

**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 1180300308

班　　 级 03003

学 生 刘义

指 导 教 师 史先俊

**计算机科学与技术学院**

**2019年12月**

**摘 要**

本论文旨在研究 hello 在 linux 系统下的整个生命周期：以hello.c源程序为起点，从编译、链接，到加载、运行，再到终止、回收。结合CSAPP课本，通过gcc等工具进行实际操作，顺着hello的生命周期，漫游整个计算机系统。

**关键词：**hello；生命周期；CSAPP；

**（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

P2P(From Program to Process)：在linux中，hello.c经过cpp预处理、ccl编译、as 汇编、ld 链接最终成为可执行目标程序 hello，在 shell 中键入执行命令后，shell为其fork，产生子进程，于是 hello 便从Program摇身一变成为 Process，这便是P2P的过程。

020(From Zero-0 to Zero-0)：之后shell为其execve，映射虚拟内存，进入程序入口后程序开始载入物理内存，然后进入 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。当程序运行结束后，shell 父进程负责回收hello进程，内核删除相关数据结构，以上全部便是020的过程。

## 1.2 环境与工具

1.2.1 硬件环境

Intel(R) Core(TM) i7-8550U X64CPU\_@\_1.80GHz

2GHz;

16G RAM;

256G SSD + 256G SSD

1.2.2 软件环境

Windows10 64位；

Vmware 14；

Ubuntu 18.04 64位；

1.2.3开发和调试工具

vim，gedit，gcc，as，ld，readelf，objdump，edb

## 1.3 中间结果

列出你为编写本论文，生成的中间结果文件的名字，文件的作用等。

|  |  |
| --- | --- |
| 文件名称 | 文件作用 |
| hello.c | hello程序c语言源文件 |
| hello.i | hello.c预处理生成的文本文件 |
| hello.s | hello.i编译得到的汇编文件 |
| hello.o | hello.s汇编得到的可重定位目标文件 |
| hello | hello.o与动态链接库链接得到的可执行目标文件 |
| hello.o.objdmp | 使用objdump反汇编hello.o得到的反汇编文件 |
| hello.objdmp | 使用objdump反汇编hello得到的反汇编文件 |
| hello.o.elf | hello.o的ELF格式 |
| hello.elf | hello的ELF格式 |

## 1.4 本章小结

// 我是Hello，我是每一个程序猿¤的初恋（羞羞……）

// 却在短短几分钟后惨遭每个菜鸟的无情抛弃（呜呜……），他们很快喜欢上sum、sort、matrix、PR、AI、IOT、BD、MIS……，从不回头。

// 只有我自己知道，我的出身有多么高贵，我的一生多么坎坷！

// 多年后，那些真懂我的大佬（也是曾经的菜鸟一枚），才恍然感悟我的伟大！

// 想当年： 俺才是第一个玩 P2P的: From Program to Process

// 懵懵懂懂的你笨笨磕磕的将我一字一键敲进电脑存成hello.c（Program），无意识中将我预处理、编译、汇编、链接，历经艰辛，我-Hello一个完美的生命诞生了。

// 你知道吗？在壳(Bash)里，伟大的OS（进程管理）为我fork（Process）,为我execve,为我mmap，分我时间片，让我得以在Hardware(CPU/RAM/IO)上驰骋（取指译码执行/流水线等）；

// 你知道吗？OS（存储管理）与MMU为VA到PA操碎了心；TLB、4级页表、3级Cache，Pagefile等等各显神通为我加速；IO管理与信号处理使尽了浑身解数，软硬结合，才使我能在键盘、主板、显卡、屏幕间游刃有余, 虽然我在台上的表演只是一瞬间、演技看起来还很Low、效果很惨白。

// 感谢 OS！感谢 Bash！在我完美谢幕后回收了我。 我赤条条来去无牵挂！

// 我朝 CS（计算机系统-Editor+Cpp+Compiler+AS+LD + OS + CPU/RAM/IO等）挥一挥手，不带走一片云彩！ 俺也是 O2O: From Zero-0 to Zero-0。

// 历史长河中一个个菜鸟与我擦肩而过，只有CS知道我的生、我的死，我的坎坷，“只有 CS 知道……我曾经……来…………过……”————未来一首关于Hello的歌曲绕梁千日不绝 !!

**（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，修改原始的C程序，得到另一个c程序，通常以.i作为文件扩展名。

**作用**：

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<stdio.h> 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. 以实际字符代替宏定义
3. 根据条件编译确定需要编译的代码

## 2.2在Ubuntu下预处理的命令

命令：gcc -E hello.c -o hello.i

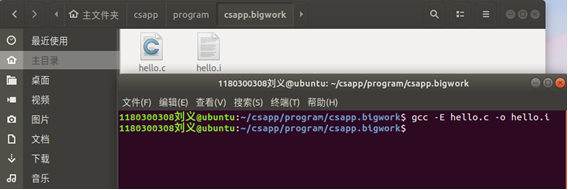


图 2.1 使用 gcc -E命令生成 hello.i 文件

## 2.3 Hello的预处理结果解析

gedit打开hello.i文件可以发现，整个hello.i程序已经拓展为3113行，hello.c中main函数出现在3100行：

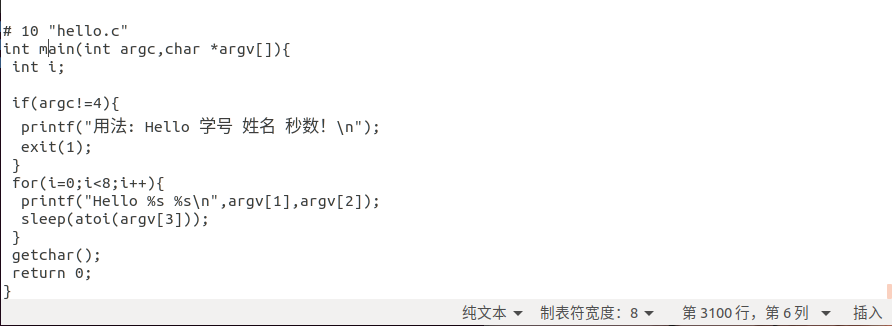


图 2.2 hello.i 中 main 函数的位置

在这之前出现的是 stdio.h unistd.h stdlib.h 的依次展开，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，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编译相应代码。unistd.h、stdlib.h的展开的展开与之类似。

## 2.4 本章小结

hello.c在正式编译前，先要经过预处理，预处理过程只是简单的文本插入过程，生成的hello.i文件仍然是文本文件。

本章主要介绍了预处理的定义与作用、并结合预处理之后的程序对预处理结果进行了解析。

**（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(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。这个过程被称为编译。

**作用**：将c语言程序翻译为汇编语言程序。

## 3.2 在Ubuntu下编译的命令

命令：gcc -m64 -no-pie -fno-PIC -S hello.i -o hello.s

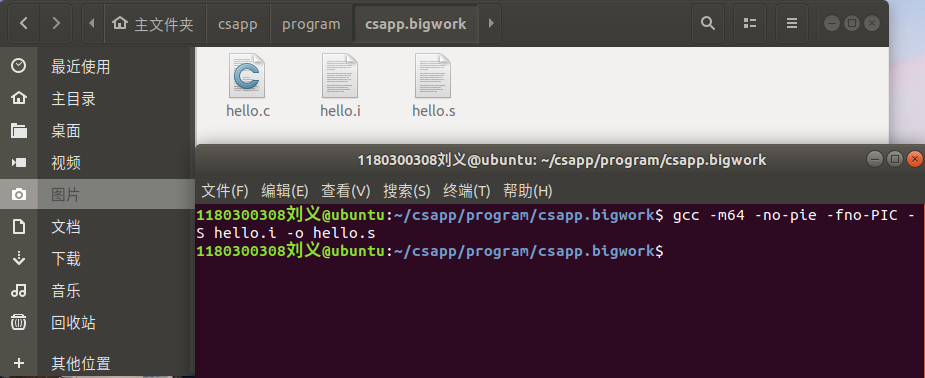


图 3.1 使用 gcc -S命令生成hello.s文件

## 3.3 Hello的编译结果解析

### 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、整数

1. int i：局部变量存储在寄存器或者栈上，在 hello.s 中编译器将 i存储在栈上-4(%rbp)。



图 3.2 i初始化为0

1. if(argc!=4)语句中的变量argc与常量4：

argc：变量，main的参数之一，x64编译下由寄存器传递，然后保存在栈上。

4：立即数，在汇编语句中出现。

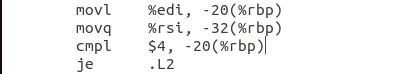


图 3.3 argc的保存、与立即数4的比较

1. 其他立即数（如8）：同4，在汇编语句中出现。

二：字符串

程序中的字符串有：

1. "用法: Hello 学号 姓名 秒数！\n"
2. "Hello %s %s\n"

