

哈爾濱工業大學

计算机系统

大作业

题	目	<u>程序人生-Hello's P2P</u>
专	业	<u>软件工程</u>
学	号	<u>1173710205</u>
班	级	<u>1737102</u>
学	生	<u>郑君烨</u>
指	导	教
师		<u>吴锐</u>

计算机科学与技术学院

2018 年 12 月

摘 要

从最基础的 `hello.c` 程序出发，结合 CSAPP 课程的相关知识，介绍一个程序的生命周期：预处理、编译、汇编、链接等等。通过本次大作业对程序的生命周期有一个更深入的认识和了解，加深课程知识的理解。

关键词：程序生命周期；P2P；CSAPP；020

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	- 5 -
2.4 本章小结	- 6 -
第 3 章 编译	- 7 -
3.1 编译的概念与作用	- 7 -
3.2 在 UBUNTU 下编译的命令	- 7 -
3.3 HELLO 的编译结果解析	- 7 -
3.4 本章小结	- 11 -
第 4 章 汇编	- 12 -
4.1 汇编的概念与作用	- 12 -
4.2 在 UBUNTU 下汇编的命令	- 12 -
4.3 可重定位目标 ELF 格式	- 12 -
4.4 HELLO.O 的结果解析	- 13 -
4.5 本章小结	- 14 -
第 5 章 链接	- 15 -
5.1 链接的概念与作用	- 15 -
5.2 在 UBUNTU 下链接的命令	- 15 -
5.3 可执行目标文件 HELLO 的格式	- 15 -
5.4 HELLO 的虚拟地址空间	- 16 -
5.5 链接的重定位过程分析	- 17 -
5.6 HELLO 的执行流程	- 18 -
5.7 HELLO 的动态链接分析	- 19 -
5.8 本章小结	- 19 -
第 6 章 HELLO 进程管理	- 20 -
6.1 进程的概念与作用	- 20 -

6.2 简述壳 SHELL-BASH 的作用与处理流程.....	- 20 -
6.3 HELLO 的 FORK 进程创建过程	- 20 -
6.4 HELLO 的 EXECVE 过程	- 20 -
6.5 HELLO 的进程执行.....	- 21 -
6.6 HELLO 的异常与信号处理	- 21 -
6.7 本章小结	- 23 -
第 7 章 HELLO 的存储管理.....	- 24 -
7.1 HELLO 的存储器地址空间	- 24 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理	- 24 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理	- 24 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换.....	- 24 -
7.5 三级 CACHE 支持下的物理内存访问	- 25 -
7.6 HELLO 进程 FORK 时的内存映射	- 26 -
7.7 HELLO 进程 EXECVE 时的内存映射	- 26 -
7.8 缺页故障与缺页中断处理.....	- 26 -
7.9 动态存储分配管理	- 26 -
7.10 本章小结	- 27 -
第 8 章 HELLO 的 IO 管理	- 28 -
8.1 LINUX 的 IO 设备管理方法	- 28 -
8.2 简述 UNIX IO 接口及其函数	- 28 -
8.3 PRINTF 的实现分析.....	- 28 -
8.4 GETCHAR 的实现分析.....	- 28 -
8.5 本章小结	- 29 -
结论	- 29 -
附件	- 30 -
参考文献.....	- 31 -

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。

P2P:from program to process.

hello.c 是通过 C 语言这种高级语言进行编写的程序，需要通过预处理、编译、汇编、链接的步骤，一步步由 hello.i、hello.s、hello.o，最后得到一个目标程序的二进制可执行文件。在执行目标文件时，shell 运行并传入命令行参数，shell 使用 fork 函数形成子进程，分配相应的资源以及权限，再用 execve 函数运行程序。

020:from zero to zero

在程序执行完毕后，父进程 shell 会与操作系统一起把进程回收，并释放运行时占用的内存，shell 同时变成 hello.c 执行前的状态。

1.2 环境与工具

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

软件环境：Windows7 64 位以上；VirtualBox/Vmware 11 以上；Ubuntu 16.04 LTS 64 位/优麒麟 64 位；

开发工具：Visual Studio 2010 64 位以上；CodeBlocks；vi/vim/gpedit+gcc

1.3 中间结果

hello.i	hello.c 预处理后的文件
hello.s	汇编文件
hello.o	可重定位目标程序
hello	二进制可执行文件

1.4 本章小结

以 hello.c 为例，讲述了 p2p 与 020 的过程。看似简单的执行其实经过了复杂的步骤，这是 hello.c 的一生，也是大多数程序的一生。

第 2 章 预处理

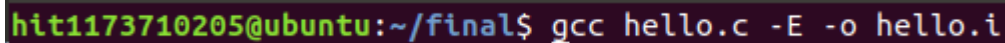
2.1 预处理的概念与作用

在程序进行编译之前，将#开头的标识符文本，也就是“宏名”替换为文本，就 hello.c 而言，`#include <stdio.h>`、`#include <unistd.h>`、`#include <stdlib.h>` 进行替换。从头文件包找到这三个头文件，把设定的标识符替换成中间码。预处理后会形成一个特殊的 c 程序，hello.i 文件。

