

**计算机系统**

**大作业**

题 目 程序人生-Hello’s P2P

专 业 计算机类

学　　 号 1170500913

班　　 级 1703002

学 生 熊健羽

指 导 教 师 史先俊

**计算机科学与技术学院**

**2018年12月**

**摘 要**

本文主要研究了hello这一简单c程序的整个生命周期。从hello.c源程序为起点，从编译、链接，到加载、运行，再到终止、回收。结合《深入理解计算机系统》一书的内容及计算机系统课上的讲授，在Ubuntu系统下对hello程序的编译、链接、调试、运行等实际操作，顺着hello的生命周期，漫游了整个计算机系统，把计算机系统整个的体系串联在一起。

**关键词：**hello程序；计算机系统；生命周期；

**（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 - 5 -](#_Toc532238400)

[第2章 预处理 - 7 -](#_Toc532238401)

[2.1 预处理的概念与作用 - 7 -](#_Toc532238402)

[2.2在Ubuntu下预处理的命令 - 7 -](#_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 - 21 -](#_Toc532238416)

[第5章 链接 - 22 -](#_Toc532238417)

[5.1 链接的概念与作用 - 22 -](#_Toc532238418)

[5.2 在Ubuntu下链接的命令 - 22 -](#_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 - 25 -](#_Toc532238423)

[5.7 Hello的动态链接分析 - 26 -](#_Toc532238424)

[5.8 本章小结 - 27 -](#_Toc532238425)

[第6章 hello进程管理 - 28 -](#_Toc532238426)

[6.1 进程的概念与作用 - 28 -](#_Toc532238427)

[6.2 简述壳Shell-bash的作用与处理流程 - 28 -](#_Toc532238428)

[6.3 Hello的fork进程创建过程 - 28 -](#_Toc532238429)

[6.4 Hello的execve过程 - 28 -](#_Toc532238430)

[6.5 Hello的进程执行 - 28 -](#_Toc532238431)

[6.6 hello的异常与信号处理 - 30 -](#_Toc532238432)

[6.7本章小结 - 32 -](#_Toc532238433)

[第7章 hello的存储管理 - 33 -](#_Toc532238434)

[7.1 hello的存储器地址空间 - 33 -](#_Toc532238435)

[7.2 Intel逻辑地址到线性地址的变换-段式管理 - 34 -](#_Toc532238436)

[7.3 Hello的线性地址到物理地址的变换-页式管理 - 35 -](#_Toc532238437)

[7.4 TLB与四级页表支持下的VA到PA的变换 - 36 -](#_Toc532238438)

[7.5 三级Cache支持下的物理内存访问 - 37 -](#_Toc532238439)

[7.6 hello进程fork时的内存映射 - 38 -](#_Toc532238440)

[7.7 hello进程execve时的内存映射 - 38 -](#_Toc532238441)

[7.8 缺页故障与缺页中断处理 - 38 -](#_Toc532238442)

[7.9动态存储分配管理 - 39 -](#_Toc532238443)

[7.10本章小结 - 40 -](#_Toc532238444)

[第8章 hello的IO管理 - 41 -](#_Toc532238445)

[8.1 Linux的IO设备管理方法 - 41 -](#_Toc532238446)

[8.2 简述Unix IO接口及其函数 - 41 -](#_Toc532238447)

[8.3 printf的实现分析 - 42 -](#_Toc532238448)

[8.4 getchar的实现分析 - 44 -](#_Toc532238449)

[8.5本章小结 - 44 -](#_Toc532238450)

[结论 - 45 -](#_Toc532238451)

[附件 - 46 -](#_Toc532238452)

[参考文献 - 48 -](#_Toc532238453)

# 第1章 概述

## 1.1 Hello简介

P2P:在linux 中，hello.c 经过cpp 的预处理、ccl 的编译、as 的汇编、ld 的链接最终成为可执行目标程序hello，在shell 中键入启动命令后，shell 为其fork，产生子进程，内核为新进程创建数据结构， hello 便从可执行程序（Program）变成为进程（Process）。

之后shell 为其execve，删除父进程的原有用户区域，映射私有区域、共享区域。进入程序入口后程序开始载入物理内存，然后进入main 函数执行目标代码，CPU 为运行的hello执行逻辑控制流。当程序运行结束后，shell 父进程负责回收hello 进程，内核删除相关数据结构。

以上整个过程便是020 。

## 1.2 环境与工具

### 1.2.1 硬件环境

CPU：Intel(R) Core(TM) i5-7200U @ 2.50GHz (64位)

GPU：Intel(R) HD Graphics 620

Nvidia GeForce 940MX

物理内存：8.00GB

磁盘：1TB HDD

128GB SSD

### 1.2.2 软件环境

Windows10 64位；

Vmware 14.11；

Ubuntu 18.04 64位；

### 1.2.3工具

Visual Studio 2010 64位；

gedit，gcc，notepad++，readelf, objdump, hexedit, edb

## 1.3 中间结果

hello.c：hello程序c语言源文件



hello.i：hello.c预处理生成的文本文件



hello.s：由hello.i编译得到的汇编文件



hello.o：hello.i汇编得到的可重定位目标文件



hello.o.elf：使用readelf读取hello.o得到的信息



hello.o.asm：使用objdump反汇编hello.o得到的反汇编代码



hello：由hello.o与动态链接库链接得到的可执行目标文件



hello.elf：使用readelf读取hello得到的信息



hello.asm：使用objdump反汇编hello得到的反汇编代码



## 1.4 本章小结

本章主要简单介绍了hello 的p2p，020 过程，列出了本次实验的环境、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

预处理器（cpp）根据以字符#开头是我命令，修改原始的C程序。比如将源文件中用#include形式声明的文件复制到新的程序中。比如hello.c第6-8行中的#include<stdio.h> 等命令告诉预处理器读取系统头文件stdio.h unistd.h stdlib.h 的内容，并把它直接插入到程序文本中。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 预处理的过程

## 2.3 Hello的预处理结果解析

打开hello.i文件可以发现，整个hello.i程序已经拓展为3189行。main函数出现在hello.c中的代码自3102行开始

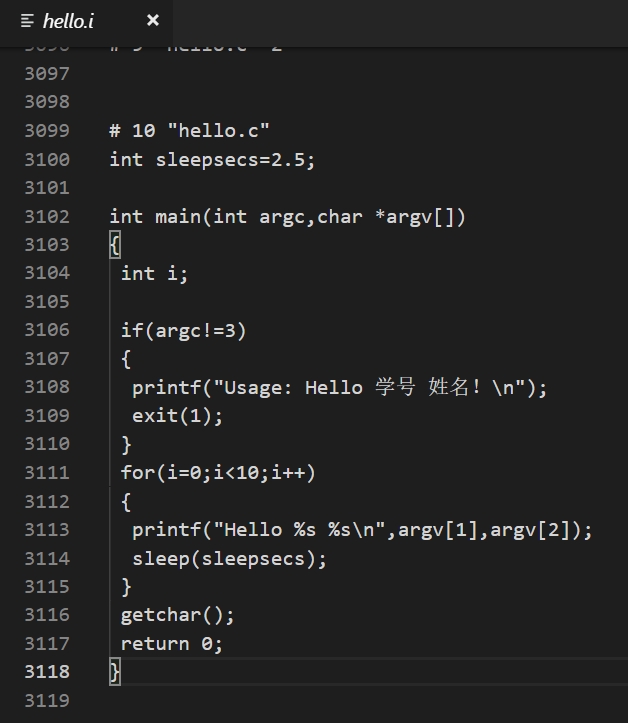


图2.2 预处理后生成的hello.i文件

在这之前出现的是stdio.h unistd.h stdlib.h的依次展开，以stdio.h的展开为例，cpp到默认的环境变量下寻找stdio.h，打开/usr/include/stdio.h 发现其中依然使用了#define语句，cpp对此递归展开，所以最终.i程序中是没有#define的。

## 2.4 本章小结

hello在正式编译前，先要经过预处理。预处理过程只是简单的文本插入过程。生成的hello.i文件仍然是文本文件。

本章主要介绍了预处理的定义与作用、并结合预处理之后的程序对预处理结果进行了解析。

**（第2章0.5分）**

# 第3章 编译

## 3.1 编译的概念与作用

编译器（ccl）将文本文件hello.i翻译成文本文件hello.s，他是一个汇编语言程序。它以文本的形式描述了一条条低级机器语言指令。

## 3.2 在Ubuntu下编译的命令

gcc -m64 -no-pie -fno-PIC -S hello.i -o hello.s

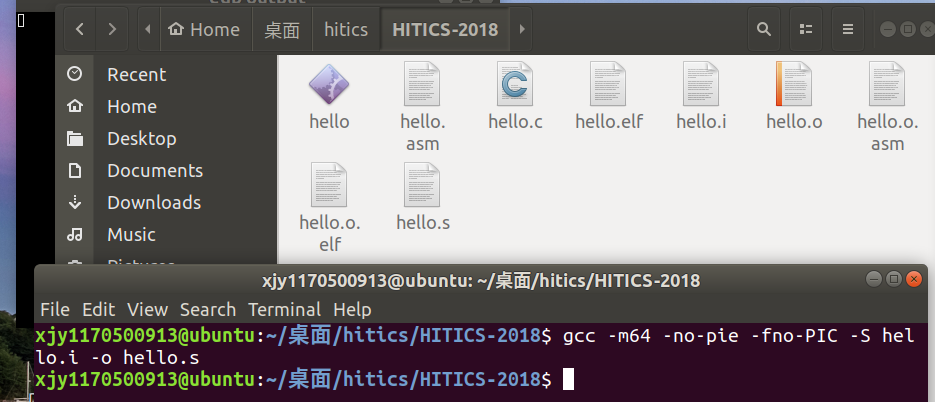


图3.1 编译的过程

## 3.3 Hello的编译结果解析

### 3.3.1 数据

在hello.s中，常量有整数、字符串、数组

1. 整数

程序中涉及的整数有：

* 1. int sleepsecs：sleepsecs 在C 程序中被声明为全局变量，且已经被赋值，编译器处理时在.data 节声明该变量，.data 节存放已经初始化的全局和静态C 变量。在图3.3 中，可以看到，编译器首先将sleepsecs在.text 代码段中声明为全局变量，其次在.data 段中，设置对齐方式为设置类型为对象、设置大小为4 字节、设置为long 类型其值为2。

