, `pause()`, `connect()`, `accept()`, `read()` on terminals, sockets, pipes or files in `/proc`, `write()` on terminals, sockets, pipes or the line printer, `open()` on FIFOs, PTYs or serial lines, `ioctl()` on terminals, `fcntl()` with command `F_SETLKW`, `wait4()`, `syslog()`, any TCP or NFS operations.

For other operating systems you may have to include the system calls `creat()`, `close()`, `getmsg()`, `putmsg()`, `msgrcv()`, `msgsnd()`, `recv()`, `send()`, `wait()`, `waitpid()`, `wait3()`, `tcdrain()`, `sigpause()`, `semop()` to this list.

If a signal (that the program has installed a handler for) occurs during a system call, the handler is called.

When the handler returns (to the system call) it detects that it was interrupted, and immediately returns with −1 and `errno = EINTR`. The program is not expecting that to happen, so bottles out.

You may choose between two fixes.

(1) For every signal handler that you install, add `SA_RESTART` to the sigaction flags. For example, change

```
  signal (sig_nr, my_signal_handler);
```

to

```
  signal (sig_nr, my_signal_handler);
  { struct sigaction sa;
    sigaction (sig_nr, (struct sigaction *)0, 38;sa);
#ifdef SA_RESTART
    sa.sa_flags |= SA_RESTART;
#endif
#ifdef SA_INTERRUPT
    sa.sa_flags 38;= ~ SA_INTERRUPT;
#endif
    sigaction (sig_nr, 38;sa, (struct sigaction *)0);
  }
```

Note that while this applies to most system calls, you must still check for `EINTR` yourself on `read()`, `write()`, `ioctl()`, `select()`, `pause()` and `connect()`. See below.

(2) Check for `EINTR` explicitly, yourself:

Here are two examples for `read()` and `ioctl()`,

Original piece of code using `read()`

```
int result;
while (len > 0) {
  result = read(fd,buffer,len);
  if (result < 0) break;
  buffer += result; len -= result;
}
```

becomes

```
int result;
while (len > 0) {
  result = read(fd,buffer,len);
  if (result < 0) { if (errno != EINTR) break; }
  else { buffer += result; len -= result; }
}13;
```

and a piece of code using `ioctl()`

```
int result;
result = ioctl(fd,cmd,addr);
```

becomes

```
int result;
do { result = ioctl(fd,cmd,addr); }
while ((result == -1) 38;38; (errno == EINTR));
```

Note that in some versions of BSD Unix the default behaviour is to restart system calls. To get system calls interrupted you have to use the `SV_INTERRUPT` or `SA_INTERRUPT` flag.

4.3.9. Writable strings (program seg faults randomly)

GCC has an optimistic view of its users, believing that they intend string constants to be exactly that ––– constant. Thus, it stores them in the text (code) area of the program, where they can be paged in and out from the program's disk image (instead of taking up swapspace), and any attempt to rewrite them will cause a segmentation fault. This is a feature!

It may cause a problem for old programs that, for example, call `mktemp()` with a string constant as argument. `mktemp()` attempts to rewrite its argument in place.

To fix, either (a) compile with `–fwritable–strings`, to get gcc to put constants in data space, or (b) rewrite the offending parts to allocate a non–constant string and strcpy the data into it before calling.

4.3.10. Why does the `execl()` call fail?

Because you're calling it wrong. The first argument to `execl` is the program that you want to run. The second and subsequent arguments become the `argv` array of the program you're calling. Remember: `argv[0]` is traditionally set even when a program is run with `no' arguments. So, you should be writing

```
execl("/bin/ls","ls",NULL);
```

not just

```
execl("/bin/ls", NULL);
```

Executing the program with no arguments at all is construed as an invitation to print out its dynamic library dependencies, at least using a.out. ELF does things differently.

(If you want this library information, there are simpler interfaces; see the section on dynamic loading, or the manual page for `ldd`).

---

5. Debugging and Profiling

5.1. Preventative maintenance (lint)

There is no widely−used lint for Linux, as most people are satisfied with the warnings that gcc can generate. Probably the most useful is the `−Wall` switch −−− this stands for `Warnings, all' but probably has more mnemonic value if thought of as the thing you bang your head against.

