

**计算机系统**

**大作业**

题 目 程序人生-Hello’s P2P

专 业 计算机类

学　　 号 1190200526

班　　 级 1903002

学 生 沈城有

指 导 教 师 郑贵滨

**计算机科学与技术学院**

**2021年6月**

**摘 要**

本文以Hello程序为中心，阐述了Hello程序在Linux系统的生命周期，分析了从hello.c源代码文件经过预处理、编译、汇编、链接、执行以至终止的全过程，并结合课程所学知识说明了Linux操作系统如何对Hello程序进行进程管理、存储管理和I/O管理。完整体现了我个人对计算机系统课程主体框架和内容的理解。

**关键词：**Hello程序；程序生命周期；操作系统管理；计算机系统课程

**目 录**

[第1章 概述 - 4 -](#_Toc74157311)

[1.1 Hello简介 - 4 -](#_Toc74157312)

[1.2 环境与工具 - 4 -](#_Toc74157313)

[1.3 中间结果 - 5 -](#_Toc74157314)

[1.4 本章小结 - 5 -](#_Toc74157315)

[第2章 预处理 - 6 -](#_Toc74157316)

[2.1 预处理的概念与作用 - 6 -](#_Toc74157317)

[2.2在Ubuntu下预处理的命令 - 6 -](#_Toc74157318)

[2.3 Hello的预处理结果解析 - 7 -](#_Toc74157319)

[2.4 本章小结 - 9 -](#_Toc74157320)

[第3章 编译 - 10 -](#_Toc74157321)

[3.1 编译的概念与作用 - 10 -](#_Toc74157322)

[3.2 在Ubuntu下编译的命令 - 10 -](#_Toc74157323)

[3.3 Hello的编译结果解析 - 11 -](#_Toc74157324)

[3.3.1 数据与赋值 - 11 -](#_Toc74157325)

[3.3.2 算术操作 - 12 -](#_Toc74157326)

[3.3.3 数组、指针、结构操作 - 12 -](#_Toc74157327)

[3.3.4 关系操作及控制转移 - 12 -](#_Toc74157328)

[3.3.5 函数操作 - 13 -](#_Toc74157329)

[3.4 本章小结 - 15 -](#_Toc74157330)

[第4章 汇编 - 16 -](#_Toc74157331)

[4.1 汇编的概念与作用 - 16 -](#_Toc74157332)

[4.2 在Ubuntu下汇编的命令 - 16 -](#_Toc74157333)

[4.3 可重定位目标elf格式 - 16 -](#_Toc74157334)

[4.3.1 readelf命令 - 16 -](#_Toc74157335)

[4.3.2 ELF头 - 17 -](#_Toc74157336)

[4.3.3 节头目表 - 17 -](#_Toc74157337)

[4.3.4 重定位节 - 18 -](#_Toc74157338)

[4.3.5 符号表 - 19 -](#_Toc74157339)

[4.4 Hello.o的结果解析 - 19 -](#_Toc74157340)

[4.5 本章小结 - 22 -](#_Toc74157341)

[第5章 链接 - 23 -](#_Toc74157342)

[5.1 链接的概念与作用 - 23 -](#_Toc74157343)

[5.2 在Ubuntu下链接的命令 - 23 -](#_Toc74157344)

[5.3 可执行目标文件hello的格式 - 23 -](#_Toc74157345)

[5.4 hello的虚拟地址空间 - 25 -](#_Toc74157346)

[5.5 链接的重定位过程分析 - 27 -](#_Toc74157347)

[5.6 hello的执行流程 - 32 -](#_Toc74157348)

[5.7 Hello的动态链接分析 - 32 -](#_Toc74157349)

[5.8 本章小结 - 32 -](#_Toc74157350)

[第6章 hello进程管理 - 33 -](#_Toc74157351)

[6.1 进程的概念与作用 - 33 -](#_Toc74157352)

[6.2 简述壳Shell-bash的作用与处理流程 - 33 -](#_Toc74157353)

[6.3 Hello的fork进程创建过程 - 33 -](#_Toc74157354)

[6.4 Hello的execve过程 - 33 -](#_Toc74157355)

[6.5 Hello的进程执行 - 33 -](#_Toc74157356)

[6.6 hello的异常与信号处理 - 33 -](#_Toc74157357)

[6.7本章小结 - 33 -](#_Toc74157358)

[第7章 hello的存储管理 - 34 -](#_Toc74157359)

[7.1 hello的存储器地址空间 - 34 -](#_Toc74157360)

[7.2 Intel逻辑地址到线性地址的变换-段式管理 - 34 -](#_Toc74157361)

[7.3 Hello的线性地址到物理地址的变换-页式管理 - 34 -](#_Toc74157362)

[7.4 TLB与四级页表支持下的VA到PA的变换 - 34 -](#_Toc74157363)

[7.5 三级Cache支持下的物理内存访问 - 34 -](#_Toc74157364)

[7.6 hello进程fork时的内存映射 - 34 -](#_Toc74157365)

[7.7 hello进程execve时的内存映射 - 34 -](#_Toc74157366)

[7.8 缺页故障与缺页中断处理 - 34 -](#_Toc74157367)

[7.9动态存储分配管理 - 34 -](#_Toc74157368)

[7.10本章小结 - 35 -](#_Toc74157369)

[第8章 hello的IO管理 - 36 -](#_Toc74157370)

[8.1 Linux的IO设备管理方法 - 36 -](#_Toc74157371)

[8.2 简述Unix IO接口及其函数 - 36 -](#_Toc74157372)

[8.3 printf的实现分析 - 36 -](#_Toc74157373)

[8.4 getchar的实现分析 - 36 -](#_Toc74157374)

[8.5本章小结 - 36 -](#_Toc74157375)

[结论 - 37 -](#_Toc74157376)

[附件 - 38 -](#_Toc74157377)

[参考文献 - 39 -](#_Toc74157378)

# 第1章 概述

## 1.1 Hello简介

1. **P2P**

P2P是指from program to process，即从程序到进程。在Linux系统中，源代码文件hello.c(program)先经cpp预处理生成文本文件hello.i，hello.i经cc1编译生成汇编文件hello.s，hello.s经ld链接生成可执行程序文件hello。最后在shell中键入命令(./hello)后，操作系统(OS)的进程管理为其fork创建子进程 (process)。

1. **020**

020是指from zero to zero。Hello程序执行前，不占用内存空间（第一个0）。P2P过程后，子进程首先调用execve，依次进行虚拟内存映射、物理内存载入；随后进入主函数执行程序代码，程序调用各种系统函数实现屏幕输出信息等功能；最终程序结束，shell父进程回收此子进程，其相关的所有内存中的状态信息和数据结构被清除（第二个0）。以上即为020的全部过程。

## 1.2 环境与工具

1. **硬件环境**

CPU: Intel® Core™ i7-8565U CPU @ 1.80GHz

RAM: 16.00GB

1. **软件环境**

Windows10 64位

Oracle VM VirtualBox 6.1.16 r140961 (Qt5.6.2)

Ubuntu 20.04.1

1. **开发与调试工具**

Visual Studio Code

cpp（预处理器）

gcc（编译器）

as（汇编器）

ld（链接器）

GNU readelf

GNU gdb

EDB等

## 1.3 中间结果

|  |  |  |
| --- | --- | --- |
| 文件名称 | 说明 | 对应本文章节 |
| hello.i | hello.c经预处理得到的ASCII文本文件 | 第2章 |
| hello.s | hello.i经编译得到的汇编代码ASCII文本文件 | 第3章 |
| hello.o | hello.s经汇编得到的可重定位目标文件 | 第4章 |
| hello\_elf.txt | hello.o经readelf分析得到的文本文件 | 第4章 |
| hello\_dis.txt | hello.o经objdump反汇编得到的文本文件 | 第4章 |
| hello | hello.o经链接得到的可执行文件 | 第5章 |
| hello1\_elf.txt | hello经readelf分析得到的文本文件 | 第5章 |
| hello1\_dis.txt | hello经objdump反汇编得到的文本文件 | 第5章 |

## 1.4 本章小结

本章首先以Hello程序为例，简要解释了P2P、O2O的概念，随后提供了我使用的环境及工具的相关信息，最后以表格形式介绍了完成本论文过程中用到的所有中间文件。

# 第2章 预处理

## 2.1 预处理的概念与作用

1. **预处理的概念**

预处理是在编译前进行的处理，此过程中会扫描源代码，检查包含预处理指令的语句和宏定义并进行相应的替换，还会删除程序中的注释和多余的空白字符等，最终产生调整后的源代码提供给编译器。

1. **预处理的作用**

预处理的作用主要可分为以下三部分：

1. 宏展开：预处理程序中的“#define”标识符文本，用实际值（可以是字符串、代码等）替换用“#define”定义的字符串；
2. 文件包含复制：预处理程序中用“#include”格式包含的文件，将文件的内容插入到该命令所在的位置并删除原命令，从而把包含的文件和当前源文件连接成一个新的源文件，这与复制粘贴类似；
3. 条件编译处理：根据“#if”和“#endif”、“#ifdef”和“#ifndef”后面的条件确定需要编译的源代码。

