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　软件工程

学　　号　1183710113

班　　级　1837101

学　　生　许健

指 导 教 师　史先俊

计算机科学与技术学院

2019 年 12 月

# 摘　要

对于每个程序员来说，Hello World 是一个开始，本论文目的在于利用 gcc、edb 等工具，结合 CSAPP 教材，研究 hello 程序在 Linux 系统下的整个生命周期，从而达到融会贯通所学知识的效果。

**关键词：**CSAPP；HIT；大作业；hello

**（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

## 1.1 Hello 简介

P2P：程序员用键盘输入 hello.c 文件，一个 hello 的 C 语言文件诞生，然后经过预处理器、汇编器、编译器、链接器的一系列处理，hello 可执行文件诞生了，在 shell 中键入启动命令后，shell 为其 fork，产生子进程，于是 hello 便从 Program 变成了 Process。

020：shell 调用 execve 函数在新的子进程中加载并运行 hello，在 hello 运行的过程中，还需要 CPU 为 hello 分配内存、时间片，使得 hello 看似独享 CPU 资源。系统的进程管理帮助 hello 切换上下文、shell 的信号处理程序使得 hello 在运行过程中可以处理各种信号，当程序员主动地按下 Ctrl+Z 或者 hello 运行到 return 0 时，hello 所在进程将被杀死，shell 会回收它的僵死进程，内核删除相关数据结构。

## 1.2 环境与工具

硬件环境：Inter(R) Core(TM) i5-7300HQ CPU；2.5GHz；8G RAM；128G SSD+1T HDD

软件环境：Windows 10 64 位；Vmware 15；Ubuntu 19.04 64 位

开发与调试工具：gcc；edb； readelf；objdump；gedit；hexedit；

## 1.3 中间结果

hello.c——原文件
hello.i——预处理之后文本文件
hello.s——编译之后的汇编文件
hello.o——汇编之后的可重定位目标执行
hello——链接之后的可执行目标文件
hello.elf——hello.o 的 elf 格式，用来看 hello.o 的各节信息
hello.ob——hello.o 的反汇编文件，用来看汇编器翻译后的汇编代码
hello1.ob——hello 的反汇编文件，用来看链接器链接后的汇编代码

## 1.4 本章小结

本章主要简单介绍了 hello 的 P2P，020 过程，列出了本次实验信息：环境、中间结果。

（第 1 章 0.5 分）

# 第 2 章 预处理

## 2.1 预处理的概念与作用

预处理的概念：预处理器(cpp)根据以字符#开头的命令，修改原始的 C 程序。将所引用的所有库展开，处理所有的条件编译，并执行所有的宏定义，得到另一个通常是以.i 作为文件扩展名的 C 程序。

预处理的作用：

1. 将 c 程序中所有#include 声明的头文件复制到新的程序中。比如 hello.c 中第 6~8 行的#include <stdio.h>、#include <unistd.h>、#include <stdlib.h>命令告诉预处理器读取系统头文件 stdio.h、unistd.h、stdlib.h 的内容，并把它直接插人程序文本中；
2. 条件编译。根据条件#if 决定是否处理之后的代码；
3. 执行宏替换。用实际值替换用#define 定义的字符串。

## 2.2 在 Ubuntu 下预处理的命令

命令：cpp hello.c > hello.i

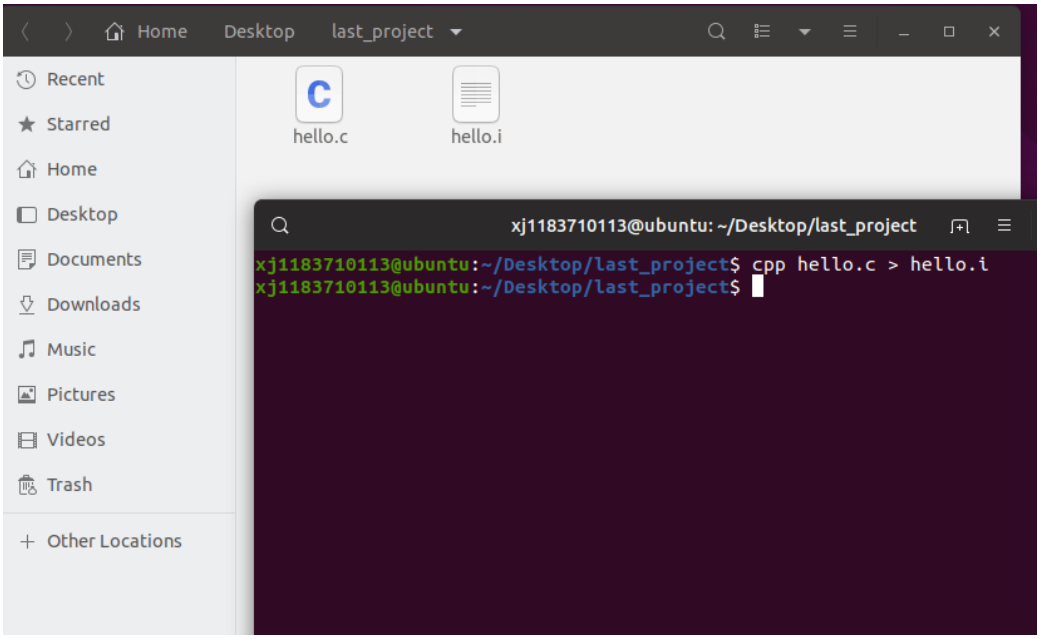


图 2.1 对 hello.c 进行预处理

## 2.3 Hello 的预处理结果解析

使用 Text Editor 打开 hello.i，发现原来的 helloc.c 已经被拓展成了 3042 行，前面的内容是 hello.c 的三个#include 指令包含的头文件的代码，先寻找 main 函数，main 函数从第 3029 行开始，如下图。

```
3028 # 10 "hello.c"
3029 int main(int argc,char *argv[]){
3030  int i;
3031
3032  if(argc!=4){
3033   printf("用法: Hello 学号 姓名 秒数！\n");
3034   exit(1);
3035  }
3036  for(i=0;i<8;i++){
3037   printf("Hello %s %s\n",argv[1],argv[2]);
3038   sleep(atoi(argv[3]));
3039  }
3040  getchar();
3041  return 0;
3042 }
```

图 2.2 hello.i 中的 main 函数

再看之前的头文件的处理，以第一条#include 指令为例，cpp 到默认的环境变量下搜索 stdio.h 头文件，打开/usr/include/stdio.h，发现其中仍有#include 指令，于是再去搜索包含的头文件，直到最后的文件中没有#include 指令，并把所有文件中的所有#define 和#ifdef 指令进行处理，执行宏替换和通过条件确定是否处理定义的指令。如图是对 stdio.h 包含文件的展开。

```
13 # 1 "/usr/include/stdio.h" 1 3 4
14 # 27 "/usr/include/stdio.h" 3 4
15 # 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
16 # 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
17 # 1 "/usr/include/features.h" 1 3 4
18 # 446 "/usr/include/features.h" 3 4
19 # 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
20 # 452 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
21 # 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
22 # 453 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
23 # 1 "/usr/include/x86_64-linux-gnu/bits/long-double.h" 1 3 4
24 # 454 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
25 # 447 "/usr/include/features.h" 2 3 4
26 # 470 "/usr/include/features.h" 3 4
27 # 1 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 1 3 4
28 # 10 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 3 4
29 # 1 "/usr/include/x86_64-linux-gnu/gnu/stubs-64.h" 1 3 4
30 # 11 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 2 3 4
31 # 471 "/usr/include/features.h" 2 3 4
32 # 34 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 2 3 4
33 # 28 "/usr/include/stdio.h" 2 3 4
```

图 2.3 #include<stdio.h>包含文件展开

## 2.4 本章小结

本章主要介绍了预处理的概念及作用，并结合 hello.c 处理后的 hello.i 对处理过程进行分析。

（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：编译器将文本文件 hello.i 翻译成另一个文本文件 hello.s，它包含一个汇编语言程序。

编译的作用：将字符串转化成内部的表示结构，然后得到一系列记号，生成语法树，最后将语法树转化为目标代码。

## 3.2 在 Ubuntu 下编译的命令

命令：gcc -S hello.i -o hello.s

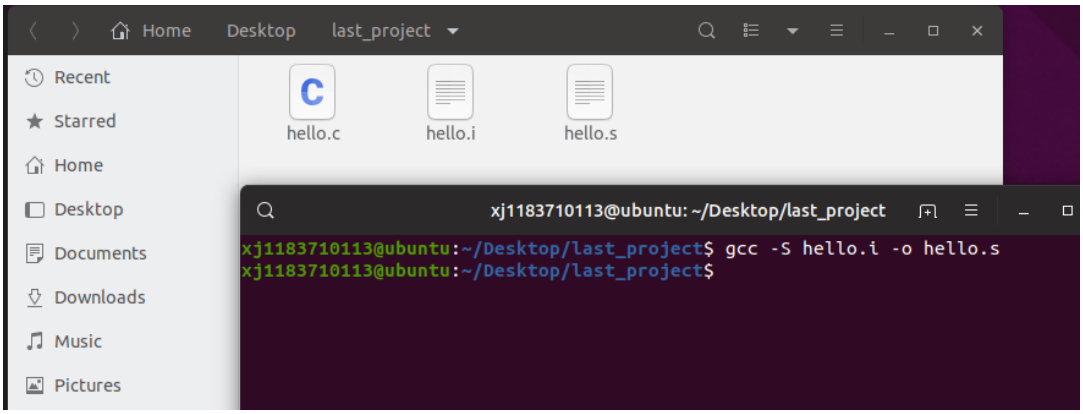


图 3.1 hello.i 编译生成 hello.s

## 3.3 Hello 的编译结果解析

### 3.3.1 数据

1. 字符串

程序中用到的字符串有：“用法: Hello 学号 姓名 秒数！\n”和“Hello %s %s\n”。编译器一般将字符串存放在.rodata 节，这两个个字符串在 hello.s 中的存储如下图，可以看到第一个字符串中的汉字被编码成 UTF-8 格式，一个汉字占三个字节，每个字节用\分隔。第二个字符串中的两个%s 为用户在终端运行 hello 时输入的两个参数。

```
3     .section    .rodata
4     .align 8
5 .LC0:
6     .string "\347\224\250\346\263\225: Hello \345\255\246\345\217\267
  \345\247\223\345\220\215 \347\247\222\346\225\260\357\274\201"
7 .LC1:
8     .string "Hello %s %s\n"
```

图 3.2 hello.s 存储的两个字符串

2. 整数

hello.c 中的整型变量有 argc 和 i。

其中 argc 是从终端传入的参数个数，也是 main 函数的第一个参数，所以由寄存器%edi 进行保存。由图 3.3 的 21 行可知，argc 又被存入了栈中-20(%rbp)的位置。

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

图 3.3 argc 被保存在栈中

i 则是局部变量，用来控制循环次数的计数器，编译器会将局部变量保存在寄存器或者栈中，由图 3.4 的 30 行看出 hello.s 将 i 存储在栈中-4(%rbp)的位置。

```
29 .L2:
30     movl    $0, -4(%rbp)
31     jmp .L3
```

图 3.4 .L2 中的 i 的位置

3. 数组

hello.c 中数组是 main 函数的第二个参数，char *argv[]，是字符指针数组，由于是第二个参数因而被保存在寄存器%rsi 中，由图 3.5 的第 22 行可知它随后又被保存在了栈中-32(%rbp)的位置。

```
21     movl    %edi, -20(%rbp)
22     movq    %rsi, -32(%rbp)
23     cmpl    $4, -20(%rbp)
```

图 3.5 argv 被保存在栈中

在访问 argv[]所指向的内容时，每次先获得数组的起始地址，如图 3.6 的第 33、36、43 行，然后通过加 8*i 来访问之后的字符指针，如图 3.6 中的第 34、37、44，原因是每个字符指针所占的空间大小围为 8 个字节。然后通过获得的字符指针寻找字符串，如图 3.6 中的第 35、38、45 行。

```
32 .L4:
33     movq    -32(%rbp), %rax
34     addq    $16, %rax
35     movq    (%rax), %rdx
36     movq    -32(%rbp), %rax
37     addq    $8, %rax
38     movq    (%rax), %rax
39     movq    %rax, %rsi
40     leaq    .LC1(%rip), %rdi
41     movl    $0, %eax
42     call    printf@PLT
43     movq    -32(%rbp), %rax
44     addq    $24, %rax
45     movq    (%rax), %rax
46     movq    %rax, %rdi
47     call    atoi@PLT
48     movl    %eax, %edi
49     call    sleep@PLT
50     addl    $1, -4(%rbp)
```

图 3.6 访问 argv 数组元素

### 3.3.2 赋值

hello.c 中的赋值操作只有 i=0 这一条，这条语句在汇编中用 mov 指令实现，由于 int 占 4 个字节，所以以‘l’作为后缀。如图 3.7 中的第 30 行。

```
29 .L2:
30     movl    $0, -4(%rbp)
31     jmp .L3
```

图 3.7 给 i 赋值

### 3.3.3 类型转换

程序中涉及的类型转换只有一处，如图 3.8 所示的第 19 行，使用 atoi 函数将命令行的第三个字符串参数转换成了整型。

```
10 int main(int argc,char *argv[]){
11     int i;
12
13     if(argc!=4){
14         printf("用法: Hello 学号 姓名 秒数！\n");
15         exit(1);
16     }
17     for(i=0;i<8;i++){
18         printf("Hello %s %s\n",argv[1],argv[2]);
19         sleep(atoi(argv[3]));
20     }
21     getchar();
22     return 0;
23 }
```

图 3.8 hello.c 的 main 函数

### 3.3.4 算术操作

汇编语言中有如下几种算术操作：

| 指令 | 行为 | 描述 |
|---|---|---|
| inc D | D=D+1 | 加 1 |
| dec D | D=D-1 | 减 1 |
| neg D | D=-D | 取反 |
| add S,G | D=D+S | D 加 S |
| sub S,D | D=D-S | D 减 S |
| imul S,D | D=D*S | D 乘 S |
| imulq S | R[%rdx]:R[%rax]=S*R[%rax] | 有符号乘法 |
| mulq S | R[%rdx]:R[%rax]=S*R[%rax] | 无符号乘法 |
| idivq S | R[%rdx]=R[%rdx]:R[%rax] mod S<br>R[%rax]=R[%rdx]:R[%rax] div S | 有符号除法 |
| divq S | R[%rdx]=R[%rdx]:R[%rax] mod S<br>R[%rax]=R[%rdx]:R[%rax] div S | 无符号触发 |
| leaq S,D | D = &S | 加载有效地址 |

helo.c 中的算术操作只有一处，循环变量 i 的自增运算，在 hello.s 中处理成如图 3.9 的形式。

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

图 3.9 i 的自增运算

### 3.3.5 关系操作

C 语言中的关系操作有==、!=、>、<、>=、<=，这些操作在汇编语言中主要依赖于 cmp 和 test 指令实现，cmp 指令根据两个操作数之差来设置条件码。cmp 指令与 SUB 指令的行为是一样，而 test 指令的行为与 and 指令一样，除了它们只设置条件码而不改变目的寄存器的值。

