哈爾濱工業大學

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　1170300821

班　　级　1703008

学　　生　罗瑞欣

指 导 教 师　郑贵滨

计算机科学与技术学院

2018 年 12 月

摘　要

本论文旨在研究 hello 在 linux 系统下的整个生命周期。结合 CSAPP 课本，通过 gcc 等工具进行实验，从而将课本知识落实、融会贯通，通过一个程序深入 挖掘知识点，对于学生对于课程的理解以及知识的升华有很大帮助。。

关键词：CSAPP；HIT；大作业；Hello 程序；生命周期；

（摘要 0 分，缺失-1 分，根据内容精彩称都酌情加分 0-1 分）

目 录

第 1 章 概述

1.1 Hello 简介

P2P：

1. Program：在 editor 中键入代码得到 hello.c 程序
2. Process：hello.c（在 Linux 中），经过过 cpp 的预处理、ccl 的编译、as 的汇编、ld 的链接最终成为可执目标程序 hello。在 shell 中键入启动命令后，shell 为其 fork，产生子进程。

020：

1. shell 为 hello 进程 execve，映射虚拟内存，进入程序入口后程序开始载入物理内存。
2. 进入 main 函数执行目标代码，CPU 为运行的 hello 分配时间片执行逻辑控制流。
3. 当程序运行结束后，shell 父进程负责回收 hello 进程，内核删除相关数据结构。

1.2 环境与工具

硬件环境：Intel Core i7-6700HQ x64CPU,16G RAM,256G SSD.

软件环境：Ubuntu18.04.1 LTS

开发与调试工具：vim，gcc，as，ld，edb，readelf，HexEdit

1.3 中间结果

文件名	文件功能
hello.i	预处理之后的文本文件
hello.s	编译之后的汇编文件
hello.o	汇编之后可重定位目标执行文件
hello	链接之后的可执行目标文件
hello2.c	测试程序代码
hello2	测试程序
helloo.objdmp	hello.o 的反汇编文件
hello.elf	hello 的 ELF 格式

hello.objdmp	hello 的反汇编文件
tmp.txt	存放临时数据

1.4 本章小结

Hello world 的来源已经不可考（虽然也没几十年），有人说这是程序员对新世界的呼喊，有人说这是搭建环境变量成功的标志，但是我个人最喜欢的是这个解释：在某本知名语言的知名教科书里如是说：“程序员通常在用新的语言时输出‘hello world’，使以后的编程比较吉利”。。。。。。

本章主要简单介绍了 hello 的 p2p，020 过程，列出了本次实验信息：环境、中间结果

（第 1 章 0.5 分）

第 2 章 预处理

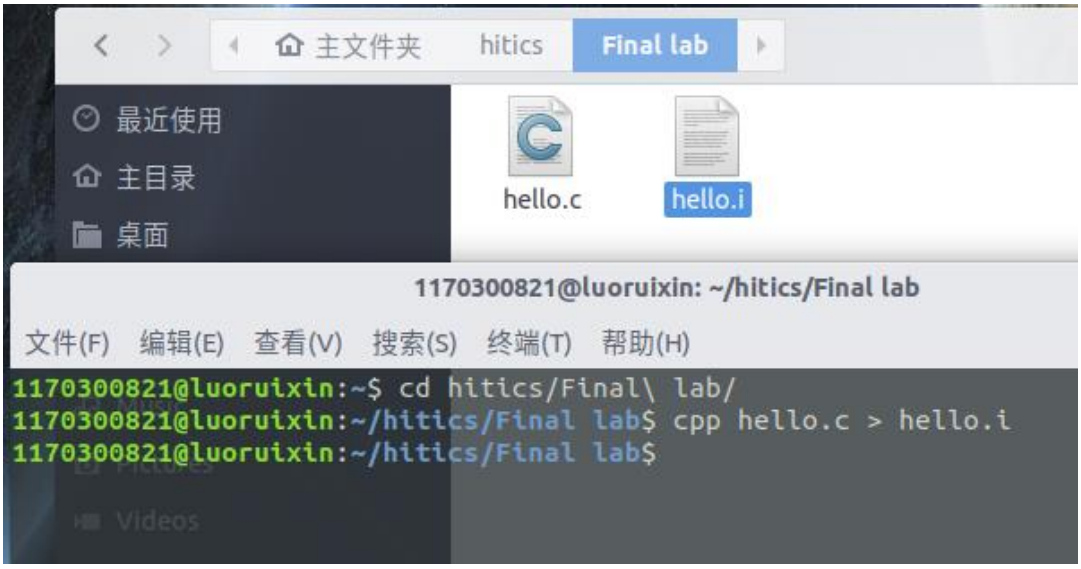
2.1 预处理的概念与作用

预处理中会展开以#起始的行，试图解释为预处理指令(preprocessing directive)，其中 ISO C/C++要求支持的包括#if、#ifdef、#ifndef、#else、#elif、#endif（条件编译）、#define（宏定义）、#include（源文件包含）、#line（行控制）、#error（错误指令）、#pragma（和实现相关的杂注）以及单独的#（空指令）。预处理指令一般被用来使源代码在不同的执行环境中被方便的修改或者编译。

1. 将源文件中用#include 形式声明的文件复制到新的程序中。比如 hello.c 第 6-8 行中的#include<stdio.h> 等命令告诉预处理器读取系统头文件 stdio.h unistd.h stdlib.h 的内容，并把它直接插入到程序文本中。
2. 用实际值替换用#define 定义的字符串
3. 根据#if 后面的条件决定需要编译的代码
4. 特殊符号，预编译程序可以识别一些特殊的符号，预编译程序对于在源程序中出现的这些串将用合适的值进行替换。

2.2 在 Ubuntu 下预处理的命令

命令：cpp hello.c > hello.i



2.3 Hello 的预处理结果解析

使用 vim 打开 hello.i 之后发现，整个 hello.i 程序已经拓展为 3188 行，main 函数出现在 hello.c 中的代码自 3099 行开始。

在这之前出现的是头文件 stdio.h unistd.h stdlib.h 的依次展开。以 stdio.h 的展开为例：stdio.h 是标准库文件，cpp 到 Ubuntu 中默认的环境变量下寻找 stdio.h，打开文件/usr/include/stdio.h ，发现其中依然使用了#define 语句，cpp 对 stdio 中的 define 宏定义递归展开。所以最终的.i 文件中是没有#define 的；发现其中使用了大量的#ifdef #ifndef 条件编译的语句，cpp 会对条件值进行判断来决定是否执行包含其中的逻辑。特殊符号，预编译程序可以识别一些特殊的符号，预编译程序对于在源程序中出现的这些串将用合适的值进行替换。

2.4 本章小结

hello.c 功能的实现需要很多很多标准（套路）的函数等支持，套路之多，犹记起第一次用 hexedit 时，反汇编 printf(“hello world”);好几千行的不知所措。。。

本章主要介绍了预处理的定义与作用、并结合预处理之后的程序对预处理结果进行了解析。

（第 2 章 0.5 分）

第 3 章 编译

3.1 编译的概念与作用

编译程序所要作得工作就是通过词法分析和语法分析，在确认所有的指令都符合语法规则之后，将其翻译成等价的中间代码表示或汇编代码。编译器将文本文件 hello.i 翻译成文本文件 hello.s，它包含一个汇编语言程序。 这个过程称为编译，同时也是编译的作用。

编译包括以下基本流程：

(1) 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，方法分为两种：自上而下分析法和自下而上分析法。

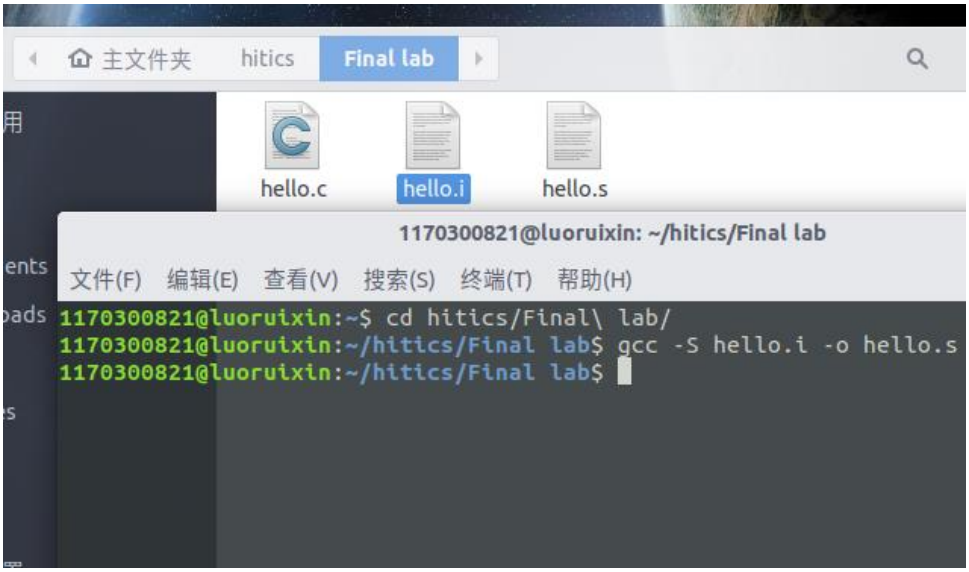
(2) 中间代码：源程序的一种内部表示，或称中间语言。中间代码的作用是可使编译程序的结构在逻辑上更为简单明确，特别是可使目标代码的优化比较容易实现中间代码。

(3) 代码优化：指对程序进行多种等价变换，使得从变换后的程序出发，能生成更有效的目标代码。

(4) 目标代码：生成是编译的最后一个阶段。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。此处指汇编语言代码，须经过汇编程序汇编后，成为可执行的机器语言代码。

3.2 在 Ubuntu 下编译的命令

命令：gcc -S hello.i -o hello.s



3.3 Hello 的编译结果解析

3.3.1 汇编指令

指令	含义
.file	声明源文件
.text	以下是代码段
.section .rodata	以下是 rodata 节
.globl	声明一个全局变量
.type	用来指定是函数类型或是对象类型
.size	声明大小
.long、.string	声明一个 long、string 类型
.align	声明对指令或者数据的存放地址进行对齐的方式

3.3.2 数据类型

hello.s 中用到的数据类型有：整数、字符串、数组。

3.3.2.1 字符串

程序中的字符串分别是：

(1) “Usage: Hello 1170300821 罗瑞欣！\n”，第一个 printf 传入的输出格式化参数，在 hello.s 中声明,可以发现字符串被编码成 UTF-8 格式,一个数字一个字节，一个汉字在 utf-8 编码中占三个字节，一个\代表一个字节。

(2) “Hello %s %s\n”，第二个 printf 传入的输出格式化参数，在 hello.s 中。其中后两个字符串都声明在了.rodata 只读数据节。

```
.LC0:
    .string "Usage: Hello 1170300821 \347\275\227\347\221\236\346\254\243\357\274\201"
.LC1:
    .string "Hello %s %s\n"
```

3.3.2.2 整数

程序中涉及的整数有：

(1) int sleepsecs：sleepsecs 在 hello.c 中被声明为全局变量，且已经被赋值，编译器处理时在.data 节声明该变量（.data 节存放已经初始化的全局和静态 C 变量）。编译器首先将 sleepsecs 在.text 代码段中声明为全局变量，其次在.data 段中，设置对齐方式为 4、设置类型为对象、设置大小为 4 字节、设置为 long 类型其值为 2。

```
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   .align 8
```

(2) int i：编译器将局部变量存储在寄存器或者栈空间中，在 hello.s 中编译器将 i 存储在栈上空间-4(%rbp)中，可看出 i 占据了栈中的 4B。

```
36 .L2:
37     movl    $0, -4(%rbp)
38     jmp .L3
```

(3) int argc：作为第一个参数传入。

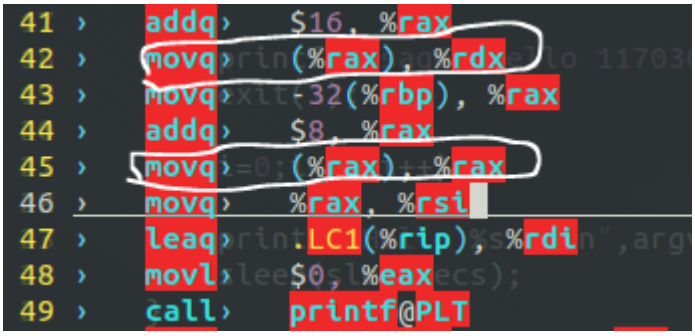
(4) 立即数：其他整形数据的出现都是以立即数的形式出现的，直接硬编码在汇编代码中。

3.3.2.3 数组

程序中涉及数组的是：char *argv[]和 int argv。 前者是 main 函数执行时输入的命令行；后者作为存放 char 指针的数组同时是第二个参数传入。

argv 单个元素 char*大小为 8B，argv 指针指向存放着字符指针的连

续空间（已分配），起始地址为 argv。main 函数中访问数组元素 argv[1],argv[2]时，按照起始地址 argv 大小 8B 计算数据地址取数据，在 hello.s 中，使用两次(%rax)（两次 rax 分别为 argv[1]和 argv[2]的地址）取出其值。



3.3.3 数据类型

程序中涉及的赋值操作有：

(1) int sleepsecs=2.5 ：因为 sleepsecs 是全局变量，所以直接在.data 节中将 sleepsecs 声明为值 2 的 long 类型数据。

(2) i=0：整型数据的赋值使用 mov 指令完成，根据数据的大小不同使用不同后缀，分别为：

指令	b	w	l	q
大小	8b (1B)	16b (2B)	32b (4B)	64b (8B)

3.3.4 类型转换

程序中涉及隐式类型转换的是：int sleepsecs=2.5，将浮点数类型的 2.5 转换为 int 类型。

当浮点型向 int 进行类型转换的时候，程序改变数值和位模式的原则是：值会向零舍入。进一步来讲，类型转换可能会产生值溢出的情况，与 Intel 兼容的微处理器指定位模式[10…000]为整数不确定值，一个浮点数到整数的转换，如果不能为该浮点数找到一个合适的整数近似值，就会产生一个整数不确定值。

浮点数默认类型为 double，所以上述强制转化是 double 强制转化为 int 类型。遵从向零舍入的原则，将 2.5 舍入为 2。

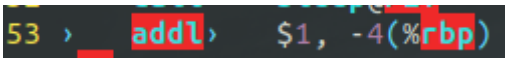
3.3.5 算数操作

进行数据算数操作的汇编指令有：

指令	效果
leaq s,d	d=&s
inc d	d+=1
dec d	d-=1
neg d	d=-d
add s,d	d=d+s
sub s,d	d=d-s
imulq s	r[%rdx]:r[%rax]=s*r[%rax]（有符号）
mulq s	r[%rdx]:r[%rax]=s*r[%rax]（无符号）
idivq s	r[%rdx]=r[%rdx]:r[%rax] mod s（有符号） r[%rax]=r[%rdx]:r[%rax] div s
divq s	r[%rdx]=r[%rdx]:r[%rax] mod s（无符号） r[%rax]=r[%rdx]:r[%rax] div s

程序中涉及的算数操作有：

(1) i++，对计数器 i 自增，使用程序指令 addl，后缀 l 代表操作数是一个 4B 大小的数据。



(2) 汇编中使用 leaq .LC1(%rip),%rdi，使用了加载有效地址指令 leaq 计算 LC1 的段地址%rip+.LC1 并传递给%rdi。

3.3.5 关系操作

进行关系操作的汇编指令有：

指令	效果	描述
CMP S1,S2	S2-S1	比较-设置条件码
TEST S1,S2	S1&S2	测试-设置条件码
SET** D	D=**	按照**将条件码设置
D J**	——	根据**与条件码进行跳转

程序中涉及的关系运算为：

(1) argc!=3：判断 argc 不等于 3。hello.s 中使用 cmpl $3,-20(%rbp)，计算 argc-3 然后设置条件码，为下一步 je 利用条件码进行跳转作准备。

```
30 ›    cmpl›  $3, -20(%rbp)
31 ›    je›    .L2
```

(2) i<10：判断 i 小于 10。hello.s 中使用 cmpl $9,-4(%rbp)，计算 i-9 然后设置条件码，为下一步 jle 利用条件码进行跳转做准备。

```
55 ›    cmpl›   $9, -4(%rbp)
56 ›    jle›    .L4
```

3. 3. 5 控制转移

程序中涉及的控制转移有：

(1) if (argv!=3) jemp L2； ：当 argv 不等于 3 的时候执行程序段中的代码。

```
30 ›    cmpl›   $3, -20(%rbp)
31 ›    je›     .L2
32 ›    leaq›   .LC0(%rip), %rdi
33 ›    call›   puts@PLT
34 ›    movl›   $1, %edi
35 ›    call›   exit@PLT
36 .L2:
37 ›    movl›   $0, -4(%rbp)
38 ›    jmp›    .L3
```

对于 if 判断，编译器使用跳转指令实现。首先 cmpl 比较 argv 和 3，设置条件码，使用 je 判断 ZF 标志位，如果为 0，说明 argv-3=0，则不执行 if 中的代码直接跳转到.L2，否则顺序执行下一条语句，即执行 if 中的代码。

```
if(argc!=3)
{
    printf("Usage: Hello 1170300821 罗瑞欣! \n");
    exit(1);
}
```