## 2.2在Ubuntu下预处理的命令

以下两条命令均可实现预处理（效果等价，输出至文件hello.i）：

* gcc -E hello.c -o hello.i
* cpp hello.c > hello.i

预处理命令执行截图：

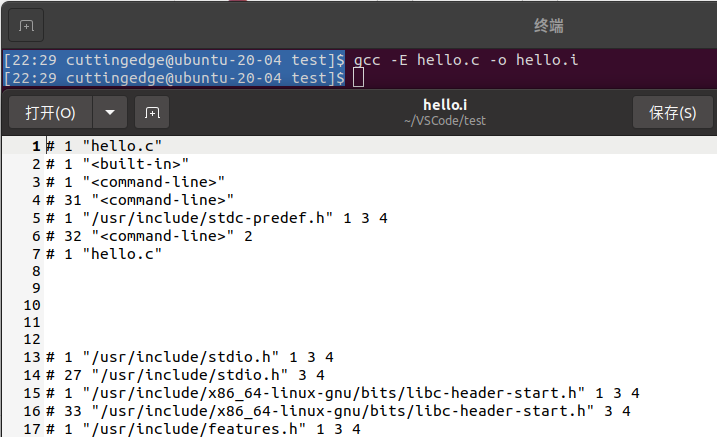


图 2-1 预处理命令1执行截图

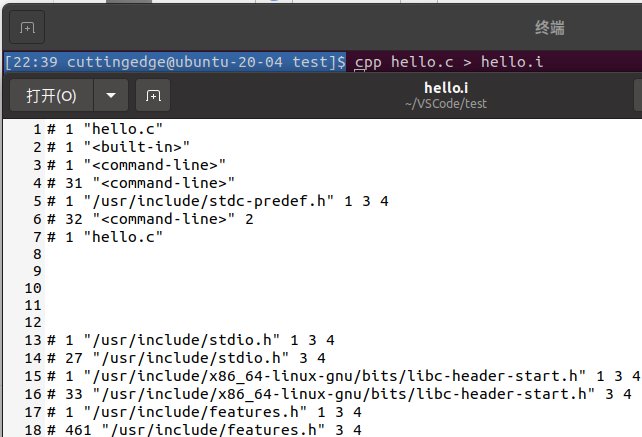


图 2-2 预处理命令2执行截图

## 2.3 Hello的预处理结果解析

原本的28行hello.c文件经过预处理环节，扩展成了3065行的ASCII码中间文本文件hello.i。具体解析如下：

首先是源代码文件等相关的一些信息（第1 ~ 7行），如下图：

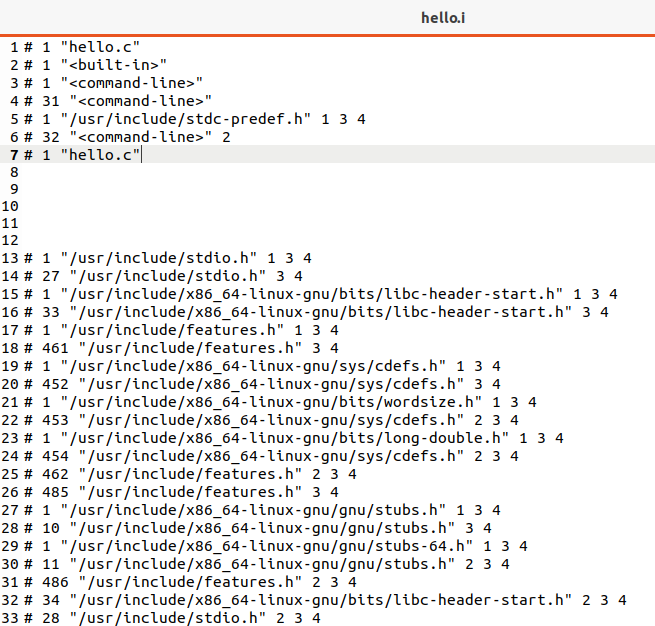


图 2‑3 预处理结果截图1（源代码文件信息）

随后是预处理扩展的内容（第13 ~ 3041行），部分内容见下图（图 2‑4至图 2-6）：

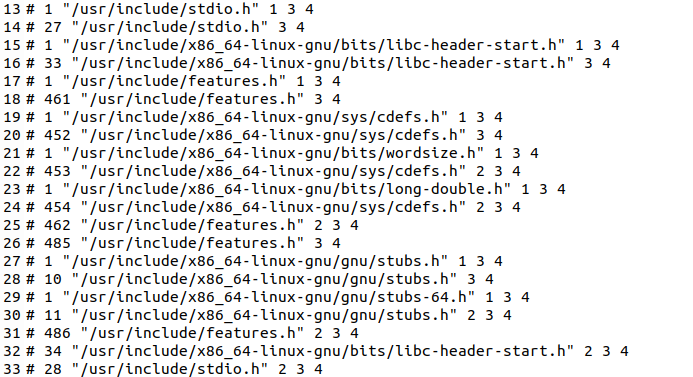


图 2‑4 预处理结果截图2（文件包含信息）

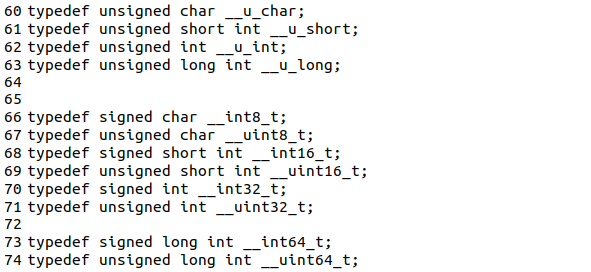


图 2‑5 预处理结果截图3（类型定义信息）

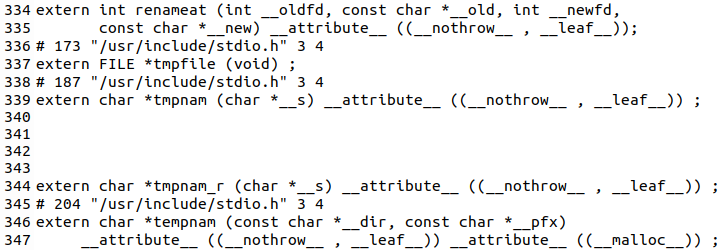


图 2‑6 预处理结果截图4（函数声明信息）

预处理的具体过程如下（以stdio.h为例说明）：

作为hello.c中包含的头文件，stdio.h是标准库文件（非标准库文件包含时一般使用双引号，cpp会在当前目录下进行查找），cpp到Linux系统的环境变量下寻找stdio.h，打开文件/usr/include/stdio.h，发现其中使用了“#define”、“#include”等，故cpp对它们进行递归展开替换，最终的hello.i文件中删除了原有的这部分；对于其中使用的“#ifdef”、“#ifndef”等条件编译语句，cpp会对条件值进行判断来决定是否对此部分进行包含。最终得到如上图（2-4包含信息、2-5类型定义及2-6函数声明）等部分。

下图为hello.c包含的头文件在系统中的路径位置，cpp从这里读取、复制和处理这些头文件，将它们添加至hello.i：

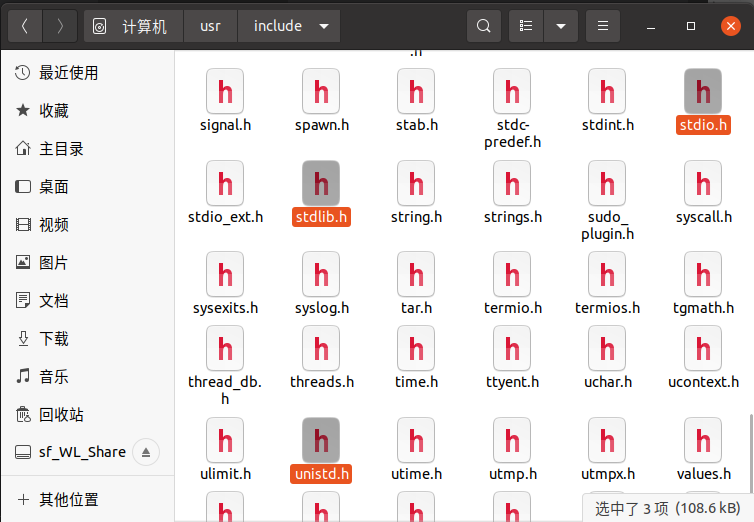


图 2‑7 头文件路径位置截图

最后的部分是hello.c中的源代码（第3043 ~ 3065行），除注释和“#include”语句被删除外，内容保持基本不变，如下图：

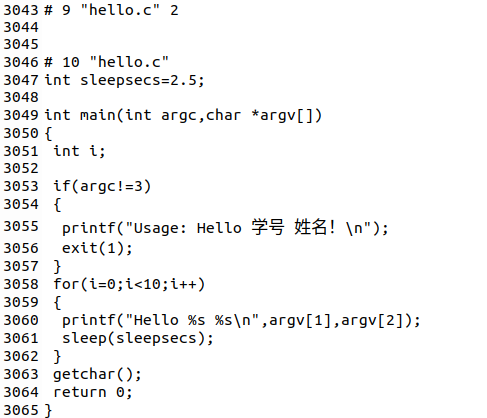


图 2‑8 预处理结果截图5（保留的源代码）

## 2.4 本章小结

本章主要探讨了预处理的概念、作用和命令，分析了hello.c源代码文件的预处理过程和结果，此过程深化了我对C语言预处理的理解。C语言预处理一般由预处理器(cpp)进行，主要完成四项工作：宏展开、文件包含复制、条件编译处理和删除注释及多余空白字符，为之后的编译等流程奠定了基础。

# 第3章 编译

## 3.1 编译的概念与作用

1. **编译的概念**

编译是指对经过预处理之后的源程序代码进行分析检查，确认所有语句均符合语法规则后将其翻译成等价的中间代码或汇编代码(assembly code)的过程。在此处指编译器将hello.i翻译成hello.s。

1. **编译的作用**

编译一般包括以下流程，每个步骤都有一定的作用：

1. 词法分析：对由字符组成的单词进行处理，从左至右逐个字符地对源程序进行扫描，产生一个个的单词符号，把作为字符串的源程序改造成为单词符号串的中间程序。
2. 语法分析：编译程序的语法分析器以单词符号作为输入，分析单词符号串是否形成符合语法规则的语法单位，如表达式、赋值、循环等，最后看是否构成一个符合要求的程序。
3. 中间代码：使程序的结构在逻辑上更为简单明确。
4. 代码优化：对程序进行多种等价变换，便于生成更有效的目标代码。
5. 目标代码：目标代码生成器把语法分析后或优化后的中间代码变换成目标代码，此处指目标代码为汇编代码。

