

**计算机系统**

**大作业**

题 目 程序人生-Hello’s P2P

专 业 软件工程

学　　 号 1183710113

班　　 级 1837101

学 生 许健

指 导 教 师 史先俊

**计算机科学与技术学院**

**2019年12月**

**摘 要**

对于每个程序员来说，Hello World是一个开始，本论文目的在于利用gcc、edb等工具，结合CSAPP教材，研究hello程序在Linux系统下的整个生命周期，从而达到融会贯通所学知识的效果。

**关键词：**CSAPP；HIT；大作业；hello

**（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 - 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 - 16 -](#_Toc532238410)

[第4章 汇编 - 17 -](#_Toc532238411)

[4.1 汇编的概念与作用 - 17 -](#_Toc532238412)

[4.2 在Ubuntu下汇编的命令 - 17 -](#_Toc532238413)

[4.3 可重定位目标elf格式 - 17 -](#_Toc532238414)

[4.4 Hello.o的结果解析 - 20 -](#_Toc532238415)

[4.5 本章小结 - 22 -](#_Toc532238416)

[第5章 链接 - 23 -](#_Toc532238417)

[5.1 链接的概念与作用 - 23 -](#_Toc532238418)

[5.2 在Ubuntu下链接的命令 - 23 -](#_Toc532238419)

[5.3 可执行目标文件hello的格式 - 23 -](#_Toc532238420)

[5.4 hello的虚拟地址空间 - 24 -](#_Toc532238421)

[5.5 链接的重定位过程分析 - 25 -](#_Toc532238422)

[5.6 hello的执行流程 - 26 -](#_Toc532238423)

[5.7 Hello的动态链接分析 - 27 -](#_Toc532238424)

[5.8 本章小结 - 29 -](#_Toc532238425)

[第6章 hello进程管理 - 30 -](#_Toc532238426)

[6.1 进程的概念与作用 - 30 -](#_Toc532238427)

[6.2 简述壳Shell-bash的作用与处理流程 - 30 -](#_Toc532238428)

[6.3 Hello的fork进程创建过程 - 30 -](#_Toc532238429)

[6.4 Hello的execve过程 - 31 -](#_Toc532238430)

[6.5 Hello的进程执行 - 31 -](#_Toc532238431)

[6.6 hello的异常与信号处理 - 32 -](#_Toc532238432)

[6.7本章小结 - 36 -](#_Toc532238433)

[第7章 hello的存储管理 - 37 -](#_Toc532238434)

[7.1 hello的存储器地址空间 - 37 -](#_Toc532238435)

[7.2 Intel逻辑地址到线性地址的变换-段式管理 - 37 -](#_Toc532238436)

[7.3 Hello的线性地址到物理地址的变换-页式管理 - 39 -](#_Toc532238437)

[7.4 TLB与四级页表支持下的VA到PA的变换 - 41 -](#_Toc532238438)

[7.5 三级Cache支持下的物理内存访问 - 43 -](#_Toc532238439)

[7.6 hello进程fork时的内存映射 - 44 -](#_Toc532238440)

[7.7 hello进程execve时的内存映射 - 44 -](#_Toc532238441)

[7.8 缺页故障与缺页中断处理 - 45 -](#_Toc532238442)

[7.9动态存储分配管理 - 45 -](#_Toc532238443)

[7.10本章小结 - 48 -](#_Toc532238444)

[第8章 hello的IO管理 - 49 -](#_Toc532238445)

[8.1 Linux的IO设备管理方法 - 49 -](#_Toc532238446)

[8.2 简述Unix IO接口及其函数 - 49 -](#_Toc532238447)

[8.3 printf的实现分析 - 50 -](#_Toc532238448)

[8.4 getchar的实现分析 - 51 -](#_Toc532238449)

[8.5本章小结 - 52 -](#_Toc532238450)

[结论 - 52 -](#_Toc532238451)

[附件 - 54 -](#_Toc532238452)

[参考文献 - 55 -](#_Toc532238453)

# 第1章 概述

## 1.1 Hello简介

P2P：程序员用键盘输入hello.c文件，一个hello的C语言文件诞生，然后经过预处理器、汇编器、编译器、链接器的一系列处理，hello可执行文件诞生了，在shell中键入启动命令后，shell为其fork，产生子进程，于是hello便从Program变成了Process。

020：shell调用execve函数在新的子进程中加载并运行hello，在hello运行的过程中，还需要CPU为hello分配内存、时间片，使得hello看似独享CPU资源。系统的进程管理帮助hello切换上下文、shell的信号处理程序使得hello在运行过程中可以处理各种信号，当程序员主动地按下Ctrl+Z或者hello运行到return 0时，hello所在进程将被杀死，shell会回收它的僵死进程，内核删除相关数据结构。

## 1.2 环境与工具

硬件环境：Inter(R) Core(TM) i5-7300HQ CPU；2.5GHz；8G RAM；128G SSD+1T HDD

软件环境：Windows 10 64位；Vmware 15；Ubuntu 19.04 64位

开发与调试工具：gcc；edb； readelf；objdump；gedit；hexedit；

## 1.3 中间结果

hello.c——原文件

hello.i——预处理之后文本文件

hello.s——编译之后的汇编文件

hello.o——汇编之后的可重定位目标执行

hello——链接之后的可执行目标文件

hello.elf——hello.o的elf格式，用来看hello.o的各节信息

hello.ob——hello.o的反汇编文件，用来看汇编器翻译后的汇编代码

hello1.ob——hello的反汇编文件，用来看链接器链接后的汇编代码

## 1.4 本章小结

本章主要简单介绍了hello的P2P，020过程，列出了本次实验信息：环境、中间结果。

**（第1章0.5分）**

# 第2章 预处理

## 2.1 预处理的概念与作用

预处理的概念：预处理器(cpp)根据以字符#开头的命令，修改原始的C程序。将所引用的所有库展开，处理所有的条件编译，并执行所有的宏定义，得到另一个通常是以.i作为文件扩展名的C程序。

预处理的作用：

1. 将c程序中所有#include声明的头文件复制到新的程序中。比如hello.c中第6~8行的#include <stdio.h>、#include <unistd.h>、#include <stdlib.h>命令告诉预处理器读取系统头文件stdio.h、unistd.h、stdlib.h的内容，并把它直接插人程序文本中；
2. 条件编译。根据条件#if决定是否处理之后的代码；
3. 执行宏替换。用实际值替换用#define定义的字符串。

## 2.2在Ubuntu下预处理的命令

命令：cpp hello.c > hello.i

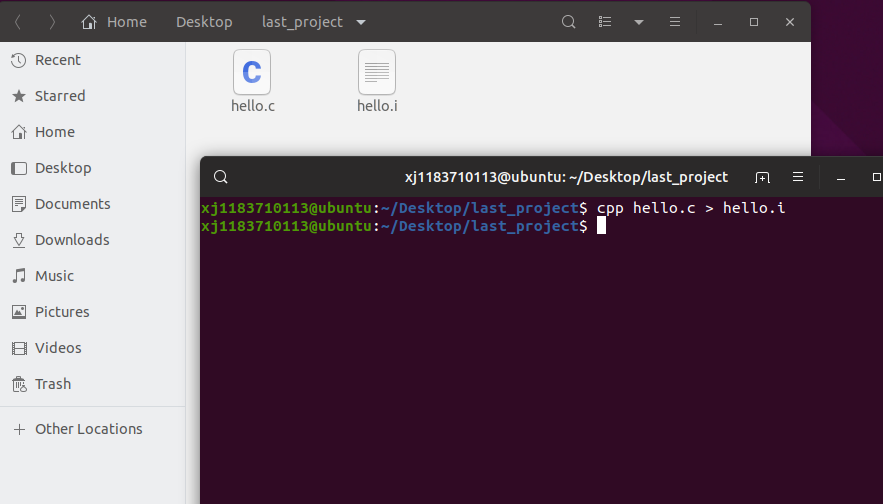


图2.1 对hello.c进行预处理

## 2.3 Hello的预处理结果解析

使用Text Editor打开hello.i，发现原来的helloc.c已经被拓展成了3042行，前面的内容是hello.c的三个#include指令包含的头文件的代码，先寻找main函数，main函数从第3029行开始，如下图。



图2.2 hello.i中的main函数

再看之前的头文件的处理，以第一条#include指令为例，cpp到默认的环境变量下搜索stdio.h头文件，打开/usr/include/stdio.h，发现其中仍有#include指令，于是再去搜索包含的头文件，直到最后的文件中没有#include指令，并把所有文件中的所有#define和#ifdef指令进行处理，执行宏替换和通过条件确定是否处理定义的指令。如图是对stdio.h包含文件的展开。

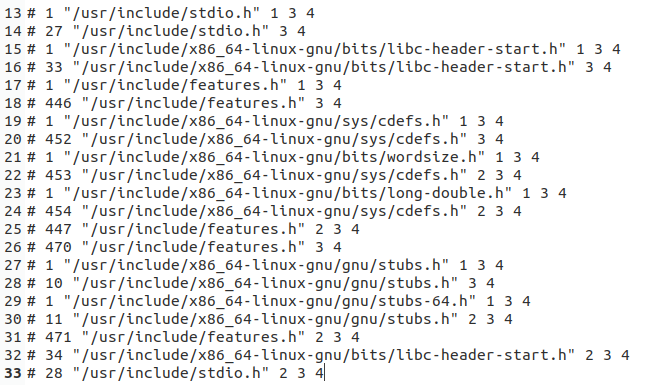


图2.3 #include<stdio.h>包含文件展开

## 2.4 本章小结

本章主要介绍了预处理的概念及作用，并结合hello.c处理后的hello.i对处理过程进行分析。

**（第2章0.5分）**

# 第3章 编译

## 3.1 编译的概念与作用

编译的概念：编译器将文本文件hello.i翻译成另一个文本文件hello.s，它包含一个汇编语言程序。

编译的作用：将字符串转化成内部的表示结构，然后得到一系列记号，生成语法树，最后将语法树转化为目标代码。

## 3.2 在Ubuntu下编译的命令

命令：gcc -S hello.i -o hello.s

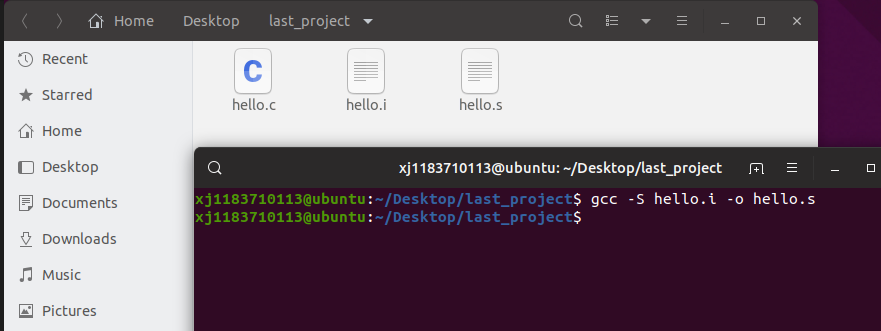


图3.1 hello.i编译生成hello.s

## 3.3 Hello的编译结果解析

* + 1. **数据**

1. 字符串

程序中用到的字符串有：“用法: Hello 学号 姓名 秒数！\n”和“Hello %s %s\n”。编译器一般将字符串存放在.rodata节，这两个个字符串在hello.s中的存储如下图，可以看到第一个字符串中的汉字被编码成UTF-8格式，一个汉字占三个字节，每个字节用\分隔。第二个字符串中的两个%s为用户在终端运行hello时输入的两个参数。