There is a public domain lint available from . I don't know how good it is.

5.2. Debugging

5.2.1. How do I get debugging information into a program ?

You need to compile and link all its bits with the `−g` switch, and without the `−fomit−frame−pointer` switch. Actually, you don't need to recompile all of it, just the bits you're interested in debugging.

On a.out configurations the shared libraries are compiled with `−fomit−frame−pointer`, which gdb won't get on with. Giving the `−g` option when you link should imply static linking; this is why.

If the linker fails with a message about not finding libg.a, you don't have `/usr/lib/libg.a`, which is the special debugging−enabled C library. It may be supplied in the libc binary package, or (in newer C library versions) you may need to get the libc source code and build it yourself. You don't actually *need* it though; you can get enough information for most purposes simply by symlinking it to `/usr/lib/libc.a`

5.2.1.1. How do I get it out again?

A lot of GNU software comes set up to compile and link with `−g`, causing it to make very big (and often static) executables. This is not really such a hot idea.

If the program has an autoconf generated `configure` script, you can usually turn off debugging information by doing `./configure CFLAGS=` or `./configure CFLAGS=−O2`. Otherwise, check the Makefile. Of course, if you're using ELF, the program is dynamically linked regardless of the `−g` setting, so you can just `strip` it.

5.2.2. Available software

Most people use *gdb*, which you can get in source form from GNU archive sites, or as a binary from tsx−11 or sunsite. *xxgdb* is an X debugger based on this (i.e. you need gdb installed first). The source may be found at

Also, the *UPS* debugger has been ported by Rick Sladkey. It runs under X as well, but unlike xxgdb, it is not merely an X front end for a text based debugger. It has quite a number of nice features, and if you spend any time debugging stuff, you probably should check it out. The Linux precompiled version and patches for the stock UPS sources can be found in , and the original source at .

Another tool you might find useful for debugging is `*strace*', which displays the system calls that a process makes. It has a multiplicity of other uses too, including figuring out what pathnames were compiled into binaries that you don't have the source for, exacerbating race conditions in programs that you suspect contain them, and generally learning how things work. The latest version of strace (currently 3.0.8) can be found at .

5.2.3. Background (daemon) programs

Daemon programs typically execute `fork()` early, and terminate the parent. This makes for a short debugging session.

The simplest way to get around this is to set a breakpoint for `fork`, and when the program stops, force it to return 0.

```
(gdb) list
1       #include <stdio.h>
2
3       main()
4       {
5         if(fork()==0) printf("child\n");
6         else printf("parent\n");
7       }
(gdb) break fork
Breakpoint 1 at 0x80003b8
(gdb) run
Starting program: /home/dan/src/hello/./fork
Breakpoint 1 at 0x400177c4

Breakpoint 1, 0x400177c4 in fork ()
(gdb) return 0
Make selected stack frame return now? (y or n) y
#0  0x80004a8 in main ()
    at fork.c:5
5         if(fork()==0) printf("child\n");
(gdb) next
Single stepping until exit from function fork,
which has no line number information.
child
7       }
```

5.2.4. Core files

When Linux boots it is usually configured not to produce core files. If you like them, use your shell's builtin command to re−enable them: for C−shell compatibles (e.g. tcsh) this is

```
% limit core unlimited
```

while Bourne−like shells (sh, bash, zsh, pdksh) use

```
$ ulimit -c unlimited
```

If you want a bit more versatility in your core file naming (for example, if you're trying to conduct a post−mortem using a debugger that's buggy itself) you can make a simple mod to your kernel. Look for the code in `fs/binfmt_aout.c` and `fs/binfmt_elf.c` (in newer kernels, you'll have to grep around a little in older ones) that says

```
        memcpy(corefile,"core.",5);
#if 0
        memcpy(corefile+5,current->comm,sizeof(current->comm));
#else
        corefile[4] = '\0';
#endif
```

and change the `0`s to `1`s.

5.3. Profiling

Profiling is a way to examine which bits of a program are called most often or run for longest. It is a good way to optimize code and look at where time is being wasted. You must compile all object files that you require timing information for with `−p`, and to make sense of the output file you will also need `gprof` (from the binutils package). See the `gprof` manual page for details.