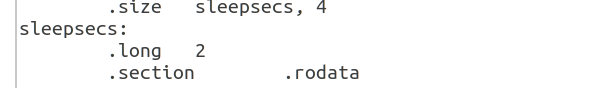


图3.3 hello.s中的sleepsecs

* 1. int i: i作为局部变量，并不占用文件的实际节的空间，仅仅在运行时栈中存在。对其的操作，也都是对寄存器或栈的操作。具体操作在后面操作类型中详述。

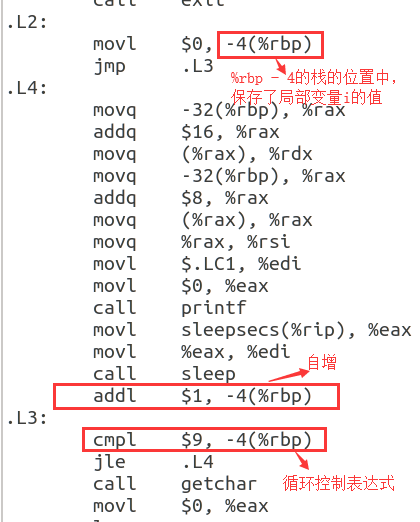


图3.4 局部整型变量i在hello.s中的体现

* 1. if(argc!=3)语句中的变量argc与常量3：
     1. 变量argc：是main函数的参数之一，64位编译下，由寄存器传入，进而保存在堆栈中。
     2. 常量3：直接在汇编语句中，以$3的立即数形式体现。

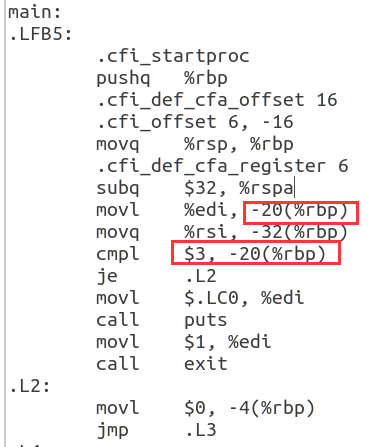


图3.5 变量argc与常量3在hello.s中的体现

1. 字符串

程序中涉及的字符串为

"Usage: Hello 学号 姓名！\n"和"Hello %s %s\n"：

这两者是字符串常量，根据hello.s中的信息，它们存储在.text的数据段中。

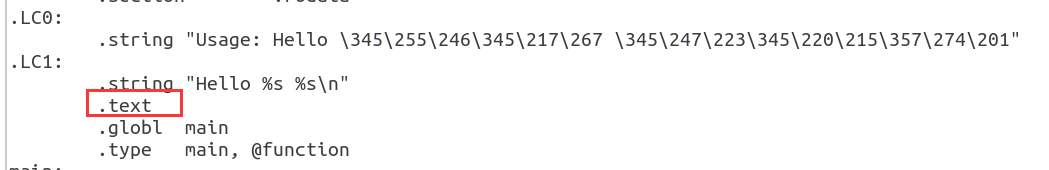


图3.6 两个字符串常量在hello.s中的体现

1. 数组

程序中涉及到的数组为字符串数组（字符数组指针），即main函数的第二个参数char \*argv[]。首先将数组的首元素地址存入栈中。

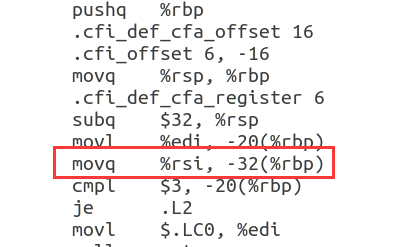


图3.7 将argv存入栈中

访问数组元素时，使用寄存器寻址的方法

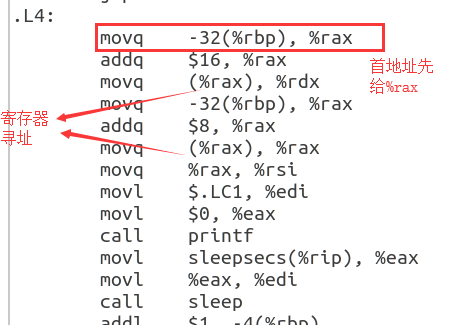
。

图3.8 访问数组元素

### 3.3.2赋值

* 1. int sleepsecs=2.5 ：因为sleepsecs是全局变量，所以直接在.data节中将sleepsecs声明为值2的long类型数据。
  2. i=0：整型数据的赋值使用mov指令完成，根据数据的大小不同使用不同后缀，分别为：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 指令 | b | w | l | q |
| 大小 | 8b (1B) | 16b (2B) | 32b (4B) | 64b (8B) |

### 3.3.3类型转换

程序中涉及隐式类型转换的是：int sleepsecs=2.5，将浮点数类型的2.5转换为int类型。

当在double或float向int进行类型转换的时候，值会向零舍入。例如1.999将被转换成1，-1.999将被转换成-1。进一步来讲，可能会产生值溢出的情况，与Intel兼容的微处理器指定位模式[10…000]为整数不确定值，一个浮点数到整数的转换，如果不能为该浮点数找到一个合适的整数近似值，就会产生一个整数不确定值。

### 3.3.4算数操作

汇编语言中，指令操作如下：

leaq S,D D=&S

INC D D+=1

DEC D D-=1

NEG D D=-D

ADD S,D D=D+S

SUB S,D D=D-S

IMULQ S R[%rdx]:R[%rax]=S\*R[%rax]（有符号）

MULQ S R[%rdx]:R[%rax]=S\*R[%rax]（无符号）

IDIVQ S R[%rdx]=R[%rdx]:R[%rax] mod S（有符号）

R[%rax]=R[%rdx]:R[%rax] div S

DIVQ S R[%rdx]=R[%rdx]:R[%rax] mod S（无符号）

R[%rax]=R[%rdx]:R[%rax] div S

程序中涉及的算数操作有：

1）i++，对计数器i自增，使用程序指令addl

2）汇编中使用leaq .LC1(%rip),%rdi，使用了加载有效地址指令leaq计算LC1的段地址%rip+.LC1并传递给%rdi。

### 3.3.5关系操作

进行关系操作的汇编指令有：

CMP S1,S2 S2-S1 比较-设置条件码

TEST S1,S2 S1&S2 测试-设置条件码

SET\*\* D D=\*\* 按照\*\*将条件码设置D

J\*\* —— 根据\*\*与条件码进行跳转

程序中涉及的关系运算为：

1）argc!=3：判断argc不等于3。hello.s中使用cmpl $3,-20(%rbp)，计算argc-3然后设置条件码，为下一步je利用条件码进行跳转作准备。

2）i<10：判断i小于10。hello.s 中使用cmpl $9,-4(%rbp)，计算i-9 然后设置条件码，为下一步jle 利用条件码进行跳转做准备。

### 3.3.6数组操作

程序中涉及到的数组为字符串数组（字符数组指针），即main函数的第二个参数char \*argv[]。首先将数组的首元素地址存入栈中。

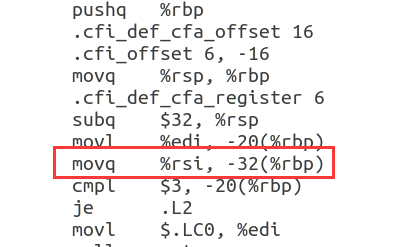


图3.7 将argv存入栈中

访问数组元素时，使用寄存器寻址的方法

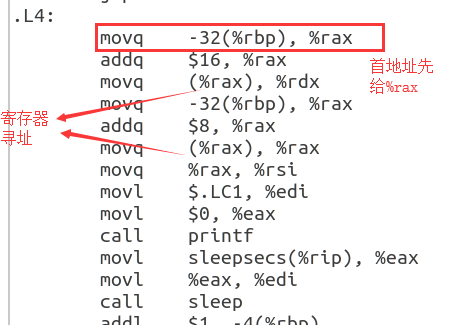
。

图3.8 访问数组元素

### 3.3.7控制转移

程序中的控制转移：

* 1. if (argv!=3)：当argv不为3时，执行后面的代码。hello.s中采用cmpl+je指令，cmp所置的条件码为相等，则证明argv!=3不成立，则简单跳过后面的一段代码。

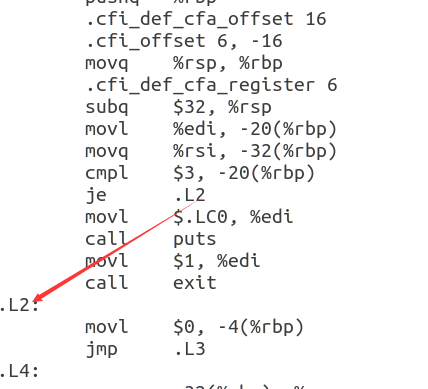


图3.9 控制转移（if语句）

* 1. for(i=0;i<10;i++)：当i < 10时继续进行循环。首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。

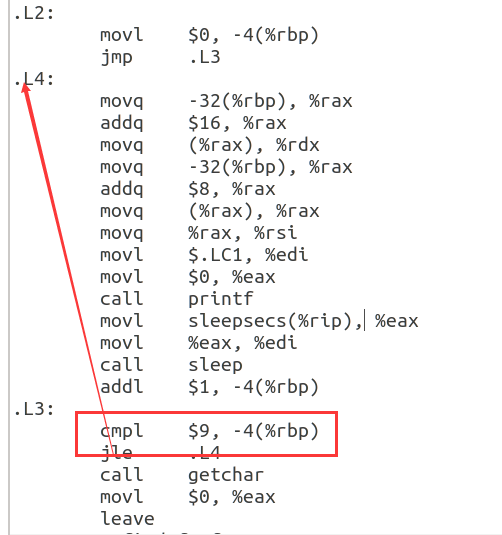


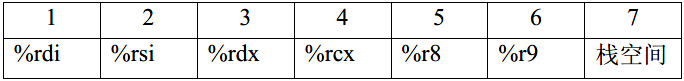
图3.10 控制转移（for循环语句）

### 3.3.8 函数操作

1）传递控制：准备进行过程Q的时候，先将程序计数器（PC）的原值压入栈中，以便之后能够返回。同时PC设置为Q的代码的起始地址，然后在返回时， Q后面那条指令的地址出栈，控制回到P过程。

