

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>计算机类</u>
学	号	<u>1170300424</u>
班	级	<u>1736101</u>
学	生	<u>谢卓芯</u>
指	导	教
师		<u>刘宏伟</u>

计算机科学与技术学院

2018 年 12 月

摘 要

作为每一个程序员接触到的第一个程序,hello world 对于我们来说并不陌生。本文则是利用我们对于计算机系统这门课程的学习,基于 linux 系统对于 hello.c 程序从代码到运行到最终实现并被回收的一个过程的分析。向大家展示了一个 c 语言程序的生命周期所经历的过程。通过学习这些内容,也使我们整体梳理了计算机系统的知识,加深了我们对于《深入理解计算机系统》这本书的理解。

关键词: 计算机系统; 编译; 汇编; 程序; 进程; hello; 内存; 编译系统

(摘要 0 分, 缺失-1 分, 根据内容精彩程度酌情加分 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.....	- 6 -
2.1 预处理的概念与作用.....	- 6 -
2.2 在 UBUNTU 下预处理的命令.....	- 6 -
2.3 HELLO 的预处理结果解析.....	- 7 -
2.4 本章小结.....	- 7 -
第 3 章 编译.....	- 8 -
3.1 编译的概念与作用.....	- 8 -
3.2 在 UBUNTU 下编译的命令.....	- 9 -
3.3 HELLO 的编译结果解析.....	- 9 -
3.4 本章小结.....	- 13 -
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.....	- 14 -
4.4 HELLO.O 的结果解析.....	- 16 -
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.....	- 20 -
5.5 链接的重定位过程分析.....	- 21 -
5.6 HELLO 的执行流程.....	- 23 -
5.7 HELLO 的动态链接分析.....	- 23 -
5.8 本章小结.....	- 23 -
第 6 章 HELLO 进程管理.....	- 25 -
6.1 进程的概念与作用.....	- 25 -

6.2 简述壳 SHELL-BASH 的作用与处理流程.....	- 25 -
6.3 HELLO 的 FORK 进程创建过程.....	- 25 -
6.4 HELLO 的 EXECVE 过程.....	- 25 -
6.5 HELLO 的进程执行.....	- 26 -
6.6 HELLO 的异常与信号处理.....	- 27 -
6.7 本章小结.....	- 28 -
第 7 章 HELLO 的存储管理.....	- 29 -
7.1 HELLO 的存储器地址空间.....	- 29 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理.....	- 29 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理.....	- 29 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换.....	- 30 -
7.5 三级 CACHE 支持下的物理内存访问.....	- 30 -
7.6 HELLO 进程 FORK 时的内存映射.....	- 31 -
7.7 HELLO 进程 EXECVE 时的内存映射.....	- 31 -
7.8 缺页故障与缺页中断处理.....	- 31 -
7.9 动态存储分配管理.....	- 32 -
7.10 本章小结.....	- 33 -
第 8 章 HELLO 的 IO 管理.....	- 36 -
8.1 LINUX 的 IO 设备管理方法.....	- 36 -
8.2 简述 UNIX IO 接口及其函数.....	- 36 -
8.3 PRINTF 的实现分析.....	- 36 -
8.4 GETCHAR 的实现分析.....	- 37 -
8.5 本章小结.....	- 40 -
结论.....	- 41 -
附件.....	- 42 -
参考文献.....	- 43 -

第 1 章 概述

1.1 Hello 简介

Hello 的 P2P 过程:

hello 程序的生命周期从一个高级 C 语言程序开始。之后其中的每条 C 语句都必须被其他程序转化为一组低级机器语言指令。然后这些指令按照可执行目标程序（目标程序也称可执行目标文件）格式打包，并以二进制磁盘文件的形式存放起来。

从源文件到目标文件的转化是编译器驱动程序完成的。这一翻译过程分为四个阶段完成：预处理阶段、编译阶段、汇编阶段和链接阶段。

最后，要运行这个翻译完成被存在磁盘上的可执行目标文件，需要将 hello 这一应用名输入到 shell 这一程序中，然后 shell 会加载并且运行 hello 程序，然后等待程序终止。Hello 程序在屏幕上输出它的消息，然后终止。

Hello 的 020 过程:

Shell 开始运行 hello 程序后，为其映射虚拟内存，然后在调用程序入口时载入物理内存，进入 main 函数，执行目标代码，然后分配时间片。hello 程序运行结束后，父进程回收该子进程，内核删除进程产生的相关数据和分配的结构，恢复程序执行前的状态。

1.2 环境与工具

硬件环境:

Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz 2.90 GHz

软件环境:

Win10 64 位 ; VMware ; Ubuntu 64 位

开发与调试工具:

codeblocks,objdump,gdb,edb 等

1.3 中间结果

hello.i: 预处理的结果。

hello.s: hello.i 编译后的结果。

hello.o: hello.s 汇编后的结果，可重定位目标程序。

hello.out: hello.o 链接后生成的可执行目标文件。

1.4 本章小结

本章主要介绍了 hello 程序的 P2P 以及 O2O 的过程，进行 处理 hello 程序的软硬件环境以及调试开发工具；除此之外还列出了处理 hello 程序的中间结果。这一章的内容也为接下来的叙述做出了铺垫。

(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,修改原始的 C 程序。比如 hello.c 中第 1 行的#include <stdio.h>命令告诉预处理器读取系统头文件 stdio.h 的内容,并把它直接插入程序文本中。结果就得到了另一个 C 程序,通常是以.i 作为文件扩展名。

作用：

预处理中会展开以#起始的行,试图解释为预处理指令,其中 ISO C/C++要求的包括#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)。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。

2.2 在 Ubuntu 下预处理的命令

指令: gcc -E hello.c -o hello.i

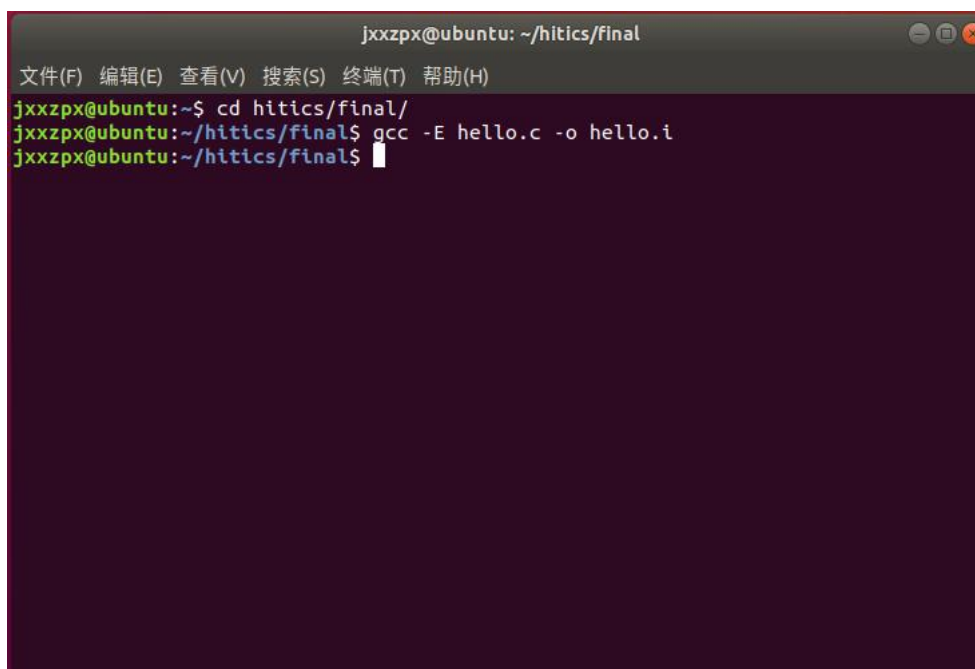
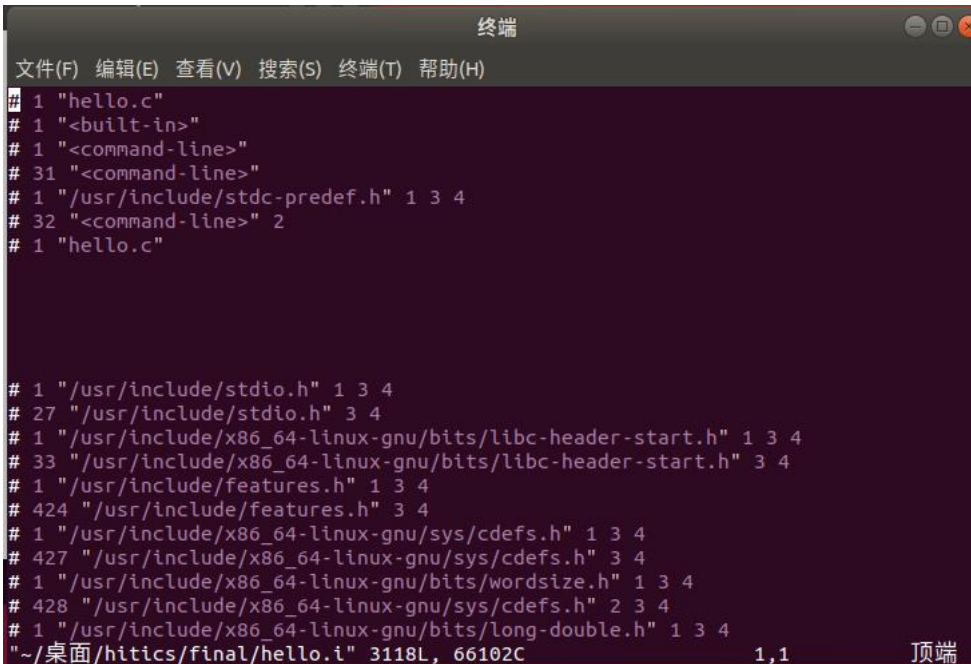
A screenshot of a terminal window on an Ubuntu system. The window title is 'jxxzpx@ubuntu: ~/hitics/final'. The terminal shows three lines of commands and their outputs: 1. 'jxxzpx@ubuntu:~\$ cd hitics/final/' followed by a new prompt. 2. 'jxxzpx@ubuntu:~/hitics/final\$ gcc -E hello.c -o hello.i' followed by a new prompt. 3. 'jxxzpx@ubuntu:~/hitics/final\$' followed by a cursor. The terminal has a dark background with light-colored text. The window has standard Ubuntu window controls (minimize, maximize, close) in the top right corner.

