

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>1170301026</u>
班 級	<u>1703010</u>
學 生	<u>魏佳琦</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

本文主要是通过合理运用这个学期在计算机系统课程上学习的知识,分析研究 hello 程序在 Linux 下的 P2P 和 O2O 过程,通过熟练使用各种工具,学习 Linux 框架下整个程序的生命周期。

**关键词:** 计算机系统; 进程; P2P; O2O……;

# 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	- 4 -
1.3 中间结果	- 4 -
1.4 本章小结	- 5 -
<b>第 2 章 预处理</b>	<b>- 6 -</b>
2.1 预处理的概念与作用	- 6 -
2.2 在 UBUNTU 下预处理的命令	- 6 -
2.3 HELLO 的预处理结果解析	- 6 -
2.4 本章小结	- 8 -
<b>第 3 章 编译</b>	<b>- 9 -</b>
3.1 编译的概念与作用	- 9 -
3.2 在 UBUNTU 下编译的命令	- 9 -
3.3 HELLO 的编译结果解析	- 9 -
3.4 本章小结	- 17 -
<b>第 4 章 汇编</b>	<b>- 18 -</b>
4.1 汇编的概念与作用	- 18 -
4.2 在 UBUNTU 下汇编的命令	- 18 -
4.3 可重定位目标 ELF 格式	- 18 -
4.4 HELLO.O 的结果解析	- 20 -
4.5 本章小结	- 21 -
<b>第 5 章 链接</b>	<b>- 22 -</b>
5.1 链接的概念与作用	- 22 -
5.2 在 UBUNTU 下链接的命令	- 22 -
5.3 可执行目标文件 HELLO 的格式	- 22 -
5.4 HELLO 的虚拟地址空间	- 24 -
5.5 链接的重定位过程分析	- 25 -
5.6 HELLO 的执行流程	错误!未定义书签。
5.7 HELLO 的动态链接分析	- 31 -
5.8 本章小结	- 32 -
<b>第 6 章 HELLO 进程管理</b>	<b>- 33 -</b>
6.1 进程的概念与作用	- 33 -
6.2 简述壳 SHELL-BASH 的作用与处理流程	- 33 -
6.3 HELLO 的 FORK 进程创建过程	- 33 -

---

6.4 HELLO 的 EXECVE 过程 .....	- 34 -
6.5 HELLO 的进程执行 .....	- 35 -
6.6 HELLO 的异常与信号处理 .....	- 36 -
6.7 本章小结 .....	- 39 -
<b>第 7 章 HELLO 的存储管理 .....</b>	<b>- 40 -</b>
7.1 HELLO 的存储器地址空间 .....	- 40 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理 .....	- 40 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理 .....	- 40 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换 .....	- 43 -
7.5 三级 CACHE 支持下的物理内存访问 .....	- 44 -
7.6 HELLO 进程 FORK 时的内存映射 .....	- 46 -
7.7 HELLO 进程 EXECVE 时的内存映射 .....	- 46 -
7.8 缺页故障与缺页中断处理 .....	- 48 -
7.9 动态存储分配管理 .....	- 48 -
7.10 本章小结 .....	- 49 -
<b>第 8 章 HELLO 的 IO 管理 .....</b>	<b>- 50 -</b>
8.1 LINUX 的 IO 设备管理方法 .....	- 50 -
8.2 简述 UNIX IO 接口及其函数 .....	- 50 -
8.3 PRINTF 的实现分析 .....	- 50 -
8.4 GETCHAR 的实现分析 .....	- 53 -
8.5 本章小结 .....	- 53 -
<b>结论 .....</b>	<b>- 53 -</b>
<b>附件 .....</b>	<b>- 55 -</b>
<b>参考文献 .....</b>	<b>- 56 -</b>

## 第 1 章 概述

### 1.1 Hello 简介

在 linux 中, `hello.c` 经过 `cpp` 的预处理、`ccl` 的编译、`as` 的汇编、`ld` 的链接最终成为可执行目标程序 `hello`, 在 `shell` 中键入启动命令后, `shell` 为其 `fork`, 产生子进程, 于是 `hello` 便从 Program 变成 Process, 这便是 P2P 的过程。

之后 `shell` 为其 `execve`, 映射虚拟内存, 进入程序入口后程序开始载入物理内存, 然后进入 `main` 函数执行目标代码, 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, 以上全部便是 020 的过程。

### 1.2 环境与工具

硬件环境:

X64 CPU; 2GHz; 8G RAM; 128GHD Disk

软件环境:

Windows10 64 位; Vmware 14.12; Ubuntu 16.04 LTS 64 位

使用工具:

`codeblocks`, `objdump`, `gdb`, `edb`, `hexedit`

### 1.3 中间结果

`hello.i`(`hello.c` 预处理之后的程序文本)

`hello.s`(`hello.i` 编译成汇编语言之后的程序文本)

`hello3.c`(用于测试 `sleepsecs` 的值的程序文本)

`hello3`(`hello3.c` 生成的可执行二进制文件)

`hello.o`(`hello.s` 生成的二进制文件)

`hello_o.s`(`hello.o` 反汇编的结果)

hello(可执行的 hello 二进制文件)

hello.s(可执行文件 hello，直接用 objdump 反编译之后的汇编代码)

hello.elf(可执行文件 hello 的 elf 表)

## 1.4 本章小结

本章简单列举了整个实验过程中的中间文件以及所在的软硬件环境和将要使用的工具。

**(第 1 章 0.5 分)**

## 第 2 章 预处理

### 2.1 预处理的概念与作用

(以下格式自行编排, 编辑时删除)

概念: 在编译之前先对源文件进行处理

作用: 1. 可用来把多个源文件连接成一个源文件进行编译, 结果将生成一个目标文件。

2. 条件编译允许只编译源程序中满足条件的程序段, 使生成的目标程序较短, 从而减少了内存的开销并提高了程序的效率。

3. 宏定义可用一个标识符来表示一个字符串, 这个字符串可以是常量、变量或表达式。在宏调用中将用该字符串代换宏名

### 2.2 在 Ubuntu 下预处理的命令

```
wjq1170301026@ubuntu:~/Desktop/hitics$ gcc -E -o hello.i hello.c
wjq1170301026@ubuntu:~/Desktop/hitics$
```

上图即为 ubuntu 中的预处理命令, `gcc -E -o xxx.i xxx.c`

### 2.3 hello 的预处理结果解析

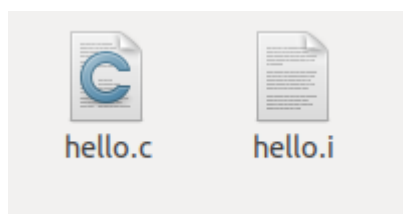


图 2-1 预处理命令执行结果

执行命令后出现了一个 `hello.i` 文件, 内容如下

与原 `hello.c` 文件比较, 其实 `main` 函数里面的内容都没太大的变化, 而三个头文件变成了头文件的源码。虽然长度变长了许多, 但是总的来说并不影响程序的可读性。

```
# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
# 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 424 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
# 427 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 428 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/long-double.h" 1 3 4
# 429 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 425 "/usr/include/features.h" 2 3 4
# 448 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 1 3 4
# 10 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs-64.h" 1 3 4
# 11 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 2 3 4
# 449 "/usr/include/features.h" 2 3 4
# 34 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 2 3 4
# 28 "/usr/include/stdio.h" 2 3 4
```

图 2-2 hello.i 用来描述调用的库在计算机中的位置的语句

上图语句用来描述调用的库在计算机中的位置

```
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;

typedef signed char __int8_t;
typedef unsigned char __uint8_t;
typedef signed short int __int16_t;
typedef unsigned short int __uint16_t;
typedef signed int __int32_t;
typedef unsigned int __uint32_t;

typedef signed long int __int64_t;
typedef unsigned long int __uint64_t;
```

图 2-3 hello.i 用来声明变量和函数的语句

上图语句用来声明变量和函数

将 hello.i 拖到最后可发现 main 函数



```
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;
}
```

图 2-4 hello.i 中的 main 函数

并没有太大变化

## 2.4 本章小结

本章节简单介绍了 c 语言在编译前的预处理过程，简单介绍了预处理过程的概念和作用，对预处理过程进行演示，并举例说明预处理的结果还有解析预处理的过程。

(第 2 章 0.5 分)

## 第 3 章 编译

### 3.1 编译的概念与作用

概念：将预处理好的高级语言程序文本翻译成能执行相同操作的汇编语言的过程。

作用：1.扫描（词法分析），2.语法分析，3.语义分析，4.源代码优化（中间语言生），5.代码生成，目标代码优化。

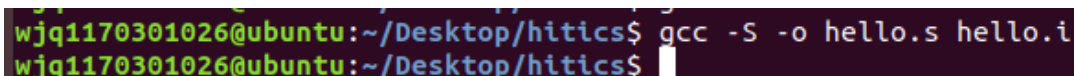
【1】将源代码程序输入扫描器，将源代码的字符序列分割成一系列记号。例  
`array[index] = (index + 4) * (2 + 6);`

【2】基于词法分析得到的一系列记号，生成语法树。

【3】由语义分析器完成，指示判断是否合法，并不判断对错。又分静态语义：隐含浮点型到整型的转换，会报 `warning`，

注意：这儿的编译是指从 `.i` 到 `.s` 即预处理后的文件到生成汇编语言程序

### 3.2 在 Ubuntu 下编译的命令

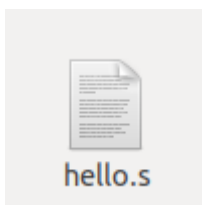


```
wjq1170301026@ubuntu:~/Desktop/hitics$ gcc -S -o hello.s hello.i
wjq1170301026@ubuntu:~/Desktop/hitics$
```

图 3.1 在 Ubuntu 下编译的命令

上图为在 Ubuntu 下编译的命令：`gcc -S -o xxx.s xxx.i`

### 3.3 Hello 的编译结果解析



生成了一个汇编语言程序 `hello.s`

```
\345\247\223\345\220\215\357\274\201"
.LC1:
    .string "Hello %s %s\n"
    .text
    .globl main
    .type   main, @function

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
```

图 3.2 hello.s

### 3.3.0 汇编指令

- a) `.file`  
声明源文件
- b) `.text`  
以下是代码段
- c) `.section .rodata`  
以下是 `rodata` 节
- d) `.globl`  
声明一个全局变量
- e) `.type`  
用来指定是函数类型或是对象类型
- f) `.size`  
声明大小
- g) `.long`、`.string`

声明一个 long、string 类型

#### h) .align

声明对指令或者数据的存放地址进行对齐的方式

### 3.3.1 数据

hello.s 中用到的 C 数据类型有：整数、字符串、数组。

字符串

程序中的字符串分别是：

“Usage: Hello 学号 姓名! \n”，第一个 printf 传入的输出格式化参数，在 hello.s 中声明如图 3.2，可以发现字符串被编码成 UTF-8 格式，一个汉字在 utf-8 编码中占三个字节，一个\代表一个字节。

“Hello %s %s\n”，第二个 printf 传入的输出格式化参数，在 hello.s 中声明如图 3.2。

其中后两个字符串都声明在了.rodata 只读数据节。

```
.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
main:
```

图 3.2 hello.s 中声明在.LC0 和.LC1 段中的字符串

整数

程序中涉及的整数有：

int sleepsecs: sleepsecs 在 C 程序中被声明为全局变量，且已经被赋值，编译器处理时在.data 节声明该变量，.data 节存放已经初始化的全局和静态 C 变量。在图 3.3 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（long 类型在 linux 下与 int 相同为 4B，将 int 声明为 long 应该是编译器偏好）。

```

        .text
        .globl sleepsecs
        .data
        .align 4
        .type sleepsecs, @object
        .size sleepsecs, 4
sleepsecs:
        .long 2
        .section .rodata

```

图 3.3 hello.s 中 sleepsecs 的声明

int i: 编译器将局部变量存储在寄存器或者栈空间中，在 hello.s 中编译器将 i 存储在栈上空间-4(%rbp)中，可以看出 i 占据了栈中的 4B。

int argc: 作为第一个参数传入。

立即数: 其他整形数据的出现都是以立即数的形式出现的，直接硬编码在汇编代码中。

数组

程序中涉及数组的是: char \*argv[] main, 函数执行时输入的命令行, argv 作为存放 char 指针的数组同时是第二个参数传入。

argv 单个元素 char\*大小为 8B, argv 指针指向已经分配好的、一片存放着字符指针的连续空间，起始地址为 argv, main 函数中访问数组元素 argv[1],argv[2]时，按照起始地址 argv 大小 8B 计算数据地址取数据，在 hello.s 中，使用两次(%rax)（两次 rax 分别为 argv[1]和 argv[2]的地址）取出其值。如图 3.4。

```

.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)

