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　计算机科学与技术

学　　号　1170300928

班　　级　1703009

学　　生　杨新宇

指 导 教 师　史先俊

计算机科学与技术学院

2018 年 12 月

摘　要

本次实验主要围绕 hello.C 的整个人生经历来开展，其中主要包含以下几个内容，hello 的预处理，编译，hello.O 汇编文件的形成与解析，与其他文件链接形成可执行文件，hello 文件中所涉及到的进程管理，hello 的存储管理，与 hello 的 IO

关键词：hello.c 的形成可执行过程

（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：From Program to Process
使用 C 语言编写的到 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 i7-7700HQ x64CPU,16G RAM,128G SSD +1T HDD.

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
hello.objdmp	hello.o 的反汇编代码
hello.elf	hello.o 的 ELF 格式
hello1.objdump	hello 的反汇编代码
hello1.elf	hello 的 ELF 格式

1.4 本章小结

整个实验的总体概括，说明了 hello 的 P2P 和 O2O 过程，并列出了整个实验中的相关文件

（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。

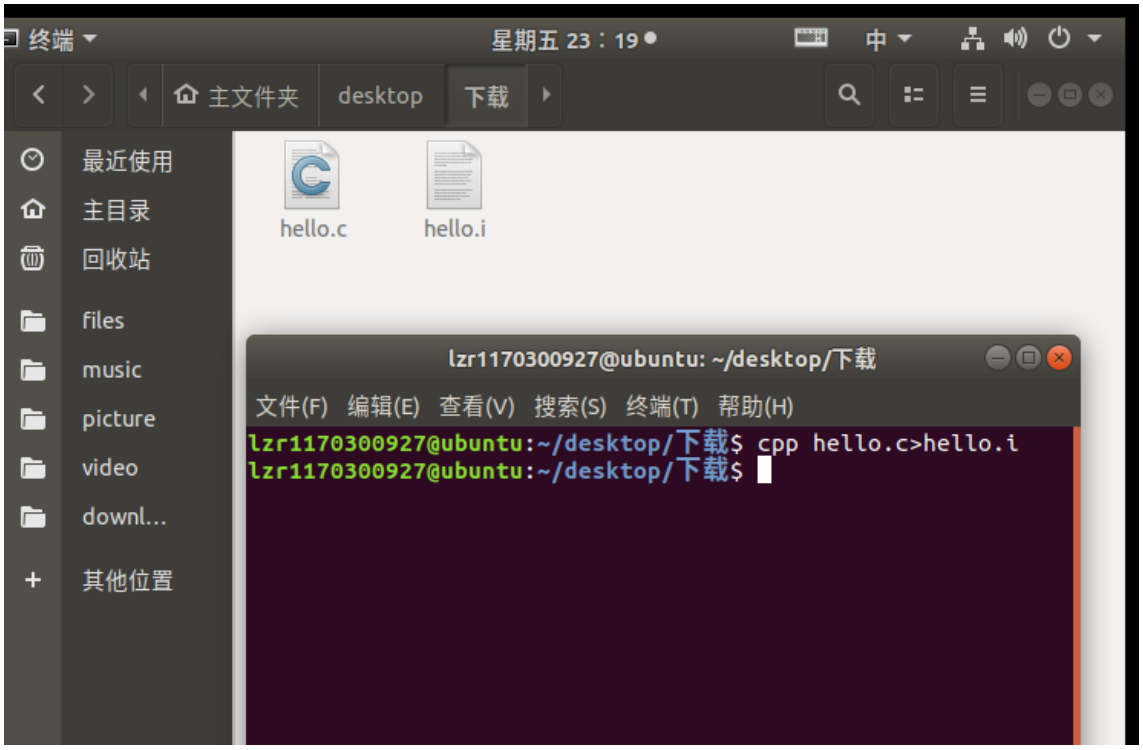
作用:

①加载头文件

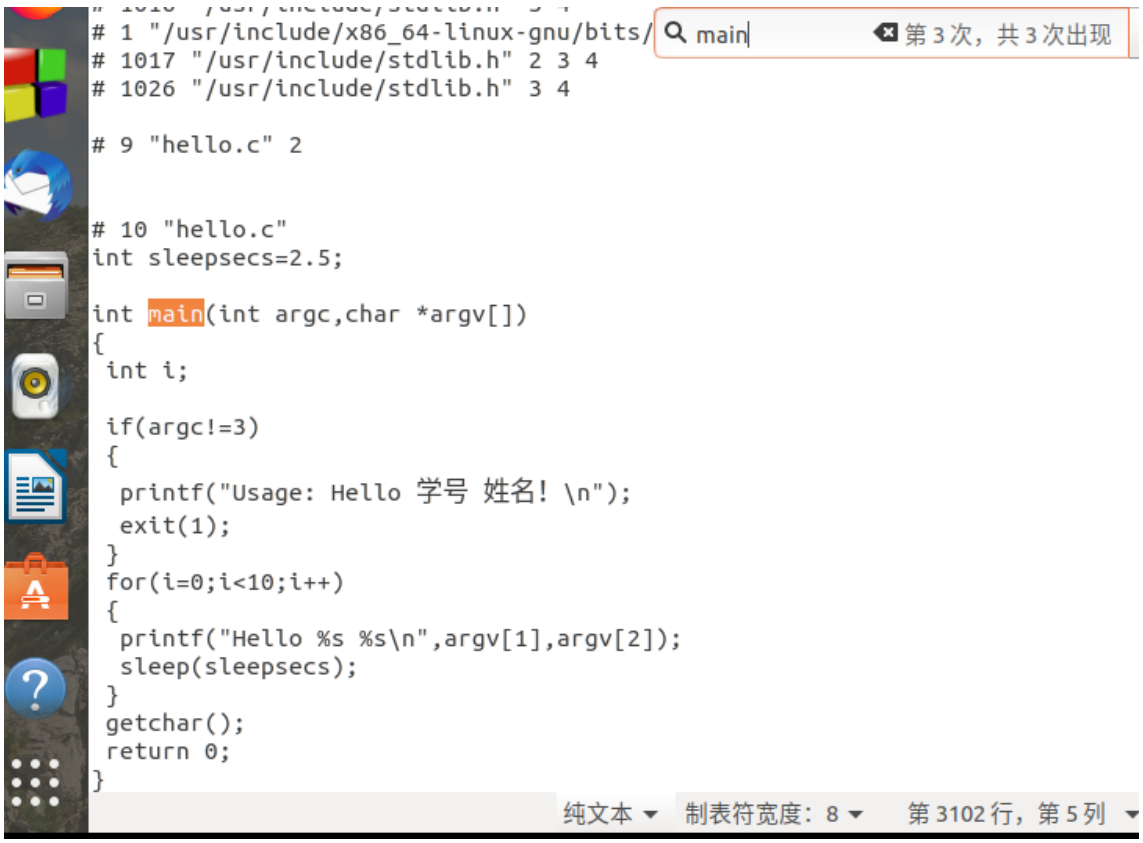
②进行宏替换

③条件编译

2.2 在 Ubuntu 下预处理的命令



2.3 Hello 的预处理结果解析



我们直接打开可以看到 hello.i 中的内容，可以发现，其已经把#后面的内容引入，我们在对应 3102 行发现了 main 主函数的位置

2.4 本章小结

Hello.C 变为 hello.i 的过程就是将各种宏引入到函数的内容中，这取决于你引入多少的头文件（各种宏），你也可以理解为我们的可爱的 hello.C 在拉上战场之前的第一步就是装填配件