(2) for(i=0;i<10;i++) ：使用计数变量 i 循环 10 次。

```
39 .L4:
40 ›    movq›   -32(%rbp), %rax
41 ›    addq›   $16, %rax
42 ›    movq›   (%rax), %rdx
43 ›    movq›   -32(%rbp), %rax
44 ›    addq›   $8, %rax
45 ›    movq›   (%rax), %rax
46 ›    movq›   %rax, %rsi
47 ›    leaq›   .LC1(%rip), %rdi
48 ›    movl›   $0, %eax
49 ›    call›   printf@PLT
50 ›    movl›   sleepsecs(%rip), %eax
51 ›    movl›   %eax, %edi
52 ›    call›   sleep@PLT
53 ›    addl›   $1, -4(%rbp)
54 .L3:
55 ›    cmpl›   $9, -4(%rbp)
56 ›    jle›    .L4
```

编译器的编译逻辑是，首先无条件跳转到位于循环体.L4 之后的比较

代码，使用 cmpl 进行比较，如果 i<=9，则跳入.L4 for 循环体执行，否则说明循环结束，顺序执行 for 之后的逻辑。

3.3.5 函数操作

C 语言中，子程序的作用是由一个主函数和若干个函数构成。由主函数调用其他函数，其他函数也可以互相调用。同一个函数可以被一个或多个函数调用任意多次。在程序设计中，常将一些常用的功能模块编写成函数，放在函数库中供公共选用。要善于利用函数，以减少重复编写程序段的工作量。

函数包括如下内容：

(1) 函数表达式：函数作为表达式中的一项出现在表达式中，以函数返回值参与表达式的运算。这种方式要求函数是有返回值的。

(2) 函数语句：函数调用的一般形式加上分号即构成函数语句。

(3) 函数实参：函数作为另一个函数调用的实际参数出现。这种情况是把该函数的返回值作为实参进行传送，因此要求该函数必须是有返回值的。

调用函数的动作如下：

(1) 传递控制：进行过程 Q 的时候，程序计数器必须设置为 Q 的代码的起始地址，然后在返回时，要把程序计数器设置为 P 中调用 Q 后面那条指令的地址。

(2) 传递数据：P 必须能够向 Q 提供一个或多个参数，Q 必须能够向 P 中返回一个值。

(3) 分配和释放内存：在开始时，Q 可能需要为局部变量分配空间，而在返回前，又必须释放这些空间。

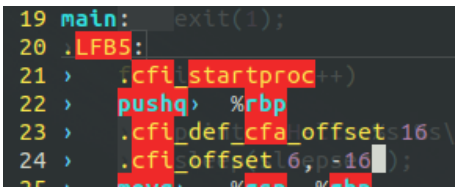
64 位程序参数存储顺序：

1	2	3	4	5	6	7
%rdi	%rsi	%rdx	%rcx	%r8	%r9	栈空间

浮点数使用 xmm

程序中涉及函数操作的有：

(1) main 函数：



a) 传递控制，main 函数因为被调用 call 才能执行（被系统启动函数 __libc_start_main 调用），call 指令将下一条指令的地址 dest 压栈，然后跳转到 main 函数。

b) 传递数据，外部调用过程向 main 函数传递参数 argc 和 argv，分别使用%rdi 和%rsi 存储，函数正常出口为 return 0，将%eax 设置 0 返回。

c) 分配和释放内存，使用%rbp 记录栈帧的底，函数分配栈帧空间在%rbp 之上，程序结束时，调用 leave 指令，leave 相当于 mov %rbp,%rsp,pop %rbp，恢复栈空间为调用之前的状态，然后 ret 返回，ret 相当 pop IP，将下一条要执行指令的地址设置为 dest。

(2) printf 函数：

```
33 ›    call›   puts@PLT
```

a) 传递数据：第一次 printf 将%rdi 设置为“Usage: Hello 学号 姓名！\n”字符串的首地址。第二次 printf 设置%rdi 为“Hello %s %s\n”的首地址，设置%rsi 为 argv[1]，%rdx 为 argv[2]。

b) 控制传递：第一次 printf 因为只有一个字符串参数，所以 callputs@PLT；第二次 printf 使用 call printf@PLT。

(3) exit 函数：

```
35 ›    call›   exit@PLT
```

a) 传递数据：将%edi 设置为 1。

b) 控制传递：call exit@PLT。

(4) sleep 函数：

```
52 ›    call›   sleep@PLT
```

a) 传递数据：将%edi 设置为 sleepsecs。

b) 控制传递：call sleep@PLT

(5) getchar 函数：

```
57 ›    call›   getchar@PLT
```

a) 控制传递：call gethcar@PLT

3.4 本章小结

本章主要阐述了编译器是如何处理 C 语言的各个数据类型以及各类操作的，基本都是先给出原理然后结合 hello.c C 程序到 hello.s 汇编代码之间的映射关系作出合理解释。编译器将.i 的拓展程序编译为.s 的汇编代码。经过编译之后，我们

的 hello 自 C 语言解构为更加低级的汇编语言。

（第 3 章 2 分）

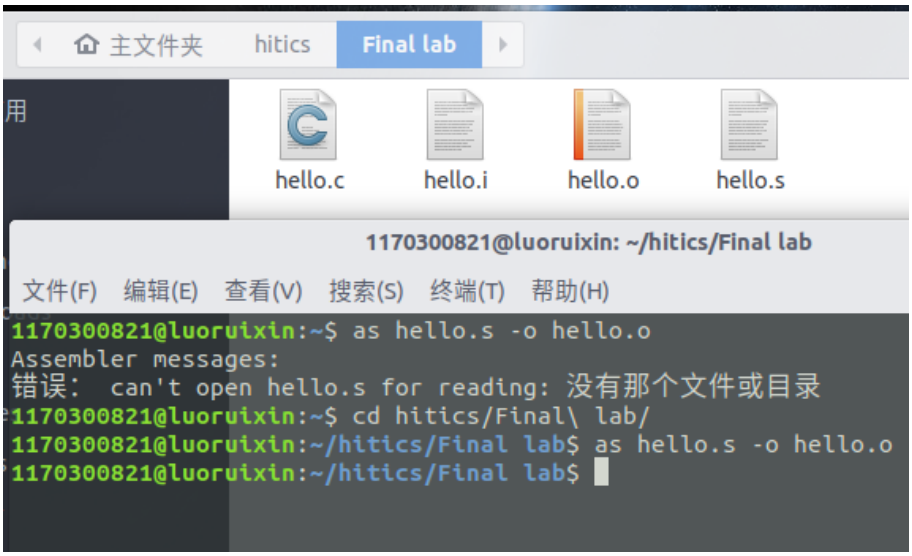
第 4 章 汇编

4.1 汇编的概念与作用

汇编器（as）将.s 汇编程序翻译成机器语言指令，把这些指令打包成可重定位目标程序的格式，并将结果保存在.o 目标文件中，.o 文件是一个二进制文件，它包含程序的指令编码。这个过程称为汇编，亦即汇编的作用。

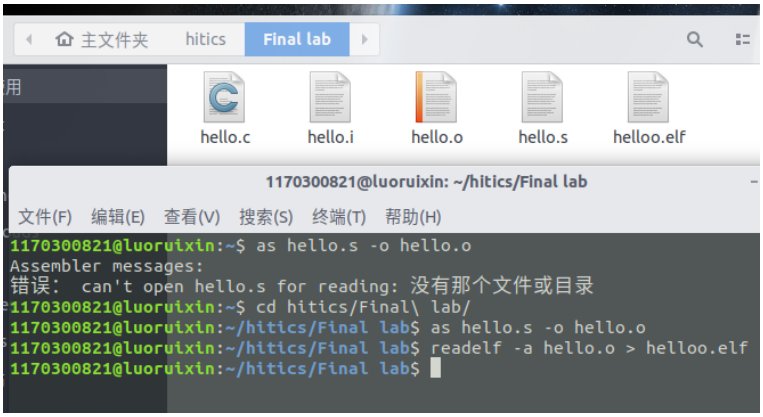
4.2 在 Ubuntu 下汇编的命令

指令：as hello.s -o hello.o



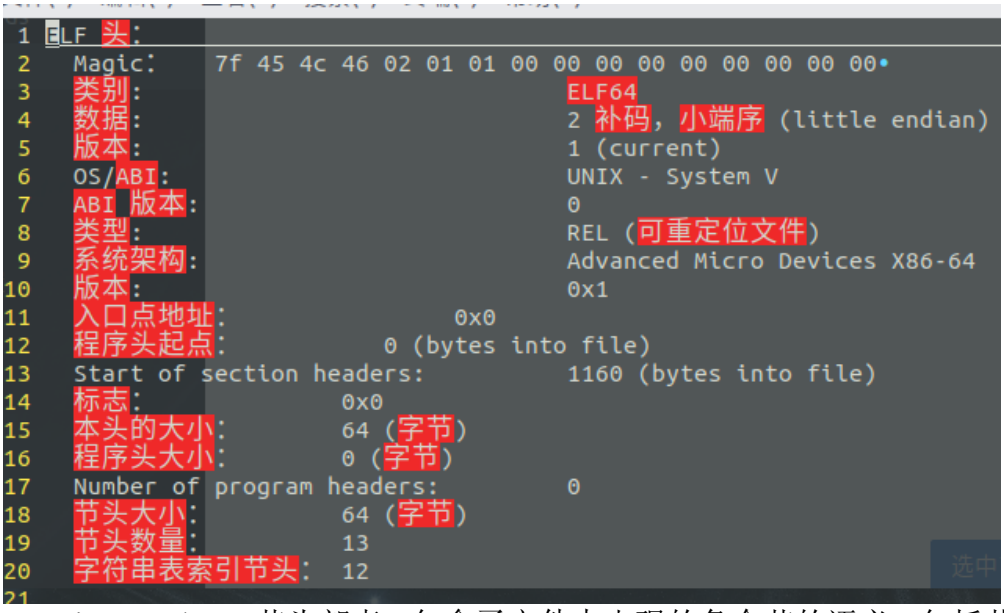
4.3 可重定位目标 elf 格式

使用 readelf -a hello.o > helloo.elf 指令获得 hello.o 文件的 ELF 格式。



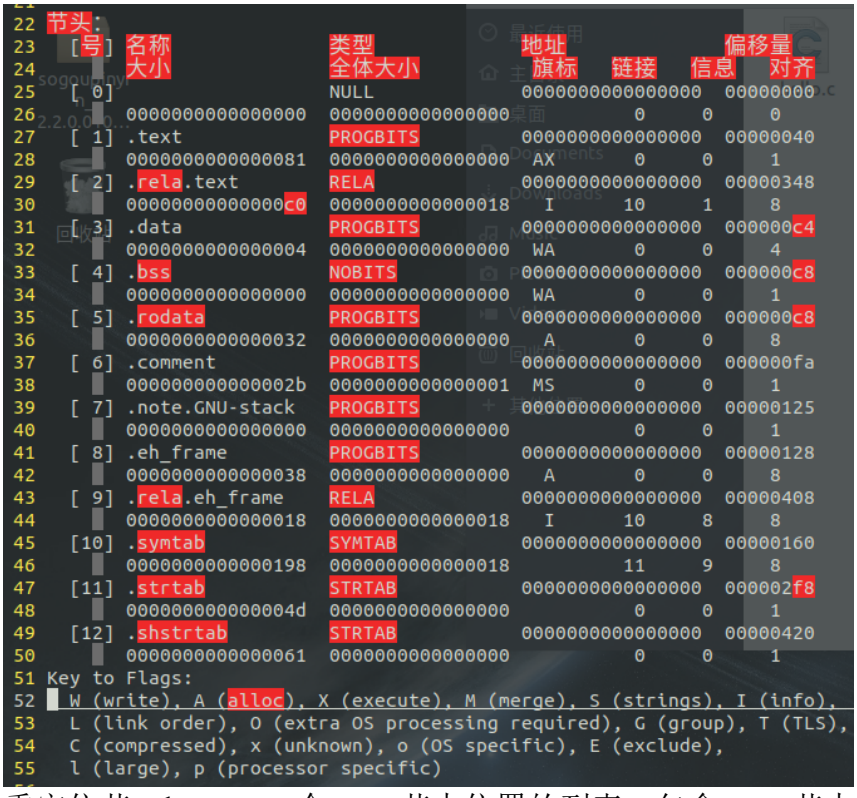
其组成如下：

(1) ELF Header：以 16B 的序列 Magic 开始，Magic 描述了生成该文件的系统的字的大小和字节顺序，ELF 头剩下的部分包含帮助链接器语法分析和解释目标文件的信息，其中包括 ELF 头的大小、目标文件的类型、机器类型、字节头部表（section header table）的文件偏移，以及节头部表中条目的大小和数量等信息。



```
ELF 头：
  Magic：   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  类别：                             ELF64
  数据：                             2 补码，小端序 (little endian)
  版本：                             1 (current)
  OS/ABI：                           UNIX - System V
  ABI 版本：                         0
  类型：                             REL (可重定位文件)
  系统架构：                         Advanced Micro Devices X86-64
  版本：                             0x1
  入口点地址：               0x0
  程序头起点：          0 (bytes into file)
  Start of section headers:          1160 (bytes into file)
  标志：             0x0
  本头的大小：       64 (字节)
  程序头大小：       0 (字节)
  Number of program headers:         0
  节头大小：         64 (字节)
  节头数量：         13
  字符串表索引节头： 12
```

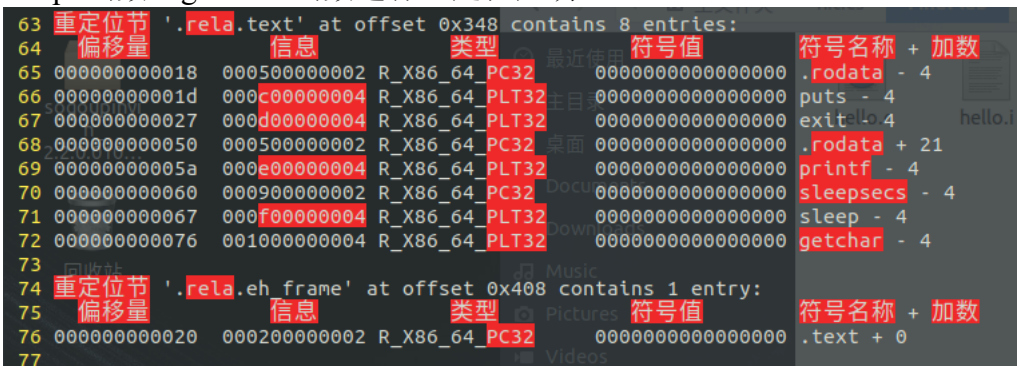
(2) Section Headers：节头部表，包含了文件中出现的各个节的语义，包括节的类型、位置和大小等信息。



```
节头：
  [号] 名称              类型             地址              偏移量
       大小              全体大小         旗标   链接   信息   对齐
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0      0     0
  [ 1] .text             PROGBITS         0000000000000000  00000040
       0000000000000081  0000000000000000  AX       0      0     1
  [ 2] .rela.text        RELA             0000000000000000  00000348
       00000000000000c0  0000000000000018   I       10     1     8
  [ 3] .data             PROGBITS         0000000000000000  000000c4
       0000000000000004  0000000000000000  WA       0      0     4
  [ 4] .bss              NOBITS           0000000000000000  000000c8
       0000000000000000  0000000000000000  WA       0      0     1
  [ 5] .rodata           PROGBITS         0000000000000000  000000c8
       0000000000000032  0000000000000000   A       0      0     8
  [ 6] .comment          PROGBITS         0000000000000000  000000fa
       000000000000002b  0000000000000001  MS       0      0     1
  [ 7] .note.GNU-stack   PROGBITS         0000000000000000  00000125
       0000000000000000  0000000000000000           0      0     1
  [ 8] .eh_frame         PROGBITS         0000000000000000  00000128
       0000000000000038  0000000000000000   A       0      0     8
  [ 9] .rela.eh_frame    RELA             0000000000000000  00000408
       0000000000000018  0000000000000018   I       10     8     8
  [10] .symtab           SYMTAB           0000000000000000  00000160
       0000000000000198  0000000000000018           11     9     8
  [11] .strtab           STRTAB           0000000000000000  000002f8
       000000000000004d  0000000000000000           0      0     1
  [12] .shstrtab         STRTAB           0000000000000000  00000420
       0000000000000061  0000000000000000           0      0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

(3) 重定位节.rela.text ,一个.text 节中位置的列表，包含.text 节中需要进行重

定位的信息，当链接器把这个目标文件和其他文件组合时，需要修改这些位置。8 条重定位信息分别是对.L0（第一个 printf 中的字符串）、puts 函数、exit 函数、.L1（第二个 printf 中的字符串）、printf 函数、sleepsecs、sleep 函数、getchar 函数进行重定位声明。



.rela 节的包含的信息有：

offset	需要进行重定向的代码在.text 或.data 节中的偏移位置，8 个字节。
Info	包括 symbol 和 type 两部分，其中 symbol 占前 4 个字节，type 占后 4 个字节，symbol 代表重定位到的目标在.symtab 中的偏移量，type 代表重定位的类型
Addend	计算重定位位置的辅助信息，共占 8 个字节
Type	重定位到的目标的类型
Name	重定向到的目标的名称

下面以.L1 的重定位为例阐述之后的重定位过程：链接器根据 info 信息向.symtab 节中查询链接目标的符号，由 info.symbol=0x05，可以发现重定位目标链接到.rodata 的.L1，设重定位条目为 r，

r 的构造为：r.offset=0x18, r.symbol=.rodata, r.type=R_X86_64_PC32, r.addend=-4

重定位一个使用 32 位 PC 相对地址的引用。计算重定位目标地址的算法如下（设需要重定位的.text 节中的位置为 src,设重定位的目的位置 dst）：

refptr = s +r.offset （1）

refaddr = ADDR(s) + r.offset （2）

*refptr = (unsigned) (ADDR(r.symbol) + r.addend-refaddr)（3）

其中（1）指向 src 的指针（2）计算 src 的运行时地址，（3）中，

ADDR(r.symbol)计算 dst 的运行时地址，在本例中，ADDR(r.symbol)获得的是 dst 的运行时地址，因为需要设置的是绝对地址，即 dst 与下一条指令之间的地址之差，所以需要加上 r.addend=-4。之后将 src 处设

置为运行时值*refptr，完成该处重定位。

(4) 符号表（Symbol Table）目标文件的符号表中包含用来定位、重定位程序中符号定义和引用的信息。符号表索引是对此数组的索引。索引 0 表示表中的第一表项，同时也作为定义符号的索引。

