**计算机系统基础**

**结业论文**

题 目 hello.c程序的一生

专 业 软件工程

学　　 号 201831043133

班　　 级 1802

姓 名 张家林

**计算机科学学院**

**2020年6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 - 3 -](#_Toc42249187)

[1.1 Hello简介 - 3 -](#_Toc42249188)

[1.2 环境与工具 - 3 -](#_Toc42249189)

[1.3 中间结果 - 3 -](#_Toc42249190)

[1.4 本章小结 - 4 -](#_Toc42249191)

[第2章 预处理 - 5 -](#_Toc42249192)

[2.1 预处理的概念与作用 - 5 -](#_Toc42249193)

[2.2在Ubuntu下预处理的命令 - 5 -](#_Toc42249194)

[2.3 Hello的预处理结果解析 - 6 -](#_Toc42249195)

[2.4 本章小结 - 10 -](#_Toc42249196)

[第3章 编译 - 11 -](#_Toc42249197)

[3.1 编译的概念与作用 - 11 -](#_Toc42249198)

[3.2 在Ubuntu下编译的命令 - 11 -](#_Toc42249199)

[3.3 Hello的编译结果解析 - 12 -](#_Toc42249200)

[3.4 本章小结 - 26 -](#_Toc42249201)

[第4章 汇编 - 27 -](#_Toc42249202)

[4.1 汇编的概念与作用 - 27 -](#_Toc42249203)

[4.2 在Ubuntu下汇编的命令 - 27 -](#_Toc42249204)

[4.3 Hello.o的结果解析 - 28 -](#_Toc42249205)

[4.5 本章小结 - 30 -](#_Toc42249206)

[第5章 链接 - 30 -](#_Toc42249207)

[5.1 链接的概念与作用 - 30 -](#_Toc42249208)

[5.2 在Ubuntu下链接的命令 - 30 -](#_Toc42249209)

[5.3 链接的重定位过程分析 - 31 -](#_Toc42249210)

[5.4 hello的执行流程 - 34 -](#_Toc42249211)

[5.8 本章小结 - 35 -](#_Toc42249212)

# 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。

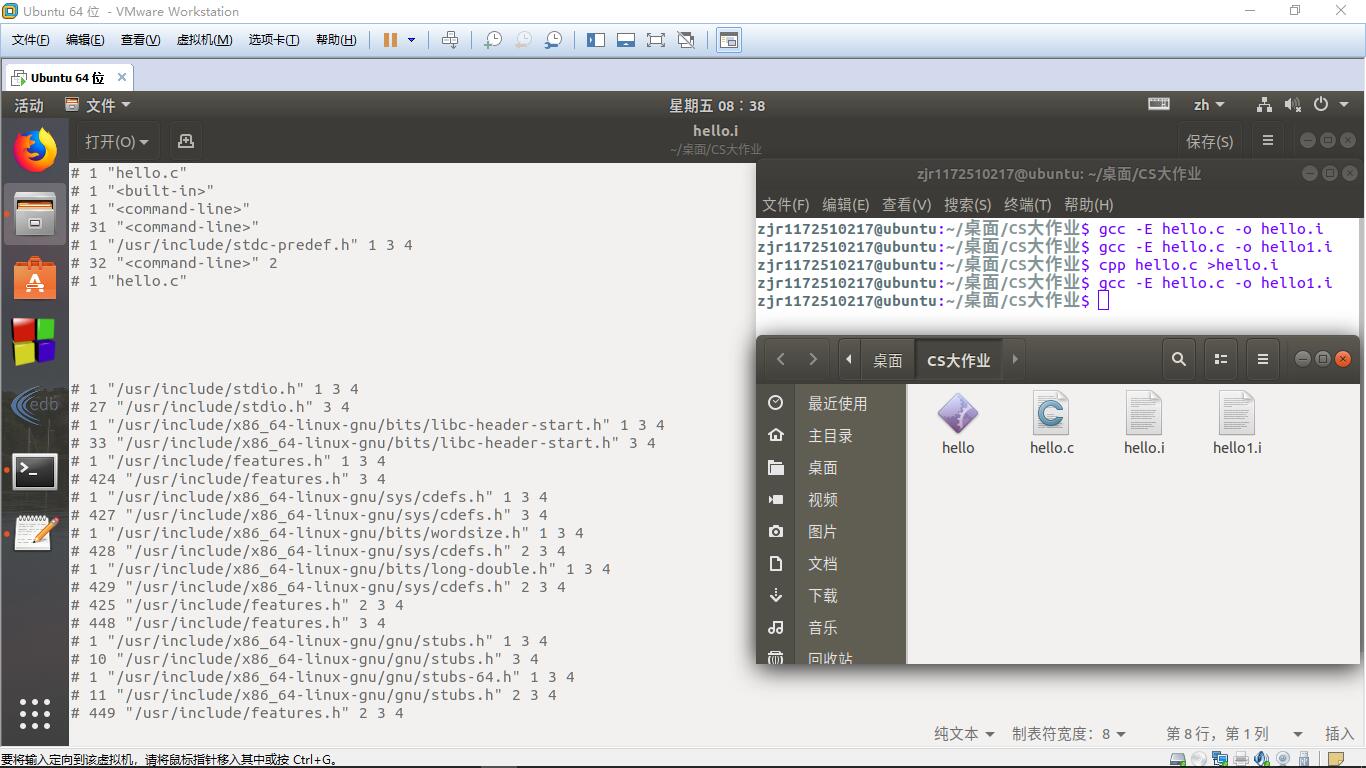
作用主要与三部分有关：宏定义，文件包含，条件编译

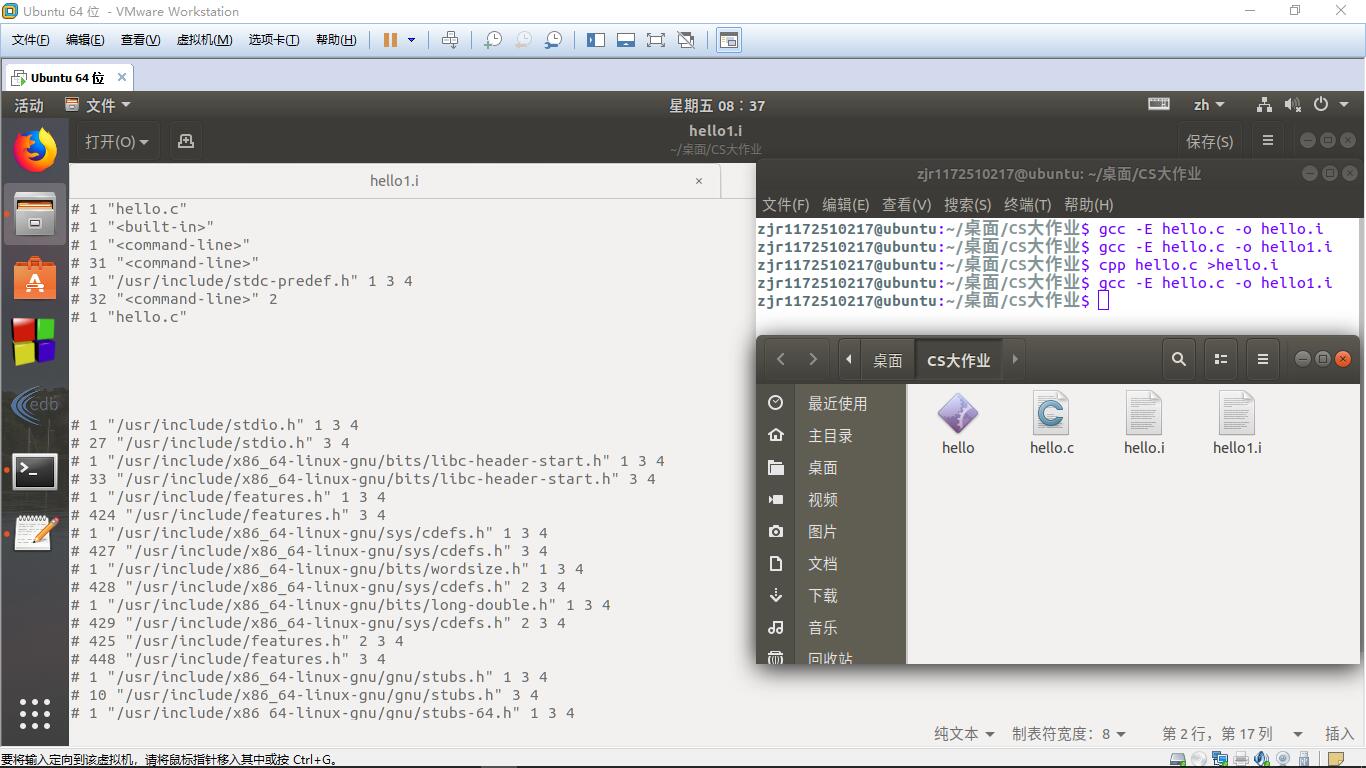
1. 宏定义相关。预处理程序中的#define 标识符文本，预处理工作也叫做宏展开：将宏名替换为文本（这个文本可以是字符串、可以是代码等）。
2. 文件包含相关。预处理程序中的#include，将头文件的内容插入到该命令所在的位置，从而把头文件和当前源文件连接成一个源文件，这与复制粘贴的效果相同。
3. 条件编译相关。根据#if以及#endif和#ifdef以及#ifndef来判断执行编译的条件。

