

**计算机系统**

**大作业**

题 目 程序人生-Hello’s P2P

专 业 计算机

学　　 号 1170300232

班　　 级 1703002

学 生 赵海莎

指 导 教 师 史先俊

**计算机科学与技术学院**

**2018年12月**

**摘 要**

本论文旨在通过研究hello.c在linux系统下的整个生命周期。通过对hello.c的概述；hello.c的预处理；编译；汇编；链接；一直到hello.c的进程；存储；IO管理。了解hello的一生。通过使用linux操作系统，对hello.c进行执行，用gcc等工具进行实验。对深入理解计算机系统这本书进行总结与贯通,了解整个程序的周期

**关键词：计算机系统**；csapp；hello.c；

**（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 - 5 -](#_Toc532238400)

[第2章 预处理 - 6 -](#_Toc532238401)

[2.1 预处理的概念与作用 - 6 -](#_Toc532238402)

[2.2在Ubuntu下预处理的命令 - 6 -](#_Toc532238403)

[2.3 Hello的预处理结果解析 - 6 -](#_Toc532238404)

[2.4 本章小结 - 8 -](#_Toc532238405)

[第3章 编译 - 9 -](#_Toc532238406)

[3.1 编译的概念与作用 - 9 -](#_Toc532238407)

[3.2 在Ubuntu下编译的命令 - 9 -](#_Toc532238408)

[3.3 Hello的编译结果解析 - 10 -](#_Toc532238409)

[3.4 本章小结 - 16 -](#_Toc532238410)

[第4章 汇编 - 17 -](#_Toc532238411)

[4.1 汇编的概念与作用 - 17 -](#_Toc532238412)

[4.2 在Ubuntu下汇编的命令 - 17 -](#_Toc532238413)

[4.3 可重定位目标elf格式 - 17 -](#_Toc532238414)

[4.4 Hello.o的结果解析 - 20 -](#_Toc532238415)

[4.5 本章小结 - 23 -](#_Toc532238416)

[第5章 链接 - 24 -](#_Toc532238417)

[5.1 链接的概念与作用 - 24 -](#_Toc532238418)

[5.2 在Ubuntu下链接的命令 - 24 -](#_Toc532238419)

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

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

[5.5 链接的重定位过程分析 - 27 -](#_Toc532238422)

[5.6 hello的执行流程 - 28 -](#_Toc532238423)

[5.7 Hello的动态链接分析 - 28 -](#_Toc532238424)

[5.8 本章小结 - 28 -](#_Toc532238425)

[第6章 hello进程管理 - 30 -](#_Toc532238426)

[6.1 进程的概念与作用 - 30 -](#_Toc532238427)

[6.2 简述壳Shell-bash的作用与处理流程 - 30 -](#_Toc532238428)

[6.3 Hello的fork进程创建过程 - 30 -](#_Toc532238429)

[6.4 Hello的execve过程 - 31 -](#_Toc532238430)

[6.5 Hello的进程执行 - 31 -](#_Toc532238431)

[6.6 hello的异常与信号处理 - 31 -](#_Toc532238432)

[6.7本章小结 - 38 -](#_Toc532238433)

[第7章 hello的存储管理 - 39 -](#_Toc532238434)

[7.1 hello的存储器地址空间 - 39 -](#_Toc532238435)

[7.2 Intel逻辑地址到线性地址的变换-段式管理 - 39 -](#_Toc532238436)

[7.3 Hello的线性地址到物理地址的变换-页式管理 - 39 -](#_Toc532238437)

[7.4 TLB与四级页表支持下的VA到PA的变换 - 39 -](#_Toc532238438)

[7.5 三级Cache支持下的物理内存访问 - 40 -](#_Toc532238439)

[7.6 hello进程fork时的内存映射 - 41 -](#_Toc532238440)

[7.7 hello进程execve时的内存映射 - 41 -](#_Toc532238441)

[7.8 缺页故障与缺页中断处理 - 42 -](#_Toc532238442)

[7.9动态存储分配管理 - 42 -](#_Toc532238443)

[7.10本章小结 - 43 -](#_Toc532238444)

[第8章 hello的IO管理 - 44 -](#_Toc532238445)

[8.1 Linux的IO设备管理方法 - 44 -](#_Toc532238446)

[8.2 简述Unix IO接口及其函数 - 44 -](#_Toc532238447)

[8.3 printf的实现分析 - 45 -](#_Toc532238448)

[8.4 getchar的实现分析 - 47 -](#_Toc532238449)

[8.5本章小结 - 47 -](#_Toc532238450)

[结论 - 48 -](#_Toc532238451)

[附件 - 49 -](#_Toc532238452)

[参考文献 - 50 -](#_Toc532238453)

# 第1章 概述

## 1.1 Hello简介

## P2P：From Program to Process

## gcc编译器驱动程序读取程序文件hello.c，然后由预处理器（cpp）根据以#字符开头的命令，修改该程序获得hello.i文件，通过编译器（cll）将文本文件hello.i翻译成文本文件形式的汇编程序hello.s，再通过汇编器（cs）将hello.s翻译成机器语言指令，将指令打包成可重定位的目标文件hello.o，然后通过链接器（ld）合并获得可执行目标文件hello。Linux系统中通过内置命令行解释器shell加载运行hello程序，为hello程序fork进程，至此，hello.c完成了P2P的过程。

## O2O：From Zero-0 to Zero -0

## Shell通过execve在fork产生的子进程中加载hello，先删除当前虚拟地址的用户部分已存在的数据结构，为hello的代码、数据、bss和栈区域创建新的区域结构，然后映射共享区域，设置程序计数器，使之指向代码区域的入口点，进入main函数，CPU为hello分配时间片执行逻辑控制流。hello通过Unix I/O管理来控制输出。hello执行完成后shell会回收hello进程，并且内核会从系统中删除hello所有痕迹，至此，hello完成O2O的 过程。

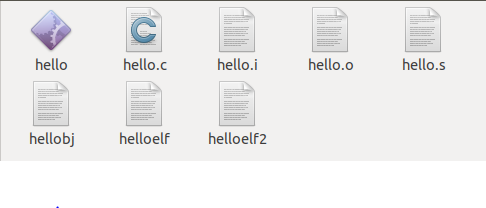
## 1.2 环境与工具

硬件环境：Intel Core i7-6700HQ x64CPU

软件环境：vmware Ubuntu 64位

开发与调试工具：vim，gcc，as，ld，edb，readelf，HexEdit。

## 1.3 中间结果



## hello hello链接后的可执行文件

## hello.c 源代码

## hello.i 预处理生成文件

## hello.o 汇编之后的可重定位目标执行

## hello.s 编译之后的汇编文件

## hellobj Hello.o的反汇编代码

## helloelf Hello.o的ELF格式

## helloelf2 Hello的ELF格式

## 1.4 本章小结

这一章主要了解了p2p和020的概念，并记录了个人软硬件的基本信息，中间文件产物

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

功能：

1.将源文件中用#include形式声明的文件复制到新的程序中，比如类似stdio.h，

Cpp读取相关文件并插入到源程序

2.用实际值替换用#define定义的字符串，就是替换宏定义

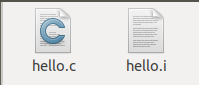
3.根据#if后面的条件决定需要编译的代码

## 2.2在Ubuntu下预处理的命令



应截图，展示预处理过程！

生成文件



## 2.3 Hello的预处理结果解析

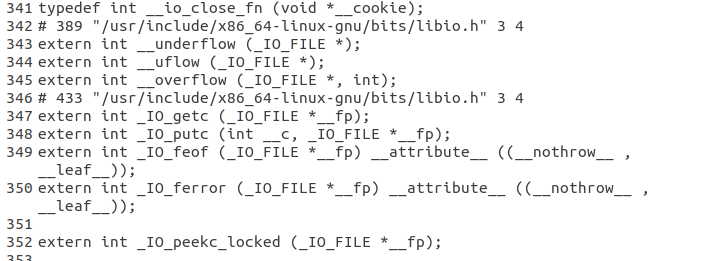
打开 hello.i 之后发现，整个 hello.i 程序已经拓展为 3118 行，main 函数出现在 hello.c 中的代码自 3102 行开始。如下



我们可以发现，其实main函数里面的内容都没太大的变化，变化的是三个头文件，他们直接变成了那三个头文件的源码。虽然长度变长了许多，但是总的来说并不影响程序的可读性（毕竟现在仍然还是c语言的语法）。



用来描述使用的运行库在计算机中的位置。



用来声明可能使用到的函数的名字。

## 2.4 本章小结

本阶段完成了对hello.c的预处理工作。使用Ubuntu下的预处理指令可以将其转换为.i文件。完成该阶段转换后，可以进行下一阶段的汇编处理。

**（第2章0.5分）**

# 第3章 编译

## 3.1 编译的概念与作用

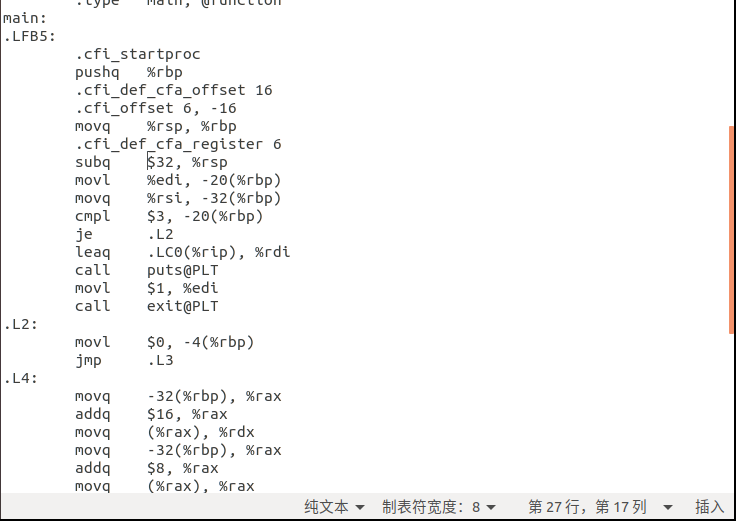
编译程序也称为编译器，是指把用高级程序设计语言书写的源程序，翻译成等价的汇编语言格式目标程序的翻译程序。编译程序属于采用生成性实现途径实现的翻译程序。它以高级程序设计语言书写的源程序作为输入，而以汇编语言表示的目标程序作为输出。