2）传递数据：P必须能够向Q提供一个或多个参数，Q必须能够向P中返回一个值。64位编译系统采用寄存器传参数。一般使用%rax保存返回值。

64位编译系统的传参次序：



3）分配和释放内存：在开始时，Q可能需要为局部变量分配空间，而在返回前，又必须释放这些空间。

程序中涉及函数操作的有：

程序中涉及函数操作的有：

1. 1）main 函数：

传递控制：main 函数因为被调用call 才能执行（被系统启动函数\_\_libc\_start\_main 调用），call 指令将下一条指令的地址dest 压栈，然后跳转到main 函数。

传递数据：外部调用过程向main 函数传递参数argc 和argv，分别使用%rdi 和%rsi 存储，函数正常出口为return 0，将%eax 设置0返回。

分配和释放内存：使用%rbp 记录栈帧的底，函数分配栈帧空间在%rbp 之上，程序结束时，调用leave 指令，恢复栈空间为调用之前的状态，然后ret返回，ret 相当pop IP，将下一条要执行指令的地址设置为dest。

1. printf 函数：

传递数据：第一次printf 将%rdi 设置为“Usage: Hello 学号，姓名！”字符串的首地址。第二次printf 设置%rdi 为“Hello %s %s\n”的首地址，设置%rsi 为argv[1]，%rdx 为argv[2]。

控制传递：第一次printf 因为只有一个字符串参数，所以call puts；第二次printf 使用call printf。

1. exit 函数：

传递数据：将%edi 设置为1。

控制传递：call exit。

1. sleep 函数：

传递数据：将%edi 设置为sleepsecs。

控制传递：call sleep。

1. getchar 函数：

控制传递：call gethcar

## 3.4 本章小结

编译器将.i 的拓展程序编译为.s 的汇编代码。经过编译之后，我们的hello 自C 语言翻译为更加低级的汇编语言。

本章主要阐述了编译器是如何处理各个数据类型以及各类操作，并结合所给的hello.c程序，生成并查看了hello.s代码，验证了大部分数据、操作在汇编代码中的实现。

**（第3章2分）**

# 第4章 汇编

## 4.1 汇编的概念与作用

汇编器（as）将.s 汇编程序翻译成机器语言指令，把这些指令打包成可重定位目标程序的格式，并将结果保存在.o 目标文件中，.o 文件是一个二进制文件，它包含程序的指令编码、数据。

## 4.2 在Ubuntu下汇编的命令

指令：gcc -m64 -no-pie -fno-PIC -c hello.s -o hello.o

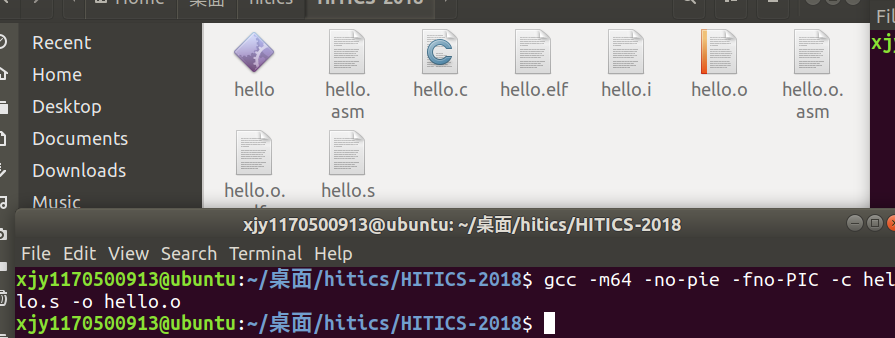


图4.1汇编操作的执行

## 4.3 可重定位目标elf格式

分析hello.o的ELF格式，用readelf等列出其各节的基本信息，特别是重定位项目分析。

使用readelf -a hello.o > hello.elf命令，将readelf的结果全部输出到hello.elf中。

1. ELF头：描述了生成该文件的系统的字大小、字节顺序（大/小）、ELF头的大小、目标文件的类型、机器类型（如x86-64）、节头部表的偏移、节头部表中条目的大小和数量

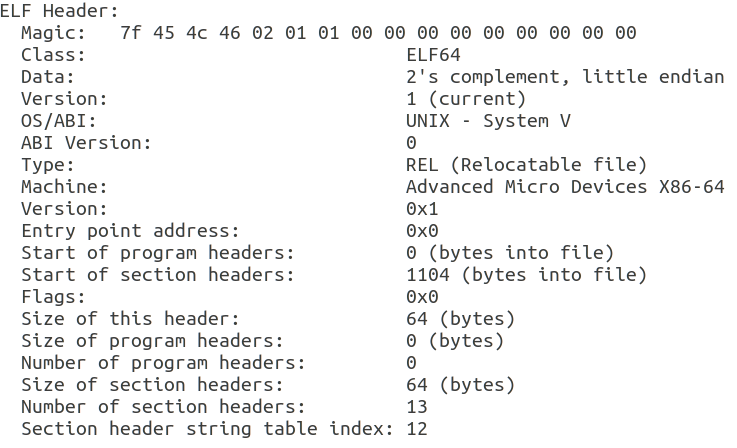


图4.2 ELF头

1. 节头：记录每个节的名称、偏移量、大小、位置等信息

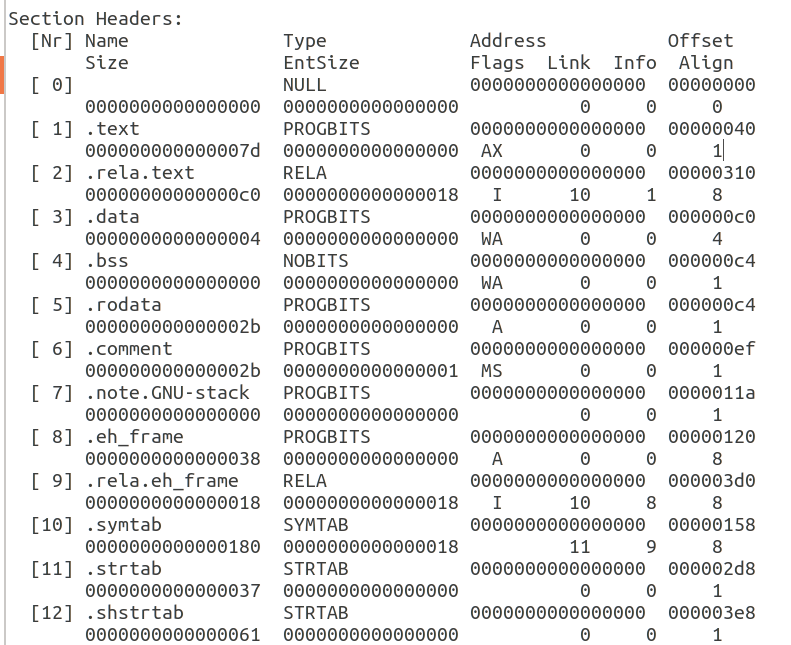


图4.3 节头

1. 重定位节.rela.text: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。

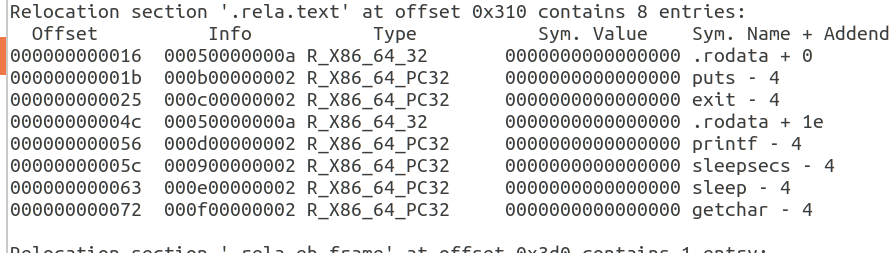


图4.4 重定位节.rela.text

重定位条目的结构成员如下：

|  |  |
| --- | --- |
| 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 | 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 |
| 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算法如下（设需要重定位的.text 节中的位置为src,设重定位的目的位置dst）：

refptr = s +r.offset （1）

refaddr = ADDR(s) + r.offset （2）

\*refptr = (unsigned) (ADDR(r.symbol) + r.addend-refaddr)（3）

其中（1）指向src 的指针（2）计算src 的运行时地址，（3）中， ADDR(r.symbol)计算dst 的运行时地址，在本例中，ADDR(r.symbol)获得的是dst 的运行时地址，因为需要设置的是相对地址，即dst 与下一条指令之间的地址之差，所以要减去src的运行时地址。

1. 重定位节.rela.eh\_frame: eh\_frame 节的重定位信息。

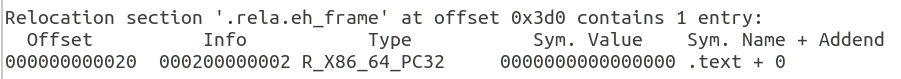


图4.5 重定位节.rela.eh\_frame

1. 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。

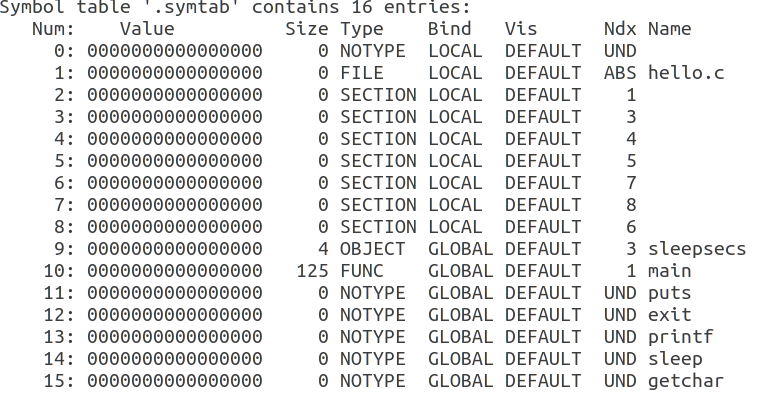


图4.6 符号表

## 4.4 Hello.o的结果解析

objdump -d -r hello.o > hello.o.asm 分析hello.o的反汇编，并请与第3章的 hello.s进行对照分析。

说明机器语言的构成，与汇编语言的映射关系。特别是机器语言中的操作数与汇编语言不一致，特别是分支转移函数调用等。

生成的反汇编文件内容如下：

hello.o: file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:

0: 55 push %rbp

1: 48 89 e5 mov %rsp,%rbp

4: 48 83 ec 20 sub $0x20,%rsp

8: 89 7d ec mov %edi,-0x14(%rbp)

b: 48 89 75 e0 mov %rsi,-0x20(%rbp)

f: 83 7d ec 03 cmpl $0x3,-0x14(%rbp)

13: 74 14 je 29 <main+0x29>

15: bf 00 00 00 00 mov $0x0,%edi

16: R\_X86\_64\_32 .rodata

1a: e8 00 00 00 00 callq 1f <main+0x1f>

1b: R\_X86\_64\_PC32 puts-0x4

1f: bf 01 00 00 00 mov $0x1,%edi

24: e8 00 00 00 00 callq 29 <main+0x29>

25: R\_X86\_64\_PC32 exit-0x4

29: c7 45 fc 00 00 00 00 movl $0x0,-0x4(%rbp)

30: eb 39 jmp 6b <main+0x6b>

32: 48 8b 45 e0 mov -0x20(%rbp),%rax

36: 48 83 c0 10 add $0x10,%rax

3a: 48 8b 10 mov (%rax),%rdx

3d: 48 8b 45 e0 mov -0x20(%rbp),%rax

41: 48 83 c0 08 add $0x8,%rax

45: 48 8b 00 mov (%rax),%rax

48: 48 89 c6 mov %rax,%rsi

4b: bf 00 00 00 00 mov $0x0,%edi

4c: R\_X86\_64\_32 .rodata+0x1e

50: b8 00 00 00 00 mov $0x0,%eax

55: e8 00 00 00 00 callq 5a <main+0x5a>

56: R\_X86\_64\_PC32 printf-0x4

5a: 8b 05 00 00 00 00 mov 0x0(%rip),%eax # 60 <main+0x60>

5c: R\_X86\_64\_PC32 sleepsecs-0x4

60: 89 c7 mov %eax,%edi

62: e8 00 00 00 00 callq 67 <main+0x67>

63: R\_X86\_64\_PC32 sleep-0x4

67: 83 45 fc 01 addl $0x1,-0x4(%rbp)

6b: 83 7d fc 09 cmpl $0x9,-0x4(%rbp)

6f: 7e c1 jle 32 <main+0x32>

71: e8 00 00 00 00 callq 76 <main+0x76>

72: R\_X86\_64\_PC32 getchar-0x4

76: b8 00 00 00 00 mov $0x0,%eax

7b: c9 leaveq

7c: c3 retq

1. 机器语言的构成

（1）操作码。它具体说明了操作的性质及功能。一台计算机可能有几十条至几百条指令，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。

（2）操作数的地址。CPU通过该地址就可以取得所需的操作数。

（3）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。

1. 机器语言与汇编语言的映射关系

指令码代之以记忆符号，地址码代之以符号地址，使得其含义显现在符号上而不再隐藏在编码中。

1. 两者在hello.s与反汇编代码中的区别

与hello.s进行对比后发现，反汇编的代码多出了机器语言部分。汇编语言部分大部分相同，只有函数调用、过程转移、全局变量访问、字符串常量不一样，hello.s中，这四者都是用符号直接表示，而反汇编代码中与此不同：

跳转目标：反汇编中，跳转目标在机器代码中体现，用目标指令地址与当前指令下一条指令的地址之差的补码表示。

过程转移：由于hello程序中调用的都是共享库中的函数，其地址还需重定位。因此暂时定为call xx下条指令的地址，后面附有重定位标识。

全局变量访问：暂时使用0x0(%rip)标识，其地址也许重定位。后面同样附有重定位标识。

字符串常量：使用0暂时代替，后面同样重定位到.rodata节的标识。

## 4.5 本章小结

本章介绍了hello 从hello.s 到hello.o 的汇编过程，查看了hello.o 的elf 格式、使用objdump 得到反汇编代码与hello.s 进行比较，了解从汇编语言映射到机器语言汇编器需要实现的转换。在这个过程中，生成了重定位条目，为之后的链接中的重定位过程奠定了基础。

**（第4章1分）**

# 第5章 链接

## 5.1 链接的概念与作用

链接是将各种代码和数据片段收集并组合成一个单一文件的过程，这个文件可被加载到内存并执行。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。链接是由叫做链接器的程序执行的。链接器使得分离编译成为可能。

## 5.2 在Ubuntu下链接的命令

命令如下：

ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib/x86\_64-linux-gnu/crt1.o /usr/lib/x86\_64-linux-gnu/crti.o /usr/lib/gcc/x86\_64-linux-gnu/7/crtbegin.o /usr/lib/gcc/x86\_64-linux-gnu/7/crtend.o /usr/lib/x86\_64-linux-gnu/crtn.o hello.o -lc -z relro -o hello

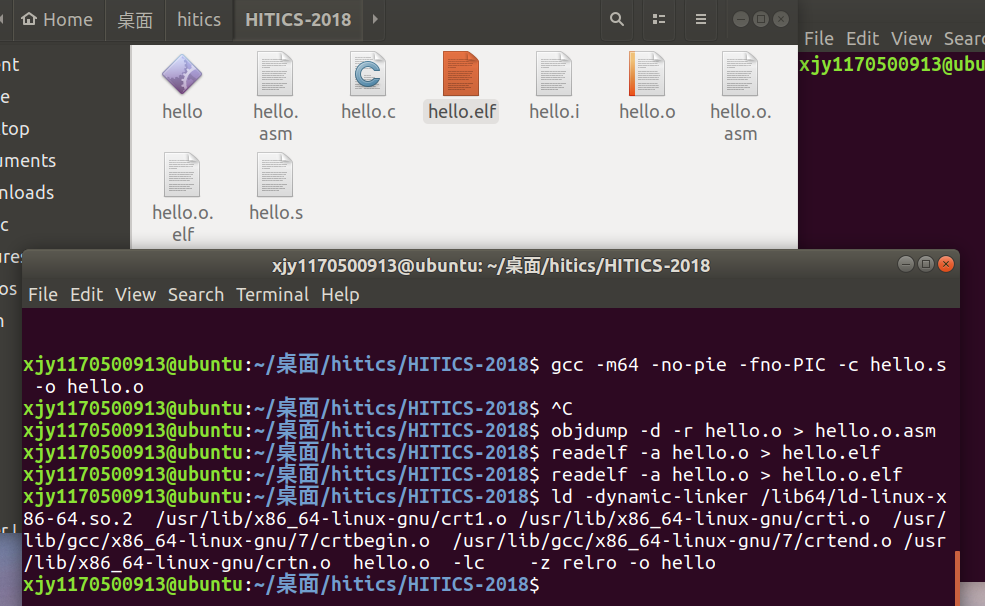


图5.1 链接过程截图

## 5.3 可执行目标文件hello的格式

分析hello的ELF格式，用readelf等列出其各段的基本信息，包括各段的起始地址，大小等信息。

使用readelf -a hello > hello.elf 命令生成hello 程序的ELF 格式文件。

1. ELF头：描述文件的总体格式。还包括程序的入口点。

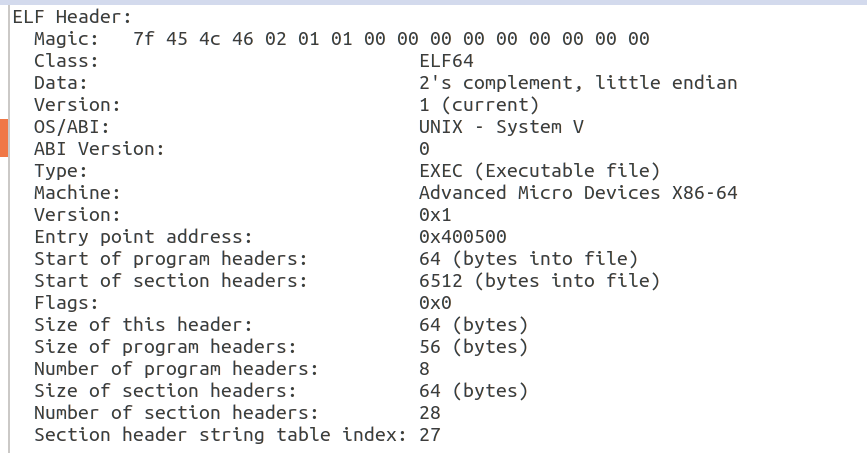


图5.2 ELF头内容

1. 节头：记录每个节的名称、偏移量、大小、位置等信息

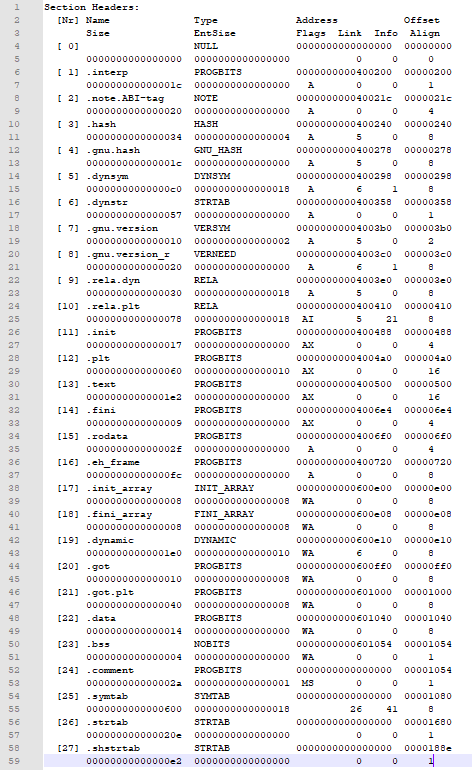


图5.3 节头内容

## 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，每个节排列都同5.3中的地址声明，在0~fff空间中，与0x400000~0x401000段的存放的程序相同，在fff之后存放的是.dynamic~.shstrtab节

