

哈爾濱工業大學

计算机系统

大作业

题 目 程序人生-Hello's P2P

专 业 计算机类

学 号 1170300616

班 级 1703006

学 生 段晨婕

指 导 教 师 吴锐

计算机科学与技术学院

2018 年 12 月

摘 要

本文将通过跟踪 hello 程序的生命周期来开始对系统的学习——从它被程序员创建开始，到在系统上运行，输出简单的消息，然后终止。本文将逐步围绕预处理、编译、汇编、链接、进程、存储、IO 管理展开，通过详细介绍各个过程的具体状态和操作，将计算机系统各个组成部分的工作有机地统一起来，达到搭建知识体系、将知识融会贯通的目标。

关键词：预处理；编译；汇编；链接；进程；存储；IO 管理。

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 | - 5 - |
| 1.4 本章小结 | - 5 - |
| 第 2 章 预处理 | - 6 - |
| 2.1 预处理的概念与作用 | - 6 - |
| 2.2 在 UBUNTU 下预处理的命令 | - 6 - |
| 2.3 HELLO 的预处理结果解析 | - 7 - |
| 2.4 本章小结 | - 7 - |
| 第 3 章 编译 | - 8 - |
| 3.1 编译的概念与作用 | - 8 - |
| 3.2 在 UBUNTU 下编译的命令 | - 8 - |
| 3.3 HELLO 的编译结果解析 | - 8 - |
| 3.4 本章小结 | - 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 | - 15 - |
| 第 5 章 链接 | - 16 - |
| 5.1 链接的概念与作用 | - 16 - |
| 5.2 在 UBUNTU 下链接的命令 | - 16 - |
| 5.3 可执行目标文件 HELLO 的格式 | - 16 - |
| 5.4 HELLO 的虚拟地址空间 | - 18 - |
| 5.5 链接的重定位过程分析 | - 19 - |
| 5.6 HELLO 的执行流程 | - 20 - |
| 5.7 HELLO 的动态链接分析 | - 20 - |
| 5.8 本章小结 | - 20 - |
| 第 6 章 HELLO 进程管理 | - 22 - |
| 6.1 进程的概念与作用 | - 22 - |
| 6.2 简述壳 SHELL-BASH 的作用与处理流程 | - 22 - |
| 6.3 HELLO 的 FORK 进程创建过程 | - 22 - |

| | |
|-------------------------------------|---------------|
| 6.4 HELLO 的 EXECVE 过程 | - 22 - |
| 6.5 HELLO 的进程执行 | - 23 - |
| 6.6 HELLO 的异常与信号处理 | - 23 - |
| 6.7 本章小结 | - 25 - |
| 第 7 章 HELLO 的存储管理 | - 26 - |
| 7.1 HELLO 的存储器地址空间 | - 26 - |
| 7.2 INTEL 逻辑地址到线性地址的变换-段式管理 | - 26 - |
| 7.3 HELLO 的线性地址到物理地址的变换-页式管理 | - 26 - |
| 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 | - 27 - |
| 7.6 HELLO 进程 FORK 时的内存映射 | - 28 - |
| 7.7 HELLO 进程 EXECVE 时的内存映射 | - 28 - |
| 7.8 缺页故障与缺页中断处理 | - 28 - |
| 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 | - 31 - |
| 8.5 本章小结 | - 31 - |
| 结论 | - 32 - |
| 附件 | - 33 - |
| 参考文献 | - 34 - |

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。

Hello 的 p2p:

hello 程序的生命周期是从一个高级 C 语言程序（program）开始的，因为这种形式能够被人读懂。然而，为了在系统上运行 hello.c 程序，每条 C 语句都必须被其他程序转化成一系列的低级机器语言指令。然后这些指令按照一种称为可执行目标程序的格式打好包，并以二进制磁盘文件的形式存放起来。目标程序也称为可执行目标文件。

从源文件到目标文件的转化是由编译器驱动程序完成的，整个过程可分为四个阶段，执行这四个阶段的程序（预处理器、编译器、汇编器和链接器）一起构成了编译系统。

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

这就是 Hello 的 p2p 过程。

Hello 的 020:

在程序的执行过程中会使用到内存中的数据。一个新进程被创建时，内核为新进程创建了各种数据结构，并分配给它一个唯一的 PID，从而给这个新进程创建一个虚拟内存空间。数据通过各级存储，包括磁盘、主存、Cache 等，并使用 TLB、4 级页表等辅助存储，实现访存的加速。在这个过程中还涉及操作系统的信号处理，而 IO 管理与信号处理通过软硬结合，完成程序从键盘、主板、显卡，再到屏幕的工作。当进程执行结束后，操作系统进行进程回收。这就是 Hello 的 020 过程。

1.2 环境与工具

列出你为编写本论文，折腾 Hello 的整个过程中，使用的软硬件环境，以及开发与调试工具。

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

开发工具：gcc；gdb；objdump；

1.3 中间结果

| 中间结果文件名 | 文件作用 |
|---------|----------------|
| hello.i | 预处理后得到的预处理文件 |
| hello.s | 编译后得到的汇编程序 |
| hello.o | 汇编后得到的可重定位目标程序 |
| hello | 链接后得到的可执行目标 |

1.4 本章小结

Hello.c 程序从编写到预处理、编译、汇编、链接再到执行，体现了计算机系统各部分的协同合作。

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。

作用：

- ①删除”#define”并展开所定义的宏。
- ②处理所有条件预编译指令，如”#if”、”#ifdef”、”#endif”等。
- ③插入头文件到”#include”处，可以递归方式处理。
- ④删除所有的注释”//”和”/* */”。
- ⑤添加行号和文件名标识，以便编译时编译器产生调试用的行号信息。
- ⑥保留所有#pragma 编译指令（编译器要用）。

2.2 在 Ubuntu 下预处理的命令

命令：gcc -E hello.c -o hello.i

```
dcj1170300616@ubuntu:~/HITICS-2018-Project$ gcc -E hello.c -o hello.i
dcj1170300616@ubuntu:~/HITICS-2018-Project$
```