在 hello.c 中有两处用到了关系操作，分别是图 3.8 中的第 13 行的 argc！=4 和第 17 行的 i<8。这两句在 hello.s 中被分别处理为图 3.10 和图 3.11 的形式。cmp 之后设置条件码，为之后的 je 和 jle 提供判断依据。

```
23      cmpl    $4, -20(%rbp)
24      je  .L2
```

图 3.10 argc!=4 在 hello.s 中的体现

```
52      cmpl    $7, -4(%rbp)
53      jle .L4
```

图 3.11 i<8 在 hello.s 中的体现

### 3.3.6 数组/指针/结构操作

在 hello.c 中通过下标访问 argv 数组，在 hello.s 中访问 argv[1]的操作如图 3.12 所示，第 36 行是取 argv 首地址，第 37 行是通过首地址加 8 字节找到 argv[1]的地址，第 38 行是通过 argv[1]中的内容找到对应的字符串，保存在寄存器%rax 中。对 argv 数组其他元素所指的字符串也同理。

```
36      movq    -32(%rbp), %rax
37      addq    $8, %rax
38      movq    (%rax), %rax
```

图 3.12 访问 argv[1]所指的字符串

### 3.3.7 控制转移

程序涉及到的控制转移有两处。

第一处是判断 argc 是否与 4 相等，在 hello.s 中如图 3.13 所示，第 23 行 cmpl 比较 argc 和 4 设置条件码之后，第 24 行通过判断条件码 ZF 位是否为零决定是否跳转到.L2，如果为 0，说明 argc 等于 4，代码跳转到.L2 继续执行，如果不为 0，则执行图中第 25 行的指令。

```
23      cmpl    $4, -20(%rbp)
24      je  .L2
25      leaq    .LC0(%rip), %rdi
26      call    puts@PLT
27      movl    $1, %edi
28      call    exit@PLT
29 .L2:
```

图 3.13 对 if 语句的处理

第二处是判断循环变量 i 是否满足循环条件 i<8。如图 3.14 所示，在第 30 行循环变量 i 被初始化为 0，第 30 行无条件跳转到.L3，进入循环判断，在 52 行 cmpl 比较 i 和 7 之后设置条件码，然后第 53 行判断是否满足 i<=7 的要求，如果满足，跳转到.L4 执行循环体，如果不满足，则退出循环，执行第 54 行的指令。

```
29 .L2:
30     movl    $0, -4(%rbp)
31     jmp .L3
32 .L4:
33     movq    -32(%rbp), %rax
34     addq    $16, %rax
35     movq    (%rax), %rdx
36     movq    -32(%rbp), %rax
37     addq    $8, %rax
38     movq    (%rax), %rax
39     movq    %rax, %rsi
40     leaq    .LC1(%rip), %rdi
41     movl    $0, %eax
42     call    printf@PLT
43     movq    -32(%rbp), %rax
44     addq    $24, %rax
45     movq    (%rax), %rax
46     movq    %rax, %rdi
47     call    atoi@PLT
48     movl    %eax, %edi
49     call    sleep@PLT
50     addl    $1, -4(%rbp)
51 .L3:
52     cmpl    $7, -4(%rbp)
53     jle .L4
54     call    getchar@PLT
```

图 3.14 对 for 循环的处理

### 3.3.8 函数操作

函数是一种过程，提供了一种封装代码的方式。P 调用 Q 时有如下行为：

传递控制：开始执行 Q 的时候，PC 必须设置为 Q 的代码的起始地址，而在返回时要把 PC 设置为 P 中调用 Q 之后一条语句的地址。

传递数据：P 能够向 Q 传递任意个数的参数，Q 能够向 P 返回 0 或 1 个值。P 向 Q 传递参数时，64 为程序参数存储顺序如下表：

| 第一个 | 第二个 | 第三个 | 第四个 | 第五个 | 第六个 | 第七个及之后 |
|---|---|---|---|---|---|---|
| %rdi | %rsi | %rdx | %rcx | %r8 | %r9 | 栈中 |

分配和释放内存：开始 Q 时为 Q 分配必要的空间，而在 Q 返回前需要把已分配给 Q 的空间释放。

程序中涉及的函数有 6 个。

1. main 函数

main 函数被保存在.text 节，程序运行时，由系统启动调用 main 函数，mian 函数的两个参数分别是由命令行传入的 argc 和 argv[]，分别被保存在%rdi 和%rsi 中。

2. printf 函数

hello.c 有两处调用了 printf 函数，第一个 printf 函数由于只有一个参数，所以被编译器优化为 puts 函数，如图 3.15。参数被保存在寄存器%rdi 中

```
25      leaq    .LC0(%rip), %rdi
26      call    puts@PLT
```

图 3.15 第一个 printf 函数

第二个 printf 函数有三个参数，从内存中取出参数之后，如图 3.16 红线部分，第三、二、一个参数分别被保存在寄存器%rdx、%rsi、%rdi 中。

```
33      movq    -32(%rbp), %rax
34      addq    $16, %rax
35      movq    (%rax), %rdx
36      movq    -32(%rbp), %rax
37      addq    $8, %rax
38      movq    (%rax), %rax
39      movq    %rax, %rsi
40      leaq    .LC1(%rip), %rdi
41      movl    $0, %eax
42      call    printf@PLT
```

图 3.16 第二个 printf 函数

3. exit 函数

hello.c 中在用户输入的不是四个参数时会调用 exit 函数结束程序，在 hello.s 中如图 3.17 所示，把参数 1 用 mov 指令传给%edi，然后调用 exit 函数。

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

图 3.17 exit 函数

4. atoi 函数

hello.c 中通过 atoi 函数把用户输入的第四个参数从字符串转化成整型，对应的汇编代码如图 3.18 所示，第 45 行是取得用户输入的第四个参数，第 46 行把这个参数作为函数 atoi 的参数保存在%rdi 中，然后调用 atoi 函数。

```
43      movq    -32(%rbp), %rax
44      addq    $24, %rax
45      movq    (%rax), %rax
46      movq    %rax, %rdi
47      call    atoi@PLT
```

图 3.18 atoi 函数

5. sleep 函数

sleep 函数的参数是 atoi 函数的返回值，返回值被保存在%eax 中，所以图 3.19 中第 48 行把%eax 中的值传送给%rdi 作为 sleep 函数的参数。

```
48      movl    %eax, %edi
49      call    sleep@PLT
```

图 3.19 sleep 函数

6. getchar 函数

由于 getchar 函数没有参数，所以在退出循环之后直接 call getchar@PLT 即可。

## 3.4 本章小结

本章主要介绍了编译的概念及作用，并且结合 hello.i 编译生成的 hello.s 汇编代码详细阐述了编译器是如何处理 C 语言的各种数据类型、各种运算操作及函数调用。经过本次处理，最初的 C 语言版本的 hello.c 已经被转行成了更加低级的汇编程序。

**（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：汇编器 as 将.s 文件翻译成机器指令，把这些指令打包成一中叫做可重定位目标程序格式，并将结果保存在目标文件中。

汇编的作用：将编译器产生的汇编语言进一步翻译为计算机可以理解的机器语言，生成.o 文件。

## 4.2 在 Ubuntu 下汇编的命令

命令：as hello.s -o hello.o

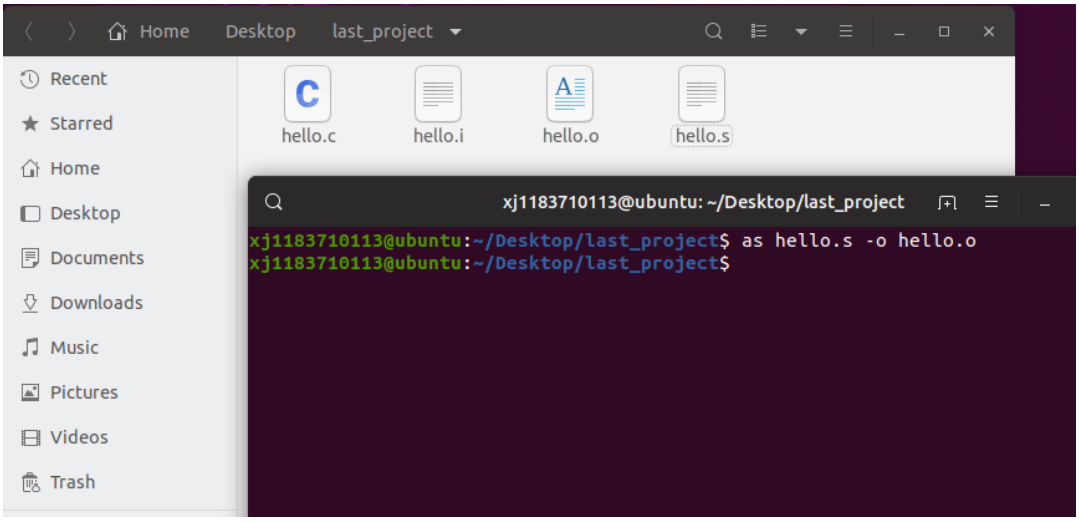


图 4.1 hello.s 汇编生成 hello.o

## 4.3 可重定位目标 elf 格式

使用 readelf -a -W hello.o > hello.elf 命令获得 hello.o 文件 elf 格式，并将结果输出到名为 hello.elf 的文件中。该文件由以下几个部分组成。

1. ELF 头

ELF 头有一个 16 字节的 Magic 序列开始，这个序列描述了生成该文件的系统的字大小和字节顺序。ELF 头剩下的部分包含帮助连接器语法分析和解释目标文件的信息。其中包括 ELF 头的大小、目标文件类型、机器类型、字节头部表的文件偏移，以及节头部表中条目的大小和数量等信息。

```
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 4.2 ELF 头

2. 节头部表

节头部表包含了文件中出现的各个节的含义，包括节的地址、偏移量、大小等信息。如.text 节，地址是从 0x0 开始，偏移量是 0x40，大小是 0x8e。

```
Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        0000000000000000 000040 00008e 00  AX  0   0  1
  [ 2] .rela.text        RELA            0000000000000000 000340 0000c0 18   I 10   1  8
  [ 3] .data             PROGBITS        0000000000000000 0000ce 000000 00  WA  0   0  1
  [ 4] .bss              NOBITS          0000000000000000 0000ce 000000 00  WA  0   0  1
  [ 5] .rodata           PROGBITS        0000000000000000 0000d0 000033 00   A  0   0  8
  [ 6] .comment          PROGBITS        0000000000000000 000103 000024 01  MS  0   0  1
  [ 7] .note.GNU-stack   PROGBITS        0000000000000000 000127 000000 00      0   0  1
  [ 8] .eh_frame         PROGBITS        0000000000000000 000128 000038 00   A  0   0  8
  [ 9] .rela.eh_frame    RELA            0000000000000000 000400 000018 18   I 10   8  8
  [10] .symtab           SYMTAB          0000000000000000 000160 000198 18     11   9  8
  [11] .strtab           STRTAB          0000000000000000 0002f8 000048 00      0   0  1
  [12] .shstrtab         STRTAB          0000000000000000 000418 000061 00      0   0  1
```

图 4.3 节头部表

3. .rela.text 节

存放着代码的重定位条目。当链接器吧这个目标文件和其他文件组合时，会结合这个节，修改.text 节中相应位置的信息。如图 4.4 中的重定位信息依次对应.L0、puts 函数、exit 函数、.L1、printf 函数、atoi 函数、sleep 函数、getchar 函数。

```
Relocation section '.rela.text' at offset 0x340 contains 8 entries:
    Offset             Info             Type               Symbol's Value  Symbol's Name + Addend
0000000000000018  0000000500000002 R_X86_64_PC32          0000000000000000 .rodata - 4
000000000000001d  0000000b00000004 R_X86_64_PLT32         0000000000000000 puts - 4
0000000000000027  0000000c00000004 R_X86_64_PLT32         0000000000000000 exit - 4
0000000000000050  0000000500000002 R_X86_64_PC32          0000000000000000 .rodata + 22
000000000000005a  0000000d00000004 R_X86_64_PLT32         0000000000000000 printf - 4
000000000000006d  0000000e00000004 R_X86_64_PLT32         0000000000000000 atoi - 4
0000000000000074  0000000f00000004 R_X86_64_PLT32         0000000000000000 sleep - 4
0000000000000083  0000001000000004 R_X86_64_PLT32         0000000000000000 getchar - 4
```

图 4.4 .rela.text 节

.rela.text 包含的信息有如下几部分：

| | |
|---|---|
| Offset | 需要进行重定向的代码在.text 或.data 节中的偏移位置，8 个字节。 |
| Info | 包括 symbol 和 type 两部分，其中 symbol 占前 4 个字节，type 占后 4 个字节，symbol 代表重定位到的目标在.symtab 中的偏移量，type 代表重定位的类型。 |
| Addend | 有符号常数，一些类型的重定位要使用它对被修改引用的值做偏移调整。 |
| Type | 重定位到的目标的类型。 |
| Name | 重定向到的目标的名称。 |

重定位一个使用 32 位 PC 相对地址的引用，计算重定位目标地址的方法如下：

先计算指向原位置 src 的指针：refptr=s+r.offset

再计算 src 的运行时地址：refaddr=ADDR(s)+r.offset

将 src 处设置为运行值：*refptr=(unsigned)(ADDR(r.symbol)+r.addend-refaddr)

在 hexedit 中查看 hello.o，图 4.5 是.L1 的重定位条目 r，可以得到以下的信息：r.offset=0x18，r.symbol=.rodata，r.type=R_X86_64_PC32，r.addend=-4。

