

**计算机系统**

**大作业**

题 目 程序人生-Hello’s P2P

专 业 软件工程

学　　 号 1183710208

班　　 级 1837102

学 生 宋雨辰

指 导 教 师 史先俊

**计算机科学与技术学院**

**2019年12月**

**摘 要**

本论文将研究在linux系统下的hello的一生。结合CSAPP课本，运用各种工具，实现源程序从预处理，编译，汇编等到进程加载，动态内存分配再到结束的整个流程，从而将课本知识进行重新的梳理和贯通。

**关键词：**大作业；CSAPP；Hello的一生；程序实现流程

**（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

在文件夹内新建一个为翁当并命名为hello.c。

通过Editor将代码输入到hello.c中。

P2P过程：在linux中，hello.c经过如图所示的处理，最终成为可执行目标程序hello，在shell中通过fork产生子进程，这就是From Program to Process，即P2P。

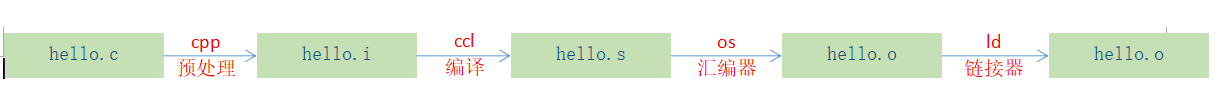


图1.1 P2P过程的大致流程

020过程：之后，shell通过execve加载hello，映射虚拟内存并载入物理内存，之后进入main函数执行目标代码，CPU为运行的hello分配时间片执行逻辑控制流，运行结束后，通过shell父进程回收hello进程，内核将相关数据结构删除。

## 1.2 环境与工具

硬件环境：Intel Core i5-8300H CPU @ 2.30GHz,8G RAM,256G SSD +2T HDD.

软件环境：Ubuntu18.04.3 LTS

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

## 1.3 中间结果

|  |  |
| --- | --- |
| hello.i | 预处理得到的文本 |
| hello.s | 编译得到的汇编文件 |
| hello.o | 汇编得到的可重定位目标文件 |
| hello | 链接生成的可执行目标文件 |
| hello.objdump | hello的反汇编代码 |
| helloo.objdump | hello.o的反汇编代码 |
| hello.elf | hello的ELF文件 |
| helloo.elf | hello.o的ELF文件 |

## 1.4 本章小结

本章主要介绍了 hello 的 p2p，020 过程，列出了本次实验的环境和中间结果。

**（第1章0.5分）**

# 第2章 预处理

## 2.1 预处理的概念与作用

概念：预处理器cpp根据以字符#开头的命令，对原始的C程序进行修改，并将引用的库展开与原始C文件合并成一个完整的文本文件。

作用：1.文件包含（引用库）：使用#include <文件名>（一般为.h头文件），可将头文件的内容直接插入到程序文本中。编译时以插入后的程序文本为编译单位，最后只得到一个目标文件.obj；

2.宏定义：使用#define 标识符 实际符，可将标识符（即宏名）替换为实际符；

3.条件编译：使用#if 表达式 格式，可通过表达式里的条件真假来决定是否对代码进行编译，只有在满足条件时，该段代码才会出现在被编译后的文件中，从而有效地减短了运行时间。

## 2.2在Ubuntu下预处理的命令

命令：cpp hello.c > hello.i

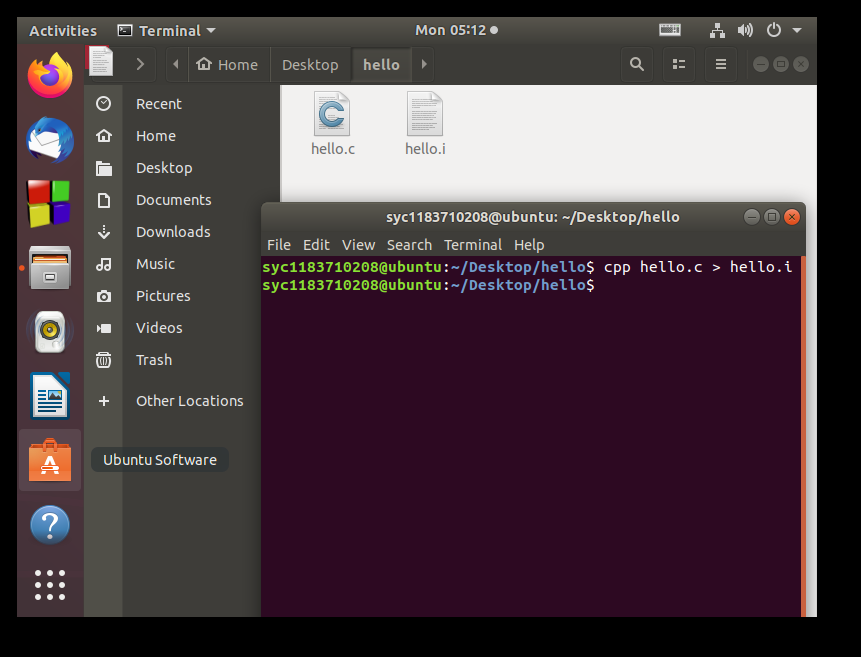


图2.1 生成hello.i文件

## 2.3 Hello的预处理结果解析

经过预处理之后我们得到文件hello.i，通过vim打开之后我们发现，整个文件长度为3113行，main函数被置于最底端。

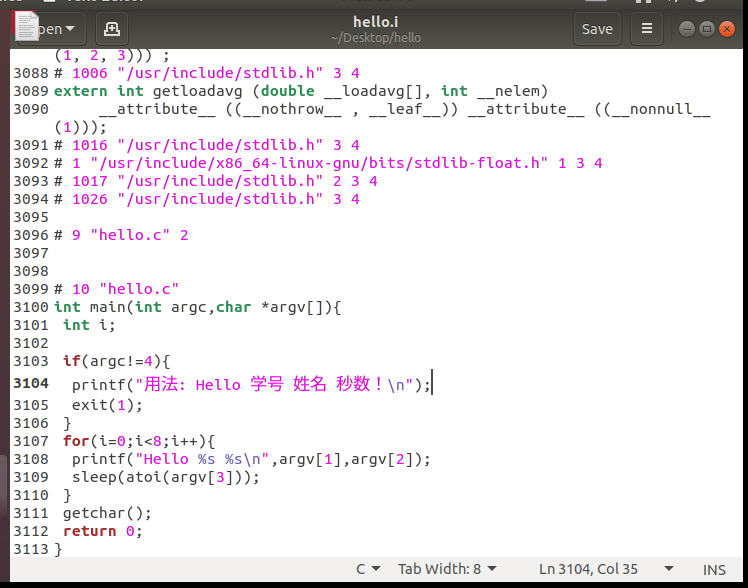


图2.2 hello.i中main函数的位置

在这之前的三千多行均为各种头文件的依次展开。

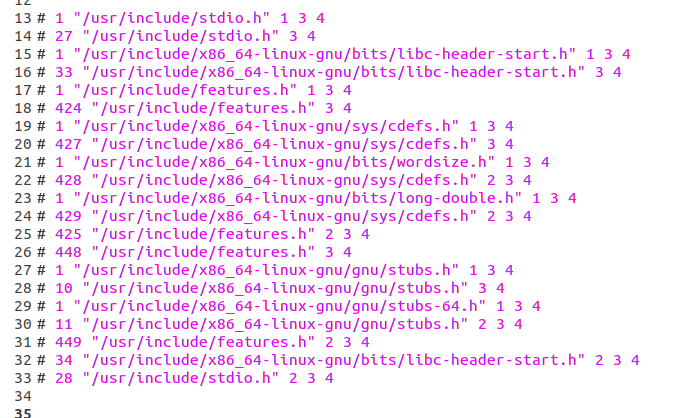


图2.3 hello.i中各种头文件的展开。

## 2.4 本章小结

本章朱亚奥介绍了预处理的定义与作用，并对预处理得到的文件进行了简要的观察和解析。

**（第2章0.5分）**

# 第3章 编译

## 3.1 编译的概念与作用

概念：编译器将文本文件hello.i翻译成文本文件hello.s，其中包含一个汇编语言程序。

主要流程为：1.通过词法分析器小苗，将字符串转化成为内部的表示结构； 2.通过语法分析和语义分析，将上一步得到的标记流生成为语法树； 3.通过上步得到的语法树，将代码转化为目标代码。

