

**计算机系统**

**大作业**

题 目 程序人生-Hello’s P2P

专 业 计算机

学　　 号

班　　 级 1903002

学 生

指 导 教 师 郑贵滨

**计算机科学与技术学院**

**2021年6月**

**摘 要**

摘要是论文内容的高度概括，应具有独立性和自含性，即不阅读论文的全文，就能获得必要的信息。摘要应包括本论文的目的、主要内容、方法、成果及其理论与实际意义。摘要中不宜使用公式、结构式、图表和非公知公用的符号与术语，不标注引用文献编号，同时避免将摘要写成目录式的内容介绍。

**关键词：**hello；程序的一生；预处理；编译；汇编；链接；进程管理；存储管理；I/O管理

本文在Ubuntu系统下，通过介绍hello程序从编写到最终运行结束的过程来深入了解计算机系统，利用 linux 中的工具进行查看和解析，理解计算机内部机制，加深对计算机系统的理解。hello的一生主要经过预处理，编译，汇编，链接，再结合进程管理，存储管理，I/O管理完成程序的一生。

**（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 -](#_Toc75464192)

[1.1 Hello简介 - 4 -](#_Toc75464193)

[1.2 环境与工具 - 4 -](#_Toc75464194)

[1.3 中间结果 - 4 -](#_Toc75464195)

[1.4 本章小结 - 5 -](#_Toc75464196)

[第2章 预处理 - 6 -](#_Toc75464197)

[2.1 预处理的概念与作用 - 6 -](#_Toc75464198)

[2.2在Ubuntu下预处理的命令 - 6 -](#_Toc75464199)

[2.3 Hello的预处理结果解析 - 7 -](#_Toc75464200)

[2.4 本章小结 - 8 -](#_Toc75464201)

[第3章 编译 - 9 -](#_Toc75464202)

[3.1 编译的概念与作用 - 9 -](#_Toc75464203)

[3.2 在Ubuntu下编译的命令 - 9 -](#_Toc75464204)

[3.3 Hello的编译结果解析 - 10 -](#_Toc75464205)

[3.3.1 数据 - 10 -](#_Toc75464206)

[3.3.2 赋值=，逗号操作符，赋初值/不赋初值 - 11 -](#_Toc75464207)

[3.3.3 类型转换（隐式或显式） - 11 -](#_Toc75464208)

[3.3.4 算术操作 - 11 -](#_Toc75464209)

[3.3.5 关系操作 - 11 -](#_Toc75464210)

[3.3.6 数组/指针/结构操作 - 12 -](#_Toc75464211)

[3.3.7 控制转移 - 12 -](#_Toc75464212)

[3.3.8 函数操作 - 12 -](#_Toc75464213)

[3.4 本章小结 - 14 -](#_Toc75464214)

[第4章 汇编 - 15 -](#_Toc75464215)

[4.1 汇编的概念与作用 - 15 -](#_Toc75464216)

[4.2 在Ubuntu下汇编的命令 - 15 -](#_Toc75464217)

[4.3 可重定位目标elf格式 - 15 -](#_Toc75464218)

[4.4 Hello.o的结果解析 - 18 -](#_Toc75464219)

[4.5 本章小结 - 19 -](#_Toc75464220)

[第5章 链接 - 20 -](#_Toc75464221)

[5.1 链接的概念与作用 - 20 -](#_Toc75464222)

[5.2 在Ubuntu下链接的命令 - 20 -](#_Toc75464223)

[5.3 可执行目标文件hello的格式 - 20 -](#_Toc75464224)

[5.4 hello的虚拟地址空间 - 23 -](#_Toc75464225)

[5.5 链接的重定位过程分析 - 24 -](#_Toc75464226)

[5.6 hello的执行流程 - 25 -](#_Toc75464227)

[5.7 Hello的动态链接分析 - 26 -](#_Toc75464228)

[5.8 本章小结 - 27 -](#_Toc75464229)

[第6章 hello进程管理 - 28 -](#_Toc75464230)

[6.1 进程的概念与作用 - 28 -](#_Toc75464231)

[6.2 简述壳Shell-bash的作用与处理流程 - 28 -](#_Toc75464232)

[6.3 Hello的fork进程创建过程 - 29 -](#_Toc75464233)

[6.4 Hello的execve过程 - 29 -](#_Toc75464234)

[6.5 Hello的进程执行 - 30 -](#_Toc75464235)

[6.6 hello的异常与信号处理 - 32 -](#_Toc75464236)

[6.7本章小结 - 35 -](#_Toc75464237)

[第7章 hello的存储管理 - 36 -](#_Toc75464238)

[7.1 hello的存储器地址空间 - 36 -](#_Toc75464239)

[7.2 Intel逻辑地址到线性地址的变换-段式管理 - 36 -](#_Toc75464240)

[7.3 Hello的线性地址到物理地址的变换-页式管理 - 36 -](#_Toc75464241)

[7.4 TLB与四级页表支持下的VA到PA的变换 - 37 -](#_Toc75464242)

[7.5 三级Cache支持下的物理内存访问 - 37 -](#_Toc75464243)

[7.6 hello进程fork时的内存映射 - 38 -](#_Toc75464244)

[7.7 hello进程execve时的内存映射 - 38 -](#_Toc75464245)

[7.8 缺页故障与缺页中断处理 - 39 -](#_Toc75464246)

[7.9动态存储分配管理 - 40 -](#_Toc75464247)

[7.10本章小结 - 41 -](#_Toc75464248)

[第8章 hello的IO管理 - 42 -](#_Toc75464249)

[8.1 Linux的IO设备管理方法 - 42 -](#_Toc75464250)

[8.2 简述Unix IO接口及其函数 - 42 -](#_Toc75464251)

[8.3 printf的实现分析 - 43 -](#_Toc75464252)

[8.4 getchar的实现分析 - 44 -](#_Toc75464253)

[8.5本章小结 - 44 -](#_Toc75464254)

[结论 - 45 -](#_Toc75464255)

[附件 - 46 -](#_Toc75464256)

[参考文献 - 47 -](#_Toc75464257)

# 第1章 概述

## 1.1 Hello简介

P2P(Program to Process)：首先是编写高级语言程序文件hello.c，文本文件在Linux系统下经过预处理器cpp，编译器ccl，汇编器as，链接器ld的处理转最终形成一个可执行二进制目标文件hello。shell 通过fork子进程，分配内存资源，然后通过exevce函数去加载运行这个进程。

O2O(0 to 0)：执行该目标文件，首先shell中使用execve加载并执行该程序时，操作系统为程序分配一部分虚拟空间，将程序加载到虚拟空间所映射的物理内存空间中。然后执行目标程序。在程序运行结束后，shell回收创建的进程，释放进程的虚拟空间、删除相关数据结构。

## 1.2 环境与工具

硬件环境：处理器Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz；16GB RAM；1 TB SSD

软件环境：Windows 10 21H1；VirtualBox；Ubuntu 20.04 LTS

开发工具：EDB；GDB；CodeBlocks；vi/vim/gpedit；gcc；Vscode

## 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。

|  |  |
| --- | --- |
| **中间结果文件的名字** | **文件的作用** |
| hello.i | 修改了的源程序（文本） |
| hello.s | 汇编程序（文本） |
| hello.o | 可重定位目标程序（二进制） |
| hello | 可执行目标程序（二进制） |
| elf.txt | 可重定位目标ELF格式 |
| linked\_elf.txt | 可执行目标ELF格式 |
| objdump.txt | hello.o的反汇编代码 |
| objdump2.txt | hello的反汇编代码 |

## 1.4 本章小结

第一章主要对论文讨论的主要内容，首先对hello过程进行了总体概况，包括P2P、020的整个过程，然后介绍个人使用电脑的硬件环境、软件环境和开发工具，最后介绍了过程中产生的文件及其作用等。

**（第1章0.5分）**

# 第2章 预处理

## 2.1 预处理的概念与作用

预处理概念：预处理一般是指程序在编译系统处理过程中，预处理器(cpp)根据以符号“#”开头的命令，修改原始的C程序代码文本，主要是进行代码文本的替换工作，得到的结果再由编译器(ccl)进一步编译。用于在编译器处理程序之前预扫描代码，完成头文件包含、宏扩展、条件编译、行控制等操作。

预处理作用：这个过程并不对程序的源代码进行解析,但它把源代码分割或处理成为特定的单位，将预处理指令（以#开头）转化为实际代码中的内容，但只是单纯的替换和展开。例如，读取命令#include中包含的系统头文件并把它插入系统文本中，扩展所有用#define声明指定的宏。预处理过程还会删除程序中的注释和多余的空白字符。

## 2.2在Ubuntu下预处理的命令

命令行：Linux> cpp hello.c >hello.i

预处理命令及结果：



图1：预处理命令

预处理后源代码部分，注意到此时行数已经到了三千多行：

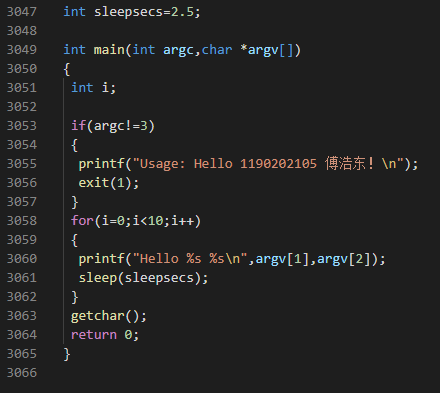


图2：预处理结果

预处理从头文件中插入的文本等：



图3：预处理插入内容

## 2.3 Hello的预处理结果解析

发现预处理之后，程序已经从原来的几十行变为了大约三千六十六行，并且源代码出现在最后，并且#include命令和注释等全都消失不见，推测之前的代码应该就是用头文件stdio.h\unistd.h\stdlib.h中的实际内容代替的对于命令行。其中包括了大量的相对路径、typedef类型创建名、extern关键字函数等。

