

**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 1170300823

班　　 级 1703008

学 生 陈伟伟

指 导 教 师 郑贵滨

**计算机科学与技术学院**

**2018年12月**

**摘 要**

本文介绍了在Linux环境下，以hello.c文件从progarm到process的过程为依据，系统编译c文件的工作环节，包括预处理、编译、汇编、链接等四个方面。同时在hello变为可执行程序时，介绍了hello的进程管理，数据的存储管理和IO管理。 以深入理解计算机系统课本为主要参考资料， 通过 gcc 等工具进行实验，从而将课本知识落实、融会贯通，通过一个程序深入 挖掘知识点，对于学生对于课程的理解以及知识体系的升华有很大帮助。

**关键词：**进程；编译；链接；内存管理；IO管理

**（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

hello的P2P过程：通过键盘等输入代码得到hello.c文件。如何GCC编译器读取源程序。预处理器cpp将hello.c源文件处理生成hello.i，如何通过编译器ccl将hello.i生成汇编文件hello.s，之后经过汇编器as的处理，hello.s变成可重定位的目标文件，此时hello.o中的汇编语言变成了机器指令。最后通过连接器ld将hello.o变成可执行的目标程序hello。在shell中输入./hello，shell调用fork生成子进程，于是hello就从一个program变成了process，这就是hello的P2P过程

hello的020过程：hello经过P2P过程后变成了process。之后shell调用execve，在虚拟内存空间中给hello分配空间。运行时通过地址翻译把hello的虚拟地址翻译成物理地址， cpu更具汇编语言指令执行取指、译码、执行、更新等操作，形成软件硬件共同工作的一个整体。hello运行结束后，进程终止，内存回收，内核将与hello有关的数据清除，这就是hello的020过程。

## 1.2 环境与工具

硬件环境：Intel Core i7 16GRAM，256GSSD+1THDD

软件环境：windows10，Ubuntu18.04

开发与调试工具：codeblocks，notepad++，edb。

## 1.3 中间结果

|  |  |
| --- | --- |
| 中间结果文件 | 文件的作用 |
| hello.i | 预处理后生成的文件，加载了头文件和打开宏定义并实行条件编译 |
| hello.s | 编译之后的汇编文件，包含汇编语句 |
| hello.o | 汇编后的生成的可重地位目标文件，汇编语言被翻译成机器指令 |
| hello.elf | hello.o的ELF格式，可以查看节头，程序头符号表等等 |
| helloobjdump | hello.o可重定位目标文件的反汇编代码 |
| hellolink.elf | hello的ELF格式，可以查看节头，程序头符号表等等 |
| hellolink.objdump | 链接后hello的反汇编代码 |

|  |  |
| --- | --- |
| hello | 链接后的可执行程序 |

## 1.4 本章小结

本章简单介绍了hello的P2P和020过程，是对hello一生亦是本文的一个简单概括。除此之外还列举了环境工具以及编写过程中出现的中间文件。

**（第1章0.5分）**

# 第2章 预处理

## 2.1 预处理的概念与作用

预处理的概念：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。

作用:

1. 将所有的#define删除，并展开所有的宏定义；：将宏名替换为文本（这个文本可以是字符串、可以是代码等）。即在对相关命令或语句的含义和功能作具体分析之前就要换。例如：#define PI 3.1415926就会把程序中全部的标识符PI换成3.1415926

2. 添加行号信息文件名信息，便于调试；

3. 处理#include预编译指令，将被包含的文件插入到预编译指令的位置；。格式：#include "文件名"或#include <文件名>。

4. 处理所有的预编译指令，例如：#if,#elif,#else,#endif;

格式：

（1）#ifdef 标识符 程序段1，#else，只有标识符已经定义时，才参加编译。

（2）#ifndef 标识符 #define 标识1 程序段1#endif，如果标识符没有被定义，则重定义标识1，且执行程序段1.

（3）#if 表达式1 程序段1 #elif 表达式2….. 当表达式1成立时，编译程序段1，当不成立时，编译程序段2。使用条件编译可以使目标程序变小，运行时间变短。

## 2.2在Ubuntu下预处理的命令

cpp hello.c > hello.i 或 gcc -E hello.c -o hello.i

使用指令生成hello.i文件，打开hello.i文件，发现程序由原来的30行变为3000多行，行数大大增加

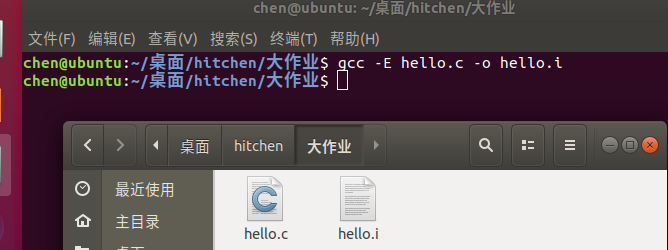


图2.1使用gcc命令生成hello.i文件

## 2.3 Hello的预处理结果解析

打开hello.i文件，发现程序由原来的30行变为3000多行，行数大大增加。int main出现在第3102行，如下图：

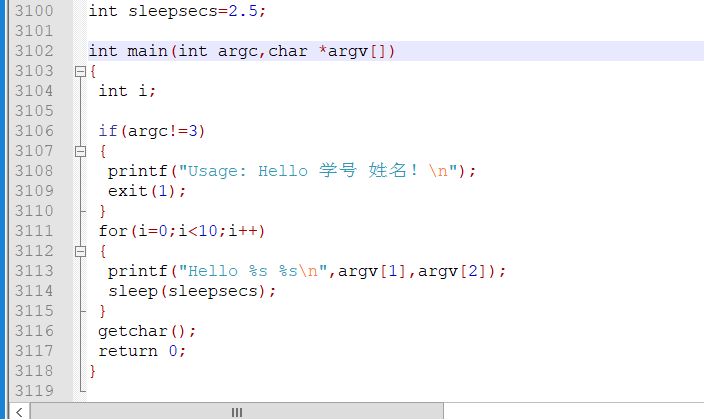


图2.2生成hello.i中main函数出现的位置

在main行数之前，在hello.i中依次处理#include预编译指令，将被包含的文件插入到预编译指令的位置。

一开始处理stdio.h文件



图2.2.1预处理中对头文件的展开

在打开stdio.h后，发现之后依旧打开了许多头文件。



图2.2.2预处理中对头文件的展开

应该是stdio.h中还有#define，预处理会继续打开头文件，直到没有#define出现。中间还出现了大量的typedef，对类型进行了重命名，如下图：

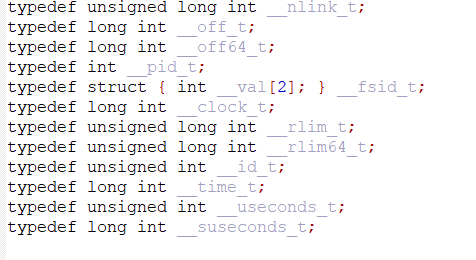


图2.3预处理中对类型的重命名

还有许多头文件中定义的文件指针，对应标准输入输出等等

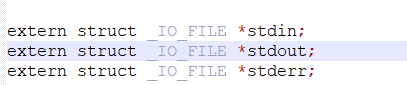


图2.4预处理中定义的各种文件指针

之后就有如上的类似的大量定义的指针，变量，数组。

总的来说，gcc先打开stdio.h然后发现里面还有#define，就继续打开，直到最后的文件中没有#difine为止。在这之间会有大量的typedef重命名，以及定义大量文件输入输出指针等等。

然后依次展开unistd.h，stdlib.h。期间的过程和stdio.h类似

## 2.4 本章小结

本章主要介绍了预处理的概念，作用功能。并且Ubuntu在通过gcc进行了预处理，通过阅读生成的hello.i文件，知道了gcc是如何打开头文件，解析指令的。

**（第2章0.5分）**

# 第3章 编译

## 3.1 编译的概念与作用

编译(compilation , compile) 1、利用编译程序从源语言编写的源程序产生目标程序的过程。 2、用编译程序产生目标程序的动作。 编译程序把一个源程序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；目标代码生成。主要是进行词法分析和语法分析，又称为源程序分析，分析过程中发现有语法错误，给出提示信息。

作用：

1.词性分析，对由字符组成的单词进行处理，从左至右逐个字符地对源程序进行扫描，产生一个个的单词符号，把作为字符串的源程序改造成为单词符号串的中间程序。

2.语法分析：以单词符号作为输入，分析单词符号串是否形成符合语法规则的语法单位，如表达式、赋值、循环等，最后看是否构成一个符合要求的程序，按该语言使用的语法规则分析检查每条语句是否有正确的逻辑结构，程序是最终的一个语法单位。

3.中间代码：可使编译程序的结构在逻辑上更为简单明确

4. 代码优化是指对程序进行多种等价变换，使得从变换后的程序出发，能生成更有效的目标代码。

5. 目标代码生成是编译的最后一个阶段。目标代码生成器把语法分析后或优化后的中间代码变换成目标代码。

## 3.2 在Ubuntu下编译的命令

gcc -S hello.c -o hello.s

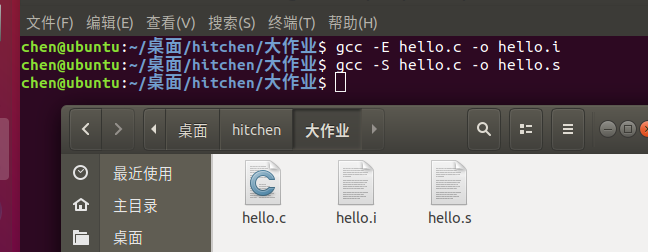


图3.1使用gcc命令生成的hello.s文件

## 3.3 Hello的编译结果解析

**3.3.1汇编伪指令**

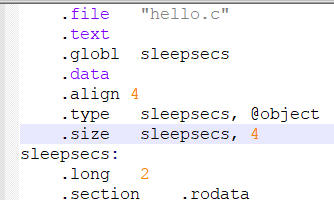


图3.2.1汇编文件中的伪指令

.s文件开头中有给出以“.”开头的行都是指导汇编器和连接器工作的伪指令，他们的作用如下：

|  |  |
| --- | --- |
| .file | 源文件 |
| .text | 保存代码，是只读和可执行的 |
| .globl | 告诉编译器，该符号会被连接器用到 |
| .data | 保存程序的数据，是可读可写的，相当于C程序中的全局变量 |
| .align | 在于对指令或者数据的存放地址进行对齐 |
| .type | 用来指定是函数还是对象 |
| .size | 声明大小 |
| .section | 把代码划分为若干个段，程序执行加载时，每个段被加载到不同地址，操作系统对不同的页面设置不同的权限 |
| .long .string | 声明变量的类型 |

**3.3.2数据**

hello.s中包含字符串，变量，常量，数组

1）字符串：

hello.s中出现的字符串有以下两个：

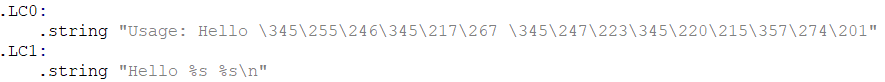


图3.3.2.1 hello.s中对.LC0,.LC1字符串的声明

① Usage: Hello \345\255\246\345\217\267 \345\247\223\345\220\215\357\274\201

对比原c程序中的Usage: Hello 学号 姓名！\n。可以看见汉字和感叹号变成了一堆以三个数字为一个编码的串。即将汉字转换为utf8码，在utf8码中，类似\345的为一个字节，三个字节表示一个汉字。

②"Hello %s %s\n"。这是第二个字符串。与原来c程序相比没有声明变化

2）变量

hello.s中出现的变量有三个，分别是sleepsecs，i，argc

①int sleepsecs：在原c程序中sleepsecs是一个全局变量且被赋值为2.5。编译时，伪指令.data保存程序的数据，相当于c中的全局变量，那么sleepsecs是在.data中的。然后.type中申明sleepsecs为object类型，.size设置其大小为4。sleepsecs下又有.long和.section .rodata。图片如下

