

哈爾濱工業大學

計算機系統

大作業

題 目	<u>程序人生-Hello's P2P</u>
專 業	<u>計算機類</u>
學 號	<u>1170300133</u>
班 級	<u>1703001</u>
學 生	<u>孟月陽</u>
指 導 教 師	<u>史先俊</u>

計算機科學與技術學院
2018 年 12 月

摘 要

本文主要通过这学期在计算机系统课程和实验中学习到的知识，分析一个五脏俱全的 hello 程序在 64 位 ubuntu 环境下运行的过程，着重了解程序的 p2p, 020 过程，并利用现阶段会使用的各种工具，见证 Linux 框架下一个 hello 程序的生命周期，并对其中的知识进行深入了解和探讨。

关键词：hello；程序的生命周期；p2p, 020；

（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 -
1.1 HELLO 简介	- 4 -
1.2 环境与工具	- 4 -
1.3 中间结果	- 4 -
1.4 本章小结	- 4 -
第 2 章 预处理	- 6 -
2.1 预处理的概念与作用	- 6 -
2.2 在 UBUNTU 下预处理的命令	- 6 -
2.3 HELLO 的预处理结果解析	- 7 -
2.4 本章小结	- 8 -
第 3 章 编译	- 10 -
3.1 编译的概念与作用	- 10 -
3.2 在 UBUNTU 下编译的命令	- 10 -
3.3 HELLO 的编译结果解析	- 11 -
3.4 本章小结	- 14 -
第 4 章 汇编	- 16 -
4.1 汇编的概念与作用	- 16 -
4.2 在 UBUNTU 下汇编的命令	- 16 -
4.3 可重定位目标 ELF 格式	- 16 -
4.4 HELLO.O 的结果解析	- 19 -
4.5 本章小结	- 21 -
第 5 章 链接	- 22 -
5.1 链接的概念与作用	- 22 -
5.2 在 UBUNTU 下链接的命令	- 22 -
5.3 可执行目标文件 HELLO 的格式	- 23 -
5.4 HELLO 的虚拟地址空间	- 25 -
5.5 链接的重定位过程分析	- 25 -
5.6 HELLO 的执行流程	- 27 -
5.7 HELLO 的动态链接分析	- 27 -
5.8 本章小结	- 28 -
第 6 章 HELLO 进程管理	- 29 -
6.1 进程的概念与作用	- 29 -

6.2 简述壳 SHELL-BASH 的作用与处理流程.....	- 29 -
6.3 HELLO 的 FORK 进程创建过程	- 29 -
6.4 HELLO 的 EXECVE 过程	- 30 -
6.5 HELLO 的进程执行.....	- 30 -
6.6 HELLO 的异常与信号处理	- 31 -
6.7 本章小结	- 33 -
第 7 章 HELLO 的存储管理.....	- 35 -
7.1 HELLO 的存储器地址空间	- 35 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理.....	- 35 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理	- 36 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换.....	- 37 -
7.5 三级 CACHE 支持下的物理内存访问	- 38 -
7.6 HELLO 进程 FORK 时的内存映射	- 39 -
7.7 HELLO 进程 EXECVE 时的内存映射	- 40 -
7.8 缺页故障与缺页中断处理.....	- 40 -
7.9 动态存储分配管理	- 41 -
7.10 本章小结	- 42 -
第 8 章 HELLO 的 IO 管理	- 43 -
8.1 LINUX 的 IO 设备管理方法	- 43 -
8.2 简述 UNIX IO 接口及其函数	- 43 -
8.3 PRINTF 的实现分析.....	- 43 -
8.4 GETCHAR 的实现分析.....	- 44 -
8.5 本章小结	- 44 -
结论	- 44 -
附件	- 45 -
参考文献.....	- 46 -

第 1 章 概述

1.1 Hello 简介

P2P 过程：程序员通过使用编辑器从 0 创建 `hello.c`，`hello.c` 文件通过 IO 设备经由总线到达并存入主存，然后经由预处理，编译，汇编，链接成为一个可执行目标文件 `hello`，即一个 `program`。使用 `shell` 创建一个新的进程并使用 `execve` 将 `hello` 加载进内存，开始作为 `process` 运行，即：P2P（program to process）

020 过程：`hello` 从不占用内存开始运行到在内存中拥有私有的内存地址，再到程序结束，进程被回收，整个过程中，`hello` 从硬盘中到在内存占有一定空间，之后又从内存中被删除，不留下一点东西在内存中，所以是 020（Zero to Zero）

1.2 环境与工具

环境：

硬件：intel i7 8550u x86-64，128G SSD + 1T HDD

软件：Windows 10 家庭版，Vmware Workstation Pro 14.1.3，Ubuntu 64 位

工具：sublime，gcc，readelf，预处理器 `cpp`，Hexdit

1.3 中间结果

<code>hello.i</code>	预处理文件
<code>hello.s</code>	汇编语言文件
<code>hello.o</code>	可重定位目标文件
<code>hello</code>	可执行程序文件
<code>hello.elf</code>	<code>hello.o</code> 的 elf 格式文件
<code>hello_objdump.txt</code>	objdump 的反汇编文件

1.4 本章小结

本章简单介绍了 `hello` 程序的 P2P，020 过程，并列出了实验环境，所用工具和中

间产生的结果

(第 1 章 0.5 分)

第 2 章 预处理

2.1 预处理的概念与作用

概念：C 语言标准规定，预处理是指前 4 个编译阶段：

- 1: 三字符组与双字符组的替换；
- 2: 行拼接；
- 3: 单词化；
- 4: 宏扩展与预处理指令处理。

作用：预扫描源代码，完成头文件的包含，宏扩展，条件编译，行控制（line control）等操作：

1: 三字符组与双字符组的替换：将 C 程序设计语言中 3 个或 2 个字符的序列替换为单个字符。

2: 行拼接（Line splicing）：把物理源码行（Physical source line）中的换行符转义字符处理为普通的换行符，从而把源程序处理为逻辑行的顺序集合。

3: 单词化（Tokenization）：处理每行的空白、注释等，使每行成为 token 的顺序集。删除所有的//，/**/类型注释；

4: 宏扩展与预处理指令处理：根据以字符‘#’开头的命令，修改原始的 C 程序。插入所有用#include 命令指定的文件；扩展所用#define 声明指定的宏；处理所有条件编译指令，如#if, #ifdef 等

2.2 在 Ubuntu 下预处理的命令

```
gcc -E hello.c
```

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

2-0-1在Ubuntu下预处理的命令

```
myy1170300133@ubuntu-64-2018ics:~/hitcs/homework$ gcc -E hello.c
# 1 "hello.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "hello.c"
```

2.3 Hello 的预处理结果解析

使用 sublime 编辑器打开：

Main 函数在 3102 行，hello.c 完整代码出现在 3099 行。前面的 3000 多行则是相应的预处理，和一堆换行符。

注意此时注释已经全部去掉

2-0-2 Hello的预处理结果解析1

```
3101
3102 int main(int argc, char *argv[])
3103 {
3104     int i;
3105
3106     if(argc!=3)
3107     {
3108         printf("Usage: Hello 学号 姓名!\n");
3109         exit(1);
3110     }
3111     for(i=0;i<10;i++)
3112     {
3113         printf("Hello %s %s\n",argv[1],argv[2]);
3114         sleep(sleepsecs);
3115     }
3116     getchar();
3117     return 0;
3118 }
```

前面 57 到 83 行，以及后面的 typedef 等都是对数据类型的定义解释：

2-0-3 Hello的预处理结果解析2

```
57 typedef signed char __int8_t;
58 typedef unsigned char __uint8_t;
59 typedef signed short int __int16_t;
60 typedef unsigned short int __uint16_t;
61 typedef signed int __int32_t;
62 typedef unsigned int __uint32_t;
63
64 typedef signed long int __int64_t;
65 typedef unsigned long int __uint64_t;
66
67
68
69
70
71
72
73 typedef long int __quad_t;
74 typedef unsigned long int __u_quad_t;
```

