

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>計算機科學與技術</u>
學 號	<u>1170301007</u>
班 級	<u>1703010</u>
學 生	<u>沈子鳴</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

本文以 Linux 环境下，简单的 C 程序 `hello.c` 从 `program` 到 `process` 的过程为线索，介绍了 GCC 编译系统的四个工作环节。又以运行 `hello` 程序为核心，展开介绍了程序的进程管理，相关数据的存储管理，和 I/O 管理。以《深入理解计算机系统》（第三版）为主要参考文献，结合实际操作，图文并茂地介绍了一些具体的概念、原理和实践。对读者理解计算机中程序的一生有一定的帮助作用。

**关键词：**编译系统；进程；信号与异常；内存管理；Linux I/O 管理

（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	- 4 -
1.3 中间结果	- 5 -
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	- 9 -
<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	- 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	- 23 -
4.5 本章小结	- 25 -
<b>第 5 章 链接</b>	<b>- 27 -</b>
5.1 链接的概念与作用	- 27 -
5.2 在 UBUNTU 下链接的命令	- 27 -
5.3 可执行目标文件 HELLO 的格式	- 27 -
5.4 HELLO 的虚拟地址空间	- 31 -
5.5 链接的重定位过程分析	- 32 -
5.6 HELLO 的执行流程	- 35 -
5.7 HELLO 的动态链接分析	- 36 -
5.8 本章小结	- 39 -
<b>第 6 章 HELLO 进程管理</b>	<b>- 40 -</b>
6.1 进程的概念与作用	- 40 -

6.2 简述壳 SHELL-BASH 的作用与处理流程.....	- 40 -
6.3 HELLO 的 FORK 进程创建过程 .....	- 41 -
6.4 HELLO 的 EXECVE 过程 .....	- 41 -
6.5 HELLO 的进程执行.....	- 42 -
6.6 HELLO 的异常与信号处理 .....	- 44 -
6.7 本章小结 .....	- 47 -
<b>第 7 章 HELLO 的存储管理.....</b>	<b>- 49 -</b>
7.1 HELLO 的存储器地址空间 .....	- 49 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理.....	- 49 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理 .....	- 51 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换.....	- 53 -
7.5 三级 CACHE 支持下的物理内存访问 .....	- 55 -
7.6 HELLO 进程 FORK 时的内存映射 .....	- 57 -
7.7 HELLO 进程 EXECVE 时的内存映射 .....	- 58 -
7.8 缺页故障与缺页中断处理.....	- 59 -
7.9 动态存储分配管理 .....	- 60 -
7.10 本章小结 .....	- 63 -
<b>第 8 章 HELLO 的 IO 管理 .....</b>	<b>- 64 -</b>
8.1 LINUX 的 IO 设备管理方法 .....	- 64 -
8.2 简述 UNIX IO 接口及其函数 .....	- 64 -
8.3 PRINTF 的实现分析.....	- 65 -
8.4 GETCHAR 的实现分析.....	- 67 -
8.5 本章小结 .....	- 68 -
<b>结论 .....</b>	<b>- 68 -</b>
<b>附件 .....</b>	<b>- 70 -</b>
<b>参考文献.....</b>	<b>- 71 -</b>

## 第 1 章 概述

### 1.1 Hello 简介

Hello 的 P2P (from program to process) 过程，就是从编程到处理执行的过程。我们在编译器中编写高级语言代码，比如常用的 Visual Studio, codeblocks 等，用编译器构建 (Build) 成功后，就生成了一个 .exe 可执行程序文件。这只是 P2P 的表面过程。

P2P 的深层过程，可以在 Linux 下编译代码中得到体现。假设我们在 Linux 环境下，用 GCC 编译器驱动程序来编译 hello.c，有四个阶段：预处理阶段、编译阶段、汇编阶段、链接阶段，这四个阶段的主角分别是：预处理器 (cpp)、编译器 (ccl)、汇编器 (as) 和链接器 (ld)。

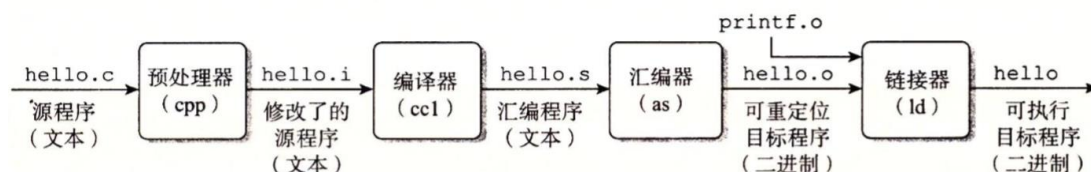


图 1.1 编译系统

经历了这四个阶段，hello.c 就成为了 hello.out (Linux 环境)，P2P 的过程就结束了。

Hello 的 O2O (from zero-0 to zero-0) 过程，就是“赤条条地来，赤条条地走”。从在编辑工具里编辑好 hello 程序之后，经过曲折的 P2P 之路，由 hello.c 经历 hello.i, hello.s, hello.o 和其他.o 一起链接成为 hello.out。接着，我们在进程中调用 hello，实现了 hello 的逻辑控制流的意义所在。Execve 函数运行起 hello，在虚拟内存空间中给 hello 分配空间，又有地址翻译把 hello 的虚拟地址翻译成物理地址，硬件根据物理地址在主存中取址，形成软硬件结合的运行体系。Hello 运行结束后，进程终止，内存回收，内核把关于 hello 的一切数据全部抹去，这样 hello “挥一挥手，不带走一片云彩”。

不仅仅是针对 hello，正是有了 P2P 和 O2O，计算机中的程序才得以正常运行，有条不紊，不会出错。

### 1.2 环境与工具

硬件环境：Intel Core i5 7200U, 2.50GHz, 4GB RAM, 256GB SSD

软件环境：Windows 10 家庭中文版, Ubuntu 18.04.1 LTS (VMware)

工具：codeblocks, gedit, gcc, objdump, readelf 等

### 1.3 中间结果

hello.c	源程序 C 语言代码
hello.i	hello.c 预处理后的文本文件
hello.s	编译后的汇编语言文本文件
hello.o	汇编后的可重定位目标文件
hello	链接后的可执行目标文件
helloo_obj.s	hello.o 利用 objdump 工具的反汇编代码
hello_obj.s	hello 利用 objdump 工具的反汇编代码
其他中间结果	readelf 和 objdump 的其他调试代码，反映在了标准输出中，没有保存到本地文件

### 1.4 本章小结

本章介绍了 hello 的 P2P (from program to process) 和 020 (from zero-0 to zero-0) 过程，是全文的一个简单概括。本章还列举了环境工具和中间结果，以供读者总览。

(第 1 章 0.5 分)

## 第 2 章 预处理

### 2.1 预处理的概念与作用

预处理是源文件到目标文件转化的第一环节。GCC 编译器驱动程序把源程序文件翻译成可执行目标文件，首先要经历预处理阶段。

在预处理阶段，预处理器（cpp）根据以字符#开头的命令，修改原始的 C 程序。这些命令通常是.c 文件开头的一些以#开头的命令。

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
```

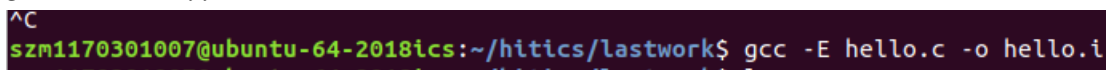
图 2.1 几行 C 中的预处理命令

这些命令告诉预处理器读取相应的文件（如头文件），并把这些文件直接插入程序文本中。结果就得到了另一个 C 程序，通常是以.i 作为文件扩展名。

### 2.2 在 Ubuntu 下预处理的命令

GCC 编译器驱动程序的预处理器的预处理命令格式如下：

```
gcc -E xx.c -o yy.i
```



```
^C
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$ gcc -E hello.c -o hello.i
```

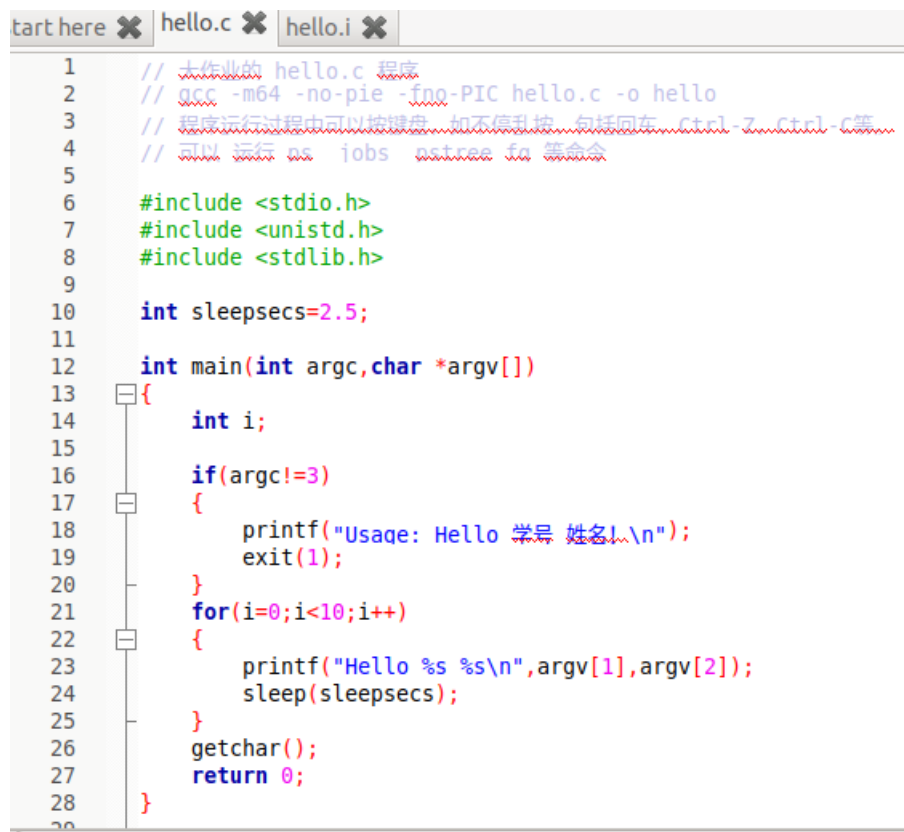
图 2.2 Linux 下 gcc 预处理

其中，-E 选项指只预处理，不编译。xx.c 是源文件。-o 选项指定了预处理输出文件的文件名，这个文件名就是 yy.i，通常我们建议输出文件名与源文件同名，而不同后缀。

Gcc 选项还可以加-C 选项，表示预处理时不删除注释信息，配合-E 选项使用。

### 2.3 Hello 的预处理结果解析

预处理过程其实是把源程序.c 文件，扩展成为一个新的文本模式的.i 文件，扩展内容就是预处理的插入内容。



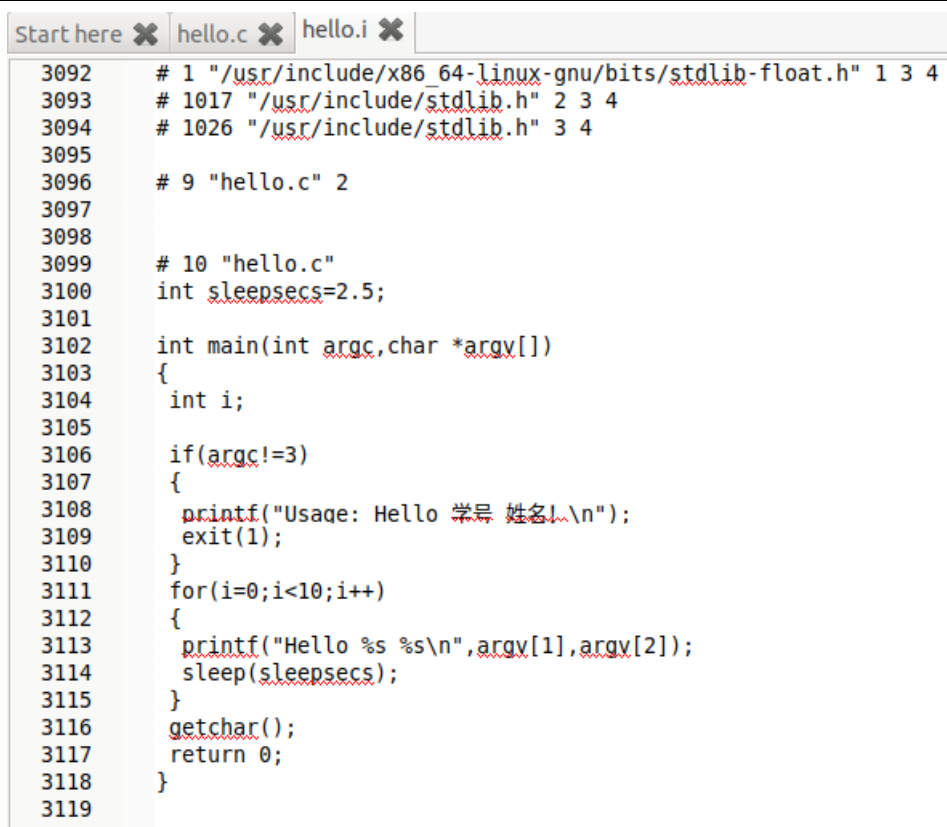
```
1 // 大作业的 hello.c 程序
2 // gcc -m64 -no-pie -fno-PIC hello.c -o hello
3 // 程序运行过程中可以按键盘, 如不停机按, 包括回车, Ctrl-Z, Ctrl-C等
4 // 可以运行 ns jobs nstree fa 等命令
5
6 #include <stdio.h>
7 #include <unistd.h>
8 #include <stdlib.h>
9
10 int sleepsecs=2.5;
11
12 int main(int argc, char *argv[])
13 {
14     int i;
15
16     if(argc!=3)
17     {
18         printf("Usage: Hello 学号 姓名!\n");
19         exit(1);
20     }
21     for(i=0;i<10;i++)
22     {
23         printf("Hello %s %s\n", argv[1], argv[2]);
24         sleep(sleepsecs);
25     }
26     getchar();
27     return 0;
28 }
```

图 2.3 hello.c



```
art here x hello.c x hello.i x
1 # 1 "hello.c"
2 # 1 "<built-in>"
3 # 1 "<command-line>"
4 # 31 "<command-line>"
5 # 1 "/usr/include/stdc-predef.h" 1 3 4
6 # 32 "<command-line>" 2
7 # 1 "hello.c"
8
9
10
11
12
13 # 1 "/usr/include/stdio.h" 1 3 4
14 # 27 "/usr/include/stdio.h" 3 4
15 # 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
16 # 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
17 # 1 "/usr/include/features.h" 1 3 4
18 # 424 "/usr/include/features.h" 3 4
19 # 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
20 # 427 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
21 # 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
22 # 428 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
23 # 1 "/usr/include/x86_64-linux-gnu/bits/long-double.h" 1 3 4
24 # 429 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
25 # 425 "/usr/include/features.h" 2 3 4
26 # 448 "/usr/include/features.h" 3 4
27 # 1 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 1 3 4
28 # 10 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 3 4
29 # 1 "/usr/include/x86_64-linux-gnu/gnu/stubs_64.h" 1 3 4
```

图 2.4.1 hello.i (一部分)



```
3092 # 1 "/usr/include/x86_64-linux-gnu/bits/stdc++-float.h" 1 3 4
3093 # 1017 "/usr/include/stdc++lib.h" 2 3 4
3094 # 1026 "/usr/include/stdc++lib.h" 3 4
3095
3096 # 9 "hello.c" 2
3097
3098
3099 # 10 "hello.c"
3100 int sleepsecs=2.5;
3101
3102 int main(int argc, char *argv[])
3103 {
3104     int i;
3105
3106     if(argc!=3)
3107     {
3108         printf("Usage: Hello 学号 姓名\n");
3109         exit(1);
3110     }
3111     for(i=0; i<10; i++)
3112     {
3113         printf("Hello %s %s\n", argv[1], argv[2]);
3114         sleep(sleepsecs);
3115     }
3116     getchar();
3117     return 0;
3118 }
3119
```

图 2.4.2 hello.i (仍保留 hello.c 的部分)

由图 2.4 可以看出, hello.c 经过预处理后, 源代码前面插入了大量的其它代码, 这是由预处理器完成的。并且源代码的预处理命令已经被移除掉了, 事实上, 这些预处理命令被翻译成了插入的代码。

## 2.4 本章小结

预处理阶段是 hello 的“P2P”之路的第一步, 从此, hello 的代码被逐步解析成为机器能够理解的代码。

在 Linux 中, 预处理阶段由 GCC 编译器驱动程序的预处理器 (cpp) 执行, 转换过程由.c 文件变成.i 文件。

(第 2 章 0.5 分)

## 第 3 章 编译

### 3.1 编译的概念与作用

编译是源文件到目标文件转化的第二环节。预处理器（`cpp`）对 `.c` 文件预处理为 `.i` 文件后，由编译器（`cc1`）继续对 `.i` 文件在编译阶段中加工。

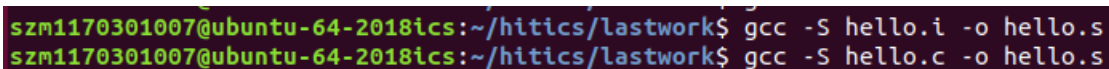
在编译阶段，编译器（`cc1`）将 `.i` 文件翻译成另一种文本格式的 `.s` 文件，它包含一个汇编语言程序。汇编语言是一种低级机器语言指令，为不同高级语言的不同编译器提供了通用的输出语言，是高级程序语言和机器语言之间的桥梁。

### 3.2 在 Ubuntu 下编译的命令

GCC 编译器驱动程序的编译器的编译命令格式如下：

`gcc -S xx.i -o yy.s`

或者 `gcc -S xx.c -o yy.s`