编译程序的基本功能是把源程序（高级语言）翻译成目标程序。除了基本功能之外，编译程序还具备语法检查、调试措施、修改手段、覆盖处理、目标程序优化、不同语言合用以及人机联系等重要功能。

## 3.2 在Ubuntu下编译的命令



生成的汇编程序如下：



## 3.3 Hello的编译结果解析

3.3.0 汇编指令

指令 含义

.file 声明源文件

.text 以下是代码段

.section .rodata 以下是rodata节

.globl 声明一个全局变量

.type 用来指定是函数类型或是对象类型

.size 声明大小

.long、.string 声明一个long、string类型

.align 声明对指令或者数据的存放地址进行对齐的方式

3.3.1 数据

hello.s 中用到的 C 数据类型有：整数、字符串、数组。

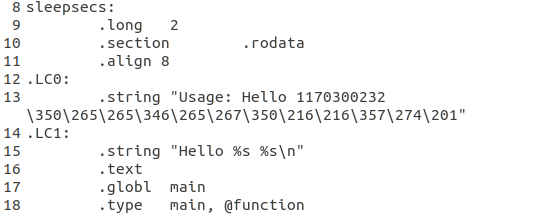
一、 字符串

程序中的字符串分别是：

1）“Usage: Hello 1170300232 赵海莎！ \n”，第一个 printf 传入的输出格式化参数， 在 hello.s 中声明

可以发现字符串被编码成 UTF-8 格式，一个汉字在 utf-8 编码中占三个字节，一个\代表一个字节。

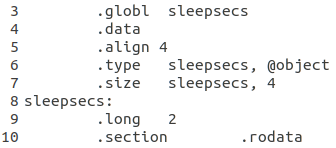
2） “Hello %s %s\n”，第二个 printf 传入的输出格式化参数，在 hello.s 中声明，其中后两个字符串都声明在了.rodata 只读数据节。



1），2）图

二、 整数

1） int sleepsecs：sleepsecs 在 C 程序中被声明为全局变量，且已经被赋值，编译器处理时在.data 节声明该变量，.data 节存放已经初始化的全局和静态 C 变量。在图中，可以看到，编译器首先将 sleepsecs声明为全局变量，其次在.data 段中，设置对齐方式为 4、设置类型为对象、设置大小为 4 字节、设置为 long 类型其值为 2。



2） int i：编译器将局部变量存储在寄存器或者栈空间中，在 hello.s 中编译器将 i 存储在栈上空间-4(%rbp)中，可以看出i占据了栈中的4B



3） int argc：作为第一个参数传入。

4） 立即数：其他整形数据的出现都是以立即数的形式出现的，直接硬编码在汇编代码中。

三、 数组

程序中涉及数组的是：char argv[] main，函数执行时输入的命令行， argv 作为存放char 指针的数组同时是第二个参数传入。 argv 单个元素 char大小为 8B，argv 指针指向已经分配好的、一片存放着字符指针的连续空间，起始地址为 argv，main函数中访问数组元素 argv[1],argv[2]时，按照起始地址 argv大小 8B 计算数据地址取数据，在 hello.s 中，使用两次(%rax)（两次 rax 分别为 argv[1]和 argv[2]的地址）取出其值。

取出argv[1]



取出argv[2]



3.3.2 赋值

程序中涉及的赋值操作有：

1） int sleepsecs=2.5 ：因为 sleepsecs 是全局变量，所以直接在.data 节中 将 sleepsecs 声明为值 2 的 long 类型数据。

2） i=0：整型数据的赋值使用 mov 指令完成，根据数据的大小不同使用不 同后缀，分别为：

指令 b w l q

大小 8b (1B) 16b (2B) 32b (4B) 64b (8B)

因为 i 是 4B 的 int 类型，所以使用 movl 进行赋值，汇编代码如图



3.3.3 类型转换

程序中涉及隐式类型转换的是：int sleepsecs=2.5，将浮点数类型的 2.5 转换为 int 类型。

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1.999 将被转换成1，-1.999 将被转换成-1。进一步来讲，可能会产生值溢出的情况，与 Intel 兼容的微处理器指定位模式[10…000]为整数不确定值，一个浮点数到整数的转换，如果不能为该浮点数找到一个合适的整数近似值，就会产生一个整数不确定值。

浮点数默认类型为 double，所以上述强制转化是double 强制转化为 int 类型。 遵从向零舍入的原则，将2.5舍入为 2。

3.3.4 算数操作

进行数据算数操作的汇编指令有：

指令 效果

leaq S,D D=&S

INC D D+=1

DEC D D-=1

NEG D D=-D

ADD S,D D=D+S

SUB S,D D=D-S

IMULQ S R[%rdx]:R[%rax]=SR[%rax]（有符号）

MULQ S R[%rdx]:R[%rax]=SR[%rax]（无符号）

IDIVQ S R[%rdx]=R[%rdx]:R[%rax] mod S（有符号）

R[%rax]=R[%rdx]:R[%rax] div S

DIVQ S R[%rdx]=R[%rdx]:R[%rax] mod S（无符号）

R[%rax]=R[%rdx]:R[%rax] div S

程序中涉及的算数操作有：

1） i++，对计数器 i 自增，使用程序指令 addl，后缀 l 代表操作数是一个 4B 大小的数据。



3.3.5 关系操作

进行关系操作的汇编指令有：

指令 效果 描述

CMP S1,S2 S2-S1 比较-设置条件码

TEST S1,S2 S1&S2 测试-设置条件码

SET\*\* D D=\*\* 按照将条件码设置 D

J —— 根据\*\*与条件码进行跳转

程序中涉及的关系运算为：

1） argc!=3：判断 argc 不等于 3。hello.s 中使用 cmpl $3,-20(%rbp)，计算 argc-3 然后设置条件码，为下一步 je 利用条件码进行跳转作准备。



2） i<10：判断 i 小于 10。hello.s 中使用 计算 i-9 然后设置条件码，为下一步 jle 利用条件码进行跳转做准备。



3.3.6 控制转移

程序中涉及的控制转移有：



1） if (argv!=3)：当 argv不等于3 的时候执行程序段中的代码。对于 if 判断，编译器使用跳转指令实现，首先 cmpl 比较 argv 和 3，设置条件码，使用 je 判断 ZF 标志位，如果为 0，说明 argv-3=0

argv==3，则不执行 if 中的代码直接跳转到.L2，



否则顺序执行下一条语句，即执行 if 中的代码。

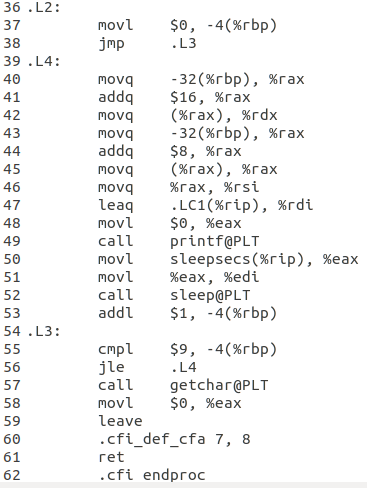


2） for(i=0;i<10;i++)：使用计数变量 i 循环 10 次。编译器的编译逻辑是，首先无条件跳转到位于循环体.L4 之后的比较代码，使用 cmpl 进行比较，如果 i<=9，则跳入.L4 for 循环体执行，否则说明循环结束，顺序执行 for 之后的逻辑。

I小于10时跳转到L4，执行for循环

循环内部

无条件跳转到L3



3.3.7 函数操作

函数是一种过程，过程提供了一种封装代码的方式，用一组指定的参数和可选的返回值实现某种功能。P 中调用函数 Q 包含以下动作：

1） 传递控制：进行过程 Q 的时候，程序计数器必须设置为 Q 的代码的起始地址，然后在返回时，要把程序计数器设置为 P 中调用 Q 后面那条指令的地址。

2） 传递数据：P 必须能够向 Q 提供一个或多个参数，Q 必须能够向 P 中返回一个值。

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

64 位程序参数存储顺序（浮点数使用 xmm，不包含）：

1 2 3 4 5 6 7

%rdi %rsi %rdx %rcx %r8 %r9 栈空间

程序中涉及函数操作的有：

1） main 函数：

a) 传递控制，main 函数因为被调用 call 才能执行（被系统启动函数 \_\_libc\_start\_main 调用），call 指令将下一条指令的地址 dest 压栈， 然后跳转到 main 函数。

b) 传递数据，外部调用过程向 main 函数传递参数 argc 和 argv，分别 使用%rdi 和%rsi 存储，函数正常出口为 return 0，将%eax 设置 0 返回。

c) 分配和释放内存，使用%rbp 记录栈帧的底，函数分配栈帧空间 在%rbp 之上，程序结束时，调用 leave 指令，leave 相当于 mov %rbp,%rsp,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。

