

**计算机系统**

**大作业**

题 目 程序人生-Hello’s P2P

专 业 计算机专业

学　　 号 1172510217

班　　 级 1703002

学 生 张景润

指 导 教 师 史先俊

**计算机科学与技术学院**

**2018年12月**

**摘 要**

本论文将CSAPP课程所学内容通过hello小程序的一生，对我们所学进行全面的梳理与回顾。我们主要在Ubuntu下进行相关操作，合理运用了Ubuntu下的操作工具，进行细致的历程分析，目的是加深对计算机系统的了解。

**关键词：**历程；hello程序；知识梳理；Ubuntu；

**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 - 5 -](#_Toc532238405)

[第3章 编译 - 6 -](#_Toc532238406)

[3.1 编译的概念与作用 - 6 -](#_Toc532238407)

[3.2 在Ubuntu下编译的命令 - 6 -](#_Toc532238408)

[3.3 Hello的编译结果解析 - 6 -](#_Toc532238409)

[3.4 本章小结 - 6 -](#_Toc532238410)

[第4章 汇编 - 7 -](#_Toc532238411)

[4.1 汇编的概念与作用 - 7 -](#_Toc532238412)

[4.2 在Ubuntu下汇编的命令 - 7 -](#_Toc532238413)

[4.3 可重定位目标elf格式 - 7 -](#_Toc532238414)

[4.4 Hello.o的结果解析 - 7 -](#_Toc532238415)

[4.5 本章小结 - 7 -](#_Toc532238416)

[第5章 链接 - 8 -](#_Toc532238417)

[5.1 链接的概念与作用 - 8 -](#_Toc532238418)

[5.2 在Ubuntu下链接的命令 - 8 -](#_Toc532238419)

[5.3 可执行目标文件hello的格式 - 8 -](#_Toc532238420)

[5.4 hello的虚拟地址空间 - 8 -](#_Toc532238421)

[5.5 链接的重定位过程分析 - 8 -](#_Toc532238422)

[5.6 hello的执行流程 - 8 -](#_Toc532238423)

[5.7 Hello的动态链接分析 - 8 -](#_Toc532238424)

[5.8 本章小结 - 9 -](#_Toc532238425)

[第6章 hello进程管理 - 10 -](#_Toc532238426)

[6.1 进程的概念与作用 - 10 -](#_Toc532238427)

[6.2 简述壳Shell-bash的作用与处理流程 - 10 -](#_Toc532238428)

[6.3 Hello的fork进程创建过程 - 10 -](#_Toc532238429)

[6.4 Hello的execve过程 - 10 -](#_Toc532238430)

[6.5 Hello的进程执行 - 10 -](#_Toc532238431)

[6.6 hello的异常与信号处理 - 10 -](#_Toc532238432)

[6.7本章小结 - 10 -](#_Toc532238433)

[第7章 hello的存储管理 - 11 -](#_Toc532238434)

[7.1 hello的存储器地址空间 - 11 -](#_Toc532238435)

[7.2 Intel逻辑地址到线性地址的变换-段式管理 - 11 -](#_Toc532238436)

[7.3 Hello的线性地址到物理地址的变换-页式管理 - 11 -](#_Toc532238437)

[7.4 TLB与四级页表支持下的VA到PA的变换 - 11 -](#_Toc532238438)

[7.5 三级Cache支持下的物理内存访问 - 11 -](#_Toc532238439)

[7.6 hello进程fork时的内存映射 - 11 -](#_Toc532238440)

[7.7 hello进程execve时的内存映射 - 11 -](#_Toc532238441)

[7.8 缺页故障与缺页中断处理 - 11 -](#_Toc532238442)

[7.9动态存储分配管理 - 11 -](#_Toc532238443)

[7.10本章小结 - 12 -](#_Toc532238444)

[第8章 hello的IO管理 - 13 -](#_Toc532238445)

[8.1 Linux的IO设备管理方法 - 13 -](#_Toc532238446)

[8.2 简述Unix IO接口及其函数 - 13 -](#_Toc532238447)

[8.3 printf的实现分析 - 13 -](#_Toc532238448)

[8.4 getchar的实现分析 - 13 -](#_Toc532238449)

[8.5本章小结 - 13 -](#_Toc532238450)

[结论 - 14 -](#_Toc532238451)

[附件 - 15 -](#_Toc532238452)

[参考文献 - 16 -](#_Toc532238453)

# 第1章 概述

## 1.1 Hello简介

1）P2P简介

用户通过Editor编写代码，得到hello.c程序；在Linux的Ubuntu操作系统下，调用C预处理器（C Pre-Processor）得到ASCII码的中间文件hello.i；接着调用C编译器（ccl）得到ASCII汇编语言文件hello.s；然后运行汇编器（as）得到可重定位目标文件hello.o；最后通过链接器（ld）得到可执行目标文件hello。用户在Ubuntu shell键入./hello启动此程序，shell调用fork函数为其产生子进程，hello便成为了进程（process）。

2）O2O 简介

OS的进程管理调用fork函数产生子进程（process），调用execve函数，并进行虚拟内存映射（mmp），并为运行的hello分配时间片以执行取指译码流水线等操作；OS的储存管理以及MMU解决VA到PA的转换，cache、TLB、页表等加速访问过程，IO管理与信号处理综合软硬件对信号等进行处理；程序结束时，shell回收hello进程，内核将其所有痕迹从系统中清除。

## 1.2 环境与工具

硬件工具：X64 AMD A10 CPU，8GRAM，512GHD DISK

软件工具：Windows10 64位，Vmware 14.1.3，Ubuntu18.04.1 LTS

开发者与调试工具：gcc，gdb，edb，Winhex，vim，ld，readelf，objdump，ldd等

## 1.3 中间结果

|  |  |  |
| --- | --- | --- |
| 中间结果文件 | 文件作用 | 使用时期 |
| hello.i | 预处理得到的文件  ASCII码的中间文件 | 第二章-预处理 |
| hello.s | ASCII汇编语言文件 | 第三章-编译 |
| hello.o | as得到可重定位目标文件 | 第四章-汇编 |
| hello.asm | 反汇编得到的文本文件 | 第四章-汇编 |
| hello.elf | hello.o的elf文件 | 第四章-汇编 |
| hello | ld得到可执行目标文件 | 第五章-链接 |
| hello\_elf | hello的elf文件 | 第五章-链接 |
| hello\_asm | hello的反汇编文件 | 第五章-链接 |

## 1.4 本章小结

本章首先简要介绍了hello.c的P2P与O2O，然后对整篇论文写作过程中对hello文件操作的所有需要的环境与工具进行了大致的列举，最后对实验过程中用到的所有中间文件及其作用和使用时期以一个表格的形式进行了大致的展现。

# 第2章 预处理

## 2.1 预处理的概念与作用

**预处理的概念**

预处理是在编译之前进行的处理，一般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，预处理中会展开以#起始的行，试图解释为预处理指令(preprocessing directive)。

预处理过程扫描源代码，对其进行初步的转换，产生新的源代码提供给编译器。预处理过程读入源代码，检查包含预处理指令的语句和宏定义，并对源代码进行响应的转换。预处理过程还会删除程序中的注释和多余的空白字符。

**预处理的作用**：扩展源代码，插人所有用#include命令指定的文件。并扩展所有用#define声明指定的宏。

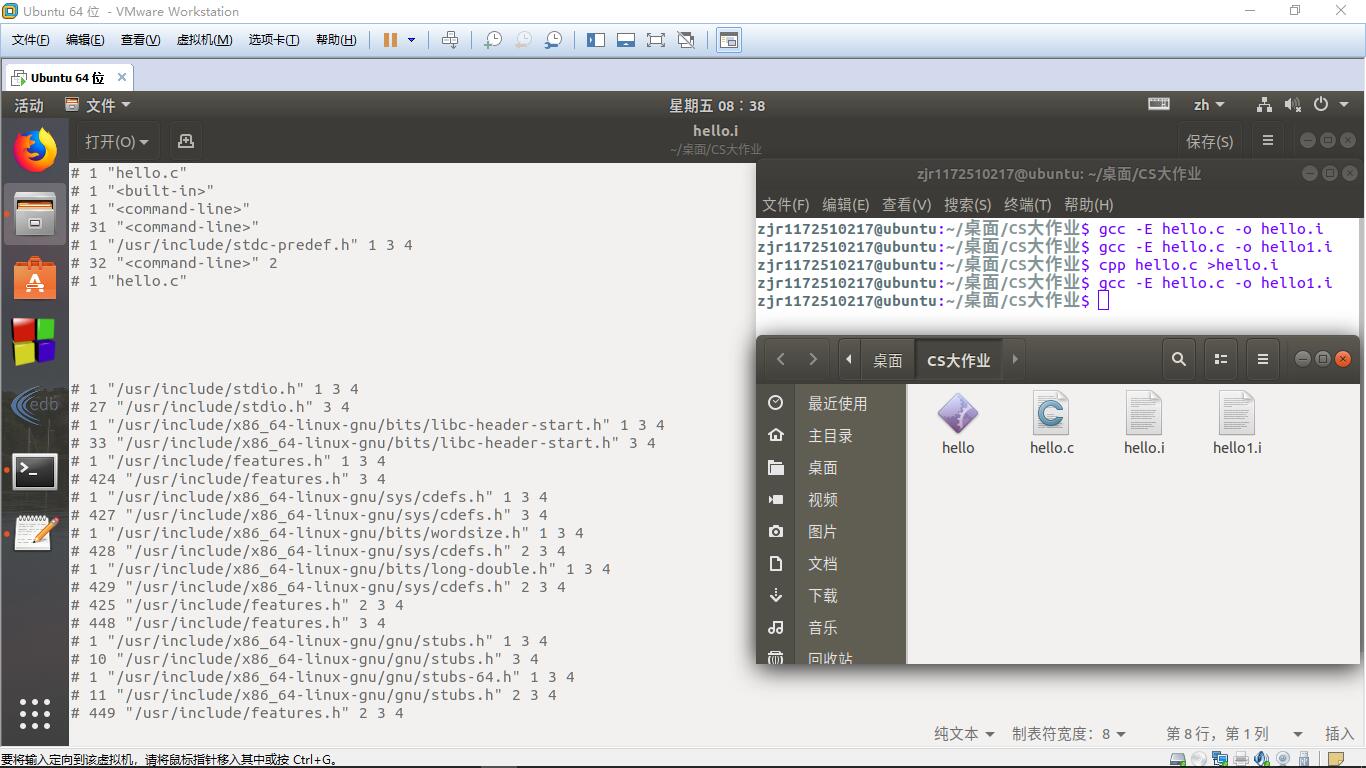
作用主要与三部分有关：宏定义，文件包含，条件编译

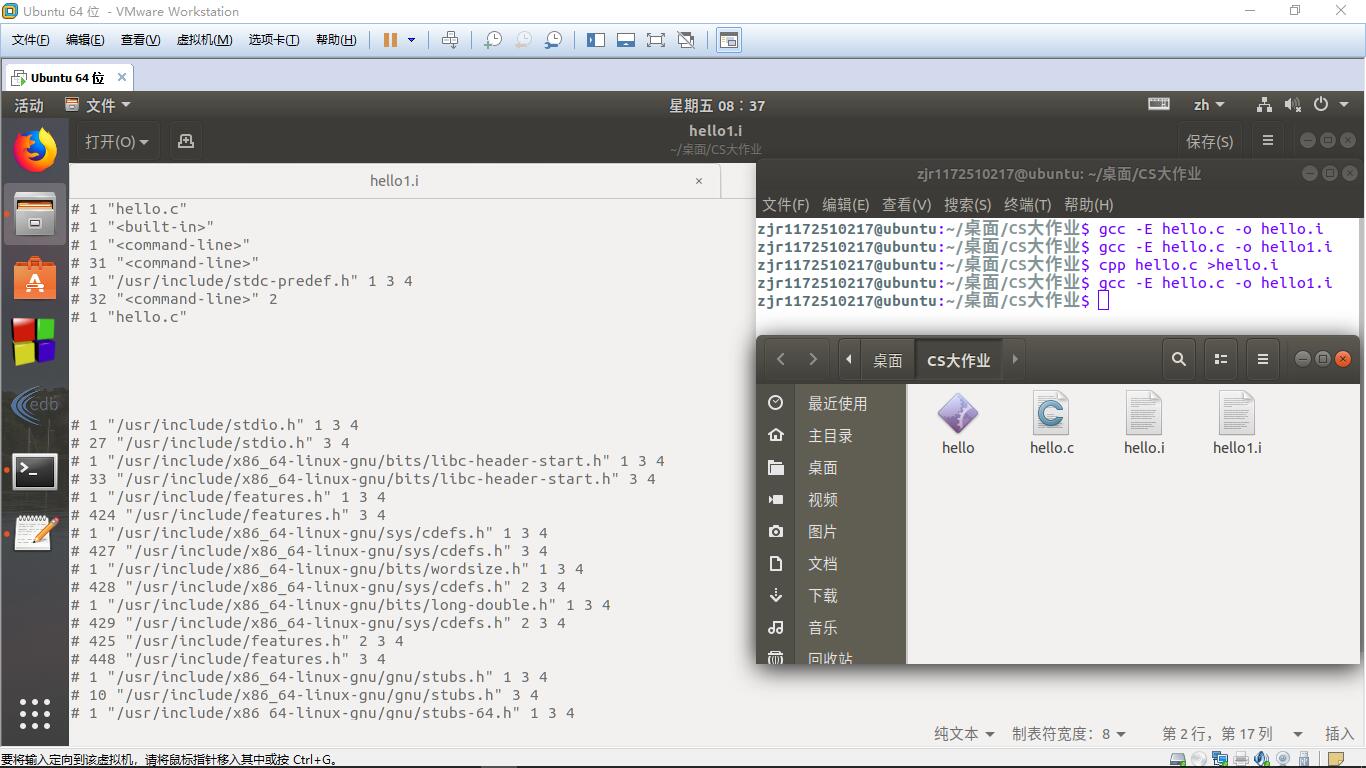
1. 宏定义相关。预处理程序中的#define 标识符文本，预处理工作也叫做宏展开：将宏名替换为文本（这个文本可以是字符串、可以是代码等）。
2. 文件包含相关。预处理程序中的#include，将头文件的内容插入到该命令所在的位置，从而把头文件和当前源文件连接成一个源文件，这与复制粘贴的效果相同。
3. 条件编译相关。根据#if以及#endif和#ifdef以及#ifndef来判断执行编译的条件。

