

**计算机系统基础**

**大作业**

题 目 程序人生-Hello’s P2P

专 业 测控技术与仪器

学　　 号 1160100327

班　　 级 1601101

学 生 谢宇轩

指 导 教 师 史先俊

**计算机科学与技术学院**

**2019年3月**

**摘 要**

本论文将CSAPP课程的内容通过对hello小程序的一生经历，对我们所学的内容进行梳理与回顾，加深对CSAPP课程的理解。在Ubutun下进行操作，学习各种工具的使用，建立在ubutun下开发的能力。

**关键词：**Ubutun；hello；

**（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 - 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

根据Hello的自白，利用计算机系统的术语，简述Hello的P2P，020的整个过程。

P2P：Program是我们写的。我们在命令行中输入gcc hello.c -o hello. 生成可执行文件，这一个操作，实际上经历了图1中的四个步骤。



图 1 程序编译过程

在预编译截断，主要处理文件中的#ifdef、#include和#define，生成一个中间文件\*.i。在编译过程中，将代码翻译成汇编语言，生成\*.s文件。在汇编阶段，将汇编指令变成二进制的机器代码，生成\*.o文件。链接是编译的最后一步，将各个目标文件链接起来生成可执行文件。用户在shell中用./hello启动程序，shell调用fork产生子进程，hello便成为了进程。

020：程序未运行时，程序存储在硬盘中，OS调用fork函数产生子进程，调用execve，并进行虚拟内存映射（mmp），为hello进程分配时间片，执行程序，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

列出你为编写本论文，折腾Hello的整个过程中，使用的软硬件环境，以及开发与调试工具。

硬件工具：Intel-i5 CPU，8GRAM，256SSD.

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

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

## 1.3 中间结果

列出你为编写本论文，生成的中间结果文件的名字，文件的作用等。

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

## 1.4 本章小结

第一章首先介绍了hello的P2P与O2O过程，简要地讲述了hello从出生到结束。然后列举了对hello分析操作所需要的工具以及化境，最后以一个表格的形式对实验过程中用到的所有中间文件进行了总结。

**（第1章0.5分）**

# 第2章 预处理

## 2.1 预处理的概念与作用

**预处理的概念**

预处理是指在编译之前进行的处理，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。

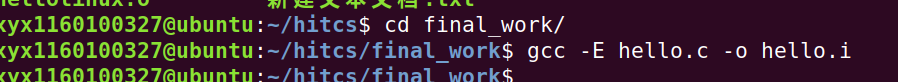
**预处理的作用**：扩展源代码，插人所有用#include命令指定的文件。并扩展所有用#define声明指定的宏。

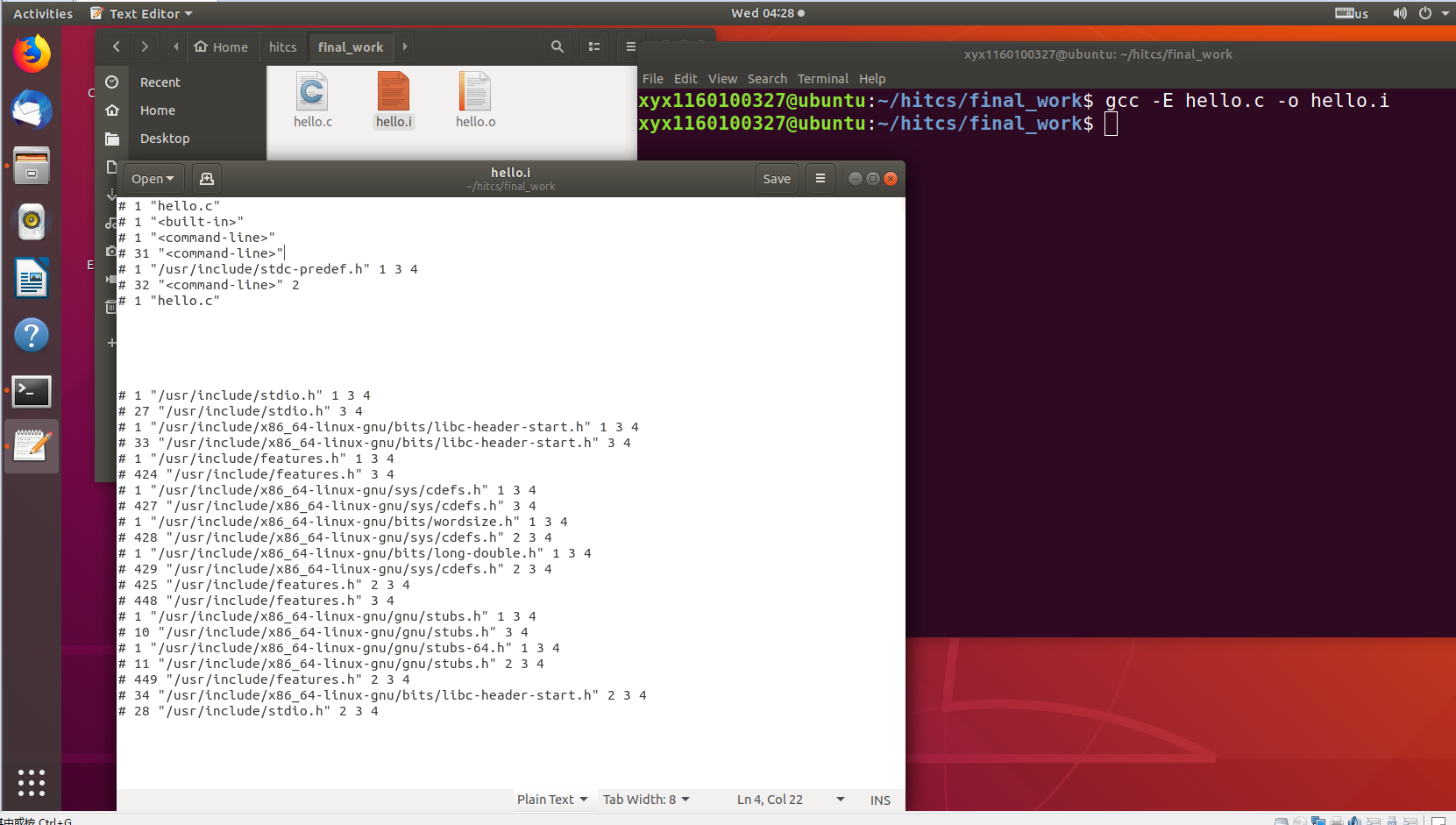
作用主要与三部分有关：宏定义，文件包含，条件编译

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

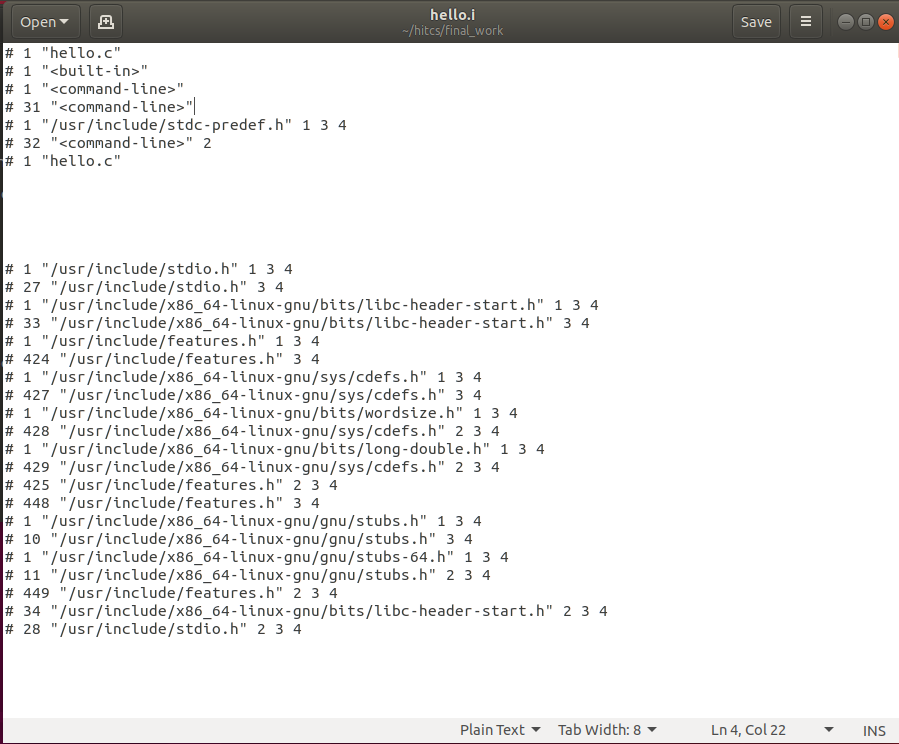
应截图，展示预处理过程！





命令1：gcc -E hello.c -o hello.i输出预编译文本文件

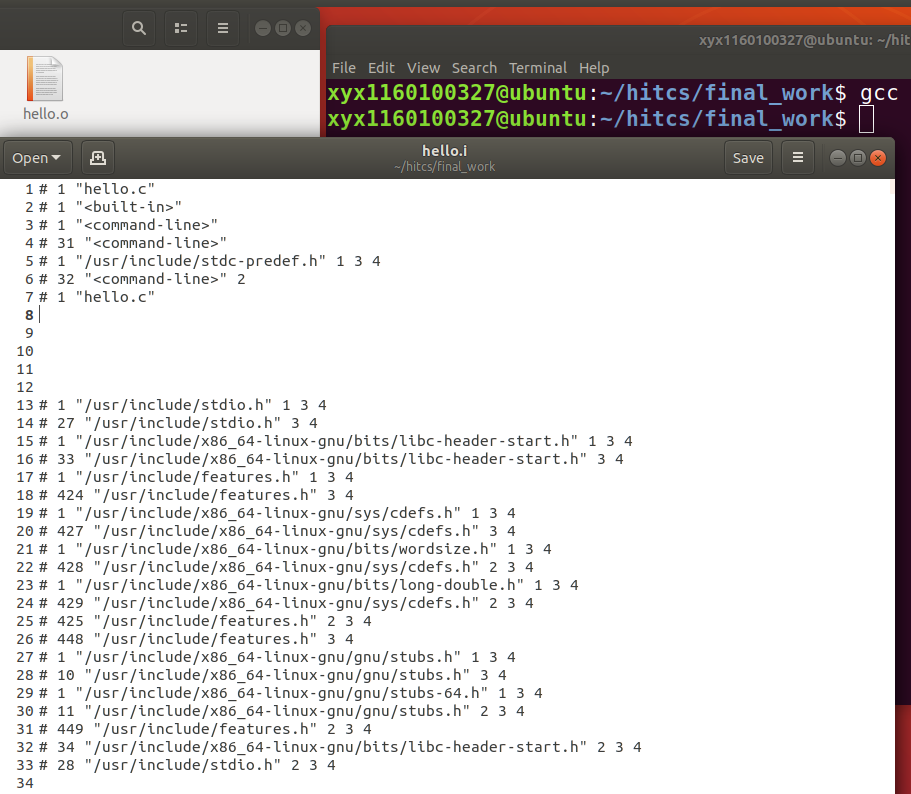
## 2.3 Hello的预处理结果解析



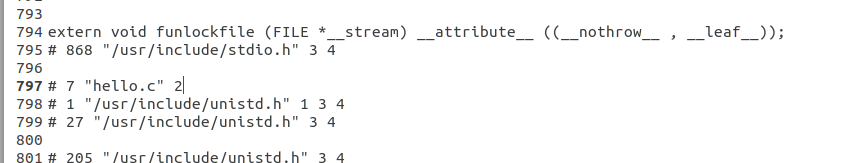
2.3.1 hello.i文件部分截图

通过hello.i文本文件，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，宏替换和并去除注释。

可以看出来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。预处理对头文件stdio.h的展开开始于13行，结束于795行（如截图2.3.2）；同理我们可以得到头文件unistd.h的展开开始于798行，结束于2027行；头文件stdlibd.h的展开开始于2036行，结束于3094行。

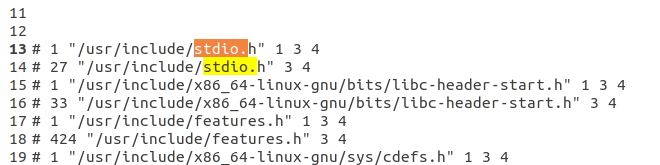


2.3.2 (a) stdio的展开开始



2.3.2 (a) stdio的展开开始

接下来我们介绍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；当使用双引号时，表示首先在当前源文件目录中查找，找不到再到系统头文件目录中查找。以hello.c为例，三个头文件的寻找路径分别是：usr/inlcude/stdio.h，usr/include/unistd.h和usr/inlcude/stdlib.h（如截图2.3.3）。