```

图 3.4 计算地址取出数组值

### 3.3.2 赋值

程序中涉及的赋值操作有：

`int sleepsecs=2.5` : 因为 `sleepsecs` 是全局变量, 所以直接在 `.data` 节中将 `sleepsecs` 声明为值 2 的 `long` 类型数据。

`i=0`: 整型数据的赋值使用 `mov` 指令完成, 根据数据的大小不同使用不同后缀, 分别为:

指令	大小
<b>b</b>	8b (1B)
<b>w</b>	16b (2B)
<b>l</b>	32b (4B)
<b>q</b>	64b (8B)

因为 `i` 是 4B 的 `int` 类型, 所以使用 `movl` 进行赋值, 汇编代码如图 3.5。

```
movl    $0, -4(%rbp)
```

图 3.5 `hello.s` 中变量 `i` 的赋值

### 3.3.3 类型转换

程序中涉及隐式类型转换的是: `int sleepsecs=2.5`, 将浮点数类型的 2.5 转换为 `int` 类型。

当在 `double` 或 `float` 向 `int` 进行类型转换的时候, 程序改变数值和位模式的原则是: 值会向零舍入。例如 1.999 将被转换成 1, -1.999 将被转换成 -1。进一步来讲, 可能会产生值溢出的情况, 与 Intel 兼容的微处理器指定位模式 `[10...000]` 为整数不确定值, 一个浮点数到整数的转换, 如果不能为该浮点数找到一个合适的整数近似值, 就会产生一个整数不确定值。

浮点数默认类型为 `double`, 所以上述强制转化是 `double` 强制转化为 `int` 类型。遵从向零舍入的原则, 将 2.5 舍入为 2。

### 3.3.4 算数操作

进行数据算数操作的汇编指令有：

指令	效果
leaq S,D	D=&S
INC D	D+=1
DEC D	D-=1
NEG D	D=-D
ADD S,D	D=D+S
SUB S,D	D=D-S
IMULQ S	R[%rdx]:R[%rax]=S*R[%rax] (有符号)
MULQ S	R[%rdx]:R[%rax]=S*R[%rax] (无符号)
IDIVQ S	R[%rdx]=R[%rdx]:R[%rax] mod S (有符号) R[%rax]=R[%rdx]:R[%rax] div S
DIVQ S	R[%rdx]=R[%rdx]:R[%rax] mod S (无符号) R[%rax]=R[%rdx]:R[%rax] div S

程序中涉及的算数操作有：

1.i++, 对计数器 i 自增, 使用程序指令 `addl`, 后缀 l 代表操作数是一个 4B 大小的数据。

2.汇编中使用 `leaq .LC1(%rip),%rdi`, 使用了加载有效地址指令 `leaq` 计算 LC1 的段地址 `%rip+.LC1` 并传递给 `%rdi`。

### 3.3.5 关系操作

进行关系操作的汇编指令有：

指令	效果	描述
CMP S1,S2	S2-S1	比较-设置条件码
TEST S1,S2	S1&S2	测试-设置条件码
SET** D	D=**	按照**将条件码设置 D

J**	——	根据**与条件码进行跳转
-----	----	--------------

程序中涉及的关系运算为：

1. `argc!=3`：判断 `argc` 不等于 3。hello.s 中使用 `cmpl $3,-20(%rbp)`，计算 `argc-3` 然后设置条件码，为下一步 `je` 利用条件码进行跳转作准备。

2. `i<10`：判断 `i` 小于 10。hello.s 中使用 `cmpl $9,-4(%rbp)`，计算 `i-9` 然后设置条件码，为下一步 `jle` 利用条件码进行跳转做准备。

### 3.3.6 控制转移

程序中涉及的控制转移有：

1. `if(argv!=3)`：当 `argv` 不等于 3 的时候执行程序段中的代码。如图 3.6，对于 `if` 判断，编译器使用跳转指令实现，首先 `cmpl` 比较 `argv` 和 3，设置条件码，使用 `je` 判断 ZF 标志位，如果为 0，说明 `argv-3=0` `argv==3`，则不执行 `if` 中的代码直接跳转到 `.L2`，否则顺序执行下一条语句，即执行 `if` 中的代码。

```

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

.L2:

```

图 3.6 if 语句的编译

2. `for(i=0;i<10;i++)`：使用计数变量 `i` 循环 10 次。如图 3.7，编译器的编译逻辑是，首先无条件跳转到位于循环体 `.L4` 之后的比较代码，使用 `cmpl` 进行比较，如果 `i<=9`，则跳入 `.L4` `for` 循环体执行，否则说明循环结束，顺序执行 `for` 之后的逻辑。



```

.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

```

图 3.7 for 循环的编译

### 3.3.7 函数操作

函数是一种过程，过程提供了一种封装代码的方式，用一组指定的参数和可选的返回值实现某种功能。P 中调用函数 Q 包含以下动作：

**传递控制：**进行过程 Q 的时候，程序计数器必须设置为 Q 的代码的起始地址，然后在返回时，要把程序计数器设置为 P 中调用 Q 后面那条指令的地址。

**传递数据：**P 必须能够向 Q 提供一个或多个参数，Q 必须能够向 P 中返回一个值。

**分配和释放内存：**在开始时，Q 可能需要为局部变量分配空间，而在返回前，又必须释放这些空间。

64 位程序参数存储顺序（浮点数使用 xmm，不包含）：

1	2	3	4	5	6	7
%rdi	%rsi	%rdx	%rcx	%r8	%r9	栈空间

程序中涉及函数操作的有：

**main 函数：**

传递控制，main 函数因为被调用 call 才能执行（被系统启动函数

\_\_libc\_start\_main 调用), call 指令将下一条指令的地址 dest 压栈, 然后跳转到 main 函数。

传递数据, 外部调用过程向 main 函数传递参数 argc 和 argv, 分别使用 %rdi 和 %rsi 存储, 函数正常出口为 return 0, 将 %eax 设置 0 返回。

分配和释放内存, 使用 %rbp 记录栈帧的底, 函数分配栈帧空间在 %rbp 之上, 程序结束时, 调用 leave 指令, leave 相当于 mov %rbp,%rsp, pop %rbp, 恢复栈空间为调用之前的状态, 然后 ret 返回, ret 相当 pop IP, 将下一条要执行指令的地址设置为 dest。

printf 函数:

传递数据: 第一次 printf 将 %rdi 设置为 “Usage: Hello 学号 姓名! \n” 字符串的首地址。第二次 printf 设置 %rdi 为 “Hello %s %s\n” 的首地址, 设置 %rsi 为 argv[1], %rdx 为 argv[2]。

控制传递: 第一次 printf 因为只有一个字符串参数, 所以 call puts@PLT; 第二次 printf 使用 call printf@PLT。

exit 函数:

传递数据: 将 %edi 设置为 1。

控制传递: call exit@PLT。

sleep 函数:

传递数据: 将 %edi 设置为 sleepsecs。

控制传递: call sleep@PLT。

getchar 函数:

控制传递: call gethcar@PLT

### 3.4 本章小结

本章简述了编译的概念和作用, 具体分析了一个 c 程序是如何被编译器编译成一个汇编程序的过程, 还详细分析了编译器是如何处理 C 语言的各个数据类型以及各类操作的。 (第 3 章 2 分)

## 第 4 章 汇编

### 4.1 汇编的概念与作用

汇编的概念是指的将汇编语言(`xxx.s`)翻译成机器指令,并将这些指令打包成一种叫做可重定位目标程序,并将这个结果保留在(`xxx.o`)中。这里的 `xxx.o` 是二进制文件。汇编过程的作用是将汇编语言转换成机器可以直接读取分析的机器语言。

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

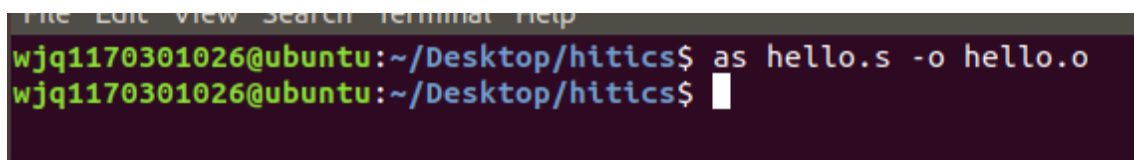


图 4-1 汇编命令

以上即为 Ubuntu 下的汇编命令: `as xxx.s -o xxx.o`

### 4.3 可重定位目标 elf 格式

使用 `readelf -a hello.o > hello.elf` 指令获得 `hello.o` 文件的 ELF 格式。

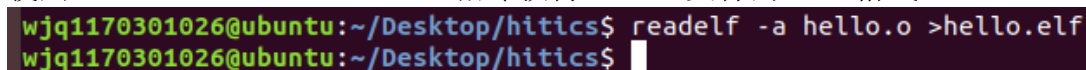


图 4-2 readelf 命令

其组成如下:

**ELF Header:** 以 16B 的序列 Magic 开始, Magic 描述了生成该文件的系统的字的大小和字节顺序, ELF 头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,其中包括 ELF 头的大小、目标文件的类型、机器类型、字节头部表(section header table)的文件偏移,以及节头部表中条目的大小和数量等信息。

```

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

```

图 4-3ELF header

2. Section Header: 描述了.o 文件中出现的各个节的类型、位置、所占空间大小等信息

```

Section Headers:
[Nr] Name           Type           Address          Offset
     Size           EntSize          Flags Link Info Align
[ 0]                NULL           0000000000000000 00000000
     0000000000000000 0000000000000000 0 0 0
[ 1] .text          PROGBITS       0000000000000000 00000040
     000000000000000081 0000000000000000 AX 0 0 1
[ 2] .rela.text     RELA          0000000000000000 00000340
     0000000000000000c0 000000000000000018 I 10 1 8
[ 3] .data          PROGBITS       0000000000000000 000000c4
     000000000000000004 0000000000000000 WA 0 0 4
[ 4] .bss           NOBITS        0000000000000000 000000c8
     000000000000000000 000000000000000000 WA 0 0 1
[ 5] .rodata        PROGBITS       0000000000000000 000000c8
     00000000000000002b 0000000000000000 A 0 0 1
[ 6] .comment       PROGBITS       0000000000000000 000000f3
     00000000000000002b 000000000000000001 MS 0 0 1
[ 7] .note.GNU-stack PROGBITS       0000000000000000 0000011e
     000000000000000000 0000000000000000 0 0 1
[ 8] .eh_frame       PROGBITS       0000000000000000 00000120
     000000000000000038 000000000000000000 A 0 0 8
[ 9] .rela.eh_frame  RELA          0000000000000000 00000400
     000000000000000018 000000000000000018 I 10 8 8
[10] .symtab         SYMTAB        0000000000000000 00000158
     0000000000000000198 000000000000000018 11 9 8
[11] .strtab         STRTAB        0000000000000000 000002f0
     00000000000000004d 000000000000000000 0 0 1
[12] .shstrtab      STRTAB        0000000000000000 00000418

```

图 4-4Section Header

3. `.rela.text`: 重定位节, 这个节包含了 `.text` (具体指令) 节中需要进行重定位的信息。这些信息描述的位置, 在由 `.o` 文件生成可执行文件的时候需要被修改 (重定位)。在这个 `hello.o` 里面需要被重定位的有 `printf`, `puts`, `exit`, `sleepsecs`, `getchar`, `sleep`, `rodata` 里面的两个元素 (`.L0` 和 `.L1` 字符串)。`.rela.eh_frame` 是 `eh_frame` 节的重定位信息。

```
Relocation section '.rela.text' at offset 0x340 contains 8 entries:
```

