

**计算机系统**

**大作业**

题 目 程序人生-Hello’s P2P

专 业 计 算 机

学　　 号 1190201625

班　　 级 1903009

学 生 高 雨 涵

指 导 教 师 吴 锐

**计算机科学与技术学院**

**2021年5月**

**摘 要**

摘要是论文内容的高度概括，应具有独立性和自含性，即不阅读论文的全文，就能获得必要的信息。摘要应包括本论文的目的、主要内容、方法、成果及其理论与实际意义。摘要中不宜使用公式、结构式、图表和非公知公用的符号与术语，不标注引用文献编号，同时避免将摘要写成目录式的内容介绍。

**关键词：预处理 编译 汇编 链接 加载 进程 存储管理 IO管理**

**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：hello.c经过cpp预处理，ccl编译，as汇编，ld链接形成可执行目标文件hello。

020：在shell中键入命令，shell调用execve函数加载hello，载入物理内存。main函数执行目标代码，结束后shell回收hello进程，删除数据结构。

## 1.2 环境与工具

X64 CPU；2GHz；2G RAM；256GHD Disk 以上

Windows7 64位以上；VirtualBox/Vmware 11以上；Ubuntu 16.04 LTS 64位/优麒麟 64位

gcc edb objdump readelf

## 1.3 中间结果

hello.c:源程序

hello.i:预处理后生成的文件

hello.s:编译后生成文件

hello.o:汇编后生成的可重定位目标文件

hello:链接后的可执行目标文件

hello1.txt:hello.o的elf格式

hello2.txt:hello的elf格式

objhello1.txt:hello.o的反汇编

objhello2.txt:hello的反汇编

## 1.4 本章小结

介绍了P2P，020的含义，过程，介绍了本次大作业中的硬件环境，软件环境等，简要概括.c文件到可执行文件hello的过程。

**（第1章0.5分）**

# 第2章 预处理

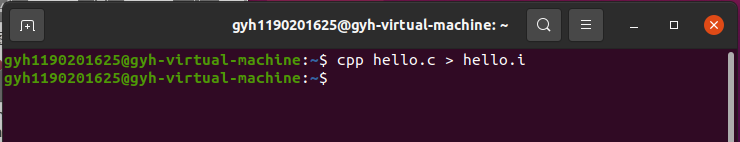
## 2.1 预处理的概念与作用

概念：预处理中会展开以#起始的行，其中ISO C/C++要求支持的包括#if/#ifdef/#ifndef/#else/#elif/#endif（条件编译）、#define（宏定义）、#include（源文件包含）、#line（行控制）、#error（错误指令）、#pragma（和实现相关的杂注）以及单独的#（空指令）

作用：1.宏定义； 2.文件包含； 3.条件编译。

## 2.2在Ubuntu下预处理的命令

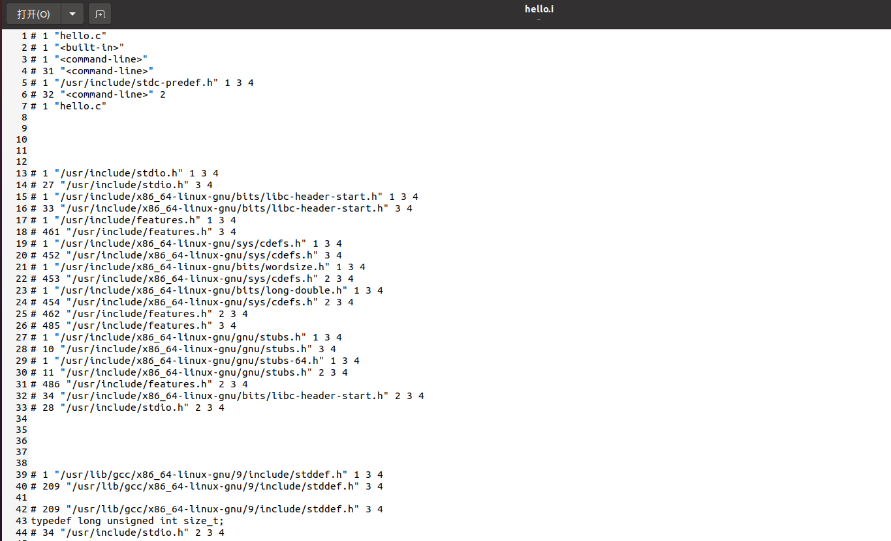
cpp hello.c > hello.i

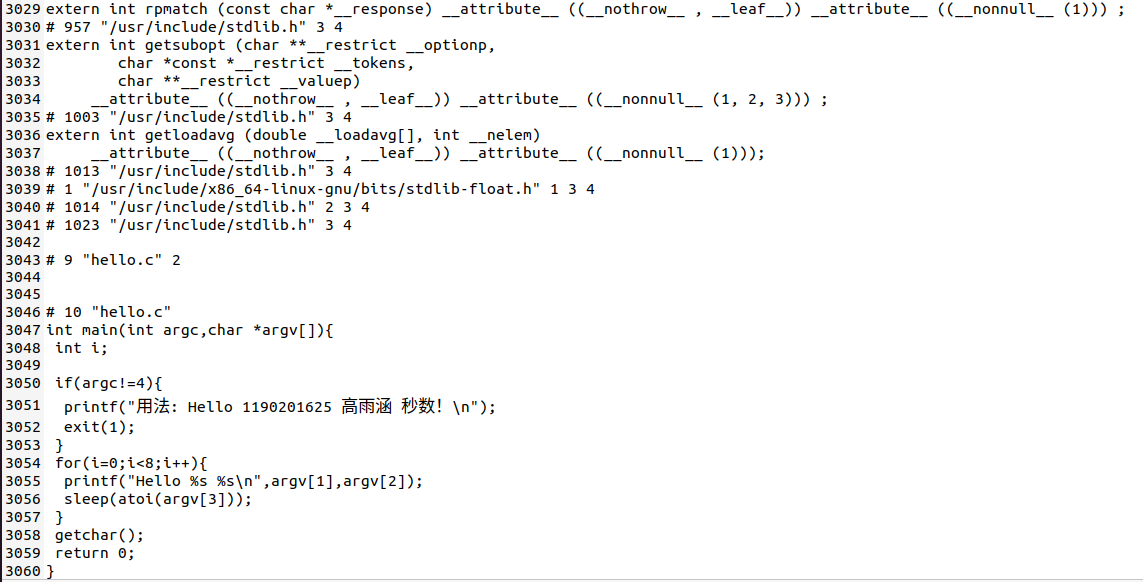


2.2预处理指令截图

## 2.3 Hello的预处理结果解析

我们打开hello.i文件可以看到原来的#已经全部消失，程序拓展为3060行，头文件 stdio.h unistd.h stdlib.h展开。编译器回先删除#define的宏，然后处理所有条件预编译指令，插入头文件到#include处，删除所有的注释，添加行号和文件名标识，保留所有#pragma编译指令。