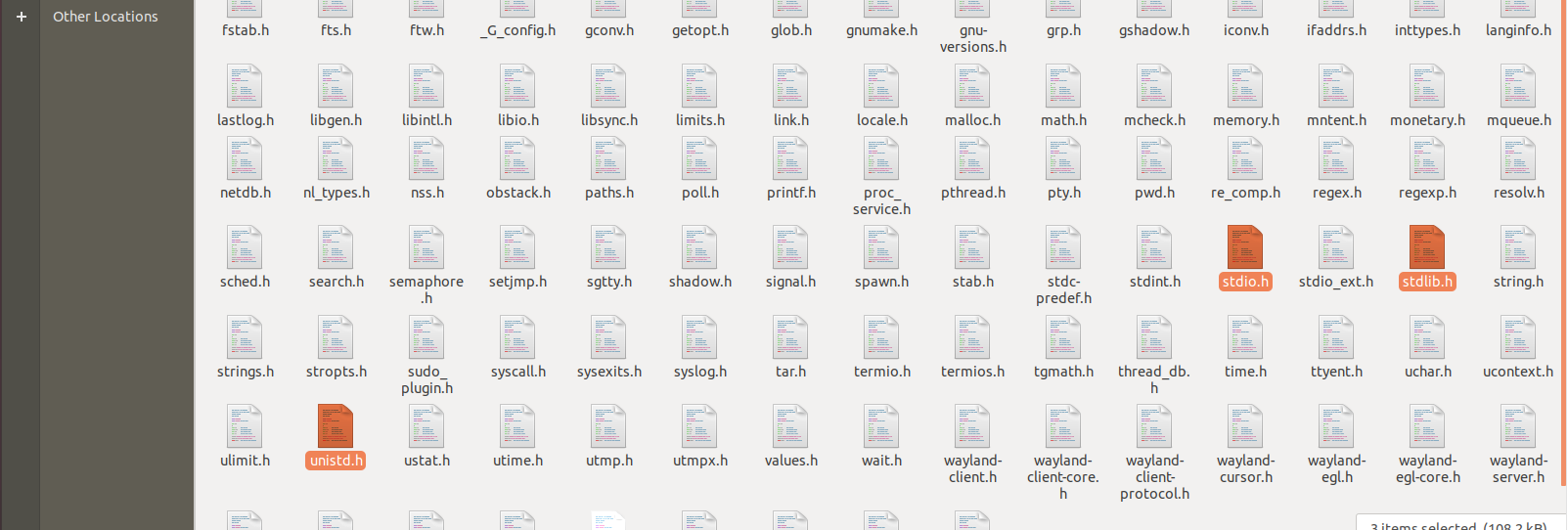
2.3.3 (a) usr/inlcude/stdio.h



2.3.3 (b) usr/inlcude/unistd.h



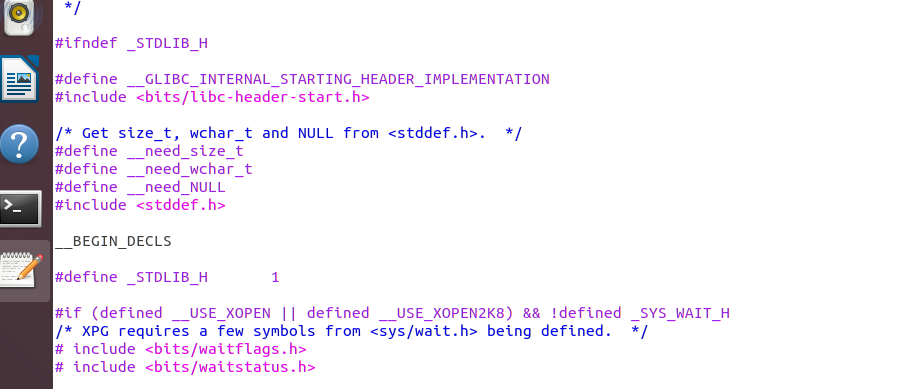
2.3.3 (c) usr/inlcude/stdlib.h



2.3.3 (d) 头文件路径

而且头文件中有大量的宏定义和条件编译语句存在（如截图2.3.4）。预处理阶段会对这些语句进行相应的替换。

原来的程序从第3099行开始，结束于3118行，但是删除了我们的注释信息。（如图2.3.5）



2.3.4 宏定义和条件编译



2.3.5 程序主体

## 2.4 本章小结

本章主要介绍了预处理部分的作用，以及Ubuntu预处理指令，同时对 hello.c文件的预处理结果hello.i文件详细解析其组成部分。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个工作：**头文件的展开；宏替换；去掉注释；条件编译**。

**（第2章0.5分）**

# 第3章 编译

## 3.1 编译的概念与作用

编译的概念：

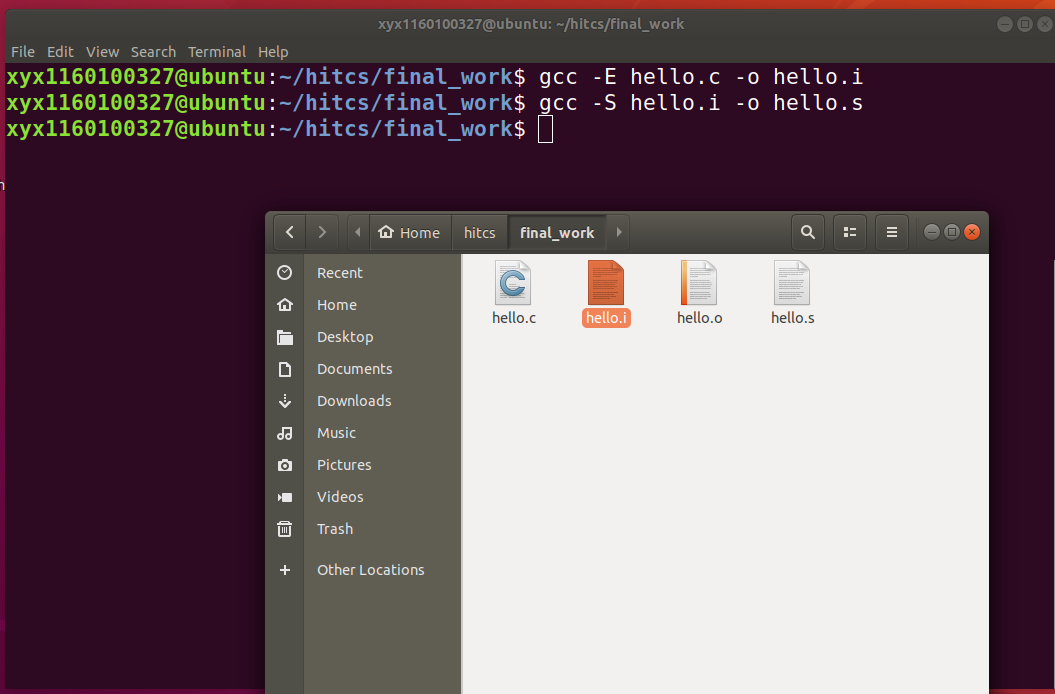
编译就是将高级语言经过词法分析、语法分析、语义分析以及经过一系列优化后生成汇编代码的过程。具体到hello的一生，就是将预处理得到的中间文件hello.i翻译成汇编语言文件hello.s的过程。

编译的作用：

是将高级语言程序转化为机器可直接识别处理执行的机器代码的中间步骤。它包括词法分析、语法分析、中间代码、代码优化四个截断。

3.2 在Ubuntu下编译的命令

编译的命令：gcc -S hello.i -o hello.s



3.3 Hello的编译结果解析

3.3.1 hello.s文件分析：

|  |  |
| --- | --- |
| 节 | 含义 |
| .file | 源文件 |
| .text | 声明以下是代码段 |
| .globl | 声明全局变量 |
| .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long类型 |
| .string | 声明string类型 |
| .size | 声明大小 |
| .type | 声明是类型 |

数据类型：

hello.s中用到的全局变量（int sleepsecs），局部变量（如int i），以及其对应的操作（i++）。

3.3.2.1 整形：

程序中的整数有int sleepsecs, int i, int argc以及常数。对于int sleepsecs，sleepsecs是已经初始化的全局变量。首先在global中声明全局变量，然后在.data节设置对齐方式、类型、大小、类型。如图3-1

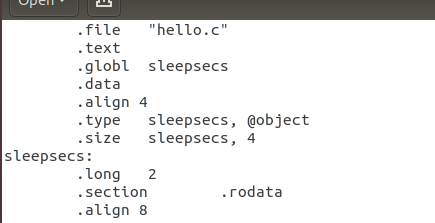


图3-1 sleepsecs的声明

对于int i。局部C变量在运行时被保存在对战或者寄存器。Hello中的i被保存在堆栈中（如图3-2）。

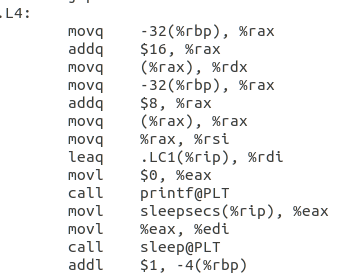


图3-2 变量i

argc 是main函数的第一个形参，分析可得，argc（即%edi）赋值给了-20(%rbp).

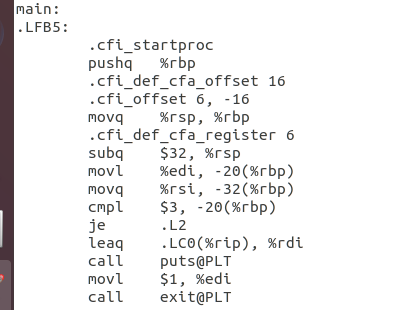


图3-2 变量argc

在源程序hello.c中出现的常数，在hello.s中直接以立即数的形式出现。可以用$0x\_\_来表示。

3.3.2.2数组：

hello.c中还出现了数组，是main函数的第二个形参（char \*argv[]）。表示我们的命令行命令。argv数组中一个元素大小为8个字节（从图3-3中的addq $8, %rax 可以看出）.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。

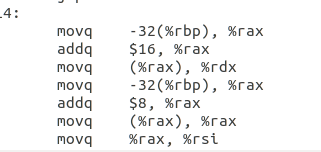


图3-3 数组内容的取出

3.3.2.3字符串：

在hello.s文件中，我们可以看到一个字符串（如图3-4），347\224\250\346\263\225: Hello \345\255\246\345\217\267 \345\247\223\345\220\215\357\274\201，这些是学号、姓名的UTF-8格式。

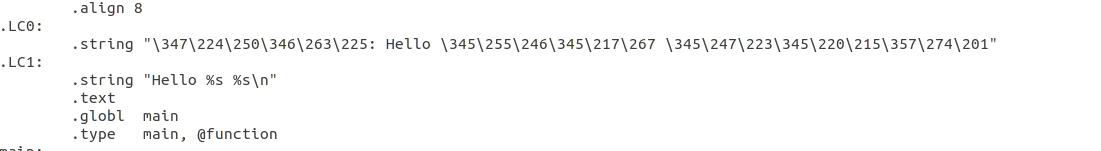


图3-4 字符串

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。

3.3.2.4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。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。（如图3-5）

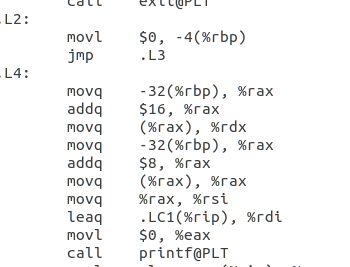


图3-5

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（如图3-6）。

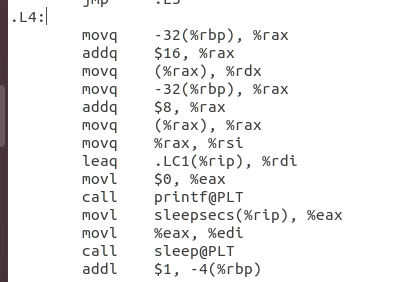


图3-6

3.3.2.5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）。

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：向零舍入。

3.3.2.6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 |

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3-6）

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。

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。

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（多个算术操作）

3.3.2.7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。

关系操作是cmpl $3, -20(%rbp);配合跳转语句je .L2。

3.3.2.8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。

3.3.2.9 控制转移

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

3.3.2.10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。

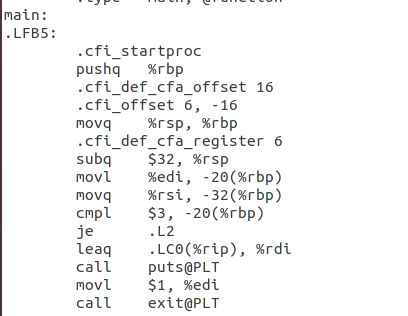


图3-7

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。

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。

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）。

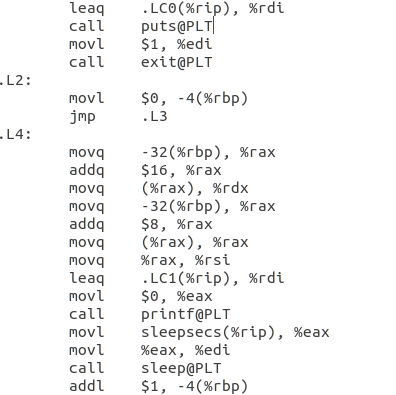


图3-7

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）。

## 3.4 本章小结

本章主要详细介绍了编译的概念与作用，在Ubuntu下进行编译的指令。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）的分析。

