

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>1170300107</u>
班	級	<u>1703001</u>
學	生	<u>嚴未圻</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

本文通过一个小程序 `hello.c` 从开始到结束的“一生”，具体讲述了在 `linux` 系统下，`hello` 经历的从编译到最终被回收的过程。借助 CSAPP 课本，我们对每章内容结合 `hello` 程序具体分析，理解了较为底层的计算机知识。这篇文章对于理解 CSAPP 有一定帮助，也对于理解计算机运行的原理有帮助。

关键词：CSAPP，编译，计算机组成原理，虚拟内存

（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	- 7 -
第 3 章 编译	- 8 -
3.1 编译的概念与作用	- 8 -
3.2 在 UBUNTU 下编译的命令	- 8 -
3.3 HELLO 的编译结果解析	- 8 -
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	- 19 -
第 5 章 链接	- 20 -
5.1 链接的概念与作用	- 20 -
5.2 在 UBUNTU 下链接的命令	- 20 -
5.3 可执行目标文件 HELLO 的格式	- 20 -
5.4 HELLO 的虚拟地址空间	- 23 -
5.5 链接的重定位过程分析	- 24 -
5.6 HELLO 的执行流程	- 26 -
5.7 HELLO 的动态链接分析	- 26 -
5.8 本章小结	- 27 -
第 6 章 HELLO 进程管理	- 28 -
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	- 30 -
6.6 HELLO 的异常与信号处理	- 31 -
6.7 本章小结	- 34 -
第 7 章 HELLO 的存储管理	- 35 -
7.1 HELLO 的存储器地址空间	- 35 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理	- 35 -
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	- 37 -
7.5 三级 CACHE 支持下的物理内存访问	- 38 -
7.6 HELLO 进程 FORK 时的内存映射	- 38 -
7.7 HELLO 进程 EXECVE 时的内存映射	- 39 -
7.8 缺页故障与缺页中断处理	- 39 -
7.9 动态存储分配管理	- 40 -
7.10 本章小结	- 41 -
第 8 章 HELLO 的 IO 管理	- 42 -
8.1 LINUX 的 IO 设备管理方法	- 42 -
8.2 简述 UNIX IO 接口及其函数	- 42 -
8.3 PRINTF 的实现分析	- 42 -
8.4 GETCHAR 的实现分析	- 46 -
8.5 本章小结	- 46 -
结论	- 46 -
附件	- 48 -
参考文献	- 49 -

第 1 章 概述

1.1 Hello 简介

根据 Hello 的自白，利用计算机系统的术语，简述 Hello 的 P2P，020 的整个过程。

hello.c 经过预处理器（cpp）的预处理生成 hello.i（文本文件），接下来经过编译器（ccl）生成汇编程序 hello.s（文本文件），然后进入汇编器（as）生成可重定位的目标程序（二进制文档）再和其他的共享库链接，通过连接器生成最终可执行的目标文件 hello。同时，shell 中的 fork 函数生成进程，这样就完成了 P2P（from program to process）

接下来，由于 shell 要执行 execve 函数来执行这个可执行文件，所以我们要通过映射虚拟内存，来把这段程序导入 DRAM，进入程序入口之后程序开始载入到物理内存，利用系统 start 函数进入 main 主函数，开始执行 hello 主体，CPU 给 hello 分配执行的时间片，以及进程控制的逻辑流程。程序结束通过 exit 退出，这时候 shell 的父进程来执行回收子进程的函数，将分配给 hello 的相关虚拟内存删除，这样就是 020（zero to zero）

1.2 环境与工具

硬件环境：Intel Core i7-7700HQ x64CPU,16G RAM,256G SSD +1T HDD.

软件环境：Ubuntu18.04.1 LTS

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

1.3 中间结果

hello hello 的可执行目标文件
hello.c 原始代码
hello.elf hello 的 elf 文件
hello.i hello.c 经过预处理的文件
hello.o hello 的可重定位文件
helloo.elf hello.o 的 elf 文件
helloo.txt hello 的反汇编代码
hello.s hello 的汇编代码
hello.txt hello.o 的反汇编代码

1.4 本章小结

本章简介了这次大作业中利用到的工具以及中间文件的作用
(第 1 章 0.5 分)

第 2 章 预处理

2.1 预处理的概述与作用

预处理的主要是处理这里：

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

图 2.1

将宏常量，条件编译以及库文件全部载入。将库文件从系统中调用出来直接插入文本文档，把宏常量的定义修改成实际值，最后根据条件编译后面的条件选择需要编译的代码

2.2 在 Ubuntu 下预处理的命令

```
ywq1170300107@ubuntu:~/hitcs/helloproject$ cpp hello.c > hello.i
ywq1170300107@ubuntu:~/hitcs/helloproject$ ls
hello.c  hello.i
```

图 2.2

利用 `cpp xxx.c > xxx.i`

或者 `gcc -E hello.c -o hello.i` 来进行预处理

2.3 Hello 的预处理结果解析

```
# 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;
}
/mnt/hgfs/hitcs/helloproject/hello.i CWD: /mnt/hgfs/hitcs/helloprojectE5 Line: 3102/3118:1
```

图 2.3

可以看到经过预处理之后，我们的文件一下变成了 3000 多行，其中前面有很多行的代码，就是 `stdio.h` `unistd.h` `stdlib.h` 的具体内容

2.4 本章小结

本章主要是探讨了预处理过程中发生的事情，预处理时，我们将程序文本开头调用的库文件直接展开，并且插入到文本文档中，将宏常量替换，将条件编译按条件进行了编译，为后面的程序执行奠定了基础。

(第 2 章 0.5 分)

第 3 章 编译

3.1 编译的概念与作用

编译是在预处理结束之后，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汇编代码，这个过程是程序构建的核心部分。

3.2 在 Ubuntu 下编译的命令

```
ywq1170300107@ubuntu:~/hitcs/helloproject$ gcc -S hello.c -o hello.s
ywq1170300107@ubuntu:~/hitcs/helloproject$ ls
hello.c  hello.i  hello.s
```

图 3.1

利用 gcc 的-S 选项

3.3 Hello 的编译结果解析

3.3.0 符号解析

```
.file "hello.c"
.text
.global sleepsecs
.data
.align 4
.type sleepsecs, @object
.size sleepsecs, 4
```

图 3.2

.file 源文件名称

.text 代码段

.global 全局变量

.data 数据段

.align 对齐数据

.type 定义这个是函数或者是对象类型

.size 声明大小

```
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
```

图 3.3

.long 定义了一个 long 型数据
 .section .rodata 后面是 rodata 节、
 .string 定义了字符串
 .LCx 跳转的位置

3.3.1 数据分析

hello.s 中用到的 C 数据类型有：整数、字符串、数组。

（0）字符串

用到的字符串有两个，一个是

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

图 3.4

第二个是

```
.LC1:
    .string "Hello %s %s\n"
    .text
    .globl  main
    .type   main, @function
```

图 3.5

这两个字符串分别定义在 rodata 节的.LC0 和.LC1 段

（1）整数

我们首先定义了整数

hello 程序中涉及的整数有 int sleepsecs，这个是被定义为全局变量，并且已经被赋值，属于强名称，所以应该定义在.data 节中，我们可以看到，data 节里面确实有这个数据（至于为什么编译器将他定义为 long 可能是因为这台机器中，int 型和 long 型大小长度一样，编译器是更偏向于使用 long）

```

.data
.align 4
.type    sleepsecs, @object
.size    sleepsecs, 4
sleepsecs:
.long    2
.section .rodata

```

图 3.6

还有就是 `int i`，一般的编译器会将局部变量储存在寄存器中或者存到栈里面。这个编译器把它存进栈里。我们可以从下图中看到，编译器选择将 `i` 存放在 `rbp-4` 的地方（我之所以这么判断是因为程序主体是将 `i` 和 10 比较，这里是判断 `i <= 9`，也就是 `i < 10`）

```

    addl    $1, -4(%rbp)
.L3:
    cmpl    $9, -4(%rbp)
    jle     .L4

```

图 3.7

然后是 `argc`，作为参数传入，同样保存在栈中，保存在 `rbp - 20`；
 剩余的都是属于立即数，这些立即数直接编码在汇编代码中。

（2）数组

程序中涉及数组 `char *argv[]` 是 `main` 函数传入的参数，`argv` 元素每个的指针数组大小是 8 字节，`argv` 指针指向已经分配好的、一片存放着字符指针的连续空间，起始地址为 `argv`。在 `hello.s` 中，取值的方法是

```

movl    %edi, -20(%rbp)
movq    %rsi, -32(%rbp)

```

图 3.8

刚才我们看到了 `argc` 是放在 `rbp-20`，这里 `argv` 放在 `rbp-32`。
 然后在循环内部

```

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

```

图 3.9

红色箭头部分分别是将 `argv[2]` 传入 `rdx` 作为 `printf` 的第三个参数，
 将 `argv[1]` 传入 `rsi` 作为 `printf` 的第二个参数，
 最后把 `.LC1` 的字符串传入 `rdi` 作为第一个参数

3.3.2 赋值操作

程序中涉及的赋值操作有：

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

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

`movb`: 1Byte `movw`: 2Byte `movl`: 4Byte `movq`: 8Byte

3.3.3 类型转换

由于 `sleepsecs` 是一个 `int` 型，所以默认把 2.5 设置成 2 赋值给 `sleepsecs`。在这里的舍入规则是向零舍入。

3.3.4 算术操作

<code>leaq S,D</code>	$D = \&S$
<code>INC D</code>	$D += 1$
<code>DEC D</code>	$D -= 1$
<code>NEG D</code>	$D = -D$
<code>ADD S,D</code>	$D = D + S$
<code>SUB S,D</code>	$D = D - S$
<code>IMULQ S</code>	$R[\%rdx]:R[\%rax] = S * R[\%rax]$ (有符号)
<code>MULQ S</code>	$R[\%rdx]:R[\%rax] = S * R[\%rax]$ (无符号)
<code>IDIVQ S</code>	$R[\%rdx] = R[\%rdx]:R[\%rax] \bmod S$ (有符号) $R[\%rax] = R[\%rdx]:R[\%rax] \div S$
<code>DIVQ S</code>	$R[\%rdx] = R[\%rdx]:R[\%rax] \bmod S$ (无符号) $R[\%rax] = R[\%rdx]:R[\%rax] \div S$

本次程序中有：

0.i++程序使用了 `addl`，也可用 `inc`，但是 `hello.s` 使用的是 `addl`

1.汇编中使用了 `leaq` 来计算 `rip+.LC1` 并传递给 `%rdi`

3.3.5 关系操作

进行关系操作的汇编指令：

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

程序中涉及的关系判断：

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

图 3.10

根据 argc 是不是 3 来进行跳转。

```
.L3:
    cmpl    $9, -4(%rbp)
    jle     .L4
```

图 3.11

根据 i 是不是小于等于 9，也就是是不是小于 10 来判断，进行的是循环。

3.3.6 控制转移

程序中涉及的控制转移：

0. argc != 3 处，当程序不等于 3，我们执行.L2 处指令，否则我们继续执行。

```
cmpl    $3, -20(%rbp)
je      .L2
leaq    .LC0(%rip), %rdi
call    puts@PLT
movl    $1, %edi
call    exit@PLT
.L2:
```

图 3.12

1. for 循环处，我们根据 i 是否小于 10 来进行判断，所以我们需要的是一个判断并且跳转的指令。

```
addl    $1, -4(%rbp)
.L3:
    cmpl    $9, -4(%rbp)
```

图 3.13

跳转到前面之后反复执行，直到 i 不再小于 10.

