

**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 1190202405

班　　 级 1903005

学 生 林逸灏

指 导 教 师 史先俊

**计算机科学与技术学院**

**2021年5月**

**摘 要**

程序的开发和执行涉及计算机系统的各个层面，因而计算机系统层次结构的思想体现在程序开发和执行过程的各个环节中。本文通过分析简单的hello程序，介绍程序开发与执行的过程，从而加深对计算机系统层次结构的理解。

**关键词：计算机系统；可执行文件；虚拟内存；异常控制流；I/O操作**

**（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 - 5 -](#_Toc75202792)

[1.1 Hello简介 - 5 -](#_Toc75202793)

[1.2 环境与工具 - 5 -](#_Toc75202794)

[1.3 中间结果 - 5 -](#_Toc75202795)

[1.4 本章小结 - 6 -](#_Toc75202796)

[第2章 预处理 - 7 -](#_Toc75202797)

[2.1 预处理的概念与作用 - 7 -](#_Toc75202798)

[2.2在Ubuntu下预处理的命令 - 7 -](#_Toc75202799)

[2.3 Hello的预处理结果解析 - 8 -](#_Toc75202800)

[2.4 本章小结 - 9 -](#_Toc75202801)

[第3章 编译 - 10 -](#_Toc75202802)

[3.1 编译的概念与作用 - 10 -](#_Toc75202803)

[3.2 在Ubuntu下编译的命令 - 10 -](#_Toc75202804)

[3.3 Hello的编译结果解析 - 10 -](#_Toc75202805)

[3.3.1 数据 - 10 -](#_Toc75202806)

[3.3.2算术操作 - 11 -](#_Toc75202807)

[3.3.3 关系操作 - 11 -](#_Toc75202808)

[3.3.4数组操作 - 12 -](#_Toc75202809)

[3.3.5函数操作 - 12 -](#_Toc75202810)

[3.4 本章小结 - 13 -](#_Toc75202811)

[第4章 汇编 - 14 -](#_Toc75202812)

[4.1 汇编的概念与作用 - 14 -](#_Toc75202813)

[4.2 在Ubuntu下汇编的命令 - 14 -](#_Toc75202814)

[4.3 可重定位目标elf格式 - 14 -](#_Toc75202815)

[4.4 Hello.o的结果解析 - 17 -](#_Toc75202816)

[4.5 本章小结 - 18 -](#_Toc75202817)

[第5章 链接 - 19 -](#_Toc75202818)

[5.1 链接的概念与作用 - 19 -](#_Toc75202819)

[5.2 在Ubuntu下链接的命令 - 19 -](#_Toc75202820)

[5.3 可执行目标文件hello的格式 - 19 -](#_Toc75202821)

[5.4 hello的虚拟地址空间 - 21 -](#_Toc75202822)

[5.5 链接的重定位过程分析 - 23 -](#_Toc75202823)

[5.6 hello的执行流程 - 24 -](#_Toc75202824)

[5.7 Hello的动态链接分析 - 25 -](#_Toc75202825)

[5.8 本章小结 - 26 -](#_Toc75202826)

[第6章 hello进程管理 - 27 -](#_Toc75202827)

[6.1 进程的概念与作用 - 27 -](#_Toc75202828)

[6.2 简述壳Shell-bash的作用与处理流程 - 27 -](#_Toc75202829)

[6.3 Hello的fork进程创建过程 - 27 -](#_Toc75202830)

[6.4 Hello的execve过程 - 28 -](#_Toc75202831)

[6.5 Hello的进程执行 - 28 -](#_Toc75202832)

[6.6 hello的异常与信号处理 - 29 -](#_Toc75202833)

[6.7本章小结 - 32 -](#_Toc75202834)

[第7章 hello的存储管理 - 33 -](#_Toc75202835)

[7.1 hello的存储器地址空间 - 33 -](#_Toc75202836)

[7.2 Intel逻辑地址到线性地址的变换-段式管理 - 33 -](#_Toc75202837)

[7.3 Hello的线性地址到物理地址的变换-页式管理 - 34 -](#_Toc75202838)

[7.4 TLB与四级页表支持下的VA到PA的变换 - 35 -](#_Toc75202839)

[7.5 三级Cache支持下的物理内存访问 - 35 -](#_Toc75202840)

[7.6 hello进程fork时的内存映射 - 36 -](#_Toc75202841)

[7.7 hello进程execve时的内存映射 - 36 -](#_Toc75202842)

[7.8 缺页故障与缺页中断处理 - 37 -](#_Toc75202843)

[7.9动态存储分配管理 - 38 -](#_Toc75202844)

[7.10本章小结 - 38 -](#_Toc75202845)

[第8章 hello的IO管理 - 39 -](#_Toc75202846)

[8.1 Linux的IO设备管理方法 - 39 -](#_Toc75202847)

[8.2 简述Unix IO接口及其函数 - 39 -](#_Toc75202848)

[8.3 printf的实现分析 - 40 -](#_Toc75202849)

[8.4 getchar的实现分析 - 41 -](#_Toc75202850)

[8.5本章小结 - 42 -](#_Toc75202851)

[结论 - 42 -](#_Toc75202852)

[附件 - 43 -](#_Toc75202853)

[参考文献 - 44 -](#_Toc75202854)

# 第1章 概述

## 1.1 Hello简介

P2P：

在linux系统中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文件，我们可以在shell中运行这个可执行文件，shell为其fork一个子进程，hello从一个程序(program)变成了进程(process)。

020：

shell为hello进程execve，映射虚拟内存，进入程序入口后程序开始载入物理内存，然后然后进入main函数执行目标代码，CPU 为运行的 hello 分配时间片执行逻辑控制流。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。

## 1.2 环境与工具

硬件环境：处理器：Intel® Core™ i7-9750U CPU @ 1.80GHz 1.99GHz

RAM：16.00GB 系统类型：64位操作系统，基于x64的处理器

软件环境：Windows10 64位；Ubuntu 20.04

开发与调试工具：gcc，as，ld，vim，edb，readelf，VScode，Hexedit

## 1.3 中间结果

|  |  |
| --- | --- |
| **文件名** | **文件的作用** |
| **hello.i** | 预处理后的文件 |
| **hello.s** | 编译之后的汇编文件 |
| **hello.o** | 汇编之后的可重定位目标文件 |
| **Hello** | 链接之后的可执行目标文件 |
| **Elf.txt** | Hello.o 的 ELF 格式 |
| **Disas\_hello.s** | Hello.o 的反汇编代码 |
| **hello1.elf** | hello的ELF 格式 |
| **hello1\_objdump.s** | hello 的反汇编代码 |

## 1.4 本章小结

本章简要介绍了hello的P2P与020过程，介绍了本次大作业的实验环境与产生的中间文件。

**（第1章0.5分）**

# 第2章 预处理

## 2.1 预处理的概念与作用

预处理的概念：

预处理中会展开以#起始的行，试图解释为预处理指令(preprocessing directive) ，其中 ISO C/C++要求支持的包括#if、 #ifdef、 #ifndef、 #else、 #elif、 #endif（条件编译）、 #define（宏定义）、 #include（源文件包含）、 #line（行控制）、 #error（错误指令）、 #pragma（和实现相关的杂注）以及单独的#（空指令）。预处理指令一般被用来使源代码在不同的执行环境中被方便的修改或者编译。

预处理的作用：

1. 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。
2. 用实际值替换用#define 定义的字符串
3. 根据#if 后面的条件决定需要编译的代码
4. 特殊符号，预编译程序可以识别一些特殊的符号， 预编译程序对于在源程序中出现的这些串将用合适的值进行替换。

## 2.2在Ubuntu下预处理的命令

命令：cpp hello.c > hello.i



图 1 预处理

## 2.3 Hello的预处理结果解析



图 2 预处理文件中的main函数

运行了cpp指令后，产生了具有3000多行的hello.i文件，我们可以看到，main函数出现在该文件的末尾，之前的内容大多为头文件stdio.h、unistd.h、stdlib.h的依次展开。以 stdio.h 的展开为例： stdio.h 是标准库文件， cpp 到 Ubuntu 中默认的环境变量下寻找 stdio.h，打开文件/usr/include/stdio.h ，发现其中依然使用了#define 语句，cpp 对 stdio 中的define 宏定义递归展开。 所以最终的.i 文件中是没有#define 的； 发现其中使用了大量的#ifdef #ifndef 条件编译的语句， cpp 会对条件值进行判断来决定是否执行包含其中的逻辑。预编译程序可识别一些特殊的符号，预编译程序对在源程序中出现的这些串将用合适的值进行替换。对于其他的头文件，处理形式也类似。