图 2-1 预处理命令

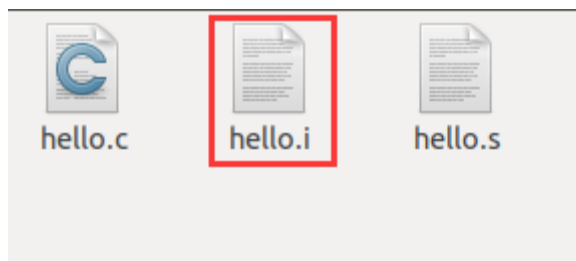
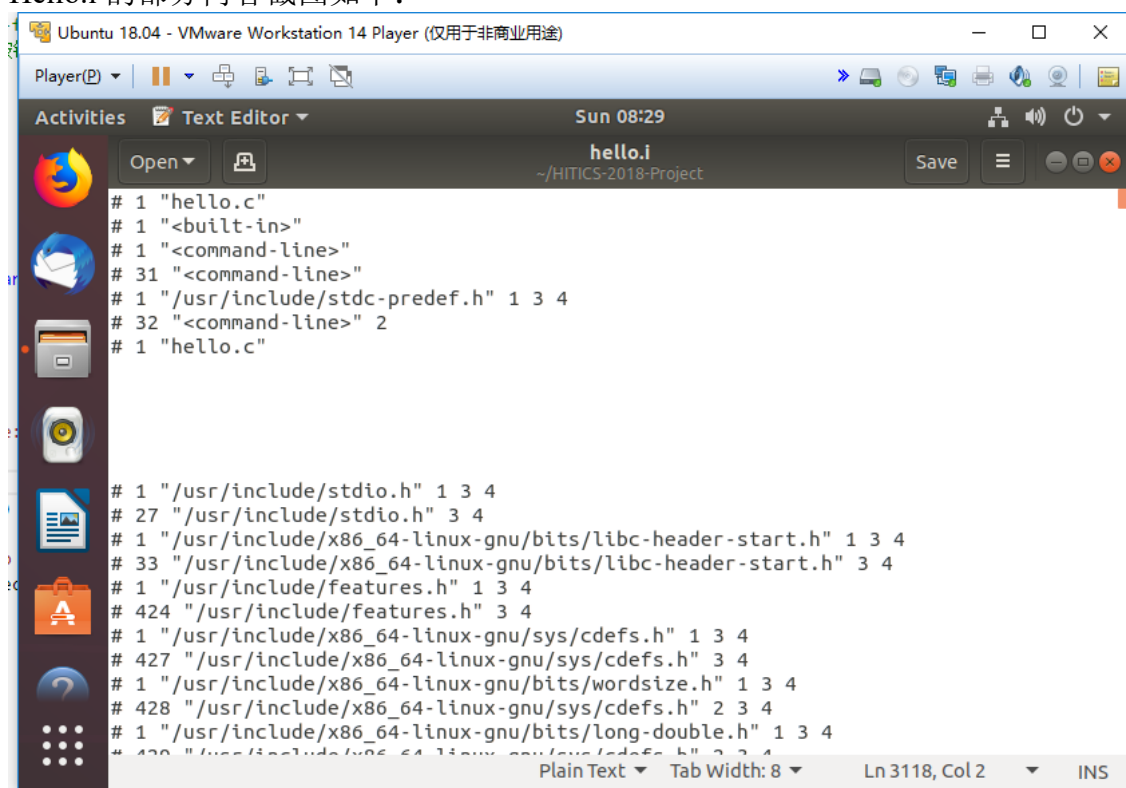


图 2-2 生成 hello.i 文件

2.3 Hello 的预处理结果解析

Hello.i 的部分内容截图如下：



```
# 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
```

图 2-3 hello.i 内容截图（部分）

经过预处理之后，hello.c 文件转化为 hello.i 文件。原文件中的宏进行了宏展开，头文件中的内容被包含进该文件中。打开该文件可以发现，文件长度变为 3118 行。文件的内容增加，且仍为可以阅读的 C 语言程序文本文件。

2.4 本章小结

预处理阶段根据以字符#开头的命令，修改原始的 C 程序，得到了修改了的源程序。

第 3 章 编译

3.1 编译的概念与作用

概念：编译器（cc1）将文本文件 `hello.i` 翻译成包含一个汇编语言程序的文本文件 `hello.s` 的过程。

作用：为接下来的汇编阶段作准备。

3.2 在 Ubuntu 下编译的命令

```
dcj1170300616@ubuntu:~/HITICS-2018-Project$ gcc -S hello.i -o hello.s
```

图 3-1 编译命令

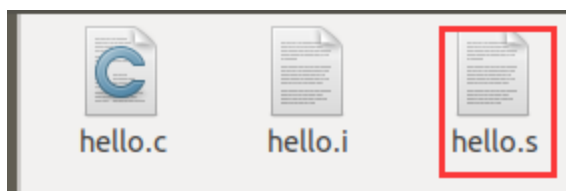


图 3-2 生成 `hello.s` 文件

3.3 Hello 的编译结果解析

3.3.1 全局变量

```
int sleepsecs=2.5;
```

图 3-3 C 语言程序里的全局变量

```
.data
.align 4
.type    sleepsecs, @object
.size    sleepsecs, 4
sleepsecs:
.long    2
```

图 3-4 `hello.s` 文件里的全局变量表示

在 `hello.c` 中，包含一个全局变量 `int sleepsecs = 2.5`，经过编译后，`sleepsecs` 被存放在 `.data` 节中。由于 `sleepsecs` 被定义为 `int` 型，所以赋值为 2.5 后，会进行隐式的类型转化，变为 2。

3.3.2 常量

```

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);
}

```

图 3-5 C 语言程序里的字符串常量

```

.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

```

图 3-6 hello.s 中的字符串常量

printf 语句中的格式串等字符串常量被放在.rodata 节中。

3.3.3 函数操作

```

int main(int argc, char *argv[])

```

图 3-7 C 语言程序中主函数参数

```

subq    $32, %rsp
movl    %edi, -20(%rbp)
movq    %rsi, -32(%rbp)

```

图 3-8 hello.s 中主函数参数的处理

主函数的参数部分给出了 int argc 和 char *argv[] 两个参数。在汇编代码中，分别将其存放在内存中地址为(%rbp-20)和(%rbp-32)处。

```

getchar();
return 0;

```

图 3-9 C 语言程序中的函数调用

```

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

```

图 3-10 hello.s 中的函数调用

在汇编代码中函数调用是通过 call 语句实现的。

3.3.4 控制转移

```
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);
}
```

图 3-11 C 语言程序中的控制转移

```
    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
    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 3-12 hello.s 中的控制转移

从汇编代码可以看出，程序的控制转移是通过跳转语句实现的。

3.4 本章小结

本阶段完成了对 `hello.i` 的编译工作。使用 `Ubuntu` 下的编译指令可以将其转换为 `.s` 汇编语言文件。此外，本章通过与源文件 `C` 程序代码进行比较，完成了对汇编代码的解析工作。完成该阶段转换后，可以进行下一阶段的汇编处理。

第 4 章 汇编

4.1 汇编的概念与作用

概念：汇编器（as）将 hello.s 翻译成机器语言指令，把这些指令打包成一种叫可重定位目标程序的格式，并将结果保存在二进制目标文件 hello.o 中。

作用：生成可重定位目标程序，为下一阶段的链接作准备。

4.2 在 Ubuntu 下汇编的命令