3.3.7 函数操作

函数是一种封装的机制，用一组参数和可选的返回值实现某种功能。调用函数可以实现的功能有：

0. 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。

1. 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。

2. 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。同时在函数内部的时候要注意调用者寄存器要保存进栈，最后再恢复。

64 位机器中函数中的参数储存顺序

1	2	3	4	5	6	7~n
%rdi	%rsi	%rdx	%rcx	%r8	%r9	栈

Hello 中调用的函数

0.main 函数：

0.控制转移，main 函数是被系统启动函数（_libc_start_main 调用），call 指令将下一条指令的地址压入栈中，然后再跳转到 main 函数

1.传递参数，外部的调用过程向 main 函数传递的参数是 int 型的 argc 和 char* 的数组 argv，分别利用的是 %rdi %rsi，函数正常返回为 0

2.分配和释放内存，使用 %rbp 记录栈帧的底，函数分配栈的空间在 %rbp 指向的位置来储存，leave 指令相当于 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 rip，将下一条需要执行的指令设置为以前压入的地址，这样就能返回函数调用之前的状态

1.printf 函数

0.传递数据：第一次将 %rdi 设置为 “Usage: Hello 学号 姓名! \n” 字符串的首地址。

```
leaq    .LC0(%rip), %rdi
call    puts@PLT
```

图 3.14

第二次设置设置 %rdi 为 “Hello %s %s\n” 的首地址 %rsi 为 argv[1], %rdx 为 argv[2]。

```
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)
```

图 3.15

1.控制传递：第一我们调用的是 puts@PLT，第二次调用的是 printf@PLT

2.exit 函数

0.传递数据：将%edi 设置为 1

1.控制传递：call exit@PLT

3.sleep 函数：

0.传递数据：将%edi 设置为 sleepsecs

1.控制传递：call sleep@PLT

4.getchar 函数：

0.控制传递：call getchar@PLT

3.4 本章小结

这一章主要阐述了编译器怎么将 C 语言处理成.s 的汇编代码的。

编译器将.i 的拓展程序编译成.s 的汇编代码。经过编译之后，我们 hello.c 从 C 语言变成更加接近机器看的懂的底层汇编代码

(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，将这些指令打包成可重定位目标程序，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

```
ywq1170300107@ubuntu:~/hitcs/helloproject$ as hello.s -o hello.o
ywq1170300107@ubuntu:~/hitcs/helloproject$ ls
hello.c hello.i hello.o hello.s
ywq1170300107@ubuntu:~/hitcs/helloproject$
```

图 4.1

或者 `gcc -c hello.s -o hello.o`

4.3 可重定位目标 elf 格式

使用 `readelf -a hello.o > helloo.elf` 指令获得 hello.o 文件的 ELF 格式。其组成如下：

```
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:               1144 (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
```

图 4.2

0.ELf 头: 包括了从 16 字节的 Magic 序列开始, 接下来的部分包括帮助链接器语法分析和解释目标文件的信息, 包括 ELF 头大小, 目标文件类型, 机器类型, 字节头部表的文件偏移等等。

1.节头部表

节头:

[号]	名称 大小	类型 全体大小	地址 旗标	链接 信息	偏移量 对齐
[0]	0000000000000000	NULL	0000000000000000	0 0	0
[1]	.text 0000000000000081	PROGBITS 0000000000000000	0000000000000000 AX	0 0	00000040 1
[2]	.rela.text 00000000000000c0	RELA 0000000000000018	0000000000000000 I	10 1	00000338 8
[3]	.data 0000000000000004	PROGBITS 0000000000000000	0000000000000000 WA	0 0	000000c4 4
[4]	.bss 0000000000000000	NOBITS 0000000000000000	0000000000000000 WA	0 0	000000c8 1
[5]	.rodata 000000000000002b	PROGBITS 0000000000000000	0000000000000000 A	0 0	000000c8 1
[6]	.comment 0000000000000025	PROGBITS 0000000000000001	0000000000000000 MS	0 0	000000f3 1
[7]	.note.GNU-stack 0000000000000000	PROGBITS 0000000000000000	0000000000000000	0 0	00000118 1
[8]	.eh_frame 0000000000000038	PROGBITS 0000000000000000	0000000000000000 A	0 0	00000118 8
[9]	.rela.eh_frame 0000000000000018	RELA 0000000000000018	0000000000000000 I	10 8	000003f8 8
[10]	.syntab 00000000000000198	SYMTAB 0000000000000018	0000000000000000	11 9	00000150 8
[11]	.strtab 000000000000004d	STRTAB 0000000000000000	0000000000000000	0 0	000002e8 1
[12]	.shstrtab 0000000000000061	STRTAB 0000000000000000	0000000000000000	0 0	00000410 1

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

There are no section groups in this file.

图 4.3

节头部表中包含了文件中各个节的语义, 包括了节的类型以及节的位置以及大小, 后面给出的信息是关于 flags 的具体意思。

2.代码重定位节.rela.text

重定位节 '.rela.text' at offset 0x338 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

图 4.4

开始交代了重定位节的偏移。接下来通过一个表的方式列出了.text 节中需要重定位的信息，包括.L0, puts 函数、exit 函数、.L1、printf 函数、sleepsecs、sleep 函数、getchar 函数。

整个.rela.text 节中的内容有 offset 指示重定位代码在.text 中偏移量，info 包括两部分，symbol 和 type，symbol 代表了重定位到的目标在.symtab 节中的偏移量，type 代表重定位的类型。Addend 重定位计算的辅助信息，Type 是重定位到的目标的类型，Name 是重定位到的目标的名称。

定位一个类型 R_X86_64_PC32 的引用。计算重定位目标地址的算法如下（设需要重定位的.text 节中的位置为 src,设重定位的目的位置 dst）：

$refptr = s + r.offset$ (1)

$refaddr = ADDR(s) + r.offset$ (2)

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

其中 s 是节开头 (1) 指向 src 的指针 (2) 计算 src 的运行时地址，(3) 中，ADDR(r.symbol)计算 dst 的运行时地址，在本例中，ADDR(r.symbol)获得的是 dst 的运行时地址，因为需要设置的是绝对地址，即 dst 与下一条指令之间的地址之差，所以需要加上 r.addend=-4。

3..rela.eh_frame 是 eh_frame 节的重定位信息

```
重定位节 '.rela.eh_frame' at offset 0x3f8 contains 1 entry:
  偏移量      信息      类型      符号值      符号名称 + 加数
0000000000020 0002000000002 R_X86_64_PC32 0000000000000000 .text + 0

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

Symbol table '.symtab' contains 17 entries:
  Num:      Value              Size Type Bind Vis      Ndx Name
  0: 0000000000000000          0 NOTYPE LOCAL DEFAULT UND
  1: 0000000000000000          0 FILE  LOCAL DEFAULT ABS hello.c
  2: 0000000000000000          0 SECTION LOCAL DEFAULT 1
  3: 0000000000000000          0 SECTION LOCAL DEFAULT 3
  4: 0000000000000000          0 SECTION LOCAL DEFAULT 4
  5: 0000000000000000          0 SECTION LOCAL DEFAULT 5
  6: 0000000000000000          0 SECTION LOCAL DEFAULT 7
  7: 0000000000000000          0 SECTION LOCAL DEFAULT 8
  8: 0000000000000000          0 SECTION LOCAL DEFAULT 6
  9: 0000000000000000          4 OBJECT GLOBAL DEFAULT 3 sleepsecs
 10: 0000000000000000        129 FUNC  GLOBAL DEFAULT 1 main
 11: 0000000000000000          0 NOTYPE GLOBAL DEFAULT UND _GLOBAL_OFFSET_TABLE_
 12: 0000000000000000          0 NOTYPE GLOBAL DEFAULT UND puts
 13: 0000000000000000          0 NOTYPE GLOBAL DEFAULT UND exit
 14: 0000000000000000          0 NOTYPE GLOBAL DEFAULT UND printf
 15: 0000000000000000          0 NOTYPE GLOBAL DEFAULT UND sleep
 16: 0000000000000000          0 NOTYPE GLOBAL DEFAULT UND getchar
```

图 4.5

主要是对于.symtab 表的重定位进行规定

4.4 Hello.o 的结果解析

对于.o 文件反汇编代码

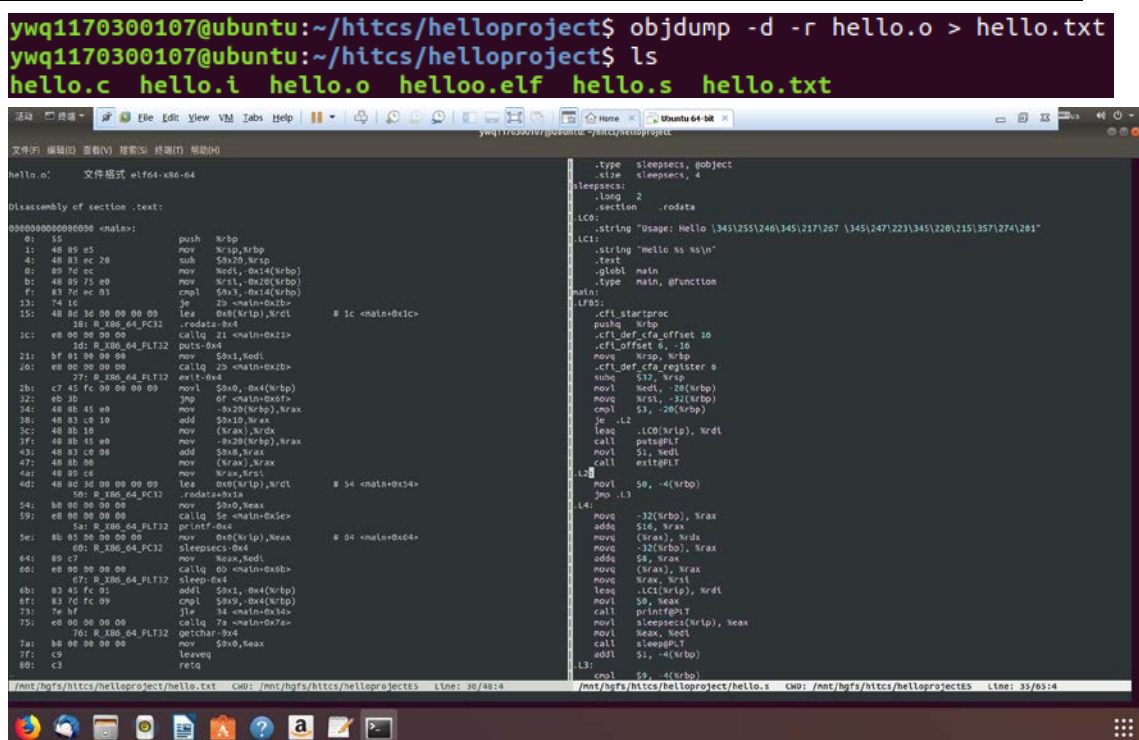


图 4.6

可以通过观察发现,大致来说两者的区别并不是特别大,尤其是代码部分并无显著区别。区别点主要在这几个方面:

0.分支跳转:反汇编代码的分支跳转是经过汇编得来的,所以自然没有什么像.s文件中那样的助记符.L1,在反汇编代码中他们都是确定的地址

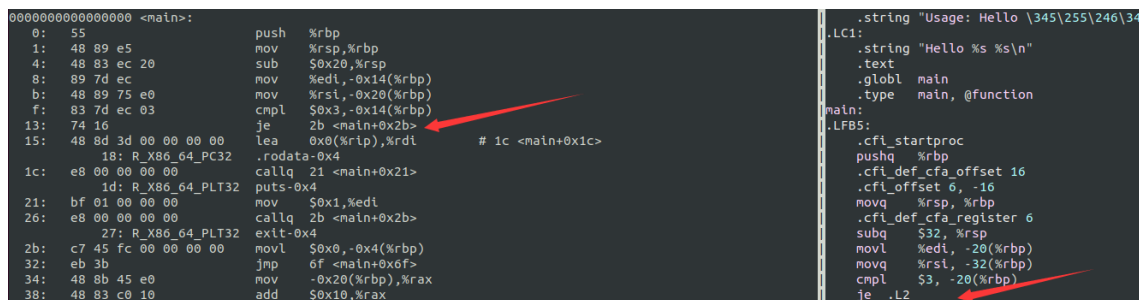
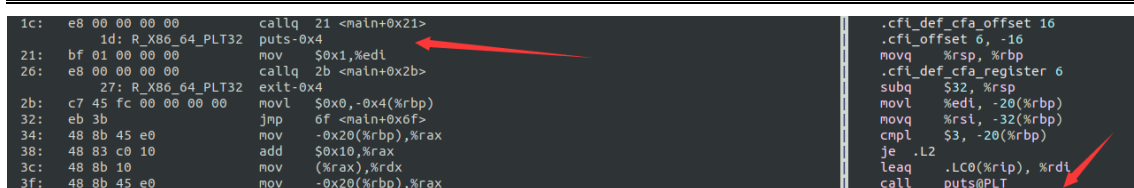


图 4.7

1.函数调用:.s文件中的函数调用跟的都是函数的名字,而反汇编代码中call的目标地址是当前指令的下一条指令,这是因为,我们调用的函数都是库中的函数如printf, getchar, 在未完成重定位的可重定位的目标文件中,不确定这些地址的位置,只有到动态连接器最终确定,利用.rela.text中的重定位条目为他确定具体地址。



```

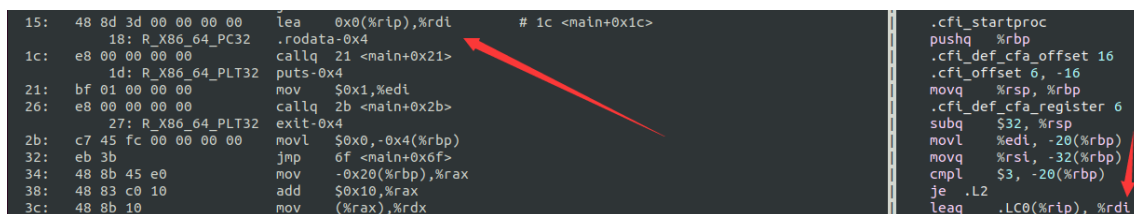
1c: e8 00 00 00 00 callq 21 <main+0x21>
    1d: R_X86_64_PLT32 puts-0x4
21: bf 01 00 00 00 mov $0x1,%edi
26: e8 00 00 00 00 callq 2b <main+0x2b>
    27: R_X86_64_PLT32 exit-0x4
2b: c7 45 fc 00 00 00 movl $0x0,-0x4(%rbp)
32: eb 3b jmp 6f <main+0x6f>
34: 48 8b 45 e0 mov -0x20(%rbp),%rax
38: 48 83 c0 10 add $0x10,%rax
3c: 48 8b 10 mov (%rax),%rdx
3f: 48 8b 45 e0 mov -0x20(%rbp),%rax

.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

```

图 4.8

2.全局变量：.s 中访问.rodata 节直接使用段名称，在反汇编代码中，是 0+%rip 因为.rodata 节地址也是在链接器中才能确定下来，所以访问需要重定位。所以，汇编代码汇编过程中，操作数设置为 0 并且添加重定位条目



```

15: 48 8d 3d 00 00 00 leaq 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 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

.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

```

图 4.9

4.5 本章小结

本章介绍的过程是 hello 从 hello.s 到啊 hello.o 的汇编过程，将原始的汇编代码生成成为没有重定位的机器代码，我们利用 objdump 将两者对比，更加清晰的看到了两者映射之间的关系。

(第 4 章 1 分)

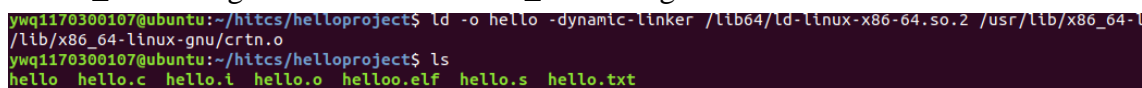
第 5 章 链接

5.1 链接的概念与作用

链接是将各种代码和数据片段收集并组合成一个单一文件的过程，这个文件可被加载到内存并执行。链接可以执行于编译时，也就是在源代码被编译成机器代码时；也可以执行于加载时，也就是在程序被加载器加载到内存并执行时；甚至于运行时，也就是由应用程序来执行。链接完成了对代码中的重定位，链接是由叫做链接器的程序执行的。Ld -o hello -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o hello.o /usr/lib/x86_64-linux-gnu/libc.so /usr/lib/x86_64-linux-gnu/crtn.o

5.2 在 Ubuntu 下链接的命令

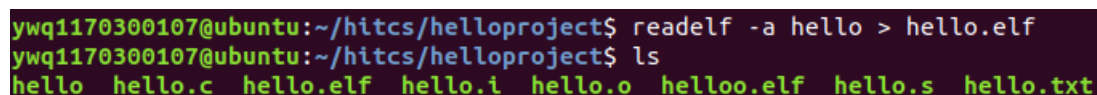
```
Ld -o hello -dynamic-linker /lib64/ld-linux-x86-64.so.2
/usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o hello.o
/usr/lib/x86_64-linux-gnu/libc.so /usr/lib/x86_64-linux-gnu/crtn.o
```



```
ywq1170300107@ubuntu:~/hitcs/helloproject$ ld -o hello -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o /usr/lib/x86_64-linux-gnu/libc.so /usr/lib/x86_64-linux-gnu/crtn.o
ywq1170300107@ubuntu:~/hitcs/helloproject$ ls
hello hello.c hello.i hello.o hello.o hello.o hello.s hello.txt
```

图 5.1

5.3 可执行目标文件 hello 的格式



```
ywq1170300107@ubuntu:~/hitcs/helloproject$ readelf -a hello > hello.elf
ywq1170300107@ubuntu:~/hitcs/helloproject$ ls
hello hello.c hello.elf hello.i hello.o hello.o hello.o hello.s hello.txt
```

图 5.2

进入 elf 文件中查看

开始还是看到的是 elf 表头，我们可以发现这次文件类型是可执行文件

```
ELF 头:
Magic:      7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
类别:      ELF64
数据:      2 补码, 小端序 (little endian)
版本:      1 (current)
OS/ABI:     UNIX - System V
ABI 版本:   0
类型:      EXEC (可执行文件)
系统架构:   Advanced Micro Devices X86-64
版本:      0x1
入口点地址: 0x400500
程序头起点: 64 (bytes into file)
Start of section headers: 5920 (bytes into file)
标志:      0x0
本头的大小: 64 (字节)
程序头大小: 56 (字节)
Number of program headers: 8
节头大小:  64 (字节)
节头数量:  25
字符串表索引节头: 24
```

图 5.3

我们接下来查看节头，其中包括了大小和偏移量，我们就可以根据这个来定位到程序中各个节所占区间。其中地址是我们载入虚拟地址的起始地址。

节头:

[号]	名称 大小	类型 全体大小	地址 旗标	链接	偏移量 信息	对齐
[0]	0000000000000000	NULL	0000000000000000	0	0	0
[1]	.interp 000000000000001c	PROGBITS 0000000000000000	0000000000400200 A	0	0	1
[2]	.note.ABI-tag 0000000000000020	NOTE 0000000000000000	000000000040021c A	0	0	4
[3]	.hash 0000000000000034	HASH 0000000000000004	0000000000400240 A	5	0	8
[4]	.gnu.hash 000000000000001c	GNU_HASH 0000000000000000	0000000000400278 A	5	0	8
[5]	.dynsym 00000000000000c0	DYNSYM 0000000000000018	0000000000400298 A	6	1	8
[6]	.dynstr 0000000000000057	STRTAB 0000000000000000	0000000000400358 A	0	0	1
[7]	.gnu.version 0000000000000010	VERSYM 0000000000000002	00000000004003b0 A	5	0	2
[8]	.gnu.version_r 0000000000000020	VERNEED 0000000000000000	00000000004003c0 A	6	1	8
[9]	.rela.dyn 0000000000000030	RELA 0000000000000018	00000000004003e0 A	5	0	8
[10]	.rela.plt 0000000000000078	RELA 0000000000000018	0000000000400410 AI	5	19	8
[11]	.init 0000000000000017	PROGBITS 0000000000000000	0000000000400488 AX	0	0	4
[12]	.plt 0000000000000060	PROGBITS 0000000000000010	00000000004004a0 AX	0	0	16
[13]	.text 0000000000000132	PROGBITS 0000000000000000	0000000000400500 AX	0	0	16
[14]	.fini 0000000000000009	PROGBITS 0000000000000000	0000000000400634 AX	0	0	4
[15]	.rodata 000000000000002f	PROGBITS 0000000000000000	0000000000400640 A	0	0	4
[16]	.eh_frame 00000000000000fc	PROGBITS 0000000000000000	0000000000400670 A	0	0	8
[17]	.dynamic 00000000000001a0	DYNAMIC 0000000000000010	0000000000600e50 WA	6	0	8
[18]	.got 0000000000000010	PROGBITS 0000000000000008	0000000000600ff0 WA	0	0	8
[19]	.got.plt 0000000000000040	PROGBITS 0000000000000008	0000000000601000 WA	0	0	8
[20]	.data 0000000000000008	PROGBITS 0000000000000000	0000000000601040 WA	0	0	4
[21]	.comment 0000000000000024	PROGBITS 0000000000000001	0000000000000000 MS	0	0	1
[22]	.symtab 0000000000000498	SYMTAB 0000000000000018	0000000000000000	23	28	8

图 5.4

```

000000000000024 000000000000001 MS 0 0 1
[22] .symtab SYMTAB 0000000000000000 00001070
0000000000000498 0000000000000018 23 28 8
[23] .strtab STRTAB 0000000000000000 00001508
0000000000000150 0000000000000000 0 0 1
[24] .shstrtab STRTAB 0000000000000000 00001658
00000000000000c5 0000000000000000 0 0 1
Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
l (large), p (processor specific)
There are no section groups in this file.

