

**计算机系统**

**大作业**

题 目 程序人生-Hello’s P2P

专 业 计算机系

学　　 号 1170300703

班　　 级 1703007

学 生 兰鸿兵

指 导 教 师 郑贵滨

**计算机科学与技术学院**

**2018年12月**

**摘 要**

本文运用计算机系统知识，通过各类工具和辅助输出信息，分析hello程序在Linux系统下的P2P和020过程，对hello程序的整个生命周期进行了解析，对计算机系统这门课程内容有了更深刻的认识。

**关键词：**P2P；020；操作系统；进程；内存分配；系统IO；

**（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过程：From Program to Process，是将hello.c的文本代码，经过预处理->编译->汇编->链接四个步骤，生成一个二进制可执行文件hello，通过Bash为其fork一个子进程，并最终使用execve载入进程执行。

020过程：From Zero-0 to Zero-0，shell执行，进程管理OS为hello分配虚拟内存并映射，在开始执行进程时分配载入物理内存，执行hello程序。再进行取值，译码，访存，计算，写回，更新pc等操作，流水线执行hello的每条指令。将其输出内容显示到屏幕，然后hello进程结束，shell回收其内存。

## 1.2 环境与工具

硬件环境：X64CPU; 4.5GHz; 8GRAM; 1TBHD Disk

软件环境：Windows10 64位；VMware14.12; Ubuntu 18.04 LTS 64位;WPS

使用工具：codeblocks, Visual Studio 2017, objdump, gdb, edb, hexedit

## 1.3 中间结果

hello.i hello.c预处理后的程序文本

hello.s hello.i编译成汇编语言后的程序文本

hello.o hello.s生成的二进制文件

hello.objdump hello.o反汇编后的文件

hello hello通过链接操作后生成的二进制可执行文件

hello\_back.txt hello反汇编后重定向生成的程序文本

helloelf.txt hello.o的readelf输出的重定向文本

helloelf\_second.txt hello的readelf输出的重定向文本

## 1.4 本章小结

本章简单介绍了P2P与020的过程，并列出实验中所使用的环境与工具，以及整个过程产生的中间结果。

**（第1章0.5分）**

# **第2章 预处理**

## 2.1 预处理的概念与作用

概念：预处理是在编译之前进行的处理。C语言的预处理主要有三个方面的内容： 1.宏定义； 2.文件包含； 3.条件编译。 预处理命令以符号“#”开头。具体操作是在程序编译之前，根据这些以符号“#”开头的命令，修改原始的c程序。

例如大作业的hello.c文件，就有以下命令：

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

预处理要做的是从系统头文件包中找到这三个头文件，把需要的内容插入hello.c的文本中，生成hello.i文件。

作用：预处理将一些宏进行文本替换，将需要的代码写入文件，成为一份完整能够进行编译的代码，方便编译器对程序进行编译。

## 2.2在Ubuntu下预处理的命令

命令：gcc -E -o name.i name.c

截图：

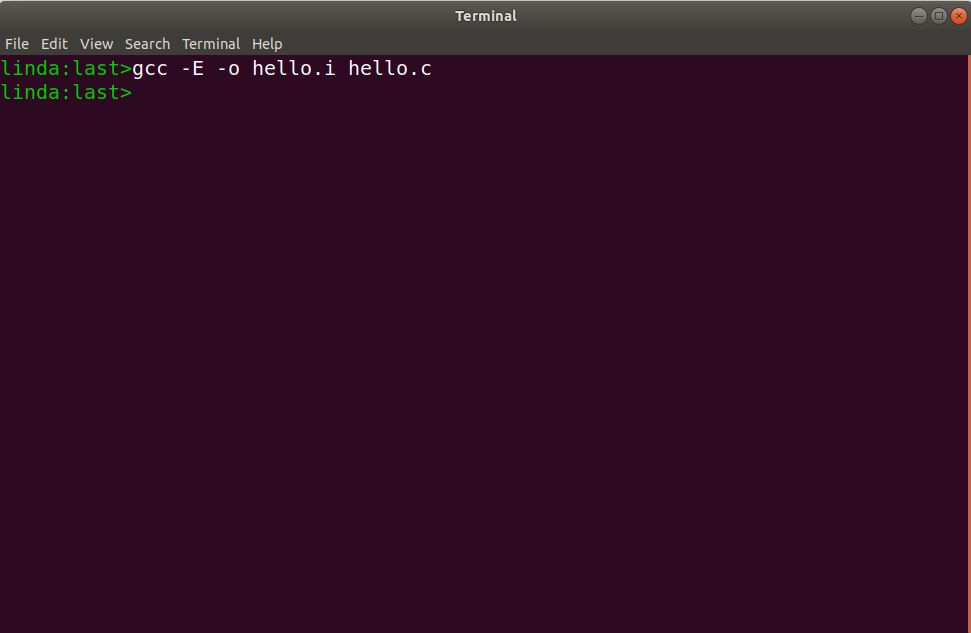


图2.2.1

## 2.3 Hello的预处理结果解析

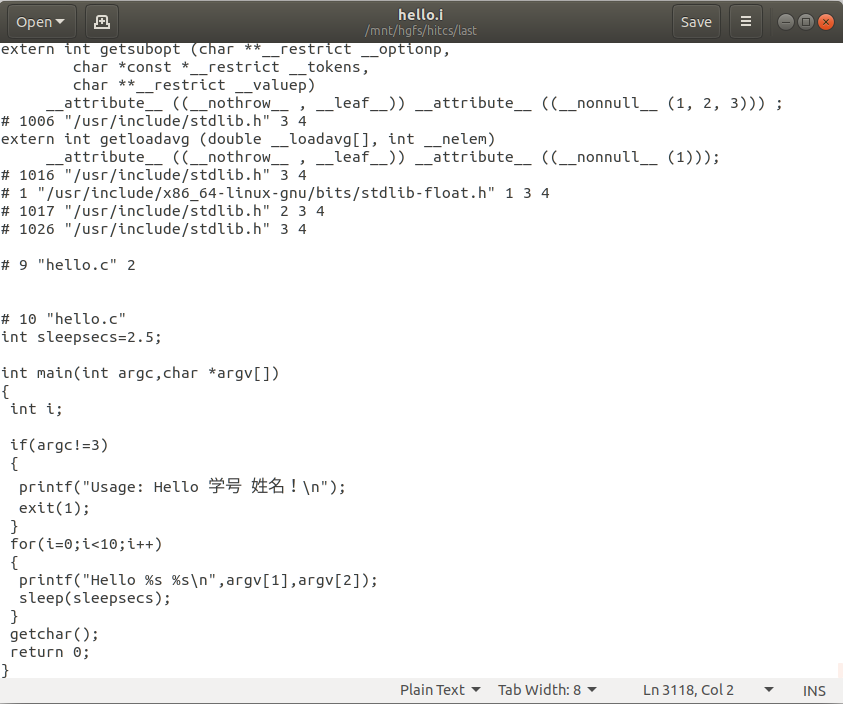


图2.3.1

预处理后得到的hello.i如上。可以看到，代码依然是c语言的代码，main函数的内容没有发生变化，但行数却增加到了3118行，这是因为预编译将#include的头文件中的代码展开插入到了hello.c中。并添加了许多注释，用来描述使用的这些代码所在计算机中的位置。

