

**计算机系统**

**大作业**

题 目 程序人生-Hello’s P2P

专 业 计算机类

学　　 号 1190200122

班　　 级 1903001

学 生 袁野

指 导 教 师 郑贵滨

**计算机科学与技术学院**

**2021年6月**

**摘 要**

本论文旨在研究hello在linux系统下的整个生命周期。 结合《深入理解计算机系统》课本，通过gcc、edb、bash等工具进行实验，亲手将一个代码的所有历程直观地展示在电脑界面上，从而使学生在学习这门课时得以更加深刻地感受。

**关键词：**CSAPP；Hello；预处理；编译；汇编； 链接；进程；C语言；

**博客：** [深入理解计算机系统——程序人生 | Youngsc's Blog](https://youngsc.cc/2021/06/30/CSAPP_Final_Work/)

**目 录**

[第1章 概述 - 5 -](#_Toc79938663)

[1.1 Hello简介 - 5 -](#_Toc79938664)

[1.2 环境与工具 - 6 -](#_Toc79938665)

[1.2.1 硬件环境 - 6 -](#_Toc79938666)

[1.2.2 软件环境 - 6 -](#_Toc79938667)

[1.2.3 开发工具 - 6 -](#_Toc79938668)

[1.3 中间结果 - 7 -](#_Toc79938669)

[1.4 本章小结 - 7 -](#_Toc79938670)

[第2章 预处理 - 8 -](#_Toc79938671)

[2.1 预处理的概念与作用 - 8 -](#_Toc79938672)

[2.2在Ubuntu下预处理的命令 - 8 -](#_Toc79938673)

[2.3 Hello的预处理结果解析 - 8 -](#_Toc79938674)

[2.4 本章小结 - 9 -](#_Toc79938675)

[第3章 编译 - 10 -](#_Toc79938676)

[3.1 编译的概念与作用 - 10 -](#_Toc79938677)

[3.2 在Ubuntu下编译的命令 - 11 -](#_Toc79938678)

[3.3 Hello的编译结果解析 - 11 -](#_Toc79938679)

[3.3.0 指令 - 11 -](#_Toc79938680)

[3.3.1 数据 - 11 -](#_Toc79938681)

[3.3.2 赋值 - 12 -](#_Toc79938682)

[3.3.3 算术操作 - 13 -](#_Toc79938683)

[3.3.4 关系操作 - 13 -](#_Toc79938684)

[3.3.5 类型转换 - 13 -](#_Toc79938685)

[3.3.6 数组操作 - 13 -](#_Toc79938686)

[3.3.7 控制转移 - 14 -](#_Toc79938687)

[3.3.8 函数操作 - 14 -](#_Toc79938688)

[3.4 本章小结 - 15 -](#_Toc79938689)

[第4章 汇编 - 16 -](#_Toc79938690)

[4.1 汇编的概念与作用 - 16 -](#_Toc79938691)

[4.2 在Ubuntu下汇编的命令 - 16 -](#_Toc79938692)

[4.3 可重定位目标elf格式 - 16 -](#_Toc79938693)

[4.3.0 ELF结构与指令 - 16 -](#_Toc79938694)

[4.3.1 ELF Header - 17 -](#_Toc79938695)

[4.3.2 Section Headers - 17 -](#_Toc79938696)

[4.3.3 重定位节 - 18 -](#_Toc79938697)

[4.3.4 符号表 - 19 -](#_Toc79938698)

[4.4 Hello.o的结果解析 - 19 -](#_Toc79938699)

[4.5 本章小结 - 20 -](#_Toc79938700)

[第5章 链接 - 21 -](#_Toc79938701)

[5.1 链接的概念与作用 - 21 -](#_Toc79938702)

[5.2 在Ubuntu下链接的命令 - 21 -](#_Toc79938703)

[5.3 可执行目标文件hello的格式 - 21 -](#_Toc79938704)

[5.3.0 ELF结构与指令 - 21 -](#_Toc79938705)

[5.3.1 ELF Header - 22 -](#_Toc79938706)

[5.3.2 Section Headers - 23 -](#_Toc79938707)

[5.3.3 程序头表 - 23 -](#_Toc79938708)

[5.3.4 其余信息 - 24 -](#_Toc79938709)

[5.4 hello的虚拟地址空间 - 25 -](#_Toc79938710)

[5.5 链接的重定位过程分析 - 26 -](#_Toc79938711)

[5.6 hello的执行流程 - 26 -](#_Toc79938712)

[5.7 Hello的动态链接分析 - 27 -](#_Toc79938713)

[5.8 本章小结 - 27 -](#_Toc79938714)

[第6章 hello进程管理 - 28 -](#_Toc79938715)

[6.1 进程的概念与作用 - 28 -](#_Toc79938716)

[6.2 简述壳Shell-bash的作用与处理流程 - 28 -](#_Toc79938717)

[6.3 Hello的fork进程创建过程 - 28 -](#_Toc79938718)

[6.4 Hello的execve过程 - 29 -](#_Toc79938719)

[6.5 Hello的进程执行 - 30 -](#_Toc79938720)

[6.6 hello的异常与信号处理 - 31 -](#_Toc79938721)

[6.7本章小结 - 33 -](#_Toc79938722)

[第7章 hello的存储管理 - 34 -](#_Toc79938723)

[7.1 hello的存储器地址空间 - 34 -](#_Toc79938724)

[7.2 Intel逻辑地址到线性地址的变换-段式管理 - 34 -](#_Toc79938725)

[7.3 Hello的线性地址到物理地址的变换-页式管理 - 35 -](#_Toc79938726)

[7.4 TLB与四级页表支持下的VA到PA的变换 - 36 -](#_Toc79938727)

[7.5 三级Cache支持下的物理内存访问 - 36 -](#_Toc79938728)

[7.6 hello进程fork时的内存映射 - 37 -](#_Toc79938729)

[7.7 hello进程execve时的内存映射 - 37 -](#_Toc79938730)

[7.8 缺页故障与缺页中断处理 - 38 -](#_Toc79938731)

[7.9动态存储分配管理 - 38 -](#_Toc79938732)

[7.9.1 隐式空闲链表 - 39 -](#_Toc79938733)

[7.9.2 显式空闲链表 - 39 -](#_Toc79938734)

[7.10本章小结 - 39 -](#_Toc79938735)

[第8章 hello的IO管理 - 41 -](#_Toc79938736)

[8.1 Linux的IO设备管理方法 - 41 -](#_Toc79938737)

[8.2 简述Unix IO接口及其函数 - 41 -](#_Toc79938738)

[8.3 printf的实现分析 - 42 -](#_Toc79938739)

[8.4 getchar的实现分析 - 43 -](#_Toc79938740)

[8.5本章小结 - 44 -](#_Toc79938741)

[结 论 - 45 -](#_Toc79938742)

[附件 - 46 -](#_Toc79938743)

[参考文献 - 47 -](#_Toc79938744)

# 第1章 概述

## 1.1 Hello简介

**1、P2P-From Program to Process**

P2P的过程即为程序到进程的过程，首先我们需要了解程序与进程这两个名词的概念。

程序：计算机程序（Computer Program）是一组计算机能识别和执行的指令，运行于电子计算机上，满足人们某种需求的信息化工具。它以某些程序设计语言编写，运行于某种目标结构体系上。为了使计算机程序得以运行，计算机需要加载代码，同时也要加载数据。从计算机的底层来说，这是由高级语言（例如Java，C/C++，C#等）代码转译成机器语言而被CPU所理解，进行加载。如果在一个符合大多数的计算机上，操作系统例如Windows、Linux等，加载并执行很多的程序，在这种情况下，每一个程序是一个单独的映射，并不是计算机上的所有可执行程序。它是指为了得到某种结果而可以由计算机等具有信息处理能力的装置执行的代码化指令序列，或者可以被自动转换成代码化指令序列的符号化指令序列或者符号化语句序列。同一计算机程序的源程序和目标程序为同一作品。

进程：进程（Process）是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。在早期面向进程设计的计算机结构中，进程是程序的基本执行实体；在当代面向线程设计的计算机结构中，进程是线程的容器。程序是指令、数据及其组织形式的描述，进程是程序的实体。

GCC编译器读取源代码文件hello.c，并且将其翻译成一个可执行文件hello，过程如下：

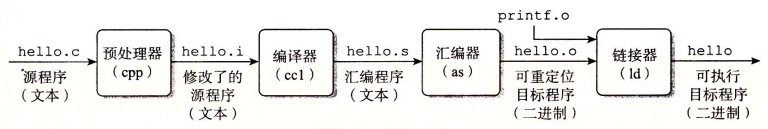


图 1‑1 hello.c的编译过程

预处理阶段：预处理器（cpp）根据以字符#开头的命令，修改原始的C程序，将#include命令包含的头文件/源代码读取并把它直接插入程序文本中。结果就得到了另一个C程序hello.i；

编译阶段：编译器（ccl）将文本文件hello.i翻译成文本文件hello.s，它包含一个汇编语言程序；

汇编阶段：接下来，汇编器（as）将hello.s翻译成机器语言指令，把这些指令打包成一种叫做可重定位目标程序的格式，并将结果保存在目标文件hello.o中。它是一个二进制文件；

链接阶段：最后，链接器（ld）将程序与函数库中需要使用的二进制文件进行链接，形成可执行目标程序二进制文件。

生成可执行文件hello之后，该文件加载在内存中，由系统执行，在shell中键入启动命令后，shell为其fork形成一个子进程，分配相应的内存资源，使用execve函数加载进程（Process），于是hello便从Program变为Process，这便是P2P的过程。

**2、020-From Zero-0 to Zero-0**

从“零”到“零”指的是一个程序从开始执行到结束，从“零”始到“零”终。

为执行hello程序，系统Shell首先fork一个子进程，再用execve函数加载目标程序，并用目标程序的进程取代当前进程。在这之前，在程序头部表的引导下，加载器将可执行文件的片复制到代码段和数据段，栈空间和堆空间也都被重新初始化，其余各段也被替代。接着，CPU为新进程进程分配时间片执行逻辑控制流，系统为进程映射虚拟内存，然后跳转到程序的入口点，也就是\_start函数的地址。这个函数是在系统目标文件ctrl.o中定义的，对所有C程序来说都是如此。\_start函数调用系统启动函数\_\_libc\_start\_main，该函数定义在libc.o中。它初始化执行环境，调用main函数，处理main函数的返回值，并在及程序结束后把控制返回给内核。进程结束后，父进程将其回收，避免资源浪费，整个系统恢复到程序执行之前，即为020。

## 1.2 环境与工具

### 1.2.1 硬件环境

Legion Y7000P 2019 PG0

CPU:Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz (12 CPUs), ~2.6GHz

RAM: 16384MB

### 1.2.2 软件环境

Windows 10 家庭中文版 64-bit

Ubuntu 20.04.2 LTS

VMware® Workstation 16 Player 16.1.0 build-17198959

### 1.2.3 开发工具

Microsoft Visual Studio Community 2019版本 16.9.2

Microsoft Visual 1.54.3

GCC 9.3.0

## 1.3 中间结果

|  |  |
| --- | --- |
| **文件名称** | **文件作用** |
| hello.c | hello的C语言源代码 |
| hello.i | hello.c预处理生成的代码 |
| hello.s | hello.i编译生成的汇编代码 |
| hello.o | hello.s汇编生成的可重定位目标文件 |
| hello | hello.o链接生成的可执行文件 |

表格 1-1 中间结果文件名称及其作用说明

## 1.4 本章小结

阐述了程序和进程的概念、hello.c到可执行文件的编译过程以及可执行文件执行过程，提供了本实验的软硬件环境、开发工具等信息，列出了程序编译的中间过程的文件名称及作用。

# 第2章 预处理

## 2.1 预处理的概念与作用

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。主要功能如下：

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hello.c第6-8行中的#include等命令告诉预处理器读取系统头文件stdio.h、unistd.h、stdlib.h的内容，并把它直接插入到程序文本中。

