What is the thinnest linker feature set required to produce a bootable kernel?

Rachel Samuelsson
Advisor: Sofie Kjellgren

NTI Johanneberg

HT 2020-VT 2021

Abstract

In order to create an operating system one must produce a bootable kernel. This kernel, like any piece of compiled software, is composed of many different object files that must be linked by a linker. Existing linkers are large pieces of code, which makes them hard to port to new platforms. This makes it harder to create a self hosting operating system. This report aims to answer the query: what is the thinnest linker feature set required to produce a bootable kernel? To find out a linker, dubbed "relocatable to executable format" (or "rtef" for short), was implemented in the C programming language. During research and implementation it was found that to produce a bootable kernel a linker needs to be able to read a configuration specifying section addresses, alignments, and offsets; read any number of object files; merge the object files' sections; resolve the object files' symbols, and write an executable file.

Contents

1 Introduction

1.1 Query

What is the thinnest linker feature set required to produce a bootable kernel?

1.2 Background

In modern society software is not only essential but omnipresent. The most essential piece of software of all is undeniably the kernel. The kernel is the core of an operating system which closes the gap between user applications and the hardware.

During compiled software development there is a toolchain which converts source files into the finished program. One of the major tools in this chain is the linker. The linkers job is to combine compiled object files into a complete program. Like all compiled software, a kernel must be linked. However, existing linkers are large, arguably over-engineered, programs that are difficult to port to new operating systems. This portability issue makes it much harder to create a self-hosting operating system since you'd have to port the linker.

This report aims to find the thinnest linker feature which can produce a bootable kernel by designing a program capable of ELF object file linking which retains high portability and uses no POSIX or other non-standard C extensions.

To limit the scope of this project, the program will only target the x86_64 processor architecture, only support the ELF format for object and executable files, and have no support for dynamic linking.

1.3 Theory

When a computer runs a program the operating system must load an executable file into memory and tell the CPU to start running it. Many modern programming languages such as ruby for example are interpreted, which means that rather than executing the program itself the OS executes an interpreter which then runs the file.[5] However, this process is quite slow and as such there is still a need to write compiled programs.

During the compilation of a program from source code to machine code, there are generally two steps.[4]

First, the source files containing the code are compiled into object files by a

compiler. These objects contain machine code and metadata.[4]

These object files are then processed by a linker whose job is to resolve missing symbols in object files with defined symbols in other object files, calculating absolute memory addresses, and then combining the code from these objects into a file that the OS can load and the CPU can run.[5]

x86_64 is the name of a processor architecture. A processor architecture is in essence a description of a CPU's available storage units and instructions. Different architectures will differ in features and will therefore have different linking processes.[3] At the time of writing x86_64 is the dominant CPU architecture for many years. This largely due to it being the only CPU architecture well supported by Microsoft Windwos, which has been the market leading operating system since 2009.[1]

ELF, executable and linkable format, is a file format that can be used for both object and executable files.[2] It is the standard format for these files on nearly all all UNIX based operating systems since System V Release 4.[5]

ELF files are based on headers and sections.[2]

At the beginning of an ELF file lies the ELF header, a header that describes the general contents of the file, if the file is an executable or object file, as well as the positions of the files section and program headers.[2]

Program headers describe what parts of the file to load into memory upon execution of the file as well as at what address these parts should be loaded. Since program headers only describe execution they need only be present in executable files.[2]

Section headers describe the type and position of sections in the file. The content of sections may include (but isn't limited to) machine code, string tables, symbol tables, relocation tables. String table entries contain the name of the files sections and symbol's. Symbol table entries describe symbol's names, which sections they are defined relative to and their values. Relocation table entries describe missing values within the files machine code which need to be resolved by the linker, as well as information regarding how to calculate the correct value. Section headers are not needed at execution and need only be present in object files.[2]

2 Process

The first decision made was to implement the program in the C programming language. There were several factors behind this decision, first of all, C is an incredibly portable language and one of, if not the first one to be implemented for new platforms. Second of all, the elf standard is defined by a C header file. And lastly because of the developer's familiarity with the language.

The first step in the implementation was planning the general flow of the program as well as defining the required data structures.

The following data structures were defined: elf_file, a structure which contains all input data read from a file; sym_def, a structure which contains all definitions of a single symbol; sec_def, a structure which contains all definitions of a single section.

It was determined that the program flow would begin with an optional argument handler, which has the job of parsing the arguments given to the program on the command line, to define input and output files, as well as letting the user specify additional options. In the final product, this wasn't present as no options other than input and output files were implemented.

