

**计算机系统**

**大作业**

题 目 程序人生-Hello’s P2P

专 业 计算机科学与技术

学　　 号 2022113573

班　　 级 2203101

学 生 张宇杰

指 导 教 师 史先俊

**计算机科学与技术学院**

**2023年12月**

**摘 要**

hello作为程序语言初学者都能读懂并执行的程序，其有着简单的外表，但有着丰富的内涵。本论文借由hello从预处理到IO管理的整个过程，结合《深入理解计算机系统》，简单介绍并分析了其中涉及的计算机系统的知识与内容，并对其中涉及到的一些机制进行了深入的剖析。

**关键词：**CSAPP；hello；

**目 录**

[第1章 概述 - 4 -](#_Toc154005274)

[1.1 Hello简介 - 4 -](#_Toc154005275)

[1.2 环境与工具 - 4 -](#_Toc154005276)

[1.3 中间结果 - 5 -](#_Toc154005277)

[1.4 本章小结 - 5 -](#_Toc154005278)

[第2章 预处理 - 6 -](#_Toc154005279)

[2.1 预处理的概念与作用 - 6 -](#_Toc154005280)

[2.2在Ubuntu下预处理的命令 - 6 -](#_Toc154005281)

[2.3 Hello的预处理结果解析 - 7 -](#_Toc154005282)

[2.4 本章小结 - 8 -](#_Toc154005283)

[第3章 编译 - 9 -](#_Toc154005284)

[3.1 编译的概念与作用 - 9 -](#_Toc154005285)

[3.2 在Ubuntu下编译的命令 - 9 -](#_Toc154005286)

[3.3 Hello的编译结果解析 - 9 -](#_Toc154005287)

[3.4 本章小结 - 16 -](#_Toc154005288)

[第4章 汇编 - 17 -](#_Toc154005289)

[4.1 汇编的概念与作用 - 17 -](#_Toc154005290)

[4.2 在Ubuntu下汇编的命令 - 17 -](#_Toc154005291)

[4.3 可重定位目标elf格式 - 17 -](#_Toc154005292)

[4.4 Hello.o的结果解析 - 23 -](#_Toc154005293)

[4.5 本章小结 - 25 -](#_Toc154005294)

[第5章 链接 - 26 -](#_Toc154005295)

[5.1 链接的概念与作用 - 26 -](#_Toc154005296)

[5.2 在Ubuntu下链接的命令 - 26 -](#_Toc154005297)

[5.3 可执行目标文件hello的格式 - 27 -](#_Toc154005298)

[5.4 hello的虚拟地址空间 - 29 -](#_Toc154005299)

[5.5 链接的重定位过程分析 - 30 -](#_Toc154005300)

[5.6 hello的执行流程 - 33 -](#_Toc154005301)

[5.7 Hello的动态链接分析 - 34 -](#_Toc154005302)

[5.8 本章小结 - 35 -](#_Toc154005303)

[第6章 hello进程管理 - 36 -](#_Toc154005304)

[6.1 进程的概念与作用 - 36 -](#_Toc154005305)

[6.2 简述壳Shell-bash的作用与处理流程 - 36 -](#_Toc154005306)

[6.3 Hello的fork进程创建过程 - 36 -](#_Toc154005307)

[6.4 Hello的execve过程 - 37 -](#_Toc154005308)

[6.5 Hello的进程执行 - 37 -](#_Toc154005309)

[6.6 hello的异常与信号处理 - 38 -](#_Toc154005310)

[6.7本章小结 - 41 -](#_Toc154005311)

[第7章 hello的存储管理 - 42 -](#_Toc154005312)

[7.1 hello的存储器地址空间 - 42 -](#_Toc154005313)

[7.2 Intel逻辑地址到线性地址的变换-段式管理 - 42 -](#_Toc154005314)

[7.3 Hello的线性地址到物理地址的变换-页式管理 - 43 -](#_Toc154005315)

[7.4 TLB与四级页表支持下的VA到PA的变换 - 45 -](#_Toc154005316)

[7.5 三级Cache支持下的物理内存访问 - 46 -](#_Toc154005317)

[7.6 hello进程fork时的内存映射 - 47 -](#_Toc154005318)

[7.7 hello进程execve时的内存映射 - 47 -](#_Toc154005319)

[7.8 缺页故障与缺页中断处理 - 48 -](#_Toc154005320)

[7.9动态存储分配管理 - 49 -](#_Toc154005321)

[7.10本章小结 - 52 -](#_Toc154005322)

[第8章 hello的IO管理 - 53 -](#_Toc154005323)

[8.1 Linux的IO设备管理方法 - 53 -](#_Toc154005324)

[8.2 简述Unix IO接口及其函数 - 53 -](#_Toc154005325)

[8.3 printf的实现分析 - 54 -](#_Toc154005326)

[8.4 getchar的实现分析 - 56 -](#_Toc154005327)

[8.5本章小结 - 57 -](#_Toc154005328)

[结论 - 58 -](#_Toc154005329)

[附件 - 59 -](#_Toc154005330)

[参考文献 - 60 -](#_Toc154005331)

# 第1章 概述

## 1.1 Hello简介

我们通过键盘，向计算机输入一行行代码，这串代码组合成了一个C源文件，也就是我们的主角hello.c。

接下来，hello.c经过了预处理器cpp，编译器cc1，汇编器as，链接器ld这些家伙一顿好生折腾，最终生成一个可以加载到内存执行的可执行目标文件hello。

然后，我们在Shell中执行命令“./hello 2022113573 张宇杰 1”，Shell通过fork函数创建一个新的进程，然后在子进程里通过execve函数将hello程序加载到内存。虚拟内存机制通过mmap为hello进程规划了一片虚拟空间，调度器为hello进程规划进程执行的时间片，使其能够与其他进程一起合理利用CPU与内存的资源。hello完成了其P2P(From Program to Process)的过程。

之后，CPU一条条的从hello的.text段取指令，寄存器们的值随着程序的执行而不断变化着，异常处理程序监视着键盘的输入。hello中的syscall系统调用会使进程触发陷阱，让内核接手进程，执行write函数，将一串字符传递给屏幕IO的映射文件。

映射文件对传入数据进行分析，读取VRAM，然后在屏幕上将字符显示出一行行字符串“Hello 2022113573 张宇杰”。

最后，hello程序运行结束，Shell通过waitpid函数通知内核回收hello进程，hello进程消失。至此，hello完成了其程序执行的一生，从不带来什么，也不带走什么，是真正的O2O(From Zero to Zero)。

## 1.2 环境与工具

硬件环境：

12th Gen Intel(R) Core(TM) i5-12500H 3.10 GHz

RAM 16.0 GB

NVIDIA GeForce RTX 3050 Ti Laptop GPU

软件环境：

Windows 11 家庭中文版 22H2

VMware Workstation 16.2.4 build-20089737

Ubuntu 22.04.1

Linux version 6.2.0-37-generic

gcc version 11.4.0

调试工具：

GNU gdb 12.1

edb 1.4.0

## 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。

|  |  |
| --- | --- |
| hello.c | C源文件 |
| hello.i | C预处理文件，由hello.c预处理得到 |
| hello.s | 汇编语言文件，由hello.i编译得到 |
| hello.o | 可重定位目标文件，由hello.s汇编得到 |
| hello\_elf.txt | 由readelf生成的关于hello.o的ELF信息 |
| hello\_asm.txt | 由objdump生成的关于hello.o的反汇编信息 |
| hello | 可执行文件，由hello.o链接得到 |
| hello\_exe\_elf.txt | 由readelf生成的关于hello的ELF信息 |
| hello\_exe\_asm.txt | 由objdump生成的关于hello的反汇编信息 |

## 1.4 本章小结

本章描述了hello从出生到死亡的完整过程，以及本人在撰写论文时所使用的软硬件环境，测试使用的调试工具，和产生的中间文件。

# 第2章 预处理

## 2.1 预处理的概念与作用

**2.1.1 预处理的概念**

预处理，指预处理器根据预处理器指令，修改C源程序的过程。C预处理器不是编译器的组成部分，但是它是编译过程中一个单独的步骤。简而言之，预处理器只不过是一个文本替换工具而已，它们会指示编译器在实际编译之前完成所需的预处理。我们把C预处理器(C Preprocessor)简写为CPP。

**2.1.2 预处理的作用**

所有的预处理器命令都是以井号(#)开头，其大致有以下种类：

1、#define，#undef

定义宏，进行单纯的字符串替换，以及取消已定义的宏

2、#include

包含一个源代码文件，将包含文件的内容插入源代码中

3、#ifdef，#ifndef，#if，#else等

条件编译，根据条件的真假，选择性地编译随后的代码

4、#error

生成编译错误提示消息，并停止编译

5、#pragma

设置编译器的状态，或者指示编译器完成特定动作

## 2.2在Ubuntu下预处理的命令

可用以下命令进行预处理：

gcc -m64 hello.c -E -o hello.i

预处理过程如图：

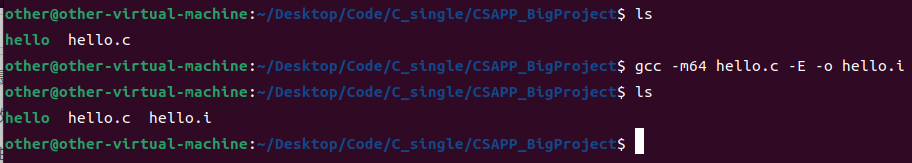


图2.2-1 在Ubuntu下预处理的过程

## 2.3 Hello的预处理结果解析

使用文本编辑器打开生成的hello.i与源文件hello.c，对二者进行对比。容易发现，hello.c本来只有24行，但生成的hello.i文件有3092行，其内容被大大扩充了。

对比发现，原来hello.c中的main函数，在hello.i中位于文件末尾的位置，且内容与hello.c中的一模一样。

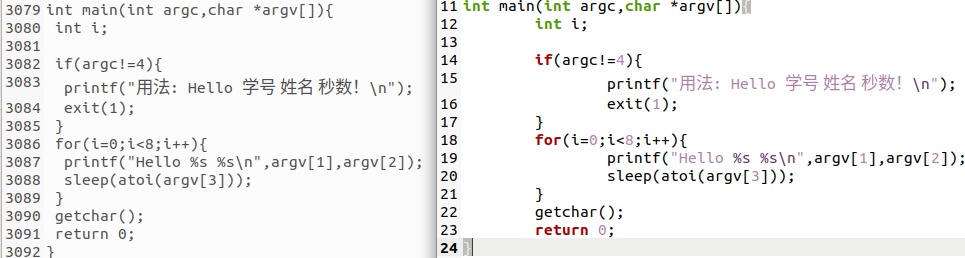


图2.3-1 hello.i (左) 与 hello.c (右) 中的main函数内容相同

对比发现，除了main函数，hello.c中的注释部分遭到了删除。

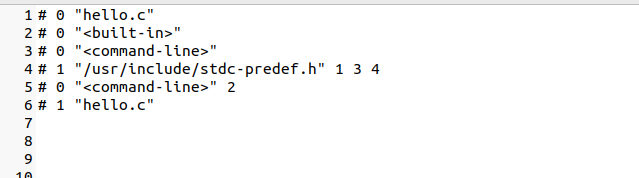
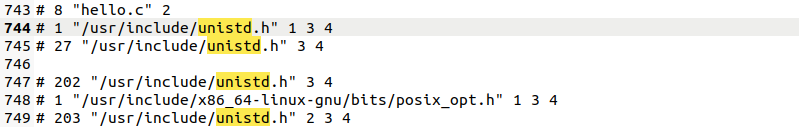
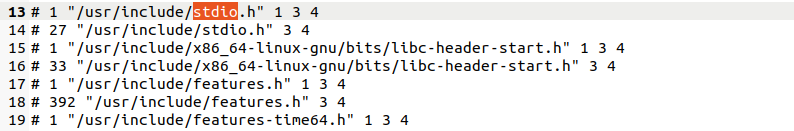


图2.3-2 hello.c中的注释在hello.i中不见踪影

对比发现，hello.c中的#include部分不见了，取而代之的是相应的头文件的内容，如stdio.h，unistd.h与stdlib.h。



组图2.3-3 hello.i中多出的相关头文件的内容

除此之外，hello.i中还多出了原先并没有被直接包含在hello.c中的头文件的内容，它们是由hello.c中引用的头文件stdio.h，unistd.h与stdlib.h所间接应用的，均被递归地插入到了hello.i之中。



图2.3-4 hello.i中被递归插入的部分头文件信息

## 2.4 本章小结

本章介绍了预处理的概念，以及预处理的几个主要的功能：宏定义及其取消，头文件的包含，条件编译，#error实现的编译期断言，#pragma调整编译器状态等。

之后，使用gcc -m64 hello.c -E -o hello.i命令，在Ubuntu下对hello.c进行了预处理，得到了hello.i文件。

最后，将hello.i与hello.c两者的内容进行了对比，发现了预处理对注释的丢弃以及对头文件的包含。

# 第3章 编译

## 3.1 编译的概念与作用

**3.1.1 编译的概念**

在这里，编译是指编译器(cc1)将文本文件.i翻译成汇编语言程序文件.s的过程。

**3.1.2 编译的作用**

在编译的过程中，会对预处理之后的.i文件进行词法分析、语法分析、语义分析等一系列工作，还会根据编译选项对代码进行适当优化。

## 3.2 在Ubuntu下编译的命令

可用以下命令进行编译：

gcc -m64 hello.i -S -o hello.s

编译过程如图：

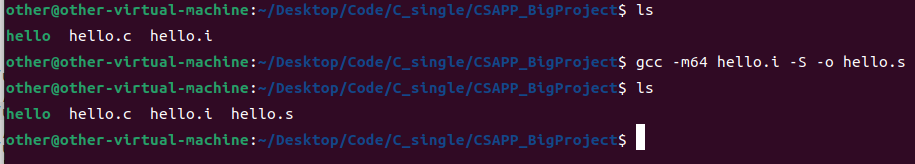


图3.2-1 在Ubuntu下编译的过程

之后，本文使用gcc -m64 -Og -no-pie -fno-stack-protector -fno-PIC hello.i -S -o hello.s生成的结果进行分析。

## 3.3 Hello的编译结果解析

**3.3.1 数据**

1、整形常量

在hello.c中出现的整形常量在hello.s中都有对应出现。编译器将整形常量编译为立即数。

* 整型变量argc与整型常量4的不等比较，其中的整型常量4以立即数$4的形式出现

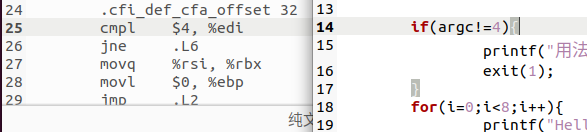


图3.3.1-1.1 比较中的整型常量翻译为立即数

* exit(1)的调用，其中的整型常量1以立即数$1的形式出现



图3.3.1-1.2 传参中的整型常量翻译为立即数

* 对整型变量i的赋值i=0，其中的整型常量0以立即数$0的形式出现

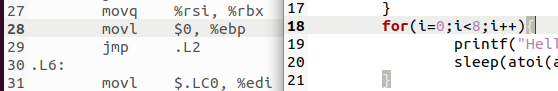


图3.3.1-1.3 赋值中的整型常量翻译为立即数

2、字符串常量

在hello.c中出现的字符串常量在hello.s中都有对应出现。编译器将字符串常量存入了 .rodata节之中。

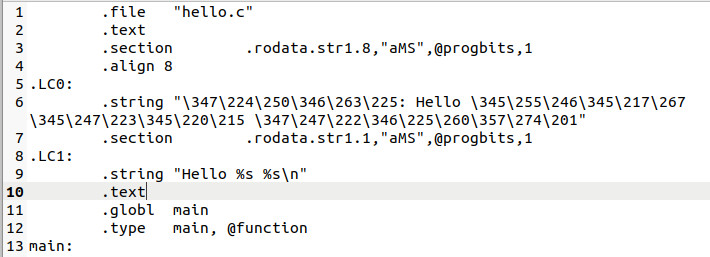


图3.3.1-2.1 存储在 .rodata节的字符串常量

在使用字符串常量的时候，编译器将字符串常量的地址根据传参规则赋值至对应寄存器中，具体如图所示：

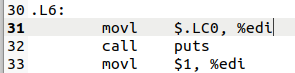


图3.3.1-2.2 puts传参中的字符串常量

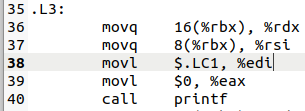


图3.3.1-2.3 printf传参中的字符串常量

3、局部变量

hello.c中仅有一个局部变量，即循环变量int i。其被编译器翻译为了对寄存器 %ebp的相应操作。



图3.3.1-3.1 %ebp与i的行为对照

**3.3.2 赋值**

hello.c中仅出现了一次对整型变量i的赋值，翻译为了mov语句



图3.3.2 对整型变量的赋值翻译为了mov语句

**3.3.3 类型转换**

hello.c中有语句：sleep(atoi(argv[3]));

atoi返回类型为int，sleep接收类型为unsigned int的参数。发生了隐式类型转换。

但分析其对应汇编语句后发现，atoi将返回值保存在%eax中，随后直接将%eax作为参数赋值给%edi，便开始调用sleep，好像所谓的类型转换根本不存在一样。

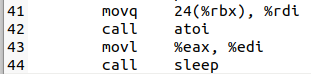


图3.3.3 类型转换——什么都不做 (int转unsigned int)