## 3.2 在Ubuntu下编译的命令

命令：gcc -S hello.i -o hello.s

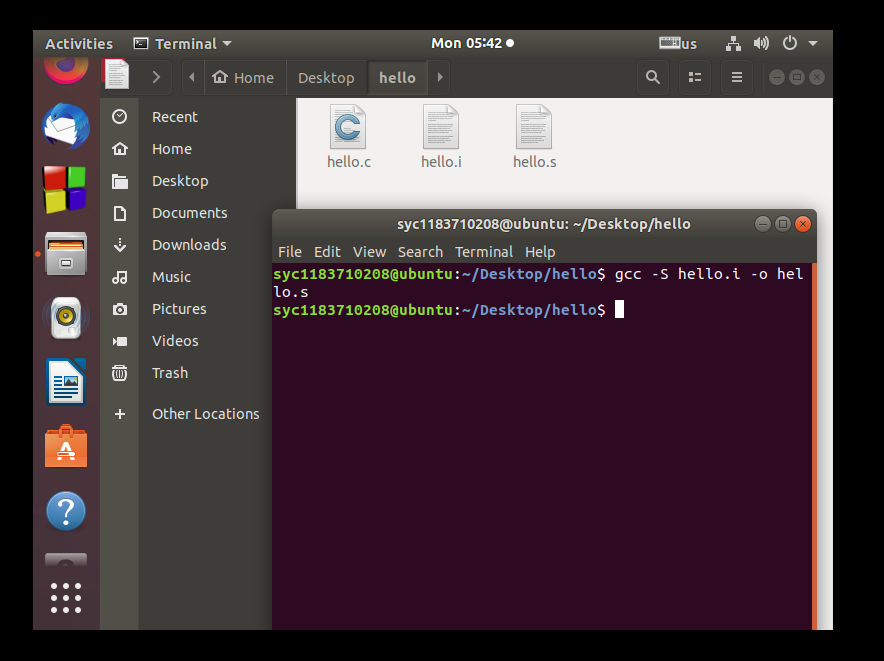


图3.1 生成hello.s

## 3.3 Hello的编译结果解析

3.3.1 数据

1.字符串

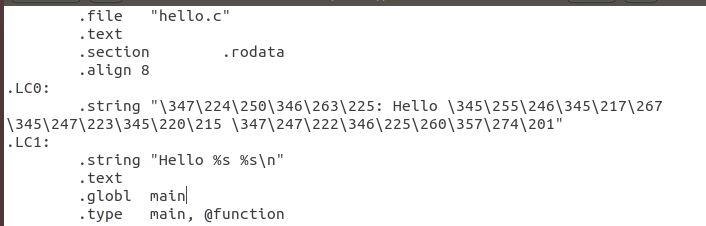


图3.2 hello.s中声明的字符串

声明了两个字符串：

第一个是"用法: Hello 学号 姓名 秒数！\n"，其中学号，姓名两个字符串被声明为UTF-8格式，其中，一个汉字栈三个字节，而一个、代表一个字节。

第二个是"Hello %s %s\n"，两个%s为格式化参数。

2.整数

1）

图3.3 hello.s中i的定义

int i：编译器将局部变量i压入栈中，可以通过.L2得到，i被亚茹到-4(%rbp)中；

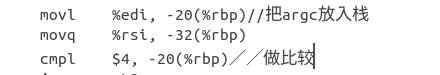
2）

图3.4 hello.s中argc的定义

int argc：作为函数的第一个参数，实际上也是后面的数组argv的一个标示量，被放置在%rdi寄存器中。

3）其他整数：几乎均以立即数的形式出现。

3.3.2数组

通过与hello.i对照可以发现，穿入main函数的除了argv4c这一整型之外，还存在一个数组：char \*argv[]，它作为char指针的数组，为函数的第二个参数。其中单个元素的大小为8字节，起始地址为argv。

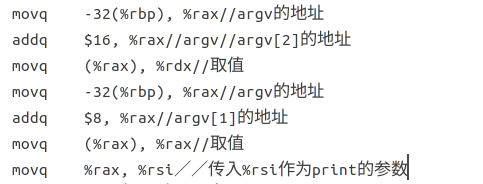


图3.5 hello.s中argv的定义和使用

3.3.3 算术操作

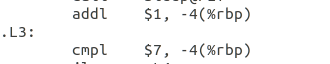
1. 

图3.6 对应i++；

1. 

图3.7 取地址运算，取出"Hello %s %s\n"作为printf的参数。

3.3.4 关系操作

①

图3.8 条件判断语句argc!=4

-20(%rbp)中为argc，与4进行比较，相等则跳入L2，不等则继续。

这段实际上对应了下图的C代码：

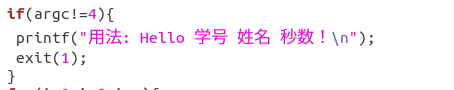


图3.9 汇编语言对应C代码（1）

其中，je .L2跳转指令里.L2内的内容为if判断不成立之后，下条语句开始执行的内容。

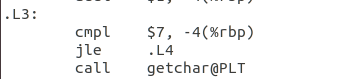
②

图3.10 （循环内）条件判断i<7

-4(%rbp)中为i，与7进行比较，小于等于则跳入.L4，不满足则继续。

这段实际上对应了下图的C代码：

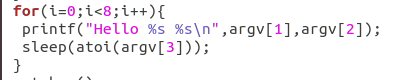


图 3.11 汇编语言对应C代码（2）

其中jle .L4跳转指令里.L4内的内容即为for循环内执行的内容。

3.3.5 控制转移

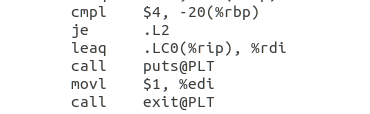
①

图 3.12 条件判断语句全景

其中比较部分同3.3.3所述，下接部分即为条件判断成立后括号里执行的内容。

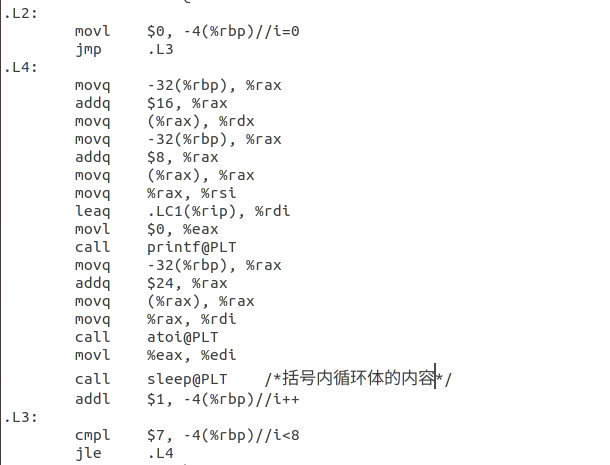
②

图 3.13 for循环语句全景

各部分内容如图内标注所示。

3.3.6 函数操作

函数一般包含以下几个步骤：

**参数传递**

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。

64位系统内含有六个存放参数的寄存器，按照顺序分别为：

%rdi,%rsi,%rdx,%rcx,%r8,%r9

包含此步操作的有main函数（在外部调用时使用了%rdi和%rsi用于分别存放argc的值和argv的头地址），printf函数等。

**控制传递**

利用call指令，将下一条指令的地址dest压栈，然后跳转。

例如：



 图 3.14 几个call指令的举例



**分配和释放内存：**

在被调用函数的开始阶段，可能需要为将局部变量分配空间，而在返回之前，又需要将这些空间释放出来。

例如：main函数在程序结束时，调用了leave指令，将栈空间恢复为调用之前的状态，之后通过ret进行跳出。

## 3.4 本章小结

本章主要介绍了编译器是如何处理C语言的各个数据类型以及各类操作。通过各种汇编的基本原理，并结合hello.c的代码以及hello.s的代码，找到其映射关系并给出合理解释。

编译器将.i编译为.s的汇编代码。经过编译之后，hello自C语言变为较为低级的汇编语言。

**（第3章2分）**

# 第4章 汇编

## 4.1 汇编的概念与作用

汇编器（as）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

指令：as hello.s -o hello.o

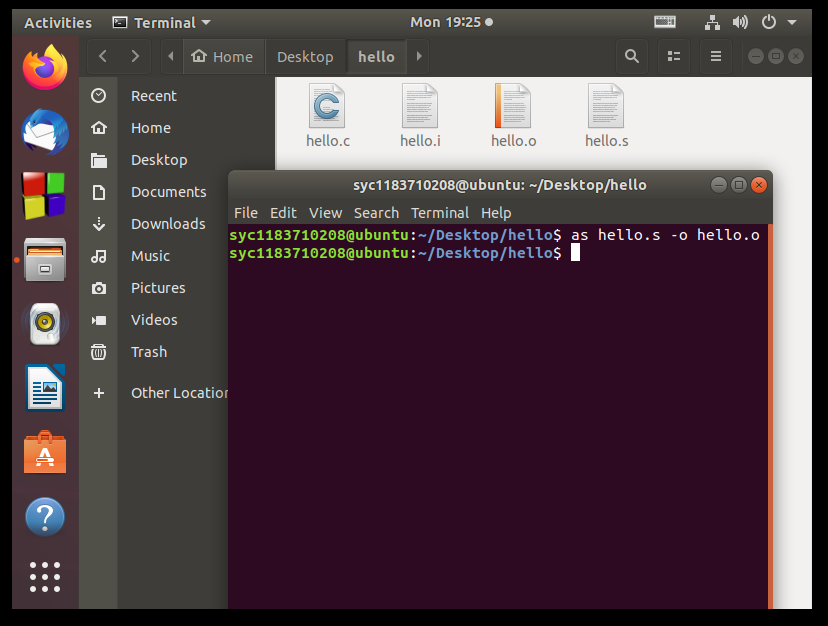


图 4.1 生成hello.o

## 4.3 可重定位目标elf格式

ELF头：以一个十六字节的序列开始，名字为Magic，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。