查看ELF文件的程序头，程序头在为链接器提供运行时的加载内容和提供动态链接的信息，每一个表项提供了各段在虚拟地址空间大小和物理地址空间大小，位置，标志，访问权限和对齐方式，红框表示虚拟地址空间信息

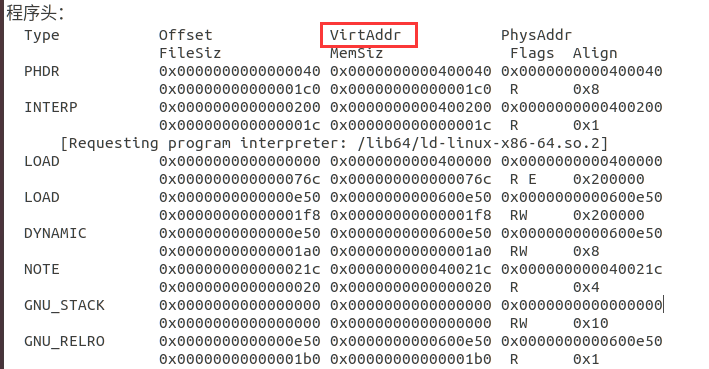


图5.4 ELF文件的程序头

从图中可以看出，程序头包括8个段

|  |  |
| --- | --- |
| **PHDR** | **保存程序头表** |
| **INTERP** | 指定在程序已经从可执行文件映射到内存之后，必须调用的解释器（如动态链接器）。 |
| **LOAD** | 表示一个需要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。 |
| **DYNAMIC** | 保存了由动态链接器使用的信息 |
| **NOTE** | 保存辅助信息 |
| **GNU\_STACK** | 权限标志，标志栈是否是可执行的 |
| **GNU\_RELRO** | 指定在重定位结束之后那些内存区域是需要设置只读 |

## 5.5 链接的重定位过程分析

objdump -d -r hello 分析hello与hello.o的不同，说明链接的过程。

结合hello.o的重定位项目，分析hello中对其怎么重定位的。

使用objdump -d -r hello > hello.asm命令生成反汇编文件,与之前的hello.o.asm文件对比。有以下几点的不同：

1. 多出了.plt，puts@plt，printf@plt，getchar@plt，exit@plt，sleep@plt等函数的代码.

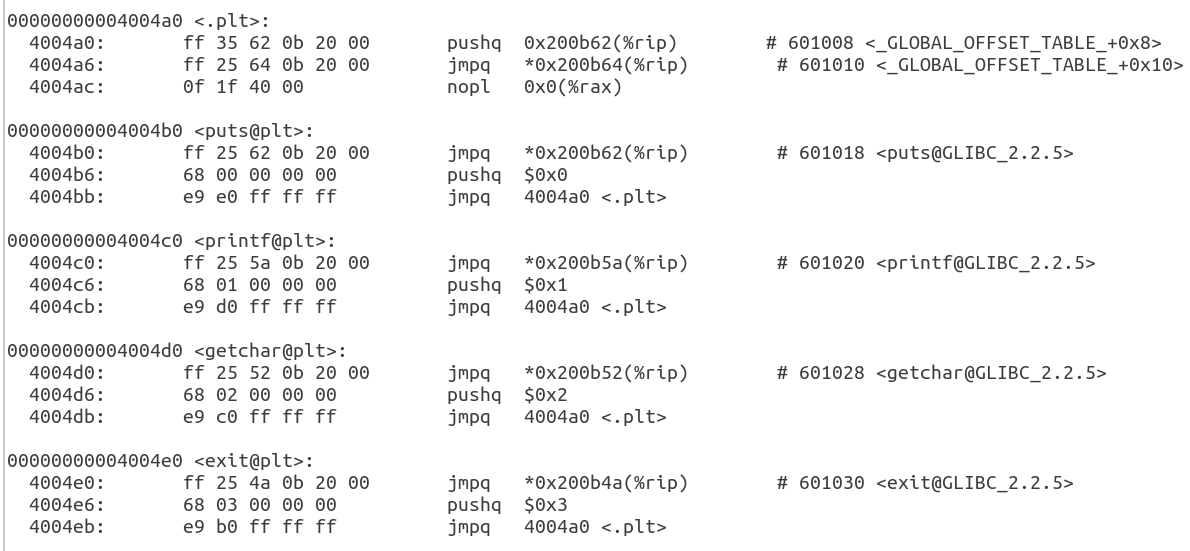


图5.5 多出的函数代码

1. 之前未重定位的全局变量引用、过程调用、控制转移全部定位了。比如，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.rodata的引用也发生了改变，.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。

## 5.6 hello的执行流程

使用edb执行hello，说明从加载hello到\_start，到call main,以及程序终止的所有过程。请列出其调用与跳转的各个子程序名或程序地址。

使用edb 执行hello，观察函数执行流程，将过程中执行的主要函数列在下面：

|  |  |
| --- | --- |
| 程序名称 | 程序地址 |
| ld-2.27.so!\_dl\_start | 0x7fce 8cc38ea0 |
| ld-2.27.so!\_dl\_init | 0x7fce 8cc47630 |
| hello!\_start | 0x400500 |
| libc-2.27.so!\_\_libc\_start\_main | 0x7fce 8c867ab0 |
| -libc-2.27.so!\_\_cxa\_atexit | 0x7fce 8c889430 |
| -libc-2.27.so!\_\_libc\_csu\_init | 0x4005c0 |
| hello!\_init | 0x400488 |
| libc-2.27.so!\_setjmp | 0x7fce 8c884c10 |
| -libc-2.27.so!\_sigsetjmp | 0x7fce 8c884b70 |
| --libc-2.27.so!\_\_sigjmp\_save | 0x7fce 8c884bd0 |
| hello!main | 0x400532 |
| hello!puts@plt | 0x4004b0 |
| hello!exit@plt | 0x4004e0 |
| ld-2.27.so!\_dl\_runtime\_resolve\_xsave | 0x7fce 8cc4e680 |
| -ld-2.27.so!\_dl\_fixup | 0x7fce 8cc46df0 |
| --ld-2.27.so!\_dl\_lookup\_symbol\_x | 0x7fce 8cc420b0 |
| libc-2.27.so!exit | 0x7fce 8c889128 |

## 5.7 Hello的动态链接分析

因为编译器没有办法知道函数运行时的地址，需要链接器进行连接处理，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，在加载时，动态链接器会重定位GOT中的每个条目，使得它包含目标的正确的绝对地址。

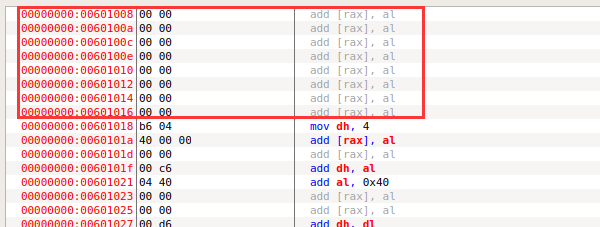


图5.6 执行dl\_init前

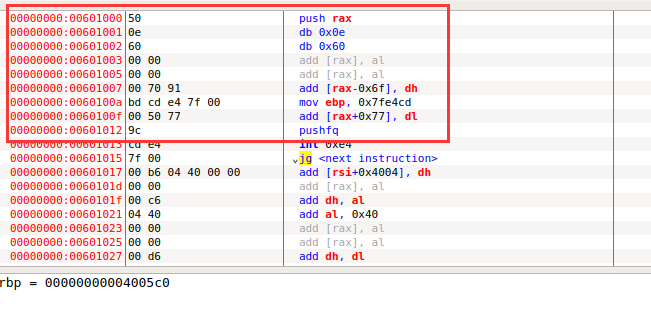


图5.7 执行dl\_init后

在dl\_init调用之后，0x601008和0x601010处的两个8B数据分别发生改变，其中变化便是GOT[1]指向重定位表，用来确定调用的函数地址，然后在调用函数时，先跳转到PLT执行.plt中逻辑，然后访问动态链接器确定函数地址，重写GOT（以便再次访问该函数时可直接跳转），将控制传递给目标函数。

## 5.8 本章小结

本章主要介绍了链接的概念与作用，可执目标文件的格式，分析了hello的虚拟地址空间，重定位的过程。

**（第5章1分）**

# 第6章 hello进程管理

## 6.1 进程的概念与作用

进程是一个执行中的程序的实例，每一个进程都有它自己的地址空间，一般情况下，包括文本区域、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；堆栈区域存储区着活动过程调用的指令和本地变量。

进程为用户提供了以下假象：我们的程序好像是系统中当前运行的唯一程序一样，我们的程序好像是独占的使用处理器和内存，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 的作用：Shell 是一个用C 语言编写的程序，他是用户使用Linux 的桥梁。

Shell 是指一种应用程序，Shell 应用程序提供了一个界面，用户通过这个界面访问操作系统内核的服务。

处理流程：

1）从终端读入输入的命令。

2）将输入字符串切分获得所有的参数

3）如果是内置命令则立即执行

4）否则调用相应的程序为其分配子进程并运行

5）shell 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

在终端中键入./hello 1170500913 xjy，运行的终端程序会对输入的命令行进行解析，因为hello 不是一个内置的shell 命令。所以解析之后终端程序判断./hello 的语义为执行当前目录下的可执行目标文件hello，之后终端程序首先会调用fork 函数创建一个新的运行的子进程，新创建的子进程几乎但不完全与父进程相同，子进程得到与父进程用户级虚拟地址空间相同的（但是独立的）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最大的区别在于它们拥有不同的PID。

