

哈爾濱工業大學

計算機系統

大作業

題 目	<u>程序人生-Hello's P2P</u>
專 業	<u>計算機類</u>
學 號	<u>1190200526</u>
班 級	<u>1903002</u>
學 生	<u>沈城有</u>
指 導 教 師	<u>鄭貴濱</u>

計算機科學與技術學院

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 -
1.1 HELLO 简介	- 4 -
1.2 环境与工具	- 4 -
1.3 中间结果	- 5 -
1.4 本章小结	- 5 -
第 2 章 预处理	- 6 -
2.1 预处理的概念与作用	- 6 -
2.2 在 UBUNTU 下预处理的命令	- 6 -
2.3 HELLO 的预处理结果解析	- 7 -
2.4 本章小结	- 9 -
第 3 章 编译	- 10 -
3.1 编译的概念与作用	- 10 -
3.2 在 UBUNTU 下编译的命令	- 10 -
3.3 HELLO 的编译结果解析	- 11 -
3.3.1 数据与赋值	- 11 -
3.3.2 算术操作	- 12 -
3.3.3 数组、指针、结构操作	- 12 -
3.3.4 关系操作及控制转移	- 12 -
3.3.5 函数操作	- 13 -
3.4 本章小结	- 15 -
第 4 章 汇编	- 16 -
4.1 汇编的概念与作用	- 16 -
4.2 在 UBUNTU 下汇编的命令	- 16 -
4.3 可重定位目标 ELF 格式	- 16 -
4.3.1 readelf 命令	- 16 -
4.3.2 ELF 头	- 17 -
4.3.3 节头目表	- 18 -
4.3.4 重定位节	- 18 -
4.3.5 符号表	- 19 -
4.4 HELLO.O 的结果解析	- 20 -
4.5 本章小结	- 22 -
第 5 章 链接	- 23 -
5.1 链接的概念与作用	- 23 -
5.2 在 UBUNTU 下链接的命令	- 23 -
5.3 可执行目标文件 HELLO 的格式	- 23 -

5.4 HELLO 的虚拟地址空间	- 25 -
5.5 链接的重定位过程分析.....	- 27 -
5.6 HELLO 的执行流程	- 29 -
5.7 HELLO 的动态链接分析.....	- 30 -
5.8 本章小结.....	- 31 -
第 6 章 HELLO 进程管理.....	- 32 -
6.1 进程的概念与作用.....	- 32 -
6.2 简述壳 SHELL-BASH 的作用与处理流程.....	- 32 -
6.3 HELLO 的 FORK 进程创建过程	- 33 -
6.4 HELLO 的 EXECVE 过程	- 33 -
6.5 HELLO 的进程执行.....	- 33 -
6.6 HELLO 的异常与信号处理	- 34 -
6.7 本章小结	- 37 -
第 7 章 HELLO 的存储管理.....	- 38 -
7.1 HELLO 的存储器地址空间	- 38 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理	- 38 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理	- 39 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换.....	- 40 -
7.5 三级 CACHE 支持下的物理内存访问	- 41 -
7.6 HELLO 进程 FORK 时的内存映射	- 42 -
7.7 HELLO 进程 EXECVE 时的内存映射	- 42 -
7.8 缺页故障与缺页中断处理.....	- 42 -
7.9 动态内存分配管理	- 43 -
7.10 本章小结	- 44 -
第 8 章 HELLO 的 IO 管理	- 45 -
8.1 LINUX 的 IO 设备管理方法	- 45 -
8.2 简述 UNIX IO 接口及其函数	- 45 -
8.3 PRINTF 的实现分析.....	- 46 -
8.4 GETCHAR 的实现分析.....	- 48 -
8.5 本章小结	- 49 -
结论	- 50 -
附件	- 51 -
参考文献	- 52 -

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)。

2. 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

2. 软件环境

Windows10 64 位

Oracle VM VirtualBox 6.1.16 r140961 (Qt5.6.2)

Ubuntu 20.04.1

3. 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. 预处理的定义

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。

2. 预处理的作用

预处理的作用主要可分为以下三部分：

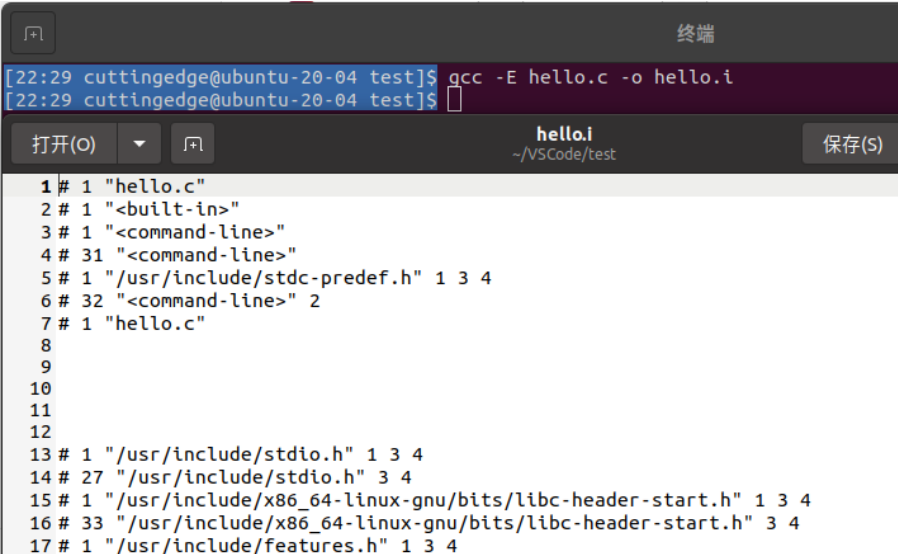
- (1) 宏展开：预处理程序中的“`#define`”标识符文本，用实际值（可以是字符串、代码等）替换用“`#define`”定义的字符串；
- (2) 文件包含复制：预处理程序中用“`#include`”格式包含的文件，将文件的内容插入到该命令所在的位置并删除原命令，从而把包含的文件和当前源文件连接成一个新的源文件，这与复制粘贴类似；
- (3) 条件编译处理：根据“`#if`”和“`#endif`”、“`#ifdef`”和“`#ifndef`”后面的条件确定需要编译的源代码。

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：



```
终端
[22:29 cuttingedge@ubuntu-20-04 test]$ gcc -E hello.c -o hello.i
[22:29 cuttingedge@ubuntu-20-04 test]$ 
hello.i
~/VSCode/test
1 # 1 "hello.c"
2 # 1 "<built-in>"
3 # 1 "<command-line>"
4 # 31 "<command-line>"
5 # 1 "/usr/include/stdc-predef.h" 1 3 4
6 # 32 "<command-line>" 2
7 # 1 "hello.c"
8
9
10
11
12
13 # 1 "/usr/include/stdio.h" 1 3 4
14 # 27 "/usr/include/stdio.h" 3 4
15 # 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
16 # 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
17 # 1 "/usr/include/features.h" 1 3 4
```

图 2-1 预处理命令1执行截图

```

[22:39 cuttingedge@ubuntu-20-04 test]$ cpp hello.c > hello.i
1 # 1 "hello.c"
2 # 1 "<built-in>"
3 # 1 "<command-line>"
4 # 31 "<command-line>"
5 # 1 "/usr/include/stdc-predef.h" 1 3 4
6 # 32 "<command-line>" 2
7 # 1 "hello.c"
8
9
10
11
12
13 # 1 "/usr/include/stdio.h" 1 3 4
14 # 27 "/usr/include/stdio.h" 3 4
15 # 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
16 # 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
17 # 1 "/usr/include/features.h" 1 3 4

```

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：

```

hello.i
1 # 1 "hello.c"
2 # 1 "<built-in>"
3 # 1 "<command-line>"
4 # 31 "<command-line>"
5 # 1 "/usr/include/stdc-predef.h" 1 3 4
6 # 32 "<command-line>" 2
7 # 1 "hello.c"

```

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）：

```

13 # 1 "/usr/include/stdio.h" 1 3 4
14 # 27 "/usr/include/stdio.h" 3 4
15 # 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
16 # 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
17 # 1 "/usr/include/features.h" 1 3 4
18 # 461 "/usr/include/features.h" 3 4
19 # 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
20 # 452 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
21 # 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
22 # 453 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
23 # 1 "/usr/include/x86_64-linux-gnu/bits/long-double.h" 1 3 4
24 # 454 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
25 # 462 "/usr/include/features.h" 2 3 4
26 # 485 "/usr/include/features.h" 3 4
27 # 1 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 1 3 4
28 # 10 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 3 4