插入库文件位置：

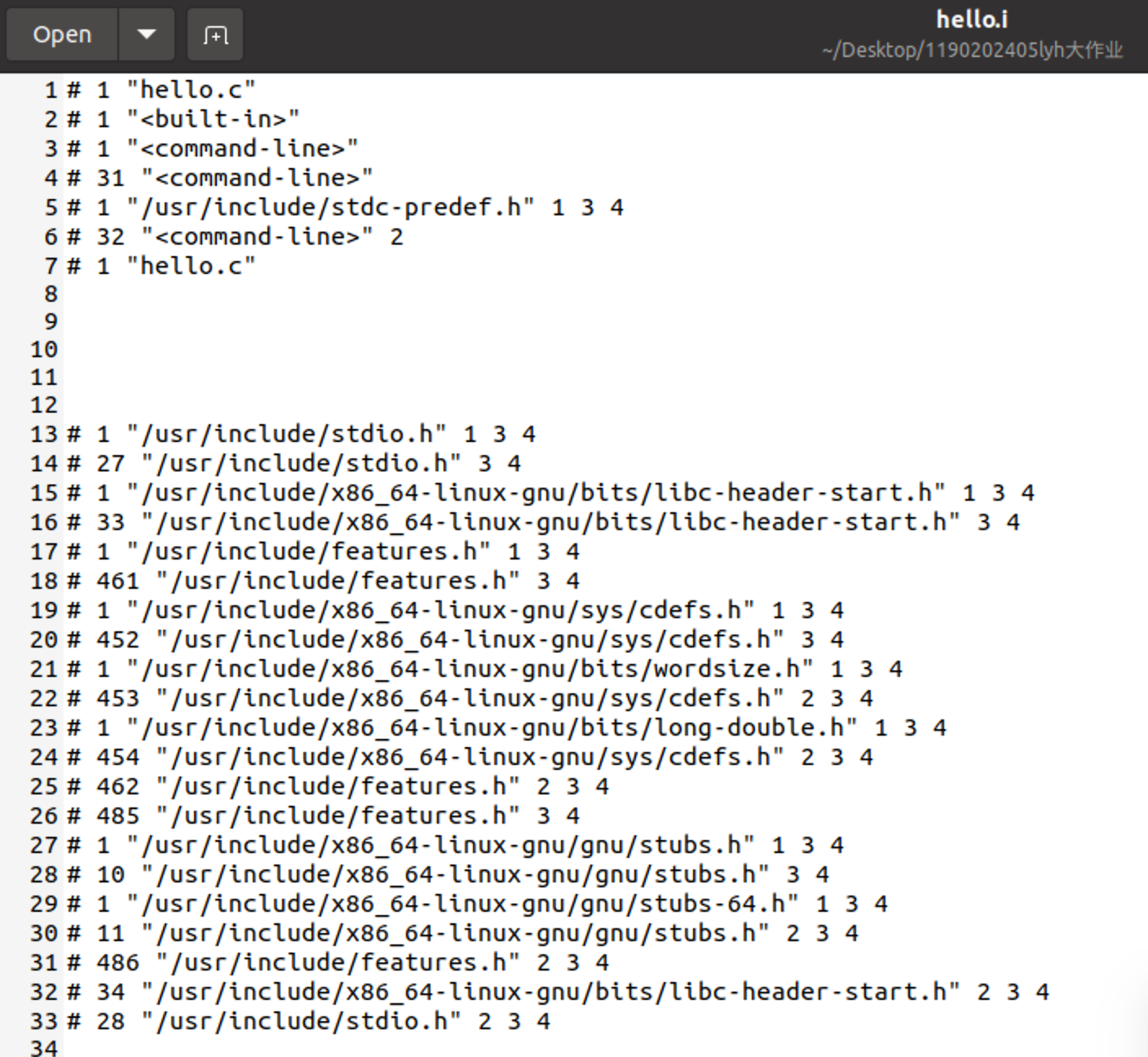


图 3 头文件中的库的地址

## 2.4 本章小结

本章主要介绍了预处理的概念与作用，并使用cpp命令生成了hello.i文件，对该文件进行了解析，对#include插入头文件的作用过程进行了说明。

**（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：

编译即将某种高级程序语言写的程序翻译成汇编语言，通过语法分析，将高级程序语言文本程序翻译成汇编语言文本程序，将hello.i预处理文件翻译成hello.s。

编译的作用：经过词法分析；语法分析；语义检查和中间代码生成；代码优化；目标代码生成等过程，将高级语言翻译成更接近机器语言的汇编语言，便于生成机器能够运行的程序。

## 3.2 在Ubuntu下编译的命令

命令：gcc -S hello.i -o hello.s

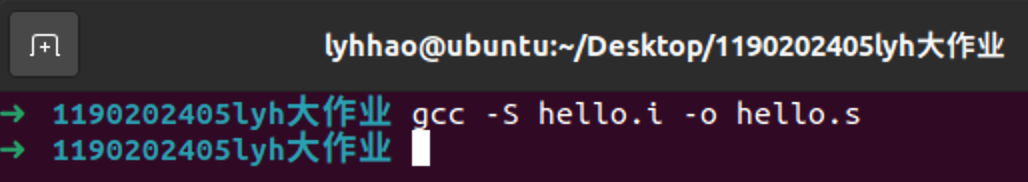


图 4 编译

## 3.3 Hello的编译结果解析

### 3.3.1 数据

1. 常量

if(argc!=4)

该表达式中的4为常量，该段表达式对应的汇编代码为：

|  |
| --- |
| movl %edi, -20(%rbp)  movq %rsi, -32(%rbp)  cmpl $4, -20(%rbp)  je .L2 |

该常量直接存储在.text中。

printf("用法: Hello 学号 姓名 秒数！\n");

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

这两句printf函数中的字符串存储在.rodata段中：

|  |
| --- |
| .LC0:  .string "\347\224\250\346\263\225: Hello \345\255\246\345\217\267 \345\247\223\345\220\215 \347\247\222\346\225\260\357\274\201" |
| .LC1:  .string "Hello %s %s\n" |

2.变量

全局变量：初始化的全局变量存储在.data段中，未初始化的全局变量存储在.bss段中。

局部变量：局部变量存储在寄存器或者栈中，如main函数中定义的int i; 在汇编代码中：

|  |
| --- |
| .L2:  movl $0, -4(%rbp)  jmp .L3 |

显示该局部变量存储在栈中。

### 3.3.2算术操作

使用mov指令能对寄存器、栈中的参数进行赋值

在for循环中，每次循环结束，都进行++，自增运算符。

addl $1, -4(%rbp)

### 3.3.3 关系操作

在if判断语句中，判断argc是否等于4

汇编代码为：

cmpl $4, -20(%rbp)

je .L2

je 判断 cmpl产生的条件码，若两个操作数的值不同，则跳转到.L2

在for循环中也有判断循环的终止条件for (i = 0; i < 8; i++)

汇编代码为：

.L3:

cmpl $7, -4(%rbp)

jle .L4

jle判断cmpl产生的条件码，若后一个操作数的值小于等于前一个，则跳转到.L4.

### 3.3.4数组操作

源代码中用到了char \*argv[] 这一数组。char \*类型占8个字节。在该数组中，argv[0]为输入程序名的字符串，argv[1]、argv[2]为两个命令行参数。

在汇编指令中的位置：

|  |
| --- |
| .L4:  movq -32(%rbp), %rax //argv 的位置  addq $16, %rax //argv[2]的位置  movq (%rax), %rdx  movq -32(%rbp), %rax  addq $8, %rax //argv[1]的位置  movq (%rax), %rax |

### 3.3.5函数操作

参数传递：64位，前6个参数在寄存器中，按顺序为%rdi,%rsi,%rdx,%rcx,%r8,%r9.

函数调用：使用call指令，例如：call puts 将调用指令的下一条指令的地址压入栈中。

函数返回：使用ret指令，把call压入栈的指令弹出并跳到这条指令的位置。

以main函数为例：

参数传递：传入参数argc和argv[]，分别用寄存器%rdi和%rsi存储。

函数调用：运行该程序时，由操作系统底层调用

函数返回：设置%eax为0并且返回，对应return 0 。

源代码：

int main(int argc, char \*argv[])

汇编代码：

main:

.LFB6:

.cfi\_startproc

pushq %rbp

.cfi\_def\_cfa\_offset 16

.cfi\_offset 6, -16

movq %rsp, %rbp

.cfi\_def\_cfa\_register 6

subq $32, %rsp

movl %edi, -20(%rbp)

movq %rsi, -32(%rbp)

可见argc存储在%edi中，argv存储在%rsi中；

## 3.4 本章小结

本章介绍了编译的指令与编译所需要的步骤，并将编译得到的汇编代码进行了详解，介绍了其中的参数、变量、控制转移、关系操作、函数操作等。

**（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

驱动程序运行汇编器as，将汇编语言（这里是hello.s）翻译成机器语言（hello.o）的过程称为汇编，同时这个机器语言文件也是可重定位目标文件。

汇编的作用

汇编就是将高级语言转化为机器可直接识别执行的代码文件的过程，汇编器将.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。