Offset	Info	Type	Sym. Value	Sym. Name + Addend
0000000000018	0005000000002	R_X86_64_PC32	0000000000000000	.rodata - 4
000000000001d	000c000000004	R_X86_64_PLT32	0000000000000000	puts - 4
0000000000027	000d000000004	R_X86_64_PLT32	0000000000000000	exit - 4
0000000000050	0005000000002	R_X86_64_PC32	0000000000000000	.rodata + 1a
000000000005a	000e000000004	R_X86_64_PLT32	0000000000000000	printf - 4
0000000000060	0009000000002	R_X86_64_PC32	0000000000000000	sleepsecs - 4
0000000000067	000f000000004	R_X86_64_PLT32	0000000000000000	sleep - 4
0000000000076	0010000000004	R_X86_64_PLT32	0000000000000000	getchar - 4

```
Relocation section '.rela.eh_frame' at offset 0x400 contains 1 entry:
```

Offset	Info	Type	Sym. Value	Sym. Name + Addend
0000000000020	0002000000002	R_X86_64_PC32	0000000000000000	.text + 0

图 4-5 `hello.c` 中的重定位节

## 4.4 `hello.o` 的结果解析

1. 分支转移: 反汇编代码跳转指令的操作数使用的不是段名称如 `.L3`, 因为段名称只是在汇编语言中便于编写的助记符, 所以在汇编成机器语言之后显然不存在, 而是确定的地址。

2. 函数调用: 在 `.s` 文件中, 函数调用之后直接跟着函数名称, 而在反汇编程序中, `call` 的目标地址是当前下一条指令。这是因为 `hello.c` 中调用的函数都是共享库中的函数, 最终需要通过动态链接器才能确定函数的运行时执行地址, 在汇编成为机器语言的时候, 对于这些不确定地址的函数调用, 将其 `call` 指令后的相对地址设置为全 0 (目标地址正是下一条指令), 然后在 `.rela.text` 节中为其添加重定位条目, 等待静态链接的进一步确定。

3. 全局变量访问: 在 `.s` 文件中, 访问 `rodata` (`printf` 中的字符串), 使用段名称 `+%rip`, 在反汇编代码中 `0+%rip`, 因为 `rodata` 中数据地址也是在运行时确定, 故访问也需要重定位。所以在汇编成为机器语言时, 将操作数设置为全 0 并添加重定位条目。

```

.LC1:
    .string "Hello %s %s\n"
    .text
    .globl main
    .type main, @function

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
    .cfi_endproc

```

```

0000000000000000 <main>:
0: 55          push %rbp
1: 48 89 e5    mov %rsp,%rbp
4: 48 83 ec 20  sub $0x20,%rsp
8: 89 7d ec    mov %edi,-0x14(%rbp)
b: 48 89 75 e0  mov %rsi,-0x20(%rbp)
f: 83 7d ec 03  cmpl $0x3,-0x14(%rbp)
13: 74 16       je 2b <main+0x2b>
15: 48 8d 3d 00 00 00 00 lea 0x0(%rip),%rdi # 1c <main+0x1c>
18: R_X86_64_PC32 .rodata-0x4
1c: e8 00 00 00 00 callq 21 <main+0x21>
1d: R_X86_64_PLT32 puts-0x4
21: bf 01 00 00 00 mov $0x1,%edi
26: e8 00 00 00 00 callq 2b <main+0x2b>
27: R_X86_64_PLT32 exit-0x4
2b: c7 45 fc 00 00 00 00 movl $0x0,-0x4(%rbp)
32: eb 3b       jmp 6f <main+0x6f>
34: 48 8b 45 e0  mov -0x20(%rbp),%rax
38: 48 83 c0 10  add $0x10,%rax
3c: 48 8b 10     mov (%rax),%rdx
3f: 48 8b 45 e0  mov -0x20(%rbp),%rax
43: 48 83 c0 08  add $0x8,%rax
47: 48 8b 00     mov (%rax),%rax
4a: 48 89 c6     mov %rax,%rsi
4d: 48 8d 3d 00 00 00 00 lea 0x0(%rip),%rdi # 54 <main+0x54>
50: R_X86_64_PC32 .rodata+0x1a
54: b8 00 00 00 00 mov $0x0,%eax
59: e8 00 00 00 00 callq 5e <main+0x5e>
5a: R_X86_64_PLT32 printf-0x4
5e: 8b 05 00 00 00 00 mov 0x0(%rip),%eax # 64 <main+0x64>
60: R_X86_64_PC32 sleepsecs-0x4
64: 89 c7       mov %eax,%edi
66: e8 00 00 00 00 callq 6b <main+0x6b>
67: R_X86_64_PLT32 sleep-0x4
6b: 83 45 fc 01  addl $0x1,-0x4(%rbp)
6f: 83 7d fc 09  cmpl $0x9,-0x4(%rbp)
73: 7e bf       jle 34 <main+0x34>
75: e8 00 00 00 00 callq 7a <main+0x7a>
76: R_X86_64_PLT32 getchar-0x4
7a: b8 00 00 00 00 mov $0x0,%eax
7f: c9         leaveq
80: c3         retq

```

图 4-6hello.s 与 hello.o 的反汇编代码比较

## 4.5 本章小结

本章简述了 hello.s 汇编指令被转换成 hello.o 机器指令的过程，通过 readelf 查看 hello.o 的 ELF、反汇编的方式查看了 hello.o 反汇编的内容，比较其与 hello.s 之间的差别。学习了汇编指令映射到机器指令的具体方式。

(以下格式自行编排，编辑时删除)

(第 4 章 1 分)

## 第 5 章 链接

### 5.1 链接的概念与作用

概念：链接是将多个文件拼接合并成一个可执行文件的过程。

作用：链接行为可以在编译/汇编/加载/运行时执行。链接的存在降低了模块化编程的难度。

### 5.2 在 Ubuntu 下链接的命令

```
wjq1170301026@ubuntu:~/Desktop/hitics$ 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
wjq1170301026@ubuntu:~/Desktop/hitics$
```

图 5-1 链接命令

### 5.3 可执行目标文件 hello 的格式

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

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

图 5-2 hello 的 elf 文件

节头部表数：25



Section Headers:						
[Nr]	Name Size	Type EntSize	Address Flags Link Info	Offset Align		
[ 0]	0000000000000000	NULL	0000000000000000	0 0	0	0
[ 1]	.interp 000000000000001c	PROGBITS	000000000400200 A 0 0	00000200	1	
[ 2]	.note.ABI-tag 0000000000000020	NOTE	00000000040021c A 0 0	0000021c	4	
[ 3]	.hash 0000000000000034	HASH	000000000400240 A 5 0	00000240	8	
[ 4]	.gnu.hash 000000000000001c	GNU_HASH	000000000400278 A 5 0	00000278	8	
[ 5]	.dynsym 00000000000000c0	DYNSYM	000000000400298 A 6 1	00000298	8	
[ 6]	.dynstr 0000000000000057	STRTAB	000000000400358 A 0 0	00000358	1	
[ 7]	.gnu.version 0000000000000010	VERSYM	0000000004003b0 A 5 0	000003b0	2	
[ 8]	.gnu.version_r 0000000000000020	VERNEED	0000000004003c0 A 6 1	000003c0	8	
[ 9]	.rela.dyn 0000000000000030	RELA	0000000004003e0 A 5 0	000003e0	8	
[10]	.rela.plt 0000000000000078	RELA	000000000400410 AI 5 19	00000410	8	
[11]	.init 0000000000000017	PROGBITS	000000000400488 AX 0 0	00000488	4	
[12]	.plt 0000000000000060	PROGBITS	0000000004004a0 AX 0 0	000004a0	16	
[13]	.text 0000000000000132	PROGBITS	000000000400500 AX 0 0	00000500	16	
[14]	.fini 0000000000000009	PROGBITS	000000000400634 AX 0 0	00000634	4	
[15]	.rodata 000000000000002f	PROGBITS	000000000400640 A 0 0	00000640	4	
[16]	.eh_frame 00000000000000fc	PROGBITS	000000000400670 A 0 0	00000670	8	
[17]	.dynamic 00000000000001a0	DYNAMIC	000000000600e50 WA 6 0	00000e50	8	
[18]	.got 0000000000000010	PROGBITS	000000000600ff0 WA 0 0	00000ff0	8	
[19]	.got.plt 0000000000000040	PROGBITS	000000000601000 WA 0 0	00001000	8	
[20]	.data 0000000000000008	PROGBITS	000000000601040 WA 0 0	00001040	4	
[21]	.comment 000000000000002a	PROGBITS	000000000000000 MS 0 0	00001048	1	
[22]	.symtab 0000000000000498	SYMTAB	000000000000000 23 28	00001078	8	
[23]	.strtab 0000000000000150	STRTAB	000000000000000 0 0	00001510	1	
[24]	.shstrtab 00000000000000c5	STRTAB	000000000000000 0 0	00001660	1	

图 5-3 Section Headers 部分

Size 表示的是节头表对应的节的大小

Address 表示的是被载入到虚拟地址后的地址是多少

Offset 表示的是这个节在程序里面的地址偏移量

在 ELF 格式文件中，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 Section Headers 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 Address 是程序被载入到虚拟地址的起始地址。

## 5.4 hello 的虚拟地址空间

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

用 edb 打开 hello，可以看出程序是在 0x00400000 地址开始加载的，结束的地址大约是 0x00400fff

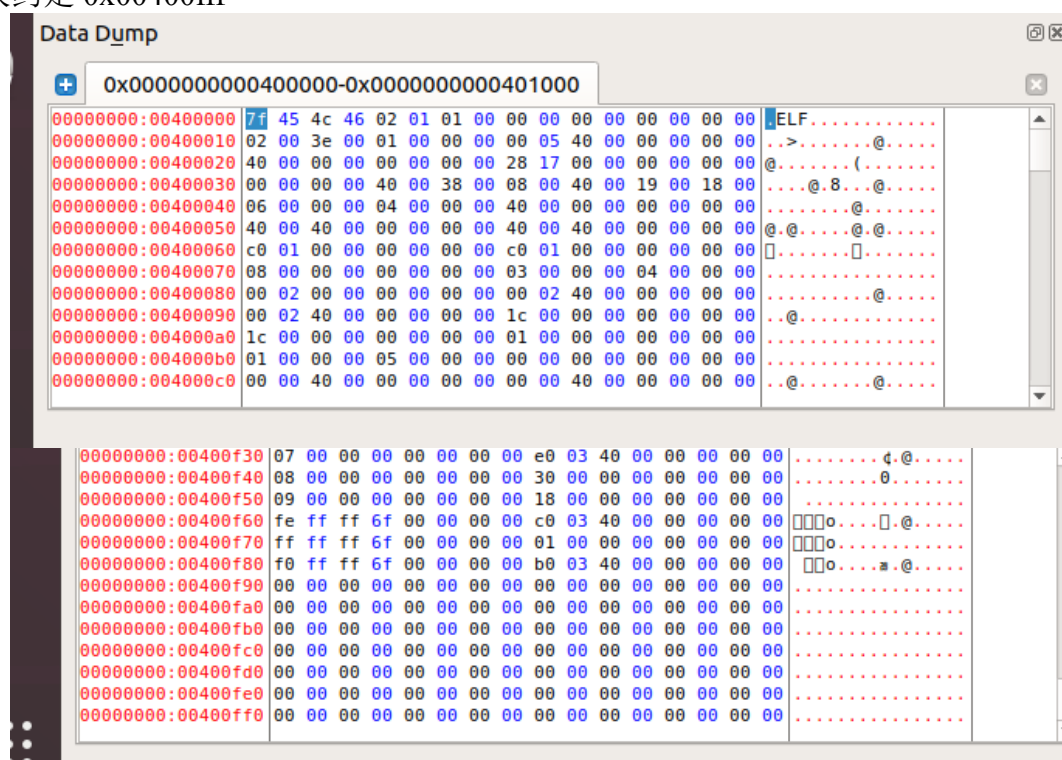


图 5-4 edb 查看 hello

Program Headers:					
Type	Offset FileSiz	VirtAddr MemSiz	PhysAddr Flags Align		
PHDR	0x0000000000000040 0x00000000000001c0	0x0000000000400040 0x00000000000001c0	0x0000000000400040	R	0x8
INTERP	0x0000000000000200 0x000000000000001c	0x0000000000400200 0x000000000000001c	0x0000000000400200	R	0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]					
LOAD	0x0000000000000000 0x000000000000076c	0x0000000000400000 0x000000000000076c	0x0000000000400000	R E	0x200000
LOAD	0x0000000000000e50 0x00000000000001f8	0x0000000000600e50 0x00000000000001f8	0x0000000000600e50	RW	0x200000
DYNAMIC	0x0000000000000e50 0x00000000000001a0	0x0000000000600e50 0x00000000000001a0	0x0000000000600e50	RW	0x8
NOTE	0x000000000000021c 0x0000000000000020	0x000000000040021c 0x0000000000000020	0x000000000040021c	R	0x4
GNU_STACK	0x0000000000000000 0x0000000000000000	0x0000000000000000 0x0000000000000000	0x0000000000000000	RW	0x10
GNU_RELRO	0x0000000000000e50 0x00000000000001b0	0x0000000000600e50 0x00000000000001b0	0x0000000000600e50	R	0x1

