

哈爾濱工業大學

計算機系統

大作業

題 目	<u>程序人生-Hello's P2P</u>
專 業	<u>計算機科學與技術</u>
學 號	<u>1170300924</u>
班 級	<u>1736101</u>
學 生	<u>王梓舟</u>
指 導 教 師	<u>劉宏偉</u>

計算機科學與技術學院

2018 年 12 月

摘 要

本次大作业将 `hello.c` 从源代码到运行出来的每个过程进行了深刻的剖析，本次实验就是对程序运行的解读，也是对计算机系统课程的圆满句号。大作业将会运用计算机系统课程的各方面知识，分八章对代码的各阶段进行阐释

关键词：CSAPP，HIT，大作业

（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	- 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	- 10 -
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	- 14 -
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	- 17 -
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	- 21 -
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.....	- 23 -
6.5 HELLO 的进程执行.....	- 23 -
6.6 HELLO 的异常与信号处理.....	- 24 -
6.7 本章小结.....	- 24 -
第 7 章 HELLO 的存储管理.....	- 25 -
7.1 HELLO 的存储器地址空间.....	- 25 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理.....	- 25 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理.....	- 25 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换.....	- 26 -
7.5 三级 CACHE 支持下的物理内存访问.....	错误！未定义书签。
7.6 HELLO 进程 FORK 时的内存映射.....	- 27 -
7.7 HELLO 进程 EXECVE 时的内存映射.....	- 27 -
7.8 缺页故障与缺页中断处理.....	- 27 -
7.9 动态存储分配管理.....	- 28 -
7.10 本章小结.....	- 28 -
第 8 章 HELLO 的 IO 管理.....	- 30 -
8.1 LINUX 的 IO 设备管理方法.....	- 30 -
8.2 简述 UNIX IO 接口及其函数.....	- 30 -
8.3 PRINTF 的实现分析.....	- 30 -
8.4 GETCHAR 的实现分析.....	- 30 -
8.5 本章小结.....	- 32 -
结论.....	- 33 -
附件.....	- 34 -
参考文献.....	- 35 -

第 1 章 概述

1.1 Hello 简介

通过文本输入得到 `Hello.c`，为源程序 对 `Hello.c` 这一源程序依次进行预处理，编译，汇编，链接的操作之后，生成可执行程序 `Hello`。在 `shell` 中使其运行，`fork` 产生子进程，则 `Hello` 从 `program` 成为 `Process`，这个过程即为 `P2P`。紧接着进行 `execve`，通过依次进行对虚拟内存的映射，物理内存的载入，进入主函数执行代码。`hello` 调用 `write` 等系统函数在屏幕打印信息，之后 `shell` 父进程 `bash` 回收 `Hello` 的进程，一切相关的数据结构被删除。以上即为 `020` 的全部过程。

1.2 环境与工具

硬件：CPU Intel Core i3 3110m 500GB HDD 4G RAM

软件环境 Microsoft Windows7 专业版 64 位；VMware Workstation 14 Pro；Ubuntu 16.04

开发工具 Visual Studio 2017 64 位；CodeBlocks 64 位；`readelf`；`vim`；`gcc`；`Hexedit`；`edb`

1.3 中间结果

<code>hello.c</code>	大作业的 <code>hello.c</code> 源程序
<code>hello.i</code>	经过预处理后的中间文件
<code>hello.s</code>	经过编译后的汇编文件
<code>hello.o</code>	经过汇编后的可重定位目标执行文件
<code>hello</code>	经过链接后的可执行程序
<code>Asm1.txt</code>	<code>hello</code> 反汇编生成的代码
<code>hello.elf</code>	<code>hello</code> 的 ELF
<code>Asm2.txt</code>	<code>hello.o</code> 反汇编生成的代码
<code>helloo.elf</code>	<code>hello.o</code> 的 ELF

1.4 本章小结

1. 介绍了 Hello 的 P2P 与 O2O 过程
2. 列出了大作业的各种环境及中间各结果

(第 1 章 0.5 分)

第 2 章 预处理

2.1 预处理的概念与作用

程序设计领域中，预处理一般是指在程序源代码被翻译为目标代码的过程中，生成二进制代码之前的过程。典型地，由预处理器(preprocessor) 对程序源代码文本进行处理，得到的结果再由编译器核心进一步编译。这个过程并不对程序的源代码进行解析，但它把源代码分割或处理成为特定的单位——（用 C/C++的术语来说是）预处理记号(preprocessing token)用来支持语言特性（如 C/C++的宏调用）。

最常见的预处理是 C 语言和 C++语言。ISO C 和 ISO C++都规定程序由源代码被翻译分为若干有序的阶段(phase) [1] [2]，通常前几个阶段由预处理器实现。预处理中会展开以#起始的行，试图解释为预处理指令(preprocessing directive)，其中 ISO C/C++要求支持的包括#if/#ifdef/#ifndef/#else/#elif/#endif（条件编译）、#define（宏定义）、#include（源文件包含）、#line（行控制）、#error（错误指令）、#pragma（和实现相关的杂注）以及单独的#（空指令）[1] [2]。预处理指令一般被用来使源代码在不同的执行环境中被方便的修改或者编译。[3]

