

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>軟件工程</u>
學	號	<u>1173710230</u>
班	級	<u>1737102</u>
學	生	<u>劉晶仟</u>
指	導	教
師		<u>吳銳</u>

計算機科學與技術學院

2018 年 12 月

摘 要

本文介绍了一个 `hello.c` 小程序从编写到执行到介绍的全过程。通过详细介绍各个过程中的具体状态和操作，将计算机系统各个组成部分的工作有机地结合统一起来，实现知识骨架的建成。本文主要包括示例程序 `hello.c` 的预处理、编译、汇编、链接、进程管理、存储管理、I/O 管理等部分的具体操作和结果。

关键词：机器级表示；内存管理；链接；异常；`hello.c`

本文链接地址：<https://github.com/sculpture12/CSAPP-HELLOWORLD>

（摘要 0 分，缺失-1 分，根据内容精彩称都酌情加分 0-1 分）

目 录

第 1 章 概述.....	- 4 -
1.1 HELLO 简介	- 4 -
1.2 环境与工具	- 4 -
1.3 中间结果	- 4 -
1.4 本章小结	- 4 -
第 2 章 预处理.....	- 5 -
2.1 预处理的概念与作用	- 5 -
2.2 在 UBUNTU 下预处理的命令	- 5 -
2.3 HELLO 的预处理结果解析	- 6 -
2.4 本章小结	- 7 -
第 3 章 编译.....	- 8 -
3.1 编译的概念与作用	- 8 -
3.2 在 UBUNTU 下编译的命令	- 8 -
3.3 HELLO 的编译结果解析	- 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	- 15 -
4.5 本章小结	- 17 -
第 5 章 链接.....	- 18 -
5.1 链接的概念与作用	- 18 -
5.2 在 UBUNTU 下链接的命令	- 18 -
5.3 可执行目标文件 HELLO 的格式	- 18 -
5.4 HELLO 的虚拟地址空间	- 21 -
5.5 链接的重定位过程分析	- 21 -
5.6 HELLO 的执行流程	- 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	- 26 -
6.5 HELLO 的进程执行	- 26 -
6.6 HELLO 的异常与信号处理	- 26 -
6.7 本章小结	- 27 -
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	- 32 -
第 8 章 HELLO 的 IO 管理	- 33 -
8.1 LINUX 的 IO 设备管理方法	- 33 -
8.2 简述 UNIX IO 接口及其函数	- 33 -
8.3 PRINTF 的实现分析	- 34 -
8.4 GETCHAR 的实现分析	- 35 -
8.5 本章小结	- 35 -
结论	- 35 -
附件	- 36 -
参考文献	- 37 -

第 1 章 概述

1.1 Hello 简介

P2P 指的是 hello.c 由程序 (program)，也就是第一个 'P' 转变为进程 (process)，也就是第二个 'p'。首先 hello.c 文件要经过预处理，编译，汇编，链接一系列操作转变为 hello.out 可执行文件。（链接时不要忘记在命令行中添加本地函数库）。之后执行此文件，shell 为其 fork，生成子进程，实现向进程 (process) 的转变。

020: 接着上面，shell 调用 execve 函数，加载进程，映射虚拟内存。进入程序入口后，程序开始载入物理内存，之后进入 main 函数执行目标代码，CPU 为 hello 分配时间片执行逻辑控制流。当程序运行结束后，内核删除相关数据。

1.2 环境与工具

这是我的实验配置，可以流畅运行本实验。

硬件环境: 1.8 GHz Intel Core i5; 8 GB RAM; 128G SSD

软件环境: Ubuntu 18.04 LTS

开发工具: gcc; gdb; objdump; vim; Code::blocks IDE;

1.3 中间结果

中间文件	作用
Hello.i	Hello.c 经预处理 (cpp) 得到
Hello.s	Hello.i 经编译器 (cc1) 得到汇编文件
Hello.o	Hello.s 经汇编器 (as) 得到可重定位文件
Hello	链接之后得到可执行文件

1.4 本章小结

本阶段简要介绍一下 p2p, 020 过程，以及实验的运行环境和产生的中间文件。

(第 1 章 0.5 分)

第 2 章 预处理

2.1 预处理的概念与作用

预处理 (cpp) 根据以字符#开头的命令, 修改原始的 C 程序, 就拿下图为例 hello.c 中第 6 行 `#include <stdio.h>` 命令告诉预处理器读取系统文件 `stdio.h` 内容, 并把它插入程序文本中, 结果得到另一个 C 程序, 通常以 .i 为文件扩展名。

```
1 // 大作业的 hello.c 程序
2 // gcc -m64 -no-pie -fno-PIC hello.c -o hello
3 // 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。
4 // 可以 运行 ps jobs pstree fg 等命令
5
6 #include <stdio.h>
7 #include <unistd.h>
8 #include <stdlib.h>
9
10 int sleepsecs=2.5;
11
12 int main(int argc,char *argv[])
13 {
14     int i;
15
16     if(argc!=3)
17     {
18         printf("Usage: Hello 1173710230 刘晶仟!\n");
19         exit(1);
20     }
21     for(i=0;i<10;i++)
22     {
23         printf("Hello %s %s\n",argv[1],argv[2]);
24         sleep(sleepsecs);
25     }
26     getchar();
27     return 0;
```

