

**计算机系统**

**大作业**

题 目 程序人生-Hello’s P2P

专 业 计算机

学　　 号 1170300815

班　　 级 1703008

学 生 范天祥

指 导 教 师 郑贵滨

**计算机科学与技术学院**

**2018年12月**

**摘 要**

本文描述了一个高级语言程序从编写到执行的全过程，详细地介绍了高级语言程序如何通过预处理、编译、汇编和链接生成完全链接的可执行文件。然后介绍了进程与存储管理，描述了一个可执行文件如何装载到内存，如何通过shell（linux下）运行程序，以及对故障的处理。最后还介绍了I/O管理，描述了如何调用函数利用硬件设计，实现输出到屏幕。全文对一个简单的C语言程序生命周期内操作系统与硬件实现机制进行了粗略的介绍，将计算机系统各个组成部分有机地结合起来，搭建出一个较为完整的知识体系。

**关键词：**编译；汇编；链接；进程存储管理；虚拟内存；操作系统

**（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**

[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 - 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 - 8 -](#_Toc532238403)

[2.3 Hello的预处理结果解析 - 8 -](#_Toc532238404)

[2.4 本章小结 - 9 -](#_Toc532238405)

[第3章 编译 - 10 -](#_Toc532238406)

[3.1 编译的概念与作用 - 10 -](#_Toc532238407)

[3.2 在Ubuntu下编译的命令 - 10 -](#_Toc532238408)

[3.3 Hello的编译结果解析 - 10 -](#_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 - 23 -](#_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 - 29 -](#_Toc532238422)

[5.6 hello的执行流程 - 32 -](#_Toc532238423)

[5.7 Hello的动态链接分析 - 32 -](#_Toc532238424)

[5.8 本章小结 - 33 -](#_Toc532238425)

[第6章 hello进程管理 - 34 -](#_Toc532238426)

[6.1 进程的概念与作用 - 34 -](#_Toc532238427)

[6.2 简述壳Shell-bash的作用与处理流程 - 34 -](#_Toc532238428)

[6.3 Hello的fork进程创建过程 - 34 -](#_Toc532238429)

[6.4 Hello的execve过程 - 35 -](#_Toc532238430)

[6.5 Hello的进程执行 - 35 -](#_Toc532238431)

[6.6 hello的异常与信号处理 - 37 -](#_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 - 41 -](#_Toc532238437)

[7.4 TLB与四级页表支持下的VA到PA的变换 - 42 -](#_Toc532238438)

[7.5 三级Cache支持下的物理内存访问 - 43 -](#_Toc532238439)

[7.6 hello进程fork时的内存映射 - 44 -](#_Toc532238440)

[7.7 hello进程execve时的内存映射 - 45 -](#_Toc532238441)

[7.8 缺页故障与缺页中断处理 - 45 -](#_Toc532238442)

[7.9动态存储分配管理 - 46 -](#_Toc532238443)

[7.10本章小结 - 47 -](#_Toc532238444)

[第8章 hello的IO管理 - 48 -](#_Toc532238445)

[8.1 Linux的IO设备管理方法 - 48 -](#_Toc532238446)

[8.2 简述Unix IO接口及其函数 - 48 -](#_Toc532238447)

[8.3 printf的实现分析 - 48 -](#_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 - 53 -](#_Toc532238452)

[参考文献 - 54 -](#_Toc532238453)

# 第1章 概述

## 1.1 Hello简介

P2P: From Program to Process

在linux环境下，hello.c文件经过预处理器 cpp 编译器 ccl 汇编器 as 链接器 ld 处理之后，采用虚拟地址形式装载进入内存。然后我们要将它的文件名输入到称为外壳（shell）的应用程序中，外壳是一个命令行解释器，它输出一个提示符，等待你输入一个命令，然后执行这个命令。OS（进程管理）为hello进行fork（Process）,为其execve,为其mmap，分其时间片，让程序得以在Hardware(CPU/RAM/IO)上驰骋（取指译码执行/流水线等）

O2O: From Zero-0 to Zero-0

当程序运行结束后，shell父进程会负责回收hello子进程，内核删除相关数据结构，完美谢幕。

## 1.2 环境与工具

硬件环境：Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz 2.70GHz 8G RAM 256G SSD x64处理器

软件环境：VMware 14 player、ubuntu-18.04.1、Windows10

开发与调试工具：gcc、vim、edb、hexedit、readelf、objdump

## 1.3 中间结果

|  |  |
| --- | --- |
| 文件名称 | 文件作用 |
| hello.c | 提供源程序 |
| hello.i | cpp预处理之后的文件 |
| hello.s | cc1之后的汇编语言格式文件 |
| hello.o | as之后的可重定位目标文件 |
| hello | ld之后的可执行目标文件 |
| helloftx.elf | hello.o的ELF格式文件 |
| hello.objdump | Objdump得到的hello的反汇编代码 |

## 1.4 本章小结

本章介绍了hello程序的P2P，020的整个过程。

提供了完成本次实验的硬件软件环境和所使用的开发与调试工具

粗略介绍了研究过程中的中间结果，以及其用途和对研究过程产生的作用。

**（第1章0.5分）**

# 第2章 预处理

## 2.1 预处理的概念与作用

**概念：**

预处理一般是在编译之前进行的处理。C语言的预处理主要有三个方面的内容： 1.宏定义； 2.文件包含； 3.条件编译。 预处理命令以符号“#”开头。

预处理器cpp根据以字符#开头的命令（宏定义、条件编译），修改原始的C程序，将引用的所有库展开合并成为一个完整的文本文件。

**作用：**

**经过预编译处理后，得到的是预处理文件（如：hello.i) ，它还是一个可读的文本文件，但不包含任何宏定义。**

**1. 宏定义**

#define标识符 字符串：将宏名替换为字符串。

#define 宏名（参数表） 字符串：除了一般的字符串替换，还要做参数代换。

**2. 文件包含**

编辑：一个文件包含另一个文件的内容

格式：#include "文件名" 或 #include <文件名>

编译时以包含处理以后的文件为编译单位，被包含的文件是源文件的一部分。

编译以后只得到一个目标文件.obj

被包含的文件又被称为“标题文件”或“头部文件”、“头文件”，并且常用.h作扩展名。修改头文件后所有包含该文件的文件都要重新编译。头文件的内容除了函数原型和宏定义外，还可以有结构体定义，全局变量定义。

**3. 条件编译**

**A.双分支条件编译**

形式：

#if 条件表达式

代码段1

#else

代码段2

#endif

**作用：**

//#endif结束条件编译

//#if,#else和C语言里的if else功能一样，但是时间开销不一样

//if else会编译所有的代码，源码会较长，编译时间会较长

//程序体积大，占用更多内存，运行时间长

//#if,#else只编译符合条件的语句，有效减少被编译的语句，

//缩短源码长度，缩短程序执行时间

**B.多分支条件编译**

**条件编译的概括作用：**

使用条件编译可以使目标程序变小，运行时间变短。

预编译使问题或算法的解决方案增多，有助于我们选择合适的解决方案。

**PS:** 此外，还有布局控制：#pragma，这也是我们应用预处理的一个重要方面，主要功能是为编译程序提供非常规的控制流信息。

**具体形式、作用：**

#if 条件表达式1

代码段1

#elif 条件表达式2

代码段2

#elif 条件表达式3

代码段3

#elif 条件表达式4

代码段4

#else

代码段5

#endif

检测宏是否定义

#ifdef 宏定义

代码段1

#endif

#ifdef M 检测M这个宏是否定义，定义了就执行代码段1，没有定义就不执行任何操作

#ifdef一般用于开启某个功能

检测宏是否未定义

#ifndef 宏定义

代码段1

#endif

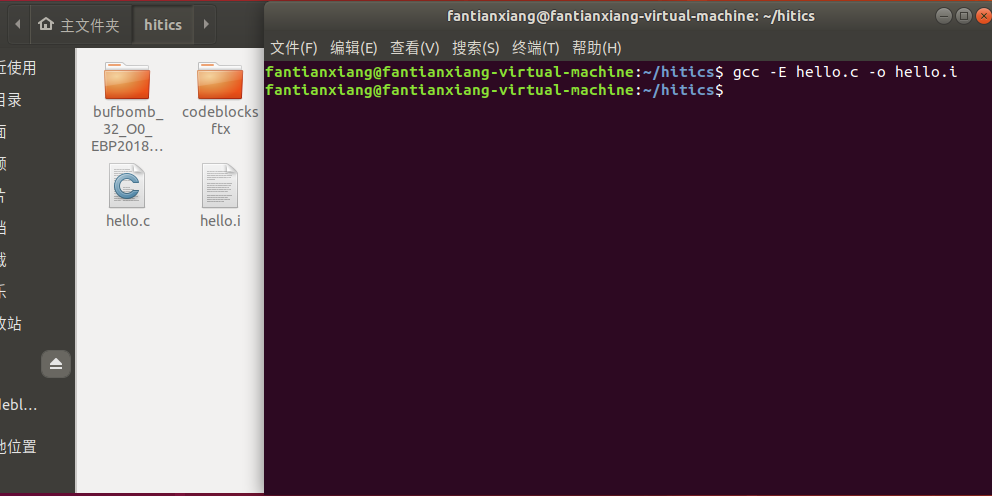
#ifndef M 检测M这个宏是否未定义，没有定义就执行代码段1，定义就不执行任何操作

#ifndef一般用于开启某个功能或者include 重包含排错

## 2.2在Ubuntu下预处理的命令

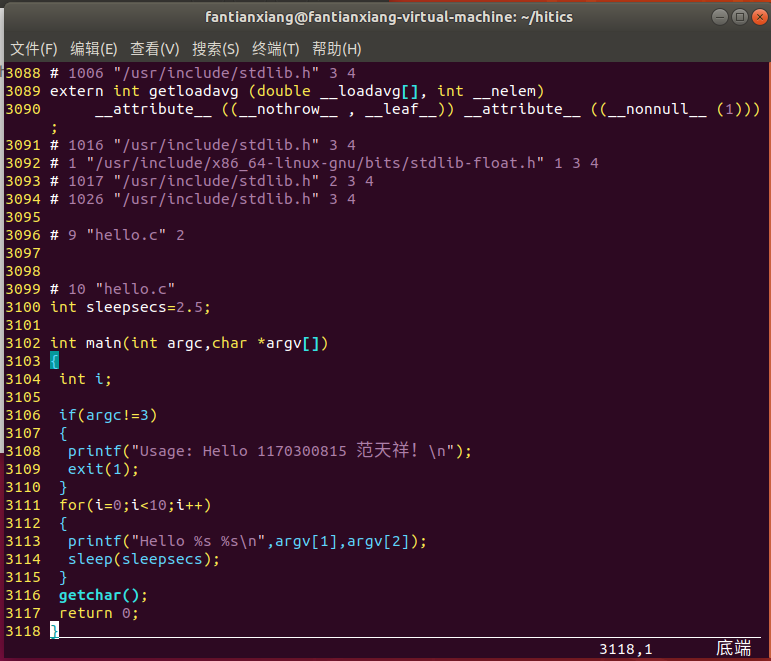
命令：linux> gcc –E hello.c –o hello.i

预处理过程截图：



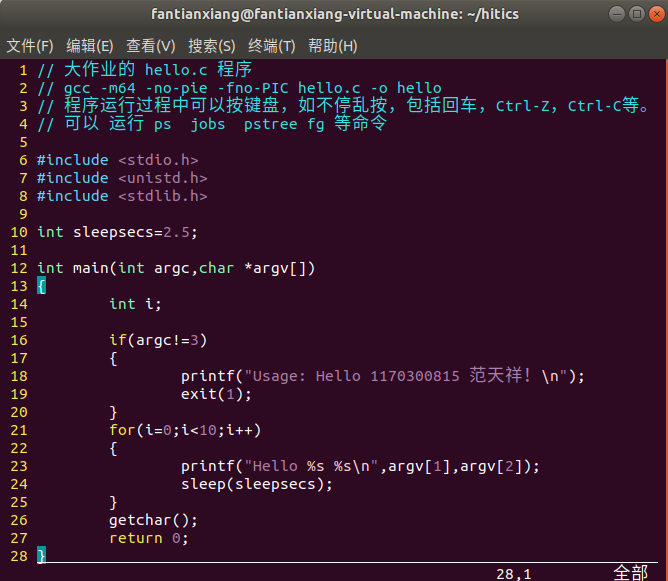
## 2.3 Hello的预处理结果解析

1.使用命令linux> vim hello.i查看预处理结果的内容：



可以发现程序已经拓展至3000多行，main函数的代码在hello.i的最下面。表示这个程序已经预处理完毕。

2.再来查看原来的hello.c文件，如下：



可以看见这个程序里面包含了3个头文件，故前面的部分应该为3个头文件的**依次包含（多次出现，只展示部分截图）**，







当然这3个.h文件里面也包含了很多其它的.h文件和有很多#define语句#if,#else语句等，gcc会对此进行递归展开，上面的介绍里面已经描述过，在此不再赘述。