2、用实际值替换用#define定义的字符串

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

预处理过程我们有两种选择：

A、gcc -E hello.c -o hello.i  
 B、cpp hello.c > hello.i



图 2‑1 预处理命令

## 2.3 Hello的预处理结果解析

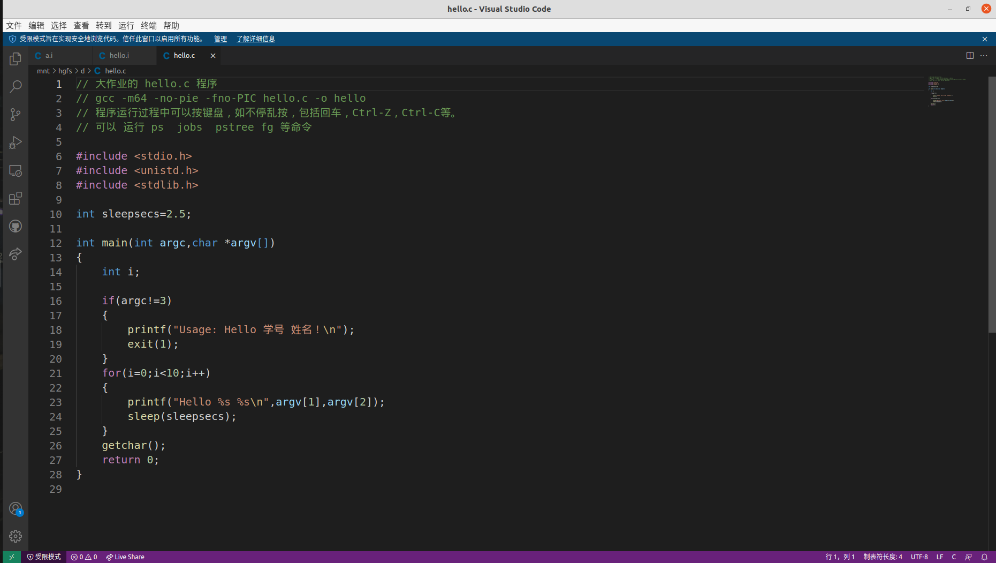


图 2‑2 VS code打开hello.c内容

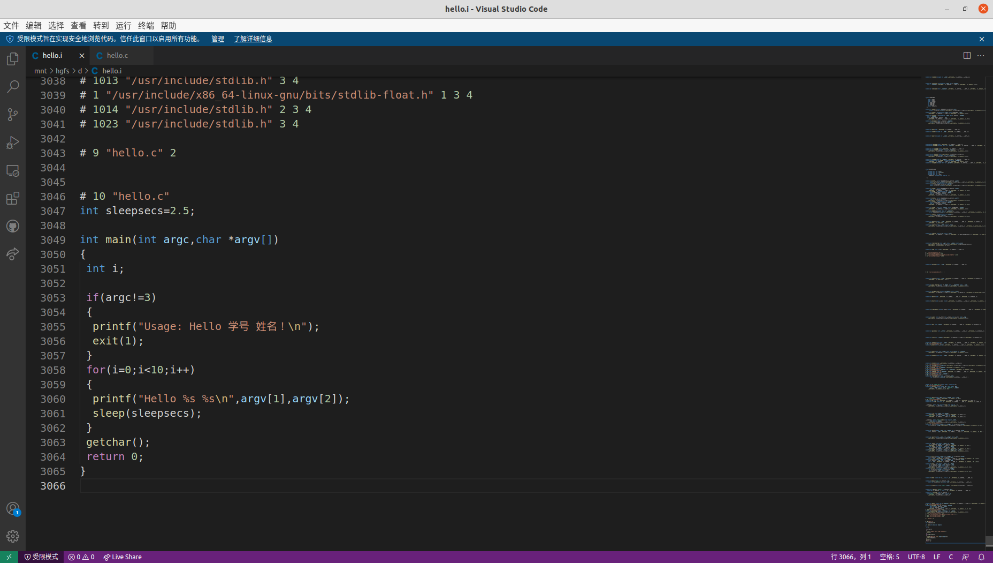


图 2‑3 VS code打开hello.i的内容

经过预处理后的hello.i文件为3065行，远大于原先hello.c的28行。

通过观察我们发现之前的#include部分内容全部消失，取而代之的是各个头文件的具体内容。以stdio.h的展开为例，cpp到默认的环境变量下寻找stdio.h，打开/usr/include/stdio.h发现其中依然使用了#define语句，cpp对此递归展开，所以最终.i程序中是没有#define的。而且发现其中使用了大量的#ifdef #ifndef的语句，cpp会对条件值进行判断来决定是否执行包含其中的逻辑。同时hello.c中对于代码翻译无意义的注释也消失。代码的主体内容得以保留。

## 2.4 本章小结

我们介绍了对于.c文件的预处理的指令以及预处理的效果，解析了预处理之后的结果，为后续步骤的进行做好基础。

# 第3章 编译

## 3.1 编译的概念与作用

编译器会将某一种编程语言携程的源代码转换为另外一种编程语言，该过程我们称之为编译。在gcc编译C语言源程序的过程中，ccl将预处理后的.i文件翻译为.s文本文件，该文件包含一个汇编语言程序。

高级计算机语言便于人编写，阅读，维护。低阶机器语言是计算机能直接解读、运行的。编译器主要的目的是将便于人编写，阅读，维护的高级计算机语言所写作的源代码，翻译为计算机能解读、运行的低阶机器语言的程序。编译器将原始程序（Source program）作为输入，翻译产生使用目标语言（Target language）的等价程序。源代码一般为高阶语言 (High-level language), 如Pascal、C、C++、C#、Java等，而目标语言则是汇编语言或目标机器的目标代码（Object code），有时也称作机器代码（Machine code）。

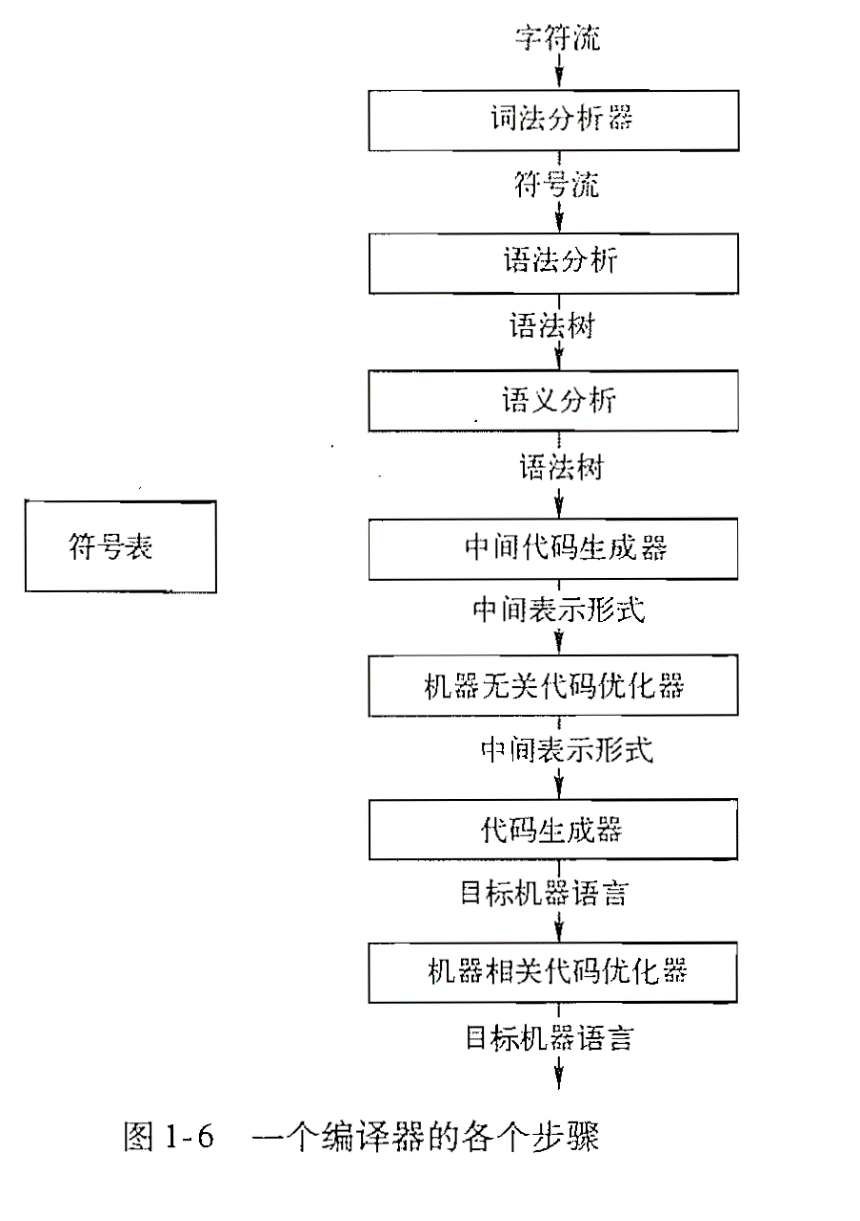


图 3‑1 编译器的各个步骤

编译程序把一个源程序翻译成目标程序的工作过程分为六个阶段：词法分析；语法分析；语义分析；中间代码生成；代码优化；目标代码生成。

## 3.2 在Ubuntu下编译的命令



图 3‑2 gcc的编译过程指令

## 3.3 Hello的编译结果解析

### 3.3.0 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中使用到的数据类型有：整数、字符串、数组。

#### 3.3.1.1 整数

hello.c中的整数有：i、argc、sleepsecs。

1. **i:** 上述代码为i变量的初始化为0操作，由于i为临时变量，因此被定义在栈上，通过栈顶相对寻址来进行访问。



图 3‑3 i在栈中的赋值初始化

1. **argc:** 作为第一个参数，argc最初在寄存器%edi中，随后被复制到栈里进行操作，通过栈顶相对寻址的方式进行访问。



图 3‑4 argc在栈中的赋值操作

1. **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3.3中，可以看到，编译器首先将sleepsecs在.text代码段中声明为全局变量，其次在.data段中，设置对齐方式为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。

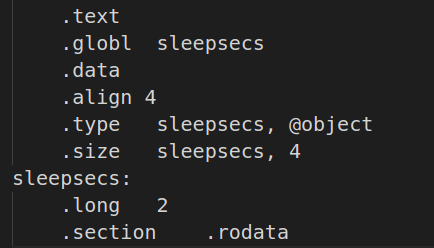


图 3‑5 sleepsecs的定义

#### 3.3.1.2 字符串

程序中的字符串分别是：

1) “Usage: Hello 学号 姓名！\n”，第一个printf传入的输出格式化参数，在hello.s中声明如图3-6，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中声明如图3-6。

字符串都声明在.section与.rodata中。

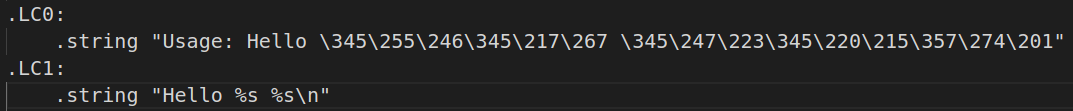


图 3‑6 字符串在hello.s中的定义

#### 3.3.1.3 数组

程序中的数组为argv[]，作为第二个参数传入，是一个字符串数组，存储在栈上，在hello.s中，我们可以通过-32(%rbp)访问。



图 3‑7 argv在程序中的访问

### 3.3.2 赋值

程序中的赋值操作有：

1. int sleepsecs=2.5: 由于sleepsecs为全局变量，因此直接在.data节中将sleepsecs声明为值2的long类型数据。



图 3‑8 sleepsecs的定义

1. i=0：整型数据的赋值通过mov指令来完成，由于i是int类型,因此使用movl指令。



图 3‑9 i的赋值

### 3.3.3 算术操作

代码中的算术操作为i的自增，通过addl指令来实现。



图 3‑10 i自增在代码中的体现

### 3.3.4 关系操作

hello中的关系比较操作有：“!=”和“<”。

1. argc!=3：判断argc是否不等于3，通过cmpl函数来设置条件码ZF，若ZF等于1则两个数字相等，反之不等，为其他指令判断该条件是否成立做准备。



图 3‑11 argc!=3在代码中的体现

1. i<10：判断i是否小于10，同样通过cmpl函数，将i与9进行比较来设置条件码，若后者小于等于前者则表示循环未结束，接着进行循环，否则跳出。



图 3‑12 i<10在代码中的体现

### 3.3.5 类型转换

程序中设计的隐式类型转换int sleepsecs=2.5。由于在C中浮点数默认的数据类型为double，因此该语句为double转换为int，遵循向零舍入的原则，2.5被转化为int类型的2。