（以下格式自行编排，编辑时删除）

（第 2 章 0.5 分）

第 3 章 编译

3.1 编译的概念与作用

编译

1、利用编译程序从源语言编写的源程序产生目标程序的过程。

2、用编译程序产生目标程序的动作。 编译就是把高级语言变成计算机可以识别的 2 进制语言，计算机只认识 1 和 0，编译程序把人们熟悉的语言换成 2 进制的。

编译程序把一个源程序翻译成目标程序的工作过程分为五个阶段：词法分析；语法分析；语义检查和中间代码生成；代码优化；目标代码生成。主要是进行词法分析和语法分析，又称为源程序分析，分析过程中发现有语法错误，给出提示信息。

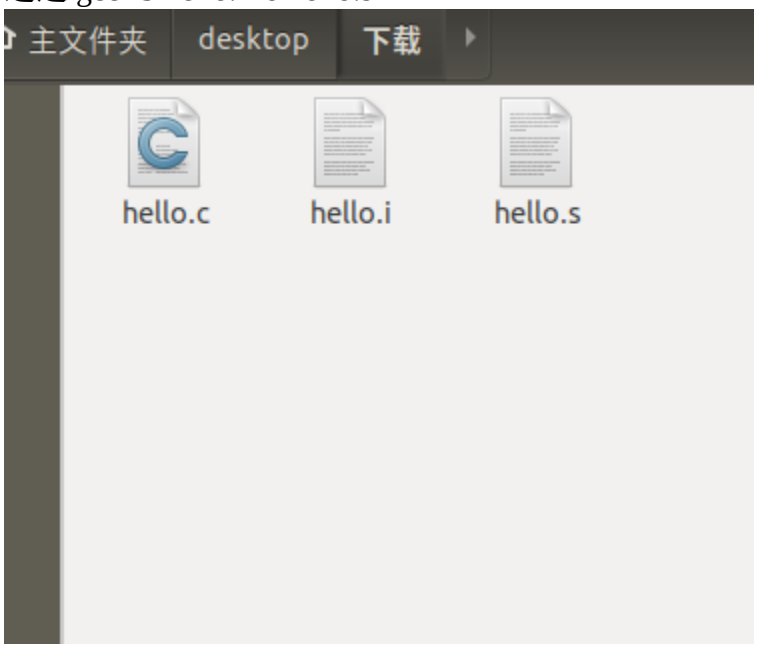
编译语言是一种以编译器来实现的编程语言。它不像直译语言一样，由解释器将代码一句一句运行，而是以编译器，先将代码编译为机器码，再加以运行。理论上，任何编程语言都可以是编译式，或直译式的。它们之间的区别，仅与程序的应用有关。

其中编译的主要过程为

1. 词法分析器，用于将字符串转化成内部的表示结构。
2. 语法分析器，将词法分析得到的标记流（token）生成一棵语法树。
3. 目标代码的生成，将语法树转化成目标代码。

1.2 在 Ubuntu 下编译的命令

通过 gcc -S hello.i -o hello.s



3.3 Hello 的编译结果解析

指令：（列出如下）

.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

数据：

（1） 整数：

（2） 字符串

（3） 数组

整数：

int sleepsecs：sleepsecs 在 C 程序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。

int i：编译器将局部变量存储在寄存器或者栈空间中，占 4 字节

int argc：作为第一个参数传入。

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。

字符串：

“Usage: Hello 学号 姓名！\n”

“Hello %s %s\n” （位于 printf 中）

数组

char *argv[] main

赋值与类型转换

这两点一起说的原因是他们两个操作出现在同一个操作中：

Int sleepsecs =2.5

（单股赋值操作还有个 i=0，因为同样可以概括到，这里不加讨论）

赋值操作在汇编语言中是以 mov 的形式来体现的，例如对 int 型进行的赋值操作，一般用 movl

而对于第一个赋值操作，还涉及到强制类型转换，将浮点型数据转换为 int 性数据

数据转换时采用值向 0 舍入原则，所以此时 2.5 会被舍入为 2

关系操作：

CMP S1,S2

比较-设置条件码

TEST S1,S2

测试-设置条件码

SET** D

按照**将条件码设置 D

涉及到的关系运算的操作为

1. argc!=3：判断 argc 不等于 3，汇编代码如下：（将 S2-S1 设置为条件码）

   cmpl $3,-20(%rbp)

2. i<10：判断 i 小于 10 汇编代码如下：

   cmpl $9,-4(%rbp)