```

图 2-4 预处理结果截图2（文件包含信息）


```

60 typedef unsigned char __u_char;
61 typedef unsigned short int __u_short;
62 typedef unsigned int __u_int;
63 typedef unsigned long int __u_long;
64
65
66 typedef signed char __int8_t;
67 typedef unsigned char __uint8_t;
68 typedef signed short int __int16_t;
69 typedef unsigned short int __uint16_t;
70 typedef signed int __int32_t;
71 typedef unsigned int __uint32_t;
72
73 typedef signed long int __int64_t;
74 typedef unsigned long int __uint64_t;

```

图 2-5 预处理结果截图3（类型定义信息）

```

334 extern int renameat (int __oldfd, const char *__old, int __newfd,
335     const char *__new) __attribute__ ((__nothrow__ , __leaf__));
336 # 173 "/usr/include/stdio.h" 3 4
337 extern FILE *tmpfile (void) ;
338 # 187 "/usr/include/stdio.h" 3 4
339 extern char *tmpnam (char *__s) __attribute__ ((__nothrow__ , __leaf__));
340
341
342
343
344 extern char *tmpnam_r (char *__s) __attribute__ ((__nothrow__ , __leaf__));
345 # 204 "/usr/include/stdio.h" 3 4
346 extern char *tempnam (const char *__dir, const char *__pfx)
347     __attribute__ ((__nothrow__ , __leaf__)) __attribute__ ((__malloc__));

```

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。

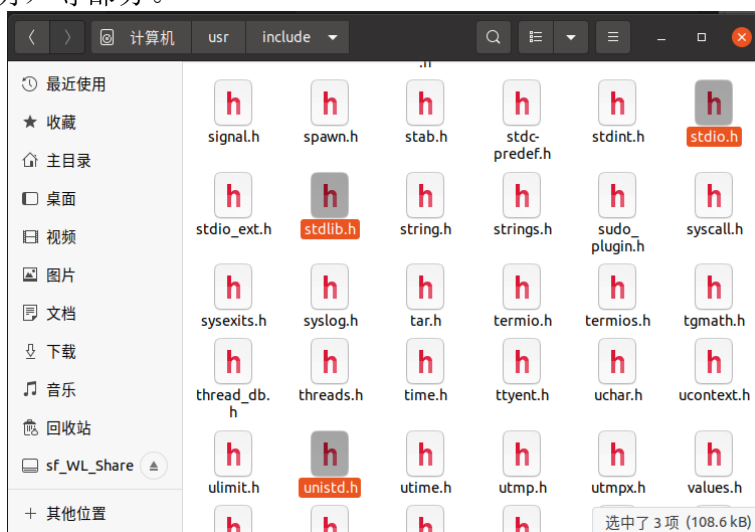


图 2-7 头文件路径位置截图

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：

```
3043 # 9 "hello.c" 2
3044
3045
3046 # 10 "hello.c"
3047 int sleepsecs=2.5;
3048
3049 int main(int argc,char *argv[])
3050 {
3051     int i;
3052
3053     if(argc!=3)
3054     {
3055         printf("Usage: Hello 学号 姓名! \n");
3056         exit(1);
3057     }
3058     for(i=0;i<10;i++)
3059     {
3060         printf("Hello %s %s\n",argv[1],argv[2]);
3061         sleep(sleepsecs);
3062     }
3063     getchar();
3064     return 0;
3065 }
```

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，主要完成四項工作：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`。

2. 编译的作用

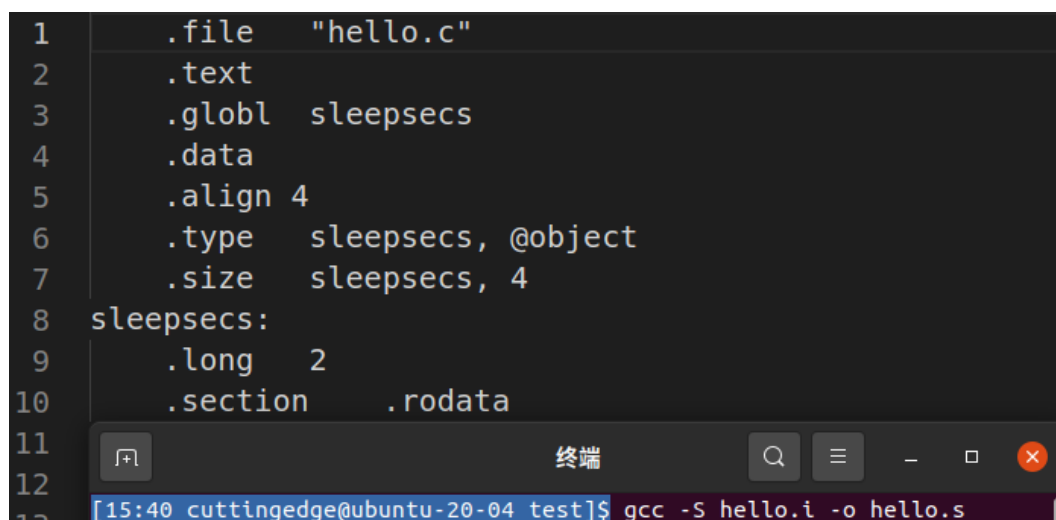
编译一般包括以下流程，每个步骤都有一定的作用：

- (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
```



The screenshot shows a terminal window with a dark background. The top part displays assembly code for a file named 'hello.c'. The code includes directives like `.file`, `.text`, `.globl`, `.data`, `.align`, `.type`, `.size`, and `.section`. It also shows a label `sleepsecs:` followed by `.long 2`. The bottom part of the terminal shows a command prompt with the command `gcc -S hello.i -o hello.s` being executed. The terminal window has a title bar with the text '终端' (Terminal) and standard window controls.

```
1      .file    "hello.c"
2      .text
3      .globl  sleepsecs
4      .data
5      .align  4
6      .type   sleepsecs, @object
7      .size   sleepsecs, 4
8  sleepsecs:
9      .long   2
10     .section .rodata
11
12 [15:40 cuttingedge@ubuntu-20-04 test]$ 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

a) 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"
```

b) 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)`

(2) 变量数据

a) 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。

b) 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。

(2) 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 $(SF \wedge OF) \vee ZF = 1$ ，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 $(SF \wedge OF) \vee ZF = 0$ ，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：

a) 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。

b) 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。

c) 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

```

(2) `printf()` 函数：

a) 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 `argv[1]`、`argv[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。

b) 函数调用：主函数通过 `call` 指令调用。

c) 函数返回：返回值被忽略。

(3) `exit()` 函数：

a) 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。

b) 函数调用：主函数通过 `call` 指令调用。

c) 函数返回：函数不返回，直接退出程序。

(4) `sleep()` 函数：

a) 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。

b) 函数调用：主函数通过 `call` 指令调用。

c) 函数返回：返回值被忽略（返回的是实际休眠时间）。

(5) `getchar()` 函数：

a) 参数传递：无。

b) 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
```

- c) 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。

2. 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`

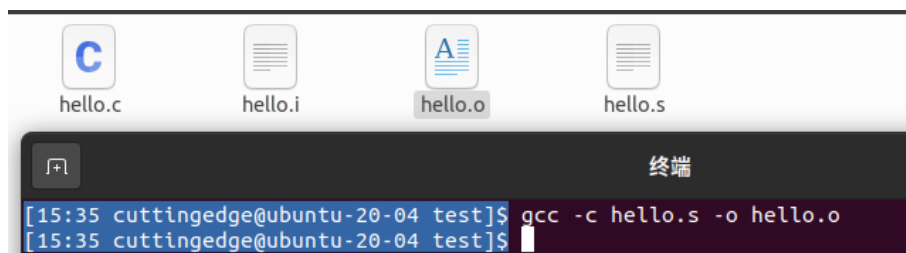


图 4-1 汇编命令执行截图1

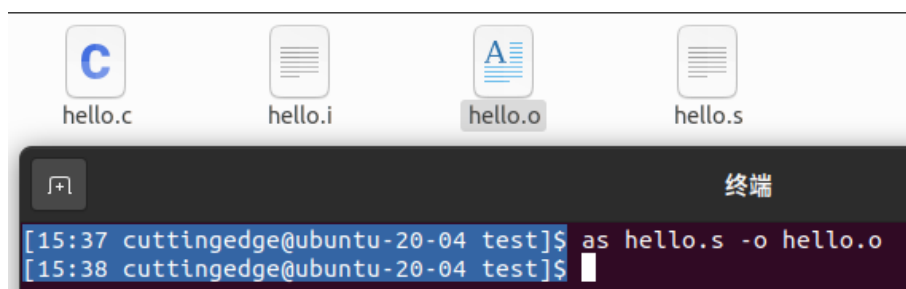


图 4-2 汇编命令执行截图2

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：

