《计算机系统》

ELF 文件与链接实验报告

班级: GitHub

学号: 78268851

姓名: AnicoderAndy

目录

1 实验项目 **3**

1.1 项目名称 3

1.2 实验目的 3

1.3 实验目标 3

2 实验任务 **4**

2.1 优化编译指令 4

2.2 编写汇编代码 4

2.3 调整汇编代码 4

2.4 编写 ELF 文件 6

2.5 优化 ELF 文件 8

2.6 最终优化步骤 9

3 总结 **13**

3.1 实验中出现的问题 13

3.2 心得体会 13

1 实验项目

1.1 项目名称

手搓最小可执行文件——ELF 文件与链接实验

1.2 实验目的

- 了解 ELF 文件格式的基本结构，掌握 ELF 文件的基本组成部分。
- 尝试分析链接过程的各个环节，探寻编译过程中文件大小膨胀的原因。

1.3 实验目标

对于下面的源代码文件 `andy.c`：

```c
int main() { return 7; }
```

本实验要分析编译和连接过程中增加了什么内容导致可执行文件的大小产生膨胀，并尽可能获得更小的可执行文件。

实验在 64 位 Ubuntu 22.04 LTS 上进行，使用编译器、汇编器、链接器模拟 32 位环境。使用 gcc 11.4.0，nasm 2.15.05，ld 2.38 作为工具链。Shell 环境为 zsh 5.8.1。

2 实验任务

2.1 优化编译指令

使用 `gcc -m32 andy.c -o andy` 命令编译，得到 `andy` 可执行文件。使用 bash 命令 `ls -l andy` 查看文件大小，得到的结果是此文件大小为 14908B。

从源代码层面来看，该源代码在 C 层面已经无法再精简，因为根据 C 标准要求，每个程序必须含有 `main` 函数作为入口，且必须显示返回非零状态码，此源代码无法再精简。

尝试使用 gcc 提供的`-Os` 选项进行优化，`-s` 选项剔除符号表和重定位信息编译，命令为：`gcc andy.c -m32 -Os -s -o step1`。使用 bash 命令 `ls -l step1` 查看文件大小，得到的结果是此文件大小为 13656B，可以发现文件大小减小并不明显。

2.2 编写汇编代码

为了尝试在汇编层次优化代码，本节熟悉 nasm 的使用方法。新建 `test.asm` 文件，编写汇编代码：

```
bits 32
global main
section .text

main:
    mov eax, 5
    mov ebx, 10
    add eax, ebx
    ret
```

通过 `nasm -f elf32 test.asm ; gcc -m32 -Wall -s test.o` 命令汇编并链接代码。运行得到的程序发现其返回了错误码 15。

2.3 调整汇编代码

虽然 C 源码逻辑很简单，但默认的动态链接过程会引入一整套运行时启动与结束框架，其中包括启动代码（链接 `crt1.o` 等文件，包含 `_start` 等调用）、动态链接器（链接 `ld-linux.so` 等文件）、符号重定位。

为了实现最简单的返回状态码的功能，没有必要调用外部的库函数，所以首先尝试使用`-nostdlib` 来取消连接标准库和启动代码：

```
prompt> gcc -m32 -Wall -s -nostdlib test.o
/usr/bin/ld: warning: cannot find entry symbol _start; defaulting to
↪  0000000000001000
```

```
prompt> ./a.out
[1]    10410 segmentation fault (core dumped)  ./a.out
```

执行命令时 gcc 提出找不到 _start 的警告，运行时因为没有 _start 而导致段错误。所以接下来尝试直接把逻辑写入 _start：

```
bits 32
global _start
section .text

_start:
    mov eax, 7
    ret
```

再次执行前面的汇编、链接指令。尽管这次没有任何警告，但执行时仍然会遇到段错误，原因是 _start 并不是函数，而是程序的入口，不应该有返回值。这里强制执行 ret 命令，调试时发现会尝试返回到 0x0001 的位置，由于权限问题而触发段错误。

对汇编代码进行修改，我们应该使用 int 0x80 唤醒内核，让操作系统结束进程。进行系统调用时，eax 寄存器中需要放入系统调用号（这里放入 1 表示结束进程），ebx 寄存器中需要放入系统调用参数（这里放入 7 表示返回值）：

```
bits 32
global _start
section .text

_start:
    mov eax, 1
    mov ebx, 7
    int 0x80
```

通过 readelf -l a.out 可以发现该文件还包含了很多链接相关的内容：

```
Section to Segment mapping:
 Segment Sections...
  00
  01     .interp
  02     .interp .note.gnu.build-id .gnu.hash .dynsym .dynstr
  03     .text
  04     .eh_frame
  05     .dynamic
  06     .dynamic
  07     .note.gnu.build-id
  08     .dynamic
```