关系操作后自带的为跳转操作，在此不加赘述

算数操作：

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。

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。

函数操作：

P 中调用函数 Q 包含以下动作：

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。

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。

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。

main 函数：

传递控制，main 函数因为被调用 call 才能执行 call 指令将下一条指令的地址压栈，然后跳转到 main 函数。

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。

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。

printf 函数：

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]。

控制传递：第一次 printf 因为只有一个字符串参数，所以 call

puts@PLT；第二次 printf 使用 call printf@PLT。

exit 函数：

传递数据：将%edi 设置为 1。

控制传递：call exit@PLT。

sleep 函数：

传递数据：将%edi 设置为 sleepsecs。

控制传递：call sleep@PLT。

getchar 函数：

控制传递：call gethcar@PLT

1.3 本章小结

涉及到的主要内容就是如何将 hello.i 映射到 hello.s
相关的操作主要是由汇编代码的形式呈现

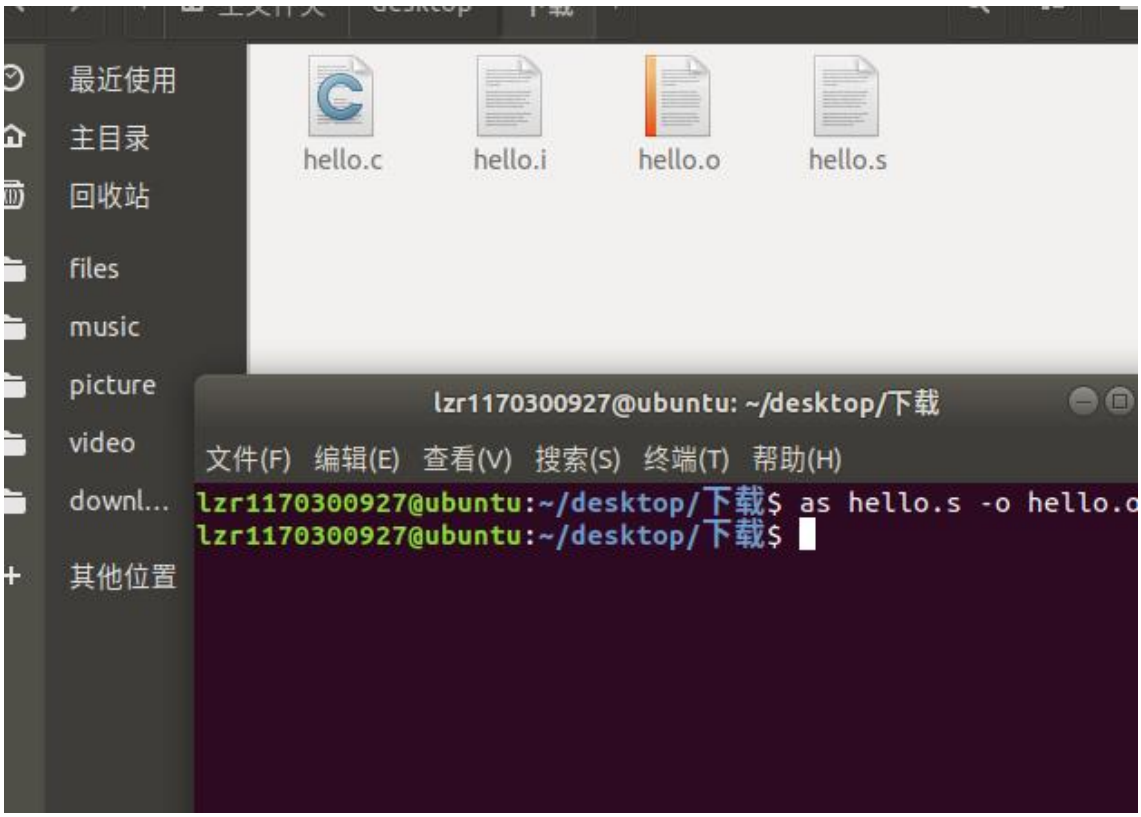
（第 3 章 2 分）

第 4 章 汇编

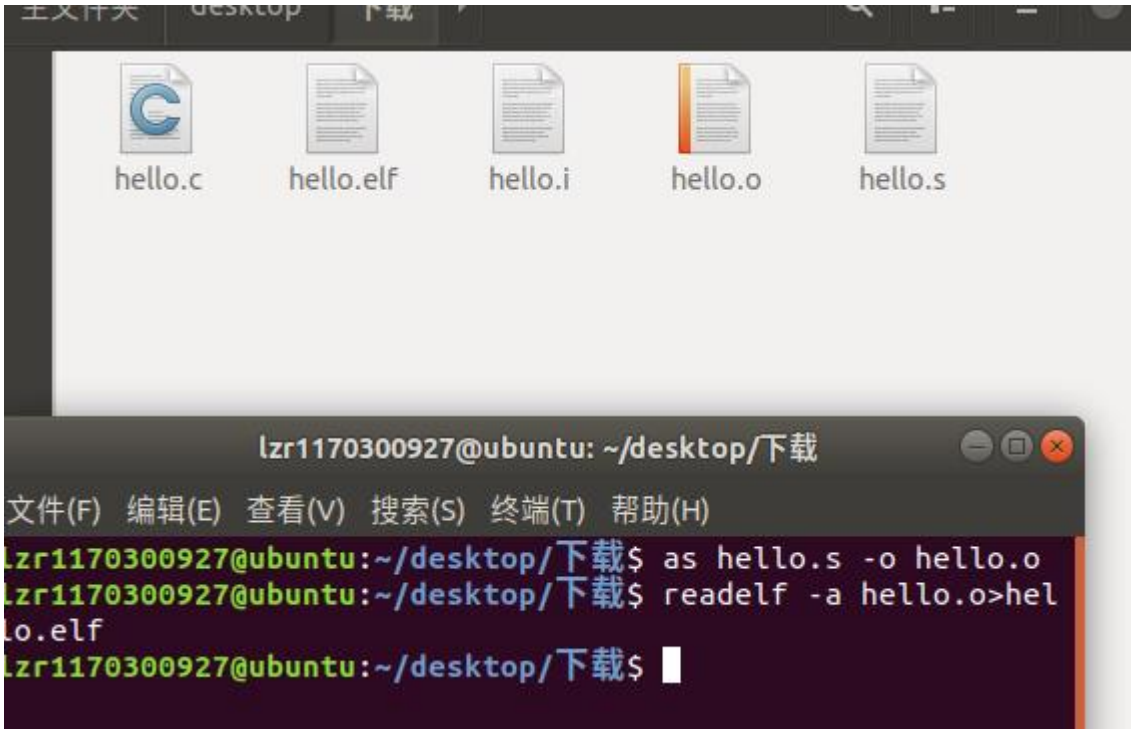
4.1 汇编的概念与作用

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



4.3 可重定位目标 elf 格式




```
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:          1152 (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
```

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	00000340
	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
	000000000000002b	0000000000000000	A 0	0 1
[6]	.comment	PROGBITS	0000000000000000	000000f3
	000000000000002b	0000000000000001	MS 0	0 1
[7]	.note.GNU-stack	PROGBITS	0000000000000000	0000011e
	0000000000000000	0000000000000000	0	0 1
[8]	.eh_frame	PROGBITS	0000000000000000	00000120
	0000000000000038	0000000000000000	A 0	0 8
[9]	.rela.eh_frame	RELA	0000000000000000	00000400
	0000000000000018	0000000000000018	I 10	8 8
[10]	.symtab	SYMTAB	0000000000000000	00000158
	0000000000000198	0000000000000018	11	9 8

There is no dynamic section in this file.

重定位节 '.rela.text' at offset 0x340 contains 8 entries:

偏移量	信息	类型	符号值	符号名称 + 加数
000000000018	000500000002	R_X86_64_PC32	0000000000000000	.rodata - 4
00000000001d	000c00000004	R_X86_64_PLT32	0000000000000000	puts - 4
000000000027	000d00000004	R_X86_64_PLT32	0000000000000000	exit - 4
000000000050	000500000002	R_X86_64_PC32	0000000000000000	.rodata + 1a
00000000005a	000e00000004	R_X86_64_PLT32	0000000000000000	printf - 4
000000000060	000900000002	R_X86_64_PC32	0000000000000000	sleepsecs - 4
000000000067	000f00000004	R_X86_64_PLT32	0000000000000000	sleep - 4
000000000076	001000000004	R_X86_64_PLT32	0000000000000000	getchar - 4

重定位节 '.rela.eh_frame' at offset 0x400 contains 1 entry:

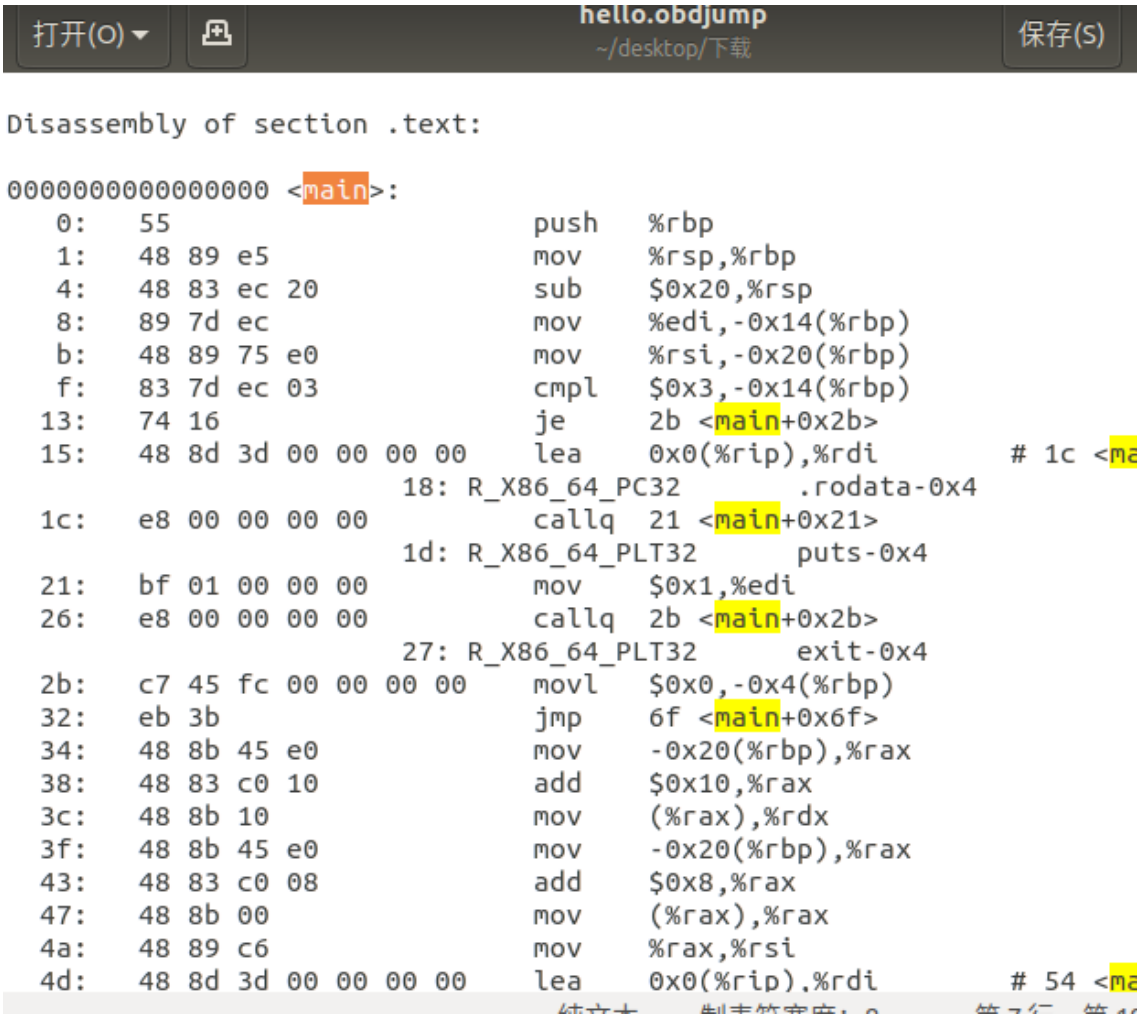
偏移量	信息	类型	符号值	符号名称 + 加数
000000000020	000200000002	R_X86_64_PC32	0000000000000000	.text + 0