2.3 hello.i预处理结果截图

## 2.4 本章小结

本章介绍了hello.c到hello.i的预处理过程，在ubuntu下进行实际操作，观察相应变化，更加加深对预处理的理解。

**（第2章0.5分）**

# 第3章 编译

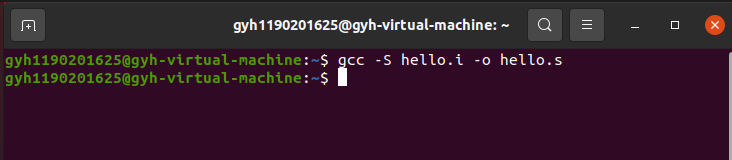
## 3.1 编译的概念与作用

概念：编译程序把一个预处理文件翻译成目标程序的工作过程分为五个阶段：词法分析；语法分析；语义检查和中间代码生成；代码优化。五个阶段后生成汇编代码文件。

作用：把源程序翻译成等价的汇编语言，使之进一步接近机器代码。此时，计算机仍然无法识别执行它。

## 3.2 在Ubuntu下编译的命令

gcc -S hello.i -o hello.s



3.2 编译命令截图

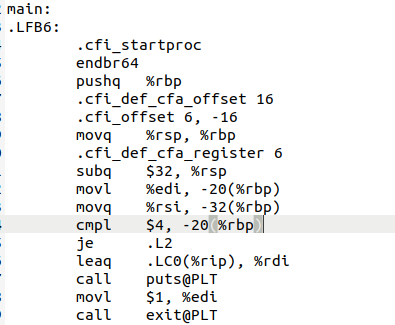
## 3.3 Hello的编译结果解析

1.常量：常量被存储在.text节中。



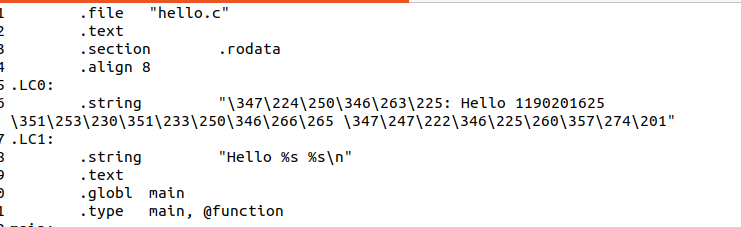
3.3.1 源程序截图

由源码可知，常量分别为4，0，8，1，2，3。打开hello.s观察可以看到，常量在编译以后在.text节中以常量的形式出现。



3.3.2 部分反汇编截图

而字符串常量存在.rodata节。



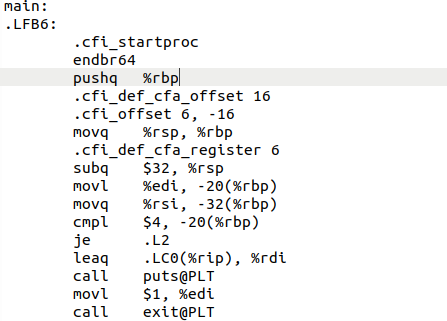
3.3.3 部分反汇编截图

2.变量：局部变量保存在栈或者寄存器中，全局变量保存在.data节。



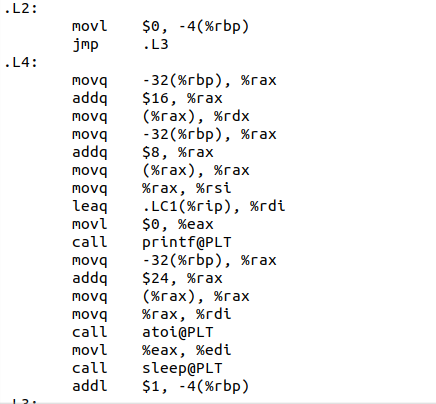
3.3.4 源程序截图

由源码知，i、argc、argv为局部变量



3.3.5 部分反汇编截图

%rsi，%rdi分别存储argv，argc，由反汇编可知，其被放在栈中。



3.3.6 部分反汇编截图

变量i在%eax中，初始化为0，然后随着循环次数而增加。

3.赋值：

在for循环中，每次i加1后赋值给i。



3.3.7 部分反汇编截图

4.算术操作：

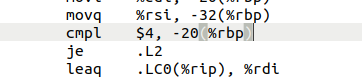
在for循环中，每次i加1。



3.3.8 部分反汇编截图

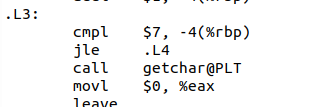
5.关系操作：

将argc与4比较



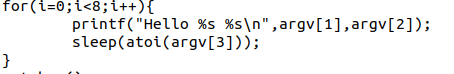
3.3.9 部分反汇编截图

将i与8比较



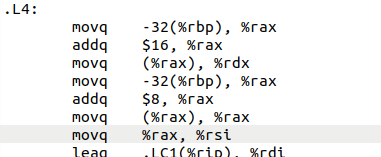
3.3.10 部分反汇编截图

6.数组操作：



3.3.11 部分反汇编截图

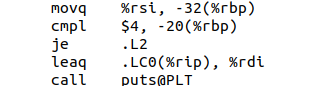
此段源码使用数组，反汇编可知，argv[0]地址从栈中传递给%rax，argv[1]存在%rsi ，argv[2]存在%rdx。



3.3.12 部分反汇编截图

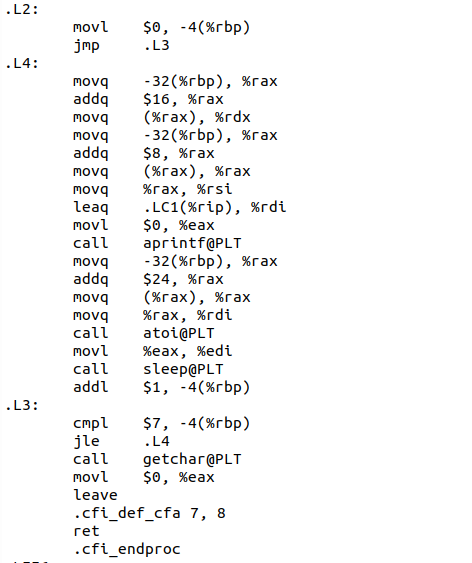
7.控制转移：

If的控制转移：如果argc不等于4，则继续执行，如果等于，则跳转到.L2



3.3.13 部分反汇编截图

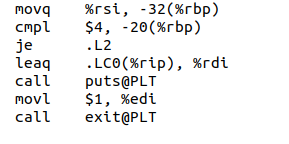
for循环的控制转移：如果条件不成立，直接跳转至.L3，成立则执行.L4循环体。



3.3.14 部分反汇编截图

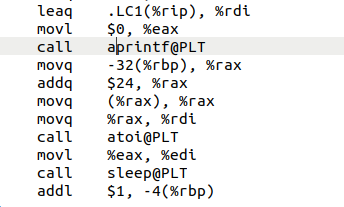
8.函数操作：

调用printf函数和exit函数。



3.3.15 部分反汇编截图

for循环体中调用printf，sleep，atoi函数



3.3.16 部分反汇编截图

## 3.4 本章小结

介绍编译器的编译过程，解析在编译过程中常量变量，相关操作的汇编表示，要求我们能看懂汇编，接触底层。

**（第3章2分）**

# 第4章 汇编

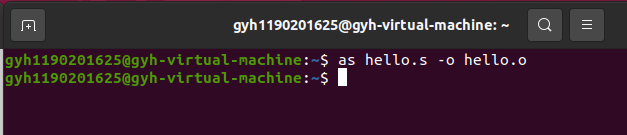
## 4.1 汇编的概念与作用

概念：汇编将.s文件翻译成二进制机器语言指令，形成可重定位文件hello.o。

作用：将编译生成的汇编语言代码转换为机器语言。

## 4.2 在Ubuntu下汇编的命令

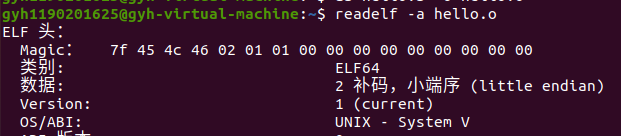
as hello.s -o hello.o



4.2 汇编命令截图

## 4.3 可重定位目标elf格式

命令：readelf -a hello.o



4.3.1 readelf命令截图

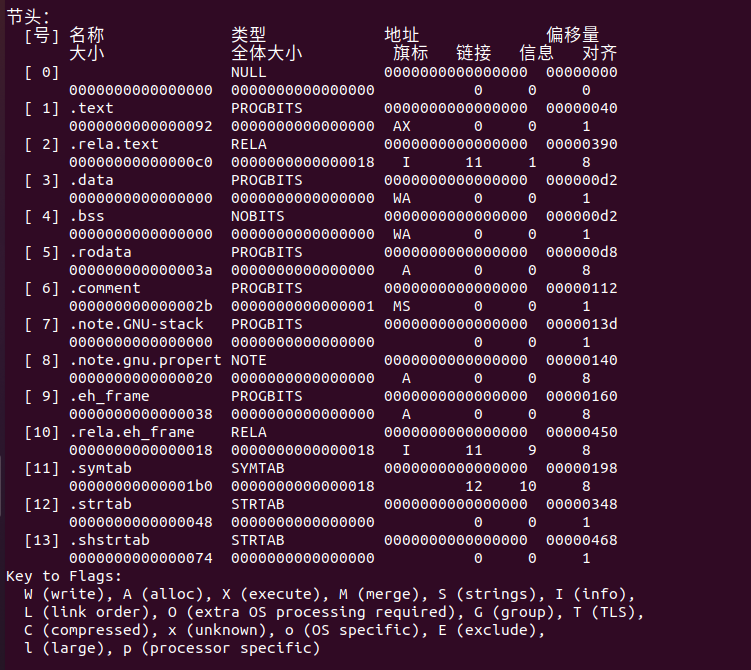
1. ELF头

文件开头4个字节是魔数，通常用来确定文件的类型或格式。剩下的12个字节，主要包含一些标识信息。剩下的部分包含可重定位文件的一些信息，包括ELF头大小，目标文件类型，机器类型，节头部表的文件偏移，节头部表的大小和数量等。