**3.3.4 算术运算**

hello.c中仅出现了一次整型变量i自增的算术运算i++。其翻译为了对应寄存器 %ebp的加1操作。



图3.3.4 被翻译为add操作的自增运算

**3.3.5 关系操作**

对于关系操作，编译器一般会将其翻译为cmp操作，随后通过je、jne、ja、jb、jg、jl等跳转命令实现分支结构。在hello.c中，出现了两处关系操作：

* 整型变量argc与整型常量4的不等比较，翻译为cmp + jne

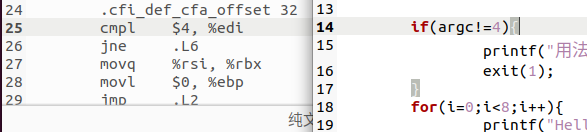


图3.3.5-1 不等比较翻译为cmp + jne

* 整型变量i与整型常量8的小于比较，翻译为cmp + jle

(对于整数，小于8等价于小于等于7)



图3.3.5-2 小于比较翻译为cmp + jle

**3.3.6 数组操作**

编译器对数组操作往往翻译为对地址的加减操作，取值时使用mov操作，取地址时使用lea操作。在hello.c中，出现了对数组argv的操作，使用下标运算符 [] 对argv数组中的元素进行访问。

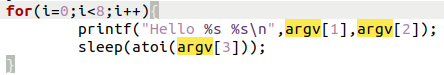


图3.3.6-1 hello.c中对数组argv的操作

分析编译结果，hello.c中对数组argv的下标访问，在hello.s中翻译为了在对数组argv的首地址(在这里是%rbx)的加法操作后的mov操作。

printf原语句为printf("Hello %s %s\n",argv[1],argv[2]);

argv[1]作为printf的第二个实参，对应操作为movq 8(%rbx), %rsi

argv[2]作为printf的第三个实参，对应操作为movq 16(%rbx), %rdx

argv数组中元素的类型为char\*，在64位中占8字节，故上述对%rbx的操作中的增量为8。

对atoi的有关分析也是类似的。

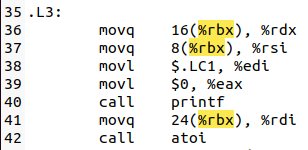
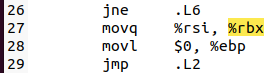


图3.3.6-2 hello.s中对数组argv的操作

**3.3.7 控制转移**

1、if

hello.c中通过argc!=4以判断输入参数的正确与否，并使用if进行分支处理。在hello.s中，这是通过cmp与jne的组合实现的(由于编译器的原因，此处编译出了无条件跳转jmp)。

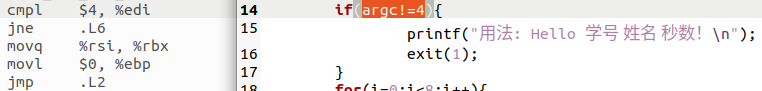


图3.3.7-1.1 hello.s中的分支控制

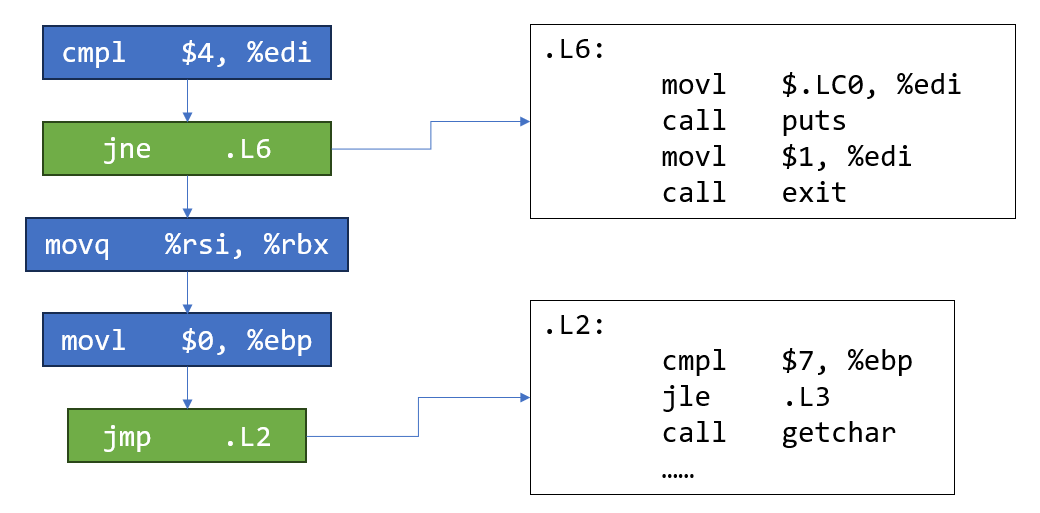


图3.3.7-1.2 hello.s中分支控制的示意图

2、for

hello.c中使用了for循环来进行对语句的重复执行。在hello.s中，这主要是通过往回跳转的跳转语句实现的。

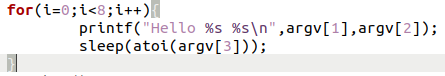
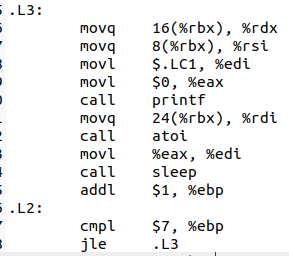


图3.3.7-2.1 hello.s中的for循环

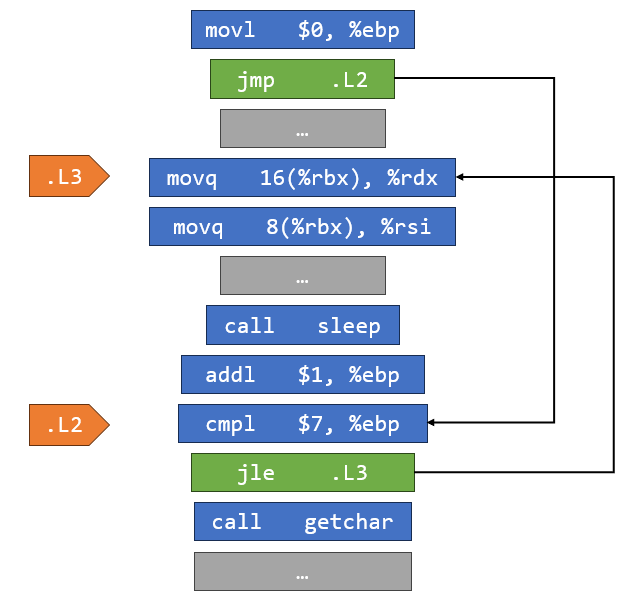


图3.3.7-2.2 hello.s中for循环的示意图

**3.3.8 函数操作**

hello.c中一共出现了6次函数调用，具体见下：

* 对printf的调用，仅进行字符串的简单输出，被优化为了对puts的调用

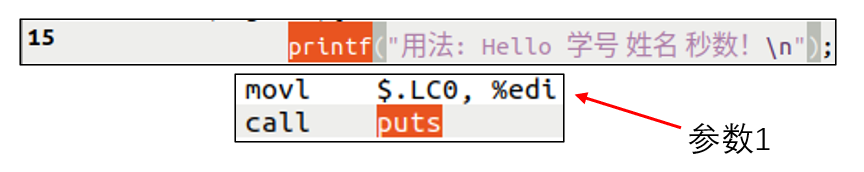


图3.3.8-1 第一次函数调用——puts

* 对exit的调用

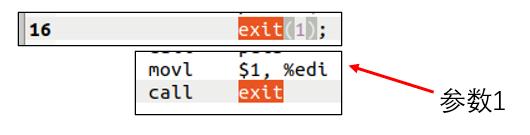


图3.3.8-2 第二次函数调用——exit

* 对printf的调用，进行了字符串的格式化输出

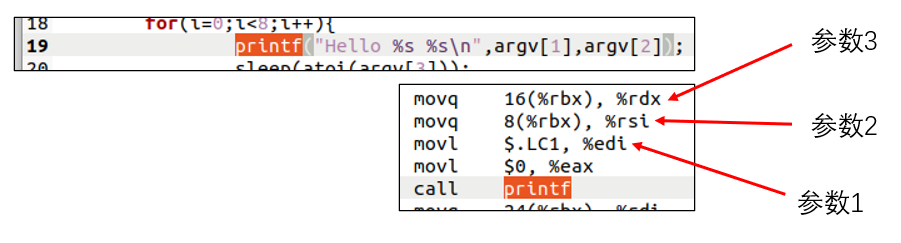


图3.3.8-3 第三次函数调用——printf

* 对atoi的调用，将字符串转换为整型



图3.3.8-4 第四次函数调用——atoi

* 对sleep的调用，进行程序的阻塞。其将atoi的返回值作为了输入实参。

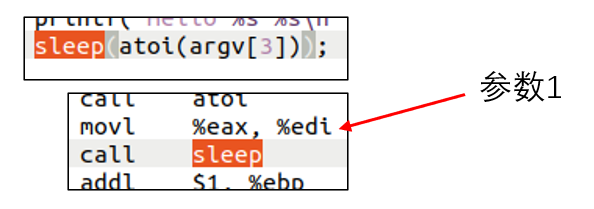


图3.3.8-5 第五次函数调用——sleep

* 对getchar的调用，进行程序的阻塞。

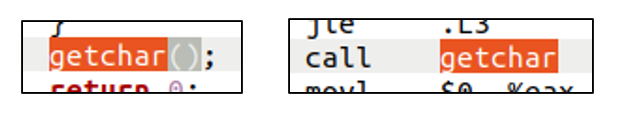


图3.3.8-6 第六次函数调用——getchar

## 3.4 本章小结

本章主要介绍了编译的的概念以及作用，其作用包括将高级程序语言翻译为汇编语言，并根据编译选项进行适当优化。

之后，使用gcc -m64 -Og -no-pie -fno-stack-protector -fno-PIC hello.i -S -o hello.s命令，在Ubuntu下对hello.i进行编译，得到了hello.s文件。

最后，根据要求中列出的C语言的数据与操作，从数据、赋值、类型转换、算术运算、关系操作、数组操作、控制转移、函数操作这几个角度分析了hello.c源程序中的语句是怎样被编译器转化为hello.s中的汇编语言的。

# 第4章 汇编

## 4.1 汇编的概念与作用

**4.1.1 汇编的概念**

在这里，汇编是指汇编器(as)将.s文件翻译成二进制的机器语言指令，把这些指令打包成一种叫做可重定位目标程序的格式，将结果保存在目标文件.o中的过程。

**4.1.2 汇编的作用**

汇编将人尚能看懂，机器无法识别的汇编语言，翻译为了人看不懂，但机器可以直接识别的机器语言。

## 4.2 在Ubuntu下汇编的命令

可用以下命令进行汇编：

as hello.s -o hello.o

汇编过程如图：

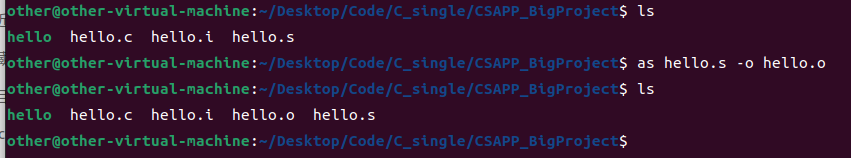


图4.2 在Ubuntu下汇编的过程

## 4.3 可重定位目标elf格式

使用readelf -a hello.o命令，可在终端查看hello.o的ELF格式。也可使用readelf -a hello.o > hello\_elf.txt将结果重定向至文本文件hello\_elf.txt中，便于查看。

4.3.1 ELF头

ELF 文件头位于目标文件最开始的位置，含有整个文件的一些基本信息。文件头中含有整个文件的结构信息，包括一些控制单元的大小。

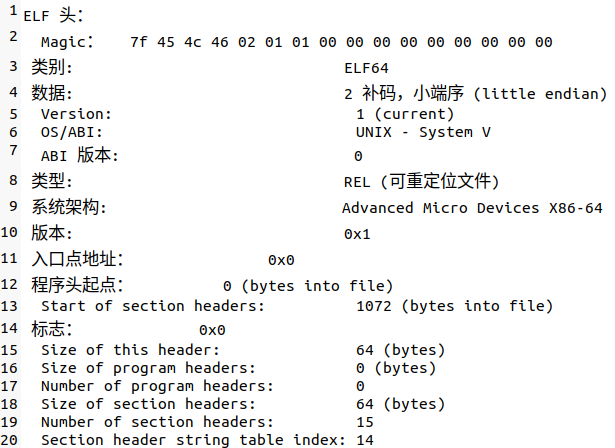


图4.3.1 hello.o的ELF头

可以在elf.h源码中找到64位 ELF的文件头的数据结构，其结构体定义如下：

1. **typedef** **struct**
2. {
3. unsigned **char** e\_ident[EI\_NIDENT]; /\* Magic number and other info \*/
4. Elf64\_Half    e\_type;         /\* Object file type \*/
5. Elf64\_Half    e\_machine;      /\* Architecture \*/
6. Elf64\_Word    e\_version;      /\* Object file version \*/
7. Elf64\_Addr    e\_entry;        /\* Entry point virtual address \*/
8. Elf64\_Off e\_phoff;        /\* Program header table file offset \*/
9. Elf64\_Off e\_shoff;        /\* Section header table file offset \*/
10. Elf64\_Word    e\_flags;        /\* Processor-specific flags \*/
11. Elf64\_Half    e\_ehsize;       /\* ELF header size in bytes \*/
12. Elf64\_Half    e\_phentsize;        /\* Program header table entry size \*/
13. Elf64\_Half    e\_phnum;        /\* Program header table entry count \*/
14. Elf64\_Half    e\_shentsize;        /\* Section header table entry size \*/
15. Elf64\_Half    e\_shnum;        /\* Section header table entry count \*/
16. Elf64\_Half    e\_shstrndx;     /\* Section header string table index \*/
17. } Elf64\_Ehdr;

以下将按照顺序介绍Elf64\_Ehdr结构体中的成员。

1. e\_ident (ELF Header-Identification)

这16个字节含有 ELF 文件的识别标志。作为一个数组，它的各个索引位置的字节数据有固定的含义，提供一些用于解码和解析文件内容的数据，是不依赖于具体操作系统的。

ELF 格式提供的目标文件框架可以支持多种处理器，以及多种编码方式。针对不同的体系结构和编码格式，ELF 文件的内容是会截然不同的。如果不知道编码格式，系统将无法知道怎么去读取目标文件；如果系统结构与本机不同，也将无法解析和运行。这些信息需要以独立的格式存放在一个默认的地方，所有系统都约定好从文件的同一个地方来读取这些信息，这就是 ELF 标识的作用。

e\_ident中每个元素对应的内容见下表。

|  |  |  |
| --- | --- | --- |
| 名称 | 位置 | 说明 |
| EI\_MAG0 | 0 | 文件标识(0x7f) |
| EI\_MAG1 | 1 | 文件标识(E, 0x45) |
| EI\_MAG2 | 2 | 文件标识(L, 0x4C) |
| EI\_MAG3 | 3 | 文件标识(F, 0x46) |
| EI\_CLASS | 4 | 指明文件位数，取值：0-非法，1-32位，2-64位 |
| EI\_DATA | 5 | 数据编码格式，取值：0-非法，1-小端，2-大端 |
| EI\_VERSION | 6 | ELF 文件头的版本 |
| EI\_OSABI | 7 | 指明ELF文件操作系统的二进制接口的版本标识符。  值为0：指明为UNIX System V ABI |
| EI\_ABIVERSION | 8 | ABI 版本 |
| EI\_PAD | 9~15 | 补齐字节，一般为0 |

表4.3.1-1 e\_ident中元素对应内容的说明

1. 剩余成员

其余成员的对应含义可见下表：

|  |  |
| --- | --- |
| 成员 | 作用 |
| e\_type | 表明本目标文件属于哪种类型  1：重定位文件；值为2：可执行文件；值为3：动态链接库文件 |
| e\_machine | 指定该文件适用的处理器体系结构 |
| e\_version | 指明目标文件的版本 |
| e\_entry | 指明程序入口的虚拟地址 |
| e\_phoff | 指明程序头表开始处在文件中相对于 ELF 文件初始位置的偏移量 |
| e\_shoff | 指明节头表开始处在文件中的偏移量 |
| e\_flags | 处理器特定的标志位 |
| e\_ehsize | 表明ELF文件头的大小，以字节为单位 |
| e\_phentsize | 表明在程序头表中表项的大小，以字节为单位 |
| e\_phnum | 表明程序头表中的表项数 |
| e\_shentsize | 表明在节头表中表项的大小，以字节为单位 |
| e\_shnum | 表明节头表中的表项数 |
| e\_shstrndx | 表明节头表中与节名字表相对应的表项的索引，存放着节的名字 |