2） printf 函数：

a) 传递数据：第一次 printf 将%rdi 设置为“Usage: Hello 学号 姓名！ \n”字符串的首地址。第二次 printf 设置%rdi 为“Hello %s %s\n” 的首地址，设置%rsi 为 argv[1]，%rdx 为 argv[2]。

b) 控制传递：第一次 printf 因为只有一个字符串参数，所以 call puts@PLT；第二次 printf 使用 call printf@PLT。

3） exit 函数：

a) 传递数据：将%edi 设置为 1。

b) 控制传递：call exit@PLT。

4） sleep 函数：

a) 传递数据：将%edi 设置为 sleepsecs。

b) 控制传递：call sleep@PLT。

5） getchar 函数：

a) 控制传递：call gethcar@PLT

此部分是重点，说明编译器是怎么处理C语言的各个数据类型以及各类操作的。应分3.3.1~ 3.3.x等按照类型和操作进行分析，**只要hello.s中出现的属于大作业PPT中P4给出的参考C数据与操作，都应解析**。

## 3.4 本章小结

编译器将.i 的拓展程序编译为.s 的汇编代码。经过编译之后，我们的 hello 自 C 语言解构为更加低级的汇编语言。

本章介绍了编译的概念与作用，通过对hello.s的解析，使学生能够读懂汇编语句，了解汇编语句下：数据：常量、变量(全局/局部/静态)、表达式、类型、宏，赋值语句，类型转换，算数操作，逻辑/位操作，关系操作，控制转移以及函数操作的基本知识。

**（第3章2分）**

# 第4章 汇编

## 4.1 汇编的概念与作用

汇编程序是把汇编语言书写的程序翻译成与之等价的机器语言程序的翻译程序。汇编程序输入的是用汇编语言书写的源程序，输出的是用机器语言表示的目标程序。汇编语言的指令与机器语言的指令大体上保持一一对应的关系，汇编算法采用的基本策略是简单的。通常采用两遍扫描源程序的算法。第一遍扫描源程序根据符号的定义和使用，收集符号的有关信息到符号表中；第二遍利用第一遍收集的符号信息，将源程序中的符号化指令逐条翻译为相应的机器指令。

## 4.2 在Ubuntu下汇编的命令



## 4.3 可重定位目标elf格式

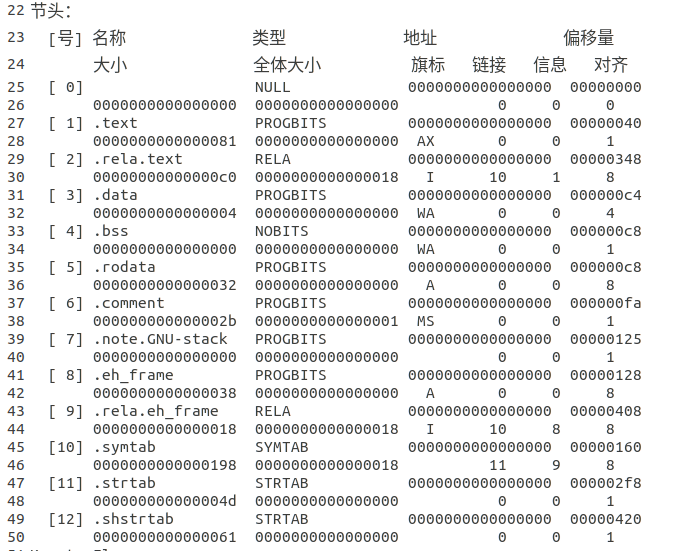
使用 readelf -a hello.o > helloelf 指令获得 hello.o 文件的 ELF 格式。

内容如下：

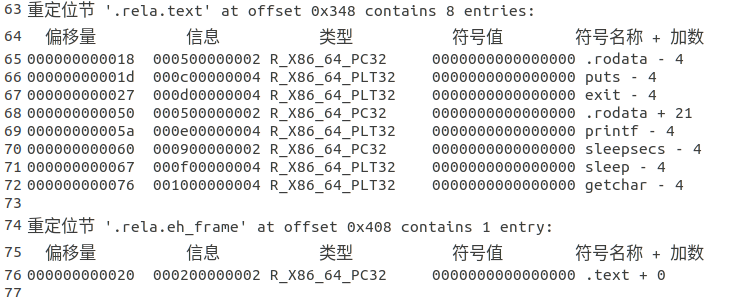
1） ELF Header：以 16B 的序列 Magic 开始，Magic 描述了生成该文件的系统的字的大小和字节顺序，ELF 头剩下的部分包含帮助链接器语法分析和解 释目标文件的信息，其中包括 ELF 头的大小、目标文件的类型、机器类型、 字节头部表（section header table）的文件偏移，以及节头部表中条目的大小和数量等信息



2） Section Headers：节头部表，包含了文件中出现的各个节的语义，包括节 的类型、位置和大小等信息。



3） 重定位节.rela.text ,一个.text 节中位置的列表，包含.text 节中需要进行重定位的信息，当链接器把这个目标文件和其他文件组合时，需要修改这些位置。如图 4.4，图中 8 条重定位信息分别是对.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 函数进行重定位声明。



.rela节的包含的信息有：

offset 需要进行重定向的代码在.text 或.dat节中的偏移位置，8 个字节。

Info 包括 symbol 和 type 两部分， 其中 symbol 占前 4 个字节， type 占后 4 个字节，symbol 代表重定位到的目标在.symtab 中的偏移量，type 代表重定位 的类型

Addend 计算重定位位置的辅助信息， 共占 8 个字节

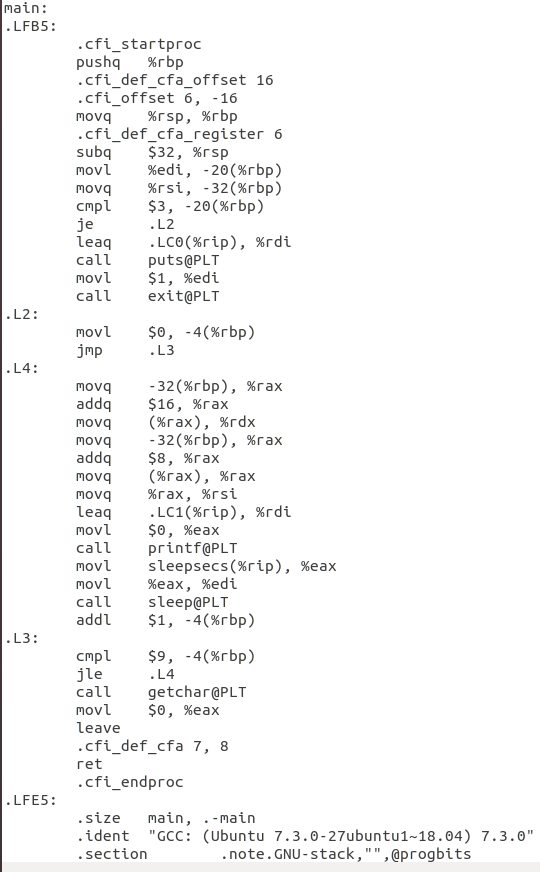
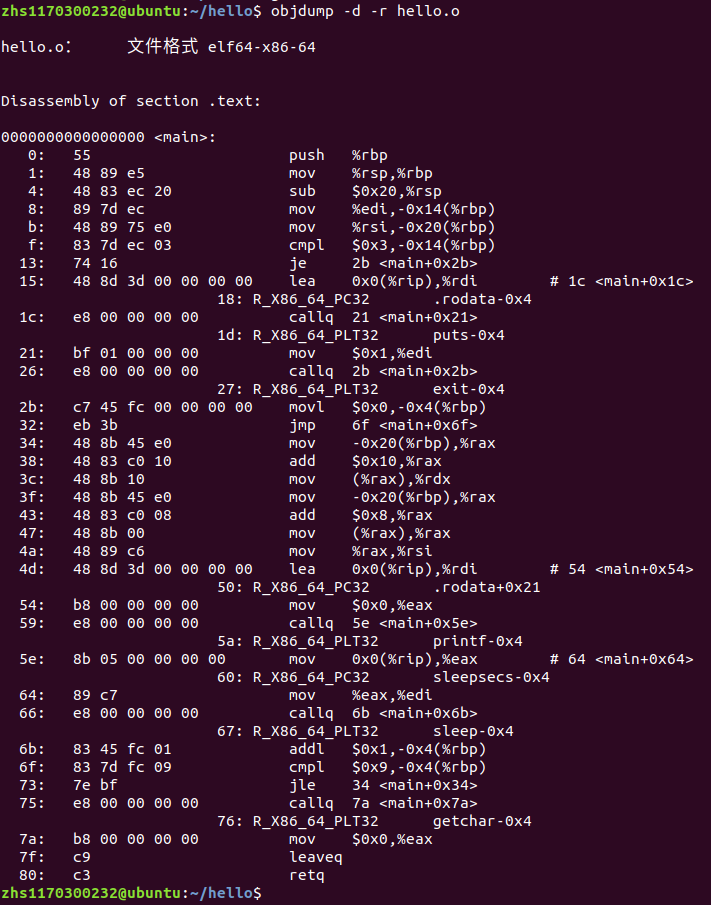
Type 重定位到的目标的类型

