摘　要

本文以最简单的 hello 程序为例介绍整个程序的生命周期。以 hello.c 源程序为起点，从预处理、编译、汇编、链接，到加载、运行，再到终止、回收。对各个过程的内容与实现进行分析与解释。目的是增强对整个计算机系统的了解。

关键词：计算机系统；程序的生命周期

（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: From Program to Process

Program：在编辑器中中输入代码得到的 hello.c 源程序，此时仅是文本文件

Process：hello.c（在 Linux 中），经过过 cpp 的预处理、ccl 的编译、as 的汇编、ld 的链接最终成为可执目标程序 hello。在 shell 中输入./hello 后，shell 对命令行进行解析，随后 fork 产生子进程，在子进程中调用 execeve，实际执行 hello 中的代码。

O2O: From Zero-0 to Zero-0

shell 为 hello 进程 execve，映射虚拟内存，进入程序入口后程序开始载入物理内存。进入 main 函数执行目标代码，CPU 为运行的 hello 分配时间片执行逻辑控制流。当程序运行结束后，shell 父进程负责回收 hello 进程，内核删除相关数据结构。

1.2 环境与工具

硬件环境：CPU: Intel(R)_Core(TM)_i5-8265U_CPU_@_1.60GHz, RAM: 8.00GB 系统类型：64 位操作系统，基于 x64 的处理器

软件环境：Windows10 64 位；Ubuntu 16

开发与调试工具：gcc，vim，edb，readelf，objdump

1.3 中间结果

文件名	作用
hello.c	源代码
hello.i	对 hello.c 预处理后的文件
hello.s	对 hello.i 进行编译后的文件
hello.o	对 hello.s 进行汇编后的文件
hello	对 hello.o 进行链接后的文件
hello_o_elf	用 readelf 对 hello.o 进行解析的结果
hello_o_objdump	用 objdump 对 hello.o 进行反汇编的结果
hello_elf	用 readelf 对 hello 进行解析的结果
hello_objdump	用 objdump 对 hello 进行反汇编的结果

1.4 本章小结

本章简述了 hello 程序生命周期的大致过程，同时列出了实验的基本信息，包括实验环境、工具以及中间结果。

（第 1 章 0.5 分）

第 2 章 预处理

2.1 预处理的概念与作用

概念：指以”#”开头的预处理指令，比如#include、#define 等，在编译的第一遍扫描之前，系统根据这些指令调用预处理程序对源代码进行一些文本性质的操作，生成扩展的 C 源程序的过程。在这里具体指使用 cpp 处理 hello.c 文件得到 hello.i 的过程。

作用：主要作用是将头文件中的内容直接插入到源文件中，其次是根据宏定义在源程序中做一些替换工作，最后是删除所有注释。经过预处理过程就可以得到便于编译器工作的、以.i 为后缀的文件。

2.2 在 Ubuntu 下预处理的命令

命令：gcc hello.c -E -o hello.i

在 hello.c 所在目录在开启终端，在终端中键入上述指令得到 hello.i，通过命令”vim hello.i”可以打开，发现这仍然是一个文本文件。

图 2-1 输入命令后得到 hello.i 文件

```
extern int rpmatch (const char *__response) __attribute__ ((__nothrow__ , __leaf__)) __attribute__ ((__nonnull__ (1))) ;
# 958 "/usr/include/stdlib.h" 3 4
extern int getsubopt (char **__restrict __optionp,
        char *const *__restrict __tokens,
        char **__restrict __valuep)
     __attribute__ ((__nothrow__ , __leaf__)) __attribute__ ((__nonnull__ (1, 2, 3))) ;
# 1004 "/usr/include/stdlib.h" 3 4
extern int getloadavg (double __loadavg[], int __nelem)
     __attribute__ ((__nothrow__ , __leaf__)) __attribute__ ((__nonnull__ (1)));
# 1014 "/usr/include/stdlib.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/stdlib-float.h" 1 3 4
# 1015 "/usr/include/stdlib.h" 2 3 4
# 1026 "/usr/include/stdlib.h" 3 4

# 9 "hello.c" 2


# 10 "hello.c"
int main(int argc,char *argv[]){
 int i;

 if(argc!=4){
  printf("用法：Hello 学号 姓名 秒数！\n");
  exit(1);
 }
 for(i=0;i<8;i++){
  printf("Hello %s %s\n",argv[1],argv[2]);
  sleep(atoi(argv[3]));
 }
 getchar();
 return 0;
}
```

图 2-2 使用 vim 打开 hello.i 后部分内容

2.3 Hello 的预处理结果解析

这个文本文件的内容相较 hello.c 来说添加了很多内容，一共有 3069 行代码，在 vim 中键入”/main”寻找 main 函数，发现在文件的末尾。前面所有的内容都是预处理插入的代码。然后尝试通过注释的内容进行检索，发现没有找到，说明注释信息也在预处理过程中被删除了。

2.4 本章小结

本章的内容相对简单，描述了 hello 声明周期中的预处理阶段。对预处理的概念及作用进行了简单的描述。并在 Ubuntu 下对示例的 hello 程序进行预处理操作，使用 vim 编辑器查看了预处理后得到的 hello.i 文件的大体内容。

（第 2 章 0.5 分）

第 3 章 编译

3.1 编译的概念与作用

概念：在这里编译指利用 ccl 将文本文件 hello.i 翻译成文本文件 hello.s 的过程。这个过程以高级语言作为输入，汇编语言作为输出。

作用：主要作用是进行词法分析、语法分析、目标代码的生成，检查源代码中是否有错误，如果没有错误就生成汇编代码。同时现代编译器还有一定的优化功能，一定程度上可以提高程序的性能。

3.2 在 Ubuntu 下编译的命令

命令：gcc -S hello.i -o hello.s

图 3-1 输入命令以后得到 hello.s 文件

使用命令：vim hello.s 打开 hello.s 后，发现该文件还是一个文本文件，可以直接阅读。

```
        .file   "hello.c"
        .text
        .section        .rodata
        .align 8
.LC0:
        .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"
.LC1:
        .string "Hello %s %s\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB6:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $32, %rsp
        movl    %edi, -20(%rbp)
        movq    %rsi, -32(%rbp)
        cmpl    $4, -20(%rbp)
        je      .L2
        leaq    .LC0(%rip), %rdi
        call    puts@PLT
        movl    $1, %edi
        call    exit@PLT
.L2:
        movl    $0, -4(%rbp)
        jmp     .L3
.L4:
        movq    -32(%rbp), %rax
        addq    $16, %rax
```

图 3-2 使用 vim 打开 hello.s 后得到的部分内容

3.3 Hello 的编译结果解析

在文件头部可以得到如下的内容：

```
        .file   "hello.c"
        .text
        .section        .rodata
        .align 8
.LC0:
        .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"
.LC1:
        .string "Hello %s %s\n"
        .text
        .globl  main
        .type   main, @function
```

图 3-3 hello.s 文件头部的内容

这些内容分别代表：

.file：声明源文件的文件名

.text：声明代码节

.section　.rodata：声明只读数据节

.align 8：表示地址以 8 字节对齐

.string：用于声明字符串，这里声明了 2 个，分别是.LC0 和.LC1

.global：声明 main 为一个全局变量

.type：指定 main 是一个函数

3.3.1 数据

1.字符串常量

程序中有两个字符串常量，又 hello.s 的头部信息可知，这两个字符串都是存储在只读数据节的。

```
.LC0:
        .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"
.LC1:
        .string "Hello %s %s\n"
```

图 3-4 hello.s 中两个字符串常量

从 hello.c 中看，这两个字符串分别是 printf 中输出的固定的字符串，另一个是 printf 中的格式串。

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

        if(argc!=4){
                printf("用法：Hello 学号 姓名 秒数！\n");
                exit(1);
        }
        for(i=0;i<8;i++){
                printf("Hello %s %s\n",argv[1],argv[2]);
                sleep(atoi(argv[3]));
        }
        getchar();
        return 0;
```

两个字符串常量

图 3-5 hello.c 中的两个字符串常量

2.变量

main 函数中定义了一个局部变量 i 作为循环变量，这里 i 被编译器处理成存储在栈上，并使用 movl 为其赋初值。

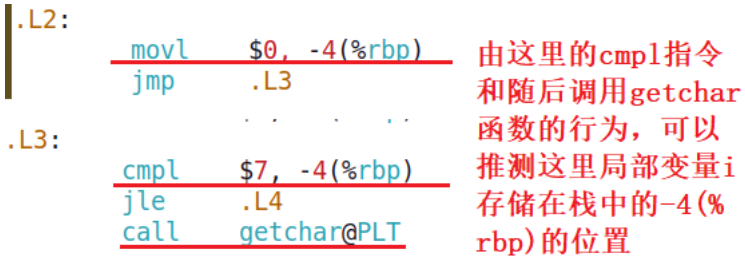

图 3-6 hello.s 中局部变量 i 的行为

对于 main 函数自身来说，它有两个参数 argc 和 argv，分别表示参数个数和参数列表的首地址，在进入 main 函数时这两个参数就如正常的函数调用一样分别存储在%rdi、%rsi 中，在 main 函数中，它们又被存储到了栈上，防止后面调用其他函数时被修改。

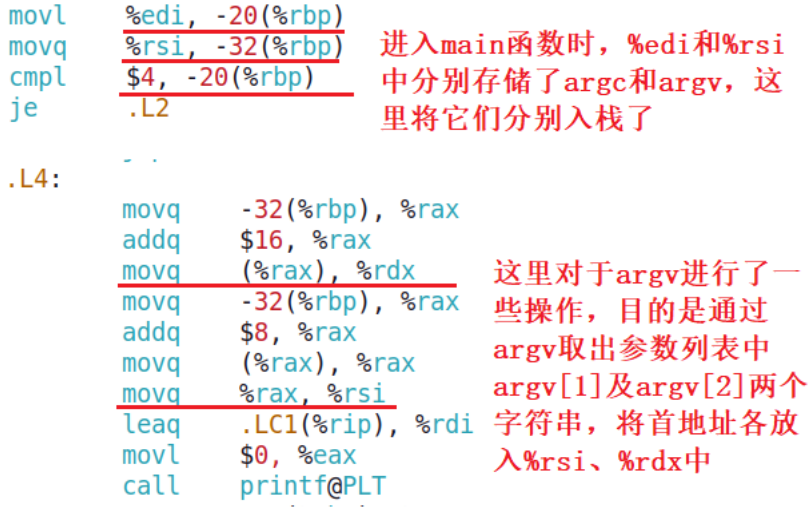

图 3-7 main 函数的两个参数在汇编代码中的行为

本程序中不含显式的静态、全局变量。

3.一般常量

这里主要是对局部变量 i 赋初值、循环控制条件中的边界值、数组元素索引等常量，这些常量大部分情况下都是直接表现在汇编代码中。只有数组元素索引中的常量由于涉及到数组元素大小问题，在汇编代码中体现为“对应常量*元素大小”的形式

3.3.2 赋值

本程序中只有一个赋值操作：对循环变量 i 赋初值。这里使用命令”movl　$0, -4(%rbp)”对 i 进行赋初值。”movl”是四字节数据传送指令，对应变量 i 的数据类型为 int 型。

3.3.3 类型转换

本程序中也只涉及一个类型转换的操作：main 函数要求在第四个参数位置以字符串格式传入挂起时间。这就要求在调用 sleep 函数时需要把这个参数从字符串类型转换成整数类型。

```
movq    -32(%rbp), %rax
addq    $24, %rax
movq    (%rax), %rax
movq    %rax, %rdi
call    atoi@PLT
```

取出argv[3]，将其传入%rdi作为atoi函数的参数

图 3-8 调用 atoi 函数

atoi 函数最终将返回值放在%eax 中。

3.3.4 算术操作

本程序中涉及的算数操作只有每次循环后循环变量 i 自增 1 的操作，编译器在这里并没有使用 INC 指令，主要是因为 i 并没有存储在寄存器上，i 在汇编代码中一直体现为存储在栈上，所以只能使用 ADD 指令。具体来说这里的指令是：”addl $1, -4(%rbp)”。

3.3.5 关系操作

1.在判断参数输入个数是否正确时，使用了 argc!=4 进行判断，这是第一个关系操作。在源代码中的行为是：如果参数个数不等于 4，对用户进行提示后退出 main 函数。在编译器进行处理后的行为是：如果 agrc 等于 4，那么就跳转到.L2 处的代码，否则不跳转继续执行。可以看出这里编译器对 if 的解析使用了跳转或不跳转。