```
80 Symbol table '.symtab' contains 17 entries:
81    Num:    Value          Size Type    Bind   Vis      Ndx Name
82      0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
83      1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS hello.c
84      2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
85      3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
86      4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
87      5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
88      6: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
89      7: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
90      8: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
91      9: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 sleepsecs
92     10: 0000000000000000   129 FUNC    GLOBAL DEFAULT    1 main
93     11: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _GLOBAL_OFFSET_TABLE_
94     12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
95     13: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND exit
96     14: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
97     15: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND sleep
98     16: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND getchar
99
```

(5) 程序头部表（Program Header Table），ELF 文件头结构就像是一个总览图，描述了整个文件的布局情况。因此在 ELF 文件头结构允许的数值范围内，整个文件的大小是可以动态增减的。如果存在的话，告诉系统如何创建进程映像。

很显然，对于 hello.o（换言之 hello.c），不存在程序头

```
59 本文件中没有程序头。
60
```

(6) 节头表（section header），通过目标文件中的节头表，我们就可以轻松地定位文件中所有的节。节头表是若干结构 Elf32_Shdr 或者结构 Elf64_Shdr 组成的数组。节头表索引（section header table index）是用来定位节头表的表项的。ELF 文件头中的 e_shoff 代表的是节头表在文件中的偏移值；e_shnum 代表的是节头表的表项总数；e_shentsize 代表的是节头表的表项大小。

这个也没有。。。

```
57 There are no section groups in this file.
58
```

(7) Dynamic section，如果目标文件参与动态链接，则其程序头表将包含一个类型为 PT_DYNAMIC 的元素。此段包含 .dynamic 节。特殊符号 _DYNAMIC 用于标记包含以下结构的数组的节

这个还是没有。。。。。。

```
61 There is no dynamic section in this file.
```

4.4 Hello.o 的结果解析

使用 objdump -d -r hello.o > helloo.objdump 获得反汇编代码。Hello.s 和 helloo.objdunp 除去显示格式之外两者差别不大，主要差别如下：

(1) 分支转移：反汇编代码跳转指令的操作数使用的不是段名称如.L3，因为段名称只是在汇编语言中便于编写的助记符，所以在汇编成机器语言之后显然不存在，而是确定的地址。

(2) 函数调用：在.s 文件中，函数调用之后直接跟着函数名称，而在反汇编序中，call 的目标地址是当前下一条指令。这是因为 hello.c 中调用的函数都是共享库中的函数，最终需要通过动态链接器才能确定函数的运行时执行地址，在汇编成为机器语言的时候，对于这些不确定地址的函数调用，将其 call 指令后的相对地址设置为全 0（目标地址正是下一条指令），然后在.rela.text 节中为其添加重定位条目，等待静态链接的进一步确定。

(3) 全局变量访问：在.s 文件中，访问 rodata（printf 中的字符串），使用段名称+%rip，在反汇编代码中 0+%rip，因为 rodata 中数据地址也是在运行时确定，故访问也需要重定位。所以在汇编成为机器语言时，将操作数设置为全 0 并添加重定位条目。

```
hello.o:     文件格式 elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 83 ec 20             sub    $0x20,%rsp
   8:   89 7d ec                mov    %edi,-0x14(%rbp)
   b:   48 89 75 e0             mov    %rsi,-0x20(%rbp)
   f:   83 7d ec 03             cmpl   $0x3,-0x14(%rbp)
  13:   74 16                   je     2b <main+0x2b>
  15:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 1c <main+0x1c>
                        18: R_X86_64_PC32       .rodata-0x4
  1c:   e8 00 00 00 00          callq  21 <main+0x21>
                        1d: R_X86_64_PLT32      puts-0x4
  21:   bf 01 00 00 00          mov    $0x1,%edi
  26:   e8 00 00 00 00          callq  2b <main+0x2b>
                        27: R_X86_64_PLT32      exit-0x4
  2b:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  32:   eb 3b                   jmp    6f <main+0x6f>
  34:   48 8b 45 e0             mov    -0x20(%rbp),%rax
  38:   48 83 c0 10             add    $0x10,%rax
  3c:   48 8b 10                mov    (%rax),%rdx
  3f:   48 8b 45 e0             mov    -0x20(%rbp),%rax
  43:   48 83 c0 08             add    $0x8,%rax
  47:   48 8b 00                mov    (%rax),%rax
  4a:   48 89 c6                mov    %rax,%rsi
  4d:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 54 <main+0x54>
                        50: R_X86_64_PC32       .rodata+0x21
  54:   b8 00 00 00 00          mov    $0x0,%eax
  59:   e8 00 00 00 00          callq  5e <main+0x5e>
                        5a: R_X86_64_PLT32      printf-0x4
  5e:   8b 05 00 00 00 00       mov    0x0(%rip),%eax        # 64 <main+0x64>
                        60: R_X86_64_PC32       sleepsecs-0x4
  64:   89 c7                   mov    %eax,%edi
  66:   e8 00 00 00 00          callq  6b <main+0x6b>
                        67: R_X86_64_PLT32      sleep-0x4
  6b:   83 45 fc 01             addl   $0x1,-0x4(%rbp)
  6f:   83 7d fc 09             cmpl   $0x9,-0x4(%rbp)
  73:   7e bf                   jle    34 <main+0x34>
  75:   e8 00 00 00 00          callq  7a <main+0x7a>
                        76: R_X86_64_PLT32      getchar-0x4
  7a:   b8 00 00 00 00          mov    $0x0,%eax
  7f:   c9                      leaveq
  80:   c3                      retq
```

4.5 本章小结

本章介绍了 hello 从 hello.s 到 hello.o 的汇编过程，通过查看 hello.o 的 elf 格式和使用 objdump 得到反汇编代码与 hello.s 进行比较的方式，间接了解到从汇编语言映射到机器语言汇编器需要实现的转换

（第 4 章 1 分）

第 5 章 链接

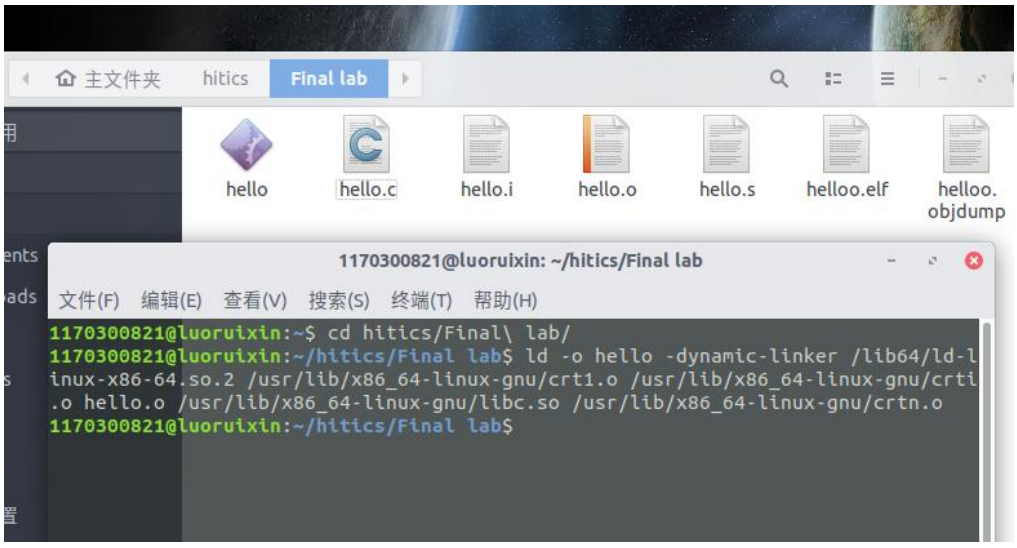
5.1 链接的概念与作用

链接是将各种代码和数据片段收集并组合成一个单一文件的过程，这个文件可被加载到内存并执行。链接可以执行于编译时，也就是在源代码被编译成机器代码时；也可以执行于加载时，也就是在程序被加载器加载到内存并执行时；甚至于运行时，也就是由应用程序来执行。链接是由叫做链接器的程序执行的。链接器使得分离编译成为可能。

5.2 在 Ubuntu 下链接的命令

动态链接器和链接的目标文件都是 64 位的。

ld -o hello -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o hello.o /usr/lib/x86_64-linux-gnu/libc.so /usr/lib/x86_64-linux-gnu/crtn.o



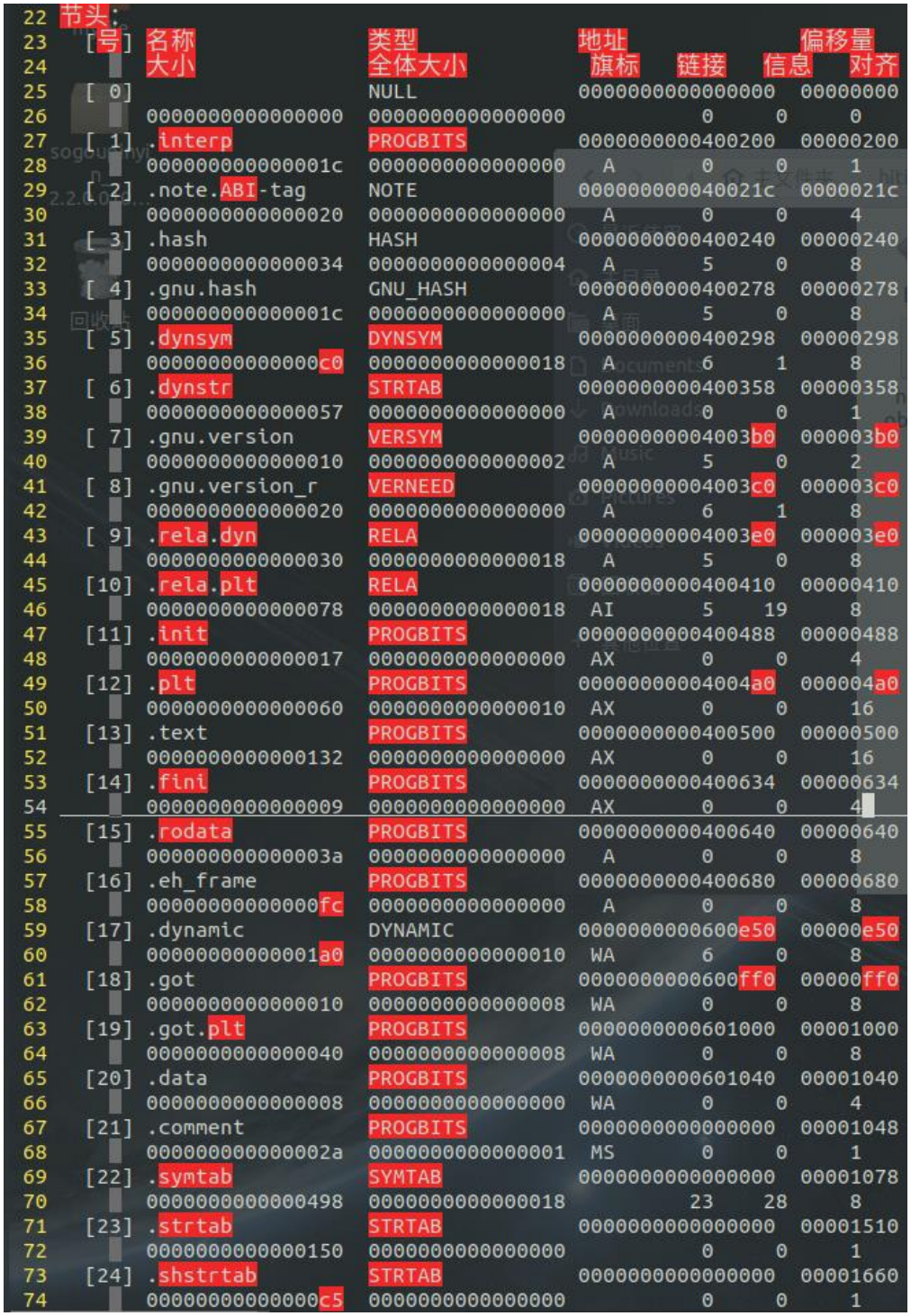
5.3 可执行目标文件 hello 的格式

分析 hello 的 ELF 格式，用 readelf 等列出其各段的基本信息，包括各段的起始地址，大小等信息。

使用 readelf -a hello > hello.elf 命令生成 hello 程序的 ELF 格式文件。

(1) 在 ELF 格式文件中，Section Headers 对 hello 中所有的节信息进行了声明，其中包括大小 Size 以及在程序中的偏移量 Offset，因此根据 Section

Headers 中的信息我们就可以用 HexEdit 定位各个节所占的区间（起始位置，大小）。其中 Address 是程序被载入到虚拟地址的起始地址。



```
节头：
  [号] 名称              类型             地址              偏移量
       大小              全体大小         旗标   链接   信息   对齐
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000400200  00000200
       000000000000001c  0000000000000000  A        0     0     1
  [ 2] .note.ABI-tag     NOTE             000000000040021c  0000021c
       0000000000000020  0000000000000000  A        0     0     4
  [ 3] .hash             HASH             0000000000400240  00000240
       0000000000000034  0000000000000004  A        5     0     8
  [ 4] .gnu.hash         GNU_HASH         0000000000400278  00000278
       000000000000001c  0000000000000000  A        5     0     8
  [ 5] .dynsym           DYNSYM           0000000000400298  00000298
       00000000000000c0  0000000000000018  A        6     1     8
  [ 6] .dynstr           STRTAB           0000000000400358  00000358
       0000000000000057  0000000000000000  A        0     0     1
  [ 7] .gnu.version      VERSYM           00000000004003b0  000003b0
       0000000000000010  0000000000000002  A        5     0     2
  [ 8] .gnu.version_r    VERNEED          00000000004003c0  000003c0
       0000000000000020  0000000000000000  A        6     1     8
  [ 9] .rela.dyn         RELA             00000000004003e0  000003e0
       0000000000000030  0000000000000018  A        5     0     8
  [10] .rela.plt         RELA             0000000000400410  00000410
       0000000000000078  0000000000000018  AI       5    19     8
  [11] .init             PROGBITS         0000000000400488  00000488
       0000000000000017  0000000000000000  AX       0     0     4
  [12] .plt              PROGBITS         00000000004004a0  000004a0
       0000000000000060  0000000000000010  AX       0     0    16
  [13] .text             PROGBITS         0000000000400500  00000500
       0000000000000132  0000000000000000  AX       0     0    16
  [14] .fini             PROGBITS         0000000000400634  00000634
       0000000000000009  0000000000000000  AX       0     0     4
  [15] .rodata           PROGBITS         0000000000400640  00000640
       000000000000003a  0000000000000000  A        0     0     8
  [16] .eh_frame         PROGBITS         0000000000400680  00000680
       00000000000000fc  0000000000000000  A        0     0     8
  [17] .dynamic          DYNAMIC          0000000000600e50  00000e50
       00000000000001a0  0000000000000010  WA       6     0     8
  [18] .got              PROGBITS         0000000000600ff0  00000ff0
       0000000000000010  0000000000000008  WA       0     0     8
  [19] .got.plt          PROGBITS         0000000000601000  00001000
       0000000000000040  0000000000000008  WA       0     0     8
  [20] .data             PROGBITS         0000000000601040  00001040
       0000000000000008  0000000000000000  WA       0     0     4
  [21] .comment          PROGBITS         0000000000000000  00001048
       000000000000002a  0000000000000001  MS       0     0     1
  [22] .symtab           SYMTAB           0000000000000000  00001078
       0000000000000498  0000000000000018          23    28     8
  [23] .strtab           STRTAB           0000000000000000  00001510
       0000000000000150  0000000000000000           0     0     1
  [24] .shstrtab         STRTAB           0000000000000000  00001660
       00000000000000c5  0000000000000000           0     0     1
```

(2) ELF Header：以 16B 的序列 Magic 开始，Magic 描述了生成该文件的系统的字的大小和字节顺序，ELF 头剩下的部分包含帮助链接器语法分析和解释目标文件的信息，其中包括 ELF 头的大小、目标文件的类型、机器类型、字节头部表（section header table）的文件偏移，以及节头部表中条目

的大小和数量等信息。

与链接前的 ELF header 比较，可见除系统决定的基本信息不变外，section header 和 program header 均增加，并获得了入口地址。