## 4.2 在Ubuntu下汇编的命令

命令：as hello.s -o hello.o

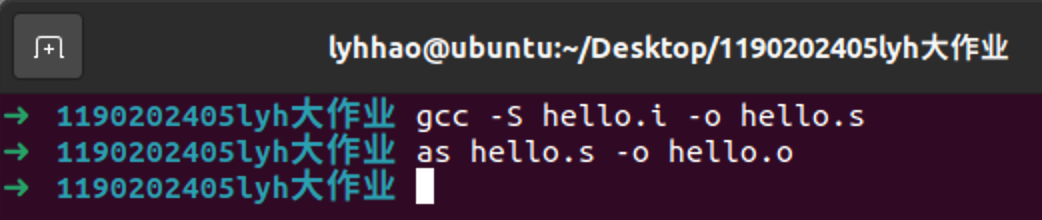


图 5 汇编

## 4.3 可重定位目标elf格式

命令： readelf -a hello.o > ./elf.txt

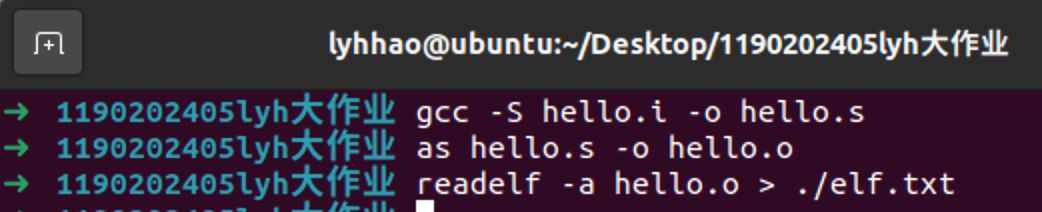


图 6 elf文件

ELF文件组成如下

1.ELF header：包含了ELF头大小、段大小、编码方式、系统信息等一系列信息。内容如下：

|  |
| --- |
| 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 |

2.节头表：包含了文件中出现的各个节的类型、位置、大小等信息

|  |
| --- |
| 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  0000000000000092 0000000000000000 AX 0 0 1  [ 2] .rela.text RELA 0000000000000000 00000388  00000000000000c0 0000000000000018 I 11 1 8  [ 3] .data PROGBITS 0000000000000000 000000d2  0000000000000000 0000000000000000 WA 0 0 1  [ 4] .bss NOBITS 0000000000000000 000000d2  0000000000000000 0000000000000000 WA 0 0 1  [ 5] .rodata PROGBITS 0000000000000000 000000d8  0000000000000033 0000000000000000 A 0 0 8  [ 6] .comment PROGBITS 0000000000000000 0000010b  0000000000000027 0000000000000001 MS 0 0 1  [ 7] .note.GNU-stack PROGBITS 0000000000000000 00000132  0000000000000000 0000000000000000 0 0 1  [ 8] .note.gnu.pr[...] NOTE 0000000000000000 00000138  0000000000000020 0000000000000000 A 0 0 8  [ 9] .eh\_frame PROGBITS 0000000000000000 00000158  0000000000000038 0000000000000000 A 0 0 8  [10] .rela.eh\_frame RELA 0000000000000000 00000448  0000000000000018 0000000000000018 I 11 9 8  [11] .symtab SYMTAB 0000000000000000 00000190  00000000000001b0 0000000000000018 12 10 8  [12] .strtab STRTAB 0000000000000000 00000340  0000000000000048 0000000000000000 0 0 1  [13] .shstrtab STRTAB 0000000000000000 00000460  0000000000000074 0000000000000000 0 0 1  Key to Flags:  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),  L (link order), O (extra OS processing required), G (group), T (TLS),  C (compressed), x (unknown), o (OS specific), E (exclude),  l (large), p (processor specific) |

3.重定位节：告诉链接器在将目标文件合并成可执行文件是如何对这些位置的地址进行修改。

|  |
| --- |
| Offset Info Type Sym. Value Sym. Name + Addend  00000000001c 000500000002 R\_X86\_64\_PC32 0000000000000000 .rodata - 4  000000000021 000c00000004 R\_X86\_64\_PLT32 0000000000000000 puts - 4  00000000002b 000d00000004 R\_X86\_64\_PLT32 0000000000000000 exit - 4  000000000054 000500000002 R\_X86\_64\_PC32 0000000000000000 .rodata + 22  00000000005e 000e00000004 R\_X86\_64\_PLT32 0000000000000000 printf - 4  000000000071 000f00000004 R\_X86\_64\_PLT32 0000000000000000 atoi - 4  000000000078 001000000004 R\_X86\_64\_PLT32 0000000000000000 sleep - 4  000000000087 001100000004 R\_X86\_64\_PLT32 0000000000000000 getchar - 4 |

其中，偏移量（offset）：需要被修改的引用的节偏移。

信息（info）：提供了符号表中的一个位置，同时还包括重定位类型的有个信息。这是通过将值划分为两部分来达到的。该值的低8位表示重定位入口的类型， 高24位表示重定位入口的符号在符号表重的下标。

类型（type）：告知连接器如何修改新的引用

符号名称（name）：重定位目标的名称

加数（addend）：一个有符号常数，一些类型的重定位要使用它对被修改引用的值做便宜调整

4.符号表：.symtab，存放在程序中定义和引用的函数与全局变量的信息

|  |
| --- |
| 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 146 FUNC GLOBAL DEFAULT 1 main  11: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND \_GLOBAL\_OFFSET\_TABLE\_  12: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND puts  13: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND exit  14: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND printf  15: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND atoi  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：objdump -d -r hello.o > Disas\_hello.s

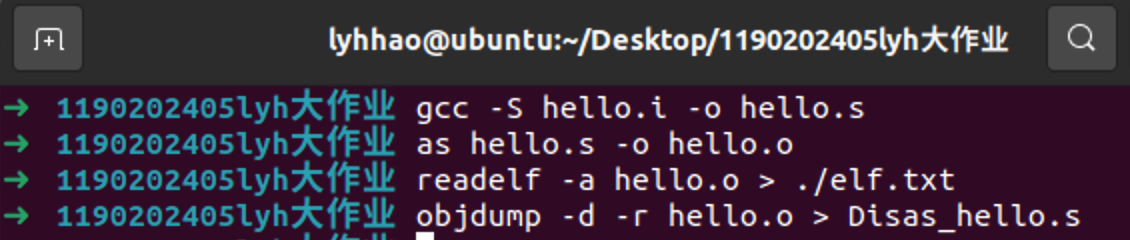


图 7 objdump导出反汇编文件

分析hello.o的反汇编，并请与第3章的 hello.s进行对照分析。

反汇编文件与hello.s文件在整体内容上差别不大，但有一点细节上的差别

1.分支转移：反汇编直接使用相对偏移的地址进行跳转，hello.s文件中使用.L2等段名称进行跳转。

|  |  |
| --- | --- |
| 反汇编 | hello.s |
| je 2f <main+0x2f> | je .L2 |

2.函数调用：在反汇编程中，call 的目标地址是当前下一条指令,在hello.s 文件中，函数调用之后直接跟着函数名称。

|  |  |
| --- | --- |
| 反汇编 | hello.s |
| callq 62 <main+0x62> | call printf@PLT |

3.数制表示：反汇编中为十六进制数，hello.s中为十进制数

。

## 4.5 本章小结

本章介绍了汇编器的工作原理，将hello.s文件翻译成hello.o文件，再通过反汇编工具，将hello.o文件反汇编，比较反汇编代码与hello.s中的差别，经过这一过程，我们能更加深刻地理解汇编语言到机器语言的转换，以及可重定位目标文件的结构。

**（第4章1分）**

# 第5章 链接

## 5.1 链接的概念与作用

概念：链接是将各种代码和数据片段收集并组合成一个单一文件的过程。

作用：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：ld -o hello -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib/x86\_64-linux-gnu/crt1.o /usr/lib/x86\_64-linux-gnu/crti.o hello.o /usr/lib/x86\_64-linux-gnu/libc.so /usr/lib/x86\_64-linux-gnu/crtn.o

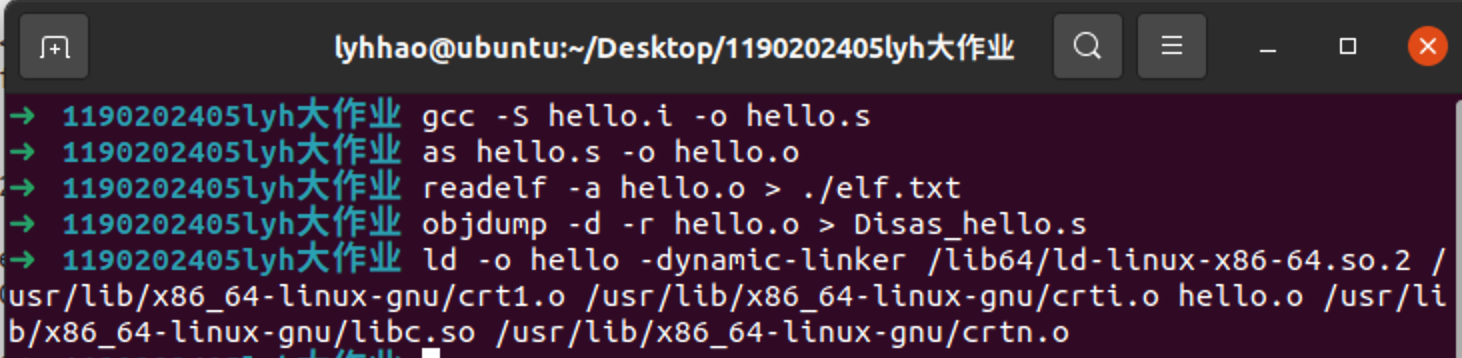


图 8 链接

## 5.3 可执行目标文件hello的格式

命令： readelf -a hello > hello.elf

ELF Header:

|  |
| --- |
| 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: 0x4010f0  Start of program headers: 64 (bytes into file)  Start of section headers: 14296 (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 |