可以说，编译的作用是通过一系列步骤让源代码更接近机器语言。编译是汇编阶段翻译成机器语言的前提。

## 3.2 在Ubuntu下编译的命令

gcc -S hello.i -o hello.s

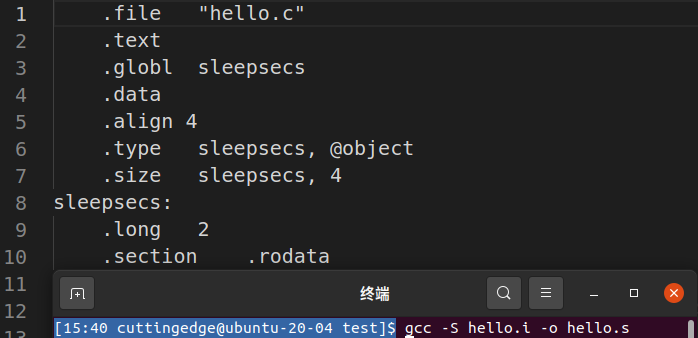


图 3‑1 编译命令执行截图

## 3.3 Hello的编译结果解析

### 3.3.1 数据与赋值

1. 常量数据
2. printf函数中用到的格式字符串、输出字符串被保存在.rodata段

* 源程序代码：

第18行：printf("Usage: Hello 学号 姓名！\n");

第23行：printf("Hello %s %s\n",argv[1],argv[2]);

* 汇编代码（第10 ~ 14行）：

.section   .rodata

.LC0:

    .string "Usage: Hello \345\255\246\345\217\267 \345\247\223\345\220\215\357\274\201"

.LC1:

.string "Hello %s %s\n"

1. if条件判断值、for循环终止条件值在.text段，运行时使用

* 源程序代码：

第16行：if(argc!=3)

第21行：for(i=0;i<10;i++)

* 汇编代码：

第30行：cmpl   $3, -20(%rbp)

第55行：cmpl   $9, -4(%rbp)

1. 变量数据
2. 全局变量：sleepsecs保存在.data段，赋值无需具体汇编语句。

* 源程序代码：

第10行：int sleepsecs=2.5;

* 汇编代码（第2 ~ 9行）：

    .text

    .globl  sleepsecs

    .data

    .align 4

    .type   sleepsecs, @object

    .size   sleepsecs, 4

sleepsecs:

.long   2

注：可以注意到，sleepsecs被编译器从int类型隐式转换成了long类型，设定对齐为4字节，实际值为2（小数截断），这种隐式转换不影响数据的精度和正确性，应该是编译器的内置默认规则。

1. 局部变量：局部变量i（4字节int型）在运行时保存在栈中，使用一条movl指令进行赋值，使用一条addl指令进行增一。

* 源程序代码：

第14行：int i;

第21行：for(i=0;i<10;i++)

* 汇编程序代码：

第37行（初始化）：movl   $0, -4(%rbp)

第53行（增一）：addl   $1, -4(%rbp)

由此可见，局部变量i在赋初值后被保存在地址为%rbp-4的栈位置上。

### 3.3.2 算术操作

在for循环体中，对循环变量i的更新使用了++自增运算，汇编代码翻译成addl指令（4字节int型对应后缀“l”）：

源程序代码（第21行）：for(i=0;i<10;i++)

汇编代码对应操作（第53行）：addl   $1, -4(%rbp)

### 3.3.3 数组、指针、结构操作

主函数main()的第二个参数是char \*argv[]（参数字符串数组指针），在argv数组中，argv[0]为输入程序的路径和名称字符串起始位置，argv[1]和argv[2]为其后的两个参数字符串的起始位置。汇编代码中相关的指令如下：

.LFB6代码块中（第29行）：movq    %rsi, -32(%rbp)

这条指令将main()的第二个参数从寄存器写到了栈空间中。

.L4代码块中（第40 ~ 45行）：

     movq    -32(%rbp), %rax

     addq    $16, %rax

     movq    (%rax), %rdx

     movq    -32(%rbp), %rax

     addq    $8, %rax

     movq    (%rax), %rax

这6条指令从栈上取这一参数，并按照基址-变址寻址法访问argv[1]和argv[2]（由于指针char\*大小为8字节，分别偏移8、16字节来访问）。

### 3.3.4 关系操作及控制转移

1. 程序中if条件判断处的关系操作与控制转移：

源程序代码（第16行）：if(argc!=3)

汇编代码对应操作（第30、31行）：

     cmpl    $3, -20(%rbp)

     je  .L2

je使用cmpl设置的条件码(ZF)，若ZF = 0，说明argc等于3，条件不成立，控制转移至.L2（for循环部分，程序主体功能）；若ZF = 1，说明argc不等于3（即执行程序时传入的参数个数不符合要求），继续执行输出提示信息并退出。

1. 程序中for循环终止条件判断涉及的关系操作与控制转移：

源程序代码（第21行）：for(i=0;i<10;i++)

汇编代码对应操作（第55、56行）：

     cmpl    $9, -4(%rbp)

     jle .L4

与(1)类似，此处jle使用cmpl设置的条件码(ZF SF OF)，若(SF^OF) | ZF = 1，说明循环终止条件不成立（变量i的值小于或等于9），控制转移至.L4，继续执行循环体；若(SF^OF) | ZF = 0，则循环终止条件成立（变量i的值达到10），不再跳转至循环体开始位置，继续向后执行直至退出。

值得注意的是，源程序代码的逻辑与编译器翻译生成的逻辑有细微的差别。源代码中判断i<10，而编译器将其调整为判断i<=9，但实际上二者等价。

### 3.3.5 函数操作

源代码中的函数有main()函数，printf()函数（第一处被编译器优化为puts函数），exit()函数，sleep()函数，getchar()函数，以下为对每个函数的具体分析。

1. main()函数：
2. 参数传递：int argc, char \*argv[]

相关汇编代码：

.LFB6代码块中（第28、29行）：

     movl    %edi, -20(%rbp)

     movq    %rsi, -32(%rbp)

由此可见，第一个参数通过寄存器EDI传递，第二个参数通过寄存器RSI传递，这一步将两个参数写入栈空间。

1. 函数调用：

被启动函数调用，hello.s中没有体现，但为汇编器进行相关处理提供了信息。

相关汇编代码（第18 ~ 26行）：

main:

.LFB6:

    .cfi\_startproc

    endbr64

    pushq   %rbp

    .cfi\_def\_cfa\_offset 16

    .cfi\_offset 6, -16

    movq    %rsp, %rbp

    .cfi\_def\_cfa\_register 6

此部分汇编指令标记了程序入口等信息，应该是提供给汇编器。

1. 函数返回：正常情况返回0，参数个数不正确返回1。

正常情况相关汇编代码（第58、59行）：

      movl    $0, %eax

     leave

返回1情况（调用exit()函数，第34、35行）：

     movl    $1, %edi

     call    exit@PLT

1. printf()函数：
2. 参数传递：需要输出的字符串（可能含格式）。

源代码（第18、23行）及对应汇编代码（第32 ~ 33、40 ~ 49行）：

printf("Usage: Hello 学号 姓名！\n");

     leaq    .LC0(%rip), %rdi

     call    puts@PLT

printf("Hello %s %s\n",argv[1],argv[2]);

     movq    -32(%rbp), %rax

     addq    $16, %rax

     movq    (%rax), %rdx

     movq    -32(%rbp), %rax

     addq    $8, %rax

     movq    (%rax), %rax

     movq    %rax, %rsi

     leaq    .LC1(%rip), %rdi

     movl    $0, %eax

     call    printf@PLT

注：从栈空间取argc[1]、argc[2]，从只读数据段取格式/输出字符串，作为参数传递给printf()进行输出。

1. 函数调用：主函数通过call指令调用。
2. 函数返回：返回值被忽略。
3. exit()函数：
4. 参数传递：退出状态值（int类型）

源代码（第19行）及对应汇编代码（第34、35行）：

exit(1);

     movl    $1, %edi

     call    exit@PLT

注：使用寄存器EDI传递参数（整数值1），调用exit()函数以状态1退出。

1. 函数调用：主函数通过call指令调用。
2. 函数返回：函数不返回，直接退出程序。
3. sleep()函数：
4. 参数传递：休眠时间（int类型）

源代码（第24行）及对应汇编代码（第50 ~ 52行）：

sleep(sleepsecs);

movl   sleepsecs(%rip), %eax

movl    %eax, %edi

call    sleep@PLT

注：使用全局变量sleepsecs（分析见3.3.1节全局变量部分）作为参数调用sleep()函数。

1. 函数调用：主函数通过call指令调用。
2. 函数返回：返回值被忽略（返回的是实际休眠时间）。
3. getchar()函数：
4. 参数传递：无。
5. 函数调用：主函数通过call指令调用，相关汇编代码如下（第57行）：

call   getchar@PLT

1. 函数返回：返回char类型值，在此程序中被忽略。

## 3.4 本章小结

本章围绕hello.i经编译器处理得到hello.s的过程，介绍了编译的概念、过程并具体分析了Hello程序的编译结果。编译阶段分析检查源程序，确认所有的语句都符合语法规则后将其翻译成等价的汇编代码（中间代码）表示。完成本章内容的过程加深了我对编译阶段的理解，也引导我进行了课程第三章相关知识的复习。

# 第4章 汇编

## 4.1 汇编的概念与作用

1. **汇编的概念**

驱动程序运行（或直接运行）汇编器as，将汇编语言程序（这里指hello.s）翻译成机器语言指令，并将这些指令打包成可重定位目标文件（hello.o）的过程称为汇编，hello.o是二进制编码文件，包含程序的机器指令编码。