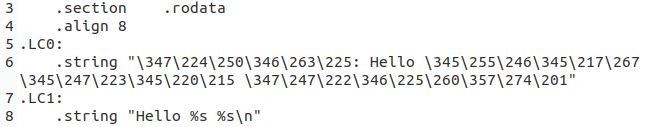


图3.2 hello.s存储的两个字符串

1. 整数

hello.c中的整型变量有argc和i。

其中argc是从终端传入的参数个数，也是main函数的第一个参数，所以由寄存器%edi进行保存。由图3.3的21行可知，argc又被存入了栈中-20(%rbp)的位置。



图3.3 argc被保存在栈中

i则是局部变量，用来控制循环次数的计数器，编译器会将局部变量保存在寄存器或者栈中，由图3.4的30行看出hello.s将i存储在栈中-4(%rbp)的位置。



图3.4 .L2中的i的位置

1. 数组

hello.c中数组是main函数的第二个参数，char \*argv[]，是字符指针数组，由于是第二个参数因而被保存在寄存器%rsi中，由图3.5的第22行可知它随后又被保存在了栈中-32(%rbp)的位置。



图3.5 argv被保存在栈中

在访问argv[]所指向的内容时，每次先获得数组的起始地址，如图3.6的第33、36、43行，然后通过加8\*i来访问之后的字符指针，如图3.6中的第34、37、44，原因是每个字符指针所占的空间大小围为8个字节。然后通过获得的字符指针寻找字符串，如图3.6中的第35、38、45行。

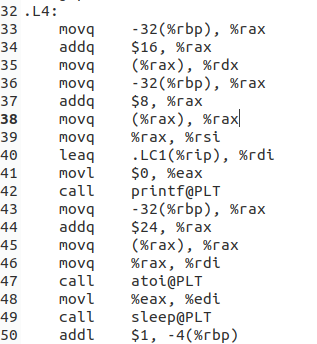


图3.6 访问argv数组元素

* + 1. **赋值**

hello.c中的赋值操作只有i=0这一条，这条语句在汇编中用mov指令实现，由于int占4个字节，所以以‘l’作为后缀。如图3.7中的第30行。



图3.7 给i赋值

* + 1. **类型转换**

程序中涉及的类型转换只有一处，如图3.8所示的第19行，使用atoi函数将命令行的第三个字符串参数转换成了整型。



图3.8 hello.c的main函数

* + 1. **算术操作**

汇编语言中有如下几种算术操作：

|  |  |  |
| --- | --- | --- |
| 指令 | 行为 | 描述 |
| inc D | D=D+1 | 加1 |
| dec D | D=D-1 | 减1 |
| neg D | D=-D | 取反 |
| add S,G | D=D+S | D加S |
| sub S,D | D=D-S | D减S |
| imul S,D | D=D\*S | D乘S |
| imulq S | R[%rdx]:R[%rax]=S\*R[%rax] | 有符号乘法 |
| mulq S | R[%rdx]:R[%rax]=S\*R[%rax] | 无符号乘法 |
| idivq S | R[%rdx]=R[%rdx]:R[%rax] mod S R[%rax]=R[%rdx]:R[%rax] div S | 有符号除法 |
| divq S | R[%rdx]=R[%rdx]:R[%rax] mod S R[%rax]=R[%rdx]:R[%rax] div S | 无符号触发 |
| leaq S,D | D = &S | 加载有效地址 |

helo.c中的算术操作只有一处，循环变量i的自增运算，在hello.s中处理成如图3.9的形式。



图3.9 i的自增运算

* + 1. **关系操作**

C语言中的关系操作有==、!=、>、<、>=、<=，这些操作在汇编语言中主要依赖于cmp和test指令实现，cmp指令根据两个操作数之差来设置条件码。cmp指令与SUB指令的行为是一样，而test指令的行为与and指令一样，除了它们只设置条件码而不改变目的寄存器的值。

在hello.c中有两处用到了关系操作，分别是图3.8中的第13行的argc！=4和第17行的i<8。这两句在hello.s中被分别处理为图3.10和图3.11的形式。cmp之后设置条件码，为之后的je和jle提供判断依据。

图3.10 argc!=4在hello.s中的体现 图3.11 i<8在hello.s中的体现

* + 1. **数组/指针/结构操作**

在hello.c中通过下标访问argv数组，在hello.s中访问argv[1]的操作如图3.12所示，第36行是取argv首地址，第37行是通过首地址加8字节找到argv[1]的地址，第38行是通过argv[1]中的内容找到对应的字符串，保存在寄存器%rax中。对argv数组其他元素所指的字符串也同理。



图3.12 访问argv[1]所指的字符串

* + 1. **控制转移**

程序涉及到的控制转移有两处。

第一处是判断argc是否与4相等，在hello.s中如图3.13所示，第23行cmpl比较argc和4设置条件码之后，第24行通过判断条件码ZF位是否为零决定是否跳转到.L2，如果为0，说明argc等于4，代码跳转到.L2继续执行，如果不为0，则执行图中第25行的指令。

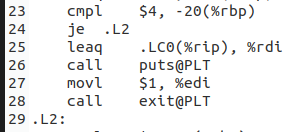


图3.13 对if语句的处理

第二处是判断循环变量i是否满足循环条件i<8。如图3.14所示，在第30行循环变量i被初始化为0，第30行无条件跳转到.L3，进入循环判断，在52行cmpl比较i和7之后设置条件码，然后第53行判断是否满足i<=7的要求，如果满足，跳转到.L4执行循环体，如果不满足，则退出循环，执行第54行的指令。

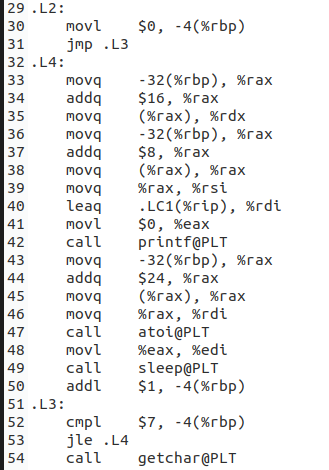


图3.14 对for循环的处理

* + 1. **函数操作**

函数是一种过程，提供了一种封装代码的方式。P调用Q时有如下行为：

传递控制：开始执行Q的时候，PC必须设置为Q的代码的起始地址，而在返回时要把PC设置为P中调用Q之后一条语句的地址。

传递数据：P能够向Q传递任意个数的参数，Q能够向P返回0或1个值。P向Q传递参数时，64为程序参数存储顺序如下表：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 第一个 | 第二个 | 第三个 | 第四个 | 第五个 | 第六个 | 第七个及之后 |
| %rdi | %rsi | %rdx | %rcx | %r8 | %r9 | 栈中 |

分配和释放内存：开始Q时为Q分配必要的空间，而在Q返回前需要把已分配给Q的空间释放。

程序中涉及的函数有6个。

1. main函数

main函数被保存在.text节，程序运行时，由系统启动调用main函数，mian函数的两个参数分别是由命令行传入的argc和argv[]，分别被保存在%rdi和%rsi中。

1. printf函数

hello.c有两处调用了printf函数，第一个printf函数由于只有一个参数，所以被编译器优化为puts函数，如图3.15。参数被保存在寄存器%rdi中



图3.15 第一个printf函数

第二个printf函数有三个参数，从内存中取出参数之后，如图3.16红线部分，第三、二、一个参数分别被保存在寄存器%rdx、%rsi、%rdi中。

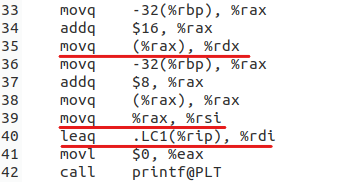


图3.16 第二个printf函数

1. exit函数

hello.c中在用户输入的不是四个参数时会调用exit函数结束程序，在hello.s中如图3.17所示，把参数1用mov指令传给%edi，然后调用exit函数。



图3.17 exit函数

1. atoi函数

hello.c中通过atoi函数把用户输入的第四个参数从字符串转化成整型，对应的汇编代码如图3.18所示，第45行是取得用户输入的第四个参数，第46行把这个参数作为函数atoi的参数保存在%rdi中，然后调用atoi函数。

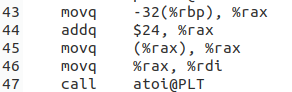


图3.18 atoi函数

1. sleep函数

sleep函数的参数是atoi函数的返回值，返回值被保存在%eax中，所以图3.19中第48行把%eax中的值传送给%rdi作为sleep函数的参数。



图3.19 sleep函数

1. getchar函数

由于getchar函数没有参数，所以在退出循环之后直接call getchar@PLT即可。

## 3.4 本章小结

本章主要介绍了编译的概念及作用，并且结合hello.i编译生成的hello.s汇编代码详细阐述了编译器是如何处理C语言的各种数据类型、各种运算操作及函数调用。经过本次处理，最初的C语言版本的hello.c已经被转行成了更加低级的汇编程序。

**（第3章2分）**

# 第4章 汇编

## 4.1 汇编的概念与作用

汇编的概念：汇编器as将.s文件翻译成机器指令，把这些指令打包成一中叫做可重定位目标程序格式，并将结果保存在目标文件中。

汇编的作用：将编译器产生的汇编语言进一步翻译为计算机可以理解的机器语言，生成.o文件。

## 4.2 在Ubuntu下汇编的命令

命令：as hello.s -o hello.o

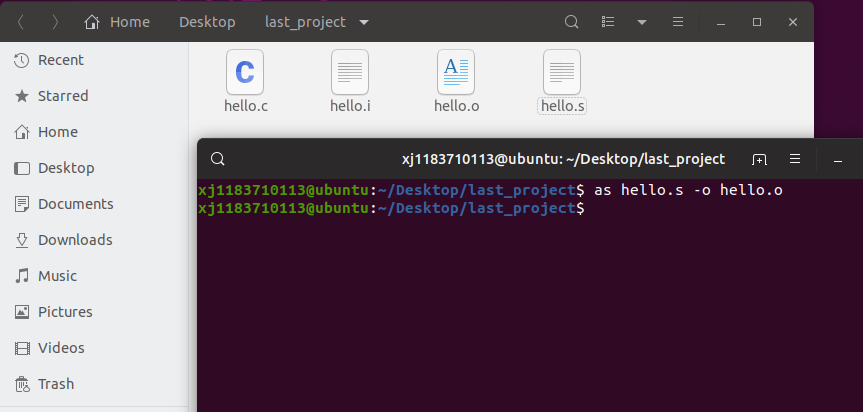


图4.1 hello.s汇编生成hello.o

## 4.3 可重定位目标elf格式

使用readelf -a -W hello.o > hello.elf命令获得hello.o文件elf格式，并将结果输出到名为hello.elf的文件中。该文件由以下几个部分组成。

