

**计算机系统**

**大作业**

题 目 程序人生-Hello’s P2P

专 业 计算机科学与技术

学　　 号 1171000410

班　　 级 1703005

学 生 强文杰

指 导 教 师 吴锐

**计算机科学与技术学院**

**2018年12月**

**摘 要**

摘要：本文通过对hello程序P2P和020的整体介绍，阐述了hello经过多种处理后成为Process，并且hello进程创建直到回收的全部过程。以这些过程的分析为例，我们更好地说明了计算机的底层实现，并且更深地阐明了整个程序的生命周期。

**关键词：**P2P；020；进程；计算机系统

**（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 。linux中，hello.c经过cpp的预处理、ccl的编译、as的汇编、ld的链接最终成为可执行目标程序hello，在shell中键入启动命令后，shell为其fork产生子进程的过程。

020: shell通过execve加载并执行hello，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。

## 1.2 环境与工具

硬件环境：X64 CPU ，2.50GHz ， 8G RAM

软件环境：Windows 10 64位 ，Vmware 14 ，Ubuntu 16.04 LTS 64 位

开发工具：gcc + gedit , Codeblocks , gdb edb

## 1.3 中间结果

hello.c ：hello源代码

hello.i ：预处理后的文本文件

hello.s ：hello.i编译后的汇编文件

hello.o ：hello.s汇编后的可重定位目标文件

hello\_objdump ：hello的反汇编代码

hello.0\_objdump ：hello.o的反汇编代码

hello ：链接后的可执行文件

## 1.4 本章小结

本章对hello进行了简单的介绍，分析了其P2P和020的过程，列出了本次任务的环境和工具，并且阐明了任务过程中出现的中间产物及其作用。

**（第1章0.5分）**

# 第2章 预处理

## 2.1 预处理的概念与作用

预处理器(cpp)根据以#开头的命令，修改原始的C程序。比如hello.c中第6行的#include<stdio.h>命令高速预处理器读取系统头文件stdio.h的内容，并把它直接插入程序文本中。结果就得到了另一个C程序，通常是以.i作为文件拓展名

## 2.2在Ubuntu下预处理的命令

命令：gcc -E hello.c -o hello.i

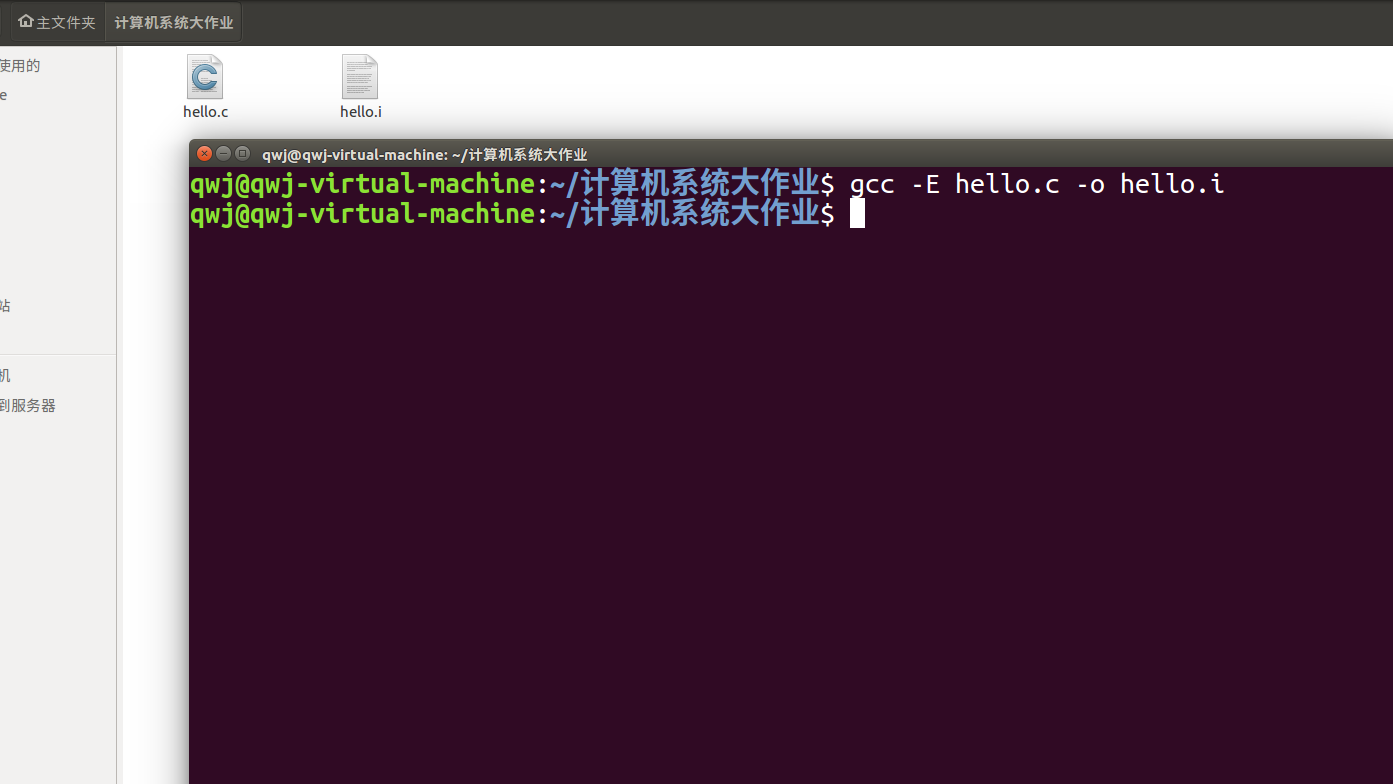


图2-1 hello.c预处理产生hello.i

## 2.3 Hello的预处理结果解析

修改得到的C程序hello.i已经从原来hello.c的534个字节增加到63364个字节，并且增加到3110行。再用gedit打开hello.i，发现在main函数在文件的最后部分。

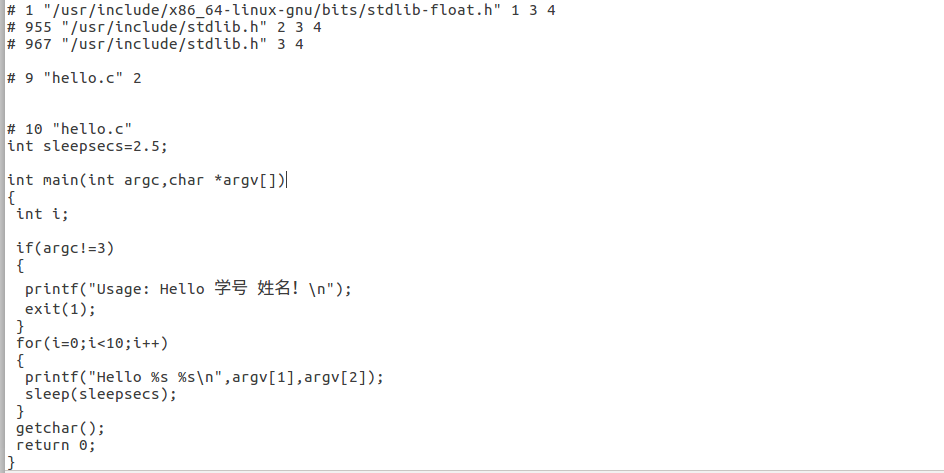


图2-2 hello.o中main函数的位置

而在main函数之前，预处理器(cpp)读取头文件stdio.h 、stdlib.h 、和unistd.h中的内容，三个系统头文件依次展开。比如stdio.h的展开，打开usr/include/stdio.h发现了其中还含有#开头的宏定义等，预处理器会对此继续递归展开，最终的.i程序中没有#define，并且针对#开头的条件编译语句，cpp根据#if后面的条件决定需要编译的代码。

## 2.4 本章小结

本章介绍了hello.c的预处理阶段，根据预处理命令得到了修改后的hello.i程序，并且对hello.i程序进行了预处理结果解析，理解了预处理器读取系统头文件中内容，并把它插入程序文本中的过程。

**（第2章0.5分）**

# 第3章 编译

## 3.1 编译的概念与作用

在这个阶段，编译器首先要检查代码的规范性，是否有语法错误等，以确定代码的实际要做的工作，再检查无误后，编译器（ccl）见文本文件hello.i翻译成文本文件hello.s，它包含一个汇编语言程序。该程序包含函数main的定义，语句以一种文本格式描述了一条低级机器语言指令。汇编语言位不同高级语言的不同编译器提供了通用的输出语言。

## 3.2 在Ubuntu下编译的命令

命令：gcc -S hello.i -o hello.s

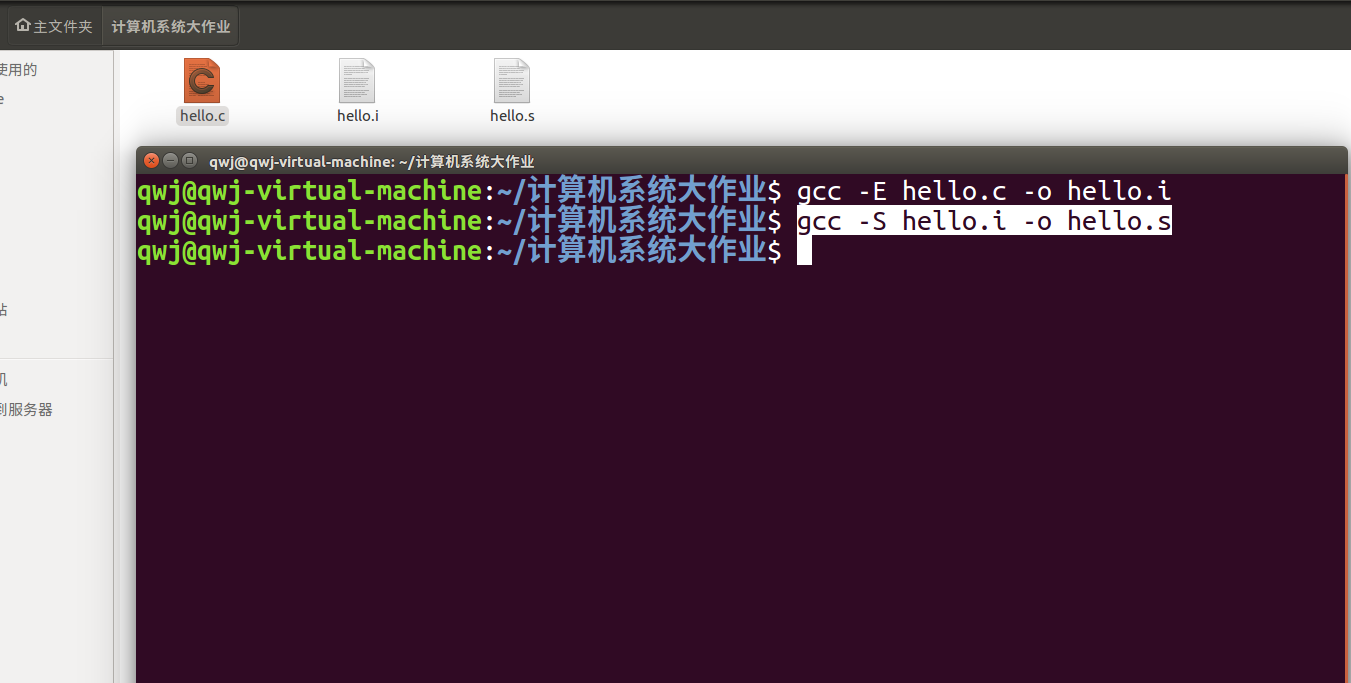


图3-1 hello.i编译生成hello.s文件

## 3.3 Hello的编译结果解析

**先介绍hello.s文件开头的部分：**

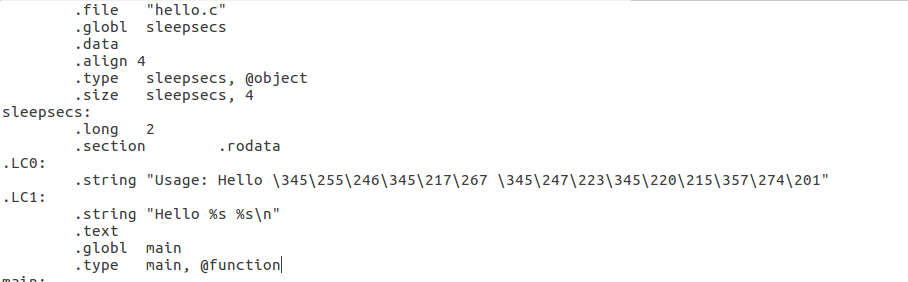


图3-1 hello.s各种汇编指令

.file：源文件名

.globl：全局变量

.data：数据段

.align：对齐方式

.type：指定是对象类型或是函数类型

.size：大小

.long：长整型

.section .rodata：下面是.rodata节

.string：字符串

.text：代码段

**再根据PPT中P4给出的参考C数据与操作，进行如下解析：**

**3.3.1 数据**

hello.s中C语言的数据类型主要有：全局变量，局部变量，指针数组

整型变量：

1. int sleepsecs=2.5；在C程序中sleepsecs是已初始化的全局变量，存放在.data节中。根据图3-2，可知sleepsecs被定义为全局变量globl，在.data段中，sleepsecs被设置为对齐方式是4，类型是object，大小是4字节。

