

哈爾濱工業大學

計算機系統

大作業

題	目	<u>程序人生-Hello's P2P</u>
專	業	<u>計算機類</u>
學	號	<u>1170301027</u>
班	級	<u>1703010</u>
學	生	<u>馮帥</u>
指	導	教
師		<u>史先俊</u>

計算機科學與技術學院

2018 年 12 月

## 摘 要

本论文 主要讲述了 hello.c 源程序从预处理到编译到汇编到链接等一系列操作完成从源程序到可执行程序的转化，同时也是计算机如何完成 hello.c 从 program 到 process 的转化，同时又介绍了 Linux 系统下的 shell bash 技术对可执行文件的处理，从创建子进程，到执行程序，到上下文切换，到内存管理，到异常信号处理，到父进程对子进程的回收等等，系统又详细的说明了 Linux 系统的各方面协调工作的能力，从多方面阐述了 Linux 系统如何为可执行文件提供了一个如此优秀全面的运行平台

**关键词：**编译，运行，进程，信号处理，

**（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

<b>第 1 章 概述.....</b>	<b>- 4 -</b>
1.1 HELLO 简介.....	- 4 -
1.2 环境与工具.....	- 5 -
1.3 中间结果.....	- 5 -
1.4 本章小结.....	- 5 -
<b>第 2 章 预处理.....</b>	<b>- 7 -</b>
2.1 预处理的概念与作用.....	- 7 -
2.2 在 UBUNTU 下预处理的命令.....	- 8 -
2.3 HELLO 的预处理结果解析.....	- 8 -
2.4 本章小结.....	- 8 -
<b>第 3 章 编译.....</b>	<b>- 10 -</b>
3.1 编译的概念与作用.....	- 10 -
3.2 在 UBUNTU 下编译的命令.....	- 10 -
3.3 HELLO 的编译结果解析.....	- 10 -
3.4 本章小结.....	- 15 -
<b>第 4 章 汇编.....</b>	<b>- 16 -</b>
4.1 汇编的概念与作用.....	- 16 -
4.2 在 UBUNTU 下汇编的命令.....	- 16 -
4.3 可重定位目标 ELF 格式.....	- 16 -
4.4 HELLO.O 的结果解析.....	- 18 -
4.5 本章小结.....	- 20 -
<b>第 5 章 链接.....</b>	<b>- 21 -</b>
5.1 链接的概念与作用.....	- 21 -
5.2 在 UBUNTU 下链接的命令.....	- 21 -
5.3 可执行目标文件 HELLO 的格式.....	- 21 -
5.4 HELLO 的虚拟地址空间.....	- 23 -
5.5 链接的重定位过程分析.....	- 24 -
5.6 HELLO 的执行流程.....	- 26 -
5.7 HELLO 的动态链接分析.....	- 26 -
5.8 本章小结.....	- 28 -
<b>第 6 章 HELLO 进程管理.....</b>	<b>- 29 -</b>
6.1 进程的概念与作用.....	- 29 -

6.2 简述壳 SHELL-BASH 的作用与处理流程.....	- 29 -
6.3 HELLO 的 FORK 进程创建过程.....	- 29 -
6.4 HELLO 的 EXECVE 过程.....	- 30 -
6.5 HELLO 的进程执行.....	- 31 -
6.6 HELLO 的异常与信号处理.....	- 33 -
6.7 本章小结.....	- 35 -
<b>第 7 章 HELLO 的存储管理.....</b>	<b>- 36 -</b>
7.1 HELLO 的存储器地址空间.....	- 36 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理.....	- 36 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理.....	- 38 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换.....	- 40 -
7.5 三级 CACHE 支持下的物理内存访问.....	- 43 -
7.6 HELLO 进程 FORK 时的内存映射.....	- 45 -
7.7 HELLO 进程 EXECVE 时的内存映射.....	- 45 -
7.8 缺页故障与缺页中断处理.....	- 46 -
7.9 动态存储分配管理.....	- 47 -
7.10 本章小结.....	- 49 -
<b>第 8 章 HELLO 的 IO 管理.....</b>	<b>- 51 -</b>
8.1 LINUX 的 IO 设备管理方法.....	- 51 -
8.2 简述 UNIX IO 接口及其函数.....	- 51 -
8.3 PRINTF 的实现分析.....	- 52 -
8.4 GETCHAR 的实现分析.....	- 54 -
8.5 本章小结.....	- 55 -
<b>结论.....</b>	<b>- 55 -</b>
<b>附件.....</b>	<b>- 56 -</b>
<b>参考文献.....</b>	<b>- 57 -</b>

## 第 1 章 概述

### 1.1 Hello 简介

我是 `hello.c`。

在主人有意地将我从普通的 `.txt` 改成 `.c` 的一刹那，我觉察到我的人生可能因此而完全走上了另一条道路。

在 `linux` 终端下，我看到了一条预处理的指令，然后我觉察到什么东西在看着我一样，我虽然只能隐隐约约看到 `cpp` 的字样，但是我却能清醒的觉察到我的身体仿佛被看穿了一样。那一刻，我仿佛像一个不经世事的孩童一般，在 `cpp` 面前毫无保留，接着我看到了我自己！我很惊奇，我发现我的扩展名变了，我变成了 `hello.i`，索性我的大部分还在，只是比之前的我“壮”了一些。就这样我变成了 `hello.i`，窥探过自身我才觉察到我的升华，我感到了自己从十几行代码到上千行代码的前所未有的充实感，但是这还远远不够..

我仿佛像一个冒险家一般，对于外来事物来者不拒，我想要成长，我急切的渴望着力量，于是我走访了 `ccl`，之后我被编译成为 `hello.s`，再然后我找上了 `as`，之后我被汇编成为 `hello.o`，但是，我只觉察到了自己自身的变化，随着变化的越来越多，我的眼光也越来越大。

我诞生到这个世界中肯定有我自身的道理的，我不怕被改变，我只怕自己还没来得及知道自己的使命就悄无声息的离开这个世界，那将是最悲哀的，因而，怀揣着对力量和存在意义的渴望，我义无反顾的冲向了 `ld`，在那里，我见到了好多好多和我一样带着同等标签（`.o`）的一群家伙，我知道，来到这里是对的，一定是上天的安排让我们相遇。就这样，在 `ld` 的帮助下，我们结合（链接）了。

我变成了 `hello`，是的你没有看错，到那时我才真正体会到可执行程序的强大。我真正的接触了高层，步入了上流社会，从此可能走向人生巅峰！

`shell` 中一行启动命令，我觉察到我开始执行我的使命了，`shell` 为我 `fork`，一刹那，我觉察到了子进程的诞生，仿佛是血脉之间的牵挂。（此间便是从 `program` 到 `process` 的全过程）

再然后，我觉察到 `shell` 为我的小子进程 `execve`，`mmap`，分我时间片，不知为何我清晰地认识到，他就是我，我们是一样的，无论 `OS` 与 `MMU` 怎么折腾他，各种辅助（`TLB`、4 级页表、3 级 `Cache`，`Pagefile`）怎么为他加速，`IO`

管理和信号处理如何运作，他都没有抱怨过一句，只是咬紧牙关坚持了下来，那一刻我感觉到仿佛我的内心就是他的内心，我能清晰的读懂他狰狞坚毅面庞下的心情，这是我的演出，不管别人怎么看我，不管我的演技有多么拙劣，也不管我是否只有这一次出场的机会，但是现在这个时间，这里，是我的主场！

运行的时间转瞬即逝，我收到了命令，我要回收这个子程序，这也是命中注定，我看不清他的情绪，但是我心里是欣慰的，他应该也是这样吧，我自顾自的这么认为着，然后，在看到下一个始终信号之前我也陷入了深度睡眠中。

我不知道在我睡着之后外面发生了什么，我也不知道在我再一次清醒过来的时候我会在哪里，还会有这段记忆，但是在我睡着的前一秒，我的内心是欣慰的，至少那个时候还有我自己，知道我真的来过。（从 zero 到 zero）

## 1.2 环境与工具

### 1.2.1 硬件环境

Intel Core i7-7700HQ 2.81GHz, 8GB RAM, 128GB SSD

### 1.2.2 软件环境

Windows10 64 位; Vmware 11; Ubuntu 16.04 LTS 64 位

### 1.2.3 开发工具

Visual Studio 2010 64 位; CodeBlocks 64 位; vi/vim/gedit+gcc; readelf; edb、gdb;

## 1.3 中间结果

中间文件	文件说明
hello.c	原始 c 程序（源程序）
hello.i	预处理操作后生成的文本文件
hello.s	编译之后生成的汇编语言文件
hello.o	汇编之后生成的可重定位文件
hello	链接之后生成的可执行程序
hello.txt	可执行文件 hello 的反汇编语言代码

helloo.txt	可重定位文件 <code>hello.o</code> 的反汇编语言代码
hello.elf	可执行文件 <code>hello</code> 的 ELF 文件格式

## 1.4 本章小结

通过 `hello` 的简介叙述 `hello` 的一生，简述本篇论文主要讲述的各种操作过程以及为了研究这些操作所用到的工具和环境，描述出一个本篇论文的大体结构。

(第 1 章 0.5 分)

## 第 2 章 预处理

### 2.1 预处理的概述与作用

**预处理的概念：**在编译之前进行的处理。预编译器（cpp）根据以字符#开头的命令，将头文件 `stdio.h` 的内容直接插入到 `Hello.c` 文件中，最终的得到一个以 `i` 为扩展名的 C 文件—`Hello.i` 文件。

这个文件的含义同没有经过预处理的源文件是相同的，但内容有所不同。

C 语言的预处理主要有三个方面的内容： 1. 宏定义； 2. 文件包含； 3. 条件编译。预处理命令由#(hash 字符)开头，它独占一行，#之前只能是空白符。以#开头的语句就是预处理命令，不以#开头的语句为 C 中的代码行

e. g:

(1) **【非含参数宏定义】**`#define PI 3.1415926` 把程序中出现的 `PI` 全部换成 `3.1415926`

(2) **【含参数宏定义】**`#define S(a,b) a*b` 把程序中的 `S(a,b)` 参数替换成 `a*b`

(3) **【文件包含】**`#include <stdio.h>` 把 `stdio.h` 头文件添加到当前源文件中，变成源文件的一部分

(4) **【条件编译】**

`#ifdef`

程序段 1

`#endif`

当标识符已经定义时，程序段 1 才参加编译。

**预处理的作用：**预编译器根据程序中以“#”字符开头的一些命令来处理源程序，比如将一些头文件的处理，“#”开头的命令告诉预处理器读取头文件的内容并将其插入到源程序文本中，从而得到一个 `.i` 为扩展名的新的 `c` 程序。

(1) **【宏定义】：**使用宏可提高程序的通用性和易读性，减少不一致性，减少输入错误和便于修改；

(2) **【文件包含】：**已编写好的头文件可以极大程度上帮助程序员缩短代码行数，程序编写过程中必要的方法在已有的头文件中声明，大大缩短了代码的行数，提高代码的整洁性，可读性，有助于软件开发（源程序编写）。

(3) **【条件编译】：**该预处理使得问题或算法的解决方案增多，便于我们选择合适的解决方案。



## 2.2 在 Ubuntu 下预处理的命令

```
fs1170301027@ubuntu:~/hitics$ gcc -m64 -no-pie -fno-PIC -E hello.c > hello.i
fs1170301027@ubuntu:~/hitics$
```

图 2.1-预处理指令

通常: `gcc -E hello.c > hello.i`

## 2.3 Hello 的预处理结果解析



```
打开(O)  hello.i  保存(S)
~/hello

# 1006 "/usr/include/stdlib.h" 3 4
extern int getloadavg (double __loadavg[], int __nelem)
__attribute__((__nothrow__ , __leaf__)) __attribute__((__nonnull__ (1)));
# 1016 "/usr/include/stdlib.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/stdlib-float.h" 1 3 4
# 1017 "/usr/include/stdlib.h" 2 3 4
# 1026 "/usr/include/stdlib.h" 3 4

# 9 "hello.c" 2

# 10 "hello.c"
int sleepsecs=2.5;

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

    if(argc!=3)
    {
        printf("Usage: Hello 学号 姓名! \n");
        exit(1);
    }
    for(i=0;i<10;i++)
    {
        printf("Hello %s %s\n",argv[1],argv[2]);
        sleep(sleepsecs);
    }
    getchar();
    return 0;
}
```

纯文本 制表符宽度: 8 第 3117 行, 第 11 列 插入

图 2.2-hello.i 部分文件

预处理之后的程序从十几行变到了几千行（如图 2.2），且在源程序中编写的代码在程序最下方。

对比于删除所有“#”C 语言指令生成的.i 程序（仍只有几十行），可以确定该阶段预编译器是处理了源程序中的#include 把相应头文件加入到源程序中。

预处理时 cpp 在给源程序加入头文件的过程中对头文件中的某些需要预处理的语句同样需要进行处理，一层层递归，直到完全处理所有“#”才算完成预处理过程

处理结果分析可见，其中只有常量，如数字、字符串、变量等的定义，以及 C 语言的关键字，如 main, if, else, for, while, {, }, +, -, \*, \, 等等

## 2.4 本章小结

预处理可以方便程序员的代码编写，使程序的简洁性大大提高，且预先编辑好的方法函数在头文件中包含，大大降低了代码的编写难度，减少了代码行数，提高可读性，有助于软件开发（源程序编写）效率的提高。

**（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（ccl）将.i 文件（处理过得源程序）编译成.s 文件。由于计算机并不能直接接受和执行用高级语言编写的源程序（此处指.c，.i 文件），因而利用编译器，能将高级语言编写的程序全盘扫描，翻译成用机器语言表示的与之等价的目标程序（此处指.s，即汇编语言程序），该目标程序能被计算机接受和执行，以便后续翻译等操作进行。

**编译的作用：**编译过程中，编译器会对文件内部的语法和语义做处理，（至少扫描源文件一遍），保证无误才能生成目标程序。由此可见，编译操作实际上是对源程序进行整体全面的检查，确保无误才能进一步执行后续操作进而生成可执行文件，所以编译出错的程序注定无法解释和运行。

同时，编译操作实际上将计算机无法理解接受和执行的文件，转变成了低级的但是计算机可以接受和执行的机器语言所写的目标程序，该目标程序是应用汇编语言编写的，它为不同高级语言的不同编译器提供了通用的输出语言，因而编译操作也是从高级语言到机器语言的过渡操作，有不可或缺的作用。

