

**计算机系统**

**大作业**

题 目 程序人生-Hello’s P2P

**计算机科学与技术学院**

**2018年12月**

**摘 要**

简简单单的hello.c应该是每一个程序员所写出的人生中的第一个程序，但是不要看hello.c简短，但是其在计算机中运行却有着许许多多的奥妙值得每一个程序员去探索，由简单的程序开始去一步一步了解更加复杂，更加值得去探索的程序，才会一步一步的成长，这次的大作业就从hello开始第一步。

**关键词：hello，运行，系统**

**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 - 5 -](#_Toc532238400)

[第2章 预处理 - 6 -](#_Toc532238401)

[2.1 预处理的概念与作用 - 6 -](#_Toc532238402)

[2.2在Ubuntu下预处理的命令 - 6 -](#_Toc532238403)

[2.3 Hello的预处理结果解析 - 7 -](#_Toc532238404)

[2.4 本章小结 - 8 -](#_Toc532238405)

[第3章 编译 - 9 -](#_Toc532238406)

[3.1 编译的概念与作用 - 9 -](#_Toc532238407)

[3.2 在Ubuntu下编译的命令 - 9 -](#_Toc532238408)

[3.3 Hello的编译结果解析 - 9 -](#_Toc532238409)

[3.4 本章小结 - 15 -](#_Toc532238410)

[第4章 汇编 - 16 -](#_Toc532238411)

[4.1 汇编的概念与作用 - 16 -](#_Toc532238412)

[4.2 在Ubuntu下汇编的命令 - 16 -](#_Toc532238413)

[4.3 可重定位目标elf格式 - 16 -](#_Toc532238414)

[4.4 Hello.o的结果解析 - 19 -](#_Toc532238415)

[4.5 本章小结 - 20 -](#_Toc532238416)

[第5章 链接 - 21 -](#_Toc532238417)

[5.1 链接的概念与作用 - 21 -](#_Toc532238418)

[5.2 在Ubuntu下链接的命令 - 21 -](#_Toc532238419)

[5.3 可执行目标文件hello的格式 - 21 -](#_Toc532238420)

[5.4 hello的虚拟地址空间 - 26 -](#_Toc532238421)

[5.5 链接的重定位过程分析 - 26 -](#_Toc532238422)

[5.6 hello的执行流程 - 29 -](#_Toc532238423)

[5.7 Hello的动态链接分析 - 30 -](#_Toc532238424)

[5.8 本章小结 - 31 -](#_Toc532238425)

[第6章 hello进程管理 - 32 -](#_Toc532238426)

[6.1 进程的概念与作用 - 32 -](#_Toc532238427)

[6.2 简述壳Shell-bash的作用与处理流程 - 32 -](#_Toc532238428)

[6.3 Hello的fork进程创建过程 - 32 -](#_Toc532238429)

[6.4 Hello的execve过程 - 33 -](#_Toc532238430)

[6.5 Hello的进程执行 - 35 -](#_Toc532238431)

[6.6 hello的异常与信号处理 - 35 -](#_Toc532238432)

[6.7本章小结 - 38 -](#_Toc532238433)

[第7章 hello的存储管理 - 39 -](#_Toc532238434)

[7.1 hello的存储器地址空间 - 39 -](#_Toc532238435)

[7.2 Intel逻辑地址到线性地址的变换-段式管理 - 39 -](#_Toc532238436)

[7.3 Hello的线性地址到物理地址的变换-页式管理 - 41 -](#_Toc532238437)

[7.4 TLB与四级页表支持下的VA到PA的变换 - 43 -](#_Toc532238438)

[7.5 三级Cache支持下的物理内存访问 - 45 -](#_Toc532238439)

[7.6 hello进程fork时的内存映射 - 47 -](#_Toc532238440)

[7.7 hello进程execve时的内存映射 - 48 -](#_Toc532238441)

[7.8 缺页故障与缺页中断处理 - 48 -](#_Toc532238442)

[7.9动态存储分配管理 - 49 -](#_Toc532238443)

[7.10本章小结 - 52 -](#_Toc532238444)

[第8章 hello的IO管理 - 53 -](#_Toc532238445)

[8.1 Linux的IO设备管理方法 - 53 -](#_Toc532238446)

[8.2 简述Unix IO接口及其函数 - 53 -](#_Toc532238447)

[8.3 printf的实现分析 - 54 -](#_Toc532238448)

[8.4 getchar的实现分析 - 56 -](#_Toc532238449)

[8.5本章小结 - 56 -](#_Toc532238450)

[结论 - 57 -](#_Toc532238451)

[附件 - 58 -](#_Toc532238452)

[参考文献 - 59 -](#_Toc532238453)

# 第1章 概述

## 1.1 Hello简介

hello.c文件是通过gedit、vim、Code:Blocks、sublime等文档编辑器(或者是IDE)应用程序创建对应类型文件进入代码所得到的。

在Linux及其发行版中在终端通过gcc编译器，分别键入cpp(预处理)、gcc –S(编译)、as(汇编)、ld(链接)最后还可以再通过ld指令来生成可执行文件hello。

在shell中启动指令，执行fork()产生子进程，此时hello.c从Program(程序)变为Process(进程)，便为hello的P2P。

而后在子进程中调用execve()，映射虚拟内存，而后程序开始时载入进物理内存，进入CPU处理，进入main函数执行目标代码，CPU为执行文件hello分配时间片，执行逻辑控制流，根据汇编语言指令执行取指、译码、执行、更新等操作。内存管理器和CPU在执行过程中通过L1、L2、L3三级缓存和TLB多级页表在物理内存中取的数据，通过I\O系统根据代码指令进行输出。当程序运行结束时回收进程，释放内存，删除与执行程序相关的数据结构，便为hello的O2O。

## 1.2 环境与工具

硬件环境：X64 CPU Intel Core i7 6700HQ; 3.2GHz; 16G RAM; 1TB HD Disk

软件环境：Windows10，Ubuntu 18.04.1LTS

开发与调试工具：gedit、vim、gcc、edb、IDA Pro64、readelf、hexedit

## 1.3 中间结果

列出你为编写本论文，生成的中间结果文件的名字，文件的作用等。

|  |  |
| --- | --- |
| 文件名字 | 文件作用 |
| hello.i | 预处理产生文件 |
| hello.s | 编译产生文件 |
| hello.o | 汇编产生文件 |
| hello.out | 链接产生文件 |
| hello | 链接产生可执行文件 |
| hello.elf | hello.o的elf文件 |
| hello-d.txt | hello反汇编生成文件 |
| hello-a.txt | hello的elf文件 |
| hello-d2.txt | hello.o反汇编生成的文件 |

## 1.4 本章小结

主要介绍的hello的P2P以及O2O过程，并介绍了此次大作业所使用的硬件设备以及软件环境，并且列出了此次大作业产生的一系列文件

# 第2章 预处理

## 2.1 预处理的概念与作用

**概念：**

在计算机科学中，预处理器是程序中处理输入数据，产生能用来输入到其他程序的数据的程序。输出被称为输入数据预处理过的形式，常用在之后的程序比如编译器中。所作处理的数量和种类依赖于预处理器的类型，一些预处理器只能够执行相对简单的文本替换和宏展开，而另一些则有着完全成熟的编程语言的能力。

一个来自计算机编程的常见的例子是在进行下一步编译之前，对源代码执行处理。在一些计算机语言（例如：C语言）中有一个叫做预处理的翻译阶段。

采用以'#'为行首的指示。

**作用：**

1. 文件包含：通过#include为文件的引用(库文件)组合源程序正文
2. 条件编译：#if、#endif等为进行编译时有选择的挑选，注释掉一些指定代码，以达到版本，防止对文件重复包含的作用。
3. 布局控制：#progma为编译程序提供非常规的控制流信息。
4. 宏替换：#define，可以定义符号常量、函数功能、重新命名、字符串拼接等功能。

## 2.2在Ubuntu下预处理的命令

命令：cpp hello.c > hello.i

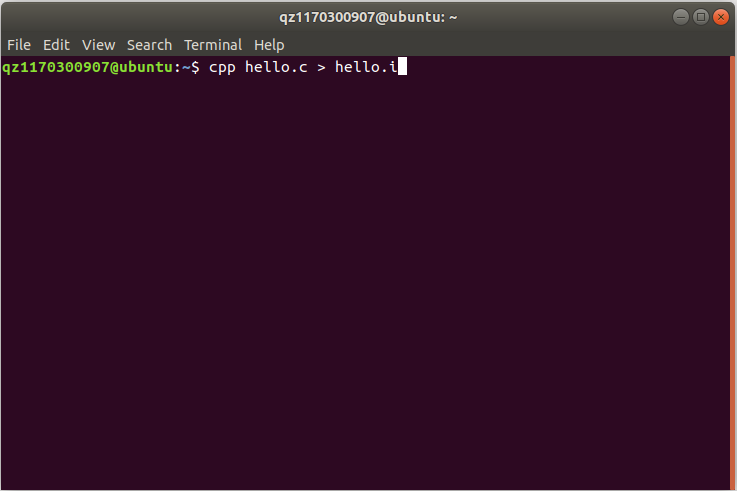


图2.2预处理指令

## 2.3 Hello的预处理结果解析

预处理后的hello.i文件在文件开头通过宏定义和externa等定义了一些程序基础的常量，extern定义的全局变量并不占据内存空间(通过查询资料得知)。

在main之前调用了linux对应文件夹中的.h头文件。

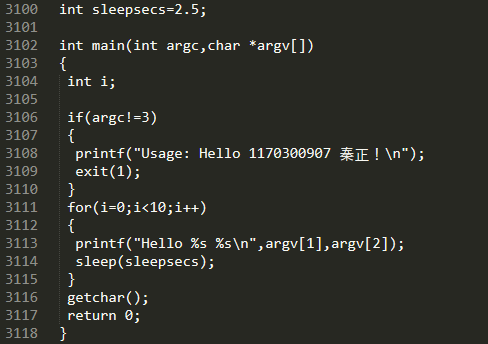


图2.3.1 main函数预处理结果

main函数加载了此程序调用的<stdio.h>、<unistd.h>、<stdlib.h>三个头文件

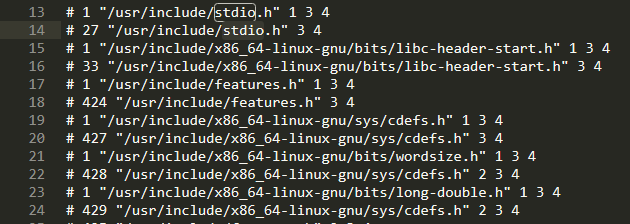


图2.3.2 stdio.h预处理结果

如图所示为stdio.h，执行cpp指令后进入linux的/usr/include文件夹中调用stdio.h，后由于stdio中有类似于下图的#定义语句和调用语句，cpp会根据代码继