Secondly, the program runs a file validator, this ensures all input files are valid ELF files. In practice, this was implemented by reading files ELF headers and checking their validity, as well as ensuring they targeted the correct platform (x86-64, in this case).

Thirdly, the program collects all information from the input files, in essence filling out all fields of the elf_file, sym_def and sec_def data structures.

Fourth, the program calculates the memory address for each section in the output file by parsing the sec_def data structures.

Fifth, the program resolves missing symbols in all files by looking at each elf_file's relocation entries and then finding the matching sym_def entry, which after a short calculation dependent on the relocation info resolves the value of the defined symbol.

Sixth, ELF header and program headers are generated based on the values calculated in the fourth step.

Finally, the program writes all data to a file and marks it as executable.

2.1 Issues

Throughout the process, there have been several issues. These were mostly caused by a lack of experience going into the project. For instance, it was initially planned to calculate section addresses after the symbol relocations. Though this is impossible due to some symbols being defined relative to the memory address of sections. Another example of this is the previously mentioned unimplemented first step. This was ultimately skipped as it turned out the program did not need it.

There were also issues with the used data structures as it was found they didn't contain enough information several times. This lead to having to add additional fields of information to the data structures and having to rewrite old code to write information to them. The sym_def data-structure was also seemingly forgotten halfway through development.

There was also a minor issue with marking the output executable as executable by the operating system. This proved problematic as the C standard library doesn't allow us to do this, and using any other extensions would make the program less portable. In the end, it was solved by using preprocessor macros that include extensions only if they are supported.

The final issue encountered during the process was the more advanced relocation types. While writing the program it was discovered that some of the relocation types did not only require a calculation but also a creation of additional tables. One of these types could be circumvented, however, the rest had to be ignored and have as such not been implemented.

3 Results

3.1 Feature set

In its current state rtef is capable of retrieving a number of file paths on the command line, which it interprets as a list of input ELF object files. The sections of these ELF object files alignment, memory address and offset are then calculated, after which all symbols are resolved and an ELF executable is written.

rtef does not support all relocation types, at the moment only 32 and 64 bit direct and instruction relative relocation types are supported.

rtef does not support any additional arguments or options, nor any way to specify the layout of the output file.

rtef has no dynamic linking capabilities and cannot be used to load libraries at runtime.

3.2 Demonstration

Below is a demonstrated test run, omitting the output of running rtef to preserve readability.

For the contents of the files test1.nasm test2.nasm and test3.nasm, please refer to the attachments.

```
deppy@Tower:/home/deppy/proj/c/rtef/test
$ ls
test1.nasm  test2.nasm  test3.nasm
deppy@Tower:/home/deppy/proj/c/rtef/test
$ for x in *.nasm; do nasm -felf64 $x; done
deppy@Tower:/home/deppy/proj/c/rtef/test
$ ../rtef test1.o test2.o test3.o
[...]
deppy@Tower:/home/deppy/proj/c/rtef/test
$ ls
rtef.out  test1.nasm   test1.o      test2.nasm   test2.o      test3.nasm   test3.o
deppy@Tower:/home/deppy/proj/c/rtef/test
$ ./rtef.out
Hi world!
```

The executables contents can be verified using the ‘readelf‘ utility

```
deppy@Tower:/home/deppy/proj/c/rtef/test
$ readelf -h -l rtef.out
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 0c 40 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x20001000
  Start of program headers:          64 (bytes into file)
  Start of section headers:          0 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         2
  Size of section headers:           64 (bytes)
  Number of section headers:         0
  Section header string table index: 0

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000001000 0x0000000020001000 0x0000000000000000
                 0x0000000000000044 0x0000000000000044  R E    0x1000
  LOAD           0x0000000000002000 0x0000000020002000 0x0000000000000000
                 0x000000000000000a 0x000000000000000a  RW     0x1000
```

In this example, the linker resolves symbols in the test1 object file with values from both the test2 and test3 object files. Finally, a single binary is created. Once executed this binary runs code defined in the test1 object file which calls code defined in the test3 object file, passing values defined in the test2 object file.

This not only demonstrates the linker's capability of resolving symbols but also of merging sections as both the test1 and test2 object files have a .text section. This is seen in the output of 'readelf' as there are only two program headers, one representing the .data section, and the other the merged .text section.