## 2.4 本章小结

经过预编译处理后，得到的是一个完整的预处理文件（如：hello.i) ，它还是一个可读的文本文件，但不包含任何宏定义。

**（第2章0.5分）**

# 第3章 编译

## 3.1 编译的概念与作用

**概念：**

编译过程就是将预处理后得到的预处理文件（如hello.i）进行

词法分析、语法分析、语义分析、优化后，生成汇编代码文件。

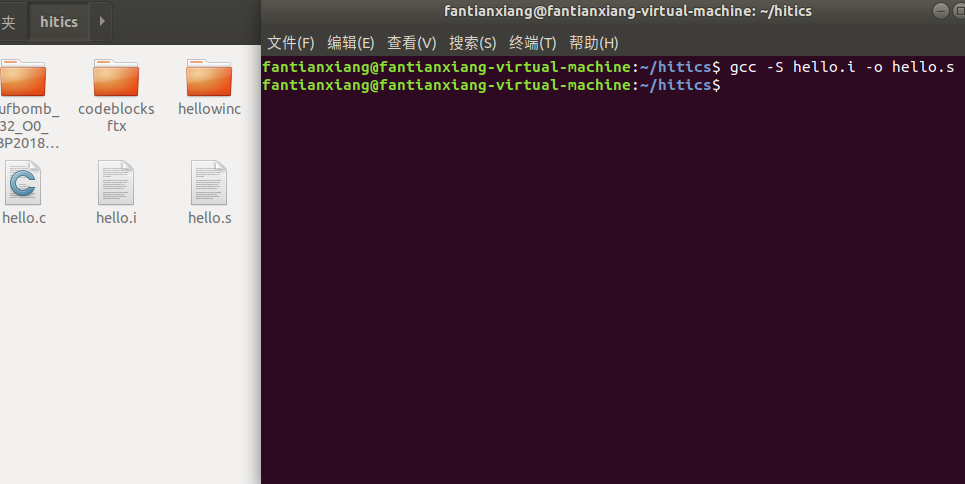
**作用：**

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。

## 3.2 在Ubuntu下编译的命令

命令行：linux> gcc –S hello.i –o hello.s

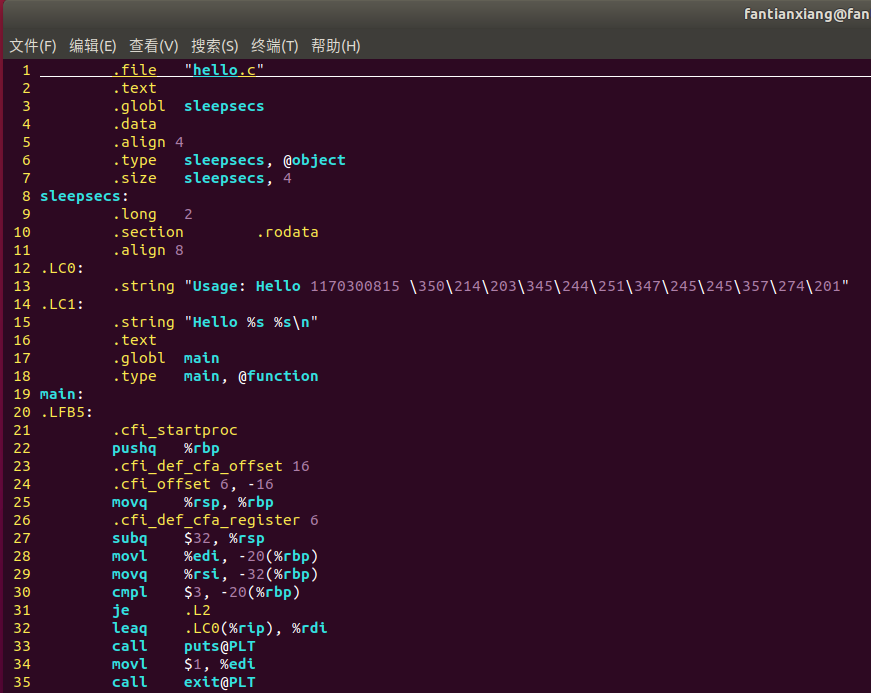
预处理过程截图：

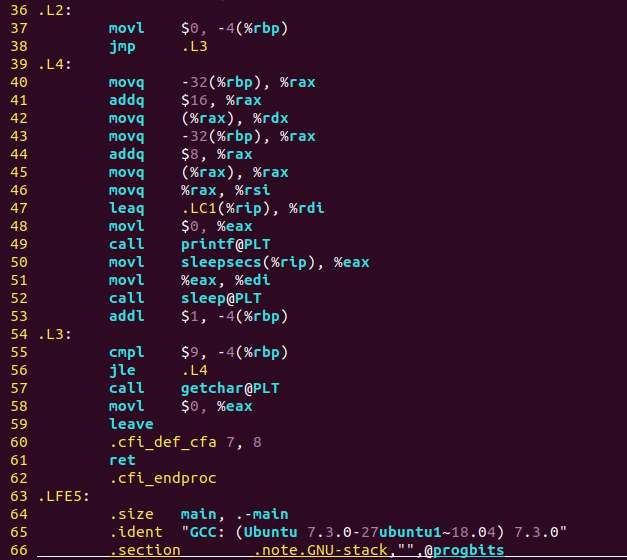


## 3.3 Hello的编译结果解析

**编译器是怎么处理C语言的各个数据类型以及各类操作**

**首先查看hello.s文件内容：**





**3.3.1数据类型（整数、字符串、数组）**

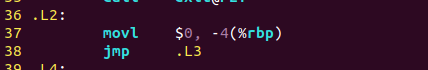
整数：可以在C文件里面看见，有int sleepsecs=2.5; int argc; int i;这3个整型变量和很多立即数，编译器处理分析如下：

a. int sleepsecs在C程序中被声明为全局变量，且已经被赋值，编译器处理时：编译器首先将sleepsecs在.text代码段中声明为全局变量，然后在.data节声明该变量（.data节存放已经初始化的全局和静态C变量）。



b. int argc：声明参变量个数。

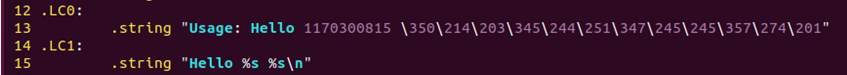
c. int i：编译器将局部变量存储在寄存器或者栈空间中，在hello.s中编译器将i存储在栈上空间-4(%rbp)中。



d. 立即数：其他整形数据的出现都是以立即数的形式出现的，直接出现在汇编代码中。

**字符串：**

声明在.LC0和.LC1段中的字符串（.rodata节）

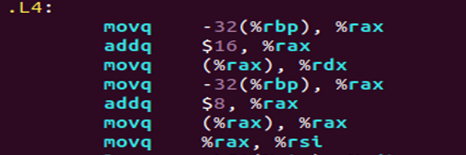


**数组：**

数组是一段数据类型相同的物理位置相邻的变量集合，对数组的索引实际上就是在第一个元素地址的基础上通过加索引值乘以数据大小来实现

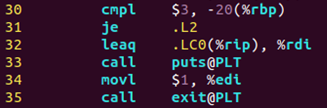
argv数组作为一个char\* 类型的数组，char\* 的大小是8个字节，所以argv[1]加0x8，而argv[2]加0x10。

对应的汇编位置为：

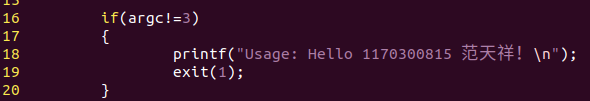


**3.3.2控制转移：**

a.if语句跳转实现：

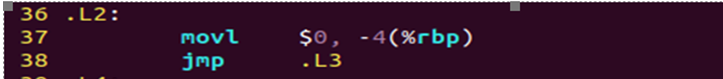


对应的C代码：

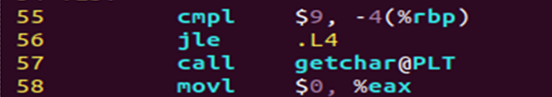


b. for循环跳转实现:

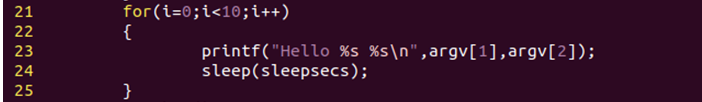
使用了一个局部变量i，该变量存放在栈中rbp寄存器指向地址-4处。



首先对其置零进行初始化，接着使用cmpl进行比较，如果i<=9，则跳入.L4 for循环体执行，否则说明循环结束，顺序执行for之后的逻辑。



对应的C代码实现：

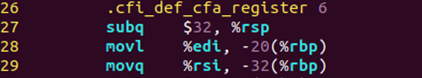


**3.3.3函数操作：**

函数调用的基本过程。首先将%rbp通过push指令存储在栈中，并将%rsp栈指针赋值给%rbp。通过sub指令减小栈指针地址从而为局部变量分配空间。然后将%edi和%rsi寄存器中的值存储在栈中。具体函数分析如下：

**main函数本身的参数读取**

参数部分给出了int argc,char \*argv[]两个参数，其中%edi代表argc，%rsi代表argv[]。



然后分配和释放内存

**printf的调用**

先传递数据后控制转移

第一次实际上调用的是puts（只有一个字符串参数）



第二次为printf



**exit函数的调用**

exit状态为1

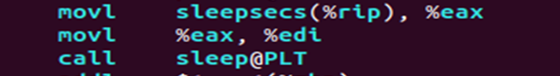


所以movl $1,%edi



**sleep函数的调用**

先将sleepsecs的值传给%edi，再调用sleep@PLT（PLT后面PIC函数动态链接时使用）



**getchar函数的调用：**

%eax置零，call getchar@PLT



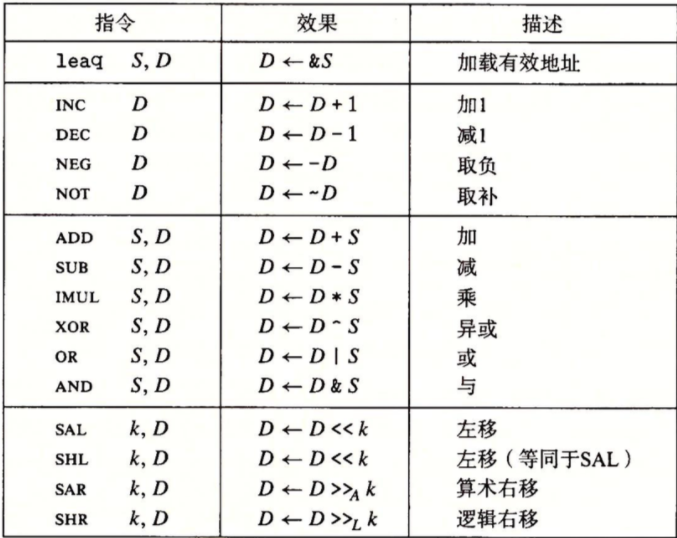
**3.3.4 赋值操作**

根据不同大小的数据类型有movb、movw、movl、movq、movabsq



**3.3.5 算数操作**

1.算术和逻辑操作类指令分四类：加载有效地址，一元操作，二元操作和移位，如下：



2. leaq指令，类似mov指令，它左侧的数看似是给出一个地址，在内存中从给定的地址取操作数，传给右边的目的地。但其实没有取，而是直接将左侧的数对应的地址传给了右侧的目的地。