### 3.2 在 Ubuntu 下编译的命令

```
fs1170301027@ubuntu:~/hitics$ gcc -m64 -no-pie -fno-PIC -S hello.i > hello.s
fs1170301027@ubuntu:~/hitics$
```

图 3.3 编译指令

通常：gcc -S hello.i > hello.s

### 3.3 Hello 的编译结果解析

```
main:
.LFB5:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $32, %rsp
    movl    %edi, -20(%rbp)
    movq    %rsi, -32(%rbp)
    cmpl    $3, -20(%rbp)
    je      .L2
    leaq    .LC0(%rip), %rdi
    call    puts@PLT
    movl    $1, %edi
    call    exit@PLT

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

.L4:
    movq    -32(%rbp), %rax
    addq    $16, %rax
    movq    (%rax), %rdx
    movq    -32(%rbp), %rax
    addq    $8, %rax
    movq    (%rax), %rax
    movq    %rax, %rsi
    leaq    .LC1(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    movl    sleepsecs(%rip), %eax
    movl    %eax, %edi
    call    sleep@PLT
    addl    $1, -4(%rbp)

.L3:
    cmpl    $9, -4(%rbp)
    jle     .L4
    call    getchar@PLT
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
```

图 3.2-hello.s

### 3.3.1 数据

hello.c 中的数据类型有：全局变量，局部整型变量，字符串，数组

```

.file    "hello.c"
.text
.globl   sleepsecs
.data
.align   4
.type    sleepsecs, @object
.size    sleepsecs, 4
sleepsecs:
    .long   2
    .section .rodata
.LC0:
    .string "Usage: Hello \345\255\246\345\217\267 \345\247\223\345\220\215\357\274\201"
.LC1:
    .string "Hello %s %s\n"
    .text
    .globl   main
    .type    main, @function

```

图 3.3-hello.s 文件部分

**初步探索：**打开 hello.s 文件，在 main 函数执行之前可以看到看到如上图（图 3.3）情景，可以分析得

程序声明的全局变量为 sleepsecs 并且已经被赋值，存放于 .data 节（存放已初始化的全局变量和静态 c 变量）。上图中可以看到编译器首先将 sleepsecs 在 .text 代码段中声明为全局变量，其次在 .data 段中，设置对齐方式为 4、设置类型为对象、设置大小为 4 字节、设置为 long 类型其值为 2。

.rodata 中声明两个字符串，"Usage: Hello 学号 姓名！ \n" 和 "Hello %s %s\n"，在源程序中可见，上述图片中的 .LC0,.LC1，其中汉字是以 UTF-8 编码形式存在的。

**进一步探索：**对比 hello.c 源程序和 hello.s 汇编语言程序，可得

程序中的局部整型变量 i 存储于栈空间中，阅读汇编语言程序可知 i 存储于 [rbp-4] 中占四个字节。

程序中的参数整型变量 argc，在程序运行时由用户输入字符串个数决定

程序中直接做比较赋值等操作出现的整型数在汇编语言中以立即数的形式出现。

程序中参数字符型指针数组，用于读取用户从命令行中读入的字符串，在程序循环语句中有体现。

### 3.3.2 赋值

```
movl    $0, %eax
```

图 3.4-mov 指令示例

汇编语言中，movx 指令中，x 可以为 qlwbh（其中 q 用于 64 位的长字值，l 为 32 位，w 为 16 位，b 为 8 位，h 为 4 位），上图是给局部整型变量 i 赋值 0 的

操作。

源程序中对于全局变量的赋值操作，详见 3.3.1 节。

### 3.3.3 类型转换

程序中存在的隐式类型转换是在执行“`int sleepsecs = 2.5`”操作时，强制将浮点数类型的 2.5 转化为整数类型（此间并不是四舍五入）。`double/float->int` 进行舍入时候，默认截掉小数部分，近似于向零舍入，但是当数值超范围或者 NaN 时无定义，（通常设置为 Tmin）。

### 3.3.4 算数操作

2) 算术运算指令(20条)  
 ADD、ADC、AAA、DAA: 加法;  
 INC: 加“1”;  
 SUB、SBB、AAS、DAS: 减法;  
 DEC: 减“1”;  
 CMP: 比较;  
 NEG: 求补;  
 MUL、IMUL、AAM: 乘法;  
 DIV、IDIV、AAD: 除法;  
 CBW, CWD: 符号扩展。

图 3.5-汇编语言下的算数运算指令示例

程序中出现的算数操作有如下实例：

<code>subq</code>	<code>\$32, %rsp</code>	//栈分配存储空间
<code>addl</code>	<code>\$1, -4(%rbp)</code>	//i++
<code>addq</code>	<code>\$16, %rax</code>	//argv[1]地址偏移（寻址）
<code>addq</code>	<code>\$8, %rax</code>	//argv[2]地址偏移（寻址）

### 3.3.5 关系操作

程序中出现的关系操作有如下实例：

<code>cmpl</code>	<code>\$3, -20(%rbp)</code>	//argc=3(?)
<code>cmpl</code>	<code>\$9, -4(%rbp)</code>	//i=9(?)

（1）选择语句判断分支

（2）循环语句判断分支

### 3.3.6 数组操作

```

.L4:
    movq    -32(%rbp), %rax
    addq    $16, %rax
    movq    (%rax), %rdx
    movq    -32(%rbp), %rax
    addq    $8, %rax
    movq    (%rax), %rax
    movq    %rax, %rsi
    movl    $.LC1, %edi
    movl    $0, %eax
    call    printf
    movl    sleepsecs(%rip), %eax
    movl    %eax, %edi
    call    sleep
    addl    $1, -4(%rbp)

```

图 3.6-部分循环语句汇编语言示例

在（图 3.6）图示循环体中，第二第五条语句分别索引到 `argv[1]`,`argv[2]`的地址偏移，然后分别将两个地址存储于`%rdx`, `%rsi` 寄存器中，方便以后输出调用。

### 3.3.7 控制转移

程序中的跳转指令有如下实例：

<code>je</code>	<code>.L2</code>	//argc=3 时跳转，进入循环体设置条件处（for 括号里）
<code>jmp</code>	<code>.L3</code>	//设置 i=0 后跳转进入循环体
<code>jle</code>	<code>.L4</code>	//判断 i<9 不满足循环退出条件时候跳转执行循环语句

程序中的一个 if 语句和一个循环语句解析如上

### 3.3.8 函数操作

程序中实现的是函数的调用：

<b>main 函数：</b> 参数： <code>argc</code> , <code>*argv[]</code> 返回： <code>0</code>
<b>printf 函数</b> 参数: <code>string</code> ,相应参数 1， 相应参数 2， ... 返回: <code>void</code>
<b>exit 函数</b> 参数： <code>int</code> （1） 返回： <code>void</code>
<b>sleep 函数</b>

参数: sleepsecs
getchar 函数 参数: void 返回: void

六十四位系统：函数调用（栈）

第二个参数（寄存器传参%rdi %rsi %rdx %rcx %r8 %r9）
第一个参数
返回地址
rbp
栈空间
rsp

### 3.4 本章小结

编译在 C 语言文件生成可执行文件中起着不可或缺的作用,由于编译过程有着对源程序的语法和词法分析,可以初步检验程序中存在的 bug 在哪,（或者有无 bug）,同时他又是将各种高级语言翻译成统一的汇编语言的过程,为后续翻译过程带来一定的便捷性和可行性

（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 hello.s 翻译成机器语言指令，并把这些指令打包成一种叫做可重定位目标程序的格式，并将结果保留在目标文件 hello.o 中。

目标文件中所存放的也就是与源程序等效的目标的机器语言代码。

目标文件由段组成。通常一个目标文件中至少有两个段：

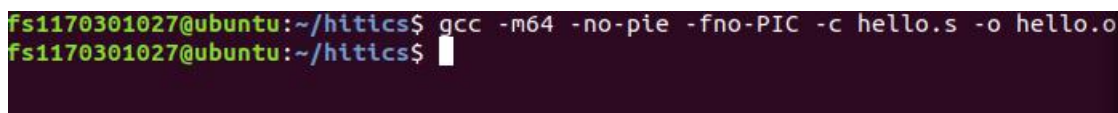
**代码段** 该段中所包含的主要是程序的指令。该段一般是可读和可执行的，但一般却不可写。

**数据段** 主要存放程序中要用到的各种全局变量或静态的数据。一般数据段都是可读，可写，可执行的。

**汇编的作用：**汇编操作将汇编语言程序转变为一个可重定位文件，该文件不是最终的可执行文件，但是该文件可以和一些静态连接库或者动态连接库链接共同加入到可执行文件中，

注意：这儿的汇编是指从 .s 到 .o 即编译后的文件到生成机器语言二进制程序的过程。

### 4.2 在 Ubuntu 下汇编的命令



```
fs1170301027@ubuntu:~/hitics$ gcc -m64 -no-pie -fno-PIC -c hello.s -o hello.o
fs1170301027@ubuntu:~/hitics$
```

图 4.1-汇编指令

通常：gcc -c hello.s -o hello.o

### 4.3 可重定位目标 elf 格式

ELF 文件格式：

1) ELF header (ELF 头)：在文件的开始，保存了路线图，描述了该文件的组织情况。

2) Program header table (程序头部表)：告诉系统如何创建进程映像。用来构造进程映像的目标文件必须具有程序头部表，可重定位文件不需要这个表。

3) Section header table (节头部表)：包含了描述文件节区的信息，每个

节区在表中都有一项，每一项给出诸如节区名称、节区大小这类信息。用于链接的目标文件必须包含节区头部表，其他目标文件可以有，也可以没有这个表。

```
fs1170301027@ubuntu:~/hello$ readelf -h hello.o
ELF 头:
  Magic:      7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  类别:                               ELF64
  数据:                               2 补码, 小端序 (little endian)
  版本:                               1 (current)
  OS/ABI:                               UNIX - System V
  ABI 版本:                               0
  类型:                               REL (可重定位文件)
  系统架构:                               Advanced Micro Devices X86-64
  版本:                               0x1
  入口点地址:                               0x0
  程序头起点:                               0 (bytes into file)
  Start of section headers:               1104 (bytes into file)
  标志:                               0x0
  本头的大小:                               64 (字节)
  程序头大小:                               0 (字节)
  Number of program headers:               0
  节头大小:                               64 (字节)
  节头数量:                               13
  字符串表索引节头:                       12
```

图 4.2-ELF 头

```
fs1170301027@ubuntu:~/hitics$ readelf -S hello.o
There are 13 section headers, starting at offset 0x450:

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                0  0  0
     0000000000000000 0000000000000000
[ 1] .text      PROGBITS    0000000000000000 00000040
     000000000000007d 0000000000000000 AX  0  0  1
[ 2] .rela.text  RELA        0000000000000000 00000310
     00000000000000c0 0000000000000018 I  10  1  8
[ 3] .data      PROGBITS    0000000000000000 000000c0
     0000000000000004 0000000000000000 WA  0  0  4
[ 4] .bss       NOBITS      0000000000000000 000000c4
     0000000000000000 0000000000000000 WA  0  0  1
[ 5] .rodata    PROGBITS    0000000000000000 000000c4
     000000000000002b 0000000000000000 A  0  0  1
[ 6] .comment   PROGBITS    0000000000000000 000000ef
     000000000000002b 0000000000000001 MS  0  0  1
[ 7] .note.GNU-stack PROGBITS    0000000000000000 0000011a
     0000000000000000 0000000000000000  0  0  1
[ 8] .eh_frame   PROGBITS    0000000000000000 00000120
     0000000000000038 0000000000000000 A  0  0  8
[ 9] .rela.eh_frame RELA        0000000000000000 000003d0
     0000000000000018 0000000000000018 I  10  8  8
[10] .symtab     SYMTAB      0000000000000000 00000158
     0000000000000018 0000000000000018  11  9  8
[11] .strtab     STRTAB      0000000000000000 000002d8
     0000000000000037 0000000000000000  0  0  1
[12] .shstrtab   STRTAB      0000000000000000 000003e8
     0000000000000061 0000000000000000  0  0  1

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

图 4.3-节头部表

```
fs1170301027@ubuntu:~/hello$ readelf -r hello.o
```

重定位节 '.rela.text' at offset 0x310 contains 8 entries:

偏移量	信息	类型	符号值	符号名称 + 加数
000000000016	00050000000a	R_X86_64_32	0000000000000000	.rodata + 0
00000000001b	000b00000002	R_X86_64_PC32	0000000000000000	puts - 4
000000000025	000c00000002	R_X86_64_PC32	0000000000000000	exit - 4
00000000004c	00050000000a	R_X86_64_32	0000000000000000	.rodata + 1e
000000000056	000d00000002	R_X86_64_PC32	0000000000000000	printf - 4
00000000005c	000900000002	R_X86_64_PC32	0000000000000000	sleepsecs - 4
000000000063	000e00000002	R_X86_64_PC32	0000000000000000	sleep - 4
000000000072	000f00000002	R_X86_64_PC32	0000000000000000	getchar - 4

重定位节 '.rela.eh\_frame' at offset 0x3d0 contains 1 entry:

偏移量	信息	类型	符号值	符号名称 + 加数
000000000020	000200000002	R_X86_64_PC32	0000000000000000	.text + 0

图 4.4-重定位节

#### 4.4 Hello.o 的结果解析