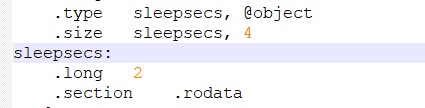


图3.3.2.2hello.s中对全局变量sleepsecs的声明

②int i :在原c程序中是一个局部变量，由汇编的知识可以知道，局部变量存储在寄存器或者栈中，在hello.s中寻找。得到如下：

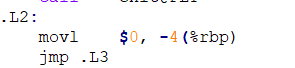




图3.3.2.3hello.s中的变量i

与原c程序的循环比较，知道一开始i赋值0，每次循环i加1，大于九时就退出循环，跳过getchar函数。所以i存储在-4（%rbp）中。

③int argc ：作为main函数传入的参数，会与常数3比较。



图3.3.2.4hello.s中的变量argc

由此图，argc存储在%edi中。

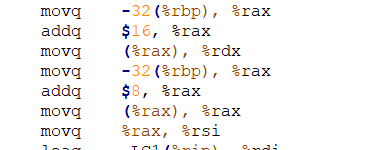
3）常量

原程序中出现的3，1等在hello.s中是立即数，与原程序中没有不同，直接出现在汇编代码中。

4）数组

main程序中有char \*argv[]数组，作为第二个参数。由局部性可知，数组元素应该存储在连续的空间中。；两次通过%rax取得存储数组argv的地址和值。

argv[1]地址传送给%rax



先获取argv[2]的地址，再获得值

获取argv[1]存储值

图3.3.2.5hello.s中的数组

其中argv[1] 存储地址为%rbx-16

argv[2]存储地址在%rbx-24

**3.3.3赋值：**

1）原程序中int sleepsecs=2.5，在.s文件的头部声明如下：

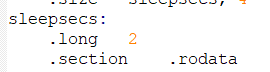


图3.3.3.1hello.s中sleppsecs的值

sleepsecs为全局变量，在此节中，直接将sleepsecs赋值为2的long类型数据

2）变量i的赋值：

通过movl指令，对i进行赋值



图3.3.3.2hello.s中变量i的赋值

**3.3.4 类型转换**

隐式转化：在.c文件中，sleepsecs为int类型，int sleepsecs=2.5，将浮点数类型的2.5转换为int类型。当double或float类型向int类型转换时，值会向0舍入。所以在.s文件中，sleepsecs的值变为2。

**3.3.5算术操作**

汇编中的算术指令有如下几种：



图3.3.5汇编中的算术指令

程序中涉及的有如下几种：

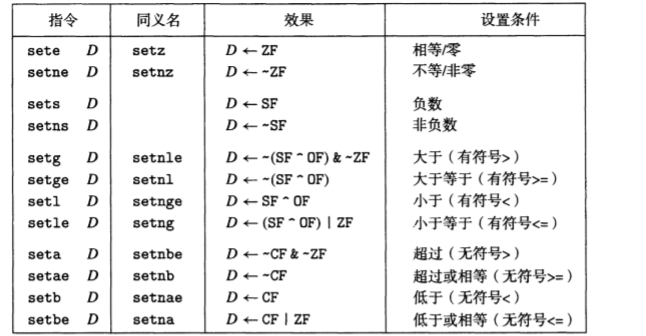
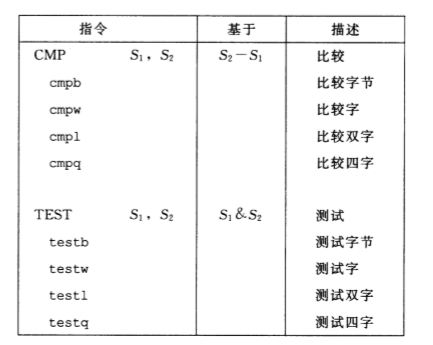
1）i++，对i进行加1操作，.s文件中中用了如下指令： addl $1, -4(%rbp)

l表示字节数

2）加载有效地址：.s文件中使用leaq来计算LC0和LC1的地址，并且将结果村春到%rdi中。涉及指令: leaq .LC0(%rip), %rdi leaq .LC1(%rip), %rdi

**3.3.6关系操作**

汇编中的关系操作指令有：



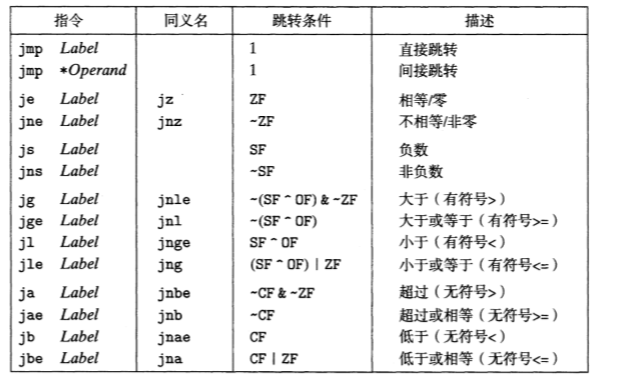


图3.3.4汇编中的操作指令

程序中涉及到的有：

1）argc！=3：判断argc是否等于3，在.s文件中用

cmpl $3, -20(%rbp)

je .L2

两条指令来完成。首先计算argc-3，得到条件码，为之后的je指令提供条件码

2）i<10：判断i是否小于10，在.s文件中用

cmpl $9, -4(%rbp)

jle .L4

两条指令来完成。首先也是计算i-9，得到条件码，为之后的jle指令提供条件码。

**3.3.7数组/指针/结构操作**

数组

main程序中有char \*argv[]数组，作为第二个参数。由局部性可知，数组元素应该存储在连续的空间中。；两次通过%rax取得存储数组argv的地址和值。

其中argv[1] 存储地址为%rbx-16

argv[2]存储地址在%rbx-24。

**3.3.8控制转移**

编译器将if，for等控制语句都用cmp来比较，然后使用条件跳转指令来决定接下来代码执行。

1）if（argv！=3）:当argv不等于三时才会执行if内的语句。

对于if 内的判断，汇编首先计算argc-3，得到条件码，为之后的jle指令提供条件码，相等时就跳转到L2，不相等执行接下来的命令

汇编代码如下（带注释）：

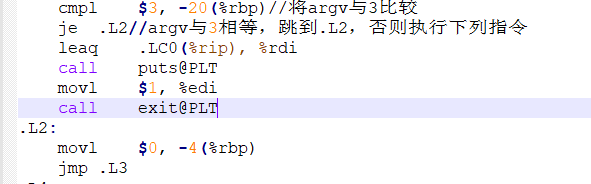


图3.3.5.1hello.s中对if语句的编译

2）for（i=0;i<10;i++）:在.L4最后，i++，然后进入.L3，首先也是计算i-9，得到条件码，为之后的jle指令提供条件码。小于9的话就跳到.L4，否则执行call getchar语句。

汇编代码如下：

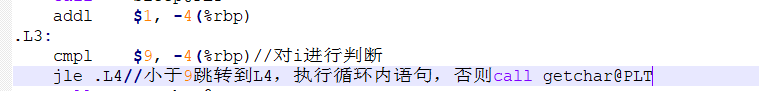


图3.3.5.2hello.s对for语句的编译

**3.3.9函数操作**

参数传递：函数的参数传递是一个参数压栈的过程。函数的所有参数，都会依次被push到栈中。参数总是从右到左压栈，也就是最后一个参数先入栈。

函数调用：

1）压栈: 函数参数压栈，返回地址压栈

2）跳转: 跳转到函数所在代码处执行

3）执行: 执行函数代码

4）返回: 平衡堆栈，找出之前的返回地址，跳转回之前的调用点之后，完成函数调用

程序中涉及的函数调用：

1）puts函数：

①参数传递：将%rdi设置为.LC0（%rip）,即第一个字符串的首地址。

函数调用：call puts@PLT

2）exit函数：

①参数传递：将%edi设置为1

②函数调用：call exit@PLT

3）printf函数：

①参数传递：将%rdi设置为.LC1（%rip），即第二个字符串的首地址。%eax设置为0

②函数调用：call printf@PLT

4）sleep函数：

①参数传递：将%edi设置为sleepsecs

②函数调用：call sleep@PLT

5）getchar函数：

没有参数传递，直接调用

6）main函数：

①参数传递：向main函数传递argv和argc。

②返回，平衡堆栈。通过%rbp保存原来栈起始位置，即栈底。程序的最后，通过leave指令恢复栈（leave相当于mov %rbp，%rsp。pop%rbp）。然后调用ret指令，ret即pop IP。即得到下一条要执行指令的地址。

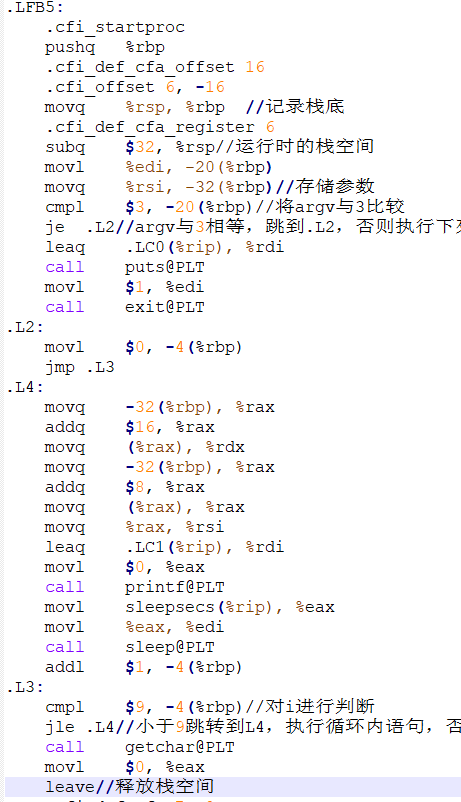


图3.3.6hello.s中main函数的解析

## 3.4 本章小结

编译(compilation , compile) 1、利用编译程序从源语言编写的源程序产生目标程序的过程。 2、用编译程序产生目标程序的动作。

编译器将.i 的拓展程序编译为.s 的汇编代码。经过编译之后，我们的 hello 自 C 语言解构为更加低级的汇编语言。汇编语言是直接面向处理器（Processor）的程序设计语言。处理器是在指令的控制下工作的，处理器可以识别的每一条指令称为机器指令。

本章主要介绍了编译器是怎么处理C语言中的各种数据，包括常量变量数组等等。接着解释了c中的赋值，类型转换，算术操作，关系操作，数组/指针操作，控制转移，函数操作等在汇编中是如何实现的。

**（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hello.s 翻译成机器语言指令，把这些指令打包成一种叫做可重定位目标程序(relocatable object program) 的格式，并将结果保存在目标文件hello.o 中。

## 作用：汇编器是将汇编代码转变成机器可以执行的命令，每一个汇编语句几乎都对应一条机器指令。.o文件是一个二进制文件，它包含程序的指令编码。

## 4.2 在Ubuntu下汇编的命令

as hello.s -o hello.o 或gcc -c hello.s -o hello.o

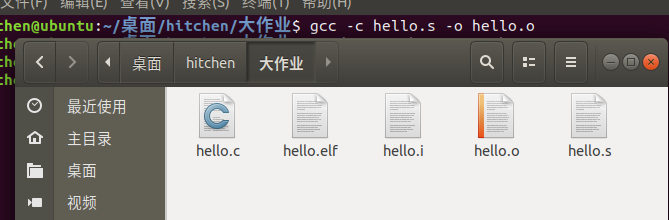


图4.1使用gcc命令生成hello.o文件

## 4.3 可重定位目标elf格式

使用readelf -a hello.o > hello.elf 指令获得 hello.o 文件的 ELF 格式，打开得到以下几个部分：

1）ELF头：

ELF头部，是用来描述整个文件的组织。

最开头是16个字节的e\_ident, 其中包含用以表示ELF文件的字符，以及其他一些与机器无关的信息。开头的4个字节值固定不变，为0x7f和ELF三个字符。之后ELF头介绍了ELF的类别，数据，版本，程序头节点，表头节点，大小，节头大小等等一系列有关的信息

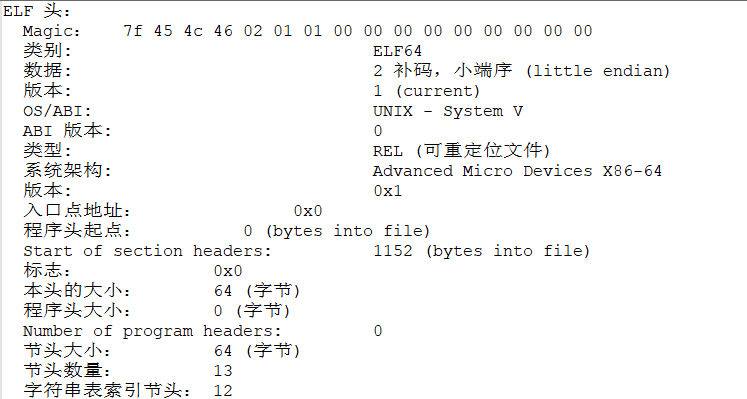


图4.3.1 ELF格式中的ELF头

2）节头：

包含了描述文件节区的信息，每个节区在表中 都有一项，每一项给出诸如节区名称、节区大小，地址，偏移量等信息。

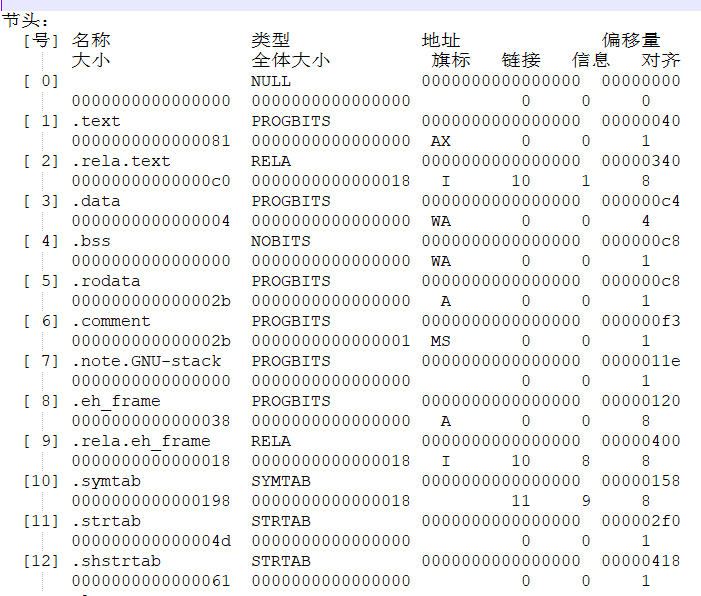


图4.3.2 ELF格式中的节头

3）重定位节：

重定位节中包含了需要重定位的信息。该程序包含八个需要重定位的信息，分别是两个字符串，puts、exit、printf、sleep、getchar函数以及sleepsecs变量。