```
cmpl    $4, -20(%rbp)
je      .L2
leaq    .LC0(%rip), %rdi
call    puts@PLT
movl    $1, %edi
call    exit@PLT
```

图 3-9 编译器对 argc!=4 的处理

2.第二个关系操作也类似，这里是判断循环是否结束时使用了 i<8 这个关系操作。在源代码中的行为是如果 i 小于等于 7，那么就往回跳转到循环开始，否则继续执行。相当于也是利用关系操作的结果作为跳转的条件。

```
cmpl    $7, -4(%rbp)
jle     .L4
call    getchar@PLT
movl    $0, %eax
leave
```

图 3-10 编译器对 i<8 的处理

3.3.6 数组

程序中涉及的数组只有 main 函数的参数列表，定义为 char *argv[]，是一个字符串数组，数组中每一个元素都是一个字符串。观察汇编代码中的行为，可以发现利用索引获取数组元素时的步骤：

1. 获取数组首地址
2. 根据索引值对首地址进行加法操作得到目标元素的首地址
3. 通过内存解析得到目标元素并存储在某块其他区域

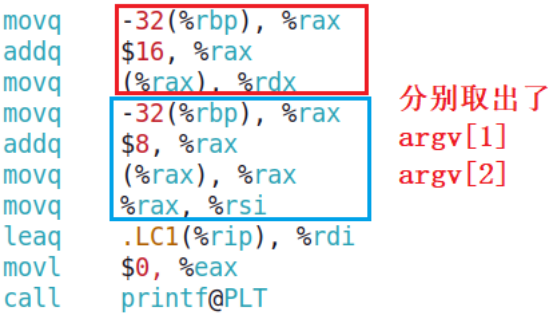

图 3-10 利用索引取出 argv 数组中的元素

3.3.7 控制转移

控制转移的实现离不开上面 3.3.5 节中的关系操作符，简单来说，编译器解析控制转移就是通过首先设置某个条件，随后检查这个条件是否满足跳转条件(也有可能是无条件跳转)，如果满足就跳转，反之继续顺序执行这个模板做的。

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

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

图 3-11 利用 cmpl 和跳转实现控制转移

3.3.8 函数操作

1.参数传递

编译器使用寄存器保存参数从而实现参数传递，按照%rdi，%rsi，%rdx，%rcx，%r8，%r9 的顺序依次填入第 1-6 个参数。这个程序中并没有出现参数个数大于 6 个的情况，在 64 位的机器下，如果参数个数大于 6 个，那么多出来的其它参数都会被存储在栈上。

```
movq    -32(%rbp), %rax
addq    $16, %rax
movq    (%rax), %rdx        调用printf的参数3
movq    -32(%rbp), %rax
addq    $8, %rax
movq    (%rax), %rax
movq    %rax, %rsi          调用printf的
leaq    .LC1(%rip), %rdi    参数1和参数2
movl    $0, %eax
call    printf@PLT
movq    -32(%rbp), %rax
addq    $24, %rax
movq    (%rax), %rax
movq    %rax, %rdi          调用atoi的参数
call    atoi@PLT
movl    %eax, %edi          调用sleep的参数
call    sleep@PLT
addl    $1, -4(%rbp)
```

图 3-12 利用寄存器实现参数传递

2.函数调用

函数调用的第一步就是上一步的参数传递，参数传递完成后只需要调用 call 指令即可，call 指令相当于是将程序计数器设置为目标调用函数的起始地址，同时把 call 指令后面一条指令的地址存入栈中作为返回地址。

3.函数返回

一般来说被调用函数没有返回值的情况只需要释放栈上分配的空间即可，随后调用 ret 指令回到调用函数的 call 指令的下一行。如果有返回值，要根据返回值的类型确定返回值的存储位置：如果返回值为简单的数据类型，那么编译器会使用%rax 存储返回值；如果返回值为结构体、联合等，那么编译器会把返回值存储在栈中。

以本程序中的 sleep(atoi(arg[3]))为例，atoi 将返回值置于%eax 中，然后 sleep 函数需要一个参数，这个参数应该被存储在%edi 中，所以调用 sleep 函数前还有一个 mov 指令，将%eax 的值传送到%edi

```
call    atoi@PLT
movl    %eax, %edi
call    sleep@PLT
```

图 3-13 %rax 存储返回值

3.4 本章小结

本章主要分析了编译阶段中编译器如何将高级语言转换成汇编语言，以 hello.s 为例描述了包括数据、计算、函数调用等在汇编语言层面的具体实现方式。发现一方面逻辑功能相似的指令往往在汇编语言层面上实现方式也十分类似；另一方面汇编语言在代码顺序上往往和高级语言有一定的差别，特别是控制转移的操作几乎都是使用跳转实现，导致原来的顺序被错开。

（第 3 章 2 分）

第 4 章 汇编

4.1 汇编的概念与作用

概念：在这里指使用汇编器 as，将以.s 为后缀的文件处理成.o 为后缀的文件的过程，这些文件服从可重定位目标文件的格式。此时的文件不再是文本文件，使用 vim 将这个文件打开后得到的是乱码，此时已经是二进制文件。

作用：将汇编代码转为机器指令，使其在链接后能被链接器识别并执行链接操作。

4.2 在 Ubuntu 下汇编的命令

命令：gcc hello.s -c -o hello.o

图 4-1 执行上述命令后获得 hello.o 文件

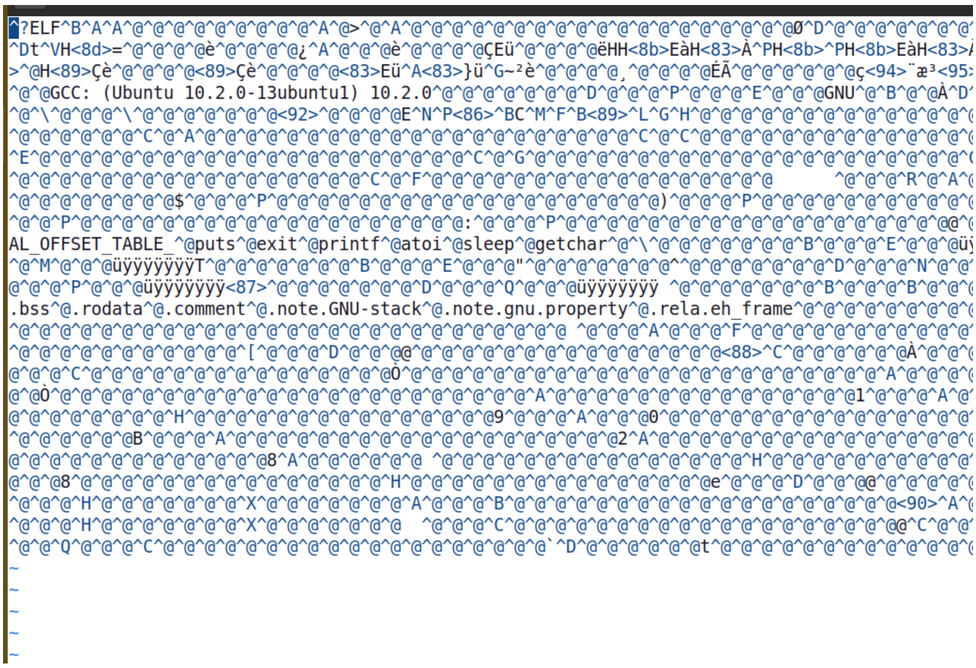

图 4-2 vim 打开 hello.o 得到乱码

4.3 可重定位目标 elf 格式

使用命令：readelf -a hello.o > hello_o_elf.txt 后得到 hello.o 以 ELF 格式解析之后的结果。

1.ELF 头

以一个 16B 的大小的魔数开头，这个魔数描述了该文件的系统的字的大小和字节顺序。剩下的部分包含帮助链接器语法分析和解释目标文件的信息，包括 ELF 头的大小、目标文件的类型、机器类型、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 头：
  Magic:    7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  类别：                             ELF64
  数据：                             2 补码，小端序 (little endian)
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI 版本：                         0
  类型：                             REL (可重定位文件)
  系统架构：                         Advanced Micro Devices X86-64
  版本：                             0x1
  入口点地址：               0x0
  程序头起点：          0 (bytes into file)
  Start of section headers:          1240 (bytes into file)
  标志：             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:         14
  Section header string table index: 13
```

图 4-3 使用 readelf 解析出的 ELF 头信息

2.节头表(节头部表)

简单来说，节头部表显示了 hello.o 中每个节的名字、类型、地址、偏移、每一个节的操作权限等信息。由于 hello.o 还是可重定位目标文件，所以每个节的地址都从 0 开始。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。

```
节头：
  [号] 名称              类型             地址              偏移量
       大小              全体大小          旗标    链接    信息    对齐
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         0000000000000000  00000040
       0000000000000092  0000000000000000  AX       0     0     1
  [ 2] .rela.text        RELA             0000000000000000  00000388
       00000000000000c0  0000000000000018   I      11     1     8
  [ 3] .data             PROGBITS         0000000000000000  000000d2
       0000000000000000  0000000000000000  WA       0     0     1
  [ 4] .bss              NOBITS           0000000000000000  000000d2
       0000000000000000  0000000000000000  WA       0     0     1
  [ 5] .rodata           PROGBITS         0000000000000000  000000d8
       0000000000000033  0000000000000000   A       0     0     8
  [ 6] .comment          PROGBITS         0000000000000000  0000010b
       0000000000000027  0000000000000001  MS       0     0     1
  [ 7] .note.GNU-stack   PROGBITS         0000000000000000  00000132
       0000000000000000  0000000000000000           0     0     1
  [ 8] .note.gnu.pr[...] NOTE             0000000000000000  00000138
       0000000000000020  0000000000000000   A       0     0     8
  [ 9] .eh_frame         PROGBITS         0000000000000000  00000158
       0000000000000038  0000000000000000   A       0     0     8
  [10] .rela.eh_frame    RELA             0000000000000000  00000448
       0000000000000018  0000000000000018   I      11     9     8
  [11] .symtab           SYMTAB           0000000000000000  00000190
       00000000000001b0  0000000000000018          12    10     8
  [12] .strtab           STRTAB           0000000000000000  00000340
       0000000000000048  0000000000000000           0     0     1
  [13] .shstrtab         STRTAB           0000000000000000  00000460
       0000000000000074  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 4-4 hello.o 中节头部表解析出的信息

3.重定位条目所在节

这个节用于记录.text 节中需要进行重定位的信息，根据这些信息之后链接器会对.text 节进行修改，在链接器执行完连接后这个节会被舍弃。

```
重定位节 '.rela.text' at offset 0x388 contains 8 entries:
  偏移量          信息           类型           符号值        符号名称 + 加数
00000000001c  000500000002 R_X86_64_PC32     0000000000000000 .rodata - 4
000000000021  000c00000004 R_X86_64_PLT32    0000000000000000 puts - 4
00000000002b  000d00000004 R_X86_64_PLT32    0000000000000000 exit - 4
000000000054  000500000002 R_X86_64_PC32     0000000000000000 .rodata + 22
00000000005e  000e00000004 R_X86_64_PLT32    0000000000000000 printf - 4
000000000071  000f00000004 R_X86_64_PLT32    0000000000000000 atoi - 4
000000000078  001000000004 R_X86_64_PLT32    0000000000000000 sleep - 4
000000000087  001100000004 R_X86_64_PLT32    0000000000000000 getchar - 4
```

图 4-5 重定位条目所在节

重定位条目的数据结构如图 4-6 所示，包含一个相对于所在节首地址的偏移量、一个绑定符号在.symtab 中的位置的索引、一个可重定位类型、一个有符号常

数，这个常数用于重定位时做一些偏移。重定位条目便于之后链接器对符号进行重定位。

可重定位类型在 ELF 格式中多达 32 种，不给过最常见的只有两种：R_X86_64_PC32 和 R_X86_64_32，分别是 32 位下相对于 PC 的地址引用和绝对地址引用，对应不同的重定位算法。

code/link/elfstructs.c

