ELF: From The Programmer's Perspective

Hongjiu Lu
`hjl@nynexst.com`
NYNEX Science & Technology, Inc.
500 Westchester Avenue
White Plains, NY 10604, USA

May 17, 1995

Abstract

In this paper, we discuss the new ELF binary format for Linux specifically from the view of the programmer. We introduce some techniques which can be used with ELF to control the execution of a program at run time. We show how to use dynamic linking and dynamic loading under ELF. We also demonstrate how to use the GNU `C/C++` compiler and binary utilities to create shared `C/C++` libraries under Linux.

1 Introduction

The Executable and Linking Format (ELF) is a binary format originally developed and published by UNIX System Laboratories (USL). It is the default binary format for the executable files used by SVR4 and Solaris 2.x. ELF is more powerful and flexible than the a.out and COFF binary formats. Combined with appropriate tools, programmers can use ELF to control the flow of execution at run time.

2 ELF Types

There are three main types for ELF files.

- An *executable* file contains code and data suitable for execution. It specifies the memory layout of the process.
- A *relocatable* file contains code and data suitable for linking with other *relocatable* and *shared object* files.

- A *shared object* file (a.k.a. shared library) contains code and data suitable for the *link editor* **ld** at link time and the *dynamic linker* at run time. The dynamic linker may be called **ld.so.1**, **libc.so.1** or **ld-linux.so.1**, depending on the implementation.

The most useful part of ELF lies in its section structure. With the right tools and techniques, programmers can manipulate the execution of executables with great flexibility.

3 The .init and .fini Sections

On an ELF system, a program consists of one executable file and zero or more shared object files. To execute such a program, the system uses those files to create a process image in memory. A process image has *segments* which contain executable instructions, data and so on. For an ELF file to be loaded into memory, it has to have a program header which is an array of structures which describe segments and other information which the system needs to prepare the program for execution.

A segment consists of *sections*, which is the most important aspect of ELF from the programmer's point of view.

Each executable or shared object file generally contains a section table, which is an array of structure describing the sections inside the ELF object file. There are several special sections defined by the ELF documentations which hold program and control information. The following ones are very useful to programmers.

.fini
: This section holds executable instructions that contribute to the process termination code. That is, when a program exits normally, the system arranges to execute the code in this section.

.init
: This section holds executable instructions that contribute to the process initialization code. That is, when a program starts to run the system arranges to execute the code in this section before the main program entry point (called *main* in C programs).

The **.init** and **.fini** sections have a special purpose. If a function is placed in the **.init** section, the system will execute it before the *main* function. Also the functions placed in the **.fini** section will be executed by the system after the *main* function returns. This feature is utilized by compilers to implement global constructors and destructors in `C++`.

When an ELF executable is executed, the system will load in all the shared object files before transferring control to the executable. With the properly constructed **.init** and **.fini** sections, constructors and destructors will be called in the right order.

3.1 Global Constructors and Destructors in `C++`

Global constructors and destructors in `C++` have to be handled very carefully to meet the language specification. Constructors have to be called before the *main* function. Destructors have to be executed after it returns. Under ELF, this can be treated gracefully by the compiler. For example, the `GNU C/C++` compiler, *gcc*, provides two auxiliary start up files called *crtbegin.o* and *crtend.o*, in addition to two normal auxiliary files *crti.o* and *crtn.o*. Together with the **.ctors** and **.dtors** sections described below, the `C++` global constructors and destructors can be executed in the proper order with minimal run-time overhead.

.ctors
: This section holds an array of the global constructor function pointers of a program.

.dtors
: This section holds an array of the global destructor function pointers of a program.

crtbegin.o
: There are four sections:

- The **.ctors** section. It has a local symbol, `__CTOR_LIST__`, which is the head of the global constructor function pointer array. This array in *crtbegin.o* only has one dummy element.
- The **.dtors** section. It has a local symbol, `__DTOR_LIST__`, which is the head of the global destructor function pointer array. This array in *crtbegin.o* only has only one dummy element.
- The **.text** section. It contains only one function, `__do_global_dtors_aux`, which goes through `__DTOR_LIST__` from the head and calls each destructor function on the list.
- The **.fini** section. It contains only a call to `__do_global_dtors_aux`. Please remember it has just a function call without return since the **.fini** section in **crtbegin.o** is part of the body of a function.

crtend.o
: There are also four sections:

- The **.ctors** section. It has a local symbol, `__CTOR_END__`, which is the label for the tail of the global constructor function pointer array.
- The **.dtors** section. It has a local symbol, `__DTOR_END__`, which is the label for the tail of the global destructor function pointer array.
- The **.text** section. It contains only one function, `__do_global_ctors_aux`, which goes through `__CTOR_LIST__` from the tail and calls each constructor function on the list.
- The **.init** section. It contains only a function call to `__do_global_ctors_aux`. Please remember it has just a function call without return since the **.init** section in **crtend.o** is part of the body of a function.

crti.o
: It has only a function label `_init` in the **.init** section and a function label `_fini` in the **.fini** section.

crtn.o
It has only a return instruction each in the **.init** and **.fini** sections.