图 2-1 预处理

2.2 在 Ubuntu 下预处理的命令

执行 `gcc hello.c -E -o hello.i` 命令, 生成 `hello.i` 文件。

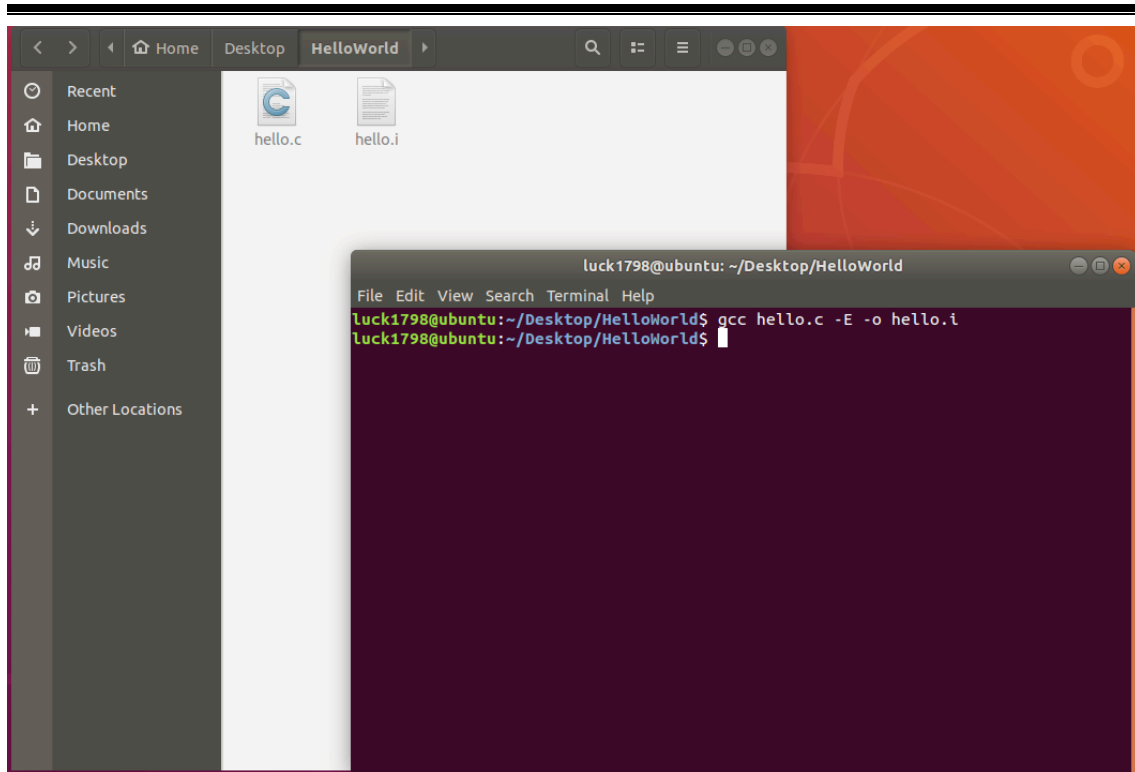


图 2-2 预处理执行

2.3 Hello 的预处理结果解析

查看生成的 hello.i 文件，执行命令 `vim hello.i`，

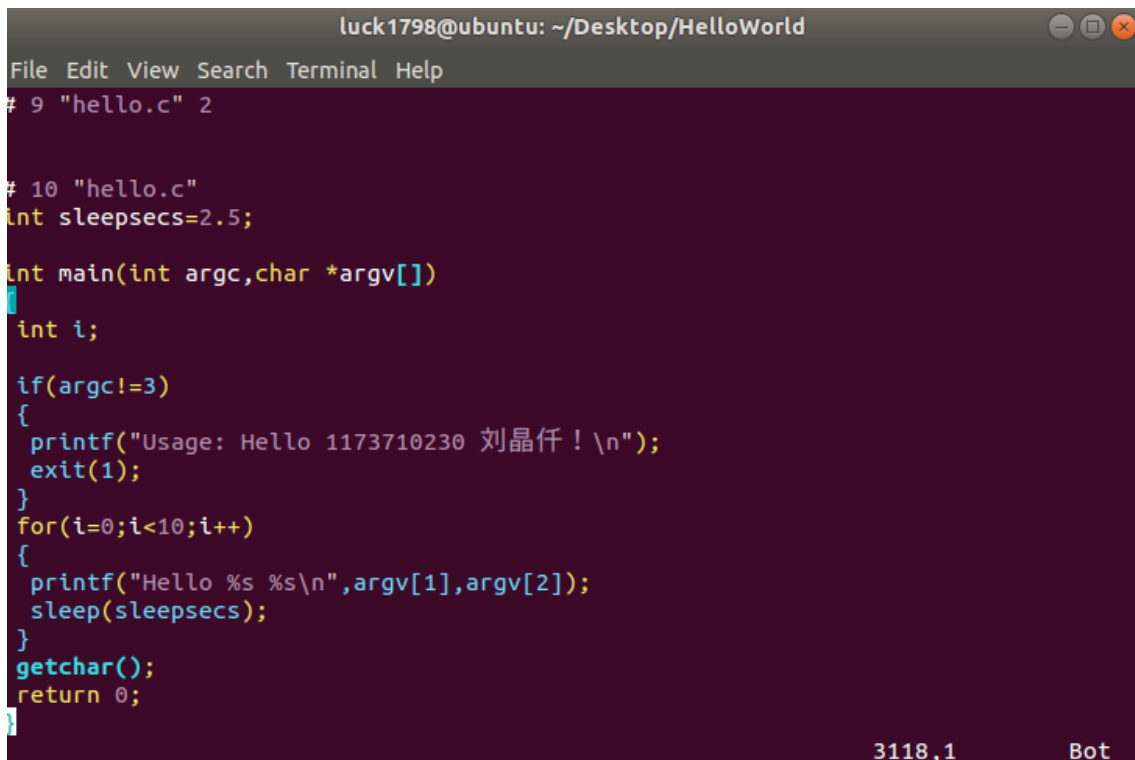


图 2-3 hello.i

WOW,3118 行，main 函数出现在第 3102 行。

经过预处理，对原文件中的宏进行展开，并把头文件内容写入 hello.i 文件。致使代码长度飙升。

2.4 本章小结

这一阶段经过预处理器实现 hello.c 向 hello.i 的转化，新生成的 hello.i 代码长度增加，仍具有可读性。

(第 2 章 0.5 分)

第 3 章 编译

3.1 编译的概念与作用

编译器（cc1）将由高级语言编写的文本文件 `hello.i` 翻译成由汇编语言构成的文本文件 `hello.s`，它包含一个汇编语言程序。编译程序的基本功能是把源程序（高级语言）翻译成目标程序。除了基本功能之外，编译程序还具备语法检查、调试措施、修改手段、覆盖处理、目标程序优化、不同语言合用以及人机联系等重要功能。

3.2 在 Ubuntu 下编译的命令

执行 `gcc hello.i -S -o hello.s` 命令，生成 `hello.s` 文件

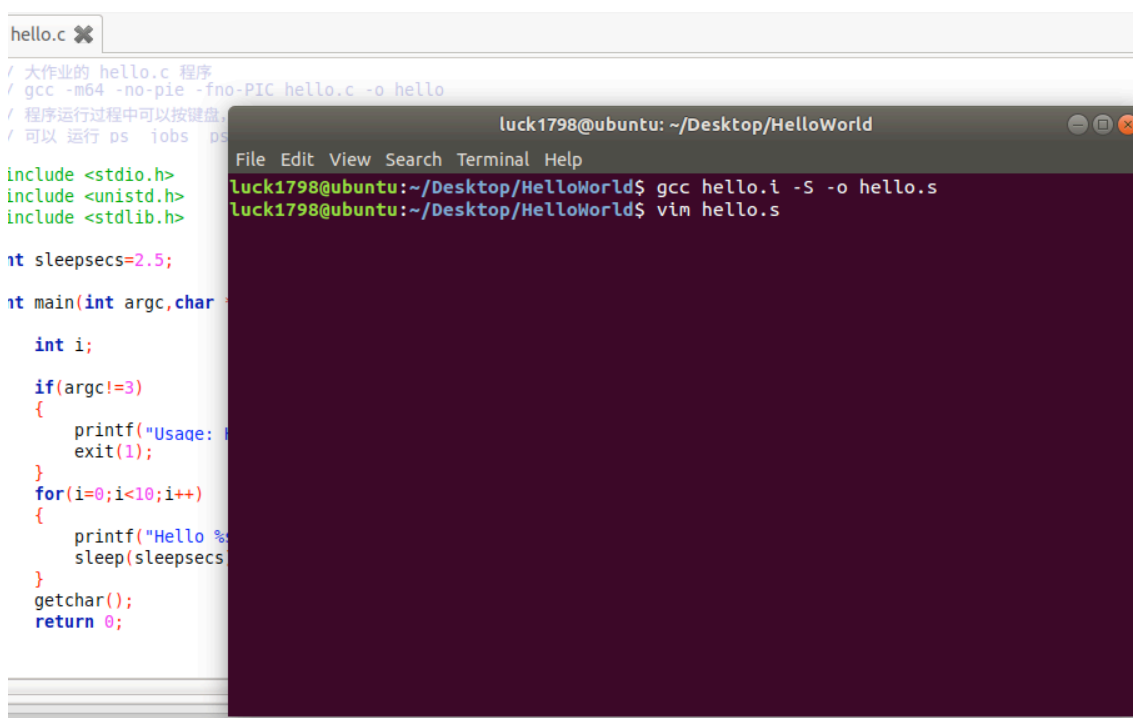


图 3-1 编译执行

3.3 Hello 的编译结果解析

3.3.1 字符串

```

.LC0:
.string "Usage: Hello 1173710230 \345\210\230\346\231\266\344\273\237\357\274\201"

```

图 3-2 hello.s 字符串

```

15
16     if(argc!=3)
17     {
18         printf("Usage: Hello 1173710230 刘晶仟!\n");
19         exit(1);
20     }

```

图 3-3 hello.c 字符串

3.3.2 全局变量与全局函数

在 hello.c 中存在全局变量(int sleepsecs=2)与全局函数(int main(int argc, char *argv[]);)。经过编译之后,全局变量 sleepsecs 被存放在.rodata 节中,全局函数 main 中使用的字符串常量也被存放在数据区。此外,全局变量 sleepsecs 被定义为 int 型,故变为 2。

```

luck1798@ubuntu: ~/Desktop/HelloWorld
File Edit View Search Terminal Help
.type    sleepsecs, @object
.size    sleepsecs, 4
sleepsecs:
    .long    2
    .section    .rodata
    .align 8
.LC0:
.string  "Usage: Hello 1173710230 \345\210\230\346\231\266\344\273\237\357\274\201"
.LC1:
.string  "Hello %s %s\n"
    .text
    .globl  main
    .type   main, @function
main:
.LFB5:

```

图 3-4 全局变量与全局函数

3.3.3 主函数的参数

主函数的参数部分给出了 int argc, char *argv[] 两个参数。在汇编代码中,分别将其存放在栈中 rbp 寄存器指向地址-20 和-32 处,如图 3-5 所示。其中%edi 代表 argc, %rsi 代表 argv[]。

```

subq    $32, %rsp
movl    %edi, -20(%rbp)
movq    %rsi, -32(%rbp)
cmpl    $3, -20(%rbp)
je       .L2
leaq    .LC0(%rip), %rdi
call    puts@PLT
movl    $1, %edi

