The ELF Object File Format: Introduction

By Eric Youngdale
Created 1995-04-01 02:00

The Executable and Linking Format (ELF) has been a popular topic lately. People wonder why the kernel configuations script asks whether or not to donfigure loading ELF executables. As ELF will eventually be the common object file format for Linux binaries, it is appropriate to document it a bit. This month, Eric introduces us to ELF, and next month he will give us a guided tour of real ELF files.

Now that we are on the verge of a public release of ELF file format compilers and utilities, it is a logical time to explain the differences between **a.out** and ELF, and discuss how they will be visible to the user. As long as I am at it, I will also guide you on a tour of the internals of the ELF file format and show you how it works. I realize that Linux users range from people brand new to Unix to people who have used Unix systems for years--for this reason I will start with a fairly basic explanation which may be of little use to the more experienced users, because I would like this article to be useful in some way to as many people as possible.

People often ask why we are bothering with a new file format. A couple reasons come to mind--first, the current shared libraries can be somewhat cumbersome to build, especially for large packages such as the X Window System that span multiple directories. Second, the current **a.out** shared library scheme does not support the **dlopen()** function, which allows you to tell the dynamic loader to load additional shared libraries. Why ELF? The Unix community seems to be standardizing this file format; various implementations of SVr4 such as MIPS, Solaris, Unixware currently use ELF; SCO will reportedly switch to ELF in the near future; and there are rumors of other vendors switching to ELF. One interesting sidenote--Windows NT uses a file format based upon the COFF file format, the SVr3 file format that the Unix community is abandoning in favor of ELF.

Let us start at the beginning. Users will generally encounter three types of ELF files--**.o** files, regular executables, and shared libraries. While all of these files serve different purposes, their internal structure files are quite similar. Thus we can begin with a general description, and proceed to a discussion of the specifics of the three file types. Next month, I will demonstrate the use of the readelf program, which can be used to display and interpret various portions of ELF files.

One universal concept among all different ELF file types (and also **a.out** and many other executable file formats) is the notion of a section. This concept is important enough to spend some time explaining. Simply put, a section is a collection of information of a similar type. Each section represents a portion of the file. For example, executable code is always placed in a section known as **.text**; all data variables initialized by the user are placed in a section known as **.data**; and uninitialized data is placed in a section known as **.bss**

In principle, one could devise an executable file format where everything is jumbled together--MS-DOS binaries come to mind. But dividing executables into sections has important advantages. For example, once you have loaded the executable portions of an executable into memory, these memory locations need not change. (In principle, program executable code could modify itself, but this is considered to be extremely poor programming practice.) On modern machine architectures, the memory manager can mark portions of memory read-only, such that any attempt to modify a read-only memory location results in the program dying and dumping core. Thus, instead of merely saying that we do not expect a particular memory location to change, we can specify that any attempt to modify a read-only memory location is a fatal error indicating a bug in the application. That being said, typically you cannot individually set the read-only status for each byte of memory--instead you can individually set the protections of regions of memory known as pages. On the i386 architecture the page size is 4096 bytes--thus you could indicate that addresses **0-4095** are read-only, and bytes **4096** and up are writable, for example.

Given that we want all executable portions of an executable in read-only memory and all modifiable locations of memory (such as variables) in writable memory, it turns out to be most efficient to group all of the executable portions of an executable into one section of memory (the **.text** section), and all modifiable data areas together into another area of memory (henceforth known as the **.data** section).

A further distinction is made between data variables the user has initialized and data variables the user has not initialized. If the user has not specified the initial value of a variable, there is no sense wasting space in the executable file to store the value. Thus, initialized variables are grouped into the **.data** section, and uninitialized variables are grouped into the **.bss** section, which is special because it doesn't take up space in the file--it only tells how much space is needed for uninitialized variables.

When you ask the kernel to load and run an executable, it starts by looking at the image header for clues about how to load the image. It locates the **.text** section within the executable, loads it into the appropriate portions of memory, and marks these pages as read-only. It then locates the **.data** section in the executable and loads it into the user's address space, this time in read-write memory. Finally, it finds the location and size of the **.bss** section from the image header, and adds the appropriate pages of memory to the user's address space. Even though the user has not specified the initial values of variables placed in **.bss**, by convention the kernel will initialize all of this memory to zero.

