

哈爾濱工業大學

計算機系統

大作業

題 目 程序人生-Hello's P2P

專 業 計算機科學與技術

學 號

班 級

學 生

指 導 教 師

計算機科學與技術學院

2018 年 12 月

摘 要

本文是在 Linux 环境下，对 `hello.c` 这一简单的源代码文件通过编译器编译生成可执行文件，然后在系统中执行完毕的整个过程进行了阐述。

在程序整个生命进程中，本文介绍了与程序执行相关的编译器、系统方方面面的功能和作用，阐述了系统支持进程执行的机制。包括编译器对源代码进行的预处理、编译、汇编、链接，程序运行时，系统中提供的进程管理、储存管理和 IO 管理等。对计算机系统课程内容有比较强的概括作用。

关键词：操作系统、编译器

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	- 7 -
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	- 12 -
第 4 章 汇编	- 13 -
4.1 汇编的概念与作用	- 13 -
4.2 在 UBUNTU 下汇编的命令	- 13 -
4.3 可重定位目标 ELF 格式	- 13 -
4.4 HELLO.O 的结果解析	- 16 -
4.5 本章小结	- 16 -
第 5 章 链接	- 17 -
5.1 链接的概念与作用	- 17 -
5.2 在 UBUNTU 下链接的命令	- 17 -
5.3 可执行目标文件 HELLO 的格式	- 18 -
5.4 HELLO 的虚拟地址空间	- 19 -
5.5 链接的重定位过程分析	- 20 -
5.6 HELLO 的执行流程	- 21 -
5.7 HELLO 的动态链接分析	- 21 -
5.8 本章小结	- 22 -
第 6 章 HELLO 进程管理	- 23 -
6.1 进程的概念与作用	- 23 -

6.2 简述壳 SHELL-BASH 的作用与处理流程.....	- 23 -
6.3 HELLO 的 FORK 进程创建过程	- 23 -
6.4 HELLO 的 EXECVE 过程	- 24 -
6.5 HELLO 的进程执行.....	- 24 -
6.6 HELLO 的异常与信号处理	- 25 -
6.7 本章小结	- 28 -
第 7 章 HELLO 的存储管理.....	- 29 -
7.1 HELLO 的存储器地址空间	- 29 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理.....	- 29 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理	- 30 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换.....	- 31 -
7.5 三级 CACHE 支持下的物理内存访问	- 32 -
7.6 HELLO 进程 FORK 时的内存映射	- 33 -
7.7 HELLO 进程 EXECVE 时的内存映射	- 34 -
7.8 缺页故障与缺页中断处理.....	- 34 -
7.9 动态存储分配管理	- 34 -
7.10 本章小结	- 35 -
第 8 章 HELLO 的 IO 管理	- 36 -
8.1 LINUX 的 IO 设备管理方法	- 36 -
8.2 简述 UNIX IO 接口及其函数	- 36 -
8.3 PRINTF 的实现分析.....	- 37 -
8.4 GETCHAR 的实现分析.....	- 40 -
8.5 本章小结	- 40 -
结论	- 41 -
附件	- 42 -
参考文献.....	- 43 -

第 1 章 概述

1.1 Hello 简介

首先在编辑器(Editor)中编辑生成源代码文件 `hello.c` (program)。使用编译器套件对它进行预处理(Cpp)、编译(Compiler)、汇编(AS)、链接(LD)之后生成可执行程序 `hello`。

在 shell 中运行 `hello`，在 OS 中为运行此程序，使用 `fork` 新建进程，使用 `execve` 加载可执行程序，为 `hello` 的代码、数据、栈等区域分配内存区域，CPU 通过虚拟地址利用 MMU 从页表中获取实际的物理地址，访问到相应的代码和数据，调整程序计数器之后从代码的入口进入 `main` 函数，CPU 通过一定的控制逻辑执行代码，IO 管理控制程序的屏幕输出。最终 shell 回收 `hello` 进程。从内核中删除 `hello` 的痕迹。

1.2 环境与工具

1.2.1 硬件环境

虚拟机内存：3GB

处理器类型：CORE i7-6700HQ 2.60GHz

虚拟机处理器内核总数：2

虚拟机硬盘：20GB HDD

1.2.2 软件环境

VMware14

Ubuntu-16.04.4-desktop-amd64

1.2.3 开发工具

Windows 下：CodeBlocks

Linux 下：Vim+gcc

1.3 中间结果

列出你为编写本论文，生成的中间结果文件的名字，文件的作用等。

hello.i	预处理生成
hello.s	编译生成
hello.o	汇编生成
hello_obj.s	hello.s 反汇编生成
hello.elf	hello.o 的 ELF 格式
hello_out.elf	hello 的 ELF 格式
hello_out_obj.s	hello 反汇编生成
hello	可执行文件

1.4 本章小结

本章概述了 hello 程序从生成源代码，到在系统中执行完毕的整个 P2P 过程。

第 2 章 预处理

2.1 预处理的概念与作用

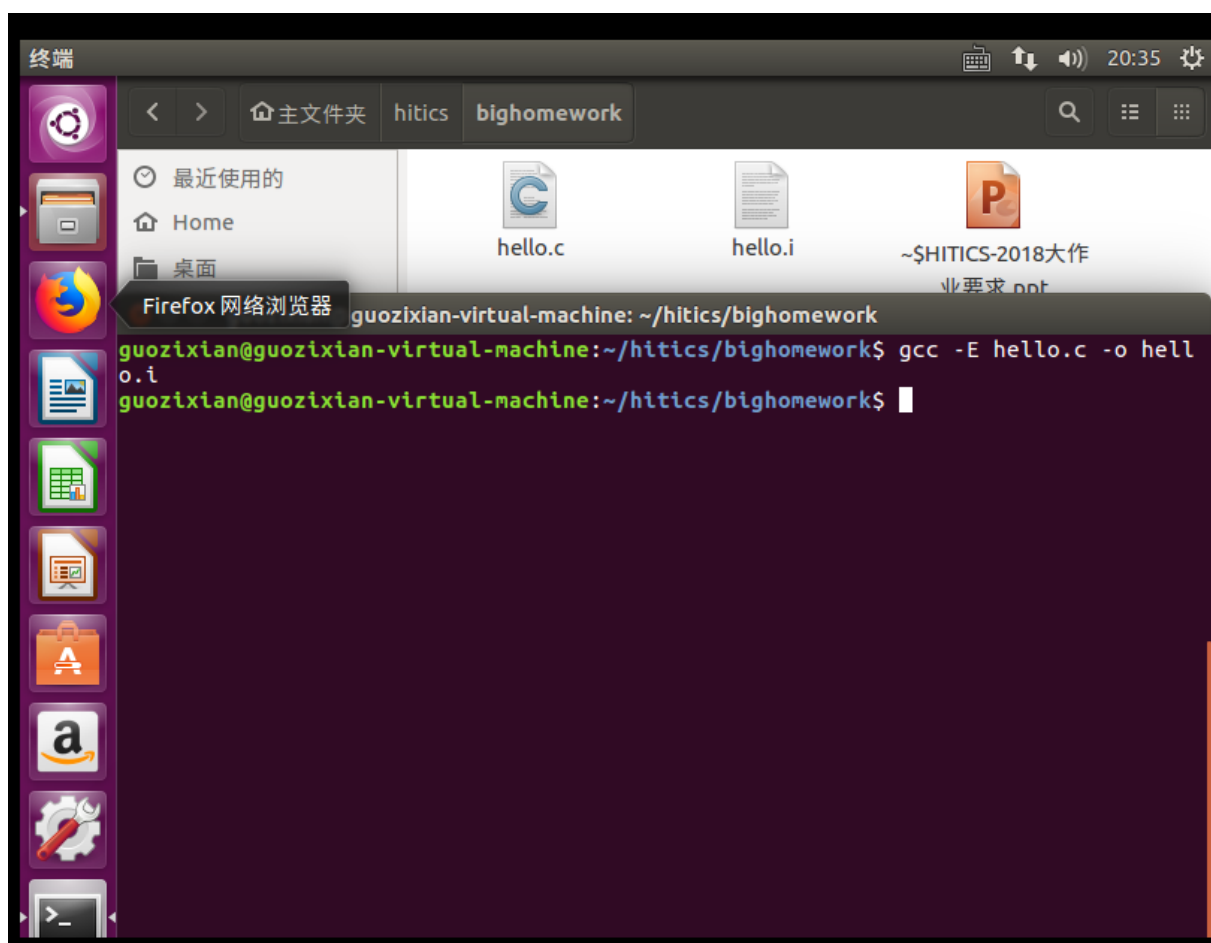
预处理过程会读入程序源代码，检查包含的预处理指令和宏定义，会对代码进行一定的转换，删除注释。

预处理主要处理程序中宏定义、文件包含、条件编译的问题。

`#include` 命令会使预处理器读取系统头文件，直接插入程序文本，`#if` 命令使编译器选择编译某些程序段，`#define` 命令使编译器进行宏替换，预处理得到新的 C 程序文件，`hello.i`

2.2 在 Ubuntu 下预处理的命令

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



2.3 Hello 的预处理结果解析

使用 vim 查看 `hello.i`，在最后找到了 `hello.c` 文件中的部分，可以看出删去了注释和 `#include` 语句，而整个程序文件已经 3126 行，远多于原文件，`#include` 使预处理器将目标文件全部插入到了 `hello.c` 文件开头。

