

哈爾濱工業大學

計算機系統

大作業

題 目 程序人生-Hello's P2P

專 業 計算機類

學 號 1190501614

班 級 1903006

學 生 cgh

指 導 教 師 史先俊

計算機科學與技術學院

2021 年 5 月

## 摘 要

本论文着重运用 CSAPP 所学知识分析 hello 程序执行的过程，从预处理开始到回收 hello 进程整个过程。这里我们主要在 Linux 系统来分析 hello 的变化，通过对比分析文件差异的方法来展示每一个过程的变化规律，从而得到不同阶段的处理过程，此外我们紧扣每一阶段的联系来得到 hello 的演变过程。本论文的意义在于通过一个简单的程序来展现程序运行的基本原理，站在更高的维度上认识程序。

**关键词：**计算机系统，hello，ubuntu，程序；

# 目 录

<b>第 1 章 概述</b> .....	<b>4 -</b>
1.1 HELLO 简介 .....	4 -
1.2 环境与工具 .....	4 -
1.3 中间结果 .....	4 -
1.4 本章小结 .....	4 -
<b>第 2 章 预处理</b> .....	<b>6 -</b>
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 .....	7 -
<b>第 3 章 编译</b> .....	<b>9 -</b>
3.1 编译的概念与作用 .....	9 -
3.2 在 UBUNTU 下编译的命令 .....	9 -
3.3 HELLO 的编译结果解析 .....	9 -
3.3.1 数据 .....	9 -
3.3.2 操作符 .....	10 -
3.3.3 赋值 .....	10 -
3.3.4 关系操作和控制转移 .....	10 -
3.3.5 数组操作 .....	11 -
3.3.5 函数操作 .....	11 -
3.4 本章小结 .....	12 -
<b>第 4 章 汇编</b> .....	<b>14 -</b>
4.1 汇编的概念与作用 .....	14 -
4.2 在 UBUNTU 下汇编的命令 .....	14 -
4.3 可重定位目标 ELF 格式 .....	14 -
4.4 HELLO.O 的结果解析 .....	16 -
4.5 本章小结 .....	17 -
<b>第 5 章 链接</b> .....	<b>19 -</b>
5.1 链接的概念与作用 .....	19 -
5.2 在 UBUNTU 下链接的命令 .....	19 -
5.3 可执行目标文件 HELLO 的格式 .....	19 -
5.4 HELLO 的虚拟地址空间 .....	23 -
5.5 链接的重定位过程分析 .....	24 -

5.6 HELLO 的执行流程.....	- 25 -
5.7 HELLO 的动态链接分析.....	- 26 -
5.8 本章小结.....	- 26 -
<b>第 6 章 HELLO 进程管理.....</b>	<b>- 28 -</b>
6.1 进程的概念与作用 .....	- 28 -
6.2 简述壳 SHELL-BASH 的作用与处理流程.....	- 28 -
6.3 HELLO 的 FORK 进程创建过程.....	- 28 -
6.4 HELLO 的 EXECVE 过程 .....	- 29 -
6.5 HELLO 的进程执行 .....	- 29 -
6.6 HELLO 的异常与信号处理 .....	- 30 -
6.7 本章小结 .....	- 32 -
<b>第 7 章 HELLO 的存储管理.....</b>	<b>- 34 -</b>
7.1 HELLO 的存储器地址空间 .....	- 34 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理 .....	- 34 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理.....	- 34 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换 .....	- 35 -
7.5 三级 CACHE 支持下的物理内存访问 .....	- 35 -
7.6 HELLO 进程 FORK 时的内存映射.....	- 35 -
7.7 HELLO 进程 EXECVE 时的内存映射.....	- 36 -
7.8 缺页故障与缺页中断处理.....	- 36 -
7.9 动态存储分配管理.....	- 36 -
7.10 本章小结 .....	- 38 -
<b>第 8 章 HELLO 的 IO 管理 .....</b>	<b>- 39 -</b>
8.1 LINUX 的 IO 设备管理方法 .....	- 39 -
8.2 简述 UNIX IO 接口及其函数 .....	- 39 -
8.3 PRINTF 的实现分析 .....	- 40 -
8.4 GETCHAR 的实现分析.....	- 41 -
8.5 本章小结 .....	- 41 -
<b>结论.....</b>	<b>- 42 -</b>
<b>附件.....</b>	<b>- 43 -</b>
<b>参考文献.....</b>	<b>- 44 -</b>

## 第 1 章 概述

### 1.1 Hello 简介

#### **P2P:**

**Program:** 在编辑器中编写 `hello.c` 程序

**Process:** `hello.c` 经过 `gcc` 的预处理、编译、汇编、链接最终变为可执行目标程序 `hello`。在 `shell` 中启动命令后，为其 `fork` 子进程，然后在当前进程上下文中加载运行 `hello`。

#### **O2O:**

`Shell` 开始为 `hello` 进程映射虚拟内存，执行时加载到物理内存中。运行时，`CPU` 为运行的 `hello` 分配时间片执行逻辑控制流。程序运行结束后，`shell` 回收 `hello` 进程。

### 1.2 环境与工具

软件环境: Windows10 64 位; Ubuntu 20.04

开发与调试工具: `gcc`, `ld`, `gedit`, `edb`, `readelf`

### 1.3 中间结果

<b>hello.i</b>	预处理后文件
hello.s	编译后汇编文件
hello.o	链接后的可重定位文件
helloelf.txt	hello.o 的 elf 格式文件
hello	可执行目标文件
hello_outelf.txt	hello（可执行文件）的 elf 格式文件

### 1.4 本章小结

本章主要从总体上分析 `hello`，介绍了 P2P、O2O，以及需要用到的软件环境和开发工具，最后展示了从 `.c` 文件到可执行文件中间经历的过程。

(第 1 章 0.5 分)

## 第 2 章 预处理

### 2.1 预处理的概念与作用

预处理是指在程序源代码被翻译为目标代码的过程中，在编译之前的过程。这个过程由预处理器完成对程序源代码文本中的预处理命令进行处理。通常在 C 语言程序源代码中包含一些预处理命令,这些命令包括`#include`(文件包含)、`#define`(宏定义)、`#if/#ifdef/#ifndef/#else/#elif/#endif`(条件编译)、`#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)。而预处理器则是对这些指令进行处理，最典型的的就是宏替换。

预处理的主要作用如下：

1. 将源文件中以“include”格式包含的文件拷贝一份到要编译的源文件中
2. 用实际值或含参表达式替换“#define”定义的符号。
3. 根据“#if”后面的条件决定需要编译的代码。
4. 对其它预处理命令的处理，例如`#pragma`，设定编译器的状态。

### 2.2 在 Ubuntu 下预处理的命令

`gcc -E hello.c -o hello.i`

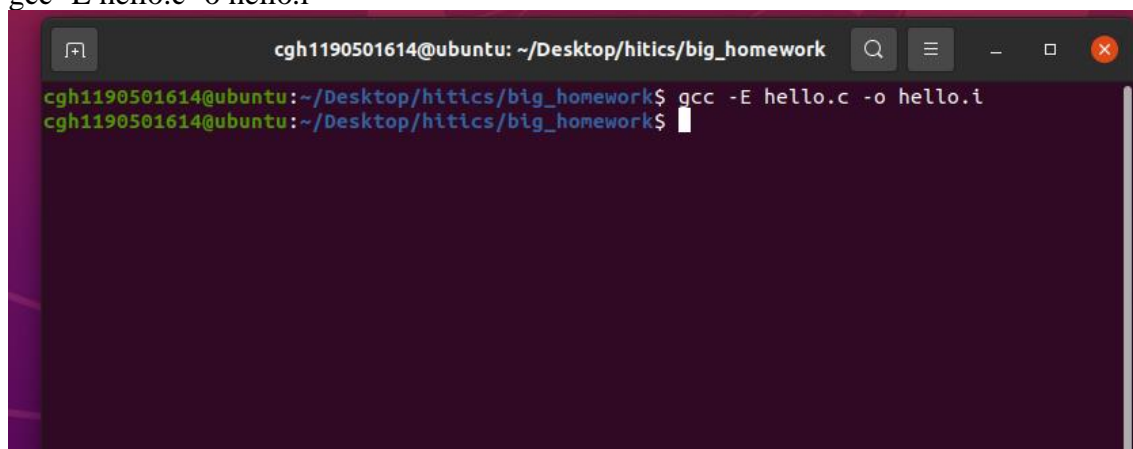


图 1 预处理

## 2.3 Hello 的预处理结果解析

发现生成的 `hello.i` 源文件多了很多代码，其中 `hello.c` 文件的几行代码（除去以 `#` 开头的命令）放在了 `hello.i` 文件的最后面，前面的三千多行代码是拷贝 `<stdio.h>`、`<unistd.h>`、`<stdlib.h>` 到源文件中，并进行相应的处理和声明，比如用于内存管理的结构体声明和一些函数和符号类型的重命名。

```

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
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
29 # 1 "/usr/include/x86_64-linux-gnu/gnu/stubs-64.h" 1 3 4
30 # 11 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 2 3 4
31 # 486 "/usr/include/features.h" 2 3 4
32 # 34 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 2 3 4
33 # 28 "/usr/include/stdio.h" 2 3 4
34
35
36
37
38
39 # 1 "/usr/lib/gcc/x86_64-linux-gnu/9/include/stddef.h" 1 3 4
40 # 209 "/usr/lib/gcc/x86_64-linux-gnu/9/include/stddef.h" 3 4
41
42 # 209 "/usr/lib/gcc/x86_64-linux-gnu/9/include/stddef.h" 3 4
3023
3024
3025
3026
3027
3028
3029 extern int rpmatch (const char *_response) __attribute__((__nothrow__, __leaf__)) __attribute__((__nonnull__(1))) ;
3030 # 957 "/usr/include/stdlib.h" 3 4
3031 extern int getsuopt (char ** restrict __optionp,
3032   char *const __restrict __tokens,
3033   char ** restrict __valuep)
3034   __attribute__((__nothrow__, __leaf__)) __attribute__((__nonnull__(1, 2, 3))) ;
3035 # 1003 "/usr/include/stdlib.h" 3 4
3036 extern int getloadavg (double __loadavg[], int __nelem)
3037   __attribute__((__nothrow__, __leaf__)) __attribute__((__nonnull__(1))) ;
3038 # 1013 "/usr/include/stdlib.h" 3 4
3039 # 1 "/usr/include/x86_64-linux-gnu/bits/stdlib-float.h" 1 3 4
3040 # 1014 "/usr/include/stdlib.h" 2 3 4
3041 # 1023 "/usr/include/stdlib.h" 3 4
3042
3043 # 9 "hello.c" 2
3044
3045
3046 # 10 "hello.c"
3047 int main(int argc, char *argv[]) {
3048   int i;
3049
3050   if(argc!=4){
3051     printf("用法: Hello 学号 姓名 秒数!\n");
3052     exit(1);
3053   }
3054   for(i=0;i<8;i++){
3055     printf("Hello %s %s\n",argv[1],argv[2]);
3056     sleep(atoi(argv[3]));
3057   }
3058   getchar();
3059   return 0;
3060 }

```

图 2 `hello.i` 文件内容

## 2.4 本章小结

本章主要介绍 C 程序预处理的概念和作用，包括一些预处理指令（宏）的工作方式。以及在 `ubuntu` 下的对 `.c` 源程序进行预处理的命令。最后了解预处理后的 `.i` 文件的基本结构和变化。



(第 2 章 0.5 分)

## 第 3 章 编译

### 3.1 编译的概念与作用

狭义的编译指的是就是把代码转化为汇编指令的过程，实际上广义的编译，其实包括预处理、编译、汇编、链接整个过程。在这里仅讨论狭义的编译，实际可以理解为将（高级语言程序）C 程序转化为汇编程序的过程。