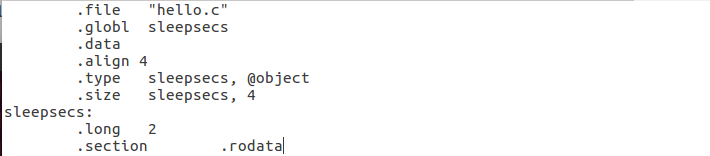


图3-2 hello.s中的全局变量sleepsecs

2. int argc；argc是函数传入的第一个int型参数，存储在%edi中

3. int i；局部变量，通常保存在寄存器或是栈中。根据movl $0, -4(%rbp)操作可知i的数据类型占用了4字节的栈空间。

常量：在hello.s中一些如3和9的常量以立即数的形式出现。

字符串，argv[1]和argv[2]都声明在.rodata只能读数据段中，并且给出了字符串的编码。具体见图3-4.

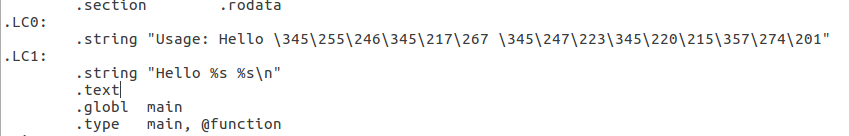


图3-3 hello.s中的字符串

**3.3.2 赋值**

首先是对全局变量sleepsecs的赋值，赋初值为2：

9

图3-4 sleepsecs的赋值

其次是对局部变量i的赋值：使用movl语句，对应于C程序中i=0 。

10

图3-5 i的赋值

**3.3.3 类型转换**

因为2.5是浮点数类型，因为sleepsecs是int型，当给它赋值为2.5时，值会向零舍入，正如图3-5中sleepsecs隐式地被赋值为2 。

**3.3.4 算术操作**

编译器将i++编译为：

11

图3-6 i++的算术操作

**3.3.5 关系操作**

1. i<10的关系操作编译为：

12

图3-7 i<10

2. argc!=3的关系操作编译为：

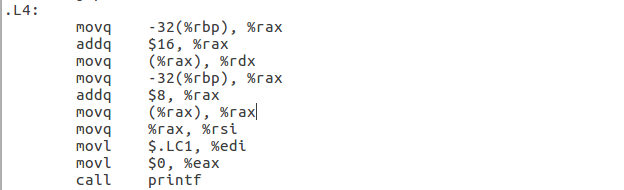
13

图3-8 argc!=3

以上的关系操作均是为设置条件码，并且下一步的跳转做准备。

**3.3.6 数组/指针/结构操作**

指针数组：char \*argv[]：在argv数组中，argv[0]指向输入程序的路径和名称，argv[1]和argv[2]分别表示两个字符串。其中char\* 数据类型占8个字节，根据图3-3，可知通过(%rax)和%rax+8，分别得到argv[1]和argc[2]两个字符串。



argv[2]

argv[1]

图3-9 解析指针数组argv

**3.3.7 控制转移**

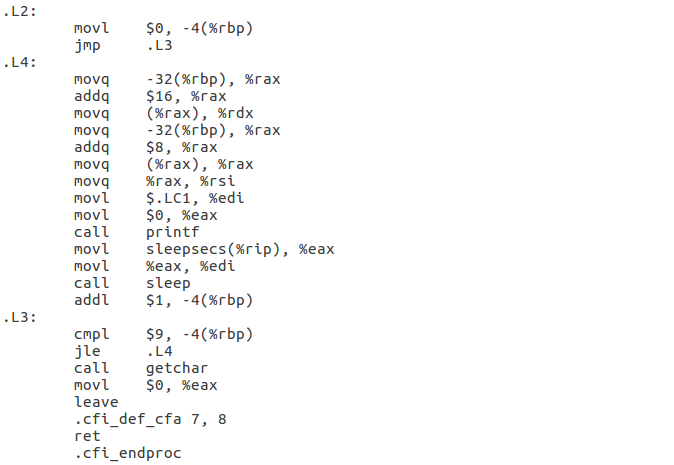
1. if(argc!=3) 。当argc不等于3时进行跳转。cmpl语句比较 -20(%rbp)和-3，设置条件码，判断ZF零标志，如果最近的操作得出的结果为0，则跳到.L2中，否则顺序执行下一条语句。

13

图3-10 argc的控制转移

2. for(i=0;i<10;i++) ：for循环里面的比较和转移过程，如图3-11.

for循环的控制时比较cmpl $9, -4(%rbp) ，当i大于9时跳出循环，否则进入.L4循环体内部执行。



循环体

循环直到i大于9

图3-11 for循环的控制转移

**3.3.8 函数操作**

1.main函数：

参数传递：传入参数argc和argv，分别用寄存器%rdi和%rsi存储。

函数调用：被系统启动函数调用。

函数返回：设置%eax为0并且返回，对应return 0 。

2.printf函数：

参数传递：call puts时只传入了字符串参数首地址；for循环中call printf时传入了 argv[1]和argc[2]的地址。

函数调用：for循环中被调用

3.exit函数：

参数传递：传入的参数为1，再执行退出命令

函数调用：if判断条件满足后被调用

4.sleep函数：

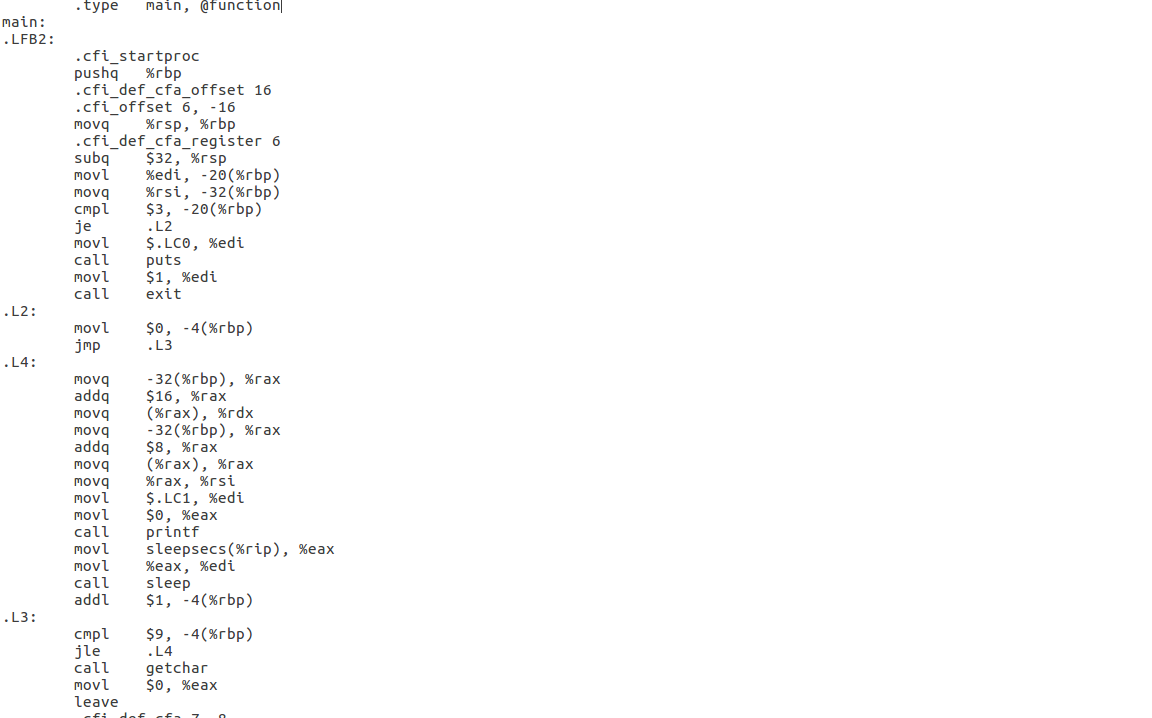
参数传递：传入参数sleepsecs，传递控制call sleep

函数调用：for循环下被调用

5.getchar

传递控制：call getchar

函数调用：在main中被调用



exit函数

getchar函数

sleep函数

printf函数

main函数

## 3.4 本章小结

编译程序所要作得工作就是通过词法分析和语法分析，在确认所有的指令都符合语法规则之后，将其翻译成等价的中间代码表示或汇编代码。

通过对编译的结果进行解析，更深刻地理解了C语言的数据与操作，并且对C语言翻译成汇编语言有了更好的掌握。

**（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hello.s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，并将结果保存在目标文件hello.o中。hello.o文件是一个二进制文件，它包含的是程序的指令编码。