预处理能够方便编译器编译程序，优化运行花费的时间，并且使代码模块化。

2.2 在 Ubuntu 下预处理的命令

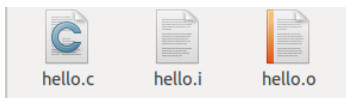
命令行语句：`gcc hello.c -E -o hello.i`

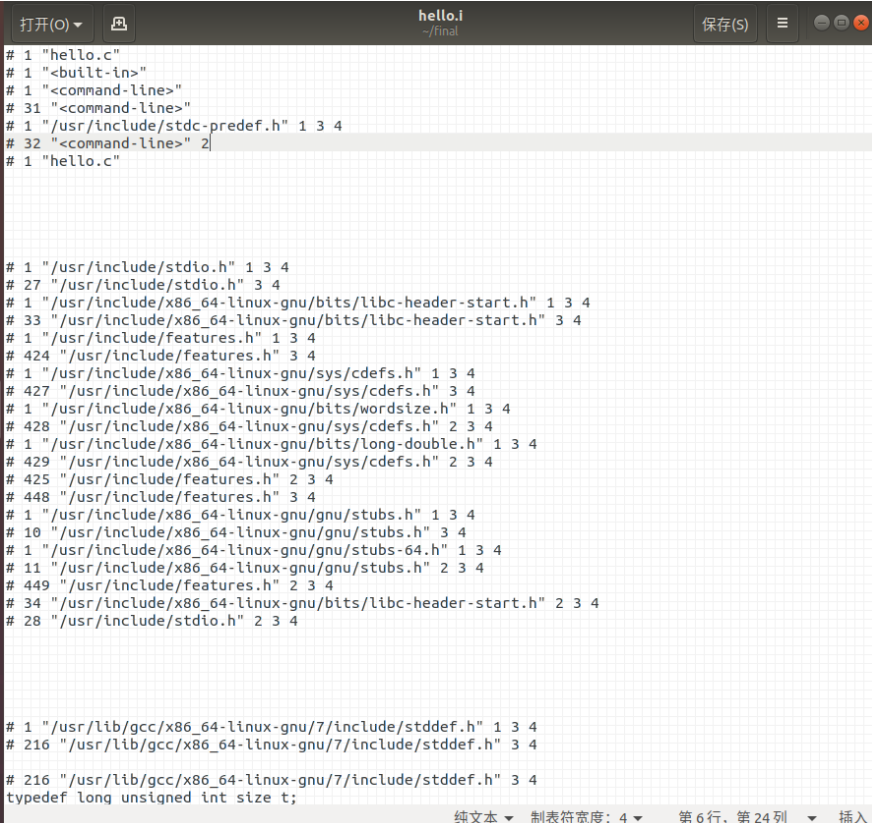


```
hit1173710205@ubuntu:~/final$ gcc hello.c -E -o hello.i
```

2.3 Hello 的预处理结果解析

得到了 hello.i 文件





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

# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
# 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 424 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
# 427 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 428 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/long-double.h" 1 3 4
# 429 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 425 "/usr/include/features.h" 2 3 4
# 448 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 1 3 4
# 10 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs-64.h" 1 3 4
# 11 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 2 3 4
# 449 "/usr/include/features.h" 2 3 4
# 34 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 2 3 4
# 28 "/usr/include/stdio.h" 2 3 4

# 1 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 1 3 4
# 216 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 3 4

# 216 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 3 4
typedef long unsigned int size_t;
```

i 文件足足有 3118 行。

与原来的.c 文件相比，没有了注释部分，但是原来的头文件被宏展开，替换成了头文件的源码。

源码中包含声明函数和定义的变量等内容，程序长度变长很多，不方便阅读。

2.4 本章小结

本章着重介绍了预处理的定义，用处，以及在实际中的应用。预编译是在为接下来的编译做好准备。通过对预处理结果的解析，进一步理解了程序在编译前进行预编译的必要性。

第 3 章 编译

3.1 编译的概念与作用

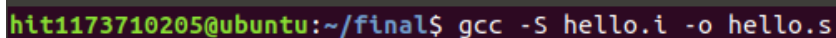
编译器将.i 文件编译成.s 文件的过程称为编译，即预处理后的文件到生成汇编语言程序。

汇编经历的步骤有：词法分析、语法分析、中间代码、代码优化、目标代码生成。

汇编将高级语言汇编成汇编语言，更接近机器语言，是生成二进制可执行文件的中间步骤。

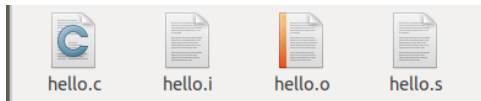
3.2 在 Ubuntu 下编译的命令

命令行：gcc -S hello.i -o hello.s



```
hit1173710205@ubuntu:~/final$ gcc -S hello.i -o hello.s
```

得到了.s 文件



(以下格式自行编排，编辑时删除)

应截图，展示编译过程！

3.3 Hello 的编译结果解析


```

打开(O)  hello.s  保存(S)
~/.final

.file "hello.c"
.text
.globl sleepsecs
.data
.align 4
.type sleepsecs, @object
.size sleepsecs, 4
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
main:
.LFB5:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $32, %rsp
movl %edi, -20(%rbp)
movq %rsi, -32(%rbp)
cmpl $3, -20(%rbp)
je .L2
leaq .LC0(%rip), %rdi
call puts@PLT
movl $1, %edi
call exit@PLT
.L2:
movl $0, -4(%rbp)
jmp .L3
.L4:
movq -32(%rbp), %rax
addq $16, %rax
movq (%rax), %rdx
movq -32(%rbp), %rax
addq $8, %rax

```

3.3.1 数据

字符串 1: “Usage: Hello 学号姓名! \n”

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

学号姓名部分采用 utf-8 编码，一个汉字占用三个字节。

字符串 2: “Hello %s %s\n”

```
14 .string "Hello %s %s\n"
```

声明在.rodata 中 | 10 .section .rodata

int sleepsecs:

```

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

```

sleepsecs 是全局变量，初始化赋值是 2.5，但是作为整型变量，实际的值是 2。

在汇编文本中并没有找到相关的类型转换语句，所以认定是隐式转换。sleepsecs 大小 4 字节，对齐方式 4，设置 long 类型

int i:

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

储存在 栈-4(%rbp)中，占据 4 个字节

3.3.2 赋值

int sleepsecs=2.5

```
8 sleepsecs:
9     .long    2
10    .section .rodata
```

并非将 sleepsecs 声明为 int，而是声明为 long。两种类型在 linux 环境都是 4 字节的，可能是编译器默认一律声明为 long，也可能是在进行隐式类型转换时变成了 long。

i=0

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

对于 4 字节的 int 变量，用 movl 进行赋值

3.3.3 类型转换

```
8 sleepsecs:
9     .long    2
```

程序中存在隐式类型转换，将 2.5 这个浮点数转为了整形变量 2。

3.3.4 算术操作

i++

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

-4(%rbp) 储存着变量 i，使用指令 addl，每次 i 加一

3.3.5 关系操作

i<10

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

使用 cmpl 与 jle，当 i<=9 时进行循环

argc!=3

```
29    cml     $3, -20(%rbp)
30    je     .L2
```

3.3.6 数组/指针/结构操作

```
39    movq    -32(%rbp), %rax
40    addq    $16, %rax
41    movq    (%rax), %rdx
42    movq    -32(%rbp), %rax
43    addq    $8, %rax
44    movq    (%rax), %rax
45    movq    %rax, %rsi
```

arg[]作为一个 char* 的数组，char* 大小为 8 字节。若 arg[0] 为地址，则 arg[1] 为地址+8，arg[2] 为地址+16，以此类推。

3.3.7 控制转移

if-else 和 for 语句都算是控制转移

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