2.4 本章小结

预处理将程序要引用的头文件插入代码文件中，使得代码真正“完整”，并且删去了没用的注释部分。整理之后的代码文件进行下一步处理。

第 3 章 编译

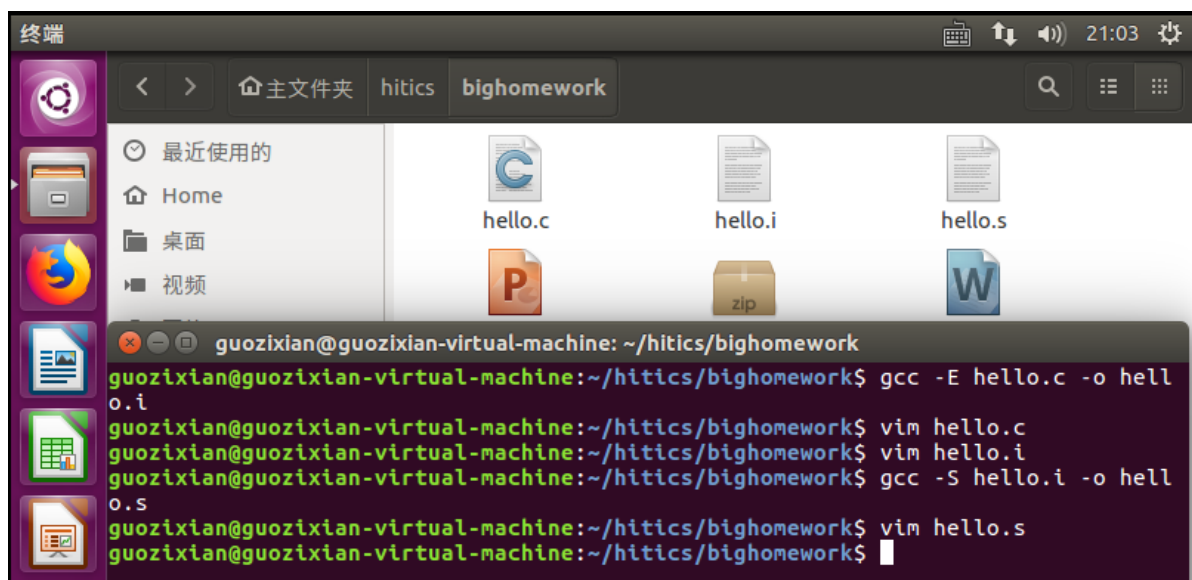
3.1 编译的概念与作用

编译是将预处理之后的 C 语言程序经过词法分析，语法分析，语义分析 转化为汇编语言程序的过程，但实际的编译系统处理基本功能之外，还具备调试措施、程序优化等功能。

（注意：这儿的编译是指从 .i 到 .s 即预处理后的文件到生成汇编语言程序）

3.2 在 Ubuntu 下编译的命令

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



3.3 Hello 的编译结果解析

3.3.1 汇编文件中的指令：

指令	含义
.file	文件名
.global	全局变量
.data	可读写数据（已初始化）
.align	对齐方式
.type	数据类型

.size	数据大小
.long	定义长整型
.rodata	只读数据
.string	定义字符串
.text	可执行代码

3.3.2 数据

1、字符串

```

9      .section    .rodata
10     .LC0:
11     .string "Usage: Hello \345\255\246\345\217\267 \345\247\223\345\220\215\
12     .LC1:
13     .string "Hello %s %s\n"

```

开头的 section、rodata 标明为字符串常量。

第一个字符串为“Usage: Hello 学号 姓名! \n”，对汉字使用了 utf-8 编码，三个字节表示一个汉字。

第二个字符串为“Hello %s %s\n”，为 printf 格式化输出的字符串。

2、整型

```

2      .globl    sleepsecs
3      .data
4      .align    4
5      .type     sleepsecs, @object
6      .size     sleepsecs, 4
7  sleepsecs:
8      .long     2

```

源代码中定义了全局变量 sleepsecs，汇编文件中相关声明如上图，汇编文件将赋初值的全局变量放置在.data 区。

其中指明了变量为全局变量，对齐方式和大小都是 4 字节，并且将整型当做 long 型处理。

局部整型 i：局部变量被存储在栈中，从循环控制部分可以看出 i 存储在-4(%rbp) argc 是第一个参数，保存在寄存器%rdi 中，程序一开始就将其转存如-20(%rbp) 立即数，汇编代码中临时使用的数据

3.3.3 赋值

源代码对整型的 sleepsecs 赋了初值 2.5，hello.s 文件中可以看出进行了向下取整，

实际存入的值为 2

整型变量 i 的赋值使用汇编 `movl` 指令，1 指明操作数为 4 字节

3.3.4 类型转换

`int sleepsecs = 2.5` 此语句涉及隐式类型转换，2.5 默认是 `double` 类型的数据，浮点数向整型转化的默认操作是舍去小数点之后的部分，所以直接得到 2.

3.3.5 算数操作

指令	操作
<code>leaq a(b,c,d), e</code>	<code>e=a+b+c*d</code>
<code>add a, b</code>	<code>b=b+a</code>
<code>inc a</code>	<code>a++</code>
<code>sub a, b</code>	<code>b=b-a</code>
<code>dec a</code>	<code>a--</code>
<code>IMULQ S</code>	<code>R[%rdx]:R[%rax]=S*R[%rax]</code> (有符号)
<code>MULQ S</code>	<code>R[%rdx]:R[%rax]=S*R[%rax]</code> (无符号)
<code>IDIVQ S</code>	<code>R[%rdx]=R[%rdx]:R[%rax] mod S</code> (有符号) <code>R[%rax]=R[%rdx]:R[%rax] div S</code>
<code>DIVQ S</code>	<code>R[%rdx]=R[%rdx]:R[%rax] mod S</code> (无符号) <code>R[%rax]=R[%rdx]:R[%rax] div S</code>

hello.s 中涉及的算术操作有：

地址计算用到加减：`addq, subq`

i 循环增 1：`addl`

3.3.6 关系操作

此代码中用到的关系操作：

`argc!=3`：汇编代码为 `cmpl $3, -20(%rbp)`

使用 `cmpl` 比较，设置条件码，并根据条件码进行下一步的跳转。

`i<10`：汇编代码 `cmpl $9, -4(%rbp)` 同理

3.3.7 控制转移

此代码中涉及的控制转移：

1、`if(argc!=3)`

如果等于 3，则跳转到 L2，程序正常执行，如果不等于 3，则调用 `exit` 返回 1

```
26     movl    %edi, -20(%rbp)
27     movq    %rsi, -32(%rbp)
28     cmpl    $3, -20(%rbp)
29     je      .L2
30     movl    $.LC0, %edi
31     call    puts
32     movl    $1, %edi
33     call    exit
34 .L2:
35     movl    $0, -4(%rbp)
36     jmp     .L3
```

2、for(i=0;i<10;i++)

i 从 0 到 9 循环，满足条件时继续循环，否则结束循环，注意到汇编将 i<10 处理成 i<=9，使用 jle 进行条件跳转。

```
34 .L2:
35     movl    $0, -4(%rbp)
36     jmp     .L3
37 .L4:
38     movq    -32(%rbp), %rax
39     addq    $16, %rax
40     movq    (%rax), %rdx
41     movq    -32(%rbp), %rax
42     addq    $8, %rax
43     movq    (%rax), %rax
44     movq    %rax, %rsi
45     movl    $.LC1, %edi
46     movl    $0, %eax
47     call    printf
48     movl    sleepsecs(%rip), %eax
49     movl    %eax, %edi
50     call    sleep
51     addl    $1, -4(%rbp)
52 .L3:
53     cmpl    $9, -4(%rbp)
54     jle     .L4
```

3.3.8 数组操作

程序中存在字符数组 argv，在调用 printf 之前将两个字符串当做传入参数。

argv 是二维字符数组的指针，所以 argv 指向的为第一个字符串地址，下一个地址为第二个字符串的地址。程序中将两个字符串地址分别存入 rdx 和 rsi 中，然后调用 printf

```
.L4:
    movq    -32(%rbp), %rax
    addq    $16, %rax
    movq    (%rax), %rdx
    movq    -32(%rbp), %rax
    addq    $8, %rax
    movq    (%rax), %rax
    movq    %rax, %rsi
    movl    $.LC1, %edi
    movl    $0, %eax
    call    printf
```

3.3.9 函数操作

调用函数涉及的操作有：

将调用函数返回地址入栈，调整 `rip` 为被调函数入口，将调用函数 `rbp` 入栈，调整 `rsp`，`rbp` 建立被调函数栈帧，构建出局部变量、传入参数的储存空间。函数执行完之后调整栈帧，`pop` 出调用函数的 `rbp`，利用返回地址回到调用函数的下一条语句。汇编代码 `leave` 可以一并完成恢复 `rsp`，出栈 `rbp` 的操作。

`hello.c` 中涉及到的函数：

1、`main` 函数：由内核中特殊的启动例程调用，从内核中获取命令行参数作为 `main` 函数的输入，在整个程序的开始和结束部分可以看到上述的栈帧变化。

2、`printf` 函数：当命令行输入参数错误时将 `%rdi` 设置为字符串常量“Usage: Hello 学号 姓名! \n”的地址，然后调用 `puts`。命令行输入正确时，传入三个参数，字符串常量“Hello %s %s\n”和 `argv` 中的两个字符串，然后调用 `printf`。

3、`exit` 函数：当命令行参数错误时，给 `%edi` 赋值为 1，调用函数。

4、`sleep` 函数：将 `%edi` 赋值为 `sleepsecs`，然后调用函数。

5、`getchar` 函数，直接调用，没有参数。

3.4 本章小结

本章介绍了编译器将 `hello.i` 文件转化为 `hello.s` 的过程，程序从高级语言转化为了更低一级的汇编语言。并且解释了 `hello.s` 中对 C 语言程序中的数据、计算、过程等各个方面的处理方式。

第 4 章 汇编

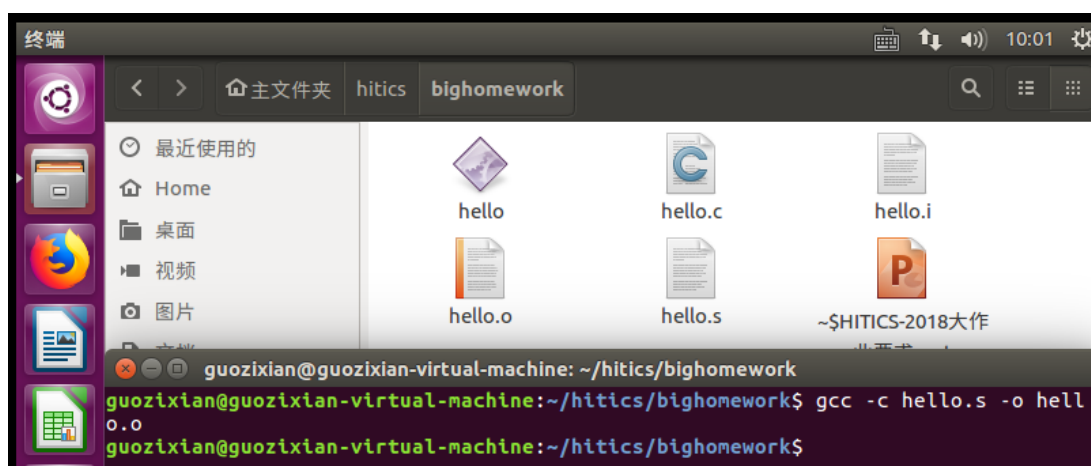
4.1 汇编的概念与作用

汇编是将汇编语言转化成机器语言的过程，汇编器将 `hello.s` 中的汇编语言翻译成机器码，然后将这些指令打包成可重定位目标程序，保存在 `hello.o` 文件中。

