

**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。全文内容涵盖了计算机系统课程的主体框架，梳理、回顾了课程所学的主要知识点。

**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 -](#_Toc74926028)

[1.1 Hello简介 - 4 -](#_Toc74926029)

[1.2 环境与工具 - 4 -](#_Toc74926030)

[1.3 中间结果 - 5 -](#_Toc74926031)

[1.4 本章小结 - 5 -](#_Toc74926032)

[第2章 预处理 - 6 -](#_Toc74926033)

[2.1 预处理的概念与作用 - 6 -](#_Toc74926034)

[2.2在Ubuntu下预处理的命令 - 6 -](#_Toc74926035)

[2.3 Hello的预处理结果解析 - 7 -](#_Toc74926036)

[2.4 本章小结 - 9 -](#_Toc74926037)

[第3章 编译 - 10 -](#_Toc74926038)

[3.1 编译的概念与作用 - 10 -](#_Toc74926039)

[3.2 在Ubuntu下编译的命令 - 10 -](#_Toc74926040)

[3.3 Hello的编译结果解析 - 11 -](#_Toc74926041)

[3.3.1 数据与赋值 - 11 -](#_Toc74926042)

[3.3.2 算术操作 - 12 -](#_Toc74926043)

[3.3.3 数组、指针、结构操作 - 12 -](#_Toc74926044)

[3.3.4 关系操作及控制转移 - 12 -](#_Toc74926045)

[3.3.5 函数操作 - 13 -](#_Toc74926046)

[3.4 本章小结 - 15 -](#_Toc74926047)

[第4章 汇编 - 16 -](#_Toc74926048)

[4.1 汇编的概念与作用 - 16 -](#_Toc74926049)

[4.2 在Ubuntu下汇编的命令 - 16 -](#_Toc74926050)

[4.3 可重定位目标elf格式 - 16 -](#_Toc74926051)

[4.3.1 readelf命令 - 16 -](#_Toc74926052)

[4.3.2 ELF头 - 17 -](#_Toc74926053)

[4.3.3 节头目表 - 18 -](#_Toc74926054)

[4.3.4 重定位节 - 18 -](#_Toc74926055)

[4.3.5 符号表 - 19 -](#_Toc74926056)

[4.4 Hello.o的结果解析 - 20 -](#_Toc74926057)

[4.5 本章小结 - 22 -](#_Toc74926058)

[第5章 链接 - 23 -](#_Toc74926059)

[5.1 链接的概念与作用 - 23 -](#_Toc74926060)

[5.2 在Ubuntu下链接的命令 - 23 -](#_Toc74926061)

[5.3 可执行目标文件hello的格式 - 23 -](#_Toc74926062)

[5.4 hello的虚拟地址空间 - 25 -](#_Toc74926063)

[5.5 链接的重定位过程分析 - 27 -](#_Toc74926064)

[5.6 hello的执行流程 - 29 -](#_Toc74926065)

[5.7 Hello的动态链接分析 - 30 -](#_Toc74926066)

[5.8 本章小结 - 31 -](#_Toc74926067)

[第6章 hello进程管理 - 32 -](#_Toc74926068)

[6.1 进程的概念与作用 - 32 -](#_Toc74926069)

[6.2 简述壳Shell-bash的作用与处理流程 - 32 -](#_Toc74926070)

[6.3 Hello的fork进程创建过程 - 33 -](#_Toc74926071)

[6.4 Hello的execve过程 - 33 -](#_Toc74926072)

[6.5 Hello的进程执行 - 33 -](#_Toc74926073)

[6.6 hello的异常与信号处理 - 34 -](#_Toc74926074)

[6.7本章小结 - 37 -](#_Toc74926075)

[第7章 hello的存储管理 - 38 -](#_Toc74926076)

[7.1 hello的存储器地址空间 - 38 -](#_Toc74926077)

[7.2 Intel逻辑地址到线性地址的变换-段式管理 - 38 -](#_Toc74926078)

[7.3 Hello的线性地址到物理地址的变换-页式管理 - 39 -](#_Toc74926079)

[7.4 TLB与四级页表支持下的VA到PA的变换 - 40 -](#_Toc74926080)

[7.5 三级Cache支持下的物理内存访问 - 41 -](#_Toc74926081)

[7.6 hello进程fork时的内存映射 - 42 -](#_Toc74926082)

[7.7 hello进程execve时的内存映射 - 42 -](#_Toc74926083)

[7.8 缺页故障与缺页中断处理 - 42 -](#_Toc74926084)

[7.9动态内存分配管理 - 43 -](#_Toc74926085)

[7.10本章小结 - 44 -](#_Toc74926086)

[第8章 hello的IO管理 - 45 -](#_Toc74926087)

[8.1 Linux的IO设备管理方法 - 45 -](#_Toc74926088)

[8.2 简述Unix IO接口及其函数 - 45 -](#_Toc74926089)

[8.3 printf的实现分析 - 46 -](#_Toc74926090)

[8.4 getchar的实现分析 - 48 -](#_Toc74926091)

[8.5本章小结 - 49 -](#_Toc74926092)

[结论 - 50 -](#_Toc74926093)

[附件 - 51 -](#_Toc74926094)

[参考文献 - 52 -](#_Toc74926095)