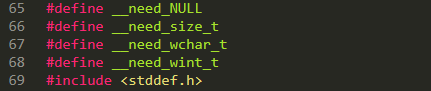


图2.3.3 stdio.h部分代码

续展开后处理，.h文件中的#if、#ifdef等让cpp指令对其进行有选择的编译决定是否执行定义中的语句。

## 2.4 本章小结

本章对于hello.c的预处理进行了解释，介绍了预处理的概念以及预处理的作用，列举出了预处理所处理的不同种类的预处理指令，并对hello.i文件进行了解析。

# 第3章 编译

## 3.1 编译的概念与作用

## **概念：**

## 编译将用某种编程语言写成的源代码(原始语言)，转换成另一种编程语言。便于人编写、阅读、维护的高级计算机语言所写作的源代码程序，翻译为计算机能解读、运行的低阶机器语言的程序，也就是可执行文件。

**作用：**

将便于人编写阅读的高级程序语言转换成计算机已与运行的汇编怨言以让计算机能够运行程序。

## 3.2 在Ubuntu下编译的命令

命令：gcc –S hello.c –o hello.s

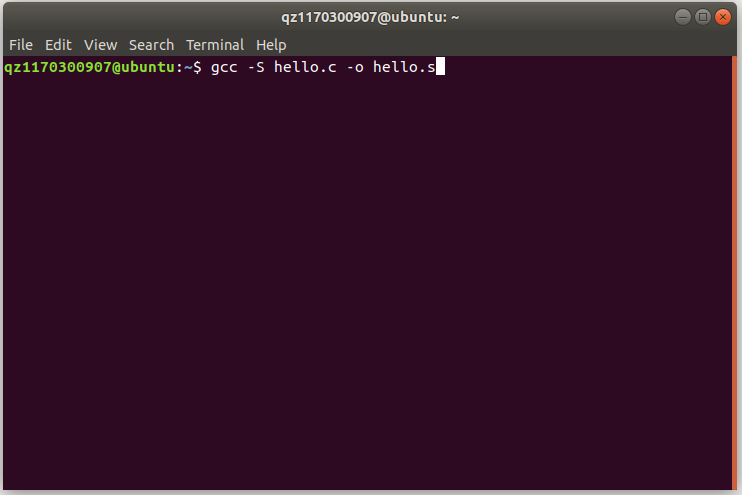


图3.2 编译指令

## 3.3 Hello的编译结果解析

3.3.1 编译关键词

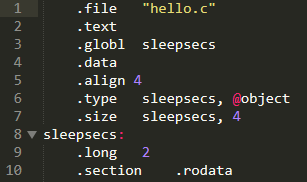


图3.3.1 hello.s编译关键词

|  |  |
| --- | --- |
| 关键词 | 作用 |
| .file | 声明源文件 |
| .text | 保存代码 |
| .globl | 声明全局变量 |
| .data | 数据段 |
| .align | 声明对指令或者数据的存放地址进行对齐的方式 |
| .section | 把代码划分成若干段 |
| .long | 声明long类型，其他等同 |
| .type | 用来指定是函数类型或是对象类型 |
| .size | 声明大小 |

3.3.2 数据

hello.s中定义的数据类型有：⑴字符串，⑵整型，⑶数组

1. 字符串：

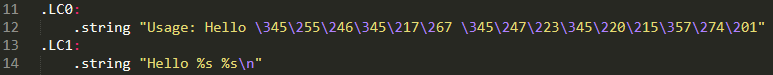


图3.3.2.1 字符串LC0，LC1

hello.s声明了.LC0,.LC1两个字符串

1. LC0字符串作为第一个printf的参数传入，字符串” \345\255\246\345\217\267 \345\247\223\345\220\215\357\274\201”为UTF-8编码的汉字，一个汉字占据三个字节
2. LC1字符串作为第二个printf的参数传入。输出的字符串为getchar所获取的两个字符串。
3. 整型：

hello.s声明了三个整形变量分别为sleepsecs、i、argc、立即数

* 1. sleepsecs

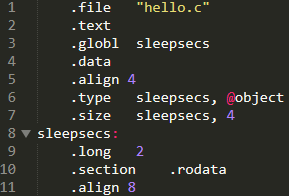


图3.3.2.2 整型slppesecs

sleepsecs在hello.c中被声明为全局变量，类型为int，并且已被赋值由上图可知在hello.s中定义在.data段，并且在.data段设置成对齐方式为4，类型为对象，大小为4字节，设置为long类型值为2

* 1. i

在hello.s中int i被定义在栈上%rbp-4的位置上占据4个字节

* 1. argc

argc作为main函数的第一个参数传入

* 1. 立即数

其他整形数据的出现都是以立即数的形式出现的，直接硬编码在汇编代码中。

1. 数组

hello.c中涉及的数组形式为main函数的参数函数执行时输入的命令行， argv 作为存放 char 指针的数组同时是第二个参数传入。

argv[]为char类型单个数组元素的大小为8个字节起始地址为argv，传入的两个参数分别记为argv[1]，argv[2]，值为getchar()在函数运行时获取的两个%s，后通过指令取出了数组储存值。

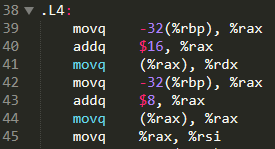


图3.3.2.3 argv数组

* + 1. 赋值

hello.c文件中赋值操作共有两次：⑴int sleepsecs = 2.5 ⑵int i = 0

⑴ sleepsecs被定义为全局变量被定义为long类型，值为2

⑵ i的赋值通过mov操作来完成

对于mov操作来说可分为movb（8位）、movw（16位）、movl（32位）、movq（64位）

3.3.4类型转换

hello.c中的类型转换涉及的是隐式类型转换，浮点数2.5被赋值为整型



图3.3.4 隐式类型转换

C语言的取整方式有4种：⑴直接赋值给整数变量，此时直接舍去小数部分 ⑵‘/’取整算符此时结果与C语言编译器有关 ⑶ floor(x)函数，返回的是小于等于x的整数 ⑷ ceil(x)函数，返回的是大于等于x的整数

根据以上四种情况所以sleepsecs被赋值为2

3.3.5算数操作

|  |  |  |
| --- | --- | --- |
| 指令 | 语法 | 功能 |
| ADD,ADC | ADD OP1 OP2  ADC OP1 OP2 | 加法指令 |
| SUB,SBB | SUB OP1 OP2  SBB OP1 OP2 | 减法指令 |
| INC,DEC | INC OP  DEC OP | 将OP值减一 |
| NEG | NEG OP | 将OP的符号反相 |
| MUL,IMUL | MUL OP  IMUL OP | 乘法指令 |
| DIV,IDIV | DIV OP  IDIV OP | 除法指令 |
| CBW,CWD | CBW CWD | 有符号数扩展 |

hello.s中的算数操作有：

⑴i++ 在for循环中i累加，在hello.s中的汇编操作为

⑵leaq指令中地址%rip被加上了.LC0，.LC1并分别传给了%rdi进行操作





图3.3.5 地址加操作

3.3.6关系操作

|  |  |  |
| --- | --- | --- |
| 指令 | 语法 | 功能 |
| CMP | CMP OP1 OP2 | 比较OP1和OP2 |
| TEST | TEST OP1 OP2 | 测试 |
| JN\JE… | JN & | 跳转到地址 |

hello.s中的操作有：

1. argc!=3判断argc若不为三则执行后面的指令，在hello.s中先将寄存器中的值给到-20(%rbp)后在cmpl $3 -20(%rbp)即为判断后执行下一条je语句
2. i<10 判断i是否大于等于10跳出for循环，在hello.s中比较-4(%rbp)与9的大小即为i的值与9进行比较，判断后决定是否执行下一条je语句

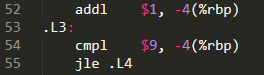


图3.3.6.1 i判断大小

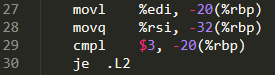


图3.3.6.2 aegc判断大小

3.3.7数组结构

见3.3.1，该程序定义了一个char类型的数组，名字为argv，单个元素8个字节。

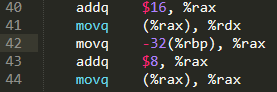


图3.3.7 argv数组

3.3.8控制转移

|  |  |  |
| --- | --- | --- |
| 指令 | 语法 | 功能 |
| JN\JE… | JN & | 跳转到地址 |

hello.s中的控制转移有：

1. if (argv!=3)：当argv不等于 3 的时候执行程序段中的代码。对标志位ZF进行判断，ZF=1即相等时跳转到.L2

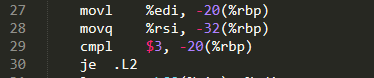


图3.3.8.1 argc跳转

1. for(i=0;i<10;i++)运行时先无条件跳转到位于循环体.L4之后的比较代码，使用cmpl进行比较，如果i<=9，则跳入.L4 for循环体执行，否则说明循环结束，顺序执行for之后的逻辑。

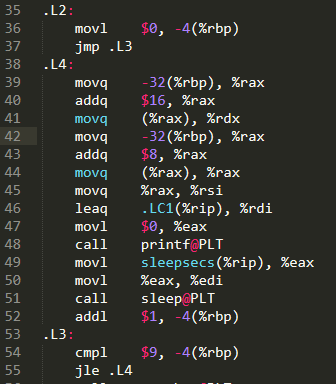


图3.3.8.2 for循环判断

3.3.9函数操作

函数操作包括参数传递、函数调用、函数返回

对于64位编译来说，传参顺序如下

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7~~ |
| %rdi | %rsi | %rdx | %rcx | %r8 | %r9 | 栈 |

该程序设计的函数操作有：

1. main函数：
2. 参数传递：将int argc, char argv[]传入main函数。
3. 函数调用：main函数调用了printf(),exit(),sleep(),getchar()四个函数
4. 函数返回：当main函数正常运行结束后返回1
5. printf函数：

① 参数传递：将需要输出的变量传入的printf函数

② 函数返回：printf函数返回所需要打印的值

⑶ exit函数：

① 参数传递：传入了1作为参数

② 函数返回：exit(1)表示程序出现异常时退出

⑷ sleep函数：

① 参数传递：传入了sleepsecs

② 函数返回：传入了sleepsecs之后，程序在传参大小时间过后继续进行

⑸ getchar函数：

① 函数返回：函数返回输入字符的ASCII码或者EOF表示输入有误

## 3.4 本章小结

在本章的工作中，解释了编译器如何处理C语言的各个数据类型以及C语言执行的一系列的操作，从头到尾对与hello.s进行了分析，对hello.s有了更进一步的了解。

# 第4章 汇编

## 4.1 汇编的概念与作用

**概念：**

把汇编语言翻译成机器语言的过程叫做汇编

**作用：**

.o是一个二进制文件其中包含程序的指令，可让计算机进行操作

## 4.2 在Ubuntu下汇编的命令

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

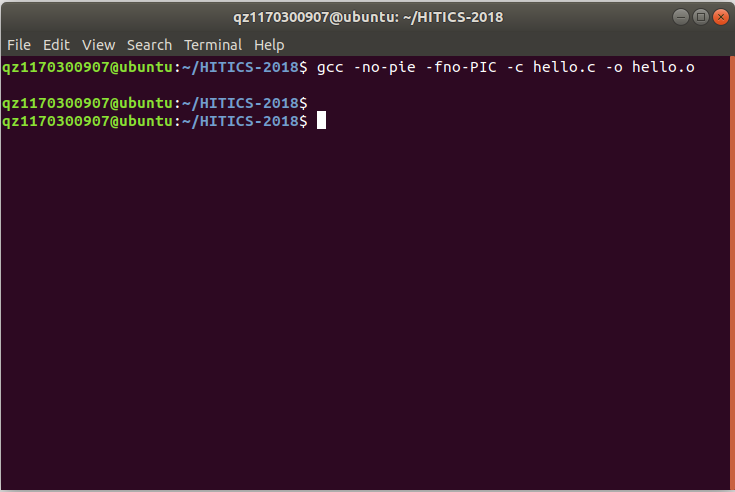


图4.2 汇编指令

## 4.3 可重定位目标elf格式

⑴ ELF HEADER:

elf头大小为64字节，其内容可以表示其为elf对象，目标文件类型，头文件版本，处理器体系结构，字节头部表的文件偏移以及节头部表中条目的大小和数量等信息。

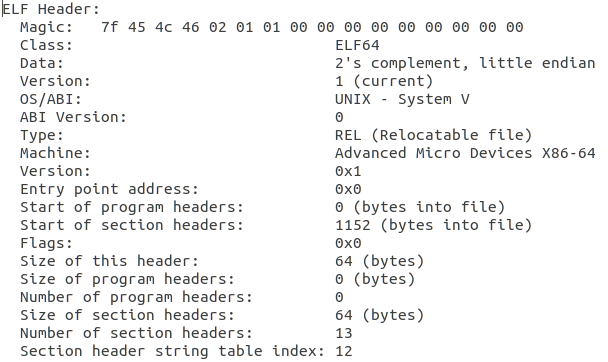
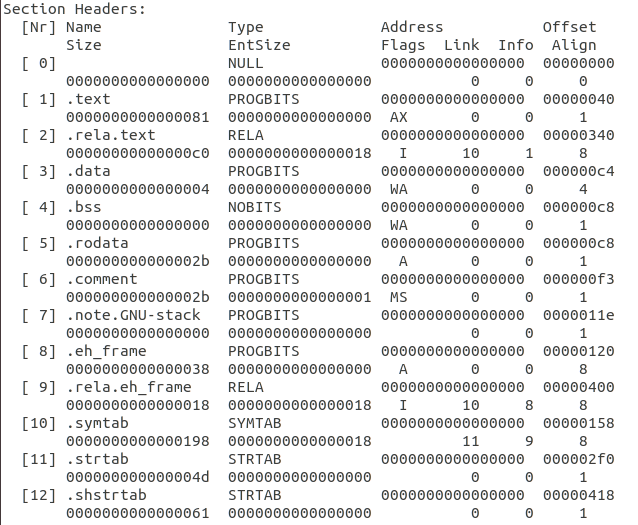