4.3.3 elf头截图

1. 节头部表

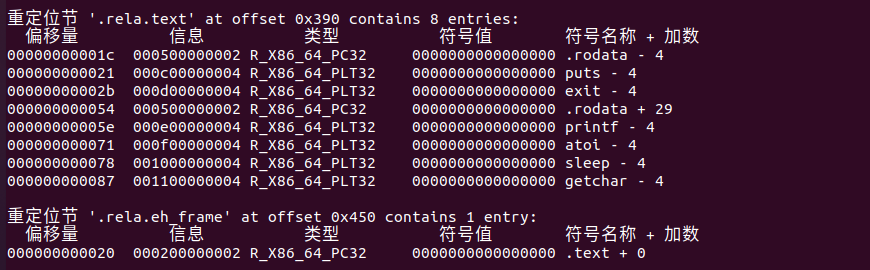


4.3.4 节头部表截图

节头表由若干个表项组成，每个表项描述相应的一个节的节名，在文件中的偏移，大小，访问属性，对齐方式等，目标文件中的每个节都有一个表项与之对应。

1. 重定位节

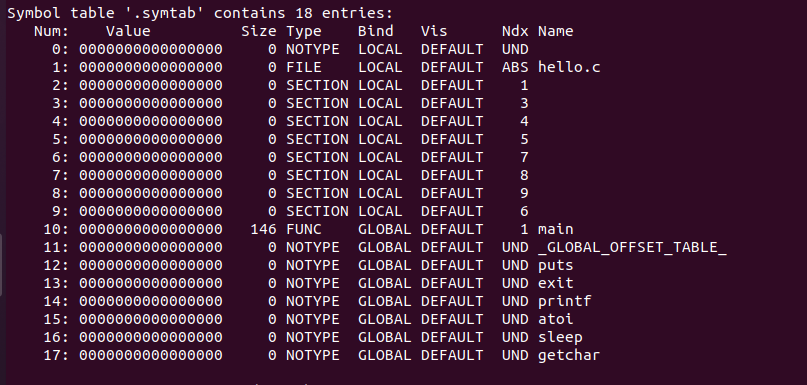
描述了需要重定位的符号的位置和重定位方式。



4.3.5 重定位节截图

1. 符号表

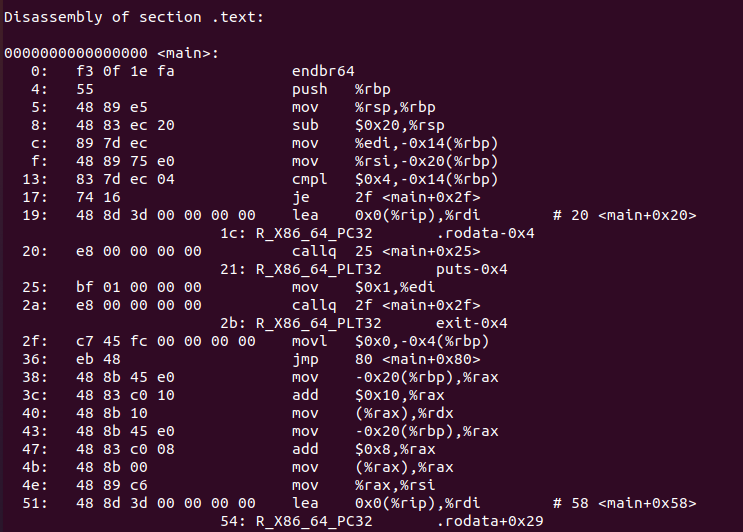
描述符号类型，符合位置，符号所在节，符号属性等符号的相关信息。

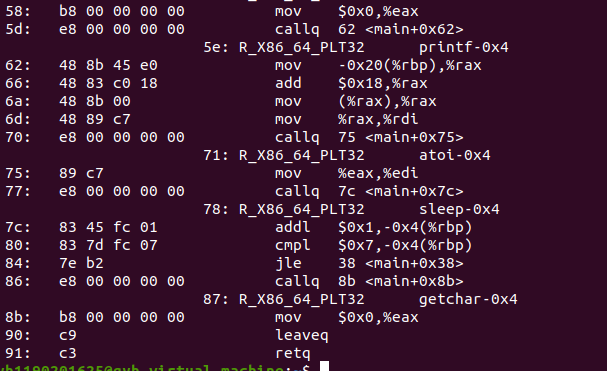


4.3.6 符号表截图

## 4.4 Hello.o的结果解析

命令：objdump -d -r hello.o





4.4 hello.o反汇编截图

由反汇编可以看出，首先，重定位文件的起始地址是0。并且在每一个符号引用的位置都有一条相应的重定位信息，分别显示了重定位位置，重定位方式，以及重定位的符号位置。R\_X86\_64\_32：重定位绝对引用。重定位时使用一个32位的绝对地址的引用。。R\_X86\_64\_PC32：重定位PC相对引用。重定位时使用一个32位PC相对地址的引用。一个PC相对地址就是据程序计数器的当前运行值的偏移量来进行重定位。

## 4.5 本章小结

本章介绍了可重定位目标文件的格式，通过ELF头条目，反汇编等对可重定位目标文件进行了一个大体的了解概括。

**（第4章1分）**

# 第5章 链接

## 5.1 链接的概念与作用

概念：链接器将所有的关联到的目标文件代码，包括用到的标准库函数目标文件，按照某种形式组合在一起，形成一个具有统一地址空间的可被加载到存储器直接执行的程序。

作用：产生一个完全链接的，可以加在运行的可执行目标文件。

## 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



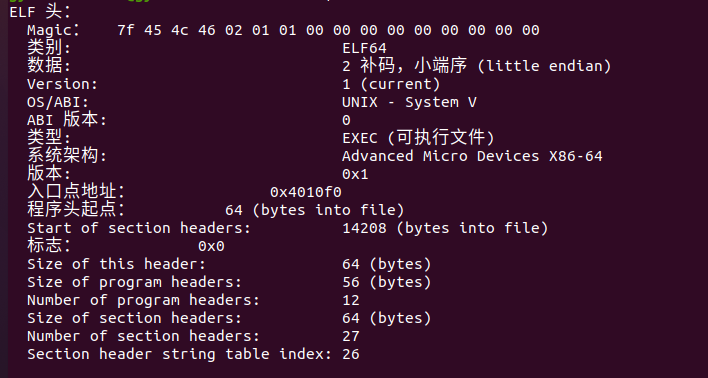
5.2 链接命令截图

## 5.3 可执行目标文件hello的格式

命令：readelf -a hello

1. ELF头

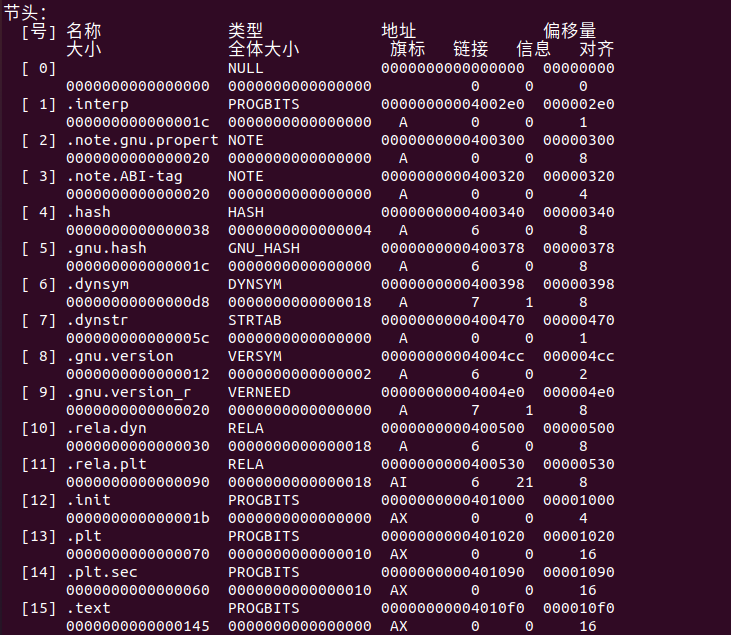
字段入口点地址为执行程序时第一条指令地址。文件开头4个字节是魔数，通常用来确定文件的类型或格式。剩下的12个字节，主要包含一些标识信息。剩下的部分包含可重定位文件的一些信息，包括ELF头大小，目标文件类型，机器类型，节头部表的文件偏移，节头部表的大小和数量等。