```
54    cml     $9, -4(%rbp)
55    jle     .L4
```

if(argc!=3)

```
29    cml     $3, -20(%rbp)
30    je     .L2
```

3.3.8 函数操作

```
54    cml     $9, -4(%rbp)
55    jle     .L4
56    call    getchar@PLT
57    movl    $0, %eax
58    leave
59    .cfi_def_cfa 7, 8
60    ret
```

返回值一般都储存在 eax 寄存器中，用 movl 传入返回值到 eax，再用 ret 操作返回

用 call 操作调用函数，如果需要参数的话要先将参数传入函数使用的寄存器中：

exit(1)

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

sleep(sleepsecs)
49     movl    sleepsecs(%rip), %eax
50     movl    %eax, %edi
51     call    sleep@PLT

printf("Usage: Hello 学号 姓名! \n")
31     leaq    .LC0(%rip), %rdi
32     call    puts@PLT
```

3.4 本章小结

编译是连接高级语言和机器语言的桥梁，既与机器的操作有相似性，又有一定的可读性。

将 `hello.i` 变为 `hello.s`,就离可以执行的程序更进一步了。通过对比阅读，可以更加深入的了解编译的过程。

(第3章2分)

第 4 章 汇编

4.1 汇编的概念与作用

汇编器将.s 文件翻译成机器语言指令,并打包指令而生成.o 文件的过程成为汇编,.o 文件是一个包含 hello 程序执行机器指令的二进制文件。

只有经过汇编,变为机器能够理解的机器语言,程序才能执行。

4.2 在 Ubuntu 下汇编的命令

命令行: `gcc -c hello.s -o hello.o`

```
hit1173710205@ubuntu:~/final$ gcc -c hello.s -o hello.o
```

4.3 可重定位目标 elf 格式

命令行: `readelf -a hello.o`

文件头信息:

```
hit1173710205@ubuntu:~/final$ readelf -h hello.o
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:             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
```

节头表:

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]		NULL	0000000000000000		00000000	
	0000000000000000	0000000000000000		0	0	0
[1]	.text	PROGBITS	0000000000000000		00000040	
	0000000000000081	0000000000000000	AX	0	0	1
[2]	.rela.text	RELA	0000000000000000		00000338	
	00000000000000c0	0000000000000018	I	10	1	8
[3]	.data	PROGBITS	0000000000000000		000000c4	
	0000000000000004	0000000000000000	WA	0	0	4
[4]	.bss	NOBITS	0000000000000000		000000c8	
	0000000000000000	0000000000000000	WA	0	0	1
[5]	.rodata	PROGBITS	0000000000000000		000000c8	
	000000000000002b	0000000000000000	A	0	0	1
[6]	.comment	PROGBITS	0000000000000000		000000f3	
	0000000000000025	0000000000000001	MS	0	0	1
[7]	.note.GNU-stack	PROGBITS	0000000000000000		00000118	
	0000000000000000	0000000000000000		0	0	1
[8]	.eh_frame	PROGBITS	0000000000000000		00000118	
	0000000000000038	0000000000000000	A	0	0	8
[9]	.rela.eh_frame	RELA	0000000000000000		000003f8	
	0000000000000018	0000000000000018	I	10	8	8
[10]	.symtab	SYMTAB	0000000000000000		00000150	
	0000000000000198	0000000000000018		11	9	8
[11]	.strtab	STRTAB	0000000000000000		000002e8	
	000000000000004d	0000000000000000		0	0	1
[12]	.shstrtab	STRTAB	0000000000000000		00000410	
	0000000000000061	0000000000000000		0	0	1

重定位节:

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
0000000000018	000500000002	R_X86_64_PC32	0000000000000000	.rodata - 4
000000000001d	000c00000004	R_X86_64_PLT32	0000000000000000	puts - 4
0000000000027	000d00000004	R_X86_64_PLT32	0000000000000000	exit - 4
0000000000050	000500000002	R_X86_64_PC32	0000000000000000	.rodata + 1a
000000000005a	000e00000004	R_X86_64_PLT32	0000000000000000	printf - 4
0000000000060	000900000002	R_X86_64_PC32	0000000000000000	sleepsecs - 4
0000000000067	000f00000004	R_X86_64_PLT32	0000000000000000	sleep - 4
0000000000076	001000000004	R_X86_64_PLT32	0000000000000000	getchar - 4

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	000200000002	R_X86_64_PC32	0000000000000000	.text + 0

4.4 Hello.o 的结果解析

objdump -d -r hello.o 分析 hello.o 的反汇编，并请与第 3 章的 hello.s 进行对照分析。

反汇编结果:

```

Disassembly of section .text:
0000000000000000 <main>:
0: 55                push    %rbp
1: 48 89 e5          mov     %rsp,%rbp
4: 48 83 ec 20       sub     $0x20,%rsp
8: 89 7d ec          mov     %edi,-0x14(%rbp)
b: 48 89 75 e0       mov     %rsi,-0x20(%rbp)
f: 83 7d ec 03       cmpl    $0x3,-0x14(%rbp)
13: 74 16             je      2b <main+0x2b>
15: 48 8d 3d 00 00 00 lea     0x0(%rip),%rdi    # 1c <main+0x1c>
18: R_X86_64_PC32    .rodata-0x4
1c: e8 00 00 00 00    callq   21 <main+0x21>
1d: R_X86_64_PLT32    puts-0x4
21: bf 01 00 00 00    mov     $0x1,%edi
26: e8 00 00 00 00    callq   2b <main+0x2b>
27: R_X86_64_PLT32    exit-0x4
2b: c7 45 fc 00 00 00 movl    $0x0,-0x4(%rbp)
32: eb 3b            jmp     6f <main+0x6f>
34: 48 8b 45 e0       mov     -0x20(%rbp),%rax
38: 48 83 c0 10       add     $0x10,%rax
3c: 48 8b 10          mov     (%rax),%rdx
3f: 48 8b 45 e0       mov     -0x20(%rbp),%rax
43: 48 83 c0 08       add     $0x8,%rax
47: 48 8b 00          mov     (%rax),%rax
4a: 48 89 c6          mov     %rax,%rsi
4d: 48 8d 3d 00 00 00 lea     0x0(%rip),%rdi    # 54 <main+0x54>
50: R_X86_64_PC32    .rodata+0x1a
54: b8 00 00 00 00    mov     $0x0,%eax
59: e8 00 00 00 00    callq   5e <main+0x5e>
5a: R_X86_64_PLT32    printf-0x4
5e: 8b 05 00 00 00 00 mov     0x0(%rip),%eax    # 64 <main+0x64>
60: R_X86_64_PC32    sleepsecs-0x4
64: 89 c7            mov     %eax,%edi
66: e8 00 00 00 00    callq   6b <main+0x6b>
67: R_X86_64_PLT32    sleep-0x4
6b: 83 45 fc 01       addl    $0x1,-0x4(%rbp)
6f: 83 7d fc 09       cmpl    $0x9,-0x4(%rbp)
73: 7e bf            jle     34 <main+0x34>
75: e8 00 00 00 00    callq   7a <main+0x7a>
76: R_X86_64_PLT32    getchar-0x4
7a: b8 00 00 00 00    mov     $0x0,%eax
7f: c9              leaveq  %eax
80: c3              retq