```
1  typedef struct {
2      long offset;    /* Offset of the reference to relocate */
3      long type:32,   /* Relocation type */
4           symbol:32; /* Symbol table index */
5      long addend;    /* Constant part of relocation expression */
6  } Elf64_Rela;
```

code/link/elfstructs.c

图 4-6 重定位条目数据结构

4.符号表

符号表用于存放程序中定义和引用的函数和全局变量的信息。链接器进行重定位时需要使用这张符号表。

符号表中每一项包含符号的名称、在符号表中索引、原来所在节的索引、类型、绑定属性等信息。

```
Symbol table '.symtab' contains 18 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    9
     9: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
    10: 0000000000000000   146 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 atoi
    16: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND sleep
    17: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND getchar
```

图 4-7 hello.o 中的符号表解析结果

4.4 Hello.o 的结果解析

整体上说，反汇编得到的代码和 hello.s 相比，在行为上几乎完全一致。只是部分表示方式发生了变化：

1.跳转的表示：在汇编代码种，跳转使用的是.L 开头的标记，jmp 及条件 jmp 指令的操作数也是使用对应的.L 开头的标记。但是在反汇编代码中，跳转使用的是一个常数，这个常数表示的是相对 main 函数起始地址的偏移量。

2.函数调用：这个类似于跳转，在汇编代码中函数的调用使用的是函数名称，而在反汇编代码中使用的也是 main 加上偏移量。除此之外，还会在.rela.text 节中添加一条重定位条目以便于链接器进行重定位。

3.访问全局变量：在汇编代码中，访问全局变量使用的是类似于.L0(%rip)这样的方式，而在反汇编代码中使用的是 0x0(%rip)这样暂时没有意义的表示方法，同时在.rela.text 节中添加重定位条目等待处理。

使用objdump对hello.o进行解析的反汇编文件

```
0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 83 ec 20             sub    $0x20,%rsp
   c:   89 7d ec                mov    %edi,-0x14(%rbp)
   f:   48 89 75 e0             mov    %rsi,-0x20(%rbp)
  13:   83 7d ec 04             cmpl   $0x4,-0x14(%rbp)
  17:   74 16                   je     2f <main+0x2f>
  19:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 20 <main+0x20>
                        1c: R_X86_64_PC32       .rodata-0x4
  20:   e8 00 00 00 00          callq  25 <main+0x25>
                        21: R_X86_64_PLT32      puts-0x4
  25:   bf 01 00 00 00          mov    $0x1,%edi
  2a:   e8 00 00 00 00          callq  2f <main+0x2f>
                        2b: R_X86_64_PLT32      exit-0x4
  2f:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  36:   eb 48                   jmp    80 <main+0x80>
  38:   48 8b 45 e0             mov    -0x20(%rbp),%rax
  3c:   48 83 c0 10             add    $0x10,%rax
  40:   48 8b 10                mov    (%rax),%rdx
  43:   48 8b 45 e0             mov    -0x20(%rbp),%rax
  47:   48 83 c0 08             add    $0x8,%rax
  4b:   48 8b 00                mov    (%rax),%rax
  4e:   48 89 c6                mov    %rax,%rsi
  51:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 58 <main+0x58>
                        54: R_X86_64_PC32       .rodata+0x22
  58:   b8 00 00 00 00          mov    $0x0,%eax
  5d:   e8 00 00 00 00          callq  62 <main+0x62>
                        5e: R_X86_64_PLT32      printf-0x4
  62:   48 8b 45 e0             mov    -0x20(%rbp),%rax
  66:   48 83 c0 18             add    $0x18,%rax
  6a:   48 8b 00                mov    (%rax),%rax
  6d:   48 89 c7                mov    %rax,%rdi
  70:   e8 00 00 00 00          callq  75 <main+0x75>
                        71: R_X86_64_PLT32      atoi-0x4
  75:   89 c7                   mov    %eax,%edi
  77:   e8 00 00 00 00          callq  7c <main+0x7c>
                        78: R_X86_64_PLT32      sleep-0x4
  7c:   83 45 fc 01             addl   $0x1,-0x4(%rbp)
  80:   83 7d fc 07             cmpl   $0x7,-0x4(%rbp)
  84:   7e b2                   jle    38 <main+0x38>
  86:   e8 00 00 00 00          callq  8b <main+0x8b>
                        87: R_X86_64_PLT32      getchar-0x4
  8b:   b8 00 00 00 00          mov    $0x0,%eax
  90:   c9                      leaveq
  91:   c3                      retq
```

hello.s文件

```
main:
.LFB6:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $32, %rsp
        movl    %edi, -20(%rbp)
        movq    %rsi, -32(%rbp)
        cmpl    $4, -20(%rbp)
        je      .L2
        leaq    .LC0(%rip), %rdi
        call    puts@PLT
        movl    $1, %edi
        call    exit@PLT
.L2:
        movl    $0, -4(%rbp)
        jmp     .L3
.L4:
        movq    -32(%rbp), %rax
        addq    $16, %rax
        movq    (%rax), %rdx
        movq    -32(%rbp), %rax
        addq    $8, %rax
        movq    (%rax), %rax
        movq    %rax, %rsi
        leaq    .LC1(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        movq    -32(%rbp), %rax
        addq    $24, %rax
        movq    (%rax), %rax
        movq    %rax, %rdi
        call    atoi@PLT
        movl    %eax, %edi
        call    sleep@PLT
        addl    $1, -4(%rbp)
.L3:
        cmpl    $7, -4(%rbp)
        jle     .L4
        call    getchar@PLT
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
```

图 4-8 objdump 反汇编结果和 hello.s 对比

有关机器语言：

机器语言又若干二进制字节构成。一般包含的信息有：指令类型、寄存器指示符、常数字，根据指令类型的不同，后面两者不一定都有。指令类型给出这条指令需要执行的功能。如果需要寄存器参加，一般来说会有寄存器提示符提示涉及的寄存器，在机器语言中寄存器一般用一个特定的数字表示，而不会像汇编代码中用%+字母的格式表示。如果需要常数，如表示立即数或者偏移指令中就会有常数字。

汇编语言和机器语言的映射关系：

机器语言能够直接被机器执行。但是直接使用机器语言存在不便于阅读、难以记忆的问题。汇编语言本质上也是直接对硬件操作，由于采用了助记符，相比机器语言更加方便书写与阅读。机器语言是比汇编语言更加低级的语言，能直接

在机器上运行，与汇编语言可以相互转化。

4.5 本章小结

本章主要分析了从 hello.s 到 hello.o 的过程。利用 readelf 命令对生成的 hello.o 中各个部分内容进行了分析。随后使用 objdump 对 hello.o 进行了反汇编，将反汇编的结果和上一步中得到的 hello.s 进行了比较。

（第 4 章 1 分）

第 5 章 链接

5.1 链接的概念与作用

概念：指将各种代码和数据片段收集并组合称为一个单一文件的过程，这个文件可以被加载到内存并执行。链接可以执行与编译时，也可以执行于加载时，还可以执行于运行时。

作用：将不同功能的片段组合在一起，形成一个能执行完整功能的程序。链接的存在可以使得分离编译成为可能。这样就可以不用将一个大型的应用程序组成一个巨大的源文件，而是将其分解成更小、更好管理的模块。可以通过独立修改这些模块实现功能的改变。

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 执行上述命令后得到可执行目标文件 hello

5.3 可执行目标文件 hello 的格式

使用命令：readelf -a hello > hello_elf.txt 后可以得到可执行目标文件 hello 的 ELF 格式解析。

1.ELF 头

```
ELF 头：
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  类别：                             ELF64
  数据：                             2 补码，小端序 (little endian)
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI 版本：                         0
  类型：                             EXEC (可执行文件)
  系统架构：                         Advanced Micro Devices X86-64
  版本：                             0x1
  入口点地址：               0x4010f0
  程序头起点：          64 (bytes into file)
  Start of section headers:          14296 (bytes into file)
  标志：             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         12
  Size of section headers:           64 (bytes)
  Number of section headers:         27
  Section header string table index: 26
```

图 5-2 hello 使用 ELF 格式解析后 ELF 头信息

ELF 头中描述了文件的总体格式，包括了程序的入口(也就是 main 函数的位置)，文件本身是一个可执行文件，节头部表的位置，字符串表的索引等信息。

2.节头部表

同可重定位目标文件一样，其节头部表对 hello 中所有的节信息进行了声明，包括大小、在程序中的偏移量等，因此根据节头部表中的信息就可以用定位各个节所占的区间(起始位置，大小)。

```
节头：
  [号] 名称              类型             地址              偏移量
       大小              全体大小          旗标    链接    信息    对齐
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         00000000004002e0  000002e0
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.gnu.pr[...] NOTE             0000000000400300  00000300
       0000000000000020  0000000000000000   A       0     0     8
  [ 3] .note.ABI-tag     NOTE             0000000000400320  00000320
       0000000000000020  0000000000000000   A       0     0     4
  [ 4] .hash             HASH             0000000000400340  00000340
       0000000000000038  0000000000000004   A       6     0     8
  [ 5] .gnu.hash         GNU_HASH         0000000000400378  00000378
       000000000000001c  0000000000000000   A       6     0     8
  [ 6] .dynsym           DYNSYM           0000000000400398  00000398
       00000000000000d8  0000000000000018   A       7     1     8
  [ 7] .dynstr           STRTAB           0000000000400470  00000470
       000000000000005c  0000000000000000   A       0     0     1
  [ 8] .gnu.version      VERSYM           00000000004004cc  000004cc
       0000000000000012  0000000000000002   A       6     0     2
  [ 9] .gnu.version_r    VERNEED          00000000004004e0  000004e0
       0000000000000020  0000000000000000   A       7     1     8
  [10] .rela.dyn         RELA             0000000000400500  00000500
       0000000000000030  0000000000000018   A       6     0     8
  [11] .rela.plt         RELA             0000000000400530  00000530
       0000000000000090  0000000000000018  AI       6    21     8
  [12] .init             PROGBITS         0000000000401000  00001000
       000000000000001b  0000000000000000  AX       0     0     4
  [13] .plt              PROGBITS         0000000000401020  00001020
       0000000000000070  0000000000000010  AX       0     0     16
  [14] .plt.sec          PROGBITS         0000000000401090  00001090
       0000000000000060  0000000000000010  AX       0     0     16
  [15] .text             PROGBITS         00000000004010f0  000010f0
       0000000000000145  0000000000000000  AX       0     0     16
  [16] .fini             PROGBITS         0000000000401238  00001238
       000000000000000d  0000000000000000  AX       0     0     4
  [17] .rodata           PROGBITS         0000000000402000  00002000
       000000000000003b  0000000000000000   A       0     0     8
  [18] .eh_frame         PROGBITS         0000000000402040  00002040
       00000000000000fc  0000000000000000   A       0     0     8
  [19] .dynamic          DYNAMIC          0000000000403e50  00002e50
       00000000000001a0  0000000000000010  WA       7     0     8
  [20] .got              PROGBITS         0000000000403ff0  00002ff0
       0000000000000010  0000000000000008  WA       0     0     8
  [21] .got.plt          PROGBITS         0000000000404000  00003000
```

图 5-3 hello 的节头部表(部分)

3.程序头表

可执行目标文件相对可重定位目标文件来说多出了程序头表这个部分。程序头表也称段头表，使用这张表可以使得可执行文件能够很容易被加载到内存。程序头表描述了如何把可执行文件连续的片映射到连续的内存段。