图 2-1

2.3 Hello 的预处理结果解析



```
终端
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
"~/桌面/hitcs/final/hello.i" 3118L, 66102C 1,1 顶端
```

图 2-2

图 3-2 展示了部分预处理后的代码，可以看出预处理后，引入了头文件，替换了所有的宏，去除了注释等。

2.4 本章小结

本章主要讲了预处理的概念、作用。并且展示了预处理的指令和处理结果。预处理这一步为接下来的编译做出了准备，使其能够正确进行。

(第 2 章 0.5 分)

第 3 章 编译

3.1 编译的概念与作用

编译的概念：

编译是指编译器将文本文件 `hello.i` 翻译成文本文件 `hello.s` 的过程，这个文本文件内包含了一个汇编语言程序。实现了经过词法分析、语法分析、语义检查等过程，在检查无错误后将代码翻译成汇编语言。得到的汇编语言代码可供编译器进行生成机器代码、链接等操作。

编译的作用：

1.词法分析：

对由字符组成的单词进行处理，从左至右逐个字符地对源程序进行扫描，产生一个个的单词符号，把作为字符串的源程序改造成为单词符号串的中间程序。

2.语法分析：

以单词符号作为输入，分析单词符号串是否形成符合语法规则的语法单位，如表达式、赋值、循环等，最后看是否构成一个符合要求的程序，按该语言使用的语法规则分析检查每条语句是否有正确的逻辑结构，程序是最终的一个语法单位。

3.中间代码生成：

中间代码是源程序的一种内部表示，或称中间语言。中间代码的作用是可使编译程序的结构在逻辑上更为简单明确，特别是可使目标代码的优化比较容易实现中间代码，即为中间语言程序。

4.代码优化：

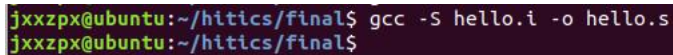
代码优化是指对程序进行多种等价变换，使得从变换后的程序出发，能生成更有效的目标代码。

5.目标代码生成：

目标代码生成是编译的最后一个阶段。目标代码生成器把语法分析后或优化后的中间代码变换成目标代码。

3.2 在 Ubuntu 下编译的命令

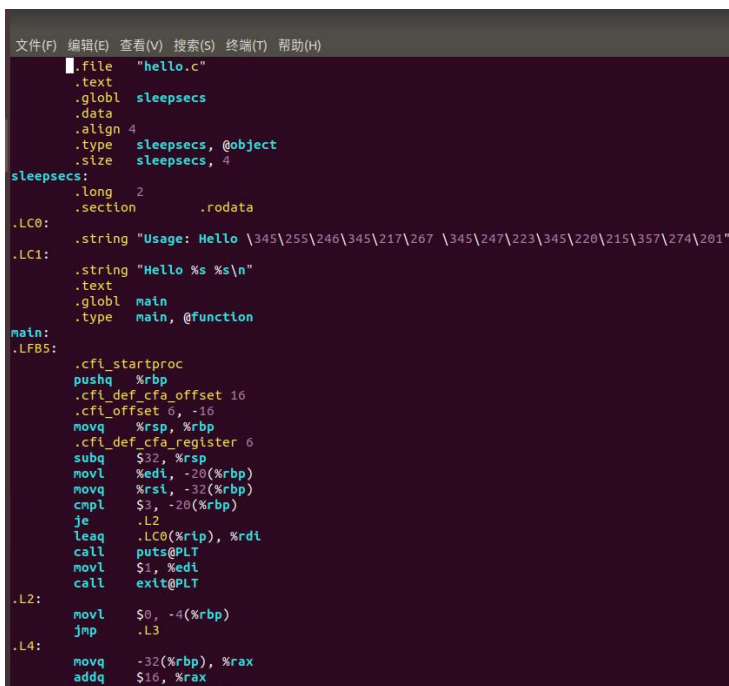
指令: `gcc -S hello.i -o hello.s`



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

图 3-1

3.3 Hello 的编译结果解析



```

.file "hello.c"
.text
.globl _start
.data
.align 4
.type _start, @function
.size _start, 4
_start:
.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 %i, %edi
call exit@PLT
.L2:
movl $0, -4(%rbp)
jmp .L3
.L4:
movq -32(%rbp), %rax
addq $16, %rax
```

```

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
.LFE5:
.size    main, .-main
.ident   "GCC: (Ubuntu 7.3.0-16ubuntu3) 7.3.0"
.section .note.GNU-stack,"",@progbits

```

图 3-2 编译结果展示

3.3.1 数据

3.3.1.1 全局变量

```

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

```

图 3-3 全局变量 sleepsecs

图中是 hello.s 中有关全局变量 sleepsecs 的内容。可以看出它的 size 是 4 字节的，然后其类型是 long，值是 2。

3.3.1.2 局部变量

```

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

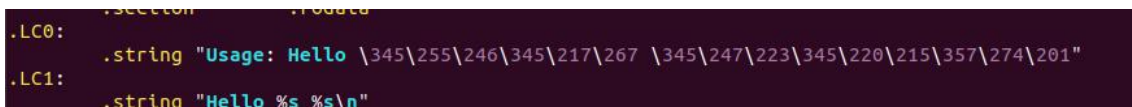
```

图 3-4 局部变量 i

图中是局部变量 `i`，是循环中用来记数的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，可以看出 `i` 占据了栈中的 4 字节。

除此之外还有局部变量 `argc` 作为第一个参数传入。

3.3.1.3 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"
```

图 3-5 字符串

`hello.c` 中共有两个字符串：“Usage: Hello 学号 姓名!\n”以及“Hello %s %s\n”，

3.3.2 类型转换

程序中含有一个隐式类型转换：将 `sleepsecs` 的浮点数类型 `2.5` 转换为 `int` 类型，遵从向零舍入的规则，将 `2.5` 舍入为 `2`。

3.3.3 赋值

编译器对赋值操作的处理是将其编译为汇编指令 `MOV`。根据不同大小的数据类型有 `movb`、`movw`、`movl`、`movq`、`movabsq`。

程序中共出现了两次赋值操作：`i=0` 以及 `argc=2.5`。

3.3.4 算术操作

程序中出现的算数操作是“`i++`”，每次循环后 `i` 的值加一，汇编指令如图



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

图 3-6

3.3.5 关系操作

编译器通常通过将其编译为 `CMP` 指令实现。根据不同的数据大小，有 `cmpb`、`cmpw`、`cmpl` 和 `cmpq`。

程序中共出现了两次关系操作：`argc != 3` 以及 `i < 10`。汇编指令如图



```
cmpl    $3, -20(%rbp)
cmpl    $9, -4(%rbp)
```

图 3-7

3.3.6 控制转移

程序中一共出现了两次控制转移：

1. `if(argv!=3)`：当 `argv` 不等于 3 的时候执行程序段中的代码，如果等于，那么跳过接下来程序段中的代码。汇编代码如图。

```

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

```

图 3-8

2. for(i=0;i<10;i++): 使用计数变量 i 循环 10 次。首先无条件跳转到位于循环体.L4 之后的比较代码, 使用 `cmpl` 进行比较, 如果 $i \leq 9$, 则跳入.L4 for 循环体执行, 否则说明循环结束, 顺序执行 for 之后的内容。

```

.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

```

图 3-9

3.3.7 函数操作

1. main 函数: 储存在.test 节, 程序运行时由系统启动函数调用。如图 3-10

```

.text
.globl  main
.type  main, @function

```

图 3-10

2. printf 函数：这里的共有两个 printf()函数，其中第一个由于只有一个参数所以被优化为 puts。图 3-11 3-12

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

图 3-10