预编译所得到的代码就是完整的可以进行编译的代码，能够进行下一步编译。

## 2.4 本章小结

本章介绍了预编译的概念与作用，简单介绍了其过程，对预编译过程进行演示，并根据演示结果进行了结果分析，验证其过程与作用。

**（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，是将高级语言文本程序翻译成相应汇编语言文本程序的过程。

作用：编译将机器所不能读懂的高级语言文本翻译成更加接近机器语言的汇编文本程序，方便将程序最终翻译成二进制可执行文件。

## 3.2 在Ubuntu下编译的命令

命令：gcc -S -o name.s name.i

截图：

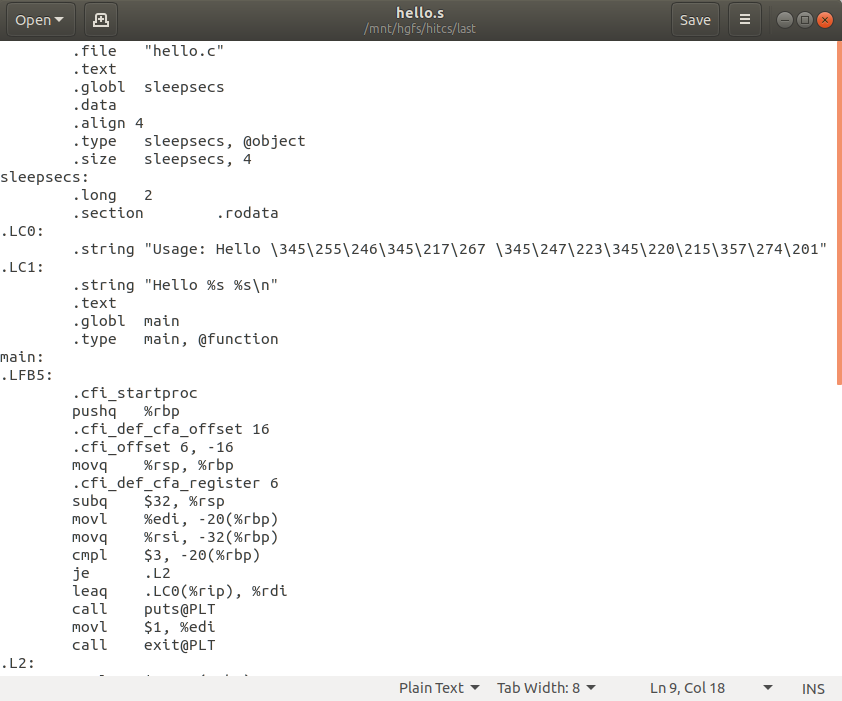
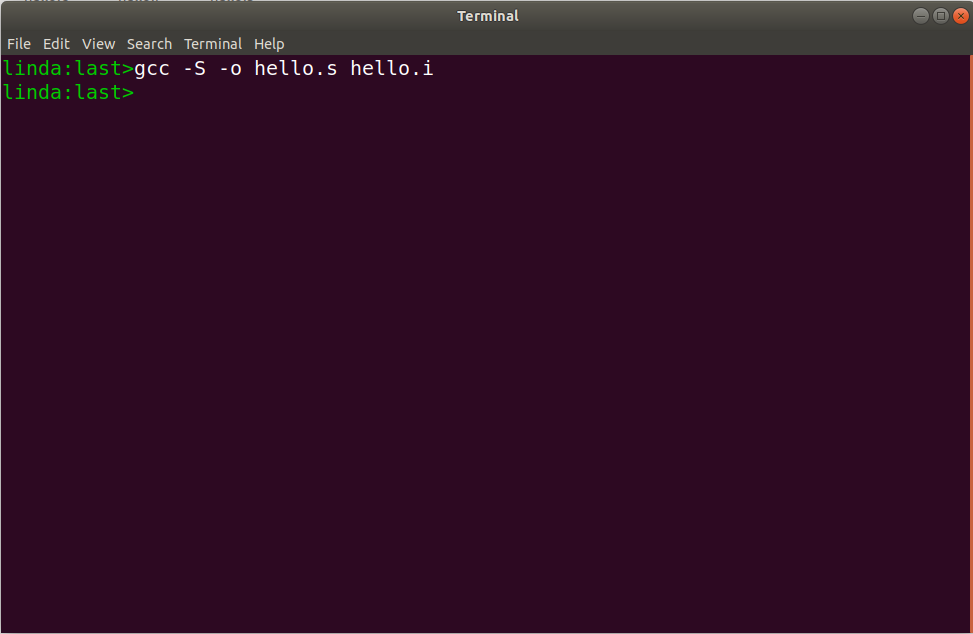


图3.3.1

## 3.3 Hello的编译结果解析

编译所得到的.s文件其长度只有65行，所进行的工作是将main()函数进行了翻译，变成汇编语言。

3.3.1 变量处理

全局变量：

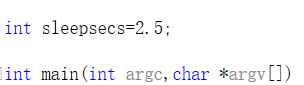


图3.3.2

在我们的C语言源程序中，有以上全局变量sleepsecs。

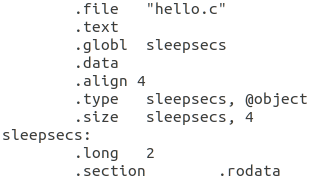


图3.3.3

在编译后的.s文件中，我们找到其相应的代码段。其被称为sleepsecs，存在.data段。并记录其相关信息，类型为object，长度为4个字节。并且因为源程序中是int类型，赋值为2.5发生了隐式转换，最终其值为2。

局部变量：

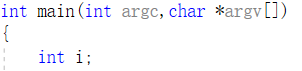


图3.3.4

再来看局部变量，下图中可以看到，在c语言源程序的main函数中有一个局部int类型变量i，用来对循环进行计次。

我们来寻找其在汇编代码中的具体位置：

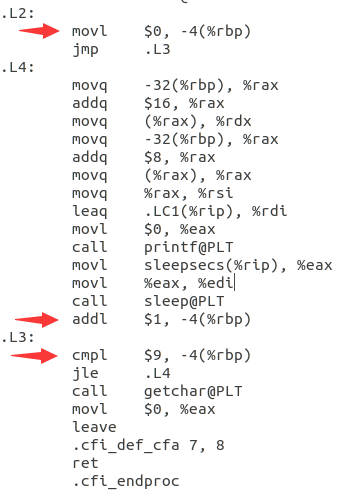


图3.3.4

图中箭头所指的操作即是对局部变量i的操作。可以看到，在汇编代码中，局部变量i只是被存储在了内存空间 -4(%rbp) 中，在使用到它时访问这个内存位置即可。

常量：

接着来说常量，在图3.3.4中，几个对内存 -4(%rbp) 的操作都使用到了常量。在汇编代码中，常量即立即数，其以$开头，后跟上具体的数值，可以在指令中直接使用。

数组：

在图3.3.4中可以看到，main函数在传参时传入了一个argv[]数组。

对于数组的访问，是通过首地址加偏移量的方式进行的。



图3.3.5

如图是汇编代码中对argv的访问，首先将首地址传给 %rax ，之后偏移16个字节，就能够访问到我们所需要的位置。

字符串：

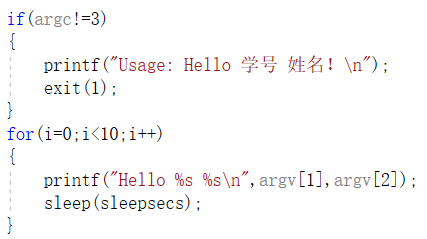


图3.3.6

在c语言源程序中，有以上两段字符串。同样的我们找到它们在汇编代码中的位置：

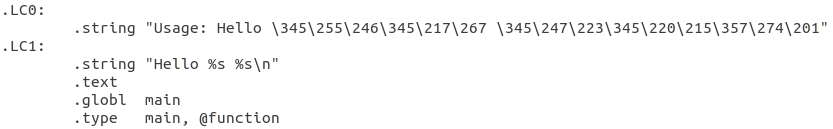


图3.3.7

可以看到，字符串被存储在了.rodata的只读存储段中，在使用的时候，访问其存储位置首地址即可。

3.3.2 操作处理

赋值与四则运算操作：

在图3.3.6中，我们可以看到，局部变量i在循环过程中进行了赋值和运算操作。循环开始时赋值为0，之后逐次加1。

再看回图3.3.4，我们找到这些操作在汇编代码中的位置。从图3.3.4中可以看到，代码 movl $0, -4(%rbp) 是对i的赋值操作，将立即数0赋值给 -4(%rbp) 内存。代码 addl $1, -4(%rbp) 是对i的加法操作，将 -4(%rbp) 内存加1。

对于关系操作符与控制语句的处理：

图3.3.6中，有 if(argc!=3) 这一行代码，其中 != 就是关系操作符。

在汇编代码中没有这样的关系操作符，取而代之的是如下操作：

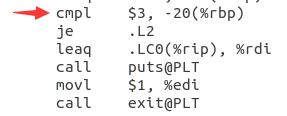


图3.3.8

从cmpl操作开始分析，cmpl是比较的操作，判断3与 -20(%rbp) 是否相等，不难看出 -20(%rbp) 就是argc。cmpl判断是否相等，设置条件码后执行下一句代码，je是跳转操作，根据条件码来决定是否跳转到后面的地址。此处是若argc等于3，就跳过if语句中的操作，直接执行之后的代码。

用cmpl和je的结合表示关系操作，这样就形成了控制语句。

3.3.3 函数操作

函数调用：

在汇编代码中，函数调用使用call指令，通过call指令来访问所跟的相应函数代码。如图，是调用输出函数。

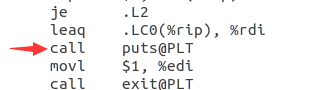


图3.3.9

函数传参：

汇编代码中函数传参，需要先找几个寄存器，将参数传给这些寄存器。

x86-64 Linux平台下，约定前6个参数存到寄存器rdi, rsi, rdx, rcx, r8, r9中，其余参数按照传参顺序依次压入栈中。

如图是printf函数的传参过程：

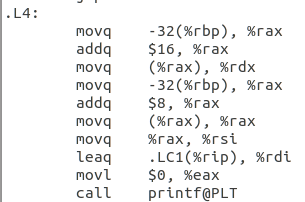


图3.3.10

在调用printf函数之前，先进行了传参，将参数传给了rdi, rsi, rdx，以供printf函数使用。

函数返回值：

函数返回值一般存在寄存器eax中，如果要设定返回值的话，那就先将返回值传入eax，再用ret语句返回。

如图是main函数的返回：

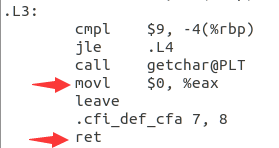


图3.3.11

图中，先对eax赋值，最后使用ret返回。

## 3.4 本章小结

本章主要介绍了编译这一过程。对编译的概念与作用进行了简述，并根据hello.c从C语言源码编译到机器语言的案例，进行了各部分的过程解析。具体讲述了从高级语言逐渐接近机器语言的这一过程。

**（第3章2分）**

# 第4章 汇编

## 4.1 汇编的概念与作用

概念：汇编，是指将汇编语言翻译成机器指令，并将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 类型文件中。