### 3.3.6 数组操作

hello.c中只对argv数组进行访问，通过间接寻址的方式，每次通过对%rax获得argv+x的值，然后通过(%rax)访问argv[x]。

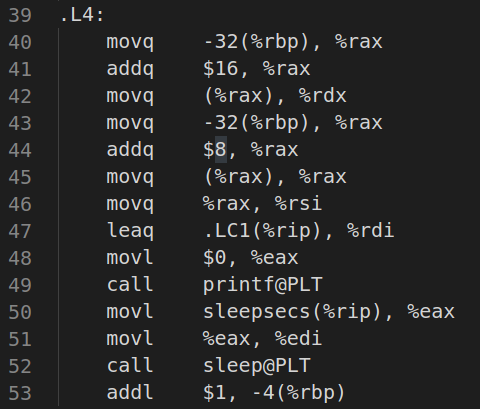


图 3‑13 数组元素的访问

### 3.3.7 控制转移

程序中涉及转移的有：

1. if语句中如果argc!=3将执行花括号中的代码片段，这里通过if实现了控制的跳转。



图 3‑14 if的跳转

1. for语句中在最后判断循环终止条件时，如果不满足则跳转到循环开始的地方重新执行，否则跳出循环。



图 3‑15 for的跳转

### 3.3.8 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。

程序中涉及函数操作的有：

1. main 函数：
   1. 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。
   2. 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。
   3. 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：
   1. 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]。
   2. 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：
   1. 传递数据：将%edi设置为1。
   2. 控制传递：call exit@PLT。
4. sleep 函数：
   1. 传递数据：将%edi设置为sleepsecs。
   2. 控制传递：call sleep@PLT。
5. getchar 函数：
   1. 控制传递：call gethcar@PLT

## 3.4 本章小结

本章了解了代码有高级源代码转换为低级汇编代码的编译过程，ccl将hello.c转换为hello.s的过程即为编译，此外解析了hello的编译结果，涵盖了许多常见的指令。由于编译平台的不同，导致最终的运行效率产生差异。

# 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。这个过程称为汇编，亦即汇编的作用。通过这个过程，我们可以将汇编代码转化为机器可以理解的机器码。

## 4.2 在Ubuntu下汇编的命令

预处理过程我们有两种指令：

A、as hello.s -o hello.o  
 B、gcc hello.s -c -o hello.o



图 4‑1 汇编指令

## 4.3 可重定位目标elf格式

### 4.3.0 ELF结构与指令

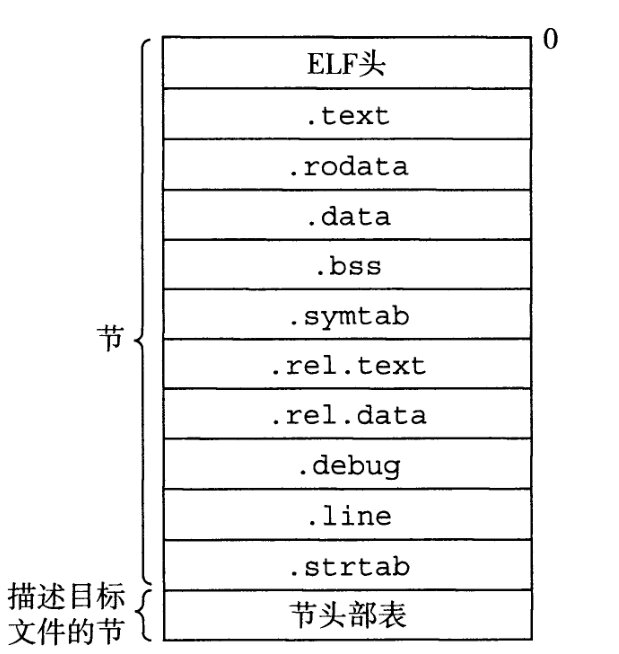


图 4‑2 可重定位目标文件ELF格式



图 4‑3 elf文件指令

ELF是一种Unix二进制文件，它可能是可链接文件，也可能是可执行文件。

### 4.3.1 ELF Header

ELF头以一个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。

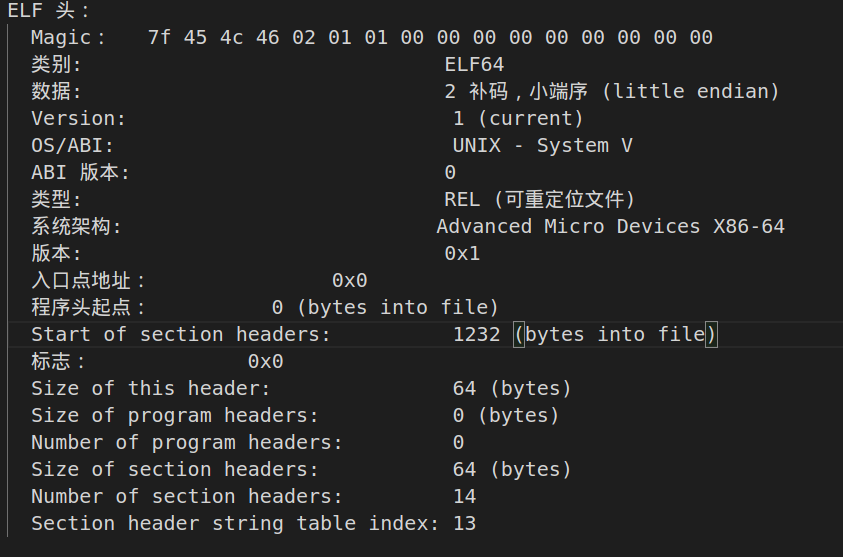


图 4‑4 ELF Header

### 4.3.2 Section Headers

节头部表，用于描述目标文件的节，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。

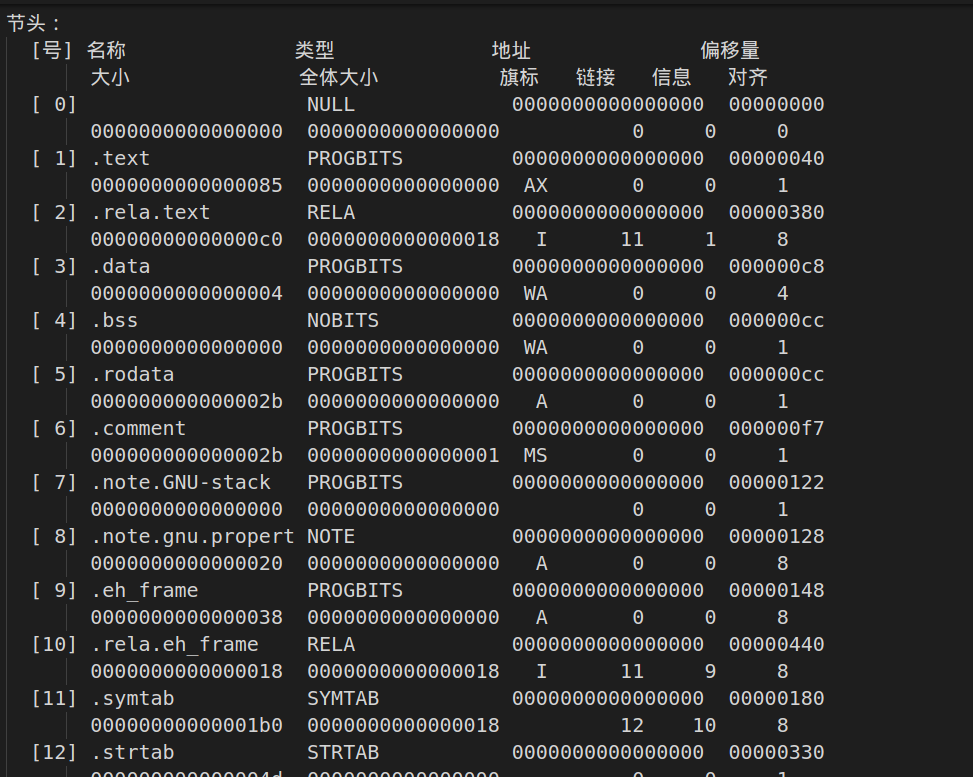


图 4‑5 ELF Section Headers

### 4.3.3 重定位节

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4-6，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。

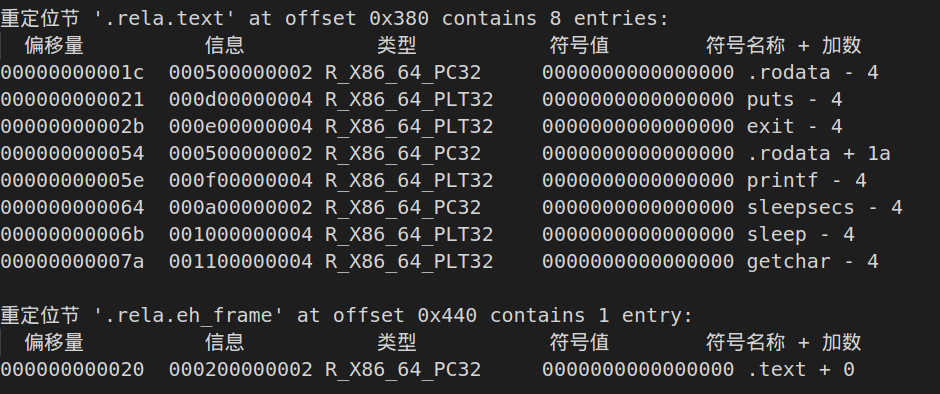


图 4‑6 重定位节

1. 偏移量：指所引用的符号的相对偏移，或者说符号应该填在程序的哪个位置。例如，第二行中，puts的偏移量为0x000000000021。这就相当于告诉链接器，需要修改开始于偏移量0x21处的32位PC相对引用，使它在运行时指向puts函数。
2. 信息，包括符号和类型两部分，共占8个字节。其中，前4个字节表示符号，后4个字节表示类型。符号代表重定位到的目标在.symtab节中的偏移量，类型则包括相对地址引用和绝对地址应用。
3. 类型，就是对第二列中类型信息的翻译。
4. 符号值，就是符号代表的值。
5. 符号名称是重定位目标的名字，可能是节名、变量名、函数名等；加数则是用于对被修改的引用值做偏移调整。