At compile time while generating the relocatable files, *gcc* puts each global constructor on `__CTOR_LIST__` by putting a pointer to the constructor function in the **.ctors** section. It also puts each global destructor on `__DTOR_LIST__` by putting a pointer to the destructor function in the **.dtors** section.

At link time, the *gcc* driver places **crtbegin.o** immediately before all the relocatable files and **crtend.o** immediately after all the relocatable files. In addition, **crti.o** was placed before **crtbegin.o** and **crtn.o** was placed after **crtend.o**.

While generating the executable file, the link editor, **ld**, concatenates the **.ctors** sections and the **.dtors** sections from all the relocatable files to form `__CTOR_LIST__` and `__DTOR_LIST__`, respectively. The **.init** sections from all the relocatable files form the `_init` function and the **.fini** sections form the `_fini` function.

At run time, the system will execute the `_init` function before the *main* function and execute the `_fini` function after the *main* function returns.

4 Dynamic Linking and Dynamic Loading in ELF

4.1 Dynamic Linking

When one uses a C compiler under a Unix system to generate an executable from the C source code, the C compiler driver will usually invoke a C preprocessor, compiler, assembler and link editor in that order to translate the C language code into the executable file.

- The C compiler driver will first pass the C source code into a C preprocessor which outputs the pure C language code with the processed macros and directives,
- The C compiler translates the resultant C language code into machine-dependent assembly language code.
- The assembler translates the resultant assembly language code into the machine instructions of the target machine. The resultant machine instructions are stored in an object file in a specific binary format. In our case, the object files use the ELF binary format.
- In the last stage, the link editor links all the object files together with the start up codes and library functions which are referenced in the program. There are two kinds of libraries one can use:
  - A *static library* is a collection of object files which contain library routines and data. It is built in such a way that the link editor will incorporate a copy of only those object files that hold the functions and data referenced in the program into the executable at link time.
  - A *shared library* is a shared object file that contains functions and data. It is built in such a way that the link editor will only store in the executable the name of the shared

library and information about the symbols referenced by the executable. At run time the dynamic linker, a.k.a. the *program interpreter* in ELF, will map the shared library into the virtual address space of the process image of the executable and resolve by name the symbols in the shared library used by the executable. That is process is also called *dynamic linking*.

There is nothing special which needs to be done by the programmer to take advantage of shared libraries with dynamic linking. Everything is transparent to programmers as well as to users.

4.2 Dynamic Loading

Dynamic loading is the process in which one can attach a shared library to the address space of the process during execution, look up the address of a function in the library, call that function and then detach the shared library when it is no longer needed. It is implemented as an interface to the services of the dynamic linker.

Under ELF, the programming interface is usually defined in `<dlfcn.h>`. These are:

```
void *dlopen (const char * filename, int flag);
const char * dlerror (void);
const void * dlsym (void  handle*, const char * symbol);
int dlclose (void * handle);
```

These functions are contained in **libdl.so**. Here is an example of how dynamic loading works.

We have a main program which loads in the shared library dynamically at the run time. One can specify which shared library to use and which function to call. One can also access the data in the shared library.

```
# cat dltest.c

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <dlfcn.h>
#include <ctype.h>

typedef void (*func_t) (const char *);

void
dltest (const char *s)
{
  printf ("From dltest: ");
  for (; *s; s++)
  {
    putchar (toupper (*s));
```

```
  }
  putchar ('\n');
}

main (int argc, char **argv)
{
  void *handle;
  func_t fptr;
  char *libname = "libfoo.so";
  char **name = NULL;
  char *funcname = "foo";
  char *param = "Dynamic Loading Test";
  int ch;
  int mode = RTLD_LAZY;

  while ((ch = getopt (argc, argv, "a:b:f:l:")) != EOF)
  {
    switch (ch)
    {
    case 'a': /* argument. */
      param = optarg;
      break;

    case 'b': /* how to bind. */
      switch (*optarg)
      {
      case 'l': /* lazy */
        mode = RTLD_LAZY;
        break;

      case 'n': /* now */
        mode = RTLD_NOW;
        break;
      }
      break;

    case 'l': /* which shared library. */
      libname = optarg;
      break;

    case 'f': /* which function? */
      funcname = optarg;
    }
  }
```

```
  handle = dlopen (libname, mode);
  if (handle == NULL)
  {
    fprintf (stderr, "%s: dlopen: `%s'\n", libname, dlerror ());
    exit (1);
  }

  fptr = (func_t) dlsym (handle, funcname);
  if (fptr == NULL)
  {
    fprintf (stderr, "%s: dlsym: `%s'\n", funcname, dlerror ());
    exit (1);
  }

  name = (char **) dlsym (handle, "libname");
  if (name == NULL)
  {
    fprintf (stderr, "%s: dlsym: `libname'\n", dlerror ());
    exit (1);
  }

  printf ("Call `%s' in `%s':\n", funcname, *name);

  /* call that function with `param' */
  (*fptr) (param);

  dlclose (handle);
  return 0;
}
```

