《计算机系统实验三》

班级：信安2302班

学号：202308060227

姓名：石云博

目录

[一. 实验描述 2](#_Toc196328324)

[二. Step1 2](#_Toc196328325)

[三. Step2 2](#_Toc196328326)

[四. Step3 2](#_Toc196328327)

[五. Step4 2](#_Toc196328328)

[六. Step5 3](#_Toc196328329)

[七. Step6 3](#_Toc196328330)

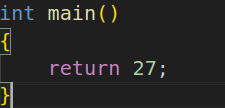
# 实验描述

作为C语言的初学者，第一个编写的 C 代码一般是如下所示的“Hello World”： 但这区区四行 C 代码，在 X86架构下Ubuntu 中使用 gcc 形成的可执行文件大 小可能超过 7KB ！ 实验目标：请尝试分析 TA 到底增加了什么内容导致可执行文件的大小产生这样的膨胀？基于前述分析，如果一段极其简单的代码如下，TA仅仅返回一个数字XX（请使用你学号的最后两位取代下面代码中的 “XX”）： int main ( ) { return XX; } 你可以尽你所能（后文我们会为你提供指导参考）获取功能没有任何变化的更小 （字节数）可执行文件，你能获得多小的可执行文件？

# Step1

前述仅仅返回你学号（最后两位）的代码，ta的可执行文件有多大？如果要减小可执行代码，其C源码层面还能进一步简化吗？如果不能，为什么？求助于gcc编译器呢？ta有优化选项啊？请尝试，是否缩小了可执行文件？

1 代码如下：



使用gcc将其编译成可执行文件，然后我们查看该可执行文件的属性：



可以看到此时的文件大小为15.8kb

对于问题“如果要减小可执行代码，其C源码层面还能进一步简化吗？如果不能，为什么？”，经过网上资料的查询后，得出如下结论：

这段代码，在符合现代 C 标准（C99 及以后）的前提下，已经在源代码层面达到了最简形式。不能进一步删除或合并任何部分，因为：

1 main 的签名必须返回 int 并且不能省略（C99 及以后不再允许隐式 int）​。

2 非零返回值必须显式写出 return 27；只有当返回 0 时，才可依靠 C99 中对 main 的特殊隐式返回规则（隐式 return 0; 仅适用于返回值为 0 的情况）​。

3 可执行文件体积 主要由运行时初始化代码（CRT，包含启动例程、符号表、重定位信息等）以及链接器选项决定，单纯修改极简的 main 源码无法再减小这些部分​。

更具体的原理如下：

1. C 标准对 main 的定义与限制

返回类型必须为 int：ISO C99 及之后的标准明确移除了对函数隐式 int 的支持，所有函数都必须显式声明返回类型；对此 C 标准的设计说明中有明确描述：“在 C99 标准中，如果声明缺少类型说明符，将不再假定 int”​。函数签名固定：标准要求 main 函数“必须具有 int main(void) 或 int main(int, char \*[]) 之一的形式”。

由于这些语法层面的硬性规定，源代码无法省略 int、更改参数列表，或者在现代编译环境中省略函数体的花括号。

2. 隐式返回仅适用于 return 0;

隐式 return 0;：从 C99 开始，若 main 的执行流到达末尾而无显式 return，编译器会自动插入等同于 return 0; 的行为；但这一特殊规则仅针对返回值为 0 的情况。若需返回非零值（例如 27），必须显式编写 return 27;，否则行为未定义或返回值不可靠​。其他函数不适用：该隐式返回规则不适用一般函数，亦不会生成非零返回码。

3. 可执行文件大小的真正影响因素

CRT 启动代码：在大多数平台上，编译器和链接器会自动引入一段“C 运行时启动例程”（通常来自 crt0.o、crt1.o 等），用于设置环境、调用全局构造函数、传递参数，然后转到 main，并在返回后调用 exit。这部分汇编代码往往占用大量字节​。链接和优化选项：要减小最终可执行文件大小，需要在编译和链接时使用诸如 -Os（优化代码尺寸）、-s（去除符号表）、strip（移除调试信息）等选项；在源代码层面无法再做精简。