1. **汇编的作用**

汇编过程将汇编程序转化为机器可直接识别执行的机器语言程序。

## 4.2 在Ubuntu下汇编的命令

以下两条命令效果等价：

* gcc -c hello.s -o hello.o
* as hello.s -o hello.o

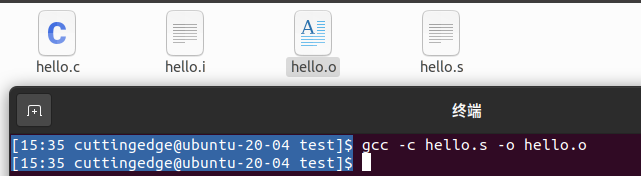


图 4‑1 汇编命令执行截图1

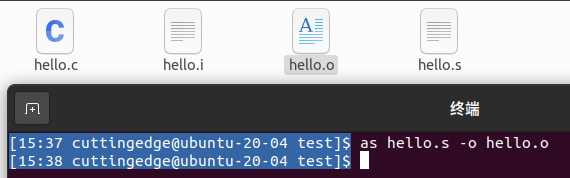


图 4‑2 汇编命令执行截图2

## 4.3 可重定位目标elf格式

### 4.3.1 readelf命令

使用命令：readelf -a hello.o > hello\_elf.txt

将hello.o中ELF格式相关信息重定向至文件hello\_elf.txt。

命令执行效果截图：

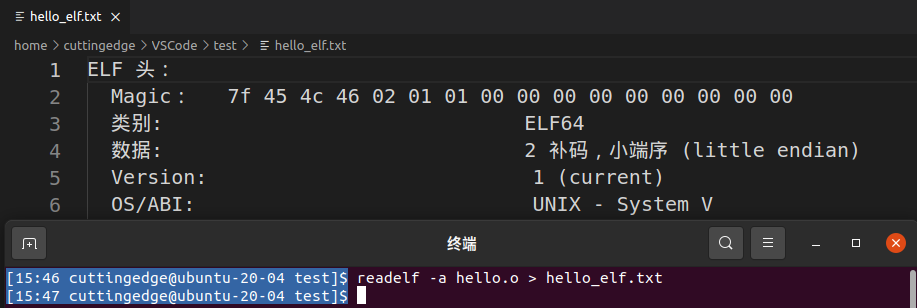


图 4‑3 readelf命令执行截图

### 4.3.2 ELF头

此部分内容（hello\_elf.txt文件第1 ~ 20行）如下：

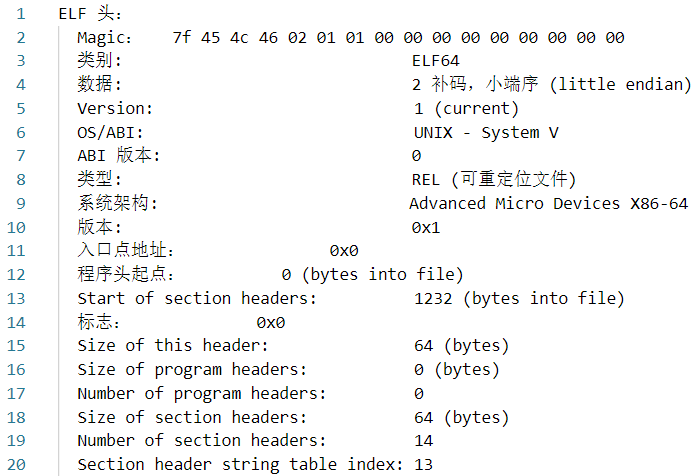


图 4‑4 ELF头信息

**分析：**

1. Magic用于标识ELF文件，7f 45 4c 46分别对应ASCII码的Del、字母E、字母L、字母F，操作系统在加载可执行文件时会确认是否正确，如果不正确则拒绝加载，其余标识位数、小/大端序、版本号等，后九个字节未定义；
2. 根据头文件的信息，可知该文件是可重定位目标文件，有14个节，其余部分的信息此处不再一一列举说明；

### 4.3.3 节头目表

此部分列出了hello.o中的14个节的名称、类型、地址、偏移量、大小等信息。具体内容（hello\_elf.txt文件第22 ~ 52行）如下：

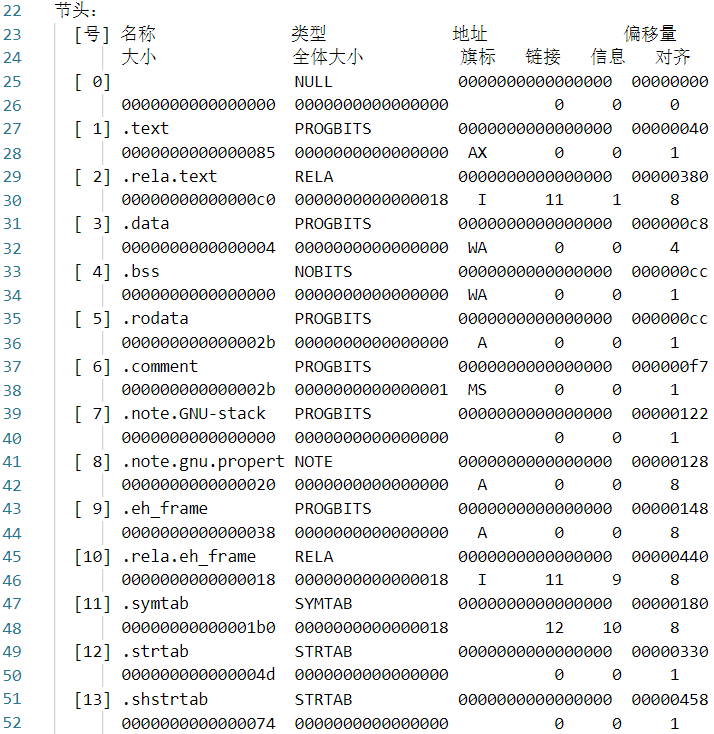


图 4‑5 节头目表信息

**分析：**

1. 由于是可重定位目标文件，所以每个节都从0开始，用于重定位；
2. .text段是可执行的，但是不能写；
3. .data段和.rodata段都不可执行且.rodata段不可写；
4. .bss段大小为0。

### 4.3.4 重定位节

重定位节记录了各段引用的符号相关信息，在链接时，需要通过重定位节对这些位置的地址进行重定位。链接器会通过重定位条目的类型判断如何计算地址值并使用偏移量等信息计算出正确的地址。

具体内容（hello\_elf.txt文件第65 ~ 78行）如下：

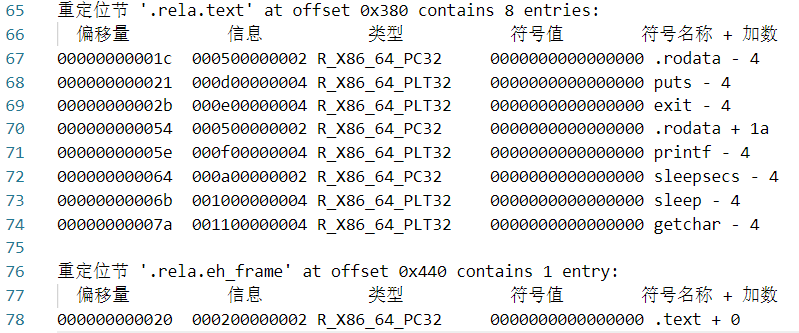


图 4-6 重定位节信息

**分析：**本程序需要重定位的符号有：.rodata，puts，exit，printf，sleepsecs，sleep，getchar及.text等。注意到重定位类型仅有R\_X86\_64\_PC32（PC相对寻址）和R\_X86\_64\_PLT32（使用PLT表寻址）两种，而未出现R\_X86\_64\_S32绝对寻址。

### 4.3.5 符号表

符号表（.symtab）存放在程序中定义和引用的函数和全局变量的信息。

具体内容（hello\_elf.txt文件第82 ~ 101行）如下：

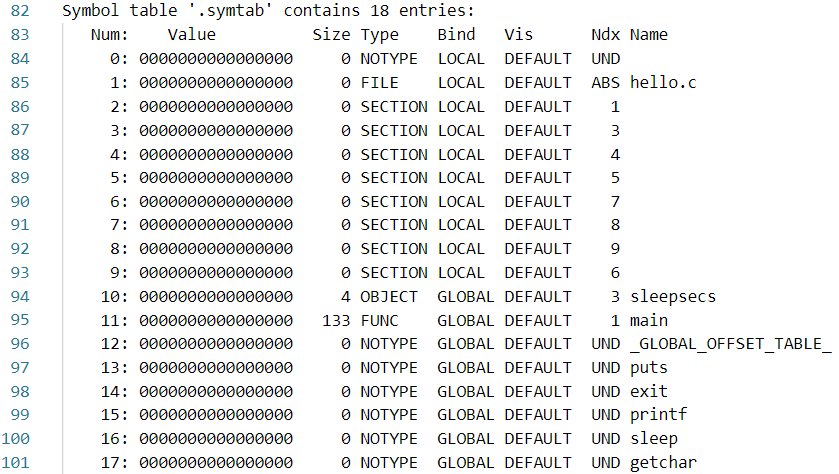


图 4-7 符号表信息

## 4.4 Hello.o的结果解析

**命令：**objdump -d -r hello.o > hello\_dis.txt

**反汇编代码：**

hello.o：     文件格式 elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:

   0: f3 0f 1e fa             endbr64

   4: 55                      push   %rbp

   5: 48 89 e5                mov    %rsp,%rbp

   8: 48 83 ec 20             sub    $0x20,%rsp

   c: 89 7d ec                mov    %edi,-0x14(%rbp)

   f: 48 89 75 e0             mov    %rsi,-0x20(%rbp)

  13: 83 7d ec 03             cmpl   $0x3,-0x14(%rbp)

  17: 74 16                   je     2f <main+0x2f>

  19: 48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 20 <main+0x20>

         1c: R\_X86\_64\_PC32 .rodata-0x4

  20: e8 00 00 00 00          callq  25 <main+0x25>

         21: R\_X86\_64\_PLT32   puts-0x4

  25: bf 01 00 00 00          mov    $0x1,%edi

  2a: e8 00 00 00 00          callq  2f <main+0x2f>

         2b: R\_X86\_64\_PLT32   exit-0x4

  2f: c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)