# 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：

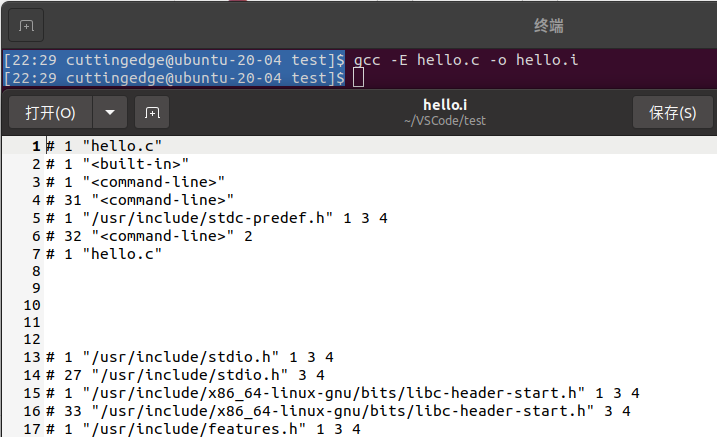


图 2-1 预处理命令1执行截图

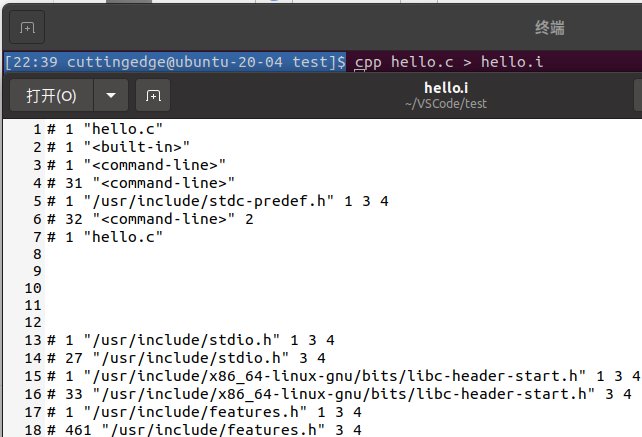


图 2-2 预处理命令2执行截图

## 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：

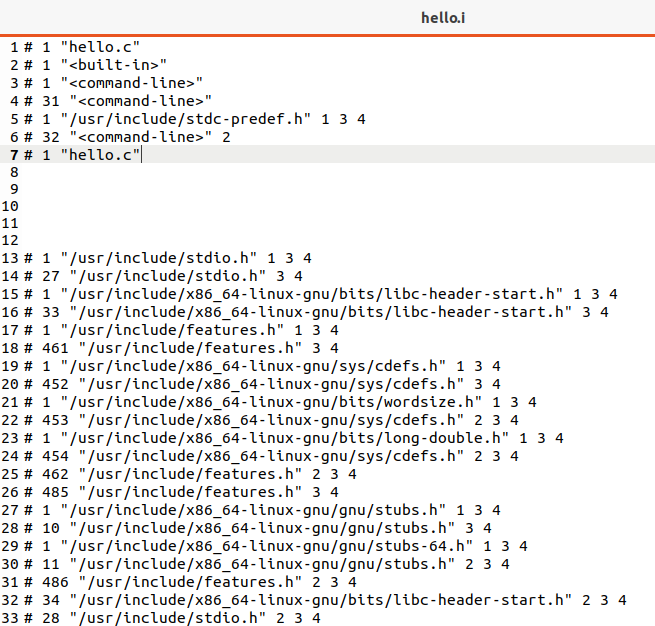


图 2‑3 预处理结果截图1（源代码文件信息）

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）：

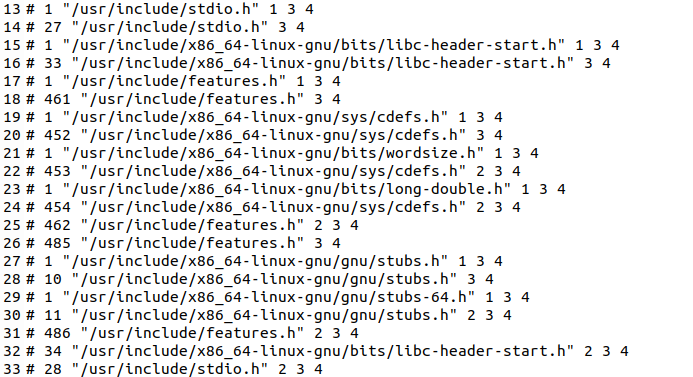


图 2‑4 预处理结果截图2（文件包含信息）

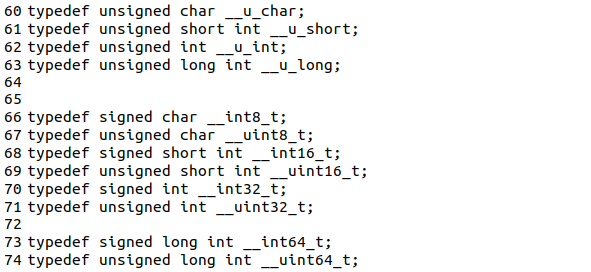


图 2‑5 预处理结果截图3（类型定义信息）

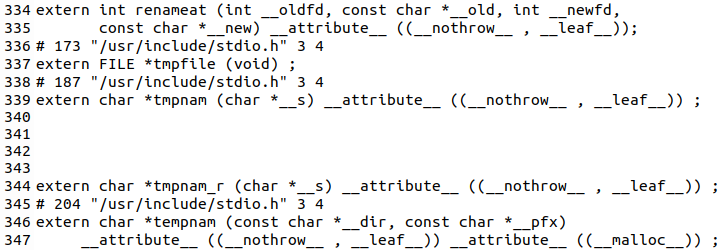


图 2‑6 预处理结果截图4（函数声明信息）

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。

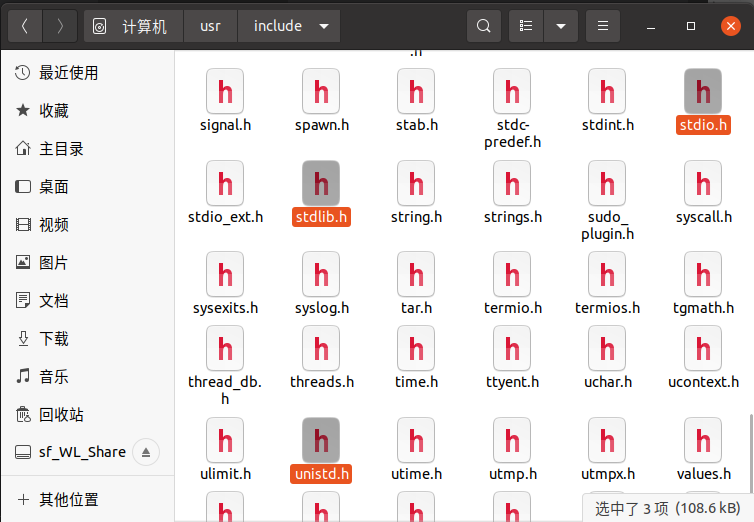


图 2‑7 头文件路径位置截图

上图为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。

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：

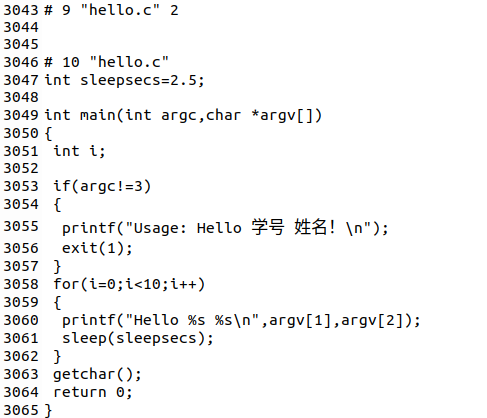


图 2‑8 预处理结果截图5（保留的源代码）

## 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

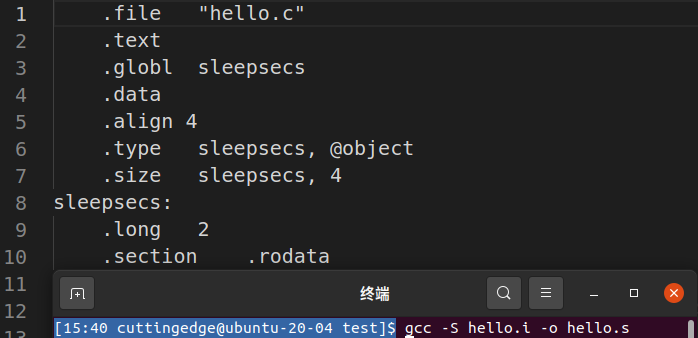


图 3‑1 编译命令执行截图

## 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

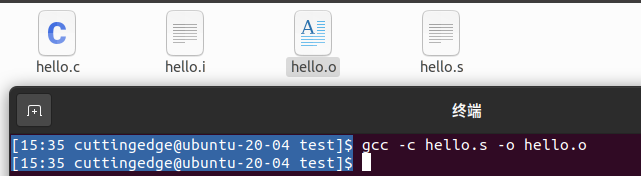


图 4‑1 汇编命令执行截图1

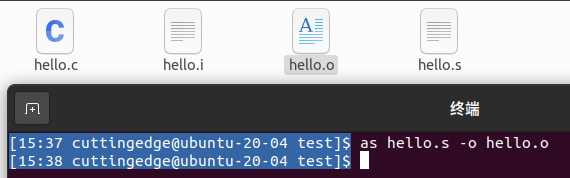


图 4‑2 汇编命令执行截图2

## 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：

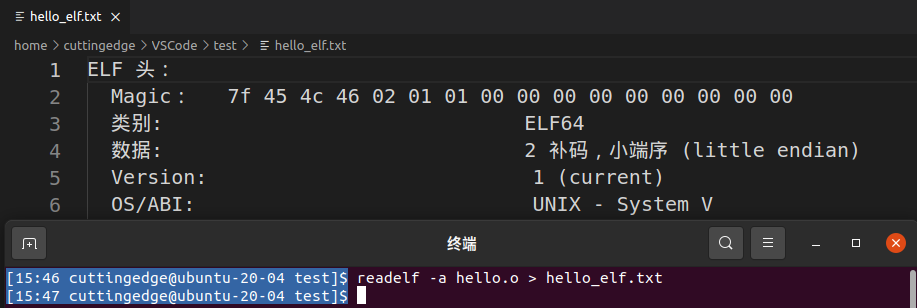