对于下一个问题“求助于gcc编译器呢？ta有否优化选项？请尝试，是否缩小了可执行文件？”，我们查到可以使用如下的优化指令：

-Os：启用“优化尺寸”（optimize for size），GCC 会选择一系列优化选项来最大限度地减小代码体积，而不仅仅是限制在 -O2 或 -O3 的子集​。

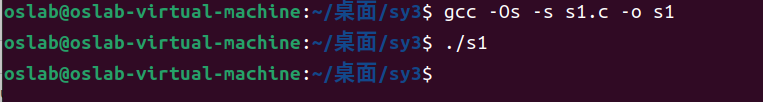
-s：通过链接器移除符号表、调试信息等多余段（等同于 strip 工具），这是减小可执行文件体积最直接的手段​。

其他常用缩小体积选项（适用于对尺寸有极端要求的场景）：

-fno-ident：移除 .comment 段中记录的 GCC 标识字符串​

-fno-asynchronous-unwind-tables：删除用于异常栈展开的表格，适合无异常或不需要栈回溯信息的程序​。

利用上述指令对我们的文件进行优化编译：



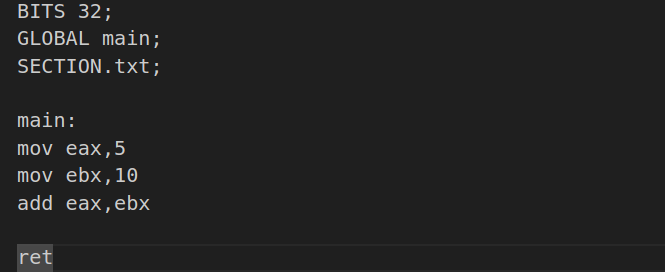
得到结果：

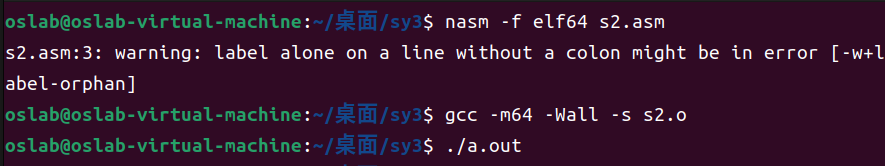


发现此时的可执行文件缩小到了14.3kb。

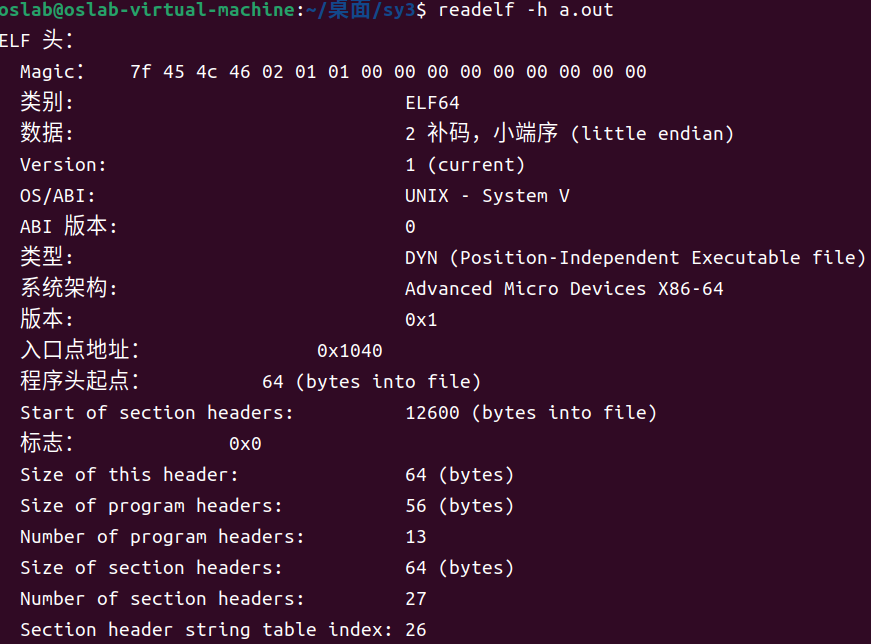