---

6. Linking

Between the two incompatible binary formats, the static vs shared library distinction, and the overloading of the verb `link' to mean both `what happens after compilation' and `what happens when a compiled program is invoked' (and, actually, the overloading of the word `load' in a comparable but opposite sense), this section is complicated. Little of it is much more complicated than that sentence, though, so don't worry too much about it.

To alleviate the confusion somewhat, we refer to what happens at runtime as `dynamic loading' and cover it in the next section. You will also see it described as `dynamic linking', but not here. This section, then, is exclusively concerned with the kind of linking that happens at the end of a compilation.

6.1. Shared vs static libraries

The last stage of building a program is to `link' it; to join all the pieces of it together and see what is missing. Obviously there are some things that many programs will want to do ––– open files, for example, and the pieces that do these things are provided for you in the form of libraries. On the average Linux system these can be found in `/lib` and `/usr/lib/`, among other places.

When using a static library, the linker finds the bits that the program modules need, and physically copies them into the executable output file that it generates. For shared libraries, it doesn't ––– instead it leaves a note in the output saying `when this program is run, it will first have to load this library'. Obviously shared libraries tend to make for smaller executables; they also use less memory and mean that less disk space is used. The default behaviour of Linux is to link shared if it can find the shared libraries, static otherwise. If you're getting static binaries when you want shared, check that the shared library files (`*.sa` for a.out, `*.so` for ELF) are where they should be, and are readable.

On Linux, static libraries have names like `libname.a`, while shared libraries are called `libname.so.x.y.z` where `x.y.z` is some form of version number. Shared libraries often also have links pointing to them, which are important, and (on a.out configurations) associated `.sa` files. The standard libraries come in both shared and static formats.

You can find out what shared libraries a program requires by using `ldd` (List Dynamic Dependencies)

```
$ ldd /usr/bin/lynx
        libncurses.so.1 => /usr/lib/libncurses.so.1.9.6
        libc.so.5 => /lib/libc.so.5.2.18
```

This shows that on my system the WWW browser `lynx' depends on the presence of `libc.so.5` (the C library) and `libncurses.so.1` (used for terminal control). If a program has no dependencies, `ldd` will say `` `statically linked' `` or `` `statically linked (ELF)' ``.

6.2. Interrogating libraries (`which library is `sin()` in?')

`nm` *libraryname* should list all the symbols that *libraryname* has references to. It works on both static and shared libraries. Suppose that you want to know where `tcgetattr()` is defined: you might do

```
$ nm libncurses.so.1 |grep tcget
         U tcgetattr
```

The `U` stands for `undefined' --- it shows that the ncurses library uses but does not define it. You could also do

```
$ nm libc.so.5 | grep tcget
00010fe8 T __tcgetattr
00010fe8 W tcgetattr
00068718 T tcgetpgrp
```

The ``W`' stands for `weak', which means that the symbol is defined, but in such a way that it can be overridden by another definition in a different library. A straightforward `normal' definition (such as the one for `tcgetpgrp`) is marked by a ``T`'

The short answer to the question in the title, by the way, is `libm.(so|a)`. All the functions defined in `<math.h>` are kept in the maths library; thus you need to link with `-lm` when using any of them.

6.3. Finding files

`ld: Output file requires shared library `libfoo.so.1``

The file search strategy of ld and friends varies according to version, but the only default you can reasonably assume is `/usr/lib`. If you want libraries elsewhere to be searched, specify their directories with the `-L` option to gcc or ld.

If that doesn't help, check that you have the right file in that place. For a.out, linking with `-lfoo` makes ld look for `libfoo.sa` (shared stubs), and if unsuccessful then for `libfoo.a` (static). For ELF, it looks for `libfoo.so` then `libfoo.a`. `libfoo.so` is usually a symbolic link to `libfoo.so.x`.

6.4. Building your own libraries

6.4.1. Version control

As any other program, libraries tend to have bugs which get fixed over time. They also may introduce new features, change the effect of existing ones, or remove old ones. This could be a problem for programs using them; what if it was depending on that old feature?