```
dcj1170300616@ubuntu:~/HITICS-2018-Project$ gcc hello.s -c -o hello.o
```

图 4-1 汇编命令

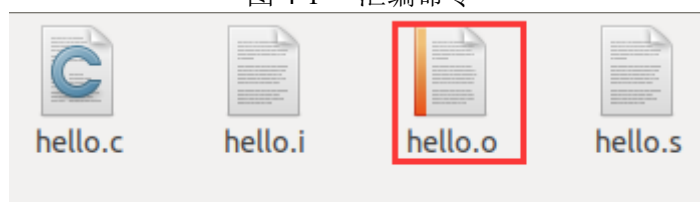


图 4-2 生成 hello.o 文件

4.3 可重定位目标 elf 格式

分析 hello.o 的 ELF 格式，用 readelf 等列出其各节的基本信息，特别是重定位项目分析。

```
dcj1170300616@ubuntu:~/HITICS-2018-Project$ readelf -h hello.o
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                                ELF64
  Data:                                      2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                UNIX - System V
  ABI Version:                           0
  Type:                                  REL (Relocatable file)
  Machine:                               Advanced Micro Devices X86-64
  Version:                               0x1
  Entry point address:                   0x0
  Start of program headers:              0 (bytes into file)
  Start of section headers:              1152 (bytes into file)
  Flags:                                  0x0
  Size of this header:                   64 (bytes)
  Size of program headers:               0 (bytes)
  Number of program headers:              0
  Size of section headers:               64 (bytes)
  Number of section headers:              13
  Section header string table index:     12
```

图 4-3 hello.o 的 ELF 头

```
dcj1170300616@ubuntu:~/HITICS-2018-Project$ readelf -S hello.o
There are 13 section headers, starting at offset 0x480:

Section Headers:
 [Nr] Name              Type              Address            Offset
     Size            EntSize          Flags   Link  Info  Align
 [ 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   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
     00000000000000198 0000000000000018          11    9    8
[11] .strtab            STRTAB            0000000000000000   000002f0
     000000000000004d 0000000000000000          0    0    1
[12] .shstrtab          STRTAB            0000000000000000   00000418
     0000000000000061 0000000000000000          0    0    1
```

图 4-4 hello.o 的节头部表

根据图 4-4 可以得到各节的基本信息。由于是可重定位目标文件，所以每个节的起始地址都从 0 开始，用于重定位。

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。

说明机器语言的构成，与汇编语言的映射关系。特别是机器语言中的操作数与汇编语言不一致，特别是分支转移函数调用等。

```

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
    |             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
    |             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 00 mov     0x0(%rip),%eax
    |             60: R_X86_64_PC32 sleepsecs-0x4

```

图 4-5 hello.o 的反汇编结果

将该反汇编结果与第 3 章的 hello.s 的主函数部分进行对照，可以发现其主要流程没有不同，只是对栈的使用有所差别。

机器语言程序的是二进制的机器指令序列集合，是纯粹的二进制数据表示的语言，是电脑可以真正识别的语言。机器指令由操作码和操作数组成。机器语言与汇编语言具有一一对应的映射关系。

机器语言中的操作数与汇编语言不一致主要体现在分支转移函数的调用，汇编语言中函数调用需要通过链接时重定位才能确定地址，故 call 地址后为占位符（4 个字节的 0），指向的是下一条指令的地址；机器语言则根据重定位类型给

出调用函数的地址。

4.5 本章小结

本阶段完成了对 `hello.s` 的汇编工作。使用 Ubuntu 下的汇编指令可以将其转换为 `.o` 可重定位目标文件。此外，本章通过将 `.o` 文件反汇编结果与 `.s` 汇编程序代码进行比较，了解了二者之间的差别。完成该阶段转换后，可以进行下一阶段的链接工作。

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为一个单一文件的过程，这个文件可被加载（复制）到内存并执行。

作用：生成可执行目标文件，可以被加载到内存中，由系统执行。

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`

截图：

```
dcj1170300616@ubuntu:~/HITICS-2018-Project$ 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
```

图 5-1 链接命令

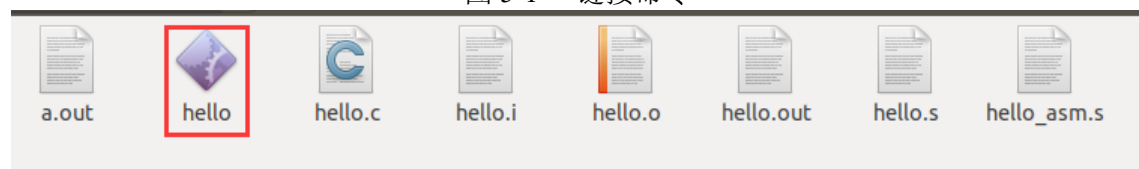


图 5-2 生成可执行目标程序 hello

5.3 可执行目标文件 hello 的格式

分析 hello 的 ELF 格式，用 `readelf` 等列出其各段的基本信息，包括各段的起始地址，大小等信息。

```
dcj1170300616@ubuntu:~/HITICS-2018-Project$ readelf -h hello
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                                ELF64
  Data:                                      2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                UNIX - System V
  ABI Version:                           0
  Type:                                  EXEC (Executable file)
  Machine:                               Advanced Micro Devices X86-64
  Version:                               0x1
  Entry point address:                   0x400500
  Start of program headers:              64 (bytes into file)
  Start of section headers:              5928 (bytes into file)
  Flags:                                  0x0
  Size of this header:                    64 (bytes)
  Size of program headers:                56 (bytes)
  Number of program headers:              8
  Size of section headers:                64 (bytes)
  Number of section headers:              25
  Section header string table index:      24
```

图 5-3 hello 的 ELF 头

```
dcj1170300616@ubuntu:~/HITICS-2018-Project$ readelf -S hello
There are 25 section headers, starting at offset 0x1728:

Section Headers:
 [Nr] Name              Type              Address            Offset
      Size            EntSize          Flags  Link  Info  Align
 [ 0]                      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             DYSYM             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
      0000000000000020 0000000000000000    A      6    1    8
 [ 9] .rela.dyn           RELA              00000000004003e0  000003e0
      0000000000000030 0000000000000018    A      5    0    8
[10] .rela.plt           RELA              0000000000400410  00000410
      0000000000000078 0000000000000018    AI     5    19    8
[11] .init               PROGBITS          0000000000400488  00000488
      0000000000000017 0000000000000000    AX     0    0    4
[12] .plt                PROGBITS          00000000004004a0  000004a0
      0000000000000060 0000000000000010    AX     0    0   16
```

图 5-4 hello 的节头部表 (1)