# Step2

如果Step1中的步骤都尝试过了，优化尺寸了吗？下面我们尝试一下直接开整汇编代码。编写汇编代码如下：





使用如上指令。

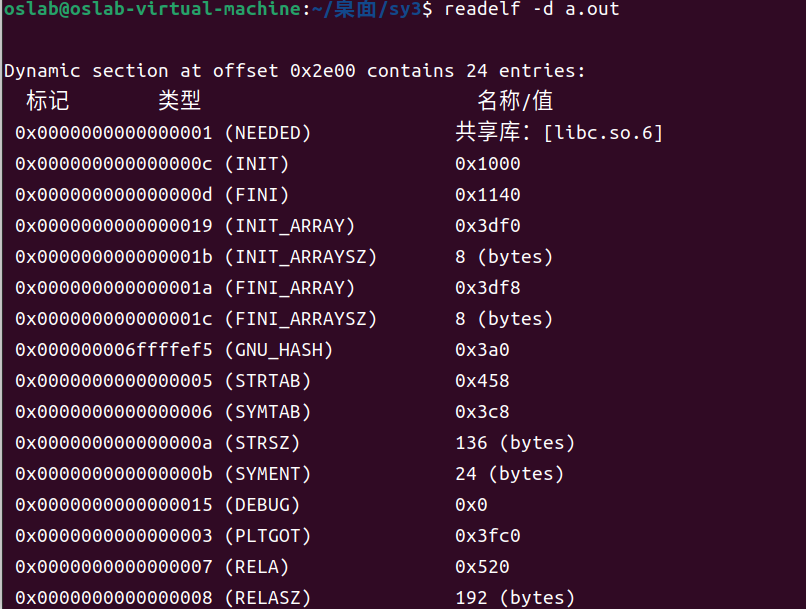


查看结果

# Step3

1你是否注意到可执行ELF文件中包含与链接有关的部分。想一想，我们并没有明显调用 外部库函数，为什么还会出现这些链接部分以及其有什么作用？

要查看其与链接有关的部分，我们可以使用指令“readelf -d”来进行查看其动态链接：



可以看到结果中确实存在链接的行为。为什么会出现链接在这里，查询资料得知：这种现象的背后，并非源自我们的代码本身，而是由GCC 默认的链接行为造成的。其中gcc的默认行为有：

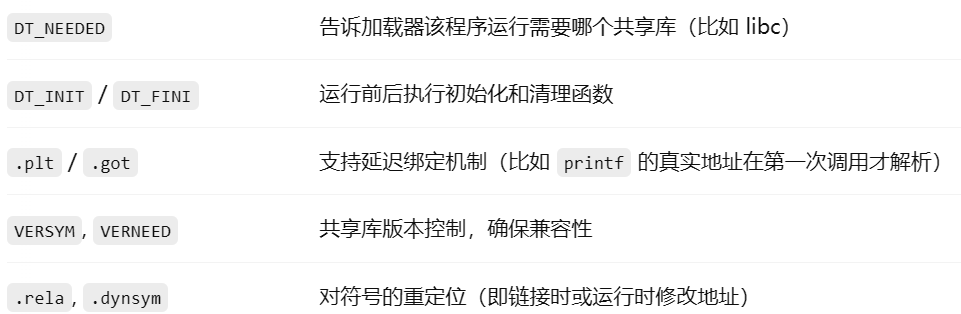
自动链接 crt1.o, crti.o, crtn.o 等启动代码（提供 \_start 入口、调用构造函数/析构函数）。

自动链接 libc.so.6，除非显式取消。

默认生成动态链接的 ELF 可执行文件，这就引入了 .dynamic 段、NEEDED 字段、GOT/PLT 表等。

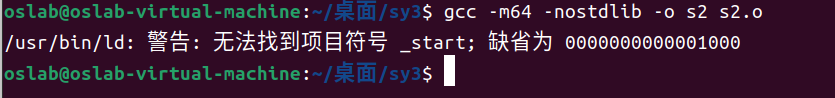
其自动链接的作用大致可以概况成构建的是完整的 Linux 用户空间 ELF 程序。