## 4.2 在Ubuntu下汇编的命令

汇编命令：gcc -c hello.s -o hello.o

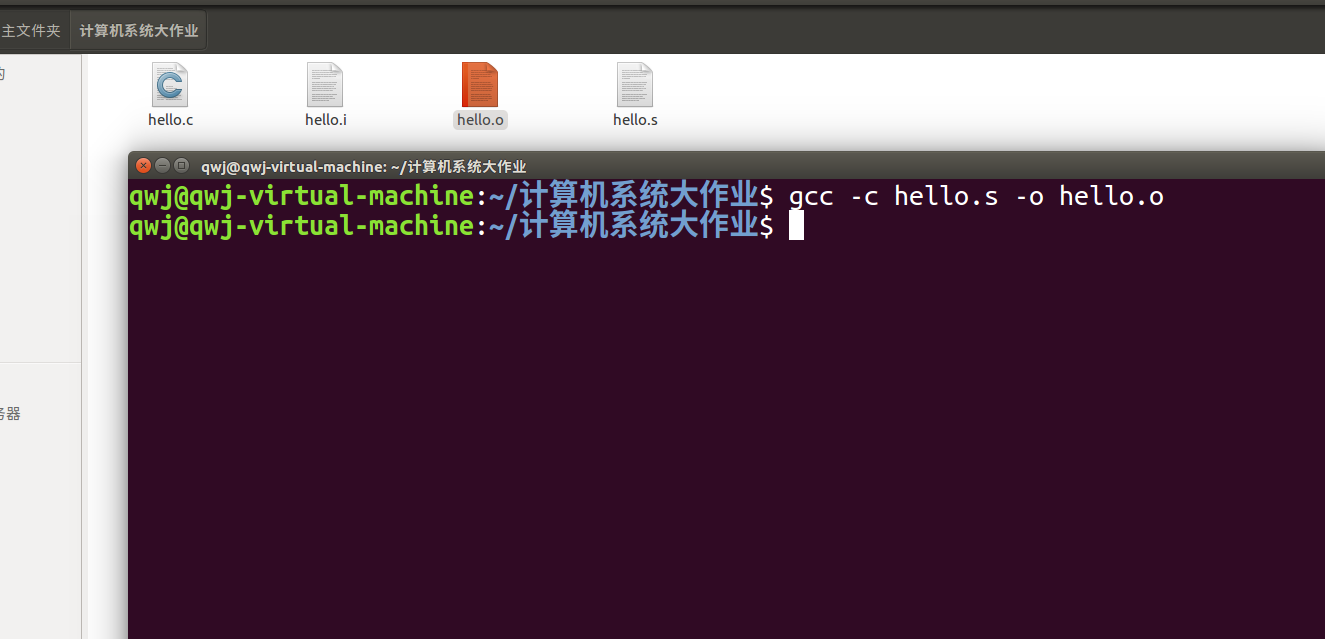


图4-1 hello.s汇编生成hello.o文件

## 4.3 可重定位目标elf格式

1. ELF头描述了生成该文件的系统的字的大小和字节顺序，并且包含帮助链接器语法分析和解释目标文件的信息。

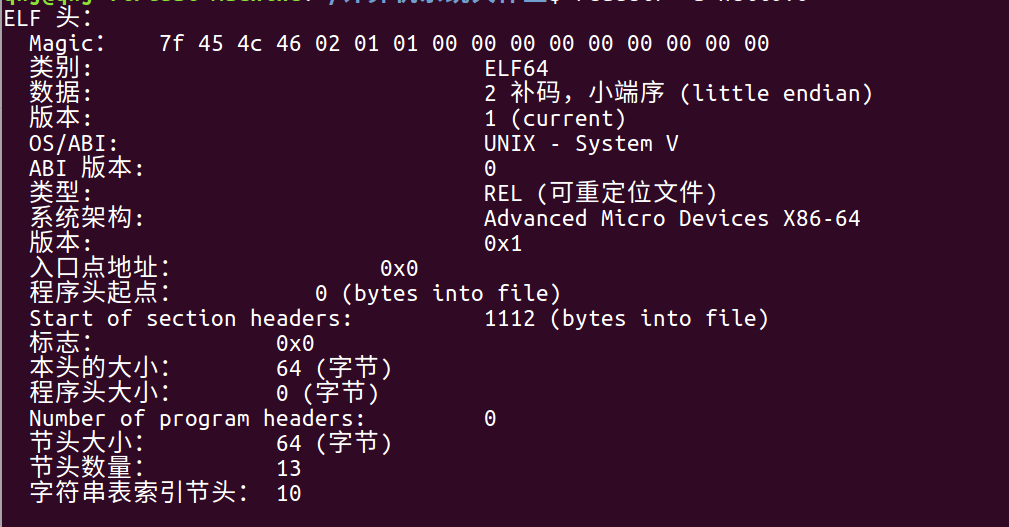


图4-2 ELF头

2. 节头部表描述了不同节的位置和大小，其中目标文件中每个节都有一个固定大小的条目。具体的描述包括节的名称、类型、地址和偏移量等。



图4-3 节头部表

3. 当汇编器生成一个目标模块是，它并不知道数据和代码最终将放在内存中的什么位置，它也不知道这个模块引用的任何外部定义的函数或者全局变量的位置。所以，无论何时汇编器遇到对最终位置未知的目标引用，它就会生成一个**重定位条目**，告诉链接器在将目标文件合并成可执行目标文件时如何修改这个引用。代码的重定位条目放在.rel.text中，已初始化数据的重定位条目放在.rel.data中。

ELF重定位条目的数据结构如下：

typedef struct{

long offset; /\*需要被修改的引用的节偏移\*/

long type:32, /\*重定位类型\*/

symbol:32; /\*标识被修改引用应该指向的符号\*/

long attend; /\*符号常数，对修改引用的值做偏移调整\*/

}Elf64\_Rela;

两种最基本的重定位类型：

R\_X86\_64\_PC32 ：重定位一个使用32位PC相对地址的引用。

R\_X86\_64\_32 ：重定位一个使用32位PC绝对地址的引用。

根据图4-4，可以看出8条重定位信息的详细情况，分别对符号.rodata，函数puts，exit等，加数也在符号名称之后。

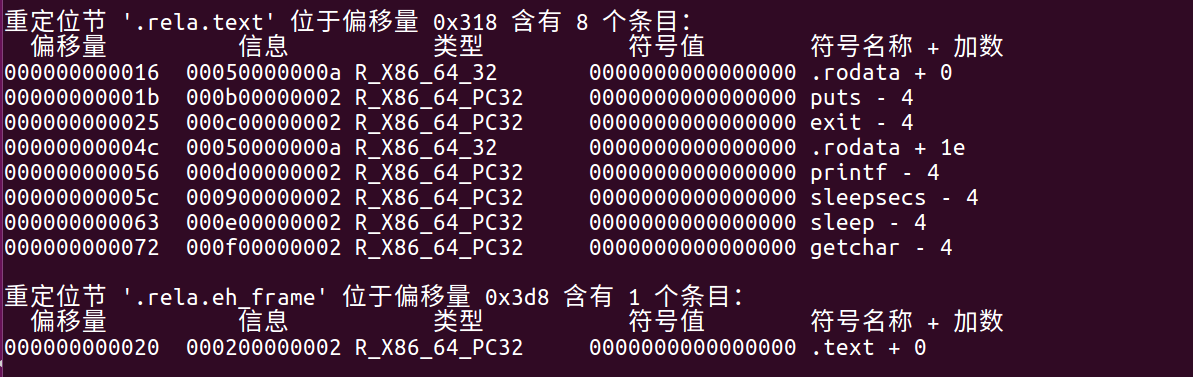


图4-4 重定位节

4. .symtab是一个符号表，它存放在程序中定义和引用的函数和全局变量的信息。

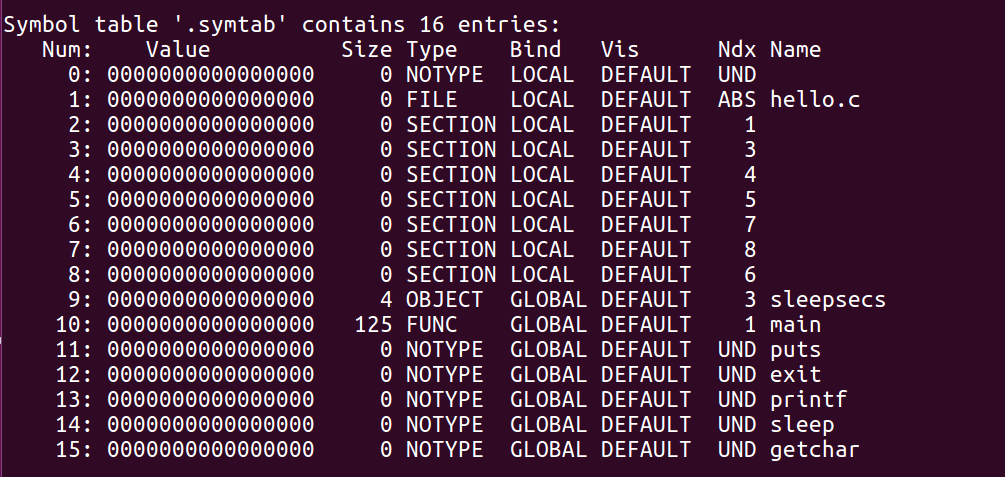


图4-5 符号表.symtab

## 4.4 Hello.o的结果解析

反汇编命令：objdump -d -r hello.o

对比hello.s文件和反汇编代码，主要有以下的差别

1. 操作数：hello.s中的操作数时十进制，hello.o反汇编代码中的操作数是十六进制。

2. 分支转移：跳转语句之后，hello.s中是.L2和.L3等段名称，而反汇编代码中跳转指令之后是相对偏移的地址。

3. 函数调用：hello.s中，call指令之后直接是函数名称，而反汇编代码中call指令之后是函数的相对偏移地址。因为函数只有在链接之后才能确定运行执行的地址，因此在.rela.text节中为其添加了重定位条目。

4. 全局变量的访问：在hello.s文件中，对于.rodata和sleepsecs等全局变量的访问，是$.LC0和sleepsecs(%rip)，而在反汇编代码中是$0x0和0(%rip)，是因为它们的地址也是在运行时确定的，因此访问也需要重定位，在汇编成机器语言时，将操作数全部置为0，并且添加重定位条目。

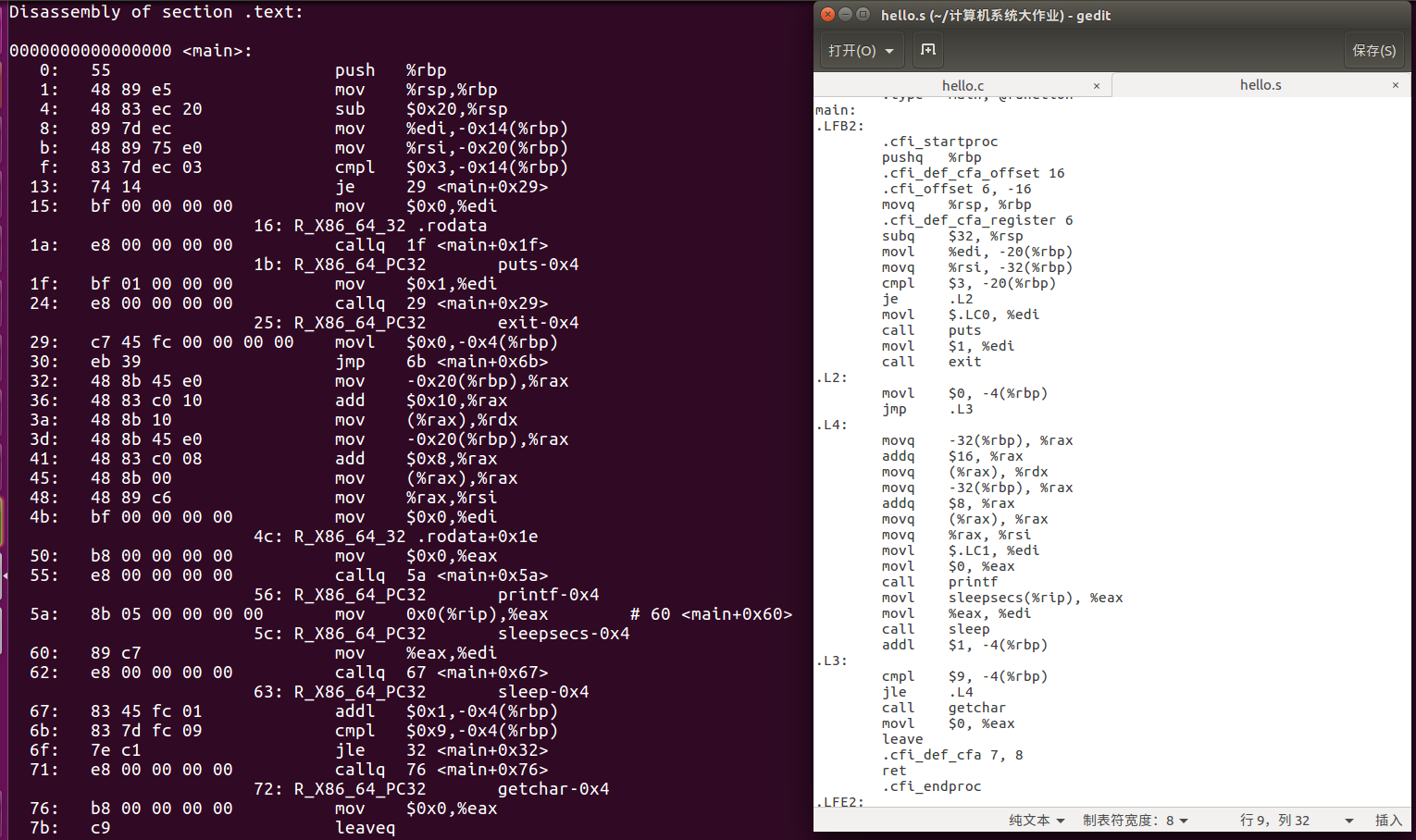


图4-6 hello.s和反汇编代码的对比

## 4.5 本章小结

通过汇编操作，汇编语言转化为机器语言，hello.o可重定位目标文件为后面的链接做了准备。通过对比hello.s和反汇编代码的区别，更深刻地理解了汇编语言到机器语言实现地转变，和这过程中为链接做出的准备（设置重定位条目等）。