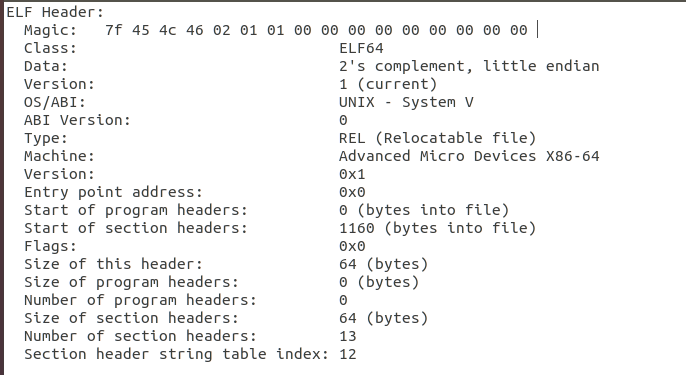


图4.2 ELF头

**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。

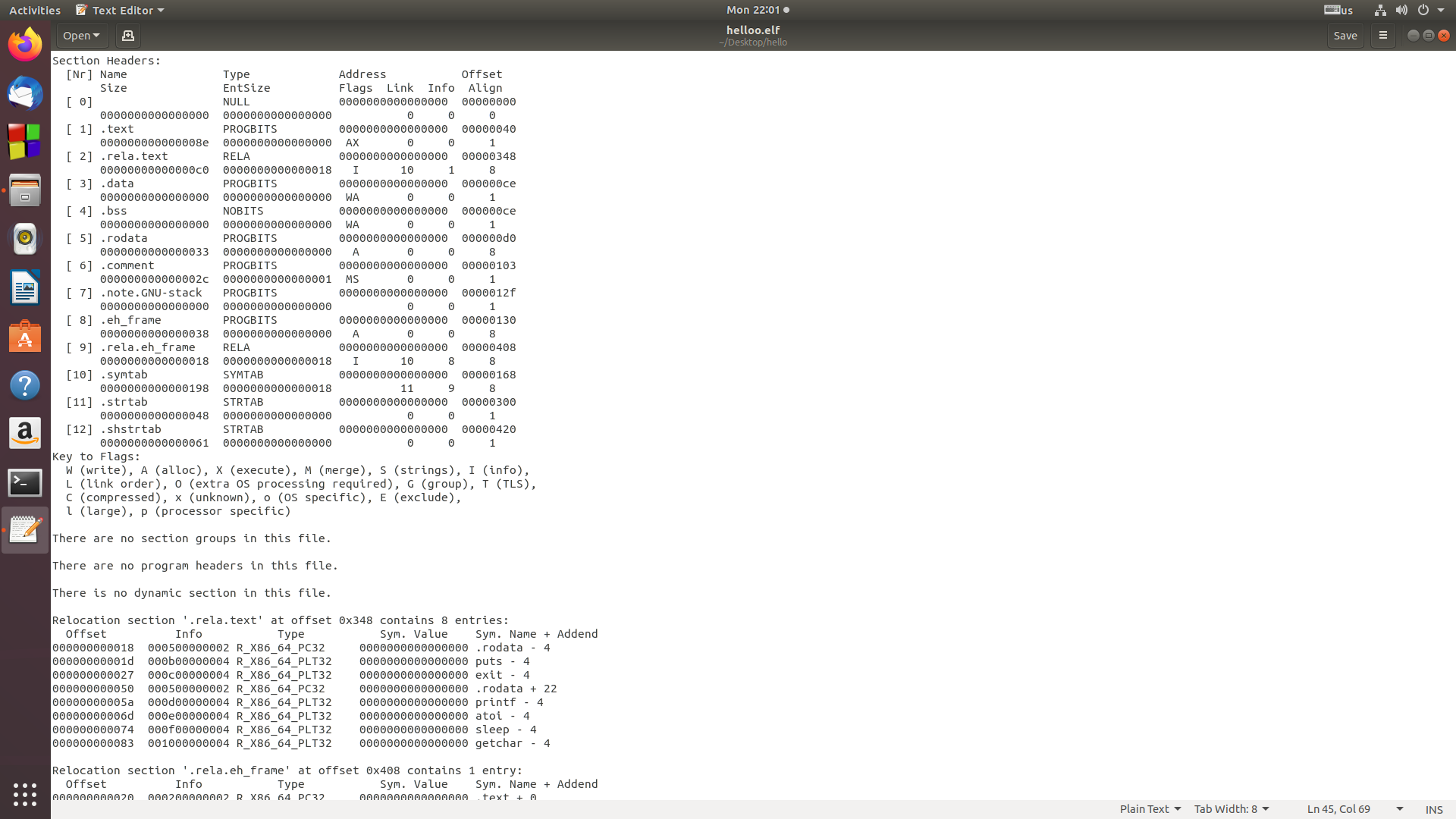


图 4.3 节头部表

**重定位节**：代码的重定位条目，包含.text 节中需要进行重定位的信息，并告诉链接器在将目标文件合并成可执行文件时如何修改这个应用。

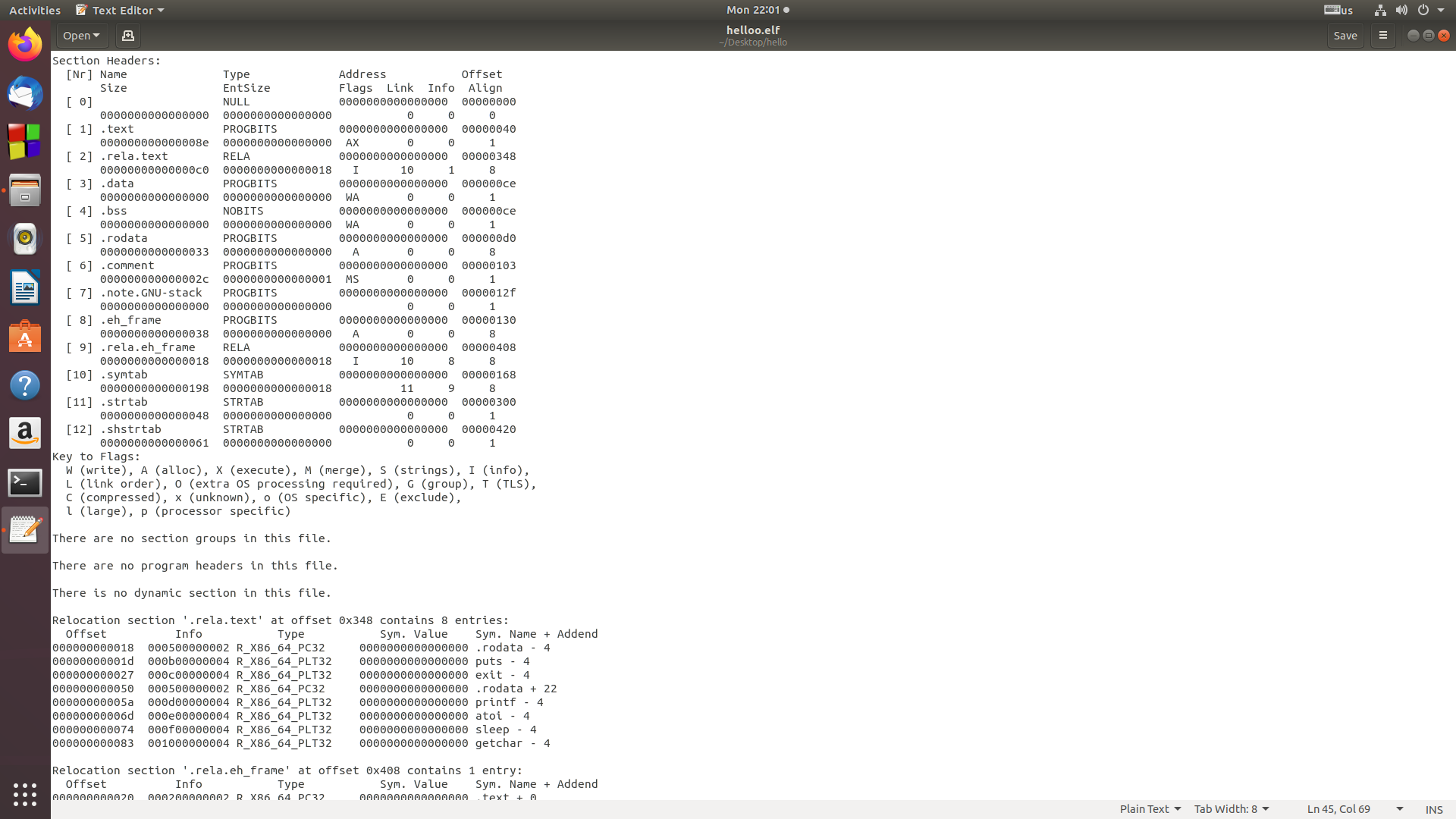


图 4.4 重定位节 .rela.text

对于一条重定位声明，包含以下基本内容：

1)偏移量（offset）：需要被修改的引用的节偏移； 2)信息（info）：提供了符号表中的一个位置，同时还包括重定位类型的有个信息。这是通过将值划分为两部分来达到的。该值的低8位表示重定位入口的类型（type），高24位表示重定位入口的符号在符号表重的下标； 3)类型（type）：告知连接器如何修改新的引用； 4)符号名称（name）：重定位目标的名称； 5)加数（addend）：一个有符号常数，计算重定位位置的辅助信息，共占8个字节。

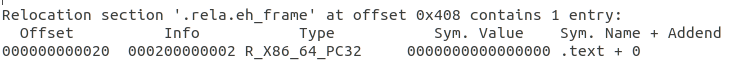


图 4.5 重定位节 .rela.eh\_frame

**符号表：**包含重定位模块定义和引用的符号的信息（包括函数和全局变量）。重定位需要引用的符号都需要在此处进行声明。

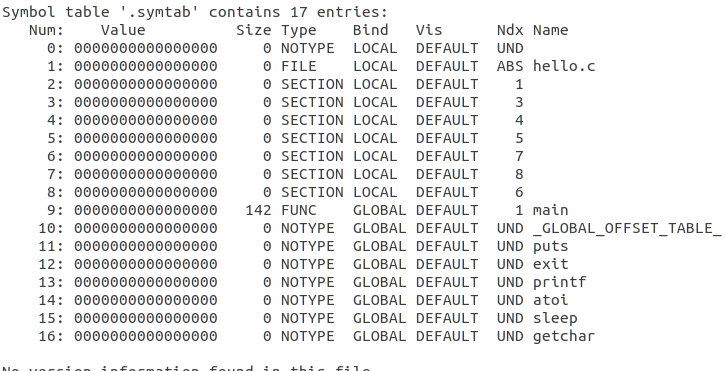


图 4.6 符号表

## 4.4 Hello.o的结果解析

使用 objdump -d -r hello.o > hello.objdump 获得反汇编代码。

通过观察可以发现，两者除了显示格式之外，差别基本不大，主要体现在以下几个地方：

1）分支跳转：由于段名称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时不会出现，转而用确定的地址对其进行替代。