```
程序头：
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000002a0 0x00000000000002a0  R      0x8
  INTERP         0x00000000000002e0 0x00000000004002e0 0x00000000004002e0
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000005c0 0x00000000000005c0  R      0x1000
  LOAD           0x0000000000001000 0x0000000000401000 0x0000000000401000
                 0x0000000000000245 0x0000000000000245  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000402000 0x0000000000402000
                 0x000000000000013c 0x000000000000013c  R      0x1000
  LOAD           0x0000000000002e50 0x0000000000403e50 0x0000000000403e50
                 0x00000000000001fc 0x00000000000001fc  RW     0x1000
  DYNAMIC        0x0000000000002e50 0x0000000000403e50 0x0000000000403e50
                 0x00000000000001a0 0x00000000000001a0  RW     0x8
  NOTE           0x0000000000000300 0x0000000000400300 0x0000000000400300
                 0x0000000000000020 0x0000000000000020  R      0x8
  NOTE           0x0000000000000320 0x0000000000400320 0x0000000000400320
                 0x0000000000000020 0x0000000000000020  R      0x4
  GNU_PROPERTY   0x0000000000000300 0x0000000000400300 0x0000000000400300
                 0x0000000000000020 0x0000000000000020  R      0x8
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000002e50 0x0000000000403e50 0x0000000000403e50
                 0x00000000000001b0 0x00000000000001b0  R      0x1
```

图 5-4 hello 的程序头表部分

程序头表中每一项提供了一个 VirtAddr 和一个 PhysAddr 就是用来表示映射关系。除此之外还给出了 Flags 用于表示这一个连续的片的读写权限，以及给出了 Align 用于表示对齐要求。

4.重定位节

对于可执行目标文件来说，仍然会存在重定位信息，因为有些需要动态链接的块还没有被链接，重定位节中就给出了这些符号的相关信息。

```
重定位节 '.rela.dyn' at offset 0x500 contains 2 entries:
  偏移量          信息           类型           符号值        符号名称 + 加数
000000403ff0  000300000006 R_X86_64_GLOB_DAT 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0
000000403ff8  000500000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0

重定位节 '.rela.plt' at offset 0x530 contains 6 entries:
  偏移量          信息           类型           符号值        符号名称 + 加数
000000404018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
000000404020  000200000007 R_X86_64_JUMP_SLO 0000000000000000 printf@GLIBC_2.2.5 + 0
000000404028  000400000007 R_X86_64_JUMP_SLO 0000000000000000 getchar@GLIBC_2.2.5 + 0
000000404030  000600000007 R_X86_64_JUMP_SLO 0000000000000000 atoi@GLIBC_2.2.5 + 0
000000404038  000700000007 R_X86_64_JUMP_SLO 0000000000000000 exit@GLIBC_2.2.5 + 0
000000404040  000800000007 R_X86_64_JUMP_SLO 0000000000000000 sleep@GLIBC_2.2.5 + 0
```

图 5-5 重定位节信息

5.符号表

对于可执行目标文件来说，包含两个符号表，一个符号表的名称为.dynsym，从名称和符号表中的内容来看应该是还没有动态链接的一些未知符号。另一张符号表就是熟知的.symtab，里面保存了程序中定义和引用的函数以及全局变量等的信息。

```
Symbol table '.dynsym' contains 9 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 [...]@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND [...]@GLIBC_2.2.5 (2)
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND [...]@GLIBC_2.2.5 (2)
     5: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     6: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND atoi@GLIBC_2.2.5 (2)
     7: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND exit@GLIBC_2.2.5 (2)
     8: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND sleep@GLIBC_2.2.5 (2)

Symbol table '.symtab' contains 53 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000004002e0     0 SECTION LOCAL  DEFAULT    1
     2: 0000000000400300     0 SECTION LOCAL  DEFAULT    2
     3: 0000000000400320     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000400340     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000400378     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000400398     0 SECTION LOCAL  DEFAULT    6
     7: 0000000000400470     0 SECTION LOCAL  DEFAULT    7
     8: 00000000004004cc     0 SECTION LOCAL  DEFAULT    8
     9: 00000000004004e0     0 SECTION LOCAL  DEFAULT    9
    10: 0000000000400500     0 SECTION LOCAL  DEFAULT   10
    11: 0000000000400530     0 SECTION LOCAL  DEFAULT   11
    12: 0000000000401000     0 SECTION LOCAL  DEFAULT   12
    13: 0000000000401020     0 SECTION LOCAL  DEFAULT   13
    14: 0000000000401090     0 SECTION LOCAL  DEFAULT   14
    15: 00000000004010f0     0 SECTION LOCAL  DEFAULT   15
    16: 0000000000401238     0 SECTION LOCAL  DEFAULT   16
    17: 0000000000402000     0 SECTION LOCAL  DEFAULT   17
    18: 0000000000402040     0 SECTION LOCAL  DEFAULT   18
    19: 0000000000403e50     0 SECTION LOCAL  DEFAULT   19
```

图 5-6 符号表信息

5.4 hello 的虚拟地址空间

从程序头表中可以得知，程序虚拟地址的最低地址大小为 0x400000，大小为 0x5c0，也就是说从 0x400000 到 0x4005c0 是保存有有效数据的。利用 edb 的 data dump 功能查看这一段的内存，发现是符合我们的预期的

```
LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
               0x00000000000005c0 0x00000000000005c0  R      0x1000
LOAD           0x0000000000001000 0x0000000000401000 0x0000000000401000
               0x0000000000000245 0x0000000000000245  R E    0x1000
LOAD           0x0000000000002000 0x0000000000402000 0x0000000000402000
               0x000000000000013c 0x000000000000013c  R      0x1000
LOAD           0x0000000000002e50 0x0000000000403e50 0x0000000000403e50
```

图 5-7 程序头表中的部分信息

```
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 f0 10 40 00 00 00 00 00 ..>.....  .@.....
00000000:00400020 40 00 00 00 00 00 00 00 d8 37 00 00 00 00 00 00 @.......□7......
00000000:00400030 00 00 00 00 40 00 38 00 0c 00 40 00 1b 00 1a 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 a0 02 00 00 00 00 00 00 a0 02 00 00 00 00 00 00  ....... .......
```

•••

```
00000000:00400580 07 00 00 00 06 00 00 00 00 00 00 00 00 00 00 00 ................
00000000:00400590 38 40 40 00 00 00 00 00 07 00 00 00 07 00 00 00 8@@.............
00000000:004005a0 00 00 00 00 00 00 00 00 40 40 40 00 00 00 00 00 ........@@@.....
00000000:004005b0 07 00 00 00 08 00 00 00 00 00 00 00 00 00 00 00 ................
00000000:004005c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000000:004005d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

图 5-8 从 0x400000 到 0x4005c0

然后下一节，可以看到这一节的起始地址为 0x401000，大小为 0x245，也就是说只有 0x401000 到 0x401245 中存在有效数据，使用 edb 查看对应位置，发现也是符合预期的。

```
00000000:00401000 f3 0f 1e fa 48 83 ec 08 48 8b 05 e9 2f 00 00 48  .. H.↓.H..←/..H
00000000:00401010 85 c0 74 02 ff d0 48 83 c4 08 c3 00 00 00 00 00 .□t.□□H.├.╢.....
00000000:00401020 ff 35 e2 2f 00 00 f2 ff 25 e3 2f 00 00 0f 1f 00 □5-/.. □%¯/.....
00000000:00401030 f3 0f 1e fa 68 00 00 00 00 f2 e9 e1 ff ff ff 90  .. h.... ←£□□□.
00000000:00401040 f3 0f 1e fa 68 01 00 00 00 f2 e9 d1 ff ff ff 90  .. h.... ←□□□□.
00000000:00401050 f3 0f 1e fa 68 02 00 00 00 f2 e9 c1 ff ff ff 90  .. h.... ←□□□□.
00000000:00401060 f3 0f 1e fa 68 03 00 00 00 f2 e9 b1 ff ff ff 90  .. h.... ←□□□□.
00000000:00401070 f3 0f 1e fa 68 04 00 00 00 f2 e9 a1 ff ff ff 90  .. h.... ←¬□□□.
00000000:00401080 f3 0f 1e fa 68 05 00 00 00 f2 e9 91 ff ff ff 90  .. h.... ←.□□□.
00000000:00401090 f3 0f 1e fa f2 ff 25 7d 2f 00 00 0f 1f 44 00 00  ..  □%}/....D..
00000000:004010a0 f3 0f 1e fa f2 ff 25 75 2f 00 00 0f 1f 44 00 00  ..  □%u/....D..
00000000:004010b0 f3 0f 1e fa f2 ff 25 6d 2f 00 00 0f 1f 44 00 00  ..  □%m/....D..
00000000:004010c0 f3 0f 1e fa f2 ff 25 65 2f 00 00 0f 1f 44 00 00  ..  □%e/....D..
00000000:004010d0 f3 0f 1e fa f2 ff 25 5d 2f 00 00 0f 1f 44 00 00  ..  □%]/....D..
00000000:004010e0 f3 0f 1e fa f2 ff 25 55 2f 00 00 0f 1f 44 00 00  ..  □%U/....D..
00000000:004010f0 f3 0f 1e fa 31 ed 49 89 d1 5e 48 89 e2 48 83 e4  .. 1▪I.□^H.─H.¦
00000000:00401100 f0 50 54 49 c7 c0 30 12 40 00 48 c7 c1 c0 11 40  PTI╢□0.@.H╢□□.@
00000000:00401110 00 48 c7 c7 25 11 40 00 ff 15 d2 2e 00 00 f4 90 .H╢╢%.@.□.╨... .
00000000:00401120 f3 0f 1e fa c3 f3 0f 1e fa 55 48 89 e5 48 83 ec  .. ╢ .. UH.¥H.↓
00000000:00401130 20 89 7d ec 48 89 75 e0 83 7d ec 04 74 16 48 8d  .}↓H.u¢.}↓.t.H.
00000000:00401140 3d c3 0e 00 00 e8 46 ff ff ff bf 01 00 00 00 e8 =╢...|F□□□.....|
00000000:00401150 7c ff ff ff c7 45 fc 00 00 00 00 eb 48 48 8b 45 |□□□╢E ....→HH.E
00000000:00401160 e0 48 83 c0 10 48 8b 10 48 8b 45 e0 48 83 c0 08 ¢H.□.H..H.E¢H.□.
00000000:00401170 48 8b 00 48 89 c6 48 8d 3d b1 0e 00 00 b8 00 00 H..H.╢H.=□...ɀ..
00000000:00401180 00 00 e8 19 ff ff ff 48 8b 45 e0 48 83 c0 18 48 ..|.□□□H.E¢H.□.H
00000000:00401190 8b 00 48 89 c7 e8 26 ff ff ff 89 c7 e8 3f ff ff ..H.╢|&□□□.╢|?□□
00000000:004011a0 ff 83 45 fc 01 83 7d fc 07 7e b2 e8 00 ff ff ff □.E ..} .~≡|.□□□
00000000:004011b0 b8 00 00 00 00 c9 c3 66 0f 1f 84 00 00 00 00 00 ɀ....□╢f........
00000000:004011c0 f3 0f 1e fa 41 57 4c 8d 3d 83 2c 00 00 41 56 49  .. AWL.=.,..AVI
00000000:004011d0 89 d6 41 55 49 89 f5 41 54 41 89 fc 55 48 8d 2d ┌AUI. ATA. UH.-
00000000:004011e0 6c 2c 00 00 53 4c 29 fd 48 83 ec 08 e8 0f fe ff l,..SL)▼H.↓.|.□□
00000000:004011f0 ff 48 c1 fd 03 74 1f 31 db 0f 1f 80 00 00 00 00 □H□▼.t.1_........
00000000:00401200 4c 89 f2 4c 89 ee 44 89 e7 41 ff 14 df 48 83 c3 L. L.◦D.□A□.□H.╢
00000000:00401210 01 48 39 dd 75 ea 48 83 c4 08 5b 5d 41 5c 41 5d .H9□u↑H.├.[]A\A]
00000000:00401220 41 5e 41 5f c3 66 66 2e 0f 1f 84 00 00 00 00 00 A^A_╢ff.........
00000000:00401230 f3 0f 1e fa c3 00 00 00 f3 0f 1e fa 48 83 ec 08  .. ╢... .. H.↓.
00000000:00401240 48 83 c4 08 c3 00 00 00 00 00 00 00 00 00 00 00 H.├.╢...........
00000000:00401250 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

图 5-8 从 0x401000 到 0x401245

通过这种方法依次检查程序头表中所有的节，可以发现均是符合的。

5.5 链接的重定位过程分析

使用命令：objdump -d -r hello > hello_objdump.txt 就可以获取到可执行目标文件 hello 的反汇编代码，使用 vim 编辑器打开后和 hello.o 的分汇编代码进行比较，可以发现如下的不同：

1.hello 的反汇编代码中已经有明确的虚拟地址，而 hello.o 的反汇编代码中的 main 函数的起始地址还是 0，说明还没有经过重定位。说明链接的过程中需要为代码确定虚拟内存空间中的地址。