预处理器在 UNIX 传统中通常缩写为 PP，在自动构建脚本中 C 预处理器被缩写为 CPP 的宏指代。为了不造成歧义，C++(cee-plus-plus) 经常并不是缩写为 CPP，而改成 CXX。

2.2 在 Ubuntu 下预处理的命令

```
cpp hello.c > hello.i
```

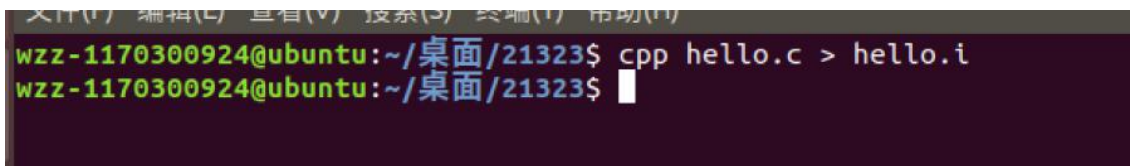


图 2.1 在 Ubuntu 下预处理的命令

2.3 Hello 的预处理结果解析

```

wzz-1170300924@ubuntu: ~/桌面/21323
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
# 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
"hello.i" [只读] 3118L, 66102C
1,1 顶端

```

图 2.2 Hello 的预处理结果解析

如图文件有 3118 行，其中主函数只是如下图的短短几行，而函数的其他部分都是引用自如上图的各种头文件

```

int main(int argc, char *argv[])
{
    int i;

    if(argc!=3)
    {
        printf("Usage: Hello 学号 姓名! \n");
        exit(1);
    }
    for(i=0; i<10; i++)
    {
        printf("Hello %s %s\n", argv[1], argv[2]);
        sleep(sleepsecs);
    }
    getchar();
    return 0;
}
3118,1 底端

```

图 2.3 Hello 的预处理结果解析

2.4 本章小结

hello.c 在编译之前先经过预处理生成 hello.i, 在源代码的基础上插入了在头文件处引入的代码, 以便于接下来的编译工作。

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, 即将经过预处理的 源代码转换为汇编代码

作用: 将更偏向于人的高级语言翻译为更偏向于机器的汇编语言, 以便于之后机器的执行

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
```

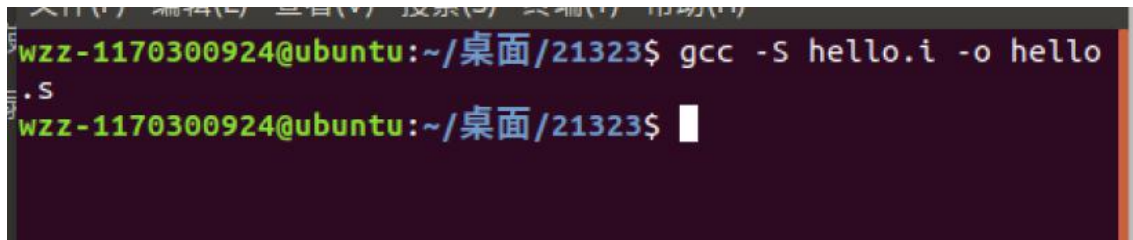


图 3.1 在 Ubuntu 下编译的命令

3.3 Hello 的编译结果解析

(以下格式自行编排, 编辑时删除)

此部分是重点, 说明编译器是怎么处理 C 语言的各个数据类型以及各类操作的。应分 3.3.1~3.3.x 等按照类型和操作进行分析, 只要 hello.s 中出现的属于大作业 PPT 中 P4 给出的参考 C 数据与操作, 都应解析。

file 源文件

.data 数据段
 .globl 全局标识符
 .string 字符串类型
 .string 字符串类型
 .long long 类型
 .text 代码段

3.3.1 Sleepsecs 是全局变量，长度为四字节，初值为 2

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

图 3.2 sleepsecs

3.3.2 %rbp 为指向数组的指针，通过地址依次减八来改变数组。

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

图 3.3 %rbp

3.3.3 输出的 printf 函数中的字符串

```

.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:

```

图 3.4 printf

3.3.4 用 movl 完成将 i 赋值为 0

```

.L2:
    movl    $0, -4(%rbp)
    jmp     .L3
.L4:

```

图 3.5 movl 的赋值

3.3.5 条件判断转移：通过比较 cmpl 与跳转 je, jle 来实现 argc!=3 和 i

```

movq    %rsi, -32(%rbp)
cmpl    $3, -20(%rbp)
je       .L2
leaq    .LC0(%rip), %rdi

```

图 3.6 判断与转移

3.3.6 通过比较和加法完成 for 循环

先通过赋值给 i 一个初值 0，然后通过条件控制转移，当 $i < 10$ 时反复执行，当 $i = 10$ 则跳出，执行后面的语句