1. ELF头

ELF头有一个16字节的Magic序列开始，这个序列描述了生成该文件的系统的字大小和字节顺序。ELF头剩下的部分包含帮助连接器语法分析和解释目标文件的信息。其中包括ELF头的大小、目标文件类型、机器类型、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。

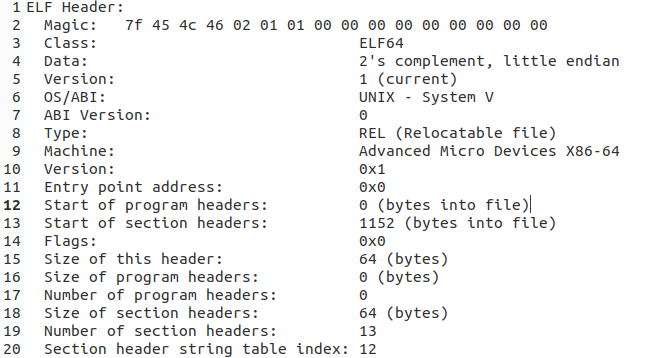


图4.2 ELF头

1. 节头部表

节头部表包含了文件中出现的各个节的含义，包括节的地址、偏移量、大小等信息。如.text节，地址是从0x0开始，偏移量是0x40，大小是0x8e。

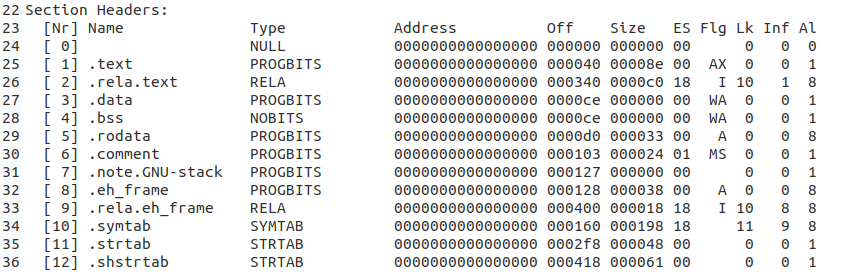


图4.3 节头部表

1. .rela.text节

存放着代码的重定位条目。当链接器吧这个目标文件和其他文件组合时，会结合这个节，修改.text节中相应位置的信息。如图4.4中的重定位信息依次对应.L0、puts函数、exit函数、.L1、printf函数、atoi函数、sleep函数、getchar函数。

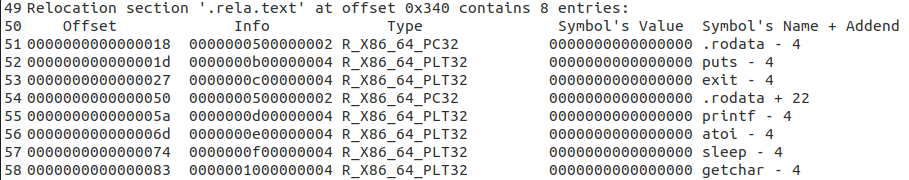


图4.4 .rela.text节

.rela.text包含的信息有如下几部分：

|  |  |
| --- | --- |
| Offset | 需要进行重定向的代码在.text或.data节中的偏移位置，8个字节。 |
| Info | 包括symbol和type两部分，其中symbol占前4个字节，type占后4个字节，symbol代表重定位到的目标在.symtab中的偏移量，type代表重定位的类型。 |
| Addend | 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。 |
| Type | 重定位到的目标的类型。 |
| Name | 重定向到的目标的名称。 |

重定位一个使用32位PC相对地址的引用，计算重定位目标地址的方法如下：

先计算指向原位置src的指针：refptr=s+r.offset

再计算src的运行时地址：refaddr=ADDR(s)+r.offset

将src处设置为运行值：\*refptr=(unsigned)(ADDR(r.symbol)+r.addend-refaddr)

在hexedit中查看hello.o，图4.5是.L1的重定位条目r，可以得到以下的信息：r.offset=0x18，r.symbol=.rodata，r.type=R\_X86\_64\_PC32，r.addend=-4。

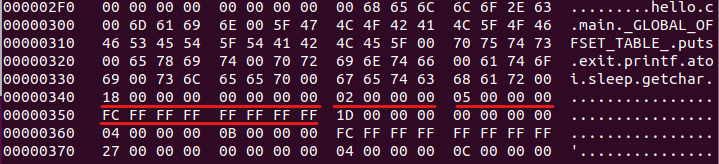


图4.5 hexedit查看hello.o

用上述的计算方法计算r的重定位之后的运行值信息，再把\*refptr写到src处，完成.L1的重定位。

1. .rela.eh\_frame节

.eh\_frame节的重定位信息。

1. .symtab节

符号表，用来存放程序中的定义和引用函数的全局变量的信息。重定位需要引用的符号都在其中声明。name是字符串表中的字节偏移，指向符号的以null结尾的字符串名字，value是符号的地址，对于可重定位的模块来说，value是距定义目标的节的起始位置的偏移。对于可执行目标文件来说，该值是一个绝对运行时地址，size是目标的大小，type通常要么是数据，要么是函数，binding表示符号是本地的还是全局的。ABS代表不该被重定位的符号，UNDEF代表未定义的符号，也就是在本目标模块中引用，但是却在其他地方定义的符号，COMMON表示还未被分配位置的未初始化的数据目标。

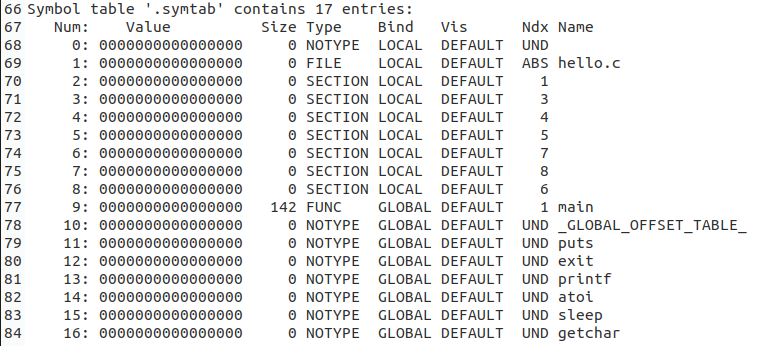


图4.6 符号表

## 4.4 Hello.o的结果解析

使用命令objdump -d -r hello.o > hello.ob将hello.o的反汇编代码输出到名为hello.ob的文件中。与hello.s对比如图4.7所示。

1.机器语言的构成

机器语言是计算机能直接理解的语言，完全由二进制数构成，为了阅读的方便显示成了16 进制。每两个16进制数构成一个字节编码，是机器语言中能解释一个运算符或操作数的最小单位。

机器语言由三种数据构成。一是操作码，它具体说明了操作的性质和功能，每一条指令都有一个相应的操作码，计算机通过识别该操作码来完成不同的操作；二是操作数的地址，CPU通过地址取得所需的操作数；三是操作结果的存储地址，把对操作数的处理所产生的结果保存在该地址中，以便再次使用。

2.机器语言与汇编语言的映射关系

由图4.7可以看出，机器语言反汇编得到的汇编代码与直接生成的hello.s的代码大致相同，只在以下几个部分中存在差别：

（1）分支转移：反汇编得到的代码中，跳转指令的操作数使用的不再是如hello.s中的.L2、.L3之类的代码段名称，而是具体的地址，因为这类名称只是在编写hello.s时为了便于编写所使用的一些符号，这些符号在汇编成机器语言之后不再存在，变成了语句地址，所以跳转指令的操作数也随之发生了变化。

（2）函数调用：在hello.s中，调用函数的形式是call指令加调用的函数名，如图4.7中左第28行，而在反汇编文件中是call加下一条指令的地址，如图4.7中右第20行。由于hello.c所调用的函数都是函数共享库中的函数，所以在调用这类函数时会产生重定位条目，这些条目在动态链接时会被修改为运行时的执行地址，而在汇编成的机器语言中，对于这些函数调用的相对地址全部被设置成0，所以call后面加的是下一条指令，而它的重定位信息则会被添加到.rela.text节，等链接后再确定。

（3）访问字符串常量：在hello.s中，使用.L0(%rip)的形式访问，而在反汇编文件中使用0x0(%rip)的方式访问。因为.rodata节中的地址也是没有确定的，在运行的时才会确定，所以需要重定位，同函数调用的处理方式一样，也是将其设置为0，并把重定位信息则会被添加到.rela.text节，等链接后再确定。

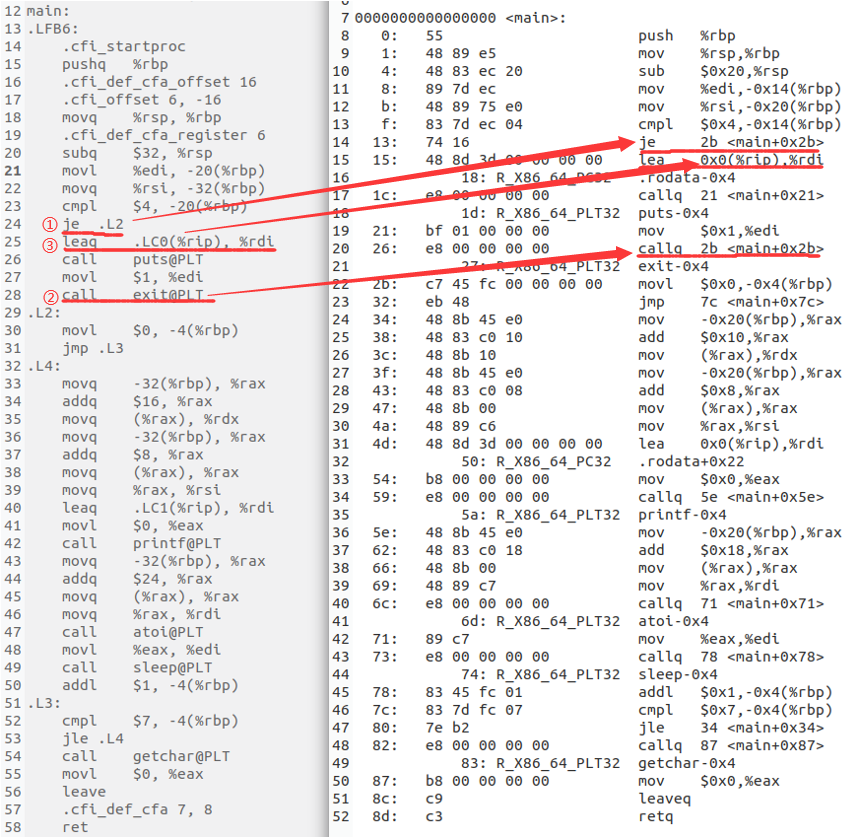


图4.7 hello.s（左）与hello.ob（右）的对比

## 4.5 本章小结

本章主要介绍了从hello.s 到hello.o的汇编过程，通过查看hello.o的elf格式和使用objdump得到反汇编代码与hello.s进行比较，了解了从汇编语言映射到机器语言汇编器需要实现的转换。

**（第4章1分）**

# 第5章 链接

## 5.1 链接的概念与作用

链接的概念：链接是将各种代码和数据片段收集并组合成为一个单一文件的过程，这个文件可被加载到内存并执行。在现代系统中，链接是由较做链接器的程序自动执行的。

链接的作用：链接器使得分离编译成为可能。我们不用将一个大型的应用程序组织为一个巨大的源文件，而是可以把它分解为更小、更好管理的模块，可以独立地修改和编译这些模块。当我们改变这些模块中的一个时，只需简单地重新编译它，并重新链接应用，而不必重新编译其他文件。