图 4‑3 readelf命令执行截图

### 4.3.2 ELF头

此部分内容（hello\_elf.txt文件第1 ~ 20行）如下：

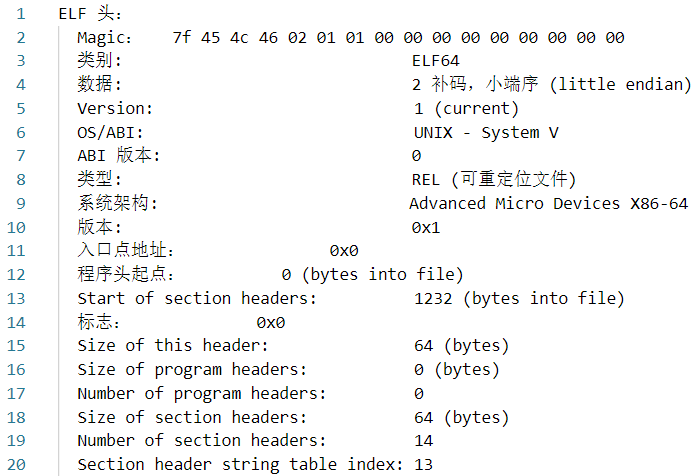


图 4‑4 ELF头信息

**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：

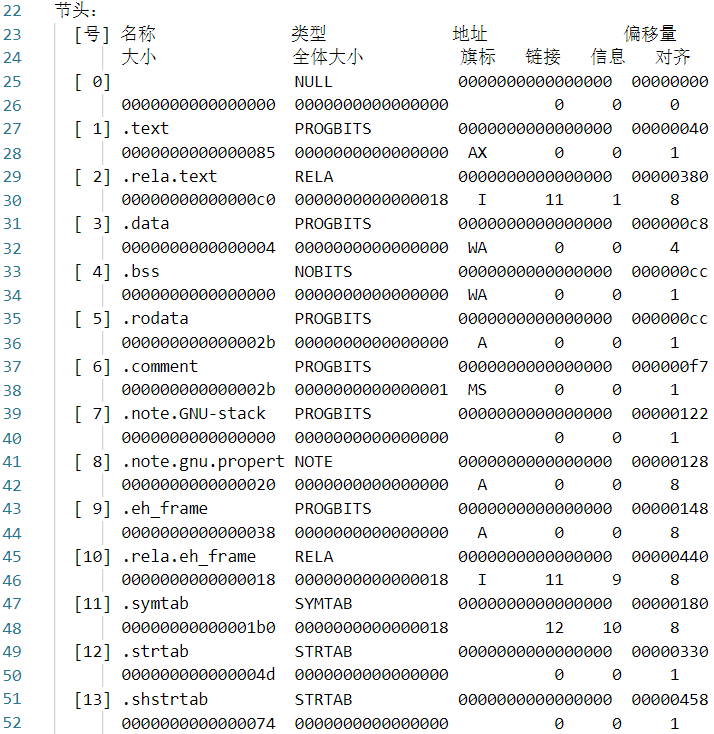


图 4‑5 节头目表信息

**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：

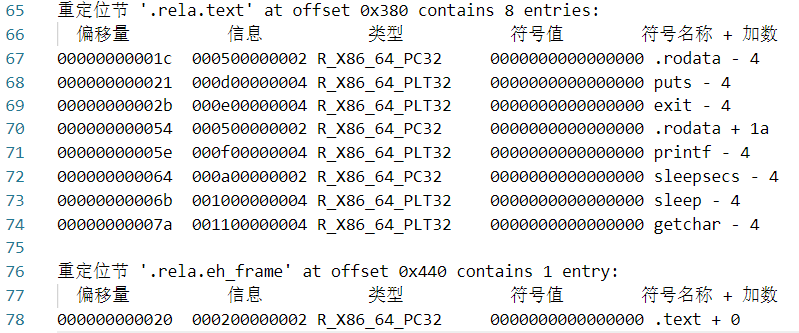


图 4-6 重定位节信息

**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\_32（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：

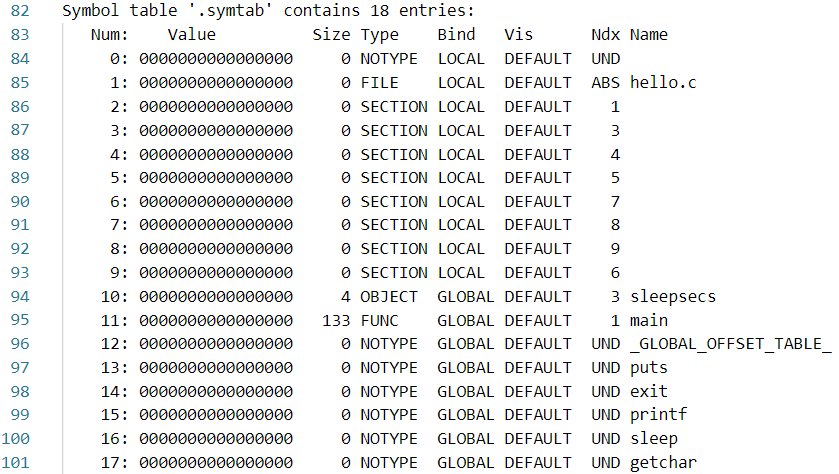


图 4-7 符号表信息

## 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

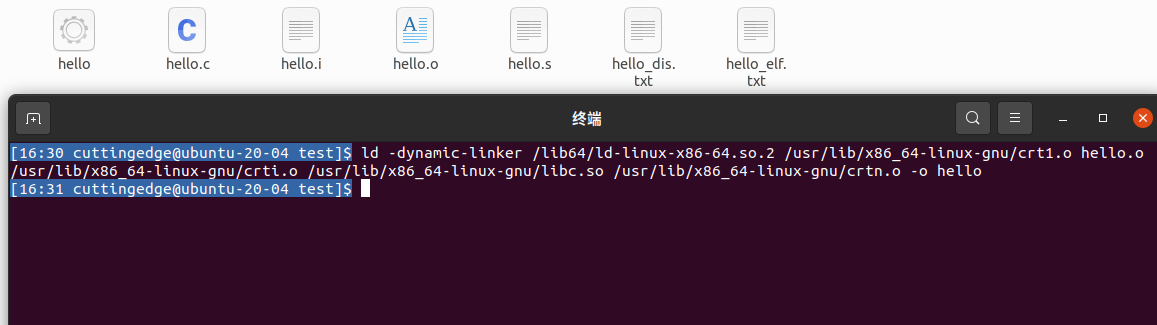


图 5-1 链接命令执行截图

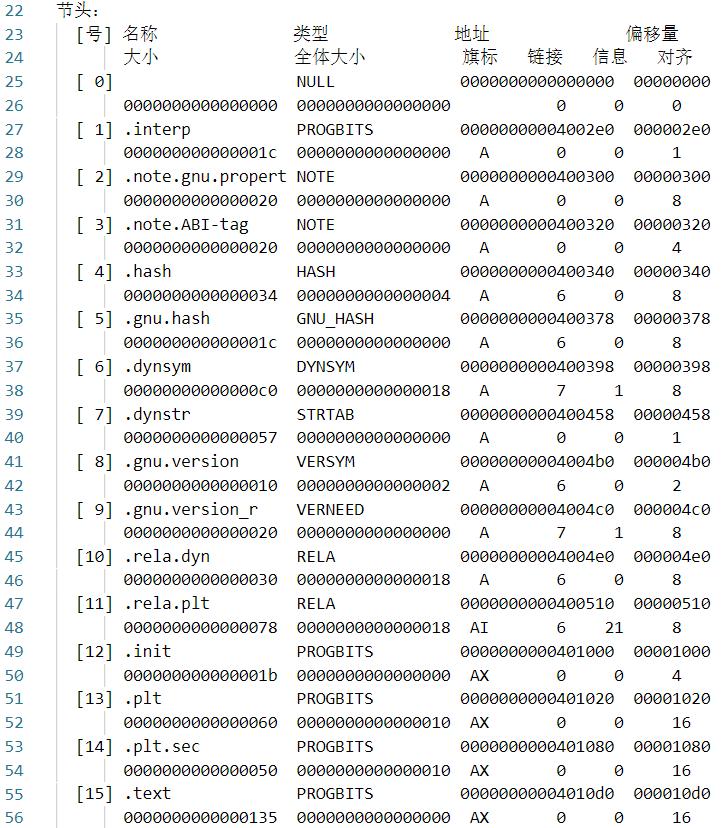
## 5.3 可执行目标文件hello的格式

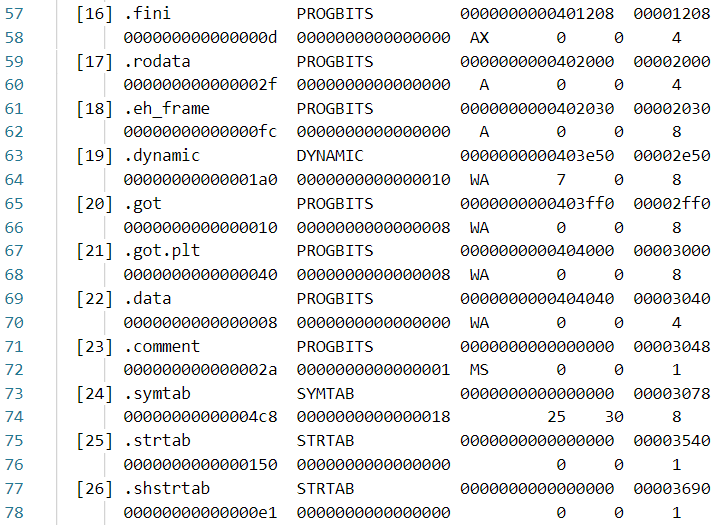
使用命令：readelf -a hello > hello1\_elf.txt

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）

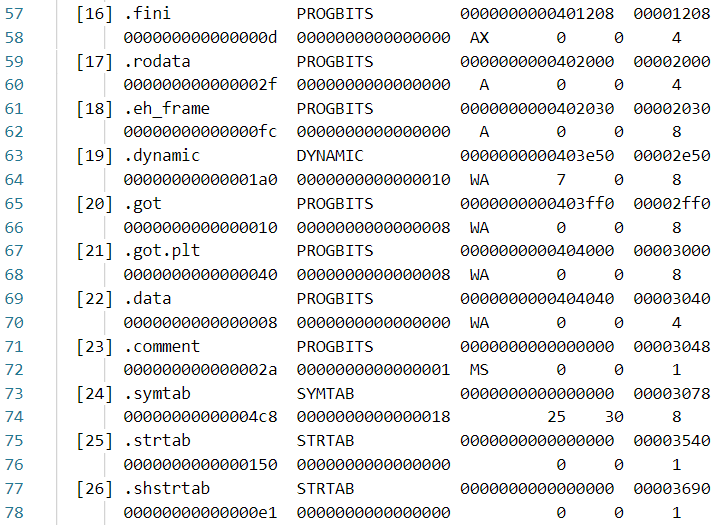


图 5-2 节头部表内容

## 5.4 hello的虚拟地址空间

使用edb加载hello，界面如下图：

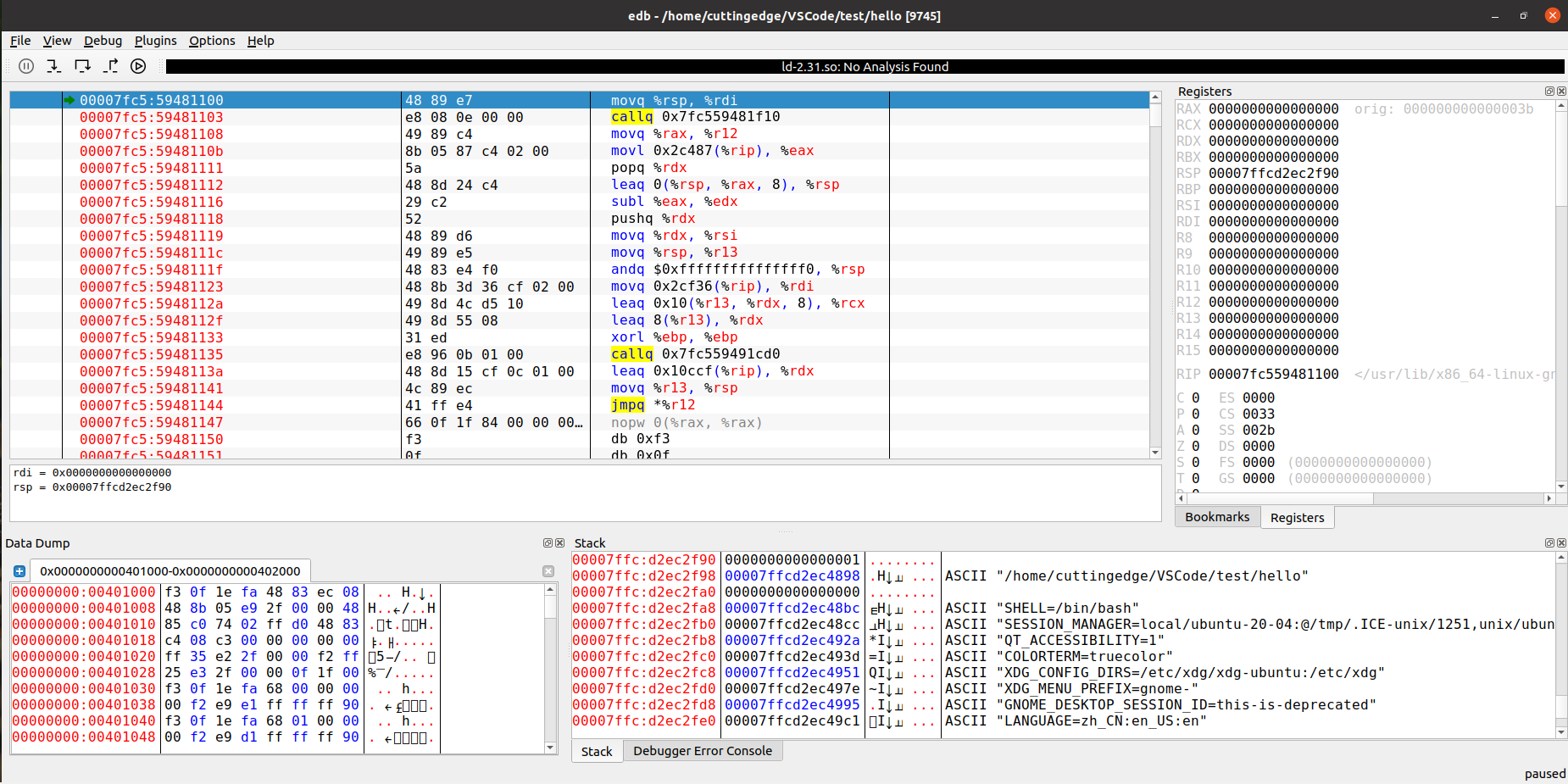


图 5-3 edb加载hello截图

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。

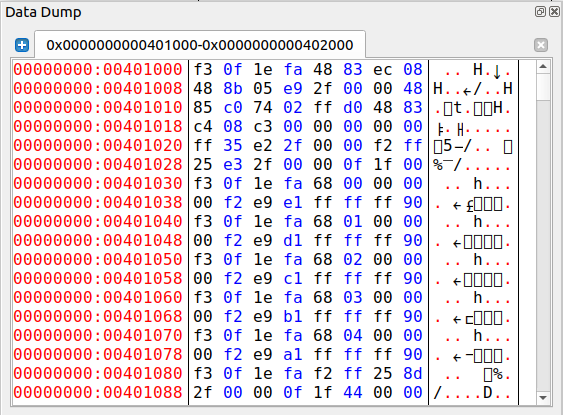


图 5-4 Data Dump窗口

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）：

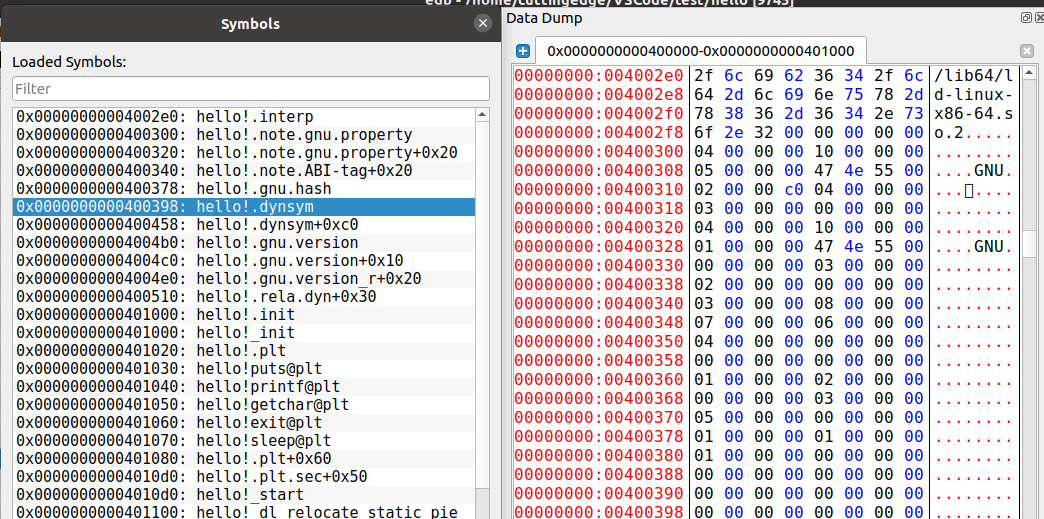