其中各链接及其具体功能如下：



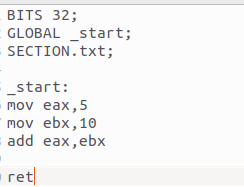
对于问题“对于一个如此简单的程序，我们是否有必要去调用外部的库函数呢？”，我们认为是没有必要进行那么多的外部库函数调用的，只需要调用最基本的就可以了。

尝试使用 gcc 指令 -nostdlib 来取消链接标准库和启动代码：



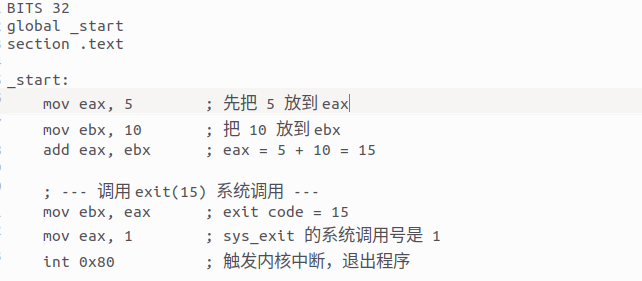
发现确实存在\_start 未定义的报错，实验文档中写了，这是由于取消链接后，无法调用外部函数引起的而这又是 程序运行的入口，所以只能由我们自己在汇编代码中进行定义，且由于此 C 程序所要 执行的功能非常简单，所以我们完全可以让任务在 \_start 中完成，这样就不需要再 定义 main 函数了。

所以我们需要修改我们的汇编代码，在其中加入入口：

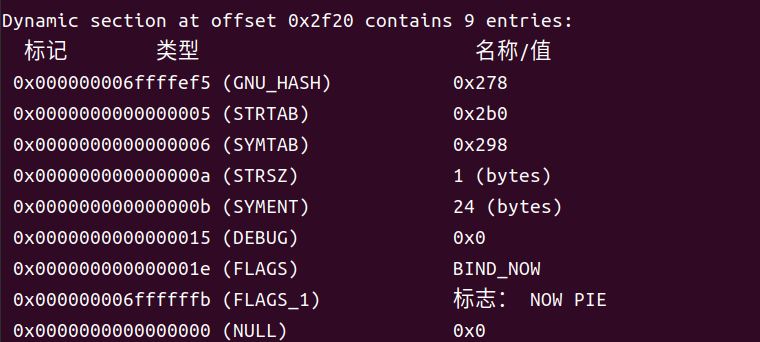


将main改成start。

看实验文档，我们还需要添加结束标志，int 0X80，以及对 eax 赋值为 1来表明进程的结束。



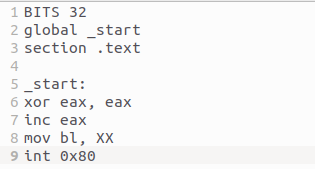
我们此时查看该可执行文件的链接部分：



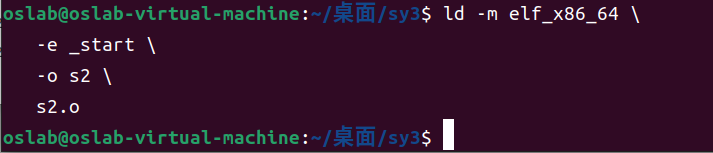
虽然其中没有了libc.so.6等部分链接，但是还保留有一些链接相关内容，我们可以在此时查看该可执行文件的大小：



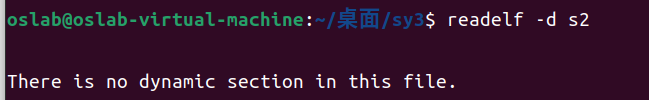
发现此时的大小来到了13.4kb，接着我们按照实验文档的要求对汇编代码进行修改：