## 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**

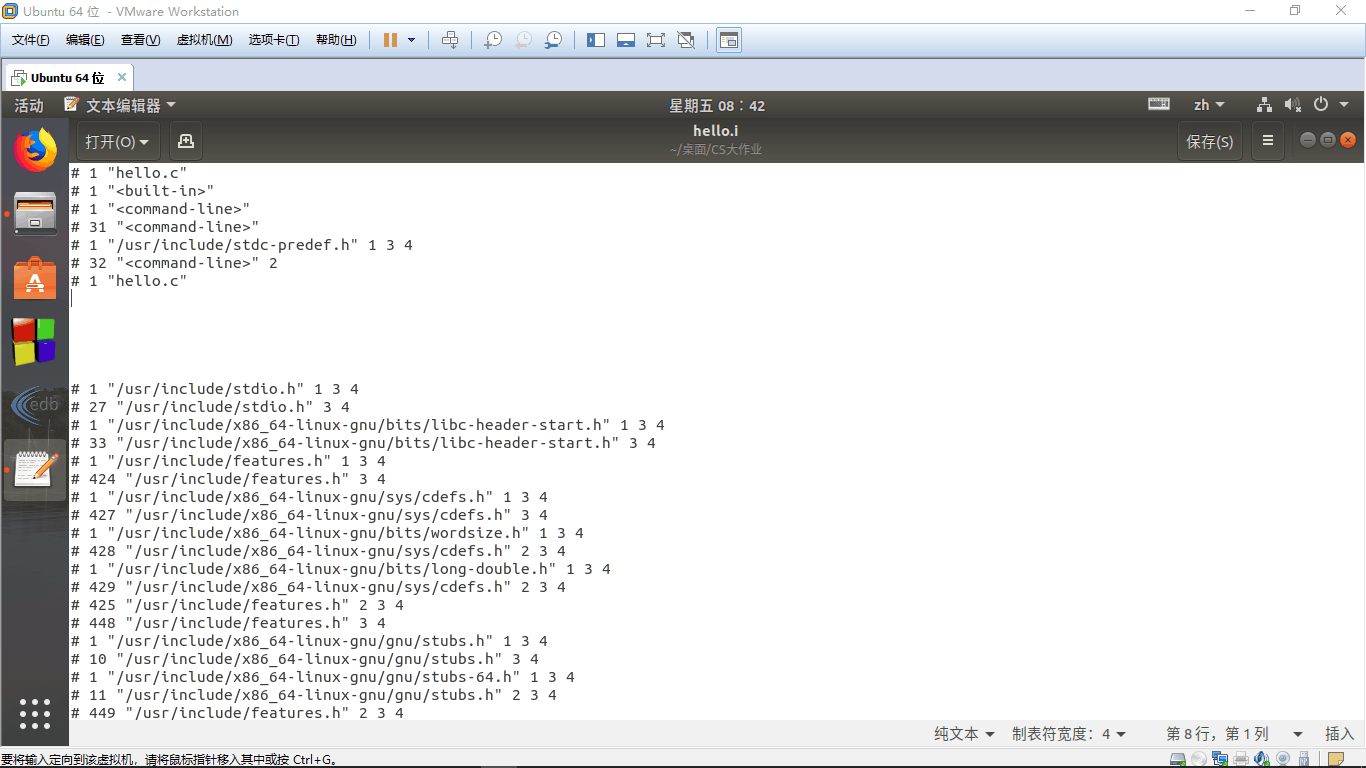


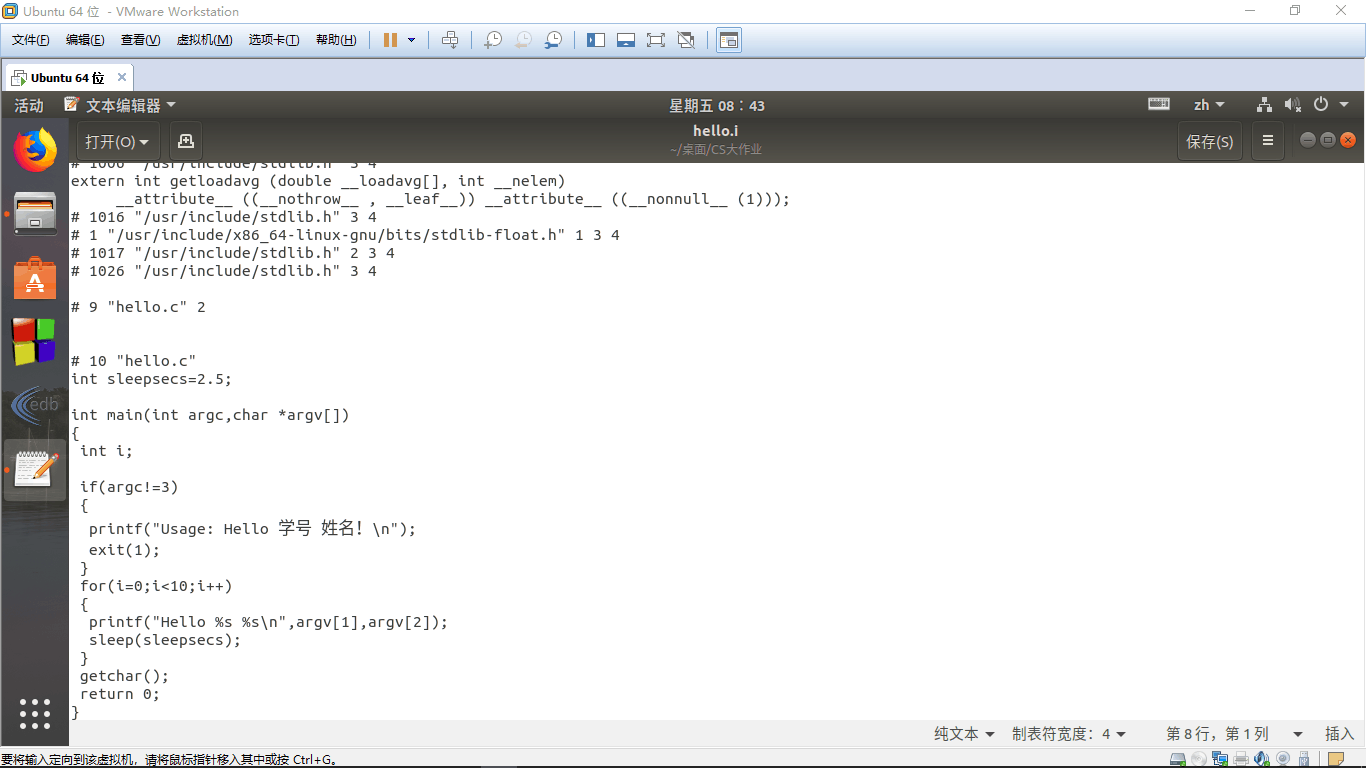
命令1：cpp hell.c >hello.i输出预编译文本文件

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）。

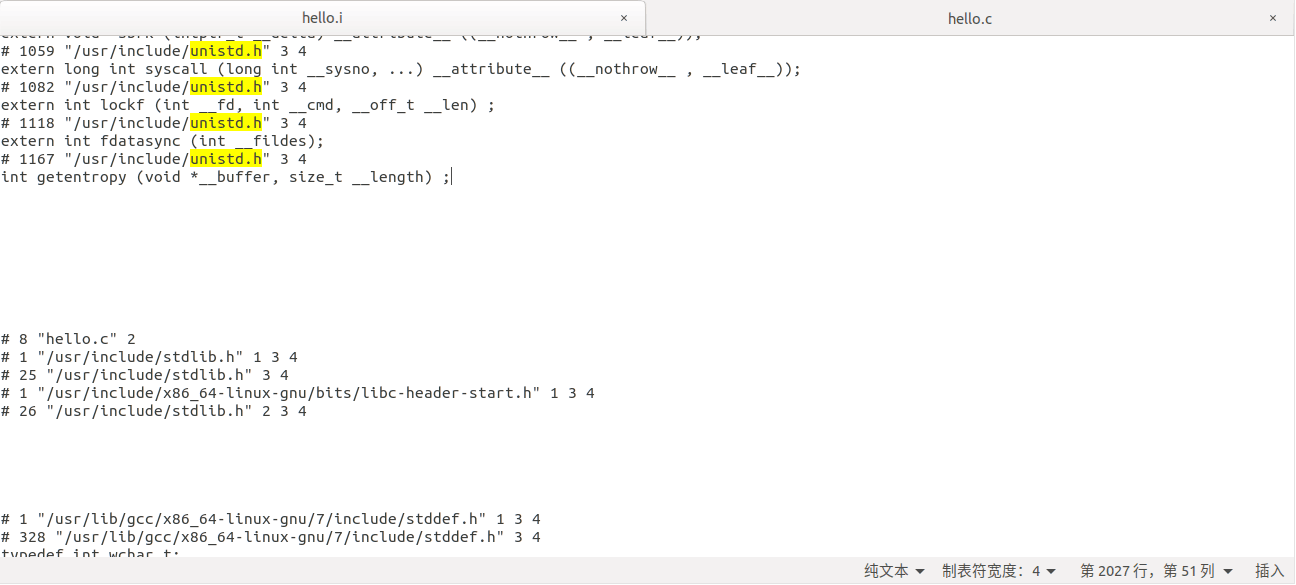
关于文件包含（头文件）在预处理阶段的处理方法。当使用尖括号括起来头文件时，表示在系统头文件目录中查找（由用户在设置编程环境时设置），而不在源文件目录中查找；当使用双引号括起来头文件，则表示首先在当前的源文件目录中查找，找不到再到系统头文件目录中查找。比如以我们的hello.c的头文件处理为例，cpp到默认的环境变量下寻找相应的头文件。具体到三个头文件，寻找路径分别是：usr/inlcude/stdio.h，usr/include/unistd.h和usr/inlcude/stdlib.h（如截图2.2.4，标红加注）

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。

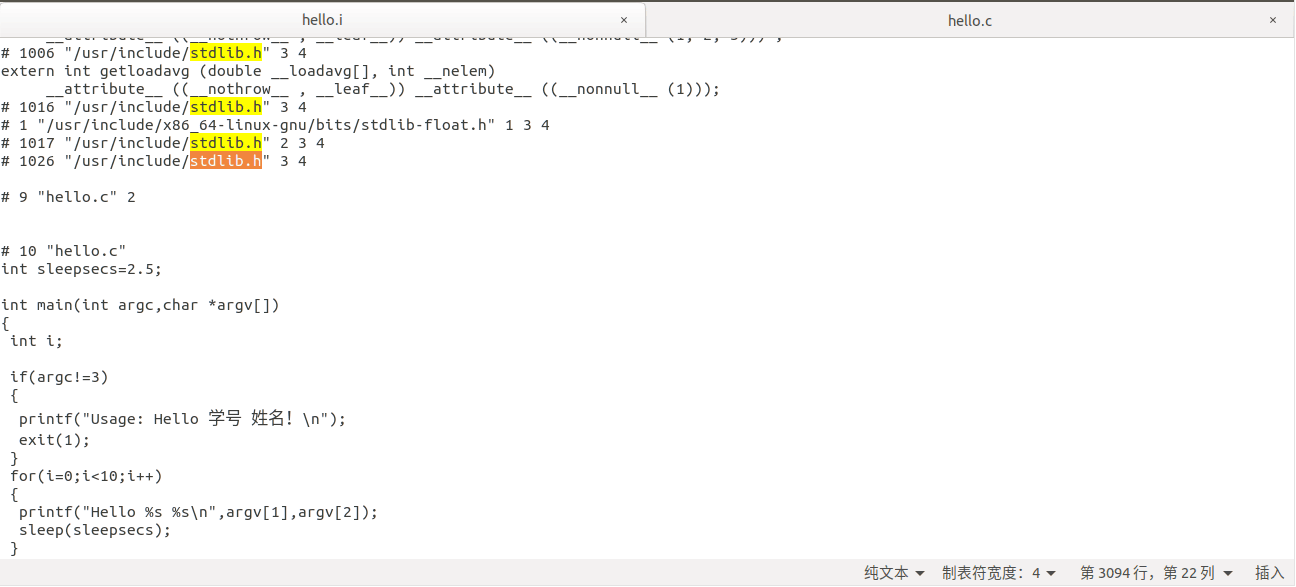
而我们原来的程序主体段开始于第3099行，结束于3118行，预处理删除了我们的注释信息（如截图2.2.5）。除了注释部分以及头文件部分，预编译文件与源文件无太大差别。