## 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

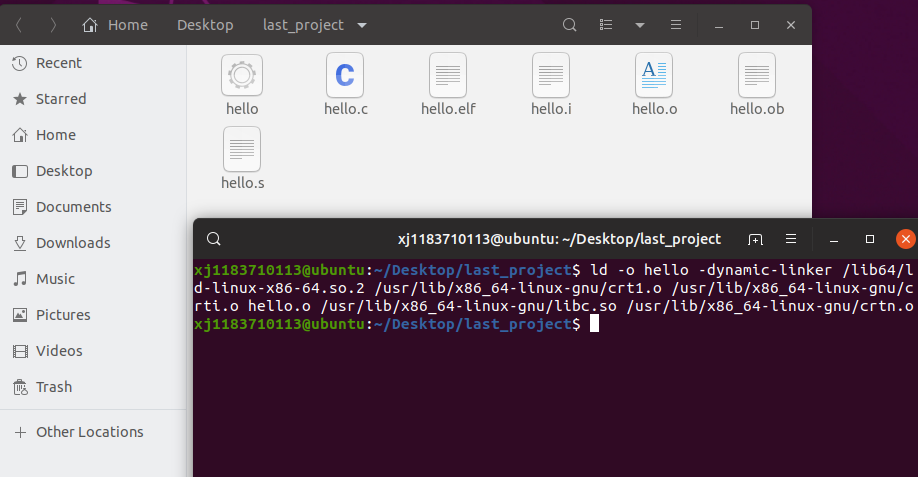


图5.1 hello.o与库文件链接生成hello

## 5.3 可执行目标文件hello的格式

使用readelf -a -W hello命令查看hello的elf格式，其中的节头表部分如图5.2所示。节头表记录了各个节的信息，Address是程序被载入到虚拟地址的起始地址，off是在程序中的偏移量，size是节的大小。

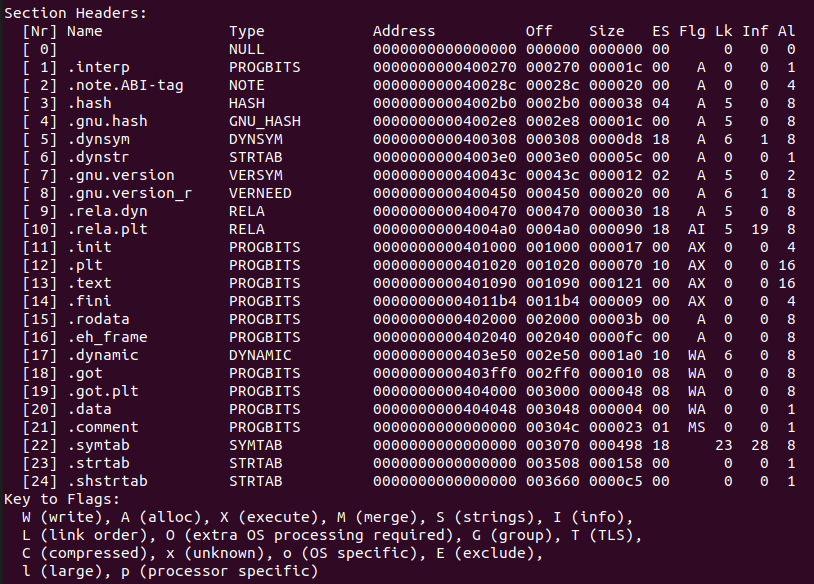


图5.2 hello的elf格式的Section Headers

## 5.4 hello的虚拟地址空间

用edb查看程序hello，发现程序在地址0x400000~0x401000中被载入，从0x400000开始到0x400fff结束，这之间每个节的排列同图5.2中Address中声明。在0x400fff之后存放的是.dynamic~.shstrtab节。

在Data Dump中查看地址0x400000开始的内容，可以看到开头是ELF头部分。如图5.3。

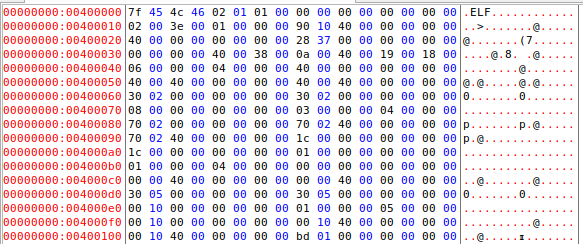


图5.3在Data Dump中查看地址0x400000

查看地址0x0x400200，发现是.interp节，保存着linux动态共享库的路径。如图5.4。

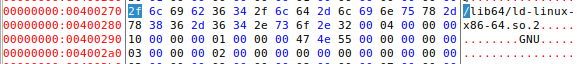


图5.4 在Data Dump中查看地址0x400270

查看地址0x0x400308，发现是.dynsym节，保存动态符号表。如图5.5。

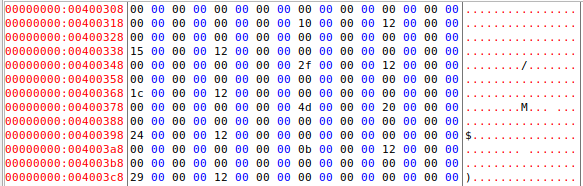


图5.5 在Data Dump中查看地址0x400308

查看地址0x0x402000，发现是.rodata节，其中保存着hello.c中的两个字符串。如图5.6。

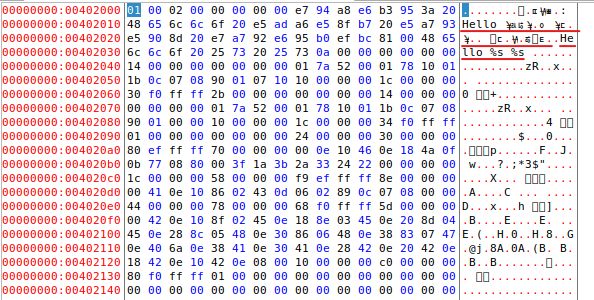


图5.6 在Data Dump中查看地址0x402000

在图5.2中的其他节也都能够通过对应的Address在Data Dump中找到，这里就不一一列举了。

## 5.5 链接的重定位过程分析

使用命令objdump -d -r hello > hello1.ob 生成hello的反汇编文件。

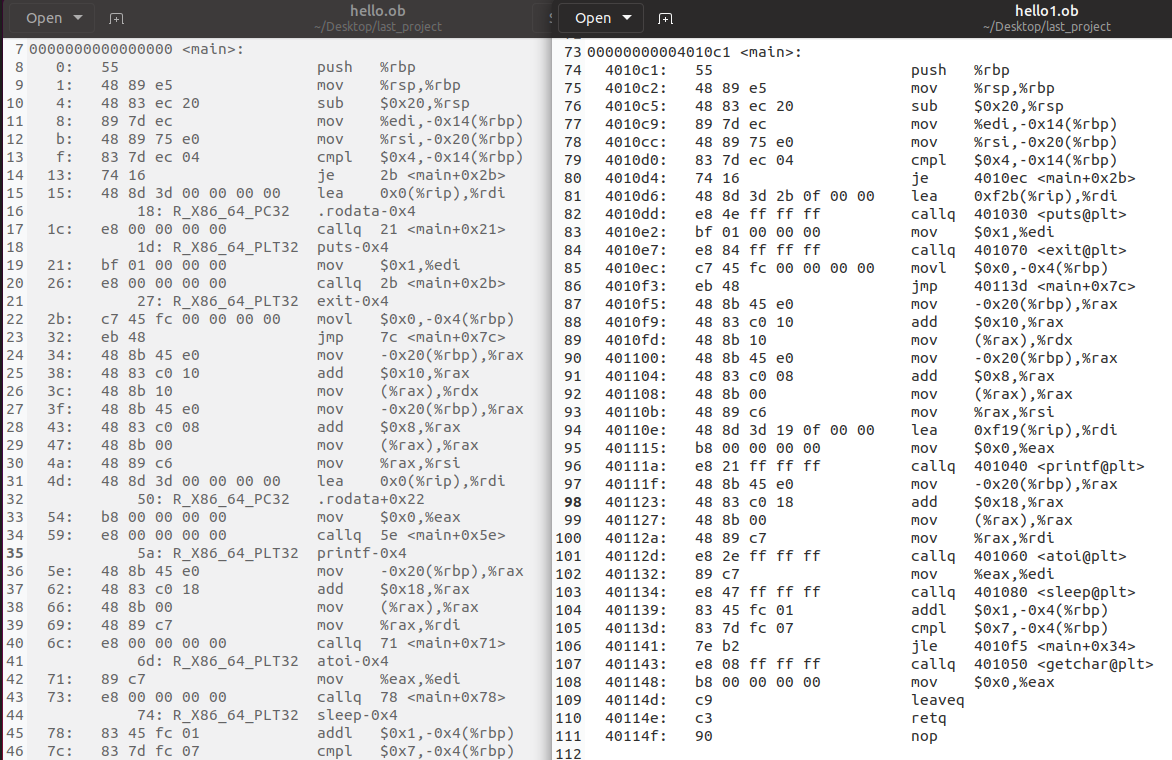


图5.7 hello.ob和hello1.ob的mian函数对比

对于hello.ob和hello1.ob来说，两者main函数的汇编指令完全相同，除了地址由相对偏移变成了可以由CPU直接寻址的绝对地址。链接器把hello.o中的偏移量加上程序在虚拟内存中的起始地址0x400000和.text节的偏移量就得到了hello1.ob中的地址。函数内的控制转移即jmp指令后的地址由偏移量变为了偏移量+函数的起始地址；call后的地址由链接器执行重定位后计算出实际地址。

函数调用：链接器解析重定条目时发现对外部函数调用的类型为R\_X86\_64\_PLT32的重定位，此时动态链接库中的函数已经加入到了PLT中，.text与.plt节相对距离已经确定，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中相应函数与下条指令的相对地址，指向对应函数。对于此类重定位链接器为其构造.plt与.got.plt。

.rodata引用：链接器解析重定条目时发现两个类型为R\_X86\_64\_PC32的对.rodata的重定位（printf中的两个字符串），.rodata与.text节之间的相对距离确定，因此链接器直接修改call之后的值为目标地址与下一条指令的地址之差，指向相应的字符串。这里以计算第一条字符串相对地址为例说明计算相对地址的算法：

refptr=s+r.offset=Pointerto0x4010dd

refaddr=ADDR(s)+r.offset=ADDR(main)+r.offset=0x4010c1+0x1c=0x4010dd

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

=ADDR(str1)+r.addend-refaddr

=0x402008-0x4010dd=(unsigned)0xf2b

在汇编中验证如图5.8。



图5.8 hello1.ob引用.rodata中的第一个字符串

除了main函数，hello1.ob比hello.ob多出了几个函数：printf、sleep、puts、getchar、atoi、exit。

除了.text节的区别外，hello1.ob比hello.ob多出了几个节：.init节、.plt节、.fini节。其中.init节是程序初始化需要执行的代码，.fini节是程序正常终止时需要执行的代码，.plt节是动态链接中的过程链接表。