### 4.3.4 符号表

符号表保存了程序中所用的各种符号的信息，包括文件名、函数名、全局变量名、静态（私有）变量名等。

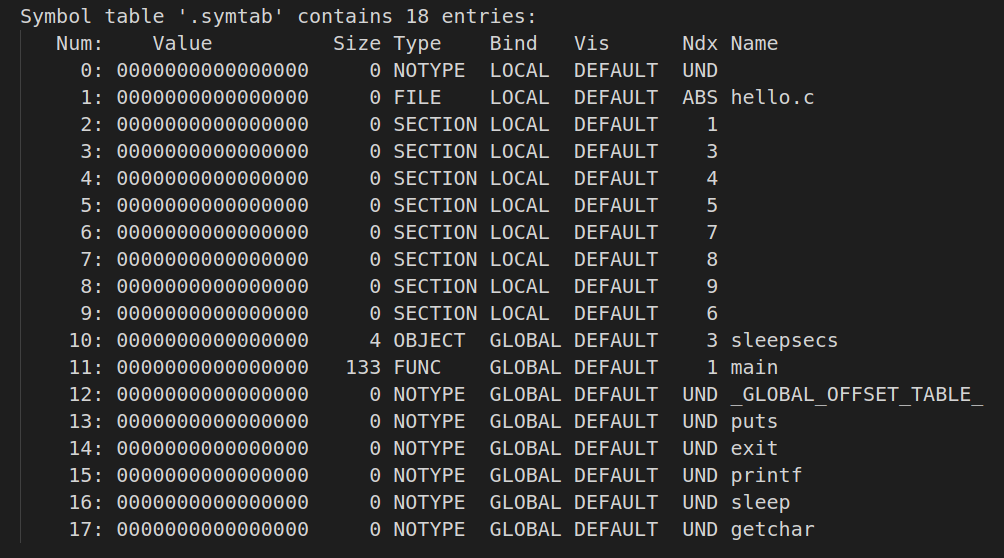


图 4‑7 符号表

## 4.4 Hello.o的结果解析

我们使用命令objdump -d -r hello.o > helloo.objdump获得反汇编代码。

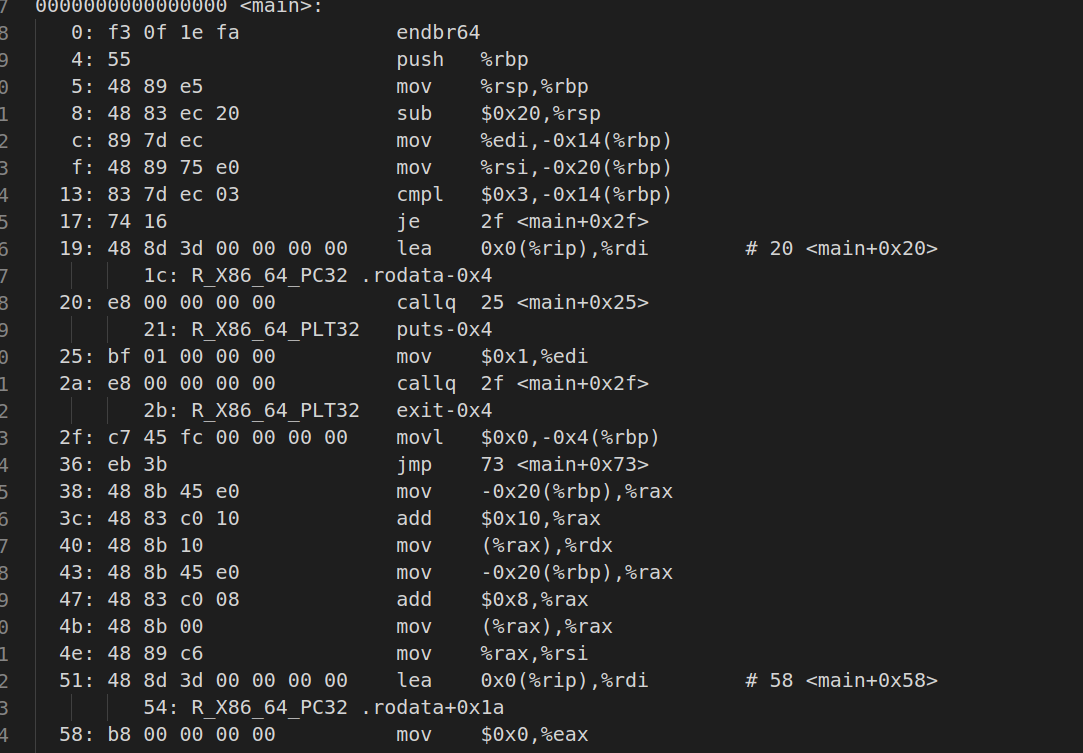


图 4‑8 反汇编代码

将hello.s与hello.objdump对比之后我们发现：

1. 伪指令消失：hello.s中许多“.”开头的伪指令在hello.objdump消失。
2. 条件分支变化：在hello.s中的段名称（如.L1、.L2）全部消失，取而代之的则是确定的相对偏移地址。如hello.s中的jmp .L3在hello.objdump中变为了jmp 73 <main+0x73>。
3. 函数调用变化：在hello.s我们调用puts等来自静态库或者其他的文件，需要进行链接才能调用的函数时，直接call+函数名，但是在反汇编的代码中我们发现对应的机器码是e8 00 00 00 00（即call 0），反汇编的汇编指令是call+相对地址，后面添加注释便于重定位。
4. 数据访问变化：在hello.s中，我们访问字符串常量是通过一些助记符访问的。而在反汇编的代码中是通过0x0(%rip)访问，同样有注释，便于重定位。如hello.s中的leaq .LC0(%rip), %rdi在hello.objdump中变为了lea 0x0(%rip),%rdi。

## 4.5 本章小结

本章介绍了从 hello.s到hello.o的汇编过程，汇编器（as）将汇编代码hello.s转化为可重定位目标文件hello.o，得到一个可以用于链接的二进制文件，通过查看hello.o的elf格式和使用objdump得到反汇编代码与hello.s进行比较的方式，间接了解到从汇编语言映射到机器语言汇编器需要实现的转换。

# 第5章 链接

## 5.1 链接的概念与作用

链接是C源代码编译的最后一步，它是指为了生成可执行文件，将有关的目标文件连接起来，使得这些目标文件成为操作系统可以装载执行的统一整体的过程。例如，在A文件中引用的符号将同该符号在B文件中的定义连接起来，从而形成一个可执行的整体。

链接分为两种模式：

1. 静态链接。静态链接时，外部函数的代码将从其所在的静态链接库中*拷贝*到最终的可执行程序中。这样，程序执行时，这些代码就会被装入到对应进程的虚拟内存空间里。这里的静态链接库实际上是一个目标文件的集合，其中的每个文件含有库中的一个或者一组相关函数的代码。
2. 动态链接。动态链接中，外部函数的代码被放到动态链接库或共享对象的某个目标文件中（通常以.so为后缀名）。链接器在链接时所做的只是在生成的可执行文件中记下共享对象的名字等少量信息。在可执行文件运行行时，动态链接库的全部内容将被映射到相应进程的虚拟内存空间。动态链接程序将根据可执行程序中记录的信息找到相应的函数代码。

## 5.2 在Ubuntu下链接的命令

执行的命令为：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

s

图 5‑1 链接命令

## 5.3 可执行目标文件hello的格式

### 5.3.0 ELF结构与指令

使用 readelf -a hello > hello.elf 命令生成 hello 程序的 ELF 格式文件。

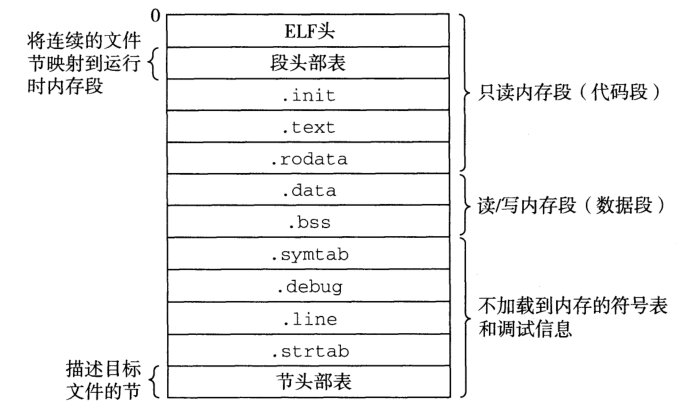


图 5‑2 可执行目标文件的ELF格式

### 5.3.1 ELF Header

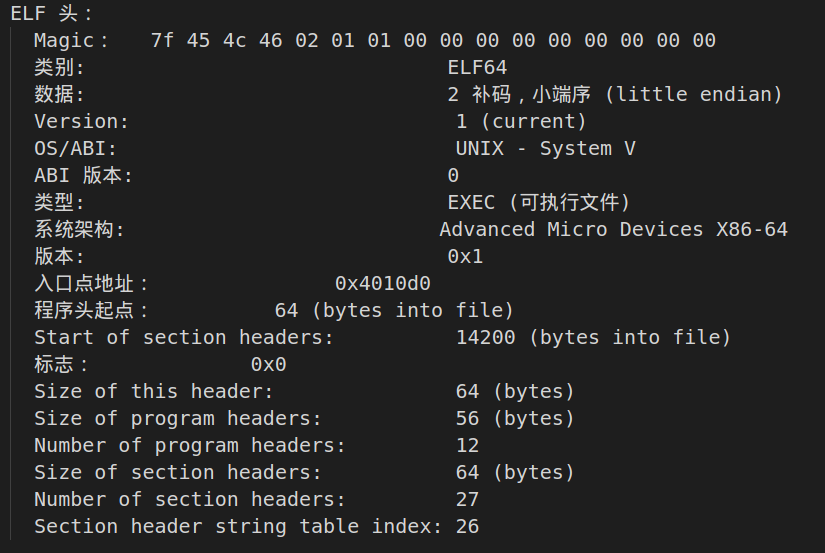


图 5‑3 ELF Header

也可以直接输入命令 readelf -a hello查看ELF头信息。

ELF头以一个16字节的序列开始，这个序列描述了生成该文件的系统的节的大小和字节顺序。ELF头剩下的部分还包括ELF头的大小、目标文件的类型、机器类型、节头部表的文件偏移、以及节头部表的大小和数目。

### 5.3.2 Section Headers

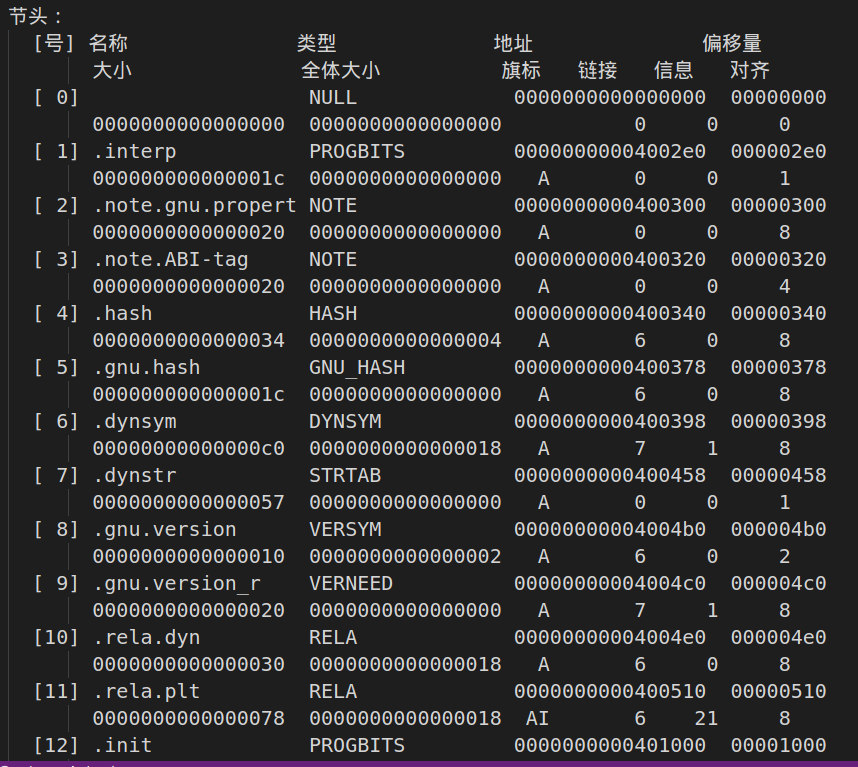


图 5‑4 Section Headers

我们会发现相比与hello.o，hello的节头部表有25项，比hello.o多了12项。节头部表对hello中所有的节信息进行了声明，包括大小size以及在程序中的偏移量offset，大小、全体大小、旗标、链接、信息、对齐等信息，并可以根据此定位各个节所占的区间。由于链接时使用的是动态链接，所以hello可执行文件仍然具有.rela.\*的重定位节，便于使用动态链接共享库。.init节定义了一个小函数\_init，程序的初始化代码会调用它。

