

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</u>
學 號	<u>1170300913</u>
班 級	<u>1703009</u>
學 生	<u>譚通海</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

摘要是论文内容的高度概括，应具有独立性和自含性，即不阅读论文的全文，就能获得必要的信息。摘要应包括本论文的目的、主要内容、方法、成果及其理论与实际意义。摘要中不宜使用公式、结构式、图表和非公知公用的符号与术语，不标注引用文献编号，同时避免将摘要写成目录式的内容介绍。

关键词：关键词 1；关键词 2；……；

（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	- 4 -
第 2 章 预处理	- 5 -
2.1 预处理的概念与作用	- 5 -
2.2 在 UBUNTU 下预处理的命令	- 5 -
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	- 9 -
3.4 本章小结	- 9 -
第 4 章 汇编	- 14 -
4.1 汇编的概念与作用	- 14 -
4.2 在 UBUNTU 下汇编的命令	- 14 -
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	- 18 -
第 5 章 链接	- 19 -
5.1 链接的概念与作用	- 19 -
5.2 在 UBUNTU 下链接的命令	- 19 -
5.3 可执行目标文件 HELLO 的格式	- 19 -
5.4 HELLO 的虚拟地址空间	- 21 -
5.5 链接的重定位过程分析	- 21 -
5.6 HELLO 的执行流程	- 25 -
5.7 HELLO 的动态链接分析	- 25 -
5.8 本章小结	- 26 -
第 6 章 HELLO 进程管理	- 27 -
6.1 进程的概念与作用	- 27 -

6.2 简述壳 SHELL-BASH 的作用与处理流程.....	- 27 -
6.3 HELLO 的 FORK 进程创建过程	- 27 -
6.4 HELLO 的 EXECVE 过程	- 28 -
6.5 HELLO 的进程执行.....	- 28 -
6.6 HELLO 的异常与信号处理	- 29 -
6.7 本章小结	- 32 -
第 7 章 HELLO 的存储管理.....	- 33 -
7.1 HELLO 的存储器地址空间	- 33 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理.....	- 33 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理	- 34 -
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	- 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	- 42 -
第 8 章 HELLO 的 IO 管理	- 44 -
8.1 LINUX 的 IO 设备管理方法	- 44 -
8.2 简述 UNIX IO 接口及其函数	- 44 -
8.3 PRINTF 的实现分析.....	- 45 -
8.4 GETCHAR 的实现分析.....	- 47 -
8.5 本章小结	- 48 -
结论	- 48 -
附件	- 49 -
参考文献.....	- 50 -

第 1 章 概述

1.1 Hello 简介

P2P: 在 linux 中 hello.c 经过 cpp 的预处理、ccl 的编译、as 的汇编、ld 的链接最终 成为可执行目标程序 hello。在 shell 中输入./hello 启动命令后, shell 为其 fork, 产生子进程, 于是 hello 从 Program 变成 Process。

020: hello 变成 process 后, shell 为其 execve, 更新代码段, 数据段, 堆 栈段。映射虚拟内存, 进入程序入口后程序开始载入物理内存, 然后进入 main 函数执行目标代码, CPU 为运行的 hello 分配时间片执行逻辑控制流。当程序运行结束后, shell 父进程负责回收 hello 进程, 内核删除相关数据结构。

1.2 环境与工具

硬件环境: X64 CPU; 2GHz; 2G RAM; 256GHD Disk

软件环境: Ubuntu 16.04 LTS 64 位/优麒麟 64 位

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

1.3 中间结果

文件名称	文件作用
hello.i	hello预处理之后文本文件
hello.s	hello编译之后的汇编文件
hello.o	hello汇编之后的可重定位的二进制文件
hello	hello链接后的可执行文件
helloelf	Hello.o 的 ELF 格式
helloobj	Hello.o 的反汇编代码
helloelf2	Hello的 ELF 格式
helloobj	Hello 的反汇编代码

1.4 本章小结

本章主要简单介绍了 hello 的 p2p, 020 过程, 列出了本次实验信息: 硬件与软件环境、开发与测试工具、中间结果等。

(第 1 章 0.5 分)

第 2 章 预处理

2.1 预处理的概念与作用

概念：预处理器 `cpp` 根据以字符`#`开头的命令（宏定义、条件编译），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。

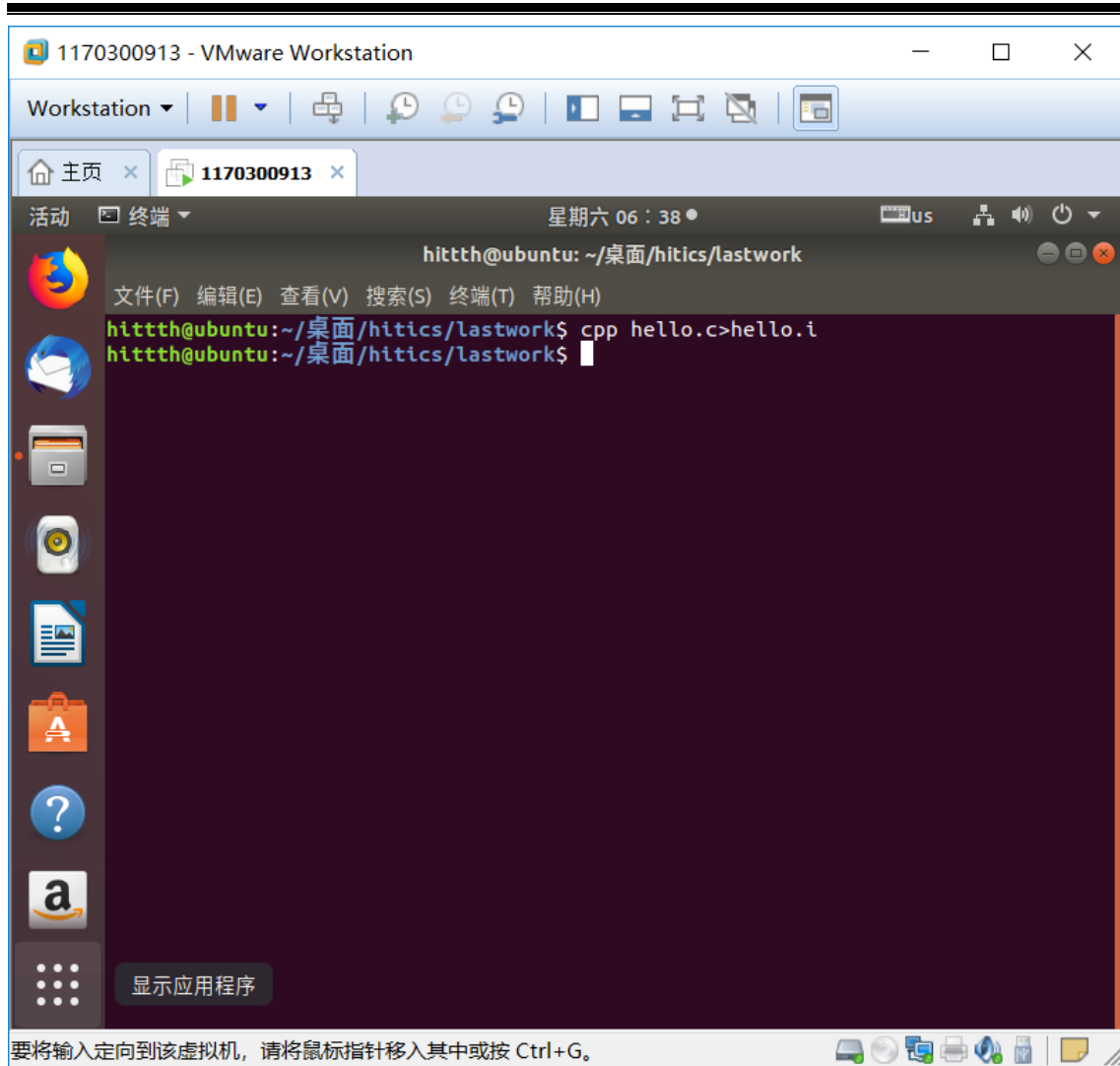
主要功能如下：

1、将源文件中用`#include` 形式声明的文件复制到新的程序中。比如 `hello.c` 第 6-8 行中的`#include` 等命令告诉预处理器读取系统头文件 `stdio.h` `unistd.h` `stdlib.h` 的内容，并把它直接插入到程序文本中。

2、用实际值替换用`#define` 定义的字符串

3、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



2.3 Hello 的预处理结果解析

用 VS 打开 hello.c 之后发现，整个文件已经被扩展成了 3118 行：

```
3101
3102 int main(int argc, char *argv[])
3103 {
3104     int i;
3105
3106     if(argc!=3)
3107     {
3108         printf("Usage: Hello 学号 姓名! \n");
3109         exit(1);
3110     }
3111     for(i=0;i<10;i++)
3112     {
3113         printf("Hello %s %s\n", argv[1], argv[2]);
3114         sleep(sleepsecs);
3115     }
3116     getchar();
3117     return 0;
3118 }
3119
```

而前面的3000行就是.c文件中包含的头文件,这里体现的就是预处理器根据以字符#开头的命令,修改原始的C程序.

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

2.4 本章小结

本章主要介绍了预处理的概念与作用,执行预处理的命令,并结合hello.i文件,解析了hello的预处理结果.

可见预处理过程是只后所有操作的基础,是无可或缺的重要过程.

(第2章 0.5分)

第 3 章 编译

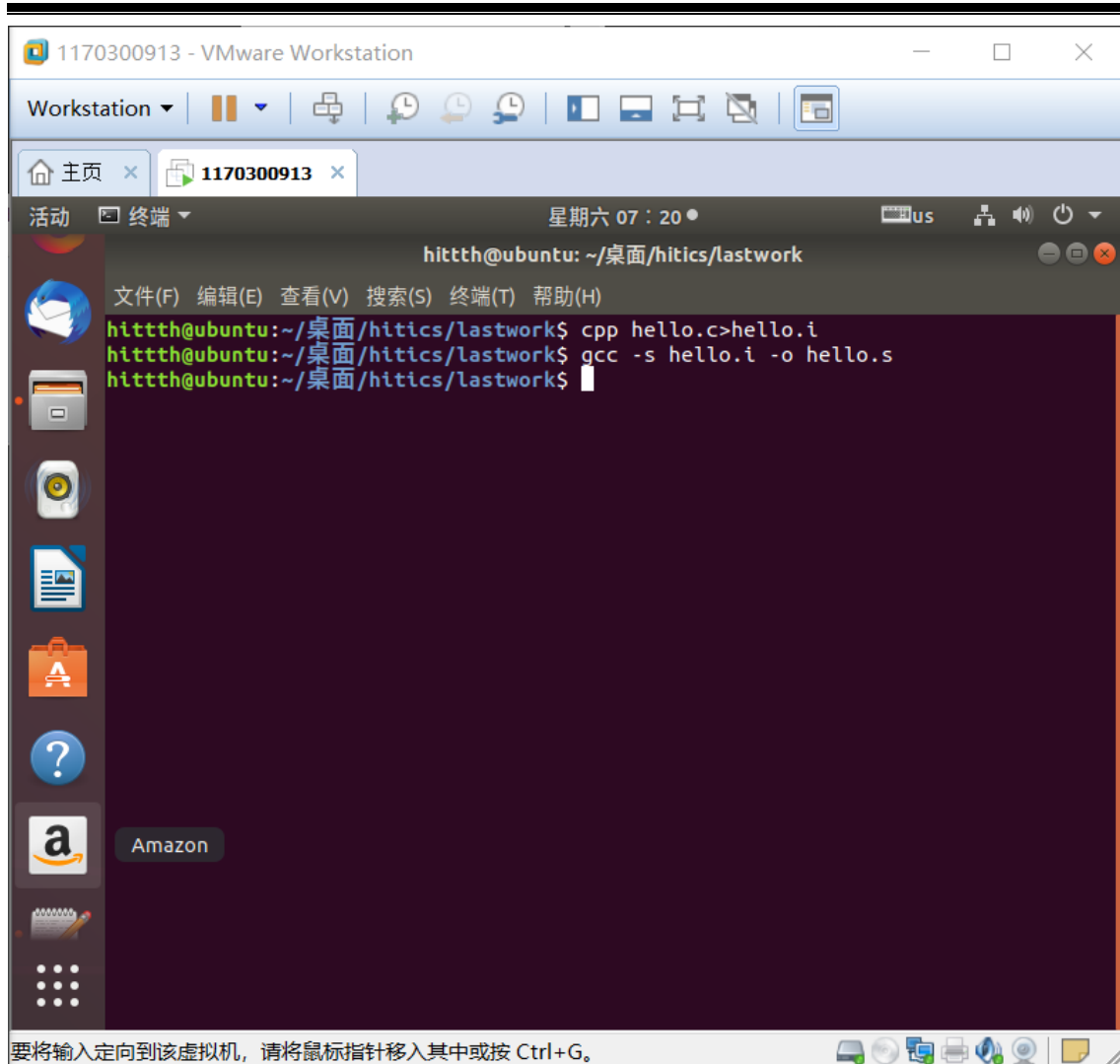
3.1 编译的概念与作用

编译器（cc1）将文本文件hello.i翻译成文本文件hello.s,它包含一个汇编语言程序.该程序包含函数main的定义,这个过程称为编译.