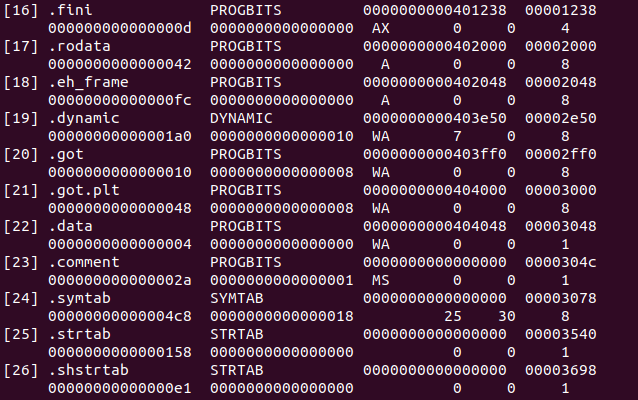


5.3.1 elf头截图

1. 节头表

节头表由若干个表项组成，每个表项描述相应的一个节的节名，在文件中的偏移，大小，访问属性，对齐方式等，目标文件中的每个接都有一个表项与之对应。

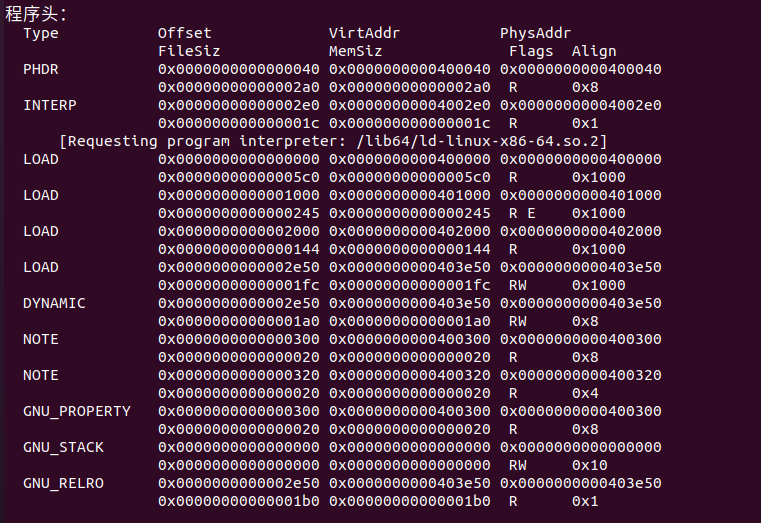




5.3.2 节头表截图

1. 程序头表

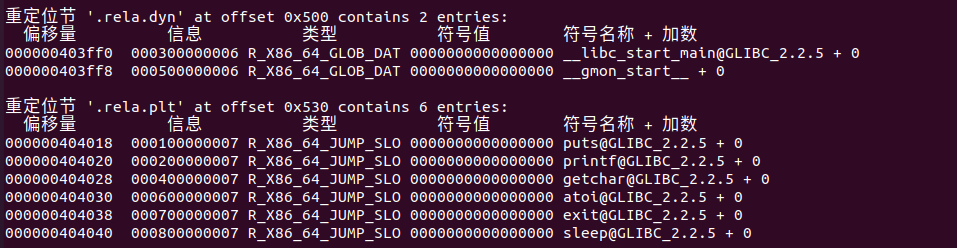
用于可执行文件装载。



5.3.3 程序头表截图

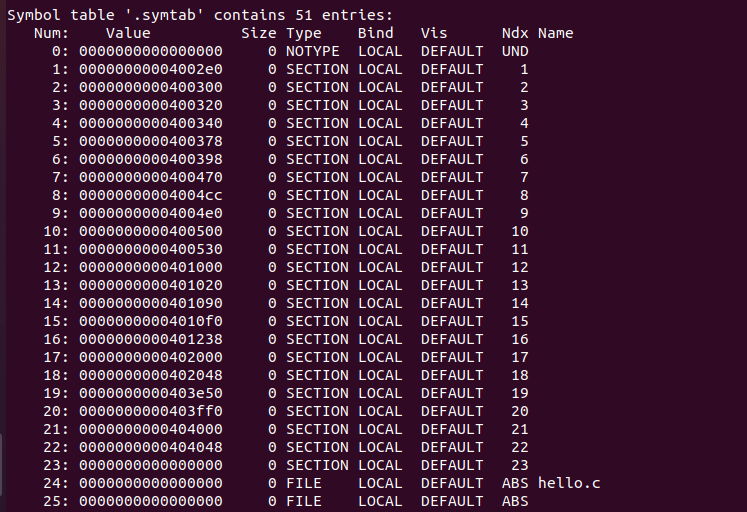
1. 重定位节

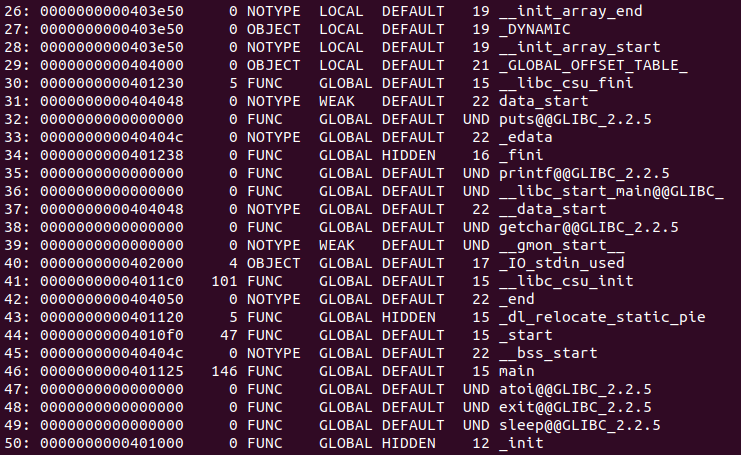
可执行文件中的重定位节应该是动态链接库中函数的重定位信息。



5.3.4 重定位节截图

1. 符号表



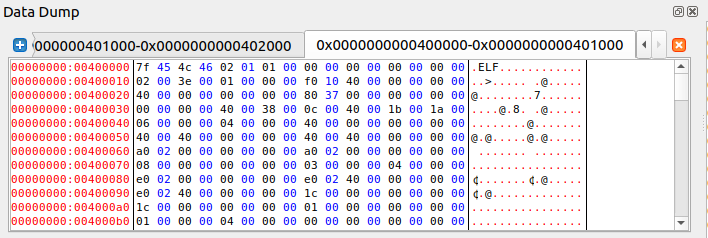


5.3.5 符号表截图

描述符号类型，符合位置，符号所在节，符号属性等符号的相关信息。

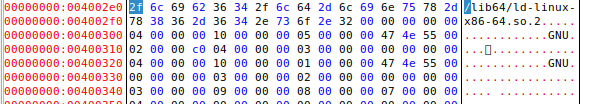
## 5.4 hello的虚拟地址空间

edb加载hello，查看虚拟地址如下



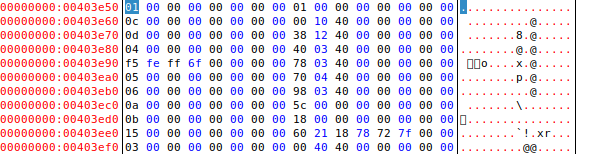
5.4.1 虚拟地址起始地址截图

由此我们可以看出，程序虚拟地址开始于0x400000。



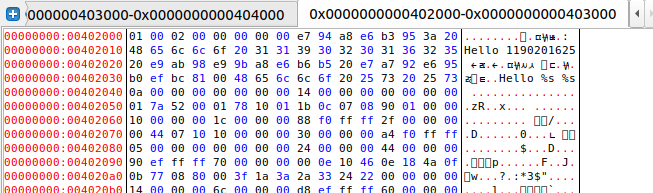
5.4.2 .interp段截图

此为.interp段，其中包含了动态链接器路径名ld-linux.so，加载器根据指定路径加载并启动动态链接器运行。



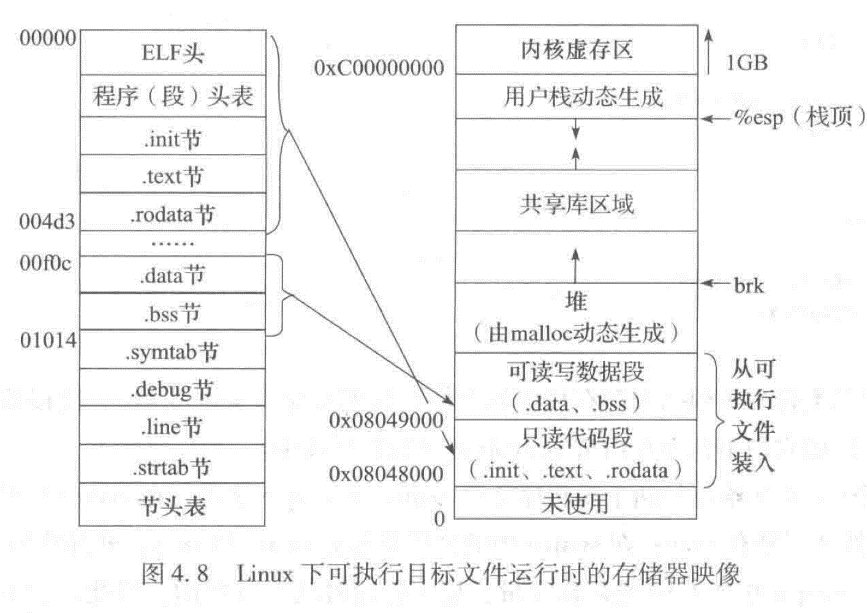
5.4.3 数据段截图

此为数据段，其中包括.bss,.data节。



5.4.4 代码段截图

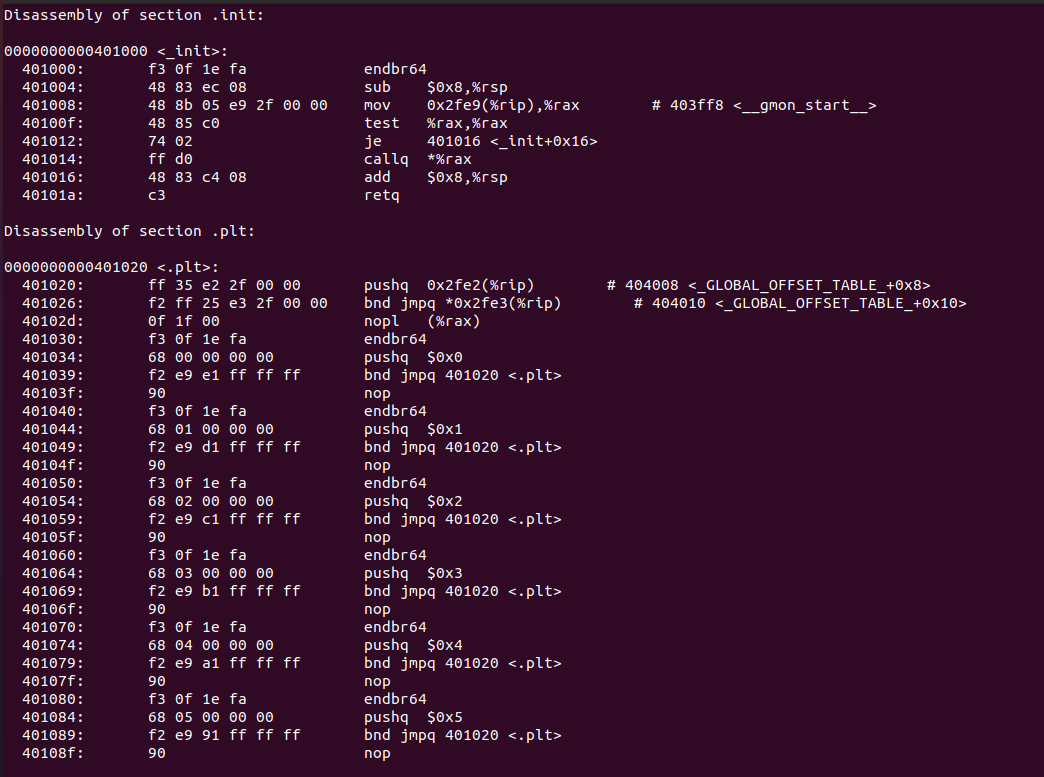
此为代码段，包含.text,.init,.rodata,elf头。

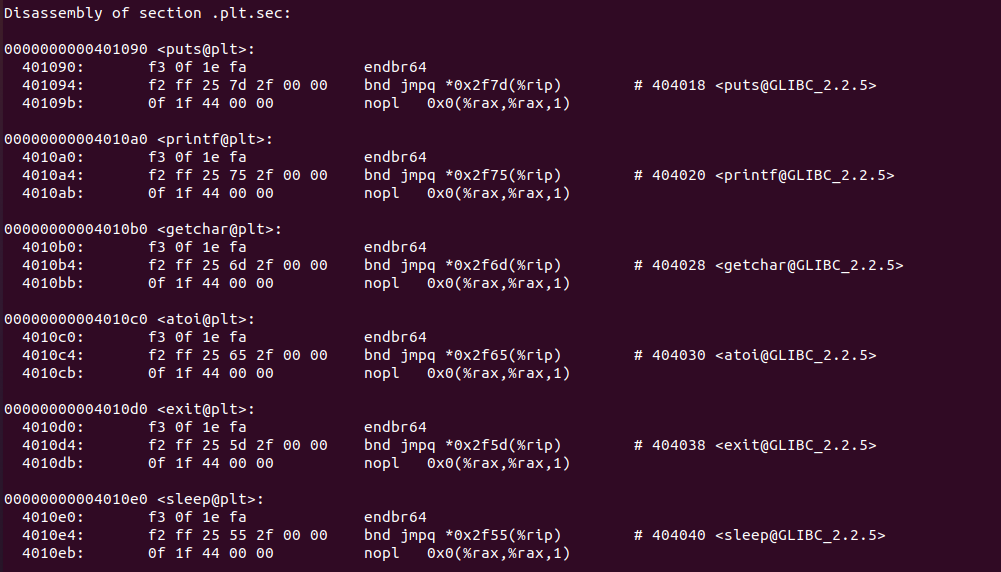


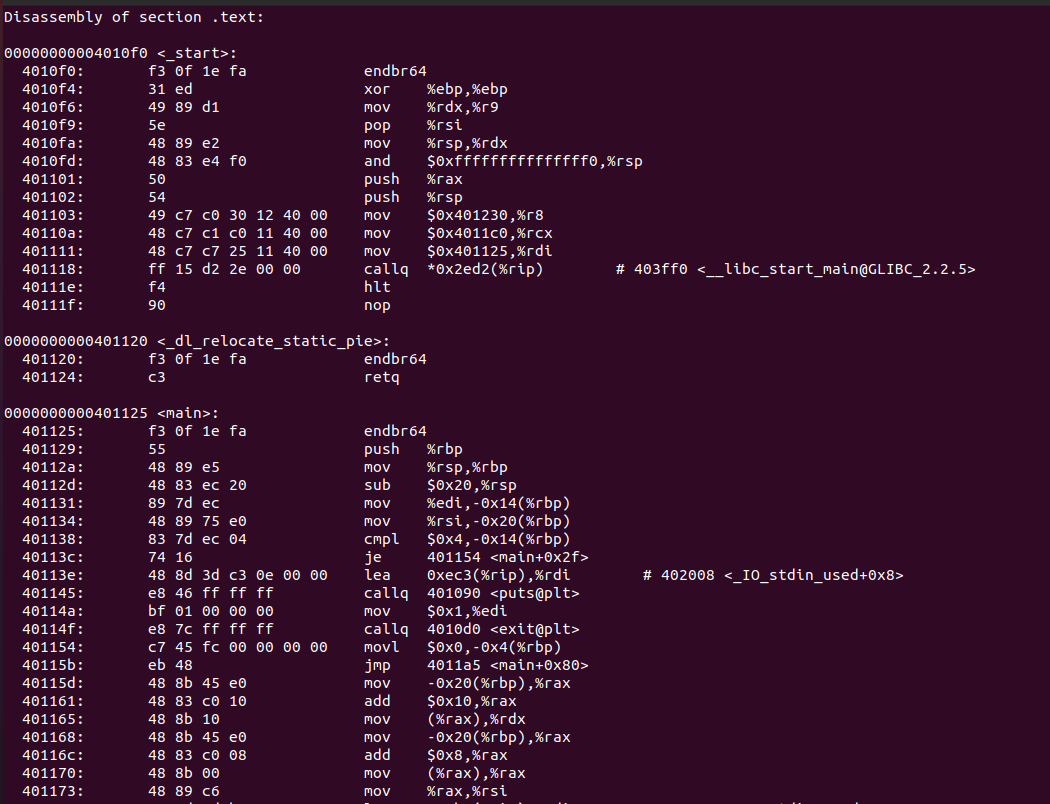
5.4.5 可执行文件映射到虚拟地址空间

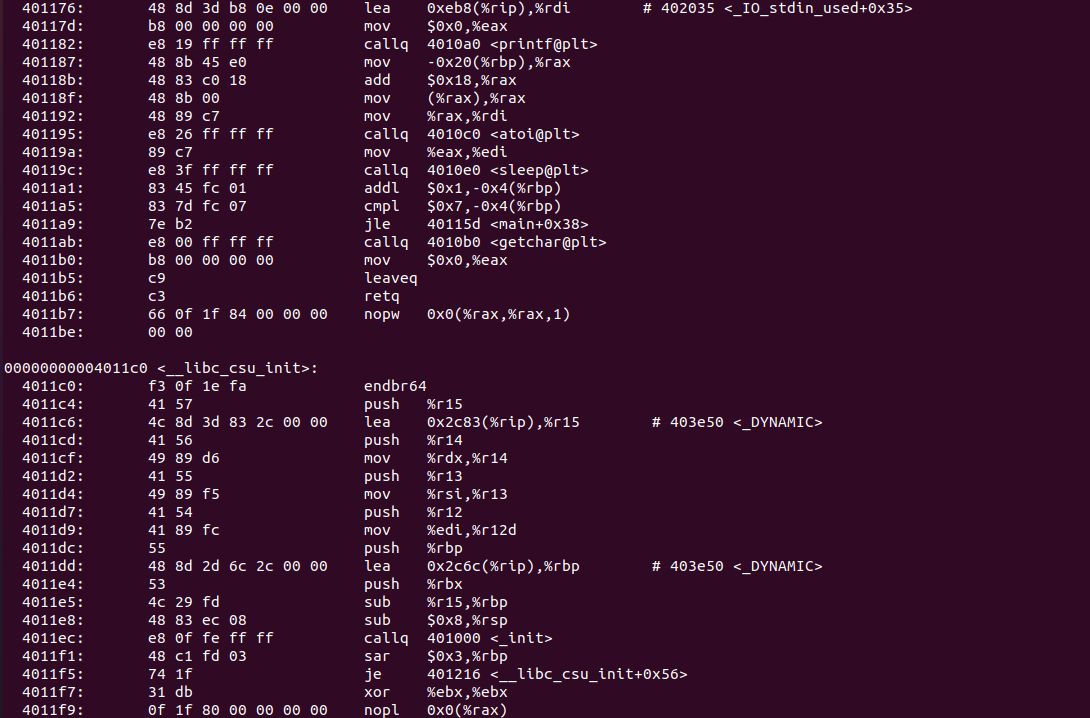
## 5.5 链接的重定位过程分析

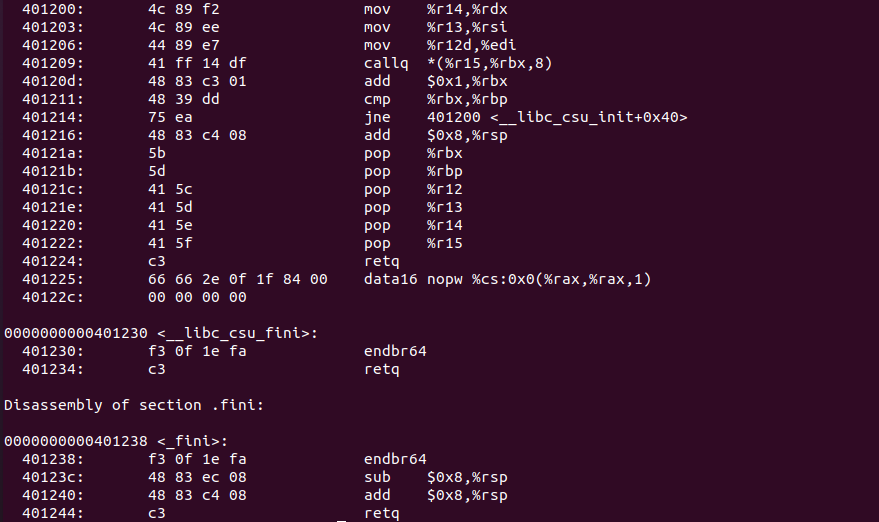
命令：objdump -d -r hello











5.5 hello.o反汇编截图

hello.o起始地址为0,hello起始地址为0x400000。hello.o具有可重定位条目，hello已经将相应的符号地址填入相应位置。hello中新增加了一些节。

## 5.6 hello的执行流程

401000 <\_init>

401020 <.plt>

401030 <puts@plt>