所以我们采用 ld -m elf_i386 test.o -o step3 命令来链接。得到的文件大小为 4472B。

为了后续操作能顺利进行，我们修改 `test.asm` 文件以缩减汇编代码：

```
bits 32
global _start
section .text

_start:
    xor eax, eax
    inc eax
    mov bl, 7
    int 0x80
```

经过汇编、链接得到的 `step3` 文件大小为 4468B。

2.4 编写 ELF 文件

通过 `readelf -h step3` 检查 ELF 头：

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8049000
  Start of program headers:          52 (bytes into file)
  Start of section headers:          4268 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         5
  Section header string table index: 4
```

其中包括了 ELF 格式（ELF32）、编码方式（二进制补码）、操作系统（UNIX）、文件类型（可执行文件）、机器类型（Intel 80386）、入口地址（0x8049000）、程序头表偏移（52 字节）、节头表偏移（4268 字节）等信息。

通过 `readelf -l step3` 检查程序头：

```
Elf file type is EXEC (Executable file)
Entry point 0x8049000
There are 2 program headers, starting at offset 52
```

```
Program Headers:
  Type      Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD      0x000000 0x08048000 0x08048000 0x00074 0x00074 R   0x1000
  LOAD      0x001000 0x08049000 0x08049000 0x00007 0x00007 R E 0x1000

 Section to Segment mapping:
  Segment Sections...
   00
   01     .text
```

其中定义了如何将文件加载到内存并执行。它告诉操作系统如何将 ELF 文件的各个 Segment 映射到内存，还标记了各个段的控制权限。本 ELF 文件中程序头表只包含了两个类型为 LOAD 的段，并且给出了它们在文件中的偏移量、虚拟地址、物理地址（本质上也是虚拟地址）、文件大小、内存大小、控制权限和对齐方式。

通过 `readelf -S step3` 可以发现可执行文件的节与预期相类似：

```
There are 5 section headers, starting at offset 0x10ac:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        08049000 001000 000007 00  AX  0   0 16
  [ 2] .symtab           SYMTAB          00000000 001008 000060 10      3   2  4
  [ 3] .strtab           STRTAB          00000000 001068 000022 00      0   0  1
  [ 4] .shstrtab         STRTAB          00000000 00108a 000021 00      0   0  1
```

`.rodata`、`.data`、`.bss` 分别存放只读变量、已初始化全局变量和静态变量、未初始化全局变量和静态变量。由于本实验没有使用这些变量，所以它们并不存在，也并不必需。对于一个可执行文件来说，其必要的部分是 ELF header、Program header 以及 `.text` 节，其他部分（如节头表、符号表、字符串表、重定位表、调试段等）主要服务于链接器、调试器、分析工具等其他功能。

了解以上信息后，即可开始编写 ELF 文件 `simple_elf.asm`：

```
1  BITS 32
2              org 0x08048000
3  ehdr:                       ; Elf32_Ehdr
4              db 0x7F, "ELF", 1, 1, 1, 0 ; e_ident
5              times 8 db 0
6              dw 2            ; e_type
7              dw 3            ; e_machine
8              dd 1            ; e_version
9              dd _start       ; e_entry
10             dd phdr - $$    ; e_phoff
11             dd 0            ; e_shoff
12             dd 0            ; e_flags
```

```
13              dw ehdrsize     ; e_ehsize
14              dw phdrsize     ; e_phentsize
15              dw 1            ; e_phnum
16              dw 0            ; e_shentsize
17              dw 0            ; e_shnum
18              dw 0            ; e_shstrndx
19  ehdrsize    equ $ - ehdr
20  phdr:                       ; Elf32_Phdr
21              dd 1            ; p_type
22              dd 0            ; p_offset
23              dd $$           ; p_vaddr
24              dd $$           ; p_paddr
25              dd filesize     ; p_filesz
26              dd filesize     ; p_memsz
27              dd 5            ; p_flags
28              dd 0x1000       ; p_align
29  phdrsize    equ $ - phdr
30  _start:
31              xor eax, eax
32              inc eax
33              mov bl, 7
34              int 0x80
35  filesize    equ $ - $$
```

通过 `nasm -f bin simple_elf.asm -o step4` 汇编，通过 `chmod u+x step4` 赋予文件可执行权限。执行`./step4`，正常返回 7。通过 `ls -l step4` 得到的文件大小为 91B。

2.5 优化 ELF 文件

通过 hexdump 命令查看 `step4` 文件的十六进制内容如表 1 所示。

Addr	7f	45	4c	46	01	01	01	00	00	00	00	00	00	00	00	00
00000010	02	00	03	00	01	00	00	00	54	80	04	08	34	00	00	00
00000020	00	00	00	00	00	00	00	00	34	00	20	00	01	00	00	00
00000030	00	00	00	00	01	00	00	00	00	00	00	00	00	80	04	08
00000040	00	80	04	08	5b	00	00	00	5b	00	00	00	05	00	00	00
00000050	00	10	00	00	31	c0	40	b3	07	cd	80					

表 1: `step4` 文件的十六进制内容

不难注意到，ELF 文件头部的 `e_ident` 部分末尾有很多空置的 0（橙色部分），我们程序主体部分的代码一共只有 7 字节，可以填充在这一部分。

同样不难注意到，ELF 头部的末 8 字节（蓝色部分）与程序头的前 8 字节（绿色部分）是完全一致的，所以可以将程序头的标签直接放在 ELF 头部的尾部对应位置。

基于此，修改前面的 `simple_elf.asm` 文件，重命名为 `tiny.asm`:

```
1   BITS 32
```

```
2               org 0x08048000
3   ehdr:                       ; Elf32_Ehdr
4               db 0x7F, "ELF", 1, 1, 1, 0, 0 ; e_ident
5   _start:
6               xor eax, eax
7               inc eax
8               mov bl, 7
9               int 0x80
10              dw 2            ; e_type
11              dw 3            ; e_machine
12              dd 1            ; e_version
13              dd _start       ; e_entry
14              dd phdr - $$    ; e_phoff
15              dd 0            ; e_shoff
16              dd 0            ; e_flags
17              dw ehdrsize     ; e_ehsize
18              dw phdrsize     ; e_phentsize
19  phdr:
20              dw 1            ; e_phnum, p_type low_8
21              dw 0            ; e_shentsize, p_type high_8
22              dw 0            ; e_shnum, p_offset low_8
23              dw 0            ; e_shstrndx, p_offset high_8
24  ehdrsize    equ $ - ehdr
25              dd $$           ; p_vaddr
26              dd $$           ; p_paddr
27              dd filesize     ; p_filesz
28              dd filesize     ; p_memsz
29              dd 5            ; p_flags
30              dd 0x1000       ; p_align
31  phdrsize    equ $ - phdr
32  filesize    equ $ - $$
```

使用 `nasm -f bin tiny.asm -o step5` 汇编后，使用 `chmod u+x step5` 赋予文件可执行权限。执行`./step5`，正常返回 7。通过 `ls -l step5` 得到的文件大小为 76B。

2.6 最终优化步骤

分析 ELF Header 各部分的作用见表 2 所示。

偏移	字段名	大小	作用说明
0x00	e_ident	16	魔数、ELF 类、字节序、版本等识别信息
0x10	e_type	2	文件类型
0x12	e_machine	2	目标平台架构类型（如 x86 是 3）
0x14	e_version	4	ELF 文件版本（目前固定为 1）
0x18	e_entry	4	程序的入口点虚拟地址
0x1C	e_phoff	4	程序头表（Program Header Table）的文件偏移

偏移	字段名	大小	作用说明
0x20	e_shoff	4	段头表（Section Header Table）的文件偏移
0x24	e_flags	4	处理器特定的标志位（如 ARM 有相关定义）
0x28	e_ehsize	2	ELF 文件头的总大小，通常为 52 字节
0x2A	e_phentsize	2	程序头表中每个表项的大小
0x2C	e_phnum	2	程序头表中的表项数量
0x2E	e_shentsize	2	段头表中每个表项的大小
0x30	e_shnum	2	段头表中的表项数量
0x32	e_shstrndx	2	段名称字符串表在段表中的索引

表 2: 32 位 ELF Header 各字段说明表

不难发现，ELF Header 的大部分必要字段都在前半部分，后半部分几乎都可以自由修改，于是我们试图让程序头尽可能占据 ELF 头的后半部分，将此文件命名为 `ultimate.asm`：