图 5-5 hello 的 elf 文件

分析 elf 里面的 Program Headers:

PHDR: 程序头表

INTERP: 程序执行前需要调用的解释器

LOAD: 程序目标代码和常量信息

DYNAMIC: 动态链接器所使用的信息

NOTE:: 辅助信息

GNU\_EH\_FRAME: 保存异常信息

GNU\_STACK: 使用系统栈所需要的权限信息

GNU\_RELRO: 保存在重定位之后只读信息的位置

其余的从.dynamic 到.strtab 节的内容是存放在 0x00400fff 后面

## 5.5 链接的重定位过程分析

(以下格式自行编排, 编辑时删除)

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。

[ 0]	NULL	0000000000000000	00000000
	0000000000000000	0 0 0	
[ 1] .text	PROGBITS	0000000000000000	00000040
	0000000000000081	AX 0 0 1	
[ 2] .rela.text	RELA	0000000000000000	00000340
	00000000000000c0	I 10 1 8	
[ 3] .data	PROGBITS	0000000000000000	000000c4
	0000000000000004	WA 0 0 4	
[ 4] .bss	NOBITS	0000000000000000	000000c8
	0000000000000000	WA 0 0 1	
[ 5] .rodata	PROGBITS	0000000000000000	000000c8
	000000000000002b	A 0 0 1	
[ 6] .comment	PROGBITS	0000000000000000	000000f3
	000000000000002b	MS 0 0 1	
[ 7] .note.GNU-stack	PROGBITS	0000000000000000	0000011e
	0000000000000000	0 0 1	
[ 8] .eh_frame	PROGBITS	0000000000000000	00000120
	0000000000000038	A 0 0 8	
[ 9] .rela.eh_frame	RELA	0000000000000000	00000400
	0000000000000018	I 10 8 8	
[10] .symtab	SYMTAB	0000000000000000	00000158
	00000000000000198	11 9 8	
[11] .strtab	STRTAB	0000000000000000	000002f0
	000000000000004d	0 0 1	
[12] .shstrtab	STRTAB	0000000000000000	00000418
	0000000000000061	0 0 1	

图 5-6 hello.o 的 elf 文件 System Section 部分

Section Headers.						
[Nr]	Name	Type	Address	Offset		
	Size	EntSize	Flags Link Info	Align		
[ 0]	0000000000000000	NULL	0000000000000000	00000000		
	0000000000000000	0000000000000000	0 0	0		
[ 1]	.interp	PROGBITS	0000000000400200	00000200		
	000000000000001c	0000000000000000	A 0 0	1		
[ 2]	.note.ABI-tag	NOTE	000000000040021c	0000021c		
	0000000000000020	0000000000000000	A 0 0	4		
[ 3]	.hash	HASH	0000000000400240	00000240		
	0000000000000034	0000000000000004	A 5 0	8		
[ 4]	.gnu.hash	GNU_HASH	0000000000400278	00000278		
	000000000000001c	0000000000000000	A 5 0	8		
[ 5]	.dynsym	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 19	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		
	00000000000000132	0000000000000000	AX 0 0	16		
[14]	.fini	PROGBITS	0000000000400634	00000634		
	0000000000000009	0000000000000000	AX 0 0	4		
[15]	.rodata	PROGBITS	0000000000400640	00000640		
	000000000000002f	0000000000000000	A 0 0	4		
[16]	.eh_frame	PROGBITS	0000000000400670	00000670		
	00000000000000fc	0000000000000000	A 0 0	8		
[17]	.dynamic	DYNAMIC	0000000000600e50	00000e50		
	000000000000001a0	0000000000000010	WA 6 0	8		
[18]	.got	PROGBITS	0000000000600ff0	00000ff0		
	0000000000000010	0000000000000008	WA 0 0	8		
[19]	.got.plt	PROGBITS	0000000000601000	00001000		
	0000000000000040	0000000000000008	WA 0 0	8		
[20]	.data	PROGBITS	0000000000601040	00001040		
	0000000000000008	0000000000000000	WA 0 0	4		
[21]	.comment	PROGBITS	0000000000000000	00001048		
	000000000000002a	0000000000000001	MS 0 0	1		
[22]	.symtab	SYMTAB	0000000000000000	00001078		
	00000000000000498	0000000000000018	23 28	8		
[23]	.strtab	STRTAB	0000000000000000	00001510		
	00000000000000150	0000000000000000	0 0	1		
[24]	.shstrtab	STRTAB	0000000000000000	00001660		
	00000000000000c5	0000000000000000	0 0	1		

图 5-7 hello 的 elf 文件 System Section 部分

分析一下比 hello.o 多出来的这些节头表：

.interp: 保存 ld.so 的路径

.note.ABI-tag

.note.gnu.build-id: 编译信息表

.gnu.hash: gnu 的扩展符号 hash 表

.dynsym: 动态符号表

.dynstr: 动态符号表中的符号名称  
 .gnu.version: 符号版本  
 .gnu.version\_r: 符号引用版本  
 .rela.dyn: 动态重定位表  
 .rela.plt: .plt 节的重定位条目  
 .init: 程序初始化  
 .plt: 动态链接表  
 .fini: 程序终止时需要的执行的指令  
 .eh\_frame: 程序执行错误时的指令  
 .dynamic: 存放被 ld.so 使用的动态链接信息  
 .got: 存放程序中变量全局偏移量  
 .got.plt: 存放程序中函数的全局偏移量  
 .data: 初始化过的全局变量或者声明过的函数  
 再观察 hello.o 的反汇编代码与 hello 的反汇编代码

```
Disassembly of section .text:

0000000000000000 <main>:
 0: 55                push    %rbp
 1: 48 89 e5          mov     %rsp,%rbp
 4: 48 83 ec 20       sub     $0x20,%rsp
 8: 89 7d ec          mov     %edi,-0x14(%rbp)
 b: 48 89 75 e0       mov     %rsi,-0x20(%rbp)
 f: 83 7d ec 03       cmpl    $0x3,-0x14(%rbp)
13: 74 16             je      2b <main+0x2b>
15: 48 8d 3d 00 00 00 lea     0x0(%rip),%rdi    # 1c <main+0x1c>
18: R_X86_64_PC32    .rodata-0x4
1c: e8 00 00 00 00    callq   21 <main+0x21>
1d: R_X86_64_PLT32    puts-0x4
21: bf 01 00 00 00    mov     $0x1,%edi
26: e8 00 00 00 00    callq   2b <main+0x2b>
27: R_X86_64_PLT32    exit-0x4
XTerm c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%rbp)
32: eb 3b            jmp     6f <main+0x6f>
34: 48 8b 45 e0       mov     -0x20(%rbp),%rax
```

图 5-7 hello.o 的反汇编代码

```

Disassembly of section .init:

0000000000400488 <_init>:
400488: 48 83 ec 08          sub    $0x8,%rsp
40048c: 48 8b 05 65 0b 20 00 mov    0x200b65(%rip),%rax
ff8 <__gmon_start__>
400493: 48 85 c0            test   %rax,%rax
400496: 74 02              je     40049a <_init+0x12>
400498: ff d0             callq  *%rax
40049a: 48 83 c4 08        add    $0x8,%rsp
40049e: c3               retq

```

图 5-9 hello 的反汇编代码

我们可以发现，hello.o.s 从.text 节开始，而 hello.s 从.init 节开始

```

Disassembly of section .text:

0000000000000000 <main>:
0: 55                push   %rbp
1: 48 89 e5          mov    %rsp,%rbp
4: 48 83 ec 20       sub    $0x20,%rsp
8: 89 7d ec          mov    %edi,-0x14(%rbp)
b: 48 89 75 e0       mov    %rsi,-0x20(%rbp)
f: 83 7d ec 03       cmpl   $0x3,-0x14(%rbp)
13: 74 16            je     2b <main+0x2b>
15: 48 8d 3d 00 00 00 00 lea     0x0(%rip),%rdi    # 1c <main+0x1c>
18: R_X86_64_PC32    .rodata-0x4
1c: e8 00 00 00 00    callq 21 <main+0x21>
1d: R_X86_64_PLT32    puts-0x4
21: bf 01 00 00 00    mov    $0x1,%edi
26: e8 00 00 00 00    callq 2b <main+0x2b>
27: R_X86_64_PLT32    exit-0x4
2b: c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%rbp)
32: eb 3b            jmp     6f <main+0x6f>
34: 48 8b 45 e0       mov    -0x20(%rbp),%rax
38: 48 83 c0 10       add    $0x10,%rax
3c: 48 8b 10          mov    (%rax),%rdx
3f: 48 8b 45 e0       mov    -0x20(%rbp),%rax
43: 48 83 c0 08       add    $0x8,%rax
47: 48 8b 00          mov    (%rax),%rax
4a: 48 89 c6          mov    %rax,%rsi
4d: 48 8d 3d 00 00 00 00 lea     0x0(%rip),%rdi    # 54 <main+0x54>
50: R_X86_64_PC32    .rodata+0x1a
54: b8 00 00 00 00    mov    $0x0,%eax
59: e8 00 00 00 00    callq 5e <main+0x5e>
5a: R_X86_64_PLT32    printf-0x4
5e: 8b 05 00 00 00 00 00 mov    0x0(%rip),%eax    # 64 <main+0x64>
60: R_X86_64_PC32    sleepsecs-0x4
64: 89 c7            mov    %eax,%edi
66: e8 00 00 00 00    callq 6b <main+0x6b>
67: R_X86_64_PLT32    sleep-0x4
6b: 83 45 fc 01       addl   $0x1,-0x4(%rbp)
6f: 83 7d fc 09       cmpl   $0x9,-0x4(%rbp)
73: 7e bf            jle    34 <main+0x34>
75: e8 00 00 00 00    callq 7a <main+0x7a>
76: R_X86_64_PLT32    getchar-0x4
7a: b8 00 00 00 00    mov    $0x0,%eax
7f: c9              leaveq
80: c3              retq

```

图 5-10 hello.o 的反汇编代码



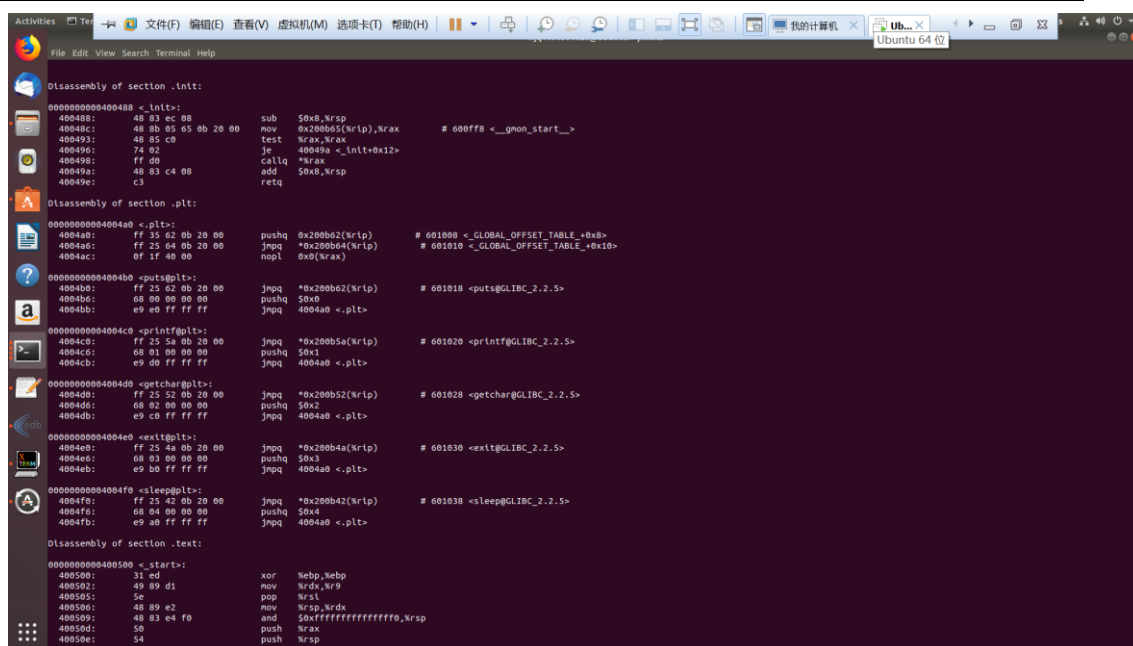


图 5-11 hello 的反汇编代码

在 hello.s 中导入了诸如 puts、printf、getchar、sleep 等在 hello 程序中使用的函数，而这些函数的在 hello.o.s 中就没有出现