## 5.6 hello的执行流程

子程序名：

ld-2.27.so!\_dl\_start

ld-2.27.so!\_dl\_init

hello!\_start

libc-2.27.so!\_\_libc\_start\_main

libc-2.27.so!\_\_cxa\_atexit

libc-2.27.so!\_\_libc\_csu\_init

libc-2.27.so!\_setjmp

hello!main

hello!puts@plt

hello!exit@plt

hello!printf@plt

hello!sleep@plt

hello!getchar@plt

ld-2.27.so!\_dl\_runtime\_resolve\_xsave

ld-2.27.so!\_dl\_fixup

ld-2.27.so!\_dl\_lookup\_symbol\_x

libc-2.27.so!exit

## 5.7 Hello的动态链接分析

对于动态共享链接库中PIC函数，编译器没有办法预测函数的运行时地址，所以需要添加重定位记录，等待动态链接器处理，为避免运行时修改调用模块的代码段，链接器采用延迟绑定的策略。动态链接器使用过程链接表PLT+全局偏移量表GOT实现函数的动态链接，GOT中存放函数目标地址，PLT使用GOT中地址跳转到目标函数。

在dl\_init调用之前，对于每一条PIC函数调用，调用的目标地址都实际指向PLT中的代码逻辑，GOT存放的是PLT中函数调用指令的下一条指令地址。由图5.2可知，.got.plt的起始地址是0x404000，其内容如图5.9。

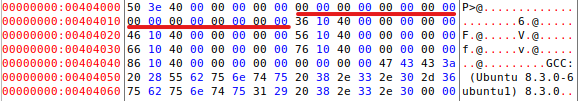


图5.9 执行dl\_init前的.got.plt节

如图5.10，可以看到调用dl\_init后0x404008和0x404010处的两个8字节的数据发生改变，出现了两个地址0x7f85442c2190和0x7f85442ad200。这就是GOT[1]和GOT[2]。

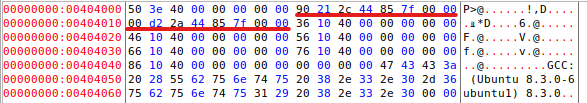


图5.10 执行dl\_init后的.got.plt节

如图5.11红线部分，其中GOT[1]指向重定位表（依次为.plt节需要重定位的函数的运行时地址）用来确定调用的函数地址。

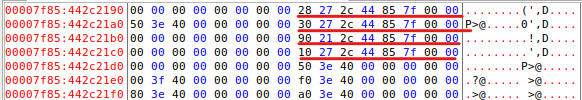


图5.11 0x7f85442c2190指向的重定位表

如图5.12，GOT[2]指向的目标程序是动态链接器ld-linux.so运行时地址。

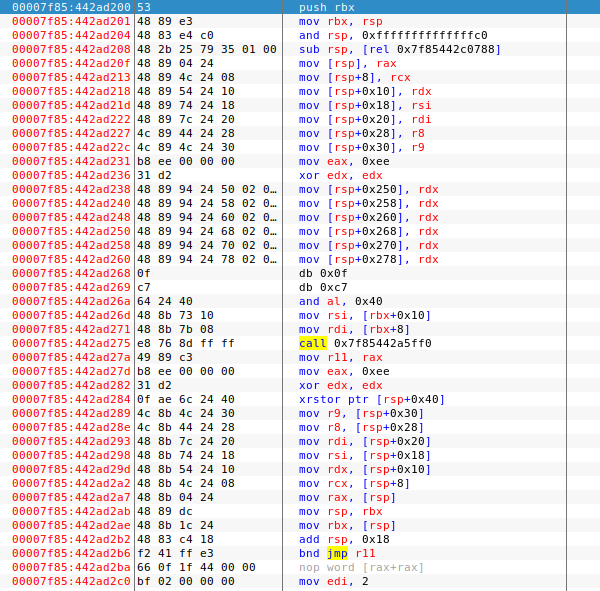


图5.12 0x7f85442ad200处的动态链接器

## 5.8 本章小结

本章介绍了链接的概念和作用，对链接后生成的可执行文件hello的elf格式文件进行了分析，分析了hello的虚拟地址空间、重定位过程、执行过程的各种处理操作。

**（第5章1分）**

# 第6章 hello进程管理

## 6.1 进程的概念与作用

进程的概念：一个执行中的程序的实例，同时也是系统进行资源分配和调度的基本单位。一般情况下，包括文本区域、数据区域和堆栈。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。

进程的作用：，它提供一个假象，好像我们的程序独占地使用内存系统，处理器好像是无间断的执行我们程序中的指令，我们程序中的代码和数据好像是系统内存中唯一的对象。

## 6.2 简述壳Shell-bash的作用与处理流程

shell-bash的作用：shell-bash是一个C语言程序，它代表用户执行进程，它交互性地解释和执行用户输入的命令，能够通过调用系统级的函数或功能执行程序、建立文件、进行并行操作等。同时它也能够协调程序间的运行冲突，保证程序能够以并行形式高效执行。bash还提供了一个图形化界面，提升交互的速度。

shell-bash的处理流程：

(1)终端进程读取用户由键盘输入的命令行。

(2)分析命令行字符串，获取命令行参数，并构造传递给execve的argv向量

(3)检查第一个命令行参数是否是一个内置的shell命令

(3)如果不是内部命令，调用fork()创建新进程/子进程

(4)在子进程中，用步骤2获取的参数，调用execve()执行指定程序。

(5)如果用户没要求后台运行(命令末尾没有&号）否则shell使用waitpid等待作业终止后返回。

(6)如果用户要求后台运行(如果命令末尾有&号），则shell返回；

## 6.3 Hello的fork进程创建过程

首先，打开Terminal输入：./hello 1183710113 许健 1

接下来shell会分析这条命令，由于./hello不是一条内置的命令，于是判断./hello的语义是执行当前目录下的可执行目标文件hello，然后Terminal会调用fork床架一个新的运行的子进程，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）一份副本，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。父进程与子进程之间的区别在于它们拥有不同的PID。

流程图如图6.1。

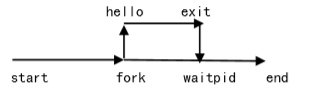


图6.1 fork创建子进程流程图

## 6.4 Hello的execve过程

在fork之后，子进程调用execve函数，execve函数在新创建的子进程的上下文中加载并运行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。只有发生错误时execve才会返回到调用程序。所以，execve调用一次且从不返回。

加载并运行hello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。

（3）映射共享区域。如果hello程序与共享对象链接，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。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。下一次调度这个进程时，它将从这个入口点开始执行。

## 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。

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，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。

一个进程执行它的控制流的一部分的每一时间段叫做时间片。

处理器通常用某个控制寄存器的一个模式位来提供用户模式和内核模式的功能。设置了模式位时，进程就运行在内核模式中，该进程可以执行指令集中的任何指令，可以访问系统中的任何内存位置。没有设置模式位时，进程就运行在用户模式中，用户模式中的进程不允许执行特权指令。

在进程执行的某些时刻，内核可以决定抢占当前进程，并重新开始一个先前被抢占了的进程的决定叫做调度。如图6.2，上下文切换的流程是：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。

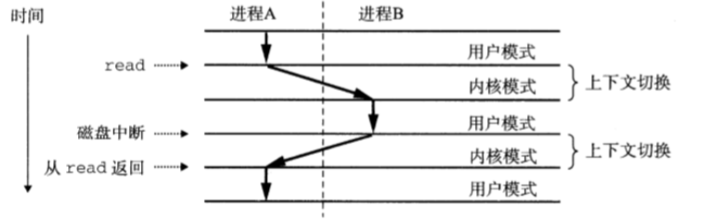


图6.2 进程的上下文切换

然后分析hello的进程调度，hello在刚开始运行时内核为其保存一个上下文，进程在用户模式下运行，当没有异常或中断信号的产生，hello将一直正常地执行，而当出现异常或系统中断时，内核将启用调度器休眠当前进程，并在内核模式中完成上下文切换，将控制传递给其他进程。

当程序在执行sleep函数时，系统调用显式地请求让调用进程休眠，调度器抢占当前进程，并发生上下文切换，将控制转移到新的进程，此时计时器开始，当计时器达到传入的第四个参数大小（这里是1s）时，产生一个中断信号，中断当前正在进行的进程，进行上下文切换恢复hello的上下文信息，控制会回到hello进程中。当循环结束后，程序调用getchar函数用getchar时，由用户模式进入内核模式，内核中的陷阱处理程序请求来自键盘缓冲区的信号传输，并执行上下文切换把控制转移给其他进程。数据传输结束之后，引发一个中断信号，控制回到hello进程中，执行return，进程终止。

## 6.6 hello的异常与信号处理

1.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。

2.hello执行过程中可能出现的信号如图6.3。



图6.3 信号

3. hello对各种信号的处理的分析。

（1）正常运行hello程序。结果如图6.4，可以看出，程序在执行结束后，进程被回收。

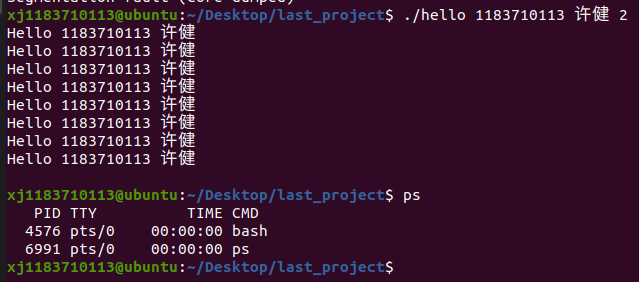


图6.4 正常运行hello

（2）随便乱按。结果如图6.5。发现乱按会将输入的内容保存在缓冲区，等进程结束后作为命令行的内容输入。

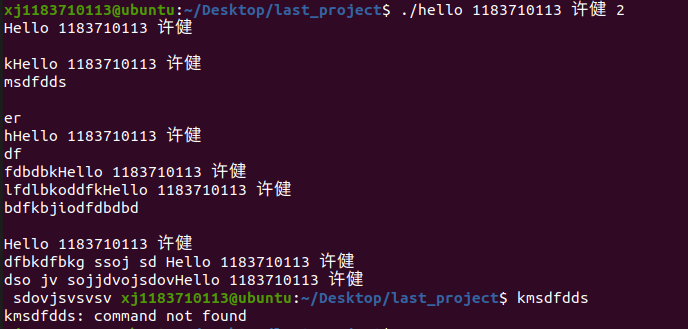


图6.5 运行过程中不停乱按

（3）运行过程中按下Ctrl-C。结果如图6.6，发现会向进程发送SIGINT信号。信号处理程序终止并回收进程。

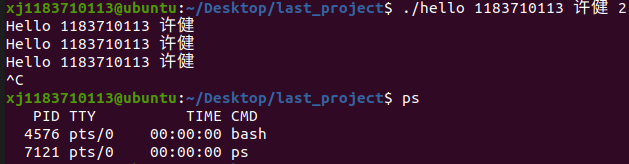


图6.6 运行过程中按下Ctrl-C

（4）运行过程中按下Ctrl-Z。结果如图6.7所示，当按下Ctrl-Z之后，shell进程收到SIGSTP信号，信号处理函数的逻辑是打印屏幕回显、将hello进程挂起，通过ps命令我们可以看出hello进程没有被回收，其进程号时7308，用jobs命令看到job ID是1，状态是Stopped，使用fg 1命令将其调到前台，此时shell程序首先打印hello的命令行命令，然后继续运行打印剩下的信息，之后再按下Ctrl-Z，将进程挂起。