重定位节中包含以下信息：

offset是需要改的引用的节偏移

symbol标识被修改应该指向的符号。

type告知链接器如何修改新的引用。

attend是一个有符号常数，一些类型的重定位要使用它对被修改引用的值做偏移调整。

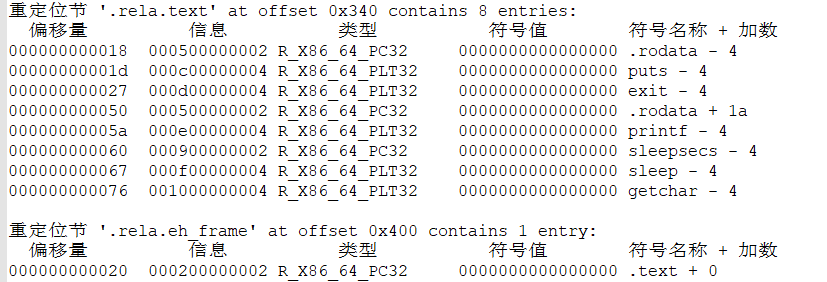


图4.3.3 ELF格式中的重定位节

重定位引用分为两种：

①重定位PC相对引用

refaddr=ADDR（s）+r.offset;

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

在得到的可执行目标文件中，call指令的机械码后面会跟着\*refptr的值

②重定位PC绝对引用

\*refptr=(unsigned)(ADDR(r.symbol)+r.addend);

4）符号表：符号表保存了查找程序符号、为符号赋值、重定位符号所需的全部信息。符号的主要任务是讲一个字符串和一个值关联起来。

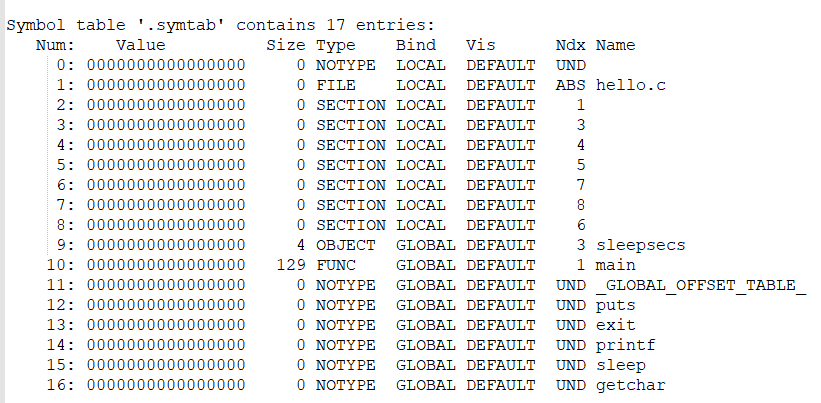


图4.3.4ELF各值中的符号表

## 4.4 Hello.o的结果解析

使用objdump -d -r hello.o >helloobjdump指令得到反汇编代码。

机器语言指的是二进制的机器指令集合，而机器指令是由操作码和操作数构成的。汇编语言的主体是汇编指令。汇编指令和机器指令的差别在于指令的表示方法上，汇编指令是机器指令便于记忆的书写格式。

对照hello.s得到如下几点不同：

1）分支转移：反汇编代码跳转指令不像hello.s文件中有许多.L2,.L3.L4等段，

然后通过jmp指令后面跟着段名称跳转，而是直接通过jmp指令后面跟地址来进行跳转。

图4.4.1hello.i中分支转移指令

图4.4.2hello.o中分支转移指令

2）函数调用：：在.s 文件中，函数调用是由call之后直接跟着函数名称，而在反汇编程 序中，call 的目标地址是当前下一条指令。这是因为 hello.c 中调用的函数最终需要通过动态链接器才能确定函数的运行时执 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，然后在重定位节中为其添加重定位条目，等待链接过程。

下面给出对比图



图4.4.3hello.o与hello.s的调用函数对比

3）对全局变量的访问：.s文件中，访问字符串是通过类似.LC0(%rip)这种形式，使用段名称和%rip，并存储在%rdi中来实现的。访问sleepsecs变量也是通过

movl sleepsecs(%rip), %eax，将值赋给%eax。在反汇编代码中，则是通过0x0（%rip）来访问的，在上文中已经知道各全局变量都处于重定位节中，地址不确定，所以访问时也需要等待链接后才能确定地址，因此用0来代替。

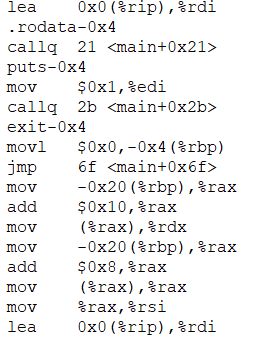
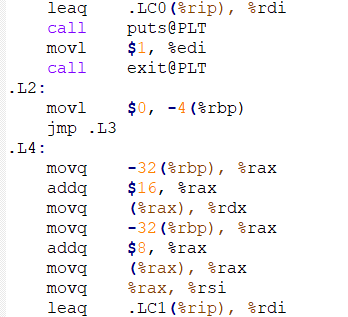


图4.4.4hello.s与hello.o对全局变量访问的对比

## 4.5 本章小结

本章介绍了 hello 从 hello.s 到 hello.o 的汇编过程，通过查看 hello.o 的 elf 格式 和使用objdump -d -r hello.o >helloobjdump指令得到反汇编代码。将得到反汇编代码与 hello.s 进行比较，发现了，机器语言中的操作与汇编语言由很大不同，特别是分支转移函数调用已经对全局变量的访问。从而也知道了机器语言的构成，与汇编语言的映射关系。

**（第4章1分）**

# 第5章 链接

## 5.1 链接的概念与作用

链接是将各种代码和数据片段收集并组合称为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。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。

作用：使得分离编译成为可能；动态绑定(binding):使定义、实现、使用分离

## 5.2 在Ubuntu下链接的命令

命令：

## ld -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 /usr/lib/gcc/x86\_64-linux-gnu/7/crtbegin.o /usr/lib/gcc/x86\_64-linux-gnu/7/crtend.o /usr/lib/x86\_64-linux-gnu/crtn.o hello.o -lc -z relro -o hello

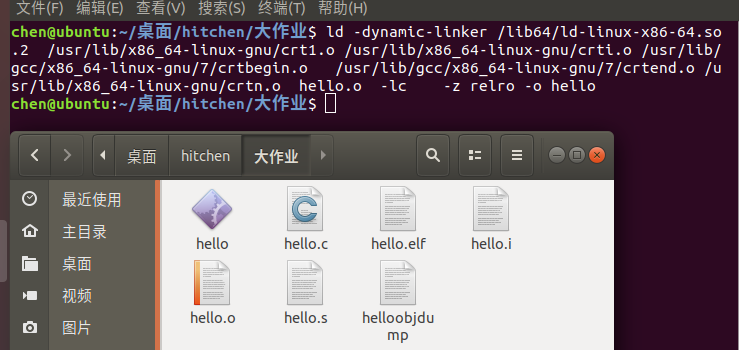
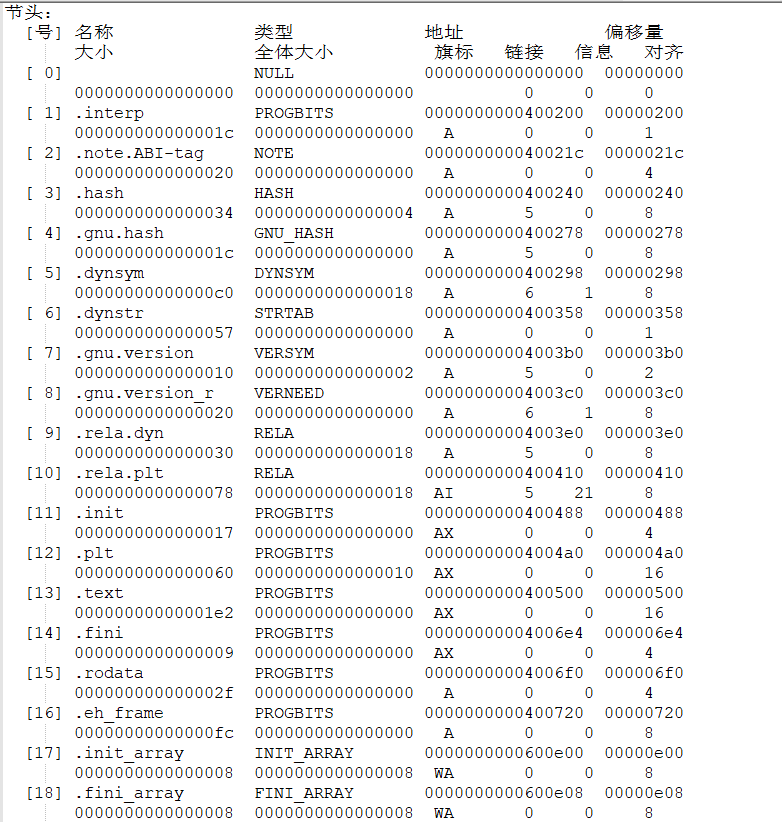


图5.1 使用ld链接命令生成可执行程序hello

## 5.3 可执行目标文件hello的格式

1）在ELF格式文件中，节头列出了hello中所有的节的各项信息，包括：名称，大小，类型，全体大小，地址，偏移量等等。



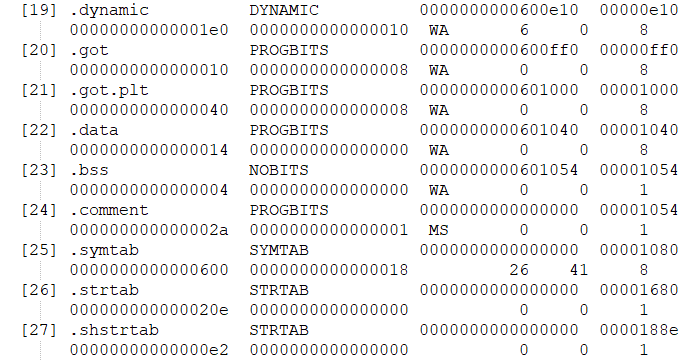


图5.2.1 hello节头

2）程序头中提供了各段在虚拟地址空间和物理地址空间的位置、大小、标志、访问授权和对齐方面的信息。

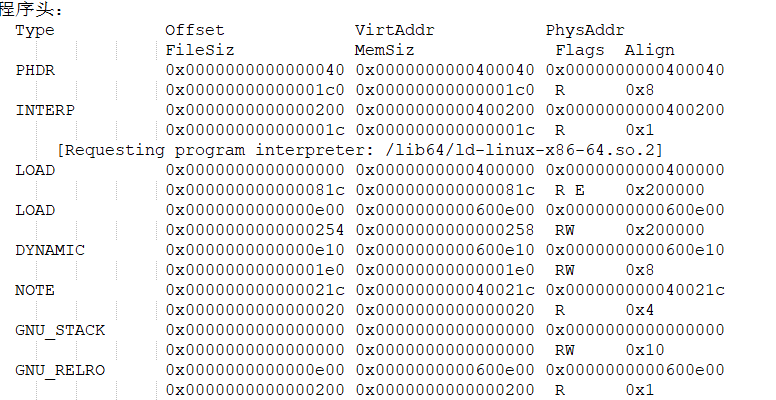


图5.2.2 hello的程序头

该程序头列出了八个段，这些段组成了最终在内存中执行的程序。

PHDR保存程序表头

INTERP指定在程序已经从可执行映射到内存之后，必须调用解释器。在这里解释器并不意味着二进制文件的内存必须由另一个程序解释。它指的是这样的一个程序：通过链接其他库，来满足未解决的引用。

