

《计算机系统》

LinkLab 实验报告

班级: XXX

学号: XXX

姓名: Smile_Laughter

目录

实验项目	3
项目名称	3
实验目的	3
实验资源	3
实验任务	4
Step 1: C 程序基础分析	4
Step 2: 汇编层优化	4
Step 3: 链接层优化	5
Step 4: 手工构造 ELF 文件	10
Step 5: 进一步压缩 ELF 文件	12
Step 6: 最后的优化	14
总结	19
实验中出现的問題	19
心得体会	19

实验项目

项目名称

LinkLab

实验目的

- 尝试构造尽可能小的可执行文件，且该文件的返回值为学号的最后两位数字

实验资源

- 操作系统：Ubuntu 20.04 (32 位环境)
- 工具链：gcc 9.4.0, nasm 2.14.02, ld 2.34
- 分析工具：readelf 2.34, objdump 2.34
- 开发环境：Vscode, zsh

实验任务

Step 1: C 程序基础分析

1.1 初始文件尺寸测量

```
int main(){  
    return 16;  
}
```

编译命令与结果:

```
$ gcc -m32 1.c
```

```
$ ls -l a.out
```

使用 `ls -l` 指令输出的 `16464` 是文件的字节数

```
-rwxr-xr-x 1 smile_laughter smile_laughter 16464 Apr 21 11:47 a.out
```

发现: 默认编译生成 17KB 可执行文件, 包含 ELF 头部、程序头表、.text 段、.data 段和调试信息

1.2 源码简化可能性分析

- 函数签名: 无法省略 main 函数声明
- 返回值: return 语句不可删除, 否则无法设置程序的返回值

1.3 GCC 编译器优化测试

使用 `O2` 优化

```
$ gcc -m32 -O2 1.c
```

```
$ ls -l a.out
```

```
-rwxr-xr-x 1 smile_laughter smile_laughter 15512 Apr 21 11:55 a.out
```

优化结果: 使用-O2 优化后, 文件体积从 16464 字节缩小到 15512 字节

Step 2: 汇编层优化

2.1 使用汇编模板

`BITS 32` ; 指定生成 32 位代码

`GLOBAL main` ; 声明 main 为全局符号

`SECTION .text` ; 定义代码段

```
main:
    mov eax, 16
    ret
```

编译运行指令:

```
nasm -f elf32 1.s
gcc -m32 -Wall -s 1.o
ls -l a.out
-rwxr-xr-x 1 smile_laughter smile_laughter 13656 Apr 21 12:12 a.out
```

Step 3: 链接层优化

3.1 ELF 文件结构分析

使用 readelf 查看 Program Header 结构:

```
$ readelf -l a.out
```

Elf file **type** is DYN (Shared object file)

Entry point 0x1060

There are 11 program headers, starting at offset 52

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg
↪ Align						
PHDR	0x000034	0x00000034	0x00000034	0x00160	0x00160	R 0x4
INTERP	0x000194	0x00000194	0x00000194	0x00013	0x00013	R 0x1
[Requesting program interpreter: /lib/ld-linux.so.2]						
LOAD	0x000000	0x00000000	0x00000000	0x00398	0x00398	R
↪ 0x1000						
LOAD	0x001000	0x00001000	0x00001000	0x00244	0x00244	R E
↪ 0x1000						
LOAD	0x002000	0x00002000	0x00002000	0x00120	0x00120	R
↪ 0x1000						
LOAD	0x002edc	0x00003edc	0x00003edc	0x0012c	0x00130	RW
↪ 0x1000						
DYNAMIC	0x002ee4	0x00003ee4	0x00003ee4	0x000f8	0x000f8	RW 0x4

NOTE	0x0001a8	0x000001a8	0x000001a8	0x00044	0x00044	R	0x4
GNU_EH_FRAME	0x002008	0x00002008	0x00002008	0x0003c	0x0003c	R	0x4
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RWE	0x10
GNU_RELRO	0x002edc	0x00003edc	0x00003edc	0x00124	0x00124	R	0x1

Section to Segment mapping:

Segment Sections...

```

00
01      .interp
02      .interp .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym
    ↪ .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt
03      .init .plt .plt.got .text .fini
04      .rodata .eh_frame_hdr .eh_frame
05      .init_array .fini_array .dynamic .got .data .bss
06      .dynamic
07      .note.gnu.build-id .note.ABI-tag
08      .eh_frame_hdr
09
10      .init_array .fini_array .dynamic .got

```

发现如下:

- ELF 中包含.interp 段（动态链接器路径）、DYNAMIC 段（动态链接信息）等

3.2 标准库依赖消除

报错示例:

```

$ gcc -m32 -nostdlib 1.o
/usr/bin/ld: warning: cannot find entry symbol _start;
defaulting to 0000000000001000

```

报错分析:

- `__start` 函数是程序的真正入口点。`__start` 在完成初始化后，才会调用 `main`。
- `__start` 函数缺失导致链接器无法找到入口点，导致警告

修改汇编代码，将 `main` 函数改为 `__start` 函数:

BITS 32 ; 指定生成 32 位代码

GLOBAL _start ; 声明 _start 为全局符号

SECTION .text ; 定义代码段

_start:

mov eax , 16

ret

编译并执行:

```
$ nasm -f elf32 1.s
```

```
$ gcc -m32 -nostdlib 1.o
```

```
$ ./a.out
```

```
[1] 5245 segmentation fault (core dumped) ./a.out
```

3.3 段错误原因分析及解决方案

- 堆栈状态: ESP 指令初始指向 argc, 而不是返回地址
- 段错误根源: ret 指令尝试从 [esp] 获取返回地址执行
- 正确退出流程:
 1. 设置 EAX=1 (系统调用号)
 2. 设置 EBX= 退出码
 3. 执行 int 0x80, 使用系统调用来退出程序

汇编代码示例:

BITS 32 ; 指定生成 32 位代码

GLOBAL _start ; 声明 main 为全局符号

SECTION .text ; 定义代码段

_start:

mov eax , 1

mov ebx , 16

int 0x80

再次编译运行, 查看生成的可执行文件的 Program Header:

```
$ readelf -l a.out
```

Elf file type is DYN (Shared object file)

Entry point 0x1000

There are 9 program headers, starting at offset 52

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x00000034	0x00000034	0x00120	0x00120	R	0x4
INTERP	0x000154	0x00000154	0x00000154	0x00013	0x00013	R	0x1
[Requesting program interpreter: /lib/ld-linux.so.2]							
LOAD	0x000000	0x00000000	0x00000000	0x001b5	0x001b5	R	0x1000
LOAD	0x001000	0x00001000	0x00001000	0x0000c	0x0000c	R E	0x1000
LOAD	0x002000	0x00002000	0x00002000	0x00000	0x00000	R	0x1000
LOAD	0x002f90	0x00002f90	0x00002f90	0x00070	0x00070	RW	0x1000
DYNAMIC	0x002f90	0x00002f90	0x00002f90	0x00070	0x00070	RW	0x4
NOTE	0x000168	0x00000168	0x00000168	0x00024	0x00024	R	0x4
GNU_RELRO	0x002f90	0x00002f90	0x00002f90	0x00070	0x00070	R	0x1

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

发现仍然包含链接相关信息:

- 包含.interp 段 (动态链接器路径)
- 包含.dynsym/.dynstr 段 (动态符号表)

原因分析:

- gcc 在链接时会增添一些额外信息

3.4 尝试手动链接

编译、链接并运行：

```
$ nasm -f elf32 1.s
$ ld -m elf_i386 1.o
$ ./a.out
-rwxr-xr-x 1 smile_laughter smile_laughter 4484 Apr 21 17:15 a.out
the return value of a.out is:
16
```

3.5 进一步缩减汇编代码

注意：这里如果不缩减汇编代码，在 Step 5 中，当前的汇编代码会由于太大（12 字节）而无法放入 ELF Header 的 `e_ident` 字段中

```
BITS 32
GLOBAL _start
SECTION .text
_start:
    xor eax, eax      ; 清零 EAX (2 字节)
    inc eax           ; EAX=1 (1 字节)
    mov bl, 16         ; 设置返回码 (2 字节)
    int 0x80           ; 触发系统调用 (2 字节)
```