除此之外出现的还有头文件 `stdio.h` `unistd.h` `stdlib.h` 的插入展开等宏扩展和预处理指令处理：

2-0-4 Hello的预处理结果解析3

```
12 # 1 "/usr/include/stdio.h" 1 3 4
13 # 27 "/usr/include/stdio.h" 3 4
```

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`，因为发现其中依然使用了 `#define` 语句，所以预处理器 `cpp` 对此进行递归展开，所以最终 `.i` 预处理文件中是没有 `#define` 的。对于 `.c` 源程序文件中使用的 `#ifdef` `#ifndef` 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。其他类似。

2.4 本章小结

预处理的过程我一开始是忽略了的，复习前还直接认为源文件直接通过编译从.c 文件到.s 文件，复习时才发现中间还有一个预处理的环节。处理宏扩展，插入相应文件，行拼接和单词化，这样编译器就能够更好的读懂这些代码了吧。

(第 2 章 0.5 分)

第 3 章 编译

3.1 编译的概念与作用

概念：编译器（cc1）将文本文件 `hello.i` 翻译成文本文件 `hello.s`，`hello.s` 中包含一个相应的汇编语言程序。

作用：将预处理完的文件进行一系列词法分析，语法分析，语义分析及优化后生成相应的汇编代码文件

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过程中的汇编指令

指令	含义
<code>.file</code>	声明源文件
<code>.text</code>	以下是代码段
<code>.section .rodata</code>	以下是 rodata 节
<code>.globl</code>	声明一个全局变量
<code>.type</code>	用来指定是函数类型或是对象型
<code>.size</code>	声明大小
<code>.long .string</code>	声明一个 long、string 类型
<code>.align</code>	声明对指令或者数据的存放地址进行对齐的方式

3.2 在 Ubuntu 下编译的命令

还是使用 gcc 编译器

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

3-0-1在Ubuntu下编译的命令

```

myy1170300133@ubuntu-64-2018ics:~/hitics/homework$ gcc -S hello.i -o hello.s
myy1170300133@ubuntu-64-2018ics:~/hitics/homework$ gedit hello.s

```

```

.file "hello.c"
.text
.globl sleepsecs
.data
.align 4
.type sleepsecs, @object
.size sleepsecs, 4
sleepsecs:
    .long 2
.section .rodata
.LC0:
    .string "Usage: Hello \345\255\246\345\217\267 \345\247\223\345\220\215\357\274\201"
.LC1:
    .string "Hello %s %s\n"
.text
.globl main
.type main, @function
main:
.LFB5:
    .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)
    cmpl $3, -20(%rbp)
    je .L2
    leaq .LC0(%rip), %rdi
    call puts@PLT
    movl $1, %edi

```

应截图，展示编译过程！

3.3 Hello 的编译结果解析

1: 数据:

1. `.globl sleepsecs`#数据 `sleepsecs`，指明是全局变量
2. `.data`
3. `.align 4`
4. `.type sleepsecs, @object`
5. `.size sleepsecs, 4`#大小为 4 个字节
6. `sleepsecs:`#赋值操作：指明全局变量 `sleepsecs`，并赋值为 2，放在 `rodata` 只读数据段
7. `.long 2`#存在隐式类型转换，将 2.5 转换为 `int` 型 2 赋值给 `sleepsecs`
8. `.section .rodata`

2: 赋值操作:

`sleepsecs` 赋值:

1. `sleepsecs:`#赋值操作：指明全局变量 `sleepsecs`，并赋值为 2，放在 `rodata` 只读数据段
2. `.long 2`#存在隐式类型转换，将 2.5 转换为 `int` 型 2 赋值给 `sleepsecs`

`Int i` 赋值

1. `movl $0, -4(%rbp)`#赋值操作：将 0 赋值给变量 `i`

3: 类型转换:

`sleepsecs` 定义为 `int` 型数据, 赋值时使用 2.5 浮点数, 存在隐式类型转换, 将 2.5 转换为 `int` 型数据 2

```
1. sleepsecs:#赋值操作: 指明全局变量 sleepsecs, 并赋值为 2, 放在 rodata 只读数据段
2.     .long    2#存在隐式类型转换, 将 2.5 转换为 int 型 2 赋值给 sleepsecs
3.     .section  .rodata
```

4: 算术运算:

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

`{}` 循环中, 存在 `i++` 算术运算

```
1.     addl     $1, -4(%rbp)#算术操作: i++
```

5: 关系操作:

Main 函数体中的选择语句: `if(argc!=3)` 中, 将参数 `argc` 和 3 进行比较:

```
1.     cmpl     $3, -20(%rbp)#关系操作 argc!=3
```

main 函数体中的 `for(i=0;i<10;i++)` 循环条件, 将 `i` 和 10 进行比较, 编译成 `i` 和 9 进行比较:

```
1.     cmpl     $9, -4(%rbp)#关系操作: i<10
```

6: 控制转移:

利用关系操作, 不等于时跳转到.L2

```
1.     cmpl     $3, -20(%rbp)#关系操作 argc!=3
2.     je      .L2#控制转移: 第一个参数不等于 3 时, 跳入.L2 中执行
```

进入循环体的控制转移

```
3.     movl     $0, -4(%rbp)#赋值操作: 将 0 赋值给变量 i
4.     jmp     .L3 #控制转移
```

利用关系操作, 控制循环体退出和继续

```
5.     cmpl     $9, -4(%rbp)#关系操作: i<10
6.     jle     .L4 #控制转移: i<=9 时继续.L4 的循环
```

7: 函数操作:

调用 printf 函数

```

1. #函数操作: printf("Usage: Hello 学号 姓名! \n");
2.     leaq    .LC0(%rip), %rdi

    call    puts@PLT

1. #函数操作: printf("Hello %s %s\n",argv[1],argv[2]);
2.     movq    -32(%rbp), %rax
3.     addq    $16, %rax
4.     movq    (%rax), %rdx
5.     movq    -32(%rbp), %rax
6.     addq    $8, %rax
7.     movq    (%rax), %rax
8.     movq    %rax, %rsi
9.     leaq    .LC1(%rip), %rdi
10.    movl    $0, %eax
11.    call    printf@PLT

```

调用 sleep 函数

```

1. #函数操作: sleep(sleepsecs);
2.     movl    sleepsecs(%rip), %eax
3.     movl    %eax, %edi
4.     call    sleep@PLT

```

调用

```

7.     .file    "hello.c"
8.     .text
9.     .globl  sleepsecs#数据 sleepsecs, 指明是全局变量
10.    .data
11.    .align 4
12.    .type   sleepsecs, @object
13.    .size   sleepsecs, 4#大小为 4 个字节
14. sleepsecs:#赋值操作: 指明全局变量 sleepsecs, 并赋值为 2, 放在 rodata 只读数据段
15.     .long   2#存在隐式类型转换, 将 2.5 转换为 int 型 2 赋值给 sleepsecs
16.     .section .rodata
17. .LC0:#两个 printf 函数中的字符串, 表明在 main 函数中出现, 存放在 text 代码段
18.     .string "Usage: Hello \345\255\246\345\217\267 \345\247\223\345\220\215\357\274\201"
19. .LC1:
20.     .string "Hello %s %s\n"
21.     .text
22.     .globl  main
23.     .type   main, @function
24. main: #main 函数主体开始
25. .LFB5:
26.     .cfi_startproc
27.     pushq   %rbp
28.     .cfi_def_cfa_offset 16
29.     .cfi_offset 6, -16