（注意：这儿的汇编是指从 `.s` 到 `.o` 即编译后的文件到生成机器语言二进制程序的过程。）

4.2 在 Ubuntu 下汇编的命令

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



应截图，展示汇编过程！

4.3 可重定位目标 elf 格式

使用 `readelf -a hello.o > hello.elf` 生成 `hello.o` 文件的 ELF 格式

1、ELF 头

ELF 头以一个 16 字节的序列开始，这个序列描述了生成该文件的系统的字的大小和字节顺序。ELF 头剩下的部分包含帮助链接器语法分析和解释目标文件的信息。其中包括 ELF 头的大小、目标文件的类型（如可重定位、可执行或共享的）、机器类型（如 x86-64）、节头部表（section header table）的文件偏移，以及节头部表中条目的大小和数量。不同节的位置和大小是由节头部表描述的，其中目标文件中每个节都有一个固定大小的头目（entry）。`hello.o` 的 ELF 头如下：

ELF 头:

```

Magic:      7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
类别:                               ELF64
数据:                               2 补码, 小端序 (little endian)
版本:                               1 (current)
OS/ABI:                               UNIX - System V
ABI 版本:                               0
类型:                               REL (可重定位文件)
系统架构:                               Advanced Micro Devices X86-64
版本:                               0x1
入口点地址:                               0x0
程序头起点:                               0 (bytes into file)
Start of section headers:               1112 (bytes into file)
标志:                               0x0
本头的大小:                               64 (字节)
程序头大小:                               0 (字节)
Number of program headers:               0
节头大小:                               64 (字节)
节头数量:                               13
字符串表索引节头: 10

```

2、节头部表

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]	.text 000000000000007d	PROGBITS 0000000000000000	0000000000000000 AX	0	0	1
[2]	.rela.text 00000000000000c0	RELA 0000000000000018	0000000000000000 I	11	1	8
[3]	.data 0000000000000004	PROGBITS 0000000000000000	0000000000000000 WA	0	0	4
[4]	.bss 0000000000000000	NOBITS 0000000000000000	0000000000000000 WA	0	0	1
[5]	.rodata 000000000000002b	PROGBITS 0000000000000000	0000000000000000 A	0	0	1
[6]	.comment 0000000000000036	PROGBITS 0000000000000001	0000000000000000 MS	0	0	1
[7]	.note.GNU-stack 0000000000000000	PROGBITS 0000000000000000	0000000000000000	0	0	1
[8]	.eh_frame 0000000000000038	PROGBITS 0000000000000000	0000000000000000 A	0	0	8
[9]	.rela.eh_frame 0000000000000018	RELA 0000000000000018	0000000000000000 I	11	8	8
[10]	.shstrtab 0000000000000061	STRTAB 0000000000000000	0000000000000000	0	0	1
[11]	.symtab 0000000000000180	SYMTAB 0000000000000018	0000000000000000	12	9	8
[12]	.strtab 0000000000000037	STRTAB 0000000000000000	0000000000000000	0	0	1

Key to Flags:

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

节头部表指出了各个节的名称、类型、地址、大小等信息。

3、重定位节

重定位节 `'.rela.text'` 位于偏移量 `0x318` 含有 8 个条目：

偏移量	信息	类型	符号值	符号名称 + 加数
000000000016	00050000000a	R_X86_64_32	0000000000000000	.rodata + 0
00000000001b	000b00000002	R_X86_64_PC32	0000000000000000	puts - 4
000000000025	000c00000002	R_X86_64_PC32	0000000000000000	exit - 4
00000000004c	00050000000a	R_X86_64_32	0000000000000000	.rodata + 1e
000000000056	000d00000002	R_X86_64_PC32	0000000000000000	printf - 4
00000000005c	000900000002	R_X86_64_PC32	0000000000000000	sleepsecs - 4
000000000063	000e00000002	R_X86_64_PC32	0000000000000000	sleep - 4
000000000072	000f00000002	R_X86_64_PC32	0000000000000000	getchar - 4

重定位节 `'.rela.eh_frame'` 位于偏移量 `0x3d8` 含有 1 个条目：

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

`.rel.text` 记录了一个 `.text` 节中位置的列表，当链接器把这个目标文件和其他文件组合时需要修改这些位置。一般而言，任何调用外部函数或者引用全局变量的指令都需要修改。另一方面，调用本地函数的指令则不需要修改。

`hello.o` 的重定位节中，记录了 `main` 函数调用的函数 `puts`, `exit`, `printf`, `sleep`, `getchar`，和全局变量 `sleepsecs`，`.rodata` 节中字符串常量，`rela.eh_frame` 记录了 `.text` 的位置、信息、类型、符号值、符号名称和加数。

ELF 定义了 32 种不同的重定位类型，`hello.o` 中出现了两个：`R_X86_64_32` 和 `R_X86_64_PC32`

`R_X86_64_PC32` 的重定位方式是使用目标位置与 PC 地址的相对地址进行调用，PC 是程序运行过程中下一条指令的地址。

`R_X86_64_32` 的重定位方式是使用绝对地址作为寻址方式，CPU 直接使用指令中的 32 位地址访问存储器。

以 PC 相对寻址方式，计算重定位目标地址的方式：

1. `refptr = s + r.offset;`
 2. `refaddr = ADDR(s) + r.offset;`
 3. `*refptr = (unsigned)(ADDR(r.symbol) + r.addend - refaddr);`
- 之后将 `*refptr` 的值作为偏移量加到当前 PC 的值上即可得到目标地址。

4、符号表

`.symtab` 是符号表，其中存放着程序中定义和引用函数和全局变量的信息，不包含局部变量的条目。`hello.elf` 的 `.symtab` 表如下：

Symbol table '.symtab' contains 16 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	125	FUNC	GLOBAL	DEFAULT	1	main
11:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	puts
12:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	exit
13:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	printf
14:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	sleep
15:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	getchar

4.4 Hello.o 的结果解析

使用 `objdump -d -r hello.o > hello_obj.s` 得到 `hello.o` 的反汇编文件, 与 `hello.s` 的主要区别如下:

1、反汇编文件中只有对文件最简单的描述, 记录了文件格式和 `.text` 段。而 `hello.s` 中有对文件名的描述, 全局变量的大小、类型等信息和 `.rodata` 中字符串常量的一些描述。

2、`hello.s` 的汇编代码是由一段段的语句构成。而反汇编代码则是由完整的代码构成, 除需要链接后才能确定的地址之外, 代码包含有完整的跳转和函数调用的逻辑。例如 `hello.s` 中有利用 `L1`, `L2` 等助记符进行的跳转, 反汇编代码中只有相对地址的跳转。汇编语言中 `call` 之后接函数名, 而反汇编中, `call` 之后是一条相对地址, 实际的机器代码中函数调用时都使用相对地址进行跳转。

3、`hello.s` 文件中对字符串常量的访问是采用直接访问 `.rodata` 中条目的方式, 反汇编代码中未指明相对地址, 需要重定位之后才能获得 `rodata` 段的具体地址。

4.5 本章小结

本章通过对汇编后产生的 `hello.o` 和可重定位的 `ELF` 格式的考察、对重定位项目的分析和对反汇编文件与 `hello.s` 的对比, 了解了汇编这一过程的变化。

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称为一个单一文件的过程，这个文件可被加载（复制）到内存并执行。

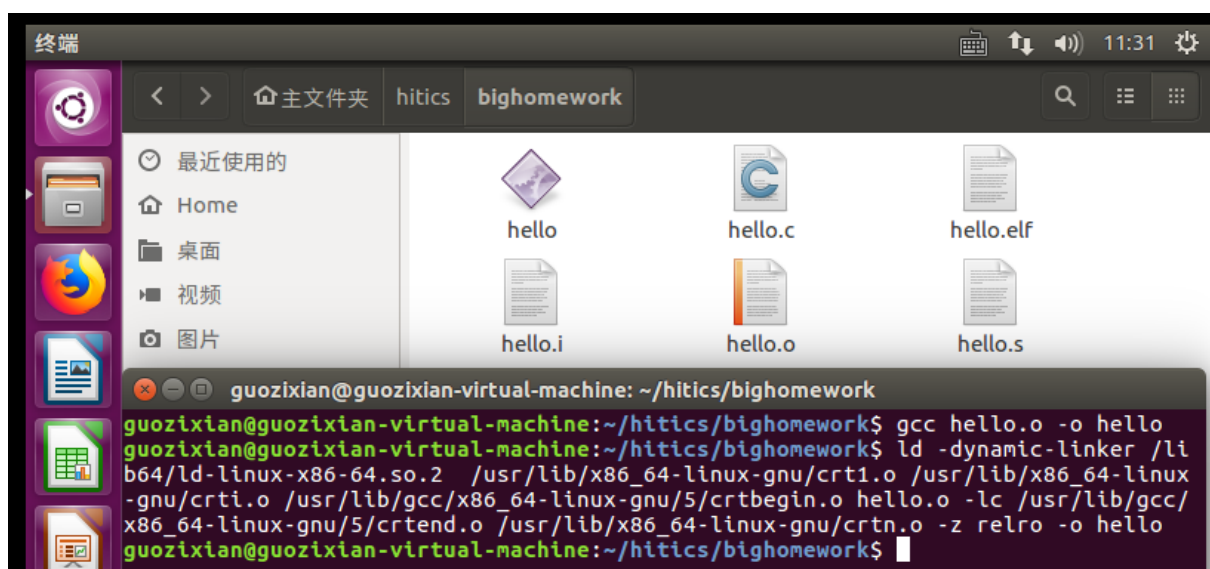
链接可以执行于编译时，也就是在源代码被翻译成机器代码时；也可以执行于加载时，也就是在程序被加载到内存并执行时；甚至执行于运行时，也就是由应用程序来执行。早期计算机系统中链接时手动执行的，在现代系统中，链接器由链接器自动执行。链接器使得分离编译成为可能。开发过程中无需将大型的应用程序组织为一个巨大的源文件，而是可以把它分解为更小、更好管理的模块，可以独立地修改和编译这些模块。

注意：这儿的链接是指从 `hello.o` 到 `hello` 生成过程。

5.2 在 Ubuntu 下链接的命令

`gcc hello.o -o hello` 或者

```
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/5/crtbegin.o hello.o -lc
/usr/lib/gcc/x86_64-linux-gnu/5/crtend.o /usr/lib/x86_64-linux-gnu/crtn.o -z relro -o
hello
```