```
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$ gcc -S hello.i -o hello.s
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$ gcc -S hello.c -o hello.s
```

图 3.1 Linux 下 gcc 编译

其中，`-S` 选项指只编译，不汇编和链接。`-o` 选项指定了输出的文件。至于操作对象，既可以是源文件 `.c`，也可以是预处理后的 `.i` 文件，如果是对 `.c` 文件操作，则 `gcc` 默认对 `.c` 文件先进行预处理，再进行编译。

另外，`-O` 选项给编译过程提出了指定的优化级别。`Gcc` 编译器有几种优化模式：`-O0`, `-O`, `-O1`, `-O2`, `-Os`, `-O3`，优化级别越高优化效果越好，但编译时间会变长。

### 3.3 Hello 的编译结果解析

`hello.c` 中包含了很多数据类型和指令操作，下面逐一解析。

#### 3.3.1 数据

##### 1) 常量



字节对齐, 并把 `sleepsecs` 声明为 `@object` 类型, 大小为 4 字节。接着, 又给 `sleepsecs` 赋值为 `long` 类型的值 2 (源文件中, `int sleepsecs = 2.5`, `long` 和 `int` 类型大小相同, 编译器将 `int` 转成了 `long` 存储, 并且把赋值语句右侧的 2.5 直接舍成 2)。

至于局部变量 `i`, 是在 `main` 函数中声明的。对于编译器来说, 局部变量要么保存在寄存器中, 要么保存在栈空间中。从编译结果看出, 此处的变量 `i` 被保存到了栈空间中, 还可以找到循环的大致位置。

```

.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

```

局部变量 `i` 赋初值 0

循环步长为 1, `i++`

循环终止条件

图 3.4 `hello.s` 中的局部变量 `i`

### 3.3.2 操作

#### 1) 赋值

`hello.c` 中一共有两处赋值, 一个是全局变量 `sleepsecs` 的初始化赋值, 一个是局部变量 `i` 的初始化赋值。全局变量的赋值是在 `.s` 的头部就已经做好了, 值在 `.data` 节中。至于局部变量的赋值, 用的是 `MOV` 指令。

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

局部变量 `i` 保存在栈中, 具体位置在 `-4(%rbp)`。这句操作指令的含义是把立即数 0, 传送到栈上的 `-4(%rbp)` 中, 即对 `i` 赋值。需要注意的是 64 位系统有几种不同的 `MOV` 指令:

指令	效果	描述
<code>MOV S, D</code>	$D \leftarrow S$	传送
<code>movb</code>		传送字节
<code>movw</code>		传送字 (2 字节)

movl		传送双字（4 字节）
movq		传送四字（8 字节）
movabsq I, R		传送绝对的四字

表 3.1 64 位的数据传送指令

拿 `hello.c` 中的 `i` 赋值为例，这里用的是 `movl`，也就是传送 4 字节的数据，而 `i` 是 `int` 类型的。

## 2) 类型转换

`hello.c` 中只有一个隐式的类型转换，就是在全局变量的初始化赋值中。这里的类型转换是把浮点数转换为整型，这种情况是基本类型转换中比较复杂的情形。因为对于浮点数来说，向整数舍入，通常是找到最接近浮点数的整数来作为舍入结果。但是，如果有两个整数距离这个浮点数相同，就需要决策了。一种比较好的方式是向偶数舍入，这种方式决定把 2.5 舍入为 2，而把 3.5 舍入为 4。在统计学中，这种舍入方法是简单方法中，较优的一个。因此我们也看到了，在 `.data` 节，`sleepsecs` 的值被声明为 2。

简单数据类型的转换还有很多种。通常来说，低精度向高精度的舍入不太可能造成数据丢失，但高精度向低精度的转换可能会损失精度，甚至出现意想不到的效果。比如 `int` 类型的 -1 其实是 `unsigned int` 类型的最大值，负数转换为 `unsigned` 类型时，会有这些麻烦。

## 3) 算术操作和逻辑操作

`hello.c` 中只有一处算术操作，就是循环语句中的循环变量 `i++`，汇编语言中的算术操作使用一系列指令集来完成的。

指令	效果	描述
leaq S, D	$D \leftarrow \&S$	加载有效地址
INC D	$D \leftarrow D+1$	加 1
DEC D	$D \leftarrow D-1$	减 1
NEG D	$D \leftarrow -D$	取负
NOT D	$D \leftarrow \sim D$	取补
ADD S, D	$D \leftarrow D+S$	加
SUB S, D	$D \leftarrow D-S$	减
IMUL S, D	$D \leftarrow D*S$	乘
XOR S, D	$D \leftarrow D^S$	异或
OR S, D	$D \leftarrow D S$	或
AND S, D	$D \leftarrow D\&S$	与
SAL k, D	$D \leftarrow D \ll k$	左移
SHL k, D	$D \leftarrow D \ll k$	左移（等同于 SAL）
SAR k, D	$D \leftarrow D \gg_A k$	算术右移
SHR k, D	$D \leftarrow D \gg_L k$	逻辑右移

表 3.2 简单的整数算术操作

`hello.c` 中无逻辑操作。

## 4) 关系操作

hello.c 中的关系操作在判断语句中, 包括循环终止条件的判断。而这些关系操作也仅限于大小比较。汇编语言中的比较不仅限于大小的比较。

指令	基于	描述
<code>cmpb S<sub>1</sub>, S<sub>2</sub></code>	$S_2 - S_1$	比较字节
<code>cmpw S<sub>1</sub>, S<sub>2</sub></code>		比较字
<code>cmpl S<sub>1</sub>, S<sub>2</sub></code>		比较双字
<code>cmpq S<sub>1</sub>, S<sub>2</sub></code>		比较四字
<code>testb S<sub>1</sub>, S<sub>2</sub></code>	$S_1 \& S_2$	测试字节
<code>testw S<sub>1</sub>, S<sub>2</sub></code>		测试字
<code>testl S<sub>1</sub>, S<sub>2</sub></code>		测试双字
<code>testq S<sub>1</sub>, S<sub>2</sub></code>		测试四字

表 3.3 比较和测试指令

**CMP** 指令根据两个操作数之差来设置条件码。除了只设置条件码而不更新目的寄存器之外, **CMP** 指令和 **SUB** 指令行为是一样的。

**TEST** 指令行为与 **AND** 指令一样, 除了它们只设置条件码而不改变目的寄存器的值。

以 hello.c 中的第一个条件判断为例:

`argc!=3`

它的汇编代码是:

`cmpl $3, -20(%rbp)`

`je .L2`

这是指, 用 `argc` 和 3 作差, 根据结果设置条件码, 再根据条件码, 判断是否跳转到 `L2` 处。

## 5) 控制转移

hello.c 中的控制转移也是伴随着条件判断出现的。比如说, 如果 `argc!=3`, 那么就执行 `if` 块的语句。如果 `i<10` 的话, 那么就继续 `i++`, 并转而执行新的迭代。汇编语言中的控制转移有两种, 一种是通过访问条件码, 用 **jump** 指令配合条件控制来执行控制转移。另一种是用条件传送来实现条件分支。不加优化编译的 `hello.s` 中无条件传送指令。

指令	跳转条件	描述
<code>jmp Label</code>	1	直接跳转
<code>jmp *Operand</code>	1	间接跳转
<code>je Label</code>	ZF	相等/零
<code>jne Label</code>	~ZF	不相等/非零
<code>js Label</code>	SF	负数
<code>jns Label</code>	~SF	非负数
<code>jg Label</code>	~(SF^OF)&~ZF	大于 (有符号)

<code>jge Label</code>	$\sim(\text{SF} \wedge \text{OF})$	大于或等于（有符号）
<code>jl Label</code>	$\text{SF} \wedge \text{OF}$	小于（有符号）
<code>jle Label</code>	$(\text{SF} \wedge \text{OF})   \text{ZF}$	小于或等于（有符号）
<code>ja Label</code>	$\sim \text{CF} \& \sim \text{ZF}$	超过（无符号）
<code>jae Label</code>	$\sim \text{CF}$	超过或相等（无符号）
<code>jbe Label</code>	$\text{CF}$	低于（无符号）
<code>jbe Label</code>	$\text{CF}   \text{ZF}$	低于或相等（无符号）

表 3.4 jump 指令。当跳转条件满足时，这些指令会跳转到一条带标号的目的地

对 hello.c 中的循环终止条件判断解析：

```

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

```

图 3.5 hello.s 中循环终止条件判断

## 6) 函数操作以及参数

hello.c 中共有五个函数调用，除了 main 函数外，还有 printf、exit、sleep 和 getchar 函数。其中，只有 getchar 没有参数。

函数作为一种过程，假设过程 P 调用过程 Q，有这样的机制：

传递控制。在进入过程 Q 的时候，程序计数器必须被设置为 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 提供一个或者多个参数，Q 必须能够向 P 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，又必须释放掉这些空间。这些空间往往是运行时栈。



以 main 函数为例，作为分析。

```

main:
.LFB5:
    .cfi_startproc
    pushq   %rbp                栈指针
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16         记录栈底
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $32, %rsp          申请运行时栈空间
    movl    %edi, -20(%rbp)
    movq    %rsi, -32(%rbp)    参数
    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   0, %eax            函数返回值在%rax中
                                leave等价于释放栈空间

```

图 3.6 hello.s 中 main 函数解析

## 7) 字符串和数组

hello.c 中也是有字符串和数组的。事实上这里面的字符串作为 printf 的参数，在主函数中是以如下形式声明的：

```

.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 3.7 hello.s 中的字符串声明

而数组 `char *argv[]` 是 `main` 的参数，在栈空间中，`argv` 的首地址被存放在 `-32(%rbp)` 的位置，调用数组元素时，根据栈指针加上偏移量来寻址。

```
.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 3.8 `hello.s` 中 `argv` 的元素调用寻址

### 3.4 本章小结

编译是“hello” P2P 之路的第二步，结果是把源程序代码转化成了汇编语言代码。汇编语言是高级程序语言和机器语言之间的桥梁。编译系统的工作就是把源程序代码，不断地朝着机器能够识别的代码的方向转化。

(第 3 章 2 分)

## 第 4 章 汇编

### 4.1 汇编的概念与作用

汇编是源文件到目标文件转化的第三环节。经过了预处理和编译，源文件已经变成了由汇编语言编写的.s 文件。接下来，由汇编器（as）将.s 文件翻译成为机器语言指令，把这些指令打包成一种叫做可重定位目标程序的格式，并把结果保存在目标文件.o 中。如果我们在文本编辑器中打开.o 文件，只能看到一堆乱码。

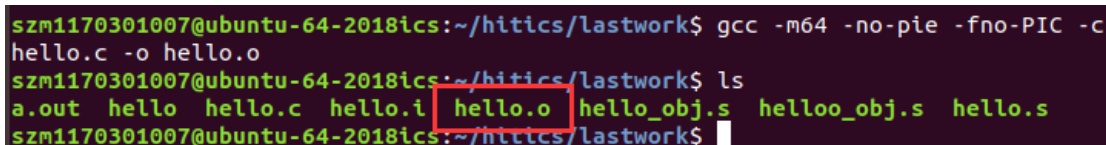
### 4.2 在 Ubuntu 下汇编的命令

GCC 的汇编指令格式如下：

```
gcc -c xx.c -o yy.o
```

应作业要求，我们加上如下选项：

```
gcc -m64 -no-pie -fno-PIC -c hello.c -o hello.o
```



```
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$ gcc -m64 -no-pie -fno-PIC -c  
hello.c -o hello.o  
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$ ls  
a.out hello hello.c hello.i hello.o hello_obj.s hello_obj.s hello.s  
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$
```

图 4.1 Linux 下 gcc 汇编

其中，-c 选项是只进行预处理、编译和汇编，而不进行链接。xx.c 是操作对象，也可以是.i 或.s 文件。-o 指定了输出文件，输出文件是.o 格式文件。建议输出文件名与操作对象同名不同后缀。

### 4.3 可重定位目标 elf 格式

ELF 头
.text
.rodata
.data
.bss
.symtab
.rel.text
.rel.data
.debug
.line

.strtab
节头部表

表 4.1 典型的 ELF 可重定位目标文件

要分析 hello.o 的 ELF 格式，参考表 4.1，用

`readelf -a hello.o`

命令可以将 hello.o 的 ELF 信息打印到标准输出上。

```
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$ readelf -a 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

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 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
```

图 4.2 hello.o 的 ELF 信息概览

### 4.3.1 ELF 头

```
ELF 头:
Magic:      7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
类别:      ELF64
数据:      2 补码, 小端序 (little endian)
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 4.3 hello.o 的 ELF 头

ELF 头以一个 16 字节的序列开始, 这个序列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。可以得知 ELF 头的大小为 64 字节, 目标文件类型为 ELF64, 系统架构为 X86-64, 节头部表的文件偏移 1152 字节等。

#### 4.3.2 节头部表

节头:						
[号]	名称 大小	类型 全体大小	地址 旗标	链接	信息	偏移量 对齐
[ 0]		NULL	0000000000000000			00000000
	0000000000000000	0000000000000000		0	0	0
[ 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
	00000000000000180	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 4.4 hello.o 的节头部表

节头部表描述了不同节的位置和大小，目标文件中每个节都有一个固定大小的条目。

#### 4.3.3 重定位节

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
00000000000016	000500000000a	R_X86_64_32	0000000000000000	.rodata	+ 0
0000000000001b	000b000000002	R_X86_64_PC32	0000000000000000	puts	- 4
00000000000025	000c000000002	R_X86_64_PC32	0000000000000000	exit	- 4
0000000000004c	000500000000a	R_X86_64_32	0000000000000000	.rodata	+ 1e
00000000000056	000d000000002	R_X86_64_PC32	0000000000000000	printf	- 4
0000000000005c	0009000000002	R_X86_64_PC32	0000000000000000	sleepsecs	- 4
00000000000063	000e000000002	R_X86_64_PC32	0000000000000000	sleep	- 4
00000000000072	000f000000002	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
00000000000020	0002000000002	R_X86_64_PC32	0000000000000000	.text	+ 0

图 4.5 hello.o 的重定位节

hello.o 的重定位节中一共有 8 个条目，给出了偏移量、信息、类型、符号值、符号名称+加数等信息。

R\_X86\_64\_PC32 是和 R\_X86\_64\_32 相对应的，比较常见的重定位类型。前者是重定位一个使用 32 位 PC 相对地址的引用，而后者是重定位一个使用 32 位绝对地址的引用。我在会汇编时，最开始是从之前生成的.s 文件汇编过来的，这样的话图 4.5 会不一样，偏移量和 .rodata 的加数会有些许不同，另外还会遇到 R\_X86\_64\_PLT32 类型，Linux Kernels 网站上给出如下解释：

*This Linux kernel change "x86: Treat R\_X86\_64\_PLT32 as R\_X86\_64\_PC32" is included in the Linux 3.18.100 release. This change is authored by H.J. Lu <hjl.tools[at]gmail.com> on Wed Feb 7 14:20:09 2018 -0800.*

重定位 PC 相对引用的重定位算法为：

$refaddr = ADDR(s) + r.offset;$

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

假设算法运行时，链接器为每个节（用 ADDR(s)表示）和每个符号都选择了运行时地址（用 ADDR(r.symbol)）表示。拿 .rodata 的重定位为例，它的重定位地址为 refptr。则应先计算引用的运行时地址  $refaddr = ADDR(s) + r.offset$ ，.rodata 的 offset 为 0x16，ADDR(s)是由链接器确定的。然后，更新引用， $refptr = (unsigned)(ADDR(r.symbol) + r.addend - refaddr)$ ，.rodata 的 ADDR(r.symbol)也是由链接器确定的，addend 查表可知为+0，refaddr 已经算出来了，所以，.rodata 的重定位地址我们就可以算出来了。

#### 4.3.4 符号表

Symbol table '.syntab' contains 16 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	hello.c
2:	0000000000000000	0	SECTION	LOCAL	DEFAULT	1	
3:	0000000000000000	0	SECTION	LOCAL	DEFAULT	3	
4:	0000000000000000	0	SECTION	LOCAL	DEFAULT	4	
5:	0000000000000000	0	SECTION	LOCAL	DEFAULT	5	
6:	0000000000000000	0	SECTION	LOCAL	DEFAULT	7	
7:	0000000000000000	0	SECTION	LOCAL	DEFAULT	8	
8:	0000000000000000	0	SECTION	LOCAL	DEFAULT	6	
9:	0000000000000000	4	OBJECT	GLOBAL	DEFAULT	3	sleepsecs
10:	0000000000000000	125	FUNC	GLOBAL	DEFAULT	1	main
11:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	puts
12:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	exit
13:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	printf
14:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	sleep
15:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	getchar

图 4.6 hello.o 的符号表

符号表是由汇编器构造的，使用编译器输出到汇编语言.s 文件中的符号。 .syntab 节中包含 ELF 符号表。这张符号表包含一个条目的数组，每个条目的格式有如下数据结构：

```
typedef struct{
    int name;
    char type:4,
```



```
    binding:4;  
    char reserved;  
    short section;  
    long value;  
    long size;  
}Elf64_Symbol;
```

其中，`name` 是字符串表中的字节偏移，指向符号的以 `null` 结尾的字符串的名字。`value` 是符号的地址。对于可重定位的模块来说，`value` 是距定义目标的节的起始位置的偏移。`size` 是目标的大小（以字节为单位）。`type` 通常要么是数据，要么是函数。`binding` 字段表示符号是本地的还是全局的。

#### 4.3.5 其它节

The image shows a terminal window with a dark purple background. It contains two lines of text in a light-colored font. The first line is "There are no section groups in this file." and the second line is "There is no dynamic section in this file." The text is centered in the window.