```

```

30.    movq    %rsp, %rbp
31.    .cfi_def_cfa_register 6
32.    subq    $32, %rsp
33.    movl    %edi, -20(%rbp)#main 函数的第一个参数
34.    movq    %rsi, -32(%rbp)
35.    cmpl    $3, -20(%rbp)#关系操作 argc!=3
36.    je     .L2#控制转移: 第一个参数不等于 3 时, 跳入.L2 中执行
37.
38.    #函数操作: printf("Usage: Hello 学号 姓名! \n");
39.    leaq    .LC0(%rip), %rdi
40.    call    puts@PLT
41.
42.    #函数操作: exit(1);
43.    movl    $1, %edi
44.    call    exit@PLT
45. .L2:
46.    movl    $0, -4(%rbp)#赋值操作: 将 0 赋值给变量 i
47.    jmp     .L3 #控制转移
48. .L4:
49.    #函数操作: printf("Hello %s %s\n",argv[1],argv[2]);
50.    movq    -32(%rbp), %rax
51.    addq    $16, %rax
52.    movq    (%rax), %rdx
53.    movq    -32(%rbp), %rax
54.    addq    $8, %rax
55.    movq    (%rax), %rax
56.    movq    %rax, %rsi
57.    leaq    .LC1(%rip), %rdi
58.    movl    $0, %eax
59.    call    printf@PLT
60.    #函数操作: sleep(sleepsecs);
61.    movl    sleepsecs(%rip), %eax
62.    movl    %eax, %edi
63.    call    sleep@PLT
64.    addl    $1, -4(%rbp)#算术操作: i++
65. .L3:
66.    cmpl    $9, -4(%rbp)#关系操作: i<10
67.    jle     .L4 #控制转移: i<=9 时继续.L4 的循环
68.    call    getchar@PLT
69.    movl    $0, %eax
70.    leave
71.    .cfi_def_cfa 7, 8
72.    ret
73.    .cfi_endproc
74. .LFE5:
75.    .size   main, .-main
76.    .ident  "GCC: (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0"
77.    .section .note.GNU-stack,"",@progbits

```

3.4 本章小结

编译章节主要讲述了编译器是如何处理 C 语言的各个数据类型以及各中操作的, 并结合 hello.c C 代码到 hello.s 汇编代码之间的映射关系作出关于汇编代码操

作的解释。

编译器将.i 的预处理文件编译为.s 的汇编代码文件。经过编译之后，我们的hello 自 C 语言解析为更加靠近机器底层的汇编语言。而汇编语言是直接面向处理器的程序设计语言，作用对象是寄存器或者存储器，执行效率很高，同时也因为靠近底层而更加复杂，又因为不同处理器使用的指令集不同而有多种汇编语言，所以使用汇编语言编程往往很困难，很头秃。

编译特点：1：预处理文件.i 转汇编语言文件.s

2：执行效率较高；

3：编写和调试复杂

(第3章2分)

第 4 章 汇编

4.1 汇编的概念与作用

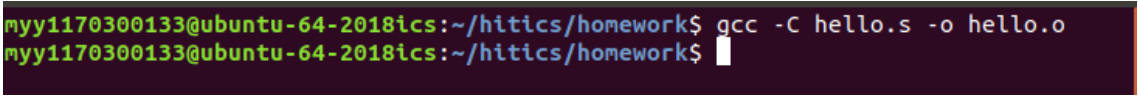
概念：汇编器(as) 将汇编语言文件 `hello.s` 翻译成机器语言指令，把这些指令打包成一种叫做可重定位目标程序(relocatable object program)的格式，并将翻译结果保存在目标文件 `hello.o` 中。

作用：汇编器是将汇编代码转变成机器可以执行的命令，每一个汇编语句几乎都对应一条机器指令。汇编相对于编译过程比较简单，根据汇编指令和机器指令的对照表一一翻译即可。

4.2 在 Ubuntu 下汇编的命令

```
gcc -c hello.s -o hello.o
```

4-0-1在Ubuntu下汇编的命令



```
myy1170300133@ubuntu-64-2018ics:~/hitics/homework$ gcc -c hello.s -o hello.o
myy1170300133@ubuntu-64-2018ics:~/hitics/homework$
```

4.3 可重定位目标 elf 格式

elf 头：以 16b 的序列 magic 开始，magic 描述了生成该文件的系统的字的大小和字节顺序，elf 头剩下的部分主要包含了帮助链接器进行语法分析和解释目标文件的信息，其中包括目标文件的类型、elf 头的大小、机器大小端类型、，以及节头部表中条目的大小和数量等信息。

4-0-2可重定位目标elf格式

```
ELF 头:
Magic:    7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
类别:                                ELF64
数据:                                2 补码, 小端序 (little endian)
版本:                                1 (current)
OS/ABI:    UNIX - System V
ABI 版本:    0
类型:                                DYN (共享目标文件)
系统架构:    Advanced Micro Devices X86-64
版本:                                0x1
入口点地址:    0x660
程序头起点:    64 (bytes into file)
Start of section headers:    6656 (bytes into file)
标志:    0x0
本头的大小:    64 (字节)
程序头大小:    56 (字节)
Number of program headers:    9
节头大小:    64 (字节)
节头数量:    29
字符串表索引节头:    28
```

节头信息：包含了文件中出现的各个节的语义，包括节的名称，类型、位置和大小等信息。

4-0-3可重定位目标elf格式

节头:						
[号]	名称 大小	类型 全体大小	地址 旗标	链接 信息	偏移量 对齐	
[0]	0000000000000000	NULL	0000000000000000	0 0	0	
[1]	.interp 000000000000001c	PROGBITS 0000000000000000	000000000000238 A	0 0	0 1	
[2]	.note.ABI-tag 0000000000000020	NOTE 0000000000000000	000000000000254 A	0 0	0 4	
[3]	.note.gnu.build-id 0000000000000024	NOTE 0000000000000000	000000000000274 A	0 0	0 4	
[4]	.gnu.hash 000000000000001c	GNU_HASH 0000000000000000	000000000000298 A	5 0	8	
[5]	.dynsym 0000000000000108	DYNSYM 0000000000000018	0000000000002b8 A	6 1	8	
[6]	.dynstr 000000000000009c	STRTAB 0000000000000000	0000000000003c0 A	0 0	1	
[7]	.gnu.version 0000000000000016	VERSYM 0000000000000002	00000000000045c A	5 0	2	
[8]	.gnu.version_r 0000000000000020	VERNEED 0000000000000000	000000000000478 A	6 1	8	
[9]	.rela.dyn 00000000000000c0	RELA 0000000000000018	000000000000498 A	5 0	8	
[10]	.rela.plt 0000000000000078	RELA 0000000000000018	000000000000558 AI	5 22	8	
[11]	.init 0000000000000017	PROGBITS 0000000000000000	0000000000005d0 AX	0 0	4	
[12]	.plt 0000000000000060	PROGBITS 0000000000000010	0000000000005f0 AX	0 0	16	
[13]	.plt.got 0000000000000008	PROGBITS 0000000000000008	000000000000650 AX	0 0	8	
[14]	.text 0000000000000202	PROGBITS 0000000000000000	000000000000660 AX	0 0	16	
[15]	.fini 0000000000000009	PROGBITS 0000000000000000	000000000000864 AX	0 0	4	
[16]	.rodata 000000000000002f	PROGBITS 0000000000000000	000000000000870 A	0 0	4	
[17]	.eh_frame_hdr 000000000000003c	PROGBITS 0000000000000000	0000000000008a0 A	0 0	4	
[18]	.eh_frame 0000000000000108	PROGBITS 0000000000000000	0000000000008e0 A	0 0	8	

```

[19] .init_array      INIT_ARRAY      0000000000200d98 00000d98
      0000000000000008 0000000000000008 WA      0      0      8
[20] .fini_array      FINI_ARRAY      0000000000200da0 00000da0
      0000000000000008 0000000000000008 WA      0      0      8
[21] .dynamic         DYNAMIC        0000000000200da8 00000da8
      00000000000001f0 000000000000010 WA      6      0      8
[22] .got            PROGBITS       0000000000200f98 00000f98
      0000000000000068 0000000000000008 WA      0      0      8
[23] .data            PROGBITS       0000000000201000 00001000
      0000000000000014 0000000000000000 WA      0      0      8
[24] .bss             NOBITS         0000000000201014 00001014
      0000000000000004 0000000000000000 WA      0      0      1
[25] .comment         PROGBITS       0000000000000000 00001014
      000000000000002a 0000000000000001 MS      0      0      1
[26] .symtab           SYMTAB         0000000000000000 00001040
      0000000000000660 0000000000000018      27 43      8
[27] .strtab           STRTAB         0000000000000000 000016a0
      000000000000025b 0000000000000000      0      0      1
[28] .shstrtab         STRTAB         0000000000000000 000018fb
      00000000000000fe 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)

There are no section groups in this file.

重定位信息：包含节重定位的信息，包括位置和偏移量等，当链接器进行链接时，会对这些位置进行修改：

4-0-4重定位

