

哈爾濱工業大學

計算機系統

大作業

題 目	<u>程序人生-Hello's P2P</u>
專 業	<u>計算機科學與技術</u>
學 號	<u>1170300131</u>
班 級	<u>1703001</u>
學 生	<u>金宇航</u>
指 導 教 師	<u>史先俊</u>

計算機科學與技術學院
2018 年 12 月

摘 要

通过遍历 `hello.c` 在 Linux 下的生命周期，对其预处理、编译、汇编等过程进行分步解读和对比来学习各个过程在 Linux 下实现机制及原因。同时通过对 `hello` 在 Shell 中的动态链接、进程运行、内存管理、I/O 管理等过程的探索来继续理解 Linux 系统下的动态链接机制、存储层次结构、异常控制流、虚拟内存及 Unix I/O 等相关内容。

关键词：操作系统；编译；汇编；链接；内存；`hello`

（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	- 4 -
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 -
第 2 章 预处理	- 6 -
2.1 预处理的概念与作用	- 6 -
2.2 在 UBUNTU 下预处理的命令	- 6 -
2.3 HELLO 的预处理结果解析	- 6 -
2.4 本章小结	- 8 -
第 3 章 编译	- 9 -
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 -
第 4 章 汇编	- 15 -
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	- 15 -
4.4 HELLO.O 的结果解析	- 17 -
4.5 本章小结	- 23 -
第 5 章 链接	- 24 -
5.1 链接的概念与作用	- 24 -
5.2 在 UBUNTU 下链接的命令	- 24 -
5.3 可执行目标文件 HELLO 的格式	- 24 -
5.4 HELLO 的虚拟地址空间	- 26 -
5.5 链接的重定位过程分析	- 29 -
5.6 HELLO 的执行流程	- 30 -
5.7 HELLO 的动态链接分析	- 31 -
5.8 本章小结	- 32 -
第 6 章 HELLO 进程管理	- 33 -
6.1 进程的概念与作用	- 33 -

6.2 简述壳 SHELL-BASH 的作用与处理流程.....	- 33 -
6.3 HELLO 的 FORK 进程创建过程.....	- 33 -
6.4 HELLO 的 EXECVE 过程.....	- 33 -
6.5 HELLO 的进程执行.....	- 33 -
6.6 HELLO 的异常与信号处理.....	- 34 -
6.7 本章小结.....	- 35 -
第 7 章 HELLO 的存储管理.....	- 36 -
7.1 HELLO 的存储器地址空间.....	- 36 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理.....	- 36 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理.....	- 36 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换.....	- 36 -
7.5 三级 CACHE 支持下的物理内存访问.....	- 37 -
7.6 HELLO 进程 FORK 时的内存映射.....	- 37 -
7.7 HELLO 进程 EXECVE 时的内存映射.....	- 37 -
7.8 缺页故障与缺页中断处理.....	- 38 -
7.9 动态存储分配管理.....	- 38 -
7.10 本章小结.....	- 38 -
第 8 章 HELLO 的 IO 管理.....	- 40 -
8.1 LINUX 的 IO 设备管理方法.....	- 40 -
8.2 简述 UNIX IO 接口及其函数.....	- 40 -
8.3 PRINTF 的实现分析.....	- 41 -
8.4 GETCHAR 的实现分析.....	- 43 -
8.5 本章小结.....	- 43 -
结论.....	- 44 -
附件.....	- 45 -
参考文献.....	- 46 -

第 1 章 概述

1.1 Hello 简介

P2P:

编译器驱动程序读取 `hello.c`

->由预处理器根据以#开头的命令对其进行修改得到文本文件 (`hello.i`)

->通过编译器翻译 `hello.i` 得到汇编程序 (`hello.s`)

->汇编器将 `hello.s` 翻译成机器语言指令, 将指令打包成可重定位的 `hello.o`

->链接器合并获得可执行目标文件

->Linux 系统中通过内置命令行解释器 `shell` 加载运行 `hello` 程序, 为 `hello` 程序 `fork` 进程。

O2O:

(1)Shell 通过 `execve` 在 `fork` 产生的子进程中加载 `hello`, 先删除当前虚拟地址的用户部分已存在的数据结构, 为 `hello` 的代码、数据、`bss` 和栈区域创建新的区域结构。

(2)映射共享区域, 设置程序计数器, 指向代码区域的入口点, 进入 `main` 函数, CPU 为 `hello` 分配时间片执行逻辑控制流。`hello` 通过 Unix I/O 管理来控制输出。

(3)`hello` 执行完成后 `shell` 会回收 `hello` 进程, 并且内核会从系统中删除 `hello` 所有痕迹。

1.2 环境与工具

Intel Core i7-6700HQ x64 CPU; 2.5GHz; 8G RAM;

Windows10 64 位操作系统; vmware 14; Ubuntu 16.04 LTS;

`gcc`; `edb`; `readelf`; `objdump`; `gedit`; `CodeBlocks` 17.12

1.3 中间结果

<code>hello.i</code>	预处理后得到的文本文件
<code>hello.s</code>	编译后得到的汇编文件
<code>hello.o</code>	汇编后得到的可重定位目标执行文件
<code>hello.elf</code>	<code>hello.o</code> 的 ELF 格式
<code>hello</code>	链接后得到的可执行目标文件
<code>hello1.elf</code>	<code>hello</code> 的 ELF 格式

1.4 本章小结

hello.c 被编写出来、被编译成可执行文件、在系统的操作下被执行、最后被回收是这个程序的普通却不平凡的一生。是每一个程序员都要深入了解并参与的一生。这一章带领我们走过了这个历程、让我们对这部分知识有了系统的了解，同时让我们列出了这个实验的基本信息。

(第 1 章 0.5 分)

第 2 章 预处理

2.1 预处理的概念与作用

预处理的概念：调用预处理器根据以#符号开头的命令修改原始的 C 程序，将引用的所有库展开合并成为一个完整的文本文件。

预处理的作用：

将源文件中用#include 形式声明的文件复制到新的程序中，执行文件包含；

删除注释（//和/**/）；

替换宏定义；

条件编译（根据#if 后面的条件决定需要编译的代码）。

2.2 在 Ubuntu 下预处理的命令

指令为：gcc -E -o hello.i hello.c，在终端输入即可。

```
jyh1170300131@ubuntu: ~  
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)  
jyh1170300131@ubuntu:~$ gcc -E -o hello.i hello.c  
jyh1170300131@ubuntu:~$
```

2.3 Hello 的预处理结果解析

```
// 大作业的 hello.c 程序  
// gcc -m64 -no-pie -fno-PIC hello.c -o hello  
// 程序运行过程中可以按键盘，如不停乱按，包括回车，ctrl-Z, ctrl-C等。  
// 可以运行 ps jch  
#include <stdio.h>  
#include <unistd.h>  
#include <stdlib.h>  
  
int sleepsecs=2.5;  
  
int main(int argc, char *argv[])  
{  
    int i;  
}
```

hello.c

注释被删除

头文件被插入到了 hello.i

```

extern int getsubopt (char **__restrict __optionp,
                    char *const *__restrict __tokens,
                    char **__restrict __valuep)
    __attribute__((__nothrow__, __leaf__)) __attribute__((__nonnull__(1,
2, 3))) ;
# 1006 "/usr/include/stdlib.h" 3 4
extern int getloadavg (double __loadavg[], int __nelem)
    __attribute__((__nothrow__, __leaf__)) __attribute__((__nonnull__(1)));
# 1016 "/usr/include/stdlib.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/stdlib-float.h" 1 3 4
# 1017 "/usr/include/stdlib.h" 2 3 4
# 1026 "/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)
    {

```

hello.i

通过 hello.c 与 hello.i 中的内容对比，可以看出，通过操作，hello.c 中的注释被删除掉了，同时头文件被插入到了 hello.i 中。

```

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

# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
# 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 424 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
# 427 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 428 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/long-double.h" 1 3 4
# 429 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 425 "/usr/include/features.h" 2 3 4
# 448 "/usr/include/features.h" 3 4

```

hello.i 中，首先是对文件包含中系统头文件的寻址和解析。


```
typedef unsigned int __uid_t;
typedef unsigned int __gid_t;
typedef unsigned long int __ino_t;
typedef unsigned long int __ino64_t;
typedef unsigned int __mode_t;
typedef unsigned long int __nlink_t;
typedef long int __off_t;
typedef long int __off64_t;
typedef int __pid_t;
typedef struct { int __val[2]; } __fsid_t;
typedef long int __clock_t;
typedef unsigned long int __rlim_t;
typedef unsigned long int __rlim64_t;
typedef unsigned int __id_t;
typedef long int __time_t;
typedef unsigned int __useconds_t;
typedef long int __suseconds_t;

typedef int __daddr_t;
typedef int __key_t;

typedef int __clockid_t;

typedef void * __timer_t;
```

因为 hello.c 包含的头文件中还有其他头文件，系统递归的寻址、展开。同时引入头文件中所有 typedef 关键字，结构体类型、枚举类型、通过 extern 关键字调用并声明外部的结构体及函数定义。

