

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 1170300827

班 級 1703008

學 生 曾逸

指 導 教 師 鄭貴濱

計算機科學與技術學院

2018 年 12 月

## 摘 要

本论文将在 linux 系统下，探究 hello.c 程序，从最初的编写到最后的运行使用整个过程中涉及到各方面知识，例如预处理、编译、汇编、链接等等。文章充分结合了深入理解计算机系统这本书，按照其章节顺序排布本篇论文，涉及的知识都可以在该书中进行查询。

**关键词：**深入理解计算机系统；P2P；hello.c；

**（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

<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 .....	- 5 -
<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 .....	- 6 -
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 -
3.4 本章小结 .....	- 14 -
<b>第 4 章 汇编 .....</b>	<b>- 15 -</b>
4.1 汇编的概念与作用 .....	- 15 -
4.2 在 UBUNTU 下汇编的命令 .....	- 15 -
4.3 可重定位目标 ELF 格式 .....	- 16 -
4.4 HELLO.O 的结果解析 .....	- 19 -
4.5 本章小结 .....	- 20 -
<b>第 5 章 链接 .....</b>	<b>- 21 -</b>
5.1 链接的概念与作用 .....	- 21 -
5.2 在 UBUNTU 下链接的命令 .....	- 21 -
5.3 可执行目标文件 HELLO 的格式 .....	- 21 -
5.4 HELLO 的虚拟地址空间 .....	- 23 -
5.5 链接的重定位过程分析 .....	- 23 -
5.6 HELLO 的执行流程 .....	- 25 -
5.7 HELLO 的动态链接分析 .....	- 25 -
5.8 本章小结 .....	- 27 -
<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 .....	- 31 -
<b>第 7 章 HELLO 的存储管理.....</b>	<b>- 32 -</b>
7.1 HELLO 的存储器地址空间 .....	- 32 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理 .....	- 32 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理 .....	- 33 -
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 .....	- 36 -
7.6 HELLO 进程 FORK 时的内存映射 .....	- 36 -
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.....	- 37 -
7.9 动态存储分配管理 .....	- 37 -
7.10 本章小结 .....	- 37 -
<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>- 44 -</b>
<b>参考文献 .....</b>	<b>- 45 -</b>

# 第 1 章 概述

## 1.1 Hello 简介

### P2P: From Program to Process

使用 C 语言编写的 `hello.c` 代码文件，C 语言是高级语言，所以这个形式的代码能让人读懂，但是系统不认识，为了让系统能够读懂代码，需要将 `hello.c` 转化成一系列机器能够读懂的语言指令，然后将这些指令按照一种称为可执行目标程序的格式进行打包，并将以二进制磁盘文件形式存放，目标程序也可以称为执行文件。

使用 GCC 编译器编译解析 `hello.c`，依次经历预处理阶段，编译阶段，汇编阶段，链接阶段（这几个阶段将会在下面论述中详细展开），最后生成了可执行目标文件 `hello`。

在 shell 中建立 `./hello` 的命令后，shell 将自动为 `hello` 起 `fork` 一个进程，这就实现了 P2P。

### 020: From Zero-0 to Zero -0

在执行 `hello` 这个目标文件中，系统 `fork` 了一个子进程。之后 `execve` 函数加载进程，创建新的内存区域以及西南的数据、堆、栈等，映射虚拟内存，进入程序入口后程序开始加载物理内存，然后从 `main` 函数执行目标代码，CPU 为 `hello` 分配时间片执行逻辑控制流。`hello` 通过 I/O 管理来控制设备的输入和输出，实现软硬件结合。当整个程序运行完成之后，进程结束，父进程回收结束的子进程，防止资源的浪费，实现 020。

## 1.2 环境与工具

硬件环境：Intel Core i5-6300HQ x64CPU, 8G RAM, 120G SSD+1T HDD

软件环境：Windows 10 64 位；Vmware 14; Ubuntu 16.04 LTS

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

## 1.3 中间结果

文件名称	文件作用
<code>Hello.c</code>	源代码

Hello.i	Hello.c 预处理后生成文本文件
Hello.s	Hello.i 编译后汇编文件
Hello.o	Hello.s 汇编后可重定位目标执行
Hello	链接之后的可执行目标文件
Hello.o.txt	反汇编代码
Hello,elf.txt	Hello 的 ELF 格式

## 1.4 本章小结

简单的叙述了 `hello.c` 从编写到程序执行最后到程序结束的整个过程，同时提供了运行环境和工具的信息，最后给出了涉及到的各种文件信息。

(第 1 章 0.5 分)

## 第 2 章 预处理

### 2.1 预处理的概念与作用

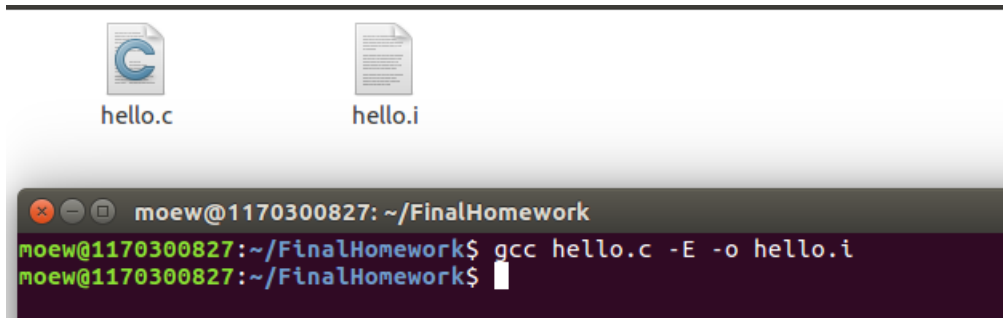
概念: 预处理器(cpp)主要处理根据以字符#开头的命令,修改原始的 C 程序。比如 hello.c 中第一行的 `#include <stdio.h>` 命令会告诉预处理其读区系统头文件 `stdio.h` 的内容,并把它直接插入到程序文本中。此过程,会得到以.i 作为扩展名。

作用:

- ①加载头文件
- ②进行宏替换
- ③条件编译

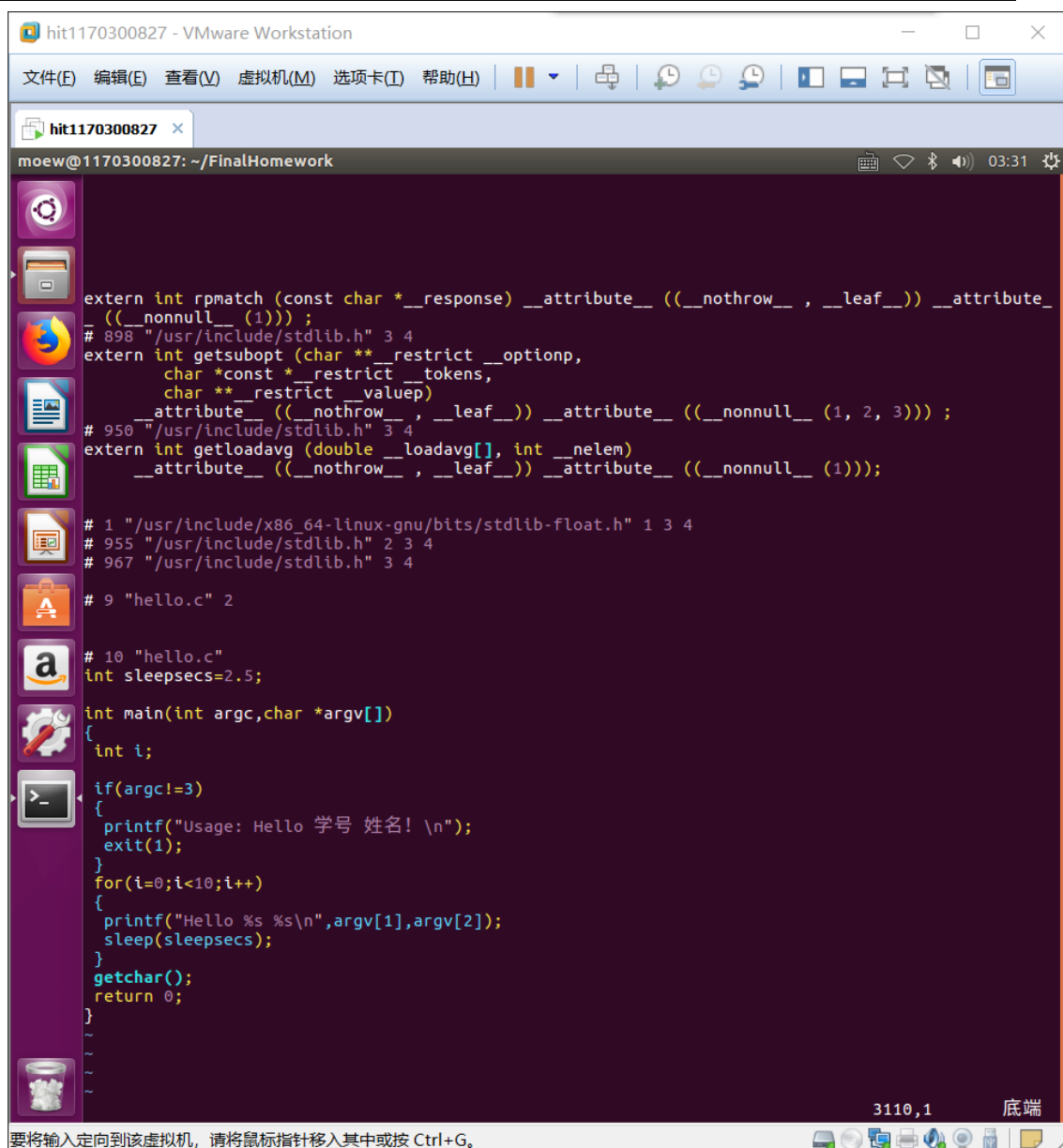
### 2.2 在 Ubuntu 下预处理的命令

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



2.2.1 预处理截图

### 2.3 Hello 的预处理结果解析



```

extern int rpmatch (const char *__response) __attribute__ ((__nothrow__ , __leaf__)) __attribute__
((__nonnull__ (1)));
# 898 "/usr/include/stdlib.h" 3 4
extern int getsubopt (char **__restrict __optionp,
                     char *const *__restrict __tokens,
                     char **__restrict __valuep)
__attribute__ ((__nothrow__ , __leaf__)) __attribute__ ((__nonnull__ (1, 2, 3)));
# 950 "/usr/include/stdlib.h" 3 4
extern int getloadavg (double __loadavg[], int __nelem)
__attribute__ ((__nothrow__ , __leaf__)) __attribute__ ((__nonnull__ (1)));

# 1 "/usr/include/x86_64-linux-gnu/bits/stdlib-float.h" 1 3 4
# 955 "/usr/include/stdlib.h" 2 3 4
# 967 "/usr/include/stdlib.h" 3 4

# 9 "hello.c" 2

# 10 "hello.c"
int sleepsecs=2.5;

int main(int argc,char *argv[])
{
    int i;

    if(argc!=3)
    {
        printf("Usage: Hello 学号 姓名! \n");
        exit(1);
    }
    for(i=0;i<10;i++)
    {
        printf("Hello %s %s\n",argv[1],argv[2]);
        sleep(sleepsecs);
    }
    getchar();
    return 0;
}
~
~
~
3110,1 底端

