

**计算机系统**

**大作业**

题 目 程序人生-Hello’s P2P

专 业 计算机类

学　　 号 xxxxxxxxxx

班　　 级 xxxxxxx

学 生 cycleke

指 导 教 师 史先俊

**计算机科学与技术学院**

**2019年12月**

**摘 要**

本文介绍了在Linux操作系统下hello的整个生命周期。借助gcc，objdump等工具，对hello的预处理、编译、汇编、链接等过程进行分析。并对程序hello运行过程中的动态链接库调用、内存管理、系统级I/O等进行介绍。

**关键词：**预处理；编译；汇编； 链接；进程；内存管理；IO；

**目 录**

[第1章 概述 - 4 -](#_Toc532238396)

[1.1 Hello简介 - 4 -](#_Toc532238397)

[1.2 环境与工具 - 4 -](#_Toc532238398)

[1.3 中间结果 - 5 -](#_Toc532238399)

[1.4 本章小结 - 5 -](#_Toc532238400)

[第2章 预处理 - 6 -](#_Toc532238401)

[2.1 预处理的概念与作用 - 6 -](#_Toc532238402)

[2.2在Ubuntu下预处理的命令 - 6 -](#_Toc532238403)

[2.3 Hello的预处理结果解析 - 6 -](#_Toc532238404)

[2.4 本章小结 - 7 -](#_Toc532238405)

[第3章 编译 - 8 -](#_Toc532238406)

[3.1 编译的概念与作用 - 8 -](#_Toc532238407)

[3.2 在Ubuntu下编译的命令 - 8 -](#_Toc532238408)

[3.3 Hello的编译结果解析 - 8 -](#_Toc532238409)

[3.4 本章小结 - 14 -](#_Toc532238410)

[第4章 汇编 - 15 -](#_Toc532238411)

[4.1 汇编的概念与作用 - 15 -](#_Toc532238412)

[4.2 在Ubuntu下汇编的命令 - 15 -](#_Toc532238413)

[4.3 可重定位目标elf格式 - 15 -](#_Toc532238414)

[4.4 Hello.o的结果解析 - 18 -](#_Toc532238415)

[4.5 本章小结 - 20 -](#_Toc532238416)

[第5章 链接 - 21 -](#_Toc532238417)

[5.1 链接的概念与作用 - 21 -](#_Toc532238418)

[5.2 在Ubuntu下链接的命令 - 21 -](#_Toc532238419)

[5.3 可执行目标文件hello的格式 - 21 -](#_Toc532238420)

[5.4 hello的虚拟地址空间 - 24 -](#_Toc532238421)

[5.5 链接的重定位过程分析 - 28 -](#_Toc532238422)

[5.6 hello的执行流程 - 30 -](#_Toc532238423)

[5.7 Hello的动态链接分析 - 30 -](#_Toc532238424)

[5.8 本章小结 - 31 -](#_Toc532238425)

[第6章 hello进程管理 - 32 -](#_Toc532238426)

[6.1 进程的概念与作用 - 32 -](#_Toc532238427)

[6.2 简述壳Shell-bash的作用与处理流程 - 32 -](#_Toc532238428)

[6.3 Hello的fork进程创建过程 - 32 -](#_Toc532238429)

[6.4 Hello的execve过程 - 32 -](#_Toc532238430)

[6.5 Hello的进程执行 - 33 -](#_Toc532238431)

[6.6 hello的异常与信号处理 - 34 -](#_Toc532238432)

[6.7本章小结 - 35 -](#_Toc532238433)

[第7章 hello的存储管理 - 36 -](#_Toc532238434)

[7.1 hello的存储器地址空间 - 36 -](#_Toc532238435)

[7.2 Intel逻辑地址到线性地址的变换-段式管理 - 36 -](#_Toc532238436)

[7.3 Hello的线性地址到物理地址的变换-页式管理 - 37 -](#_Toc532238437)

[7.4 TLB与四级页表支持下的VA到PA的变换 - 38 -](#_Toc532238438)

[7.5 三级Cache支持下的物理内存访问 - 38 -](#_Toc532238439)

[7.6 hello进程fork时的内存映射 - 39 -](#_Toc532238440)

[7.7 hello进程execve时的内存映射 - 40 -](#_Toc532238441)

[7.8 缺页故障与缺页中断处理 - 41 -](#_Toc532238442)

[7.9动态存储分配管理 - 42 -](#_Toc532238443)

[7.10本章小结 - 45 -](#_Toc532238444)

[第8章 hello的IO管理 - 46 -](#_Toc532238445)

[8.1 Linux的IO设备管理方法 - 46 -](#_Toc532238446)

[8.2 简述Unix IO接口及其函数 - 46 -](#_Toc532238447)

[8.3 printf的实现分析 - 47 -](#_Toc532238448)

[8.4 getchar的实现分析 - 49 -](#_Toc532238449)

[8.5本章小结 - 50 -](#_Toc532238450)

[结论 - 50 -](#_Toc532238451)

[附件 - 52 -](#_Toc532238452)

[参考文献 - 53 -](#_Toc532238453)

# 第1章 概述

## 1.1 Hello简介

P2P：GCC编译器驱动程序读取源代码文件hello.c，并且把它翻译成一个可执行文件hello。这个翻译过程可以划分为四个阶段：

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，将#include命令包含的头文件/源代码读取并把它直接插入程序文本中。结果就得到了另一个C程序hello.i；

编译阶段：编译器（ccl）将文本文件hello.i翻译成文本文件hello.s，它包含一个汇编语言程序；

汇编阶段：接下来，汇编器（as）将hello.s翻译成机器语言指令，把这些指令打包成一种叫做可重定位目标程序的格式，并将结果保存在目标文件hello.o中。它是一个二进制文件；

链接阶段：最后，链接器（ld）将程序与函数库中需要使用的二进制文件进行链接，形成可执行目标程序二进制文件。

之后执行该文件会将该文件加载在内存中，由系统执行。操作系统会使用fork函数形成一个子进程，分配相应的内存资源，使用execve函数加载进程（Process），完成P2P（From Program to Process）过程。

O2O：hello运行时系统会为其分配对应的虚拟内存空间。Hello读取数据时通过层层存储结构，下一级作为上一级的缓存，使数据可以到达CPU进行处理。系统会删除当前虚拟地址的用户部分已存在的数据结构，为hello创建新的区域结构。程序运行完成后系统会回收hello进程并且删除内存中对应的数据，完成O2O（From Zero-0 to Zero-0）过程。

## 1.2 环境与工具

1.2.1 硬件环境

小米笔记本电脑Pro 15.6”

CPU: Intel(R)\_Core(TM)\_i5-8250U\_CPU\_@\_1.60GHz

内存(RAM): 8GB

硬盘: SAMSUNG\_MZVLW256

1.2.2 软件环境

Microsoft Windows 10 家庭中文版 10.0.17763 64位

Ubuntu 18.04 LTS 64位

Oracle VM VirtualBox 6.0.12 r133076 (Qt5.6.2)

1.2.3 开发工具

gcc (GCC) 9.2.0

GNU Make 4.2.1

GNU Emacs 26.3

## 1.3 中间结果

|  |  |
| --- | --- |
| 文件名 | 作用 |
| hello.c | 源代码 |
| hello.i | hello.c预处理生成的文本文件 |
| hello.s | hello.i编译后得到的汇编语言文本文件 |
| hello.o | hello.s汇编后得到的可重定位目标文件 |
| hello | hello.o链接后得到的汇编语言文本文件 |

表1-1 中间结果

## 1.4 本章小结

本章简述了hello的P2P，O2O过程，介绍了编写本文时的工作环境。

# 第2章 预处理

## 2.1 预处理的概念与作用

概念：编译预处理器（cpp）根据以字符#开头命令，修改原始c程序。比如hello.c中第六行，#include<stdio.h>命令告诉预处理器读取系统头文件stdio.h的内容，并直接插入程序文本，就得到了另一个C程序，以.i为文件扩展名。

作用：用于在编译器处理程序之前预扫描源代码，完成头文件的包含, 宏扩展, 条件编译, 行控制（line control）等操作，便于编译器进行翻译。

## 2.2在Ubuntu下预处理的命令

Ubuntu下可以使用gcc指令预处理，也可以直接使用cpp指令预处理。如下图：

2.2

图2-1 预处理指令

## 2.3 Hello的预处理结果解析

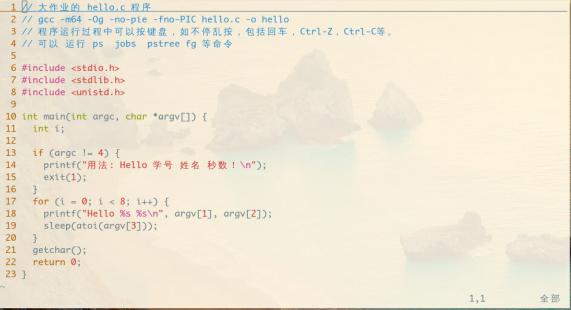


图2-2 hello.c（源代码）

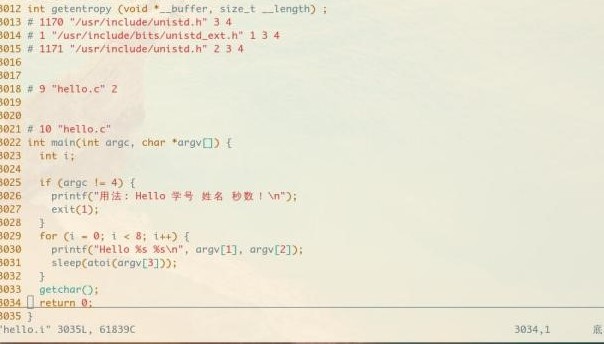


图2-3 hello.i（预处理后代码）

观察上面两图，经过预处理后的hello.i变为3035行，远大于hello.c的23行。原本的#include全部消失，变为stdio.h等头文件的代码。同时hello.c中对于代码翻译无意义的注释也消失。但是我们原本写的主要代码基本没变。

## 2.4 本章小结

本章主要介绍了预处理的概念以及作用，并通过hello.c以及预处理后hello.i的文件对比，对hello预处理结果进行解析，深入的了解了预处理过程的相关知识点，为接下来的处理步骤做准备。

# 第3章 编译

## 3.1 编译的概念与作用

概念：编译器会将某种编程语言写成的源代码转换成另一种编程语言,如ccl将预处理后的.i文本文件翻译为.s文本文件。.s文本文件包含一个汇编语言程序。

作用：编译程序把一个源程序翻译成目标程序的工作过程分为五个阶段：词法分析；语法分析；语义检查和中间代码生成；代码优化；目标代码生成。

编译器会检查代码是否有一些简单的语法错误，还会对代码进行优化，以生成较短的目标代码，充分利用计算机中的寄存器，减少目标代码访问存储单元的次数，充分利用计算机指令系统的特点，以提高目标代码的质量。

## 3.2 在Ubuntu下编译的命令

Ubuntu下可以使用gcc编译，如下图：

3.2

图3-1 编译指令

## 3.3 Hello的编译结果解析

* + 1. 数据

hello.c中有用到的C数据类型有：整型，字符串和数组。

* + - 1. 整型

1. 临时变量i：编译器将局部变量存储在栈上，在hello.s中，我们可以发现i通过-4(%rbp)访问。

3.3_1

图3-2 i在汇编代码的体现

1. 传入参数argc：作为第一个参数传入，存储在栈上，在hello.s中，我们可以通过-20(%rbp)访问。

3.3_2

图3-3 argc在汇编代码的体现

1. 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。
   * + 1. 字符串

hello.c中包含两个字符串，出现在printf中，即：

1. printf("用法: Hello 学号 姓名 秒数！\n");
2. printf("Hello %s %s\n", argv[1], argv[2]);