**（第3章2分）**

# 第4章 汇编

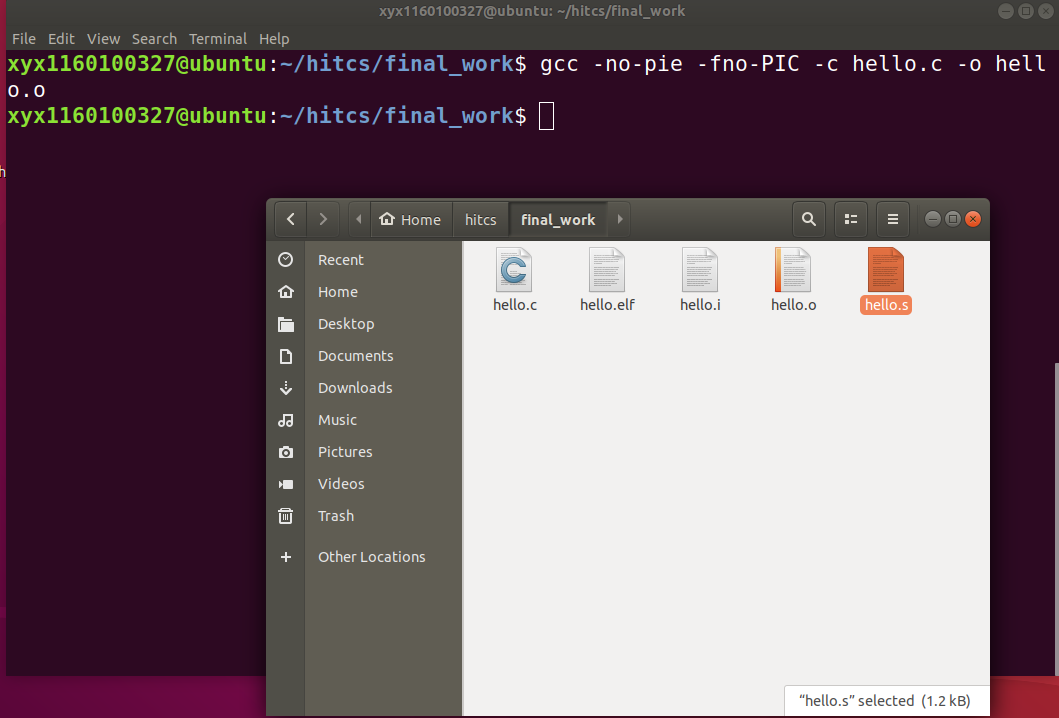
## 4.1 汇编的概念与作用

概念：把汇编语言翻译成机器语言的过程叫做汇编

作用：.o是一个二进制文件其中包含程序的指令，可让计算机进行操作

## 4.2 在Ubuntu下汇编的命令

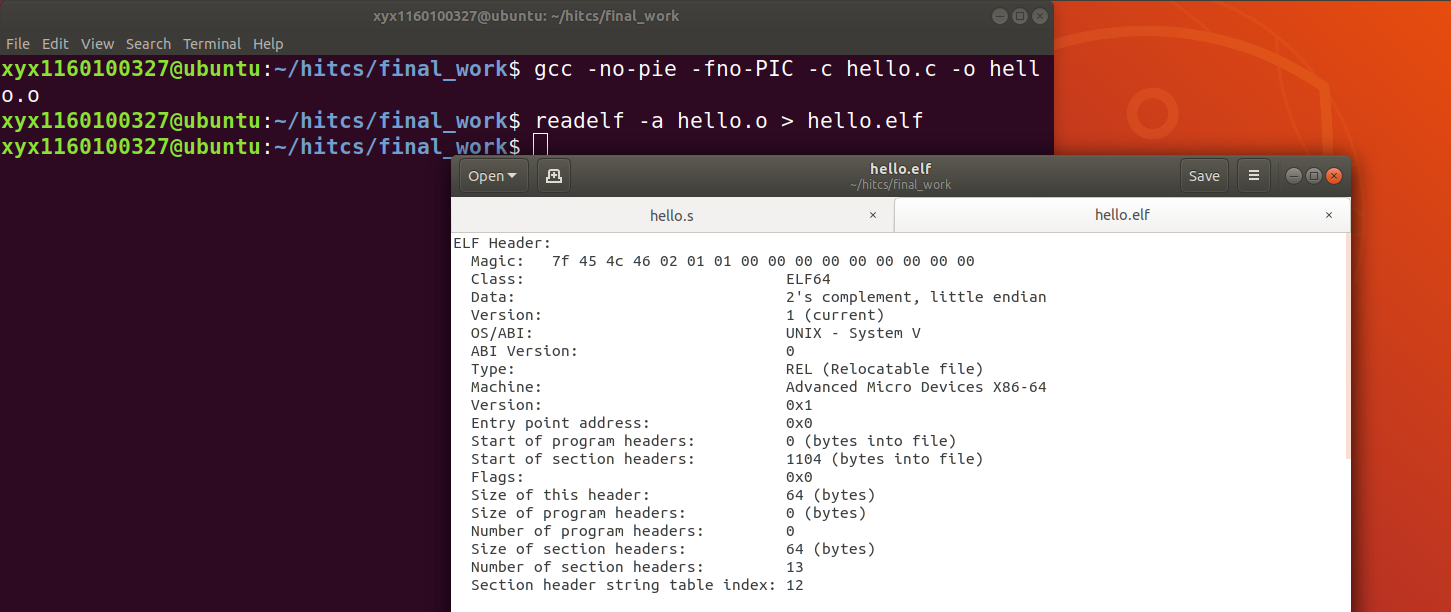
命令：gcc -no-pie -fno-PIC -c hello.c -o hello.o



## 4.3 可重定位目标elf格式

1）读取可重定位目标文件。

分析hello.o的ELF格式，用readelf等列出其各节的基本信息，特别是重定位项目分析。



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 |
| .text |
| .rodata |
| .data |
| .bss |
| .symtab |
| .rel.text |
| .rel.data |
| .debug |
| .line |
| .strtab |
| 节头部表 |

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。

.text：已编译程序的机器代码。

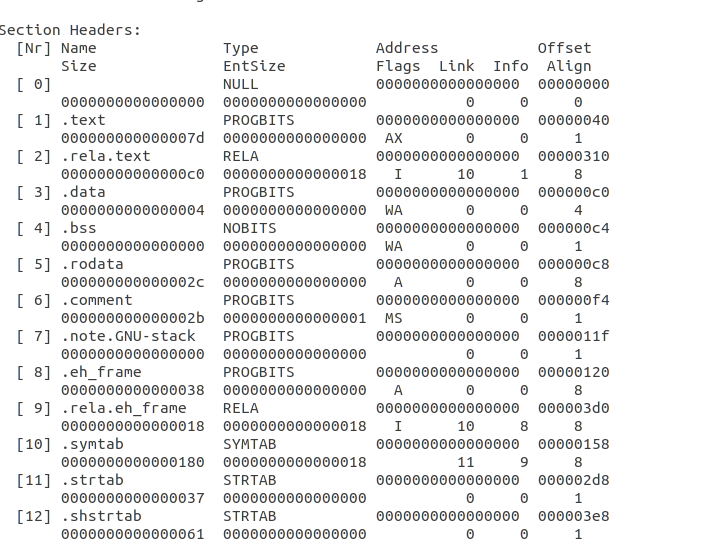
.rodata：只读数据，比如printf语句中的格式串和开关语句的跳转表。

.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。



.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）；

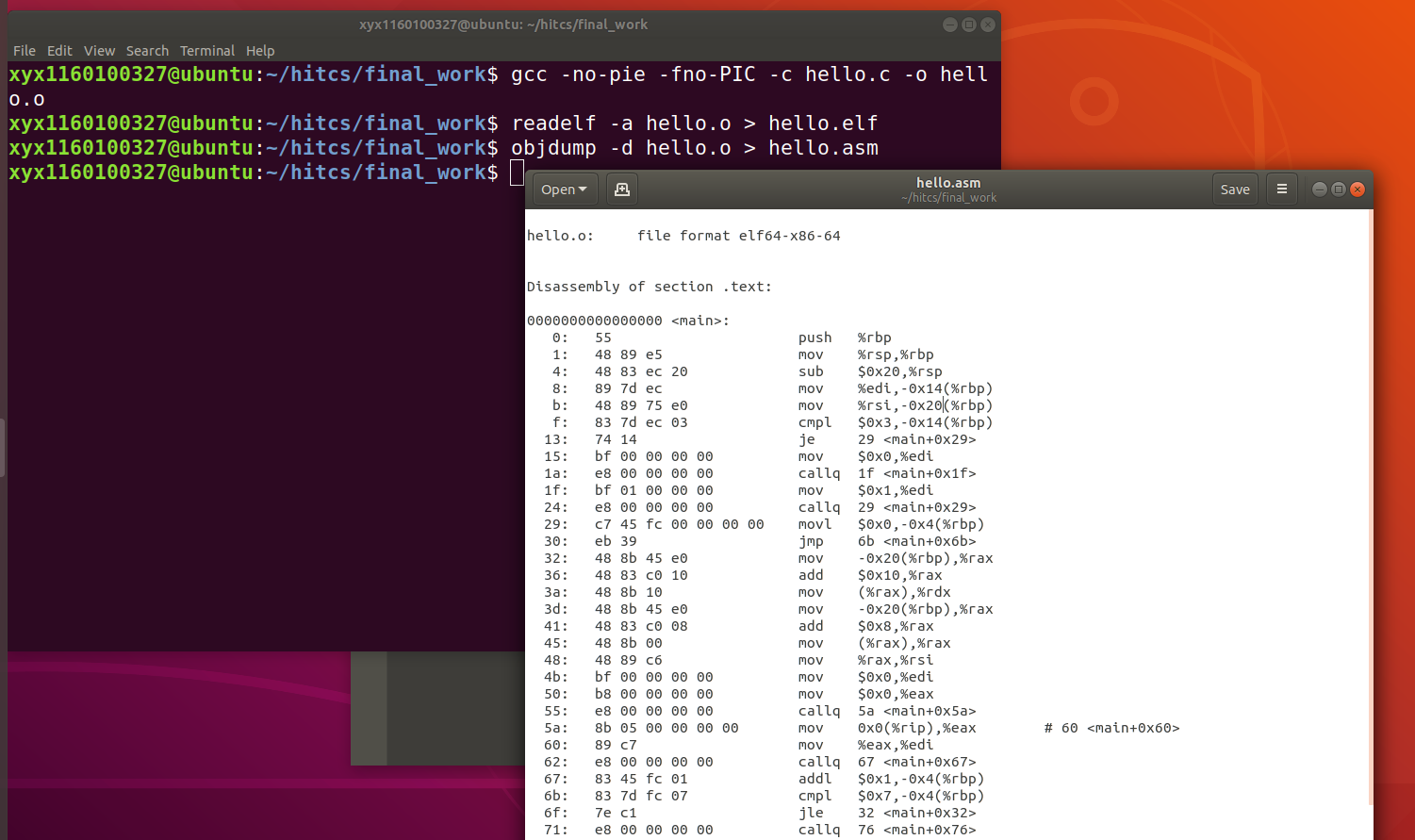
第一条PLT指令通过GOT[5]进行间接跳转（因为每个GOT条目初始时都指向它对应的PLT条目的第二条指令，这个间接跳转只是简单的把控制传送回PLT[3]中的下一条指令）；