```

24,1-8 27%

图 3-5 主函数参数

3.3.4 条件判断语句

在 main 函数中，使用 if 语句进行了条件判断。cmpl 语句进行判断条件的比较。如果条件满足则继续顺序执行，调用 puts 输出给定字符串（这里 puts 是对 printf 的优化），然后使用参数 1 调用 exit 结束程序。对应的汇编代码如下。

```

luck1798@ubuntu: ~/Desktop/HelloWorld
File Edit View Search Terminal Help
cmpl    $3, -20(%rbp)
je       .L2
leaq    .LC0(%rip), %rdi
call    puts@PLT
movl    $1, %edi
call    exit@PLT
.L2:

```

图 3-6 条件判断

```

15
16
17
18
19
20
if(argc!=3)
{
    printf("Usage: Hello 1173710230 刘晶仟!\n");
    exit(1);
}

```

图 3-6-1 C 语言条件判断

3.3.5 循环结构

在 main 函数中，使用 for 进行循环部分。循环部分存在局部变量 i，该变量存放在栈中 rbp 寄存器指向地址-4 处。首先对其置零进行初始化。接着使用 jump to middle 模式进入.L3 使用 cmpl 语句先进行条件判断。如果条件满足，那么进入.L4 循环体部分调用 printf 函数和 sleep 函数。

在调用 printf 的过程中，进行了数组访问（argv[1]和 argv[2]）。而 argv 是指针数组，所以会进行二次寻址。在汇编代码中，.L4 前三行取出 argv[2] 对应的内容，并放入三号参数寄存器%rdx 中。.L4 的四到六行取出 argv[1] 对应的内容，并放入二号参数寄存器%rsi 中。4.L4 第七行将格式字符串放到一

号参数寄存器%edi 中，然后调用 printf 函数进行显示。之后读取全局变量 sleepsecs 并调用 sleep 函数。最后对计数器进行加一，结束循环体部分。

```

.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
    movl    sleepsecs(%rip), %eax
    movl    %eax, %edi
    call    sleep
    addl    $1, -4(%rbp)
.L3:
    cmpl    $9, -4(%rbp)
    jle     .L4
    call    getchar
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc

```

图 3-7 循环结构

3.4 本章小结

这一阶段编译器处理实现 hello.i 到 hello.s 的转化，同时实现编译语言由高级程序语言到汇编语言的转化。

(第 3 章 2 分)

第 4 章 汇编

4.1 汇编的概念与作用

汇编程序是把汇编语言书写的程序翻译成与之等价的机器语言程序的翻译程序。汇编程序输入的是用汇编语言书写的源程序，输出的是用机器语言表示的目标程序。汇编语言的指令与机器语言的指令大体上保持一一对应的关系，

4.2 在 Ubuntu 下汇编的命令

执行 `gcc hello.i -S -o hello.s` 命令，生成 `hello.s` 文件

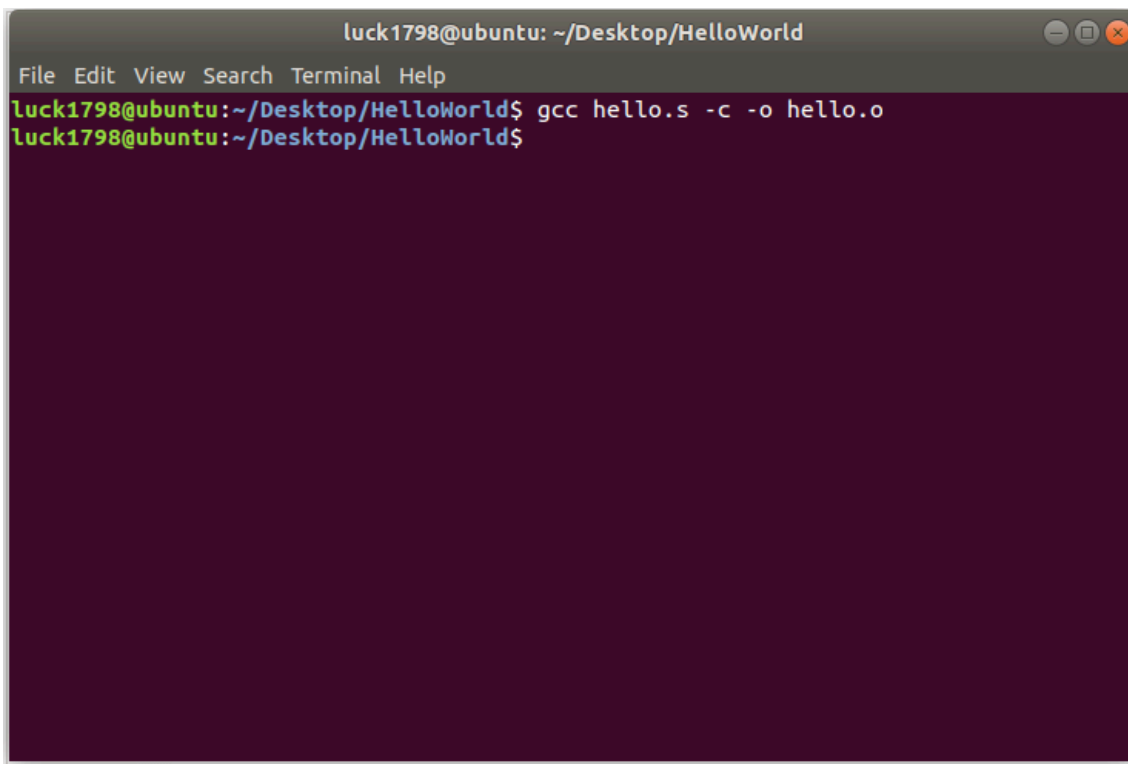
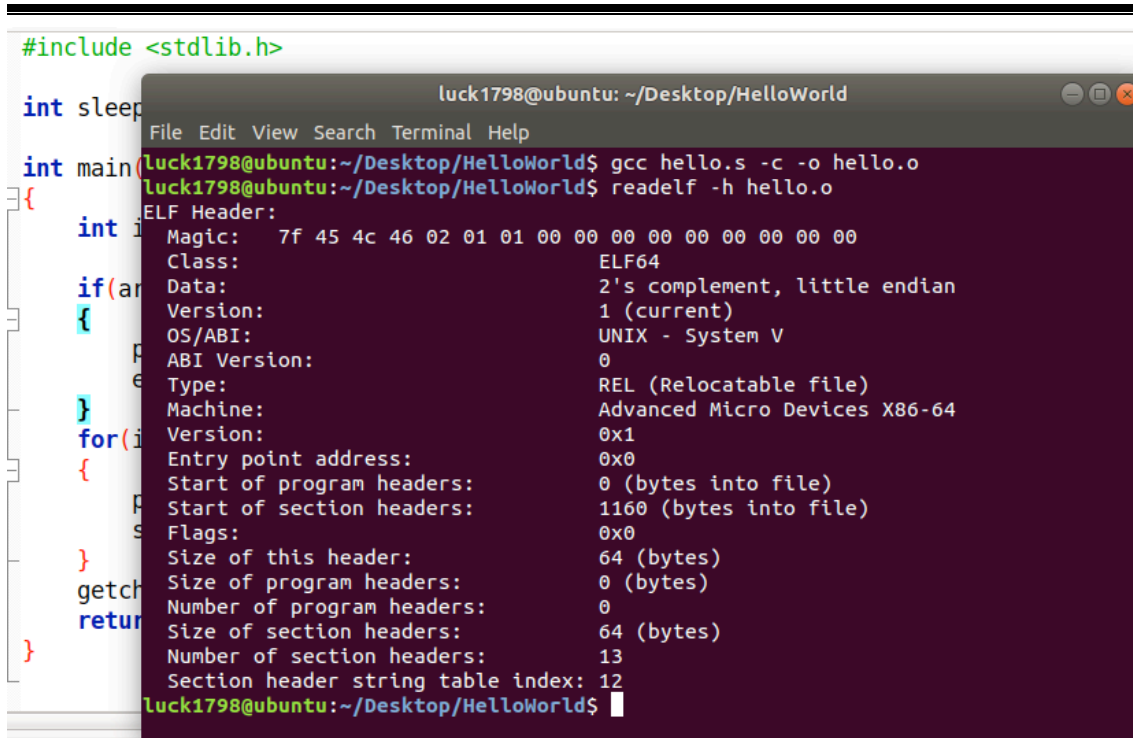