```

要将输入定向到该虚拟机，请将鼠标指针移入其中或按 Ctrl+G。

### 2.3.1 预处理代码

Hello.c 中的 main 函数开始于第 3110 行，之前的出现过的头文件在.i 函数里面得到了展开，调用的三个库的完整代码都被插入到了.i 文件中，其语言依然是 c 语言。不仅仅是包括头文件，其中的#define 也会得到展开，#ifdef 这类的语句则进行条件编译。

## 2.4 本章小结

本阶段完成了对 hello.c 的预处理阶段工作。了解了 cpp 具体的工作内容。



(第 2 章 0.5 分)

## 第 3 章 编译

### 3.1 编译的概念与作用

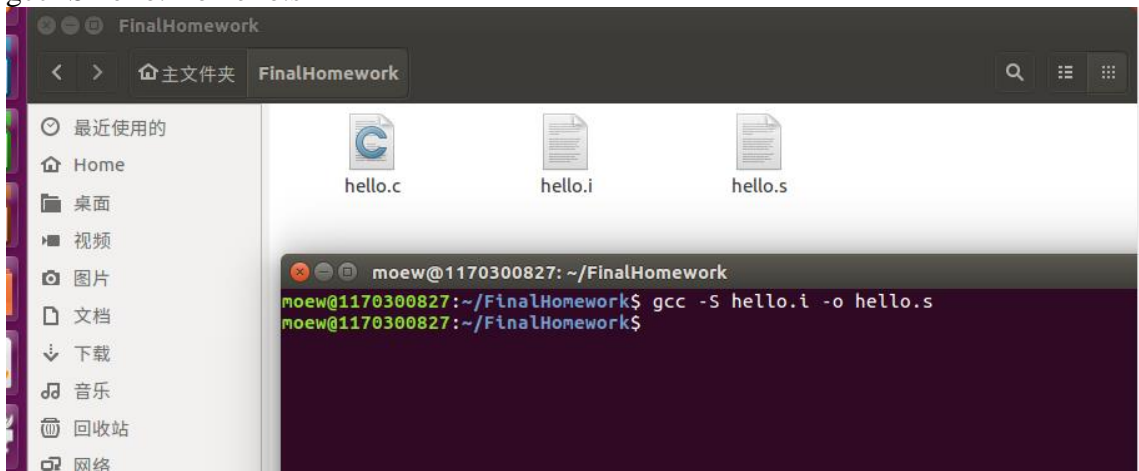
概念：编译器(cc1)将 `hello.i` 翻译成文件 `hello.s` 文件，它包含一个汇编语言程序,汇编程序中的每条语句都以一种标准的文本格式确切地描述了一条条低级机器语言指令.所以该过程会检查代码规范，语法,词法分析,具体如下图.只有编译成功之后，才能生成具体的汇编代码。

作用：

- ①将每条语句描述成一条条低级机器语言指令
- ②检查代码规范

### 3.2 在 Ubuntu 下编译的命令

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



3.2.1 编译处理

### 3.3 Hello 的编译结果解析

```

hit1170300827 - VMware Workstation
文件(F) 编辑(E) 查看(V) 虚拟机(M) 选项卡(T) 帮助(H)
hit1170300827 x
moew@1170300827: ~/FinalHomework
.file "hello.c"
.globl sleepsecs
.data
.align 4
.type sleepsecs, @object
.size sleepsecs, 4
sleepsecs:
.long 2
.section .rodata
.LC0:
.string "Usage: Hello \345\255\246\345\217\267 \345\247\223\345\220\215\357\274\201"
.LC1:
.string "Hello %s %s\n"
.text
.globl main
.type main, @function
main:
.LFB2:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $32, %rsp
movl %edi, -20(%rbp)
movq %rsi, -32(%rbp)
cmpl $3, -20(%rbp)
je .L2
movl $.LC0, %edi
call puts
movl $1, %edi
call exit
.L2:
movl $0, -4(%rbp)
jmp .L3
.L4:
movq -32(%rbp), %rax
addq $16, %rax
movq (%rax), %rdx
movq -32(%rbp), %rax
addq $8, %rax
movq (%rax), %rax
movq %rax, %rsi
movl $.LC1, %edi
movl $0, %eax
call printf
sleepsecs(%rip), %eax
movl %eax, %edi

```

6, 2-9 顶端

要将输入定向到该虚拟机, 请将鼠标指针移入其中或按 Ctrl+G.

### 3.3.1 编译代码

#### 3.3.1 常量

在程序中涉及到第一个字符串常量“Usage: Hello 学号 姓名! \n”，字符串常数存储再只读代码区的,rodata，再程序运行时会直接通过寻址找到常量，在调用printf 前，会读出地址，然后存储在寄存器中，然后传函。

```

.LC0:
.string "Usage: Hello \345\255\246\345\217\267 \345\247\223\345\220\215\357\274\201"

```

#### 3.3.2magic

不难发现，字符串被编码为 UTF-8 模式，中文字符串中每个汉字用三个字节的编码，并且以\隔开。

第二个字符串常量为“Hello %s %s\n”

```
.LC1:
.string "Hello %s %s\n"
```

### 3.3.3 第二个字符串常量

### 3.3.2 变量

除了常量，在整个程序中还存在一个全局变量 `int sleepsecs`，并且被初始化为 2.5，由于已经被初始化了，所以 `sleepsecs` 定义在 `.data` 节，而未被初始化的屈居变量和静态变量则被定义在只读代码区的 `.bss` 节。

```
.file "hello.c"
.globl sleepsecs
.data
.align 4
.type sleepsecs, @object
.size sleepsecs, 4
sleepsecs:
.long 2
```

### 3.3.4 `int sleepsecs` 属性

汇编代码给出了 `sleepsecs` 的详细定义，`.data` 代表被定义在了 `.data` 段中，由于 `int` 为整数，故小数点后面的数字直接被抹去，值为 2，隐式类型转换，设置类型为 `long`。

在 `main` 函数中存在局部变量 `i`，局部变量存储的位置是栈。

`Movl $0, -4(%rbp)` 相当于 c 语言中的 `i=0`，`addl $1, -4(%rbp)` 相当于源代码中的 `i++`，`cmpl $9, -4(%rbp)` 相当于 `i<10` 的判断句。

局部变量存储在内存中，通过 `%rbp` 相对寻址进行读写。

```
.L2:
    movl    $0, -4(%rbp)
    jmp     .L3
.L4:
    movq    -32(%rbp), %rax
    addq    $16, %rax
    movq    (%rax), %rdx
    movq    -32(%rbp), %rax
    addq    $8, %rax
    movq    (%rax), %rax
    movq    %rax, %rsi
    movl    $.LC1, %edi
    movl    $0, %eax
    call    printf
    movl    sleepsecs(%rip), %eax
    movl    %eax, %edi
    call    sleep
    addl    $1, -4(%rbp)
```

### 3.3.5 `for` 的编译代码

### 3.3.3 赋值

第一个赋值操作是 `int sleepsecs=2.5`，在上文中已经提过了它的赋值方法。在 `.data` 中直接赋值为 2。

第二个赋值则是 `i=0`，在 `for` 语句中，在汇编中的代码 `Movl $0, -4(%rbp)`，由于 `int` 是四个字节所以用 `l` 后缀。

```

movb #完成1个字节的复制
movw #完成2个字节的复制
movl #完成4个字节的复制
movq #完成8个字节的复制

```

### 3.3.6 不同字节的 mov 操作

### 3.3.4 类型转换

只涉及到了一个隐式类型转换，sleepsecs 定义是 int，但是赋值的时候赋了 2.5，故需要转换为整数，小数转换为整数的规则是向下取整，将小数点后面的数字直接抹除，故 sleepsecs 实际值变为 2。

### 3.3.5 算数操作

在 for 循环中存在 i++，使用指令 `addl $1, -4(%rbp)`

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

### 3.3.7 算数操作指令

### 3.3.6 关系操作

`if(argc!=3)` 条件判断语句存在一个判断关系。

汇编的代码为 `cmpl $3, -20(%rbp)`, 用 `argv-3`, 然后设置条件码, 与 `je .L2` 连用。

`for(i=0; i<10; i++)` `i<10` 需进行大小比较

汇编语句为 `cmpl $9, -4(%rbp)`, 用 `i-9` 设置条件码, 与 `jle .L4` 连用实现跳转。

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

### 3.3.8 比较操作汇编指令

#### 3.3.7 数组

`char *argv[]` 为该函数涉及到数组, `argv` 存放的是 `char` 的指针, 在之前的作业中, 我们有写过一篇关于数组的栈帧分析报告。数组存储在连续空间中。同时是指针, 故需要取地址操作。由 `printf` 函数的操作, 可以推断出 `argc` 的位置。直接由寄存器保存。

```
addq $16, %rax
movq (%rax), %rdx 第一个 argv
movq (%rax), %rax
movq %rax, %rsi 第二个 argv
```

### 3.3.9 和 3.3.10 数组地址

#### 3.3.8 控制转移

由关系操作中的判断完成后, 则根据条件进行控制转移。

If 判断中使用 `je` 判断 `ZF` 标志位, 如果 `argv=3`, 则通过跳转表, 执行 `.L2` 代码段, 进行 `for` 循环语句, 如果 `argv` 不等于 3, 顺序执行。

```
cmpl $3, -20(%rbp)
je .L2
.L2:
    movl $0, -4(%rbp)
    jmp .L3
```