```

1 ELF 头:
2   Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
3   类别:                               ELF64
4   数据:                               2 补码, 小端序 (little endian)
5   Version:                             1 (current)
6   OS/ABI:                               UNIX - System V

[15:46 cuttingedge@ubuntu-20-04 test]$ readelf -a hello.o > hello_elf.txt
[15:47 cuttingedge@ubuntu-20-04 test]$

```

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：

```

1 ELF 头:
2   Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
3   类别:                               ELF64
4   数据:                               2 补码, 小端序 (little endian)
5   Version:                             1 (current)
6   OS/ABI:                               UNIX - System V
7   ABI 版本:                             0
8   类型:                               REL (可重定位文件)
9   系统架构:                             Advanced Micro Devices X86-64
10  版本:                               0x1
11  入口点地址:                             0x0
12  程序头起点:                             0 (bytes into file)
13  Start of section headers:               1232 (bytes into file)
14  标志:                               0x0
15  Size of this header:                   64 (bytes)
16  Size of program headers:               0 (bytes)
17  Number of program headers:              0
18  Size of section headers:               64 (bytes)
19  Number of section headers:              14
20  Section header string table index: 13

```

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 节头目录表

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：

22	节头:						
23	[号]	名称	类型	地址	偏移量		
24		大小	全体大小	旗标	链接	信息	对齐
25	[0]		NULL	0000000000000000	0	0	0
26		0000000000000000	0000000000000000		0	0	0
27	[1]	.text	PROGBITS	0000000000000000			00000040
28		0000000000000085	0000000000000000	AX	0	0	1
29	[2]	.rela.text	RELA	0000000000000000			00000380
30		00000000000000c0	0000000000000018	I	11	1	8
31	[3]	.data	PROGBITS	0000000000000000			000000c8
32		0000000000000004	0000000000000000	WA	0	0	4
33	[4]	.bss	NOBITS	0000000000000000			000000cc
34		0000000000000000	0000000000000000	WA	0	0	1
35	[5]	.rodata	PROGBITS	0000000000000000			000000cc
36		000000000000002b	0000000000000000	A	0	0	1
37	[6]	.comment	PROGBITS	0000000000000000			000000f7
38		000000000000002b	0000000000000001	MS	0	0	1
39	[7]	.note.GNU-stack	PROGBITS	0000000000000000			00000122
40		0000000000000000	0000000000000000		0	0	1
41	[8]	.note.gnu.property	NOTE	0000000000000000			00000128
42		0000000000000020	0000000000000000	A	0	0	8
43	[9]	.eh_frame	PROGBITS	0000000000000000			00000148
44		0000000000000038	0000000000000000	A	0	0	8
45	[10]	.rela.eh_frame	RELA	0000000000000000			00000440
46		0000000000000018	0000000000000018	I	11	9	8
47	[11]	.symtab	SYMTAB	0000000000000000			00000180
48		00000000000001b0	0000000000000018		12	10	8
49	[12]	.strtab	STRTAB	0000000000000000			00000330
50		000000000000004d	0000000000000000		0	0	1
51	[13]	.shstrtab	STRTAB	0000000000000000			00000458
52		0000000000000074	0000000000000000		0	0	1

图 4-5 节头目录表信息

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：

```

65 重定位节 '.rela.text' at offset 0x380 contains 8 entries:
66 | 偏移量      信息      类型      符号值      符号名称 + 加数
67 | 00000000001c 000500000002 R_X86_64_PC32 0000000000000000 .rodata - 4
68 | 000000000021 000d00000004 R_X86_64_PLT32 0000000000000000 puts - 4
69 | 00000000002b 000e00000004 R_X86_64_PLT32 0000000000000000 exit - 4
70 | 000000000054 000500000002 R_X86_64_PC32 0000000000000000 .rodata + 1a
71 | 00000000005e 000f00000004 R_X86_64_PLT32 0000000000000000 printf - 4
72 | 000000000064 000a00000002 R_X86_64_PC32 0000000000000000 sleepsecs - 4
73 | 00000000006b 001000000004 R_X86_64_PLT32 0000000000000000 sleep - 4
74 | 00000000007a 001100000004 R_X86_64_PLT32 0000000000000000 getchar - 4
75
76 重定位节 '.rela.eh_frame' at offset 0x440 contains 1 entry:
77 | 偏移量      信息      类型      符号值      符号名称 + 加数
78 | 000000000020 000200000002 R_X86_64_PC32 0000000000000000 .text + 0

```

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：

```

82 Symbol table '.symtab' contains 18 entries:
83 | Num:      Value      Size Type Bind Vis      Ndx Name
84 | 0: 0000000000000000 0 NOTYPE LOCAL DEFAULT UND
85 | 1: 0000000000000000 0 FILE LOCAL DEFAULT ABS hello.c
86 | 2: 0000000000000000 0 SECTION LOCAL DEFAULT 1
87 | 3: 0000000000000000 0 SECTION LOCAL DEFAULT 3
88 | 4: 0000000000000000 0 SECTION LOCAL DEFAULT 4
89 | 5: 0000000000000000 0 SECTION LOCAL DEFAULT 5
90 | 6: 0000000000000000 0 SECTION LOCAL DEFAULT 7
91 | 7: 0000000000000000 0 SECTION LOCAL DEFAULT 8
92 | 8: 0000000000000000 0 SECTION LOCAL DEFAULT 9
93 | 9: 0000000000000000 0 SECTION LOCAL DEFAULT 6
94 | 10: 0000000000000000 4 OBJECT GLOBAL DEFAULT 3 sleepsecs
95 | 11: 0000000000000000 133 FUNC GLOBAL DEFAULT 1 main
96 | 12: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND _GLOBAL_OFFSET_TABLE_
97 | 13: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND puts
98 | 14: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND exit
99 | 15: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND printf
100 | 16: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND sleep
101 | 17: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND getchar

```

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+0x2
0>
    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+0x5
8>
    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+0x6
8>
    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:

```

```
    cml    $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`。

2. 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`

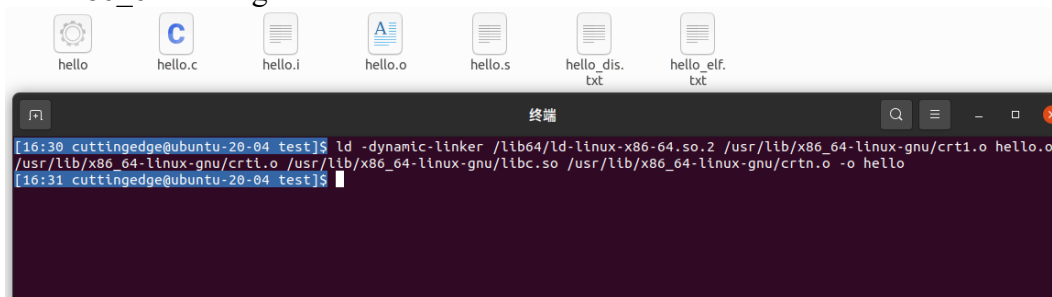


图 5-1 链接命令执行截图

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】

22	节头:						
23	[号]	名称	类型	地址		偏移量	
24		大小	全体大小	旗标	链接	信息	对齐
25	[0]		NULL	0000000000000000			00000000
26		0000000000000000	0000000000000000		0	0	0
27	[1]	.interp	PROGBITS	00000000004002e0			000002e0
28		000000000000001c	0000000000000000	A	0	0	1
29	[2]	.note.gnu.propert	NOTE	0000000000400300			00000300
30		0000000000000020	0000000000000000	A	0	0	8
31	[3]	.note.ABI-tag	NOTE	0000000000400320			00000320
32		0000000000000020	0000000000000000	A	0	0	4
33	[4]	.hash	HASH	0000000000400340			00000340
34		0000000000000034	0000000000000004	A	6	0	8
35	[5]	.gnu.hash	GNU_HASH	0000000000400378			00000378
36		000000000000001c	0000000000000000	A	6	0	8
37	[6]	.dynsym	DYSYM	0000000000400398			00000398
38		00000000000000c0	0000000000000018	A	7	1	8
39	[7]	.dynstr	STRTAB	0000000000400458			00000458
40		0000000000000057	0000000000000000	A	0	0	1
41	[8]	.gnu.version	VERSYM	00000000004004b0			000004b0
42		0000000000000010	0000000000000002	A	6	0	2
43	[9]	.gnu.version_r	VERNEED	00000000004004c0			000004c0
44		0000000000000020	0000000000000000	A	7	1	8
45	[10]	.rela.dyn	RELA	00000000004004e0			000004e0
46		0000000000000030	0000000000000018	A	6	0	8
47	[11]	.rela.plt	RELA	0000000000400510			00000510
48		0000000000000078	0000000000000018	AI	6	21	8
49	[12]	.init	PROGBITS	0000000000401000			00001000
50		000000000000001b	0000000000000000	AX	0	0	4
51	[13]	.plt	PROGBITS	0000000000401020			00001020