节头表：描述各个节的大小、偏移量、类型等信息，同第四章。

|  |
| --- |
| Section Headers:  [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.pr[...] 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  0000000000000038 0000000000000004 A 6 0 8  [ 5] .gnu.hash GNU\_HASH 0000000000400378 00000378  000000000000001c 0000000000000000 A 6 0 8  [ 6] .dynsym DYNSYM 0000000000400398 00000398  00000000000000d8 0000000000000018 A 7 1 8  [ 7] .dynstr STRTAB 0000000000400470 00000470  000000000000005c 0000000000000000 A 0 0 1  [ 8] .gnu.version VERSYM 00000000004004cc 000004cc  0000000000000012 0000000000000002 A 6 0 2  [ 9] .gnu.version\_r VERNEED 00000000004004e0 000004e0  0000000000000020 0000000000000000 A 7 1 8  [10] .rela.dyn RELA 0000000000400500 00000500  0000000000000030 0000000000000018 A 6 0 8  [11] .rela.plt RELA 0000000000400530 00000530  0000000000000090 0000000000000018 AI 6 21 8  [12] .init PROGBITS 0000000000401000 00001000  000000000000001b 0000000000000000 AX 0 0 4  [13] .plt PROGBITS 0000000000401020 00001020  0000000000000070 0000000000000010 AX 0 0 16  [14] .plt.sec PROGBITS 0000000000401090 00001090  0000000000000060 0000000000000010 AX 0 0 16  [15] .text PROGBITS 00000000004010f0 000010f0  0000000000000145 0000000000000000 AX 0 0 16  [16] .fini PROGBITS 0000000000401238 00001238  000000000000000d 0000000000000000 AX 0 0 4  [17] .rodata PROGBITS 0000000000402000 00002000  000000000000003b 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  0000000000000048 0000000000000008 WA 0 0 8  [22] .data PROGBITS 0000000000404048 00003048  0000000000000004 0000000000000000 WA 0 0 1  [23] .comment PROGBITS 0000000000000000 0000304c  0000000000000026 0000000000000001 MS 0 0 1  [24] .symtab SYMTAB 0000000000000000 00003078  00000000000004f8 0000000000000018 25 32 8  [25] .strtab STRTAB 0000000000000000 00003570  0000000000000183 0000000000000000 0 0 1  [26] .shstrtab STRTAB 0000000000000000 000036f3  00000000000000e1 0000000000000000 0 0 1  Key to Flags:  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),  L (link order), O (extra OS processing required), G (group), T (TLS),  C (compressed), x (unknown), o (OS specific), E (exclude),  l (large), p (processor specific) |

## 5.4 hello的虚拟地址空间

使用edb加载hello，查看本进程的虚拟地址空间各段信息，并与5.3对照分析说明。

Data Dump中可以看到加载到细腻地址空间的hello程序。ELF文件中的Header告诉链接器运行时加载的内容，并提供动态链接的信息。提供了被载入后的虚拟地址空间与物理地址空间。

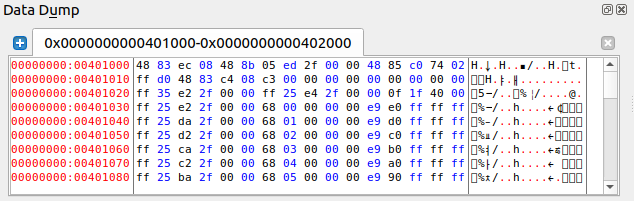


图 9 edb Data Dump

|  |
| --- |
| Program Headers:  Type Offset VirtAddr PhysAddr  FileSiz MemSiz Flags Align  PHDR 0x0000000000000040 0x0000000000400040 0x0000000000400040  0x00000000000002a0 0x00000000000002a0 R 0x8  INTERP 0x00000000000002e0 0x00000000004002e0 0x00000000004002e0  0x000000000000001c 0x000000000000001c R 0x1  [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]  LOAD 0x0000000000000000 0x0000000000400000 0x0000000000400000  0x00000000000005c0 0x00000000000005c0 R 0x1000  LOAD 0x0000000000001000 0x0000000000401000 0x0000000000401000  0x0000000000000245 0x0000000000000245 R E 0x1000  LOAD 0x0000000000002000 0x0000000000402000 0x0000000000402000  0x000000000000013c 0x000000000000013c R 0x1000  LOAD 0x0000000000002e50 0x0000000000403e50 0x0000000000403e50  0x00000000000001fc 0x00000000000001fc RW 0x1000  DYNAMIC 0x0000000000002e50 0x0000000000403e50 0x0000000000403e50  0x00000000000001a0 0x00000000000001a0 RW 0x8  NOTE 0x0000000000000300 0x0000000000400300 0x0000000000400300  0x0000000000000020 0x0000000000000020 R 0x8  NOTE 0x0000000000000320 0x0000000000400320 0x0000000000400320  0x0000000000000020 0x0000000000000020 R 0x4  GNU\_PROPERTY 0x0000000000000300 0x0000000000400300 0x0000000000400300  0x0000000000000020 0x0000000000000020 R 0x8  GNU\_STACK 0x0000000000000000 0x0000000000000000 0x0000000000000000  0x0000000000000000 0x0000000000000000 RW 0x10  GNU\_RELRO 0x0000000000002e50 0x0000000000403e50 0x0000000000403e50  0x00000000000001b0 0x00000000000001b0 R 0x1 |

## 5.5 链接的重定位过程分析

命令：objdump -d -r hello > hello\_objdump.s

|  |
| --- |
| hello: file format elf64-x86-64  Disassembly of section .init:  0000000000401000 <\_init>:  401000: f3 0f 1e fa endbr64  401004: 48 83 ec 08 sub $0x8,%rsp  401008: 48 8b 05 e9 2f 00 00 mov 0x2fe9(%rip),%rax # 403ff8 <\_\_gmon\_start\_\_>  40100f: 48 85 c0 test %rax,%rax  401012: 74 02 je 401016 <\_init+0x16>  401014: ff d0 callq \*%rax  401016: 48 83 c4 08 add $0x8,%rsp  40101a: c3 retq |

分析hello与hello.o的不同：

1.hello中多出了很多新的函数，这些函数是链接器从共享库中提取出来的，如exit、printf、sleep、getchar等函数，将其加入可执行目标文件中使其更加完整。