图4.3.1 elf头

⑵ SECTION HEADER TABLE：

图4.3.2 节头数据表

节头表包含了文件中节的类型，名称，位置，大小等信息，通过节头表包含的信息，找到文件头的偏移量，根据索引找到指定的section

⑶ RELOCATION SECTION

重定位节.text段给出8个条目，包含有偏移量、信息、类型、符号值、符号名称+加数5个内容。

eh\_frame段包含一个条目

R\_X86\_64\_PC32是和R\_X86\_64\_32相对应的，比较常见的重定位类型。前者是重定位一个使用32位PC相对地址的引用，而后者是重定位一个使用32位绝对地址的引用。

重定位PC相对引用的重定位算法为：

1. refaddr = ADDR(s) + r.offset;
2. \*refptr = (unsigned) (ADDR(r.symbol) + r.addend – refaddr);

假设算法运行时，链接器为每个节（用ADDR(s)表示）和每个符号都选择了运行时地址（用ADDR(r.symbol)）表示。拿.rodata的重定位为例，它的重定位地址为refptr.则应先计算引用的运行时地址refaddr=ADDR(s)+ r.offset, .rodata的offset为0x18，ADDR(s)是由链接器确定的。然后，更新引用，refptr = (unsigned) (ADDR(r.symbol) + r.addend – refaddr), .rodata的ADDR(r.symbol)也是由链接器确定的，addend查表可知为-4，refaddr已经算出来了，所以，.rodata的重定位地址我们就可以算出来了。

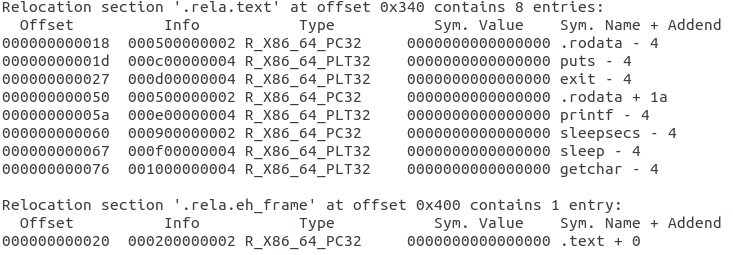
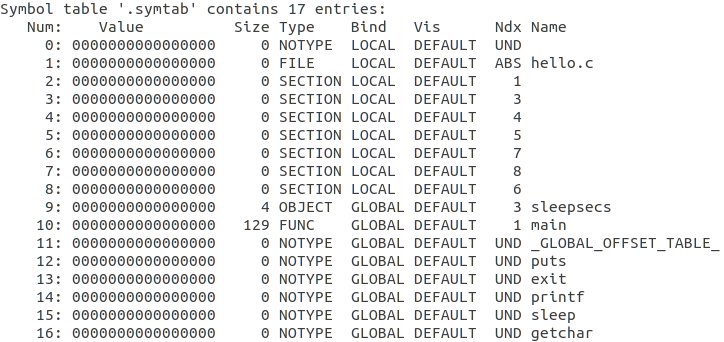


图4.3.3 重定位节

⑷ 符号表

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



## 4.4 Hello.o的结果解析

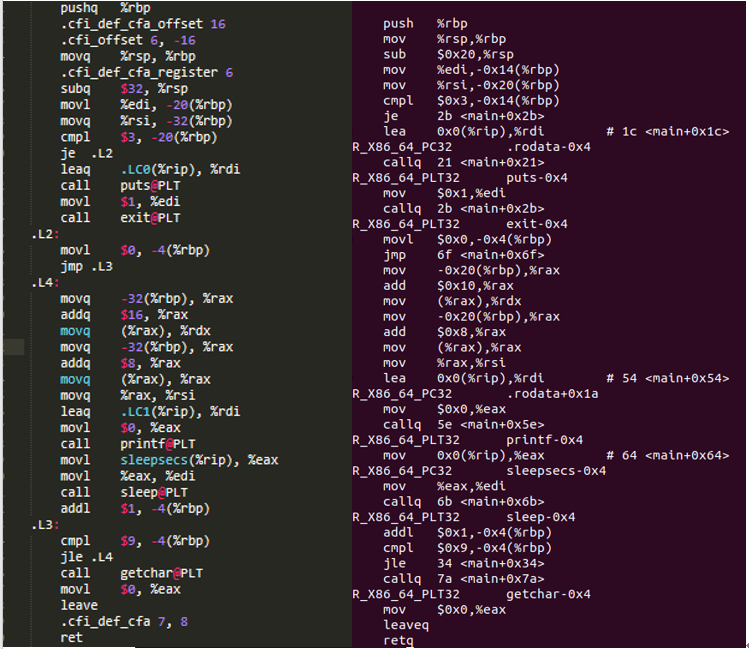


图4.4 hello.s与hello.o反汇编对比图

hello.o是有hello.o经由gcc编译器通过as汇编指令后得到的，但是hello.o经过反汇编后二者却产生了细微的差别。如上图所示。

主要原因在于：

⑴ 分支转移：反汇编代码跳转指令的操作数使用的不是段名称如.L3，因为段名称只是在汇编语言中便于编写的助记符，所以在汇编成机器语言之后显然不存在，而是确定的地址。

⑵ 函数调用：在.s文件中，函数调用之后直接跟着函数名称，而在反汇编程序中，call的目标地址是当前下一条指令。这是因为hello.c中调用的函数都是共享库中的函数，最终需要通过动态链接器才能确定函数的运行时执行地址，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（目标地址正是下一条指令），然后在.rela.text节中为其添加重定位条目，等待静态链接的进一步确定。

⑶ 全局变量访问：在.s文件中，访问rodata（printf中的字符串），使用段名称+%rip，在反汇编代码中0+%rip，因为rodata中数据地址也是在运行时确定，故访问也需要重定位。所以在汇编成为机器语言时，将操作数设置为全0并添加重定位条目。

## 4.5 本章小结

在本章中了解了程序汇编的过程，了解了汇编之后产生的可重定位的各种信息的具体内容，了解了汇编和反汇编之中存在的各种差异。

# 第5章 链接

## 5.1 链接的概念与作用

**概念：**

链接是将各种代码和数据片段收集并组合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 -L/usr/lib/gcc/x86\_64-linux-gnu/7 -L/usr/lib/x86\_64-linux-gnu -L/usr/lib -L/lib/x86\_64-linux-gnu -L/lib/../lib hello.o -lc /usr/lib/gcc/x86\_64-linux-gnu/7/crtend.o /usr/lib/x86\_64-linux-gnu/crtn.o -z relro -o hello

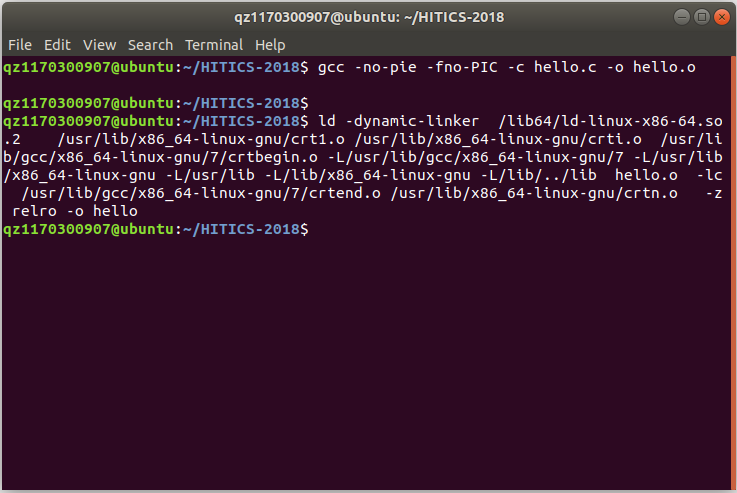


图5.2 链接命令

## 5.3 可执行目标文件hello的格式

elf头大小为64字节，其内容可以表示其为elf对象，目标文件类型，头文件版本，处理器体系结构，字节头部表的文件偏移以及节头部表中条目的大小和数量等信息。

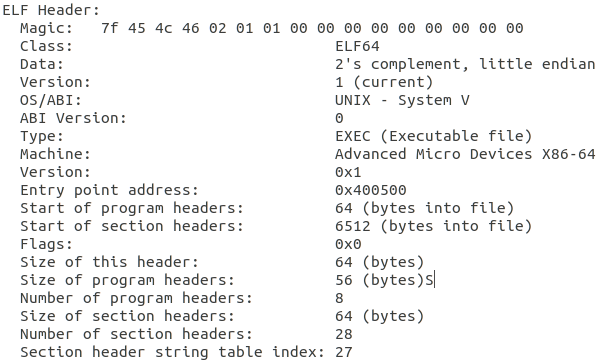
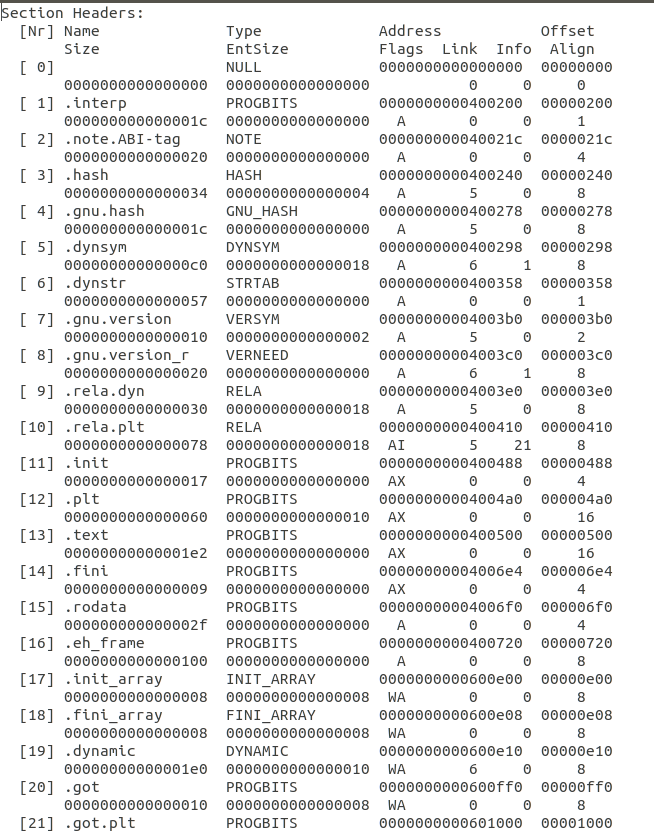