然后抛弃GCC的自动编译，改成我们自己使用编译器：



此时我们查看该文件的链接部分：



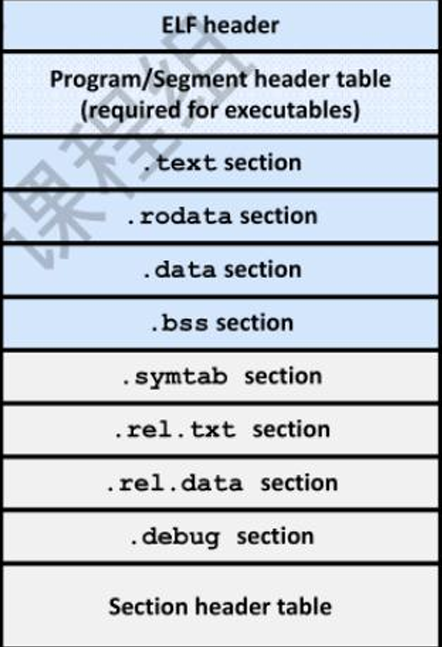
可以看到里面已经没有了动态链接。这样我们得到的可执行文件的大小：



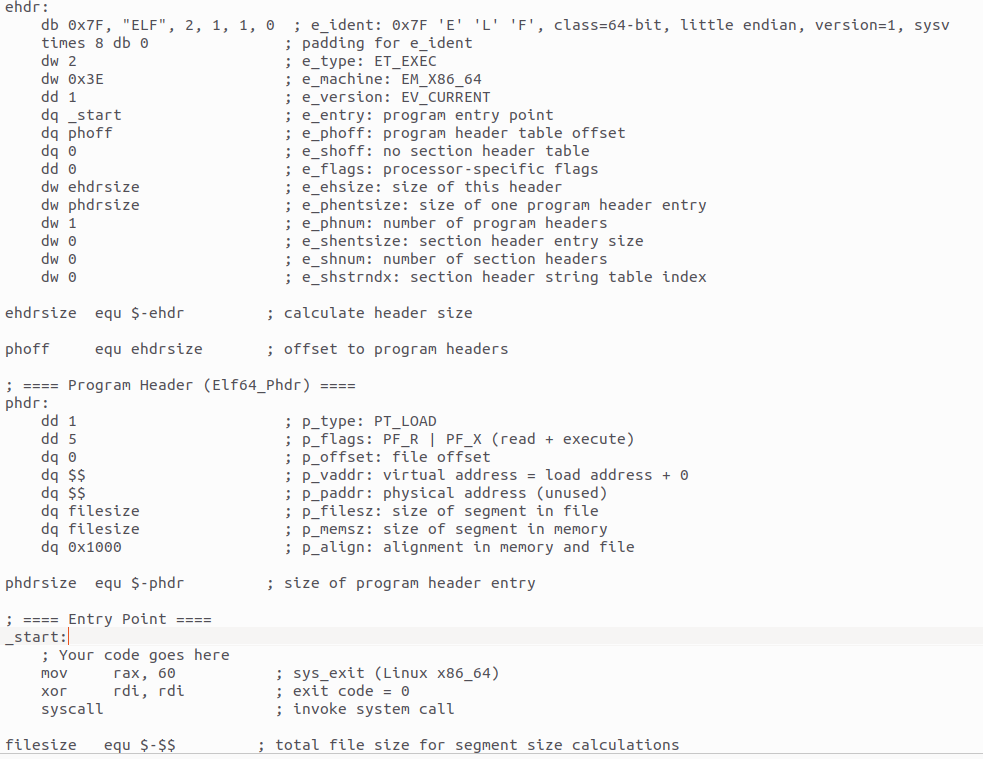
变成了4.6kb，变成了原来的三分之一左右。

# Step4

现在我们获得一个相对“简单”的可执行文件，ta有多大？如果想进一步减小ta，就 只能直接从ELF文件下手了。



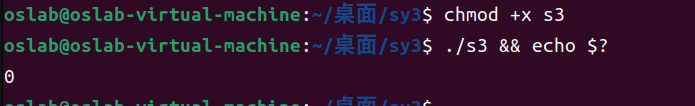