```
000002F0   00 00 00 00  00 00 00 00  00 68 65 6C  6C 6F 2E 63  .........hello.c
00000300   00 6D 61 69  6E 00 5F 47  4C 4F 42 41  4C 5F 4F 46  .main._GLOBAL_OF
00000310   46 53 45 54  5F 54 41 42  4C 45 5F 00  70 75 74 73  FSET_TABLE_.puts
00000320   00 65 78 69  74 00 70 72  69 6E 74 66  00 61 74 6F  .exit.printf.ato
00000330   69 00 73 6C  65 65 70 00  67 65 74 63  68 61 72 00  i.sleep.getchar.
00000340   18 00 00 00  00 00 00 00  02 00 00 00  05 00 00 00  ................
00000350   FC FF FF FF  FF FF FF FF  1D 00 00 00  00 00 00 00  ................
00000360   04 00 00 00  0B 00 00 00  FC FF FF FF  FF FF FF FF  ................
00000370   27 00 00 00  00 00 00 00  04 00 00 00  0C 00 00 00  '...............
```

图 4.5 hexedit 查看 hello.o

用上述的计算方法计算 r 的重定位之后的运行值信息，再把*refptr 写到 src 处，完成.L1 的重定位。

4. .rela.eh_frame 节

.eh_frame 节的重定位信息。

5. .symtab 节

符号表，用来存放程序中的定义和引用函数的全局变量的信息。重定位需要引用的符号都在其中声明。name 是字符串表中的字节偏移，指向符号的以 null 结尾的字符串名字，value 是符号的地址，对于可重定位的模块来说，value 是距定义目

标的节的起始位置的偏移。对于可执行目标文件来说，该值是一个绝对运行时地址，size 是目标的大小，type 通常要么是数据，要么是函数，binding 表示符号是本地的还是全局的。ABS 代表不该被重定位的符号，UNDEF 代表未定义的符号，也就是在本目标模块中引用，但是却在其他地方定义的符号，COMMON 表示还未被分配位置的未初始化的数据目标。

```
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   142 FUNC    GLOBAL DEFAULT    1 main
    10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _GLOBAL_OFFSET_TABLE_
    11: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
    12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND exit
    13: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
    14: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND atoi
    15: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND sleep
    16: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND getchar
```

图 4.6 符号表

## 4.4 Hello.o 的结果解析

使用命令 objdump -d -r hello.o > hello.ob 将 hello.o 的反汇编代码输出到名为 hello.ob 的文件中。与 hello.s 对比如图 4.7 所示。

1.机器语言的构成

机器语言是计算机能直接理解的语言，完全由二进制数构成，为了阅读的方便显示成了 16 进制。每两个 16 进制数构成一个字节编码，是机器语言中能解释一个运算符或操作数的最小单位。

机器语言由三种数据构成。一是操作码，它具体说明了操作的性质和功能，每一条指令都有一个相应的操作码，计算机通过识别该操作码来完成不同的操作；二是操作数的地址，CPU 通过地址取得所需的操作数；三是操作结果的存储地址，把对操作数的处理所产生的结果保存在该地址中，以便再次使用。

2.机器语言与汇编语言的映射关系

由图 4.7 可以看出，机器语言反汇编得到的汇编代码与直接生成的 hello.s 的代码大致相同，只在以下几个部分中存在差别：

（1）分支转移：反汇编得到的代码中，跳转指令的操作数使用的不再是如 hello.s 中的.L2、.L3 之类的代码段名称，而是具体的地址，因为这类名称只是在编写 hello.s 时为了便于编写所使用的一些符号，这些符号在汇编成机器语

言之后不再存在，变成了语句地址，所以跳转指令的操作数也随之发生了变化。

（2）函数调用：在 hello.s 中，调用函数的形式是 call 指令加调用的函数名，如图 4.7 中左第 28 行，而在反汇编文件中是 call 加下一条指令的地址，如图 4.7 中右第 20 行。由于 hello.c 所调用的函数都是函数共享库中的函数，所以在调用这类函数时会产生重定位条目，这些条目在动态链接时会被修改为运行时的执行地址，而在汇编成的机器语言中，对于这些函数调用的相对地址全部被设置成 0，所以 call 后面加的是下一条指令，而它的重定位信息则会被添加到.rela.text 节，等链接后再确定。

（3）访问字符串常量：在 hello.s 中，使用.L0(%rip)的形式访问，而在反汇编文件中使用 0x0(%rip)的方式访问。因为.rodata 节中的地址也是没有确定的，在运行的时才会确定，所以需要重定位，同函数调用的处理方式一样，也是将其设置为 0，并把重定位信息则会被添加到.rela.text 节，等链接后再确定。

```
12 main:
13 .LFB6:
14     .cfi_startproc
15     pushq   %rbp
16     .cfi_def_cfa_offset 16
17     .cfi_offset 6, -16
18     movq    %rsp, %rbp
19     .cfi_def_cfa_register 6
20     subq    $32, %rsp
21     movl    %edi, -20(%rbp)
22     movq    %rsi, -32(%rbp)
23     cmpl    $4, -20(%rbp)
24   ① je  .L2
25   ③ leaq    .LC0(%rip), %rdi
26     call    puts@PLT
27     movl    $1, %edi
28   ② call    exit@PLT
29 .L2:
30     movl    $0, -4(%rbp)
31     jmp .L3
32 .L4:
33     movq    -32(%rbp), %rax
34     addq    $16, %rax
35     movq    (%rax), %rdx
36     movq    -32(%rbp), %rax
37     addq    $8, %rax
38     movq    (%rax), %rax
39     movq    %rax, %rsi
40     leaq    .LC1(%rip), %rdi
41     movl    $0, %eax
42     call    printf@PLT
43     movq    -32(%rbp), %rax
44     addq    $24, %rax
45     movq    (%rax), %rax
46     movq    %rax, %rdi
47     call    atoi@PLT
48     movl    %eax, %edi
49     call    sleep@PLT
50     addl    $1, -4(%rbp)
51 .L3:
52     cmpl    $7, -4(%rbp)
53     jle .L4
54     call    getchar@PLT
55     movl    $0, %eax
56     leave
57     .cfi_def_cfa 7, 8
58     ret
```

```
 7 0000000000000000 <main>:
 8    0:   55                      push   %rbp
 9    1:   48 89 e5                mov    %rsp,%rbp
10    4:   48 83 ec 20             sub    $0x20,%rsp
11    8:   89 7d ec                mov    %edi,-0x14(%rbp)
12    b:   48 89 75 e0             mov    %rsi,-0x20(%rbp)
13    f:   83 7d ec 04             cmpl   $0x4,-0x14(%rbp)
14   13:   74 16                   je     2b <main+0x2b>
15   15:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi
16                 18: R_X86_64_PC32  .rodata-0x4
17   1c:   e8 00 00 00 00          callq  21 <main+0x21>
18                 1d: R_X86_64_PLT32 puts-0x4
19   21:   bf 01 00 00 00          mov    $0x1,%edi
20   26:   e8 00 00 00 00          callq  2b <main+0x2b>
21                 27: R_X86_64_PLT32 exit-0x4
22   2b:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
23   32:   eb 48                   jmp    7c <main+0x7c>
24   34:   48 8b 45 e0             mov    -0x20(%rbp),%rax
25   38:   48 83 c0 10             add    $0x10,%rax
26   3c:   48 8b 10                mov    (%rax),%rdx
27   3f:   48 8b 45 e0             mov    -0x20(%rbp),%rax
28   43:   48 83 c0 08             add    $0x8,%rax
29   47:   48 8b 00                mov    (%rax),%rax
30   4a:   48 89 c6                mov    %rax,%rsi
31   4d:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi
32                 50: R_X86_64_PC32  .rodata+0x22
33   54:   b8 00 00 00 00          mov    $0x0,%eax
34   59:   e8 00 00 00 00          callq  5e <main+0x5e>
35                 5a: R_X86_64_PLT32 printf-0x4
36   5e:   48 8b 45 e0             mov    -0x20(%rbp),%rax
37   62:   48 83 c0 18             add    $0x18,%rax
38   66:   48 8b 00                mov    (%rax),%rax
39   69:   48 89 c7                mov    %rax,%rdi
40   6c:   e8 00 00 00 00          callq  71 <main+0x71>
41                 6d: R_X86_64_PLT32 atoi-0x4
42   71:   89 c7                   mov    %eax,%edi
43   73:   e8 00 00 00 00          callq  78 <main+0x78>
44                 74: R_X86_64_PLT32 sleep-0x4
45   78:   83 45 fc 01             addl   $0x1,-0x4(%rbp)
46   7c:   83 7d fc 07             cmpl   $0x7,-0x4(%rbp)
47   80:   7e b2                   jle    34 <main+0x34>
48   82:   e8 00 00 00 00          callq  87 <main+0x87>
49                 83: R_X86_64_PLT32 getchar-0x4
50   87:   b8 00 00 00 00          mov    $0x0,%eax
51   8c:   c9                      leaveq
52   8d:   c3                      retq
```

图 4.7 hello.s（左）与 hello.ob（右）的对比

## 4.5 本章小结

本章主要介绍了从 hello.s 到 hello.o 的汇编过程，通过查看 hello.o 的 elf 格式和使用 objdump 得到反汇编代码与 hello.s 进行比较，了解了从汇编语言映射到机器语言汇编器需要实现的转换。

**（第 4 章 1 分）**

# 第 5 章 链接

## 5.1 链接的概念与作用

链接的概念：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，这个文件可被加载到内存并执行。在现代系统中，链接是由较做链接器的程序自动执行的。

链接的作用：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模块。当我们改变这些模块中的一个时，只需简单地重新编译它，并重新链接应用，而不必重新编译其他文件。

## 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

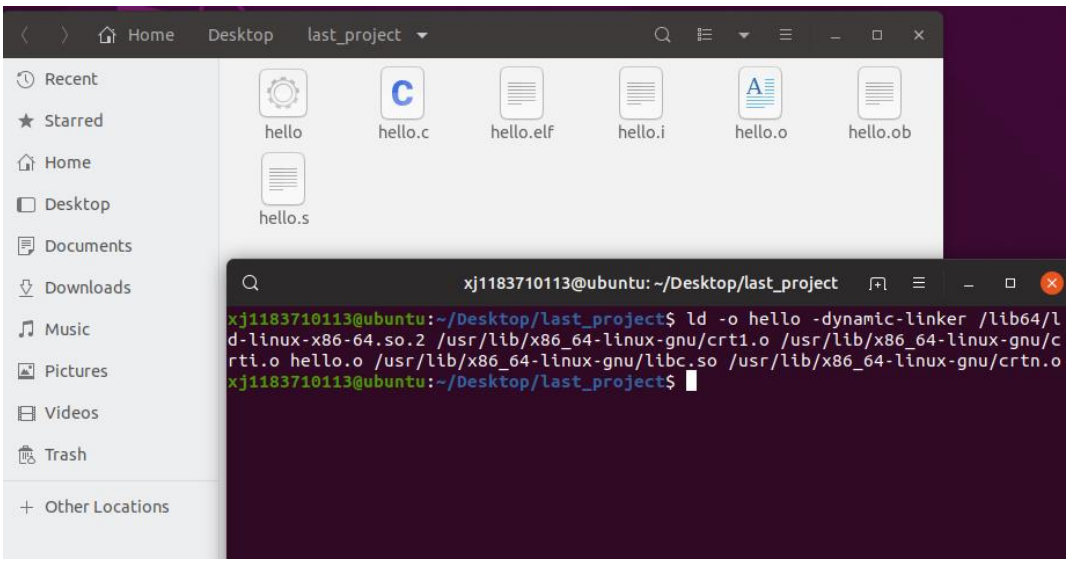


图 5.1 hello.o 与库文件链接生成 hello

## 5.3 可执行目标文件 hello 的格式

使用 readelf -a -W hello 命令查看 hello 的 elf 格式，其中的节头表部分如图 5.2 所示。节头表记录了各个节的信息，Address 是程序被载入到虚拟地址的起始地址，

off 是在程序中的偏移量，size 是节的大小。

```
Section Headers:
  [Nr] Name              Type             Address           Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL             0000000000000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS         0000000000400270 000270 00001c 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE             000000000040028c 00028c 000020 00   A  0   0  4
  [ 3] .hash             HASH             00000000004002b0 0002b0 000038 04   A  5   0  8
  [ 4] .gnu.hash         GNU_HASH         00000000004002e8 0002e8 00001c 00   A  5   0  8
  [ 5] .dynsym           DYNSYM           0000000000400308 000308 0000d8 18   A  6   1  8
  [ 6] .dynstr           STRTAB           00000000004003e0 0003e0 00005c 00   A  0   0  1
  [ 7] .gnu.version      VERSYM           000000000040043c 00043c 000012 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED          0000000000400450 000450 000020 00   A  6   1  8
  [ 9] .rela.dyn         RELA             0000000000400470 000470 000030 18   A  5   0  8
  [10] .rela.plt         RELA             00000000004004a0 0004a0 000090 18  AI  5  19  8
  [11] .init             PROGBITS         0000000000401000 001000 000017 00  AX  0   0  4
  [12] .plt              PROGBITS         0000000000401020 001020 000070 10  AX  0   0 16
  [13] .text             PROGBITS         0000000000401090 001090 000121 00  AX  0   0 16
  [14] .fini             PROGBITS         00000000004011b4 0011b4 000009 00  AX  0   0  4
  [15] .rodata           PROGBITS         0000000000402000 002000 00003b 00   A  0   0  8
  [16] .eh_frame         PROGBITS         0000000000402040 002040 0000fc 00   A  0   0  8
  [17] .dynamic          DYNAMIC          0000000000403e50 002e50 0001a0 10  WA  6   0  8
  [18] .got              PROGBITS         0000000000403ff0 002ff0 000010 08  WA  0   0  8
  [19] .got.plt          PROGBITS         0000000000404000 003000 000048 08  WA  0   0  8
  [20] .data             PROGBITS         0000000000404048 003048 000004 00  WA  0   0  1
  [21] .comment          PROGBITS         0000000000000000 00304c 000023 01  MS  0   0  1
  [22] .symtab           SYMTAB           0000000000000000 003070 000498 18     23  28  8
  [23] .strtab           STRTAB           0000000000000000 003508 000158 00      0   0  1
  [24] .shstrtab         STRTAB           0000000000000000 003660 0000c5 00      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 5.2 hello 的 elf 格式的 Section Headers

## 5.4 hello 的虚拟地址空间

用 edb 查看程序 hello，发现程序在地址 0x400000~0x401000 中被载入，从 0x400000 开始到 0x400fff 结束，这之间每个节的排列同图 5.2 中 Address 中声明。在 0x400fff 之后存放的是.dynamic~.shstrtab 节。