```
[13] .text               PROGBITS          0000000000400500  00000500
      0000000000000132 0000000000000000    AX     0    0   16
[14] .fini               PROGBITS          0000000000400634  00000634
      0000000000000009 0000000000000000    AX     0    0    4
[15] .rodata             PROGBITS          0000000000400640  00000640
      000000000000002f 0000000000000000    A      0    0    4
[16] .eh_frame           PROGBITS          0000000000400670  00000670
      00000000000000fc 0000000000000000    A      0    0    8
[17] .dynamic            DYNAMIC           0000000000600e50  00000e50
      00000000000001a0 0000000000000010    WA     6    0    8
[18] .got                PROGBITS          0000000000600ff0  00000ff0
      0000000000000010 0000000000000008    WA     0    0    8
[19] .got.plt            PROGBITS          0000000000601000  00001000
      0000000000000040 0000000000000008    WA     0    0    8
[20] .data               PROGBITS          0000000000601040  00001040
      0000000000000008 0000000000000000    WA     0    0    4
[21] .comment            PROGBITS          0000000000000000  00001048
      000000000000002a 0000000000000001    MS     0    0    1
[22] .symtab             SYMTAB            0000000000000000  00001078
      0000000000000498 0000000000000018          23   28    8
[23] .strtab             STRTAB            0000000000000000  00001510
      0000000000000150 0000000000000000          0    0    1
[24] .shstrtab           STRTAB            0000000000000000  00001660
      00000000000000c5 0000000000000000          0    0    1
```

图 5-5 hello 的节头部表 (2)

```
dcj1170300616@ubuntu:~/HITICS-2018-Project$ readelf -x .text hello

Hex dump of section '.text':
0x00400500 31ed4989 d15e4889 e24883e4 f0505449 1.I..^H..H...PTI
0x00400510 c7c03006 400048c7 c1c00540 0048c7c7 ..0.@.H....@.H..
0x00400520 32054000 ff15c60a 2000f40f 1f440000 2.@.....D..
0x00400530 f3c35548 89e54883 ec20897d ec488975 ..UH..H..}.H.u
0x00400540 e0837dec 03741648 8d3df600 0000e85d ..}.t.H.=....]
0x00400550 ffffffff 01000000 e883ffff ffc745fc .....E.
0x00400560 00000000 eb3b488b 45e04883 c010488b .....;H.E.H..H.
0x00400570 10488b45 e04883c0 08488b00 4889c648 .H.E.H..H..H.H
0x00400580 8d3ddc00 0000b800 000000e8 30ffffff .=.....0...
0x00400590 8b05ae0a 200089c7 e853ffff ff8345fc ....S....E.
0x004005a0 01837dfc 097ebfe8 24ffffff b8000000 ..}..~..$......
0x004005b0 00c9c366 2e0f1f84 00000000 000f1f00 ...f.....
0x004005c0 41574156 4989d741 5541544c 8d257e08 AWAVI..AUATL.%~.
0x004005d0 20005548 8d2d7608 20005341 89fd4989 .UH.-v..SA..I.
0x004005e0 f64c29e5 4883ec08 48c1fd03 e897feff .L).H..H.....
0x004005f0 ff4885ed 742031db 0f1f8400 00000000 .H..t 1.....
0x00400600 4c89fa4c 89f64489 ef41ff14 dc4883c3 L..L..D..A...H..
0x00400610 014839dd 75ea4883 c4085b5d 415c415d .H9.u.H...[A\A]
0x00400620 415e415f c390662e 0f1f8400 00000000 A^A_..f.....
0x00400630 f3c3 ..
```

图 5-6 readelf 查看.text 节具体信息

根据图 5-4 与图 5-5 可以得到各节的基本信息。由于是可执行目标文件，所以每个节的起始地址都不相同，它们的起始地址分别对应着装载到虚拟内存中的虚拟地址。

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各段信息，并与 5.3 对照分析说明。

| Data Dump | | | |
|---------------------------------------|---|-------------------|--|
| 0x0000000000400000-0x0000000000401000 | | | |
| 00000000:004004f0 | ff 25 42 0b 20 00 68 04 00 00 00 e9 a0 ff ff ff | 0%B .h....+ 000 | |
| 00000000:00400500 | 31 ed 49 89 d1 5e 48 89 e2 48 83 e4 f0 50 54 49 | 1.I..^H..H. PTI | |
| 00000000:00400510 | c7 c0 30 06 40 00 48 c7 c1 c0 05 40 00 48 c7 c7 | 000.@.H000.@.H000 | |
| 00000000:00400520 | 32 05 40 00 ff 15 c6 0a 20 00 f4 0f 1f 44 00 00 | 2.@.0.} .D.. | |
| 00000000:00400530 | f3 c3 55 48 89 e5 48 83 ec 20 89 7d ec 48 89 75 | UH.H..}.H.u | |
| 00000000:00400540 | e0 83 7d ec 03 74 16 48 8d 3d f6 00 00 00 e8 5d | c.}.t.H.= ...] | |
| 00000000:00400550 | ff ff ff bf 01 00 00 00 e8 83 ff ff ff c7 45 fc | 0000... .000E | |
| 00000000:00400560 | 00 00 00 00 eb 3b 48 8b 45 e0 48 83 c0 10 48 8b |;H.E.H..H. | |
| 00000000:00400570 | 10 48 8b 45 e0 48 83 c0 08 48 8b 00 48 89 c6 48 | .H.E.H..H..H.H | |
| 00000000:00400580 | 8d 3d dc 00 00 00 b8 00 00 00 00 e8 30 ff ff ff | . = ...x... 0000 | |
| 00000000:00400590 | 8b 05 ae 0a 20 00 89 c7 e8 53 ff ff ff 83 45 fc | ..x... S000.E | |
| 00000000:004005a0 | 01 83 7d fc 09 7e bf e8 24 ff ff ff b8 00 00 00 | ..}..~ \$000x... | |
| 00000000:004005b0 | 00 c9 c3 66 2e 0f 1f 84 00 00 00 00 0f 1f 00 | .00f..... | |
| 00000000:004005c0 | 41 57 41 56 49 89 d7 41 55 41 54 4c 8d 25 7e 08 | AWAVI..AUATL.%~. | |
| 00000000:004005d0 | 20 00 55 48 8d 2d 76 08 20 00 53 41 89 fd 49 89 | .UH.-v..SA..I. | |
| 00000000:004005e0 | f6 4c 29 e5 48 83 ec 08 48 c1 fd 03 e8 97 fe ff | L)H..H000 .00 | |
| 00000000:004005f0 | ff 48 85 ed 74 20 31 db 0f 1f 84 00 00 00 00 00 | H..t 1 | |
| 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 | .H9uH..[A\A] | |

图 5-7 用 edb 查看本进程的虚拟地址空间

与 5.3 对比发现，使用 readelf 与 edb 查看得到的内容相同。

5.5 链接的重定位过程分析

objdump -d -r hello 分析 hello 与 hello.o 的不同，说明链接的过程。

结合 hello.o 的重定位项目，分析 hello 中对其怎么重定位的。