```

图 5.5

5.4 hello 的虚拟地址空间

使用 edb 加载 hello

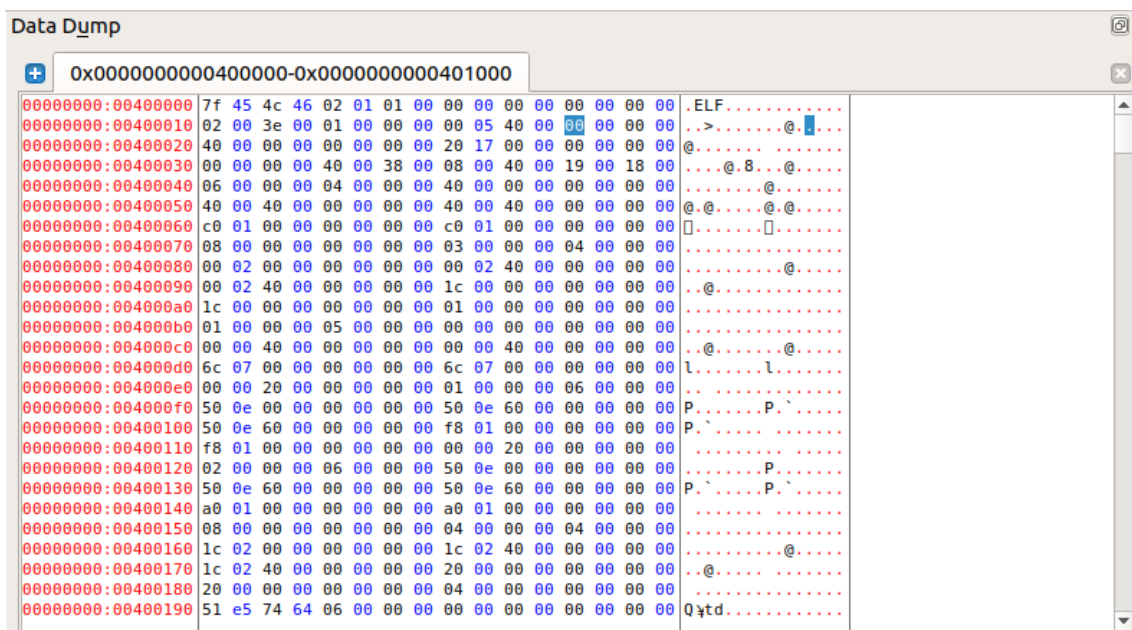


图 5.6

从 0x400000 程序被载入到最后程序结束我们可以看到这些节的排列顺序是和 5.3 中图相同的。以 .rodata 节为例，在 0x400640，我们找到 0x400640

```

00000000:00400640 01 00 02 00 55 73 61 67 65 3a 20 48 65 6c 6c 6f ... Usage: Hello
00000000:00400650 20 e5 ad a6 e5 8f b7 20 e5 a7 93 e5 90 8d ef bc ... %s %s %s
00000000:00400660 81 00 48 65 6c 6c 6f 20 25 73 20 25 73 0a 00 00 ... zR..x..
00000000:00400670 14 00 00 00 00 00 00 00 01 7a 52 00 01 78 10 01 ...
00000000:00400680 1b 0c 07 08 90 01 07 10 10 00 00 00 1c 00 00 00 ...
00000000:00400690 70 fe ff ff 2b 00 00 00 00 00 00 00 14 00 00 00 ...
00000000:004006a0 00 00 00 00 01 7a 52 00 01 78 10 01 1b 0c 07 08 ...
00000000:004006b0 90 01 00 00 10 00 00 00 1c 00 00 00 74 fe ff ff ... t

```

图 5.7

接下来我们继续查看 elf 文件

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 0x000000000000076c	0x0000000000400000 0x000000000000076c	0x0000000000400000 R E	0x200000
LOAD	0x00000000000000e50 0x00000000000001f8	0x0000000000600e50 0x00000000000001f8	0x0000000000600e50 RW	0x200000
DYNAMIC	0x00000000000000e50 0x00000000000001a0	0x0000000000600e50 0x00000000000001a0	0x0000000000600e50 RW	0x8
NOTE	0x000000000000021c 0x0000000000000020	0x000000000040021c 0x0000000000000020	0x000000000040021c R	0x4
GNU_STACK	0x0000000000000000 0x0000000000000000	0x0000000000000000 0x0000000000000000	0x0000000000000000 RW	0x10
GNU_RELRO	0x00000000000000e50 0x00000000000001b0	0x0000000000600e50 0x00000000000001b0	0x0000000000600e50 R	0x1

图 5.8

程序头表告诉链接器运行时加载的内容，每一个表项都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以及访问权限。从表项种类我们可以看出程序包含了 8 段：

PHDR
INTERP
LOAD
DYNAMIC
NOTE
GNU_STACK
GNU_RELRO

再通过 edb 查看

可以看到在 0x600000 到 0x600fff 中和 0x400000 到 0x400fff 是相同的，在 fff 之后存的其实是 .dynamic~.shstrtab 节

5.5 链接的重定位过程分析

hello.o 的反汇编文本和 hello 的反汇编文本相比，hello 的反汇编多了许多的节。

去网上搜索了一下这些节的用途

.节的名称	描述
.interp	保存 ld.so 的路径
.note.ABI-tag	Linux 下特有的 section
.hash	符号的哈希表
.gnu.hash	GNU 拓展的符号的哈希表
.dynsym	运行时/动态符号表
.dynstr	存放 .dynsym 节中的符号名称
.gnu.version	符号版本
.gnu.version_r	符号引用版本

.rela.dyn	运行时/动态重定位表
.rela.plt	.plt 节的重定位条目
.init	程序初始化需要执行的代码
.plt	动态链接-过程链接表
.fini	当程序正常终止时需要执行的代码
.eh_frame	包括了异常和源语言的信息
.dynamic	存放被 ld.so 使用的动态链接信息
.got	动态链接-全局偏移量表-存放变量
.got.plt	动态链接-全局偏移量表-存放函数
.data	初始化了的数据
.comment	一串包含编译器的 NULL-terminated 字符串

比较两个反汇编代码的区别

(0) 函数的个数发生改变:

我们在使用 ld 命令链接的时候, 定义了程序入口函数_start, 初始化函数_init, 通过_start 函数调用 hello 中的 main 函数, 同时我们链接的 libc.so 是动态链接共享库, 定义了我们用到的 printf, sleep, getchar 和 exit。链接器会将上述函数加入

(1) 函数调用过程

链接器解析重定位条目发现了类型为 R_X86_64_PLT32 的重定位条目

```
1c: e8 00 00 00 00 callq 21 <main+0x21>
    1d: R_X86_64_PLT32 puts-0x4
```

图 5.9

而此时 PLT 中已经加入了我们需要的函数, .text 和.plt 节的相对距离确定, 链接器计算相对距离, 并且将函数调用之改为 PLT 中函数与下一条指令之间的相对距离, 从而指向对应的函数例如 puts。

(2) .rodata 节的引用: 链接器解析重定位条目发现 R_X86_64_PC32 类型的对.rodata 节的引用

```
15: 48 8d 3d 00 00 00 00 lea 0x0(%rip),%rdi # 1c <main+0x1c>
    18: R_X86_64_PC32 .rodata-0x4
```

图 5.10

现在.rodata 节与.text 节之间的相对距离已经确定, 因此链接器修改 lea 之后的地址之差来指向.rodata 中相应的字符串。我们计算另一个的

```
4d: 48 8d 3d 00 00 00 00 lea 0x0(%rip),%rdi # 54 <main+0x54>
    50: R_X86_64_PC32 .rodata+0x1a
```

图 5.11

Refptr = s + r.offset(Pointer point to 0x400582)

```
40057f: 48 8d 3d dc 00 00 00 lea 0xdc(%rip),%rdi # 400662 <_IO_stdin_used+0x22>
400586: b8 00 00 00 00 mov $0x0,%eax
40058b: e8 30 ff ff ff callq 4004c0 <printf@plt>
400590: 8b 05 ae 0a 20 00 mov 0x200aae(%rip),%eax # 601044 <sleepsecs>
400596: 89 c7 mov %eax,%edi
```

图 5.12

$\text{Refaddr} = \text{ADDR}(s) + r.\text{offset} = \text{ADDR}(\text{main}) + r.\text{offset} = 0x400532 + 0x50 = 0x400582$

$*\text{refptr} = (\text{unsigned})(\text{ADDR}(r.\text{symbol}) + r.\text{addend} - \text{refaddr}) = \text{ADDR}(\text{str2}) + r.\text{addend} - \text{refaddr} = (\text{unsigned}) 0xdc$

可以看到计算的值和实际的相同。

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

5.7 Hello 的动态链接分析

调用动态共享库定义的函数，编译器没有办法预测这个函数运行时的地址，因为定义它的共享模块可以在运行的时候加载到任意位置。GNU 使用延迟绑定来解决这个问题，动态链接在函数第一次调用的时候启动，使用 PLT+GOT，GOT 中存放函数的目标地址，PLT 使用 GOT 中地址跳转到目标函数。

在 `dl_init` 调用之前，对于每一条 PIC 函数调用，调用的目标地址都实际指向 PLT 中的代码逻辑，GOT 存放的是 PLT 中函数调用指令的下一条指令地址。

从上面我们可以看到 `.got.plt` 开始于 `0x601000`，我们去对应的位置寻找。

```
00000000:00601000 50 0e 60 00 00 00 00 00 00 00 00 00 00 00 00 P.
00000000:00601010 00 00 00 00 00 00 00 00 b6 04 40 00 00 00 00 .@
00000000:00601020 c6 04 40 00 00 00 00 00 d6 04 40 00 00 00 00 t.@
00000000:00601030 e6 04 40 00 00 00 00 00 f6 04 40 00 00 00 00 h.@
00000000:00601040 00 00 00 00 02 00 00 00 47 43 43 3a 20 28 55 62 .GCC: (Ub
```