The decoding of unwind sections for machine type Advanced Micro Devices X86-64

以上为包含的主要的部分

1. 节头部表
2. 重定位节

1.4 Hello.o 的结果解析



反汇编的代码如图所示

注意以下几点与原编代码的差别：

1. 汇编代码跳转指令的操作数使用是确定的地址。
2. 函数调用：在.s 文件中，函数调用之后直接跟着函数名称，而在反汇编程序中，call 的目标地址是当前下一条指令。（涉及到链接与重定位）
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。

4.5 本章小结

本章介绍了 hello 从 hello.s 到 hello.o 的汇编过程，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。

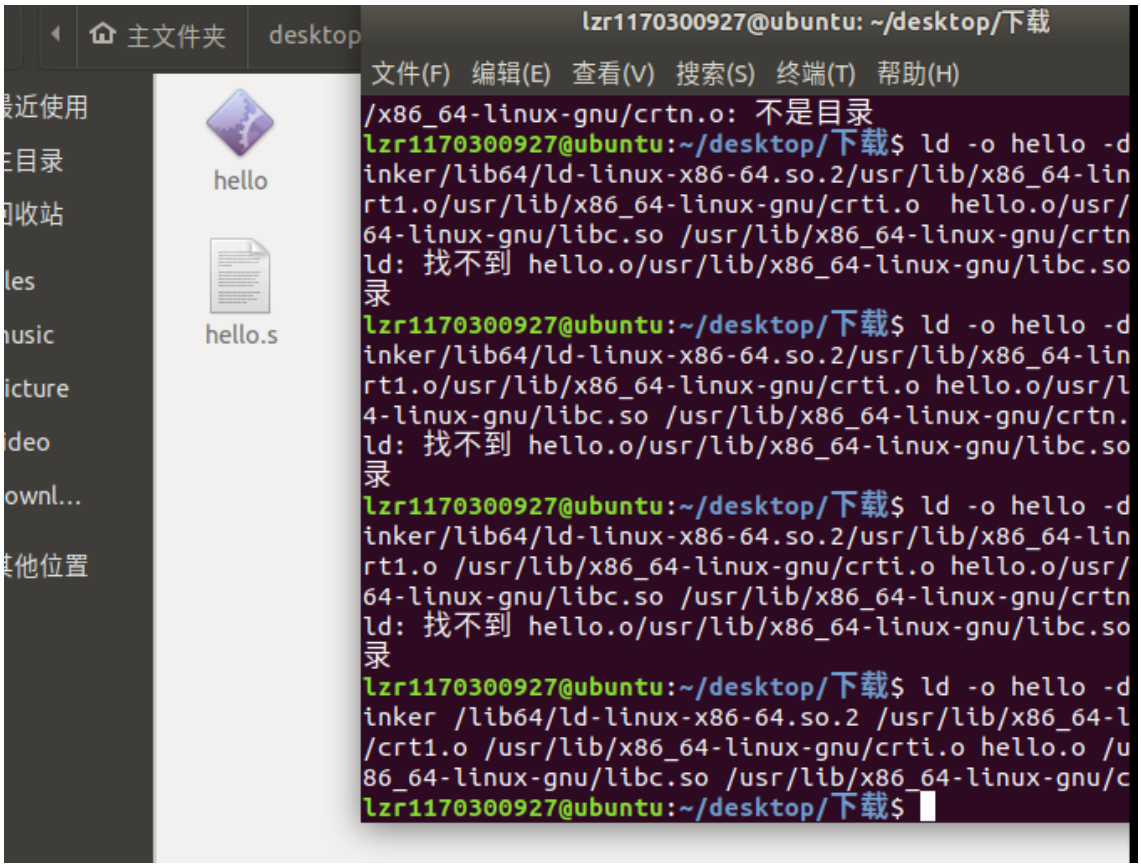
（第 4 章 1 分）

第 5 章 链接

5.1 链接的概念与作用

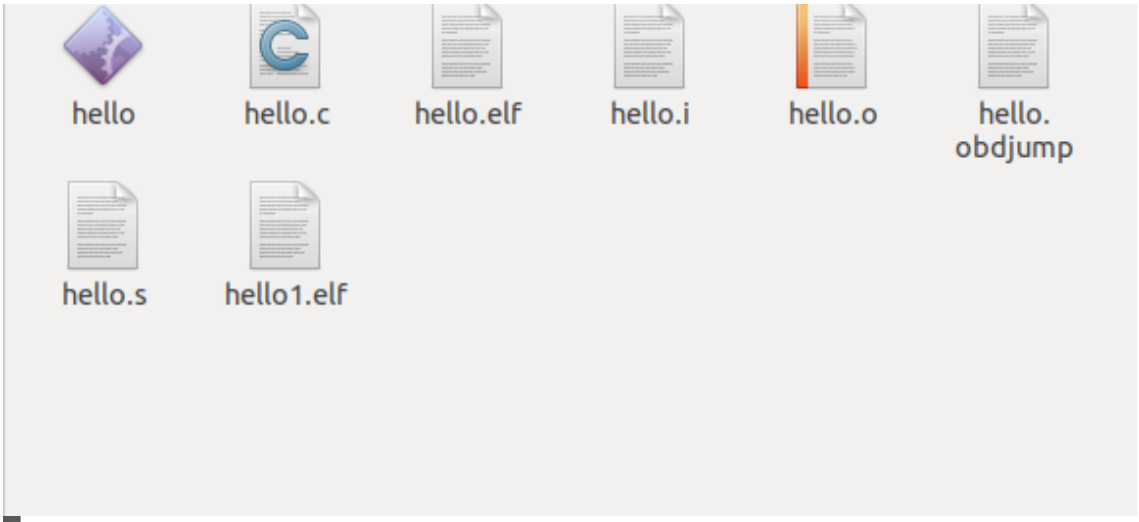
链接是指在电子计算机程序的各模块之间传递参数和控制命令，并把它们组成一个可执行的整体的过程。链接也称超级链接，是指从一个网页指向一个目标的连接关系，所指向的目标可以是另一个网页，也可以是相同网页上的不同位置，还可以是图片、电子邮件地址、文件、甚至是应用程序。