表4.3.1-2 ELF头中剩余成员对应内容的说明

4.3.2 节头表

ELF文件中含有众多的节，这些节携带了ELF文件的所有信息。每一个节又对应有一个节头，节头中有着对节的描述信息，将这些节头组织在一起的数据结构就是节头表(section header table)。

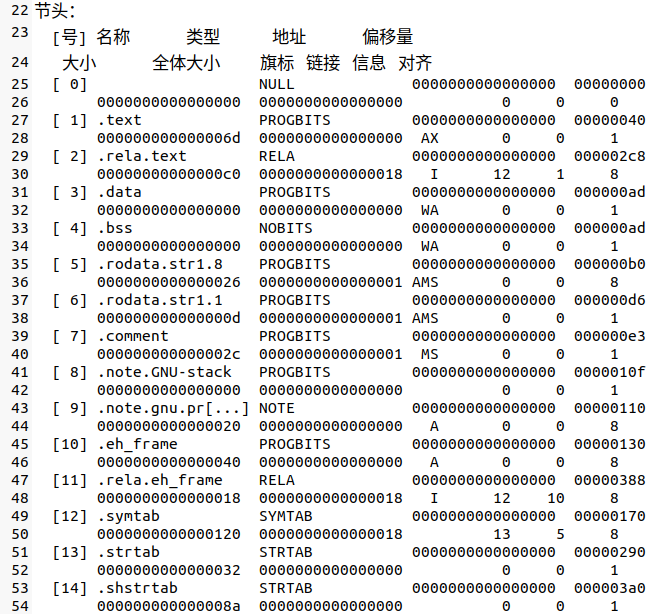


图4.3.2 hello.o的节头表

节头表中的每一项都对应了一个结构体，其定义如下：

1. **typedef** **struct**
2. {
3. Elf64\_Word    sh\_name;     /\* Section name (string tbl index) \*/    // 4 bytes
4. Elf64\_Word    sh\_type;     /\* Section type \*/
5. Elf64\_Xword   sh\_flags;    /\* Section flags \*/                      // 8 bytes
6. Elf64\_Addr    sh\_addr;     /\* Section virtual addr at execution \*/  // 8 bytes
7. Elf64\_Off sh\_offset;   /\* Section file offset \*/                // 8 bytes
8. Elf64\_Xword   sh\_size;     /\* Section size in bytes \*/
9. Elf64\_Word    sh\_link;     /\* Link to another section \*/
10. Elf64\_Word    sh\_info;     /\* Additional section information \*/
11. Elf64\_Xword   sh\_addralign;    /\* Section alignment \*/
12. Elf64\_Xword   sh\_entsize;  /\* Entry size if section holds table \*/
13. } Elf64\_Shdr;

各个成员的含义见下表：

|  |  |
| --- | --- |
| 成员 | 含义 |
| sh\_name | 一个偏移量，指向本节的名字 |
| sh\_type | 指明本节的类型 |
| sh\_flags | 指明本节的属性 |
| sh\_offset | 指明本节的位置 |
| sh\_size | 指明该节的大小，以字节为单位 |
| sh\_link | 指向节头表中本节所对应的位置 |
| sh\_info | 指明该节的附加信息 |
| sh\_addralign | 指明该节内容对齐字节的数量 |
| sh\_entsize | 指明该节对应的每一个表项的大小 |

图4.3.2-1 节头表项各个数据成员的含义

4.3.3 重定位节

重定位是连接符号引用与符号定义的过程。例如，程序调用函数时，关联的调用指令必须在执行时将控制权转移到正确的目标地址。可重定位文件必须包含说明如何修改其节内容的信息。重定位节即包含了这些用于重定位的数据信息。

在hello.o中，重定位节有两个，即 .rela.text节与 .rela.eh\_frame节。



图4.3.3 hello.o的重定位节

上述重定位节中的每一个表项都对应了一个结构体，其定义如下：

1. **typedef** **struct** {
2. Elf64\_Addr      r\_offset;
3. Elf64\_Xword     r\_info;
4. Elf64\_Sxword    r\_addend;
5. } Elf64\_Rela;

r\_offset成员指定了应用可重定位操作的位置。对于可重定位文件，该值表示节偏移值。

r\_info成员指定了必须对其进行重定位的符号表索引以及要应用的重定位类型。

r\_addend成员指定了一个常量加数，用于计算将存储在可重定位字段中的值。

通过对r\_info成员施用宏

#define ELF64\_R\_TYPE\_ID(info) (((Elf64\_Xword)(info)<<56)>>56)

可以得到其对应的重定位类型，即图4.3.3中的“类型”一项。在图4.3.3中，我们见到了三种ELF中最为基本的重定位类型：

* R\_X86\_64\_32 重定位一个使用32位绝对地址的引用。
* R\_X86\_64\_PC32 重定位一个使用32位PC相对地址的引用。
* R\_X86\_64\_PLT32 过程链接表延迟绑定。

它们的具体含义在链接一章再展开说明。

4.3.4 符号表

符号表保存了程序实现或使用的所有全局变量和函数，如果程序引用一个自身代码未定义的符号，则称之为未定义符号。这类引用必须在静态链接期间用其他目标模块或库解决，或在加载时通过动态链接解决。

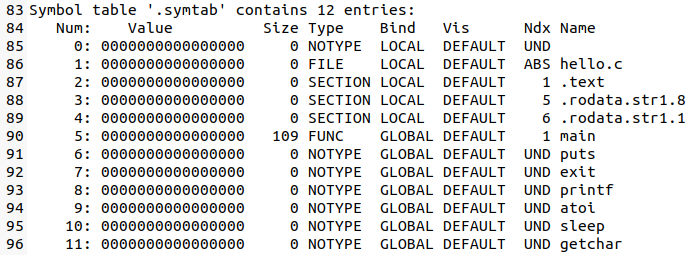


图4.3.4 hello.o的符号表

符号表的每一个表项都对应了一个结构体，其定义如下：

1. **typedef** **struct** {
2. Elf64\_Word  st\_name;
3. unsigned **char**   st\_info;
4. unsigned **char**   st\_other;
5. Elf64\_Half  st\_shndx;
6. Elf64\_Addr  st\_value;
7. Elf64\_Xword st\_size;
8. } Elf64\_Sym;

各个成员的含义见下表：

|  |  |
| --- | --- |
| 成员 | 含义 |
| st\_name | 目标文件的符号字符串表的索引 |
| st\_info | 指明了符号类型与绑定属性 |
| st\_others | 符号的可见性 |
| st\_shndx | 表明该符号对应的节头在节头表中的偏移 |
| st\_value | 相关符号的值。根据上下文，这可以是绝对值，地址等。 |
| st\_size | 符号的大小 |

图4.3.4-1 符号表项中各个数据成员的含义

我们特别关注一下图4.3.4中的Ndx这一列(即结构体中的st\_shndx成员)。符号main，.text，.rodata.str1.1与 .rodata.str1.8这些符号的对应项是有值的。如main对应值为1，对应了节头表中的 .text节，.rodata.str1.8对应值为5，对应了节头表中的 .rodata.str1.8节，等等。而我们先前讨论过的6个调用的函数(puts, exit, printf, atoi, sleep, getchar)的对应值都是UND，即该符号未定义。将这些符号的定义找到即为下一章“链接”时的工作。

## 4.4 Hello.o的结果解析

使用objdump -d -r hello.o命令可以在终端查看hello.o对应的反汇编代码。其中 -d指disassemble，反汇编，-r 指reloc，显示文件的重定位入口。

也可使用objdump -d -r hello.o > hello\_asm.txt将结果重定向至文本文件hello\_asm.txt中，便于查看。

经过hello\_asm.txt与hello.s二者之间的对比，发现了以下的不同：

1、没有了汇编指示符

在hello.s中时常出现的汇编指示符.cfi\_\*\*\*没有在hello\_asm.txt中出现。

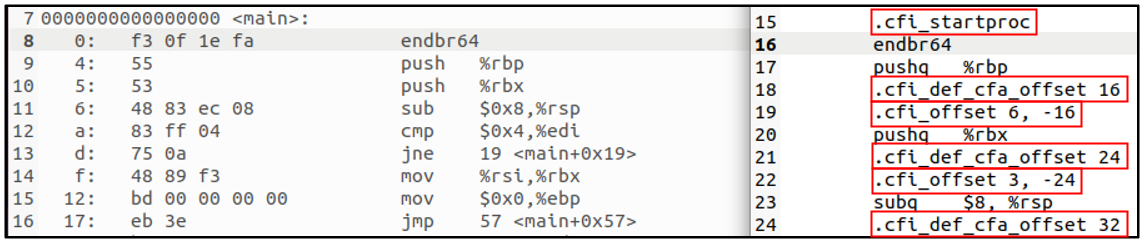


图4.4.1 汇编指示符的消失

2、操作数的进制不同

hello.s中操作数是十进制的，而在hello\_asm.txt中，操作数以十六进制表示。

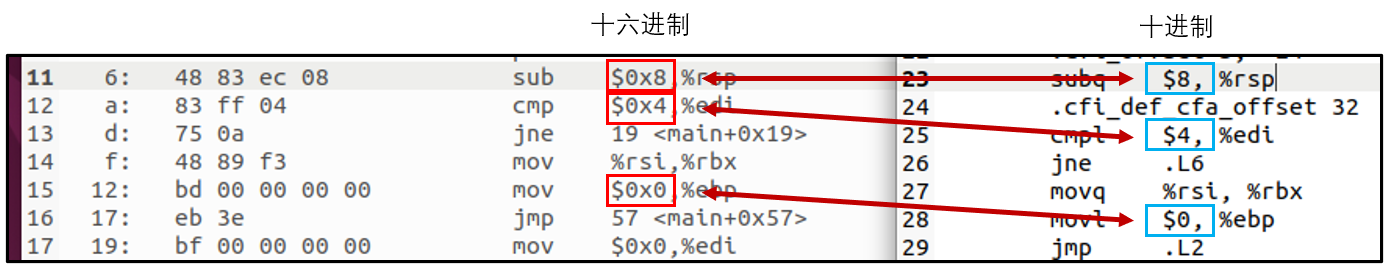


图4.4.2 操作数进制的不同

3、分支转移的不同

hello.s中为汇编语言代码的一些行增加了标签(如 .L3，.L6)，分支转移时，在跳转指令后用对应标签名称表示跳转位置。

而在hello\_asm.txt中，每一行反汇编都有着明确的地址，跳转指令后用相应的地址表示跳转位置。

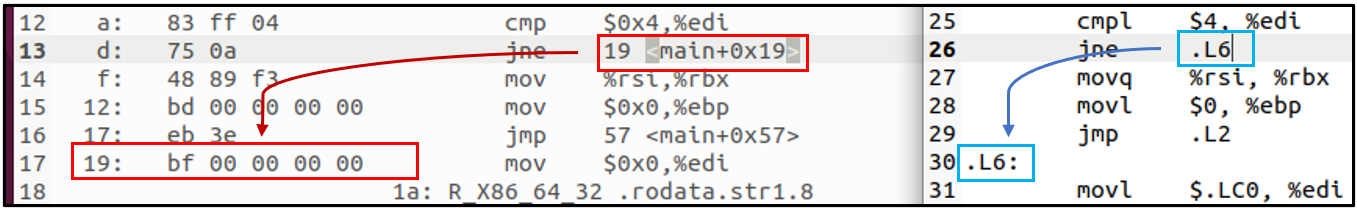


图4.4.3-1 分支转移的不同

有必要指出，hello\_asm.txt中出现的寻址是相对寻址。

如 [d: 75 0a jne 19 <main+0x19>] 这行。当程序运行至这行时，其 %rip指向下一行指令的地址0xf，要从0xf处跳转至0x19处，有增量0x19 - 0xf = 0xa，对应了机器码75 0a中的0a。

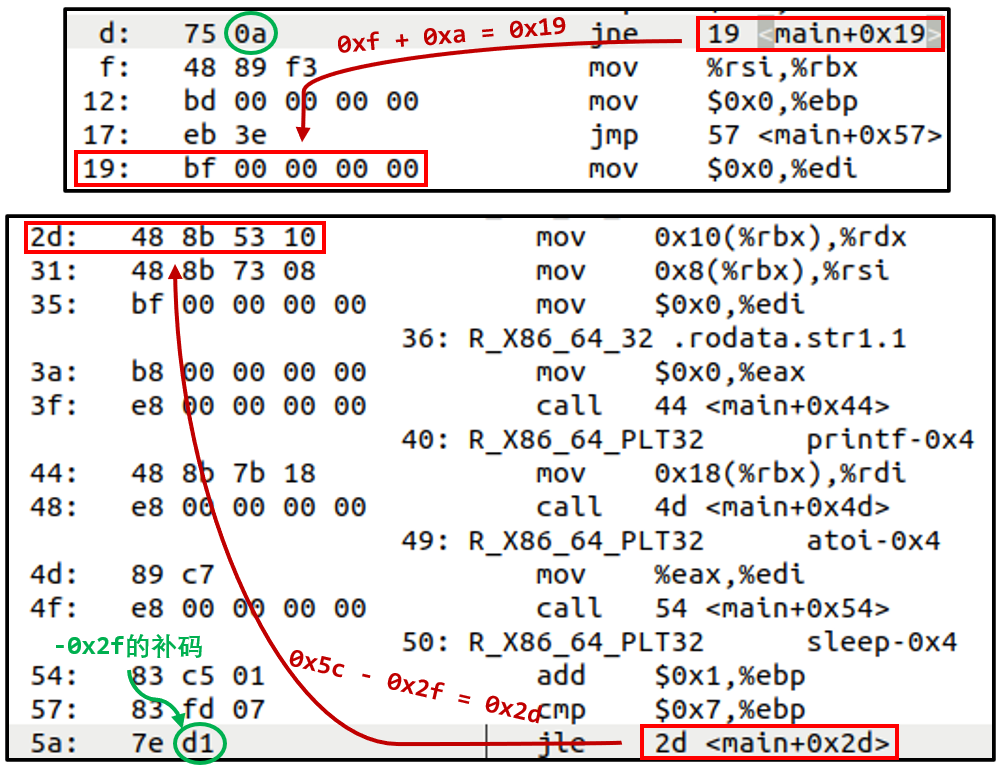


图4.4.3-2 分支转移中的相对寻址

4、字符串常量的引用形式不同

hello.s中用标签对字符串常量进行引用，而在hello\_asm.txt中，使用字符串常量的虚拟地址进行引用。不过这里尚未进行重定向的工作，故地址值都先用0进行占位。

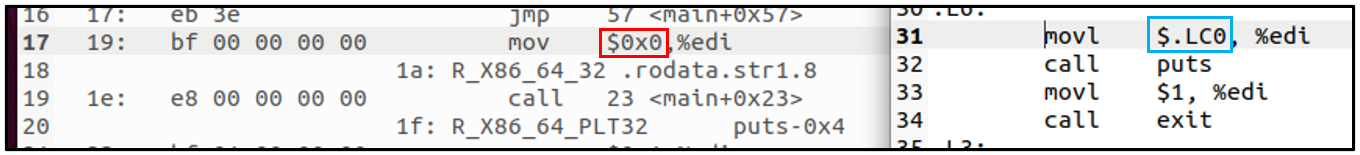


图4.4.4 字符串常量引用形式的不同

5、函数调用的不同

hello.s中直接使用函数名对函数进行引用，而在hello\_asm.txt中，应使用相对下一行指令的地址的偏移值对函数进行引用(与先前第3点“分支转移的不同”中所述内容一致)。不过这里尚未进行重定向的工作，故地址偏移值都先用0进行占位。

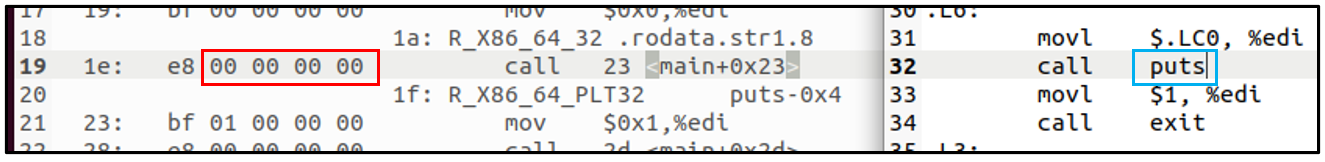


图4.4.5 函数调用的不同

## 4.5 本章小结

本章主要介绍了汇编的概念与作用。汇编语言程序经过编译器(as)转化为机器语言，并把这些指令打包成可重定位目标程序的格式，并保存在目标文件.o中，成为机器可以直接识别的程序。

然后，使用as hello.s -o hello.o命令，在Ubuntu下对hello.s进行汇编，得到了hello.o文件。

之后，使用readelf -a hello.o > hello\_elf.txt命令，对hello.o文件的可重定位目标ELF格式从ELF头、节头表、重定位头、符号表四个内容进行了简单的分析。

最后，使用objdump -d -r hello.o > hello\_asm.txt命令，得到了hello.o的反汇编文件，并将其与hello.s进行对比，从汇编指示符、操作数进制、分支转移、字符串常量、函数调用五个角度进行了不同之处的分析。