编译的作用就是将高级语言源程序翻译成等价的目标程序,并且进行语法检查、调试措施、修改手段、覆盖处理、目标程序优化等步骤.

3.2 在 Ubuntu 下编译的命令

输入 `gcc -S hello.i -o hello.s`



3.3 Hello 的编译结果解析

3.3.1 数据的声明和赋值

1. Int型数据sleepsecs

被定义为long型并赋值的全局变量，拿来存放已初始化操作的全局和静态C变量的data节,所以编译器处理时在.data节声明该变量

声明:

```
.globl sleepsecs
.data
.align 4
.type sleepsecs, @object
.size sleepsecs, 4
```

赋值操作:

```
sleepsecs:
    .long    2
    .section .rodata
```

2. 局部变量 `int i`

编译器将局部变量存储在寄存器或者栈空间中.在hello.s中编译器将*i*存储在栈上空间-4(%rbp)中

对*i*进行赋值:

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

3.3.2 类型转换

在对int 数据sleepsecs的赋值过程有一次隐式类型转换,.c文件中int sleepsecs=2.5;但是2.5会被改成2,原因是当在double或float向int进行类型转换的时候,值会向零舍入.例如1.999将被转换成1,-1.999将被转换成-1.

其他类型转换原则如下: (摘自书本)

当在 int、float 和 double 格式之间进行强制类型转换时,程序改变数值和位模式的原则如下(假设 int 是 32 位的):

- 从 int 转换成 float, 数字不会溢出, 但是可能被舍入。
- 从 int 或 float 转换成 double, 因为 double 有更大的范围(也就是可表示值的范围), 也有更高的精度(也就是有效位数), 所以能够保留精确的数值。
- 从 double 转换成 float, 因为范围要小一些, 所以值可能溢出成 $+\infty$ 或 $-\infty$ 。另外, 由于精确度较小, 它还可能被舍入。
- 从 float 或者 double 转换成 int, 值将会向零舍入。例如, 1.999 将被转换成 1, 而-1.999 将被转换成-1。进一步来说, 值可能会溢出。C 语言标准没有对这种情况指定固定的结果。与 Intel 兼容的微处理器指定位模式 $[10\cdots 00]$ (字长为 w 时的 $TMin_w$)为整数不确定(integer indefinite)值。一个从浮点数到整数的转换, 如果不能为该浮点数找到一个合理的整数近似值, 就会产生这样一个值。因此, 表达式 $(int)+1e10$ 会得到-21483648, 即从一个正值变成了一个负值。

3.3.3 算数操作

汇编指令如下:

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>	$D \leftarrow D \vee S$	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

编译器将`i++`编译成

```
addl $1, -4(%rbp)
```

3.3.4 逻辑/位操作

编译器将`argc!=3`编译成

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

`je`指令在上一个比较语句结果是相同时生效

3.3.5 关系操作

编译器将`i<10`编译成

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

小于等于9,符合条件跳转

3.3.6 数组/指针/结构操作

数组`char *argv[]`

是一个指针数组,其中每个元素指向函数执行时输入的各个命令行参数,起始地址`argv`存在`%rsi`中,在`main`中被存入栈中。

```
movq %rsi, -32(%rbp) //保存argv
```

然后通过`%rbp+`（偏移量）的方式读取：

`.L4:`

```
movq -32(%rbp), %rax
addq $16, %rax //argv[1]的地址
movq (%rax), %rdx //取出argv[1]指向的数据
movq -32(%rbp), %rax
addq $8, %rax //argv[2]的地址
movq (%rax), %rax //取出argv[2]指向的数据
```

3.3.7 控制转移

1. if(argc!=3)这个条件判断被编译成

```

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

```

je指令在上一个比较语句结果是相同时执行跳转,实现了这个if判断

2. for(i=0;i<10;i++)这个循环语句被编译成

```

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

```

-4(%rbp)就是存放的i,每次执行+1,cmpl指令把i和立即数9比较,小于等于9,符合条件,jle指令执行跳转,实现了这个for循环.

3.3.8 函数操作**1.main 函数**

函数调用: Main函数被.call调用 (__libc_start_main),call指令将下一个指令的地址压入栈中,然后跳转到main执行.

参数传递: 向main函数传递的参数argc和argv,分别使用%rdi和%rsi存储

函数返回: 函数设置%eax为0后就正常退出leave.

```

    movl    $0, %eax
    leave

```

2.printf 函数

函数调用&参数传递: 两次.

第一次:

```
printf("Usage: Hello 学号 姓名! \n");
```

将%rdi设置为“Usage: Hello 学号姓名! \n”字符串的首地址.

```
leaq .LC0(%rip), %rdi
```

其中.LC0(%rip)为:

```

.LC0:
    .string "Usage: Hello \345\255\246\345\217\267 \345\247\2

```

然后callputs@PLT (只有一个字符串参数)

第二次:

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

将%rdi设置%rdi为“Hello %s %s\n”的首地址

```
leaq .LC1(%rip), %rdi
```

其中.LC1(%rip)为

```

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

```

然后callprintf@PLT

3.sleep 函数

函数调用&参数传递：只有一个参数,将%edi设置为想要休眠的秒数传参,然后call sleep@PLT:

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

3. getchar 函数

函数调用&参数传递：无参数,直接 call getchar@PLT

3.4 本章小结

本章节主要叙述了编译器是如何处理c语言程序的,结合c语言中的各种数据类型,各类运算,各类函数操作,逐个分析编译器的具体行为.

在编译阶段,编译器将高级语言编译成汇编语言.汇编语言是直接面向处理器的语言.汇编语言指令是机器指令的一种符号表示,而不同类型的CPU 有不同的机器指令系统,也就有不同的汇编语言,所以,汇编语言程序与机器有着密切的关系.

(第3章2分)

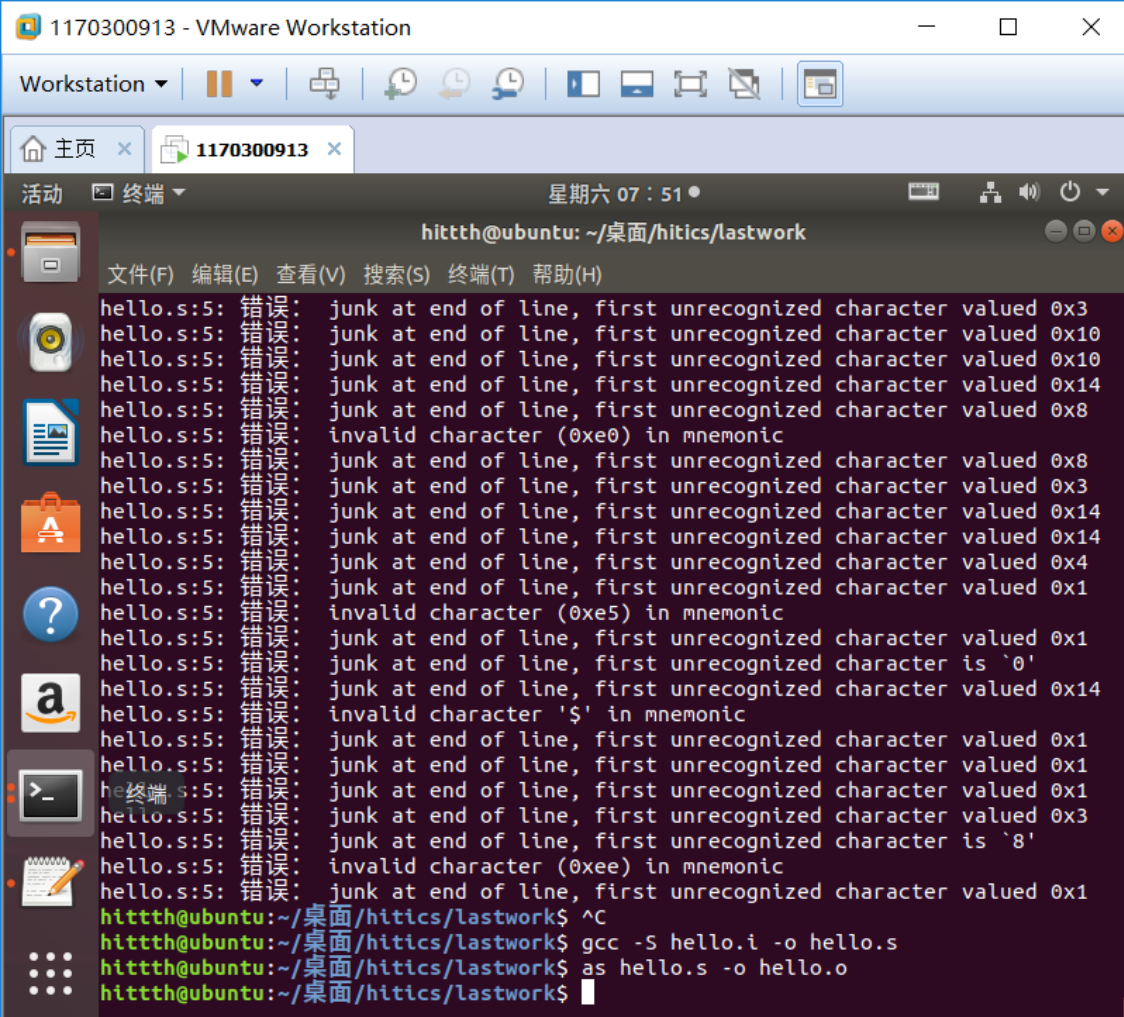
第 4 章 汇编

4.1 汇编的概念与作用

汇编器(as)将.s 汇编程序翻译成机器语言指令,把这些指令打包成可重定位目标程序的格式,并将结果保存在.o 目标文件中,.o 文件是一个二进制文件,它包含程序的指令编码。这个过程称为汇编,亦即汇编的作用。

4.2 在 Ubuntu 下汇编的命令

命令: `as hello.s -o hello.o`



```
1170300913 - VMware Workstation
Workstation
1170300913
活动 终端 星期六 07:51
hitth@ubuntu: ~/桌面/hitcs/lastwork
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
hello.s:5: 错误: junk at end of line, first unrecognized character valued 0x3
hello.s:5: 错误: junk at end of line, first unrecognized character valued 0x10
hello.s:5: 错误: junk at end of line, first unrecognized character valued 0x10
hello.s:5: 错误: junk at end of line, first unrecognized character valued 0x14
hello.s:5: 错误: junk at end of line, first unrecognized character valued 0x8
hello.s:5: 错误: invalid character (0xe0) in mnemonic
hello.s:5: 错误: junk at end of line, first unrecognized character valued 0x8
hello.s:5: 错误: junk at end of line, first unrecognized character valued 0x3
hello.s:5: 错误: junk at end of line, first unrecognized character valued 0x14
hello.s:5: 错误: junk at end of line, first unrecognized character valued 0x14
hello.s:5: 错误: junk at end of line, first unrecognized character valued 0x4
hello.s:5: 错误: junk at end of line, first unrecognized character valued 0x1
hello.s:5: 错误: invalid character (0xe5) in mnemonic
hello.s:5: 错误: junk at end of line, first unrecognized character valued 0x1
hello.s:5: 错误: junk at end of line, first unrecognized character is `0`
hello.s:5: 错误: junk at end of line, first unrecognized character valued 0x14
hello.s:5: 错误: invalid character `$' in mnemonic
hello.s:5: 错误: junk at end of line, first unrecognized character valued 0x1
hello.s:5: 错误: junk at end of line, first unrecognized character valued 0x1
hello.s:5: 错误: junk at end of line, first unrecognized character valued 0x1
hello.s:5: 错误: junk at end of line, first unrecognized character valued 0x3
hello.s:5: 错误: junk at end of line, first unrecognized character is `8`
hello.s:5: 错误: invalid character (0xee) in mnemonic
hello.s:5: 错误: junk at end of line, first unrecognized character valued 0x1
hitth@ubuntu:~/桌面/hitcs/lastwork$ ^C
hitth@ubuntu:~/桌面/hitcs/lastwork$ gcc -S hello.i -o hello.s
hitth@ubuntu:~/桌面/hitcs/lastwork$ as hello.s -o hello.o
hitth@ubuntu:~/桌面/hitcs/lastwork$
```