## 2.4 本章小结

本章主要介绍了C语言的预处理过程，包括预处理的概念和作用，在Linux系统下预处理的命令，以及预处理的结果及生成文件hello.i，还有解析预处理。

**（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：编译是编译器（一种计算机程序）运行过程，会将某种编程语言写成的源代码（原始语言）通过词法语法分析之后转换成另一种编程语言（目标语言）。

编译的作用：编译主要做词法分析、语法分析、语义分析、优化后生成相应的汇编代码。在C语言的编译中，编译器将高级语言C转化为了机器码汇编语言，在这里将文本文件hello.i翻译成文本文件hello.s，它包含一个汇编语言程序。

注意：这儿的编译是指从 .i 到 .s 即预处理后的文件到生成汇编语言程序

## 3.2 在Ubuntu下编译的命令

命令行：Linux> gcc -S hello.i -o hello.s

编译过程截图：

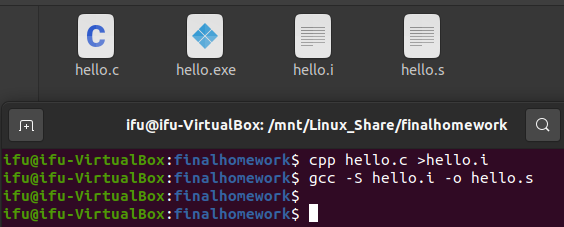


图4：编译命令

编译结果部分截图：

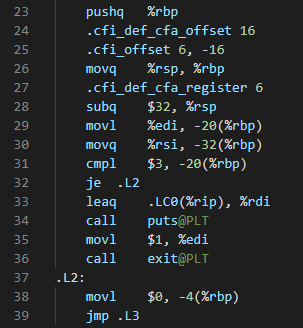


图5：编译结果

## 3.3 Hello的编译结果解析

此部分是重点，说明编译器是怎么处理C语言的各个数据类型以及各类操作的。应分3.3.1~ 3.3.x等按照类型和操作进行分析，**只要hello.s中出现的属于大作业PPT中P4给出的参考C数据与操作，都应解析**。

### 3.3.1 数据

1. **常量**

首先对于代码 if(argc!=3) printf("Hello %s %s\n",argv[1],argv[2]); 中存在常量3、1、2，类似于它们这些常数，被存放在代码段，所以保存在.text中。例如，对于第一句中的3，存放在 cmpl   $3, -20(%rbp) 其他也同理可得。

对于函数printf("Usage: Hello 1190202105 傅浩东！\n");中的字符串常量存放在.rodata节的.LC0中，printf("Hello %s %s\n",argv[1],argv[2]);中的字符串存放在.rodata节的.LC1中。如下所示：

.LC0:

.string "Usage: Hello 1190202105 \345\202\205\346\265\251\344\270\234\357\274\201"

.LC1:

    .string "Hello %s %s\n"

1. **变量**

**全局变量：**已初始化的全局和静态变量在.data节。在本节中，全局变量sleepsecs最开始赋值为2.5，但被隐式地转为int数据类型，所以会变为2 int sleepsecs=2.5; 所以在汇编代码中可以看到：

    .data

    .align 4

    .type   sleepsecs, @object

    .size   sleepsecs, 4

sleepsecs:

    .long   2

**局部变量：**原始代码中定义了局部变量 int I 储存在寄存器或者栈中，所以汇编代码有如下，循环前值为0的i被保存在 %rsp-4 的位置：

.L2:

    movl    $0, -4(%rbp)

    jmp .L3

**函数参数：**主函数main的两个参数int argc,char \*argv[] 分别都存放在栈中，由寄存器的偏移来分别表示。

    subq    $32, %rsp

    movl    %edi, -20(%rbp)

    movq    %rsi, -32(%rbp)

### 赋值=，逗号操作符，赋初值/不赋初值

在循环最开始有给循环条件变量i赋初值的操作：for(i=0;i<10;i++)

通过汇编语句 movl   $0, -4(%rbp) 给局部变量 i 赋初值0。

### 类型转换（隐式或显式）

int 类型全局变量 sleepsecs 赋值为 2.5 时进行了隐式类型转换（将浮点数类型转为整型），变量值变为 2：

int sleepsecs=2.5;

### 算术操作

在循环操作中，实现了i++操作 for(i=0;i<10;i++)

每次循环结束之后，对i进行一次自加，栈上对应的存储变量i的值加1：

addl    $1, -4(%rbp)

addl    $1, -4(%rbp) 实现为栈开辟空间；

### 关系操作

判断 argc 是否是3和循环终止条件用到了关系操作，如下所示，在源代码中有argc!=3和i<10 判断不等和小于。

    if(argc!=3)

    {

        printf("Usage: Hello 1190202105 傅浩东！\n");

        exit(1);

    }

    for(i=0;i<10;i++)

    {

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

        sleep(sleepsecs);

    }

它们对映的汇编指令有：

    cmpl    $3, -20(%rbp)

    je  .L2

比较3与%rbp-20位置的数值是否相等，不相等则跳转到.L2处。

    cmpl    $9, -4(%rbp)

    jle .L4

比较9与%rbp-4的操作数大小，若后者小于前者即9，则跳转到.L4处。

### 数组/指针/结构操作

最开始可以知道，主函数main的参数中有指针数组char \*argv[]，源代码中对数组的引用是输出 argv[1]和 argv[2]，利用在栈帧中位置，通过%rbp-16和%rbp-24，分别得到 argv[1]和 argc[2]两个字符串。

    movl    %edi, -20(%rbp)//argc

    movq    %rsi, -32(%rbp)//argv

### 控制转移

if(argc!=3)

for(i=0;i<10;i++)

上述原函数中两个部分使用了控制转移，这部分内容在关系操作部分已经提及，就不再赘述。

### 函数操作

主要从函数传递（地址/值）、函数调用（）、函数返回return三个方面来对以下函数进行编译结果解析。

**Main函数：**

参数传递：int argc,char \*argv[] 分别储存在寄存器%rdi和%rsi中

函数调用：被系统函数调用

函数返回：函数return 0，返回值储存在寄存器%eax中

**Printf函数：**

参数传递：首先对于puts，将.LC0作为参数传递，即只传入了字符串首地址；对于printf，将栈中的两个数据传给printf，即argv[1]和argv[2]的地址，另外还传入了.LC0参数，即字符串首地址。

函数调用：判断argc!=3，若该不等式成立，则调用printf；在for循环中，即i在范围0到9之间都调用函数printf。但是它们对应汇编指令分别有puts和printf。

函数返回：暂时未知

汇编指令：call puts@PLT以及call printf@PLT

对于第一条 printf("Usage: Hello 1190202105 傅浩东！\n");有如下汇编：

    cmpl    $3, -20(%rbp)

    je  .L2

    leaq    .LC0(%rip), %rdi

    call    puts@PLT

对于printf("Hello %s %s\n",argv[1],argv[2]);有：

.L4:

    movq    -32(%rbp), %rax

    addq    $16, %rax

    movq    (%rax), %rdx

    movq    -32(%rbp), %rax

    addq    $8, %rax

    movq    (%rax), %rax

    movq    %rax, %rsi

    leaq    .LC1(%rip), %rdi

    movl    $0, %eax

    call    printf@PLT

**Exit函数：**

参数传递：源代码exit(1);传递的参数为1，从下面的汇编代码可知该参数储存在寄存器%edi之中。

    movl    $1, %edi

    call    exit@PLT

函数调用：判断argc!=3，若该不等式成立，则在调用printf之后调用函数exit。

函数返回：暂时不知

**Sleep函数：**

参数传递：根据源代码易知，将全局变量sleepsecs作为参数传递给了sleep函数，从汇编指令可知应该是将sleepsecs储存在%edi中来传递。

    movl    sleepsecs(%rip), %eax

    movl    %eax, %edi

    call    sleep@PLT

函数调用：循环条件成立时，即i在0到9之间，每一次循环都在调用printf之后调用函数sleep。

函数返回：暂时不知。

**Getchar函数：**

参数传递：无

函数调用：main函数return 0之前调用。

函数返回：暂时不知

    call    getchar@PLT

## 3.4 本章小结

本章首先介绍了编译的概念与作用，接着是编译命令和结果，重点在对hello编译结果解析。分别对数据、赋值、隐式类型转换、算术操作、关系操作、数组/指针操作、控制转移、函数操作等多个方面来对编译结果进行了详细的解释，也对 hello.s 其中的语句进行分析，找出指令与源代码的对应情况等。

**（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 hello.s 翻译成机器语言指令，把这些指令打包成一种叫做可重定位目标代码 (relocatable object program)的格式，并将结果保存在目标文件 hello.o 中。hello.o 文件是一个二进制文件，包含所有指令的二进制表示，但是还没有填入全局值的地址，如果在文本编辑器中打开 hello.o文件，将看到一堆乱码。