图5.3.1 elf节头

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。



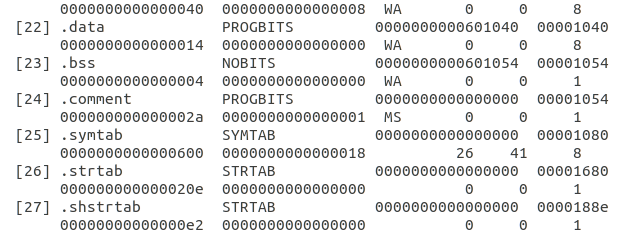


图5.3.2 节头

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。

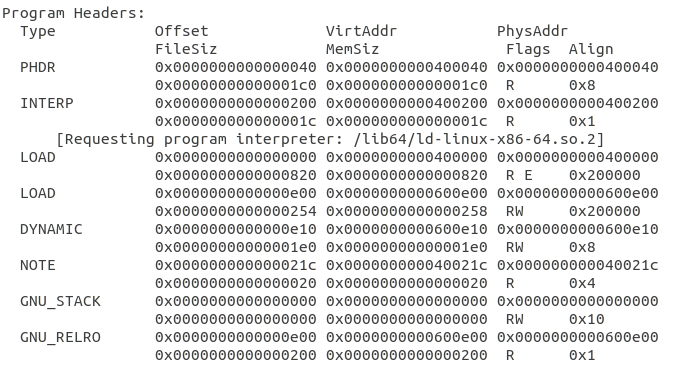


图5.3.3 程序头

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。

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。

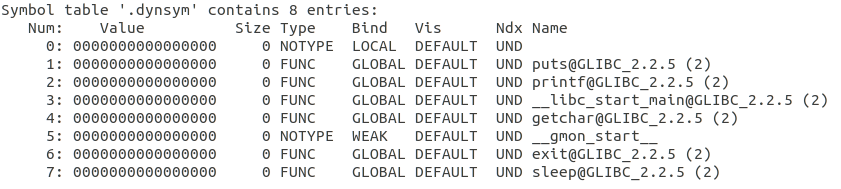
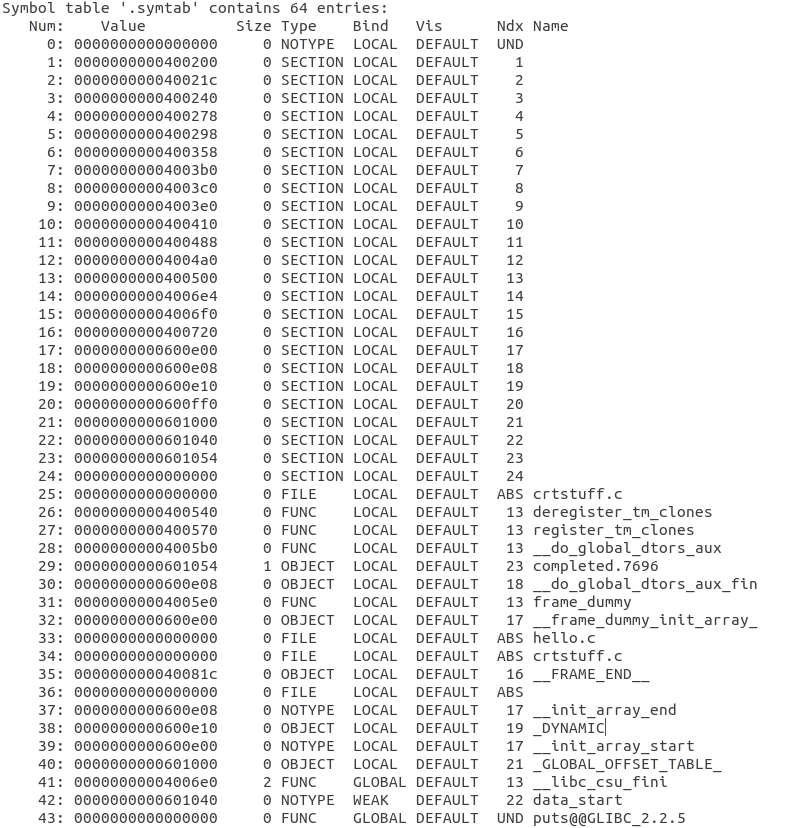