## 2.2在Ubuntu下预处理的命令

**预处理命令1：cpp hello.c > hello.i**

**预处理命令2：gcc -E hello.c -o hello.i**

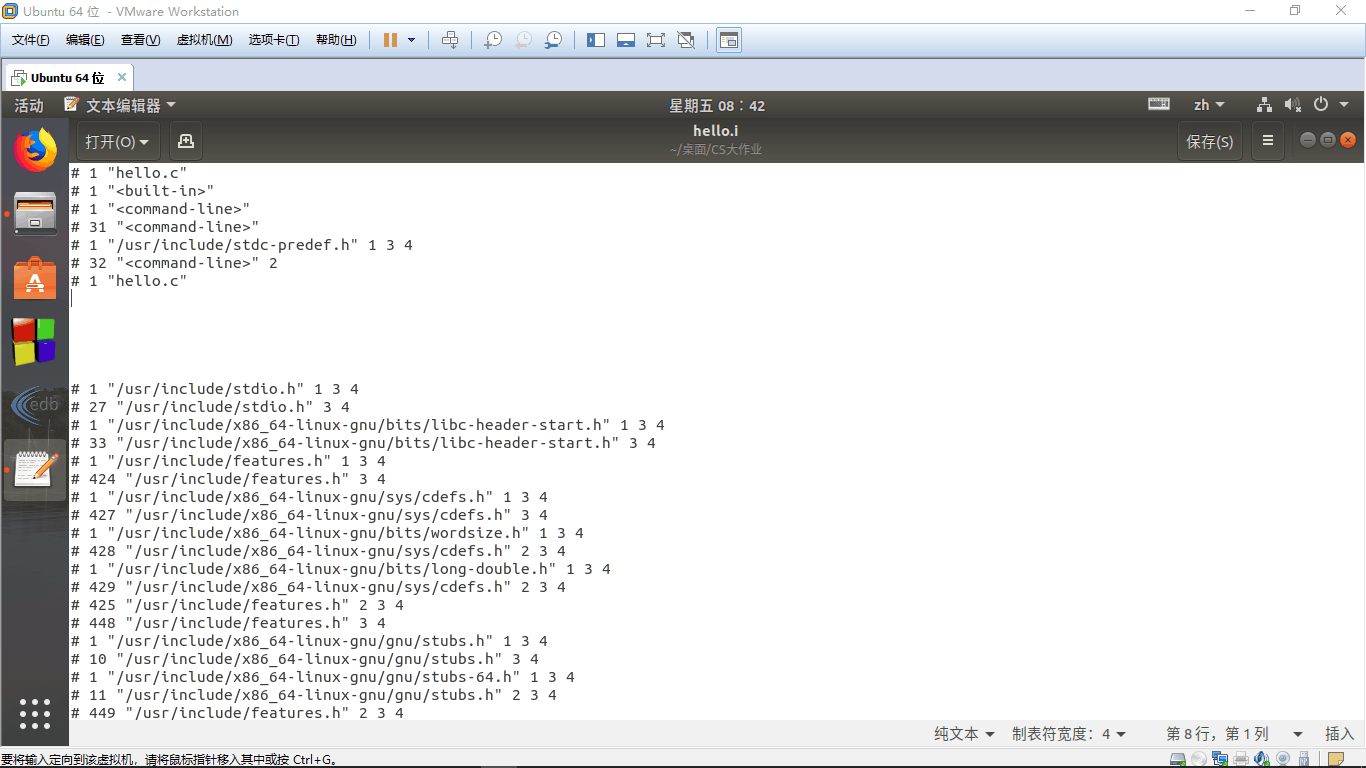


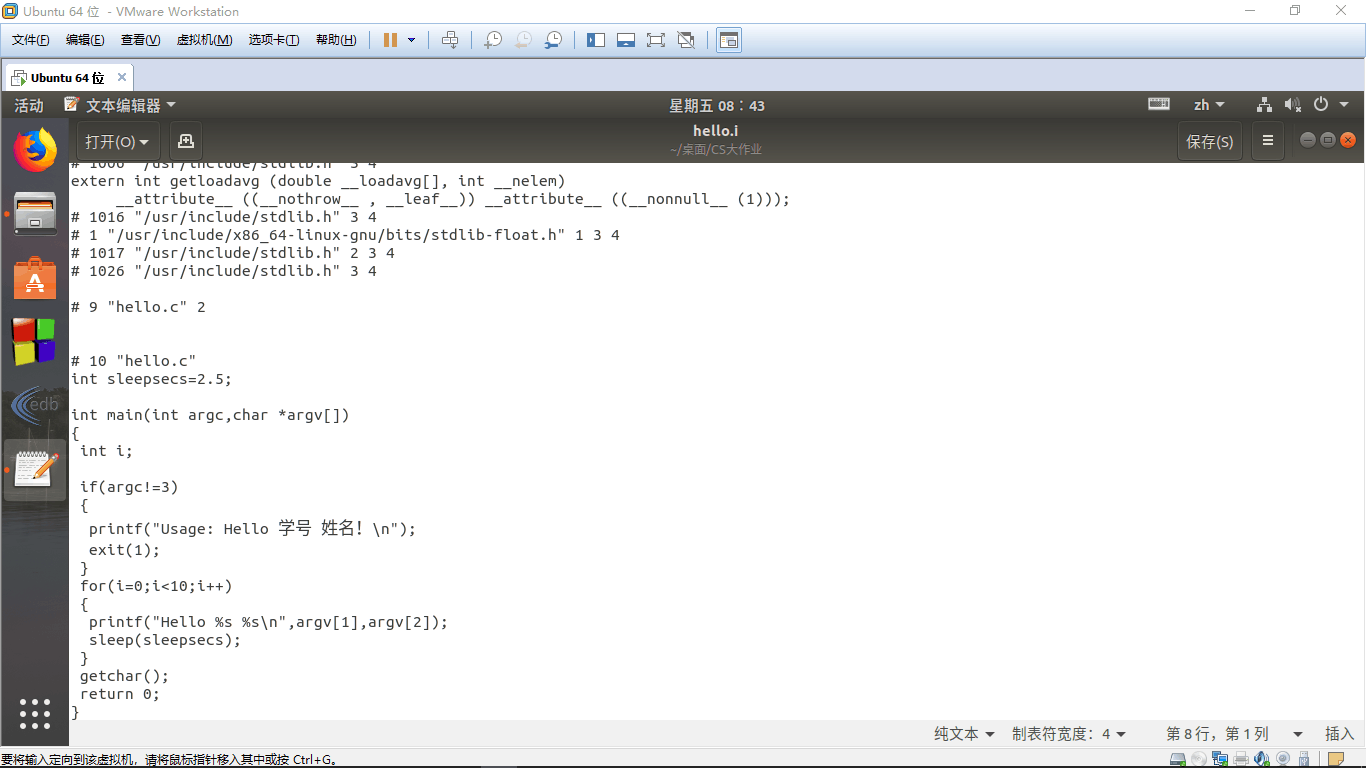
命令1：cpp hell.c >hello.i输出预编译文本文件

命令2：gcc -E hello.c -o hello.i输出预编译文本文件

## 2.3 Hello的预处理结果解析

首先附上hello.i文本文件的部分截图，如下。





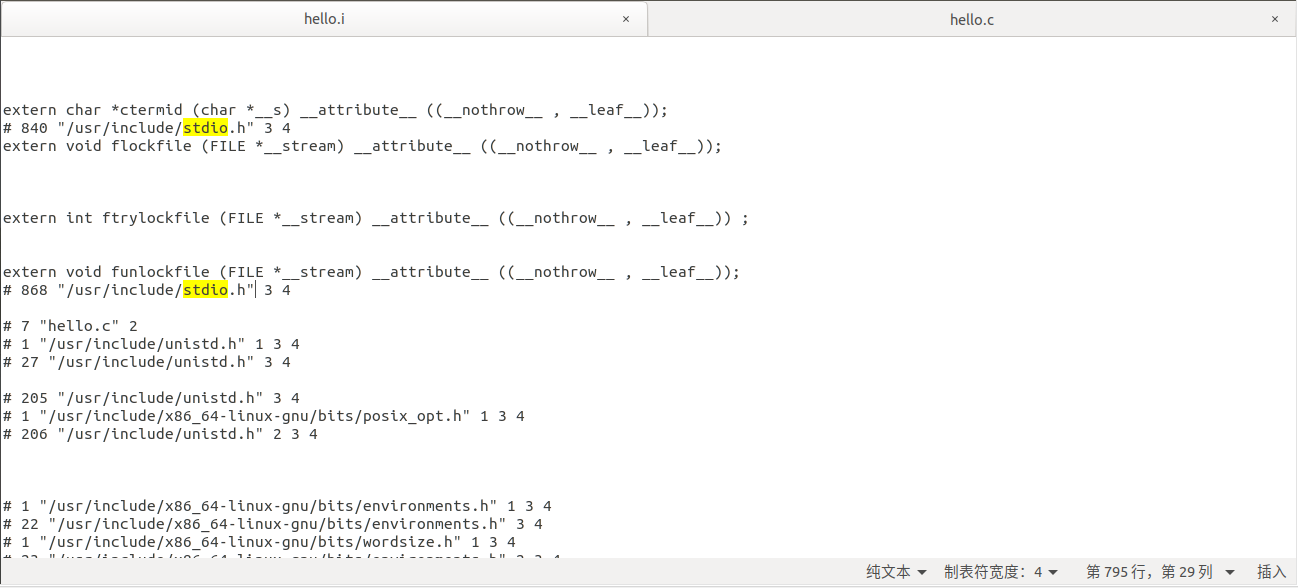
通过hello.i文本文件，我们可以看到原本的28行hello.c文件经过预处理环节，变成了3118行的ASCII码中间文本文件。这是因为预处理器完成预处理工作，实现头文件的展开，宏替换和去注释并作条件编译。

我们发现hello.i文件的篇幅巨大是因为预处理工作进行了头文件的展开。我们通过在文本文件中查找具体操作发现，预处理对头文件stdio的展开开始于13行，结束于795行（如截图2.3.1）；类似的发现头文件unistd的展开开始于798行，结束于2027行（如截图2.3.2）；头文件stdlib的展开开始于2036行，结束于3094行（如截图2.3.3）。

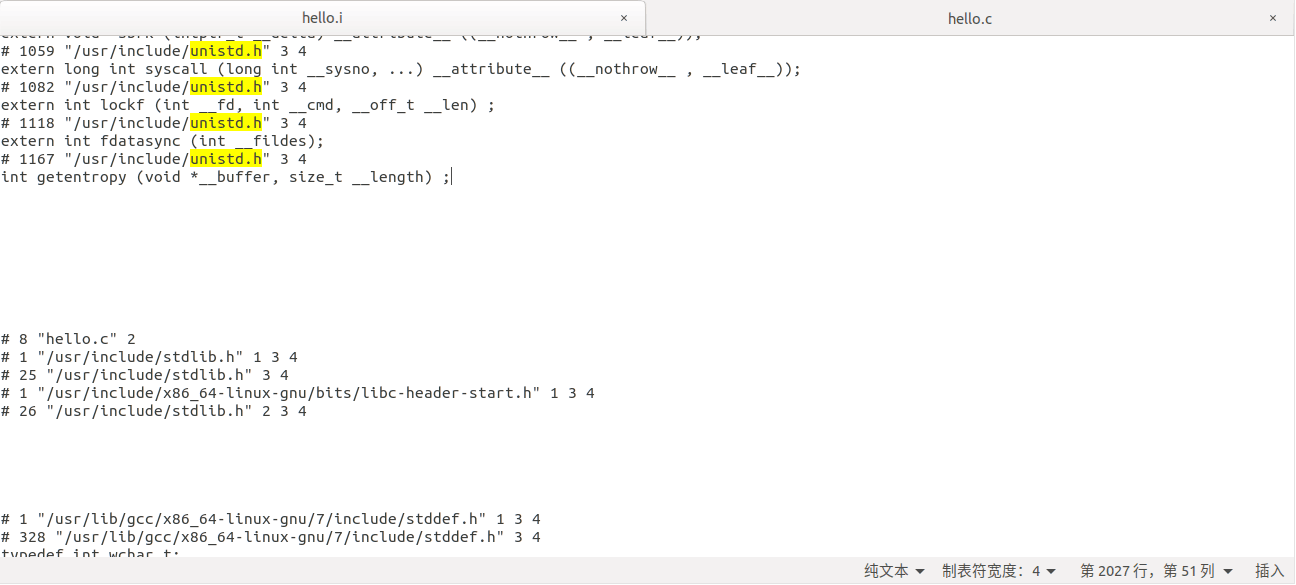
关于文件包含（头文件）在预处理阶段的处理方法。当使用尖括号括起来头文件时，表示在系统头文件目录中查找（由用户在设置编程环境时设置），而不在源文件目录中查找；当使用双引号括起来头文件，则表示首先在当前的源文件目录中查找，找不到再到系统头文件目录中查找。比如以我们的hello.c的头文件处理为例，cpp到默认的环境变量下寻找相应的头文件。具体到三个头文件，寻找路径分别是：usr/inlcude/stdio.h，usr/include/unistd.h和usr/inlcude/stdlib.h（如截图2.2.4，标红加注）

同时我们发现，我们的头文件里有大量的宏定义和条件编译语句存在（截图2.2.5），预处理阶段同样需要对这些语句进行相应的宏替换和条件编译处理。

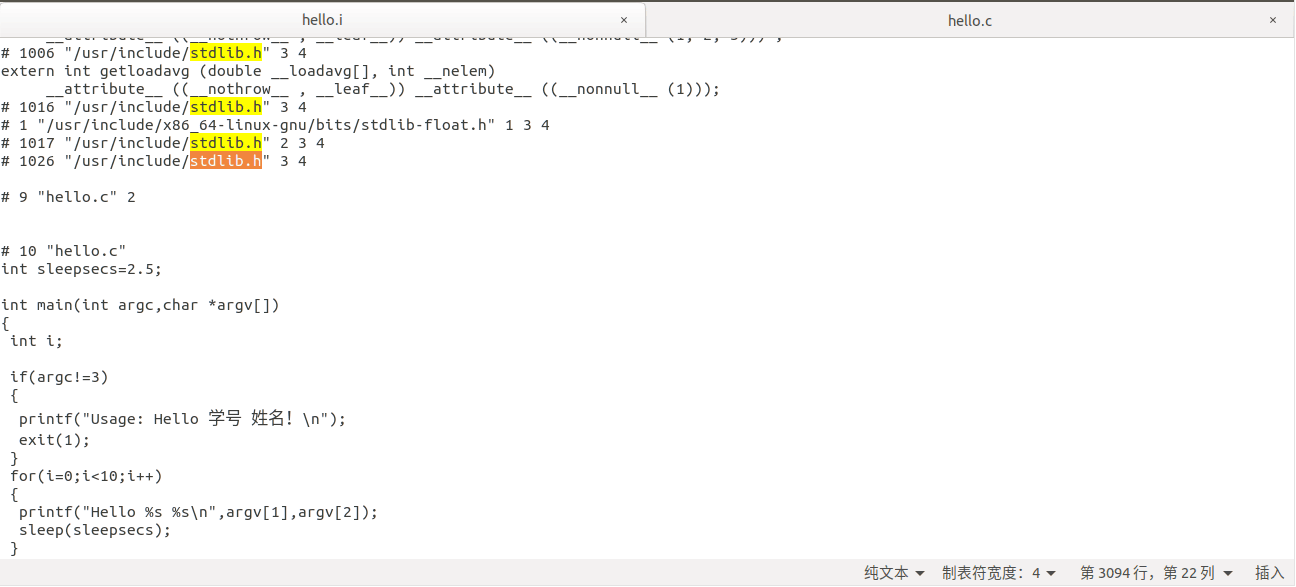
而我们原来的程序主体段开始于第3099行，结束于3118行，预处理删除了我们的注释信息（如截图2.2.5）。除了注释部分以及头文件部分，预编译文件与源文件无太大差别。



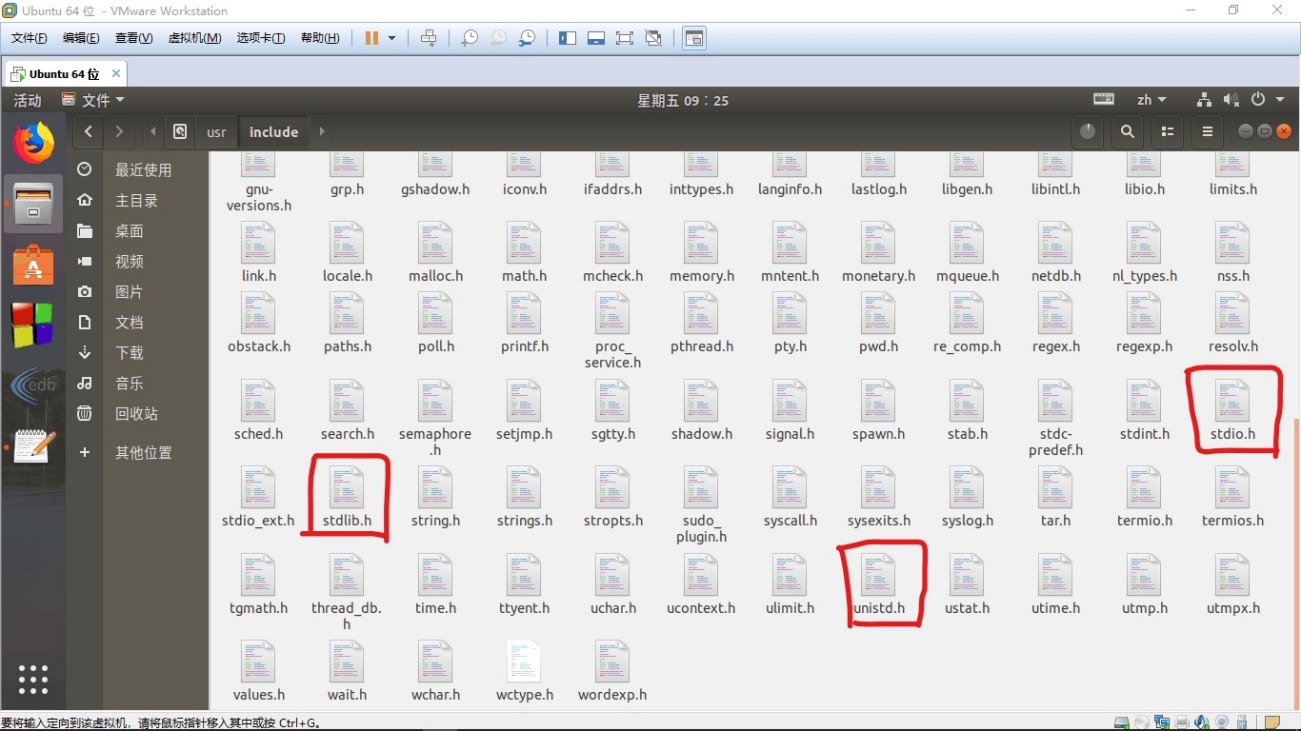
截图2.3.1，stdio.h结尾处



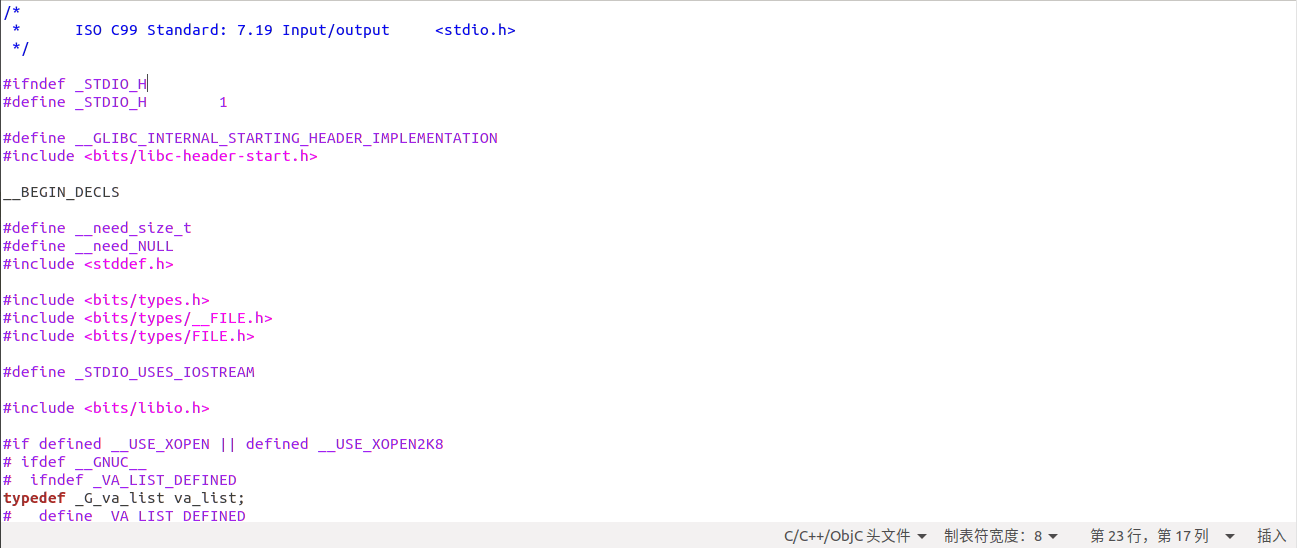
截图2.3.2，unistd.h结尾处



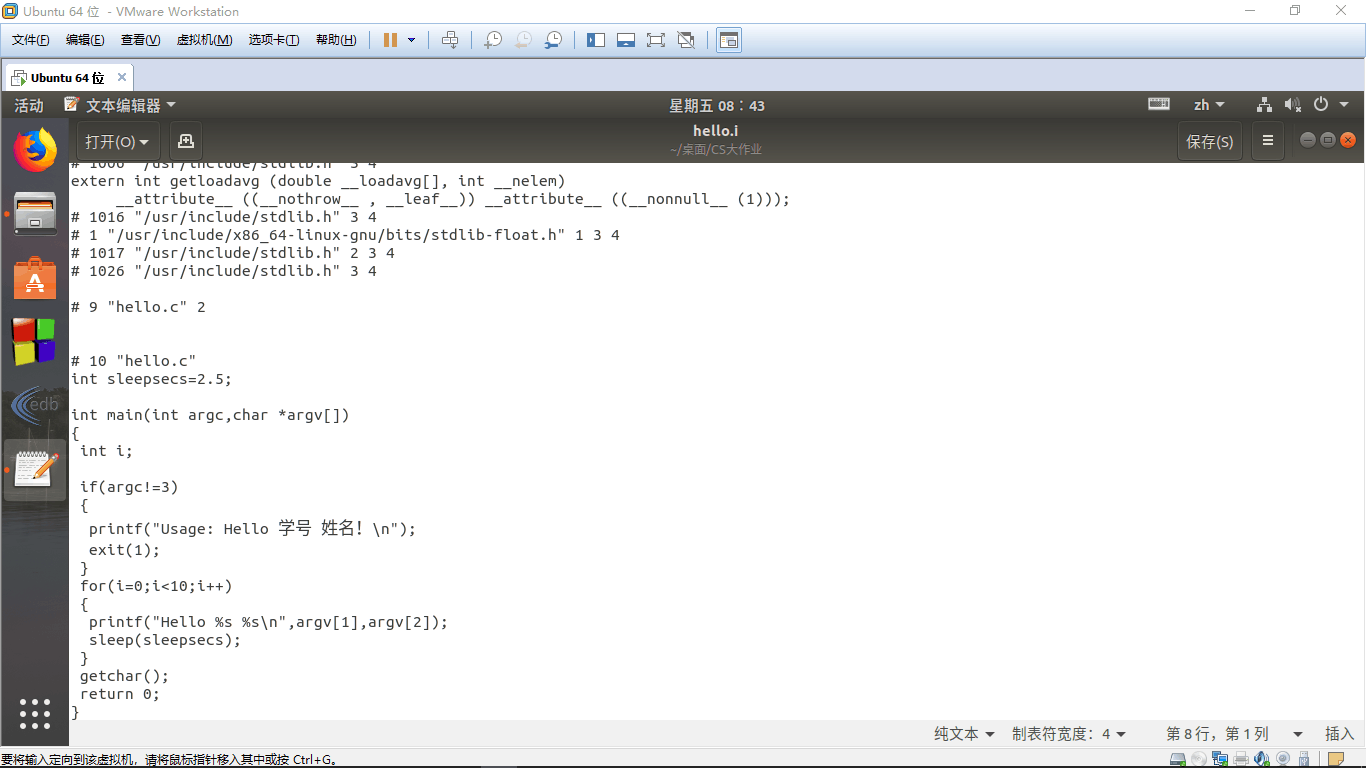
截图2.3.3，stdlib.h结尾处



截图2.2.4，头文件寻找路径



截图2.2.5，头文件里的宏定义和条件编译



截图2.2.6，原程序主体部分

## 2.4 本章小结

本章主要介绍了预处理的概念和应用功能，以及Ubuntu下预处理的两个指令，同时具体到我们的hello.c文件的预处理结果hello.i文本文件解析，详细了解了预处理的内涵。我们发现预处理主要由预处理器完成，这一阶段一共完成4件事：**头文件的展开；宏替换；去掉注释；条件编译**

# 第3章 编译

## 3.1 编译的概念与作用

编译的概念

编译就是将源语言经过词法分析、语法分析、语义分析以及经过一系列优化后生成汇编代码的过程。具体到我们实验，就是将预处理得到的ASCII码的中间文件hello.i翻译成ASCII汇编语言文件hello.s的过程。