# 第5章 链接

## 5.1 链接的概念与作用

**5.1.1 链接的概念**

链接是将各种代码和数据片段收集并组合为一个单一文件的过程，所得到的文件可以被加载到内存之中并执行。链接由链接器(ld)程序执行，链接执行的时机可以是编译时，即源代码被翻译成机器码的时候；以及加载时，即程序被加载器加载到内存并执行的时候；甚至是运行时，即在应用程序执行链接命令的时候。

**5.1.2 链接的作用**

链接的两个主要任务是符号解析和重定位。符号解析将目标文件中的每个全局符号都绑定到一个唯一的定义，而重定位确定每个符号的最终内存地址，并修改对那些目标的引用。链接的存在也使得分离式编译成为了可能。

## 5.2 在Ubuntu下链接的命令

在Ubuntu下，使用ld进行链接的命令如下：

ld -plugin /usr/lib/gcc/x86\_64-linux-gnu/11/liblto\_plugin.so -plugin-opt=/usr/lib/gcc/x86\_64-linux-gnu/11/lto-wrapper -plugin-opt=-fresolution=/tmp/ccbaDZBp.res -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc\_s -plugin-opt=-pass-through=-lc -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc\_s --build-id --eh-frame-hdr -m elf\_x86\_64 --hash-style=gnu --as-needed -dynamic-linker /lib64/ld-linux-x86-64.so.2 -z relro /usr/lib/gcc/x86\_64-linux-gnu/11/../../../x86\_64-linux-gnu/crt1.o /usr/lib/gcc/x86\_64-linux-gnu/11/../../../x86\_64-linux-gnu/crti.o /usr/lib/gcc/x86\_64-linux-gnu/11/crtbegin.o -L/usr/lib/gcc/x86\_64-linux-gnu/11 -L/usr/lib/gcc/x86\_64-linux-gnu/11/../../../x86\_64-linux-gnu -L/usr/lib/gcc/x86\_64-linux-gnu/11/../../../../lib -L/lib/x86\_64-linux-gnu -L/lib/../lib -L/usr/lib/x86\_64-linux-gnu -L/usr/lib/../lib -L/usr/lib/gcc/x86\_64-linux-gnu/11/../../.. hello.o -lgcc --push-state --as-needed -lgcc\_s --pop-state -lc -lgcc --push-state --as-needed -lgcc\_s --pop-state /usr/lib/gcc/x86\_64-linux-gnu/11/crtend.o /usr/lib/gcc/x86\_64-linux-gnu/11/../../../x86\_64-linux-gnu/crtn.o -o hello

链接过程如图：



图5.2 在Ubuntu下链接的过程

## 5.3 可执行目标文件hello的格式

使用readelf -a hello.o命令，可在终端查看hello的ELF格式。也可使用readelf -a hello > hello\_exe\_elf.txt将结果重定向至文本文件hello\_exe\_elf.txt中，便于查看。

下图为hello的ELF头(左)与hello.o的ELF头(右)：

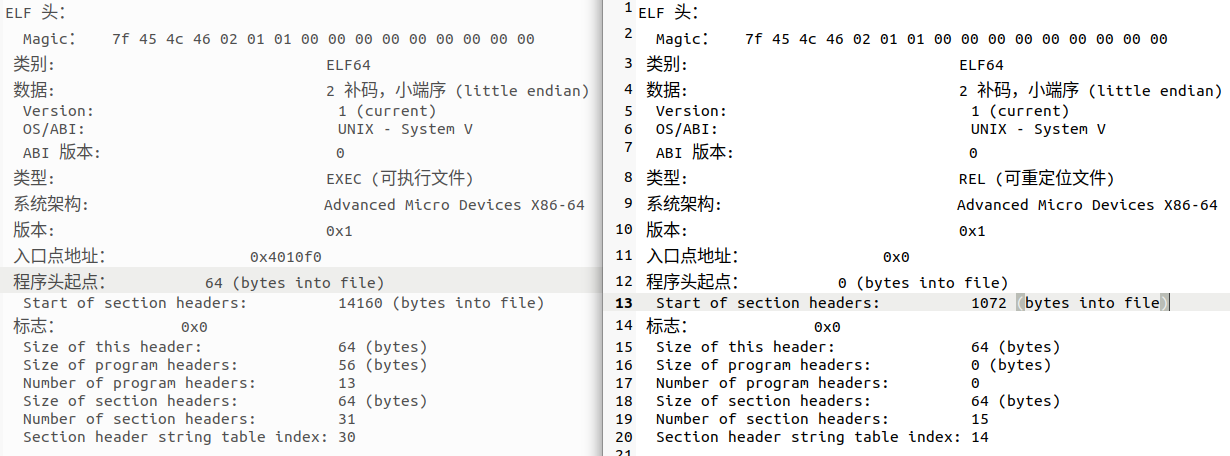


图5.3-1 hello与hello.o的ELF头对比

通过查看hello\_exe\_elf.txt中的节头表，可以看出各段的基本信息。第一列按地址顺序列出了各段的名称及大小，第三列有着各段的起始地址。

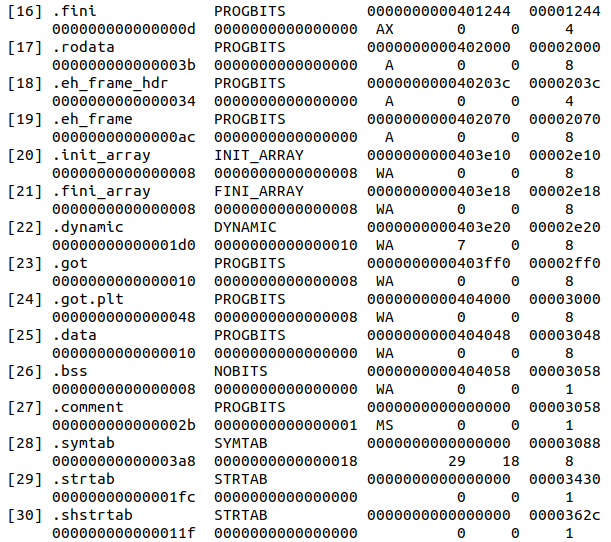
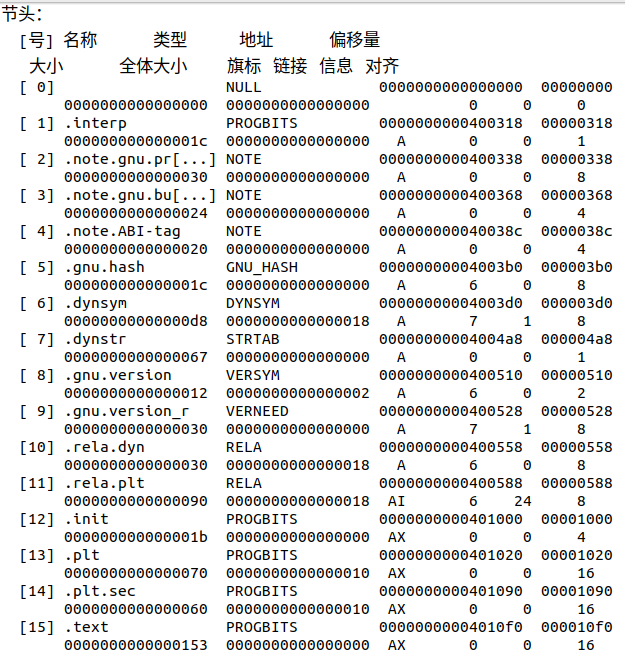


图5.3-2 hello的节头表

## 5.4 hello的虚拟地址空间

使用edb --run hello命令在edb中加载hello。

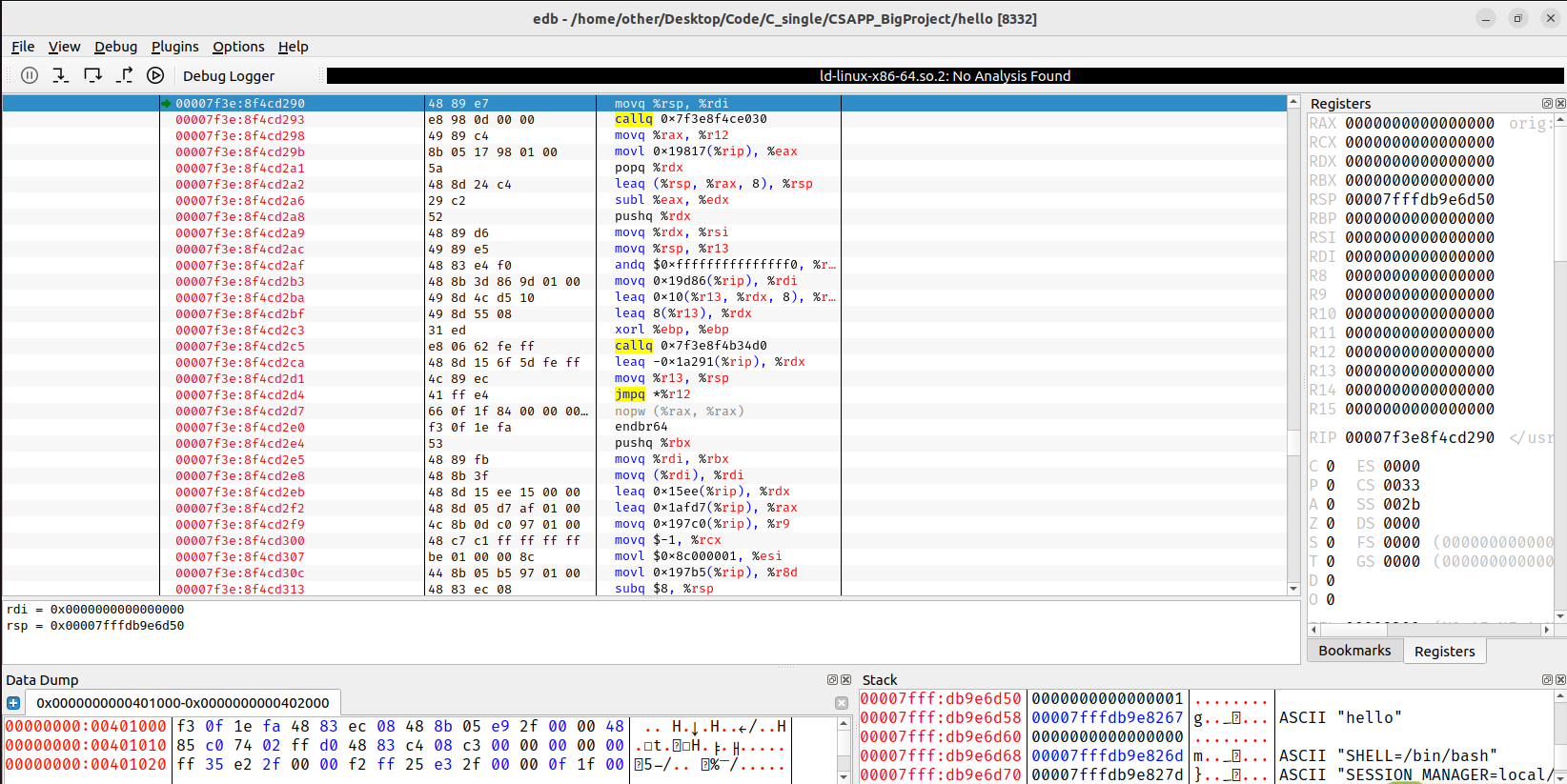


图5.4-1 在edb中加载hello

通过edb的Memory Regions可以查看每块内存区域的读写权限：

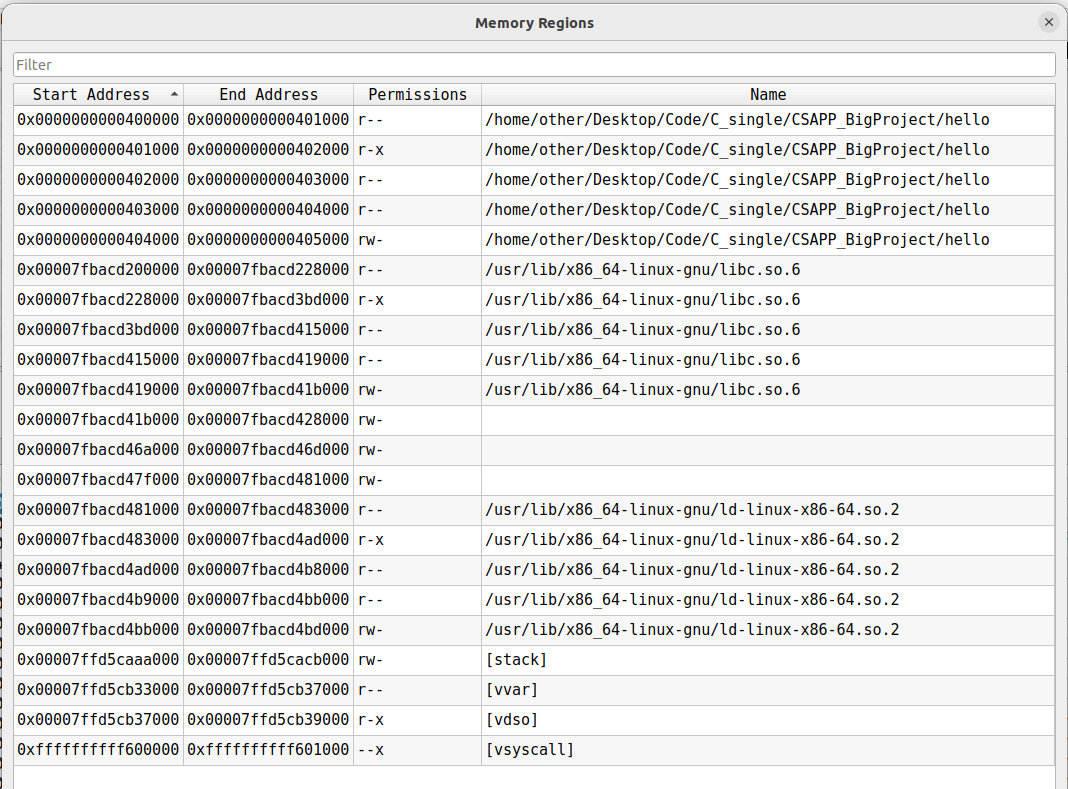


图5.4-2 edb中的Memory Regions

根据5.3中的节头表的内容，我们可以在edb的Data Dump中跳转至对应地址，从而查看某段的原始数据。例如，下图在edb中对 .dynstr段进行了查看：

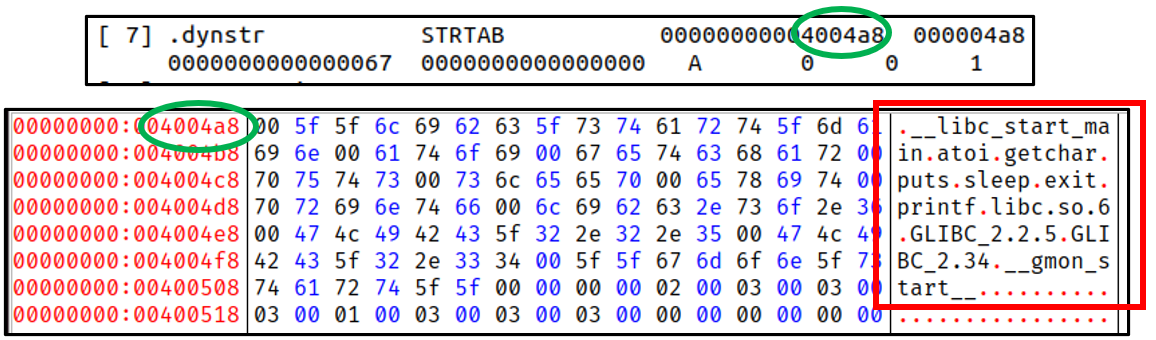


图5.4-3 在edb中查看.dynstr段

再如，下图在edb中对 .rodata段进行了查看：

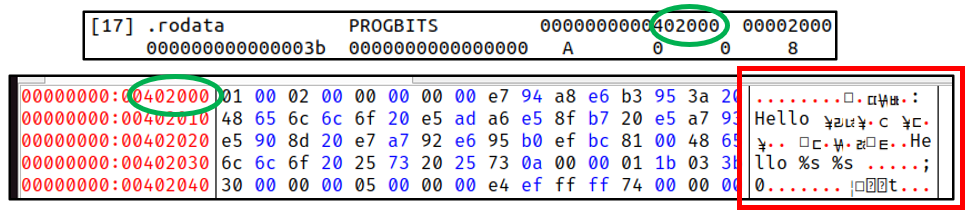


图5.4-4 在edb中查看.rodata段

## 5.5 链接的重定位过程分析

使用objdump -d -r hello命令可以在终端查看hello对应的反汇编代码。也可使用objdump -d -r hello > hello\_exe\_asm.txt将结果重定向至文本文件hello\_exe\_asm.txt中，便于查看。