作用：通过汇编过程，汇编语言被翻译成一条条机器可以直接读取分析的机器指令。

## 4.2 在Ubuntu下汇编的命令

命令：as name.s -o name.o

截图：

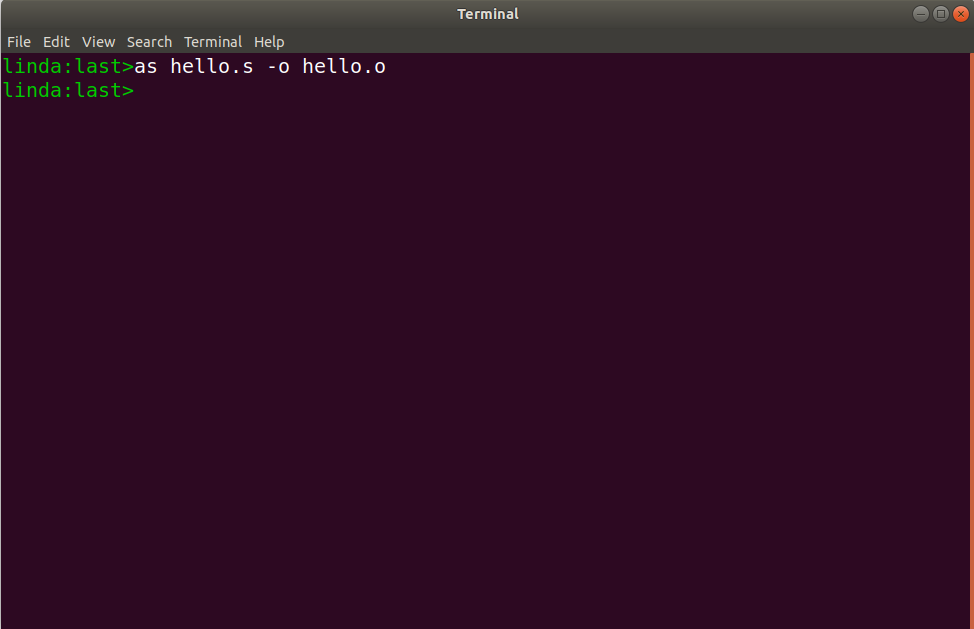


图4.2.1

## 4.3 可重定位目标elf格式

先使用指令 readelf -a hello.o > helloelf.txt 读取main.o的elf文件并重定向输出到helloelf.txt中，方便阅读与分析。

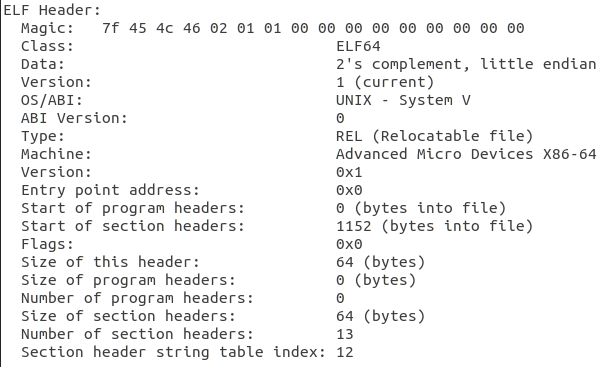


图4.3.1

ELF Header : 用于总的描述ELF文件各个信息的段，包含生成该文件的系统字的大小和字节顺序。

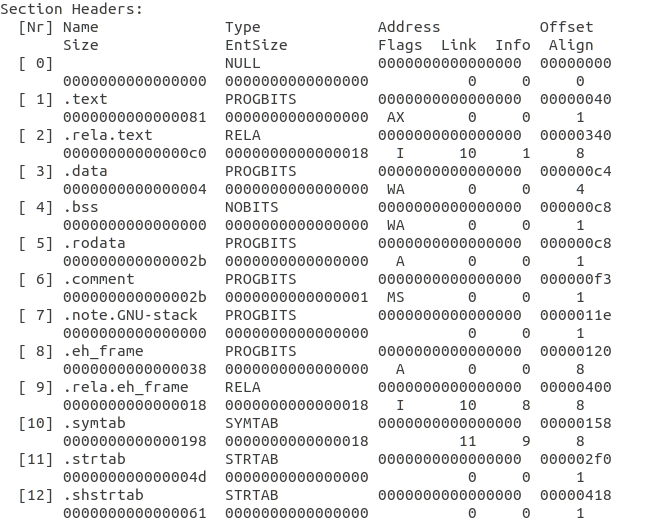
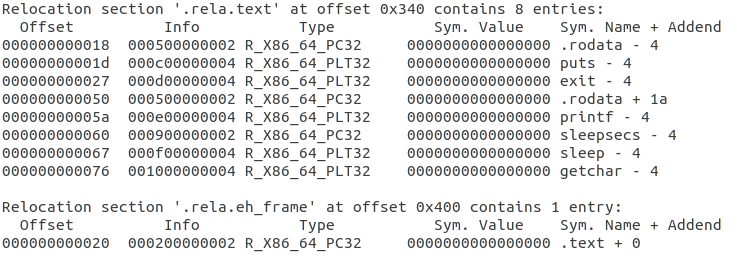


图4.3.2

Section Header : 描述.o文件中出现的各个节的类型、位置、所占空间大小等信息。

图4.3.3



.rela.text : 重定位节，这个节包含了.text节中需要进行重定位的信息——偏移量（offset）,信息（info），类型（type），符号值（sym.value）。这些信息描述的位置，在由.o文件生成可执行文件的时候需要被重定位。在hello.o里需要被重定位的有printf , puts , exit , sleepsecs , getchar , sleep 以及 rodata里的两个字符串。

## 4.4 Hello.o的结果解析

使用 objdump -d -r hello.o > hello.objdump 获得反汇编代码。将文件hello.s与文件hello.objdump进行比较。

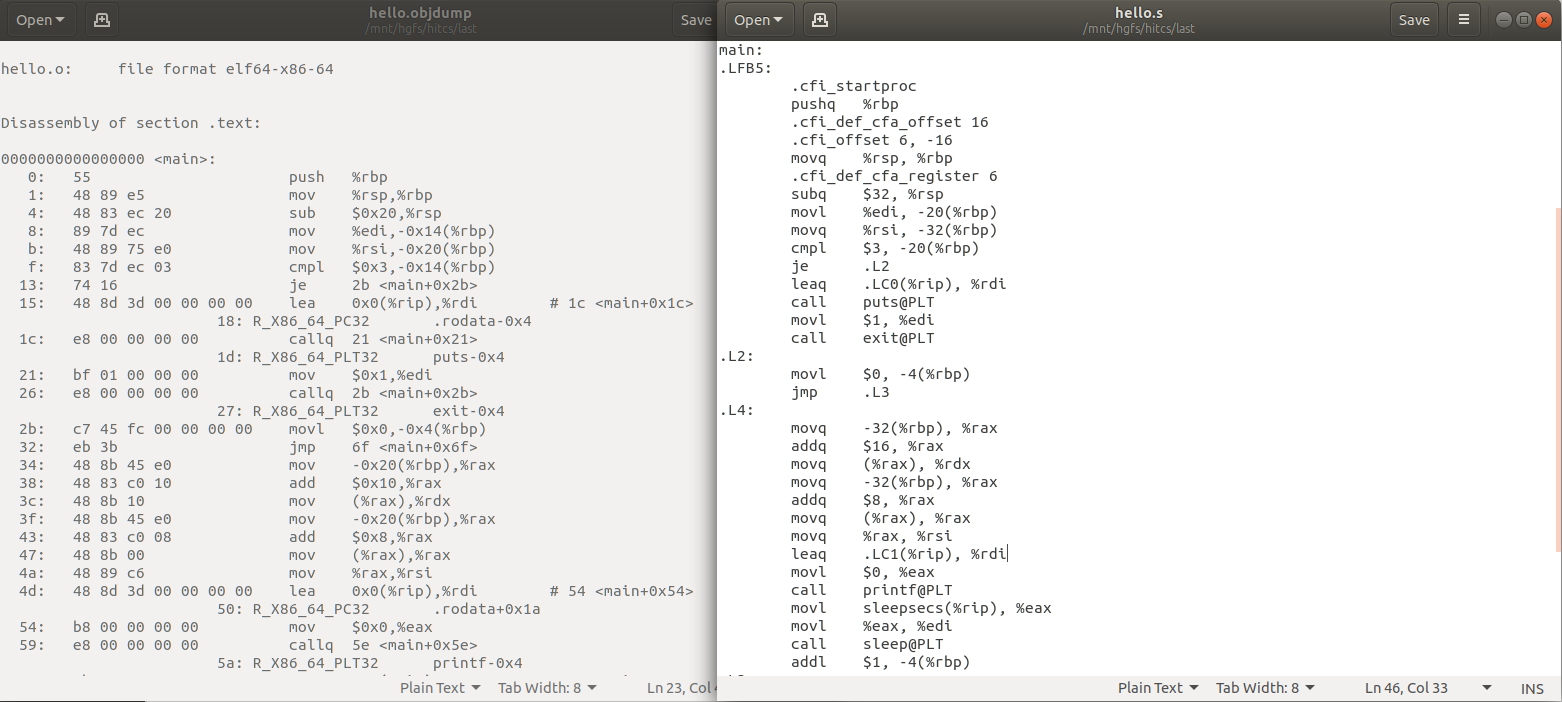


图4.4.1

对比可以看出，hello.o反汇编之后，与原来的汇编代码相比，左测多了机器指令，在冒号前面的是运行时机器指令的位置，冒号后的是每一行汇编语句所对应的机器指令。机器指令完全由0/1构成，此处显示为16进制。

分支跳转语句：我们可以发现，在hello.s中跳转是用.L3/.L4等助记符来表示的，而hello.o反汇编之后，这些助记符消失了，改用具体的地址位置代替。

函数调用：在hello.s中，调用一个函数使用的是call+函数名，但在hello.o反汇编结果中，call后所跟的是一个具体的地址位置。

## 4.5 本章小结

本章主要介绍了汇编这一过程，简介了汇编的概念和作用，并通过readelf查看hello.o的ELF，反汇编查看hello.o反汇编内容并比较其与hello.s之间差别，这两个过程，分析了汇编指令映射到机器指令的这一过程。