```
hello:      file format elf64-x86-64

Disassembly of section .init:

0000000000400488 <_init>:
 400488:  48 83 ec 08          sub    $0x8,%rsp
 40048c:  48 8b 05 65 0b 20 00  mov    0x200b65(%rip),%rax
 400493:  48 85 c0             test   %rax,%rax
 400496:  74 02              je     40049a <.plt>
 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),%rax
 4004a6:  ff 25 64 0b 20 00    jmpq   *0x200b64(%rip),%rax
 4004ac:  0f 1f 40 00          nopl   0x0(%rax)

00000000004004b0 <puts@plt>:
 4004b0:  ff 25 62 0b 20 00    jmpq   *0x200b62(%rip),%rax
 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),%rax
 4004c6:  68 01 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),%rax
```

图 5-8 hello 的反汇编码（部分）

与 hello.o 的反汇编相比，hello 的反汇编多了许多内容。在 hello.o 的反汇编中仅有 .text 节中 main 函数的反汇编代码，而 hello 的反汇编代码中新增了各个节的

内容。链接器解析重定条目时发现两个类型为 R_X86_64_PC32 的对 .rodata 的重定位（printf 中的两个字符串），.rodata 与 .text 节之间的相对距离确定，因此链接器直接修改 call 之后的值为目标地址与下一条指令的地址之差，指向相应的字符串。

5.6 hello 的执行流程

通过使用 objdump 查看反汇编代码，以及使用 gdb 单步运行，可以找出 .text 节中 main 函数前后执行的函数名称。在 main 函数之前执行的程序有：_start、__libc_start_main@plt、__libc_csu_init、_init、frame_dummy、register_tm_clones。在 main 函数之后执行的程序有：exit、cxa_thread_atexit_impl、fini。

5.7 Hello 的动态链接分析

分析 hello 程序的动态链接项目，通过 edb 调试，分析在 dl_init 前后，这些项目的内容变化。要截图标识说明。

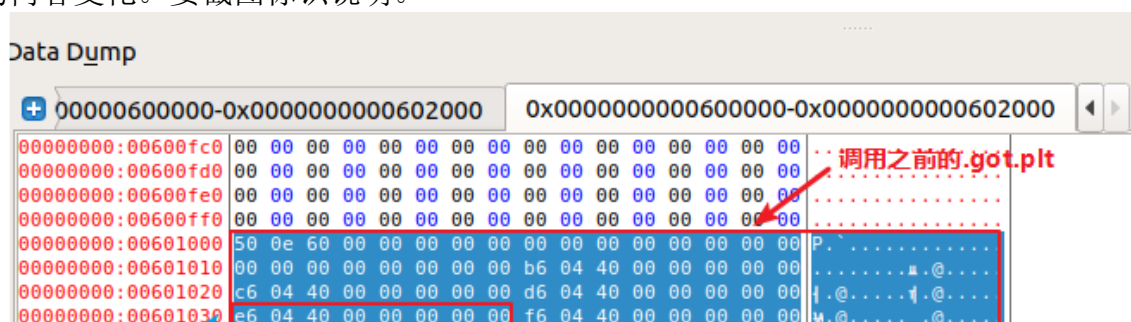


图 5-9 调用 dl_init 前的.got.plt

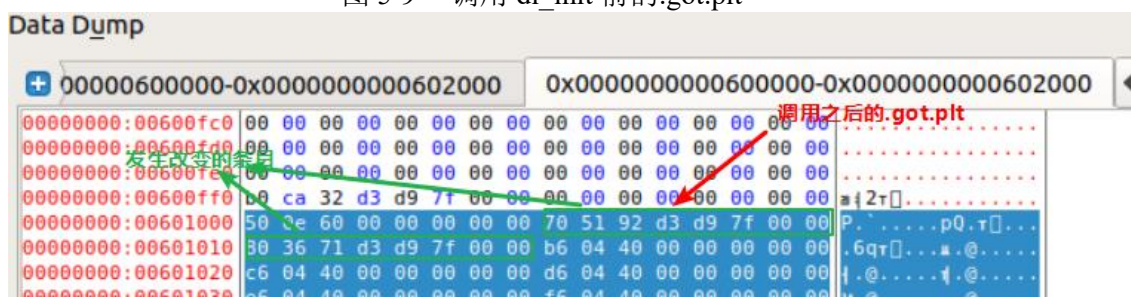


图 5-10 调用 dl_init 后的.got.plt

在 dl_init 调用之前，对于每一条 PIC 函数调用，GOT 存放的是 PLT 中函数调用指令的下一条指令地址。

在 dl_init 调用之后，如图 5-10，在 dl_init 调用之后，如图 5.4 (b)，0x601008 和 0x601010 处的两个 8B 数据分别发生改变为 0x7fd9 d3925170 和 0x7fd9 d3713680，通过重定位确定调用的函数地址。

5.8 本章小结

本阶段完成了对 hello.o 的链接工作。使用 Ubuntu 下的链接指令可以将其转换

为.out 可执行目标文件。此外，本章通过一系列细致分析，了解了链接过程中的具体细节。完成该阶段转换后，即得到了可以执行的二进制文件了。

第 6 章 hello 进程管理

6.1 进程的概念与作用

概念：进程的经典定义就是一个执行中程序的实例。

作用：系统中的每个程序都运行在某个进程的上下文中，进程给应用程序提供了两个关键抽象：

- 1) 一个独立的逻辑控制流，它提供一个假象，好像我们的程序独占地使用处理器。
- 2) 一个私有的地址空间，它提供一个假象，好像我们的程序独占地使用内存系统。

6.2 简述壳 Shell-bash 的作用与处理流程

Shell-bash 的作用：Shell-bash 是一个用 C 语言编写的程序，它是用户使用 linux 的桥梁，它提供了一个界面，用户通过这个界面访问操作系统内核的服务。

处理流程：

- 1) 从终端读入输入的命令。
- 2) 将输入字符串切分获得所有的参数。
- 3) 如果是内置命令则立即执行。
- 4) 否则调应的程序为其分配子进程并运行。
- 5) shell 应该接受键盘输入信号，对这些信号进行相应的处理。

6.3 Hello 的 fork 进程创建过程

在终端的命令行中输入 `./hello 1170300616 dcj`，运行的终端程序会对命令行进行解析，因为 `hello` 不是一个内置的 shell 命令，所以解析之后终端程序判断 `./hello` 的语义为执行当前目录下的可执行程序 `hello`，之后终端程序首先会调用 `fork` 函数创建一个新的运行的子进程，新创建的子进程几乎但不完全与父进程相同，子进程得到与父进程用户级虚拟地址相同的（但是独立的）一份副本，这就意味着，当父进程调用 `fork` 后，子进程可以读写父进程中打开的任意文件。子进程与父进程最大的区别在于它们拥有不同的 PID。

6.4 Hello 的 execve 过程

当 `fork` 后，子进程调用 `execve` 函数（传入命令行参数）在当前进程的上下文中加载并运行一个新程序即 `hello` 程序，`execve` 调用驻留在内存中的被称为启动加载器的操作系统代码来执行 `hello` 程序，加载器删除子进程现有的虚拟内存段，并创建一组新的代码、数据、堆和栈段。

6.5 Hello 的进程执行

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。

上下文：上下文就是内核重新启动一个被抢占的进程所需的状态，它由一些对象的值组成，这些对象包括通用目的寄存器、浮点寄存器、程序计数器、用户栈、状态寄存器、内核栈和各种内核数据结构，比如描述地址空间的页表、包含有关当前进程信息的进程表，以及包含进程已打开文件的信息的文件表。

进程时间片：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。

进程调度：在进程执行的某些时刻，内核可以决定抢占当前进程，并重新开始一个先前被抢占了的进程。这种决策就叫调度。

用户态与核心态的转换：运行应用程序代码的进程初始时是在用户模式中的。进程从用户模式变为内核模式的唯一方法是通过诸如中断、故障或者陷入系统调用这样的异常。

6.6 hello 的异常与信号处理

hello 执行过程中会出现哪几类异常，会产生哪些信号，又怎么处理的。

异常可以分为四类：中断、陷阱、故障和终止。一旦硬件触发了异常，剩下的工作就是由异常处理程序在软件中完成。

信号是一种更高层的软件形式的异常，它允许进程和内核中断其他进程。当一个进程捕获了一个类型为 `k` 的信号时，为调用为信号 `k` 设置的处理程序。

| 序号 | 名称 | 默认行为 | 相应事件 |
|----|-----------|------------------------------|-------------------|
| 1 | SIGHUP | 终止 | 终端线挂断 |
| 2 | SIGINT | 终止 | 来自键盘的中断 |
| 3 | SIGQUIT | 终止 | 来自键盘的退出 |
| 4 | SIGILL | 终止 | 非法指令 |
| 5 | SIGTRAP | 终止并转储内存 ^① | 跟踪陷阱 |
| 6 | SIGABRT | 终止并转储内存 ^① | 来自 abort 函数的终止信号 |
| 7 | SIGBUS | 终止 | 总线错误 |
| 8 | SIGFPE | 终止并转储内存 ^① | 浮点异常 |
| 9 | SIGKILL | 终止 ^② | 杀死程序 |
| 10 | SIGUSR1 | 终止 | 用户定义的信号 1 |
| 11 | SIGSEGV | 终止并转储内存 ^① | 无效的内存引用 (段故障) |
| 12 | SIGUSR2 | 终止 | 用户定义的信号 2 |
| 13 | SIGPIPE | 终止 | 向一个没有读用户的管道做写操作 |
| 14 | SIGALRM | 终止 | 来自 alarm 函数的定时器信号 |
| 15 | SIGTERM | 终止 | 软件终止信号 |
| 16 | SIGSTKFLT | 终止 | 协处理器上的栈故障 |
| 17 | SIGCHLD | 忽略 | 一个子进程停止或者终止 |
| 18 | SIGCONT | 忽略 | 继续进程如果该进程停止 |
| 19 | SIGSTOP | 停止直到下一个 SIGCONT ^③ | 不是来自终端的停止信号 |
| 20 | SIGTSTP | 停止直到下一个 SIGCONT | 来自终端的停止信号 |
| 21 | SIGTTIN | 停止直到下一个 SIGCONT | 后台进程从终端读 |
| 22 | SIGTTOU | 停止直到下一个 SIGCONT | 后台进程向终端写 |
| 23 | SIGURG | 忽略 | 套接字上的紧急情况 |
| 24 | SIGXCPU | 终止 | CPU 时间限制超出 |
| 25 | SIGXFSZ | 终止 | 文件大小限制超出 |
| 26 | SIGVTALRM | 终止 | 虚拟定时器期满 |
| 27 | SIGPROF | 终止 | 剖析定时器期满 |
| 28 | SIGWINCH | 忽略 | 窗口大小变化 |
| 29 | SIGIO | 终止 | 在某个描述符上可执行 I/O 操作 |
| 30 | SIGPWR | 终止 | 电源故障 |

图 6-1 linux 中的信号类别

程序运行过程中可以按键盘，如不停乱按，包括回车，Ctrl-Z, Ctrl-C 等，Ctrl-z 后可以运行 ps jobs pstree fg kill 等命令，请分别给出各命令及运行结果截屏，说明异常与信号的处理。