**汇编的作用：**汇编就是将.s程序翻译成机器语言指令，并将这些指令打包为可重定位程序格式，保存在二进制文件.o中。便于机器在此后的链接与运行。

注意：这儿的汇编是指从 .s 到 .o 即编译后的文件到生成机器语言二进制程序的过程。

## 4.2 在Ubuntu下汇编的命令

命令行：Linux> gcc -c hello.s -o hello.o

汇编过程截图：

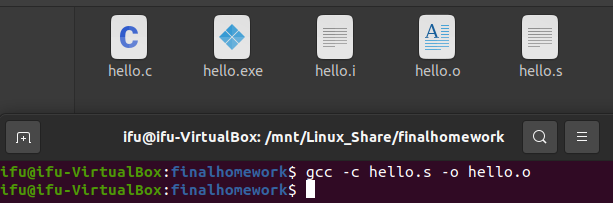


图6：汇编命令

## 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。

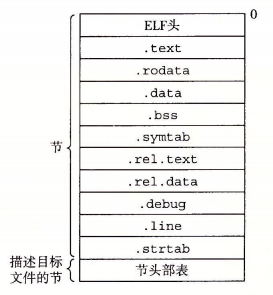


图7：可重定位目标ELF格式

命令行：Linux> readelf -a hello.o >elf.txt

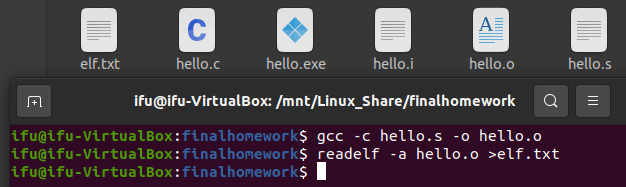


图8：ELF命令

**ELF header：**以一个16字节的序列开始，这个序列描述了生成该文件的系统的字的大小和字节顺序。ELF头剩下的部分包含帮助链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、目标文件的类型（这里是可重定位）、机器类型（X86-64）、节头部表（section header table）的文件偏移，以及节头部表中条目的大小和数量。不同节的位置和大小是由节头部表描述的，其中目标文件中每个节都有一个固定大小的条目。如下所示：

ELF Header:

  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00

  Class:                             ELF64

  Data:                              2's complement, little endian

  Version:                           1 (current)

  OS/ABI:                            UNIX - System V

  ABI Version:                       0

  Type:                              REL (Relocatable file)

  Machine:                           Advanced Micro Devices X86-64

  Version:                           0x1

  Entry point address:               0x0

  Start of program headers:          0 (bytes into file)

  Start of section headers:          1240 (bytes into file)

  Flags:                             0x0

  Size of this header:               64 (bytes)

  Size of program headers:           0 (bytes)

  Number of program headers:         0

  Size of section headers:           64 (bytes)

  Number of section headers:         14

  Section header string table index: 13

**节：**夹在ELF头和节头部表之间的都是节。典型的ELF可重定位目标文件包含如下几个节：.text: 已编译程序的机器代码。.rodata: 只读数据。.data: 已初始化的全局和静态C变量。.bss: 未初始化的全局和静态C变量，以及所有被初始化为0的全局或静态变量。.symtab：符号表，存放程序中定义和引用的函数和全局变量的信息。.rel.text: —个.text 节中位置的列表，当链接器把这个目标文件和其他文件组合时，需要修改这些位置。.rel.data: 被模块引用或定义的所有全局变量的重定位信息。.debug: 个调试符号表，只有以-g选项调用编译器驱动程序时才会得到这张表。.line: 原始C源程序中的行号和.text 节中机器指令之间的映射，以-g选项调用编译器驱动程序时才会得到。.strtab:字符串表，内容包括 .symtab和 .debug节中的符号表，以及节头部中的令名字。如下是节头部表：

Section Headers:

  [Nr] Name              Type             Address           Offset

       Size              EntSize          Flags  Link  Info  Align

  [ 0]                   NULL             0000000000000000  00000000

       0000000000000000  0000000000000000           0     0     0

  [ 1] .text             PROGBITS         0000000000000000  00000040

       0000000000000085  0000000000000000  AX       0     0     1

  [ 2] .rela.text        RELA             0000000000000000  00000388

       00000000000000c0  0000000000000018   I      11     1     8

  [ 3] .data             PROGBITS         0000000000000000  000000c8

       0000000000000004  0000000000000000  WA       0     0     4

  [ 4] .bss              NOBITS           0000000000000000  000000cc

       0000000000000000  0000000000000000  WA       0     0     1

  [ 5] .rodata           PROGBITS         0000000000000000  000000d0

       0000000000000032  0000000000000000   A       0     0     8

  [ 6] .comment          PROGBITS         0000000000000000  00000102

       000000000000002b  0000000000000001  MS       0     0     1

  [ 7] .note.GNU-stack   PROGBITS         0000000000000000  0000012d

       0000000000000000  0000000000000000           0     0     1

  [ 8] .note.gnu.propert NOTE             0000000000000000  00000130

       0000000000000020  0000000000000000   A       0     0     8

  [ 9] .eh\_frame         PROGBITS         0000000000000000  00000150

       0000000000000038  0000000000000000   A       0     0     8

  [10] .rela.eh\_frame    RELA             0000000000000000  00000448

       0000000000000018  0000000000000018   I      11     9     8

  [11] .symtab           SYMTAB           0000000000000000  00000188

       00000000000001b0  0000000000000018          12    10     8

  [12] .strtab           STRTAB           0000000000000000  00000338

       000000000000004d  0000000000000000           0     0     1

  [13] .shstrtab         STRTAB           0000000000000000  00000460

       0000000000000074  0000000000000000           0     0     1

除此以外，在本例中主要还有如下两个部分：首先是重定位节.rela.text和.rela.eh\_frame，其次就是符号表.symtab。对于重定位节，在链接时需要对其进行修改，通过偏移量等信息计算出正确的地址。

Relocation section '.rela.text' at offset 0x388 contains 8 entries:

  Offset          Info           Type           Sym. Value    Sym. Name + Addend

00000000001c  000500000002 R\_X86\_64\_PC32     0000000000000000 .rodata - 4

000000000021  000d00000004 R\_X86\_64\_PLT32    0000000000000000 puts - 4

00000000002b  000e00000004 R\_X86\_64\_PLT32    0000000000000000 exit - 4

000000000054  000500000002 R\_X86\_64\_PC32     0000000000000000 .rodata + 21

00000000005e  000f00000004 R\_X86\_64\_PLT32    0000000000000000 printf - 4

000000000064  000a00000002 R\_X86\_64\_PC32     0000000000000000 sleepsecs - 4

00000000006b  001000000004 R\_X86\_64\_PLT32    0000000000000000 sleep - 4

00000000007a  001100000004 R\_X86\_64\_PLT32    0000000000000000 getchar - 4

Relocation section '.rela.eh\_frame' at offset 0x448 contains 1 entry:

  Offset          Info           Type           Sym. Value    Sym. Name + Addend

000000000020  000200000002 R\_X86\_64\_PC32     0000000000000000 .text + 0

Symbol table '.symtab' contains 18 entries:

   Num:    Value          Size Type    Bind   Vis      Ndx Name

     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND

     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS hello.c

     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1

     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3

     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4

     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5

     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    7

     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    8

     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    9

     9: 0000000000000000     0 SECTION LOCAL  DEFAULT    6

    10: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 sleepsecs

    11: 0000000000000000   133 FUNC    GLOBAL DEFAULT    1 main

    12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND \_GLOBAL\_OFFSET\_TABLE\_

    13: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND puts

    14: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND exit

    15: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf

    16: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND sleep

    17: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND getchar

## 4.4 Hello.o的结果解析

命令行：Linux> objdump -d -r hello.o >objdump.s

分析hello.o的反汇编，与第3章的 hello.s进行对照分析，发现有如下区别：

1. 分支转移：在.s文件中依靠.L+字段来确定位置的，而反汇编代码中通过间接寻址来跳转到相对偏移地址。
2. 函数调用：hello.s中用 call+函数名来实现对该函数的调用，而反汇编代码中callq使用的是相对偏移地址。但是现在该地址全部为0，将其写在重定位节，链接器后续进行重定位确定真正的地址。
3. 全局变量使用：同样待重定位。
4. 指令表示：在反汇编代码中，省略了许多与字节大小相关的后缀，例如“q”、“l”等；但是call指令又变为了callq（强调这是x86-64版本）。
5. 数据表示：在hello.s文件中数据一般用十进制来表示，而在反汇编代码中是用十六进制来表示的。

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。

## 4.5 本章小结

本章主要介绍了汇编过程及其结果、可重定位目标ELF格式、以及hello1.o反汇编。经过汇编器汇编语言转化为机器语言指令，打包为二进制可重定位目标文件hello.o。对ELF分析，以及对比hello.s和hello.o反汇编结果，发现汇编过程为链接做了不小的准备，接下来自然要分析链接，进一步了解P2P过程。

**（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（linking）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（复制）到内存并执行。链接可以执行于编，译时（compile time），也就是在源代码被翻译成机器代码时；也可以执行于加栽时（load time），也就是在程序被加载器（loader）加载到内存并执行时；甚至执行于运行时（runtime），也就是由应用程序来执行。在早期的计算机系统中，链接是手动执行的。在现代系统中，链接是由叫做链接器（linker）的 程序自动执行的。

**链接的作用：**链接在软件开发中扮演着一个关键的角色，因为它们使得分离编译（separate compilation）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。

注意：这儿的链接是指从 hello.o 到hello生成过程。

## 5.2 在Ubuntu下链接的命令

Ld链接命令行：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

过程截图展示：

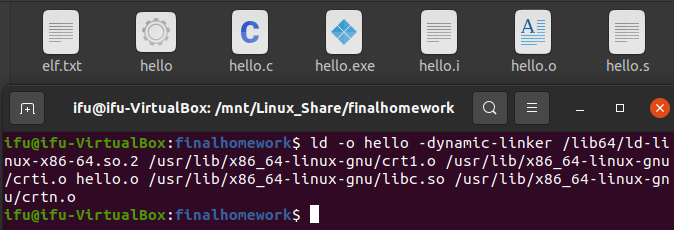


图9：ld链接命令

## 5.3 可执行目标文件hello的格式

Hello（可执行目标文件）的ELF格式：

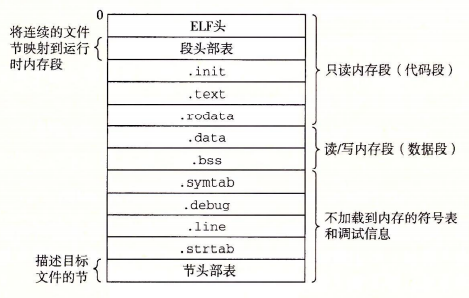


图10：可执行目标文件ELF格式

命令行：Linux> readelf -a hello >linked\_elf.txt

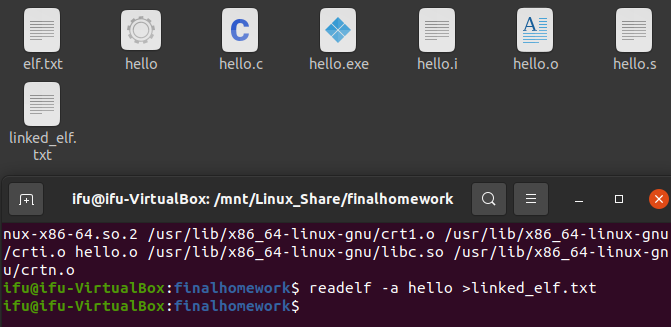


图11：ELF获取命令

ELF header：

  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00

  Class:                             ELF64

  Data:                              2's complement, little endian

  Version:                           1 (current)