这两个字符串是常量，汉字被转换为IEEE754编码，存储在.rodata只读数据节中。

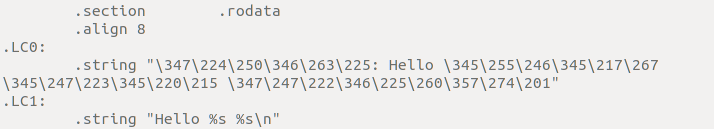


图 3.4 hello.s中声明在.LC0 和.LC1 段中的字符串

三：数组

程序中数组：argv

argv是main函数参数char \*argv[]，进入main函数后将其起始地址存放在栈上：

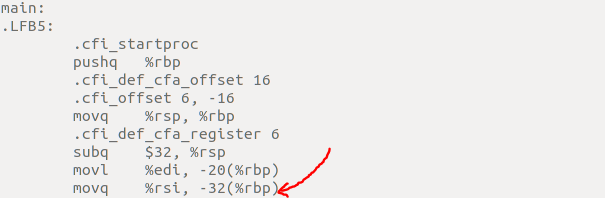


图 3.5 argv首地址存储在栈上

### 3.3.2 赋值

程序中的赋值操作为：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：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 指令后缀 | b | w | l | q |
| 数据大小 | 8b | 16b | 32b | 64b |

int型数据32个字节，使用movl指令传送：



图 3.6 i赋值为0

### 3.3.3 算术操作

汇编语言中，算术指令操作如下：

|  |  |
| --- | --- |
| 指令 | 效果 |
| leaq S,D | D=&S(地址传递) |
| 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]=S\*R[%rax]（有符号） |
| MULQ S | R[%rdx]:R[%rax]=S\*R[%rax] （无符号） |
| IDIVQ S | R[%rdx]=R[%rdx]:R[%rax] mod S  R[%rax]=R[%rdx]:R[%rax] div S  （有符号） |
| DIVQ S | R[%rdx]=R[%rdx]:R[%rax] mod S  R[%rax]=R[%rdx]:R[%rax] div S  （无符号） |

程序中涉及的算术操作有：

1）i++，对计数器i自增，i是32位int，使用程序指令addl 。

### 3.3.4 关系操作

进行关系操作的汇编指令有：

|  |  |
| --- | --- |
| 指令 | 效果 |
| 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有：

1. argc!=4，使用指令cmp进行比较，设置条件码，然后分别进行跳转：

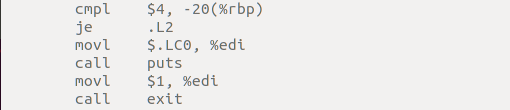


图3.7 argc与4进行比较

1. i<8，等价于i<=7，同样使用指令cmp进行比较：

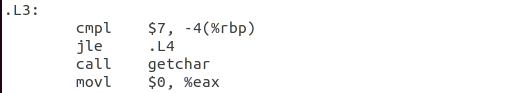


图3.8 i与7进行比较

### 3.3.6 数组操作

程序中唯一的数组为argv，argv[]中存储着各个字符串的首地址（64位地址，8B），访问数组元素 argv[1],argv[2]时，按照起始地址+（下标\*8B）计算数据地址，r然后取数据：

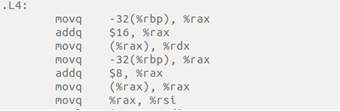


图 3.9 计算地址取出数组值

### 3.3.7 控制转移

程序中涉及的控制转移有：

1. if (argv!=4)：首先使用cmpl指令比较argv与4，如果argv等于4，跳转到.L2执行；否则（不相等）顺序执行下一条指令，即if中的语句。
2. for(i=0;i<8;i++)：首先无条件跳转到位于循环体.L4 之后的比较代码，使用cmpl指令比较i与7，如果 i<=7，跳转回.L4执行；否则循环结束、顺序执行下一条指令。

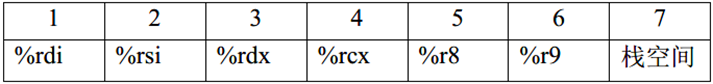
### 3.3.8 函数操作

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：准备进行过程Q的时候，先将程序计数器（PC）的原值压入栈中，以便之后能够返回。同时PC设置为Q的代码的起始地址，然后在返回时， Q后面那条指令的地址出栈，控制回到P过程。

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。64位编译系统采用寄存器传参数。一般使用%rax保存返回值。

64位程序的传参次序(浮点数使用xmm，不包含)：



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：

传递控制：main 函数被系统启动函数\_\_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。

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。

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，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。

1. printf 函数：

传递数据：第一次printf 将%rdi 设置为“Usage: Hello 学号，姓名！”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]。