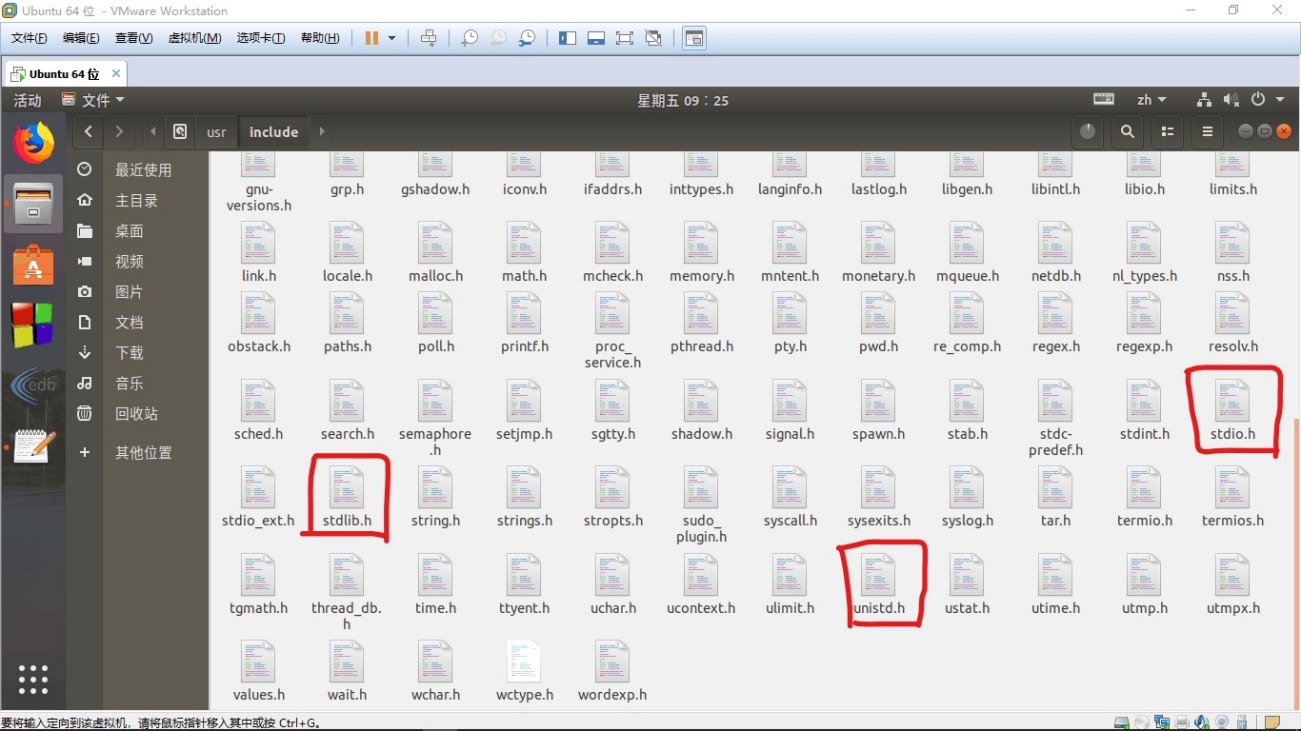
截图2.3.1，stdio.h结尾处



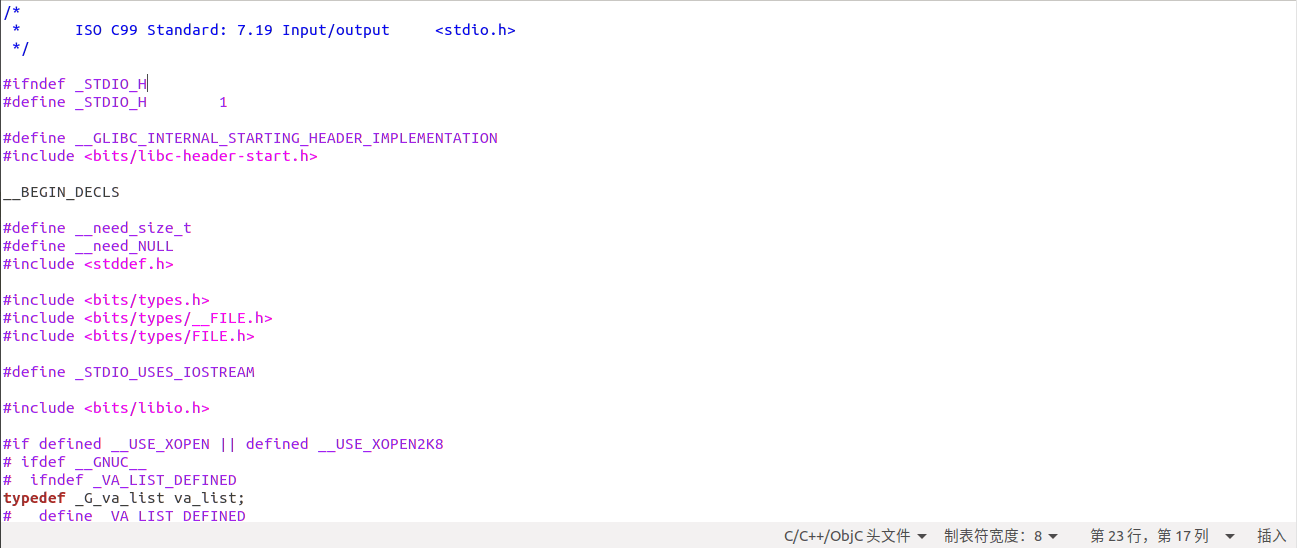
截图2.3.2，unistd.h结尾处



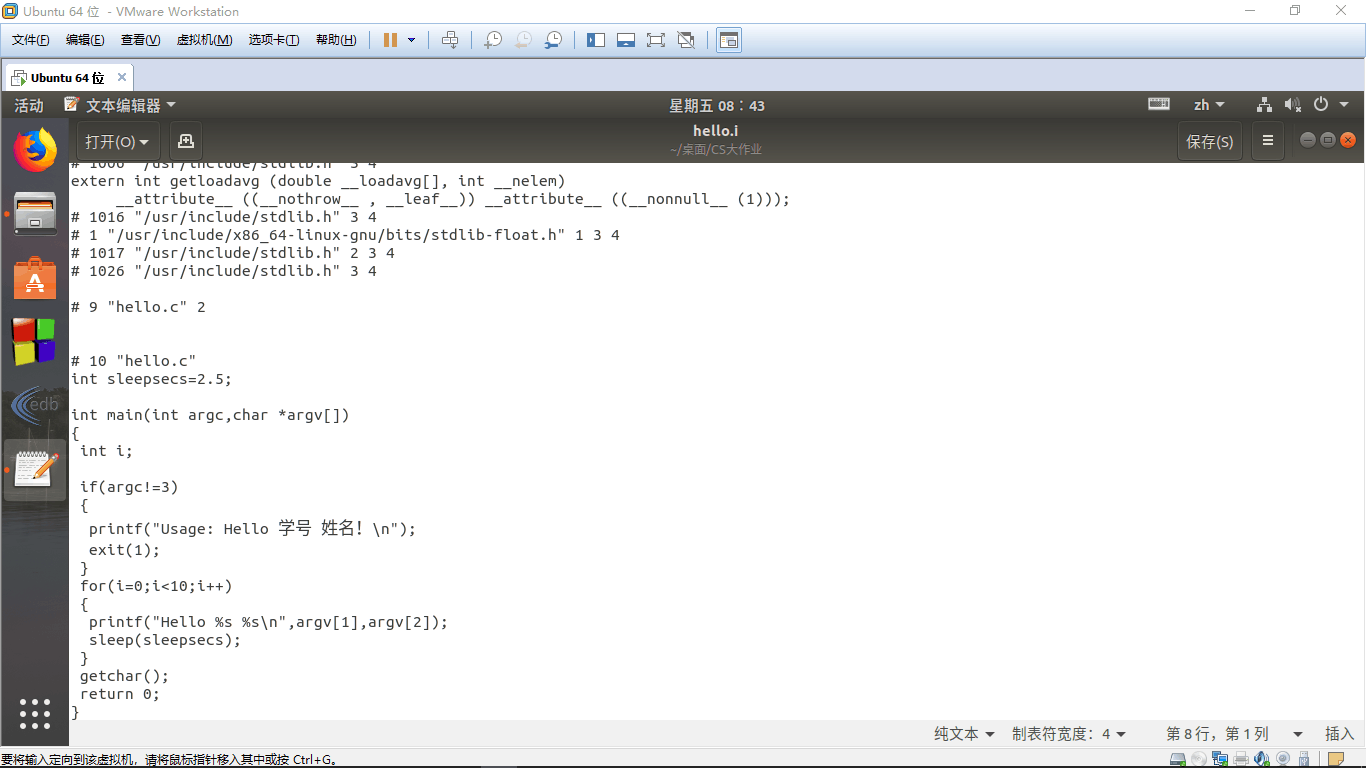
截图2.3.3，stdlib.h结尾处



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。

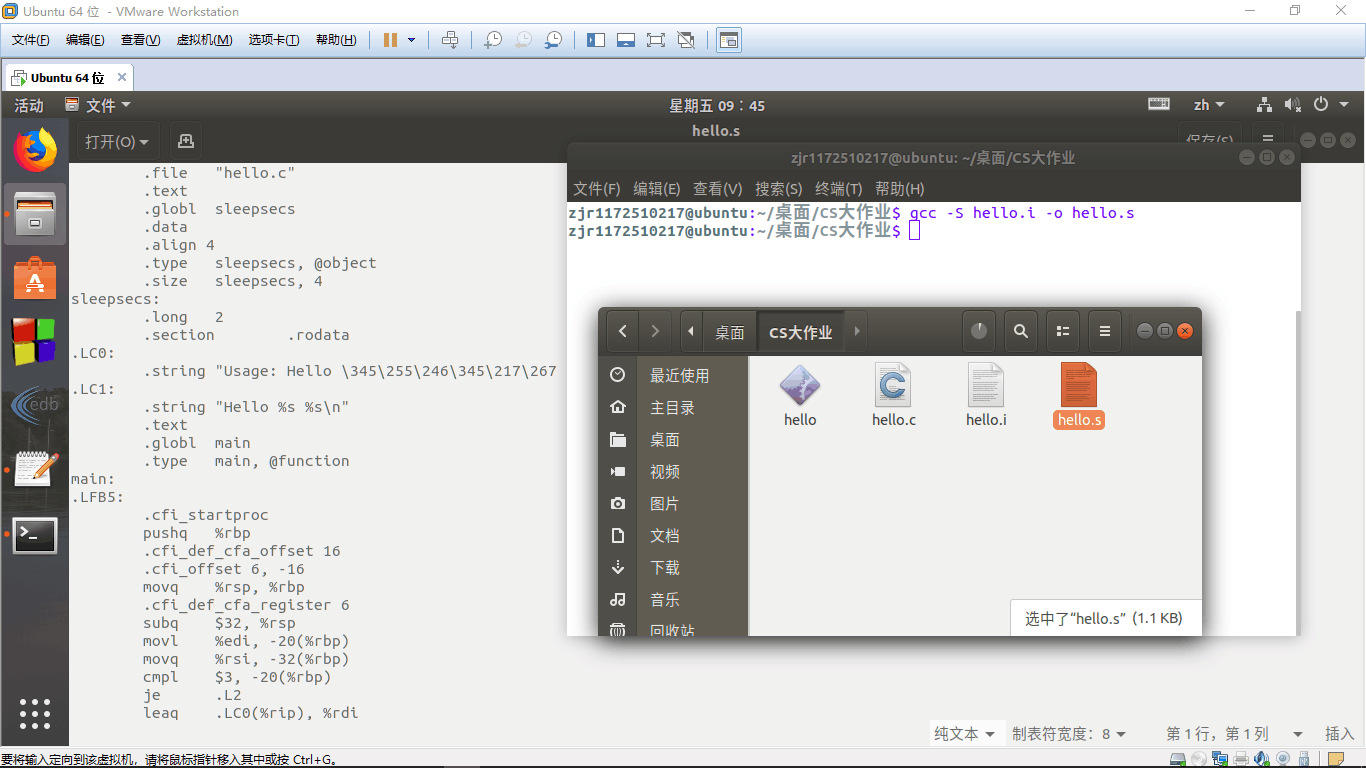
编译的作用

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++）。