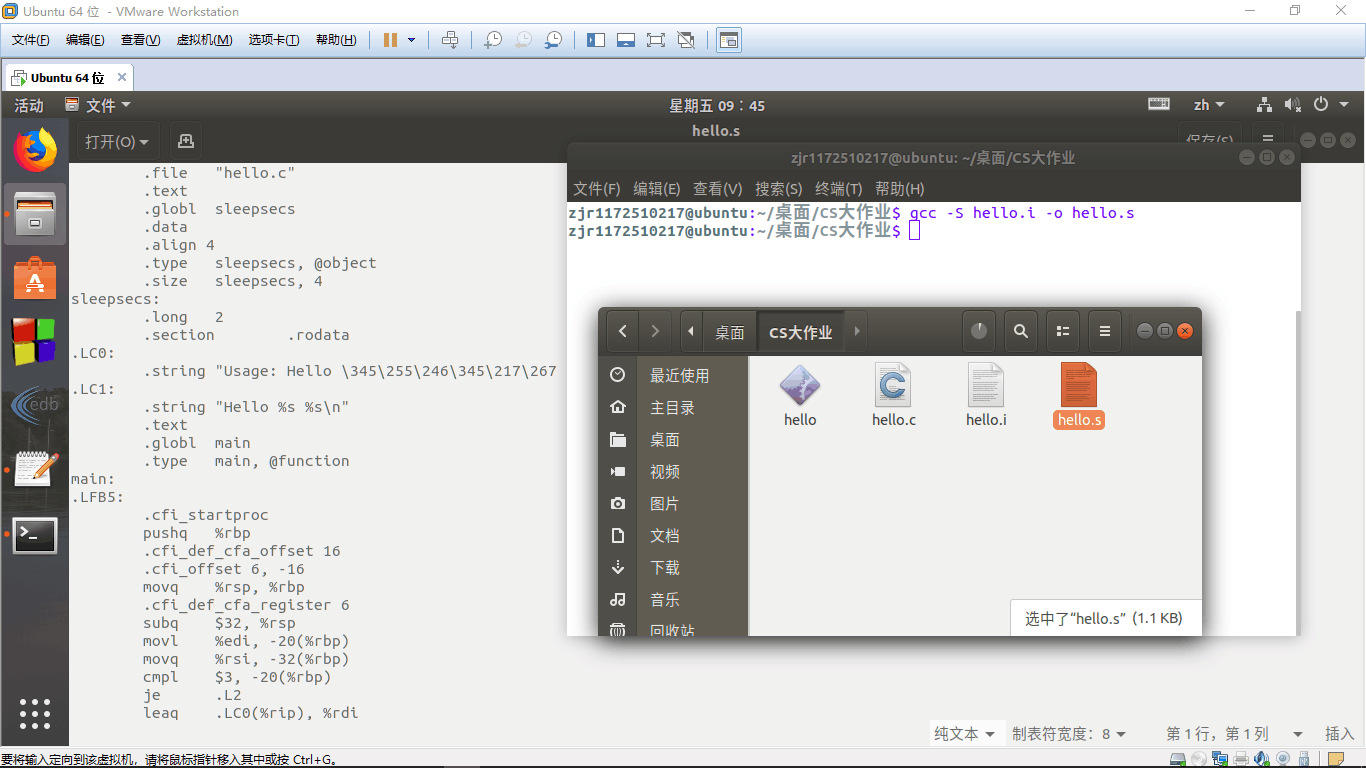
编译的作用

编译的目的是将高级语言程序转化为机器可直接识别处理执行的的机器码的中间步骤。它包括以下几个部分。

1. 词法分析。对输入的字符串进行分析和分割，形成所使用的源程序语言所允许的记号（token），同时标注不规范记号，产生错误提示信息。
2. 语法分析。分析词法分析得到的记号序列，并按一定规则识别并生成中间表示形式，以及符号表。同时将不符合语法规则的记号识别出其位置并产生错误提示语句。
3. 语义分析。即静态语法检查，分析语法分析过程中产生的中间表示形式和符号表，以检查源程序的语义是否与源语言的静态语义属性相符合。
4. 代码优化。将中间表示形式进行分析并转换为功能等价但是运行时间更短或占用资源更少的等价中间代码。

## 3.2 在Ubuntu下编译的命令

编译的命令：gcc -S hello.i -o hello.s



截图3.2，.i编译生成.s文件

## 3.3 Hello的编译结果解析

1. hello.s文件分析

|  |  |
| --- | --- |
| 内容 | 含义 |
| .filet | 声明源文件 |
| .text | 声明以下是代码段 |
| .globl | 声明一个全局变量 |
| .secetion .rodata | 声明以下是rodata节 |
| .align | 声明对指令或者数据的存放地址进行对齐的 方式 |
| .long | 声明一个long类型 |
| .string | 声明一个string类型 |
| .size | 声明大小 |
| .type | 声明是函数类型还是对象类型 |

1. 数据类型之整数

hello.s中用到的数据全局变量（int sleepsecs），局部变量（如int i），表达式（如i++）。

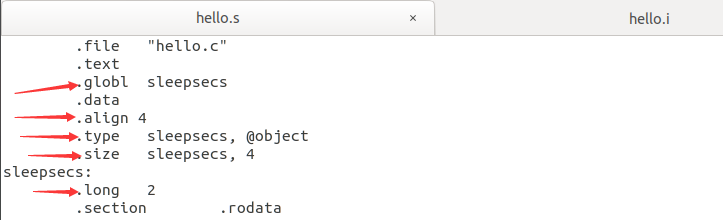
数据类型有整型，数组，字符串。

我们首先看整数。

程序中的整数有int sleepsecs，int i，int argc，常数立即数（如3，0，10等）

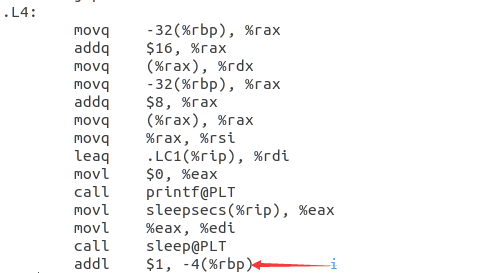
1）对于int sleepsecs。sleepsecs是已经被初始化的全局变量（初始化语句为int sleepsecs=2.5;）由于.data 节存放已经初始化的全局和静态C变量，编译器首先将sleepsecs在.text代码段中声明为全局变量；其次在.data 段中，设置对齐方式（.align）为4字节对齐，设置类型（.type）为对象，设置大小（.size）为4字节，设置为long类型（.long），其值为2（如截图3.3.2-1）。

有趣的是，在这个过程中，我们发现全局变量sleepsecs被转化为long类型。这个问题是因为自动转换是内置规则，即隐式转换，而且int转化为long不会丢失数据。这与编译器缺省有关。



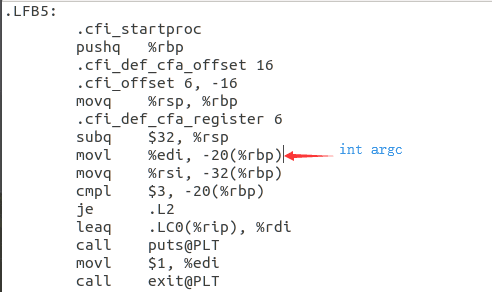
截图3.2.2-1，int sleepsecs分析

2）对于int i。首先局部C变量在运行时被保存在栈或者是寄存器里。具体到局部变量int i，在hello.s文件中，编译器将i存储在栈上空间-4(%rbp)中（如截图3.3.2-2），而且可以看到的信息是i在栈上占据了4个字节的空间。



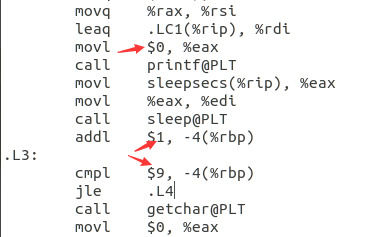
截图3.3.2-2，int i分析

3）对于int argc。argc是我们main函数的第一个形式参数。观察分析 hello.s文件，我们可以看到这样的结构（如截图3.3.2-3）（%edi是第一个参数），我们可以看到我们将int argc赋值给了-20(%rbp)，因此第一个形式参数页储存在栈上，且所在栈空间位置为-20(%rbp)。



截图3.3.2-3.，int argc分析

4）对于常数立即数。对于源程序中出现的常数如0，1，2，10等是直接在汇编代码中存在的（如截图3.3.2-4），因为汇编代码是允许立即数以$常数形式存在的。



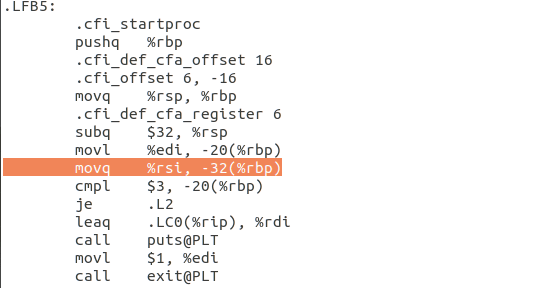
截图3.3.2-4，立即数分析

3.3.3 数据类型之数组

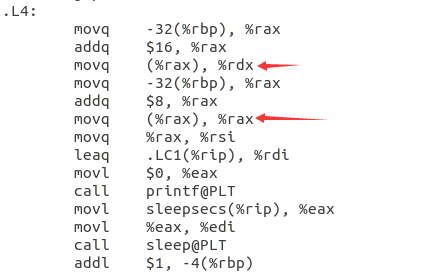
对于数组char \*argv[]。它是我们main函数的第二个形式参数，来源于我们终端键入的数据（我键入的是：./hello 1172510217 张景润）。argv同时作为存放char指针的数组。

argv数组中一个元素大小为8个字节（判断来源于截图3.3.3-1和截图3.3.3-2）.而在main函数内部，对argv[1]，argv[2]的访问来源于对数组首地址argv进行加法计算得到相应的地址。

我们可以看到在hello.s中，运用了2次movq %rax, %rig，目的是取出内容，即我们终端输入的命令参数。



截图3.3.3，int argc[]数组传入函数



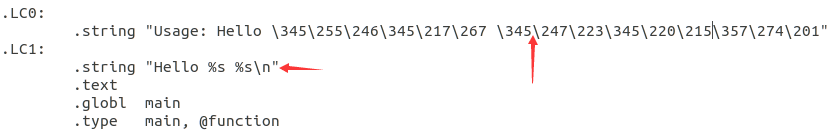
截图3.3.3，int argc[]数组内容使用

3.3.4 数据类型之字符串

很明显，我们可以看到我们的字符串有，"Usage: Hello 学号 姓名！\n"，以及我们终端键入的储存在argc[]为地址的数组中，和“Hello %s %s\n”（如截图3.3.4-1）。

而同时我们可以在hello.s中看到字符串\345\255\246\345\217\267 \345\247\223\345\220\215\357\274\201，这些其实是学号 姓名的UTF-8格式，而由于一个汉字在该编码中占据3个字节，因此与我们的想法是一致的（其中\是汉字分隔符）。

而对于字符串“Hello %s %s\n”，这是第二个printf传入的输出格式化参数。而且值得注意的是，这些printf格式输出控制串是在.rodata声明的。



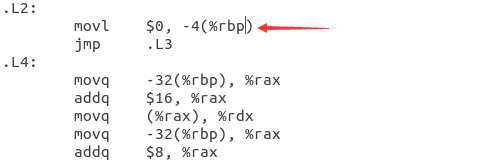
截图3.3.4-1

3.3.5 赋值操作

源程序中的赋值操作有int sleepsecs=2.5; i=0;i++;

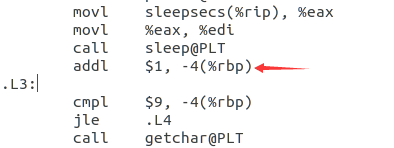
1）对于第一个赋值操作int sleepsecs = 2.5。前面大致已经对此进行分析，sleepsecs是全局变量，而且在这里进行了赋初值操作，因此直接在.data节中将sleepsecs 声明为值为2的long类型数据（隐式转换，编译器缺省）。

2）对于第二个赋值操作i=0。在hello.s文件中通过汇编语句movl $0, -4(%rbp)将立即数赋值给我们的局部变量int i。而且值得我们注意的是汇编语句用的是movl，这是因为我们的局部变量是int型，4个字节，因此使用字母l。（截图3.3.5-1）



截图3.3.5-1

3）对于第三个赋值操作i++。在hello.s文件中是通过语句addl $1, -4(%rbp)实现的，因为-4(%rbp)继承自原来的i=0，因此通过addl（有意思的是，这里仍然要使用addl，因为是int操作）达到每次循环+1的目的（如截图3.3.5-2）。



截图3.3.5-2

3.3.6 类型转换

源程序中用到的类型转换有int sleepsecs=2.5;（隐式类型转换，将浮点数2.5转化为int整数2）

值得注意的是，浮点数常数默认的均是double类型，因此此处隐式转换类型为由double型转化为int型（编译器缺省转换为long型）。

同样，思考这样的一个问题。2.5被隐式转换之后，为什么变成了long类型的2。当在 double 或 float 向 int 进行类型转换的时候，程序改变数值和位模式的原则 是：值会向零舍入。例如 1.6 将被转换成 1，-1.9 将被转换成-1。而对于我们的转化前的浮点数2.5当然是舍入到整数2。

]OJ77XTP52B11M_NE_%0{[W

截图3.3.6-1，隐式转换操作

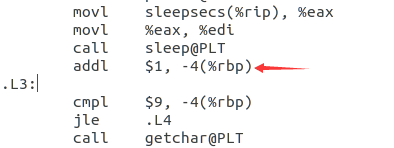
3.3.7 算术操作+逻辑操作

首先，汇编语言中算术操作指令，效果以及描述有以下指令。

|  |  |  |
| --- | --- | --- |
| 指令 | 效果 | 描述 |
| 1eaq S,D | D←&S | 加载有效地址 |
| INC D  DEC D  NEG D  NOT D | D←D十1  D←D-1  D←一D  D←一D | 加1  减l  取负  取补 |
| ADD s,D | D←D十s | 加 |
| SUB S,D | D←D-S | 减 |
| IMUL S,D | D←D\*S | 乘 |
| XOR S,D | D←D^S | 异或 |
| OR S,D | D←D|S | 或 |
| AND S,D | D←D&S | 与 |
| SAL k,D | D←D<<k | 左移 |
| SHL k,D | D←D<<k | 左移 |
| SAR k,D | D←D>>Ak | 算术右移 |
| SHR k,D | D←D<<Lk | 逻辑右移 |

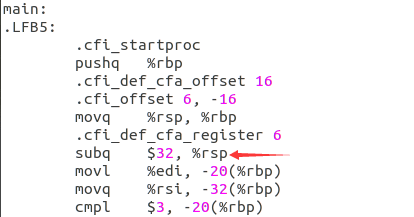
要注意的是，上表的操作顺序与ATT格式的汇编代码中的相反。（同时还有一些特殊的算术操作未被列出，比如有符号和无符号数的全128位乘法与除法）

1）具体到我们的源程序中，算术操作有i++（即i=i+1），这个是通过汇编语句addl $1, -4(%rbp)实现的，因为-4(%rbp)继承自原来的i=0，因此通过addl（有意思的是，这里仍然要使用addl，因为是int操作）达到每次循环+1的目的（如截图3.3.7-1）。



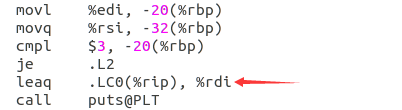
截图3.3.7-1，i++操作

2）还有汇编语句subq $32, %rsp。由于%rsp总是指向栈顶元素，因此这里对栈指针进行减法操作，目的是开辟一断栈空间，而这里开辟的空间是32B（如截图3.3.7-2）。栈顶地址自高处向低处变化。



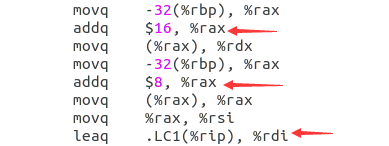
截图3.3.7-2，减法操作

3）算法操作还有leaq .LC0(%rip), %rdi。加载有效地址，计算LC1的段地址：%rip+.LC1，同时将此地址送给%rdi。



截图3.3.7-3，加载有效地址操作

4）同时类似的还有addq $16, %rax，addq $8, %rax和leaq .LC1(%rip), %rdi，他们的功能分别是为了取出argv数组中的指针指向的内容，和计算LC2的段地址：%rip+.LC2，同时将此地址送给%rdi（如截图3.3.8-4，多个算术操作）。



截图3.3.7-4，多个算术操作

3.3.8 关系操作

首先汇总与我们的程序相关的关系操作的汇编指令，及其效果和描述。

|  |  |  |
| --- | --- | --- |
| 指令 | 效果 | 描述 |
| CMP S1,S2 | S2-Sl | 比较 |
| TEST S1, S2 | S1&S2 | 测试 |
| SETX D | —— | —— |
| JX | —— | —— |

值得注意的是：比较和测试指令不修改任何寄存器的值,只是设置条件码。而对于set指令，每条指令根据条件码的某种组合,将一个字节设置为0或者1。当跳转条件满足时,这些指令会跳转到一条带标号的目的地。

现在回到我们具体的函数当中去。

1）首先关系操作是cmpl $3, -20(%rbp);配合跳转语句je .L2。这两句汇编代码对应我们源程序中的语句如下

1. **if**(argc!=3)
2. {
3. printf("Usage: Hello 学号 姓名！\n");
4. exit(1);
5. }

有趣的是，我们发现我们编写代码的逻辑与编译器处理的逻辑是有细微的差别，我们的逻辑是如果argc!=3，则执行提示输出并退出语句；对于汇编代码，是如果==3（je .L2）则跳转执行相关语句。我们的编译器将!=3时执行，优化为==3，跳转。

对于具体操作，cmpl $3, -20(%rbp)语句计算20(%rbp)-3，并设置条件码，随之je利用这些条件码，进行相应的跳转处理。

2）接着是cmpl $9, -4(%rbp)，随之执行jle .L4。这两句汇编语句对应我们源程序代码语句如下

1. **for**(i=0;i<10;i++)
2. {
3. printf("Hello %s %s\n",argv[1],argv[2]);
4. sleep(sleepsecs);
5. }

同样有意思的是，我们编写的源程序代码的逻辑与编译器处理的逻辑是有细微的差别。我们的逻辑是判断i<10，则执行，而我们的编译器将其优化为i<=9，则执行。

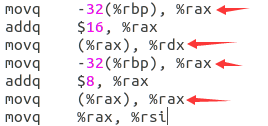
对于具体操作，cmpl $9, -4(%rbp)计算-4(%rbp)-9，并设置条件码，随之jle语句利用这些条件码，进行相应的跳转处理。

3.3.9 数组/指针/结构操作

大致说明：对于汇编语句，有关的操作大多数是通过数据传送mov指令实现的。

1）首先是语句movl %edi, -20(%rbp)和movq %rsi, -32(%rbp)。分别是将寄存器%edi的内容赋值给-20(%rbp)指针指向的地址，将寄存器%rsi的内容赋值给-32(%rbp)指针指向的内容。这2句汇编语句对应源程序中main函数形参的传入部分。

2）对于源程序中的输出argv[1]和argv[2]部分。在编译器的处理下变成了截图所示部分。箭头标注的部分代表取出指针所指的内存中的内容。



截图3.3.9-1，取出指针指向的内容

3.3.10 控制转移

控制转移部分在3.3.8部分已有所介绍。常常是配合指令CMP和TEST存在的。

1）cmpl $3, -20(%rbp)配合je .L2，对应源程序中C语句（如下）。

1. **if**(argc!=3)
2. {
3. printf("Usage: Hello 学号 姓名！\n");
4. exit(1);
5. }

具体分析。

cmpl $3, -20(%rbp)语句计算20(%rbp)-3，并设置条件码。随之je利用这些条件码，发现等于0的话，则跳转到.L2段（如截图标注）；若不等于0，则继续向下执行，调用puts函数输出命令行要求，并调用exit函数退出。



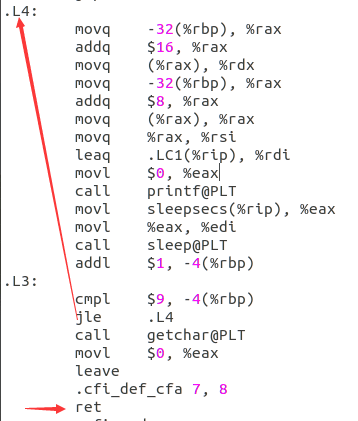
截图3.3.10-1，比较字符串数和3的大小

2）接着是cmpl $9, -4(%rbp)，随之执行jle .L4。这两句汇编语句对应我们源程序代码语句如下

1. **for**(i=0;i<10;i++)
2. {
3. printf("Hello %s %s\n",argv[1],argv[2]);
4. sleep(sleepsecs);
5. }

具体分析。

cmpl $9, -4(%rbp)计算-4(%rbp)-9，并设置条件码。随之jle语句利用这些条件码，若小于等于0，则跳转到.L4段（如截图标注）；而若大于0，则继续向下执行，结束程序。即循环执行10次.L4段，然后退出，表现在终端上，就是10次输出字符串。



截图3.3.10-2，比较循环变量i与10的大小

3.3.11 函数操作

总的来说，函数时过程的一种形式。而过程是软件中一种很重要的抽象。它提供了一种封装代码的方式，用一组指定的参数和一个可选的返回值实现了某种功能。然后,可以在程序中不同的地方调用这个函数。

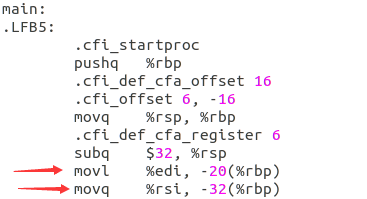
当调用一个过程时,除了要把控制传递给它并在过程返回时再传递回来之外,过程调用还可能包括把数据作为参数传递,而从过程返回还有可能包括返回一个值。

