LINUX JOURNAL
(/)

Linkers and Loaders

Software (/taxonomy/term/17)

by Sandeep Grover on November 26, 2002

Linking is the process of combining various pieces of code and data together to form a single executable that can be loaded in memory. Linking can be done at compile time, at load time (by loaders) and also at run time (by application programs). The process of linking dates back to late 1940s, when it was done manually. Now, we have *linkers* that support complex features, such as dynamically linked shared libraries. This article is a succinct discussion of all aspects of linking, ranging from relocation and symbol resolution to supporting position-independent shared libraries. To keep things simple and understandable, I target all my discussions to ELF (executable and linking format) executables on the x86 architecture (Linux) and use the GNU compiler (GCC) and linker (ld). However, the basic concepts of linking remain the same, regardless of the operating system, processor architecture or object file format being used.

Compiler, Linker and Loader in Action: the Basics

Consider two program files, a.c and b.c. As we invoke the GCC on a.c b.c at the shell prompt, the following actions take place:

```
gcc a.c b.c
```

- Run preprocessor on a.c and store the result in intermediate preprocessed file.

```
cpp other-command-line options a.c /tmp/a.i
```

- Run compiler proper on a.i and generate the assembler code in a.s

```
cc1 other-command-line options /tmp/a.i  -o /tmp/a.s
```

- Run assembler on a.s and generate the object file a.o

```
as other-command-line options /tmp/a.s  -o /tmp/a.o
```

cpp, cc1 and as are the GNU's preprocessor, compiler proper and assembler, respectively. They are a part of the standard GCC distribution.

Repeat the above steps for file b.c. Now we have another object file, b.o. The linker's job is to take these input object files (a.o and b.o) and generate the final executable:

```
   ld other-command-line-options /tmp/a.o /tmp/b.o -o a.out
```

The final executable (a.out) then is ready to be loaded. To run the executable, we type its name at the shell prompt:

```
./a.out
```

The shell invokes the loader function, which copies the code and data in the executable file a.out into memory, and then transfers control to the beginning of the program. The loader is a program called execve, which loads the code and data of the executable object file into memory and then runs the program by jumping to the first instruction.

a.out was first coined as the Assembler OUTput in a.out object files. Since then, object formats have changed variedly, but the name continues to be used.

Linkers vs. Loaders

Linkers and loaders perform various related but conceptually different tasks:

- Program Loading. This refers to copying a program image from hard disk to the main memory in order to put the program in a ready-to-run state. In some cases, program loading also might involve allocating storage space or mapping virtual addresses to disk pages.

- Relocation. Compilers and assemblers generate the object code for each input module with a starting address of zero. Relocation is the process of assigning load addresses to different parts of the program by merging all sections of the same type into one section. The code and data section also are adjusted so they point to the correct runtime addresses.
- Symbol Resolution. A program is made up of multiple subprograms; reference of one subprogram to another is made through symbols. A linker's job is to resolve the reference by noting the symbol's location and patching the caller's object code.

So a considerable overlap exists between the functions of linkers and loaders. One way to think of them is: the loader does the program loading; the linker does the symbol resolution; and either of them can do the relocation.

Object Files

Object files comes in three forms:

- Relocatable object file, which contains binary code and data in a form that can be combined with other relocatable object files at compile time to create an executable object file.
- Executable object file, which contains binary code and data in a form that can be directly loaded into memory and executed.
- Shared object file, which is a special type of relocatable object file that can be loaded into memory and linked dynamically, either at load time or at run time.

Compilers and assemblers generate relocatable object files (also shared object files). Linkers combine these object files together to generate executable object files.

Object files vary from system to system. The first UNIX system used the a.out format. Early versions of System V used the COFF (common object file format). Windows NT uses a variant of COFF called PE (portable executable) format; IBM uses its own IBM 360 format. Modern UNIX systems, such as Linux and Solaris use the UNIX ELF (executable and linking format). This article concentrates mainly on ELF.