2.在函数调用上，hello中的call后面跟着的是所调用函数的虚拟内存地址，不同于hello.o中的相对偏移地址，链接器为我们算好了跳转的位置。

3.在hello增加了.init节

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。

## 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。

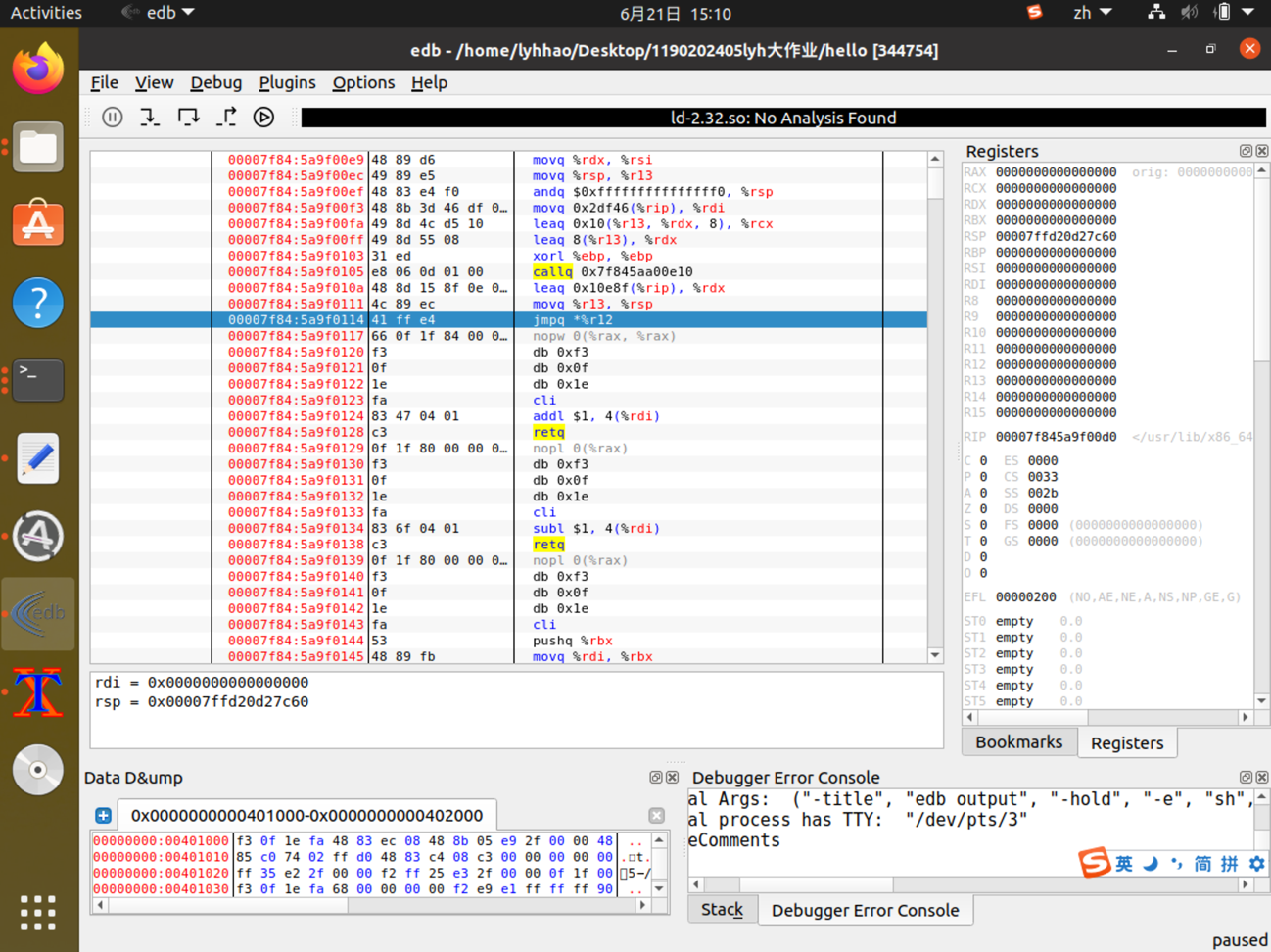


图 10 edb执行界面

子函数名与地址：

00401000 <\_init>

00401020 <.plt>

00401030 <puts@plt>

00401040 <printf@plt>

00401050 <getchar@plt>

00401060 <atoi@plt>

00401070 <exit@plt>

00401080 <sleep@plt>

00401090 <\_start>

004010c0 <\_dl\_relocate\_static\_pie>

004010c1 <main>

00401150 <\_\_libc\_csu\_init>

004011b0 <\_\_libc\_csu\_fini>

004011b4 <\_fini>

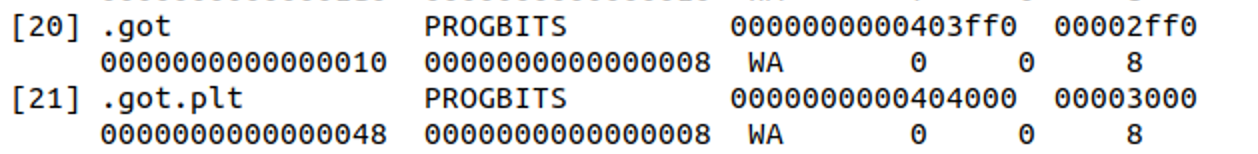
## 5.7 Hello的动态链接分析

分析hello程序的动态链接项目，通过edb调试，分析在dl\_init前后，这些项目的内容变化。要截图标识说明。

动态链接的基本思想是：程序按照模块拆分成各个相对独立部分，在程序运行时将它们链接在一起形成一个完整的程序，不是像静态链接一样把所有程序模块都链接成一个单独的可执行文件。虽然动态链接把链接过程推迟到了程序运行时，但是在形成可执行文件时，还是需要用到动态链接库。比如我们在形成可执行程序时，发现引用了一个外部的函数，此时会检查动态链接库，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。

在调用共享库函数时，编译器没有办法预测这个函数的运行时地址，因为定义它的共享模块在运行时可以加载到任意位置。正常的方法是为该引用生成一条重定位记录，然后动态链接器在程序加载的时候再解析它。GNU编译系统使用延迟绑定,将过程地址的绑定推迟到第一次调用该过程时。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elf文件中，我们能够找到.got与.got.plt的地址与偏移量



在edb中查看：

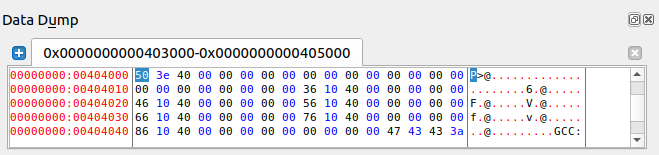


图 11 edb执行init之前的地址

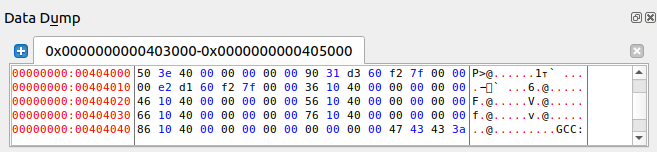


图 12 edb执行init之后的地址

## 5.8 本章小结

本章主要介绍了链接的概念与作用，通过对比hello与hello.o ,能更好地理解链接与重定位的相关过程。本章也解析了ELF文件各部分的含义、hello的执行过程、动态链接过程等。

**（第5章1分）**

# 第6章 hello进程管理

## 6.1 进程的概念与作用

概念：执行中的程序的抽象

作用：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。进程的经典定义就是一个执行中程序的实例。系统中的每个程序都运行在某个进程的上下文中。上下文是由程序正确运行所需要的状态组成的。这个状态包括存放在内存中的程序的代码和数据，他的栈，通用目的寄存器的内容、程序计数器、环境变量以及打开文件描述符的集合。

## 6.2 简述壳Shell-bash的作用与处理流程

作用：shell是一个解释命令的应用程序，为用户提供了一个界面，用于连接用户和操作系统内核。

处理流程：

1. 从终端读取输入的指令
2. 将读取的指令字符串进行切分，获得需要执行的程序名称以及运行该程序的各项参数。
3. 若为内置命令，shell程序立即调用操作系统内核执行该命令
4. 否则调用相应的程序，为其分配一个子进程并运行该程序
5. 在执行前的最后一步，需要初始化所有的输入输出重定向

## 6.3 Hello的fork进程创建过程

输入命令执行hello程序时，父进程若判断不是内置指令，则会通过调用fork函数来创建子进程。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空间相同且独立的副本，其中包含数据段、代码、堆和用户栈。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。fork函数只会被调用一次，但会返回两次，在父进程中，fork返回子进程的PID，在子进程中，fork返回0。

## 6.4 Hello的execve过程

调用fork函数后，在子进程中加载execve函数，载入并运行hello函数，execve带列表argv以及环境变量列表envp。只有当出现错误时，例如找不到filename，execve才会返回到调用程序。所以与fork调用一次返回两次不一样，execve调用一次并且从不返回。在当前进程的上下文中加载并运行一个新的程序。加载了hello之后，调用启动代码，覆盖当前进程的代码，数据，栈，并将控制转移至新程序的主函数。

## 6.5 Hello的进程执行

操作系统保持跟踪进程运行所需的所有状态信息。这种状态，也称上下文，包括许多信息，比如PC和寄存器文件的当前值，以及主存的内容。在任何一个时刻，单处理器系统都只能执行一个进程的代码。当操作系统决定要把控制权从当前进程转移到某个新进程时，就会执行上下文切换，即保存当前进程的上下文、恢复新进程的上下文，然后将控制权传递到新进程。新进程就会从它上次停止的地方开始。

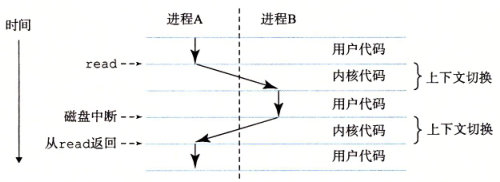


图 13 进程的上下文切换

hello程序运行场景中有两个并发的进程：shell进程和hello进程。最开始，只有shell进程在运行，即等待命令行上的输入。当运行hello程序时，shell通过调用一个专门的函数，即系统调用，来执行用户的请求，系统调用会将控制权传递给操作系统。操作系统保存shell的上下文，创建一个新的hello进程及其上下文，然后控制权传给新的hello进程。hello进程终止后，操作系统恢复shell进程的上下文，并将控制权传回给它，shell进程会继续等待下一行命令的输入。

从一个进程到另一个进程的转换是由操作系统内核管理的。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。内核不是一个独立的进程，相反，它是系统管理的全部进程所用代码和数据结构的集合。

实现进程这个抽象的概念需要低级硬件和操作系统软件之间的紧密合作。

逻辑控制流：即为一系列程序计数器PC的值序列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：进程执行它的控制流的部分时间段。

用户模式和内核模式：用户模式即为未设置模式位，不允许执行特权指令直接引用地址空间中内核区内的代码和数据；内核模式为设置模式位时，该进程执行所有命令访问所有数据。

上下文信息：上下文就是内核重启被抢占的进程所需要的状态，它由通用寄存器等各种内核数据构成。分别三个步骤：

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。

## 6.6 hello的异常与信号处理

hello执行过程中会出现哪几类异常，会产生哪些信号，又怎么处理的。

程序运行过程中可以按键盘，如不停乱按，包括回车，Ctrl-Z，Ctrl-C等，Ctrl-z后可以运行ps jobs pstree fg kill 等命令，请分别给出各命令及运行结截屏，说明异常与信号的处理。

正常运行时：

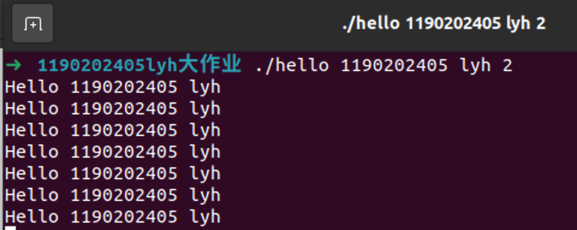


图 14 hello正常运行

异常类型：

|  |  |  |  |
| --- | --- | --- | --- |
| ***类别*** | **原因** | **异步/同步** | **返回行为** |
| *中断* | 来自I/O设备的信号 | 异步 | 总是返回到下一条指令 |
| *陷阱* | 有意的异常 | 同步 | 总是返回到下一条指令 |
| *故障* | 潜在可恢复的错误 | 同步 | 可能返回到当前指令 |
| *终止* | 不可恢复的错误 | 同步 | 不会返回 |