针对该elf文件的格式，我们需要自己手搓一个elf文件出来。



我们将上述的文件编译成可执行文件：



发现其只有130字节了。同时我们运行该文件查看其是否能够正常执行：

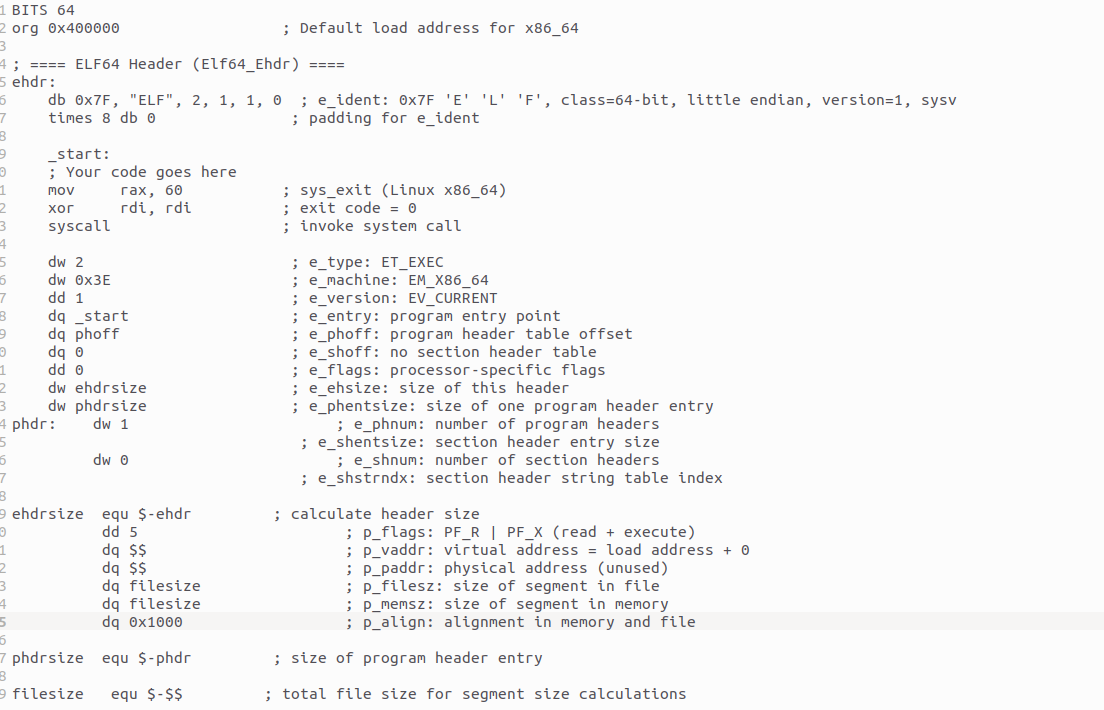


可以看到此时文件是可以正常执行的，会输出结果0.

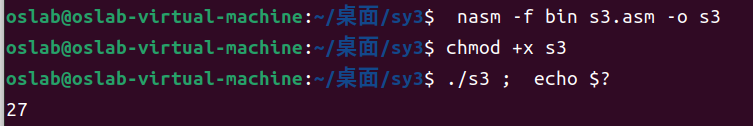
# Step5

在 ELF 头中，有一些实际并没作用的内容，例如在标识符末尾那几个滥竽充数的 0 （数一数，共有多少个字节？）。而我们的程序，数一数需要多少个字节即可完成？所以我们是不是可以将程序实现部分放入在标识符末位的填充部分上呢？