2.4 本章小结

这一章主要对预处理进行了学习，了解了预处理的概念与作用、Ubuntu 下预处理的命令并对 hello 的预处理结果进行了解析。

(第 2 章 0.5 分)

第 3 章 编译

3.1 编译的概念与作用

概念：

编译主要包含五个阶段：词法分析、语法分析、语义检查、中间代码生成、目标代码生成。是利用编译器（ccl）将文本文件（hello.i）翻译成汇编程序（hello.s）的过程。

作用：

把预处理完的文件进行一系列词法分析，语法分析，语义分析及优化后生成相应的汇编代码文件。

3.2 在 Ubuntu 下编译的命令

指令为：gcc -S -o hello.s hello.i，在终端输入即可



```
jyh1170300131@ubuntu: ~  
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)  
jyh1170300131@ubuntu:~$ gcc -S -o hello.s hello.i  
jyh1170300131@ubuntu:~$
```

3.3 Hello 的编译结果解析

3.3.1 数据

1.sleepsecs: int 型全局变量，赋值时强制类型转换。

```
.file "hello.c"  
.text  
.globl sleepsecs  
.data  
.align 4  
.type sleepsecs, @object  
.size sleepsecs, 4  
sleepsecs:  
.long 2  
.section .rodata  
.align 8
```

2. 字符串

有两个字符串存在于只读数据段中，分别为 “Usage:Hello 1170300131 金宇航！” 和 “Hello %s %s\n”

```
.LC0:
    .string "Usage: Hello 1170300131
\351\207\221\345\256\207\350\210\252\357\274\201"
.LC1:
    .string "Hello %s %s\n"
```

3.i int 型局部变量，存放于-4（%rbp）

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

4. 数组 argv[] 首地址是-32（%rbp）

```
movq    -32(%rbp), %rax
addq    $16, %rax
movq    (%rax), %rdx
```

首地址

5. 其他数 立即数

3.3.2 赋值

1. 全局变量 `int sleepsecs = 2.5`（.data）
2. 局部变量 `int i = 0`

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

3.3.3 类型转换

1. 全局变量 `sleepsecs` 原本为整型，将 2.5 赋值的时候强制类型转换为 2。

3.3.4 算术操作

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

for 中的 i++

常用的整数算数操作（在网上找的图）

指令	效果	描述
<code>leaq S, D</code>	$D \leftarrow \&S$	加载有效地址
<code>INC D</code>	$D \leftarrow D + 1$	加1
<code>DEC D</code>	$D \leftarrow D - 1$	减1
<code>NEG D</code>	$D \leftarrow -D$	取负
<code>NOT D</code>	$D \leftarrow \sim D$	取补
<code>ADD S, D</code>	$D \leftarrow D + S$	加
<code>SUB S, D</code>	$D \leftarrow D - S$	减
<code>IMUL S, D</code>	$D \leftarrow D * S$	乘
<code>XOR S, D</code>	$D \leftarrow D \wedge S$	异或
<code>OR S, D</code>	$D \leftarrow D S$	或
<code>AND S, D</code>	$D \leftarrow D \& S$	与
<code>SAL k, D</code>	$D \leftarrow D \ll k$	左移
<code>SHL k, D</code>	$D \leftarrow D \ll k$	左移 (等同于SAL)
<code>SAR k, D</code>	$D \leftarrow D \gg_A k$	算术右移
<code>SHR k, D</code>	$D \leftarrow D \gg_L k$	逻辑右移

3.3.5 关系操作

1. `argc != 3`

```

.LFB5:
    .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
    leaq    .LC0(%rip), %rdi
    call    puts@PLT
    movl    $1, %edi
    call    exit@PLT

```

`argc!=3`

2. `i<10`

```

.L3:
    cmpl    $9, -4(%rbp)
    jle     .L4
    call    getchar@PLT
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc

```

`i<10`

3.3.6 结构操作

```

.L4:
    movq    -32(%rbp), %rax
    addq    $16, %rax
    movq    (%rax), %rdx
    movq    -32(%rbp), %rax
    addq    $8, %rax
    movq    (%rax), %rax
    movq    %rax, %rsi
    leaq    .LC1(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    movl    sleepsecs(%rip), %eax
    movl    %eax, %edi
    call    sleep@PLT
    addl    $1, -4(%rbp)

```

循环体中取数组元素值

其中-32(%rbp)存放的是 argv 首地址,+8 得到 argv[[1]]地址、+16 得到 argv[[2]]地址。

3.3.7 控制转移

1.if(argc!=3)

```

.LFB5:
    .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
    leaq    .LC0(%rip), %rdi
    call    puts@PLT
    movl    $1, %edi
    call    exit@PLT

```

如果 argc==3, 则函数继续进行, 执行下面的语句, 反之跳转至 L2。

3. for(i=0;i<10;i++)

```

.L3:
    cmpl    $9, -4(%rbp)
    jle     .L4
    call    getchar@PLT
    movl    $0, %eax
    leave   .cfi_def_cfa 7, 8
    ret
    .cfi_endproc

```

如果不满足条件则跳转至 L4。

3.3.8 函数操作

函数调用：参数入栈（将参数从右向左依次压入系统栈）、返回地址入栈、代码区跳转（处理器从当前代码区跳转到被调用函数的入口）、栈帧调整（保存

当前栈帧状态值、将当前栈帧切换到新栈帧、给新栈帧分配空间）。

函数返回：保存返回值、弹出当前帧，恢复上一个栈帧（在堆栈平衡的基础上，给 ESP 加上栈帧的大小，降低栈顶，回收当前栈帧的空间、将当前栈帧底部保存的前栈帧 EBP 值弹入 EBP 寄存器，恢复出上一个栈帧、将函数返回地址弹给 EIP 寄存器、跳转。

1. main

被系统启动函数 `_libc_start_main` 调用

2. Printf

(1) `printf("Usage: Hello 1170300131 金宇航! \n")`

.LFB5:

```
.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
leaq    .LC0(%rip), %rdi
call    puts@PLT
movl    $1, %edi
call    exit@PLT
```

(2) `printf("Hello %s %s\n" ,argv[1],argv[2])`

.L4:

```
movq    -32(%rbp), %rax
addq    $16, %rax
movq    (%rax), %rdx
movq    -32(%rbp), %rax
addq    $8, %rax
movq    (%rax), %rax
movq    %rax, %rsi
leaq    .LC1(%rip), %rdi
movl    $0, %eax
call    printf@PLT
movl    sleepsecs(%rip), %eax
movl    %eax, %edi
call    sleep@PLT
addl    $1, -4(%rbp)
```

3. exit

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

4.sleep

```
    movl    sleepsecs(%rip), %eax
    movl    %eax, %edi
    call    sleep@PLT
```

5.getchar

```
    jle     .L4
    call    getchar@PLT
    movl    $0, %eax
```

3.4 本章小结

这一章我们了解了编译的概念，作用和命令。并通过 hello.i 的编译结果 hello.s，就 C 语言中的数据与操作如何被翻译成汇编语言进行了学习。

(第3章2分)

第 4 章 汇编

4.1 汇编的概念与作用

概念：

把汇编语言翻译成机器语言的过程。

作用：

汇编器（as）将 hello.s 翻译成机器语言指令，并把这些指令打包成一种叫做可重定位目标程序的格式，并将结果保存在二进制目标文件 hello.o 中。

4.2 在 Ubuntu 下汇编的命令

```
jyh1170300131@ubuntu:~$ gcc -c -o hello.o hello.s
jyh1170300131@ubuntu:~$
```

4.3 可重定位目标 elf 格式

先生成 hello.o 的 elf 格式

```
jyh1170300131@ubuntu:~$ readelf -a hello.o > hello.elf
jyh1170300131@ubuntu:~$ gedit hello.elf
```

(1) ELF 头

以一个 16 字节的序列开始，描述生成该文件的系统的字的大小和字节顺序。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、节头部表的文件偏移，以及节头部表中条目的大小和数量。不同节的位置和大小是由节头部表描述的。


```

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
Ubuntu 软件 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

```

(2) 节头部表

记录各节名称、类型、地址、偏移量、大小、全体大小、旗标、连接、信息、对齐信息。

大小	全体大小	旗标	链接	信息	对齐
[0]	NULL	0000000000000000	0	0	0
[1] .text	PROGBITS	0000000000000000	0	0	00000040
[2] .rela.text	RELA	0000000000000000	0	0	000000340
[3] .data	PROGBITS	0000000000000000	I 10	1	8
[4] .bss	NOBITS	0000000000000000	WA 0	0	4
[5] .rodata	PROGBITS	0000000000000000	WA 0	0	000000c8
[6] .comment	PROGBITS	0000000000000000	A 0	0	8
[7] .note.GNU-stack	PROGBITS	0000000000000000	MS 0	0	000000fa
[8] .eh_frame	PROGBITS	0000000000000000	0	0	1
[9] .rela.eh_frame	RELA	0000000000000000	A 0	0	8
[10] .symtab	SYMTAB	0000000000000000	I 10	8	8
[11] .strtab	STRTAB	0000000000000000	11	9	8
[12] .shstrtab	STRTAB	0000000000000000	0	0	000002f0
			0	0	1