图 4.7 `hello.o` 中的一些不存在的节

有些其它节在我们的 `hello.o` 是不存在的。

#### 4.4 `Hello.o` 的结果解析

说明机器语言的构成，与汇编语言的映射关系。特别是机器语言中的操作数与汇编语言不一致，特别是分支转移函数调用等。



```
.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 4.8 hello.s 的部分代码

```

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             callq   1f <main+0x1f>
1a: e8 00 00 00 00                   1b: R_X86_64_PC32 puts-0x4
                                mov     $0x1,%edi
1f: bf 01 00 00 00                   callq   29 <main+0x29>
24: e8 00 00 00 00                   25: R_X86_64_PC32 exit-0x4
                                movl    $0x0,-0x4(%rbp)
29: c7 45 fc 00 00 00 00           jmp     6b <main+0x6b>
30: eb 39
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)

```

图 4.9 hello.o 的反汇编代码

hello.o 是由 hello.s 汇编得到的，而 hello.o 的反汇编却和 hello.s 不一样了，两者不同之处，主要是汇编器对 hello.s 做的手脚。汇编之前，hello.s 只是把扩展后的源代码翻译成了汇编代码，像是一张使用说明书。而汇编之后，hello.o 获得了重定位信息，符号表等 ELF 格式信息，像是实战视频教程。

汇编之后，hello.o 比 hello.s 更加具体，比如说函数调用的地址，从函数名称变成了主函数首地址加上偏移量，而条件跳转也从跳转到段名称变成了跳转到指定偏移地址。还有对全局变量的引用，之前是.LC0(%rip)，而现在是\$0x0，至于这个全局变量的地址到底在哪，这些信息都保存在重定位信息里了。还有一些细节，立即数的表示由十进制变成了十六进制。

## 4.5 本章小结

可以说，汇编之后的 hello 变得更加丰满，更加难懂，也更贴近机器了。hello.o 作为可重定位的目标文件，即将和其它目标文件一起链接成最后的可执行目标文件。

件。

到此，我们的 `hello` 已经拥有了汇编语言解释，ELF 格式的诸多信息，即将达成最后的可执行文件了。

**(第 4 章 1 分)**

## 第 5 章 链接

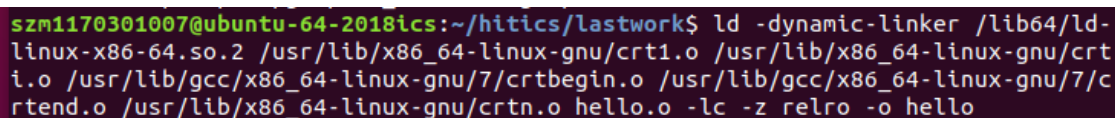
### 5.1 链接的概念与作用

链接过程是 `hello` 成为可执行目标文件的最后一步。在经历了预处理、编译和汇编之后生成的 `.o` 文件，只需要再经过链接器 (`ld`) 和其它可重定位目标文件链接，就可以生成最终的可执行目标程序了。

### 5.2 在 Ubuntu 下链接的命令

按照要求，我们将如下文件与 `hello.o` 链接生成可执行目标文件 `hello`。参考命令如下：

```
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  
/usr/lib/gcc/x86_64-linux-gnu/7/crtend.o /usr/lib/x86_64-linux-gnu/crtn.o hello.o -lc -z  
relro -o hello
```



```
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$ ld -dynamic-linker /lib64/ld-  
linux-x86-64.so.2 /usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crt  
i.o /usr/lib/gcc/x86_64-linux-gnu/7/crtbegin.o /usr/lib/gcc/x86_64-linux-gnu/7/c  
rtend.o /usr/lib/x86_64-linux-gnu/crtn.o hello.o -lc -z relro -o hello
```

图 5.1 Linux 下 gcc 链接

链接的文件都是 64 位库中的一些可重定位目标文件。

### 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。

```
readelf -a hello
```

命令可以将 `hello` 文件的 ELF 格式信息输出到标准输出上。

```
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:        EXEC (可执行文件)
系统架构:    Advanced Micro Devices X86-64
版本:        0x1
入口点地址:  0x400500
程序头起点:  64 (bytes into file)
Start of section headers: 6512 (bytes into file)
标志:        0x0
本头的大小:  64 (字节)
程序头大小:  56 (字节)
Number of program headers: 8
节头大小:    64 (字节)
节头数量:    28
字符串表索引节头: 27
```

图 5.2 hello 的 ELF 头

节头:						
[号]	名称	类型	地址	链接	偏移量	
	大小	全体大小	旗标		信息	对齐
[ 0]	0000000000000000	NULL	0000000000000000	0	0	0
[ 1]	.interp	PROGBITS	000000000400200	0	0	00000200
	000000000000001c	0000000000000000	A	0	0	1
[ 2]	.note.ABI-tag	NOTE	00000000040021c	0	0	0000021c
	0000000000000020	0000000000000000	A	0	0	4
[ 3]	.hash	HASH	000000000400240	0	0	00000240
	0000000000000034	0000000000000004	A	5	0	8
[ 4]	.gnu.hash	GNU_HASH	000000000400278	0	0	00000278
	000000000000001c	0000000000000000	A	5	0	8
[ 5]	.dynsym	DYNSYM	000000000400298	0	0	00000298
	00000000000000c0	0000000000000018	A	6	1	8
[ 6]	.dynstr	STRTAB	000000000400358	0	0	00000358
	0000000000000057	0000000000000000	A	0	0	1
[ 7]	.gnu.version	VERSYM	0000000004003b0	0	0	000003b0
	0000000000000010	0000000000000002	A	5	0	2
[ 8]	.gnu.version_r	VERNEED	0000000004003c0	0	0	000003c0
	0000000000000020	0000000000000000	A	6	1	8
[ 9]	.rela.dyn	RELA	0000000004003e0	0	0	000003e0
	0000000000000030	0000000000000018	A	5	0	8
[10]	.rela.plt	RELA	000000000400410	0	0	00000410
	0000000000000078	0000000000000018	AI	5	21	8
[11]	.init	PROGBITS	000000000400488	0	0	00000488
	0000000000000017	0000000000000000	AX	0	0	4
[12]	.plt	PROGBITS	0000000004004a0	0	0	000004a0
	0000000000000060	0000000000000010	AX	0	0	16
[13]	.text	PROGBITS	000000000400500	0	0	00000500
	00000000000001e2	0000000000000000	AX	0	0	16
[14]	.fini	PROGBITS	0000000004006e4	0	0	000006e4
	0000000000000009	0000000000000000	AX	0	0	4
[15]	.rodata	PROGBITS	0000000004006f0	0	0	000006f0
	000000000000002f	0000000000000000	A	0	0	4
[16]	.eh_frame	PROGBITS	000000000400720	0	0	00000720
	00000000000000fc	0000000000000000	A	0	0	8
[17]	.init_array	INIT_ARRAY	000000000600e00	0	0	00000e00
	0000000000000008	0000000000000008	WA	0	0	8
[18]	.fini_array	FINI_ARRAY	000000000600e08	0	0	00000e08
	0000000000000008	0000000000000008	WA	0	0	8
[19]	.dynamic	DYNAMIC	000000000600e10	0	0	00000e10
	00000000000001e0	0000000000000010	WA	6	0	8
[20]	.got	PROGBITS	000000000600ff0	0	0	00000ff0

图 5.3 hello 的节头部表（一部分）

这里有用的信息包括各段的大小，起始地址等信息。

程序头:

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 0x0000000000000081c	0x0000000000400000 0x0000000000000081c	0x0000000000400000 R E 0x200000
LOAD	0x00000000000000e00 0x0000000000000254	0x0000000000600e00 0x0000000000000258	0x0000000000600e00 RW 0x200000
DYNAMIC	0x00000000000000e10 0x000000000000001e0	0x0000000000600e10 0x000000000000001e0	0x0000000000600e10 RW 0x8
NOTE	0x0000000000000021c 0x00000000000000020	0x000000000040021c 0x00000000000000020	0x000000000040021c R 0x4
GNU_STACK	0x00000000000000000 0x00000000000000000	0x00000000000000000 0x00000000000000000	0x00000000000000000 RW 0x10
GNU_RELRO	0x00000000000000e00 0x0000000000000200	0x0000000000600e00 0x0000000000000200	0x0000000000600e00 R 0x1

图 5.4 hello 的程序头

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 hello 的动态符号表

Symbol table '.syntab' contains 64 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000400200	0	SECTION	LOCAL	DEFAULT	1	
2:	000000000040021c	0	SECTION	LOCAL	DEFAULT	2	
3:	0000000000400240	0	SECTION	LOCAL	DEFAULT	3	
4:	0000000000400278	0	SECTION	LOCAL	DEFAULT	4	
5:	0000000000400298	0	SECTION	LOCAL	DEFAULT	5	
6:	0000000000400358	0	SECTION	LOCAL	DEFAULT	6	
7:	00000000004003b0	0	SECTION	LOCAL	DEFAULT	7	
8:	00000000004003c0	0	SECTION	LOCAL	DEFAULT	8	
9:	00000000004003e0	0	SECTION	LOCAL	DEFAULT	9	
10:	0000000000400410	0	SECTION	LOCAL	DEFAULT	10	
11:	0000000000400488	0	SECTION	LOCAL	DEFAULT	11	
12:	00000000004004a0	0	SECTION	LOCAL	DEFAULT	12	
13:	0000000000400500	0	SECTION	LOCAL	DEFAULT	13	
14:	00000000004006e4	0	SECTION	LOCAL	DEFAULT	14	
15:	00000000004006f0	0	SECTION	LOCAL	DEFAULT	15	
16:	0000000000400720	0	SECTION	LOCAL	DEFAULT	16	
17:	0000000000600e00	0	SECTION	LOCAL	DEFAULT	17	
18:	0000000000600e08	0	SECTION	LOCAL	DEFAULT	18	
19:	0000000000600e10	0	SECTION	LOCAL	DEFAULT	19	
20:	0000000000600ff0	0	SECTION	LOCAL	DEFAULT	20	
21:	0000000000601000	0	SECTION	LOCAL	DEFAULT	21	
22:	0000000000601040	0	SECTION	LOCAL	DEFAULT	22	
23:	0000000000601054	0	SECTION	LOCAL	DEFAULT	23	
24:	0000000000000000	0	SECTION	LOCAL	DEFAULT	24	
25:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	crtstuff.c
26:	0000000000400540	0	FUNC	LOCAL	DEFAULT	13	deregister_tm_clones
27:	0000000000400570	0	FUNC	LOCAL	DEFAULT	13	register_tm_clones
28:	00000000004005b0	0	FUNC	LOCAL	DEFAULT	13	__do_global_dtors_aux
29:	0000000000601054	1	OBJECT	LOCAL	DEFAULT	23	completed.7696
30:	0000000000600e08	0	OBJECT	LOCAL	DEFAULT	18	__do_global_dtors_aux_fin
31:	00000000004005e0	0	FUNC	LOCAL	DEFAULT	13	frame_dummy
32:	0000000000600e00	0	OBJECT	LOCAL	DEFAULT	17	__frame_dummy_init_array_
33:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	crtstuff.c
34:	00000000004007a0	0	OBJECT	LOCAL	DEFAULT	16	__FRAME_END__
35:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	hello.c
36:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	
37:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	17	__init_array_end

图 5.6 hello 的符号表（包括动态符号表）（局部）

## 5.4 hello 的虚拟地址空间



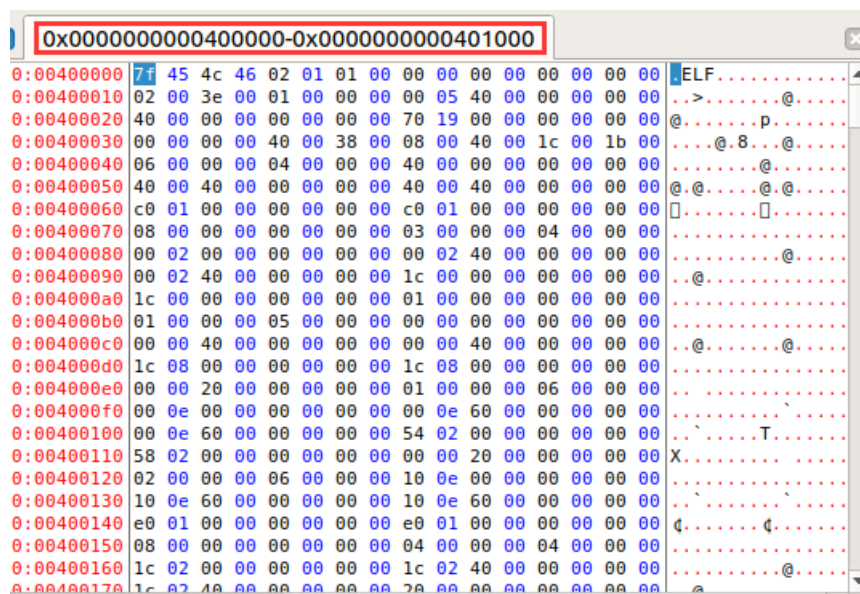


图 5.7 hello 的段虚拟空间信息（局部）

正如图 5.3 所示，hello 的各段的虚拟地址空间从 0x400000 到 0x401000，可以在 edb 的 data dump 中查看。

## 5.5 链接的重定位过程分析

## 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 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

```

图 5.8 hello.o 的反汇编代码 (局部)

```

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

```

图 5.9 hello 的反汇编代码（局部，main 函数）

乍一看，hello 和 hello.o 的区别有，在链接之前，各段的地址仅仅是一个偏移量，而非“地址”，链接之后，各段有了实质的虚拟地址，每一条指令也有对应的虚拟地址。之前提到过了，从 hello.s 到 hello.o 有一个区别，是有些段寻址时，从.L0(%rip)变成了\$0x0，而后者的具体地址放在了重定位信息里。现在，hello.o 经过链接中的重定位后，基本上所有未确定的信息都有了确定的地址，比如说，在加载全局变量字符串时（printf 的参数字符串），hello 中给出的指令是：

```
mov     $0x4006f4,%edi
```

可以说，每一个条目都找到了虚拟内存地址。调用的函数，也都是 call 函数的虚拟地址。

这个重定位的过程，需要 PC 相对引用重定位算法好好解析一下。下面就以 hello.o 中的调用 exit 函数为例，其指令是这样的：

```
24: e8 00 00 00 00      callq   29 <main+0x29>
```

首先，对任意一条指令的重定位，我们需要知道一些信息。

```

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)

```

图 5.10 段 main 的运行时地址

```

00000000004004e0 <exit@plt>:
4004e0: ff 25 4a 0b 20 00   jmpq    *0x200b4a(%rip)        # 601030 <exit@GLIBC_2.2.5>
4004e6: 68 03 00 00 00     pushq   $0x3
4004eb: e9 b0 ff ff ff     jmpq     4004a0 <.plt>

```

图 5.11 exit 函数的运行时地址

```

重定位节 '.rel.text' at offset 0x310 contains 8 entries:
偏移量      信息      类型      符号值      符号名称 + 加数
00000000000016 000500000000a R_X86_64_32 0000000000000000 .rodata + 0
0000000000001b 000b000000002 R_X86_64_PC32 0000000000000000 puts - 4
00000000000025 000c000000002 R_X86_64_PC32 0000000000000000 exit - 4
0000000000004c 000500000000a R_X86_64_32 0000000000000000 .rodata + 1e
00000000000056 000d000000002 R_X86_64_PC32 0000000000000000 printf - 4
0000000000005c 0009000000002 R_X86_64_PC32 0000000000000000 sleepsecs - 4
00000000000063 000e000000002 R_X86_64_PC32 0000000000000000 sleep - 4
00000000000072 000f000000002 R_X86_64_PC32 0000000000000000 getchar - 4

```

图 5.12 exit 函数的重定位信息

按照重定位 PC 相对引用算法，先计算引用的运行时地址：

$$refaddr = 0x4005e7 + 0x25 = 0x40060c$$

再更新该引用：

$$*refptr = 0x4004e0 - 0x4 - 0x40060c = 0xffffffff(-0x130)$$

```

40060b: e8 d0 fe ff ff     callq   4004e0 <exit@plt>

```

图 5.13 调用 exit 指令的重定位结果

其它的指令重定位，如果也是 R\_X86\_64\_PC32(R\_X86\_64\_PLT32)的，重定位结果也是按这个算法算。如果是绝对引用的话，有如下算法：

$$*refptr = (\text{unsigned})(ADDR(r.symbol) + r.addend)$$

具体示例不再列举了，计算过程和相对引用相似，更简单。

## 5.6 hello 的执行流程

子程序名	程序地址	简述
------	------	----

ld-2.27.so!_dl_start	0x7fefaff21ea0	开始加载
ld-2.27.so!_dl_init	0x7fefaff30630	
libc-2.27.so!__libc_start_main	0x7fefafb50ab0	
libc-2.27.so!__cxa_atexit	0x7fefafb72430	
hello!__libc_csu_init	0x400670	
libc-2.27.so!_setjmp	0x7fefafb6dc10	
hello!main	0x4005e7	hello.c 的主函数
hello!puts@plt	0x4004b0	hello.c 中调用
hello!exit@plt	0x4004e0	hello.c 中调用
hello!printf@plt	0x4004c0	hello.c 中调用
hello!sleep@plt	0x4004f0	hello.c 中调用
hello!getchar@plt	0x4004d0	hello.c 中调用
libc-2.27.so!exit		

表 5.1 加载 hello 到程序终止中经历的一些子程序

因为 hello 程序有一个分支调用了 exit 函数，故程序在那里就终止了，如果在 hello 中进入了另一条分支，事实上是会运行到 libc-2.27.so!exit 终止的。而且，每次加载 hello 的虚拟地址也是不同的。但 hello 中的 main 和调用的一些函数的地址是经过重定位之后，固定下来的虚拟地址。