```
movq    -32(%rbp), %rax
addq    $16, %rax
movq    (%rax), %rdx
movq    -32(%rbp), %rax
addq    $8, %rax
movq    (%rax), %rax
movq    %rax, %rsi
leaq    .LC1(%rip), %rdi
movl    $0, %eax
call    printf@PLT
```

图 3-11

3. exit 函数：参数为 1，将 1 赋值给%edi，然后调用 exit 函数。

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

图 3-12

4. sleep 函数：参数为 sleepsecs+%rip，其中%rip 用于保存下一条指令地址。

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

图 3-13

5. getchar 函数：没有参数，执行完循环后直接用指令调用。

3.4 本章小结

本章主要讲述了编译的概念原理及作用。详细描述了 hello.i 被编译为 hello.s 后对 hello.s 的解析。并针对 hello.s 解析了生成汇编代码时需要进行的各种对于数据和操作的转换。这一步生成的汇编语言是高级语言（c 语言）向机器语言的国度，为接下来生成机器码做出铺垫。

（第 3 章 2 分）

第 4 章 汇编

4.1 汇编的概念与作用

汇编的概念：

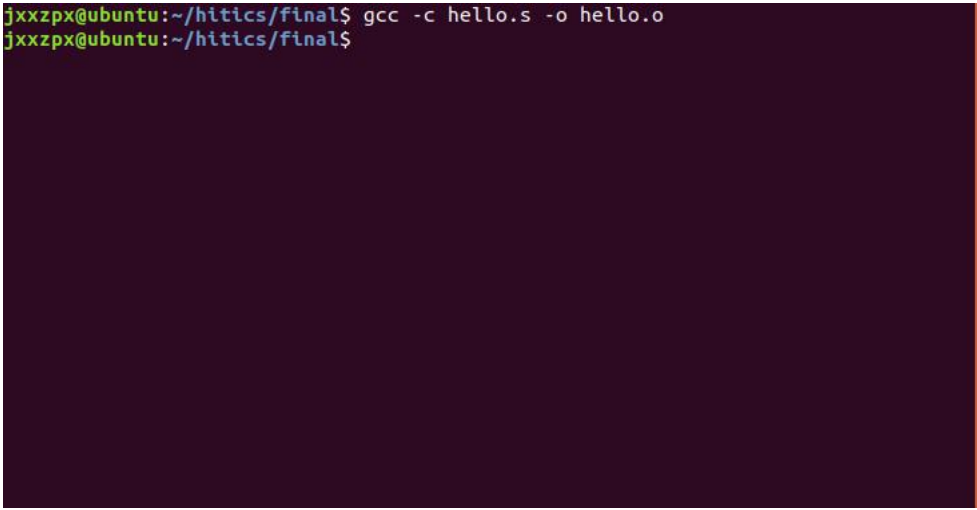
汇编是指从 .s 到 .o 即编译后的文件到生成机器语言二进制程序的过程。汇编器（as）将.s 汇编程序翻译成机器语言指令，把这些指令打包成可重定位目标程序的格式，并将结果保存在.o 目标文件中，.o 文件是一个二进制文件，它包含程序的指令编码。

汇编的作用：

将汇编代码转换为二进制文件，使其在之后被链接完成之后可以直接执行。

4.2 在 Ubuntu 下汇编的命令

指令：gcc -c hello.s -o hello.o



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

图 4-1

4.3 可重定位目标 elf 格式

1. ELF 头：以 Magic 序列开始，描述了生成该文件的系统的字的大小和字节顺序，剩下的部分包含类别、数据、版本、系统架构等信息。


```
jxxzpx@ubuntu:~/hitics/final$ readelf -a 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
```

图 4-2

2.节头: 总的描述 ELF 文件各个信息的段, 包括节的类型、位置和大小等信息。

```
节头:
[号] 名称      类型      地址      偏移量
     大小      全体大小  旗标  链接  信息  对齐
[ 0]      NULL      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
     00000000000000198 0000000000000018 11 9 8
[11] .strtab      STRTAB   0000000000000000 000002e8
     000000000000004d 0000000000000000 0 0 1
[12] .shstrtab    STRTAB   0000000000000000 00000410
     0000000000000061 0000000000000000 0 0 1
```

图 4-3

3.重定位节 ‘.rela.text’: 个.text 节中位置的列表, 包含.text 节中需要进行重定位的信息, 当链接器把这个目标文件和其他文件组合时, 需要修改这些位置。


```

重定位节 '.rel.text' at offset 0x338 contains 8 entries:
偏移量      信息      类型      符号值      符号名称 + 加数
000000000018 000500000002 R_X86_64_PC32 0000000000000000 .rodata - 4
00000000001d 000c00000004 R_X86_64_PLT32 0000000000000000 puts - 4
000000000027 000d00000004 R_X86_64_PLT32 0000000000000000 exit - 4
000000000050 000500000002 R_X86_64_PC32 0000000000000000 .rodata + 1a
00000000005a 000e00000004 R_X86_64_PLT32 0000000000000000 printf - 4
000000000060 000900000002 R_X86_64_PC32 0000000000000000 sleepsecs - 4
000000000067 000f00000004 R_X86_64_PLT32 0000000000000000 sleep - 4
000000000076 001000000004 R_X86_64_PLT32 0000000000000000 getchar - 4

```

图 4-4

4. 重定位节 '.rel.eh_frame'：保存 eh_frame 节的重定位信息。

```

重定位节 '.rel.eh_frame' at offset 0x3f8 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 4-5

5. 符号表：存放程序中定义和引用的函数和全局变量的信息。

```

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

```

图 4-6

4.4 Hello.o 的结果解析

```

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
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   %rax,%rsi
80: c3                             retq

```

图 4-7

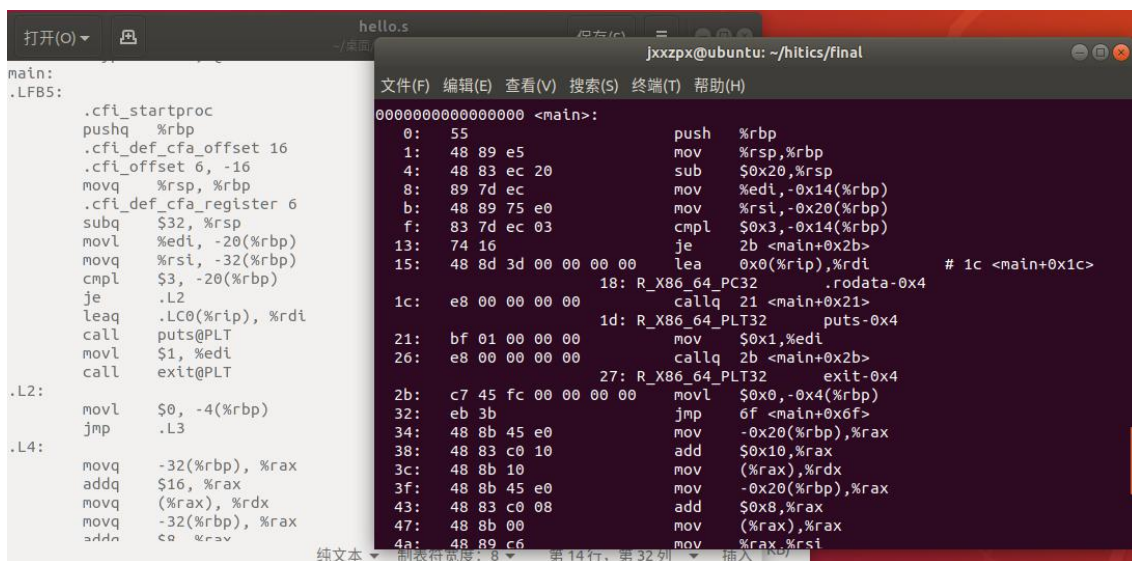


图 4-8

图 4-7 为反汇编后的文件，图 4-8 为该反汇编代码与 hello.s 的对比。编译与反

汇编得到代码都包含程序实现基本的汇编语言代码。然而对于分支转移，编译生成的代码跳转目标通过例如.L2 表示，而反汇编的代码通过一个地址表示。除此之外，函数调用时，`hello.s` 文件中调用一个函数只需被表示成 `call+函数名`，但是在 `hello.o` 反汇编的结果中的 `call` 是 `call` 一个具体的地址位置。

除此之外，在反汇编的代码中还包含了一些的机器语言代码及相关注释。

4.5 本章小结

本章介绍了汇编的概念及作用，解析了 `hello.o` 的 `elf` 格式，除此之外还使用 `objdump` 得到反汇编代码与 `hello.s` 进行了比较，了解了汇编语言与机器语言的映射关系。

(第 4 章 1 分)

第 5 章 链接

5.1 链接的概念与作用

链接是将各种代码和数据片段收集并组合成一个单一文件的过程，这个文件可被加载到内存并执行。链接可以执行于编译时，也就是在源代码被编译成机器代码时；也可以执行于加载时，也就是在程序被加载器加载到内存并执行时；甚至于运行时，也就是由应用程序来执行。链接是由叫做链接器的程序执行的。