```

.L2:
    movl    $0, -4(%rbp)
    jmp     .L3
.L4:
    movq    -32(%rbp), %rax
    addq    $16, %rax
    movq    (%rax), %rdx
    movq    -32(%rbp), %rax
    addq    $8, %rax
    movq    (%rax), %rax
    movq    %rax, %rsi
    leaq    .LC1(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    movl    sleepsecs(%rip), %eax
    movl    %eax, %edi
    call    sleep@PLT
    addl    $1, -4(%rbp)
.L3:
    cmpl    $9, -4(%rbp)
    jle     .L4
    call    getchar@PLT
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc

```

图 3.7 for 循环

3.3.7 调用函数设置参数并调用了 printf, getchar 和 sleep

```

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

```

图 3.8 调用函数

3.4 本章小结

详细分析了 hello.c 翻译成汇编语言时里面各变量各条件控制以及各函数调用。

(第 3 章 2 分)

第 4 章 汇编

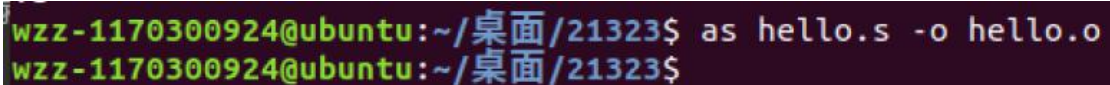
4.1 汇编的概念与作用

概念：汇编器（as）将汇编语言 `hello.s` 翻译成机器语言，打包生成可重定位目标程序的形式，储存结果生成 `hello.o`（为二进制文件）

作用：将偏向于机器的汇编语言翻译成完全可以由机器执行的二进制命令，以便于之后的链接

4.2 在 Ubuntu 下汇编的命令

`as hello.s -o hello.o`



```
wzz-1170300924@ubuntu:~/桌面/21323$ as hello.s -o hello.o
wzz-1170300924@ubuntu:~/桌面/21323$
```

图 4.1 在 Ubuntu 下汇编的命令

4.3 可重定位目标 elf 格式

Magic: Magic 为 16 字节序列，包含了该文件的大小与字节顺序

其余部分为各种信息，如 ELF 头的大小，目标文件类型，小端法还是大端法，文件类型，机器类型，节头部表的文件偏移条目的大小和数量等

ELF 头:

```

Magic:      7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
类别:                               ELF64
数据:                               2 补码, 小端序 (little endian)
版本:                               1 (current)
OS/ABI:                               UNIX - System V
ABI 版本:                               0
类型:                               REL (可重定位文件)
系统架构:                               Advanced Micro Devices X86-64
版本:                               0x1
入口点地址:                               0x0
程序头起点:                               0 (bytes into file)
Start of section headers:               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

```

图 4.2, 4.3 Magic

节头部表: 用来描述各个节的大小, 类型, 位置等各种信息

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]	.text	PROGBITS	0000000000000000	00000040
	0000000000000081	0000000000000000	AX 0 0	1
[2]	.rela.text	RELA	0000000000000000	00000340
	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
	000000000000002b	0000000000000001	MS 0 0	1
[7]	.note.GNU-stack	PROGBITS	0000000000000000	0000011e
	0000000000000000	0000000000000000	0 0	1
[8]	.eh_frame	PROGBITS	0000000000000000	00000120
	0000000000000038	0000000000000000	A 0 0	8
[9]	.rela.eh_frame	RELA	0000000000000000	00000400
	0000000000000018	0000000000000018	I 10 8	8
[10]	.symtab	SYMTAB	0000000000000000	00000158
	0000000000000198	0000000000000018	11 9	8

[11]	.strtab	STRTAB	0000000000000000	000002f0
	000000000000004d	0000000000000000	0	1
[12]	.shstrtab	STRTAB	0000000000000000	00000418
	0000000000000061	0000000000000000	0	1

图 4.4, 4.5 节头部表

重定位节:

描述了.text 中的各个重定位信息, 每个信息包含: 名称; 8 字节的计算辅助信息; 类型; 8 字节的信息 Info (由 4 字节的偏移量 symbol 和 4 字节的类型 type 组成); 8 字节的偏移位置 offset

下图为 hello.c 中各函数与字符串的重定位信息

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

重定位节 '.rela.eh_frame' at offset 0x400 contains 1 entry:

偏移量	信息	类型	符号值	符号名称 + 加数
000000000020	000200000002 R_X86_64_PC32		0000000000000000	.text + 0

图 4.6 重定位节

4.4 Hello.o 的结果解析

```
wzz-1170300924@ubuntu:~/桌面/21323$ objdump -d -r hello.o >asm.txt
```

```
hello.o:      文件格式 elf64-x86-64

Disassembly of section .text:

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



```

27: R_X86_64_PLT32      exit-0x4
2b:  c7 45 fc 00 00 00 00    movl    $0x0,-0x4(%rbp)
32:  eb 3b                    jmp     6f <main+0x6f>
34:  48 8b 45 e0              mov     -0x20(%rbp),%rax
38:  48 83 c0 10              add     $0x10,%rax
3c:  48 8b 10                  mov     (%rax),%rdx
3f:  48 8b 45 e0              mov     -0x20(%rbp),%rax
43:  48 83 c0 08              add     $0x8,%rax
47:  48 8b 00                  mov     (%rax),%rax
4a:  48 89 c6                  mov     %rax,%rsi
4d:  48 8d 3d 00 00 00 00    lea     0x0(%rip),%rdi      # 54 <main+0x54>
50: R_X86_64_PC32      .rodata+0x1a
54:  b8 00 00 00 00 00      mov     $0x0,%eax
59:  e8 00 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 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 00      callq   7a <main+0x7a>
76: R_X86_64_PLT32      getchar-0x4
7a:  b8 00 00 00 00 00      mov     $0x0,%eax
7f:  c9                      leaveq   %eax
80:  c3                      retq