要将输入定向到该虚拟机, 请将鼠标指针移入其中或按 Ctrl+G。

4.3 可重定位目标 elf 格式

使用 `readelf -a hello.o > helloelf` 指令获得 `hello.o` 文件的 ELF 格式。

```
hittth@ubuntu:~/桌面/hitics/lastwork$ readelf -a hello.o > elf.txt
hittth@ubuntu:~/桌面/hitics/lastwork$
```

1. Elf 头

Elf头以一个16进制的序列开始,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、机器类型、字节头部表(section header table)的文件偏移,以及节头部表中条目的大小和数量等信息.不同节的位置和大小是由节头部表描述的,其中目标文件中每个节都有一个固定大小的条目.

```

ELF 头:
  Magic:      7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  类别:                               ELF64
  数据:                               2 补码, 小端序 (little endian)
  版本:                               1 (current)
  OS/ABI:                               UNIX - System V
  ABI 版本:                               0
  类型:                               REL (可重定位文件)
  系统架构:                               Advanced Micro Devices X86-64
  版本:                               0x1
  入口点地址:                               0x0
  程序头起点:                               0 (bytes into file)
  Start of section headers:               1152 (bytes into file)
  标志:                               0x0
  本头的大小:                               64 (字节)
  程序头大小:                               0 (字节)
  Number of program headers:               0
  节头大小:                               64 (字节)
  节头数量:                               13
  字符串表索引节头:                       12
  
```

1. 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 0000000000000081	PROGBITS 0000000000000000	0000000000000000 AX	0	0	1
[2]	.rela.text 00000000000000c0	RELA 0000000000000018	0000000000000000 I	10	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 000000000000002b	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	10	8	8
[10]	.symtab 00000000000000198	SYMTAB 0000000000000018	0000000000000000	11	9	8
[11]	.strtab 000000000000004d	STRTAB 0000000000000000	0000000000000000	0	0	1
[12]	.shstrtab 0000000000000061	STRTAB 0000000000000000	0000000000000000	0	0	1

包含文件中出现的各个节的语义,包括节的类型、位置和大小等信息.

2. 重定位节 '.rela.text'

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

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

其中

偏移量: 需要进行重定向的代码在.text或.data节中的偏移位置.

信息: 包括symbol和type两部分,其中symbol占前4个字节,type占后4个字节,symbol代表重定位到的目标在.symtab中的偏移量,type代表重定位的类型: 由信息决定的替换类型

符号名称: 重定位目标的名字

加数: 重定位过程需要加减的常量

3. 符号表.symtab

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

以sleepsecs为例:

链接器根据info信息向.symtab节中查询链接目标的符号,由info.symbol=0x09,可以发现重定位目标链接到sleepsecs处.

重定位条目r由四个字段组成:

r.offset=0x60

r.symbol=sleepsecs

r.type=R_X86_64_PC32

r.addend=-4,

R_X86_64_PC32 重定位算法摘抄书本如下:

```
if (r.type == R_X86_64_PC32) {
    refaddr = ADDR(s) + r.offset; /* ref's run-time address */
    *refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr);
}
```

4.4 Hello.o 的结果解析

使用命令 `objdump -d -r hello.o > objdump.txt` :

```
hitth@ubuntu:~/桌面/hitcs/lastwork$ objdump -d -r hello.o>objump.txt
hitth@ubuntu:~/桌面/hitcs/lastwork$
```

机器语言指的是二进制的机器指令集合,而机器指令是由操作码和操作数构成的.汇编语言的主体是汇编指令.汇编指令和机器指令的差别在于指令的表示方法上,汇编指令是机器指令便于记忆的书写格式.

在对比两个文件后,汇编器在汇编hello.s时:

函数调用..s文件中,函数调用之后直接跟着函数名称,而在反汇编程序中,call的目标地址是全0.这是因为hello.c中调用的函数都需要通过动态链接器才能确定函数的运行时地址,在汇编成为机器语言的时候,对于这些不确定地址的函数调用,将

其call指令后的相对地址设置为全0,然后在.rela.text节中为其添加重定位条目,等待被修改.

全局变量访问和分支转移:在.s文件中的段名称会被修改为0并且添加重定位条目.

反汇编代码跳转指令的操作数使用的不是段名称如.L3,因为段名称只是在汇编语言中便于编写的助记符.

4.5 本章小结

本章介绍了通过汇编器(as)将 hello.s 汇编程序翻译成机器语言指令,把这些指令打包成可重定位目标程序的格式的过程。最终生成文件为 hello.o。并且生成了 hello.o 的 ELF 文件格式,介绍了其中的内容。通过 hello.o 获得反汇编代码,并且与 hello.s 文件进行了对比。发现反汇编后分支转移,函数调用,全局变量访问的方式有所不同。间接了解到从汇编语言映射到机器语言汇编器需要实现的转换。

(第4章1分)

第 5 章 链接

5.1 链接的概念与作用

链接是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。链接是由叫做链接器的程序执行的。链接器使得分离编译成为可能。

5.2 在 Ubuntu 下链接的命令

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`

```
hitth@ubuntu:~/桌面/hitcs/lastwork$ objdump -d -r hello.o>objump.txt
hitth@ubuntu:~/桌面/hitcs/lastwork$ 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
hitth@ubuntu:~/桌面/hitcs/lastwork$
```

5.3 可执行目标文件 hello 的格式

指令：`readelf -a hello > 5.3helloelf.txt`

```
hitth@ubuntu:~/桌面/hitcs/lastwork$ readelf -a hello > 5.3helloelf.txt
hitth@ubuntu:~/桌面/hitcs/lastwork$
```

打开导出的 TXT 文件可以发现：

1170300913 - VMware Workstation

Workstation

主页 1170300913

活动 文本编辑器 星期六 08:53 us

5.3helloelf.txt
~/桌面/hitics/lastwork

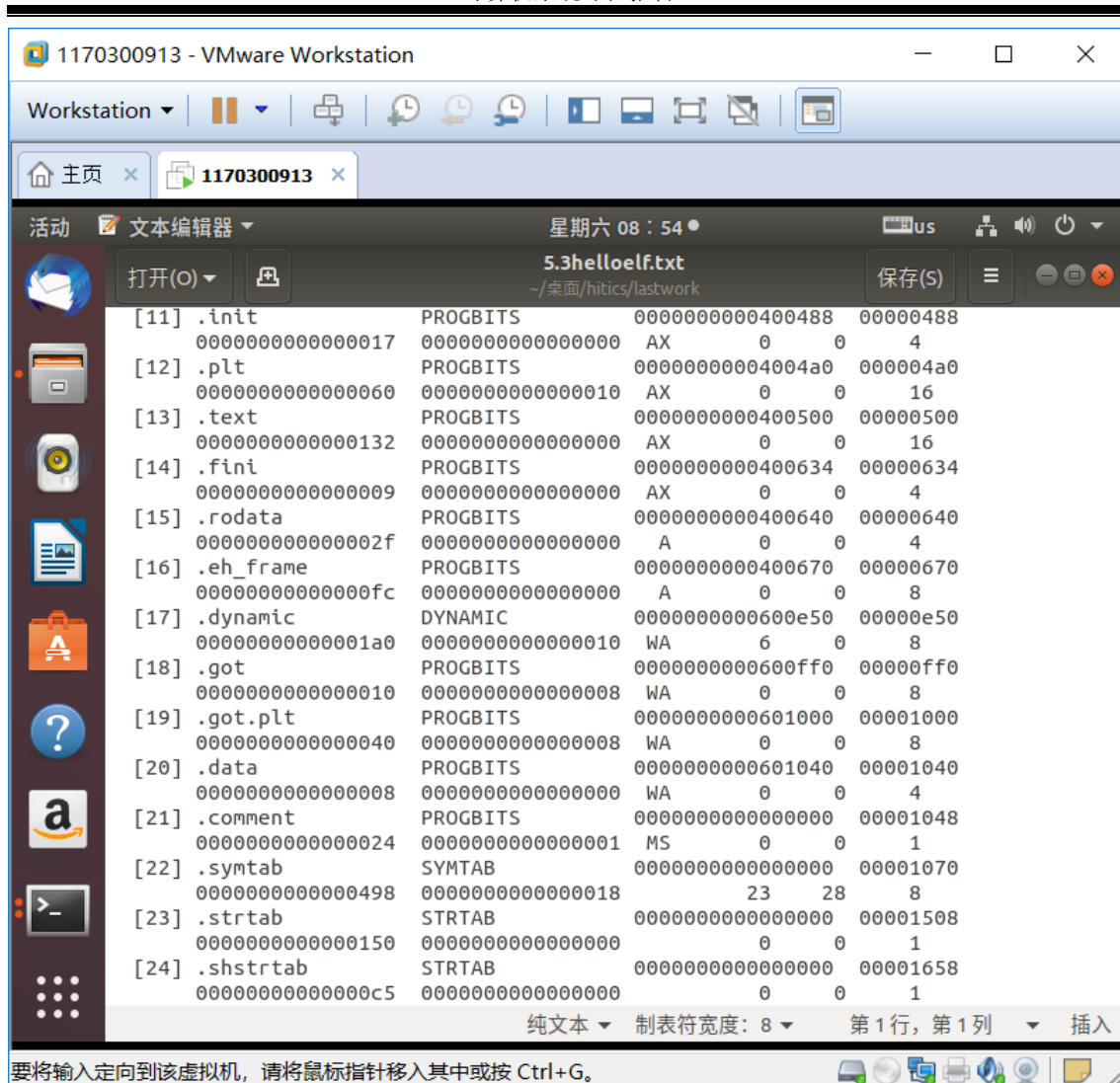
保存(S)

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 000000000000001c	PROGBITS	000000000400200 A 0 0	00000200 1
[2]	.note.ABI-tag 0000000000000020	NOTE	00000000040021c A 0 0	0000021c 4
[3]	.hash 0000000000000034	HASH	000000000400240 A 5 0	00000240 8
[4]	.gnu.hash 000000000000001c	GNU_HASH	000000000400278 A 5 0	00000278 8
[5]	.dynsym 00000000000000c0	DYNSYM	000000000400298 A 6 1	00000298 8
[6]	.dynstr 0000000000000057	STRTAB	000000000400358 A 0 0	00000358 1
[7]	.gnu.version 0000000000000010	VERSYM	0000000004003b0 A 5 0	000003b0 2
[8]	.gnu.version_r 0000000000000020	VERNEED	0000000004003c0 A 6 1	000003c0 8
[9]	.rela.dyn 0000000000000030	RELA	0000000004003e0 A 5 0	000003e0 8
[10]	.rela.plt 0000000000000078	RELA	000000000400410 AI 5 19	00000410 8
[11]	.init	PROGBITS	000000000400488	00000488