### 5.3.3 程序头表

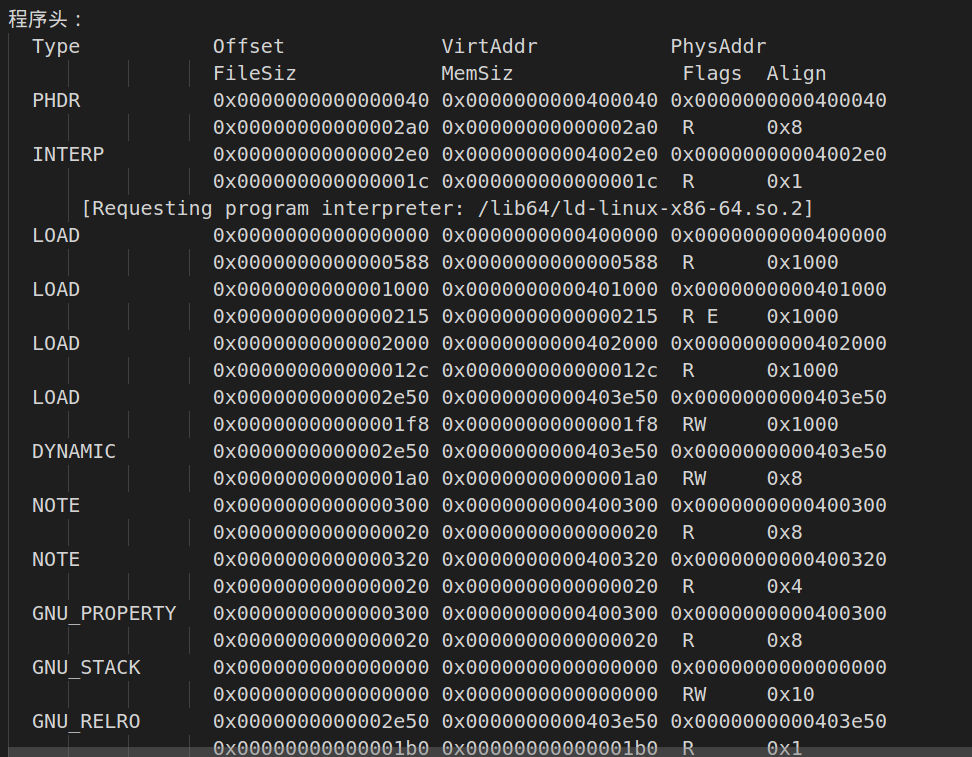


图 5‑5 程序头表

程序头部表描述了可执行文件的连续的片和连续的内存段的映射关系。

### 5.3.4 其余信息

段映射如下：

s

图 5‑6 段映射

动态节项目信息如下：

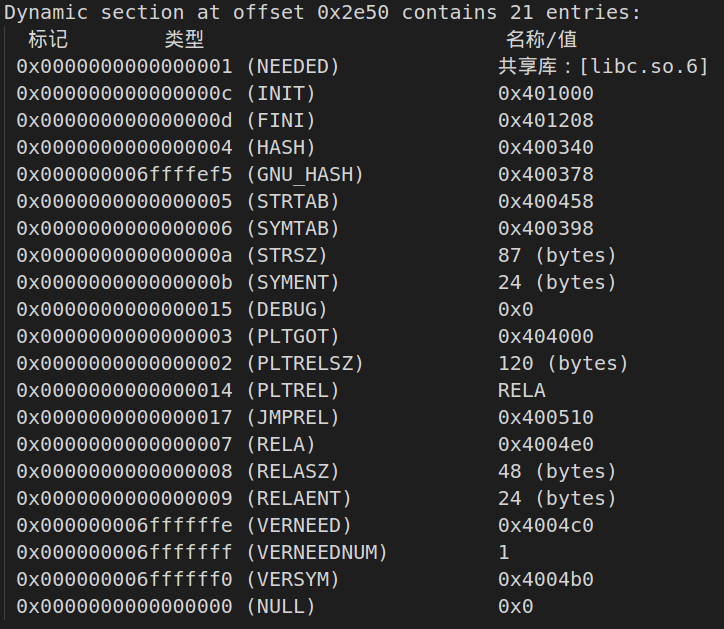


图 5‑7 动态节项目信息

重定位节信息如下：

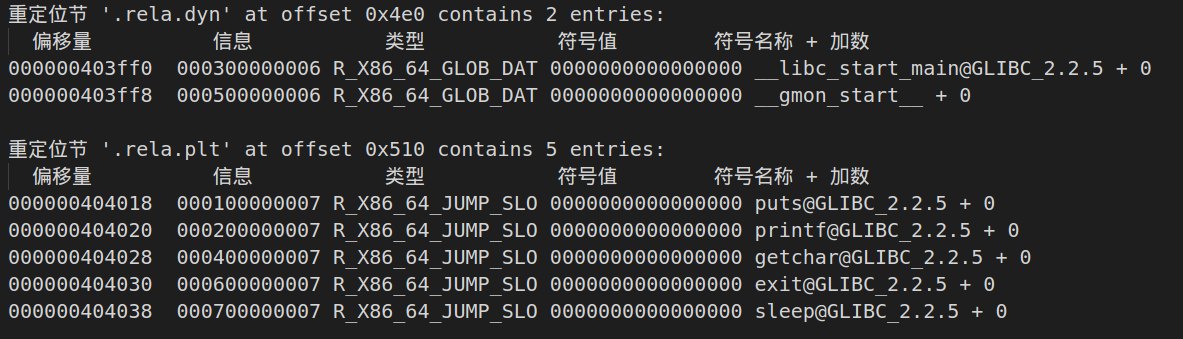


图 5‑8 重定位节

符号表如下：

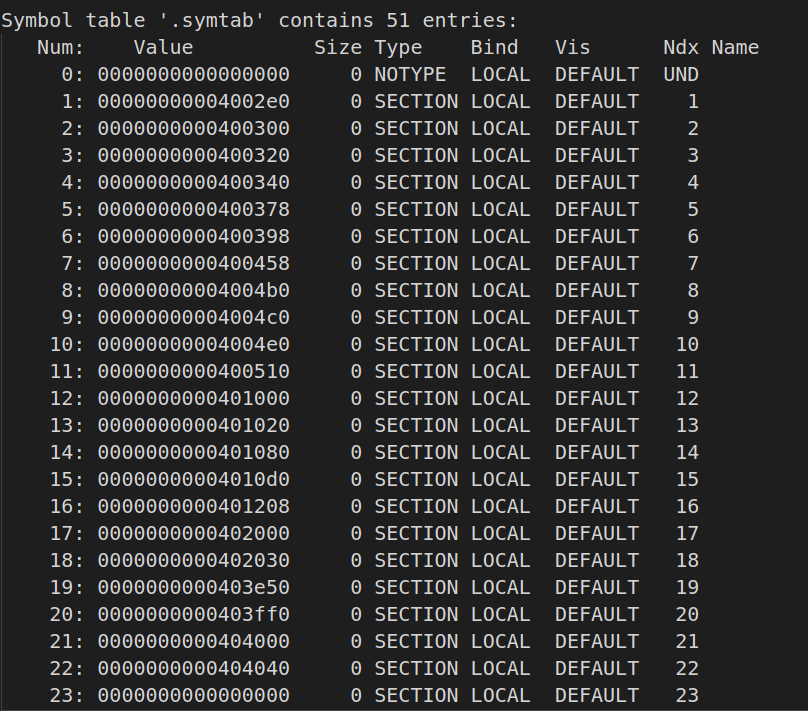


图 5‑9 符号表

## 5.4 hello的虚拟地址空间

Edb打开hello后的Data Dump如下：

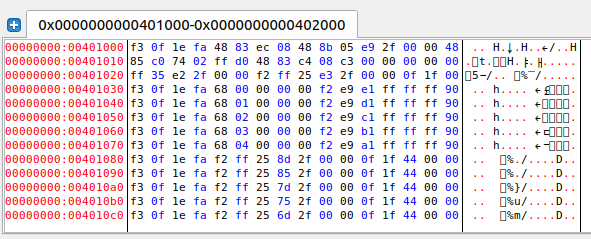


图 5‑10 Data Dump

由图5-10我们可以看到hello隔断的虚拟地址空间被限制在0x401000到0x402000之间。

查看图5-5我么可以看到ELF格式文件中的Program Headers，程序头表在执行的时候被使用，它告诉链接器运行时加载的内容并提供动态链接的信息。每一个表项提供了各段在虚拟地址空间和物理地址空间的大小、位置、标志、访问权限和对齐方面的信息。在下面可以看出，程序包含7个段：

1. PHDR 保存程序头表。
2. INTERP 指定在程序已经从可执行文件映射到内存之后，必须调用的解释器（如动态链接器）。
3. LOAD 表示一个需要从二进制文件映射到虚拟地址空间的段。其中保存了常量数据（如字符串）、程序的目标代码等。
4. DYNAMIC 保存了由动态链接器使用的信息。
5. NOTE 保存辅助信息。
6. GNU\_STACK：权限标志，标志栈是否是可执行的。
7. GNU\_RELRO：指定在重定位结束之后那些内存区域是需要设置只读。

## 5.5 链接的重定位过程分析

使用objdump -d -r hello > hello.d得到hello的反汇编的代码。

将hello.d与hello.objdump进行对比我们可以发现：

1. 函数个数：在使用ld命令链接的时候，指定了动态链接器为64的/lib64/ld-linux-x86-64.so.2，crt1.o、crti.o、crtn.o中主要定义了程序入口\_start、初始化函数\_init，\_start程序调用hello.c中的main函数，libc.so是动态链接共享库，其中定义了hello.c中用到的printf、sleep、getchar、exit函数和\_start中调用的\_\_libc\_csu\_init，\_\_libc\_csu\_fini，\_\_libc\_start\_main。链接器将上述函数加入。
2. 函数调用：链接器解析重定条目时发现对外部函数调用的类型为R\_X86\_64\_PLT32的重定位，此时动态链接库中的函数已经加入到了PLT中，.text与.plt节相对距离已经确定，链接器计算相对距离，将对动态链接库中函数的调用值改为PLT中相应函数与下条指令的相对地址，指向对应函数。对于此类重定位链接器为其构造.plt与.got.plt。
3. .rodata引用：链接器解析重定条目时发现两个类型为R\_X86\_64\_PC32的对.rodata的重定位（printf中的两个字符串），.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。

## 5.6 hello的执行流程

|  |  |
| --- | --- |
| 程序名 | 程序地址 |
| 加载hello | |
| ld-linux-x86-64.so!\_dl\_start | 0x00007ffff7fd4d30 |
| ld-linux-x86-64.so!\_dl\_init | 0x00007ffff7fe27b0 |
| hello!\_start | 0x0000000000401090 |
| hello!\_\_libc\_csu\_init | 0x00000000004010d0 |
| hello!\_init | 0x0000000000401000 |
| libc.so!\_setjmp | 0x00007ffff7e08b10 |
| 程序运行 | |
| hello!main | 0x0000000000401149 |
| hello!puts@plt | 0x0000000000401030 |
| ld-linux-x86-64.so!\_dl\_runtime\_resolve\_xsave | 0x00007ffff7fe87a0 |
| ld-linux-x86-64.so!\_dl\_fixup | 0x00007ffff7fe1de0 |
| ld-linux-x86-64.so!\_dl\_lookup\_symbol\_x | 0x00007ffff7fdd610 |
| 退出程序 | |
| hello!exit@plt | 0x0000000000401070 |
| libc.so!exit | 0x00007ffff7e0b840 |
| hello!\_fini | 0x00000000004011d4 |

表格 2 hello的执行流程

## 5.7 Hello的动态链接分析

函数调用一个由共享库定义的函数时，编译器无法预先判断出函数的地址，因为定义它的共享模块在运行时可以加载到任意位置。GNU编译系统使用延迟绑定的方式解决该问题，在运行时动态载入。

延迟绑定通过两个数据结构之间简洁但又有些复杂的交互来实现，即过程链接表（PLT）和全局偏移量表（GOT）。

过程链接表（PLT）：PLT是一个数组，其中每个条目是16字节代码。PLT [0]是一个特殊条目，它跳转到动态链接器中。每个被可执行程序调用的库函数都有它自己的PLT条目。每个条目都负责调用一个具体的函数。每个条目都负责调用一个具体的函数。

全局偏移量表（GOT）：GOT是一个数组，其中每个条目是8字节地址。和PLT联合使用时，GOT [0]和GOT [1]包含动态链接器在解析函数地址时会使用的信息。GOT [2]是动态链接器在1d-linux.so模块中的入口点。其余的每个条目对应于一个被调用的函数，其地址需要在运行时被解析。每个条目都有一个相匹配的PLT条目。