**5.5.1 不同之处**

经过hello\_exe\_asm.txt与hello\_asm.txt二者之间的对比，发现了以下的不同：

1、文件行数不同

hello\_asm.txt中仅有main函数的相关内容，为46行。hello\_exe\_asm.txt还多出了许多不同，总共有199行。

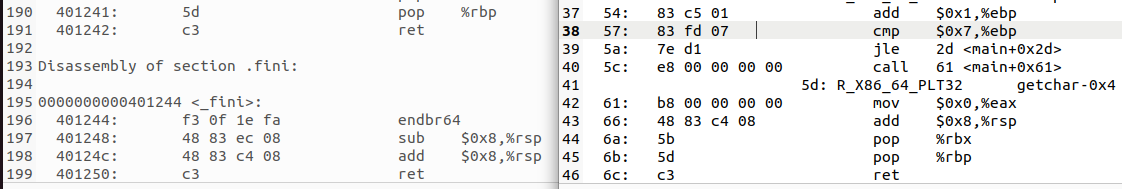


图5.5.1-1 文件行数的对比

2、指令地址不同

hello\_asm.txt中的指令地址是从0开始的，而hello\_exe\_asm.txt中，每行指令都被分配了相应的虚拟地址。

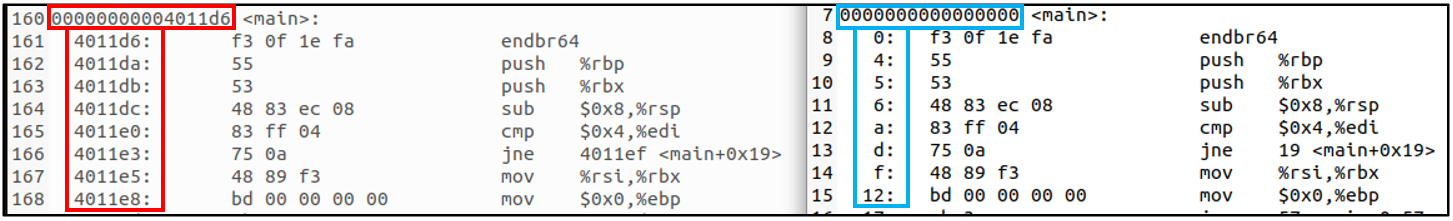


图5.5.1-2 指令地址的对比

3、插入了用于调用共享库中的函数的指令

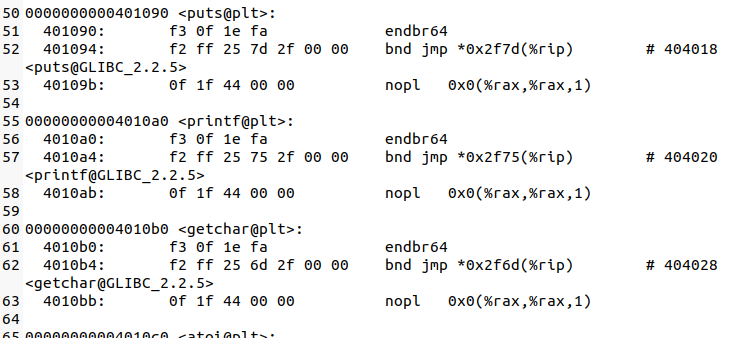


图5.5.1-3 插入了用于调用共享库中的函数的指令

4、字符串常量的引用不同

在hello\_asm.txt中，尚未分配虚拟地址，对字符串常量的引用都用0进行代替。而在hello\_exe\_asm.txt中，由于虚拟地址已经分配，对字符串常量的引用直接使用字符串常量虚拟地址。

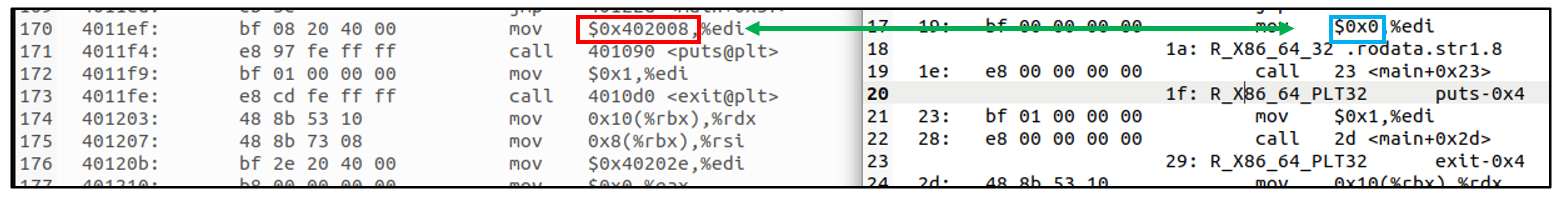


图5.5.1-4 字符串常量的引用不同

5、控制转移的不同

在hello\_asm.txt中，尚未分配虚拟地址，对指令地址的引用都使用相对偏移进行代替。而在hello\_exe\_asm.txt中，由于虚拟地址已经分配，对指令地址的引用直接使用字符串常量虚拟地址。表现在控制转移中就是跳转指令之后的参数变为了虚拟地址。(机器码中为相对寻址)

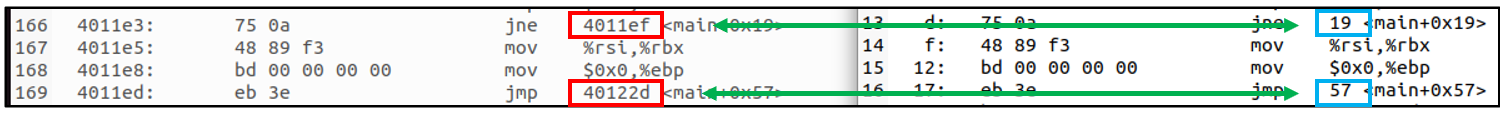


图5.5.1-5 控制转移的不同

6、函数调用的不同

在hello\_asm.txt中，尚未分配虚拟地址，对函数的引用都使用0进行代替。而在hello\_exe\_asm.txt中，由于虚拟地址已经分配，使用函数对应的虚拟地址对函数进行引用。(机器码中为相对寻址)

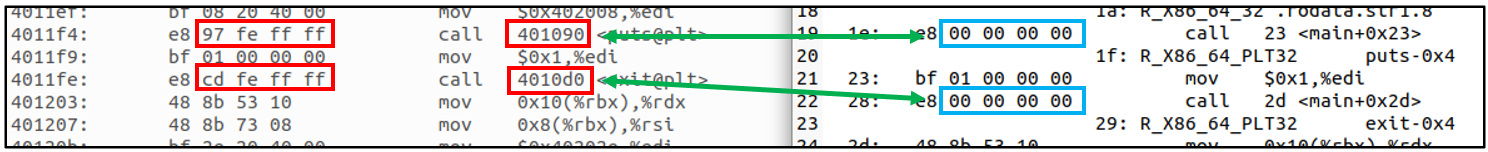


图5.5.1-6 函数调用的不同

**5.5.2 链接过程**

链接的过程主要分为符号解析和重定位这两步：

1、符号解析。链接器解析符号引用的方法就是将每个引用与它对应的可重定位目标文件的符号表中的一个确定的符号定义关联起来。

对于局部符号及静态局部变量解析比较简单：只需要保证每个模块中的每个局部符号只有一个定义。对于全局符号的解析：当编译器遇到一个不是在当前模块定义的符号时，会假设该符号时在其他某个模块中定义的，生成一个链接器符号表条目，并交给链接器处理；如果链接器在任何输入的模块中都找不到该定义就报错并且终止。

2、重定位。重定位将每个符号引用和符号定义关联起来，并且为每个符号分配运行时地址。

重定位包括：重定位节和符号定义：链接器将所有相同类型的数据节合并为同一类型的聚合节，并且将运行时的内存地址赋值给新的聚合节及每个模块定义的符号；重定位节中的符号引用：链接器修改代码节和数据节中对每个符号的引用，使得其指向正确的运行地址。

**5.5.3 重定位过程分析**

由于在hello.s中仅出现了R\_X86\_64\_32与R\_X86\_64\_PLT32两种重定位类型，故接下来，将借由具体例子，分析这两种重定位类型的重定位过程。

1、重定位R\_X86\_64\_32

重定位绝对引用是相当简单的，只要确定了该符号的虚拟地址，那么，对该符号的引用就是其虚拟地址。如图，该字符串的虚拟地址为0x402008，那么直接将先前填充的0改为0x402008即可，以小端序填充。

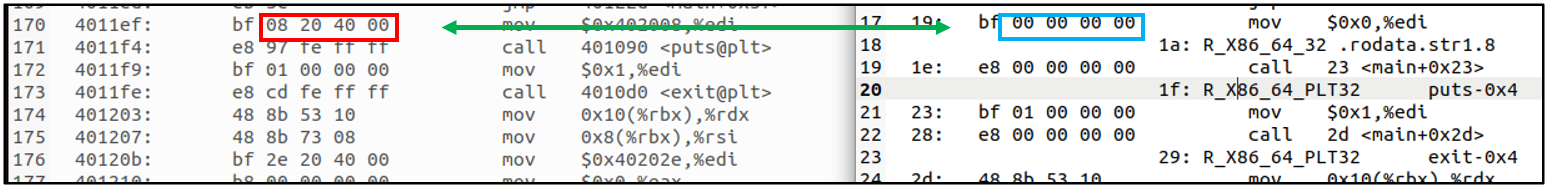


图5.5.3-1 R\_X86\_64\_32的重定位

2、重定位R\_X86\_64\_PLT32

虽然R\_X86\_64\_PLT32与R\_X86\_64\_PC32有很多不同之处，但在call的相对偏移计算上二者是一致的。

计算式为 (unsigned) (ADDR(r.symbol) + r.addend - refaddr)

其中r.symbol = 0x401090, r.addend = -4, refaddr = 0x4011f5

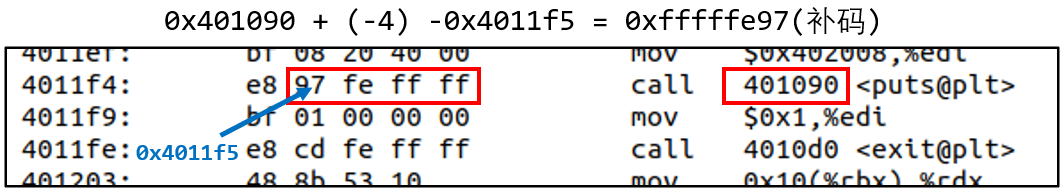


图5.5.3-1 R\_X86\_64\_PLT32的重定位

## 5.6 hello的执行流程

hello!\_start：0x4010f0

libc.so.6!\_\_libc\_start\_main：0x7f649f829dc0

libc.so.6!\_\_cxa\_atexit：0x7f649f8458c0

hello!\_init：0x401000

hello!frame\_dummy：0x4011d0

hello!register\_tm\_clones：0x401160

hello!main：0x4011d6

未正确输入命令行参数：

hello!puts@plt：0x401030

hello!exit@plt：0x401060

libc.so.6!exit：0x7f46864455f0

正确输入命令行参数：

hello!\_\_printf\_chk@plt：0x401050

hello!strtol@plt：0x401040

hello!sleep@plt：0x401070

(重复若干次)

hello!getc@plt：0x401080

libc.so.6!exit：0x7f46864455f0

## 5.7 Hello的动态链接分析

我们知道，动态库是在进程启动的时候加载进来的。加载后，动态链接器需要对其作一系列的初始化，如重定位，这些工作是比较费时的，特别是对函数的重定位。那么我们能不能把对函数的重定位延迟进行呢？这个改进是很有意义的，毕竟很多时候，一个动态库里可能包含很多的全局函数，但是我们往往可能只用到了其中一小部分而已，而且在这用到的一小部分里，很可能其中有些还压根不会执行到，因此完全没必要把那些没用到的函数也过早进行重定位。具体来说，就是应该等到第一次发生对该函数的调用时才进行符号绑定。而这就是所谓的延迟绑定。

通过观察.got.plt节的变化，就能观察到动态链接的过程。

通过readelf找到.got.plt节在地址为0x404000的地方开始，大小为0x48。因此，结束地址为0x40400047，这两个地址之间部分便是.got.plt的内容。



图5.7-1 .got.plt节信息

在edb的Data Dump中找到该地址的内容，观察发现，在dl\_init前后.got.plt节发生了变化。这些变化的内容分别对应.got[1]和.got[2]的位置。其中，.got[1]包括动态链接器在解析函数地址时使用的信息，而.got[2]则是动态链接器ld-linux.so模块中的入口点。

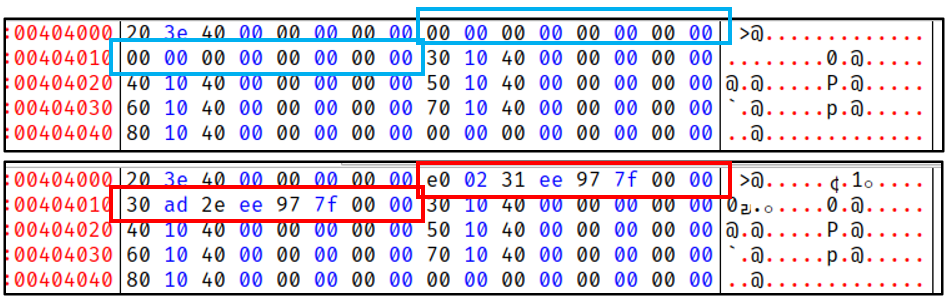


图5.7-2 变化的.got.plt节内容

当程序需要调用一个动态链接库内定义的函数时(例如printf) ,call指令并没有让控制流直接跳转到对应的函数中去，由于延迟绑定的机制，还不知道printf的确切位置。取而代之的是，控制流会跳转到该函数对应的plt表中，然后通过plt表将当前将要调用的函数的序号压入栈中。接下来，调用动态链接器。动态链接器会根据栈中的信息忠实的执行重定位，将真实的printf的运行时地址写入got表，取代了got原先用来跳转到plt的地址，变为了真正的函数地址。

## 5.8 本章小结

本章主要介绍了链接的概念及作用，其主要指是将各种代码和数据片段收集并组合为一个单一文件的过程。

之后，在Ubuntu下将hello.o文件经由链接器生成了可执行目标文件hello。

随后，通过readelf列出了其各节的基本信息，包括起始位置、大小等信息。

然后，用edb查看了hello的虚拟地址空间，同时查看了各节的起始位置与大小。

尔后，通过objdump对hello进行反汇编，得到了其反汇编程序hello\_exe\_asm.txt，并与hello.o的反汇编程序hello \_asm.txt进行了多方面的比较。

之后，分析了hello的重定位过程与执行过程。

最后，通过edb，分析了hello程序的动态链接项目在dl\_init前后的内容变化。

# 第6章 hello进程管理

## 6.1 进程的概念与作用

**6.1.1 进程的概念**

进程是一个执行中程序的实例，是一段程序的执行过程。

**6.1.2 进程的作用**

进程提供了独立的逻辑控制流，好像我们的程序是独占地使用着处理器；也提供一个私有的地址空间，好像我们的程序独占地使用着内存系统。通过进程逻辑控制流，可以使CPU被科学有效地划分成多个部分以并行地运行多个进程。

## 6.2 简述壳Shell-bash的作用与处理流程

**6.1.1 Shell的作用**

Shell是一个交互型应用级程序，可代表用户运行其他程序。

Shell最重要的功能是命令解释，从这种意义上说，Shell是一个命令解释器。Linux系统上的所有可执行文件都可以作为Shell命令来执行。

**6.1.2 Shell的处理流程**

当用户提交了一个命令后，Shell首先判断它是否为内置命令，如果是就通过Shell内部的解释器将其解释为系统功能调用并转交给内核执行；若是外部命令或应用程序就试图在硬盘中查找该命令并将其调入内存，再将其解释为系统功能调用并转交给内核执行。

执行外部命令时，Shell创建会通过fork创建一个子进程，并通过execve加载并运行该外部命令(可执行目标文件)，当该进程执行结束时在信号处理子程序中用waitpid命令对其进行回收，从内核中将其删除。

当执行的进程为前台进程时，Shell会阻塞命令的输入直到前台进程终止运行。

## 6.3 Hello的fork进程创建过程

hello的执行是通过在终端中输入命令 ./hello来完成的。

当我们在终端中输入命令 ./hello时，Shell会先判断发现这个参数并不是内置的命令，从而把这条命令当作一个可执行程序的名字尝试执行。

接下来，Shell会执行fork函数，创建一个子进程。我们的hello将会在这个进程中执行。

fork函数的作用是创建一个与当前进程平行运行的子进程。内核会将父进程的上下文，包括代码，数据段，堆，共享库以及用户栈，甚至于父进程打开的文件的描述符，都创建一份副本。然后利用这个副本执行子进程。从这个角度上来说，子进程与父进程直到执行完fork的瞬间都是完全相同的。