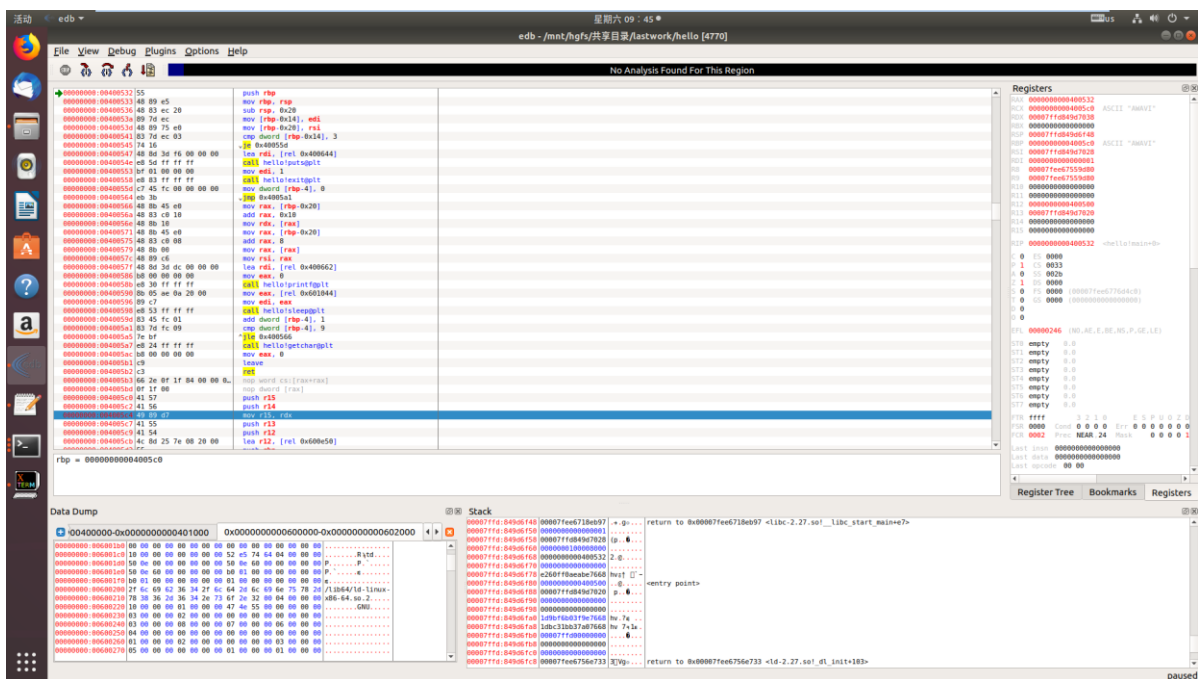
纯文本 制表符宽度: 8 第1行, 第1列 插入

要将输入定向到该虚拟机, 请将鼠标指针移入其中或按 Ctrl+G。



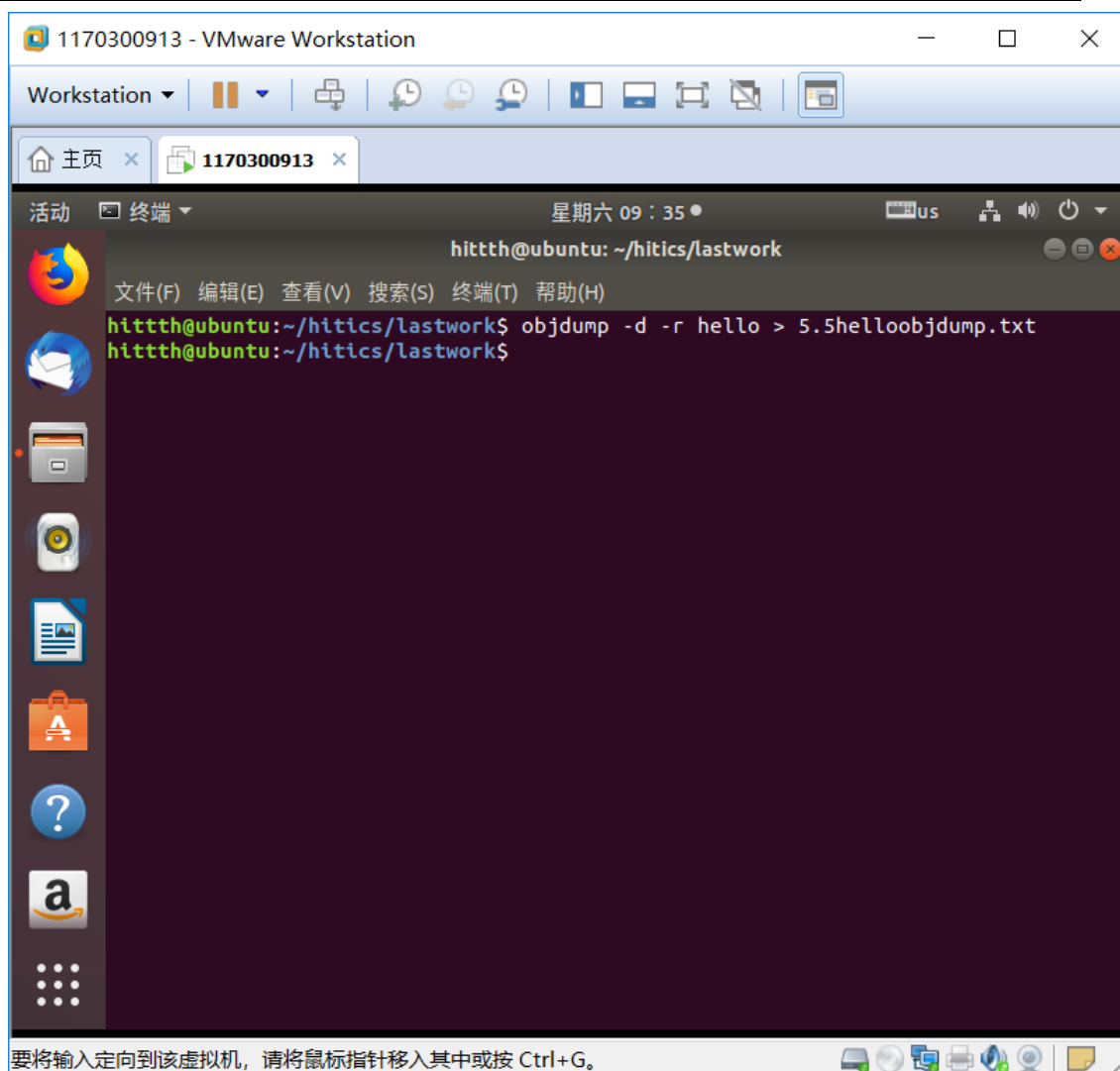
5.4 hello 的虚拟地址空间

使用edb打开hello程序,通过edb的Data Dump窗口查看加载到虚拟地址中的hello程序.



5.5 链接的重定位过程分析

输入命令：objdump -d -r hello > 5.5helloobjdump.txt



打开5.5shelloobjdump.txt:

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

Disassembly of section .plt:

00000000004004a0 <.plt>:
4004a0: ff 35 62 0b 20 00 pushq  0x200b62(%rip)      # 601008 <_GLOBAL_OFFSET_TABI
4004a6: ff 25 64 0b 20 00 jmpq    *0x200b64(%rip)      # 601010 <_GLOBAL_OFFSET_TAE
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 pushq  $0x0
4004bb: e9 e0 ff ff ff jmpq    4004a0 <.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>

00000000004004d0 <getchar@plt>:
4004d0: ff 25 52 0b 20 00 jmpq    *0x200b52(%rip)      # 601028 <getchar@GLIBC_2.2.
4004d6: 68 02 00 00 00 00 pushq  $0x2
4004db: e9 c0 ff ff ff jmpq    4004a0 <.plt>
```

1.在使用ld命令链接的时候,指定了动态链接器为64的/lib64/ld-linux-x86-64.so.2, _start程序调用hello.c中的main函数,libc.so是动态链接共享库,其中定义了hello.c中用到的printf、sleep、getchar、exit函数和_start中调用的__libc_csu_init,__libc_csu_fini,__libc_start_main.链接器将上述函数加入.

2.将所有的R_X86_64_PC32和R_X86_64_PLT32替换成计算好的地址
计算如下:

以sleepsecs为例:

链接器根据info信息向.symtab节中查询链接目标的符号,由info.symbol=0x09,可以发现重定位目标链接到sleepsecs处.

重定位条目r由四个字段组成:

r.offset=0x60

r.symbol=sleepsecs

r.type=R_X86_64_PC32

r.addend=-4,

R_X86_64_PC32重定位算法摘抄书本如下:

```
if (r.type == R_X86_64_PC32) {
    refaddr = ADDR(s) + r.offset; /* ref's run-time address */
    *refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr);
}
```

Refaddr=0x400532+0x60=0x400592

*refptr=(unsigned)(ADDR(r.sleepsecs)+r.addend-refaddr)==0x601044+(-0x4)-0x400592=(unsigned) 0x200aae,

验证一下:

```

40058b:  e8 30 ff ff ff      callq 4004c0 <printf@plt>
400590:  8b 05 ae 0a 20 00    mov     0x200aae(%rip),%eax      # 601044 <sleepsec
400596:  00 07               mov     %eax,%edi

```

其他重定位类似。

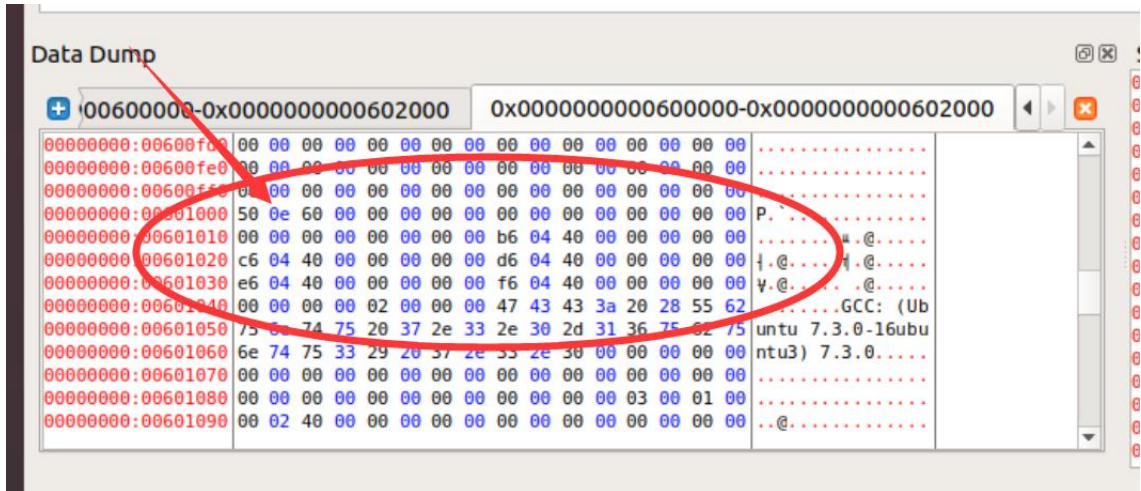
5.6 hello 的执行流程

函数	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

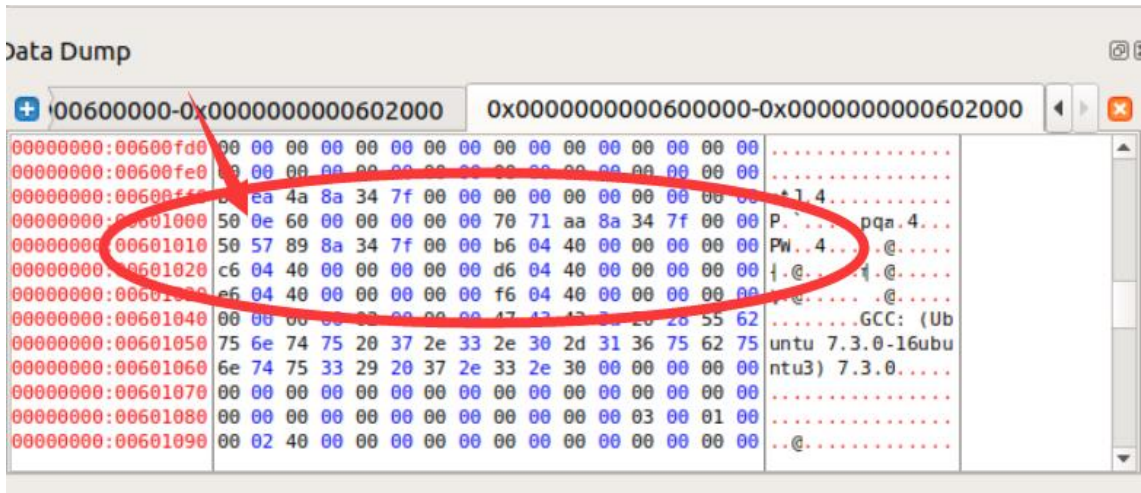
使用edb执行hello,说明从加载hello到_start,到call main,以及程序终止的所有过程.请列出其调用与跳转的各个子程序名或程序地址.

5.7 Hello 的动态链接分析

run 前:



run 后:



5.8 本章小结

本章主要介绍了链接的概念与作用：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，分析了 UBUNTU 下链接的命令，通过对 hello 的反汇编了解了链接的中间过程，所做的工作及函数的调用。了解了链接前后地址的变化以及调用函数的变化。

(第 5 章 1 分)

第 6 章 hello 进程管理

6.1 进程的概念与作用

进程是一个执行中的程序的实例，每一个进程都有它自己的地址空间，一般情况下，包括文本区域、数据区域、和堆栈。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储区着活动过程调用的指令和本地变量。进程为用户提供了以下假象：我们的程序好像是系统中当前运行的唯一程序一样，我们的程序好像是独占的使用处理器和内存，处理器好像是无间断的执行我们程序中的指令，我们程序中的代码和数据好像是系统内存中唯一的对象。

作用：每次用户通过向 shell 输入一个可执行目标文件的名字，运行程序时，shell 就会创建一个新的进程，然后在这个新进程的上下文中运行这个可执行目标文件。应用程序也能够创建新进程，并且在这个新进程的上下文中运行它们自己的代码或其他应用程序。

6.2 简述壳 Shell-bash 的作用与处理流程

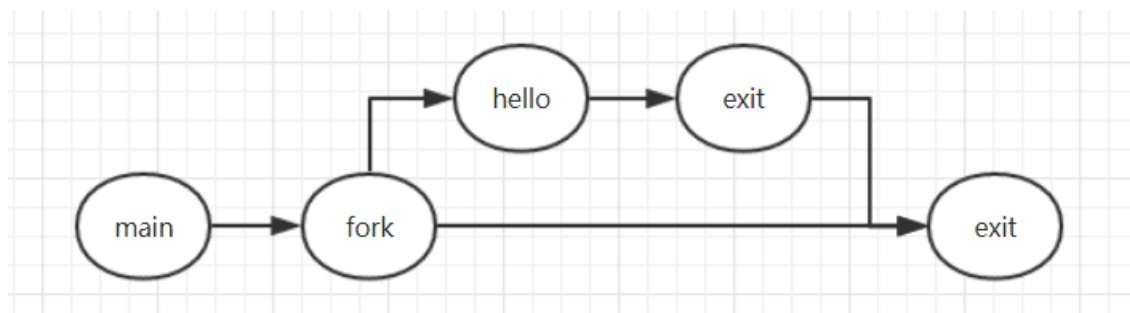
作用：shell 是一个交互型的应用级程序，它代表用户运行其他程序。

处理流程：

- 1) 从终端读入输入的命令。
- 2) 将输入字符串切分获得所有的参数
- 3) 如果是内置命令则立即执行
- 4) 否则调用相应的程序为其分配子进程并运行
- 5) 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

Shell（父进程）通过fork 函数创建一个新的运行的子进程.新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的（但是独立的）一份副本,包括代码和数据段、堆、共享库以及用户栈.子进程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.



6.4 Hello 的 execve 过程

`execve` 函数加载并运行可执行目标文件 `filename`, 且带参数列表 `argv` 和环境变量列表 `envp`. 只有当出现错误时, 例如找不到 `filename`, `execve` 才会返回到调用程序. 所以, 与 `fork` 一次调用返回两次不同, `execve` 调用一次并从不返回.

`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 `main` 函数. 除了一些头部信息, 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, 这时, 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.

旁注 加载器实际是如何工作的?

我们对于加载的描述从概念上来说是正确的, 但也不是完全准确, 这是有意为之. 要理解加载实际是如何工作的, 你必须理解进程、虚拟内存和内存映射的概念, 这些我们还没有加以讨论. 在后面第 8 章和第 9 章中遇到这些概念时, 我们将重新回到加载的问题上, 并逐渐向你揭开它的神秘面纱.

对于不够有耐心的读者, 下面是关于加载实际是如何工作的一个概述: 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. 除了一些头部信息, 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, 此时, 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.

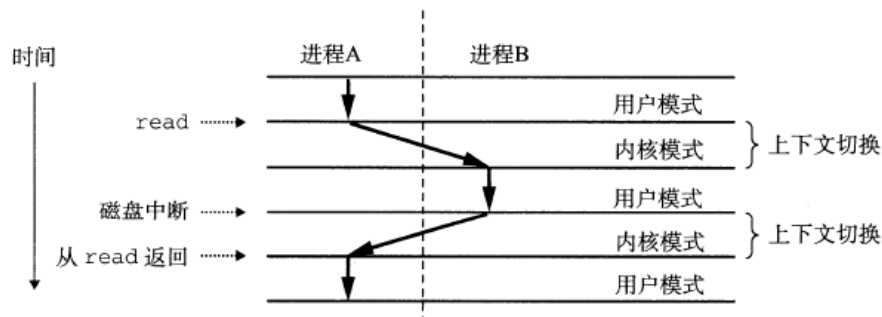
6.5 Hello 的进程执行

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, 这个模式中, 硬件防止特权指令的执行, 并对内存和 I/O 空间的访问操作进行检查. 设置模式位时, 进程处于内核模式, 一切程序都可运行. 任务可以执行特权级指令, 对任何 I/O 设备有全部的访问权, 还能够访问任何虚地址和控制虚拟内存硬件.

上下文信息：上下文程序正确运行所需要的状态,包括存放在内存中的程序的代码和数据,用户栈、寄存器、程序计数器、环境变量和打开的文件描述符的集合构成。

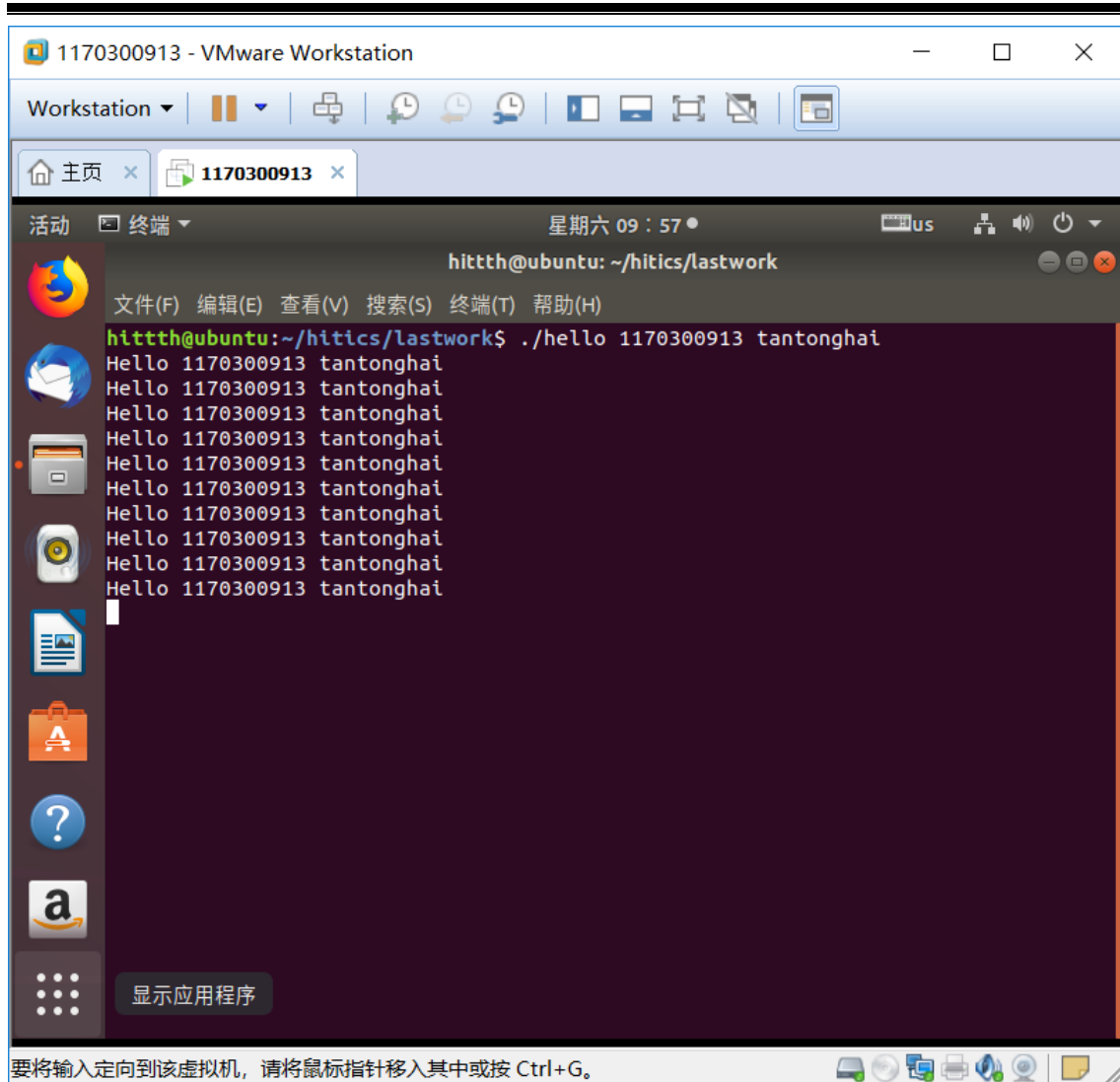
Hello进程执行分析：

Hello起初在用户模式下运行,在hello进程调用sleep之后转入内核模式,内核休眠,并将hello进程从运行队列加入等待队列,定时器开始计时2s,当定时器到时,发送一个中断信号,此时进入内核状态执行中断处理,将hello进程从等待队列中移出重新加入到运行队列,hello进程继续执行。

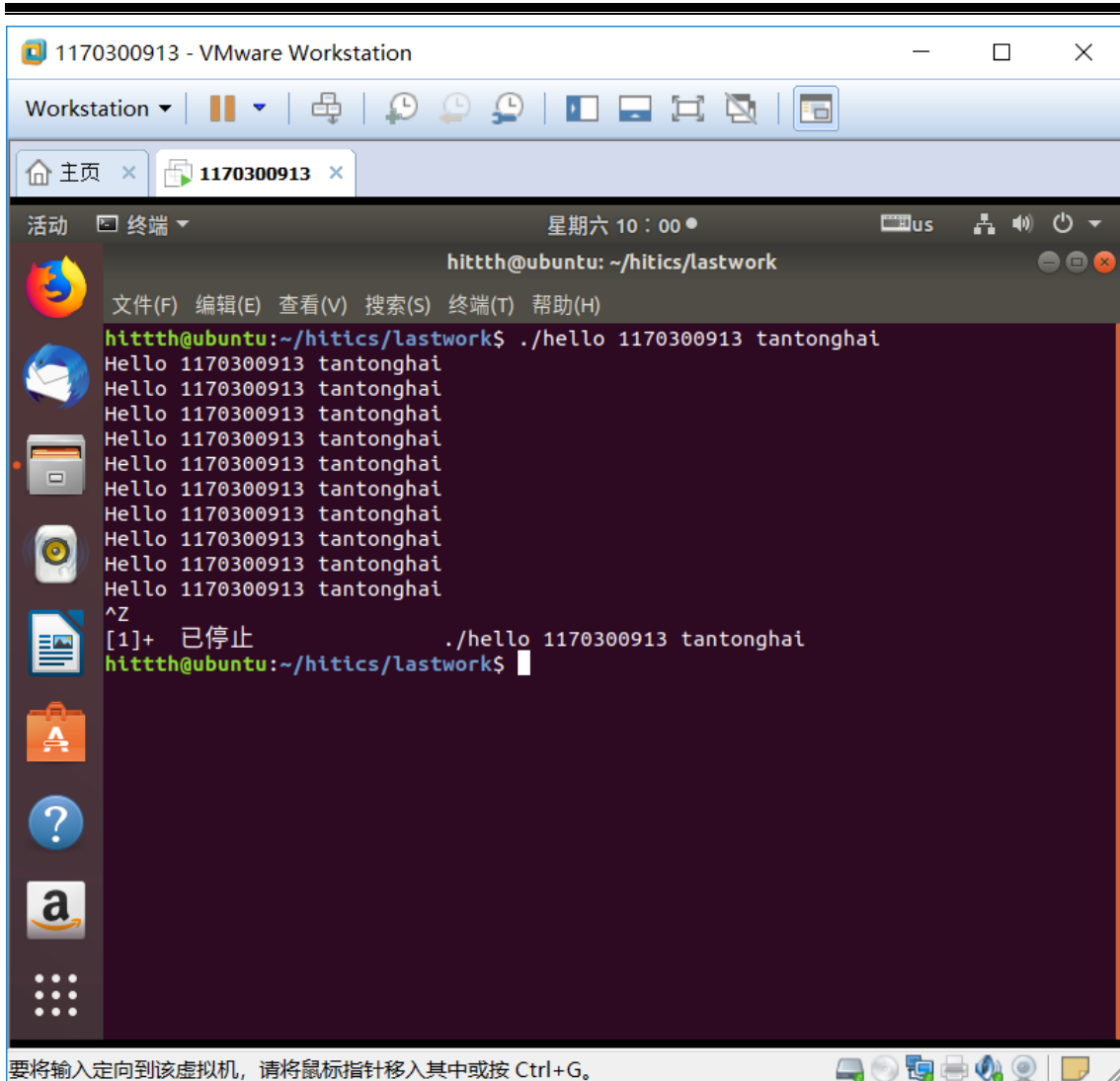


6.6 hello 的异常与信号处理

正常结束：



中途按下ctrl+z,程序挂起,对应的是shell接收到SIGSTP信号,



输入ps指令：可以发现hello进程没有结束