(3) 重定位节

包含 main 函数调用的 puts、exit、printf、sleep、getchar 函数以及全局变量 sleepsecs, 还有 .rodata 节的偏移量、信息、类型、符号值、符号名称及加数。

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 + 21
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

4.4 Hello.o 的结果解析

利用 `objdump -d -r hello.o > hello_o_asm.txt` 生成 hello.o 对应的反汇编文件，并与 hello.s 比较

```
jyh1170300131@ubuntu:~$ objdump -d -r hello.o > hello_o_asm.txt
jyh1170300131@ubuntu:~$
```

4.4.1 文件内容

```
hello.o:      文件格式 elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
0:  55                push    %rbp
1:  48 89 e5          mov     %rsp,%rbp
4:  48 83 ec 20       sub     $0x20,%rsp
8:  89 7d ec          mov     %edi,-0x14(%rbp)
b:  48 89 75 e0       mov     %rsi,-0x20(%rbp)
f:  83 7d ec 03       cmpl    $0x3,-0x14(%rbp)
13: 74 16            je      2b <main+0x2b>
15: 48 8d 3d 00 00 00 00 lea     0x0(%rip),%rdi      # 1c <main+0x1c>
18: R_X86_64_PC32    .rodata-0x4
1c: e8 00 00 00 00    callq  21 <main+0x21>
1d: R_X86_64_PLT32    puts-0x4
21: bf 01 00 00 00    mov     $0x1,%edi
26: e8 00 00 00 00    callq  2b <main+0x2b>
27: R_X86_64_PLT32    exit-0x4
2b: c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%rbp)
2e: eb 3b            jmp     6f <main+0x6f>
34: 48 8b 45 e0       mov     -0x20(%rbp),%rax
38: 48 83 c0 10       add     $0x10,%rax
3c: 48 8b 10          mov     (%rax),%rdx
3f: 48 8b 45 e0       mov     -0x20(%rbp),%rax
```

```

.file "hello.c"
.text
.globl sleepsecs
.data
.align 4
.type sleepsecs, @object
.size sleepsecs, 4
sleepsecs:
    .long 2
    .section .rodata
    .align 8
.LC0:
    .string "Usage: Hello 1170300131|
\351\207\221\345\256\207\350\210\252\357\274\201"
.LC1:
    .string "Hello %s %s\n"
    .text
    .globl main
    .type main, @function
main:
.LFB5:
    .cfi_startproc
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq %rsp, %rbp

```

hello_o_asm.txt 中只有对文件最简单的描述，记录了文件格式和.text 代码段；而 hello.s 中有对文件的描述，全局变量的完整描述和.rodata 只读数据段。

他们都有 main 函数的汇编代码，但是 hello.s 的汇编代码是由一段段的语句构成，声明了程序基本信息，而 hello_o_asm.txt 则是由一整块的代码构成，代码包含有完整的跳转逻辑。

4.4.2 分支转移

```

0000000000000000 <main>:
0: 55                                push    %rbp
1: 48 89 e5                          mov     %rsp,%rbp
4: 48 83 ec 20                        sub     $0x20,%rsp
8: 89 7d ec                          mov     %edi,-0x14(%rbp)
b: 48 89 75 e0                       mov     %rsi,-0x20(%rbp)
f: 83 7d ec 03                       cmpl    $0x3,-0x14(%rbp)
13: 74 16                             je      2b <main+0x2b>
15: 48 8d 3d 00 00 00 00              lea     0x0(%rip),%rdi          # 1c <main+0x1c>
                                18: R_X86_64_PC32      .rodata-0x4
                                callq   21 <main+0x21>
1c: e8 00 00 00 00                  1d: R_X86_64_PLT32      puts-0x4
                                mov     $0x1,%edi
21: bf 01 00 00 00                  callq   2b <main+0x2b>
26: e8 00 00 00 00                  27: R_X86_64_PLT32      exit-0x4
                                movl    $0x0,-0x4(%rbp)
2b: c7 45 fc 00 00 00 00              jmp     6f <main+0x6f>
32: eb 3b
34: 48 8b 45 e0                      mov     -0x20(%rbp),%rax
38: 48 83 c0 10                      add     $0x10,%rax
3c: 48 8b 10                        mov     (%rax),%rdx
3f: 48 8b 45 e0                      mov     -0x20(%rbp),%rax
43: 48 83 c0 08                      add     $0x8,%rax
47: 48 8b 00                        mov     (%rax),%rax
4a: 48 89 c6                        mov     %rax,%rsi
4d: 48 8d 3d 00 00 00 00              lea     0x0(%rip),%rdi          # 54 <main+0x54>
                                50: R_X86_64_PC32      .rodata+0x21

```

已确定的实际指令地址

```

.LFB5:
    .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
    leaq    .LC0(%rip), %rdi
    call    puts@PLT
    movl    $1, %edi
    call    exit@PLT
.L2:
    movl    $0, -4(%rbp)
    jmp     .L3
.L4:
    movq    -32(%rbp), %rax
    addq    $16, %rax
    movq    (%rax), %rdx
    movq    -32(%rbp), %rax
    addq    $8, %rax
    movq    (%rax), %rax

```

hello_o_asm.txt 包含了由操作数和操作码构成的机器语言，跳转指令使用已确定的实际指令地址；而 hello.s 使用助记符表示的段完成内部跳转。

4.4.3 函数调用

```

0000000000000000 <main>:
0: 55                push    %rbp
1: 48 89 e5          mov     %rsp,%rbp
4: 48 83 ec 20       sub     $0x20,%rsp
8: 89 7d ec          mov     %edi,-0x14(%rbp)
b: 48 89 75 e0       mov     %rsi,-0x20(%rbp)
f: 83 7d ec 03       cmpl    $0x3,-0x14(%rbp)
13: 74 16            je      2b <main+0x2b>
15: 48 8d 3d 00 00 00 00 lea     0x0(%rip),%rdi    # 1c <main+0x1c>
                                18: R_X86_64_PC32    .rodata+0x4
1c: e8 00 00 00 00    callq   21 <main+0x21>
                                1d: R_X86_64_PLT32    puts-0x4
21: bf 01 00 00 00    mov     $0x1,%edi
26: e8 00 00 00 00    callq   2b <main+0x2b>
                                27: R_X86_64_PLT32    exit-0x4
2b: c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%rbp)
32: eb 3b            jmp     6f <main+0x6f>
34: 48 8b 45 e0       mov     -0x20(%rbp),%rax
38: 48 83 c0 10       add     $0x10,%rax
3c: 48 8b 10          mov     (%rax),%rdx
3f: 48 8b 45 e0       mov     -0x20(%rbp),%rax
43: 48 83 c0 08       add     $0x8,%rax
47: 48 8b 00          mov     (%rax),%rax
4a: 48 89 c6          mov     %rax,%rsi
4d: 48 8d 3d 00 00 00 00 lea     0x0(%rip),%rdi    # 54 <main+0x54>
                                50: R_X86_64_PC32    .rodata+0x21

```

```

50: R_X86_64_PC32    .rodata+0x21
54: b8 00 00 00 00    mov     $0x0,%eax
59: e8 00 00 00 00    callq  5e <main+0x5e>
5a: R_X86_64_PLT32    printf-0x4
5e: 8b 05 00 00 00 00    mov     0x0(%rip),%eax    # 64 <main+0x64>
60: R_X86_64_PC32    sleepsecs-0x4
64: 89 c7             mov     %eax,%edi
66: e8 00 00 00 00    callq  6b <main+0x6b>
67: R_X86_64_PLT32    sleep-0x4
6b: 03 45 fc 01       addl    $0x1,-0x4(%rbp)
6f: 83 7d fc 09       cmpl    $0x9,-0x4(%rbp)
73: 7e bf             jle     34 <main+0x34>
75: e8 00 00 00 00    callq  7a <main+0x7a>
76: R_X86_64_PLT32    getchar-0x4
7a: b8 00 00 00 00    mov     $0x0,%eax
7f: c9               leaveq  %eax
80: c3               retq

```

```

.LFB5:
.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
leaq    .LC0(%rip), %r
call    puts@PLT
movl    $1, %edi
call    exit@PLT

.L2:
movl    $0, -4(%rbp)
jmp     .L3

.L4:
movq    -32(%rbp), %rax
addq    $16, %rax
movq    (%rax), %rdx
movq    -32(%rbp), %rax
addq    $8, %rax
(%rax). %rax

```

Call+函数名

显示应用程序


```

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

```

hello_o_asm.txt 中，由于库函数调用需要通过链接时重定位才能确定地址，call 地址后为占位符，指向下一条地址的位置，而 hello.s 中的函数调用 call+ 函数名。