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。

## 4.4 Hello.o的结果解析



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

**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）。

**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；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。

* **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。

（*以下格式自行编排，编辑时删除*）

**（第4章1分）**

# 第5章 链接

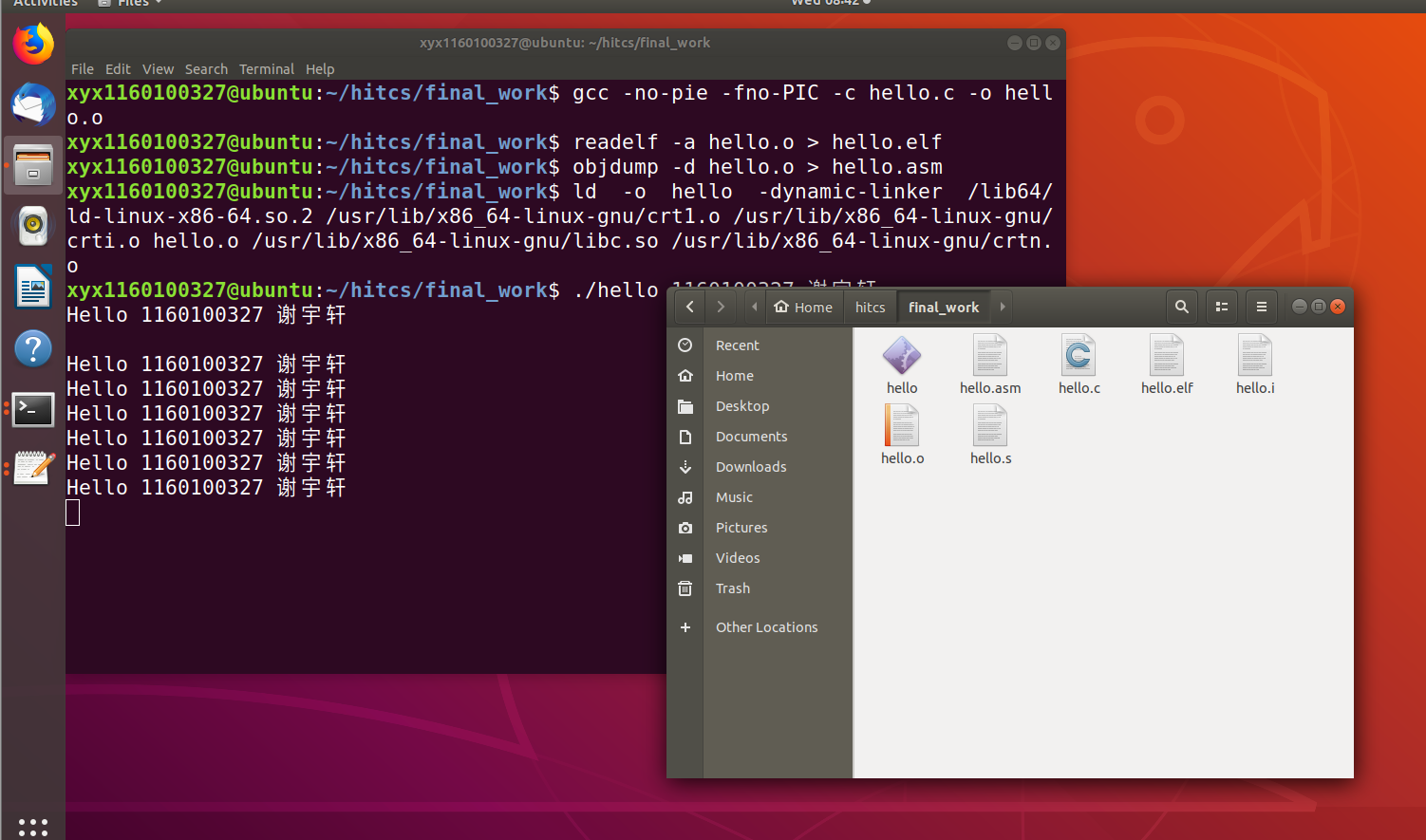
## 5.1 链接的概念与作用

## 概念：链接是将各种代码和数据片段收集并组合称为一个单一文件的过程，这个文件可被加载（复制）到内存并执行。链接可以执行于编译时，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：使得分离编译成为可能；动态绑定(binding):使定义、实现、使用分离。

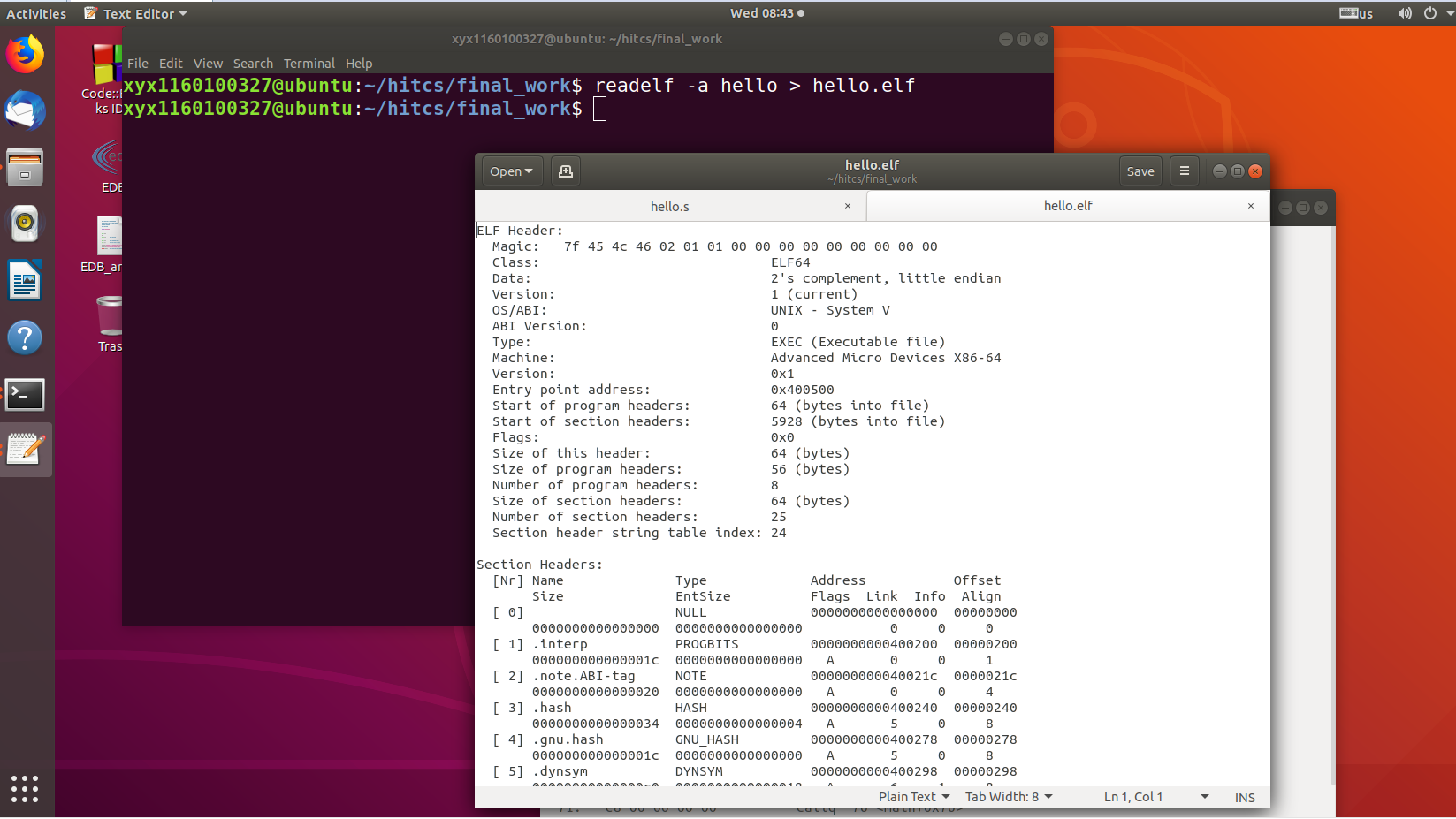
## 5.2 在Ubuntu下链接的命令

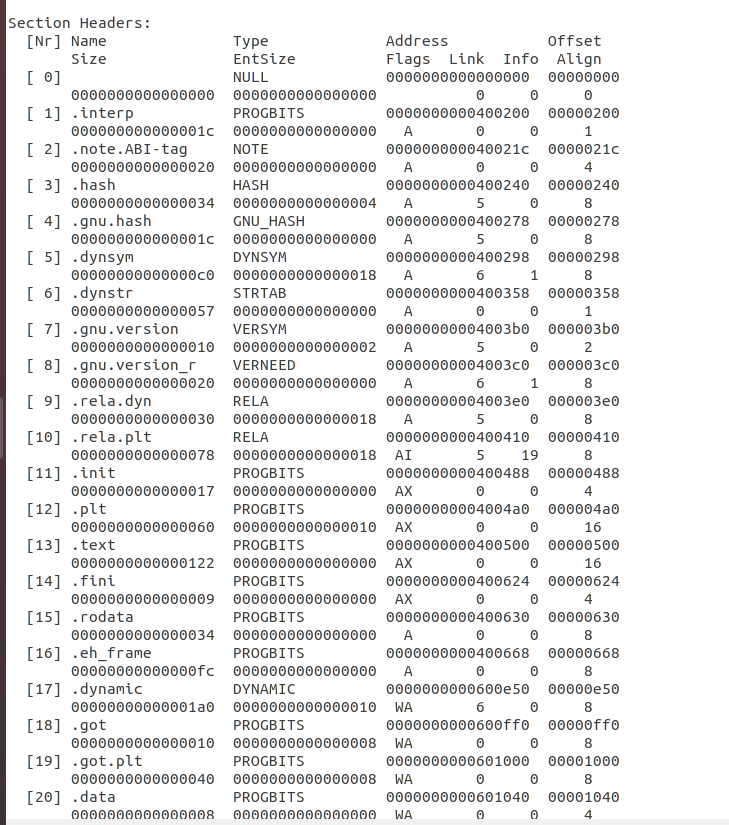
ld -o hello -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib/x86\_64-linux-gnu/crt1.o /usr/lib/x86\_64-linux-gnu/crti.o hello.o /usr/lib/x86\_64-linux-gnu/libc.so /usr/lib/x86\_64-linux-gnu/crtn.o