源代码中的函数有main函数，printf函数（第一处被优化为puts函数），sleep函数，getchar函数和exit函数。

1）参数传递（地址/值）。

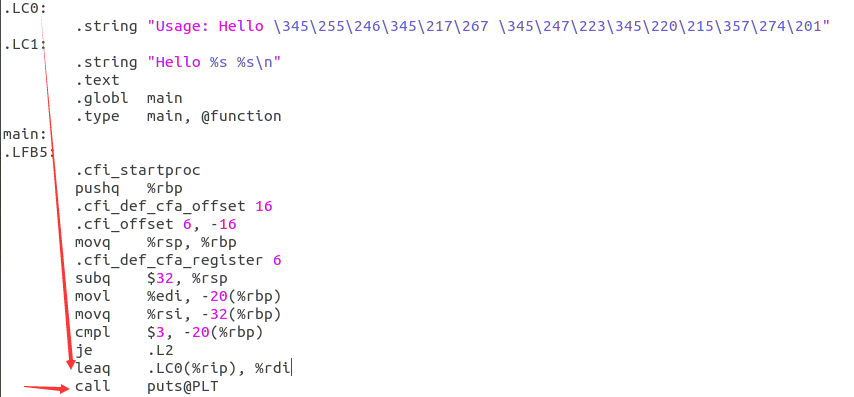
即过程中的传递数据，P必须能够向Q提供一个或多个参数。

对于main函数。函数形参有2个，在汇编代码中分别是用这两条语句达到传送参数的功能的（如截图）。即函数原来将我们要传入的参数储存在%edi和%rsi中，然后在栈上保存。更具体一步，传入的两个参数分别是值和地址。



截图，main函数传递形式参数

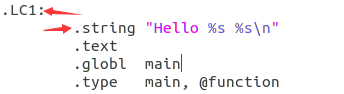
对于printf函数。printf函数在具体的汇编代码中被优化为puts函数。我们发现第一次在汇编代码中（截图1）首先将rdi赋值为字符串“Usage: Hello 学号 姓名！ \n”字符串的首地址（leaq .LC0(%rip), %rdi）。然后调用了puts函数，即将第一处字符串参数传入。对于第二处，类似的，我们发现在汇编代码中（截图2和截图3）首先将rdi赋值为字符串"Hello %s %s\n"的首地址。这里没有被优化为puts函数，而是直接调用printf函数。同时设置%rsi argv[1]，%rdx 为argv[2]。这样就可以根据控制字符串，直接输出终端键入的命令行。



截图1，第一次（可能的）调用printf函数

VG{)%MO~(M]PCI7KJUE7$19

截图2



截图3

对于sleep函数。根据下面的截图我们可以发现，传入参数的过程为movl sleepsecs(%rip), %eax和movl %eax, %edi，对应原来函数的形式参数为全局变量sleepsecs。

3~DTLEGYU9G{IQ523I)[[}T

截图

对于getchar函数。如截图。

L`@LE({)K0BYF03VY~TY_@9

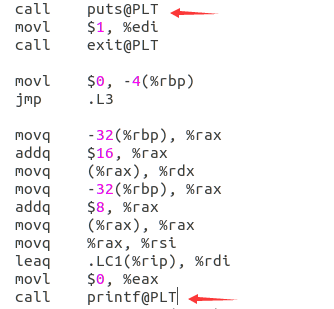
对于exit函数（如截图）。通过汇编语句movl $1, %edi将%edi寄存器内容设置为1。

E00HHA9%QQ{}UB9$LHI1O$C

2）函数调用

对于main函数。main函数被调用即call才能执行（被系统启动函数 \_\_libc\_start\_main调用）。对于call指令，它将下一条指令的地址dest压栈， 然后跳转到main 函数，即完成对main函数的调用。

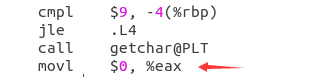
对于printf函数。在main函数内部，通过汇编语句call puts@PLT调用（第一次），通过汇编语句call printf@PLT调用（第二次）。附上截图



对于sleep函数。在main函数内部，被多次调用（在for循环内部），调用了10次，通过汇编语句call sleep@PLT除法此调用。（附上截图）

_}`KOHNC7$0{QCR`3$3TJ$L

对于getchar函数。在main函数内部，最后被调用，调用它的汇编语句是：call getchar@PLT（附上截图）。

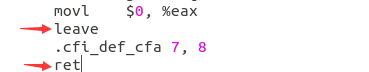


对于exit函数。在main函数内部被调用，调用它的汇编语句是call exit@PLT（附上截图）。

O7R_CCHB)L{U}IE7DX7RMI6

3）函数返回

对于main函数。程序结束时，调用leave指令（leave相当于mov %rbp,%rsp和pop %rbp），恢复栈空间为调用之前的状态，然后 ret 返回（ret 相当 pop IP，将下一条要执行指令的地址设置为dest）。附上截图



## 3.4 本章小结

本章主要详细介绍了编译的概念与作用，以及在Ubuntu下编译的指令，最后我们具体到对hello.c源文件的编译文件hello.s进行数据类型（主要包括整数，字符串，数组）和操作（赋值操作，类型转换，算术和位级操作，关系操作，指针数组结构操作以及控制转移和函数操作）的细致分析和探讨。

# 第4章 汇编

## 4.1 汇编的概念与作用

**概念**

驱动程序运行汇编器as，将汇编语言（这里是hello.s）翻译成机器语言（hello.o）的过程称为汇编，同时这个机器语言文件也是可重定位目标文件。

**作用**

汇编就是将高级语言转化为机器可直接识别执行的代码文件的过程，汇编器将.s 汇编程序翻译成机器语言指令，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，.o 文件是一个二进制文件，它 包含程序的指令编码。

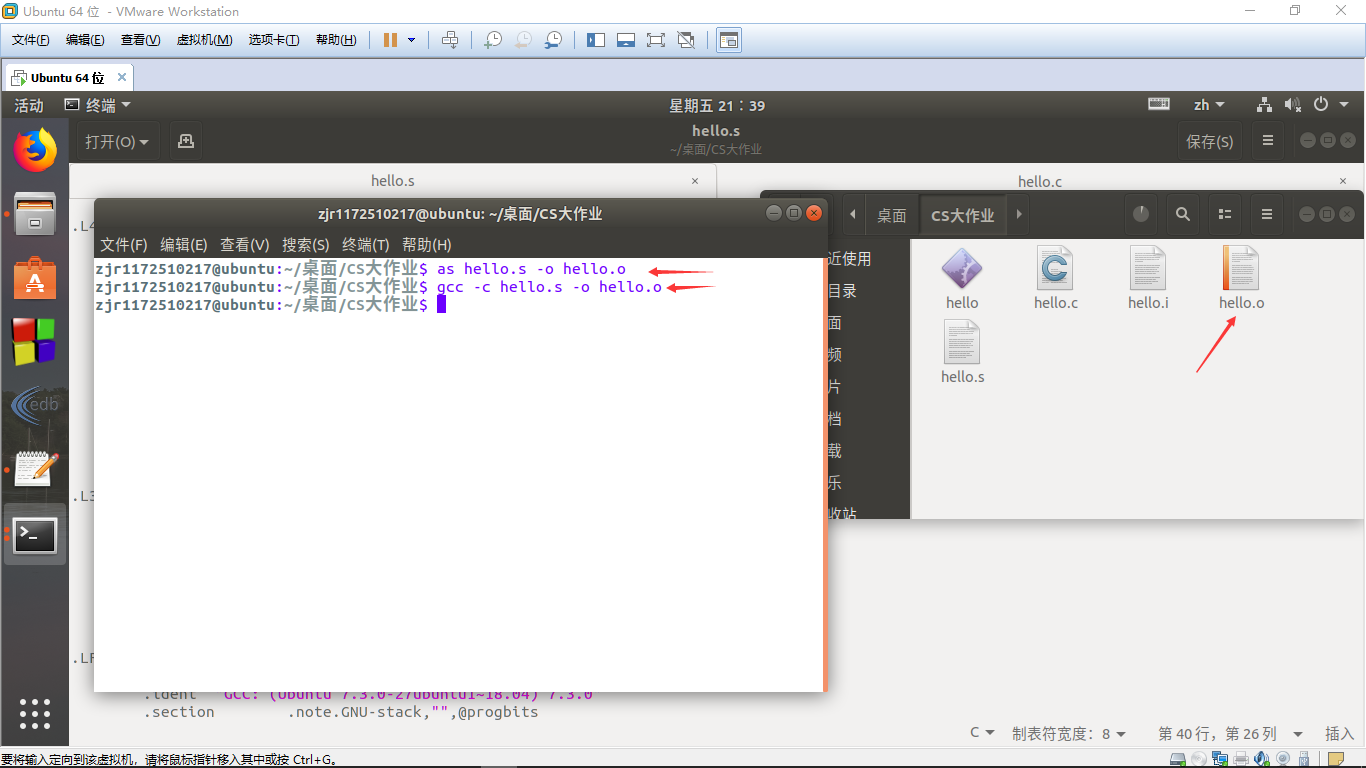
## 4.2 在Ubuntu下汇编的命令

汇编的命令

as hello.s -o hello.o或者

gcc -o hello.s -o hello.o

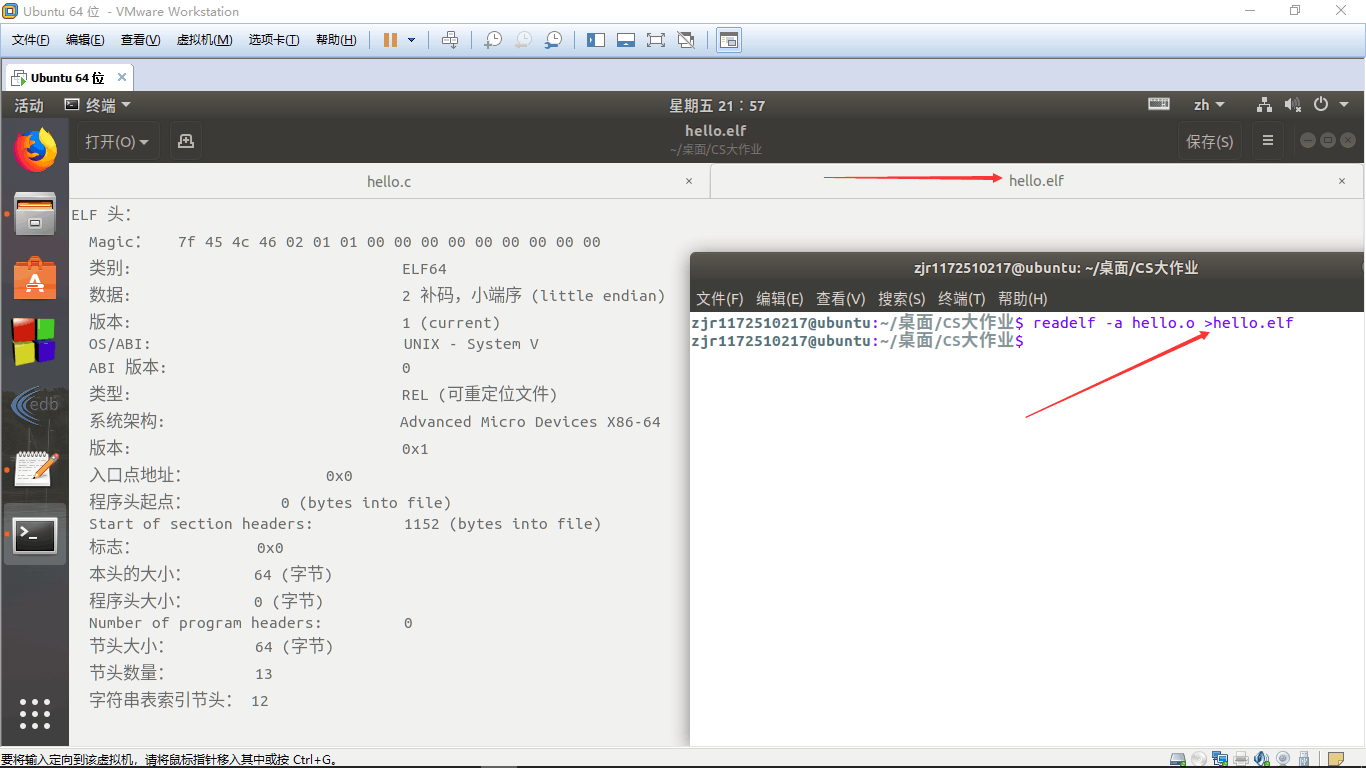
附上截图



## 4.3 可重定位目标elf格式

1）读取可重定位目标文件。

键入命令行readelf -a hello.o >hello.elf将elf可重定位目标文件输出定向到文本文件hello.elf中，附上截图。



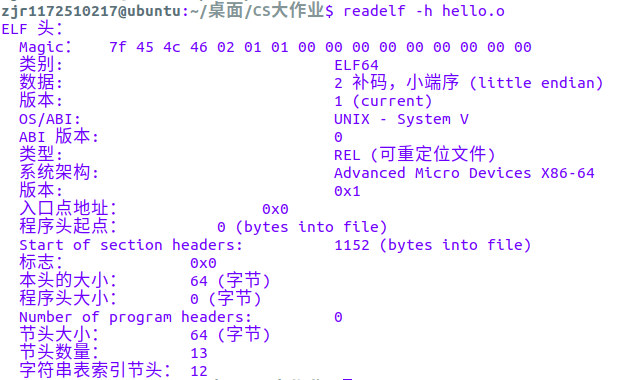
readelf指令以及生成的elf文件

2）作出典型的ELF可重定位目标文件的表格。

|  |  |
| --- | --- |
| ELF头 | 节 |
| .text |
| .rodata |
| .data |
| .bss |
| .symtab |
| .rel.text |
| .rel.data |
| .debug |
| .line |
| .strtab |
| 节头部表 | 描述目标  文件的节 |

3）列出ELF文件的节的内容

ELF 头：以 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、机器类型（如x86-64）、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。不同节的位置和大小是由节头部表描述的，其中目标文件中每个节都有一个固定大小的条目。（附上截图）



ELF头

.text：已编译程序的机器代码。

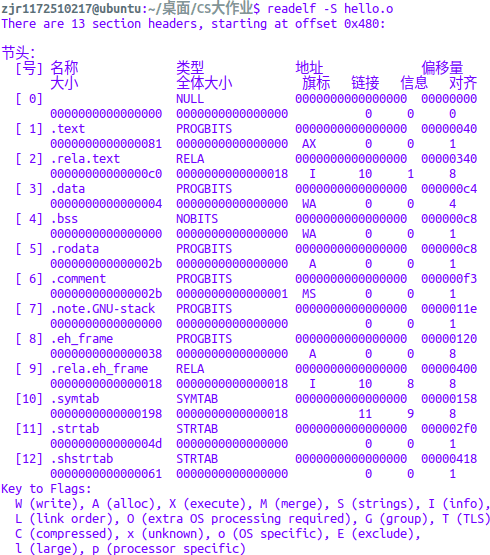
.rodata：只读数据，比如printf语句中的格式串和开关语句的跳转表。

.data：已初始化的全局和静态C变量。局部C变量在运行时被保存在栈中，既不出现在.data节中，也不出现在.bss中。

.bss：未初始化的全局和静态C变量，以及所有被初始化为0的全局或静态C变量。

.symtab：一个符号表，他存放在程序中定义和引用的函数和全局变量的信息

节头部表：节头表包括节名称，节的类型，节的属性（读写权限），节在ELF文件中所占的长度以及节的对齐方式和偏移量。我们可以使用终端指令readelf -S hello.o来查看节头表。附上截图



节头表

.rela重定位节。该节包括的内容是：偏移量，信息，类型，符号值，符名称和加数。附上截图。我们可以看到截图中有8条重定位信息，分别对应对.L0（第一个 printf 中的字符串），puts 函数，exit 函数，.L1（第二个 printf 中的字符串）、printf 函数、 sleepsecs、sleep 函数、getchar 函数。

（关于ELF64位.rel.text节在ORACLE官网上找到了相关内容）关于.rela的结构声明如下。

typedef struct {

Elf64\_Addr r\_offset;

Elf64\_Xword r\_info;

Elf64\_Sxword r\_addend;

} Elf64\_Rela;

现在就这三部分内容进行说明。

**a）r\_offset**

此成员指定应用重定位操作的位置。不同的目标文件对于此成员的解释会稍有不同。但对于可重定位文件，该值表示节偏移。重定位节说明如何修改文件中的其他节。重定位偏移会在第二节中指定一个存储单元。

**b）r\_info**

此成员指定必须对其进行重定位的符号表索引以及要应用的重定位类型。重定位类型特定于处理器。重定位项的重定位类型或符号表索引是ELF32\_R\_TYPE 或ELF32\_R\_SYM分别应用于项的r\_info成员所得的结果。对于 64 位 SPARC Elf64\_Rela 结构，r\_info 字段可进一步细分为 8 位类型标识符和 24 位类型相关数据字段。对于现有的重定位类型，数据字段为零。但是，新的重定位类型可能会使用数据位。

**c）r\_addend**

此成员指定常量加数，用于计算将存储在可重定位字段中的值，Rela 项包含显式加数。64 位 x86 仅使用 Elf64\_Rela 重定位项。因此，r\_addend 成员用作重定位加数。

接下来进行重定位计算（我们的机器是x86-64位）。

一个基于32位x86的重定位类型的计算。

a）对于R\_386\_PC32，计算方式为S + A - P；

b）对于R\_386\_PLT32，计算方式为L + A - P；

（其中A：用于计算可重定位字段的值的加数；P：使用 r\_offset 计算出的重定位的存储单元的节偏移或地址；S：索引位于重定位项中的符号的值；L：符号的过程链接表项的节偏移或地址）

设r为重定位条目，offset为偏移量，symbol为符号，type为类型，addend为加数

接下来具体到我们的实验中进行探索。

**a）对于第一个重定位条目：.rodata（对应第一个printf的字符串）。**r.offset=0x18，r.sympol=.rodata，r.type=R\_X86\_X64\_PC32，r.addend=-4（以上信息均来源于截图第一个重定位条目信息）。

首先，链接器计算出引用的运行时地址，refaddr = ADDR(s) +r.offset（节的地址加上偏移量），这里计算为：refaddr = ADDR(s) + 0x18。

然后，更新该引用，使得它在运行时指向真正的内容（.L1），\*refptr = (unsigned)(ADDR(r.sympol) + r.addend - refaddr)。这里计算为：\*refptr = (unsigned) (ADDR(r.sympol) + (-4) - refaddr)（ADDR(r.sympol)代表每个符号的运行时地址）。

最后，在得到的可执行目标文件中，我们便可以得到正确的引用地址，即完成对第一个重定位条目的重定位计算。

**b）对于第4和第6个和第7个重定位条目：.rodata（对应第二个printf的字符串）和sleepsecs（全局变量int型）和.text。**根据类型可知，它们都是32 位 PC 相对地址的引用，与上述第一条计算方式相同，不在过多说明。

**c）而对于puts，printf，sleep和getchar函数是共享库内的函数，根据本节开始时的知识，计算方式为L + A - P。**具体到函数中，是这样的。

对于printf函数，第一次调用时，不直接调用printf函数，而是程序通过调用进入PLT[3]（它是printf的PLT条目）；

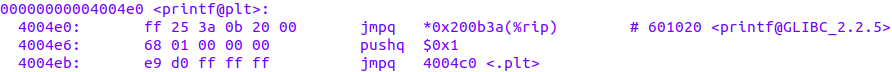
第一条PLT指令通过GOT[5]进行间接跳转（因为每个GOT条目初始时都指向它对应的PLT条目的第二条指令，这个间接跳转只是简单的把控制传送回PLT[3]中的下一条指令）；

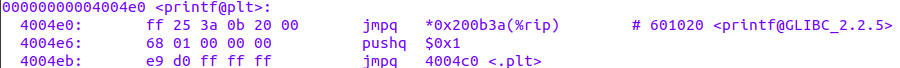
第三步，将printf函数的ID（0x1）压入栈中，PLT[3]跳转到PLT[0]；

第四步，PLT[0]将GOT[1]间接的把动态链接器的第一个参数压入栈中，然后通过GOT[3]间接跳转到动态链接器中。动态连接器使用两条栈条目确定printf函数的运行时位置，用这个地址重写GOT[5]，再把控制传递给printf函数。