图 5-5 Symbols与Data Dump对照截图1

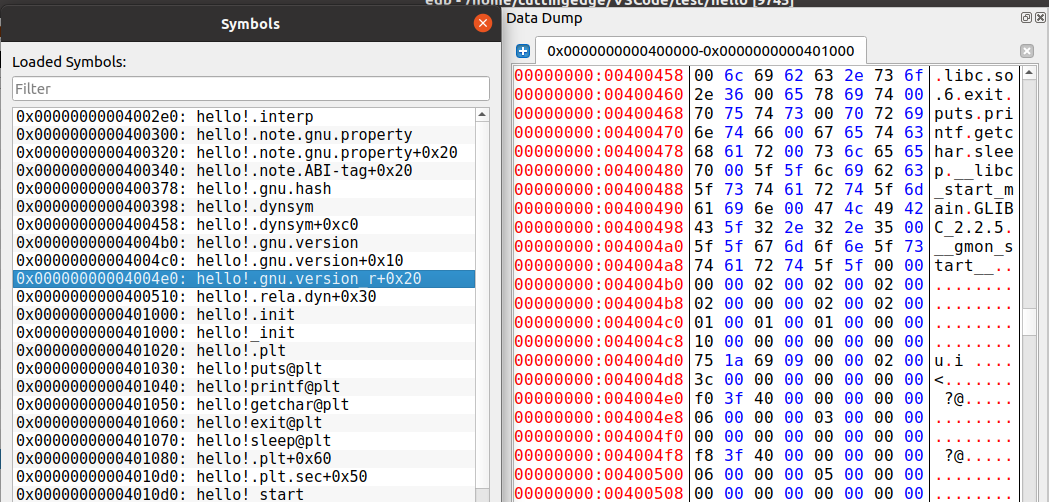


图 5-6 Symbols与Data Dump对照截图2

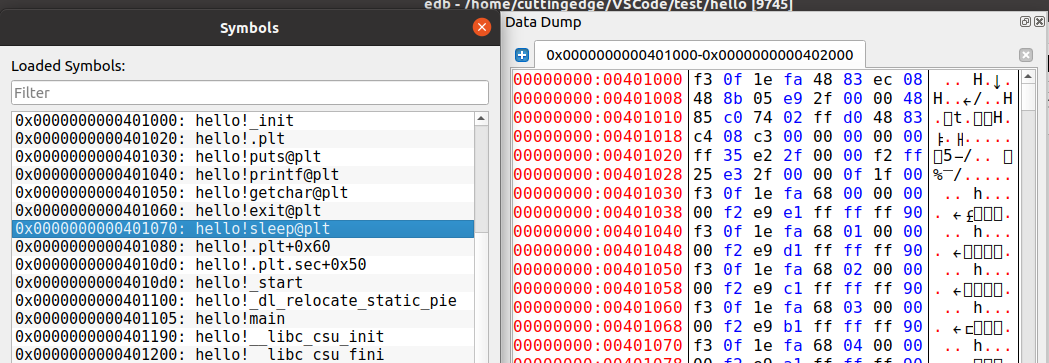


图 5-7 Symbols与Data Dump对照截图3

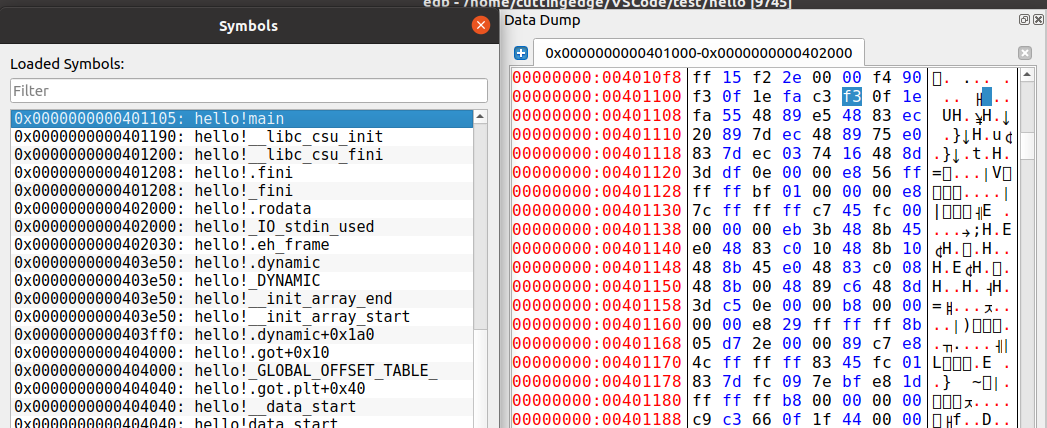


图 5-8 Symbols与Data Dump对照截图4（.text段部分）

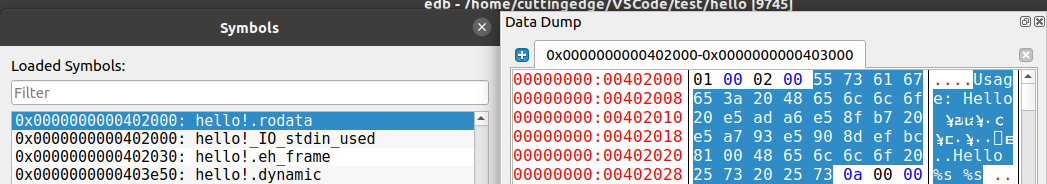


图 5-9 Symbols与Data Dump对照截图5（.rodata段部分）

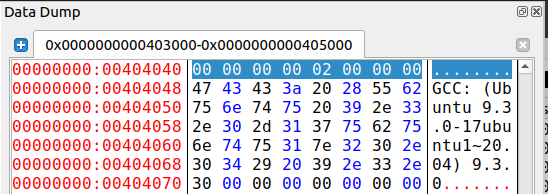


图 5-10 Symbols与Data Dump对照截图6（.data段部分）

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

注：hello的完整反汇编代码见hello1\_dis.txt，hello.o的反汇编代码见4.4节（或hello\_dis.txt文件），此节进行分析和说明。

**分析：**

1. 整体上来看，hello的反汇编代码比hello.o的反汇编代码多了一些节（如.init, .plt, .plt.sec等），如下图所示：

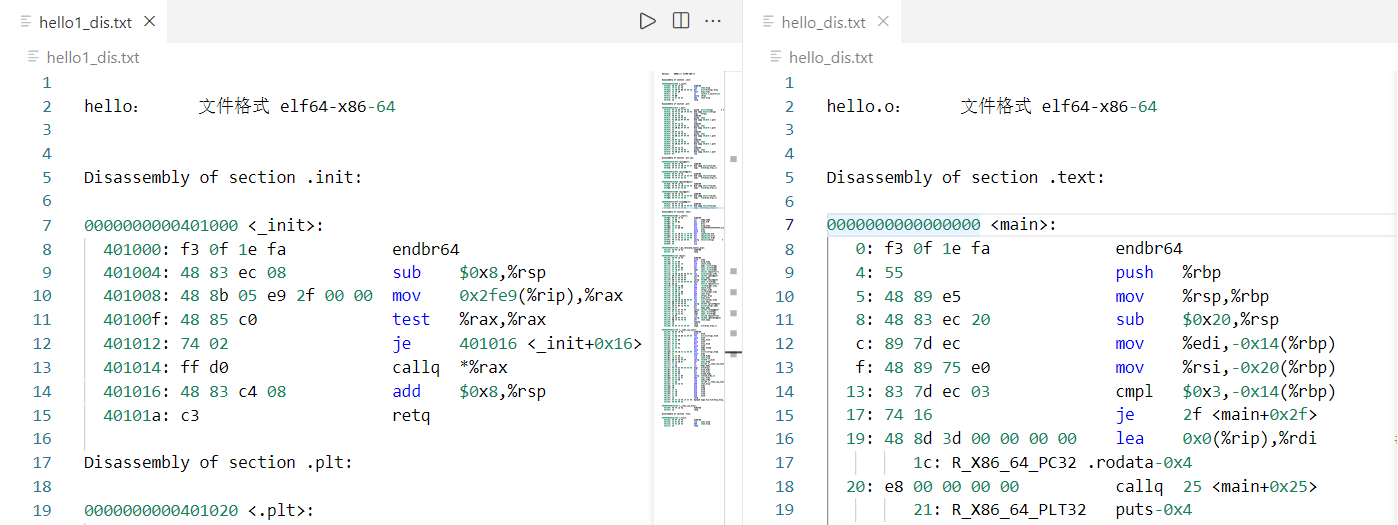


图 5-11 hello相较于hello.o多出了一些节

1. hello中加入了一些函数，如\_init()，\_start()以及一些主函数中调用的库函数，如下图所示：

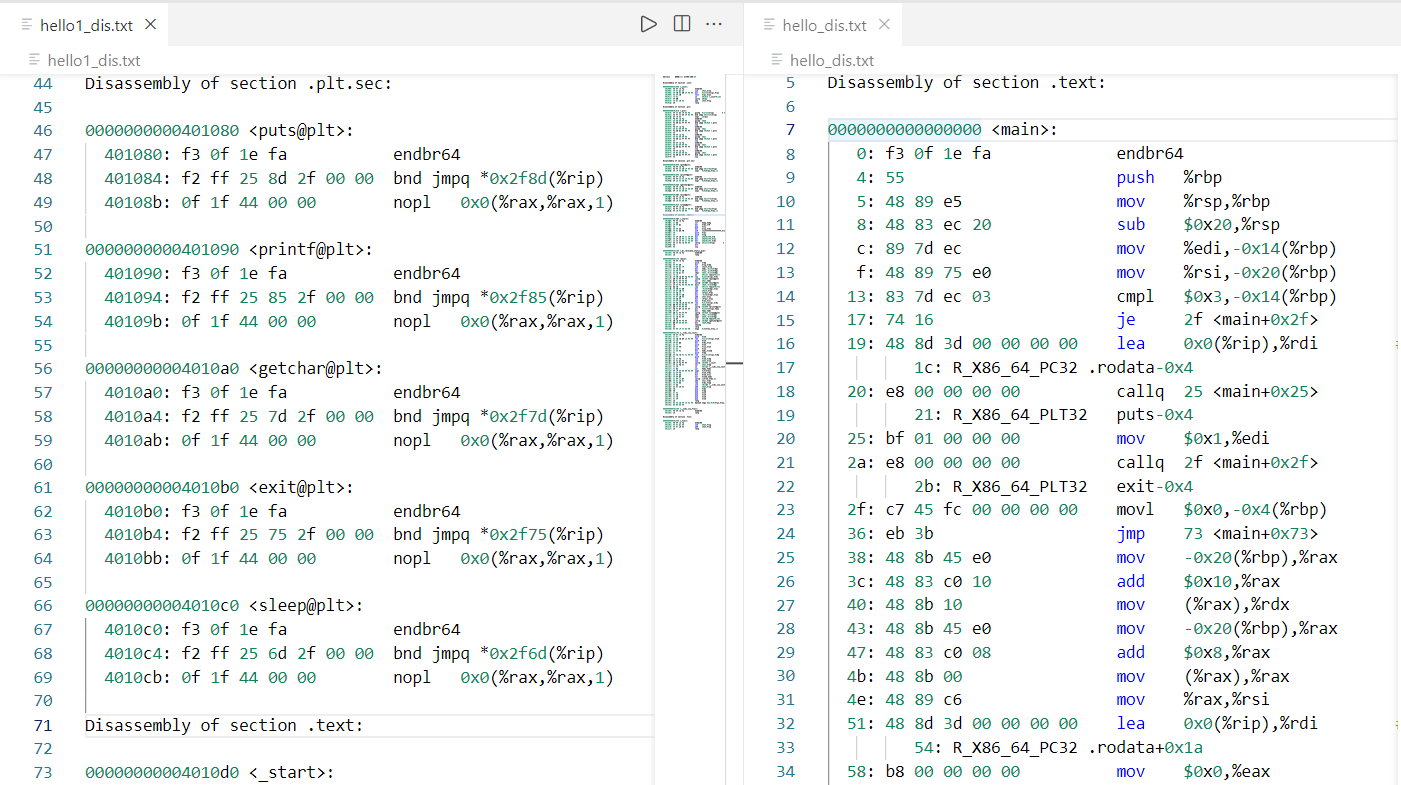


图 5-12 hello相较于hello.o增加了一些函数

1. hello中不再存在hello.o中的重定位条目，并且跳转和函数调用的地址在hello中都变成了虚拟内存地址，如下图：

注：图见下页，图中使用相同颜色代表同一跳转语句或函数调用语句中重定位条目(hello\_dis.txt)及处理后的虚拟空间地址(hello1\_dis.txt)。

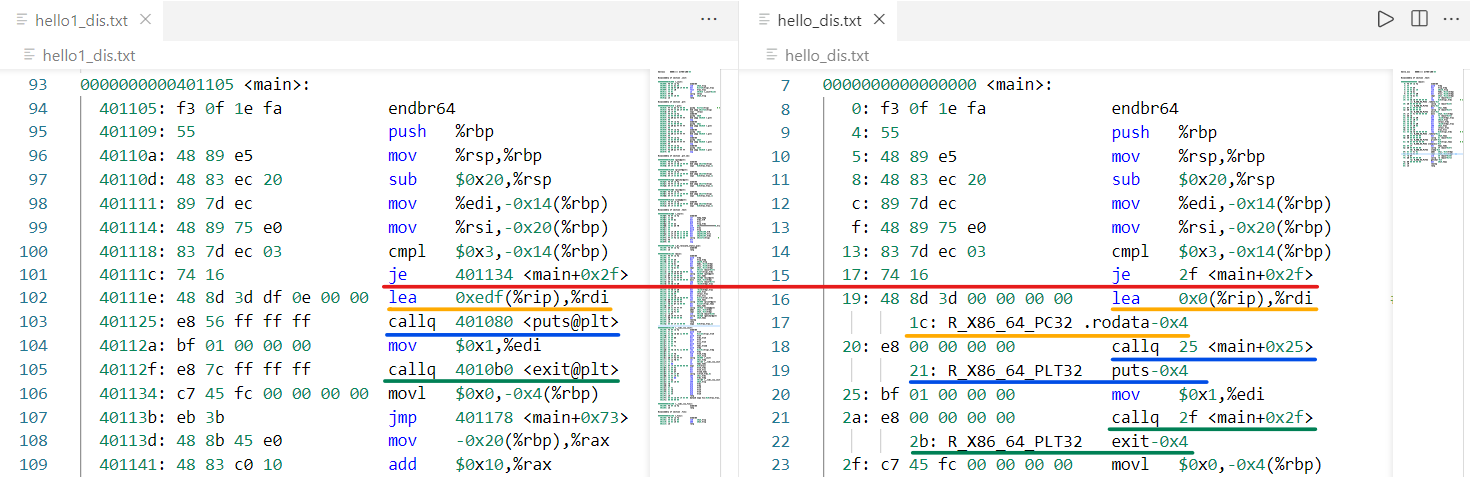


图 5-13 hello对于函数调用和跳转语句中重定向的处理

根据以上分析我们可以看出，链接过程会扫描分析所有相关的可重定位目标文件，并完成两个主要任务：首先进行符号解析，将每个符号引用与一个符号定义关联起来；随后进行重定位，链接器使用汇编器产生的重定位条目的详细指令，把每个符号定义与一个内存位置关联起来。最终的结果是将程序运行所需的各部分组装在一起，形成一个可执行目标文件。