```
0000000000401125 <main>:
  401125:       f3 0f 1e fa             endbr64
  401129:       55                      push   %rbp
  40112a:       48 89 e5                mov    %rsp,%rbp
  40112d:       48 83 ec 20             sub    $0x20,%rsp
  401131:       89 7d ec                mov    %edi,-0x14(%rbp)
  401134:       48 89 75 e0             mov    %rsi,-0x20(%rbp)
  401138:       83 7d ec 04             cmpl   $0x4,-0x14(%rbp)
  40113c:       74 16                   je     401154 <main+0x2f>
  40113e:       48 8d 3d c3 0e 00 00    lea    0xec3(%rip),%rdi
  401145:       e8 46 ff ff ff          callq  401090 <puts@plt>
  40114a:       bf 01 00 00 00          mov    $0x1,%edi
  40114f:       e8 7c ff ff ff          callq  4010d0 <exit@plt>
  401154:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  40115b:       eb 48                   jmp    4011a5 <main+0x80>
  40115d:       48 8b 45 e0             mov    -0x20(%rbp),%rax
```

可执行目标文件

```
0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 83 ec 20             sub    $0x20,%rsp
   c:   89 7d ec                mov    %edi,-0x14(%rbp)
   f:   48 89 75 e0             mov    %rsi,-0x20(%rbp)
  13:   83 7d ec 04             cmpl   $0x4,-0x14(%rbp)
  17:   74 16                   je     2f <main+0x2f>
  19:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi
                        1c: R_X86_64_PC32       .rodata-0x4
  20:   e8 00 00 00 00          callq  25 <main+0x25>
                        21: R_X86_64_PLT32      puts-0x4
  25:   bf 01 00 00 00          mov    $0x1,%edi
  2a:   e8 00 00 00 00          callq  2f <main+0x2f>
                        2b: R_X86_64_PLT32      exit-0x4
  2f:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  36:   eb 48                   jmp    80 <main+0x80>
  38:   48 8b 45 e0             mov    -0x20(%rbp),%rax
  3c:   48 83 c0 10             add    $0x10,%rax
```

可重定位目标文件

图 5-9 可执行目标文件与可重定位目标文件地址比较

2.hello 的反汇编代码中多出了一些节，包括.init、.plt、.plt.sec、.fini，同时在代码节中也添加了如<_start>这样的内容。这些内容主要是有关程序初始化的一些需要执行的代码、动态链接相关的一些信息、程序正常终止需要执行的代码等。

```
0000000000401090 <puts@plt>:
  401090:       f3 0f 1e fa             endbr64
  401094:       f2 ff 25 7d 2f 00 00    bnd jmpq *0x2f7d(%rip)        # 404018 <puts@GLIBC_2.2.5>
  40109b:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)

00000000004010a0 <printf@plt>:
  4010a0:       f3 0f 1e fa             endbr64
  4010a4:       f2 ff 25 75 2f 00 00    bnd jmpq *0x2f75(%rip)        # 404020 <printf@GLIBC_2.2.5>
  4010ab:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)

00000000004010b0 <getchar@plt>:
  4010b0:       f3 0f 1e fa             endbr64
  4010b4:       f2 ff 25 6d 2f 00 00    bnd jmpq *0x2f6d(%rip)        # 404028 <getchar@GLIBC_2.2.5>
  4010bb:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```

图 5-10 hello 反汇编代码中部分多出的节

重定位过程分析：

1. 链接器将所有待合并的目标模块中类型相同的节合并在一起，这个节就作为可执行目标文件的节。然后链接器把运行时的内存地址赋给新的聚合节，赋给输入模块定义的每个节，以及赋给输入模块定义的每个符号，当这一步完成时，程序中每条指令和全局变量都有唯一运行时的地址。

2. 重定位时还有一个主要解决的问题是符号解析，链接器修改代码节和数据节中对每个符号的引用，使他们指向正确的运行时地址。执行这一步，链接器依赖于可重定位目标模块中在汇编阶段创建的重定位条目。重定位条目的每一项是一个固定的数据结构，如图 5-11 所示，这里在第 4 部分已经做了相关的分析。利用重定位算法即可在数据节中确定地址。

code/link/elfstructs.c

```
1  typedef struct {
2      long offset;    /* Offset of the reference to relocate */
3      long type:32,   /* Relocation type */
4           symbol:32; /* Symbol table index */
5      long addend;    /* Constant part of relocation expression */
6  } Elf64_Rela;
```

code/link/elfstructs.c

图 5-11 重定位条目数据结构

```
1   foreach section s {
2       foreach relocation entry r {
3           refptr = s + r.offset;  /* ptr to reference to be relocated */
4
5           /* Relocate a PC-relative reference */
6           if (r.type == R_X86_64_PC32) {
7               refaddr = ADDR(s) + r.offset; /* ref's run-time address */
8               *refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr);
9           }
10
11          /* Relocate an absolute reference */
12          if (r.type == R_X86_64_32)
13              *refptr = (unsigned) (ADDR(r.symbol) + r.addend);
14      }
15  }
```

图 5-12 重定位算法

5.6 hello 的执行流程

00000000 00401000 <_init>
00000000 00401020 <.plt>
00000000 00401030 <puts@plt>
00000000 00401040 <printf@plt>
00000000 00401050 <getchar@plt>
00000000 00401060 <atoi@plt>
00000000 00401070 <exit@plt>
00000000 00401080 <sleep@plt>
00000000 004010f0 <_start>
00000000 00401120 <_dl_relocate_static_pie>
00000000 00401125 <main>
00000000 004011c0 <__libc_csu_init>
00000000 00401230 <__libc_csu_fini >
00000000 00401238 <_fini >

5.7 Hello 的动态链接分析

使用 readelf 对可执行目标文件 hello 进行解析后，可以在其节头部表中发现如下的内容：

```
[20] .got              PROGBITS         0000000000403ff0  00002ff0
     0000000000000010  0000000000000008  WA       0     0     8
[21] .got.plt          PROGBITS         0000000000404000  00003000
     0000000000000048  0000000000000008  WA       0     0     8
```

图 5-13 hello 的节头部表中 got 的地址

可以发现 got 的地址在 00000000 00403ff0 处，进入 edb 进行调试，在执行 dl_init 前，内容如下：

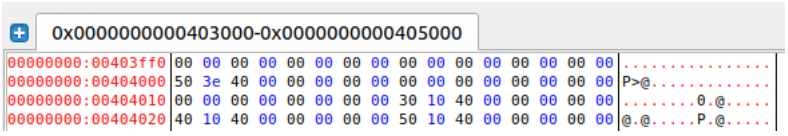

图 5-14 执行 dl_init 前动态链接项目情况

在执行 dl_init 后，内容如下：

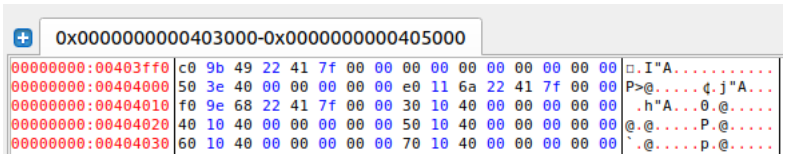

图 5-15 执行 dl_init 后动态链接项目情况

对于变量而言，我们利用代码段和数据段的相对位置不变的技巧计算地址。对于库函数而言，需要 plt、got 合作，plt 初始存的是一批代码，它们跳转到 got 所指示的位置，然后调用链接器。初始时 got 里面存的都是 plt 的第二条指令，随

后链接器修改 got，下一次再调用 plt 时，指向的就是正确的内存地址。plt 就能跳转到正确的区域。

5.8 本章小结

本章主要分析了 linux 中链接的过程。详细介绍了 hello.o 的 ELF 格式和各个节的含义，并且分析了 hello 的虚拟地址空间、重定位过程、执行流程、动态链接过程。并且对比了 hello 与 hello.o 的反汇编代码，更好地掌握了重定位的过程。

（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：一个执行中程序的实例。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。

作用：进程为用户提供了以下假象

1. 我们的程序好像是系统中当前运行的唯一程序一样，并且好像是独占使用处理器和内存。
2. 每次运行程序时，shell 创建一新进程，在这个进程的上下文切换中运行这个可执行目标文件。应用程序也能够创建新进程，并且在新进程的上下文中运行它们自己的代码或其他应用程序。

6.2 简述壳 Shell-bash 的作用与处理流程

作用：Linux 系统中，Shell 是一个交互型应用级程序，代表用户运行其他程序，是命令行解释器，以用户态方式运行的终端进程，其基本功能是解释并运行用户的指令。

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. 检查首个命令行参数是否是一个内置的 shell 命令。
4. 若不是内部命令，调用 fork()创建子进程。
5. 在子进程中，用步骤 2 获取的参数，调用 execve 执行指定程序。
6. 如果用户未要求后台运行，则 shell 使用 waitpid 等待作业终止后返回。
7. 若要求后台运行，则 shell 返回。

6.3 Hello 的 fork 进程创建过程

根据 shell 的处理流程，可以推断，输入命令执行 hello 后，父进程如果判断不是内部指令，即会通过 fork 函数创建子进程。子进程与父进程近似，并得到一份与父进程用户级虚拟空间相同且独立的副本——包括数据段、代码、共享库、堆和用户栈。父进程打开的文件，子进程也可读写。二者之间最大的不同或许在于 PID 的不同。Fork 函数只会被调用一次，但会返回两次，在父进程中，fork 返回子进程的 PID，在子进程中，fork 返回 0。

6.4 Hello 的 execve 过程

execve 函数在当前进程的上下问中加载并运行一个新程序，函数原型如下所示：

```
#include <unistd.h>

int execve(const char *filename, const char *argv[],
           const char *envp[]);
                                    如果成功，则不返回，如果错误，则返回－1。
```

图 6-1 execve 函数

这个函数加载并运行可执行目标文件 filename，且待参数列表 argv 和环境变量列表 envp。只有当出现错误时，比如找不到 filename，execve 才会返回到调用程序，所以这个函数只调用一次并从不返回。

具体来说这个函数的过程：

(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。所有这些区域结构都是私有的，写时复制的。虚拟地址空间的代码和数据区域被映射为 hello 文件的.txt 和.data 区。bss 区域是请求二进制零的，映射匿名文件，其大小包含在 hello 文件中。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，比如标准 C 库 libc.so，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。

(4)设置程序计数器。exceve 做的最后一件事就是设置当前进程的上下文中的程序计数器，使之指向代码区域的入口点。下一次调用这个进程时，它将从这个入口点开始执行。Linux 将根据需要换入代码和数据页面。

6.5 Hello 的进程执行

上下文信息：上下文就是内核重新启动一个被抢占的进程所需要恢复的原来的状态，由寄存器、程序计数器、用户栈、内核栈和内核数据结构等对象的值构成。

上下文切换：当内核选择一个新的进程运行时，则内核调度了这个进程。在内核调度了一个新的进程运行后，它就抢占当前进程，并使用一种称为上下文切换的机制来将控制转移到新的进程：

(1) 保存以前进程的上下文

(2) 恢复新恢复进程被保存的上下文

(3) 将控制传递给这 个新恢复的进程 ，来完成上下文切换。

逻辑控制流：一系列程序计数器 PC 的值的序列叫做逻辑控制流。由于进程是轮流使用处理器的，同一个处理器每个进程执行它的流的一部分后被抢占，然后轮到其他进程。

进程时间片：一个进程执行它的控制流的一部分的每一个时间片段。

用户态和核心态：处理器通常使用一个寄存器提供两种模式的区分，该寄存器描述了进程当前享有的特权，当没有设置模式位时，进程就处于用户态中，处在用户态的进程不允许执行特权指令，也不允许直接引用地址空间中内核区内的代码和数据；设置模式位时，进程进入内核态，该进程可以执行指令集中的任何命令，并且可以访问系统中的任何内存位置。

下面给出一个上下文切换的例子：

如图 6-2 所示，这里展示了进程 A、B 进行的上下文切换。进程 A 最开始运行在用户态下，直到它执行系统调用 read 陷入内核。内核中的陷阱处理程序请求来自磁盘控制器 DMA 传输，并且安排在磁盘控制器完成从磁盘到内存的数据传输后，磁盘中断处理器。