## 6.4 Hello的execve过程

在父进程执行fork函数后，父进程将继续运行Shell的程序，而子进程将通过execve加载用户输入的程序，即我们的hello。由于hello是前台运行的，所以Shell会阻塞命令输入，等待hello运行结束。

execve函数加载并运行可执行目标文件。只有当出现错误时，execve才会返回到调用程序，否则execve调用一次而从不返回。

在execve加载了hello之后，它会调用内核提供的启动代码。内核会将原上下文替换为hello的上下文，然后将控制传递给新程序的程序入口。

值得一提的是，execve只是简单的更换了自己所处进程的上下文，并没有改变进程的pid，也没有改变进程的父子归属关系。

## 6.5 Hello的进程执行

当hello进程创建之时，操作系统会为hello进程分配时间片，让hello进程得以运行。若一个操作系统中运行着多个进程，处理器的一个物理控制流就被分成了多个逻辑控制流，分别交替执行这几个进程。逻辑流的执行是交错的，它们轮流使用处理器，会存在并发执行的现象。其中，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。

hello进程在内存中执行的过程中，并不是一直占用着CPU的资源。因为当内核代表用户执行系统调用时，可能会发生上下文切换，如执行hello中的sleep函数时，或者当操作系统认为hello进程了运行足够久的时候。在这时候，程序将由用户态转换至核心态，内核中的调度器执行上下文切换，将当前的上下文信息保存到内核中，恢复某个先前被抢占的进程的上下文，然后再由核心态转换至用户态，将控制传递给这个先前被抢占的进程。

这样的控制转移将一直存在，直到hello进程运行结束。

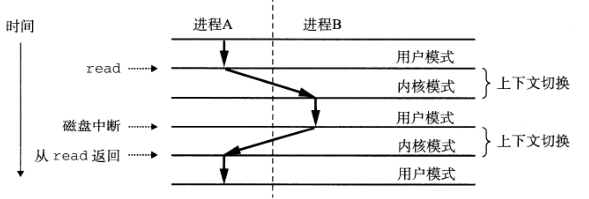


图6.5 进程上下文切换的图解(来自CSAPP原书)

## 6.6 hello的异常与信号处理

6.6.1正常运行

在终端执行./hello 2022113573 张宇杰 1命令，不干扰程序执行，即可完成一次hello的正常运行。

根据上述输入的命令行参数，hello正常运行时，每隔1秒将在屏幕上打印“Hello 2022113573 张宇杰”字样，一共会打印8次。打印结束后，调用getchar()函数阻塞程序执行，等待用户输入。在用户输入回车之后，hello程序终止，Shell回收hello进程，由于不在有前台作业，Shell将等待用户输入下一条命令。



图6.6.1 正常运行的hello

6.6.2随意输入(不包括Ctrl-Z，Ctrl-C)

在hello程序执行时，在键盘进行随意的输入(不包括Ctrl-Z，Ctrl-C)，按下的字符串会直接显示，但不会干扰程序的运行。



图6.6.2 随意乱按时的hello

6.6.3 Ctrl-C

在hello程序执行时，输入Ctrl-C，会中断hello的执行。输入Ctrl-C会发送 SIGINT 信号给Shell，再由Shell将信号转发给前台进程组中的所有进程，终止前台进程组。用ps命令进行查看，找不到hello进程。

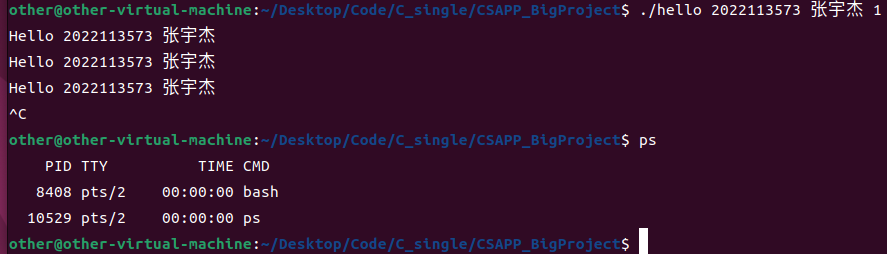


图6.6.3 输入Ctrl+C时的hello

6.6.4 Ctrl-Z

在hello程序执行时，输入Ctrl-Z，会将hello进程挂起。输入Ctrl-Z会发送 SIGTSTP 信号给Shell，再由Shell将信号转发给前台进程组中的所有进程，挂起前台进程组。用ps命令进行查看，会找到被挂起的hello进程，其状态显示为S(休眠)。



图6.6.4-1 输入Ctrl+Z时的hello

在将hello进程挂起后，使用jobs命令可以看到被挂起的hello进程的jid及状态标识。

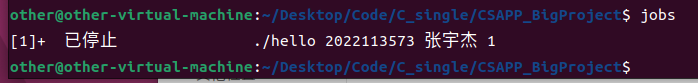


图6.6.4-2 jobs查看hello进程

在将hello进程挂起后，使用pstree命令可以查看hello进程的继承关系。在这里，hello进程的继承路径为systemd→systemd→gnome-terminal-→bash→hello

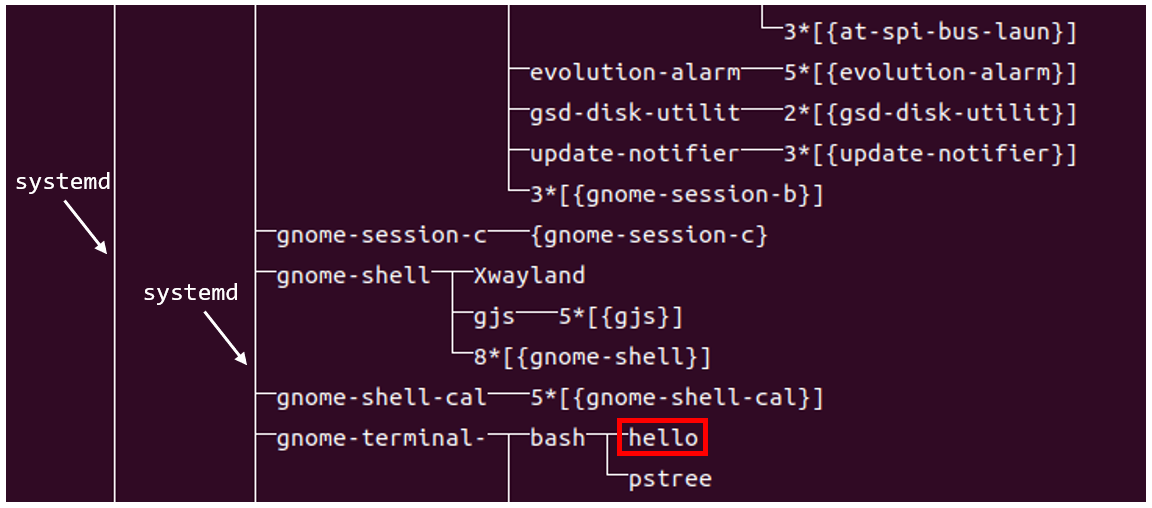


图6.6.4-2 pstree查看进程树

在将hello进程挂起后，使用fg + hello进程对应的jid即可将挂起的hello进程重新回到前台执行，打印剩余内容，并进行正常的程序退出。

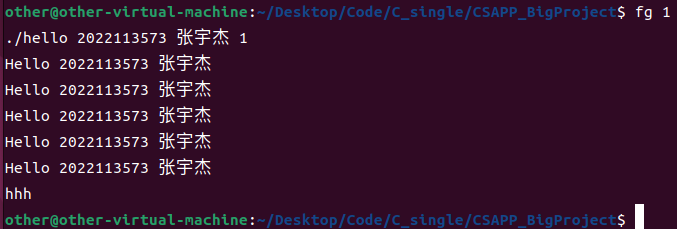


图6.6.4-3 fg命令恢复hello进程的执行

在将hello进程挂起后，使用kill命令可对hello进程发送信号。通过ps查看hello的PID为10683，使用kill -9 10683向hello进程发送SIGKILL信号，将其杀死。之后，再使用ps命令就看不到hello进程了。

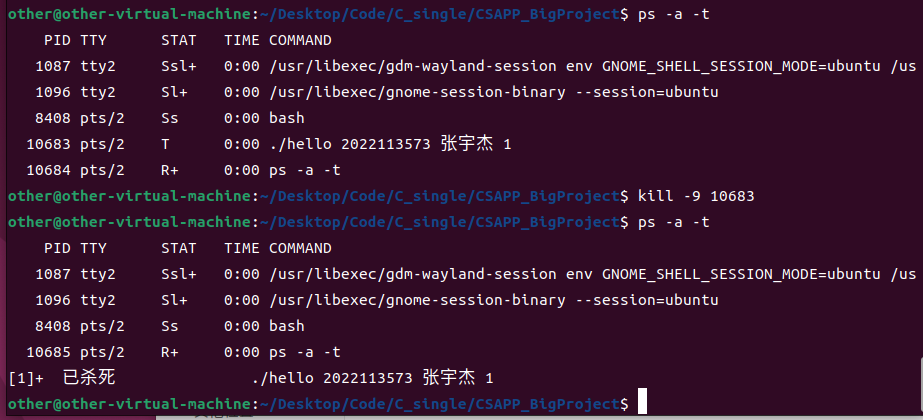


图6.6.4-4 kill命令向hello进程发送SIGKILL信号

## 6.7本章小结

本章介绍了进程的概念和作用，同时简述了Shell的作用及执行流程。

之后，解析了hello的fork过程与execve过程，通过调用fork()函数与execve()来实现。同时，结合了进程上下文信息、进程时间片、用户态与核心态转换等内容，介绍了hello的进程执行流程。

最后，通过分析了在hello执行过程中不停乱按，Ctrl-Z，Ctrl-C，在Ctrl-Z后运行ps、jobs、pstree、fg、kill等命令所造成的现象，说明了hello中异常与信号的处理。

# 第7章 hello的存储管理

## 7.1 hello的存储器地址空间

7.1.1 逻辑地址

逻辑地址是用户编程时使用的与段有关的偏移地址，分为段基址和段偏移量两部分，这是程序员可以见到的地址。

例如，在hello\_exe\_asm.txt中出现的mov $0x402008, %edi中的地址$0x402008即为逻辑地址，需要加上相应的DS数据段基址才能得到对应的线性地址。

7.1.2 线性地址

指虚拟地址到物理地址变换的中间层，是处理器可寻址的内存空间(称为线性地址空间)中的地址。hello\_exe\_asm.txt中出现的地址是逻辑地址，加上相应段基址就成了一个线性地址。

在Linux中，所有的段(用户代码段、用户数据段、内核代码段、内核数据段)的线性地址都是从0开始，长度为4G，这样 线性地址 = 逻辑地址 + 0，也就是说，逻辑地址在数值上等同于线性地址了。

7.1.3 虚拟地址

虚拟地址是指由程序产生的由段选择符和段内偏移地址组成的地址。经过CPU页部件转换成具体的物理地址，进而通过地址总线访问内存。在Linux中，虚拟地址在数值上等同于线性地址。

7.1.4 物理地址

主存被组织成一个由M个连续的字节大小的单元组成的数组，其中每个字节都被赋予了一个唯一的物理地址。进程在运行时指令的执行和数据的访问最后都要通过将虚拟地址转换为物理地址来对主存进行存取。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

逻辑地址分为段选择符 / 段基址和段偏移量两部分

在保护模式下，段选择符并不直接指向段基址段选择符，而是指向段描述符表中定义段的段描述符。

段选择符的3个字段分别是：请求特权级RPL(Requested Privilege Level)，表指示标志TI(Table Index)，与索引值(Index)。

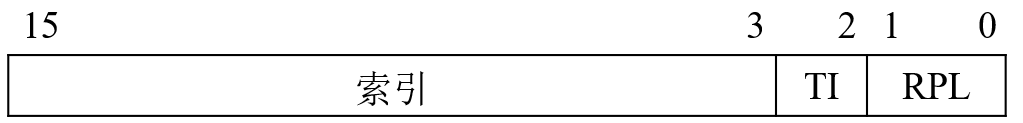


图7.2-1 段选择符的结构

根据段选择符，首先根据TI判断应该选择全局描述符表还是局部描述符表，从GDT与LDT所对应的寄存器GTDR和LDTR获取GDT与LDT的首地址，将段选择符的索引字段的值乘8，加上GDT或LDT的首地址，就能得到当前段描述符的地址。

得到段描述符的地址后，可以通过段描述符中BASE字段获得段的基地址。将其与段偏移量相加，即可得到线性地址。

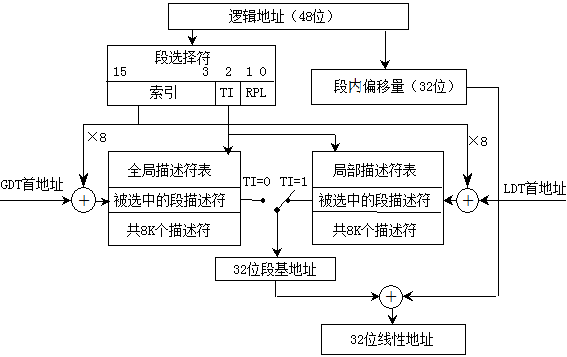


图7.2-2 线性地址的求解流程

## 7.3 Hello的线性地址到物理地址的变换-页式管理

线性地址(虚拟地址)由虚拟页号VPN和虚拟页偏移VPO组成。

由线性地址到物理地址的变换通过以下步骤进行：首先从页表基址寄存器PTBR中，得到hello进程的页表地址。同时根据线性地址前n-p位，即虚拟页号，在页表中找到与之对应的索引项，得到物理页号PPN。最后将物理页号与线性地址中最后p位，即偏移量，将它们相加，就可以得到物理地址。

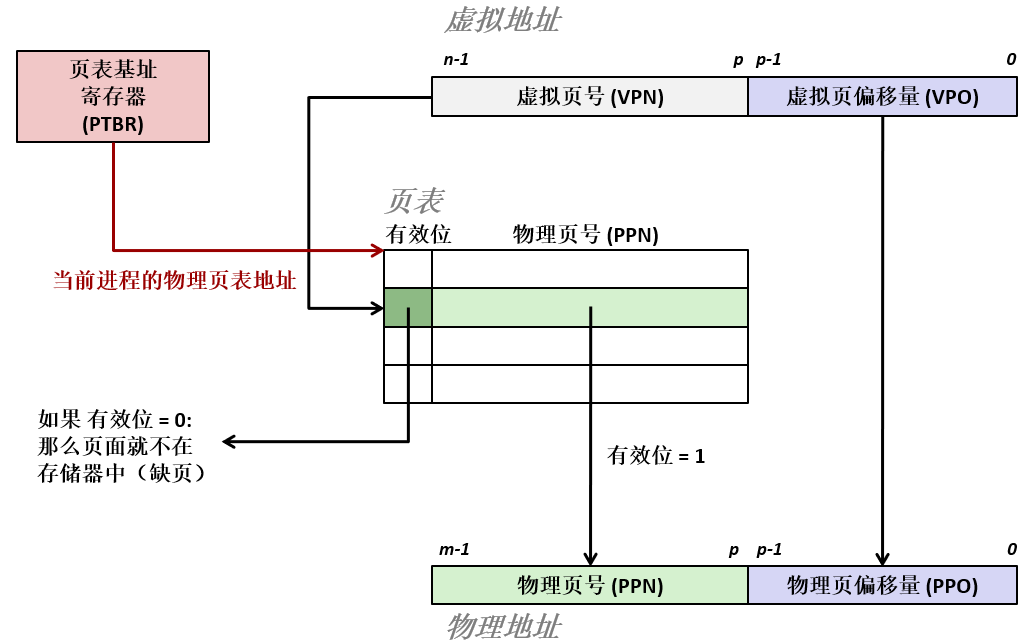


图7.3-1 虚拟地址到物理地址的转换(图来自ppt)

当引用内容时，首先内存管理单元从线性地址中抽取出虚拟页号，检查高速缓存/主存，看它是否缓存于高速缓存/主存中。若命中，将缓存的内容返回给处理器。若不命中，即需要的内容不在物理内存中，则产生缺页中断，需要从虚拟内存所给出对应的磁盘的内容重新加载到物理内存中。

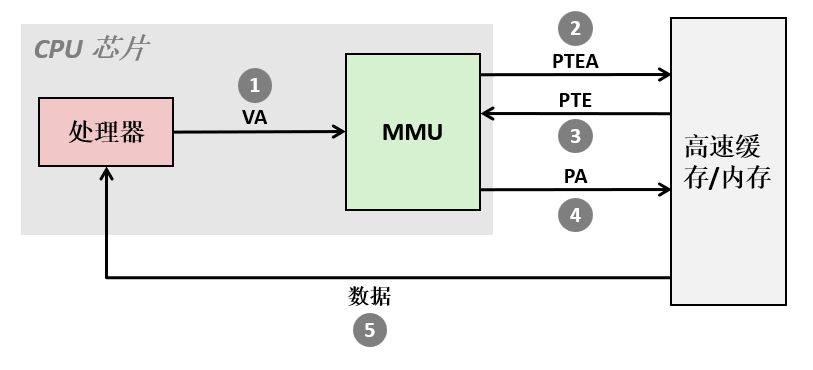


图7.3-2 页面命中的地址翻译流程(图来自ppt)