52		0000000000000060	0000000000000010	AX	0	0	16
53	[14]	.plt.sec	PROGBITS	0000000000401080			00001080
54		0000000000000050	0000000000000010	AX	0	0	16
55	[15]	.text	PROGBITS	00000000004010d0			000010d0
56		0000000000000135	0000000000000000	AX	0	0	16
57	[16]	.fini	PROGBITS	0000000000401208			00001208
58		000000000000000d	0000000000000000	AX	0	0	4
59	[17]	.rodata	PROGBITS	0000000000402000			00002000
60		000000000000002f	0000000000000000	A	0	0	4
61	[18]	.eh_frame	PROGBITS	0000000000402030			00002030
62		00000000000000fc	0000000000000000	A	0	0	8
63	[19]	.dynamic	DYNAMIC	0000000000403e50			00002e50
64		00000000000001a0	0000000000000010	WA	7	0	8
65	[20]	.got	PROGBITS	0000000000403ff0			00002ff0
66		0000000000000010	0000000000000008	WA	0	0	8
67	[21]	.got.plt	PROGBITS	0000000000404000			00003000
68		0000000000000040	0000000000000008	WA	0	0	8
69	[22]	.data	PROGBITS	0000000000404040			00003040
70		0000000000000008	0000000000000000	WA	0	0	4

(接上页)

71	[23]	.comment	PROGBITS	0000000000000000	00003048
72		000000000000002a	0000000000000001	MS	0 0 1
73	[24]	.symtab	SYMTAB	0000000000000000	00003078
74		00000000000004c8	0000000000000018		25 30 8
75	[25]	.strtab	STRTAB	0000000000000000	00003540
76		0000000000000150	0000000000000000		0 0 1
77	[26]	.shstrtab	STRTAB	0000000000000000	00003690
78		00000000000000e1	0000000000000000		0 0 1

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：

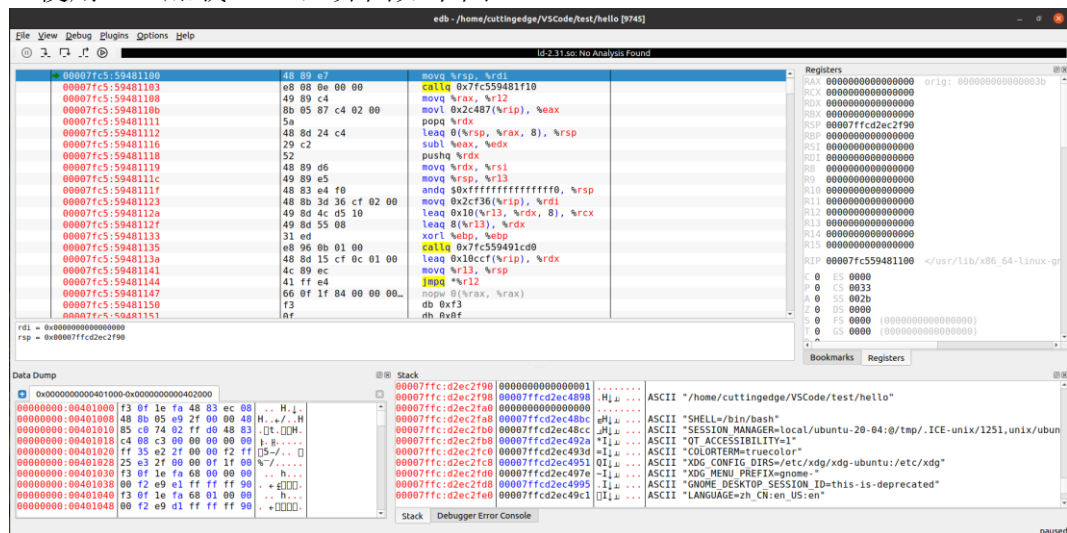


图 5-3 edb加载hello截图

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。

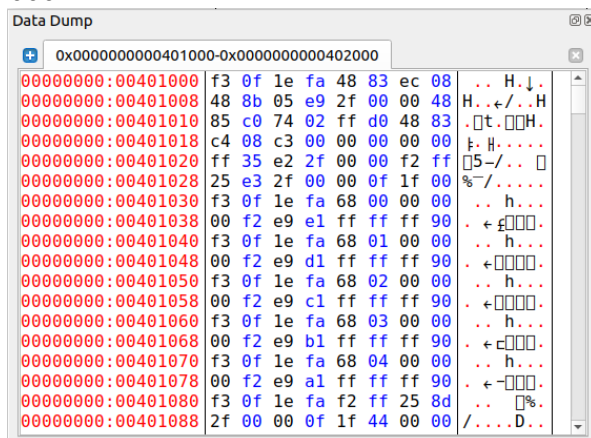


图 5-4 Data Dump窗口

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）：