```

0000000000000000 <main>:
 0: 55                      push   %rbp
 1: 48 89 e5                mov     %rsp,%rbp
 4: 48 83 ec 20             sub     $0x20,%rsp
 8: 89 7d ec                mov     %edi,-0x14(%rbp)
 b: 48 89 75 e0             mov     %rsi,-0x20(%rbp)
 f: 83 7d ec 03            cmpl    $0x3,-0x14(%rbp)
13: 74 14                   je      29 <main+0x29>
15: bf 00 00 00 00          mov     $0x0,%edi
                        16: R_X86_64_32 .rodata
1a: e8 00 00 00 00          callq   1f <main+0x1f>
                        1b: R_X86_64_PC32 puts-0x4
1f: bf 01 00 00 00          mov     $0x1,%edi
24: e8 00 00 00 00          callq   29 <main+0x29>
                        25: R_X86_64_PC32 exit-0x4
29: c7 45 fc 00 00 00 00    movl    $0x0,-0x4(%rbp)
30: eb 39                   jmp     6b <main+0x6b>
32: 48 8b 45 e0             mov     -0x20(%rbp),%rax
36: 48 83 c0 10             add     $0x10,%rax
3a: 48 8b 10                mov     (%rax),%rdx
3d: 48 8b 45 e0             mov     -0x20(%rbp),%rax
41: 48 83 c0 08             add     $0x8,%rax
45: 48 8b 00                mov     (%rax),%rax
48: 48 89 c6                mov     %rax,%rsi
4b: bf 00 00 00 00          mov     $0x0,%edi
                        4c: R_X86_64_32 .rodata+0x1e
50: b8 00 00 00 00          mov     $0x0,%eax
55: e8 00 00 00 00          callq   5a <main+0x5a>
                        56: R_X86_64_PC32 printf-0x4
5a: 8b 05 00 00 00 00      mov     0x0(%rip),%eax    # 60 <main+0x60>
                        5c: R_X86_64_PC32 sleepsecs-0x4
60: 89 c7                   mov     %eax,%edi
62: e8 00 00 00 00          callq   67 <main+0x67>
                        63: R_X86_64_PC32 sleep-0x4
67: 83 45 fc 01             addl    $0x1,-0x4(%rbp)
6b: 83 7d fc 09             cmpl    $0x9,-0x4(%rbp)
6f: 7e c1                   jle     32 <main+0x32>
71: e8 00 00 00 00          callq   76 <main+0x76>
                        72: R_X86_64_PC32 getchar-0x4
76: b8 00 00 00 00          mov     $0x0,%eax
7b: c9                      leaveq  %rax
7c: c3                      retq

```

图 4.5-hello.txt--hello.o 反汇编结果

**对照分析：**上述编译产生的汇编语言程序不存在重定位节之类的信息，同时也没有该汇编操作之后生成的机器语言代码。比较之后可以得出当前的可重定位文件反汇编得到的汇编语言是已经到达机器级别的可以被计算机识别层次上的汇编语言代码（由于机器语言指令），较之编译生成的.s 文件，该汇编语言程序由于存在重定位节，因而多采用 pc 相对或者绝对寻址的方法进行系统函数的调用等，但得到的这两种汇编语言大体并没有区别。

**机器语言与汇编语言：**机器语言是由二进制字节码构成的（图中十六进制表示），程序中每一句汇编语言都对应着一句机器语言指令，换句话说，每一句汇编语言指令都与特定的机器语言指令一一映射，所以本质上看汇编语言程序可以

把每一条指令看成一条机器语言指令，（因为汇编语言只需要一次编译就到机器语言了）。

该反汇编生成程序中，有些操作数和汇编语言不匹配，同时我们也能看到一诸多汇编语言代码中包含着一个个重定位节（例如`.rotate`），相应的函数调用诸如 `R_x86_64_PC32` 等重定位节采用 `pc` 相对寻址得到想要函数的有效地址，而诸如 `R_x86_64_32` 等重定位节采用绝对寻址得到想要函数的有效地址，由于需要重定位，所以有些操作数在对比机器语言和汇编语言的时候会有不一致，但想要实现的操作无差别。

## 4.5 本章小结

汇编操作将汇编语言编译成为机器级的指令，但生成的文件（`.o`）并不是可执行程序，而是可重定位文件，其需要与其他文件或者连接库进行链接生成可执行文件。

**（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：**链接是处理可重定位文件，由链接器负责将所有程序的目标文件与所需的素有附加的目标文件连接起来并最终生成可执行文件。附加的目标文件可以是静态连接库（通常以.a 结尾），也可以是动态链接库（通常以.so 结尾）。**链接的最终产物是可执行文件**

**链接的作用：**链接完成了从重定位文件到可执行文件的转化。由于汇编程序通过汇编生成的目标文件不能被立即执行，其中可能存在着很多没有解决的问题，例如系统函数的调用，所以链接操作将所有程序中可能用到的文件彼此连接成一个可执行文件，这样的可执行文件才能被系统正确的执行。

### 5.2 在 Ubuntu 下链接的命令

```
fs1170301027@ubuntu:~/hitics$ ld -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 /usr/lib/gcc/x86_64-linux-gnu/7/crtbegin.o -L/usr/lib/gcc/x86_64-linux-gnu/7 -L/usr/lib/x86_64-linux-gnu -L/usr/lib -L/lib/x86_64-linux-gnu -L/lib/../lib hello.o -lc /usr/lib/gcc/x86_64-linux-gnu/7/crtend.o /usr/lib/x86_64-linux-gnu/crtn.o -z relro -o hello
fs1170301027@ubuntu:~/hitics$
```

图 5.1-链接指令

指令：`ld -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 /usr/lib/gcc/x86_64-linux-gnu/7/crtbegin.o -L/usr/lib/gcc/x86_64-linux-gnu/7 -L/usr/lib/x86_64-linux-gnu -L/usr/lib -L/lib/x86_64-linux-gnu -L/lib/../lib hello.o -lc /usr/lib/gcc/x86_64-linux-gnu/7/crtend.o /usr/lib/x86_64-linux-gnu/crtn.o -z relro -o hello`

### 5.3 可执行目标文件 hello 的格式

通过 `readelf` 将查看到的所有信息重定向到一个 `hello.elf` 文件中可以看到如下信息

节头部表中（Section Headers）对 `hello` 中所有的节信息进行了声明，其中包括大小 `Size` 以及在程序中的偏移量 `Offset`，因此根据节头中的信息我们就可以用

HexEdit 定位各个节所占的区间（起始位置，大小）。其中地址信息是程序被载入到虚拟地址的起始地址

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	0
[ 1]	.interp	PROGBITS	0000000000400200	00000200
	000000000000001c	0000000000000000	A 0 0	1
[ 2]	.note.ABI-tag	NOTE	000000000040021c	0000021c
	0000000000000020	0000000000000000	A 0 0	4
[ 3]	.hash	HASH	0000000000400240	00000240
	0000000000000034	0000000000000004	A 5 0	8
[ 4]	.gnu.hash	GNU_HASH	0000000000400278	00000278
	000000000000001c	0000000000000000	A 5 0	8
[ 5]	.dynsym	DYNSYM	0000000000400298	00000298
	00000000000000c0	0000000000000018	A 6 1	8
[ 6]	.dynstr	STRTAB	0000000000400358	00000358
	0000000000000057	0000000000000000	A 0 0	1
[ 7]	.gnu.version	VERSYM	00000000004003b0	000003b0
	0000000000000010	0000000000000002	A 5 0	2
[ 8]	.gnu.version_r	VERNEED	00000000004003c0	000003c0
	0000000000000020	0000000000000000	A 6 1	8
[ 9]	.rela.dyn	RELA	00000000004003e0	000003e0
	0000000000000030	0000000000000018	A 5 0	8
[10]	.rela.plt	RELA	0000000000400410	00000410
	0000000000000078	0000000000000018	AI 5 21	8
[11]	.init	PROGBITS	0000000000400488	00000488
	0000000000000017	0000000000000000	AX 0 0	4
[12]	.plt	PROGBITS	00000000004004a0	000004a0
	0000000000000060	0000000000000010	AX 0 0	16
[13]	.text	PROGBITS	0000000000400500	00000500
	00000000000001e2	0000000000000000	AX 0 0	16
[14]	.fini	PROGBITS	00000000004006e4	000006e4
	0000000000000009	0000000000000000	AX 0 0	4
[15]	.rodata	PROGBITS	00000000004006f0	000006f0
	000000000000002f	0000000000000000	A 0 0	4
[16]	.eh_frame	PROGBITS	0000000000400720	00000720
	0000000000000100	0000000000000000	A 0 0	8
[17]	.init_array	INIT_ARRAY	0000000000600e00	00000e00
	0000000000000008	0000000000000008	WA 0 0	8
[18]	.fini_array	FINI_ARRAY	0000000000600e08	00000e08
	0000000000000008	0000000000000008	WA 0 0	8
[19]	.dynamic	DYNAMIC	0000000000600e10	00000e10
	00000000000001e0	0000000000000010	WA 6 0	8
[20]	.got	PROGBITS	0000000000600ff0	00000ff0
	0000000000000010	0000000000000008	WA 0 0	8
[21]	.got.plt	PROGBITS	0000000000601000	00001000
	0000000000000040	0000000000000008	WA 0 0	8
[22]	.data	PROGBITS	0000000000601040	00001040

	0000000000000014	0000000000000000	WA	0	0	8
[23] .bss	NOBITS	0000000000000000	0000000000000000	00001054		
	0000000000000004	0000000000000000	WA	0	0	1
[24] .comment	PROGBITS	0000000000000000	0000000000000000	00001054		
	000000000000002a	0000000000000001	MS	0	0	1
[25] .symtab	SYMTAB	0000000000000000	0000000000000000	00001080		
	00000000000000600	0000000000000018		26	41	8
[26] .strtab	STRTAB	0000000000000000	0000000000000000	00001680		
	0000000000000020e	0000000000000000		0	0	1
[27] .shstrtab	STRTAB	0000000000000000	0000000000000000	0000188e		
	00000000000000e2	0000000000000000		0	0	1

## 5.4 hello 的虚拟地址空间

使用 edb 加载 hello，查看本进程的虚拟地址空间各段信息，并与 5.3 对照分析说明。

程序头:

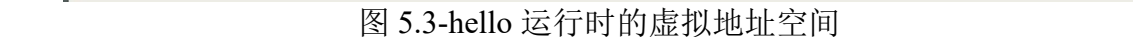
Type	Offset FileSiz	VirtAddr MemSiz	PhysAddr Flags Align
PHDR	0x0000000000000040	0x0000000000000040	0x0000000000000040
	0x00000000000001c0	0x00000000000001c0	R 0x8
INTERP	0x0000000000000200	0x0000000000000200	0x0000000000000200
	0x000000000000001c	0x000000000000001c	R 0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]			
LOAD	0x0000000000000000	0x0000000000000000	0x0000000000000000
	0x0000000000000820	0x0000000000000820	R E 0x200000
LOAD	0x0000000000000e00	0x0000000000000e00	0x0000000000000e00
	0x0000000000000254	0x0000000000000258	RW 0x200000
DYNAMIC	0x0000000000000e10	0x0000000000000e10	0x0000000000000e10
	0x00000000000001e0	0x00000000000001e0	RW 0x8
NOTE	0x000000000000021c	0x000000000000021c	0x000000000000021c
	0x0000000000000020	0x0000000000000020	R 0x4
GNU_STACK	0x0000000000000000	0x0000000000000000	0x0000000000000000
	0x0000000000000000	0x0000000000000000	RW 0x10
GNU_RELRO	0x0000000000000e00	0x0000000000000e00	0x0000000000000e00
	0x0000000000000200	0x0000000000000200	R 0x1

图 5.2-helloELF 文件中的程序头部分

在可执行程序的 ELF 文件中可以看到，程序头（program headers）表在执行的时候被使用，告诉链接器运行时需要加载的内容并且提供动态链接信息。如上图（图 5.2）中可见各节的偏移量，虚拟地址和相映射的物理地址。

对应 edb 所见到的虚拟地址空间为图 5.3。





### 5.5 链接的重定位过程分析

objdump -d -r hello 分析 hello 与 hello.o 的不同，说明链接的过程。  
结合 hello.o 的重定位项目，分析 hello 中对其怎么重定位的。

1) 函数个数: 在使用 ld 命令链接的时候, 指定了动态链接器为 64 的 /lib64/ld-linux-x86-64.so.2, crt1.o、crti.o、crtn.o 中主要定义了程序入口 \_start、初始化函数 \_init, \_start 程序调用 hello.c 中的 main 函数, libc.so 是动态链接共享库, 其中定义了 hello.c 中用到的 printf、sleep、getchar、exit 函数和 \_start 中调用的 libc csu init, libc csu fini, libc start main。链接器将上述函数加入。

2) 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。

- 24 -

```

0000000000000000 <main>:
 0: 55                push    %rbp
 1: 48 89 e5          mov     %rsp,%rbp
 4: 48 83 ec 20       sub     $0x20,%rsp
 8: 89 7d ec          mov     %edi,-0x14(%rbp)
 b: 48 89 75 e0       mov     %rsi,-0x20(%rbp)
 f: 83 7d ec 03       cmpl    $0x3,-0x14(%rbp)
13: 74 14            je      29 <main+0x29>
15: bf 00 00 00 00    mov     $0x0,%edi
16: R_X86_64_32      .rodata
1a: e8 00 00 00 00    callq   1f <main+0x1f>
1b: R_X86_64_PC32    puts-0x4
1f: bf 01 00 00 00    mov     $0x1,%edi
24: e8 00 00 00 00    callq   29 <main+0x29>
25: R_X86_64_PC32    exit-0x4
29: c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%rbp)
30: eb 39            jmp     6b <main+0x6b>
32: 48 8b 45 e0       mov     -0x20(%rbp),%rax
36: 48 83 c0 10       add     $0x10,%rax
3a: 48 8b 10          mov     (%rax),%rdx
3d: 48 8b 45 e0       mov     -0x20(%rbp),%rax
41: 48 83 c0 08       add     $0x8,%rax
45: 48 8b 00          mov     (%rax),%rax
48: 48 89 c6          mov     %rax,%rsi
4b: bf 00 00 00 00    mov     $0x0,%edi
4c: R_X86_64_32      .rodata+0x1e
50: b8 00 00 00 00    mov     $0x0,%eax
55: e8 00 00 00 00    callq   5a <main+0x5a>
56: R_X86_64_PC32    printf-0x4
5a: 8b 05 00 00 00 00 00 mov     0x0(%rip),%eax    # 60 <main+0x60>
5c: R_X86_64_PC32    sleepsecs-0x4
60: 89 c7            mov     %eax,%edi
62: e8 00 00 00 00    callq   67 <main+0x67>
63: R_X86_64_PC32    sleep-0x4
67: 83 45 fc 01       addl    $0x1,-0x4(%rbp)
6b: 83 7d fc 09       cmpl    $0x9,-0x4(%rbp)
6f: 7e c1            jle     32 <main+0x32>
71: e8 00 00 00 00    callq   76 <main+0x76>
72: R_X86_64_PC32    getchar-0x4
76: b8 00 00 00 00    mov     $0x0,%eax
7b: c9              leaveq  %eax

```

图 5.2-hello.o 反汇编程序