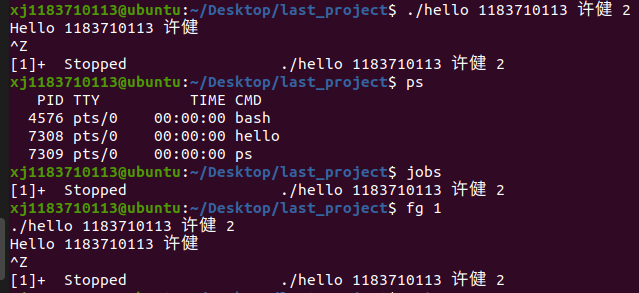


图6.7 运行过程中按下Ctrl-Z

此时，用pstree查看进程，发现hello进程在图6.8的位置。

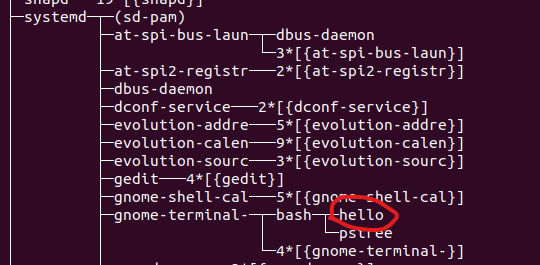


图6.8 在pstree中查看hello

再输入kill -9 7308终止hello进程，再用jobs命令查看发现刚才的hello进程已经被终止了，在ps命令下看也没有hello进程了，说明进程被终止，然后被回收。如图6.9。

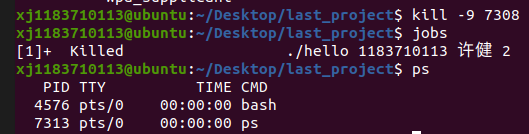


图6.9 用kill命令给hello进程发送SIGINT信号

## 6.7本章小结

本章主要介绍了进程的概念与作用，阐述了shell的作用和处理流程以及hello的fork进程的创建过程和execve的过程，最后分析了hello的执行过程和过程中出现的异常的处理。

**（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：在有地址变换功能的计算机中，访内指令给出的地址(操作数)叫逻辑地址，也叫相对地址。分为两个部分，一个部分为段基址，另一个部分为段偏移量。

线性地址：逻辑地址经过段机制后转化为线性地址，为描述符:偏移量的组合形式。分页机制中线性地址作为输入。

虚拟地址：CPU启动保护模式后，程序运行在虚拟地址空间中。与物理地址相似，虚拟内存被组织为一个存放在磁盘上的N个连续的字节大小的单元组成的数组，其每个字节对应的地址成为虚拟地址。虚拟地址包括VPO（虚拟页面偏移量）、VPN（虚拟页号）、TLBI（TLB索引）、TLBT（TLB标记）。

物理地址：CPU通过地址总线的寻址，找到真实的物理内存对应地址。CPU对内存的访问是通过连接着CPU和北桥芯片的前端总线来完成的。在前端总线上传输的内存地址都是物理内存地址。

结合hello的反汇编文件，如图7.1中的第34行面函数的起始地址0x4010c1，这里的0x4010c1是逻辑地址的偏移量部分，偏移量再加上代码段的段地址就得到了main函数的虚拟地址（线性地址），虚拟地址是现代系统的一个抽象概念，再经过MMU的处理后将得到实际存储在计算机存储设备上的地址。

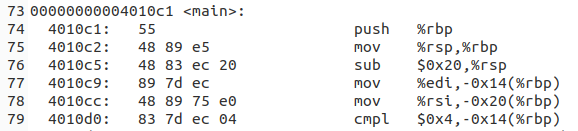


图7.1 hello反汇编文件中的main函数

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

1.基本原理：

在段式存储管理中，将程序的地址空间划分为若干个段，这样每个进程有一个二维的地址空间。在段式存储管理系统中，为每个段分配一个连续的分区，而进程中的各个段可以不连续地存放在内存的不同分区中。程序加载时，操作系统为所有段分配其所需内存，这些段不必连续，物理内存的管理采用动态分区的管理方法。

在为某个段分配物理内存时，可以采用首先适配法、下次适配法、最佳适配法等方法。在回收某个段所占用的空间时，要注意将收回的空间与其相邻的空间合并。

程序通过分段划分为多个模块，如代码段、数据段、共享段：可以分别编写和编译；可以针对不同类型的段采取不同的保护；可以按段为单位来进行共享，包括通过动态链接进行代码共享。这样做的优点是：可以分别编写和编译源程序的一个文件，并且可以针对不同类型的段采取不同的保护，也可以按段为单位来进行共享。

总的来说，段式存储管理的优点是：没有内碎片，外碎片可以通过内存紧缩来消除；便于实现内存共享。缺点与页式存储管理的缺点相同，进程必须全部装入内存。

2.段式管理的数据结构：

为了实现段式管理，操作系统需要如下的数据结构来实现进程的地址空间到物理内存空间的映射，并跟踪物理内存的使用情况，以便在装入新的段的时候，合理地分配内存空间。

（1）进程段表：描述组成进程地址空间的各段，可以是指向系统段表中表项的索引。每段有段基址，即段内地址。在系统中为每个进程建立一张段映射表，其结构如图7.2。

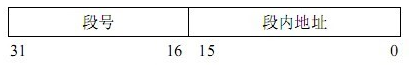


图7.2 段映射表结构

（2）系统段表：系统所有占用段（已经分配的段）。

（3）空闲段表：内存中所有空闲段，可以结合到系统段表中。

3.段式管理的地址变换

在段式管理系统中，整个进程的地址空间是二维的，即其逻辑地址由段号和段内地址两部分组成。为了完成进程逻辑地址到物理地址的映射，处理器会查找内存中的段表，由段号得到段的首地址，加上段内地址，得到实际的物理地址。这个过程也是由处理器的硬件直接完成的，操作系统只需在进程切换时，将进程段表的首地址装入处理器的特定寄存器当中。这个寄存器一般被称作段表地址寄存器。过程如图7.3所示。

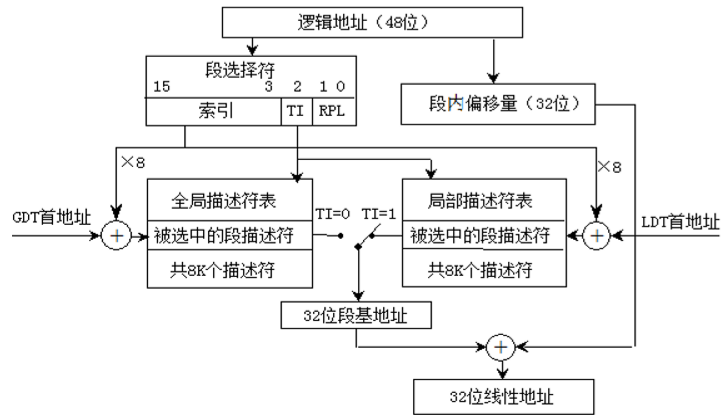


图7.3 段表地址的地址变换

## 7.3 Hello的线性地址到物理地址的变换-页式管理

1.基本原理

将程序的逻辑地址空间划分为固定大小的页，而物理内存划分为同样大小的页框。程序加载时，可将任意一页放入内存中任意一个页框，这些页框不必连续，从而实现了离散分配。该方法需要CPU的硬件支持，来实现逻辑地址和物理地址之间的映射。在页式存储管理方式中地址结构由两部构成，前一部分是VPN（虚拟页号），后一部分是VPO（虚拟页偏移量）。如图7.4所示。



图7.4 物理地址的结构

页式管理方式的优点：没有外碎片；一个程序不必连续存放；便于改变程序占用空间的大小(主要指随着程序运行，动态生成的数据增多，所要求的地址空间相应增长)。

页式管理方式的缺点：要求程序全部装入内存，没有足够的内存，程序就不能执行。

2.页式管理的数据结构

在页式系统中进程建立时，操作系统为进程中所有的页分配页框。当进程撤销时收回所有分配给它的页框。在程序的运行期间，如果允许进程动态地申请空间，操作系统还要为进程申请的空间分配物理页框。操作系统为了完成这些功能，必须记录系统内存中实际的页框使用情况。操作系统还要在进程切换时，正确地切换两个不同的进程地址空间到物理内存空间的映射。这就要求操作系统要记录每个进程页表的相关信息。为了完成上述的功能，—个页式系统中，一般要采用如下的数据结构。

页表：页表将虚拟内存映射到物理页。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。页表是一个页表条目（PTE）的数组。虚拟地址空间的每个页在页表中一个固定偏移量处都有一个PTE。假设每个PTE是由一个有效位和一个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。

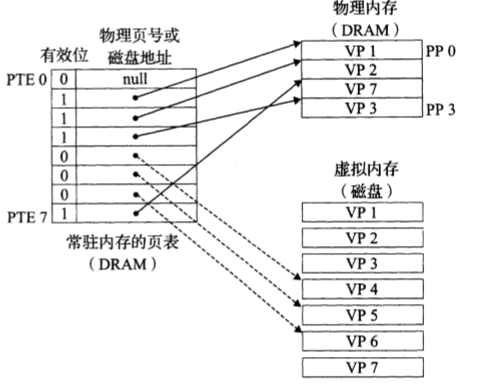


图7.5 页表

3.页式管理地址变换

MMU利用VPN来选择适当的PTE，将列表条目中PPN和虚拟地址中的VPO串联起来，就得到相应的物理地址。

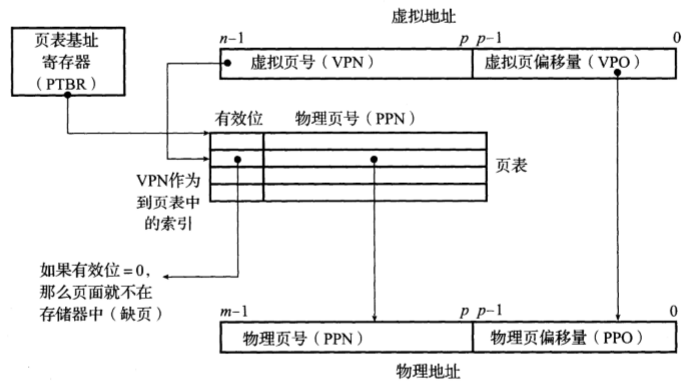


图7.6 使用页表的地址翻译

## 7.4 TLB与四级页表支持下的VA到PA的变换

为了消除每次CPU产生一个虚拟地址MMU就查阅一个PTE带来的时间开销，许多系统都在MMU中包括了一个关于PTE的小的缓存，称为翻译后被缓冲器（TLB），TLB的速度快于L1 cache。