磁盘读取数据需要的时间相对较长，所以内核执行从进程 A 切换到进程 B。在切换到进程 B 前，内核代表进程 A 在内核态下执行指令，然后某一时刻它开始代表进程 B 在内核态下执行指令。在切换之后，进程 B 在用户态下执行指令。

进程 B 在用户模式下运行一段时间后，磁盘发出一个中断信号，表明数据已经从磁盘传输到了内存。内核判定进程 B 已经运行了足够长的时间，于是就执行一个从进程 B 到进程 A 的上下文切换，将控制返回给进程 A 中紧随在系统调用 read 之后的那条指令，进程 A 继续运行。

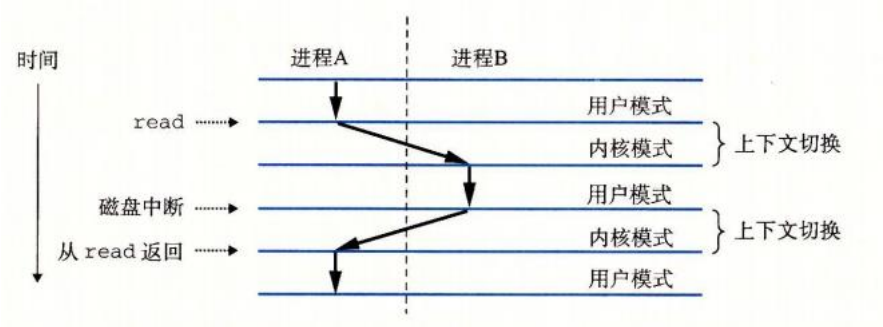

图 6-2 上下文切换示例

6.6 hello 的异常与信号处理

异常的分类：

类别	原因	异步 / 同步	返回行为
中断	来自 I/O 设备的信号	异步	总是返回到下一条指令
陷阱	有意的异常	同步	总是返回到下一条指令
故障	潜在可恢复的错误	同步	可能返回到当前指令
终止	不可恢复的错误	同步	不会返回

图 6-3 异常的分类

异常处理方式：

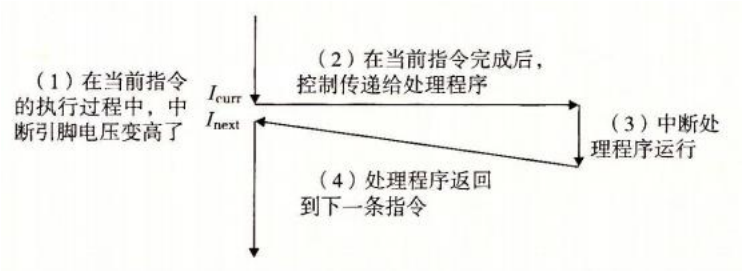

图 6-4 中断处理方式

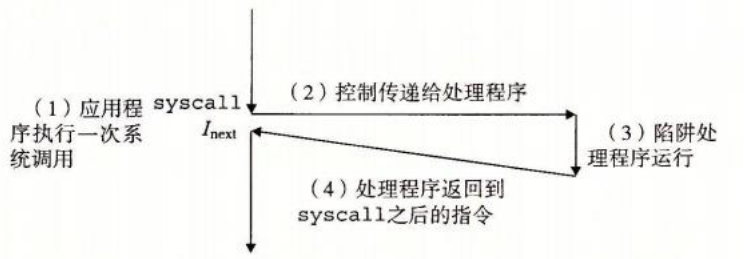

图 6-5 陷阱处理方式

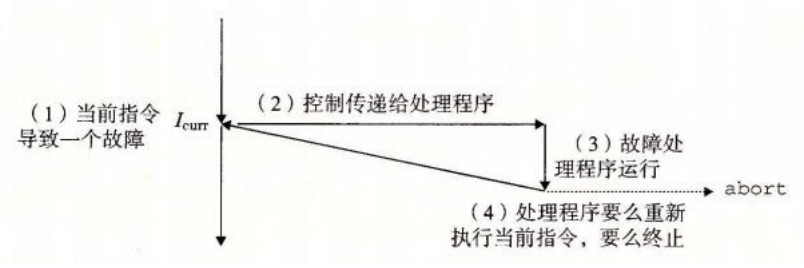

图 6-6 故障处理方式

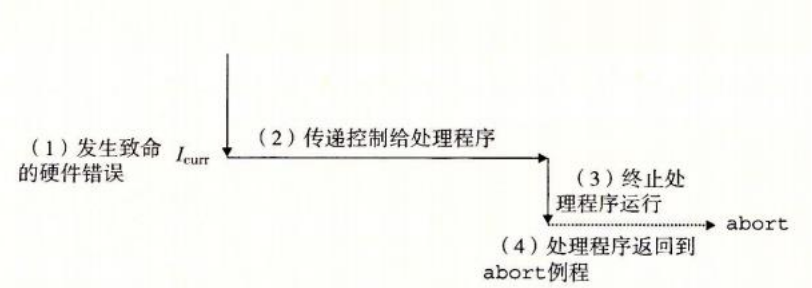

图 6-7 终止处理方式

信号的分类：

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 6-8 信号的常见种类

信号处理：

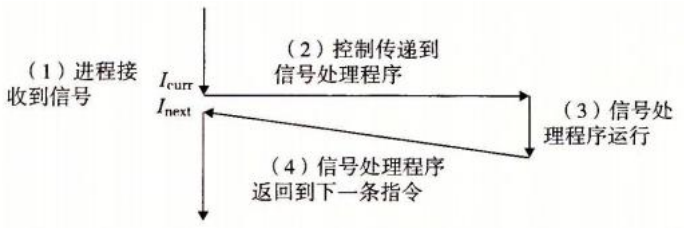

图 6-9 信号处理流程

以 hello 程序为例：

Ctrl+Z：此时内核会发送一个 SIGSTP 信号到前台进程组中的每个进程，这里也就是 hello 进程。默认情况下，结果时挂起前台作业。如图 6-10 所示，此时进程没有终止，使用 ps 命令可以看到 PID 为 9433 的进程是 hello，这个进程依然存在。此时再使用 fg 指令能将其调回前台。

```
edmundlai@edmundlai-virtual-machine:/mnt/hgfs/hitics/lab9$ ./hello 119100311 赖俊宇 2
Hello 119100311 赖俊宇
Hello 119100311 赖俊宇
Hello 119100311 赖俊宇
^Z
[1]+  已停止               ./hello 119100311 赖俊宇 2
edmundlai@edmundlai-virtual-machine:/mnt/hgfs/hitics/lab9$ ps
    PID TTY          TIME CMD
   8477 pts/0    00:00:00 bash
   9433 pts/0    00:00:00 hello
   9434 pts/0    00:00:00 ps
edmundlai@edmundlai-virtual-machine:/mnt/hgfs/hitics/lab9$ fg
./hello 119100311 赖俊宇 2
Hello 119100311 赖俊宇
Hello 119100311 赖俊宇
Hello 119100311 赖俊宇
```

图 6-10 Ctrl+Z 示例

Ctrl+C：这个输入会让内核发送一个 SIGINT 信号到前台进程组中的每个进程，默认情况下是终止前台作业。如图 6-11 所示，输入 Ctrl+C 后进程停止打印，此时再使用 ps 指令查看进程情况，发现没有 hello 进程了，再使用 fg 指令也自然无法生效。

```
Hello 119100311 赖俊宇
Hello 119100311 赖俊宇
Hello 119100311 赖俊宇
^C
edmundlai@edmundlai-virtual-machine:/mnt/hgfs/hitics/lab9$ ps
    PID TTY          TIME CMD
   8477 pts/0    00:00:00 bash
   9448 pts/0    00:00:00 ps
edmundlai@edmundlai-virtual-machine:/mnt/hgfs/hitics/lab9$ fg
bash: fg: 当前: 无此任务
```

图 6-11 Ctrl+C 示例

乱按情况：乱按情况下，在 hello 进程仍然在运行时只是默认在屏幕上显示，就和正常情况下在 shell 中输入任何指令也会显示一样。在 hello 进程结束后，输入的每一行内容都会被试图解释成一条指令。由于是乱按的，这些指令并没有办法被解释成真正有意义的指令。

```
edmundlai@edmundlai-virtual-machine:/mnt/hgfs/hitics/lab9$ ./hello 119100311 赖俊宇 1
Hello 119100311 赖俊宇
dasdas
Hello 119100311 赖俊宇
fsaf
Hello 119100311 赖俊宇
qwe
Hello 119100311 赖俊宇
fgrh
Hello 119100311 赖俊宇
eqe
Hello 119100311 赖俊宇
Hello 119100311 赖俊宇
wq
Hello 119100311 赖俊宇
edmundlai@edmundlai-virtual-machine:/mnt/hgfs/hitics/lab9$ fsaf
Command 'fsaf' not found, did you mean:
  command 'fsa' from deb fsa (1.15.9+dfsg-4build1)
Try: sudo apt install <deb name>
edmundlai@edmundlai-virtual-machine:/mnt/hgfs/hitics/lab9$ qwe
Command 'qwe' not found, did you mean:
  command 'xwe' from deb xwpe (1.5.30a-2.1build3)
  command 'qwo' from deb qwo (0.5-3)
  command 'we' from deb xwpe (1.5.30a-2.1build3)
Try: sudo apt install <deb name>
```

图 6-12 乱按情况示例

kill 命令：这个命令可以给指定的进程发送信号 9，也就是 SIGKILL，杀死对应的进程。如图 6-13 所示，先将 hello 进程挂起，使用 ps 指令查看 hello 进程的 PID，然后使用 kill 指令发送信号，此时再使用 ps 查看进程信息，发现 hello 进程已经消失，使用 fg 命令自然也无法生效。

```
edmundlai@edmundlai-virtual-machine:/mnt/hgfs/hitics/lab9$ ./hello 119100311 赖俊宇 2
Hello 119100311 赖俊宇
Hello 119100311 赖俊宇
Hello 119100311 赖俊宇
^Z
[1]+  已停止               ./hello 119100311 赖俊宇 2
edmundlai@edmundlai-virtual-machine:/mnt/hgfs/hitics/lab9$ ps
    PID TTY          TIME CMD
   8477 pts/0    00:00:00 bash
   9604 pts/0    00:00:00 hello
   9605 pts/0    00:00:00 ps
edmundlai@edmundlai-virtual-machine:/mnt/hgfs/hitics/lab9$ kill -9 9604
edmundlai@edmundlai-virtual-machine:/mnt/hgfs/hitics/lab9$ ps
    PID TTY          TIME CMD
   8477 pts/0    00:00:00 bash
   9606 pts/0    00:00:00 ps
[1]+  已杀死               ./hello 119100311 赖俊宇 2
edmundlai@edmundlai-virtual-machine:/mnt/hgfs/hitics/lab9$ fg
bash: fg: 当前：无此任务
```

图 6-13 kill 命令示例

6.7 本章小结

本章从进程的角度入手，主要描述了 hello 进程的创建、加载、终止的流程。同时结合 shell 中一些内置命令如 kill、Ctrl+C、Ctrl+Z 等对信号进行了分析，同时

观察了这些信号对进程行为的影响。

（第 6 章 1 分）

第 7 章 hello 的存储管理

7.1 hello 的存储器地址空间

逻辑地址：程序经过编译后出现在汇编代码中的地址，也叫相对地址。逻辑地址用来指定一个操作数或者是一条指令的地址。对于 hello 来说，就是由 hello 产生的与段相关的偏移地址，即 hello.o 使用 objdump 解析后得到的结果中的地址以及 hello 使用 objdump 解析后结果中的地址。

线性地址：也叫虚拟地址，是逻辑地址到物理地址变换之间的中间层。在分段部件中逻辑地址是段中的偏移地址，然后加上基地址就是线性地址。如果启用了分页机制，那么线性地址可以再经过变换以产生一个物理地址。如果没有启用分页机制，那么线性地址直接就是物理地址。对于 hello 来说，就是 hello 运行时 CPU 发出寻址相关的指令中的地址。

虚拟地址：也就是线性地址。

物理地址：CPU 地址总线传来的地址，由硬件电路控制其具体含义。物理地址中很大一部分是留给内存条中的内存的，但也常被映射到其他存储器上（如显存、BIOS 等）。在没有使用虚拟存储器的机器上，虚拟地址被直接送到内存总线上，使具有相同地址的物理存储器被读写；而在使用了虚拟存储器的情况下，虚拟地址不是被直接送到内存地址总线上，而是送到存储器管理单元 MMU，把虚拟地址映射为物理地址。