```
dcj1170300616@ubuntu:~/HITICS-2018-Project$ ./hello 1170300616 dcj
Hello 1170300616 dcj
Hello 1170300616 dcj
deHello 1170300616 dcj
  def fHello 1170300616 dcj
r Hello 1170300616 dcj
```

图 6-2 不停乱按

```
dcj1170300616@ubuntu:~/HITICS-2018-Project$ ps
  PID TTY          TIME CMD
 1818 pts/0        00:00:00 bash
 1886 pts/0        00:00:00 hello
 1887 pts/0        00:00:00 ps
```

图 6-3 Ctrl-z 后运行 ps 查看进程及其运行时间

```
dcj1170300616@ubuntu:~/HITICS-2018-Project$ jobs
[1]+  Stopped                  ./hello 1170300616 dcj
```

图 6-4 Ctrl-z 后运行 jobs 查看当前暂停的进程

```
dcj1170300616@ubuntu:~/HITICS-2018-Project$ fg
./hello 1170300616 dcj
Hello 1170300616 dcj
Hello 1170300616 dcj
Hello 1170300616 dcj
Hello 1170300616 dcj
```

图 6-5 Ctrl z 后输入 fg 使进程在前台执行

```

└─4*[{evolution-addre}]
evolution-calen└─evolution-calen──8*[{evolution-calen}]
└─4*[{evolution-calen}]
evolution-sourc──3*[{evolution-sourc}]
gnome-shell-cal──5*[{gnome-shell-cal}]
gnome-terminal-└─bash└─hello
└─pstree
└─3*[{gnome-terminal-}]
goa-daemon──3*[{goa-daemon}]
goa-identity-se──3*[{goa-identity-se}]
gvfs-afc-volume──3*[{gvfs-afc-volume}]
gvfs-goa-volume──2*[{gvfs-goa-volume}]
gvfs-gphoto2-vo──2*[{gvfs-gphoto2-vo}]
gvfs-mtp-volume──2*[{gvfs-mtp-volume}]
gvfs-udisks2-vo──2*[{gvfs-udisks2-vo}]
gvfsd└─gvfsd-trash──2*[{gvfsd-trash}]
└─2*[{gvfsd}]
gvfsd-fuse──5*[{gvfsd-fuse}]
ibus-portal──2*[{ibus-portal}]
systemd-journal
systemd-logind
systemd-resolve
systemd-timesyn──{systemd-timesyn}
systemd-udev
udisksd──4*[{udisksd}]
upowerd──2*[{upowerd}]
vmtoolsd──{vmtoolsd}
vmtoolsd──3*[{vmtoolsd}]
vmware-vmblock-──2*[{vmware-vmblock-}]
whoopsie──2*[{whoopsie}]
wpa supplicant
```

图 6-6 输入 pstree

```
dcj1170300616@ubuntu:~/HITICS-2018-Project$ kill 1886
dcj1170300616@ubuntu:~/HITICS-2018-Project$ fg
./hello 1170300616 dcj
Terminated
```

图 6-7 输入 kill 杀死特定进程

6.7 本章小结

本阶段通过在 hello.out 运行过程中执行各种操作，了解了与系统相关的若干概念、函数和功能。分析了在程序运行过程中，计算机硬件、软件和操作系统之间的配合和协作的方式。

第 7 章 hello 的存储管理

7.1 hello 的存储器地址空间

逻辑地址：相对于当前进程段的地址（偏移地址）。

线性地址：线性地址是逻辑地址到物理地址变换之间的中间层。程序代码会产生逻辑地址，或说是段中的偏移地址，加上相应段的基地址就生成了一个线性地址。如果启用了分页机制，那么线性地址能再经变换以产生一个物理地址。若没有启用分页机制，那么线性地址直接就是物理地址。