Name 重定向到的目标的名称

4）重定位节 ‘.rela.eh\_frame’ ：eh\_frame 节的重定位信息

（见上图）

## 4.4 Hello.o的结果解析



上图为objdump -d -r hello.o hello.o的反汇编，与第3章的 hello.s对照

1）分支转移：反汇编代码跳转指令的操作数使用的不是段名称如.L3，因为段 名称只是在汇编语言中便于编写的助记符，所以在汇编成机器语言之后显 然不存在，而是确定的地址。

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

3）全局变量访问：在.s文件中，访问 rodata（printf 中的字符串），使用段名称+%rip，在反汇编代码中0+%rip，因为 rodata 中数据地址也是在运行时确定，故访问也需要重定位。所以在汇编成为机器语言时，将操作数设置为全 0 并添加重定位条目。

## 4.5 本章小结

本阶段完成了对hello.s的汇编工作。使用Ubuntu下的汇编指令可以将其转换为.o可重定位目标文件。此外，本章通过将.o文件反汇编结果与.s汇编程序代码进行比较，了解了二者之间的差别。完成该阶段转换后，可以进行下一阶段的链接工作。

**（第4章1分）**

# 第5章 链接

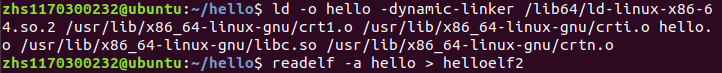
## 5.1 链接的概念与作用

链接是将各种代码和数据片段收集并组合成一个单一文件的过程，这个文件可被加载到内存并执行。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。链接是由叫做链接器的程序执行的。链接器使得分离编译成为可能。

注意：这儿的链接是指从 hello.o 到hello生成过程。

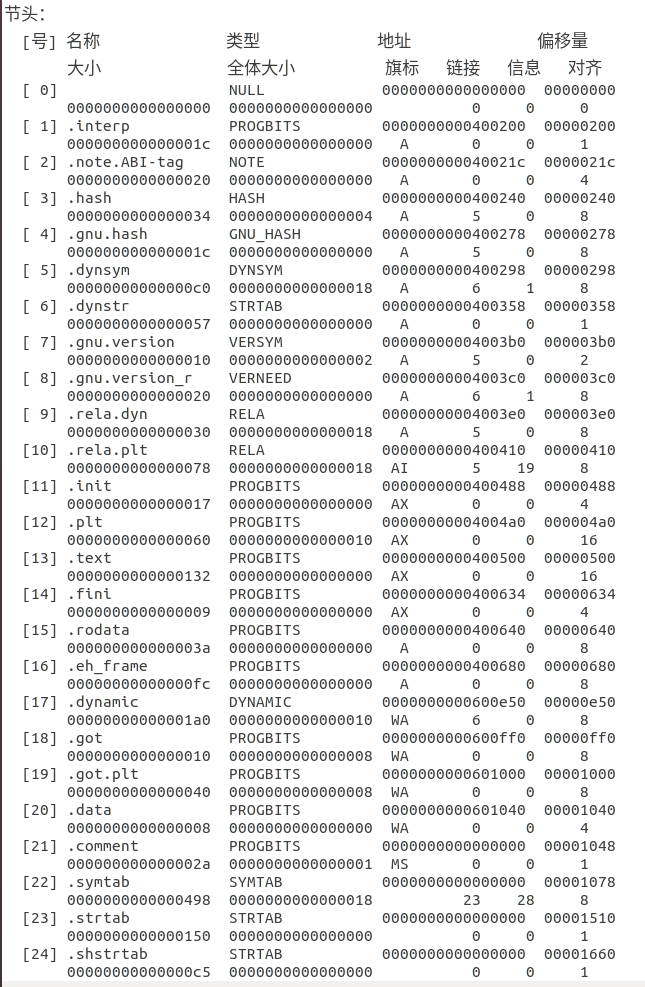
## 5.2 在Ubuntu下链接的命令

使用ld的链接命令，应截图，展示汇编过程！ 注意不只连接hello.o文件



## 5.3 可执行目标文件hello的格式

使用 readelf -a hello > helloelf2命令生成 hello 程序的 ELF 格式文件。（上图）

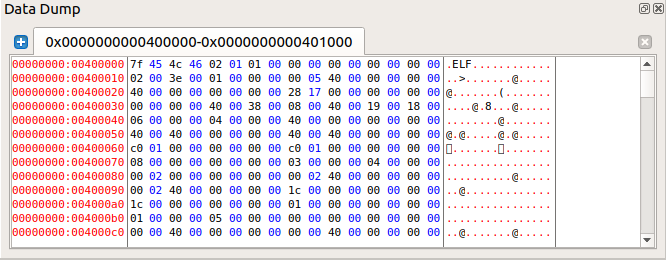


根据上图可以得到各段的基本信息。由于是可执行目标文件，所以每个段的起始地址都不相同，它们的起始地址分别对应着装载到虚拟内存中的虚拟地址。这样可以直接从文件起始处得到各段的起始位置，以及各段所占空间的大小。同时可以观察到，代码段是可执行的，但是不能写；数据段和只读数据段都不可执行，而且只读数据段也不可写

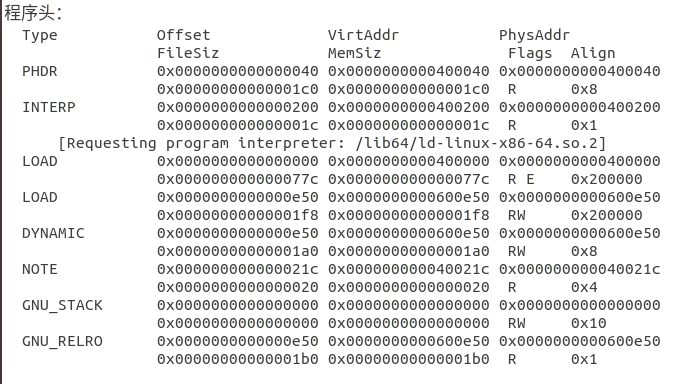
分析hello的ELF格式，用readelf等列出其各段的基本信息，包括各段的起始地址，大小等信息。

## 5.4 hello的虚拟地址空间

用edb打开hello，可以在Data Dump窗口看见hello加载到虚拟地址中的状况：



在 0x400000~0x401000 段中，程序被载入，自虚拟地址 0x400000 开始



再来分析分析elf里面的Program Headers：

PHDR：程序头表

INTERP：程序执行前需要调用的解释器

LOAD：程序目标代码和常量信息

DYNAMIC：动态链接器所使用的信息

NOTE:：辅助信息

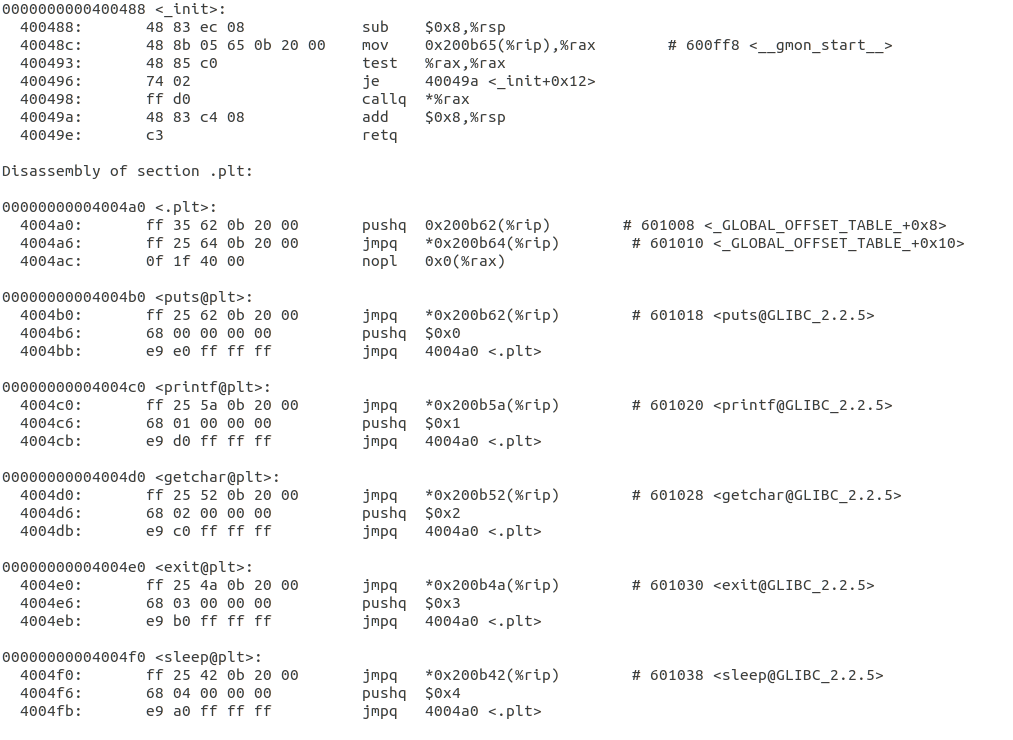
GNU\_EH\_FRAME：保存异常信息

GNU\_STACK：使用系统栈所需要的权限信息

GNU\_RELRO：保存在重定位之后只读信息的位置

## 5.5 链接的重定位过程分析

（*以下格式自行编排，编辑时删除*）



hello.o中的相对偏移地址到了hello中变成了虚拟内存地址

hello中相对hello.o增加了许多的外部链接来的函数。

hello相对hello.o多了很多的节类似于.init,.plt等

hello.o中跳转以及函数调用的地址在hello中都被更换成了虚拟内存地址。

重定位：链接器在完成符号解析以后，就把代码中的每个符号引用和正好一个符号定义（即它的一个输入目标模块中的一个符号表条目）关联起来。此时，链接器就知道它的输入目标模块中的代码节和数据节的确切大小。然后就可以开始重定位步骤了，在这个步骤中，将合并输入模块，并为每个符号分配运行时的地址。在hello.o到hello中，首先是重定位节和符号定义，链接器将所有输入到hello中相同类型的节合并为同一类型的新的聚合节。例如，来自所有的输入模块的.data节被全部合并成一个节，这个节成为hello的.data节。然后，链接器将运行时内存地址赋给新的聚合节，赋给输入模块定义的每个节，以及赋给输入模块定义的每一个符号。当这一步完成时，程序中的每条指令和全局变量都有唯一的运行时内存地址了。然后是重定位节中的符号引用，链接器会修改hello中的代码节和数据节中对每一个符号的引用，使得他们指向正确的运行地址。