在 Data Dump 中查看地址 0x400000 开始的内容，可以看到开头是 ELF 头部分。如图 5.3。

```
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 90 10 40 00 00 00 00 00 ..>.......@.....
00000000:00400020 40 00 00 00 00 00 00 00 28 37 00 00 00 00 00 00 @.......(7......
00000000:00400030 00 00 00 00 40 00 38 00 0a 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 30 02 00 00 00 00 00 00 30 02 00 00 00 00 00 00 0.......0.......
```

图 5.3 在 Data Dump 中查看地址 0x400000

查看地址 0x0x400200，发现是.interp 节，保存着 linux 动态共享库的路径。如图 5.4。

```
00000000:00400270 2f 6c 69 62 36 34 2f 6c 64 2d 6c 69 6e 75 78 2d /lib64/ld-linux-
00000000:00400280 78 38 36 2d 36 34 2e 73 6f 2e 32 00 04 00 00 00 x86-64.so.2.....
00000000:00400290 10 00 00 00 01 00 00 00 47 4e 55 00 00 00 00 00 ........GNU.....
00000000:004002a0 03 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 ................
```

图 5.4 在 Data Dump 中查看地址 0x400270

查看地址 0x0x400308，发现是.dynsym 节，保存动态符号表。如图 5.5。

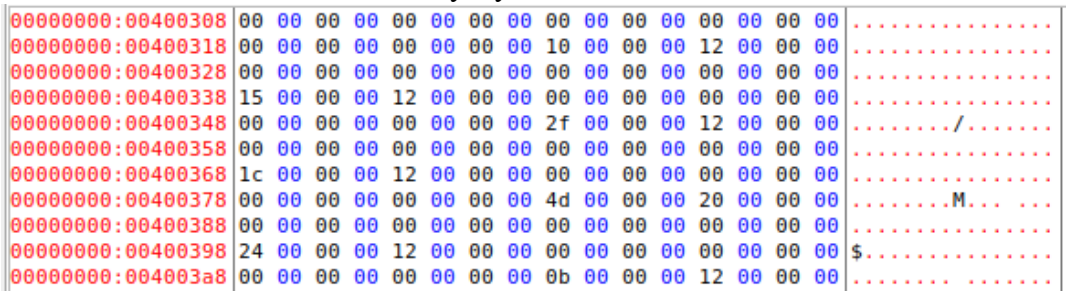


图 5.5 在 Data Dump 中查看地址 0x400308

查看地址 0x0x402000，发现是.rodata 节，其中保存着 hello.c 中的两个字符串。如图 5.6。

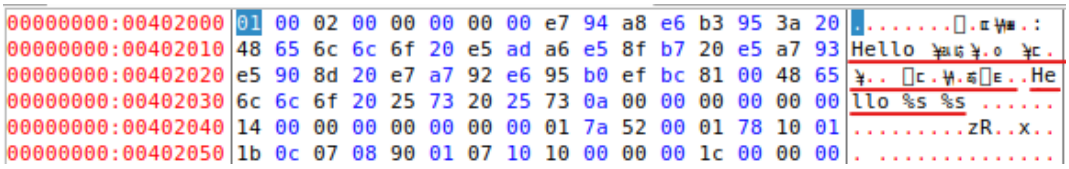


图 5.6 在 Data Dump 中查看地址 0x402000

在图 5.2 中的其他节也都能够通过对应的 Address 在 Data Dump 中找到，这里就不一一列举了。

## 5.5 链接的重定位过程分析

使用命令 objdump -d -r hello > hello1.ob 生成 hello 的反汇编文件。

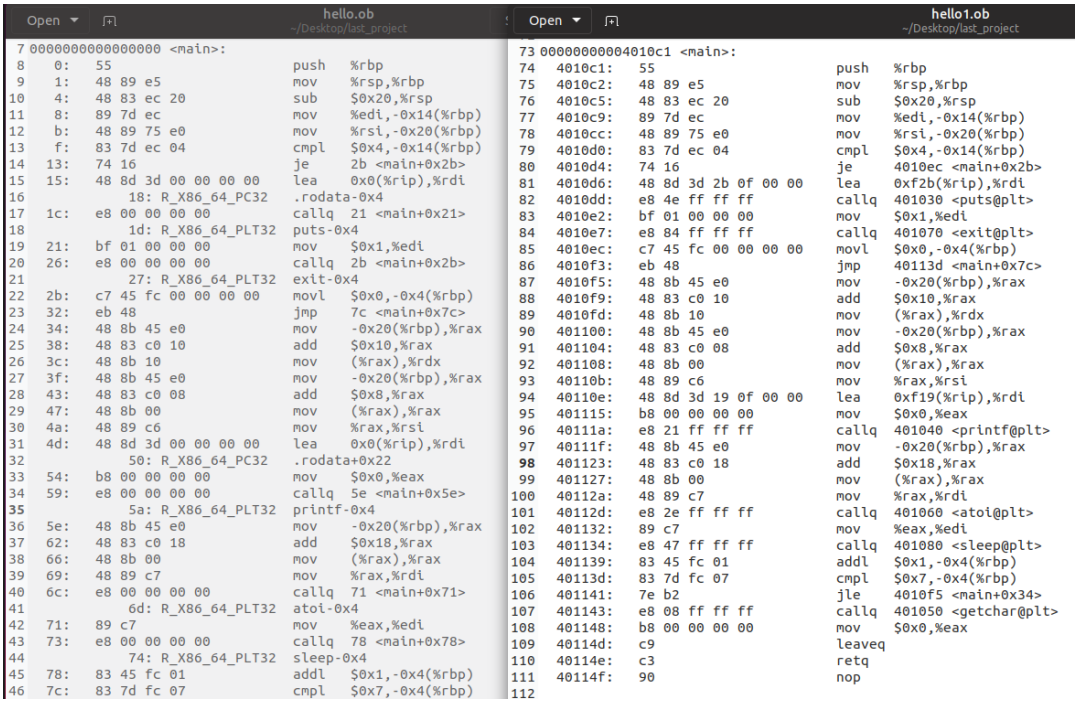


图 5.7 hello.ob 和 hello1.ob 的 mian 函数对比

对于 hello.ob 和 hello1.ob 来说，两者 main 函数的汇编指令完全相同，除了地址由相对偏移变成了可以由 CPU 直接寻址的绝对地址。链接器把 hello.o 中的偏移量加上程序在虚拟内存中的起始地址 0x400000 和.text 节的偏移量就得到了 hello1.ob 中的地址。函数内的控制转移即 jmp 指令后的地址由偏移量变为了偏移量+函数的起始地址；call 后的地址由链接器执行重定位后计算出实际地址。

函数调用：链接器解析重定条目时发现对外部函数调用的类型为 R_X86_64_PLT32 的重定位，此时动态链接库中的函数已经加入到了 PLT 中，.text 与.plt 节相对距离已经确定，链接器计算相对距离，将对动态链接库中函数的调用值改为 PLT 中相应函数与下条指令的相对地址，指向对应函数。对于此类重定位链接器为其构造.plt 与.got.plt。

.rodata 引用：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。这里以计算第一条字符串相对地址为例说明计算相对地址的算法：

refptr=s+r.offset=Pointerto0x4010dd

refaddr=ADDR(s)+r.offset=ADDR(main)+r.offset=0x4010c1+0x1c=0x4010dd

*refptr=(unsigned)(ADDR(r.symbol)+r.addend-refaddr)

=ADDR(str1)+r.addend-refaddr

=0x402008-0x4010dd=(unsigned)0xf2b

在汇编中验证如图 5.8。

```
4010d6:   48 8d 3d 2b 0f 00 00    lea    0xf2b(%rip),%rdi    |  # 402008 <_IO_stdin_used+0x8>
4010dd:   e8 4e ff ff ff          callq  401030 <puts@plt>
```

图 5.8 hello1.ob 引用.rodata 中的第一个字符串

除了 main 函数，hello1.ob 比 hello.ob 多出了几个函数：printf、sleep、puts、getchar、atoi、exit。

除了.text 节的区别外，hello1.ob 比 hello.ob 多出了几个节：.init 节、.plt 节、.fini 节。其中.init 节是程序初始化需要执行的代码，.fini 节是程序正常终止时需要执行的代码，.plt 节是动态链接中的过程链接表。

## 5.6 hello 的执行流程

子程序名：

ld-2.27.so!_dl_start

ld-2.27.so!_dl_init

hello!_start

libc-2.27.so!__libc_start_main
libc-2.27.so!__cxa_atexit
libc-2.27.so!__libc_csu_init
libc-2.27.so!_setjmp
hello!main
hello!puts@plt
hello!exit@plt
hello!printf@plt
hello!sleep@plt
hello!getchar@plt
ld-2.27.so!_dl_runtime_resolve_xsave
ld-2.27.so!_dl_fixup
ld-2.27.so!_dl_lookup_symbol_x
libc-2.27.so!exit

## 5.7 Hello 的动态链接分析

对于动态共享链接库中 PIC 函数，编译器没有办法预测函数的运行时地址，所以需要添加重定位记录，等待动态链接器处理，为避免运行时修改调用模块的代码段，链接器采用延迟绑定的策略。动态链接器使用过程链接表 PLT+全局偏移量表 GOT 实现函数的动态链接，GOT 中存放函数目标地址，PLT 使用 GOT 中地址跳转到目标函数。

在 dl_init 调用之前，对于每一条 PIC 函数调用，调用的目标地址都实际指向 PLT 中的代码逻辑，GOT 存放的是 PLT 中函数调用指令的下一条指令地址。由图 5.2 可知，.got.plt 的起始地址是 0x404000，其内容如图 5.9。

```
00000000:00404000 50 3e 40 00 00 00 00 00 00 00 00 00 00 00 00 00 P>@.............
00000000:00404010 00 00 00 00 00 00 00 00 36 10 40 00 00 00 00 00 ........6.@.....
00000000:00404020 46 10 40 00 00 00 00 00 56 10 40 00 00 00 00 00 F.@.....V.@.....
00000000:00404030 66 10 40 00 00 00 00 00 76 10 40 00 00 00 00 00 f.@.....v.@.....
00000000:00404040 86 10 40 00 00 00 00 00 00 00 00 00 47 43 43 3a ..@.........GCC:
```

图 5.9 执行 dl_init 前的.got.plt 节

如图 5.10，可以看到调用 dl_init 后 0x404008 和 0x404010 处的两个 8 字节的数据发生改变，出现了两个地址 0x7f85442c2190 和 0x7f85442ad200。这就是 GOT[1]和 GOT[2]。

```
00000000:00404000 50 3e 40 00 00 00 00 00 90 21 2c 44 85 7f 00 00 P>@......!,D....
00000000:00404010 00 d2 2a 44 85 7f 00 00 36 10 40 00 00 00 00 00 ..*D....6.@.....
00000000:00404020 46 10 40 00 00 00 00 00 56 10 40 00 00 00 00 00 F.@.....V.@.....
00000000:00404030 66 10 40 00 00 00 00 00 76 10 40 00 00 00 00 00 f.@.....v.@.....
00000000:00404040 86 10 40 00 00 00 00 00 00 00 00 00 47 43 43 3a ..@.........GCC:
```

图 5.10 执行 dl_init 后的.got.plt 节

如图 5.11 红线部分，其中 GOT[1]指向重定位表（依次为.plt 节需要重定位的函数的运行时地址）用来确定调用的函数地址。

```
00007f85:442c2190 00 00 00 00 00 00 00 00 28 27 2c 44 85 7f 00 00 ........('.D....
00007f85:442c21a0 50 3e 40 00 00 00 00 00 30 27 2c 44 85 7f 00 00 P>@.....0'.D....
00007f85:442c21b0 00 00 00 00 00 00 00 00 90 21 2c 44 85 7f 00 00 .........!.D....
00007f85:442c21c0 00 00 00 00 00 00 00 00 10 27 2c 44 85 7f 00 00 .........'.D....
00007f85:442c21d0 00 00 00 00 00 00 00 00 50 3e 40 00 00 00 00 00 ........P>@.....
00007f85:442c21e0 00 3f 40 00 00 00 00 00 f0 3e 40 00 00 00 00 00 .?@..... >@.....
00007f85:442c21f0 80 3e 40 00 00 00 00 00 a0 3e 40 00 00 00 00 00 .>@..... >@.....
```

图 5.11 0x7f85442c2190 指向的重定位表

如图 5.12，GOT[2]指向的目标程序是动态链接器 ld-linux.so 运行时地址。

```
00007f85:442ad200 53                    push rbx
00007f85:442ad201 48 89 e3              mov rbx, rsp
00007f85:442ad204 48 83 e4 c0           and rsp, 0xfffffffffffffff0
00007f85:442ad208 48 2b 25 79 35 01 00  sub rsp, [rel 0x7f85442c0788]
00007f85:442ad20f 48 89 04 24           mov [rsp], rax
00007f85:442ad213 48 89 4c 24 08        mov [rsp+8], rcx
00007f85:442ad218 48 89 54 24 10        mov [rsp+0x10], rdx
00007f85:442ad21d 48 89 74 24 18        mov [rsp+0x18], rsi
00007f85:442ad222 48 89 7c 24 20        mov [rsp+0x20], rdi
00007f85:442ad227 4c 89 44 24 28        mov [rsp+0x28], r8
00007f85:442ad22c 4c 89 4c 24 30        mov [rsp+0x30], r9
00007f85:442ad231 b8 ee 00 00 00        mov eax, 0xee
00007f85:442ad236 31 d2                 xor edx, edx
00007f85:442ad238 48 89 94 24 50 02 0…  mov [rsp+0x250], rdx
00007f85:442ad240 48 89 94 24 58 02 0…  mov [rsp+0x258], rdx
00007f85:442ad248 48 89 94 24 60 02 0…  mov [rsp+0x260], rdx
00007f85:442ad250 48 89 94 24 68 02 0…  mov [rsp+0x268], rdx
00007f85:442ad258 48 89 94 24 70 02 0…  mov [rsp+0x270], rdx
00007f85:442ad260 48 89 94 24 78 02 0…  mov [rsp+0x278], rdx
00007f85:442ad268 0f                    db 0x0f
00007f85:442ad269 c7                    db 0xc7
00007f85:442ad26a 64 24 40              and al, 0x40
00007f85:442ad26d 48 8b 73 10           mov rsi, [rbx+0x10]
00007f85:442ad271 48 8b 7b 08           mov rdi, [rbx+8]
00007f85:442ad275 e8 76 8d ff ff        call 0x7f85442a5ff0
00007f85:442ad27a 49 89 c3              mov r11, rax
00007f85:442ad27d b8 ee 00 00 00        mov eax, 0xee
00007f85:442ad282 31 d2                 xor edx, edx
00007f85:442ad284 0f ae 6c 24 40        xrstor ptr [rsp+0x40]
00007f85:442ad289 4c 8b 4c 24 30        mov r9, [rsp+0x30]
00007f85:442ad28e 4c 8b 44 24 28        mov r8, [rsp+0x28]
00007f85:442ad293 48 8b 7c 24 20        mov rdi, [rsp+0x20]
00007f85:442ad298 48 8b 74 24 18        mov rsi, [rsp+0x18]
00007f85:442ad29d 48 8b 54 24 10        mov rdx, [rsp+0x10]
00007f85:442ad2a2 48 8b 4c 24 08        mov rcx, [rsp+8]
00007f85:442ad2a7 48 8b 04 24           mov rax, [rsp]
00007f85:442ad2ab 48 89 dc              mov rsp, rbx
00007f85:442ad2ae 48 8b 1c 24           mov rbx, [rsp]
00007f85:442ad2b2 48 83 c4 18           add rsp, 0x18
00007f85:442ad2b6 f2 41 ff e3           bnd jmp r11
00007f85:442ad2ba 66 0f 1f 44 00 00     nop word [rax+rax]
00007f85:442ad2c0 bf 02 00 00 00        mov edi, 2
```