图 4.7 分支跳转对比（上为.s文件，下为.objdump文件反汇编）

2）函数调用：由于hello.c中调用的都是共享库中的函数，需要通过动态链接器确定地址。对于这些不确定地址的函数调用，在汇编成为机器语言的时候，将其call指令后的相对地址设置为0，然后再重定位节中添加重定位条目，等待确认。



图 4.8 函数调用对比（上为.s文件，下为.objdump文件反汇编）

## 4.5 本章小结

本章介绍了 hello 从 hello.s 到 hello.o 的汇编过程，通过查看elf格式的hello.o和使用objdump得到的反汇编代码与hellos进行对照，了解到汇编器在汇编语言映射到机器语言时所做的转换。

**（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。

## 5.2 在Ubuntu下链接的命令

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

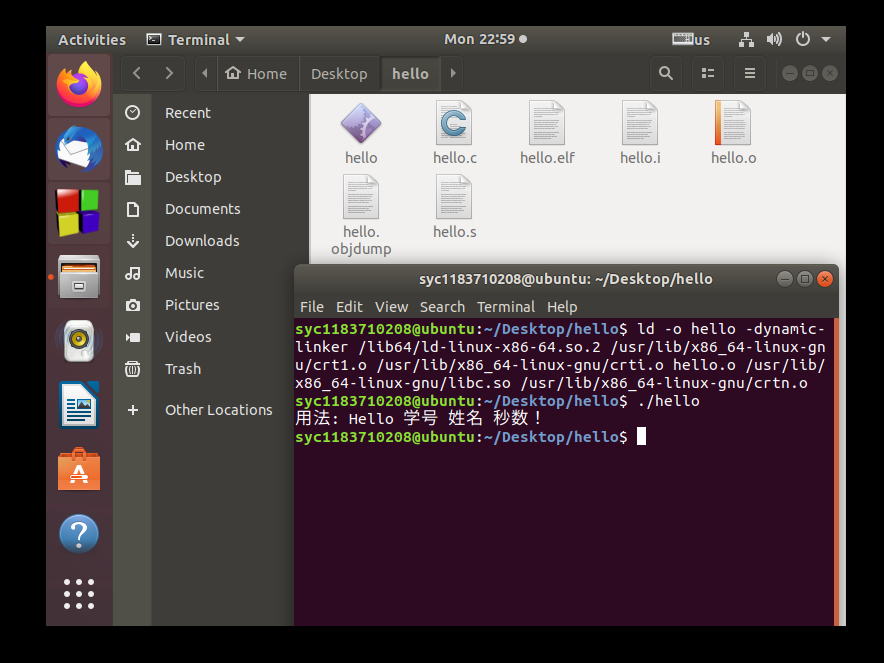


图5.1 使用ld链接命令生成可执行程序hello

## 5.3 可执行目标文件hello的格式

节头部表基本信息：大小（size），类型（type），地址（address），offset（偏移量）。其中address为被载入到的虚拟地址的起始地址，size为大小

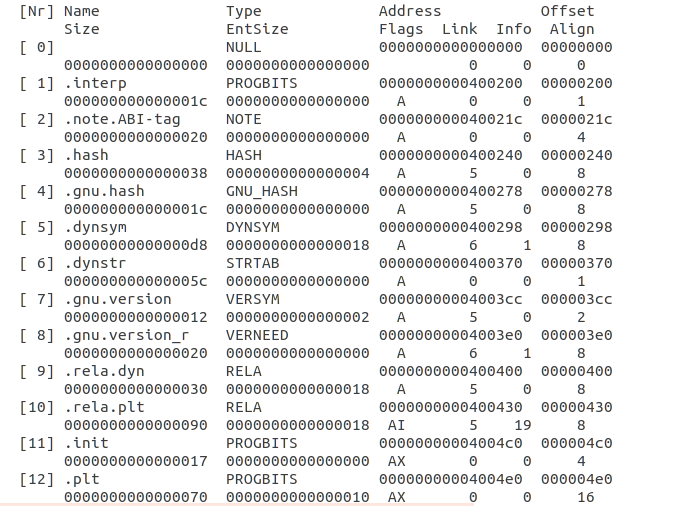
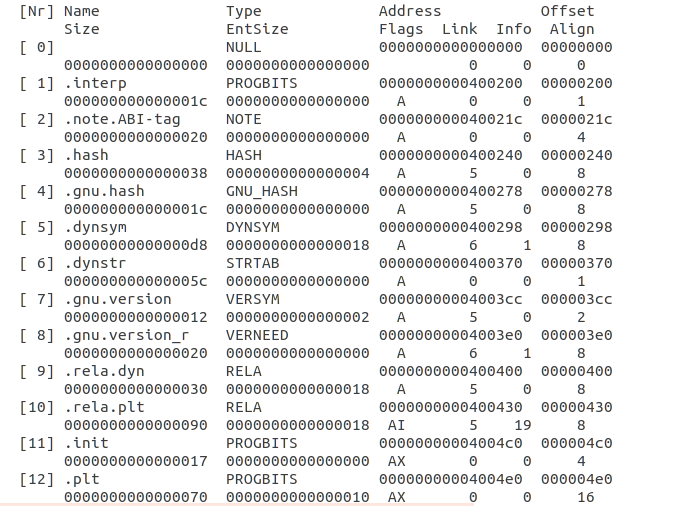