图5.3.4 与动态链接相关的符号表



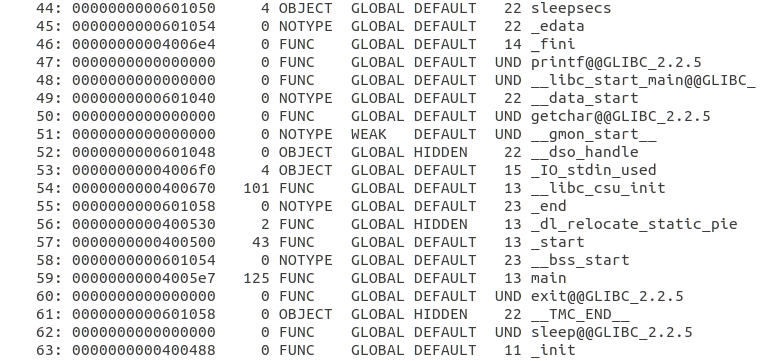


图5.3.4 全部符号表

## 5.4 hello的虚拟地址空间

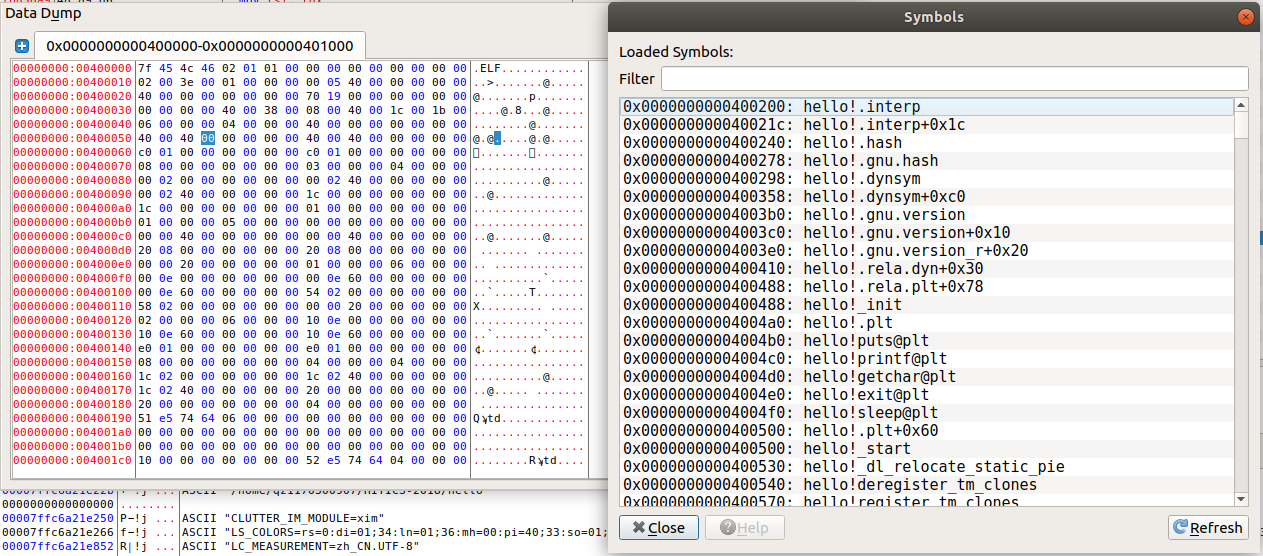


图5.4 虚拟地址空间

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

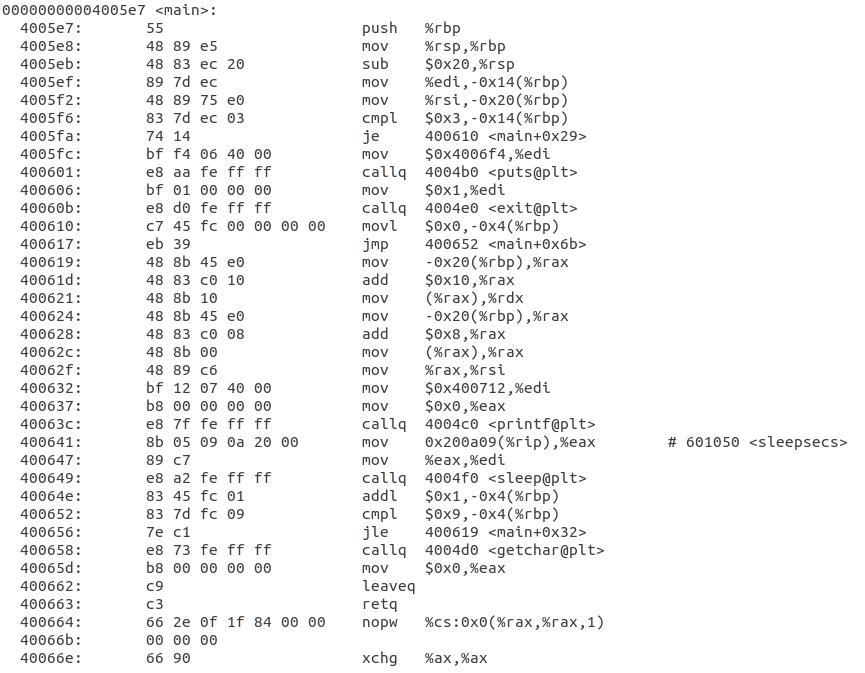


图5.5.1 链接后反汇编代码

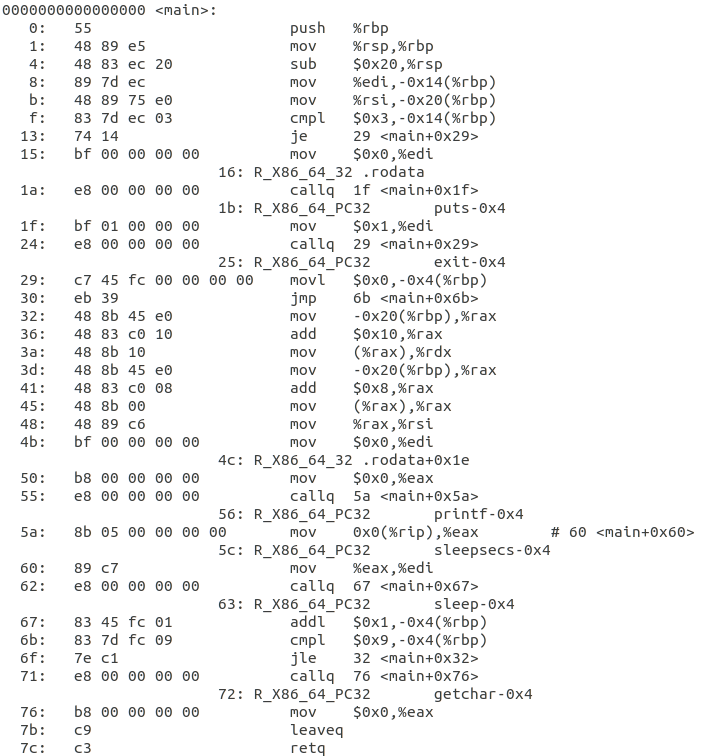


图5.5.2 hello.o反汇编

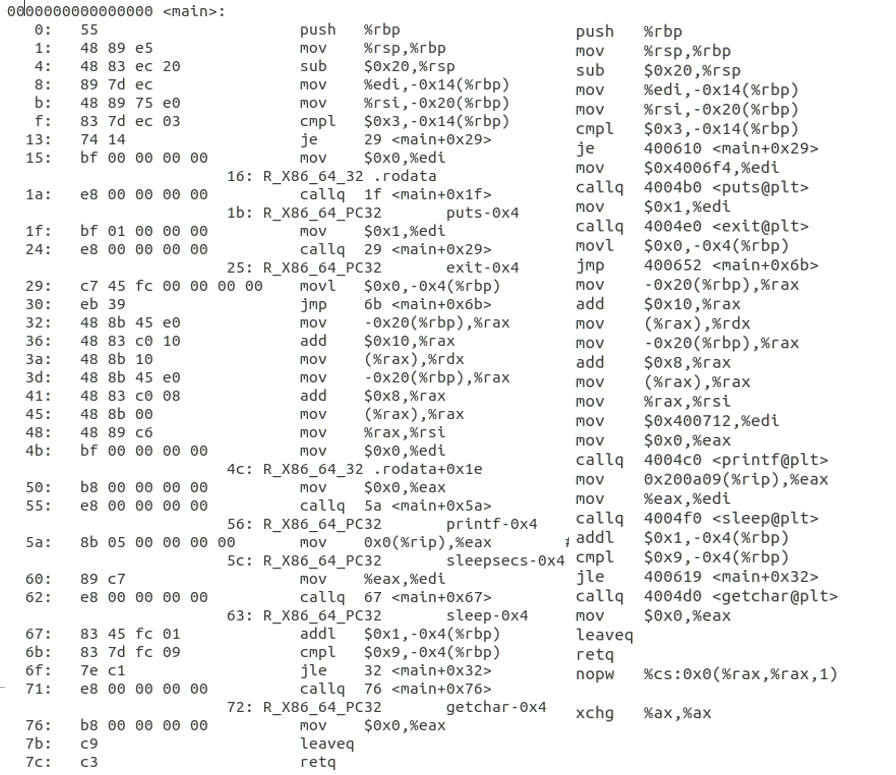


图5.5.3 二者对比图

可以发现，经过链接的hello汇编生成的文件还是和hello.o反汇编生成的文件有所区别。

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：

以puts为例

ADDR(s)=ADDR（main）=0x4005e7

ADDR(r.symbol)=0x4004b0



图5.5.4 puts地址

refaddr=ADDR(s)+0x1d=0x400604

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



图5.5.5 最终地址验证

## 5.6 hello的执行流程

|  |  |
| --- | --- |
| 函数名称 | 地址 |
| ld-2.27.so!\_dl\_start | 0x7f4ea4db4ea0 |
| hello!\_init | 0x0000000000400488 |
| hello!main | 0x00000000004005e7 |
| 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!\_dl\_fixup | 0x00007f83ffb93f64 |
| ld-2.27.so!\_dl\_lookup\_symbol\_x | 0x00007f83ffb8f0b0 |
| ld-2.27.so!do\_lookup\_x | 0x00007f83ffb8e240 |
| 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 |
| libc-2.27.so!exit | 0x00007fce8c889128 |

## 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,0x601008he 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

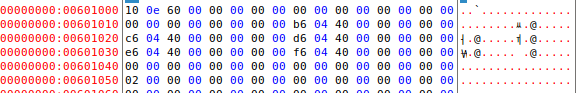


图5.7.1 调用前

调用之后的got.plt

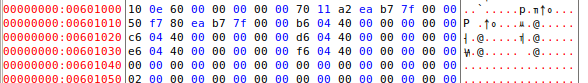


图5.7.2 调用后

重定位表：

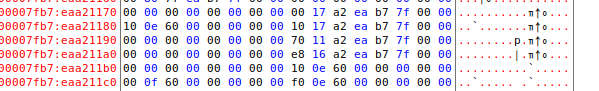


图5.7.3 重定位表

## 5.8 本章小结

本章介绍了链接的相关内容，通过反汇编对比分析elf信息，利用调试工具了解了重定位过程，虚拟地址空间等。并且分析了链接前后程序的区别。

# 第6章 hello进程管理

## 6.1 进程的概念与作用

**概念：**

是指计算机中已运行的程序。进程为曾经是分时系统的基本运作单位。在面向进程设计的系统（如早期的UNIX，Linux 2.4及更早的版本）中，进程是程序的基本执行实体；在面向线程设计的系统（如当代多数操作系统、Linux 2.6及更新的版本）中，进程本身不是基本运行单位，而是线程的容器。程序本身只是指令、数据及其组织形式的描述，进程才是程序（那些指令和数据）的真正运行实例。若干进程有可能与同一个程序相关系，且每个进程皆可以同步（循序）或异步（平行）的方式独立运行。现代计算机系统可在同一段时间内以进程的形式将多个程序加载到存储器中，并借由时间共享（或称时分复用），以在一个处理器上表现出同时（平行性）运行的感觉。同样的，使用多线程技术（多线程即每一个线程都代表一个进程内的一个独立执行上下文）的操作系统或计算机体系结构，同样程序的平行线程，可在多CPU主机或网络上真正同时运行（在不同的CPU上）。

**作用：**

给应用程序提供了关键的抽象，一个独立地逻辑控制流，提供一个假象，好像我们的程序独占地使用处理器，还提供了一个私有的地址空间，提供一个假象，好像我们的程序独占地使用内存系统。

## 6.2 简述壳Shell-bash的作用与处理流程

**概念：**

是一个命令处理器，通常运行于文本窗口中，并能执行用户直接输入的命令。Bash还能从文件中读取命令，这样的文件称为脚本。和其他Unix shell 一样，它支持文件名替换（通配符匹配）、管道、here文档、命令替换、变量，以及条件判断和循环遍历的结构控制语句。包括关键字、语法在内的基本特性全部是从sh借鉴过来的。其他特性，例如历史命令，是从csh和ksh借鉴而来。总的来说，Bash虽然是一个满足POSIX规范的shell，但有很多扩展。

**流程：**

⑴读取命令

⑵分割字符串获得参数

⑶判断是否为内置命令，若是则执行

⑷若不是则调用相应的程序为其分配子进程并运行

⑸shell应该接受键盘输入信号，并对这些信号进行相应处理

## 6.3 Hello的fork进程创建过程

父函数通过fork()函数来创建子进程，其函数声明为

pid\_t fork(**void**);

hello的进程创建示意图如下：

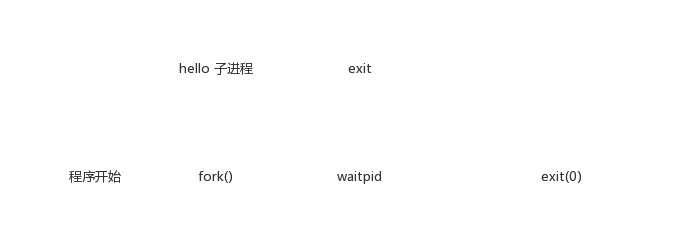


图6.3 fork子进程示意图

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。

通常父进程用waitpid函数来等待子进程终止或停止，其函数声明为：

pid\_t waitpid(pid\_t pid, **int** \*statusp, **int** options);

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。

## 6.4 Hello的execve过程

execve函数在当前进程的上下文中加载并运行一个新程序。函数声明如下：

**int** execve(**const** **char** \*filename, **const** **char** \*argv[], **const** **char** \*envp[]);

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。

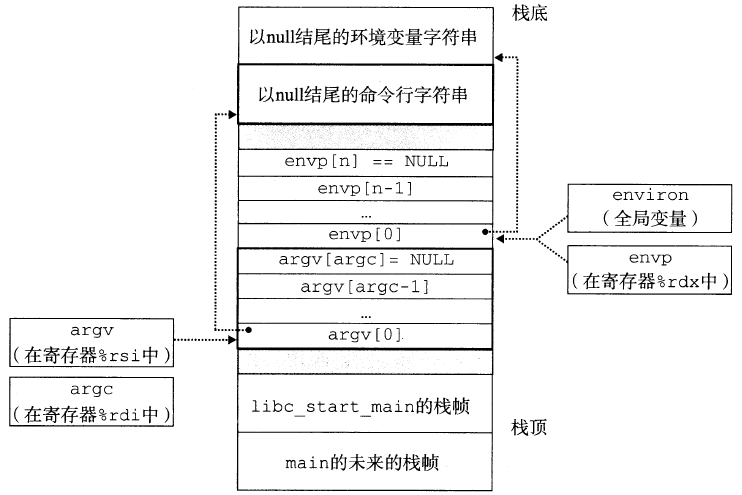


图6.4.1 一个新程序开始时，用户栈的典型组织结构

从栈底（高地址）到栈顶（低地址），首先是参数和环境字符串。栈往上紧随其后的是以null结尾的指针数组，其中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]。紧随环境变量数组之后的是以null结尾的argv[]数组，其中每个元素都指向栈中的一个参数字符串。在栈的顶部是系统启动函数libc\_start\_main的栈帧。

程序运行fork后，子进程调用exceve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 调用驻留在内存中的被称为启动加载器的操作系统代码来执行 hello 程序，加载器删除子进程现有的虚拟内存段，并创建一组新的代码、数据、堆和栈段。新的栈和堆段被初始化为零，通过将虚拟地址空间中的页映射到可执行文件的页大小的片，新的代码和数据段被初始化为可执行文件中的内容。最后加载器设置 PC 指向\_start 地址，\_start 最终调用 hello中的 main 函数。

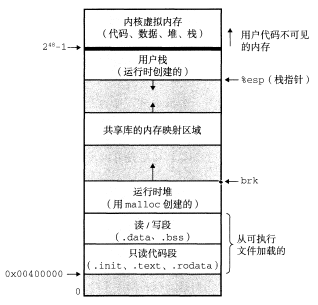


图6.4.2 创建的进程地址空间

## 6.5 Hello的进程执行

结合进程上下文信息、进程时间片，阐述进程调度的过程，用户态与核心态转换等等。

hello在执行时，有自己的逻辑控制流。多个进程的逻辑控制流在时间上可以交错，表现为交替运行。每个进程执行它的流的一部分，然后被抢占（暂时挂起），然后轮到其它进程。

一个逻辑流的执行在时间上和另一个流重叠，成为并发流，这两个流并发地运行。一个进程执行它的控制流的一部分的每一时间段叫时间片。

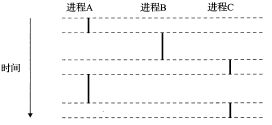


图6.5.1 逻辑控制流

如上图所示，处理器的一个物理控制流被分成了三个逻辑流，每个进程一个。每个竖直的条表示一个进程的逻辑流的一部分。在这个例子中，三个逻辑流的执行是交错的。进程A运行了一会儿，然后是进程B开始运行到完成。然后，进程C运行了一会儿，进程A接着运行直到完成。最后，进程C可以运行到结束了。

hello的进程调度可以按照下图来理解：

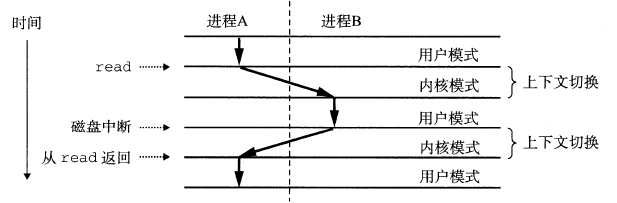


图6.5.2 hello进程切换示意图

hello的sleep的进程调度过程：当调用sleep之前，如果hello程序不被抢占则顺序执行，当hello执行到sleep函数时，会被挂起一段时间。挂起就是指进程被抢占，假如发生被抢占的情况，则进行上下文切换，上下文切换是由内核中调度器完成的，当内核调度新的进程运行后，它就会抢占当前进程，并进行⑴保存以前进程的上下文⑵恢复新恢复进程被保存的上下文，⑶将控制传递给这个新恢复的进程 ，来完成上下文切换。

具体为运行到sleep时，当前进程挂起2.5秒，如果请求的时间量到了，sleep返回0，否则返回还剩下的要休眠的秒数。调用sleep后，内核中的调度器将hello进程挂起，然后进入到内核模式由于hello调用sleep的这个过程没有显式地创建新的进程，所以，在hello被抢占了secs秒后，内核又会选择hello进程，恢复它被抢占时的上下文，并把控制交给它，这时，又回到了用户模式。