```
ELF 头:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  类别:                              ELF64
  数据:                              2 补码，小端序 (little endian)
  版本:                              1 (current)
  OS/ABI:                            UNIX - System V
  ABI 版本:                          0
  类型:                              EXEC (可执行文件)
  系统架构:                          Advanced Micro Devices X86-64
  版本:                              0x1
  入口点地址:               0x400500
  程序头起点:               64 (bytes into file)
  Start of section headers:          5928 (bytes into file)
  标志:                     0x0
  本头的大小:               64 (字节)
  程序头大小:               56 (字节)
  Number of program headers:         8
  节头大小:                 64 (字节)
  节头数量:                 25
  字符串表索引节头:         24
```

(3) 程序头部表（Program Header Table），ELF 文件头结构就像是一个总览图，描述了整个文件的布局情况。因此在 ELF 文件头结构允许的数值范围内，整个文件的大小是可以动态增减的。告诉系统如何创建进程映像。

```
程序头:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001c0 0x00000000000001c0  R      0x8
  INTERP         0x0000000000000200 0x0000000000400200 0x0000000000400200
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x000000000000077c 0x000000000000077c  R E    0x200000
  LOAD           0x0000000000000e50 0x0000000000600e50 0x0000000000600e50
                 0x00000000000001f8 0x00000000000001f8  RW     0x200000
  DYNAMIC        0x0000000000000e50 0x0000000000600e50 0x0000000000600e50
                 0x00000000000001a0 0x00000000000001a0  RW     0x8
  NOTE           0x000000000000021c 0x000000000040021c 0x000000000040021c
                 0x0000000000000020 0x0000000000000020  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000000e50 0x0000000000600e50 0x0000000000600e50
                 0x00000000000001b0 0x00000000000001b0  R      0x1
```

段映射（segment mapping），映射操作仅适用于本机架构互连，如 Dolphin-SCI 或 NewLink。映射段授予 CPU 内存操作访问该段的权限，从而节省调用内存访问原语的开销。

```
Section to Segment mapping:
段节...
  00
  01     .interp
  02     .interp .note.ABI-tag .hash .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .text .fini .rodata .eh_frame
  03     .dynamic .got .got.plt .data
  04     .dynamic
  05     .note.ABI-tag
  06
  07     .dynamic .got
```

(4) Dynamic section，如果目标文件参与动态链接，则其程序头表将包含一个类型为 PT_DYNAMIC 的元素。此段包含 .dynamic 节。特殊符号 _DYNAMIC 用于标记包含以下结构的数组的节

```
115 Dynamic section at offset 0xe50 contains 21 entries:
116   标记        类型                         名称/值
117  0x0000000000000001 (NEEDED)             共享库: [libc.so.6]
118  0x000000000000000c (INIT)               0x400488
119  0x000000000000000d (FINI)               0x400634
120  0x0000000000000004 (HASH)               0x400240
121  0x000000006ffffef5 (GNU_HASH)           0x400278
122  0x0000000000000005 (STRTAB)             0x400358
123  0x0000000000000006 (SYMTAB)             0x400298
124  0x000000000000000a (STRSZ)              87 (bytes)
125  0x000000000000000b (SYMENT)             24 (bytes)
126  0x0000000000000015 (DEBUG)              0x0
127  0x0000000000000003 (PLTGOT)             0x601000
128  0x0000000000000002 (PLTRELSZ)           120 (bytes)
129  0x0000000000000014 (PLTREL)             RELA
130  0x0000000000000017 (JMPREL)             0x400410
131  0x0000000000000007 (RELA)               0x4003e0
132  0x0000000000000008 (RELASZ)             48 (bytes)
133  0x0000000000000009 (RELAENT)            24 (bytes)
134  0x000000006ffffffe (VERNEED)            0x4003c0
135  0x000000006fffffff (VERNEEDNUM)         1
136  0x000000006ffffff0 (VERSYM)             0x4003b0
137  0x0000000000000000 (NULL)               0x0
```

(5) 重定位节.rela.text ,一个.text 节中位置的列表，包含.text 节中需要进行重定位的信息，当链接器把这个目标文件和其他文件组合时，需要修改这些位置。六条重定位信息，分别描述了原 hello 的函数 main、标准头文件的函数 puts、函数 printf、函数 getchar、函数 exit、函数 sleep 的重定位声明

```
139 重定位节 '.rela.dyn' at offset 0x3e0 contains 2 entries:
140   偏移量          信息           类型            符号值        符号名称 + 加数
141 000000600ff0  000300000006 R_X86_64_GLOB_DAT 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0
142 000000600ff8  000500000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0
143
144 重定位节 '.rela.plt' at offset 0x410 contains 5 entries:
145   偏移量          信息           类型            符号值        符号名称 + 加数
146 000000601018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
147 000000601020  000200000007 R_X86_64_JUMP_SLO 0000000000000000 printf@GLIBC_2.2.5 + 0
148 000000601028  000400000007 R_X86_64_JUMP_SLO 0000000000000000 getchar@GLIBC_2.2.5 + 0
149 000000601030  000600000007 R_X86_64_JUMP_SLO 0000000000000000 exit@GLIBC_2.2.5 + 0
150 000000601038  000700000007 R_X86_64_JUMP_SLO 0000000000000000 sleep@GLIBC_2.2.5 + 0
```

(6) 符号表节（symbol table），目标文件的符号表包含定位和重定位程序的符号定义和符号引用所需的信息。符号表索引是此数组的下标。索引 0 指定表中的第一项并用作未定义的符号索引。

动态符号表 (.dynsym) 用来保存与动态链接相关的导入导出符号，不包括模块内部的符号。

```
154 Symbol table '.dynsym' contains 8 entries:
155    Num:    Value          Size Type    Bind   Vis      Ndx Name
156      0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
157      1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
158      2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.2.5 (2)
159      3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
160      4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND getchar@GLIBC_2.2.5 (2)
161      5: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
162      6: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND exit@GLIBC_2.2.5 (2)
163      7: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND sleep@GLIBC_2.2.5 (2)
```

其中 .symtab 段只保存函数名和变量名等基本的符号的地址和长度等信息。

```
Symbol table '.symtab' contains 49 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000400200     0 SECTION LOCAL  DEFAULT    1
     2: 000000000040021c     0 SECTION LOCAL  DEFAULT    2
     3: 0000000000400240     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000400278     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000400298     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000400358     0 SECTION LOCAL  DEFAULT    6
     7: 00000000004003b0     0 SECTION LOCAL  DEFAULT    7
     8: 00000000004003c0     0 SECTION LOCAL  DEFAULT    8
     9: 00000000004003e0     0 SECTION LOCAL  DEFAULT    9
    10: 0000000000400410     0 SECTION LOCAL  DEFAULT   10
    11: 0000000000400488     0 SECTION LOCAL  DEFAULT   11
    12: 00000000004004a0     0 SECTION LOCAL  DEFAULT   12
    13: 0000000000400500     0 SECTION LOCAL  DEFAULT   13
    14: 0000000000400634     0 SECTION LOCAL  DEFAULT   14
    15: 0000000000400640     0 SECTION LOCAL  DEFAULT   15
    16: 0000000000400680     0 SECTION LOCAL  DEFAULT   16
    17: 0000000000600e50     0 SECTION LOCAL  DEFAULT   17
    18: 0000000000600ff0     0 SECTION LOCAL  DEFAULT   18
    19: 0000000000601000     0 SECTION LOCAL  DEFAULT   19
    20: 0000000000601040     0 SECTION LOCAL  DEFAULT   20
    21: 0000000000000000     0 SECTION LOCAL  DEFAULT   21
    22: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS hello.c
    23: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS
    24: 0000000000600e50     0 NOTYPE  LOCAL  DEFAULT   17 __init_array_end
    25: 0000000000600e50     0 OBJECT  LOCAL  DEFAULT   17 _DYNAMIC
    26: 0000000000600e50     0 NOTYPE  LOCAL  DEFAULT   17 __init_array_start
    27: 0000000000601000     0 OBJECT  LOCAL  DEFAULT   19 _GLOBAL_OFFSET_TABLE_
    28: 0000000000400630     2 FUNC    GLOBAL DEFAULT   13 __libc_csu_fini
    29: 0000000000601040     0 NOTYPE  WEAK   DEFAULT   20 data_start
    30: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@@GLIBC_2.2.5
    31: 0000000000601044     4 OBJECT  GLOBAL DEFAULT   20 sleepsecs
    32: 0000000000601048     0 NOTYPE  GLOBAL DEFAULT   20 _edata
    33: 0000000000400634     0 FUNC    GLOBAL DEFAULT   14 _fini
    34: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@@GLIBC_2.2.5
    35: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@@GLIBC_
    36: 0000000000601040     0 NOTYPE  GLOBAL DEFAULT   20 __data_start
    37: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND getchar@@GLIBC_2.2.5
    38: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    39: 0000000000400640     4 OBJECT  GLOBAL DEFAULT   15 _IO_stdin_used
    40: 00000000004005c0   101 FUNC    GLOBAL DEFAULT   13 __libc_csu_init
    41: 0000000000601048     0 NOTYPE  GLOBAL DEFAULT   20 _end
    42: 0000000000400530     2 FUNC    GLOBAL HIDDEN    13 _dl_relocate_static_pie
    43: 0000000000400500    43 FUNC    GLOBAL DEFAULT   13 _start
    44: 0000000000601048     0 NOTYPE  GLOBAL DEFAULT   20 __bss_start
    45: 0000000000400532   129 FUNC    GLOBAL DEFAULT   13 main
    46: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND exit@@GLIBC_2.2.5
    47: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND sleep@@GLIBC_2.2.5
    48: 0000000000400488     0 FUNC    GLOBAL DEFAULT   11 _init
```

5.4 hello 的虚拟地址空间

在 0x400000~0x401000 段中，程序被载入，自虚拟地址 0x400000 开始，自 0x400fff 结束，这之间每个节（开始 ~.eh_frame 节）的排列即开始结束 Address 中声明。

查看 ELF 格式文件中的 Program Headers，程序头表在执行的时候被使用，告诉链接器运行时加载的内容并提供动态链接的信息。每一个表项提供了各段在虚拟地址空间和物理地址空间的大小、位置、标志、访问权限和对齐方面的信息。

在下面可以看出，程序包含 8 个段：

(1) PHDR 保存程序头表。

(2) INTERP 指定在程序已经从可执行文件映射到内存之后，必须调用的解释

器（如动态链接器）。

(3) LOAD 表示一个需要从二进制文件映射到虚拟地址空间的段。其中保存了常量数据（如字符串）、程序的目标代码等。

(4) DYNAMIC 保存了由动态链接器使用的信息。

(5) NOTE 保存辅助信息。

(6) GNU_STACK：权限标志，标志栈是否是可执行的。

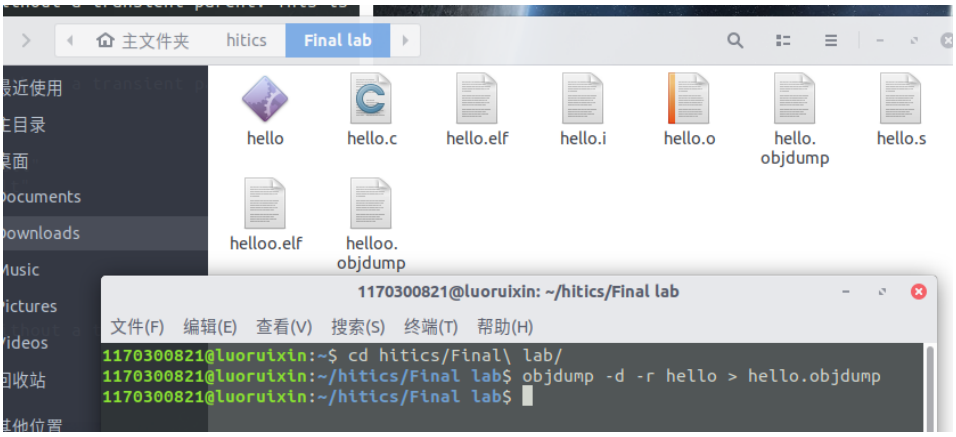
(7) GNU_RELRO：指定在重定位结束之后那些内存区域是需要设置只读。