优化分析：

- 原始代码机器码：`mov eax,1` (5B) + `mov ebx,16` (5B) + `int 0x80` (2B) = 12 字节
- 优化后机器码：`xor eax,eax` (2B) + `inc eax` (1B) + `mov bl,16` (2B) + `int 0x80` (2B) = 7 字节

运行结果如下：

```
-rwxr-xr-x 1 smile_laughter smile_laughter 4480 Apr 21 17:23 a.out
the return value of a.out is:
16
```

Step 4: 手工构造 ELF 文件

4.1 ELF 结构必要部分分析

- ELF Header 关键字段:

e_ident: Magic number + 平台标识 (16 字节)
e_type: 可执行类型 (2 字节)
e_machine: x86 架构标识 (2 字节)
e_version: 文件版本 (4 字节)
e_entry: 入口地址 (4 字节)
e_phoff: 程序头表偏移 (4 字节)

- Program Header 必要属性:

p_type: LOAD 类型 (必须包含)
p_offset: 文件偏移 (需与虚拟地址对齐)
p_flags: 执行权限 (RWE=5)
p_align: 内存页对齐 (0x1000)

- 段必要性分析:

段类型	必需性	当前程序
.text	必需	包含执行代码
.data	不必需	无全局变量
.bss	不必需	无未初始化数据
.rodata	不必需	无常量数据

4.2 使用模板构造 ELF 文件

BITS 32

org 0x08048000

ehdr: ; *Elf32_Ehdr*

db 0x7F, "ELF", 1, 1, 1, 0 ; *e_ident*

times 8 **db** 0

dw 2 ; *e_type (EXEC)*

dw 3 ; *e_machine (x86)*

dd 1 ; *e_version*

dd _start ; *e_entry*

dd phdr - \$\$; *e_phoff*

```
    dd 0          ; e_shoff
    dd 0          ; e_flags
    dw ehdrsize   ; e_ehsize
    dw phdrsize   ; e_phentsize
    dw 1          ; e_phnum
    dw 0          ; e_shentsize
    dw 0          ; e_shnum
    dw 0          ; e_shstrndx

ehdrsize equ $ - ehdr

phdr: ; Elf32_Phdr
    dd 1          ; p_type (LOAD)
    dd 0          ; p_offset
    dd $$         ; p_vaddr
    dd $$         ; p_paddr
    dd filesize   ; p_filesz
    dd filesize   ; p_memsz
    dd 5          ; p_flags (RWE)
    dd 0x1000     ; p_align

phdrsize equ $ - phdr

_start:
    xor eax, eax    ; 2 字节
    inc eax         ; 1 字节
    mov bl, 16      ; 2 字节
    int 0x80        ; 2 字节

filesize equ $ - $$
```

4.3 编译运行

- 编译运行命令:

```
$ nasm 1.s -f bin -g -o a.out
```

```
$ chmod +x a.out
$ ls -l a.out
$ echo "the return value of a.out is:"
$ ./a.out ; echo $?
```

- 运行结果:

```
-rwxr-xr-x 1 smile_laughter smile_laughter 91 Apr 21 17:34 a.out
the return value of a.out is:
16
```

- 发现在去掉多余的段后，文件体积从 4480 字节缩小到 91 字节

Step 5: 进一步压缩 ELF 文件

5.1 文件尺寸现状分析

通过 Step 4 的 ELF 手工构造，当前可执行文件尺寸为：

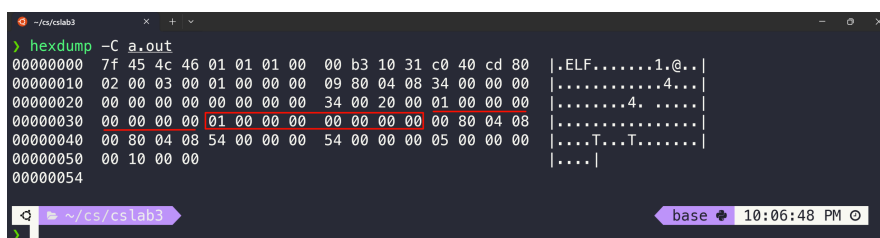
```
-rwxr-xr-x 1 user user 91 Apr 30 09:00 a.out
```