Typically each **a.out** or ELF file also includes a symbol table. This contains a list of all of the symbols (program entry points, addresses of variables, etc.) that are defined or referenced within the file, the address associated with the symbol, and some kind of tag indicating the type of the symbol. In an **a.out** file, this is more or less the extent of the information present; as we shall see later, ELF files have considerably more information. In some cases, the symbol tables can be removed with the strip utility. The advantage is that the executable is smaller once stripped, but you lose the ability to debug the stripped binary. With **a.out** it is always possible to remove the symbol table from a file, but with ELF you typically need some symbolic information in the file for the program to load and run. Thus, in the case of ELF, the strip program will remove a portion of the symbol table, but it will never remove all of the symbol table.

Finally, we need to discuss the concept of relocations. Let us say you compile a simple "hello world" program:

```
main( )
{
        printf("Hello World\n");
}
```

The compiler generates an object file which contains a reference to the function **printf** . Since we have not defined this symbol, it is an external reference. The executable code for this function will contain an instruction to call **printf**, but in the object code we do not yet know the actual location to call to perform this function. The assembler notices that the function **printf** is external, and it generates a relocation, which contains several components. First, it contains an index into the symbol table--this way, we know which symbol is being referenced. Second, it contains an offset into the **.text** section, which refers to the address of the operand of the call instructions. Finally, it contains a tag which indicates what type of relocation is actually present. When you link this file, the linker walks through the relocations, looks up the final address of the external function **printf**, then patches this address back into the operand of the call instruction so the call instruction now points to the actual function **print**.

a.out executables have no relocations. The kernel loader cannot resolve any symbols and will reject any attempt to run such a binary. An **a.out** object file will of course have relocations, but the linker must be able to fully resolve these to generate a usable executable.

So far everything I have described applies to both **a.out** and ELF. Now I will enumerate the shortcomings of **a.out** so that it is more clear why we would want to switch to ELF.

First, the header of an **a.out** file (struct exec, defined in **/usr/include/linux/a.out.h**) contains limited information. It only allows the above-described sections to exist and does not directly support any additional sections. Second, it contains only the sizes of the various sections, but does not directly specify the offsets within the file where the section starts. Thus the linker and the kernel loader have some unwritten understanding about where the various sections start within a file. Finally, there is no built-in shared library support--**a.out** was developed before shared library technology was developed, so implementations of shared libraries based on **a.out** must abuse and misuse some of the existing sections in order to accomplish the tasks required.

About 6 months ago, the default file format switched from ZMAGIC to QMAGIC files. Both of these are **a.out** formats, and the only real difference is the different set of unwritten understandings between the linker and kernel. Both formats of executable have a 32 byte header at the start of the file, but with ZMAGIC the **.text** section starts at byte offset 1024, while with QMAGIC the **.text** section starts at the beginning of the file and includes the header. Thus ZMAGIC wastes disk space, but, more importantly, the 1024 byte offset used with ZMAGIC makes efficient buffer cache management within the kernel more difficult. With a QMAGIC binary, the mapping from the file offset to the block representing a given page of memory is more natural, and should allow for some performance enhancements in the kernel. ELF binaries are also formatted in a natural way that is compatible with possible future changes to the buffer cache.

I have said that shared library support in **a.out** is lacking--while this is true, it is not impossible to design shared library implementations that work with **a.out**. The current Linux shared libraries are certainly one example; another example is SunOS-style shared libraries which are currently used by BSD-*du-jour*. SunOS-style shared libraries contain a lot of the same concepts as ELF shared libraries, but ELF allows us to discard some of the really silly hacks that were required to piggyback a shared library implementation onto **a.out**.

Before we go into our hands-on description of how ELF works, it would be worthwhile to spend a little time discussing some general concepts related to shared libraries. Then when we start to pick apart an ELF file, it will be easier to see what is going on.

First, I should explain a little bit about what a shared library is; a surprising number of people look at shared libraries as sort of black boxes without a good understanding of what goes on inside. Most users are at least aware of the fact that if they mess up their shared libraries, the system can become nearly unusable. This leads most people to treat them with a certain reverence.