```
程序头：
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001c0 0x00000000000001c0  R      0x8
  INTERP         0x0000000000000200 0x0000000000400200 0x0000000000400200
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x000000000000077c 0x000000000000077c  R E    0x200000
  LOAD           0x0000000000000e50 0x0000000000600e50 0x0000000000600e50
                 0x00000000000001f8 0x00000000000001f8  RW     0x200000
  DYNAMIC        0x0000000000000e50 0x0000000000600e50 0x0000000000600e50
                 0x00000000000001a0 0x00000000000001a0  RW     0x8
  NOTE           0x000000000000021c 0x000000000040021c 0x000000000040021c
                 0x0000000000000020 0x0000000000000020  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000000e50 0x0000000000600e50 0x0000000000600e50
                 0x00000000000001b0 0x00000000000001b0  R      0x1
```

通过 Data Dump 查看虚拟地址段 0x600000~0x602000，在 0~fff 空间中，与 0x400000~0x401000 段存放的程序相同，在 fff 之后存放的是.dynamic~.shstrtab 节。

5.5 链接的重定位过程分析

使用 objdump -d -r hello > hello.objdump 获得 hello 的反汇编代码。



与 hello.o 反汇编文本 helloo.objdump 相比，在 hello.objdump 中多了许多节，列在下面。

节名称	描述
.interp	保存 ld.so 的路径

.note.ABI-tag	Linux 下特有的 section
.hash	符号的哈希表
.gnu.hash	GNU 拓展的符号的哈希表
.dynsym	运行时/动态符号表
.dynstr	存放.dynsym 节中的符号名称
.gnu.version	符号版本
.gnu.version_r	符号引用版本
.rela.dyn	运行时/动态重定位表
.rela.plt	.plt 节的重定位条目
.init	程序初始化需要执行的代码
.plt	动态链接-过程链接表
.fini	当程序正常终止时需要执行的代码
.eh_frame	包含异常展开和源语言信息。
.dynamic	存放被 ld.so 使用的动态链接信息
.got	动态链接-全局偏移量表-存放变量
.got.plt	动态链接-全局偏移量表-存放函数
.data	初始化了的数据
.comment	包含编译器的 NULL-terminated 字符串

通过比较 hello.objdump 和 helloo.objdump 了解链接器。

(1) 函数个数：在使用 ld 命令链接的时候，指定了动态链接器为 64 的/lib64/ld-linux-x86-64.so.2，crt1.o、crti.o、crtn.o 中主要定义了程序入口_start、初始化函数_init，_start 程序调用 hello.c 中的 main 函数，libc.so 是动态链接共享库，其中定义了 hello.c 中用到的 printf、sleep、getchar、exit 函数和_start 中调用的__libc_csu_init，__libc_csu_fini，__libc_start_main。链接器将上述函数加入。

(2) 函数调用：链接器解析重定条目时以类型 R_X86_64_PLT32 对外部函数调用重定位，此时 PLT 中已经加入到动态链接库中的函数，链接器根据已经确定的.text 与.plt 节的相对距离计算相对距离，将对动态链接库中函数的调用值改为 PLT 中相应函数与下条指令的相对地址，指向对应函数。对于此类重定位链接器为其构造.plt 与.got.plt。（链接器根据相对距离修改调用值）

(3) .rodata 引用：链接器解析重定条目时以 R_X86_64_PC32 类型对.rodata 两个重定位（printf 中的两个字符串），确定.rodata 与.text 节之间的相对距离，call 之后的值被链接器直接修改为目标地址与下一条指令的地址之差，指向相应的字符串。（取相对距离确定相对距离）

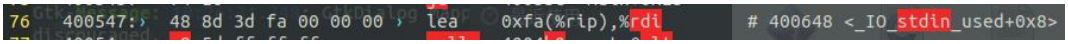
以计算第一条字符串相对地址为例说明计算相对地址的算法：

refptr = s + r.offset = Pointer to 0x40054A

refaddr = ADDR(s) + r.offset= ADDR(main)+r.offset=0x400532+0x18=0x40054A

*refptr = (unsigned) (ADDR(r.symbol) + r.addend-refaddr) =ADDR(str1)+r.addend-refaddr=0x400644+(-0x4)-0x40054A=(unsigned) 0xF6，

观察反汇编验证计算：



5.6 hello 的执行流程

使用 edb 执行 hello，观察函数执行流程，将过程中执行的主要函数列在下面

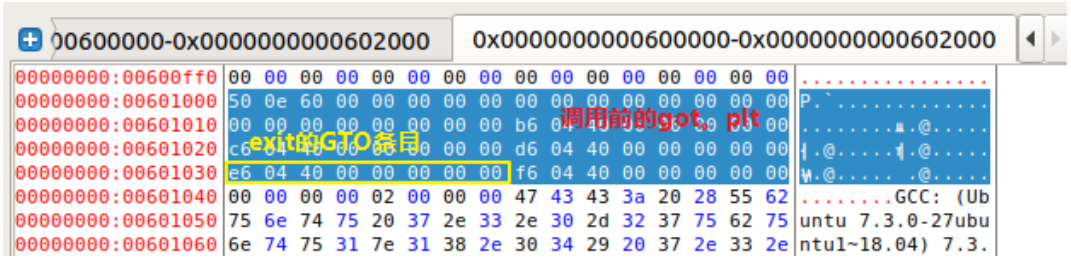
程序名称	程序地址
ld-2.27.so!_dl_start	0x7fce 8cc38ea0
ld-2.27.so!_dl_init	0x7fce 8cc47630
hello!_start	0x400500
libc-2.27.so!__libc_start_main	0x7fce 8c867ab0
-libc-2.27.so!__cxa_atexit	0x7fce 8c889430
-libc-2.27.so!__libc_csu_init	0x4005c0
hello!_init	0x400488
libc-2.27.so!_setjmp	0x7fce 8c884c10
-libc-2.27.so!_sigsetjmp	0x7fce 8c884b70
--libc-2.27.so!__sigjmp_save	0x7fce 8c884bd0
hello!main	0x400532
hello!puts@plt	0x4004b0
hello!exit@plt	0x4004e0
*hello!printf@plt	--
*hello!sleep@plt	--
*hello!getchar@plt	--
ld-2.27.so!_dl_runtime_resolve_xsave	0x7fce 8cc4e680
-ld-2.27.so!_dl_fixup	0x7fce 8cc46df0
--ld-2.27.so!_dl_lookup_symbol_x	0x7fce 8cc420b0
libc-2.27.so!exit	0x7fce 8c889128

5.7 Hello 的动态链接分析

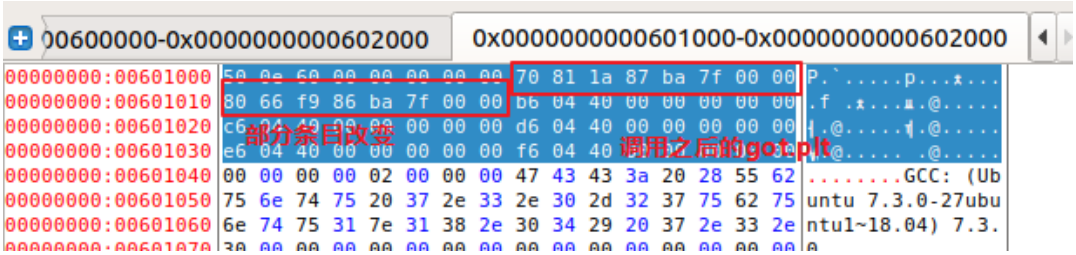
由于无法预测函数的运行时地址，对于动态共享链接库中 PIC 函数，编译器需要添加重定位记录，等待动态链接器处理。链接器采用延迟绑定的策略，防止运行时修改调用模块的代码段。

动态链接器使用过程链接表 PLT+全局偏移量表 GOT 实现函数的动态链接，GOT 中存放函数目标地址，PLT 使用 GOT 中地址跳转到目标函数。

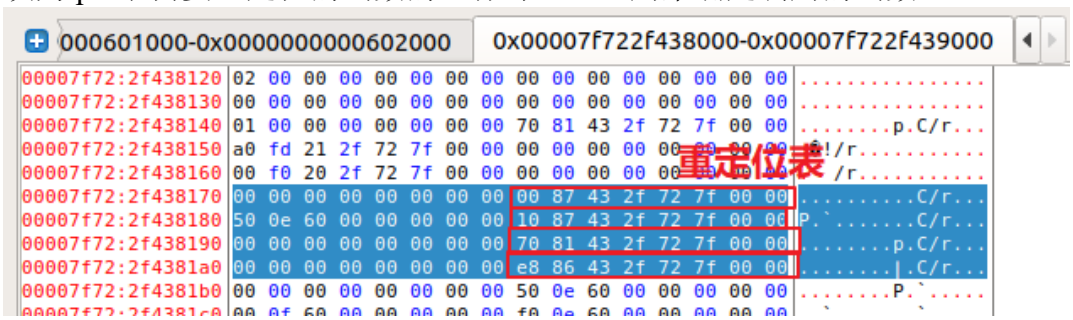
在 dl_init 调用之前，0x601008 和 0x601010 处的两个 8B 数据为空。对于每一条 PIC 函数调用，调用的目标地址都实际指向 PLT 中的代码逻辑，GOT 存放的是 PLT 中函数调用指令的下一条指令地址。



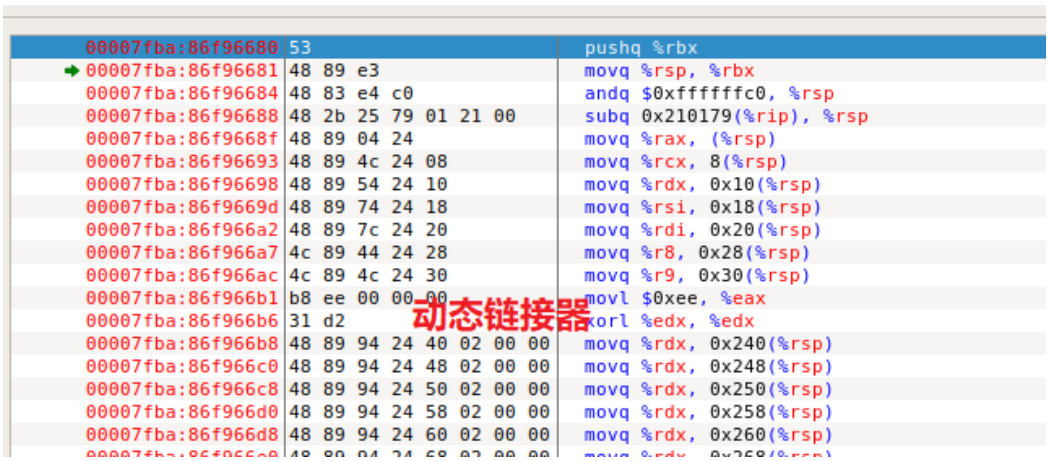
在 dl_init 调用之后，0x601008 和 0x601010 处的两个 8B 数据分别发生改变为 0x7fba 871a8170 和 0x7fba 86f96680。



其中 GOT[1]指向重定位表（另一次运行，不过真的是根据 GOT[1]指向的）（依次为.plt 节需要重定位的函数的运行时地址）用来确定调用的函数地址。



在 dl_init 调用之后的函数调用时，跳转到 PLT 执行.plt 中逻辑，下一条指令压栈，第一次访问跳转时 GOT 地址为函数序号，然后跳转到 PLT[0]。在 PLT[0]中将重定位表地址压栈，然后访问动态链接器。



在动态链接器中使用函数序号和重定位表确定函数运行时地址，重写 GOT，

再将控制传递给目标函数。根据 jmp 的原理（执行完目标函数之后的返回地址为最近 call 指令下一条指令地址），之后如果对同样函数调用，第一次访问跳转直接跳转到目标函数。

5.8 本章小结

关于链接器的相对地址，想起了 linklab 的悲惨经历，话说工大和南大的这门课是什么关系：）

在本章中主要介绍了链接的概念与作用、hello 的 ELF 格式，分析了 hello 的虚拟地址空间、重定位过程、执行流程、动态链接过程。

（第 5 章 1 分）

第 6 章 hello 进程管理

6.1 进程的概念与作用

进程是一个执行中的程序的实例，每一个进程都有它自己的地址空间，一般情况下，包括文本区域、数据区域、和堆栈。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储区着活动过程调用的指令和本地变量。

进程为用户提供了以下假象：我们的程序好像是系统中当前运行的唯一程序一样，我们的程序好像是独占的使用处理器和内存，处理器好像是无间断的执行我们程序中的指令，我们程序中的代码和数据好像是系统内存中唯一的对象。

（源于教材）

6.2 简述壳 Shell-bash 的作用与处理流程

Shell 的作用：在计算机科学中，Shell 俗称壳（用来区别于核），是指“为使用者提供操作界面”的软件（命令解析器）。它类似于 DOS 下的 command.com 和后来的 cmd.exe。它接收用户命令，然后调用相应的应用程序。Shell 是一个用 C 语言编写的程序，他是用户使用 Linux 的桥梁。Shell 是指一种应用程序，Shell 应用程序提供了一个界面，用户通过这个界面访问操作系统内核的服务。

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) 否则调用相应的程序为其分配子进程并运行

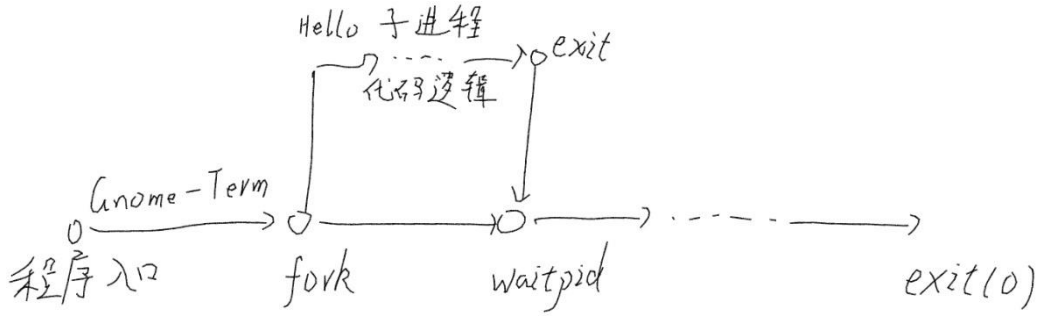
(5) shell 应该接受键盘输入信号，并对这些信号进行相应处理

6.3 Hello 的 fork 进程创建过程

在终端 Terminal 中键入./hello 1170300821 罗瑞欣，运行的终端程序会对输入的命令行进行解析。

1. hello 不是一个内置的 shell 命令所以解析之后终端程序判断./hello 的语义为执行当前目录下的可执行目标文件 hello。
2. 之后终端程序首先会调用 fork 函数创建一个新的运行的子进程，新创建的子进程几乎父进程相同，但不完全与相同。

3. 父进程与子进程之间最大的区别在于它们拥有不同的 PID。子进程得到与父进程用户级虚拟地址空间相同的一份副本，当父进程调用 fork 时，子进程可以读写父进程中打开的任何文件。
4. 内核能够以任意方式交替执行父子进程的逻辑控制流的指令，父进程与子进程是并发运行而独立的。在子进程执行期间，父进程默认选项是显示等待子进程的完成。



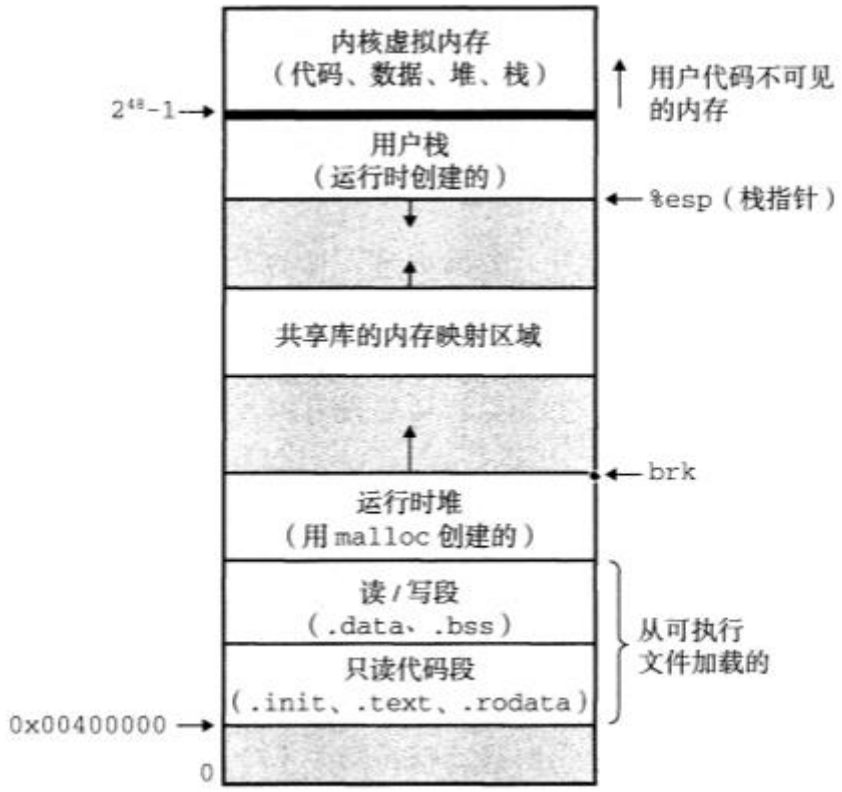
6.4 Hello 的 execve 过程

(1) 为子进程调用函数 fork 之后，子进程调用 execve 函数（传入命令行参数）在当前进程的上下文中加载并运行一个新程序 hello。

(2) 为执行 hello 程序加载器、删除子进程现有的虚拟内存段，execve 调用驻留在内存中的、被称为启动加载器的操作系统代码，并创建一组新的代码、数据、堆和栈段。

(3) 新的栈和堆段被初始化为零，通过将虚拟地址空间中的页映射到可执行文件的页大小的片，新的代码和数据段被初始化为可执行文件中的内容。最后加载器设置 PC 指向_start 地址，_start 最终调用 hello 中的 main 函数。

(4) 除了一些头部信息，在加载过程中没有任何从磁盘到内存的数据复制。直到 CPU 引用一个被映射的虚拟页时才会进行复制，这时，操作系统利用它的页面调度机制自动将页面从磁盘传送到内存。



6.5 Hello 的进程执行

(1) 逻辑控制流：一系列程序计数器 PC 的值的序列叫做逻辑控制流，在同一个处理器核心中，每个进程执行它的流的一部分后被抢占（暂时挂起），然后轮到其他进程，进程轮流使用处理器，。

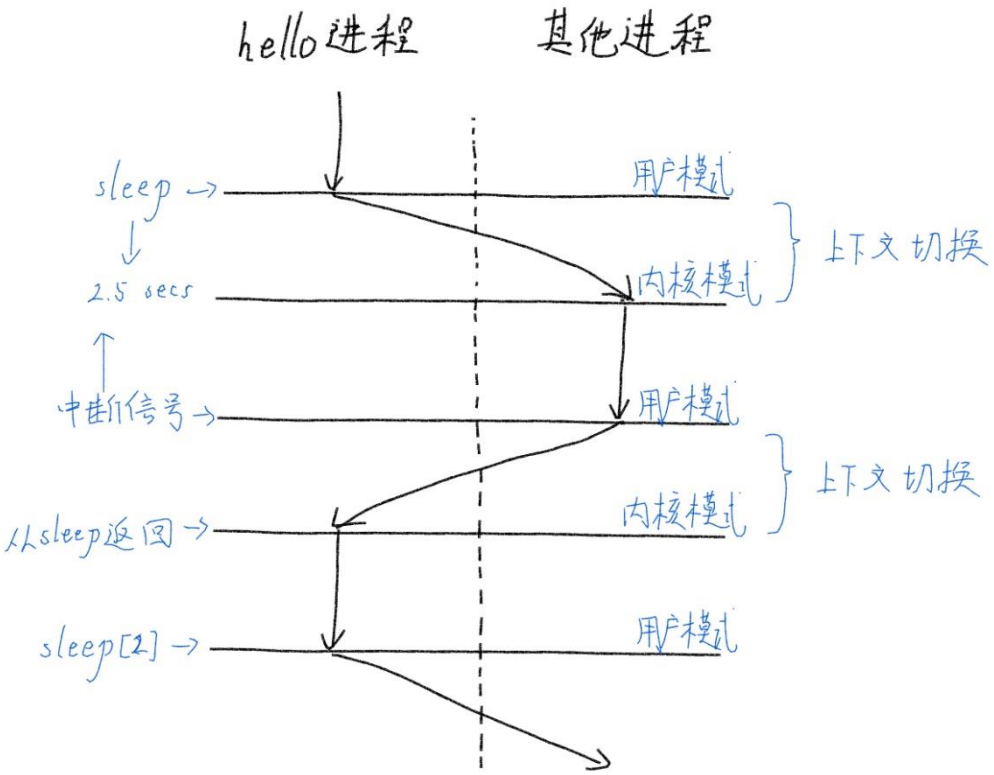
(2) 时间片：一个进程执行它的控制流的一部分的每一时间段叫做时间片。

(3) 用户模式和内核模式：处理器通常使用一个寄存器描述了进程当前享有的特权，对两种模式区分。设置模式位时，进程处于内核模式，该进程可以访问系统中的任何内存位置，可以执行指令集中的任何命令；当没有设置模式位时，进程就处于用户模式中，用户模式的进程不允许执行特权指令，也不允许直接引用地址空间中内核区内的代码和数据。

(4) 上下文信息：上下文就是内核重新启动一个被抢占的进程所需要的状态。它由通用寄存器、浮点寄存器、程序计数器、用户栈、状态寄存器、内核栈和各种内核数据结构等对象的值构成。

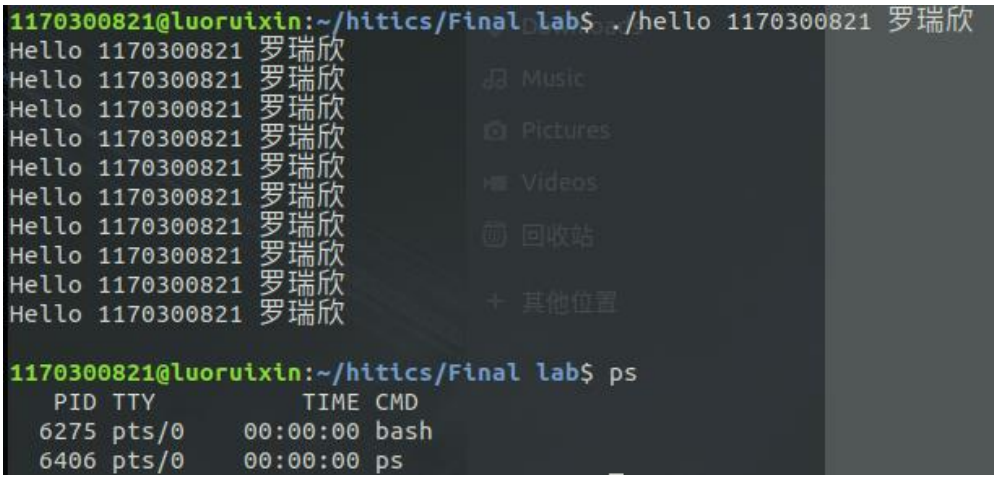
hello sleep 进程调度的过程：

1. 当调用 sleep 之前，如果 hello 程序不被抢占则顺序执行，假如发生被抢占的情况，则进行上下文切换
2. 上下文切换是由内核中调度器完成的，当内核调度新的进程运行后，它就会抢占当前进程，并进行
   a) 保存以前进程的上下文
   b) 恢复新恢复进程被保存的上下文，
   c) 将控制传递给这个新恢复的进程 ，来完成上下文切换。
3. hello 初始运行在用户模式，在 hello 进程调用 sleep 之后陷入内核模式，内核处理休眠请求主动释放当前进程，并将 hello 进程从运行队列中移出加入等待队列
4. 定时器开始计时，内核进行上下文切换将当前进程的控制权交给其他进程，当定时器到时时（2.5secs）发送一个中断信号，
5. 进入内核状态执行中断处理，将 hello 进程从等待队列中移出重新加入到运行队列，成为就绪状态，hello 进程就可以继续进行自己的控制逻辑流了。



6.6 hello 的异常与信号处理

(1) 正常执行 hello 程序的结果：当程序执行完成之后（以键入回车结束），进程回收。

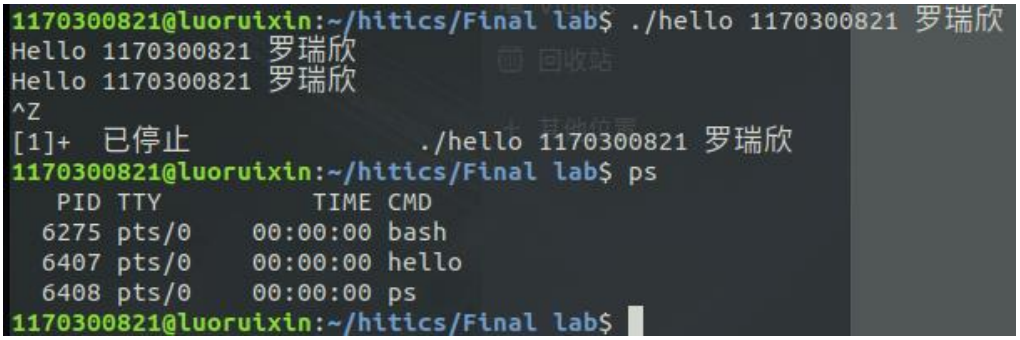


(2) 在程序输出 2 条 info 之后按下 ctrl-z ：

当按下 ctrl-z 之后，shell 父进程收到 SIGSTP 信号，信号处理函数的逻辑是打印屏幕回显、将 hello 进程挂起。

通过 ps 命令我们可以看出 hello 进程没有被回收，此时他的后台 job 号是 1。

调用 fg 1 将其调到前台，此时 shell 程序首先打印 hello 的命令行命令，hello 继续运行打印剩下的 8 条 info，之后输入字串，程序结束，同时进程被回收。



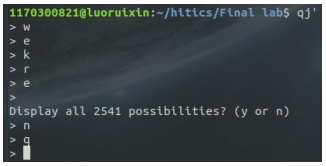
(3) 在程序输出 2 条 info 之后按下 ctrl-c ： 当按下 ctrl-c 之后，shell 父进程收到 SIGINT 信号，信号处理函数的逻辑是结束 hello，并回收 hello 进程。