图 4-1 汇编执行

4.3 可重定位目标 elf 格式

执行 `gcc hello.s -c -o hello.o` 命令，生成 `hello.o` 文件。

执行 `readelf -h hello.o` 命令查看 `hello.h` 文件头信息。由图 4-2，知道该这个可重定位目标文件有 13 个节。



```
#include <stdlib.h>

int sleep

int main(
{
    int i
    if(a
    {
        p
        e
    }
    for(i
    {
        p
        s
    }
    getch
    retur
}
```

```
luck1798@ubuntu: ~/Desktop/HelloWorld
File Edit View Search Terminal Help
luck1798@ubuntu:~/Desktop/HelloWorld$ gcc hello.s -c -o hello.o
luck1798@ubuntu:~/Desktop/HelloWorld$ 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:              1160 (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
luck1798@ubuntu:~/Desktop/HelloWorld$
```

图 4-2 hello.o 文件头

执行 `readelf -S hello.o` 命令查看 `hello.o` 节头表。由图 4-3，得知各节的大小及作用。

```
luck1798@ubuntu:~/Desktop/HelloWorld$ readelf -S hello.o
There are 13 section headers, starting at offset 0x488:

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  00000348
      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
      0000000000000032  0000000000000000  A      0      0      8
 [ 6] .comment             PROGBITS          0000000000000000  000000fa
      000000000000002b  0000000000000001  MS      0      0      1
 [ 7] .note.GNU-stack      PROGBITS          0000000000000000  00000125
      0000000000000000  0000000000000000  0      0      1
 [ 8] .eh_frame            PROGBITS          0000000000000000  00000128
      0000000000000038  0000000000000000  A      0      0      8
 [ 9] .rela.eh_frame        RELA              0000000000000000  00000408
      0000000000000018  0000000000000018  I      10      8      8
[10] .symtab              SYMTAB            0000000000000000  00000160
      00000000000000198  0000000000000018  11      9      8
[11] .strtab              STRTAB            0000000000000000  000002f8
      000000000000004d  0000000000000000  0      0      1
[12] .shstrtab            STRTAB            0000000000000000  00000420
      0000000000000061  0000000000000000  0      0      1

Key to Flags:
 W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
 L (link order), O (extra OS processing required), G (group), T (TLS),
 C (compressed), x (unknown), o (OS specific), E (exclude),
 l (large), p (processor specific)
```

图 4-3 hello.o 节头表

根据图 4-3 可以得到各节的基本信息。由于是可重定位目标文件，所以每个节都从 0 开始，用于重定位。在文件头中得到节头表的信息，然后再使用节头表中的字节偏移信息得到各节在文件中的起始位置，以及各节所占空间的大小。同时可以观察到，代码段是可执行的，但是不能写；数据段和只读数据段都不可执行，而且只读数据段也不可写。

执行使用 `readelf -s hello.o` 命令，由图 4-4，可知 hello.o 符号表的信息。

```

Luck1798@ubuntu:~/Desktop/HelloWorld$ readelf -s hello.o
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 4-4 hello.o 的符号表

4.4 Hello.o 的结果解析

执行命令 `readelf -s hello.o` 查看 hello.o 反汇编结果


```

Disassembly of section .text:

0000000000000000 <main>:
 0: 55          push    %rbp
 1: 48 89 e5    mov     %rsp,%rbp
 4: 48 83 ec 20  sub     $0x20,%rsp
 8: 89 7d ec    mov     %edi,-0x14(%rbp)
 b: 48 89 75 e0  mov     %rsi,-0x20(%rbp)
 f: 83 7d ec 03  cmpl    $0x3,-0x14(%rbp)
13: 74 16       je      2b <main+0x2b>
15: 48 8d 3d 00 00 00 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+0x21
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        # 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 4-5 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。反汇编代码的栈空间利用率较高，而 `.s` 文件中对栈的使用有一定的浪费。

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。汇编语言是以人们比较熟悉的词句直接表述 CPU 动作形成的语言，是最接近 CPU 运行原理的较为通俗的比较容易理解的语言。在不同的设备中，汇编语言对应着不同的机器语言指令集，通过汇编过程转换成机器指令。机器语言与汇编语言具有一一对应的映

射关系，一条机器语言程序对应一条汇编语言语句，但不同平台之间不可直接移植。

4.5 本章小结

本阶段完成了对 `hello.s` 到可重定位目标文件 `hello.o` 的转变。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。

在现代系统中链接是由叫做链接器的程序自动执行的。链接器使得分离编译成为可能。我们不用将一个大型的应用程序组织为一个巨大的源文件，而是可以把它分解为更小，更好管理的模块，可以独立地修改和编译这些文件，当我们改变这些模块中的一个时，只需重新编译它，并重新链接，而不必重新编译其他文件。

5.2 在 Ubuntu 下链接的命令

执行 `ld -o hello -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o hello.o /usr/lib/x86_64-linux-gnu/libc.so /usr/lib/x86_64-linux-gnu/crtn.o` 命令，好长的一串。

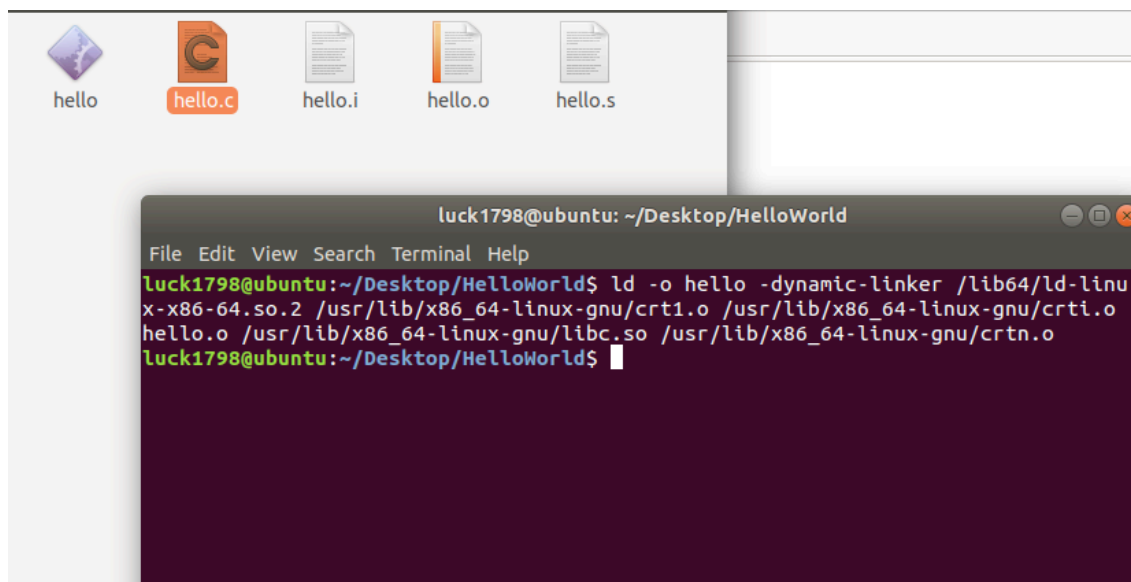


图 5-1 链接执行

5.3 可执行目标文件 hello 的格式

执行 `readelf -h hello` 命令，查看 `hello` 文件头信息。由图 5-2 可知，该文件是可执行目标文件，有 25 个节。

```
luck1798@ubuntu:~/Desktop/HelloWorld$ 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 5-2 `hello` 的文件头

执行 `readelf -S hello` 命令，查看 `hello` 节头表信息。由图 5-3 可知各节的大小，以及它们可以进行的操作。