可以发现，在 hello.o.s 中调用函数都是使用 call+<main+偏移量的做法>，而在 hello.s 是直接使用 call+<函数名> 的方法来直接调用的。

总结 hello.o 和 hello 的区别如下：

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、crtm.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。

3) .rodata 引用：链接器解析重定条目时发现两个类型为 R\_X86\_64\_PC32 的对 .rodata 的重定位（printf 中的两个字符串），.rodata 与 .text 节之间的相对距离确定，因此链接器直接修改 call 之后的值为目标地址与下一条指令的地址之差，指向相应的字符串。

## 5.6 hello 的执行流程

期间主要执行的函数及其地址罗列如下：

```
ld-2.27.so!_dl_start: 0x7fce 8cc38ea0
ld-2.27.so!_dl_init: 0x7fce 8cc47630
hello!_start: 0x400500
libc-2.27.so!__libc_start_main: 0x7fce 8c867ab0
-libc-2.27.so!__cxa_atexit: 0x7fce 8c889430
-libc-2.27.so!__libc_csu_init: 0x4005c0
hello!_init: 0x400488
libc-2.27.so!_setjmp: 0x7fce 8c884c10
-libc-2.27.so!_sigsetjmp: 0x7fce 8c884b70
--libc-2.27.so!__sigjmp_save: 0x7fce 8c884bd0
hello!main: 0x400532
hello!puts@plt: 0x4004b0
hello!exit@plt: 0x4004e0
*hello!printf@plt: --
*hello!sleep@plt: --
*hello!getchar@plt: --
ld-2.27.so!_dl_runtime_resolve_xsave: 0x7fce 8cc4e680
-ld-2.27.so!_dl_fixup: 0x7fce 8cc46df0
--ld-2.27.so!_dl_lookup_symbol_x: 0x7fce 8cc420b0
libc-2.27.so!exit: 0x7fce 8c889128
```

## 5.7 Hello 的动态链接分析

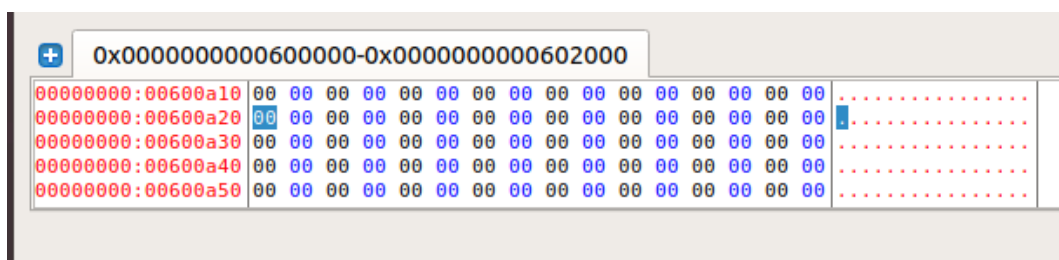


图 5-12 dl\_init 前



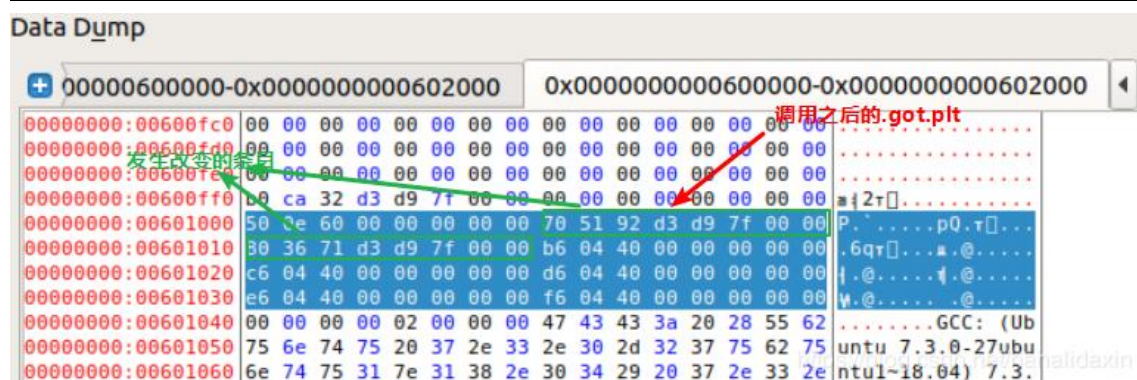


图 5-13dl\_init 后

在 edb 调试之后我们发现原先 0x00600a10 开始的 global\_offset 表是全 0 的状态，在执行过 dl\_init 之后被赋上了相应的偏移量的值。这说明 dl\_init 操作是给程序赋上当前执行的内存地址偏移量，这是初始化 hello 程序的一步。

## 5.8 本章小结

本章介绍了链接过程中对 hello 的处理以及对链接后生成的可执行文件 hello 的分析，包括 hello 的 elf 格式、hello 的虚拟地址空间、hello 的重定位以及动态链接的过程。链接完成后，hello 终于成为了一个可以运行的程序

(第 5 章 1 分)

## 第 6 章 hello 进程管理

### 6.1 进程的概念与作用

概念：进程（Process）是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。

作用：进程是程序的执行实例，即运行中的程序

### 6.2 简述壳 shell-bash 的作用与处理流程

shell 是一个应用程序，在操作系统中提供了一个用户与系统内核进行交互的界面。处理过程如下

1. 读取用户的输入
2. 分析输入内容，获得输入参数
3. 如果是内核命令则直接执行，否则调用相应的程序执行命令
4. 在程序运行期间，shell 需要监视键盘的输入内容，并且做出相应的反应

### 6.3 Hello 的 fork 进程创建过程

```
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
```

图 6-1 hello 的 fork 过程

在终端中键入 `./hello 1170301026 wjq`，运行的终端程序会对输入的命令行进

行解析，因为 `hello` 不是一个内置的 `shell` 命令所以解析之后终端程序判断 `./hello` 的语义为执行当前目录下的可执行目标文件 `hello`，之后终端程序首先会调用 `fork` 函数创建一个新的运行的子进程，新创建的子进程几乎但不完全与父进程相同，子进程得到与父进程用户级虚拟地址空间相同的（但是独立的）一份副本，这就意味着，当父进程调用 `fork` 时，子进程可以读写父进程中打开的任何文件。父进程与子进程之间最大的区别在于它们拥有不同的 `PID`。

父进程与子进程是并发运行的独立进程，内核能够以任意方式交替执行它们的逻辑控制流的指令。在子进程执行期间，父进程默认选项是显示等待子进程的完成。

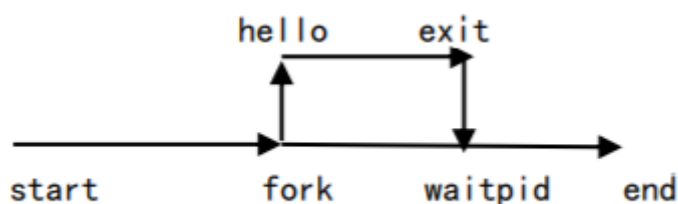


图 6-2 简单流程图

## 6.4 `hello` 的 `execve` 过程

`fork` 之后，`shell` 在子进程中调用 `execve` 函数，在当前进程的上下文中加载并运行 `hello` 程序，`execve` 调用驻留在内存中的被称为启动加载器的操作系统代码来执行 `hello` 程序，加载器删除子进程现有的虚拟内存段，并创建一组新的代码、数据、堆和栈段。新的栈和堆段被初始化为零，通过将虚拟地址空间中的页映射到可执行文件的页大小的片，新的代码和数据段被初始化为可执行文件中的内容，然后跳转到 `_start`，`_start` 函数调用系统启动函数 `__libc_start_main` 来初始化环境，调用用户层中 `hello` 的 `main` 函数，并在需要的时候将控制返回给内核。

每一个进程都有一段唯一属于自己的内存地址段，在 `execve` 运行时，开始先是从 `0x00400000`（对于 32 位系统来说是 `0x8048000`）开始程序的执行。先是从可执行文件中加载的内容，然后是运行时的堆栈和共享库的存储器映射区域。

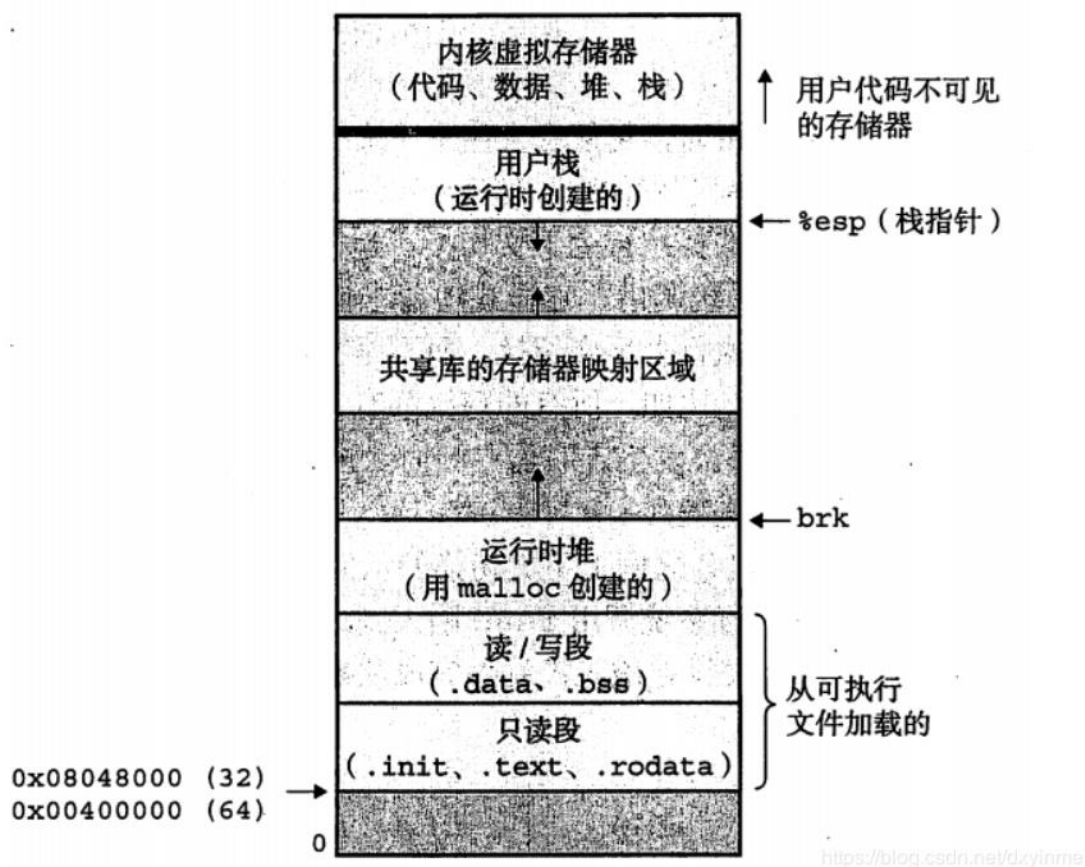


图 6-3 加载器创建的内存映像

## 6.5 Hello 的进程执行

操作系统内核使用一中称为上下文切换的较高层形式的异常控制流来实现多任务：内核为每个进程维持一个上下文，上下文就是内核重新启动一个被抢占的进程所需的状态，它由一些对象的值组成，这些对象包括通用目的寄存器、浮点寄存器、程序计数器、用户栈、状态寄存器、内核栈和各种内核数据结构，比如描述地址空间的页表、包含有关当前进程信息的进程表，以及包含进程一打开文件的信息的文件表。上下文切换的流程是：1.保存当前进程的上下文。2.恢复某个先前被抢占的进程被保存的上下文。3.将控制传递给这个新恢复的进程。