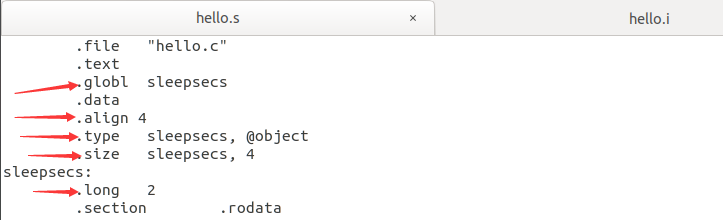
数据类型有整型，数组，字符串。

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）

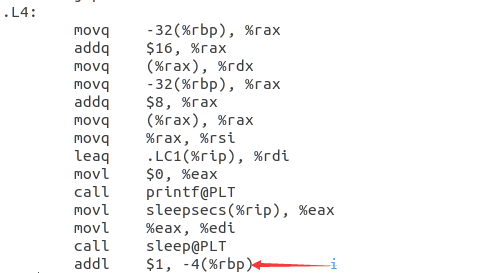
1）对于int sleepsecs。sleepsecs是已经被初始化的全局变量（初始化语句为int sleepsecs=2.5;）由于.data 节存放已经初始化的全局和静态C变量，编译器首先将sleepsecs在.text代码段中声明为全局变量；其次在.data 段中，设置对齐方式（.align）为4字节对齐，设置类型（.type）为对象，设置大小（.size）为4字节，设置为long类型（.long），其值为2（如截图3.3.2-1）。