```

与 hello.s 的关系：基本上相同但有几处形式上的不同

- 1.反汇编中有.o 文件中机器语言代码及注释，这是原本.s 文件中没有的
2. 操作数与汇编语言的描述有很明显的直接对应，而相对寻址则需要经过处理。
- 3.hello.s 中跳转指令用到了段名称.L1 之类的，但是反汇编之后显示的段名称代表的地址（相对偏移地址）。
- 4.与 3 类似，call 命令在.s 文件中加上的是名称，但在反汇编中是函数所在的地址（相对偏移地址）

4.5 本章小结

只有转换成机器语言，机器才能理解并运行。机器语言对机器效率高，但对人来说无异于天书，使用高级语言大大提高了编写程序的效率。再通过转变为机器语言达到效率最大化。通过反汇编与.s 文件的对比，对汇编语言到机器语言的过程有了更深的理解。

（第 4 章 1 分）

第 5 章 链接

5.1 链接的概念与作用

将在不同的目标文件中调用的各种函数和静态库，动态库链接，目标程序和用于标准库函数的代码进行集中形成可执行文件的过程成为链接。只有经过链接，一个可执行文件才算正式生成。

5.2 在 Ubuntu 下链接的命令

命令行：`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`

```
hit1173710205@ubuntu:~/final$ 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
```

5.3 可执行目标文件 hello 的格式

命令行：`readelf -a hello`

```
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
```


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	000000000400200 A	0	0	1
[2]	.note.ABI-tag 0000000000000020	NOTE 0000000000000000	00000000040021c A	0	0	4
[3]	.hash 0000000000000034	HASH 0000000000000004	000000000400240 A	5	0	8
[4]	.gnu.hash 000000000000001c	GNU_HASH 0000000000000000	000000000400278 A	5	0	8
[5]	.dynsym 00000000000000c0	DYNSYM 0000000000000018	000000000400298 A	6	1	8
[6]	.dynstr 0000000000000057	STRTAB 0000000000000000	000000000400358 A	0	0	1
[7]	.gnu.version 0000000000000010	VERSYM 0000000000000002	0000000004003b0 A	5	0	2
[8]	.gnu.version_r 0000000000000020	VERNEED 0000000000000000	0000000004003c0 A	6	1	8
[9]	.rela.dyn 0000000000000030	RELA 0000000000000018	0000000004003e0 A	5	0	8
	.plt 0000000000000078	RELA 0000000000000018	000000000400410 AI	5	19	8
[11]	.init 0000000000000017	PROGBITS 0000000000000000	000000000400488 AX	0	0	4
[12]	.plt 0000000000000060	PROGBITS 0000000000000010	0000000004004a0 AX	0	0	16
[13]	.text 0000000000000132	PROGBITS 0000000000000000	000000000400500 AX	0	0	16
[14]	.fini 0000000000000009	PROGBITS 0000000000000000	000000000400634 AX	0	0	4
[15]	.rodata 000000000000002f	PROGBITS 0000000000000000	000000000400640 A	0	0	4
[16]	.eh_frame 00000000000000fc	PROGBITS 0000000000000000	000000000400670 A	0	0	8
[17]	.dynamic 00000000000001a0	DYNAMIC 0000000000000010	000000000600e50 WA	6	0	8
[18]	.got 0000000000000010	PROGBITS 0000000000000008	000000000600ff0 WA	0	0	8
[19]	.got.plt 0000000000000040	PROGBITS 0000000000000008	000000000601000 WA	0	0	8
[20]	.data 0000000000000008	PROGBITS 0000000000000000	000000000601040 WA	0	0	4
[21]	.comment 0000000000000024	PROGBITS 0000000000000001	0000000000000000 MS	0	0	1
[22]	.symtab 0000000000000498	SYMTAB 0000000000000018	0000000000000000 23	28	8	
[23]	.strtab 0000000000000150	STRTAB 0000000000000000	0000000000000000 0	0	1	
[24]	.shstrtab 00000000000000c5	STRTAB 0000000000000000	0000000000000000 0	0	1	

5.4 hello 的虚拟地址空间

通过 data dump 查看详细的数据段

Data Dump		
+ 0x0000000000600000-0x0000000000602000		
00000000:00600202	69 62 36 34 2f 6c 64 2d 6c 69 6e 75 78 2d 78 38	lib64/ld-linux-x8
00000000:00600212	36 2d 36 34 2e 73 6f 2e 32 00 04 00 00 00 10 00	6-64.so.2.....
00000000:00600222	00 00 01 00 00 00 47 4e 55 00 00 00 00 00 03 00GNU.....
00000000:00600232	00 00 02 00 00 00 00 00 00 00 00 00 00 00 03 00
00000000:00600242	00 00 08 00 00 00 00 07 00 00 00 06 00 00 00 04 00
00000000:00600252	00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00
00000000:00600262	00 00 02 00 00 00 00 00 00 00 03 00 00 00 05 00
00000000:00600272	00 00 00 00 00 00 01 00 00 00 01 00 00 00 01 00
00000000:00600282	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00600292	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:006002a2	00 00 00 00 00 00 00 00 00 00 00 00 00 10 00
00000000:006002b2	00 00 12 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:006002c2	00 00 00 00 00 00 15 00 00 00 12 00 00 00 00 00

