

**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 1183200123

班 级 1803003

学 生 祁 天

指 导 教 师 史先俊

**计算机科学与技术学院**

**2019年12月**

**摘 要**

本文详细介绍了hello.c程序的一生——一个小小的hello.c，竟然能折腾出这么多东西，它历经重重考验，从源代码到经过预处理、编译、汇编、链接最终生成可执行目标文件hello。然后在shell 中启动，通过fork创建子进程，execve进行加载，内核为新进程创建数据结构， hello便从可执行程序（Program）变成为进程（Process）。hello的一生是复杂的，也是有趣的，让我们一起走近它吧！

**关键词：**计算机系统；编译；汇编；链接；异常；进程；虚拟内存；I/O

**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 - 4 -](#_Toc532238399)

[1.4 本章小结 - 4 -](#_Toc532238400)

[第2章 预处理 - 5 -](#_Toc532238401)

[2.1 预处理的概念与作用 - 5 -](#_Toc532238402)

[2.2在Ubuntu下预处理的命令 - 5 -](#_Toc532238403)

[2.3 Hello的预处理结果解析 - 5 -](#_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 - 15 -](#_Toc532238410)

[第4章 汇编 - 16 -](#_Toc532238411)

[4.1 汇编的概念与作用 - 16 -](#_Toc532238412)

[4.2 在Ubuntu下汇编的命令 - 16 -](#_Toc532238413)

[4.3 可重定位目标elf格式 - 16 -](#_Toc532238414)

[4.4 Hello.o的结果解析 - 19 -](#_Toc532238415)

[4.5 本章小结 - 22 -](#_Toc532238416)

[第5章 链接 - 23 -](#_Toc532238417)

[5.1 链接的概念与作用 - 23 -](#_Toc532238418)

[5.2 在Ubuntu下链接的命令 - 23 -](#_Toc532238419)

[5.3 可执行目标文件hello的格式 - 24 -](#_Toc532238420)

[5.4 hello的虚拟地址空间 - 26 -](#_Toc532238421)

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

[5.6 hello的执行流程 - 29 -](#_Toc532238423)

[5.7 Hello的动态链接分析 - 29 -](#_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 - 33 -](#_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 - 39 -](#_Toc532238433)

[第7章 hello的存储管理 - 40 -](#_Toc532238434)

[7.1 hello的存储器地址空间 - 40 -](#_Toc532238435)

[7.2 Intel逻辑地址到线性地址的变换-段式管理 - 40 -](#_Toc532238436)

[7.3 Hello的线性地址到物理地址的变换-页式管理 - 42 -](#_Toc532238437)

[7.4 TLB与四级页表支持下的VA到PA的变换 - 44 -](#_Toc532238438)

[7.5 三级Cache支持下的物理内存访问 - 45 -](#_Toc532238439)

[7.6 hello进程fork时的内存映射 - 46 -](#_Toc532238440)

[7.7 hello进程execve时的内存映射 - 47 -](#_Toc532238441)

[7.8 缺页故障与缺页中断处理 - 47 -](#_Toc532238442)

[7.9动态存储分配管理 - 48 -](#_Toc532238443)

[7.10本章小结 - 50 -](#_Toc532238444)

[第8章 hello的IO管理 - 51 -](#_Toc532238445)

[8.1 Linux的IO设备管理方法 - 51 -](#_Toc532238446)

[8.2 简述Unix IO接口及其函数 - 51 -](#_Toc532238447)

[8.3 printf的实现分析 - 52 -](#_Toc532238448)

[8.4 getchar的实现分析 - 54 -](#_Toc532238449)

[8.5本章小结 - 54 -](#_Toc532238450)

[结论 - 55 -](#_Toc532238451)

[附件 - 56 -](#_Toc532238452)

[参考文献 - 57 -](#_Toc532238453)

# 第1章 概述

## 1.1 Hello简介

Hello 的 P2P，即 From Program to Process 的过程为：程序员通过键盘在程序编辑器（如Visual Studio、Code Blocks）输入hello.c 文件，然后经过预处理、汇编、编译、链接的一系列处理步骤，最终生成hello 可执行目标文件(Program)。

然后，程序员在 shell 中输入./hello 运行 hello 程序，shell 识别出这是一个外部命令，先调用 fork 函数创建了一个新的子进程(Process)，然后调用 execve函数在新的子进程中加载并运行 hello，运行 hello 还需要 CPU 为 hello 分配内存、时间片，使得 hello 看似独享 CPU 资源。在 hello 运行的过程中，CPU 要访问相关数据需要 MMU 的虚拟地址到物理地址的转化，其中 TLB 和四级页表为提高地址翻译的速度做出了巨大贡献，得到物理地址后三级 Cache 又帮助 CPU 快速得到需要的字节。系统的进程管理帮助 hello 切换上下文、shell 的信号处理程序使得 hello 在运行过程中可以处理各种信号，当程序员主动地按下 Ctrl+Z 或者 hello 运行到 return 0;时，hello 所在进程将被杀死，shell 会回收它的僵死进程。以上就是 hello 的 P2P（From Program to Process）、020（From Zero to Zero）的全过程。

## 1.2 环境与工具

### 1.2.1 硬件环境

CPU：Intel(R) Core(TM) i7-8550U CPU@ 1.80GHz 1.99GHz

内存：8.00GB（7.90GB可用）

磁盘： 360GB SSD + 500GB HDD

### 1.2.2 软件环境

Windows10 64位；

VMware Workstation Pro

Ubuntu 18.04 64位；

### 1.2.3工具

gedit，gcc, readelf, objdump, hexedit, edb

## 1.3 中间结果

|  |  |
| --- | --- |
| hello.c | hello源代码 |
| hello.i | 预处理之后的文本文件 |
| hello.s | hello的汇编代码 |
| hello1.s | hello.o的反汇编代码 |
| hello2.s | hello的反汇编代码 |
| hello.o | hello的可重定位文件 |
| hello | hello的可执行文件 |
| hello.elf | hello的elf文件 |
| hello1.elf | hello.o的elf文件 |

## 1.4 本章小结

本章介绍了 hello 的一生，即从一个源程序到预处理、编译、汇编、链接，最终成为一个可执行目标文件。本章还介绍了本次实验的环境以及用到的工具，以及从 hello.c 到 hello 的过程中产生的所有中间文件。

# 第2章 预处理

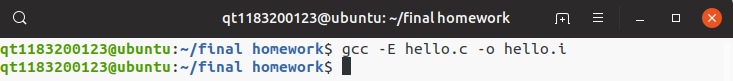
## 2.1 预处理的概念与作用

程序设计领域中，预处理一般是指在程序源代码被翻译为目标代码的过程中，生成二进制代码之前的过程。典型地，由预处理器(preprocessor，简记为cpp)对程序源代码文本进行处理，得到的结果再由编译器核心进一步编译。这个过程并不对程序的源代码进行解析，但它把源代码分割或处理成为特定的单位——（用C/C++的术语来说是）预处理记号(preprocessing token)用来支持语言特性（如C/C++的宏调用）。

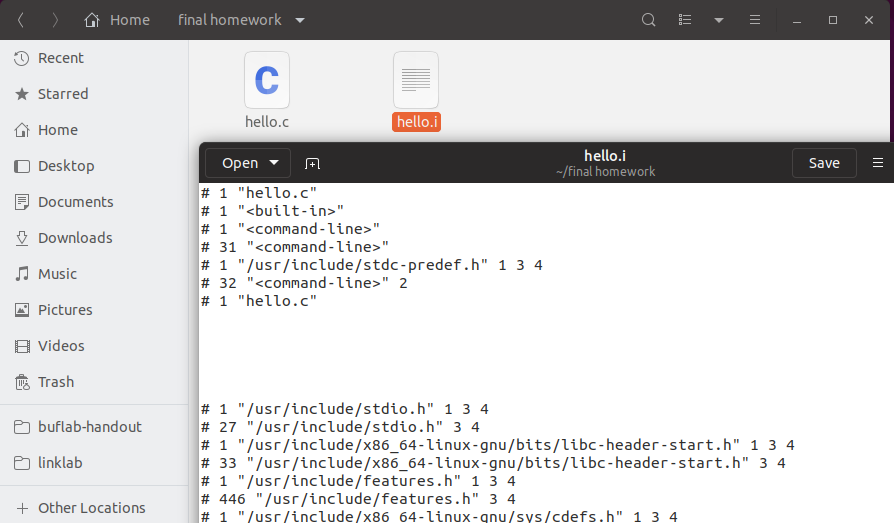
预处理器(cpp)根据以字符#开头的命令，修改原始的C程序。比如hello.c中第1行的#include <stdio.h>命令告诉预处理器读取系统头文件stdio.h的内容，把它直接插人程序文本中，并拓展所有用#define声明指定的宏。结果就得到了另一个C程序，通常是以.i作为文件扩展名。

## 2.2在Ubuntu下预处理的命令

linux> gcc -E hello.c -o hello.i



## 2.3 Hello的预处理结果解析



预处理器执行宏替换、条件编译以及包含指定的文件。

#include指令：用于在编译期间把置顶文件的内容包含进当前文件中；

#define指令：用任意字符序列替换一个标记；

2.3.1文件包含

#include指令是一个将所有声明放在一起的好方法，它保证所有的源文件都具有相同的定义与变量（函数）声明。如果某个包含文件的内容发生了变化，那么所以依赖于该包含文件的源文件都必须重新编译。

#include "file name"

#include <file name>

#include 记号序列

在源文件中，上面行将被替换为由文件名指定的文件内容。

如果文件名用引号引起来，则在源文件所在位置查找该文件；如果在该位置没有找到文件或如果文件名是尖括号括起来，则将根据相应的规则查找该文件，这个规则同具体实现有关。

2.3.2宏替换

预处理器把该标识符后续出现的各个实例用给定的记号序列替换。

#define 标识符 记号序列

#define 标识符（标识符表） 记号序列

#undef 标识符

第二次用#define指令定义同一标识符是错误的，除非第二次定义中的标记序列与第一次相同。通常情况下，#define指令占一行，但也可以把一个较长的宏定义分成若干行，这时需要在待续的行尾加上一个反斜杠\。 替换只对记号进行，对括在引号中的字符串不起作用。

如果在替换文本中，参数名以#作为前缀则结果将被扩展为由实际参数替换该参数的带引号的字符串。

#define DPRINT(expr) printf(#expr " = %g\n", expr)

DPRINT(x/y) => printf("x/y = %g\n", x/y)

预处理运算符##为宏扩展提供一种连接实际参数的手段。如果替换文本中的参数与##相邻，则该参数将被实际参数替换，##与前后的空白符将被删除，并对替换后的结果进行重新扫描。

#define PASTE(front, back) front##back

PASTE(fight, ing) => fighting

2.3.3条件包含

用于条件语句可以对预处理本身进行控制。

#if语句对其中的常量整形表达式进行求值，若该表达式的值不等于0，则包含其后的各行，直到遇到#endif、#elif或#else语句为止。

#if !defined(HDR)

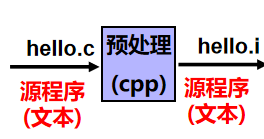
#define HDR

...

#endif

## 2.4 本章小结

本章介绍了预处理及其相关操作。预处理是hello.c源程序在编译系统里走过的第一程，它主要是拓展源代码，插入所有#include指定的文件，拓展#define声明等等。经过这个阶段，我们得到了一个修改了的源程序hello.i，它将是下一个阶段编译器（ccl）处理的对象。



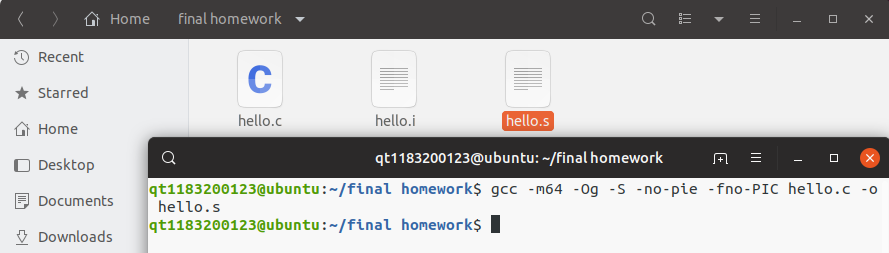
# 第3章 编译

## 3.1 编译的概念与作用

## 编译器（ccl）将文本文件hello.i编译成文本文件hello.s，它包含一个汇编语言程序。汇编语言程序中的每条语句都以一种标准的文本格式确切的描述一条低级机器语言指令。汇编语言是非常有用的，因为它为不同高级语言的不同编译器提供了通用的输出语言。例如，C编译器和Fortran编译器产生的输出文件用的都是一样的汇编语言

## 3.2 在Ubuntu下编译的命令

linux> gcc -m64 -Og -S -no-pie -fno-PIC hello.c -o hello.s



## 3.3 Hello的编译结果解析

所有以‘.’开头的行都是指导汇编器和链接器工作的伪指令

.file "hello.c" （源文件名）

.text （代码段）

.section .rodata.str1.8,"aMS",@progbits,1 （.rodata节）

.align 8 （对齐方式）

.LC0:

.string "\347\224\250\346\263\225: Hello \345\255\246\345\217\267 \345\247\223\345\220\215 \347\247\222\346\225\260\357\274\201" （字符串）

.section .rodata.str1.1,"aMS",@progbits,1 （.rodata节）

.LC1:

.string "Hello %s %s\n" （字符串）

.text （代码段）

.globl main （全局变量名）

.type main, @function （指定是对象类型或是函数类型）

下面分析C语言的数据与操作。

### 3.3.1 数据

C语言中的数据有：常量、变量、表达式、类型、宏。

**常量**大多是以立即数的形式出现在汇编代码hello.s中，

如：if(argc!=4)中的4，



再如：exit(1)中的1，

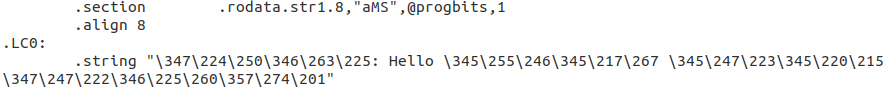


但有些也会有些小小的变化，如for(i=0;i<8;i++)中的8，编译器将i<8翻译成了i<=7.