```

luck1798@ubuntu: ~/Desktop/HelloWorld
File Edit View Search Terminal Help
luck1798@ubuntu:~/Desktop/HelloWorld$ 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    0  0  0
     0000000000000000  0000000000000000
[ 1] .interp              PROGBITS          0000000000400200    0  0  1
     000000000000001c  0000000000000000  A      0  0
[ 2] .note.ABI-tag        NOTE              000000000040021c    0  0  4
     0000000000000020  0000000000000000  A      0  0
[ 3] .hash                HASH              0000000000400240    0  0  8
     0000000000000034  0000000000000004  A      5  0
[ 4] .gnu.hash            GNU_HASH          0000000000400278    0  0  8
     000000000000001c  0000000000000000  A      5  0
[ 5] .dynsym              DYNSYM            0000000000400298    0  1  8
     00000000000000c0  0000000000000018  A      6  1
[ 6] .dynstr              STRTAB            0000000000400358    0  0  1
     0000000000000057  0000000000000000  A      0  0
[ 7] .gnu.version         VERSYM            00000000004003b0    0  0  2
     0000000000000010  0000000000000002  A      5  0
[ 8] .gnu.version_r       VERNEED           00000000004003c0    0  1  8
     0000000000000020  0000000000000000  A      6  1
[ 9] .rela.dyn            RELA              00000000004003e0    0  0  8
     0000000000000030  0000000000000018  A      5  0
[10] .rela.plt            RELA              0000000000400410    0  19 8
     0000000000000078  0000000000000018  AI     5  19
[11] .init                PROGBITS          0000000000400488    0  0  4
     0000000000000017  0000000000000000  AX     0  0
[12] .plt                 PROGBITS          00000000004004a0    0  0 16
     0000000000000060  0000000000000010  AX     0  0
[13] .text                PROGBITS          0000000000400500    0  0 16
     0000000000000132  0000000000000000  AX     0  0
[14] .fini                PROGBITS          0000000000400634    0  0  4
     0000000000000009  0000000000000000  AX     0  0
[15] .rodata              PROGBITS          0000000000400640    0  0  8
     000000000000003a  0000000000000000  A      0  0
[16] .eh_frame            PROGBITS          0000000000400680    0  0  8
     00000000000000fc  0000000000000000  A      0  0
[17] .dynamic              DYNAMIC           0000000000600e50    0  0  8
     00000000000001a0  0000000000000010  WA     6  0
[18] .got                 PROGBITS          0000000000600ff0    0  0  8
     0000000000000000  0000000000000000

```

图 5-3 hello 的段头表（部分）

根据图 5-3 可以得到各段的基本信息。由于是可执行目标文件，所以每个段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。这样可以直接从文件起始处得到各段的起始位置，以及各段所占空间的大小。同时可以观察到，代码段是可执行的，但是不能写；数据段和只读数据段都不可执行，而且只读数据段也不可写

执行 `readelf -s hello` 可以查看 hello 符号表的信息，

```

Luck1798@ubuntu:~/Desktop/HelloWorld$ readelf -s hello

Symbol table '.dynsym' contains 8 entries:

```

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	puts@GLIBC_2.2.5 (2)
2:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	printf@GLIBC_2.2.5 (2)
3:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	__libc_start_main@GLIBC_2.2.5 (2)
4:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	getchar@GLIBC_2.2.5 (2)
5:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__
6:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	exit@GLIBC_2.2.5 (2)
7:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	sleep@GLIBC_2.2.5 (2)

图 5-4 hello 的符号表（部分）

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。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 0x400500 处, 大小为 0x0132

The screenshot shows the 'Data Dump' window in the edb debugger. The address range selected is 0x0000000000400500 to 0x0000000000401000. The dump displays hexadecimal values in the first column and their ASCII representations in the second column. The ASCII column shows various characters, including spaces, tabs, and some non-printable characters represented by dots.

图 5-5 edb

5.5 链接的重定位过程分析

运行 `objdump -d -r hello` 与 `objdump -d -r hello.o` 得到反汇编结果。由图 5-6 和图 5-7, 可以发现二者反汇编结果不同。可执行文件 hello 的反汇编结果中给出了重定位结果, 即虚拟地址的确定。而可重定位文件 hello.o 的反汇编结果中, 各部分的开始地址均为 0。


```

luck1798@ubuntu: ~/Desktop/HelloWorld
File Edit View Search Terminal Help
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 fa 00 00 00 lea     0xfa(%rip),%rdi      # 400648 <
_IO_stdin_used+0x8>
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            jmp     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 e7 00 00 00 lea     0xe7(%rip),%rdi      # 40066d <
_IO_stdin_used+0x2d>
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 00 mov     0x200aae(%rip),%eax      # 6010
44 <sleepsecs>
400596: 89 c7            mov     %eax,%edi
400598: e8 53 ff ff ff    callq   4004f0 <sleep@plt>

```

图 5-6 hello 的反汇编

```

luck1798@ubuntu: ~/Desktop/HelloWorld
File Edit View Search Terminal Help
luck1798@ubuntu:~/Desktop/HelloWorld$ objdump -d -r hello.o