4.4.4 数据访问

```

0000000000000000 <main>:
0: 55                               push    %rbp
1: 48 89 e5                         mov     %rsp,%rbp
4: 48 83 ec 20                       sub     $0x20,%rsp
8: 89 7d ec                         mov     %edi,-0x14(%rbp)
b: 48 89 75 e0                       mov     %rsi,-0x20(%rbp)
f: 83 7d ec 03                       cmpl    $0x3,-0x14(%rbp)
13: 74 16                            je      2b <main+0x2b>
15: 48 8d 3d 00 00 00 00             lea     0x0(%rip),%rdi      # 1c <main+0x1c>
                                18: R_X86_64_PC32      .rodata-0x4
                                callq   21 <main+0x21>
1c: e8 00 00 00 00                 callq   1d: R_X86_64_PLT32 puts-0x4
                                mov     $0x1,%edi
21: bf 01 00 00 00                 callq   2b <main+0x2b>
26: e8 00 00 00 00                 callq   27: R_X86_64_PLT32 exit-0x4
                                movl    $0x0,-0x4(%rbp)
2b: c7 45 fc 00 00 00 00             jmp     6f <main+0x6f>
32: eb 3b                               |
34: 48 8b 45 e0                       mov     -0x20(%rbp),%rax
38: 48 83 c0 10                       add     $0x10,%rax
3c: 48 8b 10                         mov     (%rax),%rdx
3f: 48 8b 45 e0                       mov     -0x20(%rbp),%rax
43: 48 83 c0 08                       add     $0x8,%rax
47: 48 8b 00                         mov     (%rax),%rax
4a: 48 89 c6                         mov     %rax,%rsi
4d: 48 8d 3d 00 00 00 00             lea     0x0(%rip),%rdi      # 54 <main+0x54>
                                50: R_X86_64_PC32      .rodata+0x21

```

```

50: R_X86_64_PC32    .rodata+0x21
54: b8 00 00 00 00    mov     $0x0,%eax
59: e8 00 00 00 00    callq  5e <main+0x5e>
5a: R_X86_64_PLT32    printf-0x4
5e: 8b 05 00 00 00 00    mov     0x0(%rip),%eax    # 64 <main+0x64>
60: R_X86_64_PC32    sleepsecs-0x4
64: 89 c7             mov     %eax,%edi
66: e8 00 00 00 00    callq  6b <main+0x6b>
67: R_X86_64_PLT32    sleep-0x4
6b: 83 45 fc 01       addl    $0x1,-0x4(%rbp)
6f: 83 7d fc 09       cmpl    $0x9,-0x4(%rbp)
73: 7e bf             jle     34 <main+0x34>
75: e8 00 00 00 00    callq  7a <main+0x7a>
76: R_X86_64_PLT32    getchar-0x4
7a: b8 00 00 00 00    mov     $0x0,%eax
7f: c9               leaveq  %eax
80: c3               retq

```

```

.LFB5:
.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
leaq     .LC0(%rip), %rdi
call     puts@PLT
movl     $1, %edi
call     exit@PLT
.L2:
movl     $0, -4(%rbp)
jmp      .L3
.L4:
movq     -32(%rbp), %rax
addq     $16, %rax
movq     (%rax), %rdx
movq     -32(%rbp), %rax
addq     $8, %rax
(%rax). %rax

```

显示应用程序

```

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

```

帮助

两者访问参数的方式相同，通过栈帧结构及`%rbp`相对寻址访问。

但 `hello_o_asm.txt` 中，对 `.rodata` 中 `printf` 的格式串的访问需要通过链接时重定位的绝对引用确定地址，因此在汇编代码相应位置仍为占位符表示，而 `hello.s` 中访问方式为 `sleepsecs+%rip`，格式串需要通过助记符。

4.5 本章小结

这章介绍了汇编的概念、作用和命令。分析了可重定位目标文件的格式，比较了反汇编代码与 `hello.s` 的相同点与不同点。

(第4章1分)

第 5 章 链接

5.1 链接的概念与作用

概念：链接是将各种代码和数据片段收集并组合成为一个单一文件的过程。

作用：将函数库中相应的代码组合到目标文件中。

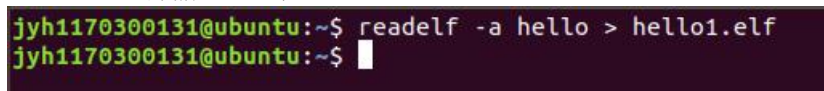
5.2 在 Ubuntu 下链接的命令

A terminal window titled 'jyh1170300131@ubuntu: ~' with a menu bar (Firefox 网络浏览器, 文件(F), 编辑(E), 查看(V), 搜索(S), 终端(T), 帮助(H)). The terminal shows the command: `ld -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 /usr/lib/gcc/x86_64-linux-gnu/7/crtbegin.o /usr/lib/gcc/x86_64-linux-gnu/7/crtend.o /usr/lib/x86_64-linux-gnu/crtn.o hello.o -lc -z relro -o hello`. The prompt is `jyh1170300131@ubuntu:~$` and a cursor is visible after the command.

```
jyh1170300131@ubuntu:~$ ld -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 /usr/lib/gcc/x86_64-linux-gnu/7/crtbegin.o /usr/lib/gcc/x86_64-linux-gnu/7/crtend.o /usr/lib/x86_64-linux-gnu/crtn.o hello.o -lc -z relro -o hello
jyh1170300131@ubuntu:~$
```

5.3 可执行目标文件 hello 的格式

通过在终端输入指令 `readelf -a hello > hello1.elf` 生成 hello 的 elf 格式文件。

A terminal window showing the command: `readelf -a hello > hello1.elf`. The prompt is `jyh1170300131@ubuntu:~$` and a cursor is visible after the command.

```
jyh1170300131@ubuntu:~$ readelf -a hello > hello1.elf
jyh1170300131@ubuntu:~$
```

(1) 节头表

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]	0000000000000000	NULL	0000000000000000	00000000
[1]	.interp 0000000000000001c	PROGBITS	0000000000400200 A 0 0	00000200 1
[2]	.note.ABI-tag 00000000000000020	NOTE	000000000040021c A 0 0	0000021c 4
[3]	.hash 00000000000000034	HASH	0000000000400240 A 5 0	00000240 8
[4]	.gnu.hash 0000000000000001c	GNU_HASH	0000000000400278 A 5 0	00000278 8
[5]	.dynsym 000000000000000c0	DYNSYM	0000000000400298 A 6 1	00000298 8
[6]	.dynstr 00000000000000057	STRTAB	0000000000400358 A 0 0	00000358 1
[7]	.gnu.version 00000000000000010	VERSYM	00000000004003b0 A 5 0	000003b0 2
[8]	.gnu.version_r 00000000000000020	VERNEED	00000000004003c0 A 6 1	000003c0 8
[9]	.rela.dyn 00000000000000030	RELA	00000000004003e0 A 5 0	000003e0 8
[10]	.rela.plt 00000000000000078	RELA	0000000000400410 AI 5 21	00000410 8
[11]	.init	PROGBITS	0000000000400480	00000480
[13]	.text 000000000000001e2	PROGBITS	0000000000400500 AX 0 0	00000500 16
[14]	.fini 00000000000000009	PROGBITS	00000000004006e4 AX 0 0	000006e4 4
[15]	.rodata 0000000000000003a	PROGBITS	00000000004006f0 A 0 0	000006f0 8
[16]	.eh_frame 000000000000000fc	PROGBITS	0000000000400730 A 0 0	00000730 8
[17]	.init_array 00000000000000008	INIT_ARRAY	0000000000600e00 WA 0 0	00000e00 8
[18]	.fini_array 00000000000000008	FINI_ARRAY	0000000000600e08 WA 0 0	00000e08 8
[19]	.dynamic 000000000000001e0	DYNAMIC	0000000000600e10 WA 6 0	00000e10 8
[20]	.got 00000000000000010	PROGBITS	0000000000600ff0 WA 0 0	00000ff0 8
[21]	.got.plt 00000000000000040	PROGBITS	0000000000601000 WA 0 0	00001000 8
[22]	.data 00000000000000014	PROGBITS	0000000000601040 WA 0 0	00001040 8
帮助	.bss 00000000000000004	NOBITS	0000000000601054 WA 0 0	00001054 1
[24]	.comment 00000000000000024	PROGBITS	0000000000000000 MS 0 0	00001054 1
[25]	.symtab 00000000000000600	SYMTAB	0000000000000000 26 41	00001078 8
[26]	.strtab 0000000000000020e	STRTAB	0000000000000000 0 0	00001678 1

在 ELF 格式文件中, Section Headers 对 hello 中所有的节信息进行了声明, 其中包括大小 Size 以及在程序中的偏移量 Offset, 可以用 HexEdit 定位各个节所占的区间。

其中 .text 节是保存程序代码指令的代码节; .rodata 保存只读数据; .plt 包含动态链接器调用从共享库导入的函数所必须的相关代码。存在于 text 段中; .bss

节保存未初始化全局数据，是 `data` 的一部分。程序加载时数据被初始化成 0，在程序执行期间可以赋值，未保存实际数据；`.got` 节保存全局偏移表；`.dynsym` 节保存共享库导入的动态符号信息；`.dynstr` 保存动态符号字符串表，存放一系列字符串，代表了符号的名称，以空字符作为终止符；`.rel` 节保存重定位信息；`.hash` 节保存一个查找符号散列表；`.symtab` 节保存 `ElfN_Sym` 类型的符号信息；`strtab` 节保存符号字符串表；`.shstrtab` 节保存节头字符串表，以空字符终止的字符串集合，保存每个节节名；`.ctors` 构造器、`.dtors` 析构器，指向构造函数和析构函数的函数指针。