处理方式：

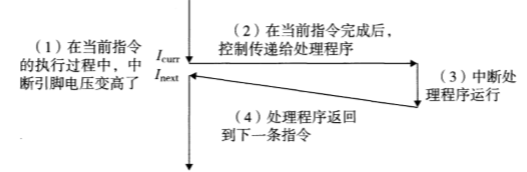


图 15 中断处理方式

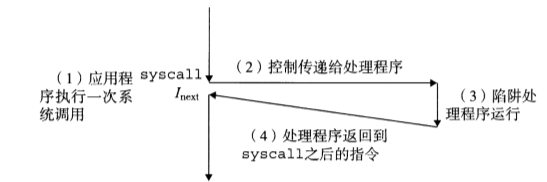


图 16 陷阱处理方式

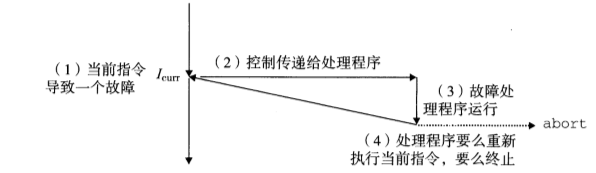


图 17 故障处理方式

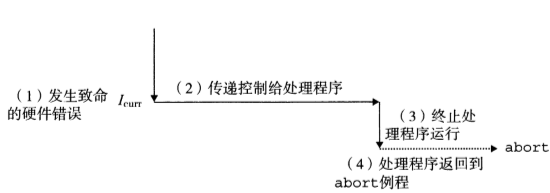


图 18 终止处理方式

Ctrl+Z:收到SIGSTP信号，进程暂时挂起，使用ps查看PID，为352617；使用jobs查看hello的后天job号，为1，再使用fg 1指令将其设置为前台。f

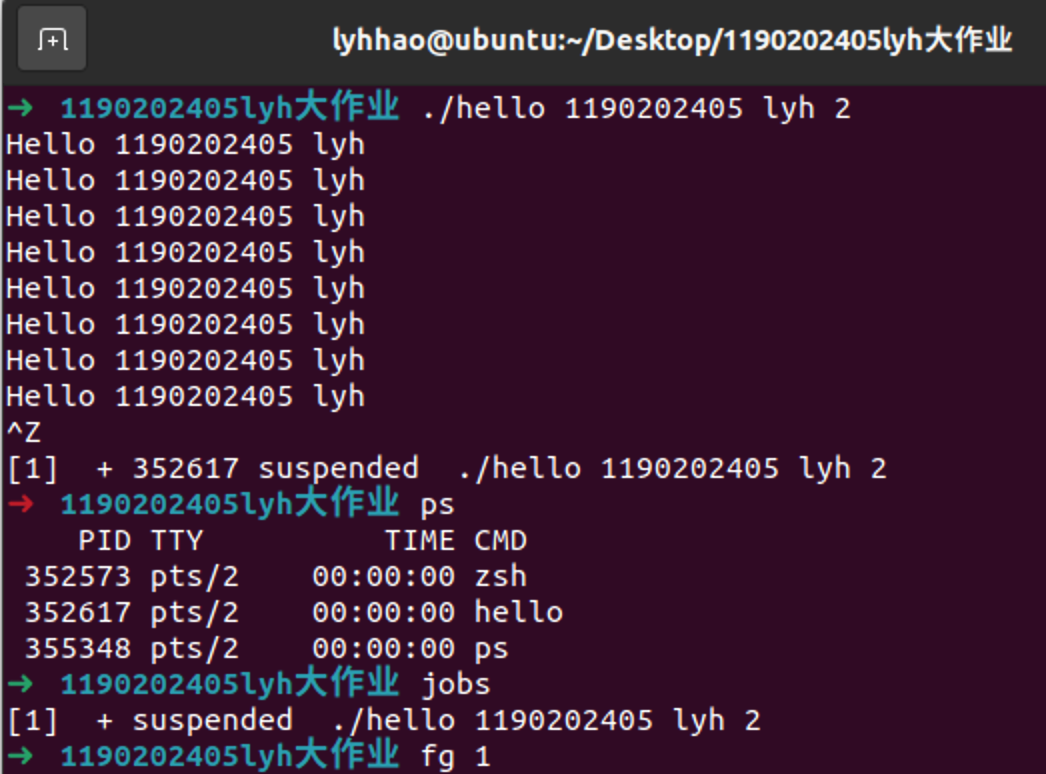


图 19 Ctrl+Z

Ctrl+C：收到SIGINT信号，终止hello。

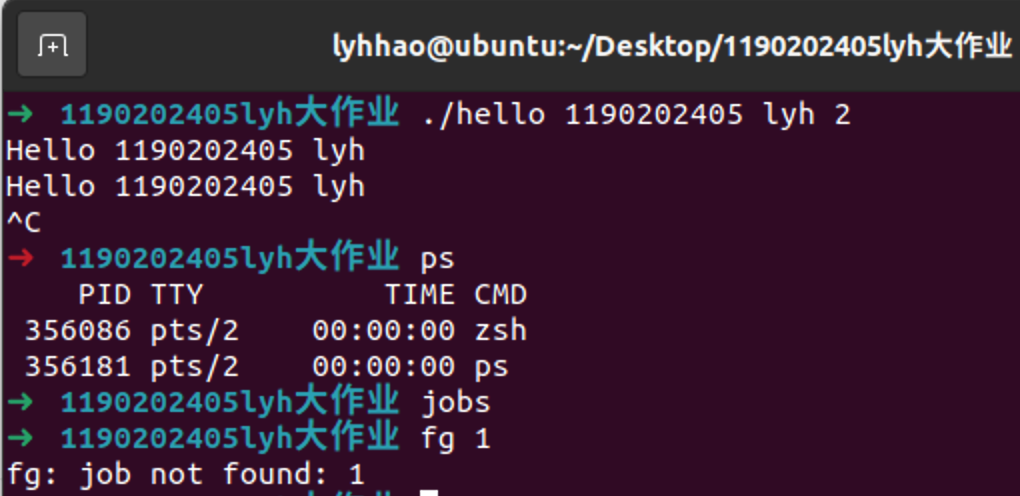


图 20 Ctrl+C

kill命令：终止后台挂起的进程

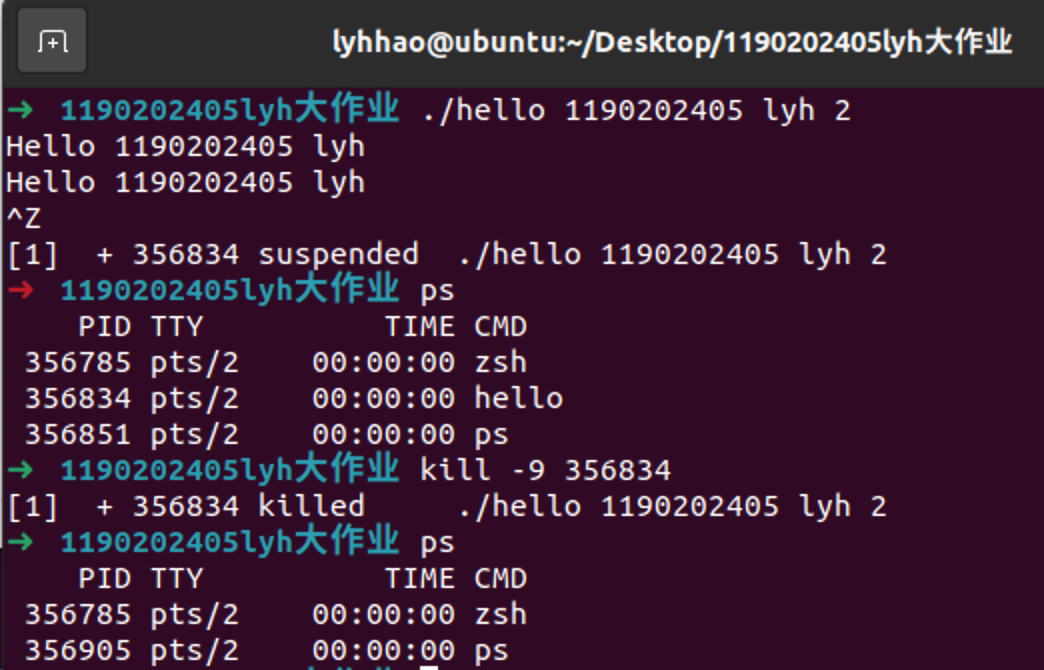


图 21 kill命令

## 6.7本章小结

本章主要介绍了进程的概念与作用以及hello进程的执行过程。了解了Shell的处理流程，进程对信号的获取，与处理。使用fork可以创建新进程，调用execve可以执行进程。对linux的异常控制流有了更加深入的理解。

**（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(logical address)**  
Intel为了兼容，将远古时代的段式内存管理方式保留了下来。逻辑地址指的是机器语言指令中，用来指定一个操作数或者是一条指令的地址。