## 5.6 hello的执行流程

如下图，在EDB执行hello前添加程序参数（相当于在终端中输入./hello 1190200526 沈城有）：

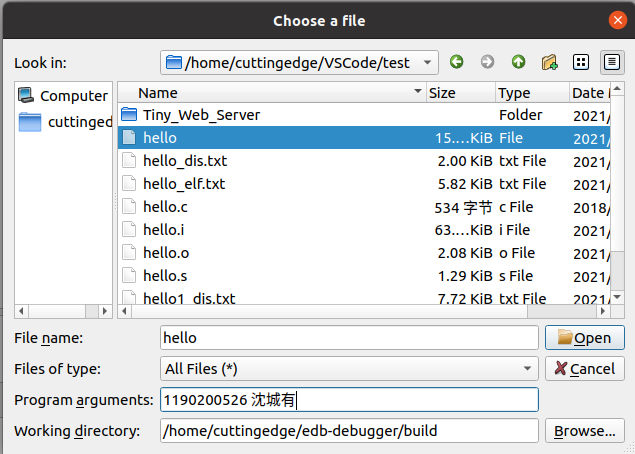


图 5-14 EDB执行hello前添加程序参数

使用EDB跟踪程序执行过程，按顺序记录如下：

|  |  |
| --- | --- |
| 子程序名 | 地址（16进制） |
| ld-2.31.so!\_dl\_start | 0x7f3d92088100 |
| ld-2.31.so!\_dl\_init | 0x7f3d92088f10 |
| hello!\_start | 0x4010d0 |
| libc-2.31.so!\_\_libc\_start\_main | 0x7ff0b4eaafc0 |
| hello!printf@plt（调用10次） | 0x401090 |
| hello!sleep@plt（调用10次） | 0x4010c0 |
| hello!getchar@plt | 0x4010a0 |
| libc-2.31.so!exit | 0x7faf2419bbc0 |

## 5.7 Hello的动态链接分析

下图为调用dl\_init之前.got.plt段的内容：

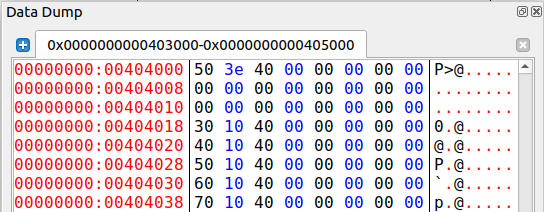


图 5-15 调用dl\_init之前.got.plt段的内容

下图为调用dl\_init之后.got.plt段的内容：

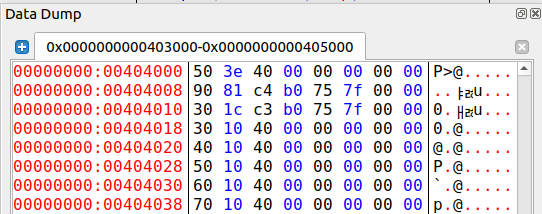


图 5-16 调用dl\_init之后.got.plt段的内容

可以很明显地看出第2、3行的变化。

实际上，这是书上（P490）提到的动态链接器的延迟绑定的初始化部分。延迟绑定通过全局偏移量表(GOT)和过程链接表(PLT)的协同工作实现函数的动态链接，其中GOT中存放函数目标地址，PLT使用GOT中地址跳转到目标函数。

在此之后，程序调用共享库函数时，会首先跳转到PLT执行指令，第一次跳转时，GOT条目为PLT下一条指令，将函数ID压栈，然后跳转到PLT[0]，在PLT[0]再将重定位表地址压栈，然后转进动态链接器，在动态链接器中使用两个栈条目确定函数运行时地址，重写GOT，再将控制传递给目标函数。以后如果再次调用同一函数，则通过间接跳转将控制直接转移至目标函数。

## 5.8 本章小结

本章围绕可重定位目标文件hello.o链接生成可执行目标文件hello的过程，首先详细介绍、分析了链接的概念、作用及具体工作。随后验证了hello的虚拟地址空间与节头部表信息的对应关系，分析了hello的执行流程。最后对hello程序进行了动态链接分析。在此过程中，我更加深刻地理解了链接和重定位的相关概念，复习了课程第7章的相关知识，了解了动态链接的过程及作用。

# 第6章 hello进程管理

## 6.1 进程的概念与作用

1. **进程的概念**

进程的经典定义是一个执行中程序的实例，是操作系统对一个正在运行的程序的一种抽象。

1. **进程的作用**

每次用户通过shell输入一个可执行目标文件的名字，运行程序时，shell就会创建一个新的进程，然后在这个新进程的上下文中运行这个可执行目标文件。应用程序也能够创建新进程，并且在这个新进程的上下文中运行它们自己的代码或其他应用程序。

进程提供给应用程序的关键抽象：一个独立的逻辑控制流，如同程序独占地使用处理器；一个私有的地址空间，如同程序独占地使用内存系统。

## 6.2 简述壳Shell-bash的作用与处理流程

**作用**：Shell-bash是一个交互型应用级程序，代表用户运行其他程序。它是系统的用户界面，提供了用户与内核进行交互操作的一种接口。它接收用户输入的命令并把它送入内核去执行。

**处理流程**：

shell首先检查命令是否是内部命令，若不是再检查是否是一个应用程序（这里的应用程序可以是Linux本身的实用程序，如ls和rm，也可以是购买的商业程序，如xv，或者是自由软件，如emacs）。然后shell在搜索路径里寻找这些应用程序（搜索路径就是一个能找到可执行程序的目录列表）。如果键入的命令不是一个内部命令并且在路径里没有找到这个可执行文件，将会显示一条错误信息。如果能够成功找到命令，该内部命令或应用程序将被分解为系统调用并传给Linux内核。

简化处理流程：

1. 从终端读入输入的命令；
2. 将输入字符串切分获得所有的参数；
3. 如果是内置命令则立即执行；
4. 若不是则调用相应的程序执行；
5. shell应该随时接受键盘输入信号，并对这些信号进行相应处理。

## 6.3 Hello的fork进程创建过程

根据shell的处理流程，键入命令（./hello 1190200526 沈城有）后，shell判断其不是内部指令，即会通过fork函数创建子进程。子进程与父进程近似，会得到一份与父进程用户级虚拟空间相同且独立的副本——包括数据段、代码、共享库、堆和用户栈等，父进程打开的文件，子进程也可读写。二者之间最大的不同在于PID的不同。fork函数被调用一次会返回两次，在父进程中，fork函数返回子进程的PID，在子进程中，fork函数返回0。

流程示意：

     +------------>新的子进程

     |

     |

-----+------------>父进程(shell)

   fork

## 6.4 Hello的execve过程

execve函数加载并运行可执行目标文件hello，且带参数列表argv和环境变量列表envp。只有出现错误时（例如找不到可执行目标文件hello），execve才会返回到调用程序，这里与调用一次返回两次的fork函数不同。

execve函数在加载了hello之后，它调用启动代码。启动代码设置栈，并将控制传递给新程序的主函数，该主函数原型如下：

int main(int argc, char \*\*argv, char \*envp)

execve函数的执行过程会覆盖当前进程的地址空间，但并没有创建一个新进程。新的程序仍然有相同的PID，并且继承了调用execve函数时已打开的所有文件描述符。

注：exceve过程中具体的内存映射可见本文7.7节。

## 6.5 Hello的进程执行

**上下文**：

内核重新启动一个被抢占的进程所需要恢复的原来的状态，由寄存器、程序计数器、用户栈、内核栈和内核数据结构等对象的值构成。

**进程上下文切换：**

在内核调度了一个新的进程运行时，它就抢占当前进程，并使用一种上下文切换的机制来控制转移到新的进程。具体过程为：①保存当前进程的上下文；②恢复某个先前被抢占的进程被保存的上下文；③将控制传递给这个新恢复的进程。

**进程时间片**：

一个进程执行它的控制流的一部分的每一个时间段叫做时间片(time slice)，多任务也叫时间分片(time slicing)。

**进程调度：**

在进程执行的某些时刻，内核可以决定抢占当前进程，并重新开始一个先前被抢占的进程，这种决策称为调度，是由内核中的调度器代码处理的。当内核选择一个新的进程运行，我们说内核调度了这个进程。在内核调度了一个新的进程运行了之后，它就抢占了当前进程，并使用上下文切换机制来将控制转移到新的进程。

hello程序调用sleep函数休眠时，内核将通过进程调度进行上下文切换，将控制转移到其他进程。当hello程序休眠结束后，进程调度使hello程序重新抢占内核，继续执行。

**用户态与核心态的转换：**

为了保证系统安全，需要限制应用程序所能访问的地址空间范围。因而存在用户态与核心态的划分，核心态拥有最高的访问权限，而用户态的访问权限会受到一些限制。处理器使用一个寄存器作为模式位来描述当前进程的特权。进程只有故障、中断或陷入系统调用时才会得到内核访问权限，其他情况下始终处于用户权限之中，一定程度上保证了系统的安全性。

**hello程序进程执行示意图：**

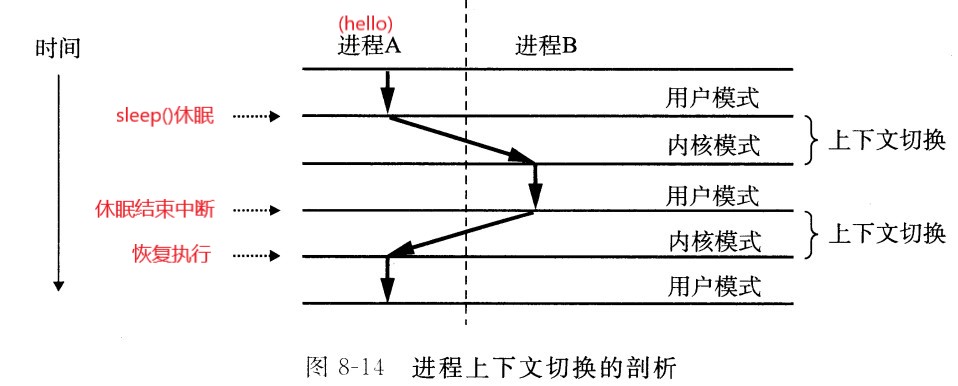
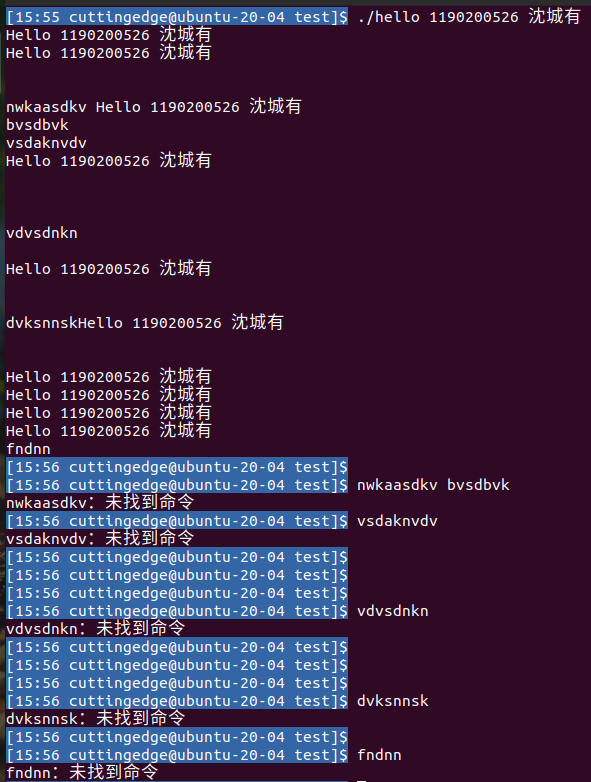