(2) 程序头表

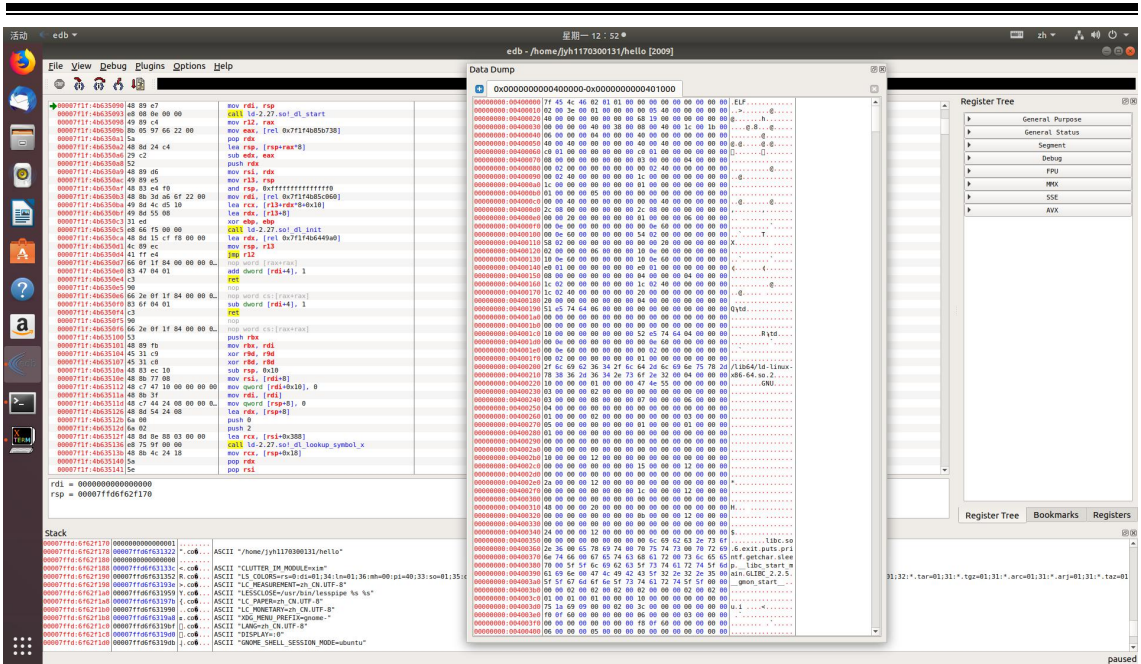
程序头:

Type	Offset FileSiz	VirtAddr MemSiz	PhysAddr Flags Align
PHDR	0x0000000000000040 0x00000000000001c0	0x0000000000400040 0x00000000000001c0	0x0000000000400040 R 0x8
INTERP	0x0000000000000200 0x000000000000001c	0x0000000000400200 0x000000000000001c	0x0000000000400200 R 0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]			
LOAD	0x0000000000000000 0x000000000000082c	0x0000000000400000 0x000000000000082c	0x0000000000400000 R E 0x200000
LOAD	0x0000000000000e00 0x0000000000000254	0x0000000000600e00 0x0000000000000258	0x0000000000600e00 RW 0x200000
DYNAMIC	0x0000000000000e10 0x00000000000001e0	0x0000000000600e10 0x00000000000001e0	0x0000000000600e10 RW 0x8
NOTE	0x000000000000021c 0x0000000000000020	0x000000000040021c 0x0000000000000020	0x000000000040021c R 0x4
GNU_STACK	0x0000000000000000 0x0000000000000000	0x0000000000000000 0x0000000000000000	0x0000000000000000 RW 0x10
GNU_RELRO	0x0000000000000e00 0x0000000000000200	0x0000000000600e00 0x0000000000000200	0x0000000000600e00 R 0x1

PTDR: 指定程序头表在文件及程序内存映像中的位置和大小；INTERP: 指定要作为解释程序调用的以空字符结尾的路径名的位置和大小；LOAD: 指定可装入段，通过 `p_filesz` 和 `p_memsz` 进行描述；DYNAMIC: 指定动态链接信息；NOTE: 指定辅助信息的位置和大小；GNU_STACK: 权限标志；GNU_RELRO: 指定在重定位结束之后哪些内存区域需要设置只读。

5.4 hello 的虚拟地址空间

计算机系统课程报告



.PDHR 起始位置为 0x400040,大小为 0x1c0

00000000:00400040	06 00 00 00 04 00 00 00 40 00 00 00 00 00 00 00	
00000000:00400050	40 00 40 00 00 00 00 00 40 00 40 00 00 00 00 00	
00000000:00400060	c0 01 00 00 00 00 00 00 c0 01 00 00 00 00 00 00	
00000000:00400070	08 00 00 00 00 00 00 00 03 00 00 00 04 00 00 00	
00000000:00400080	00 02 00 00 00 00 00 00 00 02 40 00 00 00 00 00	
00000000:00400090	00 02 40 00 00 00 00 00 1c 00 00 00 00 00 00 00	
00000000:004000a0	1c 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00	
00000000:004000b0	01 00 00 00 05 00 00 00 00 00 00 00 00 00 00 00	
00000000:004000c0	00 00 40 00 00 00 00 00 00 00 40 00 00 00 00 00	
00000000:004000d0	2c 08 00 00 00 00 00 00 2c 08 00 00 00 00 00 00	
00000000:004000e0	00 00 20 00 00 00 00 00 01 00 00 00 06 00 00 00	
00000000:004000f0	00 0e 00 00 00 00 00 00 00 0e 60 00 00 00 00 00	
00000000:00400100	00 0e 60 00 00 00 00 00 54 02 00 00 00 00 00 00	
00000000:00400110	58 02 00 00 00 00 00 00 00 00 26 00 00 00 00 00	
00000000:00400120	02 00 00 00 06 00 00 00 10 0e 00 00 00 00 00 00	
00000000:00400130	10 0e 60 00 00 00 00 00 10 0e 60 00 00 00 00 00	
00000000:00400140	e0 01 00 00 00 00 00 00 e0 01 00 00 00 00 00 00	
00000000:00400150	08 00 00 00 00 00 00 00 04 00 00 00 04 00 00 00	
00000000:00400160	1c 02 00 00 00 00 00 00 1c 02 40 00 00 00 00 00	
00000000:00400170	1c 02 40 00 00 00 00 00 20 00 00 00 00 00 00 00	
00000000:00400180	20 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00	
00000000:00400190	51 e5 74 64 06 00 00 00 00 00 00 00 00 00 00 00	
00000000:004001a0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00000000:004001b0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00000000:004001c0	10 00 00 00 00 00 00 00 52 e5 74 64 04 00 00 00	
00000000:004001d0	00 0e 00 00 00 00 00 00 0e 60 00 00 00 00 00 00	
00000000:004001e0	00 e5 60 00 00 00 00 00 00 02 00 00 00 00 00 00	
00000000:004001f0	00 02 00 00 00 00 00 00 01 00 00 00 00 00 00 00	

程序头:

Type	Offset FileSiz	VirtAddr MemSiz	PhysAddr Flags Align
PHDR	0x0000000000000040 0x00000000000001c0	0x0000000000400040 0x00000000000001c0	0x0000000000400040 R 0x8
INTERP	0x0000000000000200 0x00000000000001c	0x0000000000400200 0x00000000000001c	0x0000000000400200 R 0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]			
LOAD	0x0000000000000000 0x000000000000082c	0x0000000000400000 0x000000000000082c	0x0000000000400000 R E 0x200000
LOAD	0x0000000000000e00 0x0000000000000258	0x0000000000600e00 0x0000000000000258	0x0000000000600e00 RW 0x200000
DYNAMIC	0x0000000000000e10 0x00000000000001e0	0x0000000000600e10 0x00000000000001e0	0x0000000000600e10 RW 0x8
NOTE	0x000000000000021c 0x0000000000000020	0x000000000040021c 0x0000000000000020	0x000000000040021c R 0x4
GNU_STACK	0x0000000000000000 0x0000000000000000	0x0000000000000000 0x0000000000000000	0x0000000000000000 RW 0x10
GNU_RELRO	0x0000000000000e00 0x0000000000000200	0x0000000000600e00 0x0000000000000200	0x0000000000600e00 R 0x1

.INTERP:起始位置为 0x400200,大小为 0x1c

```

00000000:00400200 2f 6c 69 62 36 34 2f 6c 64 2d 6c 69 6e 75 78 2d /lib64/ld-linux-
00000000:00400210 78 38 36 2d 36 34 2e 73 6f 2e 32 00 04 00 00 00 x86-64.so.2.....
00000000:00400220 10 00 00 00 01 00 00 00 47 4e 55 00 00 00 00 00 .....GNU.....