编译的作用：

- 对于初步翻译得到的代码进行再次的扫描和翻译，去掉冗余
- 对语法和词法进行分析，检验程序的正确性和合法性
- 翻译生成符合机器的最终代码

### 3.2 在 Ubuntu 下编译的命令

```
gcc -S hello.i -o hello.s
```

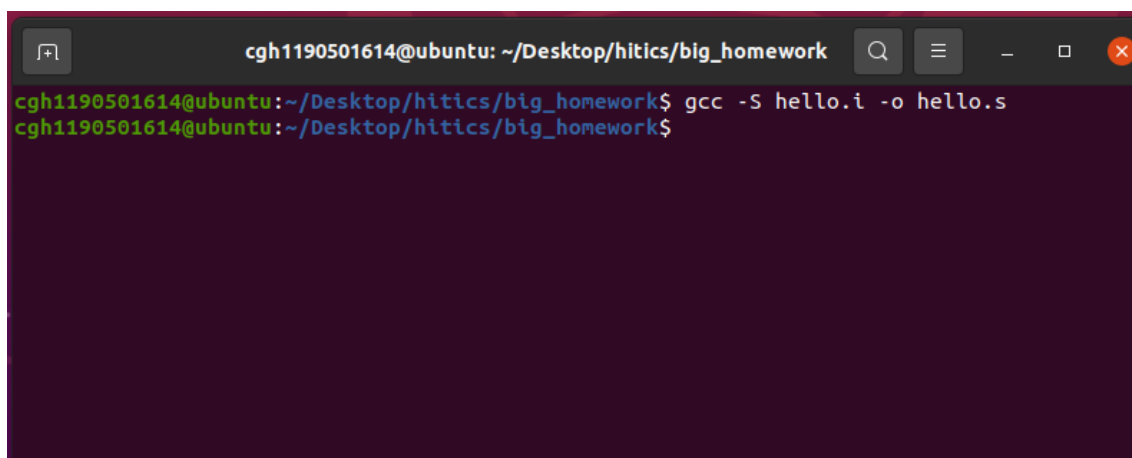
A terminal window with a dark purple background. The title bar shows the user 'cgh1190501614@ubuntu' and the directory '~/Desktop/hitcs/big\_homework'. The terminal shows the command 'gcc -S hello.i -o hello.s' being entered and executed. The prompt 'cgh1190501614@ubuntu:~/Desktop/hitcs/big\_homework\$' is visible before and after the command.

图 3 编译过程

### 3.3 Hello 的编译结果解析

#### 3.3.1 数据

##### 1. 常量

字符串常量“用法: Hello 学号 姓名 秒数!\n”和“Hello %s %s\n”，保存在.LC 中的.string（.rodata）

.LC0:

```
.string "\347\224\250\346\263\225:    Hello    \345\255\246\345\217\267
\345\247\223\345\220\215 \347\247\222\346\225\260\357\274\201"
```

.LC1:

```
.string "Hello %s %s\n"
```

常量 4、0、1、8 作为立即数放在指令里面 (.text)

## 2. 变量

局部变量 int i; 存在在栈中 (%rbp-0x4)

```
movl $0, -4(%rbp)
```

参数变量 int argc, char \*argv[]调用前在寄存器中, 之后放在栈中

```
movl %edi, -20(%rbp)
```

```
movq %rsi, -32(%rbp)
```

### 3.3.2 操作符

++操作符

for 循环中的 i++; 通过 addl 指令实现

```
addl $1, -4(%rbp)
```

### 3.3.3 赋值

for 循环中 i=0, 通过 movl 指令实现

```
movl $0, -4(%rbp)
```

### 3.3.4 关系操作和控制转移

程序中的 if(argc!=4)通过汇编指令 cmpl 和 je 共同完成

```
cmpl $4, -20(%rbp)
```

```
je .L2
```

cmpl 的作用是将 argc 和 4 比较, je 是条件跳转

而在 for(i=0;i<8;i++)中, 在 i<8 处需要判断和转移, 被翻译为如下指令

```
cmpl $7, -4(%rbp)
```

```
jle .L4
```

jle 根据前面的判断结果得到跳转码决定是否跳转

### 3.3.5 数组操作

数组操作主要体现在参数 `argv` 上，对参数 `argv[1]`、`argv[2]` 的访问和传递，主要是通过栈来进行存储和访问。

参数 `argv[1]` 和 `argv[2]`:

```
movl    %edi, -20(%rbp)
```

```
movq    %rsi, -32(%rbp)
```

然后传给调用的 `printf` 函数等函数:

```
movq    -32(%rbp), %rax
```

```
addq    $8, %rax
```

```
movq    (%rax), %rax
```

```
movq    %rax, %rsi
```

```
leaq    .LC1(%rip), %rdi
```

```
movl    $0, %eax
```

### 3.3.5 函数操作

#### 1. Main 函数

传入参数 `argc` 和 `argv[]`，分别用寄存器 `%rdi` 和 `%rsi` 存储，之后放在栈中，通过栈访问，通过系统自动调用，函数返回值为 0

main:

.LFB6:

```
.cfi_startproc
```

```
endbr64
```

```
pushq   %rbp
```

```
.cfi_def_cfa_offset 16
```

```
.cfi_offset 6, -16
```

参数传递:

```
movl    %edi, -20(%rbp)
```

```
movq    %rsi, -32(%rbp)
```

返回值

```
movl    $0, %eax
```

```
leave
```

#### 2. Printf 函数

调用 `printf` 函数通常是由 `call puts` 指令完成调用，前面需要做一些参数转递的语句。

`printf("用法: Hello 学号 姓名 秒数! \n");` 对应的汇编代码为

```
leaq    .LC0(%rip), %rdi
```

```
call    puts@PLT
```

`leaq` 指令是将字符串参数传给寄存器 `%rdi`

printf("Hello %s %s\n",argv[1],argv[2]);对应的汇编代码为

```
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
```

前面的语句是将参数 argv[1]和 argv[2]传给%rsi 和%rdx，将字符串传给%rdi，最后通过 call printf@PLT 来调用 printf 函数

### 3. exit 函数

调用 exit 函数比较简单，将参数 1 转递给%edi 然后 call exit 即可，下面是汇编代码

```
movl $1, %edi
call exit@PLT
```

调用完成后退出程序，返回值为 1

### 4. sleep 函数

由于参数 argv[3]已经加载在了%eax 中，将%eax 传给%edi 后即可开始调用 sleep 函数，汇编代码如下：

```
movl %eax, %edi
call sleep@PLT
```

### 5. atoi 函数

从栈中取出参数 argv[3]的地址传给%rax，然后将 argv[3]的值传给%rdi，调用 atoi 函数，汇编代码如下：

```
movq -32(%rbp), %rax
addq $24, %rax
movq (%rax), %rax
movq %rax, %rdi
call atoi@PLT
```

### 6. getchar 函数

该函数不需要传入参数，主要作用是清除缓存区中的字符  
下面是汇编代码：

```
call getchar@PLT
```

## 3.4 本章小结

本章主要介绍了编译的概念及其简单的工作原理，通过分析 hello.s 文件理解了

程序转化为汇编代码的过程。本章着重分析了程序中数据类型，函数调用，条件转移以及运算符操作的相关问题，通过分析从源代码到汇编代码的变化，对编译的过程有了进一步了解。

**(第 3 章 2 分)**

## 第 4 章 汇编

### 4.1 汇编的概念与作用

汇编概念：

汇编器将汇编语言（hello.s）翻译成机器语言（hello.o）的过程称为汇编，同时这个机器语言文件也是可重定位目标文件。

汇编的作用：

将 hello.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 hello.o 目标文件中，hello.o 文件是一个二进制文件，它包含程序的指令编码。

### 4.2 在 Ubuntu 下汇编的命令

```
gcc -c hello.s -o hello.o
```

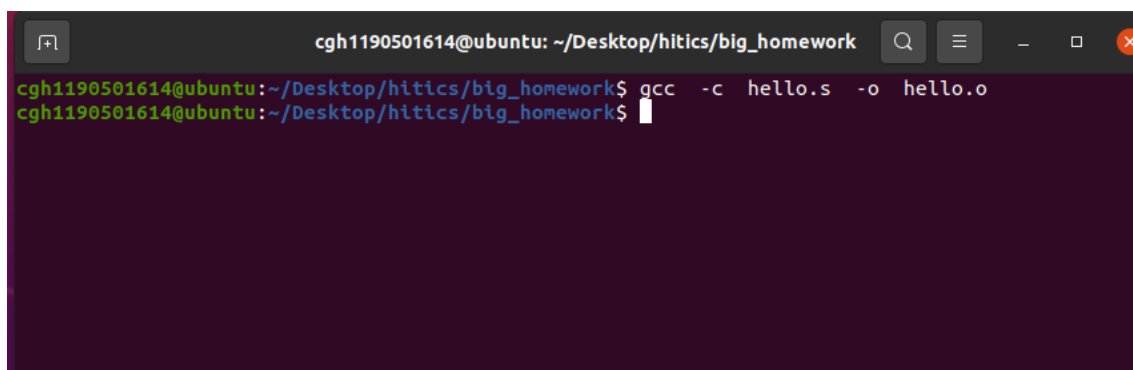


图 4 汇编过程

### 4.3 可重定位目标 elf 格式

```
readelf -a hello.o > helloelf.txt
```

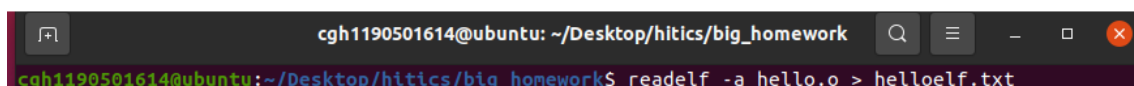
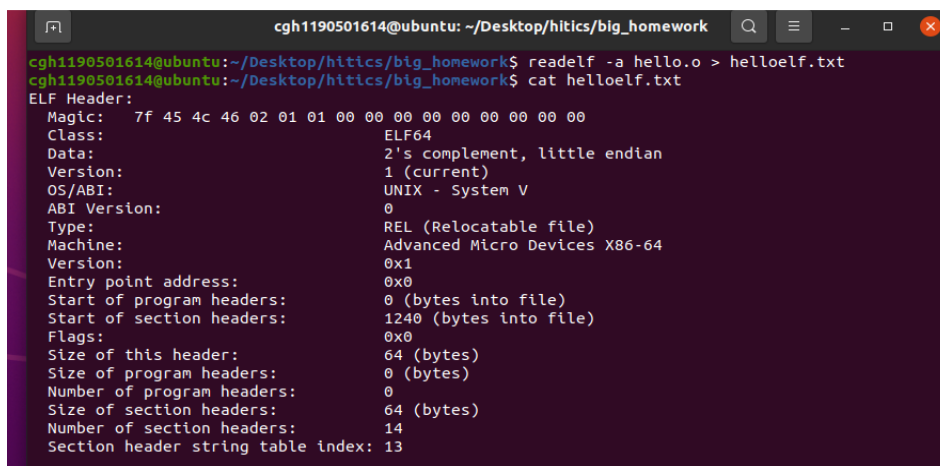