## 6.4 Hello的execve过程

当fork 之后，子进程调用execve 函数（传入命令行参数）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用户虚拟内存段，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。除了一些头部信息，在加载过程中没有任何从磁盘到内存的数据复制。直到CPU引用一个被映射的虚拟页时才会进行复制，这时，操作系统利用它的页面调度机制自动将页面从磁盘传送到内存。

加载器创建的内存映像如下：

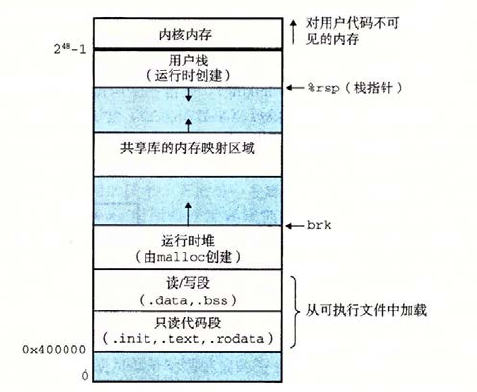


图6.1 进程的内存映像

## 6.5 Hello的进程执行

逻辑控制流：一系列程序计数器PC 的值的序列叫做逻辑控制流，进程是轮流使用处理器的，在同一个处理器核心中，每个进程执行它的流的一部分后被抢占（暂时挂起），然后轮到其他进程。

时间片：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。

用户模式和内核模式：处理器通常使用一个寄存器提供两种模式的区分，该寄存器描述了进程当前享有的特权，当没有设置模式位时，进程就处于用户模式中，用户模式的进程不允许执行特权指令，也不允许直接引用地址空间中内核区内的代码和数据；设置模式位时，进程处于内核模式，该进程可以执行指令集中的任何命令，并且可以访问系统中的任何内存位置。

上下文信息：上下文就是内核重新启动一个被抢占的进程所需要的状态，它由通用寄存器、浮点寄存器、程序计数器、用户栈、状态寄存器、内核栈和各种内核数据结构等对象的值构成。

在hello运行过程中，若hello进程不被抢占，则正常执行；若被抢占，则进入内核模式，进行上下文切换，转入用户模式，调度其他进程。

这里有个特殊的情况，当hello执行到sleep()的时候，为了不浪费处理器资源，hello进程会被抢占，直到sleep()返回，触发一个中断，使得hello进程重新被调度。

## 6.6 hello的异常与信号处理

1. 正常执行hello程序的过程中：
   1. 进程收到的信号：无
   2. 收到的异常：
      1. 时钟中断：不处理
      2. 系统调用：调用write函数打印字符
      3. 缺页异常（可能）：缺页异常处理程序



图6.2(a) 正常执行hello程序的过程

1. 执行Ctrl-z操作：
   1. 进程收到的信号：SIGSTOP，挂起（停止）
   2. 收到的异常：
      1. 时钟中断：不处理

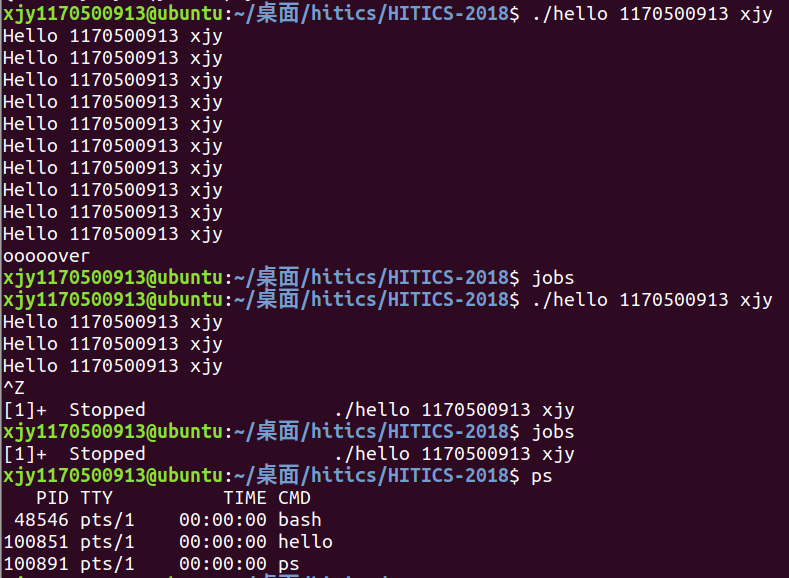


图6.2(b) 执行Ctrl-z后执行jobs、ps命令

1. 执行fg命令：
   1. 进程收到的信号：SIGCONT，继续运行
   2. 收到的异常：
      1. 时钟中断：不处理



图6.2(c) 执行Ctrl-z后执行fg命令

1. 执行 kill -9 %1命令：
   1. 进程收到的信号：SIGKILL，杀死进程
   2. 收到的异常：
      1. 时钟中断：不处理

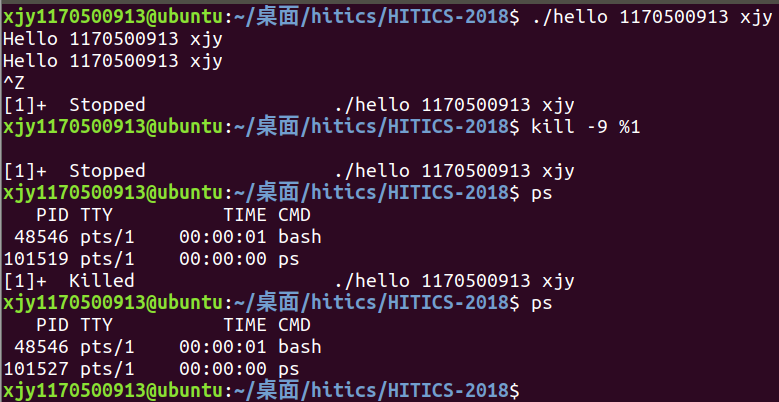


图6.2(d) 执行Ctrl-z后执行执行 kill -9 %1命令

1. 执行Ctrl-c指令：
   1. 进程收到的信号：SIGINT，shell使进程终止
   2. 收到的异常：
      1. 时钟中断：不处理

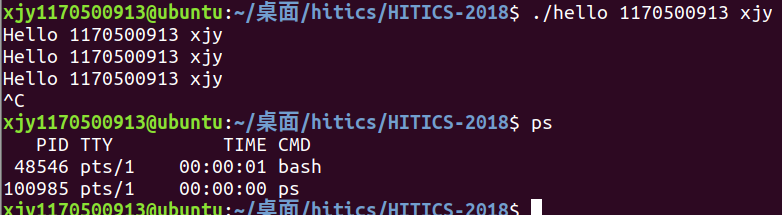


图6.2(e) 执行Ctrl-c指令

1. 运行中乱按：
   1. 进程收到的信号：无
   2. 收到的异常：
      1. 时钟中断：不处理
      2. 系统调用：调用write函数打印字符
      3. I/O中断：将键盘输入的字符读入缓冲区
      4. 缺页故障（可能）：缺页处理程序

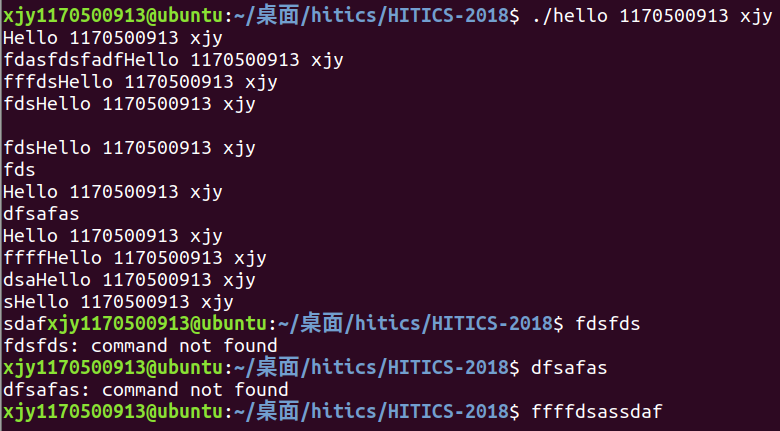


图6.2(e) 运行途中乱按

## 6.7本章小结

本章阐明了进程的定义与作用，介绍了Shell 的一般处理流程，调用fork创建新进程，调用execve 执行hello，hello 的进程执行，hello 的异常与信号处理。

**（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

1. 逻辑地址：

由段选择符+偏移地址构成。其中段选择符位于段寄存器（16位，CS、SS等）中。**而偏移地址即为汇编、c代码中显示的地址。**

段寄存器有：

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

下面以hello.asm（hello的反汇编文件）为例，说明逻辑地址：

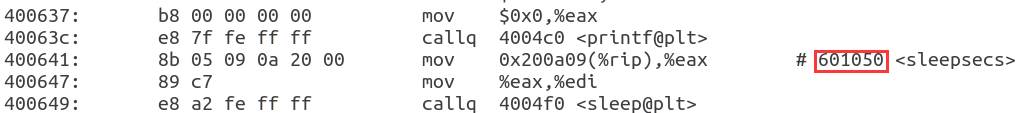


图7.1 hello.asm中的偏移地址

如图7.1所示，为程序对全局变量sleepsec的访问，程序中显式表示的0x601050即为偏移地址。由于全局变量在代码段，此时，DS寄存器存储着段选择符。使用GDB调试，可以查看当前DS寄存器的值：

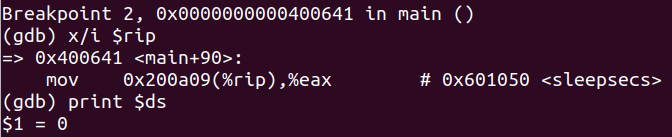


图7.2 使用gdb查看DS的值

可知DS中的值为0，即段选择符为0

综上，hello中，对全局变量sleepsec的访问，逻辑地址为

0000 : 601050

1. 线性地址：

是逻辑地址到物理地址变换的中间层，是处理器可寻址空间的地址。程序代码产生的偏移地址加上段基地址就产生了线性地址。而在实模式、保护模式下。段基地址有着不同的确定方式。

1. 虚拟地址：CSAPP课本中的虚拟地址即为上述的线性地址。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