There are two shared libraries here, **libfoo.so** and **libbar.so**. Each has the same global string variable, *libname*, but their own functions, *foo* and *bar*, respectively. They are both available to the program via *dlsym*.

```
# cat libbar.c

#include <stdio.h>

extern void dltest (const char *);
const char *const libname = "libbar.so";

void bar (const char *s)
{
  dltest ("called from libbar.");
```

```
  printf("libbar: %s\n", s);
}

# cat libfoo.c

#include <stdio.h>

extern void dltest (const char *s);
const char *const libname = "libfoo.so";

void
foo (const char *s)
{
  const char *saved = s;

  dltest ("called from libfoo");
  printf("libfoo: ");
  for (; *s; s++);
  for (s--; s >= saved; s--)
  {
    putchar (*s);
  }
  putchar ('\n');
}
```

Makefile is used to build the shared libraries and the main program since **libbar.so** and **libfoo.so** call the function *dltest* in the main program.

```
# cat Makefile

CC=gcc
LDFLAGS=-rdynamic
SHLDFLAGS=

all: dltest

libfoo.o: libfoo.c
        $(CC) -c -fPIC $<

libfoo.so: libfoo.o
        $(CC) $(SHLDFLAGS) -shared -o $@ $^

libbar.o: libbar.c
        $(CC) -c -fPIC $<
```

```
libbar.so: libbar.o
        $(CC) $(SHLDFLAGS) -shared -o $@ $^

dltest: dltest.o libbar.so libfoo.so
        $(CC) $(LDFLAGS) -o $@ dltest.o -ldl

clean:
        $(RM) *.o *.so dltest
```

Here is the flow of the program:

```
# export ELF_LD_LIBRARY_PATH=.
# dltest
Call ‘foo’ in ‘libfoo.so’:
From dltest: CALLED FROM LIBFOO
libfoo: tseT gnidaoL cimanyD
# dltest -f bar
bar: dlsym: ‘Unable to resolve symbol’
dltest -f bar -l libbare.so
Call ‘bar’ in ‘libbar.so’:
From dltest: CALLED FROM LIBBAR.
libbar: Dynamic Loading Test
```

dlopen is the first function to call in the dynamic loading process which makes a shared library, *libfoo.so*, available to a running process. *dlopen* returns a *handle* which should be used in subsequent calls to the *dlsym* and *dlclose* functions for operation on *libfoo.so*. Using **NULL** for the filename argument of *dlopen* has a special meaning — it will make the *exported* symbols in the executable and the currently loaded shared libraries available via *dlsym*.

After a shared library has been loaded into the address space of the running process, *dlsym* may be used to obtain the address of an exported symbol in that shared library. One can then access the function or data through the address returned from *dlsym*.

dlclose can be called to detach the shared library loaded in via the *dlopen* call when the shared library is no longer needed. The shared library will not be removed from the address space of the calling process if the shared library has been loaded in during startup time or through other *dlopen* calls.

dlclose returns 0 if the operation on the shared library is successful. Both *dlopen* and *dlsym* will return **NULL** in case of any error condition. *dlerror* can be called in that case to obtain diagnostic information. More detailed programming information on *dlopen/dlsym/dlclose/dlerror* can be found in [1].

5 ELF Support in GCC, GNU Link Editor and Linux

Thanks to Eric Youngdale (`eric@aib.com`), Ian Lance Taylor (`ian@cygnus.com`) and many people working on ELF support in *gcc*, GNU binary utilities and Linux, we can now use gcc and GNU binary utilities to build and run ELF executables as well as build shared libraries.