图 6-1 hello程序进程执行过程示意

## 6.6 hello的异常与信号处理

**执行截图：**



（图接上页）

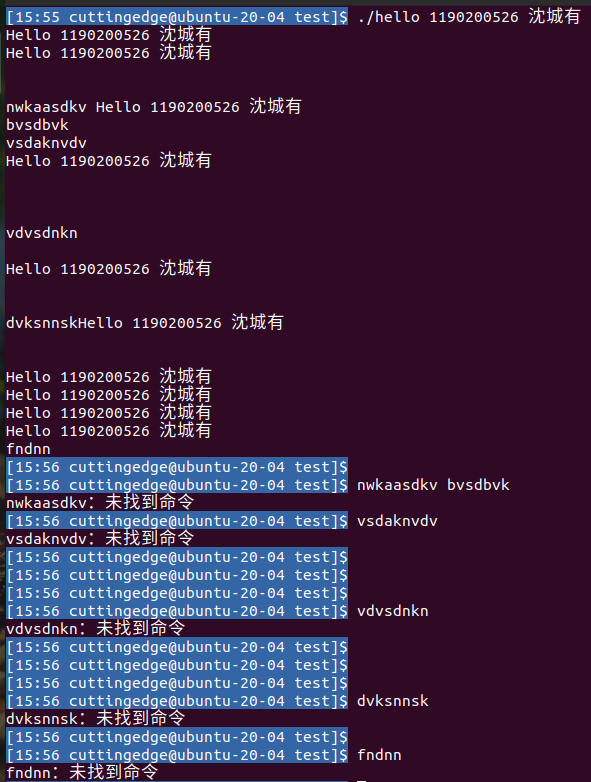


图 6-2 执行过程中随意输入，被认为是命令



图 6-3 键入Ctrl-C，程序终止

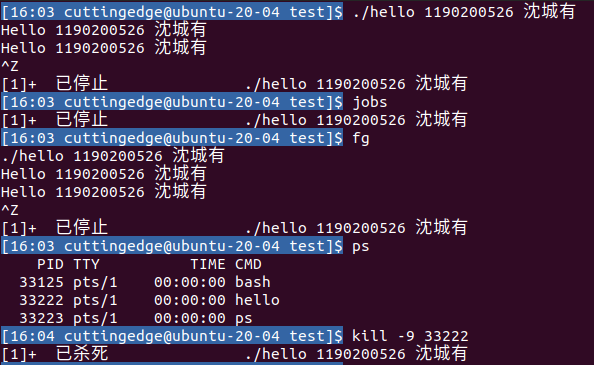


图 6-4 键入Ctrl-Z后jobs、fg

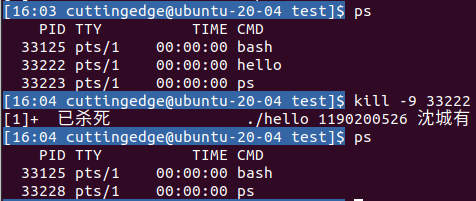


图 6-5 kill命令终止hello进程



图 6-6 终止hello进程前pstree结果（部分）



图 6-7 终止hello进程后pstree结果（部分）

**hello执行过程中的异常及处理：**

1. 中断：来自I/O设备的信号（Ctrl-C、Ctrl-Z），异步；

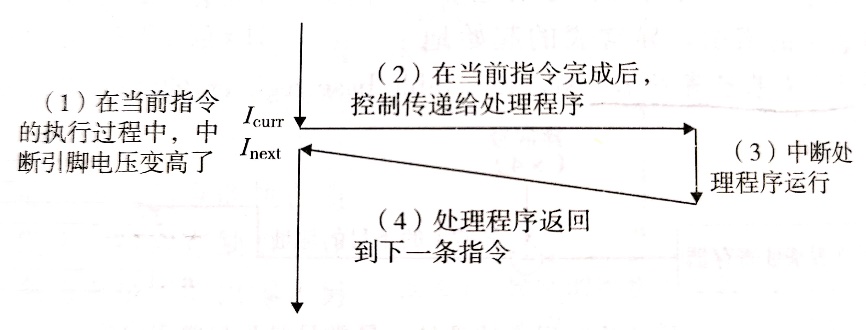


图 6-8 中断处理

1. 陷阱：有意的异常，执行指令的结果（例如：exit），同步；

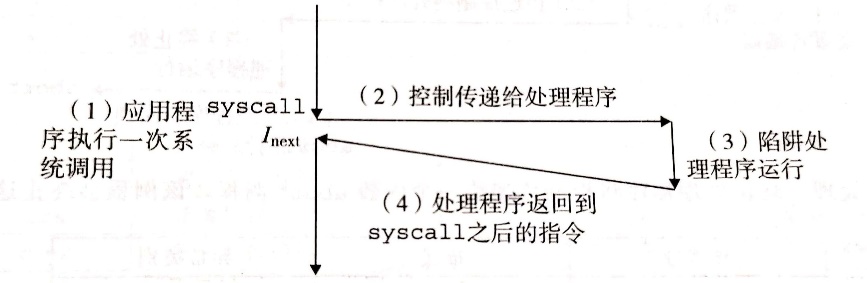


图 6-9 陷阱处理

1. 故障：潜在可恢复的错误（如：缺页异常），同步。

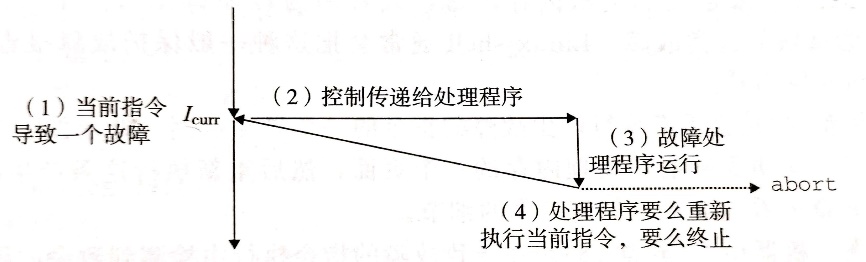


图 6-10 故障处理

**hello执行过程中的信号及处理：**

* SIGINT：键入Ctrl-C后内核向hello进程发送，终止程序。
* SIGSTP：键入Ctrl-Z后内核向hello进程发送，停止直到下一个SIGCONT。
* SIGCONT：键入fg后内核向hello进程发送，若停止则继续执行。
* SIGKILL：键入kill -9 <PID>后内核向hello进程发送，终止程序。

## 6.7本章小结

本章主要阐述了hello的进程管理，包括进程创建、加载、执行以至终止的全过程，并分析了执行过程中的异常、信号及其处理。在hello程序运行的过程中，内核对其进行进程管理，决定何时进行进程调度，在接收到不同的异常、信号时，还要及时地进行对应的处理。本章的内容引导我复习了课程第8章——异常控制流的相关内容，使我对进程、信号及异常相关概念的理解更加深刻。

# 第7章 hello的存储管理

## 7.1 hello的存储器地址空间

1. **逻辑地址**

逻辑地址（Logical Address）是指由程序产生的与段相关的偏移地址部分，是相对应用程序而言的，如hello.o中代码与数据的相对偏移地址。

1. **线性地址**

线性地址（Linear Address）是逻辑地址到物理地址变换之间的中间层。逻辑地址加上相应段的基地址就生成了一个线性地址，如hello中代码与数据的地址。

1. **虚拟地址**

有时我们也把逻辑地址称为虚拟地址（Virtual Address）。因为与虚拟内存空间的概念类似，逻辑地址也是与实际物理内存容量无关的，是hello中的虚拟地址。

1. **物理地址**

物理地址（Physical Address）是指出现在CPU外部地址总线上的寻址物理内存的地址信号，是地址变换的最终结果地址（hello程序运行时代码、数据等对应的可用于直接在内存中寻址的地址）。如果启用了分页机制（当今绝大多数计算机的情况），那么线性地址会使用页目录和页表中的项变换成物理地址；如果没有启用分页机制，那么线性地址就直接成为物理地址了。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

为了运用所有的内存空间，Intel 8086设定了四个段寄存器，专门用来保存段地址：CS（Code Segment）：代码段寄存器；DS（Data Segment）：数据段寄存器；SS（Stack Segment）：堆栈段寄存器；ES（Extra Segment）：附加段寄存器。

当一个程序要执行时，就要决定程序代码、数据和堆栈各要用到内存的哪些位置，通过设定段寄存器CS，DS，SS来指向这些起始位置。通常是将DS固定，而根据需要修改CS。所以，程序可以在可寻址空间小于64K的情况下被写成任意大小。所以，程序和其数据组合起来的大小，限制在DS所指的64K内，这就是COM文件不得大于64K的原因。

段寄存器是因为对内存的分段管理而设置的。

计算机需要对内存分段，以分配给不同的程序使用（类似于硬盘分页）。在描述内存分段时，需要有如下段的信息：1.段的大小；2.段的起始地址；3.段的管理属性（禁止写入/禁止执行/系统专用等）。

* 保护模式（如今大多数机器已经不再支持）：

段寄存器的唯一目的是存放段选择符，其前13位是一个索引号，后面3位包含一些硬件细节（还有一些隐藏位，此处略）。

寻址方式为：以段选择符作为下标，到GDT/LDT表（全局段描述符表(GDT)和局部段描述符表(LDT)）中查到段地址，段地址+偏移地址=线性地址。

* 实模式：

段寄存器含有段值，访问存储器形成物理地址时，处理器引用相应的某个段寄存器并将其值乘以16，形成20位的段基地址，段基地址·段偏移量=线性地址。

## 7.3 Hello的线性地址到物理地址的变换-页式管理

VM系统通过将虚拟内存分割为称为虚拟页的大小固定的块来处理这个问题，每个虚拟页的大小（一般为4096字节）。类似地，物理内存被分割成物理页，大小也为P字节（物理页也称为页帧）。

下图展示了线性地址（虚拟地址）到物理地址的变换过程：

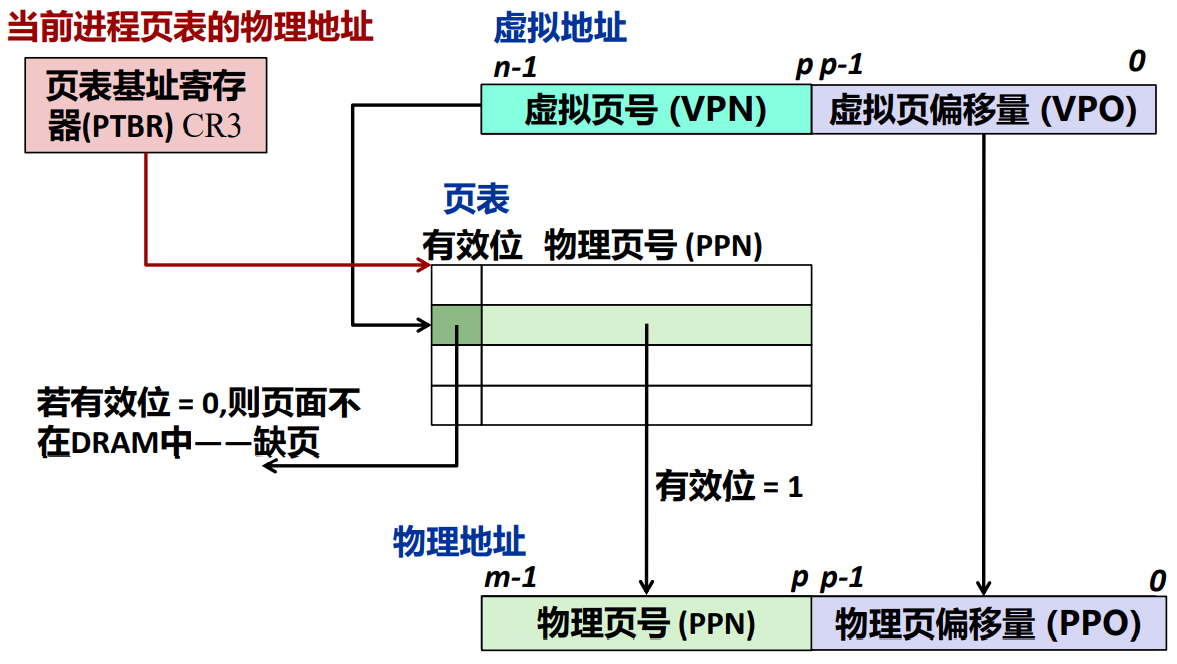


图 7-1 线性地址到物理地址的变换过程示意图

hello进程执行时，CPU中的页表基址寄存器指向hello进程的页表，当hello进程访问其虚拟空间内的指令、数据等内容时，CPU芯片上的MMU（内存管理单元）会将对应的线性地址变换为物理地址以进行寻址访问。

n位（Core i7为48位）的线性地址包含两部分：一个p位的虚拟页面偏移（VPO）和一个(n-p)位的虚拟页号。MMU利用虚拟页号（VPN）来选择适当的PTE（页表项），若PTE有效位为1，则说明其后内容为物理页号（PPN），否则缺页。而物理地址中低p位的物理页偏移量（PPO）与虚拟页偏移量（VPO）相同，PPN与PPO连接即得物理地址。