字符串都声明在.section与.rodata中，在hello.s中以如下形式出现：

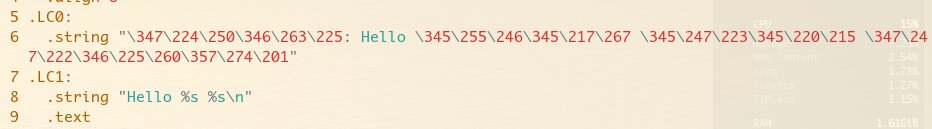


图3-4 字符串在汇编代码的存储

其中汉字以UTF8编码存储，一个汉字在utf-8编码中占三个字节，一个\xxx代表一个字节。

* + - 1. 数组

传入参数argv：作为第二个参数传入，是一个字符串数组，存储在栈上，在hello.s中，我们可以通过-32(%rbp)访问。

3.3_4

图3-5 argv在汇编代码的体现

* + 1. 赋值

Line 17: i = 0，通过mov指令实现，将一个立即数传入一个内存地址。

3.3_1

图3-6 赋值语句在汇编代码的体现

* + 1. 算术操作

Line 17: i++，加法通过add指令实现，add a b实现b = b + a，所以hello.s中对应的代码如下：

3.3_5

图3-7 自加语句在汇编代码的体现

* + 1. 关系操作

hello.c中有用到的关系操作有：“!=”和“<”。

1. “!=”运算 比较通过cmp来实现，指令根据两个操作数之间的差值来设置条件码。如果两个操作数相等，则标记条件码ZF=1，表示两个数是相等的。如果第一个操作数比第二个小，则设置条件码SF=1,表示比较结果为负数，计算机会根据这些条件码来决定跳转。所以“!=”通过如下代码实现：

3.3_6

图3-8 不等语句在汇编代码的体现

1. “<”运算 类比与“!=”实现：

3.3_7

图3-9 小于语句在汇编代码的体现

* + 1. 数组操作

hello.c中只对argv数组进行访问，通过”()”间接寻址，每次通过对%rax获得argv+x的值，然后通过(%rax)访问argv[x]。

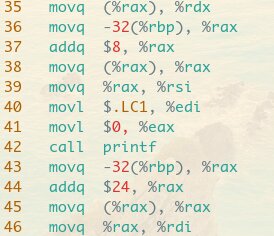


图3-10 循环访问数组在汇编代码的体现

* + 1. 控制转移

hello.c中涉及if和for的控制转移。

1. Line 13: if (argc != 4)

通过cmpl $4, -20(%rbp)进行判断，然后je跳转。实现如下图：

3.3_9

图3-11 if语句在汇编代码的体现

1. Line 17: for(i = 0; i < 8; i++)

循环首先通过movl $0, -4(%rbp)初始化，通过cmpl $7, -4(%rbp)进行判断，利用addl $1, -4(%rbp)迭代。实现如下图：

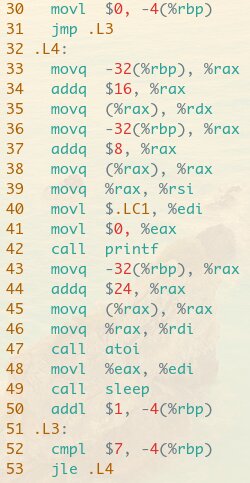


图3-12 for语句在汇编代码的体现

* + 1. 函数操作

hello.c中涉及函数main，printf，exit，sleep，atoi和getchar。

1. main函数

main是整个程序的入口，系统传入参数argc，argv，通过寄存器%rdi（%edi）和%rsi传入，在开始时将这两个参数圧入栈中便于使用寄存器，由系统调用。如下：

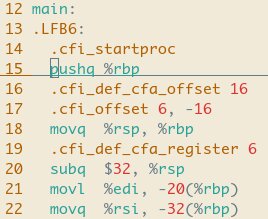


图3-13 main函数将参数压入栈在汇编代码的体现

最后main函数返回值为0，通过%rax（%eax）。

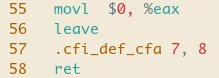


图3-14 main函数返回语句在汇编代码的体现

1. prinft函数

调用第一个prinft函数时：

3.3_12

图3-15 在hello.c中第一次调用printf语句

其对应的汇编代码如下：

3.3_13

图3-16 第一次调用printf语句的汇编代码

其中$.LC0就是字符串"用法: Hello 学号 姓名 秒数！"对应的地址，由于原字符串以'\n'结束，且printf没有其他参数用于格式化，所以编译器优化为puts函数。

观察第二个printf函数（第一个在分析main时有过简单分析），它传入的参数有三个字符串，其中第一个是一个常字符串，后两个从argv数组中读取。

3.3_15

图3-17 在hello.c中第二次调用printf语句

由于argv已经被圧入栈了，所以要通过-32(%rbp)+bias来访问。然后我们就可以将参数放入%rdi（%edi）、%rsi和%rdx三个寄存器中。

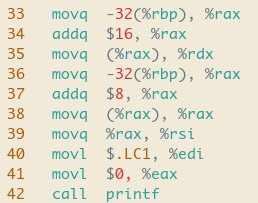


图3-18 第二次调用printf语句的汇编代码

这两个函数均通过call指令调用，被调用后会在屏幕输出两个字符串，它们的会返回打印的字符总个数，但是我们并未使用。

1. exit函数

exit的作用是直接结束程序，传入的参数就是整个程序的返回值，同样通过%edi传入。

3.3_17

图3-19 在hello.c中调用exit语句

3.3_18

图3-20 调用exit语句的汇编代码

exit无返回值，直接退出了程序。

1. sleep函数

sleep传入的参数是atoi的返回值，通过%edi传入。

3.3_19

图3-21 在hello.c中调用sleep语句

3.3_20

图3-22 调用sleep语句的汇编代码

sleep会返回还有多少秒未sleep（如被信号打断，没有则返回0），但是我们没有使用。

1. atoi函数

atoi传入的参数是一个字符串argv[3]，通过%rdi传入。

3.3_19

图3-23 在hello.c中调用atoi语句

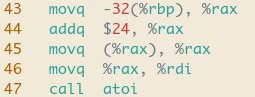


图3-24 调用atoi语句的汇编代码

atoi的返回值为argv[3]对应的数字，保存在寄存器%eax中。

1. getchar函数

getchar不需要参数，直接调用，会返回输入的字符，保存在寄存器%eax中。。

3.3_22

图3-25 在hello.c中调用getchar语句

3.3_23

图3-26 调用getchar语句的汇编代码

## 3.4 本章小结

本章介绍了编译器的概念和作用，ccl将hello.i转换为hello.s，得到一个汇编语言程序，通过分析C语言的各个数据类型以及各类操作对应的汇编代码，进而进行汇编阶段。

# 第4章 汇编

## 4.1 汇编的概念与作用

概念：将汇编程序.s文件转化为二进制机器码文件.o的过程叫做汇编，它把汇编指令打包成一种叫做可重定位目标程序的格式。

作用：通过这个过程，我们可以将汇编代码转化为机器可以理解的机器码。

## 4.2 在Ubuntu下汇编的命令

在Ubuntu下我们可以通过as汇编，也可以通过gcc进行。如下图：

4.2

图4-1 汇编指令

## 4.3 可重定位目标elf格式

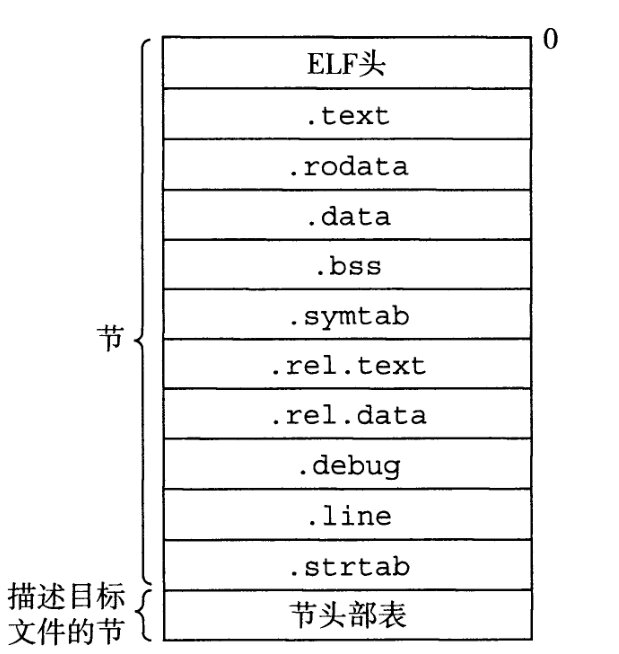


图4-2 可重定位目标文件ELF格式

EEL格式如上图，通过readelf读取hello.o，如下图：

4.3_1

图4-3 使用readelf读取hello.o

* + 1. 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节的大小和字节顺序。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、节头部表的文件偏移、以及节头部表的大小和数目。

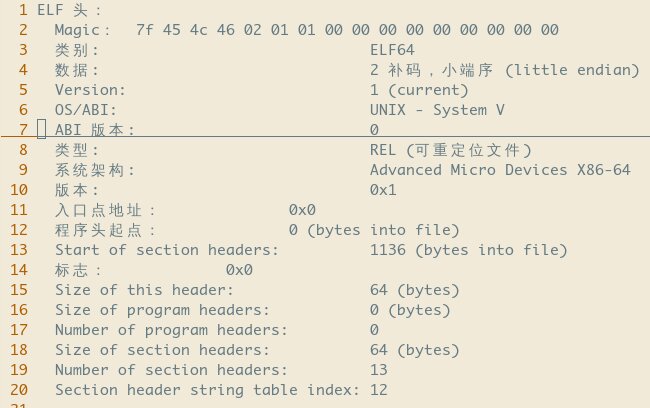


图4-4 hello.o的ELF头

当我们从汇编代码变为了机器代码，程序就真正变成了计算机可以理解的程序，我们也知道了我们的程序真正在计算机中是以什么存储的。机器代码与汇编代码会根据cpu的指令集，产生一个对应，我们也能通过objdump这样的反汇编工具查看机器码对应的汇编码，不过这里对代码已经与我们.s里的汇编代码有了些不同，已经在汇编过程中我们的代码变成了ELF格式，代码被放在代码段，全局变量放在.data段，通过重定位条目得到每个符号不同偏移量，去不同的段找到我们想要的信息。