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
```

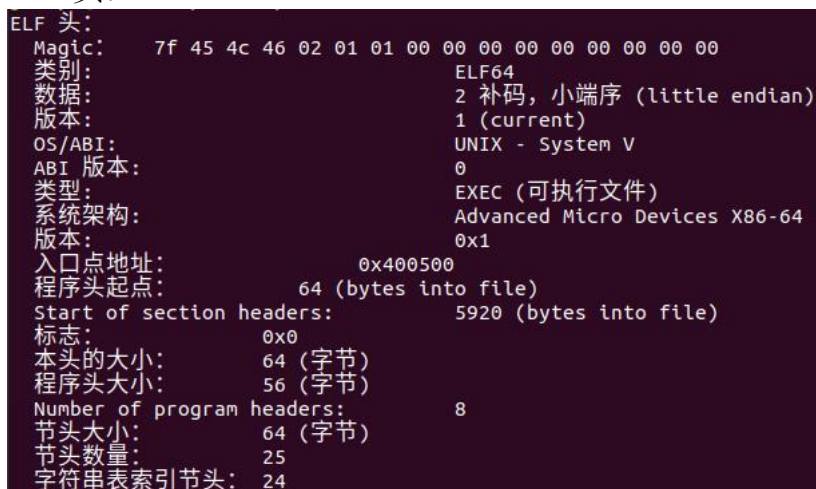


```
jxxzpx@ubuntu:~/hitics/final$ ld -o hello -dynamic-linker /lib64/ld-linux-x86-64
.so.2 /usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o hello.
o /usr/lib/x86_64-linux-gnu/libc.so /usr/lib/x86_64-linux-gnu/crtn.o
jxxzpx@ubuntu:~/hitics/final$
```

图 5-1

5.3 可执行目标文件 hello 的格式

ELF 头：



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

图 5-2

节头:

节头: [号]	名称 大小	类型 全体大小	地址 旗标	链接 链接	偏移量 信息	对齐
[0]	0000000000000000	NULL	0000000000000000	0	0	0
[1]	.interp 000000000000001c	PROGBITS 0000000000000000	0000000000400200 A	0	0	1
[2]	.note.ABI-tag 0000000000000020	NOTE 0000000000000000	000000000040021c A	0	0	4
[3]	.hash 0000000000000034	HASH 0000000000000004	0000000000400240 A	5	0	8
[4]	.gnu.hash 000000000000001c	GNU_HASH 0000000000000000	0000000000400278 A	5	0	8
[5]	.dynsym 00000000000000c0	DYNSYM 0000000000000018	0000000000400298 A	6	1	8
[6]	.dynstr 0000000000000057	STRTAB 0000000000000000	0000000000400358 A	0	0	1
[7]	.gnu.version 0000000000000010	VERSYM 0000000000000002	00000000004003b0 A	5	0	2
[8]	.gnu.version_r 0000000000000020	VERNEED 0000000000000000	00000000004003c0 A	6	1	8
[9]	.rela.dyn 0000000000000030	RELA 0000000000000018	00000000004003e0 A	5	0	8
[10]	.rela.plt 0000000000000078	RELA 0000000000000018	0000000000400410 AI	5	19	8
[11]	.init 0000000000000017	PROGBITS 0000000000000000	0000000000400488 AX	0	0	4
[12]	.plt 0000000000000060	PROGBITS 0000000000000010	00000000004004a0 AX	0	0	16
[13]	.text 0000000000000132	PROGBITS 0000000000000000	0000000000400500 AX	0	0	16
[14]	.fini 0000000000000009	PROGBITS 0000000000000000	0000000000400634 AX	0	0	4
[15]	.rodata 000000000000002f	PROGBITS 0000000000000000	0000000000400640 A	0	0	4
[16]	.eh_frame 00000000000000fc	PROGBITS 0000000000000000	0000000000400670 A	0	0	8
[17]	.dynamic 00000000000001a0	DYNAMIC 0000000000000010	0000000000600e50 WA	6	0	8
[18]	.got 0000000000000010	PROGBITS 0000000000000008	0000000000600ff0 WA	0	0	8
[19]	.got.plt 0000000000000040	PROGBITS 0000000000000008	0000000000601000 WA	0	0	8
[20]	.data 0000000000000008	PROGBITS 0000000000000000	0000000000601040 WA	0	0	4
[21]	.comment 0000000000000024	PROGBITS 0000000000000001	0000000000000000 MS	0	0	1
[22]	.symtab 0000000000000498	SYMTAB 0000000000000018	0000000000000000 23 28	0	0	8
[23]	.strtab 0000000000000150	STRTAB 0000000000000000	0000000000000000 0	0	0	1
[24]	.shstrtab 00000000000000c5	STRTAB 0000000000000000	0000000000000000 0	0	0	1

图 5-3 各节起始地址、大小

5.4 hello 的虚拟地址空间

使用 edb 打开 hello 文件，然后通过 data dump 进行查看。结果如图。

Data Dump		
0x0000000000400000-0x0000000000401000		
00000000:00400000	7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00	.ELF.....
00000000:00400010	02 00 3e 00 01 00 00 00 00 05 40 00 00 00 00 00	...>...@....
00000000:00400020	40 00 00 00 00 00 00 00 20 17 00 00 00 00 00 00	@.....
00000000:00400030	00 00 00 00 40 00 38 00 08 00 40 00 19 00 18 00	...@.8...@....
00000000:00400040	06 00 00 00 04 00 00 00 40 00 00 00 00 00 00 00@.....
00000000:00400050	40 00 40 00 00 00 00 00 40 00 40 00 00 00 00 00	@.@.....@.....
00000000:00400060	c0 01 00 00 00 00 00 00 c0 01 00 00 00 00 00 00
00000000:00400070	08 00 00 00 00 00 00 00 03 00 00 00 04 00 00 00
00000000:00400080	00 02 00 00 00 00 00 00 00 02 40 00 00 00 00 00@.....
00000000:00400090	00 02 40 00 00 00 00 00 1c 00 00 00 00 00 00 00@.....
00000000:004000a0	1c 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00
00000000:004000b0	01 00 00 00 05 00 00 00 00 00 00 00 00 00 00 00
00000000:004000c0	00 00 40 00 00 00 00 00 00 00 40 00 00 00 00 00@.....@.....
00000000:004000d0	6c 07 00 00 00 00 00 00 6c 07 00 00 00 00 00 00l.....l.....
00000000:004000e0	00 00 20 00 00 00 00 00 01 00 00 00 06 00 00 00
00000000:004000f0	50 0e 00 00 00 00 00 00 50 0e 60 00 00 00 00 00	P.....P.....
00000000:00400100	50 0e 60 00 00 00 00 00 f8 01 00 00 00 00 00 00	P.....
00000000:00400110	f8 01 00 00 00 00 00 00 00 00 20 00 00 00 00 00
00000000:00400120	02 00 00 00 06 00 00 00 50 0e 00 00 00 00 00 00P.....
00000000:00400130	50 0e 60 00 00 00 00 00 50 0e 60 00 00 00 00 00	P.....P.....
00000000:00400140	a0 01 00 00 00 00 00 00 a0 01 00 00 00 00 00 00
00000000:00400150	08 00 00 00 00 00 00 00 04 00 00 00 04 00 00 00
00000000:00400160	1c 02 00 00 00 00 00 00 1c 02 40 00 00 00 00 00@.....
00000000:00400170	1c 02 40 00 00 00 00 00 20 00 00 00 00 00 00 00@.....
00000000:00400180	20 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00
00000000:00400190	51 e5 74 64 06 00 00 00 00 00 00 00 00 00 00 00	Q3td.....
00000000:004001a0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:004001b0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:004001c0	10 00 00 00 00 00 00 00 52 e5 74 64 04 00 00 00R3td.....
00000000:004001d0	50 0e 00 00 00 00 00 00 50 0e 60 00 00 00 00 00	P.....P.....
00000000:004001e0	50 0e 60 00 00 00 00 00 b0 01 00 00 00 00 00 00	P.....
00000000:004001f0	b0 01 00 00 00 00 00 00 01 00 00 00 00 00 00 00
00000000:00400200	2f 6c 69 62 36 34 2f 6c 64 2d 6c 69 6e 75 78 2d	/lib64/ld-linux-
00000000:00400210	78 38 36 2d 36 34 2e 73 6f 2e 32 00 04 00 00 00	x86-64.so.2.....
00000000:00400220	10 00 00 00 01 00 00 00 47 4e 55 00 00 00 00 00GNU.....

图 5-4

00000000:00400350	00 00 00 00 00 00 00 00 6c 69 62 63 2e 73 6flibc.so
00000000:00400360	2e 36 00 65 78 69 74 00 70 75 74 73 00 70 72 69	.6.exit.puts.pri
00000000:00400370	6e 74 66 00 67 65 74 63 68 61 72 00 73 6c 65 65	ntf.getchar.slee
00000000:00400380	70 00 5f 5f 6c 69 62 63 5f 73 74 61 72 74 5f 6d	p....libc.start_m
00000000:00400390	61 69 6e 00 47 4c 49 42 43 5f 32 2e 32 2e 35 00	ain.GLIBC_2.2.5.
00000000:004003a0	5f 5f 67 6d 6f 6e 5f 73 74 61 72 74 5f 5f 00 00	__gmon_start__..
00000000:004003b0	00 00 02 00 02 00 02 00 02 00 00 02 00 02 00 00

图 5-5

00000000:004005f0	ff 48 85 ed 74 20 31 db 0f 1f 84 00 00 00 00 00	[H.mt l].....
00000000:00400600	4c 89 fa 4c 89 f6 44 89 ef 41 ff 14 dc 48 83 c3	L. L. D. [A] [H.]
00000000:00400610	01 48 39 dd 75 ea 48 83 c4 08 5b 5d 41 5c 41 5d	.H9[u]H. [][A\A]
00000000:00400620	41 5e 41 5f c3 90 66 2e 0f 1f 84 00 00 00 00 00	A^A. f.....
00000000:00400630	f3 c3 00 00 48 83 ec 08 48 83 c4 08 c3 00 00 00	.H.].H.]. ...
00000000:00400640	01 00 02 00 55 73 61 67 65 3a 20 48 65 6c 6c 6fUsage: Hello
00000000:00400650	20 e5 ad a6 e5 8f b7 20 e5 a7 93 e5 90 8d ef bc	3000 3.0 3c.3. e
00000000:00400660	81 00 48 65 6c 6c 6f 20 25 73 20 25 73 0a 00 00	..Hello %s %s ..
00000000:00400670	14 00 00 00 00 00 00 00 01 7a 52 00 01 78 10 01zR..x...

图 5-6

5.5 链接的重定位过程分析

计算机系统课程报告