## 5.7 Hello 的动态链接分析

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。

hello 在调用 .so 共享库函数时，会涉及到动态链接。现代系统在处理共享库在地址空间中的分配的时候，采用了**位置无关代码(PIC)**方式。位置无关代码指，编译共享模块的代码段，是把它们加载到内存的任何位置而无需链接器修改。用户对 GCC 使用 -fpic 选项指示 GNU 编译系统生成 PIC 代码。共享库的编译必须总是使用该选项。

PIC 代码引用包括数据引用和函数调用。对数据引用有一个事实，就是代码段中任何指令和数据段中任何变量之间的距离是一个运行时常量，与代码段和数据段的绝对内存位置是无关的。在编译器想要生成对 PIC 全局变量引用时，在数据段开始的地方创建了**全局偏移量表(GOT)**。

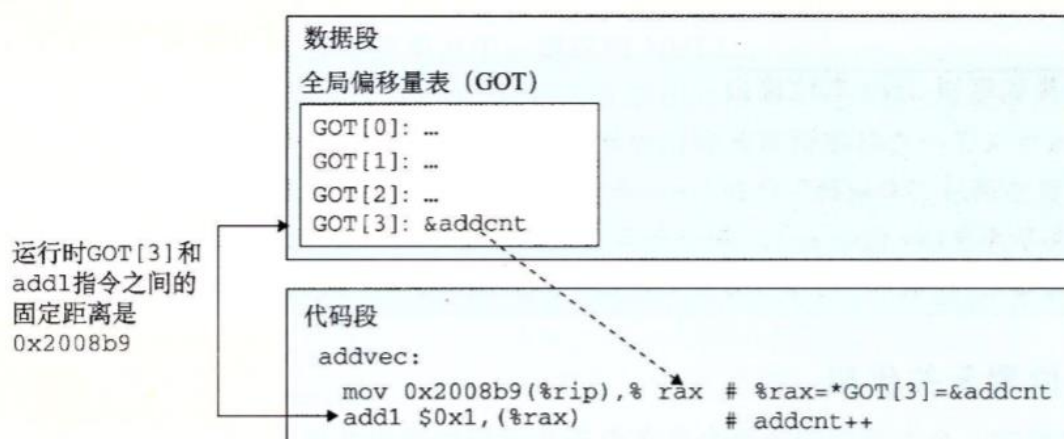


图 5.14 用 GOT 引用全局变量。libvector.so 的 addvec 例程通过 libvector.so 的 GOT 间接引用了 addcnt

以图 5.14 为例说明。在执行例程 addvec 时, mov 语句实现了 addcnt 的引用。%rip 存放的是下一条指令地址, 即 addl 指令地址, 运行时代码段中的 addl 距离数据段中的 GOT[3] 的距离是常量 0x2008b9, 故用基址偏移寻址找到 GOT[3], 而 GOT[3] 中存放的是 addcnt 的真实地址, 这样 mov 就实现了把 &addcnt 存入 %rax 的操作。

对 PIC 函数调用, 有一个现象叫**延迟绑定**, 即将过程地址的绑定推迟到第一次调用该过程时。把函数地址的解析推迟到它实际被调用的地方, 能避免动态链接器在加载时进行成百上千个其实并不需要的重定位。第一次调用过程的运行时开销很大, 但是其后的每次调用都只会花费一条指令和一个间接的内存引用。

延迟绑定是通过 GOT 和**过程链接表(PLT)**实现的。GOT 是数据段的一部分, PLT 是代码段的一部分。

对一个函数的调用, 首先程序进入函数的 PLT 条目, PLT 条目下的第一条 PLT 指令通过函数的 GOT 条目跳转, 因为首次调用 GOT 条目指向的是 PLT 条目下的第二条指令, 所以此时已经运行到了 PLT 的第二条指令。

PLT 的第二条指令通常是把函数的 ID 压栈, 随后跳转到 PLT[0]。PLT[0] 是一个特殊条目, 它跳转到动态链接器中。PLT[0] 又借助 GOT[1] 间接地把动态链接器的一个参数压入栈中, 这可能是动态链接器在解析函数地址时会使用的信息。这时, 动态链接器会通过两个栈中的条目来确定函数的运行时位置, 用这个地址重写它的 GOT 条目, 再把控制传给函数例程。

此时就是第二次调用函数了, 控制会转入到它的 PLT 条目, 然后再借它的 GOT 条目跳转, 经过动态链接器的更新, GOT 条目中存放的已经是函数的地址, 所以这次调用就进入到函数例程了。



```

Disassembly of section .plt:

00000000004004a0 <.plt>:
4004a0: ff 35 62 0b 20 00    pushq 0x200b62(%rip)    # 601008 <_GLOBAL_OFFSET_TABLE_+0x8>
4004a6: ff 25 64 0b 20 00    jmpq *0x200b64(%rip)    # 601010 <_GLOBAL_OFFSET_TABLE_+0x10>
4004ac: 0f 1f 40 00          nopl 0x0(%rax)

00000000004004b0 <puts@plt>:
4004b0: ff 25 62 0b 20 00    jmpq *0x200b62(%rip)    # 601018 <puts@GLIBC_2.2.5>
4004b6: 68 00 00 00 00 00    pushq $0x0
4004bb: e9 e0 ff ff ff      jmpq 4004a0 <.plt>

00000000004004c0 <printf@plt>:
4004c0: ff 25 5a 0b 20 00    jmpq *0x200b5a(%rip)    # 601020 <printf@GLIBC_2.2.5>
4004c6: 68 01 00 00 00 00    pushq $0x1
4004cb: e9 d0 ff ff ff      jmpq 4004a0 <.plt>

00000000004004d0 <getchar@plt>:
4004d0: ff 25 52 0b 20 00    jmpq *0x200b52(%rip)    # 601028 <getchar@GLIBC_2.2.5>
4004d6: 68 02 00 00 00 00    pushq $0x2
4004db: e9 c0 ff ff ff      jmpq 4004a0 <.plt>

00000000004004e0 <exit@plt>:
4004e0: ff 25 4a 0b 20 00    jmpq *0x200b4a(%rip)    # 601030 <exit@GLIBC_2.2.5>
4004e6: 68 03 00 00 00 00    pushq $0x3
4004eb: e9 b0 ff ff ff      jmpq 4004a0 <.plt>

00000000004004f0 <sleep@plt>:
4004f0: ff 25 42 0b 20 00    jmpq *0x200b42(%rip)    # 601038 <sleep@GLIBC_2.2.5>
4004f6: 68 04 00 00 00 00    pushq $0x4
4004fb: e9 a0 ff ff ff      jmpq 4004a0 <.plt>

```

图 5.15 hello 的 .plt 段

下面对 hello 的 dl\_start 函数的 GOT 更新环节做解析。

```

[19] .dynamic          DYNAMIC          0000000000600e10 00000e10
      00000000000001e0 000000000000010 WA      6      0      8
[20] .got             PROGBITS         0000000000600ff0 00000ff0
      0000000000000010 000000000000008 WA      0      0      8
[21] .got.plt         PROGBITS         0000000000601000 00001000
      0000000000000040 000000000000008 WA      0      0      8
[22] .data            PROGBITS         0000000000601040 00001040

```

图 5.15 hello 的 GOT 表地址

先找到 hello 的 GOT 表，之后在 edb 中对 GOT 表做跟踪分析。

```

00000000:00600fe0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000000:00600ff0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000000:00601000 10 0e 60 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000000:00601010 00 00 00 00 00 00 00 00 b6 04 40 00 00 00 00 00 .....
00000000:00601020 c6 04 40 00 00 00 00 00 d6 04 40 00 00 00 00 00 .....
00000000:00601030 e6 04 40 00 00 00 00 00 f6 04 40 00 00 00 00 00 .....
00000000:00601040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

图 5.16 执行 dl\_start 前的 GOT

```

00000000:00600fe0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000000:00600ff0 b0 9a cd 8e 35 7f 00 00 00 00 00 00 00 00 00 00 .....
00000000:00601000 10 0e 60 00 00 00 00 00 70 21 2d 8f 35 7f 00 00 .....
00000000:00601010 50 07 0c 8f 35 7f 00 00 b6 04 40 00 00 00 00 00 .....
00000000:00601020 c6 04 40 00 00 00 00 00 d6 04 40 00 00 00 00 00 .....
00000000:00601030 e6 04 40 00 00 00 00 00 f6 04 40 00 00 00 00 00 .....
00000000:00601040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

图 5.17 执行 dl\_start 后的 GOT

对 GOT[2]的地址 0x7f358f0c0750 跟踪。

00007f35:8f0c0750	53	pushq %rbx
00007f35:8f0c0751	48 89 e3	movq %rsp, %rbx
00007f35:8f0c0754	48 83 e4 c0	andq \$0xffffffffc0, %rsp
00007f35:8f0c0758	48 2b 25 a9 00 21 00	subq 0x2100a9(%rip), %rsp
00007f35:8f0c075f	48 89 04 24	movq %rax, (%rsp)
00007f35:8f0c0763	48 89 4c 24 08	movq %rcx, 8(%rsp)
00007f35:8f0c0768	48 89 54 24 10	movq %rdx, 0x10(%rsp)
00007f35:8f0c076d	48 89 74 24 18	movq %rsi, 0x18(%rsp)
00007f35:8f0c0772	48 89 7c 24 20	movq %rdi, 0x20(%rsp)

图 5.18 GOT[2], 动态链接器

00000000:00600fe0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000000:00600ff0	b0 9a cd 8e 35 7f 00 00 00 00 00 00 00 00 00 00	.....5.....
00000000:00601000	10 0e 60 00 00 00 00 00 70 21 2d 8f 35 7f 00 00	.....p!-.5....
00000000:00601010	50 07 0c 8f 35 7f 00 00 b6 04 40 00 00 00 00 00	P. .5...@.....
00000000:00601020	c6 04 40 00 00 00 00 00 d6 04 40 00 00 00 00 00	.....@.....
00000000:00601030	e6 04 40 00 00 00 00 00 f6 04 40 00 00 00 00 00	.....@.....
00000000:00601040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

图 5.19 GOT[3], GOT[4], GOT[5], GOT[6]

根据图 5.19 中的函数的 GOT 条目，配合图 5.15 的函数的 PLT 条目来看，实践证明 GOT 和 PLT 配合调用函数的机制是正确的。

## 5.8 本章小结

本章介绍了 hello 在真正成为 process 之前的最后一步——链接。简单跟踪了 hello 的链接过程，包括静态链接和动态链接。重点分析了重定位和动态链接中的 PIC 调用和引用。终于，心心念的 hello 的 P2P 之路，走完了。

(第 5 章 1 分)



## 第 6 章 hello 进程管理

### 6.1 进程的概念与作用

进程的经典定义就是一个执行中程序的实例。系统中的每个程序都运行在某个进程的上下文中。上下文是由程序正确运行所需的状态组成的。这个状态包括存放在内存中的程序的代码和数据，它的栈、通用目的寄存器的内容、程序计数器、环境变量以及打开文件描述符的集合。

进程给应用程序提供了关键的抽象，一个独立地逻辑控制流，提供一个假象，好像我们的程序独占地使用处理器，还提供了一个私有的地址空间，提供一个假象，好像我们的程序独占地使用内存系统。

### 6.2 简述壳 Shell-bash 的作用与处理流程

shell 是一个交互型的应用级程序，它代表用户运行其他程序。Shell 执行一系列的读/求值步骤，然后终止。读步骤读取来自用户的一个命令行。求值步骤解析命令行，并代表用户运行程序。

shell 的首要任务是调用 `parseline` 函数，这个函数解析了以空格分割的命令行参数，并构造最终会传递给 `execve` 的 `argv` 向量。第一个参数被假设为要么是一个内置的 shell 命令名，马上就会解释这个命令，要么是一个可执行目标文件，会在一个新的子进程的上下文中加载并运行这个文件。

如果最后一个参数是一个“&”字符，那么 `parseline` 返回 1，表示应该在后台执行该程序（shell 不会等待它完成）。否则，它返回 0，表示应该在前台执行这个程序（shell 回等待它完成）。

在解析了命令行之后，`eval` 函数调用 `builtin_command` 函数，该函数检查第一个命令行参数是否是一个内置的 shell 命令。如果是，它就立即解释这个命令，并返回值 1。否则返回 0。简单的 shell 只有一个内置命令——`quit` 命令，该命令会终止 shell。实际使用的 shell 会有大量的内置命令。

如果 `builtin_command` 返回 0，那么 shell 创建一个子进程，并在子进程中执行所请求的程序。如果用户要在后台运行该程序，那么 shell 返回到循环的顶部，等待下一个命令行。否则，shell 使用 `waitpid` 函数等待作业终止。当作业终止时，shell 就开始下一轮迭代。

## 6.3 Hello 的 fork 进程创建过程

父函数可以通过 `fork` 函数创建一个新的运行的子进程。函数声明如下：

```
pid_t fork(void);
```

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，包括代码和数据段、堆、共享库以及用户栈。子进程还获得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任何文件。父进程和新创建的子进程之间最大的差别在于它们有不同的 PID。

有几个需要注意的地方。`fork` 函数调用一次，返回两次。父进程调用一次 `fork`，但却有一次是返回到父进程，而另一次是返回到子进程的。父进程和子进程是并发运行的独立进程，内核可以以任意方式交替执行它们的逻辑控制流中的指令。父进程和子进程还具有相同但是独立的地址空间，从虚拟内存的角度看 `fork` 函数，子进程使用父进程的地址空间，但有写时复制的特性。父子进程还有共享的文件。

通常父进程用 `waitpid` 函数来等待子进程终止或停止。

```
pid_t waitpid(pid_t pid, int *statusp, int options);
```

在父进程调用 `fork` 后，到 `waitpid` 子进程终止或停止这段时间里，父进程执行的操作，和子进程的操作（如果没有其它复杂的操作的话），在时间顺序上是拓扑排序执行的。有可能，这段时间里父子进程的逻辑控制流指令交替执行。而父进程的 `waitpid` 后的指令，只能在子进程终止或停止后，`waitpid` 返回后才能执行。

## 6.4 Hello 的 execve 过程

`execve` 函数在当前进程的上下文中加载并运行一个新程序。函数声明如下：

```
int execve(const char *filename, const char *argv[], const char *envp[]);
```

`execve` 函数加载并运行可执行目标文件 `filename`，且带参数列表 `argv` 和环境变量列表 `envp`。只有当出现错误时，例如找不到 `filename`，`execve` 才会返回到调用程序。正常情况下，`execve` 调用一次，但从不返回。

在 `execve` 加载 `filename` 之后，调用启动代码。启动代码设置栈，并将控制传递给新程序的主函数，该主函数有如下形式的原型：

```
int main(int argc, char **argv, char *envp);
```

`main` 开始执行时，用户栈的组织结构由图 6.1 所示。

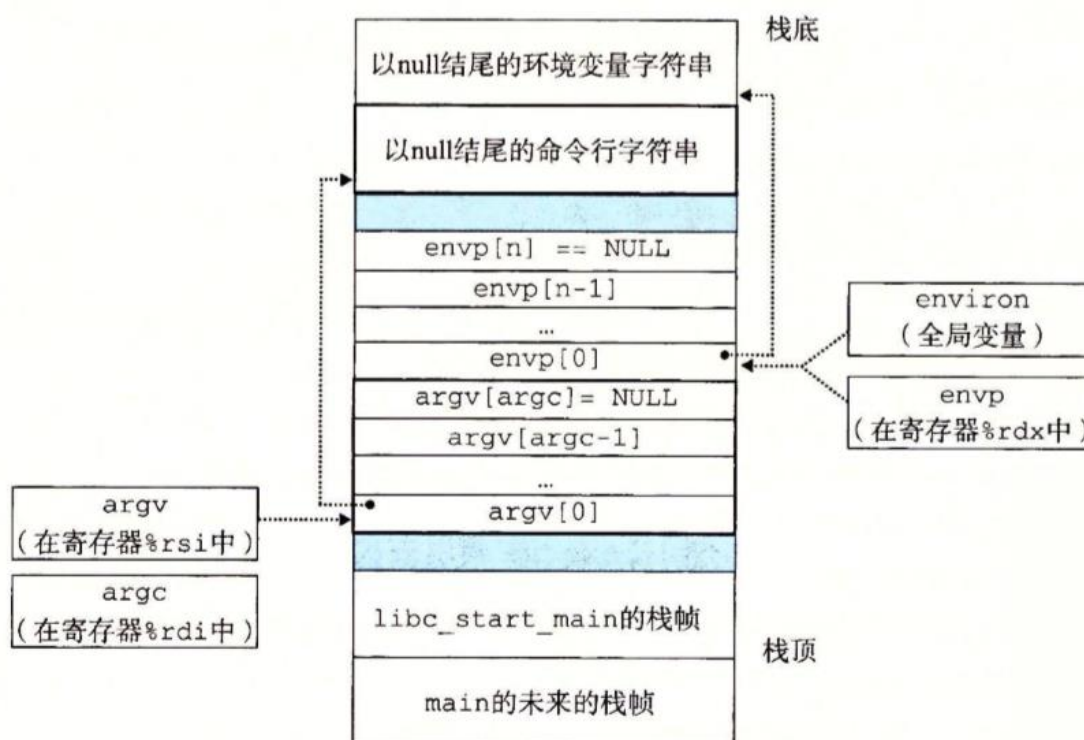


图 6.1 一个新程序开始时，用户栈的典型组织结构

从栈底（高地址）到栈顶（低地址），首先是参数和环境字符串。栈往上紧随其后的是以 `null` 结尾的指针数组，其中每个指针都指向栈中的一个环境变量字符串。全局变量 `environ` 指向这些之阵中的第一个 `envp[0]`。紧随环境变量数组之后的是以 `null` 结尾的 `argv[]` 数组，其中每个元素都指向栈中的一个参数字符串。在栈的顶部是系统启动函数 `libc_start_main` 的栈帧。