* + 1. 节头部表

节头部表，用于描述目标文件的节，包含了文件中出现的各个节的语义，包括节的类型、位置和大小等信息。

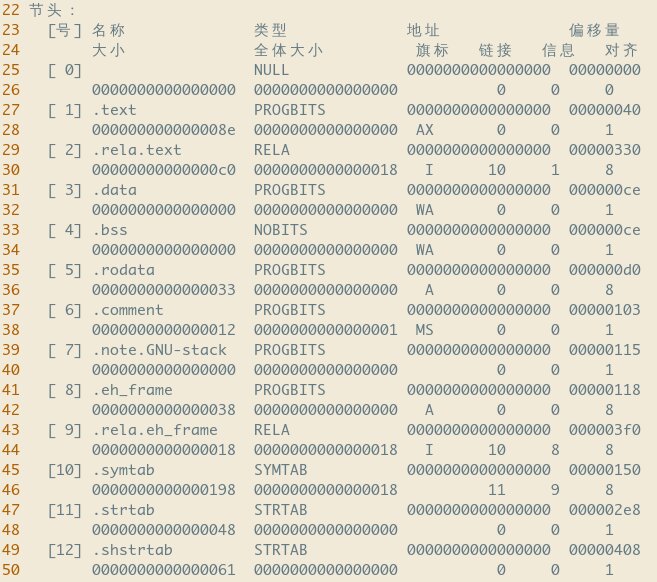


图4-5 hello.o的节头部表

* + 1. 重定位节

.rela.text：一个.text节中位置的列表，包含.text节中需要进行重定位的信息，当链接器把这个目标文件和其他文件组合时，需要修改这些位置。如下图就存储了两个字符串和puts、exit等需要重定位的常量\函数以及对应的偏移量。

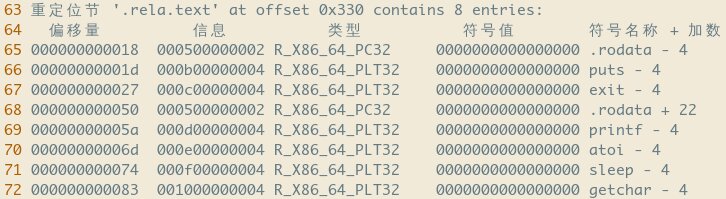


图4-6 hello.o的.rela.text

.rela.eh\_frame：这个section同.rel.text一样属于重定位信息的section，只不过它包含的是eh\_frame的重定位信息，保存了一些调试信息。

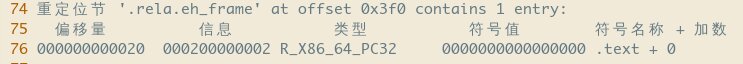


图4-7 hello.o的.rela.eh\_frame

## 4.4 Hello.o的结果解析

使用objdump反汇编的结果如下：

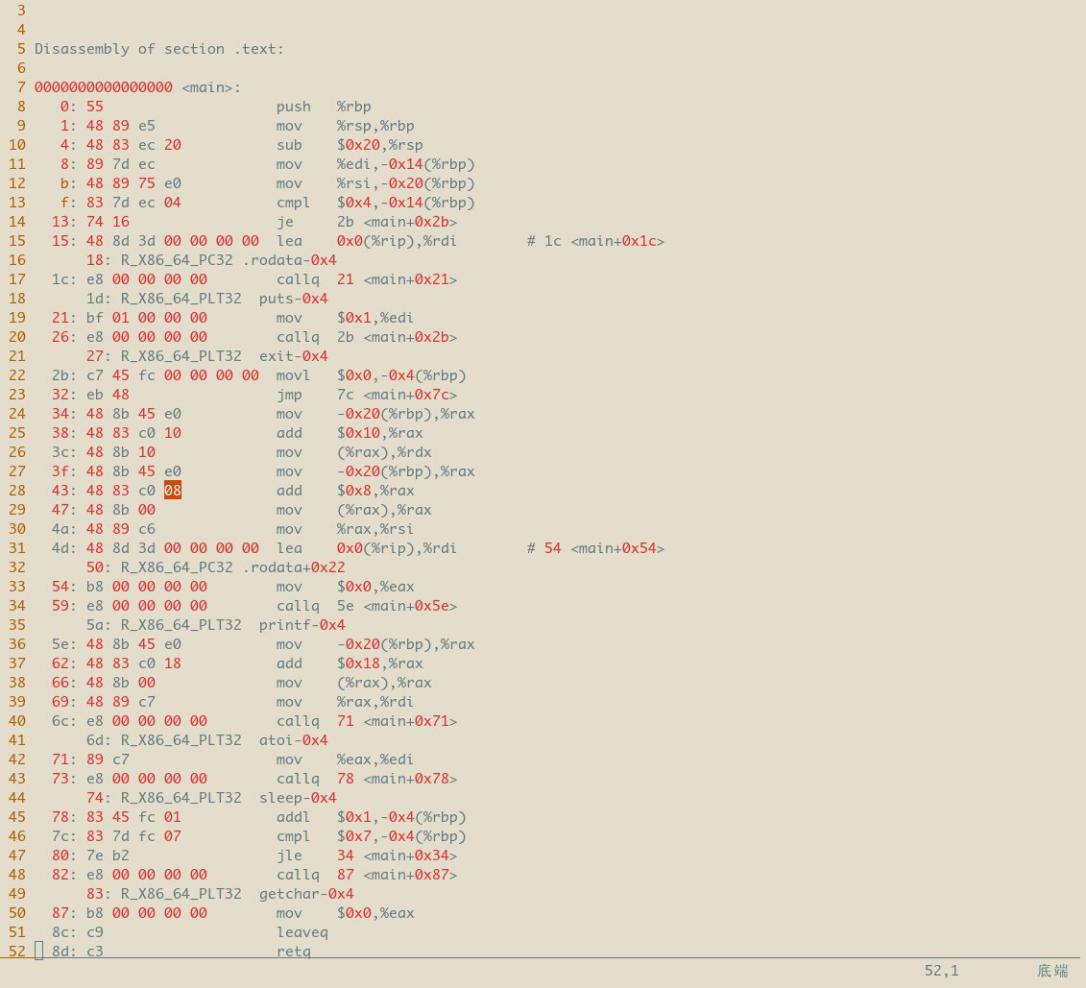


图4-8 hello.o的反汇编代码

与第3章的 hello.s进行对照我们会发现：

1. 伪指令消失

原本hello.s中的许多以'.'开始的伪指令，如下图所示，都消失了。

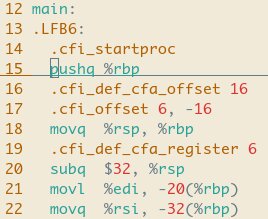


图4-9 hello.s中的伪指令

1. 条件分支变化

在hello.s中的段名称（如.L1、.L2）全部消失，取而代之的则是确定的相对偏移地址。如原本hello.s中的.L3变为7c<main+0x7c>。

4.4_2

图4-10 hello.s中的段名称

4.4_3

图4-11 hello.o中的对应的反汇编代码

1. 函数调用变化

在hello.s中，我们调用puts等来自静态库或者其他的文件，需要进行链接才能调用的函数时，直接call+函数名，但是在反汇编的代码中我们发现对应的机器码是e8 00 00 00 00（即call 0），反汇编的汇编指令是call+相对地址，后面添加注释便于重定位。

4.4_4

图4-12 hello.s中的函数调用

4.4_5

图4-13 hello.o中的对应的反汇编代码

1. 数据访问变化

在hello.s中，我们访问字符串常量是通过一些助记符访问的。而在反汇编的代码中是通过0x0(%rip)访问，同样有注释，便于重定位。

4.4_6

图4-14 hello.s中的常量数据访问

4.4_7

图4-15 hello.o中的对应的反汇编代码

## 4.5 本章小结

本章介绍了汇编。汇编器（as）将汇编代码hello.s转化为可重定位目标文件hello.o，得到一个可以用于链接的二进制文件。通过readelf我们可以查看hello.o的elf信息和重定位信息。通过对比hello.o的反汇编和第3章的hello.s，对汇编过程有更深的理解。

# 第5章 链接

## 5.1 链接的概念与作用

概念：链接是一个或多个由编译器或汇编器生成的目标文件外加库链接为一个可执行文件的过程。

作用：链接器可以解析未定义的符号引用，将目标文件中的占位符替换为符号的地址。链接器还要完成程序中各目标文件的地址空间的组织，这可能涉及重定位工作。通过增量链接还可以使大型项目编译时分离编译，修改单个文件只用编译那个文件后链接，而不用重新编译整个项目，节省编译时间。

## 5.2 在Ubuntu下链接的命令

使用ld的链接命令：

5.2

图5-1 链接指令

## 5.3 可执行目标文件hello的格式

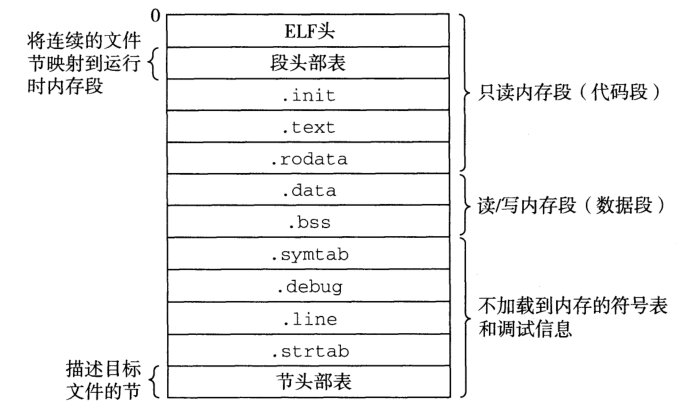


图5-2 可执行目标文件ELF格式

通过readelf可以查看hello的ELF格式。

* + 1. 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节的大小和字节顺序。ELF头剩下的部分还包括ELF头的大小、目标文件的类型、机器类型、节头部表的文件偏移、以及节头部表的大小和数目。