```

重定位节 '.rela.dyn' at offset 0x498 contains 8 entries:
  偏移量      信息      类型      符号值      符号名称 + 加数
000000200d98 000000000008 R_X86_64_RELATIVE 760
000000200da0 000000000008 R_X86_64_RELATIVE 720
000000201008 000000000008 R_X86_64_RELATIVE 201008
000000200fd8 000100000006 R_X86_64_GLOB_DAT 0000000000000000 _ITM_deregisterTMClone + 0
000000200fe0 000400000006 R_X86_64_GLOB_DAT 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0
000000200fe8 000600000006 R_X86_64_GLOB_DAT 0000000000000000 _gmon_start__ + 0
000000200ff0 000800000006 R_X86_64_GLOB_DAT 0000000000000000 _ITM_registerTMCloneTa + 0
000000200ff8 000a00000006 R_X86_64_GLOB_DAT 0000000000000000 __cxa_finalize@GLIBC_2.2.5 + 0

重定位节 '.rela.plt' at offset 0x558 contains 5 entries:
  偏移量      信息      类型      符号值      符号名称 + 加数
000000200fb0 000200000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
000000200fb8 000300000007 R_X86_64_JUMP_SLO 0000000000000000 printf@GLIBC_2.2.5 + 0
000000200fc0 000500000007 R_X86_64_JUMP_SLO 0000000000000000 getchar@GLIBC_2.2.5 + 0
000000200fc8 000700000007 R_X86_64_JUMP_SLO 0000000000000000 exit@GLIBC_2.2.5 + 0
000000200fd0 000900000007 R_X86_64_JUMP_SLO 0000000000000000 sleep@GLIBC_2.2.5 + 0

```

4.4 Hello.o 的结果解析

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

命令如下：

```
myy1170300133@ubuntu-64-20181cs:/mnt/hgfs/hitcs/homework$ objdump -d -r hello.o > hello_objdump.txt
myy1170300133@ubuntu-64-20181cs:/mnt/hgfs/hitcs/homework$ objdump -d -r hello.o > hello.objdump
```

对照分析后发现 hello.o 的反汇编和 hello.s 差异：

如图：

```
0000000000000076a <main>:
76a: 55                push    %rbp
76b: 48 89 e5          mov     %rsp,%rbp
76e: 48 83 ec 20       sub     $0x20,%rsp
772: 89 7d ec          mov     %edi,-0x14(%rbp)
775: 48 89 75 e0       mov     %rsi,-0x20(%rbp)
779: 83 7d ec 03       cmpl    $0x3,-0x14(%rbp)
77d: 74 16            je      795 <main+0x2b>
77f: 48 8d 3d ee 00 00 00 lea     0xee(%rip),%rdi
786: e8 75 fe ff ff    callq   600 <puts@plt>
78b: bf 01 00 00 00    mov     $0x1,%edi
790: e8 9b fe ff ff    callq   630 <exit@plt>
795: c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%rbp)
79c: eb 3b            jmp     7d9 <main+0x6f>
79e: 48 8b 45 e0       mov     -0x20(%rbp),%rax
7a2: 48 83 c0 10       add     $0x10,%rax
7a6: 48 8b 10          mov     (%rax),%rdx
7a9: 48 8b 45 e0       mov     -0x20(%rbp),%rax
7ad: 48 83 c0 08       add     $0x8,%rax
7b1: 48 8b 00          mov     (%rax),%rax
7b4: 48 89 c6          mov     %rax,%rsi
7b7: 48 8d 3d d4 00 00 00 lea     0xd4(%rip),%rdi
7be: b8 00 00 00 00    mov     $0x0,%eax
7c3: e8 48 fe ff ff    callq   610 <printf@plt>
7c8: 8b 05 42 08 20 00 mov     0x200842(%rip),%eax
7ce: 89 c7            mov     %eax,%edi
7d0: e8 6b fe ff ff    callq   640 <sleep@plt>

main:
.LFB5:
.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)
cmpl     $3, -20(%rbp)
je       .L2
leaq     .LC0(%rip), %rdi
call     puts@PLT
movl     $1, %edi
call     exit@PLT
.L2:
movl     $0, -4(%rbp)
jmp      .L3
.L4:
movq     -32(%rbp), %rax
addq     $16, %rax
```

1: 操作数在 hello.s 里面都是十进制，而在 hello.o 反汇编文件里更改为十六进

制。

2: `hello.o` 的每条语句都有地址，全局变量和常量都被安排到了具体的地址里面。

3: 在 `hello.s` 中跳转语句对应的符号和函数调用的函数名在 `hello.o` 的反汇编文件里都变成了相对偏移地址。

4.5 本章小结

本章介绍了 `hello` 程序从 `hello.s` 到 `hello.o` 的汇编过程,通过查看 `hello.s` 和 `hello.o` 的反汇编代码,以及 `hello.o` 的 `elf` 格式文件的内容,了解了汇编语言到机器语言的部分映射和转换关系。

(第 4 章 1 分)

第 5 章 链接

5.1 链接的概念与作用

概念：链接由链接器完成，是将各个可执行文件的代码和数据片段收集起来并组合在一起，合并成一个执行文件的过程，这个文件可以被加载到内存并执行。

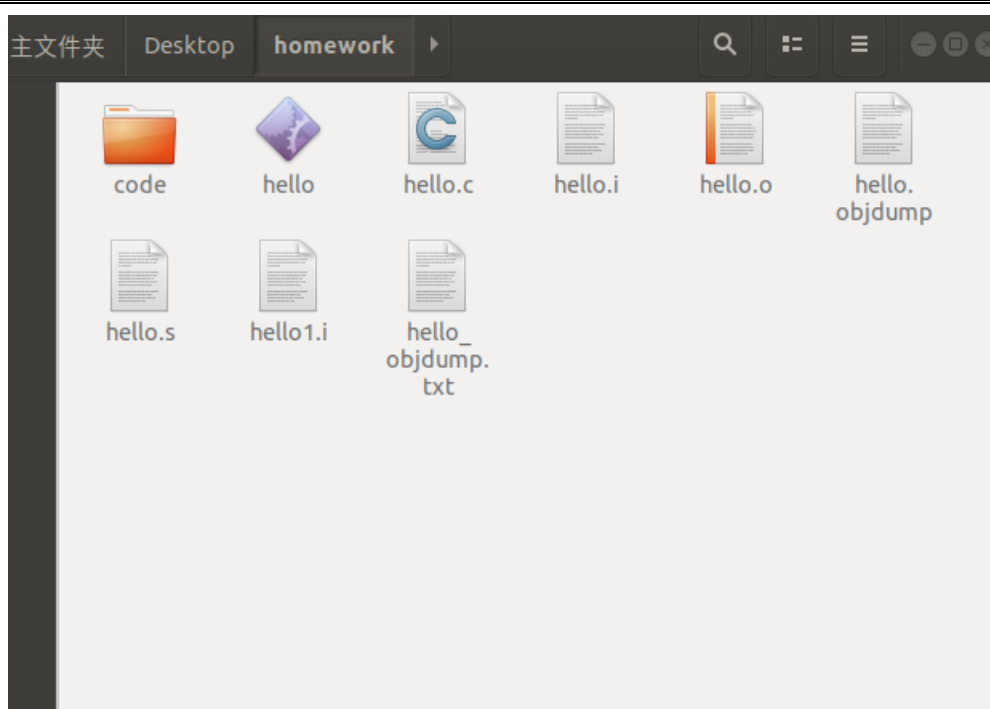
作用：链接是将文件的各种代码和数据片段收集并组合成为一个单一文件的过程，这个文件可被加载到内存并执行。链接可以执行于编译时(compile time)，也就是在源代码被翻译成机器代码时；也可以执行于加载时 Cload time)，也就是在程序被加载器(loader)加载到内存并执行时；甚至执行于运行时(run time)，也就是由应用程序来执行，这样使分离编译成为可能，大大简化了文件的编译执行。

5.2 在 Ubuntu 下链接的命令

Ld 命令行指令：`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`

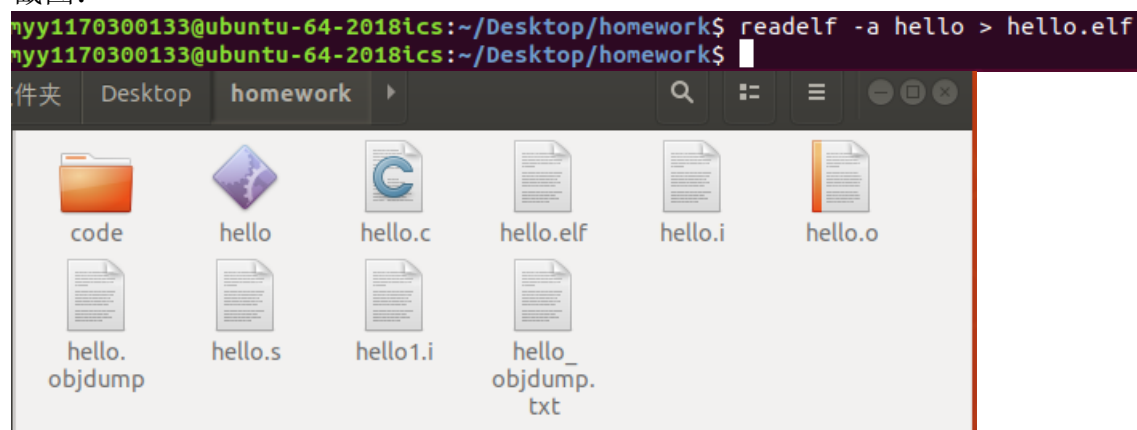
如图：生成 hello 文件