401040 <printf@plt>

401050 <getchar@plt>

401060 <atoi@plt>

401070 <exit@plt>

401080 <sleep@plt>

4010f0 <\_start>

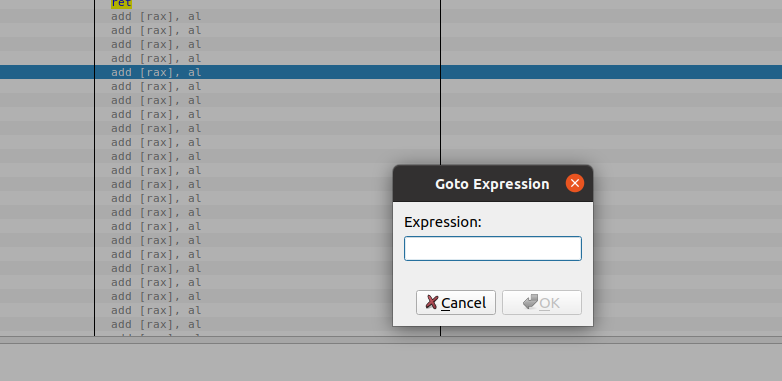
401120 <\_dl\_relocate\_static\_pie>

401125 <main>

4011c0 <\_\_libc\_csu\_init>

401230 <\_\_libc\_csu\_fini>

401238 <\_fini>



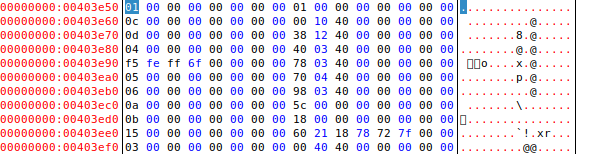
5.6 edb搜索函数地址截图

输入函数名即可

## 5.7 Hello的动态链接分析

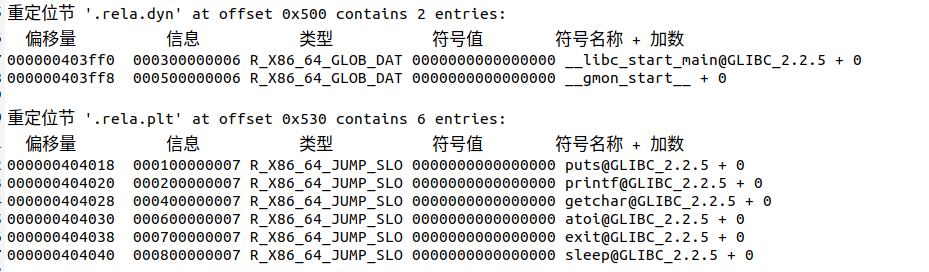
动态链接对变量，用代码段数据段相对位置不变的原则计算即可。对于外部函数，需要PLT,GOT配合。PLT存的是引用的外部函数的地址，GOT存的是引用的外部变量的地址。链接器在链接时会根据这两个条目对外部引用的符号进行重定位。

.interp段，其中包含了动态链接器路径名ld-linux.so，加载器根据指定路径加载并启动动态链接器运行。



5.7.1 .interp段截图

根据可重定位信息对动态库中链接进来的数据进行引用。



5.7.2 动态链接可重定位信息截图

## 5.8 本章小结

本章介绍链接的相关知识，可执行文件的相关信息，以及可执行文件加载的信息，动态链接的知识等。

（*以下格式自行编排，编辑时删除*）

**（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：程序的一次运行过程。

作用：进程使得计算机的所有资源看起来是自己的程序独占，简化了编译，链接，共享，加载等过程。

## 6.2 简述壳Shell-bash的作用与处理流程

shell-bash 是一种命令行解释器, 其读取用户输入的字符串命令, 解释并且执行命令。它是一种特殊的应用程序, 介于系统调用/库与应用程序之间, 其提供了运行其他程序的的接口。它可以是交互式的, 即读取用户输入的字符串;也可以是非交互式的, 即读取脚本文件并解释执行, 直至文件结束。

处理流程：1.用户输入命令，shell通过parseline函数解析命令。

2.解析命令行后，判断函数builtin\_command判断是否为内置命令。如果是，立即解释这个命令。

3.如果不是，shell创建一个子进程，调用execve函数加载可执行文件。

4.判断是否为前台运行程序，如果是，则等待前台作业结束。如果不是，则后台运行，开始处理下一次用户命令。

## 6.3 Hello的fork进程创建过程

当我们在shell命令行输入./hello时，由于hello不是内置命令，shell会通过fork函数创建一个子进程，子进程和父进程几乎完全相同，拥有相同的代码数据段，堆，共享库，用户栈，最大区别是不同的PID。fork函数调用一次返回两次，父进程得到子进程PID，子进程返回0.