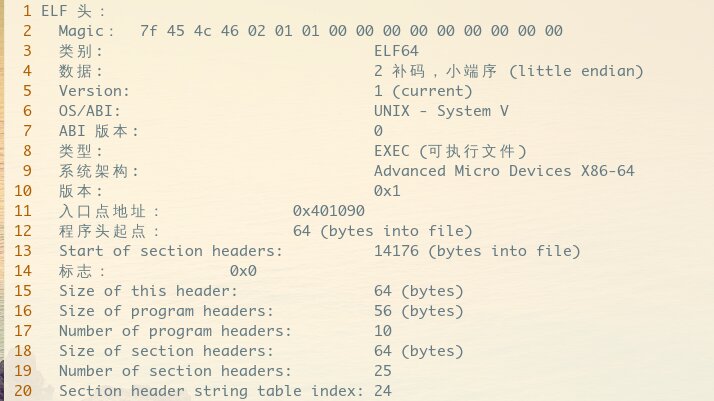


图5-3 hello的ELF头

* + 1. 节头部表

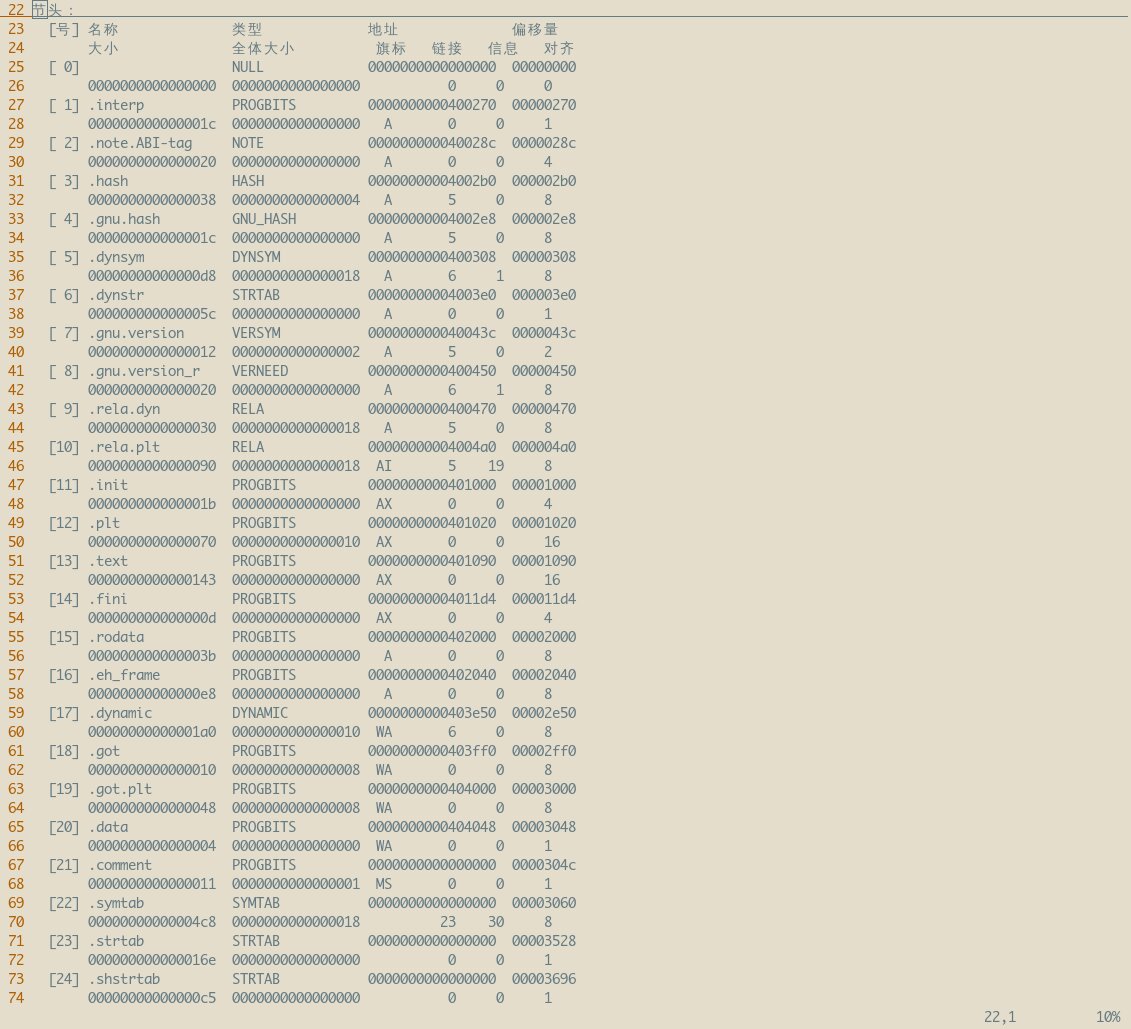


图5-4 hello的节头部表

我们会发现相比与hello.o，hello的节头部表有25项，比hello.o多了12项。节头部表对hello中所有的节信息进行了声明，包括大小size以及在程序中的偏移量offset，大小、全体大小、旗标、链接、信息、对齐等信息，并可以根据此定位各个节所占的区间。

注意：

1. 由于链接时使用的是动态链接，所以hello可执行文件仍然具有.rela.\*的重定位节，便于使用动态链接共享库。
2. .init节定义了一个小函数\_init，程序的初始化代码会调用它。
   * 1. 程序头部表

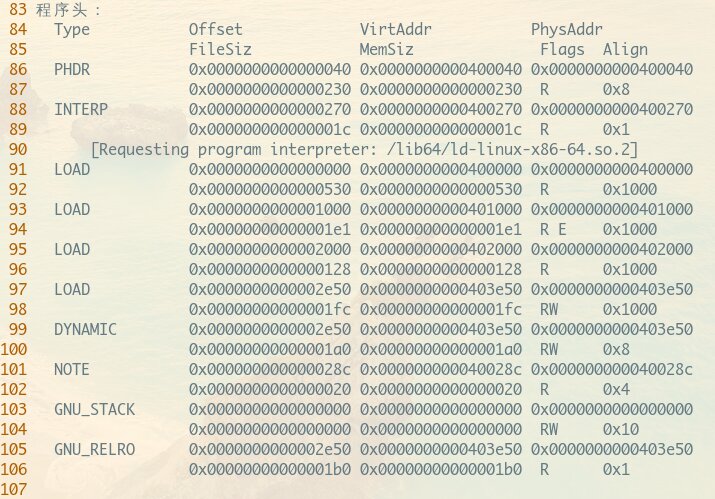


图5-5 hello的程序头部表

程序头部表描述了可执行文件的连续的片和连续的内存段的映射关系。

## 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，hello的虚拟地址从0x0000000000400000。



图5-6 hello的虚拟内存

1. PHDR：起始位置为0x400040，大小为0x230。



图5-7 hello的PHDR段

1. INTERP：起始位置为0x400270，大小为0x1c。

5.4_3

图5-8 hello的INTERP段

1. LOAD：起始位置为0x400000，大小为0x530。

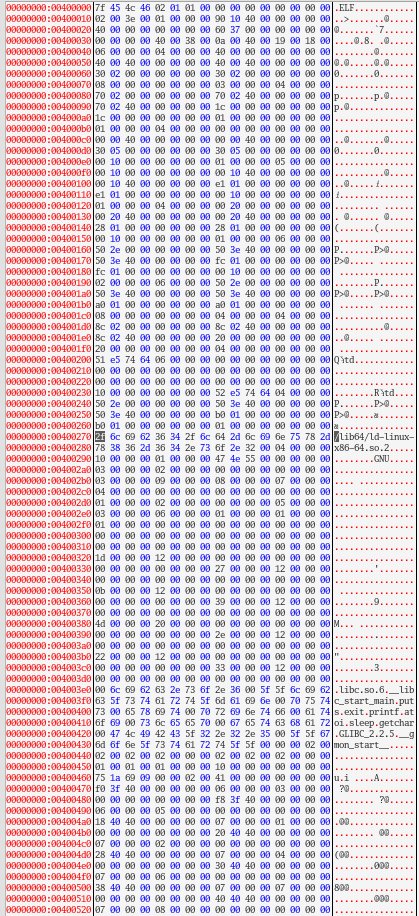


图5-9 hello的LOAD段

其他段同理可以查找到对应的内存区间。

## 5.5 链接的重定位过程分析

使用objdump -d -r hello > hello.d得到hello的反汇编的代码。

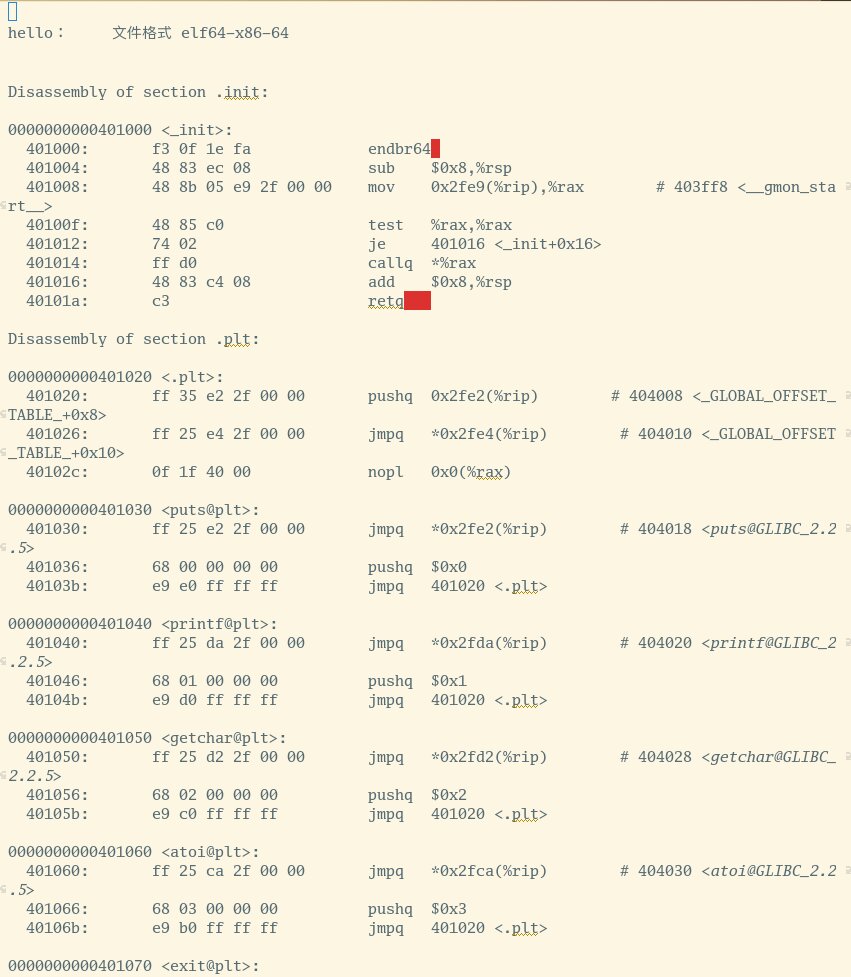


图5-10 hello的反汇编代码

对比与反汇编hello.o得到的代码可以发现：

1. hello的反汇编代码含有更多的函数。hello.o的反汇编代码中只有一个函数main，而在hello的反汇编代码中还出现了\_init，.plt，puts@plt等函数。很多外部的被hello.c调用的函数以及一些初始函数（如\_init）都被链接到hello中。

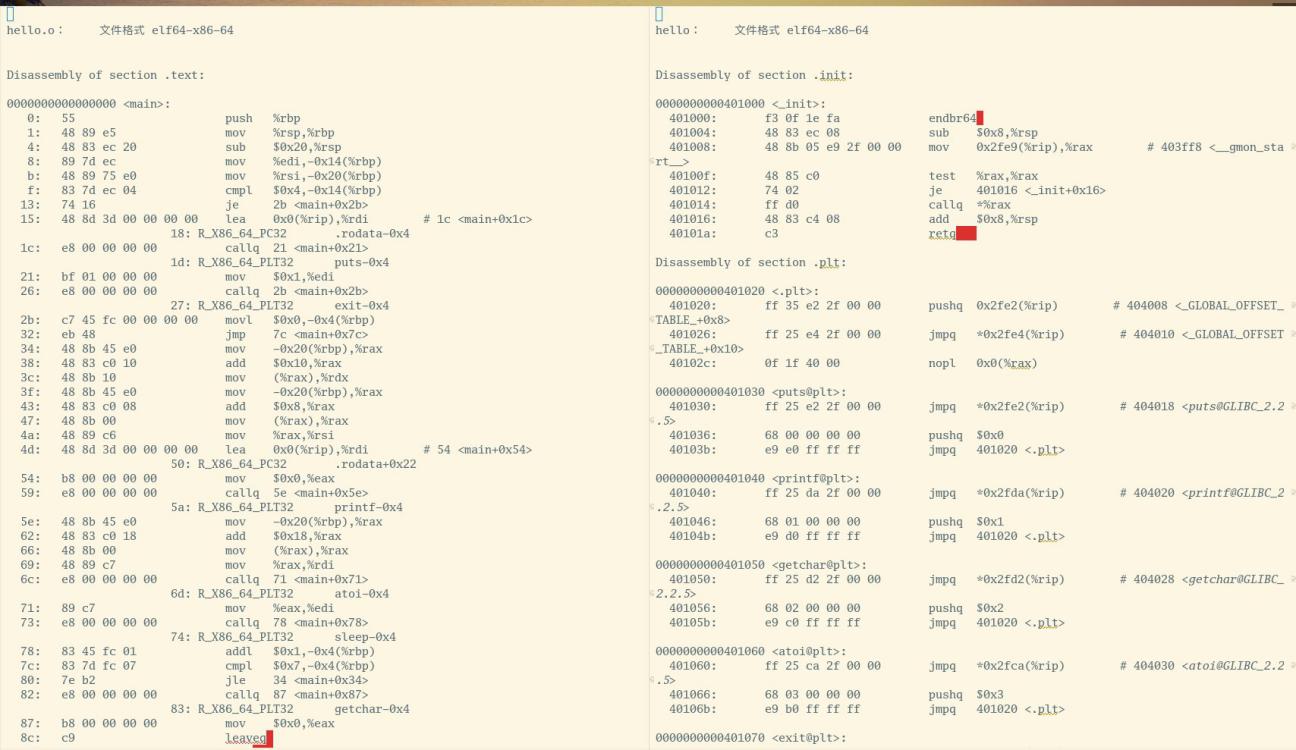


图5-11 函数数量对比

1. 函数调用变化。在hello.o中函数的地址是不确定的，但是在hello中外部函数调用的地址确定，不再是0。如puts的地址就明确为0x401030。

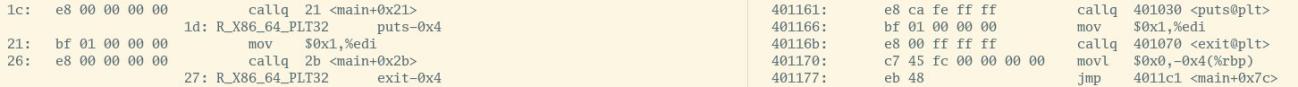


图5-12 函数调用对比

1. 数据引用变化。和函数类似，hello.o中一些数据的地址是不确定的，但是在hello中它们的地址确定了。如.rodata+0x22的地址就明确为0x40202e。