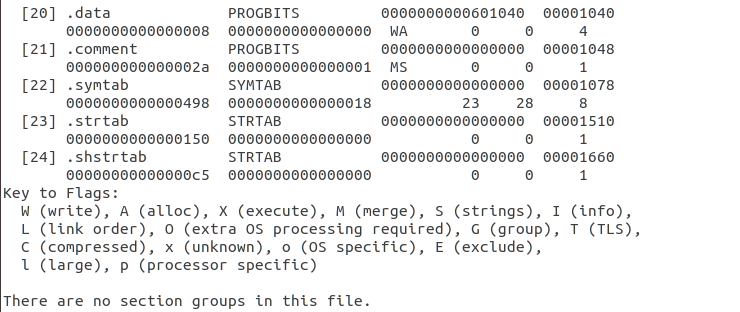
## 5.3 可执行目标文件hello的格式

分析hello的ELF格式，用readelf等列出其各段的基本信息，包括各段的起始地址，大小等信息。





节头表上半部分截图



节头表下半部分截图

## 5.4 hello的虚拟地址空间

用edb加载hello后如图所示：

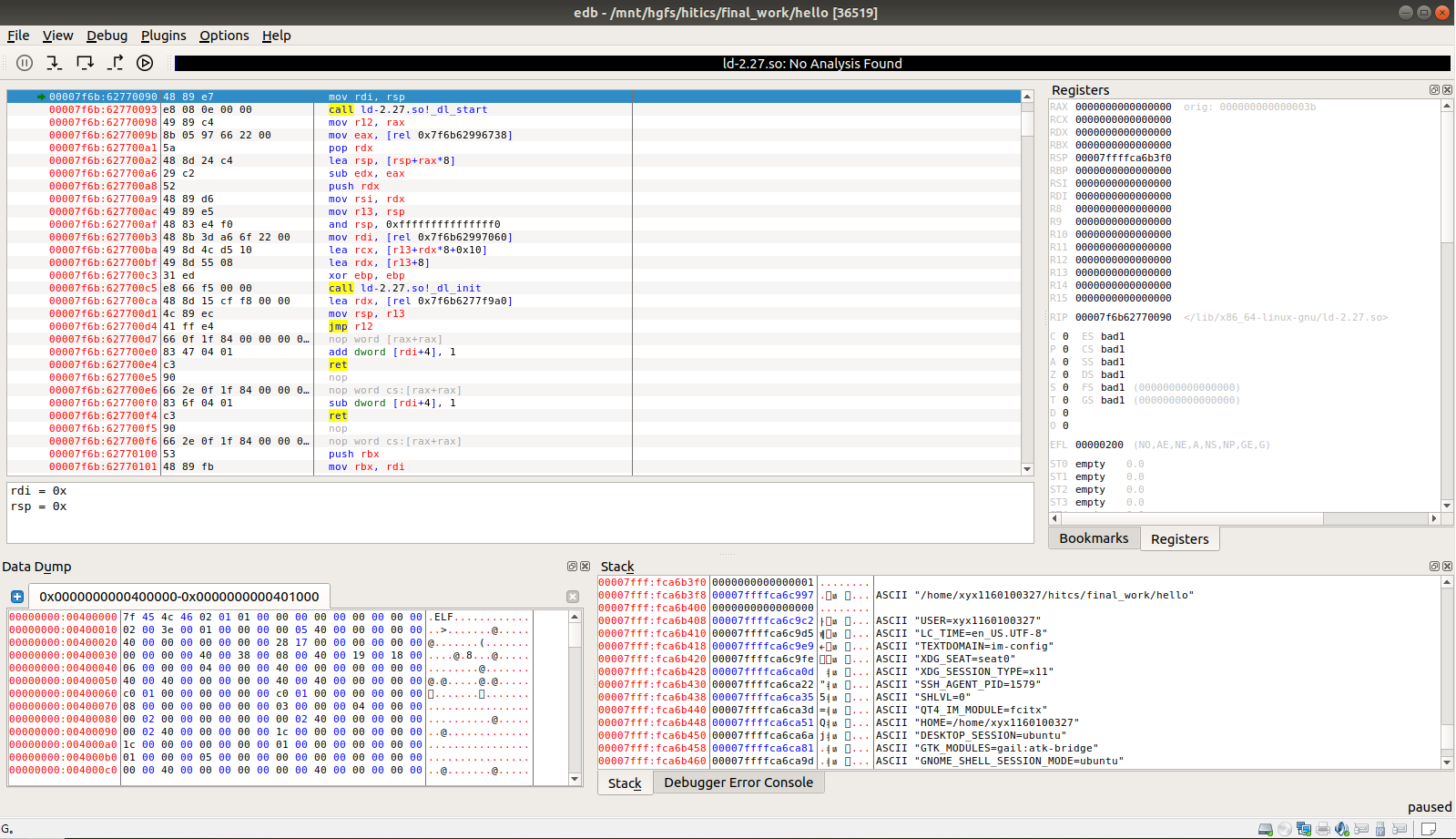


图1

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，如图2

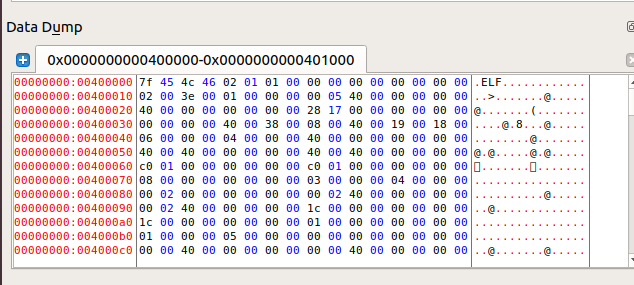


图2

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），如图3。

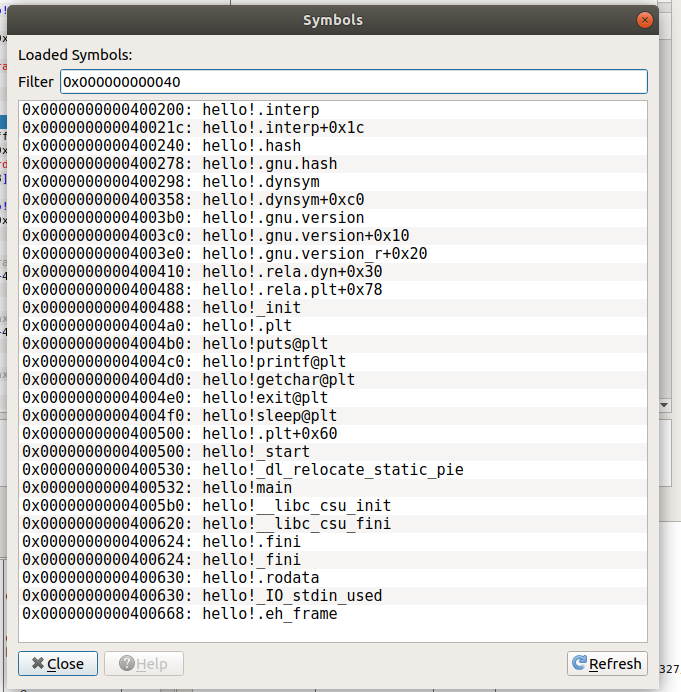


图3

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，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.

## 5.5 链接的重定位过程分析

1.objdump –d hello > hello\_exe.asm

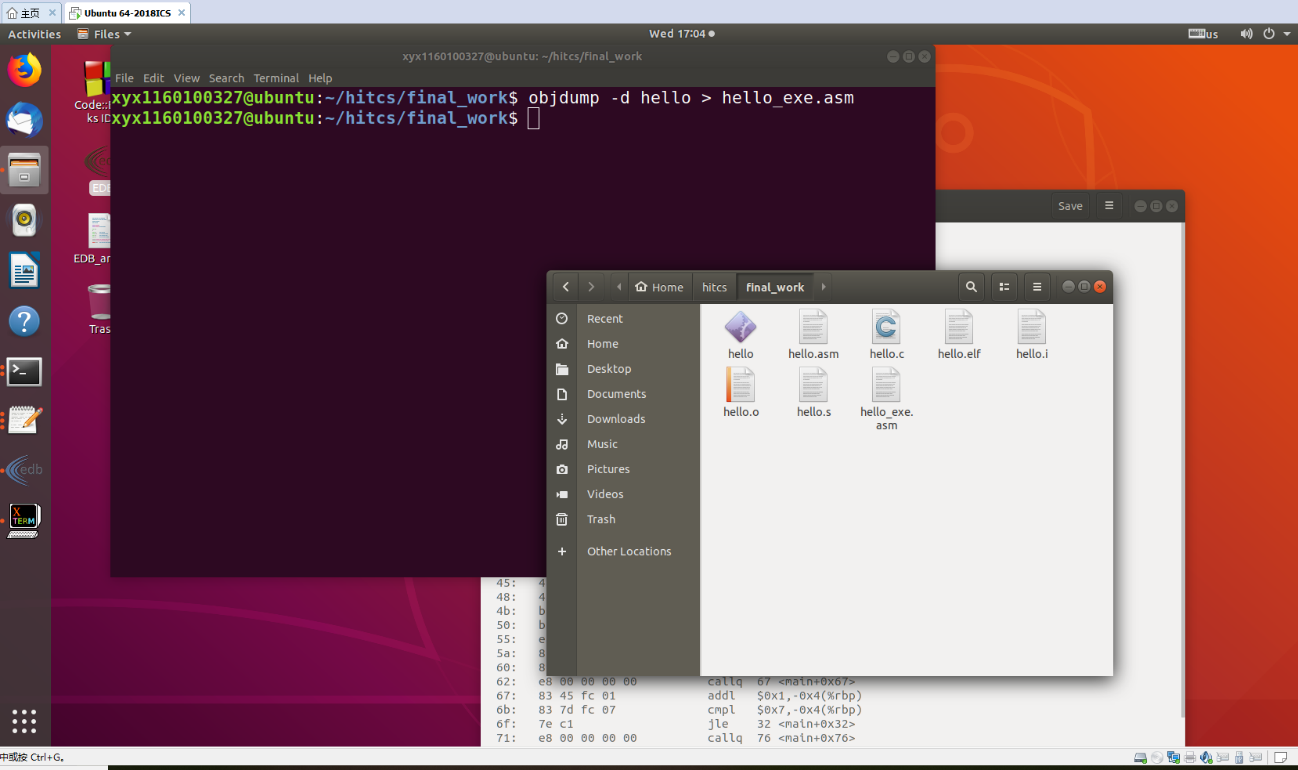


图1 输出hello的反汇编文件

2. 分析hello反汇编文件和hello.o的反汇编文件区别：

1）我们发现hello\_exe.asm比hello.asm多了许多文件节。

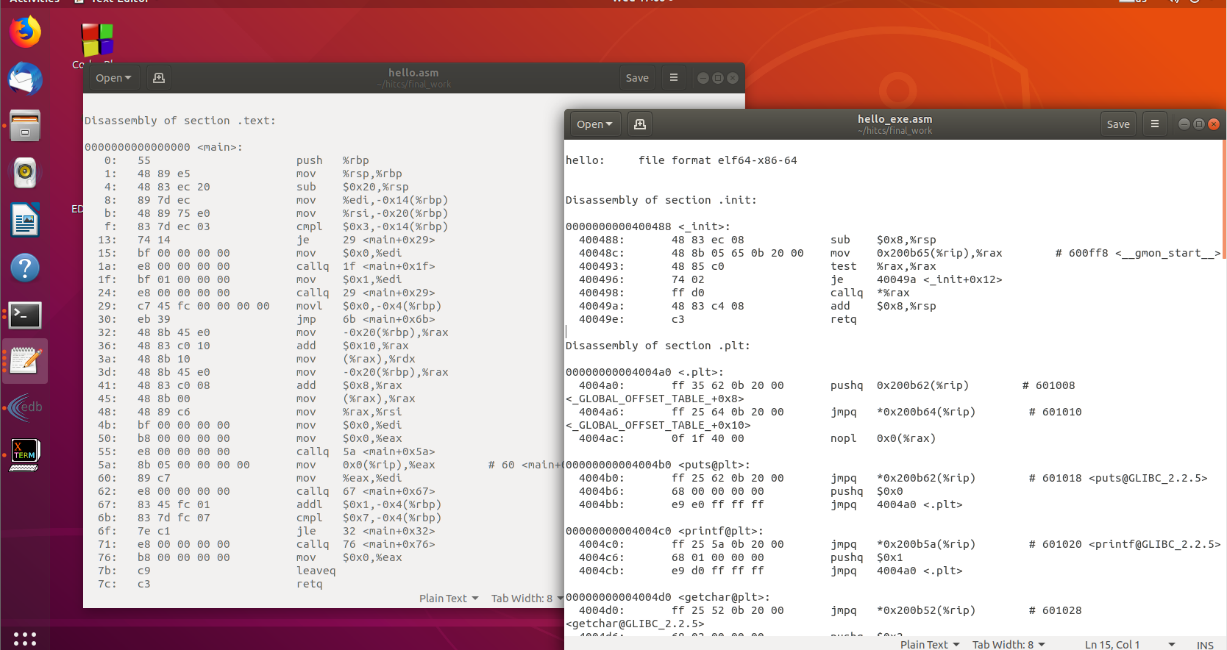


图2

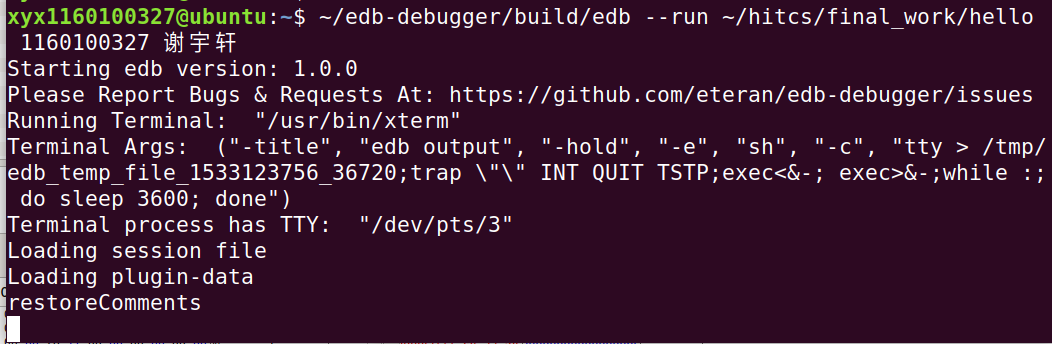
2）hello\_exe.asm地址为虚拟地址，hello.asm中的地址为相对偏移地址。（如图2）

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.