## 6.6 hello的异常与信号处理

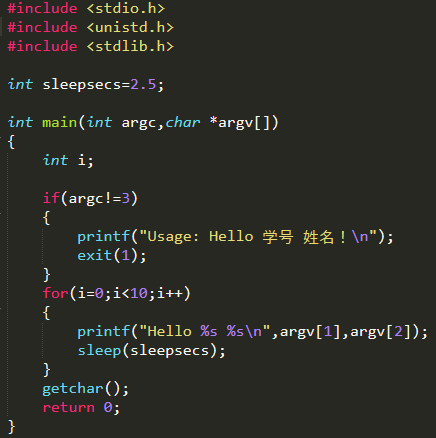


图6.6.1 hello.c文件

查看hello.c文件进行分析，首先，如果参数不为3，那么会打印一条默认语句，并异常退出。如果参数是3个，那么会执行一个循环，每次循环会使hello进程休眠2.5秒，休眠后又会恢复hello。而且循环里会输出一条格式字串，其中有输入的两个参数字串。循环结束后，有一个getchar()等待一个标准输入，然后就结束了。

**6.6.1正常运行**

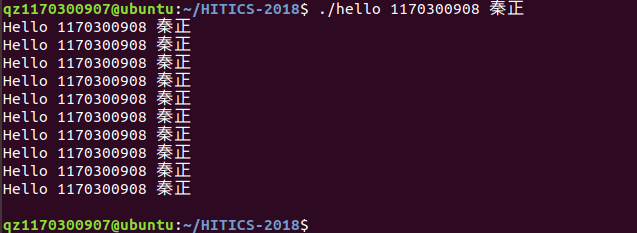


图6.6.2 正常运行截图

程序正产运行，运行结束后进程被回收

**6.6.2 Ctrl+C**

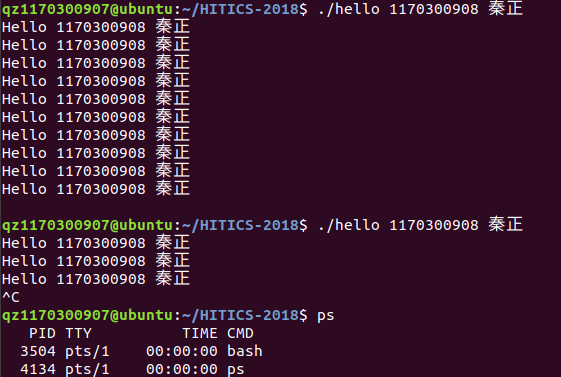


图6.6.3 Ctrl+C信号运行截图

shell父进程收到SIGINT信号，结束hello，并回收hello进程

**6.6.3 Ctrl+Z**

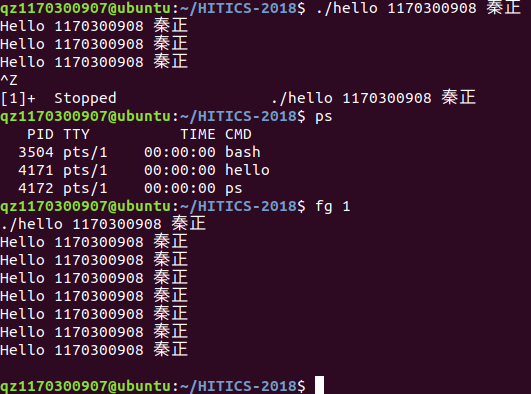


图6.6.4 Ctrl+Z信号运行截图

输出三条hello信息后键入Ctrl+Z信号，父进程收到信号后，将hello进程挂起，查看可发现hello进程并未被回收，调用fg 1后再次被调到前台执行，输出余下的字符串。

**6.6.4 随意键入**

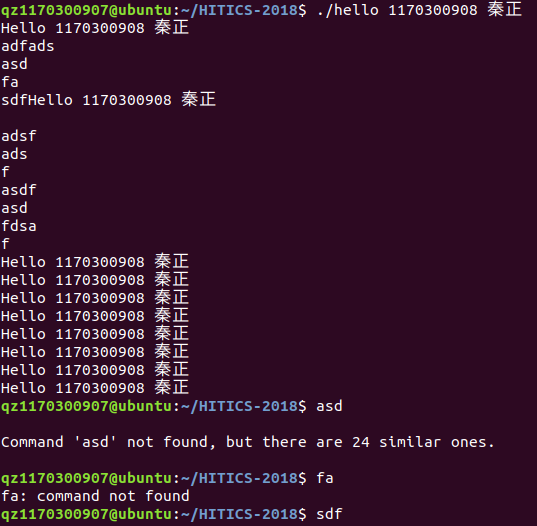


图6.6.5 随意键入运行截图

在运行时随意键入，输入的字符串被保存在了标准输入的缓冲区内在得到’\n’指令时，被认为是指令，在hello执行完后被输入到终端中执行。

## 6.7本章小结

本章介绍了进程的相关内容，介绍了shell-bash(壳)，建立了一系列指令在日常处理中起作用，同时也介绍了fork()，exceve()两函数的功能和执行过程，介绍了信号处理和异常处理。

# 第7章 hello的存储管理

## 7.1 hello的存储器地址空间

**逻辑地址：**CPU所生成的地址。逻辑地址是内部和编程使用的、并不唯一。例如，你在进行C语言指针编程中，可以读取指针变量本身值(&操作)，实际上这个值就是逻辑地址，它是相对于你当前进程数据段的地址（偏移地址），不和绝对物理地址相干。

**物理地址：**加载到内存地址寄存器中的地址，内存单元的真正地址。在前端总线上传输的内存地址都是物理内存地址，编号从0开始一直到可用物理内存的最高端。这些数字被北桥(Nortbridge chip)映射到实际的内存条上。物理地址是明确的、最终用在总线上的编号，不必转换，不必分页，也没有特权级检查(no translation, no paging, no privilege checks)。

**线性地址：**线性地址是一个32位无符号整数，可以用来表示高达4GB的地址，也就是，高达4294967296个内存单元。线性地址通常用十六进制数字表示，值的范围从0x00000000到0xffffffff）。程序代码会产生逻辑地址，通过逻辑地址变换就可以生成一个线性地址。如果启用了分页机制，那么线性地址可以再经过变换以产生一个物理地址。当采用4KB分页大小的时候，线性地址的高20位为页目录项在页目录表中的编号，中间十位为页表中的页号，其低12位则为偏移地址。如果是使用4MB分页机制，则高10位页号，低22位为偏移地址。如果没有启用分页机制，那么线性地址直接就是物理地址。

**虚拟地址：**是Windows程序时运行在386保护模式下，这样程序访问存储器所使用的逻辑地址称为虚拟地址，与实地址模式下的分段地址类似，虚拟地址也可以写为“段：偏移量”的形式，这里的段是指段选择器。

对于当前的Intel系统采用段页式存储管理(MMU实现)

**段式管理：逻辑地址->线性地址==虚拟地址**

**页式管理：虚拟地址->物理地址**

edb调试中看到的hello的指令地址都是16位的虚拟地址，有些访问指令的地址也是逻辑地址，在程序中虚拟地址和逻辑地址没有明显的界限。通常来说我们是看不到程序的物理地址的。至于线性地址，只是一个地址的概念。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

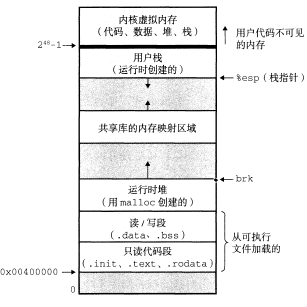


图7.2.1 内存映像

段式管理（segmentation），是指把一个程序分成若干个段（segment）进行存储，每个段都是一个逻辑实体（logical entity）。

用户栈是栈段寄存器，共享库的内存映射区域和运行时堆都是辅助段寄存器，读/写段是数据段寄存器，只读代码段是代码段寄存器。

段寄存器(16位)用于存放段选择符:

CS(代码段)：程序代码所在段

SS(栈段)：栈区所在段

DS(数据段)：全局静态数据区所在段

其他3个段寄存器ES、GS和FS可指向任意数据段

段选择符各字段含义：

CS寄存器中的RPL字段表示CPU的当前特权级（Current Privilege Level，CPL） 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。出于环保护机制，内核工作在第0环，用户工作在第3环，中间环留给中间软件用。Linux仅用第0环和第3环。TI=0，选择全局描述符表(GDT)，TI=1，选择局部描述符表(LDT)。



图7.2.2 代码段的段选择符

段描述符是一种数据结构，实际上就是段表项，分为用户的代码段和数据段描述符，还有系统控制端描述符。

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、局部描述符表LDT：存放某任务（即用户进程）专用的描述符

中断描述符表IDT：包含256个中断门、陷阱门和任务门描述符

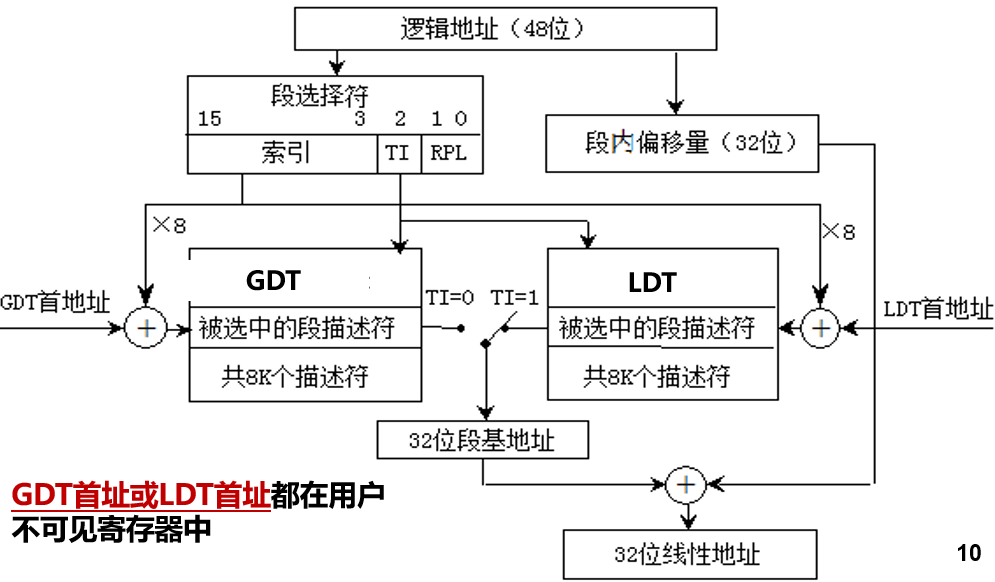


图7.2.3 逻辑地址向线性地址转换

## 7.3 Hello的线性地址到物理地址的变换-页式管理

概念上而言， 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。和存储器层次结构中其他缓存一样，磁盘（较低层）上的数据被分割成块，这些块作为磁盘和主存（较高层）之间的传输单元。 VM系统通过将虚拟内存分割为称为虚 拟页(VirtualPage, VP)的大小固定的块来处理这个问题。 每个虚拟页的大小为P= 护字节。 类似地，物理内存被分割为物理页(Physical Page, PP) , 大小也为P字节（物理页也被称为页帧(page frame) )。

为了有助于清晰理解存储层次结构中不同的缓存概念，我们将使用术语SRAM缓存 来表示位于CPU和主存之间的Ll、L2和L3高速缓存，并且用术语DRAM缓存来表示虚拟内存系统的缓存，它在主存中缓存虚拟页。