hello.o:      file format 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
43:  48 83 c0 08             add     $0x8,%rax
47:  48 8b 00               mov     (%rax),%rax

```

图 5-7 hello.o 的反汇编

5.6 hello 的执行流程

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。

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

本阶段实现了可重定位文件 `hello.o` 链接生成可执行性文件 `hello`。此外，本章通过一系列细致分析，了解可执行性文件重定位过程，以及可执行性文件的动态链接过程。。

(第 5 章 1 分)

第 6 章 hello 进程管理

6.1 进程的概念与作用

进程的经典定义是一个执行中的程序的实体。我们来分析一下这句话：第一，进程是一个实体。每一个进程都有它自己的地址空间，一般情况下，包括文本区域、数据区域和堆栈。第二，进程是一个“执行中的程序”。程序是一个没有生命的实体，只有处理器赋予程序生命时（操作系统执行），它才能成为一个活动的实体，我们称其为进程。

在现代系统上运行一个程序时，进程给我们一个假象，就好像是我们的程序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

Linux 实质上是一个操作系统内核，一般用户不能直接使用内核，而是通过外壳程序，也就是所谓的 shell 来与内核进行沟通。外壳程序可以保证操作系统的安全性，抵御用户的一些不正确操作。Linux 的外壳程序称作 shell（命令行解释器），它能够将命令翻译给内核、将内核处理结果翻译给用户。一般我们使用的 shell 为 bash。在解释命令的时候，bash 不会直接参与解释，而是创建新进程进行命令的解释，bash 只用等待结果即可，这样能保证 bash 进程的安全。

首先 shell 检查命令是否是内部命令，若不是再检查是否是一个应用程序（这里的应用程序可以是 Linux 本身的实用程序，如 ls 和 rm；也可以是购买的商业程序，如 xv；或者是自由软件，如 emacs）。然后 shell 在搜索路径里寻找这些应用程序（搜索路径就是一个能找到可执行程序的目录列表）。如果输入的命令不是一个内部命令且在路径里没有找到这个可执行文件，将会显示一条错误信息。如果能找到命令，该内部命令或应用程序分解后将被系统调用并传给 Linux 内核。

6.3 Hello 的 fork 进程创建过程

一个进程，包括代码、数据和分配给进程的资源。fork 函数通过系统调用创建一个与原来进程几乎完全相同的进程，也就是两个进程可以做完全相同的事，但如果初始参数或者传入的变量不同，两个进程也可以做不同的事。一个进程调用 fork 函数后，系统先给新的进程分配资源，例如存储数据和代码的空间。然后把原

来的进程的所有值都复制到新的新进程中，只有少数值与原来的进程的值不同。相当于克隆了一个自己。在 `fork` 函数执行完毕后，如果创建新进程成功，则出现两个进程，一个是子进程，一个是父进程。在子进程中，`fork` 函数返回 0，在父进程中，`fork` 返回新创建子进程的进程 ID。我们可以通过 `fork` 返回的值来判断当前进程是子进程还是父进程

6.4 Hello 的 `execve` 过程

一个进程，包括代码、数据和分配给进程的资源。`fork` 函数通过系统调用创建一个与原来进程几乎完全相同的进程，也就是两个进程可以做完全相同的事，但如果初始参数或者传入的变量不同，两个进程也可以做不同的事。一个进程调用 `fork` 函数后，系统先给新的进程分配资源，例如存储数据和代码的空间。然后把原来的进程的所有值都复制到新的新进程中，只有少数值与原来的进程的值不同。相当于克隆了一个自己。在 `fork` 函数执行完毕后，如果创建新进程成功，则出现两个进程，一个是子进程，一个是父进程。在子进程中，`fork` 函数返回 0，在父进程中，`fork` 返回新创建子进程的进程 ID。我们可以通过 `fork` 返回的值来判断当前进程是子进程还是父进程

6.5 Hello 的进程执行

新进程的创建，首先在内存中为新进程创建一个 `task_struct` 结构，然后将父进程的 `task_struct` 内容复制其中，再修改部分数据。分配新的内核堆栈、新的 PID、再将 `task_struct` 这个 node 添加到链表中。然后将可执行文件装入内核的 `linux_binprm` 结构体。进程调用 `execve` 时，该进程执行的程序完全被替换，新的程序从 `main` 函数开始执行。调用 `execve` 并不创建新进程，只是替换了当前进程的代码区、数据区、堆和栈。在进程调用了 `exit` 之后，该进程并非马上就消失掉，而是留下了一个成为僵尸进程的数据结构，记载该进程的退出状态等信息供其他进程收集，除此之外，僵尸进程不再占有任何内存空间。

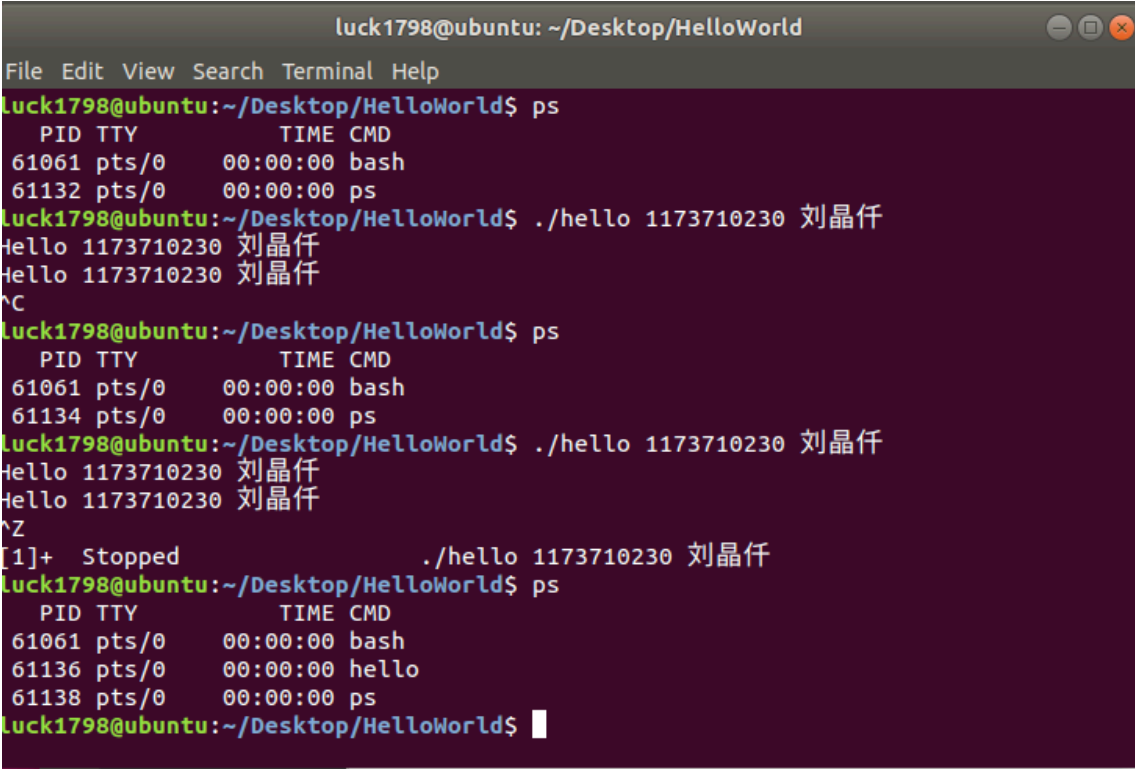
为了控制进程的执行，内核必须有能力挂起正在 CPU 上执行的进程，并恢复以前挂起的某个进程的执行，这叫做进程切换。进程上下文切换由以下 4 个步骤组成：（1）决定是否作上下文切换以及是否允许作上下文切换。包括对进程调度原因的检查分析，以及当前执行进程的资格和 CPU 执行方式的检查等。在操作系统中，上下文切换程序并不是每时每刻都在检查和分析是否可作上下文切换，它们设置有适当的时机。（2）保存当前执行进程的上下文。这里所说的当前执行进程，实际上是指调用上下文切换程序之前的执行进程。如果上下文切换不是被那个当

前执行进程所调用，且不属于该进程，则所保存的上下文应是先前执行进程的上下文，或称为“老”进程上下文。显然，上下文切换程序不能破坏“老”进程的上下文结构。（3）使用进程调度算法，选择一处于就绪状态的进程。（4）恢复或装配所选进程的上下文，将 CPU 控制权交到所选进程手中。

6.6 hello 的异常与信号处理

1.

如图 6-1，是程序执行途中输入 ctrl+c, 与 ctrl+z 的情况。输入 ctrl+c 时，程序在输出两次信息后，shell 父进程收到 SIGINT 信号，信号处理函数的逻辑是结束 hello，并回收 hello 进程。输入 ctrl+z 时，shell 会发送一个 SIGTSTP 信号给前台进程组中的进程，从而将其挂起。（ps 指令可以看出 hello 进程是否被回收）

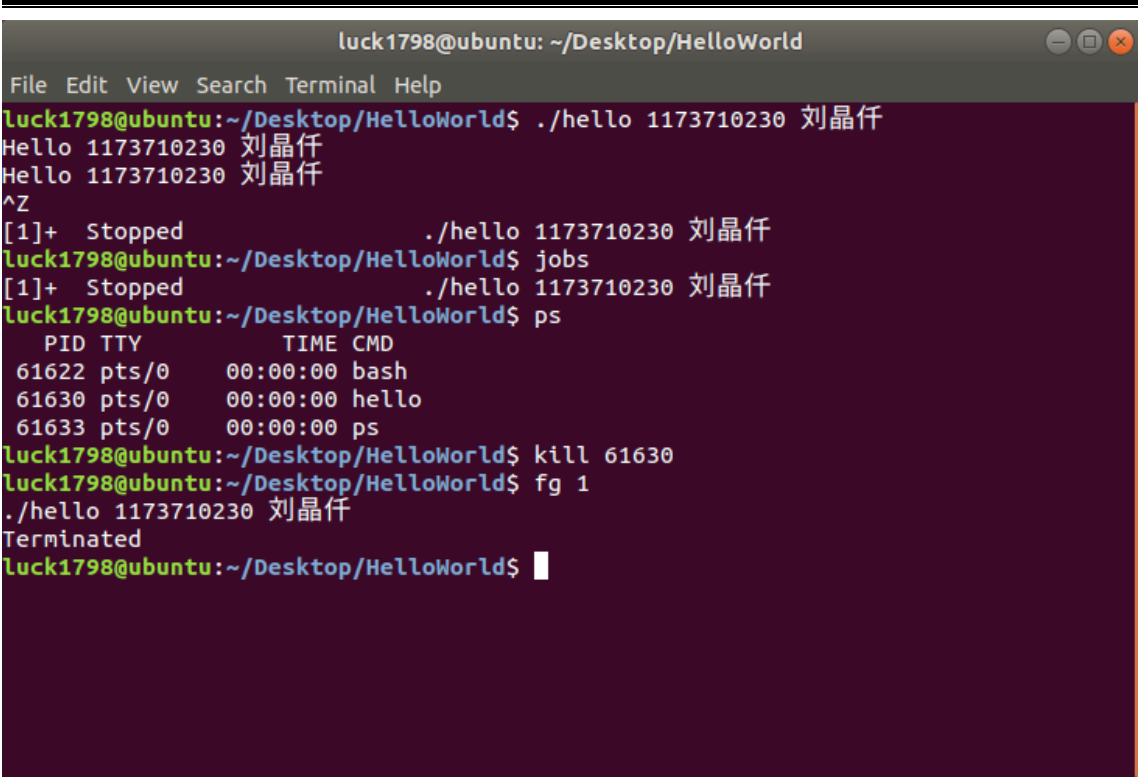