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。



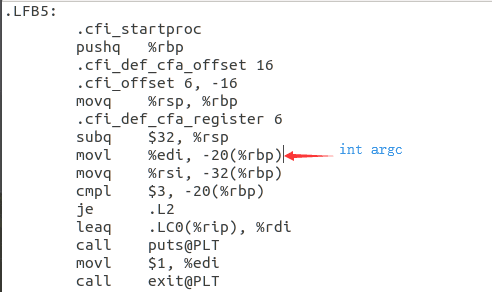
截图3.2.2-1，int sleepsecs分析

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。



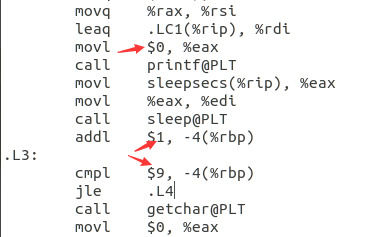
截图3.3.2-2，int i分析

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。



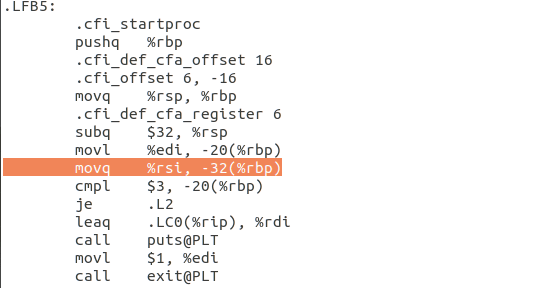
截图3.3.2-4，立即数分析

3.3.3 数据类型之数组

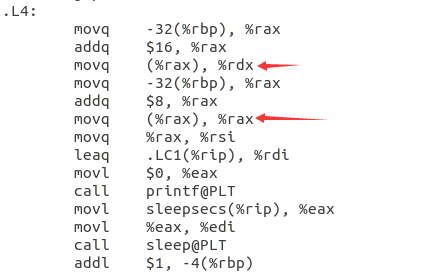
对于数组char \*argv[]。它是我们main函数的第二个形式参数，来源于我们终端键入的数据（我键入的是：./hello 1172510217 张景润）。argv同时作为存放char指针的数组。

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



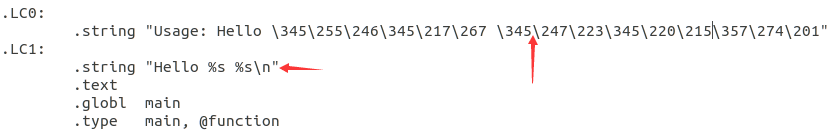
截图3.3.3，int argc[]数组内容使用

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。



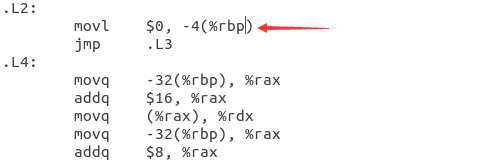
截图3.3.4-1

3.3.5 赋值操作

源程序中的赋值操作有int sleepsecs=2.5; i=0;i++;

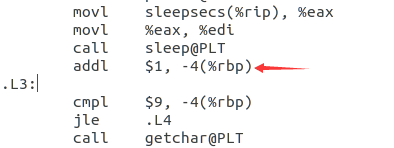
1）对于第一个赋值操作int sleepsecs = 2.5。前面大致已经对此进行分析，sleepsecs是全局变量，而且在这里进行了赋初值操作，因此直接在.data节中将sleepsecs 声明为值为2的long类型数据（隐式转换，编译器缺省）。

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

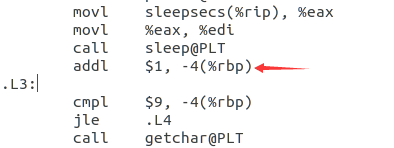
3.3.7 算术操作+逻辑操作

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 |

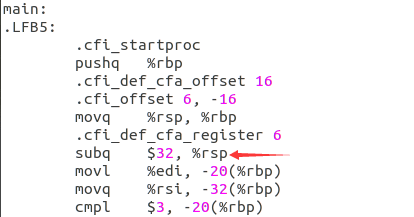
要注意的是，上表的操作顺序与ATT格式的汇编代码中的相反。（同时还有一些特殊的算术操作未被列出，比如有符号和无符号数的全128位乘法与除法）

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）。



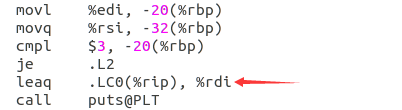
截图3.3.7-1，i++操作

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。



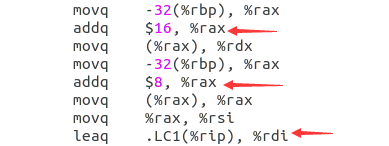
截图3.3.7-2，减法操作

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。

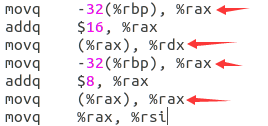
对于具体操作，cmpl $9, -4(%rbp)计算-4(%rbp)-9，并设置条件码，随之jle语句利用这些条件码，进行相应的跳转处理。

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。



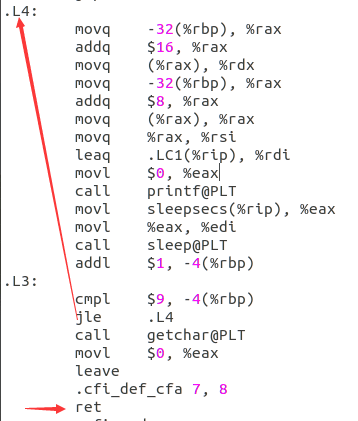
截图3.3.10-1，比较字符串数和3的大小

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。

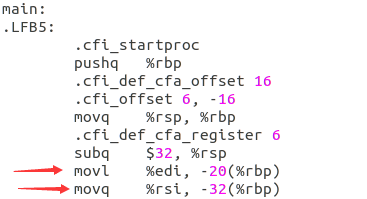
当调用一个过程时,除了要把控制传递给它并在过程返回时再传递回来之外,过程调用还可能包括把数据作为参数传递,而从过程返回还有可能包括返回一个值。

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）。

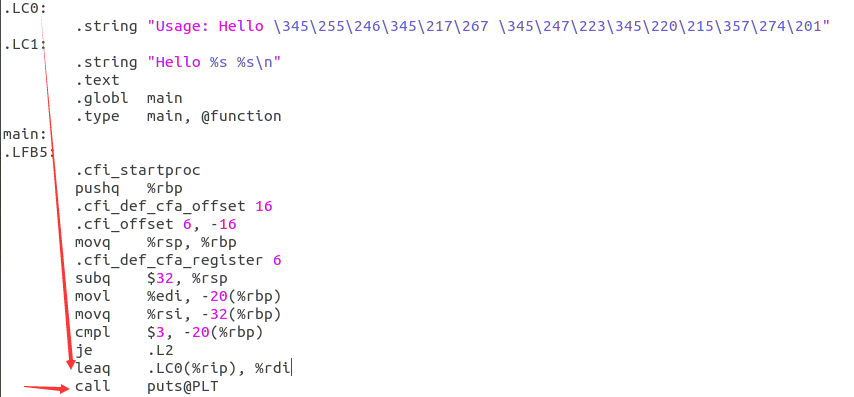
即过程中的传递数据，P必须能够向Q提供一个或多个参数。

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

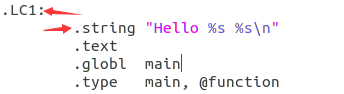
对于printf函数。printf函数在具体的汇编代码中被优化为puts函数。我们发现第一次在汇编代码中（截图1）首先将rdi赋值为字符串“Usage: Hello 学号 姓名！ \n”字符串的首地址（leaq .LC0(%rip), %rdi）。然后调用了puts函数，即将第一处字符串参数传入。对于第二处，类似的，我们发现在汇编代码中（截图2和截图3）首先将rdi赋值为字符串"Hello %s %s\n"的首地址。这里没有被优化为puts函数，而是直接调用printf函数。同时设置%rsi argv[1]，%rdx 为argv[2]。这样就可以根据控制字符串，直接输出终端键入的命令行。