还是对于printf函数，对于第二次调用。和前面一样，控制传递到PLT[3]，不过这次通过GOT[5]的间接跳转，会将控制直接转移到printf函数。（如截图）

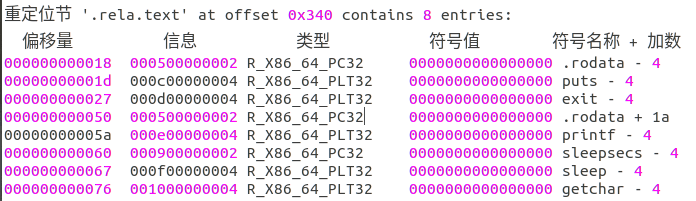


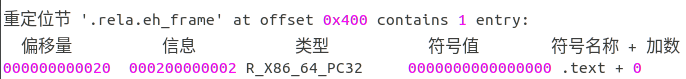




针对printf共享库函数

其他函数与之类似，不再详述了。

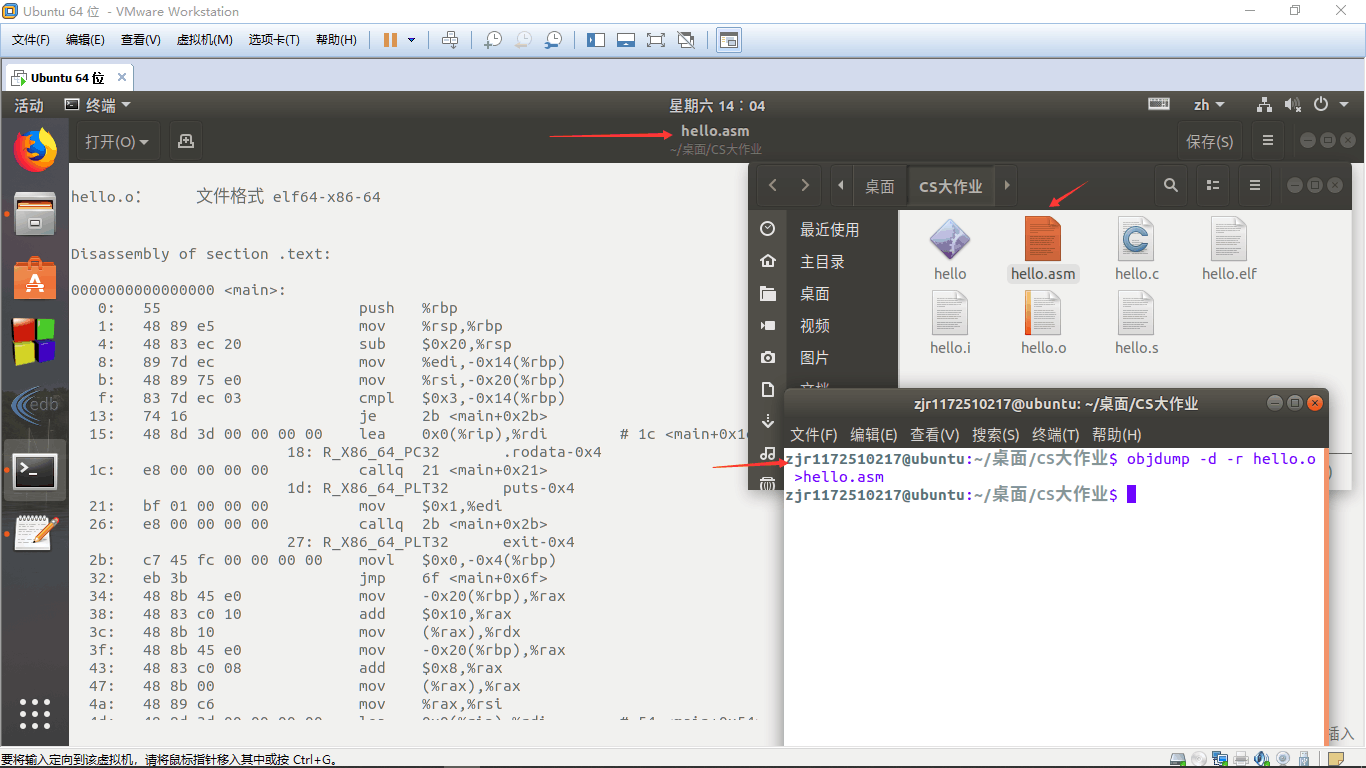




截图，重定位节.rela.text和.rela.eh\_frame

## 4.4 Hello.o的结果解析

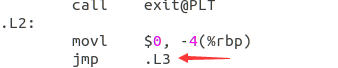
* **命令行输入：objdump -d -r hello.o >hello.asm**（附上截图）



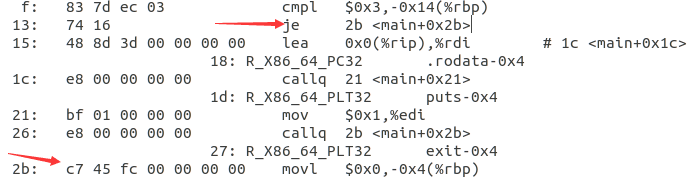
**截图：objdump -d -r hello.o >hello.asm**

* **分析hello.o的反汇编代码（即分析hello.asm文本文件）与hello.s文件的区别（总体大致相同，只有小部分区别）**

1. **分支转移：**hello.s文件中分支转移是使用段名称进行跳转的（附截图1），而hello.o文件中分支转移是通过地址进行跳转的（附截图2）。



截图1：hello.s分支跳转



截图2：hello.o分支转移

1. **函数调用：**hello.s文件中，函数调用call后跟的是函数名称（附截图3）；而在我们的hello.o文件中，call后跟的是下一条指令。而同时因为这些函数都是共享库函数，这时候地址是不确定的，因此call指令将相对地址设置为全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，等待链接的进一步确定（截图4）。

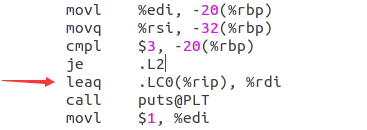
TW9P)F`86`)RKY1H(2B{(E9

截图3：hello.s调用函数



截图4：hello.o调用函数

1. **全局变量：**hello.s文件中，全局变量是通过语句：段地址+%rip完成的（截图5）；对于hello.o的反汇编来说，则是：0+%rip，因为.rodata节中的数据是在运行时确定的，也需要重定位，现在填0占位，并为其在.rela.text节中添加重定位条目（截图6）。



截图5：hello.s访问变量

LM)FTY5WRLX2LHC}E8)~OU2

截图6：hello.o访问变量

* **说明机器语言**

机器语言：二进制的机器指令的集合；

机器指令：由操作码和操作数构成的；

机器语言：灵活、直接执行和速度快。

汇编语言：主体是汇编指令，是机器指令便于记忆的表示形式，为了方便程序员读懂和记忆的语言指令。

汇编指令和机器指令在指令的表示方法上有所不同。

## 4.5 本章小结

本章对汇编结果进行了详尽的介绍。与我们的hello.o文件相结合，介绍了汇编的概念与作用，以及在Ubuntu下汇编的命令。同时本章主要部分在于对可重定位目标elf格式进行了详细的分析，侧重点在重定位项目上。同时对hello.o文件进行反汇编，将hello.asm（我的反汇编文件）与之前生成的hello.s文件进行了对比。使得我们对该内容有了更加深入地理解。

# 第5章 链接

## 5.1 链接的概念与作用

* **概念**

是将各种代码和数据片段收集并组合为单一文件的过程，这个文件可以被加载（复制）到内存并执行。

* **作用**

1）链接可以执行于编译时，也就是源代码被翻译成机器代码时；也可以执行于加载时，即程序被加载器加载到内存并执行时；甚至执行于运行时，也就是由应用程序来执行。

2）链接使得分离编译（seperate compila）成为可能。更便于我们维护管理，我们可以独立的修改和编译我们需要修改的小的模块。

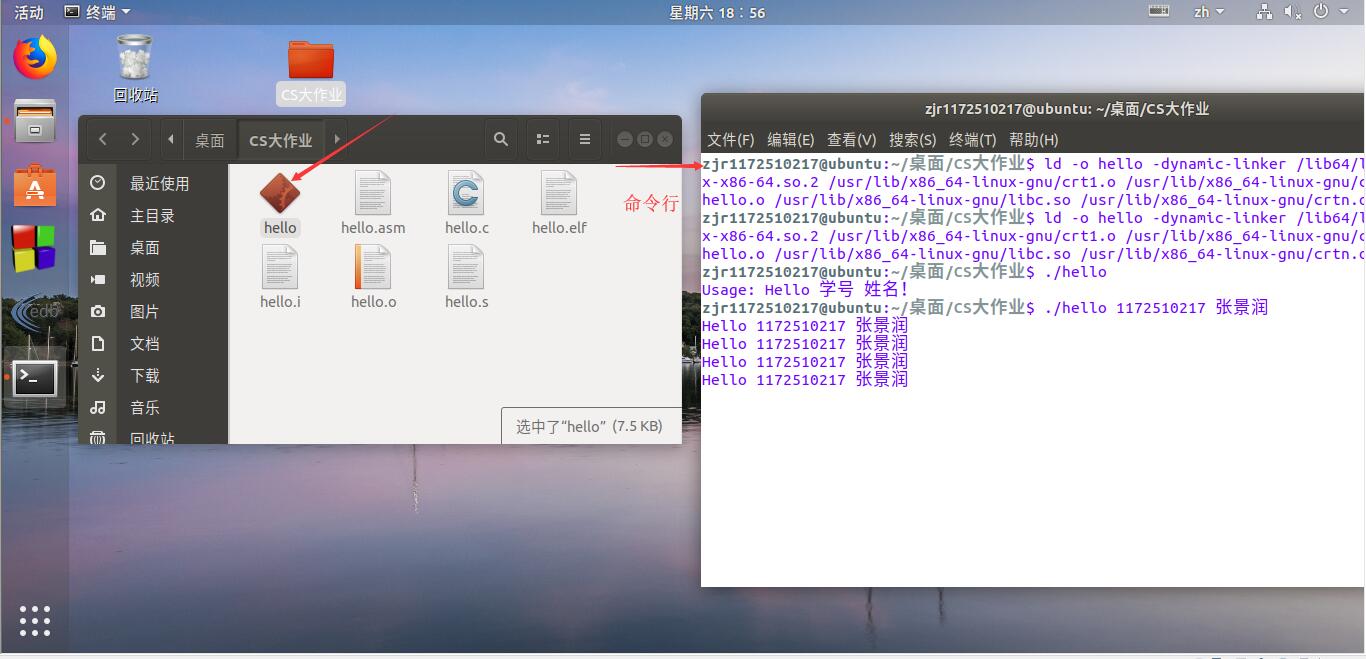
## 5.2 在Ubuntu下链接的命令

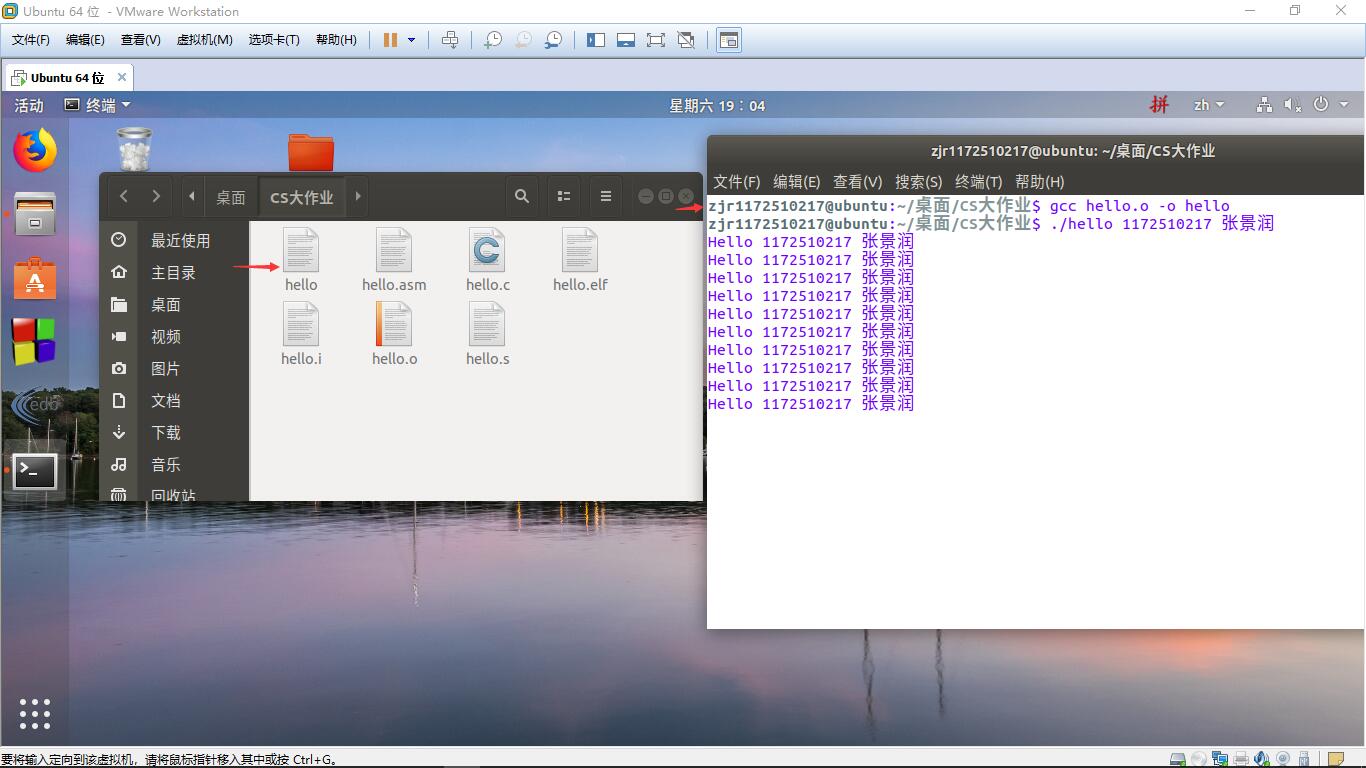
* **命令行**

**方法1：**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

**方法2：**gcchello.o -o hello

* 命令行截图

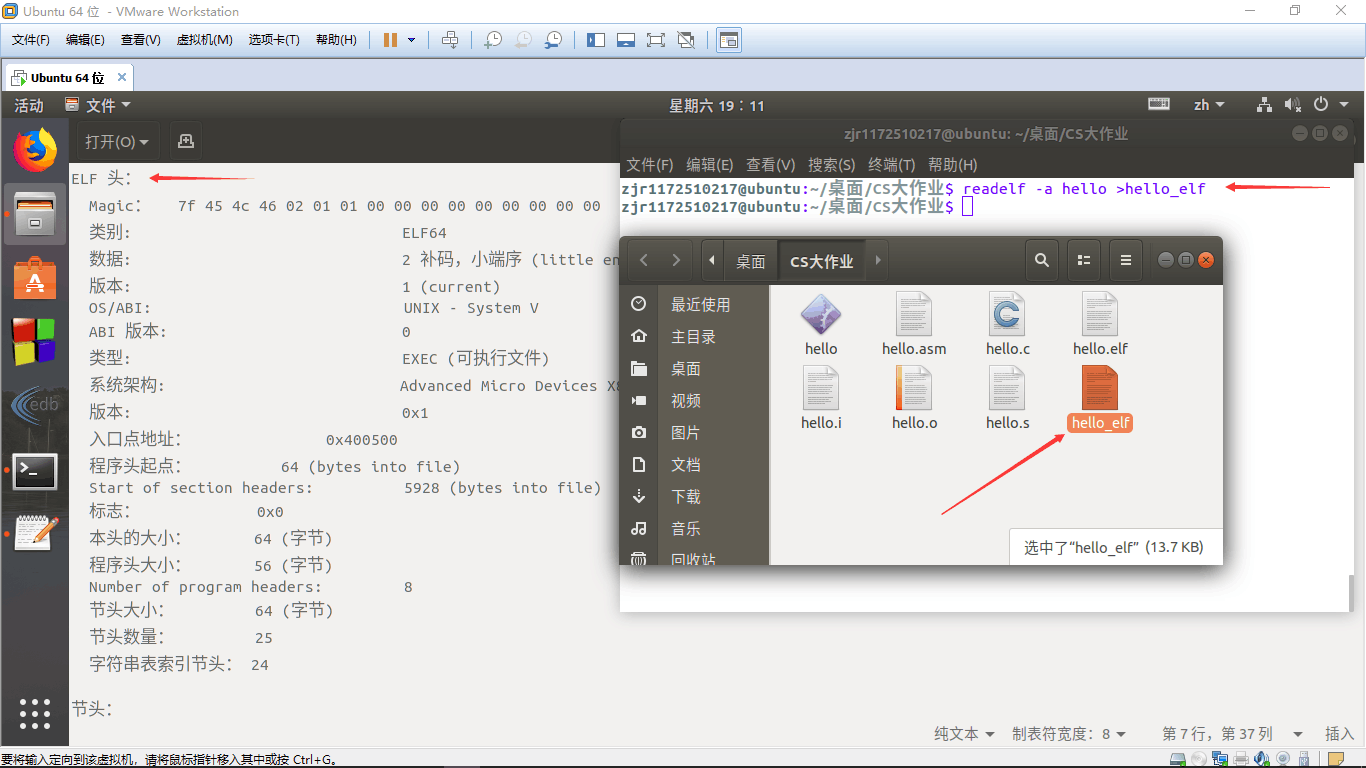


方法1截图

方法2截图

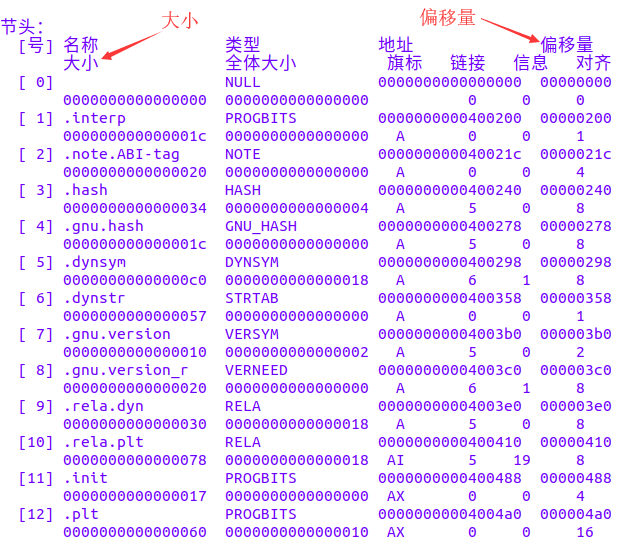
## 5.3 可执行目标文件hello的格式

* **获取hello的elf格式文件**：readelf -a hello >hello\_elf（这里输出的文件区别于前面的hello.elf），附上截图1。

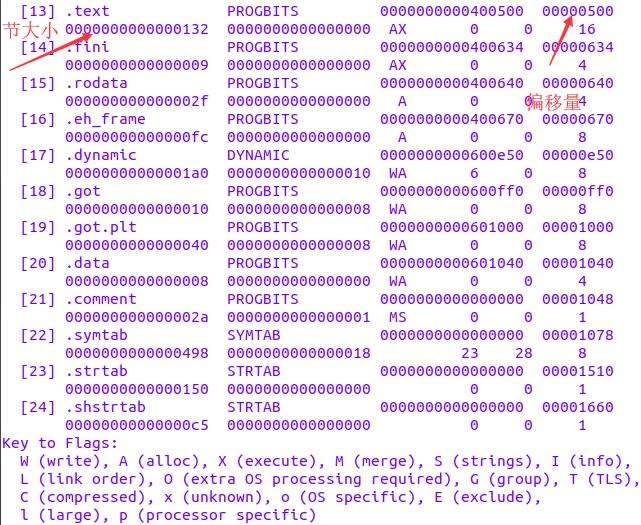


截图1：hello的ELF格式

* 各节的基本信息均在节头表（描述目标文件的节）中进行了声明。节头表（包括名称，大小，类型，全体大小，地址，旗标，偏移量，对齐等信息），下面是它的截图。



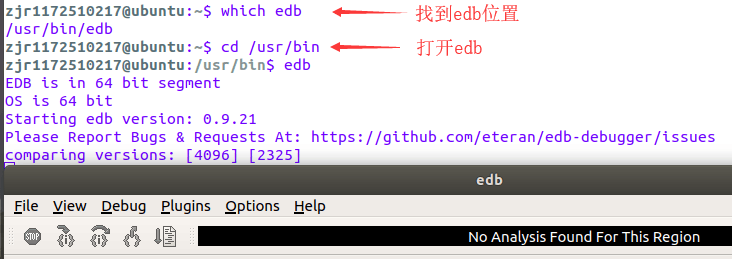
截图1，节头表截图上半部分



截图2，节头表截图下半部分

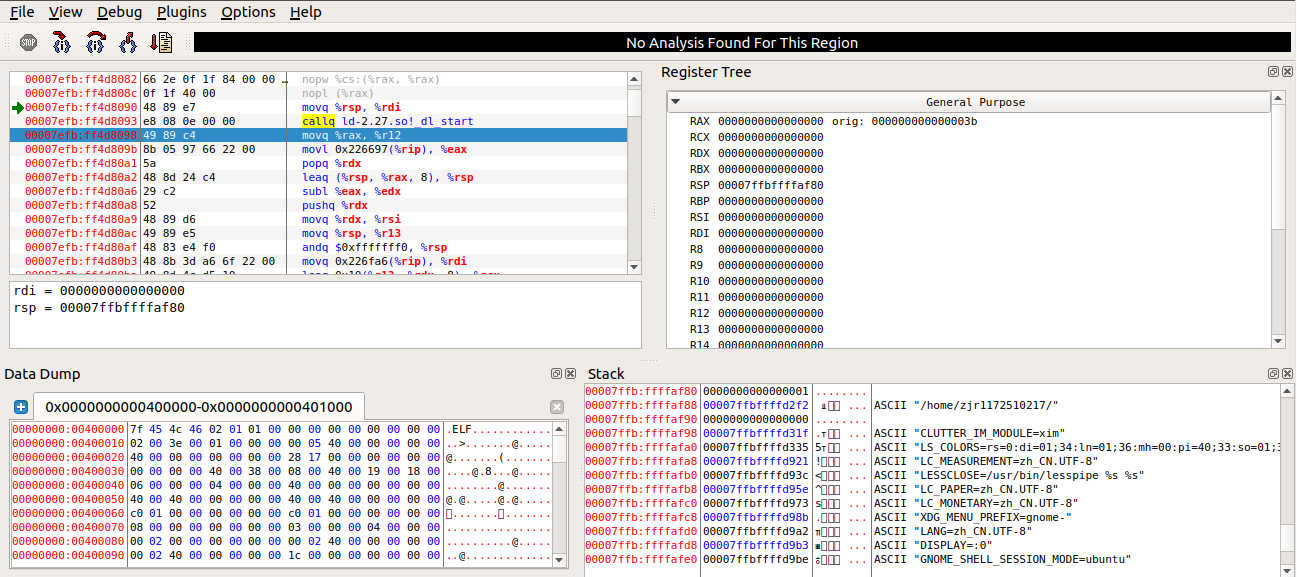
## 5.4 hello的虚拟地址空间

* 第1步，找到我的edb位置，在文件中打开edb，具体操作如截图1.