**（第4章1分）**

# 第5章 链接

## 5.1 链接的概念与作用

概念：链接是将多个代码段和数据片段收集并拼接合并成一个可执行文件的过程。

作用：由于存在链接这一过程，可以分段编写程序，最后再进行链接，成为一个完整的程序，降低了模块化编程的难度。

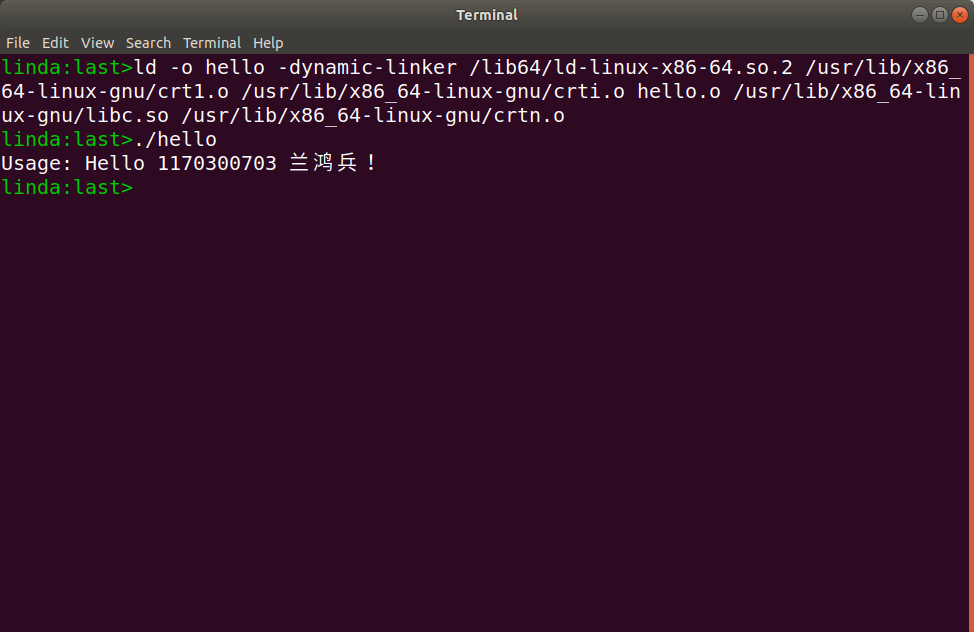
## 5.2 在Ubuntu下链接的命令

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

截图：

图5.2.1



## 5.3 可执行目标文件hello的格式

使用readelf -a hello > helloelf\_second.txt命令输出ELF内容。

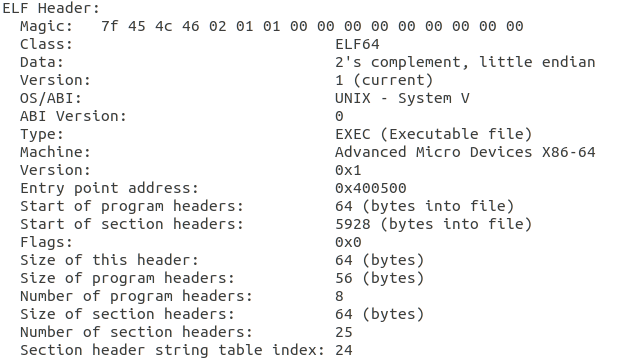


图5.3.1

以上是ELF Header的信息，可以看出节头表的数量增加了。

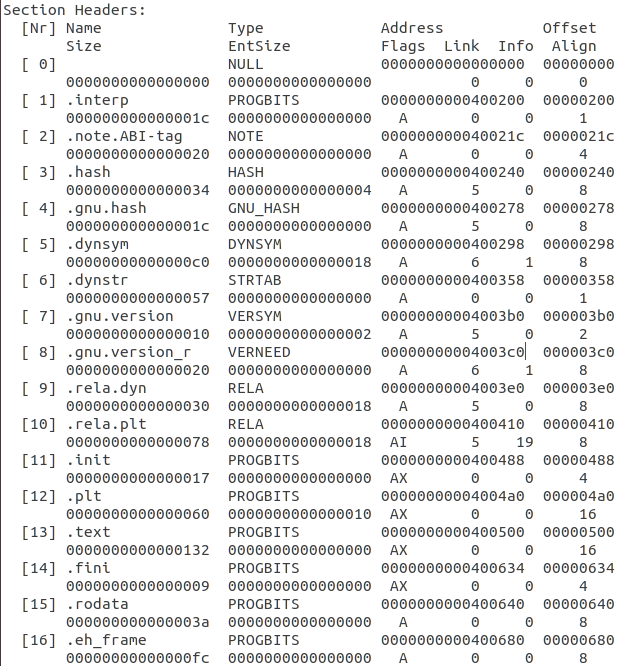
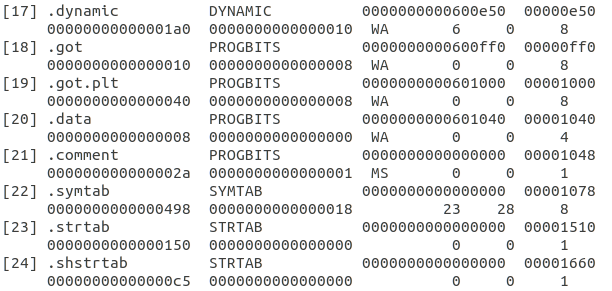


图5.3.2

图5.3.3



以上是ELF节头表的内容，包含各段类型、位置、所占空间大小等信息

## 5.4 hello的虚拟地址空间

使用edb加载hello，在Data Dump中可以看到如下虚拟地址信息。

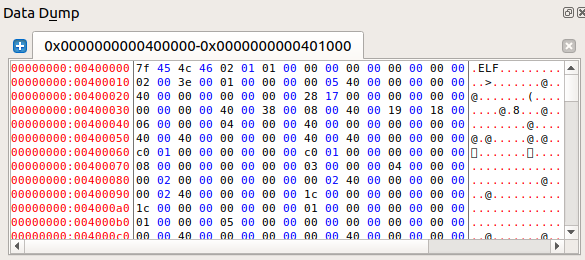
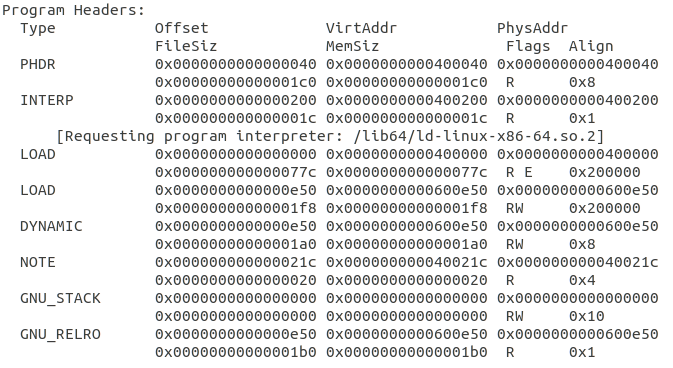