ELF Header

.text

.rodata

.data

.bss

.symtab

.rel.text

.rel.data

.debug

.line

.strtab

The above figure shows the format of a typical ELF relocatable object file. The ELF header starts with a 4-byte magic string, \177ELF. The various sections in the ELF relocatable object file are:

- .text, the machine code of the compiled program.
- .rodata, read-only data, such as the format strings in printf statements.
- .data, initialized global variables.
- .bss, uninitialized global variables. BSS stands for block storage start, and this section actually occupies no space in the object file; it is merely a placer holder.
- .symtab, a symbol table with information about functions and global variables defined and referenced in the program. This table does not contain any entries for local variables; those are maintained on the stack.
- .rel.text, a list of locations in the .text section that need to be modified when the linker combines this object file with other object files.
- .rel.data, relocation information for global variables referenced but not defined in the current module.
- .debug, a debugging symbol table with entries for local and global variables. This section is present only if the compiler is invoked with a -g option.
- .line, a mapping between line numbers in the original C source program and machine code instructions in the .text section. This information is required by debugger programs.
- .strtab, a string table for the symbol tables in the .symtab and .debug sections.

Symbols and Symbol Resolution

Every relocatable object file has a symbol table and associated symbols. In the context of a linker, the following kinds of symbols are present:

- Global symbols defined by the module and referenced by other modules. All non-static functions and global variables fall in this category.
- Global symbols referenced by the input module but defined elsewhere. All functions and variables with extern declaration fall in this category.
- Local symbols defined and referenced exclusively by the input module. All static functions and static variables fall here.

The linker resolves symbol references by associating each reference with exactly one symbol definition from the symbol tables of its input relocatable object files. Resolution of local symbols to a module is straightforward, as a module cannot have multiple definitions of local symbols. Resolving references to global symbols is trickier, however. At compile

time, the compiler exports each global symbol as either strong or weak. Functions and initialized global variables get strong weight, while global uninitialized variables are weak. Now, the linker resolves the symbols using the following rules:

1. Multiple strong symbols are not allowed.
2. Given a single strong symbol and multiple weak symbols, choose the strong symbol.
3. Given multiple weak symbols, choose any of the weak symbols.

For example, linking the following two programs produces linktime errors:

```
/* foo.c */
int foo () {
   return 0;
}
```

```
/* bar.c */
 int foo () {
    return 1;
 }
 int main () {
    foo ();
 }
```

The linker will generate an error message because foo (strong symbol as its global function) is defined twice.

```
gcc foo.c bar.c
/tmp/ccM1DKre.o: In function 'foo':
/tmp/ccM1DKre.o(.text+0x0): multiple definition of 'foo'
/tmp/ccIhvEMn.o(.text+0x0): first defined here
collect2: ld returned 1 exit status
```

Collect2 is a wrapper over linker ld that is called by GCC.

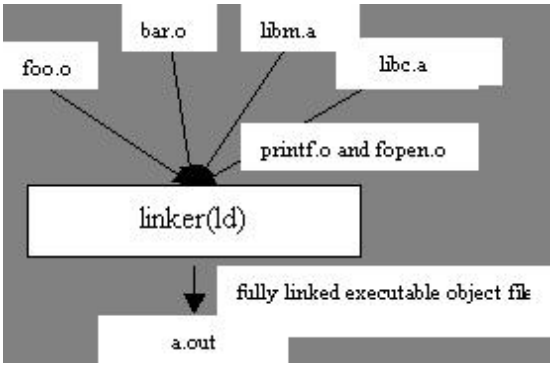
Linking with Static Libraries

A static library is a collection of concatenated object files of similar type. These libraries are stored on disk in an archive. An archive also contains some directory information that makes it faster to search for something. Each ELF archive starts with the magic eight character string !<arch>\n, where \n is a newline.

Static libraries are passed as arguments to compiler tools (linker), which copy only the object modules referenced by the program. On UNIX systems, libc.a contains all the C library functions, including printf and fopen, that are used by most of the programs.

```
gcc foo.o bar.o /usr/lib/libc.a /usr/lib/libm.a
```

libm.a is the standard math library on UNIX systems that contains the object modules for math functions such as like sqrt, sin, cos and so on.



During the process of symbol resolution using static libraries, linker scans the relocatable object files and archives from left to right as input on the command line. During this scan, linker maintains a set of O, relocatable object files that go into the executable; a set U, unresolved symbols; and a set of D, symbols defined in previous input modules. Initially, all three sets are empty.