图 5.12 0x7f85442ad200 处的动态链接器

## 5.8 本章小结

本章介绍了链接的概念和作用，对链接后生成的可执行文件 hello 的 elf 格式文件进行了分析，分析了 hello 的虚拟地址空间、重定位过程、执行过程的各种处理操作。

**（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：一个执行中的程序的实例，同时也是系统进行资源分配和调度的基本单位。一般情况下，包括文本区域、数据区域和堆栈。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储着活动过程调用的指令和本地变量。

进程的作用：，它提供一个假象，好像我们的程序独占地使用内存系统，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

shell-bash 的作用：shell-bash 是一个 C 语言程序，它代表用户执行进程，它交互性地解释和执行用户输入的命令，能够通过调用系统级的函数或功能执行程序、建立文件、进行并行操作等。同时它也能够协调程序间的运行冲突，保证程序能够以并行形式高效执行。bash 还提供了一个图形化界面，提升交互的速度。

shell-bash 的处理流程：

(1)终端进程读取用户由键盘输入的命令行。

(2)分析命令行字符串，获取命令行参数，并构造传递给 execve 的 argv 向量

(3)检查第一个命令行参数是否是一个内置的 shell 命令

(3)如果不是内部命令，调用 fork()创建新进程/子进程

(4)在子进程中，用步骤 2 获取的参数，调用 execve()执行指定程序。

(5)如果用户没要求后台运行(命令末尾没有&号）否则 shell 使用 waitpid 等待作业终止后返回。

(6)如果用户要求后台运行(如果命令末尾有&号），则 shell 返回；

## 6.3 Hello 的 fork 进程创建过程

首先，打开 Terminal 输入：./hello 1183710113 许健 1

接下来 shell 会分析这条命令，由于./hello 不是一条内置的命令，于是判断./hello 的语义是执行当前目录下的可执行目标文件 hello，然后 Terminal 会调用 fork 床架一个新的运行的子进程，子进程得到与父进程用户级虚拟地址空间相同的（但是独立的）一份副本，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。父进程与子进程之间的区别在于它们拥有不同的 PID。

流程图如图 6.1。

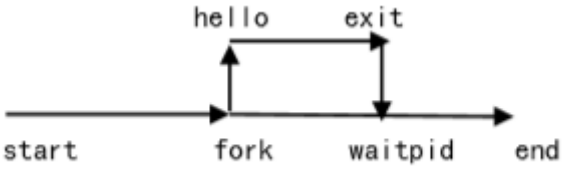


图 6.1 fork 创建子进程流程图

## 6.4 Hello 的 execve 过程

在 fork 之后，子进程调用 execve 函数，execve 函数在新创建的子进程的上下文中加载并运行 hello 程序。execve 函数加载并运行可执行目标文件 filename，且带参数列表 argv 和环境变量列表 envp。只有发生错误时 execve 才会返回到调用程序。所以，execve 调用一次且从不返回。

加载并运行 hello 需要以下几个步骤：

（1）删除已存在的用户区域。删除当前进程虚拟地址的用户部分中已存在的区域结构。

（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。所有这些新的区域都是私有的、写时复制的。代码和数据区被映射为 hello 文件中的.text 和.data 区。bss 区域是请求二进制零的，映射到匿名文件，其大小包含在 hello 中。栈和堆区域也是请求二进制零的，初始长度为零。

（3）映射共享区域。如果 hello 程序与共享对象链接，那么这些对象都是动态链接到这个程序的，然后再映射到用户虚拟地址空间中的共享区域内。

（4）设置程序计数器。设置当前进程上下文中的程序计数器，使之指向代码区域的入口点。下一次调度这个进程时，它将从这个入口点开始执行。

## 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。

系统中的每个程序都运行在某个进程的上下文中。上下文是由程序正确运行所需的状态组成的。这个状态包括存放在内存中的程序的代码和数据，它的栈、通用目的寄存器的内容、程序计数器、环境变量以及打开文件描述符的集合。

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。

处理器通常用某个控制寄存器的一个模式位来提供用户模式和内核模式的功

能。设置了模式位时，进程就运行在内核模式中，该进程可以执行指令集中的任何指令，可以访问系统中的任何内存位置。没有设置模式位时，进程就运行在用户模式中，用户模式中的进程不允许执行特权指令。

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的决定叫做调度。如图 6.2，上下文切换的流程是：1.保存当前进程的上下文。2.恢复某个先前被抢占的进程被保存的上下文。3.将控制传递给这个新恢复的进程。

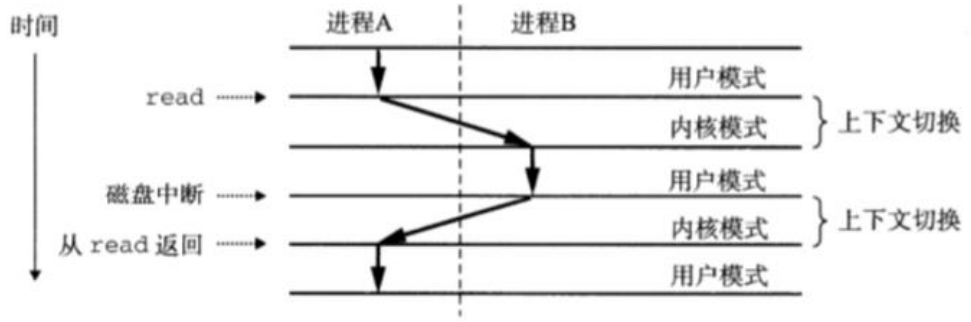


图 6.2 进程的上下文切换

然后分析 hello 的进程调度，hello 在刚开始运行时内核为其保存一个上下文，进程在用户模式下运行，当没有异常或中断信号的产生，hello 将一直正常地执行，而当出现异常或系统中断时，内核将启用调度器休眠当前进程，并在内核模式中完成上下文切换，将控制传递给其他进程。

当程序在执行 sleep 函数时，系统调用显式地请求让调用进程休眠，调度器抢占当前进程，并发生上下文切换，将控制转移到新的进程，此时计时器开始，当计时器达到传入的第四个参数大小（这里是 1s）时，产生一个中断信号，中断当前正在进行的进程，进行上下文切换恢复 hello 的上下文信息，控制会回到 hello 进程中。当循环结束后，程序调用 getchar 函数用 getchar 时，由用户模式进入内核模式，内核中的陷阱处理程序请求来自键盘缓冲区的信号传输，并执行上下文切换把控制转移给其他进程。数据传输结束之后，引发一个中断信号，控制回到 hello 进程中，执行 return，进程终止。

## 6.6 hello 的异常与信号处理

1.hello 执行过程中可能出现四类异常：中断、陷阱、故障和终止。

（1）中断是来自 I/O 设备的信号，异步发生，中断处理程序对其进行处理，返回后继续执行调用前待执行的下一条代码，就像没有发生过中断。

（2）陷阱是有意的异常，是执行一条指令的结果，调用后也会返回到下一条

指令，用来调用内核的服务进行操作。帮助程序从用户模式切换到内核模式。

（3）故障是由错误情况引起的，它可能能够被故障处理程序修正。如果修正成功，则将控制返回到引起故障的指令，否则将终止程序。

（4）终止是不可恢复的致命错误造成的结果，通常是一些硬件的错误，处理程序会将控制返回给一个 abort 例程，该例程会终止这个应用程序。

2.hello 执行过程中可能出现的信号如图 6.3。

| 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 SIGCONT② | 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 6.3 信号

3. hello 对各种信号的处理的分析。

（1）正常运行 hello 程序。结果如图 6.4，可以看出，程序在执行结束后，进程被回收。

```
xj1183710113@ubuntu:~/Desktop/last_project$ ./hello 1183710113 许健 2
Hello 1183710113 许健
Hello 1183710113 许健
Hello 1183710113 许健
Hello 1183710113 许健
Hello 1183710113 许健
Hello 1183710113 许健
Hello 1183710113 许健

xj1183710113@ubuntu:~/Desktop/last_project$ ps
    PID TTY          TIME CMD
   4576 pts/0    00:00:00 bash
   6991 pts/0    00:00:00 ps
xj1183710113@ubuntu:~/Desktop/last_project$
```

图 6.4 正常运行 hello

（2）随便乱按。结果如图 6.5。发现乱按会将输入的内容保存在缓冲区，等进程结束后作为命令行的内容输入。

```
xj1183710113@ubuntu:~/Desktop/last_project$ ./hello 1183710113 许健 2
Hello 1183710113 许健

kHello 1183710113 许健
msdfdds

er
hHello 1183710113 许健
df
fdbdbkHello 1183710113 许健
lfdlbkoddfkHello 1183710113 许健
bdfkbjiodfdbdbd

Hello 1183710113 许健
dfbkdfbkg ssoj sd Hello 1183710113 许健
dso jv sojjdvojsdovHello 1183710113 许健
 sdovjsvsvsv xj1183710113@ubuntu:~/Desktop/last_project$ kmsdfdds
kmsdfdds: command not found
```

图 6.5 运行过程中不停乱按

（3）运行过程中按下 Ctrl-C。结果如图 6.6，发现会向进程发送 SIGINT 信号。信号处理程序终止并回收进程。

```
xj1183710113@ubuntu:~/Desktop/last_project$ ./hello 1183710113 许健 2
Hello 1183710113 许健
Hello 1183710113 许健
Hello 1183710113 许健
^C
xj1183710113@ubuntu:~/Desktop/last_project$ ps
    PID TTY          TIME CMD
   4576 pts/0    00:00:00 bash
   7121 pts/0    00:00:00 ps
```

图 6.6 运行过程中按下 Ctrl-C

（4）运行过程中按下 Ctrl-Z。结果如图 6.7 所示，当按下 Ctrl-Z 之后，shell 进程收到 SIGSTP 信号，信号处理函数的逻辑是打印屏幕回显、将 hello 进程挂起，通过 ps 命令我们可以看出 hello 进程没有被回收，其进程号时 7308，用 jobs 命令看到 job ID 是 1，状态是 Stopped，使用 fg 1 命令将其调到前台，此时 shell 程序首先打印 hello 的命令行命令，然后继续运行打印剩下的信息，之后再按下 Ctrl-Z，将进程挂起。

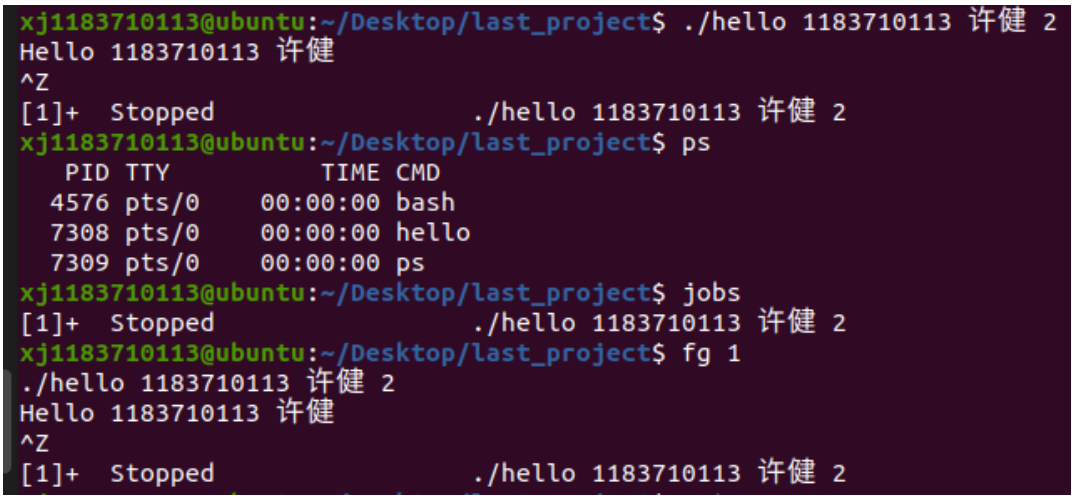


图 6.7 运行过程中按下 Ctrl-Z

此时，用 pstree 查看进程，发现 hello 进程在图 6.8 的位置。

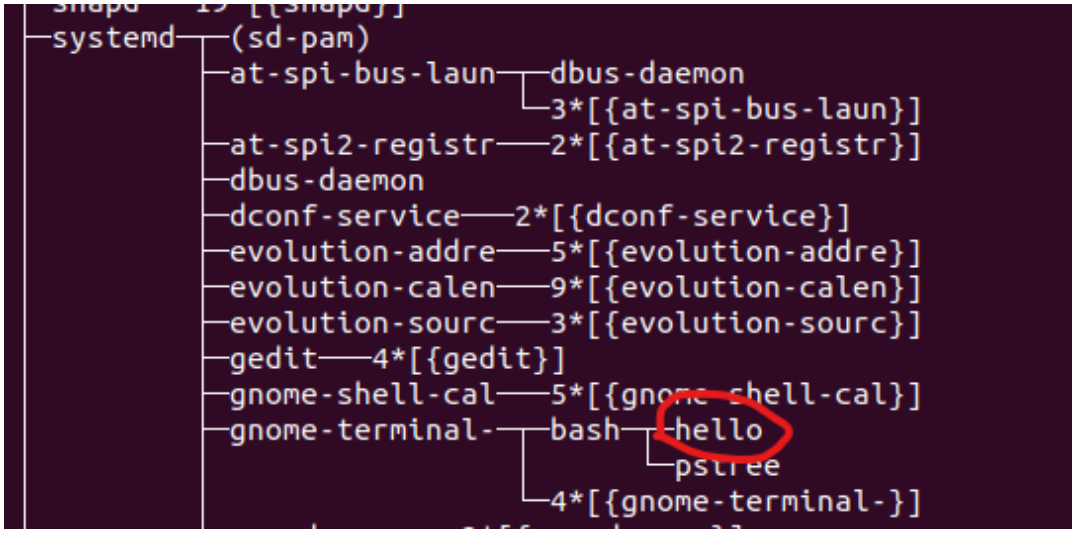


图 6.8 在 pstree 中查看 hello

再输入 kill -9 7308 终止 hello 进程，再用 jobs 命令查看发现刚才的 hello 进程已经被终止了，在 ps 命令下看也没有 hello 进程了，说明进程被终止，然后被回收。如图 6.9。

