rtef

relocatable to executable format

David Samuelson

HT 2020-VT 2021

Abstract

The aim of this study is to answer the query: what is the thinnest linker feature set required in order to produce a bootable kernel? In order to find out, a linker, dubbed rtef, was implemented in the C programming language. During research and implementation it was found that in order 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

Introduction

Query

What is the thinnest linker feature set required in order to produce a bootable kernel.

Background

An operating systems core, the kernel is made up of many different object files which must be linked by a linker. However existing linkers are large, arguably over engineered, programs which are hard to port to new operating systems. This portability issue makes it much harder to achieve a self hosting operating system.

The purpose of this project is to design a program capable of ELF object file linking which retains high portability and uses no POSIX or other non-standard C extensions.

In order to limit the scope of this project I've chosen to only target the x86_64 processor architecture as well as only supporting the ELF format for object and executable files. The linker will also not be capable of dynamic linking.

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 ruby which then runs the file.[1] 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.[2]

First the source files containing the code are compiled into object files by a compiler. These objects contain machine code and metadata.[3]

These object files are then processed by a linker who's 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 object into a file that the OS can load and the CPU can run.[4]

[1] Wikipedia contributors. (2020, November 29). Executable. In *Wikipedia, The Free Encyclopedia.*

[2] Wikipedia contributors. (2020, November 27). Linker (computing). In *Wikipedia, The Free Encyclopedia.*

[3] Wikipedia contributors. (2020, November 27). Linker (computing). In *Wikipedia, The Free Encyclopedia.*

[4] Wikipedia contributors. (2020, November 27). Linker (computing). In *Wikipedia, The Free Encyclopedia.*

x86_64 is the name of a processor architecture. A processor architecture is in essence a description of a CPU's available registers and instructions. Different architectures might differ in features and will therefore have different linking processes. At the time of writing x86_64 has been the dominant CPU architecture for many years.[5]

ELF, executable and linkable format, is a file format which can be used for both object and executable files.[6] It is the standard format for these files on nearly all all UNIX based operating systems as well as on the PS4 game's console.[7]

ELF files are based on headers and sections.[8]

At the beginning of an ELF file lies the ELF header, a header which 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.[9]

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 at. Since program headers only describe execution they need only be present in executable files.[10]

Section headers describe the type of 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 contains the name of the files sections and symbols. Symbol table entries describe symbols 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.[11]

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

[5] Wikipedia contributors. (2020, November 8). x86-64. In *Wikipedia, The Free Encyclopedia.*

[6] Tool Interface Standards committee (TIS). (1995, May). *Executable and Linkable Format (ELF).* In *TIS standard Portable Formats Specification V1.1.*

[7] Wikipedia contributors. (2020, November 4). Executable and Linkable Format. In *Wikipedia, The Free Encyclopedia.*

[8] Tool Interface Standards committee (TIS). (1995, May). *Executable and Linkable Format (ELF).* In *TIS standard Portable Formats Specification V1.1.*

[9] Tool Interface Standards committee (TIS). (1995, May). *Executable and Linkable Format (ELF).* In *TIS standard Portable Formats Specification V1.1.*

[10] Tool Interface Standards committee (TIS). (1995, May). *Executable and Linkable Format (ELF).* In *TIS standard Portable Formats Specification V1.1.*

[11] Tool Interface Standards committee (TIS). (1995, May). *Executable and Linkable Format (ELF).* In *TIS standard Portable Formats Specification V1.1.*

for new platforms. Second of all, the elf standard is defined by a C header file. And lastly because of the developer's familiarly 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, in order 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 and 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.

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 had no need for 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 in order to write information to them.

The final issue encountered during the process were 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 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.

Results

Feature set

In it's 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.

Demonstration

Below is a demonstrated test run, omitting the output of running rtef in order 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. In the end 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 linkers 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.

Discussion

Current state

In it's 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 in order 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 which 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.

Possible optimizations

rtef, while functional is an incredibly unoptimized piece of software. This has been done consciously in order 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 it's 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.

Further development

While functional in it's 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 which 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 code base. By further implementing features outside of it's original scope rtef would eventually suffer from the same issues as the currently existing linkers, rendering it's existence useless.

Conclusion

In order to be able to link a kernel a linker needs to be able to read a number of object files; parse their content; merge their sections; read some sort of configuration specifying memory addresses, offsets and alignments for sections;

calculate memory addresses for remaining sections; resolve symbols and build the required tables; and write an executable file.

References

Tool Interface Standards committee (TIS). (1995, May). *Executable and Linkable Format (ELF)*. In *TIS standard Portable Formats Specification V1.1*. Retrieved December 3.

Wikipedia contributors. (2020, November 4). Executable and Linkable Format. In *Wikipedia, The Free Encyclopedia*. Retrieved December 3, 2020, from https://en.wikipedia.org/w/index.php?title=Executable_and_Linkable_Format&oldid=987064905.

Wikipedia contributors. (2020, November 27). Linker (computing). In *Wikipedia, The Free Encyclopedia*. Retrieved December 3, 2020, from https://en.wikipedia.org/w/index.php?title=Linker_(computing)&oldid=990946784.

Wikipedia contributors. (2020, November 8). x86-64. In *Wikipedia, The Free Encyclopedia*. Retrieved December 3, 2020, from https://en.wikipedia.org/w/index.php?title=X86-64&oldid=987698018.

Wikipedia contributors. (2020, November 29). Executable. In *Wikipedia, The Free Encyclopedia*. Retrieved December 3, 2020, from https://en.wikipedia.org/w/index.php?title=Executable&oldid=991287717.

Attachments

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
```

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
```

test3.nasm

```
bits 64

global msg
global len

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