```

Disassembly of section .init:
0000000000400488 <.init>:
400488: 48 83 ec 08      sub    $0x8,%rsp
40048c: 48 8b 05 65 0b 20 00 mov    0x200b05(%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_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 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_2.2.5>
4004d6: 68 02 00 00 00 00 pushq  $0x2
4004db: e9 c0 ff ff ff  jmpq   4004a0 <.plt>

00000000004004e0 <exit@plt>:
4004e0: ff 25 4a 0b 20 00 jmpq   *0x200b4a(%rip)      # 601030 <exit@GLIBC_2.2.5>
4004e6: 68 03 00 00 00 00 pushq  $0x3
4004eb: e9 b0 ff ff ff  jmpq   4004a0 <.plt>

00000000004004f0 <sleep@plt>:
4004f0: ff 25 42 0b 20 00 jmpq   *0x200b42(%rip)      # 601038 <sleep@GLIBC_2.2.5>
4004f6: 68 04 00 00 00 00 pushq  $0x4
4004fb: e9 a0 ff ff ff  jmpq   4004a0 <.plt>

Disassembly of section .text:
0000000000400500 <_start>:
400500: 31 ed          xor     %ebp,%ebp
400502: 49 89 d1      mov     %rdx,%r9
400505: 5e          pop     %rsi
400506: 48 89 e2      mov     %rsp,%rdx
400509: 48 83 e4 f0    and     $0xffffffffffff0,%rsp
40050d: 50          push    %rax
40050e: 54          push    %rsp
40050f: 49 c7 c0 30 06 40 00 mov     $0x400630,%r8
400516: 48 c7 c1 c0 05 40 00 mov     $0x4005c0,%rcx
40051d: 48 c7 c7 32 05 40 00 mov     $0x400532,%rdi
400524: ff 15 c6 0a 20 00 callq   *0x200ac0(%rip)      # 600ff0 <__libc_start_main@GLIBC_2.2.5>
40052a: f4          hlt
40052b: 0f 1f 44 00 00 nopl    0x0(%rax,%rax,1)

0000000000400530 <.dli_relocate_static_pie>:
400530: f3 c3      repz retq

0000000000400532 <main>:
400532: 55          push    %rbp
400533: 48 89 e5      mov     %rsp,%rbp
400536: 48 83 ec 20    sub     $0x20,%rsp
40053a: 89 7d ec      mov     %edi,-0x14(%rbp)
40053d: 48 89 75 e0    mov     %rsi,-0x20(%rbp)
400541: 83 7d ec 03    cmpl    $0x3,-0x14(%rbp)
400545: 74 16          je      40055d <main+0x2b>
400547: 48 8d 3d f6 00 00 00 lea     0xf6(%rip),%rdi      # 400644 <_IO_stdin_used+0x4>
40054e: e8 5d ff ff ff callq   4004b0 <puts@plt>
400553: bf 01 00 00 00 mov     $0x1,%edi
400558: e8 83 ff ff ff callq   4004e0 <exit@plt>
40055d: c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%rbp)
400564: eb 3b          jnp     4005a1 <main+0x6f>
400566: 48 8b 45 e0    mov     -0x20(%rbp),%rax
40056a: 48 83 c0 10    add     $0x10,%rax
40056e: 48 8b 10      mov     (%rax),%rdx
400571: 48 8b 45 e0    mov     -0x20(%rbp),%rax
400575: 48 83 c0 08    add     $0x8,%rax
400579: 48 8b 00      mov     (%rax),%rax
40057c: 48 89 c6      mov     %rax,%rsi
40057f: 48 8d 3d dc 00 00 00 lea     0xdc(%rip),%rdi      # 400662 <_IO_stdin_used+0x22>
400586: b8 00 00 00 00 mov     $0x0,%eax
40058b: e8 30 ff ff ff callq   4004c0 <printf@plt>
400590: 8b 05 ae 0a 20 00 mov     0x200aae(%rip),%eax      # 601044 <sleepsecs>
400596: 89 c7          mov     %eax,%edi
400598: e8 53 ff ff ff callq   4004f0 <sleep@plt>
40059d: 83 45 fc 01    addl    $0x1,-0x4(%rbp)
4005a1: 83 7d fc 09    cmpl    $0x9,-0x4(%rbp)
4005a5: 7e bf          jle     400566 <main+0x34>
4005a7: e8 24 ff ff ff callq   4004d0 <getchar@plt>
4005ac: b8 00 00 00 00 mov     $0x0,%eax
4005b1: c9          leaveq  %eax,%edi
4005b2: c3          retq
4005b3: 66 2e 0f 1f 84 00 00 nopw    %cs:0x0(%rax,%rax,1)
4005ba: 00 00 00      nopb
4005bd: 0f 1f 00      nopl    (%rax)

```



```

0000000004005c0 <__libc_csu_init>:
4005c0: 41 57                push    %r15
4005c2: 41 56                push    %r14
4005c4: 49 89 d7            mov     %rdx,%r15
4005c7: 41 55                push    %r13
4005c9: 41 54                push    %r12
4005cb: 4c 8d 25 7e 08 20 00 lea     0x20087e(%rip),%r12    # 600e50 <_DYNAMIC>
4005d2: 55                push    %rbp
4005d3: 48 8d 2d 76 08 20 00 lea     0x200876(%rip),%rbp    # 600e50 <_DYNAMIC>
4005da: 53                push    %rbx
4005db: 41 89 fd            mov     %edi,%r13d
4005de: 49 89 f6            mov     %rsi,%r14
4005e1: 4c 29 e5            sub     %r12,%rbp
4005e4: 48 83 ec 08        sub     $0x8,%rsp
4005e8: 48 c1 fd 03        sar     $0x3,%rbp
4005ec: e8 97 fe ff ff     callq   400488 <_init>
4005f1: 48 85 ed            test    %rbp,%rbp
4005f4: 74 20              je      400616 <__libc_csu_init+0x56>
4005f6: 31 db              xor     %ebx,%ebx
4005f8: 0f 1f 84 00 00 00 00 nopl    0x0(%rax,%rax,1)
4005ff: 00
400600: 4c 89 fa            mov     %r15,%rdx
400603: 4c 89 f6            mov     %r14,%rsi
400606: 44 89 ef            mov     %r13d,%edi
400609: 41 ff 14 dc        callq   *(%r12,%rbx,8)
40060d: 48 83 c3 01        add     $0x1,%rbx
400611: 48 39 dd            cmp     %rbx,%rbp
400614: 75 ea              jne     400600 <__libc_csu_init+0x40>
400616: 48 83 c4 08        add     $0x8,%rsp
40061a: 5b                pop     %rbx
40061b: 5d                pop     %rbp
40061c: 41 5c                pop     %r12
40061e: 41 5d                pop     %r13
400620: 41 5e                pop     %r14
400622: 41 5f                pop     %r15
400624: c3                retq
400625: 90                nop
400626: 66 2e 0f 1f 84 00 00 nopw    %cs:0x0(%rax,%rax,1)
40062d: 00 00 00
0000000000400630 <__libc_csu_fini>:
400630: f3 c3              repz retq

Disassembly of section .fini:
0000000000400634 <_fini>:
400634: 48 83 ec 08        sub     $0x8,%rsp
400638: 48 83 c4 08        add     $0x8,%rsp
40063c: c3                retq

```

经过比较发现：函数的个数增加，头文件的函数加入至代码中；各类相对寻址确定，动态库函数指向 PLT；函数的起始地址也得到了确定。

5.6 hello 的执行流程