  36: eb 3b                   jmp    73 <main+0x73>

  38: 48 8b 45 e0             mov    -0x20(%rbp),%rax

  3c: 48 83 c0 10             add    $0x10,%rax

  40: 48 8b 10                mov    (%rax),%rdx

  43: 48 8b 45 e0             mov    -0x20(%rbp),%rax

  47: 48 83 c0 08             add    $0x8,%rax

  4b: 48 8b 00                mov    (%rax),%rax

  4e: 48 89 c6                mov    %rax,%rsi

  51: 48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 58 <main+0x58>

         54: R\_X86\_64\_PC32 .rodata+0x1a

  58: b8 00 00 00 00          mov    $0x0,%eax

  5d: e8 00 00 00 00          callq  62 <main+0x62>

         5e: R\_X86\_64\_PLT32   printf-0x4

  62: 8b 05 00 00 00 00       mov    0x0(%rip),%eax        # 68 <main+0x68>

         64: R\_X86\_64\_PC32 sleepsecs-0x4

  68: 89 c7                   mov    %eax,%edi

  6a: e8 00 00 00 00          callq  6f <main+0x6f>

         6b: R\_X86\_64\_PLT32   sleep-0x4

  6f: 83 45 fc 01             addl   $0x1,-0x4(%rbp)

  73: 83 7d fc 09             cmpl   $0x9,-0x4(%rbp)

  77: 7e bf                   jle    38 <main+0x38>

  79: e8 00 00 00 00          callq  7e <main+0x7e>

         7a: R\_X86\_64\_PLT32   getchar-0x4

  7e: b8 00 00 00 00          mov    $0x0,%eax

  83: c9                      leaveq

  84: c3                      retq

**hello.s中对应的汇编代码（第21 ~ 61行）：**

    endbr64

    pushq   %rbp

    .cfi\_def\_cfa\_offset 16

    .cfi\_offset 6, -16

    movq    %rsp, %rbp

    .cfi\_def\_cfa\_register 6

    subq    $32, %rsp

    movl    %edi, -20(%rbp)

    movq    %rsi, -32(%rbp)

    cmpl    $3, -20(%rbp)

    je  .L2

    leaq    .LC0(%rip), %rdi

    call    puts@PLT

    movl    $1, %edi

    call    exit@PLT

.L2:

    movl    $0, -4(%rbp)

    jmp .L3

.L4:

    movq    -32(%rbp), %rax

    addq    $16, %rax

    movq    (%rax), %rdx

    movq    -32(%rbp), %rax

    addq    $8, %rax

    movq    (%rax), %rax

    movq    %rax, %rsi

    leaq    .LC1(%rip), %rdi

    movl    $0, %eax

    call    printf@PLT

    movl    sleepsecs(%rip), %eax

    movl    %eax, %edi

    call    sleep@PLT

    addl    $1, -4(%rbp)

.L3:

    cmpl    $9, -4(%rbp)

    jle .L4

    call    getchar@PLT

    movl    $0, %eax

    leave

    .cfi\_def\_cfa 7, 8

    ret

**比较分析：**

二者总体相同，但也有一些细微的差异：

1. 分支控制转移不同：对于跳转语句跳转的位置，hello.s中是.L2、.LC1等代码块的名称，而反汇编代码中跳转指令跳转的位置是相对于main函数起始位置偏移的地址（相对地址）；
2. 函数调用表示不同：hello.s中，call指令使用的是函数名，而反汇编代码中call指令使用的是待链接器重定位的相对偏移地址，这些调用只有在链接之后才能确定运行时的实际地址，因此在.rela.text节中为其添加了重定位条目；
3. hello.s中的全局变量、printf字符串等符号被替换成了待重定位的地址；
4. 数的表示不同：hello.s中的操作数均为十进制，而hello.o反汇编代码中的操作数被转换成十六进制；
5. hello.s中提供给汇编器的辅助信息在反汇编代码中不再出现，可能是在汇编器处理过程中被移除，如“.cfi\_def\_cfa\_offset 16”等。

## 4.5 本章小结

本章对汇编的概念、作用、可重定向目标文件的结构及对应反汇编代码等进行了较为详细的介绍。经过汇编阶段，汇编语言代码转化为机器语言，生成的可重定位目标文件(hello.o)为随后的链接阶段做好了准备。完成本章内容的过程加深了我对汇编过程、ELF格式以及重定位的理解。

# 第5章 链接

## 5.1 链接的概念与作用

1. **链接的概念**

链接是将各种代码和数据片段收集并组合成为一个单一文件的过程，这个文件可被加载（复制）到内存并执行。链接可以执行于编译时、执行时甚至运行时。在现代系统中，链接是由叫做链接器的程序自动执行的。

在此处，链接是指将可重定向目标文件hello.o与其他一些文件组合成为可执行目标文件hello。

1. **链接的作用**

链接使分离编译成为可能，我们不用将一个大型的应用程序组织为一个巨大的源文件，而是可以把它分解成更小、更好管理的模块，可以独立地修改和编译这些模块。当我们改变这些模块中的一个时，只需简单地重新编译它，并重新链接应用，而不必重新编译其他文件。

## 5.2 在Ubuntu下链接的命令

命令：ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib/x86\_64-linux-gnu/crt1.o hello.o /usr/lib/x86\_64-linux-gnu/crti.o /usr/lib/x86\_64-linux-gnu/libc.so /usr/lib/x86\_64-linux-gnu/crtn.o -o hello