```
hitth@ubuntu:~/hitics/lastwork$ ps
  PID TTY          TIME CMD
  4993 pts/1        00:00:00 bash
  5001 pts/1        00:00:00 hello
  5006 pts/1        00:00:00 ps
hitth@ubuntu:~/hitics/lastwork$
```

输入jobs查看hello和jid

```
hitth@ubuntu:~/hitics/lastwork$ jobs
[1]+ 已停止 ./hello 1170300913 tantonghai
hitth@ubuntu:~/hitics/lastwork$
```

输入fg 1,就可以让hello继续进行,输出还没有输出的内容.

```
hitth@ubuntu:~/hitics/lastwork$ fg 1
./hello 1170300913 tantonghai
^C
```

如果在运行到一半的时候输入ctrl-c,就会发现进程立即结束了.


```
hitth@ubuntu:~/hitcs/lastwork$ ./hello 1170300913 tantonghai
Hello 1170300913 tantonghai
Hello 1170300913 tantonghai
Hello 1170300913 tantonghai
Hello 1170300913 tantonghai
Hello 1170300913 tantonghai
Hello 1170300913 tantonghai
Hello 1170300913 tantonghai
Hello 1170300913 tantonghai
Hello 1170300913 tantonghai
^C
hitth@ubuntu:~/hitcs/lastwork$
```

此时输入ps后台也没有hello了