```

纯文本 ▾ 制表符宽度: 8 ▾ 第 1 行, 第 1 列 ▾ 插入

图 4.7, 4.8 hello.o 结果

与 hello.s 大体相同，区别如下：

1. 跳转：

在跳转时使用了确定的地址，而非段名称跳转表

```

jmp     6f <main+0x6f>

```

图 4.9 跳转

2. 调用函数：

在 callq 时采用了 call offset 的地址方式，而非 hello.s 的 call symbol

```

callq   6b <main+0x6b>

```

3. 全局变量：

与 2 类似，使用全局变量时采用了 offset(%rip)的形式，而非 hello.s 使用的 symbol(%rip)

```

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

```

图 4.10 调用函数

三个函数：


```

27: R_X86_64_PLT32 exit-0x4
2b:  c7 45 fc 00 00 00 00    movl    $0x0, -0x4(%rbp)
32:  eb 3b                    jmp     6f <main+0x6f>
34:  48 8b 45 e0              mov     -0x20(%rbp),%rax
38:  48 83 c0 10              add     $0x10,%rax
3c:  48 8b 10                  mov     (%rax),%rdx
3f:  48 8b 45 e0              mov     -0x20(%rbp),%rax
43:  48 83 c0 08              add     $0x8,%rax
47:  48 8b 00                  mov     (%rax),%rax
4a:  48 89 c6                  mov     %rax,%rsi
4d:  48 8d 3d 00 00 00 00    lea     0x0(%rip),%rdi      # 54 <main+0x54>
50: R_X86_64_PC32 .rodata+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

```

图 4.11 三个函数

4.5 本章小结

深入分析了 hello.o 的 ELF, 看到了各表的相关信息, 比较了.o 的反汇编与.s 的异同, 研究了各个函数与全局变量的重定位

(第 4 章 1 分)

第 5 章 链接

5.1 链接的概念与作用

概念：

链接是将各种代码和数据片段收集并组合成一个单一文件的过程，这个文件可以被加载（复制）到内存执行。链接可以执行于编译时，也可以执行于加载时，甚至执行于运行时。

作用：

链接器使得分离编译成为可能。使得代码真正成为可以单独运行的可执行文件。

5.2 在 Ubuntu 下链接的命令



```
wzz-1170300924@ubuntu:~/桌面/21323$ 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
wzz-1170300924@ubuntu:~/桌面/21323$ ./hello
```