```
1170300821@luoruixin:~/hitics/Final lab$ ./hello 1170300821 罗瑞欣
Hello 1170300821 罗瑞欣
Hello 1170300821 罗瑞欣
^C
1170300821@luoruixin:~/hitics/Final lab$ ps
   PID TTY          TIME CMD
  6275 pts/0    00:00:00 bash
  6413 pts/0    00:00:00 ps
1170300821@luoruixin:~/hitics/Final lab$
```

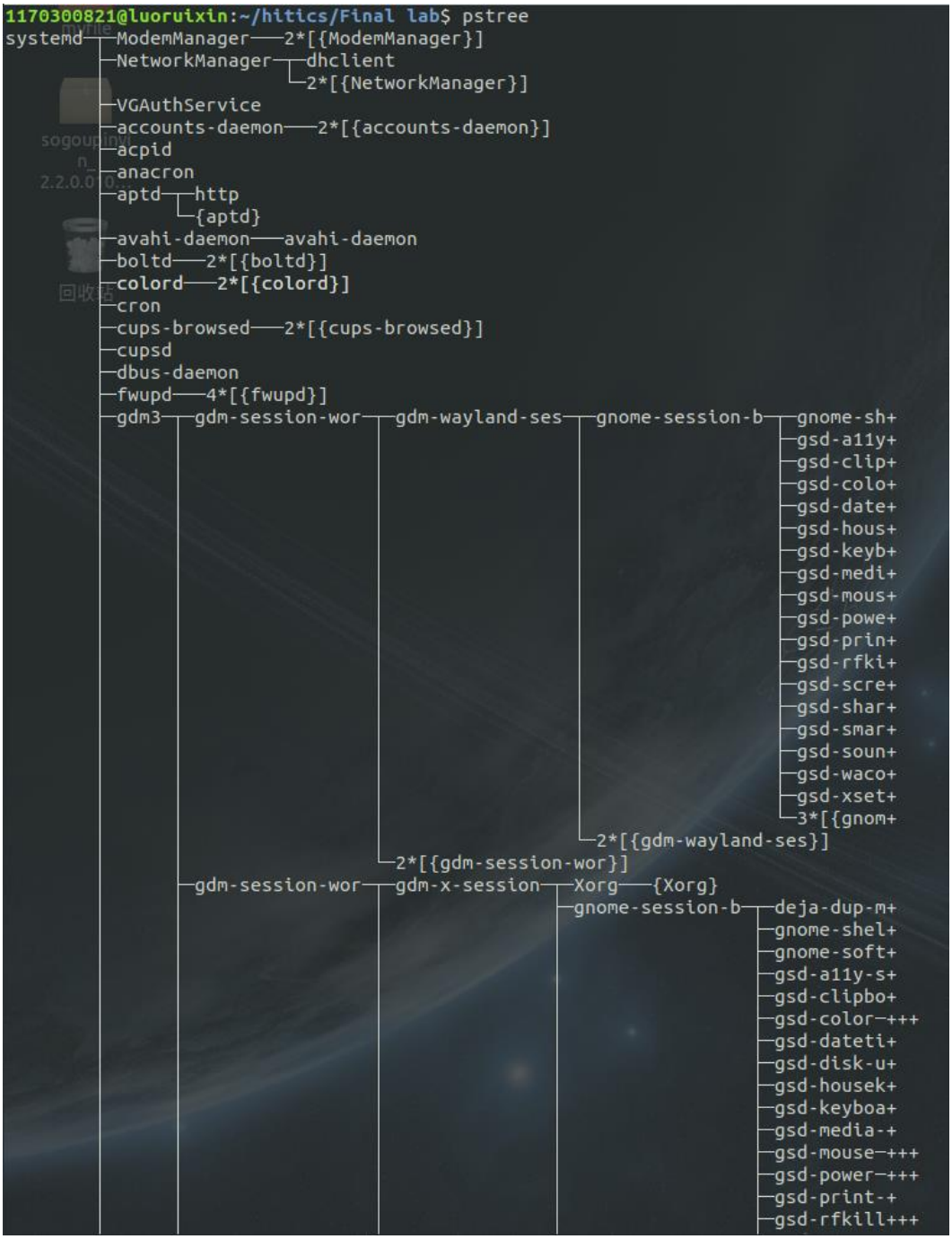
(4) 在程序运行中途乱按的结果：乱按只是将屏幕的输入缓存到 stdin，当 getchar 的时候读出一个'\n'结尾的字串（作为一次输入），hello 结束后，stdin 中的其他字串会当做 shell 若干条命令行输入。

```
1170300821@luoruixin:~/hitics/Final lab$ ./hello 1170300821 罗瑞欣
Hello 1170300821 罗瑞欣
shrekjsfHello 1170300821 罗瑞欣
w
kshfsjf
ssfj
sHello 1170300821 罗瑞欣
f;fj'
asffjk
sf'
fqfjna'
dHello 1170300821 罗瑞欣
q
wk'
wqw'
        r

r       '
        Hello 1170300821 罗瑞欣
k
qj'
w
e
k
r
        e
        Hello 1170300821 罗瑞欣