  OS/ABI:                            UNIX - System V

  ABI Version:                       0

  Type:                              EXEC (Executable file)

  Machine:                           Advanced Micro Devices X86-64

  Version:                           0x1

  Entry point address:               0x4010d0

  Start of program headers:          64 (bytes into file)

  Start of section headers:          14200 (bytes into file)

  Flags:                             0x0

  Size of this header:               64 (bytes)

  Size of program headers:           56 (bytes)

  Number of program headers:         12

  Size of section headers:           64 (bytes)

  Number of section headers:         27

  Section header string table index: 26

节头部表（包括各段的起始地址，大小等信息）：

  [Nr] Name              Type             Address           Offset

       Size              EntSize          Flags  Link  Info  Align

  [ 0]                   NULL             0000000000000000  00000000

       0000000000000000  0000000000000000           0     0     0

  [ 1] .interp           PROGBITS         00000000004002e0  000002e0

       000000000000001c  0000000000000000   A       0     0     1

  [ 2] .note.gnu.propert NOTE             0000000000400300  00000300

       0000000000000020  0000000000000000   A       0     0     8

  [ 3] .note.ABI-tag     NOTE             0000000000400320  00000320

       0000000000000020  0000000000000000   A       0     0     4

  [ 4] .hash             HASH             0000000000400340  00000340

       0000000000000034  0000000000000004   A       6     0     8

  [ 5] .gnu.hash         GNU\_HASH         0000000000400378  00000378

       000000000000001c  0000000000000000   A       6     0     8

  [ 6] .dynsym           DYNSYM           0000000000400398  00000398

       00000000000000c0  0000000000000018   A       7     1     8

  [ 7] .dynstr           STRTAB           0000000000400458  00000458

       0000000000000057  0000000000000000   A       0     0     1

  [ 8] .gnu.version      VERSYM           00000000004004b0  000004b0

       0000000000000010  0000000000000002   A       6     0     2

  [ 9] .gnu.version\_r    VERNEED          00000000004004c0  000004c0

       0000000000000020  0000000000000000   A       7     1     8

  [10] .rela.dyn         RELA             00000000004004e0  000004e0

       0000000000000030  0000000000000018   A       6     0     8

  [11] .rela.plt         RELA             0000000000400510  00000510

       0000000000000078  0000000000000018  AI       6    21     8

  [12] .init             PROGBITS         0000000000401000  00001000

       000000000000001b  0000000000000000  AX       0     0     4

  [13] .plt              PROGBITS         0000000000401020  00001020

       0000000000000060  0000000000000010  AX       0     0     16

  [14] .plt.sec          PROGBITS         0000000000401080  00001080

       0000000000000050  0000000000000010  AX       0     0     16

  [15] .text             PROGBITS         00000000004010d0  000010d0

       0000000000000135  0000000000000000  AX       0     0     16

  [16] .fini             PROGBITS         0000000000401208  00001208

       000000000000000d  0000000000000000  AX       0     0     4

  [17] .rodata           PROGBITS         0000000000402000  00002000

       000000000000003a  0000000000000000   A       0     0     8

  [18] .eh\_frame         PROGBITS         0000000000402040  00002040

       00000000000000fc  0000000000000000   A       0     0     8

  [19] .dynamic          DYNAMIC          0000000000403e50  00002e50

       00000000000001a0  0000000000000010  WA       7     0     8

  [20] .got              PROGBITS         0000000000403ff0  00002ff0

       0000000000000010  0000000000000008  WA       0     0     8

  [21] .got.plt          PROGBITS         0000000000404000  00003000

       0000000000000040  0000000000000008  WA       0     0     8

  [22] .data             PROGBITS         0000000000404040  00003040

       0000000000000008  0000000000000000  WA       0     0     4

  [23] .comment          PROGBITS         0000000000000000  00003048

       000000000000002a  0000000000000001  MS       0     0     1

  [24] .symtab           SYMTAB           0000000000000000  00003078

       00000000000004c8  0000000000000018          25    30     8

  [25] .strtab           STRTAB           0000000000000000  00003540

       0000000000000150  0000000000000000           0     0     1

  [26] .shstrtab         STRTAB           0000000000000000  00003690

       00000000000000e1  0000000000000000           0     0     1

其余部分基本就是上述表的具体内容，就不再一一列举。

## 5.4 hello的虚拟地址空间

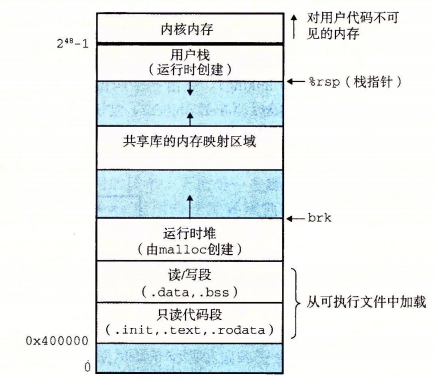


图12：hello虚拟地址空间基本结构

使用edb加载hello，查看本进程的虚拟地址空间各段信息。

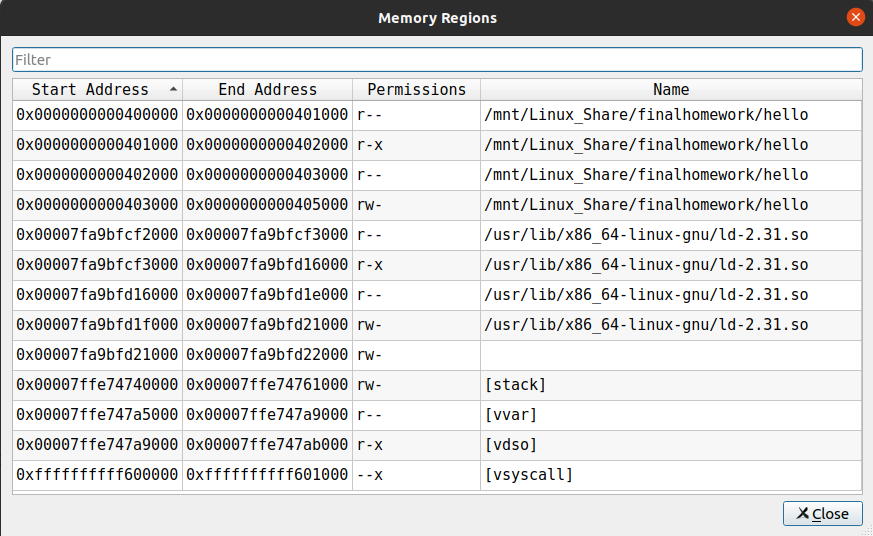
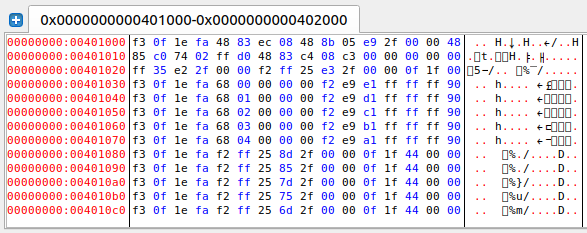


图13：Memory Regions查看结果

通过edb中工具Memory Regions查看虚拟空间各段的储存信息，可以知道从0x400000到0x401000只能读，对映节有.interp/.note.gnu.propert/……/.rela.plt；相应的0x401000到0x402000能读和执行，对应节有.init/.plt/.plt.scc/.text/.fini；从0x402000到0x403000只能读，对应节有.rodata/.eh\_frame；从0x403000到0x405000能读写，对应节有.dynamic/.got/.got.plt/.data。如下是0x401000到0x402000部分的对照：



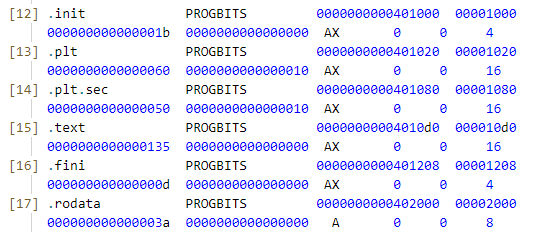


图14、15：对映节查看

## 5.5 链接的重定位过程分析

**命令行**：Linux> objdump -d -r hello >objdump2.s

**分析比较hello与hello.o反汇编结果的不同：**

1. 函数增加：相较于objdump.s只有main函数，objdump2.s还包含了函数puts ，printf ，getchar，exit，sleep 等等。
2. 节增加：在hello中增加了例如.plt，.plt.sec，.fini等节。
3. 重定位：在hello中发现有lea和call指令的操作数被填充了地址，进行了重定位，其中包括函数的虚拟地址以及储存的字符串的地址（.rodata节的地址与偏移量所决定）和全局变量sleepsecs的虚拟地址。