LOAD表示一个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），程序的目标代码等等。

DYNAMIC段保存了其他动态链接器（即，INTERP中指定的解释器）使用的信息。NOTE保存了专有信息。

3）符号表：符号表保存了查找程序符号、为符号赋值、重定位符号所需的全部信息。符号的主要任务是讲一个字符串和一个值关联起来。

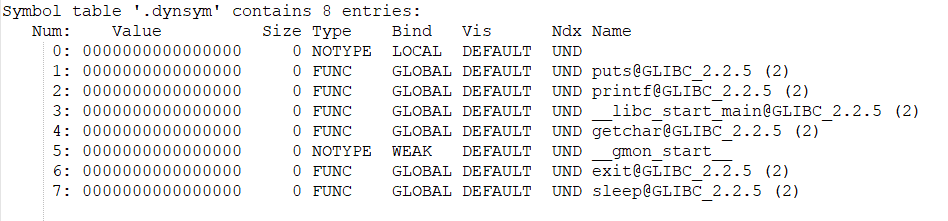


图5.2.3 hello的中与动态链接相关的符号表

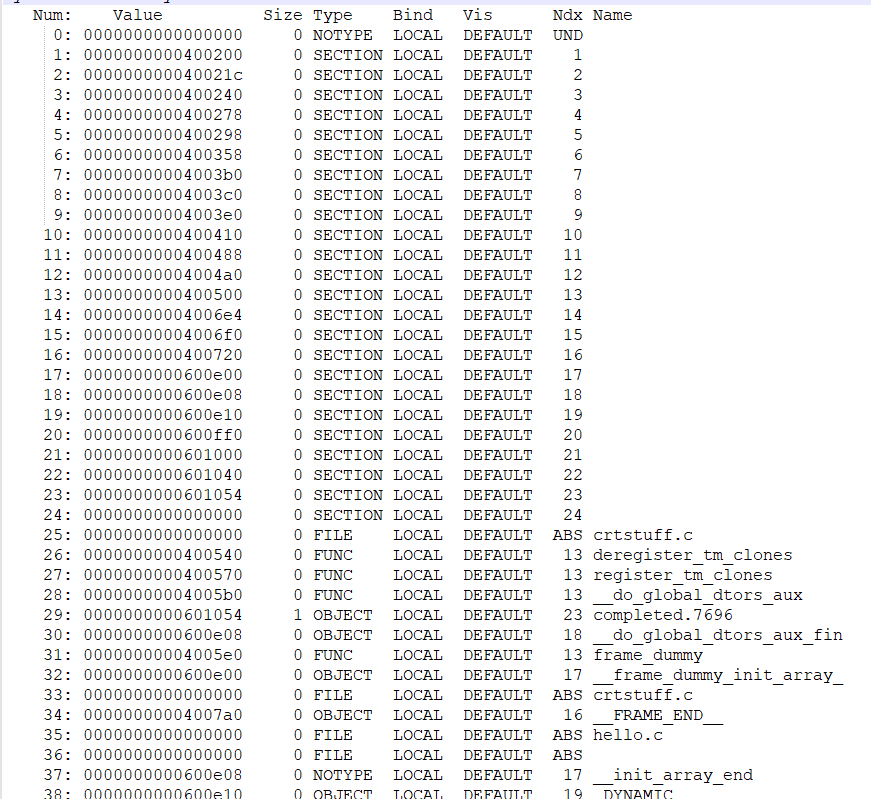


图5.2.4hello中的全部符号表

## 5.4 hello的虚拟地址空间

使用edb加载hello，如图所示

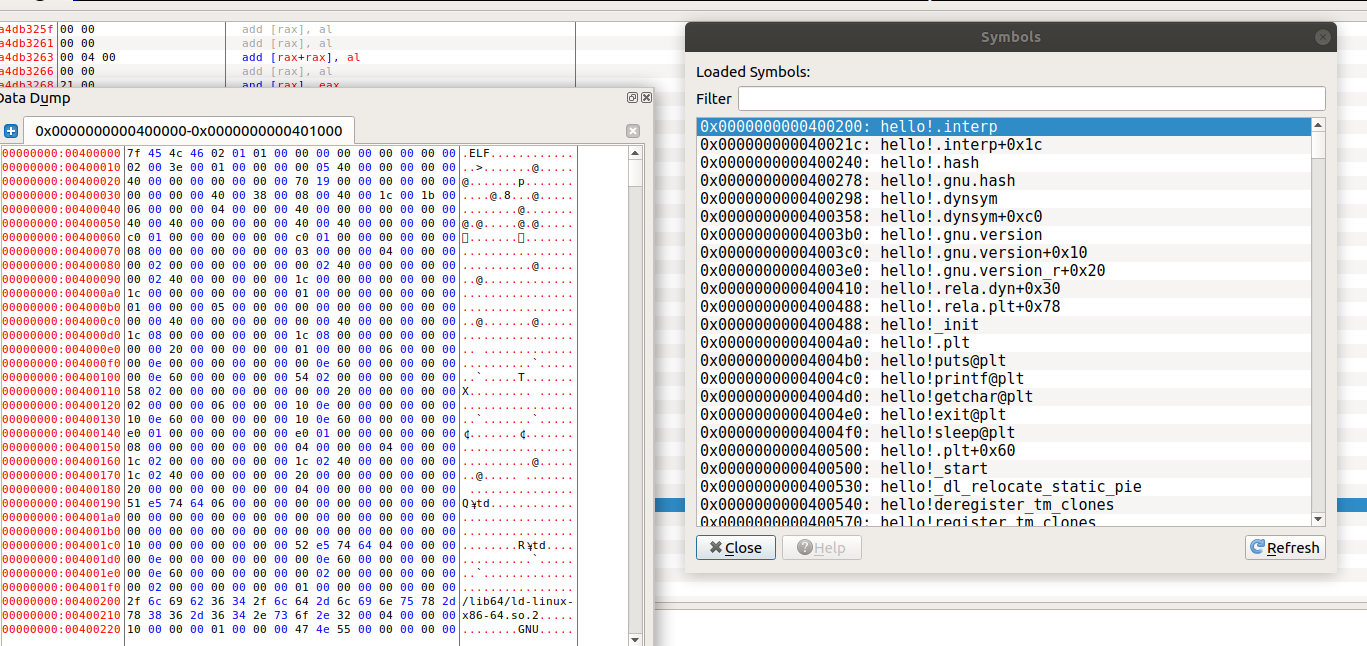


图5.3hello中的虚拟地址空间

各段虚拟空间:

其中data dump 有详细的数据段的信息，symbols中对于着5.3中节头的名称。

从虚拟地址0x400000开始，程序的各部分载入，各个节通过5.3中的信息到指定虚拟地址中，载入的顺序与5.3中节头出现的顺序相同。例如0x400200中是节头中的第一个，名称为.interp，在data dump中也是第一个载入的。

## 5.5 链接的重定位过程分析

hello相对于hello.o有如下不同：

1）hello.o中的相对偏移地址在hello中变成了虚拟内存地址

2）hello中比hello.o出现了很多外部的函数

3）hello比hello.o多了很多节，具体如下：

①.init：程序初始化所需要执行的代码

②.plt：链接过程表

③.fini:程序正常终止时所需要执行的代码。

4）hello.o中跳转以及函数调用的地址在hello中都被更换成了虚拟内存地址。

通过以上几个不同，链接的过程应该有如下几步：

1）从库中添加外部函数：在使用 ld 命令链接的时候，主要定义了初始化函数\_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。链接器将上述函数加入

2）函数调用：

此时动态链接库中的函数已经加入到了 PLT 中， .text 与.plt 的相对距离已经确定，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 中函数与下条指令的相对地址，指向对应函数。

3）.rodata引用：

在hello.o的反汇编中可以看见有两个R\_X86\_64\_PC32类型数据的重定位，（其实是printf中的两个字符串）。因为两个节之间的相对距离确定，所以在hello的反汇编中，call之后变成了目标地址与当前的下一条地址差值（hello.o的反汇编中全为0）。下面举例说明：

字符串：

以第一个字符串为例

ADDR(s)=ADDR(main)=0x4005e7

ADDR(r.symbol)=0x4006f4

refaddr=ADDR(s)+r.offset=0x4005e7+0x18=0x4005ff

\*refptr=(unsigned)(ADDR(r.symbol)+r.addend-refaddr)=(0x4006f4-4-0x4005ff)=0xf1



函数：

以puts为例

ADDR(s)=ADDR（main）=0x4005e7

ADDR(r.symbol)=0x4004b0

readdr=ADDR(s)+r.offset=0x400604

\*refptr=(unsigned) (ADDR(r.symbol)+r.addend-refaddr)=(0x4004b0-4-0x400604)=0xfffffea8



其余函数和字符串与上述算法类似

## 5.6 hello的执行流程

使用edb，观察函数执行流程，其中涉及的函数有：

|  |  |
| --- | --- |
| 函数名称 | 地址 |
| <ld-2.27.so!\_dl\_start+0> | 0x7f4ea4db4ea0 |
| hello!\_init | 0x0000000000400488 |
| hello!puts@plt | 0x00000000004004b0 |
| hello!printf@plt | 0x00000000004004c0 |
| hello!getchar@plt | 0x00000000004004d0 |
| hello!exit@plt | 0x00000000004004e0 |
| hello!sleep@plt | 0x00000000004004f0 |
| hello!\_start | 0x0000000000400500 |
| ld-2.27.so!\_\_assert\_fail | 0x00007f8a388bb790 |
| ld-2.27.so!\_\_GI\_\_\_tunables\_init | 0x00007f8a388b8c50 |
| ld-2.27.so!\_\_libc\_check\_standard\_fds | 0x00007f8a388bc6c0 |
| ld-2.27.so!\_\_strerror\_r | 0x00007f8a388bb670 |
| ld-2.27.so!\_\_tunable\_get\_val | 0x00007f8a388b9250 |
| ld-2.27.so!\_dl\_add\_to\_namespace\_list | 0x00007f8a388ac000 |
| ld-2.27.so!\_dl\_cache\_libcmp | 0x00007f8a388b7f70 |
| ld-2.27.so!\_dl\_debug\_printf | ld-2.27.so!\_dl\_debug\_printf |

## 5.7 Hello的动态链接分析

hello在调用.so共享库函数时，会涉及到动态链接。现代系统在处理共享库在地址空间中的分配的时候，采用了位置无关代码(PIC)方式。位置无关代码指，编译共享模块的代码段，是把它们加载到内存的任何位置而无需链接器修改。用户对GCC使用-fpic选项指示GNU编译系统生成PIC代码。共享库的编译必须总是使用该选项。

PIC代码引用包括数据引用和函数调用。对数据引用有一个事实，就是代码段中任何指令和数据段中任何变量之间的距离是一个运行时常量，与代码段和数据段的绝对内存位置是无关的。在编译器想要生成对PIC全局变量引用时，在数据段开始的地方创建了全局偏移量表(GOT)

动态链接库中的函数在程序执行的时候才会确定地址，所以编译器无法确定其地址，在汇编代码中也无法像静态库的函数那样体现。

当某个动态链接函数第一次被调用时先进入对应的PLT条目例如PLT[2]，然后PLT指令跳转到对应的GOT条目中例如GOT[4]，其内容是PLT[2]的下一条指令。然后将函数的ID压入栈中后跳转到PLT[0]。PLT[0]通过GOT[1]将动态链接库的一个参数压入栈中，再通过GOT[2]间接跳转进动态链接器中。动态链接器使用两个栈条目来确定函数的运行时位置，用这个地址重写GOT[4]，然后再次调用函数。经过上述操作，再次调用时PLT[2]会直接跳转通过GOT[4]跳转到函数而不是PLT[2]的下一条地址。

hello程序对动态链接库的引用，基于数据段与代码段相对距离不变这一个事实，因此代码段中任何指令和数据段中任何变量之间的距离都是一个运行时常量。

在dl\_init前后发生变化如下：

通过EDB调试，能够看出这个变化。先观察调用dl\_init前，动态库函数指向的地址。从上文节头中能够读取到GOT表的起始位置，即0x601000。在dl\_init调用之前可以查看其值，发现均为0。调用dl\_init后再次查看,0x601008， 0x601010处的两个八字节数据分别改变。这两个位置已经有了一段地址，分别为0x7fb7eaa21170和0x7fb7ea80f750。其中 GOT[1]指向重定位表，GOT[2] 存放动态链接器入口的地址

调用之前的got.plt

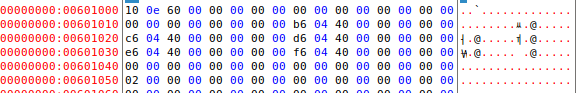


图5.4.1调用之前的全局偏移量表.got.plt

调用之后的got.plt

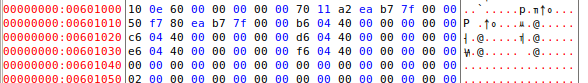


图5.4.2调用之后的全局偏移量表

重定位表：

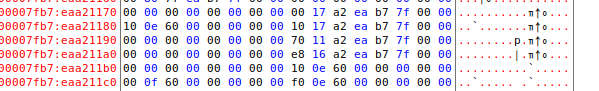


图5.4.3 0x7fb7eaa21170指向的重定位表

## 5.8 本章小结

本章介绍了链接的概念与作用。在链接的过程中，通过elf，反汇编分析了 hello 在链接后与未链接时的不同，利用edb等工具分析了 hello 的 虚拟地址空间、重定位过程、执行流程、动态链接过程。