objdump -d -r hello 分析hello与hello.o的不同，说明链接的过程。

结合hello.o的重定位项目，分析hello中对其怎么重定位的。

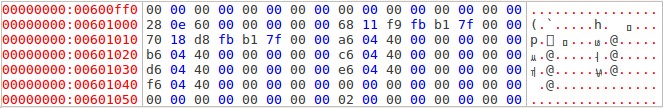
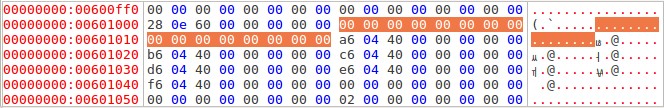
## 5.6 hello的执行流程

通过使用objdump查看反汇编代码，以及使用edb单步运行，可以找出.text节中main函数前后执行的函数名称。在main函数之前执行的程序有：\_start、\_\_libc\_start\_main@plt、\_\_libc\_csu\_init、\_init、frame\_dummy、register\_tm\_clones。在main函数之后执行的程序有：exit、cxa\_thread\_atexit\_impl、fini。

使用edb执行hello，说明从加载hello到\_start，到call main,以及程序终止的所有过程。请列出其调用与跳转的各个子程序名或程序地址。

## 5.7 Hello的动态链接分析

在dl\_init前后 \_GLOBAL\_OFFSET\_TABLE\_的变化



分析hello程序的动态链接项目，通过edb调试，分析在dl\_init前后，这些项目的内容变化。要截图标识说明。

## 5.8 本章小结

本章主要介绍了链接的概念与作用：是将各种代码和数据片段收集并组合成为一个单一文件的过程，分析了UBUNTU下链接的命令，通过对hello的反汇编了解了链接的中间过程，所做的事及函数的调用。了解了链接前后地址的变化以及调用函数的变化。

**（第5章1分）**

# 第6章 hello进程管理

## 6.1 进程的概念与作用

## 进程：进程的经典定义就是一个执行中程序的实例。简单来说，进程是程序的一次运行过程，更确切地说，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，具有动态含义。

## 作用：“进程”的引入为应用程序提供了以下两方面的抽象：一个独立的逻辑控制流和一个私有的虚拟地址空间。每个进程拥有一个独立的逻辑控制流，使得程序员以为自己的程序在执行过程中独占使用处理器；每个进程拥有一个私有的虚拟地址空间，使得程序员以为自己的程序在执行过程中独占存储器。

## 6.2 简述壳Shell-bash的作用与处理流程

## 功能：1.它接收用户命令，能解释用户输入的命令，将它传递给内核，还可以然后调用相应的应用程序

## 2.调用其他程序，给其他程序传递数据或参数，并获取程序的处理结果；

## 3.在多个程序之间传递数据，把一个程序的输出作为另一个程序的输入；

## 4.Shell 本身也可以被其他程序调用。

## 流程：1.shell命令行解释器输出命令行提示符，接受用户命令

## 2.解析命令，构建argv,envp参数列表和参数个数argc

## 3. 如果是内置命令则立即执行，否则fork子进程

## 4.以构建的argc,arhv,envp为参数调用execve以启动加载器，从而在当前进程上下文中加载并运行程序

## 5.shell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

过程： 1.shell命令行解释器输出命令行提示符，接受用户命令

2.解析命令，构建argv,envp参数列表和参数个数argc

3.fork子进程，新创建的子进程几乎但不完全与父进程相同，子进程得到与父进程用户级虚拟地址空间相同的（但是独立的）一份副本，这就意味着，当父进程调用fork时，子进程可以读写父进程中打开的任何文件。父进程与子进程之间最大的区别在于它们拥有不同的PID。

4.以构建的argc,arhv,envp为参数调用execve以启动加载器，从而在当前进程上下文中加载并运行程序

## 6.4 Hello的execve过程

使用execve就是一次系统调用，首先要做的将新的可执行文件的绝对路径从调用者（用户空间）拷贝到系统空间中。在得到可执行文件路径后，就找到可执行文件打开，由于操作系统已经为可执行文件设置了一个数据结构，就初始化这个数据结构，保存一个可执行文件必要的信息。可执行文件不是真正上能够自己运行的，需要有代理人来代理。在系统内核中有一个formats队列，循环遍历这个队列，看看现在被初始化的这个数据结构是哪个代理人可以代理的。如果没有就继续查看数据结构中的信息。按照系统配置了是否可以动态加载模块，加载一次模块，再循环遍历看是否有代理人前来认领。找到正确的代理人后，代理人首先要做的就是放弃以前从父进程继承来的资源。主要是对信号处理表，用户空间和文件大资源的处理。将父进程的信号处理表复制过来，放弃原来的用户空间。然后载入真正的程序代码和数据段，开辟堆栈，映射执行参数和环境变量。

## 6.5 Hello的进程执行

## 逻辑控制流：一系列程序计数器 PC 的值的序列叫做逻辑控制流，进程是轮流使用处理器的，在同一个处理器核心中，每个进程执行它的流的一部分后被抢占（暂时挂起），然后轮到其他进程。

## 时间片：一个进程执行它的控制流的一部分的每一时间段叫做时间片。

## 用户模式和内核模式：处理器通常使用一个寄存器提供两种模式的区分，该寄存器描述了进程当前享有的特权，当没有设置模式位时，进程就处于用户模式中，用户模式的进程不允许执行特权指令，也不允许直接引用地址空间中内核区内的代码和数据；设置模式位时，进程处于内核模式，该进程可以执行指令集中的任何命令，并且可以访问系统中的任何内存位置。

## 上下文信息：上下文就是内核重新启动一个被抢占的进程所需要的状态，它由通用寄存器、浮点寄存器、程序计数器、用户栈、状态寄存器、内核栈和各种内核数据结构等对象的值构成。

## 简单看 hello sleep 进程调度的过程：当调用sleep之前，如果hello程序不被抢占则顺序执行，假如发生被抢占的情况，则进行上下文切换，上下文切换是由内核中调度器完成的，当内核调度新的进程运行后，它就会抢占当前进程，并进行1） 保存以前进程的上下文2）恢复新恢复进程被保存的上下文，3）将控制传递给这个新恢复的进程，来完成上下文切换。

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

## 当 hello 调用 getchar 的时候，实际落脚到执行输入流是 stdin 的系统调用 read， hello之前运行在用户模式，在进行 read 调用之后陷入内核，内核中的陷阱处理程序请求来自键盘缓冲区的 DMA 传输，并且安排在完成从键盘缓冲区到内存的数据传输后，中断处理器。此时进入内核模式，内核执行上下文切换，切换到其他进程。当完成键盘缓冲区到内存的数据传输时，引发一个中断信号，此时内核从其他进程进行上下文切换回 hello进程。

## 6.6 hello的异常与信号处理

hello执行过程中会出现的异常种类有：

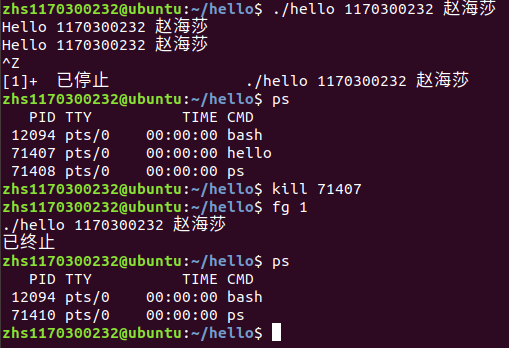
中断：SIGSTP：挂起程序

终止：SIGINT:终止程序

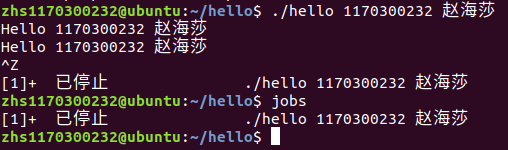
1）在程序输出时乱按的结果，可以发现，乱按只是将屏幕的输入缓存到 stdin，当 getchar 的时候读出一个’\n’结尾的字串（作为一次输入），其他字串会当做 shell 命令行输入：



2）当按下ctrl-z之后，shell父进程收到SIGSTP信号，信号处理程序将hello进程挂起，放到后台，ps看到hello进程并没有被回收，kill杀死进程：