图 5.2 hello中的ELF格式

## 5.4 hello的虚拟地址空间

两个例子：



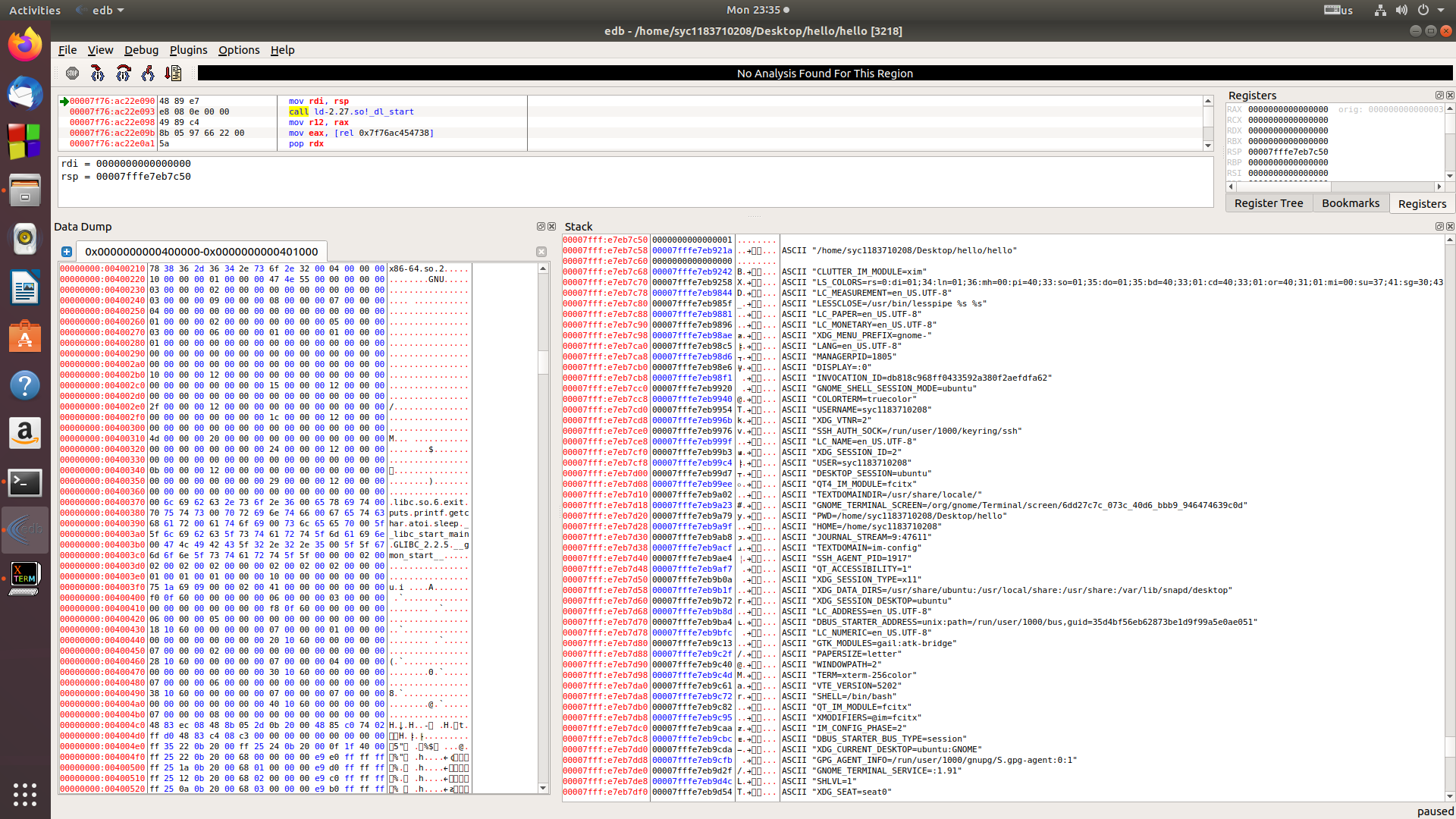


图 5.3 .init的起始地址和虚拟地址载入

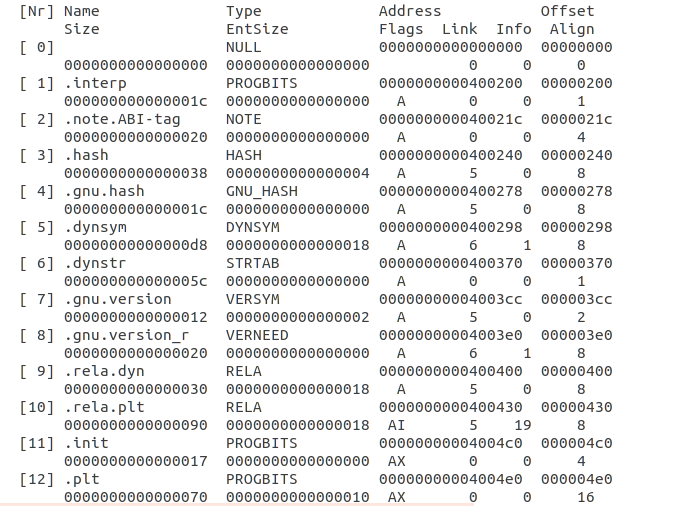
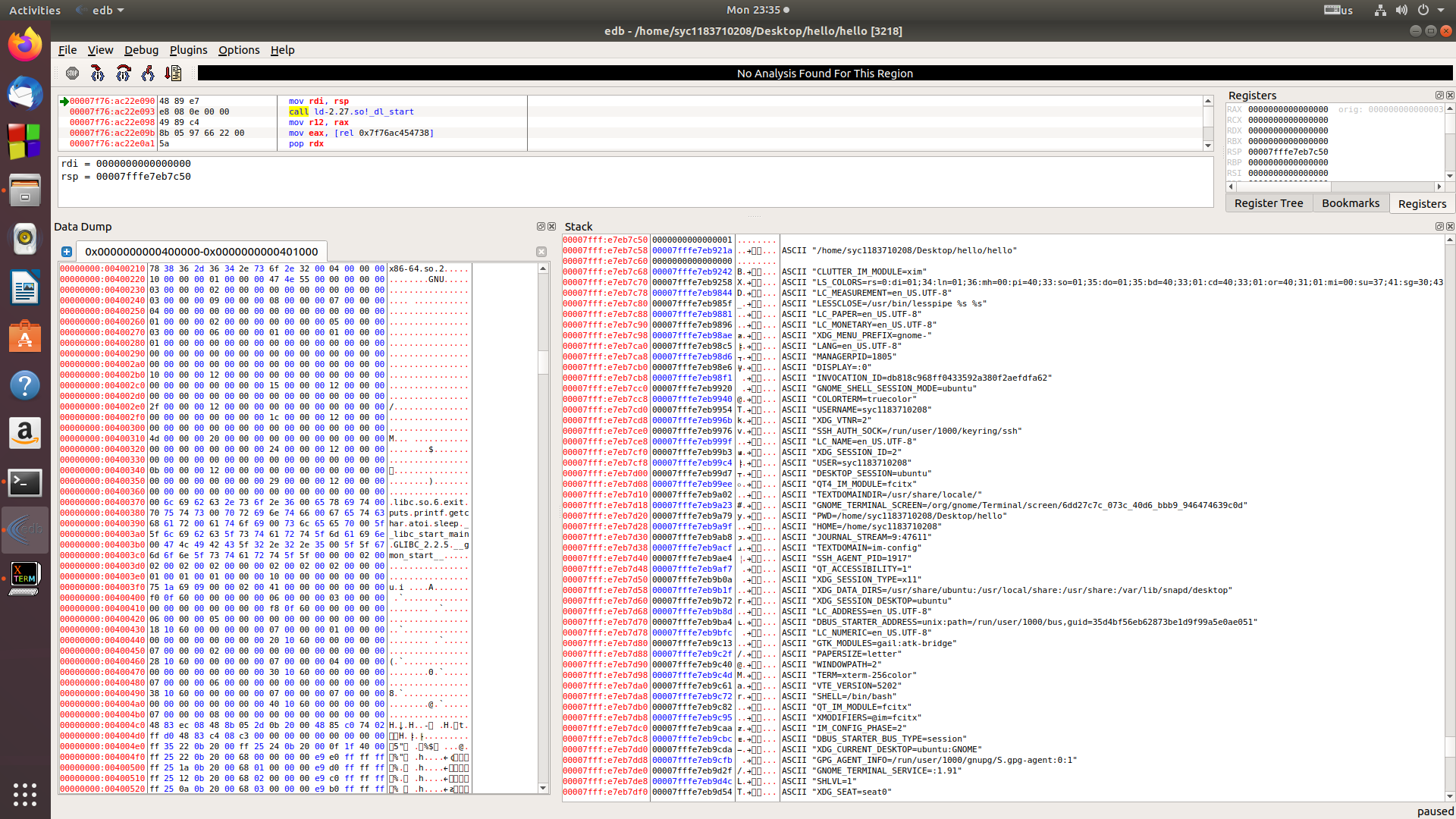
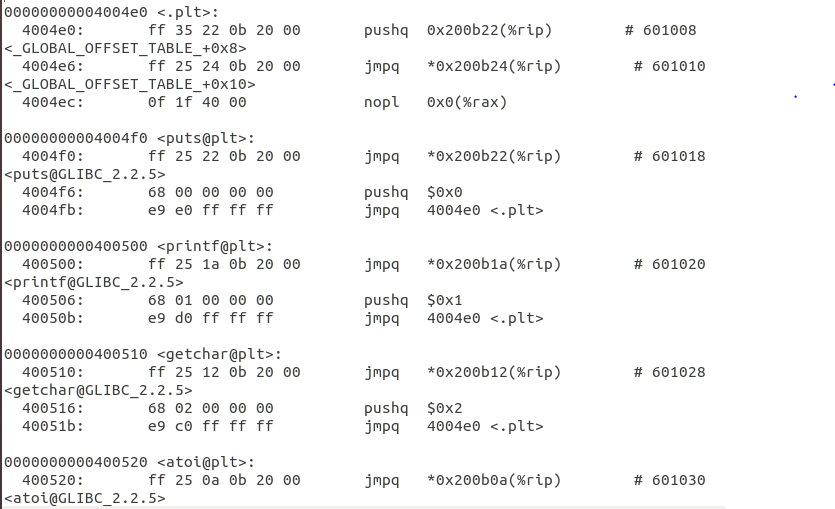
 

图5.4 .dynstr的起始地址和虚拟地址载入

通过 Data Dump查看虚拟地址段，我们发现，起始地址的确对应着被载入后的虚拟地址。

## 5.5 链接的重定位过程分析

与hello.o的反汇编文本helloo.objdump相比，hello的反汇编文本hello.objdump有以下差别：

1. 多出了很多函数：main函数通过链接器将这些函数从共享库中能够提取出来，并且把它们加入可执行目标文件使其完整；
2. 调用函数的方式：因为链接器已经计算出了位置，所以call后面跟着所调用的函数的实际地址，而不是下一条指令的地址； 图 5.5 函数调用改为直接接实际地址
3. .rodata 引用：链接器解析重定条目时发现 重定位，由于.rodata与.text节之间的相对距离确定，因此链接器直接修改call之后的值为目标地址与下一条指令的地址之差，指向相应的字符串。算法如下：

refptr = s + r.offset； refaddr = ADDR(s) + r.offset；

\*refptr = (unsigned) (ADDR(r.symbol) + r.addend-refaddr)

## 5.6 hello的执行流程

程序名：

ld-2.27.so!\_dl\_start

ld-2.27.so!\_dl\_init

hello!\_start

libc-2.27.so!\_\_libc\_start\_main

-libc-2.27.so!\_\_cxa\_atexit

-libc-2.27.so!\_\_libc\_csu\_init

hello!\_init

libc-2.27.so!\_setjmp

-libc-2.27.so!\_sigsetjmp

--libc-2.27.so!\_\_sigjmp\_save

hello!main

hello!puts@plt

hello!exit@plt

\*hello!printf@plt

\*hello!sleep@plt

\*hello!getchar@plt

ld-2.27.so!\_dl\_runtime\_resolve\_xsave

-ld-2.27.so!\_dl\_fixup

--ld-2.27.so!\_dl\_lookup\_symbol\_x

libc-2.27.so!exit

## 5.7 Hello的动态链接分析

## 对于动态共享链接库中 PIC 函数，编译器没有办法预测函数的运行时地址，所以需要添加重定位记录，等待动态链接器处理，为避免运行时修改调用模块的代码段，链接器采用延迟绑定的策略。动态链接器使用过程链接表 PLT+全局偏移量表GOT实现函数的动态链接，GOT中存放函数目标地址，PLT使用GOT中地址跳转到目标函数。在dl\_init调用之前，对于每一条PIC函数调用，调用的目标地址都实际指向 PLT中的代码逻辑，GOT存放的是PLT中函数调用指令的下一条指令地址。