物理地址：CPU 通过地址总线的寻址，找到的真实的物理内存对应地址。

虚拟地址：使用虚拟寻址，CPU 通过生成一个虚拟地址来访问主存。

7.2 Intel 逻辑地址到线性地址的变换-段式管理

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

7.3 Hello 的线性地址到物理地址的变换-页式管理

CPU 的页式管理单元负责把一个线性地址最终翻译成一个物理地址。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。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，这样，整个线性地址就被划分为一个大数组，数组中共有 2^{20} 个页。这个大数组我们称之为页目录。目录中的每一目录项，就是一个地址——对应页的地址。另一类“页”我们称之为物理页，是分页单元把所有的物理内存也划分为固定长度的管理单元，它的长度一般与内存页是一一对应的。

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

Core i7 MMU 使用四级的页表将虚拟地址翻译成物理地址。36 位 VPN 被划分为四个 9 位的片，每个片被用作到一个页表的偏移量。具体结构如图：

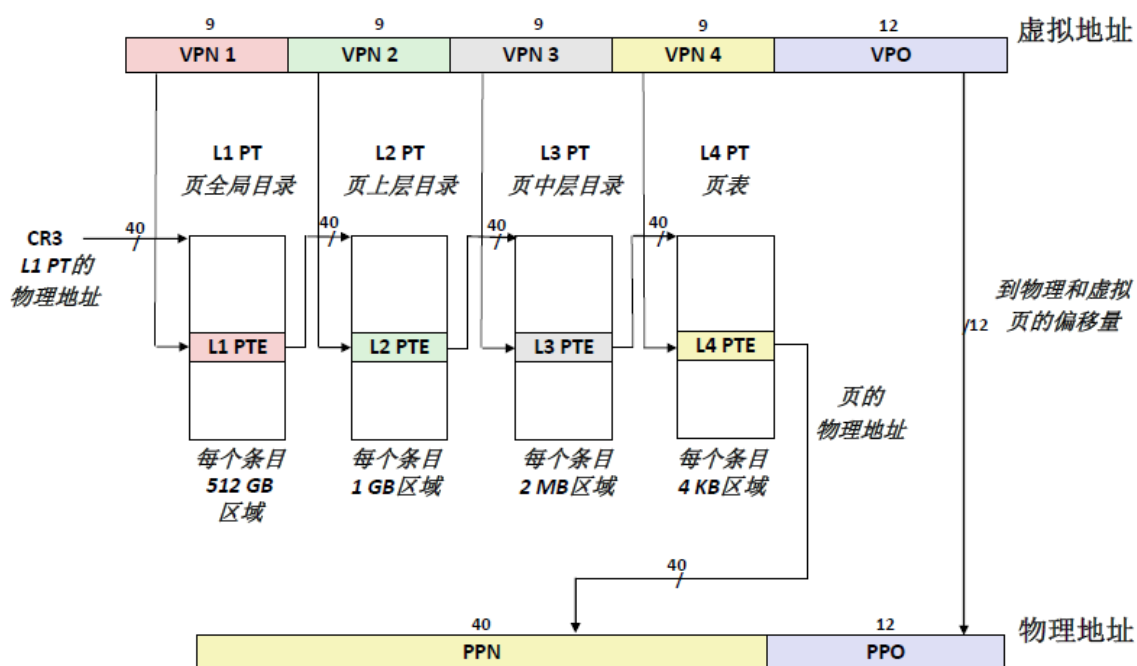


图 7-1 四级页表结构

7.5 三级 Cache 支持下的物理内存访问

首先 CPU 发出一个虚拟地址，在 TLB 中寻找，如果命中，那么将 PTE 发送给 L1Cache，否则先在页表中更新 PTE，然后再进行 L1 根据 PTE 寻找物理地址、检测是否命中的工作。具体流程如图：

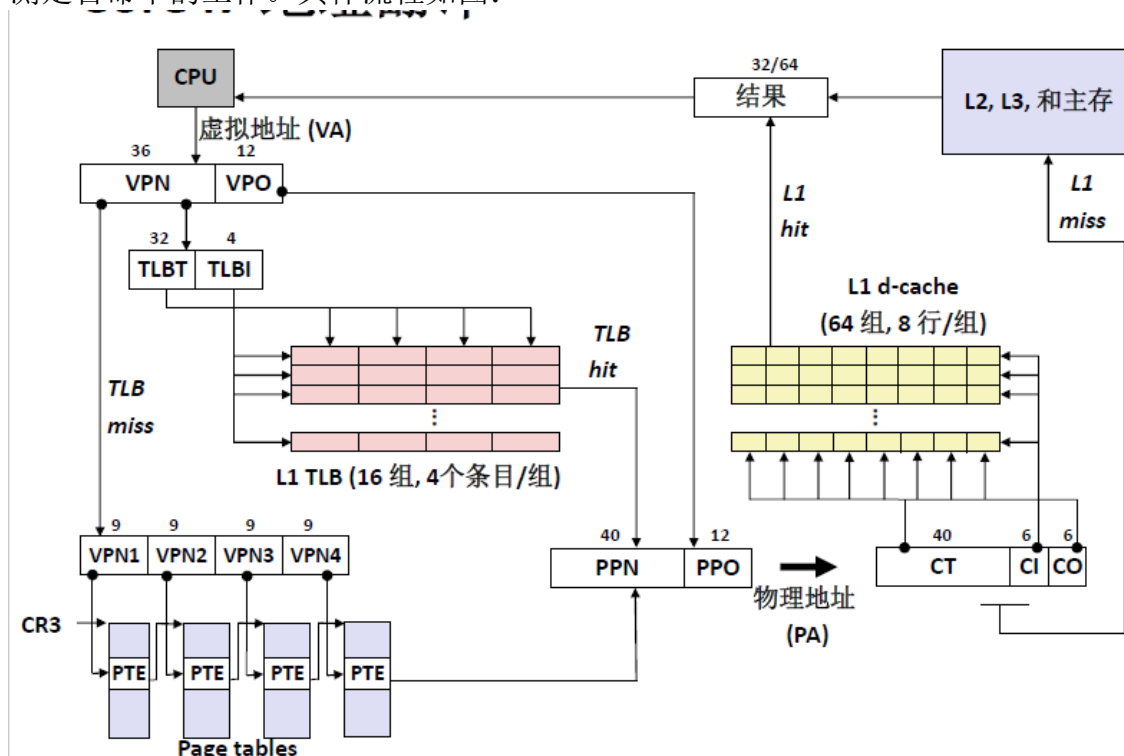


图 7-2 三级 Cache 支持下的物理内存访问

7.6 hello 进程 fork 时的内存映射

当 fork 函数被 shell 进程调用时，内核为新进程创建各种数据结构，并分配给它一个唯一的 PID，为了给这个进程创建虚拟内存，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、删除已存在的用户区域。
- 2、创建新的私有区域。
- 3、创建新的共享区域。
- 4、设置 PC，指向代码的入口点。

7.8 缺页故障与缺页中断处理

DRAM 缓存不命中称为缺页。缺页异常会调用内核中的缺页异常处理程序，该程序会选择一个牺牲页，然后将目标页加载到物理内存中。最后让导致缺页的指令重新启动，页面命中。

7.9 动态存储分配管理

Printf 会调用 malloc，请简述动态内存管理的基本方法与策略。

在程序运行时程序员使用动态内存分配器（比如 `malloc`）获得虚拟内存。动态内存分配器维护着一个进程的虚拟内存区域，称为堆。系统之间细节不同，但是不失通用性，假设堆是一个请求二进制零的区域，它紧接在未初始化的数据区域后开始，并向上生长。对于每个进程，内核维护着一个变量 `brk`，它指向堆的顶部。

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

7.10 本章小结

本章通过 hello 的内存管理，复习了与内存管理相关的重要的概念和方法。加深了

对动态内存分配的认识和了解。

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，而所有输入和输出都被当作对相应的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

Linux 提供如下 IO 接口参数：

read 和 write——最简单的读写函数；

readn 和 writen——原子性读写操作；

recvfrom 和 sendto - 增加了目标地址和地址结构长度的参数；

recv 和 send - 允许从进程到内核传递标志；

readv 和 writev - 允许指定往其中输入数据或从其中输出数据的缓冲区；

recvmsg 和 sendmsg - 结合了其他 IO 函数的所有特性，并具备接受和发送辅助数据的能力。

8.3 printf 的实现分析

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

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）。

先看 printf 函数代码：