- For each input argument on the command line, linker determines if input is an object file or an archive. If input is a relocatable object file, linker adds it to set O, updates U and D and proceeds to next input file.
- If input is an archive, it scans through the list of member modules that constitute the archive to match any unresolved symbols present in U. If some archive member defines any unresolved symbol that archive member is added to the list O, and U and D are updated per symbols found in the archive member. This process is iterated for all member object files.
- After all the input arguments are processed through the above two steps, if U is found to be not empty, linker prints an error report and terminates. Otherwise, it merges and relocates the object files in O to build the output executable file.

This also explains why static libraries are placed at the end of the linker command. Special care must be taken in cases of cyclic dependencies between libraries. Input libraries must be ordered so each symbol is referenced by a member of an archive and at least one definition of a symbol is followed by a reference to it on the command line. Also, if an unresolved symbol is defined in more than one static library modules, the definition is picked from the first library found in the command line.

Relocation

Once the linker has resolved all the symbols, each symbol reference has exactly one definition. At this point, linker starts the process of relocation, which involves the following two steps:

- Relocating sections and symbol definitions. Linker merges all the sections of the same type into a new single section. For example, linker merges all the .data sections of all the input relocatable object files into a single .data section for the final executable. A similar process is carried out for the .code section. The linker then assigns runtime memory addresses to new aggregate sections, to each section defined by the input module and also to each symbol. After the completion of this step, every instruction and global variable in the program has a unique loadtime address.
- Relocating symbol reference within sections. In this step, linker modifies every symbol reference in the code and data sections so they point to the correct loadtime addresses.

Whenever assembler encounters an unresolved symbol, it generates a relocation entry for that object and places it in the .relo.text/.relo.data sections. A relocation entry contains information about how to resolve the reference. A typical ELF relocation entry contains the following members:

- Offset, a section offset of the reference that needs to be relocated. For a relocatable file, this value is the byte offset from the beginning of the section to the storage unit affected by relocation.
- Symbol, a symbol the modified reference should point to. It is the symbol table index with respect to which the relocation must be made.
- Type, the relocation type, normally R_386_PC32, that signifies PC-relative addressing. R_386_32 signifies absolute addressing.

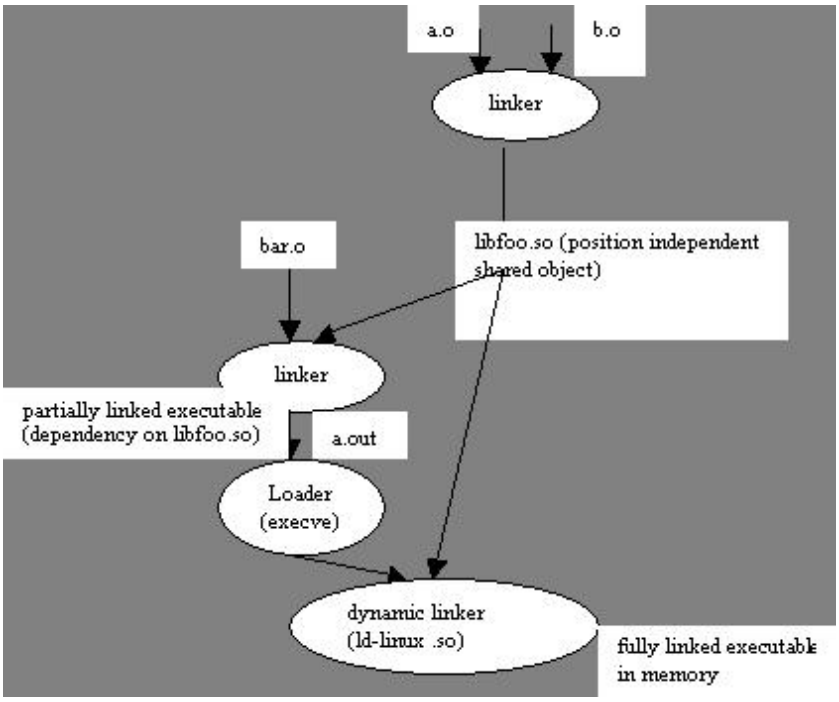
The linker iterates over all the relocation entries present in the relocatable object modules and relocates the unresolved symbols depending on the type. For R_386_PC32, the relocating address is calculated as S + A - P; for R_386_32 type, the address is calculated as S + A. In these calculations, S denotes the value of the symbol from the relocation entry, P denotes the section offset or address of the storage unit being relocated (computed using the value of offset from relocation entry) and A is the address needed to compute the value of the relocatable field.