5.1 The Shared C Library

Building a shared library has never been easier under ELF. But we need ELF support in the compiler, assembler and link editor. First we need to generate *position-independent code*. Under gcc, that is done by adding `-fPIC` to the compiler options:

```
# gcc -fPIC -O -c libbar.c
```

We now have compiled **libbar.o** suitable for building the shared library. Next we can generate the shared library by

```
# gcc -shared -o libbar.so libbar.o
```

We now have built the shared library **libbar.so** which can be used by the link editor and dynamic linker. One can add as many relocatable object files to a shared library as long as they are compiled with `-fPIC`. To link **baz.o** with the shared library you can just do

```
# gcc -O -c baz.c
# gcc -o baz  baz.o -L. -lbar
```

After installing **libbar.so** in the right place where the dynamic linker can find it, running **baz** will make **libbar.so** map into the address space of the process image of **baz**. One copy of **libbar.so** in memory will be shared by all executables which are linked with it or which load it dynamically at run time.

5.2 The Shared C++ Library

The main obstacle in the shared C++ library is how to treat global constructors and destructors. Under SunOS, building and using a shared C library is almost as easy as under ELF. But one cannot build a shared C++ library under SunOS because of the special requirements of the constructors and destructors. The **.init** and **.fini** sections in ELF provide a perfect solution for the problem.

When building a shared C++ library, we use two special versions of **crtbegin.o** and **crtend.o** which have been compiled with `-fPIC`. To the link editor, building the shared C++ library is almost exactly the same as building a normal executable. the global constructors and destructors are handled by the same **.init** and **.fini** sections we have discussed in Section 3.1.

When a shared library is mapped into the address space, the dynamic linker will execute the `_init` function before transferring the control to the program and will arrange for the `_fini` function to be executed when the shared library is no longer needed.

The link option `-shared` for the gcc driver places the necessary auxiliary files in the right order and tells the link editor to generate a shared library. The `-v` option will show what files and options are passed to the link editor.

```
# gcc -v -shared -o libbar.so libbar.o
Reading specs from
/usr/lib/gcc-lib/i486-linux/2.6.4-950305/specs
gcc driver version 2.6.4 snapshot 950305 executing gcc version 2.6.4
 ld -m elf_i386 -shared -o libbar.so /usr/lib/crti.o
/usr/lib/crtbeginS.o -L/usr/lib/gcc-lib/i486-linux/2.6.4-950305
-L/usr/i486-linux/lib libbar.o /usr/lib/crtendS.o /usr/lib/crtn.o
```

crtbeginS.o and **crtendS.o** are the special version compiled with `-fPIC`. It is very important that a shared library is always built with `gcc -shared` since those auxiliary files also serve other purposes. We will discuss this in Section 5.3.

5.3 Extended GCC Features

There are many extended features in gcc. Some of them are especially useful in ELF. One of them is `__attribute__`. One can place a function on the `__CTOR_LIST__` or `__DTOR_LIST__` with `__attribute__`. For example,

```
static void foo () __attribute__ ((constructor));
static void bar () __attribute__ ((destructor));

static void
foo ()
{
}

static void
bar ()
{
}
```

The `__attribute__ ((constructor))` causes *foo* to be called automatically before execution enters *main*. Similarly, the `__attribute__ ((destructor))` causes *bar* to be executed automatically after *main* returns or when *exit* has been called. Both *foo* and *bar* must take no arguments and must be static void functions. Under ELF, this feature works in both normal executables and shared libraries.

5.3.1 The Initialization Functions in the C library

Another gcc feature is `__attribute__` ((section ("*sectionname*"))). With this, one can put a function or a data structure in any section.

```
static void
foo (int argc, char **argv, char **envp)
        __attribute__ ((section ("_libc_foo")));


static void
foo (int argc, char **argv, char **envp)
{
}

static void
bar (int argc, char **argv, char **envp)
{
}

static void * __libc_subinit_bar__
        __attribute__ ((section ("_libc_subinit"))) = &(bar);
```

Here we place *foo* in the `_libc_foo` section and `__libc_subinit_bar__` in the `_libc_subinit` section. In the Linux C library `_libc_subinit` is a special section which contains an array of function pointers with prototype

```
void (*) (int argc, char **argv, char **envp);
```

where *argc*, *argv* and *envp* has the same meaning as in *main*. Functions in this section will be called before entering the *main* function. They are used in the Linux C library to initialize a few global variables.

5.4 Using ELF with `GCC` and `GNU ld`

There are several command line options for gcc and GNU ld which are especially useful to ELF. The `-shared` option tells the gcc driver to produce a shared library which can then be linked with other object files to form an executable at link time and which also can be mapped into the address space of the executable at run time. Using `-shared` is the preferred way to build a shared ELF library.