```
myy1170300133@ubuntu-64-2018ics:~/Desktop/homework$ 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
myy1170300133@ubuntu-64-2018ics:~/Desktop/homework$
```



5.3 可执行目标文件 hello 的格式

使用 `readelf -a hello > hello.elf` 命令，生成 hello 的 ELF 格式文件
截图：



在 sublime 编辑器中打开 `hello.elf` 文件，可以看到 ELF 格式文件和前文中的 elf 相同，文件中，节头对 `hello` 中所有的节信息进行了声明，包括名称，大小以及在程序中的偏移量和对齐。根据节头中的信息我们就可以用 HexEdit 找到各个节所占的区间。

22	节头：						
23	[号]	名称	类型	地址	偏移量		
24		大小	全体大小	旗标	链接	信息	对齐
25	[0]		NULL	0000000000000000		00000000	
26		0000000000000000	0000000000000000		0	0	0
27	[1]	.interp	PROGBITS	000000000400200		00000200	
28		000000000000001c	0000000000000000	A	0	0	1
29	[2]	.note.ABI-tag	NOTE	00000000040021c		0000021c	
30		0000000000000020	0000000000000000	A	0	0	4
31	[3]	.hash	HASH	000000000400240		00000240	
32		0000000000000034	0000000000000004	A	5	0	8
33	[4]	.gnu.hash	GNU_HASH	000000000400278		00000278	
34		000000000000001c	0000000000000000	A	5	0	8
35	[5]	.dynsym	DYNSYM	000000000400298		00000298	
36		00000000000000c0	0000000000000018	A	6	1	8
37	[6]	.dynstr	STRTAB	000000000400358		00000358	
38		0000000000000057	0000000000000000	A	0	0	1
39	[7]	.gnu.version	VERSYM	0000000004003b0		000003b0	
40		0000000000000010	0000000000000002	A	5	0	2
41	[8]	.gnu.version_r	VERNEED	0000000004003c0		000003c0	
42		0000000000000020	0000000000000000	A	6	1	8
43	[9]	.rela.dyn	RELA	0000000004003e0		000003e0	
44		0000000000000030	0000000000000018	A	5	0	8
45	[10]	.rela.plt	RELA	000000000400410		00000410	
46		0000000000000078	0000000000000018	AI	5	19	8
47	[11]	.init	PROGBITS	000000000400488		00000488	
48		0000000000000017	0000000000000000	AX	0	0	4
49	[12]	.plt	PROGBITS	0000000004004a0		000004a0	
50		0000000000000060	0000000000000010	AX	0	0	16
51	[13]	.text	PROGBITS	000000000400500		00000500	
52		00000000000000132	0000000000000000	AX	0	0	16
53	[14]	.fini	PROGBITS	000000000400634		00000634	
54		0000000000000009	0000000000000000	AX	0	0	4
55	[15]	.rodata	PROGBITS	000000000400640		00000640	
56		000000000000002f	0000000000000000	A	0	0	4
57	[16]	.eh_frame	PROGBITS	000000000400670		00000670	
58		00000000000000fc	0000000000000000	A	0	0	8
59	[17]	.dynamic	DYNAMIC	000000000600e50		00000e50	
60		000000000000001a0	0000000000000010	WA	6	0	8
61	[18]	.got	PROGBITS	000000000600ff0		00000ff0	
62		0000000000000010	0000000000000008	WA	0	0	8
63	[19]	.got.plt	PROGBITS	000000000601000		00001000	
64		0000000000000040	0000000000000008	WA	0	0	8
65	[20]	.data	PROGBITS	000000000601040		00001040	
66		0000000000000008	0000000000000000	WA	0	0	4
67	[21]	.comment	PROGBITS	0000000000000000		00001048	
68		000000000000002a	0000000000000001	MS	0	0	1
69	[22]	.symtab	SYMTAB	0000000000000000		00001078	
70		00000000000000498	0000000000000018		23	28	8
71	[23]	.strtab	STRTAB	0000000000000000		00001510	
72		00000000000000150	0000000000000000		0	0	1
73	[24]	.shstrtab	STRTAB	0000000000000000		00001660	
74		00000000000000c5	0000000000000000		0	0	1

5.4 hello 的虚拟地址空间

使用 edb 加载 hello，截图：

.plt 位于代码段的 plt 表

00000000:004004d0	ff 35 32 0b 20 00	pushq 0x200b32(%rip)
00000000:004004d6	ff 25 34 0b 20 00	jmpq *0x200b34(%rip)
00000000:004004dc	0f 1f 40 00	nopl (%rax)
00000000:004004e0	ff 25 32 0b 20 00	jmpq *0x200b32(%rip)
00000000:004004e6	68 00 00 00 00	pushq \$0
00000000:004004eb	e9 e0 ff ff ff	jmp hello!.plt
00000000:004004f0	ff 25 2a 0b 20 00	jmpq *0x200b2a(%rip)
00000000:004004f6	68 01 00 00 00	pushq \$1
00000000:004004fb	e9 d0 ff ff ff	jmp hello!.plt
00000000:00400500	ff 25 22 0b 20 00	jmpq *0x200b22(%rip)
00000000:00400506	68 02 00 00 00	pushq \$2
00000000:0040050b	e9 c0 ff ff ff	jmp hello!.plt
00000000:00400510	ff 25 1a 0b 20 00	jmpq *0x200b1a(%rip)
00000000:00400516	68 03 00 00 00	pushq \$3
00000000:0040051b	e9 b0 ff ff ff	jmp hello!.plt
00000000:00400520	ff 25 12 0b 20 00	jmpq *0x200b12(%rip)
00000000:00400526	68 04 00 00 00	pushq \$4
00000000:0040052b	e9 a0 ff ff ff	jmp hello!.plt
00000000:00400530	ff 25 0a 0b 20 00	jmpq *0x200b0a(%rip)
00000000:00400536	68 05 00 00 00	pushq \$5
00000000:0040053b	e9 90 ff ff ff	jmp hello!.plt
00000000:00400540	ff 25 b2 0a 20 00	jmpq *0x200ab2(%rip)
00000000:00400546	66 90	nop

.text 代码段 如图为 hello!_start 函数

00000000:00400550	31 ed	xorl %ebp, %ebp
00000000:00400552	49 89 d1	movq %rdx, %r9
00000000:00400555	5e	popq %rsi
00000000:00400556	48 89 e2	movq %rsp, %rdx
00000000:00400559	48 83 e4 f0	andq \$0xfffffffff0, %rsp
00000000:0040055d	50	pushq %rax
00000000:0040055e	54	pushq %rsp
00000000:0040055f	49 c7 c0 40 07 40 00	movq \$0x400740, %r8
00000000:00400566	48 c7 c1 d0 06 40 00	movq \$0x4006d0, %rcx
00000000:0040056d	48 c7 c7 46 06 40 00	movq \$0x400646, %rdi
00000000:00400574	e8 87 ff ff ff	callq hello!_libc_start_mai..
00000000:00400579	f4	hlt

.rodata 只读数据段

00000000:00400750	01 00	addl %eax, (%rax)
00000000:00400752	02 00	addb (%rax), %al
00000000:00400754	55	pushq %rbp
00000000:00400755	73 61	jae hello!.eh_frame

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 和 hello.o 的反汇编文件进行比较：

hello 相对于 hello.o 不同：

- 1: hello.o 中的相对偏移地址到了 hello 中变成了虚拟内存地址

2: hello 中相对 hello.o 增加了许多的外部链接来的函数和很多类似于 .init, .plt 的节。

3: hello.o 中跳转以及函数调用的地址在 hello 中都被更换成了虚拟内存地址。

如图