```

00000000004005e7 <main>:
4005e7: 55                push    %rbp
4005e8: 48 89 e5          mov     %rsp,%rbp
4005eb: 48 83 ec 20       sub     $0x20,%rsp
4005ef: 89 7d ec          mov     %edi,-0x14(%rbp)
4005f2: 48 89 75 e0       mov     %rsi,-0x20(%rbp)
4005f6: 83 7d ec 03       cmpl    $0x3,-0x14(%rbp)
4005fa: 74 14            je      400610 <main+0x29>
4005fc: bf f4 06 40 00    mov     $0x4006f4,%edi
400601: e8 aa fe ff ff    callq   4004b0 <puts@plt>
400606: bf 01 00 00 00    mov     $0x1,%edi
40060b: e8 d0 fe ff ff    callq   4004e0 <exit@plt>
400610: c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%rbp)
400617: eb 39            jmp     400652 <main+0x6b>
400619: 48 8b 45 e0       mov     -0x20(%rbp),%rax
40061d: 48 83 c0 10       add     $0x10,%rax
400621: 48 8b 10          mov     (%rax),%rdx
400624: 48 8b 45 e0       mov     -0x20(%rbp),%rax
400628: 48 83 c0 08       add     $0x8,%rax
40062c: 48 8b 00          mov     (%rax),%rax
40062f: 48 89 c6          mov     %rax,%rsi
400632: bf 12 07 40 00    mov     $0x400712,%edi
400637: b8 00 00 00 00    mov     $0x0,%eax
40063c: e8 7f fe ff ff    callq   4004c0 <printf@plt>
400641: 8b 05 09 0a 20 00 mov     0x200a09(%rip),%eax    # 601050
<sleepsecs>
400647: 89 c7            mov     %eax,%edi
400649: e8 a2 fe ff ff    callq   4004f0 <sleep@plt>
40064e: 83 45 fc 01       addl    $0x1,-0x4(%rbp)
400652: 83 7d fc 09       cmpl    $0x9,-0x4(%rbp)
400656: 7e c1            jle     400619 <main+0x32>
400658: e8 73 fe ff ff    callq   4004d0 <getchar@plt>
40065d: b8 00 00 00 00    mov     $0x0,%eax
400662: c9              leaveq  %eax
400663: c3              retq
400664: 66 2e 0f 1f 84 00 00 nopw    %cs:0x0(%rax,%rax,1)
40066b: 00 00 00
40066e: 66 90            xchg    %ax,%ax

```

图 5.3-hello 反汇编程序

观察上述两图中的两个反汇编程序可以看到,对比重定位条目之间前后的结果

可以直观看到，这里列举一个 PC 相对寻址的例子，前面有一小节中并未详细叙述其过程，这里稍作补充，

一个 PC 相对寻址就是距离程序计数器（PC）的偏移量。当 CPU 执行到图 5.2 中的 1a 行时候，由于当前汇编模块中，并不知道数据和代码最终将会被放在什么地方，也不知道这个模块引用的任何外部定义的函数或者全局变量的位置（此时指代 puts 函数），所以当前 CPU 会根据执行到的 PC 相对寻址的指令中编码的 32 位置加上 PC 的当前运行时的值，得到有效的地址，（此处指代 call 指令的目标），此时 PC 值通常为下一条指令在内存中的位置，由于当前 PC: 1a, 指令：

```
1a: e8 00 00 00 00      callq 1f<main+0x1f>
    1a 1b 1c 1d 1e
```

此时的 1f 就代表下一条指令所在的地址

## 5.6 hello 的执行流程

使用 edb 执行 hello，说明从加载 hello 到 \_start，到 call main,以及程序终止的所有过程。请列出其调用与跳转的各个子程序名或程序地址。

ld-2.27.so! dl_start	0x00007f0b20e07ea0
ld-2.27.so! dl_init	0x00007f0b20e16630
libc-2.27.so! __libc_start_main	0x00007f0b20a36ae0
Libc-2.27.so! cxa_atexit	0x00007fbd71729430
Libc-2.27.so! setjmp	0x00007fbd71724c10
Hello!puts@plt	0x00000000004004b0
Hello!exit@plt	0x00000000004004e0

## 5.7 Hello 的动态链接分析

（以下格式自行编排，编辑时删除）

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

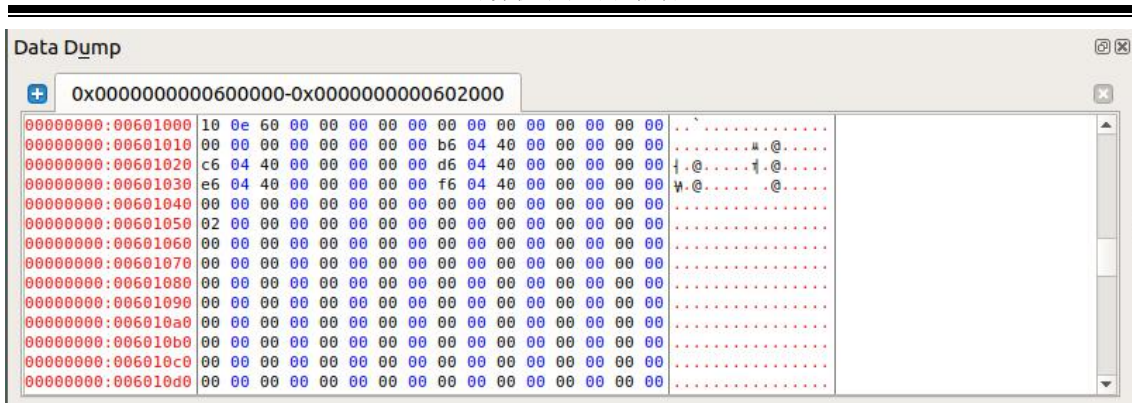


图 5.4.1-调用 dl\_init 之前的 data dump

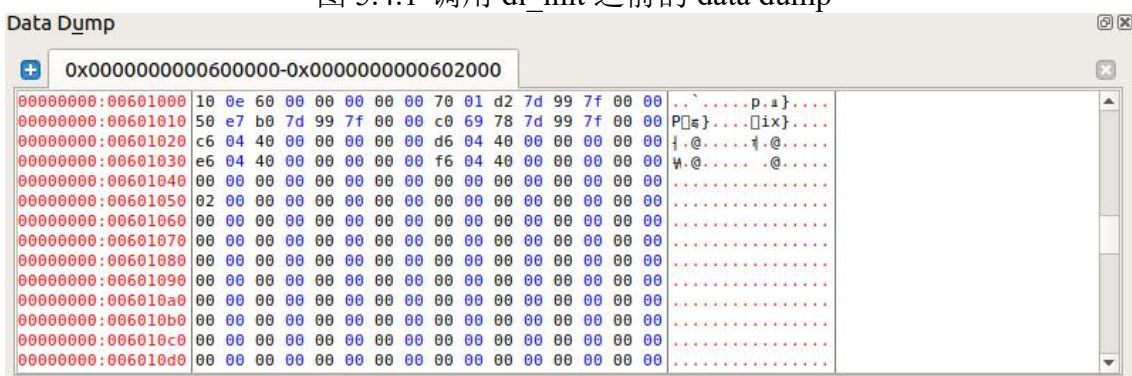


图 5.4.2-调用 dl\_init 之后的的 data dump

根据上两图分析可以看到，虚拟地址对应于 0x60000 开始到 0x6010 结束的部分有明显变化，观察下图所示发现当前变化位置为 GOT 表（全局偏移量表）对应的位置，它隶属于数据段的一部分，由 ELF 文件中重定位条目分析其实就能明确的看到

```
Symbol table '.dynsym' contains 8 entries:
Num:  Value          Size Type  Bind  Vis      Ndx Name
0: 0000000000000000  0 NOTYPE LOCAL DEFAULT UND
1: 0000000000000000  0 FUNC  GLOBAL DEFAULT UND puts@GLIBC_2.2.5 (2)
2: 0000000000000000  0 FUNC  GLOBAL DEFAULT UND printf@GLIBC_2.2.5 (2)
3: 0000000000000000  0 FUNC  GLOBAL DEFAULT UND __libc_start_main@GLIBC_2.2.5 (2)
4: 0000000000000000  0 FUNC  GLOBAL DEFAULT UND getchar@GLIBC_2.2.5 (2)
5: 0000000000000000  0 NOTYPE WEAK  DEFAULT UND __gmon_start__
6: 0000000000000000  0 FUNC  GLOBAL DEFAULT UND exit@GLIBC_2.2.5 (2)
7: 0000000000000000  0 FUNC  GLOBAL DEFAULT UND sleep@GLIBC_2.2.5 (2)
```



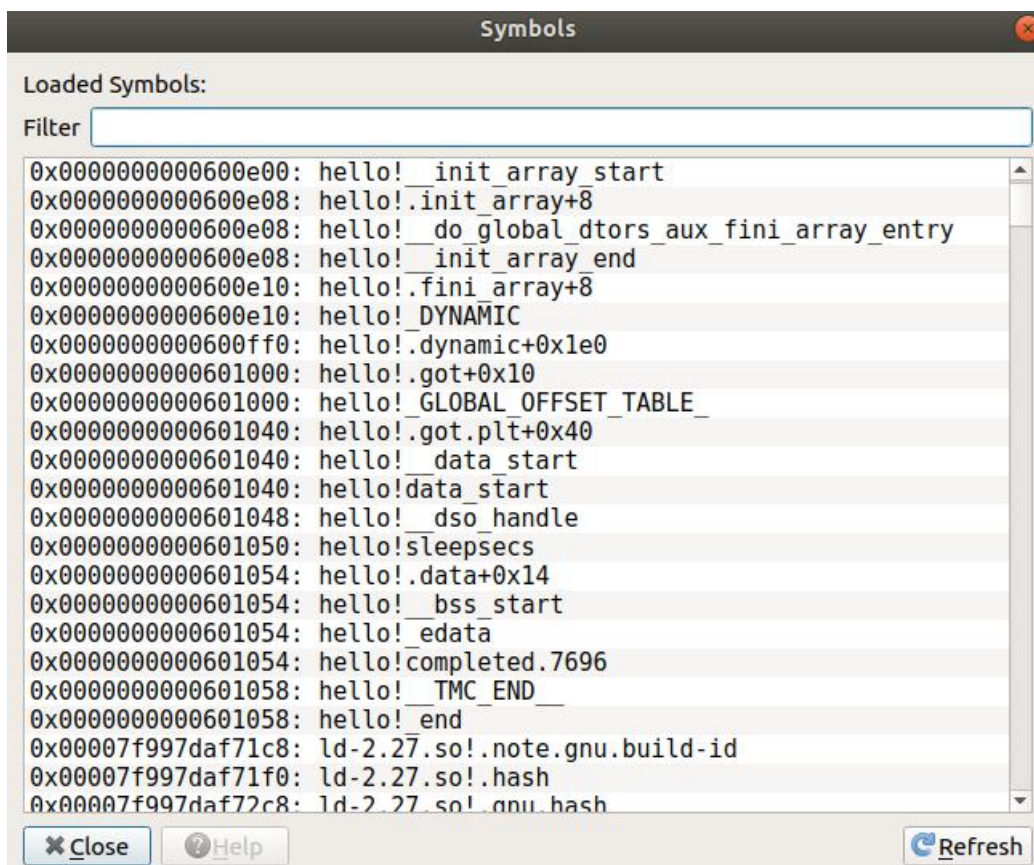


图 5.5- Loaded symbols

## 5.8 本章小结

链接将 hello.o 文件与动态连接库和静态连接库链接生成可执行程序,分析链接前后虚拟地址和物理地址之间的映射对。

(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：

进程指一个执行中程序的实例。更确切说，进程是具有独立功能的一个程序关于某个数据集合的一次运行活动，因而进程具有动态含义。同一个程序处理不同的数据就是不同的进程。计算机系统中的任务通常就是指进程。

进程为用户提供了一个假象：我们的程序好像是系统中唯一运行的程序一样，独立的使用处理器和内存，处理器好像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系统唯一的对象。

#### 进程的作用：

计算机处理的所有用户的任务由进程来完成，进程能有效提高 CPU 的执行效率，减少因为程序等待带来的 CPU 空转以及其它计算机软硬件资源浪费

### 6.2 简述壳 Shell-bash 的作用与处理流程

shell 是一个交互型应用级程序，代表用户运行其他程序，shell 提供了一个界面，用户通过这个界面访问操作系统内核的服务。

#### 处理流程：

- 1、读取用户由键盘输入的命令行。
- 2、分析命令，以命令名作为文件名，并将其它参数改造为系统调用 `execve()` 内部处理所要求的形式。
- 3、终端进程调用 `fork()` 建立一个子进程。
- 4、终端进程本身调用 `wait()` 来等待子进程完成（如果是后台命令，则不等待）。当子进程运行时调用 `execve()`，子进程根据文件名到目录中查找有关文件，调入内存，执行这个程序。
- 5、如果命令末尾有 `&`，则终端进程不用执行系统调用 `wait4()`，立即发提示符，让用户输入下一条命令；否则终端进程会一直等待，当子进程完成工作后，向父进程报告，此时中断进程醒来，作必要的判别工作后，终端发出命令提示符，重复上述处理过程。

### 6.3 Hello 的 fork 进程创建过程

在执行 hello 程序的过程中，需要在 shell 命令行中键入 `./hello 1170301027 fengshuai`，运行的终端程序会对输入的命令进行解析，由于 `./hello` 不是内置命令（`quit,fg,bg,stop` 等等），所以 shell 会调用 `fork` 函数为程序创建一个新的运行的子进程，这个子进程几乎但不完全与父进程相同，但是不可否认的子进程得到哦与父进程用户级虚拟地址空间（但是独立）的一个副本，包括代码和数据段，堆、共享库以及用户栈。子进程还同时获得了与父进程任何打开文件描述符相同的副本，这就意味着，当父进程调用 `fork` 时，子进程可以读写父进程中打开的各种文件。子进程和其父进程之间最大的区别就是他们有不同的 PID。

在 hello 中，fork 创建过程简化为

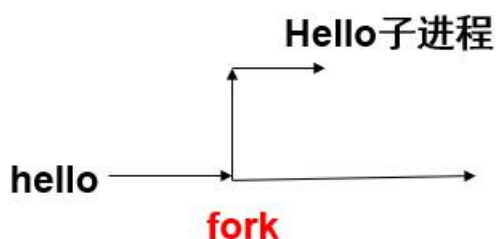


图 6.1-Hello.1

### 6.4 Hello 的 execve 过程

Hello 中，在程序调用 `fork` 函数创建一个子进程之后，当前子进程调用 `execve` 函数（传入命令行参数）在当前进程的上下文中加载并运行一个新程序，也就是 `hello` 程序，此时 shell 会调用某个驻留在存储器中成为加载器（loader）的操作系统代码来运行它。加载器会将可执行文件中的代码和数据从磁盘复制到内存中，然后通过跳转到程序的第一条指令的入口点来运行该程序。

在当前子进程的加载中，加载器运行时，会在头部表的引导下，将可执行文件的片复制到代码段和数据段，接下来，加载器跳转到程序的入口点也就是 `_start`（在 `ctrl.o` 中定义）函数的地址处，`_start` 函数会调用系统启动函数，`__libc_start_main` 函数，该函数初始化执行环境，调用用户层的 `main` 函数，处理 `main` 函数的返回值，并在必要的时候将控制返回给内核。至此 `execve` 过程执行完毕

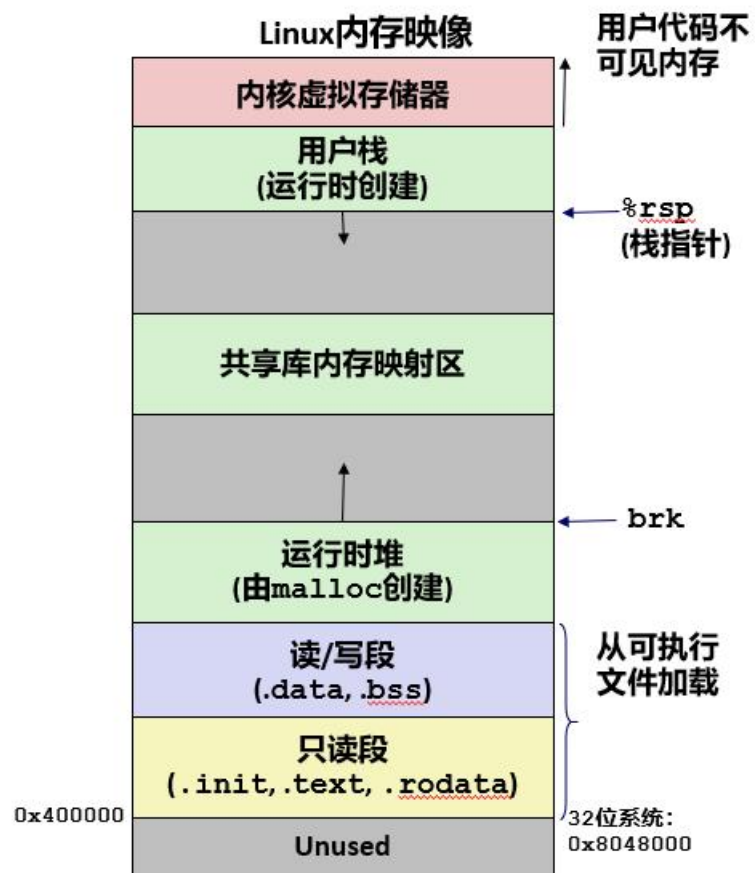


图 6.2-Linux x86-64 运行时的内存映像

Hello 的 exeve 执行过程

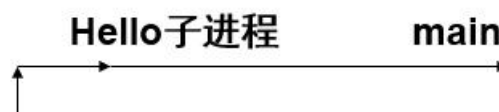


图 6.3-hello.2

## 6.5 Hello 的进程执行

(以下格式自行编排, 编辑时删除)

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。

概念定义:



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的状态，它由通用寄存器、浮点寄存器、程序计数器、用户栈、状态寄存器、内核栈和各种内核数据结构等对象的值构成。

上下文切换：由内核中的调度器完成的，当内核调度新的进程运行后，它就会抢占当前进程，（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

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，进程是轮流使用处理器的，在同一个处理器核心中，每个进程执行它的流的一部分后被抢占（暂时挂起），然后轮到其他进程。

时间片：一个进程执行它的控制流的一部分的每一时间段叫做时间片。

用户模式和内核模式：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，进程就处于用户模式中，用户模式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，进程处于内核模式，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。

### **Hello 进程执行分析：**

在正确输入命令行执行 `./hello` 程序之后程序会循环十次的显示用户的学号和姓名信息，而此时循环语句中的 `sleep` 函数的调用就满足上下文切换条件。

`Sleep` 函数属于系统调用，它显式的请求让调用进程休眠，一般而言，计时系统调用没有阻塞，内核也可以决定执行上下文切换。在调用 `sleep` 之前，如果 `hello` 程序不被抢占资源，那么将会顺序执行当前进程，（看似当前进程独立的占用所有资源），一旦发生抢占现象，就会进入上下文切换。

`Hello` 初始运行在用户模式下，在调用 `sleep` 函数之后，陷入内核模式，此时其他的进程会来抢占，完成一次上下文切换；当 `sleep` 函数调用完成等待 `sleepsecs` 秒或者因为某个中断信号（例如 `pause`）而过早的返回之后，内核处理该信号并主动释放当前正在运行的进程，切换回 `hello` 进程继续执行，完成又一次的上下文切换。这在用户的角度来看不过是程序休眠了 2 秒后又继续执行那么简单。

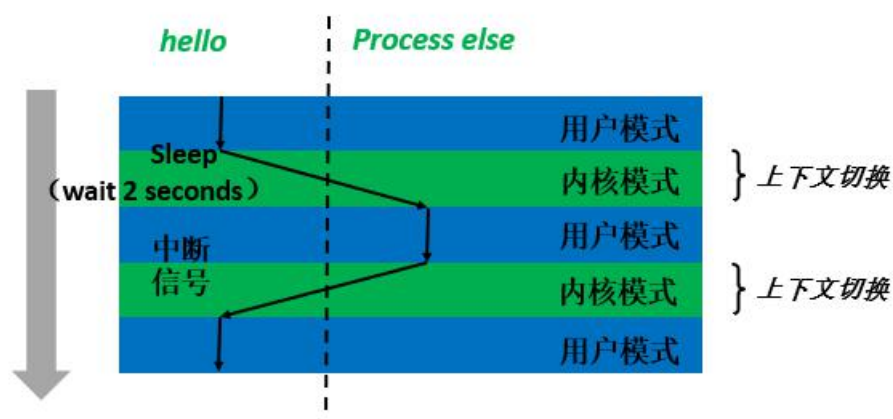


图 6.4-sleep 上下文切换

进程在调用 `getchar` 函数的时候，实际上意味着调度 `stdin` 标准输入流中的 `read` 系统调用，同样会执行上下文切换。

Hello 进程在用户模式下执行到 `getchar` 函数的时候，它通过执行系统调用 `read` 而陷入到内核模式，其他进程抢占。内核中的陷阱处理程序请求来自磁盘控制器中的 DMA 传输，并且安排在磁盘控制器完成从磁盘大内存的数据传输。此间进程在完成从用户输入数据之后，直到磁盘发出一个中断信号，表示数据已经从磁盘传入到内存中，内核判定其他进程已经运行了足够的时间，就执行又一个上下文切换，从其他进程切换回 `hello` 进程继续运行。

## 6.6 hello 的异常与信号处理

(以下格式自行编排，编辑时删除)

`hello` 执行过程中会出现哪几类异常，会产生哪些信号，又怎么处理的。

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