图 5.13

这个是初始的 `.got.plt` 表。

运行了 `dl_init` 之后

```
Data Dump
+ 0x0000000000400000-0x0000000000401000 0x0000000000602000-0x0000000000602000
00000000:00601000 50 0e 60 00 00 00 00 00 00 00 00 00 00 00 P.
00000000:00601010 00 00 00 00 00 00 00 00 b6 04 40 00 00 00 00 .@
00000000:00601020 c6 04 40 00 00 00 00 00 d6 04 40 00 00 00 00 t.@
00000000:00601030 e6 04 40 00 00 00 00 00 f6 04 40 00 00 00 00 h.@
00000000:00601040 00 00 00 00 02 00 00 00 47 43 43 3a 20 28 55 62 .GCC: (Ub
```

图 5.14

我们的表发生了变化

在之后的函数调用过程中首先跳转到 PLT 执行 `.plt` 中，第一次访问跳转时 GOT 地址为下一条指令，将函数序号压栈，然后跳转到 `PLT[0]`，在 `PLT[0]` 中将重定位表地址压栈，然后访问动态链接器，在动态链接器中使用函数序号和重定位表确定函数运行时地址，重写 GOT，再将控制传递给目标函数。

之后如果对同样函数调用，第一次访问跳转直接跳转到目标函数。

5.8 本章小结

本章中主要介绍链接的概念以及链接的作用，分析了 `hello` 可执行文件的 ELF 格式文件，分析了 `hello` 的重定位过程，执行流程，和动态链接过程。

(第 5 章 1 分)

第 6 章 hello 进程管理

6.1 进程的概念与作用

进程是一个执行中的程序的实例化，每一个进程都有它自己的地址空间，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。但实际上由进程的相互切换和分时处理完成。

6.2 简述壳 Shell-bash 的作用与处理流程

Shell 是一个用 C 语言编写的程序，是用户使用 Linux 系统的桥梁。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：

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

6.3 Hello 的 fork 进程创建过程

在终端 bash 中输入 `./hello 1170300107 YWQ`，运行的终端 bash 解析命令发现不是内置命令，并判断出 `./hello` 其实是当前目录下的可执行文件。所以终端调用 `fork` 函数为其创建新的子进程，并且为它分配与父进程相同的虚拟地址空间。并且两者开始并发的在 CPU 调控下运行。

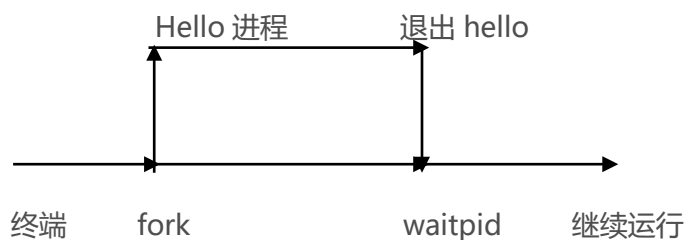


图 6.1

6.4 Hello 的 `execve` 过程

创建子进程 `hello` 之后，子进程使用 `execve` 函数，根据传入的命令行参数，在当前进程的上下文中加载 `hello` 程序。加载过程中，删除子进程现在的虚拟内存，创建一组新的代码段，数据段，堆和栈。将虚拟地址空间中的页映射到可执行文件的页大小。整个加载过程没有从磁盘到内存的数据复制，直到 CPU 引用被映射的虚拟内存页发生缺页时才开始复制。

创建的内存映射如下图

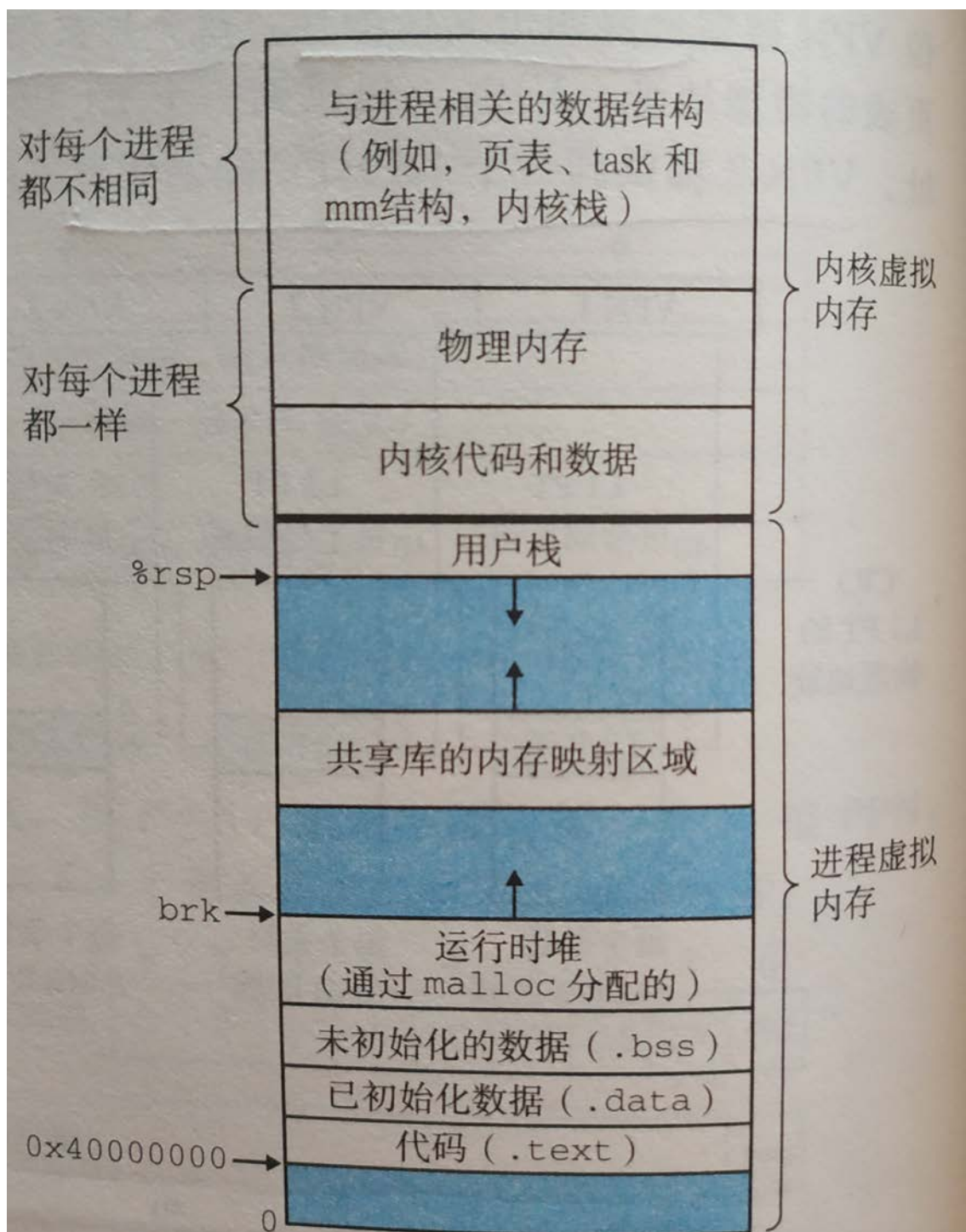


图 6.2

6.5 Hello 的进程执行

(0) 逻辑控制流: 进程是轮流使用处理器的, 在同一个处理器核心中, 每个进程执行一部分后暂时挂起, 然后轮到其他进程。

(1) 时间片: 一个进程执行它的控制流的一部分的每一时间段叫做时间片。

(2) 用户模式和内核模式：处理器通常使用一个寄存器提供两种模式的区分，该寄存器描述了进程当前享有的特权，当没有设置模式位时，进程就处于用户模式中，用户模式的进程不允许执行特权指令，也不允许直接引用地址空间中内核区内的代码和数据；设置模式位时，进程处于内核模式，该进程可以执行指令集中的任何命令，并且可以访问系统中的任何内存位置。

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

对于 `hello` 进程，在 `sleep` 函数之前假设没有被挂起，那么 `sleep` 函数时发生上下文切换。

如图

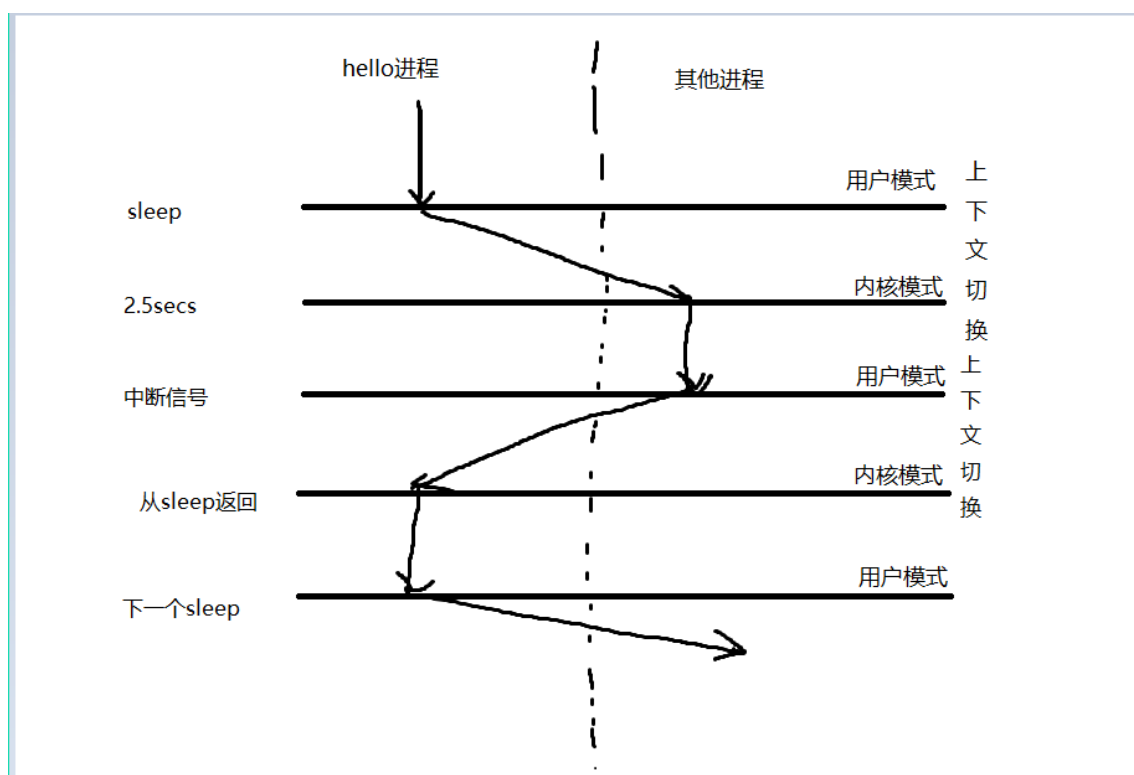


图 6.3

之后 `hello` 调用 `getchar` 时，实际执行输入流 `stdin` 的系统调用 `read`，我们之前在用户模式，在 `read` 调用之后进入内核模式，内核中的陷阱处理程序在安排从键盘缓冲区到内存的数据传输后，进行上下文切换，直到键盘缓冲区到内存传输完成，引发中断信号，内核重新上下文切换回到 `hello`。