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

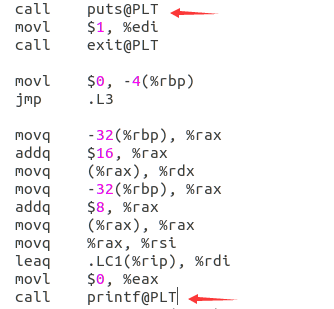
对于exit函数（如截图）。通过汇编语句movl $1, %edi将%edi寄存器内容设置为1。

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。

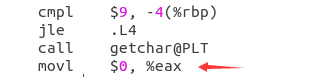
对于printf函数。在main函数内部，通过汇编语句call puts@PLT调用（第一次），通过汇编语句call printf@PLT调用（第二次）。附上截图



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）。

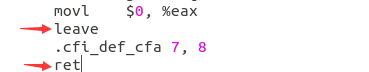


对于exit函数。在main函数内部被调用，调用它的汇编语句是call exit@PLT（附上截图）。

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。

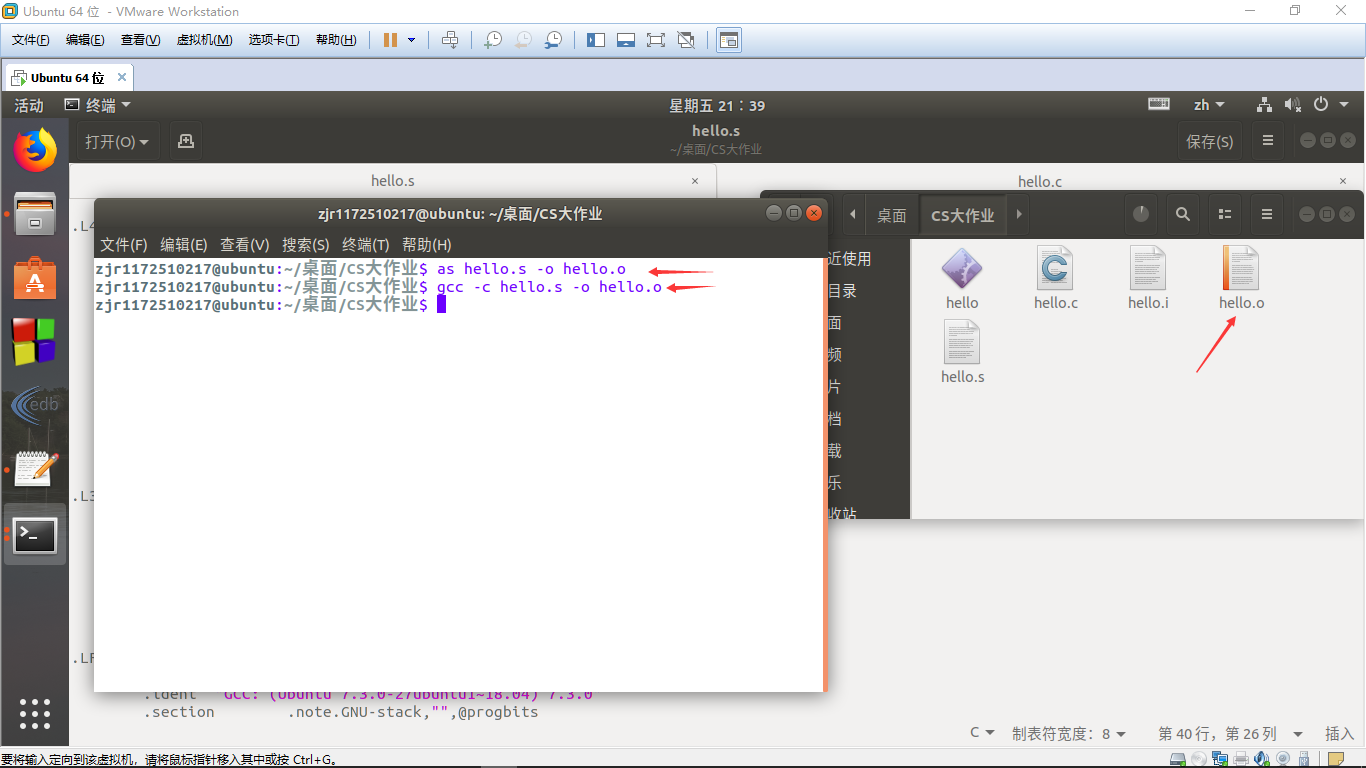
## 4.2 在Ubuntu下汇编的命令

汇编的命令

as hello.s -o hello.o或者

gcc -o hello.s -o hello.o

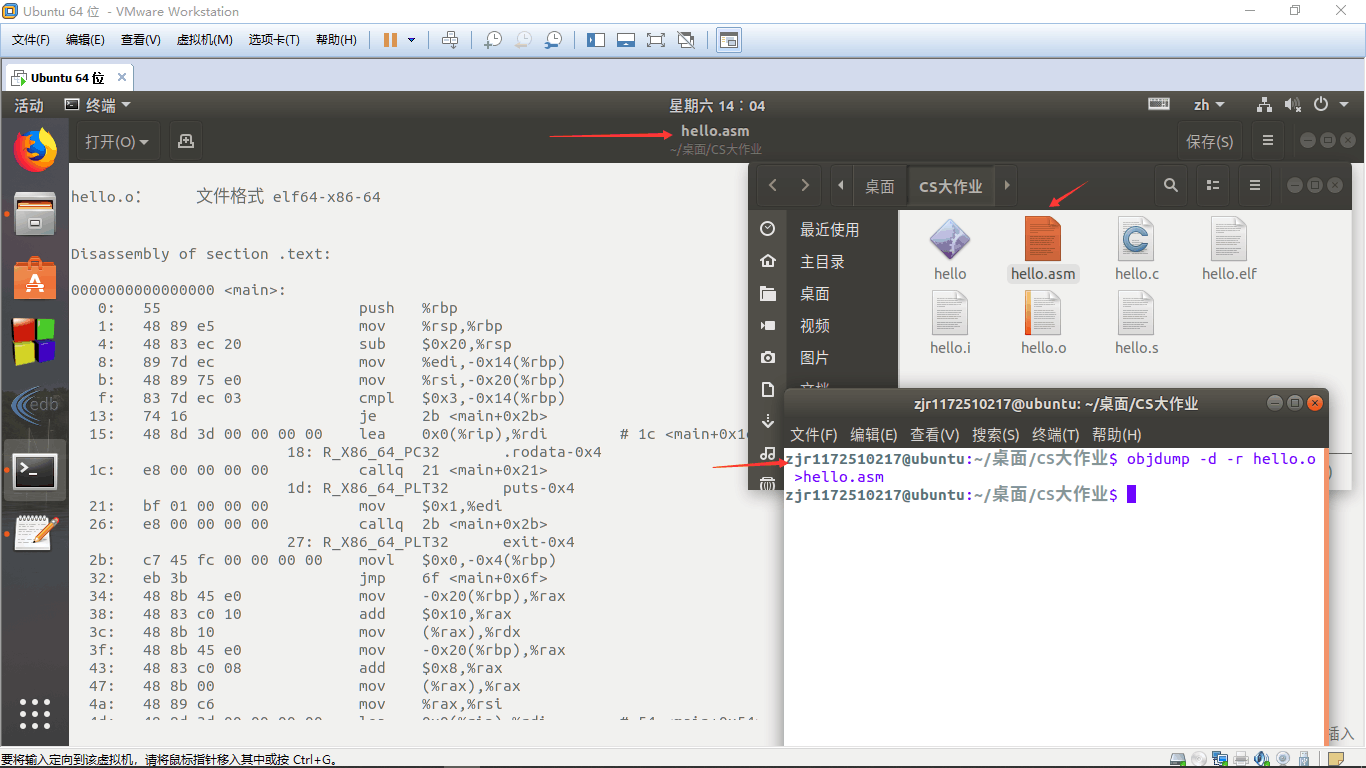
附上截图





## 4.3 Hello.o的结果解析

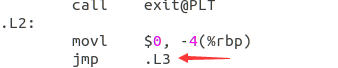
* **命令行输入：objdump -d -r hello.o >hello.asm**（附上截图）