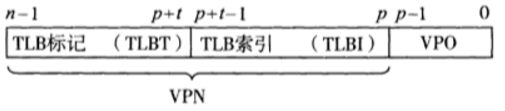


图7.7 虚拟地址中用以访问TLB的组成部分

TLB通过虚拟地址VPN部分进行索引，分为索引（TLBI）与标记（TLBT）两个部分。这样，MMU在读取PTE时会直接通过TLB，如果不命中再从内存中将PTE复制到TLB。

同时，为了减少页表太大而造成的空间损失，可以使用层次结构的页表页压缩页表大小。如图7.8所示。

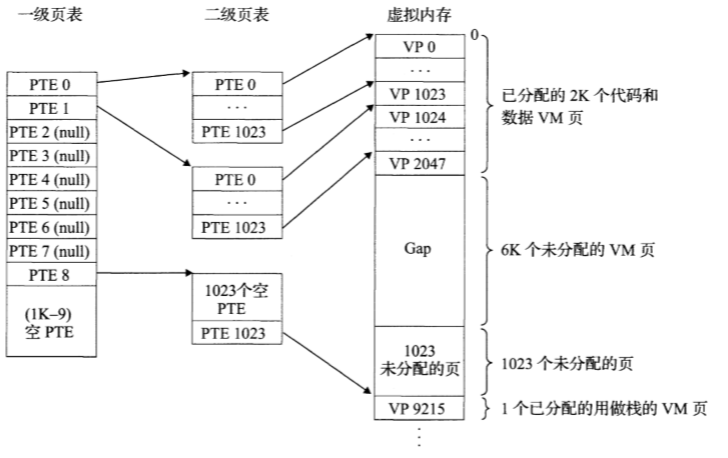


图7.8 一个二级页表层次结构

Core i7使用的是四级页表。如图7.9所示，在四级页表层次结构的地址翻译中，虚拟地址被划分为4个VPN和1个VPO。每个第i个VPN都是一个到第i级页表的索引，第j级页表中的每个PTE都指向第j+1级某个页表的基址，第四级页表中的每个PTE包含某个物理页面的PPN，或者一个磁盘块的地址。为了构造物理地址，在能够确定PPN之前，MMU必须访问四个PTE。

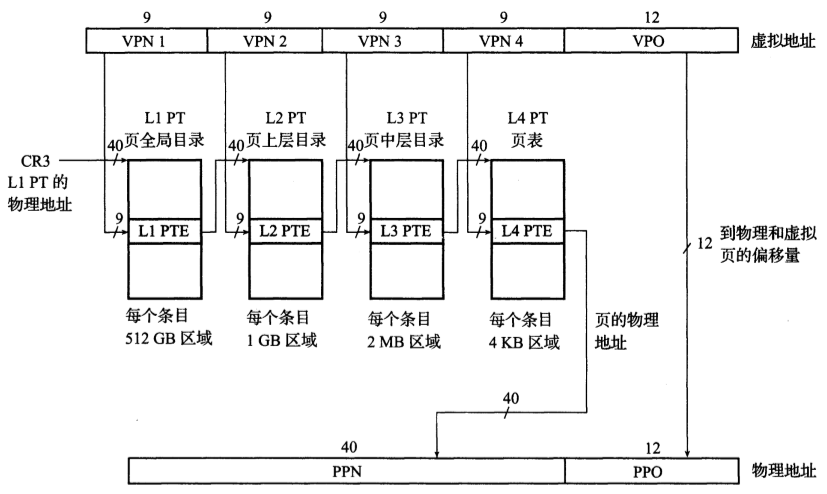


图7.9 Core i7页表翻译

## 7.5 三级Cache支持下的物理内存访问

通过7.3和7.4两节，hello的物理地址已经得知，现在需要访问该物理地址。在现代计算机中，存储器被组织成层次结构，因为这样可以最大程度地平衡访存时间和存储器成本。所以在CPU在访存时并不是直接访问内存，而是访问内存之前的三级cache。已知Core i7的三级cache是物理寻址的，块大小为64字节。LI和L2是8路组相联的，而L3是16路组相联的。Corei7实现支持48位虚拟地址空间和52位物理地址空间。

得到了52位物理地址，接下来CPU把地址发送给L1，因为L1块大小为64字节，所以B=64，b=6。又L1是8路组相联的，所以S=8，s=3。标记位t有52-6-3=43位，即是得到的52位物理地址的前43位。首先，根据物理地址的s位组索引索引到L1 cache中的某个组，然后在该组中查找是否有某一行的标记等于物理地址的标记并且该行的有效位为1，若有，则说明命中，从这一行对应物理地址b位块偏移的位置取出一个字节，若不满足上面的条件，则说明不命中，需要继续访问下一级cache，访问的原理与L1相同，若是三级cache都没有要访问的数据，则需要访问内存，从内存中取出数据并放入cache。

图7.10展示了Core i7的地址翻译过程。

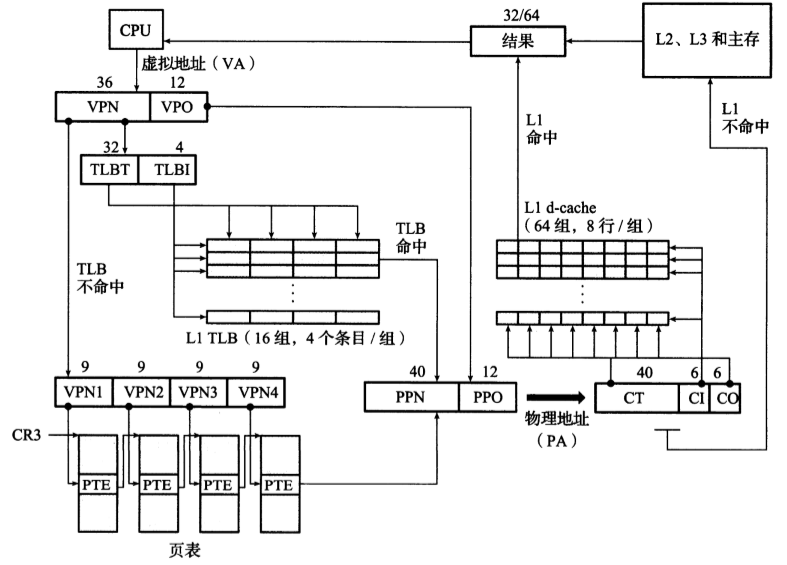


图7.10 TLB与4级页表下Core i7的地址翻译

## 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，并分配给他一个唯一的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。它将两个进程中的每个页面都标记为只读，并将两个进程中每个区域结构都标记为私有的写时复制。其内存映射如图7.11。

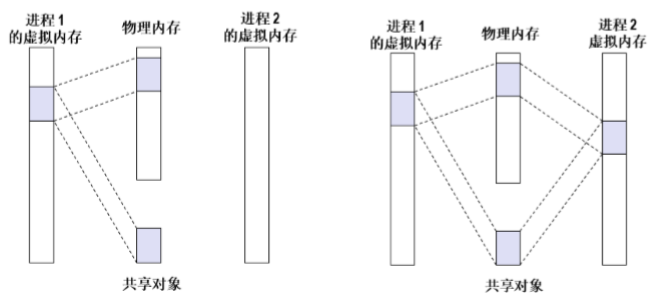


图7.11 hello进程fork时的内存映射

## 7.7 hello进程execve时的内存映射

execve函数调用驻留在内核区域的启动加载器代码，在当前进程中加载并运行包含在可执行目标文件hello中的程序，用hello程序有效地替代了当前程序。

加载并运行hello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，栈和堆地址也是请求二进制零的，初始长度为零。

（3）映射共享区域，hello程序与共享对象libc.so链接，libc.so是动态链接到这个程序中的，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

## 7.8 缺页故障与缺页中断处理

缺页故障是一种常见的故障，当指令引用一个虚拟地址，在MMU中查找页表时发现与该地址相对应的物理地址不在内存中，因此必须从磁盘中取出的时候就会发生故障。其处理流程遵循图7.12所示的故障处理流程。

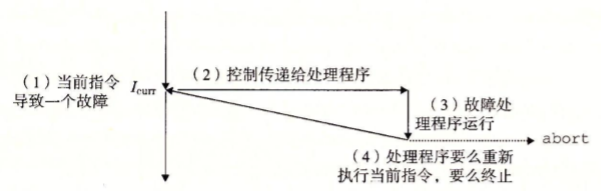
****

图7.12故障处理流程

缺页中断处理：缺页处理程序是系统内核中的代码，选择一个牺牲页面，如果这个牺牲页面被修改过，那么就将它交换出去，换入新的页面并更新页表。当缺页处理程序返回时，CPU重新启动引起缺页的指令，这条指令再次发送VA到MMU，这次MMU就能正常翻译VA了。

## 7.9动态存储分配管理

1.动态内存分配器的基本原理

在程序运行时程序员使用动态内存分配器(比如malloc)获得虚拟内存。动态内存分配器维护者一个进程的虚拟内存区域，称为堆。分配器将堆视为一组不同大小的块的集合来维护，每个块要么是已分配的，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。

分配器的类型有两种：显式分配器和隐式分配器。

显式分配器：要求应用显式地释放任何已分配的块。例如，C语言中的malloc函数申请了一块空间之后需要free函数释放这个块

隐式分配器：应用检测到已分配块不再被程序所使用，就释放这个块。比如Java，ML和Lisp等高级语言中的垃圾收集。

2.带边界标签的隐式空闲链表分配器原理

带边界标签的隐式空闲链表的堆块结构如图7.13。一个块是由一个字的头部、有效载荷、可能的一些额外的填充，以及在块的结尾处的一个字的脚部组成的。头部编码了这个块的大小（包括头部和所有的填充），以及这个块是已分配的还是空闲的。如果我们强加一个双字的对齐约束条件，那么块大小就总是8的倍数，且块大小的最低3位总是0。因此，我们只需要内存大小的29个高位，释放剩余的3位来编码其他信息。在这种情况中，我们用其中的最低位（已分配位）来指明这个块是已分配的还是空闲的。

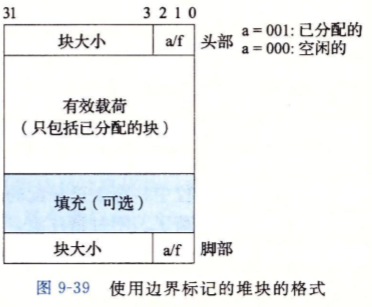


图7.13使用边界标记的堆块结构

寻找一个空闲块的方式有三种：

（1）首次适配：从头开始搜索空闲链表，选择第一个合适的空闲块：可以取总块数（包括已分配和空闲块）的线性时间，但是会在靠近链表起始处留下小空闲块的“碎片”。

（2）下一次适配：和首次适配相似，只是从链表中上一次查询结束的地方开始，优点是比首次适应更快：避免重复扫描那些无用块。但是一些研究表明，下一次适配的内存利用率要比首次适配低得多。

（3）最佳适配：查询链表，选择一个最好的空闲块适配，剩余最少空闲空间，优点是可以保证碎片最小——提高内存利用率，但是通常运行速度会慢于首次适配。

3.关于堆块的合并有如图7.14的四种情况。在情况1中，两个邻接的块都是已分配的，因此不可能进行合并。所以当前块的状态只是简单地从已分配变成空闲。在情况2中，当前块与后面的块合并。用当前块和后面块的大小的和来更新当前块的头部和后面块的脚部。在情况3中，前面的块和当前块合并。用两个块大小的和来更新前面块的头部和当前块的脚部。在情况4中，要合并所有的三个块形成一个单独的空闲块，用三个块大小的和来更新前面块的头部和后面块的脚部。在每种情况中，合并都是在常数时间内完成的。