## 6.4 Hello的execve过程

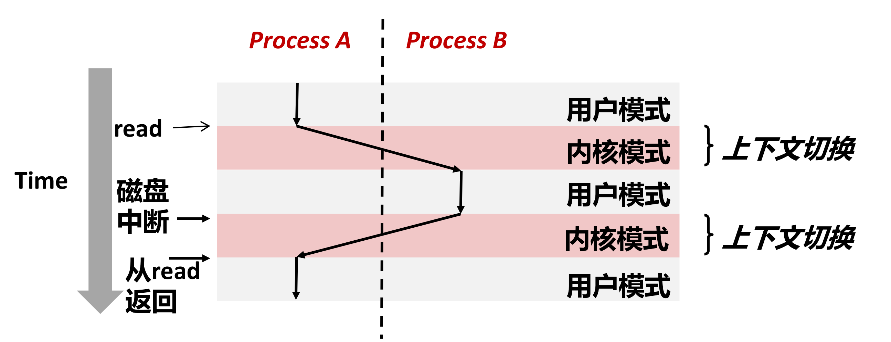
Shell创建子进程后，子进程会调用execve函数加载可执行文件hello，此函数调用一次从不返回，功能是在当前进程中载入并运行一个新程序，它将hello映射到虚拟地址空间。先删除已存在的用户区域。然后映射私有区：为hello的代码、数据、.bss和栈区域创建新的区域结构，所有这些区域都是私有的、写时才复制的。再映射共享区，然后再用户虚拟地址空间中的共享区域内。最后设置PC使之指向代码区域的入口点。

## 6.5 Hello的进程执行

进程是一个执行中程序的实例，它营造一种假象，每个程序似乎独占的使用cpu，每个程序似乎独占的使用内存系统。进程通过内核的上下文切换机制实现并发。

hello程序与其他进程并发，切换上下文，各自拥有自己的时间片。程序会在内核和用户进程中来回切换，营造出并发的现象。sleep和hello就是这样的模式。

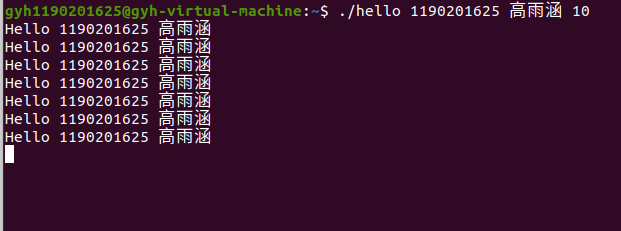
程序开始运行在用户模式下，在hello运行至调用sleep后进入内核模式，内核进程保存hello的上下文，然后切换至其他进程，定时器开始计时，计时结束后内核恢复hello的进程上下文，恢复完成后又将控制转移给hello。



6.5 上下文切换图示

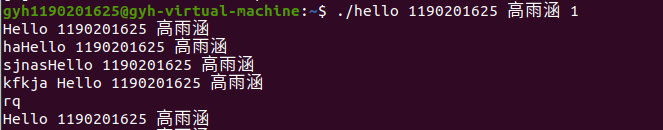
## 6.6 hello的异常与信号处理

正常运行：



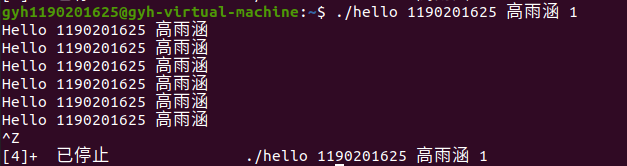
6.6.1 正常运行截图

1.乱按



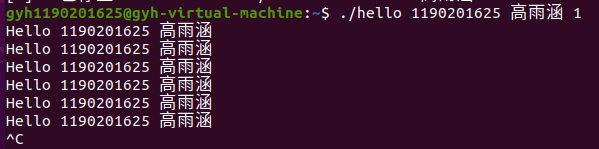
6.6.2 乱按截图

2.ctrl+z



6.6.3 ctrl+z截图

3.ctrl+c

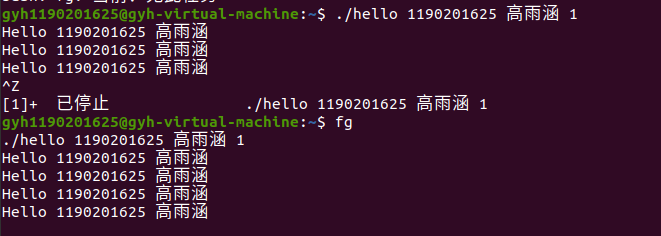


6.6.4 ctrl+c截图

4.ps，jobs，fg，kill等命令



6.6.5 ps，jobs，kill截图



6.6.6 fg截图

## 6.7本章小结

本章介绍了进程的相关知识，介绍了进程的上下文切换机制。介绍了shell如何作为用户和系统内核的接口以及shell的基本操作，内核信号命令。介绍了fork函数，execve加载函数等功能。

**（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：**指由程序产生的与段相关的偏移地址部分。

**线性地址：**逻辑地址到物理地址变换之间的中间层。在分段模式中逻辑地址等于段中的偏移地址加上基地址就是线性地址。

**虚拟地址：**CPU启动保护模式后，程序运行在虚拟地址空间中。注意，并不是所有的“程序”都是运行在虚拟地址中。CPU在启动的时候是运行在实模式的，Bootloader以及内核在初始化页表之前并不使用虚拟地址，而是直接使用物理地址的。

**物理地址：**放在寻址总线上的地址。放在寻址总线上，如果是读，电路根据这个地址每位的值就将相应地址的物理内存中的数据放到数据总线中传输。如果是写，电路根据这个地址每位的值就在相应地址的物理内存中放入数据总线上的内容。物理内存是以字节(8位)为单位编址的。

## 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分组成，段标识符，段内偏移量。段标识符是一个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。后面三位包含一些硬件细节。

索引号对应的“数组”就是段描述符表，段描述符具体描述了一个段地址，这样，很多段描述符就组成段描述符表。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。

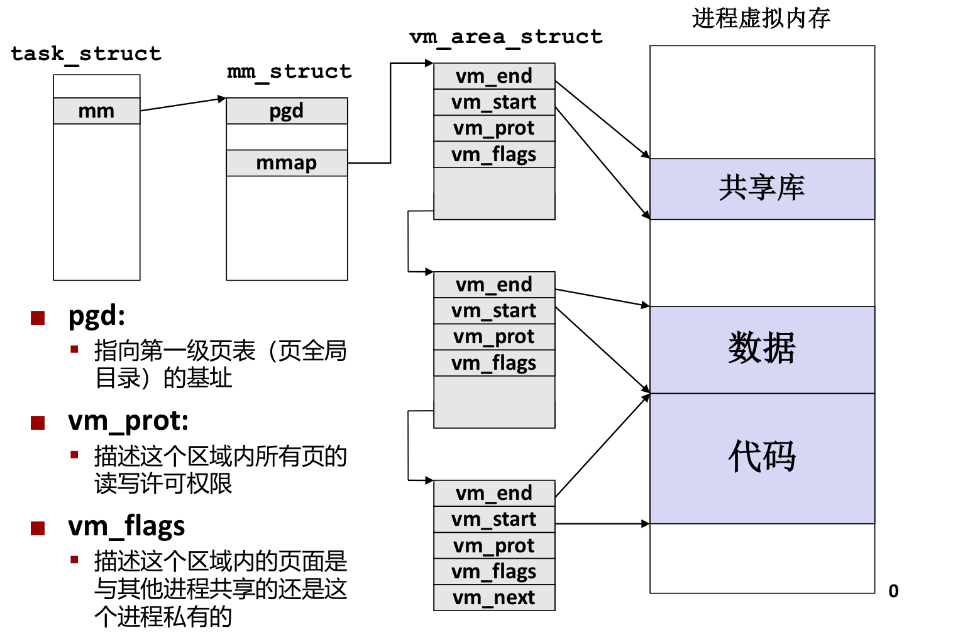
Base字段描述了一个段的开始位置的线性地址。

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。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。

首先，给定一个完整的逻辑地址[段选择符：段内偏移地址]，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。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，得到了Base，即基地址就知道了。

这样，线性地址=Base + offset。

## 7.3 Hello的线性地址到物理地址的变换-页式管理



7.3 页式管理

将各进程的虚拟空间划分成若干个长度相等的页(page)，页式管理把内页的划分存空间按页的大小划分成片或者页面（page frame），然后把页式虚拟地址与内存地址建立一一对应页表，并用相应的硬件地址变换机构，来解决离散地址变换问题。页式管理采用请求调页或预调页技术实现了内外存存储器的统一管理。