本代码里面涉及的操作为：

i++，对计数器i自增

leaq操作：



**3.3.6关系操作（比较测试指令）**

常见关系操作如下：

cmp S1,S2 比较设置条件码

test S1,S2 测试设置条件码

jne/ja/j… 按照条件码来跳转

此次程序涉及的关系运算为cmpl，分别在判断argc是否等于3和循环变量是否小于10时使用，都是利用条件码来跳转。





**3.3.7类型转换**

可以看见在声明全局变量的时候有一个类型转换



当int、float和double格式之间进行强制类型转换时，程序改变数值和位模式的原则如下：

(1)从int 转换成float，数字不会溢出，但是可能被舍入。

(2)从int 或者float转换成double,因为dobule有更大的范围和精度，所以能够保留精确的数值。

(3)从double转换成float，因为范围要小一些，所以值可能溢出为-∞或者+∞,另外，由于精度较小，它还可能被舍入。

(4)从float或者double转换成int，值将会向零舍入。

## 3.4 本章小结

经过编译后，得到的汇编代码文件（如hello.s）还是可读的文本文件，但是CPU无法理解和执行它。

**（第3章2分）**

# 第4章 汇编

## 4.1 汇编的概念与作用

**概念：**

汇编指令和机器指令一一对应，前者是后者的符号表示，它们都属于机器级指令，所构成的程序称为机器级代码。

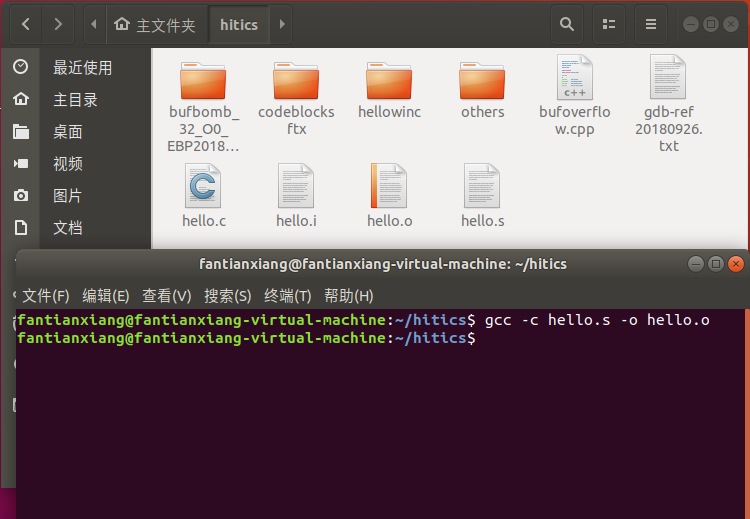
**作用：**

汇编程序（汇编器as）用来将汇编语言源程序转换为机器指令序列（机器语言程序）。把这些指令打包成可重定位目标程序的格式，并将结果保存在.o目标文件中。

## 4.2 在Ubuntu下汇编的命令

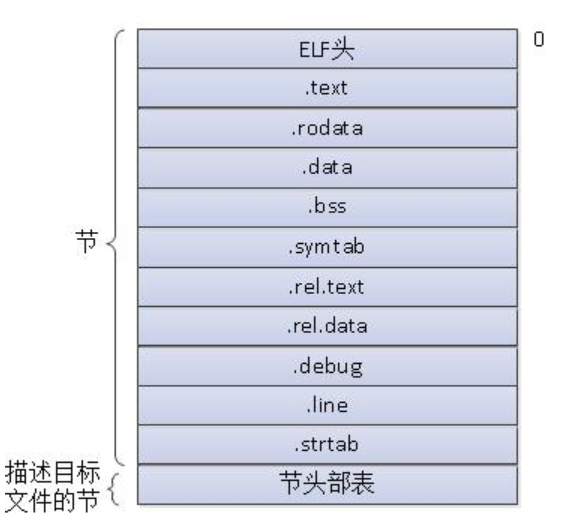
命令行：linux> gcc –c hello.s –o hello.o

预处理过程截图：



## 4.3 可重定位目标elf格式

A.很有必要先看一下典型的ELF可定位目标文件的格式：



B.分析hello.o的ELF格式（各节的基本信息）：

1.ELF头

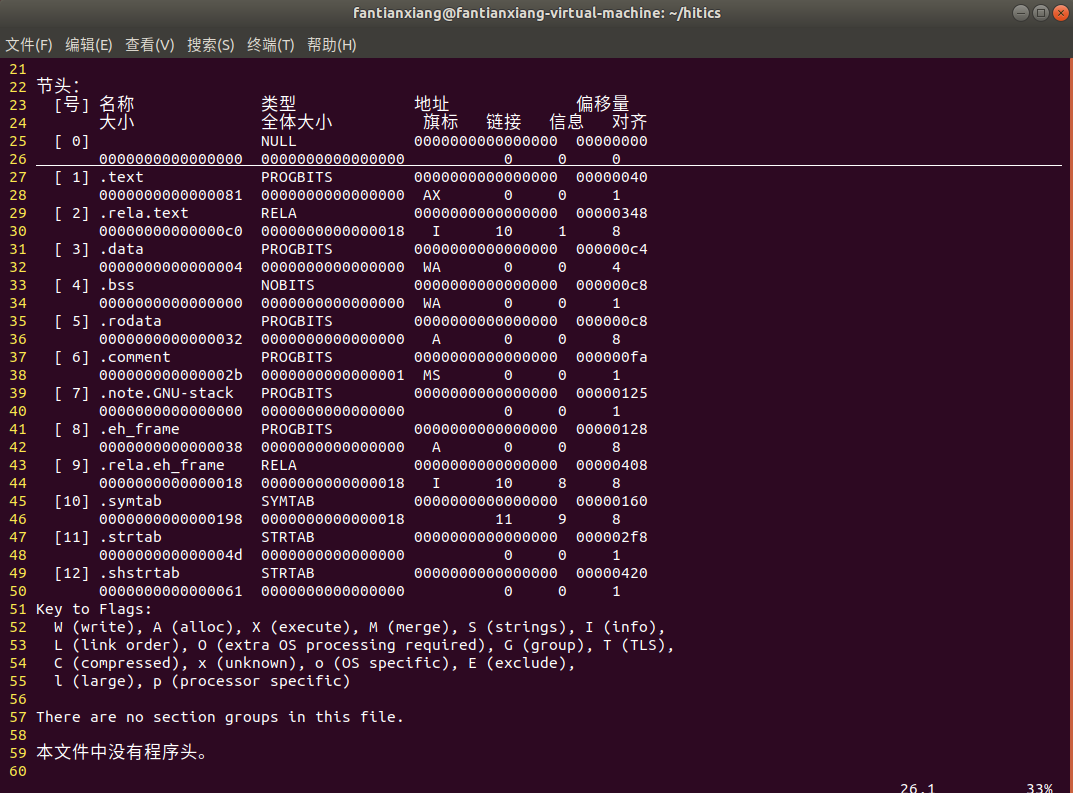
以16字节的序列开始，对应于图中的magic部分，描述了生成该文件的系统的字的大小和字节顺序。

ELF头剩下的部分包含帮助链接器语法分析和解释目标文件的信息，其中包括ELF头的大小、目标文件的类型、机器类型、字节头部表（section header table）的文件偏移，以及节头部表中条目的大小和数量等信息。



2.节头表部分：

节头表中条目的大小和数量描述了各个节的含义：节类型，节大小，节地址和偏移量。



3.重定位节：

代码的重定位条目，即为图中的.rela.text节，.rela.text节包含.text节的重定位信息，用于重新修改代码段的指令中的地址信息。

**首先是各个函数**(.rodata-4（第一个printf中的字符串）、puts函数、exit函数、.rodata+21（第二个printf中的字符串）、printf函数、sleepsecs函数、sleep函数、getchar函数)的：

重定位的偏移量（在.text节或者是.data节的偏移位置）；

重定位的信息；

重定位到目标的类型；

重定位符号值和重定位符号。



**然后就是符号表（此处包含17个符号（入口项））：**

符号表给出了：

符号的序号Num、符号的地址（偏移量）Value、符号的大小Size、符号的类型Type、符号的名称Name；

还有：Bind = GLOBAL 绑定意味着该符号对外部文件的可见. LOCAL 绑定意味着该符号只在这个文件中可见. WEAK 有点像GLOBAL, 该符号可以被重载；

（Vis）符号可以是默认的、受保护的、隐藏的或者内部的；

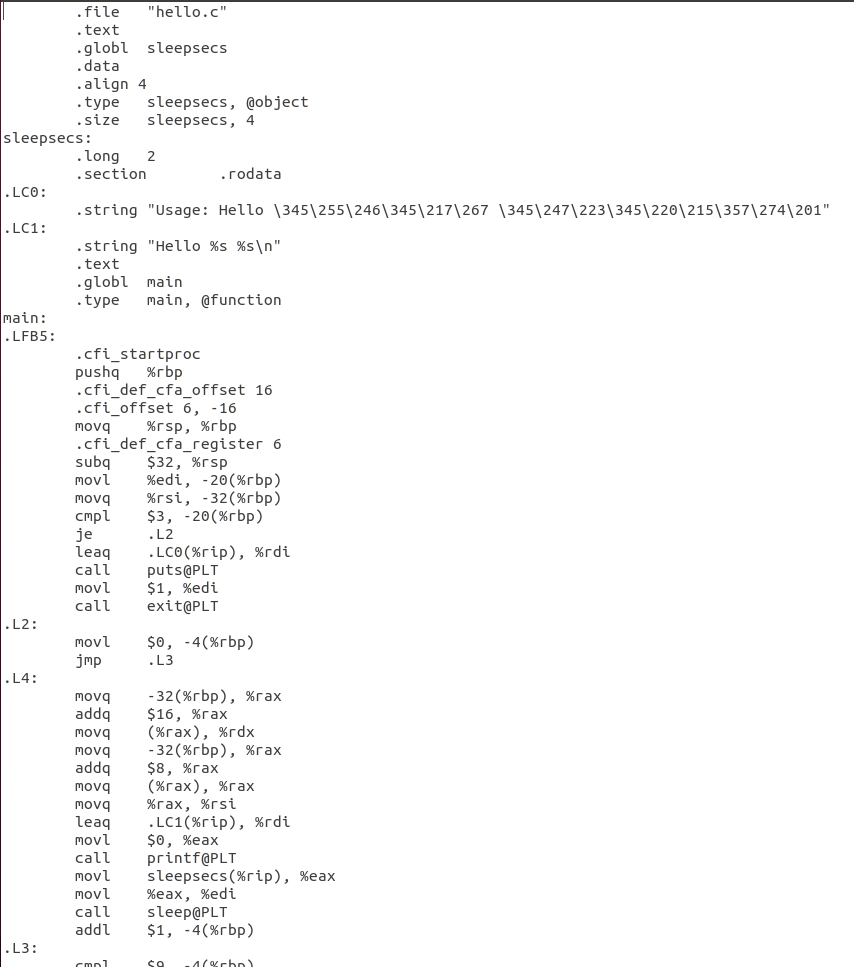
（Ndx）符号所在的区号。ABS意味着绝对：不调整到任何区段地址的迁移。

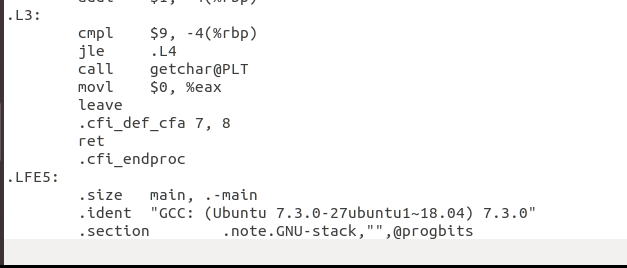


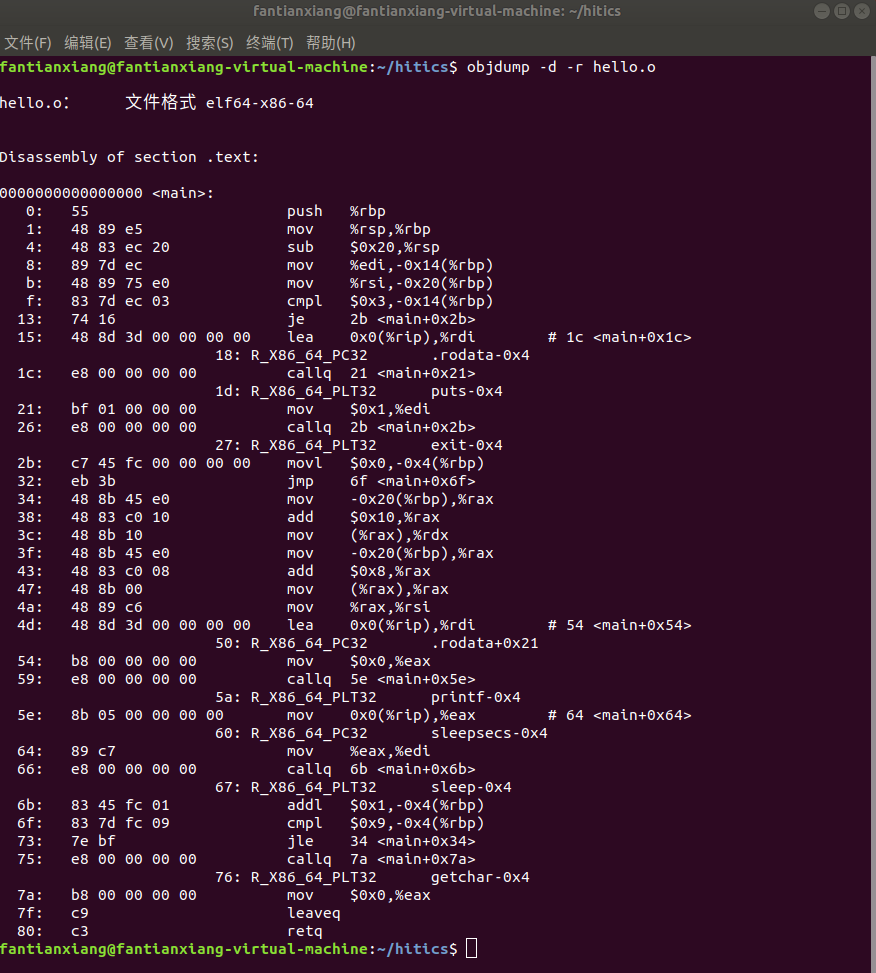
## 4.4 Hello.o的结果解析

命令行：linux> objdump -d -r hello.o查看hello.o的反汇编

与第3章的 hello.s进行对照分析，如下所示：







**4.4.1机器语言的构成与汇编语言的映射关系：**

这个就是一些汇编指令对应着相应的机器编码（由于操作数类型、寻址方式等的不同，同一个汇编语言符号可能对应着不同的机器语言操作码。），以及立即数和地址等组成的机器语言。当然要注意在看反汇编时注意小端序。

**4.4.2分支转移：**

反汇编和gcc编译得到的汇编语言代码都有具体详尽的程序实现基本代码，反汇编代码通过地址表示跳转的目标，而gcc得到的是通过抽象的形式，例如.L2来表示跳转的目标。这里的抽象形式.L2是一种段名称，作为助记符便于编写，在编译为机器语言后当然要转化为地址模式，未链接之前一般是00 00 00 00来填充。

**4.4.3函数调用：**

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

反汇编的结果中包含了供对照的来自可重定位目标文件中的机器语言代码及相关注释，包括一些相对寻址的信息与重定位信息

## 4.5 本章小结

汇编结果是一个可重定位目标文件（如：hello.o）（其中包含的是不可读的二进制代码，必须用相应的工具软件来查看其内容）。

当然，生成最终的可执行文件还需要经过链接这一步。

**（第4章1分）**

# 第5章 链接

## 5.1 链接的概念与作用

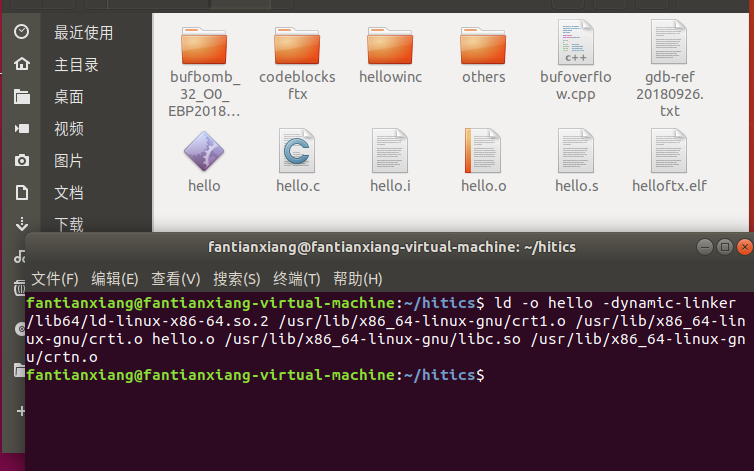
概念：预处理、编译和汇编三个阶段针对一个模块（一个\*.c文件）进行处理，得到对应的一个可重定位目标文件（一个\*.o文件）。链接可以执行于编译时，也就是在源代码被编译成机器代码时；也可以执行于加载时，也就是在程序被加载器加载到内存并执行时；甚至于运行时，也就是由应用程序来执行。