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）

3. 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



|  |  |
| --- | --- |
| 子程序名 | 程序地址 |
| ld -2.27.so!\_dl\_start | 0x7fec665deea0 |
| ld-2.27.so!\_dl\_init | 0x7fcb4af13630 |
| libc-2.27.so!\_\_libc\_start\_main | 0x00007facec822ab0 |
| hello!printf@plt | 0x4004c0 |
| hello!sleep@plt | 0x4004f0 |
| hello!getchar@plt | 0x4004d0 |
| libc-2.27.so!exit | 0x7facec844120 |

## 5.7 Hello的动态链接分析

hello在调用.so共享库函数时，会涉及到动态链接。现代系统在处理共享库在地址空间中的分配的时候，采用了位置无关代码(PIC)方式。位置无关代码指，编译共享模块的代码段，是把它们加载到内存的任何位置而无需链接器修改。用户对GCC使用-fpic选项指示GNU编译系统生成PIC代码。共享库的编译必须总是使用该选项。

PIC代码引用包括数据引用和函数调用。对数据引用有一个事实，就是代码段中任何指令和数据段中任何变量之间的距离是一个运行时常量，与代码段和数据段的绝对内存位置是无关的。在编译器想要生成对PIC全局变量引用时，在数据段开始的地方创建了全局偏移量表(GOT)

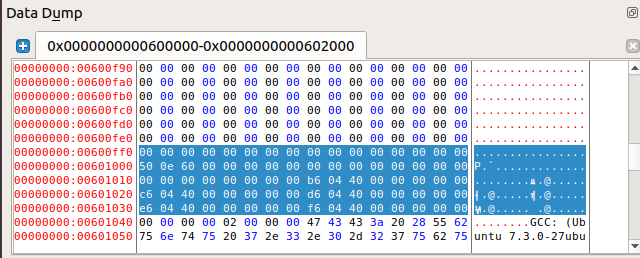
动态链接库中的函数在程序执行的时候才会确定地址，所以编译器无法确定其地址，在汇编代码中也无法像静态库的函数那样体现。

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。

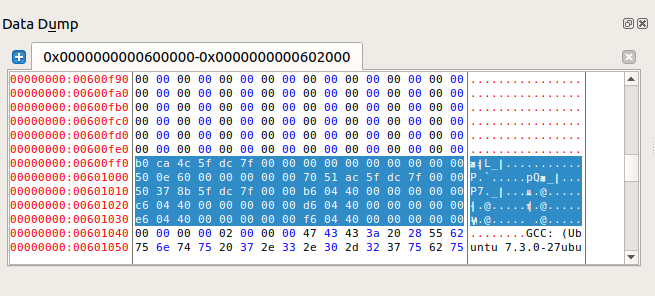
hello程序对动态链接库的引用，基于数据段与代码段相对距离不变这一个事实，因此代码段中任何指令和数据段中任何变量之间的距离都是一个运行时常量。

在dl\_init前后发生变化如下：

通过EDB调试，能够看出这个变化。先观察调用dl\_init前，动态库函数指向的地址。从上文节头中能够读取到GOT表的起始位置，即0x601000。在dl\_init调用之前可以查看其值，发现均为0。调用dl\_init后再次查看,0x601008he 0x601010处的两个八字节数据分别改变。这两个位置已经有了一段地址，分别为0x7fb7eaa21170和0x7fb7ea80f750。其中 GOT[1]指向重定位表，GOT[2] 存放动态链接器入口的地址。



调用前



调用后

## 5.8 本章小结

本章结合实验中的hello可执行程序说明了链接的概念及作用。并对hello的elf格式进行了分析比对，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，了解重定位；经历了hello的执行过程，并且分析了hello的动态链接过程。

**（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：

## 是指计算机中已运行的程序。进程为曾经是分时系统的基本运作单位。在面向进程设计的系统（如早期的UNIX，Linux 2.4及更早的版本）中，进程是程序的基本执行实体；在面向线程设计的系统（如当代多数操作系统、Linux 2.6及更新的版本）中，进程本身不是基本运行单位，而是线程的容器。程序本身只是指令、数据及其组织形式的描述，进程才是程序（那些指令和数据）的真正运行实例。若干进程有可能与同一个程序相关系，且每个进程皆可以同步（循序）或异步（平行）的方式独立运行。现代计算机系统可在同一段时间内以进程的形式将多个程序加载到存储器中，并借由时间共享（或称时分复用），以在一个处理器上表现出同时（平行性）运行的感觉。同样的，使用多线程技术（多线程即每一个线程都代表一个进程内的一个独立执行上下文）的操作系统或计算机体系结构，同样程序的平行线程，可在多CPU主机或网络上真正同时运行（在不同的CPU上）。

作用：

给应用程序提供了关键的抽象，一个独立地逻辑控制流，提供一个假象，好像我们的程序独占地使用处理器，还提供了一个私有的地址空间，提供一个假象，好像我们的程序独占地使用内存系统。

## 6.2 简述壳Shell-bash的作用与处理流程

## 概念：

## 是一个命令处理器，通常运行于文本窗口中，并能执行用户直接输入的命令。Bash还能从文件中读取命令，这样的文件称为脚本。和其他Unix shell 一样，它支持文件名替换（通配符匹配）、管道、here文档、命令替换、变量，以及条件判断和循环遍历的结构控制语句。包括关键字、语法在内的基本特性全部是从sh借鉴过来的。其他特性，例如历史命令，是从csh和ksh借鉴而来。总的来说，Bash虽然是一个满足POSIX规范的shell，但有很多扩展。

## 流程：

## ⑴读取命令

## ⑵分割字符串获得参数

## ⑶判断是否为内置命令，若是则执行

## ⑷若不是则调用相应的程序为其分配子进程并运行

## ⑸shell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

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

## 6.4 Hello的execve过程

1. 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。
2. 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).
3. 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. 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。

1. 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） |
|  |

## 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

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

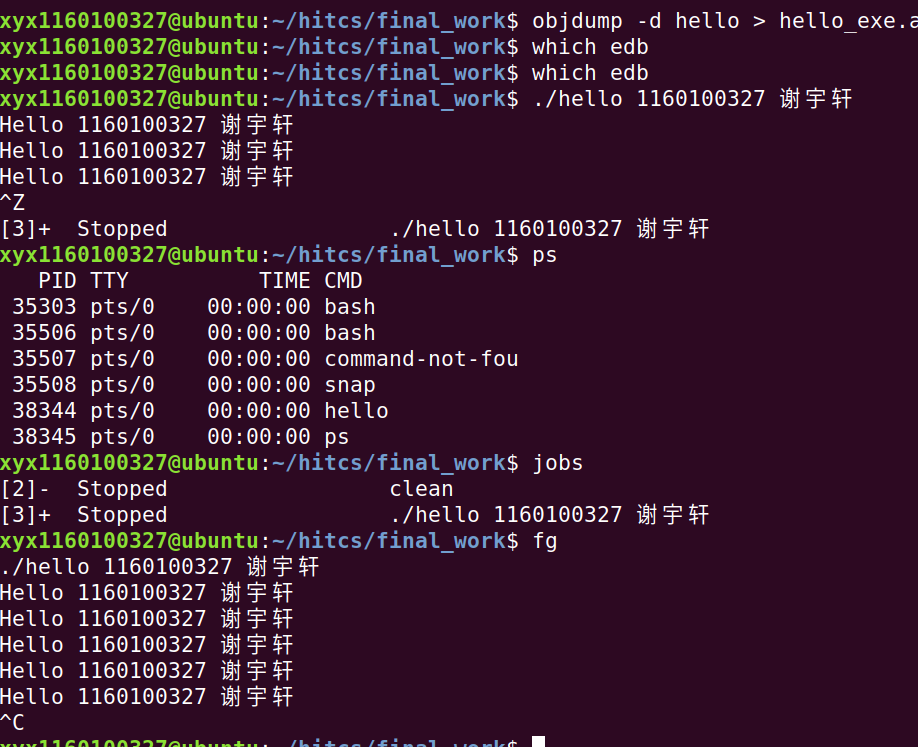
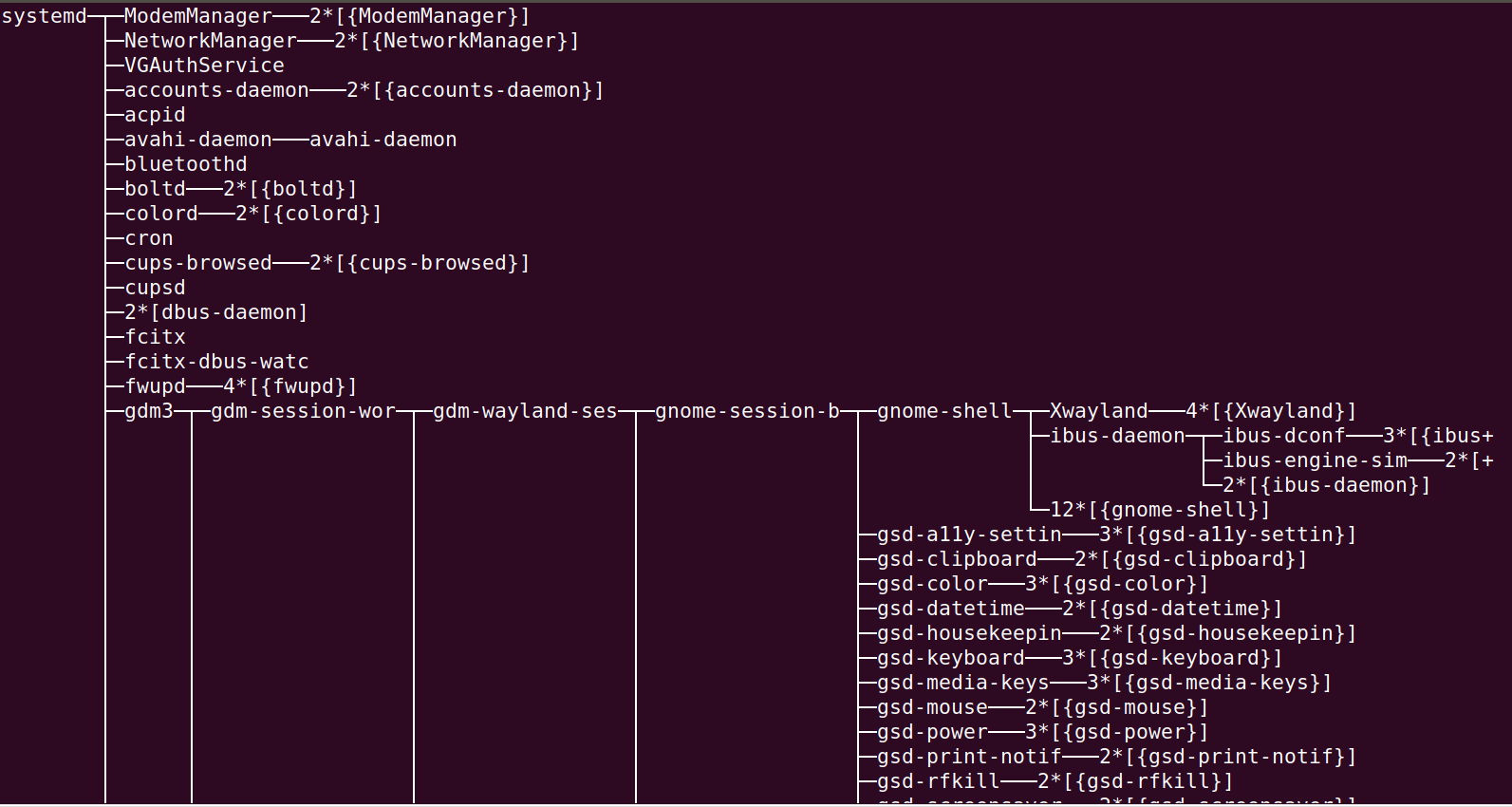
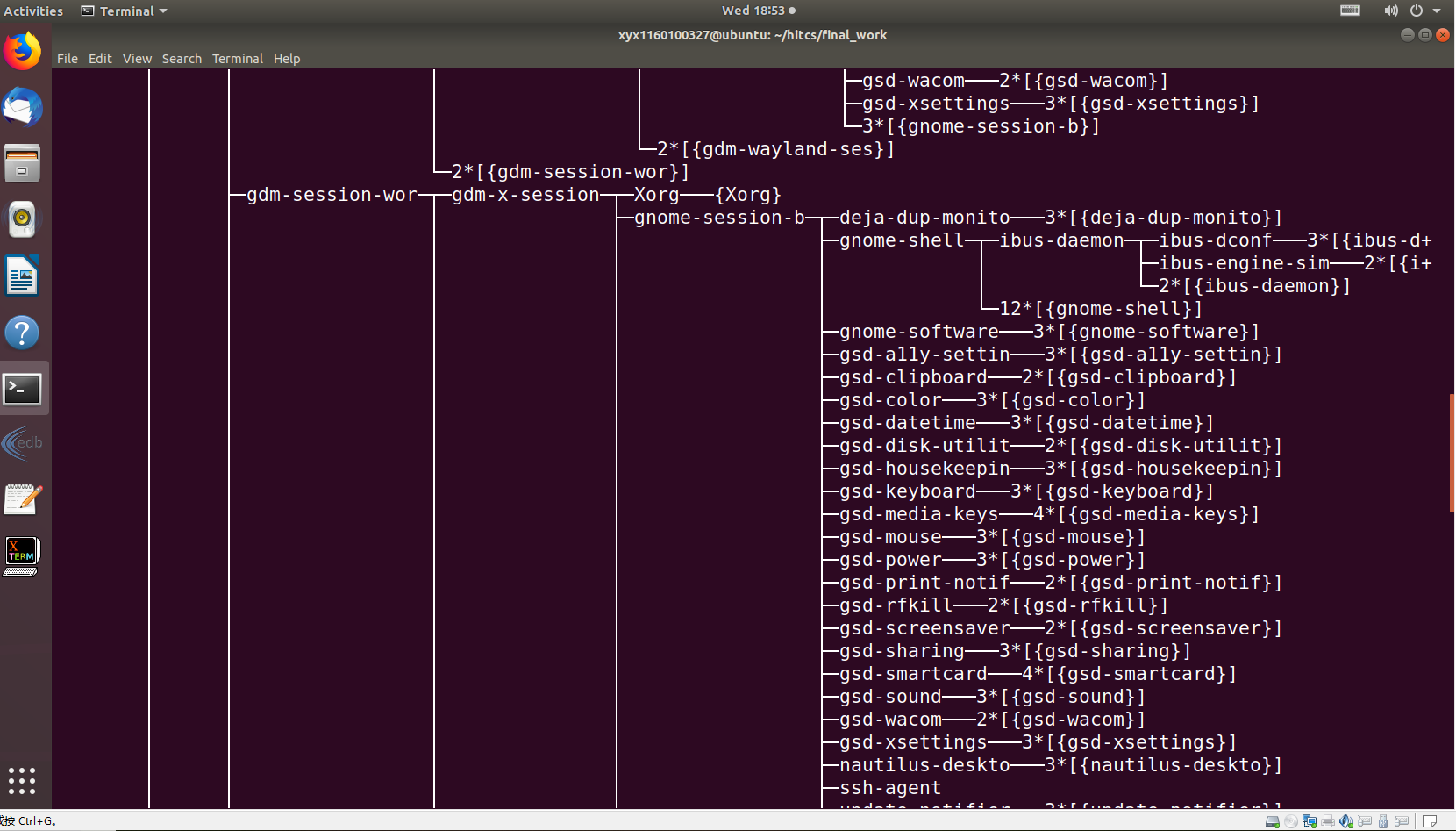