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

1. exit 函数：

传递数据：将%edi 设置为1。

控制传递：call exit。

1. sleep 函数：

传递数据：将%edi 设置为%eax(atoi的返回值)。

传递控制：call sleep。

1. getchar 函数：

传递控制：call getchar。

## 3.4 本章小结

本章主要阐述了编译器是如何处理各个数据类型以及各类操作，并结合所给的hello.c程序，生成并查看了hello.s代码，验证了数据、操作等在汇编代码中的实现。

编译器将 hello C语言程序翻译为更加低层的汇编语言程序。

**（第3章2分）**

# 第4章 汇编

## 4.1 汇编的概念与作用

概念：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、数据等。

作用：把汇编语言程序翻译为二进制机器语言程序

注意：这儿的汇编是指从 .s 到 .o 即编译后的文件到生成机器语言二进制程序的过程。

## 4.2 在Ubuntu下汇编的命令

命令：gcc -m64 -no-pie -fno-PIC -c hello.s -o hello.o

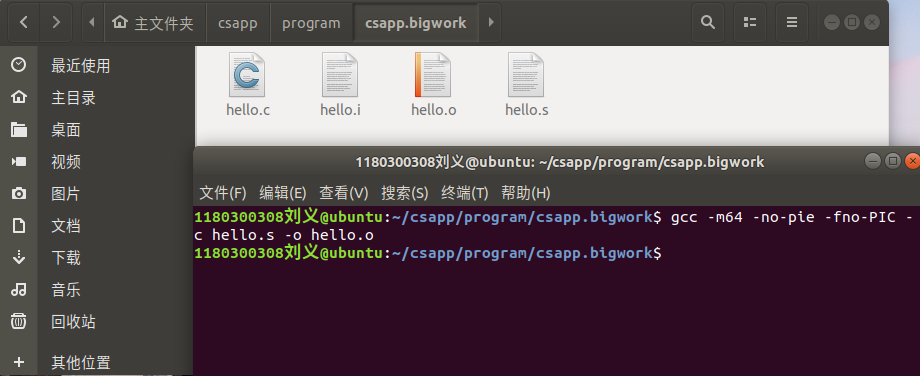


图4.1 使用 gcc -c命令生成hello.文件

## 4.3 可重定位目标elf格式

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

使用readelf -a hello.o > hello.o.elf命令，获得 hello.o 文件的 ELF 格式。其组成如下：

1. ELF头：描述了生成该文件的系统的字大小、字节顺序（大/小）、ELF头的大小、目标文件的类型、机器类型（如x86-64）、节头部表的偏移、节头部表中条目的大小和数量。

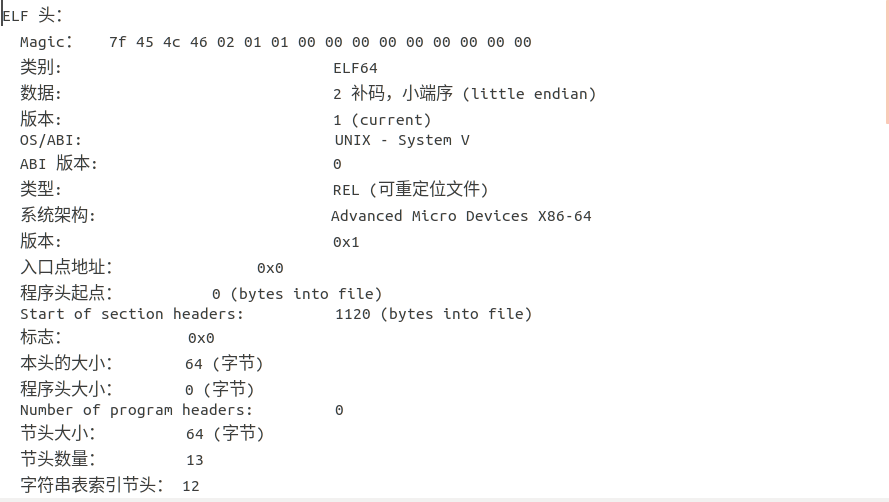


图4.2 ELF头

1. 节头：记录每个节的名称、偏移量、大小、位置等信息

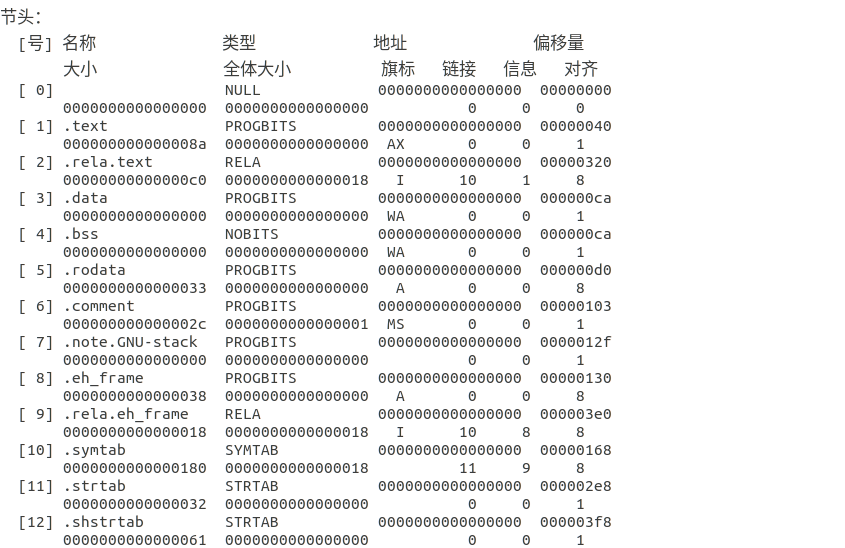


图4.3 节头

1. 重定位节.rela.text：包含.text 节的重定位条目，当链接器把这个目标文件和其他文件链接时，需要修改.text节这些位置的内容。

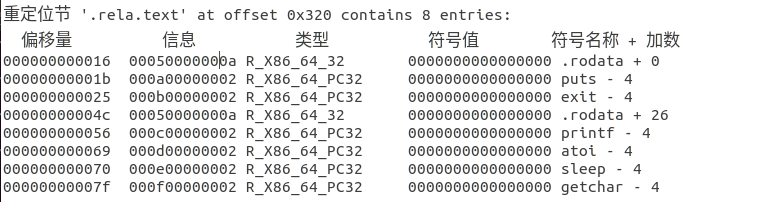


图4.4 重定位节

重定位条目的结构体成员如下：

|  |  |
| --- | --- |
| offset | 需要进行重定向的代码在.text或.data 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 | 重定位时用于偏移调整的辅助信息 |
| Type | 重定位到的目标的类型 |
| Name | 重定向到的目标的名称 |

以第一个条目为例演示重定位过程：

重定位类型：PC相对寻址。

计算重定位目标地址的算法如下（设需要重定位的.text 节中的位置为src,设重定位的目的位置dst）：

refptr = s +r.offset （1）

refaddr = ADDR(s) + r.offset （2）

\*refptr = (unsigned) (ADDR(r.symbol) + r.addend-refaddr)（3）

其中（1）指向src 的指针（2）计算src 的运行时地址，（3）中，ADDR(r.symbol)计算dst 的运行时地址，在本例中，ADDR(r.symbol)获得的是dst 的运行时地址，因为需要设置的是相对地址，即dst 与下一条指令之间的地址之差，所以需要设置“-4”的偏移调整。

1. 重定位节.rela.eh\_frame：eh\_frame 节的重定位信息。

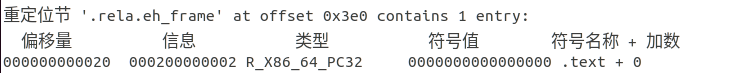


图4.5 重定位节

1. 符号表.symtab：用来存放程序中定义和引用的函数和全局变量的信息。重定位需要引用的符号都在其中声明。

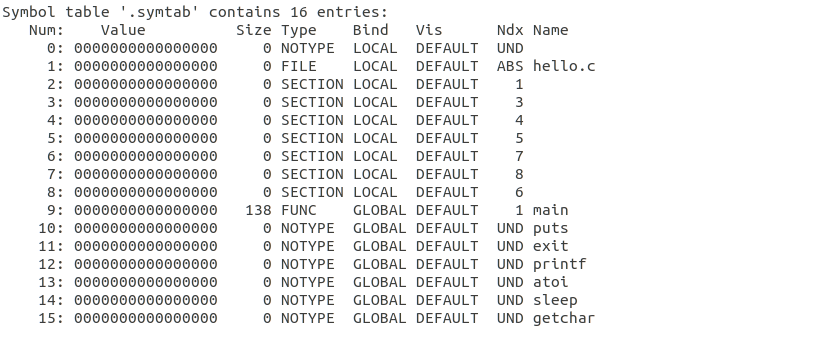


图4.6 符号表

## 4.4 Hello.o的结果解析

使用 objdump -d -r hello.o > hello.o.objdump 获得反汇编文件hello.o.objdump。与hello.s相比区别如下：

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，在汇编成机器语言时被转换成确定的地址。

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 的目标地址是下一条指令的地址。程序中call调用的函数是共享库的函数，需要进行重定位以确定函数所在地址。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（正是下一条指令），然后在.rela.text 节中为其添加重定位条目，等待链接的进一步确定。

3）字符串常量访问：传递第一个printf中的字符串（在rodata中）地址时，.s文件为：movl $.LC0, %edi；反汇编文件为：mov $0x0,%edi，之后有一个重定位标识：16: R\_X86\_64\_32 .rodata。这是因为rodata 中数据地址也是在运行时 才能确定，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。

机器语言的构成：

1. 操作码。它具体说明了操作的性质及功能。一台计算机可能有几十条至几百条指令，每一条指令都有一个相应的操作码，计算机通过识别该操作码来完成不同的操作。
2. 操作数的地址。CPU通过该地址就可以取得所需的操作数。
3. 操作结果的存储地址。把对操作数的处理所产生的结果保存在该地址中，以便再次使用。

机器语言与汇编语言的映射关系

从机器指令到汇编：指令码代之以记忆符号，地址码代之以符号地址，使得其含义显现在符号上而不再隐藏在编码中。

## 4.5 本章小结

本章介绍了hello 从hello.s 到hello.o 的汇编过程，查看了hello.o 的elf 格式、使用objdump 得到反汇编代码与hello.s 进行了比较，了解了从汇编语言映射到机器语言需要实现的转换。在这个过程中，生成了重定位条目，为之后的链接中的重定位过程奠定了基础。

**（第4章1分）**

# 第5章 链接

## 5.1 链接的概念与作用

概念：链接是将各种代码和数据片段收集并组合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。在现代系统中，链接是由叫做链接器的程序自动执行的。