So, we introduce library versioning. We categorise the changes that might be made to a library as `minor' or `major', and we rule that a `minor' change is not allowed to break old programs that are using the library. You can tell the version of a library by looking at its filename (actually, this is, strictly speaking, a lie for ELF; keep reading to find out why) : `libfoo.so.1.2` has major version 1, minor version 2. The minor version number can be more or less anything −−− libc puts a `patchlevel' in it, giving library names like `libc.so.5.2.18`, and it's also reasonable to put letters, underscores, or more or less any printable ASCII in it.

One of the major differences between ELF and a.out format is in building shared libraries. We look at ELF first, because it's simpler.

6.4.2. ELF? What is it then, anyway?

ELF (Executable and Linking Format) is a binary format originally developed by USL (UNIX System Laboratories) and currently used in Solaris and System V Release 4. Because of its increased flexibility over the older a.out format that Linux was using, the GCC and C library developers decided last year to move to using ELF as the Linux standard binary format also.

6.4.2.1. Come again?

This section is from the document '/news−archives/comp.sys.sun.misc'.

"ELF ("Executable Linking Format) is the "new, improved" object file format introduced in SVR4. ELF is much more powerful than straight COFF, in that it *is* user−extensible. ELF views an object−file as an arbitarily long list of sections (rather than an array of fixed size entities), these sections, unlike in COFF, do not HAVE to be in a certain place and do not HAVE to come in any specific order etc. Users can add new sections to object−files if they wish to capture new data. ELF also has a far more powerful debugging format called DWARF (Debugging With Attribute Record Format) − not currently fully supported on linux (but work is underway). A linked list of DWARF DIEs (or Debugging Information Entries) forms the .debug section in ELF. Instead of being a collection of small, fixed−size information records, DWARF DIEs each contain an arbitrarily long list of complex attributes and are written out as a scope−based tree of program data. DIEs can capture a large amount of information that the COFF .debug section simply couldn't (like C++ inheritance graphs etc.)." "ELF files are accessed via the SVR4 (Solaris 2.0 ?) ELF access library, which provides an easy and fast interface to the more gory parts of ELF. One of the major boons in using the ELF access library is that you will never need to look at an ELF file qua. UNIX file, it is accessed as an Elf *, after an elf_open() call and from then on, you perform elf_foobar() calls on its components instead of messing about with its actual on−disk image (something many COFFers did with impunity). "

The case for/against ELF, and the necessary contortions to upgrade an a.out system to support it, are covered in the ELF−HOWTO and I don't propose to cut/paste them here. The HOWTO should be available in the same place as you found this one.

6.4.2.2. ELF shared libraries

To build `libfoo.so` as a shared library, the basic steps look like this:

```
$ gcc -fPIC -c *.c
$ gcc -shared -Wl,-soname,libfoo.so.1 -o libfoo.so.1.0 *.o
$ ln -s libfoo.so.1.0 libfoo.so.1
$ ln -s libfoo.so.1 libfoo.so
$ LD_LIBRARY_PATH=`pwd`:$LD_LIBRARY_PATH ; export LD_LIBRARY_PATH
```

This will generate a shared library called `libfoo.so.1.0`, and the appropriate links for ld (`libfoo.so`) and the dynamic loader (`libfoo.so.1`) to find it. To test, we add the current directory to `LD_LIBRARY_PATH`.

When you're happpy that the library works, you'll have to move it to, say, `/usr/local/lib`, and recreate the appropriate links. The link from `libfoo.so.1` to `libfoo.so.1.0` is kept up to date by `ldconfig`, which on most systems is run as part of the boot process. The `libfoo.so` link must be updated manually. If you are scrupulous about upgrading all the parts of a library (e.g. the header files) at the same time, the simplest thing to do is make `libfoo.so -> libfoo.so.1`, so that ldconfig will keep both links current for you. If you *aren't*, you're setting yourself up to have *all kinds of weird things* happen at a later date. Don't say you weren't warned.

```
$ su
# cp libfoo.so.1.0 /usr/local/lib
# /sbin/ldconfig
# ( cd /usr/local/lib ; ln -s libfoo.so.1 libfoo.so )
```

6.4.2.3. Version numbering, sonames and symlinks

Each library has a *soname*. When the linker finds one of these in a library it is searching, it embeds the soname into the binary instead of the actual filename it is looking at. At runtime, the dynamic loader will then search for a file with the name of the soname, not the library filename. Thus a library called `libfoo.so` could have a soname `libbar.so`, and all programs linked to it would look for `libbar.so` instead when they started.