```

程序头:

Type	Offset FileSiz	VirtAddr MemSiz	PhysAddr Flags Align
PHDR	0x0000000000000040 0x00000000000001c0	0x0000000000400040 0x00000000000001c0	0x0000000000400040 R 0x8
INTERP	0x0000000000000200 0x00000000000001c	0x0000000000400200 0x00000000000001c	0x0000000000400200 R 0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]			
LOAD	0x0000000000000000 0x000000000000082c	0x0000000000400000 0x000000000000082c	0x0000000000400000 R E 0x200000
LOAD	0x0000000000000e00 0x0000000000000258	0x0000000000600e00 0x0000000000000258	0x0000000000600e00 RW 0x200000
DYNAMIC	0x0000000000000e10 0x00000000000001e0	0x0000000000600e10 0x00000000000001e0	0x0000000000600e10 RW 0x8
NOTE	0x000000000000021c 0x0000000000000020	0x000000000040021c 0x0000000000000020	0x000000000040021c R 0x4
GNU_STACK	0x0000000000000000 0x0000000000000000	0x0000000000000000 0x0000000000000000	0x0000000000000000 RW 0x10
GNU_RELRO	0x0000000000000e00 0x0000000000000200	0x0000000000600e00 0x0000000000000200	0x0000000000600e00 R 0x1

.LOAD: 起始位置为 0x400000, 大小为 0x82c

00000000:00400000	7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00	ELF.....
00000000:00400010	02 00 3e 00 01 00 00 00 00 05 40 00 00 00 00	...>...@...
00000000:00400020	40 00 00 00 00 00 00 00 68 19 00 00 00 00 00	@.....h...
00000000:00400030	00 00 00 00 40 00 38 00 08 00 40 00 1c 00 1b 00	...@.8...@...
00000000:00400040	06 00 00 00 04 00 00 00 40 00 00 00 00 00 00	...@.....@...
00000000:00400050	40 00 40 00 00 00 00 00 40 00 40 00 00 00 00	@. @.....@. @...
00000000:00400060	c0 01 00 00 00 00 00 00 c0 01 00 00 00 00 00	[].....[].....
00000000:00400070	08 00 00 00 00 00 00 00 03 00 00 00 04 00 00
00000000:00400080	00 02 00 00 00 00 00 00 00 02 40 00 00 00 00@.....

程序头:

Type	Offset FileSiz	VirtAddr MemSiz	PhysAddr Flags Align
PHDR	0x0000000000000040	0x0000000000400040	0x0000000000400040
INTERP	0x00000000000001c0	0x00000000000001c0	R 0x8
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]			
LOAD	0x0000000000000000	0x0000000000400000	0x0000000000400000
	0x000000000000082c	0x000000000000082c	R E 0x200000
LOAD	0x0000000000000e00	0x0000000000600e00	0x0000000000600e00
	0x0000000000000254	0x0000000000000258	RW 0x200000
DYNAMIC	0x0000000000000e10	0x0000000000600e10	0x0000000000600e10
	0x00000000000001e0	0x00000000000001e0	RW 0x8
NOTE	0x000000000000021c	0x000000000040021c	0x000000000040021c
	0x0000000000000020	0x0000000000000020	R 0x4
GNU_STACK	0x0000000000000000	0x0000000000000000	0x0000000000000000
	0x0000000000000000	0x0000000000000000	RW 0x10
GNU_RELRO	0x0000000000000e00	0x0000000000600e00	0x0000000000600e00
	0x0000000000000200	0x0000000000000200	R 0x1

以此类推，可得所有

PTDR: 指定程序头表在文件及程序内存映像中的位置和大小; INTERP: 指定要作为解释程序调用的以空字符结尾的路径名的位置和大小; LOAD: 指定可装入段, 通过 `p_filesz` 和 `p_memsz` 进行描述; DYNAMIC: 指定动态链接信息; NOTE: 指定辅助信息的位置和大小; GNU_STACK: 权限标志; GNU_RELRO: 指定在重定位结束之后哪些内存区域需要设置只读。

5.5 链接的重定位过程分析

```
jyh1170300131@ubuntu:~$ objdump -d -r hello
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
ff8 <__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
<puts@GLIBC_2.2.5>
  4004b6: 68 00 00 00 00 00
  4004bb: e9 e0 ff ff ff      jmpq .plt

00000000004004c0 <printf@plt>:
  4004c0: ff 25 5a 0b 20 00    jmpq *0x200b5a(%rip)    # 601020
<printf@GLIBC_2.2.5>
  4004c6: 68 01 00 00 00 00    pushq $0x1
  4004cb: e9 d0 ff ff ff      jmpq 4004a0 <.plt>

jyh1170300131@ubuntu: ~
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

00000000004005e7 <main>:
  4005e7: 55                  push %rbp
  4005e8: 48 89 e5            mov %rsp,%rbp
  4005eb: 48 83 ec 20         sub $0x20,%rbp
  4005ef: 89 7d ec            mov %ebx,%edi
  4005f2: 48 89 75 e0         mov %rsi,%rdi
  4005f6: 83 7d ec 03         cmpl $0x0,%edi
  4005fa: 74 16              je 400612 <main+0x2b>
  4005fc: 48 8d 3d f5 00 00 00 lea 0xf5(%rip),%rdi    # 4006f8
<_IO_stdin_used+0x8>
  400603: e8 a8 fe ff ff      callq 4004b0 <puts@plt>
  400608: bf 01 00 00 00      mov $0x1,%edi
  40060d: e8 ce fe ff ff      callq 4004e0 <exit@plt>
  400612: c7 45 fc 00 00 00 00 movl $0x0,-0x4(%rbp)
  400619: eb 3b              jmp 400656 <main+0x6f>
  40061b: 48 8b 45 e0         mov -0x20(%rbp),%rax
  40061f: 48 83 c0 10         add $0x10,%rax
  400623: 48 8b 10           mov (%rax),%rdx
  400626: 48 8b 45 e0         mov -0x20(%rbp),%rax
  40062a: 48 83 c0 08         add $0x8,%rax
  40062e: 48 8b 00           mov (%rax),%rax
  400631: 48 89 c6           mov %rax,%rsi
  400634: 48 8d 3d e2 00 00 00 lea 0xe2(%rip),%rdi    # 40071d
<_IO_stdin_used+0x2d>
  40063b: b8 00 00 00 00      mov $0x0,%eax
  400640: e8 7b fe ff ff      callq 4004c0 <printf@plt>
  400645: 8b 05 05 0a 20 00    mov 0x200a05(%rip),%eax    # 601

```

调用的函数被添加

访问的全局变量和函数有了确定地址

5.6 hello 的执行流程

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

ld-2.27.so!_dl_runtime_resolve_xsave 0x7fce 8cc4e680
-ld-2.27.so!_dl_fixup            0x7fce 8cc46df0
--ld-2.27.so!_dl_lookup_symbol_x 0x7fce 8cc420b0
libc-2.27.so!exit                0x7fce 8c889128

```

5.7 Hello 的动态链接分析

在调用共享库函数时，编译器没有办法预测这个函数的运行时地址，因为定义它的共享模块在运行时可以加载到任意位置。正常的方法是为该引用生成一条重定位记录，然后动态链接器在程序加载的时候再解析它。GNU 编译系统使用延迟绑定 (lazybinding)，将过程地址的绑定推迟到第一次调用该过程时。延迟绑定是通过 GOT 和 PLT 实现的。GOT 是数据段的一部分，而 PLT 是代码段的一部分。PLT 是一个数组，其中每个条目是 16 字节代码。PLT[0] 是一个特殊条目，它跳转到动态链接器中。每个被可执行程序调用的库函数都有它自己的 PLT 条目。每个条目都负责调用一个具体的函数。GOT 也是一个数组，其中每个条目是 8 字节地址。和 PLT 联合使用时，GOT[0] 和 GOT[1] 包含动态链接器在解析函数地址时会使用的信息。GOT[2] 是动态链接器在 ld-linux.so 模块中的入口点。其余的每个条目对应于一个被调用的函数，其地址需要在运行时被解析。每个条目都有一个相匹配的 PLT 条目。

GOT 的起始位置为 601000