**（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（复制）到内存并执行。

作用：当程序调用函数库（如标准C库）中的一个函数printf，printf函数存在于一个名为printf.o的单独的预编译好了的目标文件中，而这个函数必须通过链接器（ld）将这个文件合并到hello.o程序中，结果得到hello文件，它是一个可执行目标文件，可以被加载到内存中，由系统执行。另外，链接器在软件开发中扮演着一个关键的角色，因为它们使得分离编译成为可能。

## 5.2 在Ubuntu下链接的命令

ld链接命令：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

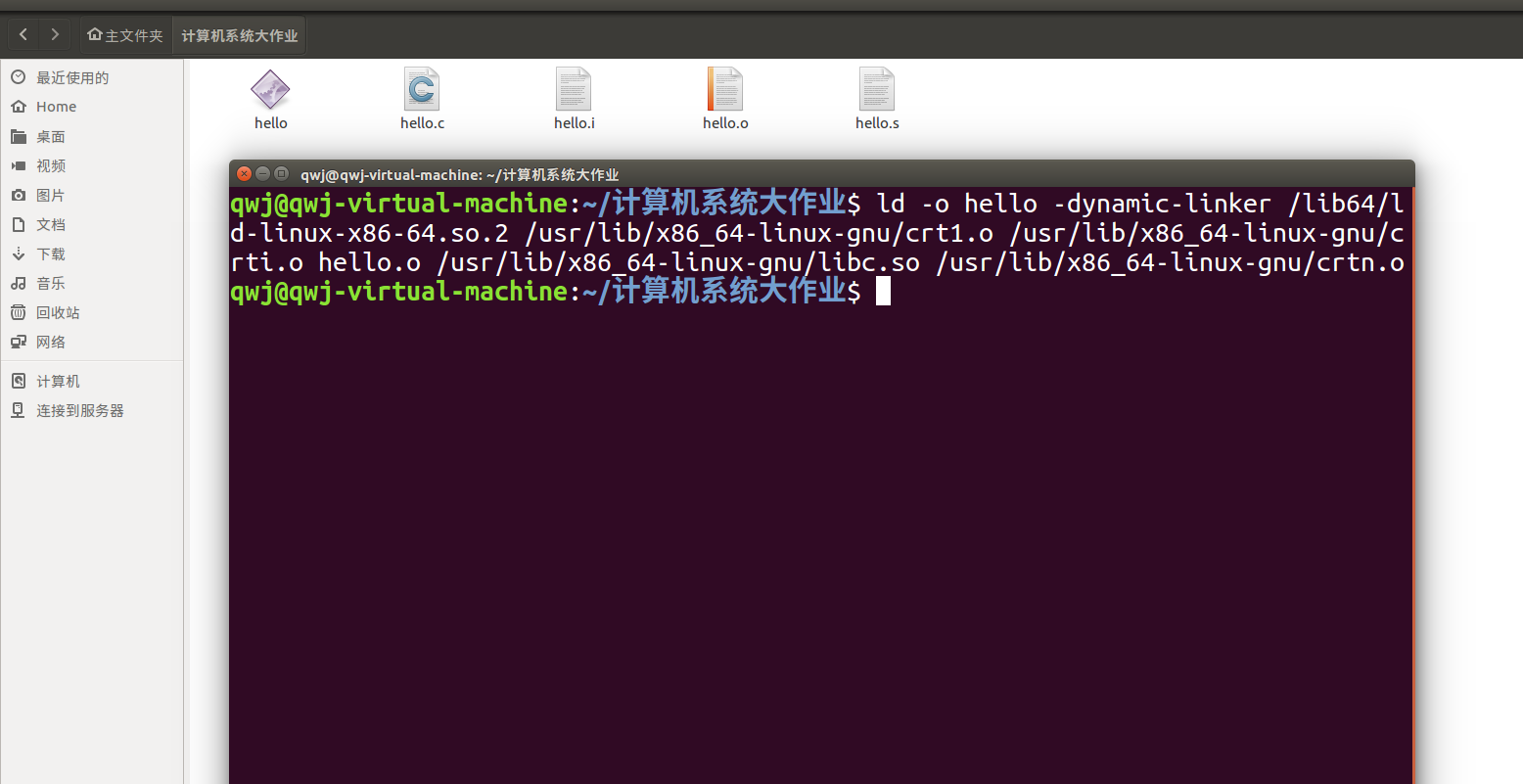


图5-1 使用ld链接生成可执行目标文件hello

## 5.3 可执行目标文件hello的格式

可执行目标文件hello的格式类似于可重定位目标文件的格式，ELF头描述文件的总体格式。它还包括程序的入口点，也就是当程序运行时要执行的第一条指令的地址。.text、.rodata和.data节与可重定位目标文件中的节是类似的，除了这些节已经被重定位到它们最终的运行时的内存地址外。.init节定义了一个小函数\_init，程序初始化代码会调用它。因为可执行文件时完全连接的，所以无.rel节。

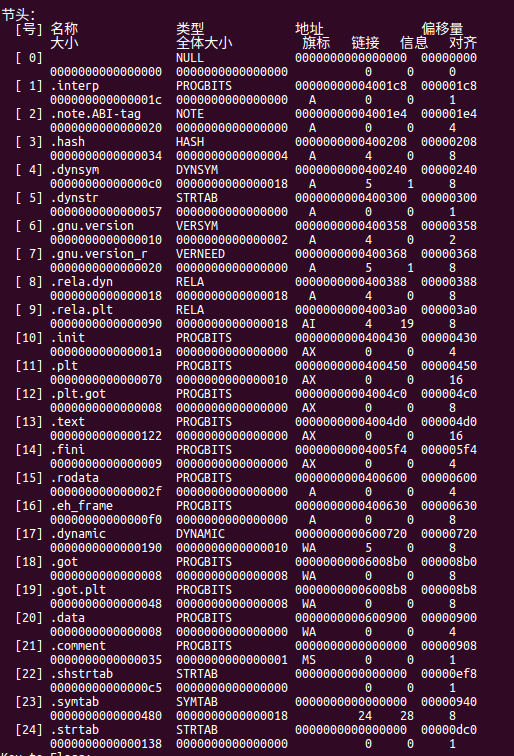
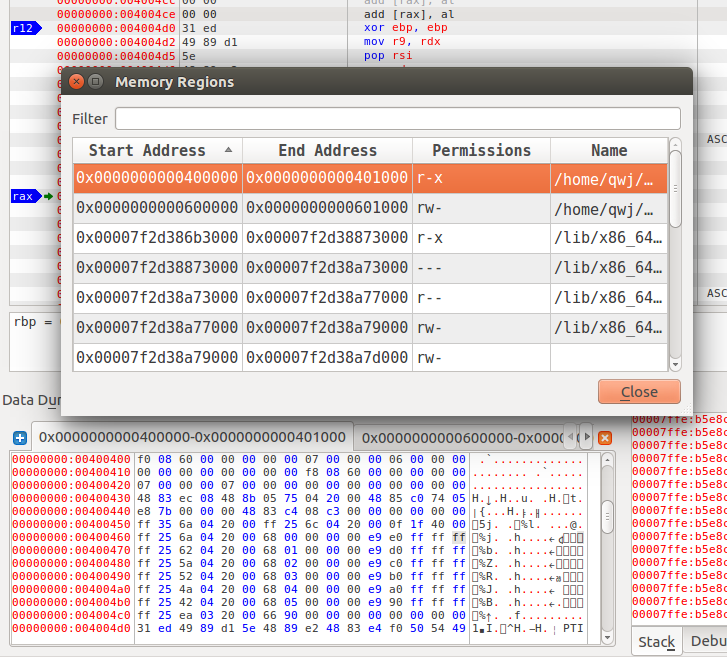


图5-2 hello文件elf各段的基本信息

## 5.4 hello的虚拟地址空间

在edb中打开hello，通过Data Dump查看hello程序的虚拟地址空间各段信息。在Memory Regions选择View in Dump可以分别在Data Dump中看到只读内存段和读写内存段的信息。图5-3展示了.text节开始于00000000004004d0，并且显示了其在虚拟地址空间的信息。



.text节开始地址

读/写段

只读代码段

.text节开始地址

图5-3 使用edb查看hello虚拟地址空间段信息

为了补充说明，图5-4是可执行文件的程序头部表，展示了ELF可执行文件的连续的片被映射到连续的内存段的映射关系。其中展示了目标文件中的偏移，内存地址，对其要求，目标文件中的段大小，内存中的段大小，运行时访问权限等信息。图中黑色底的部分是只读代码段，白色底的部分是读/写段。