```

000000000400532 <main>:
400532: 55                push    %rbp
400533: 48 89 e5          mov     %rsp,%rbp
400536: 48 83 ec 20       sub     $0x20,%rsp
40053a: 89 7d ec          mov     %edi,-0x14(%rbp)
40053d: 48 89 75 e0       mov     %rsi,-0x20(%rbp)
400541: 83 7d ec 03       cmpl    $0x3,-0x14(%rbp)
400545: 74 16            je      40055d <main+0x2b>
400547: 48 8d 3d f6 00 00 00 lea     0xf6(%rip),%rdi
40054e: e8 5d ff ff ff   callq   4004b0 <puts@plt>
400553: bf 01 00 00 00   mov     $0x1,%edi
400558: e8 83 ff ff ff   callq   4004e0 <exit@plt>
40055d: c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%rbp)
400564: eb 3b            jmp     4005a1 <main+0x6f>
400566: 48 8b 45 e0       mov     -0x20(%rbp),%rax
40056a: 48 83 c0 10       add     $0x10,%rax
40056e: 48 8b 10         mov     (%rax),%rdx
400571: 48 8b 45 e0       mov     -0x20(%rbp),%rax
400575: 48 83 c0 08       add     $0x8,%rax
40057a: 48 8b 00         mov     (%rax),%rax
000000000000076a <main>:
76a: 55                push    %rbp
76b: 48 89 e5          mov     %rsp,%rbp
76e: 48 83 ec 20       sub     $0x20,%rsp
772: 89 7d ec          mov     %edi,-0x14(%rbp)
775: 48 89 75 e0       mov     %rsi,-0x20(%rbp)
779: 83 7d ec 03       cmpl    $0x3,-0x14(%rbp)
77d: 74 16            je      795 <main+0x2b>
77f: 48 8d 3d ee 00 00 00 lea     0xee(%rip),%rdi
786: e8 75 fe ff ff   callq   600 <puts@plt>
78b: bf 01 00 00 00   mov     $0x1,%edi
790: e8 9b fe ff ff   callq   630 <exit@plt>
795: c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%rbp)
79c: eb 3b            jmp     7d9 <main+0x6f>
79e: 48 8b 45 e0       mov     -0x20(%rbp),%rax
7a2: 48 83 c0 10       add     $0x10,%rax
7a6: 48 8b 10         mov     (%rax),%rdx
7a9: 48 8b 45 e0       mov     -0x20(%rbp),%rax
7ad: 48 83 c0 08       add     $0x8,%rax
7b1: 48 8b 00         mov     (%rax),%rax
7b4: 48 89 c6         mov     %rax,%rsi
7b7: 48 8d 3d d4 00 00 00 lea     0xd4(%rip),%rdi
7be: b8 00 00 00 00   mov     $0x0,%eax

```

重定位：链接器在完成符号解析以后，就把代码中的每个符号引用和正好一个符号定义（即它的一个输入目标模块中的一个符号表条目）关联起来。此时，链接器就知道它的输入目标模块中的代码节和数据节的确切大小。然后就可以开始重定位步骤了，在这个步骤中，将合并输入模块，并为每个符号分配运行时的地址。

在 `hello` 到 `hello.o` 中，首先是重定位节和符号定义，链接器将所有输入到 `hello` 中相同类型的节合并为同一类型的新的聚合节。例如，来自所有的输入模块的 `.data` 节被全部合并成一个节，这个节成为 `hello` 的 `.data` 节。然后，链接器将运行时内存地址赋给新的聚合节，赋给输入模块定义的每个节，以及赋给输入模块定义的每一个符号。当这一步完成时，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。然后是重定位节中的符号引用，链接器会修改 `hello` 中的代码节和数据节中对每一个符号的引用，使得他们指向正确的运行地址。

5.6 `hello` 的执行流程

<code>_dl_start</code>
<code>_dl_init</code>
<code>_start</code>
<code>_libc_start_main</code>
<code>_init</code>
<code>_main</code>
<code>_printf</code>
<code>_exit</code>
<code>_sleep</code>
<code>_getchar</code>
<code>_dl_runtime_resolve_xsave</code>
<code>_dl_fixup</code>
<code>_dl_lookup_symbol_x</code>

5.7 `Hello` 的动态链接分析

使用 `readelf -a hello` 找出 `GLOBAL_OFFSET_TABLE` 的地址，为 `00601000`，在

