

Gentle Introduction to Linking and Loading

Joel Agnel Fernandes

June 17, 2010

1 Introduction

If you're reading this sentence, this is a ongoing long term effort. I would welcome donations / volunteers or any other form of support if you find this material useful. The author is really fond of linking and loading mechanisms and hasn't yet found a guide that explains the topic in a friendly and easy-to-understand manner. There are some great books out there ofcourse but they mostly don't have enough real examples or enough pictures (yes the author loves pictures :D) which can overwhelm the unsuspecting beginner. Every topic explained in this series will contain examples and figures wherever applicable to make sure the concepts register.

This series doesn't cover every architecture and platform. The goal is to understand how linkers work with real examples of a particular system and architecture. For our purposes, we will use an x86 computer running Linux. It is probably a good idea to install a GNU/Linux based system to work the examples as you read if you haven't yet. It will also help if you are familiar with C and assembly and have used a compiler before.

This series is organized into different parts. The first part explains the anatomy of an ELF object which is the standard format for binaries (executables, objects, shared objects etc..) in a typical unix system. It is important to understand the basic structure of an object file format to get an understanding how linkers and loaders work. A lot can be understood by just studying this format. We will start with a cursory overview of the ELF format and deal with specifics only as we study linking and loading.

2 Relocation

Relocation is at the heart of linking. It is often unknown at compile time what the address of a particular global variable or function is. Lets look at an example,

Listing 1: get_a.c: get variable a from someone

```
1 extern a;
2 void main(int argc, char **argv) {
3     printf("The value of a is %d\n", a);
4     return;
5 }
```

Compile this using the command line: `gcc -o -c get_a.o get_a.c`

Listing 2: define_a.c: define variable a

```
1 #include <stdio.h>
2 int a = 2;
3 void somefunc() {
4     return;
5 }
```

Compile this using the command line: `gcc -o -c define_a.o define_a.c`

Now you should have 2 object files `get_a.o` and `define_a.o` which we will analyze shortly.

One of the main stages of linking is to fix the addresses in compiled code in the object files being linked. Why does it need to do this? At the time of compiling a particular C file, the compiler has no way of knowing the addresses of external functions and variables that it requires, its only the linker that knows this because it looks at all object files at once, not just one. So because the compiler doesn't know the addresses, what it does it simply inserts 0s or some garbage in the assembly code in all these places and makes a note of them in the relocation tables. The compiler merely defers this job to the linker, and the way it communicates to the linker is by creating entries in relocation tables.

Lets understand these concepts with some examples. First lets look at the disassembly of `get_a.o`: `objdump -d get_a.o`

Listing 3: disassembly of section .text

```
1
2 00000000 <main>:
3   0: 55                push   %ebp
4   1: 89 e5            mov    %esp,%ebp
5   3: 83 e4 f0        and    $0xffffffff0,%esp
6   6: 83 ec 10        sub    $0x10,%esp
7   9: 8b 15 00 00 00 00  mov    0x0,%edx    # address of
   global variable 'a'
8   f: b8 00 00 00 00    mov    $0x0,%eax
9  14: 89 54 24 04      mov    %edx,0x4(%esp)
10 18: 89 04 24        mov    %eax,(%esp)
11 1b: e8 fc ff ff ff  call   1c <main+0x1c>
12 20: c9              leave
13 21: c3              ret
```

For the purpose of illustration, we will ignore all other lines of code, suffice to say that, as shown above, `edx` contains the value of variable `a` which is set to 0 by the compiler as the actually address of the variable `a` is unknown at compile time. Also make a note that the address from the beginning of the text section which has been hardcoded to 0 is `0xb`. Now lets look at the relocation table entries created by the compiler to account for the above zeroing.

Listing 4: Relocation table for get_a.o

```
1 readelf -r get_a.o
```

```

2
3  Offset      Info      Type           Sym.Value  Sym. Name
4 0000000b    00000901 R_386_32      00000000   a
5 00000010    00000501 R_386_32      00000000   .rodata
6 0000001c    00000a02 R_386_PC32    00000000   printf

```

As we can see an entry for global variable `a` is created, but its symbol value is not known yet. The compiler has simply stored enough information for the linker to take over later and complete unfinished business.

What information?

Lets try to understand the different parts of a relocation entry and how the linker uses this information for relocation.

First, it is important to understand that each relocation tables applies a particular symbol table to a particular section. This section cannot not only be code (which we just saw), but also special tables as we will see later. Once the linker lays out the code in the final object file, it knows the precise location of each symbol and updates the symbol table accordingly with the values, this is called symbol resolution. After the symbol table is updated, the linker now has to patch the relevant code/data sections of the corresponding with the new symbol values, this is called relocation.

Again, I repeat, each relocation section contains a relocation table which applies a particular symbol table to a particular code/data section. How do you find out which ones? This information is stored in the relocation section's header. Lets find out what these values are in our example, first dump all the section headers:

Listing 5: section headers for `get_a.o`

```

1 There are 11 section headers, starting at offset 0xe4:
2
3 Section Headers:
4  [Nr] Name                Type           Addr      Off      Size
5      ES Flg Lk Inf Al
6  [ 0]                      NULL           00000000 000000 000000
7      00      0  0  0
8  [ 1] .text                  PROGBITS       00000000 000034 000022
9      00 AX  0  0  4
10 [ 2] .rel.text              REL            00000000 000364 000018
11      08      9  1  4
12 [ 3] .data                  PROGBITS       00000000 000058 000000
13      00 WA  0  0  4
14 [ 4] .bss                   NOBITS         00000000 000058 000000
15      00 WA  0  0  4
16 [ 5] .rodata                 PROGBITS       00000000 000058 000016
17      00  A  0  0  1
18 [ 6] .comment                PROGBITS       00000000 00006e 000024
19      01 MS  0  0  1
20 [ 7] .note.GNU-stack         PROGBITS       00000000 000092 000000
21      00      0  0  1
22 [ 8] .shstrtab                STRTAB         00000000 000092 000051
23      00      0  0  1

```

```

14  [ 9] .symtab          SYMTAB          00000000 00029c 0000b0
      10      10      8  4
15  [10] .strtab          STRTAB          00000000 00034c 000017
      00      0      0  1

```

In `.rel.text` header, `Lk` is the symbol table index and `Inf` is the section to which the symbol table should be applied to during relocation. This shows us that the `.symtab` symbol table has to be applied to the `.text` section as a part of relocation.

Coming back to the different parts of a relocation entry for variable `a`, showing it again here:

Listing 6: Relocation table for `get_a.o`

```

1 readelf -r get_a.o
2
3  Offset      Info      Type           Sym.Value  Sym. Name
4 0000000b    00000901 R_386_32      00000000   a
5 00000010    00000501 R_386_32      00000000   .rodata
6 0000001c    00000a02 R_386_PC32    00000000   printf

```

The `Offset` field contains the offset location from the beginning of the text section which needs to be patched with the real address of variable `a` (which was obtained after symbol resolution). The offset in the case of variable `a` is `0xb` which also matches the calculation we did during disassembly (please refer to the disassembly we did earlier). So in this case, the linker would goto the location `0xb` in `.text` and update it with the value `a` from the `.symtab` symbol table. We will look at the symbol `a`'s final value after disassembling the final fully linked executable a little later.