```
luck1798@ubuntu: ~/Desktop/HelloWorld
File Edit View Search Terminal Help
luck1798@ubuntu:~/Desktop/HelloWorld$ ps
  PID TTY          TIME CMD
 61061 pts/0        00:00:00 bash
 61132 pts/0        00:00:00 ps
luck1798@ubuntu:~/Desktop/HelloWorld$ ./hello 1173710230 刘晶仟
Hello 1173710230 刘晶仟
Hello 1173710230 刘晶仟
^C
luck1798@ubuntu:~/Desktop/HelloWorld$ ps
  PID TTY          TIME CMD
 61061 pts/0        00:00:00 bash
 61134 pts/0        00:00:00 ps
luck1798@ubuntu:~/Desktop/HelloWorld$ ./hello 1173710230 刘晶仟
Hello 1173710230 刘晶仟
Hello 1173710230 刘晶仟
^Z
[1]+  Stopped                  ./hello 1173710230 刘晶仟
luck1798@ubuntu:~/Desktop/HelloWorld$ ps
  PID TTY          TIME CMD
 61061 pts/0        00:00:00 bash
 61136 pts/0        00:00:00 hello
 61138 pts/0        00:00:00 ps
luck1798@ubuntu:~/Desktop/HelloWorld$
```

图 6-1 异常与信号命令组 1

2.

如图 6-2，ctrl+z 挂起 hello 进程，jobs 查看暂停进程，ps 查看进程及其运行时间，kill x 杀死 PID 为 x 的进程。fg 使进程在前台执行，发现 hello 进程 Terminated，证明 hello 进程被 kill 杀死。



```
luck1798@ubuntu: ~/Desktop/HelloWorld
File Edit View Search Terminal Help
luck1798@ubuntu:~/Desktop/HelloWorld$ ./hello 1173710230 刘晶仟
Hello 1173710230 刘晶仟
Hello 1173710230 刘晶仟
^Z
[1]+  Stopped                  ./hello 1173710230 刘晶仟
luck1798@ubuntu:~/Desktop/HelloWorld$ jobs
[1]+  Stopped                  ./hello 1173710230 刘晶仟
luck1798@ubuntu:~/Desktop/HelloWorld$ ps
  PID TTY          TIME CMD
 61622 pts/0        00:00:00 bash
 61630 pts/0        00:00:00 hello
 61633 pts/0        00:00:00 ps
luck1798@ubuntu:~/Desktop/HelloWorld$ kill 61630
luck1798@ubuntu:~/Desktop/HelloWorld$ fg 1
./hello 1173710230 刘晶仟
Terminated
luck1798@ubuntu:~/Desktop/HelloWorld$
```

图 6-2 异常与信号命令组 2

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，例如调用 `fork` 创建新进程，调用 `execve` 执行 `hello`，`hello` 的进程执行，`hello` 的异常与信号处理，分析了操作系统与它的硬件环境，软件环境之间协同与配合。

(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

物理地址是用于内存芯片级的单元寻址，与处理器和 CPU 连接的地址总线相对应。现代操作系统都提供了一种内存管理的抽象，即虚拟内存。进程使用虚拟内存中的地址，即虚拟地址，由操作系统协助相关硬件，把它“转换”成真正的物理地址。hello 中使用的就是虚拟空间的虚拟地址。线性地址指虚拟地址到物理地址变换的中间层，是处理器可寻址的内存空间（称为线性地址空间）中的地址。程序代码会产生逻辑地址，或者说段中的偏移地址，加上相应段基址就成了一个线性地址。如果启用了分页机制，那么线性地址可以再经过变换产生物理地址。若是没有采用分页机制，那么线性地址就是物理地址。而逻辑地址指的是机器语言指令中，用来指定一个操作数或者是一条指令的地址。它是 Intel 为了兼容，而将段式内存管理方式保留下来的产物。

逻辑（虚拟）地址经过分段（查询段表）转化为线性地址。线性地址经过分页（查询页表）转为物理地址。

7.2 Intel 逻辑地址到线性地址的变换-段式管理

一个逻辑地址由两部分组成，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 的页式内存管理单元，负责把一个线性地址，最终翻译为一个物理地址。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 $total_page[2^{20}]$ 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——一对应的页的地址。另一类“页”，我们称之为物理页，或者是页框、页帧的。

是分页单元把所有的物理内存也划分为固定长度的管理单位，它的长度一般与内存页是一一对应的

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

如图 7-1 所示，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 被划分成四个 9 位 VPN，分别用于一个页表的偏移量。

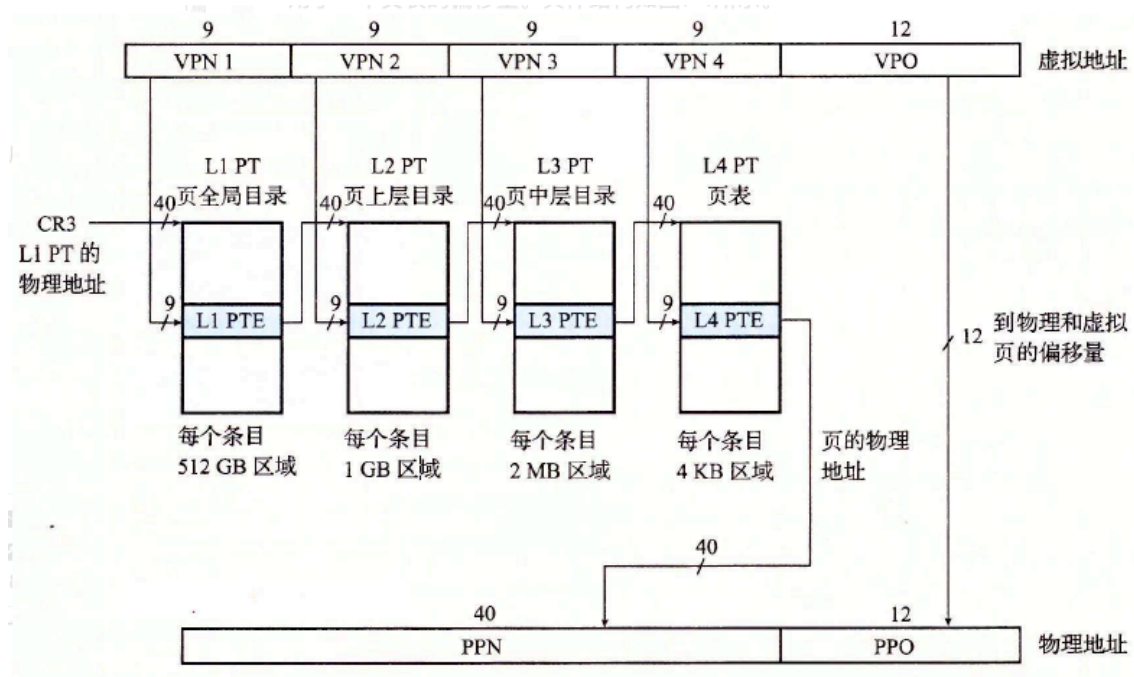


图 7-1 (转) CORE i7 页表翻译

7.5 三级 Cache 支持下的物理内存访问

如图 7-2 所示，首先 CPU 发出一个虚拟地址，在 TLB 里面寻找。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。这样就能完成 Cache 和 TLB 的配合工作。

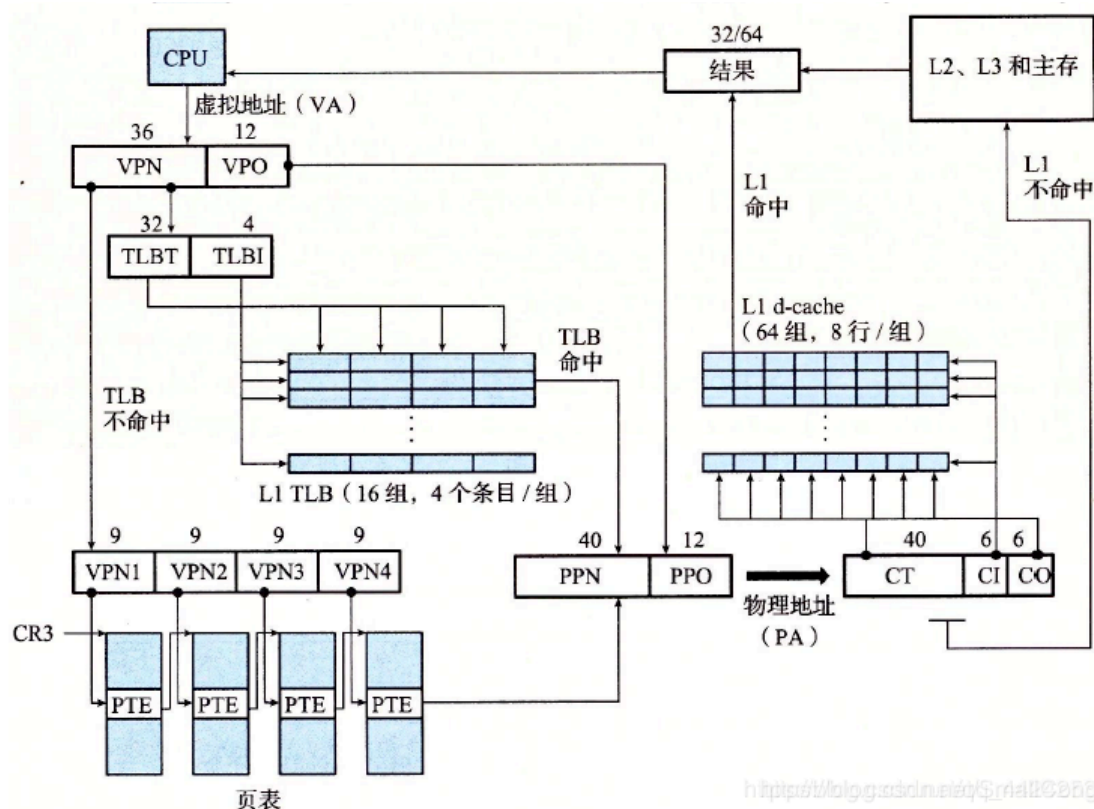


图 7-2 (转) 三级 Cache 支持下的物理内存访问

7.6 hello 进程 fork 时的内存映射

虚拟内存和内存映射解释了 fork 函数如何为每个新进程提供私有的虚拟地址空间。Fork 函数为新进程创建虚拟内存。创建当前进程的 `mm_struct`, `vm_area_struct` 和页表的原样副本，两个进程中的每个页面都标记为只读，两个进程中的每个区域结构 (`vm_area_struct`) 都标记为私有的写时复制 (COW)。在新进程中返回时，新进程拥有与调用 fork 进程相同的虚拟内存，随后的写操作通过写时复制机制创建新页面。

7.7 hello 进程 execve 时的内存映射

`execve` 函数在当前进程中加载并运行新程序 hello 的步骤：删除已存在的用户区域，创建新的区域结构，代码和初始化数据映射到 `.text` 和 `.data` 区（目标文件提供），`.bss` 和栈映射到匿名文件，设置 PC，指向代码区域的入口点。Linux 根据需要换入代码和数据页面。

7.8 缺页故障与缺页中断处理

DRAM 缓存不命中称为缺页，即虚拟内存中的字不在物理内存中。缺页导致页面出错，产生缺页异常。缺页异常处理程序选择一个牺牲页，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（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。分配器将堆视为一组不同大小的块的集合来维护，每个块要么是已分配的，要么是空闲的。分配器的类型包括显式分配器和隐式分配器。前者要求应用显式地释放任何已分配的块，后者在检测到已分配块不再被程序所使用时，就释放这个块。

动态内存管理的策略包括首次适配、下一次适配和最佳适配。首次适配会从头开始搜索空闲链表，选择第一个合适的空闲块。搜索时间与总块数（包括已分配和空闲块）成线性关系。会在靠近链表起始处留下小空闲块的“碎片”。下一次适配和首次适配相似，只是从链表中上一次查询结束的地方开始。比首次适应更快，避免重复扫描那些无用块。最佳适配会查询链表，选择一个最好的空闲块，满足适配，且剩余最少空闲空间。它可以保证碎片最小，提高内存利用率。

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 7 章 2 分）

第 8 章 hello 的 IO 管理

8.1 Linux 的 IO 设备管理方法

(以下格式自行编排, 编辑时删除)

设备的模型化: 文件

设备管理: unix io 接口

首先是设备的模型化。在设备模型中, 所有的设备都通过总线相连。每一个设备都是一个文件。设备模型展示了总线和它们所控制的设备之间的实际连接。在最底层, Linux 系统中的每个设备由一个 `struct device` 代表, 而 Linux 统一设备模型就是在 `kobject kset ktype` 的基础之上逐层封装起来的。设备管理则是通过 `unix io` 接口实现的。

8.2 简述 Unix IO 接口及其函数

1. Unix I/O 接口统一操作:

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。

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。

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`。

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` 字节的而文件, 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`。

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。

2. Unix I/O 函数:

`int open(char* filename, int flags, mode_t mode)`, 进程通过调用 `open` 函数来打开一个存在的文件或是创建一个新文件的。`open` 函数将 `filename` 转换为一个文件

描述符，并且返回描述符数字，返回的描述符总是在进程中当前没有打开的最小描述符，`flags` 参数指明了进程打算如何访问这个文件，`mode` 参数指定了新文件的访问权限位。

`int close(fd)`, `fd` 是需要关闭的文件的描述符，`close` 返回操作结果。

`ssize_t read(int fd, void *buf, size_t n)`, `read` 函数从描述符为 `fd` 的当前文件位置赋值最多 `n` 个字节到内存位置 `buf`。返回值 `-1` 表示一个错误，`0` 表示 EOF，否则返回值表示的是实际传送的字节数量。

`ssize_t write(int fd, const void *buf, size_t n)`, `write` 函数从内存位置 `buf` 复制至多 `n` 个字节到描述符为 `fd` 的当前文件位置。

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）。

`printf` 函数代码如下所示：