图5.4.1

看出程序在0x00400000地址开始加载，在0x00400fff结束。

再看ELF中的程序头表：

图5.4.2



各信息如下：

PHDR：程序头表

INTERP：程序执行前需要调用的解释器

LOAD：程序目标代码和常量信息

DYNAMIC：动态链接器所使用的信息

NOTE:：辅助信息

GNU\_EH\_FRAME：保存异常信息

GNU\_STACK：使用系统栈所需要的权限信息

GNU\_RELRO：保存在重定位之后只读信息的位置

## 5.5 链接的重定位过程分析

先使用命令objdump -d -r hello > hello\_back.txt，将hello的反汇编的代码输出到hello\_back.txt中。

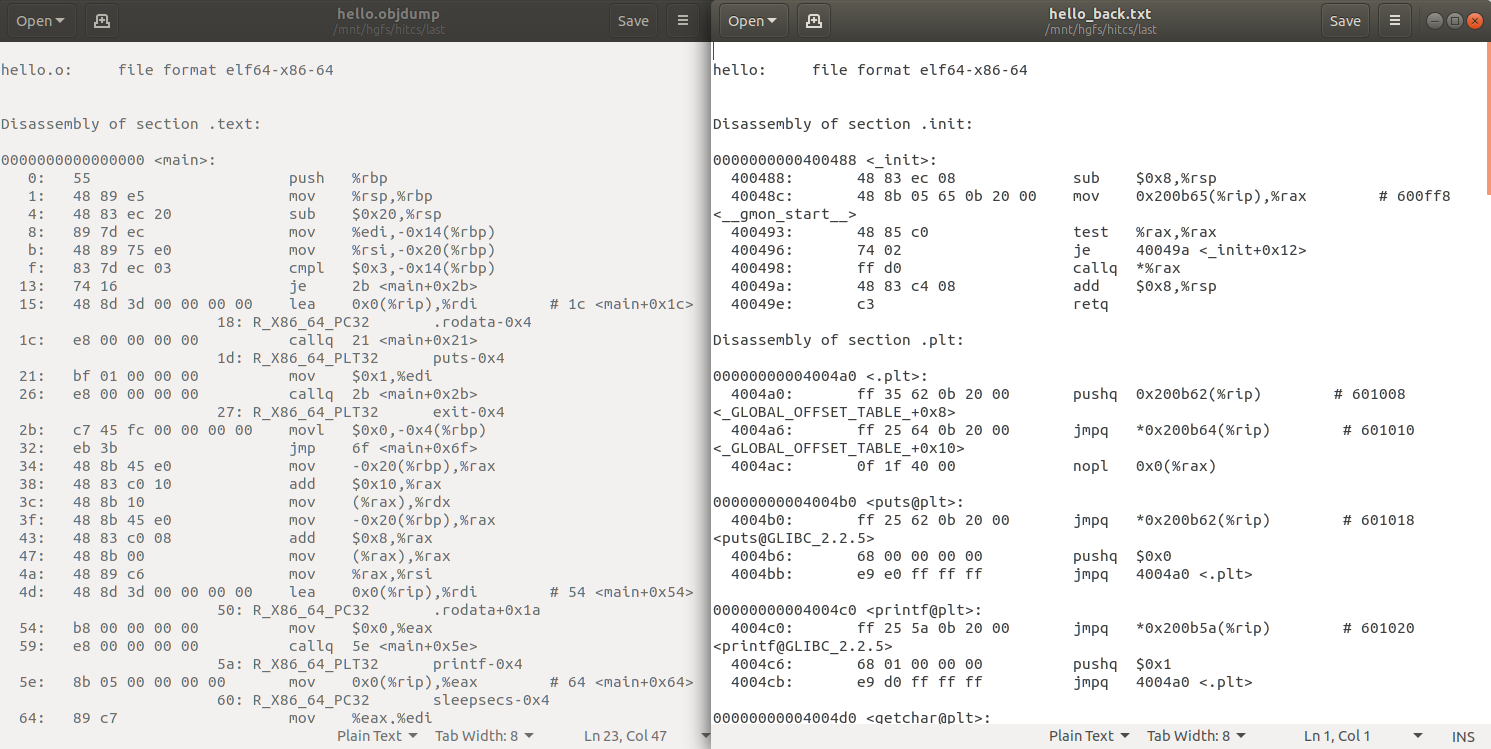


图5.5.1

与hello.o的反汇编代码进行比较可以发现，hello中是从hello.elf开始，而hello.o是从.text节开始；另外hello的反汇编比hello.o的反汇编要多出许多函数，在hello.s中导入了诸如puts、printf、getchar、sleep等在hello程序中使用的函数。

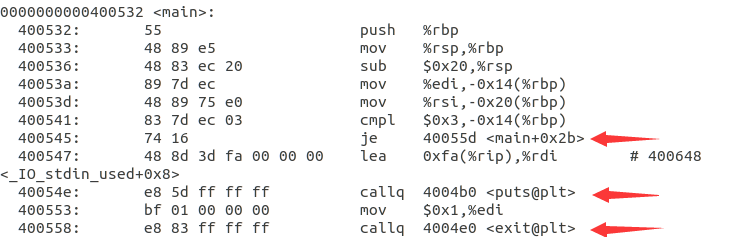


图5.5.2

再看main函数，不难看出，链接调用函数的方式的是call+<main+偏移量>。

可以得出重定位过程如下：链接器完成地址和空间的分配后，可以根据表中的内容确定所有符号的虚拟地址。根据这些符号地址，就能够对需要重定位的符号进行修正。而在elf文件中有一个重定位表，其中存储了需要重定位的位置，描述了如何修改相应的段中的内容。重定位表是一个结构体数组，每个数组元素对应一个重定位入口。

## 5.6 hello的执行流程

hello的执行流程如下：

\_dl\_start 0x00007fff6f0674a0)

0x00007f0625d5e630 <ld-2.27.so!\_dl\_init+0>

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

\*hello!printf@plt

\*hello!sleep@plt

\*hello!getchar@plt

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

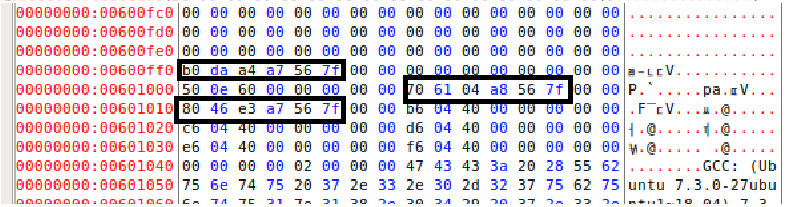
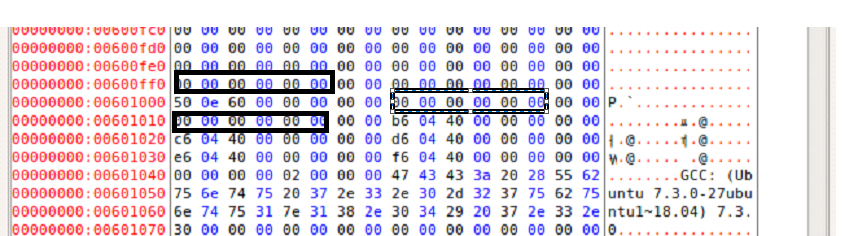


图5.7.1

前后对比我们发现原先global\_offset表是全0的，但在执行\_dl\_init后被赋上了相应的偏移量值。这表明\_dl\_init操作是给程序赋上当前执行内存地址的偏移量，以初始化hello程序。

## 5.8 本章小结

本章主要介绍了链接这一过程，简介了链接的概念和作用，分析了hello的ELF格式、虚拟地址空间分配、重定向过程、执行流程和动态链接过程。

**（第5章1分）**

# 第6章 hello进程管理

## 6.1 进程的概念与作用

概念：进程是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。

作用：程序是指令、数据及其组织形式的描述，进程是程序的实体。

## 6.2 简述壳Shell-bash的作用与处理流程

shell是linux下的一个应用程序，提供用户直接和操作系统内核进行交互的界面，其处理流程如下：

1、读取用户的输入。

2、切分命令以获得用户输入的相关参数。

3、判断是否是内置命令。

4、如果是内置命令，则直接执行。

5、否则使用fork和execve创建一个进程并替换。

6、程序运行期间，shell需要监视键盘的输入，并且做出相应反应。

## 6.3 Hello的fork进程创建过程

1、用户输入命令 ./hello

2、判断非内置命令

3、使用fork函数创建一个子进程。子进程得到与父进程相同的虚拟地址空间，拥有与父进程相同参数等信息和与父进程相同的副本。区别在于子进程和父进程的PID不同。

4、系统并发运行，内核交替执行父子进程的逻辑控制流。

## 6.4 Hello的execve过程

子进程调用execve函数，将子进程内容完全替换为hello可执行文件执行。

具体操作是将子进程的虚拟内存段全部替换为hello的虚拟内存段，不改变pid的前提下实现内容的替换，再令PC指向hello程序开始位置，在hello时间片内执行指令。execve 调用一次从不返回。

## 6.5 Hello的进程执行

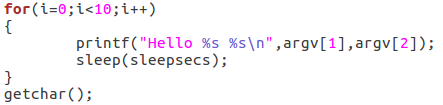


图6.5.1

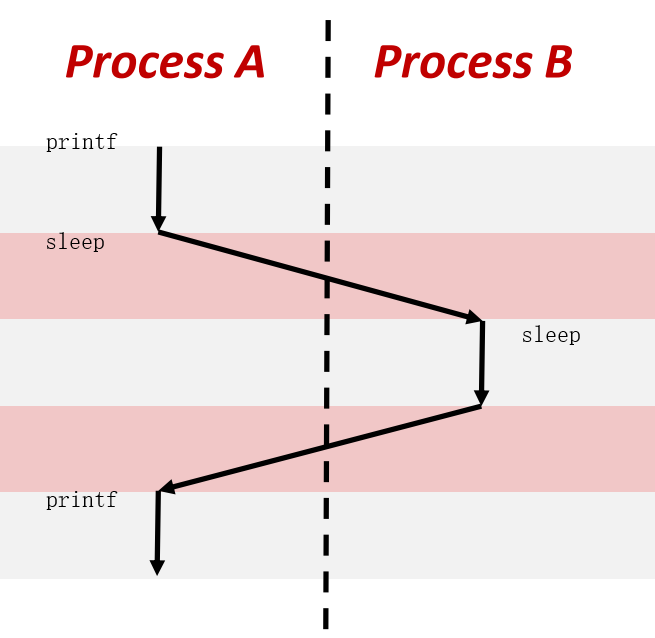


图6.5.2

图6.5.2是6.5.1中hello的C语言代码片段的执行过程样例，操作系统在进程A到进程B之间切换，进程A是用户模式，进程A中断后，操作系统会保存进程A的信息，之后进入内核模式，由进程A转到进程B，在进程B执行之后引发中断，再转回进程A。