## 5.8 本章小结

本章中主要介绍了链接的概念与作用、hello的ELF格式，分析了hello的虚拟地址空间、重定位过程、执行流程、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：进程是一个执行中的程序的实例，每一个进程都有它自己的地址空间，包括文本区域、数据区域和堆栈，分别用来存储处理器执行的代码，存储变量和进程执行期间使用的动态分配的内存，存储区活动过程中调用的指令和本地变量。

作用：使我们感到正在运行的程序是系统当前运行的唯一程序，处理器无间断地执行着我们当前程序中的指令，且内存中只存放了当前程序中的代码和数据。

## 6.2 简述壳Shell-bash的作用与处理流程

作用：Shell是一个用C语言编写的程序，他是用户使用Linux的桥梁。Shell是指一种应用程序，Shell应用程序提供了一个界面，用户通过这个界面访问操作系统内核的服务。

处理流程：1）从终端读入输入的命令； 2）将输入字符串切分获得所有的参数； 3）如果是内置命令则立即执行； 4）否则调用相应的程序为其分配子进程并运行； 5）shell应该接受键盘输入信号，并对这些信号进行相应处理。

## 6.3 Hello的fork进程创建过程

输入./hello之后，调用fork函数创建一个新的运行的子进程，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。

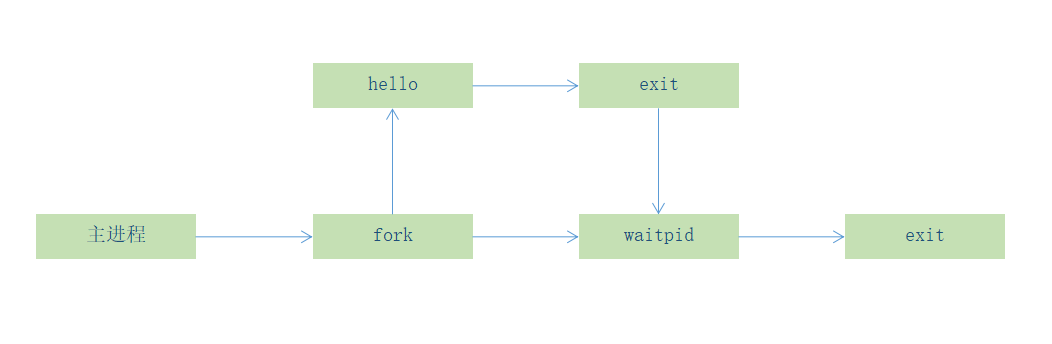


图 6.1 hello的fork进程创建

## 6.4 Hello的execve过程

调用fork函数之后，在子进程中加载execve函数，载入并运行hello函数。

覆盖当前进程的代码，数据，栈。保留相同的PID，继承已打开的文件描述符和信号上下文。

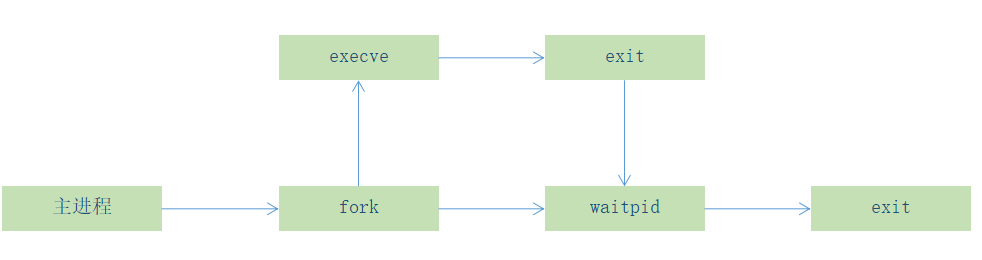


图 6.2 hello的execve过程

## 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。

## 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。

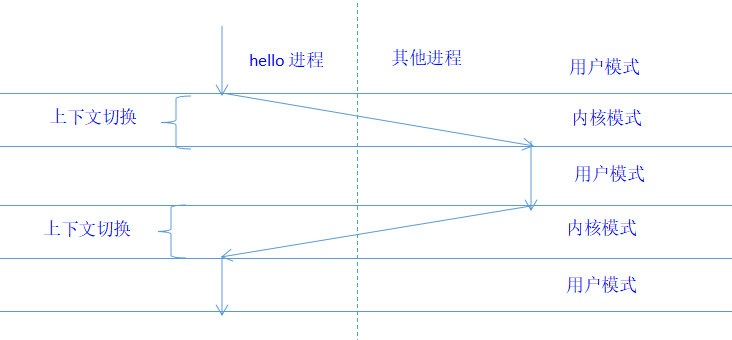


图 6.3 hello进程中sleep进程上下文切换简图

这里以sleep函数为例，进程在调用sleep后，将以此进行一下四步操作：

1）内核处理sleep发出的休眠请求，将hello挂起。（内核）

2）hello被挂起，其他进程获得当前进程控制权（用户）

3）休眠结束，收到中断信号（内核）

4）hello不再挂起重新恢复，进行自己的逻辑控制流。

## 6.6 hello的异常与信号处理

1）乱按结果：

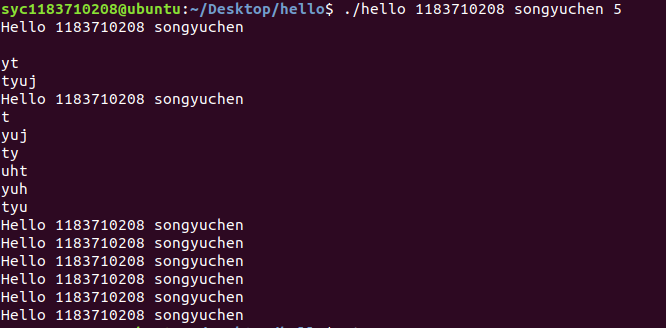


图 6.4 乱按结果

可以发现，在程序进行的过程中，乱按（包括回车）并没有对程序的运行造成影响。这是因为，在程序的最后有一个getchar()，它会将运行时的所有输入读掉（存放到stdin中），并且在当前进程结束之后把读取一起反馈到屏幕上；假如读入中存在回车，则会将之前的所有输入作为一个命令行执行。

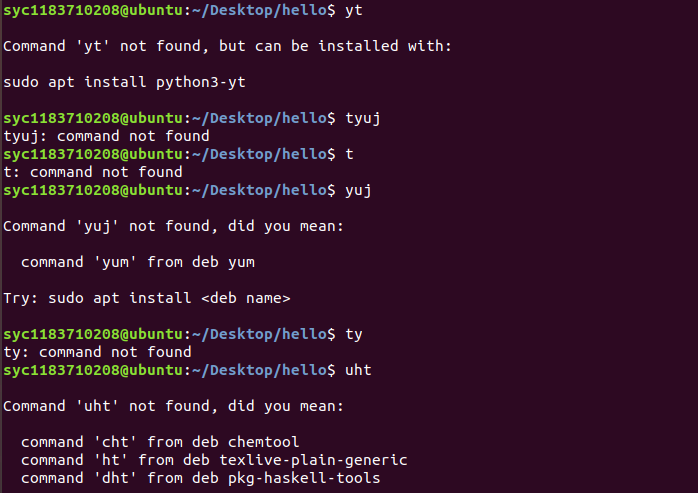


图 6.5 乱按反馈

2）输入ctrl+z

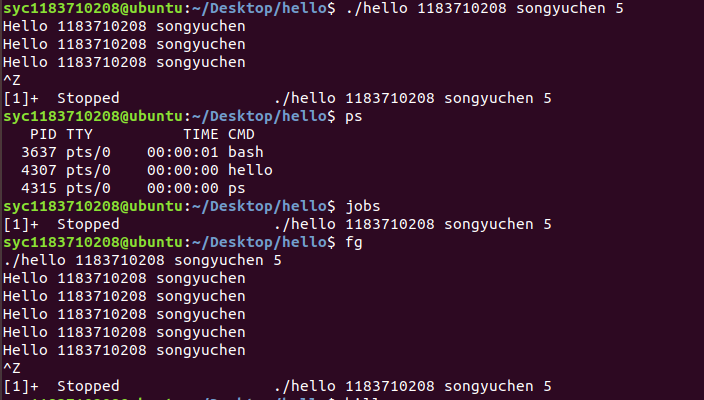


图 6.7 输入ctrl+z

可以看到，进程在收到ctrl+z时，进程会被挂起。这是因为，shell父进程收到SIGSTP信号，信号处理函数的逻辑是打印屏幕回显、将hello进程挂起。

通过ps，我们可以看出，进程并没有被终止。jobs可可以看出，目前进程只是被挂起状态，这时候可以输入fg使进程继续执行。

3）输入ctrl+c

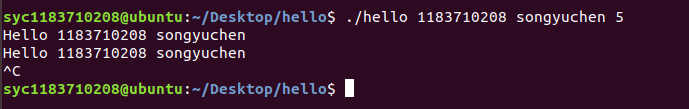


图 6.6 输入ctrl+c

可以看到，进程在收到ctrl+c时，程序直接终止。这是因为，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。

## 6.7本章小结

本章中，介绍了进程的定义与作用，Shell的一般处理流程，调用fork创建新进程，调用execve执行hello，hello的进程执行，hello的异常与信号处理等内容。

**（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：程序代码经过编译后出现在汇编程序中地址。逻辑地址由选择符（在实模式下是描述符，在保护模式下是用来选择描述符的选择符）和偏移量（偏 移部分）组成。