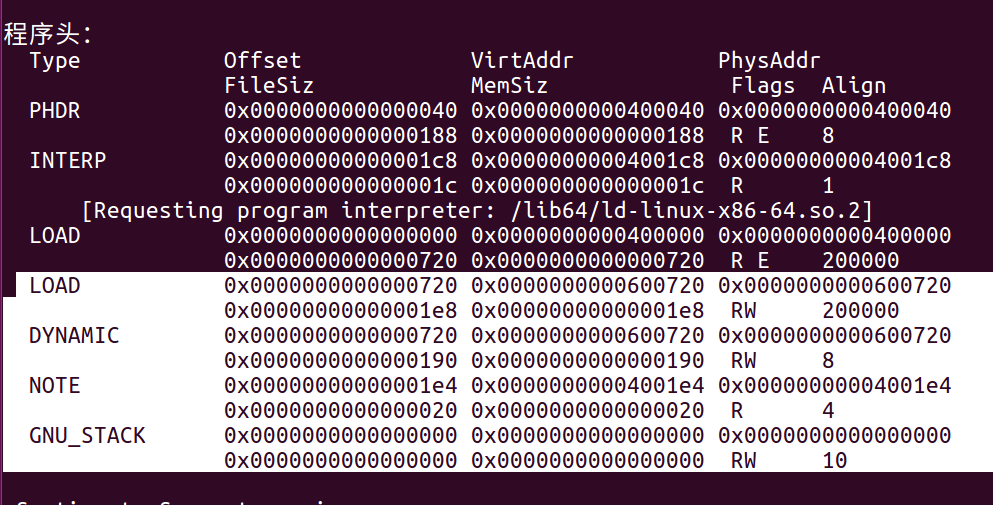


图5-4 可执行文件hello的程序头部表

## 5.5 链接的重定位过程分析

反汇编命令：objdump -d -r hello

**hello与hello.o主要有以下的不同**：

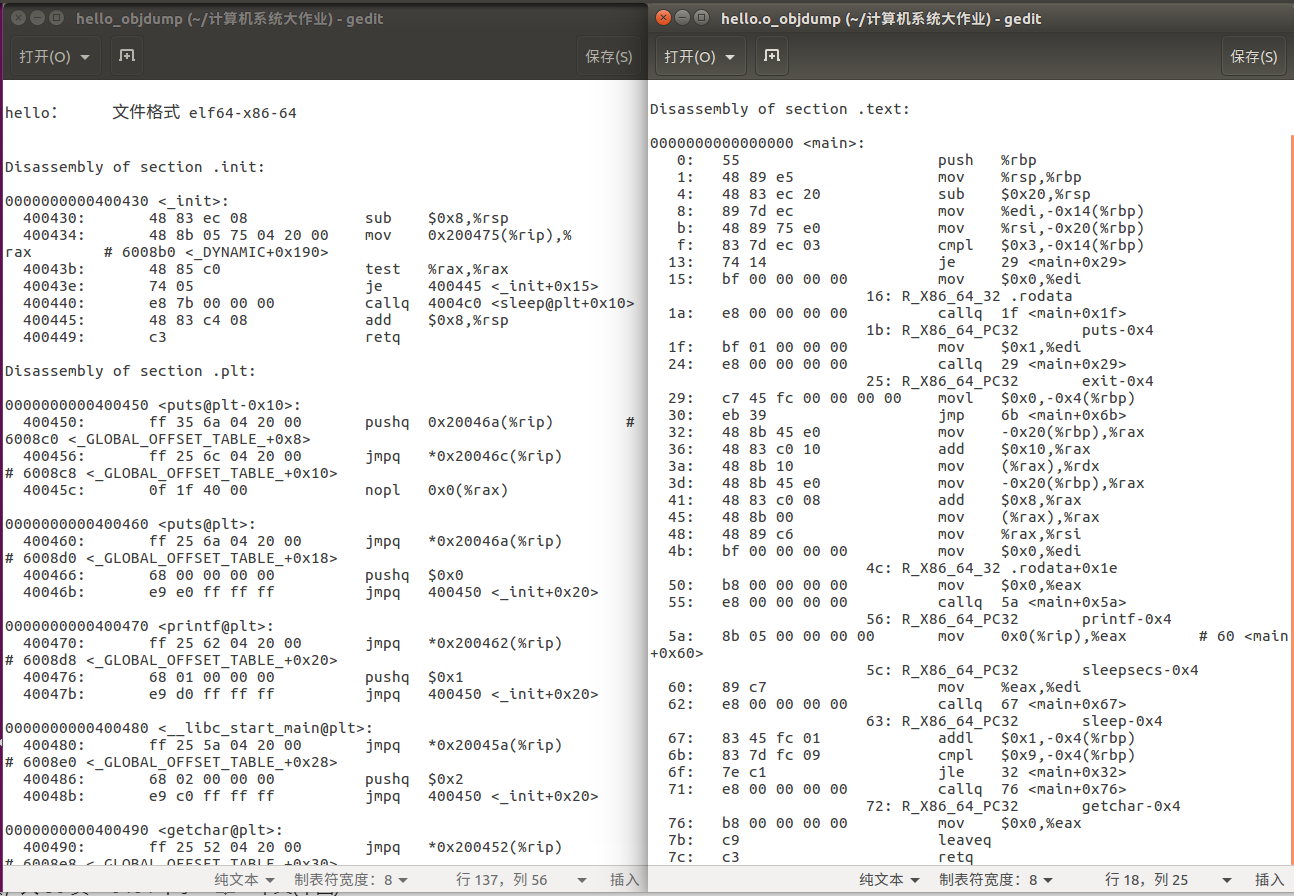
1.**链接增加新的函数：**在hello中链接加入了在hello.c中用到的函数，如exit、printf、sleep、getchar等函数。

2.**增加的节**：hello中增加了.init和.plt节，和一些节中定义的函数。

3.**函数调用**：hello中无hello.o中的重定位条目，并且跳转和函数调用的地址在hello中都变成了虚拟内存地址。对于hello.o的反汇编代码，函数只有在链接之后才能确定运行执行的地址，因此在.rela.text节中为其添加了重定位条目。

4.**地址访问**：hello.o中的相对偏移地址变成了hello中的虚拟内存地址。而hello.o文件中对于.rodata和sleepsecs等全局变量的访问，是$0x0和0(%rip)，是因为它们的地址也是在运行时确定的，因此访问也需要重定位，在汇编成机器语言时，将操作数全部置为0，并且添加重定位条目。

根据hello和hello.o的不同，分析出链接的过程为：链接就是链接器ld将各个目标文件组装在一起，就是把.o文件中的各个函数段按照一定规则累积在一起，比如规则：解决符号依赖，库依赖关系，并生成可执行文件。



跳转的虚拟地址

**链接加入的函数和其虚拟地址**

**新的节.init**

**以下为重定位项目**

**hello.o反汇编文件**

**hello反汇编文件**

图5-4 hello和hello.o反汇编代码对比分析

**根据hello.o中的重定位项目，分析hello重定位过程：**

重定位过程合并输入模块，并为每个符号分配运行时地址，主要有以下两步：

1.重定位节和符号定义。在这一步中，链接器将所有相同类型的节合并为同一类型的新的聚合节。包括hello.o在内的所有可重定位目标文件中的.data节被全部合并成一个节，这个节成为输出的可执行目标文件hello中的.data节。然后，连接器将运行时内存地址赋给新的聚合节，赋给输入模块定义的每个节，以及赋给输入模块定义的每个符号。当这一步完成时，hello中每条指令和包括.rodata、sleepsecs等全局变量都有唯一的运行时内存地址了。

2.重定位节中的符号引用。链接器依赖于hello.o中的重定位条目，修改代码节和数据节中对每个符号的引用，使得它们指向正确运行时的地址。

## 5.6 hello的执行流程

hello调用与跳转的各个子程序名或程序地址如下：

0x400430 init;

0x400460 puts@plt;

0x400470 printf@plt;

0x400480 \_\_libc\_start\_main@plt;

0x400490 getchar@plt;

0x4004a0 exit@plt;

0x4004b0 sleep@plt;

0x4004d0 \_start;

0x4004fe main;

0x400580 \_\_libc\_csu\_init;

0x4005f0 \_\_libc\_csu\_fini;

0x4005f4 \_fini;

## 5.7 Hello的动态链接分析

hello程序的动态链接项目：global\_offset表 。

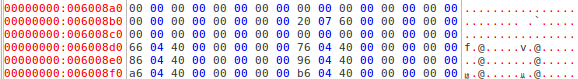


图5-5 do\_init前

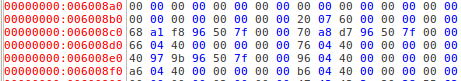


图5-6 do\_init后

如图5-5和图5-6，在do\_init前后global\_offset表发生变化。

在edb调试之后我们发现原先0x006008c0开始的global\_offset表是全0的状态，在执行过\_dl\_init之后被赋上了相应的偏移量的值。这说明dl\_init操作是给程序赋上当前执行的[内存地址](https://www.baidu.com/s?wd=%E5%86%85%E5%AD%98%E5%9C%B0%E5%9D%80&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)偏移量 。

## 5.8 本章小结

本章主要理解了Ubuntu下链接的过程，链接就是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。通过查看hello的虚拟地址空间，并且对比hello.o和hello的反汇编代码，更好地掌握了链接尤其是重定位的过程，但是我们知道链接并不止于此，hello会在它运行时要求动态链接器加载和链接某个共享库，而无需在编译时将那些库链接到应用中。

**（第5章1分）**

# 第6章 hello进程管理

## 6.1 进程的概念与作用

进程的概念：狭义上：进程是一个执行中程序的示例。广义上：进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。

进程的作用：

1.每次用户通过向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，然后在这个进程的上下文中运行这个可执行目标文件。应用程序也能够创建新进程，并且在新进程的上下文中运行它们自己的代码或其他应用程序。

2.进程提供给应用程序的关键抽象：一个独立的逻辑控制流，好像我们的程序独占地使用处理器；一个私有的地址空间，好像我们的程序独占地使用内存系统。

## 6.2 简述壳Shell-bash的作用与处理流程

1.shell的作用

实际上Shell是一个命令[解释器](https://baike.baidu.com/item/%E8%A7%A3%E9%87%8A%E5%99%A8)，它解释由用户输入的命令并且把它们送到内核。不仅如此，Shell有自己的编程语言用于对命令的编辑，它允许用户编写由shell命令组成的程序。Shell编程语言具有普通编程语言的很多特点，比如它也有[循环结构](https://baike.baidu.com/item/%E5%BE%AA%E7%8E%AF%E7%BB%93%E6%9E%84)和分支控制结构等，用这种编程语言编写的Shell程序与其他应用程序具有同样的效果

2.shell的处理流程

shell首先检查命令是否是[内部命令](https://baike.baidu.com/item/%E5%86%85%E9%83%A8%E5%91%BD%E4%BB%A4)，若不是再检查是否是一个应用程序（这里的应用程序可以是Linux本身的实用程序，如ls和rm，也可以是购买的商业程序，如xv，或者是自由软件，如emacs）。然后shell在搜索路径里寻找这些应用程序（搜索路径就是一个能找到可执行程序的目录列表）。如果键入的命令不是一个内部命令并且在路径里没有找到这个可执行文件，将会显示一条错误信息。如果能够成功找到命令，该[内部命令](https://baike.baidu.com/item/%E5%86%85%E9%83%A8%E5%91%BD%E4%BB%A4)或应用程序将被分解为系统调用并传给Linux内核。

## 6.3 Hello的fork进程创建过程

根据shell的处理流程，输入命令执行当前目录下的可执行文件hello，父进程会通过fork函数创建一个新的运行的子进程Hello。Hello进程几乎但不完全与父进程相同，Hello进程得到与父进程用户级虚拟空间相同的（但是独立的）一份副本，包括代码和数据段、堆、共享库、以及用户栈。Hello进程还获得与父进程任何打开文件描述符相同的副本，这就意味着当父进程调用fork时，Hello进程可以读写父进程中打开的任何文件。父进程和Hello进程最大的区别在于它们有不同的PID。

fork函数只被调用一次，却会返回两次。在父进程中，fork返回Hello进程的PID，在Hello进程中，fork返回0 。

下面通过进程图来说明fork进程创建过程：

shell

fork

Hello子进程

图6-1 Hello的fork进程创建图

## 6.4 Hello的execve过程

execve函数在加载并运行可执行目标文件Hello，且带列表argv和环境变量列表envp。只有当出现错误时，例如找不到Hello，execve才会返回到调用程序。在execve加载了Hello之后，它调用启动代码。启动代码设置栈，并将控制传递给新程序的主函数，该主函数有如下的原型：

int main(int argc , char \*\*argv , char \*envp);

结合**虚拟内存和内存映射**过程，可以更详细地说明exceve函数实际上是如何加载和执行程序Hello，需要以下几个步骤：

1.删除已存在的用户区域。

2.映射私有区域。为Hello的代码、数据、bss和栈区域创建新的区域结构，所有这些区域都是私有的、写时复制的。

3.映射共享区域。比如Hello程序与标准C库libc.so链接，这些对象都是动态链接到Hello的，然后再用户虚拟地址空间中的共享区域内。

4.设置程序计数器（PC）。exceve做的最后一件事就是设置当前进程的上下文中的程序计数器，使之指向代码区域的入口点。

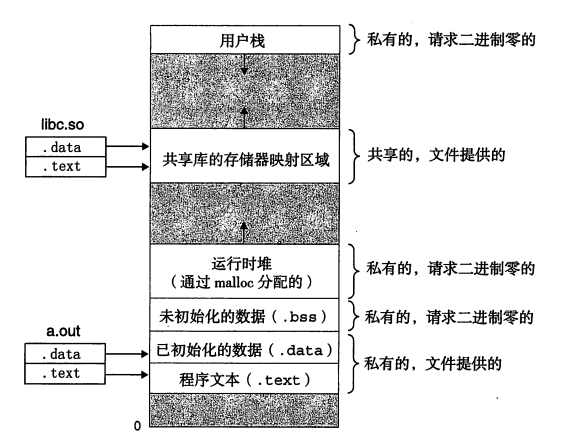


图6-2 加载器是如何映射用户地址空间的区域的

## 6.5 Hello的进程执行

**先简单阐述下面几个概念：**

**1.进程上下文信息**，就是内核重新启动一个被抢占的程序所需的状态，它由一些对象的值组成，这些对象包括通用目的寄存器、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。

**2.进程时间片**，是指一个进程和执行它的控制流的一部分的每一时间段。

**3.用户模式和内核模式：**处理器为了安全起见，不至于损坏操作系统，必须限制一个应用程序可执行指令能访问的地址空间范围。就发明了两种模式用户模式和内核模式，其中内核模式（上帝模式）有最高的访问权限，甚至可以停止处理器、改变模式位，或者发起一个I/O操作，处理器使用一个寄存器当作模式位，描述当前进程的特权。进程只有当中断、故障或者陷入系统调用时，才会将模式位设置成上帝模式，得到内核访问权限，其他情况下都始终在用户权限中，就能够保证系统的绝对安全。