```
0000000000000000: 0000000000000000 0000000000000000 0000000000000000 0000000000000000
[19] .got.plt          PROGBITS          000000000000601000 00001000
```

使用 `edb` 调试：

`dl_init` 前：

00000000:00600fe0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00600ff0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00601000	10 0e 60 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00601010	00 00 00 00 00 00 00 00 b6 04 40 00 00 00 00 00@
00000000:00601020	c6 04 40 00 00 00 00 00 d6 04 40 00 00 00 00 00@
00000000:00601030	e6 04 40 00 00 00 00 00 f6 04 40 00 00 00 00 00@
00000000:00601040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00601050	02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

00000000:00600fe0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00600ff0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00601000	10 0e 60 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00601010	00 00 00 00 00 00 00 00 b6 04 40 00 00 00 00 00@
00000000:00601020	c6 04 40 00 00 00 00 00 d6 04 40 00 00 00 00 00@
00000000:00601030	e6 04 40 00 00 00 00 00 f6 04 40 00 00 00 00 00@
00000000:00601040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00601050	02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

dl_init 后

00000000:00600fe0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00600ff0	b0 0a bb fe 7a 7f 00 00 00 00 00 00 00 00 00 00z
00000000:00601000	10 0e 60 00 00 00 00 00 70 91 1a ff 7a 7f 00 00p..z
00000000:00601010	80 76 f9 fe 7a 7f 00 00 b6 04 40 00 00 00 00 00@
00000000:00601020	c6 04 40 00 00 00 00 00 d6 04 40 00 00 00 00 00@
00000000:00601030	e6 04 40 00 00 00 00 00 f6 04 40 00 00 00 00 00@
00000000:00601040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00601050	02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

00000000:00600fe0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00600ff0	b0 0a bb fe 7a 7f 00 00 00 00 00 00 00 00 00 00z
00000000:00601000	10 0e 60 00 00 00 00 00 70 91 1a ff 7a 7f 00 00p..z
00000000:00601010	80 76 f9 fe 7a 7f 00 00 b6 04 40 00 00 00 00 00@
00000000:00601020	c6 04 40 00 00 00 00 00 d6 04 40 00 00 00 00 00@
00000000:00601030	e6 04 40 00 00 00 00 00 f6 04 40 00 00 00 00 00@
00000000:00601040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00601050	02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

在 dl_init 前后 00000000 00601010 处发生变化

5.8 本章小结

本章主要讲述了链接的概念和作用，hello 的 elf 格式文件和 hello 的虚拟地址空间和重定位过程，执行流程和动态链接过程。

(第5章1分)

第 6 章 hello 进程管理

6.1 进程的概念与作用

概念：进程是一个执行中程序的实例

作用：每次用户通过向 shell 输入一个可执行目标文件的名字，运行程序时，shell 就会创建一个新的进程，然后在这个新进程的上下文中运行这个可执行目标文件。应用程序也能够创建新进程，并且在这个新进程的上下文中运行它们自己的代码或其他应用程序。

6.2 简述壳 Shell-bash 的作用与处理流程

Shell-bash 是一个交互型应用程序，可以代替用户执行相应的程序或可执行文件

执行流程：

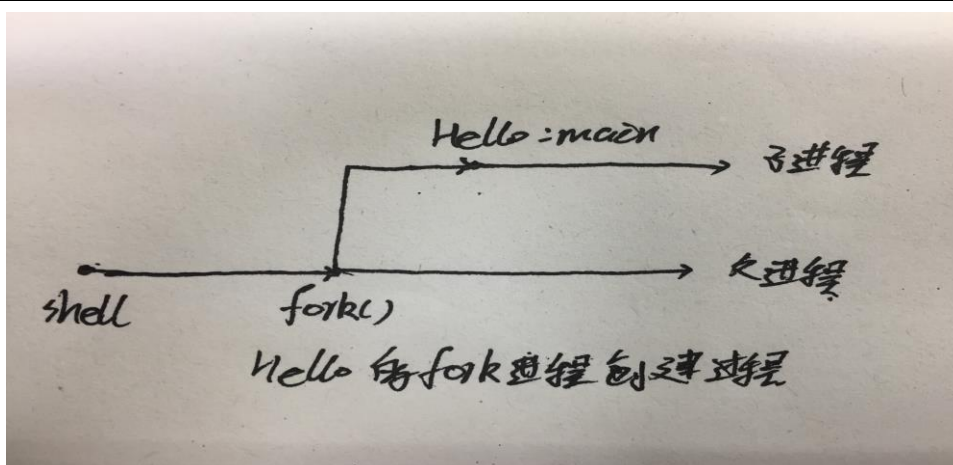
- 1：从命令行终端中读取输入的命令。
- 2：将输入字符串切分获得相应所有的参数，并处理非法输入
- 3：如果是内置命令则立即执行
- 4：否则调用相应的程序为其分配子进程运行
- 5：运行同时 shell-bash 也应该接受键盘输入转化为相应信号，并对这些信号进行相应处理

6.3 Hello 的 fork 进程创建过程

在终端中输入 “./hello 1170300133 孟月阳” 后，终端会对输入的命令行进行分割等解析，查询到 hello 不是内置命令，所以解析之后 shell 判断 ./hello 的语义为执行当前目录下的可执行目标文件 hello，之后调用 fork 函数为其创建一个新的子进程，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一份副本（虚拟内存中表示为写时复制）并共享文件，他们最容易利用的区别是拥有不同的 PID。

父子进程是并发运行的独立进程，内核能够以任意方式交替执行它们的逻辑控制流的指令。在子进程执行时，父进程默认选项是显示等待子进程的完成。

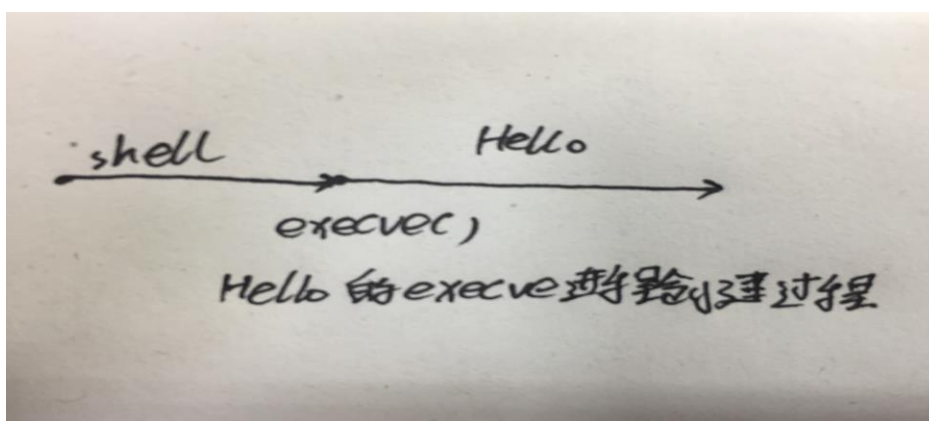
fork 函数示意图：



6.4 Hello 的 execve 过程

当 fork 执行过之后，子进程调用 execve 函数在当前进程的上下文中加载并运行 hello 程序，删除子进程现有的虚拟内存段，并创建一组新的代码、数据、堆和栈段。execve 函数只调用不返回。

示意图：



6.5 Hello 的进程执行

相关概念：

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。

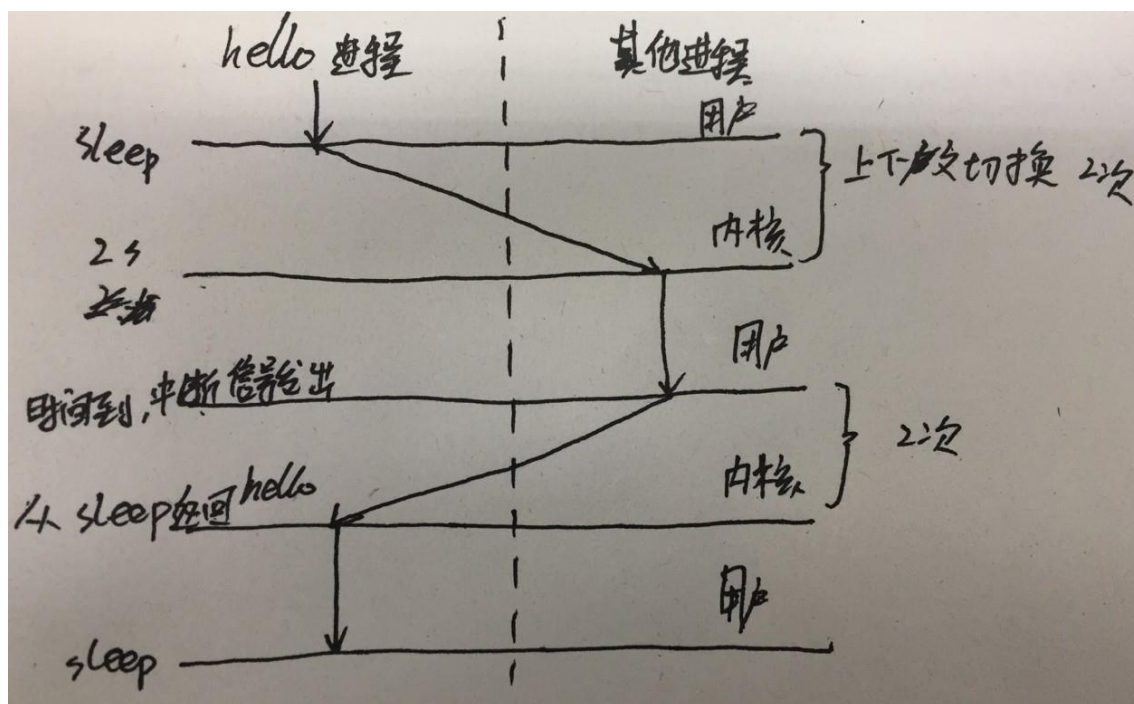
用户模式和内核模式：处理器通常使用一个寄存器提供两种模式的区分，该寄存器描述了进程当前享有的特权，当没有设置模式位时，进程就处于用户模式中，用户模式的进程不允许执行特权指令，也不允许直接引用地址空间中内核

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。

hello sleep 进程调度的过程：当调用 sleep 之前，如果 hello 程序不被切换进程则直接顺序执行 sleep，如果发生 sleep 外的进程切换，则进行上下文切换，上下文切换是由内核中调度器完成的，当内核调度新的进程运行后，它就会抢占当前进程，并保存当前进程的上下文；恢复新恢复进程被保存的上下文；将控制传递给新恢复的进程。上下文切换发生在用户模式和内核模式的切换中

一次 sleep 的运行示意图：



6.6 hello 的异常与信号处理

(以下格式自行编排，编辑时删除)

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截图，说明异常与信号的处理。

1: 正常运行: hello 运行完后, 被父进程回收