## 7.4 TLB与四级页表支持下的VA到PA的变换

**TLB：**

每次CPU产生一个虚拟地址，MMU（内存管理单元）就必须查阅一个PTE（页表条目），以便将虚拟地址翻译为物理地址。在最糟糕的情况下，这会从内存多取一次数据，代价是几十到几百个周期。如果PTE碰巧缓存在L1中，那么开销就会下降到1或2个周期。然而，许多系统都试图消除即使是这样的开销，它们在MMU中包括了一个关于PTE的小的缓存，称为翻译后备缓存器（TLB）。

**多级页表：**

多级页表为层次结构，用于压缩页表。这种方法从两个方面减少了内存要求。第一，如果一级页表中的一个PTE是空的，那么相应的二级页表就根本不会存在；第二，只有一级页表才需要总是在主存中，虚拟内存系统可以在需要时创建、页面调出或调入二级页表，最经常使用的二级页表才缓存在主存中，减少了主存的压力。

**VA到PA的变换：**

对于四级页表，虚拟地址（VA）被划分为4个VPN和1个VPO。每个VPN　i都是一个到第i级页表的索引。对于前3级页表，每级页表中的每个PTE都指向下一级某个页表的基址。最后一级页表中的每个PTE包含某个物理页面的PPN，或者一个磁盘块的地址。为了构造物理地址，在能够确定PPN之前，MMU必须访问k个PTE。和只有一级的页表结构一样，PPO和VPO是相同的。

示意图如下（Core i7为例）：

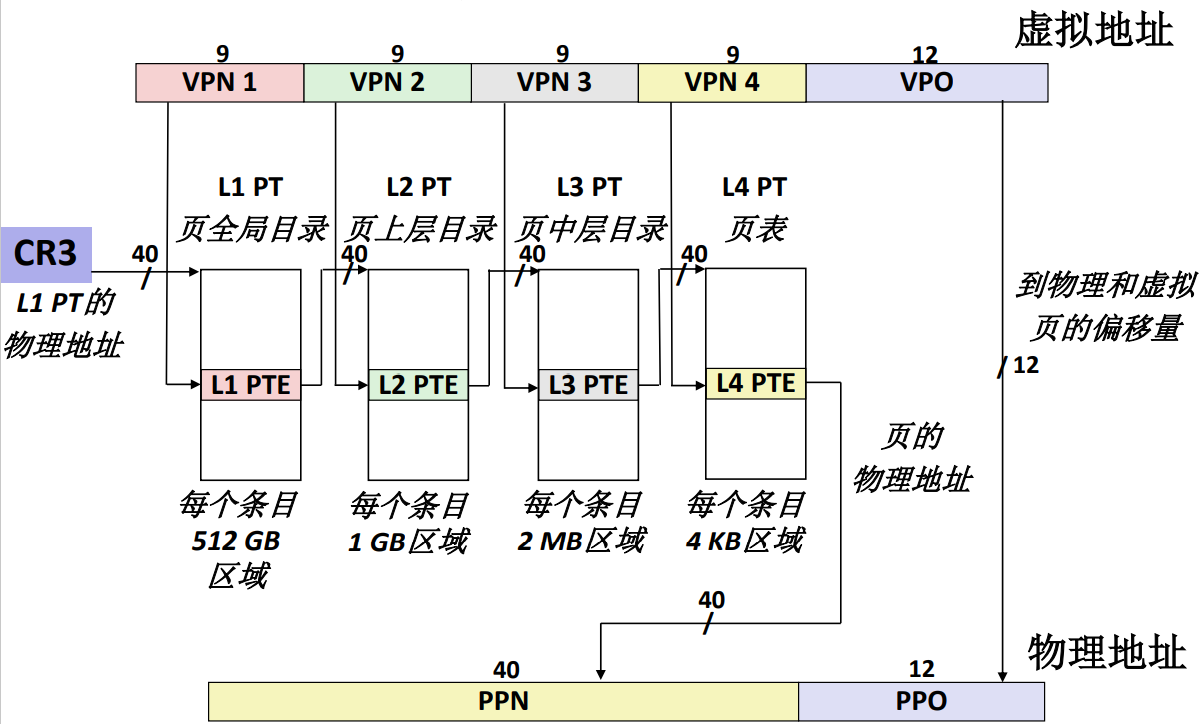


图 7-2 VA到PA的变换示意图

## 7.5 三级Cache支持下的物理内存访问

下图为通用的高速缓存存储器（Cache）组织结构示意图：

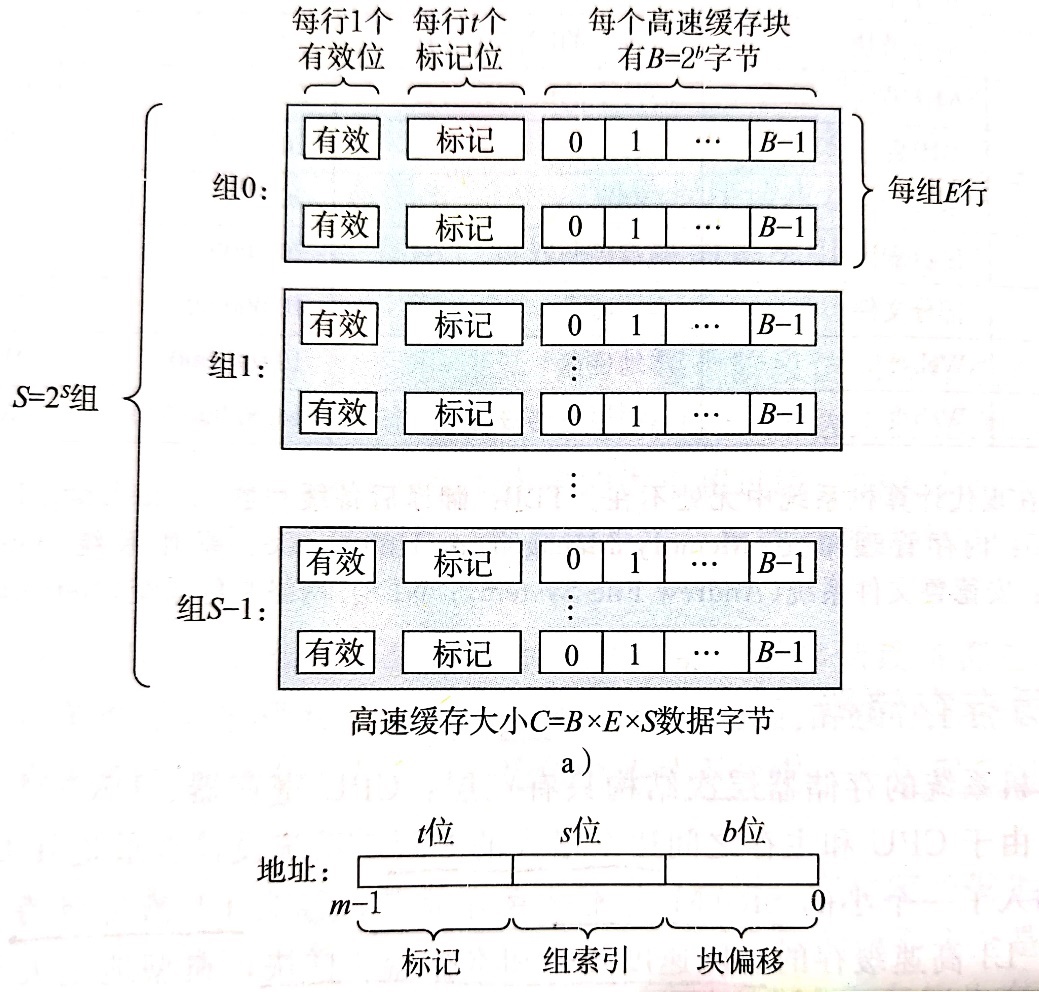


图 7-3 高速缓存存储器组织结构示意

1. 根据PA、L1高速缓存的组数和块大小确定高速缓存块偏移（CO）、组索引（CI）和高速缓存标记（CT），使用CI进行组索引，对组中每行的标记与CT进行匹配。如果匹配成功且块的valid标志位为1，则命中，然后根据CO取出数据并返回数据给CPU。
2. 若未找到相匹配的行或有效位为0，则L1未命中，继续在下一级高速缓存（L2）中进行类似过程的查找。若仍未命中，还要在L3高速缓存中进行查找。三级Cache均未命中则需访问主存获取数据。
3. 若进行了(2)步，说明至少有一级高速缓存未命中，则需在得到数据后更新未命中的Cache。首先判断其中是否有空闲块，若有空闲块（有效位为0），则直接将数据写入；若不存在，则需根据替换策略（如LRU、LFU策略等）驱逐一个块再写入。

## 7.6 hello进程fork时的内存映射

当fork函数被父进程（shell）调用时，内核为新进程（未来加载执行hello的进程）创建各种数据结构，并分配给它一个唯一的PID。为了给这个新进程创建虚拟内存，它创建了当前进程的mm\_struct、区域结构和页表的原样副本。它将两个进程中的每个页面都标记为只读，并将两个进程中的每个区域结构都标记为私有的写时复制。

当fork在新进程中返回时，新进程现在的虚拟内存刚好和调用fork时存在的虚拟内存相同。当这两个进程中的任一个后来进行写操作时，写时复制机制就会创建新页面，因此，也就为每个进程保持了私有空间地址的抽象概念。

## 7.7 hello进程execve时的内存映射

execve加载和运行hello程序会经过以下步骤：

* 删除已存在的用户区域：这里指在fork后创建于此进程用户区域中的shell父进程用户区域副本。
* 映射私有区域：为hello程序的代码、数据、bss和栈区域创建新的区域结构，所有这些新的区域都是私有的、写时复制的。代码和数据区域被映射到hello可执行文件中的.text和.data区。bss区域是请求二进制零的，映射到匿名文件，其大小包含在hello文件中。栈和堆区域也是请求二进制零的，初始长度为零。
* 映射共享区域：hello程序与一些共享对象或目标链接，比如标准C库libc.so，那么这些对象都是动态链接到这个程序的，然后再映射到用户虚拟地址空间中的共享区域内。
* 设置程序计数器（PC）：设置此进程上下文中的程序计数器，使之指向hello代码区域的入口点。

## 7.8 缺页故障与缺页中断处理

在虚拟内存的习惯说法中，DRAM缓存不命中称为缺页。缺页故障属于异常类别中的故障，是潜在可恢复的错误，主要处理流程可见本文6.6节中关于故障处理的部分。

缺页异常调用内核中的缺页异常处理程序，该程序会选择一个牺牲页，如果牺牲页已经被修改了，内核会将其复制回磁盘。随后内核从磁盘复制引发缺页异常的页面至内存，更新对应的页表项指向这个页面，随后返回。