分析 **ELF Header** 发现：在 ELF 头中，有一些实际并没作用的内容，例如在标识符末尾填充了 9 个字节的 0，而我们的程序只需要 7 字节的代码，因此我们的程序代码可以放在 ELF 头部的填充部分

5.2 结构重叠可行性验证

通过分析 ELF 文件结构，发现以下优化空间：

- 因为 32 位系统中，ELF header 固定大小为 52 个字节，所以在 0x30 是 ELF header 的第 49 个字节，因此标有红色下划线的是 ELF header 的后 8 个字节。
- 我们知道，Program header 是紧跟在 ELF header 后面的，因此用矩形框出来的 8 个字节就是 Program header 的前 8 个字节。
- 发现二者确实一模一样



```
> hexdump -C a.out
00000000  7f 45 4c 46 01 01 01 00  00 b3 10 31 c0 40 cd 80  |.ELF.....1@..|
00000010  02 00 03 00 01 00 00 00  09 80 04 08 34 00 00 00  |.....4...|
00000020  00 00 00 00 00 00 00 00  34 00 20 00 01 00 00 00  |.....4. ....|
00000030  00 00 00 00 01 00 00 00  00 00 00 00 00 80 04 08  |.....T.....|
00000040  00 80 04 08 54 00 00 00  54 00 00 00 05 00 00 00  |....T.....|
00000050  00 10 00 00
00000054
```

5.3 结构缝合实践

修改汇编代码实现字段重叠:

```
BITS 32
org 0x08048000
ehdr:
db 0x7F, "ELF" ; e_ident
db 1, 1, 1, 0, 0
_start: mov bl, 16
xor eax, eax
inc eax
int 0x80
dw 2 ; e_type
dw 3 ; e_machine
dd 1 ; e_version
dd _start ; e_entry
dd phdr - $$ ; e_phoff
dd 0 ; e_shoff
dd 0 ; e_flags
dw ehdrsize ; e_ehsize
dw phdrsize ; e_phentsize
phdr: dd 1 ; e_phnum ; p_type
; e_shentsize
dd 0 ; e_shnum ; p_offset
; e_shstrndx
ehdrsize equ $ - ehdr
dd $$ ; p_vaddr
dd $$ ; p_paddr
dd filesize ; p_filesz
dd filesize ; p_memsz
dd 5 ; p_flags
dd 0x1000 ; p_align
phdrsize equ $ - phdr
filesize equ $ - $$
```

编译运行结果:

```
-rwxr-xr-x 1 smile_laughter smile_laughter 76 Apr 21 19:11 a.out
```

the `return` value of `a.out` is:

```
16
```

Step 6: 最后的优化

注：指南在 Step 6 中给出了两个 ELF 的模板，这里只分析了第二个 ELF 模板，因为第二个模板体积更小，也是我们要的最终的文件

6.1 ELF 文件结构解析

偏移	字段名	大小	作用说明
0x00	e_ident	16	平台标识 + 魔数
0x10	e_type	2	文件类型 (EXEC=2)
0x12	e_machine	2	目标架构 (x86=3)
0x14	e_version	4	ELF 版本 (可以为其它的值)
0x18	e_entry	4	入口地址 (代码起始点)
0x1C	e_phoff	4	程序头表偏移 (决定程序头的位置)
0x20	e_shoff	4	节头表偏移 (可以为其它的值)
0x24	e_flags	4	处理器特定标志 (可以为其它的值)
0x28	e_ehsize	2	ELF 头大小 (固定 52 字节)
0x2A	e_phentsize	2	程序头条目大小 (固定 32 字节)
0x2C	e_phnum	2	程序头数量 (在我们的 ELF 文件中为 1)
0x2E	e_shentsize	2	节头条目大小 (可置零)
0x30	e_shnum	2	节头数量 (可置零)
0x32	e_shstrndx	2	节头字符串表索引 (可置零)

表 1: ELF Header 字段解析 (32 位架构)

偏移	字段名	大小	作用
0x00	p_type	4	段类型 (如 LOAD=1, DYNAMIC=2, INTERP=3)
0x04	p_offset	4	段在文件中的偏移量 (字节)
0x08	p_vaddr	4	段在内存中的虚拟地址
0x0C	p_paddr	4	段在内存中的物理地址 (通常忽略)
0x10	p_filesz	4	段在文件中的大小 (字节)
0x14	p_memsz	4	段在内存中的大小 (字节)
0x18	p_flags	4	段权限 (低三位依次是 RWX)
0x1C	p_align	4	段对齐要求