5.2 在 Ubuntu 下链接的命令



这个代码好长，嘤嘤嘤

5.3 可执行目标文件 hello 的格式



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

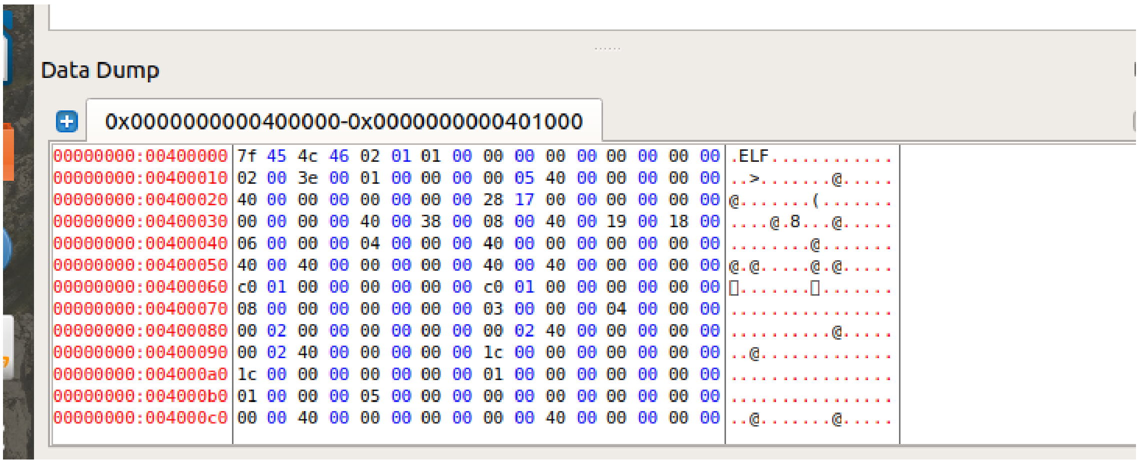
纯文本 ▾ 制表符宽度：8 ▾ 第 1 行，第 1 列 ▾

```
  [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
       000000000000002f  0000000000000000   A       0     0     4
  [16] .eh_frame         PROGBITS         0000000000400670  00000670
       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
```

纯文本 ▾ 制表符宽度：8 ▾ 第 1 行，第 1 列 ▾ 插入

Elf 节头部表的内容如上述所示

5.4 hello 的虚拟地址空间



查看 ELF 格式文件中的 Program Headers，程序头表在执行的时候被使用，它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：

PHDR 保存程序头表。

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）。
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。
DYNAMIC 保存了由动态链接器使用的信息。
NOTE 保存辅助信息。
GNU_STACK：权限标志，标志栈是否是可执行的。
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。

5.5 链接的重定位过程分析

```
hello:     文件格式 elf64-x86-64


Disassembly of section .init:

0000000000400488 <_init>:
  400488:       48 83 ec 08             sub    $0x8,%rsp
  40048c:       48 8b 05 65 0b 20 00    mov    0x200b65(%rip),%rax        #
600ff8 <__gmon_start__>
  400493:       48 85 c0                test   %rax,%rax
  400496:       74 02                   je     40049a <_init+0x12>
  400498:       ff d0                   callq  *%rax
  40049a:       48 83 c4 08             add    $0x8,%rsp
  40049e:       c3                      retq

Disassembly of section .plt:

00000000004004a0 <.plt>:
  4004a0:       ff 35 62 0b 20 00       pushq  0x200b62(%rip)        # 601008
<_GLOBAL_OFFSET_TABLE_+0x8>
  4004a6:       ff 25 64 0b 20 00       jmpq   *0x200b64(%rip)        # 601010
<_GLOBAL_OFFSET_TABLE_+0x10>
  4004ac:       0f 1f 40 00             nopl   0x0(%rax)

00000000004004b0 <puts@plt>:
  4004b0:       ff 25 62 0b 20 00       jmpq   *0x200b62(%rip)        # 601018
                                纯文本 ▾  制表符宽度：8 ▾     第 1 行，第 1 列  ▾  插入
```

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发现对外部函数调用的类型为 R_X86_64_PLT32 的重定位，此时动态链接库中的函数已经加入到了 PLT 中，.text 与.plt 节相对距离已经确定，

3）.rodata 引用：链接器解析重定条目时发现两个类型为 R_X86_64_PC32 的对.rodata 的重定位（printf 中的两个字符串），.rodata 与.text 节之间的相对距离确

定，因此链接器直接修改 call 之后的值为目标地址与下一条指令的地址之差，指向相应的字符串。这里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

相关的节名称：

.interp

.note.ABI-tag

.hash

.gnu.hash

.dynsym

.dynstr

.gnu.version

.gnu.version_r

.rela.dyn

.rela.plt

.init

.plt

.fini

.eh_frame

contains exception unwinding and source language information.

.dynamic

.got

.got.plt

.data

.comment

5.6 hello 的执行流程

. 用 edb 执行 hello，相关的主要函数如下

ld-2.27.so!_dl_start

ld-2.27.so!_dl_init

hello!_start

libc-2.27.so!__libc_start_main

-libc-2.27.so!__cxa_atexit

-libc-2.27.so!__libc_csu_init
hello!_init
libc-2.27.so!_setjmp
-libc-2.27.so!_sigsetjmp
--libc-2.27.so!__sigjmp_save
hello!main
hello!puts@plt
hello!exit@plt
*hello!printf@plt
*hello!sleep@plt
*hello!getchar@plt
ld-2.27.so!_dl_runtime_resolve_xsave
-ld-2.27.so!_dl_fixup
--ld-2.27.so!_dl_lookup_symbol_x
libc-2.27.so!exit

5.7 Hello 的动态链接分析

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

本章介绍了链接的概念和作用，分析了 hello 的 ELF 格式，（edb 让人头疼的使用方法），虚拟地址空间的分配，重定位和执行过程还有动态链接的过程。

（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。

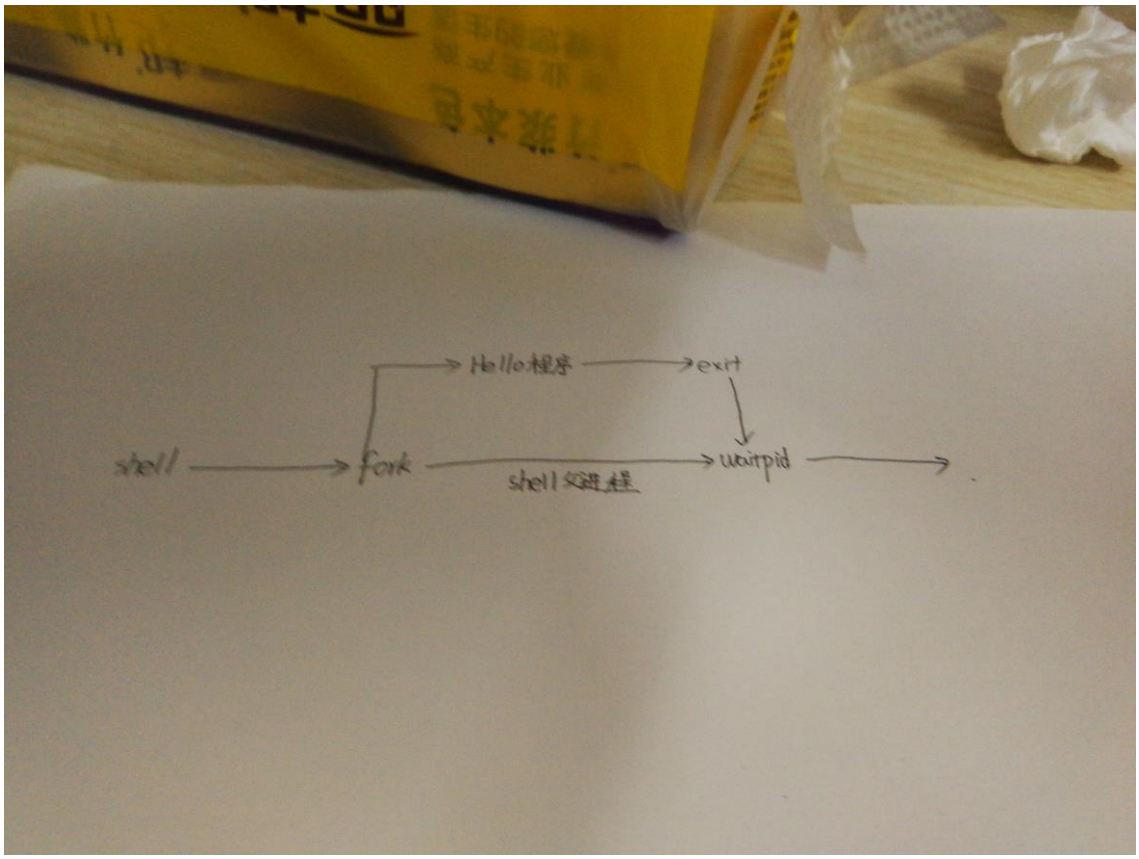
进程为用户提供了以下假象：我们的程序好像是系统中当前运行的唯一程序一样，我们的程序好像是独占的使用处理器和内存，处理器好像是无间断的执行我们程序中的指令，我们程序中的代码和数据好像是系统内存中唯一的对象。