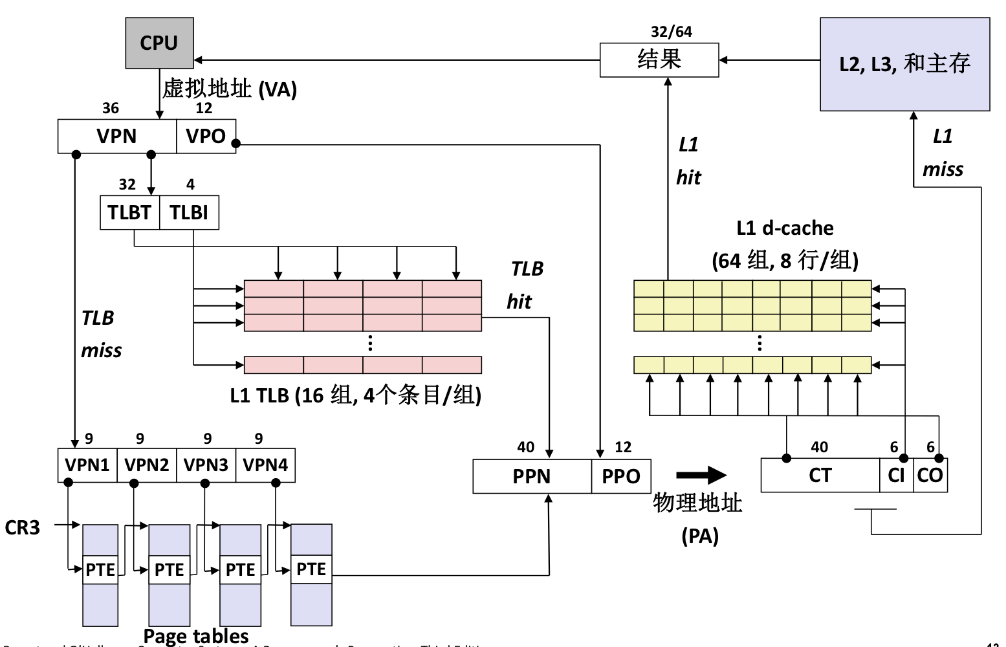
线性地址通过页表与物理地址映射，页表由内核维护，线性地址分为虚拟页号，虚拟偏移量，虚拟地址分为虚拟页号，虚拟偏移量。CPU取出虚拟页号，通过页表基址寄存器来定位页表条目，在有效位为1时，从页表条目中取出信息物理页号，通过将物理页号与虚拟页偏移量结合，得到由物理地址和物理页偏移量组合的物理地址。

## 7.4 TLB与四级页表支持下的VA到PA的变换

CPU发送一个虚拟地址，MMU向页表查阅一个PTE，用于将虚拟地址翻译成物理地址。可以引入TLB作为页表的缓存来加快这步的效率。TLB是一个小的、虚拟寻址的缓存，其中每一行都保存着一个由单一PTE组成的块。TLB通常有高的相联度，从虚拟地址中的页号提取出组选择和行匹配的索引和标记字段。

多级页表：将虚拟地址的VPN划分为相等大小的不同的部分，每个部分用于寻找由上一级确定的页表基址对应的页表条目。

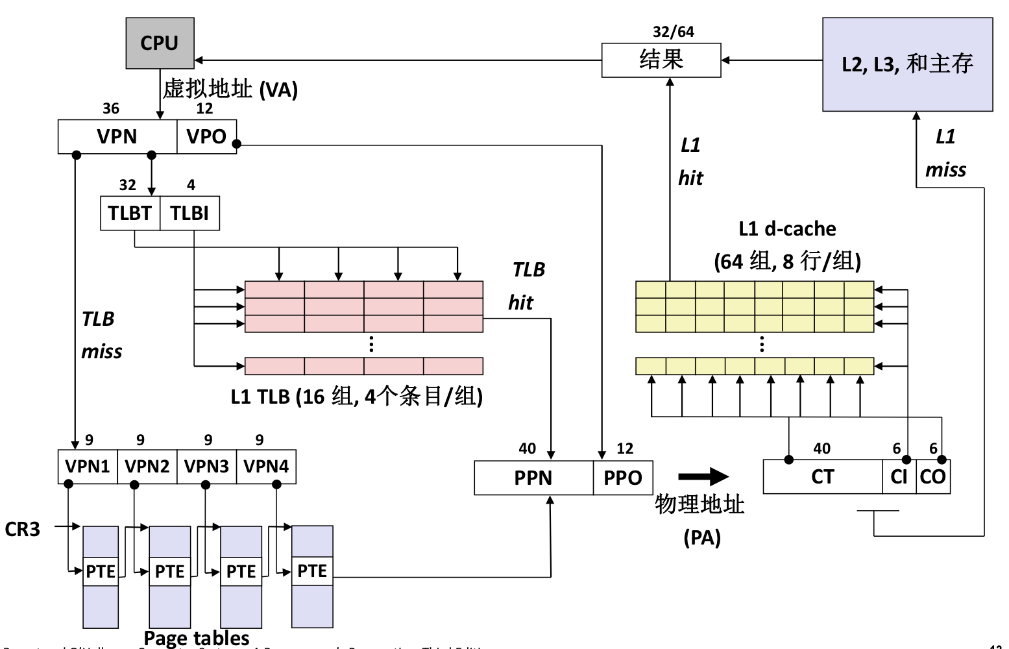
Core i7是四级页表进行的虚拟地址转物理地址。将48位虚拟地址的前36位分为四个页表的偏移量，一级页表基址放于CR3中，每一级页表指向下一个页表的基址，通过36位的偏移地址确定每一个页表的偏移量。



7.4 core i7截图

## 7.5 三级Cache支持下的物理内存访问

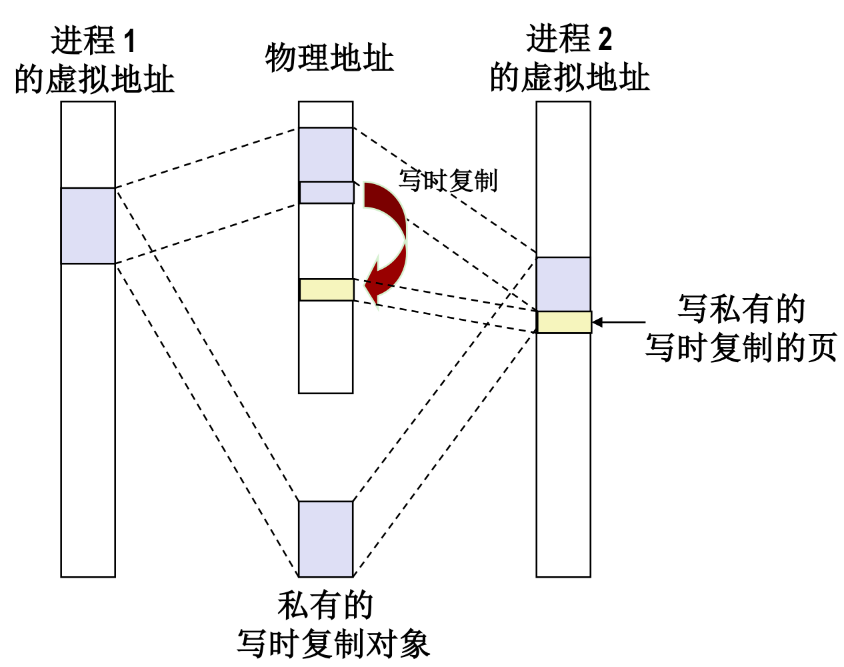
CPU发送一条虚拟地址，MMU首先去TLB中寻找，若命中，MMU便获得物理页号生成物理地址，若不命中，MMU结合多级页表得到物理地址，直接去L1中寻找，若不命中，则再在下一级L2找，若还不命中，则在L3中找，若还未找到，则直接访存。



7.5 core i7截图

## 7.6 hello进程fork时的内存映射

当shell调用fork时，内核为子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hello进程中有写操作时，由于私有写时复制，它会触发一个异常，内核会调度相应的机制去为hello再分配一块物理内存，进行写操作。



7.6 fork内存映射截图

## 7.7 hello进程execve时的内存映射

（1）删除已存在的用户区域。删除当前进程（shell）虚拟地址的用户部分中的已存在的区域结构， 映射私有区域。

（2）为hello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。栈和堆区域也是请求二进制零的，初始长度为零。映射共享区域。如果hello程序与共享对象（或目标）链接，比如标准C 库libc. so,那么这些对象都是动态链接到这个程序的，然后再映射到用户虚拟地址空间中的共享区域内。

（3）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中的程序计数器，使之指向代码区域的入口点。

## 7.8 缺页故障与缺页中断处理

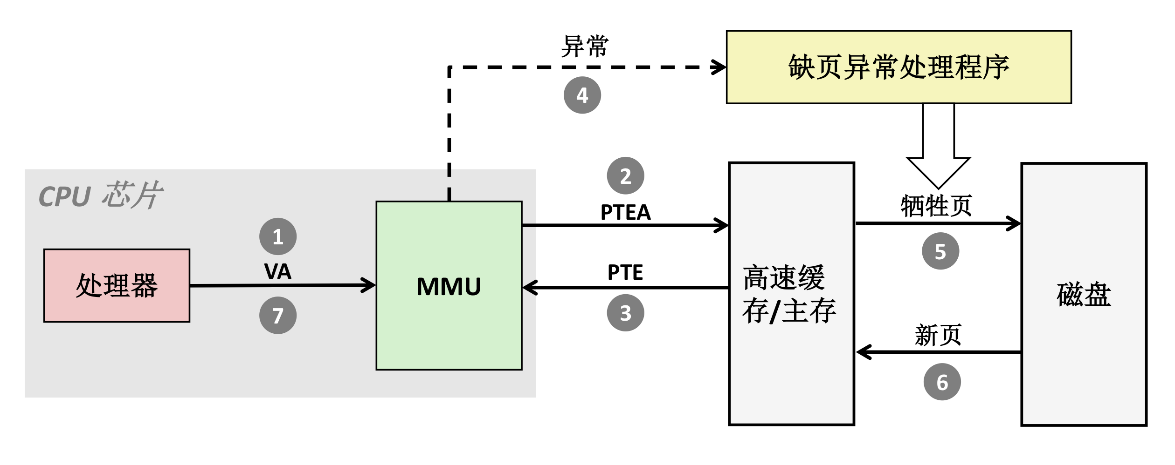
1)处理器将虚拟地址发送给MMU

2-3)MMU使用内存中的页表生成PTE地址4)有效位为零,因此MMU触发缺页异常

5)缺页处理程序检查虚拟地址是否合法，访问权限是否合法，若均合法，确定物理内存中牺牲页(若页面被修改，则换出到磁盘)