作用：程序模块化，使分离编译成为可能。

## 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

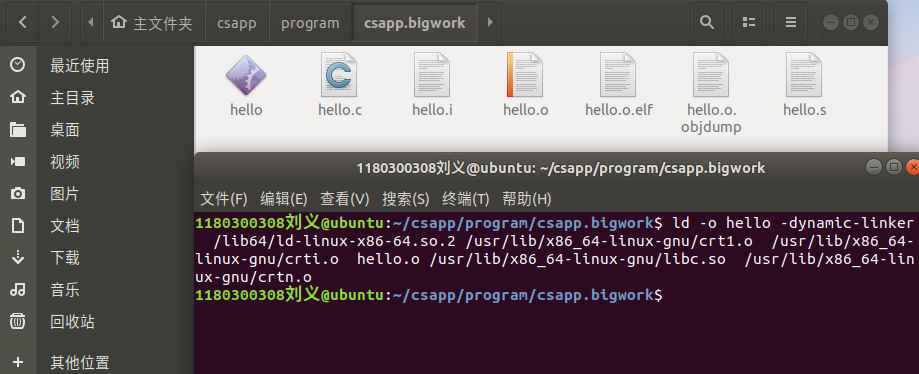


图5.1 使用ld命令生成hello可执行文件

## 5.3 可执行目标文件hello的格式

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

1. ELF头：描述文件的总体格式。还包括程序的入口点。

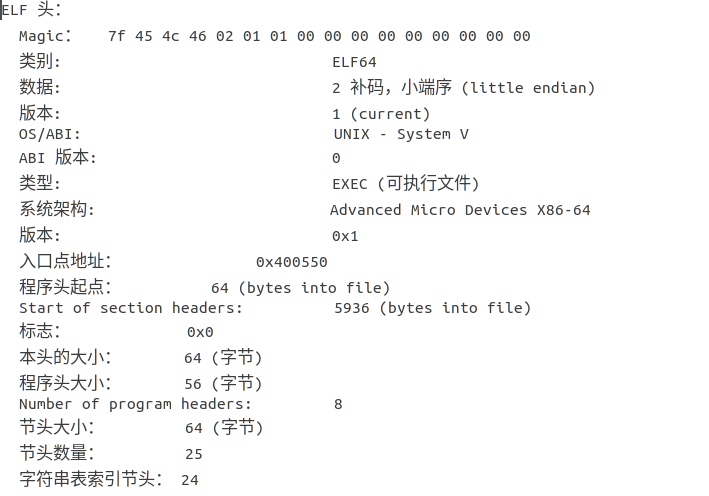


图5.2 ELF头

1. 节头：记录每个节的名称、偏移量、大小、位置等信息。

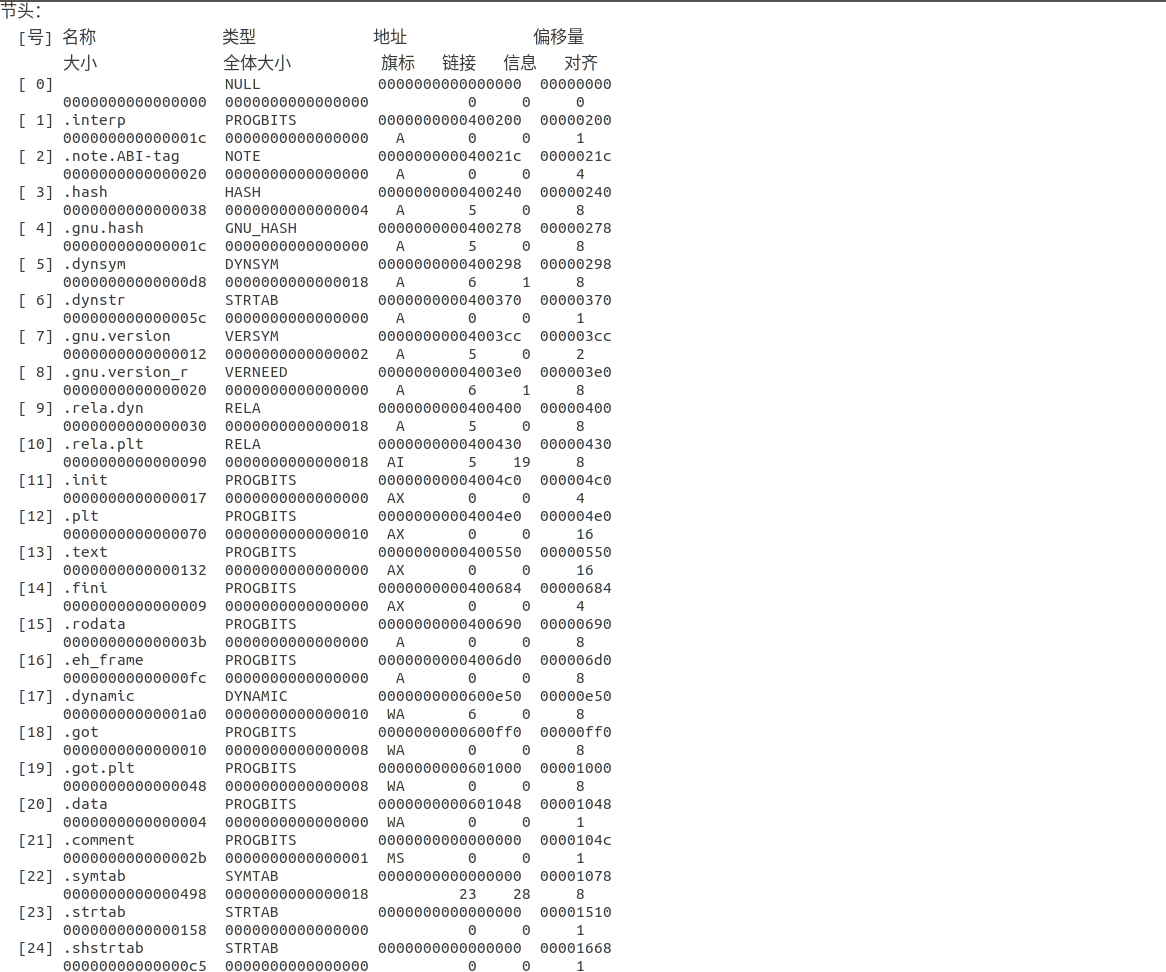


图5.3 节头

节头对 hello 中所有的节信息进行了声明，其中包括大小 Size 以及在程序中的偏移量 Offset，因此根据节头中的信息我们就可以定位各个节所占的区间（起始位置，大小）。

## 5.4 hello的虚拟地址空间

用edb查看程序hello，发现程序在地址0x400000~0x401000中被载入，每个节排列都同5.3中的地址声明，在0~fff空间中，与0x400000~0x401000段的存放的程序相同，在fff之后存放的是.dynamic~.shstrtab节。

查看ELF文件的程序头，程序头在为链接器提供运行时的加载内容和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大小和物理地址空间大小、位置、标志、访问权限和对齐方式，红框表示虚拟地址空间信息。

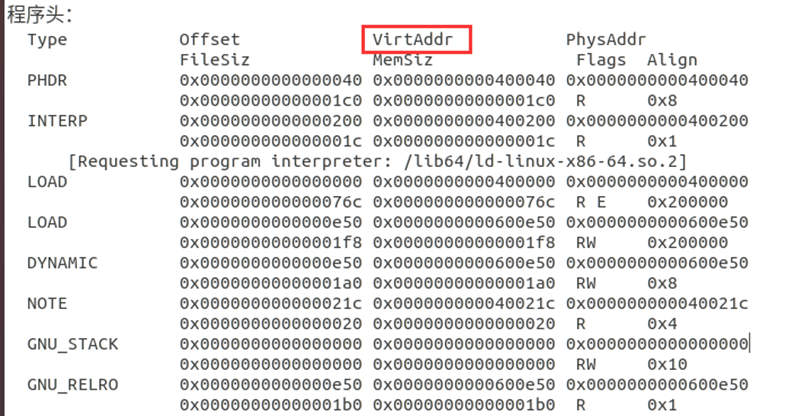


图5.4 程序头

从下表可以看出，程序包含 8 个段（其中两个LOAD）：

|  |  |
| --- | --- |
| 段名称 | 功能 |
| **PHDR** | 保存程序头表 |
| **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） |
| **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。 |
| **DYNAMIC** | 保存了由动态链接器使用的信息 |
| **NOTE** | 保存辅助信息 |
| **GNU\_STACK** | 权限标志，标志栈是否是可执行的 |
| **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

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。

使用 objdump -d -r hello > hello.objdump 获得 hello 的反汇编文件。与之前的hello.o的反汇编文件对比，有以下几点的不同：