Another very useful command line option for gcc is `-Wl,`*ldoption*, which passes *ldoption* as an option to the linker. If *ldoption* contains commas, it is split into multiple options at the commas.

The `-static` option will generate an executable linked with the static libraries. When the `-static` option is not used, the linker will first try the shared library, then the static library if the shared version is not available.

There are a few other command line options of the GNU linker which are very useful or specific to ELF.

-dynamic-linker *file*
: Set the name of the dynamic linker. The default dynamic linker is either `/usr/lib/libc.so.1` or `/usr/lib/libdl.so.1`.

-export-dynamic
: Tell the linker to make all global symbols in the executable available to the dynamic linker. It is especially useful in the dynamic loading when a dynamically loaded shared library makes references to symbols in the executable which are normally not available to the dynamic linker.

-l*file*
: Add *file* to the list of files to link. This option may be used any number of times. `ld` will search its path-list for occurrences of lib*file*.so, or lib*file*.a if the `-static` option is also used, for each specified *file*. In the former case, the shared library name lib*file*.so will be recorded in the resulting executable or shared library. When the resulting executable or shared library is loaded into memory, the dynamic linker will also map all of the recorded shared libraries into the address space of the process image. In the later case, the bodies of the needed functions and data are copied into the executables, accounting for much code bloat.

-m *emulation*
: Emulate the *emulation* linker. One can list the available emulations with the `-V` option.

-M | -Map *mapfile*
: Print, to standard output or the file *mapfile*, a link map — diagnostic information about where symbols are mapped by `ld`, and information on global common storage allocation.

-rpath *directory*
: Add a directory to the run-time library search path. All `-rpath` arguments are concatenated and passed to the dynamic linker. They are used to locate shared libraries at run time.

-soname *name*
: When creating an shared library, the specified *name* is recorded in the shared library. When an executable linked with this shared library is run, the dynamic linker will attempt to map the shared library specified by the recorded name instead of the file name given to the linker.

-static
: Tell the linker not to link with any shared libraries. It is only applied to executables.

-verbose
: Tell the linker to print out every file it tries to open.

The beta version of gcc for Linux running ELF uses the `-dynamic-linker` *file* option to set the dynamic linker to `/lib/ld-linker.so.1`. That makes both ELF and old a.out DLL shared libraries coexist without problems.

One interesting feature when one builds a shared library is if one passes an option -lfoo, e.g.:

```
# gcc -shared -o libbar.so libbar.o -lfoo
```

The side effect is that if **libfoo.so** is used for building the shared library, when **libbar.so** is mapped into the address space of a process image, the dynamic linker will also map **libfoo.so** into the memory. This feature is very useful if **libbar.so** needs **libfoo.so**. One doesn't have to add -lfoo to link an executable which uses **libbar.so**. If the archive version **libfoo.a** is used, it will only be searched when there are symbols in **libfoo.a** which are referenced by **libbar.o**. Sometimes it is desirable to include **libfoo.a** in **libbar.so** even if they are not referenced by **libbar.o** at all. In that case, one has to add the .o files in **libbar.o** literally by:

```
# rm -rf /tmp/foo
# mkdir /tmp/foo
# (cd /tmp/foo; ar -x ....../libfoo.a)
# gcc -shared -o libbar.so libbar.o /tmp/foo/*.o
# rm -rf /tmp/foo
```

The .o files in libfoo.a have to be compiled with `-fPIC` or at least are compatible with `PIC`.

When one uses

```
static void * __libc_subinit_bar__
        __attribute__ ((section ("_libc_subinit"))) = &(bar);
```

to put a symbol into a section which is not defined in the linker (in this case it is `_libc_subinit`,) the linker will put all the symbols in the `_libc_subinit` section together and create two special symbols, `__start__libc_subinit` and `__stop__libc_subinit`, which can be used as C identifiers.

A word of caution: **It is entirely possible that the linker may not even search the files which contain the `_libc_subinit` section if no symbols in them are needed by the executable. It is up to the programmer to make sure the `_libc_subinit` section is seen by the linker.**

One way to do it is to put a dummy symbol in the `_libc_subinit` section and define it in the file which makes references to the `_libc_subinit` section.

5.5 ELF under Linux

The ELF implementation under Linux has some unique features which are very useful to Linux users. It is very close to the Solaris ELF implementation [2] with a few Linux's own extentions.