查看 ELF 格式文件中的 Program Headers

```

INTERP      0x0000000000000200 0x00000000000400200 0x00000000000400200
             0x000000000000001c 0x000000000000001c R      0x1
             [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
LOAD        0x0000000000000000 0x00000000000400000 0x00000000000400000
             0x0000000000000076c 0x0000000000000076c R E      0x200000
LOAD        0x00000000000000e50 0x00000000000600e50 0x00000000000600e50
             0x00000000000001f8 0x00000000000001f8 RW      0x200000
DYNAMIC     0x00000000000000e50 0x00000000000600e50 0x00000000000600e50
             0x00000000000001a0 0x00000000000001a0 RW      0x8
NOTE        0x000000000000021c 0x0000000000040021c 0x0000000000040021c
             0x0000000000000020 0x0000000000000020 R      0x4
GNU_STACK   0x0000000000000000 0x0000000000000000 0x0000000000000000
             0x0000000000000000 0x0000000000000000 RW      0x10
GNU_RELRO   0x00000000000000e50 0x00000000000600e50 0x00000000000600e50
             0x00000000000001b0 0x00000000000001b0 R      0x1

```

使用 edb 加载 hello，查看本进程的虚拟地址空间各段信息，并与 5.3 对照分析说明。

5.5 链接的重定位过程分析

分别对 hello 和 hello.o 进行反编译，形成 hello1.s，hello2.s

```

hit1173710205@ubuntu:~/final$ objdump -d -r hello.o > hello1.s
hit1173710205@ubuntu:~/final$ objdump -d -r hello >hello2.s

```

The image shows two disassembly windows from a debugger. The left window, titled 'hello2.s', displays the disassembly of the .init and .plt sections. The right window, titled 'hello1.s', displays the disassembly of the main function. Both windows show assembly instructions with their hex addresses and mnemonics.

有以下的区别:

1.hello1.s 比 hello2.s 有更少的节

2.hello.o 未经过链接, main 的地址从 0 开始, 并且不存在调用的如 printf 这样函数的代码。

3.hello1.s 中使用相对偏移地 hello2.s 中使用虚拟内存地址

4. hello1.s 中跳转以及函数调用的地址在 hello2.s 中是虚拟内存地址。

5.6 hello 的执行流程

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。

执行流程:

```

_dl_start
_dl_init
_start
__libc_start_main @plt
__libc_csu_init
_init
frame_dummy
register_tm_clones

```

```

main
puts@plt
exit@plt
__do_global_ctors_aux
deregister_tm_clones
fini

```

5.7 Hello 的动态链接分析

查询调用 `_dl_start` 之前与之后的全局偏移量 `_GLOBAL_OFFSET_TABLE_`，发现变化

00000000:004006b0	90 01 00 00 10 00 00 00 1c 00 00 00 74 fe ff fft....
00000000:004006c0	02 00 00 00 00 00 00 00 24 00 00 00 30 00 00 00\$.0...
00000000:004006d0	d0 fd ff ff 60 00 00 00 00 0e 10 46 0e 18 4a 0fF..J.
00000000:004006e0	0b 77 08 80 00 3f 1a 3b 2a 33 24 22 00 00 00 00	w...?;*3\$...
00000000:004006f0	1c 00 00 00 58 00 00 00 3a fe ff ff 81 00 00 00X...:....
00000000:00400700	00 41 0e 10 86 02 43 0d 06 02 7c 0c 07 08 00 00	.A...C... ...
00000000:00400710	44 00 00 00 78 00 00 00 a8 fe ff ff 65 00 00 00	D...x...e...
00000000:00400720	00 42 0e 10 8f 02 42 0e 18 8e 03 45 0e 20 8d 04	.B...B...E...
00000000:00400730	42 0e 28 8c 05 48 0e 30 86 06 48 0e 38 83 07 4d	.(...H.0...H.8..M
00000000:00400740	0e 40 72 0e 38 41 0e 30 41 0e 28 42 0e 20 42 0e	..@r.8A.0A.{B. B.
00000000:00400750	18 42 0e 10 42 0e 08 00 10 00 00 00 c0 00 00 00	.B..B.....
00000000:00400760	d0 fe ff ff 02 00 00 00 00 00 00 00 00 00 00 00

5.8 本章小结

通过连接，可以将小部分结合成大的部分，反之，也可以将大项目拆分成小项目。介绍链接的概念和用处，了解了链接的更多细节，比如如何保证寻址正确的进行。在链接之后，终于可以得到二进制执行文件了。

（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：一个具有一定独立功能的程序关于某个数据集合的一次运行活动，是系统进行资源分配和调度运行的基本单位

进程（Process）是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。

6.2 简述壳 Shell-bash 的作用与处理流程

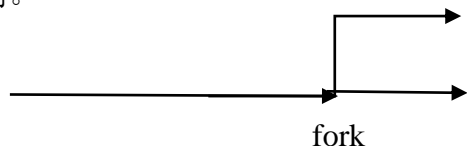
shell 是命令解释器，用于解析和执行命令。它对用户屏蔽了操作系统底层（kernel）的复杂性，是两者间的桥梁。Shell 既是一种命令语言，又是一种程序设计语言。shell 中可以同步和异步的执行命令。shell 提供了少量的内置命令，以便自身功能更加完备和高效。shell 除了执行命令，还提供了变量，流程控制，引用和函数等，类似高级语言一样，能编写功能丰富的程序。

步骤：

1. 读取输入
2. 分析内容，获得参数
3. 内置命令立即执行，非内置调用相应执行
4. 接受键盘输入信号，并进行处理

6.3 Hello 的 fork 进程创建过程

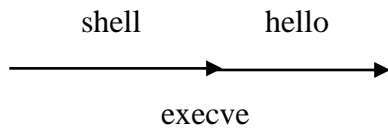
用 fork 函数创建一个与原来进程几乎相同的进程，优先分配内存，复制原有的值，只有少部分不同。hello 进程得到与 Shell 用户级虚拟地址空间相同的（但是独立的）一份副本，包括代码和数据段、堆、共享库以及用户栈。但是它们的 pid 不同。