作用：链接过程将多个可重定位目标文件合并以生成可执行目标文件

## 5.2 在Ubuntu下链接的命令

命令行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

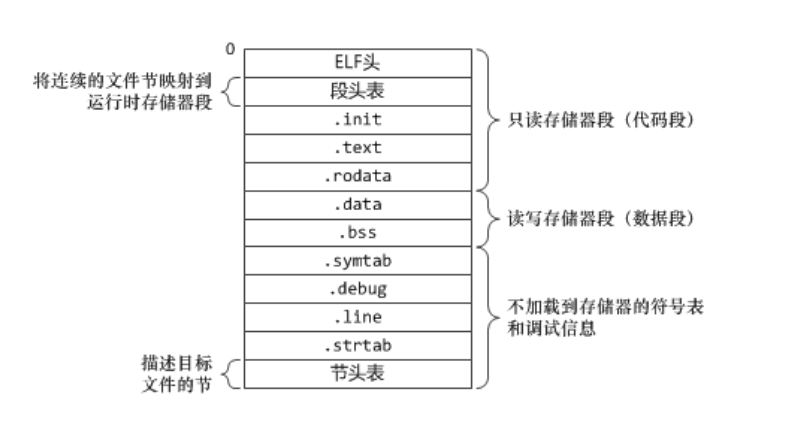
使用ld的链接命令，如图所示：



## 5.3 可执行目标文件hello的格式

命令行:linux> readelf –a hello

首先看一下可执行目标文件的格式：



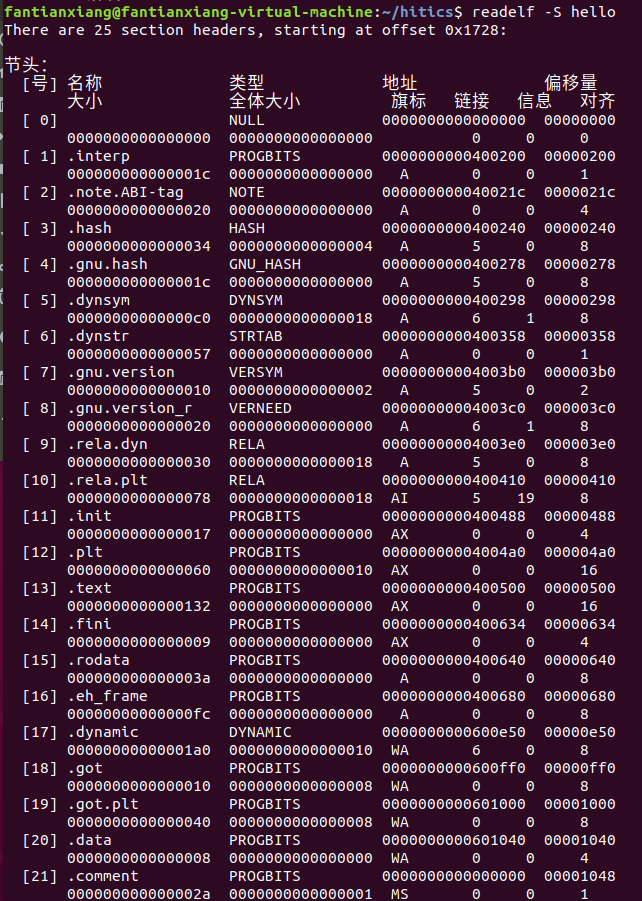
再看hello的ELF具体的形式：

a.hello.out文件的文件头

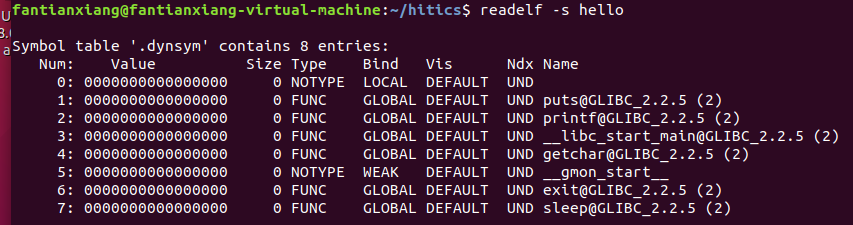
基本信息:有头的大小、程序头的大小、节头的大小、节头的数量、字符串索引节头等信息。当然也显示这是一个可执行文件。



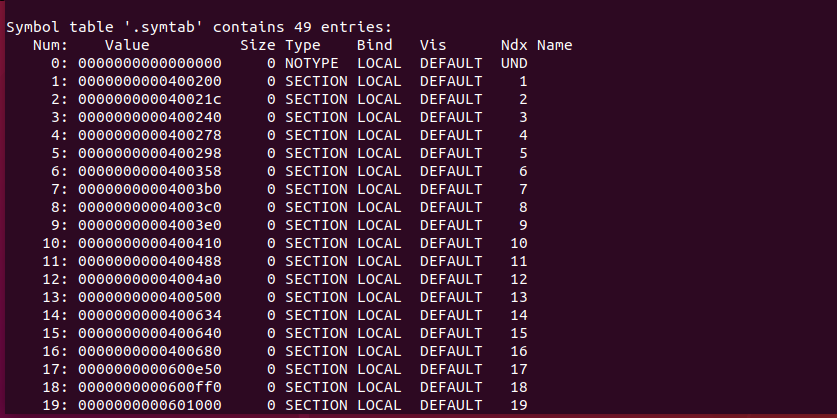
b.段头表(there are 25 section headers , starting at offset 0x1728):



c.符号表（符号表的详细信息在之前已经解释过了）：



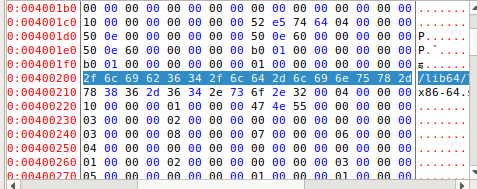
包含49个入口：



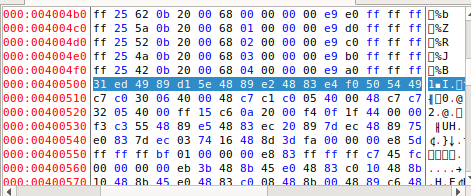
## 5.4 hello的虚拟地址空间

使用edb加载hello，查看本进程的虚拟地址空间各段信息如图所示：

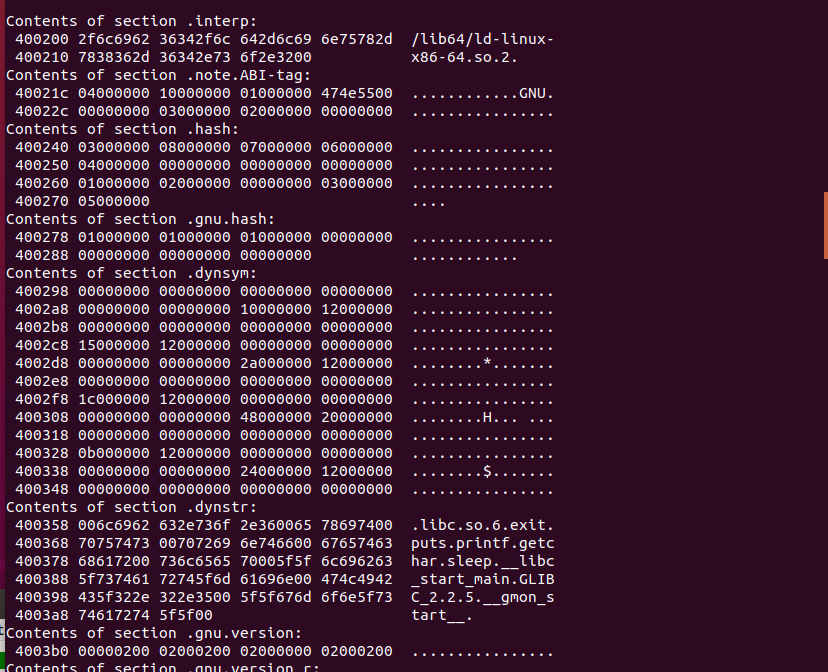
section.interp内容的虚拟地址：（**详细各段信息见本节最后面的终端截图，有各节的名称和对应的详细的虚拟地址空间的信息）**

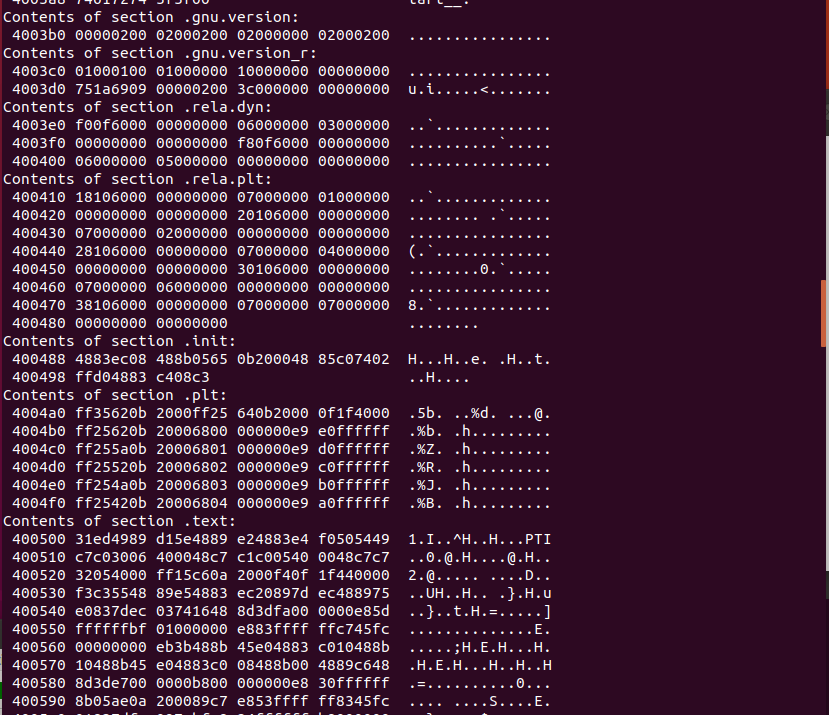


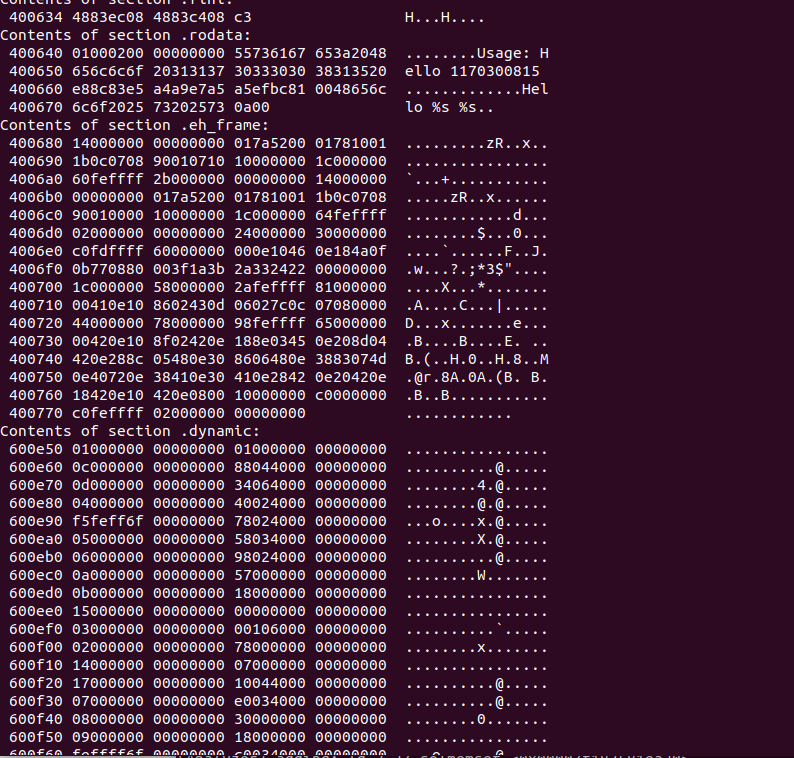
section.text内容的虚拟地址，开始于0x400500这个地方：



对照5.3来分析，可执行目标文件各节的基本信息都描述了各节在具体虚拟地址空间的相对位置关系。程序头表在执行的时候被使用，它告诉链接器运行时加载的内容并提供动态链接的信息。每一个表项提供了各段在虚拟地址空间和物理地址空间的大小、位置、标志、访问权限和对齐方面的信息。在形成的虚拟地址空间里，我们可以根据这一点来理解各个地址的含义。