根据7.1节的描述，逻辑地址=段选择符 ： 偏移地址，而线性地址 = 段基址+ 偏移地址。因此，段式管理的关。键在于：段选择符🡪段基址。

处理器有两种寻址模式：实模式与保护模式，下面分别对这两种模式，解释段式管理过程。

1. 实模式

实模式下，段选择符🡪段基址的转换极为简单，段基址 = 段选择符 \* 16。也就是说：线性地址 = 段选择符 \* 16 + 偏移地址。甚至，不存在线性地址的中间概念，物理地址 = 线性地址 = 段选择符 \* 16 + 偏移地址。

1. 保护模式

保护模式才是现代计算机常用的寻址模式。保护模式下，段选择符并不是通过直接计算得到段基址，而是作为一个索引，到一个称为描述符表的数据结构中读取段基址。此时，16位的段选择符也被划分成了几个部分，予以不同的解释。

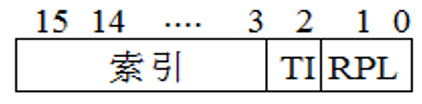


图7.3 保护模式下的段选择符

TI=0，选择全局描述符表(GDT)，TI=1，选择局部描述符表(LDT)。

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。

高13位（8K个索引）：用来确定当前使用的段描述符在描述符表中的位置。

计算机维护一系列表，称为**描述符表**。描述符表分为三种：全局描述符表（GDT）、局部描述符表（LDT）、中断描述符表（IDT）。描述符表的每个表项，大小为8个字节，称为**段描述符**。段描述符的内容如下图。

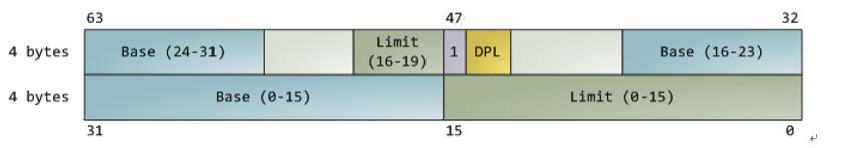


图7.4 段描述符的结构

BASE：段基址

Limit：指出这个段有多大。

DPL：描述符的特权级（privilege level），其值从0（最高特权，内核模式）到3（最低特权，用户模式），用于控制对段的访问。

于是整个过程如下图所示：

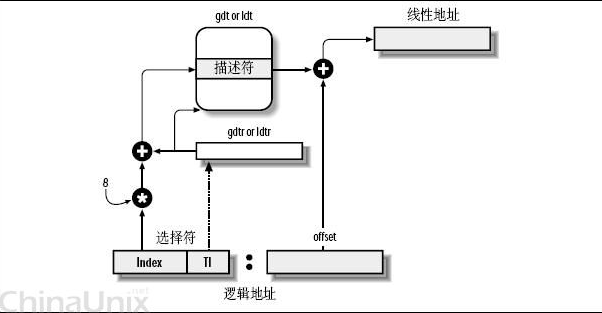


图7.5 保护模式下逻辑地址🡪线性地址的过程

根据段寄存器中的段选择符，检索到对应的描述符表中的段描述符，读取其中的BASE位，即为段基址。最后把偏移地址 + 段基址，即得到线性地址。

## 7.3 Hello的线性地址到物理地址的变换-页式管理

通过段式管理，我们得到了线性地址（虚拟地址VA）。下虚拟地址被分为两个部分：VPN（虚拟页号）和VPO（虚拟页偏移量），由于虚拟内存与物理内存的页大小相同，因此VPO与PPO（物理页偏移量）一致。而PPN（物理页号）则需通过访问页表中的页表条目（PTE）获取。如图7.6所示。

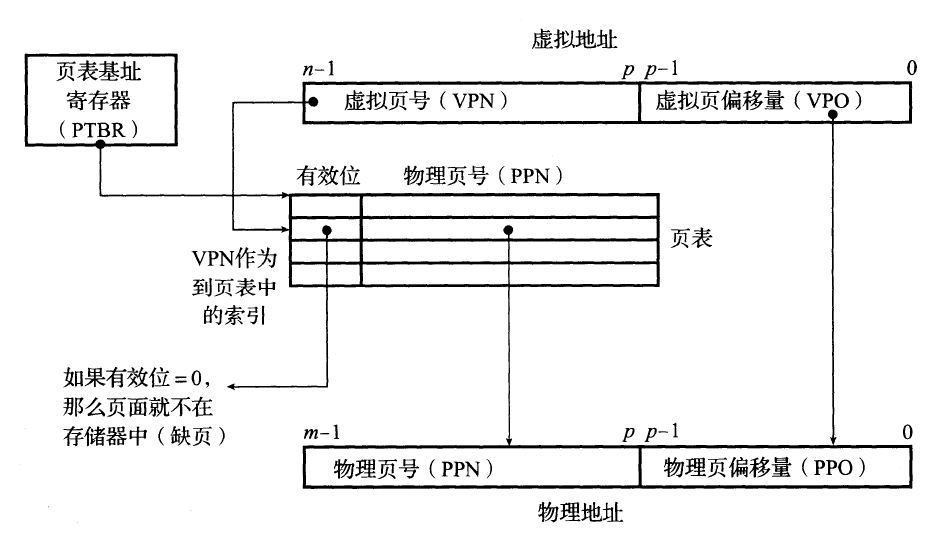


图7.6 地址翻译

若PTE的有效位为1，页命中，则获取到PPN，与PPO组成物理地址。

若PTE的有效位为0，说明对应虚拟页没有缓存到物理内存中，引发一个缺页故障，调用操作系统的内核的缺页处理程序，确定牺牲页，并调入新的页面。再返回到原来的进程，再次调用导致缺页的指令。这次页命中，则获取到PPN，与PPO组成物理地址。

## 7.4 TLB与四级页表支持下的VA到PA的变换

页表位于内存中，CPU中生成虚拟地址，根据虚拟内存中的VPN访问页表，为了提升访问速度，在这两者之间设置一个缓存，即为TLB。CPU进行地址翻译时，首先把VPN解释为TLBT（TLB标记）和TLBI（TLB索引）。

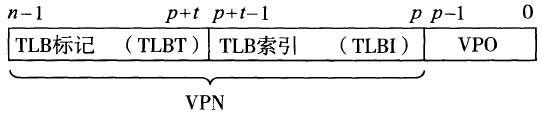


图7.7 虚拟地址的进一步划分

由TLBI，访问TLB中的某一组。遍历该组中的所有行，若找到一行的tag等于TLBT，且有效位valid为1,，则缓存命中，该行存储的即为PPN；若未找到一行的tag等于TLBT，或找到但该行的valid为0，则缓存不命中。进而需要到页表中找到被请求的块，用以替换原TLB表项中的数据。

下面考虑在TLB中缓存不命中的情况，用以说明对四级页表的访问：

缓存不命中后，VPN被解释成从低位到高位的4段，从高地址开始，第一段VPN作为第一级页表的索引，用以确定第二级页表的基址；第二段VPN作为第二级页表的索引，用以确定第三级页表的基址；第三段VPN作为第三级页表的索引，用以确定第四级页表的基址；第四段VPN作为第四级页表的索引，若该位置的有效位为1，则第四级页表中的该表项存储的是所需要的PPN的值。值得注意的是，在上述过程中，只要有一级页表条目的有效位为0，下一级页表就不存在，也就是产生缺页故障了。缺页故障的处理与上节所述相同。

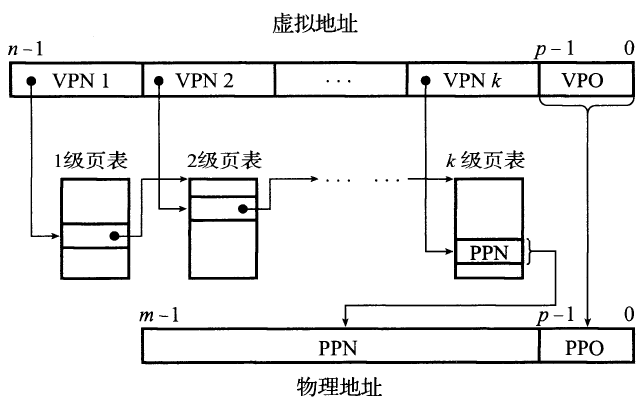


图7.8 对多级页表的访问

## 7.5 三级Cache支持下的物理内存访问

物理地址得到之后，CPU尝试访问物理内存，此时，物理地址被解释为三部分，分别是CT（缓存标记），CI（组索引），CO（块偏移）

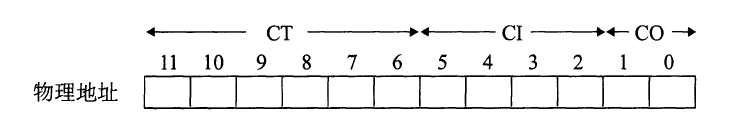


图7.9 物理地址

首先根据组索引，找到缓存中对应的组，遍历该组中的所有行，若找到一行的tag等于CT，且标志位valid为1，则缓存命中（hit）,根据CO（块偏移）读取块中对应的字；若未找到tag为CT的行，或该行的valid值为0，则缓存不命中。

若缓存不命中，则向下一级缓存中查找数据（L2 Cache->L3 Cache->主存）。找到数据之后，开始进行**行替换**。若该组中有一个空行，那就将数据缓存至这个空行，并置好tag和valid位；若该组中没有空行，有两种常用的策略来决定替换掉哪一行：最不常使用（LFU）策略和最近最少使用（LRU）策略。LFU策略会替换过去某个时间窗口内引用次数最少的那一行。LRU策略会替换最后一次访问时间最久远的那一行。