图1





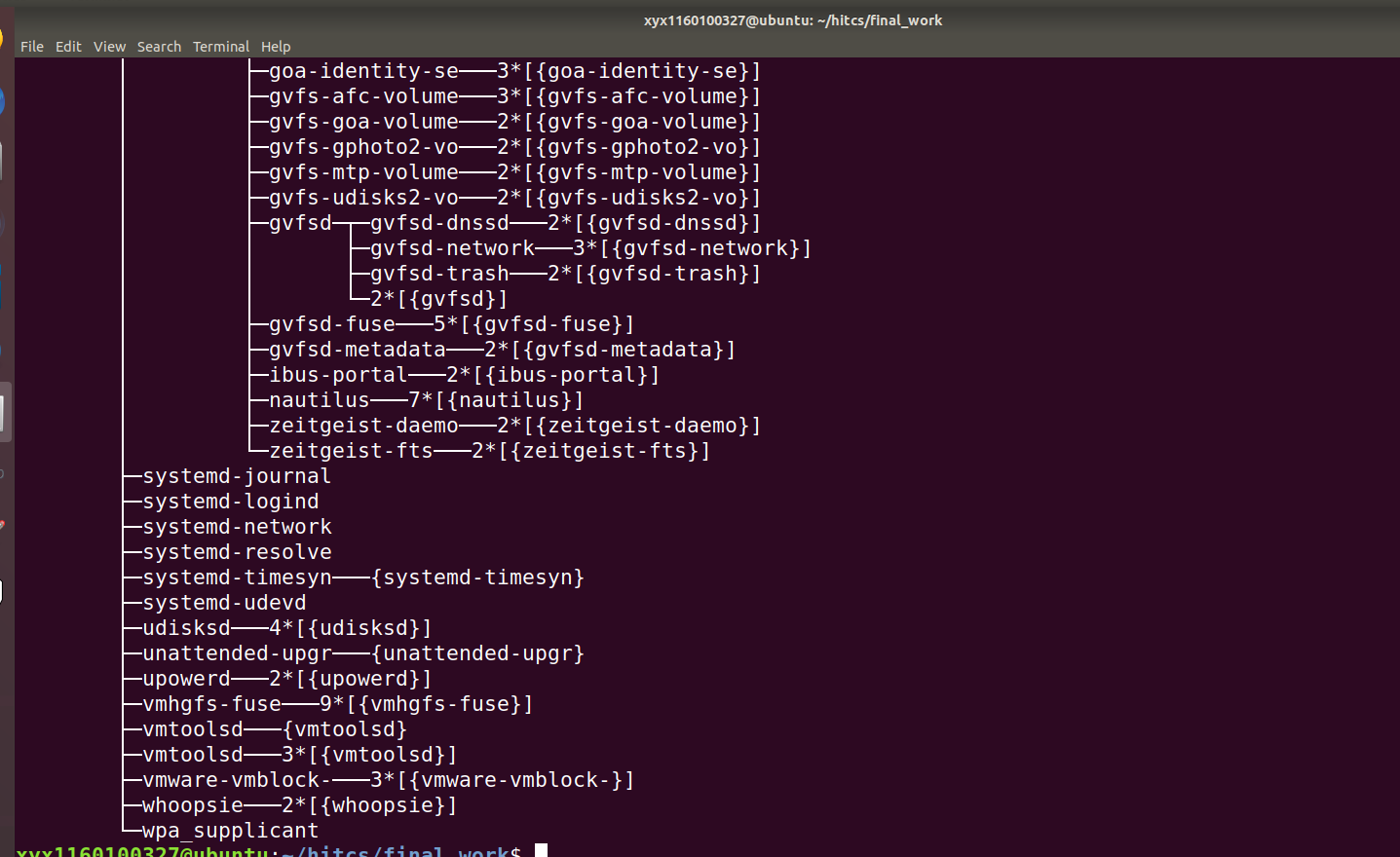


图2 pstree

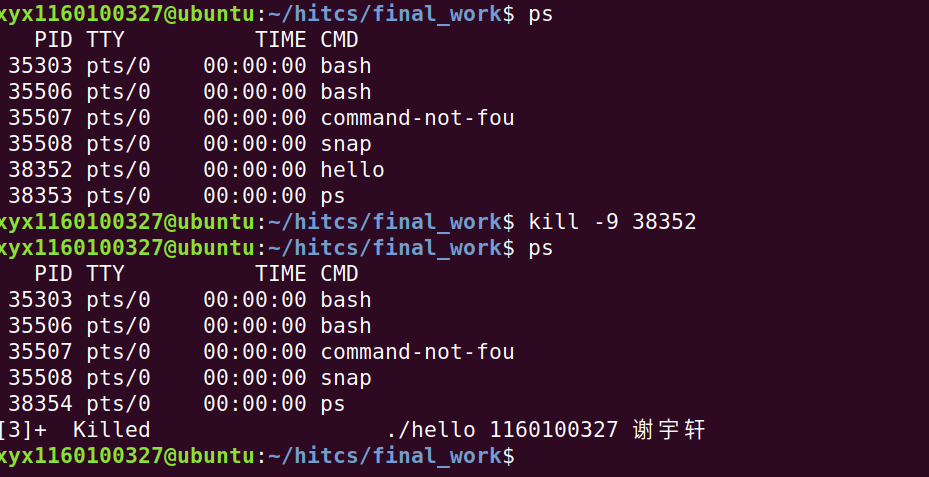


图3 kill杀死子程序

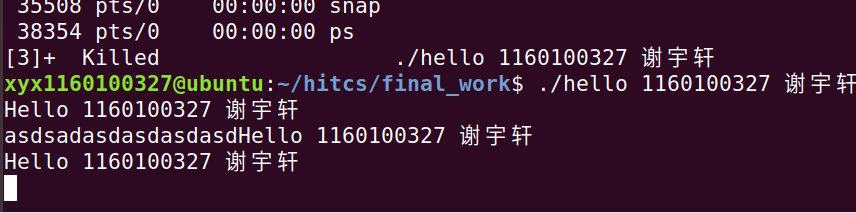


图4 终端随意输入

键盘输入ctrl+c属于中断异常：

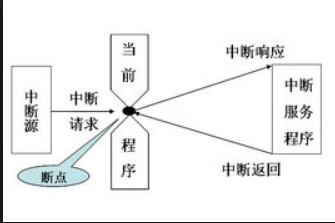


图5 ctrl+c或者ctrl+z键盘的**中断**异常

函数执行可能会执行系统调用函数exit，属于陷阱。其处理方式如图6。

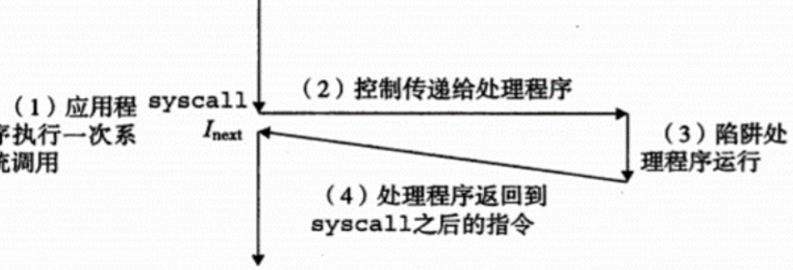


图6 exit系统调用函数的**陷阱**异常

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 38352。**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. 在操作系统层，内核用ECF提供进程的基本概念。
2. 硬件层，异常是由处理器中的事件触发的控制流中的突变.

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程，等待他们的子进程停止或者终止，捕获来自其他进程的信号。

4）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：**

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**

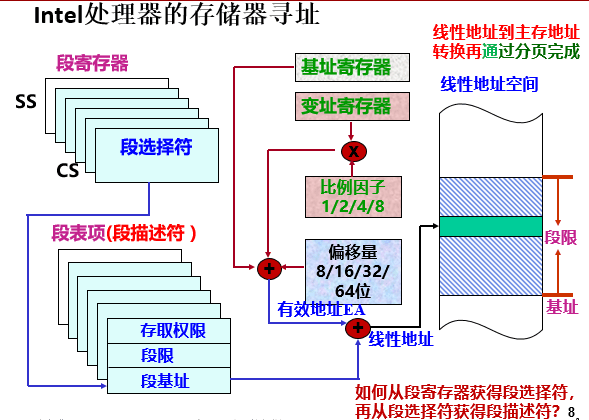


图1：intel储存器寻找

**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。**

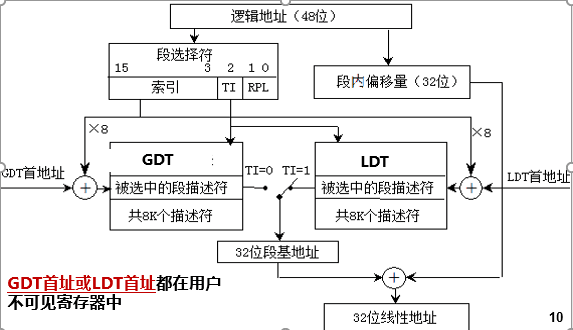


图2：逻辑地址->线性地址

## 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**

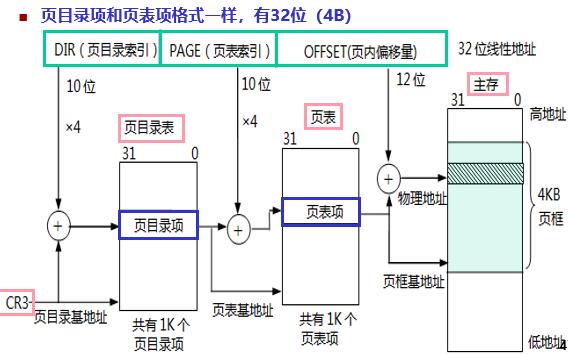


图3：线性地址到物理地址

* 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。

6）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。

7）D：修改位(dirty bit)。页目录项中无意义，只在页表项中才有。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为1。