1. hello.objdump多一些节，见下表：

|  |  |
| --- | --- |
| 节名称 | 描述 |
| \_init | 程序初始化需要执行的代码 |
| .plt | 动态链接-过程链接表 |
| puts@plt | puts对应的plt表条目 |
| printf@plt | printf对应的plt表条目 |
| getchar@plt | getchar对应的plt表条目 |
| atoi@plt | atoi对应的plt表条目 |
| exit@plt | exit对应的plt表条目 |
| sleep@plt | sleep对应的plt表条目 |
| \_fini | 当程序正常终止时需要执行的代码 |

1. 之前未重定位的全局变量引用、过程调用、控制转移全部定位了。

链接过程：

1. 链接函数：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. 函数重定位：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

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。

|  |  |
| --- | --- |
| 程序名称 | 程序地址 |
| ld-2.27.so!\_dl\_start | 0x7fce 8cc38ea0 |
| ld-2.27.so!\_dl\_init | 0x7fce 8cc47630 |
| hello!\_start | 0x400500 |
| libc-2.27.so!\_\_libc\_start\_main | 0x7fce 8c867ab0 |
| -libc-2.27.so!\_\_cxa\_atexit | 0x7fce 8c889430 |
| -libc-2.27.so!\_\_libc\_csu\_init | 0x4005c0 |
| hello!\_init | 0x400488 |
| libc-2.27.so!\_setjmp | 0x7fce 8c884c10 |
| -libc-2.27.so!\_sigsetjmp | 0x7fce 8c884b70 |
| --libc-2.27.so!\_\_sigjmp\_save | 0x7fce 8c884bd0 |
| hello!main | 0x400532 |
| hello!puts@plt | 0x4004b0 |
| hello!exit@plt | 0x4004e0 |
| ld-2.27.so!\_dl\_runtime\_resolve\_xsave | 0x7fce 8cc4e680 |
| -ld-2.27.so!\_dl\_fixup | 0x7fce 8cc46df0 |
| --ld-2.27.so!\_dl\_lookup\_symbol\_x | 0x7fce 8cc420b0 |
| libc-2.27.so!exit | 0x7fce 8c889128 |

## 5.7 Hello的动态链接分析

对于动态共享链接库中的函数，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，等待动态链接器重定位，为避免不必要的重定位，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，在加载时，动态链接器会重定位GOT中的每个条目，使得它包含目标的正确的绝对地址；在过程（函数）第一次被调用时，进行函数地址的重定位。

在 dl\_init 调用之前，GOT存放的是对应PLT条目的第二条指令。



图5.5 dl\_init 调用之前

PLT条目的第二条指令：

在 dl\_init 调用之后，GOT存放的是函数地址。



图5.5 dl\_init 调用后

在函数首次调用时，首先跳转到 PLT 执行.plt 中逻辑，第一次访问跳转时 GOT 地址为下一条指令，将函数序号(ID)压栈，然后跳转到 PLT[0]，在 PLT[0]中将重定位表地址压栈，然后访问动态链接器，在动态链接器中使用函数序号和重定位表确定函数运行时地址，重写 GOT，再将控制传递给目标函数。之后如果对同样函数调用，第一次访问跳转直接跳转到目标函数。

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

在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，让人感叹计算机系统的复杂、精巧。

经预处理、编译、汇编、链接，一个完整的hello诞生了！

**（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。

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。

## 6.2 简述壳Shell-bash的作用与处理流程

Shell 的作用：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

在终端中键入./hello 1180300308 刘义 3，运行的终端程序会对输入的命令行进行解析，因为hello 不是一个内置的shell 命令。所以解析之后终端程序判断./hello 的语义为执行当前目录下的可执行目标文件hello，之后终端程序首先会调用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。

## 6.4 Hello的execve过程

当fork 之后，子进程调用execve 函数（传入命令行参数）在当前进程的上下文中加载并运行一个新程序即hello 程序，execve 调用驻留在内存中的被称为启动加载器的操作系统代码来执行hello 程序，加载器删除子进程现有的用户虚拟内存段，并创建一组新的代码、数据、堆和栈段。新的栈和堆段被初始化为零，通过将虚拟地址空间中的页映射到可执行文件的页大小的片，新的代码和数据段被初始化为可执行文件中的内容。最后加载器设置PC 指向\_start 地址，\_start 最终调用hello中的main 函数。除了一些头部信息，在加载过程中没有任何从磁盘到内存的数据复制。直到CPU引用一个被映射的虚拟页时才会进行复制，这时，操作系统利用它的页面调度机制自动将页面从磁盘传送到内存。

加载器创建的内存映像如下：

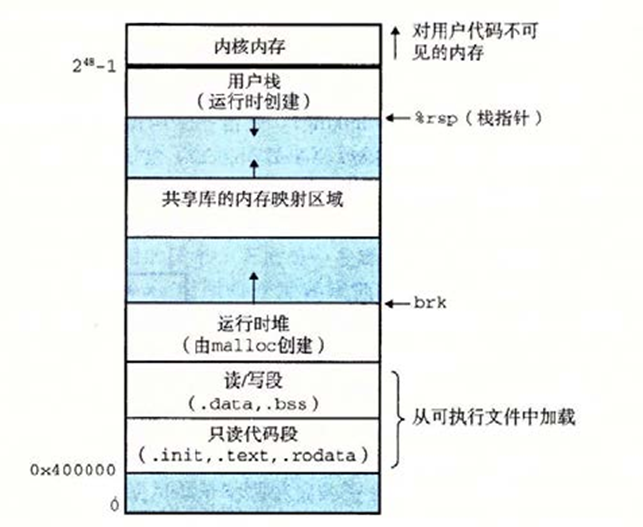


图 6.1 启动加载器创建的系统映像

## 6.5 Hello的进程执行

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。

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。

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。

在hello运行过程中，若hello进程不被抢占，则正常执行；若被抢占，则进入内核模式，进行上下文切换，转入用户模式，调度其他进程。

这里有个特殊的情况，调用sleep函数。在调用sleep函数之前，如果没有别的原因使得scheduler将hello挂起的话，sleep函数会显式地请求kernel将hello挂起，此时进入内核模式切换到其他进程，切换回用户模式运行那个进程，直到atoi(argv[3])秒之后，收到了sleep结束的信号，又会切换到内核模式，回到hello，输出我的学号和姓名，接着又一次调用sleep函数。

## 6.6 hello的异常与信号处理

1. 正常执行hello程序：

* 进程收到的信号：无
* 进程收到的异常：
  1. 时钟中断：不处理
  2. 系统调用：调用write函数打印字符
  3. 缺页异常（可能发生）：缺页异常处理程序