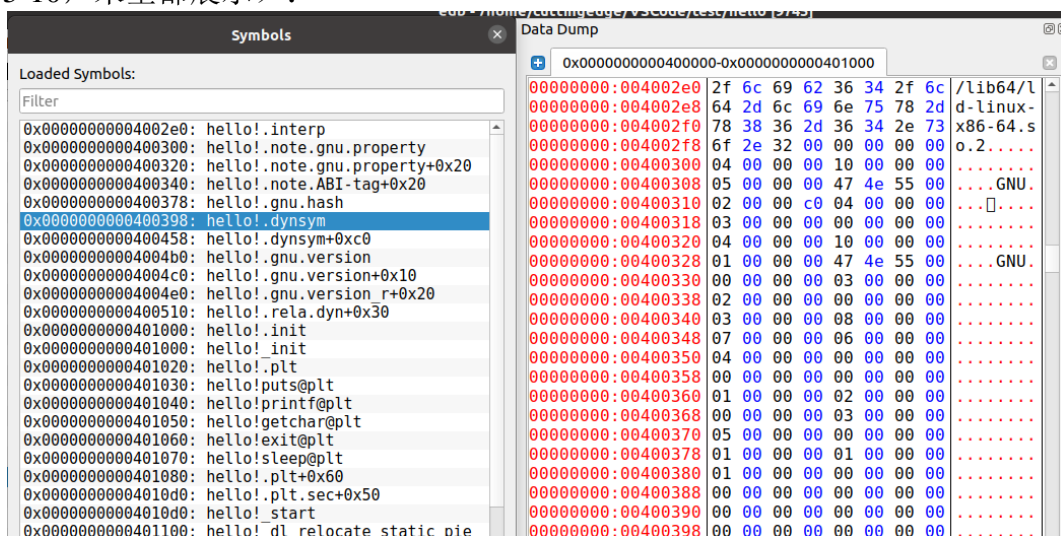


图 5-5 Symbols与Data Dump对照截图1

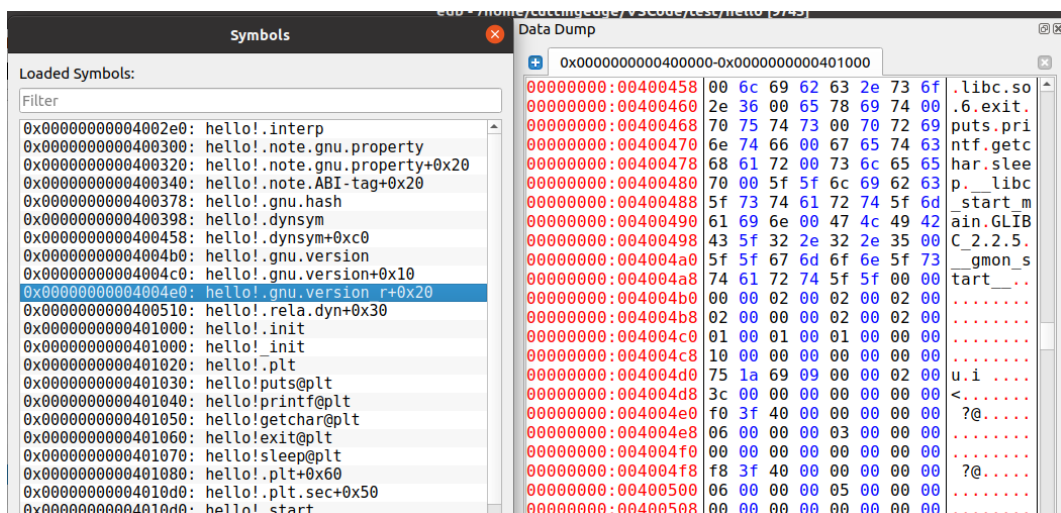


图 5-6 Symbols与Data Dump对照截图2

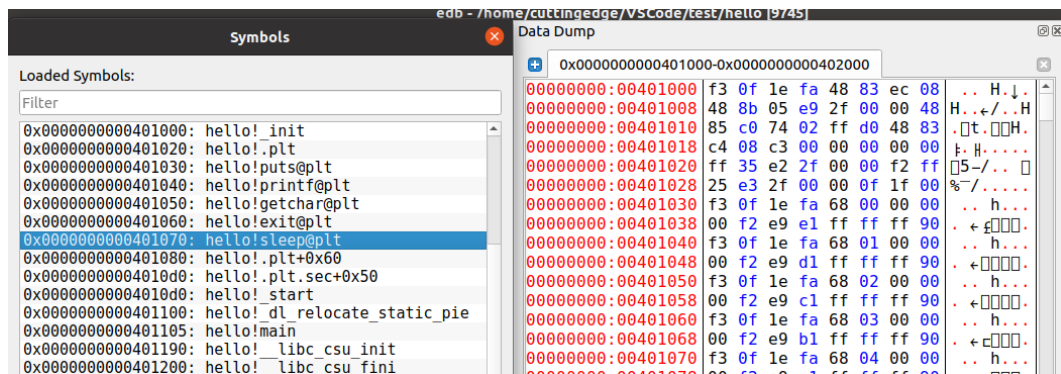


图 5-7 Symbols与Data Dump对照截图3

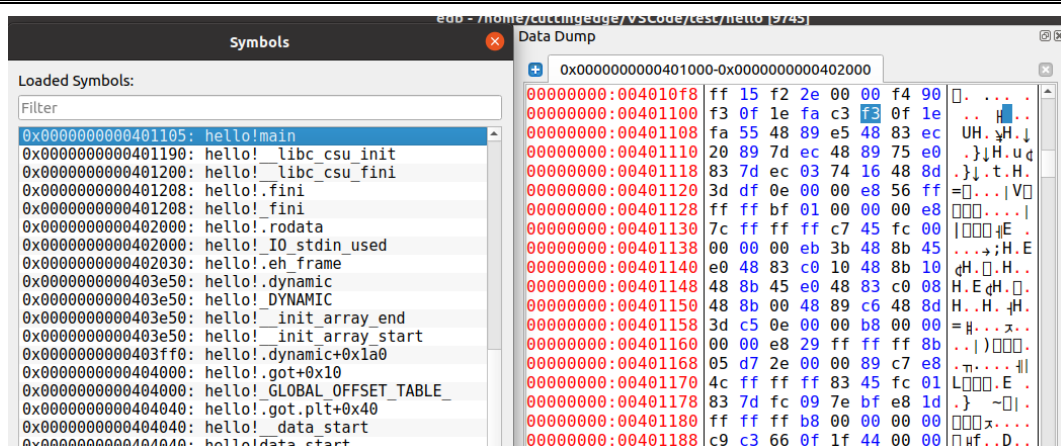


图 5-8 Symbols与Data Dump对照截图4（.text段部分）

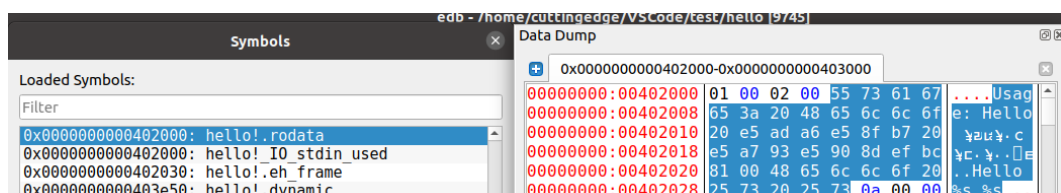


图 5-9 Symbols与Data Dump对照截图5（.rodata段部分）

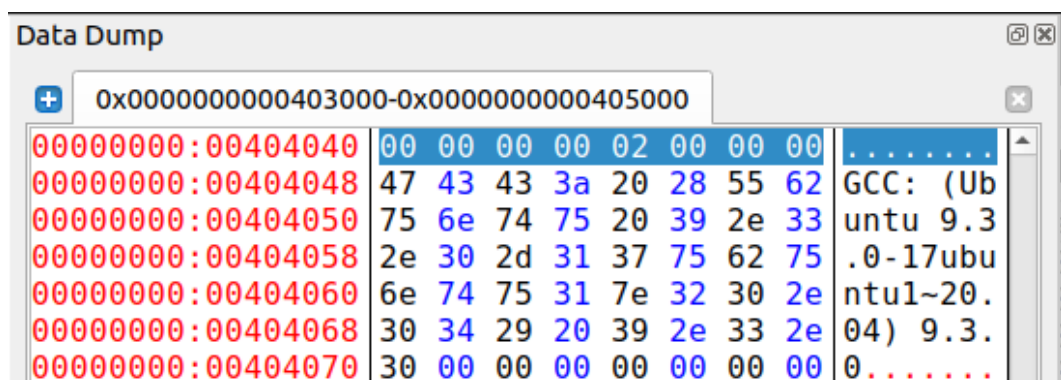


图 5-10 Symbols与Data Dump对照截图6（.data段部分）

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：

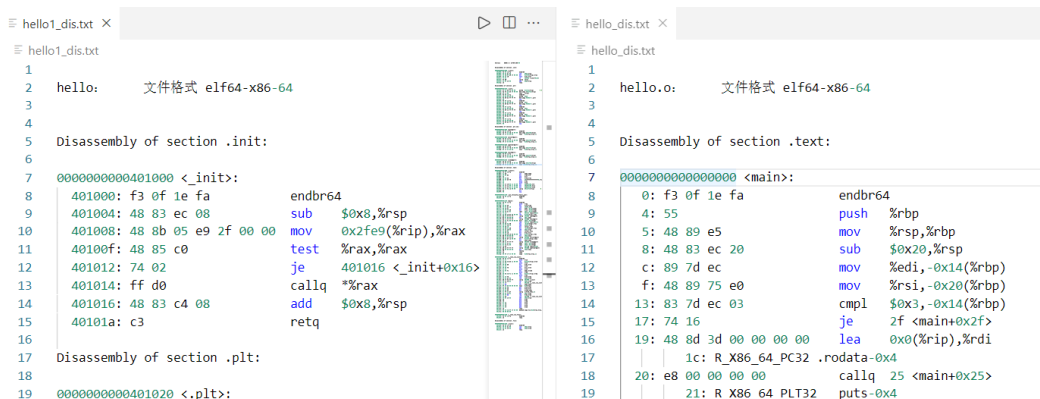


图 5-11 hello相较于hello.o多出了一些节

- (2) 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：

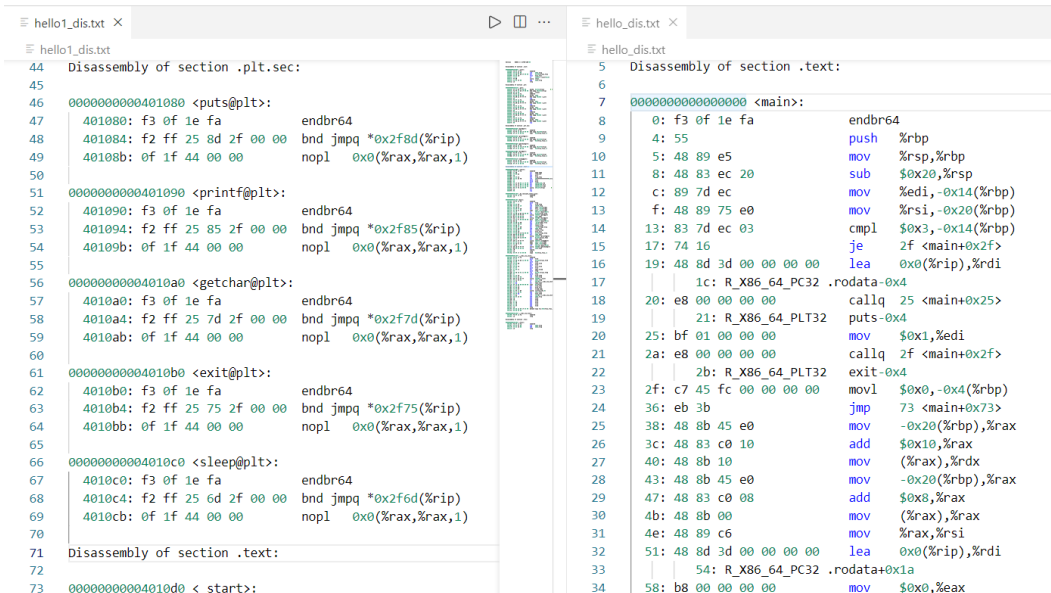


图 5-12 hello相较于hello.o增加了一些函数

- (3) 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)。

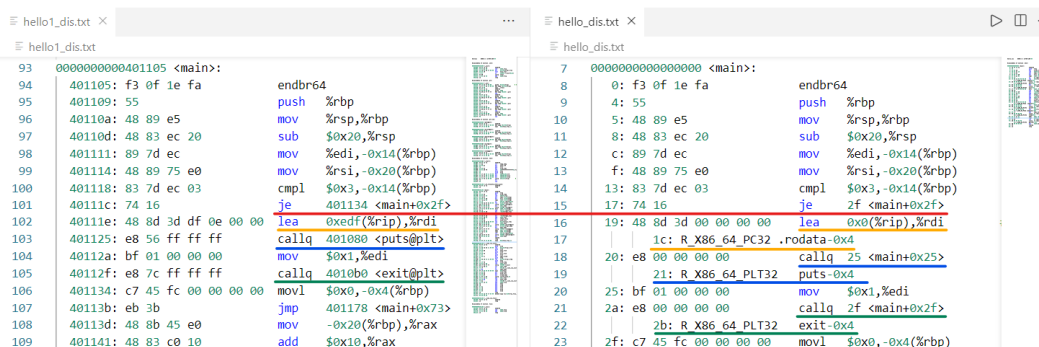


图 5-13 hello对于函数调用和跳转语句中重定向的处理

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）：

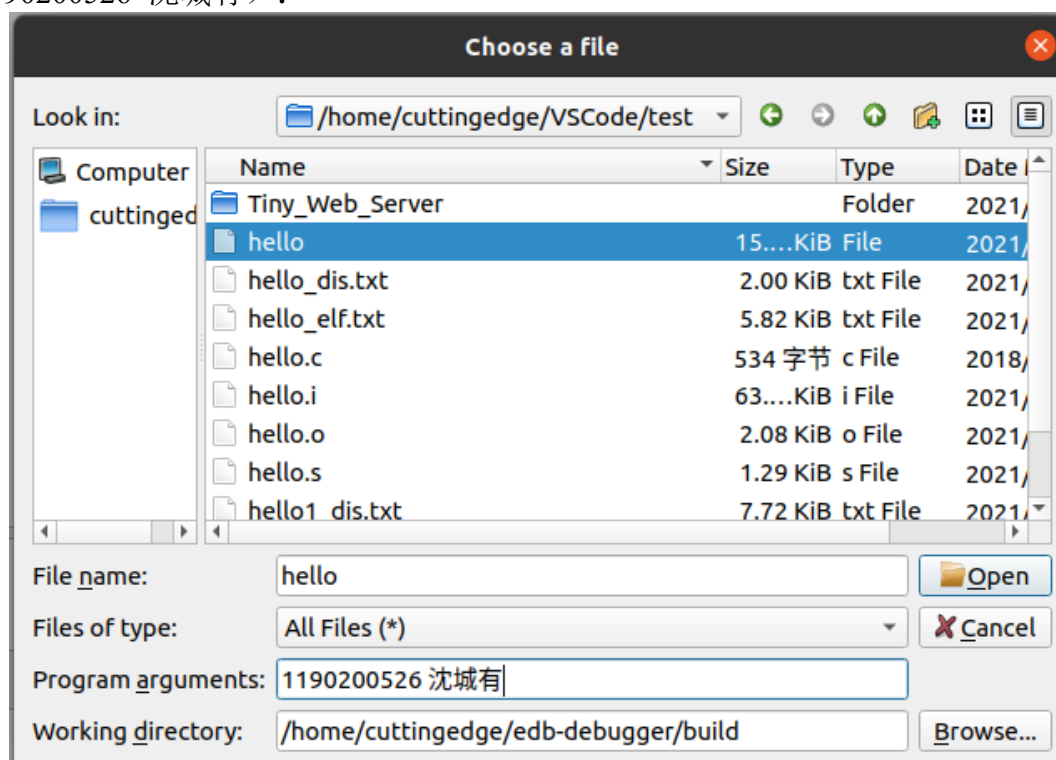


图 5-14 EDB执行hello前添加程序参数

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：

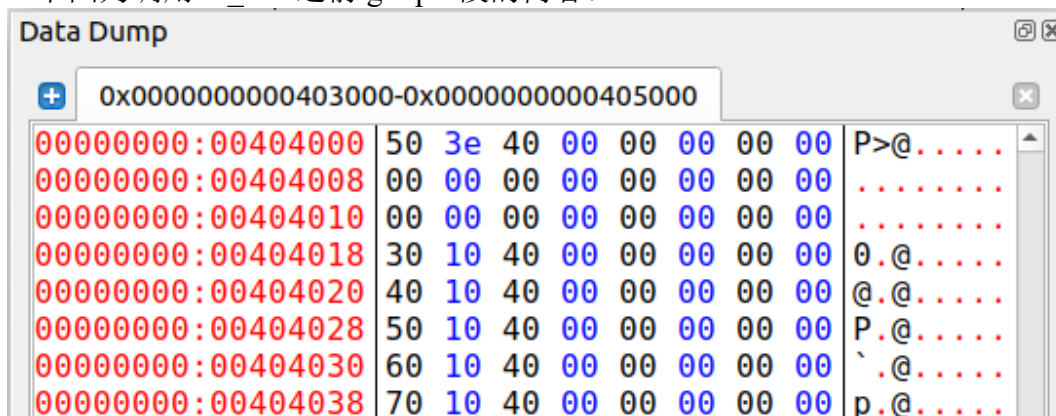


图 5-15 调用dl_init之前.got.plt段的内容

下图为调用 dl_init 之后.got.plt 段的内容：

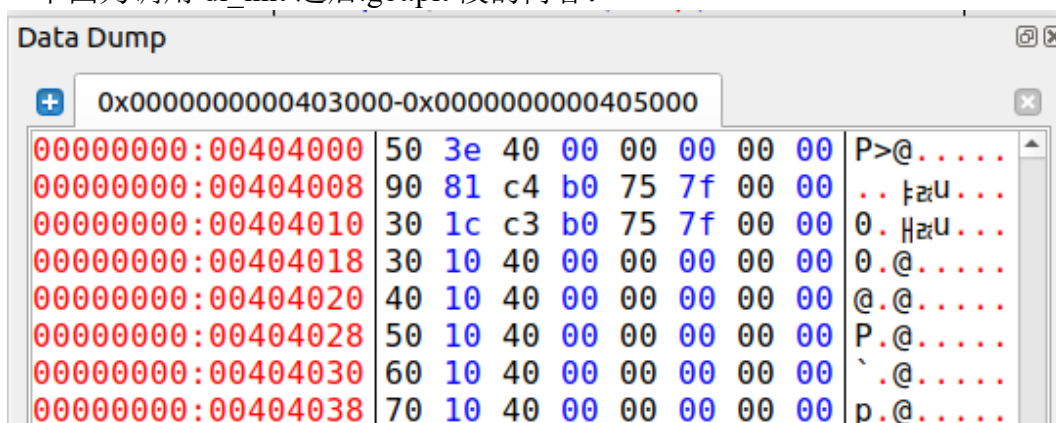


图 5-16 调用dl_init之后.got.plt段的内容

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。

2. 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。

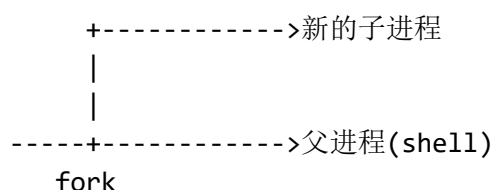
简化处理流程：

- (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：



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：

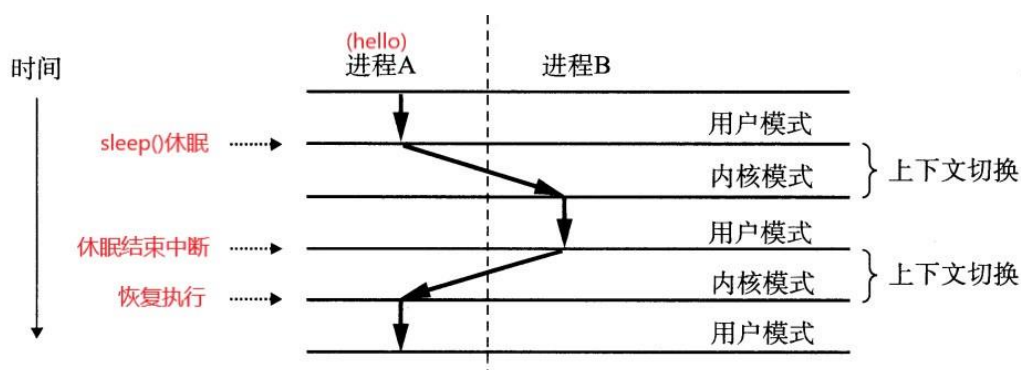


图 6-1 hello程序进程执行过程示意

6.6 hello 的异常与信号处理

执行截图：