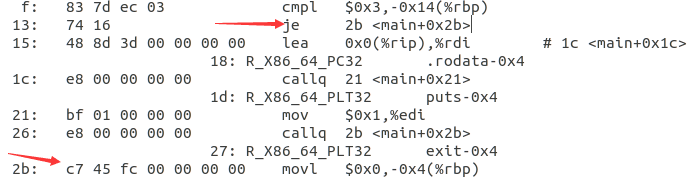
**截图：objdump -d -r hello.o >hello.asm**

* **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）。

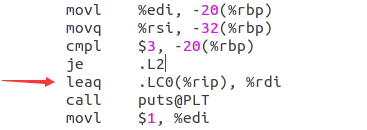
TW9P)F`86`)RKY1H(2B{(E9

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。

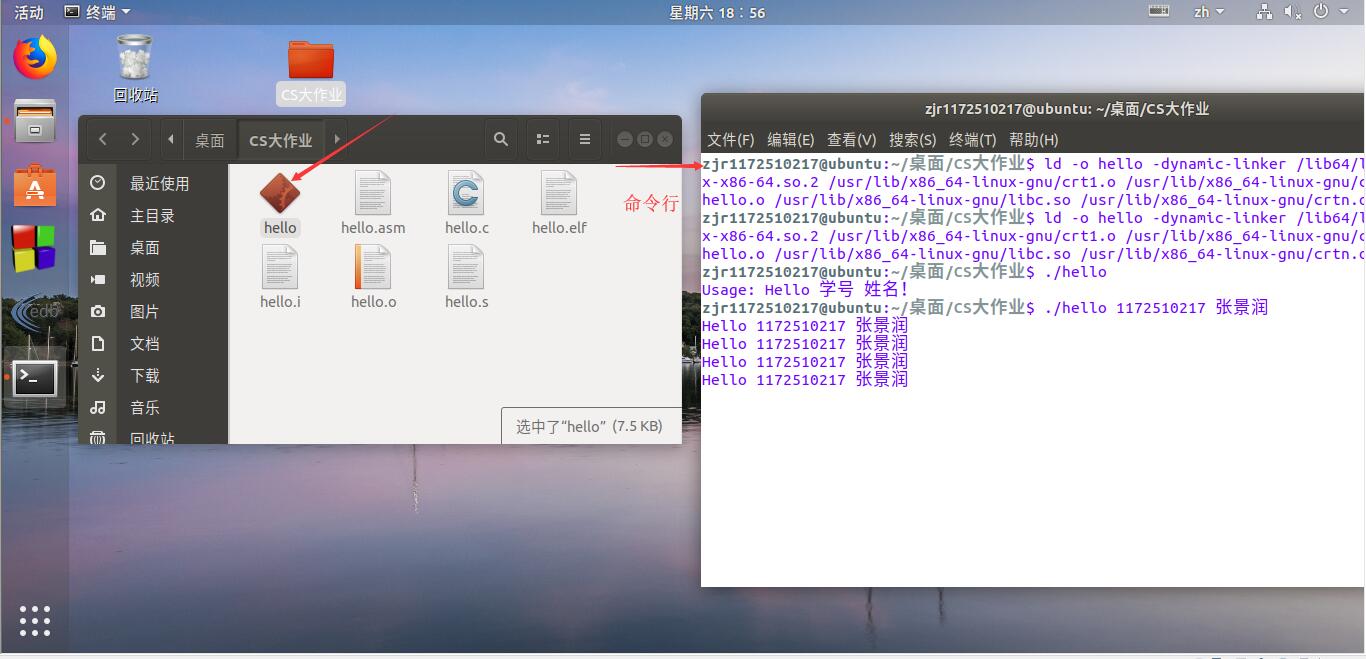
## 5.2 在Ubuntu下链接的命令

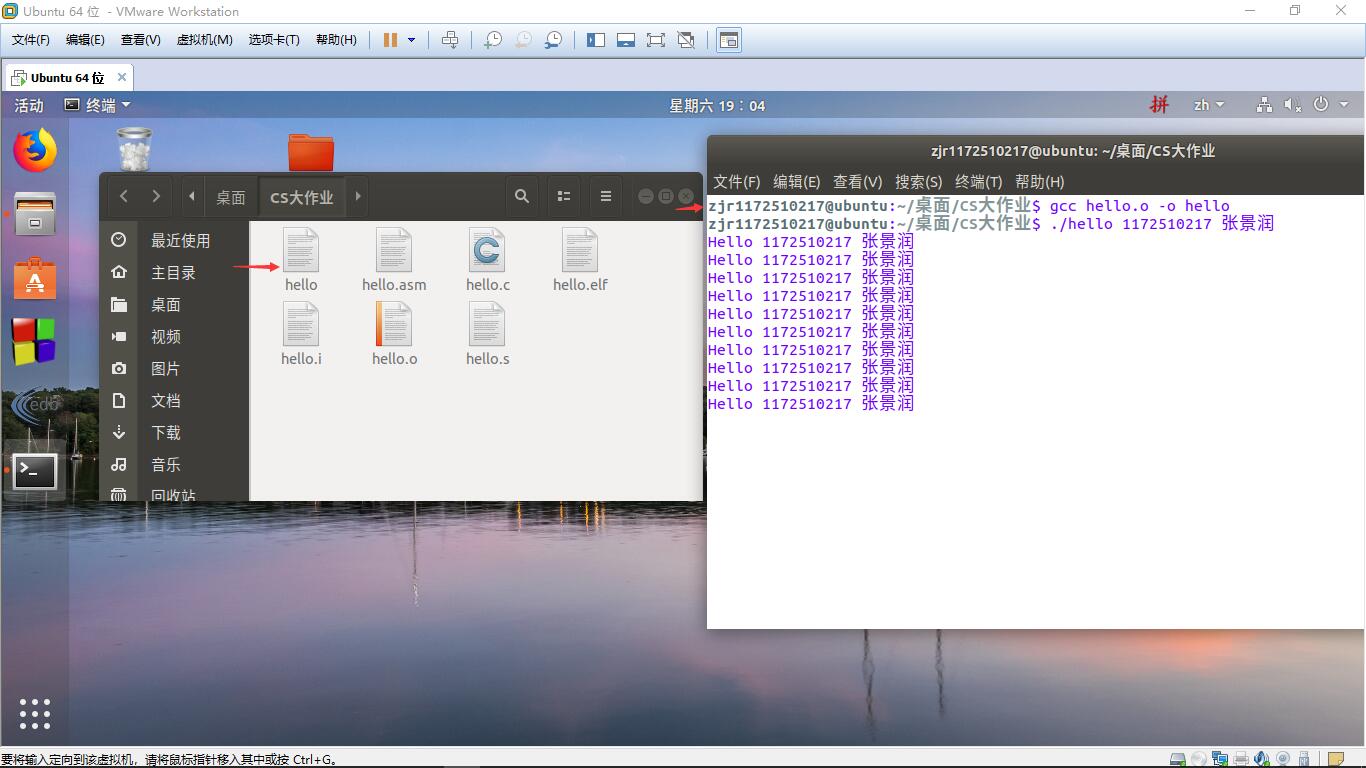
* **命令行**

**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

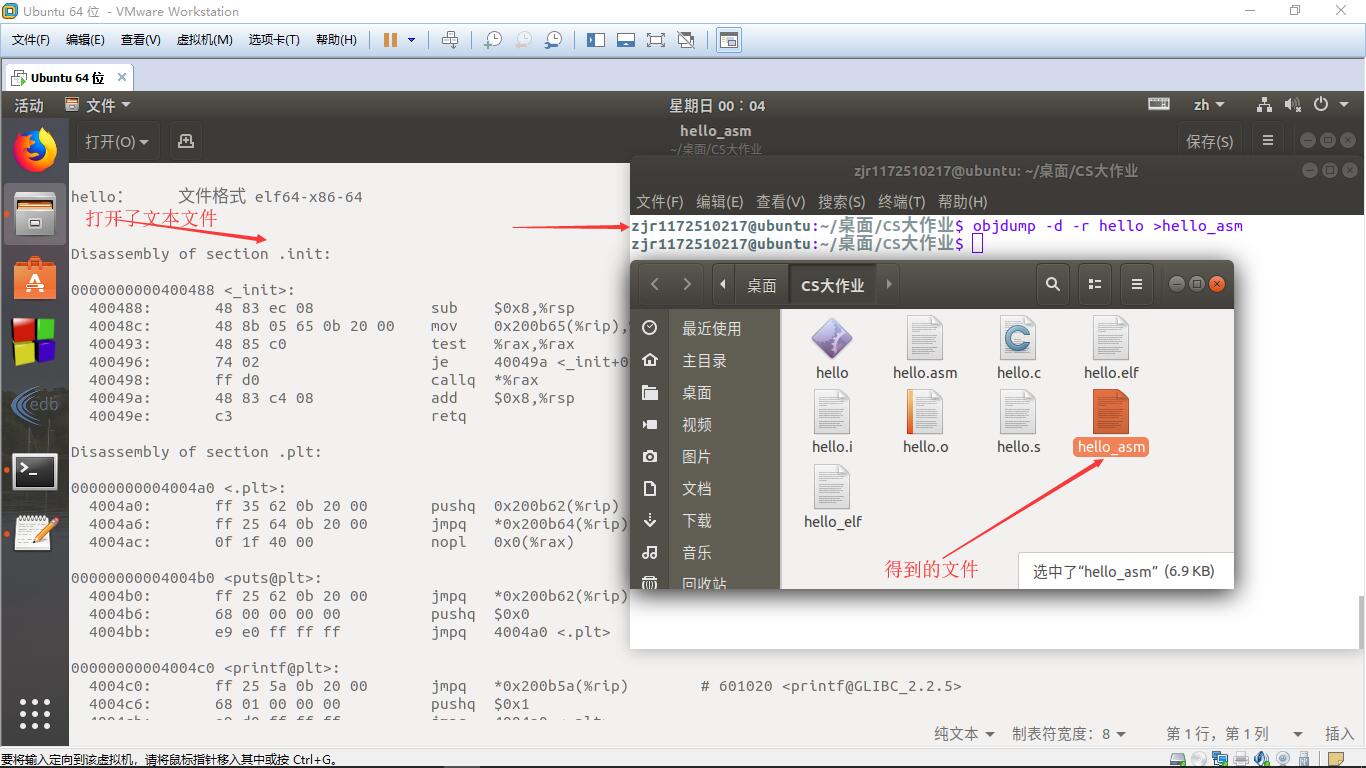


方法1截图

方法2截图

## 5.3 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

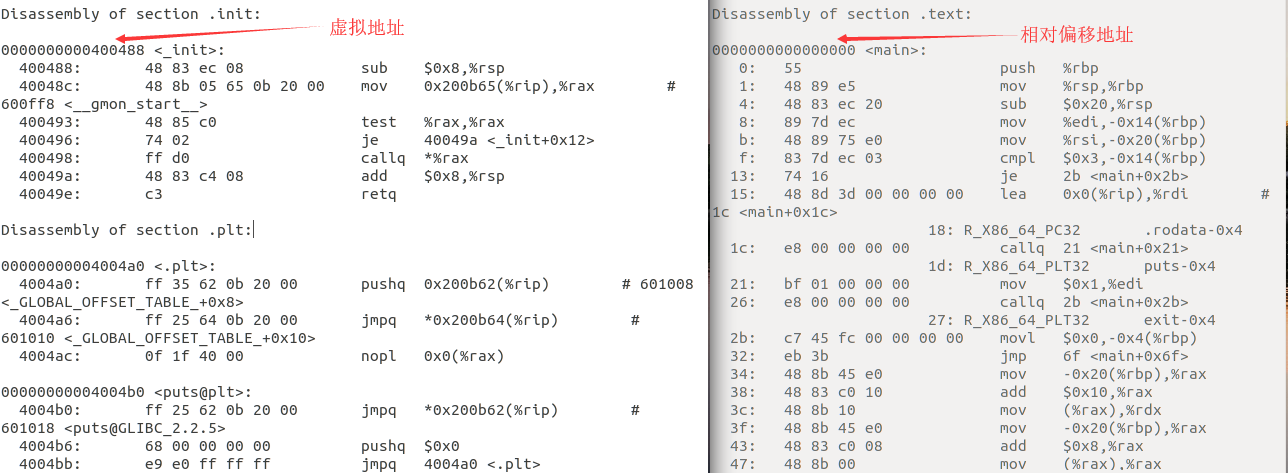