页表是一种数据结构，它用于计算机操作系统中的虚拟内存系统，其存储了虚拟地址到物理地址间的映射。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。操作系统负责维护页表的内容，以及在磁盘与DRAM之间来回传送页。最简单的分页表系统通常维护一个帧表和一个分页表。帧表处理帧映射信息。更高级系统中，帧表可以处理页属于的地址空间，统计信息和其他背景信息。分页表处理页的虚拟地址和物理帧的映射。还有一些辅助信息，如当前存在标识位(present bit)，脏数据标识位或已修改的标识位，地址空间或进程ID信息。

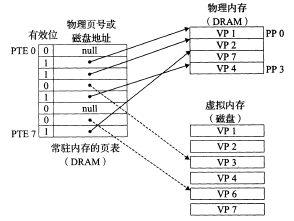


图7.3.1 页表

页表就是一个页表条目(PageTable Entry, PTE)的数组。虚拟地址空间中的每个页在页表中一固定偏移最处都有一个PTE。假设每个PTE是由一个有效位(validbit)和一个n位地址字段组成的。有效位表明了该虚拟页当前是否被缓存在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。如果没有设置有效位，那么一个空地址表示这个虚拟页还未被分配。否则，这个地址就指向该虚拟页在磁盘上的起始位置。

形式上来说，地址翻译是一个N元素的虚拟地址空间(VAS)中的元素和一个M元素 的物理地址空间(PAS)中元素之间的映射。

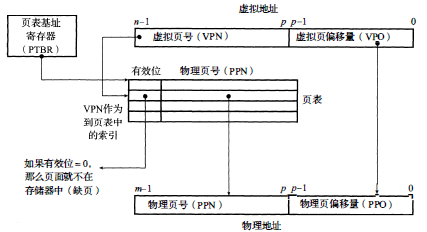


图7.3.2 页表的地址翻译

上图展示了MMU如何利用页表来实现这种映射。CPU中的一个控制寄存器，页表基址寄存器(PageTable Base Register, PTBR)指向当前页表。n位的虚拟地址包含两个部分：一个p位虚拟页面偏移(VirtualPage Offset, VPO)和一个(n-p) 位的虚拟页号(Virtual age Number, VPN)。MMU利用VPN来选择适当的PTE。例如，VPN0选择PTEO, VPN 1选择PTE1, 以此类推。将页表条目中物理页号(PhysicalPage Number, PPN)和虚拟 地址中的VPO串联起来，就得到相应的物理地址。注意，因为物理和虚拟页面都是P字节的，所以物理页面偏移(PhysicalPage Offset, PPO)和VPO是相同的。

## 7.4 TLB与四级页表支持下的VA到PA的变换

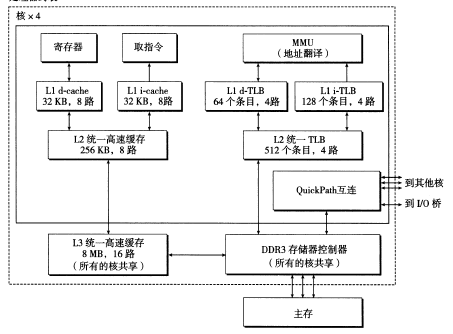


图7.4.1 i7内存系统

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。

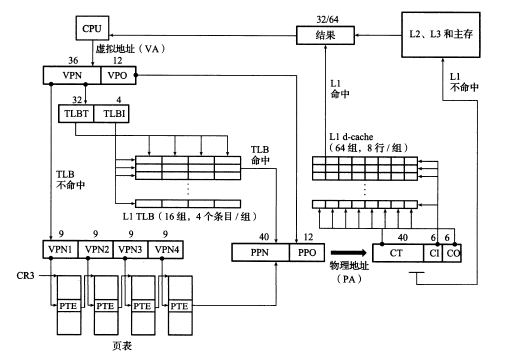


图7.4.2 i7地址翻译情况

Core i7 MMU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，每个片被用作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 PTE的偏移量，这个PTE包含L2页表的基地址。VPN2提供到一个PTE的偏移量，以此类推。

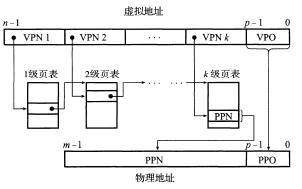


图7.4.3 k级页表地址翻译

VA到PA的变换的基础操作分为下面两种类：

⑴ 页面命中

笫l步：处理器生成一个虚拟地址， 并把它传送给MMU。

第2步：MMU生成PTE地址，并从高速缓存／主存请求得到它。

笫3步：高速缓存／主存向MMU返回PTE

第4步：MMU构造物理地址， 并把它传送给高速缓存／主存。

第5步：高速缓存／主存返回所请求的数据字给处理器 。

⑵ 缺页：页面命中完全是由硬件来处理的，与之不同的是，处理缺页要求硬件和操作系统内核协作完成，

第1步到笫3步：与上一情况前三步相同。

第4步：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。

笫5步：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。

第6步：缺页处理程序页面调入新的页面，并更新内存中的PTE。

第7步： 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执行了图7.4.4-b中的步骤之后，主存就会将所请求字返回给处理器。

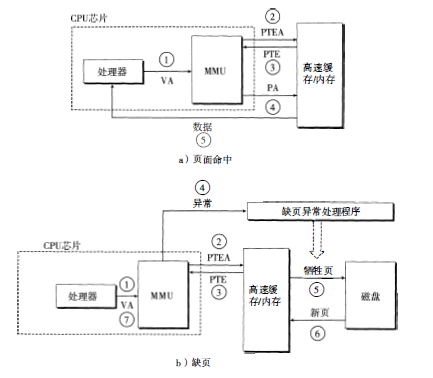


图7.4.4 页面命中和缺页操作图

## 7.5 三级Cache支持下的物理内存访问

在此仅讨论L1级缓存情况，由于L2，L3级缓存访问过程与L1相似。

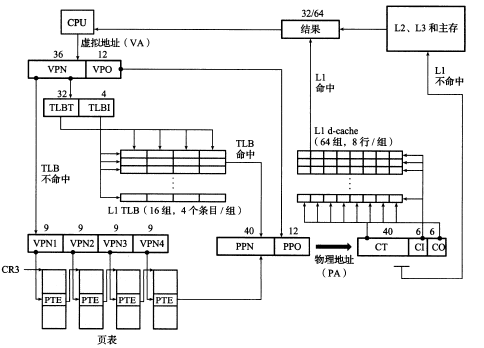


图7.5.1 i7地址翻译，右侧为缓存物理访存

物理内存访问，是基于MMU将虚拟地址翻译成物理地址之后，向cache中访问的。

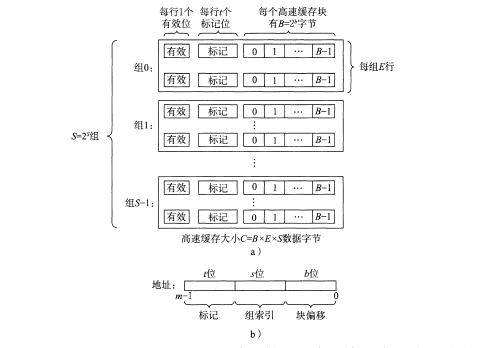


图7.5.2 高速缓存通用结构

在cache中物理地址寻址，按照三个步骤：组选择、行匹配和字选择。在冲突不命中时还会发生行替换。

高速缓存(S, E, B, m）被组织成一个有S=2s个高速缓存组(cache set)的数组。 每个组包含E 个高速缓存行(cache line).每个行是由一个B=2b字节的数据块(block)组成的， 一个有效位(valid bit) 指明这个行是否包含有意义的信息,还有t=m-(b+s)个标记位(tag bit)(是当前块的内存地址的位的一个子集），它们唯一地标识存储在这个高速缓存行中的块。

一般而言，高速缓存的结构可以用元组(S, E, B, m)来描述。 高速缓存的大小(或容量)C指的是所有块的大小的和。标记位和有效位不包括在内。因此,C=S×E×B。

高速缓存的结构将m个地址位划分为t个标记位，s个组索引位，和b个块偏移位。

在组选择中，cache按照物理地址的s个组索引位(S=2s)来定位该地址映射的组。

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，才可以说这一行中包含着地址的一个副本。也就是缓存命中了。

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。

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。

## 7.6 hello进程fork时的内存映射

当fork函数被shell进程调用时，内核为新进程创建各种数据结构，并分配给它一个唯一的PID，为了给这个新进程创建虚拟内存，它创建了当前进程的mm\_struct、区域结构和页表的原样副本。它将这两个进程的每个页面都标记为只读，并将两个进程中的每个区域结构都标记为私有的写时复制。

当fork在新进程中返回时，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。当这两个进程中的任一个后来进行写操作时，写时复制机制就会创建新页面，因此，也就为每个进程保持了私有地址空间的抽象概念。

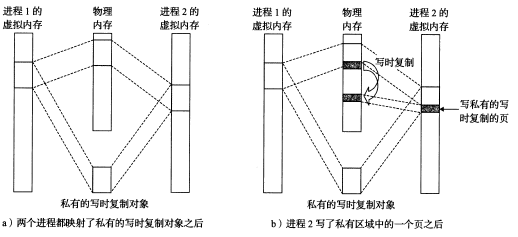


图7.6 一个私有的写时复制对象

## 7.7 hello进程execve时的内存映射

运行的当前进程调用了exceve()，其函数形式为：

execve("a.out", NULL, NULL);

hello调用execve后，execve在当前进程中加载并运行包含在可执行目标文件hello.out中的程序，用hello.out程序有效地替代了当前程序。加载并运行hello.out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a.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 a.out中。栈和堆区域也是请求二进制零的，初始长度为零。图7.7概括了私有区域的不同映射。

③ 映射共享区域。如果a. 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中的程序计数器， 使之指向代码区域的入口点。

下一次调度hello进程时，它将从这个入口点开始执行。Linux将根据需要换入代码和数据页面。

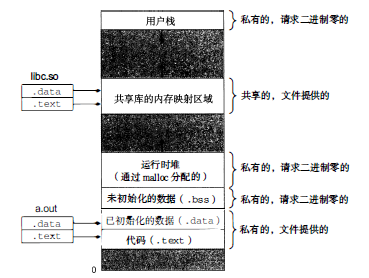


图7.7 加载器是如何映射用户地址空间的区域的

## 7.8 缺页故障与缺页中断处理

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。

之后，内核从磁盘把本来要读取的那个虚拟页，复制到内存中牺牲页的那个位置，更新它的页表条目，随后返回。当异常处理程序返回时，会重新启动导致缺页的指令，该指令会把导致缺页的虚拟地址重发送到地址翻译硬件。于是，地址翻译硬件可以正常处理现在的页命中了。

## 7.9动态存储分配管理

动态内存分配器维护着一个进程的虚拟内存区 域，称为堆(heap)系统之间细节不同，但是不失通用性，假设堆是一个请求二进制零的区域，它紧接在未初始化的数据区域后开始，并向上生长（向更高的地址）。对于每个进程，内核维护着一个变量brk(读做"break"),它指向堆的顶部。

分配器将堆视为一组不同大小的块(block)的集合来维护。每个块就是一个连续的虚拟内存片(chunk),要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可用来分配。空闲块 保持空闲，直到它显式地被应用所分配。一个已分配 的块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

分配器有两种基本风格。 两种风格都要求应用显式地分配块。 它们的不同之处在于由哪个实体来负责释放已分配的块。

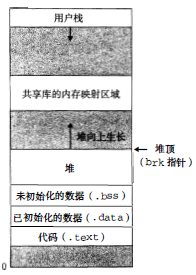


图7.9.1 堆

显式分配器(explicitallocator), 要求应用显式地释放任何已分配的块。例如，C标准库提供一种叫做ma耳oc程序包的显式分配器。C程序通过调用malloc函数来 分配一个块，并通过调用free函数来释放一个块。C++中的new和delete操作符与C中的malloc和free相当。