按照实验文档的提示，我们可以修改原文件为：



此时我们将其编译成可执行文件，再执行：



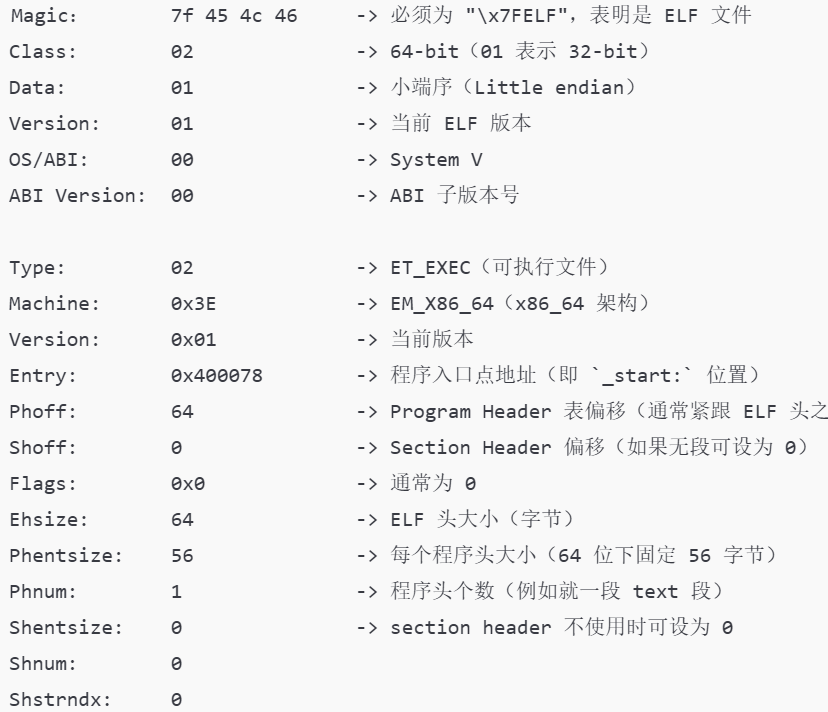
可以看到此时文件可以正常执行。

# Step6

是不是能从结构下手，更改接受的内容呢？请尝试列出所有ELF header的各部分 内容、作用及其大小！

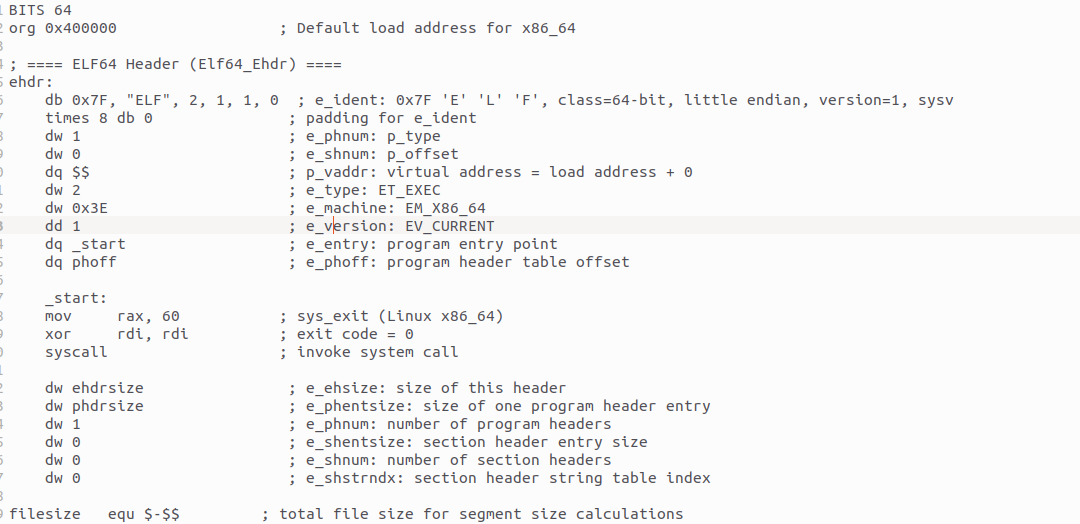


ELF header的各部分 内容、作用及其大小：



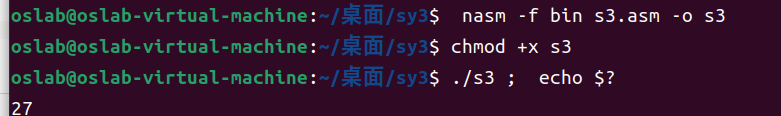
发现了吗？ELF header中的大部分必要字段都在前半部分，后半部分几乎完全可 以自由地进行修改。考虑到这一点，我们可以继续：