4 Discussion

4.1 Current state

In its current state rtef is not capable of linking any kernels. This is because of two issues. The first and most prominent issue being the inability to specify any particular file offsets or alignments for sections. This makes it impossible to create any sort of boot header which a bootloader (or the BIOS) might use to boot the kernel. The second issue is the slim scope of supported relocation methods (as mentioned in the results). These days, most, if not all operating systems will be written in a language that requires one of the unsupported relocation types. For example, C, a programming language often used in kernel development, will almost always require support for a GOT type relocation, which rtef cannot currently handle.

4.2 Possible optimizations

rtef, while functional is an incredibly unoptimized piece of software. This has been done consciously to speed up the development process and to keep the codebase as simple as possible. However, it would be possible to greatly improve the performance of the program without hurting its portability through the use of hash maps for both section and symbol definitions. In addition to this, the overall structure of the program, as well as the used data structures could all be revisited with the knowledge gained from having written the program.

4.3 Further development

While functional in its current state there is much work which could be done on rtef. Besides the aforementioned need for alignment and offset options as well as optimizations there are many things, that could be done to improve rtef. The current portability is good, however, one could improve it even further by relying less on standard library functions as well as adding an optional cmake build chain to enable development on windows. Though, the most important improvement to be made to rtef is to support more relocation types and more platforms.

One might argue a possible direction to take the project is to support dynamic linking as well as commonly used non-standard extensions. This, however, is outside of the scope of rtef. rtef is meant to be a portable program with a minimal codebase. By further implementing features outside of its original scope rtef would eventually suffer from the same issues as the currently existing

linkers, rendering its existence useless.

4.4 Conclusion

To be able to link a kernel a linker needs to be able to read several object files and parse them according to the ELF specification.

Following this, it must enumerate all symbols, sections as well as relocation entries. These data structures must be organized in such a way that they can later be retrieved based on name or index.

Subsequently, the linker must be able to generate memory addresses and alignments for every section, and in doing so, merging sections from different object files. It could be argued that it is possible to create multiple sections per file rather than merging those from different files, however as these sections are of the same name it is a fair assumption of a programmer to assume them to be loaded into the same memory segment, therefore, separating these might cause unexpected behavior. Additionally, creating more segments means the operating system will have to load many more sections, which might prove to be a hassle if many input files were used during executable generation.

Then the linker must be able to parse some sort of configuration from the user. Though this is not needed for a linker per se, it is needed to link kernels as there will be a need to specify the position of sections for the operating system to parse it correctly. This configuration could either be provided through the command line, which might worsen the user experience, or through a configuration file, which while easier for the user would add more work for the linker, as it would have to parse a file.

Following this the linker must resolve the relocation entries for all sections, this could either be implemented by resolving the value for each symbol and then resolving relocation entries or by resolving the symbol for each relocation entry. The ELF specification specifies several symbol visibility rules, however, it's been found that in most cases these are safe to ignore. Though there are many relocation types it was determined that those which need be implemented for a linker are direct 32 bit, direct 64 bit as well as position relative and global offset table relative.

The linker must then generate program headers based on these sections. While it's possible to include section headers in the executable it is by no means needed. Additionally the linker must produce an ELF header.

Finally, the headers and sections need to be written to an output file.

5 References

[1] *Desktop operating system market share worldwide*, StatCounter, (Feb. 2021) `https://gs.statcounter.com/os-market-share/desktop/worldwide/#monthly-200901-202102-bar` (visited on 02/23/2021).

[2] "Executable and linkable format (elf)", in *Tis standard portable formats specification*, 1.1 (Tool Interface Standards commitee (TIS), May 1995).

[3] J. L. Hennesey and D. Patterson, *Computer architecture: a quantitative approach*, 6th ed. (Morgan Kaufmann, Dec. 2017).

[4] L. Presser and J. R. White, *Linkers and loaders*, tech. rep. (University of California, Sept. 1972).

[5] *System v application binary interface*, 4.1, AT&T and The Santa Cruz Operation (Mar. 1997).

6 Attachments

Source Code 1: test1.nasm

```
bits 64

extern msg
extern len

extern print:function
extern exit:function

global _start:function

section .text
_start:
    mov rsi, msg
    mov rdx, len
    call print

    mov rsi, 0
    jmp exit
```

Source Code 2: test2.nasm

```
bits 64

global print:function
global exit:function

section .text
print:
    mov rax, 1
    mov rdi, 1
    syscall
    ret

exit:
    mov rax, 60
    syscall
```

Source Code 3: test3.nasm

```
bits 64
```

```
global msg
global len

section .data
msg db "Hi world!", 0x0a
len equ $ - msg
```