5.5_4

图5-13 数据引用对比

通过上面的分析可以得出在重定位过程中，链接器在完成符号解析以后，就把代码中的每个符号引用和正好一个符号定义（即它的一个输入目标模块中的一个符号表条目）关联起来。此时，链接器就知道它的输入目标模块中的代码节和数据节的确切大小。然后就可以开始重定位步骤了，在这个步骤中，将合并输入模块，并为每个符号分配运行时的地址。重定位由两步组成：

1. 首先是重定位节和符号定义，链接器将所有输入到相同类型的节合并为同一类型的新的聚合节。例如，来自所有的输入模块的.data节被全部合并成一个节，这个节成为hello的.data节。然后，链接器将运行时内存地址赋给新的聚合节，赋给输入模块定义的每个节，以及赋给输入模块定义的每一个符号。当这一步完成时，程序中的每条指令和全局变量都有唯一的运行时内存地址了。
2. 然后是重定位节中的符号引用，链接器会修改代码节和数据节中对每一个符号的引用，使得他们指向正确的运行地址。

## 5.6 hello的执行流程

|  |  |
| --- | --- |
| 程序名 | 程序地址 |
| 加载hello | |
| ld-linux-x86-64.so!\_dl\_start | 0x00007ffff7fd4d30 |
| ld-linux-x86-64.so!\_dl\_init | 0x00007ffff7fe27b0 |
| hello!\_start | 0x0000000000401090 |
| hello!\_\_libc\_csu\_init | 0x00000000004010d0 |
| hello!\_init | 0x0000000000401000 |
| libc.so!\_setjmp | 0x00007ffff7e08b10 |
| 程序运行 | |
| hello!main | 0x0000000000401149 |
| hello!puts@plt | 0x0000000000401030 |
| ld-linux-x86-64.so!\_dl\_runtime\_resolve\_xsave | 0x00007ffff7fe87a0 |
| ld-linux-x86-64.so!\_dl\_fixup | 0x00007ffff7fe1de0 |
| ld-linux-x86-64.so!\_dl\_lookup\_symbol\_x | 0x00007ffff7fdd610 |
| 退出程序 | |
| hello!exit@plt | 0x0000000000401070 |
| libc.so!exit | 0x00007ffff7e0b840 |
| hello!\_fini | 0x00000000004011d4 |

表5-1 hello的执行流程

## 5.7 Hello的动态链接分析

函数调用一个由共享库定义的函数时，编译器无法预先判断出函数的地址，因为定义它的共享模块在运行时可以加载到任意位置。GNU编译系统使用延迟绑定的方式解决该问题，在运行时动态载入。

延迟绑定通过两个数据结构之间简洁但又有些复杂的交互来实现，即过程链接表（PLT）和全局偏移量表（GOT）。

过程链接表（PLT）：PLT是一个数组，其中每个条目是16字节代码。PLT [0]是一个特殊条目，它跳转到动态链接器中。每个被可执行程序调用的库函数都有它自己的PLT条目。每个条目都负责调用一个具体的函数。每个条目都负责调用一个具体的函数。

全局偏移量表（GOT）：GOT是一个数组，其中每个条目是8字节地址。和PLT联合使用时，GOT [0]和GOT [1]包含动态链接器在解析函数地址时会使用的信息。GOT [2]是动态链接器在1d-linux.so模块中的入口点。其余的每个条目对应于一个被调用的函数，其地址需要在运行时被解析。每个条目都有一个相匹配的PLT条目。

通过readelf获得的节头部表可以知道.got.plt的起始位置为0x0000000000404000。在调用\_dl\_start之前对应的内存为：

5.7_1

图5-14 \_dl\_start调用前.got.plt对应内存

对应的内存全为0，而\_dl\_start之后：

5.7_2

5.7_3

图5-15 \_dl\_start调用后.got.plt对应内存

对应的内存有了动态链接器在解析函数地址时会使用的信息。

## 5.8 本章小结

本章介绍了链接的概念及作用，分析了hello的ELF格式，深入学习了hello.o 可重定位文件到hello可执行文件的流程，和链接的各个过程介绍了链接器如何将hello.o可重定向文件与动态库函数链接起来，。

# 第6章 hello进程管理

## 6.1 进程的概念与作用

概念：进程的经典定义就是一个执行中程序的实例。在面向进程设计的系统中，进程使程序的基本运行实体；在面向线程的系统中，进程不是基本执行单位，而是线程的容器。

作用：每次用户通过shell输入一个可执行目标文件的名字，运行程序时，shell就会创建一个新的进程，应用程序也可以创建新进程，并且在这个新进程的上下文中运行它们自己的代码或者其他应用程序。这使得我们可以同时运行多个程序。

## 6.2 简述壳Shell-bash的作用与处理流程

作用：Shell是一种壳层与命令行界面，是操作系统下传统的用户和计算机的交互界面，使可以通过这个界面访问内核提供的服务。

处理流程：Shell首先将输入的指令解析出运行的指令及其参数，然后判断其是否属于内置函数。如果是，则调用相应的函数；如果不是，则会创立一个子进程并在子进程中使用execve运行该程序。如果用户要求在后台运行该程序，那么这个程序会在后台运行。

## 6.3 Hello的fork进程创建过程

在shell输入./hello后shell会判断出这不是一个内置函数，所以会通过调用fork函数创建一个新的子进程，之后运行hello。hello进程得到与shell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。hello进程还获得与shell任何打开文件描述符相同的副本，这就意味着当Shell调用fork 时，hello可以读写shell中打开的任何文件。shell和hello进程之间最大的区别在于它们有不同的PID。

## 6.4 Hello的execve过程

shell会通过execve调用hello。execve函数加载并运行可执行目标文件hello，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hello，execve才会返回到调用函数。所以与fork调用一次返回两次不一样，execve调用一次并且从不返回。

execve函数在当前进程的上下文中加载并运行一个新的函数。它会覆盖当前进程的地址空间，但并没有创建一个新进程。新的函数仍然有相同的PID，并且继承了调用execve函数时已打开的所有文件描述符。

载入并执行hello需要以下几个步骤：

1. 删除已存在的使用者区域。删除当前程序虚拟地址的使用者部分中已存在的区域结构。
2. 对映私有区域。为新程序的代码、数据、.bss和栈区域建立新的区域结构。所有这些新的区域都是私有的、写时复制的。代码和数据被对映为hello档案中的.text和.data区。.bss区域是请求二进位制零的，对映到初值为0的匿名文件，其大小包含在hello中。栈和堆区域也是请求二进制零的，初始长度为零。
3. 对映共享区域。如果hello程式与共享物件连结，那么这些物件都是动态连结到这个程式的，然后再对映到使用者虚拟地址空间中的共享区域内。
4. 设定程式计数器。设定当前程序上下文中的程序计数器，使之指向代码区域的入口点。下一次运行这个程序时，它将从这个入口点开始执行。

## 6.5 Hello的进程执行

系统中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。

内核为每个进程维护了一个上下文。当内核选择的一个新的进程运行时，我们说内核调度了这个进程。所以当内核调度了hello这个新的进程运行后，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，上下文切换会首先保存当前进程的上下文，然后恢复新恢复进程被保存的上下文，最后控制传递给这个新恢复的进程 ，来完成上下文切换。过程如下：

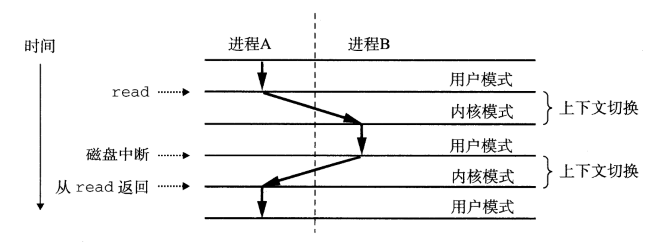


图6-1 进程间切换

上图是对于上下文切换的剖析的一个实例。hello调用sleep和getchar函数时 都会有类似的上下文切换。

## 6.6 hello的异常与信号处理

hello执行过程中出现的可能的异常种类会有四种：

1. 中断 中断是异步发生的，是来自处理器外部的I/O设备的信号的结果。硬件中断的异常处理程序被称为中断处理程序。
2. 陷阱 陷阱是有意的异常，是执行一条指令的结果。就像中断处理程序一样，陷阱处理程序将控制返回到下一条指令。陷阱最重要的用途是在用户程序和内核之间提供一个像过程一样的接口，叫做系统调用。
3. 故障 故障由错误情况引起，它可能能够被故障处理程序修正。当故障发生时，处理器将控制转移给故障处理程序。如果处理程序能够修正这个错误情况，它就将控制返回到引起故障的指令，从而重新执行它。否则处理程序返回到内核中的abort例程，abort例程会终止引起故障的应用程序。
4. 终止 终止是不可恢复的致命错误造成的结果，通常是一些硬件错误，比如DRAM或者SRAM位被损坏时发生的奇偶错误。终止处理程序从不将控制返回给应用程序。处理程序将控制返回给一个abort例程，该例程会终止这个应用程序。
   * 1. 正常运行

图6-2 正常运行

* + 1. 不停乱按

图6-3 不停乱按

如果乱按过程中没有回车，这个时候只是把输入屏幕的字符串缓存起来，如果输入了回车，getchar读入缓存，并把回车后的字符串当作shell输入的命令。

* + 1. Ctrl-Z

图6-4 Ctrl-Z

Ctrl-Z操作向进程发送了一个SIGTSTP信号，让进程暂时挂起，输入jobs、ps指令可以发现hello进程在后台挂起，通过fg指令可以恢复运行。

* + 1. Ctrl-C

图6-5 Ctrl-Z

Ctrl-Z操作向进程发送了一个SIGINT信号，让进程终止，输入jobs、ps指令可以发现hello进程已经被回收。

## 6.7本章小结

本章描述了hello进程管理，介绍了进程的概念和作用，以及shell如何运行hello程序。同时了解了hello执行过程中可能引发的异常和信号处理。

# 第7章 hello的存储管理

## 7.1 hello的存储器地址空间

结合hello说明逻辑地址、线性地址、虚拟地址、物理地址的概念。

逻辑地址：在计算机体系结构中逻辑地址是指应用程序角度看到的内存单元、存储单元、网络主机的地址，即hello.o里面的相对偏移地址。逻辑地址往往不同于物理地址，通过地址翻译器或映射函数可以把逻辑地址转化为物理地址。