```

[21] .got.plt          PROGBITS          0000000000601000 00001000
      0000000000000040 0000000000000008 WA      0      0      8

```


调用 _dl_start 之前

00000000:00601000	10 0e 60 00 00 00 00 00	00 00 00 00 00 00 00 00
00000000:00601010	00 00 00 00 00 00 00 00	b6 04 40 00 00 00 00 00
00000000:00601020	10 0e 60 00 00 00 00 00	70 a1 5f cf 78 7f 00 00
00000000:00601010	50 87 3e cf 78 7f 00 00	b6 04 40 00 00 00 00 00

调用 puts 之前

00000000:004004b0	ff 25 62 0b 20 00	jmp qword [rel 0x601018]
00000000:004004b6	68 00 00 00 00	push 0
00000000:004004bb	e9 e0 ff ff ff	jmp hello!.plt
00000000:004004c0	ff 25 5a 0b 20 00	jmp qword [rel 0x601020]

调用一次后

00000000:004004b0	ff 25 62 0b 20 00	jmp qword [rel 0x601018]
00000000:004004b6	68 00 00 00 00	push 0
00000000:004004bb	e9 e0 ff ff ff	jmp hello!.plt
00000000:004004c0	ff 25 5a 0b 20 00	jmp qword [rel 0x601020]

由此可知 printf 链接到了动态库

5.8 本章小结

本章了解了链接的概念作用，分析可执行文件 hello 的 ELF 格式及其虚拟地址空间，同时通过实例分析了重定位过程、加载以及运行时函数调用顺序以及动态链接过程，深入理解链接和重定位的过程。

(第 5 章 1 分)

第 6 章 hello 进程管理

6.1 进程的概念与作用

概念：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。

作用：进程给应用程序提供的关键抽象有两种：一个独立的逻辑控制流，提供一个假象，程序独占地使用处理器。或一个私有的地址空间，提供一个假象，程序在独占地使用系统内存。

6.2 简述壳 Shell-bash 的作用与处理流程

作用：

是用户使用 Linux 的桥梁。Shell 应用程序提供了一个界面，用户通过这个界面访问操作系统内核的服务。

处理流程：

从终端读入输入的命令，将输入字符串切分获得所有的参数，如果是内置命令则立即执行否则调用相应的程序为其分配子进程并运行，shell 接受键盘输入信号，并对这些信号进行相应处理。

6.3 Hello 的 fork 进程创建过程

父进程通过调用 fork 函数创建一个新的运行的子进程。新创建的子进程几乎但不完全与父进程相同。子进程得到与父进程用户级虚拟地址空间相同但独立的一份副本。子进程还获得与父进程任何打开文件描述符相同的副本，也就是说当父进程调用 fork 时。子进程可以读写父进程中打开的任何文件。

fork 后调用一次返回两次，在父进程中 fork 会返回子进程的 PID，在子进程中 fork 会返回 0；父进程与子进程是并发运行的独立进程。内核能够以任何方式交替执行他们逻辑控制流中的指令。

6.4 Hello 的 execve 过程

当 fork 之后，子进程调用 execve 函数在当前进程的上下文中加载并运行一

个新程序即 hello 程序，`execve` 调用驻留在内存中的被称为启动加载器的操作系统代码来执行 hello 程序，加载器删除子进程现有的虚拟内存段，并创建一组新的代码、数据、堆和栈段。新的栈和堆段被初始化为零，通过将虚拟地址空间中的页映射到可执行文件的页大小的片，新的代码和数据段被初始化为可执行文件中的内容。最后加载器设置 PC 指向 `_start` 地址，`_start` 最终调用 hello 中的 `main` 函数。在加载过程中没有任何从磁盘到内存的数据复制。直到 CPU 引用一个被映射的虚拟页时才会进行复制，操作系统利用它的页面调度机制自动将页面从磁盘传送到内存。

6.5 hello 的进程执行

Linux 系统中的每个程序都运行在一个进程上下文中，有自己的虚拟地址空间。当 shell 运行一个程序时，父 shell 进程生成一个子进程，它是父进程的一个复制。子进程通过 `execve` 系统调用启动加载器。加载器删除子进程现有的虚拟内存段，并创建一组新的代码、数据、堆和栈段。新的栈和堆段被初始化为零。通过将虚拟地址空间中的页映射到可执行文件的页大小的片(chunk)，新的代码和数据段被初始化为可执行文件的内容。最后，加载器跳转到 `_start` 地址，它最终会调用应用程序的 `main` 函数。

6.6 hello 的异常与信号处理

异常种类：中断和终止

命令运行

```
jyh1170300131@ubuntu:~$ ./hello 1170300131 金宇航
Hello 1170300131 金宇航
Hello 1170300131 金宇航
^Z
[1]+ 已停止                  ./hello 1170300131 金宇航
```

(1) ps

```
jyh1170300131@ubuntu:~$ ps
  PID TTY          TIME CMD
  6465 pts/0        00:00:00 bash
  6546 pts/0        00:00:00 hello
  6551 pts/0        00:00:00 ps
```

(2) jobs

```
jyh1170300131@ubuntu:~$ jobs
[1]+ 已停止                  ./hello 1170300131 金宇航
```

(3) pstree

```
jyh1170300131@ubuntu:~$ pstree
systemd--ModemManager--2*[{ModemManager}]
      |NetworkManager--dhclient
      |                2*[{NetworkManager}]
      |VGAuthService
      |accounts-daemon--2*[{accounts-daemon}]
      |acpid
      |avahi-daemon--avahi-daemon
      |bluetoothd
      |boltd--2*[{boltd}]
      |colord--2*[{colord}]
      |cron
      |cups-browsed--2*[{cups-browsed}]
      |cupsd
      |2*[dbus-daemon]
      |fcitx
      |fcitx-dbus-watc
      |fwupd--4*[{fwupd}]
      |显示应用程序
      |3gdm-session-wor
      |gdm-wayland-ses
      |gnome-session-b
      |gnome-s+
      |gsd-a11+
      |...
```

(4) fg

```
jyh1170300131@ubuntu:~$ fg
./hello 1170300131 金宇航
Hello 1170300131 金宇航
Hello 1170300131 金宇航
^Z
[1]+ 已停止                  ./hello 1170300131 金宇航
```

(5) kill