图 5 查看 elf 格式文件

## 1、ELF Header

ELF Header 中包含了系统信息（64 位），编码方式（小端法），文件类型（可重定位文件）ELF 头大小（64 bytes），节的大小和数量等一系列信息



```

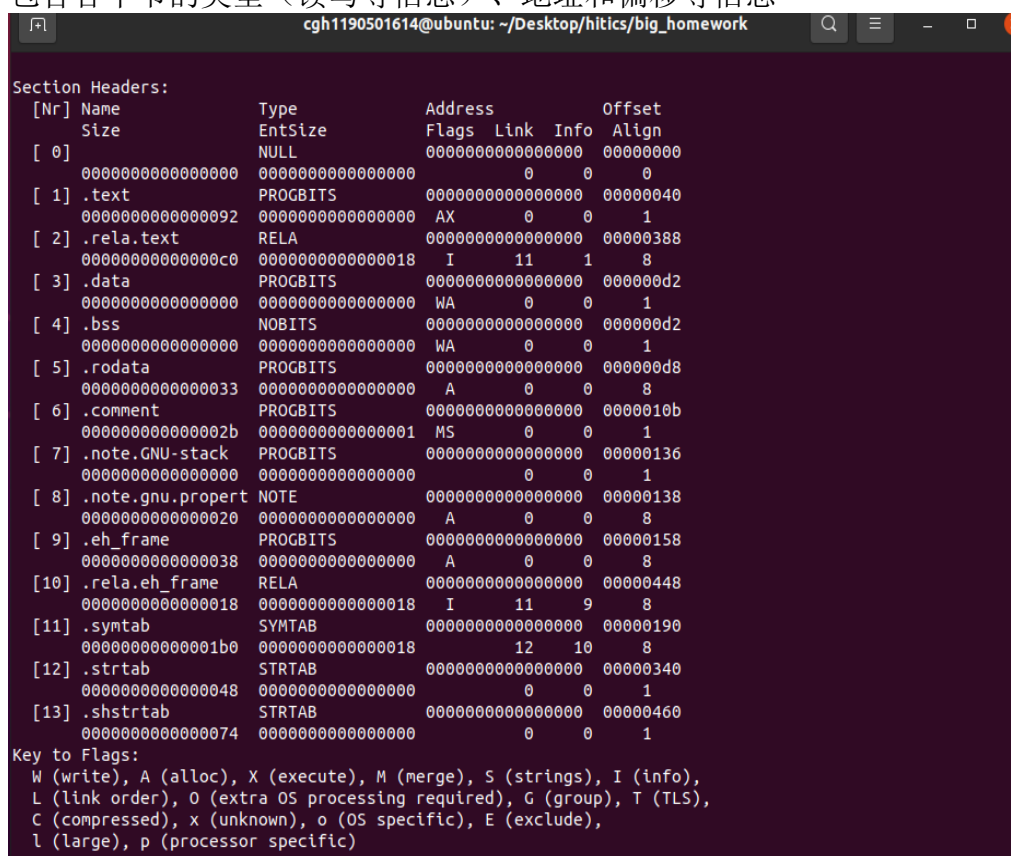
cgh1190501614@ubuntu: ~/Desktop/hitcs/big_homework
cgh1190501614@ubuntu:~/Desktop/hitcs/big_homework$ readelf -a hello.o > helloelf.txt
cgh1190501614@ubuntu:~/Desktop/hitcs/big_homework$ cat helloelf.txt
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                                ELF64
  Data:                                      2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                UNIX - System V
  ABI Version:                           0
  Type:                                  REL (Relocatable file)
  Machine:                               Advanced Micro Devices X86-64
  Version:                               0x1
  Entry point address:                   0x0
  Start of program headers:              0 (bytes into file)
  Start of section headers:             1240 (bytes into file)
  Flags:                                  0x0
  Size of this header:                   64 (bytes)
  Size of program headers:               0 (bytes)
  Number of program headers:              0
  Size of section headers:               64 (bytes)
  Number of section headers:              14
  Section header string table index:     13

```

图 6 查看 hello.o 的 elf 格式内容

## 2、Section Headers(节头目表)

包含各个节的类型（读写等信息）、地址和偏移等信息



```

cgh1190501614@ubuntu: ~/Desktop/hitcs/big_homework
cgh1190501614@ubuntu:~/Desktop/hitcs/big_homework$ readelf -S hello.o
Section Headers:
 [Nr] Name              Type              Address             Offset
     Size              EntSize          Flags Link Info Align
-----
 [ 0]                     NULL              0000000000000000    0 0 0
     0000000000000000 0000000000000000
 [ 1] .text                PROGBITS          0000000000000000 00000040
     0000000000000092 0000000000000000 AX 0 0 1
 [ 2] .rela.text           RELA              0000000000000000 00000388
     00000000000000c0 0000000000000018 I 11 1 8
 [ 3] .data                PROGBITS          0000000000000000 000000d2
     0000000000000000 0000000000000000 WA 0 0 1
 [ 4] .bss                 NOBITS            0000000000000000 000000d2
     0000000000000000 0000000000000000 WA 0 0 1
 [ 5] .rodata              PROGBITS          0000000000000000 000000d8
     0000000000000033 0000000000000000 A 0 0 8
 [ 6] .comment             PROGBITS          0000000000000000 0000010b
     000000000000002b 0000000000000001 MS 0 0 1
 [ 7] .note.GNU-stack      PROGBITS          0000000000000000 00000136
     0000000000000000 0000000000000000 0 0 1
 [ 8] .note.gnu.property  NOTE              0000000000000000 00000138
     0000000000000020 0000000000000000 A 0 0 8
 [ 9] .eh_frame            PROGBITS          0000000000000000 00000158
     0000000000000038 0000000000000000 A 0 0 8
[10] .rela.eh_frame        RELA              0000000000000000 00000448
     0000000000000018 0000000000000018 I 11 9 8
[11] .symtab               SYMTAB            0000000000000000 00000190
     00000000000001b0 0000000000000018 12 10 8
[12] .strtab               STRTAB            0000000000000000 00000340
     0000000000000048 0000000000000000 0 0 1
[13] .shstrtab             STRTAB            0000000000000000 00000460
     0000000000000074 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)

```

图 7 节头目表



### 3、Relocation section(可重定位节)

主要是各个段引用的外部符号和地址偏移，在链接时，通过重定位节对这些位置的地址根据偏移进行修改。本程序中可重定位信息包括.rodata 中字符串、puts、exit、printf、atoi、sleep 和 getchar 函数，以及.text 代码节

```
Relocation section '.rel.text' at offset 0x388 contains 8 entries:
  Offset          Info          Type           Sym. Value      Sym. Name + Addend
00000000001c 000500000002 R_X86_64_PC32 0000000000000000 .rodata - 4
000000000021 000c00000004 R_X86_64_PLT32 0000000000000000 puts - 4
00000000002b 000d00000004 R_X86_64_PLT32 0000000000000000 exit - 4
000000000054 000500000002 R_X86_64_PC32 0000000000000000 .rodata + 22
00000000005e 000e00000004 R_X86_64_PLT32 0000000000000000 printf - 4
000000000071 000f00000004 R_X86_64_PLT32 0000000000000000 atoi - 4
000000000078 001000000004 R_X86_64_PLT32 0000000000000000 sleep - 4
000000000087 001100000004 R_X86_64_PLT32 0000000000000000 getchar - 4

Relocation section '.rel.eh_frame' at offset 0x448 contains 1 entry:
  Offset          Info          Type           Sym. Value      Sym. Name + Addend
000000000020 000200000002 R_X86_64_PC32 0000000000000000 .text + 0

The decoding of unwind sections for machine type Advanced Micro Devices X86-64 is not currently supported.
```

图 8 .rel.text

### 4、.symtab (符号表)

存放在程序中定义和引用的函数和全局变量的信息。本程序中的符号信息为 hello.c、Main、全局偏移、puts、exit、printf、atoi、sleep、getchar

```
Symbol table '.symtab' contains 18 entries:
 Num:  Value          Size Type Bind Vis Ndx Name
0: 0000000000000000 0 NOTYPE LOCAL DEFAULT UND
1: 0000000000000000 0 FILE LOCAL DEFAULT ABS hello.c
2: 0000000000000000 0 SECTION LOCAL DEFAULT 1
3: 0000000000000000 0 SECTION LOCAL DEFAULT 3
4: 0000000000000000 0 SECTION LOCAL DEFAULT 4
5: 0000000000000000 0 SECTION LOCAL DEFAULT 5
6: 0000000000000000 0 SECTION LOCAL DEFAULT 7
7: 0000000000000000 0 SECTION LOCAL DEFAULT 8
8: 0000000000000000 0 SECTION LOCAL DEFAULT 9
9: 0000000000000000 0 SECTION LOCAL DEFAULT 6
10: 0000000000000000 146 FUNC GLOBAL DEFAULT 1 main
11: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND _GLOBAL_OFFSET_TABLE_
12: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND puts
13: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND exit
14: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND printf
15: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND atoi
16: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND sleep
17: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND getchar

No version information found in this file.

Displaying notes found in: .note.gnu.property
Owner      Data size  Description
GNU        0x00000010 NT_GNU_PROPERTY_TYPE_0
Properties: x86 feature: IBT, SHSTK
```

图 9 符号表

## 4.4 Hello.o 的结果解析

Hello.o 中加入了 16 进制的机器编码，hello.s 中只有指令描述；此外 hello.s 中操作数是 10 进制，而 hello.o 中操作数是 16 进制

跳转语句的差异：hello.s 中通过.L1 和.LC0 段名称来标志转移，在 hello.o 反汇编代码中通过相对偏移的地址来跳转

Hello.s 中同 call 函数名称来跳转，而反汇编代码中同相对地址来调用函数，因为函数只有在链接后才能知道绝对地址。

```

cgh1190501614@ubuntu:~/Desktop/hitcs/big_homework$ objdump -d -r hello.o

hello.o:      file format 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 04          cmpl  $0x4,-0x14(%rbp)
17:  74 16                je     2f <main+0x2f>
19:  48 8d 3d 00 00 00 00 lea    0x0(%rip),%rdi    # 20 <main+0x20>
                        1c: R_X86_64_PC32      .rodata-0x4
20:  e8 00 00 00 00      callq 25 <main+0x25>
                        21: R_X86_64_PLT32      puts-0x4
25:  bf 01 00 00 00      mov   $0x1,%edi
2a:  e8 00 00 00 00      callq 2f <main+0x2f>
                        2b: R_X86_64_PLT32      exit-0x4
2f:  c7 45 fc 00 00 00 00 movl   $0x0,-0x4(%rbp)
36:  eb 48                jmp    80 <main+0x80>
38:  48 8b 45 e0          mov   -0x20(%rbp),%rax
3c:  48 83 c0 10          add   $0x10,%rax
40:  48 8b 10             mov   (%rax),%rdx
43:  48 8b 45 e0          mov   -0x20(%rbp),%rax
47:  48 83 c0 08          add   $0x8,%rax
4b:  48 8b 00             mov   (%rax),%rax
4e:  48 89 c6             mov   %rax,%rsi
51:  48 8d 3d 00 00 00 00 lea    0x0(%rip),%rdi    # 58 <main+0x58>
                        54: R_X86_64_PC32      .rodata+0x22

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:  48 8b 45 e0          mov   -0x20(%rbp),%rax
66:  48 83 c0 18          add   $0x18,%rax
6a:  48 8b 00             mov   (%rax),%rax
6d:  48 89 c7             mov   %rax,%rdi
70:  e8 00 00 00 00      callq 75 <main+0x75>
                        71: R_X86_64_PLT32      atoi-0x4
75:  89 c7               mov   %eax,%edi
77:  e8 00 00 00 00      callq 7c <main+0x7c>
                        78: R_X86_64_PLT32      sleep-0x4
7c:  83 45 fc 01          addl   $0x1,-0x4(%rbp)
80:  83 7d fc 07          cmpl  $0x7,-0x4(%rbp)
84:  7e b2                jle    38 <main+0x38>
86:  e8 00 00 00 00      callq 8b <main+0x8b>
                        87: R_X86_64_PLT32      getchar-0x4
8b:  b8 00 00 00 00      mov   $0x0,%eax
90:  c9                   leaveq
91:  c3                   retq