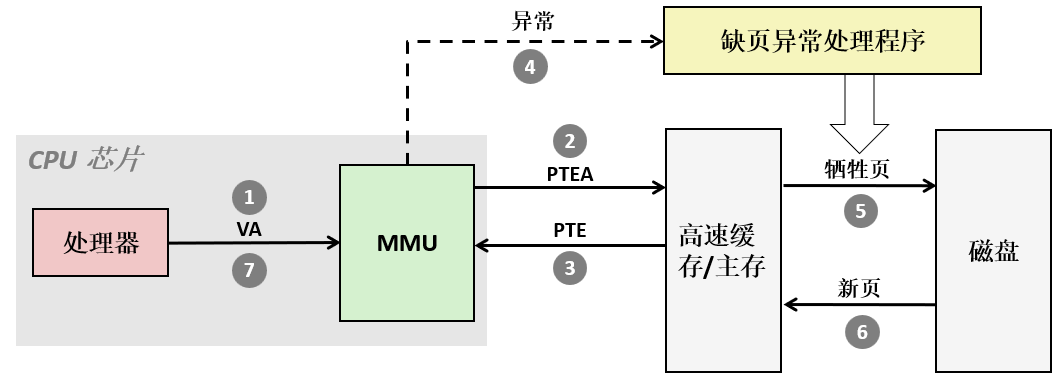


图7.3-3 页面不命中的地址翻译流程(图来自ppt)

## 7.4 TLB与四级页表支持下的VA到PA的变换

为了节约页表的内存存储空间，我们会使用多级页表。虽然多级页表节约了我们的存储空间，但是却存在问题：

原本我们对于只需要进行一次地址转换，只需要访问一次内存就能找到对应的物理页号，算出物理地址。现在我们需要多次访问内存，才能找到对应的物理页号。最终，虽然节约了空间，却带来了时间上的额外开销，变成了一个“以时间换空间”的策略，极大地限制了内存访问性能。

为了解决这种问题导致处理器性能下降的问题，现代 CPU 中都包含了一块缓存芯片TLB，全称为地址变换高速缓冲(Translation Lookaside Buffer)，简称为“快表”，用于加速对于页表的访问。简单来说，TLB就是页表的Cache，属于MMU的一部分，其中存储了当前最可能被访问到的页表项。

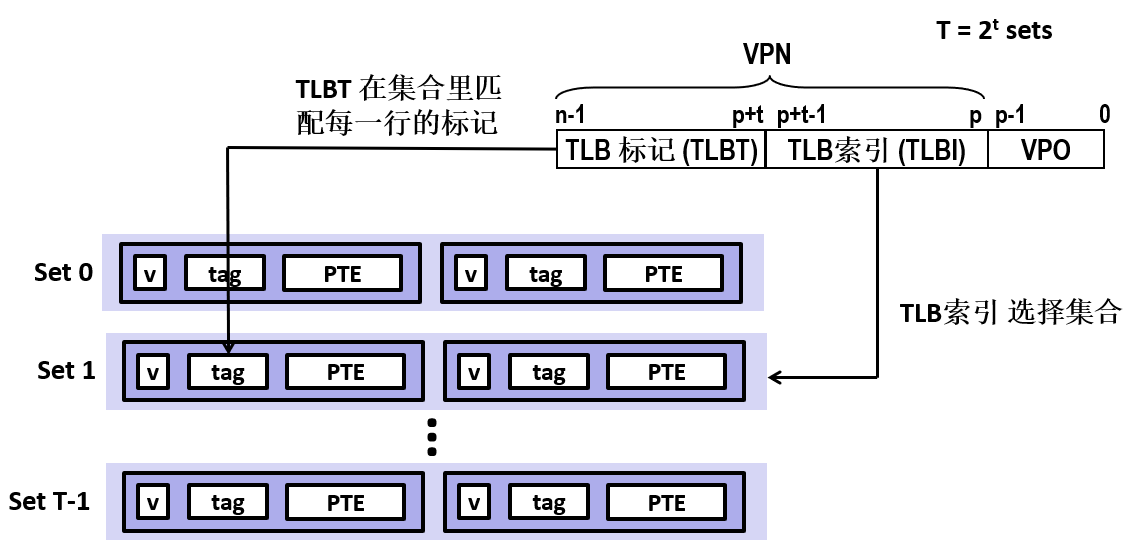


图7.4-1 MMU中访问TLB(图来自ppt)

当CPU处理虚拟地址时，首先去TLB中根据标志Tag寻找页表数据，假如TLB中正好存放所需的页表，说明TLB命中，直接从TLB中获取该虚拟页号对应的物理页号。如果TLB不命中，需要从L1缓存中根据VA取出相应的PTE，计算出PA，并将该PTE存放在TLB中，可能会覆盖原先的条目。

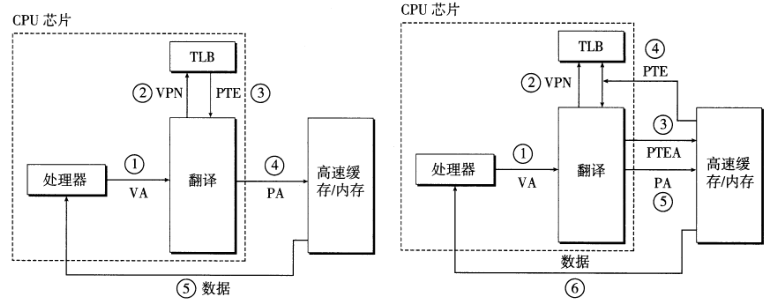


图7.4-2 加入TLB后，通过虚拟内存访问数据的流程(图来自csapp原书)

在四级页表的参与之下，当TLB不命中时，将根据VPN1、VPN2…一层层的计算出下一级页表的索引，最后在L4页表中找到相应的PTE，计算出对应的PA，并将其添加至TLB之中。

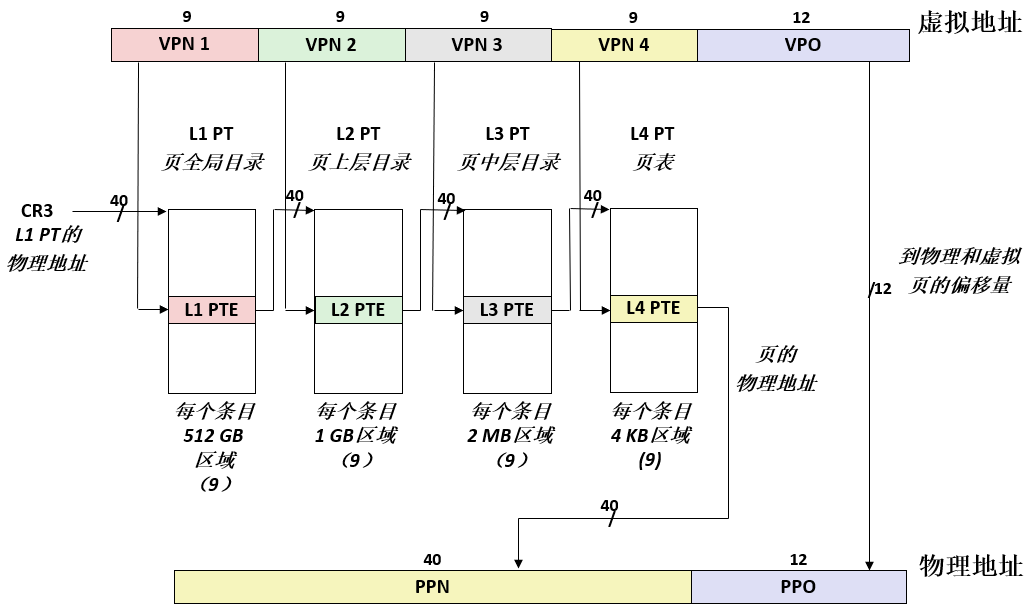


图7.4-2 Core i7的四级页表示意图(图来自ppt)

## 7.5 三级Cache支持下的物理内存访问

在MMU计算出物理地址PA之后，将其发送至L1缓存，缓存从PA中取出标记、组索引信息进行匹配。如果匹配成功，且有效位为1，则Cache命中，根据块偏移取出数据返回给CPU。如果Cache不命中，继续向下一级缓存或主存查询，按照L1-L2-L3-主存的顺序。查找成功后，将数据返回CPU，并将相应的块根据替换策略缓存在当前的Cache中。

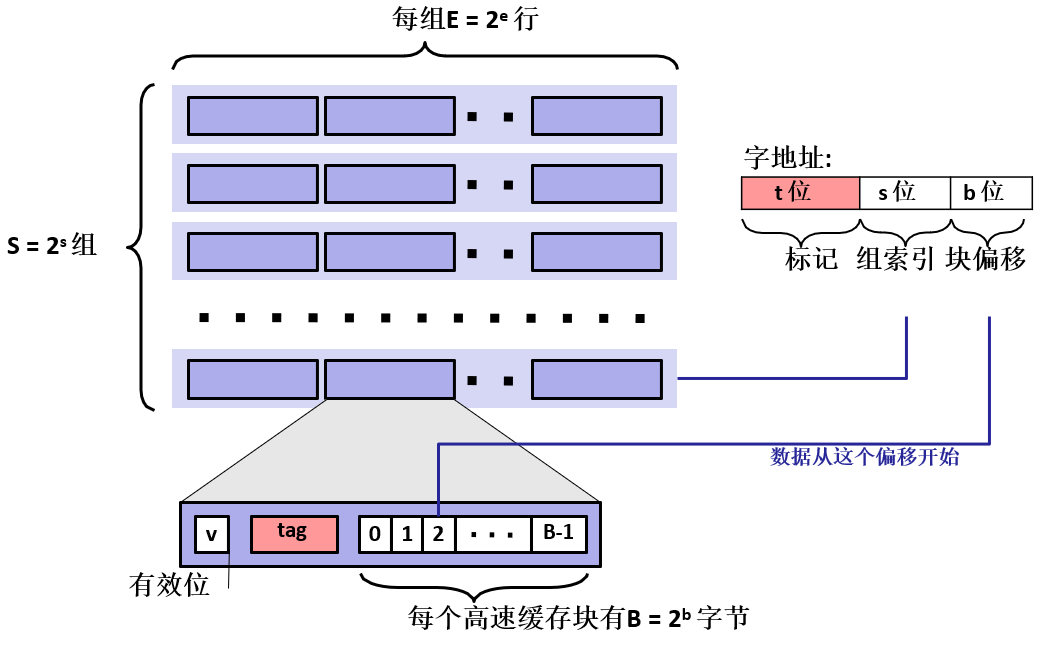


图7.5 读Cache示意图(图来自ppt)

## 7.6 hello进程fork时的内存映射

当Shell调用fork函数创建hello进程时，内核为hello进程创建各种数据结构，并分配给它一个唯一的PID。

为了给hello进程创建虚拟内存，内核创建了当前进程的mm\_struct、区域结构和页表的原样副本，将两个进程中的每个页面都标记为只读，并且把两个进程中的每个区域结构都标记为私有的写时复制。

当hello进程中fork返回时，hello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。这两个进程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，为每个进程保持了私有地址空间的抽象概念。

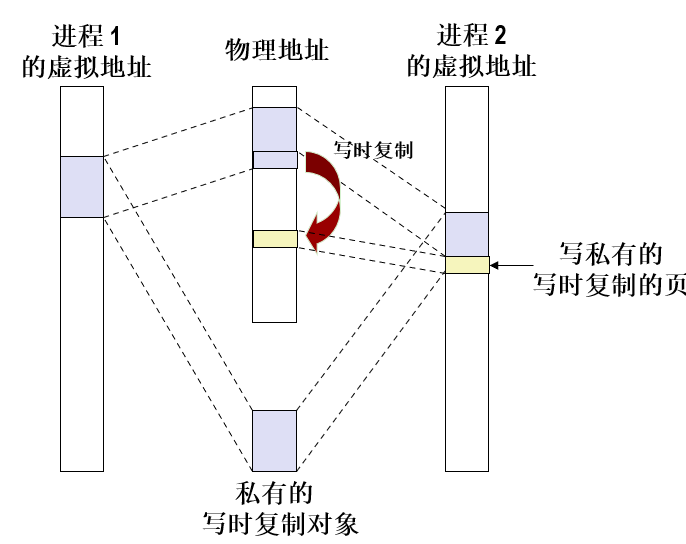


图7.6 写时复制示意图(图来自ppt)

## 7.7 hello进程execve时的内存映射

当Shell在fork出的hello进程中使用execve，在hello进程中加载并运行包含在可执行目标文件hello中的程序时，需要执行以下步骤：

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hello程序的代码、数据、bss和栈区创建新的区域结构。所有这些新的区域都是私有的、写时复制的。

代码和数据区域被映射为 hello文件中的.text和.data区。

bss区域是请求二进制零的，映射到匿名文件，其大小包含在hello中。

栈和堆区域也是请求二进制零的，初始长度为零。

3、映射共享区域

如果hello程序与共享对象(或目标)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

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。

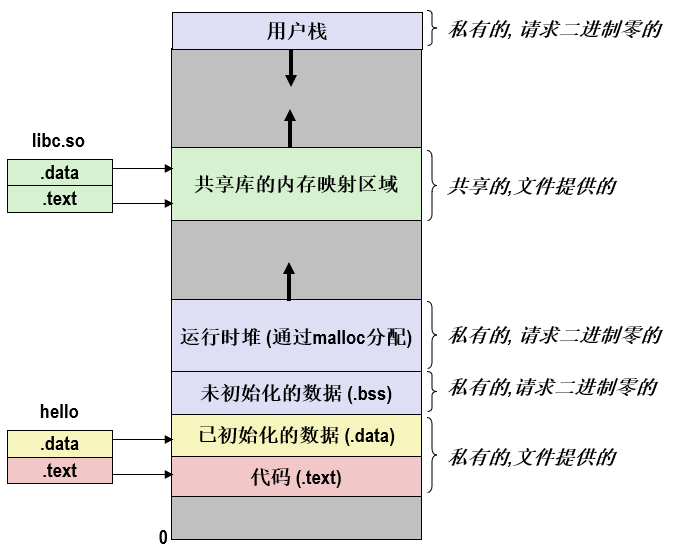


图7.7 execve执行后的内存映射(图来自ppt)

## 7.8 缺页故障与缺页中断处理

当DRAM 缓存不命中时，就发生了缺页。

以下是一个缺页处理的例子。CPU引用了VP3中的一个字，VP3并未缓存在DRAM中。地址翻译硬件从内存中读取PTE3，从有效位推断出VP3未被缓存，并且触发一个缺页异常。

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，在此例中就是存放在PP3中的VP4。如果VP4已经被修改了，那么内核就会将它复制回磁盘。无论哪种情况，内核都会修改VP4的页表条目，反映出VP4不再缓存在主存中这一事实。

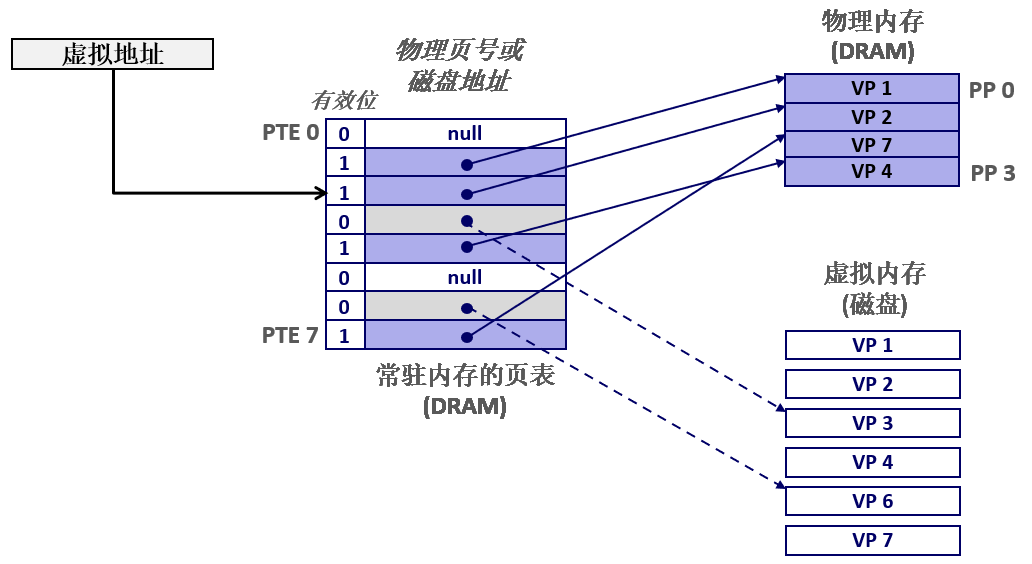


图7.8-1 缺页处理之前(图来自ppt)

接下来，内核从磁盘复制VP3到内存中的PP3，更新PTE3，随后返回。当异常处理程序返回时，它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。但是现在，VP3已经缓存在主存中了，那么页命中也能由地址翻译硬件正常处理了。