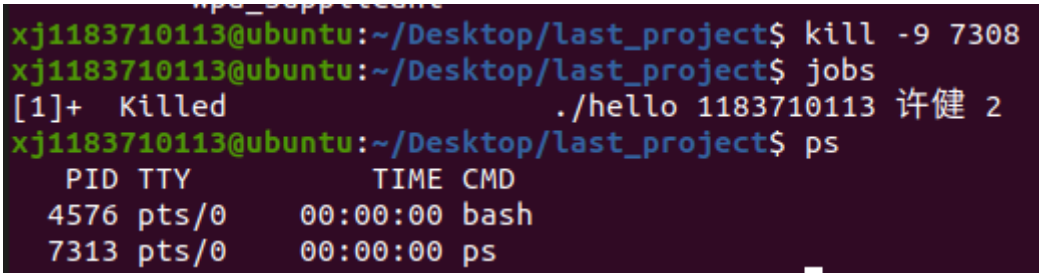


图 6.9 用 kill 命令给 hello 进程发送 SIGINT 信号

## 6.7 本章小结

本章主要介绍了进程的概念与作用，阐述了 shell 的作用和处理流程以及 hello 的 fork 进程的创建过程和 execve 的过程，最后分析了 hello 的执行过程和过程中出现的异常的处理。

**（第 6 章 1 分）**

# 第 7 章 hello 的存储管理

## 7.1 hello 的存储器地址空间

逻辑地址：在有地址变换功能的计算机中，访内指令给出的地址(操作数)叫逻辑地址，也叫相对地址。分为两个部分，一个部分为段基址，另一个部分为段偏移量。

线性地址：逻辑地址经过段机制后转化为线性地址，为描述符:偏移量的组合形式。分页机制中线性地址作为输入。

虚拟地址：CPU 启动保护模式后，程序运行在虚拟地址空间中。与物理地址相似，虚拟内存被组织为一个存放在磁盘上的 N 个连续的字节大小的单元组成的数组，其每个字节对应的地址成为虚拟地址。虚拟地址包括 VPO（虚拟页面偏移量）、VPN（虚拟页号）、TLBI（TLB 索引）、TLBT（TLB 标记）。

物理地址：CPU 通过地址总线的寻址，找到真实的物理内存对应地址。CPU 对内存的访问是通过连接着 CPU 和北桥芯片的前端总线来完成的。在前端总线上传输的内存地址都是物理内存地址。

结合 hello 的反汇编文件，如图 7.1 中的第 34 行面函数的起始地址 0x4010c1，这里的 0x4010c1 是逻辑地址的偏移量部分，偏移量再加上代码段的段地址就得到了 main 函数的虚拟地址（线性地址），虚拟地址是现代系统的一个抽象概念，再经过 MMU 的处理后将得到实际存储在计算机存储设备上的地址。

```
73 00000000004010c1 <main>:
74   4010c1:   55                      push   %rbp
75   4010c2:   48 89 e5                mov    %rsp,%rbp
76   4010c5:   48 83 ec 20             sub    $0x20,%rsp
77   4010c9:   89 7d ec                mov    %edi,-0x14(%rbp)
78   4010cc:   48 89 75 e0             mov    %rsi,-0x20(%rbp)
79   4010d0:   83 7d ec 04             cmpl   $0x4,-0x14(%rbp)
```

图 7.1 hello 反汇编文件中的 main 函数

## 7.2 Intel 逻辑地址到线性地址的变换-段式管理

1.基本原理：

在段式存储管理中，将程序的地址空间划分为若干个段，这样每个进程有一个二维的地址空间。在段式存储管理系统中，为每个段分配一个连续的分区，而进程中的各个段可以不连续地存放在内存的不同分区中。程序加载时，操作系统为所有段分配其所需内存，这些段不必连续，物理内存的管理采用动态分区的管理方法。

在为某个段分配物理内存时，可以采用首先适配法、下次适配法、最佳适配法等方法。在回收某个段所占用的空间时，要注意将收回的空间与其相邻的空间合并。

程序通过分段划分为多个模块，如代码段、数据段、共享段：可以分别编写和编译；可以针对不同类型的段采取不同的保护；可以按段为单位来进行共享，包括通过动态链接进行代码共享。这样做的优点是：可以分别编写和编译源程序的一个文件，并且可以针对不同类型的段采取不同的保护，也可以按段为单位来进行共享。

总的来说，段式存储管理的优点是：没有内碎片，外碎片可以通过内存紧缩来消除；便于实现内存共享。缺点与页式存储管理的缺点相同，进程必须全部装入内存。

2.段式管理的数据结构：

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。

（1）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，即段内地址。在系统中为每个进程建立一张段映射表，其结构如图 7.2。

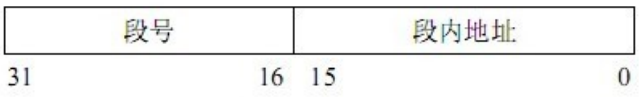


图 7.2 段映射表结构

（2）系统段表：系统所有占用段（已经分配的段）。

（3）空闲段表：内存中所有空闲段，可以结合到系统段表中。

3.段式管理的地址变换

在段式管理系统中，整个进程的地址空间是二维的，即其逻辑地址由段号和段内地址两部分组成。为了完成进程逻辑地址到物理地址的映射，处理器会查找内存中的段表，由段号得到段的首地址，加上段内地址，得到实际的物理地址。这个过程也是由处理器的硬件直接完成的，操作系统只需在进程切换时，将进程段表的首地址装入处理器的特定寄存器当中。这个寄存器一般被称作段表地址寄存器。过程如图 7.3 所示。

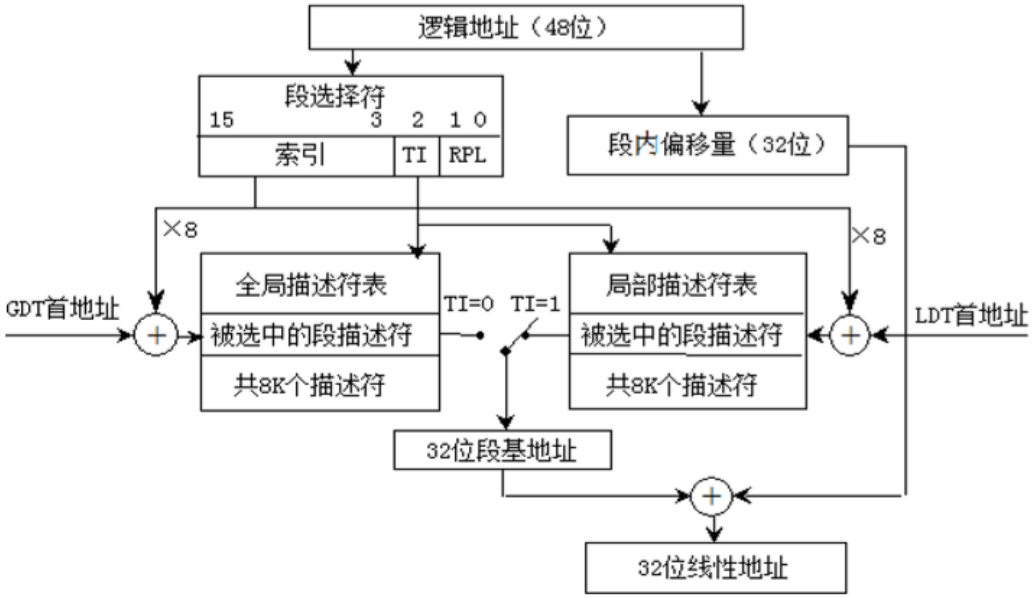


图 7.3 段表地址的地址变换

## 7.3 Hello 的线性地址到物理地址的变换-页式管理

1.基本原理

将程序的逻辑地址空间划分为固定大小的页，而物理内存划分为同样大小的页框。程序加载时，可将任意一页放入内存中任意一个页框，这些页框不必连续，从而实现了离散分配。该方法需要 CPU 的硬件支持，来实现逻辑地址和物理地址之间的映射。在页式存储管理方式中地址结构由两部构成，前一部分是 VPN（虚拟页号），后一部分是 VPO（虚拟页偏移量）。如图 7.4 所示。

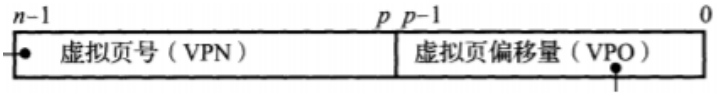


图 7.4 物理地址的结构

页式管理方式的优点：没有外碎片；一个程序不必连续存放；便于改变程序占用空间的大小(主要指随着程序运行，动态生成的数据增多，所要求的地址空间相应增长)。

页式管理方式的缺点：要求程序全部装入内存，没有足够的内存，程序就不能执行。

2.页式管理的数据结构

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。

页表：页表将虚拟内存映射到物理页。每次地址翻译硬件将一个虚拟地址转换为物理地址时，都会读取页表。页表是一个页表条目（PTE）的数组。虚拟地址空间的每个页在页表中一个固定偏移量处都有一个 PTE。假设每个 PTE 是由一个有效位和一个 n 位地址字段组成的。有效位表明了该虚拟页当前是否被缓存在 DRAM 中。如果设置了有效位，那么地址字段就表示 DRAM 中相应的物理页的起始位置，这个物理页中缓存了该虚拟页。如果没有设置有效位，那么一个空地址表示这个虚拟页还未被分配。否则，这个地址就指向该虚拟页在磁盘上的起始位置。

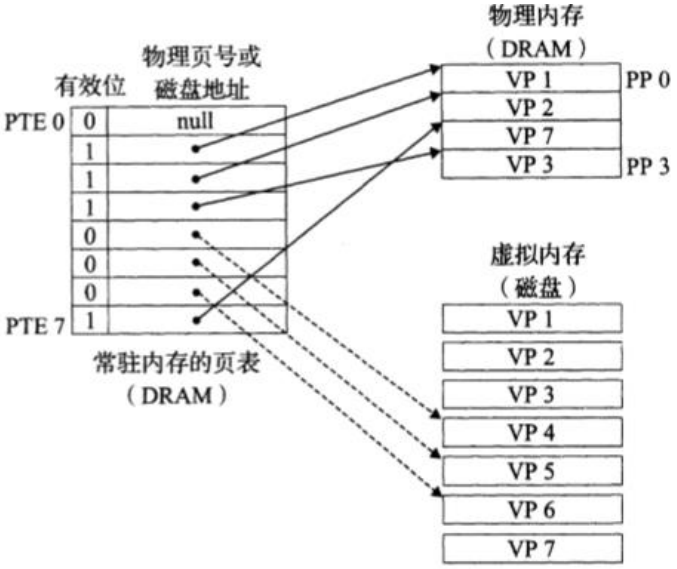


图 7.5 页表

3.页式管理地址变换

MMU 利用 VPN 来选择适当的 PTE，将列表条目中 PPN 和虚拟地址中的 VPO 串联起来，就得到相应的物理地址。

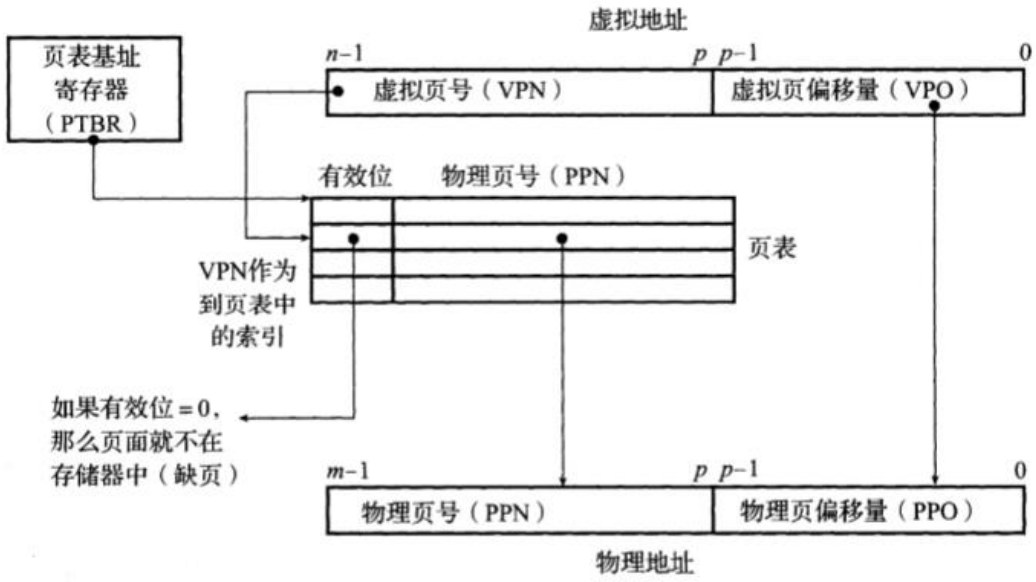


图 7.6 使用页表的地址翻译

## 7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

为了消除每次 CPU 产生一个虚拟地址 MMU 就查阅一个 PTE 带来的时间开销，许多系统都在 MMU 中包括了一个关于 PTE 的小的缓存，称为翻译后被缓冲器（TLB），TLB 的速度快于 L1 cache。