**（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：进程的经典定义就是一个执行中程序的实例。系统中的每个程序都运行在某个进程的上下文中。上下文是由程序正确运行所需的状态组成的。这个状态包括存放在内存中的程序的代码和数据，他的栈、通用目的寄存器的内容、程序计数器、环境变量以及打开文件描述符的集合。进程的概念主要有两点：第一，进程是一个实体。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（text region）、数据区域（data region）和堆栈（stack region）。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；堆栈区域存储着活动过程调用的指令和本地变量。第二，进程是一个“执行中的程序”。程序是一个没有生命的实体，只有处理器赋予程序生命时（操作系统执行之），它才能成为一个活动的实体，我们称其为进程。

作用：进程作为一个执行中程序的实例，系统中每个程序都运行在某个进程的上下文中，上下文是由程序正确运行所需的状态组成的。这个状态包括存放在内存中的程序的代码和数据，它的栈、通用目的寄存器的内容、程序计数器、环境变量以及打开文件描述符的集合。

## 6.2 简述壳Shell-bash的作用与处理流程

shell是一个交互型的应用级程序，它代表用户运行其他程序。Shell执行一系列的读/求值步骤，然后终止。读步骤读取来自用户的一个命令行。求值步骤解析命令行，并代表用户运行程序。

处理流程：shell 命令重新初始化用户的登录会话。当给出该命令时，就会重新设置进程的控制终端的端口特征，并取消对端口的所有访问。然后 shell 命令为用户把进程凭证和环境重新设置为缺省值，并执行用户的初始程序。根据调用进程的登录用户标识建立所有的凭证和环境。读步骤读取来自用户的一个命令行。求值步骤解析命令行，并代表用户运行程序。

## 6.3 Hello的fork进程创建过程

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）份副本，包括代码和数据段、堆、共享库以及用户栈。子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，子进程可以读写父进程中打开的任何文件。 父进程和新创建的子进程之间最大的区别在于它们有不同的PID。fork函数是有趣的（也常常令人迷惑）， 因为它只被调用一次，却会返回两次：一次是在调用进程（父进程）中，一次是在新创建的子进程中。在父进程中，fork返回子进程的PID。在子进程中， fork返回0。因为子进程的PID总是为非零，返回值就提供一个明确的方法来分辨程序是在父进程还是在子进程中执行。在父进程调用fork后，到waitpid子进程终止或停止这段时间里，父进程执行的操作，和子进程的操作（如果没有什么其它复杂的操作的话），在时间顺序上是拓扑排序执行的。有可能，这段时间里父子进程的逻辑控制流指令交替执行。而父进程的waitpid后的指令，只能在子进程终止或停止后，waitpid返回后才能执行。

下面为简单进程图：

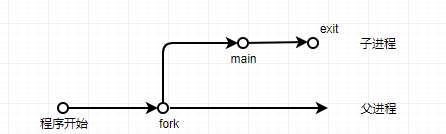


图6.1简单进程图

## 6.4 Hello的execve过程

当fork之后，子进程调用execve函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函数加载并运行可执行目标文件filename，且带参数列表argv和环境变量列表envp。只有当出现错误时，例如找不到filename，execve才会返回到调用程序。正常情况下，execve调用一次，但从不返回。

在execve加载了filename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，当main开始执行时，用户栈的结构如下图所示：

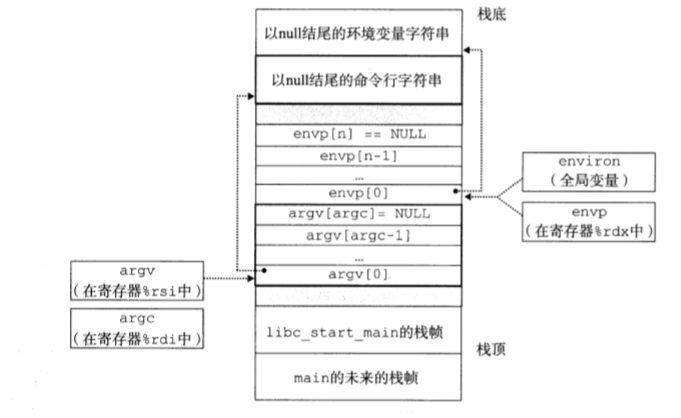


图6.2.1程序开始时，用户栈的典型结构

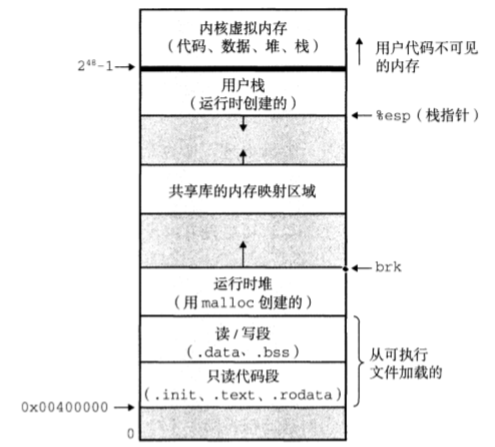


图6.2.2启动加载器创建的系统映像

首先是参数和环境字符串，然后是以null结尾的指针数组，每个指针都指向栈中的一个环境变量字符串。全局变量environ指向这些之阵中的第一个envp[0]。紧接着是以null结尾的argv数组，其中每个原色都指向栈中的一个字符串。栈顶部是系统启动函数libc\_start\_main

## 6.5 Hello的进程执行

上下文切换：上下文就是内核重新启动一个被抢占的进程所需的状态。这个状态包括在内存中的程序的代码和数据，它的栈、通用目的寄存器的内容、程序计数器、环境变量以及打开文件描述符的集合。

上下文切换包含

1）保存当前进程的上下文

2）恢复某个先前被抢占的进程被保存的上下文

3）将控制传递给这个新恢复的进程。

逻辑控制流：进程为每个程序提供了一种假象，好像程序在独占地使用处理器。

进程时间片：是分时操作系统分配给每个正在运行的进程微观上的一段CPU时间（在抢占内核中是：从进程开始运行直到被抢占的时间）。

用户态与核心态转换：用户态切换到内存态由三种方式：

1）系统调用：用户态进程通过系统调用申请使用操作系统提供的服务程序完成工作，比如前例中fork()实际上就是执行了一个创建新进程的系统调用。

2）异常：当CPU在执行运行在用户态下的程序时，发生了某些事先不可知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关程序中，也就转到了内核态，比如缺页异常。

3）外围设备的中断。

进程调度的过程：调用sleep之前，如果没有被抢占hello继续运行，若被抢占，要进行上下文切换（具体在上面已解释）

例如：hello一开始在用户态，然后hello调用sleep后会进入内核态，被挂起一段时间，hello进入等待态，并开始计时。内核进行上下文切换，将控制转移到新的进程，当前进程的上下文被保存，hello进入休眠。当计时器到2.5s时发送一个中断信号，此时内核进行中断处理，hello进入用户模式，变成运行太，继续进行自己的逻辑控制流。

下面给出简单示意图：

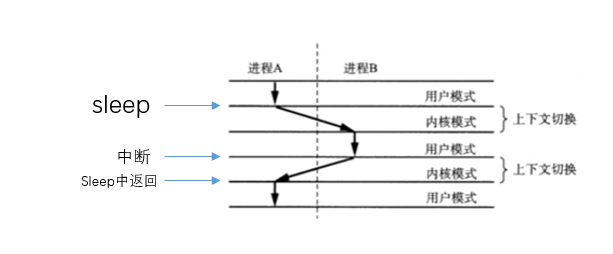


图6.3hello中sleep上下文切换

## 6.6 hello的异常与信号处理

hello执行过程中可能出现四类异常：中断、陷阱、故障和终止。

1）中断是来自I/O设备的信号，异步发生，中断处理程序对其进行处理，返回后继续执行调用前待执行的下一条代码，就像没有发生过中断。

2）陷阱是有意的异常，是执行一条指令的结果，调用后也会返回到下一条指令，用来调用内核的服务进行操作。帮助程序从用户模式切换到内核模式。

3）故障是由错误情况引起的，它可能能够被故障处理程序修正。如果修正成功，则将控制返回到引起故障的指令，否则将终止程序。

4）终止是不可恢复的致命错误造成的结果，通常是一些硬件的错误，处理程序会将控制返回给一个abort例程，该例程会终止这个应用程序。

hello执行过程中，可能会遇到各种异常，信号则是一种通知用户异常发送的机制。比如Ctrl-Z和Ctrl-C，分别触发SIGSTP和SIGINT信号。收到信号后进程会调用相应的信号处理程序对其进行处理。

**6.6.1正常运行**

程序执行完成后，进程被回收。

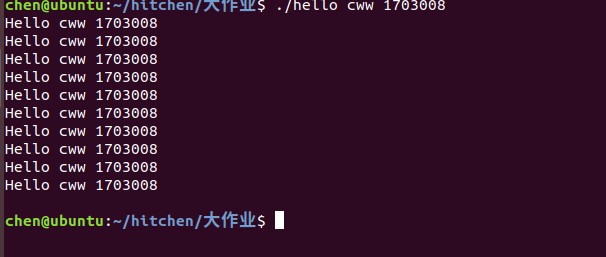


图6.4.1正常运行程序

**6.6.2程序运行过程中按Ctrl-C**

程序仅输出一个字符串，还未结束时，从键盘输入ctrl-c，可以看见进程直接停止。这是因为父进程收到了sigint信号，信号处理程序处理该信号，使进程停止，并且回收了子进程。

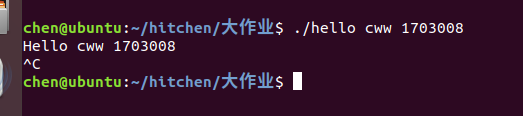


图6.4.2运行过程中按Ctrl-C

**6.6.3程序运行过程中按Ctrl-Z**

在程序运行还未结束时，输入ctrl-z，可以看见进程提示已停止。这是因为进程收到sigstp信号，信号处理程序将该程序挂起。



图6.4.3运行过程中按Ctrl-Z

①输入ptree得到一个树型结构

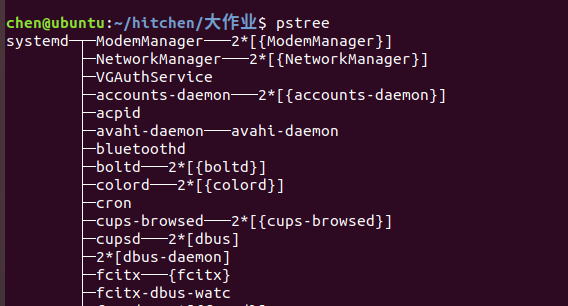


图6.4.3.1进行停止后输入pstree得到树型结构

②分别输入ps和jobs信号得到

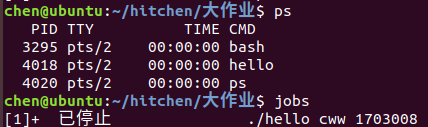


图6.4.3.2输入ps和jbos得到PID和后台等待进程

说明该进程还没有被回收，只是在后台挂起。job号为1。

③输入fg 1 使该进程恢复到前台，继续执行。

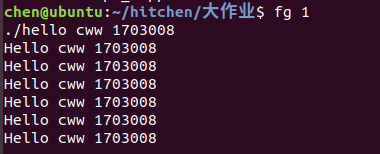


图6.4.3.3输入fg 1使程序继续运行

可以看到hello继续执行，输出六次后结束

④输入kill -9 4083 将hello进程杀死

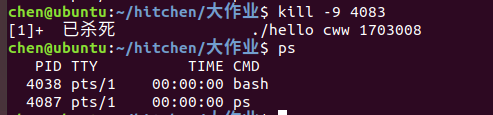


图6.4.3.4 kill命令杀死进程

**6.6.4随意输入字符串以及回车得到**

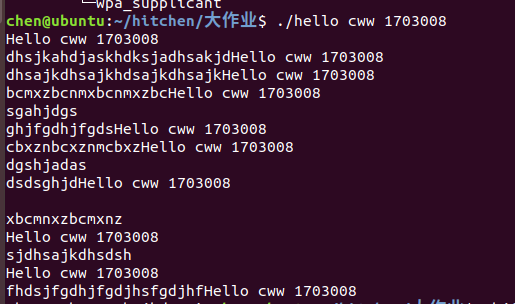


图6.4.4运行过程中输入任意字符和回车

可见cww 1703008还是照常输出，其他的只是作为屏幕的显式，不影响进程的进行。

## 6.7本章小结

本章主要对应书中第八章进程的内容。只要介绍了shell如何利用fork和execve函数创建子进程并执行程序。进程拥有着与父进程相同却又独立的环境，与其他系统进并发执行，拥有各自的时间片，在内核的调度下有条不紊的执行着各自的指令。进程在执行过程中可能遇到的上下文切换，用户态和内核态之间的转化。以及进程执行过程中碰到异常，shell如何接受异常信号，信号处理程序如何处理这些异常并作出相应反应。这样，程序能够对不同的信号做出不同的处理。