### 3.3.11 和 3.3.12 判断条件执行

For 中的从 `.L2` 先进入 `.L3` 代码段进行比较, 与 9 比较, 如果小于等于 9, 则进入 `.L4` 代码段, 否则顺序执行。

```

.L2:      movl    $0, -4(%rbp)
          jmp     .L3
.L4:      movq    -32(%rbp), %rax
          addq    $16, %rax
          movq    (%rax), %rdx
          movq    -32(%rbp), %rax
          addq    $8, %rax
          movq    (%rax), %rax
          movq    %rax, %rsi
          movl    $.LC1, %edi
          movl    $0, %eax
          call    printf
          movl    sleepsecs(%rip), %eax
          movl    %eax, %edi
          call    sleep
          addl    $1, -4(%rbp)
.L3:      cmpl    $9, -4(%rbp)
          jle     .L4
          call    getchar
          movl    $0, %eax
          leave

```

3.3.13 条件跳转代码段

### 3.3.9 函数操作

函数操作涉及调用过程，当 P 调用 Q 时，P 及所有向上的调用链全被挂起，PC 设置为 Q 代码的起始位置，P 向 Q 传参，Q 返回则会释放为其分配的空间。

X64 是用 6 位寄存器，超过 6 位则在栈上存储。

一共设计了五个函数，main、printf、exit、sleep、getchar。

Main 函数被系统调用，传入参数 argc 和 argv，分别存在 %rdi 和 %rsi, %eax 存储返回值 0，main 函数运行完后，调用 leave 指令，释放栈上的空间，然后 ret 返回。剩下的四个函数则被 main 调用，调用方式基本与 main 一致，通过 call 指令实现调用。

## 3.4 本章小结

按照数据、赋值、类型转换、算数操作、关系操作、数组、控制转移、函数操作顺序，分析了整个函数的汇编代码，对每一条的 c 语言都在汇编过程中得到了解释。

**(第 3 章 2 分)**

## 第 4 章 汇编

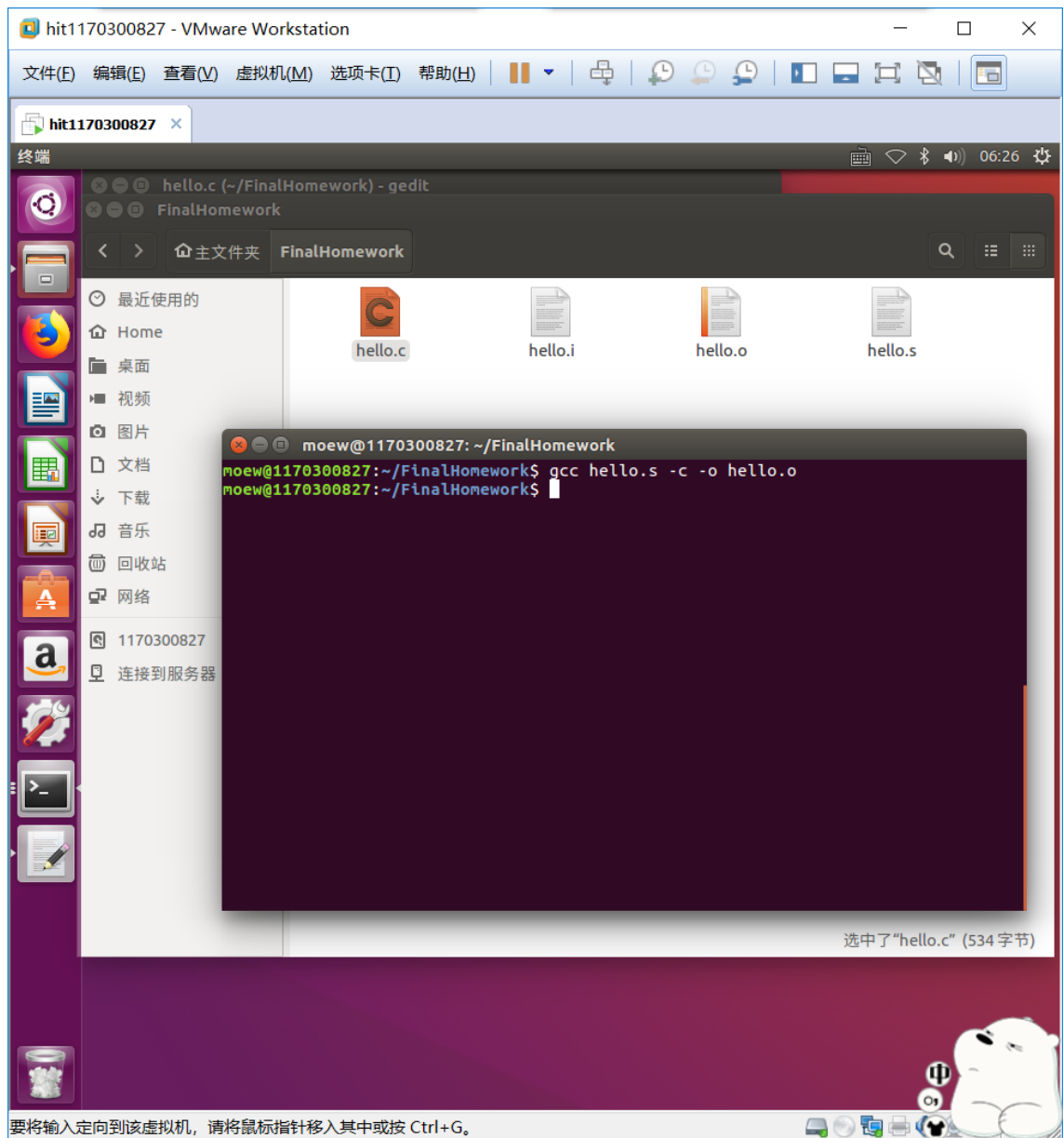
### 4.1 汇编的概念与作用

概念：汇编器(as)将 `hello.s` 文件翻译成机器语言指令，把这些指令打包成一种叫做可重定向目标程序的格式，并且保存在 `hello.o` 文件中。该文件是一个二进制文件，它的字节编码是机器语言指令而不是字符，如果用编辑器打开将是一段乱码。

作用：实现将汇编代码转换为机器指令，使之在链接后能够被计算机直接执行

### 4.2 在 Ubuntu 下汇编的命令





#### 4.2.1 汇编操作

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

#### 4.3 可重定位目标 elf 格式

```
moew@1170300827: ~/FinalHomework
moew@1170300827:~/FinalHomework$ readelf -h hello.o
ELF 头:
  Magic:      7f 45 4c 46 02 01 01 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: 1112 (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: 10
moew@1170300827:~/FinalHomework$
```

#### 4.3.1 ELF 头信息

ELF 头以一个 16 字节的序列开始, 即 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00 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。

节头:							
[号]	名称	类型	地址	链接	偏移量	信息	对齐
	大小	全体大小	旗标				
[ 0]	0000000000000000	NULL	0000000000000000	0	0	0	0
[ 1]	.text	PROGBITS	0000000000000000	0	0	0	1
[ 2]	.rela.text	RELA	0000000000000000	11	1	8	1
[ 3]	.data	PROGBITS	0000000000000000	WA	0	0	4
[ 4]	.bss	NOBITS	0000000000000000	WA	0	0	1
[ 5]	.rodata	PROGBITS	0000000000000000	A	0	0	1
[ 6]	.comment	PROGBITS	0000000000000000	MS	0	0	1
[ 7]	.note.GNU-stack	PROGBITS	0000000000000000	0	0	0	1
[ 8]	.eh_frame	PROGBITS	0000000000000000	A	0	0	8
[ 9]	.rela.eh_frame	RELA	0000000000000000	I	11	8	8
[10]	.shstrtab	STRTAB	0000000000000000	0	0	0	1
[11]	.symtab	SYMTAB	0000000000000000	12	9	8	8
[12]	.strtab	STRTAB	0000000000000000	0	0	0	1
	0000000000000037	0000000000000000					

Key to Flags:  
W (write), A (alloc), X (execute), M (merge), S (strings), l (large)  
I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)  
0 (extra OS processing required) o (OS specific), p (processor specific)

#### 4.3.2 节头信息

不同节的位置和大小是由节头部表描述的，其中，目标文件中每个节都有一个固定大小的条目（entry）。

重定位节 '.rela.text' 位于偏移量 0x318 含有 8 个条目:					
偏移量	信息	类型	符号值	符号名称	+ 加数
000000000016	00050000000a	R_X86_64_32	0000000000000000	.rodata + 0	
00000000001b	000b00000002	R_X86_64_PC32	0000000000000000	puts - 4	
000000000025	000c00000002	R_X86_64_PC32	0000000000000000	exit - 4	
00000000004c	00050000000a	R_X86_64_32	0000000000000000	.rodata + 1e	
000000000056	000d00000002	R_X86_64_PC32	0000000000000000	printf - 4	
00000000005c	000900000002	R_X86_64_PC32	0000000000000000	sleepsecs - 4	
000000000063	000e00000002	R_X86_64_PC32	0000000000000000	sleep - 4	
000000000072	000f00000002	R_X86_64_PC32	0000000000000000	getchar - 4	

#### 4.3.3 重定位信息

汇编器遇到对最终位置位置的目标引用，它就会生成一个重定位条目。

偏移量（offset）是需要被修改的引用的节偏移，例如 puts 的节偏移量为 00000000001b，告诉连接器修改开始于偏移量 0x1b 处的 32 位 PC 相对引用，使他在运行是指向 puts 例程。