This sounds like a pointless feature, but it is key to understanding how multiple versions of the same library can coexist on a system. The de facto naming standard for libraries in Linux is to call the library, say, `libfoo.so.1.2`, and give it a soname of `libfoo.so.1`. If it's added to a `standard' library directory (e.g. `/usr/lib`), `ldconfig` will create a symlink `libfoo.so.1 -> libfoo.so.1.2` so that the appropriate image is found at runtime. You also need a link `libfoo.so -> libfoo.so.1` so that ld will find the right soname to use at link time.

So, when you fix bugs in the library, or add new functions (any changes that won't adversely affect existing programs), you rebuild it, keeping the soname as it was, and changing the filename. When you make changes to the library that would break existing binaries, you simply increment the number in the soname −−− in this case, call the new version `libfoo.so.2.0`, and give it a soname of `libfoo.so.2`. Now switch the `libfoo.so` link to point to the new version and all's well with the world again.

Note that you don't *have* to name libraries this way, but it's a good convention. ELF gives you the flexibility to name libraries in ways that will confuse the pants off people, but that doesn't mean you have to use it.

Executive summary: supposing that you observe the tradition that major upgrades may break compatibility, minor upgrades may not, then link with

```
gcc −shared −Wl,−soname,libfoo.so.major −o libfoo.so.major.minor
```

and everything will be all right.

6.4.3. a.out. Ye olde traditional format

The ease of building shared libraries is a major reason for upgrading to ELF. That said, it's still possible in a.out. Get and read the 20 page document that you will find after unpacking it. I hate to be so transparently partisan, but it should be clear from context that I never bothered myself :−)

6.4.3.1. ZMAGIC vs QMAGIC

QMAGIC is an executable format just like the old a.out (also known as ZMAGIC) binaries, but which leaves the first page unmapped. This allows for easier NULL dereference trapping as no mapping exists in the range 0−4096. As a side effect your binaries are nominally smaller as well (by about 1K).

Obsolescent linkers support ZMAGIC only, semi−obsolescent support both formats, and current versions support QMAGIC only. This doesn't actually matter, though, as the kernel can still run both formats.

Your `file' command should be able to identify whether a program is QMAGIC.

6.4.3.2. File Placement

An a.out (DLL) shared library consists of two real files and a symlink. For the `foo' library used throughout this document as an example, these files would be `libfoo.sa` and `libfoo.so.1.2`; the symlink would be `libfoo.so.1` and would point at the latter of the files. What are these for?

At compile time, `ld` looks for `libfoo.sa`. This is the `stub' file for the library, and contains all exported data and pointers to the functions required for run time linking.

At run time, the dynamic loader looks for `libfoo.so.1`. This is a symlink rather than a real file so that libraries can be updated with newer, bugfixed versions without crashing any application that was using the library at the time. After the new version −−− say, `libfoo.so.1.3` −−− is completely there, running ldconfig will switch the link to point to it in one atomic operation, leaving any program which had the old version still perfectly happy.

DLL libraries (I know that's a tautology −−− so sue me) often appear bigger than their static counterparts. They reserve space for future expansion in the form of `holes' which can be made to take no disk space. A simple `cp` call or using the program `makehole` will achieve this. You can also strip them after building, as the addresses are in fixed locations. *Do not attempt to strip ELF libraries.*

6.4.3.3. ``libc−lite''?

A libc−lite is a light−weight version of the libc library built such that it will fit on a floppy and suffice for all of the most menial of UNIX tasks. It does *not* include curses, dbm, termcap etc code. If your `/lib/libc.so.4` is linked to a lite lib, you are advised to replace it with a full version.

6.4.4. Linking: common problems

Send me your linking problems! I probably won't do anything about them, but I will write them up if I get enough ...