## 5.8 本章小结

本章介绍了链接的概念及作用，分析了hello的ELF格式，深入学习了hello.o 可重定位文件到hello可执行文件的流程，和链接的各个过程介绍了链接器如何将hello.o可重定向文件与动态库函数链接起来，链接技术可以允许我们把大项目分解成小模块、小功能。当我们改变某个小模块时，只需简单地重新编译它并重新链接应用，而无需编译其他文件，正是因为有了技术的存在，代码的编译才会变得更加高效。

# 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，一般情况下，包括文本区域、数据区域、和堆栈。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储区着活动过程调用的指令和本地变量。

作用：进程为用户提供了以下假象：我们的程序好像是系统中当前运行的唯一程序一样，我们的程序好像是独占的使用处理器和内存，处理器好像是无间断的执行我们程序中的指令，我们程序中的代码和数据好像是系统内存中唯一的对象。每次用户通过shell输入一个可执行目标文件的名字，运行程序时，shell就会创建一个新的进程，应用程序也可以创建新进程，并且在这个新进程的上下文中运行它们自己的代码或者其他应用程序。这使得我们可以同时运行多个程序。

## 6.2 简述壳Shell-bash的作用与处理流程

作用：Shell是一种壳层与命令行界面，是操作系统下传统的用户和计算机的交互界面，使可以通过这个界面访问内核提供的服务。

处理流程：

1. 从终端读入输入的命令。
2. 将输入字符串切分并解析获得所有的参数。
3. 如果是内置命令则立即调用相应的函数执行。
4. 否则创立一个子进程并在子进程中使用execve运行该程序。
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

父进程可以通过fork函数创建一个新子进程。函数原型为pid\_t fork(void);

函数返回值分两种情形，父进程内返回子进程的PID，子进程内返回0。新创建的子进程与父进程几乎完全相同。子进程得到与父进程用户级虚拟地址空间相同（但是独立的）一份副本，包括代码段和数据段、堆、共享库以及用户栈。子进程还会获得父进程所打开的文件描述符的副本，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。父进程和新创建的子进程之间最大的差别在于它们有不同的PID。

fork函数调用一次，返回两次。父进程调用一次fork，一次是返回到父进程，而另一次是返回到子进程的。父进程和子进程是并发运行的独立进程，内核可以以任意方式交替执行它们的逻辑控制流中的指令。我们不能对不同进程中指令的交替执行做任何假设。两个进程有相同的用户栈、运行时堆和本地变量值等，但它们对各自内存空间的修改是相互独立的。事实上，在物理内存中，一开始，两个进程指向的地址确实是相同的；但是，一旦一方对部分共享空间做了修改，这部分空间就会被拷贝出去，不再共享。共享文件。子进程会继承父进程打开的所有文件。

## 6.4 Hello的execve过程

shell会通过execve调用hello。execve函数加载并运行可执行目标文件hello，且带参数列表argv和环境变量列表envp。只有当出现错误时，例如找不到hello，execve才会返回到调用函数。所以与fork调用一次返回两次不一样，execve调用一次并且从不返回。

execve函数在当前进程的上下文中加载并运行一个新的函数。它会覆盖当前进程的地址空间，但并没有创建一个新进程。新的函数仍然有相同的PID，并且继承了调用execve函数时已打开的所有文件描述符。

载入并执行hello需要以下几个步骤：

1. 删除已存在的使用者区域：删除当前程序虚拟地址的使用者部分中已存在的区域结构。
2. 对映私有区域：为新程序的代码、数据、.bss和栈区域建立新的区域结构。所有这些新的区域都是私有的、写时复制的。代码和数据被对映为hello档案中的.text和.data区。.bss区域是请求二进位制零的，对映到初值为0的匿名文件，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. 对映共享区域：如果hello程式与共享物件连结，那么这些物件都是动态连结到这个程式的，然后再对映到使用者虚拟地址空间中的共享区域内。
4. 设定程式计数器：设定当前程序上下文中的程序计数器，使之指向代码区域的入口点。下一次运行这个程序时，它将从这个入口点开始执行。

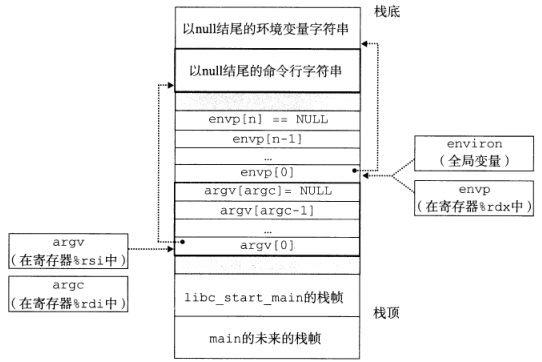


图 6‑1 新程序开始时用户栈的典型组织结构

## 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。

系统中每个程序都运行在某个进程的上下文中。上下文是由程序正确运行所需的状态组成的。这个状态包括存放在内存中的程序的代码和数据，它的栈、通用目的的寄存器的内容、程序计数器、环境变量以及打开文件描述符的集合。

内核为每个进程维护了一个上下文。当内核选择的一个新的进程运行时，我们说内核调度了这个进程。所以当内核调度了hello这个新的进程运行后，它就抢占当前进程，并使用一种称为上下文切换的机制来将控制转移到新的进程，上下文切换会首先保存当前进程的上下文，然后恢复新恢复进程被保存的上下文，最后控制传递给这个新恢复的进程 ，来完成上下文切换。

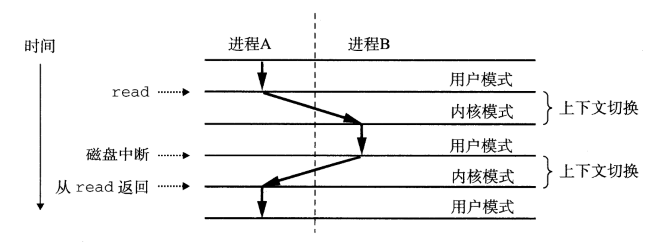
s

图 6‑2 进程间切换

## 6.6 hello的异常与信号处理

hello执行过程中出现的可能的异常种类会有四种：

1. 中断 中断是异步发生的，是来自处理器外部的I/O设备的信号的结果。硬件中断的异常处理程序被称为中断处理程序。
2. 陷阱 陷阱是有意的异常，是执行一条指令的结果。就像中断处理程序一样，陷阱处理程序将控制返回到下一条指令。陷阱最重要的用途是在用户程序和内核之间提供一个像过程一样的接口，叫做系统调用。
3. 故障 故障由错误情况引起，它可能能够被故障处理程序修正。当故障发生时，处理器将控制转移给故障处理程序。如果处理程序能够修正这个错误情况，它就将控制返回到引起故障的指令，从而重新执行它。否则处理程序返回到内核中的abort例程，abort例程会终止引起故障的应用程序。
4. 终止 终止是不可恢复的致命错误造成的结果，通常是一些硬件错误，比如DRAM或者SRAM位被损坏时发生的奇偶错误。终止处理程序从不将控制返回给应用程序。处理程序将控制返回给一个abort例程，该例程会终止这个应用程序。

hello执行时，还可以发送或接收**信号**。信号是一种系统消息，它用于通知进程系统中发生了某种类型的事件，是一种更高层的软件形式的异常。不同的事件对应不同的信号类型。信号传送到目的进程由发送和接收两个步骤组成。信号的发送者一般是内核，接收者是进程。

发送信号可以有如下两种原因：

1. 内核检测到一个系统事件（如除零错误或者子进程终止）；
2. 一个进程调用了kill函数，显式地要求内核发送一个信号给目的进程。

接收信号是内核强迫目的进程做出的反应。进程可以以默认方式做出反应，也可以通过信号处理程序捕获这个信号。每个信号只会被处理一次。

待处理信号指的是已经发送而没有接收的信号。任何时候，一种信号类型至多有一个待处理信号，即信号不会排队。

进程可以有选择性地阻塞接收某种信号。被阻塞的信号仍可以发出，但不会被目标进程接收。

正常运行如下：



图 6‑3 正常运行

过程中乱按如下：

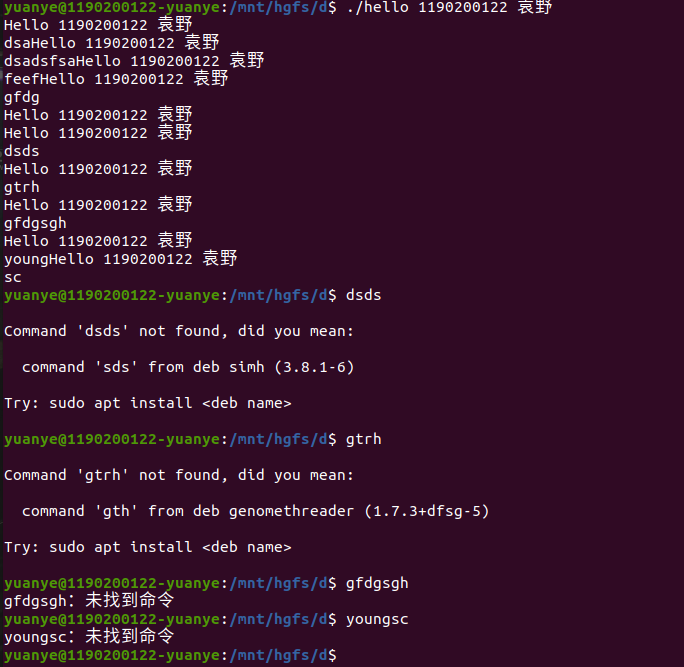


图 6‑4 过程中乱按

如果乱按过程中没有回车，这个时候只是把输入屏幕的字符串缓存起来，如果输入了回车，getchar读入缓存，并把回车后的字符串当作shell输入的命令，然后再程序结束后试图运行这些命令。

过程中输入crtl-c如下：

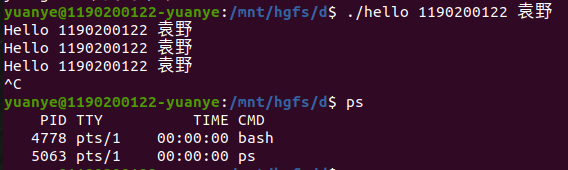


图 6‑5 ctrl-c

Ctrl-C操作向进程发送了一个SIGINT信号，让进程终止，输入ps指令可以发现hello进程已经被回收。

过程中输入crtl-z如下：



图 6‑6 ctrl-z

Ctrl-Z操作向进程发送了一个SIGTSTP信号，让进程暂时挂起，输入jobs、ps指令可以发现hello进程在后台挂起，且进程号为1，通过fg指令可以恢复运行。

## 6.7本章小结

本章描述了hello进程管理，介绍了进程的概念和作用，以及shell如何运行hello程序。为了高效地描述系统中发生的各类事件，则需要用到信号，这是一种更高层级的软件形式的异常。利用信号，内核和进程之间得以高效地传递信息并对各类事件做出相应的反应。

# 第7章 hello的存储管理

## 7.1 hello的存储器地址空间

计算机系统的**主存**被组织成一个由M个连续的字节大小的单元组成的数组。每字节都有唯一的**物理地址**，它是指在地址总线上、以电子形式存在的、使得数据总线可以访问主存的某个特定存储单元的内存地址。利用物理地址寻址，是CPU最自然的访问内存的方式。接下来是几个具体概念：

1. 逻辑地址：在计算机体系结构中逻辑地址是指应用程序角度看到的内存单元、存储单元、网络主机的地址，即hello.o里面的相对偏移地址。逻辑地址往往不同于物理地址，通过地址翻译器或映射函数可以把逻辑地址转化为物理地址。
2. 线性地址：线性地址是逻辑地址到物理地址变换之间的中间层，即hello中的虚拟地址，等于逻辑地址加上基地址。逻辑地址可转化为线性地址，其地址空间是一个非负整数地址的有序集合，如果地址空间中的整数是连续的，那么我们说它是一个线性地址空间。
3. 虚拟地址：虚拟地址是程序用于访问物理内存的逻辑地址，即线性地址，在hello中为虚拟地址。
4. 物理地址：计算机的主存被组织成一个由M个连续的字节大小的单元组成的数组。每个字节都有一个唯一的物理地址。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