```

图 10 hello.o 的反汇编内容

## 4.5 本章小结

本章对汇编的结果进行了基本介绍，分析了可重定位目标文件的 elf 格式中

各个部分的信息，得到可重定位的信息，尤其是地址偏移信息，为之后的链接做准备。此外还分析了 `hello.o` 的反汇编代码和 `hello.s` 中代码的区别，对于理解反汇编代码和机器代码的区别有了进一步了解。

**(第 4 章 1 分)**

## 第 5 章 链接

### 5.1 链接的概念与作用

链接指的是将各种不同文件（.so 和.o）的代码和数据部分（符号解析和重定位）起来并组合成一个可执行文件的过程。

链接的作用是可以程序设计更加模块化，可以将程序写成比较小的源文件的集合，构建公共库函数。此外链接可以将编译分离，提高编程效率

### 5.2 在 Ubuntu 下链接的命令

```
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
```

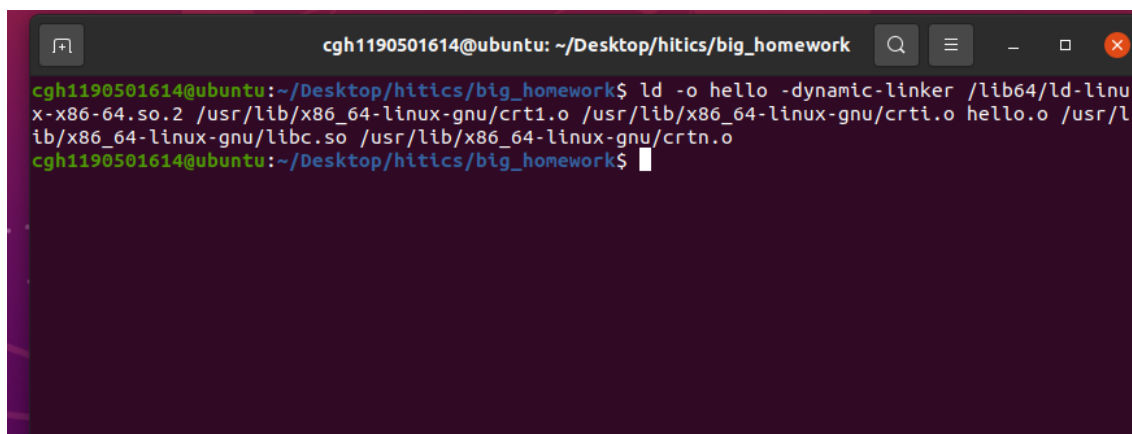


图 11 链接过程

### 5.3 可执行目标文件 hello 的格式

```
readelf -a hello > hello_out.elf
```

- ELF Header

描述一些系统信息，存储方式（小端法），以及节头的大小，数量等

```

cgh1190501614@ubuntu: ~/Desktop/hitics/big_homework
cgh1190501614@ubuntu:~/Desktop/hitics/big_homework$ readelf -a hello > hello_out.elf
cgh1190501614@ubuntu:~/Desktop/hitics/big_homework$ cat hello_out.elf
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                   ELF64
  Data:                     2's complement, little endian
  Version:                   1 (current)
  OS/ABI:                    UNIX - System V
  ABI Version:               0
  Type:                      EXEC (Executable file)
  Machine:                   Advanced Micro Devices X86-64
  Version:                   0x1
  Entry point address:       0x4010f0
  Start of program headers:  64 (bytes into file)
  Start of section headers: 14208 (bytes into file)
  Flags:                      0x0
  Size of this header:       64 (bytes)
  Size of program headers:   56 (bytes)
  Number of program headers: 12
  Size of section headers:   64 (bytes)
  Number of section headers: 27
  Section header string table index: 26

```

图 12 查看 elf 格式的 hello.out 内容

### ● 各个段的基本信息

Section Headers:						
[Nr]	Name	Type	Address	Offset		
	Size	EntSize	Flags Link Info Align			
[ 0]	0000000000000000	NULL	0000000000000000	00000000		
[ 1]	.interp	PROGBITS	00000000004002e0	000002e0		
[ 2]	.note.gnu.property	NOTE	0000000000400300	00000300		
[ 3]	.note.ABI-tag	NOTE	0000000000400320	00000320		
[ 4]	.hash	HASH	0000000000400340	00000340		
[ 5]	.gnu.hash	GNU_HASH	0000000000400378	00000378		
[ 6]	.dynsym	DYNSYM	0000000000400398	00000398		
[ 7]	.dynstr	STRTAB	0000000000400470	00000470		
[ 8]	.gnu.version	VERSYM	00000000004004cc	000004cc		
[ 9]	.gnu.version_r	VERNEED	00000000004004e0	000004e0		
[10]	.rela.dyn	RELA	0000000000400500	00000500		
[11]	.rela.plt	RELA	0000000000400530	00000530		
[12]	.init	PROGBITS	0000000000401000	00001000		
[13]	.plt	PROGBITS	0000000000401020	00001020		
[14]	.plt.sec	PROGBITS	0000000000401090	00001090		

图 13 各节基本信息

[15]	.text	PROGBITS	00000000004010f0	000010f0
	0000000000000145	0000000000000000	AX 0 0	16
[16]	.fini	PROGBITS	0000000000401238	00001238
	000000000000000d	0000000000000000	AX 0 0	4
[17]	.rodata	PROGBITS	0000000000402000	00002000
	000000000000003b	0000000000000000	A 0 0	8
[18]	.eh_frame	PROGBITS	0000000000402040	00002040
	00000000000000fc	0000000000000000	A 0 0	8
[19]	.dynamic	DYNAMIC	0000000000403e50	00002e50
	00000000000001a0	0000000000000010	WA 7 0	8
[20]	.got	PROGBITS	0000000000403ff0	00002ff0
	0000000000000010	0000000000000008	WA 0 0	8
[21]	.got.plt	PROGBITS	0000000000404000	00003000
	0000000000000048	0000000000000008	WA 0 0	8
[22]	.data	PROGBITS	0000000000404048	00003048
	0000000000000004	0000000000000000	WA 0 0	1
[23]	.comment	PROGBITS	0000000000000000	0000304c
	000000000000002a	0000000000000001	MS 0 0	1
[24]	.syntab	SYMTAB	0000000000000000	00003078
	000000000000004c8	0000000000000018	25 30	8
[25]	.strtab	STRTAB	0000000000000000	00003540
	0000000000000158	0000000000000000	0 0	1
[26]	.shstrtab	STRTAB	0000000000000000	00003698
	00000000000000e1	0000000000000000	0 0	1

续-图 13

- Program header（程序头）

描述程序中段信息，比如 load 和 DYNAMIC 等

cgh1190501614@ubuntu: ~/Desktop/hitcs/big_homework					
Program Headers:					
Type	Offset	VirtAddr	PhysAddr		
	FileSiz	MemSiz	Flags	Align	
PHDR	0x0000000000000040	0x0000000000400040	0x0000000000400040		
	0x00000000000002a0	0x00000000000002a0	R	0x8	
INTERP	0x00000000000002e0	0x00000000004002e0	0x00000000004002e0		
	0x000000000000001c	0x000000000000001c	R	0x1	
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]					
LOAD	0x0000000000000000	0x0000000000400000	0x0000000000400000		
	0x00000000000005c0	0x00000000000005c0	R	0x1000	
LOAD	0x0000000000000100	0x0000000000401000	0x0000000000401000		
	0x0000000000000245	0x0000000000000245	R E	0x1000	
LOAD	0x0000000000000200	0x0000000000402000	0x0000000000402000		
	0x000000000000013c	0x000000000000013c	R	0x1000	
LOAD	0x00000000000002e50	0x0000000000403e50	0x0000000000403e50		
	0x00000000000001fc	0x00000000000001fc	RW	0x1000	
DYNAMIC	0x00000000000002e50	0x0000000000403e50	0x0000000000403e50		
	0x00000000000001a0	0x00000000000001a0	RW	0x8	
NOTE	0x0000000000000300	0x0000000000400300	0x0000000000400300		
	0x0000000000000020	0x0000000000000020	R	0x8	
NOTE	0x0000000000000320	0x0000000000400320	0x0000000000400320		
	0x0000000000000020	0x0000000000000020	R	0x4	
GNU_PROPERTY	0x0000000000000300	0x0000000000400300	0x0000000000400300		
	0x0000000000000020	0x0000000000000020	R	0x8	
GNU_STACK	0x0000000000000000	0x0000000000000000	0x0000000000000000		
	0x0000000000000000	0x0000000000000000	RW	0x10	
GNU_RELRO	0x00000000000002e50	0x0000000000403e50	0x0000000000403e50		
	0x00000000000001b0	0x00000000000001b0	R	0x1	

图 14 程序头

- Dynamic section

提供动态链接的信息（地址、大小），主要是调用一些共享库中的函数



```

Dynamic section at offset 0x2e50 contains 21 entries:
  Tag          Type          Name/Value
  0x0000000000000001 (NEEDED)      Shared library: [libc.so.6]
  0x000000000000000c (INIT)        0x401000
  0x000000000000000d (FINI)        0x401238
  0x0000000000000004 (HASH)        0x400340
  0x000000006ffffef5 (GNU_HASH)      0x400378
  0x0000000000000005 (STRTAB)      0x400470
  0x0000000000000006 (SYMTAB)      0x400398
  0x000000000000000a (STRSZ)        92 (bytes)
  0x000000000000000b (SYMENT)      24 (bytes)
  0x0000000000000015 (DEBUG)        0x0
  0x0000000000000003 (PLTGOT)      0x404000
  0x0000000000000002 (PLTRELSZ)     144 (bytes)
  0x0000000000000014 (PLTREL)        RELA
  0x0000000000000017 (JMPREL)      0x400530
  0x0000000000000007 (RELA)        0x400500
  0x0000000000000008 (RELASZ)      48 (bytes)
  0x0000000000000009 (RELAENT)     24 (bytes)
  0x000000006ffffffe (VERNEED)     0x4004e0
  0x000000006fffffff (VERNEEDNUM)   1
  0x000000006ffffff0 (VERSYM)      0x4004cc
  0x0000000000000000 (NULL)        0x0

Symbol table '.dynsym' contains 9 entries:
  Num:  Value          Size Type Bind Vis Ndx Name
  0: 0000000000000000 0 NOTYPE LOCAL DEFAULT UND
  1: 0000000000000000 0 FUNC GLOBAL DEFAULT UND puts@GLIBC_2.2.5 (2)
  2: 0000000000000000 0 FUNC GLOBAL DEFAULT UND printf@GLIBC_2.2.5 (2)
  3: 0000000000000000 0 FUNC GLOBAL DEFAULT UND __libc_start_main@GLIBC_2.2.5 (2)
  4: 0000000000000000 0 FUNC GLOBAL DEFAULT UND getchar@GLIBC_2.2.5 (2)
  5: 0000000000000000 0 NOTYPE WEAK DEFAULT UND __gmon_start__
  6: 0000000000000000 0 FUNC GLOBAL DEFAULT UND atoi@GLIBC_2.2.5 (2)
  7: 0000000000000000 0 FUNC GLOBAL DEFAULT UND exit@GLIBC_2.2.5 (2)
  8: 0000000000000000 0 FUNC GLOBAL DEFAULT UND sleep@GLIBC_2.2.5 (2)