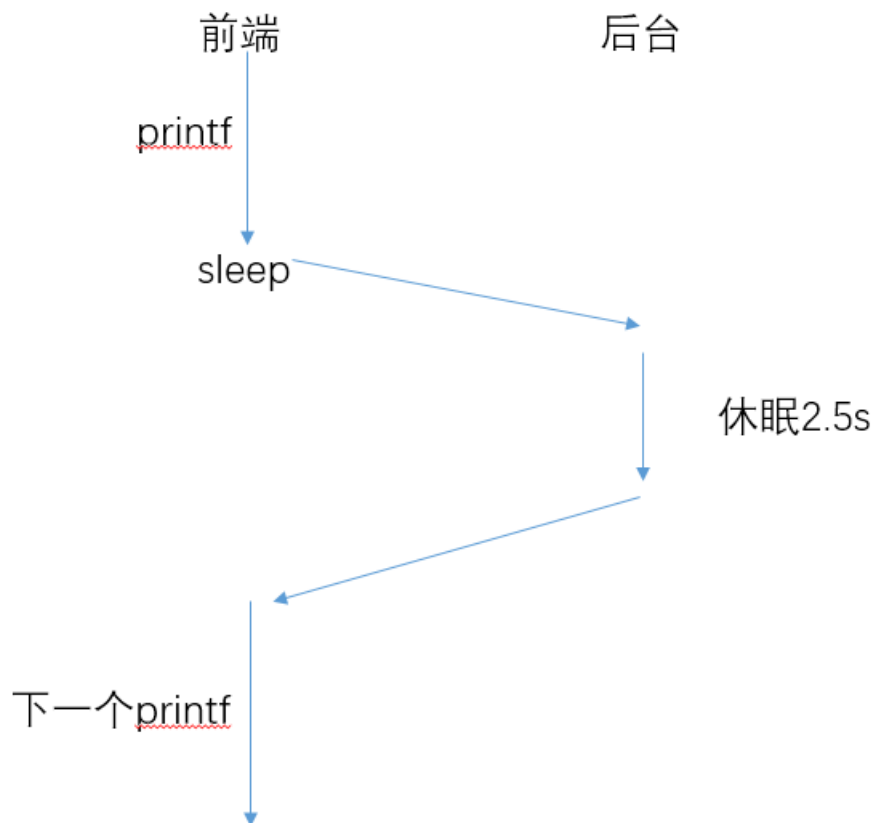
6.4 Hello 的 execve 过程

使用 execve 进行系统调用（传入命令行参数）。fork()返回值是 pid，由于父

子进程 pid 不同，当判断是子进程在当前进程的上下文中加载并运行一个新程序。
execve 调用一次且不返回。



6.5 Hello 的进程执行



6.6 hello 的异常与信号处理

```
hit1173710205@ubuntu:~/final$ ./hello 1173710205 郑君烨
Hello 1173710205 郑君烨
Hello 1173710205 郑君烨
^Z
[1]+  已停止                  ./hello 1173710205 郑君烨
hit1173710205@ubuntu:~/final$ ps
  PID TTY          TIME CMD
  3790 pts/0        00:00:00 bash
  3798 pts/0        00:00:00 hello
  3799 pts/0        00:00:00 ps
```

ctrl+Z: sigtstp 信号，进程暂时挂起

```
hit1173710205@ubuntu:~/final$ fg
./hello 1173710205 郑君烨
Hello 1173710205 郑君烨
Hello 1173710205 郑君烨
Hello 1173710205 郑君烨
Hello 1173710205 郑君烨
Hello 1173710205 郑君烨
Hello 1173710205 郑君烨
```

输入 fg 命令可使进程继续

```
hit1173710205@ubuntu:~/final$ ./hello 1173710205 郑君烨
Hello 1173710205 郑君烨
Hello 1173710205 郑君烨
^C
hit1173710205@ubuntu:~/final$ ps
  PID TTY          TIME CMD
  3810 pts/0        00:00:00 bash
  3819 pts/0        00:00:00 ps
```

ctrl+C: sigint 信号，进程结束


```

hit1173710205@ubuntu:~/final$ ./hello 1173710205 郑君烨
Hello 1173710205 郑君烨
Hello 1173710205 郑君烨
ajdfjladjlk
Hello 1173710205 郑君烨
adkfjaidflaj
Hello 1173710205 郑君烨
adjfalkjdfaif
ajdtfaHello 1173710205 郑君烨
jldj
ahdifalfdj
ajdfiejHello 1173710205 郑君烨
aifjlajfdlia
afiaejiljfaajfHello 1173710205 郑君烨
dfijaleeij
haifjaejHello 1173710205 郑君烨
aljdlaifahfawo'pofhad'
Hello 1173710205 郑君烨
ajaifjapep
a[fo[hate
\aeijaf-Hello 1173710205 郑君烨
39iajfoejapefapwjadfpa
ahit1173710205@ubuntu:~/final$ adkfjaidflaj
jfd
ajfleadkfjaidflaj: 未找到命令
hit1173710205@ubuntu:~/final$ adjfalkjdfaif
adjfalkjdfaif: 未找到命令
hit1173710205@ubuntu:~/final$ ajdifajldj
ajdtfaajldj: 未找到命令
hit1173710205@ubuntu:~/final$ ahdifalfdj
ahdifalfdj: 未找到命令
hit1173710205@ubuntu:~/final$ ajdfiejlaifjlajfdlia
ajdfiejlaifjlajfdlia: 未找到命令
hit1173710205@ubuntu:~/final$ afiaejiljfaajfdijaleeij
afiaejiljfaajfdijaleeij: 未找到命令
hit1173710205@ubuntu:~/final$ haifjaejaljdlaifahfawo'pofhad'
haifjaejaljdlaifahfawopofhad: 未找到命令
hit1173710205@ubuntu:~/final$ ajaifjapep
ajaifjapep: 未找到命令
hit1173710205@ubuntu:~/final$ a[fo[hate
> \aeijaf39iajfoejapefapwjadfpa
> ajfd
> ajfie

```

胡乱输入

6.7 本章小结

shell 是系统和用户之间的桥梁，通过 hello 进程的运行，了解了 shell 的操作流程和基本原理，对于 fork 和 execve 函数的运行流程有了进一步的了解。同样接触了 hello 的进程执行和异常信号处理流程。

(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(logical address): 程序代码经过编译后出现在汇编程序中地址, 机器语言指令中, 用来指定一个操作数或者是一条指令的地址。

线性地址(linear address): 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。

线性地址也叫虚拟地址(virtual address)