```
[15:55 cuttingedge@ubuntu-20-04 test]$ ./hello 1190200526 沈城有
Hello 1190200526 沈城有
Hello 1190200526 沈城有

nwkaasdkv Hello 1190200526 沈城有
```

(图接上页)

```

bvsdbvk
vsdaknvdv
Hello 1190200526 沈城有

vdvsdnkn
Hello 1190200526 沈城有

dvksnnskHello 1190200526 沈城有

Hello 1190200526 沈城有
Hello 1190200526 沈城有
Hello 1190200526 沈城有
Hello 1190200526 沈城有
fndnn
[15:56 cuttingedge@ubuntu-20-04 test]$
[15:56 cuttingedge@ubuntu-20-04 test]$ nwkaasdkv bvsdbvk
nwkaasdkv: 未找到命令
[15:56 cuttingedge@ubuntu-20-04 test]$ vsdaknvdv
vsdaknvdv: 未找到命令
[15:56 cuttingedge@ubuntu-20-04 test]$
[15:56 cuttingedge@ubuntu-20-04 test]$
[15:56 cuttingedge@ubuntu-20-04 test]$
[15:56 cuttingedge@ubuntu-20-04 test]$ vdvsdnkn
vdvsdnkn: 未找到命令

```

图 6-2 执行过程中随意输入，被认为是命令

```

[15:58 cuttingedge@ubuntu-20-04 test]$ ./hello 1190200526 沈城有
Hello 1190200526 沈城有
^C
[15:58 cuttingedge@ubuntu-20-04 test]$ ps
  PID TTY          TIME CMD
  33125 pts/1        00:00:00 bash
  33211 pts/1        00:00:00 ps
[15:58 cuttingedge@ubuntu-20-04 test]$ jobs
[15:58 cuttingedge@ubuntu-20-04 test]$ fg
bash: fg: 当前: 无此任务

```

图 6-3 键入Ctrl-C，程序终止

```

[16:03 cuttingedge@ubuntu-20-04 test]$ ./hello 1190200526 沈城有
Hello 1190200526 沈城有
Hello 1190200526 沈城有
^Z
[1]+  已停止                  ./hello 1190200526 沈城有
[16:03 cuttingedge@ubuntu-20-04 test]$ jobs
[1]+  已停止                  ./hello 1190200526 沈城有
[16:03 cuttingedge@ubuntu-20-04 test]$ fg
./hello 1190200526 沈城有
Hello 1190200526 沈城有
Hello 1190200526 沈城有
^Z
[1]+  已停止                  ./hello 1190200526 沈城有

```

图 6-4 键入Ctrl-Z后jobs、fg

```

[16:03 cuttingedge@ubuntu-20-04 test]$ ps
  PID TTY          TIME CMD
 33125 pts/1        00:00:00 bash
 33222 pts/1        00:00:00 hello
 33223 pts/1        00:00:00 ps
[16:04 cuttingedge@ubuntu-20-04 test]$ kill -9 33222
[1]+  已杀死                  ./hello 1190200526 沈城有
[16:04 cuttingedge@ubuntu-20-04 test]$ ps
  PID TTY          TIME CMD
 33125 pts/1        00:00:00 bash
 33228 pts/1        00:00:00 ps

```

图 6-5 kill命令终止hello进程

```

gnome-terminal- bash- hello
                  |    |
                  |    +-- pstree
                  |
                  +-- 4*[{gnome-terminal-}]

```

图 6-6 终止hello进程前pstree结果（部分）