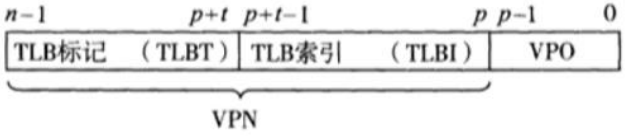


图 7.7 虚拟地址中用以访问 TLB 的组成部分

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。

同时，为了减少页表太大而造成的空间损失，可以使用层次结构的页表页压缩页表大小。如图 7.8 所示。

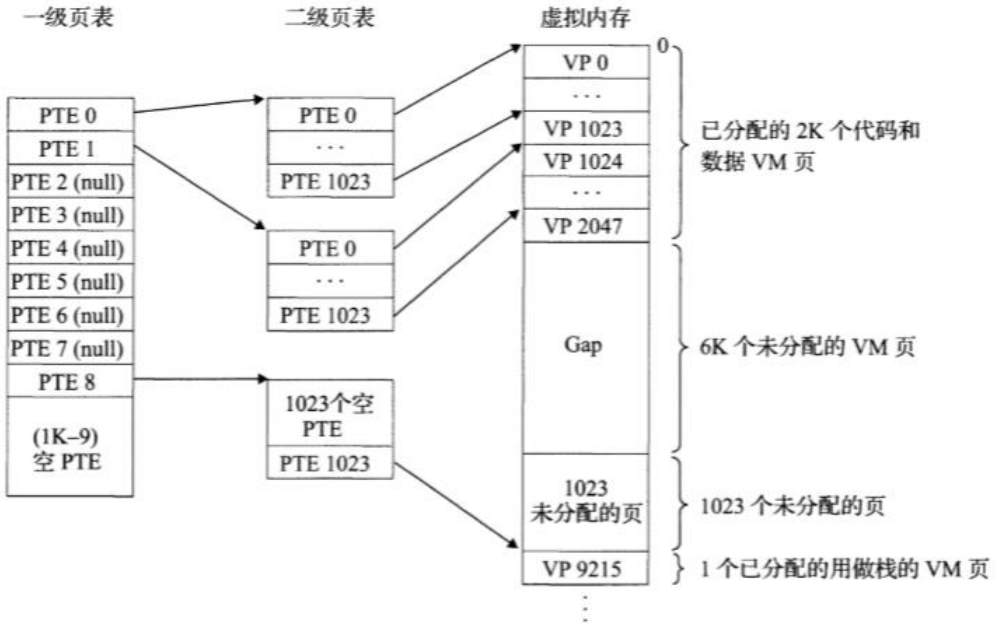


图 7.8 一个二级页表层次结构

Core i7 使用的是四级页表。如图 7.9 所示，在四级页表层次结构的地址翻译中，虚拟地址被划分为 4 个 VPN 和 1 个 VPO。每个第 i 个 VPN 都是一个到第 i 级页表的索引，第 j 级页表中的每个 PTE 都指向第 j+1 级某个页表的基址，第四级页表中的每个 PTE 包含某个物理页面的 PPN，或者一个磁盘块的地址。为了构造物理地址，在能够确定 PPN 之前，MMU 必须访问四个 PTE。

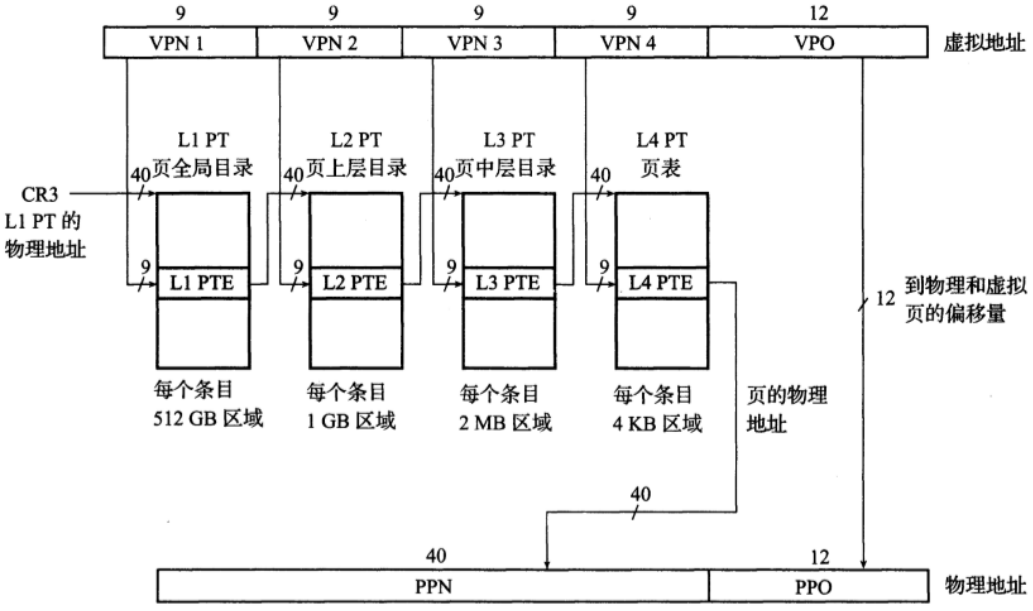


图 7.9 Core i7 页表翻译

## 7.5 三级 Cache 支持下的物理内存访问

通过 7.3 和 7.4 两节，hello 的物理地址已经得知，现在需要访问该物理地址。在现代计算机中，存储器被组织成层次结构，因为这样可以最大程度地平衡访存时间和存储器成本。所以在 CPU 在访存时并不是直接访问内存，而是访问内存之前的三级 cache。已知 Core i7 的三级 cache 是物理寻址的，块大小为 64 字节。LI 和 L2 是 8 路组相联的，而 L3 是 16 路组相联的。Corei7 实现支持 48 位虚拟地址空间和 52 位物理地址空间。

得到了 52 位物理地址，接下来 CPU 把地址发送给 L1，因为 L1 块大小为 64 字节，所以 B=64，b=6。又 L1 是 8 路组相联的，所以 S=8，s=3。标记位 t 有 52-6-3=43 位，即是得到的 52 位物理地址的前 43 位。首先，根据物理地址的 s 位组索引索引到 L1 cache 中的某个组，然后在该组中查找是否有某一行的标记等于物理地址的标记并且该行的有效位为 1，若有，则说明命中，从这一行对应物理地址 b 位块偏移的位置取出一个字节，若不满足上面的条件，则说明不命中，需要继续访问下一级 cache，访问的原理与 L1 相同，若是三级 cache 都没有要访问的数据，则需要访问内存，从内存中取出数据并放入 cache。

图 7.10 展示了 Core i7 的地址翻译过程。

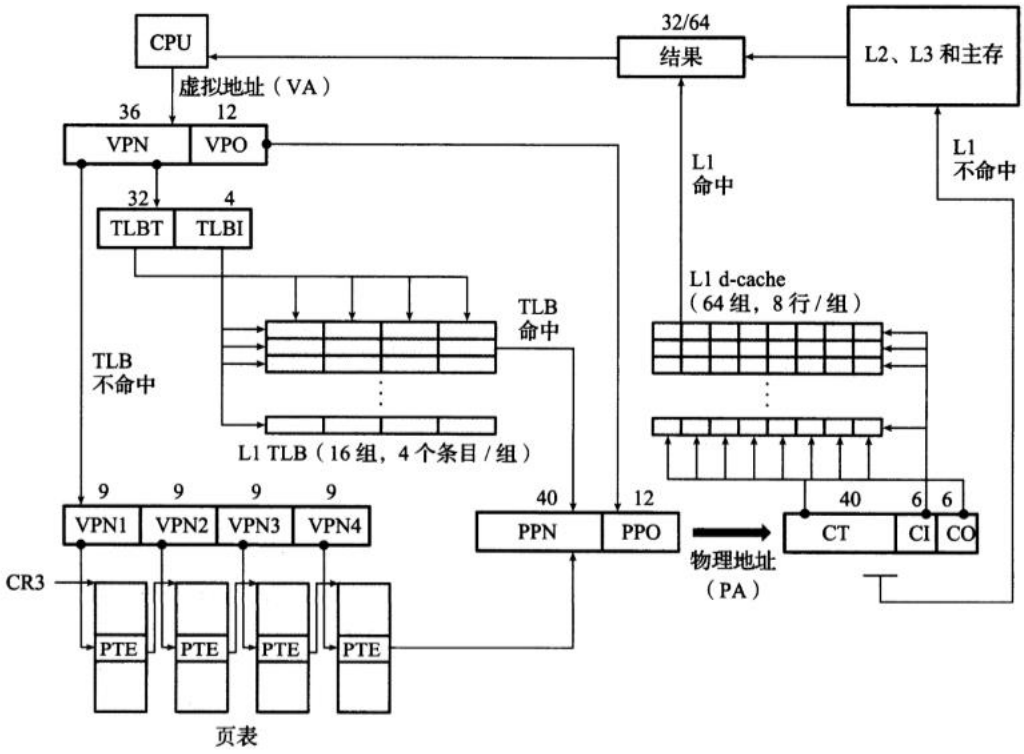


图 7.10 TLB 与 4 级页表下 Core i7 的地址翻译

## 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，并分配给他一个唯一的 PID。为了给这个新进程创建虚拟内存，它创建了当前进程的 mm_struct 区域结构和页表的原样副本。它将两个进程中的每个页面都标记为只读，并将两个进程中每个区域结构都标记为私有的写时复制。其内存映射如图 7.11。

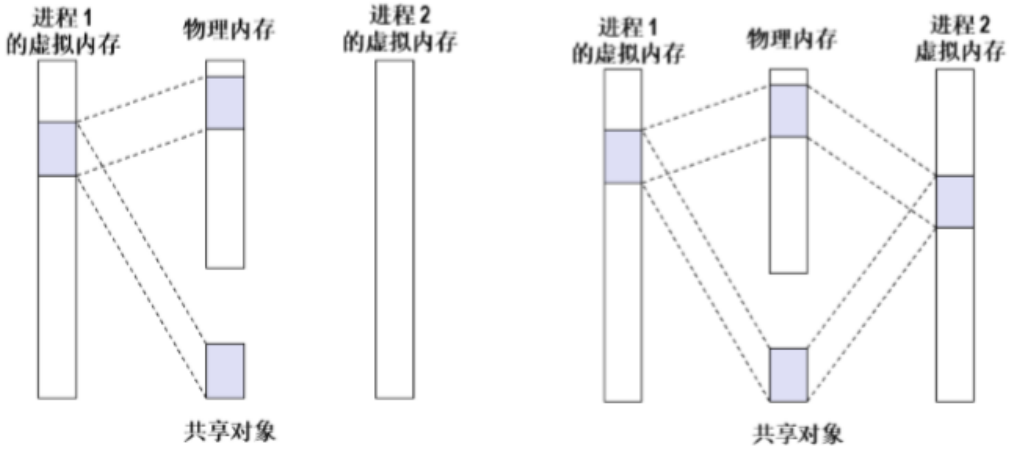


图 7.11 hello 进程 fork 时的内存映射

## 7.7 hello 进程 execve 时的内存映射

execve 函数调用驻留在内核区域的启动加载器代码，在当前进程中加载并运行包含在可执行目标文件 hello 中的程序，用 hello 程序有效地替代了当前程序。

加载并运行 hello 需要以下几个步骤：

（1）删除已存在的用户区域，删除当前进程虚拟地址的用户部分中的已存在的区域结构。

（2）映射私有区域，为新程序的代码、数据、bss 和栈区域创建新的区域结构，所有这些新的区域都是私有的、写时复制的。代码和数据区域被映射为 hello 文件中的.text 和.data 区，bss 区域是请求二进制零的，映射到匿名文件，其大小包含在 hello 中，栈和堆地址也是请求二进制零的，初始长度为零。

（3）映射共享区域，hello 程序与共享对象 libc.so 链接，libc.so 是动态链接到这个程序中的，然后再映射到用户虚拟地址空间中的共享区域内。

（4）设置程序计数器（PC），execve 做的最后一件事情就是设置当前进程上下文的程序计数器，使之指向代码区域的入口点

## 7.8 缺页故障与缺页中断处理

缺页故障是一种常见的故障，当指令引用一个虚拟地址，在 MMU 中查找页表时发现与该地址相对应的物理地址不在内存中，因此必须从磁盘中取出的时候就会发生故障。其处理流程遵循图 7.12 所示的故障处理流程。

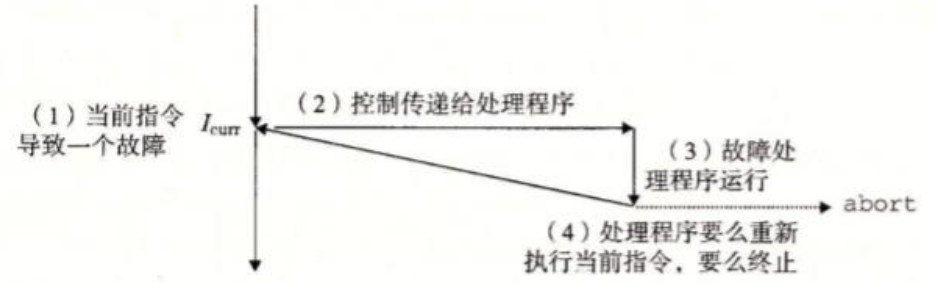


图 7.12 故障处理流程

缺页中断处理：缺页处理程序是系统内核中的代码，选择一个牺牲页面，如果这个牺牲页面被修改过，那么就将它交换出去，换入新的页面并更新页表。当缺页处理程序返回时，CPU 重新启动引起缺页的指令，这条指令再次发送 VA 到 MMU，这次 MMU 就能正常翻译 VA 了。

## 7.9 动态存储分配管理

1.动态内存分配器的基本原理

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者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。已分配的块显式地保留为供应用程序使用。空闲块可用来分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

分配器的类型有两种：显式分配器和隐式分配器。

显式分配器：要求应用显式地释放任何已分配的块。例如，C 语言中的 malloc 函数申请了一块空间之后需要 free 函数释放这个块

隐式分配器：应用检测到已分配块不再被程序所使用，就释放这个块。比如 Java，ML 和 Lisp 等高级语言中的垃圾收集。

2.带边界标签的隐式空闲链表分配器原理

带边界标签的隐式空闲链表的堆块结构如图 7.13。一个块是由一个字的头部、有效载荷、可能的一些额外的填充，以及在块的结尾处的一个字的脚部组成的。头部编码了这个块的大小（包括头部和所有的填充），以及这个块是已分配的还是空闲的。如果我们强加一个双字的对齐约束条件，那么块大小就总是 8 的倍数，且块大小的最低 3 位总是 0。因此，我们只需要内存大小的 29 个高位，释放剩余的 3 位来编码其他信息。在这种情况中，我们用其中的最低位（已分配位）来指明这个块是已分配的还是空闲的。