线性地址：逻辑地址经过段机制后转化为线性地址，为描述符:偏移量的组合 形式。分页机制中线性地址作为输入。

虚拟地址：虚拟地址在这里跟线性地址相同，即程序访问存储器所使用的逻辑地址。

物理地址：CPU 通过地址总线的寻址，找到真实的物理内存对应地址。 CPU 对内存的访问是通过连接着 CPU 和北桥芯片的前端总线来完成的。在前端总线上 传输的内存地址都是物理内存地址。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

最初8086处理器的寄存器是16位的，为了能够访问更多的地址空间但不改变寄存器和指令的位宽，所以引入段寄存器，8086共设计了20位宽的地址总线，通过将段寄存器左移4位加上偏移地址得到20位地址，这个地址就是逻辑地址。将内存分为不同的段，段有段寄存器对应，段寄存器有一个栈、一个代码、两个数据寄存器。

分段功能在实模式和保护模式下有所不同。

实模式，即不设防，也就是说逻辑地址=线性地址=实际的物理地址。段寄存器存放真实段基址，同时给出32位地址偏移量，则可以访问真实物理内存。

在保护模式下，线性地址还需要经过分页机制才能够得到物理地址，线性地址也需要逻辑地址通过段机制来得到。段寄存器无法放下32位段基址，所以它们被称作选择符，用于引用段描述符表中的表项来获得描述符。描述符表中的一个条目描述一个段，构造如图。

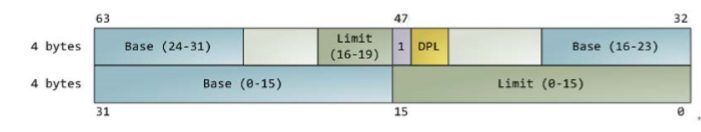


图 7.1（转） 段描述符表中的一个条目的构造

其中，Base为基地址，Limit为段界限，DPL为描述符的特权级（内核模式为0，用户模式为-3）。

在保护模式下，分段机制就可以描述为：通过解析段寄存器中的段选择符在段描述符表中根据Index选择目标描述符条目Segment Descriptor，从目标描述符中提取出目标段的基地址Base address，最后加上偏移量offset共同构成线性地址Linear Address。如图：

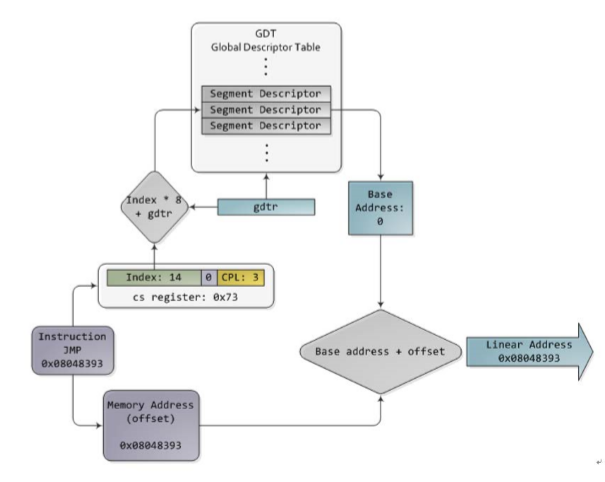


图 7.2（转） 保护模式下的分段机制

## 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转换为物理地址。从管理和效率的角度出发，线性地址被划分成固定长度单位的数组，称为页。页表中每一项存储的都是物理页的基地址。是分页单元将所有的物理内存都划分成了固定大小的单元为管理单位，其大小一般与内存页大小一致。为了能够尽可能的节约内存，CPU在页式内存管理方式中引入了两级的页表结构，如下图所示。

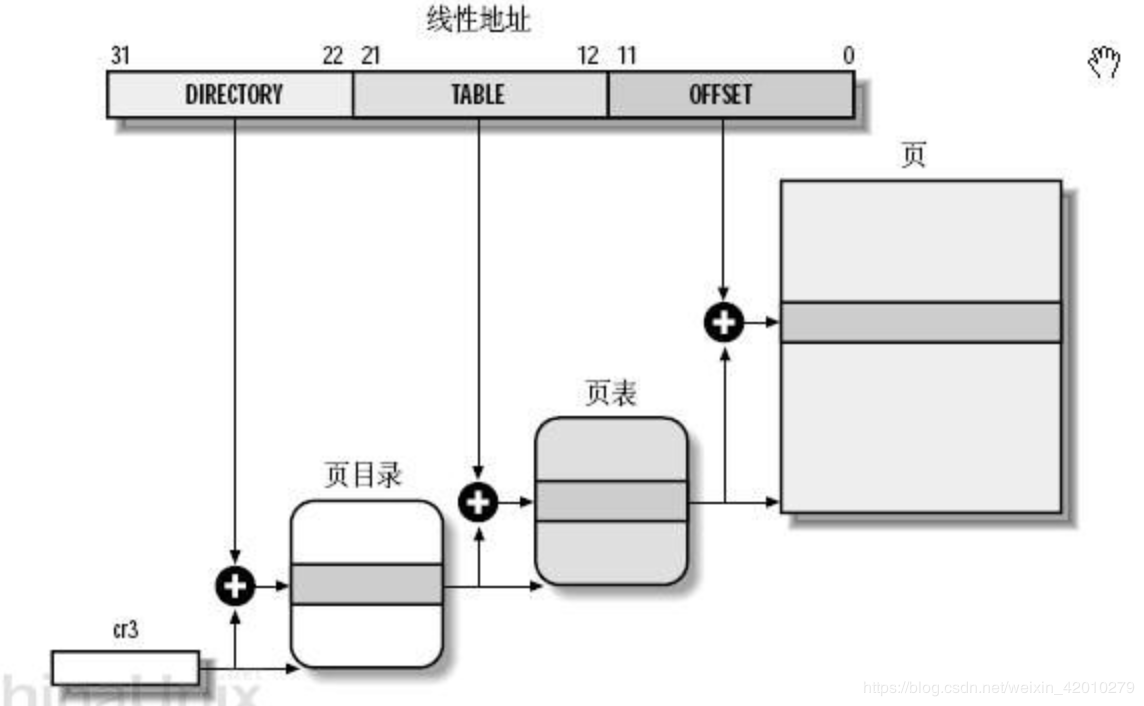


图 7.3（转） 页式管理

这种页式管理方式中，第一级的页表称之为“页目录”，用于存放页表的基地址；第二级才是真正的“页表”用于存放物理内存中页框的基地址。 1）二级页目录的页式内存管理方式中，第一级的页目录的基址存放在CPU寄存器CR3中，这也是转换的开始点； 2）每一个活动的进程，都有其对应的独立虚拟内存（页目录也是唯一的），那么它对应一个独立的页目录地址。 3、每个32位的线性地址被划分成三部分，页目录索引（10位），页表索引（10位），偏移量（12位）。

## 7.4 TLB与四级页表支持下的VA到PA的变换

每次cpu产生一个虚拟地址，MMU需要查询一个PTE,如果运气不好，需要从内存中取得，这需要花费很多时间，通过TLB（翻译后备缓冲器）能够消除这些开销。TLB是一个小的，虚拟寻址的缓存，在MMU里，其每一行都保存着一个单个PTE组成的块，TLB通常具有高度相联度用于组选择和行匹配的索引和标记字段是从虚拟地址中的虚拟页号中提取出来的。如果是32位系统，我们有一个32位地址空间，4KB的页面和一个4字节的PTE,我们总需要一个4MB的页表驻留在内存中，而对于64位系统，我们甚至需要8PB的空间来存放页表，这显然是不现实的。用来压缩页表的常见方式就是使用层次结构的页表。如果是二级页表，第一级页表的每个PTE负责一个4MB的块，每个块由1024个连续的页面组成。二级页表每一个PTE负责一个4KB的虚拟地址页面。这样的好处在于，如果一级页表中有一个PTE是空，那么二级页表就不会存在，这样会有巨大的潜在节约，因为4GB的地址空间大部分都是未分配的。现在的64位计算机采用4级页表，36位的VPN被封为4个9位的片，每个片被用作一个页面的偏移，CR3寄存器包含L1页表的物理地址。VPN1提供到一个L1PET的偏移量，这个PTE包含L2页表的基地址，VPN2提供一个到L2PTE的偏移量，以此类推。

## 7.5 三级Cache支持下的物理内存访问

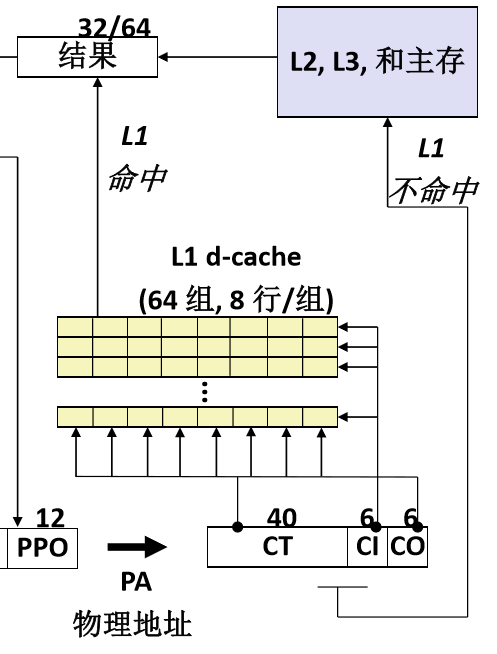


图 7.4 三级物理访存

先在第一级缓存中寻找要找的数据，根据CI在cache中找到对应组，根据CT找到匹配的标志位，若改行的有效位为1，则命中，根据CO找到数据块传输，流程完毕；若找不到对应tag或者标志位0，则发生了不命中，则在第二级缓存中寻找，找到后需要再将其缓存在第一级，若有空闲块，则放置在空闲块中，否则根据替换策略选择牺牲块；若在二级缓存中未能找到，则在第三级缓存中寻找，找到后需要缓存在第一，二级，空闲块的处理和替换策略同上；若在第三级缓存中未能找到，则在第四级缓存中寻找，找到后需要缓存在第一，二，三级，空闲块的处理和替换策略同上。