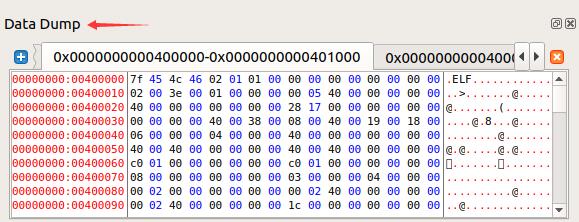
截图1，找到并打开edb

* 第2步，在edb中找到并加载hello可执行文件，操作后如截图2。



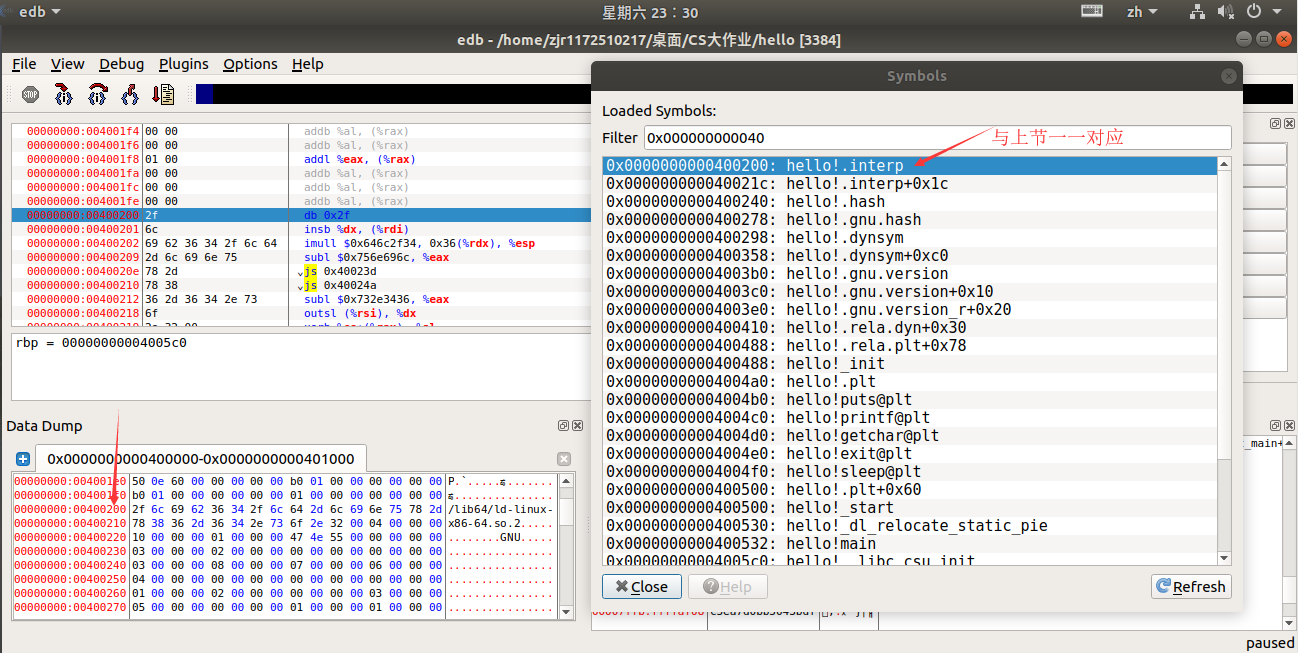
截图2，打开hello文件

* 第3步，观察edb的Data Dump窗口。窗口显示虚拟地址由0x400000开始，到0x400fff结束，这之间的每一个节对应5.3中的每一个节头表的声明，如截图3。



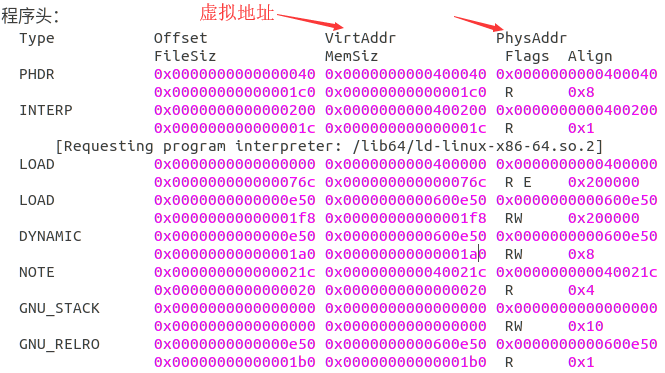
截图3，edb的Data Dump窗口

* 第4步，观察edb的Sympols小窗口。我们发现确实从虚拟地址从0x400000开始和5.3节中的节头表是一一对应的（从.interp节到..en\_frame对应），如截图4。



截图4，Symbols窗口观察

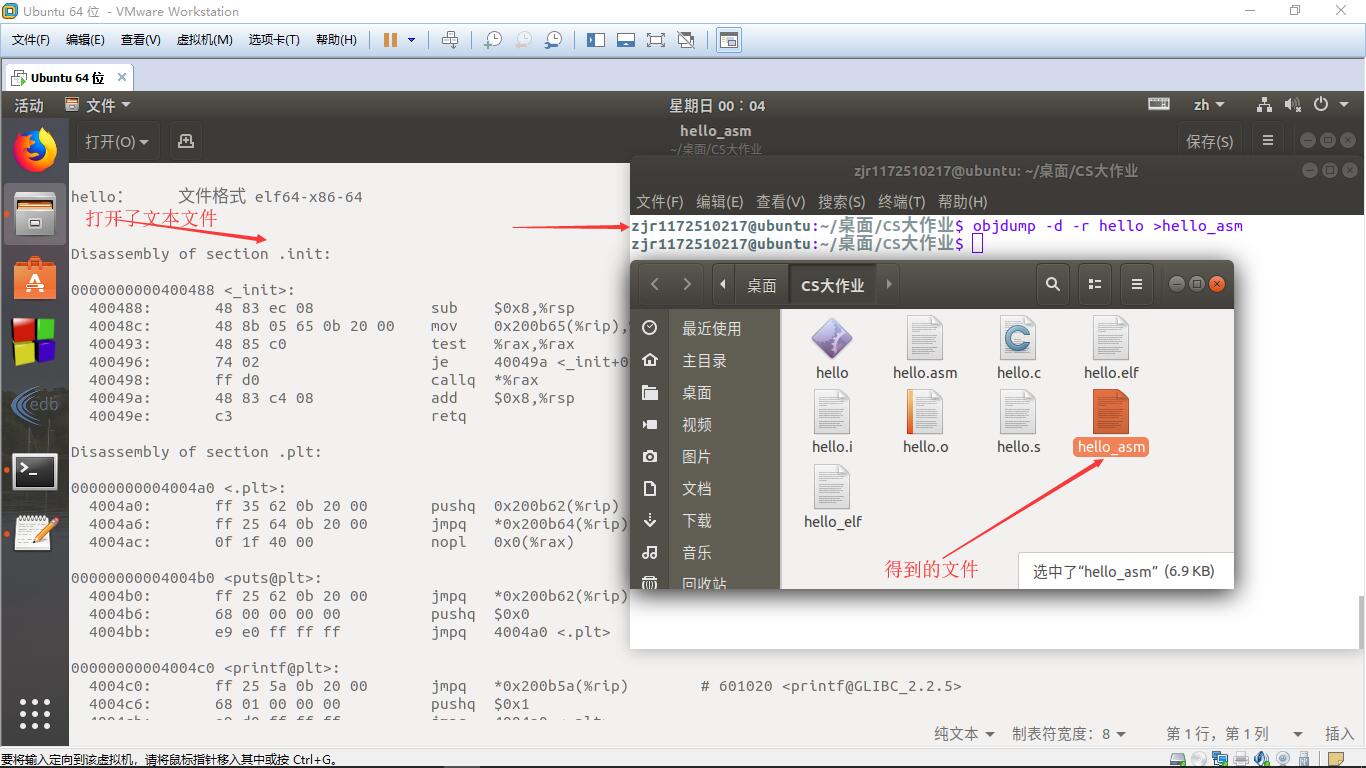
* 第5步，关于5.3节节头表中的.dynamic到.shstrtab的处理。首先查看hello的elf格式文件重的程序头，它包含的信息：类型，偏移，虚拟地址，物理地址，对齐，标志等，如截图5。通过Data Dump窗口查看虚拟地址段 0x600000到0x602000的部分，在0到fff的空间中，与0x400000到0x401000段的存放的程序相同；而在 fff之后存放的是.dynamic到.shstrtab节。



截图5，hello的elf文件中的程序头部分

## 5.5 链接的重定位过程分析

* 第1步，反汇编hello得到objdump -d -r hello >hello\_asm（将反汇编文件输出到文件hello\_asm中，要注意与hello.asm的区别），如截图1。



截图1，输出hello的反汇编文件

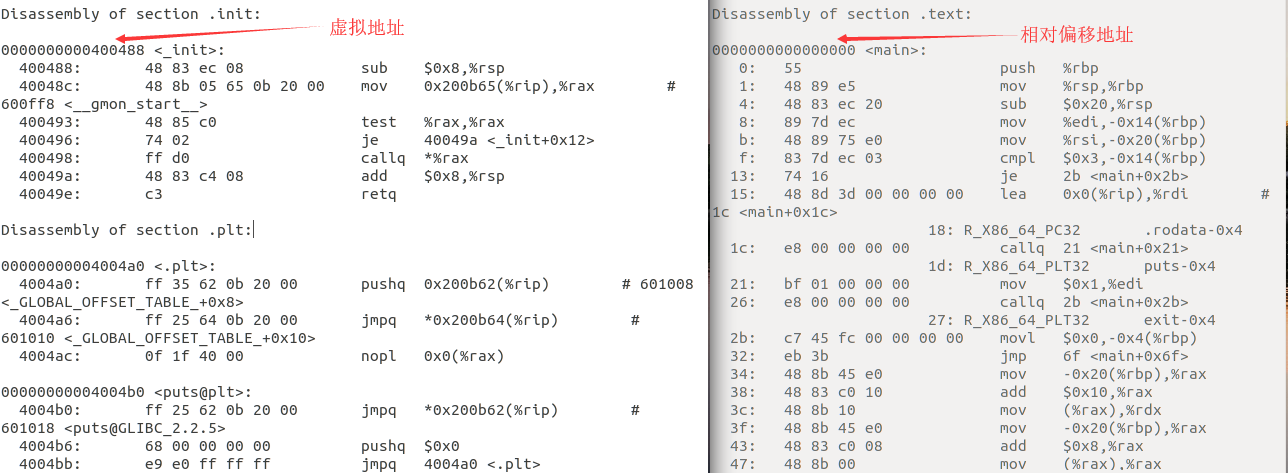
* 第2步，分析列举hello反汇编文件与hello.o反汇编文件的区别（即hello.asm与hello\_asm的对比）。

1）我们发现hello\_asm比hello.asm多了许多文件节。比如.init节和.plt节（hello.o反汇编得到的hello.asm中只有.text节），如截图2



截图2，hello\_asm比hello.asm多了许多节

2）hello\_asm（hello反汇编）文件中的地址是虚拟地址，而hello.asm（hello.o反汇编）节中的是相对偏移地址，如截图3。



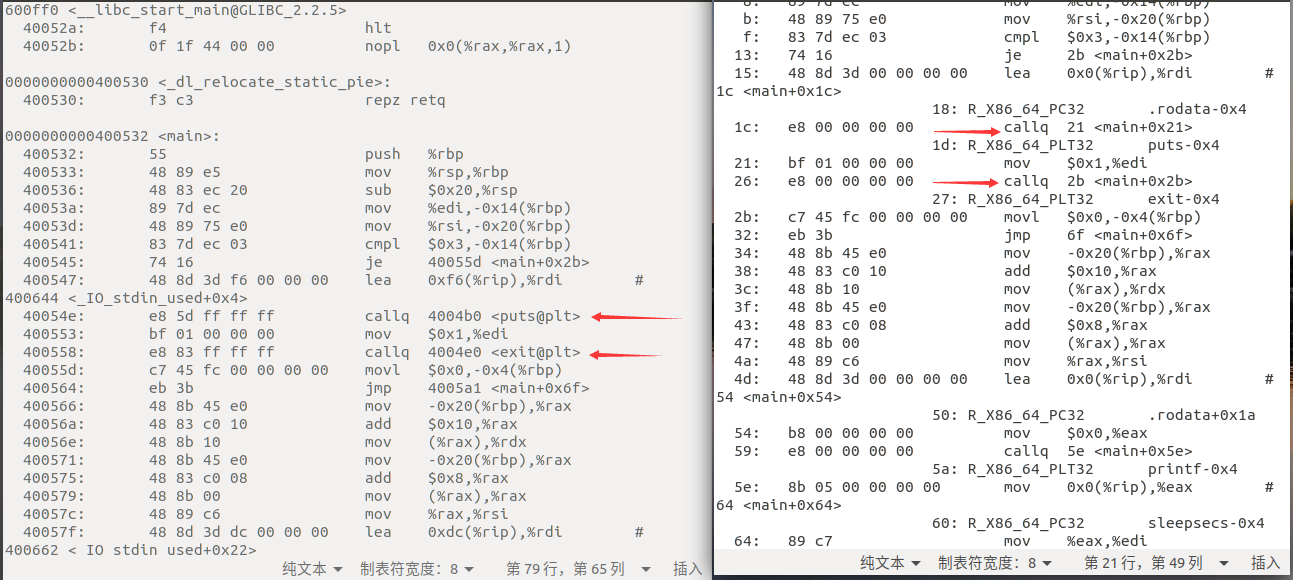
截图3，hello\_asm中各节的地址换为了虚拟地址

3）hello\_asm中增加了许多外部链接的共享库函数。如puts@plt共享库函数，printf@plt共享库函数以及getchar@plt函数等，如截图4。



截图4，hello\_asm中对比hello.asm多出来的函数

4）跳转和函数调用的地址在hello\_asm中是虚拟内存地址（都以main函数内部调用puts函数和exit函数为例），如截图5。

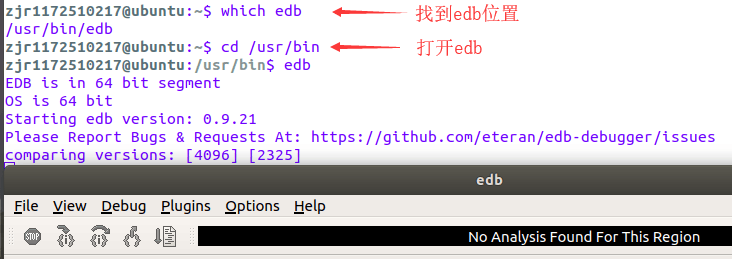


截图5，hello\_asm中函数调用使用虚拟地址

* 第3步，链接的重定位过程说明。要合并相同的节，确定新节中所有定义符号在虚拟地址空间中的地址，还要对引用符号进行重定位（确定地址），修改.text节和.data节中对每个符号的引用（地址），而这些需要用到在.rel\_data和.rel\_text节中保存的重定位信息。

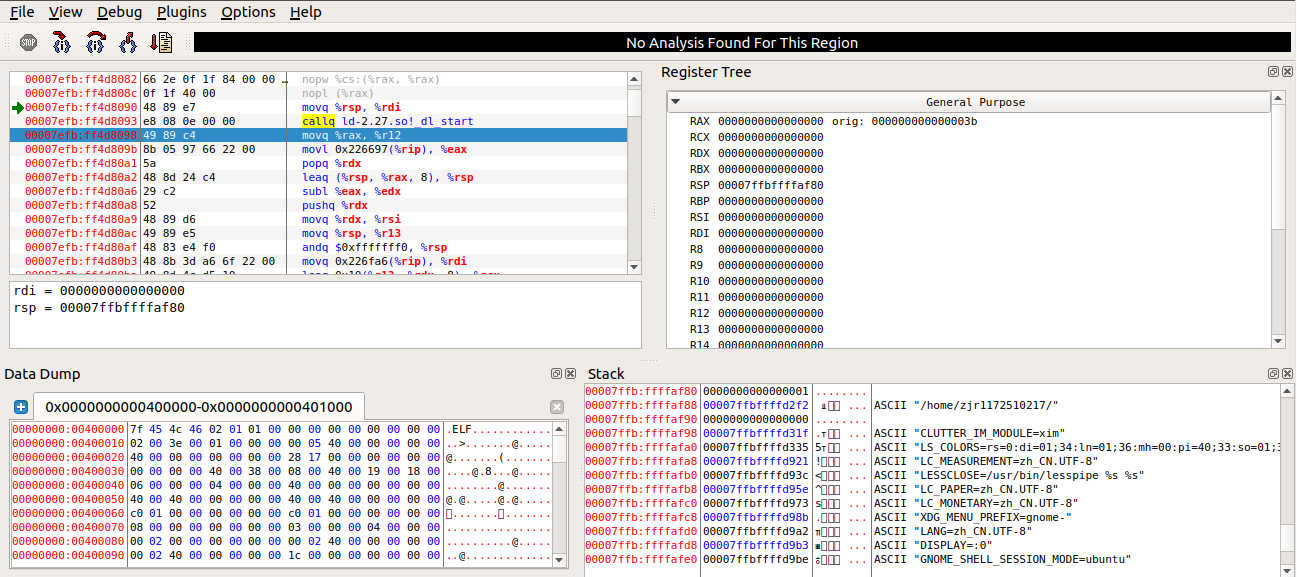
## 5.6 hello的执行流程

* **第1步，找到我的edb位置，在文件中打开edb，具体操作如截图1。**



截图1，找到并打开edb

* **第2步，在edb中找到并加载hello可执行文件，操作后如截图2。**



截图2，打开hello文件

* **第3步，列出所有过程（第一种情况：终端输入./hello 1172510217 张景润）。**

|  |  |
| --- | --- |
| 子程序名 | 程序地址（16进制） |
| ld -2.27.so!\_dl\_start | 7efb ff4d8ea0 |
| ld-2.27.so!\_dl\_init | 7efb ff4e7630 |
| hello!\_start | 400500 |
| libc-2.27.so!\_\_libc\_start\_main | 7efb ff100ab0 |
| hello!printf@plt（调用了10次） | 4004c0 |
| hello!sleep@plt（调用了10次） | 4004f0 |
| hello!getchar@plt | 4004d0 |
| libc-2.27.so!exit | 7efbff122120 |

**edb真的太强大了，感觉像开了挂一样**

* **第4步，列出所有过程（第二种情况：终端输入./hello）**

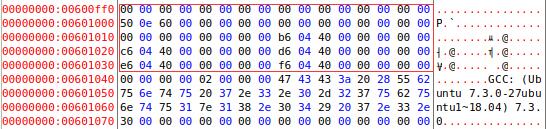
|  |  |
| --- | --- |
| 子程序名 | 程序地址（16进制） |
| ld-2.27.so!\_dl\_start | 7efb ff4d8ea0 |
| ld-2.27.so!\_dl\_init | 7efb ff4e7630 |
| hello!\_start | 400500 |
| libc-2.27.so!\_\_libc\_start\_main | 7efb ff100ab0 |
| hello!puts@plt | 4004b0 |
| hello!exit@plt | 4004e0 |