```

图 15 动态链接节

- 符号表  
用于符号解析

```

23: 0000000000000000 0 SECTION LOCAL DEFAULT 23
24: 0000000000000000 0 FILE LOCAL DEFAULT ABS hello.c
25: 0000000000000000 0 FILE LOCAL DEFAULT ABS
26: 0000000000403e50 0 NOTYPE LOCAL DEFAULT 19 __init_array_end
27: 0000000000403e50 0 OBJECT LOCAL DEFAULT 19 __DYNAMIC
28: 0000000000403e50 0 NOTYPE LOCAL DEFAULT 19 __init_array_start
29: 0000000000404000 0 OBJECT LOCAL DEFAULT 21 __GLOBAL_OFFSET_TABLE__
30: 0000000000401230 5 FUNC GLOBAL DEFAULT 15 __libc_csu_fini
31: 0000000000404048 0 NOTYPE WEAK DEFAULT 22 data_start
32: 0000000000000000 0 FUNC GLOBAL DEFAULT UND puts@@GLIBC_2.2.5
33: 000000000040404c 0 NOTYPE GLOBAL DEFAULT 22 _edata
34: 0000000000401238 0 FUNC GLOBAL HIDDEN 16 _fini
35: 0000000000000000 0 FUNC GLOBAL DEFAULT UND printf@@GLIBC_2.2.5
36: 0000000000000000 0 FUNC GLOBAL DEFAULT UND __libc_start_main@@GLIBC_
37: 0000000000404048 0 NOTYPE GLOBAL DEFAULT 22 __data_start
38: 0000000000000000 0 FUNC GLOBAL DEFAULT UND getchar@@GLIBC_2.2.5
39: 0000000000000000 0 NOTYPE WEAK DEFAULT UND __gmon_start__
40: 0000000000402000 4 OBJECT GLOBAL DEFAULT 17 _IO_stdin_used
41: 00000000004011c0 101 FUNC GLOBAL DEFAULT 15 __libc_csu_init
42: 0000000000404050 0 NOTYPE GLOBAL DEFAULT 22 _end
43: 0000000000401120 5 FUNC GLOBAL HIDDEN 15 _dl_relocate_static_pie
44: 00000000004010f0 47 FUNC GLOBAL DEFAULT 15 _start
45: 000000000040404c 0 NOTYPE GLOBAL DEFAULT 22 __bss_start
46: 0000000000401125 146 FUNC GLOBAL DEFAULT 15 main
47: 0000000000000000 0 FUNC GLOBAL DEFAULT UND atoi@@GLIBC_2.2.5
48: 0000000000000000 0 FUNC GLOBAL DEFAULT UND exit@@GLIBC_2.2.5
49: 0000000000000000 0 FUNC GLOBAL DEFAULT UND sleep@@GLIBC_2.2.5
50: 0000000000401000 0 FUNC GLOBAL HIDDEN 12 _init

```

图 16 符号表

- 其它节（一些关于 GNU 系统的信息）

## 5.4 hello 的虚拟地址空间

Data Dump															
0x0000000000401000-0x0000000000402000															
00000000:00401000	f3 0f 1e fa 48 83 ec 08 48 8b 05 e9 2f 00 00 48	...	H.↓.H.+/...H												
00000000:00401010	85 c0 74 02 ff d0 48 83 c4 08 c3 00 00 00 00	...t...H. f.↓....													
00000000:00401020	ff 35 e2 2f 00 00 f2 ff 25 e3 2f 00 00 0f 1f 00	05-/. □%-/....													
00000000:00401030	f3 0f 1e fa 68 00 00 00 00 f2 e9 e1 ff ff ff 90	...h.... +□□□.													
00000000:00401040	f3 0f 1e fa 68 01 00 00 00 f2 e9 d1 ff ff ff 90	...h.... +□□□.													
00000000:00401050	f3 0f 1e fa 68 02 00 00 00 f2 e9 c1 ff ff ff 90	...h.... +□□□.													
00000000:00401060	f3 0f 1e fa 68 03 00 00 00 f2 e9 b1 ff ff ff 90	...h.... +□□□.													
00000000:00401070	f3 0f 1e fa 68 04 00 00 00 f2 e9 a1 ff ff ff 90	...h.... +□□□.													
00000000:00401080	f3 0f 1e fa 68 05 00 00 00 f2 e9 91 ff ff ff 90	...h.... +□□□.													
00000000:00401090	f3 0f 1e fa f2 ff 25 7d 2f 00 00 0f 1f 44 00 00	... □%}/...D.													
00000000:004010a0	f3 0f 1e fa f2 ff 25 75 2f 00 00 0f 1f 44 00 00	... □%u/...D.													
00000000:004010b0	f3 0f 1e fa f2 ff 25 6d 2f 00 00 0f 1f 44 00 00	... □%m/...D.													
00000000:004010c0	f3 0f 1e fa f2 ff 25 65 2f 00 00 0f 1f 44 00 00	... □%e/...D.													
00000000:004010d0	f3 0f 1e fa f2 ff 25 5d 2f 00 00 0f 1f 44 00 00	... □%]/...D.													
00000000:004010e0	f3 0f 1e fa f2 ff 25 55 2f 00 00 0f 1f 44 00 00	... □%U/...D.													
00000000:004010f0	f3 0f 1e fa 31 ed 49 89 d1 5e 48 89 e2 48 83 e4	... 1.I.□^H.-H.↓													
00000000:00401100	f0 50 54 49 c7 c0 30 12 40 00 48 c7 c1 c0 11 40	PTI□□@.H□□.@													
00000000:00401110	00 48 c7 c7 25 11 40 00 1e ff 15 d2 2e 00 00 f4 90	..H□%..@.□.↓													
00000000:00401120	f3 0f 1e fa c3 f3 0f 1e fa 55 48 89 e5 48 83 ec	... H. ↓ UH. H. ↓													
00000000:00401130	20 89 7d ec 48 89 75 e0 83 7d ec 04 74 16 48 8d	... ↓H. u. d. ↓. t. H.													
00000000:00401140	3d c3 0e 00 00 e8 46 ff ff ff bf 01 00 00 e8	=H...  F□□□□...													
00000000:00401150	7c ff ff ff c7 45 fc 00 00 00 eb 48 48 8b 45	□□□ H.E...+HH.E													
00000000:00401160	0e 48 83 c0 10 48 8b 10 48 8b 45 e0 48 83 c0 08	qH.□.H..H.E qH.□													
00000000:00401170	48 8b 00 48 89 c6 48 8d 3d b1 0e 00 00 b8 00 00	H..H..H.=c...x..													
00000000:00401180	00 00 e8 19 ff ff ff 48 8b 45 e0 48 83 c0 18 48	... .□□□H.E qH.□.H.													
00000000:00401190	8b 00 48 89 c7 e8 26 ff ff ff 89 c7 e8 3f ff ff	...H..H &□□□.H ?□□													
00000000:004011a0	ff 83 45 fc 01 83 7d fc 07 7e b2 e8 00 ff ff ff	□.E. . . . ~w  .□□□													
00000000:004011b0	b8 00 00 00 c9 c3 66 0f 1f 84 00 00 00 00 00	x...□ Hf.....													
00000000:004011c0	f3 0f 1e fa 41 57 4c 8d 3d 83 2c 00 00 41 56 49	... AWL=.,...AVI													
00000000:004011d0	89 d6 41 55 49 89 f5 41 54 41 89 fc 55 48 8d 2d	...AVI. ATA. UH.-													
00000000:004011e0	6c 2c 00 00 53 4c 29 fd 48 83 ec 08 e8 0f fe ff	l...SL)0H.↓. .□□													
00000000:004011f0	ff 48 c1 fd 03 74 1f 31 db 0f 1f 80 00 00 00 00	□□0.t.1.....													
00000000:00401200	4c 89 f2 4c 89 ee 44 89 e7 41 ff 14 df 48 83 c3	L. L. d. □AQ. □H. H													
00000000:00401210	01 48 39 dd 75 ea 48 83 c4 08 5b 5d 41 5c 41 5d	..H9□u†H. f. [JA\A]													
00000000:00401220	41 5e 41 5f c3 66 66 2e 0f 1f 84 00 00 00 00	A^A_ Hff.....													

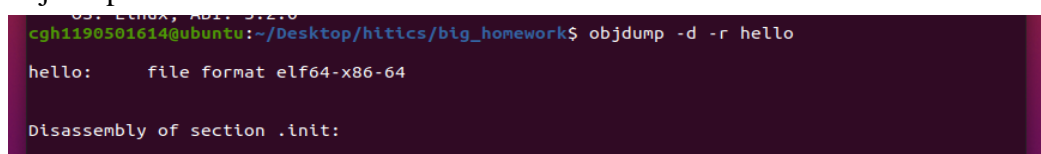
其中 PHDR 保存 program header。INTERP 指定在程序已经从可执行文件映



射到内存之后，调用的解释器。**LOAD** 表示一个需要从二进制文件映射到虚拟地址空间的段。其中包括代码和数据等。**DYNAMIC** 保存了由动态链接器使用的信息（调用 C 标准库函数采用动态链接）。**NOTE** 保存辅助信息。**GNU\_STACK** 是权限标志，用于标志栈是否是可执行。**GNU\_RELRO**：指定在重定位结束之后标志只读内存区域。

## 5.5 链接的重定位过程分析

objdump -d -r hello



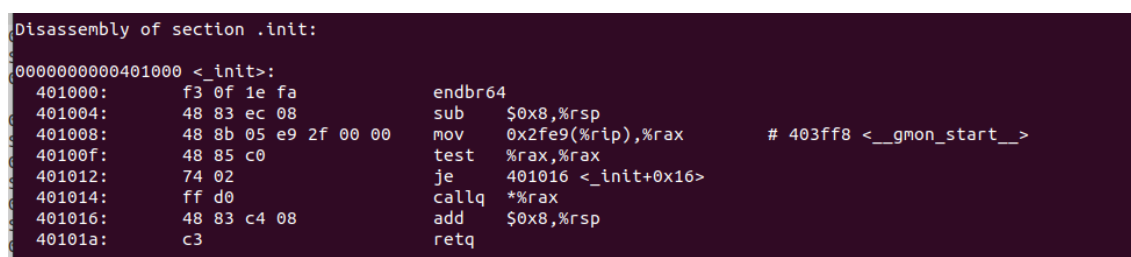
```

cgh1190501614@ubuntu:~/Desktop/hitcs/big_homework$ objdump -d -r hello
hello:      file format elf64-x86-64

Disassembly of section .init:

```

图 19 hello 反汇编



```

Disassembly of section .init:
0000000000401000 <.init>:
401000:  f3 0f 1e fa      endbr64
401004:  48 83 ec 08      sub    $0x8,%rsp
401008:  48 8b 05 e9 2f 00 00 mov    0x2fe9(%rip),%rax        # 403ff8 <__gmon_start__>
40100f:  48 85 c0          test   %rax,%rax
401012:  74 02            je     401016 <_init+0x16>
401014:  ff d0            callq  *%rax
401016:  48 83 c4 08      add    $0x8,%rsp
40101a:  c3              retq

```

图 20 .init 节

Hello 和 hello.o 的不同：

1. 链接时增加了库函数

链接后加入了在 hello.c 中用到的库函数，如 printf、exit、sleep、getchar 等函数。

2. 函数调用

在 hello 中，函数调用变成了虚拟内存地址；在 hello.o 中，函数调用的地址信息通过可重定位条目记录

3. 增加了一些头文件中的节

比如 .init 节、.plt 以及一些内置函数的节

4. 地址重定位

在 hello.o 中，函数以及全局变量的地址是相对地址，可通过可重定位表记录；而在 hello 中，函数和全局变量经过链接后地址变成虚拟内存地址。

**链接的过程:**

链接就是链接器（ld）将各个目标文件（各种.o 文件）组装在一起，经过符号解析和重定位生成一个新的可执行文件。

**5.6 hello 的执行流程**

```
<_init>
<.plt>
<puts@plt>
<printf@plt>
<getchar@plt>
<atoi@plt>
<exit@plt>
<sleep@plt>
<_start>
<_dl_relocate_static_pie>
<main>
<__libc_csu_init>
<__libc_csu_fini>
<_fini>
```