```

_dl_start -> _dl_init -> _start -> _libc_start_main -> _init -> _main -> _printf -> _exit
-> _getchar -> _dl_runtime_resolve_xsave -> _dl_fixup_dl_lookup_symbol_x -> exit

```

5.7 Hello 的动态链接分析

动态链接把程序按照模块拆分成各个相对独立部分，只在程序运行时才链接在一起形成一个完整的程序，与静态连接不同。因此编译器没有办法预测函数的运行时地址，故而需要添加重定位记录，等待动态链接器处理。GNU 编译系统采用延迟绑定技术来解决动态库函数模块调用的问题，它将过程地址的绑定推迟到了第一次调用该过程时。

5.8 本章小结

本章介绍了链接的概念和作用，还介绍了链接过程中的指令，并且对 `hello` 的 ELF 格式，`hello` 的虚拟地址空间、执行流程、重定位和动态链接进行了介绍，体现了动态链接与静态链接的区别等。

(第 5 章 1 分)

第 6 章 hello 进程管理

6.1 进程的概念与作用

进程的概念：

- 1) 进程是程序的一次执行过程。
- 2) 进程是一个程序及其数据在处理机上顺序执行时所发生的活动。
- 3) 进程是具有独立功能的程序在一个数据集合上运行的过程，它是系统进行资源分配和调度的一个独立单位。

进程的作用：

在多道程序环境下，允许多个程序并发执行，此时它们将失去封闭性，并具有间断性及不可再现性的特征。为此引入了进程的概念，以便更好地描述和控制程序的并发执行，实现操作系统的并发性和共享性。

6.2 简述壳 Shell-bash 的作用与处理流程

作用：

shell 是一种交互型的应用级程序。它能够接收用户命令，然后调用相应的应用程序，即代表用户运行其他程序。

处理流程：

1. 用户在命令行中键入命令
2. Shell 将命令字符串分割填充到参数数组
3. Shell 判断是否是内置命令，若是，立即执行
4. 若非内置则调用相应程序为其分配子进程。
5. Shell 可以接受信号，并且做出相应处理

6.3 Hello 的 fork 进程创建过程

首先，shell 会对于 ./hello 指令进行判断，判断其是否是内部指令，结果是否定的。

接下来，shell 会调用 fork() 函数创建一个一个新的子进程，通过返回值确定子进程的 pid 号。新创建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一份副本，包括代码和数据段、堆、共享库以及用户

栈。子进程还获得与父进程任何打开文件描述符相同的副本，这就意味着当父进程调用 `fork` 时。子进程可以读写父进程中打开的任何文件。父进程和新创建的子进程的区别在于 `pid` 不同。

父进程会默认等待子进程执行完之后回收子进程，但是也会有产生僵死进程的情况，父进程可以调用 `waitpid` 函数等待其子进程终止或停止。

6.4 Hello 的 `execve` 过程

对于已经开辟的子进程，`shell` 会通过 `execve` 函数在当前进程的上下文中加载并运行一个新程序。`execve` 加载并运行可执行目标文件，且带参数列表 `argv` 和环境变量列表 `envp`。只有当出现错误时，`execve` 才会返回到调用程序。

在 `execve` 加载了可执行程序之后，它调用启动代码。启动代码设置栈，并将控制传递给新程序的主函数，即可执行程序的主函数。此时用户栈已经包含了命令行参数与环境变量，进入 `main` 函数后便开始逐步运行程序。

6.5 hello 的进程执行

6.5.1 进程的相关概述

1. 逻辑控制流

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

2. 并发

同时执行的两个流被称为并发流，这两个流被称为并发地执行。多个流并发地执行的一般现象被称为并发。一个进程和其他进程轮流运行的概念被称为多任务。除此之外，一个进程执行它的控制流的一部分的每一时间段叫做时间片。因此，多任务也叫时间分片。

3. 用户模式和内核模式

`shell` 提供了一个平台供用户向操作系统提出请求，操作系统看心情满足用户，但是这种交互在一定程度上是危险的，因为它提供了用户修改内核的机会，所以 `shell` 必须要设置一些防护措施来保护内核，限制指令的类型和可以作用的范围。处理器通常会在某个控制寄存器中设置一个模式位用来描述当前进程所拥有的特权，当设置了模式位后，进程处于内核模式，该进程可以执行指令集中任何指令，同时也能访问系统中任何内存位置。

4. 上下文切换

在内核中，`shell` 对每个进程都会维持一个上下文。所谓上下文，就是内核重新启动时的一个被强占的进程所需的状态。

6.5.2 hello 执行过程

`Hello` 在刚开始时被保存为一个上下文，如果没有异常或者中断信号的话，`hello` 会继续正常执行，如果有异常或者系统中断，则会在内核模式下进行上下文切换，

将控制传递给其他进程。

上下文切换的过程：

- 1) 保存当前进程的上下文
- 2) 恢复现在调度进程的上下文
- 3) 将控制传给新恢复进程 Hello 中的 sleep 函数会显示请求调用进程休眠：

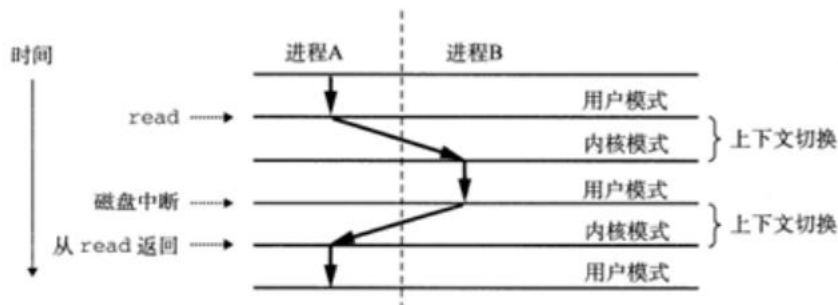


图 6-1

6.6 hello 的异常与信号处理

hello 执行过程中出现的异常种类可能会有：中断、陷阱、故障、终止。

```
jxxzpx@ubuntu:~/hitics/final$ ./hello 1170300424 谢卓芯
Hello 1170300424 谢卓芯
Hello 1170300424 谢卓芯
Hello 1170300424 谢卓芯
Hello 1170300424 谢卓芯
Hello 1170300424 谢卓芯
Hello 1170300424 谢卓芯
Hello 1170300424 谢卓芯
hbadghkHello 1170300424 谢卓芯
^Z
[1]+ 已停止                  ./hello 1170300424 谢卓芯
```

图 6-2 ctrl+Z 进程被挂起

```
jxxzpx@ubuntu:~/hitics/final$ ./hello 1170300424 谢卓芯
Hello 1170300424 谢卓芯
Hello 1170300424 谢卓芯
^C
```

图 6-3 ctrl+C 进程中止

```
jxxzpx@ubuntu:~/hitics/final$ ps
  PID TTY          TIME CMD
  5993 pts/0        00:00:01 bash
 11484 pts/0        00:00:00 hello
 11514 pts/0        00:00:00 hello
 11516 pts/0        00:00:00 hello
 11517 pts/0        00:00:00 ps
```

图 6-4 ctrl+Z 后 ps 操作查看 pid

```
jxxzpx@ubuntu:~/hitics/final$ jobs
[1] 已停止                  ./hello 1170300424 谢卓芯
[2]- 已停止                  ./hello 1170300424 谢卓芯
[3]+ 已停止                  ./hello 1170300424 谢卓芯
```

图 6-5 ctrl+Z 后 jobs 操作

```
jxxzpx@ubuntu:~/hitics/final$ pstree
systemd--ModemManager--2*[{ModemManager}]
--NetworkManager--dhclient
--2*[{NetworkManager}]
--VGAAuthService
--accounts-daemon--2*[{accounts-daemon}]
--acpid
--avahi-daemon--avahi-daemon
--bluetoothd
--boltd--2*[{boltd}]
--colord--2*[{colord}]
--cron
--cups-browsed--2*[{cups-browsed}]
--cupsd
--2*[{dbus-daemon}]
--fcitx--{fcitx}
--fcitx-dbus-watc
--fwupd--4*[{fwupd}]
--gdm3--gdm-session-wor--gdm-wayland-ses--gnome-session-b--gnome-sh+
--gnome-sh+
--gsd-a11y+
--gsd-clip+
--gsd-colo+
--gsd-date+
--gsd-hous+
--gsd-keyb+
--gsd-medi+
--gsd-mous+
--gsd-powe+
--gsd-prin+
--gsd-rfki+
--gsd-scre+
--gsd-shar+
--gsd-smar+
--gsd-soun+
--gsd-waco+
--gsd-xset+
--3*[{gnom+
--2*[{gdm-wayland-ses}]
--2*[{gdm-session-wor}]
```

图 6-6 ctrl+Z 后显示 pstree 命令（部分）

```
jxxzpx@ubuntu:~/hitics/final$ fg
./hello 1170300424 谢卓芯
Hello 1170300424 谢卓芯
Hello 1170300424 谢卓芯
q