**变量**分为全局变量、静态变量、局部变量。已初始化的全局变量和静态变量存放在.data节，未初始化的全局变量和静态变量存放在.bss节，而局部变量存放在栈中管理。

在我们的hello.c程序中，没有全局变量和静态变量，所以会发现在hello.s的最开始伪指令中没有.data和.bss。但用到了.rodata只读数据节，因为这里面存放了printf的格式串。例如：printf("用法: Hello 学号 姓名 秒数！\n")，它的汇编代码为movl $.LC0, %edi 和call puts，可以看到.LC0的string正是我们要输出的字符串，其中汉字转化成了相应的编码；



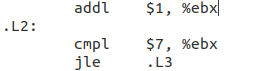
同理还有printf("Hello %s %s\n",argv[1],argv[2])，汇编代码为movl $.LC1, %esi、movl $1, %edi、movl $0, %eax和call \_\_printf\_chk，因为涉及到参数传递（我们会在后面讨论），所以过程稍微复杂些，不过此处我们只关心printf的格式串，它也在.rodata中。



for(i=0;i<8;i++)中的i是一个局部变量，编译器将它放在了寄存器%ebx中。首先将它初始化为0，



cmpl和addl分别实现比较和加一操作。



在hello.c程序中一共有3个**表达式**，分别为赋值表达式i=0，我们在上面已经分析过；两个关系表达式argc!=4和i<8，分别用cmpl $4, %edi和cmpl $7, %ebx（为什么是7不是8在上文中已解释）实现。

### 3.3.2 赋值

hello.c中只有一个赋值表达式i=0，它通过movl指令实现。



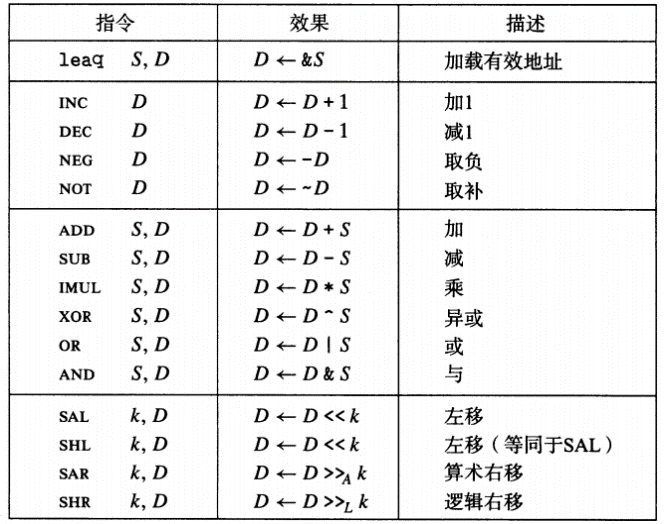
mov是一类指令，其他形式为：



还有movz（零拓展符号传送指令）和movs（符号拓展传送指令）

### 3.3.3 算术操作

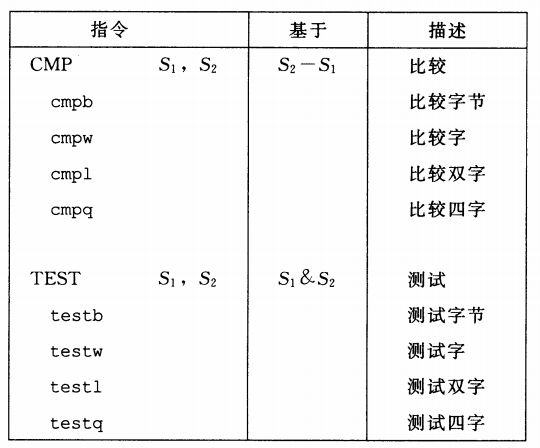
整数操作指令如下表：



在hello.s中我们用到了addl $1, %ebx，用以实现i++操作。

### 3.3.4 关系操作

常用的关系操作指令有cmp和test，具体使用规则如下：



它们只设置条件码而不改变任何其他寄存器。cmp指令根据两个操作数之差来设置条件码。TEST指令的行为与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。

在hello.s中我们使用了cmp指令来判断argc!=4和i<8，分别为cmpl $4, %edi和cmpl $7, %ebx（立即数7要和紧接着的jle指令结合着看）

### 3.3.5 控制转移

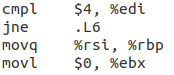
在这一部分经常用到jump指令，使用规则如下：



If语句。在hello.c中是if(argc!=4)

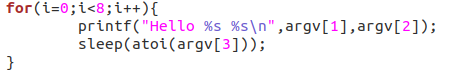


其实现如下：



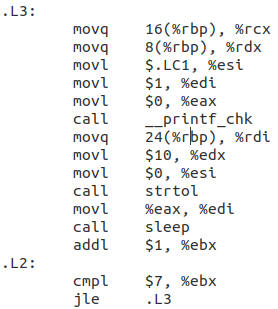
argc是main的第一个参数，所以在%edi中。首先使用cmpl指令比较argc和4的大小，设置条件码，然后jne根据条件码比较argc与4是否相等，并根据结果选择是否进行跳转：若不等，则跳转到.L6；若相等，则继续运行紧接着的下一条指令。

for循环。在hello.c中是



其实现如下：





首先是对i赋初值为0，然后跳转到循环判断表达式测试是否满足循环条件，如果是的话则跳转到循环表达式。汇编语言等价的goto版C语言如下：

int i=0;

goto test;

loop:

printf("Hello %s %s\n",argv[1],argv[2]);

sleep(atoi(argv[3]));

i++;

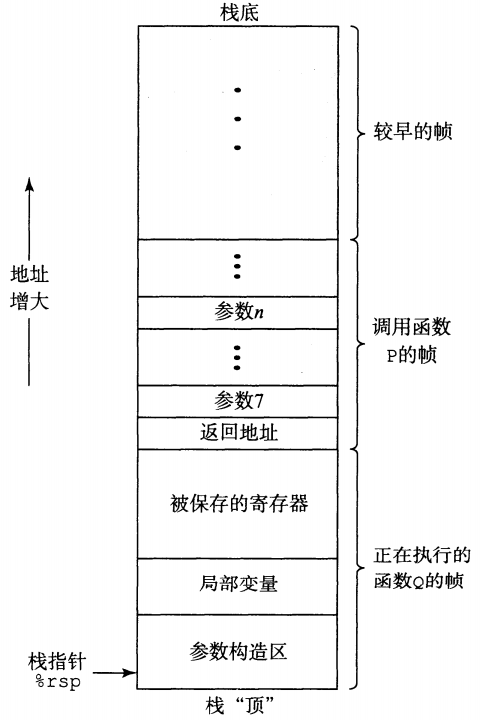
test:

if (i <=7)

goto loop;

### 3.3.6 函数调用

C语言过程调用机制的一个关键特性在于使用了栈数据结构提供的后进先出的内存管理原则。通用的栈帧结构如下：



函数P调用Q包括下面一个或多个机制：

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。该过程使用call和ret指令实现。

传递数据。P必须能够向Q提供一个或多个参数， Q必须能够向P返回一个值。P可以通过寄存器向Q传递6个参数，分别存放在%rdi、%rsi、%rdx、%rcx、%r8、%r9中，超出6个的部分就要通过栈来传递。

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

下面我们具体分析hello.s中的函数调用。先看几个简单的：

1、printf("用法: Hello 学号 姓名 秒数！\n")，对应的汇编语言如下：



直接将唯一的参数.LC0传递到%edi中，然后调用puts函数；

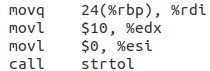
2、exit(1)，对应的汇编语言如下：



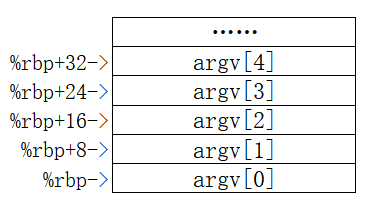
直接将唯一的参数1传递到%edi中，然后调用exit函数；

再看几个稍微复杂的：

3、atoi(argv[3])，对应的汇编语言如下：

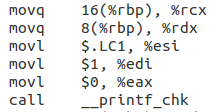


我们发现它传递了3个参数，第一个是24(%rbp)，也就是argv[3]，我们根据前面的movq %rsi, %rbp指令（将main的第2个参数argv[]的首地址传递给了%rbp）和main函数的栈帧可以得到这个结论，如下图所示：



调用strtol还传递了另外两个参数0和10，由于超出所学范围所以在此不再深究，只需知道传递了3个参数便可。

4、printf("Hello %s %s\n",argv[1],argv[2]) ，对应的汇编语言如下：



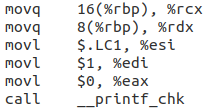
它传递了4个参数，第一个是%edi中的1，第2个时%esi中的.LC1，也就是之前分析过的在.rodata中的格式串，第3个是argv[1]，通过8(%rbp)计算得到（见上图），第4个是argv[2]，通过16(%rbp)计算得到（见上图）.

除了传递上述参数之外，函数在调用时通过call语句，将返回地址压入栈中，并将PC设置为调用函数的起始地址；结束时通过%rax作为返回值（如果需要的话），并通过ret指令从栈中弹出返回地址，并将PC设为返回地址。

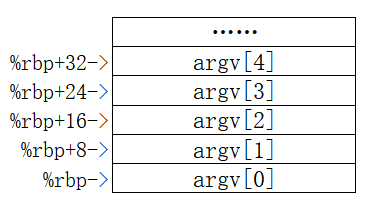
### 3.3.7 数组操作

C语言中的数组是一种将标量数据聚集成更大数据类型的方式。对于数据类型T和整型常数N，声明数组A[N]，起始位置表示为*x*0，表明它在内存中分配一个L·N字节的连续区域，这里L是数据类型T的大小；其次，它引入了标识符A，可以用A来作为指向数组开头的指针，这个指针的值就是*x*0，可以用0~N-1的整数索引来访间该数组元素。数组元素会被存放在地址为*x*0+L·*i*上的地方。

在本例中的数组是argv，这是一个指针数组，每个数组元素是一个指向参数字符串的指针。我们来看下如何调用argv的一个数组元素。C语句printf("Hello %s %s\n",argv[1],argv[2])的汇编代码为：

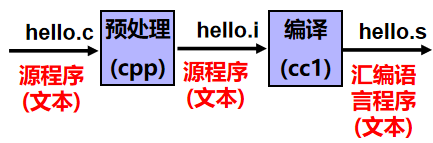


其中%rdx和%rcx分别向printf传递参数argv[1]和argv[2]



## 3.4 本章小结

本章介绍了编译及其相关操作。编译是将hello.i编译成汇编文本文件hello.s。汇编代码是机器代码的文本表示，给出程序中的每一条指令。阅读和理解汇编代码是一项很重要的技能，通过理解这些汇编代码，我们能够理解编译器的优化能力，并分析代码中隐含的低效率。



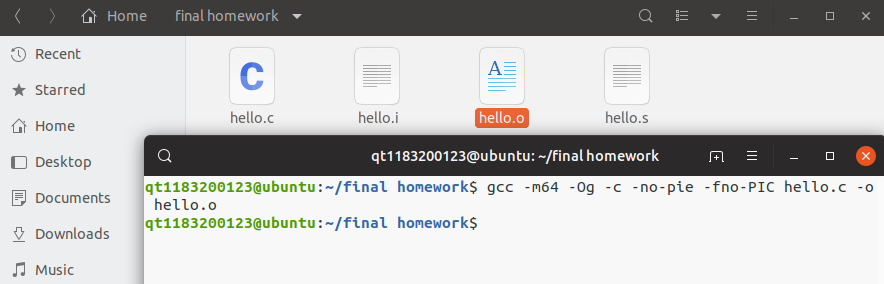
# 第4章 汇编

## 4.1 汇编的概念与作用

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(relocatable object program)的格式，并将结果保存在目标文件hello.o中。hello.o文件是一个二进制文件，它包含的是函数main的指令编码。如果我们在文本编辑器中打开hello.o文件,将看到一堆乱码。