图 5.1 链接命令

```
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
```

5.3 可执行目标文件 hello 的格式

ELF 头:

```

Magic:      7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
类别:                               ELF64
数据:                               2 补码, 小端序 (little endian)
版本:                               1 (current)
OS/ABI:                               UNIX - System V
ABI 版本:                               0
类型:                               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:               5928 (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	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 5 0	8
[4]	.gnu.hash	GNU_HASH	0000000000400278	00000278
	000000000000001c	0000000000000000	A 5 0	8
[5]	.dynsym	DYNSYM	0000000000400298	00000298
	00000000000000c0	0000000000000018	A 6 1	8
[6]	.dynstr	STRTAB	0000000000400358	00000358
	0000000000000057	0000000000000000	A 0 0	1
[7]	.gnu.version	VERSYM	00000000004003b0	000003b0
	0000000000000010	0000000000000002	A 5 0	2
[8]	.gnu.version_r	VERNEED	00000000004003c0	000003c0

[9]	.rela.dyn	RELA	00000000004003e0	000003e0
[10]	.rela.plt	RELA	0000000000400410	00000410
[11]	.init	PROGBITS	0000000000400488	00000488
[12]	.plt	PROGBITS	00000000004004a0	000004a0
[13]	.text	PROGBITS	0000000000400500	00000500
[14]	.fini	PROGBITS	0000000000400634	00000634
[15]	.rodata	PROGBITS	0000000000400640	00000640
[16]	.eh_frame	PROGBITS	0000000000400670	00000670
[17]	.dynamic	DYNAMIC	0000000000600e50	00000e50
[18]	.got	PROGBITS	0000000000600ff0	00000ff0
[19]	.got.plt	PROGBITS	0000000000601000	00001000
[20]	.data	PROGBITS	0000000000601040	00001040
[21]	.comment	PROGBITS	0000000000000000	00001048
[22]	.symtab	SYMTAB	0000000000000000	00001078
[23]	.strtab	STRTAB	0000000000000000	00001510
[24]	.shstrtab	STRTAB	0000000000000000	00001660

重定位节 '.rela.dyn' at offset 0x3e0 contains 2 entries:

偏移量	信息	类型	符号值	符号名称 + 加数
000000600ff0	000300000006	R_X86_64_GLOB_DAT	0000000000000000	__libc_start_main@GLIBC_2.2.5 + 0
000000600ff8	000500000006	R_X86_64_GLOB_DAT	0000000000000000	__gmon_start__ + 0

重定位节 '.rela.plt' at offset 0x410 contains 5 entries:

偏移量	信息	类型	符号值	符号名称 + 加数
000000601018	000100000007	R_X86_64_JUMP_SLO	0000000000000000	puts@GLIBC_2.2.5 + 0
000000601020	000200000007	R_X86_64_JUMP_SLO	0000000000000000	printf@GLIBC_2.2.5 + 0
000000601028	000400000007	R_X86_64_JUMP_SLO	0000000000000000	getchar@GLIBC_2.2.5 + 0
000000601030	000600000007	R_X86_64_JUMP_SLO	0000000000000000	exit@GLIBC_2.2.5 + 0
000000601038	000700000007	R_X86_64_JUMP_SLO	0000000000000000	sleep@GLIBC_2.2.5 + 0

图 5.2,5.3, 5.4, 5.5,5.6 elf 头

5.4 hello 的虚拟地址空间

使用 edb 加载 hello，查看本进程的虚拟地址空间。根据 5.3 节的信息，可以找到各节的二进制信息。代码段的信息如下所示。代码段开始于 0x400550 处，大小为 0x01f2。

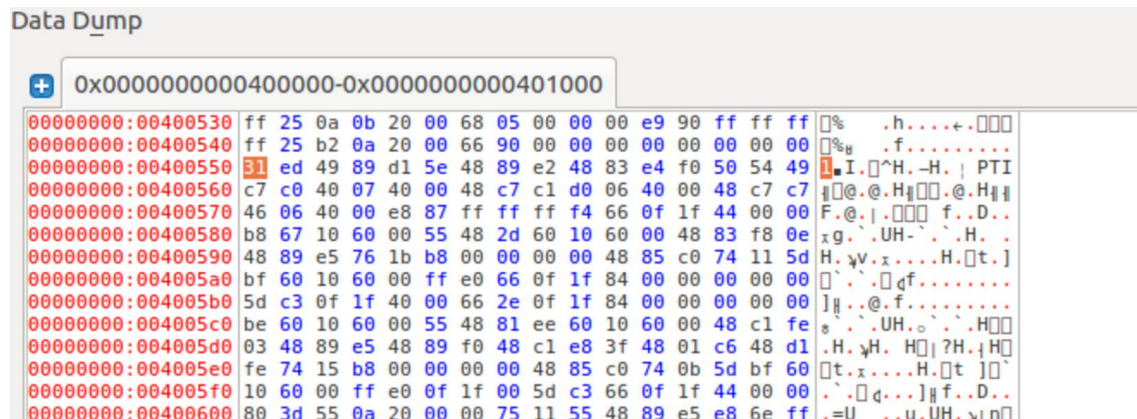


图 5.7 hello 的虚拟地址空间

5.5 链接的重定位过程分析

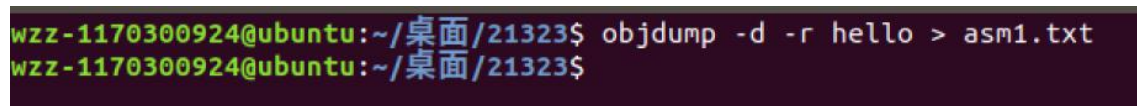


图 5.8 链接的重定位过程分析

1.增加了 plt 表

Disassembly of section .plt:

```

00000000004004a0 <.plt>:
  4004a0:    ff 35 62 0b 20 00    pushq  0x200b62(%rip)    # 601008
<_GLOBAL_OFFSET_TABLE_+0x8>
  4004a6:    ff 25 64 0b 20 00    jmpq   *0x200b64(%rip)  # 601010
<_GLOBAL_OFFSET_TABLE_+0x10>
  4004ac:    0f 1f 40 00          nopl   0x0(%rax)

00000000004004b0 <puts@plt>:
  4004b0:    ff 25 62 0b 20 00    jmpq   *0x200b62(%rip)  # 601018
<puts@GLIBC_2.2.5>
  4004b6:    68 00 00 00 00      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      pushq  $0x1
  4004cb:    e9 d0 ff ff ff      jmpq   4004a0 <.plt>

2. 函数调用：直接跳转到下一条指令的地址

00000000004004d0 <getchar@plt>:
  4004d0:    ff 25 52 0b 20 00    jmpq   *0x200b52(%rip)  # 601028
<getchar@GLIBC_2.2.5>
  4004d6:    68 02 00 00 00      pushq  $0x2
  4004db:    e9 c0 ff ff ff      jmpq   4004a0 <.plt>

```

图 5.9 plt 表

5.6 hello 的执行流程