* 第2步，分析列举hello反汇编文件与hello.o反汇编文件的区别（即hello.asm与hello\_asm的对比）。

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。



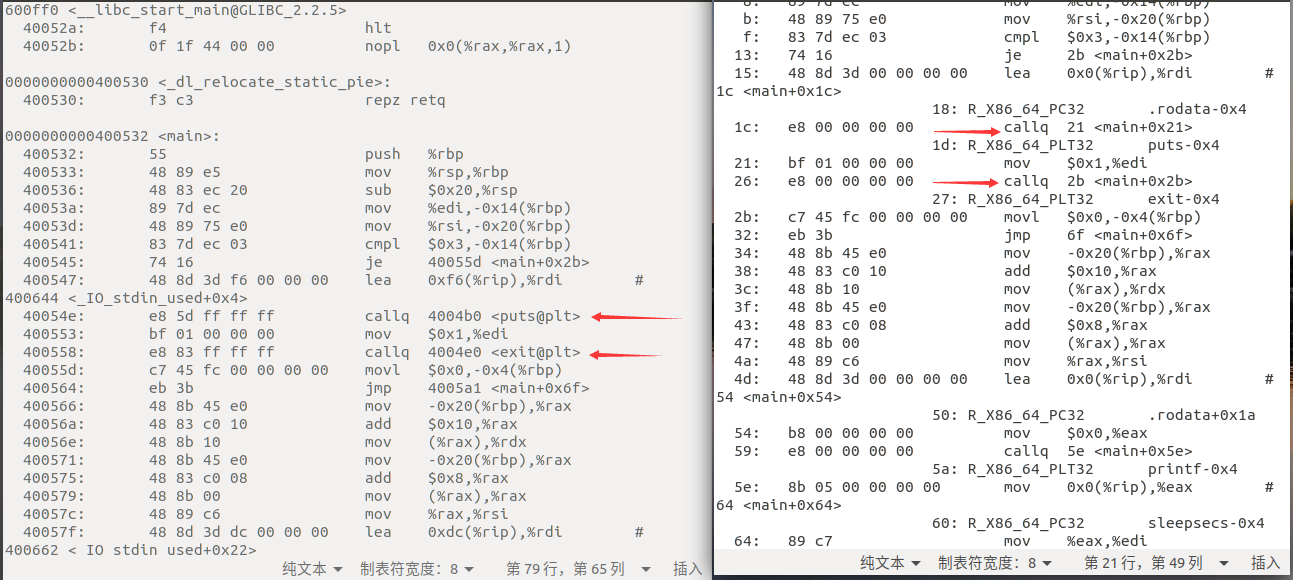
截图3，hello\_asm中各节的地址换为了虚拟地址

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。

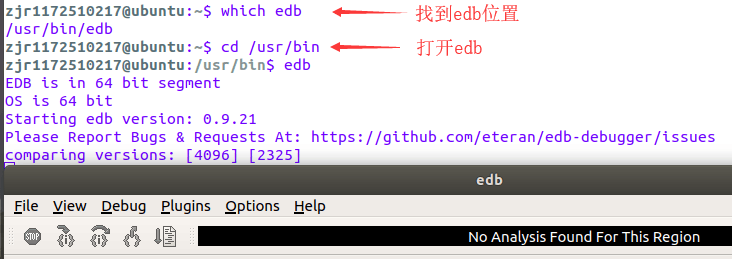


截图5，hello\_asm中函数调用使用虚拟地址

* 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。

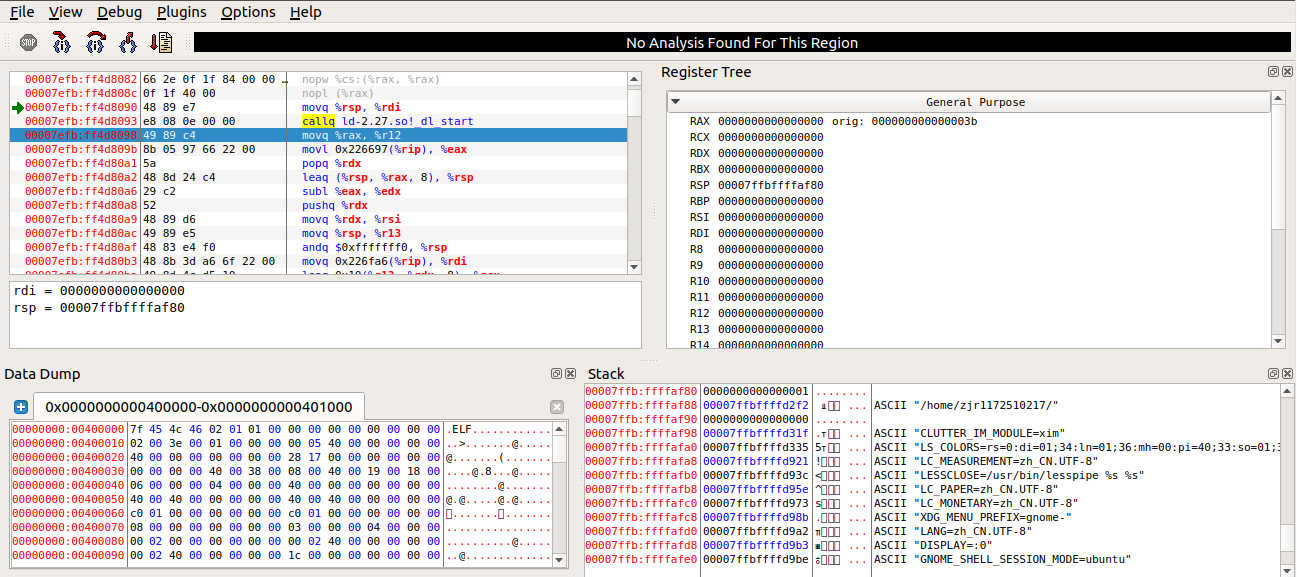
## 5.4 hello的执行流程

* **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**

## 5.8 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。

**参考文献**

[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