

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 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 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 [11pt]article
```

ELF: From The Programmer's Perspective Hongjiu Lu
hjl@nynexst.com
NYNEX Science & Technology, Inc.
500 Westchester Avenue
White Plains, NY 10604, USA

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.

5 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.

6 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.

7 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 termi-

nation 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.

7.1 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.

8 Dynamic Linking and Dynamic Loading in ELF

8.1 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.

8.2 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");
}

```