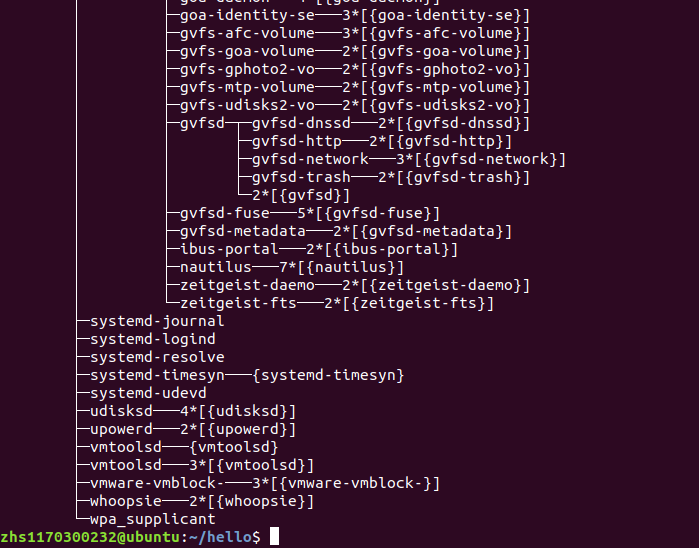
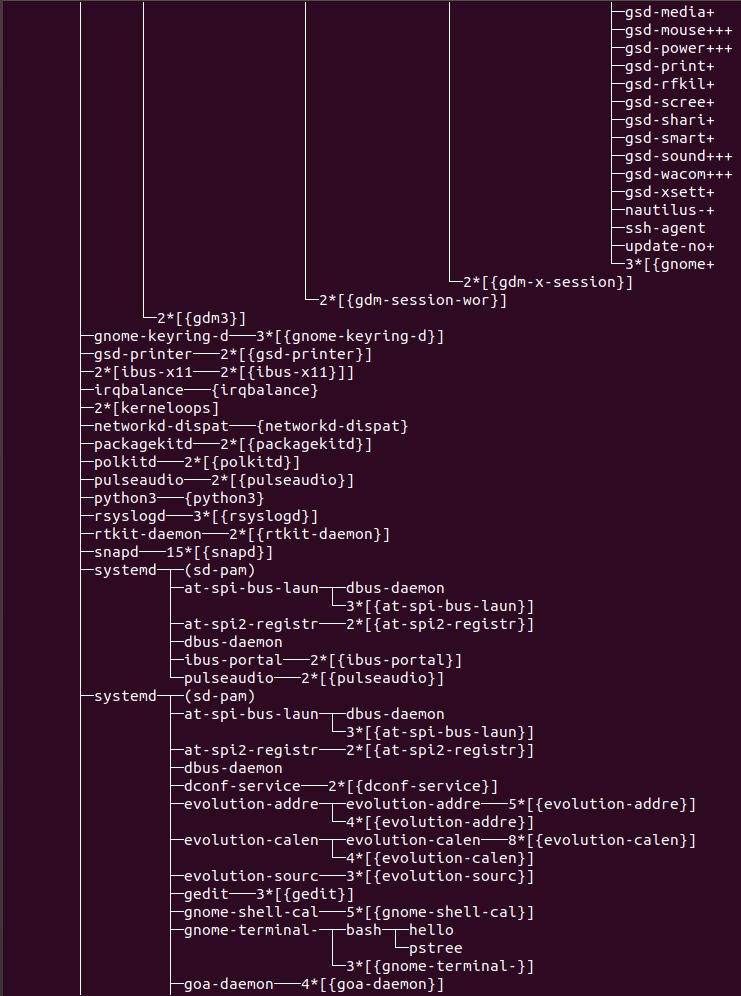
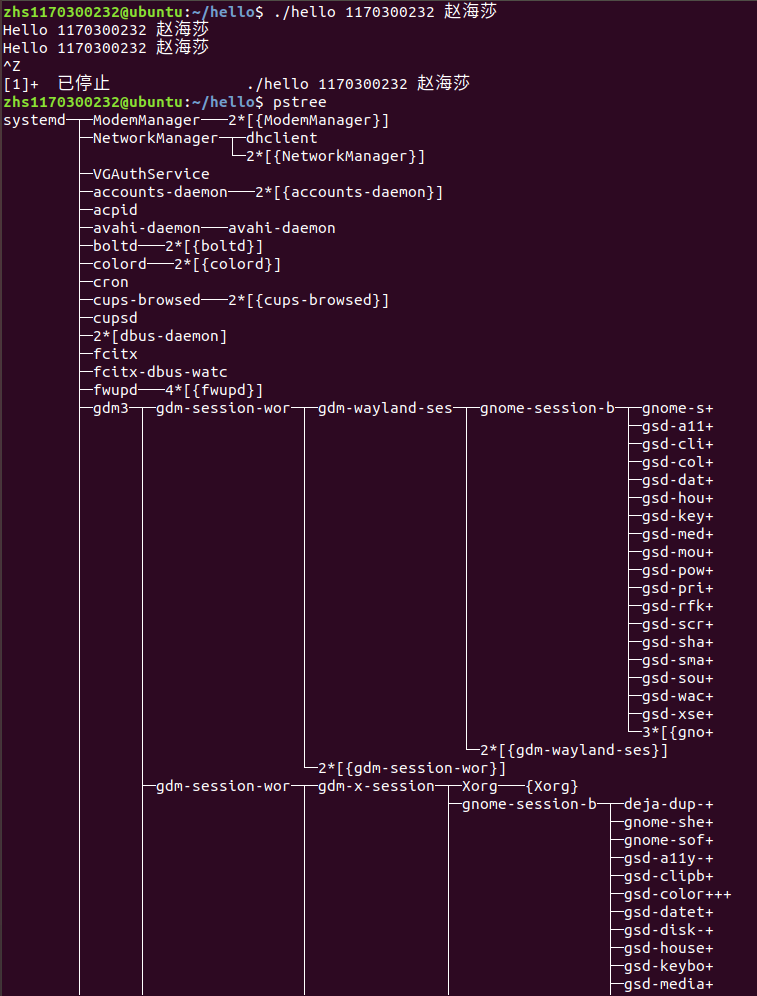
不杀死进程，jobs找到hello的job号：



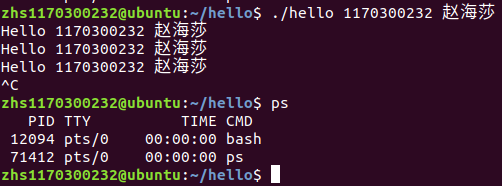
不杀死进程，fg 将hello进程调到前台执行：



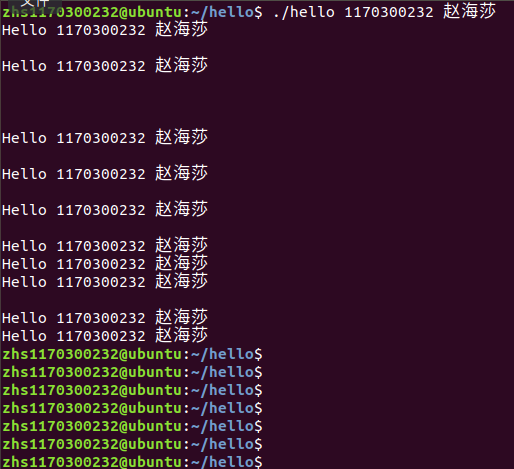
不杀死进程，输入pstree：



3）程序运行时按下 ctrl-c 的结果，当按下 ctrl-c 之后，shell 父进程收到 SIGINT 信号，信号处理函数的逻辑是结束 hello，并回收 hello 进程：



4）程序运行时按下回车的结果，与乱按类似：



## 6.7本章小结

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，每个进程都有其独立的逻辑控制流和私有的虚拟地址空间。操作系统通过进程的处理器调度，进行上下文切换，使得系统在当前进程的执行过程中发生了一个异常控制流。对于进程运行中的各种异常，相应的异常处理程序会向当前进程发送一个特定的信号，当前进程接收信号后调用相应的信号处理程序进行处理。

**（第6章1分）**

# 第7章 hello的存储管理

## 7.1 hello的存储器地址空间

物理地址是用于内存芯片级的单元寻址，与处理器和CPU连接的地址总线相对应。现代操作系统都提供了一种内存管理的抽像，即虚拟内存。进程使用虚拟内存中的地址，即虚拟地址，由操作系统协助相关硬件，把它“转换”成真正的物理地址。hello.s中使用的就是虚拟空间的虚拟地址。线性地址指虚拟地址到物理地址变换的中间层，是处理器可寻址的内存空间（称为线性地址空间）中的地址。程序代码会产生逻辑地址，或者说段中的偏移地址，加上相应段基址就成了一个线性地址。如果启用了分页机制，那么线性地址可以再经过变换产生物理地址。若是没有采用分页机制，那么线性地址就是物理地址。而逻辑地址指的是机器语言指令中，用来指定一个操作数或者是一条指令的地址。它是Intel为了兼容，而将段式内存管理方式保留下来的产物。

逻辑（虚拟）地址经过分段（查询段表）转化为线性地址。线性地址经过分页（查询页表）转为物理地址。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