**（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：逻辑地址(LogicalAddress)是指由程序产生的与段相关的偏移地址部分。要经过寻址方式的计算或变换才得到内存储器中的实际有效地址，即物理地址。

线性地址：也称为虚拟地址，也不是一个真实的地址，他是对应了硬件页式内存转换前地址。

虚拟地址：和线性地址相同

物理地址：就是物理内存中的真正地址。物理地址是在地址总线上，以电子形式存在的，使得数据总线可以访问主存的某个特定存储单元的内存地址。

比如hello汇编中callq 4004e0，其中4004e0是逻辑地址的偏移地址，必须加上基地址才能变成线性空间地址。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

在80386 的段机制中，逻辑地址由两部分组成，即段部分（选择符）及偏移部分。

段是形成逻辑地址到线性地址转换的基础。

段存储器结构大概如下：

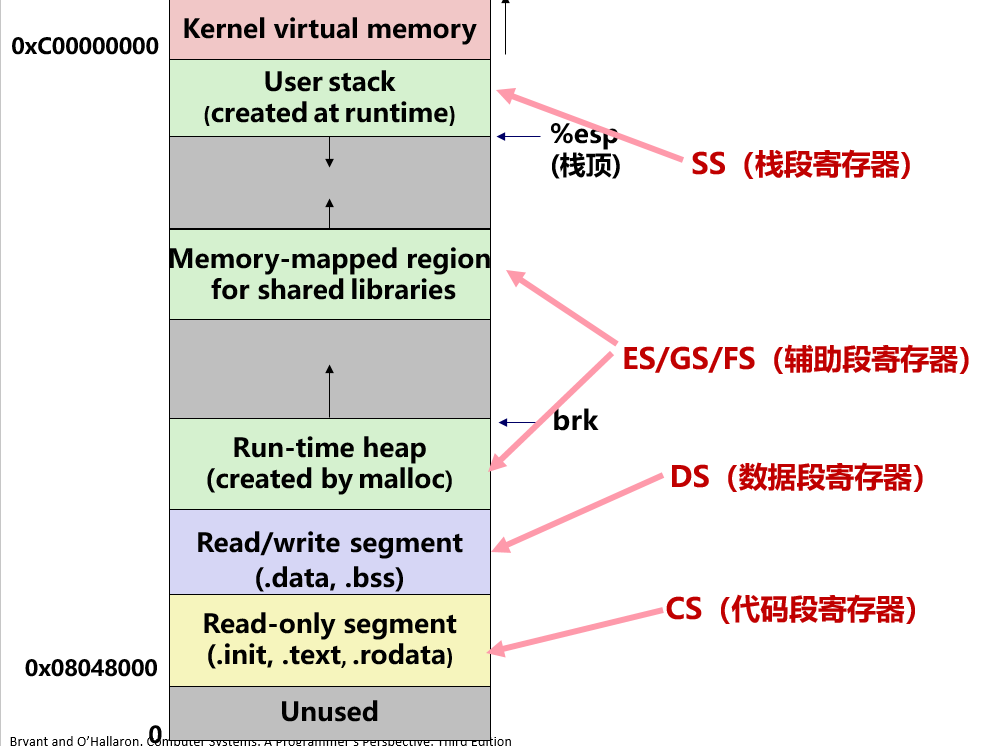


图7.1.1段存储器的结构

段存储器用于存放段选择符。由上图段存储器的结构，分为以下几个段：

1）CS(代码段)：程序代码所在段

2）SS(栈段)：栈区所在段

3）DS(数据段)：全局静态数据区所在段

4）其他3个段寄存器ES、GS和FS可指向任意数据段。

段选择符的结构如下：

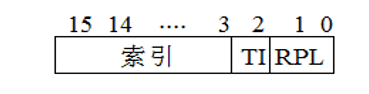


图7.1.2段选择符的结构

下面解释段选择符各段所代表的意义：

1）TI=0,选择全局描述符表，TI=1，选择局部描述表

2）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，第0级高于第3级

3）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

介绍完段选择符，接下来看段描述符。

段描述符是一种数据结构，实际上就是段表项

描述符表实际上就是段表，由段描述符(段表项)组成。有三种类型。

①全局描述符表GDT：只有一个，用来存放系统内每个任务都可能访问的描述符，例如，内核代码段、内核数据段、用户代码段、用户数据段以及TSS（任务状态段）等都属于GDT中描述的段

②局部描述符表LDT：存放某任务（即用户进程）专用的描述符

③中断描述符表IDT：包含256个中断门、陷阱门和任务门描述符

段描述符的结构：

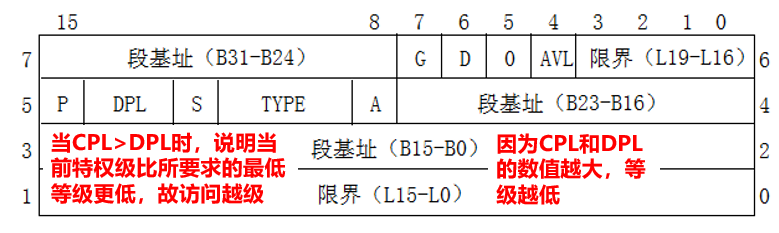


图7.1.3段描述符的结构

通过上述的段描述符，得到如下从逻辑地址到线性地址转换图：

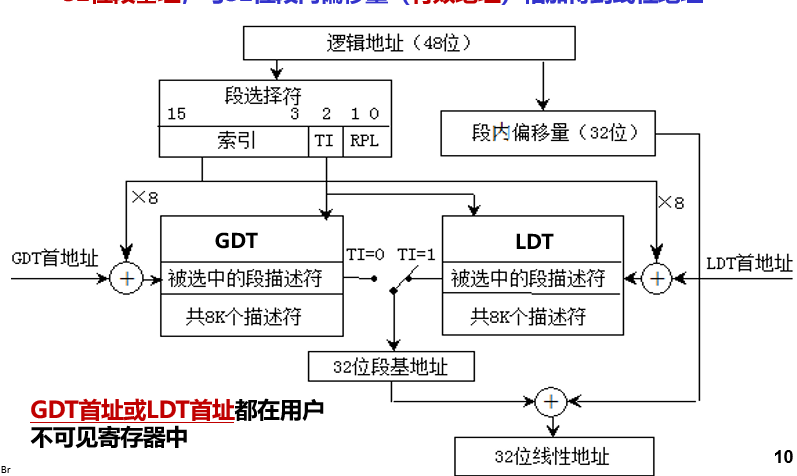


图7.1.4逻辑地址转换为线性地址

简要说明：首先从逻辑地址的高16位得到段选择符，低32位位段内偏移量。然后通过TI选择GDT或者LDT，索引值\*8+GDT或LDT首地址得到被选择的段描述符，最后得到32位段基地址，加上偏移量得到32位线性地址。

## 7.3 Hello的线性地址到物理地址的变换-页式管理

从线性地址到物理地址的变换通过页式管理实现。

虚拟内存被组织为一个由存放在磁盘上的N个连续的字节大小的单元组成的数组。每字节都有一个唯一的虚拟地址，作为到数组的索引。磁盘上数组的内容被缓存在主存中。虚拟页是带虚拟内存系统将虚拟内存分割为大小固定的块，作为磁盘和主存（较高层）之间的传输单元。VM系统通过将虚拟内存分割成虚拟虚拟页的大小固定的块来解决。任意时刻，虚拟页面的集合分为三个不相交的子集：未分配的、缓存的、未缓存的。

页表是一个存放在物理内存的数据结构，它将虚拟页映射到物理页。每次地址翻译硬件将一个虚拟地址转换为物理地址时，都会读取页表。

页表就是一个页表条目的数组。虚拟地址空间中的每个页在页表中一个固定偏移量出都有一个PTE。每个PTE由一个有效位和一个n位地址字段组成。有效位表面该虚拟页是否缓存在DRAM中，如果设置了有效位，那么地址字段就表示DRAM中相应的物理页的起始位置，这个物理页中缓存了该虚拟页，如果没有有效位，若是空地址就表明未分配，否则就执行该虚拟页在磁盘上的起始位置。

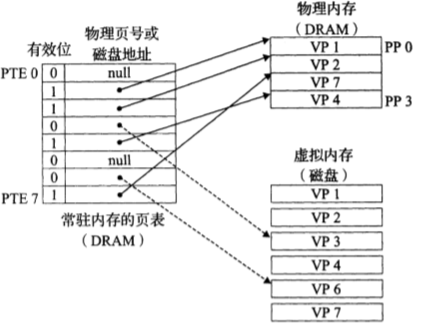


图7.2.1页表与虚拟内存、物理内存

由虚拟地址变为物理地址需要MMU来进行操作。它通过虚拟地址中的VPN来找到对应的PTE，如果有效位为1则命中，它的物理页号（PPN）加上虚拟页偏移量（VPO）就是物理地址。反之发送缺页，发生缺页时，MMU会选择牺牲页，将其替换，再进行刚刚的指令。下图为图示：

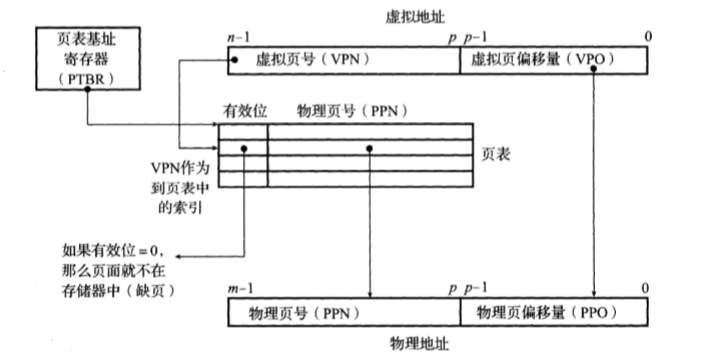


图7.2.2通过页表将虚拟地址转换为物理地址

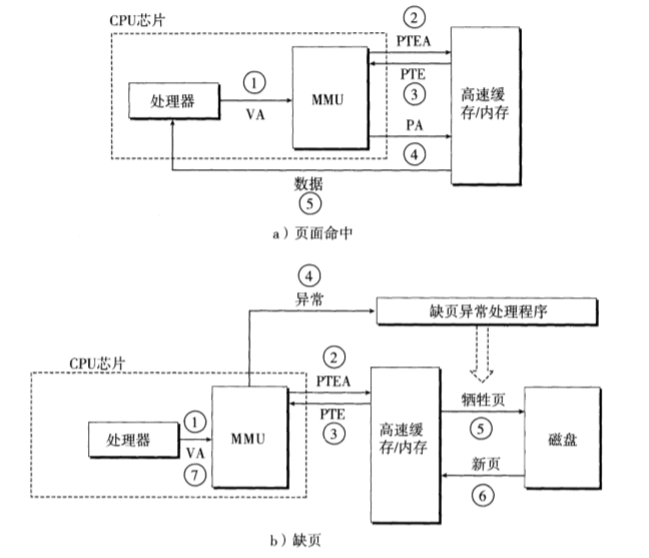


图7.2.3页面命中和缺页的操作图

## 7.4 TLB与四级页表支持下的VA到PA的变换

每次CPU产生一个虚拟地址，MMU就必须查阅一个PTE，为了减少时间开销，设计了TLB。TLB是一个关于PTE的小缓存。称为翻译后备缓冲器。

TLB是一个小的、虚拟寻址的缓存，其每一行都保存着一个由单个PTE组成的块。

结构如下图：其中TLB是标记位，TLB是组索引，VPO是虚拟页偏移量

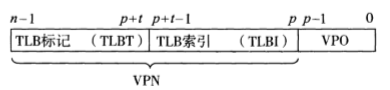


图7.3.1TLB结构

用于组选择和行匹配的索引和标记字段是从虚拟地址中的虚拟页号中提取出来的。若TLB由T=2^t各组，那么TLB索引由VPN的t各最低为盛。TLB由VPN中剩余的位组成的。下图是TLB命中与不命中的两种情况。

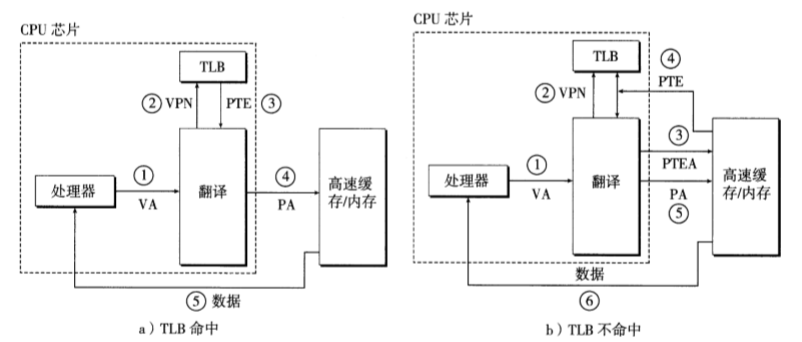


图7.3.2TLB命中与不命中的操作图

Intel Core i7 实现支持48位（256TB）虚拟地址空间和52位（4PB）物理地址空间。Linux使用的是4KB的页。X64 CPU上的PTE为64位（8bytes），所以每个页表一共有512个条目。512个PTE需要有9位VPN来定位。在四级页表的条件下，一共需要36位VPN，因为虚拟地址空间是48位的，所以低12位是VPO。TLB是四路组联的，共有16组，需要有4位TLBI来定位，所以VPN的低4位是TLBI，高32位是TLBT。