物理地址(physical address): 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。

7.2 Intel 逻辑地址到线性地址的变换-段式管理

由于线性地址是描述符+偏移量的组合形式, 将逻辑地址分为段选择符+段描述符的判别符(TI)+地址偏移量的形式。

- 1.判断 TI 字段, 判断是局部段描述符(ldt)还是全局段描述符(gdt)
- 2.将其组合成段描述符+地址偏移量的形式。

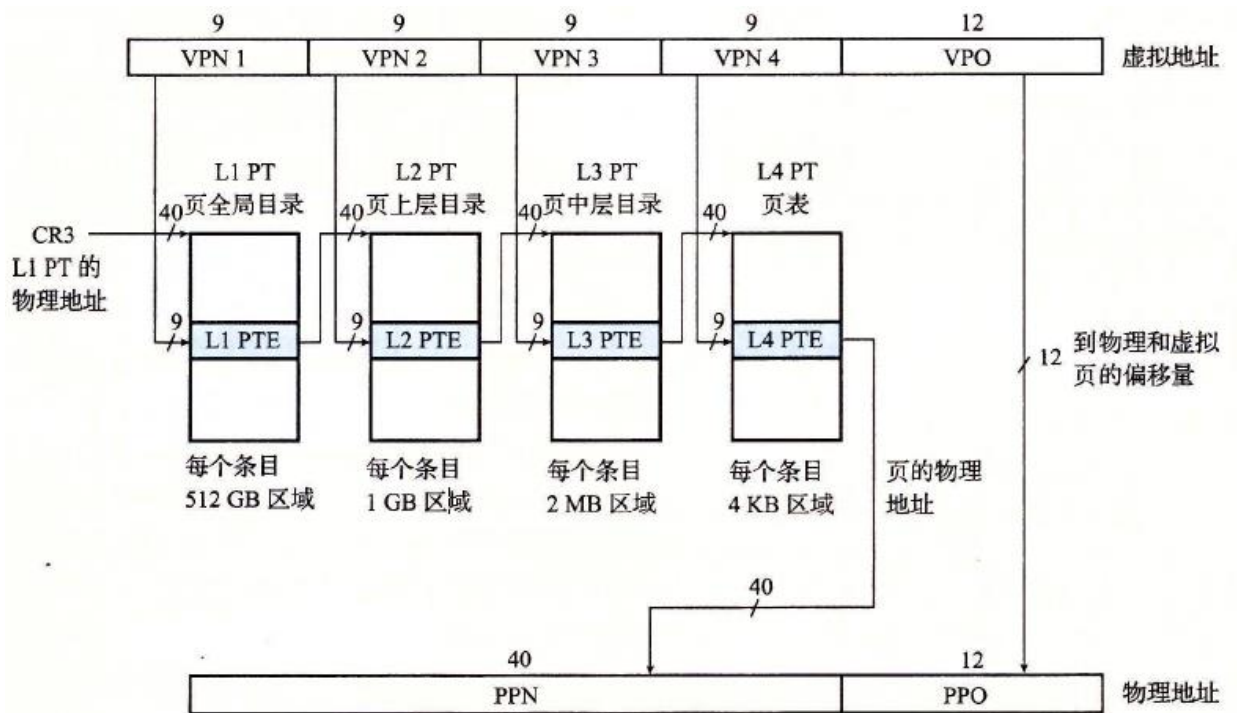
7.3 Hello 的线性地址到物理地址的变换-页式管理

页式管理由 CPU 的页式内存管理单元, 负责把一个线性地址, 最终翻译为一个物理地址。

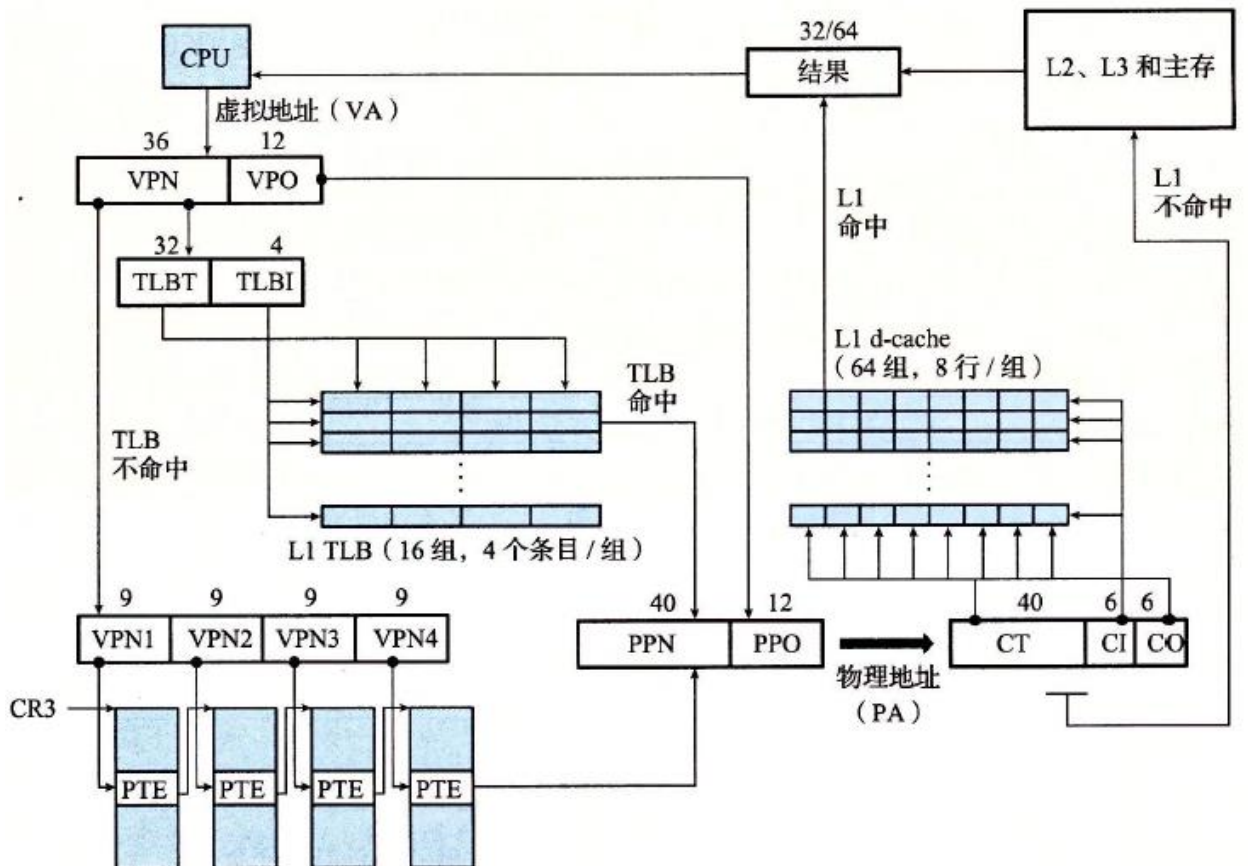
页(page): 线性地址被分为以固定长度为单位的组

1. 将线性地址分为 VPN (虚拟页号)+VPO (虚拟页偏移) 的形式。
2. 将 VPN 拆分成 TLBT (TLB 标记)+TLBI (TLB 索引)
3. 去 TLB 缓存里找所对应的 PPN (物理页号) 如果发生缺页情况则直接查找对应的 PPN,
4. PPN+VPO 就是生成的物理地址

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换



7.5 三级 Cache 支持下的物理内存访问



7.6 hello 进程 fork 时的内存映射

fork 为创建数据结构并且分配唯一的 pid。创建 mm_struct（内存描述符），vm_area_struct（区域结构描述符）的副本，将这两个页面标记为只读，两个进程的每个 vm_area_struct 都标记为私有，这样就只能在写入时复制。

7.7 hello 进程 execve 时的内存映射

- 1.删除已存在的用户区域。删除 shell 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。
- 3.映射共享区域。
- 4.设置程序计数器(PC)，指向代码入口点。

7.8 缺页故障与缺页中断处理

缺页故障分为以下三种情况：

1. 段错误：判断虚拟地址合法性，并遍历所有的合法区域结构。
2. 非法访问：查看地址权限，对进程的权限进行修改
3. 正常缺页：选择一个牺牲页，然后将目标页加载到物理内存中，再使导致缺页的指令重新启动，页面命中。

缺页中断： 选择一个牺牲页面，如果这个牺牲页面被修改过，那么就将它交换出去，换入新的页面并更新页表

7.9 动态存储分配管理

分配器将堆视为一组不同大小的块(block) 的集合来维护。每个块就是一个连续的虚拟内存片(chunk),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：

显示分配器：要求应用显示的释放已经分配的块，像 C, C++语言采用的机制。

隐式分配器：分配器检测一个已经分配的块何时不再被程序所使用时，自动 free 掉。这个就是垃圾收集。

7.10 本章小结

本章从 hello 的储存管理出发，重点讲述了段式管理、页式管理、以 intel Core7 VA 到 PA 的变换、物理内存访问的原理。对 fork, execve 内存映射函数的原理进行了一定程度的了解。

(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

在设备模型中，所有的设备都通过总线相连。所有的 I/O 设备（例如网络、磁盘和终端）都被模型化为文件，即每一个设备都是一个文件。

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，而文件的相关信息由内核记录，应用程序只需要记录这个描述符。

8.2 简述 Unix IO 接口及其函数

unix io 接口是指所有的输入和输出都被当做对相应文件的读和写来执行。

打开文件：int open();

关闭文件：int close();