信息包括 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，类型包括相对地址引用和绝对地址应用。

符号名称则是重定位目标的名字。

加数（addend）使用它对被修改引用的值做偏移调整。

地址具体算法如下：

节运行地址（ADDR(s)）符号运行地址 ADDR(r.symbol)

refptr = s + r.offset /\*指针位置\*/

R\_X86\_64\_PC32：相对寻址

refaddr = ADDR(s) + r.offset/\*引用运行地址计算\*/

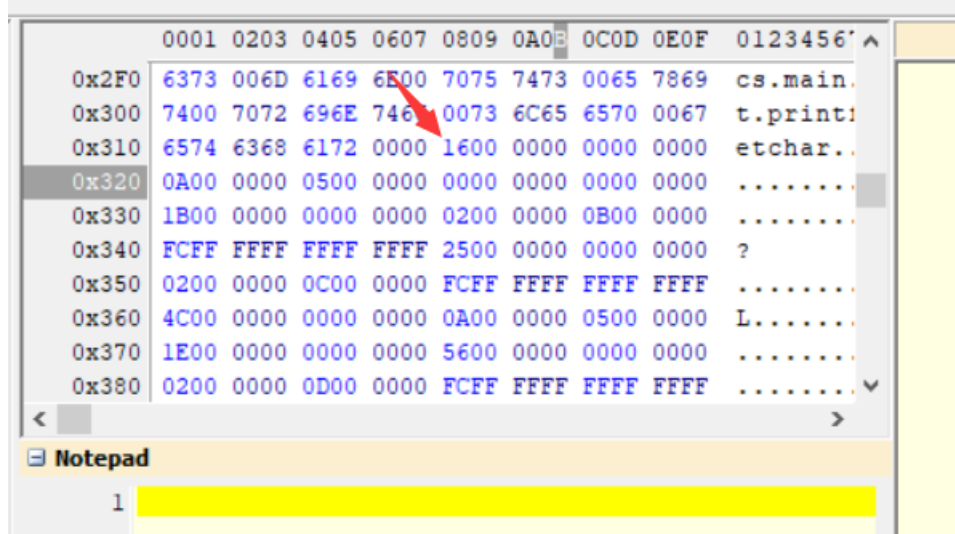
refptr = (unsigned) (ADDR(r.symbol) + r.addend-refaddr)/\*指向被引用的相对 PC 地址计算\*/

R\_X86\_64\_32：绝对寻址

refaddr = ADDR(s) + r.offset/\*引用运行地址计算\*/

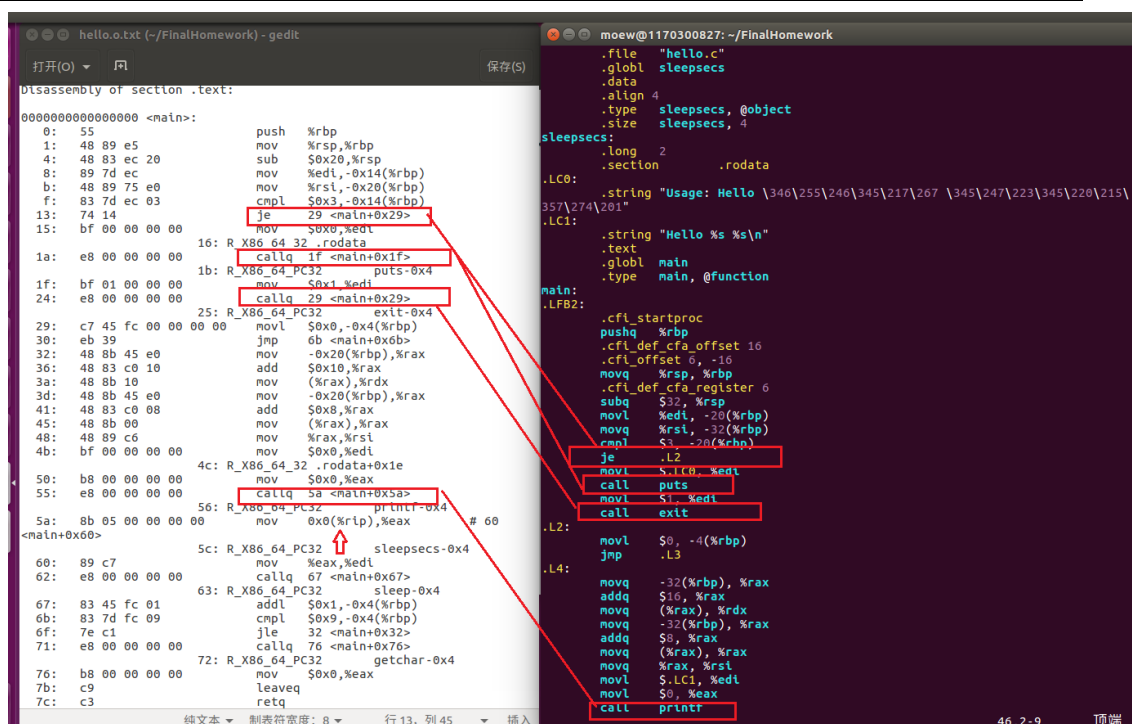
refptr = (unsigned) (ADDR(r.symbol) + r.addend)/\*指向被引用的相对 PC 地址计算\*/

以.rodata 为例，offset=0x16,addend=0，绝对寻址，用 hexedit 可以查看到：



4.3.5Hexedit 查看 hello.o

## 4.4 Hello.o 的结果解析



#### 4.4.1 汇编和反汇编代码对照

两者的并没有天差地别，可读性较高，但是还存在一定的不同。

尤其是在分支转移中，反汇编中不再出现段代码，而是变成了一个确定地址。函数的调用也变成了地址，连接器先链接静态库，变成部分链接可执行目标文件，之后再由加载器调用动态连接生成完全链接的可执行文件。

全局变量因为地址确定了，故直接访问`%rip`，而非段名称+`%rip`。

## 4.5 本章小结

本章介绍了从.s 到.o 的汇编过程，编译和汇编两个过程看起来差别不大，但是其中却发生了很多事情，.o 文件是更机器化的一种文件，更加明确具体了每一步指令。

(第4章1分)

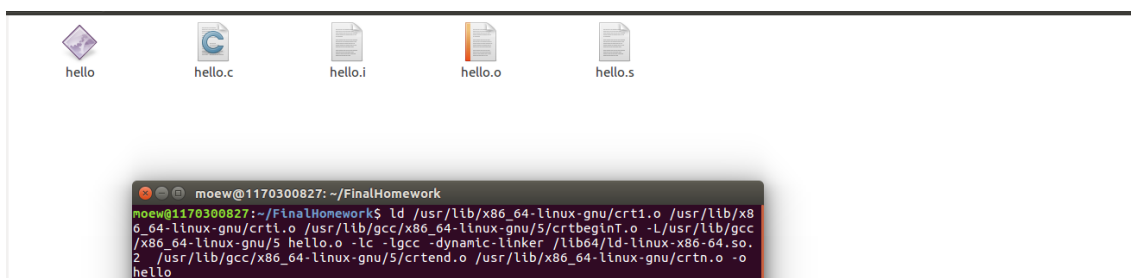
## 第 5 章 链接

### 5.1 链接的概念与作用

**链接：**链接（linking）是将各种代码和数据片段收集并组合成为一个单一文件的过程，这个文件可被加载（复制）到内存并执行

**作用：**分离编译（separate compilation）。我们不用将一个大型应用程序组织为一个巨大的源文件，而是可以把它分解为更小、更好管理的模块，可以独立的修改和编译这些模块。

### 5.2 在 Ubuntu 下链接的命令



#### 5.2.1 链接操作

```
ld      /usr/lib/x86_64-linux-gnu/crt1.o      /usr/lib/x86_64-linux-gnu/crti.o
/usr/lib/gcc/x86_64-linux-gnu/5/crtbeginT.o -L/usr/lib/gcc/x86_64-linux-gnu/5 hello.o
-lc      -lgcc      -dynamic-linker      /lib64/ld-linux-x86-64.so.2
/usr/lib/gcc/x86_64-linux-gnu/5/crtend.o /usr/lib/x86_64-linux-gnu/crtn.o -o hello
```

### 5.3 可执行目标文件 hello 的格式



节头:						
[号]	名称 大小	类型 全体大小	地址 旗标	链接 链接	偏移量 信息	对齐
[ 0]	0000000000000000	NULL	0000000000000000	0	0	0
[ 1]	.interp 000000000000001c	PROGBITS 0000000000000000	0000000000400238 A	0	0	1
[ 2]	.note.ABI-tag 0000000000000020	NOTE 0000000000000000	0000000000400254 A	0	0	4
[ 3]	.note.gnu.build-id 0000000000000024	NOTE 0000000000000000	0000000000400274 A	0	0	4
[ 4]	.gnu.hash 000000000000001c	GNU_HASH 0000000000000000	0000000000400298 A	5	0	8
[ 5]	.dynsym 00000000000000c0	DYSYM 0000000000000018	00000000004002b8 A	6	1	8
[ 6]	.dynstr 0000000000000057	STRTAB 0000000000000000	0000000000400378 A	0	0	1
[ 7]	.gnu.version 0000000000000010	VERSYM 0000000000000002	00000000004003d0 A	5	0	2
[ 8]	.gnu.version_r 0000000000000020	VERNEED 0000000000000000	00000000004003e0 A	6	1	8
[ 9]	.rela.dyn 0000000000000018	RELA 0000000000000018	0000000000400400 A	5	0	8
[10]	.rela.plt 0000000000000090	RELA 0000000000000018	0000000000400418 AI	5	24	8
[11]	.init 000000000000001a	PROGBITS 0000000000000000	00000000004004a8 AX	0	0	4
[12]	.plt 0000000000000070	PROGBITS 0000000000000010	00000000004004d0 AX	0	0	16
[13]	.plt.got 0000000000000008	PROGBITS 0000000000000000	0000000000400540 AX	0	0	8
[14]	.text 00000000000001f2	PROGBITS 0000000000000000	0000000000400550 AX	0	0	16
[15]	.fini 0000000000000009	PROGBITS 0000000000000000	0000000000400744 AX	0	0	4
[16]	.rodata 000000000000002f	PROGBITS 0000000000000000	0000000000400750 A	0	0	4
[17]	.eh_frame_hdr 0000000000000034	PROGBITS 0000000000000000	0000000000400780 A	0	0	4
[18]	.eh_frame 00000000000000f4	PROGBITS 0000000000000000	00000000004007b8 A	0	0	8
[19]	.init_array 0000000000000000	INIT_ARRAY 0000000000000000	0000000000600e10 WA	0	0	8
[20]	.fini_array 0000000000000008	FINI_ARRAY 0000000000000000	0000000000600e18 WA	0	0	8
[21]	.jcr 0000000000000008	PROGBITS 0000000000000000	0000000000600e20 WA	0	0	8
[22]	.dynamic 00000000000001d0	DYNAMIC 0000000000000010	0000000000600e28 WA	6	0	8
[23]	.got 0000000000000008	PROGBITS 0000000000000008	0000000000600ff8 WA	0	0	8
[24]	.got.plt 0000000000000048	PROGBITS 0000000000000008	0000000000601000 WA	0	0	8
[25]	.data 0000000000000014	PROGBITS 0000000000000000	0000000000601048 WA	0	0	8
[26]	.bss 0000000000000004	NOBITS 0000000000000000	000000000060105c WA	0	0	1
[27]	.comment 0000000000000035	PROGBITS 0000000000000001	0000000000000000 MS	0	0	1
[28]	.shstrtab 000000000000010c	STRTAB 0000000000000000	0000000000000000	0	0	1
[29]	.symtab 000000000000006c0	SYMTAB 0000000000000018	0000000000000000	30	47	8
[30]	.strtab 000000000000026c	STRTAB 0000000000000000	0000000000000000	0	0	1