另外，调用getchar()时，会先在前端hello进程中运行，调用时再切换到内核进程标准读取程序中，从键盘输入中读取一个字符后再回到前端hello进程。

像这样在内核和前端之前切换的操作，称为上下文切换。

## 6.6 hello的异常与信号处理

1、Ctrl-Z



图6.6.1

Ctrl-Z是将程序暂时挂起。

由图可以看出，Ctrl-Z操作向进程发送sigtstp信号，让进程暂时挂起，输入ps指令后可以看到hello进程并没有关闭。

2、Ctrl-C



图6.6.2

Ctrl-C是将程序关闭。

由图可以看出，Ctrl-C操作向进程发送sigint信号，让进程直接结束。输入ps指令后可以看到hello进程已经被关闭。

3、不停乱按，包括回车

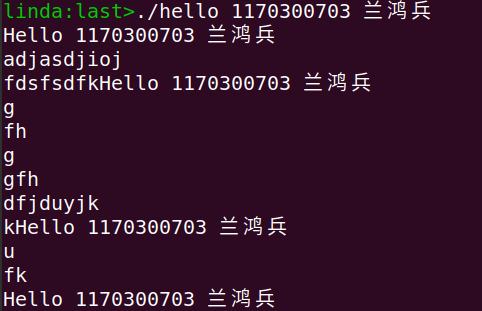


图6.6.3

如图，不停乱按后键盘读入的字符都会到缓存区中，对当前执行的程序没有影响。当按下回车后，会对getchar()产生影响。

4、fg命令



图6.6.4

fg命令可以让挂起的程序继续执行。

由图可以看出，在Ctrl-Z挂起程序后，输入fg可以让挂起的程序继续执行。

5、jobs

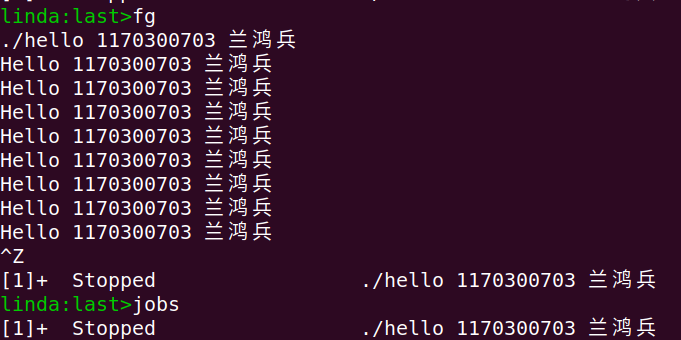


图6.6.5

jobs可以查看当前的关键命令内容，此时就输出Ctrl-Z内容。

6、pstree

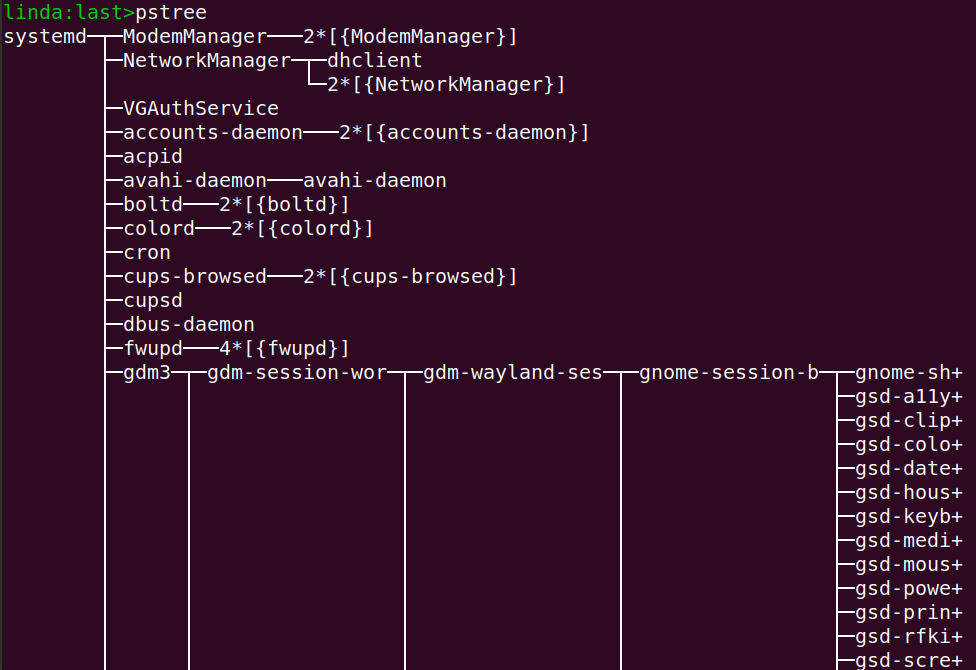


图6.6.6

pstree命令会将各进程用树状图的方式连接并输出。

7、kill

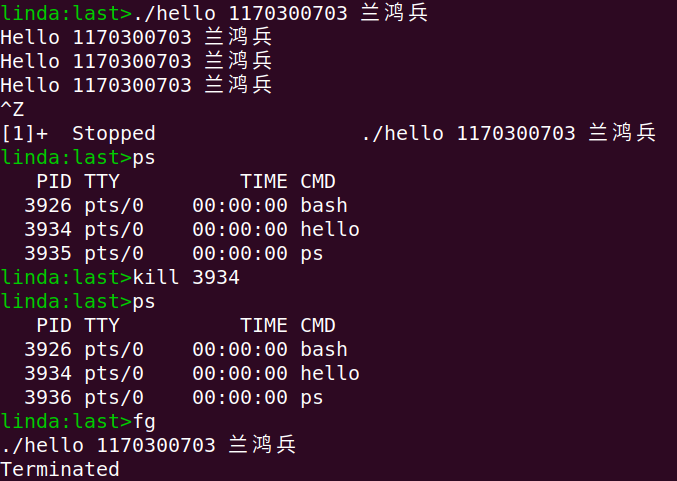


图6.6.7

如图，在挂起后输入kill指令——kill pid，能够杀死相应进程。

## 6.7本章小结

本章主要介绍了进程管理相关的内容。介绍了进程的概念和作用，简述了Shell-bash的作用与处理流程，并分析了其fork和execve的具体过程，最后分析了几个异常与信号处理。

**（第6章1分）**

# 第7章 hello的存储管理

## 7.1 hello的存储器地址空间

逻辑地址：又称相对地址，是程序运行时与段相关的偏移地址，是描述一个程序运行段的地址。

物理地址：程序运行时加载到内存地址寄存器中的地址，是内存单元真正的地址，在前端总线上传输且是唯一的。hello程序中，它代表程序运行时的指令在内存地址上的哪一个具体位置执行。

线性地址：即虚拟地址，是经过段机制转化之后，用于描述程序分页信息的地址，是对程序运行区块的一个抽象映射。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

首先给定一个完整的逻辑地址——段选择符：段内偏移地址

1、检查段选择符的T1=0还是1，以确定要转换是GDT中的段，还是LDT中的段，再根据相应的寄存器，得到其地址和大小。这样就得到了一个数组。

2、取段选择符中前13位，在数组中可以查找到对应段描述符，这样就获取了Base基地址。

3、最后把Base + offset，就是要转换的线性地址了。

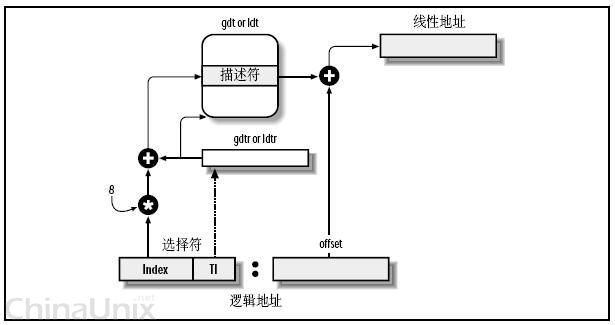


图7.2.1

## 7.3 Hello的线性地址到物理地址的变换-页式管理

转换中需要使用翻译后备缓冲器（TLB）。

1、首先将线性地址分成VPN（虚拟页号）+VPO（虚拟页偏移）的形式

2、再将VPN分成TLBT（TLB标记）+TLBI（TLB索引）的形式

3、在TLB缓存里寻找其所对应的PPN（物理页号）

4、若发生缺页情况，则查找对应PPN，将其与VPO合并为PPN+VPO，得到的就是所要生成的物理地址。

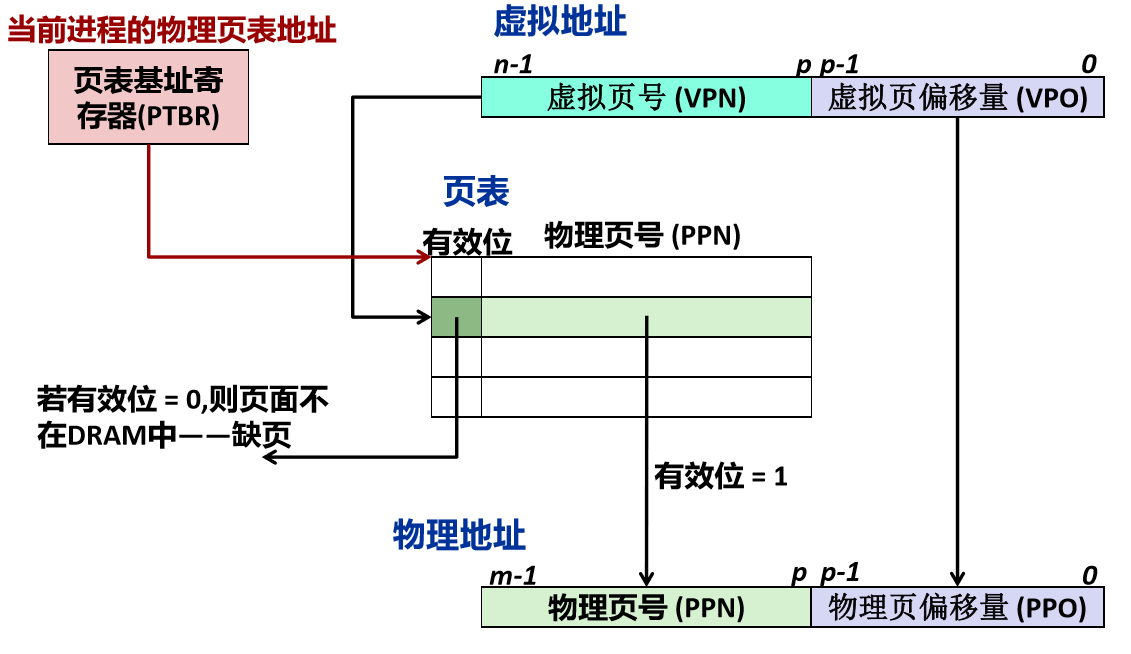


图7.3.1

## 7.4 TLB与四级页表支持下的VA到PA的变换

从VA中划分出VPN和VPO ，在TLB寻找对应PPN，若发生内存缺页，就到页表中寻找对应的物理地址，找到就一步步递进往下寻址，越往下一层每个条目所对应区域越小，寻址越细致，经过4层寻址后找到相应的PPN和VPO，拼接起来就完成了变换。