## 6.5 Hello 的进程执行

结合进程上下文信息、进程时间片，阐述进程调度的过程，用户态与核心态转换等等。

`hello` 在执行时，有自己的逻辑控制流。多个进程的逻辑控制流在时间上可以交错，表现为交替运行。每个进程执行它的流的一部分，然后被抢占（暂时挂起），然后轮到其它进程。

一个逻辑流的执行在时间上和另一个流重叠，成为并发流，这两个流并发地运行。一个进程执行它的控制流的一部分的每一时间段叫时间片。

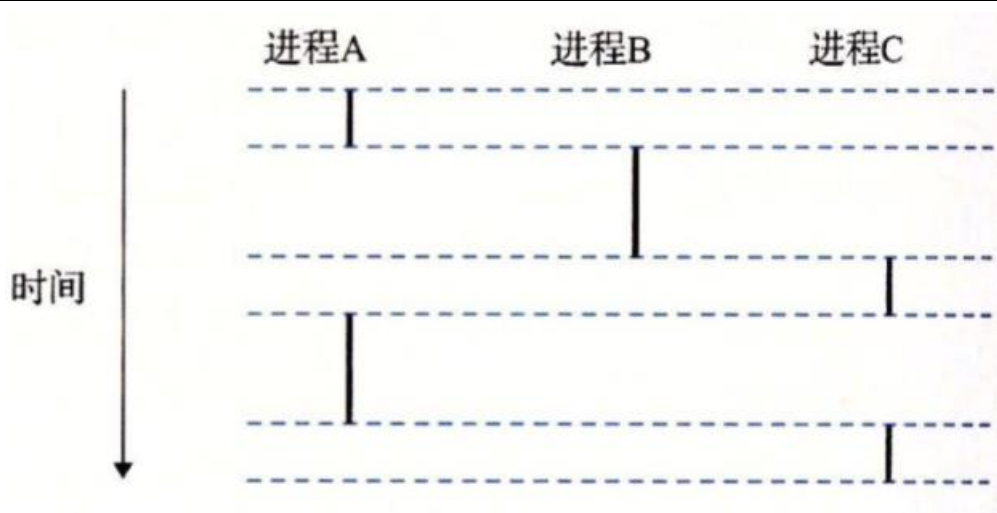


图 6.2 三个并发逻辑控制流

如图 6.2，每一个进程在时间段上的黑线段，都是一个时间片。同一时间上不会有两个进程同时占用处理器。在总体上看，并发控制流是在一段时间里是共用处理器的。

当 `hello` 执行到 `sleep` 函数时，会被挂起一段时间。挂起就是指进程被抢占，也就是 `hello` 会在逻辑控制流上出现一段断片。`sleep` 函数的声明如下：

```
unsigned int sleep(unsigned int secs);
```

这会使当前进程挂起 `secs` 秒，如果请求的时间量到了，`sleep` 返回 0，否则返回还剩下的要休眠的秒数。

正常情况下，`hello` 正常运行，直到调用函数 `sleep`，`hello` 进程会临时交出控制权。

进程控制权的交换涉及到**上下文切换**。操作系统内核使用一种成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。

在进程执行的某些时刻，内核可以决定抢占当前进程，并重新开始一个先前被抢占了的进程。这种决策就叫**调度**，是由内核中成为**调度器**的代码处理的。在内核调度了一个新的进程运行后，它就抢占当前进程，并使用一种称为上下文切换的机制来将控制转移到新的进程，上下文切换要做到：

- 1) 保存当前进程的上下文
- 2) 恢复某个先前被抢占的进程被保存的上下文
- 3) 将控制传递给这个新恢复的进程

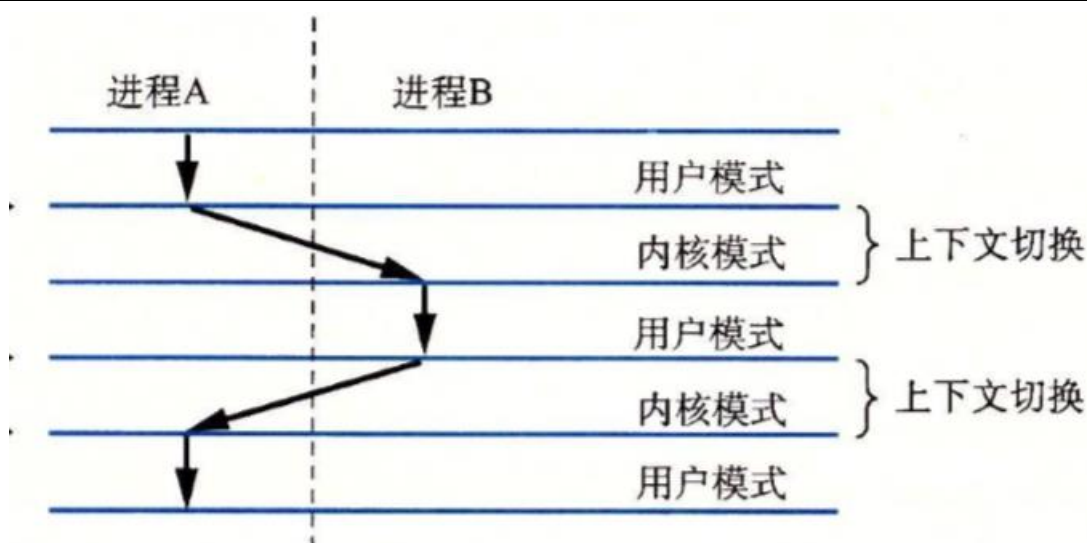


图 6.3 进程上下文切换

上下文切换有点像电影片场中的拍摄过程。一个场景是一个进程，场景的切换，要保存当前现场，把控制权交给另一个场景的导演，如果那个场景暂时结束了，就把控制再交回来，这时现场也是恢复了的。

再回到 `hello` 进程来，调用 `sleep` 后，内核中的调度器将 `hello` 进程挂起，然后进入到内核模式，由于 `hello` 调用 `sleep` 的这个过程没有显式地创建新的进程，所以，在 `hello` 被抢占了 `secs` 秒后，内核又会选择 `hello` 进程，恢复它被抢占时的上下文，并把控制交给它，这时，又回到了用户模式。

## 6.6 `hello` 的异常与信号处理

`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结果截屏，说明异常与信号的处理。

按照要求，对 `hello` 执行过程中，可能遭遇的操作进行分析。先分析一下 `hello` 程序的内容。

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int sleepsecs=2.5;

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

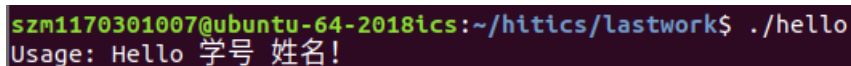
    if(argc!=3)
    {
        printf("Usage: Hello 学号 姓名! \n");
        exit(1);
    }
    for(i=0;i<10;i++)
    {
        printf("Hello %s %s\n",argv[1],argv[2]);
        sleep(sleepsecs);
    }
    getchar();
    return 0;
}
```

图 6.4 hello.c

argc 是执行 hello 时的参数个数，\*argv[] 是执行 hello 时，输入的参数数组，并且这时已经被解析过的参数字串。

首先，如果参数不为 3，那么会打印一条默认语句，并异常退出。如果参数是 3 个，那么会执行一个循环，每次循环会使 hello 进程休眠 2.5 秒，休眠后又会恢复 hello。而且循环里会输出一条格式字串，其中有输入的两个参数字串。循环结束后，有一个 getchar() 等待一个标准输入，然后就结束了。

### 6.6.1 正常运行



```
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$ ./hello
Usage: Hello 学号 姓名!
```

图 6.5 运行 hello，参数不为 3 个



```
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$ ./hello 3 个
Hello 3 个
Hello 3 个
Hello 3 个
Hello 3 个
Hello 3 个
Hello 3 个
Hello 3 个
Hello 3 个
Hello 3 个
Hello 3 个
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$
```

图 6.6 运行 hello，参数为 3 个

### 6.6.2 Ctrl+C 信号

```
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$ ./hello 3 个
Hello 3 个

Hello 3 个
Hello 3 个
^C
```

图 6.7 以 3 个参数运行 hello，运行时发送 CTRL+C 信号，进程直接终止

### 6.6.3 Ctrl+Z 信号

```
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$ ./hello 3 个
Hello 3 个
Hello 3 个
Hello 3 个

Hello 3 个
^Z
[1]+  已停止                  ./hello 3 个
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$ ps
  PID TTY          TIME CMD
  2014 pts/1        00:00:00 bash
  2226 pts/1        00:00:00 hello
  2227 pts/1        00:00:00 ps
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$ kill -9 2226
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$ ps
  PID TTY          TIME CMD
  2014 pts/1        00:00:00 bash
  2228 pts/1        00:00:00 ps
[1]+  已杀死                  ./hello 3 个
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$
```

图 6.8 以 3 个参数运行 hello，运行时发送 CTRL+Z 信号，进程停止，显式杀死进城后进程终止

### 6.6.4 标准输入

这是一个有趣的现象，如果输入 3 个参数，在 hello 循环时乱按键盘，向进程给出一些标准输入，那么在循环结束之后，getchar()会把缓冲区里的这些输入发送给 shell，等于 shell 接收到了一些不明所以的输入。

```

szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$ ./hello 3 个
Hello 3 个
340392938Hello 3 个
4
34208384-2384348-3Hello 3 个

3204923423-4382-
Hello 3 个
Hello 3 个
Hello 3 个
Hello 3 个
Hello 3 个
Hello 3 个
Hello 3 个
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$ 34208384-2384348-3
34208384-2384348-3: 未找到命令
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$ 3204923423-4382-
3204923423-4382-: 未找到命令
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$

```

图 6.9 以 3 个参数运行 hello，运行时给一些标准输入

以上四种情况，可以用 ps 和 kill 命令来测试，如图 6.8 中的 ctrl+z 信号，看似效果是和 ctrl+c 终止进程一样的，但实际上，如果用 bg 或 fg 命令是可以让 hello 继续运行的。kill 命令可以直接杀死进程。