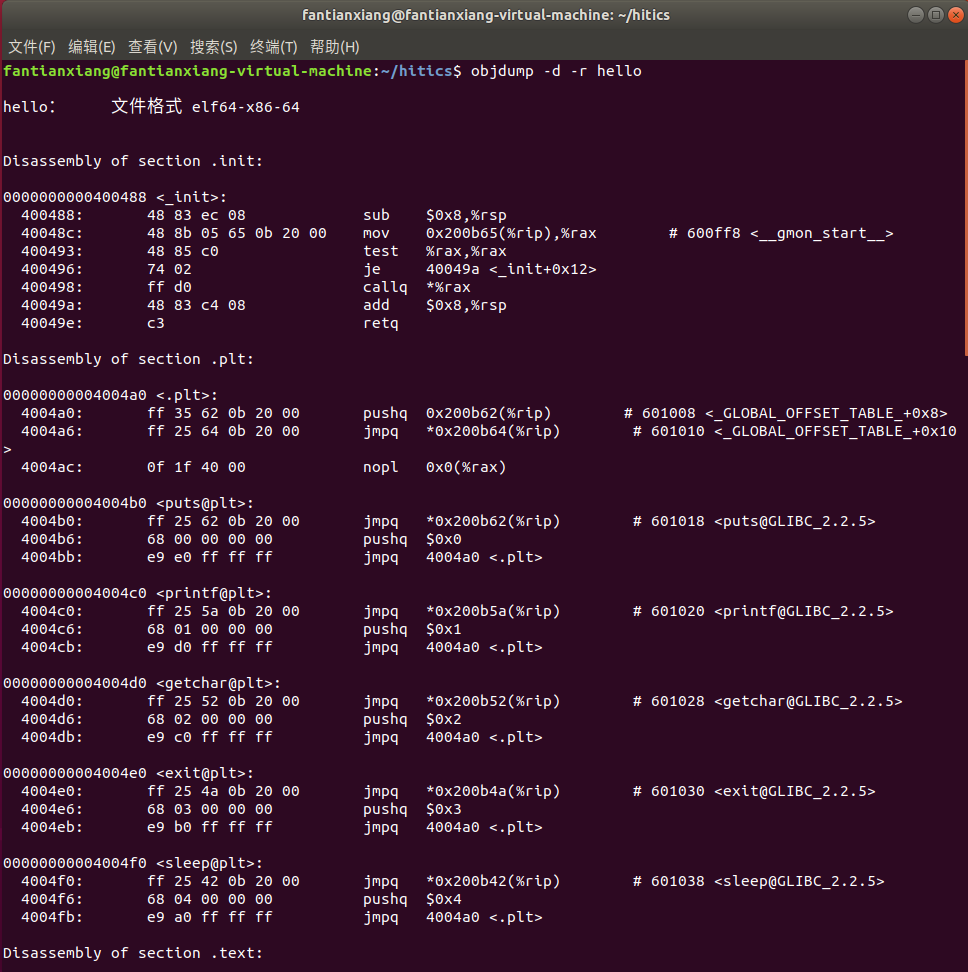


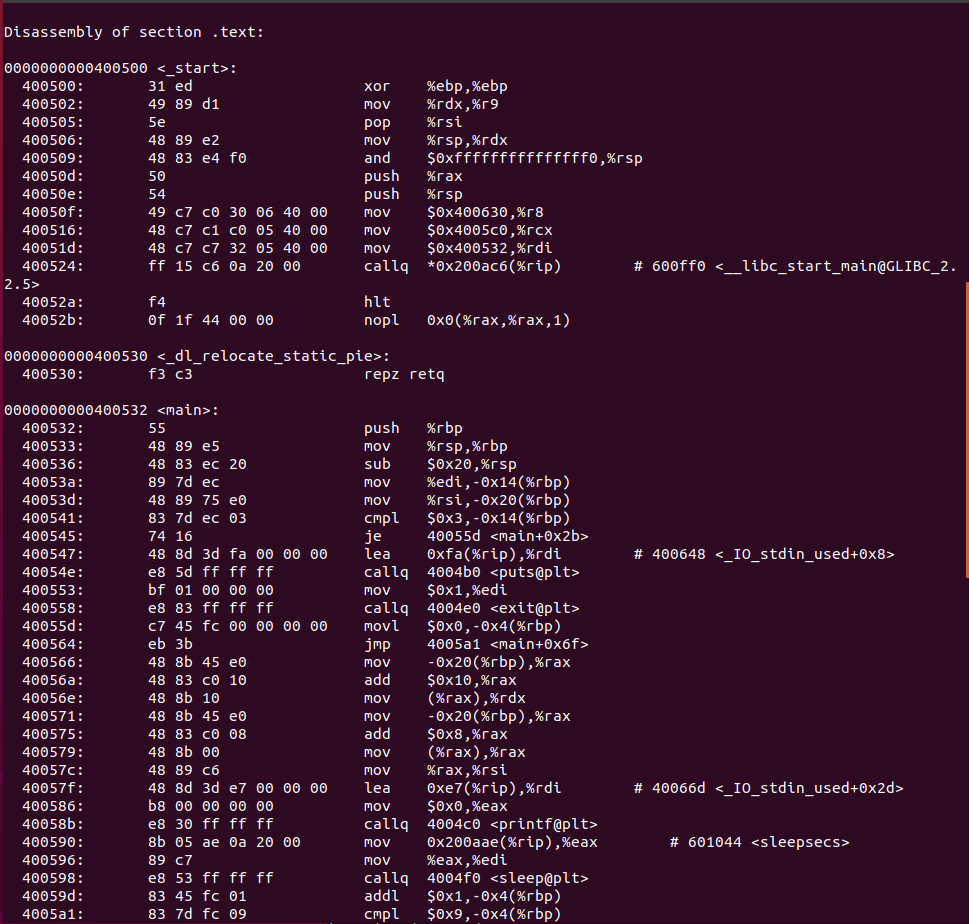


## 5.5 链接的重定位过程分析

命令行linux> objdump -d -r hello

如图所示：





**hello与hello.o的不同：**

hello.o没有经过链接，所以main的地址从0开始，并且不存在调用的如printf这样函数的代码。另外，很多地方都有重定位标记，用于后续的链接过程。hello.o反汇编代码的相对寻址部分的地址也不具有参考性，没有经过链接并不准确。

而对于hello这个已经链接好的可执行目标文件来说，库函数的代码都已经链接到了程序中，程序各个节变得更加完整，跳转的地址也具有参考性。

**结合hello.o的重定位项目，分析hello中对其怎么重定位的**：

使用ld命令链接的时候，将hello.c中用到的函数都加入链接器，当链接器解析重定条目时发现对外部函数调用，比如说：



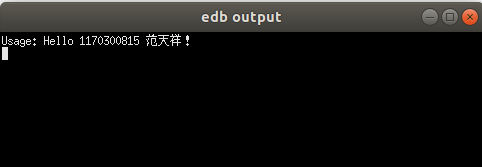
这个是一个使用32位PC相对地址的引用，未链接之前是用00 00 00 00填充的，下面及时给出了**具体重定位的信息：符号为sleepsecs，偏移量为-0x4，即加数r.addend为0x-4。**

链接器直接修改call之后的值为目标地址与下一条指令的地址之差，指向相应的字符串。使用书上的公式计算出结果00 20 0a ae，观察链接后的填充值，得到验证。（小端序）



## 5.6 hello的执行流程

edb执行hello结果如图所示：



**加载hello：**

ld-2.23.so!\_dl\_start//ld-2.23.so!\_dl\_init//LinkAddress!\_start//libc-2.23.so!\_libc\_start\_main libc-2.23.so!\_cxa\_atexit LinkAddress!\_libc\_csu.init libc-2.23.so!\_setjmp

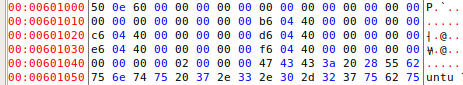
**call main ：**使用gdb单步运行，可以找出.text节中main函数前后执行的函数名称。在main函数之前执行的程序有：\_start、\_\_libc\_start\_main@plt、\_\_libc\_csu\_init、\_init、frame\_dummy、register\_tm\_clones。在main函数之后执行的程序有：exit、cxa\_thread\_atexit\_impl、fini。

**程序终止：**libc-2.23.so!exit

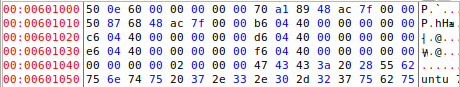
## 5.7 Hello的动态链接分析

动态链接时：函数在程序执行的时候才会确定地址。GNU编译系统采用延迟绑定技术来解决动态库函数模块调用的问题，它将过程地址的绑定推迟到了第一次调用该过程时。延迟绑定通过全局偏移量表（GOT）和过程链接表（PLT）实现。如果一个目标模块调用定义在共享库中的任何函数，那么就有自己的GOT和PLT。当某个动态链接函数第一次被调用时先进入对应的PLT条目例如PLT[2]，然后PLT指令跳转到对应的GOT条目中例如GOT[4]，其内容是PLT[2]的下一条指令。然后将函数的ID压入栈中后跳转到PLT[0]。PLT[0]通过GOT[1]将动态链接库的一个参数压入栈中，再通过GOT[2]间接跳转进动态链接器中。动态链接器使用两个栈条目来确定函数的运行时位置，用这个地址重写GOT[4]，然后再次调用函数。经过上述操作，再次调用时PLT[2]会直接跳转通过GOT[4]跳转到函数而不是PLT[2]的下一条地址，具体见书上的图例。

**先观察调用dl\_init前，动态库函数指向的地址。**



**调用dl\_init后再次查看，经过dl\_init的调用，这里已经有了一段地址：**



## 5.8 本章小结

链接可以在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。

链接器处理称为目标文件的二进制文件，它有三种不同的形式：可重定位的、可撕的和共享的：

可重定位的目标文件由静态链接器合并成一个可执行的目标文件，它可以加载到存储器中并执行。

共享目标文件（共享库）是在运行时由动态链接器链接和加载的，或者隐含地在调用程序被加载和开始执行时或者根据需要在程序调用dopen库的函数时。

链接器的两个主要任务是符号解析和重定位，符号解析将目标文件中的每个全局符号都绑定到一个唯一的定义，而重定位确定每个符号的最终存储器地址，并修改对那些目标的引用。

多个目标文件可以定义相同的符号，而链接器用来悄悄地解析这些多重定义的规则可能在用户程序中引入的微妙错误。

**（第5章1分）**

# 第6章 hello进程管理

## 6.1 进程的概念与作用

**概念：**

进程的经典定义就是：一个执行中程序的实例。

一也就是一个具有一定独立功能的程序关于某个数据集合的一次运行活动，是系统进行资源分配和调度运行的基本单位。

系统中的每个程序都运行在某个进程的上下文中，一般情况下，包括文本区域（执行代码）、数据区域、和堆栈等。

**作用：**

为用户提供了假象：我们的程序好像是系统唯一运行的程序，单独占用内存和处理器。

## 6.2 简述壳Shell-bash的作用与处理流程

作用：代表用户运行其他程序。

处理流程：

1）从终端读入输入的命令。

2）将输入字符串切分获得所有的参数

3）如果是内置命令则立即执行

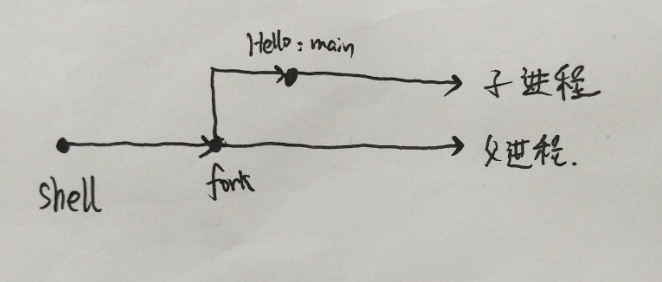
4）否则调用相应的程序为其分配子进程并运行

5）shell应该接受键盘输入信号，并对这些信号进行相应处理

## 6.3 Hello的fork进程创建过程

Shell通过调用fork 函数创建一个新的运行的子进程，也就是Hello程序，Hello进程几乎但不完全与Shell相同。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。Sehll和Hello进程之间最大的区别在于它们有不同的PID。

如图所示：

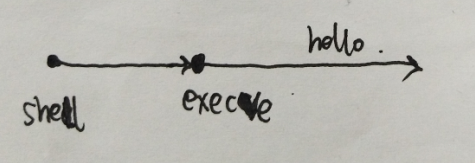


## 6.4 Hello的execve过程

创建进程后，在子进程中通过判断pid即fork()函数的返回值，判断处于子进程，则会通过execve函数在当前进程的上下文中加载并运行一个新程序。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。所以，与fork 一次调用返回两次不同， execve 调用一次并从不返回。

在execve加载了可执行程序之后，它调用启动代码。启动代码设置栈，并将控制传递给新程序的主函数，即可执行程序的main函数。此时用户栈已经包含了命令行参数与环境变量，进入main函数后便开始逐步运行程序。

具体如图所示：



## 6.5 Hello的进程执行

**上下文信息：**

**进程的上下文切换由如下组成：**

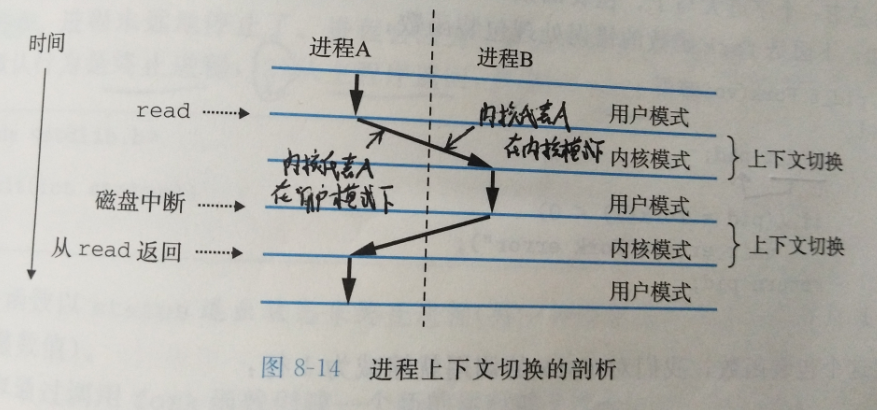
1）决定是否作上下文切换以及是否允许作上下文切换。包括对进程调度原因的检查分析，以及当前执行进程的资格和CPU执行方式的检查等。在操作系统中，上下文切换程序并不是每时每刻都在检查和分析是否可作上下文切换，它们设置有适当的时机。（2）保存当前执行进程的上下文。这里所说的当前执行进程，实际上是指调用上下文切换程序之前的执行进程。如果上下文切换不是被那个当前执行进程所调用，且不属于该进程，则所保存的上下文应是先前执行进程的上下文，或称为“老”进程上下文。显然，上下文切换程序不能破坏“老”进程的上下文结构。（3）使用进程调度算法，选择一处于就绪状态的进程。（4）恢复或装配所选进程的上下文，将CPU控制权交到所选进程手中。