**链接的过程**（主要是静态链接）：

1. 符号解析(symbol resolution)：目标文件定义和引用符号，每个符号对应于一个函数、一个全局变量或一个静态变量（即C语言中任何以static属性声明的变量）。符号解析的目的是将每个符号引用正好和一个符号定义关联起来。
2. 重定位（relocation)：编译器和汇编器生成从地址0开始的代码和数据节。链接器通过把每个符号定义与一个内存位置关联起来，从而重定位这些节，然后修改所有对这些符号的引用，使得它们指向这个内存位置。链接器使用汇编器产生的重定位条目（relocation entry）的详细指令，不加甄别地执行这样的重定位。

**分析hello中对重定位项目的重定位方式：**

一旦链接器完成了符号解析这一步，就把代码中的每个符号引用和正好一个符号定义关联起来。此时，链接器就知道它的输人目标模块中的代码节和数据节的确切大小。

1. 重定位节和符号定义。链接器将所有相同类型的节合并为同一类型的新的聚合节。然后，链接器将运行时内存地址赋给新的聚合节，赋给输人模块定义的每个节，以及赋给输人模块定义的每个符号。这样，程序中的每条指令和全局变量都有唯一的运行时内存地址了。
2. 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，使得它们指向正确的运行时地址。要执行这一步，链接器依赖于可重定位目标模块中称为重定位条目的数据结构。

## 5.6 hello的执行流程

使用edb执行hello截图：

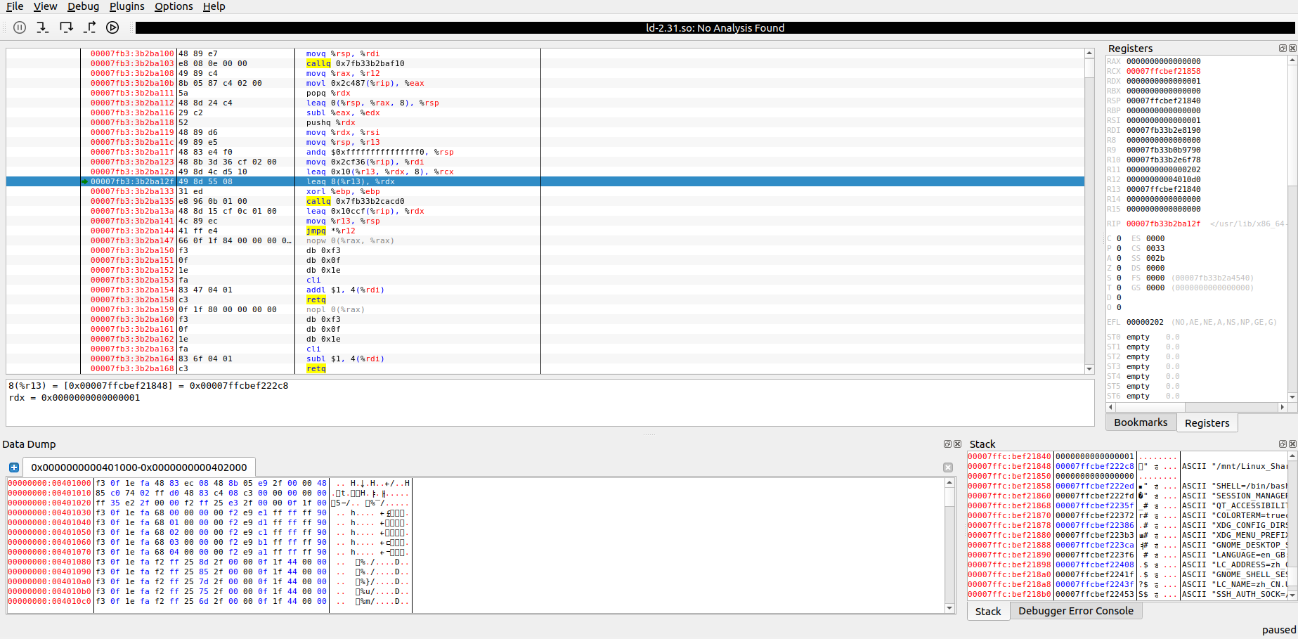


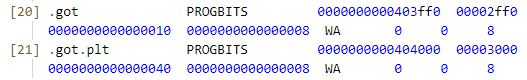
图16：edb执行hello结果

说明从加载hello到\_start，到call main,以及程序终止的所有过程其调用与跳转的各个子程序名或程序地址，如下所示：

|  |  |
| --- | --- |
| **程序地址** | **程序名** |
| 0x00000000004010d0 | \_start |
| 0x00007efda717bfc0 | \_\_libc\_start\_main |
| 0x0000000000401190 | \_\_libc\_csu\_init |
| 0x0000000000401000 | \_init |
| 0x0000000000401105 | Main |
| 0x0000000000401080 | puts@plt |
| 0x00000000004010b0 | Exit@plt |
| 0x0000000000401090 | printf@plt |
| 0x00000000004010c0 | sleep@plt |
| 0x00000000004010a0 | getchar@plt |
| 0x0000000000401200 | \_\_libc\_csu\_fini |
| 0x0000000000401208 | \_fini |

## 5.7 Hello的动态链接分析

首先可以找到.got和.got.plt两个节起始地址偏移量等。



通过edb调试，在dl\_init前后，有如下变化：

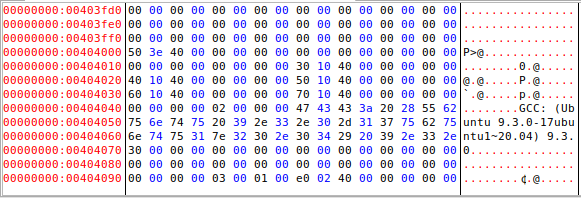


图17：dl\_init前

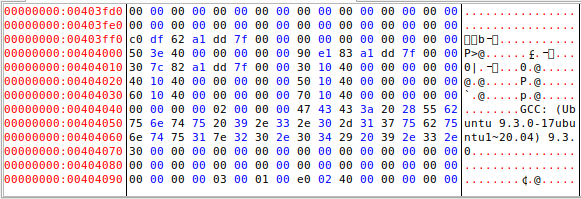


图18：dl\_init后

在形成可执行程序时，发现引用了一个外部的函数，检查动态链接库，发现这个函数名是一个动态链接符号，此时可执行程序就不对这个符号进行重定位，而把这个过程留到装载时再进行。

初始时每个got条目都指向对应plt条目的第二条指令。当库函数被调用后，链接器修改got。下一次调用plt时，指向的就是正确的内存地址。plt就能跳转到正确的区域。

## 5.8 本章小结

本章介绍了hello.o到hello的链接过程。主要包括链接的概念和作用；链接过程（虚拟地址空间、链接具体流程、还介绍了动态链接过程）；以及链接结果（可执行目标文件hello的ELF格式以及与hello.o的区别、了解了和hello的执行流程等）。

**（第5章1分）**

# 第6章 hello进程管理

## 6.1 进程的概念与作用

**进程概念：**程序是指令、数据及其组织形式的描述，进程是程序的实体。进程的经典定义就是一个执行中程序的实例。

**进程作用：**在现代系统上运行一个程序时，我们会得到一个假象，就好像我们的程序是系统中 当前运行的唯一的程序一样。我们的程序好像是独占地使用处理器和内存。处理器就好像是无间断地一条接一条地执行我们程序中的指令。最后，我们程序中的代码和数据好像是系统内存中唯一的对象。这些假象都是通过进程的概念提供给我们的。

## 6.2 简述壳Shell-bash的作用与处理流程

**壳的作用：**Shell 是系统的用户界面，提供了用户与内核进行交互操作的一种接口，Shell 是一种命令行解释器，其读取用户输入的字符串命令，解释并把它们送到内核。它是一种特殊的应用程序，介于系统调用/库和应用程序之间，提供了运行其他程序的接口。

**壳的处理流程：**

shell 先分词，判断命令是否为内部命令，如果不是，则寻找可执行文件进行执行，重复这个流程：

1. Shell 首先从命令行中找出特殊字符（元字符），在将元字符翻译成间隔符 号。元字符将命令行划分成小块 tokens。Shell 中的元字符如下所示：SPACE , TAB , NEWLINE , & , ; , ( , ) ,< , > , |
2. 程序块 tokens 被处理，检查看他们是否是 shell 中所引用到的关键字。
3. 当程序块 tokens 被确定以后，shell 根据 aliases 文件中的列表来检查命令的第一个单词。如果这个单词出现在 aliases 表中，执行替换操作并且处理过程回到第一步重新分割程序块 tokens。
4. Shell 对~符号进行替换。
5. Shell 对所有前面带有$符号的变量进行替换。
6. Shell 将命令行中的内嵌命令表达式替换成命令；他们一般都采用 $(command)标记法。
7. Shell 计算采用$(expression)标记的算术表达式。
8. Shell 将命令字符串重新划分为新的块 tokens。这次划分的依据是栏位分割符号，称为 IFS。缺省的 IFS 变量包含有：SPACE , TAB 和换行符号。
9. Shell 执行通配符\* ? [ ]的替换。
10. Shell 把所有处理的结果中用到的注释删除，並且按照下面的顺序实 行命令的检查：

I. 内建的命令

II. shell 函数（由用户自己定义的）

III. 可执行的脚本文件（需要寻找文件和 PATH 路径）

1. 在执行前的最后一步是初始化所有的输入输出重定向。
2. 最后，执行命令。

## 6.3 Hello的fork进程创建过程

父进程通过调用 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，包括代码和数据段、堆、共享库以及用户栈。子进程还获得与父进程任何打开文件描述符相同的副本，这就意味着当父进程调用 fork 时，子进程可以读写父进程中打开的任何文件。父进程和新创建的子进程之间最大的区别在于它们有不同的 PID。具体过程如下：

1. 给新进程分配一个标识符
2. 在内核中分配一个PCB,将其挂在PCB表上
3. 复制它的父进程的环境（PCB中大部分的内容）
4. 为其分配资源（程序、数据、栈等）
5. 复制父进程地址空间里的内容（代码共享，数据写时拷贝）
6. 将进程置成就绪状态，并将其放入就绪队列，等待CPU调度。