由以上，首先阐述进程调度的过程：在Hello执行的某些时刻，比如sleep函数，内核可以决定抢占当前进程，并重新开始一个先前被抢占了的进程，这种决策就叫做调度。当内核调度了一个新的进程运行后，它就抢占Hello进程，并且使用上下文切换机制来将控制转移到新的进程。

Hello进程初始运行在用户模式中，直到Hello进程中的sleep系统调用，它显式地请求让Hello进程休眠，内核可以决定执行上下文切换，进入到内核模式。当定时器2.5后中断时，内核就能判断当前Hello休眠运行了足够长的时间，切换回用户模式。下面通过图6-3详细说明。

时间

进程Hello

其他进程

sleep

中断

从sleep返回

用户模式

内核模式

用户模式

内核模式

用户模式

上下文切换

上下文切换

图6-3 Hello进程上下文切换的剖析

## 6.6 hello的异常与信号处理

程序运行过程中可以按键盘，如不停乱按，包括回车，Ctrl-Z，Ctrl-C等，Ctrl-z后可以运行ps jobs pstree fg kill 等命令，请分别给出各命令及运行结截屏，说明异常与信号的处理。

hello执行过程中会出现的异常：

中断：信号SIGTSTP，默认行为是 停止直到下一个SIGCONT

终止：信号SIGINT，默认行为是 终止

下面演示程序运行时各命令情况：

1.hello运行时什么都不按。程序执行完后，进程被回收。再按回车键，结束。

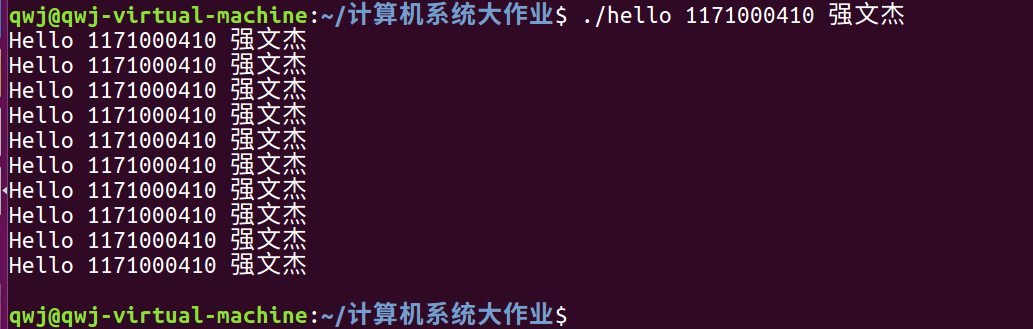


图6-4 正常运行hello程序

2.运行过程中按Ctrl+C。父进程收到SIGINT信号，终止hello进程，并且回收hello进程。

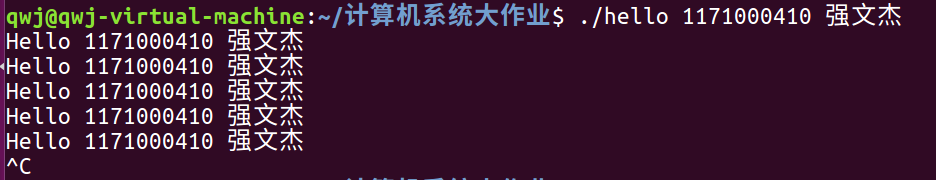


图6-5 运行时按Ctrl+C

3.运行时乱按。如图6-6，发现乱按的输入并不会影响进程的执行，当按到回车键时，getchar会读入回车符，并且后面的字符串会当作shell的命令行输入。

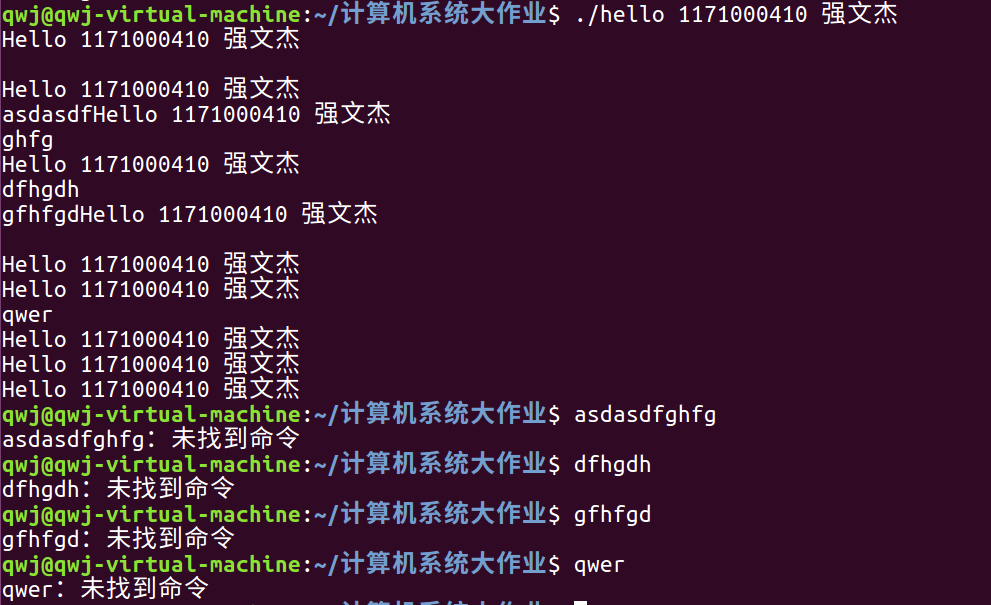


图6-6 hello运行时乱按

4.按下Ctrl+Z后运行ps命令。按下Ctrl+Z后，父进程收到SIGTSTP信号，将hello进程挂起，ps命令列出当前系统中的进程（包括僵死进程）。



图6-7 按下Ctrl+Z后运行ps命令

5. 按下Ctrl+Z后运行jobs命令。jobs命令列出 当前shell环境中已启动的任务状态。

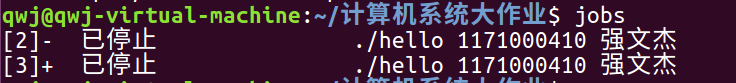


图6-8 按下Ctrl+Z后运行jobs命令

6.按下Ctrl+Z后运行pstree命令。pstree命令是以树状图显示进程间的关系。



图6-9 按下Ctrl+Z后运行pstree命令部分截图

7.fg命令将进程调到前台。



图6-10 fg命令

8.kill发送信号给一个进程或多个进程。通过kill -9 32309杀死pid为32309的进程。

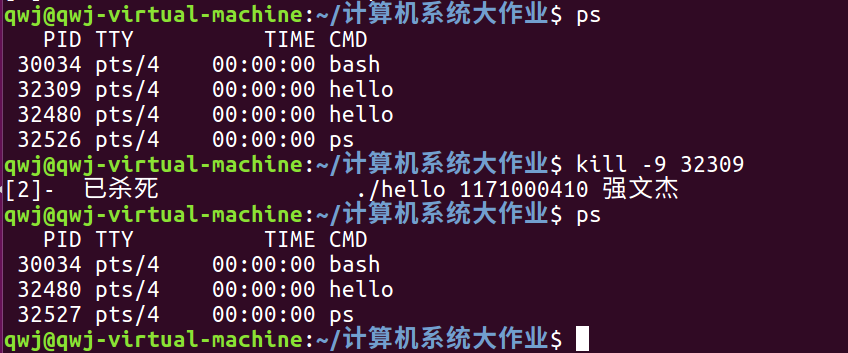


图6-11 kill命令

## 6.7本章小结

本章了解了hello进程的执行过程，主要是hello的创建、加载和终止，通过键盘输入，对hello执行过程中产生信号和信号的处理过程有了更多的认识，从而对异常的掌握加深了。

**（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：包含在机器语言中用来指定一个操作数或一条指令的地址。每一个逻辑地址都由一个段（segment）和偏移量（offset）组成，偏移量指明了从段开始的地方到实际地址之间的距离。就是hello.o里相对偏移地址。

线性地址：逻辑地址到物理地址变换之间的中间层。程序代码会产生逻辑地址，或者说是段中的偏移地址，加上相应段的基地址就生成了一个线性地址。是hello中的虚拟内存地址。

虚拟地址：一个带虚拟内存的系统中，CPU从一个有N=2^n个地址空间中生成虚拟地址。虚拟地址其实就是线性地址。

物理地址：用于内存芯片级的单元寻址，与处理器和CPU连接的地址总线相对应。地址翻译会将hello的一个虚拟地址转化为物理地址。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

一个逻辑地址由两部份组成，段标识符、段内偏移量。段标识符是由一个*16*位长的字段组成，称为段选择符。其中前*13*位是一个索引号。后面*3*位包含一些硬件细节，如图7-1：



图7-1 段选择符

索引号，是“段描述符(segment descriptor)”，段描述符具体地址描述了一个段。这样，很多个段描述符，就组了一个数组，叫“段描述符表”，这样，可以通过段标识符的前13位，直接在段描述符表中找到一个具体的段描述符，这个描述符就描述了一个段，由8个字节组成，如图7-2：

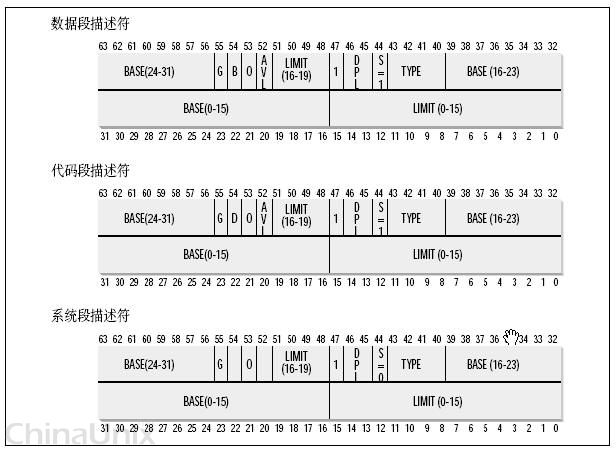


图7-2 段描述符

图示比较复杂，可以利用一个数据结构来定义它，不过，在此只关心一样，就是Base字段，它描述了一个段的开始位置的线性地址。 Intel设计的本意是，一些全局的段描述符，就放在“全局段描述符表(GDT)”中，一些局部的，例如每个进程自己的，就放在所谓的“局部段描述符表(LDT)”中。那究竟什么时候该用GDT，什么时候该用LDT呢？这是由段选择符中的T1字段表示的，=0，表示用GDT，=1表示用LDT。 GDT在内存中的地址和大小存放在CPU的gdtr控制寄存器中，而LDT则在ldtr寄存器中。再看图7-3比起来要直观些：

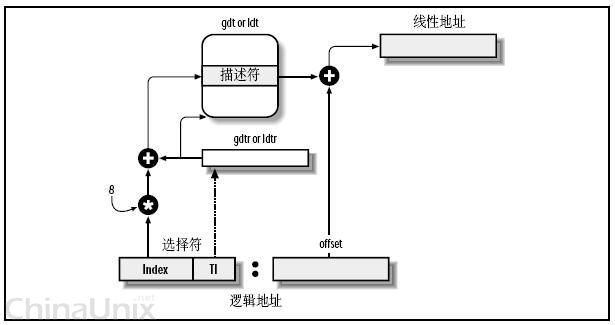


图7-3 概念间的关系说明

首先，给定一个完整的逻辑地址[段选择符：段内偏移地址]，

1、看段选择符的T1=0还是1，知道当前要转换是GDT中的段，还是LDT中的段，再根据相应寄存器，得到其地址和大小。我们就有了一个数组了。

2、拿出段选择符中前13位，可以在这个数组中，查找到对应的段描述符，这样，它了Base，即基地址就知道了。

3、把Base + offset，就是要转换的线性地址了。 还是挺简单的，对于软件来讲，原则上就需要把硬件转换所需的信息准备好，就可以让硬件来完成这个转换了。

## 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，线性地址被分为以固定长度为单位的组，称为页(page)，例如一个32位的机器，线性地址最大可为4G，可以用4KB为一个页来划分，这页，整个线性地址就被划分为一个tatol\_page[2^20]的大数组，共有2的20个次方个页。这个大数组我们称之为页目录。目录中的每一个目录项，就是一个地址——对应的页的地址。 另一类“页”，我们称之为物理页，或者是页框(frame)、页桢的。是分页单元把所有的物理内存也划分为固定长度的管理单位，它的长度一般与内存页是一一对应的。 这里注意到，这个total\_page数组有2^20个成员，每个成员是一个地址（32位机，一个地址也就是4字节），那么要单单要表示这么一个数组，就要占去4MB的内存空间。为了节省空间，引入了一个二级管理模式的机器来组织分页单元。如图7-4 。