为了使操作系统内核提供一个无懈可击的进程抽象，处理器必须提供一种机制，限制一个应用可以执行的指令以及它可以访问的地址空间范围。处理器通常使用某个控制寄存器的一个模式位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。

当调用 `sleep` 之前，如果 `hello` 程序不被抢占则顺序执行，假如发生被抢占的情况，则进行上下文切换，上下文切换是由内核中调度器完成的，当内核调度新的进程运行后，它就会抢占当前进程，并进行 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。

`hello` 初始运行在用户模式，在 `hello` 进程调用 `sleep` 之后陷入内核模式，内核处理休眠请求主动释放当前进程，并将 `hello` 进程从运行队列中移出加入等待队列，定时器开始计时，内核进行上下文切换将当前进程的控制权交给其他进程，当定时器到时 (2.5secs) 发送一个中断信号，此时进入内核状态执行中断处理，将 `hello` 进程从等待队列中移出重新加入到运行队列，成为就绪状态，`hello` 进程就可以继续进行自己的控制逻辑流了。

当 `hello` 调用 `getchar` 的时候，实际落脚到执行输入流是 `stdin` 的系统调用 `read`，`hello` 之前运行在用户模式，在进行 `read` 调用之后陷入内核，内核中的陷阱处理程序请求来自键盘缓冲区的 DMA 传输，并且安排在完成从键盘缓冲区到内存的数据传输后，中断处理器。此时进入内核模式，内核执行上下文切换，切换到其他进程。当完成键盘缓冲区到内存的数据传输时，引发一个中断信号，此时内核从其他进程进行上下文切换回 `hello` 进程。

## 6.6 `hello` 的异常与信号处理

Ctrl-Z:

```
wjq1170301026@ubuntu:~/hitics$ ./hello 1170301026 wjq
Hello 1170301026 wjq
Hello 1170301026 wjq
Hello 1170301026 wjq
Hello 1170301026 wjq
^Z
[1]+  Stopped                  ./hello 1170301026 wjq
wjq1170301026@ubuntu:~/hitics$ ps
  PID TTY          TIME CMD
  9171 pts/5        00:00:00 bash
  9180 pts/5        00:00:00 hello
  9182 pts/5        00:00:00 ps
```

图 6-4 键入 Ctrl-Z

这个操作向进程发送了一个 `sigstsp` 信号，让进程暂时挂起，输入 `ps` 命令符可以发现 `hello` 进程还没有被关闭。

Ctrl-C:

```
wjq1170301026@ubuntu:~/hitics$ ./hello 1170301026 wjq
Hello 1170301026 wjq
Hello 1170301026 wjq
^C
wjq1170301026@ubuntu:~/hitics$ ps
  PID TTY          TIME CMD
  9171 pts/5        00:00:00 bash
```

图 6-5 键入 Ctrl-C

在程序运行过程中，键入 Ctrl-C，会向当前进程发送一个 `SIGINT` 信号，从而使当前进程中断。

回车:

```
wjq1170301026@ubuntu:~/hitics$ ./hello 1170301026 wjq
Hello 1170301026 wjq
Hello 1170301026 wjq

Hello 1170301026 wjq
Hello 1170301026 wjq
Hello 1170301026 wjq
Hello 1170301026 wjq
```

图 6-6 键入回车

键入回车将会被忽略

`ps` 命令：查看当前进程及运行时间

```
wjq1170301026@ubuntu:~/hitics$ ps
  PID TTY          TIME CMD
  9171 pts/5        00:00:00 bash
  9237 pts/5        00:00:00 hello
  9238 pts/5        00:00:00 ps
```

图 6-7 `ps` 命令

`fg` 命令：使挂起的进程重新在前台执行

```
wjq1170301026@ubuntu:~/hitics$ jobs
[1]+  Stopped                  ./hello 1170301026 wjq
wjq1170301026@ubuntu:~/hitics$ fg
./hello 1170301026 wjq
Hello 1170301026 wjq
Hello 1170301026 wjq
^Z
[1]+  Stopped                  ./hello 1170301026 wjq
```

图 6-8 `jobs` 命令和 `fg` 命令



## 6.7 本章小结

本章介绍了进程的概念和作用，描述了 `shell` 如何在用户和系统内核之间建起一个交互的桥梁。讲述了 `shell` 的基本操作以及各种内核信号和命令，还总结了 `shell` 是如何 `fork` 新建子进程、`execve` 如何执行进程、`hello` 进程如何在内核和前端中反复跳跃运行的。



## 第 7 章 hello 的存储管理

### 7.1 hello 的存储器地址空间

逻辑地址：又称相对地址，是程序运行由 CPU 产生的与段相关的偏移地址部分。描述一个程序运行段的地址。

物理地址：程序运行时加载到内存地址寄存器中的地址，内存单元的真正地址。他是在前端总线上传输的而且是唯一的。在 hello 程序中，表示了这个程序运行时的一条确切的指令在内存地址上的具体哪一块进行执行。

线性地址：即虚拟地址，是经过段机制转化之后用于描述程序分页信息的地址。是对程序运行区块的一个抽象映射。以 hello 为例子，是一个描述：“hello 程序应该在内存的哪些块上运行。”

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

### 7.2 Intel 逻辑地址到线性地址的变换-段式管理

为了进行段式管理，每道程序在系统中都有一个段（映象）表来存放该道程序各段装入主存的状况信息。段表中的每一项（对应表中的每一行）描述该道程序一个段的基本状况，由若干个字段提供。段名字段用于存放段的名称，段名一般是有其逻辑意义的，也可以转换成用段号指明。由于段号从 0 开始顺序编号，正好与段表中的行号对应，如 2 段必是段表中的第 3 行，这样，段表中就可不设段号（名）字段。装入位字段用来指示该段是否已经调入主存，“1”表示已装入，“0”表示未装入。在程序的执行过程中，各段的装入位随该段是否活跃而动态变化。当装入位为“1”时，地址字段用于表示该段装入主存中起始（绝对）地址，当装入位为“0”时，则无效（有时机器用它表示该段在辅存中的起始地址）。段长字段指明该段的大小，一般以字数或字节数为单位，取决于所用的编址方式。段长字段是用来判断所访问的地址是否越出段界的界限保护检查用的。访问方式字段用来标记该段允许的访问方式，如只读、可写、只能执行等，以提供段的访问方式保护。除此之外，段表中还可以根据需要设置其它的字段。段表本身也是一个段，一般常驻在主存中，也可以存在辅存中，需要时再调入主存。假设系统在主存中最多可同时有 N 道程序，可设 N 个段表基址寄存器。对应于每道程序，由基号（程序号）指明使用哪

个段表基址寄存器。段表基址寄存器中的段表基址字段指向该道程序的段表在主存中的起始地址。段表长度字段指明该道程序所用段表的行数，即程序的段数。

段式内存管理方式就是直接将逻辑地址转换成物理地址，也就是 CPU 不支持分页机制。其地址的基本组成方式是段号+段内偏移地址。

逻辑空间分为若干个段，其中每一个段都定义了一组具有完整意义的信息，逻辑地址对应于逻辑空间，如（主程序的 main()）函数，如图 7-1 所示

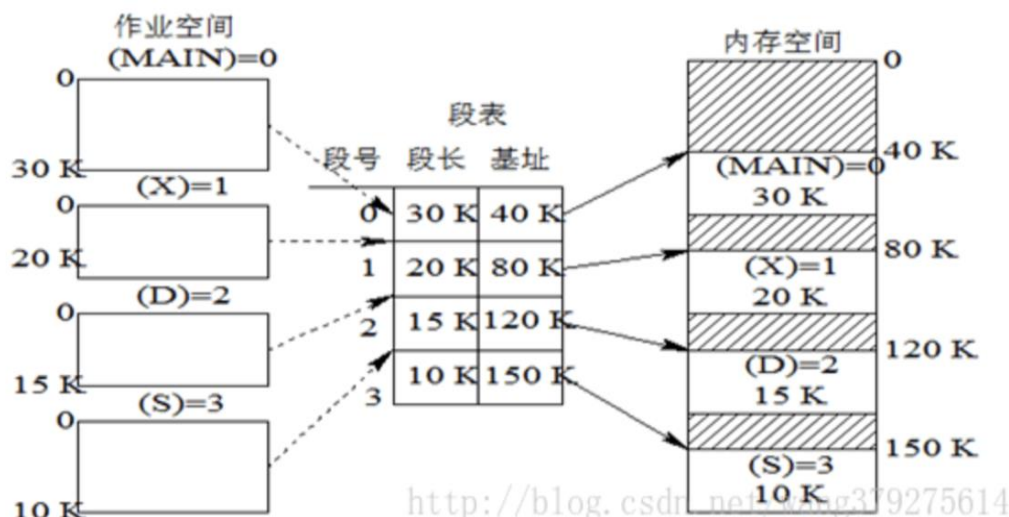


图 7-1(转) 逻辑空间

首先给定一个完整的逻辑地址[段选择符：段内偏移地址]，

1.看段选择描述符中的 T1 字段是 0 还是 1,可以知道当前要转换的是 GDT 中的段，还是 LDT 中的段，再根据指定的相应的寄存器，得到其地址和大小，我们就有了一个数组了。

2.拿出段选择符中的前 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 Base，即基地址就知道了。

3.把基地址 Base+Offset,就是要转换的下一个阶段的物理地址。

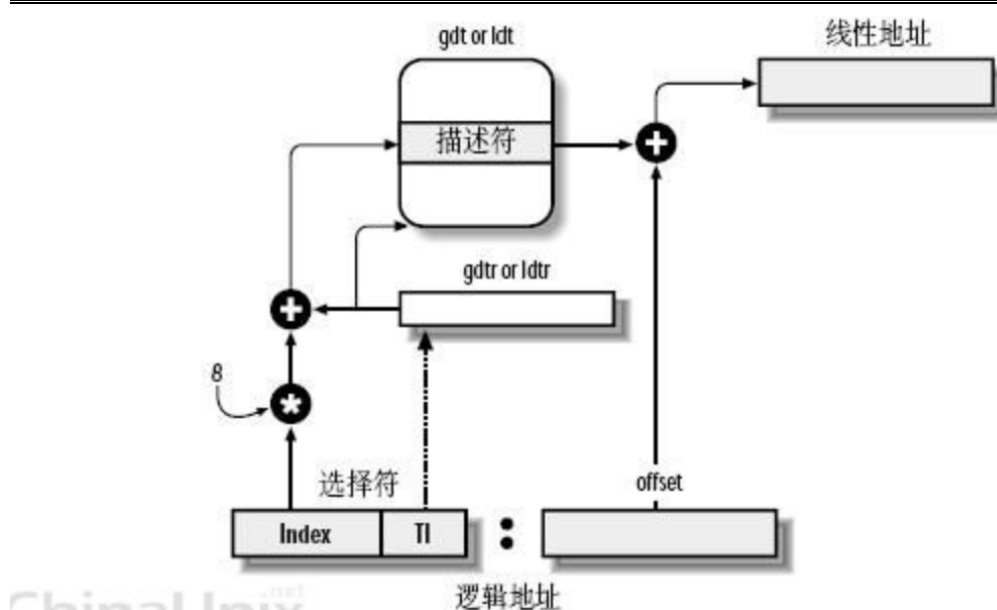


图 7-2(转) 逻辑地址的转换

### 7.3 Hello 的线性地址到物理地址的变换-页式管理

CPU 的页式内存管理单元负责把一个线性地址转换为物理地址。从管理和效率的角度出发，线性地址被划分成固定长度单位的数组，称为页 (page)。例如，一个 32 位的机器，线性地址可以达到 4G，用 4KB 为一个页来划分，这样，整个线性地址就被划分为一个  $2^{20}$  次方的的大数组，共有 2 的 20 次方个页，也就是 1M 个页，我们称之为页表，改页表中每一项存储的都是物理页的基地址。

如果内存页按照这种方式进行管理，管理内存页需要  $2^{20}$  次方的数组，其中每个数组都是 32bit，也就是 4B（其中前 20 位存储物理内存页的基地址，后面的 12 位留空，用于与给定的线性地址的后 12 位拼接起来一起组成一个真实的物理地址，寻找数据的所在。这样就需要为每个进程维护  $4B \times 2^{20} = 4MB$  的内存空间，极大地消耗了内存。