读文件：ssize_t read();

写文件：ssize_t write();

8.3 printf 的实现分析

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

*fmt 是格式化用到的字符串

调用 va_start 函数，取到 fmt 中的第一个参数的地址

调用 vsprint 函数，将所有参数格式化后存入 buf，返回格式化的数组长度。

从 vsprintf 生成显示信息，到 write 系统函数，到陷阱-系统调用 int 0x80 或 syscall.

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

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

8.4 getchar 的实现分析

调用 `read` 函数，将所有的缓冲区读到 `buf` 中，返回缓冲区长度，然后返回 `buf` 前面的元素，只有 `buf` 为空时才会调用 `read` 函数。

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

了解 `unix I/O` 的定义和原理，加深对硬件设备管理的方法，整理了打开，关闭读，写文件的接口和函数。对 `printf` 和 `getchar` 函数的原理进行了一番探讨。

(第 8 章 1 分)

结论

`hello.c` 的一生结束了，正如它自白那样，没有留下什么，也没有带走什么。但是在看似平淡无奇的一生中，却离不开那么多相关者的帮助，并不波澜万丈，却也奥秘无尽。

在预处理→编译→汇编→链接中 `hello.c` 不断成长，最后能够独当一面。

`fork` 和 `execve` 是 `shell` 的恩惠，让它平安无事

未曾相见，`Unix I/O`，虚拟内存映射，存储管理默默奉献自己，履行使命

生命的最后，`hello.c` 迎来了 `shell` 与内核对它仁慈的终结，等待下一次的轮回

大作业回顾了 `hello.c` 的一生，同时也是对一个学期 `CSAPP` 课程的总结，最最简单的 `hello.c` 文件都要无数精密的机制进行协调，一同运作。对于课程又有了新的认识与了解。

(结论 0 分，缺少 -1 分，根据内容酌情加分)

附件

文件名	用处
hello.i	hello.c 预处理后的文件
hello.s	hello.i 编译成的汇编语言文件
hello.o	hello.s 生成的二进制文件
hello1.s	hello.o 反编译后的汇编语言文件
hello2.s	hello 反编译的汇编语言文件
hello.elf	hello 的 elf 表

(附件 0 分, 缺失 -1 分)

参考文献

为完成本次大作业你翻阅的书籍与网站等

- [1] 林来兴. 空间控制技术[M]. 北京: 中国宇航出版社, 1992: 25-42.
- [2] 辛希孟. 信息技术与信息服务国际研讨会论文集: A 集[C]. 北京: 中国科学出版社, 1999.
- [3] 赵耀东. 新时代的工业工程师[M/OL]. 台北: 天下文化出版社, 1998 [1998-09-26]. <http://www.ie.nthu.edu.tw/info/ie.newie.htm> (Big5) .
- [4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨: 哈尔滨工业大学, 1992: 8-13.
- [5] KANAMORI H. Shaking Without Quaking[J]. Science, 1998, 279 (5359) : 2063-2064.
- [6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science , 1998 , 281 : 331-332[1998-09-23]. <http://www.sciencemag.org/cgi/collection/anatmorp>.

(参考文献 0 分, 确实 -1 分)