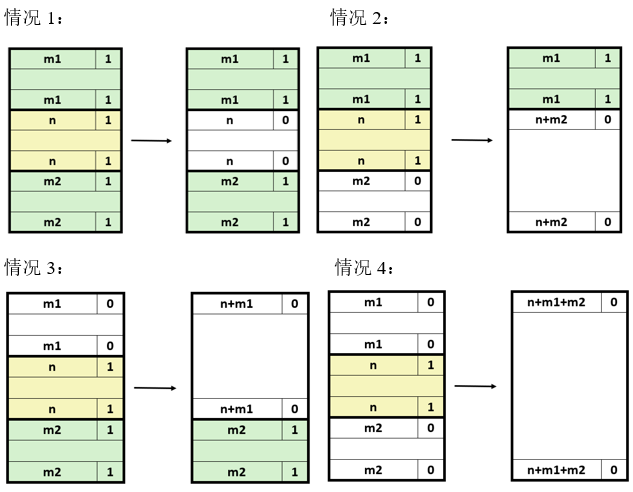


图7.14 堆块合并的四种情况

3.显式空间链表的基本原理

显式空间链表的堆块结构如图7.15。将空闲块组织成链表形式的数据结构。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。

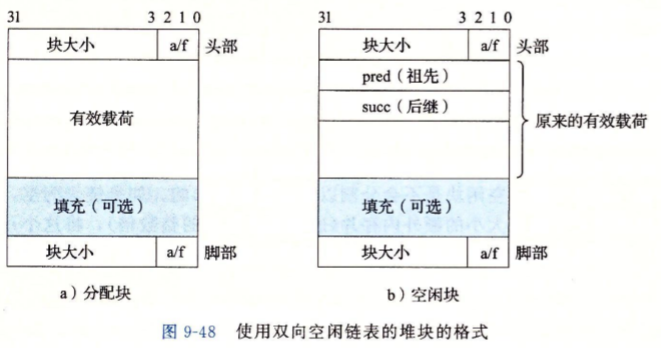


图7.15显式空间链表的堆块结构

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使线性的，也可以是一个常数，这取决于我们选择的空闲链表中块的排序策略。

链表的维护方式有两种：一种方法是用后进先出（LIFO）的顺序维护链表，将新释放的块放置在链表的开始处。使用LIFO的顺序和首次适配的放置策略，分配器会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，那么合并也可以在线性时间内完成。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内存利用率，接近最佳适配的利用率。

一般而言，显示链表的缺点是空闲块必须足够大，以包含所有需要的指针，以及头部和可能的脚部，这就导致了更大的最小块大小，也潜在地提高了内部碎片的程度。

## 7.10本章小结

本章主要介绍了hello的存储地址空间、intel的段式管理、hello的页式管理，以及TLB与四级页表支持下的VA到PA的变换过程和三级Cache支持下的物理内存访问。还阐述了hello进程fork和execve时的内存映射、缺页故障的处理流程和动态存储分配器的管理。

**（第7章 2分）**

# 第8章 hello的IO管理

## 8.1 Linux的IO设备管理方法

所有的I/O设备都被模型化为文件，而所有的输入和输出都被当作相应文件的读和写来完成，这种将设备优雅的映射为文件的方式，允许Linux内核引出一个简单的，低级的应用接口，称为Unix I/O，这使得所有的输入和输出都能以一种统一且一致的方式来执行。

## 8.2 简述Unix IO接口及其函数

Unix I/O 接口的几种操作：

（1）打开文件：程序要求内核打开文件，内核返回一个小的非负整数（描述符），用于标识这个文件。程序在只要记录这个描述符便能记录打开文件的所有信息。

（2）shell在进程的开始为其打开三个文件：标准输入、标准输出和标准错误。

（3）改变当前文件的位置：对于每个打开的文件，内核保存着一个文件位置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操作显式地设置文件的当前位置为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字节的文件，当k>=m时执行读操作会出发一个称为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。

（5）关闭文件：内核释放打开文件时创建的数据结构以及占用的内存资源，并将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。

UnixI/O函数：

（1）intopen(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。

（2）intclose(int fd);关闭一个打开的文件，返回操作结果。

（3）ssize\_t read(int fd, 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。

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

printf函数的实现如图8.1。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。

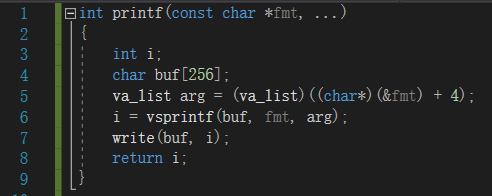


图8.1 printf函数的代码

然后查看vsprintf代码如图8.2。可以看出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。

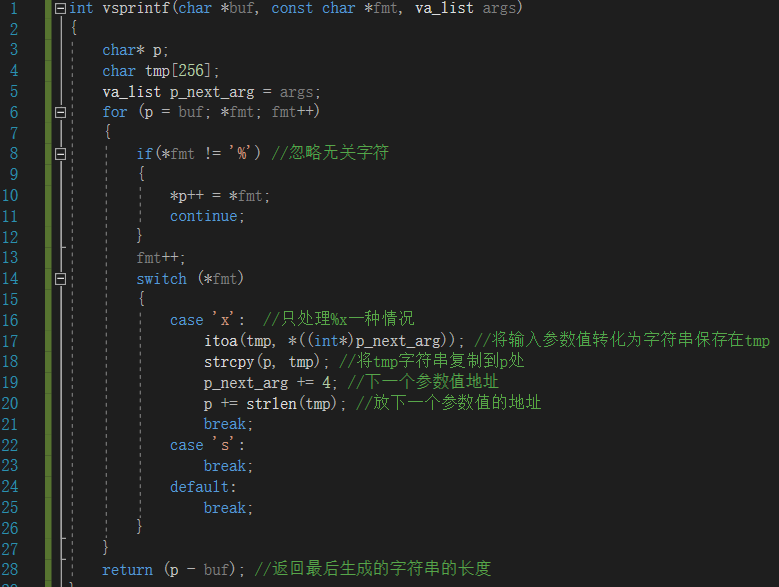


图8.2 vsprintf函数的代码

查看write函数如图8.3。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。

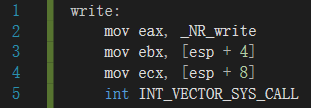


图8.3 write函数的代码

查看syscall的实现如图8.4。syscall将字符串中的字节“Hello 11183710113 许健”从寄存器中通过总线复制到显卡的显存中.

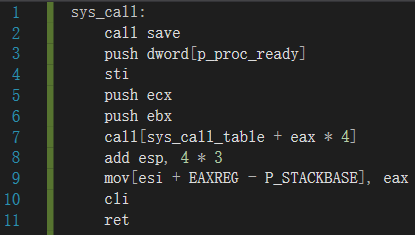


图8.4 syscall的代码

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）。于是我们的打印字符串“Hello 11183710113 许健”就显示在了屏幕上。

## 8.4 getchar的实现分析

当用户按键时，键盘接口会得到一个代表该按键的键盘扫描码，同时产生一个中断请求，中断请求抢占当前进程运行键盘中断子程序，键盘中断子程序先从键盘接口取得该按键的扫描码，然后将该按键扫描码转换成ASCII码，保存到系统的键盘缓冲区之中。

再看getchar的代码如图8.5。可以看到，getchar调用了read函数，read函数也通过sys\_call调用内核中的系统函数，将读取存储在键盘缓冲区中的ASCII码，直到读到回车符，然后返回整个字符串，getchar函数只从中读取第一个字符，其他的字符被缓存在输入缓冲区。

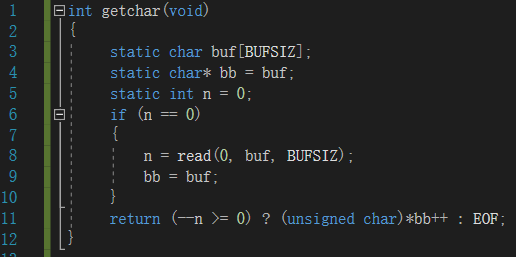


图8.5 getchar 的代码

## 8.5本章小结

本章介绍了Linux中I/O设备的管理方法，UnixI/O接口和函数，并且分析了printf和getchar函数是如何通过UnixI/O函数实现其功能的。

**（第8章1分）**

# 结论

hello程序终于完成了它艰辛但可谓精彩的一生。hello的一生大事记如下：

（1）编写，通过editor将代码键入hello.c

（2）预处理，经过预处理器cpp的预处理，处理以#开头的行，得到hello.i

（3）编译，编译器ccl将得到的hello.i编译成汇编文件hello.s

（4）汇编，汇编器as又将hello.s翻译成机器语言指令得到可重定位目标文件hello.o

（5）链接，链接器ld将hello.o与动态链接库链接生成可执行目标文件hello，至此，hello成为了一个可以运行的程序。

（6）运行，在shell中输入./hello 1183710113 许健 1，

（7）创建子进程，shell进程调用fork为其创建子进程

（8）加载，shell调用execve，execve调用启动加载器，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。

（9）执行，CPU为其分配时间片，在一个时间片中，hello享有CPU资源，顺序执行自己的控制逻辑流

（10）访问内存，当CPU访问hello时，请求一个虚拟地址，MMU把虚拟地址转换成物理地址并通过三级cache访存。

（11）动态申请内存，printf会调用malloc向动态内存分配器申请堆中的内存。

（12）信号，hello运行过程中可能遇到各种信号，shell为其提供了各种信号处理程序。

（13）结束，shell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，hello结束了它的一生。

感悟：即使是一个简单的hello.c也需要操作系统提供很多的支持，并且每一步都经过了设计者的深思熟虑，在有限的硬件水平下把程序的时间和空间性能都做到了近乎完美的利用。在做本次大作业的过程中，回顾了整个计算机系统，回顾了很多调试器的使用方法，对整个程序的运行过程有了一个新的认识。

**（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。

附件1：hello.c——原文件

附件2：hello.i——预处理之后文本文件

附件3：hello.s——编译之后的汇编文件

附件4：hello.o——汇编之后的可重定位目标执行

附件5：hello——链接之后的可执行目标文件

附件6：hello.elf——hello.o的elf格式，用来看hello.o的各节信息

附件7：hello.ob——hello.o的反汇编文件，用来看汇编器翻译后的汇编代码

附件8：hello1.ob——hello的反汇编文件，用来看链接器链接后的汇编代码

**（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. 深入理解计算机系统. 龚奕利 译.

[2] 库函数getchar()详解https://blog.csdn.net/hulifangjiayou/article/details/40480467

[3] Linux进程虚拟地址空间https://www.cnblogs.com/xelatex/p/3491305.html

[4] 虚拟地址、逻辑地址、线性地址、物理地址 https://blog.csdn.net/rabbit\_in\_android/article/details/49976101

[5] 内存地址转换与分段 https://blog.csdn.net/drshenlei/article/details/4261909

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

**（参考文献0分，缺失 -1分）**