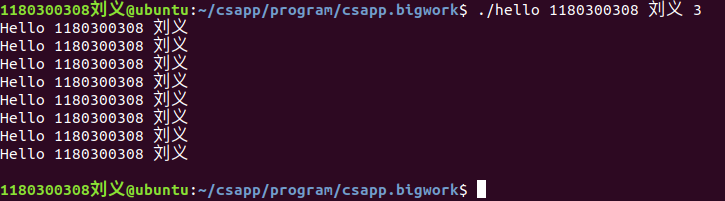


图6.2 程序正常执行

1. 执行Ctrl-z操作：

* 进程收到的信号：SIGSTOP、挂起（停止）
* 进程收到的异常：

1. 处理器中断： 不处理

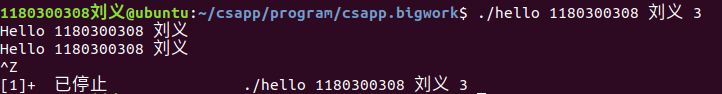


图6.3 程序输出2条info后按下 Ctrl-z

1. 执行ps命令：

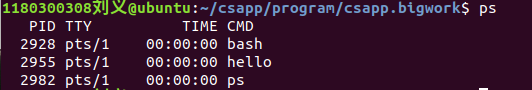


图6.4 执行ps命令

1. 执行jobs命令：



图6.5 执行jobs命令

1. 执行pstree命令：

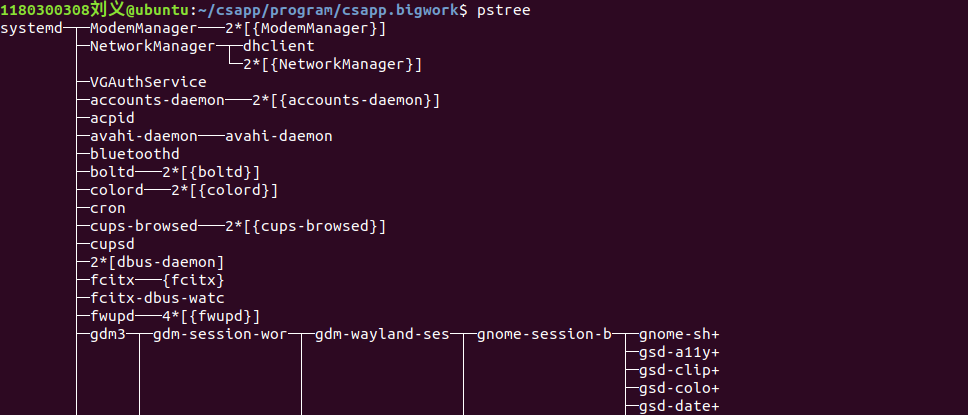


图6.6 执行pstree命令

1. 执行fg命令：

* 进程收到的信号：SIGCONT、继续运行
* 收到的异常：

1. 时钟中断：不处理

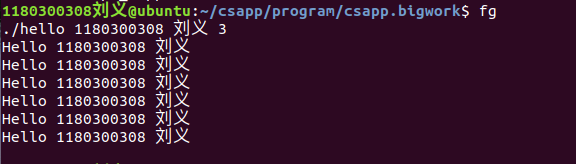


图6.7 执行fg命令

1. 执行 kill -9 %1命令：

* 进程收到的信号：SIGKILL、杀死进程
* 收到的异常：

1. 时钟中断：不处理

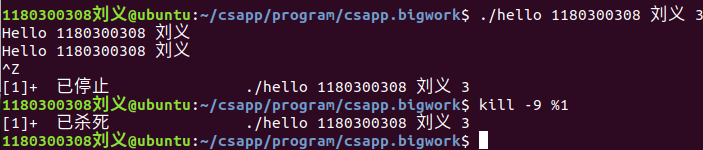


图6.8 执行kill -9 %1命令

1. 执行Ctrl-c指令：

* 进程收到的信号：SIGINT（shell使进程终止)
* 收到的异常：

1. 时钟中断：不处理

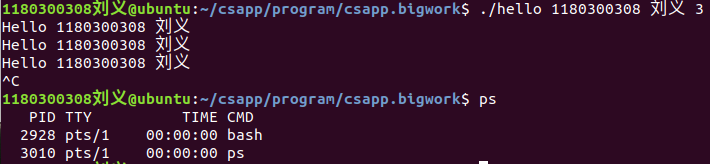


图6.9 执行Ctrl-c命令

1. 运行中乱按：

* 进程收到的信号：无
* 收到的异常：

1. 时钟中断：不处理
2. 系统调用：调用write函数打印字符
3. I/O中断：将键盘输入的字符读入缓冲区
4. 缺页故障（可能）：缺页处理程序

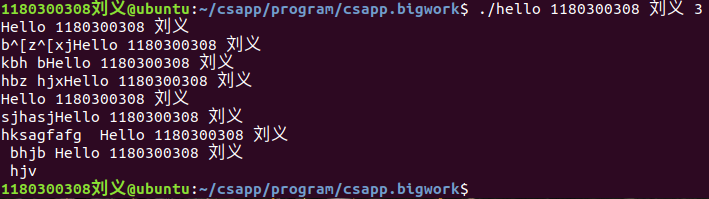


图6.10 运行中乱按

1. 回车

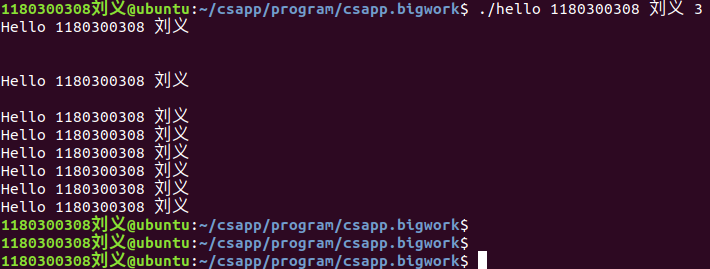


图6.11 回车

## 6.7本章小结

本章简述了进程、shell的概念与作用，并描述了fork和execve的过程，最后给出了执行hello时遇到的异常和信号的处理方式。

**（第6章1分）**

# 第7章 hello的存储管理

## 7.1 hello的存储器地址空间

逻辑地址：逻辑地址（logical address）是指由程序产生的与段相关的偏移地址部分，由段选择符+偏移地址构成。hello.o里面的相对偏移地址是逻辑地址。

线性地址：线性地址（linear address）是逻辑地址到物理地址变换的中间层，是处理器可寻址空间的地址。程序代码产生的逻辑地址加上段基地址就产生了线性地址。

虚拟地址：即为上述的线性地址，虚拟地址很明显是线性（连续）的。在之前hello链接的时候，各种函数的地址，比如main函数的地址是0x0000000000400582，实际上是虚拟地址，这是将物理地址映射成了虚拟地址，方便加载、链接等。

物理地址：物理地址（physical address）是CPU外部地址总线上的寻址信号，是地址变换的最终结果，一个物理地址始终对应实际内存中的一个存储单元。Hello实际存储的空间（内存）使用的是物理地址。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

逻辑地址=段选择符 + 偏移地址

线性地址 = 段基址 + 偏移地址

逻辑地址到线性地址的变换——段式管理的关键在于；段选择符🡪段基址

变换在处理器的两种寻址模式——实模式和保护模式下有所不同：

实模式：

段基址 = 段选择符 \* 16

线性地址 = 段选择符 \* 16 + 偏移地址

保护模式：

保护模式是现代计算机常用的寻址模式。保护模式下，段选择符作为一个索引，到一个称为描述符表的数据结构中读取段基址。此时，16位的段选择符也被划分成了几个部分，予以不同的解释：

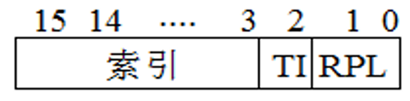


图7.1 保护模式下的段选择符

TI=0，选择全局描述符表(GDT)，TI=1，选择局部描述符表(LDT)。

RPL=00，为第0级，位于最高级的内核态，RPL=11，为第3级，位于最低级的用户态。

高13位（8K个索引）：用来确定当前使用的段描述符在描述符表中的位置。

计算机维护一系列表，称为描述符表。描述符表分为三种：全局描述符表（GDT）、局部描述符表（LDT）、中断描述符表（IDT）。描述符表的每个表项，大小为8个字节，称为段描述符。段描述符的内容如下图。

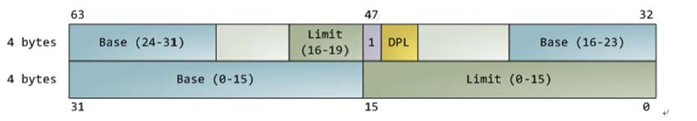


图7.2 段描述符的结构

BASE：段基址

Limit：指出这个段有多大。

DPL：描述符的特权级（privilege level），其值从0（最高特权，内核模式）到3（最低特权，用户模式），用于控制对段的访问。

整个过程如下图所示：

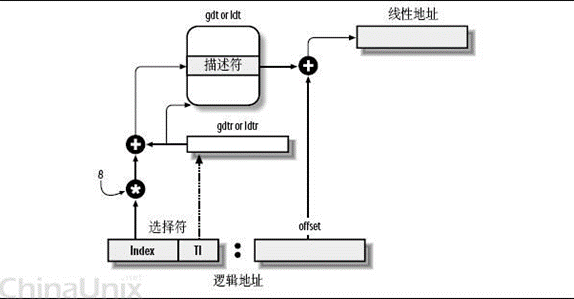


图7.3 保护模式下逻辑地址🡪线性地址的过程

根据段寄存器中的段选择符，跳转到对应的描述符表中的段描述符，其中的BASE位即为段基址。

## 7.3 Hello的线性地址到物理地址的变换-页式管理

从概念上讲，虚拟内存被组织为磁盘上存储的N个连续字节大小的单元的阵列。每个字节都有一个唯一的虚拟地址，用作数组的索引。磁盘上阵列的内容缓存在主内存中。与存储器层次结构中的任何其他高速缓存一样，磁盘上的数据被划分为块，这些块充当磁盘和主存储器之间的传输单元，这些块被称为页。每个虚拟页的大小为P = 2 ^ p字节。同样，物理内存被划分为物理页，大小也为P= 2 ^ p字节。

