Implementation of A Static Linker for the x86_64 Linux Platform

Zhaohui Yang, Grace Zhang
University of California, Santa Barbara
{zhaohui, gracezhang}@ucsb.edu

I. Overview

The linker is a critical component in the software development process, responsible for linking object files and libraries to produce executable files or shared libraries. Following our goal of implementing a much-functionality robust static linker targeted for the Linux x86_64 platforms, we have performed a detailed account of the design, implementation, and evaluation works. To accomplish the goal, our implementation of the linker is in Python and employs the LIEF(Library to Instrument Executable Files) library for handling ELF format files. In this report, we thoroughly describe the implementation, illustrate the problems addressed, solutions employed, and findings obtained during the development of the static linker.

II. Introduction

With the rapid advancement of computers and technologies, an increasing amount of capabilities is enabled to build innovative systems. In interest of developing these systems, this usually requires large programs to accomplish. Suppose that a large-scale application is written in a single huge source file; if something in the application needs to be modified, meaning that the entire file has to be recompiled again, which is quite cumbersome to do so. With the help of linkers, a large program can be broken down into small and manageable modules, enabling the development process to be more modular [1]. In addition, having separate modules improves efficiency since they can be compiled separately as well, meaning that only the modules that have been modified need to be recompiled rather than the entire program.

Although linking is one of the important steps within the software development process, it is not well understood nor studied [2]. The System V Unix's documentation used to have a detailed description of how the linker works, but it was removed in the later versions with specifics of the ELF format. However, as programmers, especially those who build large programs, it is essential to understand the linking process [1]. For instance, knowing how the linker works would help to resolve issues when faced upon errors of missing libraries or modules and to practice better programming practices.

Acknowledging the significance of the linker, we implemented a static linker to gain deeper insights of this topic. The targeted platform for our implementation of the static linker is x86-64 Linux since System V is a superset of the other common operating systems [2], such as Microsoft Windows and MacOS, and therefore allows us to comprehend the basic structure of the linker that most operating systems follow. The ultimate goal of our linker is to achieve the functionalities of a simple compile-time static linker: symbol resolution and relocation. In the following sections, we discuss the implementation of our linker, interesting findings through the development process, challenges that we faced, and future work for this project.

III. Implementation

A. Design and architecture

The overall structure of our static is divided into three main stages: symbol resolution, merging sections, and relocation of symbol references. The inputs to the linker are a list of relocatable object files generated

from the GNU compiler and the name of the executable object file; the output is a runnable executable file. Figure 1 displays the high-level diagram of our linker.

Since the goal for our linker is to solely replicate the same functionalities of an actual static compile time linker, the execution time is not considered. Thus for simplicity reasons, we implemented our linker with Python and utilized a binary format parser and modifier library, LIEF, to handle the ELF format files.

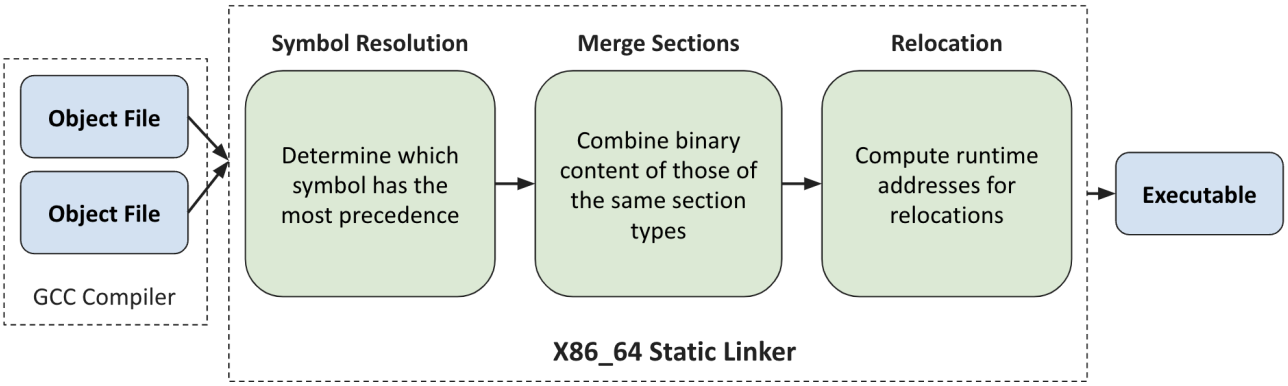

Fig. 1. High level diagram of our implementation of the linker

B. Impementation details

In the first stage of our linker, symbol resolution, all of the symbols from the input relocatable object files are iterated through to resolve any name-conflicting symbols, indicating that each symbol reference should only have one symbol definition. This is accomplished by checking each symbol's binding, type, and section header index. Local symbols are only referenced in the same input module that they are defined in, so no symbol resolution is needed to be done for these symbols. The symbols that need to be resolved are mainly the global and weak binding symbols, in which their references may not be in the same module that they are defined. When there are multiple defined symbols, there are specific rules that the linker follows to select the actual symbol definitions. For the Unix/Linux linker as well as our linker, the following rules are applied:

1) Multiple strong global symbols are not allowed. For instance, a function and an initialized and allocated data object with the same name cannot exist together.
2) Choose the strong global symbol over the weak global symbol. In this case, a weak symbol means that the symbol is either undefined or is uninitialized and not allocated yet. Therefore it makes sense to pick the strong global symbol which is already defined.
3) Choose any of the weak symbols when there are multiples of them.

In our implementation of symbol resolution, we used LIEF to extract the symbols and their information from the input relocatable object files and then applied the rules described above. Once the symbols are resolved, they are added to the symbol table of the output executable file.

The next stage is to merge the sections of the same name from the input relocatable object files together. The task here is more straightforward in comparison to symbol resolution; since our task is to achieve a minimal static linker, we only consider the .text, .data, and .bss sections. Deeper into the specifics, for each section of the same name, our linker first combines the contents of these sections together, compute the new offsets from the beginning of the combined new section in the resulting executable file, update the memory runtime addresses and the size of the section, and take care of the alignment constraints. Once these are accomplished, the sections are converted into segments with the help of the internal data

structure for sections and segments pointing towards the same data in LIEF. Figure 2 demonstrates the section and segment data structure. By default, memory addresses are mapped to 0x400000 for the 64 bit Linux platform, and for simplicity purposes, each segment's memory address is separated by a single 4KB page size.

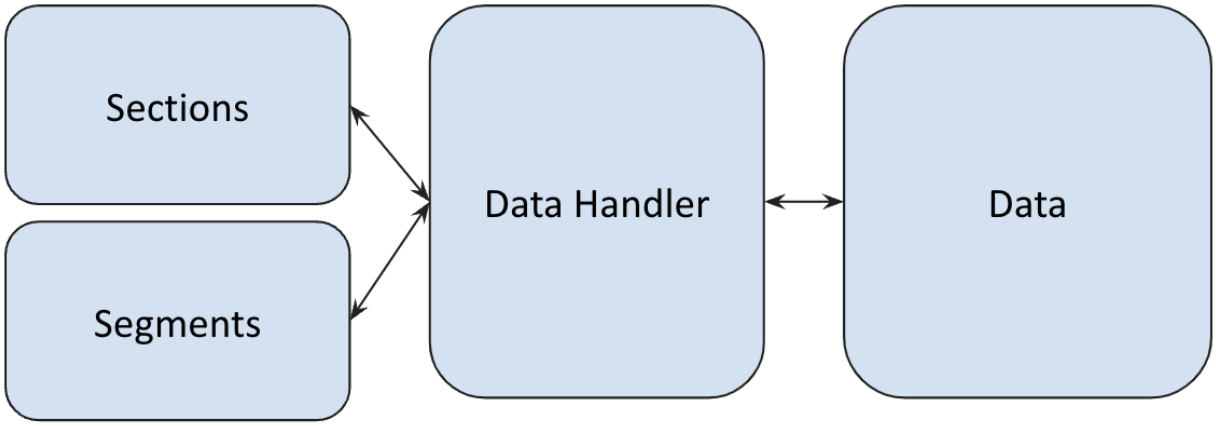

Fig. 2. Internal LIEF data structure that handles data for section and segments

The final stage of our linker is to relocate the symbols references using the relocation entries within each input relocatable file. The input modules are compiled separately and none of the locations referenced outside the module are known yet. Therefore, the compiler is designed to generate relocation entries to instruct how the linker should modify the executable file such that the symbols reference the correct run time addresses. Given the information from the relocation entries extracted with LIEF, the location or pointer of the reference to be updated is found, then the actual reference from the pointer's value is computed according to the relocation type of the entry: R_386_32 and R_386_PC32 are the types that are considered in our implementation. For R_386_32, meaning that it is an absolute reference, the reference is simply the symbol's run time address plus the relocation entry's addend. On the other hand, R_386_PC32 is a PC-relative reference adding the symbol's run-time address, the relocation entry's addend, and the reference's run-time address, which is the run-time address of the section that the reference pointer resides in plus the section offset. (386 or x86)

After all of the stages described above are completed, in which all the byte contents of the output executable file are in the correct places, there is still one thing left to take care of: the ELF header. LIEF does not automatically update the header according to the targeted platform, so information such as the program header size and offsets need to be manually updated to match those of an actual x86-64 executable file.