**线性地址(linear address)或也叫虚拟地址(virtual address)**  
跟逻辑地址类似，它也是一个不真实的地址，如果逻辑地址是对应的硬件平台段式管理转换前地址的话，那么线性地址则对应了硬件页式内存的转换前地址。

**物理地址(physical address)**  
用于内存芯片级的单元寻址，与处理器和CPU连接的地址总线相对应。

**虚拟内存(virtual memory)**  
这是对整个内存的抽象描述。它是相对于物理内存来讲的，可以直接理解成“不直实的”，“假的”内存

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

一个逻辑地址由两部分组成，段标识符，段内偏移量。段标识符是一个16位长的字段组成，称为段选择符，其中前13位是一个索引号。后面三位包含一些硬件细节。  
 索引号，可以通过段标识符的前13位，直接在段描述符表中找到一个具体的段描述符，这个描述符就描述了一个段。  
 这里面，我们只用关心Base字段，它描述了一个段的开始位置的线性地址。  
 全局的段描述符，放在“全局段描述符表(GDT)”中，一些局部的段描述符，放在“局部段描述符表(LDT)”中。  
 GDT在内存中的地址和大小存放在CPU的gdtr控制寄存器中，而LDT则在ldtr寄存器中。  
 给定一个完整的逻辑地址段选择符+段内偏移地址，  
 看段选择符的T1=0还是1，知道当前要转换是GDT中的段，还是LDT中的段，再根据相应寄存器，得到其地址和大小。我们就有了一个数组了。  
 拿出段选择符中前13位，可以在这个数组中，查找到对应的段描述符，这样，它了Base，即基地址就知道了。

把Base + offset，就是要转换的线性地址了

## 7.3 Hello的线性地址到物理地址的变换-页式管理

CPU页式内存管理：

CPU的页式内存管理单元，负责把一个线性地址，最终翻译为一个物理地址。从管理和效率的角度出发，线性地址被分为以固定长度为单位的组，成为页（page），例如一个32位的机器，线性地址最大可为4G，可以用4KB位一个页来划分，这个页，作为一个线性地址就被划分为一个total\_page[2^20]的大数组，共有2的20个次方个页，这个大数组我们称之为页目录。目录中的每一个目录项，就是一个地址一一对应的页的地址。

另一类“页“，我们称之为物理页，或者是页框,是分页段元把所有的物理内存也划分为固定长度的管理单位，它的长度一般与内存页是一一对应的。

这里注意到，total\_page数组有2^20个成员，每个成员是一个地址（32位机，一个地址也就是4字节），那么要单单表示这么一个数组，就要占去4MB的内存空间，为了节省空间，引入了一个二级管理模式的机器来组织分页单元。

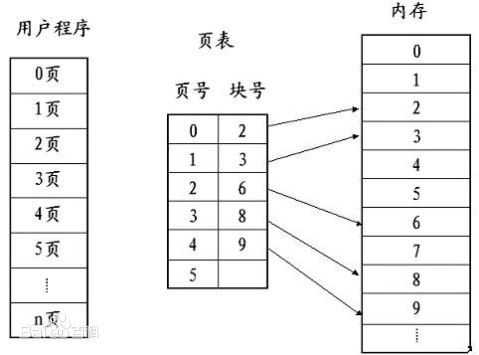


图 22 页式管理

优点

1、由于它不要求作业或进程的程序段和数据在内存中连续存放，从而有效地解决了碎片问题。

2、动态页式管理提供了内存和外存统一管理的虚存实现方式，使用户可以利用的存储空间大大增加。这既提高了主存的利用率，又有利于组织多道程序执行。

缺点

1、要求有相应的硬件支持。例如地址变换机构，缺页中断的产生和选择淘汰页面等都要求有相应的硬件支持。这增加了机器成本。

2、增加了系统开销，例如缺页中断处理机，

3、请求调页的算法如选择不当，有可能产生抖动现象。

4、虽然消除了碎片，但每个作业或进程的最后一页内总有一部分空间得不到利用果页面较大，则这一部分的损失仍然较大。

## 7.4 TLB与四级页表支持下的VA到PA的变换

每次CPU产生一个虚拟地址，MMU（内存管理单元）就必须查阅一个PTE（页表条目），以便将虚拟地址翻译为物理地址。在最糟糕的情况下，这会从内存多取一次数据，代价是几十到几百个周期。如果PTE碰巧缓存在L1中，那么开销就会下降1或2个周期。然而，许多系统都试图消除即使是这样的开销，它们在MMU中包括了一个关于PTE的小的缓存，称为翻译后备缓存器（TLB）。

多级页表：

将虚拟地址的VPN划分为相等大小的不同的部分，每个部分用于寻找由上一级确定的页表基址对应的页表条目。

TLB采用组相联的方式;TLB由TLB标记,PTE虚拟地址被划分位:a1位 a2位 a3位 a4位。

MMU先在TLB中寻找PTE；根据虚拟地址的后p+t-1位值作为组号，找到其对应的TLB的组，VPN剩下的位作为标记，与该组中的标记进行对比，如果存在标记相同切有效为为1的pte那么获得pte中的ppn如果不存在，那么将vpn发送给内存和cache。通过一个固定寄存器中的值获得以及页表的基地址，由a1位值作为索引，知道对应表中的值，将其作为二级页表的机制，由a2位值作为索引，直到有a4作为索引获得四级页表的pte的值，这样就可以得到vpn所对应的ppn的值，再由ppn+vpo就可以得到va对应的pa

## 7.5 三级Cache支持下的物理内存访问

CPU发送一条虚拟地址，随后MMU按照上述操作获得了物理地址PA。根据cache大小组数的要求，将PA分为CT（标记位）CS(组号)，CO（偏移量）。根据CS寻找到正确的组，比较每一个cacheline是否标记位有效以及CT是否相等。如果命中就直接返回想要的数据，如果不命中，就依次去L2,L3,主存判断是否命中，当命中时，将数据传给CPU同时更新各级cache的cacheline（如果cache已满则要采用换入换出策略）。

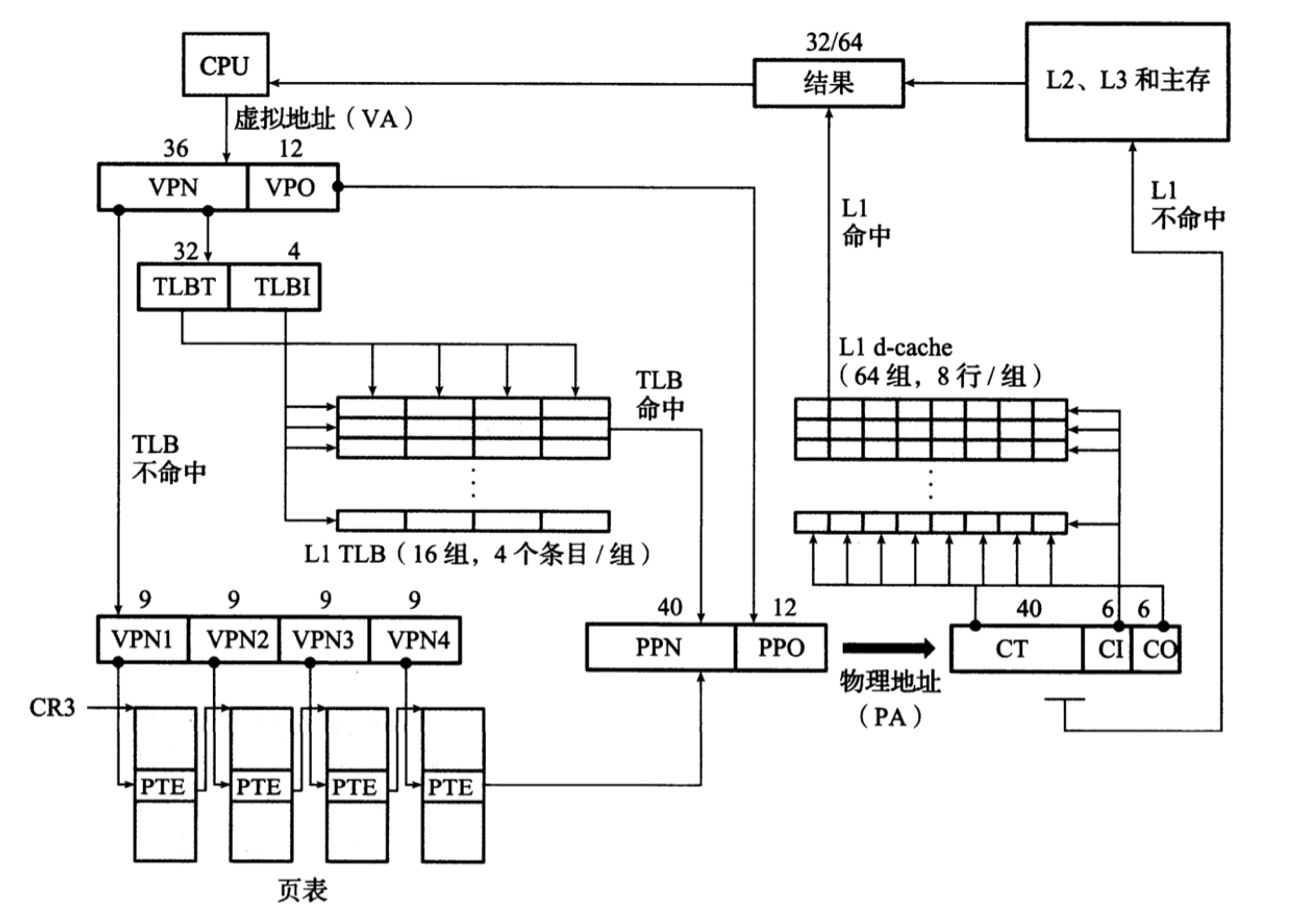


图 23 三级Cache

## 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，同时为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位只读，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空间地址的抽象概念。