**进程时间片：**

一个进程执行它的控制流的一部分的每一时间段叫做时间片。

**用户态与内核态转换：**

**首先看一下书上的用户态与内核态转换的解释（中断处理）：**



在处理操作系统陷入时，

1，内核根据系统调用号查系统调用入口表，找到相应的内核子程序的地址。

2，内核还要确定该系统调用所要求的参数个数。

3，从用户地址空间拷贝参数到U区（Unix V）。

4，保存当前上下文，执行系统调用代码。

核心态：当CPU正在运行内核代码时（内核代码是共享的）。

用户态：当CPU正在运行用户代码时。

用户模式：不可以访问内核空间（>=0x80000000)

内核模式：可以访问任何有效虚拟地址，包括内核空间。一个线程可以访问其他任何线程地址空间。

对于hello程序来说：

子进程通过execve 系统调用启动加载器。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(chunk), 新的代码和数据段袚初始化为可执行文件的内容。最后，加载器跳转到\_start地址，它最终会调用应用程序的main 函数。

再看hello程序执行时对sleep函数的分析：

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

## 6.6 hello的异常与信号处理

**6.6.1 hello执行过程中会出现哪几类异常：**

Hello在执行的过程中，可能会出现处理器外部I/O设备引起的异常，执行指令导致的陷阱、故障和终止。第一种被称为外部异常，常见的有时钟中断、外部设备的I/O中断等。第二种被称为同步异常。陷阱指的是有意的执行指令的结果，故障是非有意的可能被修复的结果，而终止是非故意的不可修复的致命错误。

**6.6.2 会产生哪些信号，又怎么处理的：**

在发生异常时会产生信号。例如缺页故障会导致OS发生SIGSEGV信号给用户进程，而用户进程以段错误退出。

中断：SIGSTP:挂起程序

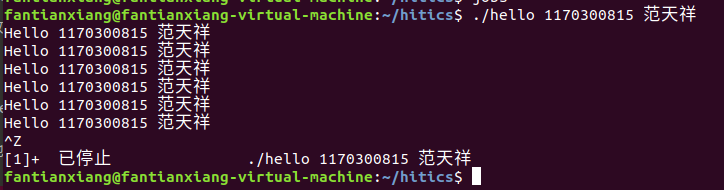
终止：SIGINT:终止程序

**6.6.3 各命令及运行结截屏：**

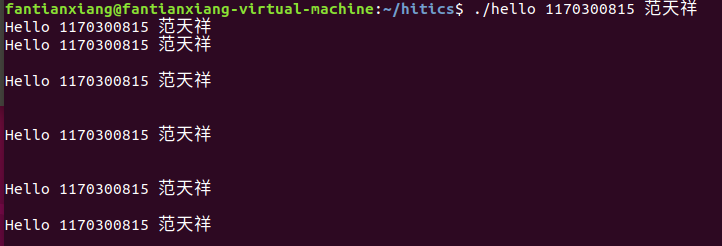
Ctrl-C:直接终止进程：



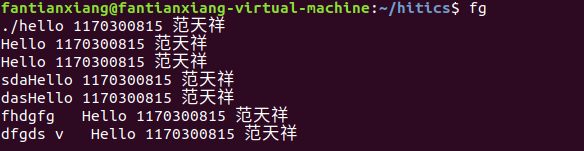
Ctrl-Z:如果在程序运行过程中输入Ctrl-Z，那么会发送一个 SIGTSTP 信号给前台进程组中的进程，从而将其挂起:



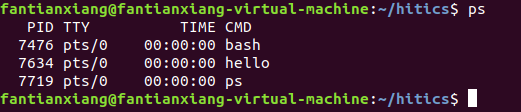
回车程序会忽略：



随机按键盘：（不响应）



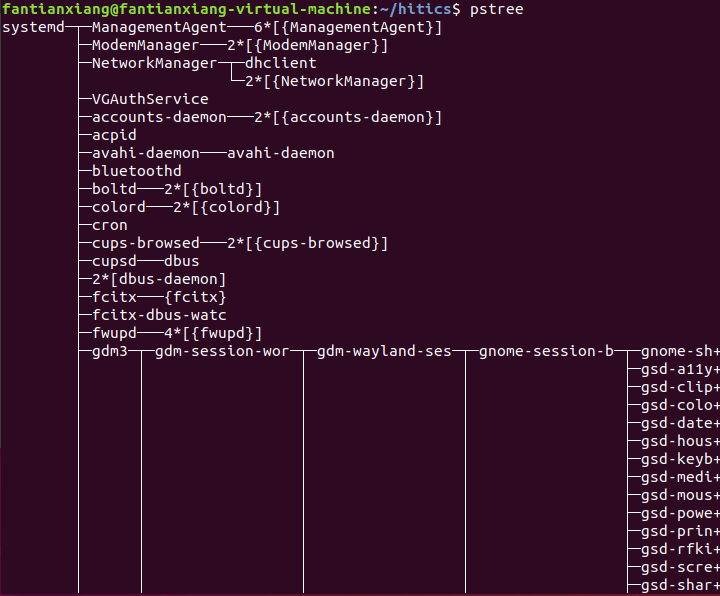
ps: 查看进程及其运行时间



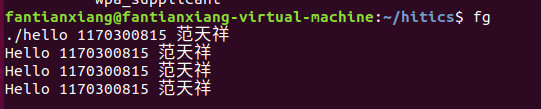
jobs: 查看当前暂停的进程



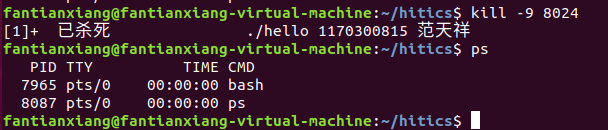
pstree（部分截图）:



fg: 输入fg使进程重新在前台执行



kill: 使用kill杀死特定进程



## 6.7本章小结

shell中执行是通过fork函数及execve创建新的进程并执行程序。进程拥有着与父进程相同却又独立的环境，与其他系统进并发执行，拥有各自的时间片，在内核的调度下有序进行程序的执行。

在应用层，一个进程可以发送信号到另一个进程，而接收者会将控制突然转移到它的一个信号处理程序。一个程序可以通过回避通常的栈规则，并执行到其他函数中任意位置的非本地跳转来对错误做出反应。

异常分为中断、陷阱、故障和终止四类，均有对应的处理方法。操作系统提供了信号这一机制，实现了异常的反馈。这样，程序能够对不同的信号调用信号处理子程序进行处理。

**（第6章1分）**

第7章 hello的存储管理

## 7.1 hello的存储器地址空间

**逻辑地址概念：**

逻辑地址是指由程序产生的与段相关的偏移地址部分。

对应于hello.o里面的相对偏移地址。

**线性地址：**

逻辑地址经过段机制后转化为线性地址，为描述符:偏移量的组合形式。分页机制中线性地址作为输入。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。Intel 80386的线性地址空间容量为4G（2的32次方即32根地址总线寻址）。

**虚拟地址：**

现代操作系统都提供了一种内存管理的抽像，即虚拟内存。进程使用虚拟内存中的地址，即虚拟地址，由操作系统协助相关硬件，把它“转换”成真正的物理地址。hello.s中使用的就是虚拟空间的虚拟地址。

**物理地址：**

在实地址方式下，物理地址是通过段地址乘以16加上偏移地址得到的。而16位的段地址乘以16等同于左移4位二进制位，这样变成20位的段基地址，最后段基地址加上段内偏移地址即可得到物理地址。

20位物理地址计算方法：物理地址=段地址\*16d+偏移地址。

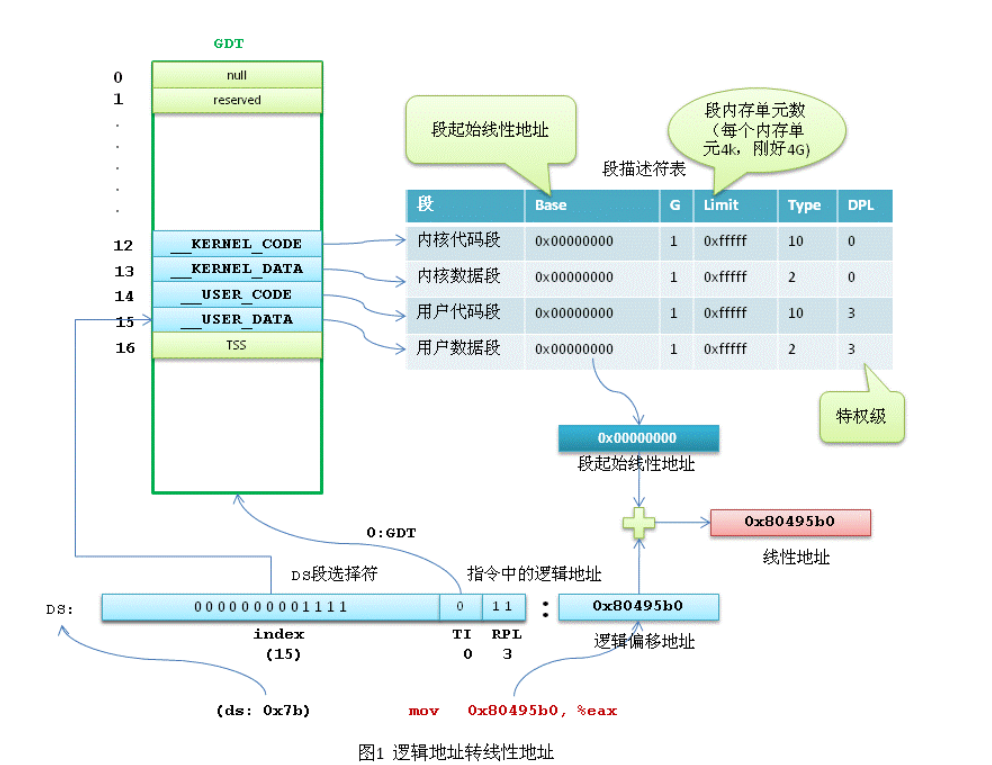
是指出目前CPU外部地址总线上的寻址物理内存的地址信号，是地址变换的最终结果地址。如果启用了分页机制，那么线性地址会使用页目录和页表中的项变换成物理地址。如果没有启用分页机制，那么线性地址就直接成为物理地址了。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

机器语言指令中出现的内存地址，都是逻辑地址，需要转换成线性地址，再经过MMU(CPU中的内存管理单元)转换成物理地址才能够被访问到。

在x86保护模式下，段的信息（段基线性地址、长度、权限等）即段描述符占8个字节，段信息无法直接存放在段寄存器中（段寄存器只有2字节）。Intel的设计是段描述符集中存放在GDT或LDT中，而段寄存器存放的是段描述符在GDT或LDT内的索引值(index)。

逻辑地址，必须加上隐含的DS 数据段的基地址，才能构成线性地址。具体如下所示：



## 7.3 Hello的线性地址到物理地址的变换-页式管理

在保护模式下，控制寄存器CR0的最高位PG位控制着分页管理机制是否生效，如果PG=1，分页机制生效，需通过页表查找才能把线性地址转换物理地址。如果PG=0，则分页机制无效，线性地址就直接做为物理地址。

分页的基本原理是把内存划分成大小固定的若干单元，每个单元称为一页（page），每页包含4k字节的地址空间（为简化分析，我们不考虑扩展分页的情况）。这样每一页的起始地址都是4k字节对齐的。为了能转换成物理地址，我们需要给CPU提供当前任务的线性地址转物理地址的查找表，即页表(page table)。注意，为了实现每个任务的平坦的虚拟内存，每个任务都有自己的页目录表和页表。

为了节约页表占用的内存空间，x86将线性地址通过页目录表和页表两级查找转换成物理地址。

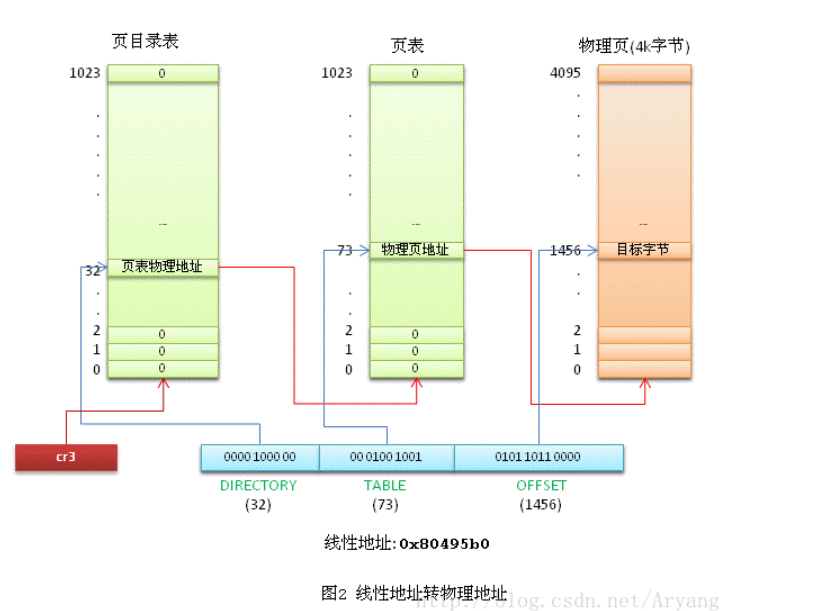
32位的线性地址被分成3个部分：

最高10位 Directory 页目录表偏移量，中间10位 Table是页表偏移量，最低12位Offset是物理页内的字节偏移量。

页目录表的大小为4k（刚好是一个页的大小），包含1024项，每个项4字节（32位），项目里存储的内容就是页表的物理地址。如果页目录表中的页表尚未分配，则物理地址填0。