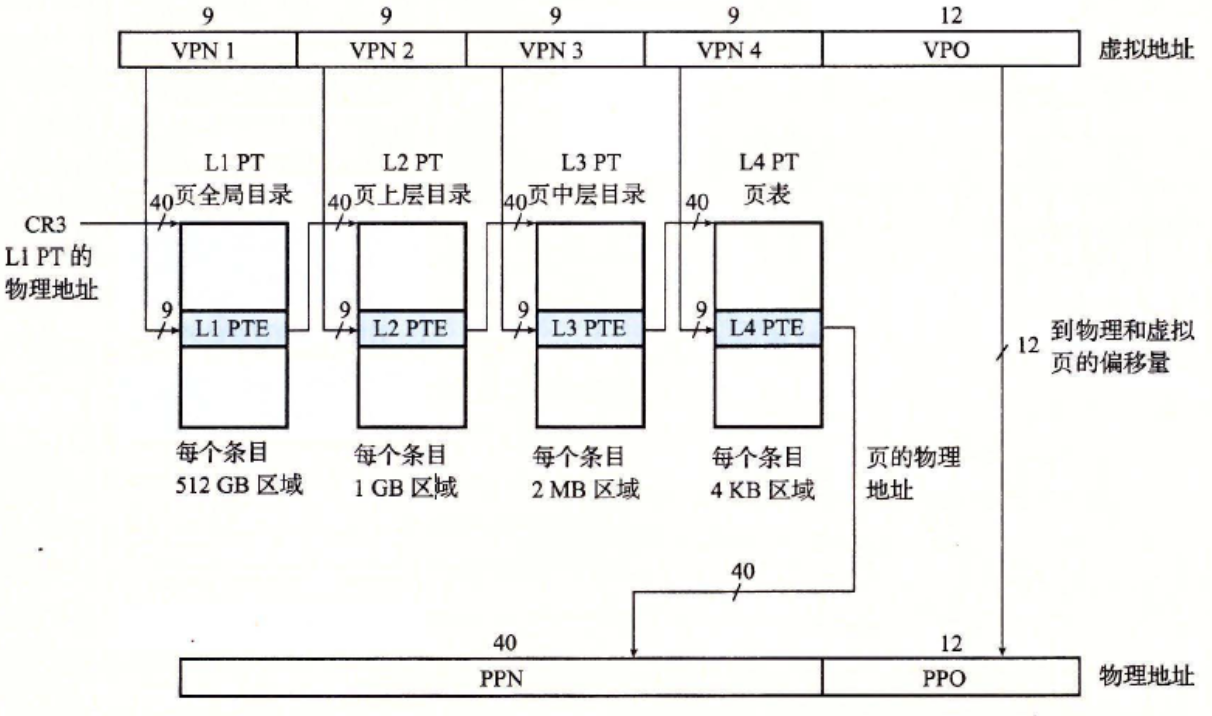
## 一个逻辑地址由两部分组成，段标识符和段内偏移量。段标识符是由一个16位长的字段组成，称为段选择符。其中前13位是一个索引号。后面3位包含一些硬件细节。可以通过段标识符的前13位，直接在段描述符表中找到一个具体的段描述符，这个描述符就描述了一个段。一些全局的段描述符，就放在“全局段描述符表(GDT)”中，一些局部的，例如每个进程自己的，就放在所谓的“局部段描述符表(LDT)”中。

## 7.3 Hello的线性地址到物理地址的变换-页式管理

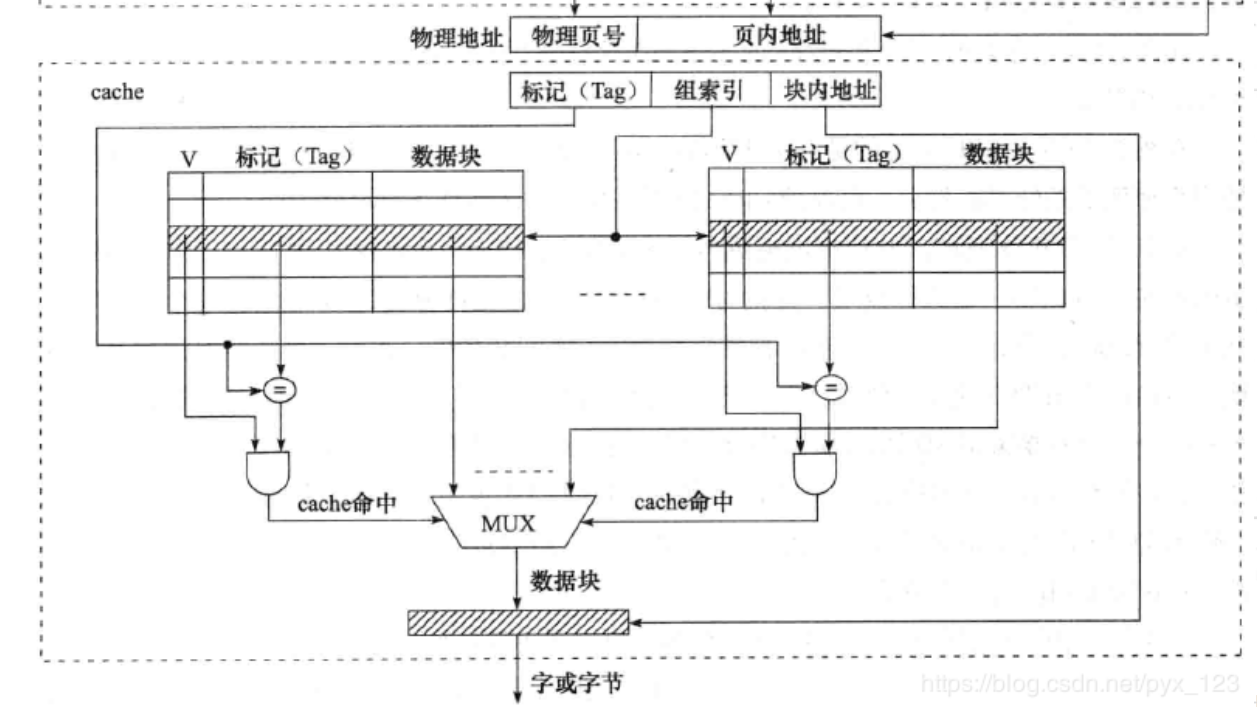
## CPU的页式内存管理单元，负责把一个线性地址，最终翻译为一个物理地址。从管理和效率的角度出发，线性地址被分为以固定长度为单位的组，称为页(page)，例如一个32位的机器，线性地址最大可为4G，可以用4KB为一个页来划分，这页，整个线性地址就被划分为一个tatol\_page[2^20]的大数组，共有2的20个次方个页。这个大数组我们称之为页目录。目录中的每一个目录项，就是一个地址——对应的页的地址。另一类“页”，我们称之为物理页，或者是页框、页桢的。是分页单元把所有的物理内存也划分为固定长度的管理单位，它的长度一般与内存页是一一对应的。

## 7.4 TLB与四级页表支持下的VA到PA的变换

首先将VPN分成三段，对于TLBT和TLBI来说，如果可以在TLB中找到对应的PPN的话那肯定是最好不过的了，但是还有可能出现缺页的情况，这时候就需要到页表中去找。此时，VPN被分成了更多段（这里是4段）CR3是对应的L1PT的物理地址，然后一步步递进往下寻址，越往下一层每个条目对应的区域越小，寻址越细致，在经过4层寻址之后找到相应的PPN让你和和VPO拼接起来。（如图）



## 7.5 三级Cache支持下的物理内存访问



讨论组相联下的读取，其他类似。

得到物理地址后，根据cache块大小和行(组)数将物理地址分为标记位，组索引和块内地址，如果组索引下的cache行标记位相同，有效位为1，则命中，读出cache行中第块内地址个处的字节。如果不命中，则向下一级缓存（或主存）取相应的cache行（主存行），此时如果原来的组中有空闲行，则直接替换，如果没有空闲行，则需要替换，常用的替换策略有随机替换，LRU，LFU等。

## 7.6 hello进程fork时的内存映射

虚拟内存和内存映射解释了fork函数如何为每个新进程提供私有的虚拟地址空间。Fork函数为新进程创建虚拟内存。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）。在新进程中返回时，新进程拥有与调用fork进程相同的虚拟内存，随后的写操作通过写时复制机制创建新页面。

## 7.7 hello进程execve时的内存映射

execve 函数在shell中加载并运行包含在可执行目标文件hello中的程序，用hello程序有效地替代了当前程序。加载并运行hello需要以下几个步骤：

1） 删除已存在的用户区域。删除shell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hello的代码、数据、bss 和栈区域创建新的区域结构。所有这些新的区域都是私有的、写时复制的。代码和数据区域被映射为hello 文件中的.text和.data 区。bss 区域是请求二进制零的，映射到匿名文件，其大小包含在hello 中。栈和堆区域也是请求二进制零的，初始长度为零。

3） 映射共享区域。如果hello程序与共享对象（或目标）链接，比如标准C 库libc. so, 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，使之指向代码区域的入口点。

## 7.8 缺页故障与缺页中断处理

缺页故障是一种常见的故障，当指令引用一个虚拟地址，在 MMU 中查找页表 时发现与该地址相对应的物理地址不在内存中，因此必须从磁盘中取出的时候就 会发生故障。

缺页中断处理：缺页处理程序是系统内核中的代码，选择一个牺牲页面，如果这个牺牲页面被修改过，那么就将它交换出去，换入新的页面并更新页表。当缺页处理程序返回时，CPU 重新启动引起缺页的指令，这条指令再次发送 VA 到 MMU，这次 MMU 就能正常翻译 VA 了。

## 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, 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。

基本方法：这里指的基本方法应该是在合并块的时候使用到的方法，有最佳适配和第二次适配还有首次适配方法，首次适配就是指的是第一次遇到的就直接适配分配，第二次顾名思义就是第二次适配上的，最佳适配就是搜索完以后最佳的方案，当然这种的会在搜索速度上大有降低。