**edb感觉想修改什么就修改什么**

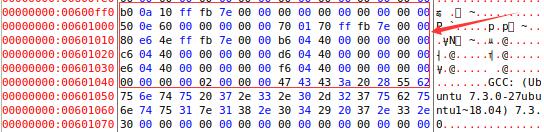
## 5.7 Hello的动态链接分析

**1）**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，所以需要为其添加重定位记录，并等待动态链接器处理。为避免运行时修改调用模块的代码段，链接器采用延迟绑定的策略。动态链接器使用过程链接表PLT和全局偏移量表GOT实现函数的动态链接。其中GOT 中存放函数目标地址，PLT使用 GO T中地址跳转到目标函数。

**2)**附上dl\_init函数调用前后GOT信息变化截图



dl\_init函数调用前



dl\_init函数调用后

**3）**我们进一步发现，改变的是：从地址0x6001008处，由00 00 00 00 00 00变为了70 01 70 ff fb 7e。由00 00 00 00 00 00变为80 e6 4e ff fb 7e。由于机器为小端，则这两处改编成的地址应该是0x7e fb ff 70 01 70和0x7e fb ff 4e e6 80。

**4）**在之后的函数调用时，首先跳转到PLT执行.plt中逻辑，第一次访问跳转时，GOT 地址为下一条指令，将函数序号压栈，然后跳转到PLT[0]，在 PLT[0]中将重 定位表地址压栈，然后访问动态链接器，在动态链接器中使用函数序号和重定位 表确定函数运行时地址，重写 GOT，再将控制传递给目标函数。之后如果对同样函数调用，第一次访问跳转直接跳转到目标函数。

## 5.8 本章小结

本章结合实验中的hello可执行程序依此介绍了链接的概念及作用，在Ubuntu下链接的命令行；并对hello的elf格式进行了详细的分析对比，同时注意到了hello的虚拟地址空间知识；并通过反汇编hello文件，将其与hello.o反汇编文件对比，详细了解了重定位过程；遍历了整个hello的执行过程，在最后对hello进行了动态链接分析。相信通过本章的介绍，你已经对hello的链接过程烂熟于心了。

# 第6章 hello进程管理

## 6.1 进程的概念与作用

* **概念**

狭义定义：进程是正在运行的程序的实例（an instance of a computer program that is being executed）。

广义定义：进程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。它是操作系统动态执行的基本单元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元。

* **作用**

1）在现代计算机中，进程为用户提供了以下假象：我们的程序好像是系统中当前运行的唯一程序 一样，我们的程序好像是独占的使用处理器和内存，处理器好像是无间断的执行 我们程序中的指令，我们程序中的代码和数据好像是系统内存中唯一的对象。

2）每次用户通过向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，然后在这个新进程的上下文中运行这个可执行目标文件。应用程序也能够创建新进程，并且在这个新进程的上下文中运行它们自己的代码或其他应用程序。

3）进程提供给应用程序两个关键抽象：一个独立的逻辑控制流；一个私有的地址空间。

## 6.2 简述壳Shell-bash的作用与处理流程

* **作用**

Shell是用户与操作系统之间完成交互式操作的一个接口程序，它为用户提供简化了的操作。而NU组织发现sh是比较好用的又进一步开发Borne Again Shell，简称bash，它是Linux系统中默认的shell程序。

* **处理流程**

1）将用户输入的命令行进行解析，分析是否是内置命令；

2）若是内置命令，直接执行；若不是内置命令，则bash在初始子进程的上下文中加载和运行它。

3）本质上就是shell在执行一系列的读和求值的步骤，在这个过程中，他同时可以接受来自终端的命令输入。

## 6.3 Hello的fork进程创建过程

执行中的进程调用fork()函数，就创建了一个子进程。其函数原型为pid\_t fork(void)；对于返回值，若成功调用一次则返回两个值，子进程返回0，父进程返回子进程ID；否则，出错返回-1。

* 首先对于hello进程。我们终端的输入被判断为非内置命令，然后shell试图在硬盘上查找该命令（即hello可执行程序），并将其调入内存，然后shell将其解释为系统功能调用并转交给内核执行。
* shell执行fork函数，创建一个子进程。这时候我们的hello程序就开始运行了。值得注意的是，hello子进程是父进程的副本，它将获得父进程数据空间、堆、栈等资源的副本。但是子进程持有的是上述存储空间的“副本”，这意味着父子进程间不共享这些存储空间。
* 同时Linux将复制父进程的地址空间给子进程，因此，hello进程就有了独立的地址空间。
* 画出进程图

---------------hello程序----

|

---------+------------------------------

fork

## 6.4 Hello的execve过程

* execve函数在当前进程的上下文中加载并运行新程序hello。函数原型为：int exeve(const char \*filename, const char \*argv[], const char \*envp[])；如果成功，则不返回；如果错误，则返回-1。
* 在execve加载了hello之后，它调用启动代码。启动代码设置栈，并将控制传递给hello的主函数（即main函数），该函数有以下原型：int main(int argv, char \*\*argv, char \*\*envp)或者等价的int main(int argc, char \*argv[], char \*envp).
* 当main开始执行时，一个典型的用户栈组织结构如下。

|  |
| --- |
| 以null结尾的环境变量字符串（栈底） |
| 以null结尾的命令行字符串 |
| envp[n] = =NULL |
| envp[n-1] |
| ... |
| envp[0] |
| argv[argc] = NULL |
| argv[argc-1] |
| ... |
| argv[0] |
| libc\_start\_main的栈帧 |
| main未来的栈帧（栈顶） |

* 1）hello子进程通过execve系统调用启动加载器。

2）加载器删除子进程所有的虚拟地址段，并创建一组新的代码、数据、堆段。新的栈和堆段被初始化为0。

3）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），新的代码和数据段被初始化为可执行文件中的内容。

4）最后加载器跳到\_start地址，它最终调用hello的main 函数。除了一些头部信息，在加载过程中没有任何从磁盘到内存的数据复制。直到CPU引用一个被映射的虚拟页时才会进行复制，此时，操作系统利用它的页面调度机制自动将页面从磁盘传送到内存。

* 做出linux x86-64运行的内存映像

|  |
| --- |
| 内核内存 |
| 用户栈（运行时创建） |
|  |
| 共享库的内存映射区域 |
|  |
| 运行时堆（由malloc创建） |
| 读/写段（.data，.bss） |
| 只读代码段（.init .text .rodata） |
|  |

黄色标注部分从hello文件中加载的部分

## 6.5 Hello的进程执行

* **进程时间片**

一个进程执行他的控制流的一部分的每一个时间段叫做时间片（time slice），多任务也叫时间分片（time slicing）

* **进程上下文切换**

调度：在进程执行的某些时刻，内核可以决定抢占当前进程，并重新开始一个先前被强占的进程。这种决策就叫调度（是由内核中的调度器的代码处理的）。

上下文切换：在内核调度了一个新的进程运行时，它就抢占当前进程，并使用一种上下文切换的机制来控制转移到新的进程。

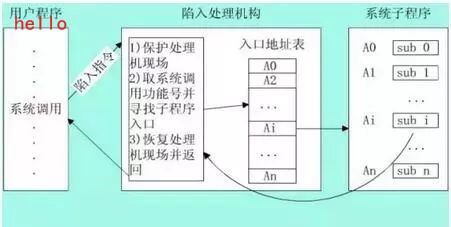
1）保存当前进程的上下文

2）恢复某个先前被强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

* 具体的用户态核心态转换

进程hello初始运行在用户模式中，直到它通过执行系统调用函数sleep或者exit时便陷入到内内核。内核中的处理程序完成对系统函数的调用。之后，执行上下文切换，将控制返回给进程hello系统调用之后的那条语句。





## 6.6 hello的异常与信号处理

* **hello的异常**

**1）中断：来自处理器外部的I/O设备的信号的结果。**

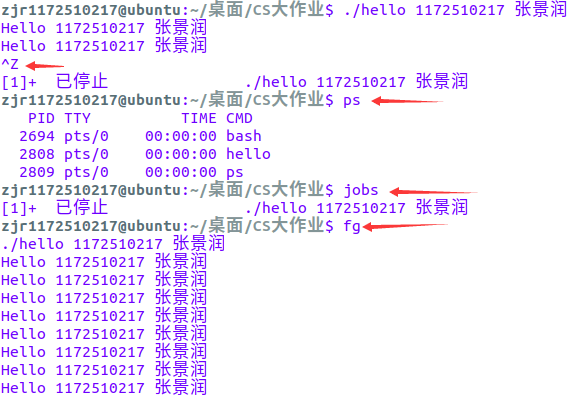
键盘上敲击CTRL -C或者CTRL-Z

**2）陷阱：有意的，执行指令的结果（例如：系统调用）**

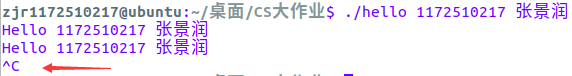
* **产生的信号**

**SIGINT,SIGSTP,SIGCONT,SIGWINCH**

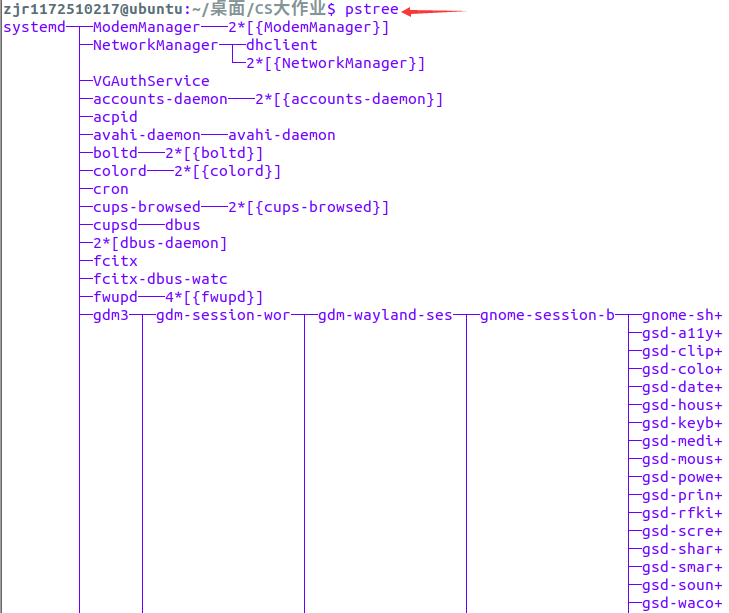
* **运行截图**



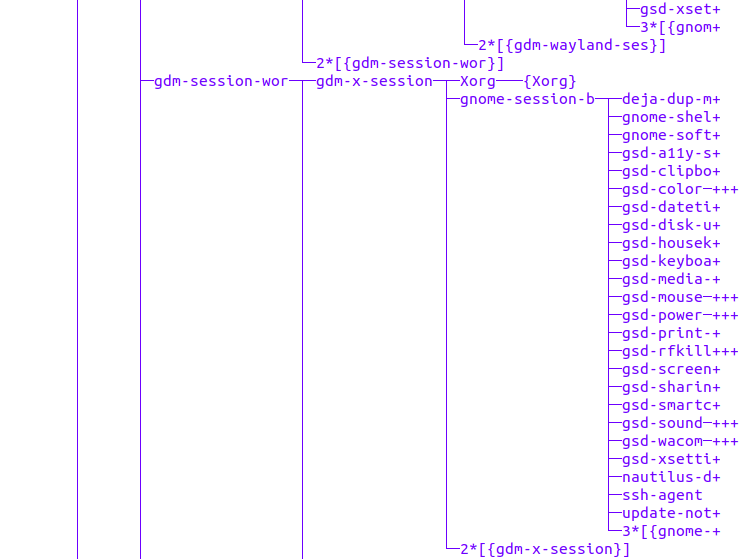
**截图1:ctrl-z，ps，jobs，fg**



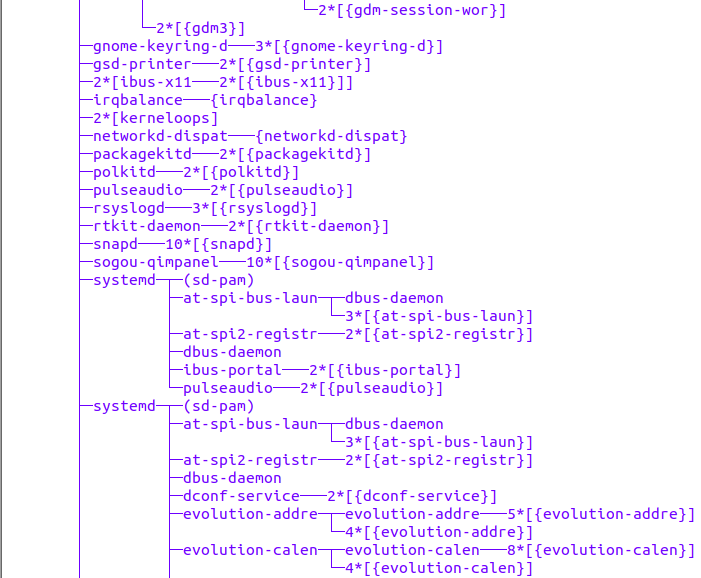
**截图2:ctrl+c（中断）**



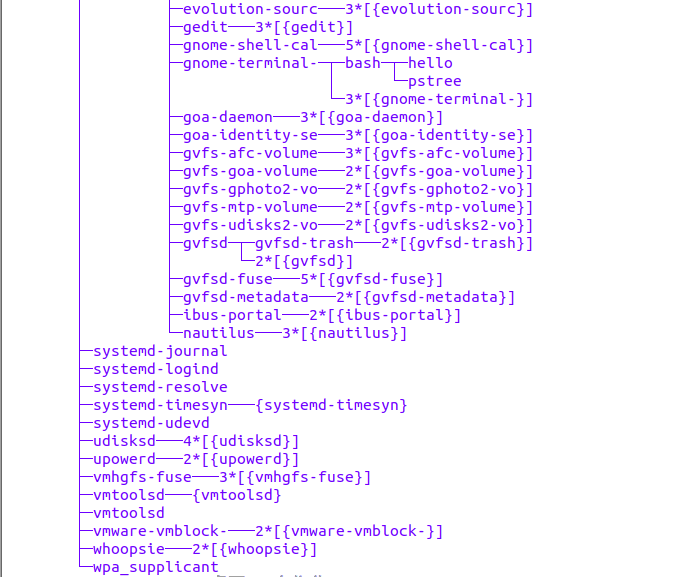
**截图3：pstree上半部分**



截图4：pstree中间部分



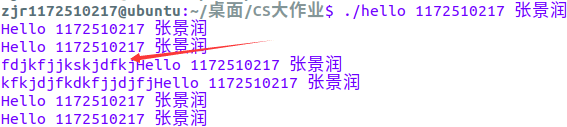
截图5：pstree中间部分



截图6：pstree最后部分



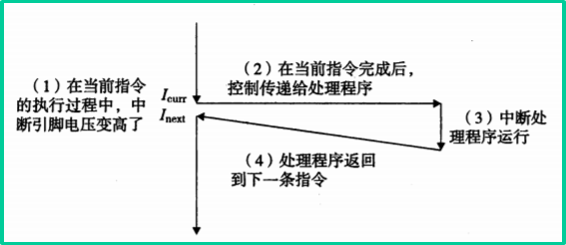
截图7：kill杀死hello程序



截图8：终端随意输入

* **异常与信号的处理**

1）键盘输入ctrl+c属于中断异常，其处理情况如截图。



截图：ctrl+c或者ctrl+z键盘的**中断**异常

2）函数执行可能会执行系统调用函数exit，属于陷阱。其处理方式如截图。



截图：exit系统调用函数的**陷阱**异常

3）信号处理

a）**对于ctrl+c或者ctrl+z**。键盘键入后，内核就会发送SIGINT或者SIGSTP。SIGINT信号默认终止前台job即程序hello，SIGSTP默认挂起前台hello作业。

b）**对于fg信号**。内核发送SIGCONT信号，我们刚刚挂起的程序hello重新在前台运行。

c）**对于kill -9 2860。**内核发送SIGKILL信号给我们指定的pid（hello程序），结果是杀死了hello程序。

## 6.7本章小结

异常控制流发生在计算机系统的各个层次，是计算机系统中提供并发的基本机制。

1）在硬件层，异常是由处理器中的事件触发的控制流中的突变

2）在操作系统层，内核用ECF提供进程的基本概念。

3）在操作系统和应用程序之间的接口处，应用程序可以创建子进，等待他们的子进程停止或者终止，运行新的程序，以及捕获来自其他进程的信号。

4）最后在应用层，C程序可以使用非本地跳转来规避正常的调用/返回栈规则，并且直接从一个函数分支到另一个函数。

同时还有四种不同类型的异常：中断，故障，终止和陷阱。

# 第7章 hello的存储管理

## 7.1 hello的存储器地址空间

* **逻辑地址**

逻辑地址（Logical Address）是指由程序hello产生的与段相关的偏移地址部分（hello.o）。

* **线性地址**

线性地址（Linear Address）是逻辑地址到物理地址变换之间的中间层。程序hello的代码会产生逻辑地址，或者说是（即hello程序）段中的偏移地址，它加上相应段的基地址就生成了一个线性地址。

* **虚拟地址**

有时我们也把逻辑地址称为虚拟地址。因为与虚拟内存空间的概念类似，逻辑地址也是与实际物理内存容量无关的，是hello中的虚拟地址。

* **物理地址**

物理地址（Physical Address）是指出现在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hello的线性地址会使用页目录和页表中的项变换成hello的物理地址；如果没有启用分页机制，那么hello的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

* 实模式下：逻辑地址CS：EA到物理地址CS\*16+EA
* 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，段地址+偏移地址=线性地址。
* 段选择符各字段含义

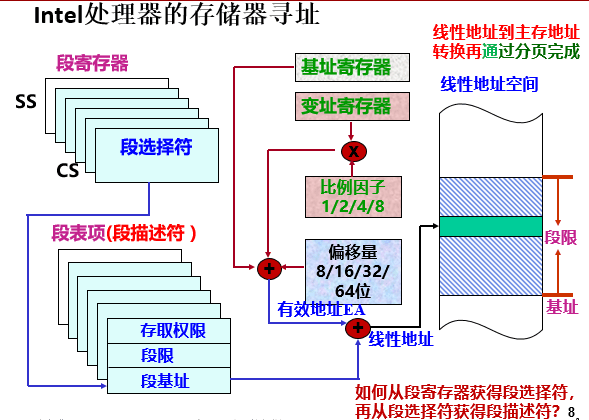
|  |  |  |
| --- | --- | --- |
| 15 14 | 32 | 10 |
| 索引 | TI | RPL |

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

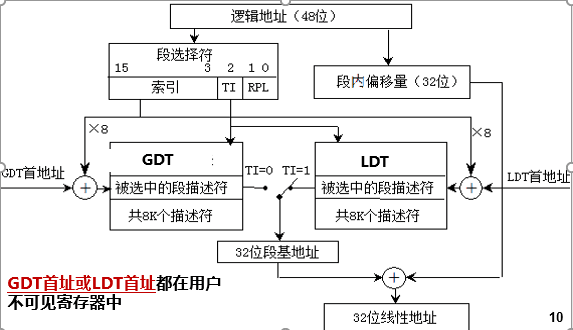
于最低级的用户态，第0级高于第3级。

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



截图1：intel储存器寻找

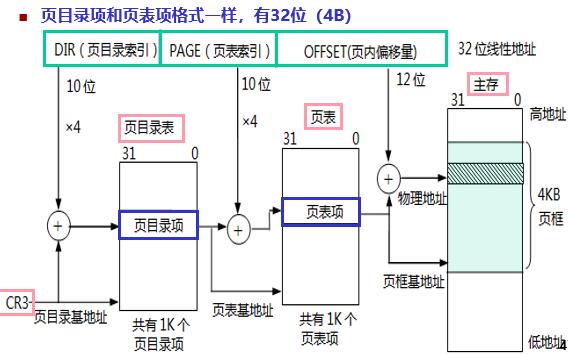
* 被选中的段描述符先被送至描述符cache，每次从描述符cache中取32位段基址，与32位段内偏移量（有效地址）相加得到线性地址（附上截图）



截图2：逻辑地址->线性地址

## 7.3 Hello的线性地址到物理地址的变换-页式管理

* hello的线性地址空间划分：4GB = 1K个子空间\*1K个页面/子空间\*4KB/页，如截图2



截图2：线性地址到物理地址

* 页目录项以及页表项，如截图。

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 基地址 | AVL | 0 | 0 | D | A | PCD | PWT | U/S | R/W | P |