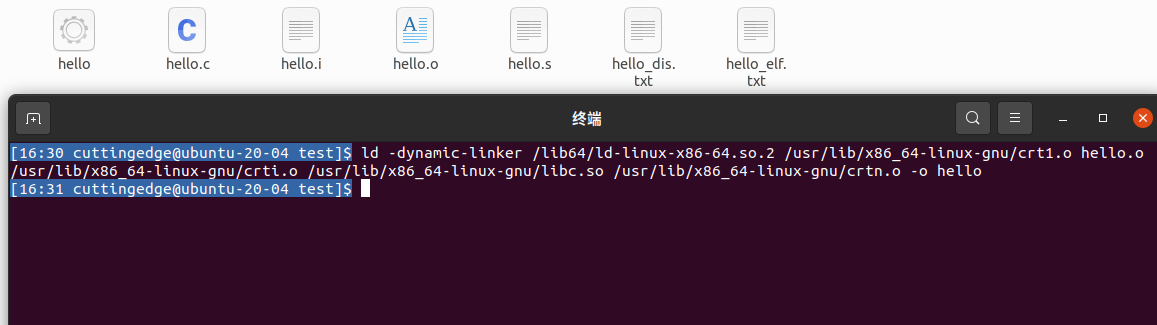


图 5-1 链接命令执行截图

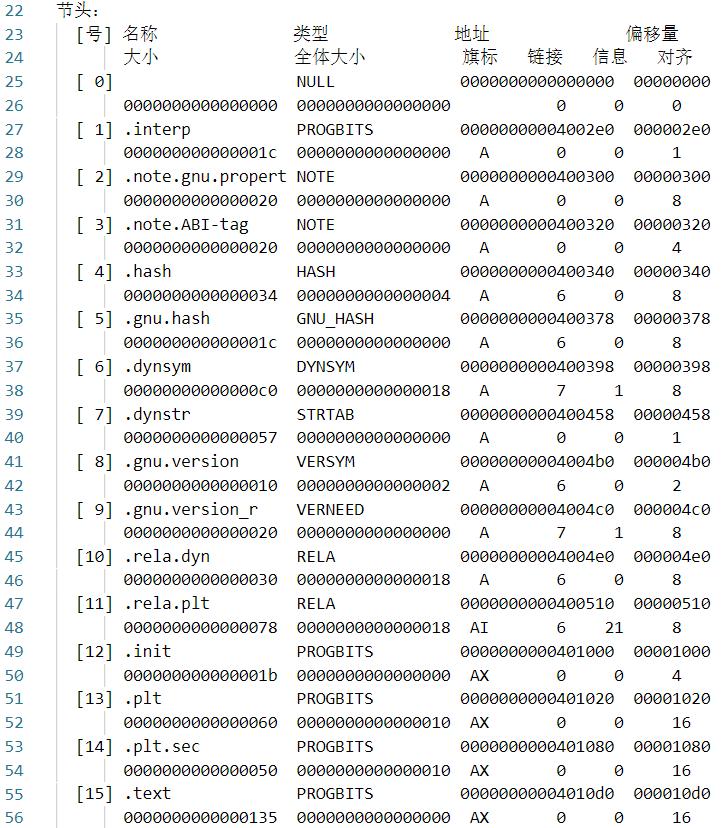
## 5.3 可执行目标文件hello的格式

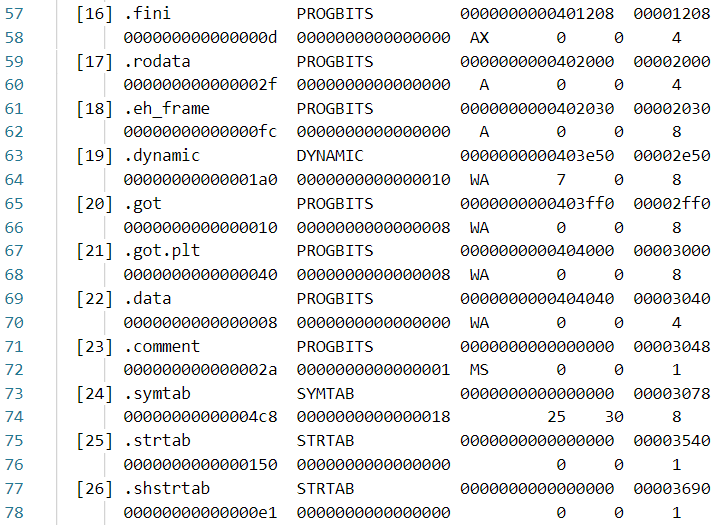
使用命令：readelf -a hello > hello1\_elf.txt

将hello中ELF格式相关信息重定向至文件hello1\_elf.txt。

各段的基本信息（起始地址、大小等）记录在节头部表中，具体内容（hello1\_elf.txt第22 ~ 78行）如下：

【见下页图】





（接上页）

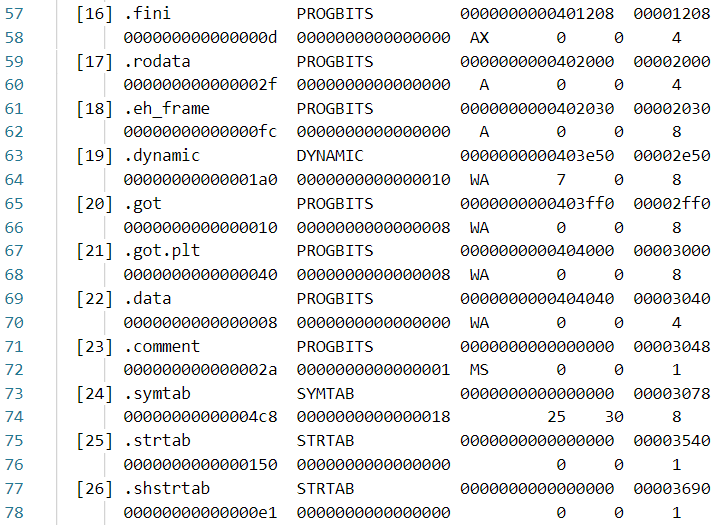


图 5-2 节头部表内容

## 5.4 hello的虚拟地址空间

使用edb加载hello，界面如下图：

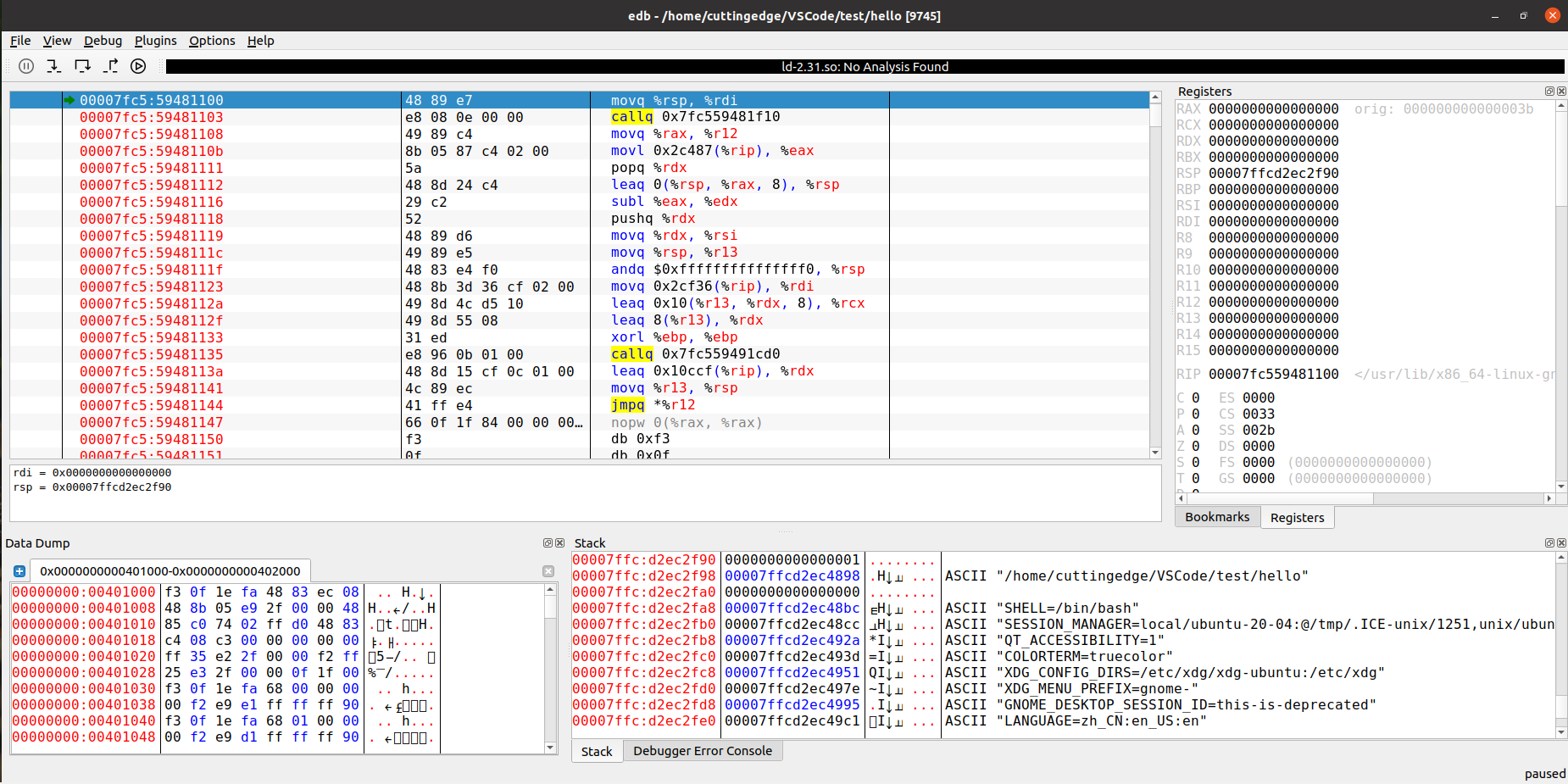


图 5-3 edb加载hello截图

此时Data Dump窗口中显示的就是hello的虚拟空间内容，如下图，显示范围为0x401000至0x402000。

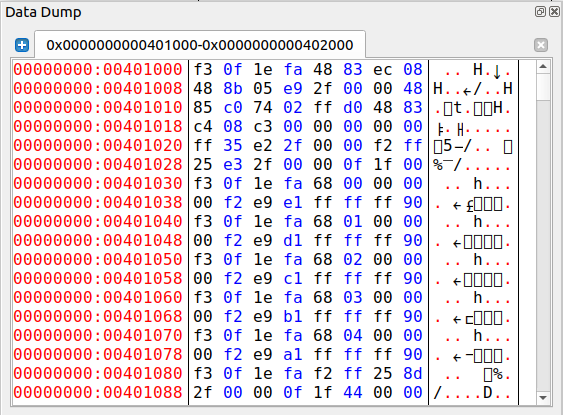


图 5-4 Data Dump窗口

通过与Symbols窗口对照，可以发现各段均一一对应，如下图（图 5-5 至 图 5-10，未全部展示）：

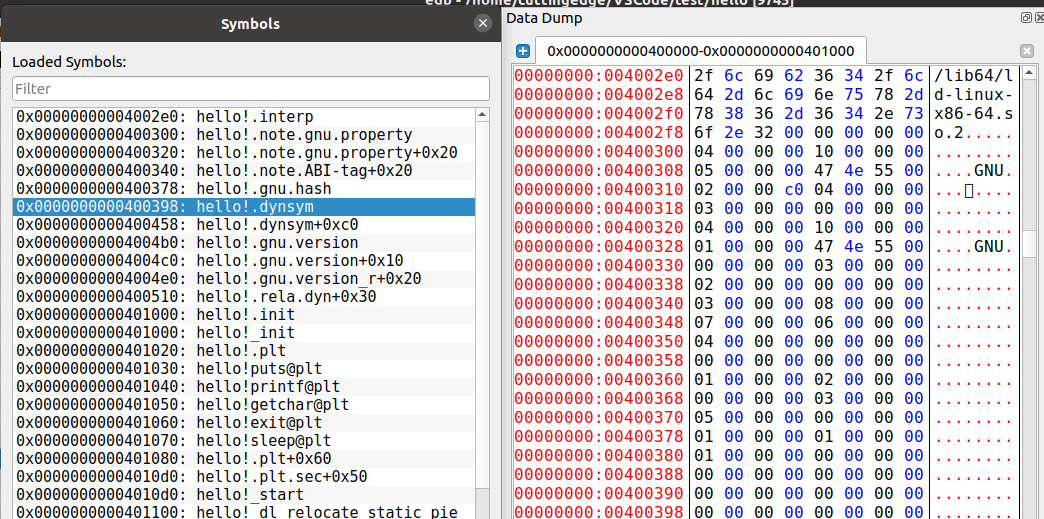


图 5-5 Symbols与Data Dump对照截图1

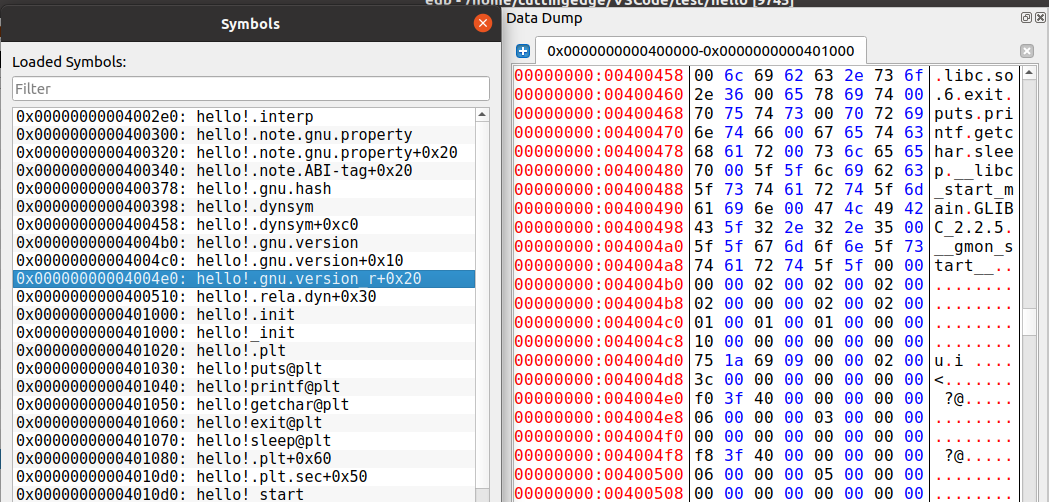


图 5-6 Symbols与Data Dump对照截图2

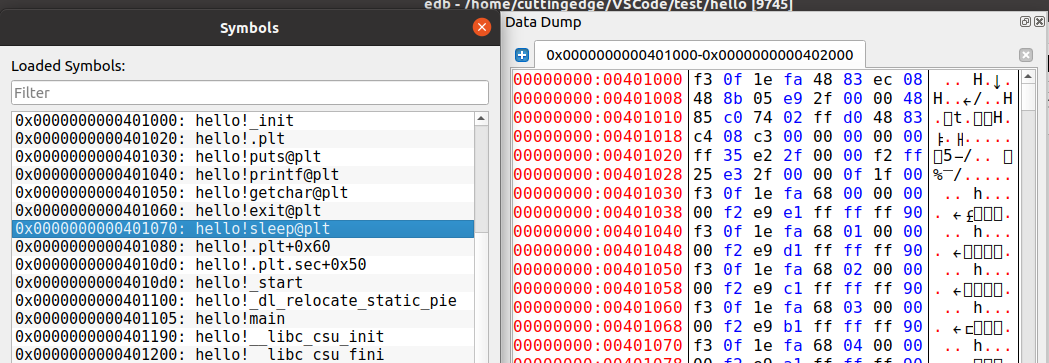


图 5-7 Symbols与Data Dump对照截图3

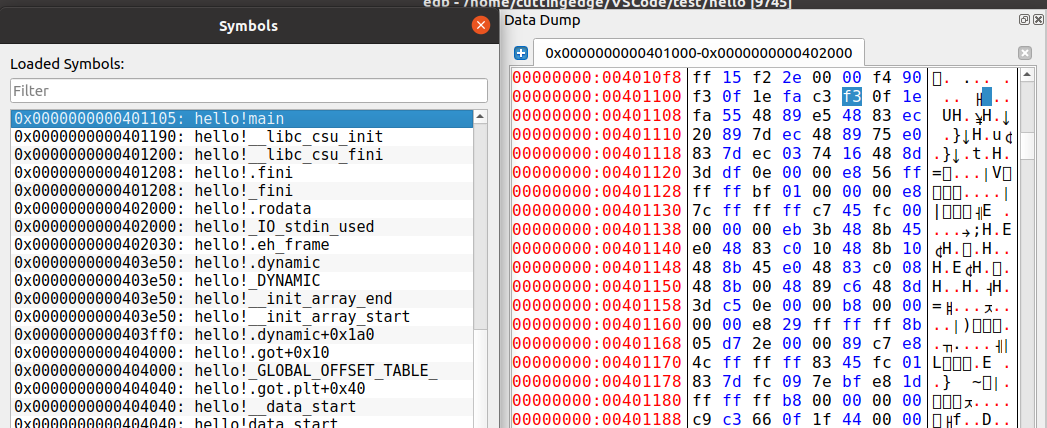


图 5-8 Symbols与Data Dump对照截图4（.text段部分）

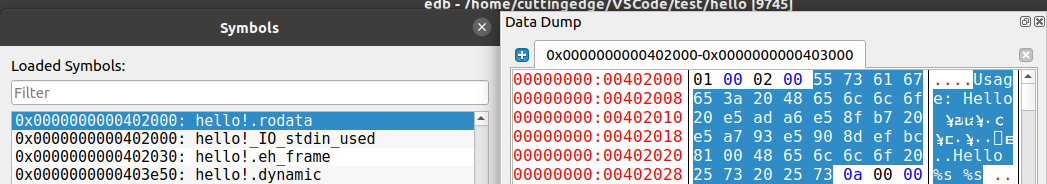


图 5-9 Symbols与Data Dump对照截图5（.rodata段部分）

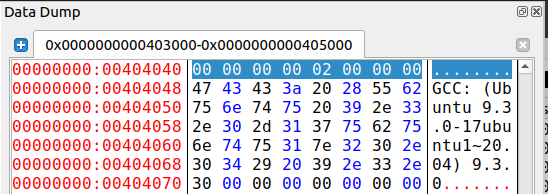


图 5-10 Symbols与Data Dump对照截图6（.data段部分）

在图5-9、5-10中，我们分别可以看到程序printf函数的字符串、全局变量sleepsecs（值为2），进一步说明了各段的虚拟地址与节头部表的对应关系。

## 5.5 链接的重定位过程分析

**命令：**objdump -d -r hello > hello1\_dis.txt

**反汇编代码：**

hello：     文件格式 elf64-x86-64

Disassembly of section .init:

0000000000401000 <\_init>:

  401000: f3 0f 1e fa           endbr64

  401004: 48 83 ec 08           sub    $0x8,%rsp

  401008: 48 8b 05 e9 2f 00 00  mov    0x2fe9(%rip),%rax        # 403ff8 <\_\_gmon\_start\_\_>

  40100f: 48 85 c0              test   %rax,%rax

  401012: 74 02                 je     401016 <\_init+0x16>

  401014: ff d0                 callq  \*%rax

  401016: 48 83 c4 08           add    $0x8,%rsp

  40101a: c3                    retq

Disassembly of section .plt:

0000000000401020 <.plt>:

  401020: ff 35 e2 2f 00 00     pushq  0x2fe2(%rip)        # 404008 <\_GLOBAL\_OFFSET\_TABLE\_+0x8>

  401026: f2 ff 25 e3 2f 00 00  bnd jmpq \*0x2fe3(%rip)        # 404010 <\_GLOBAL\_OFFSET\_TABLE\_+0x10>

  40102d: 0f 1f 00              nopl   (%rax)

  401030: f3 0f 1e fa           endbr64

  401034: 68 00 00 00 00        pushq  $0x0

  401039: f2 e9 e1 ff ff ff     bnd jmpq 401020 <.plt>

  40103f: 90                    nop

  401040: f3 0f 1e fa           endbr64

  401044: 68 01 00 00 00        pushq  $0x1

  401049: f2 e9 d1 ff ff ff     bnd jmpq 401020 <.plt>

  40104f: 90                    nop

  401050: f3 0f 1e fa           endbr64

  401054: 68 02 00 00 00        pushq  $0x2

  401059: f2 e9 c1 ff ff ff     bnd jmpq 401020 <.plt>

  40105f: 90                    nop

  401060: f3 0f 1e fa           endbr64

  401064: 68 03 00 00 00        pushq  $0x3