## 5.3.1 可执行目标文件 hello 节头





```

0000000000400646 <main>:
400646: 55                push    %rbp
400647: 48 89 e5          mov     %rsp,%rbp
40064a: 48 83 ec 20       sub     $0x20,%rsp
40064e: 89 7d ec          mov     %edi,-0x14(%rbp)
400651: 48 89 75 e0       mov     %rsi,-0x20(%rbp)
400655: 83 7d ec 03       cmpl    $0x3,-0x14(%rbp)
400659: 74 14             je      40066f <main+0x29>
40065b: bf 54 07 40 00    mov     $0x400754,%edi
400660: e8 7b fe ff ff    callq   4004e0 <puts@plt>
400665: bf 01 00 00 00    mov     $0x1,%edi
40066a: e8 b1 fe ff ff    callq   400520 <exit@plt>
40066f: c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%rbp)
400676: eb 39             jmp     4006b1 <main+0x6b>
400678: 48 8b 45 e0       mov     -0x20(%rbp),%rax
40067c: 48 83 c0 10       add     $0x10,%rax
400680: 48 8b 10          mov     (%rax),%rdx
400683: 48 8b 45 e0       mov     -0x20(%rbp),%rax
400687: 48 83 c0 08       add     $0x8,%rax
40068b: 48 8b 00          mov     (%rax),%rax
40068e: 48 89 c6          mov     %rax,%rsi
400691: bf 72 07 40 00    mov     $0x400772,%edi
400696: b8 00 00 00 00    mov     $0x0,%eax
40069b: e8 50 fe ff ff    callq   4004f0 <printf@plt>
4006a0: 8b 05 b2 09 20 00 mov     0x2009b2(%rip),%eax    # 601058 <sleepsecs>
4006a6: 89 c7             mov     %eax,%edi
4006a8: e8 83 fe ff ff    callq   400530 <sleep@plt>
4006ad: 83 45 fc 01       addl    $0x1,-0x4(%rbp)
4006b1: 83 7d fc 09       cmpl    $0x9,-0x4(%rbp)
4006b5: 7e c1             jle     400678 <main+0x32>
4006b7: e8 54 fe ff ff    callq   400510 <getchar@plt>
4006bc: b8 00 00 00 00    mov     $0x0,%eax
4006c1: c9               leaveq  %eax
4006c2: c3               retq
4006c3: 66 2e 0f 1f 84 00 00 nopw    %cs:0x0(%rax,%rax,1)
4006ca: 00 00 00          nopl    (%rax)
4006cd: 0f 1f 00          nopl    (%rax)

```

### 5.5.1 连接后的反汇编

可执行文件的反汇编结果中给出了重定位结果，即虚拟地址的确定。而 hello.o 的反汇编结果中，各部分的开始地址均为 0。

地址具体算法如下：

节运行地址（ADDR(s)）符号运行地址 ADDR(r.symbol)

refptr = s + r.offset /\*指针位置\*/

R\_X86\_64\_PC32：相对寻址

refaddr = ADDR(s) + r.offset/\*引用运行地址计算\*/

refptr = (unsigned) (ADDR(r.symbol) + r.addend-refaddr)/\*指向被引用的相对 PC 地址计算\*/

R\_X86\_64\_32：绝对寻址

refaddr = ADDR(s) + r.offset/\*引用运行地址计算\*/

refptr = (unsigned) (ADDR(r.symbol) + r.addend)/\*指向被引用的相对 PC 地址计算\*/

以.rodata 为例，offset=0x16,addend=0，绝对寻址。

在加载的时候，加载器会把这些节中的字节直接复制到内存，不在进行任何修改地执行这些指令。

## 5.6 hello 的执行流程

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

### 5.6.1 hello 的顺序执行流程

## 5.7 Hello 的动态链接分析

动态链接库中的函数在程序执行的时候才会确定地址，所以编译器无法确定其地址，在汇编代码中也无法像静态库的函数那样体现。

hello 程序对动态链接库的引用，基于数据段与代码段相对距离不变这一个事

实，因此代码段中任何指令和数据段中任何变量之间的距离都是一个运行时常量。

GNU 编译系统采用延迟绑定技术来解决动态库函数模块调用的问题，它将过程地址的绑定推迟到了第一次调用该过程时。

延迟绑定通过全局偏移量表（GOT）和过程链接表（PLT）实现。如果一个目标模块调用定义在共享库中的任何函数，那么就有自己的 GOT 和 PLT。前者是数据段的一部分，后者是代码段的一部分。

进一步介绍，PLT 是一个数组，其中每个条目是 16 字节代码。每个库函数都有自己的 PLT 条目，PLT[0]是一个特殊的条目，跳转到动态链接器中。从 PLT[2]开始的条目调用用户代码调用的函数。

GOT 同样是一个数组，每个条目是 8 字节的地址，和 PLT 联合使用时，GOT[2]是动态链接在 ld-linux.so 模块的入口点，其余条目对应于被调用的函数，在运行时被解析。每个条目都有匹配的 PLT 条目。

当某个动态链接函数第一次被调用时先进入对应的 PLT 条目例如 PLT[2]，然后 PLT 指令跳转到对应的 GOT 条目中例如 GOT[4]，其内容是 PLT[2]的下一条指令。然后将函数的 ID 压入栈中后跳转到 PLT[0]。PLT[0]通过 GOT[1]将动态链接库的一个参数压入栈中，再通过 GOT[2]间接跳转进动态链接器中。动态链接器使用两个栈条目来确定函数的运行时位置，用这个地址重写 GOT[4]，然后再次调用函数。经过上述操作，再次调用时 PLT[2]会直接跳转通过 GOT[4]跳转到函数而不是 PLT[2]的下一条地址。

Address	Value	Comment
00000000:00600fe0	00 00 00 00 00 00 00 00	
00000000:00600ff0	00 00 00 00 00 00 00 00	
00000000:00601000	28 0e 60 00 00 00 00 00	
00000000:00601010	00 00 00 00 00 00 00 00	
00000000:00601020	f6 04 40 00 00 00 00 00	
00000000:00601030	16 05 40 00 00 00 00 00	
00000000:00601040	36 05 40 00 00 00 00 00	
00000000:00601050	00 00 00 00 00 00 00 00	
00000000:00601060	00 00 00 00 00 00 00 00	
00000000:00601070	00 00 00 00 00 00 00 00	

5.7.11dl init 前

Address	Value	Comment
00000000:00601000	28 0e 60 00 00 00 00 00	
00000000:00601010	70 88 f0 83 85 7f 00 00	
00000000:00601020	f6 04 40 00 00 00 00 00	
00000000:00601030	16 05 40 00 00 00 00 00	
00000000:00601040	36 05 40 00 00 00 00 00	

5.7.2 dl init 后

在 edb 调试之后我们发现原先 0x00600a10 开始的 global\_offset 表是全 0 的状态，在执行过 dl\_init 之后被赋上了相应的偏移量的值。这说明 dl\_init 操作是给程序赋上当前执行的内存地址偏移量，这是初始化 hello 程序的一步。

## 5.8 本章小结

(以下格式自行编排, 编辑时删除)

(第 5 章 1 分)

## 第 6 章 hello 进程管理

### 6.1 进程的概念与作用

概念：

进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是操作系统动态执行的基本单元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元。

作用：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。

### 6.2 简述壳 Shell-bash 的作用与处理流程

#### 1.作用

shell 是一种交互型的应用级程序。它能够接收用户命令，然后调用相应的应用程序，即代表用户运行其他程序。

#### 2.处理流程

shell 执行一系列的读/求值步骤，然后终止。读步骤读取来自用户的一个命令行。求值步骤解析命令行，并代表用户运行程序。

### 6.3 Hello 的 fork 进程创建过程

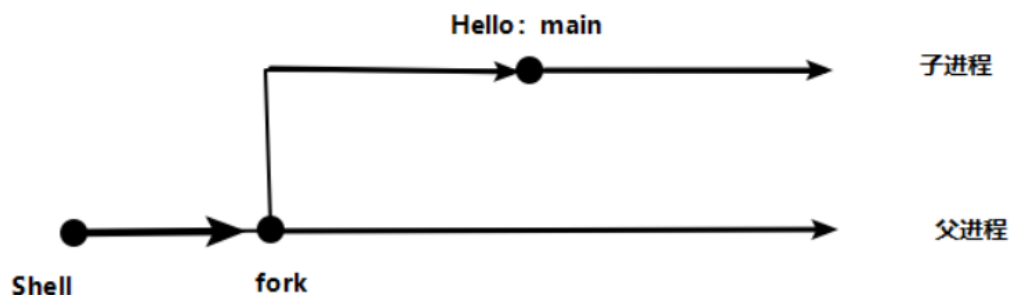
```
moew@1170300827:~/FinalHomework$ ./hello 1170300827 曾逸
Hello 1170300827 曾逸
Hello 1170300827 曾逸
Hello 1170300827 曾逸
Hello 1170300827 曾逸
Hello 1170300827 曾逸
Hello 1170300827 曾逸
Hello 1170300827 曾逸
Hello 1170300827 曾逸
Hello 1170300827 曾逸
Hello 1170300827 曾逸
Hello 1170300827 曾逸
```

#### 6.3.1 运行 HELLO

调用程序后，对命令行进行解析，./hello 意味执行当前目录下的可执行文件 hello,之后终端调用 fork 函数创建一个子进程，在子进程中运行改程序，子进程的各类属性基本与父进程基本相同，通过 PID 区分两者。一般而言父进程和子进程