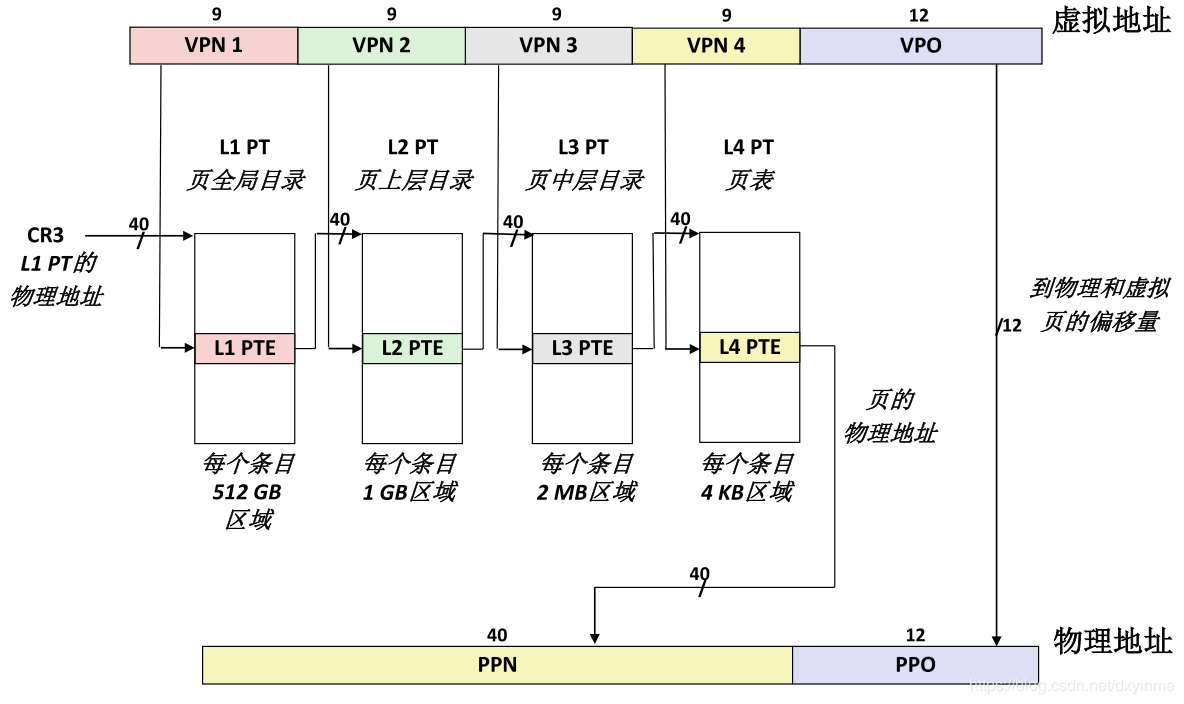


图7.4.1

## 7.5 三级Cache支持下的物理内存访问

1、得到物理地址后，将物理地址分成CT（标记）+CI（索引）+CO（偏移量）。

2、先在一级cache内部查找。根据给出的地址索引，先找到Cache中的组，在这些组中，根据地址中的标记位寻找组中的匹配行。若地址中的标记与组中某一行的标记相等，再根据偏移量得到数据块。