  401069: f2 e9 b1 ff ff ff     bnd jmpq 401020 <.plt>

  40106f: 90                    nop

  401070: f3 0f 1e fa           endbr64

  401074: 68 04 00 00 00        pushq  $0x4

  401079: f2 e9 a1 ff ff ff     bnd jmpq 401020 <.plt>

  40107f: 90                    nop

Disassembly of section .plt.sec:

0000000000401080 <puts@plt>:

  401080: f3 0f 1e fa           endbr64

  401084: f2 ff 25 8d 2f 00 00  bnd jmpq \*0x2f8d(%rip)        # 404018 <puts@GLIBC\_2.2.5>

  40108b: 0f 1f 44 00 00        nopl   0x0(%rax,%rax,1)

0000000000401090 <printf@plt>:

  401090: f3 0f 1e fa           endbr64

  401094: f2 ff 25 85 2f 00 00  bnd jmpq \*0x2f85(%rip)        # 404020 <printf@GLIBC\_2.2.5>

  40109b: 0f 1f 44 00 00        nopl   0x0(%rax,%rax,1)

00000000004010a0 <getchar@plt>:

  4010a0: f3 0f 1e fa           endbr64

  4010a4: f2 ff 25 7d 2f 00 00  bnd jmpq \*0x2f7d(%rip)        # 404028 <getchar@GLIBC\_2.2.5>

  4010ab: 0f 1f 44 00 00        nopl   0x0(%rax,%rax,1)

00000000004010b0 <exit@plt>:

  4010b0: f3 0f 1e fa           endbr64

  4010b4: f2 ff 25 75 2f 00 00  bnd jmpq \*0x2f75(%rip)        # 404030 <exit@GLIBC\_2.2.5>

  4010bb: 0f 1f 44 00 00        nopl   0x0(%rax,%rax,1)

00000000004010c0 <sleep@plt>:

  4010c0: f3 0f 1e fa           endbr64

  4010c4: f2 ff 25 6d 2f 00 00  bnd jmpq \*0x2f6d(%rip)        # 404038 <sleep@GLIBC\_2.2.5>

  4010cb: 0f 1f 44 00 00        nopl   0x0(%rax,%rax,1)

Disassembly of section .text:

00000000004010d0 <\_start>:

  4010d0: f3 0f 1e fa           endbr64

  4010d4: 31 ed                 xor    %ebp,%ebp

  4010d6: 49 89 d1              mov    %rdx,%r9

  4010d9: 5e                    pop    %rsi

  4010da: 48 89 e2              mov    %rsp,%rdx

  4010dd: 48 83 e4 f0           and    $0xfffffffffffffff0,%rsp

  4010e1: 50                    push   %rax

  4010e2: 54                    push   %rsp

  4010e3: 49 c7 c0 00 12 40 00  mov    $0x401200,%r8

  4010ea: 48 c7 c1 90 11 40 00  mov    $0x401190,%rcx

  4010f1: 48 c7 c7 05 11 40 00  mov    $0x401105,%rdi

  4010f8: ff 15 f2 2e 00 00     callq  \*0x2ef2(%rip)        # 403ff0 <\_\_libc\_start\_main@GLIBC\_2.2.5>

  4010fe: f4                    hlt

  4010ff: 90                    nop

0000000000401100 <\_dl\_relocate\_static\_pie>:

  401100: f3 0f 1e fa           endbr64

  401104: c3                    retq

0000000000401105 <main>:

  401105: f3 0f 1e fa           endbr64

  401109: 55                    push   %rbp

  40110a: 48 89 e5              mov    %rsp,%rbp

  40110d: 48 83 ec 20           sub    $0x20,%rsp

  401111: 89 7d ec              mov    %edi,-0x14(%rbp)

  401114: 48 89 75 e0           mov    %rsi,-0x20(%rbp)

  401118: 83 7d ec 03           cmpl   $0x3,-0x14(%rbp)