页表的大小也是4k，同样包含1024项，每个项4字节，内容为最终物理页的物理内存起始地址。

具体实现如图所示：

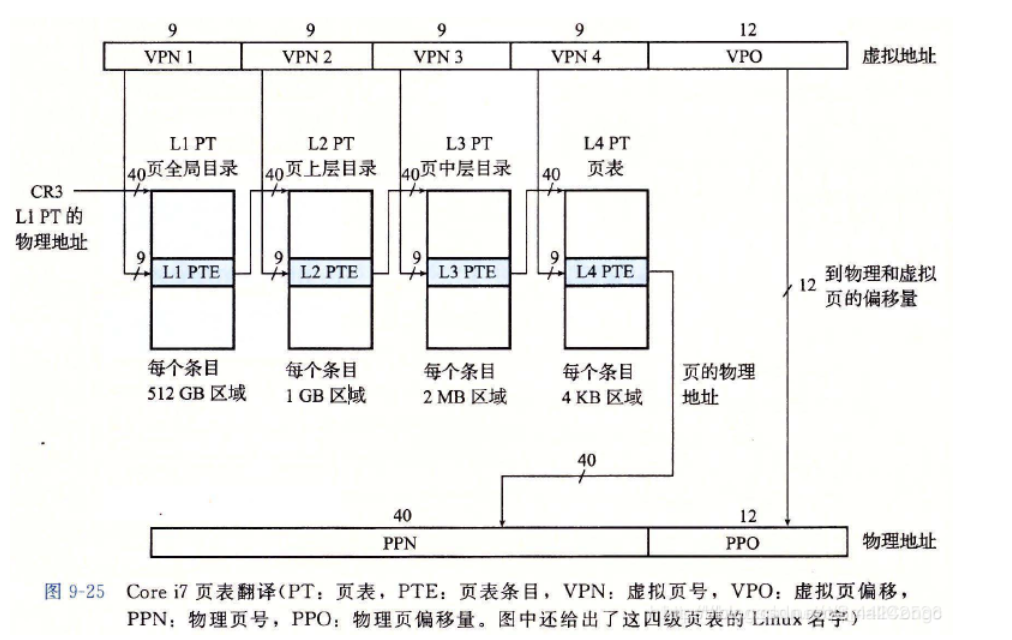


## 7.4 TLB与四级页表支持下的VA到PA的变换

页表一般都很大，并且存放在内存中，所以处理器引入MMU后，读取指令、数据需要访问两次内存：首先通过查询页表得到物理地址，然后访问该物理地址读取指令、数据。为了减少因为MMU导致的处理器性能下降，引入了TLB，TLB是Translation Lookaside Buffer的简称，可翻译为“地址转换后援缓冲器”，也可简称为“快表”。简单地说，TLB就是页表的Cache，其中存储了当前最可能被访问到的页表项，其内容是部分页表项的一个副本。只有在TLB无法完成地址翻译任务时，才会到内存中查询页表，这样就减少了页表查询导致的处理器性能下降。

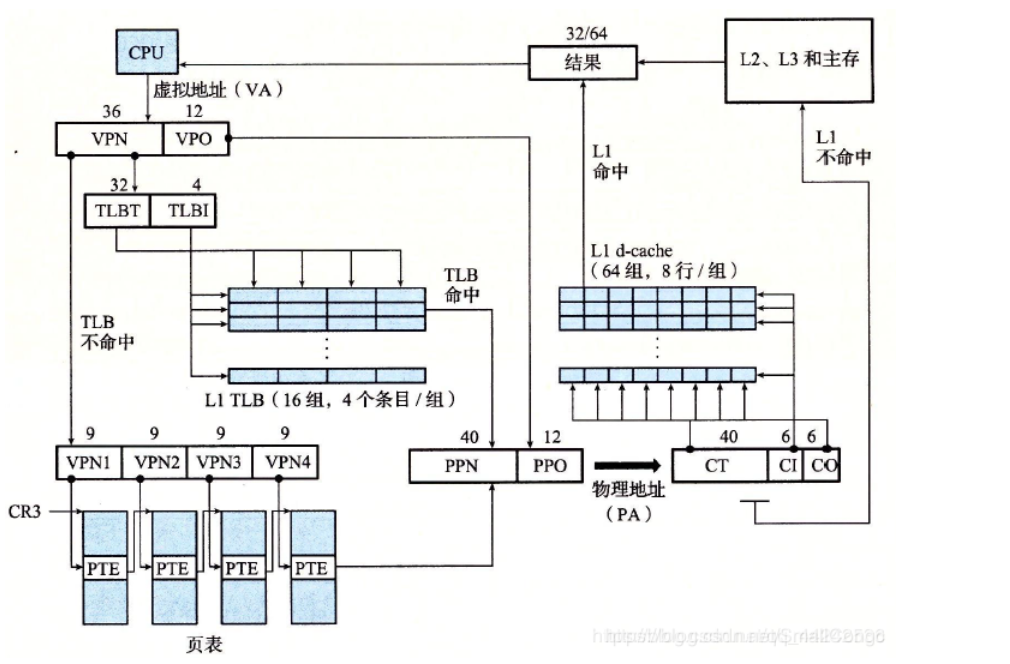
      TLB中的项由两部分组成：标识和数据。标识中存放的是虚地址的一部分，而数据部分中存放物理页号、存储保护信息以及其他一些辅助信息。虚地址与TLB中项的映射方式有三种：全关联方式、直接映射方式、分组关联方式。

Core i7 MMU 使用四级的页表将虚拟地址翻译成物理地址。36位VPN 被划分成四个9 位VPN，分别用于一个页表的偏移量。具体结构如图所示。



## 7.5 三级Cache支持下的物理内存访问

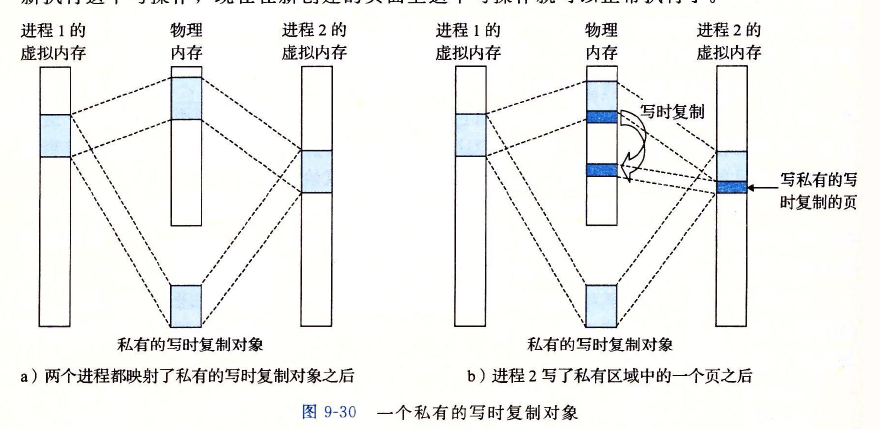
首先CPU发出一个虚拟地址，在TLB里面寻找。如果命中，那么将PTE发送给L1Cache，否则先在页表中更新PTE。然后再进行L1根据PTE寻找物理地址，检测是否命中的工作。这样就能完成Cache和TLB的配合工作。具体流程如图所示：



## 7.6 hello进程fork时的内存映射

shell通过fork为hello创建新进程。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hello进程唯一的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、区域结构和样表的原样副本。它将两个进程中的每个页面都标记为只读，并将每个进程中的每个区域结构都标记为写时复制。在新进程中返回时，新进程拥有与调用fork进程相同的虚拟内存，随后的写操作通过写时复制机制创建新页面。

具体实现如图所示：



## 7.7 hello进程execve时的内存映射

首先删除已存在的用户区域，创建新的区域结构，代码和初始化数据映射到.text和.data区（目标文件提供），.bss和栈映射到匿名文件，设置PC，指向代码区域的入口点。Linux根据需要换入代码和数据页面。下一步是映射共享区域，将一些动态链接库映射到hello的虚拟地址空间，最后设置程序计数器，使之指向hello程序的代码入口。

## 7.8 缺页故障与缺页中断处理

**缺页故障：**进程线性地址空间里的页面不必常驻内存，在执行一条指令时，如果发现他要访问的页没有在内存中（即存在位为0），那么停止该指令的执行，并产生一个页不存在的异常，对应的故障处理程序可通过从外存加载该页的方法来排除故障，之后，原先引起的异常的指令就可以继续执行，而不再产生异常。

**产生缺页中断的几种情况：**

1、当内存管理单元（MMU）中确实没有创建虚拟物理页映射关系，并且在该虚拟地址之后再没有当前进程的线性区（vma）的时候，可以肯定这是一个编码错误，这将杀掉该进程；

2、当MMU中确实没有创建虚拟页物理页映射关系，并且在该虚拟地址之后存在当前进程的线性区vma的时候，这很可能是缺页中断，并且可能是栈溢出导致的缺页中断；

3、当使用malloc/mmap等希望访问物理空间的库函数/系统调用后，由于linux并未真正给新创建的vma映射物理页，此时若先进行写操作，将和2产生缺页中断的情况一样；若先进行读操作虽然也会产生缺页异常，将被映射给默认的零页，等再进行写操作时，仍会产生缺页中断，这次必须分配1物理页了，进入写时复制的流程；

4、当使用fork等系统调用创建子进程时，子进程不论有无自己的vma，它的vma都有对于物理页的映射，但它们共同映射的这些物理页属性为只读，即linux并未给子进程真正分配物理页，当父子进程任何一方要写相应物理页时，导致缺页中断的写时复制；

**缺页中断处理：**

缺页中断处理一般流程：

1.硬件陷入内核，在堆栈中保存程序计数器，大多数当前指令的各种状态信息保存在特殊的cpu寄存器中。

2.启动一个汇编例程保存通用寄存器和其他易丢失信息，以免被操作系统破坏。

3.当操作系统发现缺页中断时，尝试发现需要哪个虚拟页面。通常一个硬件寄存器包含了这些信息，如果没有的话操作系统必须检索程序计数器，取出当前指令，分析当前指令正在做什么。

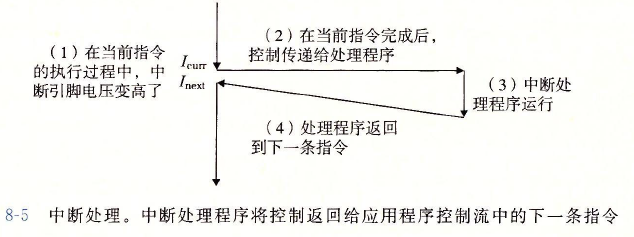
4.一旦知道了发生缺页中断的虚拟地址，操作系统会检查地址是否有效，并检查读写是否与保护权限一致，不过不一致，则向进程发一个信号或者杀死该进程。如果是有效地址并且没有保护错误发生则系统检查是否有空闲页框。如果没有，则执行页面置换算法淘汰页面。

5.调度引发缺页中断的进程，操作系统返回调用他的汇编例程

6.该例程恢复寄存器和其他状态信息，返回到用户空间继续执行，就好像缺页中断没有发生过。

等等。

**抽象图：**



## 7.9动态存储分配管理

**概念：**在进程运行的时候，动态存储器分配器维护着一个进程的虚拟存储器区域，称为堆。分配器将对视为一组大小不同的块（block）的集合来分配。每个块就是一个连续的虚拟存储器片（chunk），要么是已经分配的，要么是还没有分配的。已经分配的，显示的保留给应用程序使用（malloc得到一段内存）。空闲的，则是可以继续分配。一个已经分配的块保持已经分配的转态，直到被free掉，得以被堆重用。当已经分配的块一直没有被free，那么一直保持分配转态，这个就是内存泄漏了。

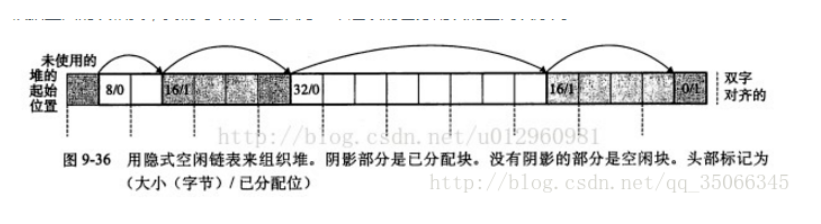
**分配器有两种基本风格：**

显示分配器：要求应用显示的释放已经分配的块，像C，C++语言采用的机制。

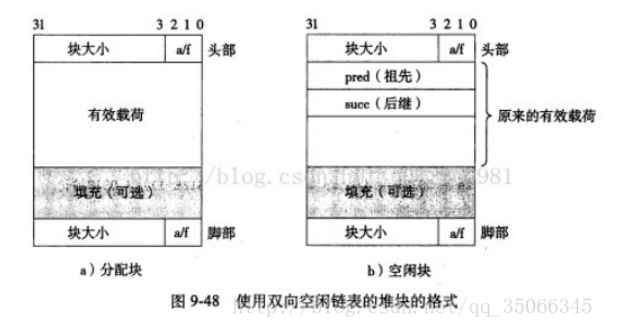
隐式分配器：分配器检测一个已经分配的块何时不再被程序所使用时，自动free掉。这个就是垃圾收集。

**记录空闲块：**

隐式：



显示空闲链表：



分离空闲链表：简单分离存储、分离适配、伙伴系统

**Printf会调用malloc：**printf会引用全局变量stdout，malloc。

## 7.10本章小结

1、存储器管理的主要任务

　　存储器管理的主要任务是为多道程序的并发运行提供皮好的存储器环境。它包括以下内容：

　　(1)能让每道程序“各得其所”，并在不受干扰的环境中运行，还可以使用户从存储空间的分配、保护等烦琐事务中解脱出来;

　　(2)向用户提供更大的存储空间，使更多的作业能同时投入运行或使更大的作业能在较小的内存空间中运行;