1）P：1表示页表或页在主存中；P=0表示页表或页不在主存，即缺页，此时需将页故障线性地址保存到CR2。

2）R/W：0表示页表或页只能读不能写；1表示可读可写。

3）U/S：0表示用户进程不能访问；1表示允许访问。

4）PWT：控制页表或页的cache写策略是全写还是回写（Write Back）。

5）PCD：控制页表或页能否被缓存到cache中。

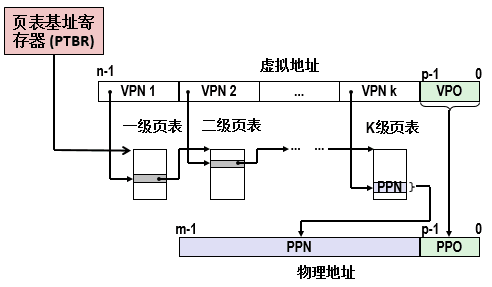
5）A：1表示指定页表或页被访问过，初始化时OS将其清0。利用该标志，OS可清楚了解哪些页表或页正在使用，一般选择长期未用的页或近来最少使用的页调出主存。由MMU在进行地址转换时将该位置1。

6）D：修改位(脏位dirty bit)。页目录项中无意义，只在页表项中有意义。初始化时OS将其清0，由MMU在进行写操作的地址转换时将该位置1。

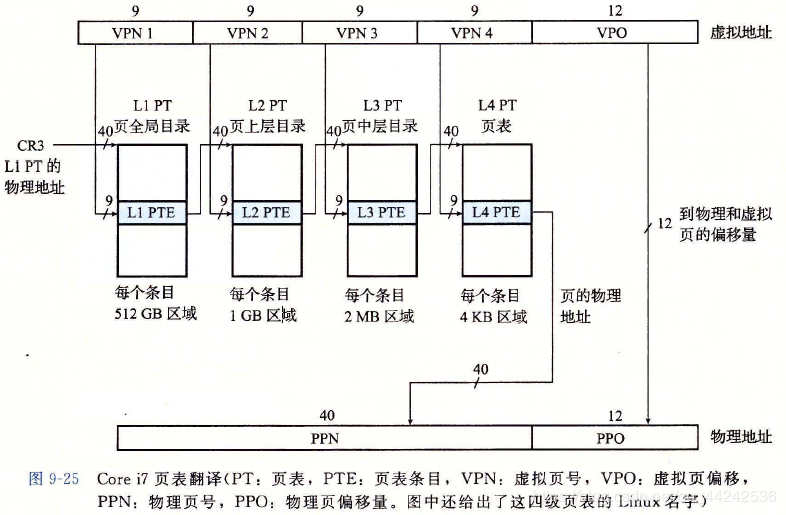
7）高20位是页表或页在主存中的首地址对应的页框号，即首地址的高20位。每个页表的起始位置都按4KB对齐。

## 7.4 TLB与四级页表支持下的VA到PA的变换

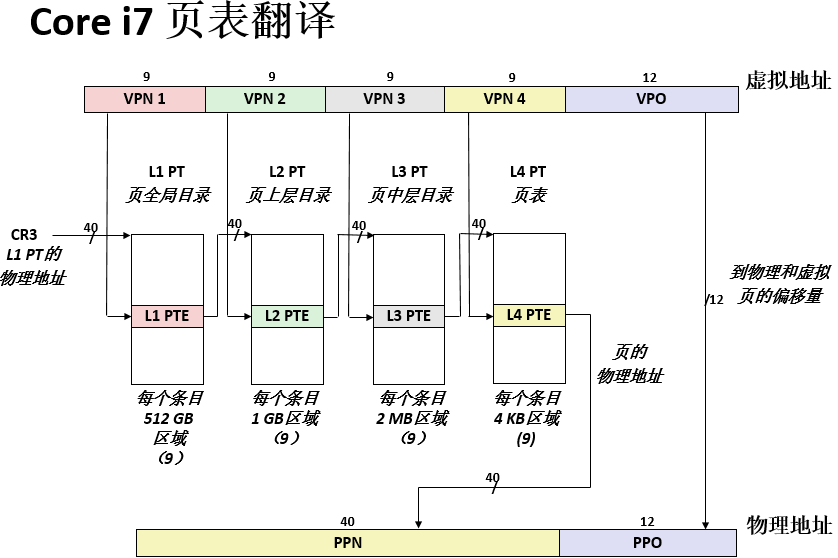
* Translating with a k-level Page Table使用K级页表的地址翻译



截图1：K级页表的地址翻译



截图2：四级页表翻译



截图3：针对core i7页表翻译

* Core i7 1-3级页表条目格式，附上表格



对于其中各表位，解释如下

每个条目引用一个 4KB子页表:

1）P: 子页表在物理内存中 (1)不在 (0).

2）R/W: 对于所有可访问页，只读或者读写访问权限.

3）U/S: 对于所有可访问页，用户或超级用户 (内核)模式访问权限.

4）WT: 子页表的直写或写回缓存策略.

5）A: 引用位 (由MMU 在读或写时设置，由软件清除).

6）PS: 页大小为4 KB 或 4 MB (只对第一层PTE定义).

7）Page table physical base address: 子页表的物理基地址的最高40位 (强制页表 4KB 对齐)

8）XD: 能/不能从这个PTE可访问的所有页中取指令。

* Core i7 第 4 级页表条目格式`1PYWCO_ZH9%$Z5FBUF(FT0

P: 子页表在物理内存中 (1)不在 (0).

R/W: 对于所有可访问页，只读或者读写访问权限.

U/S: 对于所有可访问页，用户或超级用户 (内核)模式访问权限.

WT: 子页表的直写或写回缓存策略.

A:引用位 (由MMU 在读或写时设置，由软件清除).

D: 修改位 (由MMU 在读和写时设置，由软件清除)

Page table physical base address: 子页表的物理基地址的最高40位 (强制页表 4KB 对齐)

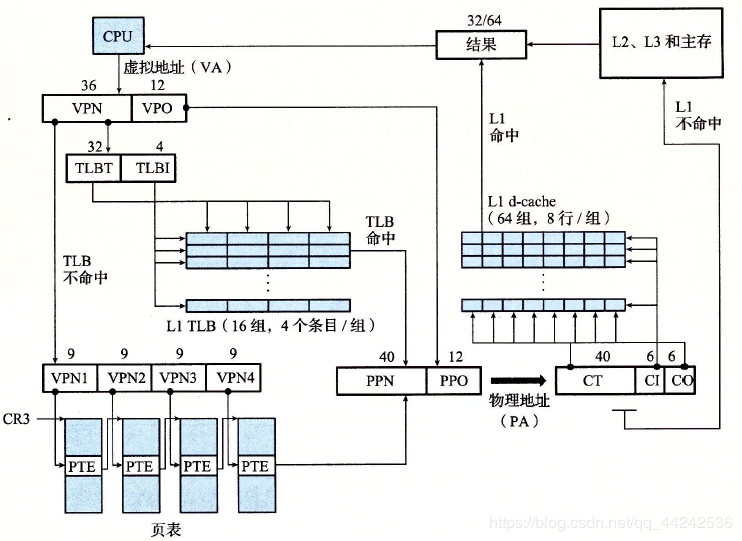
XD: 能/不能从这个PTE可访问的所有页中取指令.

## 7.5三级cache支持下的物理内存访问

1）得到了物理地址VA，首先使用物理地址的CI进行组索引（每组8路），对8路的块分别匹配 CT进行标志位匹配。如果匹配成功且块的valid标志位为1，则命中hit。然后根据数据偏移量 CO取出数据并返回。

2）若没找到相匹配的或者标志位为0，则miss。那么cache向下一级cache，这里是二级cache甚至三级cache中寻找查询数据。然后逐级写入cache。

3）在更新cache的时候，需要判断是否有空闲块。若有空闲块（即有效位为0），则写入；若不存在，则进行驱逐一个块（LRU策略）。

截图：3级cache

## 7.6 hello进程fork时的内存映射

**1）**虚拟内存和内存映射解释了fork函数如何为hello进程提供私有的虚拟地址空间。

**2）**fork为hello的进程创建虚拟内存

创建当前进程的的mm\_struct，vm\_area\_struct和页表的原样副本；两个进程中的每个页面都标记为只读；两个进程中的每个区域结构（vm\_area\_struct）都标记为私有的写时复制（COW）

**3）**在hello进程中返回时，hello进程拥有与调用fork进程相同的虚拟内存。

**4）**随后的写操作通过写时复制机制创建新页面

## 7.7 hello进程execve时的内存映射

**1）**在bash中的进程中执行了如下的execve调用：execve("hello",NULL,NULL)；

**2）**execve函数在当前进程中加载并运行包含在可执行文件hello中的程序，用hello替代了当前bash中的程序。

下面是加载并运行hello的几个步骤

**3）删除已存在的用户区域。**

**4）映射私有区域**

**5）映射共享区域**

**6）设置程序计数器（PC）**

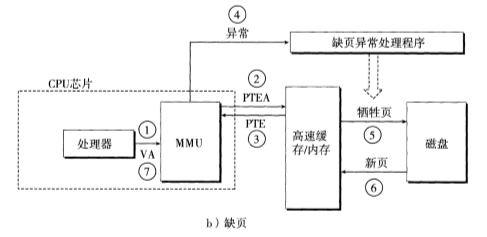
exceve做的最后一件事是设置当前进程的上下文中的程序计数器，是指指向代码区域的入口点。而下一次调度这个进程时，他将从这个入口点开始执行。Linux将根据需要换入代码和数据页面。



截图：加载器是如何映射用户地址空间的区域的

## 7.8 缺页故障与缺页中断处理

1）页面命中完全是由硬件完成的，而处理缺页是由硬件和操作系统内核协作完成的，如截图1。



截图1：缺页中断处理

下面是整体的处理流程1

2）处理器生成一个虚拟地址，并将它传送给MMU

3）MMU生成PTE地址，并从高速缓存/主存请求得到它

4）高速缓存/主存向MMU返回PTE

5）PTE中的有效位是0，所以MMU出发了一次异常，传递CPU中的控制到操作系统内核中的缺页异常处理程序。

6）缺页处理程序确认出物理内存中的牺牲页，如果这个页已经被修改了，则把它换到磁盘。

7）缺页处理程序页面调入新的页面，并更新内存中的PTE

8）缺页处理程序返回到原来的进程，再次执行导致缺页的命令。CPU将引起缺页的虚拟地址重新发送给MMU。因为虚拟页面已经换存在物理内存中，所以就会命中。

## 7.9动态存储分配管理

总的来说，动态内存分配器维护着一个进程的虚拟内存区域，称为堆（heap）。而分配器分为两种基本风格：显式分配器和隐式分配器。

1）而显式分配器必须在严格的约束条件下工作

必须处理任意请求序列；立即响应请求；只使用堆；对齐块；不修改已分配的块。

2）分配器的编写应该实现：吞吐率最大化；内存使用率最大化（两者相互冲突）。

3）我们需要注意这几个问题：空闲块组织方式；放置策略；分割策略；合并策略。

4）带边界标记的隐式空闲链表可以提高空闲块合并效率；显式空闲链表可以有效地实现空闲块的快速查找与合并等操作；分离空闲链表采用大小类的方式标记空闲块；分离适配方法快速而且内存使用效率较高。

|  |  |
| --- | --- |
| 31 3 | 210 |
| 块大小（头部） | a/f |
| pred（祖先） | |
| succ（后继） | |
|  | |
| 填充（可选） | |
| 块大小（脚部） | a/f |

显示空闲链表结构

|  |  |
| --- | --- |
| 块大小（头部） | a/f |
| 有效载荷  （只包括已分配的块） | |
| 填充（可选） | |
| 块大小（脚部） | a/f |

带边界标签的隐式空闲链表

5）适配块策略：首次适配或下一次适配或最佳适配。首次适配利用率较高；下一次适配时间较快；最佳适配可以很好的减少碎片的产生。我们在分离适配的时候采取的策略一般是首次适配，因为对分离空闲链表的简单首次适配的内存利用效率近似于整个堆的最佳适配的利用效率。

6）值得注意的是：我们的malloc就是采用的是分离适配的方法

## 7.10本章小结

通过本章，我们认识到以下几点事实。

1）虚拟内存是对主存的一个抽象。支持虚拟内存的处理器通过使用一种叫做虚拟内存寻址的间接形式来引用主存。处理器产生一个虚拟地址，在被发送到主存之前，这个地址被翻译成一个物理地址。从虚拟地址空间到物理地址空间的地址翻译要求硬件和软件紧密合作。专门的硬件使用页表来翻译虚拟地址，而页表的内容是由操作系统提供的。

2）虚拟内存提供三个功能：简化了内存保护；简化了内存管理；在主存中自动缓存最近使用的存放在磁盘上的虚拟地址空间的内容。

3）地址翻译的过程必须和系统中的所有的硬件缓存的操作集成在一起。

4）内存映射为共享数据、创建进程以及加载程序提供了一种高效的机制。

5）动态内存分配器直接操作内存，无需类型系统的很多帮助。

# 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**

**文件**（所有的I/O设备都被模型化为文件，甚至内核也被映射为文件）

例如：/dev/sda2文件是用户磁盘分区，而/dev/tty2文件是终端；/boot/vmlinuz-3.13.0-55-generic文件是内核映像，而/proc文件是内核数据结构。

**设备管理**

**unix io接口**

这种将设备优雅地映射为文件的方式，允许Linux内核引出一个简单、低级的应用接口，称为Unix I/O。

我们可以对文件的操作有：打开关闭操作open和close；读写操作read和write；改变当前文件位置lseek等

## 8.2 简述Unix IO接口及其函数

**接口**

**1.打开文件**。一个应用程序通过要求内核打开相应的文件，来宣告它想要访问一个 I/O 设备，内核返回一个小的非负整数，叫做描述符，它在后续对此文件的所有操作中标识这个文件，内核记录有关这个打开文件的所有信息，应用程序只需要记住这个描述符。

**2.linux shell 创建的每个进程开始时都有三个打开的文件：**标准输入（描述符为0) 、标准输出（描述符为1) 和标准错误（描述符为2) 。头文件< unistd.h> 定义了常量STDIN\_FILENO 、STOOUT\_FILENO 和STDERR\_FILENO, 它们可用来代替显式的描述符值。

**3.改变当前的文件位置**：对于每个打开的文件，内核保持着一个文件位 置 k，初始为 0，这个文件位置是从文件开头起始的字节偏移量，应用 程序能够通过执行 seek，显式地将改变当前文件位置 k。

**4.读写文件**。一个读操作就是从文件复制n>0 个字节到内存，从当前文件位置k 开始，然后将k增加到k+n 。给定一个大小为m 字节的文件，当k~m 时执行读操作会触发一个称为end-of-file(EOF) 的条件，应用程序能检测到这个条件。在文件结尾处并没有明确的“EOF 符号” 。类似地，写操作就是从内存复制n>0 个字节到一个文件，从当前文件位置k开始，然后更新k 。

**5.关闭文件**。当应用完成了对文件的访问之后，它就通知内核关闭这个文件。作为响应，内核释放文件打开时创建的数据结构，并将这个描述符恢复到可用的描述符池中。无论一个进程因为何种原因终止时，内核都会关闭所有打开的文件并释放它们的内存资源.

**函数**

**1.打开和关闭文件。**

**打开文件函数原型：**int open(char\* filename,int flags,mode\_t mode)

**返回值**：若成功则为新文件描述符，否则返回-1；

**flags**：O\_RDONLY（只读），O\_WRONLY（只写），O\_RDWR（可读写）

**mode：**指定新文件的访问权限位。

**关闭文件函数原型：**int close(fd)

**返回值：**成功返回0，否则为-1

**2，读和写文件**

**读文件函数原型：**ssize\_t read(int fd,void \*buf,size\_t n**)**

**返回值：**成功则返回读的字节数，若EOF则为0，出错为-1

**描述：**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

**写文件函数原型：**ssize\_t wirte(int fd,const void \*buf,size\_t n)

**返回值：**成功则返回写的字节数，出错则为-1

**描述：**从内存位置 buf 复制至多 n 个字节到描述符为 fd 的当前文件位置

## 8.3 printf的实现分析

**1）首先看printf的函数体。**

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

我们发现函数体内部调用了函数vsprintf，那么我们再继续看一下vsprintf函数。

其中va\_list的定义被定义为字符指针。

**2）再看vsprintf函数（在printf函数内部调用）。**

1. **int** vsprintf(**char** \*buf, **const** **char** \*fmt, **va\_list** args)
2. {
3. **char**\* p;
4. **char** tmp[256];
5. **va\_list** p\_next\_arg = args;
7. **for** (p=buf;\*fmt;fmt++) {
8. **if** (\*fmt != '%') {
9. \*p++ = \*fmt;
10. **continue**;
11. }
13. fmt++;
15. **switch** (\*fmt) {
16. **case** 'x':
17. itoa(tmp, \*((**int**\*)p\_next\_arg));
18. strcpy(p, tmp);
19. p\_next\_arg += 4;
20. p += strlen(tmp);
21. **break**;
22. **case** 's':
23. **break**;
24. **default**:
25. **break**;
26. }
27. }
29. **return** (p - buf);
30. }

函数描述：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。

**3）对于系统函数write**

反汇编追踪write函数

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

发现反汇编语句中的int INT\_VECTOR\_SYS\_CALL，它表示要通过系统来调用sys\_call这个函数。

**4）来看sys\_call函数**

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

函数功能：显示格式化的字符串。将要输出的字符串从总线复制到显卡的显存中。

**5）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）。

**6）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）。而我们要传输的“hello 1172510217 张景润”就会被打印输出在显示器上。

## 8.4 getchar的实现分析

**1）**运行到getchar函数时，程序将控制权交给os。当你键入时，内容进入缓寸并在屏幕上回显。按enter，通知 os输入完成，这时再将控制权在交还给程序。

**2）**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。

**3）**getchar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

在本章，我们接触了文件的操作。

**1）Linux提供了少量的基于unix I/O模型的系统级函数。**他们允许应用程序打开（open），关闭（close），读（read），写（write）文件，提取文件的元数据，以及执行I/O的重定向。

**2）printf函数小家伙，大本事。**看似简单的printf函数其实底层实现非常复杂，他调用了函数vsprintf和系统调用write，而之后有调用了sys\_call函数。

**3）getchar函数的实现也是关于中断的处理。**同时他也进行了系统调用write函数。

# 结论

**1）hello虽小，五脏俱全**

**1，hello.c经过预编译，拓展得到hello.i文本文件**

**2，hello.i经过编译，得到汇编代码hello.s汇编文件**

**3，hello.s经过汇编，得到二进制可重定位目标文件hello.o**

**4，hello.o经过链接，生成了可执行文件hello**

**5，bash进程调用fork函数，生成子进程；并由execve函数加载运行当前进程的上下文中加载并运行新程序hello**

**6，hello的变化过程中，会有各种地址，但最终我们真正期待的是PA物理地址。**

**7，hello再运行时会调用一些函数，比如printf函数，这些函数与linux I/O的设备模拟化密切相关**

**8，hello最终被shell父进程回收，内核会收回为其创建的所有信息**

**2）感悟**

1，计算机系统学习过程感觉是：由厚到更厚到更更厚到薄。遗憾的是，我作为本论文的编者与负责人，水平有限，不能够将涉及到的知识简明扼要的表述出来。还要多打磨学习

2，计算机系统学问很深。尤其是当我看到一个简简单单的hello的printf函数就包含了如此多的知识后，更加加深了我对计算机系统的崇拜。

# 附件

|  |  |  |
| --- | --- | --- |
| 中间结果文件 | 文件作用 | 使用时期 |
| hello.i | 预处理得到的文件  ASCII码的中间文件 | 第二章-预处理 |
| hello.s | ASCII汇编语言文件 | 第三章-编译 |
| hello.o | as得到可重定位目标文件 | 第四章-汇编 |
| hello.asm | 反汇编得到的文本文件 | 第四章-汇编 |
| hello.elf | hello.o的elf文件 | 第四章-汇编 |
| hello | ld得到可执行目标文件 | 第五章-链接 |
| hello\_elf | hello的elf文件 | 第五章-链接 |
| hello\_asm | hello的反汇编文件 | 第五章-链接 |

# 参考文献

**为完成本次大作业你翻阅的书籍与网站等**

[1] 心不留意外尘. 360个人图书馆. 2016：04-14. Linux 后台开发常用调试工具 http://www.360doc.com/content/16/0414/16/478627\_550597152.shtml

[2] 酷勤网. C语言预处理命令之条件编译. 2009：08-16. http://www.kuqin.com/language/20090806/66164.html

[3] 乐于其中. CSDN. 编译器工作流程详解. 2014：04-27. https://blog.csdn.net/u012491514/article/details/24590467

[4] 网络用户. 阿里云. ELF格式文件符号表全解析及readelf命令使用方法. 2018：07-19. https://www.aliyun.com/zixun/wenji/1246586.html

[5] ORACLE.链接程序和库指南. https://docs.oracle.com/cd/E38902\_01/html/E38861/chapter6-54839.html#gentextid-15180

[6]冰凌块儿.动态链接过程之重定位. http://blog.chinaunix.net/uid-24669930-id-4294759.html

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

[8]Randal E. Bryant, David R. O'Hallaon. 深入理解计算机系统. 第三版. 北京市：机械工业出版社[M]. 2018： 1-737