此时，程序头表的前 20 个字节与 ELF 头的最后 20 个字节重叠，而且还契合得 很好。重叠区域内 ELF头只有两个部分是重要的：第一个是 e\_phnum 字段，它恰 好与p\_paddr 字段重合，这是程序头表中为数不多的绝对被忽略的字段之一。另 一个是 e\_phentsize 字段，它与 p\_vaddr 字段的上半部分重合。这些为我们的 程序选择一个非标准加载地址来匹配，上半部分等于 0x0020。Program header中还有几个字段可以供我们可以进处理。注意到了嘛， p\_memsz 表 示要为内存段分配多少内存，即它至少需要与 p\_filesz 一样大，但如果它更大， 也不会有任何坏处 —— 可以“尸位素餐”。同时我们可以将 p\_flags 字段中的 可执行位设置为0，因为可读位和可执行位存在着一种微妙的共生关系（任何一个 都会暗示另一个）。基于前述事实，我们将可执行文件重新组织成：



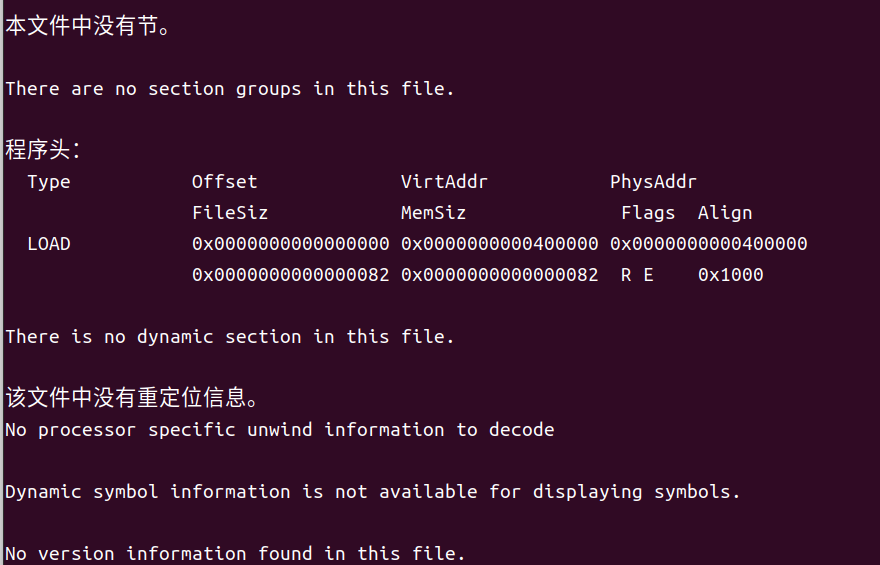
此时，我们再次将其编译执行：

可以看到最终我们得到的结果，依旧是可以执行的，并且能正常输出0.



我们查看该可执行文件的内容：



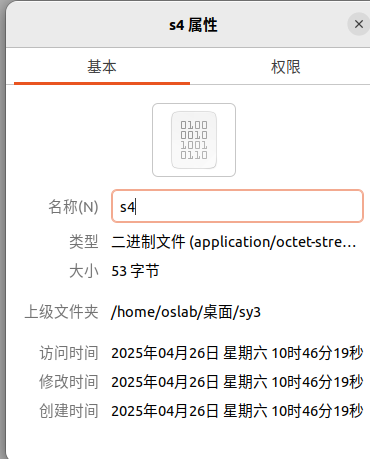


可以看到当前的文件基本上只有一个单纯的骨架了，其他任何多余的杂项都被删得干干净净了。最后我们来查看该可执行文件的大小：



可以看到是130字节。

再次将下移至 ELF头的下部，从 e\_shoff 字段开始，到 e\_flags 字段内结束。在将文件末尾的0全部删掉，再次编译，发现文件被编译成了一个二进制文件而不是直接的可执行文件，我们查看该文件的大小：



发现其只有53字节的大小。