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。其中段标识符是一个 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。后面三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，也就是段开始位置的线性地址。

全局的段描述符，放在“全局段描述符表(GDT)”中，一些局部的段描述符，放在“局部段描述符表(LDT)”中。GDT 在内存中的地址和大小存放在 CPU 的 gdtr 控制寄存器中，而 LDT 则在 ldtr 寄存器中。

给定一个完整的逻辑地址段选择符+段内偏移地址，看段选择符的 T1=0 还是 1，知道当前要转换是 GDT 中的段，还是 LDT 中的段，再根据相应寄存器，得到其地址和大小。

拿出段选择符中前 13 位，可以在这个数组中，查找到对应的段描述符，这样，就有了基地址，然后把基地址加上偏移，就是要转换的线性地址了。

7.3 Hello 的线性地址到物理地址的变换-页式管理

页式管理是一种内存空间存储管理的技术，思路是将各进程的虚拟空间划分成若干个长度相等的页，然后把内存空间划分成一些页框，页框的大小和页的大小相等，然后把页式虚拟地址与内存地址建立一一对应页表，以实现虚拟地址到物理地址的映射。在这种情况下，CPU 只需要生成需要访问的虚拟地址，然后根据页表就可以访问到实际的物理地址，从而解决主存大小和磁盘大小不一致问题下，保证行为的相似性。

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

TLB 是一个小的、虚拟寻址的缓存，其中每一行都保存着一个由单个 PTE 组成的块。TLB 通常有着高度的相联度以保证更高的命中率。

以 Core i7 为例，从 VA 到 PA 的转换如下图所示：

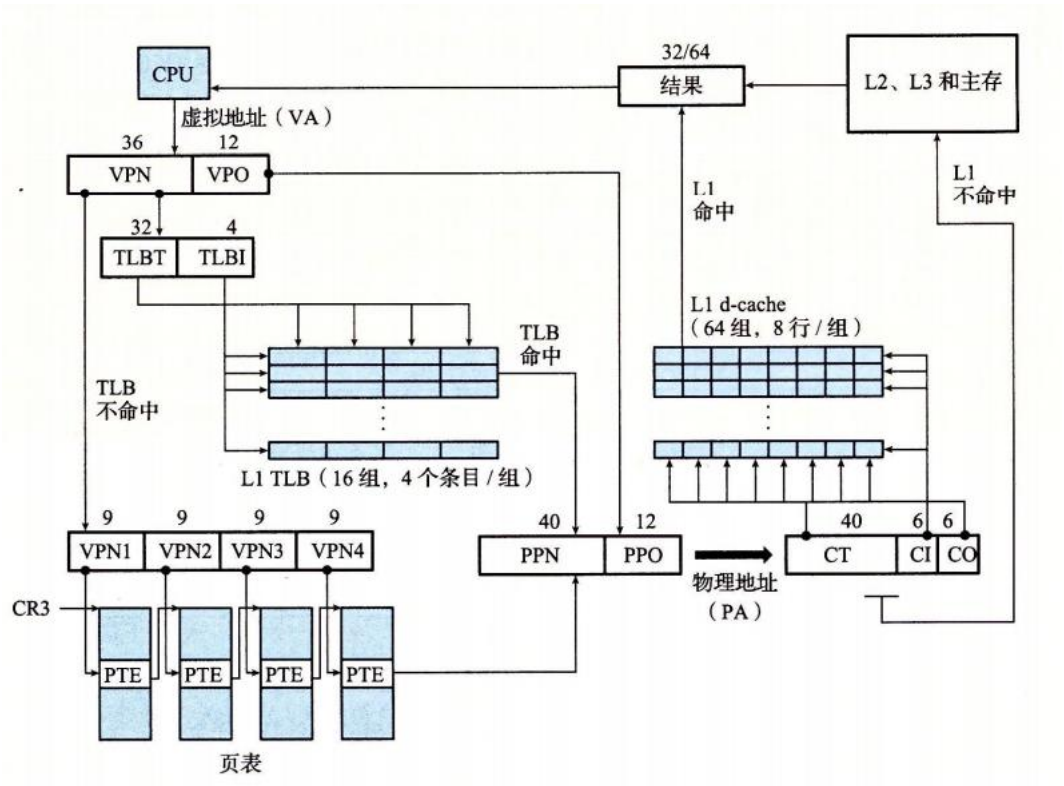

图 7-1 Core i7 地址翻译流程

同样地，对于 Core i7 而言，由于虚拟地址空间非常大，如果只使用一张页表，那么页表本身的大小将使主存无法接受，所以必须使用多级页表，具体来说，使

用了四级页表，如下图所示：

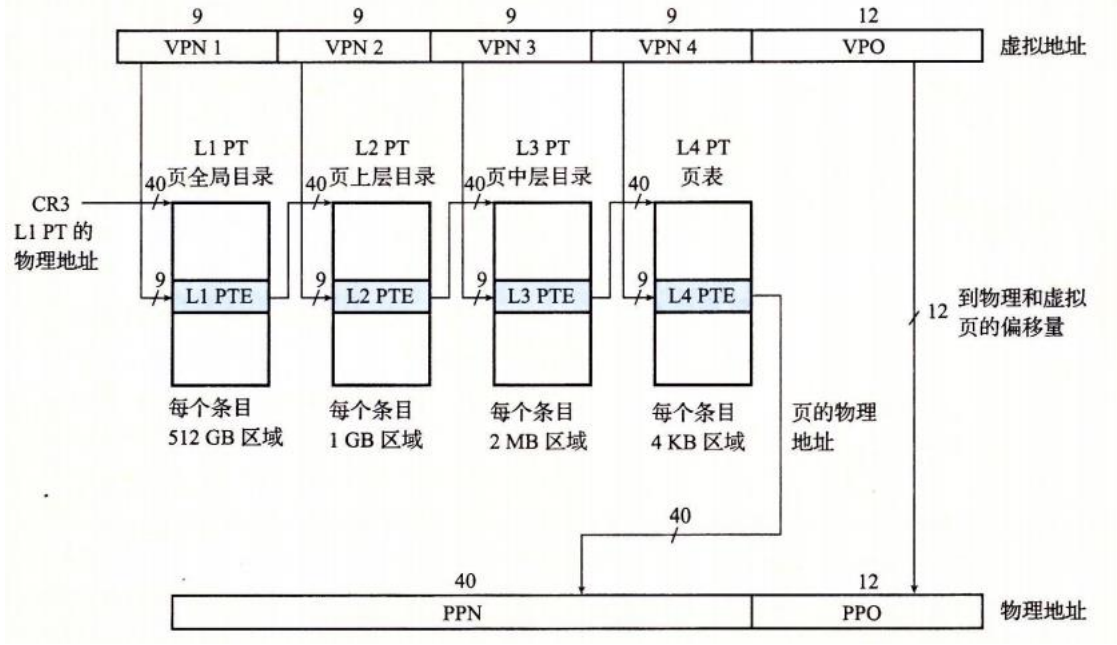

图 7-2 Core i7 四级页表翻译流程

简单来说，CPU 会产生一个虚拟地址 VA，然后传送给 MMU 进行翻译。MMU 首先会在 TLB 中查是否有缓存，如果有缓存的话，VA 此时就直接转换成 PA；如果没有缓存，那么就需要在主存中根据逐级页表进行查询，最终在第四级页表中查到对应结果，形成 PA。如果此时没有查到，说明页表还在磁盘中，没有加载，此时就会引发缺页故障，从而执行对应的异常处理程序。

7.5 三级 Cache 支持下的物理内存访问

物理访存的过程也在图 7-1 中有所体现，这里以 L1 cache 为例，三级 cache 的访问流程基本都是一致的。L1 cache 有 64 组，所以索引位长度为 6；块大小为 64B，所以偏移位长度为 6；物理地址总长度 52 位，那么剩下 40 位就是标记位。

基本访存顺序如下：

(1) 根据组索引在 cache 中找到对应组

(2) 根据标记位找到对应行，如果没有对应行或者对应行的有效位为 0，说明 cache 未命中；只有标记位对应且有效位为 0 都成立才说明命中

(3) 在命中情况下，根据偏移位找到对应字节，把这个字节的内容取出返回给 CPU

(4) 在不命中情况下，需要从存储层次结构中的下一层取出被请求的块，然后将新的块存储在组索引位所指示的组中的一个高速缓存行中。

7.6 hello 进程 fork 时的内存映射

当 fork 函数被当前进程调用时，内核为新进程创建各种数据结构，并分配给它一个唯一的 PID，同时为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位只读，并将两个进程中的每个区域结构都标记为私有的写时复制。

当 fork 在新进程中返回时，新进程现在的虚拟内存刚好和调用 fork 时存在的虚拟内存相同。当这两个进程中的任一个后来进行写操作时，写时复制机制就会创建新页面。因此，也就为每个进程保持了私有空间地址的抽象概念。

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。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。

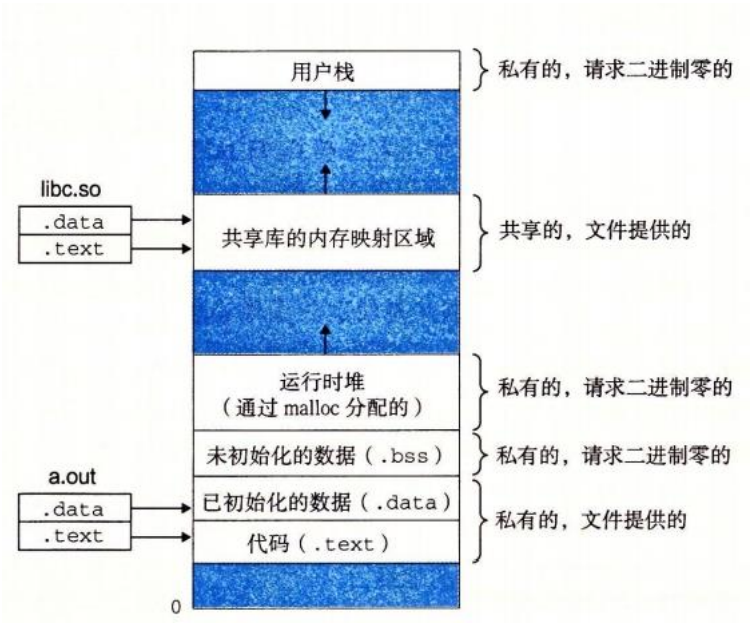

图 7-3 进程地址空间

7.8 缺页故障与缺页中断处理

缺页处理的流程如下：

1. 处理器生成一个虚拟地址，并将它传送给 MMU

2. MMU 生成 PTE 地址，并从高速缓存/主存请求得到它

3. 高速缓存/主存向 MMU 返回 PTE

4. PTE 中的有效位是 0，所以 MMU 出发了一次异常，传递 CPU 中的控制到操作系统内核中的缺页异常处理程序。

5. 缺页处理程序确认出物理内存中的牺牲页，如果这个页已经被修改了，则把它换到磁盘。

6. 缺页处理程序页面调入新的页面，并更新内存中的 PTE

7. 缺页处理程序返回到原来的进程，再次执行导致缺页的命令。CPU 将引起缺页的虚拟地址重新发送给 MMU。因为虚拟页面已经换存在物理内存中，所以就会命中。

图 7-4 也表示了同样的过程。

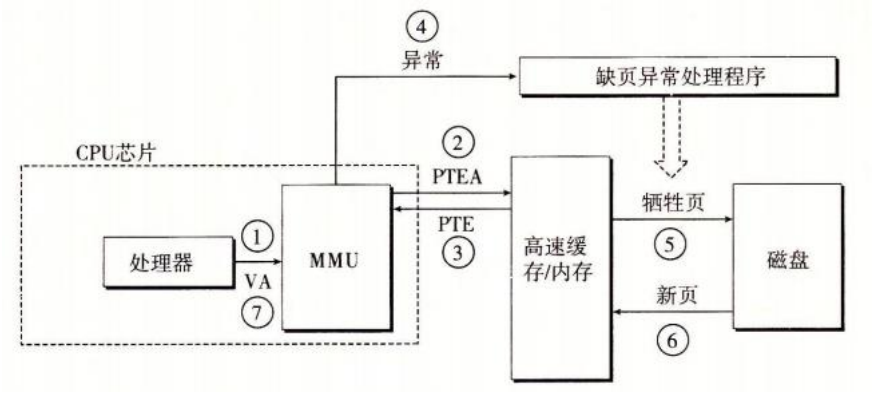

图 7-4 缺页处理

7.9 动态存储分配管理

动态内存分配器维护着一个进程的虚拟内存区域，称为堆。分配器有两种基本风格。它们的不同之处在于由哪个实体来负责释放已分配的块。

1.显式分配器：要求应用显示地释放任何已经分配的块。例如 C 标准库提供一种叫做 malloc 程序包的显式分配器。C 程序通过调用 malloc 函数来分配一个块，并通过调用 free 函数来释放一个块。