```

ld-2.23.so!_dl_start
ld-2.23.so!_dl_init
Start          0x400500
Init           0x400488
Main           0x400532
Exit
libc-2.23.so!exit
。

```

5.7 Hello 的动态链接分析

动态链接的基本思想是把程序按照模块拆分成各个相对独立部分，在程序运行时才将它们链接在一起形成一个完整的程序，而不是像静态链接一样把所有程序模块都链接成一个单独的可执行文件。虽然动态链接把链接过程推迟到了程序运行时，但是在形成可执行文件时（注意形成可执行文件和执行程序是两个概念），还是需要用到动态链接库。比如我们在形成可执行程序时，发现引用了一个外部的函数，此时会检查动态链接库，发现这个函数名是一个动态链接符号，此时可执行程序就不对这个符号进行重定位，而把这个过程留到装载时再进行。

5.8 本章小结

研究了 hello 的链接全过程，深刻分析了虚拟地址，重定位，执行流程与动态链接。

(以下格式自行编排，编辑时删除)

(第 5 章 1 分)

第 6 章 hello 进程管理

6.1 进程的概念与作用

概念：进程是一个具有独立功能的程序关于某个数据集合的一次运行活动。它可以申请和拥有系统资源，是一个动态的概念，是一个活动的实体。它不只是程序的代码，还包括当前的活动，通过程序计数器的值和处理寄存器的内容来表示。

作用：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。

6.2 简述壳 Shell-bash 的作用与处理流程

作用：shell 作为命令语言，它交互式解释和执行用户输入的命令或者自动地解释和执行预先设定好的一连串的命令；作为程序设计语言，它定义了各种变量和参数，并提供了许多在高级语言中才具有的控制结构，包括循环和分支。

处理流程：

- 1.读取命令，分析参数
- 2.如果是内部命令则直接执行
- 3.否则 fork 一个子进程执行
- 4.执行完毕后回收该进程

6.3 Hello 的 fork 进程创建过程

运行命令./hello 1170300924 WangZizhou

分析命令，发现./hello 并不是内置命令，1170300924 WangZizhou 为参数之后 fork 子进程，与它父进程虚拟地址相同，拥有不同的 PID

6.4 Hello 的 execve 过程

子进程创建完成后，execve 开始加载运行 hello, 通过栈等将参数（1170300924 WangZizhou）传递给 main(), 开始运行

6.5 Hello 的进程执行

逻辑控制流：即为一系列程序计数器 PC 的值序列进程是轮流使用处理器的，

在同一个处理器核心中，每个进程执行它的流的一部分后被抢占（暂时挂起），然后轮到其他进程。时间片：进程执行它的控制流的部分时间段。用户模式和内核模式：用户模式即为未设置模式位，不允许执行特权指令直接引用地址空间中内核区内的代码和数据；内核模式为设置模式位时，该进程执行所有命令访问所有数据。上下文信息：上下文就是内核重启被抢占的进程所需要的状态，它由通用寄存器等各种内核数据构成。分别三个步骤：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。结合进程上下文信息、进程时间片，阐述进程调度的过程，用户态与核心态转换等等。

6.6 hello 的异常与信号处理

故障：缺页异常，hello 进程的页表被映射到 hello 文件，然而实际代码拷贝至内存仍未完成，在执行到相应地址的代码时会引发缺页异常。

终止：不可恢复错误发生。

中断：接受到键盘键入的信号，如 ctrl Z ,ctrl C 等