```
fs1170301027@ubuntu:~/hello$ ./hello 1170301027 fengshuai
Hello 1170301027 fengshuai
Hello 1170301027 fengshuai
asdHello 1170301027 fengshuai

Hello 1170301027 fengshuai

Hello 1170301027 fengshuai
Hello 1170301027 fengshuai
Hello 1170301027 fengshuai
Hello 1170301027 fengshuai
Hello 1170301027 fengshuai
Hello 1170301027 fengshuai
fs1170301027@ubuntu:~/hello$
fs1170301027@ubuntu:~/hello$
```

图 6.5.1-hello 执行之不挺乱摁加回车篇

```
fs1170301027@ubuntu:~/hello$ ./hello 1170301027 fengshuai
Hello 1170301027 fengshuai
^C
fs1170301027@ubuntu:~/hello$ ps
  PID TTY          TIME CMD
  6321 pts/0        00:00:00 bash
  6358 pts/0        00:00:00 vim
  7865 pts/0        00:00:00 ps
```

图 6.5.2-hello 执行之 Ctrl+c 键加回车篇

```
fs1170301027@ubuntu:~/hello$ ./hello 1170301027 fengshuai
Hello 1170301027 fengshuai
Hello 1170301027 fengshuai
Hello 1170301027 fengshuai
^Z
[2]+  已停止                  ./hello 1170301027 fengshuai
fs1170301027@ubuntu:~/hello$ ps
  PID TTY          TIME CMD
  6321 pts/0        00:00:00 bash
  6358 pts/0        00:00:00 vim
  7902 pts/0        00:00:00 hello
  7904 pts/0        00:00:00 ps
fs1170301027@ubuntu:~/hello$ fg
./hello 1170301027 fengshuai
Hello 1170301027 fengshuai
Hello 1170301027 fengshuai
Hello 1170301027 fengshuai
Hello 1170301027 fengshuai
Hello 1170301027 fengshuai
Hello 1170301027 fengshuai
Hello 1170301027 fengshuai
kill
fs1170301027@ubuntu:~/hello$ ps
  PID TTY          TIME CMD
  6321 pts/0        00:00:00 bash
  6358 pts/0        00:00:00 vim
  7936 pts/0        00:00:00 ps
```

图 6.5.3-hello 执行之 Ctrl+z 键加回车篇

图 6.5.1 显示，在程序运行过程中，乱恩以及 enter 键并不能影响程序的运行，可见，乱摁的结果无非是将屏幕中输入的字符写到缓冲区中，在程序执行到 `getchar` 时，每读出一个 enter 带来的换行符时判定为一次输入结束，命令无效则并不处理，然后接着继续处理剩余字符串直到结束。

图 6.5.2 显示，在程序运行过程中按下 Ctrl+c 键，父进程会受到一个 SIGINT 的信号，该信号处理结果是结束并回收子进程，故而 ps 查看进程的时候发现 hello 进程已经终止。

图 6.5.3 显示，在程序执行过程中按下 Ctrl+z 键，父进程受到一个 SIGTSTP 信号，该信号的默认行为是打印屏幕回显，挂起当前进程。现在再在 shell 下输出 ps 指令，会发现 hello 进程仍然存在，并没有被回收，此时输入 fg 指令可以将其重新运行到前台，继续执行，知道运行结束输入字符串，进程终止被父进程回收，此时输入 ps 指令发现 hello 进程已经终止。

## 6.7 本章小结

hello 的实际执行过程中，shell 为其 fork 产生子进程，为其 exeve 执行程序，为其分配时间片，为其处理异常。在 shell 的支持下，hello 可以有条不紊地执行一定的操作。

**(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(LogicalAddress): 逻辑地址指的是机器语言指令中, 用来指定一个操作数或者是一条指令的地址, 其实是指由程序产生的与段相关的偏移地址部分(段内偏移量)。映射到 hello.o 里面的相对偏移地址。

线性地址(address space): 线性地址是逻辑地址到物理地址变换之间的中间层。程序代码会产生逻辑地址, 或者说是段中的偏移地址, 加上相应段的基地址就生成了一个线性地址。如果启用了分页机制, 那么线性地址可以再经变换以产生一个物理地址。若没有启用分页机制, 那么线性地址直接就是物理地址。此间映射到 hello 里面的虚拟内存地址。

虚拟地址(Virtual Address, VA) : CPU 通过生成一个虚拟地址。映射到 hello 里面的虚拟内存地址。

物理地址 (Physical Address, PA) : 物理地址用于内存芯片级的单元寻址, 与处理器和 CPU 连接的地址总线相对应。计算机系统的主存被组织成一个由 M 个连续的字节大小的单元组成的数组。每字节都有一个唯一的物理地址。映射到 hello 在运行时虚拟内存地址对应的物理地址。

### 7.2 Intel 逻辑地址到线性地址的变换-段式管理

段式管理: 逻辑地址->线性地址==虚拟地址

- 1、逻辑地址=段选择符+偏移量
- 2、每个段选择符大小为 16 位, 段描述符为 8 字节(注意单位)。
- 3、GDT 为全局描述符表, LDT 为局部描述符表。
- 4、段描述符存放在描述符表中, 也就是 GDT 或 LDT 中。
- 5、段首地址存放在段描述符中。

每个段的首地址都存放在自己的段描述符中, 而所有的段描述符都存放在一

个描述符表中（描述符表分为全局描述符表 GDT 和局部描述符表 LDT）。而要想找到某个段的描述符必须通过段选择符才能找到。

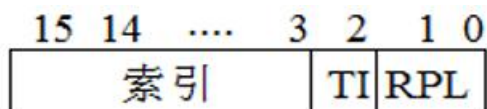


图 7.1-段选择符格式

由图 7.1 可以看出，段选择符由三部分组成，从左到右依次是 index【索引】，TI，RPL。

Index 处，我们可以将描述符表看成是一个数组，每个元素都存放一个段描述符，那么 index 就表示数组下标，亦即某个段描述符在数组中的索引。

再者，当 TI 为 0 时，表示段描述符在 GDT 中，当 TI 为 1 的时候，表示段描述符在 LDT 中。

RPL 代表请求特权级，RPL=00，为第 0 级，位于最高级的内核态，RPL=11，为第 3 级，位于最低级的用户态，第 0 级高于第 3 级。

现在假设我们有一个段的段选择符，他的 TI 是 0，Index 是 8，那么我们可以知道这个段的段描述符实在 GDT 数组中索引为 8 的位置。从而由我们知道的 GDT 的起始地址，每个段描述符的大小，就可以精确地找到我们想要的段描述符，从而获取某个段的首地址，然后再将从段描述符中获取到的段首地址与逻辑地址的偏移量相加就得到了线性地址。

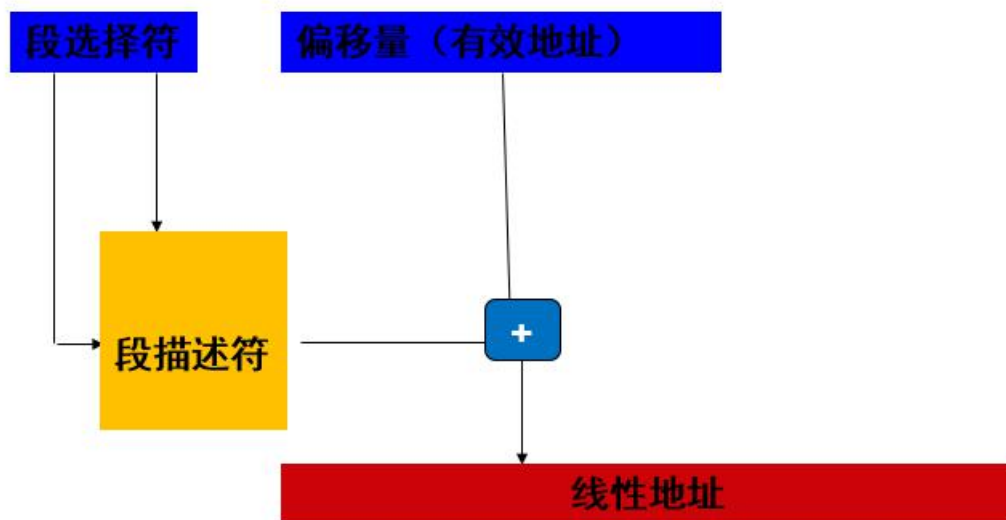


图 7.2-逻辑地址转换简化版

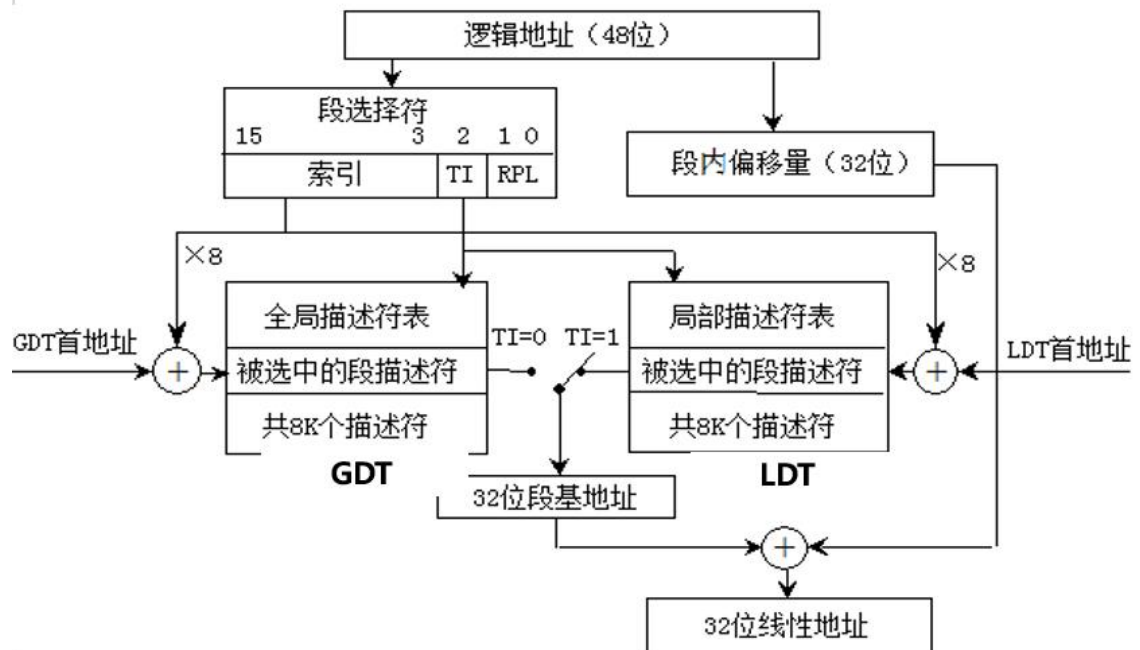


图 7.3-逻辑地址转化真实版