```

szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$ ./hello 3 ge
Hello 3 ge
Hello 3 ge
Hello 3 ge
Hello 3 ge
^Z
[1]+  已停止                  ./hello 3 ge
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$ ps
  PID TTY          TIME CMD
  2014 pts/1        00:00:00 bash
  2285 pts/1        00:00:00 hello
  2286 pts/1        00:00:00 ps
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$ jobs
[1]+  已停止                  ./hello 3 ge
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$ fg %1
./hello 3 ge
Hello 3 ge
Hello 3 ge
Hello 3 ge
Hello 3 ge
Hello 3 ge
Hello 3 ge
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$ ps
  PID TTY          TIME CMD
  2014 pts/1        00:00:00 bash
  2288 pts/1        00:00:00 ps
szm1170301007@ubuntu-64-2018ics:~/hitics/lastwork$

```

图 6.10 hello 停止后，用 fg 命令恢复它的运行

## 6.7 本章小结



正如书中所说，进程是计算机科学中最深刻、最成功的概念之一。进程给应用程序提供的关键抽象，使得进程可以并发地执行。信号和异常的处理，使得并发执行的过程变得井然有序，如信号处理程序，一切都按照规矩运行。`shell-bash`的建立，给用户和进程之间提供了一个操作平台。总之，这部分内容既底层，又贴近用户。

**(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

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**。物理地址是在地址总线上，以电子形式存在的，使得数据总线可以访问主存的某个特定存储单元的内存地址。

在一个带虚拟内存的系统中，CPU 从一个有  $N=2^n$  个地址的地址空间中生成**虚拟地址**，这个地址空间成为虚拟地址空间。

地址空间是一个非负整数的有序集合，如果地址空间中的整数是连续的，那么我们说它是一个线性地址空间。**线性地址**就是线性地址空间中的地址。

在有地址变换功能的计算机中，访问指令给出的地址（操作数）叫**逻辑地址**，也叫相对地址。要经过寻址方式的计算或变换才得到内存储器中的物理地址。

edb 调试中看到的 hello 的指令地址都是 16 位的虚拟地址，有些访问指令的地址也是逻辑地址，在程序中虚拟地址和逻辑地址没有明显的界限。通常来说我们是看不到程序的物理地址的。至于线性地址，只是一个地址的概念。

逻辑地址转换成线性地址，虚拟地址，是由段式管理执行的。

线性地址转换成物理地址，是由页式管理执行的。

### 7.2 Intel 逻辑地址到线性地址的变换-段式管理

在段式存储管理中，将程序的地址空间划分为若干个段，这样每个进程有一个二维的地址空间。在段式存储管理系统中，则为每个段分配一个连续的分区，而进程中的各个段可以不连续地存放在内存的不同分区中。

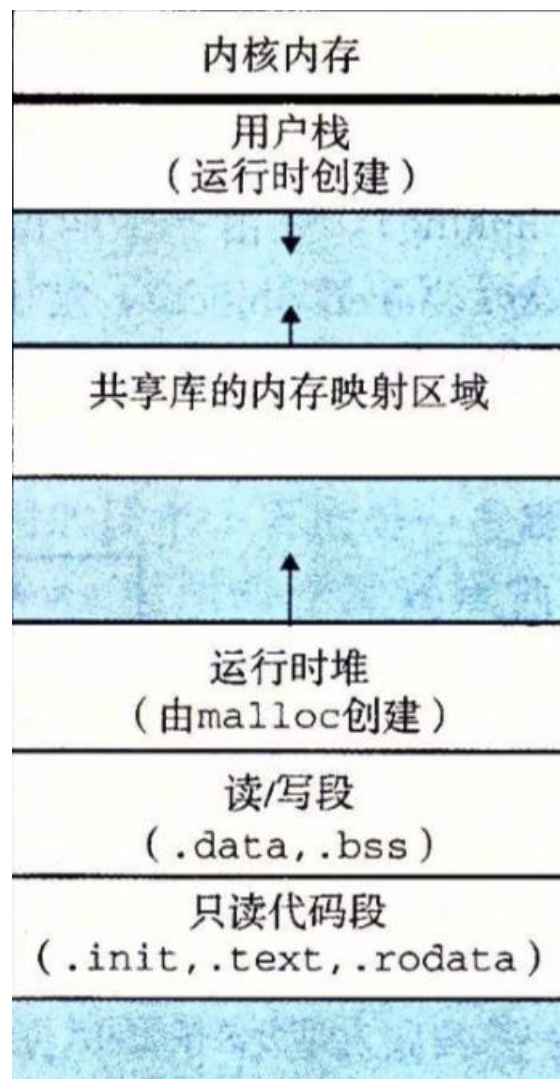


图 7.1 Linux x86-64 运行时内存映像

用户栈是栈段寄存器，共享库的内存映射区域和运行时堆都是辅助段寄存器，读/写段是数据段寄存器，只读代码段是代码段寄存器。

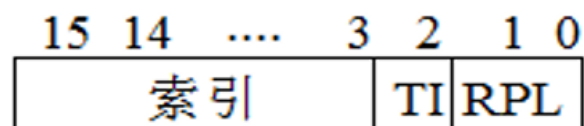


图 7.2 代码段的段选择符

代码段寄存器中的 RPL 字段表示 CPU 的当前特权级。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。出于环保护机制，内核工作在第 0 环，用户工作在第 3 环，中间环留给中间软件用。Linux 仅用第 0 环和第 3 环。TI=0，选择全局描述符表(GDT)，TI=1，选择局部描述符表(LDT)。

段描述符是一种数据结构，实际上就是段表项，分为用户的代码段和数据段描述符，还有系统控制端描述符。

全局描述符表 GDT：只有一个，用来存放系统内每个任务都可能访问的描述符，例如，内核代码段、内核数据段、用户代码段、用户数据段以及 TSS（任务状态段）等都属于 GDT 中描述的段

局部描述符表 LDT：存放某任务（即用户进程）专用的描述符

中断描述符表 IDT：包含 256 个中断门、陷阱门和任务门描述符

**被选中的段描述符先被送至描述符cache，每次从描述符cache中取32位段基址，与32位段内偏移量（有效地址）相加得到线性地址**

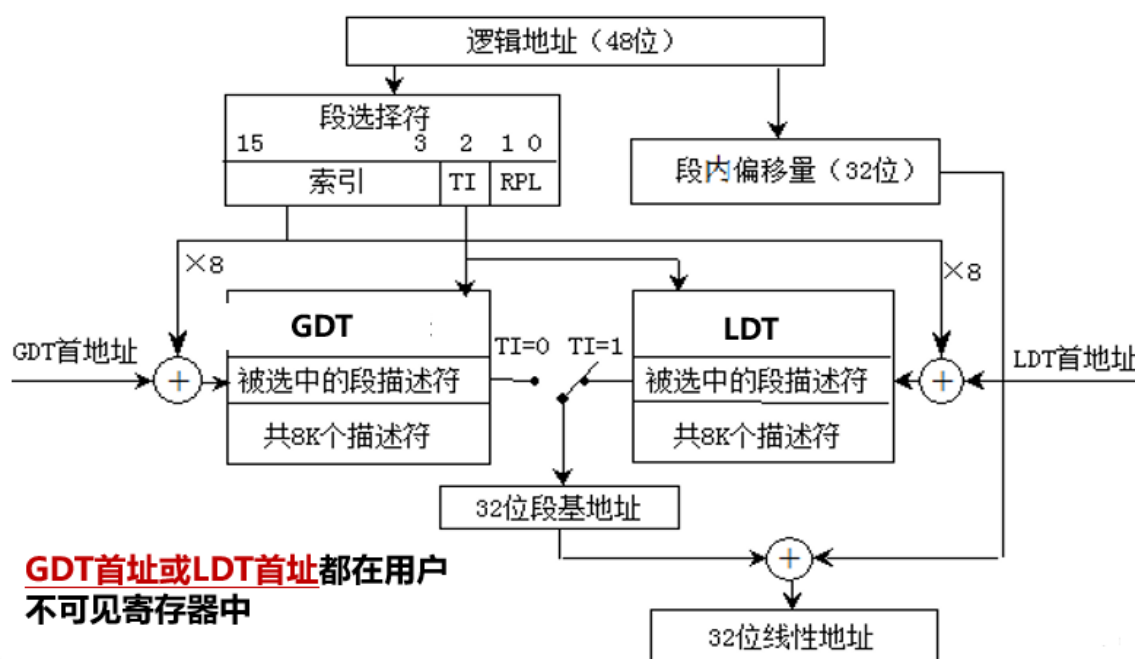


图 7.3 段式管理中逻辑地址向线性地址转换

### 7.3 Hello 的线性地址到物理地址的变换-页式管理

虚拟内存被组织为一个由存放在磁盘上的  $N$  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。**虚拟页**是带虚拟内存系统将虚拟内存分割为大小固定的块，作为磁盘和主存（较高层）之间的传输单元。任何时刻，虚拟页面只有三种情况，要么是未分配的，要么是缓存的，要么是未缓存的。

**页表**是一个存放在物理内存中的数据结构，将虚拟页映射到物理页。每次地址翻译硬件将一个虚拟地址转换为物理地址时，都会读取页表。操作系统负责维护页表的内容，以及在磁盘与 DRAM 之间来回传送页。

页表就是一个**页表条目(PTE)**的数组。虚拟地址空间中的每个页在页表中一个

固定偏移量处都有一个 PTE。可以假设每个 PTE 是由一个有效位和一个  $n$  位地址字段组成的。有效位表明了该虚拟页当前是否被缓存在 DRAM 中。如果设置了有效位，那么地址字段就表示物理内存中相应的物理页的起始位置，这个物理页中缓存了该虚拟页。如果没有设置有效位，那么一个空地址表示这个虚拟页还未被分配。否则，一个非空地址指向的是该虚拟页在磁盘上的起始位置。

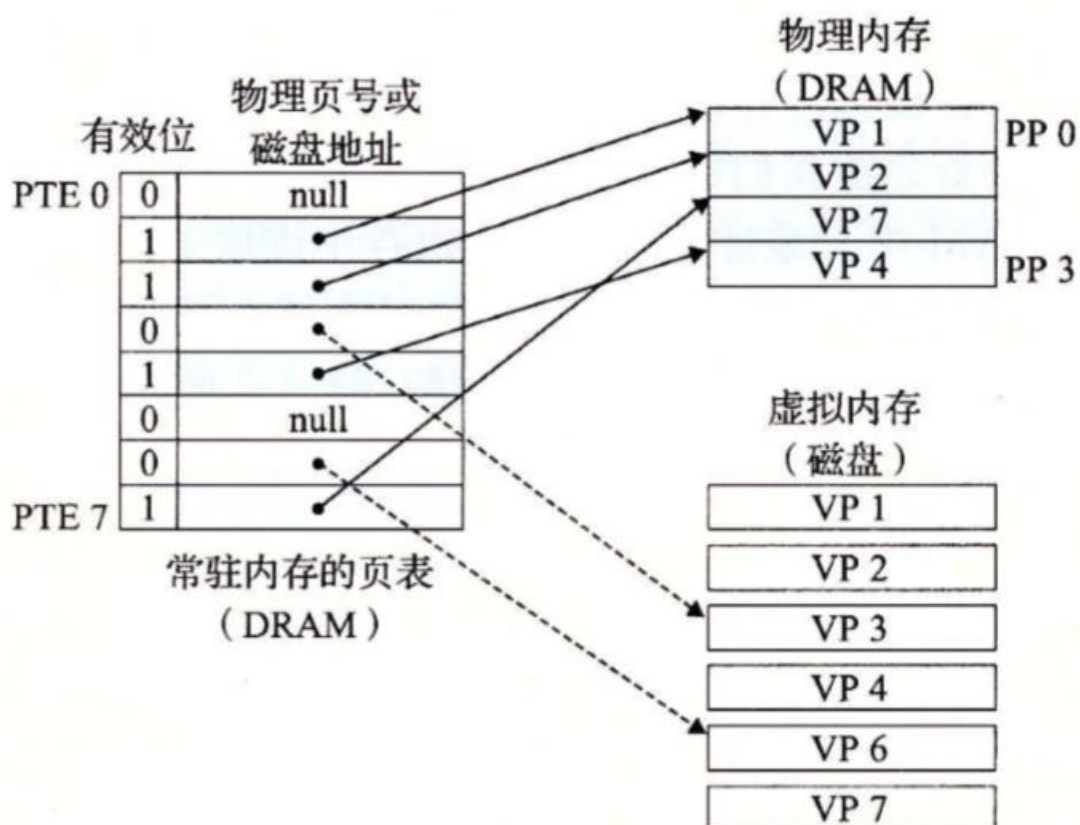


图 7.4 页表、物理内存、虚拟内存三者关系

形式上来说，地址翻译是一个  $N$  元素的虚拟地址空间中的元素到一个  $M$  元素的物理地址空间中元素的映射。

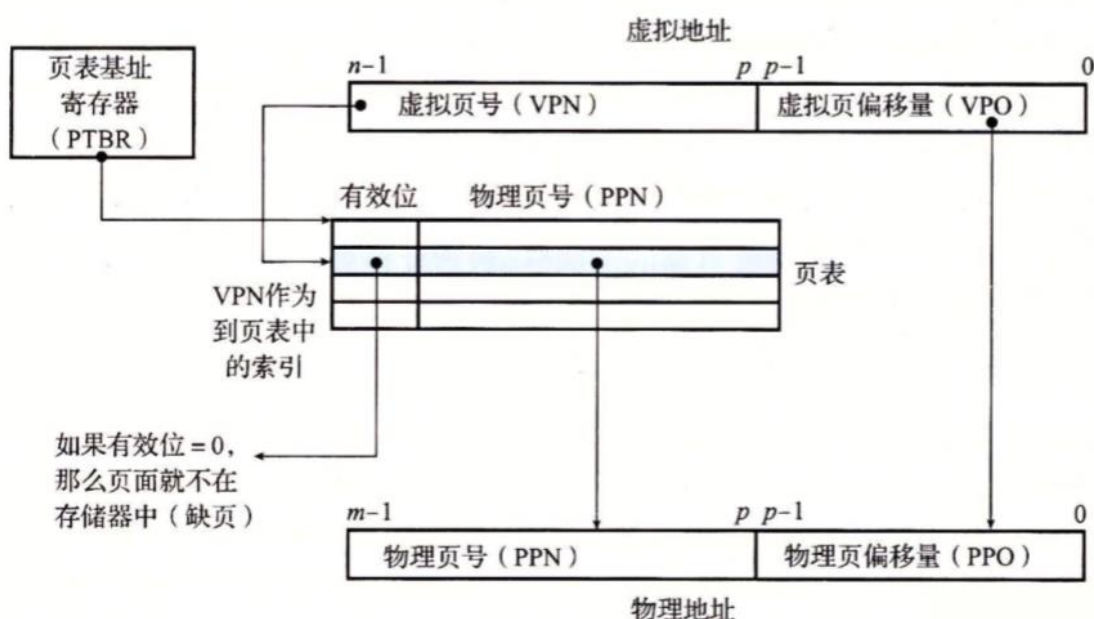
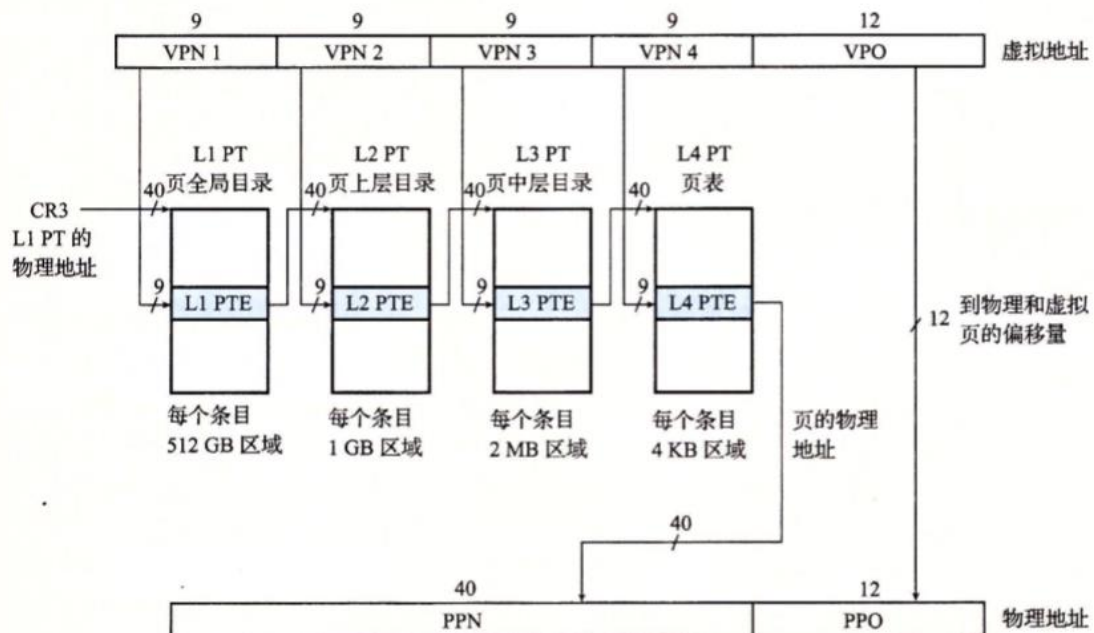
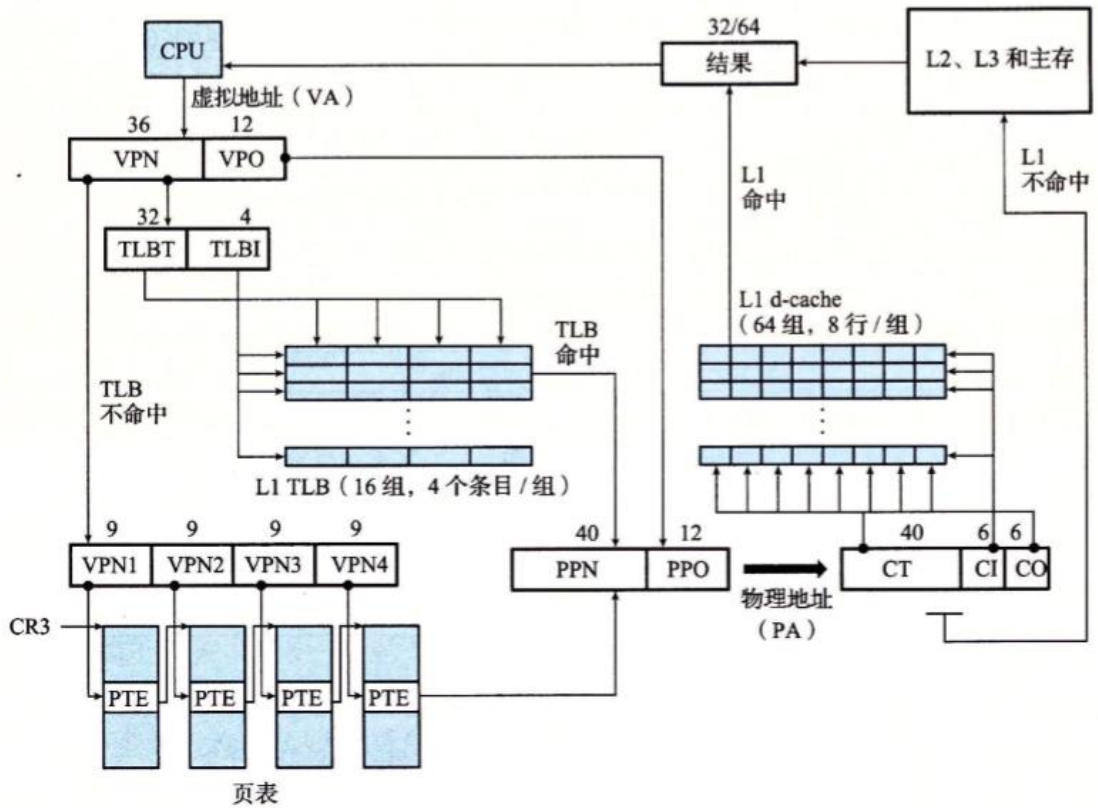


图 7.5 使用页表的地址翻译

图7.2展示了内存管理单元(MMU)如何利用页表来实现虚拟地址到物理地址的映射。CPU 中的一个控制寄存器，**页表基址寄存器**指向当前页表。 $n$  位的虚拟地址包含两个部分，一个  $p$  位的**虚拟页面偏移(VPO)**和一个  $n-p$  位的**虚拟页号(VPN)**。MMU 利用 VPN 来选择适当的 PTE。将页表条目中的**物理页号(PPN)**和虚拟地址中的 VPO 串联起来，就得到相应的物理地址。注意，因为物理和虚拟页面都是 P 字节的，所以**物理页面偏移(PPO)**和 VPO 是相同的。

## 7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

Intel Core i7 实现支持 48 位 (256TB) 虚拟地址空间和 52 位 (4PB) 物理地址空间。Linux 使用的是 4KB 的页。X64 CPU 上的 PTE 为 64 位 (8bytes)，所以每个页表一共有 512 个条目。512 个 PTE 需要有 9 位 VPN 来定位。在四级页表的条件下，一共需要 36 位 VPN，因为虚拟地址空间是 48 位的，所以低 12 位是 VPO。TLB 是四路组联的，共有 16 组，需要有 4 位 TLBI 来定位，所以 VPN 的低 4 位是 TLBI，高 32 位是 TLBT。





Core i7 MMU 用四级的页表来将虚拟地址翻译成物理地址。36 位 VPN 被划分成四个 9 位的片，每个片被用作到一个页表的偏移量。CR3 寄存器包含 L1 页表的物理地址。VPN1 提供到一个 L1 PTE 的偏移量，这个 PTE 包含 L2 页表的基地址。VPN2 提供到一个 PTE 的偏移量，以此类推。

## 7.5 三级 Cache 支持下的物理内存访问

物理内存访问，是基于 MMU 将虚拟地址翻译成物理地址之后，向 cache 中访问的。

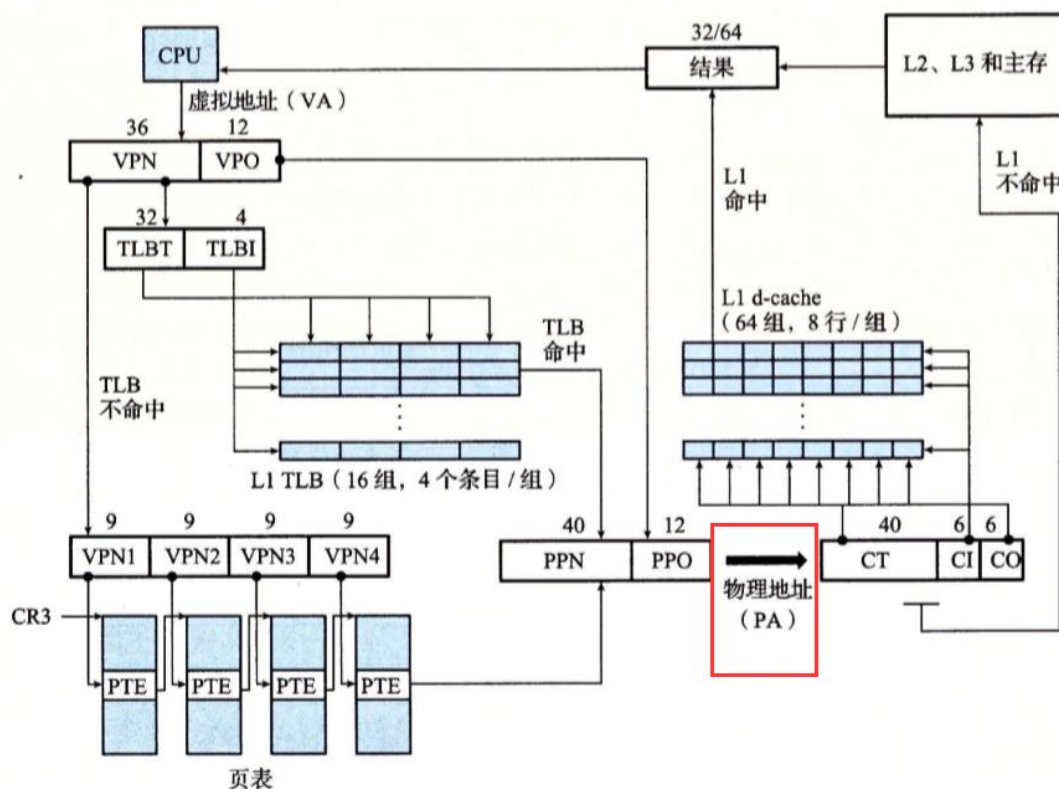


图 7.8 Core i7 地址翻译概况

图 7.8 的右半部分，是 L1 cache 中的物理地址寻址，L2 和 L3 的寻址原理和 L1 相似。

在 cache 中物理地址寻址，按照三个步骤：组选择、行匹配和字选择。在冲突不命中时还会发生行替换。



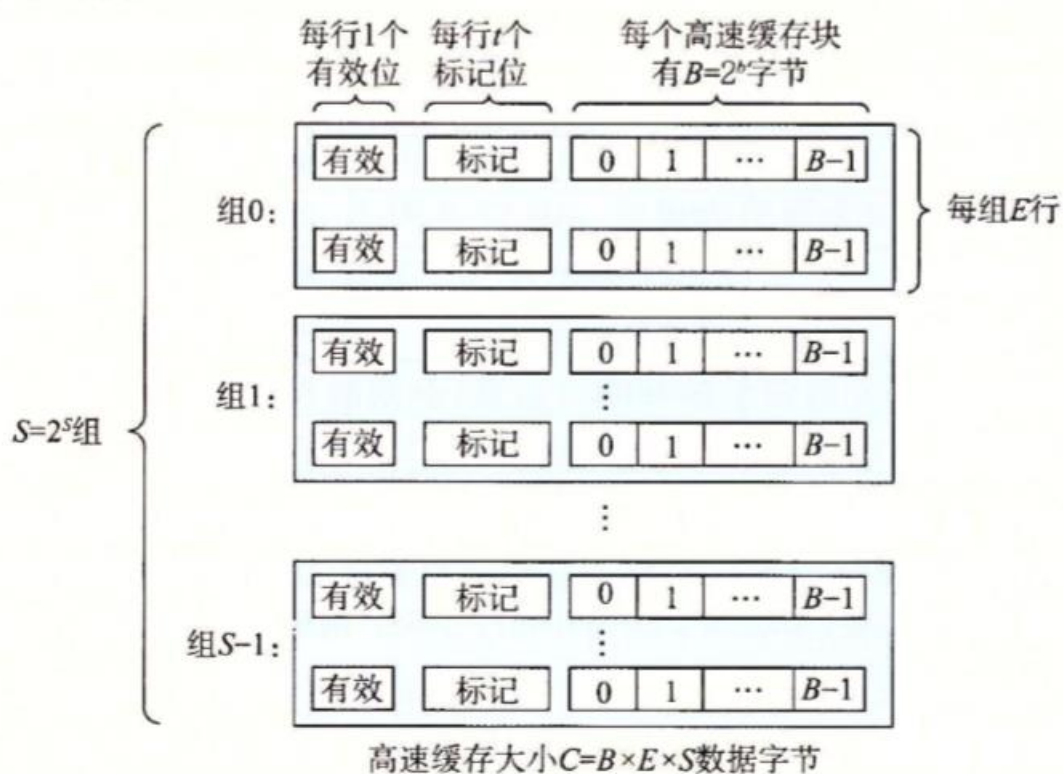


图 7.9 高速缓存的通用组织

高速缓存 ( $S, E, B, m$ ) 是一个高速缓存组的数组。一共有  $S$  个组，每个组包含  $E$  行，每行包含 1 个有效位， $t$  个标记位和一个  $\log_2 B$  位的数据块。



图 7.10 cache 中寻址时物理地址组织结构

高速缓存的结构将  $m$  个地址位划分为  $t$  个标记位， $s$  个组索引位，和  $b$  个块偏移位。

在组选择中，cache 按照物理地址的  $s$  个组索引位 ( $S=2^s$ ) 来定位该地址映射的组。

选择好组后，遍历组中的每一行，比较行的标记和地址的标记，当且仅当这两者相同，并且行的有效位设为 1 时，才可以说这一行中包含着地址的一个副本。也就是缓存命中了。

最后是字选择。定位好了要寻址的地址在哪一行之后，根据地址的块偏移量，在行的数据块中偏移寻址，最后得到的字，就是我们寻址得到的字。

如果缓存不命中，那么它需要从存储器层次结构中的下一层取出被请求的块，然后将新的块存储在组索引位指示的组中的一个高速缓存行中。这个过程，如果

有冲突不命中，就会触发行的替换。

L2 和 L3 cache 的物理地址寻址，和上述过程类似。