00007f2d:afd03108	49 89 c4	mov r12, rax
00007f2d:afd0310b	8b 05 e7 c4 02 00	mov eax, [rel 0x7f2dafd2f5f8]
00007f2d:afd03111	5a	pop rdx
00007f2d:afd03112	48 8d 24 c4	lea rsp, [rsp+rax*8]
00007f2d:afd03116	29 c2	sub edx, eax
00007f2d:afd03118	52	push rdx
00007f2d:afd03119	48 89 d6	mov rsi, rdx
00007f2d:afd0311c	49 89 e5	mov r13, rsp
00007f2d:afd0311f	48 83 e4 f0	and rsp, 0xfffffffffffffff0
00007f2d:afd03123	48 8b 3d 36 cf 02 00	mov rdi, [rel 0x7f2dafd30060]
00007f2d:afd0312a	49 8d 4c d5 10	lea rcx, [r13+rdx*8+0x10]
00007f2d:afd0312f	49 8d 55 08	lea rdx, [r13+8]
00007f2d:afd03133	31 ed	xor ebp, ebp
00007f2d:afd03135	e8 d6 0a 01 00	call 0x7f2dafd13c10
00007f2d:afd0313a	48 8d 15 0f 0c 01 00	lea rdx, [rel 0x7f2dafd13d50]
00007f2d:afd03141	4c 89 ec	mov rsp, r13
00007f2d:afd03144	41 ff e4	jmp r12
00007f2d:afd03147	66 0f 1f 84 00 00 00 0...	nop word [rax+rax]
00007f2d:afd03150	f3	db 0xf3
00007f2d:afd03151	0f	db 0xf
00007f2d:afd03152	1e	db 0x1e
00007f2d:afd03153	fa	clic
00007f2d:afd03154	83 47 04 01	add dword [rdi+4], 1
00007f2d:afd03158	c3	ret
00007f2d:afd03159	0f 1f 80 00 00 00 00 0...	nop dword [rax]
00007f2d:afd03160	f3	db 0xf3
00007f2d:afd03161	0f	db 0xf
00007f2d:afd03162	1e	db 0x1e
00007f2d:afd03163	fa	clic

Dump		Stack	
0x00000000000401000-0x00000000000402000			
0000:00401000	f3 0f 1e fa 48 83 ec 08 48 b5 e9 2f 00 00 48	00007ffc:483bcbf0	00000000000000001
0000:00401010	85 c0 74 02 ff 00 48 83 c4 08 c3 00 00 00 00	00007ffc:483bcbf8	00007ffc483be2a2
0000:00401020	ff 35 e2 2f 00 00 f2 ff 25 e3 2f 00 00 0f 1f 00	00007ffc:483bcc00	00000000000000000
0000:00401030	f3 0f 1e fa 68 00 00 00 00 f2 e9 e1 ff ff ff 90	00007ffc:483bcc08	00007ffc483be2a8
0000:00401040	f3 0f 1e fa 68 01 00 00 00 f2 e9 d1 ff ff ff 90	00007ffc:483bcc10	00007ffc483be2b8
0000:00401050	f3 0f 1e fa 68 02 00 00 00 f2 e9 c1 ff ff ff 90	00007ffc:483bcc18	00007ffc483be30a
0000:00401060	f3 0f 1e fa 68 03 00 00 00 f2 e9 b1 ff ff ff 90	00007ffc:483bcc20	00007ffc483be31d
0000:00401070	f3 0f 1e fa 68 04 00 00 00 f2 e9 a1 ff ff ff 90	00007ffc:483bcc28	00007ffc483be331
0000:00401080	f3 0f 1e fa 68 05 00 00 00 f2 e9 91 ff ff ff 90	00007ffc:483bcc30	00007ffc483be35e
0000:00401090	f3 0f 1e fa f2 ff 25 7d 2f 00 00 0f 1f 44 00 00	00007ffc:483bcc38	00007ffc483be375
0000:004010a0	f3 0f 1e fa f2 ff 25 75 2f 00 00 0f 1f 44 00 00	00007ffc:483bcc40	00007ffc483be3a1
0000:004010b0	f3 0f 1e fa f2 ff 25 6d 2f 00 00 0f 1f 44 00 00	00007ffc:483bcc48	00007ffc483be3c7
		00007ffc:483bcc50	00007ffc483be3d7
		00007ffc:483bcc58	00007ffc483be3db
		00007ffc:483bcc60	00007ffc483be3f2

图 21 hello 执行流程函数调用（片段）

## 5.7 Hello 的动态链接分析

在 elf 文件找到 .got 和 .got.plt 节

i2	00000000000000fc	0000000000000000	A	0	0	8
i3	[19] .dynamic	DYNAMIC	0000000000403e50	00002e50		
i4	00000000000001a0	0000000000000010	WA	7	0	8
i5	[20] .got	PROGBITS	0000000000403ff0	00002ff0		
i6	0000000000000010	0000000000000008	WA	0	0	8
i7	[21] .got.plt	PROGBITS	0000000000404000	00003000		
i8	0000000000000048	0000000000000008	WA	0	0	8
i9	[22] .data	PROGBITS	0000000000404048	00003048		
i0	0000000000000004	0000000000000000	WA	0	0	1

图 22 .got 和 .got.plt 节

执行 .init 之前

Data Dump			
0x0000000000401000-0x0000000000402000		0x0000000000403000-0x0000000000405000	
00000000:00403ff0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00000000:00404000	50 3e 40 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00404000	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00000000:00404010	00 00 00 00 00 00 00 00 30 10 40 00 00 00 00 00
00000000:00404020	40 10 40 00 00 00 00 00 50 10 40 00 00 00 00 00	00000000:00404030	60 10 40 00 00 00 00 00 70 10 40 00 00 00 00 00
00000000:00404040	80 10 40 00 00 00 00 00 00 00 00 00 47 43 43 3a	00000000:00404050	20 28 55 62 75 6e 74 75 20 39 2e 33 2e 30 2d 31
00000000:00404060	37 75 62 75 6e 74 75 31 7e 32 30 2e 30 34 29 20		

图 23 .init 之前的内存

执行 .init 之后

Data Dump			
0x0000000000401000-0x0000000000402000		0x0000000000403000-0x0000000000405000	
00000000:00403fe0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00000000:00404000	c0 5f df 56 7b 7f 00 00 00 00 00 00 00 00 00 00
00000000:00403ff0	50 3e 40 00 00 00 00 00 90 a1 00 57 7b 7f 00 00	00000000:00404010	b0 3b ff 56 7b 7f 00 00 30 10 40 00 00 00 00 00
00000000:00404000	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00000000:00404020	40 10 40 00 00 00 00 00 50 10 40 00 00 00 00 00
00000000:00404030	60 10 40 00 00 00 00 00 70 10 40 00 00 00 00 00	00000000:00404040	80 10 40 00 00 00 00 00 00 00 00 00 47 43 43 3a
00000000:00404050	20 28 55 62 75 6e 74 75 20 39 2e 33 2e 30 2d 31		

图 24 .init 之后的内存

我们利用代码段和数据段的相对位置不变的原则计算变量的正确地址。而对于库函数，需要 .plt、.got、.plt 初始存的是代码，它们跳转到 got 所指示的位置，然后调用链接器。初始时 got 里面存的都是 plt 的第二条指令，随后链接器修改 got，下一次再调用 plt 时，指向的就是正确的内存地址。plt 就能跳转到正确的区域。

## 5.8 本章小结

本章主要介绍在 linux 中链接的过程。通过 edb 查看 hello 的虚拟地址空间，结合 elf 文件分析，并且对比 hello 与 hello.o 的反汇编代码，更好地理解链接过程中的变化和基本的工作机制。

**(第 5 章 1 分)**

## 第 6 章 hello 进程管理

### 6.1 进程的概念与作用

**进程的概念：**程序的一次执行过程。

**进程的作用：**

1. 应用程序也能够创建新进程，并且在新进程的上下文中运行它们自己的代码或其他应用程序。使得能够实现多进程并发，提高系统和程序的工作效率
2. 进程提供给应用程序的关键抽象：一个独立的逻辑控制流，一个私有的地址空间，如同独占一个处理器

### 6.2 简述壳 Shell-bash 的作用与处理流程

Shell 是一个交互型应用级程序，是命令行解释器，以用户态方式运行的终端进程

其基本功能是解释并运行用户的指令，重复如下处理过程：

- (1)终端进程读取用户由键盘输入的命令行。
- (2)分析命令行字符串，获取命令行参数，并构造传递给 `execve` 的 `argv` 向量
- (3)检查第一个(首个、第 0 个)命令行参数是否是一个内置的 shell 命令
- (4)如果不是内部命令，调用 `fork()` 创建新进程/子进程
- (5)在子进程中，用步骤 2 获取的参数，调用 `execve()` 执行指定程序。
- (6)如果用户没要求后台运行(命令末尾没有 `&` 号) 否则 shell 使用 `waitpid()` (或 `wait...` 等待作业终止后返回。
- (7)如果用户要求后台运行(如果命令末尾有 `&` 号)，则 shell 返回；

### 6.3 Hello 的 fork 进程创建过程

根据 shell 的处理流程，可以推断，输入命令执行 hello 后，父进程如果判断不是内置命令，即会通过 `fork` 函数创建子进程。子进程与父进程近似，并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。父进程打开的文件，子进程也可读写。二者之间最大的不同或许在于 PID 的不同。Fork 函数只会被调用一次，但会返回两次，在父进程中，`fork` 返回子进程的 PID，在子进程中，`fork` 返回 0。

## 6.4 Hello 的 execve 过程

创建子进程后，`execve` 函数在加载并运行 **Hello**，且带列表 `argv` 和环境变量列表 `envp`。当出现错误时，`execve` 才会返回到调用程序。

结合虚拟内存和内存映射过程分析 `execve` 函数加载和执行程序 **Hello** 的过程：

- 删除已存在的用户区域（自父进程独立）。
- 映射私有区域：为 **Hello** 的代码、数据、`.bss` 和栈区域创建新的私有区域结构，写时才复制的。
- 映射共享区域：**Hello** 程序与共享对象链接时，比如 **Hello** 程序中需要用到标准 C 库中的函数，需要和 `libc.so` 链接，这些对象都是动态链接到 **Hello** 的，然后再用户虚拟地址空间中的共享区域内。
- 设置 PC：设置当前进程的上下文中的程序计数器，使之指向代码区域的入口。

## 6.5 Hello 的进程执行

程序进行上下文切换过程是通过逻辑控制流来完成的。逻辑控制流实际是一系列程序计数器 `PC` 的值的序列，当一个程序的逻辑控制流和另一个逻辑流有重叠时，可以实现两个流的并发。

私有地址空间：进程为每个流都提供私有地址空间，这个地址空间不能被其它进程访问。

用户模式和内核模式：处理器使用一个寄存器提供两种模式的区分。用户模式的进程不允许执行特殊指令，不允许直接引用地址空间中内核区的代码和数据；内核模式进程可以执行指令集中的任何命令，并且可以访问系统中的任何内存位置。