Dynamic Linking: Shared Libraries

Static libraries above have some significant disadvantages; for example, consider standard functions such as printf and scanf. They are used almost by every application. Now, if a system is running 50-100 processes, each process has its own copy of executable code for printf and scanf. This takes up significant space in the memory. Shared libraries, on the other hand, address the disadvantages of static libraries. A shared library is an object module that can be loaded at run time at an arbitrary memory address, and it can be linked to by a program in memory. Shared libraries often are called as shared objects. On most UNIX systems they are denoted with a .so suffix; HP-UX uses a .sl suffix and Microsoft refer to them as DLLs (dynamic link libraries).

To build a shared object, the compiler driver is invoked with a special option:

```
gcc -shared -fPIC -o libfoo.so a.o b.o
```



The above command tells the compiler driver to generate a shared library, libfoo.so, comprised of the object modules a.o and b.o. The -fPIC option tells the compiler to generate position independent code (PIC).

Now, suppose the main object module is bar.o, which has dependencies on a.o and b.o. In this case, the linker is invoked with:

```
gcc bar.o ./libfoo.so
```

This command creates an executable file, a.out, in a form that can be linked to libfoo.so at load time. Here a.out does not contain the object modules a.o and b.o, which would have been included had we created a static library instead of a shared library. The executable simply contains some relocation and symbol table information that allow references to code and data in libfoo.so to be resolved at run time. Thus, a.out here is a partially executable file that still has its dependency in libfoo.so. The executable also contains a .interp section that contains the name of the dynamic linker, which itself is a shared object on Linux systems (ld-linux.so). So, when the executable is loaded into memory, the loader passes control to the dynamic linker. The dynamic linker contains some start-up code that maps the shared libraries to the program's address space. It then does the following:

- relocates the text and data of libfoo.so into memory segment; and
- relocates any references in a.out to symbols defined by libfoo.so.

Finally, the dynamic linker passes control to the application. From this point on, location of shared object is fixed in the memory.

Loading Shared Libraries from Applications

Shared libraries can be loaded from applications even in the middle of their executions. An application can request a dynamic linker to load and link shared libraries, even without linking those shared libraries to the executable. Linux, Solaris and other systems provides a series of function calls that can be used to dynamically load a shared object. Linux provides system calls, such as dlopen, dlsym and dlclose, that can be used to load a shared object, to look up a symbol in that shared object and to close the shared object, respectively. On Windows, LoadLibrary and GetProcAddress functions replace dlopen and dlsym, respectively.

Tools for Manipulating Object Files

Here's a list of Linux tools that can be used to explore object/executable files.

- ar: creates static libraries.
- objdump: this is the most important binary tool; it can be used to display all the information in an object binary file.
- strings: list all the printable strings in a binary file.
- nm: lists the symbols defined in the symbol table of an object file.
- ldd: lists the shared libraries on which the object binary is dependent.
- strip: deletes the symbol table information.

Suggested Reading

Linkers and Loaders (http://www.iecc.com/linker) by John Levine

Linkers and Libraries Guide from Sun (http://docs.sun.com/db?p=/doc/816-1386)

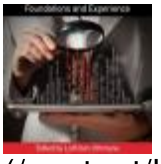
Sandeep Grover works for QuickLogic Software (http://www.quicklogic.com) (India) Pvt. Ltd.
email: sgrover@quicklogic.com (mailto:sgrover@quicklogic.com)

You May Like

For Open-Source Software, the Developers Are All of Us (/content/open-source-software-developers-are-all-us)

Derek Zimmer (/users/derek-zimmer-0)

(/content/open-source-software-developers-are-all-us)



(/content/lotfi-ben-othmane-martin-gilje-jaatun-and-edgar-weippls-empirical-research-software-security)

Lotfi ben Othmane, Martin Gilje Jaatun and Edgar Weippl's Empirical Research for Software Security (CRC Press) (/content/lotfi-ben-othmane-martin-gilje-jaatun-and-edgar-weippls-empirical-research-software-security)

James Gray (/users/james-gray)

Heirloom Software: the Past as Adventure (/content/heirloom-software-past-adventure)

Eric S. Raymond (/users/eric-s-raymond)

(/content/heirloom-software-past-adventure)

SoftMaker FreeOffice (/content/softmaker-freeoffice)

James Gray (/users/james-gray)

(/content/softmaker-freeoffice)