r
qwr     r'
e       r
        eHello 1170300821 罗瑞欣

qer
        e       wr;
l,,;dlpe
Hello 1170300821 罗瑞欣

ewefjkpfeow[p;w
Hello 1170300821 罗瑞欣
pweldwpke
woHello 1170300821 罗瑞欣
f,ldw
e       k       jqfk
e       1170300821@luoruixin:~/hitics/Final lab$ kshfsjf
kshfsjf: 未找到命令
k1170300821@luoruixin:~/hitics/Final lab$ ssfj
ssfj: 未找到命令
1170300821@luoruixin:~/hitics/Final lab$ sf;fj'
> asffjk
```

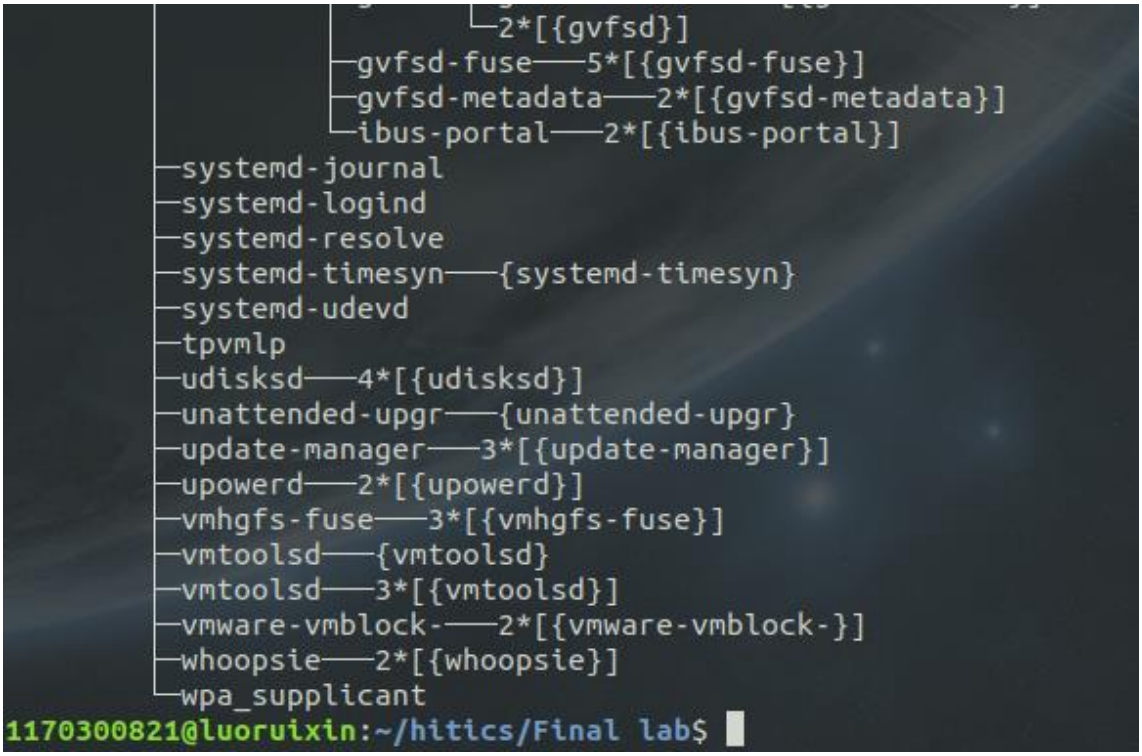
甚至还打开了神奇的东西。。。。



Pstree 如下：




myfile
sogoupinyin_2.2.0.0...
回收站
├─gsd-rfkill+++
├─gsd-screen+
├─gsd-sharin+
├─gsd-smartc+
├─gsd-sound─+++
├─gsd-wacom─+++
├─gsd-xsetti+
├─nautilus-d+
├─ssh-agent
├─update-not+
└─3*[{gnome-+
└─2*[{gdm-x-session}]
└─2*[{gdm-session-wor}]
└─2*[{gdm3}]
├─gnome-keyring-d───3*[{gnome-keyring-d}]
├─gsd-printer───2*[{gsd-printer}]
├─2*[ibus-x11───2*[{ibus-x11}]]
├─irqbalance───{irqbalance}
├─2*[kerneloops]
├─networkd-dispat───{networkd-dispat}
├─packagekitd───2*[{packagekitd}]
├─polkitd───2*[{polkitd}]
├─pulseaudio───2*[{pulseaudio}]
├─rsyslogd───3*[{rsyslogd}]
├─rtkit-daemon───2*[{rtkit-daemon}]
├─snapd───10*[{snapd}]
├─systemd─┬─(sd-pam)
├─at-spi-bus-laun─┬─dbus-daemon
└─3*[{at-spi-bus-laun}]
├─at-spi2-registr───2*[{at-spi2-registr}]
├─dbus-daemon
├─ibus-portal───2*[{ibus-portal}]
└─pulseaudio───2*[{pulseaudio}]
├─systemd─┬─(sd-pam)
├─at-spi-bus-laun─┬─dbus-daemon
└─3*[{at-spi-bus-laun}]
├─at-spi2-registr───2*[{at-spi2-registr}]
├─dbus-daemon
├─dconf-service───2*[{dconf-service}]
├─evolution-addre─┬─evolution-addre───5*[{evolution-addre}]
└─4*[{evolution-addre}]
├─evolution-calen─┬─evolution-calen───8*[{evolution-calen}]
└─4*[{evolution-calen}]
├─evolution-sourc───3*[{evolution-sourc}]
├─gnome-shell-cal───5*[{gnome-shell-cal}]
├─gnome-terminal-─┬─bash───pstree
└─3*[{gnome-terminal-}]
├─goa-daemon───3*[{goa-daemon}]
├─goa-identity-se───3*[{goa-identity-se}]
├─gvfs-afc-volume───3*[{gvfs-afc-volume}]
├─gvfs-goa-volume───2*[{gvfs-goa-volume}]
├─gvfs-gphoto2-vo───2*[{gvfs-gphoto2-vo}]
├─gvfs-mtp-volume───2*[{gvfs-mtp-volume}]
├─gvfs-udisks2-vo───2*[{gvfs-udisks2-vo}]
├─gvfsd─┬─gvfsd-trash───2*[{gvfsd-trash}]



6.7 本章小结

在本章中，阐明了进程的定义与作用，介绍了 Shell 的一般处理流程，调用 fork 创建新进程，调用 execve 执行 hello，hello 的进程执行，hello 的异常与信号处理。

（第 6 章 1 分）

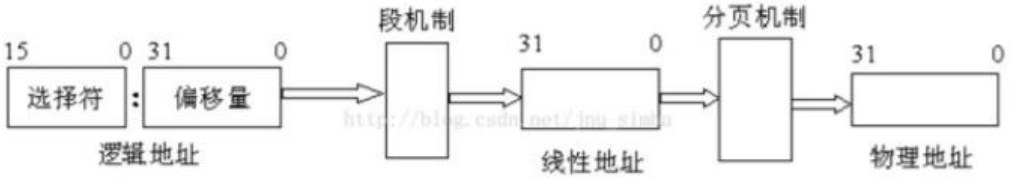
第 7 章 hello 的存储管理

7.1 hello 的存储器地址空间

(1) 逻辑地址：程序代码经过编译后出现在 汇编程序中地址。逻辑地址由选择符（在实模式下是描述符，在保护模式下是用来选择描述符的选择符）和偏移量（偏移部分）组成。

(2) 线性地址：逻辑地址经过段机制后转化为线性地址，为描述符:偏移量的组合形式。分页机制中线性地址作为输入。至于虚拟地址，只关注 CSAPP 课本中提到的虚拟地址，实际上就是这里的线性地址。

(3) 物理地址：CPU 通过地址总线的寻址，找到真实的物理内存对应地址。 CPU 对内存的访问是通过连接着 CPU 和北桥芯片的前端总线来完成的。在前端总线上传输的内存地址都是物理内存地址。



7.2 Intel 逻辑地址到线性地址的变换-段式管理

牙膏厂的心路历程。。。。。。

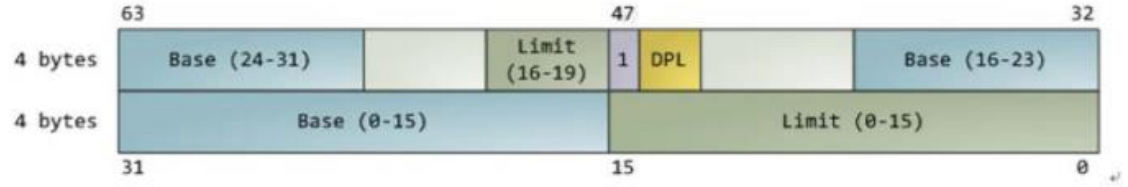
(1) 8086 处理器的寄存器是 16 位的，后来引入了段寄存器，可以访问更多的地址空间但不改变寄存器和指令的位宽。8086 共设计了 20 位宽的地址总线，逻辑地址为段寄存器左移 4 位加上偏移地址得到 20 位地址。

(2) 将内存分为不同的段，段有段寄存器对应，段寄存器有一个栈、一个代码、两个数据寄存器。分段功能在实模式和保护模式下有所不同：

a) 实模式：逻辑地址=线性地址=实际的物理地址，即不设防。段寄存器存放真实段基址，同时给出 32 位地址偏移量，可以访问真实物理内存。

b) 保护模式：线性地址还需要经过分页机制才能够得到物理地址，线性地址也需要逻辑地址通过段机制来得到。32 位段基址被称作选择符，段寄存器无法放下，用于引用段描述符表中的表项来获得描述符。

描述符表中的一个条目描述一个段，构造如下：



Base：基地址，32 位线性地址指向段的开始。

Limit：段界限，段的大小。

DPL：描述符的特权级，包括：0（内核模式）、-3（用户模式）。

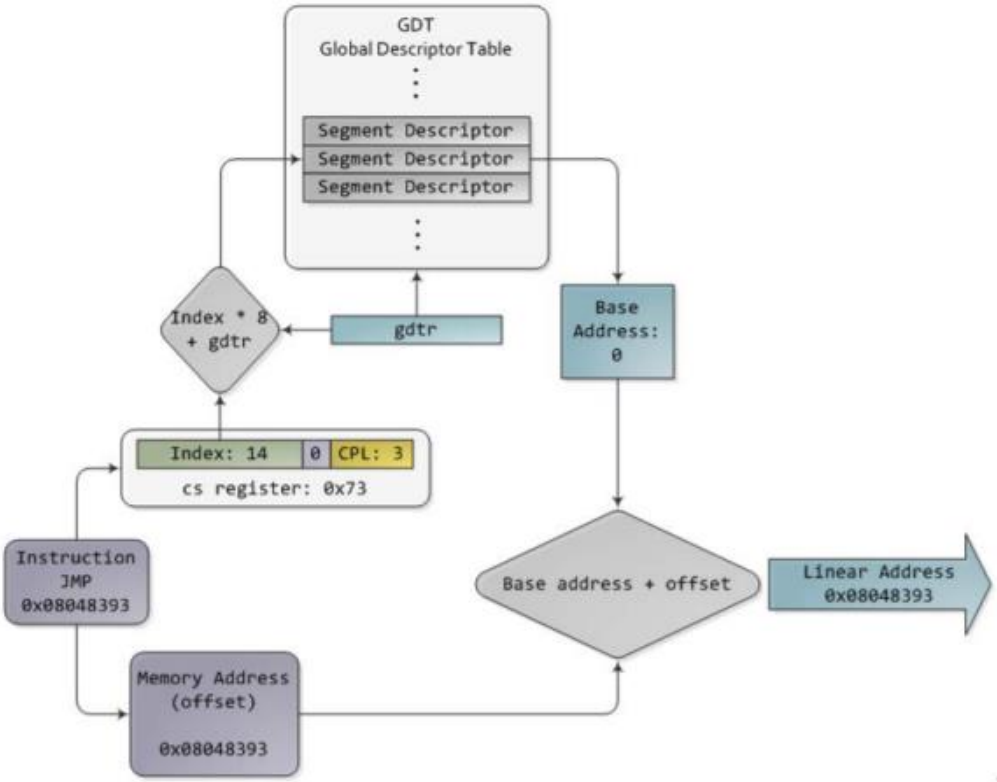
所有的段描述符被保存在两个表中：全局描述符表 GDT 和局部描述符表 LDT。gdtr 寄存器指向 GDT 表基址。

段选择符构造如下：

TI：0 为 GDT，1 为 LDT。Index 指出选择描述符表中的哪个条目，RPL 请求特权级。

在保护模式下，分段机制：段选择符在段描述符表-> Index ->目标描述符条目 Segment Descriptor->目标段的基地址 Base address+偏移量 offset=线性地址 Linear Address。

保护模式时分段机制图示如下：



当 CPU 位于 32 位模式时，内存 4GB，寄存器和指令都可以寻址整个线性地址空间，所以这时候不再需要使用基地址，将基地址设置为 0。

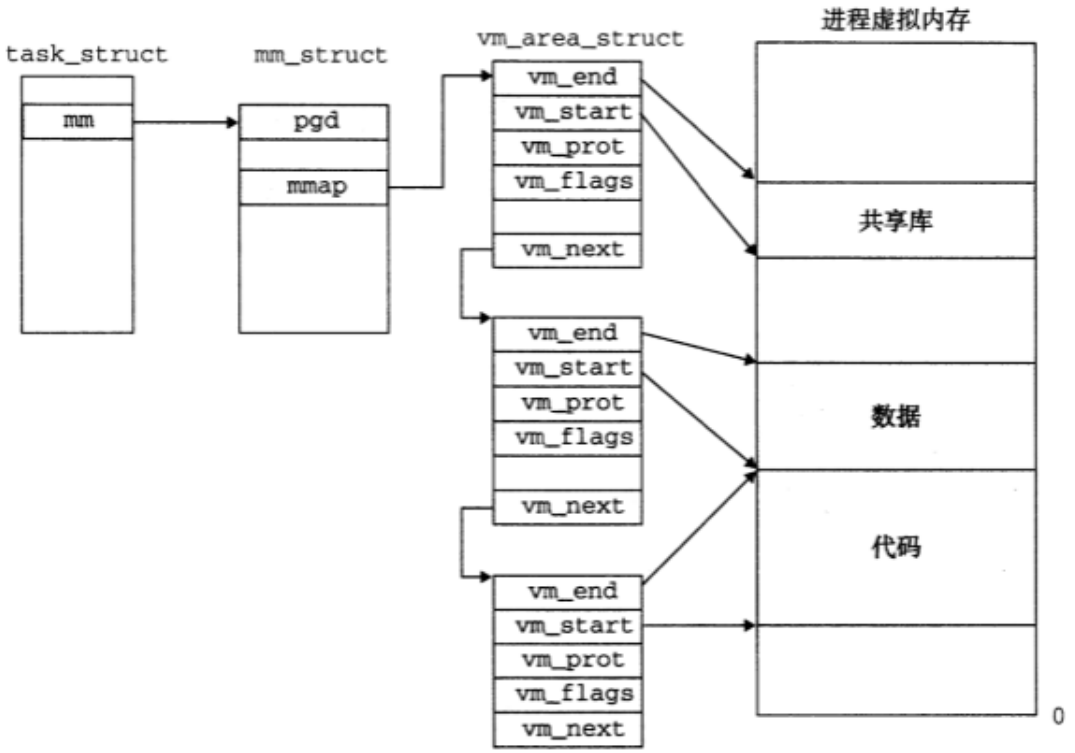
在 CPU 64 位模式中强制使用扁平的线性空间。现代的 x86 系统内核使用的是基本扁平模型，即逻辑地址=描述符=线性地址，等价于转换地址时关闭了分段功能。逻辑地址与线性地址就合二为一了。

7.3 Hello 的线性地址到物理地址的变换-页式管理

通过分页机制实现线性地址（书里的虚拟地址 VA）到物理地址（PA）之间的转换。分页机制是指对虚拟地址内存空间进行分页。

(1) 首先 Linux 系统有自己的虚拟内存系统，Linux 将虚拟内存组织成一些段的集合，段之外的虚拟内存不存在因此不需要记录。

(2) 内核为 hello 进程维护一个段的任务结构即图中的 task_struct，其中条目 mm→mm_struct(描述了虚拟内存的当前状态)，pgd→第一级页表的基地址（结合一个进程一串页表），mmap→vm_area_struct 的链表。一个链表条目对应一个段，链表相连指出了 hello 进程虚拟内存中的所有段



(3) 虚拟页（VP）：系统将每个段分割为大小固定的块，来作为进行数据传输的单元。对于 linux，每个虚拟页大小为 4KB。

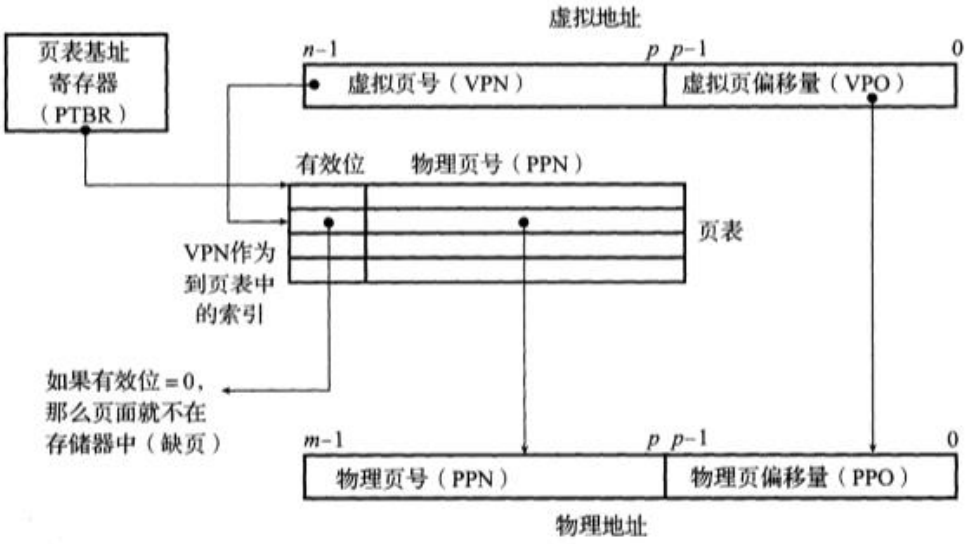
(4) 物理页（PP/页帧）：类似于虚拟页，物理内存也被分割。虚拟内存系统中 MMU 负责地址翻译，MMU 使用页表，即存一种放在物理内存中的数据结构，将虚拟页到物理页映射，即虚拟地址到物理地址的映射。

(5) 通过页表基址寄存器 PTBR+VPN 在页表中获得条目 PTE，一条 PTE 中包含有效位、权限信息、物理页号。

a) 如果有效位是 0+NULL 则代表没有在虚拟内存空间中分配该内存；

b) 如果是有效位 0+非 NULL，则代表在虚拟内存空间中分配了但是没有被缓存到物理内存中；

c) 如果有效位是 1 则代表该内存已经缓存在了物理内存中，可以得到其物理页号 PPN，与虚拟页偏移量共同构成物理地址 PA。



7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

在 Intel Core i7 环境下研究 VA 到 PA 的地址翻译问题。前提如下：

i. 虚拟地址空间 48 位，物理地址空间 52 位，页表大小 4KB，4 级页表。

ii. TLB 4 路 16 组相联。CR3 指向第一级页表的起始位置（上下文一部分）。

一个页表大小 4KB，一个 PTE 条目 8B，共 512(2e9)个条目，使用 9 位二进制索引，一共 4 个页表共使用 36 位二进制索→ VPN 共 36 位。VPO = VA 48 位- VPN36 位= 12 位； TLB 共 16 组→。因为 TLBT = VPN 36 位 - TLBI 4 位=32 位。

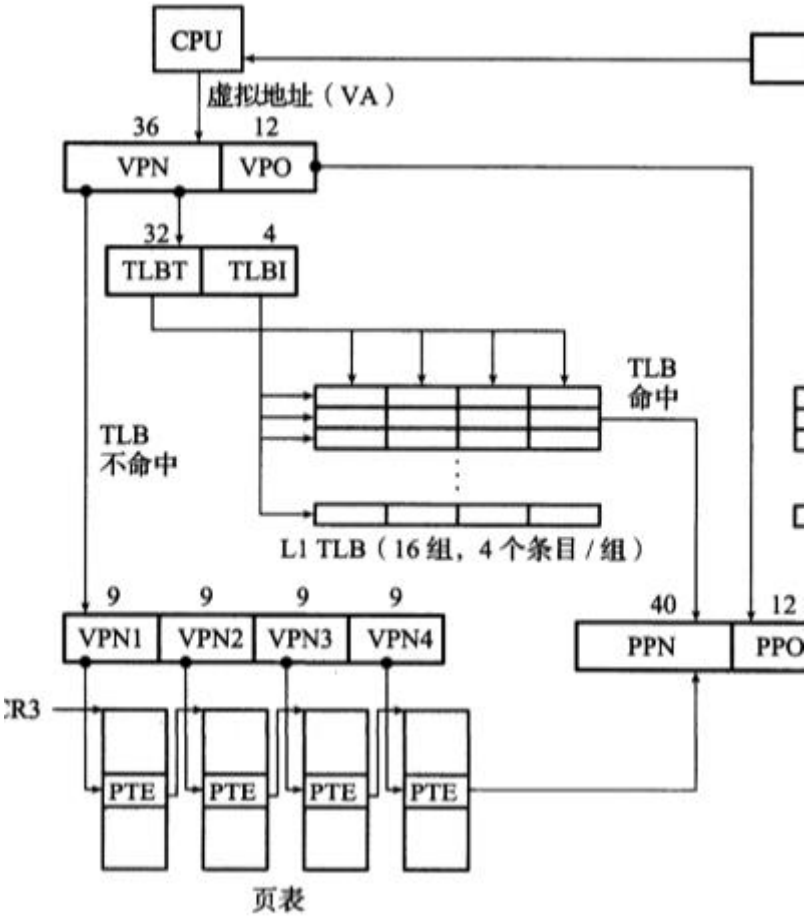
(1) CPU 产生虚拟地址 VA，VA 传送给 MMU，MMU 使用前 36 位 VPN 作为 TLBT（前 32 位）+TLBI（后 4 位）向 TLB 中匹配

(2) 如果命中，则得到 PPN（40bit）与 VPO（12bit）组合成 PA（52bit）。

(3) 如果 TLB 中没有命中，MMU 向页表中查询，CR3 确定第一级页表的起始地址，VPN1（9bit）确定在第一级页表中的偏移量，查询出 PTE，如果在物理内存中且权限符合，确定第二级页表的起始地址，

(4) 以此类推，最终在第四级页表中查询到 PPN，与 VPO 组合成 PA，并且向 TLB 中添加条目。

(5) 如果查询 PTE 的时候发现不在物理内存中，则引发缺页故障。如果发现权限不够，则引发段错误。



7.5 三级 Cache 支持下的物理内存访问

前提：只讨论 L1 Cache 的寻址细节，L2 与 L3Cache 原理相同。L1 Cache 是 8 路 64 组相联。块大小为 64B。

共 64(2e6)组→需要 6bit CI 进行组寻址。共有 8 路，块大小为 64B→需要 6bit CO 表示数据偏移位置。因为 VA 共 52bit，所以 CT 共 40bit。

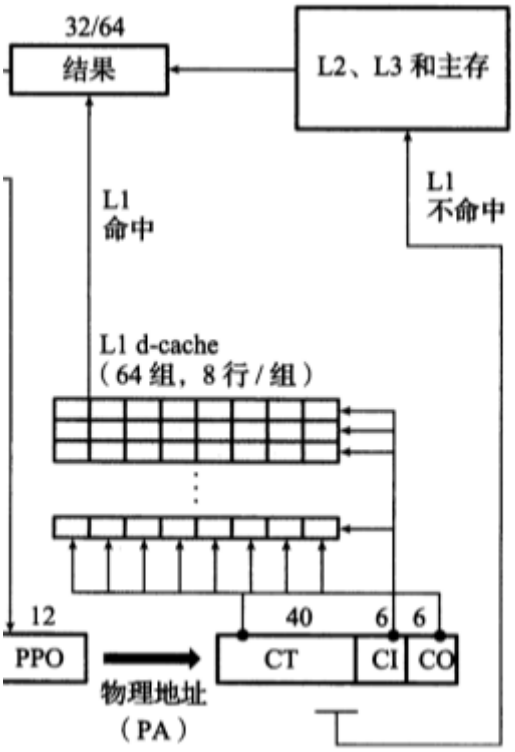
(1) 根据上一步获得的物理地址 VA，使用 CI（后六位再后六位）进行组索引，每组 8 路，对 8 路的块分别匹配 CT（前 40 位）。

(2) 如果匹配成功且块的 valid 标志位为 1，则命中（hit），根据数据偏移量 CO（后六位）取出数据返回。

(3) 如果没有匹配成功或者匹配成功但是标志位是 1，则不命中（miss），向下一级缓存中查询数据（L2 Cache->L3 Cache->主存）。

(4) 根据一种常见的简单策略，查询到数据之后，如果映射到的组内有空闲块，则直接放置；否则组内都是有效块，产生冲突（evict），则采用最近最少

使用策略 LFU 进行替换。



7.6 hello 进程 fork 时的内存映射

当 fork 函数被 shell 进程调用时，内核为新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。它将这两个进程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。

7.7 hello 进程 execve 时的内存映射

execve 函数调用驻留在内核区域的启动加载器代码，在当前进程中加载并运行包含在可执行目标文件 hello 中的程序，用 hello 程序有效地替代了当前程序加载并运行 hello 需要以下几个步骤：

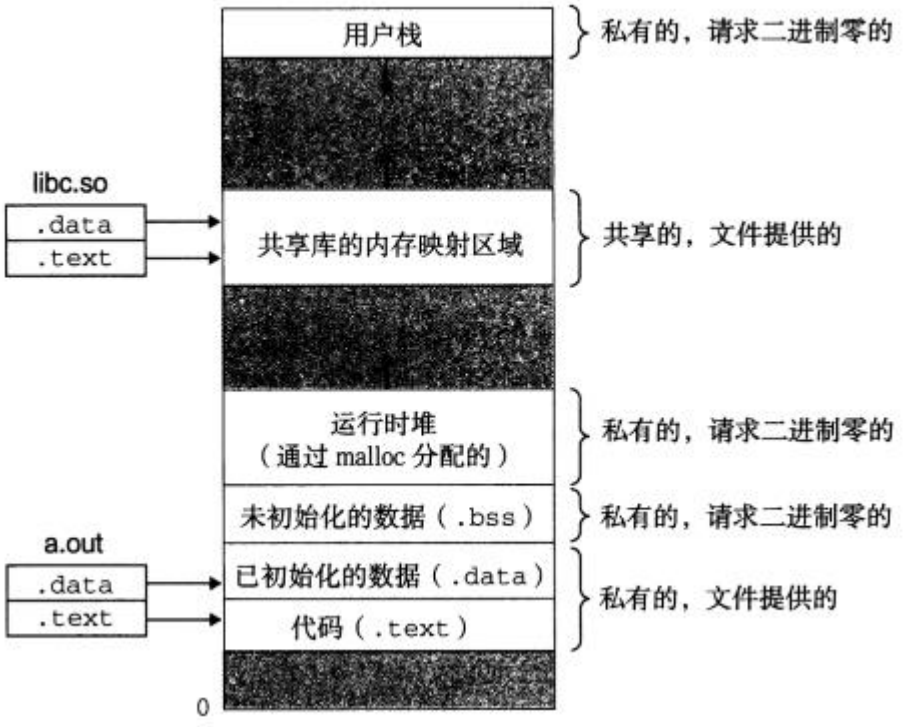
(1) 删除已存在的用户区域，删除当前进程虚拟地址的用户部分中的已存在的区域结构。

(2) 映射私有区域，为新程序的代码、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为 hello 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 中，栈和堆地址也是请求二进制零的，初始长度为零。

(3) 映射共享区域， hello 程序与共享对象 libc.so 链接，libc.so 是动态链接到这个程序中的，然后再映射到用户虚拟地址空间中的共享区域内。

(4) 设置程序计数器（PC），execve 做的最后一件事情就是设置当前进程上下文的程序计数器，使之指向代码区域的入口点。



7.8 缺页故障与缺页中断处理

缺页故障是一种常见的故障，当指令引用一个虚拟地址，在 MMU 中查找页表时发现与该地址相对应的物理地址不在内存中，因此必须从磁盘中取出的时候就会发生故障。

缺页中断处理：缺页处理程序是系统内核中的代码，选择一个牺牲页面，如果

这个牺牲页面被修改过，那么就将它交换出去，换入新的页面并更新页表。当缺页处理程序返回时，CPU 重新启动引起缺页的指令，这条指令再次发送 VA 到 MMU，这次 MMU 就能正常翻译 VA 了。

7.9 动态存储分配管理

printf 函数会调用 malloc，下面简述动态内存管理的基本方法与策略：

(1) 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。

(2) 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。

(3) 空闲块可用来分配。空闲块保持空闲，直到它显式地被应用所分配。

(4) 一个已分配的块保持已分配状态，直到它被释放。

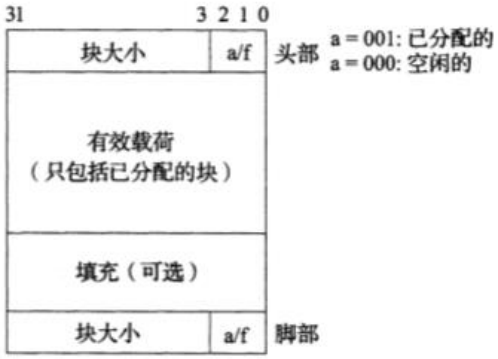
分配器分为两种基本风格：显式分配器、隐式分配器。

显式分配器：要求应用显式地释放任何已分配的块。

隐式分配器：要求分配器检测一个已分配块何时不再使用，那么就释放这个块，自动释放未使用的已经分配的块的过程叫做垃圾收集。

一、 带边界标签的隐式空闲链表

(1) 堆及堆中内存块的组织结构：



在内存块中增加 4B 的 Header（用于寻找下一个 blcok）和 4B 的 Footer（用于寻找上一个 block）。Footer 的设计是专门为了合并空闲块方便的。因为 Header 和 Footer 大小已知，所以利用 Header 和 Footer 中存放的块大小就可以寻找上下 block。

(2) 隐式链表

对比于显式空闲链表，隐式空闲链表代表并不直接对空闲块进行链接，而是将对内存空间中的所有块组织成一个大链表。隐式空闲链表中 Header 和 Footer 中的

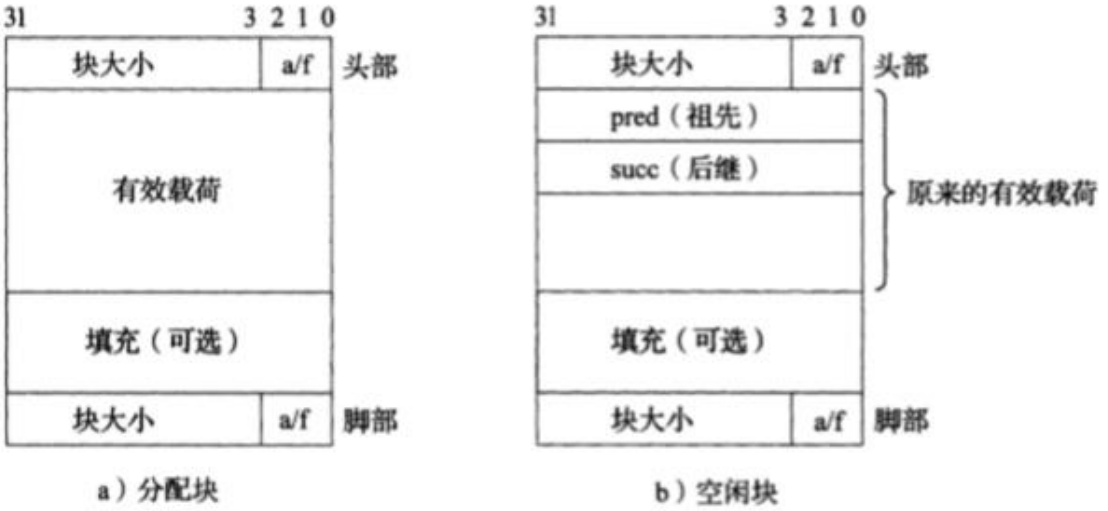
block 大小间接起到了前驱、后继指针的作用。

(3) 空闲块合并

可以利用 Footer 方便的对前面的空闲块进行合并。合并的情况一共分为四种：前空后不空，前不空后空，前后都空，前后都不空。只需要通过改变 Header 和 Footer 中的值，就可以完成对于四种情况分别进行空闲块合并。

二、 显示空间链表基本原理

将空闲块组织成链表形式的数据结构。堆可以组织成一个双向空闲链表，在每个空闲块中，都包含一个 pred（前驱）和 succ（后继）指针，如下图：



7.10 本章小结

本章主要介绍了 hello 的存储器地址空间、intel 的段式管理、hello 的页式管理， 以 intel Core7 在指定环境下介绍了 VA 到 PA 的变换、物理内存访问，还介绍 hello 进程 fork 时的内存映射、execve 时的内存映射、缺页故障与缺页中断处理、动态存储分配管理。

（第 7 章 2 分）

第 8 章 hello 的 I0 管理

8.1 Linux 的 I0 设备管理方法

设备的模型化：所有的 IO 设备都被模型化为文件，而所有的输入和输出都被当做对相应文件的读和写来执行，这种将设备优雅地映射为文件的方式，允许 Linux 内核引出一个简单低级的应用接口，称为 Unix I/O。

8.2 简述 Unix I0 接口及其函数

Unix I/O 接口统一操作：

(1) 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。描述符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。

(2) Shell 创建的每个进程都有三个打开的文件：标准输入，标准输出，标准错误。

(3) 改变当前的文件位置：内核保持着每个打开的文件的一个文件位置 k。k 初始为 0。这个文件位置 k 表示的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，显式地将改变当前文件位置 k，例如各种 fread 或 fwrite。

(4) 读写文件：

i. 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>=m 时，触发 EOF。

ii. 写操作就是从内存中复制 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=k+n。

(5) 关闭文件：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去。

Unix I/O 函数：

(1) int open(char* filename,int flags,mode_t mode) ，进程通过调用 open 函数来打开一个存在的文件或是创建一个新文件的（即 fopen 的内层函数）。open 函数将 filename（文件名，含后缀）转换为一个文件描述符（C 中表现为指针），并且返回描述符数字。返回的描述符总是在进程中当前没有打开的最小描述符，flags 参数指明了进程打算如何访问这个文件（读或写或两者兼具），mode 参数指定了新文件的访问权限位（只读等）。

(2) int close(fd)，fd 是需要关闭的文件的描述符（C 中表现为指针），close 返回

操作结果。

(3) ssize_t read(int fd,void *buf,size_t n)，read 函数从描述符为 fd 的当前文件位置赋值最多 n 个字节到内存位置 buf。返回值-1 表示一个错误，0 表示 EOF，否则返回值表示的是实际传送的字节数量。

(4) ssize_t wirte(int fd,const void *buf,size_t n)，write 函数从内存位置 buf 复制至多 n 个字节到描述符为 fd 的当前文件位置。

8.3 printf 的实现分析

前提：printf 和 vsprintf 代码是 windows 下的。

查看 printf 代码：