6.2 简述壳 Shell-bash 的作用与处理流程

shell 是一个应用程序，他在操作系统中提供了一个用户与系统内核进行交互的界面。他的处理过程一般是这样的：

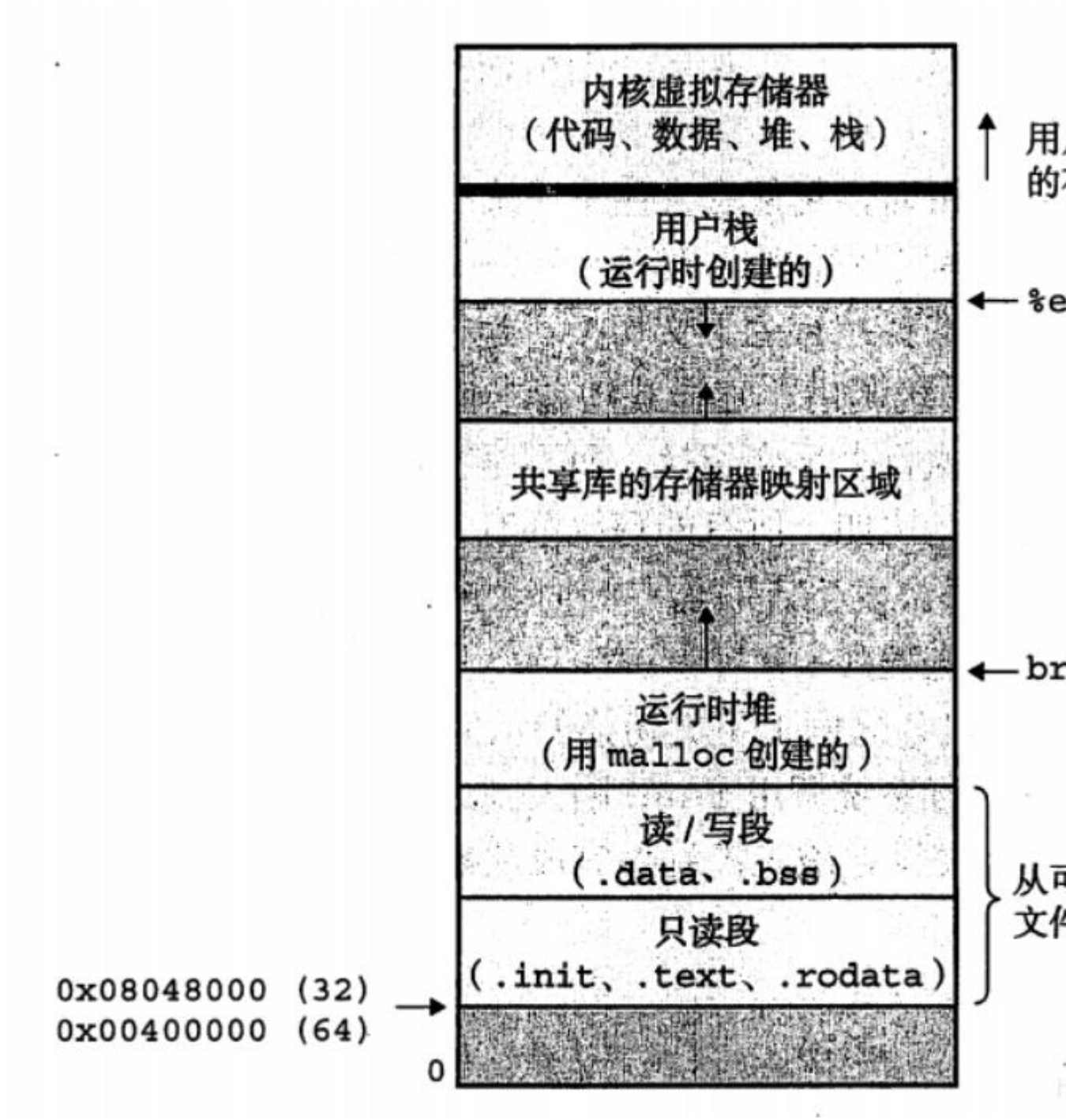
1. 读取用户的输入
2. 分析输入内容，获得输入参数
3. 如果是内核命令则直接执行，否则调用相应的程序执行命令
4. 在程序运行期间，shell 需要监视键盘的输入内容，并且做出相应的反应