5.5.1 ELF macros

In <gnu-stabs.h>, we defined a few macros which can be used to manipulate the symbols.

elf_alias(name1, name2)
: Defines an alias *name2* for the symbol *name1*. It should be used in the file where the symbol *name1* is defined.

weak_alias(name1, name2)
: Defines a weak alias *name2* for the symbol *name1*. The linker will use *name1* to resolve the reference to *name2* only if *name2* is not defined anywhere else. It should also be used in the file where the symbol *name1* is defined.

elf_set_element(set, symbol)
: Forces *symbol* becomes an element of *set*. A new section is created for each *set*.

symbol_set_declare (set)
: Declares *set* for use in this module. It actually decalres two symbols:

  - A symbol for the start of *set*.
    `extern void *const __start_`*set*.
  - A symbol for the end of *set*.
    `extern void *const __stop_`*set*.

symbol_set_first_element(set)
: Returns a pointer (`void *const *`) to the first element of *set*.

symbol_set_end_p(set, ptr)
: Returns true if and only if *ptr* (a `void *const *`) has been incremented past the last element in *set*.

Using these macros, programmers can create lists from different sources file at will.

5.5.2 Library locations and search paths

Under Linux, most of system libraries are installed under `/usr/lib`. Only a few essential shared libraries are installed in `/lib`. Those libraries are **libc.so**, **libcurses.so**, **libm.so** and **libtermcap.so** which are necessary for starting up the Linux system before other partitions are mounted. The default search paths for linker are `/lib`, `/usr/lib`, `/usr/local/lib` and `/usr/i486-linux/lib` in this order.

The environment variable `LD_LIBRARY_PATH` may hold a list of directories, separated by colons (:), which is checked by the dynamic linker to search for shared libraries. For example, the string `/usr/X11R6/lib:/usr/local/lib:` tells the dynamic linker to search first the directory `/usr/X11R6/lib`, then `/usr/local/lib`, and then the current directory to find the shared libraries required in addition to the default directories.

There is a new environment variable `ELF_LD_LIBRARY_PATH` which plays a similar role to `LD_LIBRARY_PATH`. Since `LD_LIBRARY_PATH` is also used with the old a.out DLL Linux shared libraries, to avoid unnecessary warnings from the DLL linker sometimes `ELF_LD_LIBRARY_PATH` is a better choice to provide additional directories to the ELF dynamic linker under Linux.

Another feature is the **/etc/ld.so.conf** file. This file consists of a list of directories, e.g.:

```
/lib/elf
/usr/X11R6/lib
/usr/local/lib
/usr/i486-linuxaout/lib
```

The program **ldconfig** will search the directories listed in `/etc/ld.so.conf` and store all of the shared libraries which are found in `/etc/ld.so.cache`. The Linux ELF dynamic linker will find the shared libraries in **/etc/ld.so.cache** if the shared libraries have been moved from the default directories.

5.5.3 Library versions for shared libraries

On an ELF system, if two shared libraries have the same subset of an application binary interface (ABI), those two shared libraries are interchangable for the executables which only use the subet of the ABI, assuming they have the same functionality.

When a library is changed, as long as the new ABI is 100% compatible with the previous version, all executables linked with the previous version of the shared library will run fine with the new shared library. To support this, the library **foo** has to be maintained with care:

- A shared library, should be built with

  ```
  # gcc -shared -Wl,-soname,libfoo.so.major \
      -o libfoo.so.major.minor.patch-level libfoo.o
  ```

  Then the dynamic linker will try to locate and map in **libfoo.so.major** at run time regardless the actual shared library filename **libfoo.so.major.minor.patch-level** is used at link time.

- A symbolic link should be provided to point to the correct shared library.

  ```
  # ln -s libfoo.so.major.minor.patch-level libfoo.so.major
  ```

- When the ABI is changed to be incompatible with the previous versions, the major version number should be updated.

While searching for a shared library, the Linux linker will use the latest shared library which has the highest major, minor and patch level version numbers.

5.5.4 Linking with a mix of shared and static libraries

By default, the linker will link against the shared libraries if they are available. But **-Bdynamic** and **-Bstatic** provides finer control of which library, shared or static should searched for each library specified for the linker. The library search option is toggled by each **-Bdynamic** or **-Bstatic**:

```
# ld ... -Bstatic -lfoo -Bdynamic -lbar ....
```

This command tells the linker to search the static library for **libfoo**, and search the shared version for **libbar** and any libraries after it until a **-Bstatic** is seen.

To pass the **-Bdynamic** and **-Bstatic** options to the linker, one needs to do

```
# gcc -o main main.o -Wl,-Bstatic -lfoo -Wl,-Bdynamic -lbar
```

Since the gcc driver calls the linker with some default libraries,

```
# gcc -o main main.o -Wl,-Bstatic
```

tells the linker to use the static versions for all default libraries like **libc** and etc.