8）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

## ⑴ 页面命中

## 笫l步：处理器生成一个虚拟地址， 并把它传送给MMU。

## 第2步：MMU生成PTE地址，并从高速缓存／主存请求得到它。

## 笫3步：高速缓存／主存向MMU返回PTE

## 第4步：MMU构造物理地址， 并把它传送给高速缓存／主存。

## 第5步：高速缓存／主存返回所请求的数据字给处理器 。

## ⑵ 缺页：页面命中完全是由硬件来处理的，与之不同的是，处理缺页要求硬件和操作系统内核协作完成，

## 第1步到笫3步：与前三步一致。

## 第4步：PTE的有效位是零，所以MMU触发了－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。

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

## 第6步：缺页处理程序页面调入新的页面，并更新内存中的PTE。

## 第7步： 缺页处理程序返回到原来的进程，再次执行导致缺页的指令。CPU将引起缺 页的虚拟地址重新发送给MMU。因为虚拟页面现在缓存在物理内存中，所以就会命 中，在MMU执行了图7.4.4-b中的步骤之后，主存就会将所请求字返回给处理器。

## 7.5 三级Cache支持下的物理内存访问

## 在cache中物理地址寻址，按照三个步骤：组选择、行匹配和字选择。在冲突不命中时还会发生行替换。

## 高速缓存(S, E, B, m）被组织成一个有S=2s个高速缓存组(cache set)的数组。 每个组包含E 个高速缓存行(cache line).每个行是由一个B=2b字节的数据块(block)组成的， 一个有效位(valid bit) 指明这个行是否包含有意义的信息,还有t=m-(b+s)个标记位(tag bit)(是当前块的内存地址的位的一个子集），它们唯一地标识存储在这个高速缓存行中的块。

## 一般而言，高速缓存的结构可以用元组(S, E, B, m)来描述。 高速缓存的大小(或容量)C指的是所有块的大小的和。标记位和有效位不包括在内。因此,C=S×E×B。

## 高速缓存的结构将m个地址位划分为t个标记位，s个组索引位，和b个块偏移位。

## 在组选择中，cache按照物理地址的s个组索引位(S=2s)来定位该地址映射的组。

## 选择好组后，遍历组中的每一行，比较行的标记和地址的标记，当且仅当这两者相同，并且行的有效位设为1时，才可以说这一行中包含着地址的一个副本。也就是缓存命中了。

## 最后是字选择。定位好了要寻址的地址在哪一行之后，根据地址的块偏移量，在行的数据块中偏移寻址，最后得到的字，就是我们寻址得到的字。

## 如果缓存不命中，那么它需要从存储器层次结构中的下一层取出被请求的块，然后将新的块存储在组索引位指示的组中的一个高速缓存行中。这个过程，如果有冲突不命中，就会触发行的替换。

## 7.6 hello进程fork时的内存映射

## 当fork函数被shell进程调用时，内核为新进程创建各种数据结构，并分配给它一个唯一的PID，为了给这个新进程创建虚拟内存，它创建了当前进程的mm\_struct、区域结构和页表的原样副本。它将这两个进程的每个页面都标记为只读，并将两个进程中的每个区域结构都标记为私有的写时复制。

## 当fork在新进程中返回时，新进程现在的虚拟内存刚好和调用fork时存在的虚拟内存相同。当这两个进程中的任一个后来进行写操作时，写时复制机制就会创建新页面，因此，也就为每个进程保持了私有地址空间的抽象概念。

## 7.7 hello进程execve时的内存映射

## 运行的当前进程调用了exceve()，其函数形式为：

## execve(“a”, NULL, NULL);

## hello调用execve后，execve在当前进程中加载并运行包含在可执行目标文件hello.out中的程序，用hello程序有效地替代了当前程序。加载并运行hello需要以下几个步骤：

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

## ② 映射私有区域。为新程序的代码、数据、bss和栈区域创建新的区域结构。 所有这些新的区域都是私有的、写时复制的。代码和数据区域被映射为a.out文件 中的.text和.data区。

## ③ 映射共享区域。如果a程序与共享对象(或目标)链接，那么这些对象都是动态链接到这个程序的，然后再映射到用户虚拟地址空间中的共享区域内。

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

## 下一次调度hello进程时，它将从这个入口点开始执行。

## 7.8 缺页故障与缺页中断处理

## 物理内存（DRAM）缓存不命中成为缺页。假设CPU引用了磁盘上的一个字，而这个字所属的虚拟页并未缓存在DRAM中。地址翻译硬件会从内存中读取虚拟页对应的页表，推断出这个虚拟页未被缓存，然后触发一个缺页异常。缺页异常调用内核中的缺页异常处理程序，该程序会选择一个牺牲页。如果被牺牲的页面被修改了，那么内核会把它复制回磁盘。总之，内核会修改被牺牲页的页表条目，表示它不再缓存在DRAM中了。

## 之后，内核从磁盘把本来要读取的那个虚拟页，复制到内存中牺牲页的那个位置，更新它的页表条目，随后返回。当异常处理程序返回时，会重新启动导致缺页的指令，该指令会把导致缺页的虚拟地址重发送到地址翻译硬件。于是，地址翻译硬件可以正常处理现在的页命中了。

## 7.9动态存储分配管理

## 动态内存分配器维护着一个进程的虚拟内存区域，称为堆(heap)系统之间细节不同，但是不失通用性，假设堆是一个请求二进制零的区域，它紧接在未初始化的数据区域后开始，并向上生长（向更高的地址）。对于每个进程，内核维护着一个变量brk(读做"break"),它指向堆的顶部。分配器将堆视为一组不同大小的块(block)的集合来维护。每个块就是一个连续的虚拟内存片(chunk),要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可用来分配。空闲块 保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。分配器有两种基本风格。 两种风格都要求应用显式地分配块。 它们的不同之处在于由哪个实体来负责释放已分配的块。

## 7.10本章小结

在本章中学习了内存管理，讲述了在hello运行的64位系统中内存管理方法，虚拟内存和物理内存之间的关系，了解了intel环境下的段式管理和页式管理， fork和exceve的内存映射，缺页故障和缺页中断管理机制，根据缓存或页表寻找物理内存。**（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：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

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；

## 8.2 简述Unix IO接口及其函数

**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，内核返回一个非负整数，即描述符，它在后续对此文件的所有操作中标识这个文件，内核记录有关文件的所有信息，应用程序只用记住描述符即可。

**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) 。在头文件< unistd.h>中定义。

**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，这个文件位置是从文件开头起始的字节偏移量。

**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”。

**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。无论一个进程因为什么原因终止，内核都会关闭文件并释放它们的内存资源。

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

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

首先看printf的函数体，调用了vsprintf。

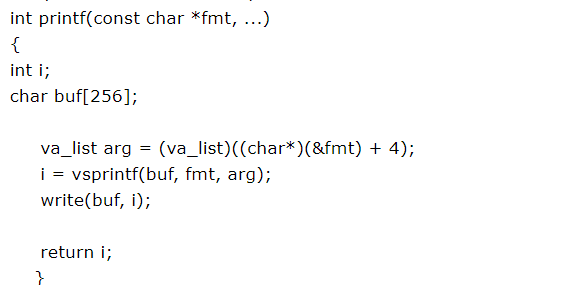
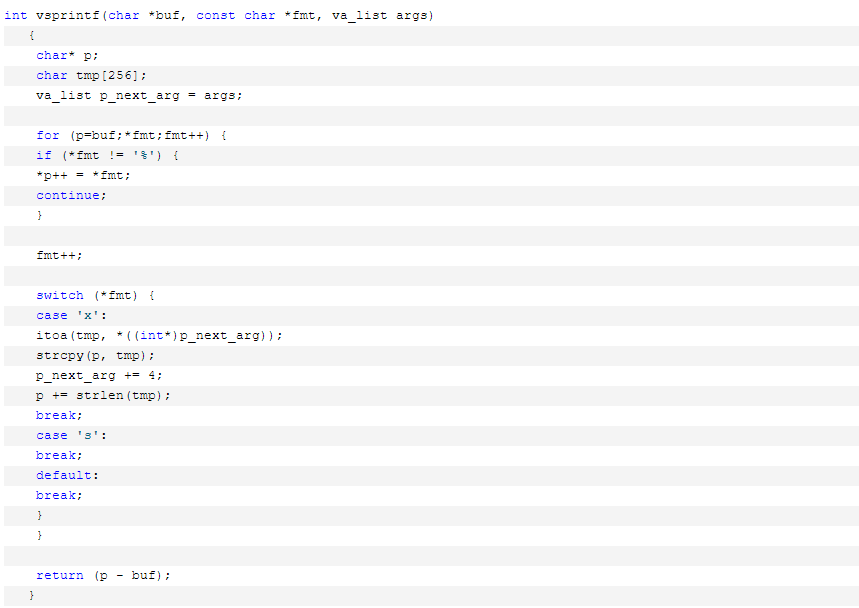


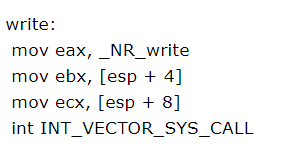
图1.printf

vsprint函数：

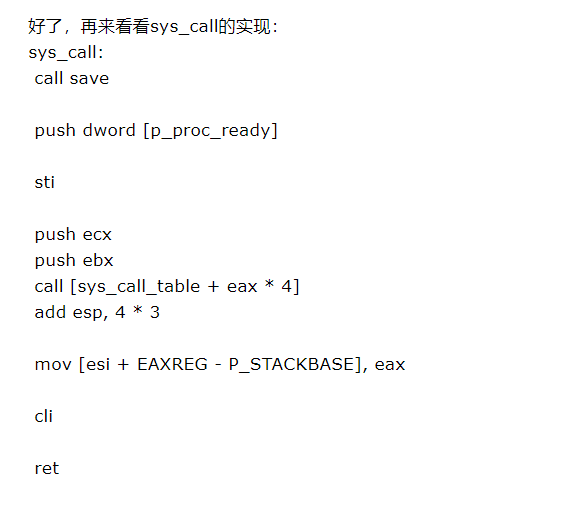


函数描述：vsprintf的作用就是格式化。它接受确定输出格式的格式字符串fmt。用格式字符串对个数变化的参数进行格式化，产生格式化输出。

**对于系统函数write：**



发现它调用了sys\_call这个函数。



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。

**字符显示驱动子程序**：从ASCII到字模库到显示vram（存储每一个点的RGB颜色信息）。

**显示芯片按照刷新频率逐行读取vram**，并通过信号线向液晶显示器传输每一个点（RGB分量）。而我们要传输的“hello 1160100327 谢宇轩”就会被打印输出在显示器上。

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

## 8.4 getchar的实现分析

**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函数看起来简单，其实很复杂。**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

本次大作业，围绕着hello程序的一次运行展开了理解和讨论，在完成的过程中，也是对自己在这一学期所学知识的一个小的总结。从hello诞生到消失，它经历了及其复杂的一生：

编写：用高级语言c完成对它的编写。

预处理：在gcc的作用下生成hello.i，对代码中的头文件、宏定义、条件编译进行了一系列处理。

编译：在gcc的作用下生成hello.s，将高级语言经过一系列的处理最终转化为汇编语言。

链接：在gcc的作用下与可重定位目标文件和其他必要的可重定位目标文件进行链接，生成可执行文件。

运行：在shell中输入运行命令，fork为其创建子进程，exceve调用加载器，进入程序入后程序开始载入物理内存，然后进入main函数。

执行：CPU为其分配时间片，hello执行逻辑控制流。

访问内存：由MMU将虚拟内存映射为物理内存。

申请动态内存：执行printf等函数时，进行动态内存申请

信号处理：在执行程序时，当键入不同指令时，结果不同

终止：父进程回收子进程，内核删除为其创建的所有结构。

# 附件

列出所有的中间产物的文件名，并予以说明起作用。

|  |  |
| --- | --- |
| hello.i | 预处理得到的文件 |
| hello.s | 汇编语言文件 |
| hello.asm | hello.o的反编译文件 |
| hello.o | 可重定位目标文件 |
| hello | 可执行文件 |
| hello\_exe.asm | Hello的反编译文件 |
| hello.elf | hello.o的elf文件 |
| hello\_exe.elf | hello的elf文件 |

**（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] 林来兴. 空间控制技术[M]. 北京：中国宇航出版社，1992：25-42.

[2] 辛希孟. 信息技术与信息服务国际研讨会论文集：A集[C]. 北京：中国科学出版社，1999.

[3] 赵耀东. 新时代的工业工程师[M/OL]. 台北：天下文化出版社，1998 [1998-09-26]. http://www.ie.nthu.edu.tw/info/ie.newie.htm（Big5）.

[4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨：哈尔滨工业大学，1992：8-13.

[5] KANAMORI H. Shaking Without Quaking[J]. Science，1998，279（5359）：2063-2064.

[6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science，1998，281：331-332[1998-09-23]. http://www.sciencemag.org/cgi/ collection/anatmorp.

[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

**（参考文献0分，缺失 -1分）**