## 6.4 Hello的execve过程

execve函数在当前进程的上下文中加载并运行一个新程序。

execve函数加载并运行可执行目标文件hello，且带参数列表argv和环境变量 列表envp。只有当出现错误时，例如找不到hello，才会返回到调用程序。所以，与fork一次调用返回两次不同，execve调用一次并从不返回。

在execve加载了hello之后调用启动代码，启动代码设置栈，并=将控制传递给新程序的主函数，该主函数如下原型：int main(int argc, char \*\*argv, char \* \*\*envp)或是等价的int main(int argc, char \*argv[], char \*\*envp[])。

当main开始执行时，用户栈的组织结构如图所示。让我们从栈底（高地址）往栈顶（低地址）依次看一看。首先是参数和环境字符串。栈往上紧随其后的是以null结尾的指针数组，其中每个指针都指向找中的一个环境变量字符串。全局变量environ指向这些指针中的第一个envp[0]。紧随环境变量数组之后的是以null 结尾的 argv[]数组，其中每个元素都指向钱中的一个参数字符串。在找的顶部是系统启动函数libc\_start\_main的栈帧。

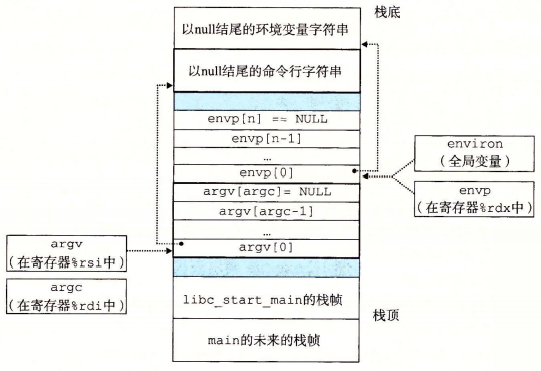


图19：用户栈的组织结构

## 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。

**进程上下文信息：**内核为每个进程维持一个上下文（context）。上下文就是内核重新启动一个被抢占的进程所需的状态。它由一些对象的值组成，这些对象包括通用目的寄存器、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，比如描述地址空间的页表、包含有关当前进程信息的进程表，以及包含进程已打开文件的信息的文件表。

**进程时间片：**时间片即CPU分配给各个程序的时间，每个线程被分配一个时间段，称作它的时间片，即该进程允许运行的时间，使各个程序从表面上看是同时进行的。如果在时间片结束时进程还在运行，则CPU将被剥夺并分配给另一个进程。如果进程在时间片结束前阻塞或结束，则CPU当即进行切换。而不会造成CPU资源浪费。在宏观上：我们可以同时打开多个应用程序，每个程序并行不悖，同时运行。但在微观上：由于只有一个CPU，一次只能处理程序要求的一部分，如何处理公平，一种方法就是引入时间片，每个程序轮流执行。

**进程调度的过程：**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。这种决策就叫做调度( scheduling)，是由内核中称为调度器(scheduler)的代码处理的。当内核选择一个新的进程运行时，我们说内核调度了这个进程。在内核调度了一个新的进程运行后，它就抢占当前进程，并使用一种称为上下文切换的机制来将控制转移到新的进程，上下文切换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。

当内核代表用户执行系统调用时，可能会发生上下文切换。如果系统调用因为等待某个事件发生而阻塞，那么内核可以让当前进程休眠，切换到另一个进程。比如，如果一个read系统调用需要访问磁盘，内核可以选择执行上下文切换，运行另外一个进程，而不是等待数据从磁盘到达。另一个示例是sleep系统调用，它显式地请求让调用进程休眠。一般而言，即使系统调用没有阻塞，内核也可以决定执行上下文切换，而不是将控制返回给调用进程。

中断也可能引发上下文切换。比如，所有的系统都有某种产生周期性定时器中断的机制，通常为每1毫秒或每10毫秒。每次发生定时器中断时，内核就能判定当前进程已经运行了足够长的时间，并切换到一个新的进程。

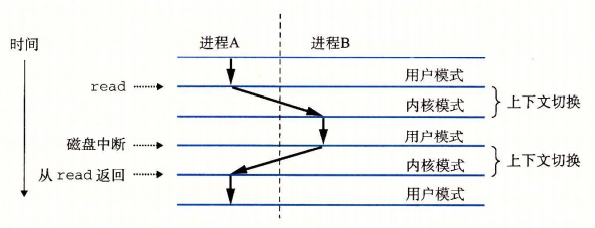


图20：进程切换

**用户态与核心态转换：**从一个进程到另一个进程的转换是由操作系统内核（kernel）管理的。内核是操作系统代码常驻主存的部分。当应用程序需要操作系统的某些操作时，比如读写文件，它就执行一条特殊的系统调用（system call）指令，将控制权传递给内核。然后内核执行被请求的操作并返回应用程序。注意，内核不是一个独立的进程。相反，它是系统管理全部进程所用代码和数据结构的集合。