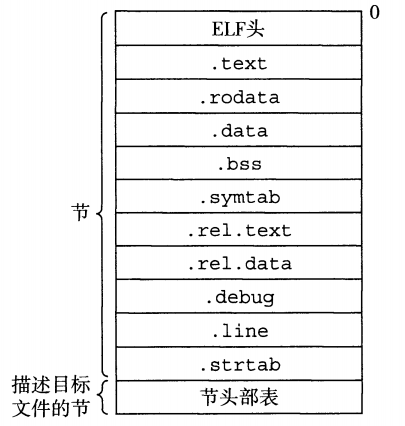
## 4.2 在Ubuntu下汇编的命令

linux> gcc -m64 -Og -c -no-pie -fno-PIC hello.c -o hello.o



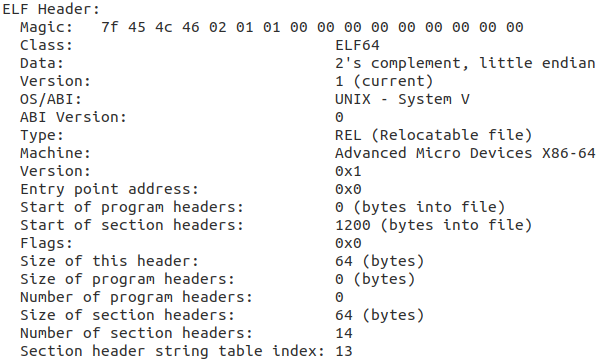
## 4.3 可重定位目标elf格式

可重定位目标文件的ELF格式如下：

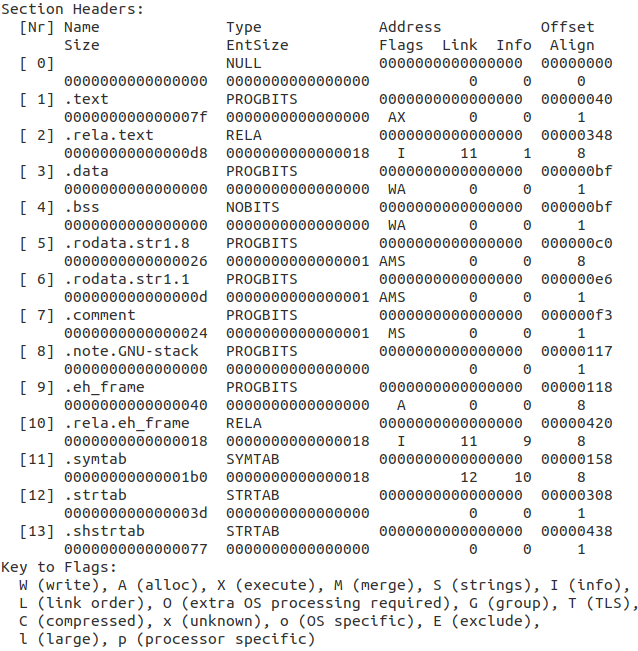


我们通过readelf工具，具体分析可重定位目标文件hello.o，看看这些节都做些什么。

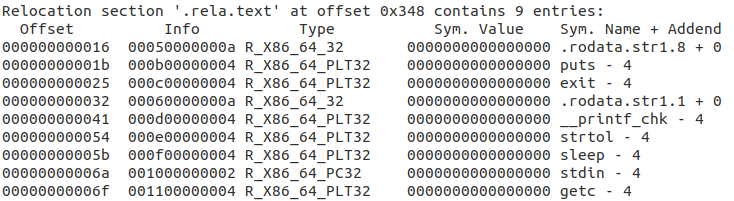
首先是**ELF头**，如下图：



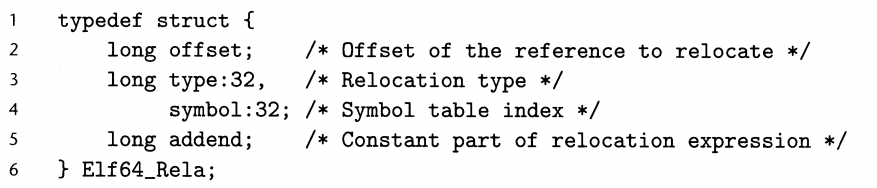
ELF头以一个16字节的序列开始，这个序列描述了生成该文件的系统的字的大小和字节顺序。ELF头剩下的部分包含帮助链接器语法分析和解释目标文件的信息。其中包括ELF头的大小，目标文件的类型（可重定位、可执行或者是共享的）、机器类型（如x86-64）、节头部表的文件偏移、以及节头部表中条目的大小和数量。



**重定位节.rela.text：**存放着代码的重定位条目。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

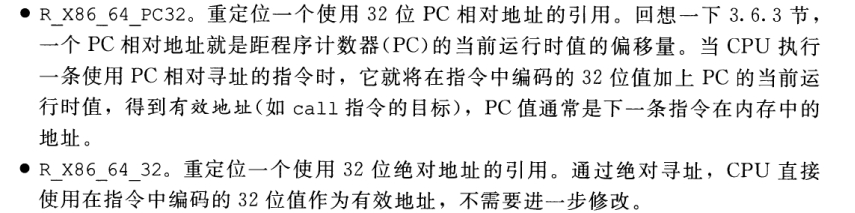


每一个重定位条目的数据结构如下：

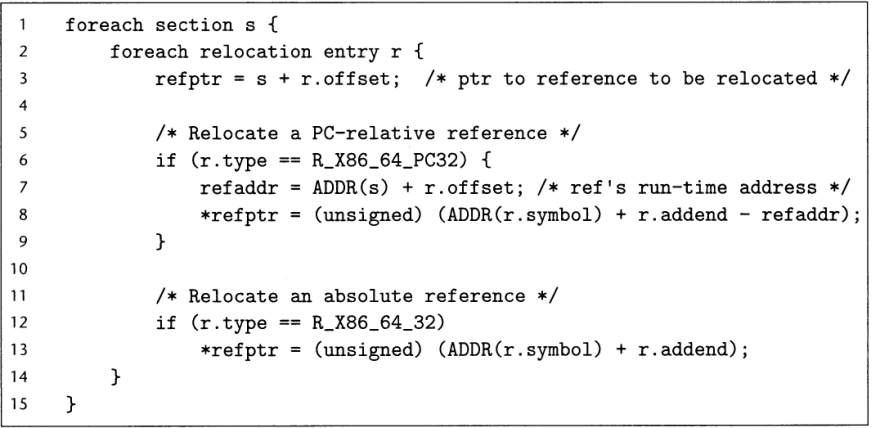


它包括offset需要被修改的引用的节偏移、symbol被修改引用应该指向的符号、type告知链接器如何修改新的引用、以及偏移调整addend

在这里我们关心两种重定位类型中：R\_X86\_64\_PC32和R\_X86\_64\_32



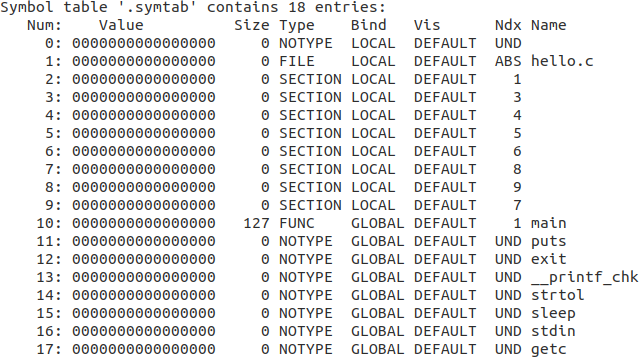
CSAPP中也给出了重定位的计算方法：



**重定位节.rela.eh\_frame：**eh\_frame 节的重定位信息

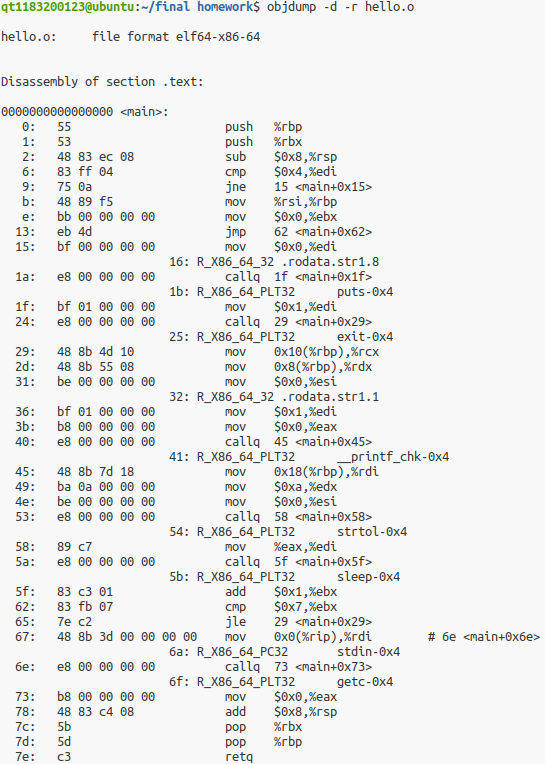


**符号表：**用来存放程序中定义和引用的函数和全局变量的信息。注意，符号表不包含局部变量的条目。



## 4.4 Hello.o的结果解析

hello.o的反汇编如下：



### 4.4.1 机器语言的构成

x86-64的指令长度从1到15个字节不等。常用的指令以及操作数较少的指令所需的字节数少，例如pop %rbx只需一个字节5b，而那些不太常用或操作数较多的指令所需字节数较多；

设计指令格式的方式是，从某个给定位置开始，可以将字节唯一地解码成机器指令。例如，只有指令pushq %rbx是以字节值53开头的；

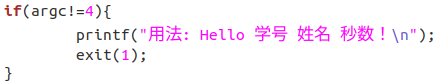
如果指令中有地址或者常数，要按小端存储顺序依次存放。

### 4.4.2 与汇编语言的映射关系

反汇编器只是基于机器代码文件中的字节序列来确定汇编代码。它不需要访问该程序的源代码或汇编代码；反汇编器使用的指令命名规则与GCC生成的汇编代码使用的有些细微的差别。例如：省略了很多指令结尾的‘q’.

在函数调用和分支跳转时，二者也是有差别的。

分支跳转时，如判断argc!=4，然后跳转



hello.s中是jump到.L6，用一个标记指示；

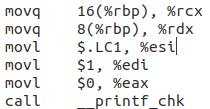


hello.o的反汇编是跳转到地址15

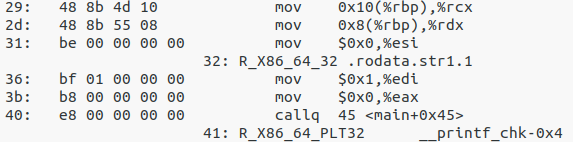


函数调用时，如printf("Hello %s %s\n",argv[1],argv[2])

hello.s中是将参数传递到寄存器之后，call后紧跟的是调用函数明



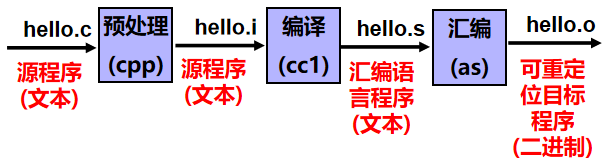
而hello.o的反汇编中call后紧跟的是45 <main+0x45>，并且还有一个重定位条目，用于链接时重定位。



值得注意的是，printf的在.rodata中的格式串也不再表示为.LC1，而用mov $0x0 , %esi表示，后面也跟着一个重定位条目，用于链接时重定位。

## 4.5 本章小结

本章介绍了汇编及其过程。汇编器将汇编语言翻译成机器语言指令，把这些指令打包可重定位目标程序，并将结果保存在hello.o中。hello.o文件是一个二进制文件，它包含的是函数main的指令编码。多个可重定位目标文件可以在链接时合并在一起，创建一个可执行的目标文件。



# 第5章 链接

## 5.1 链接的概念与作用

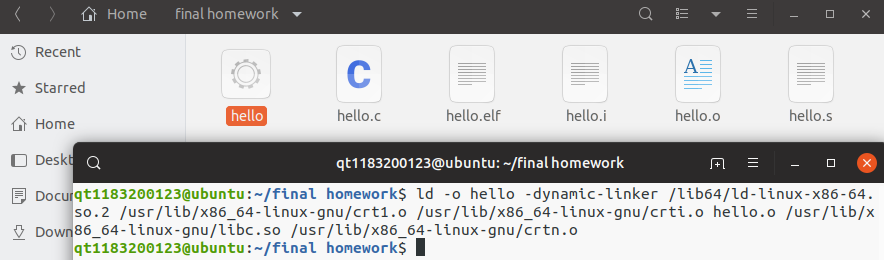
链接是将各种代码和数据片段收集并组合成为一个单一文件的过程，这个文件可被加载（复制）到内存并执行。链接可以执行于编译时、加载时、运行时。例如： hello程序调用了printf函数，它是每个C编译器都提供的标准C库中的一个函数。printf函数存在于一个名为printf.o的单独的预编译好了的目标文件中，而这个文件必须以某种方式合并到我们的hello.o程序中。链接器(ld)就负责处理这种合并。结果就得到hello文件,它是一个可执行目标文件

链接使得分离编译成为可能。我们不用将一个大型的应用程序组织为一个巨大的源文件，而是可以把它分解为更小、更好管理的模块，可以独立地修改和编译这些模块。当我们改变这些模块中的一个时，只需简单地重新编译它，并重新链接，而不必重新其他文件

## 5.2 在Ubuntu下链接的命令

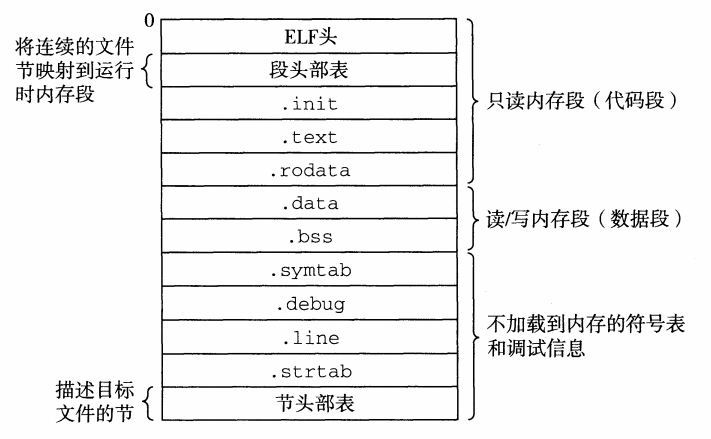
linux> ld -o hello -dynamic-linker /lib64/ld-linux-x86-64.so.2