内存分段是为了支持多任务并发执行，每一个任务对应各自的段空间，段之间支持保护访问限制，实现了程序和数据从物理地址空间到虚拟地址空间的重映射，从而达到隔离的效果。

如上所述，在段式内存管理中，程序的地址空间被划分为若干段，每个进程都有一个“二维”的地址空间。系统为每个段分配一个连续分区，而进程中的各个段可以不连续地存放在内存的各个分区中。程序加载时，操作系统为所有段分配其所需内存，这些段不必连续，物理内存的管理采用动态分区的管理方法。

为了实现段式管理，系统需要进程段表、系统段表和空闲段表等数据结构，来实现进程的地址空间到物理内存空间的映射，并跟踪物理内存的使用情况，以便在装入新的段的时候，合理地分配内存空间。

1. 程序段表：描述组成程序地址空间的各段，可以是指向系统段表中表项的索引。每段有段基址，即段内地址。
2. 系统段表：系统所有占用段（已经分配的段）。
3. 空闲段表：记忆体中所有空闲段，可以结合到系统段表中。

在段式管理系统中，整个进程的地址空间是“二维”的，逻辑地址由**段号**和段内地址两部分组成。为了完成进程逻辑地址到物理地址的映射，处理器会查找内存中的段表，由段号得到段的首地址，加上段内地址，得到实际的物理地址。这个过程也是由处理器的硬件直接完成的，操作系统只需在进程切换时，将进程段表的首地址装入处理器的特定寄存器当中。这个寄存器一般被称作**段表地址寄存器**。

## 7.3 Hello的线性地址到物理地址的变换-页式管理

虚拟内存系统将程序的虚拟地址空间划分为固定大小的虚拟页，物理内存被划分为同样大小的物理页（也被称作页帧）。在页式存储管理中，虚拟地址由两部构成，高位部分是页号，低位部分是页内地址（偏移量）。

在任意时刻，虚拟页面的集合都分为三个不相交的子集：

1. 未分配页。虚拟内存系统还未分配（或者创建）的页。未分配的块没有任何数据和它们相关联，因此也就不占用任何磁盘空间。
2. 缓存页。当前已缓存在物理内存中的已分配页
3. 未缓存页。未缓存在物理内存中的已分配页

物理内存对应存储器金字塔的DRAM（主存）一级，而虚拟页则是存储在磁盘上。与读写高速缓存一样，从虚拟内存和主存之间也存在着缓存关系，因而也拥有类似的命中、不命中的概念。

页式管理方式的优点是：没有外部碎片；一个程序不必连续存放；便于改变程序占用空间的大小（主要指随着程序执行，动态生成的数据增多，所要求的地址空间相应增长）。其缺点是：要求程序全部装入内存，没有足够的内存，程序就不能执行。

在页式系统中程序建立时，系统为程序中所有的页分配页帧。当程序撤销时收回所有分配给它的页帧。在程序的执行期间，如果允许程序动态地申请空间，系统还要为程序申请的空间分配物理页帧。系统为了完成这些功能，必须记录系统内存中实际的页帧使用情况。系统还要在程序切换时，需要正确地切换两个不同的程序地址空间到物理内存空间的对映。这就要求系统要记录每个程序页表的相关信息。为了完成上述的功能，—个页式系统中，一般要采用如下的数据结构：

页表：页表将虚拟内存对映到物理页。每次内存管理单元将一个虚拟地址转换为物理地址时，都会读取页表。页表是一个页表条目（PTE）的数组。虚拟地址空间的每个页在页表中一个固定偏移量处都有一个PTE。假设每个PTE是由一个有效位和一个n位地址栏位组成的。有效位表明了该虚拟页当前是否被缓存在DRAM中。如果设定了有效位，那么地址字段就表示DRAM中相应的物理页的起始位置，这个物理页中缓存了该虚拟页。如果没有设定有效位，那么一个空地址表示这个虚拟页还未被分配。否则，这个地址就指向该虚拟页在磁盘上的起始位置。

MMU利用虚拟页号（VPN）来选择适当的PTE，将列表条目中物理页号（PPN）和虚拟地址中的虚拟页偏移量（VPO）串联起来，就得到相应的物理地址。

## 7.4 TLB与四级页表支持下的VA到PA的变换

在Intel Core i7环境下研究VA到PA的地址翻译问题。前提如下：

虚拟地址空间48位，物理地址空间52位，页表大小4KB，4级页表。TLB 4路16组相联。CR3指向第一级页表的起始位置（上下文一部分）。

解析前提条件：由一个页表大小4KB，一个PTE条目8B，共512个条目，使用9位二进制索引，一共4个页表共使用36位二进制索引，所以VPN共36位，因为VA 48位，所以VPO 12位；因为TLB共16组，所以TLBI需4位，因为VPN 36位，所以TLBT 32位。

CPU产生虚拟地址VA，VA传送给MMU，MMU使用前36位VPN作为TLBT（前32位）+TLBI（后4位）向TLB中匹配，如果命中，则得到PPN（40bit）与VPO（12bit）组合成PA（52bit）。

如果TLB中没有命中，MMU向页表中查询，CR3确定第一级页表的起始地址，VPN1（9bit）确定在第一级页表中的偏移量，查询出PTE，如果在物理内存中且权限符合，确定第二级页表的起始地址，以此类推，最终在第四级页表中查询到PPN，与VPO组合成PA，并且向TLB中添加条目。

如果查询PTE的时候发现不在物理内存中，则引发缺页故障。如果发现权限不够，则引发段错误。

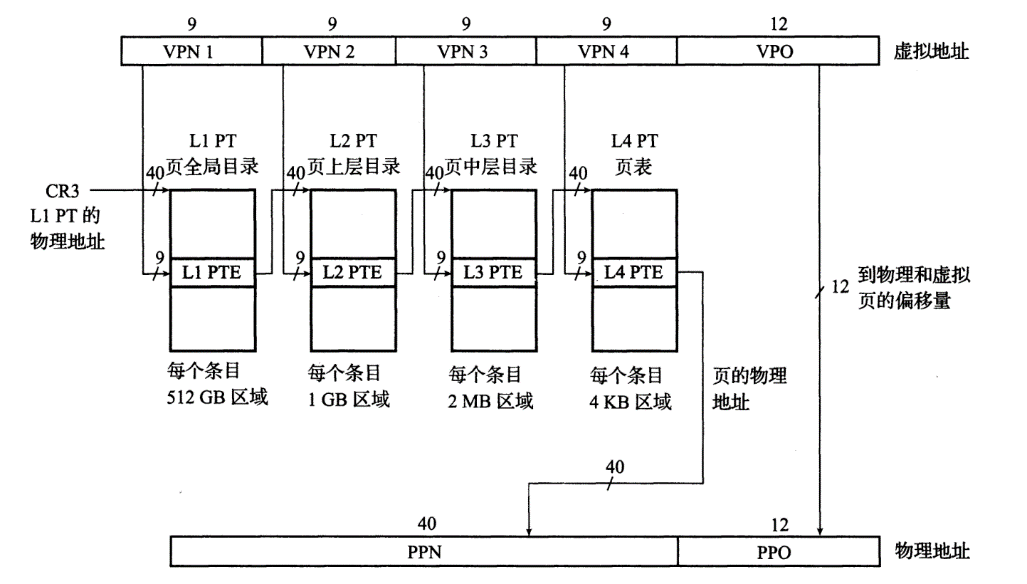


图 7‑1 四级页表下的VA到PA的变换

## 7.5 三级Cache支持下的物理内存访问

前提：只讨论L1 Cache的寻址细节，L2与L3Cache原理相同。L1 Cache是8路64组相联。块大小为64B。

解析前提条件：因为共64组，所以需要6bit CI进行组寻址，因为共有8路，因为块大小为64B所以需要6bit CO表示数据偏移位置，因为VA共52bit，所以CT共40bit。

在上一步中我们已经获得了物理地址VA，使用CI（后六位再后六位）进行组索引，每组8路，对8路的块分别匹配CT（前40位）如果匹配成功且块的valid标志位为1，则命中（hit），根据数据偏移量CO（后六位）取出数据返回。

如果没有匹配成功或者匹配成功但是标志位是1，则不命中（miss），向下一级缓存中查询数据（L2 Cache->L3 Cache->主存）。查询到数据之后，一种简单的放置策略如下：如果映射到的组内有空闲块，则直接放置，否则组内都是有效块，产生冲突（evict），则采用最近最少使用策略LFU进行替换。

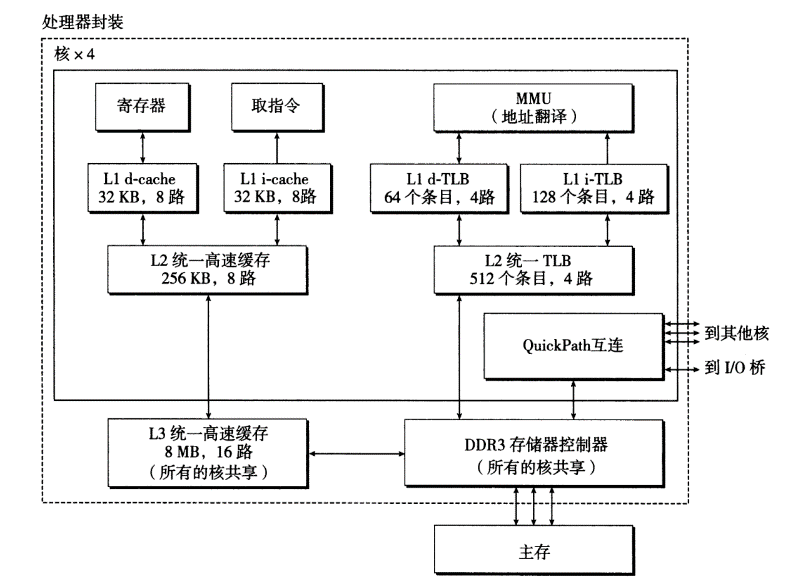


图 7‑2 三级Cache支持下的物理内存访问

## 7.6 hello进程fork时的内存映射

当fork函数被shell调用时，内核为hello创建各种数据结构，并分配一个唯一的PID。为了给hello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。它将两个进程中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hello进程中返回时，hello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的页面，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

execve函数调用驻留在内核区域的启动加载器代码，在当前进程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。

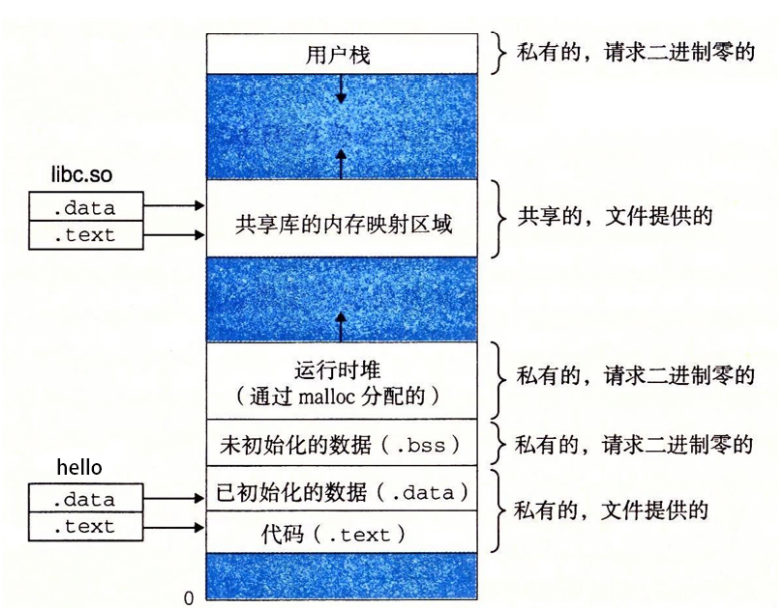


图 7‑3 加载器映射到用户地址空间区域

## 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。其处理流程遵循图7-4所示的故障处理流程。

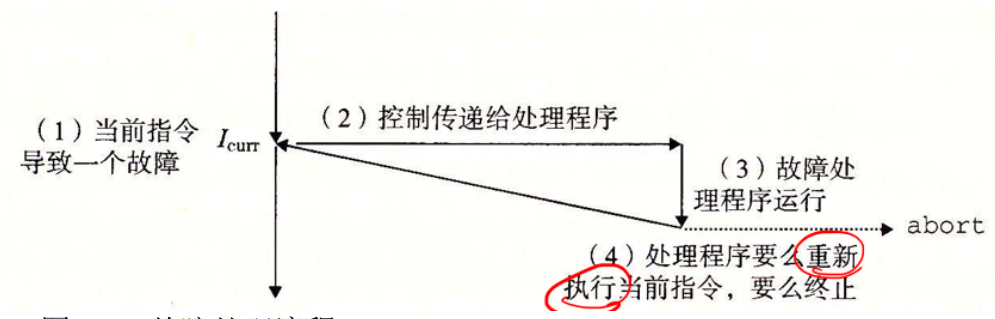


图 7‑4 故障处理流程

## 7.9动态存储分配管理

动态内存分配器用于分配和维护一个进程的虚拟内存区域，称为堆。堆在系统内存中向上生长。对于每个进程，系统维护着一个堆顶指针brk。

分配器将堆视为不同大小的块组成的集合。每个块就是一段连续的虚拟内存片（chunk），要么是已分配的（allocated），要么是空闲的（free）。已分配的块显式地保留给应用程序使用，空闲块留待分配。