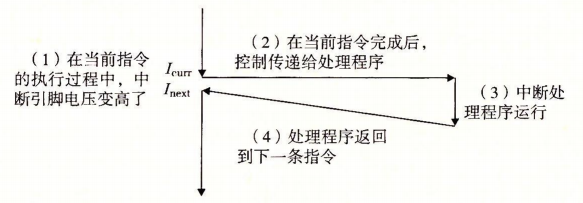
## 6.6 hello的异常与信号处理

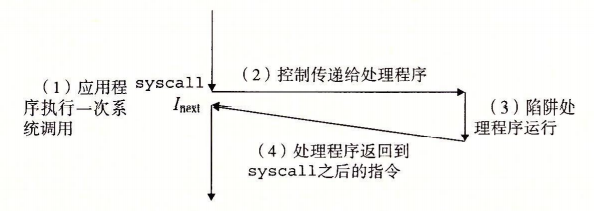
hello执行过程中会出现中断、陷进、故障、终止四类异常。

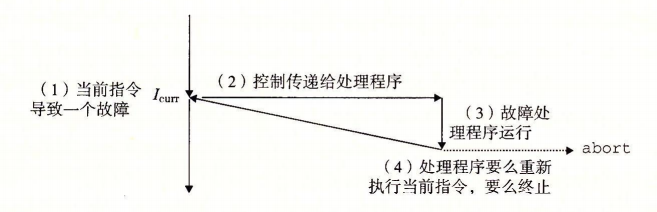


图21：异常类型

处理方式：







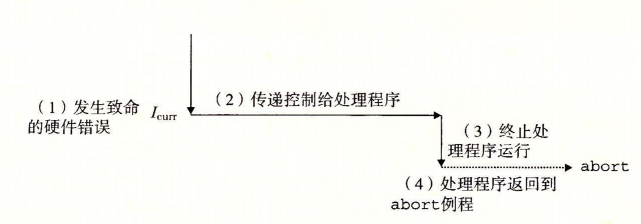


图22-24：四种异常处理过程

程序运行过程中不停乱按，包括回车：

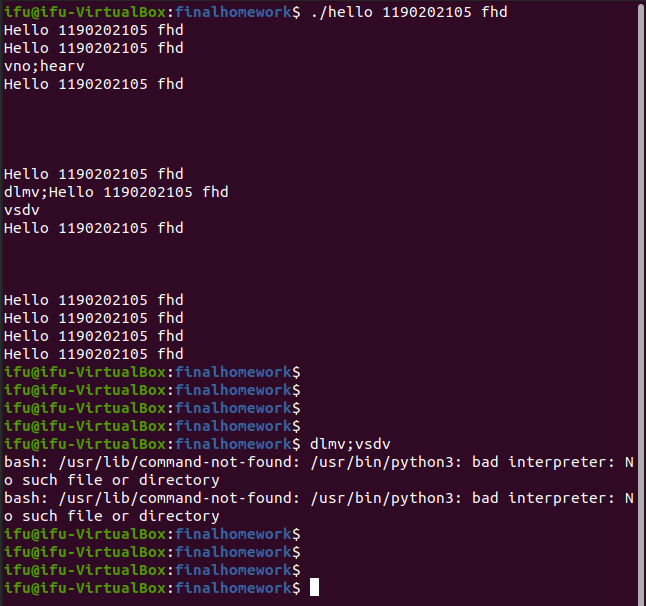


图25：运行过程乱按

按Ctrl-C截图

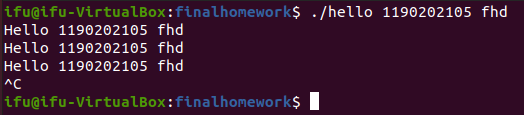
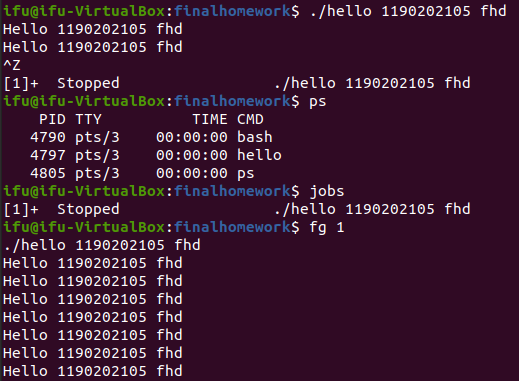
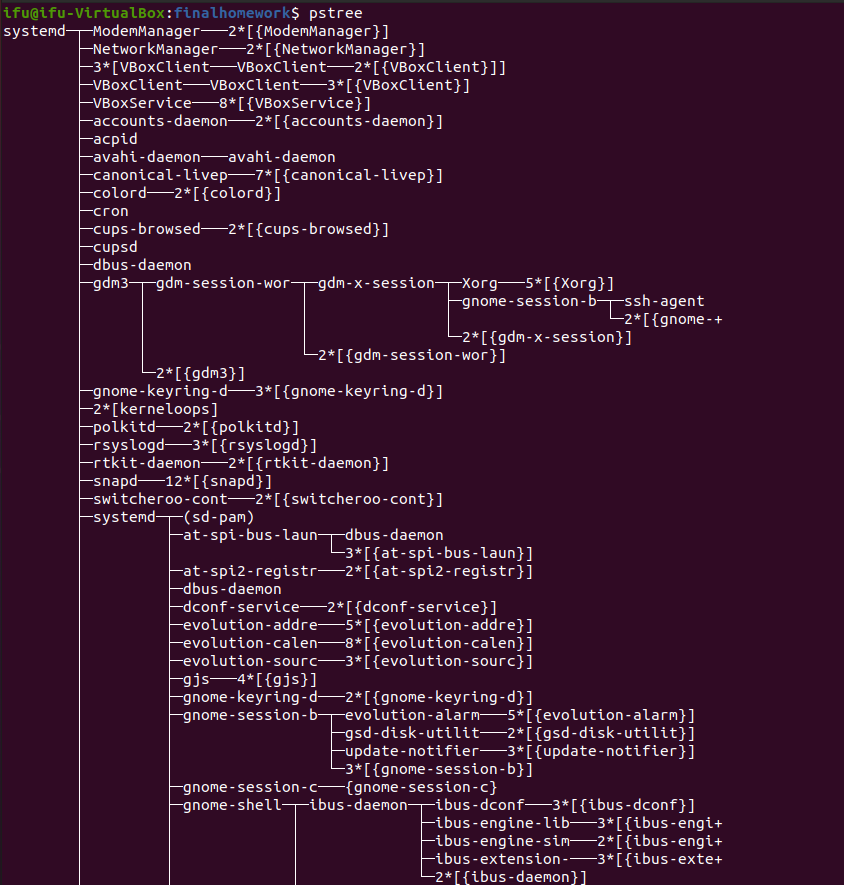


图26：按Ctrl+C

按Ctrl-Z之后，运行ps jobs pstree fg kill 等命令，运行截屏如下：





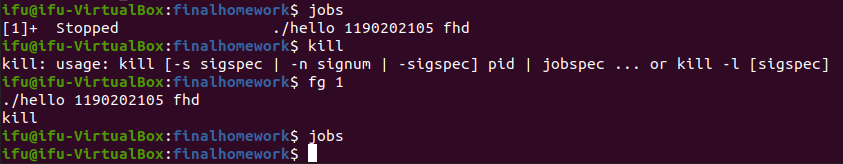


图27、28：Ctrl+Z之后各种指令结果

说明异常与信号的处理：

运行过程中乱按，输入被写入缓存区，不会对运行造成太大影响（除开某些特殊按键）。按Ctrl+C进程收到SIGINT信号，结束。按Ctrl+Z进程收到SIGSTP信号，进程被挂起，通过ps、jobs查看状态，使用fg命令将其调回前台，pstree命令以树状图显示进程间的关系。

## 6.7本章小结

本章主要介绍了进程的执行过程。具体包括概念与作用、Shell的作用与处理流程， fork和exccvc的执行过程，也从进程时间片、上下文切换用户模式和内核模式等方面介结了hello程序执行时的调度问题，异常与信号处理等。

**（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：**一个项目（[存储单元](https://en.wikipedia.org/wiki/Computer_data_storage)、存储元件、网络主机）从正在执行的[应用程序](https://en.wikipedia.org/wiki/Application_program)的角度看来驻留的地址。是指由程序 hello 产生的与段相关的偏移地址部分。

**线性地址：**是逻辑地址到物理地址变换之间的中间层。程序代码会产生逻辑地址，或者说是段中的偏移地址，加上相应段的基地址就生成了一个线性地址。

**虚拟地址：**在支持[虚拟内存](https://en.wikipedia.org/wiki/Virtual_memory)的系统中，在尝试访问之前，实际上可能没有任何物理内存映射到逻辑地址。访问触发操作系统的特殊功能，该功能重新编程 MMU 以将地址映射到某个物理内存，可能将该内存的旧内容写入磁盘并从磁盘读回内存应在新逻辑地址处包含的内容。在这种情况下，逻辑地址可以称为[虚拟地址](https://en.wikipedia.org/wiki/Virtual_address)。

**物理地址：**出现在CPU外部地址总线上的寻址物理内存的地址信号，是地址变换的最终结果地址。CPU通过地址总线的寻址，找到真实的物理内存对应地址。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

在Intel平台下，逻辑地址（logical address）是selector:offset这种形式，selector 是CS寄存器的值，offset是EIP寄存器的值。如果用selector去GDT（全局描述符表）里拿到segment base address（段基址）然后加上offset（段内偏移），这就得到了linear address。我们把这个过程称作**段式内存管理**。

一个逻辑地址由两部分组成：段标识符和段内偏移量。段标识符是多位长的字段组成，称为段选择符，其中前面部分是一个索引号，后面部分可以从段描述符表中选择一个具体的段。程序过来一个逻辑地址，使用其段选择符的Index字段去索引段描述符表。将段描述符中的索引号对应的描述符字段和逻辑地址中的offset合并即得到了线性地址。

## 7.3 Hello的线性地址到物理地址的变换-页式管理

如果再把 linear address 切成四段，用前三段分别作为索引去PGD、PMD、Page Table里查表，最终就会得到一个页表项（Page Table Entry），那里面的值就是一页物理内存的起始地址，把它加上 linear address 切分之后第四段的内容（又叫页内偏移）就得到了最终的 physical address。我们把这个过程称作**页式内存管理**。

先将线性地址分为 VPN+VPO 的形式， 然后再将VPN拆分成TLBT+TLBI 索引然后去TLB缓存里找所对应的PPN（物理页号），如果发生缺页情况则直接查找对应的PPN，找到PPN之后，将其与VPO组合变为PPN+VPO就是生成的物理地址了。

## 7.4 TLB与四级页表支持下的VA到PA的变换

一级页表中的每个PTE负责映射虚拟地址空间中一个片（chunk），这里每一 片都是由连续的页面组成的。如果片i中的每个页面都未被分配，那么一级 PTEi 就为空。依次类推，对二级到三级，三级到四级页表基本上也是如此。四级页表中的每个PTE都负责映射一个虚拟内存页面。具体步骤如下所示：

开始时，MMU从虚拟地址中抽取出VPN，并且检查TLB，看它是否因为前面的某个内存引用缓存了PTE的一个副本。TLB从VPN中抽取出TLB索引和TLB 标记，查看组中是否有条目可以有效匹配，若有则命中，然后将缓存的PPN返回给MMU。如果TLB不命中，那么MMU就需要从主存中取出相应的PTE。现在，MMU有了形成物理地址所需要的所有东西，通过将来自PTE的PPN和来自虚拟地址的VPO连接起来，就形成了物理地址。

如果TLB不命中，那么MMU必须从页表中的PTE中取出PPN，如果得到的 PTE是无效的，那么就产生一个缺页，内核必须调入合适的页面，重新运行这条加载指令。还有其他情况就不一一赘述了。

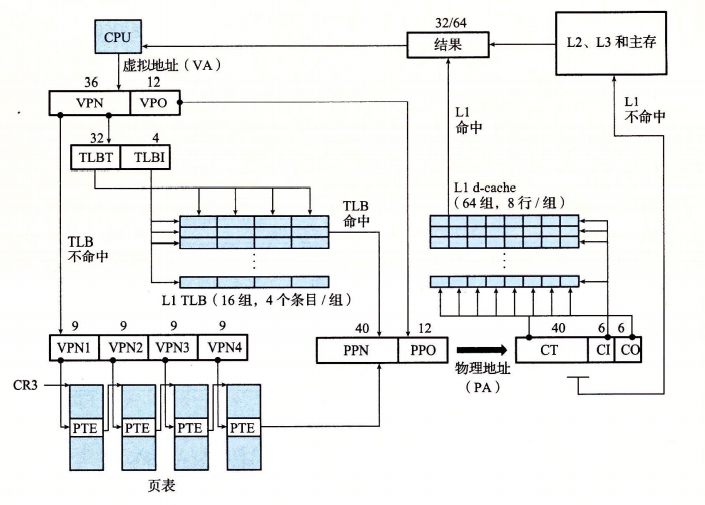


图29：地址翻译过程

## 7.5 三级Cache支持下的物理内存访问

MMU发送物理地址给缓存，缓存从物理地址中抽取出缓存偏移CO、缓存组索引CI以及缓存标记CT。若CI组中存在标记与CT相匹配，则表示缓存命中，读出在偏移量CO处的数据字节，并将它返回给MMU。随后 MMU 将它传递回 CPU。如果不命中就依次去第二三级高级缓存去取相关数据或代码所在的块。

## 7.6 hello进程fork时的内存映射

当fork函数被当前进程调用时，内核为新进程创建各种数据结构，并分配给它一个唯一的 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。它将两个进程中的每个页面都标记为只读，并将两个进程中的每个区域结构都标记为私有的写时复制。

当fork在新进程中返回时，新进程现在的虚拟内存刚好和调用fork时存在的虚拟内存相同。当这两个进程中的任一个后来进行写操作时，写时复制机制就会创建新页面，因此，也就为每个进程保持了私有地址空间的抽象概念。

## 7.7 hello进程execve时的内存映射

execve函数在当前进程中加载并运行包含在可执行目标文件filename中的程序，用filename程序有效地替代了当前程序。加载并运行filename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filename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filename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filename程序与共享对象（或目标）链接，比如标准 C库libc.so那么这些对象都是动态链接到这个程序的，然后再映射到用户虚拟地址空间中的共享区域内。
4. 设置程序计数器（PC)。execve做的最后一件事情就是设置当前进程上下文中的程序计数器，使之指向代码区域的人口点。