If we step back a little bit, we recall that non-shared libraries (also known as static libraries) contain useful procedures that programs might wish to make use of. Thus the programmer does not need to do everything from scratch, but can use a set of standard well-defined functions. This allows the programmer to be more productive. Unfortunately, when you link against a static library, the linker must extract all library functions you require and make them part of your executable, which can make it rather large.

The idea behind a shared library is that you would somehow take the contents of the static library (not literally the contents, but usually something generated from the same source tree), and pre-link it into some kind of special executable. When you link your program against the shared library, the linker merely makes note of the fact that you are calling a function in a shared library, so it does not extract any executable code from the shared library. Instead, the linker adds instructions to the executable which tell the startup code in your executable that some shared libraries are also required, so when you run your program, the kernel starts by inserting the executable into your address space, but once your program starts up, all of these shared libraries are also added to your address space. Obviously some mechanism must be present for making sure that when your program calls a function in the shared library, it actually branches to the correct location within the shared library. I will be discussing the mechanics of this for ELF in a little bit.

More info about ELF [1]

Now that we have explained shared libraries, we can start to discuss some of the general concepts related to how shared libraries are implemented under ELF. To begin with, ELF shared libraries are position independent. This means that you can load them more or less anywhere in memory, and they will work. The current **a.out** shared libraries are known as fixed address libraries: each library has one specific address where it must be loaded to work, and it would be foolish to try to load it anywhere else. ELF shared libraries achieve their position independence in a couple of ways. The main difference is that you should compile everything you want to insert into the shared library with the compiler switch **-fPIC**. This tells the compiler to generate code that is designed to be position independent, and it avoids referencing data by absolute address as much as possible.

Position independence does not come without a cost, however. When you compile something to be PIC, the compiler reserves one machine register (**%ebx** on the i386) to point to the start of a special table known as the global offset table (or GOT for short). That this register is reserved means that the compiler will have less flexibility in optimizing code, and this means that it will take longer to do the same job. Fortunately, our benchmark indicates that for most normal programs the drop in performance is less than 3% for a worst case, and in many cases much less than this.

Another ELF feature is that its shared libraries resolve symbols and externals at run time. This is done using a symbol table and a list of relocations which must be performed before the image can start to execute. While this sounds like it could be slow, a number of optimizations built into ELF make it fairly fast. I should mention that when you compile PIC code into a shared library, there are generally very few relocations, one more reason why the performance impact is not of great concern. Technically, it is possible to generate a shared library from code that was not compiled with **-fPIC**, but an incredible number of relocations would need to be performed before the shared library was usable, another reason why **-fPIC** is important.

When you reference global data within a shared library, the assembly code cannot simply load the value from memory the way you would do with non-PIC code. If you tried this, the code would not be position independent and a relocation would be associated with the instruction where you were attempting to load the value from the variable. Instead, the compiler/assembler/linker create the GOT, which is nothing more than a table of pointers, one pointer for each global variable defined or referenced in the shared library. Each time the library needs to reference a given variable, it first loads the address of the variable from the GOT (remember that the address of the GOT is always stored in **%ebx** so we only need an offset into the GOT). Once we have this, we can dereference it to obtain the actual value. The advantage of doing it this way is that to establish the address of a global variable, we need to store the address in only one place, and hence we need only one relocation per global variable.

We must do something similar for functions. It is critical that we allow the user to redefine functions which might be in the shared library, and if the user does, we want to force the shared library to always use the version the user defined and never use the version of the function in the shared library. Since the function could conceivably be used lots of times within a shared library, we use something known as the procedure linkage table (or PLT) to reference all functions. In a sense this is nothing more than a fancy name for a jumptable, an array of jump instructions, one for each function that you might need to go to. Thus if a particular function is called from thousands of locations within the shared library, control will always pass through one jump instruction. This way, you need only one relocation to determine which version of a given function is actually called, and from the standpoint of performance this is about as good as you are going to get.

Next month, we will use this information to dissect real ELF files, explaining specifics about the ELF file format.

email: ljeditor@ssc.com [2]

Links

[1] http://www.linuxjournal.com//articles/lj/0012/1059/1059s1.html
[2] http://www.linuxjournal.com/mailto:ljeditor@ssc.com
[3] https://www.ssc.com/lj/subs/NewUSA.html

Source URL: http://www.linuxjournal.com/article/1059