通过段式管理，我们得到了线性地址（虚拟地址VA）。虚拟地址被分为两个部分：VPN（虚拟页号）和VPO（虚拟页偏移量），相应的物理地址也被分为两个部分：PPN（物理页号）和PPO（物理页偏移量）。由于虚拟内存与物理内存的页大小相同，因此VPO与PPO相同。页式管理的关键在于：VPN🡪 PPN。

内存中常驻页表，描述了从虚拟页🡪物理页的映射关系。根据VPN可以在页表中找到相应的页表项（PTE），从中取出PPN（此处先不考虑多级页表）。处理流程如下：

若PTE的有效位为1，页命中，则获取到PPN，与PPO组成物理地址。

若PTE的有效位为0，说明对应虚拟页没有缓存到物理内存中，引发一个缺页故障，调用操作系统的内核的缺页处理程序，确定牺牲页，并调入新的页面。再返回到原来的进程，再次调用导致缺页的指令。这次页命中，则获取到PPN，与PPO组成物理地址。

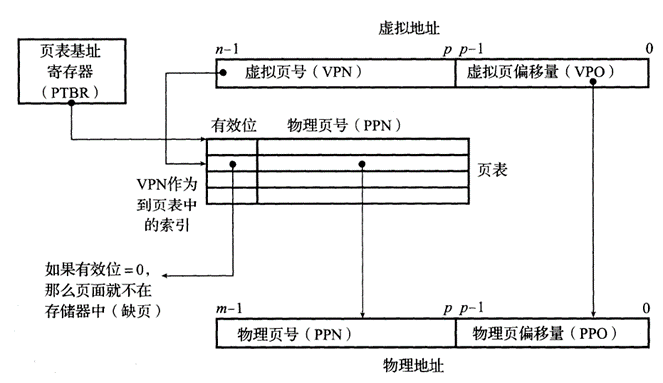


图7.4 地址翻译

## 7.4 TLB与四级页表支持下的VA到PA的变换

TLB：页表常驻内存，地址翻译时MMU需要去内存中访问，为减少这样的开销，MMU中包含了一个关于PTE的小缓存，称为翻译后备缓冲器（TLB）。

多级页表：将页表分为多个层级，以减少内存要求。

CPU将VPN进一步划分为TLBT（TLB标记）和TLBI（TLB索引）。

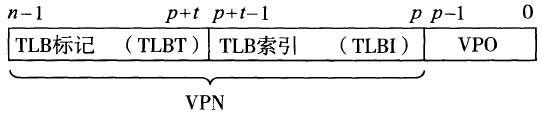


图7.5 虚拟地址的进一步划分

由TLBI，访问TLB中的某一组，遍历该组中的所有行，若找到一行的tag等于TLBT，且有效位valid为1,，则缓存命中，该行存储的即为PPN；否则缓存不命中，需要到页表中找到被请求的块替换原TLB表项中的块。

多级页表的访问：缓存不命中后，VPN被解释成从低位到高位的等长的4段，从高地址开始，第一段VPN作为第一级页表的索引，用以确定第二级页表的基址；第二段VPN作为第二级页表的索引，用以确定第三级页表的基址；第三段VPN作为第三级页表的索引，用以确定第四级页表的基址；第四段VPN作为第四级页表的索引，若该位置的有效位为1，则第四级页表中的该表项存储的是所需要的PPN的值。

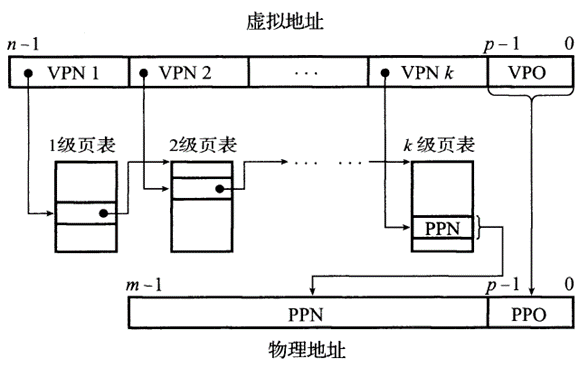


图7.6 多级页表的访问

## 7.5 三级Cache支持下的物理内存访问

CPU将物理地址解释为三部分：CT（缓存标记），CI（组索引），CO（块偏移）。

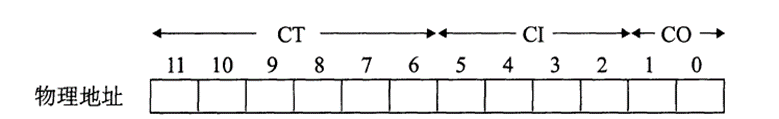


图7.7 物理地址的划分

首先根据组索引CI，遍历缓存中对应组的所有行，若找到一行的tag等于CT，且标志位valid为1，则缓存命中（hit），根据CO（块偏移）读取块中对应的数据；否则缓存不命中。

若缓存不命中，则向下一级缓存中查找数据（L1 Cache 🡪L2 Cache🡪L3Cache 🡪主存🡪磁盘）。找到数据之后，开始进行行替换。若该组中有空行，那就将数据缓存至这个空行，并设置tag和valid位；若该组中没有空行，根据一些策略来决定驱逐哪一行，如最不常使用（LFU）策略和最近最少使用（LRU）策略。LFU策略会替换过去某个时间段内访问次数最少的那一行，LRU策略会替换最后一次访问时间最久远的那一行。

## 7.6 hello进程fork时的内存映射

当前进程调用fork函数时，内核会为新进程创建各种数据结构，并分配给它一个唯一的PID。为了给这个新流程创建虚拟内存，它创建了当前流程的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标记为只读，并将两个进程中的每个区域结构标记为私有的写时复制。

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

——CS:APP

## 7.7 hello进程execve时的内存映射

execve函数在当前进程的上下文中加载并执行可执行目标文件hello 中的程序。加载并运行hello 需要执行以下步骤：

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，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。

——CS:APP

## 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。

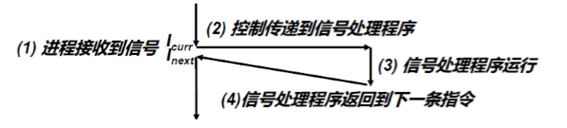


图7.8 进程对信号的响应

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，就将其写回，然后换入新的页面并更新页表。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。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。



图7.9 堆区域

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：要求分配器自动释放不再使用的已分配的块，这个过程被称为垃圾收集。

空闲块的组织方式有：

一、带边界标签的隐式空闲链表

隐式空闲链表的空闲块是通过头部中的大小字段隐含地连接着的，分配器可以通过头部遍历堆中的所有的块，从而间接地遍历整个空闲块的集合。为了方便合并空闲块，需要脚部，脚部是头部的副本，方便寻找上一个块。 如下图：

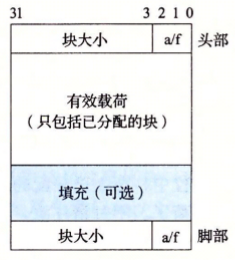


图7.10 隐式空闲链表

空闲块合并：因为有了Footer，所以我们可以方便的对前面的空闲块进行合并。合并的情况一共分为四种：前空后不空，前不空后空，前后都空，前后都不空。对于四种情况分别进行空闲块合并，我们只需要通过改变Header 和Footer 中的值就可以完成