缺页异常处理程序返回后，内核会重新启动导致缺页的指令，该指令会把导致缺页的虚拟地址重发送到地址翻译硬件，此次页面会命中。

下图为缺页异常处理过程示意图：

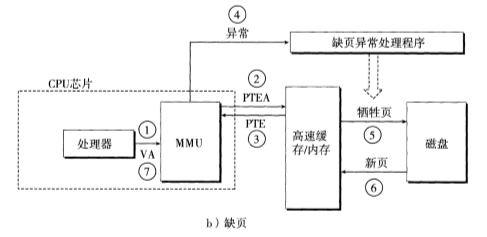


图 7-4 缺页异常处理过程示意

## 7.9动态内存分配管理

动态内存分配管理使用动态内存分配器来进行。动态内存分配器维护着一个进程的虚拟内存区域，称为堆。分配器将堆视为一组不同大小的块的集合来维护，每个块就是一个连续的虚拟内存片，要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可以用来分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配的状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

分配器有两种风格——显式分配器和隐式分配器。C语言中的malloc程序包是一种显式分配器。显式分配器必须在一些相当严格的约束条件下工作：①处理任意请求序列；②立即响应请求；③只使用堆；④对齐块（对齐要求）；⑤不修改已分配的块。在以上限制条件下，分配器要最大化吞吐率和内存使用率。

常见的放置策略：

* 首次适配：从头开始搜索空闲链表，选择第一个合适的空闲块。
* 下一次适配：类似于首次适配，但从上一次查找结束的地方开始搜索。
* 最佳适配：选择所有空闲块中适合所需请求大小的最小空闲块。

这里简要介绍一些组织内存块的方法：

1. 隐式空闲链表：空闲块通过头部中大小字段隐含连接，可添加边界标记提高合并空闲块的速度。
2. 显式空闲链表：在隐式空闲链表块结构的基础上，在每个空闲块中添加一个pred（前驱）指针和一个succ（后继）指针。
3. 分离的空闲链表：将块按块大小划分大小类，分配器维护一个空闲链表数组，每个大小类一个空闲链表，减少分配时间同时也提高了内存利用率。C语言中的malloc程序包采用的就是这种方法。
4. 红黑树等树形结构：按块大小将空闲块组织为树形结构，同样有减少分配时间和提高内存利用率的作用。

## 7.10本章小结

本章主要关注hello程序的存储管理，介绍了不同的地址概念、地址变换与寻址、内存映射、内存分配管理等内容。不难看出，现代计算机系统为提高内存存储效率和使用率以至程序运行的效率使用了大量的机制和技术。此外，本章内容与教材第6章、第9章紧密联系，且补充了本文第6章的内容，体现了计算机系统课程的整体性。

# 第8章 hello的IO管理

## 8.1 Linux的IO设备管理方法

1. **设备的模型化——文件**

所有的I/O设备（例如网络、磁盘和终端）都被模型化为文件。

例如：/dev/sda2文件是用户磁盘分区，/dev/tty2文件是终端。

1. **设备管理——Unix IO接口**

将设备模型化为文件的方式允许Linux内核引入一个简单、低级的应用接口，称为Unix IO，这使得所有的输入和输出都能以一种统一且一致的方式来执行。

## 8.2 简述Unix IO接口及其函数

**Unix IO接口：**

* 打开文件：一个应用程序通过要求内核打开相应的文件，来宣告它想要访问一个I/O设备。内核返回一个小的非负整数，叫做描述符，它在后续对此文件的所有操作中标识这个文件。内核记录有关这个打开文件的所有信息，应用程序只需要记住这个描述符。
* Linux shell创建的每个进程开始时都有三个打开的文件：标准输入（描述符为0）、标准输出（描述符为1）和标准错误（描述符为2）。头文件< unistd.h> 定义了常量STDIN\_FILENO、STOOUT\_FILENO和STDERR\_FILENO,它们可用来代替显式的描述符值。
* 改变当前的文件位置：对于每个打开的文件，内核保持着一个文件位置k，初始为0。这个文件位置是从文件开头起始的字节偏移量。应用程序能够通过执行seek操作，显式地设置文件的当前位置为k。
* 读写文件：一个读操作就是从文件复制个字节到内存，从当前文件位置k开始，然后将k增加到k+n。给定一个大小为m字节的文件，当时执行读操作会触发一个称为end-of-file(EOF)的条件，应用程序能检测这个条件。在文件末尾处并没有明确的“EOF符号”。类似地，写操作就是从内存复制个字节到一个文件，从当前文件位置k开始，然后更新k。
* 关闭文件：当应用完成了对文件的访问之后，它就通知内核关闭这个文件。作为响应，内核释放文件打开时创建的数据结构，并将这个描述符恢复到可用的描述符池中。无论一个进程因为何种原因终止时，内核都会关闭所有打开的文件并释放它们的内存资源。

**相关函数：**

1. open()函数：

函数原型：int open(char \*filename, int flags, mode\_t mode);

此函数打开一个文件，将filename转换为一个文件描述符，并返回描述符数字（总是进程中未打开的最小描述符）。flags参数指明进程如何访问文件，mode参数指定新文件的访问权限位。

1. close()函数：

函数原型：int close(int fd);

此函数关闭一个打开的文件，关闭一个已关闭的描述符会出错。

1. read()函数：

函数原型：ssize\_t read(int fd, void \*buf, size\_t n);

此函数从描述符为fd的当前文件位置复制最多n个字节到内存位置buf。返回值-1表示一个错误，返回值0表示EOF。否则，返回值表示的是实际传送的字节数量。

1. write()函数：

函数原型：ssize\_t write(int fd, const void \*buf, size\_t n);

此函数从内存位置buf复制至多n个字节到描述符fd的当前文件位置。

1. lseek()函数：

通过调用此函数，应用程序能够显式地修改当前文件的位置。

此部分不在教材的讲述范围之内。

## 8.3 printf的实现分析

* **printf函数体：**

int printf(const char \*fmt, ...)

{

    int i;

    va\_list arg = (va\_list)((char \*)(&fmt) + 4);

    i = vsprintf(buf, fmt, arg);

    write(buf, i);

    return i;

}

分析：

1. printf函数调用了vsprintf函数，最后通过系统调用函数write进行输出；
2. va\_list是字符指针类型；
3. ((char \*)(&fmt) + 4)表示...中的第一个参数。

* **printf调用的vsprintf函数：**

int vsprintf(char \*buf, const char \*fmt, va\_list args)

{

    char \*p;

    chartmp[256];

    va\_listp\_next\_arg = args;

    for (p = buf; \*fmt; fmt++)

    {

        if (\*fmt != '%')

        {

            \*p++ = \*fmt;

            continue;

        }

        fmt++;

        switch (\*fmt)

        {

        case 'x':

            itoa(tmp, \*((int \*)p\_next\_arg));

            strcpy(p, tmp);

            p\_next\_arg += 4;

            p += strlen(tmp);

            break;

        case 's':

            break;

        /\* 这里应该还有一些对于

        其他格式输出的处理 \*/

        default:

            break;

        }

        return (p - buf);

    }

}

分析：vsprintf的作用就是格式化。它接受确定输出格式的格式字符串fmt。用格式字符串对个数变化的参数进行格式化，产生格式化输出写入buf供系统调用write输出时使用。

* **write系统调用：**

write:

mov eax, \_NR\_write

mov ebx, [esp + 4]

mov ecx, [esp + 8]

int INT\_VECTOR\_SYS\_CALL

分析：这里通过几个寄存器进行传参，随后调用中断门int INT\_VECTOR\_SYS\_CALL

即通过系统来调用sys\_call实现输出这一系统服务。

* **sys\_call部分内容：**

sys\_call:

     /\*

      \* ecx中是要打印出的元素个数

      \* ebx中的是要打印的buf字符数组中的第一个元素

      \* 这个函数的功能就是不断的打印出字符，直到遇到：'\0'

      \* [gs:edi]对应的是0x80000h:0采用直接写显存的方法显示字符串

      \*/

     xor si,si

     mov ah,0Fh

     mov al,[ebx+si]

     cmp al,'\0'

     je .end

     mov [gs:edi],ax

     inc si

    loop:

     sys\_call

    .end:

     ret

分析：通过逐个字符直接写至显存，输出格式化的字符串。

* **最后一部分工作：**

字符显示驱动子程序实现从ASCII到字模库到显示vram（即显存，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

当程序运行至getchar函数时，程序通过系统调用read等待用户键入字符并按回车键（通知系统输入完成），一种getchar函数的实现如下：

#include "sys/syscall.h"

#include <stdio.h>

int getchar(void)

{

    char c;

    return (read(0,&c,1)==1)?(unsigned char)c:EOF

    //EOF定义在stdio.h文件中

}

当用户键入回车之后，getchar通过系统调用read从输入缓冲区中每次读入一个字符。getchar函数的返回值是用户输入的第一个字符的ascii码，如出错返回-1。

异步异常——键盘中断（用户输入）的处理：键盘中断处理子程序接受按键扫描码并转成ASCII码，保存在系统的键盘缓冲区。

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

## 8.5本章小结

本章主要关注Linux系统中的I/O管理，阐述了Linux操作系统的IO设备管理方法——设备被模型化为文件并使用Unix IO接口进行文件操作，最后还分析了printf函数和getchar函数的实现。从此章的内容中我们能体会到Unix IO接口在Linux系统中的重要作用，同时也了解了作为异步异常之一的键盘中断的处理。

# 结论

1. hello程序历程总结

hello程序虽然是一个简短的C语言程序，但它的产生、执行、终止和回收离不开计算机系统各方面的协同工作，具体过程如下：

1. hello.c源代码文件通过C语言预处理器的预处理，得到了调整、展开后的ASCII文本文件hello.i；
2. hello.i经过编译器的编译得到汇编代码文件hello.s；
3. hello.s经过汇编器的汇编得到可重定向目标文件hello.o；
4. hello.o经过链接器的链接过程成为可执行目标文件hello；
5. 用户在shell-bash中键入执行hello程序的命令后，shell-bash解释用户的命令，找到hello可执行目标文件并为其执行fork创建新进程；
6. fork得到的新进程通过调用execve完成在其上下文中对hello程序的加载，hello开始执行；
7. hello作为一个进程运行，接受内核的进程调度（调用sleep后内核进行上下文切换，调度其他进程执行）；
8. hello执行的过程中，可能发生缺页异常等故障、系统调用等陷阱以及接收到各种信号，这些都需要操作系统与硬件设备的协同工作进行处理；
9. hello执行的过程中会访问其虚拟空间内的指令和数据，需要借助各种硬件、软件机制来快速、高效完成；
10. hello运行时要调用printf、getchar等函数，这些函数的实现与Linux系统IO设备管理、Unix IO接口等息息相关；
11. hello程序运行结束后，父进程shell-bash会进行回收，内核也会清除在内存中为其创建的各种数据结构和信息。
12. 个人感悟
13. 计算机系统的设计和实现大量体现了抽象的思想：文件是对I/O设备的抽象，虚拟内存是对主存和磁盘设备的抽象，进程是对处理器、主存和I/O设备的抽象，进程是操作系统对一个正在运行的程序的抽象等等；
14. 计算机系统课程内容虽然繁多，但逻辑结构清晰、层次分明。完成本论文让我体会到：一个小小的hello程序就能展现计算机系统的方方面面；
15. CSAPP这本书及计算机系统课程引领我从程序员的角度第一次系统地、全面地认识了现代操作系统的各种机制、设计和运行原理。我会在未来继续深入学习相关知识并在今后的具体实践中不断尝试使用和创新。

# 附件

|  |  |  |
| --- | --- | --- |
| 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 |

# 参考文献

1. GCC online documentation. <http://gcc.gnu.org/onlinedocs/>
2. 深入理解计算机系统（原书第三版）.机械工业出版社, 2016.
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>

1. 逻辑地址、线性地址与物理地址. GitHub Blog.

<https://vosamo.github.io/2016/01/VA2PA/>

1. 深入理解计算机系统-之-内存寻址（三）--分段管理机制（段描述符，段选择子，描述符表）. CSDN博客.

<https://blog.csdn.net/gatieme/article/details/50647000>

1. printf函数实现的深入剖析. 博客园.

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

1. read和write系统调用以及getchar的实现. CSDN博客.

<https://blog.csdn.net/ww1473345713/article/details/51680017>