**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。**

### **7.9.1 隐式空闲链表**

我们可以将堆组织为一个连续的已分配块和空闲块的序列，空闲块是通过头部中的大小字段隐含地连接着的，这种结构为隐式空闲表。分配器可以通过遍历堆中所有的块，从而间接地遍历整个空闲块地集合。一个块是由一个字的头部、有效载荷、可能的填充和一个字的脚部，其中脚部就是头部的一个副本。头部编码了这个块的大小以及这个块是已分配还是空闲的。分配器就可以通过检查它的头部和脚部，判断前后块的起始位置和状态。

### **7.9.2 显式空闲链表**

将堆组成一个双向空闲链表，在每个空闲块中，都包含一个pred和succ指针。

一种方法是用后进先出（LIFO）的顺序来维护链表，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，释放一个块可以在常数时间内完成。如果使用了边界标记，那么合并也可以在常数时间内完成。

另一种方法是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。

一般而言，显式链表的缺点是空闲块必须足够大，以包含所有需要的指针，以及头部和可能的脚部。这就导致了更大的最小快大小，也潜在的提高了内部碎片的程度。

## 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，以intel Core7在指定环境下介绍了VA到PA的变换、物理内存访问，还介绍了hello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8章 hello的IO管理

## 8.1 Linux的IO设备管理方法

所有的I/O装置（例如网络、磁盘和终端）都被模型化为文件，而所有的输入和输出都被当做对相应档案的读和写来执行。这种将装置优雅地映射为文件的方式，允许Linux核心引出一个简单、低阶的应用接口，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函数实现如下：

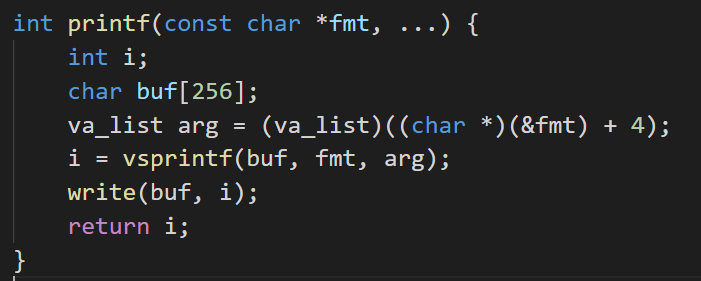


图 8‑1 printf的实现

用vsprintf函数生成显示信息，实现如下：

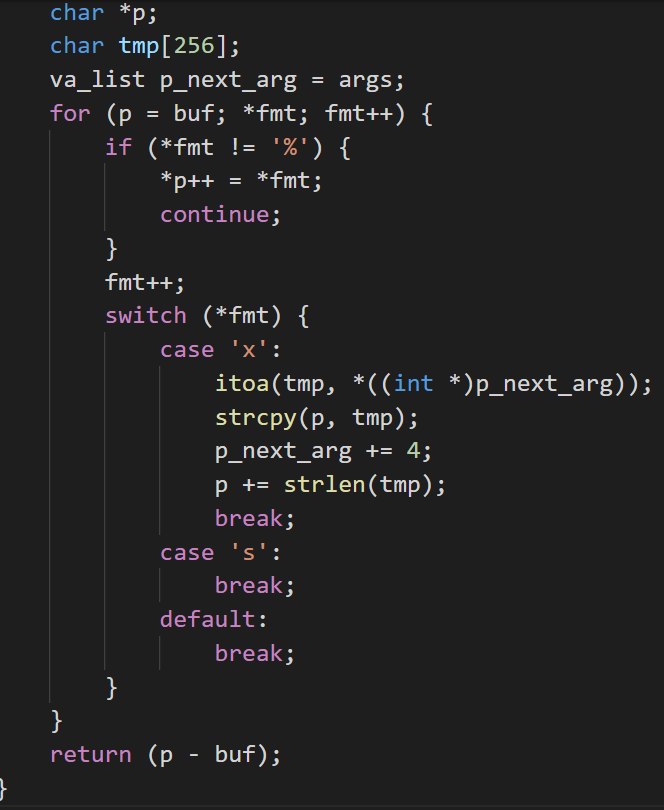


图 8‑2 vsprintf实现

在printf中调用write(buf, i)，输出长度为*i*的buf。追踪write，得到其汇编实现如下：

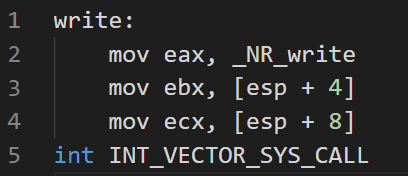


图 8‑3 write汇编

write函数中，%ecx中存储字符个数，%ebx中存储字符串首地址，int INT\_VECTOR\_SYS\_CALL的意思是通过系统调用 sys\_call。这个函数的功能就是不断地打印出字符，直到遇到\0。

追踪sys\_call，得到其汇编实现如下：

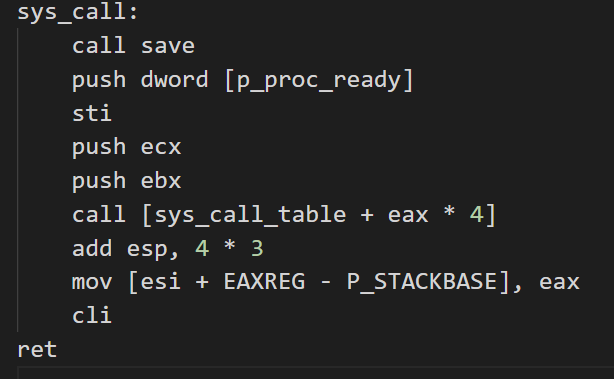


图 8‑4 sys\_call汇编

接着，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）。

## 8.4 getchar的实现分析

getchar函数的声明在stdio.h头文件中。一种实现如下：

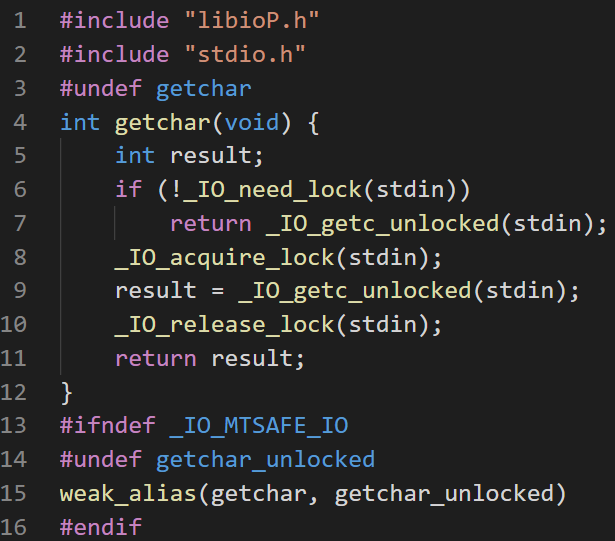


图 8‑5 getchar的实现

这个getchar每次从标准输入中读取一个字符。具体来说，若当前I/O未被锁定，它就调用系统\_IO\_getc\_unlocked内置宏，读取一个字符。

把\_IO\_getc\_unlocked写成等价的函数形式如下：

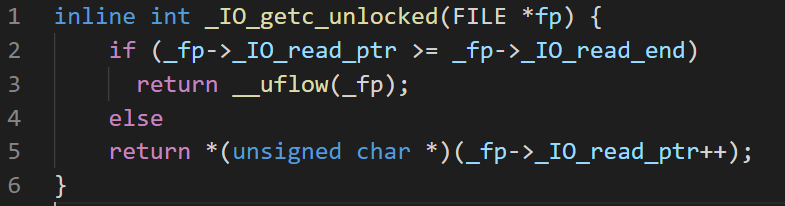


图 8‑6 IO\_getc\_unlocked写成等价的函数形式

简单地说，我们用一个指针\_IO\_read\_ptr指向缓冲区，用另一个指针\_IO\_read\_end指向缓冲区的末尾。调用\_IO\_getc\_unlocked时，先检查指针是否越界。如果没有，就返回\_IO\_read\_ptr所指向的字符并自增\_IO\_read\_ptr。若已越界，就调用\_uflow（内部使用了系统read），用这个函数重新填充缓冲区并返回重新读入的字符。

对于异步异常和键盘中断的处理：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。也就是说，getchar等调用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

本章主要介绍了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。

结 论

hello的一生包含如下阶段：

1. 预处理：将hello.c根据以字符#开头命令，修改原始c程序，得到hello.i。
2. 编译：将hello.i翻译为hello.s的汇编程序，中间对代码进行语法检查和优化。
3. 汇编：将hello.s翻译为二进制机器码，得到可重定位目标文件hello.o。
4. 链接：将hello.o同等动态库等连接，生成可执行目标文件hello。
5. 创建进程：通过shell运行hello程序。shell通过fork创建子进程，通过execve运行hello。
6. 访问内存：通过MMU将hello中的虚拟地址转换为实际的物理地址，再通过多级缓存读取数据。
7. 异常：程序执行过程中，如果从键盘输入Ctrl-C等命令，会给进程发送一个异常信号，然后通过信号处理函数对信号进行处理。
8. 结束：hello运行完后会由父进程（shell进程）回收，内核会删除对应的数据结构。

通过一个学期的CSAPP学习，我了解了一个代码从产生到实现的每一个过程以及整个过程中经历的各种细节。曾经的我只知道数组开太大会编译错误，int值太大会变成负数，但从来没由了解过为什么以及怎么样，如今了解了一台电脑底层的活动之后，尤其是储存器结构，虚拟内存，机器级表示的相关知识之后对程序编写有了更加深入的认识。

# 

# 附件

|  |  |
| --- | --- |
| 文件名 | 作用 |
| hello.c | 源代码 |
| hello.i | hello.c预处理生成的文本文件 |
| hello.s | hello.i编译后得到的汇编语言文本文件 |
| hello.o | hello.s汇编后得到的可重定位目标文件 |
| hello | hello.o链接后得到的汇编语言文本文件 |
| helloo.objdump | hello.o的反汇编结果 |
| helloo.elf | hello.o的ELF结构 |
| hello.d | hello的反汇编结果 |
| hello.elf | hello的ELF结构 |

# 参考文献

1. 兰德尔·E·布莱恩特等著；深入理解计算机系统[M]. 北京：机械工业出版社，2016.7.
2. C library - C++ Reference <http://www.cplusplus.com/reference/clibrary/>.
3. ArchWiki <https://wiki.archlinux.org/>.
4. Linux man pages <https://linux.die.net/man/>.
5. printf 函数实现的深入剖析 <https://www.cnblogs.com/pianist/p/3315801.html>
6. ELF 构造：<https://www.cs.stevens.edu/~jschauma/631/elf.html>
7. 进程的睡眠、挂起和阻塞：<https://www.zhihu.com/question/42962803>
8. Wikipedia <https://www.wikipedia.org/>.
9. jaywcjlove/linux-command: Linux命令大全搜索工具...–GitHub <https://github.com/jaywcjlove/linux-command>