```
jyh1170300131@ubuntu:~$ kill 6546

jyh1170300131@ubuntu:~$ fg 1
./hello 1170300131 金宇航
已终止
jyh1170300131@ubuntu:~$ ps
  PID TTY          TIME CMD
  6465 pts/0        00:00:00 bash
  6552 pts/0        00:00:00 ps
```

6.7 本章小结

这一章从进程的角度分别描述了 hello 子进程 fork 和 execve 过程，并针对 execve 过程中虚拟内存映像以及栈组织结构等作出说明。了解了逻辑控制流中内核的调度及上下文切换等机制。阐述了 Shell 和 Bash 运行的处理流程以 hello 执行过程中可能引发的异常和信号处理。

(第 6 章 1 分)

第 7 章 hello 的存储管理

7.1 hello 的存储器地址空间

逻辑地址：指由程序产生的与段相关的偏移地址部分。

线性地址：地址空间 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。

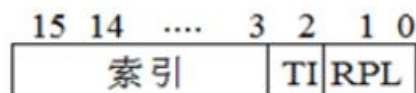
虚拟地址：CPU 通过生成一个虚拟地址。即 hello 里面的虚拟内存地址。

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。

7.2 Intel 逻辑地址到线性地址的变换-段式管理

段式寄存器用于存放段选择符：CS 是指程序代码所在段；SS 是指栈区所在段；DS 是指全局静态数据所在段；其他三个段寄存器 ES、GS 和 FS 可指向任意数据段。

段选择符中各字段含义为：



其中 TI 表示描述符表，TI=0 则为全局描述符表；TI=1 则为局部描述符表。RPL 表示环保护，为第 0 级，位于最高级的内核态，RPL=11，为第 3 级，位于最低级的用户态，中间环留给中间软件使用。高 13 位表示用来确定当前使用的段描述符在描述符表中的位置。

逻辑地址向线性地址转换的过程中被选中的描述符先被送至描述符 cache，每次从描述符 cache 中取 32 位段基址，与 32 位段内偏移量相加得到线性地址，

7.3 Hello 的线性地址到物理地址的变换-页式管理

线性地址被分为以固定长度为单位的组，称为页，一个 32 位的机器，线性地址最大可为 4G，可以用 4KB 为一个页来划分，整个线性地址就被划分为一个 2^{20} 的大数组，共有 2^{20} 个次方个页。这个大数组我们称之为页目录。目录中的每一个目录项，就是一个地址——对应的页的地址。另一类“页”，我们称之为物理页。是分页单元把所有的物理内存也划分为固定长度的管理单位，

它的长度一般与内存页是一一对应的。这个数组有 2^{20} 个成员，每个成员是一个地址，要表示这个数组，要占去 4MB 的内存空间。为了节省空间，引入二级管理模式的机器来组织分页单元。分页单元中，页目录是唯一的，它的地址放在 CPU 的 cr3 寄存器中，是进行地址转换的开始点。每一个活动的进程，因为都有其独立的对应的虚拟内存（页目录也是唯一的），那么它也对应了一个独立的页目录地址。——运行一个进程，需要将它的页目录地址放到 cr3 寄存器中每一个 32 位的线性地址被划分为三部份，面目录索引(10 位)：页表索引(10 位)：偏移(12 位)依据以下步骤进行转换：从 cr3 中取出进程的页目录地址，根据线性地址前十位，在数组中，找到对应的索引项，因为引入了二级管理模式，页目录中的项，不再是页的地址，而是一个页表的地址。页的地址被放到页表中。根据线性地址的中间十位，在页表中找到页的起始地址。将页的起始地址与线性地址中最后 12 位相加，得到最终我们想要的物理地址。

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

虚拟地址被划分成 4 个 VPN 和 1 个 VPO。每个 VPN_i 都是一个到第 i 级页表的索引，其中 $1 \leq i \leq 4$ 。第 j 级页表中的每个 PTE， $1 \leq j \leq 3$ ，都指向第 j+1 级的某个页表的基址。第四级页表中的每个 PTE 包含某个物理页面的 PPN，或者一个磁盘块的地址。为了构造物理地址，在能够确定 PPN 之前，MMU 必须访问 4 个 PTE。将得到的 PPN 和虚拟地址中的 VPO 串联起来，就得到相应的物理地址。

7.5 三级 Cache 支持下的物理内存访问

首先将高速缓存与地址翻译结合起来，CPU 发出一个虚拟地址给 TLB 搜索，如果命中直接发送到 L1cache，没有命中则在页表中找到后发送，到了 L1 里面以后，寻找物理地址也需要检测是否命中，此时通过 CPU 的高速缓存机制搭配 TLB 使机器在翻译地址的时候性能得以充分发挥。

7.6 hello 进程 fork 时的内存映射

共享对象在虚拟内存中的应用为一个对象可以被映射到虚拟内存的一个区域，要么作为共享对象，要么作为私有对象。如果一个进程将一个共享对象映射到它的虚拟地址空间的一个区域内，那么这个进程对这个区域的任何写操作，对于那些也把这个共享对象映射到它们虚拟内存的其他进程而言，也是可见的。而且，这些变化也会反映在磁盘上的原始对象中。另一方面，对于一个映射到私有对象的区域做的改变，对于其他进程来说是不可见的，并且进程对这个区域所做的任何写操作都不会反映在磁盘上的对象中。一个映射到共享对象的虚拟内存区

域叫做共享区域。类似地，也有私有区域。私有对象使用一种叫写时复制来映射至虚拟内存中，多个进程可将一个私有对象映射到其内存不同区域，共享该对象同一物理副本对于每个映射私有对象的进程，相应私有区域的页表条目都被标记为只读，并且区域结构被标记为私有的写时复制。只要没有进程试图写它自己的私有区域，它们就可以继续共享物理内存中对象的一个单独副本。然而，只要有一个进程试图写私有区域内的某个页面，那么这个写操作就会触发一个保护故障。当故障处理程序注意到保护异常是由于进程试图写私有的写时复制区域中的一个页面而引起的，它就会在物理内存中创建这个页面的一个新副本，更新页表条目指向这个新的副本，然后恢复这个页面的可写权限。当 fork 函数被系统调用时，内核会为 hello 创建子进程，同时会创建各种数据结构并分配给 hello 唯一的 PID。为了给 hello 创建虚拟内存，内核创建了当前进程的 mm_struct、区域结构和样表的原样副本，并将两个进程中的每个页面都标记为只读，并将两个进程中的每个区域结构都标记为写时复制。

7.7 hello 进程 execve 时的内存映射

execve 函数在 hello 进程中加载并运行 hello，主要步骤为：删除已存在的用户区域、映射 hello 私有区域、映射共享区域、设计程序计数器。下一次调度 hello，将从入口点开始执行。Linux 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

当地址翻译硬件从内存中读对应 PTE 时有效位为 0 则表明该页未被缓存，触发缺页异常。缺页异常调用内核中的缺页异常处理程序。缺页处理程序搜索区域结构的链表，把 A 和每个区域结构中的 vm_start 和 vm_end 做比较，如果指令不合法，缺页处理程序就触发一个段错误、终止进程。缺页处理程序检查试图访问的内存是否合法，如果不合法则触发保护异常终止此进程。缺页处理程序确认引起异常的是合法的虚拟地址和操作，则选择一个牺牲页，如果牺牲页中内容被修改，内核会将其先复制回磁盘。无论是否被修改，牺牲页的页表条目均会被内核修改。接下来内核从磁盘复制需要的虚拟页到 DRAM 中，更新对应的页表条目，重新执行导致缺页的指令。

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虚拟内存区域堆。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，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。分配器有两种

基本风格，都要求应用显式地分配块，它们的不同之处在于由哪个实体来负责释放已分配的块。显式分配器要求应用显式地释放任何已分配的块。隐式分配器要求分配器检测一个已分配块何时不再使用，那么就释放这个块，自动释放未使用的已经分配的块的过程叫做垃圾收集。而自动释放未使用的已分配的块的过程叫做垃圾收集。隐式空闲链表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。显式空闲链表有两种方式来维护一种是先进后出，另一种是地址顺序。此处不详细展开。放置空闲块的策略有三种，分别是首次适配、下一次适配、最佳适配。首次适配从头开始搜索空闲链表，选择第一个合适的空闲块。下一次适配和首次适配很相似，只不过不是从链表的起始处开始每次搜索，而是从上一次查询结束的地方开始。最佳适配检查每个空闲块，选择适合所需请求大小的最小空闲块。

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 IO 管理

8.1 Linux 的 IO 设备管理方法

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

8.2 简述 Unix IO 接口及其函数

接口操作：

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 $k \geq m$ 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 write(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

8.3.1 printf

```
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;  
}
```

arg 获得输出时格式化串对应值。

8.3.2 vsprintf

```
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);
}
```

按照格式 `fmt` 结合参数 `args` 生成格式化之后的字符串，并返回字符串的长度。

8.3.3 write:

```
mov eax, _NR_write
mov ebx, [esp + 4]
mov ecx, [esp + 8]

int INT_VECTOR_SYS_CALL
```

将栈中参数放入寄存器，`ecx` 是字符个数，`ebx` 存放第一个字符地址，`int INT_VECTOR_SYS_CALL` 代表通过系统调用 `syscall`

8.3.4 sys_call:

```
call save
push dword [p_proc_ready]

sti

push ecx
push ebx

call [sys_call_table + eax * 4]

add esp, 4 * 3

mov [esi + EAXREG - P_STACKBASE], eax

cli

ret
```

`syscall` 将字符串中的字节“Hello 1170300131 金宇航”从寄存器中通过总线复制到显卡的显存中，显存中存储的是字符的 ASCII 码。字符显示驱动子程序

将通过 ASCII 码在字模库中找到点阵信息将点阵信息存储到 vram 中。显示芯片会按照一定的刷新频率逐行读取 vram，并通过信号线向液晶显示器传输每一个点，实现 printf 格式化输出。字符串就显示在了屏幕上。

8.4 getchar 的实现分析

异步异常-键盘中断的处理：键盘中断处理是底层的硬件异常，当用户按下键盘时，内核会调用异常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 函数 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。实现读取一个字符的功能。

8.5 本章小结

本章简述了 Linux 的 I/O 设备管理机制，Unix I/O 接口及函数，分析了 printf 函数和 getchar 函数的实现。

（第 8 章 1 分）

结论

1. 首先是编写，通过编辑器创建 `hello.c`。
2. 然后是预处理，将第一步中获得的 `hello.c` 中调用的所有外部库展开合并到 `hello.i` 中。
3. 第三步是编译，将预处理得到的 `hello.i` 文件编译成 `hello.s`（汇编文件）。
4. 然后进行汇编，得到可重定位目标文件 `hello.o`。
5. 第五步是链接，和动态链接库链接成为可执行目标程序 `hello`。
6. 然后是运行，在 shell 中输入 `./hello 1170300131 金宇航`。
7. 然后是创建子进程，shell 调用 `fork` 创建子进程。
8. 第八步是运行程序，shell 调用 `execve`，`execve` 调用启动加载器，加映射虚拟内存，进入程序入口后程序载入物理内存，进入 `main` 函数。
9. 然后执行指令，CPU 分配时间片，在一个时间片中 `hello` 享有 CPU 资源，顺序执行自己的控制逻辑流。
10. 然后访问内存，MMU 将程序中使用的虚拟内存地址通过页表映射成物理地址。
11. 然后进行动态内存申请，`printf` 调用 `malloc` 向动态内存分配器申请堆中的内存。
12. 如果运行途中键入 `ctr-z` 则调用 shell 的信号处理函数挂起。如果键入 `ctr-z` 则函数停止。
13. 最后，进程结束后被 shell 回收，为这个进程创建的所有数据结构都会被内核删除，`hello` 完成了它的一生。

感悟：这次的大作业带领我分析了 `hello` 程序从产生到运行的各个阶段的底层实现，让我对这学期所学的知识有了更系统的认知。这个作业比想象中的更加复杂，调用了这学期所学的大量知识，也让我认识到了，其实我对这些知识的理解比想象中的更加不足，对于很多知识有学习时的遗漏，也有学习后的遗忘。通过这次的作业我对这些掌握不好的知识又有了进一步的学习，收获颇丰。

（结论 0 分，缺失 -1 分，根据内容酌情加分）

附件

hello.i	预处理后得到的文本文件
hello.s	编译后得到的汇编文件
hello.o	汇编后得到的可重定位目标执行文件
hello.elf	hello.o 的 ELF 格式
hello	链接后得到的可执行目标文件
hello1.elf	hello 的 ELF 格式

(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] 龚奕利. 深入理解计算机系统. 北京: 机械工业出版社, 2016.
- [2] 关于 Linux 中 sleep() 函数说明
https://blog.csdn.net/fly__chen/article/details/53175301
- [3] 百度百科 :
extern:<https://baike.baidu.com/item/extern/4443005?fr=aladdin>
- [4] 百度百科 : 机器语言 :
<https://baike.baidu.com/item/%E6%9C%BA%E5%99%A8%E8%AF%AD%E8%A8%80/2019225?fr=aladdin>
- [5] 百度百科: 编译: <https://baike.baidu.com/item/编译/1258343?fr=aladdin>

(参考文献 0 分, 缺失 -1 分)