C. User interface and use-case

The command line user interface of our linker is similar to that of GNU standard linker component `ld`, the GNU linker. A `main.py` file is set up such that it parses through the command line inputs and passes them into our implementation of the linker for linking. The first argument is a list of the relocatable files, followed by the options of our linker and then the destination executable file's name. Note that since our linker is only capable of static linking, there is only one option now. Below shows an example of using our linker:

```
python main.py main.o sum.o -e main.out
```

TABLE I
EVALUATION OF LINKER FUNCTIONALITIES

Symbol Resolution	Use the `readelf -s` option to view the symbols from both the GNU linker executable and that of ours. The resulting symbols should match in both files aside from the symbols used for the default dynamic linking in the GNU linker.
Relocation of Symbol References	Use `objdump -d` to disassemble the input relocatable files and both the GNU linker executable and that of ours. After disassembling all files, the first step is to identify the undefined references within the relocatable files. Next, check those undefined references' place amongst the executable files and compare their resulting references, which should turn out to be the same address differences from the reference pointer.

IV. EVALUATION

In order to ensure the correctness and reliability of our static linker, various validation strategies are employed. The criteria for the evaluation of the functionalities of our linker includes the following: the output executable file returns the correct result and that symbol resolution and relocation of symbol references computed by our linker are accurate. There are not necessarily any simple ways to set up validations through coding unit tests, so the techniques applied for testings are mainly through manual inspections of the outputs.

To test the first criteria, the output of the executable file generated by our linker is compared to that of the executable file from GNU; they both should return the same results. Below demonstrates the command used for generating the GNU executable file:

```
gcc -o main-gnu.out main.o sum.o
```

Next, both symbol resolution and relocation of symbol references are also verified against the GNU linker by utilizing the `readelf` and `objdump` Linux commands to display information of ELF files. Table I summarizes the approaches on how the information from the command line tools are used for verification.

V. FINDINGS AND DISCUSSION

During the development process of our linker, we have discovered some findings from researching and reading documentations: using the LIEF library has limitations and special case for handling relocation types. Although LIEF is quite a large library with many functionalities and features, it has its own weaknesses. For instance, it is not capable of creating ELF format files from scratch, which is tough to walk around with. Moreover, some functions do not return the information as described in their documentation and that it is not possible to modify some information such as which sections belong to which segment. For example, the function for adding relocation entries or objects do not work as intended, in which adding them to the executable file does not actually perform this task and does not even return an error. The next discovery was quite interesting; according to the public Linux GitHub repository, the relocation of symbol references are treated the same way for R_386_PLT32 and R_386_PC32 [3]. This is not stated on any documentation, so it was quite surprising to discover this on the repository.

VI. Conclusion

The GNU linker nowadays mainly performs dynamic linking and is capable of handling static and dynamic libraries; in general, much more than the minimal static linker that we have implemented. If given time to extend this project, the ultimate goal would be to integrate as many capabilities as possible to match the GNU linker. Furthermore, rather than employing LIEF to handle everything, implementing our own ELF format parser and modifier from scratch would be preferred to enable more control over the files.

As linker is a topic not well understood and studied, it is a great accomplishment already to be able to implement a simple static linker. Although the process was quite challenging, throughout the entire development of our linker, it has immensely improved our understanding of the x86-64 platform further: the detailed structure of the linker and the ELF format.

References

[1] R. E. Bryant and D. R. O'Hallaron, *Computer Systems A Programmer's Perspective*. Pearson; 3rd edition, 2015.

[2] S. Kell, D. P. Mulligan, and P. Sewell, "The missing link: Explaining elf static linking, semantically," *ACM SIGPLAN Notices*, vol. 51, no. 10, p. 607–623, 2016.

[3] x86: Treat R_X86_64_PLT32 as R_X86_64_PC32. [Online]. Available: https://github.com/torvalds/linux/commit/b21ebf2fb4cde1618915a97cc773e287ff49173e