6)缺页处理程序调入新的页面，并更新内存中的PTE

7)缺页处理程序返回到原来进程,再次执行导致缺页的指令



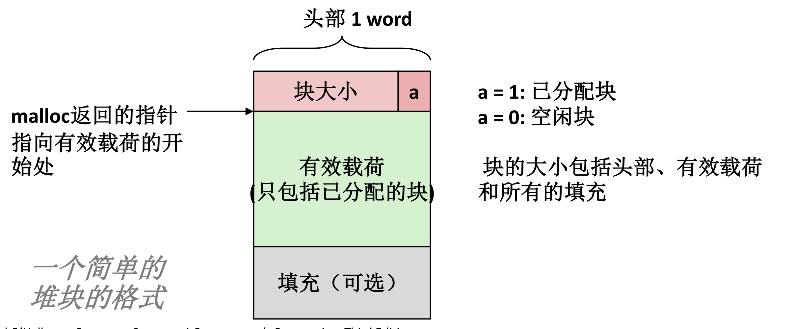
7.8 缺页异常截图

## 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(heap)。堆向上增长，内核维护的brk，指向堆顶部。

分配器：显示分配器（显示释放分配块）和隐式分配器（垃圾收集器）。

1. 隐式空闲链表



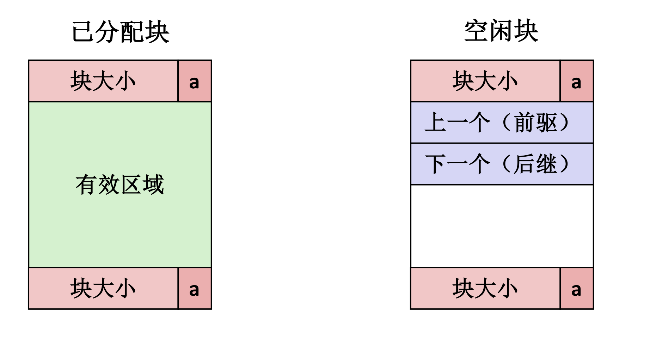
7.9.1 隐式空闲链表块结构

在隐式空闲链表中，因为空闲块是通过头部中的大小字段隐含地连接着的。

分配器可以通过遍历堆中所有的块，从而间接地遍历整个空闲块的集合。其中，一个设置了已分配的位而大小为零的终止头部将作为特殊标记的结束块。

当一个应用请求一个k字节的块时，分配器搜索空闲链表，查找一个足够大的可以放置所请求块的空闲块。分配器有三种放置策略：首次适配、下一次适配，最佳适配。分配完后可以分割空闲块减少内部碎片。同时分配器在面对释放一个已分配块时，可以合并空闲块，其中便利用隐式空闲链表的边界标记来进行合并。

1. 显示空闲链表



7.9.2 显式空闲链表块结构

显式空闲链表是将空闲块组织为某种形式的显式数据结构。因为根据定义，程序不需要一个空闲块的主体，所以实现这个数据结构的指针可以存放在这些空闲块的主体里面。如，堆可以组织成一个双向链表，在每个空闲块中，都包含一个前驱与一个后继指针。

在显式空闲链表中。可以采用后进先出的顺序维护链表，将最新释放的块放置在链表的开始处，也可以采用按照地址顺序来维护链表，其中链表中每个块的地址都小于它的后继地址，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。

## 7.10本章小结

本章介绍了hello的内存管理，介绍了虚拟地址和物理内存之间的关系，基本了解段式管理，页式管理的不同，fork和execve的内存映射方法，介绍了缺页故障等。

**（第7章 2分）**

# 第8章 hello的IO管理

## 8.1 Linux的IO设备管理方法

所有的I/O设备（例如网络、磁盘和终端）都被模型化为文件。

Unix I/O：而所有的输入和输出都被当做对相应文件的读和写来执行，这种将设备优雅地映射为文件的方式，允许Linux内核引出一个简单低级的应用接口。

设备的模型化：文件

设备管理：unix io接口

## 8.2 简述Unix IO接口及其函数

设备可以通过Unix I/O接口被映射为文件，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：

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。应用程序只需记住这个描述符。

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（描述符为0）、标准输出（描述符为1）和标准错误（描述符为2）。

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

4.读写文件。 一个读操作就是从文件复制n>0个字节到内存，从当前文件位置k开始，然后将k增加到k+n。给定一个大小为m字节的而文件，当k>=m时执行读操作会触发一个成为EOF的条件，应用程序能检测到这个条件。在文件结尾处并没有明确的“EOF符号”。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。 当应用完成了对文件的访问之后，它就通知内核关闭这个文件。作为响应，内核释放文件打开时创建的数据结构，并将这个描述符恢复到可用的描述符池中。无论一个进程因为何种原因终止时，内核都会关闭所有打开的文件并释放它们的内存资源。

 Unix I/O接口函数：

（1）进程是通过调用open函数来打开一个存在的文件或者创建一个新文件的，

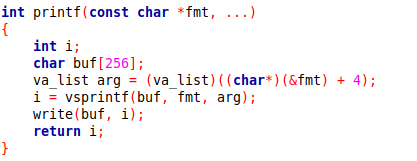
  open函数将filename转换为一个文件描述符，并且返回描述符数字。返回的描述符总是在进程中当前没有打开的最小描述符。flags参数指明了进程打算如何访问这个文件。mode参数指定了新文件的访问权限位。作为上下文的一部分，每个进程都有一个umask，它是通过调用umask函数来设置的。当进程通过带某个mode参数的open函数调用来创建一个新文件时，文件的访问权限位被设置成mode&~umask。

（2）进程通过调用close函数关闭一个打开的文件。

（3）应用程序是通过分别调用read和write函数来执行输入和输出的。read函数从描述符为fd的当前文件位置赋值最多n个字节到内存位置buf。返回值-1表示一个错误，0表示EOF，否则返回值表示的是实际传送的字节数量。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。

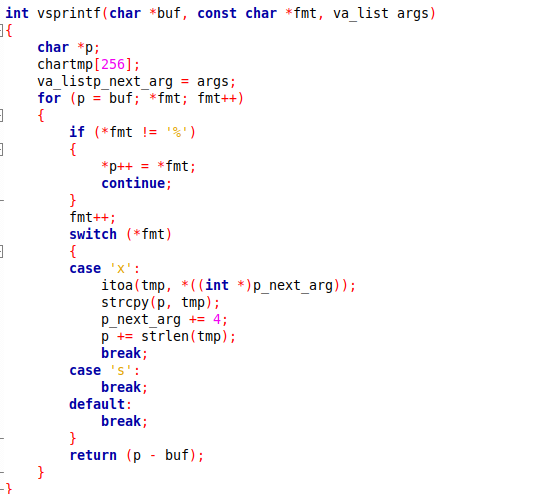
（4） 通过调用lseek函数，应用程序能够显式地修改当前文件的位置。

## 8.3 printf的实现分析



8.3.1 printf源码

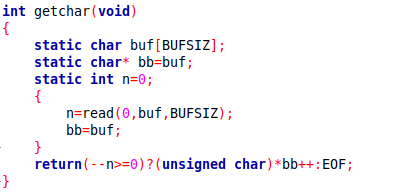
C语言中，参数压栈的方向是从右往左。 当调用printf函数的时候，先是最右边的参数入栈。 fmt是一个指针，这个指针指向const char \*fmt中的第一个元素。 fmt也是个变量，它的位置，是在栈上分配的。buf相当于缓冲行。



8.3.2 vprintf源码

vsprintf函数将所有的参数内容格式化。write函数将buf中的i个元素写到终端。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等。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



8.4 getchar源码

getchar返回值为int型。当程序调用getchar时，程序等待用户按键，用户输入的字符被存放在键盘缓冲区中直到用户按回车为止。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第一个字符的ASCII码,将用户输入的字符回显到屏幕。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,等待下一个getchar调用。

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

本章介绍了linux下IO设备管理方法，Unix IO接口及函数，printf函数，getchar函数的实现分析。

**（第8章1分）**

# 结论

用计算机系统的语言，逐条总结hello所经历的过程。

hello.c源程序经过预处理（解析带#的指令），生成hello.i文件，然后编译，将hello.i汇编生成hello.s文件，然后汇编，将hello.s文件转化成二进制可重定位目标文件hello.o，最后链接生成可执行程序hello。

执行hello：通过在命令行输入命令，shell解析命令并创建子进程，该子进程用execve函数加载hello程序，将hello映射到虚拟内存空间。内存映射完毕后，pc执行程序第一条指令。CPU发送一个虚拟地址给MMU，MMU解析该虚拟地址，相应物理地址中取出相应指令，发给CPU。在执行程序过程中，异常处理信号机制为hello提供了异常信号处理的方法，使得程序平稳运行。

hello运行结束后，shell回收子进程，内核删除数据结构。

一个小小的hello程序就需要操作系统，硬件和软件的紧密配合，这里面的逻辑性值得我们深思，值得我们感叹，计算机系统设计逻辑十分紧密，十分巧妙。计算机系统这本书以程序员的视角带我们深入的理解计算机系统，对计算机了解再也不局限于编程，而是更深层次的了解，为我们打开了新世界的大门。

**（结论0分，缺失 -1分，根据内容酌情加分）**

# 附件

列出所有的中间产物的文件名，并予以说明起作用。

hello.c:源程序

hello.i:预处理后生成的文件

hello.s:编译后生成文件

hello.o:汇编后生成的可重定位目标文件

hello:链接后的可执行目标文件

hello1.txt:hello.o的elf格式

hello2.txt:hello的elf格式

objhello1.txt:hello.o的反汇编

objhello2.txt:hello的反汇编

# 参考文献

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

[8] CSDN博客 Ubuntu系统预处理、编译、汇编、链接指令

[9]《计算机系统基础》袁春风

[10]CSDN[转]printf 函数实现的深入剖析