策略：这里的策略指的就是显式的链表的方式分配还是隐式的标签引脚的方式分配还是分离适配，带边界标签的隐式空闲链表分配器允许在常数时间内进行对前面块的合并。这种思想是在每个块的结尾处添加一个脚部，其中脚部就是头部的一个副本。如果每个块包括这样一个脚部，那么分配器就可以通过检查它的脚部，判断前面一个块的起始位置和状态，这个脚部总是在距当前块开始位置一个字的距离。显式空间链表就是将空闲块组织为某种形式的显式数据结构。因为根据定义，程序不需要一个空闲块的主体，所以实现这个数据结构的指针可以存放在这些空闲块的主体里面。例如，堆可以组织成一个双向空闲链表，在每个空闲块中，都包含一个前驱和后继指针，使首次适配的分配时间从块总数的线性时间减少到了空闲块数量的线性时间。为了分配一个块，必须确定请求的大小类，并且对适当的空闲链表做首次适配，查找一个合适的块。如果找到了一个，那么就（可选地）分割它，并将剩余的部分插入到适当的空闲链表中。如果找不到合适的块，那么就搜索下一个更大的大小类的空闲链表。如此重复，直到找到一个合适的块。如果空闲链表中没有合适的块，那么就向操作系统请求额外的堆内存，从这个新的堆内存中分配出一个块，将剩余部分放置在适当的大小类中。要释放一个块，我们执行合并，并将结果放置到相应的空闲链表中。

## 7.10本章小结

本章主要介绍了hello的存储地址空间de内容，了解了从逻辑地址到线性地址的变换以及从线性地址到物理地址的变换的知识。以及四级页表下的VA到PA的变换，了解了三级cache下的物理内存访问。能够解决缺页故障以及缺页中断的处理。了解了c语言中的malloc的具体执行规则，对malloc的使用有了更深入的了解。

**（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

## 设备管理：unix io接口

## 文件就是一个字节序列，所有的IO设备都被模型化为文件，而所有的输入和输出都被当做对相应文件的读和写来执行，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，这使得所有的输入和输出都能以一种统一且一致的方式来执行。

## 8.2 简述Unix IO接口及其函数

Unix I/O接口：

1.打开文件。一个应用程序通过要求内核打开相应的文件，来宣告它想要访间一个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 操作，显式地设置文件的当前位置为K 。

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

Unix I/O函数：

1.进程是通过调用open 函数来打开一个已存在的文件或者创建一个新文件的：

int open(char \*filename, int flags, mode\_t mode);

open 函数将filename 转换为一个文件描述符，并且返回描述符数字。返回的描述符总是在进程中当前没有打开的最小描述符。flags 参数指明了进程打算如何访问这个文件，mode 参数指定了新文件的访问权限位。

返回：若成功则为新文件描述符，若出错为-1。

2.进程通过调用close 函数关闭一个打开的文件。

int close(int fd);

返回：若成功则为0, 若出错则为-1。

3.应用程序是通过分别调用read 和write 函数来执行输入和输出的。

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

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

返回：若成功则为读的字节数，若EOF 则为0, 若出错为-1。

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

write 函数从内存位置buf 复制至多n 个字节到描述符fd 的当前文件位置。图10-3 展示了一个程序使用read 和write 调用一次一个字节地从标准输入复制到标准输出。

返回：若成功则为写的字节数，若出错则为-1。

## 8.3 printf的实现分析

前提：printf 和 vsprintf 代码是 windows 下的。

查看 printf 代码：

int printf(const char fmt, …) {

int i;

char buf[256];

va\_list arg = (va\_list)((char)(&fmt) + 4);

i = vsprintf(buf, fmt, arg);

write(buf, i);

return i;

}

首先 arg 获得第二个不定长参数，即输出的时候格式化串对应的值。 查看 vsprintf 代码：

int vsprintf(char \*buf, const char fmt, va\_list args) {

char p;

char tmp[256];

\_list p\_next\_arg = args;

for (p = buf; \*fmt; fmt++) {

if (\*fmt != ‘%’) //忽略无关字符

{

\*p++ = \*fmt;

continue;

}

fmt++;

switch (\*fmt)

{

case ‘x’: //只处理%x一种情况

itoa(tmp, ((int)p\_next\_arg)); //将输入参数值转化为字符串保存在tmp strcpy(p, tmp); //将tmp字符串复制到p处

p\_next\_arg += 4; //下一个参数值地址

p += strlen(tmp); //放下一个参数值的地址

break;

case ‘s’:

break;

default:

break;

}

}

return (p - buf); //返回最后生成的字符串的长度

}

则知道 vsprintf 程序按照格式 fmt 结合参数 args 生成格式化之后的字符串，并 返回字串的长度。 在 printf 中调用系统函数 write(buf,i)将长度为 i 的 buf 输出。write 函数如下：

write:

mov eax, \_NR\_write

mov ebx, [esp + 4]

mov ecx, [esp + 8]

int INT\_VECTOR\_SYS\_CALL

在 write 函数中，将栈中参数放入寄存器，ecx 是字符个数，ebx 存放第一个 字符地址，int INT\_VECTOR\_SYS\_CALLA 代表通过系统调用 syscall，查看 syscall 的实现：

sys\_call:

call save

push dword [p\_proc\_ready]

sti

push ecx

push ebx

call [sys\_call\_table + eax \* 4]

add esp, 4 \* 3

mov [esi + EAXREG - P\_STACKBASE], eax

cli

ret

syscall 将字符串中的字节“Hello ”从寄存器中通过总线复

制到显卡的显存中，显存中存储的是字符的 ASCII 码。字符显示驱动子程序将通过 ASCII 码在字模库中找到点阵信息将点阵信息存 储到 vram 中。显示芯片会按照一定的刷新频率逐行读取 vram，并通过信号线向液晶显示器 传输每一个点（RGB 分量）。于是我们的打印字符串“Hello i”就显示在了屏幕上。

## 8.4 getchar的实现分析

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

getchar 函数落实到底层调用了系统函数 read，通过系统调用 read 读取存储在 键盘缓冲区中的 ASCII 码直到读到回车符然后返回整个字串，getchar 进行封装， 大体逻辑是读取字符串的第一个字符然后返回

## 8.5本章小结

Linux将I/O输入都抽象为了文件，并提供相应的Unix I/O接口，用户可以通过这些接口实现输入与输出。

用户系统通常通过调用编程语言提供的库函数或者操作系统提供的API函数来实现I/O操作，这些函数最终都会调用系统调用的封装函数，通过封装函数中的陷阱指令使用户进程从用户态转到内核态执行。

**（第8章1分）**

# 结论

hello经历的过程：

1.预处理，C语言编译器对各种预处理命令进行处理，包括对头文件的包含，宏定义的扩展，条件编译的选择等，将hello.c转换为hello.i。

2.编译，将C语言文件hello.i翻译为汇编语言文件hello.s

3.汇编，将汇编语言代码文件hello.s转换为可重定位目标文件hello.o

4.链接，将多个可重定位目标文件hello.o、libc.a等经过符号解析和重定位结合，形成一个具有统一地址空间的可执行目标文件hello

5.运行，在shell下输入命令./hello 1170300916 pyx，shell解析命令并构造参数列表

6.Fork子进程，shell调用fork创建一个子进程，具有和父进程完全相同的虚拟存储空间备份

7.加载，shell将构造好的参数列表传给execve作为参数，启动加载器并开始执行hello的第一条指令，实现在当前进程上下文中运行hello，

8.执行，cpu通过上下文切换分配时间片

9.访存，hello程序运行中需要的代码和数据，通过虚拟地址在TLB和主存页表中查找转换为相应物理地址，再在cache和主存中读取

10.异常处理，如果通过键盘输入导致外部中断，相应异常处理程序发送信号给进程，进程调用相应信号处理函数

11.回收，shell父进程回收进程，内核删除为这个进程创建的所有数据结构。

感悟：计算机系统的实现是一个复杂的过程，程序员在用高级语言编写程序的时候应该考虑代码实现的过程和硬件环境，充分利用现有资源实现更好的功能。

**（结论0分，缺失 -1分，根据内容酌情加分）**

# 附件

## hello hello链接后的可执行文件

## hello.c 源代码

## hello.i 预处理生成文件

## hello.o 汇编之后的可重定位目标执行

## hello.s 编译之后的汇编文件

## hellobj Hello.o的反汇编代码

## helloelf Hello.o的ELF格式

## helloelf2 Hello的ELF格式

**（附件0分，缺失 -1分）**

# 参考文献

[1] 兰德尔E.布莱恩特 大卫R.奥哈拉伦. 深入理解计算机系统（第3版）.

机械工业出版社. 2018.4.

[2] 袁春风 计算机系统基础 机械工业出版社，2018.

[3] ZK的博客. read和write系统调用以及getchar的实现.

https://blog.csdn.net/ww1473345713/article/details/51680017. 2016-6-15.

[4] 虚拟地址、逻辑地址、线性地址、物理地址：

https://blog.csdn.net/rabbit\_in\_android/article/details/49976101

[5] Pianistx. [转]printf 函数实现的深入剖析.

https://www.cnblogs.com/pianist/p/3315801.html. 2013-09-11.

[6] gcc 简单的 hello-world 到底连接了什么

https://blog.csdn.net/hejinjing\_tom\_com/article/details/32325749

[7] 动态链接

https://blog.csdn.net/shuange3316/article/details/79221941

[8] 动态链接原理分析

https://blog.csdn.net/shenhuxi\_yu/article/details/71437167

[9] printf 函数实现的深入剖析

https://www.cnblogs.com/pianist/p/3315801.html

**（参考文献0分，缺失 -1分）**