## 7.6 hello 进程 fork 时的内存映射

进程这一抽象能够为每个进程提供自己私有的虚拟地址空间，可以免受其他进程的错误读写。不过，许多进程有同样的只读代码区域。而且，许多程序需要访问只读运行时库的相同副本。那么，如果每个进程都在物理内存中保持这些常用代码的副本，那就是极端的浪费了。为了避免这种浪费，内存映射中有了**共享对象**的概念。

如果一个进程将一个共享对象映射到它的虚拟地址空间的一个区域内，那么这个进程对这个区域的任何写操作，对于那些也把这个共享对象映射到它们虚拟内存的其他进程而言，也是可见的。而且这些变化也会反映在磁盘上的原始对象中。

还有一种更节省资源的机制，就是**写时复制**。

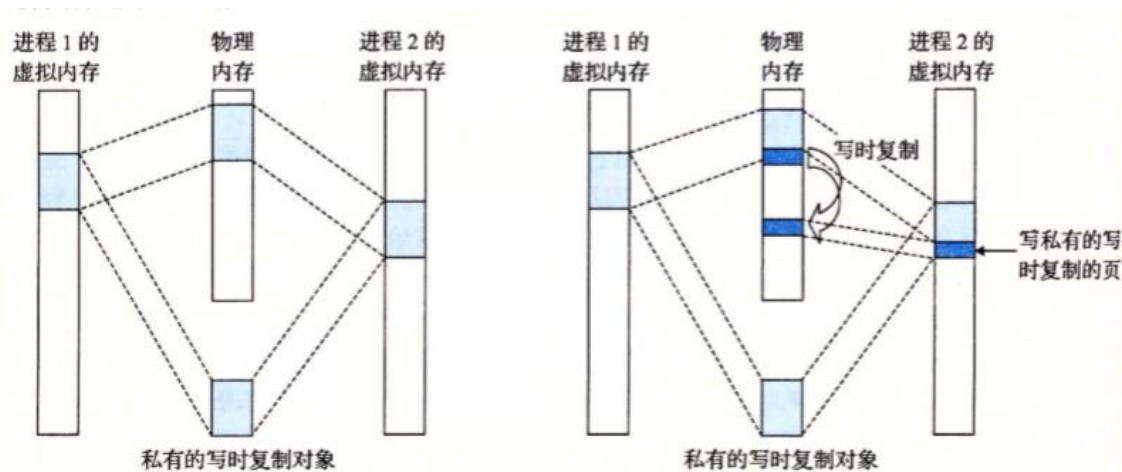


图 7.11 一个私有的写时复制对象

图 7.8 的情况，是两个进程将一个私有对象映射到它们虚拟内存的不同区域，但是共享这个对象同一个物理副本。对于每个映射私有对象的进程，相应私有区域的页表条目都被标记为只读，并且区域结构被标记为**写时复制**。只要没有进程试图写它自己的私有区域，它们就可以继续共享物理内存中对象的一个单独副本。然而，只要有一个进程试图写私有区域的某个页面，那么这个写操作会触发一个保护故障。当故障处理程序注意到保护异常是由于进程试图写私有的写时复制区域中的一个页面而引起的，它就会在物理内存中创建这个页面的一个新副本，更新页表条目指向这个新的副本，然后恢复这个页面的可写权限，当故障程序返回时，CPU 重新执行这个写操作，现在在新创建的页面上这个写操作就可以正常执行了。通过延迟私有对象中的副本直到最后可能的时刻，写时复制最充分地利用

了稀有的物理内存。

在父进程用 `fork` 调用子进程时，内核为新进程创建各种数据结构，并分配给它一个唯一的 `PID`。为了给这个新进程创建虚拟内存，它创建了当前进程的 `mm_struct`、区域结构和页表的原样副本。它将两个进程中的每个页面都标记为只读，并将两个进程中的每个区域结构都标记为私有的写时复制。

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

## 7.7 hello 进程 `execve` 时的内存映射

`hello` 调用 `execve` 后，`execve` 在当前进程中加载并运行包含在可执行目标文件 `hello.out` 中的程序，用 `hello.out` 程序有效地替代了当前程序。加载并运行 `hello.out` 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.out` 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.out` 中，栈和堆地址也是请求二进制零的，初始长度为零。图 7.9 概括了私有区域的不同映射。
3. 映射共享区域，如果 `hello.out` 程序与共享对象（或目标）链接，比如标准 C 库 `libc.so`，那么这些对象都是动态链接到这个程序的，然后再映射到用户虚拟地址空间中的共享区域内。
4. 设置程序计数器（PC）。`execve` 做的最后一件事情就是设置当前进程上下文的程序计数器，使之指向代码区域的入口点。

下一次调度 `hello` 进程时，它将从这个入口点开始执行。Linux 将根据需要换入代码和页面。

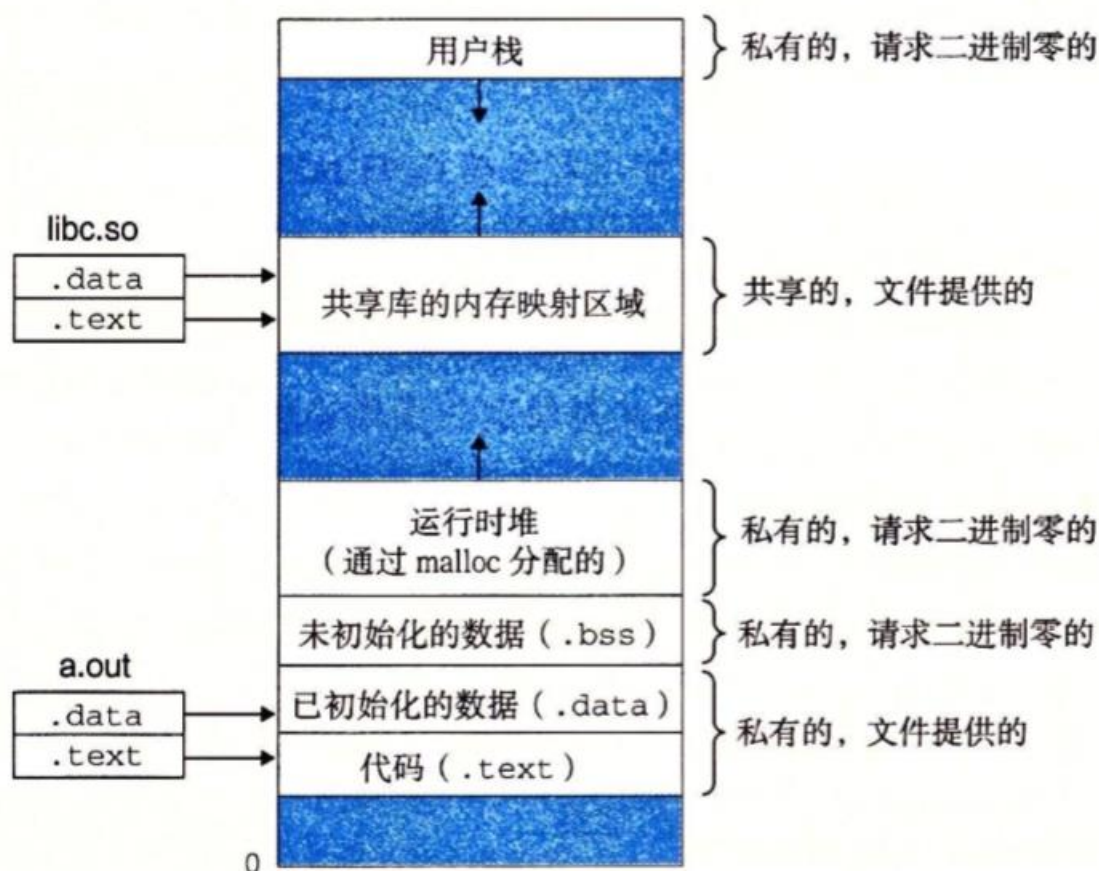


图 7.12 加载器是如何映射用户地址空间的区域的

## 7.8 缺页故障与缺页中断处理

物理内存 (DRAM) 缓存不命中成为**缺页**。假设 CPU 引用了磁盘上的一个字，而这个字所属的虚拟页并未缓存在 DRAM 中。地址翻译硬件会从内存中读取虚拟页对应的页表，推断出这个虚拟页未被缓存，然后触发一个**缺页异常**。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。如果被牺牲的页面被修改了，那么内核会把它复制回磁盘。总之，内核会修改被牺牲页的页表条目，表示它不再缓存在 DRAM 中了。

之后，内核从磁盘把本来要读取的那个虚拟页，复制到内存中牺牲页的那个位置，更新它的页表条目，随后返回。当异常处理程序返回时，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发送到地址翻译硬件。于是，地址翻译硬件可以正常处理现在的页命中了。

## 7.9 动态存储分配管理

动态内存分配器维护着一个进程的虚拟内存区域，称为堆。对于每个进程，内核维护着一个变量 `brk`，它指向堆的顶部。

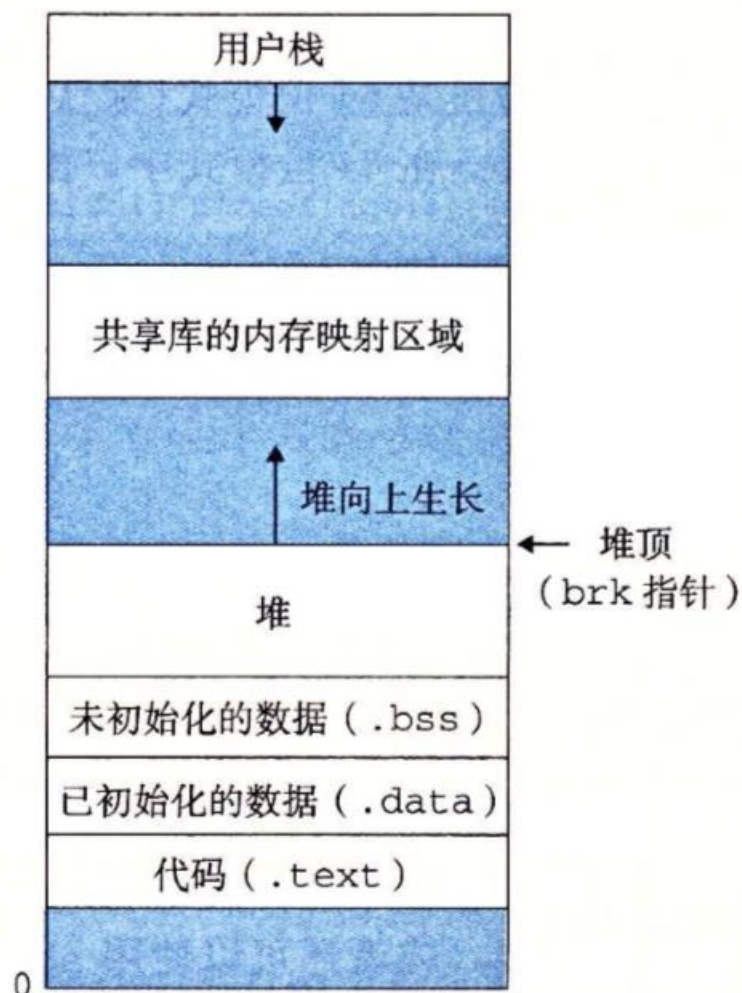


图 7.13 堆

分配器将堆视为一组不同大小的块的集合来维护。每个块就是一个连续的虚拟内存片，要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可用来分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

分配器有两种基本风格，显式分配器和隐式分配器。两种风格都要求应用显式地分配块。不同之处在于由哪个实体来负责释放已分配的块。

显式分配器：要求应用显式地释放任何已分配的块。C 程序通过调用 `malloc`



函数来分配一个块，并通过调用 `free` 函数来释放一个块。

隐式分配器：要求分配器检测一个已分配块何时不再使用，那么就释放这个块。

### 7.9.1 隐式空闲链表

隐式空闲链表是隐式分配器组织空闲块的一种形式。隐式空闲链表有如图 7.11 的堆块数据结构。

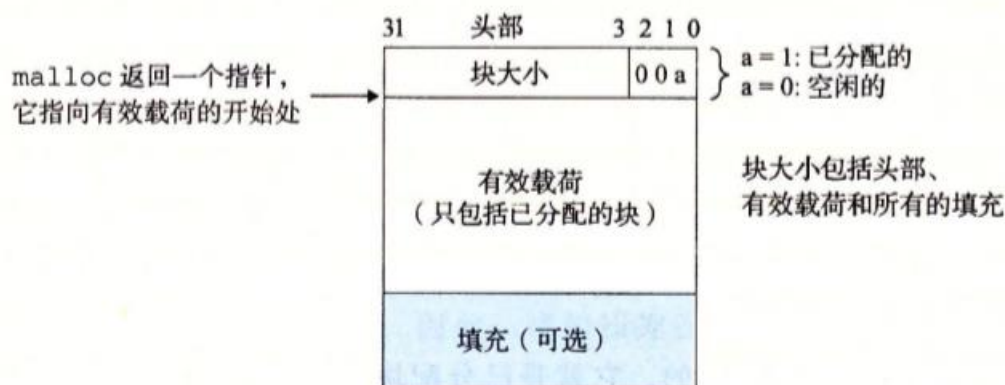


图 7.14 隐式空闲链表的堆块数据结构

在这种情况下，一个块是由一个字的头部、有效载荷，以及可能的一些额外的填充组成的，头部编码了这个块的大小（包括头部和所有的填充），以及这个块是已分配的还是空闲的。如果有双字对齐的约束条件，那么块大小就总是 8 的倍数，且块大小的最低 3 位总是 0。

头部后面就是应用调用 `malloc` 时请求的有效载荷。有效载荷后面是一片不使用的填充块，其大小可以是任意的。

当一个应用请求一个 `k` 字节的块时，分配器搜索空闲链表，查找一个足够大可以放置请求块的空闲块。分配器执行这种搜索方式是由放置策略决定的。一些常见的策略有**首次适配**、**下一次适配**和**最佳适配**。

首次适配从头开始搜索空闲链表，选择第一个合适的空闲块。下一次适配和首次适配相似，只不过是从上一次查询结束的地方开始。最佳适配检查每个空闲块，选择适合所需请求大小的最小空闲块。

如果适配到的空闲块比我们的请求块大小要大很多，那么就要把空闲块分割成两部分，一部分变成分配块，剩下的变成一个新空闲块。

如果分配器找不到一个合适的空闲块（合并后也找不到），那么分配器会调用 `sbrk` 函数，向内核申请额外的堆内存。分配器将额外的内存转化成一个大空闲块，把被请求块放置在这个新的大空闲块中。

每一次有新空闲块生成时，都涉及到空闲块的合并。为了解决**假碎片**问题，任何实际的分配器都必须合并相邻的空闲块，这个过程叫**合并**。

通常堆块的结构还有一个**脚部**，这是在每个块的结尾处，脚部就是头部的一个副本。在合并中，有头部和脚部的块，有利于从被合并块去定位相邻块，获知相邻块的信息。

假设我们有块  $m1, n, m2$ ，已分配标记为  $a$ ，空闲标记为  $f$ ，合并过程有图 7.12 所示的四种情况。

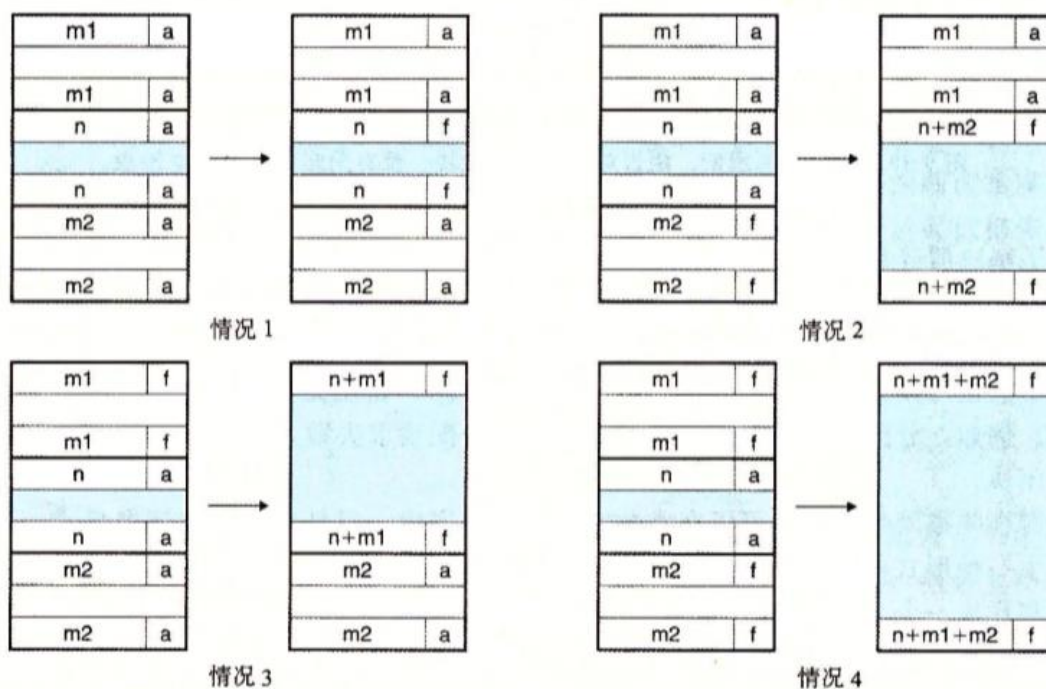


图 7.15 使用边界标记的合并

### 7.9.2 显式空闲链表

显式空闲链表和隐式空闲链表在很多地方都相似，它将空闲块组织成了一个实际的显式数据结构，就是双向链表。其堆块的数据结构如图 7.13 所示。

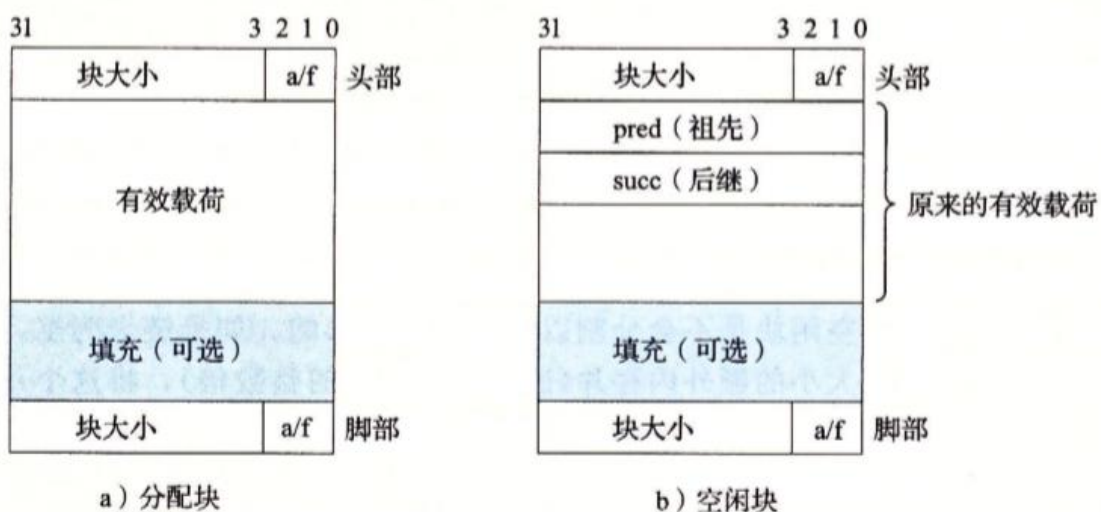


图 7.16 双向空闲链表的堆块数据结构

空闲块中的  $pred$  祖先指针，指向空闲链表中，该块的前驱， $succ$  后继指针，指向空闲链表中，该块的后继。

使用双向链表而不是隐式空闲链表，使首次适配的分配时间从块总数的线性时间减少到了空闲块数量的线性时间。不过释放一个块的时间可以是线性的，也可能是个常数，这取决于我们选择的空闲链表中块的排序策略。

一种方法是用**后进先出（LIFO）**，将新释放的块放置在链表的开始处，使用 LIFO 的顺序和首次适配的放置策略，分配器会最先检查最近使用过的块，在这种情况下，释放一个块可以在线性的时间内完成，如果使用了边界标记，那么合并也可以在常数时间内完成。

另一种方法是**按照地址顺序**来维护链表，其中链表中的每个块的地址都小于它的后继的地址，在这种情况下，释放一个块需要线性时间的搜索来定位合适的前驱。平衡点在于，按照地址排序首次适配比 LIFO 排序的首次适配有着更高的内存利用率，接近最佳适配的利用率。

## 7.10 本章小结

本章讲述了 64 位系统中的内存管理，虚拟内存和物理内存之间的关系，动态内存分配（主要表现为 malloc 和 free）等内容，对程序的内存有一个相对全面的介绍，也讨论了内存的组织形式，和程序与内存的互动。

关于本章内容，具体还请查阅《深入理解计算机系统》第三版的第九章。

**（第 7 章 2 分）**



## 第 8 章 hello 的 IO 管理

### 8.1 Linux 的 IO 设备管理方法

一个 Linux 文件就是一个  $m$  个字节的序列：

$$B_0, B_1, \dots, B_k, \dots, B_{m-1}$$

所有的 I/O 设备（例如网络、磁盘和终端）都被模型化为文件，而所有的输入和输出都被当做对相应文件的读和写来执行，这种将设备优雅地映射为文件的方式，允许 Linux 内核引出一个简单低级的应用接口，称为 Unix I/O。

### 8.2 简述 Unix IO 接口及其函数

设备可以通过 Unix I/O 接口被映射为文件，这使得所有的输入和输出都能以一种统一且一致的方式来执行：

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`，它们可以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`，显式地将改变当前文件位置  $k$ 。
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  $k \geq m$  时执行读操作会触发一个成为 `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”。类似一个写操作就是从内存中复制  $n > 0$  个字节到一个文件，从当前文件位置  $k$  开始，然后更新  $k$ 。
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。

在 Unix I/O 接口中，进程是通过调用 `open` 函数来打开一个存在的文件或者创建一个新文件的，函数声明如下：