隐式分配器(implicitallocator) , 另一方面，要求分配器检测一个已分配块何时不再被程序所使用，那么就释放这个块。隐式分配器也叫做垃圾收集器(garbagecollec­tor), 而自动释放未使用的巳分配的块的过程叫做垃圾收集(garbagecollection)。例如，诸如Lisp、ML以及Java之类的高级语言就依赖垃圾收集来释放已分配的块。

⑴ 隐式空闲链表---在此介绍带边界标签的隐式空闲链表分配器原理

假设想要释放的块为当前块。那么合并下一个空闲块很简单而且高效。当前块的头部指向下一个块的头部，可以检查这个指针以判断下一个块是否是空闲的。如果是，就将它的大小简单地加到当前块头部的大小上，这两个块在常数时间内被合并。

给定一个带头的隐式空闲链表，唯一的选择将是搜索整个链表。记住前面块的位置，直到我们打到当前块。使用隐式空闲链表，这意味着每次调用free需要的时间都于堆的大小呈线性关系。即使使用更复杂精细的空闲链表组织，搜索时间也不会是常数。

Knuth提出一种聪明而通用的技术，叫做边界标记，允许在常数时间内进行对前面块的合并，这种思想，是在每个块的结尾处添加一个脚部，其中脚部就是头部的一个副本。如下所示：

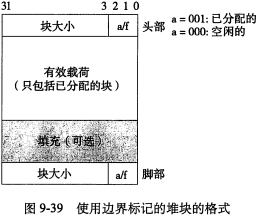


图7.9.2 带边界标记的堆块的格式

如果每个块包括这样一个脚部，那么分配器就可以通过检查它的脚部，判断前面一个块的起始位置和状态，这个脚部总是在距当期块开始位置一个字的距离。那么，分配器释放当前块时存在四种可能情况：

(1)    前面的块和后面的块都是已分配的

(2)    前面的块是已分配的，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)    前面的和后面的块都是空闲的。

下图，展示了这四种情况合并的过程：

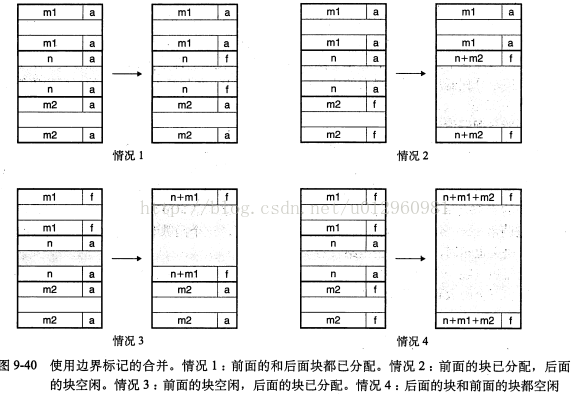


图7.9.3 带边界标记的合并

⑵ 显示空闲链表

显式空闲链表是一种更好的方式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。例如，堆可以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。

使用双向链表而不是隐式空闲链表，使首次适配的分配时间从块总数的线性时间减少到了空闲块数量的线性时间。不过，释放一个块的时间可以是线性的，也可能是常数，这取决于我们所选择的空闲链表中块的排序策略。

一种方法是用后进先出(LIFO)的顺序维护链表，将新释放的块位置放在链表的开始出。使用LIFO的顺序和首次适配的放置策略，分配器会最先检查最近使用过的块。在这种情况下，释放一个块可以在常数时间内完成。如果使用了边界标记，那么合并也可以在常数时间内完成。

另一种方法是按照地址顺序来维护链表，其中链表中每个块的地址都小于它后继的地址。在这种情况下，释放一个块需要线性时间的搜索来定位合适的前驱。平衡点在于，按照地址排序的首次适配比LIFO排序的首次适配有更高的存储器利用率，接近最佳适配的利用率。

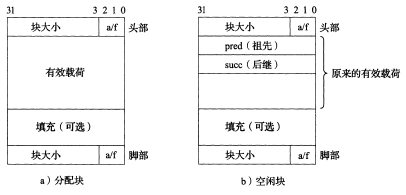


图7.9.4 使用双向空闲链表的堆块的格式

## 7.10本章小结

在本章中整理了有关内存管理的知识，讲述了在hello运行的64位系统中内存管理方法，虚拟内存和物理内存之间的关系，了解了intel环境下的段式管理和页式管理，了解了fork和exceve的内存映射，知道了缺页故障和缺页中断管理机制，了解了如何根据缓存或页表寻找物理內存。

# 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，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，称为Unix I/O。

## 8.2 简述Unix IO接口及其函数

**Unix I/O接口统一操作：**

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：

⑴ 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。

⑵ 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）。头文件<unistd.h>定义了常量STDIN\_FILENO、STDOUT\_FILENO和STDERR\_FILENO，它们可以用来代替显式的描述符值。

⑶ 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。

⑷ 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end-of-file(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。

⑸ 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：**

⑴ 在Unix I/O接口中，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，函数声明如下：

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

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。

⑵ 进程通过调用close函数关闭一个打开的文件。函数声明如下：

**int** close(**int** fd);

fd是需要关闭的文件的描述符，close返回操作结果。成功返回0错误返回EOF

⑷ 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。函数声明如下：

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

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

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。

⑸ 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。函数声明如下：

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

## 8.3 printf的实现分析

printf函数是在stdio.h头文件中声明的，具体代码实现如下：

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. }

它的参数包括一个字串fmt，和…。…表示参数个数不能确定，也就是格式化标准输出，我们也不能确定到底有几个格式串。

首先arg获得第二个不定长参数，即输出的时候格式化串对应的值。

va\_list是一个数据类型，其声明如下：

**typedef** **char** \***va\_list**

赋值语句右侧的，是一个地址偏移定位，定位到了从fmt开始之后的第一个char\*变量，也就是第二个参数了。

接下来是调用vsprintf函数，并把返回值赋给整型变量i。后来又调用write函数从内存位置buf处复制i个字节到标准输出。想必这个i就是printf需要的输出字符总数，那么vsprintf就是对参数解析了。vsprintf函数代码如下：

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++) { //p初始化为buf，下面即将把fmt解析并把结果存入buf中
8. /\* 寻找格式化字串 \*/
9. **if** (\*fmt != '%') {
10. \*p++ = \*fmt;
11. **continue**;
12. }
14. fmt++; //此时，fmt指向的是格式化字串的内容了
16. **switch** (\*fmt) {
17. /\*这是格式化字串为%x的情况\*/
18. **case** 'x':
19. itoa(tmp, \*((**int**\*)p\_next\_arg)); //把fmt对应的那个参数字串转换格式，放到tmp串中
20. strcpy(p, tmp); //tmp串存到p中，也就是buf中
21. p\_next\_arg += 4; //定位到下一个参数
22. p += strlen(tmp); //buf中的指针也要往下走
23. **break**;
24. /\* Case %s \*/
25. **case** 's':
26. **break**;
27. **default**:
28. **break**;
29. }
30. }
32. **return** (p - buf);
33. }

则知道vsprintf程序按照格式fmt结合参数args生成格式化之后的字符串，并返回字串的长度。

在printf中调用系统函数write(buf,i)将长度为i的buf输出。write函数如下：

1. mov eax, \_NR\_write
3. mov ebx, [esp + 4]
5. mov ecx, [esp + 8]
7. **int** INT\_VECTOR\_SYS\_CALL

在write函数中，将栈中参数放入寄存器，ecx是字符个数，ebx存放第一个字符地址，int INT\_VECTOR\_SYS\_CALLA代表通过系统调用syscall，查看syscall的实现：

1. sys\_call:
2. call save
3. push dword [p\_proc\_ready]
4. sti
5. push ecx
6. push ebx
7. call [sys\_call\_table + eax \* 4]
8. add esp, 4 \* 3
9. mov [esi + EAXREG - P\_STACKBASE], eax
10. cli
11. ret

syscall将字符串中的字节“Hello 学号 姓名”从寄存器中通过总线复制到显卡的显存中，显存中存储的是字符的ASCII码。

字符显示驱动子程序将通过ASCII码在字模库中找到点阵信息将点阵信息存储到vram中。

显示芯片会按照一定的刷新频率逐行读取vram，并通过信号线向液晶显示器传输每一个点（RGB分量）。

于是字符串“Hello 学号 姓名”就显示在了屏幕上。

## 8.4 getchar的实现分析

getchar的函数声明在stdio.h头文件中，具体代码实现如下：

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

bb是缓冲区的开始，int变量n初始化为0，只有在n为0的情况下，从缓冲区中读BUFSIZ个字节，就是缓冲区中的内容全部读入。这时候，n的值被修改为，成功读入的字节数，正常情况下n就是一个正数值。返回时，如果n大于0，那么就返回缓冲区的第一个字符。否则，就是没有从缓冲区读到字节，返回EOF。

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

本章了解了在Linux环境下I/O相关设备的操作，I/O设备的实现方式，并且简要的介绍了日常常用的printf函数和getchar函数。

# 结论

本次大作业，围绕着hello程序的一次运行展开了理解和讨论，在完成的过程中，也是对自己在这一学期所学知识的一个小的总结。

hello在从诞生到消失，已经经历许多，现在可以查看历史年表一一回忆：

1. 编写：在各式各样，你用着顺手或者不顺手的IDE被用高级程序语言编写出来。

2. 预处理：在gcc编译器中生成.i文件，对代码中的宏定义进行了处理。

3. 编译：在gcc编译器中生成.s文件，将高级程序语言转变为汇编语言。

4. 链接：在gcc编译器中与可重定位目标文件和其它必要的可重定位目标文件一切链接，生成可执行目标文件。

5. 运行：在Shell中输入参数运行可执行文hello，fork为其创建子进程，exceve调用加载器，加映射虚拟内存，进入程序入口后程序开始载入物理内存，然后进入 main函数。

6. 执行：CPU分配时间片，hello执行逻辑控制流。

7. 访问内存：经由MMU将虚拟内存映射为物理内存。

8. 申请动态内存：执行printf和getchar等函数时，申请动态内存。

9. 信号处理：当程序执行时键入了shell信号，会出现不同的结果。

10.终止：父进程回收子进程，内核删除为这个进程创建的所有数据结构。

《深入理解计算机系统》一书的学习可以说才刚刚开始，身为一个未来的程序猿，深入的了解一个程序运行的原理是必须的，只有了解了这些过程才能更好地服务之后的程序的编写。

一门课程即将结束，尽管他才第二年开设，可能还有着许多不完善的地方，但是，在今后仍应该继续地，更深入地了解这本书中的知识。

# 附件

|  |  |
| --- | --- |
| 文件名字 | 文件作用 |
| hello.i | 预处理产生文件 |
| hello.s | 编译产生文件 |
| hello.o | 汇编产生文件 |
| hello.out | 链接产生文件 |
| hello | 链接产生可执行文件 |
| hello.elf | hello.o的elf文件 |
| hello-d.txt | hello反汇编生成文件 |
| hello-a.txt | hello的elf文件 |
| hello-d2.txt | hello.o反汇编生成的文件 |

# 参考文献

[1] 《深入理解计算机系统》第三版，兰德尔 E.布莱恩特，大卫 R.奥哈拉伦

[2] printf函数剖析 <https://www.cnblogs.com/pianist/p/3315801.html>

[3] 页表 <https://zh.wikipedia.org/wiki/%E5%88%86%E9%A0%81%E8%A1%A8>

[4] 逻辑地址，线性地址 <https://www.cnblogs.com/bhlsheji/p/4868964.html>

[5] elf <https://blog.csdn.net/ylcangel/article/details/37997913>

[6] 内核分析 <https://blog.csdn.net/ylcangel/article/details/37997913>

[7] 汇编指令 <https://blog.csdn.net/baishuiniyaonulia/article/details/78504758>