5.5.5 Loading additional shared libraries

On an ELF system, to execute an ELF executable, the kernel passes the control to the dynamic linker which is **ld-linux.so.1** in case of Linux. The absolute pathname **/lib/ld-linux.so.1** is stored in the binary. If the dynamic linker is not present, no ELF executables will run.

The dynamic linker performs the following actions to finish the process image for the program:

- Analyzes the executable's dynamic information section and determines what shared libraries are needed.
- Locates and maps in these shared libraries, and analyzes the their dynamic information sections to determine what additional shared libraries are requried.
- Performs relocations for the executable and the required shared libraries.
- Calls any initialization functions provided by the required shared libraries and arranges the cleanup functions provided by the shared libraries to be executed when the shared libraries are detached from the address space of the program.
- Transfers control to the program.
- Provides the delayed function binding service to the application.
- Provides the dynamic loading service to the application.

The environmental varialble `LD_PRELOAD` cab be set to a shared library filename or a string of filenames separated by a colon ':'. The dynamic linker will map them into the address space *after* the dynamic executable and before any other required shared libraries. For example:

```
# LD_PRELOAD=./mylibc.so myprog
```

Here **./mylibc.so** will be mapped in first for the executable **myprog**. Since the dynamic linker always uses the first occurrence of the required symbol for the symbol search, one can use `LD_PRELOAD` to override functions in the standard shared library. This feature is useful for programmers to experiment with a new implementation of a function in a shared library without rebulding the whole shared library.

5.5.6 Dynamic loading under Linux

There is a function in the dynamic linker interface library, *_dlinfo*. It lists all modules currently mapped in by the executable and each shared library opened via *dlopen*. The output may look like:

```
List of loaded modules
        00000000 50006163 50006200 Exe 1
        50007000 5000620c 50006200 Lib 1 /lib/elf/libdl.so.1
        5000a000 500062c8 50006200 Lib 2 /lib/elf/libc.so.4
        50000000 50006000 00000000 Int 1 /lib/elf/ld-linux.so.1
        500aa000 08006f00 08005ff0 Mod 1 ./libfoo.so

Modules for application (50006200):
        50006163
        5000620c /lib/elf/libdl.so.1
        500062c8 /lib/elf/libc.so.4
        50006000 /lib/elf/ld-linux.so.1
Modules for handle 8005ff0
        08006f00 ./libfoo.so
        500062c8 /lib/elf/libc.so.4
        50006163
        5000620c /lib/elf/libdl.so.1
        500062c8 /lib/elf/libc.so.4
        50006000 /lib/elf/ld-linux.so.1
```

It can be used to examine the dynamic linking and dynamic loading.

`gcc` configured for Linux running ELF passes `-export-dynamic` to the linker if the `-rdynamic` option is used. It is highly recommended for any executables which use dynamic loading. That is the reason why `LDFLAGS=-rdynamic` was in `Makefile` in our example. For the time being, this option only works under Linux. But `-Wl,-export-dynamic` can be used to pass `-export-dynamic` to be the GNU linker on other platforms.

You can find the detailed descriptions about gcc and GNU link editor in [3] and [4].

6 Programming in Assembly Language with PIC

When one specifies `-fPIC` to gcc, gcc will generate the *position independent code* (PIC) assembly language code from the C source code. But sometimes one needs to program in assembly language and support PIC at the same time. This is what happened in the Linux C library.

PIC under ELF is implemented using the *base register*. All the symbols are referenced via the base register under PIC and therefore to write assembly language code with PIC, one has to SAVE THE BASE REGISTER. Due to *position independent code*, the target destinations of control transfer instructions must be displacements or computed under PIC. For x86-based machines, the base register is `ebx`. Here we introduce two ways to write PIC-safe assembly code under x86. These techniques are used in the Linux C library.

6.1 Inline Assembly Statements in C

`gcc` provides the inline assembly statement feature to let programmers use assembly language from C, which was very useful to write the Linux system call interface as well as to use some machine-dependent instructions not used by `gcc`.