```
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;
}
```

`(char*)&fmt + 4` 表示的是…可变参数中的第一个参数的地址。而 `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。接着从 `vsprintf` 生成显示信息，到 `write` 系统函数，直到陷阱系统调用 `int 0x80` 或 `syscall`。字符显示驱动子程序：从 ASCII

到字模库到显示 vram（存储每一个点的 RGB 颜色信息）。显示芯片按照刷新频率逐行读取 vram，并通过信号线向液晶显示器传输每一个点（RGB 分量）

8.4 getchar 的实现分析

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 下的 I/O 设备管理方法、Unix I/O 接口及其函数，分析了 printf 函数和 getchar 函数的实现。

（第 8 章 1 分）

结论

hello.c 通过键盘鼠标等 I/O 设备输入计算机，并存储在内存中。在此之后预处理器将 hello.c 预处理成为文本文件 hello.i，接着编译器将 hello.i 翻译成汇编语言文件 hello.s，汇编器将 hello.s 汇编成可重定位文件 hello.o。链接器将外部文件和 hello.o 连接起来形成可执行性文件 hello。

shell 通过 fork 和 execve 创建 hello 进程。这样 hello 摇身一变为进程。之后 shell 创建新的内存区域，并加载代码、数据和堆栈。hello 在执行的过程中遇到异常，会接受 shell 的信号完成处理。hello 在执行的过程中需要使用内存，那么就通过 CPU 和虚拟空间进行地址访问。hello 执行结束后，shell 回收其僵尸进程，最后进程从系统中消失。这就是 hello 的一生。

计算机系统的设计与实现，处处体现着抽象的含义。比如，程序的本质是机器语言。而汇编语言对机器语言进行了抽象，高级语言对汇编语言进行了抽象。所以我们进行相关计算机操作时变得更方便。再比如，各种物理内存的实现方式有磁盘、软盘等。而使用虚拟内存的概念实现了对各种物理内存的抽象。概念上的抽象使得对概念的使用变得简单，这就是我对计算机系统实现的一个感悟。

（结论 0 分，缺少 -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。

中间文件	作用
Hello.i	Hello.c 经预处理（cpp）得到
Hello.s	Hello.i 经编译器（ccl）得到汇编文件
Hello.o	Hello.s 经汇编器（as）得到可重定位文件
Hello	链接之后得到可执行文件

（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] ELF 构造.[OL].<https://www.cs.stevens.edu/~jschauma/631/elf.htm>.
- [2] 小楼一夜听春雨. linux 创建连接命令 ln -s 软链接.[OL].[2016-02-16].
<https://www.cnblogs.com/kexln/p/5193826.html>.
- [3] 赵子清. vim 模式与模式切换.[OL].[2015-07-03].
<https://www.cnblogs.com/zzqcn/p/4619012.html>.

(参考文献 0 分，确实 -1 分)