3、若未能找到标记位为有效的字节，就逐级到二级和三级cache中寻找对应的字节，找到之后返回结果。

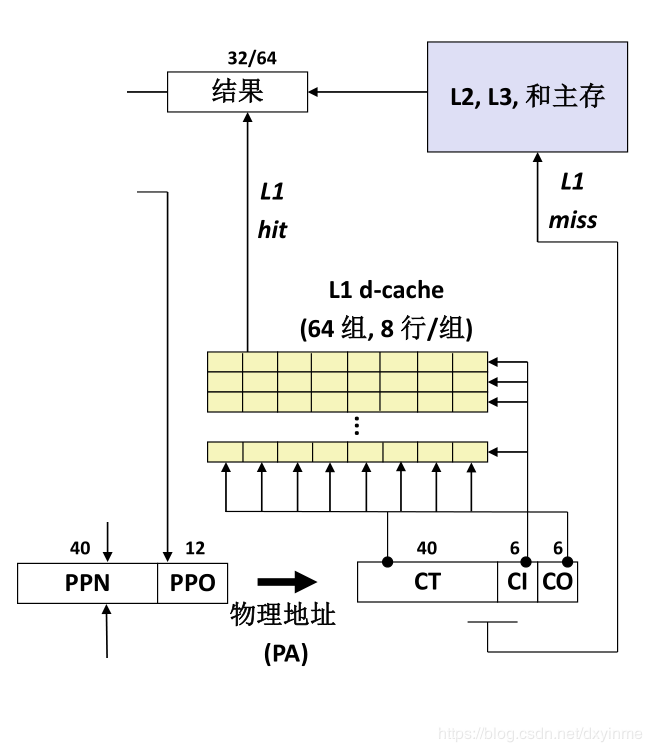


图7.5.1

## 7.6 hello进程fork时的内存映射

fork时使用到以下两个信息：

1、mm\_struct（内存描述符）：描述一个进程的整个虚拟内存空间

2、vm\_area\_struct（区域结构描述符）：描述进程虚拟内存空间的一个区间

使用fork创建虚拟内存时，有以下步骤：

1、创建当前进程的mm\_struct,vm\_area\_struct和页表的原样副本

2、将两个进程的页面都标记为只读

3、将两个进程的每个vm\_area\_struct都标记为私有，只能进行写时复制。

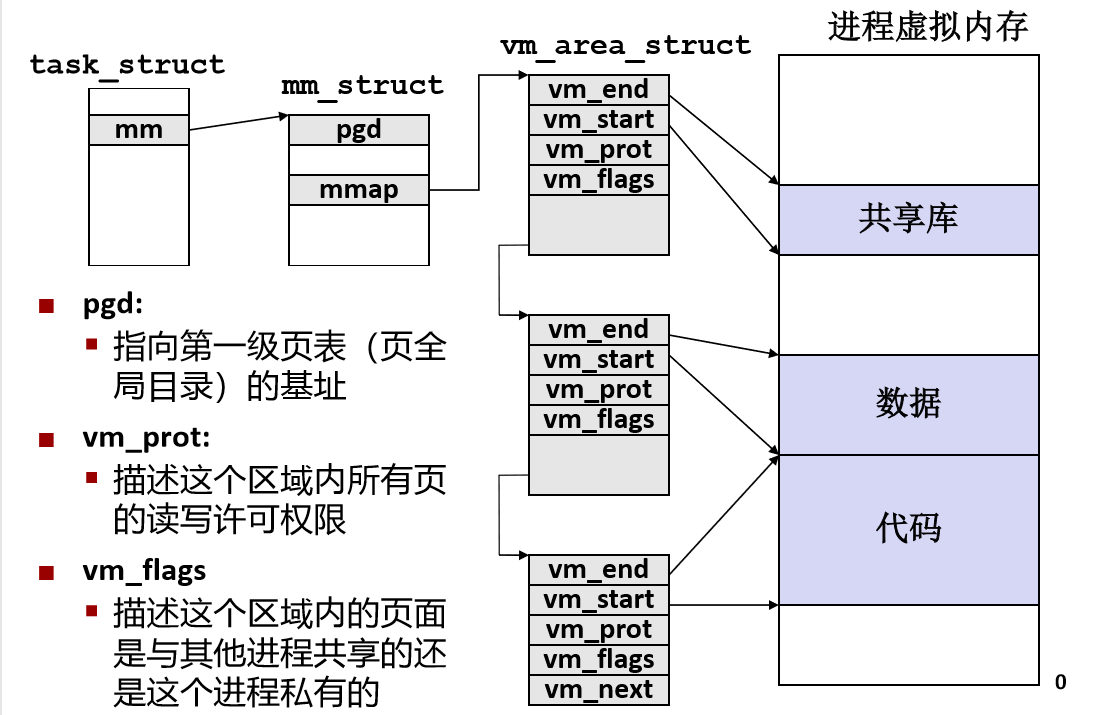


图7.6.1

## 7.7 hello进程execve时的内存映射

使用execve载入进程时，有以下步骤：

1、删除已存在的用户区域

2、创建新的私有区域（.malloc,.data,.bss,.text）

3、创建新的共享区域（libc.so.data,libc.so.text）

4、设置PC，指向代码区域的入口点

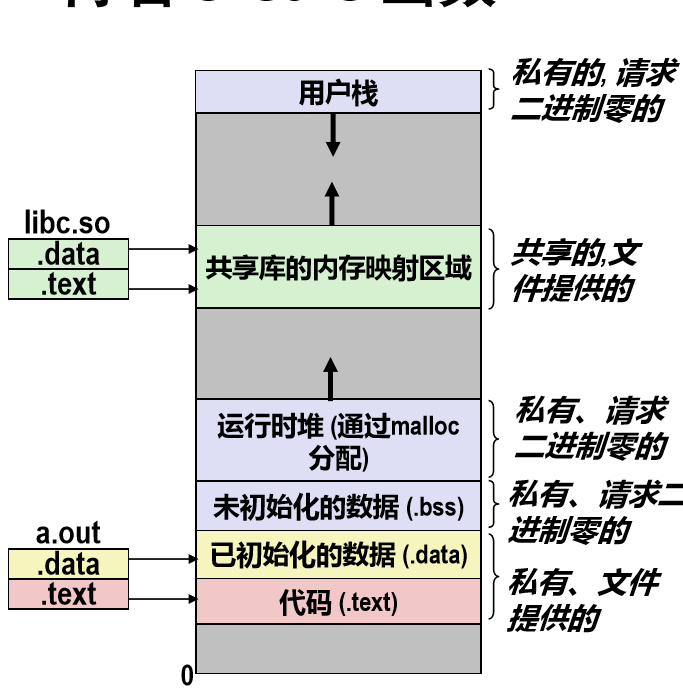


图7.7.1

## 

## 7.8 缺页故障与缺页中断处理

缺页故障：CPU给出虚拟地址，根据寄存器寻找相应的物理地址数据时，在页表条目中没有所要找的数据，这种情况就是缺页故障。

缺页中断处理：

情况1：段错误。先判断缺页的虚拟地址是否合法，遍历所有的合法区域结构，若虚拟地址对所有区域结构都不匹配，就返回一个段错误（segment fault）。

情况2：非法访问。接着检查这个地址权限，判断进程是否有读写改该地址的权限。

情况3：若非以上两种情况，就是正常缺页。选择一个页面牺牲换入新的页面，更新到页表即可。

## 7.9动态存储分配管理

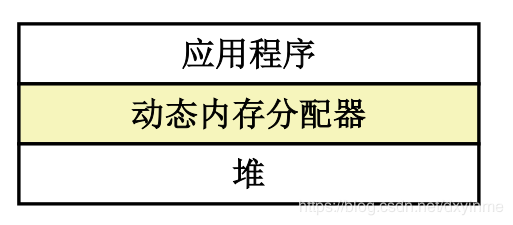


图7.7.1

在程序运行时，使用上图所示的动态内存分配器为引用程序分配内存，动态内存分配器维护进程的虚拟内存。分配的动态内存在地址段中按照下图中位置为堆栈分配内存空间，称为堆。

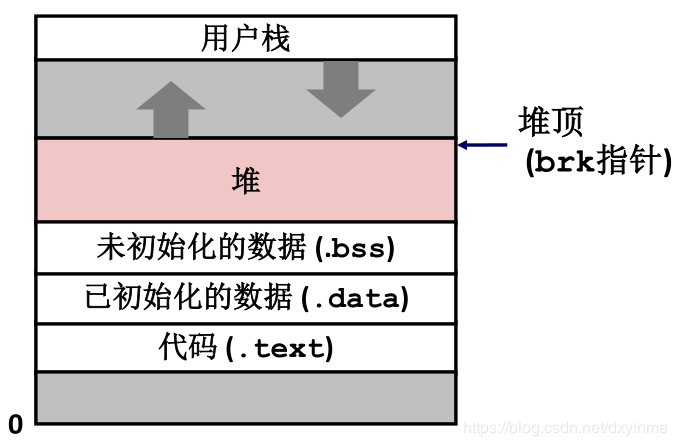


图7.7.2

hello程序使用的是C语言，其内存分配器为显式分配器，此处讨论显示分配器的内存管理方式。显示空间空间链表类似双向链表，只维护空闲空间块，不能通过块大小进行索引。

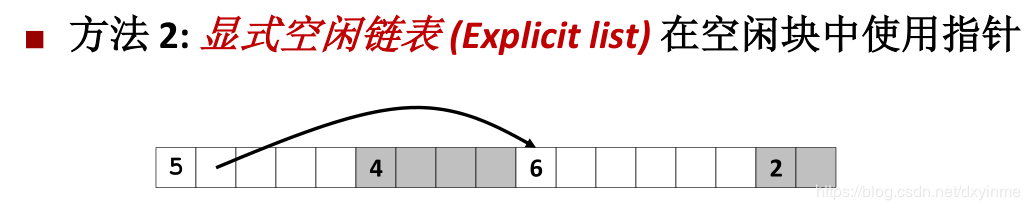


图7.7.3

使用malloc(size\_t size)函数申请内存时，每次内存空间都保证至少分配size\_t大小的内存，并保证双字对齐。每次都必须从空闲块中分配内存空间，申请内存空间时，要将空闲内存碎片合并，尽可能减少浪费。

## 7.10本章小结

本章介绍了hello的存储管理。简述了hello存储器地址空间，逻辑地址、物理地址、线性地址的概念。介绍了段式管理、页式管理的过程，以及VA到PA的转换。分析了三级Cache物理内存访问，程序fork和execve时的内存映射。最后给出了缺页故障的概念和缺页中断处理的流程，并给出了动态存储分配管理过程。

**（第7章 2分）**

# 第8章 hello的IO管理

## 8.1 Linux的IO设备管理方法

IO设备管理方法：

UNIX系统中，所有的I/O设备都被模型化为文件，而所有的输入和输出都被当做对相应文件的读和写来执行。这种将设备映射为文件的方式，允许UNIX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。

设备的模型化：文件

文件类型：

1、普通文件（regular file）：包含任意数据的文件。

2、目录（directory）：包含一组链接的文件，每个链接都将一个文件名映射到一个文件（他还有另一个名字叫做“文件夹”）。

3、套接字（socket）：用来与另一个进程进行跨网络通信的文件

4、命名通道

5、符号链接

6、字符和块设备

设备管理：unix io接口

1、打开和关闭文件

2、读取和写入文件

3、改变当前文件的位置

## 8.2 简述Unix IO接口及其函数

打开和关闭文件：

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

该函数会打开一个已经存在的文件或者创建一个新的文件如果open的返回值为-1则说明其打开该文件失败。

2.int close(int fd);

这个函数会关闭一个打开的文件。

读取和写入文件：

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

该函数会从当前文件位置复制字节到内存位置

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

该函数从内存复制字节到当前文件位置

\*读写文件时，如果返回值<0则说明出现错误

文件读写位置：

off\_t lseek(int handle, off\_t offset, int fromwhere);

该函数用于指示文件要读写位置的偏移量。

## 8.3 printf的实现分析

printf函数代码如下：

int printf(const char \*fmt, ...)   
{   
int i;   
char buf[256];   
      
     va\_list arg = (va\_list)((char\*)(&fmt) + 4);   
     i = vsprintf(buf, fmt, arg);   
     write(buf, i);   
      
     return i;

}

观察以上代码，可以发现printf函数中调用了两个外部函数，一个是vsprintf，还有一个是write。vsprintf函数的作用是，将所有参数内容格式化后存入buf，然后返回格式化后数组的长度；write函数的作用是，将buf中的i个元素写到终端。

printf运行过程：

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函数代码如下：

int getchar(void)

{

static char buf[BUFSIZ];

static char\* bb=buf;

static int n=0;

if(n==0)

{

n=read(0,buf,BUFSIZ);

bb=buf;

}

return(--n>=0)?(unsigned char)\*bb++:EOF;

}

从以上代码可以看到，getchar函数调用了一个read函数，read函数将缓冲区内容读取到buf里，返回值为缓冲区长度。代码中，若buf长度为0，才调用read函数，否则直接返回buf中最前面的元素。

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

本章讲述了linux系统I/O设备管理方法，介绍了Unix IO接口及相关函数，分析了printf和getchar函数的实现与操作过程。

**（第8章1分）**

# 结论

hello程序的执行，是在计算机系统从硬件到软件支持下才一步步完成的。

回顾hello的一生，它经过预处理和编译，才能被汇编成一个二进制文件hello.o，这样还不够，我们用链接在它身上链接上了一系列函数，才最终让它成为了一个可以被linux系统执行的二进制可执行文件。我们在shell中使用fork和execve执行hello，hello成为了一个进程，被映射到内存里去，逐条执行，CPU为其分配时间片，它拥有了自己的逻辑流。我们又暂停挂起它，然后脸滚键盘，打满缓冲区，又试着kill掉hello，看看它出现异常后的反应，调用它的异常处理程序。到了最后，我们才使用Ctrl-C，结束了它的一生。shell回收了hello的内存，让它消失得无影无踪。我们也借着它的余晖，分析了它的内存管理和系统IO。

hello是一个简单的程序，但它背后实现的过程却不简单。做完大作业，回顾了计算机系统的课程，对整个计算机系统有了新的理解。

**（结论0分，缺失 -1分，根据内容酌情加分）**

# 附件

hello.c hello的c语言源程序

hello.i hello.c预处理后的程序文本

hello.s hello.i编译成汇编语言后的程序文本

hello.o hello.s生成的二进制文件

hello.objdump hello.o反汇编后的文件

hello hello通过链接操作后生成的二进制可执行文件

hello\_back.txt hello反汇编后重定向生成的程序文本

helloelf.txt hello.o的readelf输出的重定向文本

helloelf\_second.txt hello的readelf输出的重定向文本

**（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] 林来兴. 空间控制技术[M]. 北京：中国宇航出版社，1992：25-42.

[2] 辛希孟. 信息技术与信息服务国际研讨会论文集：A集[C]. 北京：中国科学出版社，1999.

[3] 赵耀东. 新时代的工业工程师[M/OL]. 台北：天下文化出版社，1998 [1998-09-26]. http://www.ie.nthu.edu.tw/info/ie.newie.htm（Big5）.

[4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨：哈尔滨工业大学，1992：8-13.

[5] KANAMORI H. Shaking Without Quaking[J]. Science，1998，279（5359）：2063-2064.

[6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science，1998，281：331-332[1998-09-23]. http://www.sciencemag.org/cgi/ collection/anatmorp.

[7] <https://blog.csdn.net/tuxedolinux/article/details/80317419>

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

[9] <https://blog.csdn.net/gdj0001/article/details/80135196>

[10] <http://www.techlog.cn/article/list/10182663>

[11] <https://blog.csdn.net/GDJ0001/article/details/80135196>

**（参考文献0分，缺失 -1分）**