Hello 1170300424 谢卓芯
sd
sc
Hello 1170300424 谢卓芯

Hello 1170300424 谢卓芯
Hello 1170300424 谢卓芯
^Z
[2]+ 已停止 ./hello 1170300424 谢卓芯
```

图 6-7 任务重新回到前台 乱按和回车无影响

6.7 本章小结

本章主要介绍了进程的概念与作用，hello 的 fork() 和 execve() 过程等。并且对进程的各种概念进行了概述，并且简要介绍了 hello 进程的执行过程，并且切实尝试了针对 hello 的各种操作，对其对于各种异常信号的处理进行了分析和截图展示。

（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：

程序代码经过编译后出现在汇编程序中地址。逻辑地址由选择符（在实模式下是描述符，在保护模式下是用来选择描述符的选择符）和偏移量（偏移部分）组成。

物理地址：

在存储器里以字节为单位存储信息，为正确地存放或取得信息，每一个字节单元给以一个唯一的存储器地址，称为物理地址，又叫实际地址或绝对地址。在实地址方式下，物理地址是通过段地址乘以 16 加上偏移地址得到的。而 16 位的段地址乘以 16 等同于左移 4 位二进制位，这样变成 20 位的段基地址，最后段基地址加上段内偏移地址即可得到物理地址。

线性地址：

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。从管理和效率的角度出发，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 2^{20} 的大数组，共有 2 的 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。

7.2 Intel 逻辑地址到线性地址的变换-段式管理

段式管理，是指把一个程序分成若干个段进行存储，每个段都是一个逻辑实体，程序员需要知道并使用它。它的产生是与程序的模块化直接有关的。段式管理是通过段表进行的，它包括段号或段名、段起点、装入位、段的长度等。此外还需要主存占用区域表、主存可用区域表。

为了实现段式管理，操作系统需要如下的数据结构来实现进程的地址空间到物理内存空间的映射，并跟踪物理内存的使用情况，以便在装入新的段的时候，合理地分配内存空间。

·进程段表：描述组成进程地址空间的各段，可以是指向系统段表中表项的索引。每段有段基址。

·系统段表：系统所有占用段。

·空闲段表：内存中所有空闲段，可以结合到系统段表中。

7.3 Hello 的线性地址到物理地址的变换-页式管理

将各进程的虚拟空间划分成若干个长度相等的页，页式管理把内存空间按页的大小页划分成片或者页面，然后把页式虚拟地址与内存地址建立一一对应页表，并用相应的硬件地址变换机构，来解决离散地址变换问题。页式管理采用请求调页或预调页技术实现了内外存存储器的统一管理。

在页式系统中进程建立时，操作系统为进程中所有的页分配页框。当进程撤销时收回所有分配给它的页框。在程序的运行期间，如果允许进程动态地申请空间，操作系统还要为进程申请的空间分配物理页框。操作系统为了完成这些功能，必须记录系统内存中实际的页框使用情况。操作系统还要在进程切换时，正确地切换两个不同的进程地址空间到物理内存空间的映射。这就要求操作系统要记录每个进程页表的相关信息。为了完成上述的功能，一个页式系统中，一般要采用如下的数据结构。

进程页表：完成逻辑页号(本进程的地址空间)到物理页面号(实际内存空间)的映射。

每个进程有一个页表，描述该进程占用的物理页面及逻辑排列顺序。

物理页面表：整个系统有一个物理页面表，描述物理内存空间的分配使用情况，其数据结构可采用位示图和空闲页链表。

请求表：整个系统有一个请求表，描述系统内各个进程页表的位置和大小，用于地址转换也可以结合到各进程的 PCB(进程控制块)里。

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

CPU 每次产生一个虚拟地址，MMU 要查阅相应的 PTE，它造成了巨大的时间开销，为了消除它，MMU 中存在一个关于 PTE 的小的缓存，称为翻译后备缓冲器（TLB）。TLB 通过虚拟地址 VPN 部分进行索引，分为索引（TLBI）与标记（TLBT）两个部分。这样，MMU 在读取 PTE 时会直接通过 TLB，如果不命中再

从内存中将 PTE 复制到 TLB。在以上机制的基础上，如果所使用的仅仅是虚拟地址空间中很小的一部分，那么仍然需要一个与使用较多空间相同的页表，造成了内存的浪费。所以虚拟地址到物理地址的转换过程中还存在多级页表的机制：上一级的页表映射到下一级也表，直到页表映射到虚拟内存，如果下一级内容都未分配，那么页表项则为空，不映射到下一级，也不存在下一级页表，当分配时再创建相应页表，从而节约内存空间。

7.5 三级 Cache 支持下的物理内存访问

因为共 64 组，所以需要 6bit CI 进行组寻址，并且共有 8 路，因为块大小为 64B 所以需要 6bit CO 表示数据偏移位置，因为 VA 共 52bit，所以 CT 共 40bit。

根据已经获得的物理地址 VA，使用 CI 进行组索引，每组 8 路，对 8 路的块分别匹配 CT，若匹配成功且块的 valid 标志位为 1，则命中，根据数据偏移量 CO 取出数据返回。

如果没有匹配成功或者匹配成功但是标志位是 1，则不命中，向下一级缓存中查询数据。查询到数据之后，一种简单的放置策略如下：如果映射到的组内有空闲块，则直接放置，否则组内都是有效块，产生冲突，则采用最近最少使用策略 LFU 进行替换。

7.6 hello 进程 fork 时的内存映射

Fork 函数被当前进程调用时，内核为新进程创建各种数据结构，并且分配唯一 pid。为了给新进程创建虚拟内存，它创建了当前进程的 mm_struct、区域结构和页表原样副本，将两个进程中的每个页面到标记为只读，同时设置每个区域结构为私有对象。

7.7 hello 进程 execve 时的内存映射

利用 execve 函数加载 hello 程序：

- 1.删除已存在的用户区域：删除当前进程虚拟地址用户部分的区域结构
- 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，同时标记为私有的写时复制的。代码和数据段映射到 hello 的.text 及.data 段，.bss 请求二进制零，映射到匿名文件，其大小在第 5 章链接中的程序头部表中，堆栈也是请求二进制零，初始长度为零。

3.映射共享区：hello 与系统执行文件链接，如 lib.so，这部分映射到共享区域。

4.设置程序计数器 PC：设置当前进程上下文中的 PC，指向 entry point。

7.8 缺页故障与缺页中断处理

缺页故障：

进程线性地址空间里的页面不必常驻内存，在执行一条指令时，如果发现他要访问的页没有在内存中（即存在位为 0），那么停止该指令的执行，并产生一个页不存在的异常，对应的故障处理程序可通过从外存加载该页的方法来排除故障，之后，原先引起的异常的指令就可以继续执行，而不再产生异常。

缺页中断处理：

- 1) 首先硬件会陷入内核，在堆栈中保存程序计数器。大多数机器将当前指令的各种状态信息保存在 CPU 中特殊的寄存器中。
- 2) 启动一个汇编代码例程保存通用寄存器及其它易失性信息，以免被操作系统破坏。
- 3) 当操作系统发现是一个页面中断时，查找出来发生页面中断的虚拟页面。这个虚拟页面的信息通常会保存在一个硬件寄存器中，如果没有的话，操作系统必须检索程序计数器，取出这条指令，用软件分析该指令，通过分析找出发生页面中断的虚拟页面。
- 4) 检查虚拟地址的有效性及安全保护位。如果发生保护错误，则杀死该进程。
- 5) 操作系统查找一个空闲的页框(物理内存中的页面)，如果没有空闲页框则需要通过页面置换算法找到一个需要换出的页框。
- 6) 如果找的页框中的内容被修改了，则需要将修改的内容保存到磁盘上，此时会引起一个写磁盘调用，发生上下文切换（在等待磁盘写的过程中让其它进程运行）。
- 7) 页框干净后，操作系统根据虚拟地址对应磁盘上的位置，将保持在磁盘上的页面内容复制到干净的页框中，此时会引起一个读磁盘调用，发生上下文切换。
- 8) 当磁盘中的页面内容全部装入页框后，向操作系统发送一个中断。操作系统更新内存中的页表项，将虚拟页面映射的页框号更新为写入的页框，并将页框标记

为正常状态。

9) 恢复缺页中断发生前的状态，将程序指令器重新指向引起缺页中断的指令。

10) 调度引起页面中断的进程，操作系统返回汇编代码例程。

11) 汇编代码例程恢复现场，将之前保存在通用寄存器中的信息恢复。

7.9 动态存储分配管理

动态存储分配:

动态存储分配方式是不一次性将整个程序装入到主存中。可根据执行的需要，部分地动态装入。同时，在装入主存的程序不执行时，系统可以收回该程序所占据的主存空间。再者，用户程序装入主存后的位置，在运行期间可根据系统需要而发生改变。此外，用户程序在运行期间也可动态地申请存储空间以满足程序需求。由此可见，动态存储分配方式在存储空间的分配和释放上，表现得十分灵活，现代的操作系统常采用这种存储方式。

首次适应算法:

我们以空闲分区链为例来说明采用 首次适应算法(first fit)时的分配情况。FF 算法要求空闲分区链以地址递增的次序链接。在分配内存时，从链首开始顺序查找，直至找到一个大小能满足要求的空闲分区为止；然后再按照作业的大小，从该分区中划出一块内存空间分配给请求者，余下的空闲分区仍留在空闲链中。若从链首直至链尾都不能找到一个能满足要求的分区，则此次内存分配失败，返回。该算法倾向于优先利用内存中低址部分的空间分区，从而保留了高址部分的大空闲区。这给以后到达的大作业分配大的内存空间创造了条件。其缺点是低址部分不断被划分，会留下许多难以利用的、很小的空闲分区，而每次查找又都是从低址部分开始，这无疑会增加查找可用空闲分区时的开销。

循环首次适应算法:

该算法是由首次适应算法演变而成的。在为进程分配内存空间时，不再是每次都从链首开始查找，而是从上次找到的空闲分区的下一个空闲分区开始查找，直至找到一个能满足要求的空闲分区，从中划出一块与请求大小相等的内存空间分配给作业。为实现该算法，应设置一起始查寻指针，用于指示下一次起始查寻的空闲分区，并采用循环查找方式，即如果最后一个(链尾)空闲分区的大小仍不能满足

要求，则应返回到第一个空闲分区，比较其大小是否满足要求。找到后，应调整起始查寻指针。该算法能使内存中的空闲分分布得更均匀，从而减少了查找空闲分区时的开销，但这样会缺乏大的空闲分区。

最佳适应算法:

算法：将空闲分区链中的空闲分区按照空闲分区由小到大的顺序排序，从而形成空闲分区链。每次从链首进行查找合适的空闲分区为作业分配内存，这样每次找到的空闲分区是和作业大小最接近的，所谓“最佳”。

优点：第一次找到的空闲分区是大小最接近待分配内存作业大小的；

缺点:产生大量难以利用的外部碎片。

最坏适应算法:

算法：与最佳适应算法刚好相反，将空闲分区链的分区按照从大到小的顺序排序形成空闲分区链，每次查找时只要看第一个空闲分区是否满足即可。

优点：效率高，分区查找方便；

缺点：当小作业把大空闲分区分小了，那么，大作业就找不到合适的空闲分区。

快速适应算法:

该算法又称为分类搜索法，是将空闲分区根据其容量大小进行分类，对于每一类具有相同容量的所有空闲分区，单独设立一个空闲分区链表，这样，系统中存在多个空闲分区链表，同时在内存中设立一张管理索引表，该表的每一个表项对应了一种空闲分区类型，并记录了该类型空闲分区链表表头的指针。空闲分区的分类是根据进程常用的空间大小进行划分，如 2 KB、4 KB、8 KB 等，对于其它大小的分区，如 7 KB 这样的空闲区，既可以放在 8 KB 的链表中，也可以放在一个特殊的空闲区链表中。该算法的优点是查找效率高，仅需要根据进程的长度，寻找到能容纳它的最小空闲区链表，并取下第一块进行分配即可。另外该算法在进行空闲分区分配时，不会对任何分区产生分割，所以能保留大的分区，满足对大空间的需求，也不会产生内存碎片。该算法的缺点是在分区归还主存时算法复杂，系统开销较大。此外，该算法在分配空闲分区时是以进程为单位，一个分区只属于一个进程，因此在为进程所分配的一个分区中，或多或少地存在一定的浪费。空闲分区划分越细，浪费则越严重，整体上会造成可观的存储空间浪费，这

是典型的以空间换时间的作法。

7.10 本章小结

本章主要介绍了 `hello` 储存的地址空间、线性地址到物理地址的页式和段式变换、TLB 与四级页表支持下的 VA 到 PA 的变换、三级 cache 下支持的物理内存访问等。还描述了 `hello` 进程 `fork` 和 `execve` 的映射过程。除此之外还有缺页故障、缺页中断处理的方法、并且详细介绍了动态储存分配管理的算法等。

(第 7 章 2 分)

第 8 章 hello 的 IO 管理

8.1 Linux 的 IO 设备管理方法

一个 Linux 文件就是一个 m 个字节的序列，所有的 I/O 设备（例如网络、磁盘和终端）都被模型化为文件，而所有的输入和输出都被当作对相应文件的读和写来执行。这个设备映射为文件的方式，允许 Linux 内核引出一个简单、低级的应用接口，称为 Unix I/O，这使得输入和输出都能以一种统一且一致的方式来执行。

8.2 简述 Unix IO 接口及其函数

Unix I/O 接口统一操作：

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。

2) Shell 创建的每个进程都有三个打开的文件：标准输入，标准输出，标准错误。

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`，显式地将改变当前文件位置 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 $k \geq m$ 时，触发 EOF。类似一个写操作就是从内存中复制 $n>0$ 个字节到一个文件，从当前文件位置 k 开始，然后更新 k 。

5) 关闭文件，内核释放文件打开时创建的数据结构，并将这个描述符恢复到可用的描述符池中去。

Unix I/O 函数：

打开文件：`int open(char *filename, int flags, mode_t mode)`

关闭文件： `int close(int fd);`

读文件： `ssize_t read(int fd, void *buf, size_t n);`

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

8.3 printf 的实现分析

首先观察 printf 函数：