System calls under Linux are implemented via `int $0x80`. Typically, the syscall with three arguments looks like:

```
#include <sys/syscall.h>

extern int errno;

int read (int fd, void *buf, size count)
{
  long ret;

  __asm__ __volatile__ ("int $0x80"
               : "=a" (ret)
               : "0" (SYS_read), "b" ((long) fd),
                 "c" ((long) buf), "d" ((long) count): "bx");
  if (ret >= 0)
  {
    return (int) ret;
  }
  errno = -ret;
```

```
  return -1;
}
```

The asm statement above puts the system call number, `SYS_read`, into `eax`, *fd* into `ebx`, *buf* into `ecx`, *count* into `edx` and then puts `eax` in *ret* upon return from `int $0x80`. This definition works fine without `-fPIC`. Ideally, with `-fPIC` gcc should dectect that `ebx` will be clobbered and save/restore it around the asm statement. But unfortunately, this is not the case. We have to support PIC in the asm statement ourselves:

```
#include <sys/syscall.h>

extern int errno;

int read (int fd, void *buf, size count)
{
  long ret;

  __asm__ __volatile__ ("pushl %%ebx\n\t"
                        "movl %%esi,%%ebx\n\t"
                        "int $0x80\n\t"
                        "popl %%ebx"
                      : "=a" (ret)
                      : "0" (SYS_read), "S" ((long) fd),
                        "c" ((long) buf) "d" ((long) count): "bx");
  if (ret >= 0)
  {
    return (int) ret;
  }
  errno = -ret;
  return -1;
}
```

Here we put *fd* into `esi` first, then we save `ebx`, move `esi` into `ebx` and restore `ebx` after `int $0x80`. This ensures `ebx` is not changed (except during `int $0x80`, but we don't care).

The same principle also applies to other inline assembly statements. ONE HAS TO SAVE AND RESTORE `ebx` ANYTIME WHEN IT MAY BE CHANGED.

6.2 Programming in Assembly Language

If we need to pass five arguments in a system call, the inline assembly statement won't work with PIC since x86 doesn't have enough registers to go around. We need to program in assembly language directly.

The normal assembly language code for

```
syscall (int syscall_number, ...);
```

looks like:

```
        .file "syscall.S"
        .text
        .global syscall
        .global errno
        .align 16
syscall:
        pushl %ebp
        movl %esp,%ebp
        pushl %edi
        pushl %esi
        pushl %ebx
        movl 8(%ebp),%eax
        movl 12(%ebp),%ebx
        movl 16(%ebp),%ecx
        movl 20(%ebp),%edx
        movl 24(%ebp),%esi
        movl 28(%ebp),%edi
        int $0x80
        test %eax,%eax
        jge .LLexit
        negl %eax
        movl %eax,errno
        movl $-1,%eax
.LLexit:
        popl %ebx
        popl %esi
        popl %edi
        movl %ebp,%esp
        popl %ebp
        ret
        .type syscall,@function
.L_syscall_end:
        .size syscall,.L_syscall_end - syscall
```

Under PIC, we have to access any global variable via the global offset table in addition to preserving the base register `ebx`. The modified code is:

```
        .file "syscall.S"
        .text
        .global syscall
```

```
        .global errno
        .align 16
syscall:
        pushl %ebp
        movl %esp,%ebp
        pushl %edi
        pushl %esi
        pushl %ebx
        call .LL4
.LL4:
        popl %ebx
        addl $_GLOBAL_OFFSET_TABLE_+[.- .LL4],%ebx
        pushl %ebx
        movl 8(%ebp),%eax
        movl 12(%ebp),%ebx
        movl 16(%ebp),%ecx
        movl 20(%ebp),%edx
        movl 24(%ebp),%esi
        movl 28(%ebp),%edi
        int $0x80
        popl %ebx
        movl %eax,%edx
        test %edx,%edx
        jge .LLexit
        negl %edx
        movl errno@GOT(%ebx),%eax
        movl %edx,(%eax)
        movl $-1,%eax
.LLexit:
        popl %ebx
        popl %esi
        popl %edi
        movl %ebp,%esp
        popl %ebp
        ret
        .type syscall,@function
.L_syscall_end:
        .size syscall,.L_syscall_end - syscall
```

Finally, if one wants to program in assembly language for PIC and one is not sure how to do it, one can always write the C code and then do

```
# gcc -O -fPIC -S foo.c
```

This will tell gcc to generate the assembly output, `foo.s`. One can take a look at and modify it according to one's needs.

7 Conclusion

As we have discussed above, ELF is a very flexible binary format. It provides many useful functionalities. It puts very few restrictions on programs and programmers. It eases the process of building shared library and integrates dynamic loading with the shared library via an interface library to the dynamic linker. Under ELF, global constructors and destructors in `C++` can be treated in shared libraries the same way as in static libraries.

References

[1] *Operating System API Reference: UNIX SVR4.2*, UNIX Press, 1992.

[2] *SunOS 5.3 Linker and Libraries Manual*, SunSoft, 1993.

[3] Richard M. Stallman, *Using and Porting GNU CC for Version 2.6*, Free Software Foundation, September 1994.

[4] Steve Chamberlain and Roland Pesch, *Using ld: The GNU linker, ld version 2*, Cygnus Support, January 1994.