　　(3)为用户对信息的访问、保护、共享以及动态链接等方面提供方便;

　　(4)能使存储器有较高的利用率。

2、存储器管理的主要功能

　　为了实现存储器管理的主要任务，存储器管理应具有以下几个方面的功能。

　　(1)内存分配。根据分配策略，为多道程序分配内存，并实现共享。同时对程序释放的存储空间进行回收。

　　(2)地址映射。每道程序都有自己的逻辑地址，在多道程序环境中，内存空间被多道程序共享，这就必然导致程序的逻辑地址与在内存中的物理地址不一致。因此，存储器管理必须提供地址映射功能，用于逻辑地址和物理地址间的变换。地址映射通常在硬件支持下完成。

　　(3)内存保护。确保每道程序在自己的内存空间中运行，互不干扰。内存保护一般由硬件完成。

　　(4)内存扩充。利用虚拟存储技术从逻辑上扩充内存空间，为用户营造一个比实际物理内存更大的存储空间。程序运行过程中常常涉及到动态内存分配，动态内存分配通过动态内存分配器完成，能够对堆空间进行合理地分配与管理，分割与合并。现代使程序内存分配器采取多种策略来提高吞吐量以及内存占用率，从在灵活使用内存的基础上保证了效率。

**（第7章 2分）**

# 第8章 hello的IO管理

## 8.1 Linux的IO设备管理方法

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

设备管理：是通过unix io接口实现的。

## 8.2 简述Unix IO接口及其函数

**linu 提供如下 IO 接口：**

read 和 write -- 最简单的读写函数

readn 和 writen -- 原子性读写操作

recvfrom 和 sendto -- 增加了目标地址和地址结构长度的参数

recv 和 send -- 允许从进程到内核传递标志

readv 和 writev -- 允许指定往其中输入数据或从其中输出数据的缓冲区

recvmsg 和 sendmsg -- 结合了其他IO函数的所有特性，并具备接受和发送辅助数据的能力

**Unix I/O接口提供了以下函数供应用程序调用：**

打开文件：int open(char \*filename, int flags, mode\_t mode);

关闭文件：int close(int fd);

读文件：ssize\_t read(int fd, void \*buf, size\_t n);

写文件：ssize\_t write(int fd, const void \*buf, size\_t n);

## 8.3 printf的实现分析

**printf函数的函数体**  
int printf(const char \*fmt, ...)   
{   
int i;   
char buf[256];   
      
     va\_list arg = (va\_list)((char\*)(&fmt) + 4);   
     i = vsprintf(buf, fmt, arg);   
     write(buf, i);   
      
     return i;   
    }

**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。

**Write系统函数**：

write:

mov eax, \_NR\_write

mov ebx, [esp + 4]

mov ecx, [esp + 8]

int INT\_VECTOR\_SYS\_CALL

**write有个参数：1**

1表示的是tty所对应的一个文件句柄。

在linux里，所有设备都是被当作文件来看待的，这个1就是表示往当前显示器里写入数据

int INT\_VECTOR\_SYS\_CALL表示要通过系统来调用**sys\_call这个函数**。

**最后：**printf()函数不能确定参数，只会根据format中的打印格式的数目依次打印堆栈中参数format后面地址的内容。这样就存在一个**可能的缓冲区溢出问题**。

syscall将字符串中的字节“Hello 1170300815 范天祥”从寄存器中通过总线复制到显卡的显存中，显存中存储的是字符的ASCII码。

字符显示驱动子程序将通过ASCII码在字模库中找到点阵信息将点阵信息存储到vram中。

显示芯片会按照一定的刷新频率逐行读取vram，并通过信号线向液晶显示器传输每一个点（RGB分量）。

于是打印字符串“Hello 1170300825 范天祥”就显示在了屏幕上。

## 8.4 getchar的实现分析

首先，getchar()是标准I/O标准库里的库函数，其原型是

int getchar(void)

它没有参数，原因是因为它是从stdin标准输入流中读入一个字符的函数，已经有了默认的流参数stdin了。其返回值是一个整型数，是用来表示字符用的。

在C语言中有个重要的库函数getchar()，可从终端获得一个字符的ASCII码值。在终端输入字符时并非输入一个字符就会返回，而是在遇到回车换行前，所有输入的在C语言中有个重要的库函数getchar()，可从终端获得一个字符的ASCII码值。在终端输入字符时并非输入一个字符就会返回，而是在遇到回车换行前，所有输入的字符都会缓冲在键盘缓冲器中，直到回车换行一次性将所有字符按序依次赋给相应的变量，在这里一定要注意最后一个字符即'\n'，该字符也会赋给一个相应的变量。

当用getchar进行输入时，如果输入的第一个字符为有效字符(即输入是文件结束符EOF，Windows下为组合键Ctrl+Z， Unix/Linux下为组合键Ctrl+D)，那么只有当最后一个输入字符为换行符'\n'(也可以是文件结束符EOF，EOF将在后面讨论)时， getchar才会停止执行，整个程序将会往下执行。

当终端有字符输入时，Ctrl+D产生的EOF相当于结束本行的输入，将引起getchar()新一轮的输入；当终端没有字符输入或者可以说当getchar()读取新的一次输入时，输入Ctrl+D，此时产生的EOF相当于文件结束符，程序将结束getchar()的执行。字符都会缓冲在键盘缓冲器中，直到回车换行一次性将所有字符按序依次赋给相应的变量，在这里一定要注意最后一个字符即'\n'，该字符也会赋给一个相应的变量。

getchar函数通过调用read函数返回字符。read函数的返回值是读入的字符数，如果为1说明读入成功，那么直接返回字符，否则说明读到了缓冲区的最后。

read函数同样通过sys\_call中断来调用内核中的系统函数。键盘中断处理子程序会接受按键扫描码并将其转换为ASCII码后保存在缓冲区。然后read函数调用的系统函数可以对缓冲区ASCII码进行读取，直到接受回车键返回。这样，getchar函数通过read函数返回字符，实现了读取一个字符的功能。

## 8.5本章小结

输入/输出 I/O 是主存和外部设备如终端、网络之间拷贝数据的过程。输入操作是从I/O设备拷贝数据到主存。输出操作是从主存拷贝数据到I/O设备。

在UNIX中所有的I/O设备都被模型化为文件。并提供Unix I/O接口。通过这个接口，程序能够进行输入与输出，只需要找到描述符，底层硬件实现操作系统就可以实现。

**（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。）**

**总结：**

**1. 计算机系统**

我们知道计算机系统是由硬件和软件组成的。它们共同工作来运行应用程序。虽然系统的实现方式随着时间不断变化，但是系统内在的概念却没有改变。所有计算机系统都有相似的硬件和软件组件，它们执行着相似的功能，我们只有深入了解这些组件是如何工作的，以及这些组件是如何影响程序的正确性和性能的，才能写出高质量的代码。

**2. hello.c**

ASCII 码构成就是用一个唯一的单字节大小的整数来表示每个字符，以字节序列的方式存储在文件中。

**3. 程序的编译**

hello 程序的生命周期是从一个高级 C 语言程序开始的，因为这种形式能被人读懂。然而，计算机系统是读不懂高级语言的。为了在系统上运行 hello.c 程序，每条 C 语句都必须要被其他程序转化为一系列的低级机器语言指令。

　　一般来说，要将 hello.c 变成一个可执行的目标程序，必须要经过预处理器、编译器、汇编器和链接器的处理。

预处理器、编译器、汇编器和链接器 一起构成了编译系统，下面对每个步骤分别进行解析：

　　①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，比如 Hello.c 中第一行 #include<studio.h> 命令告诉预处理器读取系统文件 stdio.h 的内容，并把它直接插入到程序中。结果就得到另一个 C 程序，通常是以 .i 作为文件扩展名。

　　②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。汇编语言能为不同高级语言的不同编译器提供通用的输出语言。

　　③汇编阶段：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，hello.o 文件是一个二进制文件，它的字节编码是机器预言指令而不是字符。如果我们用文本编辑器打开 hello.o 文件，将会是一堆乱码。

④链接阶段：在 hello.c 程序中，我们看到程序调用了 printf 函数，它是每个 C 编译器都会提供的标准 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 就是负责处理这种合并，结果就得到一个 hello 文件，它是一个可执行目标程序，可以被加载到内存中，由系统运行。

**静态链接：**

在 程序执行之前就完成链接工作。也就是等链接完成后文件才能执行。但是这有一个明显的缺点，比如说库函数。如果文件A 和文件B 都需要用到某个库函数，链接完成后他们连接后的文件中都有这个库函数。当A和B同时执行时，内存中就存在该库函数的两份拷贝，这无疑浪费了存储空间。当规模扩大的时候，这种浪费尤为明显。静态链接还有不容易升级等缺点。为了解决这些问题，现在的很多程序都用动态链接。

**动态链接：**

和静态链接不一样，动态链接是在程序执行的时候才进行链接。也就是当程序加载执行的时候。还是上面的例子，如果A和B都用到了库函数Fun()，A和B执行的时候内存中就只需要有Fun()的一个拷贝。

关于链接还有很多知识，这里只涉及皮毛。

**4.装载：**

我们知道，程序要运行是必然要把程序加载到内存中的。在过去的机器里都是把整个程序都加载进入物理内存中，现在一般都采用了虚拟存储机制，即每个进程都有完整的地址空间，给人的感觉好像每个进程都能使用完成的内存。然后由一个内存管理器把虚拟地址映射到实际的物理内存地址。

按照上文的叙述，程序的地址可以分为虚拟地址和实际地址。虚拟地址即虚拟内存空间中的地址，物理地址就是被加载的实际地址。

加载的过程可以这样理解：先为程序中的各部分分配好虚拟地址，然后再建立虚拟地址到物理地址的映射。其实关键的部分就是虚拟地址到物理地址的映射过程。程序装在完成之后，cpu的程序计数器pc就指向文件中的代码起始位置，然后程序就按顺序执行。

**5.程序的运行：**

想要在 Linux 系统中运行该可执行程序，我们要将它的文件名输入到称为外壳（shell）的应用程序中，外壳是一个命令行解释器，它输出一个提示符，等待你输入一个命令，然后执行这个命令。如果该命令行的第一个单词不是一个内置的外壳命令，那么外壳就会假设这是一个可执行文件的名字，它将加载并运行这个文件。

初始时，外壳程序执行它的指令，等待我们输入一个命令。当我们在键盘上输入字符串"./hello"后，外壳程序将字符逐一读入到寄存器中，再把它放入到存储器中。

当我们在键盘上敲回车键的时候，外壳程序知道我们已经结束了命令的输入。然后外壳执行一系列指令来加载可执行的 hello 文件，将 hello 目标文件中的代码和数据从磁盘复制到主存。数据包括最终会被输出的字符串“Hello 1170300815 范天祥\n”，一旦目标文件中的代码和数据被加载到主存，处理器就开始执行 hello 程序的 main 程序中的机器语言指令。这些指令将“Hello 1170300815 范天祥\n” 字符串中的字节从主存复制到寄存器文件，再从寄存器文件中复制到显示设备，最终显示在屏幕上。

**感悟：**

**写这篇报告的目的在于梳理程序运行的机制，在一个可执行文件执行的背后都隐藏了什么。**

**从源代码到可执行文件通常要经历许多中间步骤，每一个中间步骤都生成一个中间文件。只是现在的集成开发环境都把这些步骤隐藏了，习惯于集成开发环境的我们也就逐渐地忽略了这些重要的技术内幕。这篇报告也只是介绍了一下这个过程的主线而已。其中的每一个细节展开来讲都可足已用一篇文章来论述。**

**这门课让我收获颇丰，她将会伴随我往后的程序生涯，让我受益匪浅。**

**（结论0分，缺失 -1分，根据内容酌情加分）**

# 附件

列出所有的中间产物的文件名，并予以说明起作用。

|  |  |
| --- | --- |
| 文件名称 | 文件作用 |
| hello.c | 提供源程序 |
| hello.i | cpp预处理之后的文件 |
| hello.s | cc1之后的汇编语言格式文件 |
| hello.o | as之后的可重定位目标文件 |
| hello | ld之后的可执行目标文件 |
| helloftx.elf | hello.o的ELF格式文件 |
| hello.objdump | Objdump得到的hello的反汇编代码 |

**（附件0分，缺失 -1分）**

# 参考文献

**为完成本次大作业你翻阅的书籍与网站等**

[1] nm和readelf命令的区别：https://www.cnblogs.com/foohack/p/4103074.html

[2] Linux下链接器ld链接顺序的总结：

https://www.jb51.net/LINUXjishu/236871.html

[3] printf函数实现的深入剖析https://www.cnblogs.com/pianist/p/3315801.html

[4] UNIX IO 操作函数 http://www.techlog.cn/article/list/10182663

[5] 关于C语言中getchar()的详细使用 - Dormant专栏 - CSDN博客

https://blog.csdn.net/meidong52617/article/details/44728517

[6] linux2.6 内存管理——逻辑地址转换为线性地址（逻辑地址、线性地址、物理地址、虚拟地址）http://www.cnblogs.com/diaohaiwei/p/5094959.html

[7] Linux内存管理：逻辑地址到线性地址和物理地址的转换 - pi9nc的专栏 - CSDN博客 https://blog.csdn.net/pi9nc/article/details/21031651

[8] 详解缺页中断-----缺页中断处理（内核、用户）- 柯南的博客 - CSDN博客 https://blog.csdn.net/m0\_37962600/article/details/81448553

[9] 九、完善堆内系统调用：Linux系统调用、printf的内部实现、malloc的内部实现- zhangyang249的博客 – CSDN博客

https://blog.csdn.net/zhangyang249/article/details/78582809

[10] 从hello world 解析程序运行机制 - 云+社区 - 腾讯云

https://cloud.tencent.com/developer/article/1021739

**（参考文献0分，缺失 -1分）**