```
hitth@ubuntu:~/hitcs/lastwork$ ps
  PID TTY          TIME CMD
 4993 pts/1        00:00:00 bash
 5055 pts/1        00:00:00 ps
hitth@ubuntu:~/hitcs/lastwork$
```

6. 7 本章小结

在本章中,介绍了进程的定义和作用,介绍了Shell的处理流程,介绍了shell如何调用fork创建新进程,如何调用execve执行hello,分析了hello的进程执行,还研究了hello的异常与信号处理.

异常控制流发生在计算机系统的各个层次.在硬件层,硬件检测到的事件会触发控制突然转移到异常处理程序.在操作系统层,内核通过上下文切换将控制从一个用户进程转移到另一个用户进程.在应用层,一个进程可以发送信号到另一个进程,而接收者会将控制突然转移到它的一个信号处理程序.一个程序可以通过回避通常的栈规则,并执行到其他函数中任意位置的非本地跳转来对错误做出反应.

(第6章1分)

第 7 章 hello 的存储管理

7.1 hello 的存储器地址空间

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

线性地址：地址空间(address space) 是一个非负整数地址的有序集合，如果地址空间中的整数是连续的，那么我们说它是一个线性地址空间(linear address space)。就是 hello 里面的虚拟内存地址。虚拟地址：CPU 通过生成一个虚拟地址(Virtual Address, VA)。就是 hello 里面的 虚拟内存地址。物理地址：用于内存芯片级的单元寻址，与处理器和 CPU 连接的地址总线相对应。计算机系统的主存被组织成一个由 M 个连续的字节大小的单元组成的数组。每字节都有一个唯一的物理地址。就是 hello 在运行时虚拟内存地址对应的物理地址。

7.2 Intel 逻辑地址到线性地址的变换-段式管理

一个逻辑地址由两部份组成，段标识符：段内偏移量。段标识符是由一个 16 位长的字段组成，称为段选择符。其中前 13 位是一个索引号。后面 3 位包含一些 硬件细节，如图：



图 7.1 段选择符说明

索引号是“段描述符(segment descriptor)”的索引，很多个段描述符，就组了一个数组，叫“段描述符表”，这样，可以通过段标识符的前 13 位，直接在段描述符表中找到一个具体的段描述符，这个描述符就描述了一个段，每一个段描述符由 8 个字节组成，如下图：



图 7.2 段选择符说明

其中 Base 字段，它描述了一个段的开始位置的线性地址，一些全局的段描述符，就放在“全局段描述符表(GDT)”中，一些局部的，例如每个进程自己的，就放在所谓的“局部段描述符表(LDT)”中，由段选择符中的 T1 字段表示选择使用哪个，=0，表示用 GDT，=1 表示用 LDT。GDT 在内存中的地址和大小存放在 CPU 的 gdtr 控制寄存器中，而 LDT 则在 ldtr 寄存器中。如下图：

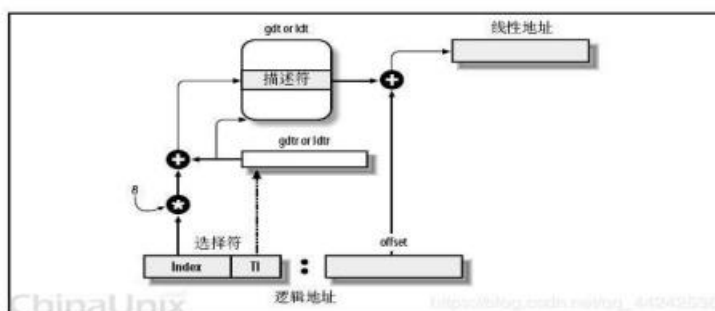


图 7.3 概念关系说明

下面是转换的具体步骤：

给定一个完整的逻辑地址[段选择符：段内偏移地址。

看段选择符的 T1=0 还是 1，知道当前要转换是 GDT 中的段，还是 LDT 中的段，再根据相应寄存器，得到其地址和大小。可以得到一个数组。

取出段选择符中前 13 位，在数组中查找到对应的段描述符，得到 Base，也就是基地址。

线性地址 = Base + offset。

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(page)，例如一个 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，这页，整个线性地址就被划分为一个 $\text{total_page}[2^{20}]$ 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，或者是页框、页帧的。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。这里注意到，这个 total_page 数组有 2^{20} 个成员，每个成员是一个地址（32 位机，一个地址也就是 4 字节），那么要单单要表示这么一个数组，就要占去 4MB 的内存空间。为了节省空间，引入了一个二级管理模式的机器来组织分页单元。如图

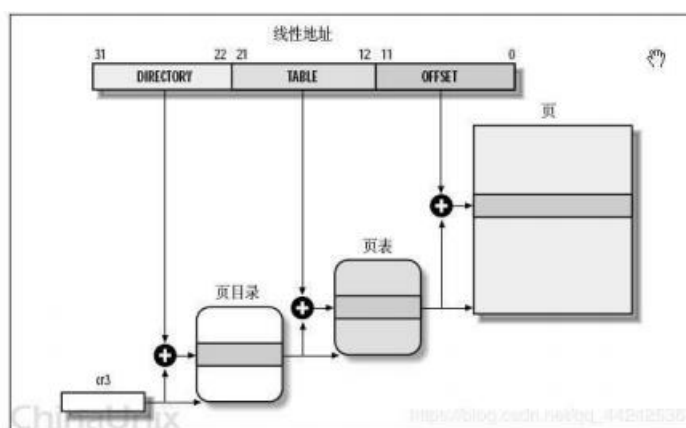


图 7.4 二级管理模式图

由上图可得：

1. 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。

2. 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），那么它也就对应了一个独立的页目录地址。——运行一个进程，需要将它页目录地址放到 cr3 寄存器中

3. 每一个 32 位的线性地址被划分为三部份，页目录索引(10 位)：页表索引(10 位)：偏移(12 位)

依据以下步骤进行转换：

1. 从 cr3 中取出进程的页目录地址（操作系统负责在调度进程的时候，把这个地址装入对应寄存器）。

2. 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。（又引入了一个数

组)，页的地址被放到页表中去了。

3.根据线性地址的中间十位，在页表（也是数组）中找到页的起始地址。

4.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

36位VPN被划分成四个9位的片,每个片被用作到一个页表的偏移量.CR3寄存器包含L1页表的物理地址.VPN1提供到一个L1 PTE的偏移量,这个PTE包含L2页表的基地址.VPN2提供到一个L2 PTE的偏移量,以此类推.

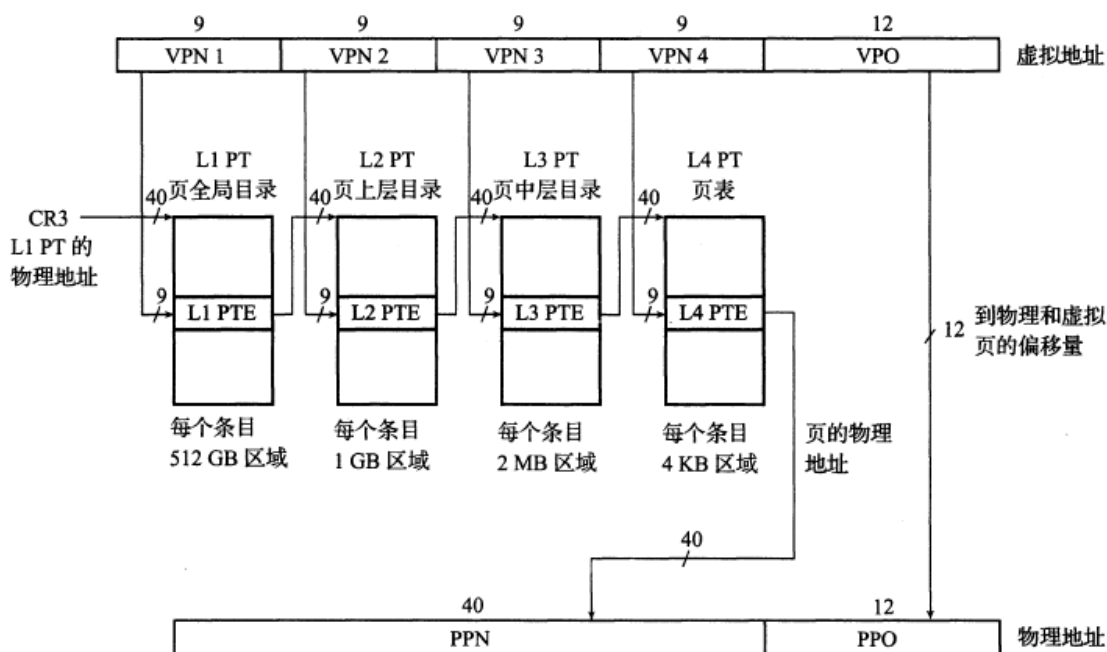


图 9-25 Core i7 页表翻译(PT: 页表, PTE: 页表条目, VPN: 虚拟页号, VPO: 虚拟页偏移, PPN: 物理页号, PPO: 物理页偏移量。图中还给出了这四级页表的 Linux 名字)

在对地址翻译的讨论中,我们描述了一个顺序的两个步骤的过程,1)MMU将虚拟地址翻译成物理地址,2)将物理地址传送到L1高速缓存.然而,实际的硬件实现使用了一个灵活的技巧,允许这些步骤部分重叠,因此也就加速了对L1高速缓存的访问.例如,页面大小为4KB的系统上的一个虚拟地址有12位的VPO,并且这些位和相应物理地址中的PPO的12位是相同的.因为八路组相联的、物理寻址的L1高速缓存有64个组和大小为64字节的缓存块,每个物理地址有6个(log264)缓存偏移位和6个(log264)索引位.这12位恰好符合虚拟地址的VPO部分,这绝不是偶然!当CPU需要翻译一个虚拟地址时,它就发送VPN到MMU,发送VPO到高速L1缓存.当MMU向TLB请求一个页表条目时,L1 高速缓存正忙着利用VPO位查找相应的组,并读出这个组里的个标记和相应的数据字.当MMU从TLB得到PPN时,缓存已经准备好试着把这

个PPN与这8个标记中的一个进行匹配了。

7.5 三级 Cache 支持下的物理内存访问

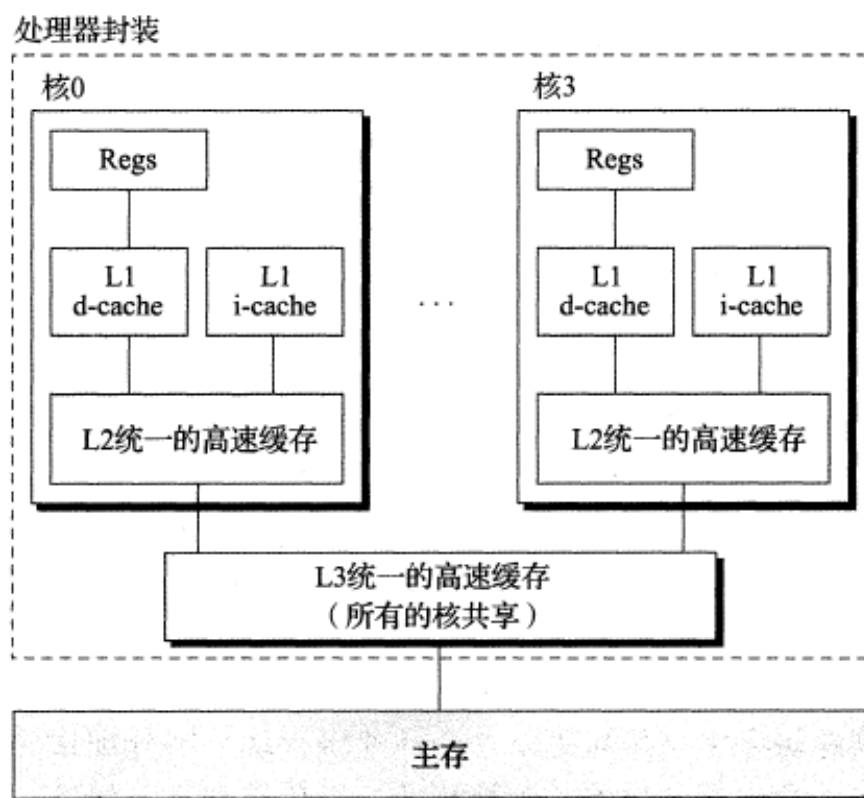
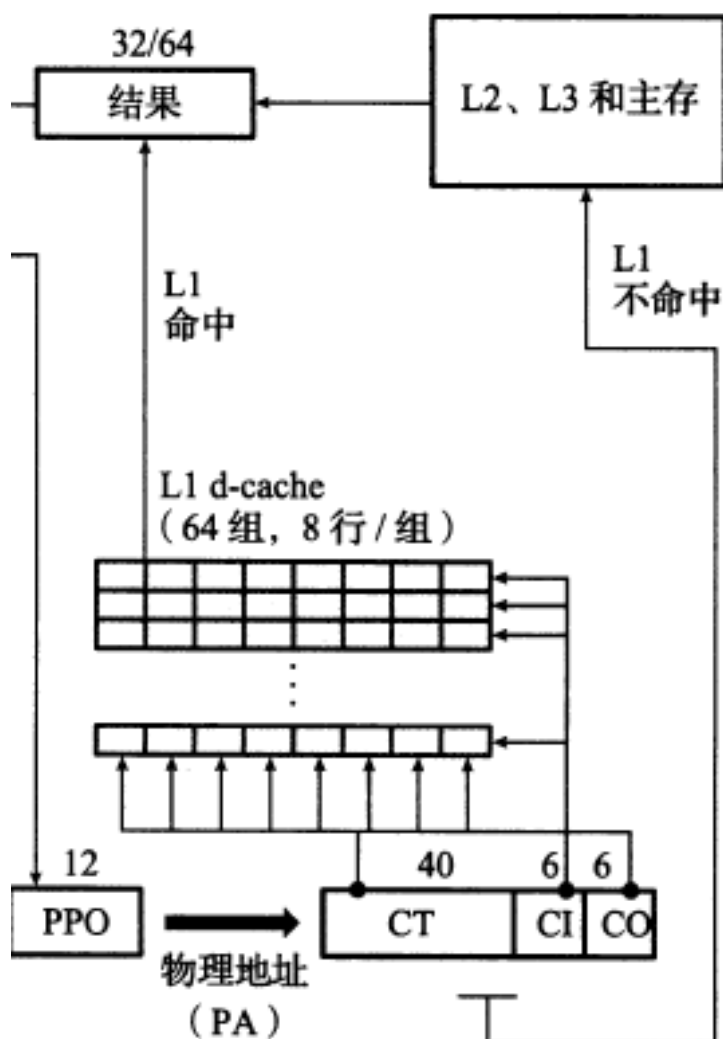


图 6-38 Intel Core i7 的高速缓存层次结构

获得了物理地址VA之后,使用CI(倒数7-12位)进行组索引,每组8路,对8路的块分别匹配CT(前40位)如果匹配成功且块的valid标志位为1,则命中,根据数据偏移量CO(后6位)取出数据返回。

如果没有匹配成功或者匹配成功但是标志位是1,则不命中,向下一级缓存中查询数据(L2 Cache->L3 Cache->主存),查询到数据之后,一种简单的放置策略如下:如果映射到的组内有空闲块,则直接放置,否则组内都是有效块,产生冲突(evict),则采用最近最少使用策略LFU(Least frequently used)进行替换.也就是替换掉最不经访问的一次数据,示意图如下:



7.6 hello 进程 fork 时的内存映射

当 fork 函数被 shell 进程调用时,内核为新进程创建各种数据结构,并分配给它一个唯一的 PID,为了给这个新进程创建虚拟内存,它创建了当前进程的 `mm_struct`、区域结构和页表的原样副本.它将这两个进程的每个页面都标记为只读,并将两个进程中的每个区域结构都标记为私有的写时复制.当 fork 在 hello 进程中返回时,hello 进程现在的虚拟内存刚好和调用 fork 时存在的虚拟内存相同.当这两个进程中的任一个后来进行写操作时,写时复制机制就会创建新页面,因此,也就为每个进程保持了私有地址空间的抽象概念.

7.7 hello 进程 execve 时的内存映射

execve 函数调用驻留在内核区域的启动加载器代码,在当前进程中加载并运行包含在可执行目标文件 hello 中的程序,用 hello 程序有效地替代了当前程序.加载并运行 hello 需要以下几个步骤:

1.删除已存在的用户区域,删除当前进程虚拟地址的用户部分中的已存在的区域结构.

2.映射私有区域,为新程序的代码、数据、bss 和栈区域创建新的区域结构,所有这些新的区域都是私有的、写时复制的.代码和数据区域被映射为 hello 文件中的.text 和.data 区,bss 区域是请求二进制零的,映射到匿名文件,其大小包含在 hello 中,栈和堆地址也是请求二进制零的,初始长度为零.

3.映射共享区域,hello 程序与共享对象 libc.so 链接,libc.so 是动态链接到这个程序中的,然后再映射到用户虚拟地址空间中的共享区域内.

4.设置程序计数器 (PC),execve 做的最后一件事情就是设置当前进程上下文的程序计数器,使之指向代码区域的入口点.

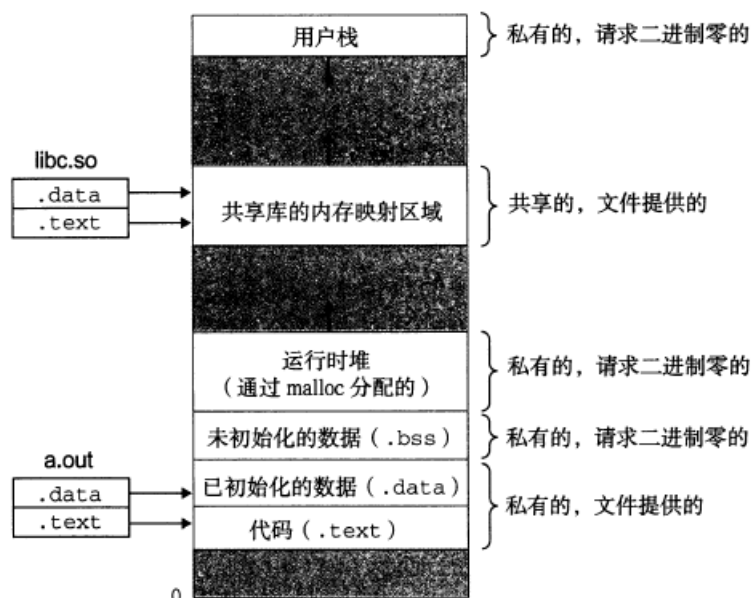


图 9-31 加载器是如何映射用户地址空间的区域的

7.8 缺页故障与缺页中断处理

缺页时的操作

第 1 步:处理器生成一个虚拟地址,并把它传送给 MMU.

第 2 步:MMU 生成 PTE 地址,并从高速缓存/主存请求得到它.

第 3 步:高速缓存/主存向 MMU 返回 PTE.

第 4 步:PTE 中的有效位是零,所以 MMU 触发了一次异常,传递 CPU 中的控制到操作系统内核中的缺页异常处理程序.

第 5 步:缺页处理程序确定出物理内存中的牺牲页,如果这个页面已经被修改了,则把它换出到磁盘.

第 6 步:缺页处理程序页面调入新的页面,并更新内存中的 PTE.

第 7 步:缺页处理程序返回到原来的进程,再次执行导致缺页的指令,CPU 将地址重新发送给 MMU.因为虚拟页面现在已经缓存在物理内存中,所以会命中,主存将所请求字返回给处理器.

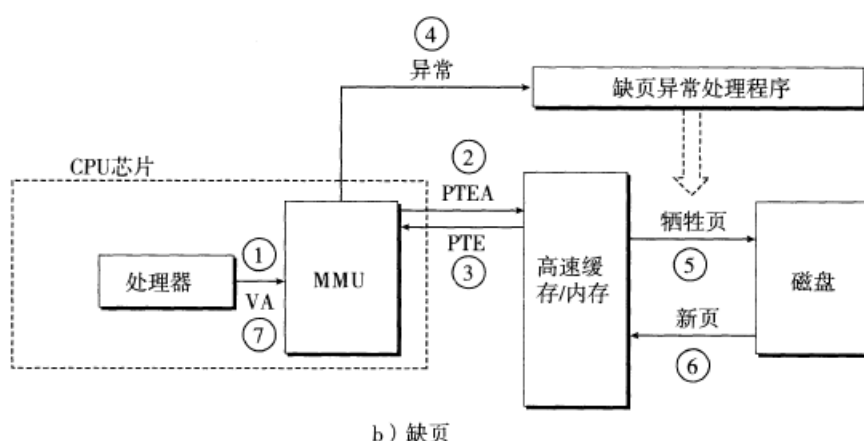


图 9-13 页面命中和缺页的操作图 (VA: 虚拟地址。PTEA: 页表条目地址。
PTE: 页表条目。PA: 物理地址)

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.系统之间细节不同,但是不失通用性,假设堆是一个请求二进制零的区域,它紧接在未初始化的数据区域后开始,并向上生长(向更高的地址).对于每个进程,内核维护着一个变量 `brk`,它指向堆的顶部.

分配器将堆视为一组不同大小的块的集合来维护.每个块就是一个连续的虚拟内存片,要么是已分配的,要么是空闲的.已分配的块显式地保留为供应用程序使用.空闲块可用来分配.空闲块保持空闲,直到它显式地被应用所分配.一个已分配的块

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

分配器有两种基本风格。两种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(explicit allocator):

要求应用显式地释放任何已分配的块。例如,C 标准库提供一种叫做 malloc 程序包的显式分配器.C 程序通过调用 malloc 函数来分配一个块,并通过调用 free 函数来释放一个块.C++中的 new 和 delete 操作符与 C 中的 malloc 和 free 相当。

隐式分配器(implicit allocator):

要求分配器检测一个已分配块何时不再被程序所使用,那么就释放这个块。隐式分配器也叫做垃圾收集器(garbage collector),而自动释放未使用的已分配的块的过程叫做垃圾收集(garbage collection)。例如,诸如 Lisp、ML 以及 Java 之类的高级语言就依赖垃圾收集来释放已分配的块。

隐式空闲链表:

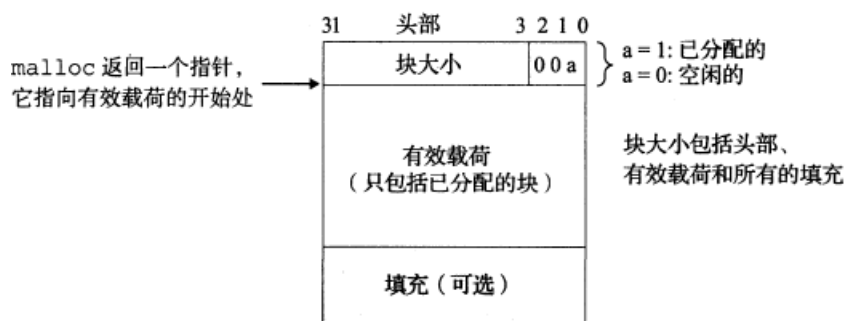


图 9-35 一个简单的堆块的格式

一个块是由一个字的头部,有效载荷,以及可能的一些额外的填充组成的。头部编码了这个块的大小,以及这个块是已分配的还是空闲的。如果我们强加一个双字的对齐约束条件,那么块大小就总是 8 的倍数,且块大小的最低 3 位总是零。因此,我们只需要内存大小的 29 个高位,释放剩余的 3 位来编码其他信息。在这种情况下,我们用其中的最低位(已分配位)来指明这个块是已分配的还是空闲的。

带边界标签的隐式空闲链表:

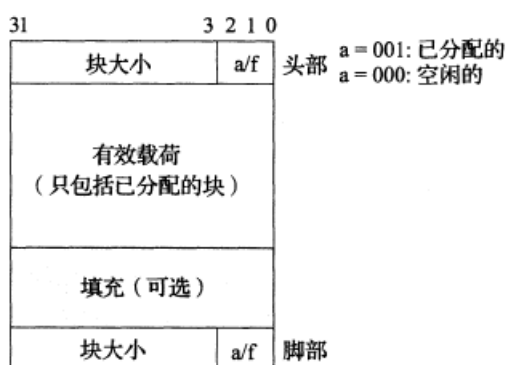


图 9-39 使用边界标记的堆块的格式

在隐式空闲链表堆块的基础上,在每个块的结尾处添加一个脚部(footer),边界标记),其中脚部就是头部的一个副本.如果每个块包括这样一个脚部,那么分配器就可以通过检查它的脚部,判断前面一个块的起始位置和状态,这个脚部总是在距当前块开始位置一个字的距离,这样就允许在常数时间内进行对前面块的合并.

显示空闲链表

一种更好的方法是将空闲块组织为某种形式的显式数据结构.因为根据定义,程序不需要一个空闲块的主体,所以实现这个数据结构的指针可以存放在这些空闲块的主体里面.例如,堆可以组织成一个双向空闲链表,在每个空闲块中,都包含一个 pred(前驱)和 succ(后继)指针.

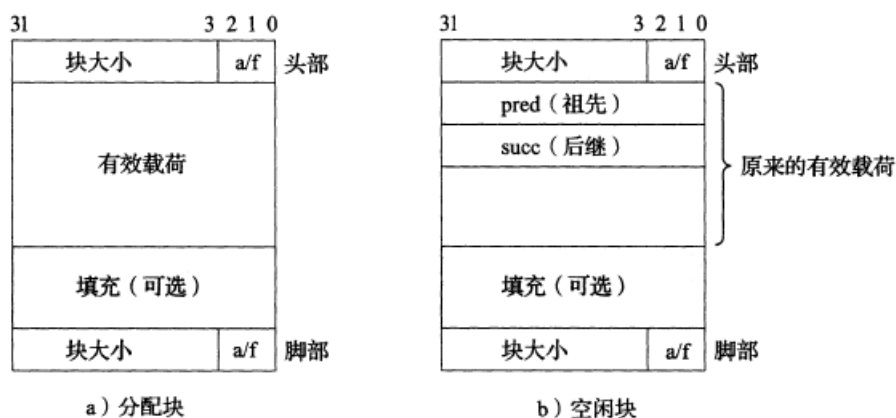


图 9-48 使用双向空闲链表的堆块的格式

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、页式管理,TLB 与四级页表支持下的 VA 到 PA 的变换、三级 cache 支持下物理内存访问,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、动态存储分配管理等内容.

不难看出作为一个程序员,我们需要理解存储器层次结构,因为它对应用程序的性能有着巨大的影响.如果我们的程序需要的数据是存储在 CPU 寄存器中的,那么在指令的执行期间,在 0 个周期内就能访问到它们.如果存储在高速缓存中,需要 4~7 5 个周期.如果存储在主存中,需要上百个周期.而如果存储在磁盘上,需要大约几千万个周期.

熟悉这一章节的内容,可以帮助我们以后编写一些对高速缓存友好的代码,或者说运行速度更快的代码,十分重要.

(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：文件

设备管理：unix io 接口

所有的 I/O 设备（例如网络、磁盘和终端）都被模型化为文件,而所有的输入和输出都被当作对相应文件的读和写来执行.这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,这使得所有的输入和输出都能以一种统一且一致的方式来执行,这就是 Unix I/O 接口.

8.2 简述 Unix IO 接口及其函数

Unix I/O 接口：

1.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.应用程序只需记住这个描述符.

2.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（描述符为 0）、标准输出（描述符为 1）和标准错误（描述符为 2）.头文件<unistd.h> 定义了常量 STDIN_FILENO、STOOUT_FILENO 和 STDERR_FILENO, 它们可用来代替显式的描述符值.

3.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 操作,显式地设置文件的当前位置为 K.

4.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 字节的文件,当 $k\sim m$ 时执行读操作会触发一个称为 end-of-file(EOF) 的条件,应用程序能检测到这个条件.在文件结尾处并没有明确的“EOF 符号”.类似地,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.

5.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.作为响应,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.无论一个进程因为何种原因终止时,内核都会关闭所有打开的文件并释放它们的内存资源.

Unix I/O 函数:

1.进程是通过调用 `open` 函数来打开一个已存在的文件或者创建一个新文件的: `int open(char *filename, int flags, mode_t mode);`

`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.返回: 若成功则为新文件描述符,若出错为-1.

2.进程通过调用 `close` 函数关闭一个打开的文件.`int close(int fd);`

返回: 若成功则为 0, 若出错则为-1.

3.应用程序是通过分别调用 `read` 和 `write` 函数来执行输入和输出的.

`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.

返回: 若成功则为读的字节数,若 EOF 则为 0, 若出错为-1.

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

`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.图 10-3 展示了一个程序使用 `read` 和 `write` 调用一次一个字节地从标准输入复制到标准输出.返回: 若成功则为写的字节数,若出错则为-1.

8.3 printf 的实现分析

研究 `printf` 的实现,首先来看看 `printf` 函数的函数体