```
wzz-1170300924@ubuntu:~/桌面/21323$ ./hello 1170300924 WangZizhou
Hello 1170300924 WangZizhou
Hello 1170300924 WangZizhou
Hello 1170300924 WangZizhou
Hello 1170300924 WangZizhou
Hello 1170300924 WangZizhou
Hello 1170300924 WangZizhou
^Z
[1]+ 已停止                  ./hello 1170300924 WangZizhou
wzz-1170300924@ubuntu:~/桌面/21323$
```

图 6.1 hello 的异常与信号处理

6.7 本章小结

本章阐述了 hello 由 Program 变身为 Process 后在 shell 中被 fork、execve 的过程，以及 hello 开始运行后的上下文切换、异常与信号处理。

（第 6 章 1 分）

第 7 章 hello 的存储管理

7.1 hello 的存储器地址空间

逻辑地址：逻辑地址 (LogicalAddress) 是指由程序产生的与段相关的偏移地址部分。

物理地址：要经过寻址方式的计算或变换才得到内存存储器中的实际有效地址，即物理地址。

线性地址：线性地址 (Linear Address) 是逻辑地址到物理地址变换之间的中间层。在分段

部件中逻辑地址是段中的偏移地址，然后加上基地址就是线性地址。

虚拟地址：这里的虚拟地址就是线性地址

7.2 Intel 逻辑地址到线性地址的变换-段式管理

索引号，或者直接理解成数组下标——那它总要对一个数组吧，它又是什么东东的索引呢？这个东东就是“段描述符(segment descriptor)”，呵呵，段描述符具体地址描述了一个段（对于“段”这个字眼的理解，我是把它想像成，拿了一把刀，把虚拟内存，砍成若干的截——段）。这样，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，我刚才对段的抽象不太准确，因为看看描述符里面究竟有什么东东——也就是它究竟是如何描述的，就理解段究竟有什么东东了，每一个段描述符由 8 个字节组成，这些东东很复杂，虽然可以利用一个数据结构来定义它，不过，我这里只关心一样，就是 Base 字段，它描述了一个段的开始位置的线性地址。

Intel 设计的本意是，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。那究竟什么时候该用 GDT，什么时候该用 LDT 呢？这是由段选择符中的 T1 字段表示的，=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。

首先，给定一个完整的逻辑地址[段选择符：段内偏移地址：

1、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。我们就有了一个数组了。

2、拿出段选择符中前 13 位，可以在这个数组中，查找到对应的段描述符，这样，

它了 Base，即基地址就知道了。

3、把 Base + offset，就是要转换的线性地址了。

7.3 Hello 的线性地址到物理地址的变换-页式管理

这种页式管理方式中，第一级的页表称之为“页目录”，用于存放页表的基地址；第二级才是真正的“页表”用于存放物理内存中页框的基地址。

1、二级页目录的页式内存管理方式中，第一级的页目录的基址存放在 CPU 寄存器 CR3 中，这也是转换的开始点；

2、每一个活动的进程，都有其对应的独立虚拟内存（页目录也是唯一的），那么它对应一个独立的页目录地址。一运行一个进程，需要将它的页目录地址放到

CR3 寄存器中，将别的页目录的基址暂时换到内存中；

3、每个 32 位的线性地址被划分成三部分，页目录索引（10 位），页表索引（10 位），偏移量（12 位）。

线性地址转换成物理地址的过程如下：

1、从 CR3 中取出进程的页目录的地址（操作系统在负责进程的调度的时候，将这个地址装入对应的 CR3 地址寄存器），取出其前 20 位，这是页目录的基地址；

2、根据取出来的页目录的基地址以及线性地址的前十位，进行组合得到线性地址的前十位的索引对应的项在页目录中地址，根据该地址可以取到该地址上的值，该值就是二级页表项的基址；当然你说地址是 32 位，这里只有 30 位，其实当取出线性地址的前十位之后还会该该前十位左移 2 位，也就是乘以 4，一共 32 位；

之所以这么做是因为每个地址都是 4B 的大小，因此其地址肯定是 4 字节对齐的，因此左移两位之后的 32 位的值恰好就是该前十位的索引项的所对应值的起始地址，

只要从该地址开始向后读四个字节就得到了该十位数字对应的页目录中的项的地址，取该地址的值就是对应的页表项的基址；

3、根据第二步取到的页表项的基址，取其前 20 位，将线性地址的 10-19 位左移 2 位（原因和第 2 步相同），按照和第 2 步相同的方式进行组合就可以得到

线性地址对应的物理页框在内存中的地址在二级页表中的地址的起始地址，根据该地址向后读四个字节就得到了线性地址对应的物理页框在内存中的地址在二级页表中的地址，然后取该地址上的值就得到线性地址对应的物理页框在内存中的基地址；（这一步的地址比较绕，还请仔细琢磨，反复推敲）

4、根据第 3 步取到的基地址，取其前 20 位得到物理页框在内存中的基址，再根据线性地址最后的 12 位的偏移量得到具体的物理地址，取该地址上的值就是最后要得到值；7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

虚拟地址被分成 VPN 和 VPO 两部分，VPN 被分为 TLBT 和 TLBI 用于在 TLB 中查询。根据 TLBI 确定 TLB 中的组索引，TLBT 判断 PPN 是否已被缓存到 TLB 中，若 TLB 命中，返回 PPN，否则会到页表中查询 PPN。在页表中查询 PPN 时，VPN 会被分为四个部分，分级页表的索引，前三级页表的查询结果均为下一级页表的基地址，第四级页表的查询结果为 PPN。将查询到的 PPN 与 VPO 组合，得到物理地址。

7.5 三级 Cache 支持下的物理内存访问

- 1.发送物理地址给一级 cache
- 2.分别解析出 CO,CT 等信息，并根据 CT 判断是否命中
- 3.若命中则读取数据否则提交给二级三级 cache 重复上述步骤
- 4.若均为命中进入主存中查找

7.6 hello 进程 fork 时的内存映射

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。

7.7 hello 进程 execve 时的内存映射

- 1.删除已有的区域 即删除已经存在的区域结构
- 2.映射私有区域 即为新程序的代码、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。例如 hello 中.data 区以及.text 区
- 3.映射共享区域 即映射动态链接，共享对象及其相关到虚拟地址的共享区域内例如 hello 中的库 libc.so

7.8 缺页故障与缺页中断处理

缺页故障是一种常见的故障，当指令引用一个虚拟地址，在 MMU 中查找页表时发现与该地址相对应的物理地址不在内存中，因此必须从磁盘中取出的时候就会发生故障。其处理流程遵循图 7.10 所示的故障处理流程。

缺页中断处理：缺页处理程序是系统内核中的代码，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。当缺页处理程序返回时，CPU 重新启动引起缺页的指令，这条指令再次发送 VA 到 MMU，这次 MMU 就能正常翻译 VA 了。

7.9 动态存储分配管理

动态内存分配器维护进程的虚拟内存，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，可以是已分配的、空闲的。空闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，直到它被释放。

分配器分为两种：显式分配器、隐式分配器。

显式分配器：要求应用显式地释放任何已分配的块。

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

隐式：分配器通过遍历堆中所有的块，从而间接地遍历整个空闲块的集合。

显式：分配块的组织形式与隐式基本一致，而空闲块组织成链表形式的数据结构。每个空闲块中，都包含一个 pred（前驱）和 succ（后继）指针。优点是首次适配的分配时间从块总数的线性时间减少到了空闲块数量的线性时间。

维护链表的顺序有：

1. 后进先出（LIFO）：将新释放的块放置在链表的开始出，释放一个块可以在线性的时间内完成，如果使用了边界标记，那额合并也可以在常熟时间内完成。

2. 按照地址顺序：

其中链表中的每个块的地址都小于它的后继的地址，在这种情况下，释放一个块需要线性时间的搜索来定位合适的前驱。优点在于，按照地址排序的首次适配比 LIFO 排序的首次适配有着更高的内存利用率，接近最佳适配的利用率。（因为这样连接空闲链表，使小的碎片更容易产生在前面）

7.10 本章小结

具体研究了存储器地址空间 逻辑地址到线性地址的变换-段式管理 线性地址到物理地址的变换-页式管理 TLB 与四级页表支持下的 VA 到 PA 的变换三级 Cache 支持下的物理内存访问 hello 进程 fork 时的内存映射 进程 fork 时的内存映射 进程 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。

(以下格式自行编排, 编辑时删除)

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

8.2 简述 Unix IO 接口及其函数

UNIX IO: 1. 打开文件

2. 标准输入输出

3. 关闭文件 对应函数：

1. `int open(char* filename, int flags, mode_t mode)` `char* filename` 文件名
(包含位置) `int flags` 打开方式 `mode_t mode` 权限

2. `ssize_t read(int fd, void *buf, size_t n)` `ssize_t write(int fd, const void *buf, size_t n)` `void *buf, size_t n` 位置信息

3. `int close(fd)`

8.3 printf 的实现分析

代码：