/usr/lib/x86\_64-linux-gnu/crt1.o/usr/lib/x86\_64-linux-gnu/crti.o hello.o

/usr/lib/x86\_64-linux-gnu/libc.so/usr/lib/x86\_64-linux-gnu/crtn.o

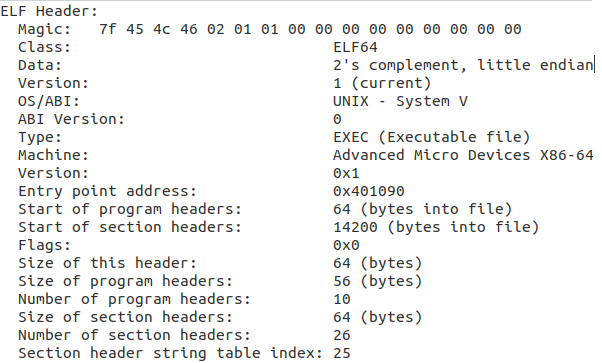
## 5.3 可执行目标文件hello的格式

可执行目标文件的格式类似于可重定位目标文件，典型的ELF可执行目标文件格式如下图所示：



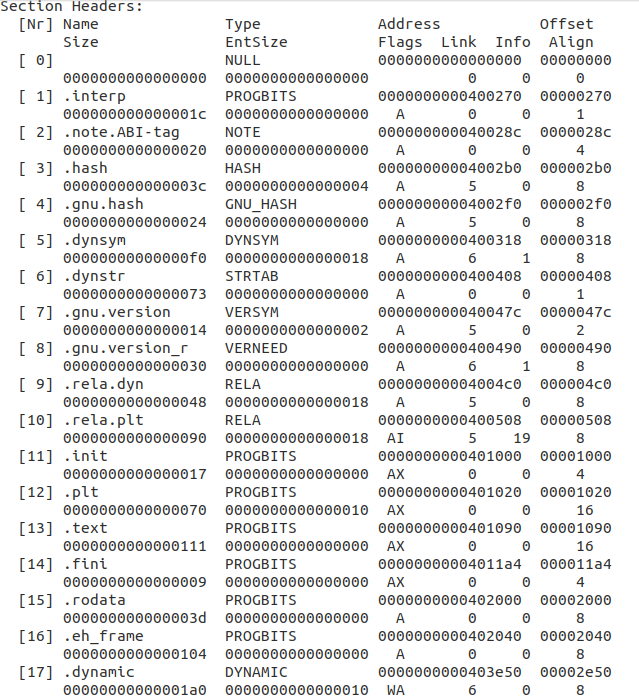
使用readelf -a hello > hello.elf 命令生成hello 程序的ELF 格式文件。

首先是**ELF头**，描述文件的总体格式，还包括程序的入口点。



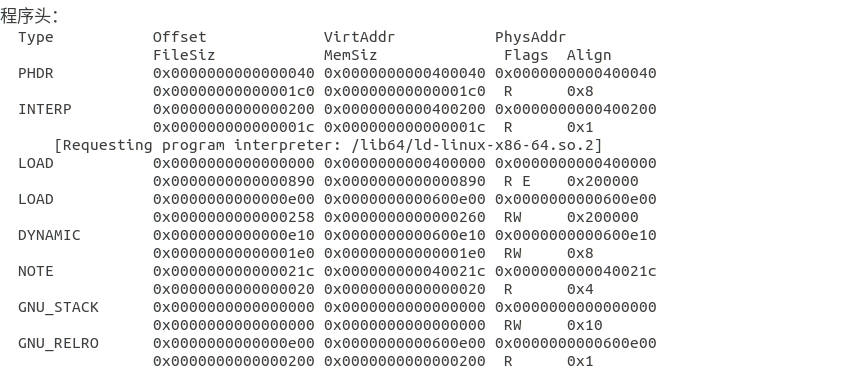
.text、.rodata和.data节与可重定位目标文件中的节是类似的，除了这些节已经被重定位到它们最终的运行时的内存地址外。.init节定义了一个小函数\_init，程序初始化代码会调用它。因为可执行文件时完全连接的，所以无.rel节。

**节头**记录每个节的名称、偏移量、大小、位置等信息





程序头部表描述了可执行文件的连续的片被映射到连续的内存段的映射关系。每一个表项提供了各段在虚拟地址空间大小和物理地址，标志，访问权限和对齐方式。我们可以以此读出各段的起始地址。

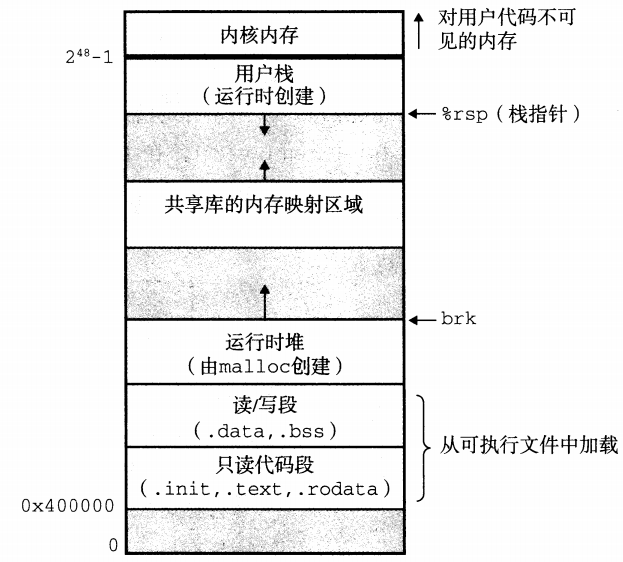


|  |  |
| --- | --- |
| **PHDR** | **保存程序头表** |
| **INTERP** | 指定在程序已经从可执行文件映射到内存之后，必须调用的解释器（如动态链接器）。 |
| **LOAD** | 表示一个需要从二进制文件映射到虚拟地址空间的段。其中保存了常量数据（如字符串）、程序的目标代码等。 |
| **DYNAMIC** | 保存了由动态链接器使用的信息 |
| **NOTE** | 保存辅助信息 |
| **GNU\_STACK** | 权限标志，标志栈是否是可执行的 |
| **GNU\_RELRO** | 指定在重定位结束之后那些内存区域是需要设置只读 |

和可重定位目标文件相比，我们发现各表中都多出了许多条目，这是因为链接过程中链接了许多其他文件，如库函数等。

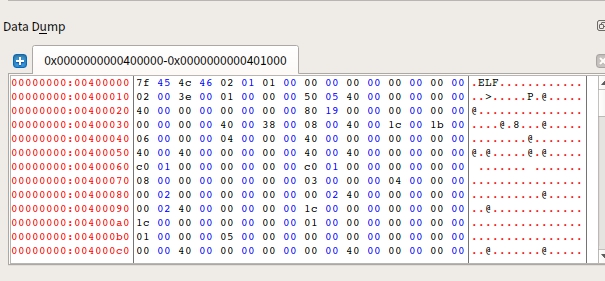
## 5.4 hello的虚拟地址空间

hello的Linux x86-64内存映像如下图所示。



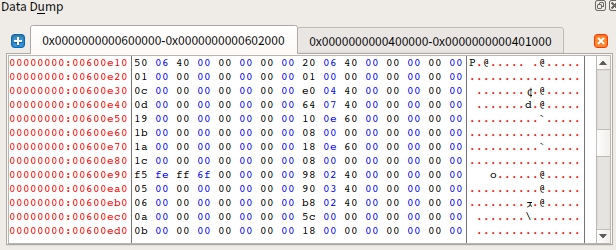
使用edb加载hello，查看本进程的虚拟地址空间各段信息。

首先从0x400000开始查看，我们发现第一行竟然和ELF头中的Magic是相同的，这绝非偶然，这说明程序是从0x400000处开始加载的



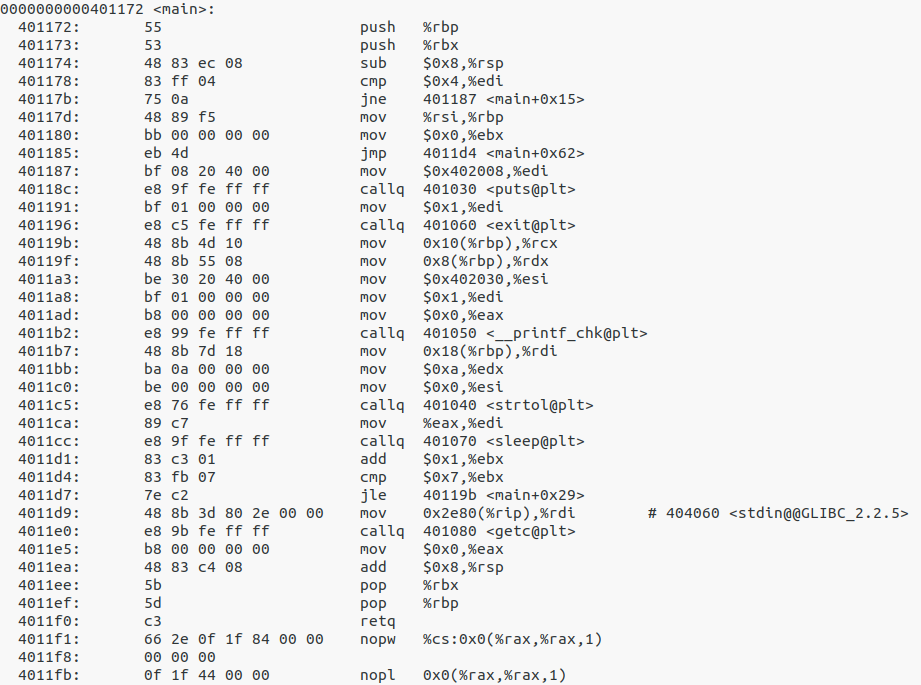


再看数据段，它是从0x600e10的。

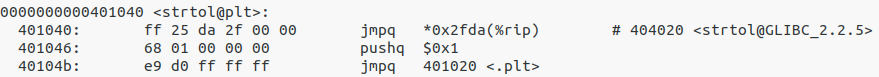


## 5.5 链接的重定位过程分析

使用objdump -d -r hello 指令反汇编hello。我们看到里面链接了如\_init、puts@plt、\_\_printf\_chk@plt等许多函数，在此只分析我们的main函数，来看一下可重定位hello.o与可执行文件hello间的区别。



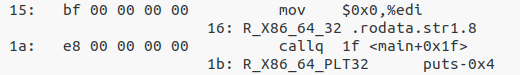
我们看到call语句中都跟上了调用函数的地址，它们是PC相对引用的。举个例子。我们看调用strtol函数，它的地址是0x401040，而call语句后紧跟的下一条指令地址是0x4011ca，两者相减0x401040-0x4011ca得0xfffffe76，再根据小端法得76 fe ff ff 这便是机器指令中的字节表示了



我们再看对数据的重定位。printf语句中的格式串也用其地址做为参数传递，



而不是像在hello.o中用0占位和一个重定位条目表示了。



## 5.6 hello的执行程

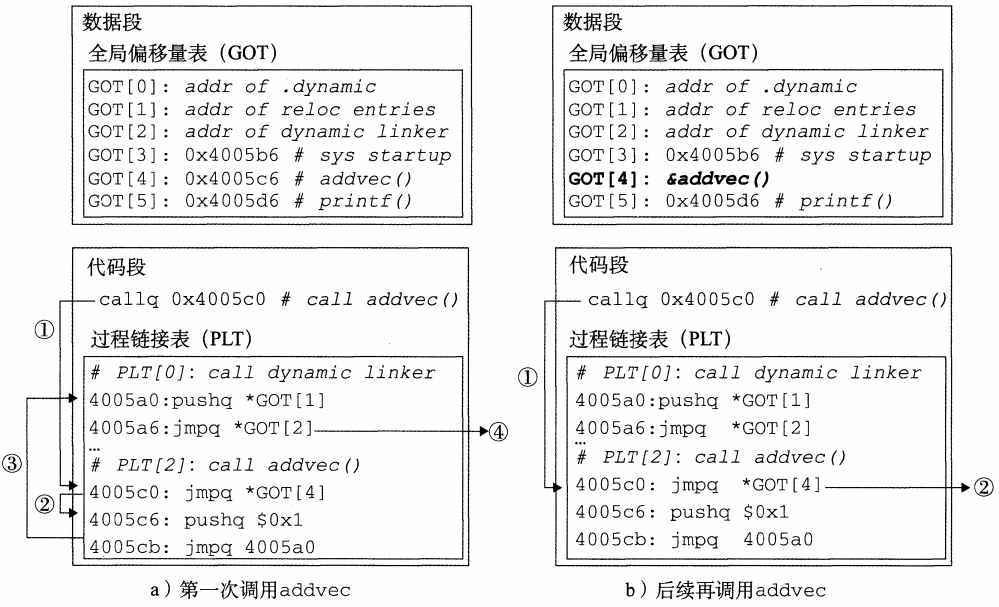
使用edb单步调试运行程序，观察其调用的函数,这里可以发现在调用main之前主要进行了初始化的工作调用了\_init，在这个函数之后动态链接的重定位工作已经完成，我们可以看到在这个函数的调用之后是一系列在这个程序中所用到的库函数（printf，exit，atoi等等）这些函数实际上在代码段并不占用实际的空间只是一个占位的符号，实际上他们的内容在共享区（高地址）处。之后调用了\_start这个就是起始的地址，准备开始执行main的内容，main函数内部所调用的函数在第三章已经进行了充分的分析这里略过main内部的函数，在执行main之后还会执行\_\_libc\_csu\_init 、\_\_libc\_csu\_fini 、\_fini 最终这个程序才结束。