  40111c: 74 16                 je     401134 <main+0x2f>

  40111e: 48 8d 3d df 0e 00 00  lea    0xedf(%rip),%rdi        # 402004 <\_IO\_stdin\_used+0x4>

  401125: e8 56 ff ff ff        callq  401080 <puts@plt>

  40112a: bf 01 00 00 00        mov    $0x1,%edi

  40112f: e8 7c ff ff ff        callq  4010b0 <exit@plt>

  401134: c7 45 fc 00 00 00 00  movl   $0x0,-0x4(%rbp)

  40113b: eb 3b                 jmp    401178 <main+0x73>

  40113d: 48 8b 45 e0           mov    -0x20(%rbp),%rax

  401141: 48 83 c0 10           add    $0x10,%rax

  401145: 48 8b 10              mov    (%rax),%rdx

  401148: 48 8b 45 e0           mov    -0x20(%rbp),%rax

  40114c: 48 83 c0 08           add    $0x8,%rax

  401150: 48 8b 00              mov    (%rax),%rax

  401153: 48 89 c6              mov    %rax,%rsi

  401156: 48 8d 3d c5 0e 00 00  lea    0xec5(%rip),%rdi        # 402022 <\_IO\_stdin\_used+0x22>

  40115d: b8 00 00 00 00        mov    $0x0,%eax

  401162: e8 29 ff ff ff        callq  401090 <printf@plt>

  401167: 8b 05 d7 2e 00 00     mov    0x2ed7(%rip),%eax        # 404044 <sleepsecs>

  40116d: 89 c7                 mov    %eax,%edi

  40116f: e8 4c ff ff ff        callq  4010c0 <sleep@plt>

  401174: 83 45 fc 01           addl   $0x1,-0x4(%rbp)

  401178: 83 7d fc 09           cmpl   $0x9,-0x4(%rbp)

  40117c: 7e bf                 jle    40113d <main+0x38>

  40117e: e8 1d ff ff ff        callq  4010a0 <getchar@plt>

  401183: b8 00 00 00 00        mov    $0x0,%eax

  401188: c9                    leaveq

  401189: c3                    retq

  40118a: 66 0f 1f 44 00 00     nopw   0x0(%rax,%rax,1)

0000000000401190 <\_\_libc\_csu\_init>:

  401190: f3 0f 1e fa           endbr64

  401194: 41 57                 push   %r15

  401196: 4c 8d 3d b3 2c 00 00  lea    0x2cb3(%rip),%r15        # 403e50 <\_DYNAMIC>

  40119d: 41 56                 push   %r14

  40119f: 49 89 d6              mov    %rdx,%r14

  4011a2: 41 55                 push   %r13

  4011a4: 49 89 f5              mov    %rsi,%r13

  4011a7: 41 54                 push   %r12

  4011a9: 41 89 fc              mov    %edi,%r12d

  4011ac: 55                    push   %rbp

  4011ad: 48 8d 2d 9c 2c 00 00  lea    0x2c9c(%rip),%rbp        # 403e50 <\_DYNAMIC>

  4011b4: 53                    push   %rbx

  4011b5: 4c 29 fd              sub    %r15,%rbp

  4011b8: 48 83 ec 08           sub    $0x8,%rsp

  4011bc: e8 3f fe ff ff        callq  401000 <\_init>

  4011c1: 48 c1 fd 03           sar    $0x3,%rbp

  4011c5: 74 1f                 je     4011e6 <\_\_libc\_csu\_init+0x56>

  4011c7: 31 db                 xor    %ebx,%ebx

  4011c9: 0f 1f 80 00 00 00 00  nopl   0x0(%rax)

  4011d0: 4c 89 f2              mov    %r14,%rdx

  4011d3: 4c 89 ee              mov    %r13,%rsi

  4011d6: 44 89 e7              mov    %r12d,%edi

  4011d9: 41 ff 14 df           callq  \*(%r15,%rbx,8)

  4011dd: 48 83 c3 01           add    $0x1,%rbx

  4011e1: 48 39 dd              cmp    %rbx,%rbp

  4011e4: 75 ea                 jne    4011d0 <\_\_libc\_csu\_init+0x40>

  4011e6: 48 83 c4 08           add    $0x8,%rsp

  4011ea: 5b                    pop    %rbx

  4011eb: 5d                    pop    %rbp

  4011ec: 41 5c                 pop    %r12

  4011ee: 41 5d                 pop    %r13

  4011f0: 41 5e                 pop    %r14

  4011f2: 41 5f                 pop    %r15

  4011f4: c3                    retq

  4011f5: 66 66 2e 0f 1f 84 00  data16 nopw %cs:0x0(%rax,%rax,1)

  4011fc: 00 00 00 00

0000000000401200 <\_\_libc\_csu\_fini>:

  401200: f3 0f 1e fa           endbr64

  401204: c3                    retq

Disassembly of section .fini:

0000000000401208 <\_fini>:

  401208: f3 0f 1e fa           endbr64

  40120c: 48 83 ec 08           sub    $0x8,%rsp

  401210: 48 83 c4 08           add    $0x8,%rsp

  401214: c3                    retq

objdump -d -r hello 分析hello与hello.o的不同，说明链接的过程。

结合hello.o的重定位项目，分析hello中对其怎么重定位的。

## 5.6 hello的执行流程

（*以下格式自行编排，编辑时删除*）

使用edb执行hello，说明从加载hello到\_start，到call main,以及程序终止的所有过程。请列出其调用与跳转的各个子程序名或程序地址。

## 5.7 Hello的动态链接分析

（*以下格式自行编排，编辑时删除*）

分析hello程序的动态链接项目，通过edb调试，分析在dl\_init前后，这些项目的内容变化。要截图标识说明。

## 5.8 本章小结

（*以下格式自行编排，编辑时删除*）

**（第5章1分）**

# 第6章 hello进程管理

## 6.1 进程的概念与作用

（*以下格式自行编排，编辑时删除*）

## 6.2 简述壳Shell-bash的作用与处理流程

（*以下格式自行编排，编辑时删除*）

## 6.3 Hello的fork进程创建过程

（*以下格式自行编排，编辑时删除*）

## 6.4 Hello的execve过程

（*以下格式自行编排，编辑时删除*）

## 6.5 Hello的进程执行

（*以下格式自行编排，编辑时删除*）

结合进程上下文信息、进程时间片，阐述进程调度的过程，用户态与核心态转换等等。

## 6.6 hello的异常与信号处理

（*以下格式自行编排，编辑时删除*）

hello执行过程中会出现哪几类异常，会产生哪些信号，又怎么处理的。

程序运行过程中可以按键盘，如不停乱按，包括回车，Ctrl-Z，Ctrl-C等，Ctrl-z后可以运行ps jobs pstree fg kill 等命令，请分别给出各命令及运行结截屏，说明异常与信号的处理。

## 6.7本章小结

（*以下格式自行编排，编辑时删除*）

**（第6章1分）**

# 第7章 hello的存储管理

## 7.1 hello的存储器地址空间

（*以下格式自行编排，编辑时删除*）

结合hello说明逻辑地址、线性地址、虚拟地址、物理地址的概念。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

（*以下格式自行编排，编辑时删除*）

## 7.3 Hello的线性地址到物理地址的变换-页式管理

（*以下格式自行编排，编辑时删除*）

## 7.4 TLB与四级页表支持下的VA到PA的变换

（*以下格式自行编排，编辑时删除*）

## 7.5 三级Cache支持下的物理内存访问

（*以下格式自行编排，编辑时删除*）

## 7.6 hello进程fork时的内存映射

（*以下格式自行编排，编辑时删除*）

## 7.7 hello进程execve时的内存映射

（*以下格式自行编排，编辑时删除*）

## 7.8 缺页故障与缺页中断处理

（*以下格式自行编排，编辑时删除*）

## 7.9动态存储分配管理

（*以下格式自行编排，编辑时删除*）

*Printf会调用malloc，请简述动态内存管理的基本方法与策略。*

## 7.10本章小结

（*以下格式自行编排，编辑时删除*）

**（第7章 2分）**

# 第8章 hello的IO管理

## 8.1 Linux的IO设备管理方法

（*以下格式自行编排，编辑时删除*）

设备的模型化：文件

设备管理：unix io接口

## 8.2 简述Unix IO接口及其函数

（*以下格式自行编排，编辑时删除*）

## 8.3 printf的实现分析

（*以下格式自行编排，编辑时删除*）

<https://www.cnblogs.com/pianist/p/3315801.html>

从vsprintf生成显示信息，到write系统函数，到陷阱-系统调用 int 0x80或syscall.

字符显示驱动子程序：从ASCII到字模库到显示vram（存储每一个点的RGB颜色信息）。

显示芯片按照刷新频率逐行读取vram，并通过信号线向液晶显示器传输每一个点（RGB分量）。

## 8.4 getchar的实现分析

（*以下格式自行编排，编辑时删除*）

异步异常-键盘中断的处理：键盘中断处理子程序。接受按键扫描码转成ascii码，保存到系统的键盘缓冲区。

getchar等调用read系统函数，通过系统调用读取按键ascii码，直到接受到回车键才返回。

## 8.5本章小结

（*以下格式自行编排，编辑时删除*）

**（第8章1分）**

# 结论

用计算机系统的语言，逐条总结hello所经历的过程。

你对计算机系统的设计与实现的深切感悟，你的创新理念，如新的设计与实现方法。

**（结论0分，缺失 -1分，根据内容酌情加分）**

# 附件

列出所有的中间产物的文件名，并予以说明起作用。

**（附件0分，缺失 -1分）**

# 参考文献

1. 深入理解计算机系统（原书第三版）.机械工业出版社, 2016.
2. GCC online documentation. <http://gcc.gnu.org/onlinedocs/>
3. 编译.百度百科. <https://baike.baidu.com/item/%E7%BC%96%E8%AF%91>
4. ELF文件头结构. CSDN博客.

<https://blog.csdn.net/king_cpp_py/article/details/80334086>