表 2: ELF Program Header 字段结构 (32 位格式)

6.2 最终 ELF 文件及代码详细解析

`BITS 32`

`; origin : 定义段基址为 0x00010000`

```
org 0x00010000
; db : define byte (1 bytes)
; dw : define word (2 bytes)
; dd : define double word (4 bytes)
db 0x7F, "ELF"          ; e_ident[0-3]
dd 1                    ; p_type (LOAD)
dd 0                    ; p_offset
; $$: NASM 特殊符号, 表示当前段的起始地址
; 下一行的地址是 Program Header 的起始地址, 这是由 e_phoff 字段决定的
dd $$                  ; p_vaddr
dw 2                    ; e_type (EXEC)
dw 3                    ; e_machine (x86)
; _start: 是下面出现的标签, 代表代码段的起始地址
; _start 的绝对地址 = org 基址 + 标签偏移 (例如 0x08048000 + 0x1C)
dd _start              ; e_version (1) + p_filesz
dd _start              ; e_entry (代码入口) + p_memsz
; 汇编器根据 e_phoff(elf_program_header offset)
; 计算出 Program Header 在 ELF 文件中的起始位置
dd 4                    ; e_phoff (程序头偏移) + p_flags (R=4)
_start:
mov bl, 16              ; 代码段起始、p_align 的低两个字节
xor eax, eax           ; e_shoff 字段、p_align 的高两个字节
inc eax                ; 使用 e_flags[0]
int 0x80               ; 系统调用
db 0                   ; 填充对齐字节
dw 0x34                ; e_ehsize (52 字节)
dw 0x20                ; e_phentsize (32 字节)
dw 1                   ; e_phnum (程序头数量)
dw 0                   ; e_shentsize (可忽略)
dw 0                   ; e_shnum (可忽略)
dw 0                   ; e_shstrndx (可忽略)
; filesize: 定义的一个标签, 表示文件大小
; equ:equal
; $:NASM 特殊符号, 表示 $ 符号所在这一行的地址
; 其实这一行定义的标签 filesize 在这个 ELF 文件中并没有用到
```

```
filesize equ $ - $$
```

6.3 重叠字段分析

指南给出的最终的 ELF 文件，将 Program Header 的起始地址设置为了 ELF Header 的第 4 字节处

ELF Header 字段	Program Header 字段	占用字节数
e_ident[4-15] (偏移 4-15, 12 字节)	p_type (4 字节)	4
	p_offset (4 字节)	4
	p_vaddr (4 字节)	4
e_type (2 字节) + e_machine (2 字节)	p_paddr (4 字节)	4
e_entry (4 字节)	p_memsz (4 字节)	4
e_version (4 字节)	p_filesz (4 字节)	4
e_phoff (4 字节)	p_flags (4 字节)	4
p_align (4 字节) 与代码指令重叠		
代码段指令 (mov bl,16 + xor eax,eax)	p_align (4 字节)	4

表 3: ELF 文件头与程序头字段重叠关系图示

具体分析如下：

- e_ident 字段（偏移量 4-15 的 12 字节）存放以下三个字段：
 - p_type
 - p_offset
 - p_vaddr
- e_type + e_machine 与 p_paddr 字段重合：
 - 约束条件：e_type=2 (EXEC) 且 e_machine=3 (x86)
 - 导致 p_paddr 被强制设为非常规值 0x00030002，但合法：
 - * 物理地址由操作系统管理，加载器会忽略非常规物理地址
 - * 虚拟地址 (VirtAddr) 仍可正常工作
- e_entry 与 p_memsz 重合，e_version 与 p_filesz 重合：
 - 强制要求 p_memsz = e_entry (因为 ELF 文件中程序的入口地址 (_start) 是固定的)
 - p_filesz 处理策略：
 - * 因 e_version 字段不重要，可自由设置 p_filesz
 - * 约束条件：p_memsz 不可大于 p_filesz

- * 最终设置 `p_filesz = p_memsz = e_entry`

- `p_flags` 与 `e_phoff` 重合:

- 我们使用的 ELF 重叠策略决定了 `e_phoff` 的值为 4, 所以 `p_flags` 的值也被设置为 4

- 权限位分析:

- * 原始权限: `RWE = 5` (`0b00000101`, 可读 + 可执行)

- * 调整后: `RWE = 4` (`0b00000100`, 仅可读)

- 可行性依据:

- * Linux 允许可读段隐式可执行 (与安全策略相关)

- * 写权限 (W) 必须显式声明, 不可隐式获得

- `p_align` 字段和我们代码的前两句重合了, 这使得 `p_align` 也被设置成为一个比较奇怪的值 (`0xc03110b3`), 但是似乎并不影响程序的运行。

6.4 零字节截断 ~ 最后的伎俩

使用 `hexdump` 工具查看文件内容:

```
$ ls -l a.out
```

```
-rwxr-xr-x 1 smile_laughter smile_laughter 52 Apr 21 20:41 a.out
```

```
$ hexdump -C a.out
```

```
00000000  7f 45 4c 46 01 00 00 00 00 00 00 00 00 01 00 |.ELF.....|
00000010  02 00 03 00 20 00 01 00 20 00 01 00 04 00 00 00 |.... ... ..|
00000020  b3 10 31 c0 40 cd 80 00 34 00 20 00 01 00 00 00 |..1.@...4. ....|
00000030  00 00 00 00                                     |....|
00000034
```

使用 `truncate` 工具截断文件末尾的 0:

```
$ truncate -s -7 a.out
```

```
$ ls -l a.out
```

```
-rwxr-xr-x 1 smile_laughter smile_laughter 45 Apr 21 20:41 a.out
```

```
$ ./a.out ; echo $?
```

16

优化阶段	大小	缩减率
Step4 初始构造	91 字节	-
Step5 结构重叠	76 字节	16.48%
Step6 终极优化	45 字节	40.79%

6.5 最终优化成果

- 文件尺寸对比:
- ELF 结构验证:

```
$ readelf -l a.out
```

```
readelf: Warning: possibly corrupt ELF file header - it has a non-zero section head
```

```
Elf file type is EXEC (Executable file)
```

```
Entry point 0x10020
```

```
There is 1 program header, starting at offset 4
```

```
Program Headers:
```

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x000000	0x00010000	0x00030002	0x10020	0x10020	R	0xc03110b3

- 机器码布局:

```
$ hexdump -C a.out
```

```
00000000  7f 45 4c 46 01 00 00 00  00 00 00 00 00 00 01 00  |.ELF.....|
00000010  02 00 03 00 20 00 01 00  20 00 01 00 04 00 00 00  |.... ... ..|
00000020  b3 10 31 c0 40 cd 80 00  34 00 20 00 01          |..1.@...4. ...|
0000002d
```

总结

实验中出现的問題

- 汇编命令的变化: `nasm -f` 的参数从 `elf32` 改为 `bin`
- 段错误问题: 使用 `-nostdlib` 后未正确处理程序退出, 通过 `int 0x80` 系统调用直接退出进程解决
- 入口点冲突: 从 `main` 改为 `_start` 后需重新定义程序入口逻辑, 通过寄存器精确控制返回码
- 字段重叠冲突: ELF 头与程序头字段重叠时, 需确保关键字段的语义兼容性

心得体会

- **ELF 结构认知:** 深入理解 ELF 头的 `e_ident` 魔数、`e_entry` 入口点、程序头的加载属性等关键字段
- **系统调用机制:** 掌握通过 `int 0x80` 进行 Linux 系统调用的寄存器传参规范
- **空间优化维度:** 发现代码优化、符号表剥离、节区合并、字段复用等多级优化空间
- **规范边界探索:** 认识到 ELF 规范中非必需字段的可缺省性及加载器的容错机制
- **工具链协作:** 熟练运用 `nasm`、`ld`、`readelf`、`objdump` 等工具进行交叉验证

通过本实验, 最终将返回学号末两位的 C 程序从初始 2KB+ 优化至 45 字节, 学会了以下内容:

- 去除所有标准库和启动代码依赖
- 合并 ELF 头和程序头字段存储空间
- 利用系统调用直接返回结果
- 精确控制代码段与头部结构重叠