二、显式空闲链表

将堆中的空闲块组织成一个双向空闲链表，在每个空闲块中，都包含一个pred（前驱）和succ（后继）指针，如下图：

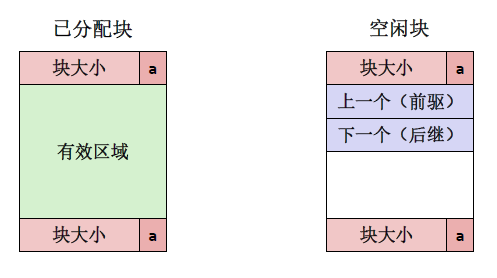


图7.11 显式空闲链表

使用双向链表而不是隐式空闲链表，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。释放一个块的时间则取决于我们选择的空闲链表中块的排序策略。

空闲链表中块的排序策略有：1）后进先出（LIFO），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。2）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，以intel Core7为例介绍了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：所有的IO设备都被模型化为文件

设备管理：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：

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. 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，标准输出，标准错误。
3. 改变当前文件位置：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时执行读操作时触发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：当应用完成了访问，它就通知内核关闭这个文件，并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。

返回：若成功则为新文件描述符，若出错为-1。

1. int close(fd)，fd是需要关闭文件的描述符。

返回：若成功则为新文件描述符，若出错为-1。

1. ssize\_t read(int fd,void \*buf,size\_t n)，该函数从描述符为fd的当前位置最多赋值n个字节到内存buf的位置，返回值为实际传送的字节数量。返回：若成功则为新文件描述符，若出错为-1。
2. 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。

返回：若成功则为新文件描述符，若出错为-1。

## 8.3 printf的实现分析

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

首先查看printf函数的函数体：

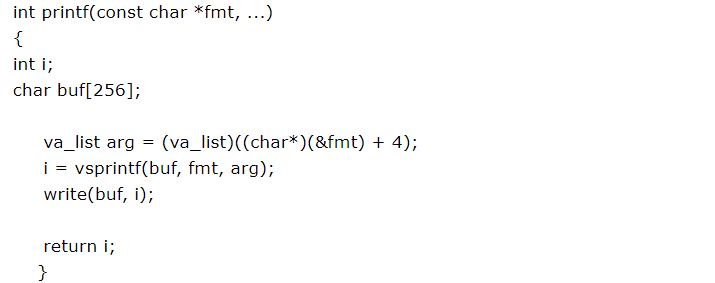


图8.1 printf函数

来看这句：va\_list arg = (va\_list)((char\*)(&fmt) + 4);

其中，va\_list的定义： typedef char \*va\_list，是字符型指针。

(char\*)(&fmt) + 4) 表示的是...中的第一个参数（函数参数从右往左依次压栈）。

然后我们再来查看vsprintf函数：

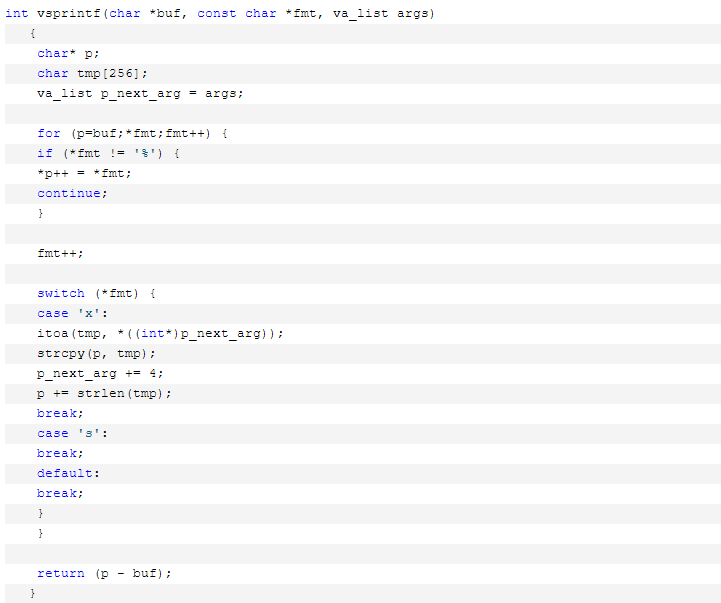


图8.2 vsprintf函数

语句i = vsprintf(buf, fmt, arg);得到要打印出来的字符串长度。

write(buf, i)将buf中的i个元素写到终端。

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

下面我们来看write函数：

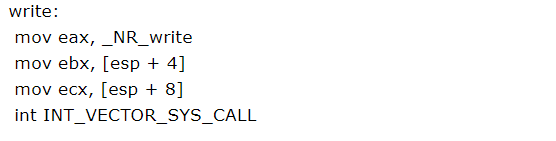


图8.3 write函数

这里是给几个寄存器传递了几个参数，然后一个语句：int INT\_VECTOR\_SYS\_CALL代表通过系统调用syscall，查看syscall的实现：

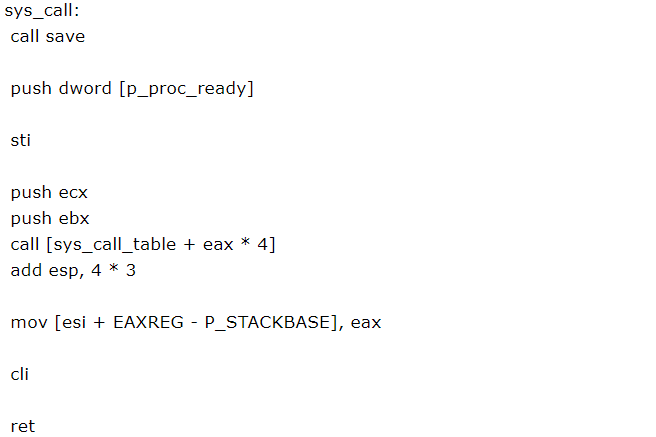


图8.4 syscall函数

syscall 将字符串中的字节“Hello 1180300308刘义\n”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：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

异步异常-键盘中断的处理：键盘接口得到一个代表该按键的键盘扫描码，同时同时产生中断请求，请求抢占当前进程运行键盘中断子程序，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，了解了 printf 函数和 getchar 函数的实现。

**（第8章1分）**

# 结论

hello一生大事记：

1. 预处理：将hello.c调用的所有外部的库展开合并到一个hello.i 文件中。
2. 编译：将hello.i 编译成汇编文件hello.s。
3. 汇编：将hello.s 汇编成可重定位目标件hello.o。
4. 链接：hello.o 和动态链接库链接成为可执行目标程序hello。
5. 加载运行：shell中输入./hello1180300308 刘义 3，shell为其新建子进程（fork），把代码和数据加载入虚拟内存空间（execve），然后进入 main 函数。
6. 执行指令：在该进程被调度时，CPU 为其分配时间片，在一个时间片中，hello 享有CPU全部资源，PC寄存器一步一步地更新，CPU不断地取指，执行自己的控制逻辑流。
7. 访存：内存管理单元（MMU）将逻辑地址转化成物理地址，通过三级高速缓存访问物理内存/磁盘中的数据。
8. 动态申请内存：比如printf 会调用malloc 向动态内存分配器申请堆中的内存。
9. 信号处理：进程时刻等待着信号，如果运行途中键入ctr-c ctr-z 等则调用shell 的信号处理函数分别进行停止、挂起等操作。
10. 终止并被回收：shell父进程等待并回收子进程，内核删除为这个进程创建的所有数据结构（task\_struct等）。

前四个步骤：预处理、编译、汇编、链接是hello“诞生”的过程。



图10.1 hello的“诞生”

后面则是hello“执行”的过程。

**（结论0分，缺失 -1分，根据内容酌情加分）**

# 附件

|  |  |
| --- | --- |
| 文件名称 | 文件作用 |
| hello.c | hello程序c语言源文件 |
| hello.i | hello.c预处理生成的文本文件 |
| hello.s | hello.i编译得到的汇编文件 |
| hello.o | hello.s汇编得到的可重定位目标文件 |
| hello | hello.o与动态链接库链接得到的可执行目标文件 |
| hello.o.objdmp | 使用objdump反汇编hello.o得到的反汇编文件 |
| hello.objdmp | 使用objdump反汇编hello得到的反汇编文件 |
| hello.o.elf | hello.o的ELF格式 |
| hello.elf | hello的ELF格式 |

**（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] 大卫R.奥哈拉伦，兰德尔E.布莱恩特. 深入理解计算机系统[M]. 机械工业出版社.2018.4.

[2] 几种地址的含义：<https://blog.csdn.net/geekcome/article/details/6263057>

[3] 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）**