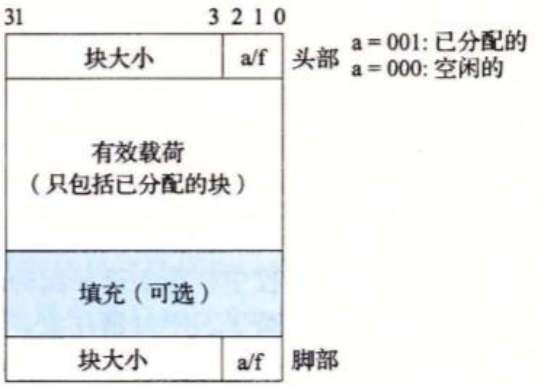


图 7.13 使用边界标记的堆块结构

寻找一个空闲块的方式有三种：

（1）首次适配：从头开始搜索空闲链表，选择第一个合适的空闲块：可以取总块数（包括已分配和空闲块）的线性时间，但是会在靠近链表起始处留下小空闲块的“碎片”。

（2）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，优点是比首次适应更快：避免重复扫描那些无用块。但是一些研究表明，下一次适配的内存利用率要比首次适配低得多。

（3）最佳适配：查询链表，选择一个最好的空闲块适配，剩余最少空闲空间，优点是可以保证碎片最小——提高内存利用率，但是通常运行速度会慢于首次适配。

3.关于堆块的合并有如图 7.14 的四种情况。在情况 1 中，两个邻接的块都是已分配的，因此不可能进行合并。所以当前块的状态只是简单地从已分配变成空闲。在情况 2 中，当前块与后面的块合并。用当前块和后面块的大小的和来更新当前块的头部和后面块的脚部。在情况 3 中，前面的块和当前块合并。用两个块大小的和来更新前面块的头部和当前块的脚部。在情况 4 中，要合并所有的三个块形成一个单独的空闲块，用三个块大小的和来更新前面块的头部和后面块的脚部。在每

种情况中，合并都是在常数时间内完成的。

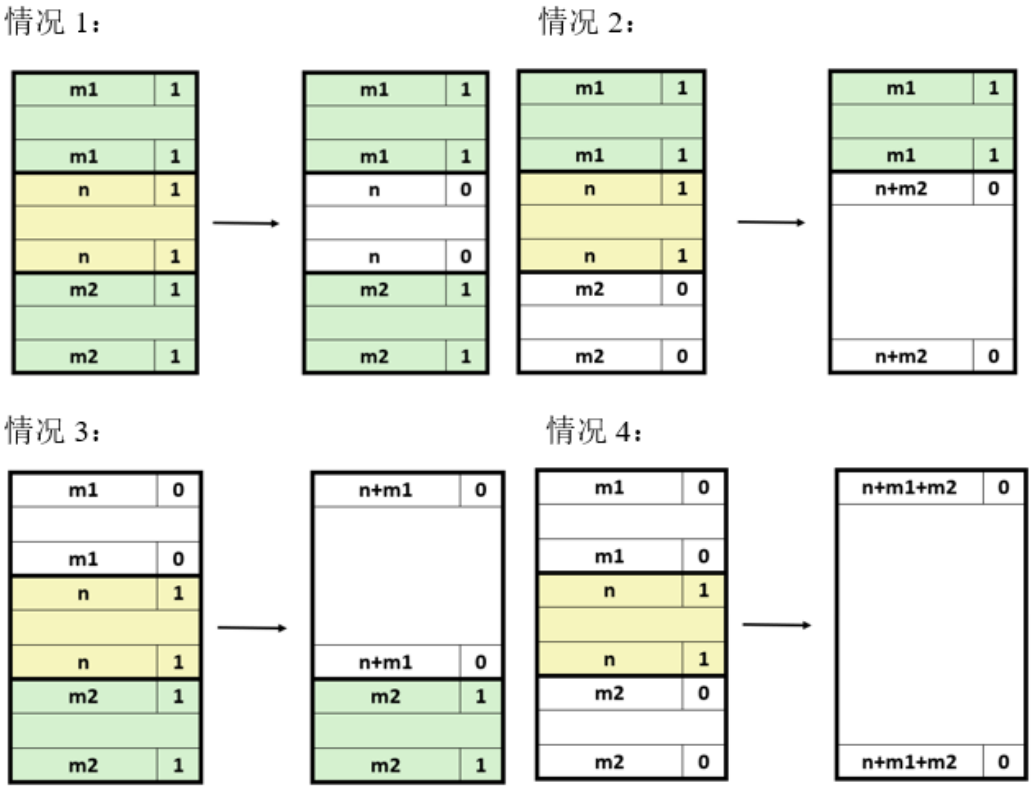


图 7.14 堆块合并的四种情况

3.显式空间链表的基本原理

显式空间链表的堆块结构如图 7.15。将空闲块组织成链表形式的数据结构。堆可以组织成一个双向空闲链表，在每个空闲块中，都包含一个 pred（前驱）和 succ（后继）指针。

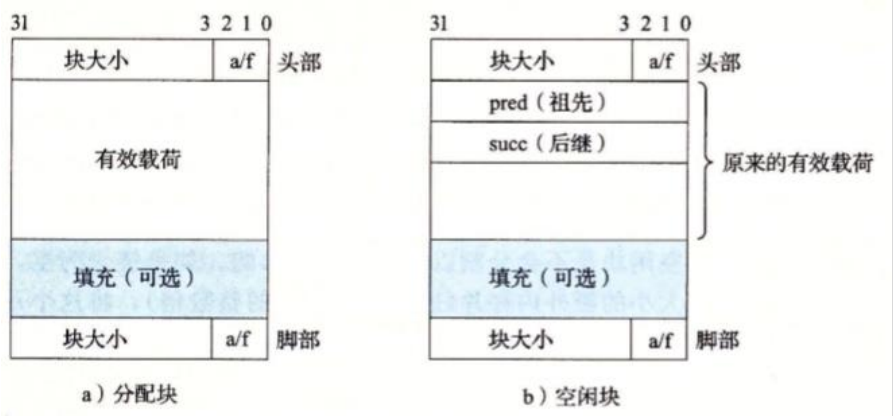


图 7.15 显式空间链表的堆块结构

使用双向链表而不是隐式空闲链表，使首次适配的分配时间从块总数的线性

时间减少到了空闲块数量的线性时间。不过，释放一个块的时间可以使线性的，也可以是一个常数，这取决于我们选择的空闲链表中块的排序策略。

链表的维护方式有两种：一种方法是用后进先出（LIFO）的顺序维护链表，将新释放的块放置在链表的开始处。使用 LIFO 的顺序和首次适配的放置策略，分配器会先检查最近使用过的块。在这种情况下，释放一个块可以在常数时间内完成。如果使用了边界标记，那么合并也可以在线性时间内完成。另一种方法是按照地址顺序来维护链表，其中链表中每个块的地址都小于它后继的地址。在这种情况下，释放一个块需要线性时间的搜索来定位合适的前驱。平衡点在于，按照地址排序的首次适配比 LIFO 排序的首次适配有更高的内存利用率，接近最佳适配的利用率。

一般而言，显示链表的缺点是空闲块必须足够大，以包含所有需要的指针，以及头部和可能的脚部，这就导致了更大的最小块大小，也潜在地提高了内部碎片的程度。

## 7.10 本章小结

本章主要介绍了 hello 的存储地址空间、intel 的段式管理、hello 的页式管理，以及 TLB 与四级页表支持下的 VA 到 PA 的变换过程和三级 Cache 支持下的物理内存访问。还阐述了 hello 进程 fork 和 execve 时的内存映射、缺页故障的处理流程和动态存储分配器的管理。

**（第 7 章 2 分）**

# 第 8 章 hello 的 I0 管理

## 8.1 Linux 的 I0 设备管理方法

所有的 I/O 设备都被模型化为文件，而所有的输入和输出都被当作相应文件的读和写来完成，这种将设备优雅的映射为文件的方式，允许 Linux 内核引出一个简单的，低级的应用接口，称为 Unix I/O，这使得所有的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 I0 接口及其函数

Unix I/O 接口的几种操作：

（1）打开文件：程序要求内核打开文件，内核返回一个小的非负整数（描述符），用于标识这个文件。程序在只要记录这个描述符便能记录打开文件的所有信息。

（2）shell 在进程的开始为其打开三个文件：标准输入、标准输出和标准错误。

（3）改变当前文件的位置：对于每个打开的文件，内核保存着一个文件位置 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 操作显式地设置文件的当前位置为 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>=m 时执行读操作会出发一个称为 EOF 的条件，应用程序能检测到这个条件，在文件结尾处并没有明确的 EOF 符号。

（5）关闭文件：内核释放打开文件时创建的数据结构以及占用的内存资源，并将描述符恢复到可用的描述符池中。无论一个进程因为何种原因终止时，内核都会关闭所有打开的文件并释放它们的内存资源。

UnixI/O 函数：

（1）intopen(char* filename, int flags, mode_t mode);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。

（2）intclose(int fd);关闭一个打开的文件，返回操作结果。

（3）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。

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

## 8.3 printf 的实现分析

printf 函数的实现如图 8.1。首先 arg 获得第二个不定长参数，即输出的时候格式化串对应的值。

```
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 函数的代码

然后查看 vsprintf 代码如图 8.2。可以看出 vsprintf 程序按照格式 fmt 结合参数 args 生成格式化之后的字符串，并返回字串的长度。在 printf 中调用系统函数 write(buf,i)将长度为 i 的 buf 输出。

```
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 != '%') //忽略无关字符
        {
            *p++ = *fmt;
            continue;
        }
        fmt++;
        switch (*fmt)
        {
            case 'x':  //只处理%x一种情况
                itoa(tmp, *((int*)p_next_arg)); //将输入参数值转化为字符串保存在tmp
                strcpy(p, tmp); //将tmp字符串复制到p处
                p_next_arg += 4; //下一个参数值地址
                p += strlen(tmp); //放下一个参数值的地址
                break;
            case 's':
                break;
            default:
                break;
        }
    }
    return (p - buf); //返回最后生成的字符串的长度
}
```

图 8.2 vsprintf 函数的代码

查看 write 函数如图 8.3。在 write 函数中，将栈中参数放入寄存器，ecx 是字符个数，ebx 存放第一个字符地址，int INT_VECTOR_SYS_CALLA 代表通过系统调用 syscall。

```
write:
    mov eax, _NR_write
    mov ebx, [esp + 4]
    mov ecx, [esp + 8]
    int INT_VECTOR_SYS_CALL
```

图 8.3 write 函数的代码

查看 syscall 的实现如图 8.4。syscall 将字符串中的字节“Hello 11183710113 许健”从寄存器中通过总线复制到显卡的显存中.

```
sys_call:
    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
```

图 8.4 syscall 的代码

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（RGB 分量）。于是我们的打印字符串“Hello 11183710113 许健”就显示在了屏幕上。

## 8.4 getchar 的实现分析

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

再看 getchar 的代码如图 8.5。可以看到，getchar 调用了 read 函数，read 函数也通过 sys_call 调用内核中的系统函数，将读取存储在键盘缓冲区中的 ASCII 码，直到读到回车符，然后返回整个字符串，getchar 函数只从中读取第一个字符，其他的字符被缓存在输入缓冲区。

```
int getchar(void)
{
    static char buf[BUFSIZ];
    static char* bb = buf;
    static int n = 0;
    if (n == 0)
    {
        n = read(0, buf, BUFSIZ);
        bb = buf;
    }
    return (--n >= 0) ? (unsigned char)*bb++ : EOF;
}
```

图 8.5 getchar 的代码

## 8.5 本章小结

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

**（第 8 章 1 分）**

# 结论

hello 程序终于完成了它艰辛但可谓精彩的一生。hello 的一生大事记如下：

（1）编写，通过 editor 将代码键入 hello.c

（2）预处理，经过预处理器 cpp 的预处理，处理以#开头的行，得到 hello.i

（3）编译，编译器 ccl 将得到的 hello.i 编译成汇编文件 hello.s

（4）汇编，汇编器 as 又将 hello.s 翻译成机器语言指令得到可重定位目标文件 hello.o

（5）链接，链接器 ld 将 hello.o 与动态链接库链接生成可执行目标文件 hello，至此，hello 成为了一个可以运行的程序。

（6）运行，在 shell 中输入./hello 1183710113 许健 1，

（7）创建子进程，shell 进程调用 fork 为其创建子进程

（8）加载，shell 调用 execve，execve 调用启动加载器，加映射虚拟内存，进入程序入口后程序开始载入物理内存，然后进入 main 函数。

（9）执行，CPU 为其分配时间片，在一个时间片中，hello 享有 CPU 资源，顺序执行自己的控制逻辑流

（10）访问内存，当 CPU 访问 hello 时，请求一个虚拟地址，MMU 把虚拟地

址转换成物理地址并通过三级 cache 访存。

（11）动态申请内存，printf 会调用 malloc 向动态内存分配器申请堆中的内存。

（12）信号，hello 运行过程中可能遇到各种信号，shell 为其提供了各种信号处理程序。

（13）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，hello 结束了它的一生。

感悟：即使是一个简单的 hello.c 也需要操作系统提供很多的支持，并且每一步都经过了设计者的深思熟虑，在有限的硬件水平下把程序的时间和空间性能都做到了近乎完美的利用。在做本次大作业的过程中，回顾了整个计算机系统，回顾了很多调试器的使用方法，对整个程序的运行过程有了一个新的认识。

**（结论 0 分，缺失 -1 分，根据内容酌情加分）**

# 附件

列出所有的中间产物的文件名，并予以说明起作用。
附件 1：hello.c——原文件
附件 2：hello.i——预处理之后文本文件
附件 3：hello.s——编译之后的汇编文件
附件 4：hello.o——汇编之后的可重定位目标执行
附件 5：hello——链接之后的可执行目标文件
附件 6：hello.elf——hello.o 的 elf 格式，用来看 hello.o 的各节信息
附件 7：hello.ob——hello.o 的反汇编文件，用来看汇编器翻译后的汇编代码
附件 8：hello1.ob——hello 的反汇编文件，用来看链接器链接后的汇编代码

**（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] 兰德尔 E.布莱恩特. 深入理解计算机系统. 龚奕利 译.

[2] 库函数 getchar()详解 https://blog.csdn.net/hulifangjiayou/article/details/40480467

[3] Linux 进程虚拟地址空间 https://www.cnblogs.com/xelatex/p/3491305.html

[4] 虚拟地址、逻辑地址、线性地址、物理地址 https://blog.csdn.net/rabbit_in_android/article/details/49976101

[5] 内存地址转换与分段 https://blog.csdn.net/drshenlei/article/details/4261909

[6] printf 函数实现的深入剖析 https://www.cnblogs.com/pianist/p/3315801.html

**（参考文献 0 分，缺失 -1 分）**