链接中的文件是 64 位库中的一些可重定位目标文件。

5.3 可执行目标文件 hello 的格式

使用指令 `readelf -a hello > hello_out.elf` 获得可执行文件 `hello` 的 ELF 格式文件。

节头部表中包含了各段的基本信息，包括名称、类型、地址、偏移量、大小、全体大小、旗标、链接、信息、对齐等信息：

节头：

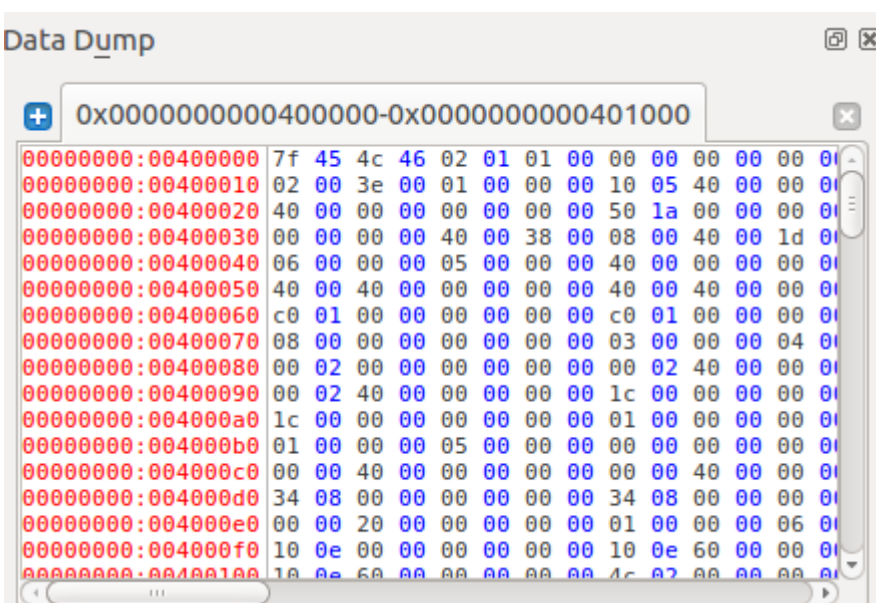
[号]	名称	类型	地址	偏移量
	大小	全体大小	旗标 链接 信息	对齐
[0]	0000000000000000	NULL	0000000000000000	00000000
	0000000000000000	0000000000000000	0 0	0
[1]	.interp	PROGBITS	0000000000400200	00000200
	000000000000001c	0000000000000000	A 0 0	1
[2]	.note.ABI-tag	NOTE	000000000040021c	0000021c
	0000000000000020	0000000000000000	A 0 0	4
[3]	.hash	HASH	0000000000400240	00000240
	0000000000000034	0000000000000004	A 4 0	8
[4]	.dynsym	DYNSYM	0000000000400278	00000278
	00000000000000c0	0000000000000018	A 5 1	8
[5]	.dynstr	STRTAB	0000000000400338	00000338
	0000000000000057	0000000000000000	A 0 0	1
[6]	.gnu.version	VERSYM	0000000000400390	00000390
	0000000000000010	0000000000000002	A 4 0	2
[7]	.gnu.version_r	VERNEED	00000000004003a0	000003a0
	0000000000000020	0000000000000000	A 5 1	8
[8]	.rela.dyn	RELA	00000000004003c0	000003c0
	0000000000000018	0000000000000018	A 4 0	8
[9]	.rela.plt	RELA	00000000004003d8	000003d8
	0000000000000090	0000000000000018	AI 4 22	8
[10]	.init	PROGBITS	0000000000400468	00000468
	000000000000001a	0000000000000000	AX 0 0	4
[11]	.plt	PROGBITS	0000000000400490	00000490
	0000000000000070	0000000000000010	AX 0 0	16
[12]	.plt.got	PROGBITS	0000000000400500	00000500
	0000000000000008	0000000000000000	AX 0 0	8
[13]	.text	PROGBITS	0000000000400510	00000510
	00000000000001f2	0000000000000000	AX 0 0	16
[14]	.fini	PROGBITS	0000000000400704	00000704
	0000000000000009	0000000000000000	AX 0 0	4
[15]	.rodata	PROGBITS	0000000000400710	00000710
	000000000000002f	0000000000000000	A 0 0	4
[16]	.eh_frame	PROGBITS	0000000000400740	00000740
	00000000000000f4	0000000000000000	A 0 0	8
[17]	.init_array	INIT_ARRAY	0000000000600e10	00000e10
	0000000000000008	0000000000000000	WA 0 0	8
[18]	.fini_array	FINI_ARRAY	0000000000600e18	00000e18
	0000000000000008	0000000000000000	WA 0 0	8

Key to Flags:

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

5.4 hello 的虚拟地址空间

使用 edb 运行 hello 查看 Data Dump 的虚拟地址范围如下:



ELF 文件中程序头部表的内容:

程序头:

Type	Offset	VirtAddr	PhysAddr
	FileSiz	MemSiz	Flags Align
PHDR	0x0000000000000040	0x0000000000400040	0x0000000000400040
	0x00000000000001c0	0x00000000000001c0	R E 8
INTERP	0x0000000000000200	0x0000000000400200	0x0000000000400200
	0x00000000000001c	0x00000000000001c	R 1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]			
LOAD	0x0000000000000000	0x0000000000400000	0x0000000000400000
	0x0000000000000834	0x0000000000000834	R E 200000
LOAD	0x0000000000000e10	0x0000000000600e10	0x0000000000600e10
	0x000000000000024c	0x0000000000000250	RW 200000
DYNAMIC	0x0000000000000e28	0x0000000000600e28	0x0000000000600e28
	0x00000000000001d0	0x00000000000001d0	RW 8
NOTE	0x000000000000021c	0x000000000040021c	0x000000000040021c
	0x0000000000000020	0x0000000000000020	R 4
GNU_STACK	0x0000000000000000	0x0000000000000000	0x0000000000000000
	0x0000000000000000	0x0000000000000000	RW 10
GNU_RELRO	0x0000000000000e10	0x0000000000600e10	0x0000000000600e10
	0x00000000000001f0	0x00000000000001f0	R 1

PHDR 表示该段具有读/执行权限，表示自身所在的程序头部表在内存中的位置为内存起始位置 0x400000 偏移 0x40 字节处、大小为 0x1c0 字节。

INTERP 表示该段具有读权限，位于内存起始位置 0x400000 偏移 0x200 字节处，大小为 0x1c 个字节，记录了程序所用 ELF 解析器（动态链接器）的位置位于：
/lib64/ld-linux-x86-64.so.2

```
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.....
```

第一个 LOAD 表示第一个段（代码段）有读权限，可以读取后执行，开始于内存地址 0x400000 处，总共内存大小是 0x834 个字节。

第二个 LOAD 表示第二个段（数据段）有读写权限，开始于内存地址 0x600e10 地址处，总的内存大小为 0x250 个字节。

NOTE 表示该段位于内存起始位置 0x400000 偏移 0x21c 字节处，大小为 0x20 个字节，该段是以 ‘\0’ 结尾的字符串，包含一些附加信息。

GNU_RELRO 表示该段在重定位后设置为只读。

DYNAMIC 段描述动态链接信息。

5.5 链接的重定位过程分析

hello 反汇编文件中包含 Disassembly of section .init .plt .plt.got .text .fini，而 hello.o 反汇编文件只有代码段的汇编呈现。

其中未链接之前值为 0 的跳转地址已经有了确定的运行时地址，链接库中的函数

也已经加入到了 `plt` 中。

以调用 `exit` 为例，说明重定位过程中地址变化。执行 `exit` 的 `call` 指令的时候，PC 值为下一条程序的地址：40062f，也可以通过偏移 `main+0x29` 获得，`call` 之后的地址为相对地址， $*refptr = 0x4004e0 - 0x40062f = 0xfffffeb1$ ，与机器码的小端序的值相同。

```
400625:  bf 01 00 00 00      mov     $0x1,%edi
40062a:  e8 b1 fe ff ff      callq   4004e0 <exit@plt>
40062f:  c7 45 fc 00 00 00 00  movl    $0x0,-0x4(%rbp)
```

使用绝对地址寻址的位置同理。

5.6 hello 的执行流程

函数名	功能
ld-2.23.so!_start	开始加载程序
ld-2.23.so!_dl_start	
ld-2.23.so!_dl_init	
libc-2.27.so!__libc_start_main	
hello!_start	
hello!_main	main 函数
hello!puts@plt	程序调用的库函数
hello!exit@plt	
hello!printf@plt	
hello!sleep@plt	
hello!getchar@plt	
libc-2.23.so!exit	程序结束

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。两表内容分别为：

PLT: 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: GOT 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。

根据 hello 的 ELF 格式文件位置得到 got 的起始位置 0x601000。

调用 ld-2.23.so!_dl_start 之前的.got

```
00000000:00601000 28 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 a6 04 40 00 00 00 00 00 .....u.@....
00000000:00601020 b6 04 40 00 00 00 00 00 c6 04 40 00 00 00 00 00 u.@....t.@....
00000000:00601030 d6 04 40 00 00 00 00 00 e6 04 40 00 00 00 00 00 t.@....h.@....
00000000:00601040 f6 04 40 00 00 00 00 00 00 00 00 00 00 00 00 00 .@.....
00000000:00601050 00 00 00 00 00 00 00 00 02 00 00 00 00 00 00 00 .....

```

调用 ld-2.23.so!_dl_start 之后的.got

```
00000000:00601000 28 0e 60 00 00 00 00 00 68 e1 24 8a 09 7f 00 00 (.`.....h$. ...
00000000:00601010 70 e8 03 8a 09 7f 00 00 a6 04 40 00 00 00 00 00 p|.. ....u.@....
00000000:00601020 b6 04 40 00 00 00 00 00 c6 04 40 00 00 00 00 00 u.@....t.@....
00000000:00601030 d6 04 40 00 00 00 00 00 e6 04 40 00 00 00 00 00 t.@....h.@....
00000000:00601040 f6 04 40 00 00 00 00 00 00 00 00 00 00 00 00 00 .@.....
00000000:00601050 00 00 00 00 00 00 00 00 02 00 00 00 00 00 00 00 .....