```

gnome-terminal- bash- pstree
                  |
                  +-- 4*[{gnome-terminal-}]

```

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；

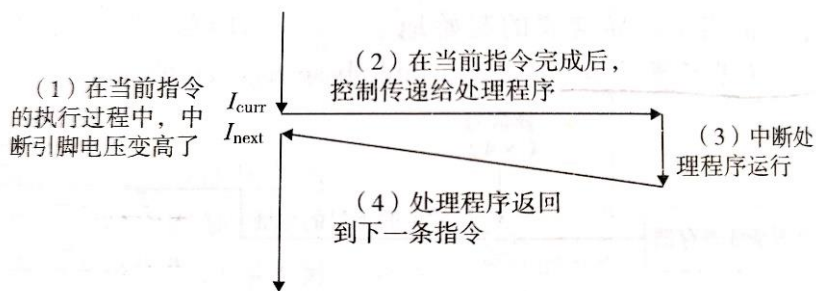


图 6-8 中断处理

(2) 陷阱：有意的异常，执行指令的结果（例如：exit），同步；

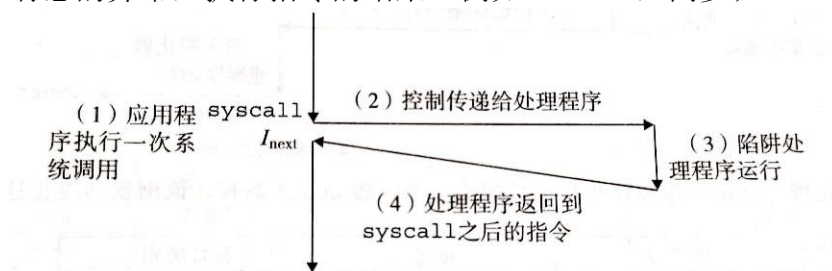


图 6-9 陷阱处理

(3) 故障：潜在可恢复的错误（如：缺页异常），同步。

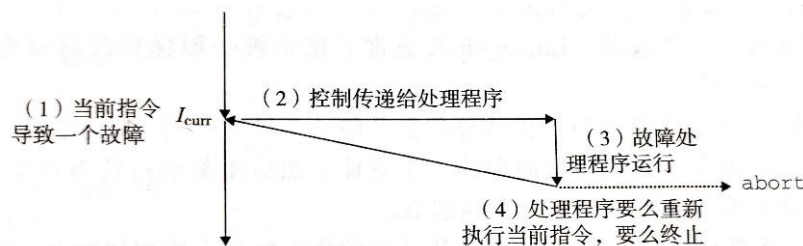


图 6-10 故障处理

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。

2. 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。

3. 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。

4. 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 $P = 2^p$ （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：

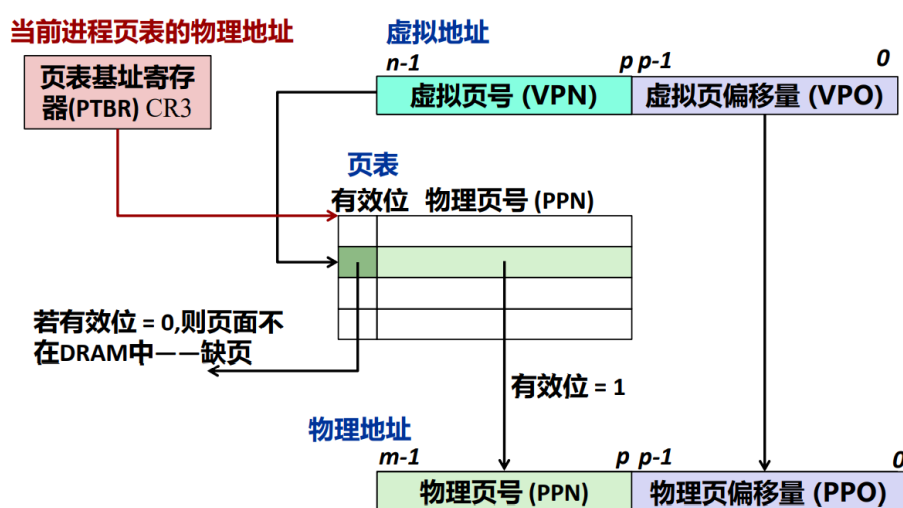


图 7-1 线性地址到物理地址的变换过程示意图

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）：

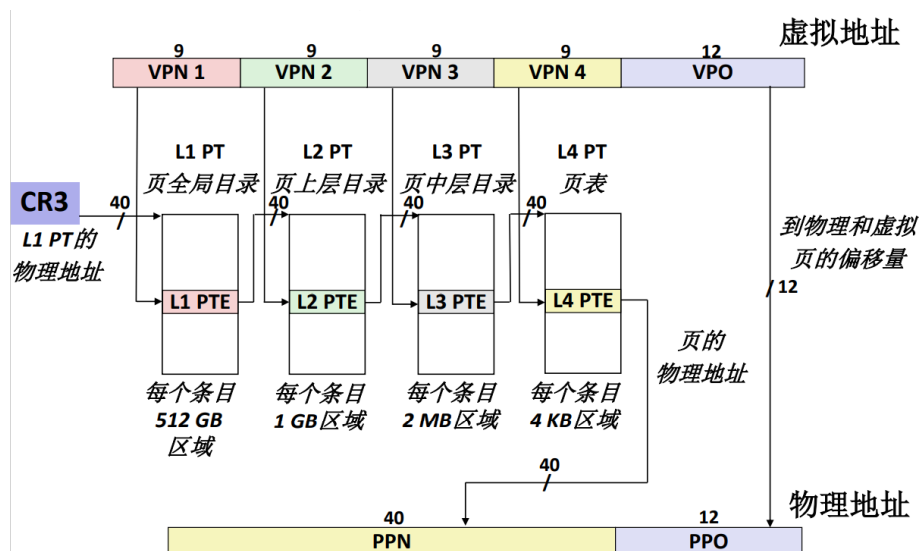


图 7-2 VA到PA的变换示意图

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：

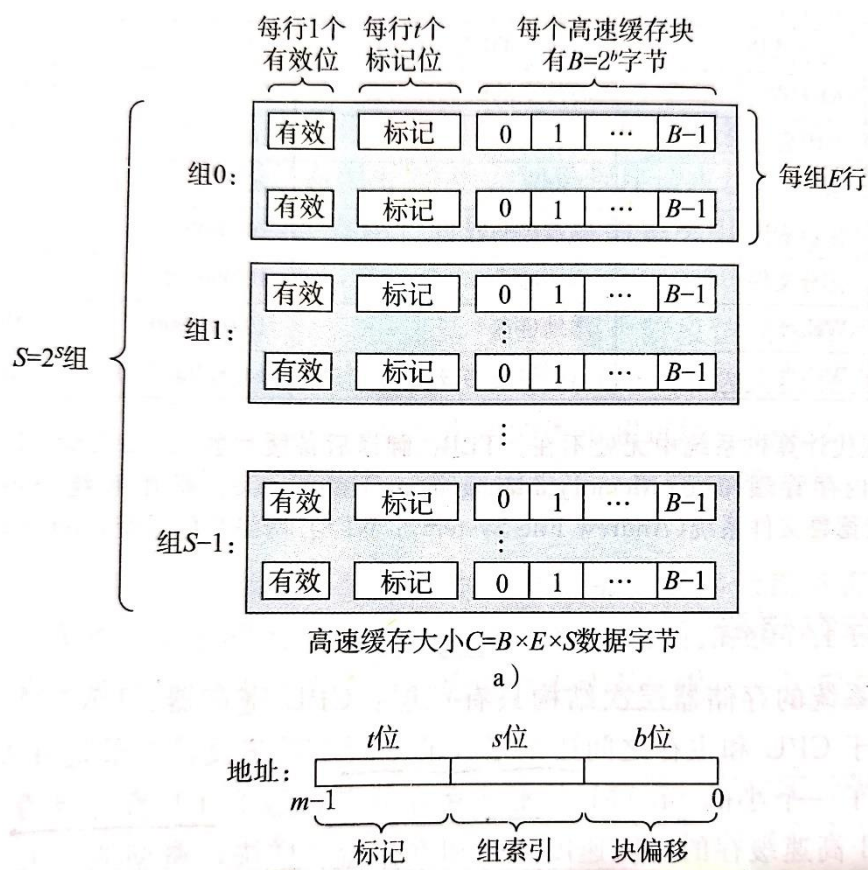


图 7-3 高速缓存存储器组织结构示意

- (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：

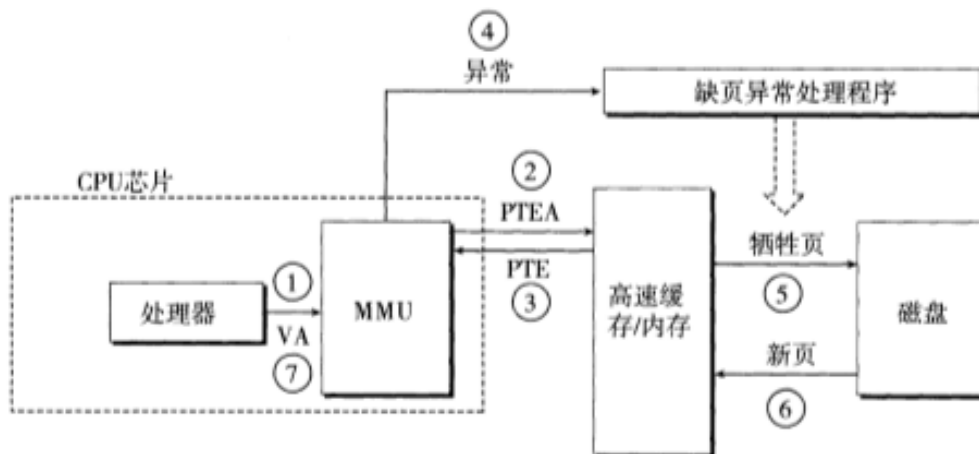


图 7-4 缺页异常处理过程示意

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。显式分配器必须满足一些相当严格的约束条件：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。

2. 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`、`STDOUT_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 $n > 0$ 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 $k \geq m$ 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 $n > 0$ 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。

(2) 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。

(3) 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。

(4) 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。

(5) 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：

- `printf` 函数调用了 `vsprintf` 函数，最后通过系统调用函数 `write` 进行输出；
- `va_list` 是字符指针类型；
- `((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;
    char tmp[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

一、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。

二、个人感悟

1. 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;
2. 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;
3. 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
- [5] 逻辑地址、线性地址与物理地址. GitHub Blog.
<https://vosamo.github.io/2016/01/VA2PA/>
- [6] 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>
- [7] printf 函数实现的深入剖析. 博客园.
<https://www.cnblogs.com/pianist/p/3315801.html>
- [8] read 和 write 系统调用以及 getchar 的实现. CSDN 博客.
<https://blog.csdn.net/ww1473345713/article/details/51680017>