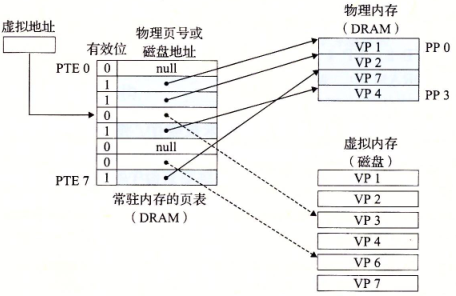
## 7.7 hello进程execve时的内存映射

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。栈和堆区域也是请求二进制零的,初始长度为零。
2. 映射私有区域。如果hello程序与共享对象(或目标)链接,比如标准C库1ibc.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
3. 设置程序计数器(PC)。 execve做的最后一件事情就是设置当前进程上下文中的程序计数器,使之指向代码区域的入口点。
4. 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。

## 7.8 缺页故障与缺页中断处理

DRAM缓存不命中称为缺页。例如，CPU引用了VP3中的一个字，VP3并未缓存，并且触发了一个缺页异常。缺页异常调用内核中的缺页异常处理程序，该应许会选择一个牺牲页，在此例中就是存放在PP3中的VP4。如果VP4已经被修改了，那么内核就会将它拷贝回磁盘。



接着，内核从磁盘拷贝VP3到存储器中的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。这时，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。

## 7.9动态存储分配管理

动态储存分配管理使用动态内存分配器来进行。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。空闲块可以用来分配。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的状态，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。动态内存分配主要有两种基本方法与策略：

**带边界标签的隐式空闲链表分配器管理**

带边界标记的隐式空闲链表的每个块是由一个字的头部、有效载荷、可能的额外填充以及一个字的尾部组成的。

隐式空闲链表：在隐式空闲链表中，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。其中，一个设置了已分配的位而大小为零的终止头部将作为特殊标记的结束块。

当一个应用请求一个k字节的块时，分配器搜索空闲链表，查找一个足够大的可以放置所请求块的空闲块。分配器有三种放置策略：首次适配、下一次适配合最佳适配。分配完后可以分割空闲块减少内部碎片。同时分配器在面对释放一个已分配块时，可以合并空闲块，其中便利用隐式空闲链表的边界标记来进行合并。

**显示空间链表管理**

显式空闲链表是将空闲块组织为某种形式的显式数据结构。因为根据定义，程序不需要一个空闲块的主体，所以实现这个数据结构的指针可以存放在这些空闲块的主体里面。如，堆可以组织成一个双向链表，在每个空闲块中，都包含一个前驱与一个后继指针。

显式空闲链表：在显式空闲链表中。可以采用后进先出的顺序维护链表，将最新释放的块放置在链表的开始处，也可以采用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的后继地址，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。

## 7.10本章小结

本章主要介绍了hello的存储器地址空间、段式管理与页式管理，介绍了VA到PA的转换、物理内存访问，还介绍了hello进程fork与execve时的内存映射，还介绍了缺页故障与缺页中断处理、动态内存分配。

**（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

设备管理：unix io接口

将设备映射为文件，允许Linux内核使用一个底层的API，称为Unix IO。

对文件有以下操作：

open：打开文件

close：关闭文件

read：读取文件

write：写入文件

lseek：改变当前文件位置

## 8.2 简述Unix IO接口及其函数

Unix IO接口操作：

打开文件：内核返回一个非负整数的文件描述符，通过对此文件描述符对文件进行所有操作。

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（文件描述符0）、标准输出（描述符为1），标准出错（描述符为2）。头文件<unistd.h>定义了常量STDIN\_FILENO、STDOUT\_FILENO、STDERR\_FILENO，他们可用来代替显式的描述符值。

改变当前的文件位置，文件开始位置为文件偏移量，应用程序通过seek操作，可设置文件的当前位置为k。

读写文件，读操作：从文件复制n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；写操作：从内存复制n个字节到文件，当前文件位置为k，然后更新k

关闭文件：当应用完成对文件的访问后，通知内核关闭这个文件。内核会释放文件打开时创建的数据结构，将描述符恢复到描述符池中

Unix IO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 是需要关闭的文件的描述符，close 返回操作结果。

3. ssize\_t read(int fd,void \*buf,size\_t n)，read 函数从描述符为 fd 的当前文 件位置赋值最多 n 个字节到内存位置 buf。返回值-1 表示一个错误，0 表示 EOF，否则返回值表示的是实际传送的字节数量。

4. ssize\_t 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。

5. off\_t lseek(int fd, off\_t offset,int whence)，lseek函数将fd文件描述符所指定的文件移动offset字节长度的距离，若成功，返回位移的长度，若失败，返回-1。

## 8.3 printf的实现分析

printf函数：

|  |
| --- |
| nt 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;       } |

vsprintf函数：

|  |
| --- |
| int vsprintf(char \*buf, const char \*fmt, va\_list args)  {  char\* p;  char tmp[256];  va\_list p\_next\_arg = args;    for (p=buf;\*fmt;fmt++) {  if (\*fmt != '%') {  \*p++ = \*fmt;  continue;  }    fmt++;    switch (\*fmt) {  case 'x':  itoa(tmp, \*((int\*)p\_next\_arg));  strcpy(p, tmp);  p\_next\_arg += 4;  p += strlen(tmp);  break;  case 's':  break;  default:  break;  }  }    return (p - buf);  } |

vsprintf函数将所有的参数内容格式化之后存入buf，然后返回格式化数组的长度，write函数将buf中的i个元素写入终端。

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等.

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）。

## 8.4 getchar的实现分析

getchar有一个int型的返回值。当程序调用getchar时.程序就等着用户按键。用户输入的字符被存放在键盘缓冲区中。直到用户按回车为止（回车字符也放在缓冲区中）。当用户键入回车之后，getchar才开始从stdin流中每次读入一个字符。getchar函数的返回值是用户输入的字符的ASCII码，若文件结尾(End-Of-File)则返回-1(EOF)，且将用户输入的字符回显到屏幕。如用户在按回车之前输入了不止一个字符，其他字符会保留在键盘缓存区中，等待后续getchar调用读取。也就是说，后续的getchar调用不会等待用户按键，而直接读取缓冲区中的字符，直到缓冲区中的字符读完后，才等待用户按键。

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设备的基本概念和管理机制，分析了Unix IO接口以及相关函数，还简要说明了printf和getchar函数的实现方法与操作过程。

**（第8章1分）**

# 结论

1. 预处理：将hello.c使用的外部库拓展整合，得到hello.i文件
2. 编译：hello.i经过编译，得到hello.s文件
3. 汇编：汇编器将汇编代码文件hello.s转为二进制可重定位目标文件hello.o
4. 链接：将hello.o以及其他可重定位目标文件和动态链接库链接成为可执行目标文件hello
5. 输入运行指令：./hello 1190202405 lyh 2
6. shell调用fork函数，为hello程序创建子进程
7. 调用execve函数加载运行hello，加载映射虚拟内存，在当前进程的上下文中加载运行hello。
8. 访存，hello运行时，会出现对内存地址的访问，内存管理单元将VA翻译为PA
9. hello运行时执行各种函数，如：printf等，调用malloc，申请堆中内存。
10. 异常处理，若有Ctrl+Z或Ctrl+C等信号，hello进程捕获这些信号，并执行相应的异常处理。
11. hello运行结束，最终被父进程或者init回收，hello运行的所有信息都被内核回收。

操作系统是计算机中最重要的部分，经过本课程的学习，我理解了程序在操作系统中的运行方式，也深入了解了计算机中数据的存储方式，汇编指令的含义，CPU的运行逻辑，程序的优化方法，存储器的层次结构，进程以及系统级IO。CSAPP教材更是深入浅出，为我们的学习提供了重要的参考。

**（结论0分，缺失 -1分，根据内容酌情加分）**

# 附件

列出所有的中间产物的文件名，并予以说明起作用。

|  |  |
| --- | --- |
| **文件名** | **文件的作用** |
| **hello.i** | 预处理后的文件 |
| **hello.s** | 编译之后的汇编文件 |
| **hello.o** | 汇编之后的可重定位目标文件 |
| **Hello** | 链接之后的可执行目标文件 |
| **Elf.txt** | Hello.o 的 ELF 格式 |
| **Disas\_hello.s** | Hello.o 的反汇编代码 |
| **hello1.elf** | hello的ELF 格式 |
| **hello1\_objdump.s** | hello 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] Randal E.Bryant，David R.O’Hallaro.深入理解计算机系统（第三版）[M].北京：机械工业出版社，2017

[2] 袁春凤.计算机系统基础[M].北京：机械工业出版社，2016

[3] 唐朔飞.计算机组成原理（第二版）[M].北京：高等教育出版社，2008

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

[5] Acronyms relevant to Executable and Linkable Format. https://www. cs.stevens.edu/~jschauma/631/elf.html

**（参考文献0分，缺失 -1分）**