线性地址：线性地址是逻辑地址到物理地址变换之间的中间层，即hello中的虚拟地址，等于逻辑地址加上基地址。逻辑地址可转化为线性地址，其地址空间是一个非负整数地址的有序集合，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。

虚拟地址：虚拟地址是程序用于访问物理内存的逻辑地址，即线性地址，在hello中为虚拟地址。

物理地址：计算机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。每个字节都有一个唯一的物理地址。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

* + 1. 基本原理

在段式存储管理中，将程序的地址空间分为若干段，这样每个程序都有一个二维的地址空间。在段式存储管理系统的，为每个段分配一个连续的分割槽，而程序中的各个段可以分配在内存的不同区域。程序载入时，操作系统为所有段分配所需的内存，这些段不需要连续。这些内存通过动态分割槽的管理方法。

总的来说，段式储存管理的优点是：没有内部碎片，外部碎片可以通过内存压缩来消除；便于实现内存共享。缺点与页式储存管理的缺点相同，程序必须全部装入内存。

* + 1. 段式管理的数据结构

为了实现段式管理，需要如下的数据结构来实现程序的地址空间到物理内存空间的对映，并跟踪物理内存的使用情况，以便在装入新的段的时候，合理地分配内存空间。为了完成上述的功能，—个段式系统中，一般要采用如下的数据结构：

1. 程序段表：描述组成程序地址空间的各段，可以是指向系统段表中表项的索引。每段有段基址，即段内地址。
2. 系统段表：系统所有占用段（已经分配的段）。
3. 空闲段表：记忆体中所有空闲段，可以结合到系统段表中。
   * 1. 段式管理的地址变化

在段式管理系统中，整个程序的地址空间是二维的，即其逻辑地址由段号和段内地址两部分组成。为了完成程序逻辑地址到实体地址的对映，处理器会查询内存中的段表，由段号得到段的首地址，加上段内地址，得到实际的实体地址。这个过程也是由处理器的硬件直接完成的，系统只需在程序切换时，将程序段表的首地址装入处理器的特定寄存器（即段寄存器）当中。

## 7.3 Hello的线性地址到物理地址的变换-页式管理

* + 1. 基本原理

将程式的逻辑地址空间划分为定大小的页（page），而物理内存小的页帧（page frame）。程式载入时，可将任意一页放入内存中任意一个页帧，这些页帧不必连续，从而实现了离散分配。该方法需要CPU的硬件支持，来实现逻辑地址和实体地址之间的对映。在页式储存管理方式中地址结构由两部构成，前一部分是虚拟页号（VPN），后一部分为虚拟页偏移量（VPO）：

页式管理方式的优点是：没有外部碎片；一个程序不必连续存放；便于改变程序占用空间的大小（主要指随着程序执行，动态生成的数据增多，所要求的地址空间相应增长）。其缺点是：要求程序全部装入内存，没有足够的内存，程序就不能执行。

* + 1. 页式管理的数据结构

在页式系统中程序建立时，系统为程序中所有的页分配页帧。当程序撤销时收回所有分配给它的页帧。在程序的执行期间，如果允许程序动态地申请空间，系统还要为程序申请的空间分配物理页帧。系统为了完成这些功能，必须记录系统内存中实际的页帧使用情况。系统还要在程序切换时，需要正确地切换两个不同的程序地址空间到物理内存空间的对映。这就要求系统要记录每个程序页表的相关信息。为了完成上述的功能，—个页式系统中，一般要采用如下的数据结构：

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

* + 1. 页式管理的地址变化

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

## 7.4 TLB与四级页表支持下的VA到PA的变换

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Ll页表的物理地址。VPN 1提供到一个Ll PET的偏移量，这个PTE包含L2页表的基地址。VPN 2 提供到一个L2 PTE 的偏移量，以此类推。

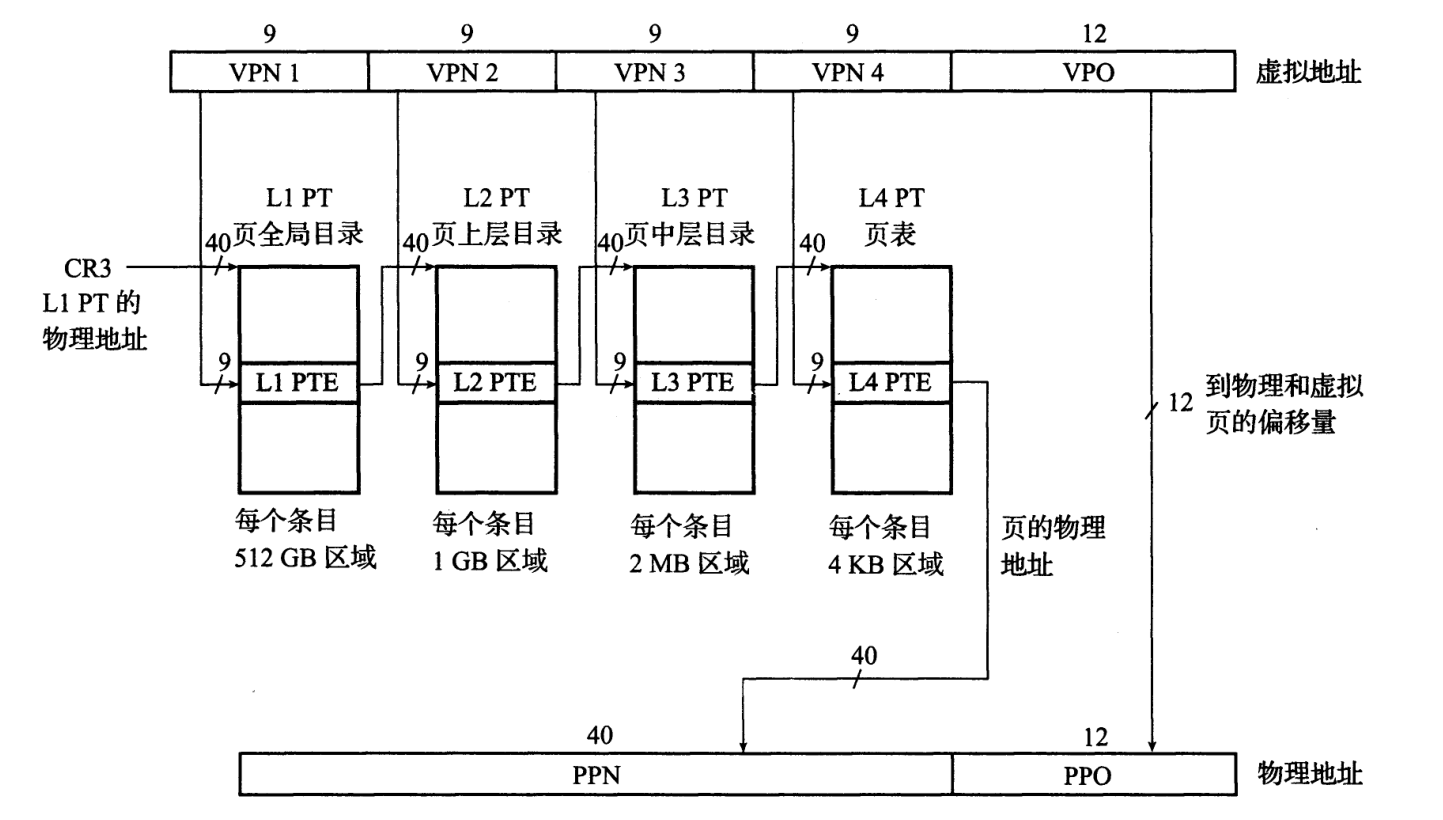


图7-1 四级页表支持下的VA到PA的变换

## 7.5 三级Cache支持下的物理内存访问

我们通过MMU获得对应的物理内存地址后，就需要在物理内存中得到对应的数据。此时我们会通过高速缓存来加速数据读取。首先L1高速缓存会通过地址解析出缓存的索引和偏移，对缓存进行访问，匹配标记查找是否含有相关的字，如果命中，则将数据发送给CPU，如果没有命中，则访问L2缓存，依次类推，直到主存，然后取出这个字，存入高一级缓存，最后返回数据给CPU。

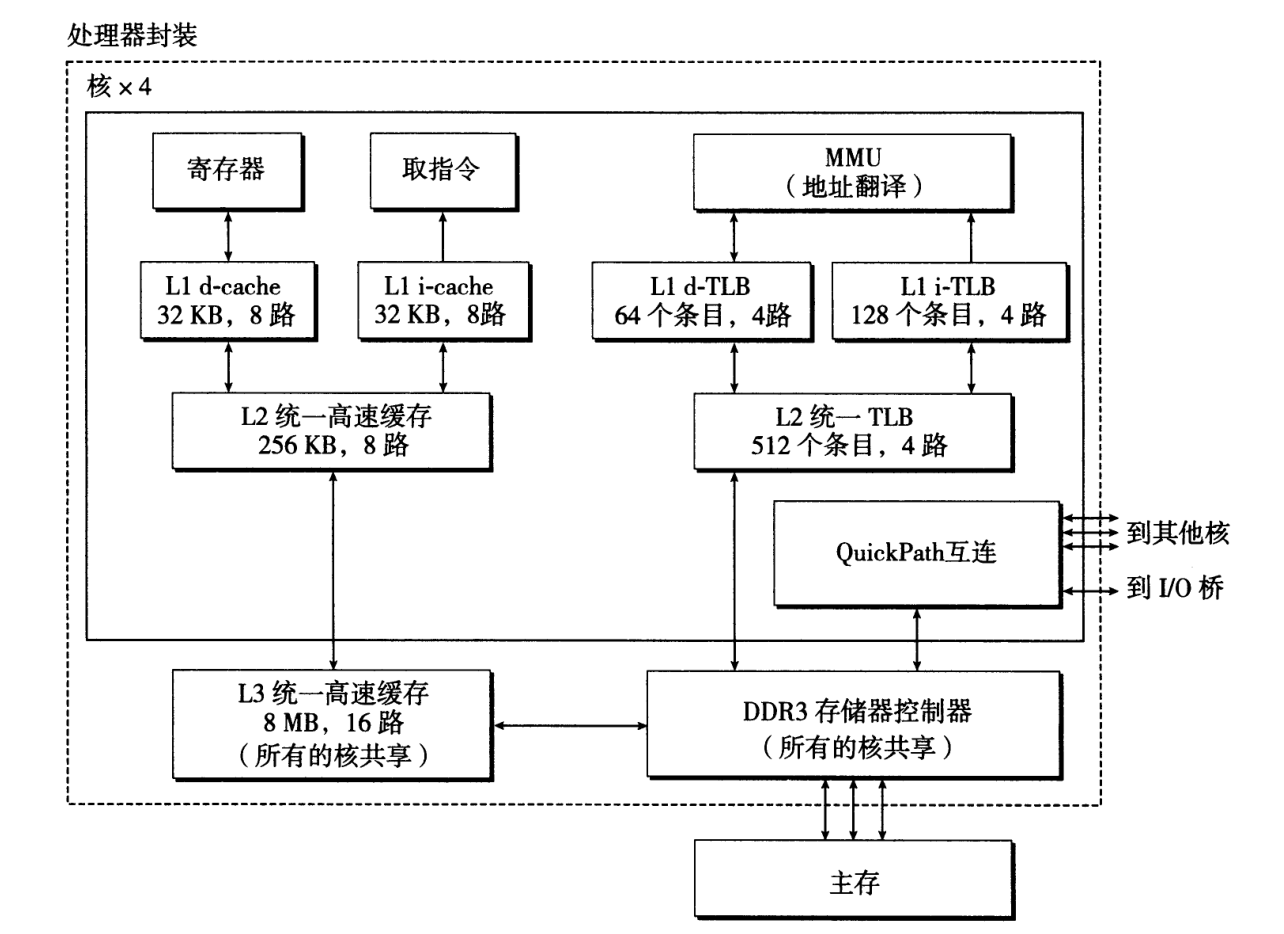


图7-2 三级Cache支持下的物理内存访问

## 7.6 hello进程fork时的内存映射

当fork函数被shell调用时，内核为hello创建各种数据结构，并分配一个唯一的PID。为了给hello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hello进程中返回时，hello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的页面，因此，也就为每个进程保持了私有地址空间的抽象概念。