```
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。作为上下文的一部分，每个进程都有一个 `umask`，它是通过调用 `umask` 函数来设置的。当进程通过带某个 `mode` 参数的 `open` 函数调用来创建一个新文件时，文件的访问权限位被设置成 `mode&~umask`。

进程通过调用 `close` 函数关闭一个打开的文件。函数声明如下：

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

关闭一个已关闭的描述符会出错。关闭成功返回 0，若出错则返回 -1。

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。函数声明如下：

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

```
ssize_t write(int fd, const 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。

`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。

通过调用 `lseek` 函数，应用程序能够显式地修改当前文件的位置。

### 8.3 printf 的实现分析

`printf` 函数是在 `stdio.h` 头文件中声明的，具体代码实现如下：

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

它的参数包括一个字串 `fmt`，和...。...表示参数个数不能确定，也就是格式化

标准输出，我们也不能确定到底有几个格式串。

在函数的第 6 行，`arg` 变量定位到了第二个参数，也就是第一个格式串。和这句话有关的具体问题，还是请看参考文献[6]，本节只是简述 `printf` 的实现过程。

`va_list` 是一个数据类型，其声明如下：

```
typedef char *va_list
```

至于赋值语句右侧的，是一个地址偏移定位，定位到了从 `fmt` 开始之后的第一个 `char*` 变量，也就是第二个参数了。

接下来是调用 `vsprintf` 函数，并把返回值赋给整型变量 `i`。后来又调用 `write` 函数从内存位置 `buf` 处复制 `i` 个字节到标准输出。想必这个 `i` 就是 `printf` 需要的输出字符总数，那么 `vsprintf` 就是对参数解析了。`vsprintf` 函数代码如下：

```
1. int vsprintf(char *buf, const char *fmt, va_list args)
2. {
3.     char* p;
4.     char tmp[256];
5.     va_list p_next_arg = args;
6.
7.     for (p=buf; *fmt; fmt++) { //p 初始化为 buf, 下面即将把 fmt 解析并把结果存入 buf 中
8.         /* 寻找格式化字串 */
9.         if (*fmt != '%') {
10.             *p++ = *fmt;
11.             continue;
12.         }
13.
14.         fmt++; //此时, fmt 指向的是格式化字串的内容了
15.
16.         switch (*fmt) {
17.             /*这是格式化字串为%x 的情况*/
18.             case 'x':
19.                 itoa(tmp, *((int*)p_next_arg)); //把 fmt 对应的那个参数字串转换格式, 放到 tmp 串中
20.                 strcpy(p, tmp); //tmp 串存到 p 中, 也就是 buf 中
21.                 p_next_arg += 4; //定位到下一个参数
22.                 p += strlen(tmp); //buf 中的指针也要往下走
23.                 break;
24.             /* Case %s */
25.             case 's':
26.                 break;
27.             default:
28.                 break;
29.         }
30.     }
31.
32.     return (p - buf);
33. }
```

这个 `vsprintf` 只处理了 `%x` 这一种格式化字串的情况。已经给出比较详细的注释了。

`write` 函数的汇编代码是这样给出的：

```

1. write:
2.     mov eax, _NR_write
3.     mov ebx, [esp + 4]
4.     mov ecx, [esp + 8]
5.     int INT_VECTOR_SYS_CALL

```

第 5 行，表示要通过系统来调用 `sys_call` 这个函数，函数实现为：

```

1. sys_call:
2.     call save
3.
4.     push dword [p_proc_ready]
5.
6.     sti
7.
8.     push ecx
9.     push ebx
10.    call [sys_call_table + eax * 4]
11.    add esp, 4 * 3
12.
13.    mov [esi + EAXREG - P_STACKBASE], eax
14.
15.    cli
16.
17.    ret

```

在 `write` 和 `sys_call` 中，`ecx` 寄存器中存放的是要打印元素的个数，`ebx` 寄存器中存放的是要打印的 `buf` 字符数组中的第一个元素。这个函数的功能就是不断地打印出字符，直到遇到 `'\0'`。

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）。

到此，`printf` 要打印的东西，就呈现在标准输出上了。

## 8.4 getchar 的实现分析

`getchar` 的函数声明在 `stdio.h` 头文件中，具体代码实现如下：

```

1. int getchar(void)
2. {
3.     static char buf[BUFSIZ];
4.     static char* bb=buf;
5.     static int n=0;
6.     if(n==0)
7.     {
8.         n=read(0,buf,BUFSIZ);
9.         bb=buf;
10.    }
11.    return(--n>=0)?(unsigned char)*bb++:EOF;
12. }

```

bb 是缓冲区的开始, int 变量 n 初始化为 0, 只有在 n 为 0 的情况下, 从缓冲区中读 BUFSIZ 个字节, 就是缓冲区中的内容全部读入。这时候, n 的值被修改为, 成功读入的字节数, 正常情况下 n 就是一个正数值。返回时, 如果 n 大于 0, 那么就返回缓冲区的第一个字符。否则, 就是没有从缓冲区读到字节, 返回 EOF。

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

getchar 等调用 read 系统函数, 通过系统调用读取按键 ascii 码, 直到接受到回车键才返回。

## 8.5 本章小结

所有的 I/O 设备都被模型化为文件, 通过文件的读写来实现 I/O 操作。Unix I/O 接口函数可以实现一些 I/O 操作。printf 的实现和 vsprintf 以及 write 有关, 要先解析格式化字符串, 再调用 I/O 函数 write 写到标准输出上。getchar 函数与键盘回车相关, 也就是需要异步异常-键盘中断的处理, 之后调用 I/O 函数 read, 读取标准输入。

(第 8 章 1 分)

## 结论

用计算机系统的语言, 逐条总结 hello 所经历的过程。

你对计算机系统的设计与实现的深切感悟, 你的创新理念, 如新的设计与实现方法。

本文围绕着 hello 在计算机系统的一生展开, 到此已落下帷幕。hello 的一生, 也是其它程序的一生, 计算机系统的学习, 就是程序在计算机中的日记, 我们和程序一起成长。

hello 在计算机系统的一生概括如下:

1. 编辑(诞生)。用 codeblocks 之类的编译器将正确的程序写用高级语言出来。
2. 预处理(编译系统第一环节)。GCC 中的预处理器将源代码中的预处理指令拓展, 成为新的文本文件。
3. 编译(编译系统第二环节)。GCC 中的编译器将拓展代码文件按照规则, 编译为汇编代码文本文件。
4. 汇编(编译系统第三环节)。GCC 中的汇编器将汇编代码翻译成机器指令格式, 打包成可重定位目标文件。
5. 链接(编译系统第四环节)。GCC 中的链接器将可重定位目标文件和其它必要的可重定位目标文件一切链接, 生成可执行目标文件。

6. 运行（进程产生）。在 shell 中运行 hello，shell 为 hello 创建一个子进程，并调用 `execve` 执行 hello。
7. 内存管理（运行同时）。在 shell 中运行 hello 的同时，为 hello 分配了虚拟地址空间。
8. 内存访问（运行过程中）。在 hello 的运行过程中，任何指令语句的执行，都调动着系统和硬件的配合，将虚拟地址翻译成物理地址，并在主存中取址。是 CPU 和主存之间的交互。
9. 信号与异常（插曲）。信号与异常是 hello 运行过程中的协奏曲，很多信号与异常是必要的。但如果有不必要的信号与异常发生，会对 hello 造成一些影响。
10. 终止（死亡）。hello 由于某种原因（正常或非正常）而终止成为僵死进程，shell 回收 hello，同时内核删除 hello 的数据相关，为 hello 善后。

在一个学期里学完计算机系统这么多知识，是非常有挑战的，的确，学期结束后，感觉自己仍有很多迷惑之处。写下这么一篇长文，既是完成作业，也是期末复习，也是对一学期知识的简单总结。

计算机系统不仅仅是硬件人士的圣经，也是软件人员的必读书目。它真正意义在于，了解程序的一生，这样我们才能真正地认识计算机中的程序。

作者才疏学浅，对计算机系统理解鄙陋，缺点和疏漏在所难免，望读者多加指正。

**（结论 0 分，缺失 -1 分，根据内容酌情加分）**

## 附件

hello.c	源程序 C 语言代码
hello.i	hello.c 预处理后的文本文件
hello.s	编译后的汇编语言文本文件
hello.o	汇编后的可重定位目标文件
hello	链接后的可执行目标文件
helloo_obj.s	hello.o 利用 objdump 工具的反汇编代码
hello_obj.s	hello 利用 objdump 工具的反汇编代码
其他中间结果	readelf 和 objdump 的其他调试代码，反映在了标准输出中，没有保存到本地文件

(附件 0 分，缺失 -1 分)

## 参考文献

- [1] 《深入理解计算机系统》第三版，兰德尔 E.布莱恩特，大卫 R.奥哈拉伦
- [2] GCC 编译命令常用选项，<https://www.cnblogs.com/clover-toeic/p/3737129.html>
- [3] x86: Treat R\_X86\_64\_PLT32 as R\_X86\_64\_PC32 [Linux 3.18.100],  
[https://www.systutorials.com/linux-kernels/209985/x86-treat-r\\_x86\\_64\\_plt32-as-r\\_x86\\_64\\_pc32-linux-3-18-100/](https://www.systutorials.com/linux-kernels/209985/x86-treat-r_x86_64_plt32-as-r_x86_64_pc32-linux-3-18-100/)
- [4] 物理地址，  
<https://baike.baidu.com/item/%E7%89%A9%E7%90%86%E5%9C%B0%E5%9D%80/2901583?fr=aladdin>
- [5] 逻辑地址，  
<https://baike.baidu.com/item/%E9%80%BB%E8%BE%91%E5%9C%B0%E5%9D%80/3283849?fr=aladdin>
- [6] <https://www.cnblogs.com/pianist/p/3315801.html>
- [7] getchar, <https://baike.baidu.com/item/getchar/919709?fr=aladdin>

(参考文献 0 分，缺失 -1 分)