## 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。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。

## 7.7 hello进程execve时的内存映射

execve加载并运行新程序hello包含以下几个步骤：

1）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，栈和堆地址也是请求二进制零的，初始长度为零； 3）映射共享区域，hello程序与共享对象libc.so链接，libc.so是动态链接到这个程序中的，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的程序计数器，使之指向代码区域的入口点。加载器对用户地址空间区域的映射可以参照下图：

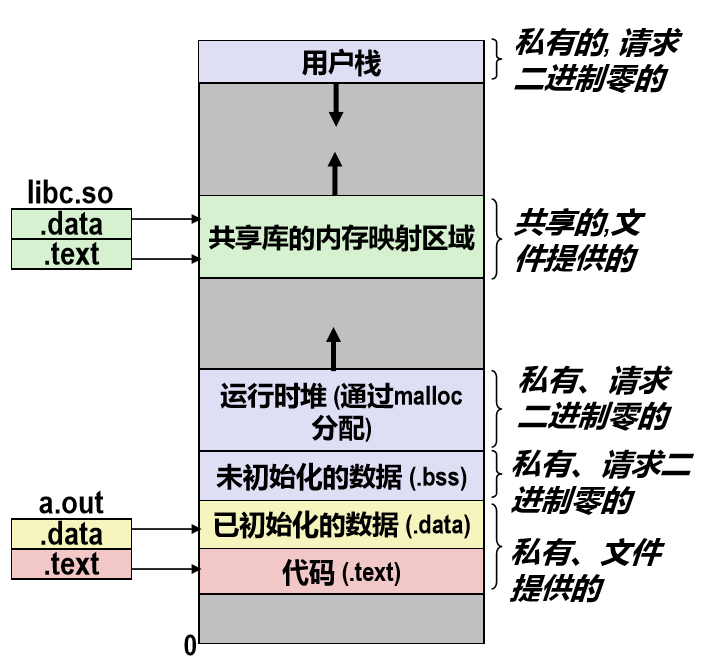


图 7.5 加载器对用户空间的映射

## 7.8 缺页故障与缺页中断处理

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缺页故障。这时候会将控制传递给处理程序，故障处理程序会选择要么重新执行当前指令，或者终止。

缺页中断会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，来替换为该虚拟地址所对应的物理页。当异常处理程序返回时，它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，从而使其可以正常进行翻译。

## 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。分配器将堆视为 一组不同大小的块的集合来维护。每个块就是一个连续的虚拟内存片，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。

分配器分为两种基本风格：显式分配器、隐式分配器。

显式分配器：要求应用显式地释放任何已分配的块。

隐式分配器：要求分配器检测一个已分配块何时不再使用，那么就释放这个块，自动释放未使用的已经分配的块的过程叫做垃圾收集。

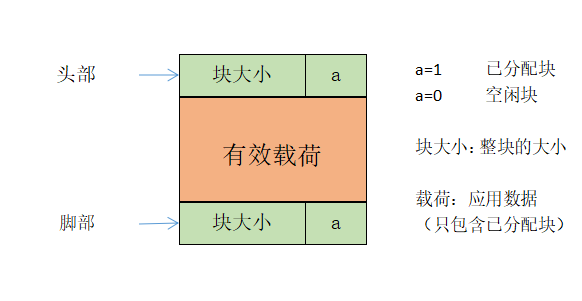


图 7.6 已分配块和空闲块格式

1.带边界标签的隐式空闲链表

一个块是有一个字的头部、有效载荷，以及可能的一些额外的填充组成的。头部编码了这个块的大小，以及这个块是已分配的还是空闲的；脚部是头部的副本，分配器可以通过检查块的脚部，判断前面一个块的起始位置和状态。

2.显式空闲链表

将空闲块组织委员某种形式的显式数据结构，堆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。

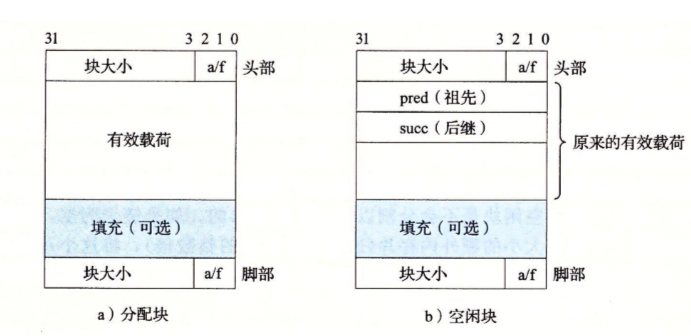


图 7.7（转） 使用双向空闲链表的堆块的格式

这种方式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。维护链表时，使用后进先出的方式进行维护，将新释放的块放置在链表的开始处，使用LIFO的顺序和首次适配的放置策略，分配器会最先检查最近使用过的块，在这种情况下，释放一个块可以在线性的时间内完成，如果使用了边界标记，那么合并也可以在常数时间内完成；或者使用按照地址顺序来维护链表，其中链表中每个块的地址都小于它后继的地址。在这种情况下，释放一个块需要线性时间的搜索来定位合适的前驱。平衡点在于，按照地址排序的首次适配比LIFO排序的首次适配有更高的内存利用率，接近最佳适配的利用率。

## 7.10本章小结

本章主要介绍了hello的存储器地址空间、intel的段式管理、hello的页式管理，VA到PA的变换、物理内存访问，ello进程fork时的内存映射、execve时的内存映射、缺页故障与缺页中断处理、动态存储分配管理。

**（第7章 2分）**

# 第8章 hello的IO管理

## 8.1 Linux的IO设备管理方法

## 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

## 8.2 简述Unix IO接口及其函数

## Unix I/O 接口统一操作：

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

## 2） Shell 创建的每个进程都有三个打开的文件：标准输入，标准输出，标 准错误。

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

## 4） 读写文件：一个读操作就是从文件复制 n>0 个字节到内存，从当前文 件位置 k 开始，然后将 k 增加到 k+n，给定一个大小为 m 字节的而文 件，当 k>=m 时，触发 EOF。类似一个写操作就是从内存中复制 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，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去。

## Unix I/O 函数：

## 1） int open(char\* filename,int flags,mode\_t mode) ，进程通过调用 open 函 数来打开一个存在的文件或是创建一个新文件的。 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。

## 2） int close(fd)，fd 是需要关闭的文件的描述符，close 返回操作结果。

## 3） ssize\_t read(int fd,void \*buf,size\_t n)，read 函数从描述符为 fd 的当前文 件位置赋值最多 n 个字节到内存位置 buf。返回值-1 表示一个错误，0 表示 EOF，否则返回值表示的是实际传送的字节数量。

## 4） ssize\_t wirte(int fd,const void \*buf,size\_t n)，write 函数从内存位置 buf 复制至多 n 个字节到描述符为 fd 的当前文件位置。

## 8.3 printf的实现分析

printf的代码：

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;   
 }

其中arg获得第二个参数，即输出的时候的格式化串。

vsprintf的代码：

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

{

char\* p;

char tmp[256];

va\_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

}

vsprintf 程序按照格式 fmt 结合参数 args 生成字符串，并返回字串的长度。

通过观察可以发现，printf函数在执行过程中调用了系统函数write(buf,i)将长度为i的buf输出到屏幕上。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

此函数将字符串中的字节从寄存器中通过总线复制到显卡的显存中，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

本章介绍了 Linux 的 IO 设备管理方法、Unix IO 接口及其函数，分析了 printf 函数和 getchar 函数。

**（第8章1分）**

# 结论

过程：

1）预处理：将hello.c文件预处理生成hello.i文件；

2）编译：将hello.i文件编译为汇编文件hello.s；

3）汇编：将hello.s汇编转换为可重定位目标文件hello.o；

4）链接：将hello.o和动态链接库以及其他可重定位目标文件链接，形成可执行目标文件hello；

5）运行：通过键入./hello 以及参数1183710208 songyuchen 5等运行文件；

6）fork和execve：fork用于创建子进程，execve用于加载映射虚拟内存，在程序开始时加载相应的物理内存；

7）执行：CPU为hello分配各种资源；

8）访存：MMU将虚拟地址翻译成物理地址，并进行内存的访问；

9）申请动态内存：通过调用malloc（如printf）或者从堆中申请所需大小的块；

10）信号反馈：在接收到如ctrl+c或ctrl+z之类的信号是，会停止或者挂起hello；

1. 11）结束：子进程被父进程回收。

**（结论0分，缺失 -1分，根据内容酌情加分）**

# 附件

|  |  |
| --- | --- |
| hello.i | 预处理得到的文本 |
| hello.s | 编译得到的汇编文件 |
| hello.o | 汇编得到的可重定位目标文件 |
| hello | 链接生成的可执行目标文件 |
| hello.objdump | hello的反汇编代码 |
| helloo.objdump | hello.o的反汇编代码 |
| hello.elf | hello的ELF文件 |
| helloo.elf | hello.o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] Bryant,R.E. 《深入理解计算机系统》. 北京：机械工业出版社，2016-11-15

[2] Pianistx， [转]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）**