是并发进行的，不可以假设两者进行的顺序。

但是 shell 中父进程会显示的等待子进程完成，子进程完成用 `waitpid` 函数进行回收，防止资源浪费。



6.3.2hello 进程图

## 6.4 Hello 的 `execve` 过程

`Fork()`之后，在子进程中调用 `execve()` 函数，在当前进程中加载并运行一个新程序。`Execve` 加载并运行可执行目标文件 `hello`，且带参数列表 `argv` 和环境变量列表 `envp`。在 `execve` 加载了可执行程序之后，它调用启动代码。启动代码设置栈，并将控制传递给新程序的主函数，即可执行程序的 `main` 函数。此时用户栈已经包含了命令行参数与环境变量，进入 `main` 函数后便开始逐步运行程序。

## 6.5 Hello 的进程执行

多个流并发地执行的一般现象被称为并发。一个进程和其他进程轮流运行的概念称为多任务。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。因此，多任务也叫做时间分片。

`Hello` 运行时，一直处于用户模式，知道调用 `sleep` 陷入内核模式，需要进行上下切换，先保存以前进程的上下文，打开新的回复进程，之后控制传递，休眠 2s 后，发送中断信号，执行信号处理，调用用户模式，`hello` 进程继续。

程序在涉及到一些操作时，例如调用一些系统函数，内核需要将当前状态从用户态切换到核心态，执行结束后再及时改用户态，从而保证系统的安全与稳定。

*(以下格式自行编排，编辑时删除)*

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

## 6.6 hello 的异常与信号处理

```
moew@1170300827: ~/FinalHomework
moew@1170300827:~/FinalHomework$ ./hello 1170300827 Zoe
Hello 1170300827 Zoe
Hello 1170300827 Zoe
^C
moew@1170300827:~/FinalHomework$ ps
  PID TTY          TIME CMD
 5547 pts/20    00:00:00 bash
 5586 pts/20    00:00:00 ps
moew@1170300827:~/FinalHomework$
```

### 6.6.1 按下 Ctrl+c

父进程收到 SIGINT 信号，终止了 hello。

```
moew@1170300827: ~/FinalHomework
moew@1170300827:~/FinalHomework$ ./hello 1170300827 Zoe
Hello 1170300827 Zoe
jskdlaHello 1170300827 Zoe
sad
asdak ldjda
asdkHello 1170300827 Zoe
alsd
wdaklsdla
dsakdljsHello 1170300827 Zoe

admak
adlaHello 1170300827 Zoe
sd0a
sdas l;asd
aHello 1170300827 Zoe
[qp[
Hello 1170300827 Zoe
Hello 1170300827 Zoe
Hello 1170300827 Zoe
Hello 1170300827 Zoe
moew@1170300827:~/FinalHomework$ asdak ldjda
asdak: 未找到命令
moew@1170300827:~/FinalHomework$ asdkalsd
asdkalsd: 未找到命令
```

### 6.6.2 乱按之后

会将乱按的东西缓存，\n 表示一行命令的终止，hello 结束后，shell 解读这些被缓存的命令



```
moew@1170300827: ~/FinalHomework
moew@1170300827:~/FinalHomework$ ./hello 1170300827 Zoe
Hello 1170300827 Zoe
Hello 1170300827 Zoe
Hello 1170300827 Zoe
^Z
[1]+ 已停止                  ./hello 1170300827 Zoe
moew@1170300827:~/FinalHomework$ ps
  PID TTY          TIME CMD
  5547 pts/20      00:00:00 bash
  5573 pts/20      00:00:00 hello
  5577 pts/20      00:00:00 ps
moew@1170300827:~/FinalHomework$ fg 1
./hello 1170300827 Zoe
Hello 1170300827 Zoe
Hello 1170300827 Zoe
Hello 1170300827 Zoe
Hello 1170300827 Zoe
Hello 1170300827 Zoe
Hello 1170300827 Zoe
Hello 1170300827 Zoe
Hello 1170300827 Zoe
bgfg
moew@1170300827:~/FinalHomework$ jobs
moew@1170300827:~/FinalHomework$
```

### 6.6.3 按下 Ctrl+z

父进程收到 SIGSTP 信号，将 hello 程序挂起，此时 hello 并未结束，而是出于后台，fg 1 将其从后台调出，继续完成剩下的程序。

## 6.7 本章小结

本章了解到了应用是如何与系统交互的，这些交互都是围绕着 ECF 的。了解了进程的相关概念。程序在 shell 中执行是通过 fork 函数及 execve 创建新的进程并执行程序。进程拥有着与父进程相同却又独立的环境，与其他系统进程并发执行，拥有各自的时间片，在内核的调度下有条不紊的执行着各自的指令。

**(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：**逻辑地址(LogicalAddress)是指由程序产生的与段相关的偏移地址部分。就是 hello.o 里面的相对偏移地址。

**线性地址：**地址空间(address space) 是一个非负整数地址的有序集合，如果地址空间中的整数是连续的，那么我们说它是一个线性地址空间(linear address space)。就是 hello 里面的虚拟内存地址。

**虚拟地址：**CPU 通过生成一个虚拟地址(Virtual Address, VA)。就是 hello 里面的虚拟内存地址。

**物理地址：**用于内存芯片级的单元寻址，与处理器和 CPU 连接的地址总线相对应。计算机系统的主存被组织成一个由 M 个连续的字节大小的单元组成的数组。每字节都有一个唯一的物理地址。就是 hello 在运行时虚拟内存地址对应的物理地址。

### 7.2 Intel 逻辑地址到线性地址的变换-段式管理

最初 8086 处理器的寄存器是 16 位的，为了能够访问更多的地址空间但不改变寄存器和指令的位宽，所以引入段寄存器，8086 共设计了 20 位宽的地址总线，通过将段寄存器左移 4 位加上偏移地址得到 20 位地址，这个地址就是逻辑地址。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。实模式，即不设防，也就是说逻辑地址=线性地址=实际的物理地址。段寄存器存放真实段基址，同时给出 32 位地址偏移量，则可以访问真实物理内存。在保护模式下，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。段寄存器无法放下 32 位段基址，所以它们被称作选择符，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。

**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 0（内核模式）-3（用户模式）。

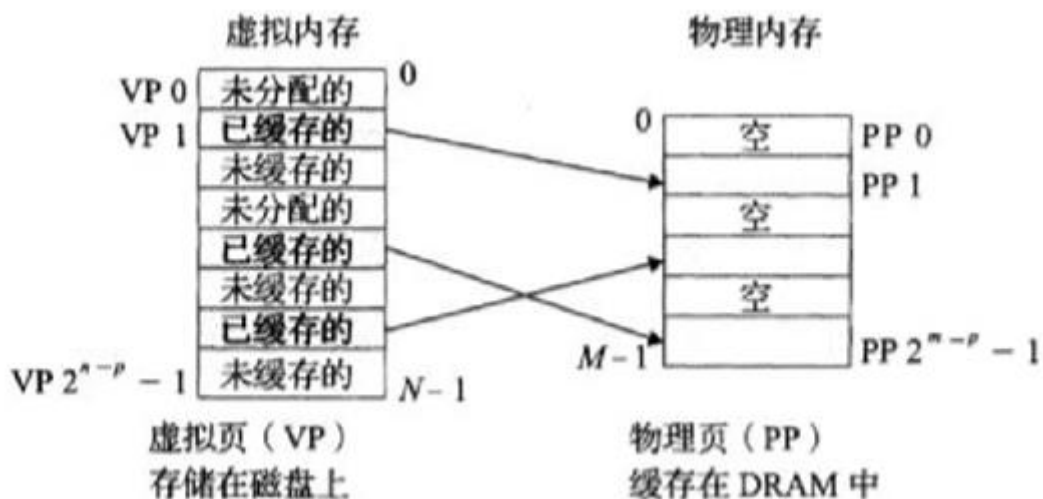
所有的段描述符被保存在两个表中：全局描述符表 GDT 和局部描述符表 LDT。gdt 寄存器指向 GDT 表基址。

在保护模式下，分段机制就可以描述为：通过解析段寄存器中的段选择符在段描述符表中根据 Index 选择目标描述符条目 Segment Descriptor，从目标描述符中提取出目标段的基地址 Base address，最后加上偏移量 offset 共同构成线性地址 Linear Address。

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，此时逻辑地址=描述符=线性地址，Intel 的文档中将其称为扁平模型（flat model），现代的 x86 系统内核使用的是基本扁平模型，等价于转换地址时关闭了分段功能。在 CPU 64 位模式中强制使用扁平的线性空间。逻辑地址与线性地址就合二为一了。

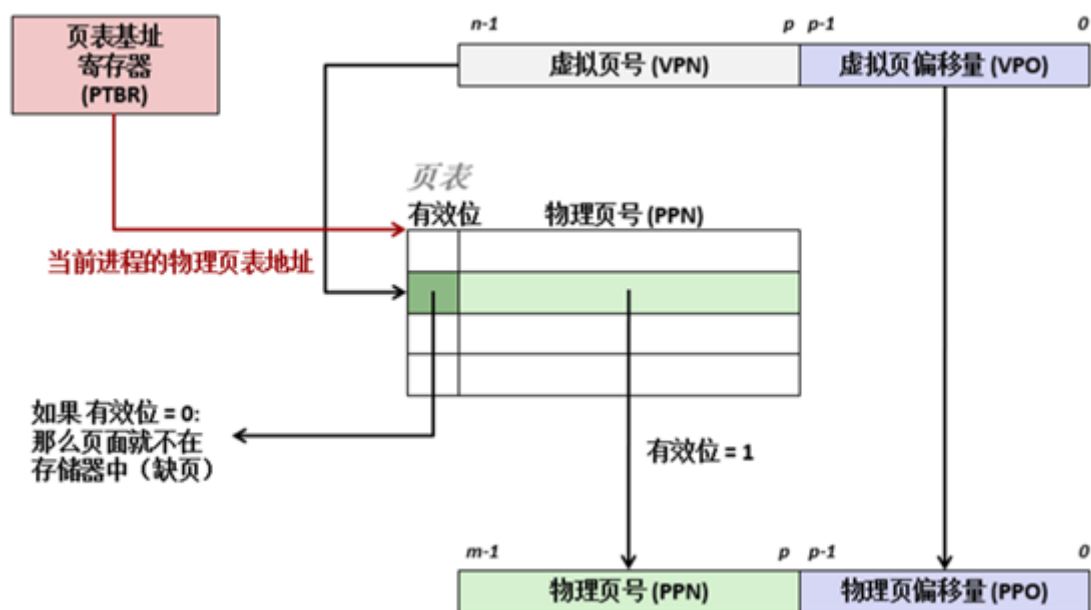
### 7.3 Hello 的线性地址到物理地址的变换-页式管理

概念上而言，虚拟内存被组织为一个由存放在磁盘上的  $N$  个连续的字节大小的单元组成的数组。每字节都有一个唯一的虚拟地址，作为到数组的索引。磁盘上数组的内容被缓存在主存中。和存储器层次结构中其他缓存一样，磁盘（较低层）上的数据被分割成块，这些块作为磁盘和主存（较高层）之间的传输单元。VM 系统通过将虚拟内存分割位称为虚拟页的大小固定的块来处理这个问题。每个虚拟页的大小为  $P = 2^p$  字节。类似地，物理内存被分割为物理页，大小也为  $P$  字节。虚拟页面集合被分为三个不相交的子集：已缓存、未缓存和未分配。