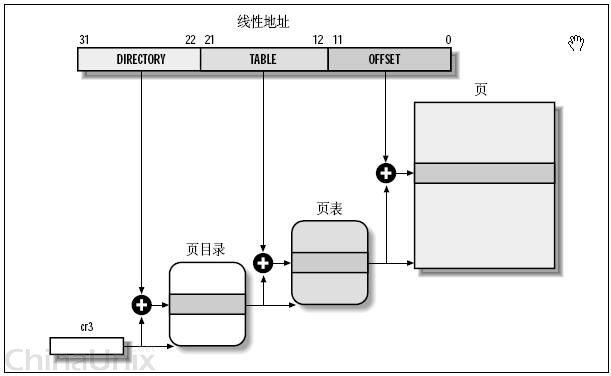


图7-4 二级管理模式图

分页单元中，页目录是唯一的，它的地址放在CPU的cr3寄存器中，是进行地址转换的开始点。

每一个活动的进程，因为都有其独立的对应的虚似内存（页目录也是唯一的），那么它也对应了一个独立的页目录地址。——运行一个进程，需要将它的页目录地址放到cr3寄存器中，将别个的保存下来。

每一个32位的线性地址被划分为三部份，面目录索引(10位)：页表索引(10位)：偏移(12位) 依据以下步骤进行转换：

(1)从cr3中取出进程的页目录地址（操作系统负责在调度进程的时候，把这个地址装入对应寄存器）；

(2)根据线性地址前十位，在数组中，找到对应的索引项，因为引入了二级管理模式，页目录中的项，不再是页的地址，而是一个页表的地址。（又引入了一个数组），页的地址被放到页表中去了。

(3)根据线性地址的中间十位，在页表（也是数组）中找到页的起始地址；

(4)将页的起始地址与线性地址中最后12位相加，得到最终的物理地址。

## 7.4 TLB与四级页表支持下的VA到PA的变换

图7-5给出了Core i7 MMU如何使用四级页表来将虚拟地址翻译成物理地址。36位的虚拟地址被分割成4个9位的片。CR3寄存器包含L1页表的物理地址。VPN1有一个到L1 PTE的偏移量，找到这个PTE以后又会包含到L2页表的基础地址；VPN2包含一个到L2PTE的偏移量，找到这个PTE以后又会包含到L3页表的基础地址；VPN3包含一个到L3PTE的偏移量，找到这个PTE以后又会包含到L4页表的基础地址；VPN4包含一个到L4PTE的偏移量，找到这个PTE以后就是相应的PPN（物理页号）。

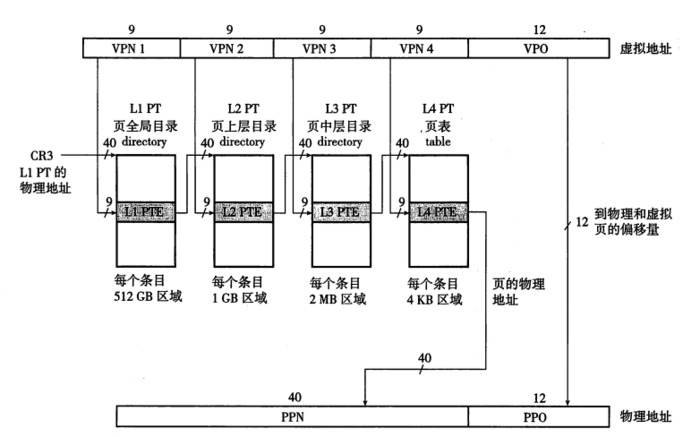


图7-5 Core i7页表翻译

## 7.5 三级Cache支持下的物理内存访问

通过7.4 Core i7 MMU使用四级页表来将虚拟地址翻译成物理地址，我们得到了物理地址PA。现在分析三级cache支持下的物理内存访问。如图7-6，以L1 d-cache的介绍为例，L2和L3同理。

L1 Cache是8路64组相联。块大小为64B。因此CO和CI都是6位，CT是40位。根据物理地址（PA），首先使用CI组索引，每组8路，分别匹配标记CT。如果匹配成功且块的有效位是1，则命中，根据块偏移CO返回数据。

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，向下一级缓存中取出被请求的块，然后将新的块存储在组索引指示的组中的一个高速缓存行中。一般而言，如果映射到的组内有空闲块，则直接放置，否则必须驱逐出一个现存的块，一般采用最近最少被使用策略LRU进行替换。

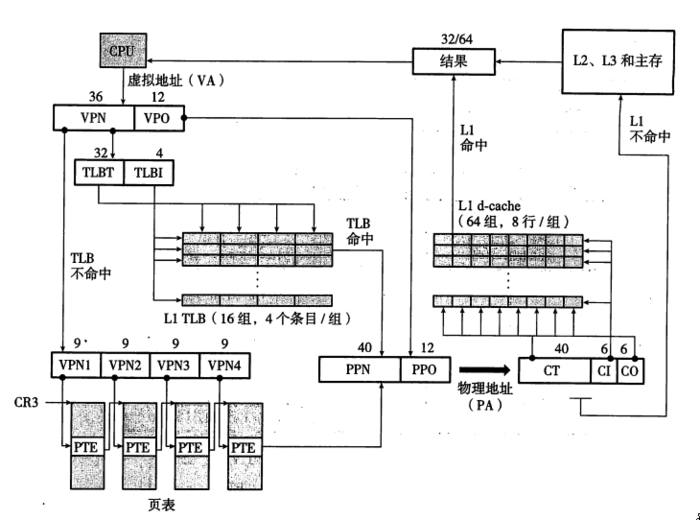


图7-6 Core i7地址翻译的概况（关注右侧）

## 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进程创建各种数据结构，并分配给它一个唯一的PID 。为了给hello进程创建虚拟内存，它创建了hello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进程现在的虚拟内存刚好和调用fork时存在的虚拟内存相同。当这两个进程中的任一个后来进行写操作时，写时复制机制就会创建新页面，因此，也就为每个进程保持了私有地址空间的抽象概念。

## 7.7 hello进程execve时的内存映射

exceve函数加载和执行程序Hello，需要以下几个步骤：

1.删除已存在的用户区域。

2.映射私有区域。为Hello的代码、数据、bss和栈区域创建新的区域结构，所有这些区域都是私有的、写时复制的。

3.映射共享区域。比如Hello程序与标准C库libc.so链接，这些对象都是动态链接到Hello的，然后再用户虚拟地址空间中的共享区域内。

4.设置程序计数器（PC）。exceve做的最后一件事就是设置当前进程的上下文中的程序计数器，使之指向代码区域的入口点。

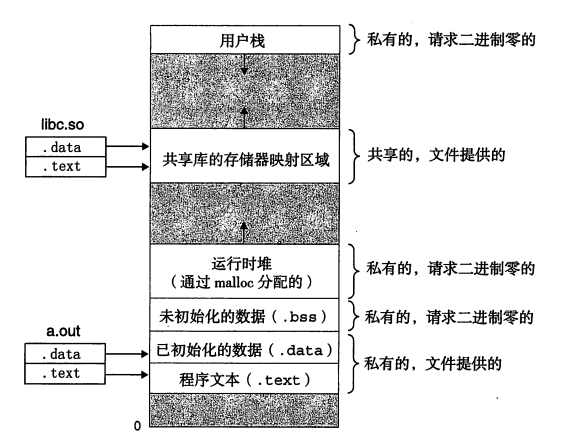


图7-7 加载器是如何映射用户地址空间的区域的

## 7.8 缺页故障与缺页中断处理

在虚拟内存的习惯说法中，DRAM缓存不命中称为缺页(page fault) 。图7-8展示了在缺页之前我们的示例页表的状态。CPU引用了VP3中的一个字，VP3并未缓存在DRAM中。地址翻译硬件从内存中读取PTE3,从有效位推断出VP3未被缓存，并且触发一个缺页异常。缺页异常调用内核中的缺页异常处理程序，该程序会选择一个牺牲页，在此例中就是存放在PP3中的VP4。如果VP4已经被修改了，那么内核就会将它复制回磁盘。无论哪种情况，内核都会修改VP4的页表条目，反映出VP4不再缓存在主存中这一事实。

缺页处理程序从磁盘上用VP3的副本取代VP4，在缺页处理程序重新启动导致缺页的指令之后，该指令将从内存中正常地读取字，而不会再产生异常。

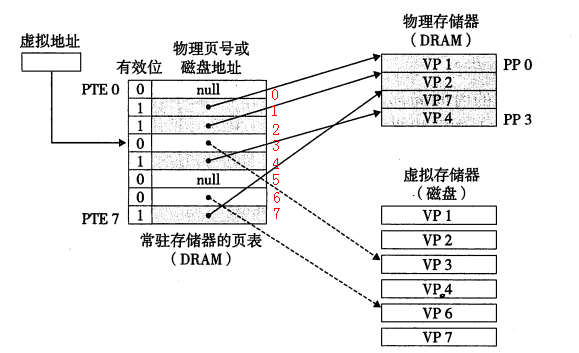
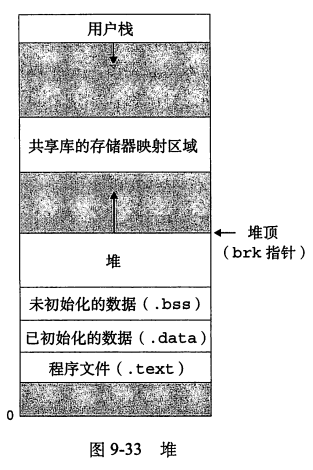


图7-8 VM缺页

## 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(如图7-9)。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。

1.显式分配器：要求应用显式地释放任何已分配的块。例如C程序通过调用malloc函数来分配一个块，通过调用free函数来释放一个块。其中malloc采用的总体策略是：先系统调用sbrk一次，会得到一段较大的并且是连续的空间。进程把系统内核分配给自己的这段空间留着慢慢用。之后调用malloc时就从这段空间中分配，free回收时就再还回来（而不是还给系统内核）。只有当这段空间全部被分配掉时还不够用时，才再次系统调用sbrk。当然，这一次调用sbrk后内核分配给进程的空间和刚才的那块空间一般不会是相邻的。