下面列出了各个函数的名称与地址（有些没有具体进入分析，只体现了hello的执行的一个大概的流程）

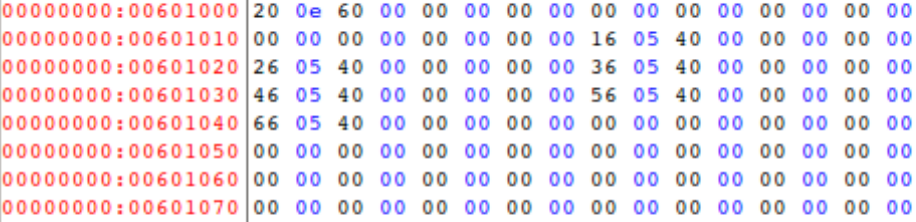
1. \_init <0x00000000004004e0>
2. puts@plt <0x0000000000400510>
3. printf@plt <0x0000000000400520>
4. getchar@plt <0x0000000000400530>
5. atoi@plt <0x0000000000400540>
6. exit@plt <0x0000000000400550>
7. sleep@plt <0x0000000000400560>
8. \_start <0x0000000000400570>
9. \_dl\_relocate\_static\_pie <0x00000000004005a0>
10. deregister\_tm\_clones <0x00000000004005b0>
11. register\_tm\_clones <0x00000000004005e0>
12. \_\_do\_global\_dtors\_aux <0x0000000000400620>
13. frame\_dummy <0x0000000000400650>
14. main <0x0000000000400657>
15. \_\_libc\_csu\_init <0x00000000004006f0>
16. \_\_libc\_csu\_fini <0x0000000000400760>
17. \_fini <0x0000000000400764>

## 5.7 Hello的动态链接分析

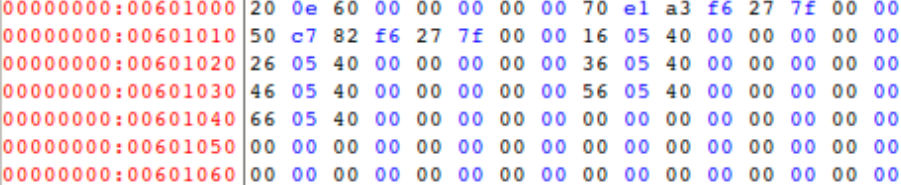
对于动态共享链接库中 PIC 函数，编译器没有办法预测函数的运行时地址，所以需要添加重定位记录，等待动态链接器处理，GNU 编译系统使用延迟绑定技术，将过程地址的绑定推迟到第一次调用该过程时。使用延迟绑定的动机是对于一个像 libc.so 这样的共享库输出的成百上千个函数中，一个典型的应用程序只会使用其中很少的一部分。把函数地址的解析推迟到它实际被调用的地方，能避免动态链接器在加载时进行成百上千个其实并不需要的重定位。第一次调用过程的运行时开销很大，但是其后的每次调用都只会花费一条指令和一个间接的内存引用。延迟绑定是通过两个数据结构的交互来实现的，这两个数据结构是 GOT(全局偏移量表)和 PLT（过程链接表）。如果一个目标模块调用定义在共享库中的任何函数，那么它就有自己的 GOT 和 PLT。GOT 是数据段的一部分，PLT 是代码段的一部分。下图介绍了 GOT 和 PLT 交互的一个例子。只需注意：GOT 和 PLT 联合使用时，GOT[0]和 GOT[1]包含动态连接器在解析函数地址时会使用的信息。GOT[2]是动态连接器在 ld-linux.so 模块中的入口点。



接下来观察 dl\_init 前后动态链接项目的变化。.got.plt 节的起始地址是 0x601000，在 DataDump 中找到该位置，



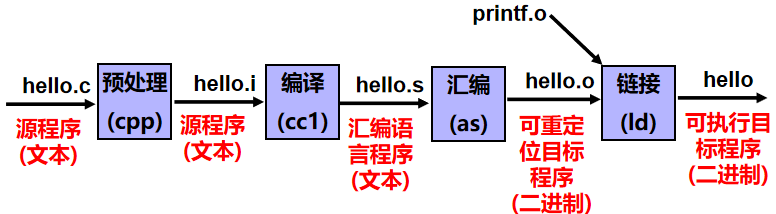
使用 edb 执行至 dl\_init，按 F8，发现地址 0x601000 后发生了变化：



可以看到 dl\_init 后出现了两个地址0x7f27f6a3el70和0x7f27f682c750，这便是 GOT[1]和 GOT[2]。

## 5.8 本章小结

本章介绍了链接及其过程。链接是将各种代码和数据片段收集并组合成为一个单一文件的过程，这个文件可被加载（复制）到内存并执行。链接可以自编译时由静态编译器来完成，也可以在加载时和运行时由动态链接器来完成。链接器处理3种形式的目标文件，分别为可重定位的、可执行的和共享的。链接器的两个主要任务是符号解析和重定位。



# 第6章 hello进程管理

## 6.1 进程的概念与作用

### 6.1.1 进程的概念

一个执行中的程序的实例，同时也是系统进行资源分配和调度的基本单位。一般情况下，包括文本区域、数据区域和堆栈。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储着活动过程调用的指令和本地变量。

### 6.1.2 进程的作用

给应用程序提供两个关键抽象：一个独立的逻辑流，它提供一个假象，好像我们的程序独占地使用处理器；一个私有的地址空间，它提供一个假象，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

Linux系统中，Shell是一个交互型应用级程序，代表用户运行其他程序(是命令行解释器，以用户态方式运行的终端进程)。Shell有许多形式1)sh 最早的shell； 2）csh/tcsh 变种；3)bash变种、缺省的Linux shell

其基本功能是解释并运行用户的指令，重复如下处理过程：

(1)终端进程读取用户由键盘输入的命令行。

(2)分析命令行字符串，获取命令行参数，并构造传递给execve的argv向量

(3)检查第一个(首个、第0个）命令行参数是否是一个内置的shell命令

(3)如果不是内部命令，调用fork( )创建新进程/子进程

(4)在子进程中，用步骤2获取的参数，调用execve( )执行指定程序。

(5)如果用户没要求后台运行(命令末尾没有&号）否则shell使用waitpid（或wait...)等待作业终止后返回。

(6)如果用户要求后台运行(如果命令末尾有&号），则shell返回；

## 6.3 Hello的fork进程创建过程

首先，要运行hello 程序，需要在shell 输入./hello 1183200123 祁天 1

接下来 shell 会分析这一串命令：

1. 先判断./hello 是否是内置命令，结果是它不是内置命令

2. 然后 shell 调用 fork()函数，创建一个子进程，这个子进程与父进程几乎没有差别，子进程的虚拟地址空间均与父进程的映射关系一致，是父进程虚拟地址空间的一份副本，包括代码和数据段、堆、共享库以及用户栈。同时，子进程还获得与父进程任何打开文件描述符相同的副本，故此时子进程可以读写父进程打开的任何文件。子进程与父进程的最大差别在于它们有不同的 PID。

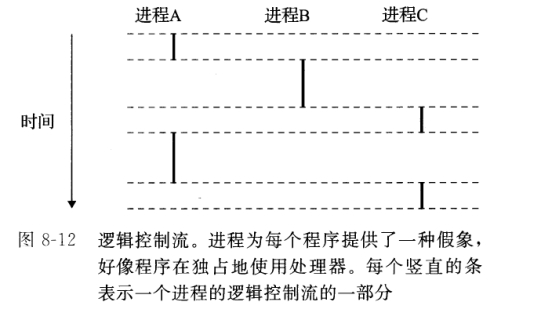
3. 接下来 hello将加载到fork 创建的子进程中，然后执行。

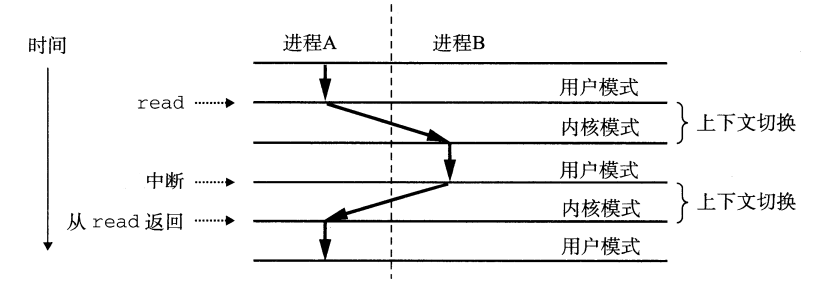
## 6.4 Hello的execve过程

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

## 6.5 Hello的进程执行

在操作系统中，每一个时刻通常有许多程序在进行，但是我们通常会认为每一个进程都独立占用CPU内存以一些其他资源，如果单步调试我们的程序可以发现在执行时一系列的（PC）程序计数器的值，这个PC值的序列就是逻辑控制流，事实上，多个程序在计算机内部执行时，采用并行的方式，他们的执行是交错的，像下图每个程序都交错运行一小会儿，进程是轮流使用处理器的，在同一个处理器核心中，每个进程执行它的流的一部分后被抢占（暂时挂起），然后轮到其他进程。



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

为了使操作系统内核提供一个无懈可击的进程抽象，处理器必须提供一种机制，限制一个应用可以执行的指令以及它可以访问的地址空间范围。处理器通常使用某个控制寄存器的一个模式位提供两种模式的区分，该寄存器描述了进程当前享有的特权，当没有设置模式位时，进程就处于用户模式中，用户模式的进程不允许执行特权指令，也不允许直接引用地址空间中内核区内的代码和数据；设置模式位时，进程处于内核模式，该进程可以执行指令集中的任何命令，并且可以访问系统中的任何内存位置。 接下来分析 hello 的进程调度，hello的一个上下文切换是调用sleep函数时，hello 显式地请求休眠，控制转移给另一个进程，此时计时器开始计时，当计时器到达argv[3]，即1s时，它会产生一个中断信号，中断当前正在进行的进程，进行上下文切换，恢复 hello 在休眠前的上下文信息，控制权回到 hello 继续执行。当循环结束后，hello 调用 getchar 函数，之前 hello 运行在用户模式下，在调用 getchar 时进入内核模式，内核中的陷阱处理程序请求来自键盘缓冲区的 DMA传输，并执行上下文切换，并把控制转移给其他进程。当完成键盘缓冲区到内存的数据传输后，引发一个中断信号，此时内核从其他进程切换回 hello 进程，然后 hello执行 return，进程终止。

## 6.6 hello的异常与信号处理

hello 执行过程中可能出现四类异常：中断、陷阱、故障和终止。

1. 中断是来自 I/O 设备的信号，异步发生，中断处理程序对其进行处理，返 回后继续执行调用前待执行的下一条代码，就像没有发生过中断。

2. 陷阱是有意的异常，是执行一条指令的结果，调用后也会返回到下一条指 令，用来调用内核的服务进行操作。帮助程序从用户模式切换到内核模式。

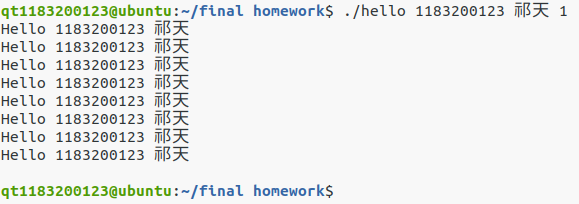
3. 故障是由错误情况引起的，它可能能够被故障处理程序修正。如果修正成 功，则将控制返回到引起故障的指令，否则将终止程序。

4. 终止是不可恢复的致命错误造成的结果，通常是一些硬件的错误，处理程 序会将控制返回给一个 abort 例程，该例程会终止这个应用程序。

下面分析hello执行过程中对各种异常和信号的处理。

1)正常运行

程序正常执行，总共循环8次每次输出提示信息之后等待我们从命令行输入的秒数，最后需要输入一个字符回车结束程序。



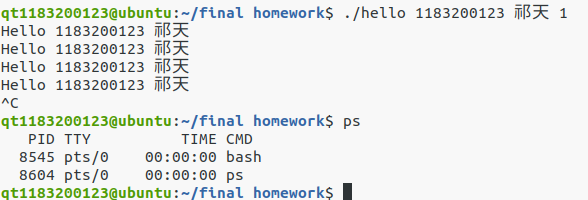
2)中途按下ctrl-Z

内核向前台进程发送一个SIGSTP信号，前台进程被挂起，直到通知它继续的信号到来，继续执行。当按下fg 1 后，输出命令行后，被挂起的进程从暂停处，继续执行。



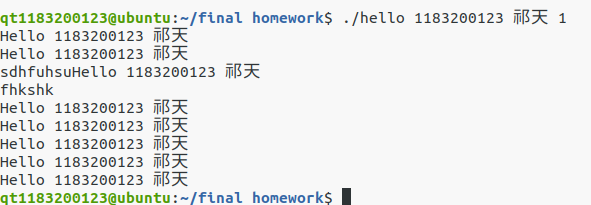
3)中途按下ctrl-C

内核向前台进程发送一个SIGINT信号，前台进程终止，内核再向父进程发送一个SIGCHLD信号，通知父进程回收子进程，此时子进程不再存在



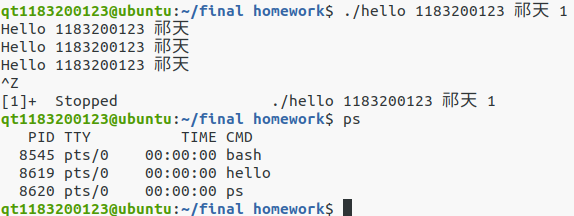
4)运行途中乱按

运行途中乱按后，只是将乱按的内容输出，程序继续执行，但是我们所输入的内容到第一个回车之前会当做getchar缓冲掉，后面的输入会简单的当做我们即将要执行的命令出现在shell的命令行处。