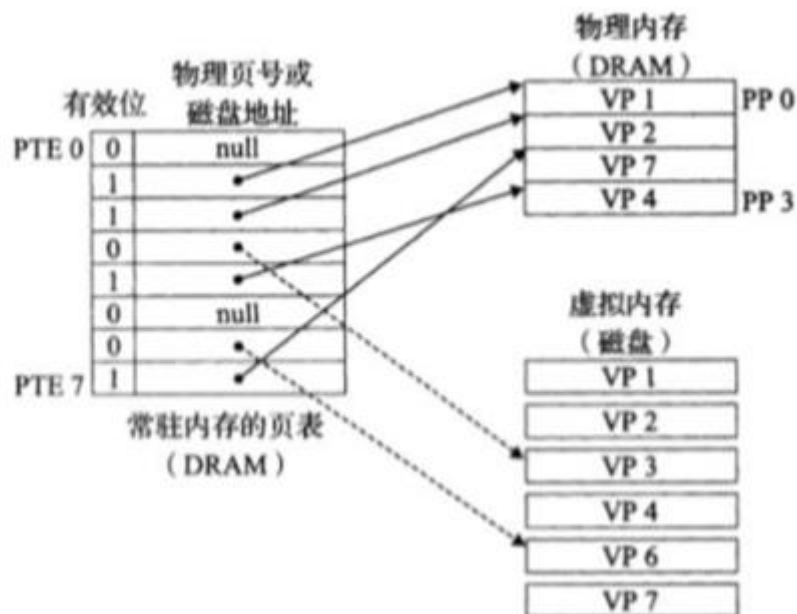
#### 7.3.1 虚拟页面

下图展示了页式管理中虚拟地址到物理地址的转换：



### 7.3.2 页式管理中虚拟地址到物理地址的转换

虚拟地址分为两部分：前一部分为虚拟页号，可以索引到当前进程的物理页表地址，后一部分为虚拟页偏移量，将来可以直接作为物理页偏移量，页表是一个存放在物理内存中的数据结构，页表将虚拟页映射到物理页。每次地址翻译硬件将一个虚拟地址转换为物理地址时，都会读取页表。



### 7.3.3 页表基本组织结构

展示了一个页表的基本组织结构。虚拟地址空间中的每个页在页表中一个固定偏移量的位置都有一个 PTE(页表条目)，而每个 PTE 是由一个有效位和一个

n 位的地址字段组成的。页表 PTE 分为三种情况：1. 已分配：PTE 有效位为 1 且地址部分不为 null，即页面已被分配，将一个虚拟地址映射到了一个对应的物理地址 2. 未缓冲：PTE 有效位为 0 且地址部分不为 null，即页面已经对应了一个虚拟地址，但虚拟内存内容还未缓存到物理内存中 3. 未分配：PTE 有效位为 0 且地址部分为 null，即页面还未分配，没有建立映射关系 现在根据图 7.4 介绍虚拟地址转换为物理地址的过程：首先根据虚拟页号在当前进程的物理页表中找到对应的页面，若符号位设置为 1，则表示命中，从页面中取出物理页号+虚拟页偏移量即组成了一个物理地址；否则表示不命中，产生一个缺页异常，需要从磁盘中读取相应的物理页到内存。

## 7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

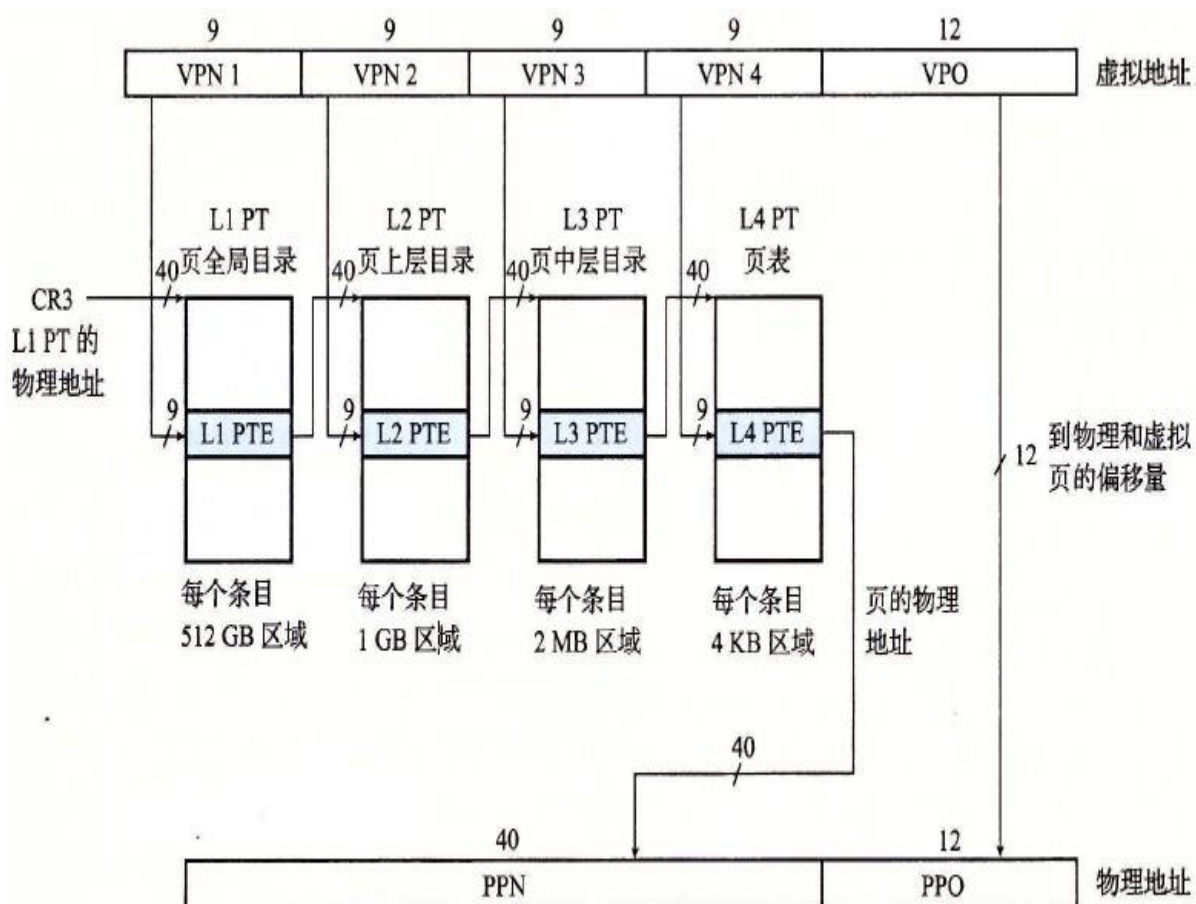


图 9-25 Core i7 页表翻译(PT: 页表, PTE: 页表条目, VPN: 虚拟页号, VPO: 虚拟页偏移, PPN: 物理页号, PPO: 物理页偏移量。图中还给出了这四级页表的 Linux 名字)

### 7.4.1 页表翻译

36 位 VPN 被划分成四个 9 位的片，每个片被用作到一个页表的偏移量。CR3

寄存器包含 L1 页表的物理地址。VPN 1 提供到一个 L1 PTE 的偏移量,这个 PTE 包含 L2 页表的基地址。VPN 2 提供到一个 L2 PTE 的偏移量,以此类推。

## 7.5 三级 Cache 支持下的物理内存访问

L1 Cache 是 8 路 64 组相联。块大小为 64B。解析前提条件: 因为共 64 组, 所以需要 6bit CI 进行组寻址, 因为共有 8 路, 因为块大小为 64B 所以需要 6bit CO 表示数据偏移位置, 因为 VA 共 52bit, 所以 CT 共 40bit。在上一步中我们已经获得了物理地址 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) 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。

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)。查询到数据之后, 一种简单的放置策略如下: 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。