2.隐式分配器：也叫做垃圾收集器，例如，诸如Lisp、ML、以及Java之类的高级语言就依赖垃圾收集来释放已分配的块。

图7-9 堆

**隐式空闲链表：**

这样的一种结构，主要是由三部分组成：头部、有效载荷、填充（可选）；

头部：是由块大小+标志位（a已分配/f空闲）；有效载荷：实际的数据

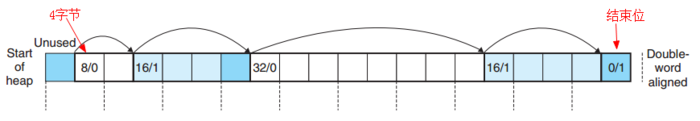


图7-10 隐式空闲链表示例图

**简单的放置策略：**

1> 首次适配：从头搜索，遇到第一个合适的块就停止；

2> 下次适配：从头搜索，遇到下一个合适的块停止；

3> 最佳适配：全部搜索，选择合适的块停止。

**分割空闲块：**

适配到合适的空闲块，分配器将空闲块分割成两个部分，一个是分配块，一个是新的空闲块，如图7-11 。

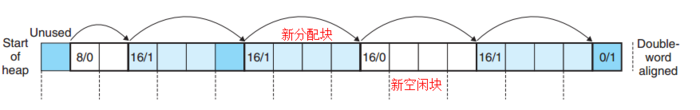


图7-11 分割空闲块

**增加堆的空间：**

通过调用sbrk函数，申请额外的存储器空间，插入到空闲链表中 。

**合并空闲块：**

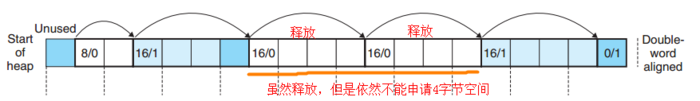


图7-11 假碎片示例

**1. 合并空闲块的目的**

如图7-11，虽然释放了两个3字节大小的数据空间，而且空闲的空间相邻，但是就是无法再分配4字节的空间了，这时候就需要进行一般合并：合并的策略是立即合并和推迟合并，立即合并，可能有不好的地方。如果我们马上合并上图的空间后又申请3字节的块，那么就会开始分割，释放以后立即合并的话，又将是一个合并分割的过程，这样的话推迟合并就有好处了。需要的时候再合并，就不会产生抖动了。

**2. 带边界标记的合并**

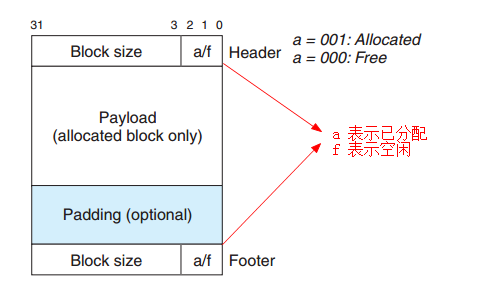


图7-12 使用边界标记的堆块的格式

如图7-12 。Knuth提出了一种边界标记技术，允许在常数时间内进行对前面快的合并。这种思想是在每个块的结尾处添加一个脚部，其中脚部就是头部的一个副本。如果每个块包括这样一个脚部，那么分配器就可以通过检查它的脚部，判断前面一个块的起始位置和状态，这个脚部总是在距当前块开始位置一个字的距离。

**显式空闲链表的基本原理：**

根据定义，程序不需要一个空闲块的主体，所以实现空闲链表数据结构的指针可以存放在这些空闲块的主体里面。

显式空闲链表结构将堆组织成一个双向空闲链表，在每个空闲块的主体中，都包含一个pred（前驱）和succ（后继）指针。

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。不过，释放一个块的时间可以是线性的，也可能是个常数，这取决于空闲链表中块的排序策略。

一种方法是用后进先出（LIFO）的顺序维护链表，将新释放的块放置在链表的开始处。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。

## 7.10本章小结

虚拟内存是对主存的一个抽象。本章通过对虚拟内存的了解，学会了TLB和四级页表支持下VA到PA的转换，以及得到了PA后，三级cache下的物理内存的访问过程。通过本章内容，更深入掌握了fork函数和exceve函数和虚拟内存的种种联系，最后还学会了动态内存分配的管理。

**（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：文件

设备管理：unix io接口

一个linux文件就是一个m个字节的序列：

B0 , B1 , … , Bk , … , Bm-1

所有的I/ O 设备（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作对相应文件的读和写来执行。这种将设备优雅地映射为文件的方式，允许Linux 内核引出一个简单、低级的应用接口，称为Unix I/O，这使得所有的输入和输出都能以一种统一且一致的方式来执行。

## 8.2 简述Unix IO接口及其函数

**Unix IO接口：**

**打开文件**，内核返回一个非负整数的文件描述符，通过对此文件描述符对文件进行所有操作。

**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（文件描述符0）、标准输出（描述符为1），标准出错（描述符为2）。头文件<unistd.h>定义了常量STDIN\_FILENO、STDOUT\_FILENO、STDERR\_FILENO，他们可用来代替显式的描述符值。

**改变当前的文件位置**，文件开始位置为文件偏移量，应用程序通过seek操作，可设置文件的当前位置为k。

**读写文件**，读操作：从文件复制n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；写操作：从内存复制n个字节到文件，当前文件位置为k，然后更新k

**关闭文件**。当应用完成对文件的访问后，通知内核关闭这个文件。内核会释放文件打开时创建的数据结构，将描述符恢复到描述符池中

Unix IO函数：

**1. open（）函数**

功能描述：用于打开或创建文件，在打开或创建文件时可以指定文件的属性及用户的权限等各种参数。

函数原型:int open(const char \*pathname,int flags,int perms)

参数：pathname:被打开的文件名（可包括路径名如"dev/ttyS0"）flags:文件打开方式,

返回值：成功：返回文件描述符；失败：返回-1

**2. close（）函数**

功能描述：用于关闭一个被打开的的文件

所需头文件： #include <unistd.h>

函数原型:int close(int fd)

参数：fd文件描述符

函数返回值：0成功，-1出错

**3. read（）函数**

功能描述： 从文件读取数据。

所需头文件： #include <unistd.h>

函数原型：ssize\_t read(int fd, void \*buf, size\_t count);

参数：fd：将要读取数据的文件描述词。buf：指缓冲区，即读取的数据会被放到这个缓冲区中去。count： 表示调用一次read操作，应该读多少数量的字符。

返回值：返回所读取的字节数；0（读到EOF）；-1（出错）。

**4. write（）函数**

功能描述： 向文件写入数据。

所需头文件： #include <unistd.h>

函数原型：ssize\_t write(int fd, void \*buf, size\_t count);

返回值：写入文件的字节数（成功）；-1（出错）

**5. lseek（）函数**

功能描述： 用于在指定的文件描述符中将将文件指针定位到相应位置。

所需头文件：#include <unistd.h>，#include <sys/types.h>

函数原型：off\_t lseek(int fd, off\_t offset,int whence);

参数：fd;文件描述符。offset:偏移量，每一个读写操作所需要移动的距离，单位是字节，可正可负（向前移，向后移）

返回值：成功：返回当前位移；失败：返回-1

## 8.3 printf的实现分析

首先来看看printf函数的函数体。 调用printf函数的时候，先是最右边的参数入栈。fmt是一个指针，这个指针指向第一个const参数（const char \*fmt)中的第一个元素。fmt也是个变量，它的位置，是在栈上分配的，它也有地址。  
 int printf(const char \*fmt, ...)   
 {  
 int I;  
 va\_list arg = (va\_list)((char\*)(&fmt) + 4);  
 i = vsprintf(buf, fmt, arg);  
 write(buf, i);  
 return i;  
 }

printf函数主要调用了vsprintf和write函数。

下面首先介绍vsprintf(buf, fmt, arg)是什么函数。

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 != '%') {

\*p++ = \*fmt;

continue;

}

fmt++;

switch (\*fmt) {

case 'x':

itoa(tmp, \*((int\*)p\_next\_arg));

strcpy(p, tmp);

p\_next\_arg += 4;

p += strlen(tmp);

break;

case 's':

break;

default:

break;

}

}

return (p - buf);

}

从上面vsprintf函数可以看出，这个函数的作用是将所有的参数内容格式化之后存入buf，然后返回格式化数组的长度。

让我们追踪下write吧：

write:

mov eax, \_NR\_write

mov ebx, [esp + 4]

mov ecx, [esp + 8]

int INT\_VECTOR\_SYS\_CALL

一个int INT\_VECTOR\_SYS\_CALL表示要通过系统来调用sys\_call这个函数。

sys\_call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

于是可以直到printf函数执行过程如下：

从vsprintf生成显示信息，到write系统函数，到陷阱-系统调用 int 0x80或syscall.

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函数内容如图8-1：

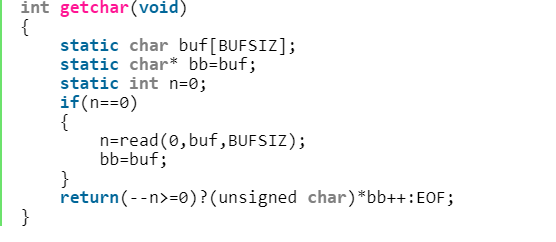


图8-1 getchar函数内容

getchar由宏实现：#define getchar() getc(stdin)。getchar有一个int型的返回值。当程序调用getchar时.程序就等着用户按键。用户输入的[字符](https://baike.baidu.com/item/%E5%AD%97%E7%AC%A6)被存放在键盘[缓冲区](https://baike.baidu.com/item/%E7%BC%93%E5%86%B2%E5%8C%BA)中。直到用户按回车为止（回车字符也放在缓冲区中）。当用户键入回车之后，getchar才开始从stdio流中每次读入一个字符。getchar函数的返回值是用户输入的字符的ASCII码，若文件结尾(End-Of-File)则返回-1(EOF)，且将用户输入的字符[回显](https://baike.baidu.com/item/%E5%9B%9E%E6%98%BE)到屏幕。如用户在按回车之前输入了不止一个字符，其他字符会保留在键盘缓存区中，等待后续getchar调用读取。也就是说，后续的getchar调用不会等待用户按键，而直接读取缓冲区中的字符，直到缓冲区中的字符读完后，才等待用户按键。

异步异常-键盘中断的处理：键盘中断处理子程序。接受按键扫描码转成ascii码，保存到系统的键盘缓冲区。

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

本章学会了linux下IO设备的管理方法，了解了Unix IO和Unix IO函数，深入分析了printf函数和getchar函数的实现。

**（第8章1分）**

# 结论

hello所经历的过程：

1.hello被IO设备编写，以文件的方式储存在主存中。

2.hello.c被预处理hello.i文件

3.hello.i被编译为hello.s汇编文件

4.hello.s被汇编成可重定位目标文件hello.o

5.链接器将hello.o和外部文件链接成可执行文件hello

6.在shell输入命令后，通过exceve加载并运行hello

7.在一个时间片中，hello有自己的CPU资源，顺序执行逻辑控制流

8.hello的VA通过TLB和页表翻译为PA

9.三级cache 支持下的hello物理地址访问

10.hello在运行过程中会有异常和信号等

11.printf会调用malloc通过动态内存分配器申请堆中的内存

12.shell父进程回收hello子进程，内核删除为hello创建的所有数据结构

深切感悟：通过对计算机系统的深入理解，我在编写代码的时候逐渐从计算机的底层考虑问题，思考自己该怎么优化程序的性能，编写对编译器有好的代码。

创新理念：我个人认为计算机系统应该向松散耦合的方向发展，每个功能部件都是一个功能主体，在主控计算机的有限控制下完成。由此，嵌入式系统应被更多的采用。而且随着技术的发展，计算机的主控系统也会越来越便宜，依赖机械系统的部分越来越少。

**（结论0分，缺少 -1分，根据内容酌情加分）**

# 附件

hello.c ：hello源代码

hello.i ：预处理后的文本文件

hello.s ：hello.i编译后的汇编文件

hello.o ：hello.s汇编后的可重定位目标文件

hello\_objdump ：hello的反汇编代码

hello.0\_objdump ：hello.o的反汇编代码

hello ：链接后的可执行文件

**（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] 深入理解计算机系统 Randal E.Bryant David R.O’Hallaron 机械工业出版社

[2] 博客园 printf函数实现的深入剖析

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

[3] 维基百科 virtua memory

<https://en.wikipedia.org/wiki/Virtual_memory>

[4] 百度百科 getchar计算机语言函数

<https://baike.baidu.com/item/getchar/919709?fr=aladdin>

[5] CSDN博客 gcc详解

<https://blog.csdn.net/qq_35144795/article/details/77580913>

[6] 博客园 内存管理

<https://www.cnblogs.com/xavierlee/p/6400230.html>

**（参考文献0分，确实 -1分）**