### 7.3 Hello 的线性地址到物理地址的变换-页式管理

页式管理： 虚拟地址->物理地址

分页机制是对虚拟地址内存空间进行分页。

Linux 系统有自己的虚拟内存系统，如图 7.4，Linux 将虚拟内存组织成一些段的集合，段之外的虚拟内存不存在。内核为 hello 进程维护一个段的任务结构即图中的 `task_struct`，其中条目 `mm` 指向一个 `mm_struct`，它描述了虚拟内存的当前状态，`pgd` 指向第一级页表的基地址（结合一个进程一串页表），`mmap` 指向一个 `vm_area_struct` 的链表，一个链表条目对应一个段，所以链表相连指出了 hello 进程虚拟内存中的所有段。



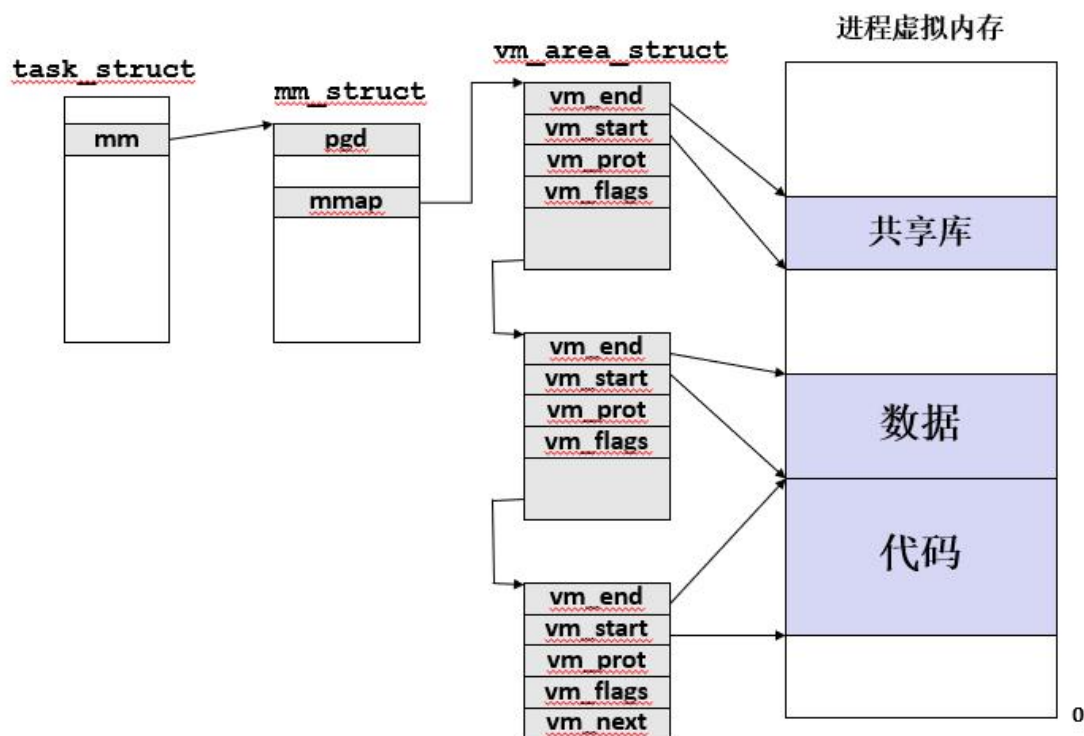


图 7.4-linux 下的虚拟内存系统

概念上而言，虚拟内存被组织为一个由存放在磁盘上  $N$  个连续的字节大小的单元组成的数组，每字节都有唯一一个虚拟地址作为到数组的索引。磁盘上数组的内容被缓存在主存中。和存储器结构中其他的缓存一样，磁盘（较低层）的数据被分割成块，此间（VM）虚拟内存系统将虚拟内存分割成拟页 VP（Virtual Page）大小固定的块来处理这个问题，linux 下通常每个虚拟页的大小为 4KB，与之相类似，物理内存也被分割成物理页 PP（Physical Page），大小和虚拟页大小一致。

此间虚拟内存系统中的 MMU 内存管理单元对地址的翻译，就形象为物理内存中叫做页表的数据结构从虚拟页映射到物理页的过程。图 7.5 详细的展示了地址翻译的全过程。



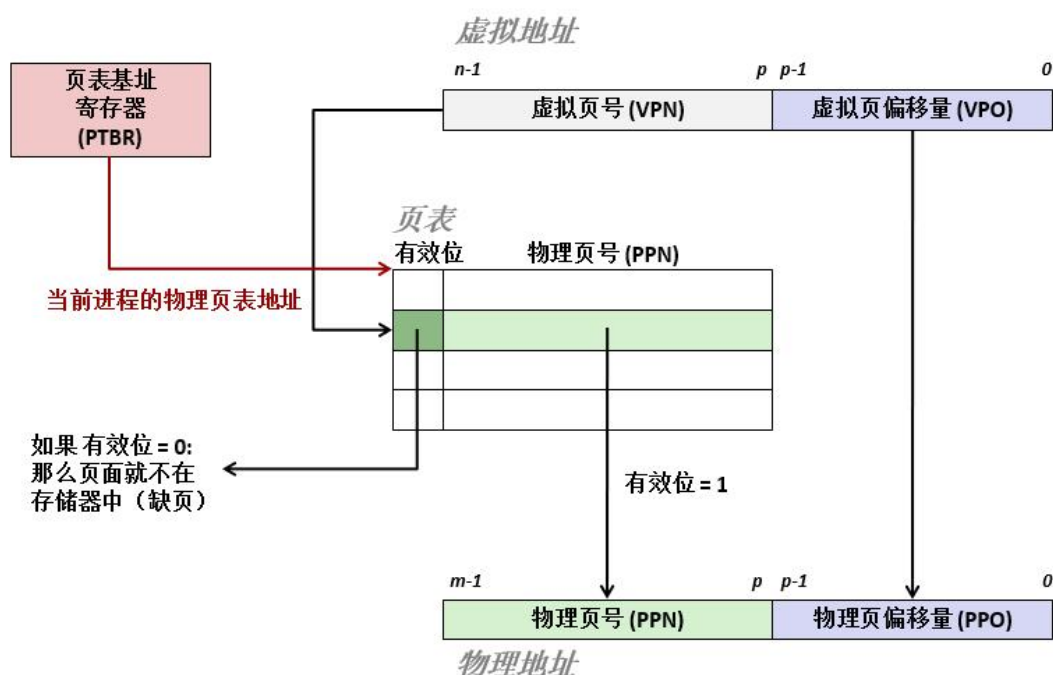


图 7.5-地址翻译

详解:

页表基址寄存器指向当前页表,  $n$  位的虚拟地址包含虚拟页号和虚拟页偏移量两部分, 同样物理地址也由物理页号和物理页偏移量组成。MMU 通过 VPN 来选择适当的 PTE, 由此, 将索引到的页表条目中的 PPN 和 VPO 串联起来就是虚拟地址

步骤如下:

- 1) 处理器生成一个虚拟地址, 并将其传送给 MMU
- 2-3) MMU 使用内存中的页表生成 PTE 地址
- 4) MMU 将物理地址传送给高速缓存/主存
- 5) 高速缓存/主存返回所请求的数据字给处理器

## 7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

**TLB:** 翻译后备缓冲器, TLB 是 MMU 中一个小的、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, 通常有高度的相连度, TLB 实现虚拟页码向物理页码的映射, 对于页码数很少的页表可以完全包含在 TLB 中。用于族选择和行匹配的索引和标记字段是从虚拟地址中的虚拟页号中提取出来的, 详情如图 7.6。

关键在于, 所有的地址翻译的步骤都是在 MMU 中执行的, 因此非常快

翻译步骤:

- 1) 处理器 CPU 生成一个虚拟地址, 并将其传送给 MMU
- 2-3) MMU 从 TLB 中取出相应的 PTE
- 4) MMU 将这个虚拟地址翻译成物理地址传送给高速缓存/主存
- 5) 高速缓存/主存返回所请求的数据字给处理器 CPU

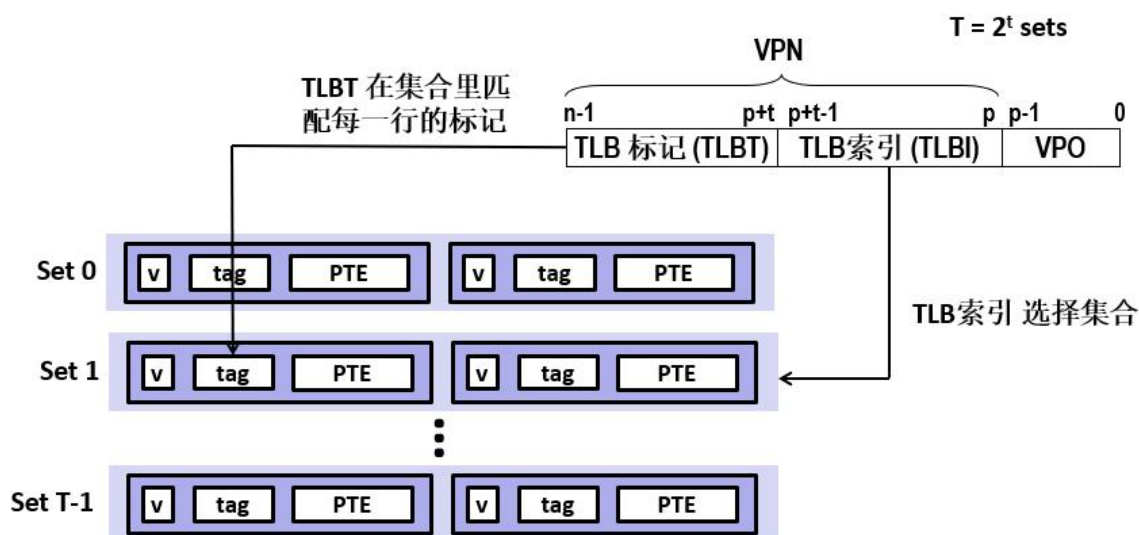


图 7.6-TLB 支持下的地址翻译

多级页表:

以二级页表为例: (如图 7.7.1)

一级页表: 每个 PTE 指向一个页表 (常驻内存)

二级页表: 每个 PTE 指向一页(paged in and out like any other data 页面可以调入或调出页表)

二级页表中的每个 PTE 都负责一个 4KB 的虚拟内存页面, 就像我们看到的只有一级的页表一样, 注意, 使用四字节的 PTE, 每一个一级和二级页表都是 4KB 的字节, 这刚好和一个页面的大小是一样的。

这种方法从两个方面减少了内存要求。第一, 如果一个一级页表是空的, 那么其对应的二级页表将不存在。这代表着一种巨大的潜在节约, 移位对于一个典型的程序, 4 GB 的虚拟地址空间的大部分都将会是未分配的。第二, 只有一级页表才需要总是在主存里, 虚拟系统可以再需要时创建, 页面调入或者调出二级页表这就减少了主存的压力, 只有经常使用的二级页表才会在主存里。

多级页表同 (图 7.7.2)

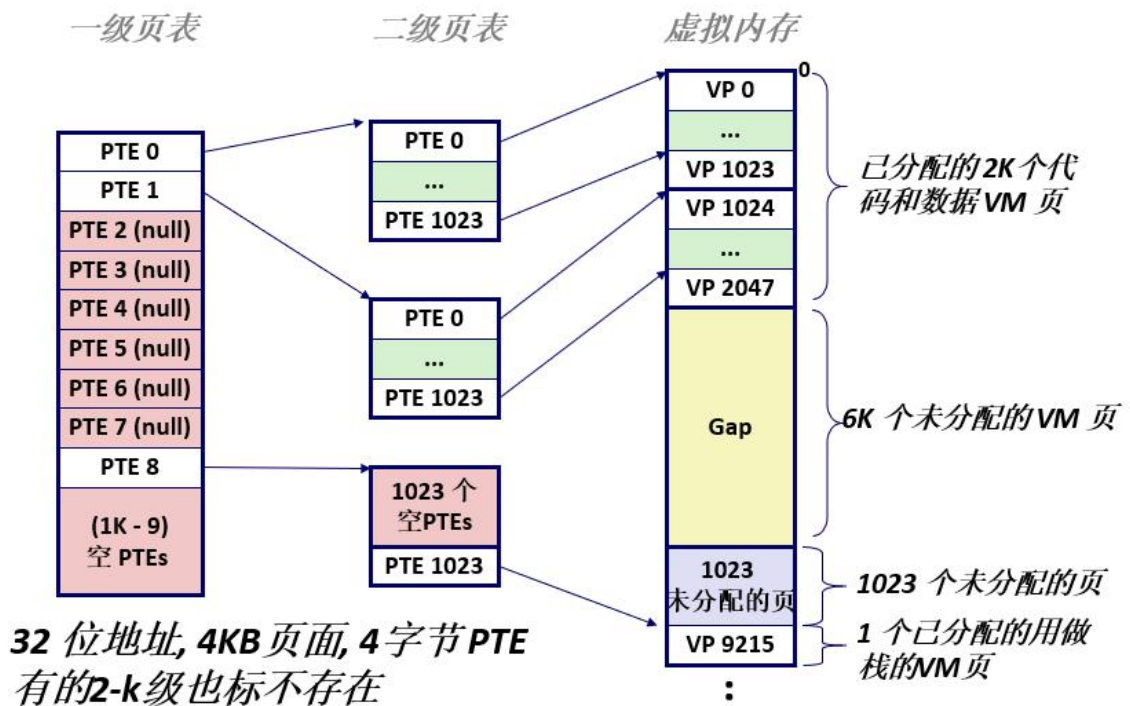


图 7.7.1-二级页表示例

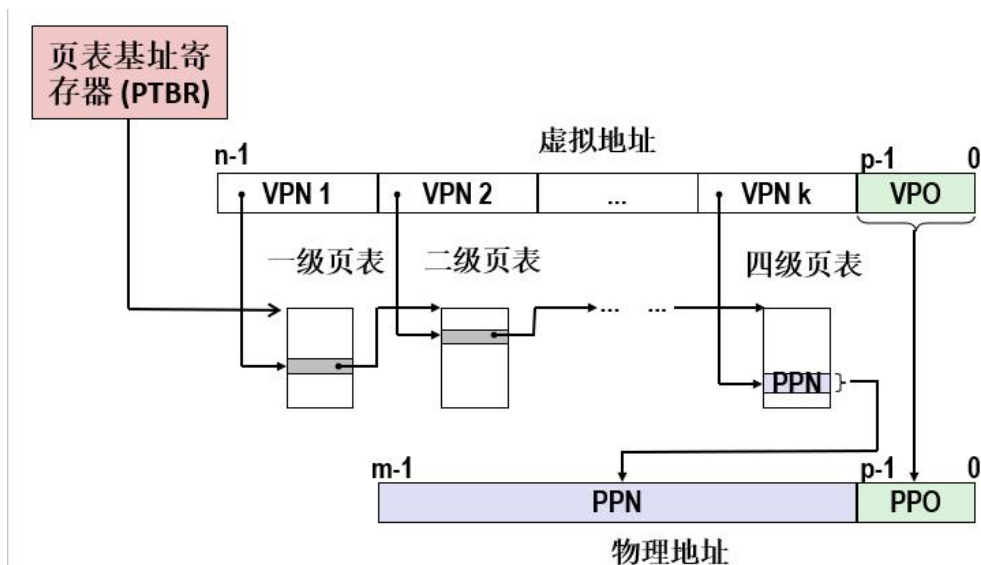


图 7.7.2-四级页表下的翻译

如下图(图 7.8)是 Core i7MMU 如何使用四级页表来讲虚拟地址翻译成物理地址的全过程。36 位 VPN 被划分成了四个 9 位的片，每个片被用作到一个也表的偏移量。CR3 寄存器包含 L1 也表的物理地址。VPN1 提供一个到 L1PTE (页表条目)

的偏移量，这个 PTE 包含 L2 也表的基地址。VPN2 提供一个到 L2PTE 的偏移量，以此类推...

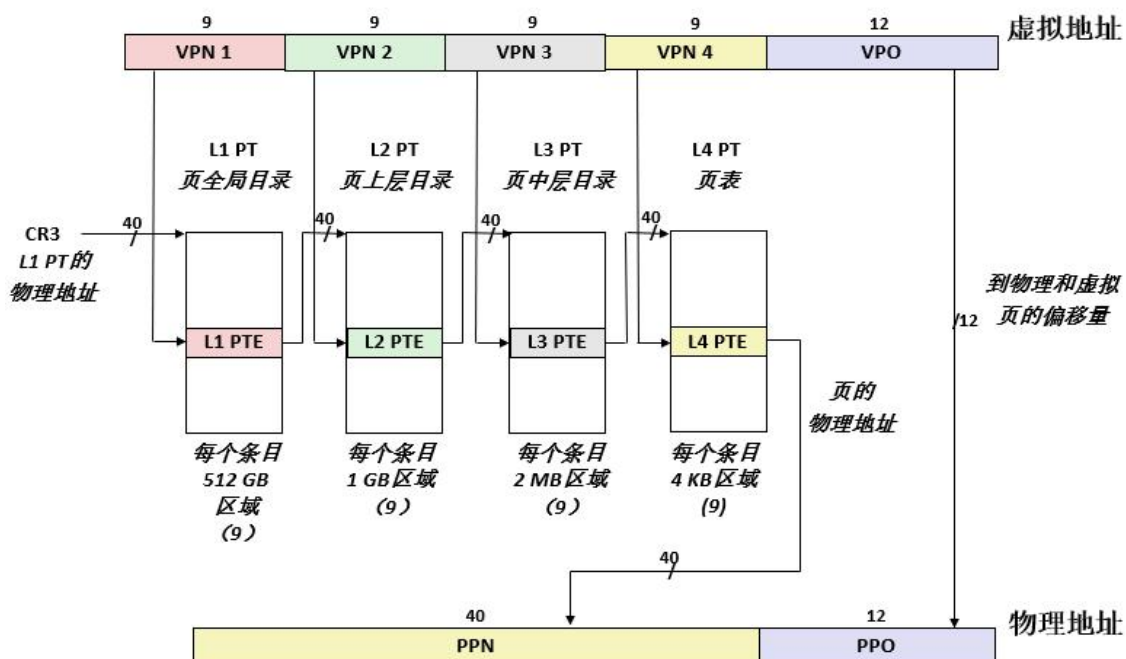
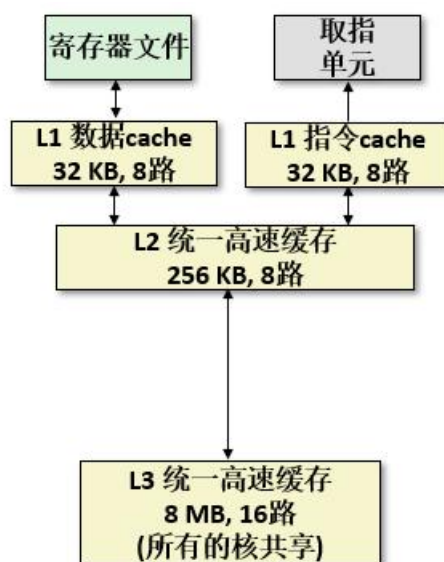


图 7.8-Core i7 下的四级页表翻译

## 7.5 三级 Cache 支持下的物理内存访问



附图-三级 cache

运行的 Linux 的 Core i7 内存系统中对于处理器的封装很有讲究（此间不做过多介绍），我们可知的是，Linux 使用的是 4KB 的页，并且通过 TLB 虚拟寻址能

得到我们想要的物理地址，然而有了这个物理地址之后，处理器可以通过对物理地址的处理而得到相应的在依赖物理地址寻址的 L1L2L3 三级高速缓存快速判断命中与否。

具体实现如下图（图 7.9）

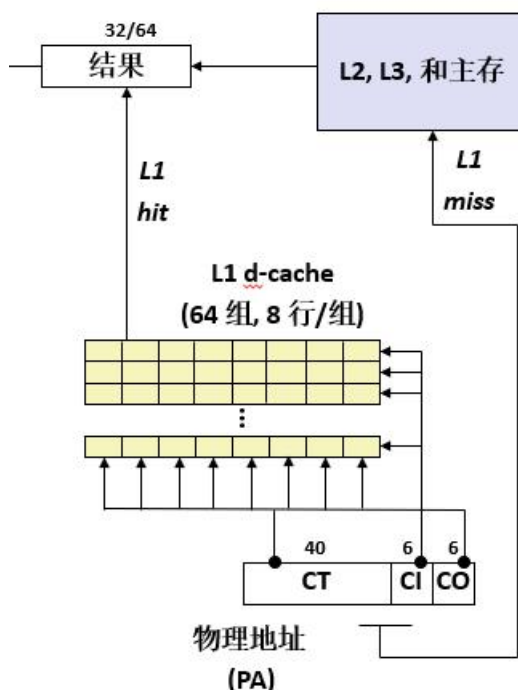


图 7.9-高速缓存对物理地址的处理阶段

由上图可知，对 L1 高速缓存（注意不是页表！！）来说，在通过页表操作获取到从虚拟地址（VA）转化来的物理地址（PA）之后，使用 CI（64 组 6 位组索引位）进行组索引，每组 8 路，对 8 路的块分别匹配 CT（前 40 位）如果匹配成功且块的 valid 标志位为 1，则造成一次 L1 缓存命中（hit），然后根据数据偏移量 CO（后六位）取出数据返回。

如果当前没有匹配成功但是此时 valid 标志位已经被设置为 1（块不匹配但是已分配），那么此时造成一次 L1 缓存不命中（miss），则此时需要向下一级 cache 中查询数据（优先级依次为 L2->L3->主存）。直到查询到数据后，判断当前组内是否有空闲块，如若有则直接写入；否则 L1cache 将会采用最近最少使用策略对组中的某个确定块进行驱逐（eviction）然后再写入。

事实上实际系统在运行的时候当在需要翻译虚拟地址的时，CPU 就已经将 VPN 发送到了高速缓存中。也就是说，理解翻译过程之后我们知道由于物理地址的 PPO 就是就是虚拟地址的 VPO，所以，在 MMU 忙着向 TLB 请求一个 PTE 页表条目的时候，L1 高速缓存实际上已经开始在分离组索引并查找相应的组了。这

极大地情况上加快了翻译效率。

## 7.6 hello 进程 fork 时的内存映射

Linux shell 下 fork 函数如何为每个新进程提供私有的虚拟地址空间.

为新进程创建虚拟内存

创建当前进程的 `mm_struct`, `vm_area_struct` 和页表的原样副本.

两个进程中的每个页面都标记为只读

两个进程中的每个区域结构 (`vm_area_struct`) 都标记为私有的写时复制 (COW)

在新进程中返回时, 新进程拥有与调用 `fork` 进程相同的虚拟内存

随后的写操作通过写时复制机制创建新页面

## 7.7 hello 进程 execve 时的内存映射

`execve` 执行步骤:

- (1) 删除已存在的用户区域
- (2) 创建新的区域结构 (私有的、写时复制)
  - 代码和初始化数据映射到 `.text` 和 `.data` 区 (目标文件提供)