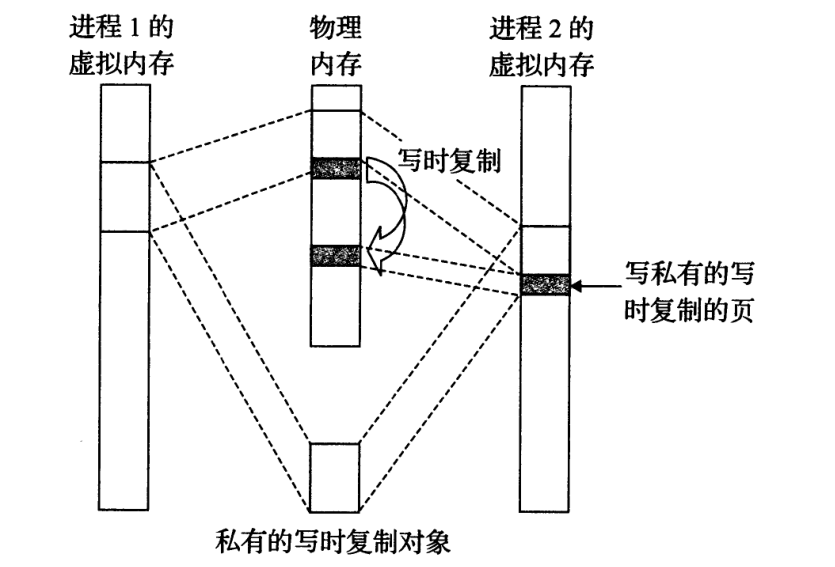


图7-3 hello进程fork时的内存映射

## 7.7 hello进程execve时的内存映射

execve 函数在shell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内存会有如下便变化：

1. 删除已存在的用户区域。删除shell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hello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。栈和堆区域也是请求二进制零的，初始长度为零。下图概括了私有区域的不同映射。

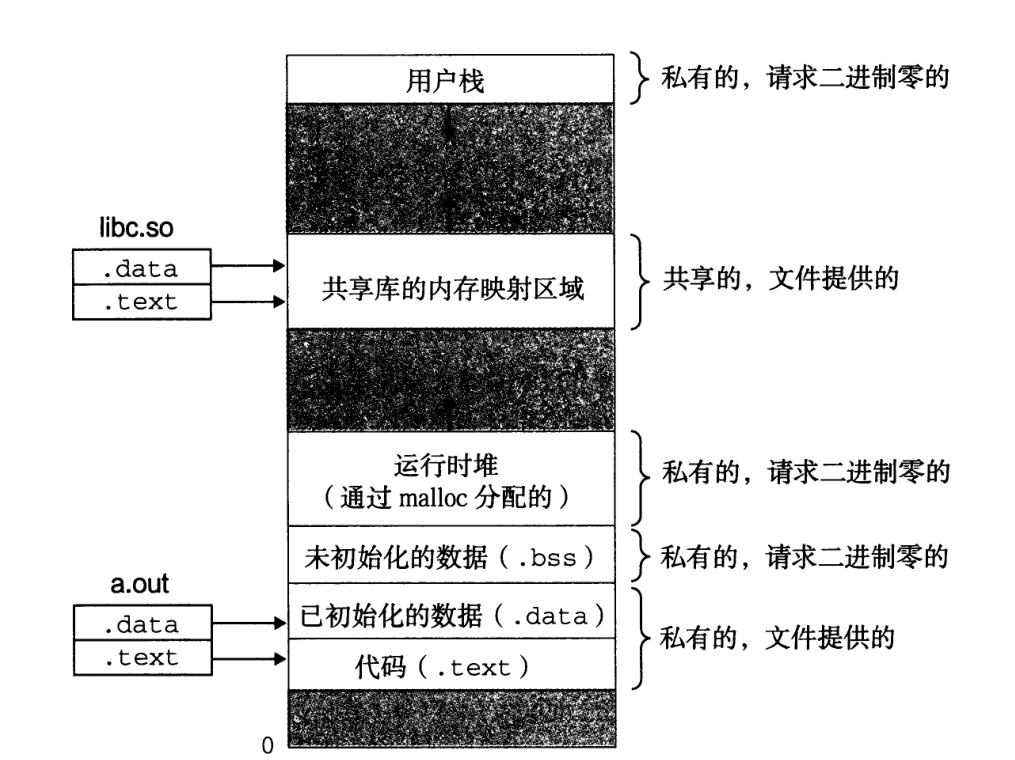


图7-4 私有区域的内存映射

1. 映射共享区域。如果hello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。

## 7.8 缺页故障与缺页中断处理

在虚拟内存的习惯说法中，DRAM缓存不命中称为缺页。例如：CPU引用了VP3中的一个字，VP3并未缓存在DRAM中。地址翻译硬件从内存中读取PTE3，从有效位推断出VP3未被缓存，并且触发一个缺页异常。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，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VP4不再缓存在主存中这一事实。缺页之前：

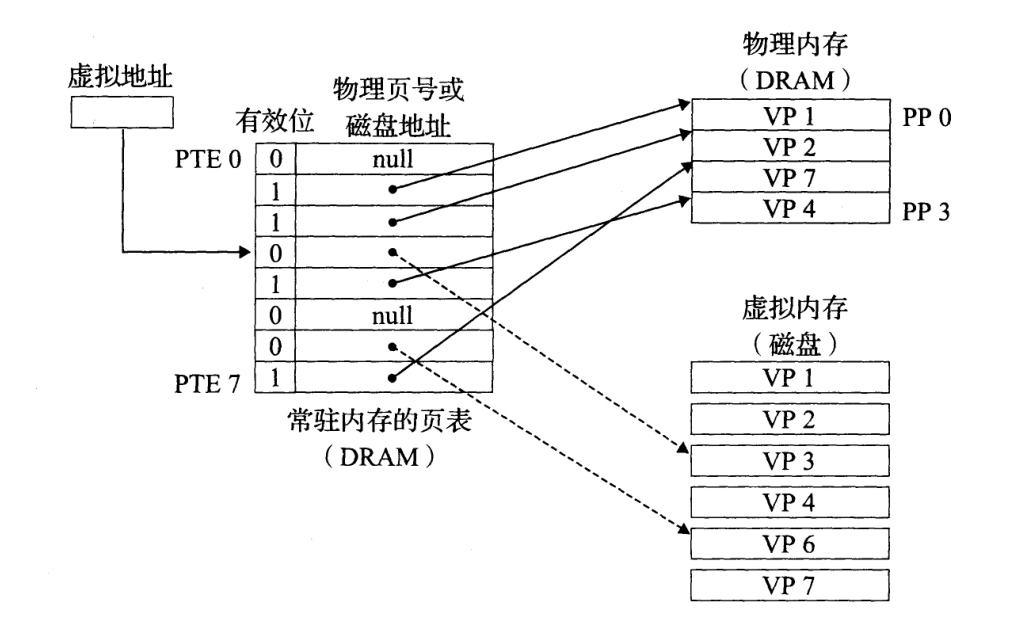


图7-5 缺页前的虚拟内存和物理内存

接下来，内核从磁盘复制VP3到内存中的PP3，更新PTE3，随后返回。当异常处理程序返回时，它会重新启动导致缺页的指令，该指令会把导致缺页的虚拟地址重发送到地址翻译硬体。但是现在，VP3已经缓存在主存中了，那么页命中也能由地址翻译硬件正常处理了。缺页之后：

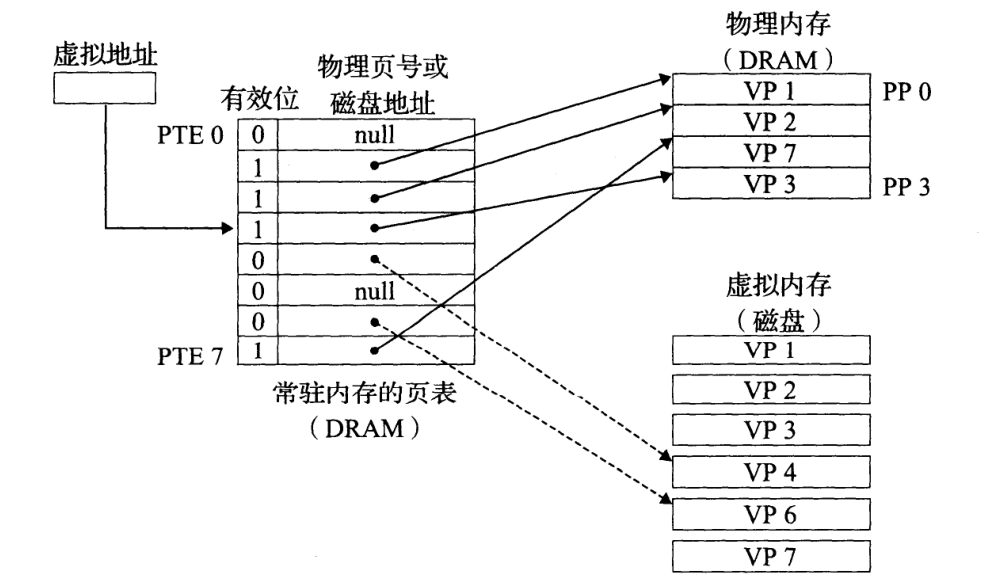


图7-6 缺页后的虚拟内存和物理内存

## 7.9动态存储分配管理

printf函数会调用malloc，下面简述动态内存管理的基本方法与策略：

动态内存分配器维护着一个进程的虚拟内存区域，称为堆。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。

分配器分为两种基本风格：

1. 显式分配器，要求应用显式地释放任何已分配的块。
2. 隐式分配器，要求分配器检测一个已分配块何时不再被程序所使用，那么就释放这个块。隐式分配器也叫做垃圾回收器，而自动释放未使用的已经分配的块的过程叫做垃圾收集。