```

经过 hello 程序中 ld-2.23.so!_dl_start 的初始化，.got 部分已经改变。

5.8 本章小结

在本章中主要介绍了链接的概念与作用、hello 的 ELF 格式，分析了 hello 的虚拟地址空间、重定位过程、执行流程、动态链接过程。

第 6 章 hello 进程管理

6.1 进程的概念与作用

进程的经典定义就是一个执行中的程序的实例。系统中的每个程序都运行在某个进程的上下文中。上下文是由程序正确运行所需的状态组成的。这个状态包括存放在内存中的程序的代码和数据，它的栈、通用目的寄存器的内容、程序计数器、环境变量、以及打开文件描述符的集合。

进程给应用程序提供的关键抽象有：

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。

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

shell 是内核与用户交互的接口，它接受用户发出的指令，调用相关的程序执行。

shell 的一项主要功能是在交互方式下解释从命令行输入的命令。

处理流程：受到用户的命令行输入，通过 `paracoline` 函数解释命令行，在执行的过程中，如果是内置命令就立即执行，如果是执行某一个文件，就 `fork`，`execve` 加载一个新的进程去执行该文件，默认是后台执行。

6.3 Hello 的 fork 进程创建过程

进程的创建过程：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。子进程得到一份与父进程虚拟地址空间相同的副本，包括代码和数据段、堆、共享库以及用户栈。子进程还获得与父进程任何打开文件描述符相同的副本，这就意味着当父进程调用 `fork` 时，子进程可以读写父进程中打开的任何文件。但两者的进程号不同，意味着是两个不同的独立的进程。

`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。

hello 进程创建过程为：shell 通过 `fork`，`execve` 创建 hello 子进程然后调用 `waitpid()` 函数等待 hello 子进程结束。

6.4 Hello 的 execve 过程

系统为 fork 子进程之后,子进程调用 execve 函数加载可执行目标文件 hello,execve 只有在运行错误时才会返回,否则不返回。加载过程中,带参数列表 argv 和环境变量列表 envp,调用启动代码,启动代码设置栈,并将控制传递给新进程的主函数。

6.5 Hello 的进程执行

进程执行中涉及到逻辑控制流、并发流、用户模式和内核模式、上下文切换等概念:

1、逻辑控制流:

在调试器单步执行程序时,会发现一系列的程序计数器(PC)的值,这些值唯一地对应于包含在程序的可执行目标文件中的指令,或是包含在运行时动态链接到程序的共享对象中的指令。这个 PC 的值的序列叫做逻辑控制流。

2、并发流:

一个逻辑流的执行在时间上与另一个流重叠,称为并发流,这两个流被称为并发地执行。

多个流并发地执行的一般现象被称为并发。一个进程和其他进程轮流运行的概念被称为多任务。

一个进程执行它的控制流的一部分的每一时间段叫做时间片。因此,多任务也叫时间分片。

3、用户模式和内核模式:

处理器通过某个控制寄存器中的一个模式位来提供限制一个应用可以执行的指令以及它可以访问的地址空间范围的功能。该寄存器描述了当前进程享有的特权。当设置了模式位时,进程就运行在内核模式中。一个运行在内核模式的进程可以执行指令集中的任何指令,并且可以访问系统中的任何内存为止;没有设置模式位时,进程就运行在用户模式中。用户模式的进程不允许和执行特权指令、也不允许用户模式中的进程直接引用地址空间中内核区内的代码和数据。

4、上下文切换:

内核为每个进程维持一个上下文,上下文就是内核重新启动的一个被强占的进程所需的状态。由包括通用目的寄存器、浮点寄存器、程序计数器、用户站、

状态寄存器、内核栈和各种内核数据结构。

上下文切换的机制：保存当前进程的上下文；恢复某个先前被抢占的进程被保存的上下文；将控制传递给这个新恢复的进程。

5、进程调度及用户态和核心态的转换：

在进程执行的某些时刻，内核可以决定抢占当前进程，并重新开始一个先前被抢占了的进程，这个决策就叫做调度，是由内核中称为调度器的代码处理的。在内和调度了一个新的进程运行后，它就抢占当前进程，并使用上文所述的上下文切换的机制将控制转移到新的进程。内核代表的用户执行系统调用时，可能会发生上下文切换；中断也有可能引发上下文切换。

在切换的第一部分中，内核代表进程 A 在内核模式下执行指令。然后在某一时刻，它开始代表进程 B(仍然是内核模式下)执行指令。在切换之后，内核代表进程 B 在用户模式下执行指令。随后，进程 B 在用户模式下运行一会儿，直到磁盘发出一个中断信号，表示数据已经从磁盘传送到了内存。内核判定进程 B 已经运行了足够长的时间，就执行一个从进程 B 到进程 A 的上下文切换，将控制返回给进程 A 中紧随在系统调用 `read` 之后的那条指令。进程 A 继续运行，直到下一次异常发生，依此类推。

6.6 hello 的异常与信号处理

hello 执行过程中出现的异常种类可能会有：中断、陷阱、故障、终止。

中断是异步发生的，是来自处理器外部的 I/O 设备的信号的结果。硬件中断的异常处理程序被称为中断处理程序。

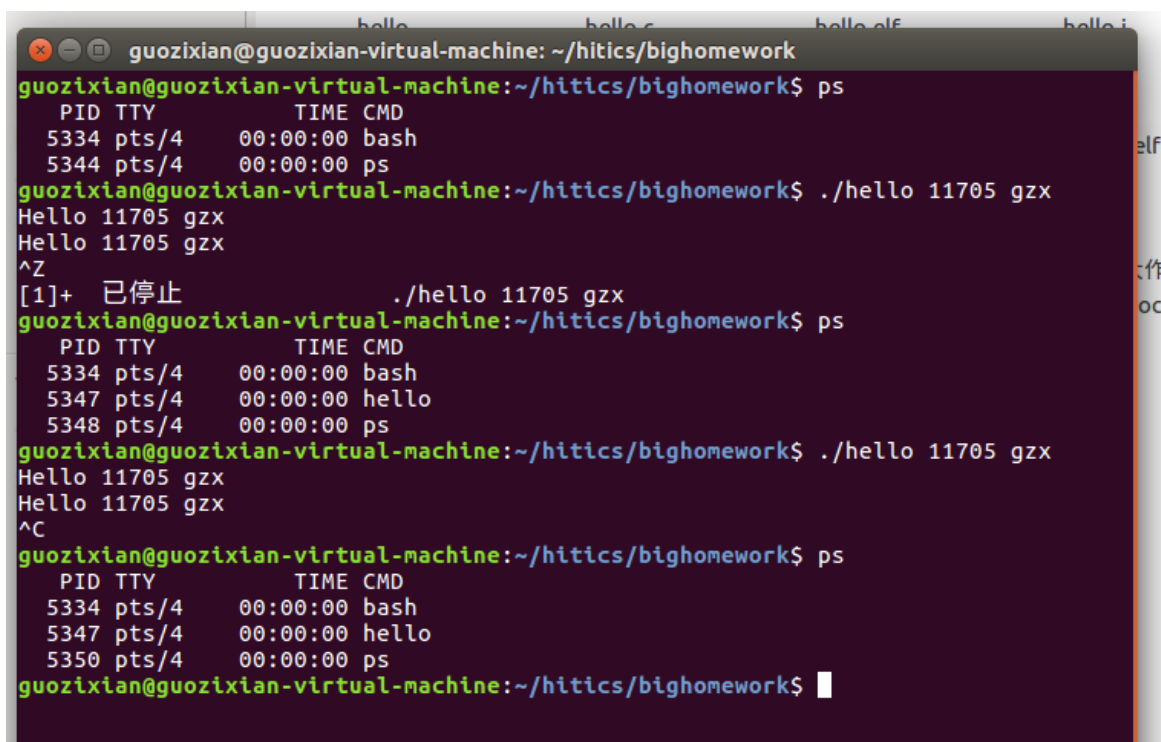
陷阱是有意的异常，是执行一条指令的结果。就像中断处理程序一样，陷阱处理程序将控制返回到下一条指令。陷阱最重要的用途是在用户程序和内核之间提供一个像过程一样的接口，叫做系统调用。

故障由错误情况引起，它可能能够被故障处理程序修正。当故障发生时，处理器将控制转移给故障处理程序。如果处理程序能够修正这个错误情况，它就将控制返回到引起故障的指令，从而重新执行它。否则处理程序返回到内核中的 `abort` 例程，`abort` 例程会终止引起故障的应用程序。

终止是不可恢复的致命错误造成的结果，通常是一些硬件错误，比如 **DRAM**

或者 SRAM 位被损坏时发生的奇偶错误。终止处理程序从不将控制返回给应用程序。处理程序将控制返回给一个 `abort` 例程，该例程会终止这个应用程序。

信号允许进程和内核中断其他进程。每种信号都对应于某种系统事件。低层的硬件异常是由内核异常处理程序处理的，正常情况下，对用户进程是不可见的。信号提供一种机制，通知用户进程发生了这些异常。例如在 `hello` 运行过程中输入 `Ctrl-Z`，`Ctrl-C` 可以挂起、终止程序：

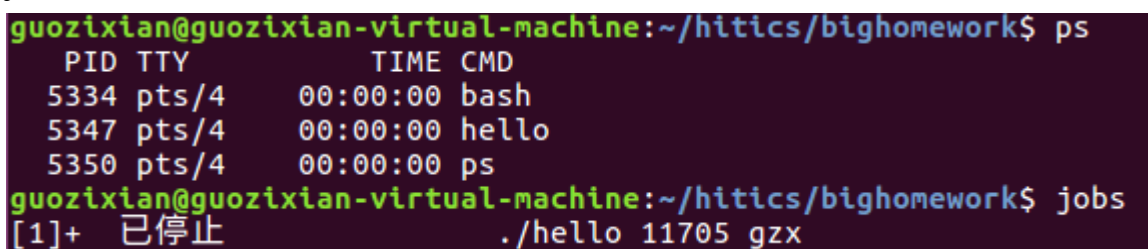


```

guozixian@guozixian-virtual-machine: ~/hitics/bighomework
guozixian@guozixian-virtual-machine:~/hitics/bighomework$ ps
  PID TTY          TIME CMD
 5334 pts/4        00:00:00 bash
 5344 pts/4        00:00:00 ps
guozixian@guozixian-virtual-machine:~/hitics/bighomework$ ./hello 11705 gzx
Hello 11705 gzx
Hello 11705 gzx
^Z
[1]+  已停止                  ./hello 11705 gzx
guozixian@guozixian-virtual-machine:~/hitics/bighomework$ ps
  PID TTY          TIME CMD
 5334 pts/4        00:00:00 bash
 5347 pts/4        00:00:00 hello
 5348 pts/4        00:00:00 ps
guozixian@guozixian-virtual-machine:~/hitics/bighomework$ ./hello 11705 gzx
Hello 11705 gzx
Hello 11705 gzx
^C
guozixian@guozixian-virtual-machine:~/hitics/bighomework$ ps
  PID TTY          TIME CMD
 5334 pts/4        00:00:00 bash
 5347 pts/4        00:00:00 hello
 5350 pts/4        00:00:00 ps
guozixian@guozixian-virtual-machine:~/hitics/bighomework$