2.隐式分配器：也叫垃圾收集器，自动释放未使用的已分配的块的过程叫做垃圾收集。

动态内存分配的方式：

1.隐式空闲链表

将区别块边界、区别已分配块和空闲块的信息嵌入块本身：

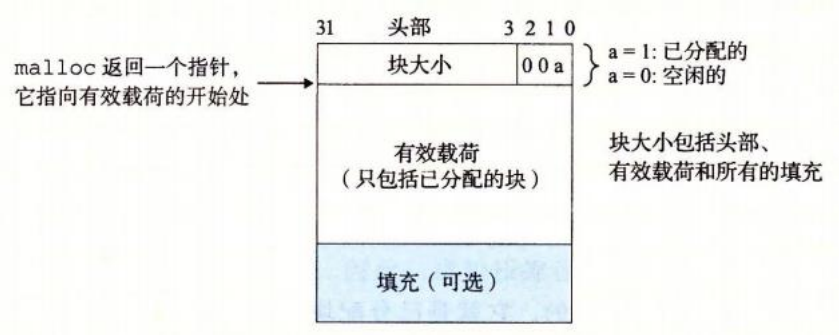

图 7-5 一个简单的堆块的格式

这种情况下，一个块由一个字的头部、有效载荷及可能的填充组成。头部用于记录当前块的基本信息，头部后面就是调用 malloc 时请求的有效载荷。在这种情况下，就可以把堆组织成一个连续的已分配块和空闲块的序列：

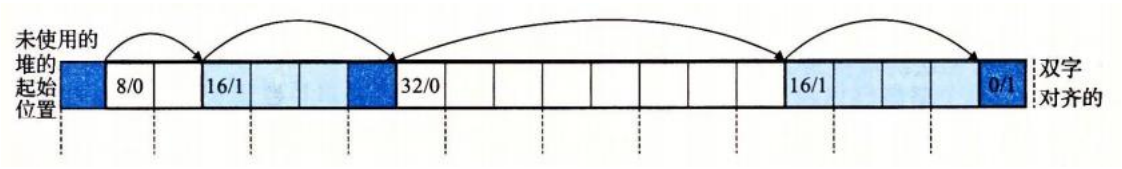

图 7-6 隐式空闲链表

隐式空闲链表中空闲块时通过头部中的大小字段隐含地连接着的。分配器可以遍历堆中所有的块，从而间接地遍历整个空闲块的集合。

2.显式空闲链表

对于隐式空闲链表而言，其块分配和堆块的总数呈线性关系，所以对于通用的分配器，隐式空闲链表时不适合的。一种更好的方式是将空闲块组织成某种形式的显式数据结构。例如，堆可以组织成一个双向的空闲链表，在每个空闲块中，都包含一个前驱和一个后继指针：

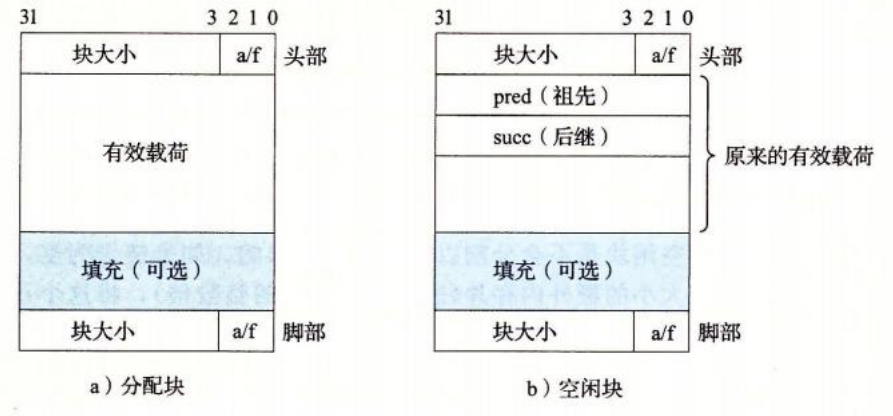

图 7-7 双向空闲链表的堆块格式

维护链表的方式有两种：第一种是后进先出的顺序维护链表；第二种是按照地址顺序来维护链表。

7.10 本章小结

本章主要介绍了 hello 的存储器的地址空间，段页式管理，地址翻译的过程。以及 hello 进程中 fork、execve 函数引发的内存映射，程序执行中发生缺页的处理，以及动态存储分配的管理。

（第 7 章 2 分）

第 8 章 hello 的 I0 管理

8.1 Linux 的 I0 设备管理方法

设备的模型化：文件，指在 Linux 中，所有的 I/O 设备(例如网络、磁盘、终端)都被模型化为文件。

设备管理：unix io 接口，上述的模型化方式将所有设备映射为文件的方式，允许 Linux 内核引出一个简单、低级的应用接口，称为 Unix I/O。我们可以对文件的操作有：打开关闭操作 open 和 close；读写操作 read 和 write；改变当前文件位置 lseek 等。

8.2 简述 Unix I0 接口及其函数

Unix I/O 接口：

打开文件：内核返回一个非负整数的文件描述符，通过对此文件描述符对文件进行所有操作。

Linux shell 创建的每个进程开始时都有三个打开的文件：标准输入（文件描述符 0）、标准输出（描述符为 1），标准出错（描述符为 2）。头文件<unistd.h>定义了常量 STDIN_FILENO、STDOUT_FILENO、STDERR_FILENO，他们可用来代替显式的描述符值。

改变当前的文件位置，文件开始位置为文件偏移量，应用程序通过 seek 操作，可设置文件的当前位置为 k。

读写文件，读操作：从文件复制 n 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；写操作：从内存复制 n 个字节到文件，当前文件位置为 k，然后更新 k

关闭文件：当应用完成对文件的访问后，通知内核关闭这个文件。内核会释放文件打开时创建的数据结构，将描述符恢复到描述符池中。

Unix I/O 函数：

1.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。

2.int close(fd)，fd 是需要关闭的文件的描述符，close 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-1 表示一个错误，0 表示 EOF，文件读取结束，否则返回值表示的是实际传送的字节数量。

4.ssize_t wirte(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

printf 函数的函数体如下：

```
1.  int printf(const char *fmt, ...)
2.  {
3.  int i;
4.  char buf[256];
5.
6.       va_list arg = (va_list)((char*)(&fmt) + 4);
7.       i = vsprintf(buf, fmt, arg);
8.       write(buf, i);
9.
10.      return i;
11.     }
```

可见这个函数中中有关“写”的逻辑在 write 这个系统级函数中，值得注意的是 write 函数上方调用了 vsprintf 函数，这个函数的函数体如下：

```
1.  int vsprintf(char *buf, const char *fmt, va_list args)
2.     {
3.      char* p;
4.      char tmp[256];
5.      va_list p_next_arg = args;
6.
7.      for (p=buf;*fmt;fmt++) {
8.      if (*fmt != '%') {
9.      *p++ = *fmt;
10.     continue;
11.     }
12.
13.     fmt++;
14.
15.     switch (*fmt) {
16.     case 'x':
17.     itoa(tmp, *((int*)p_next_arg));
18.     strcpy(p, tmp);
19.     p_next_arg += 4;
20.     p += strlen(tmp);
21.     break;
22.     case 's':
23.     break;
24.     default:
25.     break;
26.     }
```

```
    }

    return (p - buf);
}
```

可以知道这个函数的功能就是返回打印字符的长度，在获得长度后，就可以通过调用 write 系统函数来实现打印功能。

最终可以查看 write 这个函数的汇编代码，得到如下的结果：

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

这里[esp+4]和[esp+8]分别代表需要打印字符所在的地址和打印字符长度，也就是调用 write 时的两个参数，最后一行代码以 int 开头，说明这是一个系统调用。syscall 将字符串中的字节从寄存器中通过总线复制到显卡的显存中，显存中存储的是字符的 ASCII 码。字符显示驱动子程序将通过 ASCII 码在字模库中找到点阵信息将点阵信息存储到 vram 中。显示芯片会按照一定的刷新频率逐行读取 vram，并通过信号线向液晶显示器传输每一个点（RGB 分量）。于是我们的打印字符串就显示在了屏幕上。

8.4 getchar 的实现分析

getchar 函数的源码如下：

```
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;
}
```

getchar 有一个 int 型的返回值。当程序调用 getchar 时，程序就等着用户按键，用户输入的字符被存放在键盘缓冲区中直到用户按回车为止(回车字符也放在缓冲区中)。

当用户键入回车之后，getchar 才开始从标准输入流中每次读入一个字符。getchar 函数的返回值是用户输入的第一个字符的 ascii 码,如出错返回-1,且将用户输入的字符回显到屏幕。如用户在按回车之前输入了不止一个字符,其他字符会保

留在键盘缓存区中,等待后续 getchar 调用读取。也就是说,后续的 getchar 调用不会等待用户按键,而直接读取缓冲区中的字符,直到缓冲区中的字符读完为后,才等待用户按键。

8.5 本章小结

本章介绍了 Linux 的 I/O 设备的基本概念和管理方法，以及 Unix I/O 接口及其函数。最后分析了 printf 函数和 getchar 函数的工作过程。

（第 8 章 1 分）

结论

hello 所经历的过程：

1. hello.c 经过预编译，拓展得到 hello.i 文本文件
2. hello.i 经过编译，得到汇编代码 hello.s 文本文件
3. hello.s 经过汇编，得到可重定位目标文件 hello.o，此时为二进制文件
4. hello.o 经过链接，生成了可执行文件 hello，二进制文件
5. shell 进程调用 fork 函数，生成子进程；并由 execve 函数加载运行当前进程的上下文中加载并运行新程序 hello
6. hello 在程序执行时会有地址转换，此时需要硬件和软件相互协作
7. hello 运行时会调用一些函数，比如 printf 函数，这些函数与 linux I/O 的设备密切相关
8. hello 进程终止后最终被 shell 父进程回收，内核会收回为其创建的所有数据结构

感悟：

计算机系统的设计是一个复杂而巧妙，同时也有历史性的过程。从最初的 8086CPU 发展到现在的 Core 系列，可以发现相当一部分的设计思想贯穿了 Intel 的 CPU 历史。只是很多新技术、新思路的引入使得 CPU 的集成度、算力逐渐增强。

在操作系统部分，操作系统设计最精妙的地方在于抽象，文件、虚拟内存、进程三个抽象概念的提出，这使得整个计算机对硬件资源的管理可以有效地进行，同时为软件提供统一的接口，是硬件和软件之间的关键部分。

《深入理解计算机系统》这本书的核心理念在读完后才有所感悟，它涉及的知识可能比较底层，但是它的思想是指导程序员如何写出高效的软件，学完后感觉自己仍有不足，待以后操作系统相关课程进一步深化知识体系。

（结论 0 分，缺失 -1 分，根据内容酌情加分）

附件

列出所有的中间产物的文件名，并予以说明起作用。

文件名	作用
hello.c	源代码
hello.i	对 hello.c 预处理后的文件
hello.s	对 hello.i 进行编译后的文件
hello.o	对 hello.s 进行汇编后的文件
hello	对 hello.o 进行链接后的文件
hello_o_elf	用 readelf 对 hello.o 进行解析的结果
hello_o_objdump	用 objdump 对 hello.o 进行反汇编的结果
hello_elf	用 readelf 对 hello 进行解析的结果
hello_objdump	用 objdump 对 hello 进行反汇编的结果

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

[2] 《深入理解计算机系统》 Randal E.Bryant David R.O’Hallaron 机械工业出版社

（参考文献 0 分，缺失 -1 分）