```
int printf(const char *fmt, ...)
{
    int i;
    char buf[256];
    va_list arg = (va_list)((char*)(&fmt) + 4);
    i = vsprintf(buf, fmt, arg);
    write(buf, i);
    return i;
}
```

首先 arg 获得第二个不定长参数，即输出的时候格式化串对应的值。

查看 vsprintf 代码：

```
int vsprintf(char *buf, const char *fmt, va_list args)
{
    char* p;
    char tmp[256];
    va_list p_next_arg = args;
    for (p = buf; *fmt; fmt++)
    {
        if (*fmt != '%') //忽略无关字符
        {
            *p++ = *fmt;
            continue;
        }
        fmt++;
        switch (*fmt)
        {
            case 'x': //只处理%x一种情况
            itoa(tmp, *((int*)p_next_arg));//将输入参数值转化为字符串保存在
tmp
            strcpy(p, tmp); //将tmp字符串复制到p处
            p_next_arg += 4; //下一个参数值地址
            p += strlen(tmp);//放下一个参数值的地址
            break;
            case 's':
            break;
            default:
```

```
            break;
        }
    }

    return (p - buf); //返回最后生成的字符串的长度
}
```

则知道 vsprintf 程序按照格式 fmt 结合参数 args 生成格式化之后的字符串，并返回字串的长度。

在 printf 中调用系统函数 write(buf,i)将长度为 i 的 buf 输出。write 函数如下：

```
write:
 mov eax, _NR_write
 mov ebx, [esp + 4]
 mov ecx, [esp + 8]
int INT_VECTOR_SYS_CALL
```

在 write 函数中，将栈中参数放入寄存器，ecx 是字符个数，ebx 存放第一个字符地址，int INT_VECTOR_SYS_CALLA 代表通过系统调用 syscall，查看 syscall 的实现：

```
sys_call:
 call save

 push dword [p_proc_ready]

 sti

 push ecx
 push ebx
 call [sys_call_table + eax * 4]
 add esp, 4 * 3

 mov [esi + EAXREG - P_STACKBASE], eax

 cli
 ret
```

syscall 将字符串中的字节“Hello 1170300821 罗瑞欣”从寄存器中通过总线复制到显卡的显存中，显存中存储的是字符的 ASCII 码。字符显示驱动子程序将通过 ASCII 码在字模库中找到点阵信息将点阵信息存储到 vram 中。显示芯片会按照一定的刷新频率逐行读取 vram，并通过信号线向液晶显示器传输每一个点（RGB 分量）。于是打印字符串“Hello 1170300821 罗瑞欣”就显示在了屏幕上。

8.4 getchar 的实现分析

异步异常-键盘中断的处理：当用户按键时，键盘接口会得到一个代表该按键

的键盘扫描码，同时产生一个中断请求，中断请求抢占当前进程运行键盘中断子程序，键盘中断子程序先从键盘接口取得该按键的扫描码，然后将该按键扫描码转换成 ASCII 码，保存到系统的键盘缓冲区之中。

getchar 函数落实到底层调用了系统函数 read，通过系统调用 read 读取存储在键盘缓冲区中的 ASCII 码直到读到回车符然后返回整个字串，getchar 进行封装，大体逻辑是读取字符串的第一个字符然后返回。

8.5 本章小结

本章主要介绍了 Linux 的 IO 设备管理方法、Unix IO 接口及其函数，分析了 printf 函数和 getchar 函数。

（第 8 章 1 分）

结论

在千分之一秒内，（最）经典程序 hello 在二进制的世界里完成了一生。程序虽小，却遍历了宏伟的计算机世界；运行虽短，但时间记录下了一切（实际上是我:）

当然，细节如下：

所谓 P2P：

(1) 编写：通过 editor 将代码键入 hello.c
(2) 预处理：将 hello.c 调用的所有外部的库展开，所有的宏定义替换，合并到一个 hello.i 文件中
(3) 编译：将 hello.i 编译成为汇编文件 hello.s
(4) 汇编：将 hello.s 会变成为可重定位目标文件 hello.o
(5) 链接：将 hello.o 与可重定位目标文件和动态链接库链接成为可执行目标程序 hello

所谓 020：

(6) 运行：在 shell（terminal）中输入./hello 1170300821 罗瑞欣
(7) 创建子进程：shell 父进程调用 fork 函数为 hello 创建子进程
(8) 运行程序：shell 调用 execve，execve 调用启动加载器，加映射虚拟内存，进入程序入口后程序，开始载入物理内存，然后 CPU 进入 main 函数执行程序。
(9) 执行指令：CPU 为 hello 分配时间片，在一个时间片中，hello 享有 CPU 资源，顺序执行自己的控制逻辑流
(10) 访问内存：MMU 将程序中使用的虚拟内存地址，通过页表映射成物理地址。
(11) 动态申请内存：printf 会调用 malloc 向动态内存分配器申请堆中的内存。
(12) 信号：如果运行途中键入 ctr-c、ctr-z，则调用 shell 的信号处理函数分别停止、挂起。
(13) 结束：shell 父进程回收子进程，内核删除为这个进程创建的所有数据结构。

（结论 0 分，缺失 -1 分，根据内容酌情加分）

附件

列出所有的中间产物的文件名，并予以说明起作用。

文件名称	文件作用
hello.i	预处理之后文本文件
hello.s	编译之后的汇编文件
hello.o	汇编之后的可重定位目标执行
hello	链接之后的可执行目标文件
hello2.c	测试程序代码
hello2	测试程序
helloo.objdmp	Hello.o 的反汇编代码
helloo.elf	Hello.o 的 ELF 格式
hello.objdmp	Hello 的反汇编代码
hello.elf	Hellode ELF 格式
hmp.txt	存放临时数据

（附件 0 分，缺失 -1 分）

参考文献

为完成本次大作业你翻阅的书籍与网站等

[1] 林来兴. 空间控制技术[M]. 北京：中国宇航出版社，1992：25-42.

[2] 辛希孟. 信息技术与信息服务国际研讨会论文集：A 集[C]. 北京：中国科学出版社，1999.

[3] 赵耀东. 新时代的工业工程师[M/OL]. 台北：天下文化出版社，1998 [1998-09-26]. http://www.ie.nthu.edu.tw/info/ie.newie.htm（Big5）.

[4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨：哈尔滨工业大学，1992：8-13.

[5] KANAMORI H. Shaking Without Quaking[J]. Science，1998，279（5359）：2063-2064.

[6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science，1998，281：331-332[1998-09-23]. http://www.sciencemag.org/cgi/collection/anatmorp.

[7] Bryant,R.E 深入理解计算机系统第三版

[8] CSDN 若干博客：

https://blog.csdn.net/wulex/article/details/78027957
https://blog.csdn.net/xuehuafeiwu123/article/details/72963229
https://blog.csdn.net/qq_32014215/article/details/76618649
https://blog.csdn.net/qq_32014215/article/details/76618649
https://www.cnblogs.com/fanzhidongyzby/p/3519838.html
https://www.cnblogs.com/fanzhidongyzby/p/3519838.html

[9] 博客园若干博客：

https://www.cnblogs.com/jiqing9006/p/8268233.html
https://www.cnblogs.com/xmphoenix/archive/2011/10/23/2221879.html
https://www.cnblogs.com/fanzhidongyzby/p/3519838.html
https://www.cnblogs.com/zengyiwen/p/5755186.html

[10] 百度百科、维基百科的若干词条

（参考文献 0 分，缺失 -1 分）