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。

基本切换过程：

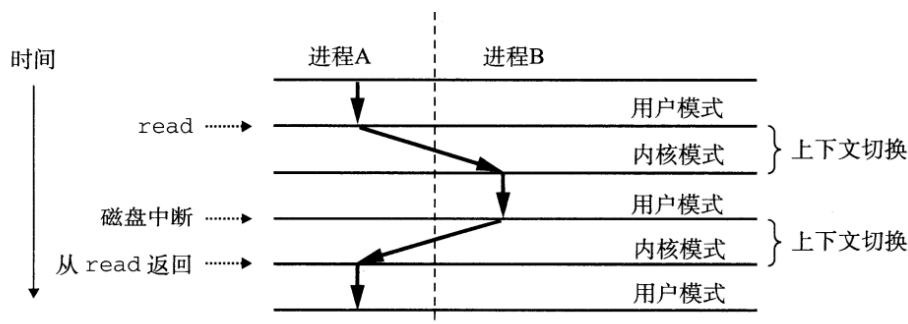


图 25 上下文切换

通过以上内容分析 hello 进程执行,首先 shell 为 hello 创建子进程,调用 `execve` 函数后,已经为 hello 程序分配了新的虚拟的私有地址空间。最初 hello 运行在用户模式下,输出信息内容,然后 hello 调用 `sleep` 函数之后进程进入内核模式,通过上下文切换处理下一个等待队列中的进程,并将 hello 进程从运行队列中移出加入等待队列,定时器开始计时,当定时器到时发送一个中断信号,此时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前台运行,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

正常运行:

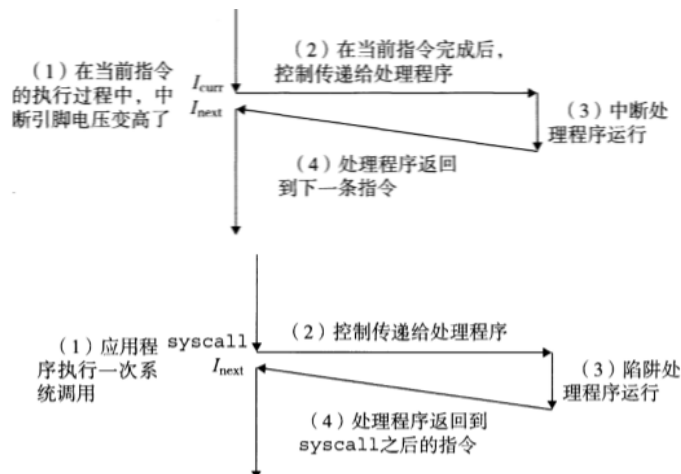
```
cgh1190501614@ubuntu: ~/Desktop/hitcs/big_homework
cgh1190501614@ubuntu:~/Desktop/hitcs/big_homework$ ./hello 1190501614 cgh 1
Hello 1190501614 cgh
Hello 1190501614 cgh
Hello 1190501614 cgh
Hello 1190501614 cgh
Hello 1190501614 cgh
Hello 1190501614 cgh
```

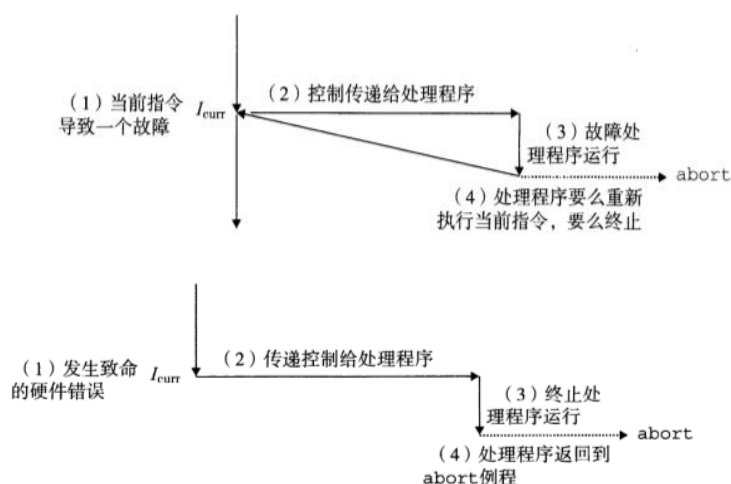
图 26 hello 正常运行结果

异常类型:

类别	原因	异步 / 同步	返回行为
中断	来自 I/O 设备的信号	异步	总是返回到下一条指令
陷阱	有意的异常	同步	总是返回到下一条指令
故障	潜在可恢复的错误	同步	可能返回到当前指令
终止	不可恢复的错误	同步	不会返回

处理方式:





结合 hello 分析：

### 1. Ctrl+Z

进程收到 SIGSTP 信号，hello 进程挂起，hello 进程的 PID 为 2344 后台序号为 1。

```

cgh1190501614@ubuntu:~/Desktop/hitcs/big_homework$ ./hello 1190501614 cgh 1
Hello 1190501614 cgh
Hello 1190501614 cgh
Hello 1190501614 cgh
Hello 1190501614 cgh
Hello 1190501614 cgh
Hello 1190501614 cgh
^Z
[1]+  Stopped                  ./hello 1190501614 cgh 1
cgh1190501614@ubuntu:~/Desktop/hitcs/big_homework$ ps
  PID TTY          TIME CMD
 2268 pts/0        00:00:00 bash
 2344 pts/0        00:00:00 hello
 2961 pts/0        00:00:00 ps
cgh1190501614@ubuntu:~/Desktop/hitcs/big_homework$ jobs
[1]+  Stopped                  ./hello 1190501614 cgh 1
cgh1190501614@ubuntu:~/Desktop/hitcs/big_homework$ fg 1
./hello 1190501614 cgh 1
Hello 1190501614 cgh
Hello 1190501614 cgh

```

图 27 按下 ctrl-Z 后

### 2. Ctrl + C

进程受到 SIGINT 信号，结束 hello 进程，后台中也查不到 hello 的 PID

```

cgh1190501614@ubuntu:~/Desktop/hitcs/big_homework$ ./hello 1190501614 cgh 1
Hello 1190501614 cgh
Hello 1190501614 cgh
Hello 1190501614 cgh
Hello 1190501614 cgh
Hello 1190501614 cgh
Hello 1190501614 cgh
Hello 1190501614 cgh
Hello 1190501614 cgh
^C
cgh1190501614@ubuntu:~/Desktop/hitcs/big_homework$ ps
  PID TTY          TIME CMD
 2268 pts/0        00:00:00 bash
 3334 pts/0        00:00:00 ps
cgh1190501614@ubuntu:~/Desktop/hitcs/big_homework$

```



图 28 按下 ctrl-C

## 3. 乱按

和 hello 输出的内容一起输出到屏幕，按下回车后，有可能被误认为是命令。

```
cgh1190501614@ubuntu:~/Desktop/hitics/big_homework$ ./hello 1190501614 cgh 1
Hello 1190501614 cgh
Hello 1190501614 cgh
Hello 1190501614 cgh
Hello 1190501614 cgh
dfvdsHello 1190501614 cgh
fdfvHello 1190501614 cgh
ddfvdHello 1190501614 cgh

Hello 1190501614 cgh
df
cgh1190501614@ubuntu:~/Desktop/hitics/big_homework$ df
Filesystem      1K-blocks    Used Available Use% Mounted on
udev             973140         0    973140    0% /dev
tmpfs            200540       1588    198952    1% /run
/dev/sda5        19992176  11557208   7396376   61% /
tmpfs            1002684         0    1002684    0% /dev/shm
tmpfs             5120          4      5116    1% /run/lock
tmpfs            1002684         0    1002684    0% /sys/fs/cgroup
/dev/loop1       66432        66432         0 100% /snap/gtk-common-themes/1514
```

图 29 乱按后结果

## 4. Kill

挂起的进程被终止，在 ps 中无法查找到其 PID

```
cgh1190501614@ubuntu:~/Desktop/hitics/big_homework$ ./hello 1190501614 cgh 1
Hello 1190501614 cgh
Hello 1190501614 cgh
Hello 1190501614 cgh
Hello 1190501614 cgh
Hello 1190501614 cgh
^Z
[1]+  Stopped                  ./hello 1190501614 cgh 1
cgh1190501614@ubuntu:~/Desktop/hitics/big_homework$ ps
  PID TTY          TIME CMD
 2268 pts/0    00:00:00 bash
 3844 pts/0    00:00:00 hello
 3851 pts/0    00:00:00 ps
cgh1190501614@ubuntu:~/Desktop/hitics/big_homework$ kill -9 3844
cgh1190501614@ubuntu:~/Desktop/hitics/big_homework$ ps
  PID TTY          TIME CMD
 2268 pts/0    00:00:00 bash
 3867 pts/0    00:00:00 ps
[1]+  Killed                  ./hello 1190501614 cgh 1
cgh1190501614@ubuntu:~/Desktop/hitics/big_homework$
```

图 30 使用 kill

## 6.7 本章小结

本章主要介绍了 hello 进程的执行过程。主要讲 hello 的创建、加载和异常处理。进程是程序运行的抽象。在 hello 运行过程中，内核有选择对其进行管理，决

定何时进行上下文切换。当接受到不同的异常信号时，异常处理程序将对异常信号做出相应的响应。对此，我们更进一步认识了 `hello` 执行过程中产生信号和信号的处理过程。

**(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

**逻辑地址：**指由程序 hello 产生的与段相关的偏移地址部分（hello.o）。

**线性地址：**是逻辑地址到物理地址变换之间的中间层。程序 hello 的代码会产生逻辑地址，它加上相应段的基地址就生成了一个线性地址。

**虚拟地址：**虚拟地址于逻辑地址类似，是对内存管理的一种抽象，程序访问存储器所使用的逻辑地址称为虚拟地址。

**物理地址：**是指在 CPU 地址总线上的寻址物理内存的地址，是访存时的最终地址。一般认为内存是一个很大的字节数组，而物理地址与内存一一对应。此外物理地址由 MMU 得到。

### 7.2 Intel 逻辑地址到线性地址的变换-段式管理

一个逻辑地址由段标识符，段内偏移量组成。段标识符是一个 16 位长的字段组成，称为段选择符，其中前 13 位是一个索引号。后面三位表示一些硬件信息。

索引号，可以通过段标识符的前 13 位，直接在段描述符表中找到一个具体的段描述符，这个描述符就描述了一个段。这里面，我们只关心 Base 字段，它描述了一个段的开始位置的线性地址。

全局的段描述符，放在“全局段描述符表(GDT)”中，一些局部的段描述符，放在“局部段描述符表(LDT)”中。

GDT 在内存中的地址和大小存放在 CPU 的 gdtr 控制寄存器中，而 LDT 则在 ldtr 寄存器中。

给定一个完整的逻辑地址段选择符+段内偏移地址，

看段选择符的 T1=0 还是 1, 知道当前要转换是 GDT 中的段, 还是 LDT 中的段, 再根据相应寄存器, 得到其地址和大小。我们就有了一个数组了。

拿出段选择符中前 13 位, 可以在这个数组中, 查找到对应的段描述符, 这样, 它了 Base, 即基地址就知道了。

把 Base + offset, 就是要转换的线性地址了

### 7.3 Hello 的线性地址到物理地址的变换-页式管理

页式管理是一种内存空间存储管理的技术，页式管理分为静态页式管理和动态页式管理。将各进程的虚拟空间划分成若干个长度相等的页(page)，页式管理把内存空间按页的大小划分成片或者页面（page frame），然后把页式虚拟地址与内

存地址建立一一对应页表，并用相应的硬件地址变换机构，来解决离散地址变换问题。页式管理采用请求调页或预调页技术实现了内外存存储器的统一管理。

由于它不要求作业或进程的程序段和数据在内存中连续存放，从而有效地解决了碎片问题。动态页式管理提供了内存和外存统一管理的虚存实现方式，使用户可以利用的存储空间大大增加。这有利于提高了主存的利用率和组织多道程序执行。

然后这种机制要求有相应的硬件支持。例如地址变换机构，缺页中断的产生和选择淘汰页面等都要求有相应的硬件支持。这会增加了机器成本。增加了系统开销。

## 7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

每次 CPU 产生一个虚拟地址，MMU 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。如果 PTE 在缓存中命中，那么开销会比较低，否则惩罚代价会比较大。为此，在 MMU 中包括了一个关于 PTE 的小的缓存（TLB）来解决这个问题。