  - `.bss` 和栈堆映射到匿名文件, 栈堆的初始长度 0
- (3) 共享对象由动态链接映射到本进程共享区域
- (4) 设置 PC, 指向代码区域的入口点。Linux 根据需要换入代码和数据页面

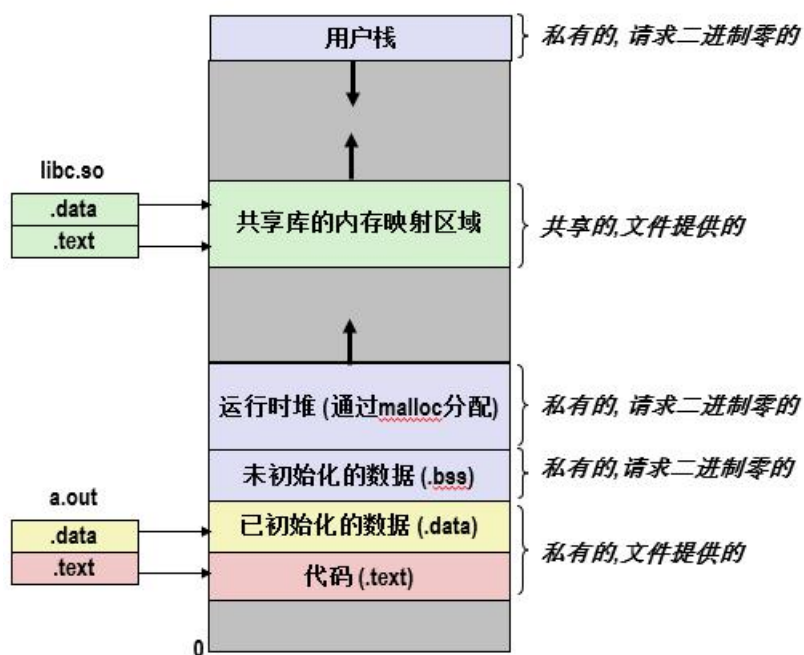


图 7.10-linux 下 exeve 时的内存映射

## 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。缺页异常调用内核中的缺页异常处理程序，该程序会选择一个牺牲页，如果该被牺牲页已经被修改过则会被内核直接复制回磁盘。总之内核会总是修改该牺牲页的页表条目（PTE），反应出该牺牲页已经不再缓存在主存中的事实。

随后，当缺页异常处理程序返回时，它会重新启动导致缺页的指令，该指令会把导致缺页的虚拟地址重新发送到地址翻译硬件进行地址翻译，此时将不再发生异常。（处理过程见于图 7.11）





图 7.11-缺页异常处理

## 7.9 动态存储分配管理

*Printf 会调用 malloc，请简述动态内存管理的基本方法与策略。*

动态内存分配器维护着一个进程的虚拟内存区域，称为堆（图 7.12），堆是一个请求二进制零的区域，它紧接在未初始化的数据区域后面开始，并向上生长（更高的地址）。对于每一个进程，内核维护着一个变量 `brk`，用它来指向堆的顶部（堆最小的地址处）。

分配器将堆视为一组不同大小的块(blocks)的集合来维护，每个块要么是已分配的，要么是空闲的。

分配器的类型：

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` 和 `free`

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 (garbage collection)

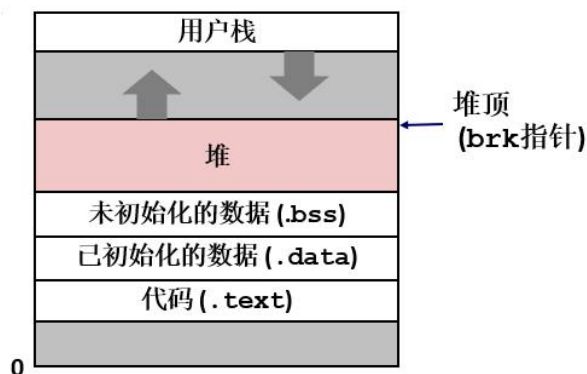


图 7.12-简化版虚拟内存空间

程序使用动态内存分配的最主要原因是经常直到程序运行时，才知道某些数据结构的大小。

在本小节中，介绍两种动态存储分配管理方法:1) 隐式空闲链表法 2) 显示空闲链表法;

(1) 隐式空闲链表(边界标记): 通过头部中的大小字段隐含的连接空闲块

1. 堆及堆中内存块的组织结构:

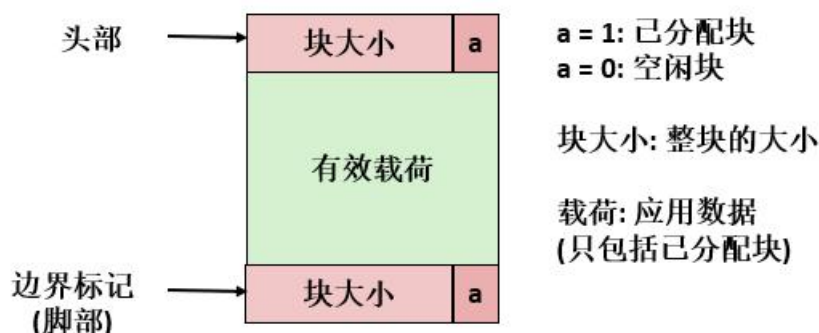


图 7.13-隐式空闲链表中堆中块组织结构

2. 适配方法:

首次适配 (First fit):

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

在靠近链表起始处留下小空闲块的 “碎片” ;

下一次适配 (Next fit):

和首次适配相似，只是从链表中上一次查询结束的地方开始

比首次适应更快: 避免重复扫描那些无用块

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;

最佳适配 (Best fit):

查询链表，选择一个 最好的 空闲块: 适配，剩余最少空闲空间

保证碎片最小——提高内存利用率

通常运行速度会慢于首次适配;

3. 分割空闲块:

在分配块小于空闲块的时候我们可以把空闲块分割成两部分;

4. 释放已分配块:

在程序中不没有用的块或者已经用完了的块需要释放回收;

5. 合并相邻的空闲块:

立即合并 (Immediate coalescing): 每次释放都合并

延迟合并 (Deferred coalescing): 尝试通过延迟合并, 即直到需要才合并来提高释放的性能. 例如: 为 malloc 扫描空闲链表时可以合并; 外部碎片达到阈值时可以合并

- (2) 显示空闲链表: 在空闲块中用指针连接空闲块  
堆中块的结构:

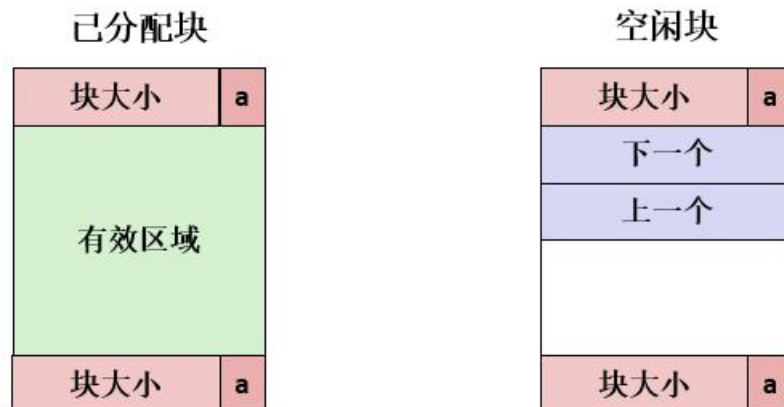


图 7.14-显示空闲链表中堆中块的结构

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, 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。

操作大致上与隐式空闲链表所要完成的操作相同, 但要点是, 我们只保留空闲块链表, 而不是所有块; 再者由于无法确定下一个块的位置以及大小 (可以在任何地方), 故而我们需要存储空闲块前后指针, 而不仅仅是大小; 同时需要合并边界标记;

插入原则: 针对已释放的块, 我们有

LIFO (last-in-first-out) policy, 后进先出法:

将新释放的块放置在链表开始处;

地址顺序法:

按照地址顺序维护链表:

$\text{Addr}(\text{祖先}) < \text{Addr}(\text{当前回收块}) < \text{Addr}(\text{后继})$

## 7.10 本章小结

本章前半章主要讲述了 linux 下的存储管理, 虚拟地址到物理地址的映射过程,

翻译过程，以及系统是通过什么怎么样辅助地址翻译的过程的（TLB，四级页表，三级 cache），后半章主要讲述了缺页故障的处理过程以及动态内存分配器工作原理原因及必要性等等，理解这些对于我们对系统存储方面的理解有很大的帮助。

**（第 7 章 2 分）**

## 第 8 章 hello 的 IO 管理

### 8.1 Linux 的 IO 设备管理方法

(以下格式自行编排, 编辑时删除)

设备的模型化:

一个 linux 的文件就是一个  $m$  字节的序列:  $B_0, B_1, \dots, B_{m-1}$

所有的 I/O 设备都被模型化为文件, 甚至于内核也被映射为文件。

设备管理: 所有的输入和输出都被当做对文件的读和写来执行。将设备优雅的映射为文件的方式, 允许 Linux 的内核引出一个简单的低级的应用接口, 成为 Unix I/O 接口, 这使得所有的输入和输出都能够以一种统一并且一直的方式来执行。

### 8.2 简述 Unix IO 接口及其函数

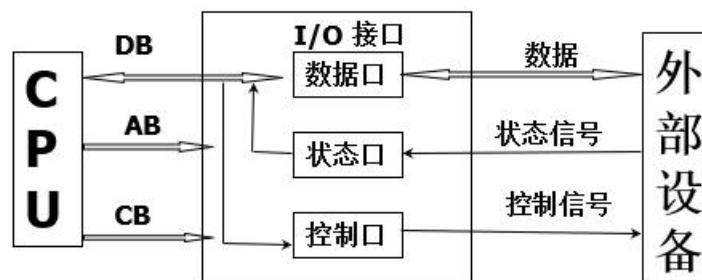


图 8.1-简单的接口框图

Unix IO 接口基本操作:

1. 打开和关闭文件

`open()` and `close()`

2. 读写文件

`read()` and `write()`

3. 改变当前的文件位置 (`seek`)

指示文件要读写位置的偏移量

`lseek()`

函数的具体声明：

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，否则返回值为-1 表示一个错误。`flags` 参数指明了进程访问这个文件的形式。`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（成功为 0 出错为-1）。关闭一个已关闭的描述符会出错！

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表示成功。

4. `ssize_t write(int fd, const void *buf, size_t n)`，`write` 函数从内存位置 `buf` 复制至多 `n` 个字节到描述符为 `fd` 的当前文件位置。返回值-1 表示一个错误，否则返回值表示实际传送的字节数量表示成功。

5. `lseek` 函数，应用程序调用该函数能够显示地修改当前文件的位置。

### 8.3 printf 的实现分析

（以下格式自行编排，编辑时删除）

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

从 `vsprintf` 生成显示信息，到 `write` 系统函数，到陷阱-系统调用 `int 0x80` 或 `syscall`。

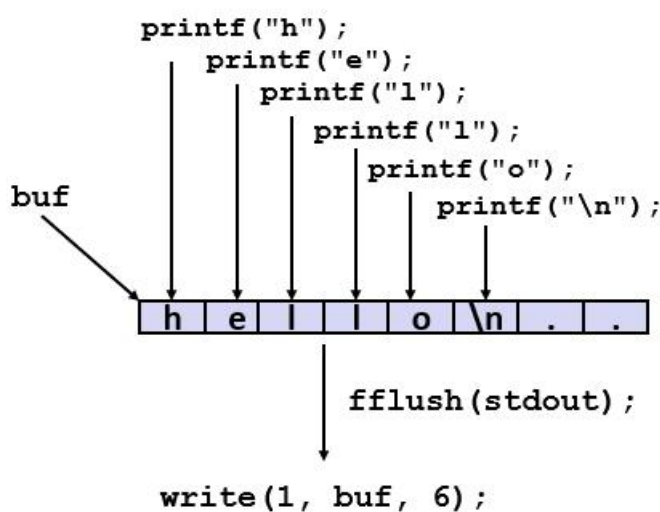


图 8.2-printf 函数的一个简单的实现过程

C 语言标准输入输出库中对于 printf 函数的定义是这样的：