Programs link static when you wanted them shared

> Check that you have the right links for `ld` to find each shared library. For ELF this means a `libfoo.so` symlink to the image, for a.out a `libfoo.sa` file. A lot of people had this problem after moving from ELF binutils 2.5 to 2.6 −−− the earlier version searched more `intelligently' for shared libraries, so they hadn't created all the links. The intelligent behaviour was removed for compatibility with other architectures, and because quite often it got its assumptions wrong and caused more trouble than it solved.

The DLL tool `mkimage' fails to find libgcc, or

> As of `libc.so.4.5.x` and above, libgcc is no longer shared. Hence you must replace occurrences of `` `-lgcc' `` on the offending line with `` `gcc -print-libgcc-file-name` `` (complete with the backquotes).
>
> Also, delete all `/usr/lib/libgcc*` files. This is important.

`__NEEDS_SHRLIB_libc_4 multiply defined` messages

> are another consequence of the same problem.

``Assertion failure'' message when rebuilding a DLL ?

> This cryptic message most probably means that one of your jump table slots has overflowed because too little space has been reserved in the original `jump.vars` file. You can locate the culprit(s) by running the `` `getsize' `` command provided in the tools−2.17.tar.gz package. Probably the only

solution, though, is to bump the major version number of the library, forcing it to be backward incompatible.

`ld: output file needs shared library libc.so.4`

This usually happens when you are linking with libraries other than libc (e.g. X libraries), and use the `-g` switch on the link line without also using `-static`.

The `.sa` stubs for the shared libraries usually have an undefined symbol `_NEEDS_SHRLIB_libc_4` which gets resolved from the `libc.sa` stub. However with `-g` you end up linking with `libg.a` or `libc.a` and thus this symbol never gets resolved, leading to the above error message.

In conclusion, add `-static` when compiling with the `-g` flag, or don't link with `-g`. Quite often you can get enough debugging information by compiling the individual files with `-g`, and linking *without* it.

---

7. Dynamic Loading

This section is a tad short right now; it will be expanded over time as I gut the ELF howto

7.1. Concepts

Linux has shared libraries, as you will by now be sick of hearing if you read the whole of the last section at a sitting. Some of the matching−names−to−places work which was traditionally done at link time must be deferred to load time.

7.2. Error messages

Send me your link errors! I won't do anything about them, but I might write them up ...

`can't load library: /lib/libxxx.so, Incompatible version`

(a.out only) This means that you don't have the correct major version of the xxx library. No, you can't just make a symlink to another version that you do have; if you are lucky this will cause your program to segfault. Get the new version. A similar situation with ELF will result in a message like

```
ftp: can't load library 'libreadline.so.2'
```

`warning using incompatible library version xxx`

(a.out only) You have an older minor version of the library than the person who compiled the program used. The program will still run. Probably. An upgrade wouldn't hurt, though.

7.3. Controlling the operation of the dynamic loader

There are a range of environment variables that the dynamic loader will respond to. Most of these are more use to `ldd` than they are to the average user, and can most conveniently be set by running ldd with various switches. They include

- `LD_BIND_NOW` −−− normally, functions are not `looked up' in libraries until they are called. Setting this flag causes all the lookups to happen when the library is loaded, giving a slower startup time. It's useful when you want to test a program to make sure that everything is linked.
- `LD_PRELOAD` can be set to a file containing `overriding' function definitions. For example, if you were testing memory allocation strategies, and wanted to replace `malloc', you could write your replacement routine, compile it into `malloc.o` and then

```
$ LD_PRELOAD=malloc.o; export LD_PRELOAD
$ some_test_program
```

`LD_ELF_PRELOAD` and `LD_AOUT_PRELOAD` are similar, but only apply to the appropriate type of binary. If `LD_`*something*`_PRELOAD` and `LD_PRELOAD` are set, the more specific one is used.