6.6 `hello` 的异常与信号处理

0.正常运行，运行完成进程被父进程回收

```

ywq1170300107@ubuntu:~/hitcs/helloproject$ ./hello 1170300107 YWQ
Hello 1170300107 YWQ
Hello 1170300107 YWQ
Hello 1170300107 YWQ
Hello 1170300107 YWQ
fuck
Hello 1170300107 YWQ
Hello 1170300107 YWQ
Hello 1170300107 YWQ
Hello 1170300107 YWQ
Hello 1170300107 YWQ
Hello 1170300107 YWQ
ywq1170300107@ubuntu:~/hitcs/helloproject$ ps
  PID TTY          TIME CMD
  2456 pts/0        00:00:00 bash
  2908 pts/0        00:00:00 vim
  3668 pts/0        00:00:00 ps
ywq1170300107@ubuntu:~/hitcs/helloproject$ jobs
[1]+  已停止                  vim hello.txt

```

图 6.4

1.在程序运行过程中，键盘输入 `ctrl+z`，父进程收到 `SIGSTP` 信号，根据显示的内容，我们推断出，信号处理函数打印信息，挂起 `hello` 进程，然后我们继续使用 `fg` 将这个后台挂起的程序调到前台程序，然后程序就继续执行。结束后仍然被回收。

```

ywq1170300107@ubuntu:~/hitcs/helloproject$ ./hello 1170300107 YWQ
Hello 1170300107 YWQ
Hello 1170300107 YWQ
Hello 1170300107 YWQ
^Z
[2]+  已停止                  ./hello 1170300107 YWQ
ywq1170300107@ubuntu:~/hitcs/helloproject$ ps
  PID TTY          TIME CMD
  2456 pts/0        00:00:00 bash
  2908 pts/0        00:00:00 vim
  3671 pts/0        00:00:00 hello
  3672 pts/0        00:00:00 ps
ywq1170300107@ubuntu:~/hitcs/helloproject$ jobs
[1]-  已停止                  vim hello.txt
[2]+  已停止                  ./hello 1170300107 YWQ
ywq1170300107@ubuntu:~/hitcs/helloproject$ fg 2
./hello 1170300107 YWQ
Hello 1170300107 YWQ
Hello 1170300107 YWQ
Hello 1170300107 YWQ
Hello 1170300107 YWQ
Hello 1170300107 YWQ
Hello 1170300107 YWQ
Hello 1170300107 YWQ
ywq1170300107@ubuntu:~/hitcs/helloproject$ ps
  PID TTY          TIME CMD
  2456 pts/0        00:00:00 bash
  2908 pts/0        00:00:00 vim
  3675 pts/0        00:00:00 ps
ywq1170300107@ubuntu:~/hitcs/helloproject$

```

图 6.5

2.进程运行过程中我们按下 `ctrl+c`，父进程收到 `SIGINT` 信号，直接结束函数，运行 `ps` 发现被回收。

```
ywq1170300107@ubuntu:~/hitcs/helloproject$ ./hello 1170300107 YWQ
Hello 1170300107 YWQ
Hello 1170300107 YWQ
Hello 1170300107 YWQ
Hello 1170300107 YWQ
Hello 1170300107 YWQ
^C
ywq1170300107@ubuntu:~/hitcs/helloproject$ ps
  PID TTY          TIME CMD
 2456 pts/0        00:00:00 bash
 2908 pts/0        00:00:00 vim
 3679 pts/0        00:00:00 ps
ywq1170300107@ubuntu:~/hitcs/helloproject$
```

图 6.6

3.在运行时乱按，实际上是都录入了 `stdin`，`getchar` 会在敲了回车的命令行前面当作它是命令继续执行，剩下的字符串会被忽略掉。

```
ywq1170300107@ubuntu:~/hitcs/helloproject$ ./hello 1170300107 YWQ
Hello 1170300107 YWQ
sadasdasHello 1170300107 YWQ
dassdasdHello 1170300107 YWQ
sddsdsfdaHello 1170300107 YWQ
daHello 1170300107 YWQ
djfkfkaHello 1170300107 YWQ
f
s
ad
a
das
dHello 1170300107 YWQ

sad
as
d
asd
saHello 1170300107 YWQ
Hello 1170300107 YWQ
Hello 1170300107 YWQ
ywq1170300107@ubuntu:~/hitcs/helloproject$ s
s: 未找到命令
ywq1170300107@ubuntu:~/hitcs/helloproject$ ad

Command 'ad' not found, but can be installed with:

sudo apt install netatalk

ywq1170300107@ubuntu:~/hitcs/helloproject$ a
a: 未找到命令
ywq1170300107@ubuntu:~/hitcs/helloproject$ das

Command 'das' not found, did you mean:

  command 'dav' from deb dav-text
  command 'dcs' from deb drbl
  command 'as' from deb binutils
  command 'dat' from deb liballegro4-dev
  command 'dab' from deb bsdgames
  command 'dar' from deb dar
  command 'dds' from deb dds
  command 'dash' from deb dash
  command 'dan' from deb emboss
  command 'cas' from deb amule-utils
  command 'dad' from deb debian-dad

Try: sudo apt install <deb name>
```

图 6.7

hello 执行过程中会出现哪几类异常，会产生哪些信号，又怎么处理的。

6.7 本章小结

本章中，阐明了进程的定义以及作用，介绍 shell 的用处，如何调用 fork，如何使用 execve，如何上下文切换，hello 的异常信号处理。

(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

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

1.逻辑地址：程序代码编译后汇编程序中地址。逻辑地址由选择符和偏移量组成。hello 中可以看到这个地址 `00000000000400530`

2.线性地址：逻辑地址经过段机制后转化为线性地址，为描述符:偏移量的组合形式。分页机制中线性地址作为输入

3.虚拟地址这里就是指的线性地址

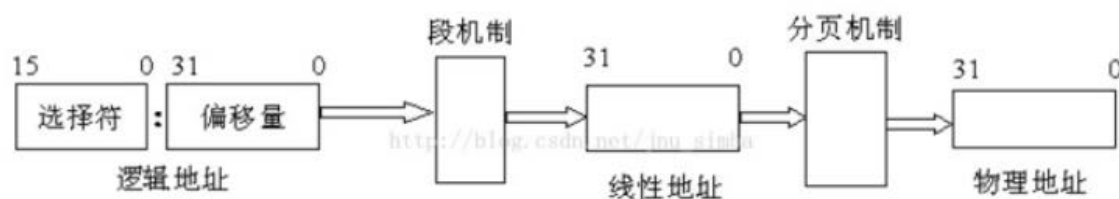


图 7.1

附上一张网上找到的图片

7.2 Intel 逻辑地址到线性地址的变换-段式管理

Intel 的地址转化是涉及原先的历史。

0.实模式下，逻辑地址=实际物理地址，段寄存器存放真实的段基址同时给出 32 位地址偏移量就可以访问真实的物理内存

1.保护模式，线性地址需要经过分页机制才能得到物理地址，线性地址也需要逻辑地址通过段机制得到。段寄存器无法放下 32 位段基址，所以他们用于引用段描述符中的表项来获得描述符，描述符表中的一个条目描述成一个段。

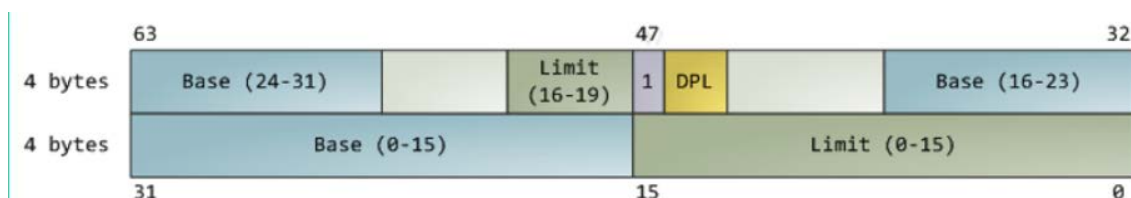


图 7.2

段选择符如下构造

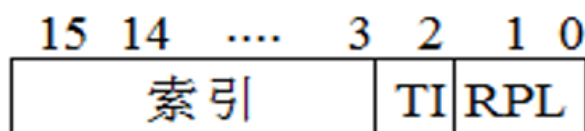


图 7.3

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

7.3 Hello 的线性地址到物理地址的变换-页式管理

虚拟地址（线性地址）到物理地址之间的转化通过分页机制。

Linux 的虚拟内存组织如图所示。

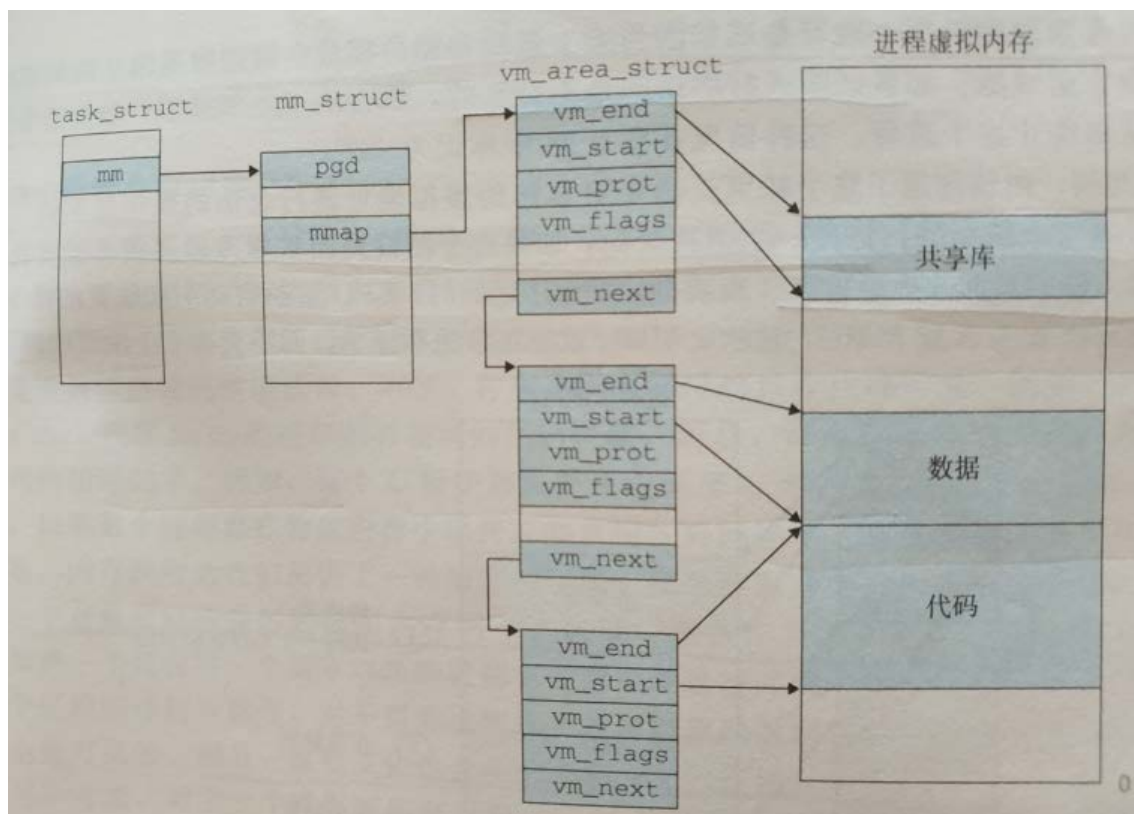


图 7.4

Linux 将虚拟内存组织成段的集合，段之外的虚拟内存不需要记录，只需要记录段内的，因为他们才有意义。内核给 hello 进程维护一个段的任务结构（task_struct），mm 条目指向 mm_struct，描述的是虚拟内存的状态，pgd 是第一级页表的首地址（用到多级页表），mmap 指向 vm_area_struct 的链表，一个链表的条目对应一个段，链表的所有节点指出了 hello 进程的虚拟内存中的所有的段，例如共享库，数据，代码段。