多级页表：将虚拟地址的 VPN 划分为相等大小的不同的部分，每个部分用于寻找由上一级确定的页表基址对应的页表条目。解析 VA，利用前 m 位 vpn1 寻找一级页表位置，接着一次重复 k 次，在第 k 级页表获得了页表条目，将 PPN 与 VPO 组合获得 PA

## 7.5 三级 Cache 支持下的物理内存访问

CPU 发送一条虚拟地址，随后 MMU 按照上述操作获得了物理地址 PA。根据 cache 大小组数的要求，将物理地址分为 CT（标记位）CS(组号)，CO（偏移量）。通过 CS 寻找到正确的组，然后比较每一个 cacheline 的标记位有效以及 CT 是否相等。如果命中就直接返回想要的数，如果不命中，就依次去 L2,L3,主存中判断是否命中，当命中时，将数据传给 CPU 同时更新各级 cache 的 cacheline（如果 cache 已满则要采用换入换出策略）。

## 7.6 hello 进程 fork 时的内存映射

当 fork 函数被当前进程调用时，内核为新进程创建各种数据结构（主要是结构体），并分配给它一个唯一的 PID，同时为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位只读，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

在 `bash` 中的进程通过 `execve` 调用指向 `hello` 程序；然后加载并运行 `hello`

- 删除已存在的用户区域。
- 映射私有区域
- 映射共享区域
- 设置程序计数器（PC）

最后设置当前进程的上下文中的程序计数器，是指指向代码区域的入口点。而下一次调度这个进程时，他将从这个入口点开始执行。`Linux` 将根据需要换入代码和数据页面。

## 7.8 缺页故障与缺页中断处理

页面命中由硬件完成的，而缺页处理子程序需要由硬件和操作系统内核协作完成的。

首先 `CPU` 生成一个虚拟地址，并将它传送给 `MMU`。然后 `MMU` 生成 `PTE` 地址，并从高速缓存/主存请求得到它。然后高速缓存/主存向 `MMU` 返回 `PTE`。因为 `PTE` 中的有效位是 0，所以 `MMU` 发出了一次异常，传递 `CPU` 中的控制到操作系统内核中的缺页异常处理程序。缺页处理程序确认出物理内存中的牺牲页，如果这个页已经被修改了，则把它换到磁盘。缺页处理程序页面调入新的页面，并更新内存中的 `PTE` 缺页处理程序返回到原来的进程，再次执行之前的指令。

## 7.9 动态存储分配管理

动态内存分配器维护着一个进程的堆。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。动态内存分配主要有两种基本方法与策略：

- 带边界标签的隐式空闲链表分配器管理

带边界标记的隐式空闲链表的每个块是由一个字的头部、有效载荷、可能的

额外填充以及一个字的尾部组成的。

**隐式空闲链表：**在隐式空闲链表中，因为空闲块是通过头部中的大小字段隐含地连接着的。分配器可以通过遍历堆中所有的块，从而间接地遍历整个空闲块的集合，以已分配的而大小为零的终止头部为结束标志。

当一个应用请求一个  $m$  字节的块时，分配器搜索空闲链表，查找一个足够大的可以放置所请求块的空闲块。分配器有三种放置策略：首次适配、下一次适配、最佳适配。分配完后可以分割空闲块减少内部碎片。同时分配器在面对释放一个已分配块时，可以合并空闲块，其中便利用隐式空闲链表的边界标记来进行合并。

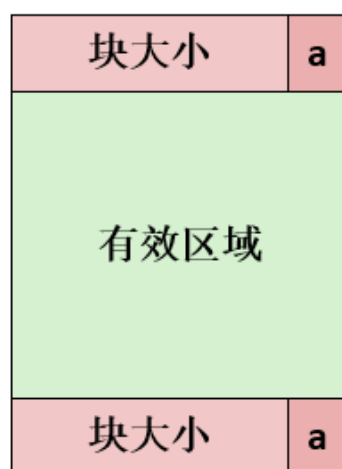


图 31 带边界隐式空闲链表的块结构

### ● 显示空间链表管理

**显式空闲链表**是将空闲块组织为某种形式的显式数据结构。因为根据定义，程序不需要一个空闲块的主体，所以实现这个数据结构的指针可以存放在这些空闲块的主体里面。如，堆可以组织成一个双向链表，在每个空闲块中，都包含一个前驱与一个后继指针。

**显式空闲链表：**在显式空闲链表中，有两种方式维护链表。可以采用后进先出的顺序维护链表，将最新释放的块放置在链表的开始处；也可以采用按照地址顺序来维护链表，其中链表中每个块的地址都小于它的后继地址。

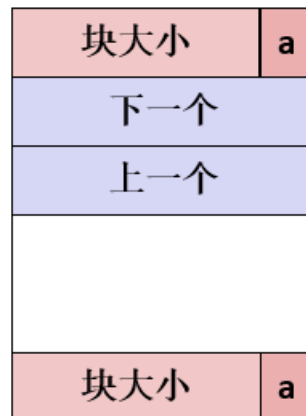


图 32 带边界显式空闲链表的块结构

## 7.10 本章小结

本章主要介绍了 `hello` 的存储地址空间、段式管理、以及页式管理，还介绍了从虚拟地址到物理地址的变换、物理内存访问机制，以及 `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、动态存储分配管理等内容。

(第 7 章 2 分)

## 第 8 章 hello 的 IO 管理

### 8.1 Linux 的 IO 设备管理方法

#### ➤ 设备的模型化：文件

所有的 I/O 设备都被模型化为文件，甚至内核也被映射为文件

#### ➤ 设备管理：unix io 接口

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。通过对文件的打开、关闭、读写、改变当前文件位置等实现设备管理

### 8.2 简述 Unix IO 接口及其函数

#### ● Unix IO 接口：

**open：**内核返回一个非负整数的文件描述符，通过对此文件描述符对文件进行所有操作。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（文件描述符 0）、标准输出（描述符为 1），标准出错（描述符为 2），通过头文件 `<unistd.h>` 中定义的宏来代替显式的描述符值。

**seek：**文件开始位置为文件偏移量，应用程序通过 `seek` 操作可设置文件的当前位置。

**read、write：**从文件复制 `m` 个字节到内存，然后更新文件位置指针；写操作时从内存复制 `m` 个字节到文件，然后更新文件位置

**close：**当应用完成对文件的访问后，通知内核关闭这个文件。

#### ● Unix IO 函数：

##### 1. `open（）` 函数

功能描述：用于打开或创建文件，在打开或创建文件时可以指定文件的属性及用户的权限等各种参数。

函数原型：`int open(const char *pathname,int flags,int perms)`

`int open(const char *pathname, int flags)`

参数：`pathname`:被打开的文件名（可加路径）`flags`:文件打开方式，

返回值：成功：返回文件描述符；失败：返回-1

##### 2. `close（）` 函数

头文件：`#include <unistd.h>`

功能描述：关闭一个被打开的文件



函数原型: `int close(int fd)`

参数: `fd`: 终止文件描述符

函数返回值: 0 成功, -1 出错

### 3. `read()` 函数

头文件: `#include <unistd.h>`

功能描述: 从文件读取数据。

函数原型: `ssize_t read(int fd, void *buf, size_t count);`

参数: `fd`: 将要读取数据的文件描述词; `buf`: 指缓冲区; `count`: 表示调用一次 `read` 操作, 应该读多少数量的字符。

返回值: 返回实际读取的字节数; 0 未读入任何数据; -1 (出错)。

### 4. `write()` 函数

头文件: `#include <unistd.h>`

功能描述: 向文件写入数据。

函数原型: `ssize_t write(int fd, void *buf, size_t count);`

返回值: 写入文件的字节数 (成功); -1 (出错)

### 5. `lseek()` 函数

头文件: `#include <unistd.h>`, `#include <sys/types.h>`

功能描述: 用于在指定的文件描述符中将文件指针定位到相应位置。

函数原型: `off_t lseek(int fd, off_t offset, int whence);`

参数: `fd`: 文件描述符。 `offset`: 偏移量, 每一个读写操作所需要移动的距离

返回值: 成功: 返回当前偏移值; 失败: 返回 -1

## 8.3 `printf` 的实现分析

通过网页的链接可以看到 `vsprintf` 的具体实现代码

```
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 != '%') {
            *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;
default:
    break;
}

return (p - buf);
}
```

`vsprintf` 函数将内容格式化（%x, %s）之后传给参数 `buf`，然后打印字符串的长度。`write` 函数将 `buf` 中的内容写到终端。从 `vsprintf` 生成显示信息，到 `write` 系统函数，到陷阱-系统调用 `int 0x80` 或 `syscall`。字符显示驱动子程序；从 ASCII 到字模库到显示 `vram`（存储每一个点的 RGB 颜色信息）。显示芯片按照刷新频率逐行读取 `vram`，并通过信号线向液晶显示器传输每一个点（RGB 分量）。

## 8.4 getchar 的实现分析

当程序调用 `getchar` 时，用户从键盘中输入内容到缓存区，然后程序从缓存区中读入一个字符，`getchar` 返回该字符的 `ascii` 码，如出错返回 -1，同时用户输入的字符会显示在屏幕。如果用户输入了多个字符，其他字符会保留在键盘缓存区中，等待后续 `getchar` 调用读取。也就是说，后续的 `getchar` 调用不会等待用户按键，而直接读取缓冲区中的字符，直到缓冲区中的字符读完为后，才等待用户按键。

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

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

## 8.5 本章小结

本章主要介绍了 Linux 的 I/O 的文件管理，以及 Unix I/O 接口及其常用的基本函数，最后粗略分析了 `printf` 函数和 `getchar` 函数的实现原理。

(第 8 章 1 分)

## 结论

### Hello 的一生心路历程:

首先是 `hello.c` 经过预处理, 拓展得到 `hello.i` 文本文件; 然后 `hello.i` 经过编译, 得到汇编代码 `hello.s` 汇编文件。再然后 `hello.s` 经过汇编, 得到二进制可重定位目标文件 `hello.o`; `hello.o` 经过链接, 生成了可执行文件 `hello`。然后通过 `bash` 进程调用 `fork` 函数, 生成子进程, 并由 `execve` 函数在当前进程的上下文中加载并运行新程序 `hello`, 其中, 通过从虚拟地址到物理地址的转化, `hello` 得以加载到内存中。`hello` 在运行时会调用一些函数与 `linux I/O` 的设备关联。最后 `hello` 被 `shell` 父进程回收, 结束一生。

CSAPP 这门课程打开了我对程序认识的新视角, 让我对程序的执行过程有了更深的理解。这门课程重在系统底层的分析, 对思维能力的锻炼有很大的作用。同时, 学习这门课程之后, 了解了程序执行的基本原理, 我对程序的优化有了新的体会, 对于创新有了新的思考方向, 它让我的思维视角看到更远、更广。

## 附件

hello.i	预处理后文件
hello.s	编译后汇编文件
hello.o	链接后的可重定位文件
helloelf.txt	hello.o 的 elf 格式文件
hello	可执行目标文件
hello_outelf.txt	hello（可执行文件）的 elf 格式文件

（附件 0 分，缺失 -1 分）

## 参考文献

- [1] 《深入理解计算机系统》 Randal E.Bryant David R.O' Hallaron 机械工业出版社
- [2] 库函数 getchar()详解  
<https://blog.csdn.net/hulifangjiayou/article/details/40480467>
- [3] 回顾 IO 库函数接口  
[https://blog.csdn.net/qq\\_48711800/article/details/116231095](https://blog.csdn.net/qq_48711800/article/details/116231095)
- [4] <https://blog.csdn.net/zhengqijun/article/details/72454714>
- [5] <https://blog.csdn.net/zycdeCSDN/article/details/102084045>

(参考文献 0 分, 缺失 -1 分)