6.3 Hello 的 fork 进程创建过程

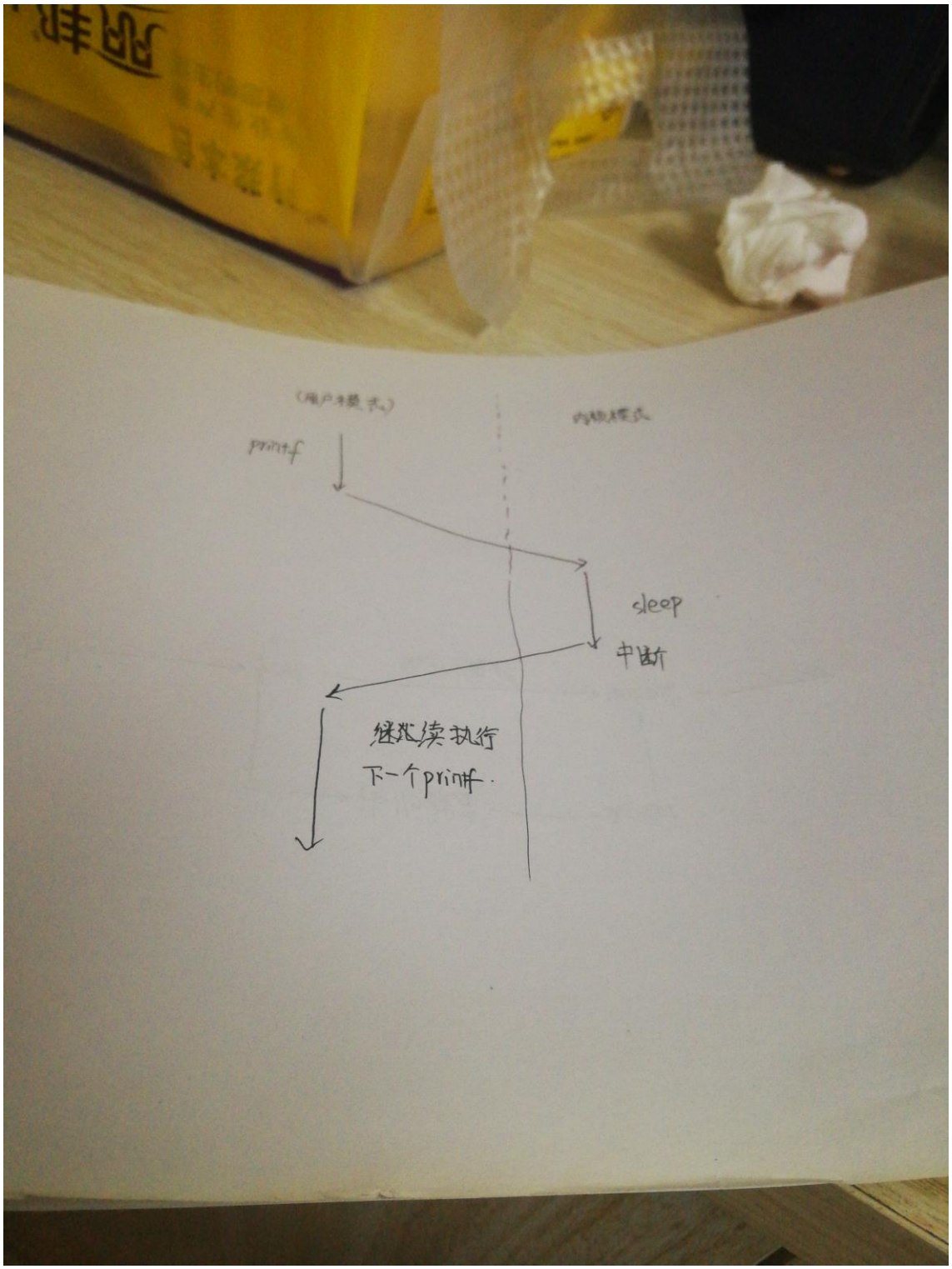


6.4 Hello 的 execve 过程

如下



6.5 Hello 的进程执行

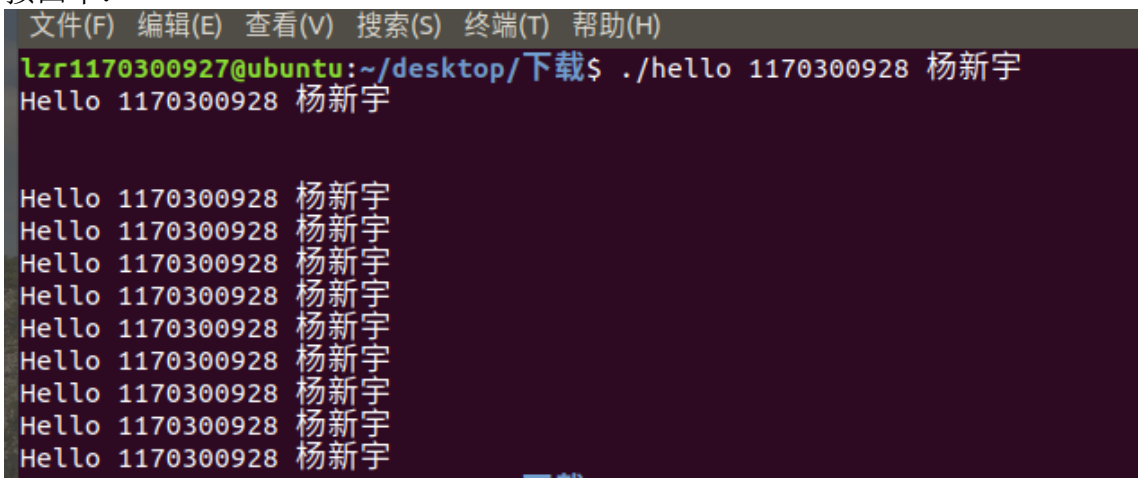


调用 getchar()时先是运行在前端 hello 进程中，然后调用时切换到内核进程的标准读取程序中，从键盘输入读取到一个字符之后再回到 hello 进程

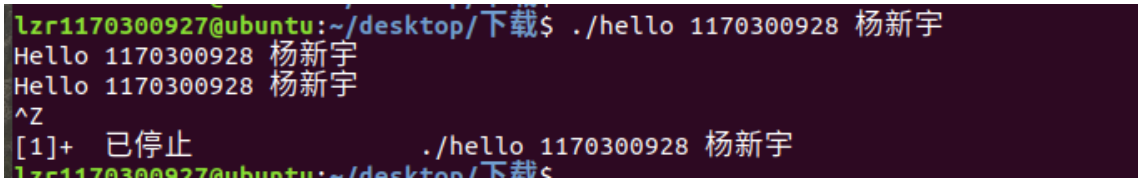
在内核和前端之前切换的动作被称为上下文切换。

6.6 hello 的异常与信号处理

按回车：



按 ctrl+z



按 ctrl+c

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

逻辑地址：又称相对地址，是程序运行由 CPU 产生的与段相关的偏移地址部分。他是描述一个程序运行段的地址。

物理地址：程序运行时加载到内存地址寄存器中的地址，内存单元的真正地址。他是在前端总线上传输的而且是唯一的。在 hello 程序中，他就表示了这个程序运行时的一条确切的指令在内存地址上的具体哪一块进行执行。

线性地址：这个和虚拟地址是同一个东西，是经过段机制转化之后用于描述程序分页信息的地址。他是对程序运行区块的一个抽象映射。以 hello 做例子的话，他就是一个描述： “我这个 hello 程序应该在内存的哪些块上运行。

7.2 Intel 逻辑地址到线性地址的变换-段式管理

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。

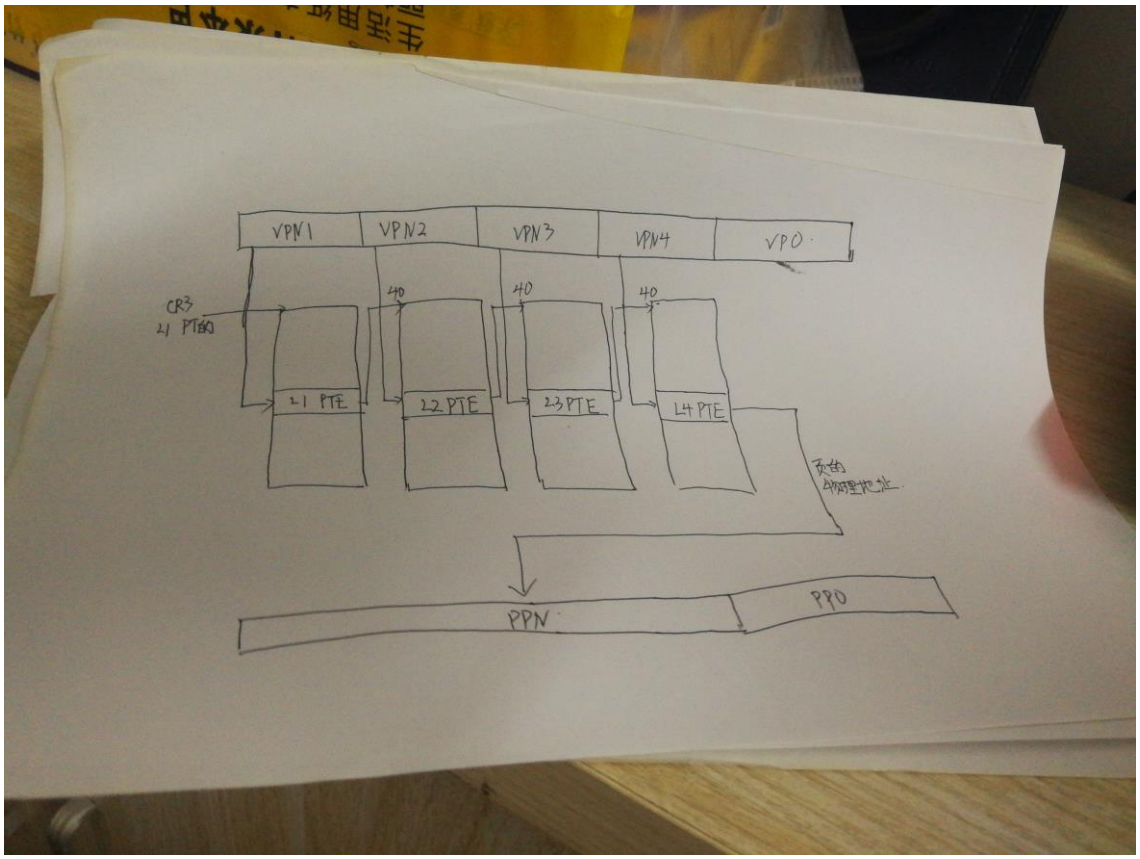
实模式，即不设防，也就是说逻辑地址=线性地址=实际的物理地址。段寄存器存放真实段基址，同时给出 32 位地址偏移量，则可以访问真实物理内存。