5)输入ps打印前台进程组

ps打印当前进程的状态

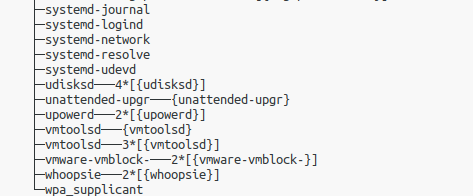


6)打印进程树

详细显示从开机开始的各个进程父子关系，以一颗树的形式展现。





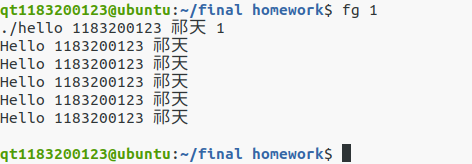


7)列出jobs当前的任务

jobs，打印进程状态信息



8)输入fg 1，继续执行前台进程1



9)输入kill

kill之后会根据不同的发送信号的值，以及要发送的进程的pid发送相应的信号，这里我们将hello杀死。



## 6.7本章小结

本章介绍了hello的进程管理。进程是一个执行中的程序的实例，它提供两个关键抽象：好像我们的程序独占地使用处理器，好像我们的程序独占地使用内存系统。每个进程都处在某个进程的上下文中，每个进程也都有属于自己的上下文，用于操作系统通过上下文切换进行进程调度。用户通过shell和操作系统交互，向内核提出请求，shell通过fork函数和execve函数来运行可执行文件。

# 第7章 hello的存储管理

## 7.1 hello的存储器地址空间

**物理地址**(physical address)用于内存芯片级的单元寻址，与处理器和CPU连接的地址总线相相应。是指出目前CPU外部地址总线上的寻址物理内存的地址信号，是地址变换的最终结果地址。如果启用了分页机制，那么线性地址会使用页目录和页表中的项变换成物理地址。如果没有启用分页机制，那么线性地址就直接成为物理地址了。

**逻辑地址**（Logical Address）是指由程式产生的和段相关的偏移地址部分。表示为 [段标识符：段内偏移量]。

**线性地址**（Linear Address）是逻辑地址到物理地址变换之间的中间层。程式代码会产生逻辑地址，或说是段中的偏移地址，加上相应段的基地址就生成了一个线性地址。如果启用了分页机制，那么线性地址能再经变换以产生一个物理地址。若没有启用分页机制，那么线性地址直接就是物理地址。

**虚拟地址**（Virtual Address）虚拟内存为每个程序提供了一个大的、一致的和私有的地址空间。其每个字节对应的地址成为虚拟地址。虚拟地址包括 VPO（虚拟页面偏移量）、VPN（虚拟页号）、TLBI（TLB 索引）、TLBT（TLB 标记）。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

逻辑地址空间表示：段地址：偏移地址。段地址+偏移地址=线性地址

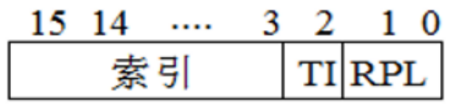
在实模式下：逻辑地址 CS：EA=CS\*16+EA 物理地址

在保护模式下：以段描述符作为下标，到 GDT/LDT 表查表获得段地址，

段内偏移量是在链接后就已经得到的 32 位地址，因此要想由逻辑地址得到线性地址，需要根据逻辑地址的前 16 位获得段地址，这 16 位存放在段寄存器中。

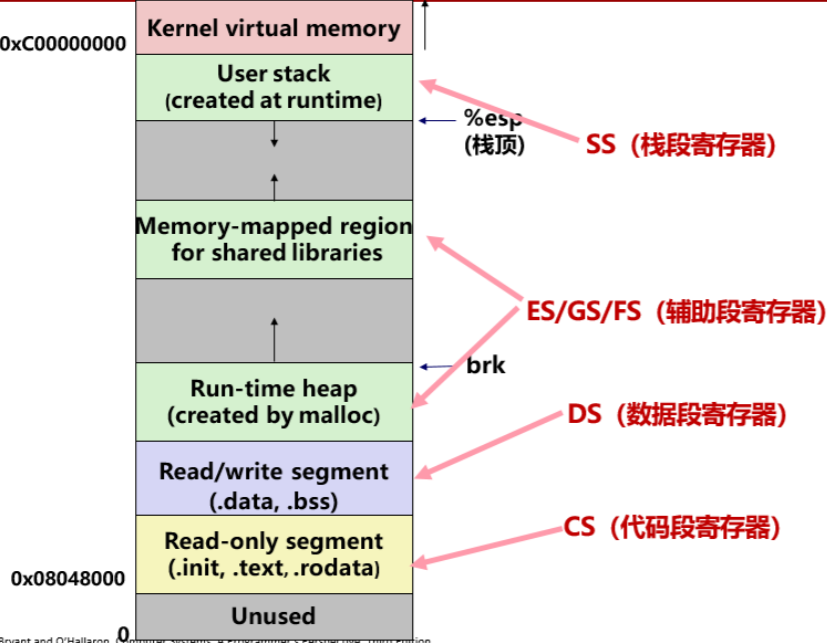
段寄存器（16 位）：用于存放段选择符。CS（代码段）：程序代码所在段；SS（栈段）：栈区所在段；DS（数据段）：全局静态数据区所在段；其他三个段寄存器 ES、GS 和 FS 可指向任意数据段。

段选择符中字段的含义：



其中 CS 寄存器中的 RPL 字段表示 CPU 的当前特权级

TI=0，选择全局描述符表（GDT）；TI=1，选择局部描述符表（LDT）；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。高 13 位-8K 个索引用来确定当前使用的段描述符在描述符表中的位置。



段描述符是一种数据结构，等价于段表项，分为两类。一类是用户的代码段和数据段描述符，一类是系统控制段描述符。

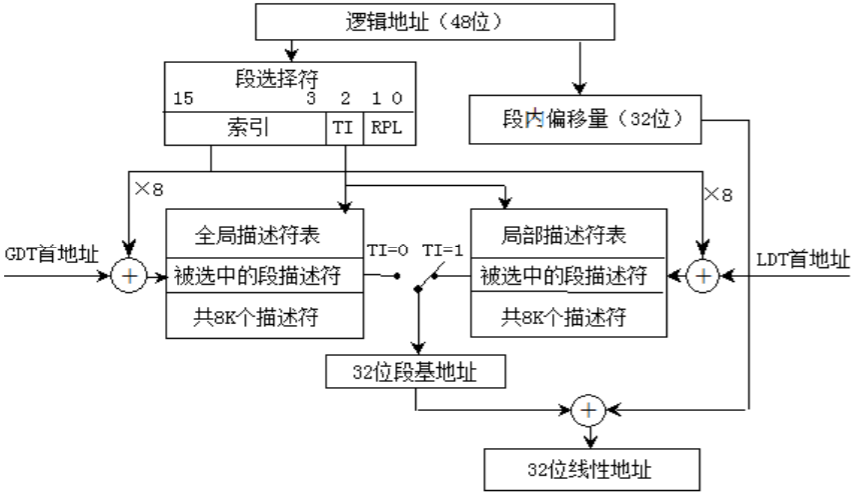
描述符表：实际上为段表，由段描述符（段表项构成）分为三种类型：

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

下图展示了逻辑地址到线性地址的转化过程：



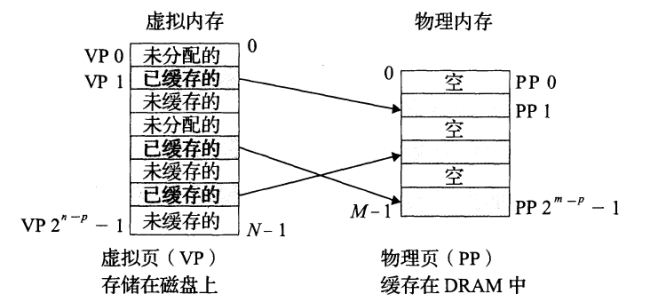
首先根据段选择符的 TI 部分判断需要用到的段选择符表是全局描述符表还是局部描述符表，随后根据段选择符的高 13 位的索引（描述符表偏移）到对应的描述符表中找到对应的偏移量的段描述符，从中取出 32 位的段基址地址，将 32 位的段基址地址与 32 位的段内偏移量相加得到 32 位的线性地址。

## 7.3 Hello的线性地址到物理地址的变换-页式管理

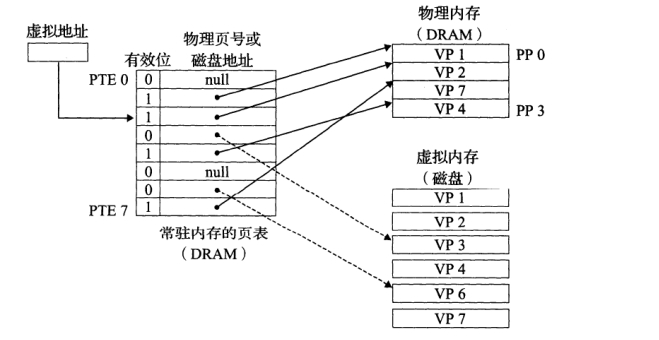
Linux下，虚拟地址到物理地址的转化与翻译是依靠页式管理来实现的，虚拟内存作为内存管理的工具。概念上而言，虚拟内存被组织为一个由存放在磁盘上的N个连续的字节大小的单元组成的数组。磁盘上数组的内容被缓存在物理内存中(DRAM cache)这些内存块被称为页 (每个页面的大小为P = 2*p*字节)。

而分页机制的作用就是通过将虚拟和物理内存分页，并且通过MMU建立起相应的映射关系，可以充分利用内存资源，便于管理。一般来说一个页面的标准大小是4KB，有时可以达到4MB。而且虚拟页面作为磁盘内容的缓存，有以下的特点：DRAM缓存为全相联，任何虚拟页都可以放置在任何物理页中需要一个更大的映射函数，不同于硬件对SRAM缓存更复杂精密的替换算法太复杂且无限制以致无法在硬件上实现DRAM缓存总是使用写回，而不是直写。

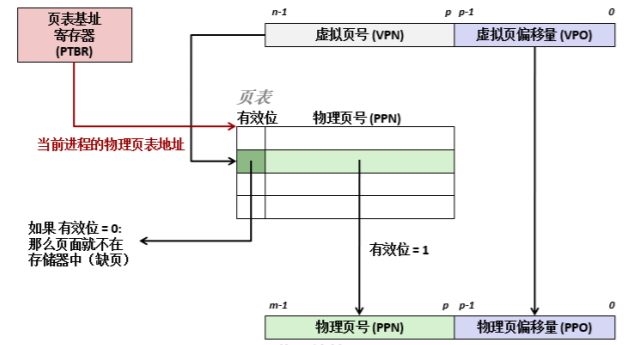
虚拟页面地集合被分为三个不相交的子集：已缓存、未缓存和未分配。



**页表**实现从虚拟页到物理页的映射，依靠的是页表，页表就是是一个页表条目 (Page Table Entry, PTE)的数组，将虚拟页地址映射到物理页地址。这个页表是常驻与主存中的。



下图展示了页式管理中虚拟地址到物理地址的转换：



下图a展示了当页面命中时，CPU硬件执行的步骤：

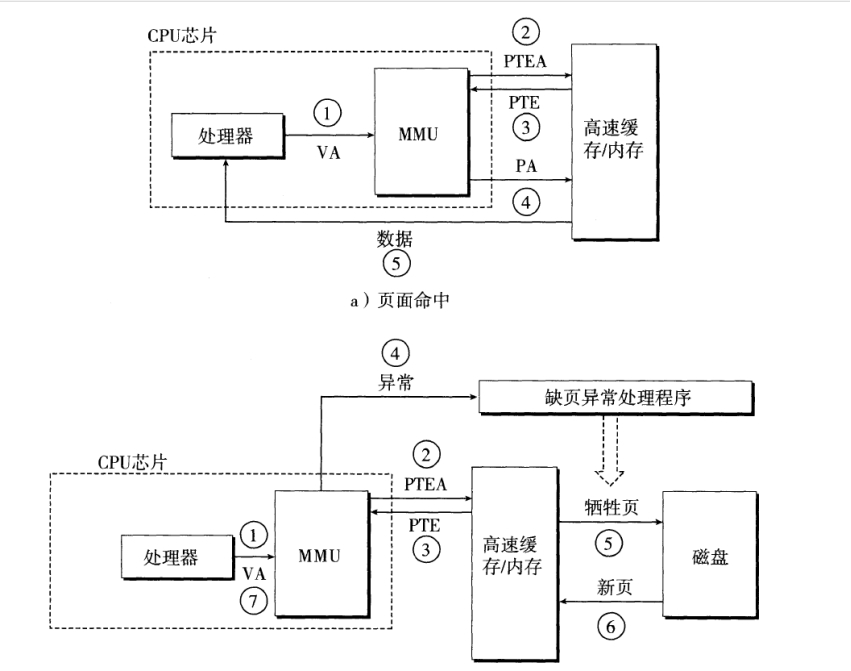
第1步：处理器生成一个虚拟地址，并把它传送给MMU；

第2步：MMU生成PTE地址，并从高速缓存/主存请求得到它；

第3步：高速缓存/主存向MMU返回PTE；

第4步：MMU构造物理地址，并把它传送给高速缓存/主存；

第5步：高速缓存/主存返回所请求的数据字给处理器



处理缺页如图b所示：

第1~3步：和图a中的第1步到第3步相同；

第4步：PTE中的有效位是零，所以MMU触发了一次异常，传给CPU中的控制到操作系统内核中的缺页异常处理程序；

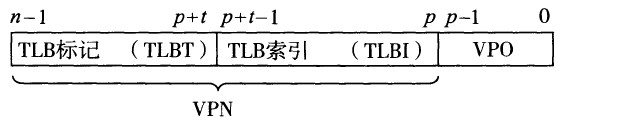
第5步：缺页处理程序确定出物理内存中的牺牲页，如果这个页面已经被修改了，则把它换出到磁盘；