```
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;
}
```

图 8-1 printf 函数代码

`(char*)&fmt + 4` 表示的是...中的第一个参数。C 语言中，参数压栈的方向是从右往左。也就是说，当调用 `printf` 函数的适合，先是最右边的参数入栈。`fmt` 是一个指针，这个指针指向第一个 `const` 参数 (`const char *fmt`) 中的第一个元素。`vsprintf` 的作用就是格式化。它接受确定输出格式的格式字符串 `fmt`。用格式字符串对个数变化的参数进行格式化，产生格式化输出。

8.4 getchar 的实现分析

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

`getchar` 等调用 `read` 系统函数，通过系统调用读取按键 `ascii` 码，直到接受到回车键才返回。

8.5 本章小结

本章通过介绍 `hello` 中包含的函数所对应的 `unix I/O`，大致了解了 `I/O` 接口及其工作方式，同时也了解了硬件设备的使用和管理的技术方法。

结论

用计算机系统的语言，逐条总结 **hello** 所经历的过程。

- 1、编写。**hello** 程序的生命周期是从一个高级 C 语言程序开始的。
- 2、预处理。本阶段完成了对 **hello.c** 的预处理工作。使用 **Ubuntu** 下的预处理指令可以将其转换为 **.i** 文件。完成该阶段转换后，可以进行下一阶段的汇编处理。
- 3、编译。本阶段完成了对 **hello.i** 的编译工作。使用 **Ubuntu** 下的编译指令可以将其转换为 **.s** 汇编语言文件。完成该阶段转换后，可以进行下一阶段的汇编处理。
- 4、汇编。本阶段完成了对 **hello.s** 的汇编工作。使用 **Ubuntu** 下的汇编指令可以将其转换为 **.o** 可重定位目标文件。完成该阶段转换后，可以进行下一阶段的链接工作。
- 5、链接。本阶段完成了对 **hello.o** 的链接工作。使用 **Ubuntu** 下的链接指令可以将其转换为 **.out** 可执行目标文件。完成该阶段转换后，即得到了可以执行的二进制文件了。
- 6、执行。该过程对异常和信号的处理体现了计算机系统的软硬结合、分工合作。
- 7、结束。**shell** 父进程回收子进程，内核删除为这个进程创建的所有数据结构。

你对计算机系统的设计与实现的深切感悟，你的创新理念，如新的设计与实现方法。

经过一学期的计算机系统课程的学习，我初步了解了计算机系统的设计与实现，感受到了一个简单的 **hello** 程序从编写到运行的复杂过程，同时十分惊叹于计算机系统的各类抽象，如进程概念的提出、IO 设备的模型化、高级语言对汇编语言的抽象、汇编语言对机器语言的抽象，这些抽象使得我们对概念的理解和运用变得相对简单了。这就是我对计算机系统设计与实现的感悟。

附件

| 中间结果文件名 | 文件作用 |
|---------|----------------|
| hello.i | 预处理后得到的预处理文件 |
| hello.s | 编译后得到的汇编程序 |
| hello.o | 汇编后得到的可重定位目标程序 |
| hello | 链接后得到的可执行目标 |

参考文献

- [1] 大卫 R.奥哈拉伦，兰德尔 E.布莱恩特. 深入理解计算机系统[M]. 机械工业出版社.2017.7