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

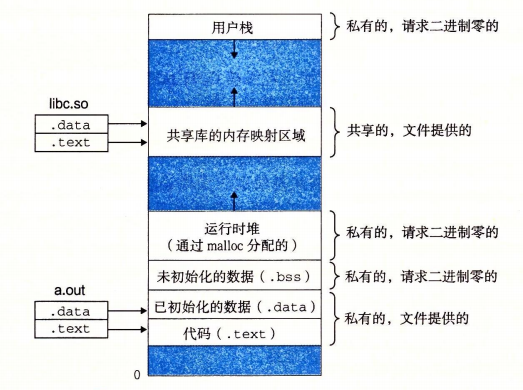


图30：内存映射结果

## 7.8 缺页故障与缺页中断处理

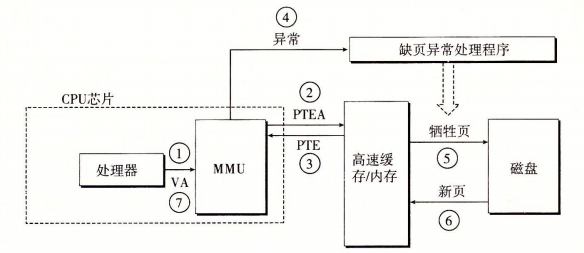


图31：缺页故障处理

处理缺页要求硬件和操作系统内核协作完成：

1. 处理器生成一个虚拟地址，并把它传送给MMU。
2. MMU生成PTE地址，并从高速缓存/主存请求得到它。
3. 高速缓存/主存向MMU返回PTE。
4. PTE中的有效位是零，所以MMU触发了一次异常，传递CPU中的控制到操作系统内核中的缺页异常处理程序。
5. 缺页处理程序确定出物理内存中的牺牲页，如果这个页面已经被修改了，则把它换出到磁盘。
6. 缺页处理程序页面调人新的页面，并更新内存中的PTE。
7. 缺页处理程序返回到原来的进程，再次执行导致缺页的指令。CPU将引起缺页的虚拟地址重新发送给MMU。因为虚拟页面现在缓存在物理内存中，所以就会命中。

## 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位供应用程序使用。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。

**隐式空闲链表：**

任何实际的分配器都需要一些数据结构，允许它来区别块边界，以及区别已分配块和空闲块。大多数分配器将这些信息嵌入块本身。

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。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位总是零。因此，我们只需要内存大小的高位，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。头部后面就是应用调用malloc时请求的有效载荷。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。

**显式空闲链表：**

把堆组织成一个双向空闲链表，在每个空闲块中，都包含一个pred（前驱）和succ（后继）指针。

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

**分离的空闲链表：**

分离存储，即维护多个空闲链表，其中每个链表中的块有大致相等的大小。一般的思路是将所有可能的块大小分成一些等价类，分配器维护着一个空闲链表数组，每个大小类一个空闲链表，按照大小的升序排列。当分配器需要一个大小为n的块时，它就搜索相应的空闲链表。如果不能找到合适的块与之匹配，它就搜索下一个链表，以此类推。两种基本的分离存储方法：简单分离存储和分离适配。

## 7.10本章小结

本章是关于储存管理的一章内容。主要从以下几个方面介绍储存形式以及读写过程等：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、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。

**（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：文件**

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。

**设备管理：unix io接口**

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, 这使得所有的输人和输出都能以一种统 一且一致的方式来执行：打开文件；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、标准输出和标准错误；改变当前的文件位置；读写文件；关闭文件。

## 8.2 简述Unix IO接口及其函数

**Unix IO接口：**

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。

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、标准输出和标准错误。头文件< unistd.h> 定义了常量 STDIN\_ FILENO、STDOUT\_FILENO 和 STDERR\_FILENO，它们可用来代替显式的描述符值。

改变当前的文件位置。对于每个打开的文件，内核保持着一个文件位置k，初始为0。这个文件位置是从文件开头起始的字节偏移量。应用程序能够通过执行 seek操作，显式地设置文件的当前位置为是。

读写文件。一个读操作就是从文件复制n>0个字节到内存，从当前文件位置是开始，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称为 end-of-file(EOF)的条件，应用程序能检测到这个条件。在文件结尾处并没有明确的 “EOF符号”。

关闭文件。当应用完成了对文件的访问之后，它就通知内核关闭这个文件。作为响应，内核释放文件打开时创建的数据结构，并将这个描述符恢复到可用的描述符池中。无论一个进程因为何种原因终止时，内核都会关闭所有打开的文件并释放它们的内存资源。

**函数：**

1. 打开和关闭文件

进程是通过调用open函数来打开一个已存在的文件或者创建一个新文件，进程通过调用close函数关闭一个打开的文件。

1. 读和写文件

应用程序是通过分别调用read和write函数来执行输入和输出的。read函数从描述符为fd的当前文件位置复制最多n个字节到内存位置 buf。返回值-1表示一个错误，而返回值0表示EOF。否则，返回值表示的是实际传送的字节数量。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。

1. 用RIO包健壮地读写

通过调用rio\_readn和rio\_writen函数，应用程序可以在内存和文件之间直接传送数据。

1. 读取文件元数据

应用程序能够通过调用stat和fstat函数，检索到关于文件的信息（有时也称为文件的元数据（metadata））。

1. 读取目录内容

应用程序可以用readdir系列函数来读取目录的内容。函数opendir以路径名为参数，返回指向目录流（directory stream）的指针。流是对条目有序列表的抽象，在这里是指目录项的列表。函数closedir关闭流并释放其所有的资源。

## 8.3 printf的实现分析

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

首先，printf函数的函数体如下表示：

int printf(const char \*fmt, ...) {

    int i;

    char buf[256];

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

    i = vsprintf(buf, fmt, arg);

    write(buf, i);

    return i;

}

从vsprintf生成显示信息，到write系统函数，到陷阱-系统调用 int 0x80或syscall.字符显示驱动子程序：从ASCII到字模库到显示vram（存储每一个点的RGB颜色信息）。显示芯片按照刷新频率逐行读取vram，并通过信号线向液晶显示器传输每一个点（RGB分量）。

该函数接受一个格式字符串是fmt，之后是一个变参列表。后面每一个参数都对应这格式字符串中的一个格式符。调用了两个函数：vsprintf和write。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。所以printf的实现就是：用参数匹配格式字符串，然后用vsprintf将结果字符串整理，最后用write函数输出。

## 8.4 getchar的实现分析

getchar()是最简单的一次读一个字符的函数，每次调用时从文本流中读入下一个字符，并将其作为结果值返回，返回值是int型。在没有输入或者输入字符有错的时候，getchar()函数将返回一个特殊值EOF。

异步异常-键盘中断的处理：键盘中断处理子程序。接受按键扫描码转成ascii码，保存到系统的键盘缓冲区。getchar等调用read系统函数，通过系统调用读取按键ascii码，直到接受到回车键才返回。

## 8.5本章小结

本章介绍了Linux的IO设备管理方法，UNIX的IO接口及其函数，还介绍了printf和getchar两个函数的实现方法。

Linux提供了少量的基于Unix I/O模型的系统级函数，它们允许应用程序打开、关闭、读和写文件，提取文件的元数据，以及执行I/O重定向。

**（第8章1分）**

# 结论

Hello的一生独白：

1. 预处理：将hello.c变为文本文档hello.i文件
2. 编译：将hello.i编译成为汇编文件hello.s
3. 汇编：将hello.s会变成为可重定位目标文件 hello.o
4. 链接：将hello.o与可重定位目标文件和动态链接库链接成为可执行目标程序 hello
5. 创建子进程：shell进程调用fork为其创建子进程
6. 运行程序：shell调用execve函数，加载运行当前进程的上下文中，execve调用启动加载器，映射虚拟内存，然后进入main函数。
7. 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。

对计算机系统的设计与实现的深切感悟，你的创新理念，如新的设计与实现方法。

用一种不同的方式向学生介绍计算机。排除了诸如硬件加法器和总线设计这样的主题。虽然谈及了机器语言，但是重点并不在于如何手工编写汇编语言，而是关注C语言编译器是如何将C语言的结构翻译成机器代码的，包括编译器是如何翻译指针、循环、过程调用以及开关（switch) 语句的。更进一步地，我们将更广泛和全盘地看待系统，包括硬件和系统软件，涵盖了包括链接、加载、进程、信号、性能优化、虚拟内存、I/O 以及网络与并发编程等在内的主题。

**（结论0分，缺失 -1分，根据内容酌情加分）**

# 附件

所有的中间产物的文件名，及其作用。

|  |  |
| --- | --- |
| **中间结果文件的名字** | **文件的作用** |
| hello.i | 修改了的源程序（文本） |
| hello.s | 汇编程序（文本） |
| hello.o | 可重定位目标程序（二进制） |
| hello | 可执行目标程序（二进制） |
| elf.txt | 可重定位目标ELF格式 |
| linked\_elf.txt | 可执行目标ELF格式 |
| objdump.txt | hello.o的反汇编代码 |
| objdump2.txt | hello的反汇编代码 |

**（附件0分，缺失 -1分）**

# 参考文献

[1] https://bbs.pediy.com/thread-249833.htm

[2] https://zh.wikipedia.org/wiki/%E9%A2%84%E5%A4%84%E7%90%86%E5%99%A8

[3] Randal E. Bryant, David R. O’Hallaron. 深入理解计算机系统[M]. 北京：机械工业出版社，2016.

[4] https://www.cnblogs.com/losing-1216/p/4884483.html

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

[6] <https://zh.wikipedia.org/wiki/%E8%99%9A%E6%8B%9F%E5%9C%B0%E5%9D%80>

[7] <https://www.jianshu.com/p/8b37d10bc504>

[8] https://www.zhihu.com/question/29918252/answer/163114415

**（参考文献0分，缺失 -1分）**