## 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。

## 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。

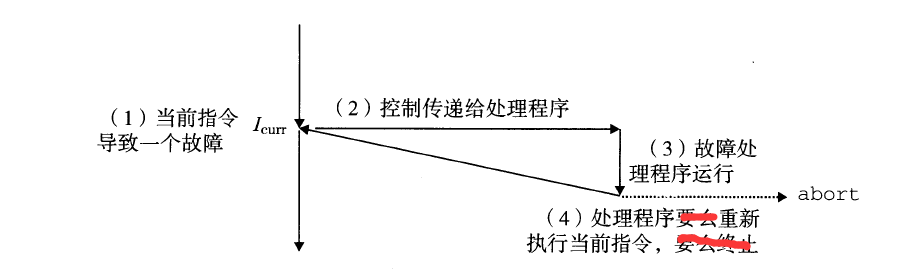


图7.10 缺页故障处理

## 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。分配器将堆视为一组不同大小的块的集合来维护。每个块就是一个连续的虚拟内存片，要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可用来分配。空闲块保持空闲，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。

分配器分为两种基本风格：显式分配器、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。

隐式分配器：要求分配器检测一个已分配块何时不再使用，那么就释放这个块，自动释放未使用的已经分配的块的过程叫做垃圾收集。

一、带边界标签的隐式空闲链表

1）堆及堆中内存块的组织结构：

在内存块中增加4B 的Header 和4B 的Footer，其中Header 用于寻找下一个blcok，Footer 用于寻找上一个block。Footer 的设计是专门为了合并空闲块方便的。因为Header 和Footer 大小已知，所以我们利用Header 和Footer 中存放的块大小就可以寻找上下block。

2）隐式链表

所谓隐式空闲链表，对比于显式空闲链表，代表并不直接对空闲块进行链接，而是将对内存空间中的所有块组织成一个大链表，其中Header 和Footer

中的block 大小间接起到了前驱、后继指针的作用。

3）空闲块合并

因为有了Footer，所以我们可以方便的对前面的空闲块进行合并。合并的情况一共分为四种：前空后不空，前不空后空，前后都空，前后都不空。对于四种情况分别进行空闲块合并，我们只需要通过改变Header 和Footer 中的值就可以完成这一操作。

二、显示空间链表基本原理

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，如下图：

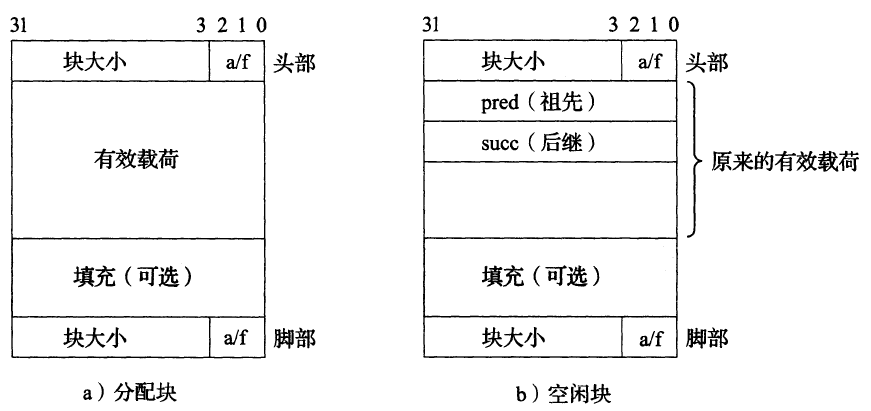


图7.10 隐式空闲链表与显式空闲链表的块结构

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。

维护链表的顺序有：后进先出（LIFO），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，释放一个块可以在线性的时间内完成，如果使用了边界标记，那么合并也可以在常数时间内完成。按照地址顺序来维护链表，其中链表中的每个块的地址都小于它的后继的地址，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，按照地址排序首次适配比IFO 排序的首次适配有着更高的内存利用率，接近最佳适配的利用率。

## 7.10本章小结

本章主要介绍了hello 的存储器地址空间、intel 的段式管理、hello 的页式管理， VA 到PA 的变换、物理内存访问，还介绍了hello进程fork 时的内存映射、execve 时的内存映射、缺页故障与缺页中断处理、动态存储分配管理。

至此，hello进程的虚拟内存空间已经加载完毕，程序执行时，涉及到访存的操作，严格按照上述的逻辑地址🡪虚拟地址🡪物理地址，利用每一个缓存，高效快速地从内存、硬盘上获取所需的数据。

**（第7章 2分）**

# 第8章 hello的IO管理

## 8.1 Linux的IO设备管理方法

设备的模型化：所有的IO设备都被模型化为文件

设备管理：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

接口：

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
2. Linux shell创建的每个进程开始时都有三个打开的文件：标准输入，标准输出，标准错误
3. 改变当前文件位置：对于每个打开的文件，内核保持着一个文件位置k，初始为0，这个文件位置是从文件开头起始的字节偏移量，应用程序能够通过执行seek，显式地将改变当前文件位置k
4. 读写文件：一个读操作就是从文件复制n>0个字节到内存，从当前文件位置k开始，然后将k增加到k+n，给定一个大小为m字节的而文件，当k>=m时执行读操作时触发EOF。类似一个写操作就是从内存中复制n>0个字节到一个文件，从当前文件位置k开始，然后更新k
5. 关闭文件：当应用完成了访问，它就通知内核关闭这个文件，并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去

函数：

1. int open(char\* filename,int flags,mode\_t mode) :进程通过调用open函数来打开一个存在的文件或是创建一个新文件的。open函数将filename转换为一个文件描述符，并且返回描述符数字，返回的描述符总是在进程中当前没有打开的最小描述符，flags参数指明了进程打算如何访问这个文件，mode参数指定了新文件的访问权限位
2. int close(fd)，fd是需要关闭文件的描述符
3. ssize\_t read(int fd,void \*buf,size\_t n)，该函数从描述符为fd的当前位置最多赋值n个字节到内存buf的位置，返回值为实际传送的字节数量
4. ssize\_t wirte(int fd,const void \*buf,size\_t n)，write函数从内存位置buf复制至多n个字节到描述符为fd的当前文件位置

## 8.3 printf的实现分析

查看printf代码

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. }

va\_list的定义：typedef char \*va\_list，说明它是一个字符指针，其中 (char\*)(&fmt) + 4) 即arg表示的是...中的第一个参数

然后我们再来查看vsprintf代码

1. **int** vsprintf(**char** \*buf, **const** **char** \*fmt, **va\_list** args)
2. {
3. **char**\* p;
4. **char** tmp[256];
5. **va\_list** p\_next\_arg = args;
7. **for** (p=buf;\*fmt;fmt++) {
8. **if** (\*fmt != '%') {
9. \*p++ = \*fmt;
10. **continue**;
11. }
13. fmt++;
15. **switch** (\*fmt) {
16. **case** 'x':
17. itoa(tmp, \*((**int**\*)p\_next\_arg));
18. strcpy(p, tmp);
19. p\_next\_arg += 4;
20. p += strlen(tmp);
21. **break**;
22. **case** 's':
23. **break**;
24. **default**:
25. **break**;
26. }
27. }
29. **return** (p - buf);
30. }

先思考printf函数的功能：接受一个格式化命令，并按指定的匹配的参数格式化输出

故I = vsprintf(buf, fmt, arg)应该是得到打印出来的字符串长度，以及后面的write(buf, i)应该是将buf中的i个元素写到终端

所以vsprintf的作用就是格式化。它接受确定输出格式的格式字符串fmt。用格式字符串对个数变化的参数进行格式化，产生格式化输出

下面我们来看write函数

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

这里是给几个寄存器传递了几个参数，然后一个int INT\_VECTOR\_SYS\_CALL代表通过系统调用syscall，查看syscall的实现：

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

syscall将字符串中的字节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，符显示驱动子程序：从ASCII到字模库到显示vram（存储每一个点的RGB颜色信息）。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

异步异常-键盘中断的处理：键盘接口得到一个代表该按键的键盘扫描码，同时同时产生中断请求，请求抢占当前进程运行键盘中断子程序，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

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

了解了printf的函数和getchar函数的底层实现，主要介绍了linux的IO设备管理方法和及其接口和函数。

**（第8章1分）**

# 

# 结论

hello程序经过以下的事件，走完了它的一生：

1. 预处理，将hello.c #include的所有外部的头文件合并到一个hello.i 文件中；
2. 编译：将hello.i 编译成为汇编文件hello.s；
3. 汇编：汇编，将hello.s 会变成为可重定位目标文件hello.o；
4. 链接：hello.o 和动态链接库链接成为可执行目标程序hello；
5. 加载运行：shell中输入./hello 1170500913 xjy，终端为其新建进程（fork），把代码和数据加载入虚拟内存空间（execve），程序开始执行；
6. 执行每一步指令：在该进程被调度时，CPU 为其分配时间片，在一个时间片中，hello 享有CPU全部资源，PC寄存器一步一步地更新，CPU不断地取指，执行自己的控制逻辑流；
7. 访存：内存管理单元（MMU）将逻辑地址，一步步转化成物理地址；通过三级高速缓存，访问物理内存/磁盘中的数据;
8. 动态申请内存：比如printf 会调用malloc 向动态内存分配器申请堆中的内存；
9. 信号处理：进程时刻等待着信号，如果运行途中键入ctr-c ctr-z 则调用shell 的信号处理函数分别进行停止、挂起等操作；
10. 终止并被回收：shell父进程等待并回收子进程，内核删除为这个进程创建的所有数据结构（task\_struct等）。

**（结论0分，缺失 -1分，根据内容酌情加分）**

# 附件

hello.c：hello程序c语言源文件



hello.i：hello.c预处理生成的文本文件



hello.s：由hello.i编译得到的汇编文件



hello.o：hello.i汇编得到的可重定位目标文件



hello.o.elf：使用readelf读取hello.o得到的信息



hello.o.asm：使用objdump反汇编hello.o得到的反汇编代码



hello：由hello.o与动态链接库链接得到的可执行目标文件



hello.elf：使用readelf读取hello得到的信息



hello.asm：使用objdump反汇编hello得到的反汇编代码



**（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] 几种地址的含义：<https://blog.csdn.net/geekcome/article/details/6263057>

[2] 大卫R.奥哈拉伦，兰德尔E.布莱恩特. 深入理解计算机系统[M]. 机械工业出版社.2018.4

[3] printf 函数实现的深入剖析：https://blog.csdn.net/zhengqijun/article/details/72454714

**（参考文献0分，缺失 -1分）**