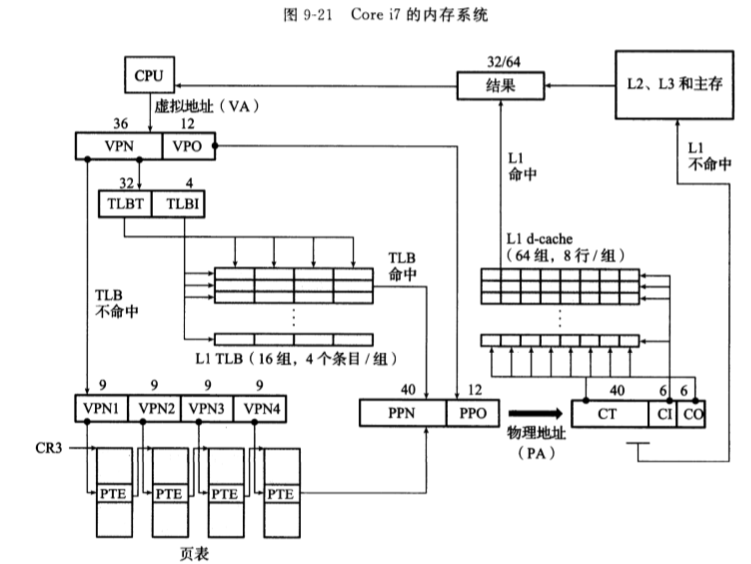


图7.3.3 Core i7地址翻译的概况

下图是上图的一部分，给出了如何用四级的页表来将虚拟地址翻译成物理地址。36位VPN被划分成四个9位的片，每个片被用做到一个页表的偏移量。CR3寄存器包含L1页表的物理地址。VPN1提供到一个L1 PET的偏移量，这个PTE包含L2页表的基地址。VPN2提供到一个L2 PTE的偏移量，以此类推。

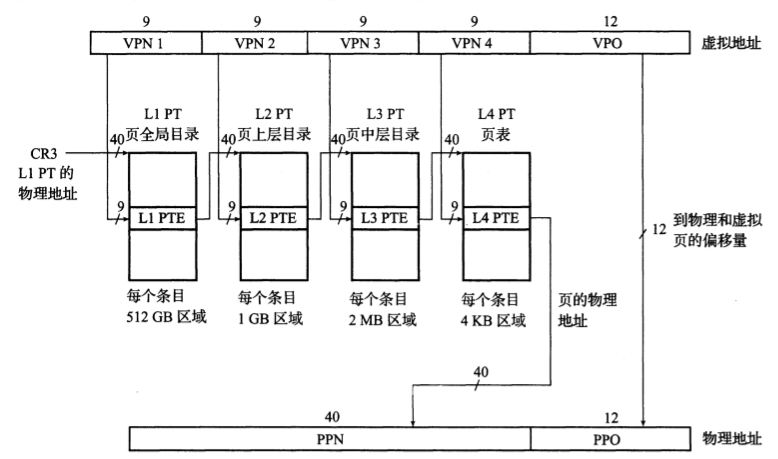


图7.3.4 Core i7页表翻译

## 7.5 三级Cache支持下的物理内存访问

物理内存访问，是MMU将虚拟地址转化为物理地址后，去cache中访问。

下图是L1 cache中的物理地址寻址，L2,L3寻址与L1相同。

L1 Cache是8路64组相联。块大小为64B。因为共有64组，所以需要6位CI来进行组寻找；块大小为64B，所以需要6位CO表示偏移量。物理地址共有52位，所以还有40位作为CT。

物理内存访问分为以下几步：

1）首先需要通过CI来选择组。

2）选择好组后，遍历组中的每一行，比较行的标记和地址的标记，当且仅当这两者相同，并且行的有效位设为1时，才可是缓存命中

3）最后是字选择。定位好了要寻址的地址在哪一行之后，根据地址的块偏移量，在行的数据块中偏移寻址，最后得到的字，就是我们寻址得到的字。

4）如果缓存不命中，那么它需要从存储器层次结构中的下一层取出被请求的块，然后将新的块存储在组索引位指示的组中的一个高速缓存行中。这个过程，如果有冲突不命中，就会触发行的替换。

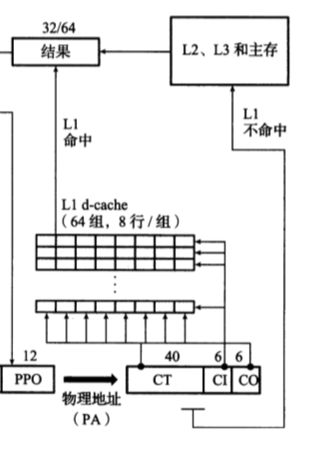


图7.4物理内存的访问

## 7.6 hello进程fork时的内存映射

一个对象可以被映射到虚拟内存的一个区域，要么作为共享对象，要么作为私有对象。如果一个进程将一个共享对象映射到它的虚拟地址空间的一个区域内，那么这个进程对这个区域的任何写操作，对于那些也把这个共享对象映射到他们虚拟内存的其他进程而言，也是可见的。而且，这些变化也会反应在磁盘上的原始对象中。

另一方面，对于一个映射到四有对象的区域做的改变，对于其他进程来说是不可见的，并且进程对这个区域所做的任何写操作都不会反映在磁盘上的对象中，一个映射到共享对象的虚拟内存区域叫做共享区域。假设进程1将一个共享对象映射到它的虚拟内存的一个区域中。现在假设进程2将同一个共享对象映射到他的地址空间。

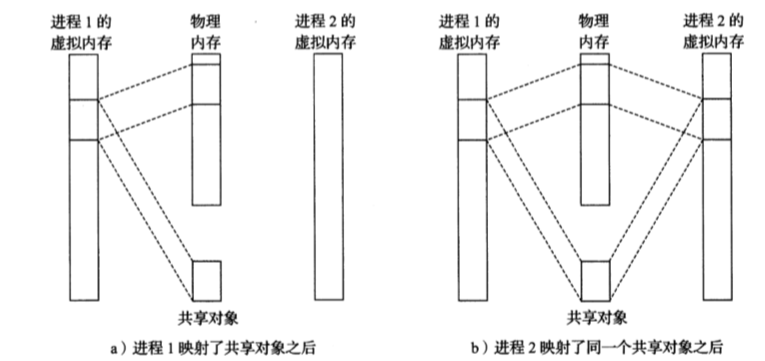


图7.5.1一个共享对象

两个进程将一个私有对象映射到它们虚拟内存的不同区域，但是共享这个对象同一个物理副本。对于每个映射私有对象的进程，相应私有区域的页表条目都被标记为只读，并且区域结构被标记为写时复制。只要没有进程试图写它自己的私有区域，它们就可以继续共享物理内存中对象的一个单独副本。然而，只要有一个进程试图写私有区域的某个页面，那么这个写操作会触发一个保护故障。当故障处理程序注意到保护异常是由于进程试图写私有的写时复制区域中的一个页面而引起的，它就会在物理内存中创建这个页面的一个新副本，更新页表条目指向这个新的副本，然后恢复这个页面的可写权限，当故障程序返回时，CPU重新执行这个写操作，现在在新创建的页面上这个写操作就可以正常执行了。通过延迟私有对象中的副本直到最后可能的时刻，写时复制最充分地利用了稀有的物理内存。

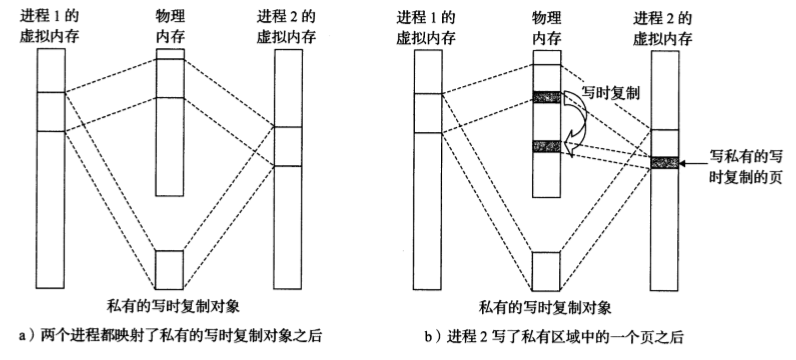


图7.5.2一个私有的写时复制对象

shell通过fork为hello创建新进程。当fork函数被当前进程调用时，内核为新进程创建各种数据结构，并分配给hello进程唯一的PID。为了给这个新进程创建虚拟内存，它创建了当前进程的mm\_struct、区域结构和样表的原样副本。它将两个进程中的每个页面都标记为只读，并将每个进程中的每个区域结构都标记为写时复制。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时存在的虚拟内存相同。当这两个进程汇总该点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

execve函数在当前进程中加载并允许包含在可执行目标文件中的程序，用该程序有效地地带了当前程序。加载并运行该程序需要以下几个步骤：

1）删除已存在的用户区域。删除当前进程虚拟地址的用户部分中的已存在的区域结构。

2）映射私有区域。为新程序的代码、数据、bss和栈区域创建新的区域结构，所有这些新的区域都是私有的、写时复制的。代码和数据区域被映射为hello.out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.out中，栈和堆地址也是请求二进制零的，初始长度为零。图7.9概括了私有区域的不同映射。

3）映射共享区域， 如果hello.out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。

4）设置程序计数器（PC）。execve做的最后一件事情就是设置当前进程上下文的程序计数器，使之指向代码区域的入口点。

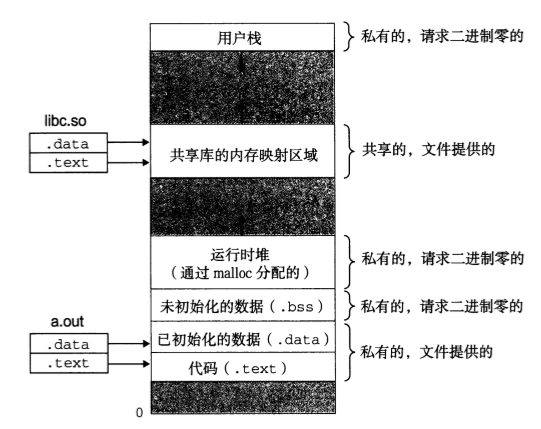


图7.6加载器是如何映射用户地址空间的区域

## 7.8 缺页故障与缺页中断处理

缺页故障：物理内存（DRAM）缓存不命中成为缺页。假设CPU引用了磁盘上的一个字，而这个字所属的虚拟页并未缓存在DRAM中。地址翻译硬件会从内存中读取虚拟页对应的页表，推断出这个虚拟页未被缓存，然后触发一个缺页异常。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。如果被牺牲的页面被修改了，那么内核会把它复制回磁盘。总之，内核会修改被牺牲页的页表条目，表示它不再缓存在DRAM中了。

处理缺页需要硬件和操作系统内核写作完成，分为以下几步：

1）处理器生成一个虚拟地址，并把它传送给MMU

2）MMU生成PTE地址，并从高速缓存/贮存请求得到它

3）高速缓存/主存向MMU返回PTE。

4）PTE中的有效位是零，所以MMU触发了－次异常，传递CPU中的控制到操作系统内核中的缺页异常处理程序。

5）缺页处理程序确定出物理内存中的牺牲页，如果这个页面已经被修改了，则把它换出到磁盘。

6）缺页处理程序页面调入新的页面，并更新内存中的PTE

7）缺页处理程序返回到原来的进程，再次执行导致缺页的指令。CPU将引起缺 页的虚拟地址重新发送给MMU。因为虚拟页面现在缓存在物理内存中，所以就会命中，在MMU执行了下图中的步骤之后，主存就会将所请求字返回给处理器。

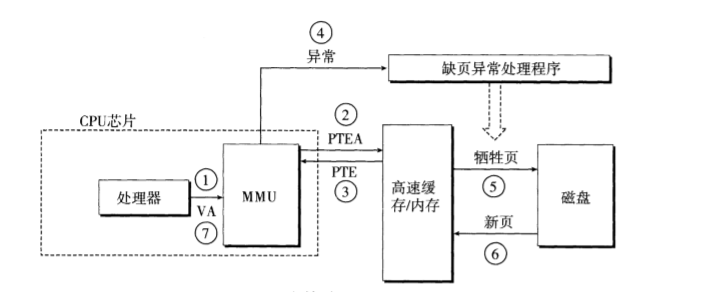


图7.7发生缺页处理操作图

## 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。对于每个进程，内核维护着一个变量brk，它指向堆的顶部。

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，知道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。