```

`jobs`：显式当前暂停的程序

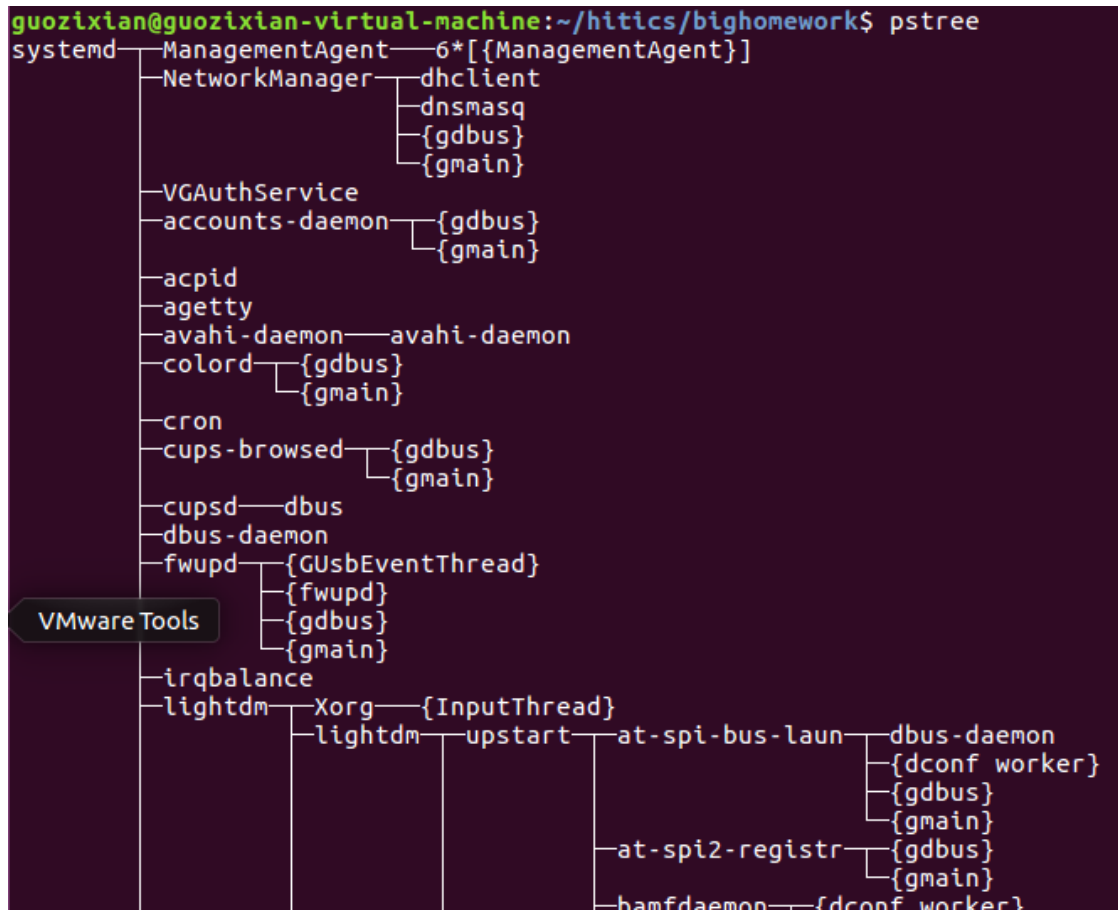


```

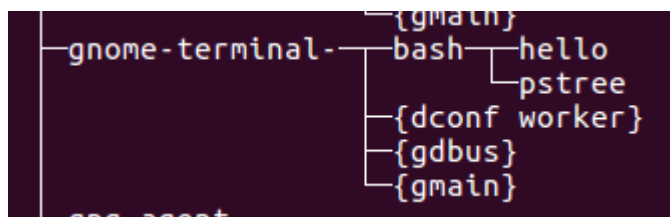
guozixian@guozixian-virtual-machine:~/hitics/bighomework$ ps
  PID TTY          TIME CMD
 5334 pts/4        00:00:00 bash
 5347 pts/4        00:00:00 hello
 5350 pts/4        00:00:00 ps
guozixian@guozixian-virtual-machine:~/hitics/bighomework$ jobs
[1]+  已停止                  ./hello 11705 gzx

```

pstree: 打印进程树



其中可以找到 shell 中的进程:



fg 将进程放到前台执行，继续打印学号姓名。

使用 kill 向进程发送信号 9，终止进程。

```
guozixian@guozixian-virtual-machine:~/hitics/bighomework$ fg
./hello 11705 gzx
Hello 11705 gzx
Hello 11705 gzx
Hello 11705 gzx
Hello 11705 gzx
Hello 11705 gzx
Hello 11705 gzx
Hello 11705 gzx
^Z
[1]+ 已停止                  ./hello 11705 gzx
guozixian@guozixian-virtual-machine:~/hitics/bighomework$ ps
  PID TTY          TIME CMD
  5334 pts/4        00:00:00 bash
  5347 pts/4        00:00:00 hello
  5366 pts/4        00:00:00 ps
guozixian@guozixian-virtual-machine:~/hitics/bighomework$ kill -9 5347
[1]+ 已杀死                  ./hello 11705 gzx
guozixian@guozixian-virtual-machine:~/hitics/bighomework$ ps
  PID TTY          TIME CMD
  5334 pts/4        00:00:00 bash
  5369 pts/4        00:00:00 ps
guozixian@guozixian-virtual-machine:~/hitics/bighomework$
```

6.7 本章小结

本章从进程的角度描述了 hello 在 shell 中运行过程中 fork 和 execve 创建进程的过程，并针对 execve 过程中建立的虚拟内存映像以及栈组织结构等作出说明。同时介绍了逻辑控制流中内核的调度及上下文切换等机制，Shell 运行的处理流程以及 hello 执行过程中可能引发的异常和信号处理。

第 7 章 hello 的存储管理

7.1 hello 的存储器地址空间

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。物理地址是在地址总线上，数据总线可以访问主存的某个特定存储单元的内存地址。

在一个带虚拟内存的系统中，CPU 从一个有 N 个地址的地址空间中生成虚拟地址，这个地址空间成为虚拟地址空间。

地址空间是一个非负整数的有序集合，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。线性地址就是线性地址空间中的地址。

逻辑地址：程序代码经过编译后出现在汇编程序中地址。逻辑地址由选择符和偏移量组成。

逻辑地址转换成虚拟地址，是由段式管理执行的，线性地址转换成物理地址，是由页式管理执行的。

hello 执行过程中，调用函数的语句中是逻辑地址，经过段地址转化为虚拟地址，然后通过 MMU 转化为物理地址，进行访问。

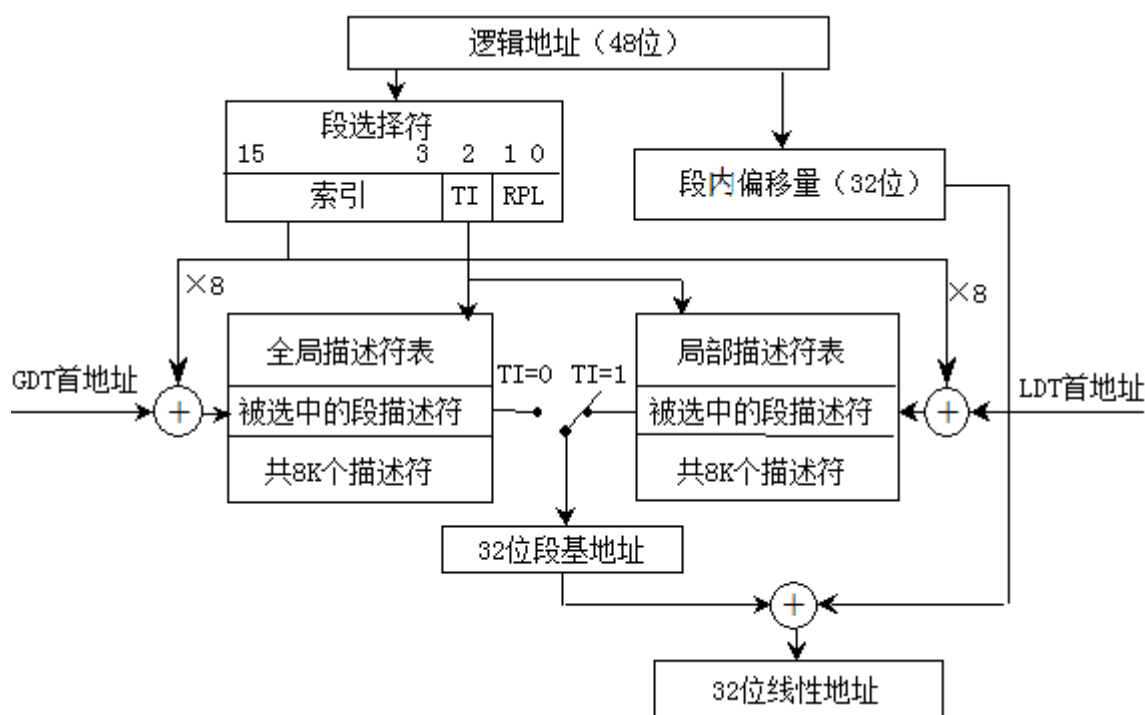
7.2 Intel 逻辑地址到线性地址的变换-段式管理

在段式存储管理中，程序的地址空间被划分为若干个段，这样每个进程有一个二维的地址空间。在段式存储管理系统中，则为每个段分配一个连续的分区，而进程中的各个段可以不连续地存放在内存的不同分区中。

linux x86_64 下运行时内存映像从低地址到高地址依次为：只读代码段、读/写段、运行时堆、共享库的内存映射区域、用户栈、内核内存。

由逻辑地址得到虚拟地址的过程如下图，16 位段寄存器用于存放段选择符：CS（代码段）是指程序代码所在段；SS（栈段）是指栈区所在段；DS（数据段）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。