```
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;
}
```

从上述代码中我们可以看到 它声明了一个缓冲区变量，大小是 256，又声明了一个类型为 va\_list（定义为指针型变量）的变量，其中((char\*)&fmt + 4)这部分代表 printf 参数中 “...” 的第一个参数，而参数中的 fmt 正好指向第一个参数。

好了了解了这些之后，我们再看看 vsprintf 的实现：

```
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++) { //从前向后扫描所有字符串中字符
        if (*fmt != '%') { //判断不是 “%” 就跳出继续循环
            *p++ = *fmt;
            continue;
        }
    }
```

fmt++; //判定是 “%”，第一个%后面接的内容很重要，与参数有关例如： %d，那第一个参数就应该是整型变量

```
switch (*fmt) { //分情况判断%后面规定的参数格式
    case 'x':
        itoa(tmp, *((int*)p_next_arg)); //参数写进缓冲区全过程
        strcpy(p, tmp);
        p_next_arg += 4;
    }
```



```

        p += strlen(tmp);
        break;
    case 's':
        break;
    default:
        break;
    }
}

return (p - buf); //循环结束返回需要打印的字符串的长度。
}

```

详见注释说明，已尽可能的详细。

再回到原 `printf`，此时 `i` 已经被设置为需要打印字符串的长度，接下来就是 `write` 的实现了，不用说也知道这句话无非是想告诉 OS，我需要打印出在缓冲区中的 `i` 个字符，下面追踪到系统 `write` 函数的反汇编语言实现：

```

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

```

其实简单来看，不过是放到六十四位系统，通过寄存器传了两个参数然后调用了一下系统函数就结束了，此间不做深究。

最后，纵观全局，`printf` 函数其实并不能确定其参数在什么地方结束，也不知道参数的个数，它只会根据 `format` 中打印格式的数目依次打印堆栈中参数 `format` 后面地址的内容直到结束，这一点其实在我们高级语言设计 C 语言代码实现过程中已经有所体会。

## 8.4 getchar 的实现分析

异步异常-键盘中断的处理：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（发生上下文切换），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。

`getchar` 函数落实到底层调用了系统函数 `read`，通过系统调用 `read` 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` 进行封装，大体逻辑是读取字符串的第一个字符然后返回

## 8.5 本章小结

本章中主要认识了 linux 的 IO 设备管理办法，知道 Unix IO 接口以及函数，具体分析了 printf 函数和 getchar 函数的实现，属于更深层次的代码层面的探究。

(第 8 章 1 分)

## 结论

hello 所经历的过程：

1. C 语言实现--文本编辑器编写完毕保存时对扩展名的修改，诞生源程序
2. 预处理--将 hello.c 源程序所有调用的外部库扩展到该源程序中诞生 hello.i
3. 编译--处理 hello.i 文件编译成为 hello.s 汇编语言文件
4. 汇编--处理 hello.s 文件汇编成为 hello.o 可重定位文件
5. 链接--处理 hello.o 文件链接外部连接库生成 hello 可执行文件
6. 运行--linux 终端 shell 下键入 “./hello 学号 姓名”，运行可执行文件
7. 创建子进程--shell 通过 fork 函数创建子进程
8. 子进程运行程序--shell 调用 execve，execve 调用启动加载器，加映射虚拟内存，虚拟地址映射到物理地址，运行到 main 函数
9. 执行指令--CPU 逐步执行 hello 中机器语言指令
10. 内存访问--访问内存空间
11. 信号处理--遇到 shell 中的个别信号进入信号处理程序
12. 回收子进程--程序执行完毕后交由父进程回收子进程，结束以及执行

高级语言编写的代码不过十几行，但是实际实现它的过程中并不容易，可能在我们最初的认识中只不过是几个按键的操作在计算机内部却掀起了轩然大波。

(结论 0 分，缺少 -1 分，根据内容酌情加分)

## 附件

，附件（中间产物）

hello.c	原始 c 程序（源程序）
hello.i	预处理操作后生成的文本文件
hello.s	编译之后生成的汇编语言文件
hello.o	汇编之后生成的可重定位文件
hello	链接之后生成的可执行程序
hello.txt	可执行文件 hello 的反汇编语言代码
helloo.txt	可重定位文件 hello.o 的反汇编语言代码
hello.elf	可执行文件 hello 的 ELF 文件格式

（附件 0 分，缺失 -1 分）

## 参考文献

- [1] 《深度理解计算机系统》--Randal E.Bryant / David R.O' Hallaron 著，机械工业出版社
- [2] Printf 函数实现的深度剖析--<https://www.cnblogs.com/pianist/p/3315801.html>
- [3] 逻辑地址、线性地址、物理地址和虚拟地址  
--<http://www.cnblogs.com/diyingyun/archive/2012/01/03/2311327.html>
- [4] Linux 系统学习笔记：虚拟存储器  
--<https://blog.csdn.net/yangxuefeng09/article/details/10066403>

(参考文献 0 分，缺失 -1 分)