```
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;
}
```

先来看 `printf` 函数的内容: 这句:

`va_list arg = (va_list)((char*)&fmt + 4);`

va_list 的定义: typedef char *va_list

这说明它是一个字符指针. 其中的: (char*)(&fmt) + 4) 表示的是...中的第一个参数.

下面我们来看看下一句:

i = vsprintf(buf, fmt, arg);

vsprintf 如下所示

```
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);
}
```

vsprintf 返回的是要打印出来的字符串的长度,vsprintf 的作用就是格式化.它接受确定输出格式的格式字符串 fmt.用格式字符串对个数变化的参数进行格式化,产生格式化输出最后调用 write,

write 函数如下:

write:

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

在 write 函数中,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, Syscall 如下

```
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
```

一个 call save,是为了保存中断前进程的状态.

ecx 中是要打印出的元素个数,ebx 中的是要打印的 buf 字符数组中的第一个元素,这个函数的功能就是不断的打印出字符,直到遇到: '\0' 停止.

就这样,syscall 将字符串中的字节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 (RGB 分量).于是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 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.

8.5 本章小结

本章通过介绍 Linux 的 IO 设备管理方法,简述 Unix IO 接口及其函数,对 `printf` 和 `getchar` 的实现的具体分析,了解了 Unix I/O 在系统中是一个什么样的存在。

(第 8 章 1 分)

结论

用计算机系统的语言,逐条总结 `hello` 所经历的过程.

1. 编写阶段,通过文本编辑器写出 `hello.c`.
2. 预处理阶段,预处理器将 `hello.c` `include` 的外部的库取出合并到 `hello.i` 文件中.
3. 编译阶段,文本文件 `hello.i` 翻译成文本文件 `hello.s`.
4. 汇编,将 `.s` 汇编程序翻译成机器语言指令,`hello.s` 会变成成为可重定位目标文件 `hello.o`.
5. 链接,将 `hello.o` 与可重定位目标文件和动态链接库链接成为可执行目标程序 `hello`.
6. 运行: 在 shell 中输入 `./hello` 并跟上两个参数 `1170300817` `linzhihao`.
7. 运行程序: shell 进程调用 `fork` 创建子进程,shell 调用 `execve`,`execve` 调用启动加载器,加载映射虚拟内存,创建新的内存区域,并创建一组新的代码、数据、堆和栈段. 程序开始运行.
8. 执行指令: CPU 为其分配时间片,在一个时间片中,`hello` 有对 CPU 的控制权,顺序执行自己的代码
10. 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. 动态申请内存: `printf` 会调用 `malloc` 向动态内存分配器申请堆中的内存.
12. 信号: 如果运行途中键入 `ctr-c` 则停止,如果运行途中键入 `ctr-z` 则挂起.
13. 结束: shell 父进程回收子进程.

我的感悟: 深入了解计算机系统是一门很深奥的学科,总感觉里面有很多东西但是却没有足够的时间去挖掘,一方面是因为自己时间安排的不合理,另一方面也和课程安排有一点关系。此次学习过程中一些优化的思路,设计的思想还有基础知识等一系列内容都对以后的职业道路有着巨大的作用,希望自己可以在这门课结束之后自习体会并沉淀一下,也希望自己可以在这条路上走丢更远吧。

(结论 0 分, 缺失 -1 分, 根据内容酌情加分)

附件

文件名	文件作用
hello.i	预处理器修改了的源程序,分析预处理器行为
hello.s	编译器生成的编译程序,分析编译器行为
hello.o	可重定位目标程序,分析汇编器行为
hello	可执行目标程序,分析链接器行为
elf.txt	hello.o 的 elf 格式,分析汇编器和链接器行为
objdump.txt	hello.o 的反汇编,主要是为了分析 hello.o
5.3helloelf.txt	可执行 hello 的 elf 格式,作用是重定位过程分析
5.5helloobjdump.txt	可执行 hello 的反汇编,作用是重定位过程分析

(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] MMU 和 cache 详解 (TLB 机制)
https://blog.csdn.net/qq_21792169/article/details/51303477
- [2] 逻辑地址到线性地址的转换
<https://blog.csdn.net/xuwq2015/article/details/48572421>
- [3] LINUX 逻辑地址、线性地址、物理地址和虚拟地址 转
<https://www.cnblogs.com/zengkefu/p/5452792.html>
- [4] 深入了解计算机系统 (第三版) 2016 Bryant,R.E. 机械工业出版社
- [5] printf 函数实现的深入剖析
https://blog.csdn.net/zhengqijun_/article/details/72454714