每个段被分割成一个个块，在虚拟内存管理中称为页。物理内存被分割成 PP

物理页，虚拟内存系统中的硬件单元 MMU 负责地址翻译，使用页表将虚拟页映射到物理页。

虚拟地址在不考虑多级页表和 TLB 的情况下简单的分为两部分，虚拟页号 VPN 和虚拟页偏移量 VPO。通过页表基值寄存器获得条目 PTE，PTE 中包含了物理页号 PPN 和虚拟页偏移量共同构成 PA 物理地址。

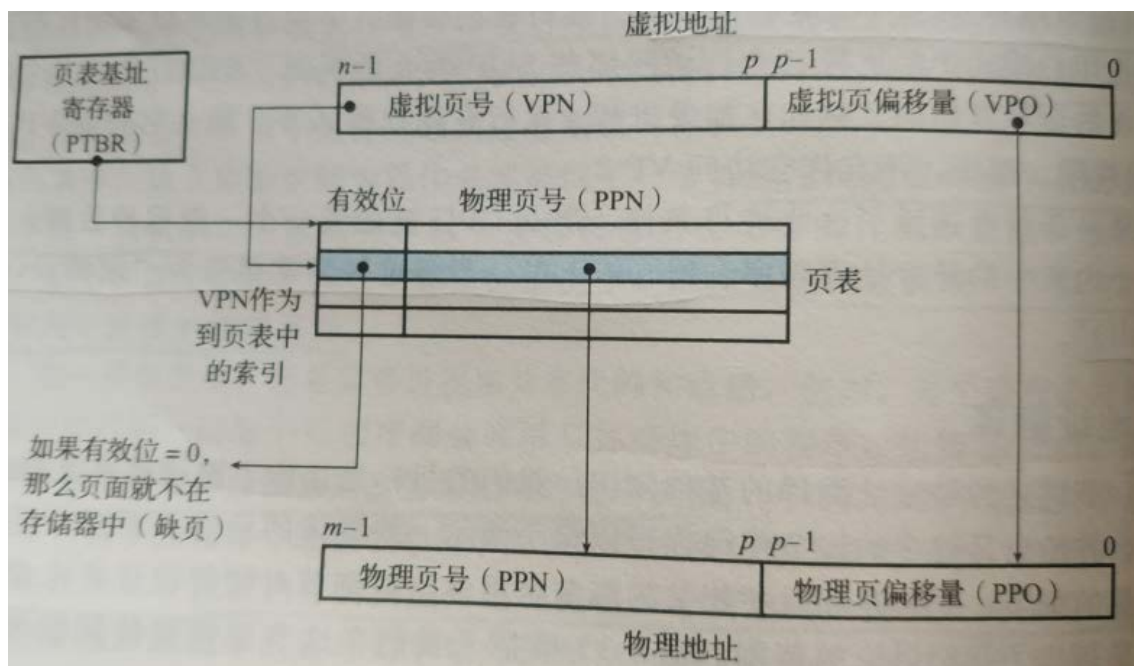


图 7.5

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

根据 CSAPP 内容，我们在 Intel core i7 条件下研究 VA 到 PA 的过程。

前提：一个页表大小 4KB，PTE 条目 8B，所以有 512 条目，使用 9 位二进制索引，4 个页表所以使用 36 位索引，VPN36 位。VA 公共 48 位，VPO = 48 - 36 = 12 位。TLB 一共 16 组，所以 TLBI 需要 4 位作为索引，TLBT 就 32 位

如图 VA 产生之后，VA 传给 MMU，MMU 首先尝试 TLB，如果 TLBI 和 TLBT 命中，我们直接得到 PPN 和 VPO 组合就得到了 PA。

如果 TLB 没有命中，MMU 向页表查询，页表基值寄存器 CR3 确定第一级页表地址，VPN1（第一部分）确定第一级页表的偏移量查出第一级 PTE，如果在物理内存中且有相应的权限，则继续查询，直到第四级页表查询到 PPN，组成 PA 后利用相应的牺牲规则向 TLB 中添加条目

如果 PTE 不在物理内存，引发缺页故障，如果权限不足则引发段错误

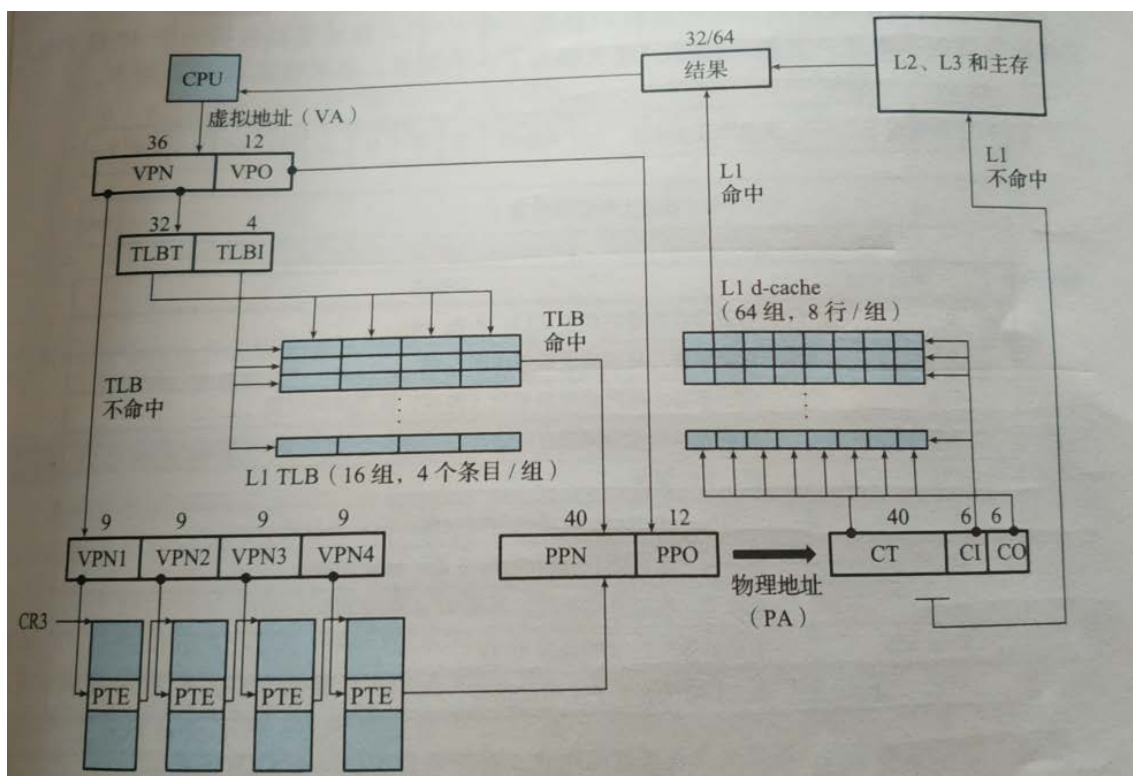


图 7.6

7.5 三级 Cache 支持下的物理内存访问

L1, L2, L3 本质相同，所以我们以 L1 为例子。

在上图中我们看到 L1 cache 有 64 组，所以我们需要 6 位来组寻址，块大小 64B 所以需要 6 位来表示块偏移，PA 一共 52 位，所以 CT 一共 40 位，如上图所示

首先我们根据 CI 进行组索引，然后利用 CT 行匹配，如果匹配成功且有效位为 1，命中，根据 CO 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 0，则不命中，向下一级缓存中查询数据。

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，则采用最近最少使用策略 LRU 进行替换。

7.6 hello 进程 fork 时的内存映射

Fork 函数被 shell 调用后，内核为新进程创建虚拟内存，分配 mm_struct、区域结构以及页表。它最初将进程页面标记位只读，将进程区域标记位私有的写时复制。

7.7 hello 进程 execve 时的内存映射

Execve 函数使用加载器加载代码，在当前进程加载并运行可执行目标文件 hello 中的程序，用 hello 代替当前程序。

加载过程分为：(0) 删除当前存在的用户区域 (1) 内存映射：包括映射私有区域以及映射共享区域 (3) 设置程序计数器，让 rip 指向程序代码入口

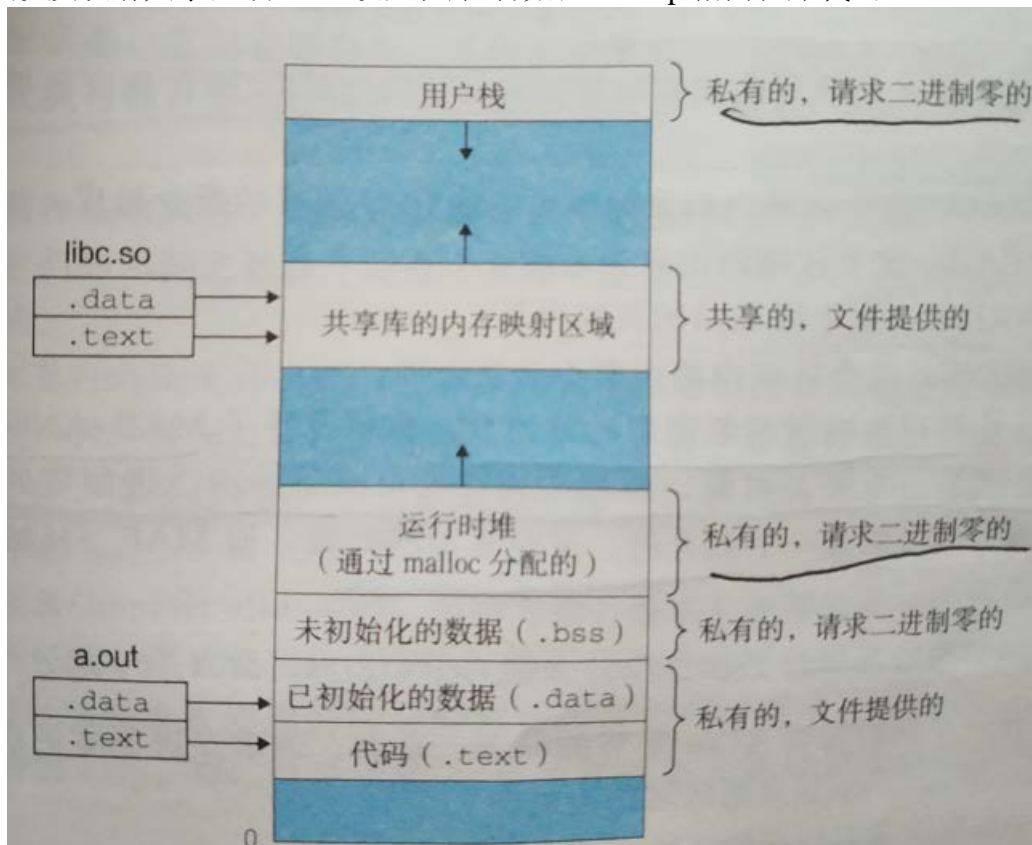


图 7.7

(图中的 a.out 在这里应该是 hello.out)