第6步：缺页处理程序页面调入新的页面，并更新内存中的PTE；

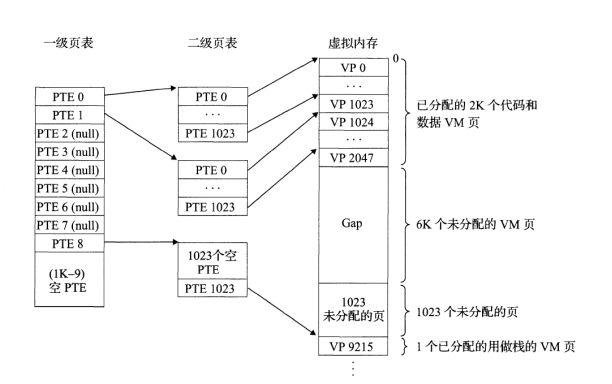
第7步：缺页处理程序返回到原来的进程，再次执行导致缺页的指令。CPU将引起缺页的虚拟地址重新发送给MMU。因为虚拟页面现在缓存在物理内存中，所以就会命中，在MMU执行了图b中的步骤之后，主存就会将所请求字返回给处理器。

## 7.4 TLB与四级页表支持下的VA到PA的变换

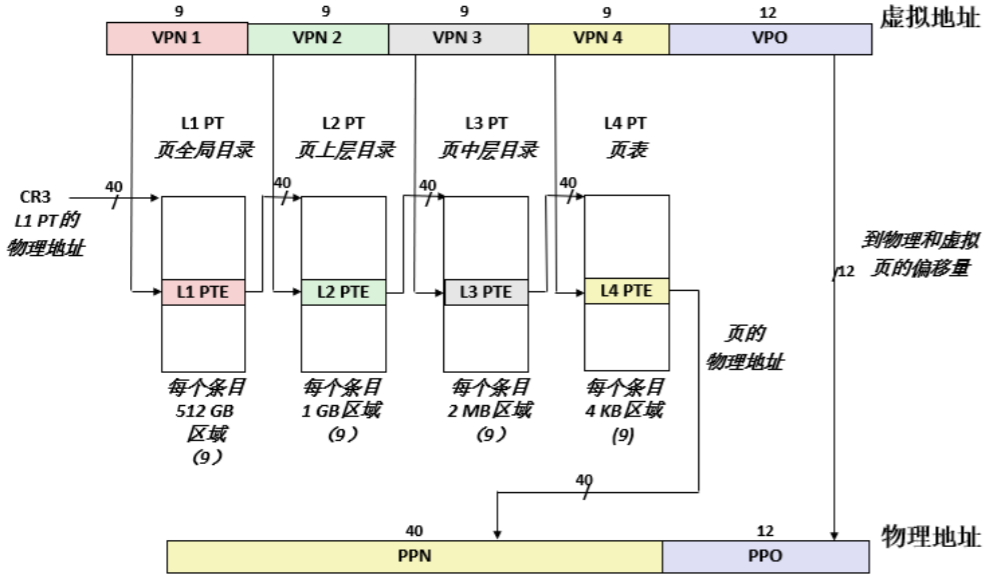
为了消除每次 CPU 产生一个虚拟地址，MMU 就查阅一个 PTE 带来的时间开销，许多系统都在 MMU 中包括了一个关于 PTE 的小的缓存，称为翻译后被缓冲器（TLB），TLB 的速度快于 L1 cache。



TLB 通过虚拟地址 VPN 部分进行索引，分为索引（TLBI）与标记（TLBT）两个部分。这样，MMU 在读取 PTE 时会直接通过 TLB，如果不命中再从内存中将PTE 复制到 TLB。同时，为了减少页表太大而造成的空间损失，可以使用层次结构的页表页压缩页表大小。core i7 使用的是四级页表。



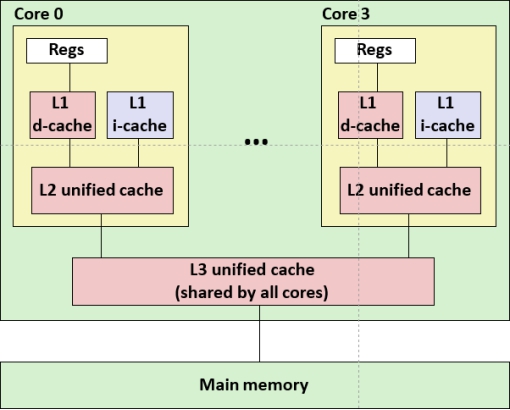
在四级页表层次结构的地址翻译中，虚拟地址被划分为 4 个 VPN 和 1 个 VPO。每个 VPNi 都是一个到第 i 级页表的索引，第 j 级页表中的每个 PTE 都指向第 j+1级某个页表的基址，第四级页表中的每个 PTE 包含某个物理页面的 PPN，或者一个磁盘块的地址。为了构造物理地址，在能够确定 PPN 之前，MMU 必须访问四个PTE。



综上，在四级页表下，MMU 根据虚拟地址不同段的数字通过 TLB 快速访问得到下一级页表的索引或者得到第四级页表中的物理页表然后与 VPO 组合，得到物理地址（PA）。

## 7.5 三级Cache支持下的物理内存访问

Core i7的内存系统如图所示。



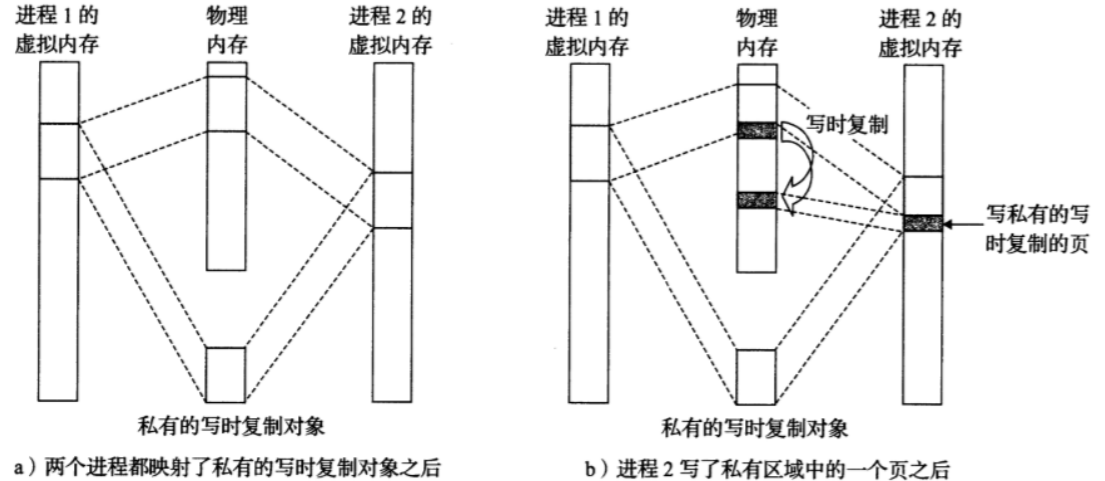
首先，根据物理地址的 s 位组索引索引到 L1 cache中的某个组，然后在该组中查找是否有某一行的标记等于物理地址的标记并且该行的有效位为 1，若有，则说明命中，从这一行对应物理地址 b 位块偏移的位置取出一个字节，若不满足上面的条件，则说明不命中，需要继续访问下一级 cache，访问的原理与 L1 相同，若是三级 cache 都没有要访问的数据，则需要访问内存，从内存中取出数据并放入cache。



## 7.6 hello进程fork时的内存映射

当fork函数被当前进程调用时，内核为新进程创建各种数据结构，并分配给它一个唯一的PID。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。它将两个进程中的页面标记为只读，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概念。



## 7.7 hello进程execve时的内存映射

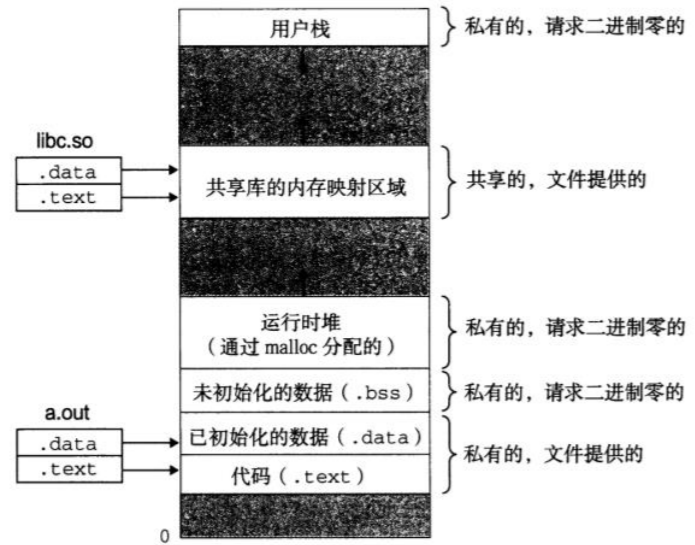
execve 函数调用驻留在内核区域的启动加载器代码，在当前进程中加载并运 行包含在可执行目标文件 hello 中的程序，用 hello 程序有效地替代了当前程序。 加载并运行 hello 需要以下几个步骤：

1. 删除已存在的用户区域，删除当前进程虚拟地址的用户部分中的已存在的 区域结构。

2. 映射私有区域，为新程序的代码、数据、bss 和栈区域创建新的区域结构， 所有这些新的区域都是私有的、写时复制的。代码和数据区域被映射为hello 文件中的.text 和.data 区，bss 区域是请求二进制零的，映射到匿名文件，其大小包含在 hello 中，栈和堆地址也是请求二进制零的，初始长度为零。

3. 映射共享区域，hello 程序与共享对象 libc.so 链接，libc.so 是动态链接到 这个程序中的，然后再映射到用户虚拟地址空间中的共享区域内。

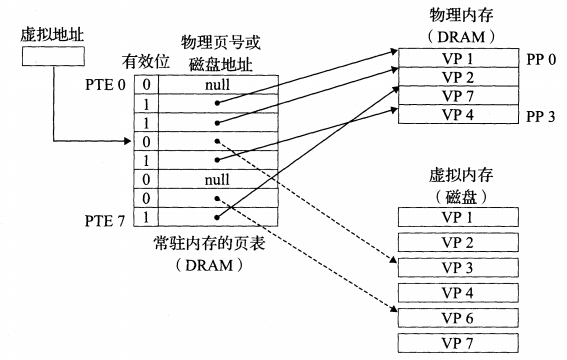
4. 设置程序计数器（PC），execve 做的最后一件事情就是设置当前进程上下 文的程序计数器，使之指向代码区域的入口点。



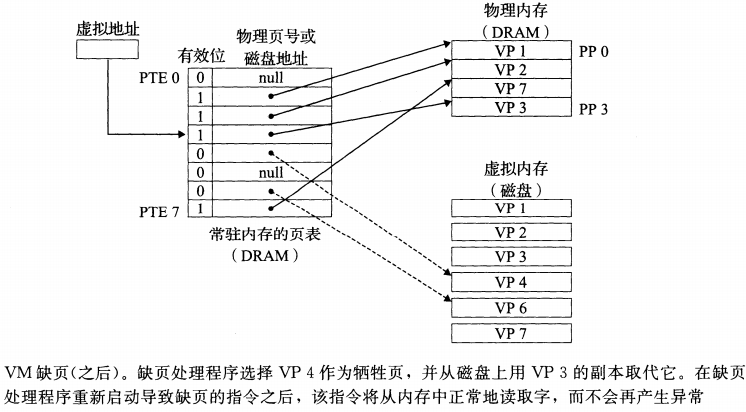
## 7.8 缺页故障与缺页中断处理

DRAM缓存不命中称为缺页，即虚拟内存中的字不在物理内存中。CPU引用了虚拟页的一个字，地址翻译硬件从内存中读取了该虚拟页对应的页表条目，从有效位推断出该页未被缓存，这样就触发了一个缺页异常，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，把要缓存的页缓存到牺牲 页的位置。如果这个牺牲页被修改过，就把它交换出去。当缺页处理程序返回时， CPU重新启动引起缺页的指令，这条指令再次发送VA到MMU，这次MMU就能正常翻译VA了。