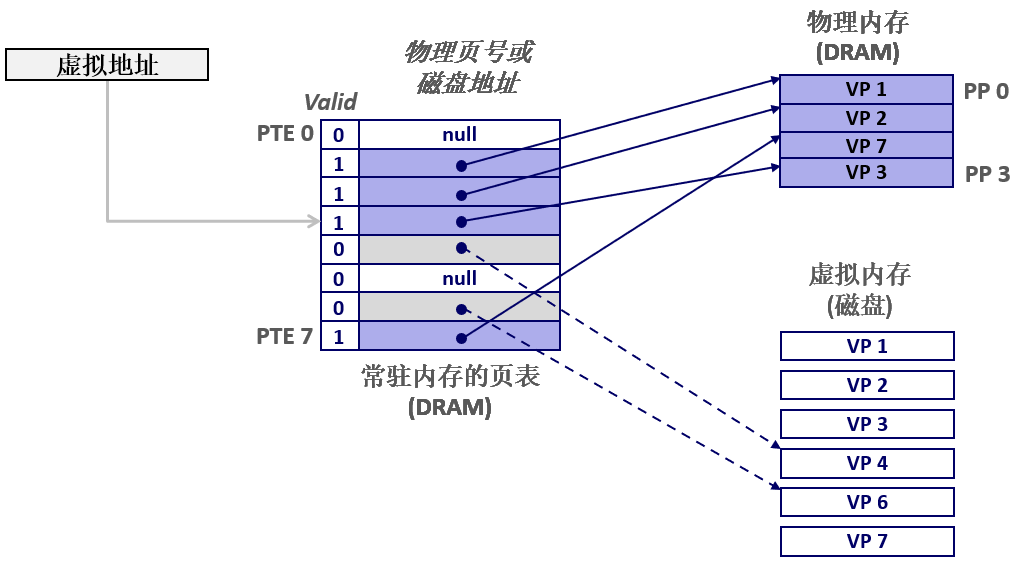


图7.8-2 缺页处理之后(图来自ppt)

## 7.9动态存储分配管理

7.9.1 动态内存分配器的基本原理

动态内存分配器维护着一个进程中称为堆(heap)的虚拟内存区域。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(.bss)后开始，并向上生长。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，空闲块则可用来分配。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，显式分配器与隐式分配器。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。

显式分配器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malloc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运算符与C中的malloc和free相当。

与之相反，隐式分配器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，而自动释放未使用的已分配的块的过程叫做垃圾收集。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。

7.9.2隐式空闲链表分配器原理

在隐式空闲链表的情况中，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(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。

有效载荷后面是一片不使用的填充块，其大小可以是任意的。需要填充有很多原因。比如，填充可能是分配器策略的一部分，用来对付外部碎片。或者也需要用它来满足对齐要求。

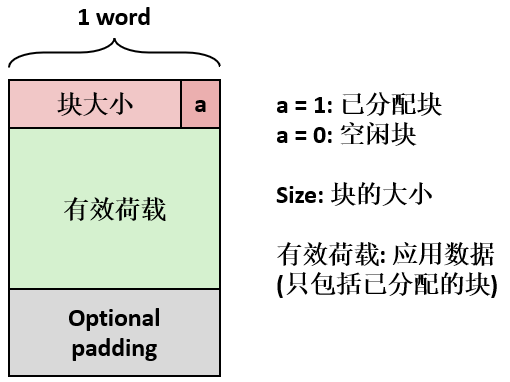


图7.9-1 简单的堆块格式(图来自ppt)

堆被组织为一个连续的已分配块和空闲块的序列。

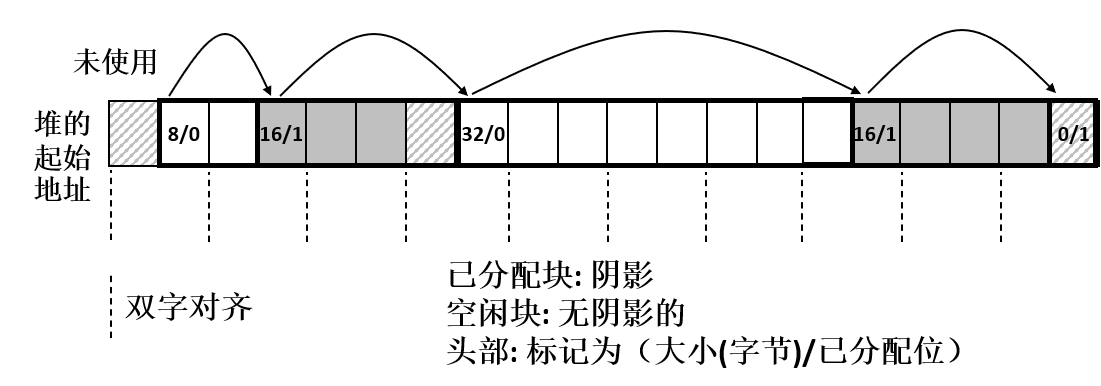


图7.9-2 用隐式空闲链表来组织堆(图来自ppt)

其中，阴影部分是已分配块。没有阴影的部分是空闲块。头部标记为(大小(字节)/ 已分配位)。

称这种结构为隐式空闲链表，是因为空闲块是通过头部中的大小字段隐含地连接着的。分配器可以通过遍历堆中所有的块，从而间接地遍历整个空闲块的集合。注意，我们需要某种特殊标记的结束块，在这个示例中，就是一个设置了已分配位而大小为零的终止头部。

隐式空闲链表的优点是简单。显著的缺点是任何操作的开销，例如放置分配的块，要求对空闲链表进行搜索，该搜索所需时间与堆中已分配块和空闲块的总数呈线性关系。

7.9.3显式空闲链表分配器原理

相比于隐式空闲链表，一种更好的方法是将空闲块组织为某种形式的显式数据结构。例如，堆可以组织成一个双向空闲链表，在每个空闲块中，都包含一个前驱和后继指针。在使用双向空闲链表的情况下，有两种维护链表的方式：

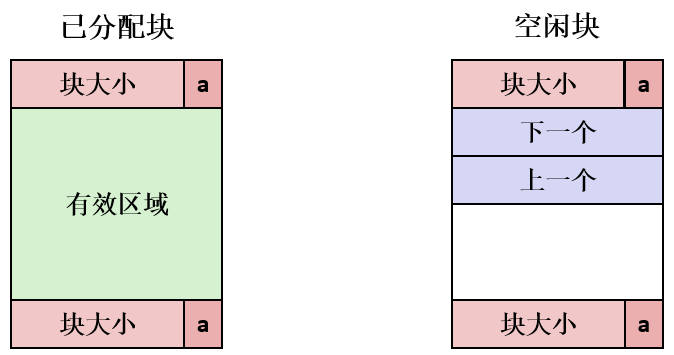


图7.9-3 显式空闲链表中的堆块格式(图来自ppt)

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内存利用率，接近最佳适配的利用率。

一般而言，显式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

本章介绍了hello的存储器地址空间。结合了hello，说明了逻辑地址、线性地址、虚拟地址、物理地址的概念，以及它们的区别与联系，互相转化的方法。

叙述了在段式管理之下，逻辑地址到线性地址(虚拟地址)的变换是如何完成的。

叙述了在页式管理之下，线性地址到物理地址的变换是如何完成的。

分析了TLB与四级页表支持下的VA到PA的变换。以四级页表为例，介绍了多级页表的层次、工作流程以及节省空间的优点。而为了弥补页表速度上的缺点，引入了高速地址变址缓存TLB。

介绍了三级Cache支持下的物理内存访问的流程，之后以hello进程为例，分析了fork与execve时的内存映射。

介绍了缺页故障与缺页中断的处理，并使用一个简单例子，描述了缺页中断的处理流程。

最后，分析了动态存储分配管理。从动态内存管理的基本方法与动态内存管理的策略两个方面对动态内存管理进行介绍。

# 第8章 hello的IO管理

## 8.1 Linux的IO设备管理方法

在Linux中，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，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

8.2.1 简述Unix I/O接口

通过Unix I/O接口，所有的输入和输出都能以统一且一致的方式来执行：

1、打开文件

一个应用程序通过要求内核打开相应的文件，来宣告它想要访问一个I/O设备。内核返回一个小的非负整数，叫做描述符，它在后续对此文件的所有操作中标识这个文件。内核记录有关这个打开文件的所有信息。应用程序只需记住这个描述符。

2、每个进程开始时都打开的三个文件

Linux shell创建的每个进程开始时都有三个打开的文件：标准输入(描述符为0)、标准输出(描述符为1)和标准错误(描述符为2)

3、改变当前的文件位置

对于每个打开的文件，内核保持着一个文件位置k，初始为0。这个文件位置是从文件开头起始的字节偏移量。通过执行seek操作，能够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，执行读操作会触发一个EOF条件。类似地，写操作就是从内存复制n>0个字节到一个文件，从当前文件位置k开始，然后更新k。

5、关闭文件

当应用完成了对文件的访问之后，它就通知内核关闭这个文件。作为响应，内核释放文件打开时创建的数据结构，并将这个描述符恢复到可用的描述符池中。

8.2.2 Unix I/O函数

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

应用程序通过调用open函数来打开一个已存在的文件或者创建一个新文件。

int close(int fd);

应用程序通过调用 close 函数关闭一个打开的文件。

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

应用程序通过调用read函数来执行文件的输入。

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

应用程序通过调用write函数来执行文件的输出。

## 8.3 printf的实现分析

以下是printf函数的定义：

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. }

在printf的形参列表中，可以看到，const char \*fmt之后的参数都用了“...”来代替。这个是可变形参的一种写法，当传递参数的个数不确定时，就可以用这种方式来表示。

在printf的函数体中，有句：

1. **va\_list** arg = (**va\_list**)((**char** \*)(&fmt) + 4);

其中，va\_list定义为：

**typedef** **char** \***va\_list**;

(char\*)(&fmt) + 4) 表示的是“…”中的第一个参数的地址。这是因为，在C语言中，参数压栈的方向是从右往左的。第一个参数fmt将在栈顶的位置，而栈顶是往地址减小的方向增加的。在32位中，第一个参数const char \*fmt的大小为4字节，将fmt的地址加上4后，指针向栈底方向移动，指向“…”中的第一个参数。

之后的下一句：

1. i = vsprintf(buf, fmt, arg);

中，调用了vsprintf函数，其简单实现为：

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. {
9. **if** (\*fmt != '%')
10. {
11. \*p++ = \*fmt;
12. **continue**;
13. }
15. fmt++;
17. **switch** (\*fmt)
18. {
19. **case** 'x':
20. itoa(tmp, \*((**int** \*)p\_next\_arg));
21. strcpy(p, tmp);
22. p\_next\_arg += 4;
23. p += strlen(tmp);
24. **break**;
25. **case** 's':
26. **break**;
27. **default**:
28. **break**;
29. }
30. }
32. **return** (p - buf);
33. }

其执行流程为，扫描格式串fmt，如果没有遇到%(格式占位符)，则将字符原封不动的输出至buf中。如果遇到了%，根据其后面接着的字母来判断需要进行格式化输出的类型，从而解读出p\_next\_arg的真实数据类型，再调用对应的具体函数进行格式化字符串的生成。

例如，格式占位符是%d，则将p\_next\_arg(char\*)解读为int\*类型(强转类型转换)，解引用得到实际的参数int，再调用itoa等函数将int格式化为字符串，输出至buf中。输出完这个参数之后，让p\_next\_arg加上这个参数的大小，使之指向下一个参数。

在获得格式化字符串buf后，printf调用write进行输出：

1. write(buf, i);

其中的i是buf中格式化字符串的长度，由vsprintf返回。

我们看一下write的实现：

1. write:
2. mov eax, \_\_NR\_write
3. mov ebx, [esp + 4]
4. mov ecx, [esp + 8]
5. int INT\_VECTOR\_SYS\_CALL

在write中，给寄存器传递了参数，之后int INT\_VECTOR\_SYS\_CALL，通过系统来调用sys\_call这个函数。

最后，我们看一下sys\_call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

这里的call [sys\_call\_table + eax\*4](调用的是sys\_call\_table[eax])中， sys\_call\_table是一个函数指针数组，每一个成员都指向一个函数，用以处理相应的系统调用。在这个实例中，此时的eax为4(即\_\_NR\_write的系统调用号)，从而对内核中的write进行调用。

接下来，系统已经确定了所要显示在屏幕上的符号。根据每个符号所对应的ASCII码，系统会从字模库中提取出每个符号的VRAM信息。

显卡使用的内存分为两部分，一部分是显卡自带的显存称为VRAM内存，另外一部分是系统主存称为GTT内存。在嵌入式系统或者集成显卡上，显卡通常是不自带显存的，而是完全使用系统内存。通常显卡上的显存访存速度数倍于系统内存，因而许多数据如果是放在显卡自带显存上，其速度将明显高于使用系统内存的情况。

显示芯片按照刷新频率逐行读取VRAM，并通过信号线向液晶显示器传输每一个点(RGB分量)。

## 8.4 getchar的实现分析

进入getchar函数之后，进程会进入阻塞状态，等待外界的输入。系统开始检测键盘的输入。此时如果按下一个键，就会产生一个异步中断，这个中断会使系统回到当前的getchar进程，然后根据按下的按键，转化成对应的ASCII码，保存到系统的键盘缓冲区。

接下来，getchar调用了read函数。read函数会产生一个陷阱，通过系统调用，读取键盘缓冲区中存储的刚刚按下的按键信息，然后返回指定大小的字符串。

最后，getchar会将这个字符串保存在一个静态的缓冲区中，并返回其第一个字符。在下次调用getchar时，将直接从静态的缓冲区中取出字符并返回，而不是通过read再次进行读取，直到静态缓冲区为空，才再调用read进行读取。

## 8.5本章小结

本章介绍了linux系统下的IO的基本知识，讨论了Linux系统中Unix I/O的形式以及实现的模式函数。最后，对printf和getchar两个函数的实现进行了深入的探究。

# 结论

我们通过键盘，向计算机输入一行行代码，这串代码组合成了一个C源文件，也就是我们的主角hello.c。

接下来，hello.c经过了预处理器cpp，编译器cc1，汇编器as，链接器ld这些家伙一顿好生折腾，最终生成一个可以加载到内存执行的可执行目标文件hello。

然后，我们在Shell中执行命令“./hello 2022113573 张宇杰 1”，Shell通过fork函数创建一个新的进程，然后在子进程里通过execve函数将hello程序加载到内存。虚拟内存机制通过mmap为hello进程规划了一片虚拟空间，调度器为hello进程规划进程执行的时间片，使其能够与其他进程一起合理利用CPU与内存的资源。hello完成了其P2P(From Program to Process)的过程。

之后，CPU一条条的从hello的.text段取指令，寄存器们的值随着程序的执行而不断变化着，异常处理程序监视着键盘的输入。hello中的syscall系统调用会使进程触发陷阱，让内核接手进程，执行write函数，将一串字符传递给屏幕IO的映射文件。

映射文件对传入数据进行分析，读取VRAM，然后在屏幕上将字符显示出一行行字符串“Hello 2022113573 张宇杰”。

最后，hello程序运行结束，Shell通过waitpid函数通知内核回收hello进程，hello进程消失。至此，hello完成了其程序执行的一生，从不带来什么，也不带走什么，是真正的O2O(From Zero to Zero)。

hello的一生结束了，而我们的计算机之路才刚刚开始。了解完hello的一生之后，相信我们学到了许多的知识，相信我们也产生了更多的疑惑。虽然hello的一生结束了，但执行它的CPU还在不停运转，内存中的比特海洋仍在波涛汹涌。这些疑惑，就让我们用一生来解开吧！

# 附件

|  |  |
| --- | --- |
| hello.c | C源文件 |
| hello.i | C预处理文件，由hello.c预处理得到 |
| hello.s | 汇编语言文件，由hello.i编译得到 |
| hello.o | 可重定位目标文件，由hello.s汇编得到 |
| hello\_elf.txt | 由readelf生成的关于hello.o的ELF信息 |
| hello\_asm.txt | 由objdump生成的关于hello.o的反汇编信息 |
| hello | 可执行文件，由hello.o链接得到 |
| hello\_exe\_elf.txt | 由readelf生成的关于hello的ELF信息 |
| hello\_exe\_asm.txt | 由objdump生成的关于hello的反汇编信息 |

# 参考文献

1. 《深入理解计算机系统》第3版
2. 老师的PPT
3. C 预处理器

<https://www.runoob.com/cprogramming/c-preprocessors.html>

1. 程序详细编译过程(预处理、编译、汇编、链接)

<https://zhuanlan.zhihu.com/p/476697014>

1. bss、data和rodata区别与联系

<https://blog.csdn.net/laiqun_ai/article/details/8528366>

1. 64位ELF文件头格式介绍

<https://www.jianshu.com/p/53751f749c4b>

1. ELF 文件解析 1-前述+文件头分析

<https://zhuanlan.zhihu.com/p/380908650>

1. ELF 文件解析 2-节

<https://zhuanlan.zhihu.com/p/386573979>

1. 程序的链接

<https://www.cnblogs.com/shuqin/p/12012906.html>

1. elf(5) — Linux manual page

<https://www.man7.org/linux/man-pages/man5/elf.5.html>

1. 虚拟地址、逻辑地址、线性地址、物理地址的区别

<https://blog.csdn.net/qiuchaoxi/article/details/79616220>

1. 操作系统-分段机制

<https://www.cnblogs.com/mdumpling/p/8494806.html>

1. 一文读懂内存管理中TLB：地址转换后援缓冲器

<https://zhuanlan.zhihu.com/p/480808324>

1. CPU 与 Memory 内存之间的三级缓存的实现原理

<https://blog.csdn.net/qq_38877888/article/details/103118068>

1. 256-Linux虚拟内存映射和fork的写时拷贝

<https://blog.csdn.net/LINZEYU666/article/details/115420915>

1. [转]printf 函数实现的深入剖析

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