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

7.3 Hello 的线性地址到物理地址的变换-页式管理

在这个转换中要用到 TLB，首先我们先将线性地址分为 VPN+VPO 的形式，然后再将 VPN 拆分成 TLBT+TLBI 然后去 TLB 缓存里找所对应的 PPN（物理页号）如果发生缺页情况则直接查找对应的 PPN，找到 PPN 之后，将其与 VPO 组合变为 PPN+VPO 就是生成的物理地址了。

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换



7.5 三级 Cache 支持下的物理内存访问

得到物理地址之后，先将物理地址拆分成 CT（标记）+CI（索引）+CO（偏移量），然后在一级 cache 内部找，如果未能寻找到标记位为有效的字节（miss）的话就去二级和三级 cache 中寻找对应的字节，找到之后返回结果。

7.6 hello 进程 fork 时的内存映射

mm_struct（内存描述符）：描述了一个进程的整个虚拟内存空间
vm_area_struct（区域结构描述符）：描述了进程的虚拟内存空间的一个区间
在用 fork 创建虚拟内存的时候，要经历以下步骤：
创建当前进程的 mm_struct,vm_area_struct 和页表的原样副本
两个进程的每个页面都标记为只读页面
两个进程的每个 vm_area_struct 都标记为私有，这样就只能在写入时复制。

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：

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。

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。

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。

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

分为以下三种类型：

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。

（第 7 章 2 分）

第 8 章 hello 的 I0 管理

8.1 Linux 的 I0 设备管理方法

1.设备的模型化。在设备模型中，所有的设备都通过总线相连。每一个设备都是一个文件。设备模型展示了总线和它们所控制的设备之间的实际连接。在最底层，Linux 系统中的每个设备由一个 struct device 代表，而 Linux 统一设备模型就是在 kobject kset ktype 的基础之上逐层封装起来的。

2.设备管理则是通过 unix io 接口实现的。

8.2 简述 Unix I0 接口及其函数

Unix I/O 接口统一操作：打开文件。一个应用程序通过要求内核打开相应的文件，来宣告它想要访问一个 I/O 设备，内核返回一个小的非负整数，叫做描述符，它在后续对此文件的所有操作中标识这个文件，内核记录有关这个打开文件的所有信息。

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。

改变当前的文件位置：对于每个打开的文件，内核保持着一个文件位置 k，初始为 0，这个文件位置是从文件开头起始的字节偏移量，应用程序能够通过执行 seek，显式地将改变当前文件位置 k。

读写文件：一个读操作就是从文件复制 n>0 个字节到内存，从当前文件位置 k 开始，然后将 k 增加到 k+n，给定一个大小为 m 字节的而文件，当 k>=m 时，触发 EOF。类似一个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。

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：

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。open 函数将 filename 转换为一个文件描述符，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，mode 参数指定了新文件的访问权限位。

int close(fd)，fd 是需要关闭的文件的描述符，close 返回操作结果。

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。

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

```
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;

}
```

其中，vsprintf 的作用是将所有的参数内容格式化之后存入 buf 数组，然后返回格式化数组的长度。write 函数是将 buf 中的 i 个元素写到终端的函数。
Printf 的运行过程：

从 vsprintf 生成显示信息，显示信息传送到 write 系统函数，write 函数再陷阱-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

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

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

我终于完成我的大作业了 QAQ，整个过程真的很难受，花了将近整整一天的时间才完成，哎，下面对以上所有的过程进行一个总结

1. 编写 hello.C
2. 预处理：将相关的宏与外部的库展开合为 hello.i
3. 编译：hello.I 变为 hello.S
4. 汇编：生成 hello.o
5. 链接：将所有的 hello.o 与可重定位文件链接在一起，生成可执行程序文

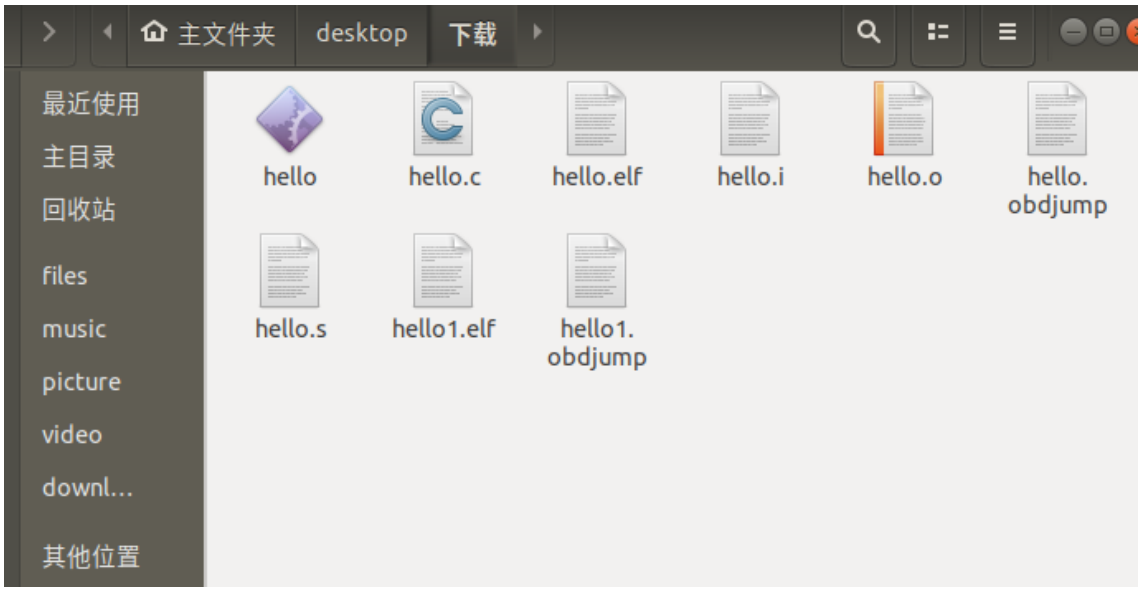
件 hello

6. 运行：输入./hello 1170300928 杨新宇
7. 用 fork 函数创建子进程
8. 调用 execve 函数，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，然后进入 main 函数。
9. CPU 为 hello 分配时间片
10. 用 MMU 将虚拟内存一映射到物理内存
11. 动态申请 malloc
12. 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. 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。



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
hello.objdmp	hello.o 的反汇编代码
hello.elf	hello.o 的 ELF 格式
hello1.objdump	hello 的反汇编代码
hello1.elf	hello 的 ELF 格式
hello.c	源文件

（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.

（参考文献 0 分，缺失 -1 分）