对于C程序，使用的是显式分配器，能够显式地释放任何已分配的块。C标准库通过malloc程序包的显示分配器。



图7-7 malloc函数

malloc函数返回一个指针，指向大小至少为size字节的内存块，这个块可能包含在这个快内的任何数据对象类型做对齐。

程序是通过free函数来释放已分配的堆块。



图7-8 free函数

* + 1. 隐式空闲链表

我们可以将堆组织为一个连续的已分配块和空闲块的序列，空闲块是通过头部中的大小字段隐含地连接着的，这种结构为隐式空闲表。分配器可以通过遍历堆中所有的块，从而间接地遍历整个空闲块地集合。

一个块是由一个字的头部、有效载荷、可能的填充和一个字的脚部，其中脚部就是头部的一个副本。头部编码了这个块的大小以及这个块是已分配还是空闲的。分配器就可以通过检查它的头部和脚部，判断前后块的起始位置和状态。

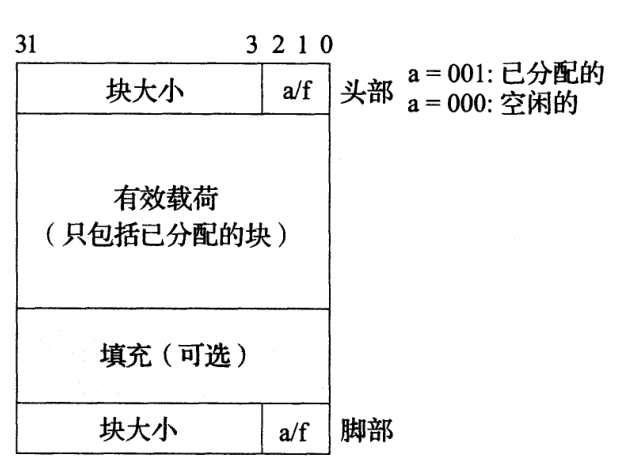


图7-9 隐式空闲链表的一个块

* + 1. 显式空闲链表

将堆组成一个双向空闲链表，在每个空闲块中，都包含一个pred和succ指针。

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

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

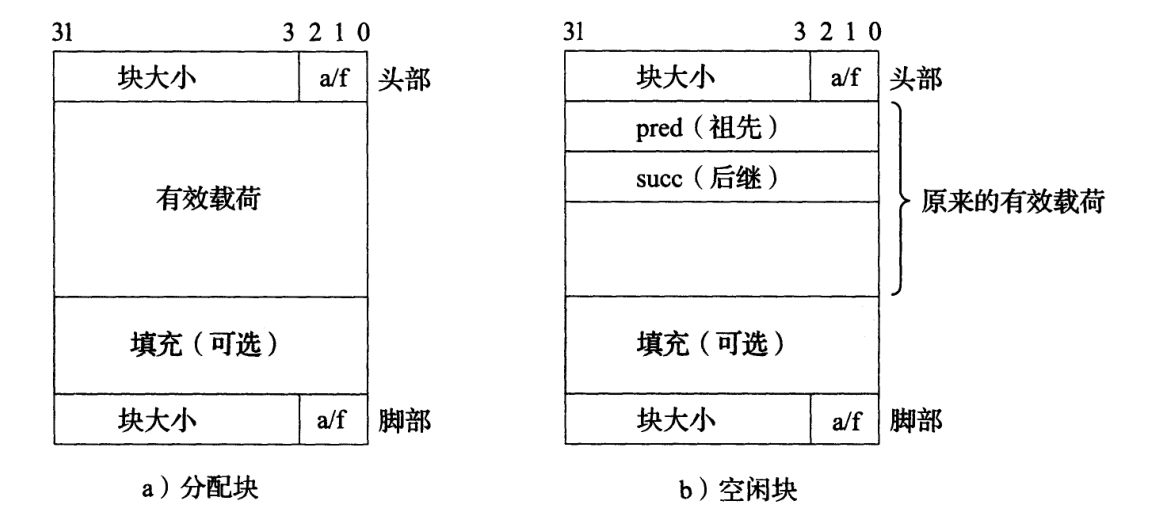


图7-10 显式空闲链表的一个块

## 7.10本章小结

在这一章中，主要介绍了程序的存储结构，介绍了段式管理和页式管理两种从虚拟地址到物理地址的管理方式。程序访问过程中的cache结构和页表结构，进程如何加载自己的虚拟内存空间，内存映射和动态内存分配。

虚拟内存可以为每个进程分配一个独立的虚拟内存空间而不不会受到其他进程影响，多级缓存可以提高数据访问的速度，动态内存分配器则可以提高内存的利用率和效率。

# 第8章 hello的IO管理

## 8.1 Linux的IO设备管理方法

所有的I/O装置（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档案的读和写来执行。这种将装置优雅地映射为文件的方式，允许Linux核心引出一个简单、低阶的应用接口，称为Unix I/O，这使得所有的输入和输出都能以一种统一且一致的方式来执行。

## 8.2 简述Unix IO接口及其函数

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。一个应用程序要求通过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标识符。
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、标准输出和标准错误。
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始的字节偏移量。应用程序能够通过执行seek操作，现显式地设置文件的位置为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的条件，应用程序能够检测到这个条件。在文件结尾处并没有明确的“EOF符号”。
5. 关闭文件。当应用程序完成了对文件的访问后，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。

对应的函数有：

1. open：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。

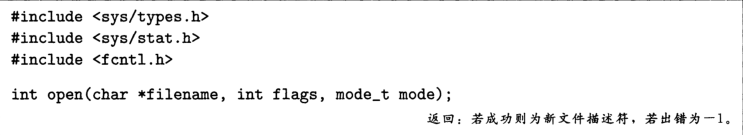


图8-1 open函数

open将filename转换为一个文件描述符，并且放回描述符数字。

1. close：进程通过调用close函数关闭一个打开的文件。

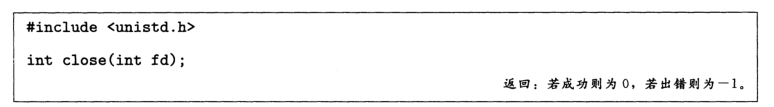


图8-2 close函数

关闭一个已关闭的描述符会出错。

1. read：应用程序通过read函数来执行输入。



图8-3 read函数

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；返回值0表示EOF；否则，返回值表示的是实际传扫的字节数量。

1. write：应用程序通过write函数来执行输出。



图8-4 write函数

write函数从内存位置buf复制至多n个字节到描述符fd的当前文件位置。

## 8.3 printf的实现分析

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

printf的代码如下：

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

其中，va\_list是C语言中解决变参问题的一组宏，本质是一个字节指针。(char\*)(&fmt) + 4) 表示的是...中的第一个参数的地址。所以arg表示函数的第二个参数。

vsprintf的代码如下：

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

vsprintf的作用就是格式化。它接受确定输出格式的格式字符串fmt。用格式字符串对个数变化的参数进行格式化，产生格式化输出。这里的代码中的vsprintf只实现了对16进制的格式化。

write的汇编代码：

1. mov eax, \_NR\_write
2. mov ebx, [esp + 4]
3. mov ecx, [esp + 8]
4. int INT\_VECTOR\_SYS\_CALL

一个int INT\_VECTOR\_SYS\_CALL表示要通过系统来调用sys\_call这个函数。所以它其实是先将参数传入寄存器中，之后调用。

1. sys\_call：
2. call save
4. push dword [p\_proc\_ready]
6. sti
8. push ecx
9. push ebx
10. call [sys\_call\_table + eax \* 4]
11. add esp, 4 \* 3
13. mov [esi + EAXREG - P\_STACKBASE], eax
15. cli
17. ret

syscall将字符串中的字节从寄存器中通过总线复制到显卡的显存中，显存中储存的是字节的ASCII码。

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

getchar的代码实现：

1. **int** getchar() {
2. **char** c;
3. **return** (read(0, &c, 1) == 1) ? c : EOF;
4. }

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。

键盘中断的处理过程：

当用户按键时，键盘接口会得到一个代表该按键的键盘扫描码，同时产生一个中断请求。键盘中断服务程序先从键盘接口取得按键的扫描码，然后根据其扫描码判断用户所按的键并作相应的处理，最后通知中断控制器本次中断结束并实现中断返回；

若用户按下双态键(如：Caps Lock、Num Lock和Scroll Lock等)，则在键盘上相应LED指示灯的状态将发生改变；

若用户按下控制键(如：Ctrl、Alt和Shift等)，则在键盘标志字中设置其标志位；

若用户按下功能键(如：F1、F2、…等)，再根据当前是否又按下控制键来确定其系统扫描码，并把其系统扫描码和一个值为0的字节存入键盘缓冲区；

若用户按下字符键(如：A、1、+、…等)，此时，再根据当前是否又按下控制键来确定其系统扫描码，并得到该按键所对应的ASCII码，然后把其系统扫描码和ASCII码一起存入键盘缓冲区。

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

本章主要介绍了Linux的IO设备管理方法、Unix IO接口及其函数，分析了printf函数和getchar函数。

# 结论

hello的一生包含如下阶段：

1. 预处理：将hello.c根据以字符#开头命令，修改原始c程序，得到hello.i。
2. 编译：将hello.i翻译为hello.s的汇编程序，中间对代码进行语法检查和优化。
3. 汇编：将hello.s翻译为二进制机器码，得到可重定位目标文件hello.o。
4. 链接：将hello.o同等动态库等连接，生成可执行目标文件hello。
5. 创建进程：通过shell运行hello程序。shell通过fork创建子进程，通过execve运行hello。
6. 访问内存：通过MMU将hello中的虚拟地址转换为实际的物理地址，再通过多级缓存读取数据。
7. 异常：程序执行过程中，如果从键盘输入Ctrl-C等命令，会给进程发送一个异常信号，然后通过信号处理函数对信号进行处理。
8. 结束：hello运行完后会由父进程（shell进程）回收，内核会删除对应的数据结构。

至此，我们对于hello的分析已经完成。

通过大作业，我对于这学期所学习的知识有了较系统的回顾。在此次大作业的过程中，我对于一个程序从编译到执行的全过程有了一个粗糙的认识，让我明白了一些计算机系统的底层是如何实现的以及各个系统之间如何协同。当然，我仍然有许多不懂的地方，这需要我今后的学习中继续努力。

# 附件

|  |  |
| --- | --- |
| 文件名 | 作用 |
| hello.c | 源代码 |
| hello.i | hello.c预处理生成的文本文件 |
| hello.s | hello.i编译后得到的汇编语言文本文件 |
| hello.o | hello.s汇编后得到的可重定位目标文件 |
| hello | hello.o链接后得到的汇编语言文本文件 |
| hello\_o.d | hello.o的反汇编结果 |
| hello\_o.elf | hello.o的ELF结构 |
| hello.d | hello的反汇编结果 |
| hello.elf | hello的ELF结构 |

表1 附件

# 参考文献

1. 兰德尔·E·布莱恩特等著；深入理解计算机系统[M]. 北京：机械工业出版社，2016.7.
2. Aho A V. 编译原理[M]. 北京：机械工业出版社, 2009.
3. C library - C++ Reference <http://www.cplusplus.com/reference/clibrary/>.
4. ArchWiki <https://wiki.archlinux.org/>.
5. Linux man pages <https://linux.die.net/man/>.
6. jaywcjlove/linux-command: Linux命令大全搜索工具 ... – GitHub <https://github.com/jaywcjlove/linux-command>.
7. Wikipedia <https://www.wikipedia.org/>.
8. printf 函数实现的深入剖析 <https://www.cnblogs.com/pianist/p/3315801.html>