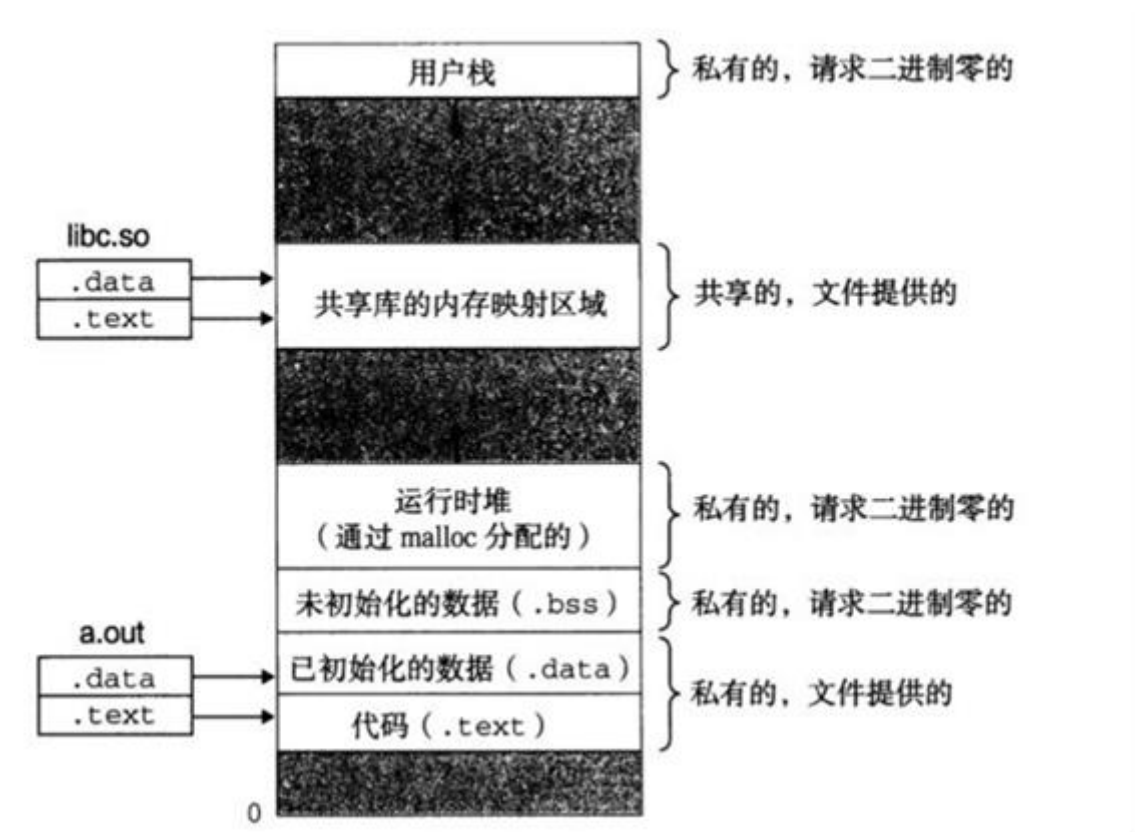
## 7.6 hello 进程 fork 时的内存映射

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



7.7.1 内存映像

## 7.8 缺页故障与缺页中断处理

1. 段错误：首先，先判断这个缺页的虚拟地址是否合法，那么遍历所有的合法区域结构，如果这个虚拟地址对所有的区域结构都无法匹配，那么就返回一个段错误（segment fault）

2. 非法访问：接着查看这个地址的权限，判断一下进程是否有读写改这个地址的权限。

3. 如果不是上面两种情况那就是正常缺页，那就选择一个页面牺牲然后换入新的页面并更新到页表。

## 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虚拟内存区域，称为堆。分配器将堆视为一组不同大小的块的集合来维护。每个块就是一个连续的虚拟内存片，要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可用来分配。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

配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

分配器分为两种基本风格：显式分配器、隐式分配器。

- 1.显式分配器：要求应用显式地释放任何已分配的块。
- 2.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。

其中涉及的组织结构知识还有显式空间链表，隐式空间链表，空间块的合并

## 7. 10 本章小结

本章介绍了本章主要介绍了 `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，还介绍了 `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。虽然 `hello` 很小，但无数个 `hello` 放在一起管理起来就变得非常棘手，计算机一定有条理清晰的存储和访问机制才能保证访存的速度。

**(第 7 章 2 分)**

## 第 8 章 hello 的 IO 管理

### 8.1 Linux 的 IO 设备管理方法

一个 Linux 文件就是一个  $m$  个字节的序列，所有的 I/O 设备（例如网络、磁盘和终端）都被模型化为文件，而所有的输入和输出都被当作对相应文件的读和写来执行。这个设备映射为文件的方式，允许 Linux 内核引出一个简单、低级的应用接口，称为 Unix I/O，这使得输入和输出都能以一种统一且一致的方式来执行。

### 8.2 简述 Unix IO 接口及其函数

1、打开文件 返回一个小的非负整数，即描述符。用描述符来标识文件。每个进程都有三个打开的文件：标准输入（0）、标准输出（1）、标准错误（2）

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
#include <fcntl.h>
```

```
int open(char *filename, int flags, mode_t mode);
```

//返回：若成功则为新文件描述符，若出错为-1

flags:进程打算如何访问文件

O\_RDONLY:只读      O\_WRONLY:只写      O\_RDWR:可读可写

也可以是一个或更多位掩码的或：

O\_CREAT:如文件不存在，则创建

O\_TRUNC: 如果文件已存在，则截断

O\_APPEND:每次写操作，设置 k 到文件结尾

mode:指定新文件的访问权限位

每个进程都有一个 `umask`，通过调用 `umask` 函数设置。所以文件的权限为被设置成 `mode & ~umask`

2、改变当前文件位置 从文件开头起始的字节偏移量。系统内核保持一个文件位置 `k`，对于每个打开的文件，起始值为 0。应用程序执行 `seek`，设置当前位置 `k`，通过调用 `lseek` 函数，显示地修改当前文件位置。

3、读写文件。读操作：从文件拷贝  $n$  个字节到存储器，从当前文件位置 `k` 开始，将 `k` 增加到 `k+n`，对于一个大小为  $m$  字节的文件，当  $k \geq m$  时，读操作触发



一个 EOF 的条件。写操作：从存储器拷贝  $n$  个字节到文件， $k$  更新为  $k+n$

```
#include <unistd.h>
```

```
ssize_t read(int fd, void *buf, size_t n);
```

//返回：若成功则为读的字节数，若 EOF 则为 0，若出错为-1.

```
ssize_t write(int fd, const void *buf, size_t n);
```

//返回：若成功则为写的字节数，若出错则为-1.

**read 函数：**从描述符为  $fd$  的当前文件位置拷贝至多  $n$  个字节到存储器位置  $buf$ 。  
返回-1 表示一个错误，返回 0 表示 EOF，否则返回实际读取的字节数。

**write 函数：**从存储器位置  $buf$  拷贝至多  $n$  个字节到描述符  $fd$  的当前文件位置。

Ps: `ssize_t` 与 `size_t` 区别： `size_t`: unsigned int, `ssize_t`: int。

4、关闭文件：内核释放文件打开时创建的数据结构，并恢复描述符到描述符池中，进程通过调用 `close` 函数关闭一个打开的文件。关闭一个已关闭的描述符会出错。

```
#include<unistd.h>
```

```
Int close(int fd);
```

//返回：若成功则为 0，若出错则为-1

### 8.3 printf 的实现分析

```
tatic int printf(const char *fmt, ...)
```

```
{
```

```
    va_list args;
```

```
    int i;
```

```
    va_start(args, fmt);
```

```
    write(1,printbuf,i=vsprintf(printbuf, fmt, args));
```

```
    va_end(args);
```

```
    return i;
```

```
}
```

其中 `*fmt` 是格式化用到的字符串，而后面省略的则是可变的形参，即 `printf(“%d”, i)` 中的 `i`，对应于字符串里面的缺省内容。

`va_start` 的作用是取到 `fmt` 中的第一个参数的地址，下面的 `write` 来自 Unix I/O，而其中的 `vsprintf` 则是用来格式化的函数。这个函数的返回值是要打印出的字符串的长度，也就是 `write` 函数中的 `i`。该函数会将 `printbuf` 根据 `fmt` 格式化字符和相应

的参数进行格式化，产生格式化的输出，从而 `write` 能够打印。

在 Linux 下，`write` 函数的第一个参数为 `fd`，也就是描述符，而 1 代表的就是标准输出。查看 `write` 函数的汇编实现可以发现，它首先给寄存器传递了几个参数，然后调用 `syscall` 结束。`write` 通过执行 `syscall` 指令实现了对系统服务的调用，从而使内核执行打印操作。

内核会通过字符显示子程序，根据传入的 ASCII 码到字模库读取字符对应的点阵，然后通过 `vram`（显存）对字符串进行输出。显示芯片将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），最终实现 `printf` 中字符串在屏幕上的输出。

## 8.4 getchar 的实现分析

```
int getchar(void)
{
    char c;
    return (read(0,&c,1)==1)?(unsigned char)c:EOF
}
}
```

可以看到，`getchar` 函数通过调用 `read` 函数返回字符。其中 `read` 函数的第一个参数是描述符 `fd`，0 代表标准输入。第二个参数输入内容的指针，这里也就是字符 `c` 的地址，最后一个参数是 1，代表读入一个字符，符号 `getchar` 函数读一个字符的设定。`read` 函数的返回值是读入的字符数，如果为 1 说明读入成功，那么直接返回字符，否则说明读到了 `buf` 的最后。

`read` 函数同样通过 `sys_call` 中断来调用内核中的系统函数。键盘中断处理子程序会接受按键扫描码并将其转换为 ASCII 码后保存在缓冲区。然后 `read` 函数调用的系统函数可以对缓冲区 ASCII 码进行读取，直到接受回车键返回。

这样，`getchar` 函数通过 `read` 函数返回字符，实现了读取一个字符的功能。

## 8.5 本章小结

本章我们就 `hello` 里面的函数对应 `unix` 的 I/O 来细致地分析了一下 I/O 对接口以及操作方法，这有助于我们以后在写函数的时候在标准 I/O 库没有的时候我们可以编写自己的 I/O 函数。

**（第 8 章 1 分）**

## 结论

1. `cpp` 预处理，处理以`#`开头的，得到 `hello.i`
2. 编译器将 `hello.i` 变成 `hello.s`
3. 汇编器将 `hello.s` 翻译成机器语言指令得到可重定位目标文件 `hello.o`
4. 链接器将 `hello.o` 与动态链接库链接生成可执行目标文件 `hello`，此时 `hello` 可以运行了
5. 运行：在 `shell` 中输入 `./hello 1170300901 侯欣宇`，`shell` 为 `hello` `fork` 一个子进程，并在子进程中调用 `execve`，加载运行 `hello`
6. `CPU` 为 `hello` 分配内存空间，`hello` 从磁盘被加载到内存。
7. 当 `CPU` 访问 `hello` 时，请求一个虚拟地址，`MMU` 把虚拟地址转换成物理地址并通过三级 `cache` 访存。
8. `Shell` 处理各种信号
9. `Unix I/O` 连接了输入输出硬件
10. 最后 `return 0`，安全结束

`Hello` 的一生如此精妙复杂，但展现给我们的只是屏幕输出 `hello` 等字符串的一瞬，不深入了解它（计算机系统），可能就丢生了一座宝库。

(结论 0 分，缺失 -1 分，根据内容酌情加分)

## 附件

列出所有的中间产物的文件名，并予以说明起作用。

(附件 0 分，缺失 -1 分)

7

<u>Hello.i</u> ↵	<u>Hello.c</u> 预处理后生成文本文件↵
<u>Hello.s</u> ↵	<u>Hello.i</u> 编译后汇编文件↵
<u>Hello.o</u> ↵	<u>Hello.s</u> 汇编后可重定位目标执行↵
Hello↵	链接之后的可执行目标文件↵
Hello.o.txt↵	反汇编代码↵
Hello,elf.txt↵	Hello 的 ELF 格式↵

## 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>.

(参考文献 0 分, 缺失 -1 分)