- `LD_LIBRARY_PATH` is a colon−separated list of directories in which to look for shared libraries. It does *not* affect ld; it only has effect at runtime. Also, it is disabled for programs that run setuid or setgid. Again, `LD_ELF_LIBRARY_PATH` and `LD_AOUT_LIBRARY_PATH` can also be used to direct the search differently for different flavours of binary. `LD_LIBRARY_PATH` shouldn't be necessary in normal operation; add the directories to `/etc/ld.so.conf/` and rerun ldconfig instead.
- `LD_NOWARN` applies to a.out only. When set (e.g. with `LD_NOWARN=true; export LD_NOWARN`) it stops the loader from issuing non−fatal warnings (such as minor version incompatibility messages).
- `LD_WARN` applies to ELF only. When set, it turns the usually fatal ``Can't find library'' messages into warnings. It's not much use in normal operation, but important for ldd.
- `LD_TRACE_LOADED_OBJECTS` applies to ELF only, and causes programs to think they're being run under `ldd`:

```
$ LD_TRACE_LOADED_OBJECTS=true /usr/bin/lynx
        libncurses.so.1 => /usr/lib/libncurses.so.1.9.6
        libc.so.5 => /lib/libc.so.5.2.18
```

7.4. Writing programs with dynamic loading

This is very close to the way that Solaris 2.x dynamic loading support works, if you're familiar with that. It is covered extensively in H J Lu's ELF programming document, and the `dlopen(3)` manual page, which can be found in the ld.so package. Here's a nice simple example though: link it with `-ldl`

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

main()
{
  void *libc;
  void (*printf_call)();

  if(libc=dlopen("/lib/libc.so.5",RTLD_LAZY))
  {
    printf_call=dlsym(libc,"printf");
    (*printf_call)("hello, world\n");
  }

}
```

---

8. Contacting the developers

8.1. Bug reports

Start by *narrowing the problem down*. Is it specific to Linux, or does it happen with gcc on other systems? Is it specific to the kernel version? Library version? Does it go away if you link static? Can you trim the program down to something *short* that demonstrates the bug?

Having done that, you'll know what program(s) the bug is in. For GCC, the bug reporting procedure is explained in the info file. For ld.so or the C or maths libraries, send mail to `linux-gcc@vger.rutgers.edu`. If possible, include a short and self–contained program that exhibits the bug, and a description both of what you want it to do, and what it actually does.

8.2. Helping with development

If you want to help with the development effort for GCC or the C library, the first thing to do is join the `linux-gcc@vger.rutgers.edu` mailing list. If you just want to see what the discussion is about, there are list archives at . The second and subsequent things depend on what you want to do!

9. The Remains

9.1. The Credits

" Only presidents, editors, and people with tapeworms have the right to use the editorial ``we"." (Mark Twain)

This HOWTO is based very closely on Mitchum DSouza's GCC–FAQ; most of the information (not to mention a reasonable amount of the text) in it comes directly from that document. Instances of the first person pronoun in this HOWTO could refer to either of us; generally the ones that say ``I have not tested this; don't blame me if it toasts your hard disk/system/spouse" apply to both of us.

Contributors to this document have included (in ASCII ordering by first name) Andrew Tefft, Axel Boldt, Bill Metzenthen, Bruce Evans, Bruno Haible, Daniel Barlow, Daniel Quinlan, David Engel, Dirk Hohndel, Eric Youngdale, Fergus Henderson, H.J. Lu, Jens Schweikhardt, Kai Petzke, Michael Meissner, Mitchum DSouza, Olaf Flebbe, Paul Gortmaker, Rik Faith, Steven S. Dick, Tuomas J Lukka, and of course Linus Torvalds, without whom the whole exercise would have been pointless, let alone impossible :–)

Please do not feel offended if your name has not appeared here and you have contributed to this document (either as HOWTO or as FAQ). Email me and I will rectify it.

9.2. Translations

- French, Eric Dumas

  `<dumas@freenix.fr>` http://www.freenix.fr/unix/linux/HOWTO/GCC–HOWTO.html

- Italian, Andrea Girotto

  `<andrea.girotto@usa.net>`

  http://www.pluto.linux.it/ildp/HOWTO/GCC–HOWTO.html

- Japanese,

  `<nakano@apm.seikei.ac.jp>`

9.3. Feedback

is welcomed. Mail me at daniel.barlow@linux.org. My PGP public key (ID 5F263625) is available from my web pages, if you feel the need to be secretive about things.

9.4. Legalese