7.8 缺页故障与缺页中断处理

缺页故障指的是指令引用的虚拟地址 MMU 查找发现与该地址对应的物理地址不在内存，必须从磁盘中取出。

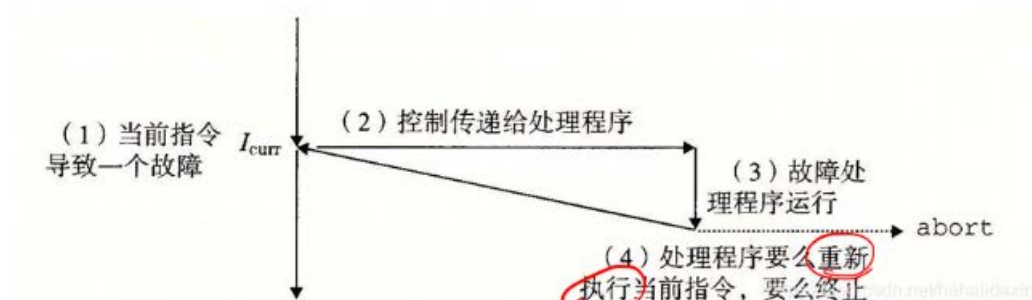


图 7.7

缺页中断处理：选择一个牺牲页，如果这个页面被修改过，先要将它交换出去保存，然后换入新页并更新页表。缺页处理正常返回到刚才引起缺页故障的命令重新执行。

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。空闲块可以用来分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，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。

分配器分为显示分配器和隐式分配器

显式分配器：要求应用显式地释放任何已分配的块。

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

先谈谈隐式空闲链表：

0.内存块结构

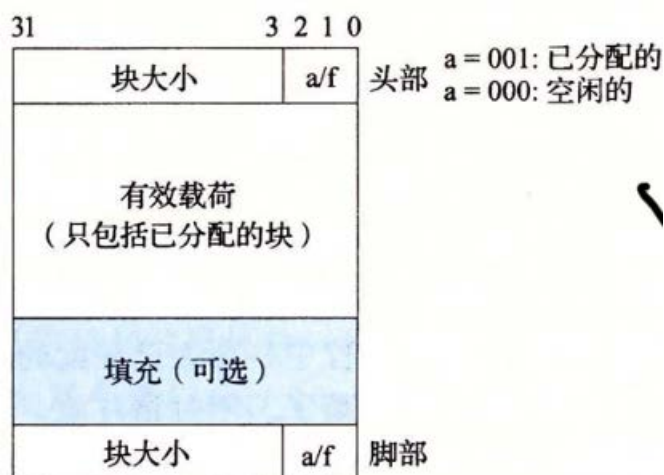


图 7.7

内存块中含有 4B 的头部和 4B 的脚部。脚部的是为了合并空闲块方便寻找上一个块的 alloc 状态设计的。

1.隐式链表

不单独为空闲块开辟链表，对内存空间所有块组织成大链表，利用头部脚部来进行前后搜索

2.空闲块合并

利用脚部可以查看前块是不是空闲，利用头部查看后块是不是空闲。在 coalesce 函数中堆四种情况分别处理进行合并（AFA,FFA,AFF,FFF，F for free A for alloc）

再来说说显示链表

0.块结构

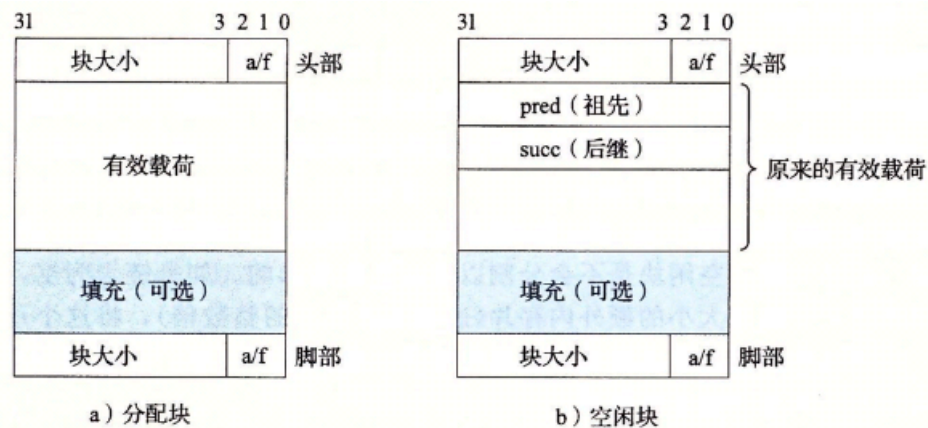


图 7.8

在空闲块中设置前驱和后继指针，指向前一个/后一个空闲块

这样子，首次适配的分配时间从块总数的线性时间减少到了空闲块数量的线性时间。维护链表的顺序可以使用 LIFO 或者按照地址顺序来维护链表。

7.10 本章小结

本章介绍了 hello 的存储器地址空间，intel 的段式管理，以及页式管理，讲述了 VA 到 PA 的转换过程，内存映射的过程，缺页故障和缺页中断处理，动态存储的分配管理

(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

Unix I/O 接口统一操作：

0. 打开文件。程序通过要求内核打开相应的文件，来宣告它想访问一个 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。

1. Shell 创建的每个进程都有三个打开的文件：标准输入 `stdin`，标准输出 `stdout`，标准错误 `stderr`。

2. 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`，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` 函数，显式地将改变当前文件位置指针 `k`。

3. 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，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，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`。

4. 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。

Unix I/O 函数：

0. `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。

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

2 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。

3 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

先分析 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 获得第二个不定长的参数, 也就是输出的时候格式控制的参数值

对于 printf 调用的 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 != '%') //忽略无关字符

        {

            *p++ = *fmt;

            continue;

        }

        fmt++;

        switch (*fmt)

        {
```

```

        case 'x':    //只处理%x 一种情况

            itoa(tmp, *((int*)p_next_arg)); //将输入参数值转化为字符串保存在
tmp

            strcpy(p, tmp); //将 tmp 字符串复制到 p 处

            p_next_arg += 4; //下一个参数值地址

            p += strlen(tmp); //放下一个参数值的地址

            break;

        case 's':

            break;

        default:

            break;

    }

}

return (p - buf); //返回最后生成的字符串的长度

}

```

Vsprintf 程序利用格式 fmt 结合参数 args 生成一个字符串，并且返回字符串的长度，从而将这两个参数传递给 write 函数输出。

Write 函数是汇编代码

write:

```

    mov eax, _NR_write
    mov ebx, [esp + 4]

```

```
mov ecx, [esp + 8]
int INT_VECTOR_SYS_CALL
```

INT_VECTOR_SYS_CALLA 代表通过系统调用 syscall，查看 syscall 的实现，syscall 将字符串中的字节从寄存器中复制到显卡的显存中。字符显示驱动子程序将 ASCII 在字模库中找到，将点阵信息输入到 vram，然后显示芯片按一定频率刷新读取 vram，向液晶显示器传输 RGB 分量表示的点。

8.4 getchar 的实现分析

异步异常-键盘中断的处理：用户按键盘的操作，让键盘的接口得到编码，产生一个中断请求，抢占当前进程的运行，执行上下文切换到键盘中断子程序，键盘中断子程序将编码转化成 ASCII 码，保存到系统的键盘缓冲区

getchar 等调用 read 系统函数，通过系统调用读取按键 ascii 码，直到接受到回车键才返回整个字符串，然后进行重新封装。

8.5 本章小结

本章介绍了 Linux 的 IO 设备管理，Unix IO 以及其函数，深入分析了两个常见函数 printf 和 getchar 与 IO 的关系

（第 8 章 1 分）

结论

Hello 的 P2P 和 020 历程经历是 CSAPP 这整本书才能讲的完的。具体的纪年表：

0.某个程序员开始敲代码，通过 vim/gedit/sublime text/codebloks……等编辑器将代码敲入.c 文件

1.预处理器将所有.c 中调用的外部库展开并插入到文件中形成.i 文件

2.编译器将 hello.i 编译成汇编代码 hello.s，保存汇编代码的助记符

3.汇编器将 hello.s 中的汇编代码翻译成 hello.o，将助记符全部改写成具体代码，保留重定位信息。

4.链接器将可重定位文件 hello.o 和动态链接库链接，进行重定位。生成可执行目标文件 hello

5.在 shell 中我们输入了 ./hello 1170300107 YWQ

6.fork 函数让内核为其创建子进程

7.shell 调用 execve，利用加载器，将虚拟内存映射，利用动态库的启动函数进入 main 函数

- 8.CPU 为其分配逻辑控制流，允许其在自己的时间片内执行
 - 9.MMU 将程序中使用的虚拟内存地址通过页表映射到相应的物理地址
 - 10.printf 向动态内存分配器申请堆中的存储空间
 - 11.有可能用户通过键盘键入控制信号，也有可能缺页信号让进程暂时挂起或者停止
 - 12.程序结束，父进程回收子进程，内核删除这个进程的虚拟内存的数据结构
- (结论 0 分，缺失 -1 分，根据内容酌情加分)**

附件

hello hello 的可执行目标文件
hello.c 原始代码
hello.elf hello 的 elf 文件
hello.i hello.c 经过预处理的文件
hello.o hello 的可重定位文件
helloo.elf hello.o 的 elf 文件
helloo.txt hello 的反汇编代码
hello.s hello 的汇编代码
hello.txt hello.o 的反汇编代码

(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] 深入理解计算机系统 Randal E.Bryant David R.O'Hallaron 机械工业出版社
- [2] 维基百科 预处理
- [3] 维基百科 ELF 文件
- [4] Pianistx. [转]printf 函数实现的深入剖析.
<https://www.cnblogs.com/pianist/p/3315801.html>. 2013-09-11.
- [5] ZK 的博客.read 和 write 系统调用以及 getchar 的实现.
<https://blog.csdn.net/ww1473345713/article/details/51680017>. 2016-6-15.

(参考文献 0 分, 缺失 -1 分)