为了能够尽可能的节约内存，CPU 在页式内存管理方式中引入了两级的页表结构，如图 7-3 所示。

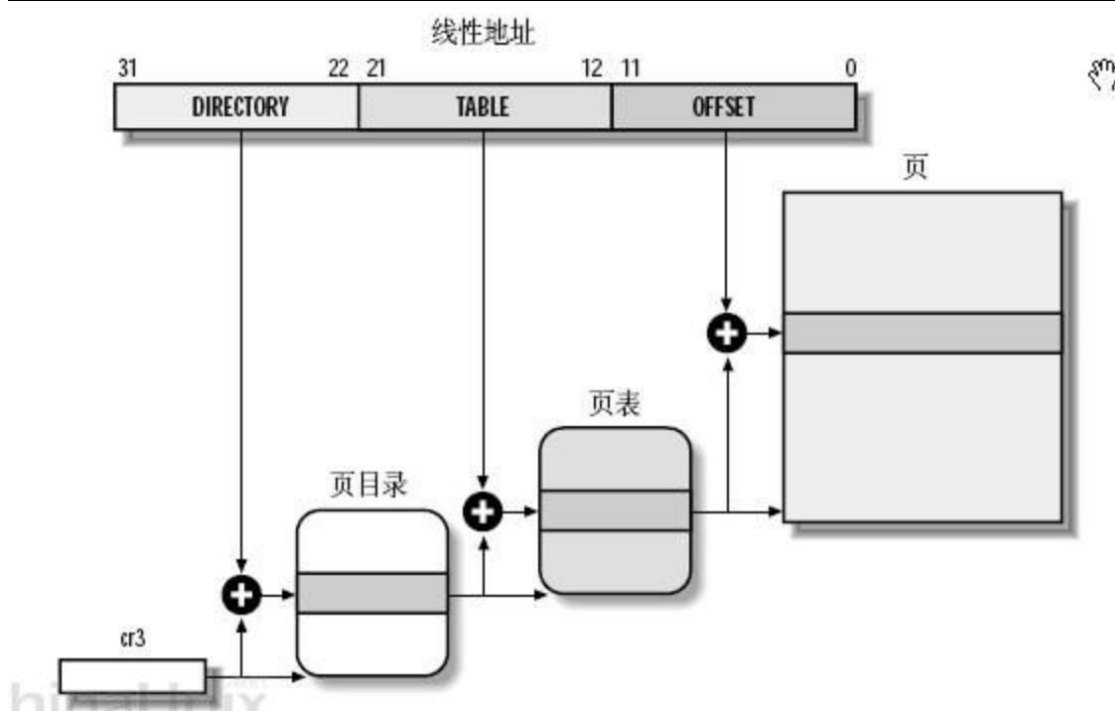


图 7-3(转) 页式管理

线性地址转换成物理地址的过程如下：

1、从 CR3 中取出进程的页目录的地址（操作系统在负责进程的调度的时候，将这个地址装入对应的 CR3 地址寄存器），取出其前 20 位，这是页目录的基地址；

2、根据取出来的页目录的基地址以及线性地址的前十位，进行组合得到线性地址的前十位的索引对应的项在页目录中地址，根据该地址可以取到该地址上的值，该值就是二级页表项的基址；当然你说地址是 32 位，这里只有 30 位，其实当取出线性地址的前十位之后还会该该前十位左移 2 位，也就是乘以 4，一共 32 位；之所以这么做是因为每个地址都是 4B 的大小，因此其地址肯定是 4 字节对齐的，因此左移两位之后的 32 位的值恰好就是该前十位的索引项的所对应值的起始地址，只要从该地址开始向后读四个字节就得到了该十位数字对应的页目录中的项的地址，取该地址的值就是对应的页表项的基址；

3、根据第二步取到的页表项的基址，取其前 20 位，将线性地址的 10-19 位左移 2 位（原因和第 2 步相同），按照和第 2 步相同的方式进行组合就可以得到线性地址对应的物理页框在内存中的地址在二级页表中的地址的起始地址，根据该地址向后读四个字节就得到了线性地址对应的物理页框在内存中的地址在二级页表中的地址，然后取该地址上的值就得到线性地址对应的物理页框在内存中的基址；（这一步的地址比较绕，还请仔细琢磨，反复推敲）

4、根据第 3 步取到的基址，取其前 20 位得到物理页框在内存中的基址，再根据线性地址最后的 12 位的偏移量得到具体的物理地址，取该地址上的值就是最后要得到值；

## 7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

我们这里说段页表的建立。

比如 32 位 CPU，4G 的寻址空间可分为 4094 个段（4G/1MB）

所以可以建立 4096 个对应的关系而实际的内存肯定没到 4G（VA-PA 可多对一）

所以首先要在内存中指定存放该对应表的其实位置（可通过 CP15 协处理器指定）

映射表的大概示意图如下

VA(0-4095)		PA（高 12 位）
0	----	有效 PA 高 12bit + AP + DOMAIN + cacheable + bufferable + 描述符
1	----	有效 PA 高 12bit + AP + DOMAIN + cacheable + bufferable + 描述符
2	----	有效 PA 高 12bit + AP + DOMAIN + cacheable + bufferable + 描述符
4094	----	有效 PA 高 12bit + AP + DOMAIN + cacheable + bufferable + 描述符
4095	----	有效 PA 高 12bit + AP + DOMAIN + cacheable + bufferable + 描述符

AP：AP 权限为所有用户可读写 在多进程时使用 用来保护进程

DOMAIN：配合 AP 使用 权限管理

cacheable：C 位 使能 MMU 缓存

bufferable：B 位 使能 MMU 缓冲区

描述符：MMU 使用段描述符(还有页描述符大页(64KB)小页(4KB)和极小页(1KB))

以下是建立 1MB 映射的 C 语言描述：

```
phyaddr = 0x30000000;
```

```
viraddr = 0xa0000000;
```

```
*(mmu_ttb_base + (viraddr >> 20)) = ((phyaddr & 0xfff00000) | 低 20 位的属性描述);
```

VA 到 PA 转化过程:

CPU 发出 VA: 比如 0xa0000123, 转化过程如下图:

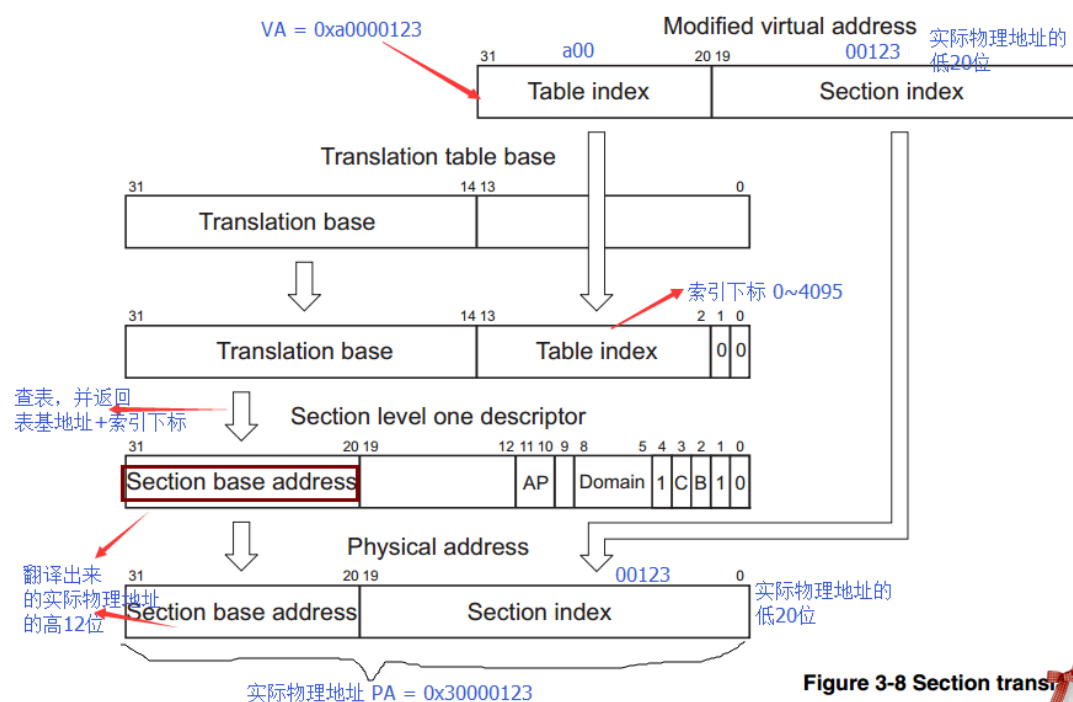


图 7-4(转) VA 到 PA 转化过程

## 7.5 三级 cache 支持下的物理内存访问

得到物理地址之后, 先将物理地址拆分成 CT (标记) + CI (索引) + CO (偏移量), 然后在一级 cache 内部找, 如果未能寻找到标记位为有效的字节 (miss) 的话就去二级和三级 cache 中寻找对应的字节, 找到之后返回结果。(如图)