```
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': //以 x 为例子代替 d c 等
            itoa(tmp, *((int*)
                strcpy(p, tmp);

                p_next_arg += 4;
                p += strlen(tmp);

                break;

            case 's':
                break;

            default:
                break;
    }
}

return (p - buf);
```



```
}

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;
}
```

分析：printf 首先确定 arg 这一格式化参数，然后调用 vsprintf 函数判断%标志符号

（如%d），其中无关符号自动略过，返回需要输出的结果串及长度，最后通过系统函数 write 输出长度 i 的串 buf 到屏幕上

8.4 getchar 的实现分析

当用户按键时，键盘接口会得到一个代表该按键的键盘扫描码，同时产生一个中断请求，中断请求抢占当前进程运行键盘中断子程序，键盘中断子程序先从键盘接口取得该按键的扫描码，然后将该按键扫描码转换成 ASCII 码，保存到系统的键盘缓冲区之中

getchar 调用了 read 函数，read 函数也通过 sys_call 调用内核中的系统函数，将读取存储在键盘缓冲区中的 ASCII 码，直到读到回车符，然后返回整个字符串，getchar 函数只从中读取第一个字符，其他的字符被缓存在输入缓冲区。

8.5 本章小结

本章介绍了 Linux 中 I/O 设备的管理方法，Unix I/O 接口和函数，并且分析了 printf 和 getchar 函数是如何通过 Unix I/O 函数实现其功能的。

（第 8 章 1 分）

结论

Hello 的一生：

1. 用户输入源代码文本 `hello.c`
2. 预处理得到文本 `hello.i`
3. 编译得到汇编代码 `hello.s`
4. 汇编得到可重定位目标文件 `hello.o`
5. 链接生成可执行文件 `hello`
6. 在 `shell` 中运行 `hello`
7. Fork 子进程，然后调用 `execve`，载入内存，进入主函数
8. Cpu 分配时间片，执行指令
9. 通过地址访问内存读取数据
10. 调用 `printf` 和 `getchar` 函数输出，读入
11. 接受键盘信号，如 `ctrl z`
12. 结束运行，父进程回收

`hello.c` 的一生表面上短暂，其中复杂的人生经历却是十分坎坷的，值得每个程序员细细品味，深入理解计算机系统，我们也会对 `hello.c` 有一个崭新的认识，计算机的魅力也正在于此。

（结论 0 分，缺失 -1 分，根据内容酌情加分）

附件

列出所有的中间产物的文件名，并予以说明起作用。

文件的名字	文件的作用
hello.c	大作业的 hello.c 源程序
hello.i	经过预处理后的中间文件
hello.s	经过编译后的汇编文件
hello.o	经过汇编后的可重定位目标执行文件
hello	经过链接后的可执行程序
Asm1.txt	hello 反汇编生成的代码
hello.elf	hello 的 ELF
Asm2.txt	hello.o 反汇编生成的代码
helloo.elf	hello.o 的 ELF

(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] 湛颖. 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>.
- [7] <https://blog.csdn.net/gdj0001/article/details/80135196> 逻辑地址、线性地址和物理地址之间的转换
- [8] <https://blog.csdn.net/a7980718/article/details/80895302> linux 内核缺页中断处理
- [9] <https://blog.csdn.net/haiross/article/details/50995750> 我理解的逻辑地址、线性地址、物理地址和虚拟地址

(参考文献 0 分, 缺失 -1 分)