```
static int printf(const char *fmt, ...)
{
    va_list args;

    int i;

    va_start(args, fmt);

    write(1, printbuf, i=vsprintf(printbuf, fmt, args));

    va_end(args);

    return i;
}
```

接下来是 vsprintf 函数：

`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);

}
```

Write;

```
mov eax, _NR_write
```

```
mov ebx, [esp + 4]
```

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

```
int INT_VECTOR_SYS_CALL
```

最后是 `sys_call` 函数:

```
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
```

其中: `va_start` 的作用是取到 `fmt` 中的第一个参数的地址, 接下来的 `write` 来自 Unix I/O, 而其中的 `vsprintf` 则是用来格式化的函数。这个函数的返回值是要打印出的字符串的长度, 也就是 `write` 函数中的 `i`。该函数会将 `printbuf` 根据 `fmt` 格式化字符和相应的参数进行格式化, 产生格式化的输出, 从而 `write` 能够打印。`syscall` 将字符串中的字节“Hello 1170300424 谢卓芯”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。于是我们的打印字符串“Hello 1170300424 谢卓芯”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

本章主要介绍了 Linux 的 IO 设备管理方法、向我们简析了 Unix IO 接口及其各种函数，最后又对 `printf` 函数和 `getchar` 函数进行了分析。

(第 8 章 1 分)

结论

通过完成这次大作业，可以看出 `hello` 的一生主要经历了以下几个阶段：

- 1) 预处理
- 2) 编译
- 3) 汇编
- 4) 链接
- 5) 运行
- 6) `fork` 子进程
- 7) 加载 `execve`
- 8) 指令执行
- 9) 终止回收

所以说，一个看似非常简单，似乎经常会被我们所忽略掉的 `hello` 程序，它所经历的 P2P、O2O 的过程，其实是非常复杂的，我们可以通过探寻 `hello` 的一生来更加深入的了解计算机系统，对其产生更加深刻的认知。

无论是多么简单的一个程序，它都需要经过一系列复杂的编译器、操作系统、硬件等才能实现。程序的运行与计算机系统多方面的协调工作密不可分。一条简单的指令可能需要成千上万条底层步骤，无论是内部处理还是输入输出。

（结论 0 分，缺失 -1 分，根据内容酌情加分）

附件

hello.i: hello.c 预编译的结果，用于研究预编译的作用以及进行编译器的下一步编译操作

hello.s: hello.i 编译后的结果，用于研究汇编语言以及编译器的汇编操作，可以与 hello.c 对应，分析底层的实现

hello.o: hello.s 汇编后的结果，可重定位目标程序，没有经过链接，用于链接器或编译器链接生成最终可执行程序

hello: 可执行文件

hello.d: 对 hello 的反汇编代码

hello1.d: 对 hello.o 反汇编得到

(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] 百度百科：动态储存分配.<https://baike.baidu.com/item/%E5%8A%A8%E6%80%81%E5%AD%98%E5%82%A8%E5%88%86%E9%85%8D/21306011?fr=aladdin>
- [2] 兰德尔 E.布莱恩特.深入理解计算机系统（第三版）.机械工业出版社.2017
- [3] linux 内核缺页中断处理: <https://blog.csdn.net/a7980718/article/details/80895302>
- [4] 百度百科：页式管理.<https://baike.baidu.com/item/%E9%A1%B5%E5%BC%8F%E7%AE%A1%E7%90%86>
- [5] 百度百科：段式储存管理.<https://baike.baidu.com/item/%E6%AE%B5%E5%BC%8F%E5%AD%98%E5%82%A8%E7%AE%A1%E7%90%86?fromtitle=%E6%AE%B5%E5%BC%8F%E7%AE%A1%E7%90%86&fromid=15988667>
- [6] 百度百科：逻辑地址，线性地址，物理地址。
- [7] 进程的概念和特征: <https://blog.csdn.net/u010530573/article/details/37812369>
- [8] printf 函数实现的深入剖析: https://blog.csdn.net/zhengqijun_/article/details/72454714

(参考文献 0 分，缺失 -1 分)