下图对VP3的引用不命中，从而触发缺页。



缺页之后，缺页处理程序选择VP4作为牺牲页，并从磁盘上用VP3的副本取代它。在缺页处理程序重新启动导致缺页的指令之后，该指令将从内存中正常地读取字，而不会再产生异常



## 7.9动态存储分配管理

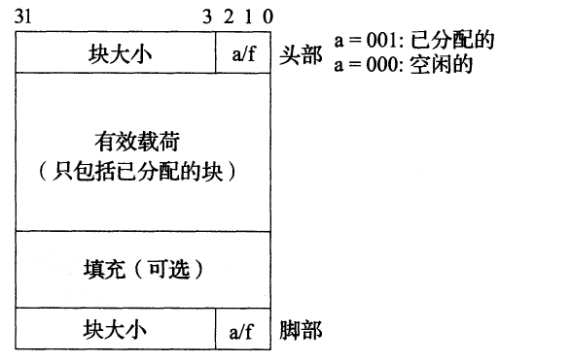
动态内存分配器维护着一个进程的虚拟内存区域，称为堆。分配器将堆视为 一组不同大小的块的集合来维护。每个块就是一个连续的虚拟内存片，要么是已 分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可用来分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

分配器分为两种基本风格：显式分配器、隐式分配器。

1. 显式分配器：要求应用显式地释放任何已分配的块。

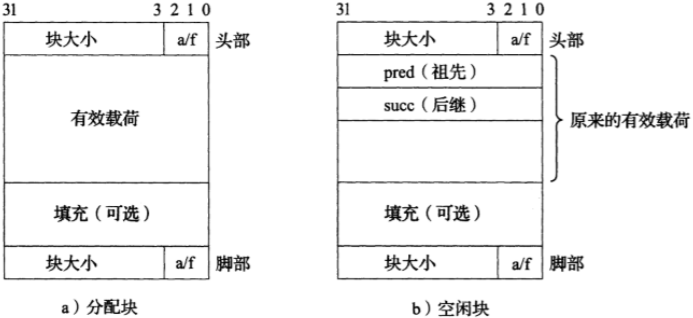
2. 隐式分配器：要求分配器检测一个已分配块何时不再使用，那么就释放这 个块，自动释放未使用的已经分配的块的过程叫做垃圾收集。

带边界标签的隐式空闲链表分配器原理：



每个块增加四字节的头部和四字节的脚部保存块大小和是否分配信息，可以在 常数时间访问到每个块的下一个和前一个块，使空闲块的合并也变为常数时间，而且可以遍历整个链表。隐式空闲链表即为，利用边界标签区分已分配块和未分配块，根据不同的分配策略(首次适配、下一次适配、最佳适配)，遍历整个链表，一旦找到符合要求的空闲块，就把它的已分配位设置为1，返回这个块的指针。隐式空闲链表并不是真正的链表，而是"隐式"地把空闲块连接了起来(中间夹杂着已分配块)。

显式空闲链表的基本原理：



因为隐式空闲链表每次查找空闲快都需要线性地遍历整个链表，而其中的已分配块显然是不需要遍历的，所以浪费了大量时间，一种更好的方式是把空闲块组织成一个双向链表，每个空闲块中包含一个 pred 和 succ 指针，指向它的前驱和后继，在申请空闲块时，就不需要遍历整个堆，只需要利用指针，在空闲链表中遍历空闲块即可。一旦空闲块被分配，它的前驱和后继指针就不再有效，变成了有效载荷的一部分。显式空闲链表的已分配块与隐式空闲链表的堆块的格式相同。

## 7.10本章小结

本章介绍了hello的存储管理机制。讨论了虚拟地址、线性地址、物理地址，介绍了段式管理与页式管理、VA 到 PA 的变换、物理内存访问，以及 hello 进程 fork 、execve 时的内存映射、缺页故障与缺页中断处理、动态存储分配管理等。

# 第8章 hello的IO管理

## 8.1 Linux的IO设备管理方法

一个 Liunx 文件就是一个 m 个字节的序列：B0，B1，…，Bm-1。所有的 I/O 设备都被模型化为文件。

文件的类型有：

1. 普通文件：包含任何数据，分两类

i. 文本文件：只含有 ASCII 码或 Unicode 字符的文件

ii. 二进制文件：所有其他文件

2. 目录：包含一组链接的文件。每个链接都将一个文件名映射到一个文件

3. 套接字：用于与另一个进程进行跨网络通信的文件

而所有的输入和输出都被当作对相应文件的读和写来执行。这种将设备优雅 的映射为文件的方式，允许 Linux 内核引出一个简单、低级的应用接口，称为 Unix I/O，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

Unix I/O 接口的几种操作：

1. 打开文件：程序要求内核打开文件，内核返回一个小的非负整数（描 述符），用于标识这个文件。程序在只要记录这个描述符便能记录打 开文件的所有信息。

2. shell 在进程的开始为其打开三个文件：标准输入、标准输出和标准错 误。

3. 改变当前文件的位置：对于每个打开的文件，内核保存着一个文件位 置 k，初始为 0。这个文件位置是从文件开头起始的字节偏移量。应用 程序能够通过执行 seek 操作显式地设置文件的当前位置为 k。

4. 读写文件：一个读操作就是从文件复制 n>0 个字节到内存，从当前文 件位置 k 开始，然后将 k 增加到 k+n。给定一个大小为 m 字节的文件， 当 k>=m 时执行读操作会出发一个称为 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。

5. 关闭文件：内核释放打开文件时创建的数据结构以及占用的内存资源， 并将描述符恢复到可用的描述符池中。无论一个进程因为何种原因终 止时，内核都会关闭所有打开的文件并释放它们的内存资源。

Unix I/O 函数：

1. 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。

2. int close(int fd);

关闭一个打开的文件。

3. ssize\_t read(int fd, void \*buf, size\_t n);

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

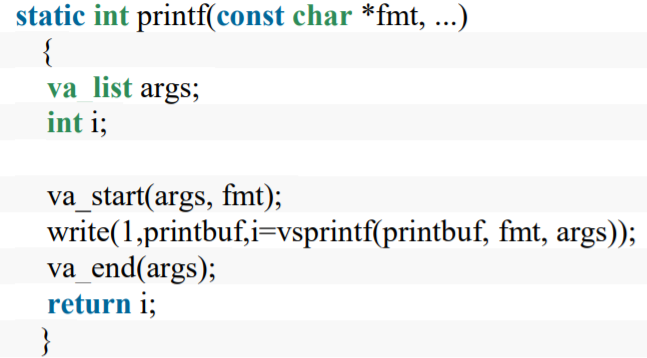
4. ssize\_t write(int fd, const void \*buf,size\_t);

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。

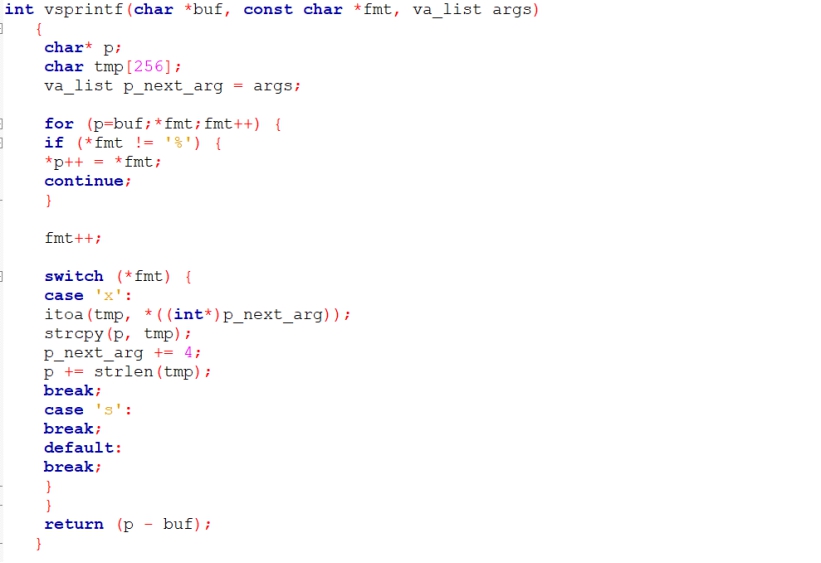
## 8.3 printf的实现分析

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

先找到 printf 的函数定义：



其中 va\_start()和 va\_end 是获取可变长度参数的函数，任何可变长度的变元被访问之前，必须先用 va\_start()初始化变元指针 argptr。初始化 argptr 后，经过对va\_arg()的调用，以作为下一个参数类型的参数类型，返回参数。最后取完所有参数并从函数返回之前。必须调用 va\_end()。由此确保堆栈的正确恢复。然后，printf 调用了 write 函数，这是 Unix I/O 函数，用以在屏幕输出长度为 i 的在 printbuf 位置的字节。这里 i = vsprintf(printbuf, fmt, args),，所以关键在于 vsprintf 函数。



vsprintf 的功能就是将 printf 的参数按照各种各种格式进行分析，将要输出的字符串存在 buf 中，最终返回要输出的字符串的长度。

接着就轮到 write 系统函数了，在 Linux 下，write 函数的第一个参数为 fd，也就是描述符，而 1 代表的就是标准输出。查看 write 函数的汇编实现可以发现，它首先给寄存器传递了几个参数，然后执行 int INT\_VECTOR\_SYS\_CALL，代表通过系统调用 syscall，syscall 将寄存器中的字节通过总线复制到显卡的显存中。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）。由此 write 函数显示一个已格式化的字符串。

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

当用户按键时，键盘接口会得到一个代表该按键的键盘扫描码，同时产生一个中断请求，中断请求抢占当前进程运行键盘中断子程序，键盘中断子程序先从键盘接口取得该按键的扫描码，然后将该按键扫描码转换成 ASCII 码，保存到系统的键盘缓冲区之中。

再看 getchar 的代码：

int getchar(void)

{

static char buf[BUFSIZ];

static char\* bb=buf;

static int n=0;

if(n==0)

{

n=read(0,buf,BUFSIZ);

bb=buf;

}

return(--n>=0)?(unsigned char)\*bb++:EOF;

}

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。

可以看到，getchar 调用了 read 函数，read 函数也通过 sys\_call 调用内核中的系统函数，将读取存储在键盘缓冲区中的 ASCII 码，直到读到回车符，然后返回整个字符串，getchar 函数只从中读取第一个字符，其他的字符被缓存在输入缓冲区。

## 8.5本章小结

本章主要讲述了Linux的I/O设备管理方法，Unix I/O接口及其函数，以及printf函数实现的分析和getchar函数的实现。

# 结论

一个小小的hello.c，竟然能折腾出这么多东西，看来任何一个程序都不简单啊。它们历经重重考验，从一个源程序变成了可执行目标文件，并不是在VS中简单地按下运行按钮就可以运行了，里面的艰辛只有学完ICS这门神课才能体会到。

我们简单梳理一下hello.c这个家伙的一生。它首先被C预处理器（cpp）处理，拓展带#的内容，变成hello.i文本文件；接着被编译器（ccl）转化成了汇编文本；但是汇编文本还是没法被机器直接处理呀，它只能识别01序列，所以还要进一步转化，由汇编器将hello.s生成可重定位目标文件hello.o；问题又来了，hello里面调用的printf等库函数是不含在hello.o里面的，所以就没法运行，于是链接器通过静态或动态链接终于生成了可执行目标文件hello。你以为hello到此就功德圆满了吗？非也，它只是经历了九九八十一难中的七七四十九难，这时它只是安静地待在磁盘里，并没有被运行。若要运行起来还需要求“各路神仙”帮忙，fork创建新进程、execve加载映射……运行中还会遇到“异常妖怪”了，需要异常处理程序才能化险为夷。hello的一生，可谓惊险而刺激！

先吐槽到这，再谈下我对学习计算机系统的感受。对于一个刚转完专业的小白来讲，一上来就开始学这么猛的课确实很难适应，上课听不懂甚至听天书的情况都很常见，好在我心态海星。第一遍跟着学的时候好多听不懂，当讲完之后再次学习的时候好多东西就看懂了。CSAPP这本教材很好，这本书我读了3遍，老师说至少读5遍才能完全搞懂，很显然书里某些东西我还是没有搞懂，但是大多数内容是没问题了。复习到期末考试前一天的时候，我突然觉得这么计算机系统挺好玩的。回想3个多月前上课时的满脸发呆，到如今对计算机系统有了初步的了解，这里面经历了太多的崩溃与自闭。

计算机系统是一个非常精细的系统，比如流水线精确到每个几微秒周期；它还是非常巧妙的，比如引入了缓存的概念。我不由赞叹计算机系统的设计者的高超智慧。在这门课我学到了很多东西，感谢一学期以来老师的辛勤付出以及耐心答疑。路漫漫其修远兮，就像hello一样，我只是刚刚经过了预处理，后面的路还长呢~

# 附件

|  |  |
| --- | --- |
| hello.c | hello源代码 |
| hello.i | 预处理之后的文本文件 |
| hello.s | hello的汇编代码 |
| hello1.s | hello.o的反汇编代码 |
| hello2.s | hello的反汇编代码 |
| hello.o | hello的可重定位文件 |
| hello | hello的可执行文件 |
| hello.elf | hello的elf文件 |
| hello1.elf | hello.o的elf文件 |

# 参考文献

[1] 大卫R.奥哈拉伦，兰德尔E.布莱恩特. 深入理解计算机系统[M]. 机械工业出版社，2016.

[2] jiangxt211. C预处理.

Available at <https://blog.csdn.net/jiangxt211/article/details/87522370>