```
BITS 32
            org 0x00200000
ehdr:                       ; Elf32_Ehdr
            db 0x7F, "ELF", 1, 1, 1, 0, 0 ; e_ident
_start:
            xor eax, eax
            inc eax
            mov bl, 7
            int 0x80
            dw 2            ; e_type
            dw 3            ; e_machine
            dd 1            ; e_version
            dd _start       ; e_entry
            dd phdr - $$    ; e_phoff
phdr:
            dd 1            ; e_shoff, ptype
            dd 0            ; e_flags, p_offset
            dd $$           ; e_ehsize, p_vaddr
                            ; e_phentsize
            dw 1            ; e_phnum, p_paddr low_8
            dw 0            ; e_shentsize, p_paddr high_8
            dd filesize     ; e_shnum, p_filesz
                            ; e_shstrndx
            dd filesize     ; p_memsz
            dd 5            ; p_flags
            dd 0x1000       ; p_align
filesize    equ $ - $$
```

注意到 Program Header 占据了 ELF Header 中两个非常重要的字段：`e_phentsize` 和 `e_phnum`，它们分别表示每个程序头表项的大小和程序头表项的数量。我们需要调整

程序加载位置使得 e_phentsize 恰与 phdr 地址高 8 位相同。将加载地址高 8 位定位 0x0020（十进制下的 32），低 8 位置零可以满足这一约束。

下面考虑处理 Program Header 的字段。注意到 p_memsz 至少要等于 p_filesz，但如果它更大不会有影响。基于这个事实可以重新组织 ELF 文件（本文件命名为 nightmare.asm）：

```
BITS 32
            org 0x00010000
            db 0x7F, "ELF"  ; e_ident
            dd 1                                ; p_type
            dd 0                                ; p_offset
            dd $$                               ; p_vaddr
            dw 2            ; e_type            ; p_paddr
            dw 3            ; e_machine
            dd _start       ; e_version         ; p_filesz
            dd _start       ; e_entry           ; p_memsz
            dd 4            ; e_phoff           ; p_flags
_start:
            mov bl, 7       ; e_shoff           ; p_align
            xor eax, eax
            inc eax         ; e_flags
            int 0x80
            db 0
            dw 0x34         ; e_ehsize
            dw 0x20         ; e_phentsize
            dw 1            ; e_phnum
            dw 0            ; e_shentsize
            dw 0            ; e_shnum
            dw 0            ; e_shstrndx
filesize    equ $ - $$
```

目前我们将程序主体放在了 ELF Header 的 e_shoff 表项和 e_flags 表项之间。我们将 e_phoff 设置为了 4，使得程序头也和 ELF Header 的较前位置重叠。检查当前程序头的表项，注意到 p_filesz 和 p_memsz 都指向了程序主体的起始位置，我们可以将加载地址设置小一些来减小占用内存（虚拟化内存下内存一般由操作系统动态分配，不会直接分配全部 e_memsz 大小的内存，所以其实这一指标并不重要）。值得注意的是 p_flags 被设置为了 4，表示内存可读而不可写，操作系统要求这种情况下 p_memsz 必须小于等于 p_filesz，我们将两个指标都设置为大于文件实际大小的值就不会引发异常。

使用 nasm 汇编为 step6 文件后通过 readelf -h -l step6 查看头文件发现很多信息已经不再正确（例如节头位置、版本信息等），但它们并不影响程序正常运行。我们又注意到文件的末 7 字节都是 0，Linux 系统在尝试加载不符合要求的 ELF 文件时空缺位都会用 0 补足，所以这些 0 也可通过 truncate -s -7 step6 删除，此时文件大

小为 45B。赋予 `step6` 可执行权限并运行，程序正常返回 7。使用 `readelf -h step6` 出错，因为删除末尾 0 后文件不再是符合规范的 ELF 文件。

3 总结

3.1 实验中出现的问题

- **readelf 与 gdb 中 Entry Point 不一致**。在 2.2 节中完成汇编链接后，使用 `readelf -h` 查看 ELF 入口点为 `0x1000`，但在 gdb 中 `start` 断点位置却在 `0x56556000` 附近。查询资料发现这是因为 `start` 并非程序的实际入口 `_start`，而是相对加载位置的偏移量，gdb 的启动断点并没有错误。

- **手写 ELF 后 nasm 无法汇编**。在 2.4 节中尝试使用 `nasm -f elf32` 编译手写 ELF 结构的汇编代码后报错。问题出在 nasm 默认生成的是中间目标文件，而精简 ELF 使用`-f bin` 生成扁平二进制格式。

3.2 心得体会

经过本次“手搓最小 ELF 可执行文件”实验，我首先深入理解了 ELF 文件格式：从一开始默认使用 GCC 编译出动辄十几 KB 的可执行文件，到手工用汇编构造完整的 ELF 头和程序头，去除 section headers、符号表、重定位表等不必要的内容，我切身感受到了 ELF 文件各字段的含义与作用；我也通过 readelf、hexdump 等工具对比分析，验证了各段在文件与内存中的映射关系，真正理解了虚拟基址与文件偏移等链接相关知识。其次，我还了解到了极限压缩可执行文件的多种手段：从调用 gcc 的`-Os`、`-s` 等选项，到使用 `ld` 链接器的`-nostdlib`，到手写 ELF 文件并且通过 `truncate` 命令删除多余字节，最终将可执行文件压缩到 45B。