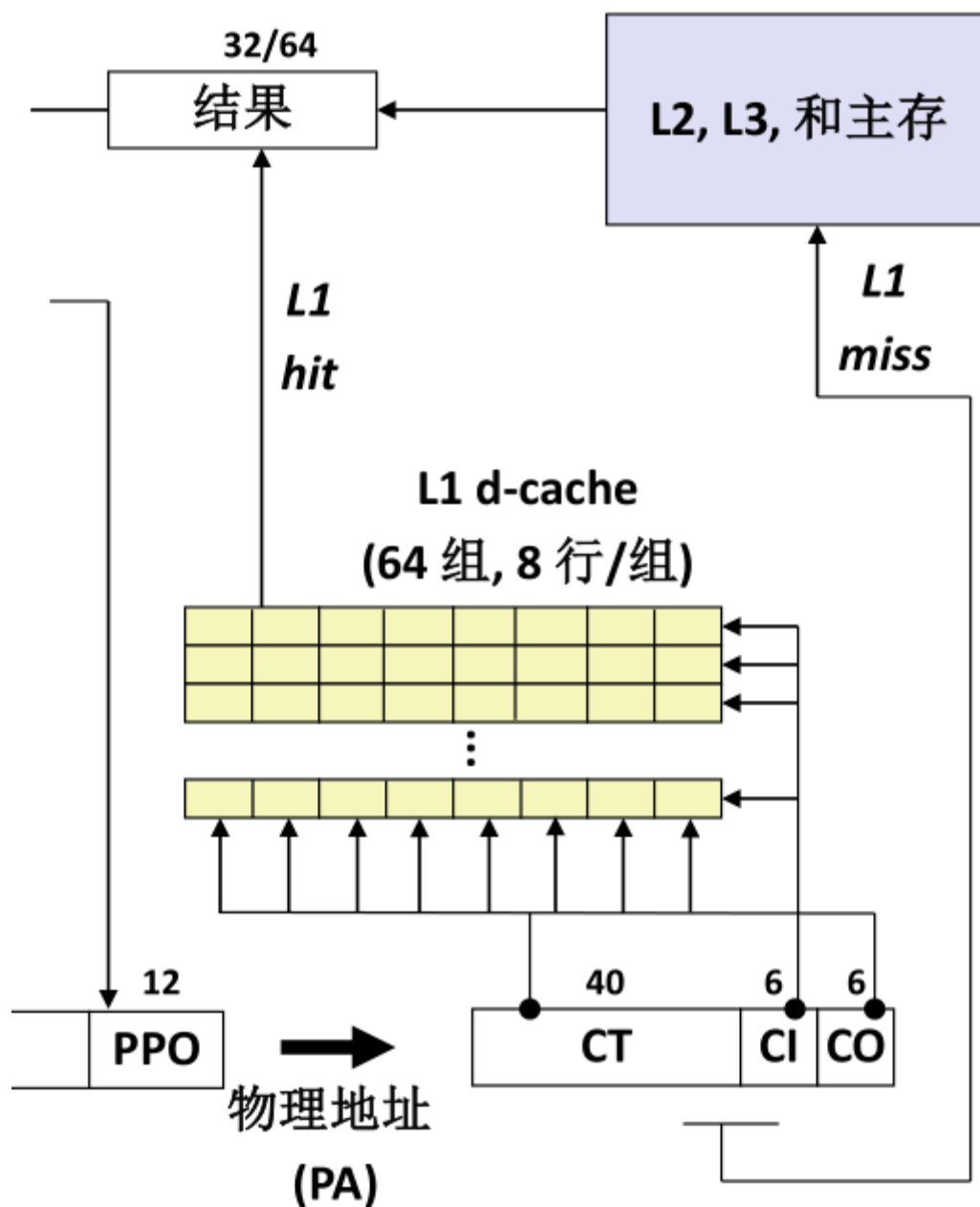


图 7-5 物理内存访问过程

## 7.6 hello 进程 fork 时的内存映射

在用 fork 创建虚拟内存的时候，要经历以下步骤：

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

两个进程的每个页面都标记为只读页面

两个进程的每个 `vm_area_struct` 都标记为私有，这样就只能在写入时复制。

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

下图为创建的进程在地址段中表示

```

2b:  c7 45 fc 00 00 00 00  mov
32:  eb 3b                    jmp
34:  48 8b 45 e0              mov
38:  48 83 c0 10              add
3c:  48 8b 10                 mov
3f:  48 8b 45 e0              mov
43:  48 83 c0 08              add
47:  48 8b 00                 mov
4a:  48 89 c6                 mov
4d:  48 8d 3d 00 00 00 00    lea
                                50: R_X86_6
54:  b8 00 00 00 00          mov
59:  e8 00 00 00 00          cal
                                5a: R_X86_6
5e:  8b 05 00 00 00 00      mov
                                60: R_X86_6
64:  89 c7                    mov
66:  e8 00 00 00 00          cal
                                67: R_X86_6
6b:  83 45 fc 01              add
6f:  83 7d fc 09              cmp
73:  7e bf                    jle
75:  e8 00 00 00 00          cal
                                76: R_X86_6
7a:  b8 00 00 00 00          mov
7f:  c9                       lea
80:  c7                       ret

```

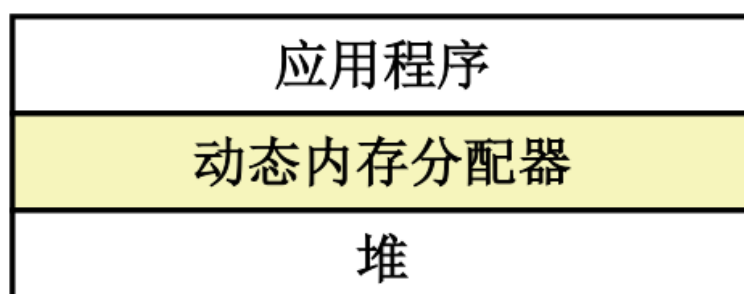
图 7-6 创建的进程在地址段中表示



## 7.8 缺页故障与缺页中断处理

DRAM 缓存不命中称为缺页，即虚拟内存中的字不在物理内存中。缺页导致页面出错，产生缺页异常。。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，在此例中就是存放在 PP3 中的 VP4 。如果 VP4 已经被修改了，那么内核就会将它复制回磁盘。无论哪种情况，内核都会修改 VP4 的页表条目，反映出 VP 4 不再缓存在主存中这一事实。

## 7.9 动态存储分配管理



在程序运行时程序员使用如上图描述的动态内存分配器给引用程序分配内存，动态内存分配器的维护着一个进程的虚拟内存（堆）。分配出来的动态的内存存在地址段中按照下图的位置给堆栈分配内存空间。这个称为堆。

由于编写 hello 程序的是 C 语言，而 C 语言的内存分配器为显式分配器，所以这里讨论显示分配器的内存管理方式。

动态内存堆分配策略原理就是在一个事先定义好大小的内存块中进行管理，其内存分配的策略是采用最快合适（ First Fit）方式，只要找到一个比所请求的内存大的空闲块，就从中切割出合适的块，并把剩余的部分返回到动态内存堆中。内存的释放时，重新将申请到的内存返回堆中。

其优点就是内存浪费小，比较简单，适合用于小内存的管理，其缺点就是如果频繁的动态分配和释放，可能会造成严重的内存碎片，如果在碎片情况严重的话，可能会导致内存分配不成功。

这其中也有个问题，就是内存合并问题。因为内存堆的管理通常为链表的形式进行管理。可选择将小的链表节点（较小的内存）进行合并。

malloc(size\_t size)每次声明内存空间都要保证至少分配 size\_t 大小的内存，保证双字对齐，每次必须从空闲块中分配空间，在申请空间的时候要记得将空闲的空

间碎片合并，这样可以尽量减少浪费。

## 7.10 本章小结

本章通过对 `hello` 在储存结构，高速缓存，虚拟内存涉及到的方面进行了详细的探索，理解了系统是如何将数据在存储器层次结构中上上下下移动的，那么就可以编写自己的应用程序，使得它们的数据项存储在层次结构中较高的地方，在那里 CPU 能更快地访问到它们，复习了与内存管理相关的重要的概念和方法。加深了对动态内存分配的认识和了解。

## 第 8 章 hello 的 I/O 管理

### 8.1 Linux 的 I/O 设备管理方法

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

### 8.2 简述 Unix I/O 接口及其函数

Unix I/O 接口：

- 1.打开文件。一个应用程序通过要求内核打开相应的文件，来宣告它想要访问一个 I/O 设备。内核返回一个小的非负整数，叫做描述符，它在后续对此文件的所有操作中标识这个文件。内核记录有关这个打开文件的所有信息。应用程序只需记住这个描述符。
- 2.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（描述符为 0）、标准输出（描述符为 1）和标准错误（描述符为 2）。头文件 `<unistd.h>` 定义了常量 `STDIN_FILENO`、`STDOUT_FILENO` 和 `STDERR_FILENO`，它们可用来代替显式的描述符值。
- 3.改变当前的文件位置。对于每个打开的文件，内核保持着一个文件位置 `k`，初始为 0。这个文件位置是从文件开头起始的字节偏移量。应用程序能够通过执行 `seek` 操作，显式地设置文件的当前位置为 `K`。
- 4.读写文件。一个读操作就是从文件复制 `n>0` 个字节到内存，从当前文件位置 `k` 开始，然后将 `k` 增加到 `k+n`。给定一个大小为 `m` 字节的文件，当 `k~m` 时执行读操作会触发一个称为 `end-of-file(EOF)` 的条件，应用程序能检测到这个条件。在文件结尾处并没有明确的“EOF 符号”。类似地，写操作就是从内存复制 `n>0` 个字节到一个文件，从当前文件位置 `k` 开始，然后更新 `k`。
- 5.关闭文件。当应用完成了对文件的访问之后，它就通知内核关闭这个文件。作为响应，内核释放文件打开时创建的数据结构，并将这个描述符恢复到可用的描述符

池中。无论一个进程因为何种原因终止时，内核都会关闭所有打开的文件并释放它们的内存资源。

Unix I/O 函数：

1.进程是通过调用 `open` 函数来打开一个已存在的文件或者创建一个新文件的：

```
int open(char *filename, int flags, mode_t mode);
```

`open` 函数将 `filename` 转换为一个文件描述符，并且返回描述符数字。返回的描述符总是在进程中当前没有打开的最小描述符。`flags` 参数指明了进程打算如何访问这个文件，`mode` 参数指定了新文件的访问权限位。

返回：若成功则为新文件描述符，若出错为-1。

2.进程通过调用 `close` 函数关闭一个打开的文件。

```
int close(int fd);
```

返回：若成功则为 0，若出错则为-1。

3.应用程序是通过分别调用 `read` 和 `write` 函数来执行输入和输出的。

```
ssize_t read(int fd, void *buf, size_t n);
```

`read` 函数从描述符为 `fd` 的当前文件位置复制最多 `n` 个字节到内存位置 `buf`。返回值-1 表示一个错误，而返回值 0 表示 EOF。否则，返回值表示的是实际传送的字节数量。

返回：若成功则为读的字节数，若 EOF 则为 0，若出错为-1。

```
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 `fd` 的当前文件位置。图 10-3 展示了一个程序使用 `read` 和 `write` 调用一次一个字节地从标准输入复制到标准输出。

返回：若成功则为写的字节数，若出错则为-1。

## 8.3 printf 的实现分析

printf 实现代码如下：

```
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;
}
```

va\_list 的定义：

```
typedef char *va_list
```

这说明它是一个字符指针。

其中的：(char\*)&fmt + 4) 表示的是...中的第一个参数。

再来看 vsprintf 函数：

```
int vsprintf(char *buf, const char *fmt, va_list args)
{
    char* p;
    char tmp[256];
    va_list p_next_arg = args;

    for (p=buf; *fmt; fmt++)
    {
        if (*fmt != '%')
        {
            *p++ = *fmt;
            continue;
        }
        fmt++;
        switch (*fmt)
        {
            case 'x':
                itoa(tmp, *((int*)p_next_arg));
                strcpy(p, tmp);
                p_next_arg += 4;
                p += strlen(tmp);
                break;
            case 's':
                break;
            default:
                break;
        }
    }
    return (p - buf);
}
```

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

字符显示驱动子程序：从 ASCII 到字模库到显示 vram（存储每一个点的 RGB 颜色信息）。

显示芯片按照刷新频率逐行读取 vram，并通过信号线向液晶显示器传输每一个点（RGB 分量）。

## 8.4 getchar 的实现分析

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

getchar 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

本章节讲述了一下 linux 的 I/O 设备管理机制，了解了开、关、读、写、转移文件的接口及相关函数，简单分析了 printf 和 getchar 函数的实现方法以及操作过程

（第 8 章 1 分）

## 结论

Hello.c 出生在一个普通家庭，介于低级语言和高级语言之间的 C 语言。但是为了能够在系统上运行，需要将他的每条语句都转化为一系列的低级机器语言指令，然后将这些指令按照可执行目标程序的格式打包，并以二进制磁盘文件的形式进行存放。hello 程序在运行时，看上去是独占使用处理器、主存和 I/O 设备，这是通过进程的概念来实现的。

进程是操作系统对于一个正在运行的程序的一种抽象。

在一个系统上可以同时运行多个进程，每个进程都像是在独占的使用硬件。

并发运行，是指一个进程和另一个进程的指令是交错执行的。

以上场景中有两个并发的进程：shell 和 hello。操作系统保持跟踪进程运行所需的所有信息，这种状态，就是上下文。

```
linux> ./hello
```

这个过程发生了什么呢？

首先，我们从键盘上输入 ./hello 到 shell 命令窗口，书中把这个输入命令的窗口程序称为“外壳”，外壳是一个命令行解释器；输入每个字符都被逐一读取到 CPU 的寄存器中，然后被保存到存储器中，当我们敲回车时，证明命令结束，此

时外壳就回去加载命令中指定的 `hello` 可执行目标文件，将文件的内容读取到主存储器中，此时代码的位置转移到主存中。

然后，等程序加载完毕，CPU 处理器就开始发挥作用，执行 `hello` 中的 `main` 程序中的机器指令。将要输出的 `"hello1170301026wjq\n"` 字符串从主存读取到 CPU 的寄存器中；

最后，再把寄存器中的字符串，复制到显示设备中，最终显示到屏幕上；

运行，暂停，`Ctrl-z`，`Ctrl-c` 用字符塞满缓冲区、用 `kill` 残忍的一刀斩去，`hello` 的一生就这样结束了。

伴随着 `hello` 的一生，我也复习了整个计算机系统相关知识，真是感谢为伟大的 CSAPP 献身的 `hello.c` 呢：)

## 附件

hello.i(hello.c 预处理之后的程序文本)

hello.s(hello.i 编译成汇编语言之后的程序文本)

hello3.c(用于测试 sleepsecs 的值的程序文本)

hello3(hello3.c 生成的可执行二进制文件)

hello.o(hello.s 生成的二进制文件)

hello\_o.s(hello.o 反汇编的结果)

hello(可执行的 hello 二进制文件)

hello\_.s(可执行文件 hello，直接用 objdump 反编译之后的汇编代码)

hello.elf(可执行文件 hello 的 elf 表)



## 参考文献

- [1] <https://www.cnblogs.com/pianist/p/3315801.html>
- [2] <https://blog.csdn.net/ZCShouCSDN/article/details/80282907>
- [3] <https://blog.csdn.net/gdj0001/article/details/80135196>