描述符被送至描述符 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

虚拟内存概念：虚拟内存是系统对主存的抽象概念，是硬件异常、硬件地址翻译、主存、磁盘文件和内存文件的完美交互。为每个进程提供了一个大的、一致的和私有的地址空间。

虚拟内存被组织为一个由存放在磁盘上的 N 个连续的字节大小的单元组成的数组。每个字节都有一个唯一的虚拟地址，作为到数组的索引。磁盘上的数据被分割成块，这些块作为磁盘和主存（较高层）之间的传输单元。虚拟页则是虚拟内存被分割为固定大小的块。物理内存被分割为物理页，大小与虚拟页大小相同。

系统为每个进程提供一个页表，将虚拟地址映射到物理地址。通过上一节的方式生成虚拟地址之后，以此为索引来定位页表中 PTE，读取内容，判断标记位，如果有效，则说明物理内存中已经缓存了相应的页，如果标记位无效，则说明此页没有缓存到物理内存中。

之后将虚拟页号使用物理页号替代，偏移量相同，就得到了物理地址。

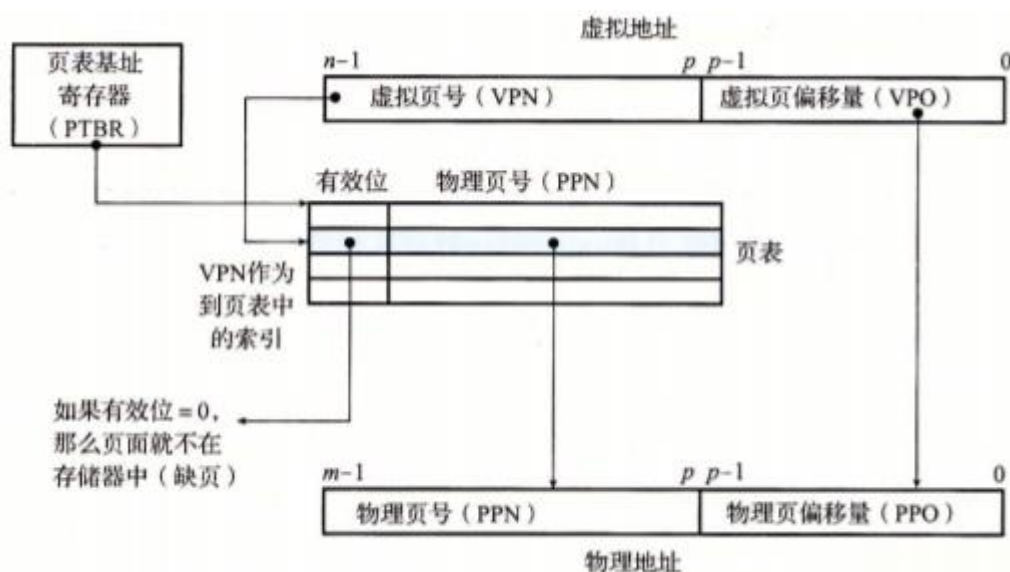


图 9-12 使用页表的地址翻译

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

在 Intel Core i7 环境下研究 VA 到 PA 的地址翻译问题。前提如下：

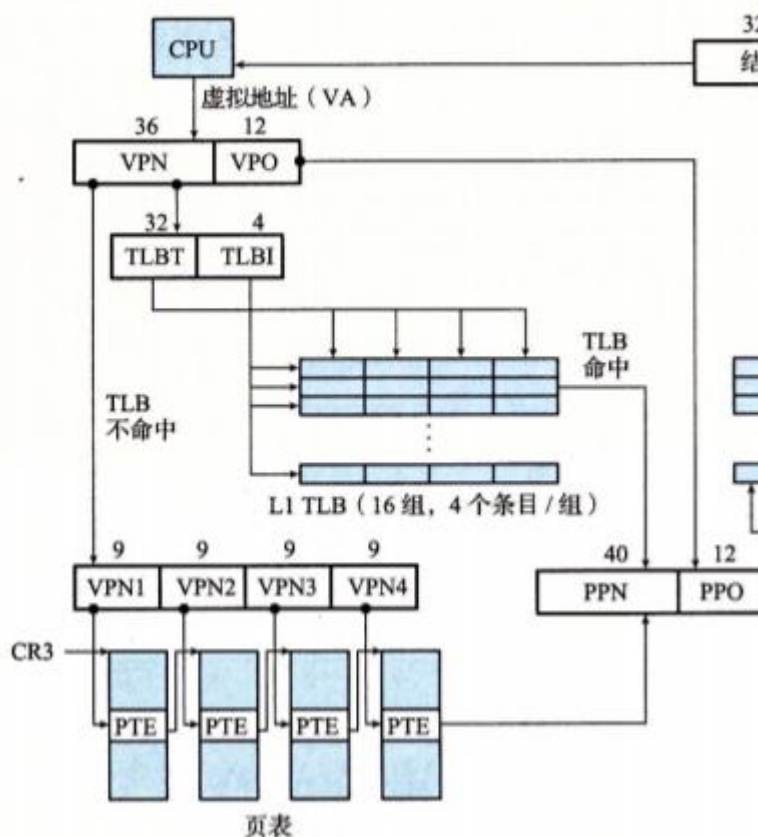
虚拟地址空间 48 位，物理地址空间 52 位，页表大小 4KB，4 级页表。TLB 4 路 16 组相联。CR3 指向第一级页表的起始位置（上下文一部分）。

解析前提条件：由一个页表大小 4KB，一个 PTE 条目 8B，共 512 个条目，使用 9 位二进制索引，一共 4 个页表共使用 36 位二进制索引，所以 VPN 共 36 位，因为 VA 48 位，所以 VPO 12 位；因为 TLB 共 16 组，所以 TLBI 需 4 位，因为 VPN 36 位，所以 TLBT 32 位。

如图，CPU 产生虚拟地址 VA，VA 传送给 MMU，MMU 使用前 36 位 VPN 作为 TLBT（前 32 位）+TLBI（后 4 位）向 TLB 中匹配，如果命中，则得到 PPN（40bit）与 VPO（12bit）组合成 PA（52bit）。

如果 TLB 中没有命中，MMU 向页表中查询，CR3 确定第一级页表的起始地址，VPN1（9bit）确定在第一级页表中的偏移量，查询出 PTE，如果在物理内存中且权限符合，确定第二级页表的起始地址，以此类推，最终在第四级页表中查询到 PPN，与 VPO 组合成 PA，并且向 TLB 中添加条目。

如果查询 PTE 的时候发现不在物理内存中，则引发缺页故障。如果发现权限不够，则引发段错误。



7.5 三级 Cache 支持下的物理内存访问

针对物理内存访问，主要对各类高速缓存存储器的读写策略做出说明：

当 CPU 执行一条读内存字 w 的指令，它向 L1 高速缓存请求这个字。如果 L1 高速缓存由 w 的一个缓存的副本，那么就得到 L1 的高速缓存命中，高速缓存会很快抽取出 w 并返回给 CPU。否则就是缓存不命中，当 L1 高速缓存向主存请求包含 w 的块的一个副本时，CPU 必须等待。当被请求的块最终从内存到达时，L1 高速缓存将这个快存放在他的一个高速缓存行里，从被缓存的块中抽取字 w ，然后返回给 CPU。总体来看，高速缓存确定一个请求是否命中，然后抽取出被请求的字的的过程，分为三步，（1）组选择、（2）行匹配、（3）字抽取。

直接映射高速缓存读策略：

直接映射高速缓存 $E=1$ ，即每组只有一行。组选择是通过组索引位标识组。高速缓存从 w 的地址中间抽取出 s 个组索引位，这些位被解释为一个对应于一个组号的无符号整数，来进行组索引。行匹配中，确定了某个组 i ，接下来需要确定是否有字 w 的一个副本存储在组 i 包含的一个高速缓存行里，因为直接映射高速缓存

存只有一行，如果有效位为 1 且标志位相同则缓存命中，根据块偏移位即可查找到对应字的地址并取出；若有效位为 1 但标志位不同则冲突不命中，有效位为 0 则为冷不命中，此时都需要从存储器层次结构下一层取出被请求的块，然后将新的块存储在组索引位指示的组中的一个高速缓存行中。

组相联高速缓存读策略：

组相联高速缓存每个组都会保存多余一个的高速缓存行，组选择与直接映射高速缓存的组选择一样，通过组索引位标识组。行匹配时需要找遍组中所有行，找到标记位有效位均相同的一行则缓存命中；如果 CPU 请求的字不在组的任何一行中，则缓存不命中，选择替换时如果存在空行选择空行，如果不存在空行则通过替换策略替换其中一行。

全相联高速缓存读策略：

全相联高速缓存只包含一个组，其行匹配和字选择与组相联高速缓存中一样，区别主要是规模大小的问题。

写策略：分为两种，直写和写回：

直写是立即将 w 的高速缓存块写回到紧接着的低一层中。虽然简单，但是只写的缺点是每次写都会引起总线流量。

写回尽可能的推迟更新，只有当替换算法要驱逐这个更新过的块时，才把它写回到紧接着的第一层中，由于局部性，写回能显著减少总线流量，但增加了复杂性。处理写不命中有两种方法一种为写分配，加载相应的的低一层的块到高速缓存中，然后更新这个高速缓存块。另一种方法为非写分配，避开高速缓存，直接把这个字写到低一层中，直写高速缓存通常是非写分配的，写回高速缓存通常是写分配的。

7.6 hello 进程 fork 时的内存映射

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。下

图为共享对象使用实例：

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`，主要步骤如下：

删除已存在的用户区域；映射 `hello` 私有区域；映射共享区域，设置出程序运行需要的栈空间。然后修改 `PC` 将控制传递给新进程的主函数。

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

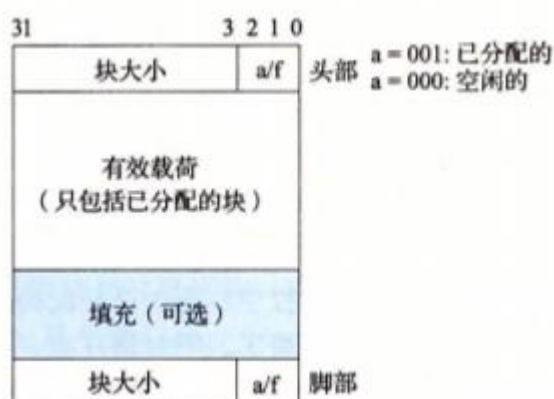
动态内存分配器维护着一个进程的虚拟内存区域，称为堆。分配器将堆视为一组不同大小的块的集合来维护。每个块就是一个连续的虚拟内存片，要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可用于分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