分配器由两种基本风格：显式分配器和隐式分配器。

一、隐式空闲链表

1）简单的堆块的格式：一个块是由一个字的头部、有效载荷，以及可能的一些额外的填充组成的。头部编码了这个块的大小，以及这个块是已分配的还是空闲的。

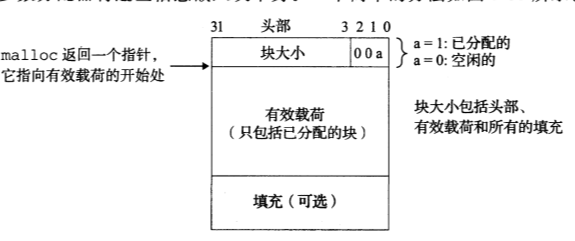


图7.8.1一个简单的堆块的格式

2）使用边界标记的堆块的格式：比简单的堆块格式多了一个脚部（footer），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。

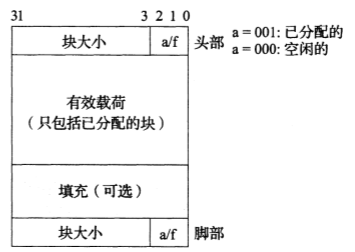


图7.8.2使用边界标记的堆块的格式

考虑以下四种情况的合并：

①前面的块和后面的块都是已分配的

②前面的块时已分配的，后面的块是空闲的

③前面的块是空闲的，而后面的块是已分配的

④前面和后面的块都是空闲的

下图为图示：

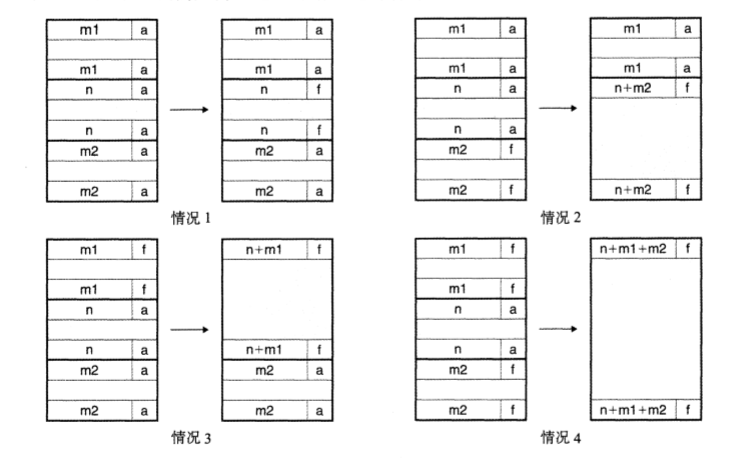


图7.8.3使用边界标记的合并

二、显式空闲链表

将堆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，如下图所示：

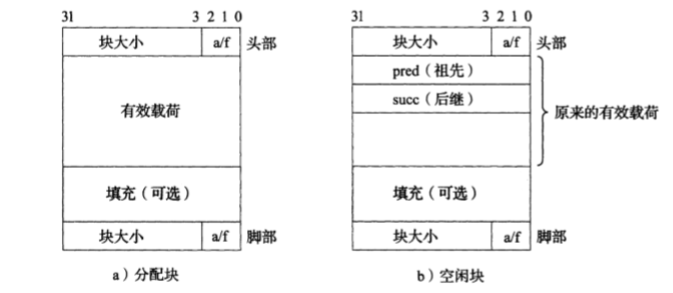


图7.8.3使用双向空闲链表的堆块的格式

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快数量的线性时间。不过，释放一个块的时间可以使线性，也可能是个常数，这取决于我们选择的排序策略：

1）使用先进后出的顺序维护链表，将新释放的块放置在链表的最开始处。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的块。在这种情况下，释放一个快可以在常数时间内完成。如果使用边界标记，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，释放一个块血药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排序的首次适配由更高的内存利用率，接近最佳适配的利用率。

三、分离的空闲链表

为了减少分配时间，采用分离存储，就是维护多个链表，其中每个链表中的块有大致相等的大小。一般的思路是将可能的块大小分成一些等价类。以下是两种基本的方法：

1）简单分离存储；每个大小类的空闲链表包含大小相等的块，每个快的大小就是这个大小类中最大元素的大小。分配块时，检查相应的空闲链表，如果不空，简单分配其中第一块的全部，如果空，就像操作系统请求一个固定大小的额外内存片，将这个片分成大小相等的块。释放一个块，分配器只需要简单地将这个块插入到相应的空闲链表的前部

2）分离适配：每个空闲链表时和一个大小类相关联的，并且呗组织成某种类型的显式或隐式链表。每个链表包含潜在的大小不同的块，这些块的大小时大小类的成员。分配一个快，必须确定请求的大小类，并且对适当的空闲链表做首次适配，查找一个空闲块。找到了，就分割，将剩余部分插入到合适的空闲链表中，否则继续查找空闲链表。若空闲链表中没有合适的块，就申请额外的内存。释放一个块时，要进行合并然后插入。

## 7.10本章小结

本章主要介绍了 hello 的存储器地址空间、intel 的逻辑地址到线性地址的转化（段式管理）、hello 的线性地址到物理地址的转换（页式管理）， 介绍了intel Core7在TLB和四级页表的情况下VA 到PA 的变换、三级cache下的内存访问，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、动态存储分配管理等等方面。

**（第7章 2分）**

# 第8章 hello的IO管理

## 8.1 Linux的IO设备管理方法

## 一个Linux文件就是一个m个字节的序列，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。这个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，这使得输入和输出都能以一种统一且一致的方式的来执行。

## 8.2 简述Unix IO接口及其函数

接口就是连接CPU与外设之间的部件，它完成CPU与外界的信息传送。还包括辅助CPU工作的外围电路，如中断控制器、DMA控制器、定时器、高速CACHE

Linux内核创建的每个进程都以与一个终端相关联的三个打开的文件开始:

0: 标准输入 (stdin)

1: 标准输出 (stdout)

2: 标准错误 (stderr)

函数：

1）打开文件：int open(char \*filename, int flags, mode\_t mode);返回一个小的描述符数字——文件描述符。返回的描述符总是在进程中当前没有打开的最小描述符。fd==-1说明发生错误

2）关闭文件：int close(int fd);关闭文件是通知内核你要结束访问一个文件，关闭一个已经关闭的文件是导致线程程序灾难的一个因素。

3）读文件：ssize\_t read(int fd, void \*buf, size\_t n);read函数从描述符为fd的当前文件复制最多n个字节到内存位置buf。返回值-1表示一个错误，而返回值0表示EOF。否则，返回值表示的是实际传送的字节数量。

4）写文件：ssize\_t write(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fd的当前文件位置。

## 8.3 printf的实现分析

首先查看printf函数：

1. **int** printf(**const** **char** \*fmt, ...)
2. {
3. **int** i;
4. **char** buf[256];
6. **va\_list** arg = (**va\_list**)((**char**\*)(&fmt) + 4);
7. i = vsprintf(buf, fmt, arg);
8. write(buf, i);
10. **return** i;
11. }

由上可知函数参数个数不确定，所以首先需要得到确定的参数个数。

其中arg是…中的第一个参数的地址。

接着看见 i = vsprintf(buf, fmt, arg);

查看vsprintf（buf,fmt,arg）函数：

1. **int** vsprintf(**char** \*buf, **const** **char** \*fmt, **va\_list** args)
2. {
3. **char**\* p;
4. **char** tmp[256];
5. **va\_list** p\_next\_arg = args;
7. **for** (p=buf;\*fmt;fmt++) {
8. **if** (\*fmt != '%') {
9. \*p++ = \*fmt;
10. **continue**;
11. }   //去除无关字符
13. fmt++;
15. **switch** (\*fmt) {
16. **case** 'x':   //只处理这种情况
17. itoa(tmp, \*((**int**\*)p\_next\_arg));   //将输入参数转化为字符串
18. strcpy(p, tmp);   //复制字符串，将tmp存到p种
19. p\_next\_arg += 4;   //定位到下一个参数
20. p += strlen(tmp);   //放下一个参数地址
21. **break**;
22. **case** 's':
23. **break**;
24. **default**:
25. **break**;
26. }
27. }
29. **return** (p - buf);   //最后生成字符串长度
30. }

该函数返回的是要打印出来的字符长的长度。vsprintf的作用就是格式化。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。

接着看write：

write:

mov eax, \_NR\_write

mov ebx, [esp + 4]

mov ecx, [esp + 8]

int INT\_VECTOR\_SYS\_CALL

其中最后一行的 INT\_VECTOR\_SYS\_CALL表示通过系统来调用sys\_call函数。

看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

其中：ecx中是要打印出的元素个数 ；ebx中的是要打印的buf字符数组中的第一个元素 ；这个函数的功能就是不断的打印出字符，直到遇到：'\0'。

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）。

到此，printf要打印的东西，就呈现在标准输出上了。

## 8.4 getchar的实现分析

getchar函数如下：

1. 1.  **int** getchar(**void**)
2. 2.  {
3. 3.      **static** **char** buf[BUFSIZ];
4. 4.      **static** **char**\* bb=buf;
5. 5.      **static** **int** n=0;
6. 6.      **if**(n==0)
7. 7.      {
8. 8.          n=read(0,buf,BUFSIZ);
9. 9.          bb=buf;
10. 10.     }
11. 11.     **return**(--n>=0)?(unsigned **char**)\*bb++:EOF;
12. 12. }

bb是缓冲区的开始，int类型的n只有在n=0的轻快下，才从缓冲区读BUFSIZE个字节，就是将缓冲区的内容全部读入。这时候n已经不为0，返回时如果n大雨淋，就返回缓冲区的第一个字符，否则，就返回EOF。这样，getchar函数通过read函数返回字符，实现了读取一个字符的功能。

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

本章主要介绍了Linux的IO设备管理方法，Unix IO接口及其函数功能，分析了printf和getchar函数的实现。所有的I/O设备都被模型化为文件，通过文件的读写来实现I/O操作。Unix I/O接口函数可以实现一些I/O操作。printf的实现和vsprintf以及write有关，要先解析格式化字串，再调用I/O函数write写到标准输出上。getchar函数与键盘回车相关，也就是需要异步异常-键盘中断的处理，调用I/O函数read，通过系统调用读取案件ascii码。

**（第8章1分）**

# 结论

hello程序终于完成了它艰辛的一生，它一生中所经历的大事有：

1）编写：程序员通过键盘输入hello.c中的代码

2）预处理：将源代码中的头文件展开，展开所有的宏定义

3）编译：编译器将.i文件按照规则，编译成汇编代码.s文件

4）汇编：将汇编代码翻译成机器指令，变成可重定位目标文件hello.o

5）链接：连接器将可重定位目标文件和动态链接哭链接称为可执行目标程序hello

6）运行：shell调用fork程序创建子进程，调用execve函数启动加载器，执行hello，cpu为其分配时间片，在时间片中，hello执行直接的逻辑控制流

7）内存管理：在shell运行hello的同时，为hello分配了地址空间

8）内存访问：在hello运行过程中，各个指令的执行，都需要将虚拟地址翻译成物理地址，并在主存中取字节返回给cpu

9）信号与异常：在运行过程中，可能会遇到各种信号，触发异常，这时需要信号处理程序已经异常处理程序工作，保证程序的执行

10）结束：shell父进程回收子进程，内核删除和hello有关的所有数据。

在短短一个学期的学习了有关计算机系统的这么多知识，脑子里对于知识的整体把握是很模糊的。通过这次大作业，对书中知识有了一次梳理，弄清了许多问题，收获很大。

**（结论0分，缺失 -1分，根据内容酌情加分）**

# 附件

|  |  |
| --- | --- |
| 文件名称 | 作用 |
| hello.i | 预处理后生成的.i文本文件 |
| hello.s | 编译之后的汇编文件 |
| hello.o | 汇编后的生成的可重地位目标文件 |
| hello.elf | hello.o的ELF格式 |
| helloobjdump | 可重定位目标文件的反汇编代码 |
| hellolink.elf | hello的ELF格式 |
| hellolink.objdump | 链接后的反汇编代码 |
| hello | 链接后的可执行程序 |

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

[2] printf 函数实现的深入剖析 https://www.cnblogs.com/pianist/p/3315801.html

[3] ELF（一）—（七） https://blog.csdn.net/ylcangel

[4] 逻辑地址、线性地址、物理地址以及虚拟存储器 https://blog.csdn.net/mrbuffoon/article/details/48947267

[5] 用户态与内核态的转换https://blog.csdn.net/u014467169/article/details/52566167

[6] 动态链接的整个过程 <https://blog.csdn.net/a1342772/article/details/77688148>

[7] LINUX 逻辑地址、线性地址、物理地址和虚拟地址 https://www.cnblogs.com/zengkefu/p/5452792.html

**（参考文献0分，缺失 -1分）**