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。

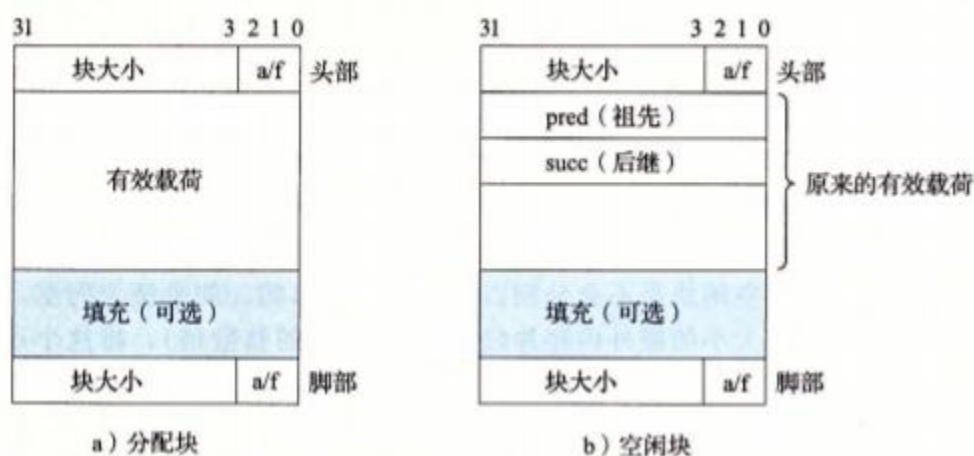
显式分配器：要求应用显式地释放任何已分配的块。

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。

隐式空闲链表块结构：



显式空闲链表块结构：



显式空闲链表要求每个块内部至少有存储两个地址的空间，所以一定程度上造成了内部碎片。

显式空闲链表的维护方法还会涉及到分离式链表、首次适配等问题。

7.10 本章小结

本章从 Linux 存储器的地址空间起，阐述了 Intel 的段式管理和页式管理机制，以及 TLB 与多级页表支持下的 VA 到 PA 的转换，同时对 cache 支持下的物理内存访问做了说明。针对内存映射及管理，简述了 hello 的 fork 和 execve 内存映射，了解了缺页故障与缺页中断处理程序，对动态分配管理做了系统阐述。

第 8 章 hello 的 IO 管理

8.1 Linux 的 IO 设备管理方法

一个 Linux 文件就是一个 m 字节的序列。所有的 I/O 设备（例如网络、磁盘和终端）都被模型化为文件，而所有的输入和输出都被当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。这使得所有输入和输出都能以一种统一且一致的方式来执行：

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。

Linux Shell 创建的每个进程开始时都有三个打开的文件：标准输入、标准输出、标准错误。

改变当前文件的位置。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` 操作，显示地设置文件的当前位置为 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 $k+n$ 。写操作就是从内存复制 $n>0$ 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。当应用完成了对文件的访问之后，它就通知内核关闭这个文件。

8.2 简述 Unix IO 接口及其函数

`open()`

函数原型： `int open(char * filename, int flags, mode_t mode);`

解析：`open` 函数将 `file` 那么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。

`close()`

函数原型： `int close(int fd);`

`read()`

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

解析：`read` 函数从描述符为 `fd` 的当前文件位置复制最多 `n` 个字节到内存位置 `buf`。返回值 -1 表示一个错误，而返回值 0 表示 EOF。否则，返回值表示的是实际传送的字节数量。

`write()`

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

解析：`write` 函数从内存位置 `buf` 复制至多 `n` 个字节到描述符 `fd` 的当前文件位置。

8.3 `printf` 的实现分析

`printf` 函数是在 `stdio.h` 头文件中声明的，具体代码实现如下：

```
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;
}
```

它的参数包括一个字串 `fmt`，和……。…表示参数个数不能确定，也就是格式化标准输出，我们也不能确定到底有几个格式串。

在函数的第 6 行，`arg` 变量定位到了第二个参数，也就是第一个格式串。和这句有关的具体问题，还是请看参考文献[6]，本节只是简述 `printf` 的实现过程。

`va_list` 是一个数据类型，其声明如下：

```
typedef char *va_list
```

至于赋值语句右侧的，是一个地址偏移定位，定位到了从 `fmt` 开始之后的第一个 `char*` 变量，也就是第二个参数了。

接下来是调用 `vsprintf` 函数，并把返回值赋给整型变量 `i`。后来又调用 `write` 函数从内存位置 `buf` 处复制 `i` 个字节到标准输出。想必这个 `i` 就是 `printf` 需要的输出字符总数，那么 `vsprintf` 就是对参数解析了。`vsprintf` 函数代码如下：

```
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++`) { //p 初始化为 `buf`，下面即将把 `fmt` 解析并把结果存入 `buf` 中

```
    /* 寻找格式化字符串 */
    if (*fmt != '%') {
        *p++ = *fmt;
        continue;
    }
```

`fmt++`; //此时，`fmt` 指向的是格式化字符串的内容了

```
switch (*fmt) {
    /* 这是格式化字符串为 %x 的情况 */
    case 'x':
```

`itoa(tmp, *((int*)p_next_arg);` //把 `fmt` 对应的那个参数字串转换格式，放到 `tmp` 串中

```
    strcpy(p, tmp); //tmp 串存到 p 中，也就是 buf 中
```

```
    p_next_arg += 4; //定位到下一个参数
```

```
    p += strlen(tmp); //buf 中的指针也要往下走
```

```
    break;
```

```
    /* Case %s */
```

```
    case 's':
```

```
        break;
```

```
    default:
```

```
        break;
```

```
    }
```

```
}
```

```

    return (p - buf);
}

```

这个 `vsprintf` 只处理了 `%x` 这一种格式化字符串的情况。已经给出比较详细的注释了。
`write` 函数的汇编代码是这样给出的：

```

1. write:
2.     mov eax, _NR_write
3.     mov ebx, [esp + 4]
4.     mov ecx, [esp + 8]
5.     int INT_VECTOR_SYS_CALL

```

第 5 行，表示要通过系统来调用 `sys_call` 这个函数，函数实现为：

```

1. sys_call:
2.     call save
3.
4.     push dword [p_proc_ready]
5.
6.     sti
7.
8.     push ecx
9.     push ebx
10.    call [sys_call_table + eax * 4]
11.    add esp, 4 * 3
12.
13.    mov [esi + EAXREG - P_STACKBASE], eax
14.
15.    cli
16.
17.    ret

```

在 `write` 和 `sys_call` 中，`ecx` 寄存器中存放的是要打印元素的个数，`ebx` 寄存器中存放的是要打印的 `buf` 字符数组中的第一个元素。这个函数的功能就是不断地打印出字符，直到遇到 `'\0'`。

字符显示驱动子程序：从 ASCII 到字模库到显示 `vram`（存储每一个点的 RGB 颜色信息）。

显示芯片按照刷新频率逐行读取 `vram`，并通过信号线向液晶显示器传输每一个点（RGB 分量）。

到此，`printf` 要打印的东西，就呈现在标准输出上了。

8.4 getchar 的实现分析

getchar 的函数声明在 stdio.h 头文件中，具体代码实现如下

```
1. int getchar(void)
2. {
3.     static char buf[BUFSIZ];
4.     static char* bb=buf;
5.     static int n=0;
6.     if(n==0)
7.     {
8.         n=read(0,buf,BUFSIZ);
9.         bb=buf;
10.    }
11.    return(--n>=0)?(unsigned char)*bb++:EOF;
12. }
```

bb 是缓冲区的开始，int 变量 n 初始化为 0，只有在 n 为 0 的情况下，从缓冲区中读 BUFSIZ 个字节，就是缓冲区中的内容全部读入。这时候，n 的值被修改为，成功读入的字节数，正常情况下 n 就是一个正数值。返回时，如果 n 大于 0，那么就返回缓冲区的第一个字符。否则，就是没有从缓冲区读到字节，返回 EOF。

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

8.5 本章小结

所有的 I/O 设备都被模型化为文件，通过文件的读写来实现 I/O 操作。Unix I/O 接口函数可以实现一些 I/O 操作。printf 的实现和 vsprintf 以及 write 有关，要先解析格式化字符串，再调用 I/O 函数 write 写到标准输出上。getchar 函数与键盘回车相关，也就是需要异步异常-键盘中断的处理，之后调用 I/O 函数 read，读取标准输入。

结论

hello 的生命进程:

- 1、用户编辑，得到 `hello.c` 源文件。
- 2、编译器套件对 `hello.c` 进行编译得到可执行文件。
- 3、在 `shell` 中执行 `hello`，`fork` 一个进程，`execve` 加载 `hello`，把控制权交给 `hello`。
- 4、`hello` 与许多进程并行执行，通过相应信号，受到到内核用户的调度。
- 5、`hello` 输出信息时需要调用 `printf` 和 `getchar` 需要调用 Unix I/O 中的 `write` 和 `read`。
- 6、`hello` 中的访存操作，需要经历逻辑地址到虚拟地址最后到物理地址的变换。
- 7、`hello` 结束进程后，`bash` 作为 `hello` 的父进程会回收 `hello` 进程。

从 `hello` 程序在 `Linux` 中的整个生命进程中，可以看出 `hello` 程序虽然很简单，但是整个实现过程相当复杂，需要整个计算机硬件、操作系统的配合。程序员应当了解程序执行过程中低层硬件、操作系统所做的工作，进而写出适应低层的优秀的程序。

附件

hello.c	编辑产生的源代码
hello.i	预处理生成
hello.s	编译生成
hello.o	汇编生成
hello_obj.s	hello.s 反汇编生成
hello.elf	hello.o 的 ELF 格式
hello_out.elf	hello 的 ELF 格式
hello_out_obj.s	hello 反汇编生成
hello	可执行文件

参考文献

[1] 兰德尔 E.布莱恩特 大卫 R.奥哈拉伦. 深入理解计算机系统（第 3 版）.机械工业出版社.

[2] 编译链接中的-可重定位目标文件

https://blog.csdn.net/ky_heart/article/details/51865526

[3] 虚拟地址、逻辑地址、线性地址、物理地址

https://blog.csdn.net/rabbit_in_android/article/details/49976101

[4] printf 函数实现的深入剖析

<https://www.cnblogs.com/pianist/p/3315801.html>

[5] 程序人生 Hello's P2P

<https://blog.csdn.net/hahalidaxin/article/details/85144974>

[6] 计算机系统大作业 《程序人生-Hello' s P2P》

<https://blog.csdn.net/HelloTheWholeWorld/article/details/85339220#1.4%E6%9C%AC%E7%AB%A0%E5%B0%8F%E7%BB%93>