```

myy1170300133@ubuntu-64-2018ics:~/Desktop/homework$ ./hello 1170300133 孟月阳
Hello 1170300133 孟月阳
Hello 1170300133 孟月阳
Hello 1170300133 孟月阳
Hello 1170300133 孟月阳
Hello 1170300133 孟月阳
Hello 1170300133 孟月阳
Hello 1170300133 孟月阳
Hello 1170300133 孟月阳
Hello 1170300133 孟月阳
hit
myy1170300133@ubuntu-64-2018ics:~/Desktop/homework$

```

2: 中途按下 ctrl+z: 在打印 3 次后按下 ctrl+z, shell 收到 SIGSTP 信号, 处理函数是将 hello 进程挂起, 并打印屏幕, 显示出 hello 后台作业号是 1。使用 ps 命令可以看到 hello 进程还在, 没有被回收。使用 fg 1 命令将 hello 调回前台执行, hello 继续打印命令行, 打印剩下的七次后, 键入字符结束程序, 同时进程被回收

```

myy1170300133@ubuntu-64-2018ics:~/Desktop/homework$ ./hello 1170300133 孟月阳
Hello 1170300133 孟月阳
Hello 1170300133 孟月阳
Hello 1170300133 孟月阳
^Z
[1]+  已停止                  ./hello 1170300133 孟月阳
myy1170300133@ubuntu-64-2018ics:~/Desktop/homework$ jobs
[1]+  已停止                  ./hello 1170300133 孟月阳
myy1170300133@ubuntu-64-2018ics:~/Desktop/homework$ ps
  PID TTY          TIME CMD
 93035 pts/0        00:00:00 bash
 93644 pts/0        00:00:00 hello
 93645 pts/0        00:00:00 ps
myy1170300133@ubuntu-64-2018ics:~/Desktop/homework$ fg 1
./hello 1170300133 孟月阳
Hello 1170300133 孟月阳
Hello 1170300133 孟月阳
Hello 1170300133 孟月阳
Hello 1170300133 孟月阳
Hello 1170300133 孟月阳
Hello 1170300133 孟月阳
Hello 1170300133 孟月阳
hit
myy1170300133@ubuntu-64-2018ics:~/Desktop/homework$

```

3: 中途按下 ctrl+c: 在打印 3 次后按下 ctrl+c, shell 收到 SIGINT 信号, 处理函数结束 hello 并回收相应进程

```

myy1170300133@ubuntu-64-2018ics:~$ cd Desktop/homework
myy1170300133@ubuntu-64-2018ics:~/Desktop/homework$ ls
code  hello1.i  hello.elf  hello.o      hello_objdump.txt
hello hello.c  hello.i    hello.objdump hello.s
myy1170300133@ubuntu-64-2018ics:~/Desktop/homework$ ./hello 1170300133 孟月阳
Hello 1170300133 孟月阳
Hello 1170300133 孟月阳
Hello 1170300133 孟月阳
^C
myy1170300133@ubuntu-64-2018ics:~/Desktop/homework$

```

4: 中途乱按: 键入内容保存在缓冲区, 程序结束后作为 shell 输入指令

```
myy1170300133@ubuntu-64-20181cs:~/Desktop/homework$ ./hello 1170300133 孟月阳
Hello 1170300133 孟月阳
hisif
Hello 1170300133 孟月阳
shifaf
sjkdjkas
cHello 1170300133 孟月阳
sjjher
134Hello 1170300133 孟月阳

sds
Hello 1170300133 孟月阳
Hello 1170300133 孟月阳
hfahkfhasjkhfHello 1170300133 孟月阳
kwhauh
Hello 1170300133 孟月阳
fhsahfl
sHello 1170300133 孟月阳
jdkj
Hello 1170300133 孟月阳
sdjkl
myy1170300133@ubuntu-64-20181cs:~/Desktop/homework$ shifaf
shifaf: 未找到命令
myy1170300133@ubuntu-64-20181cs:~/Desktop/homework$ sjkdjkas
sjkdjkas: 未找到命令
myy1170300133@ubuntu-64-20181cs:~/Desktop/homework$ csjjher
csjjher: 未找到命令
myy1170300133@ubuntu-64-20181cs:~/Desktop/homework$ 134
134: 未找到命令
myy1170300133@ubuntu-64-20181cs:~/Desktop/homework$ sds
Command 'sds' not found, but can be installed with:

sudo apt install simh

myy1170300133@ubuntu-64-20181cs:~/Desktop/homework$ hfahkfhasjkhfkwhauh
hfahkfhasjkhfkwhauh: 未找到命令
myy1170300133@ubuntu-64-20181cs:~/Desktop/homework$ fhsahfl
```

6.7 本章小结

异常控制流发生在计算机系统的各个层次。比如，在硬件层，硬件检测到的事件会触发控制突然转移到异常处理程序。在操作系统层，内核通过上下文切换将控制从一个用户进程转移到另一个用户进程。在应用层，一个进程可以发送信号到另一个进程，而接收者会将控制突然转移到它的一个信号处理程序。一个程序可以通过回避通常的栈规则，并执行到其他函数中任意位置的非本地跳转来对错误做出反应。

异常控制流在计算机系统中至关重要，同时进程也是计算机中一个很重要的实现概念

(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：程序代码经过编译后出现在汇编程序中地址。逻辑地址一般由选择符和偏移量组成。

线性地址：1：逻辑地址经过段机制后转化为线性地址，为描述符:偏移量的组合形式。分页机制中线性地址作为输入。2：：地址空间是一个非负整数地址的有序集合，如果地址空间中的整数是连续的，那么我们说它是一个线性地址空间，线性地址空间中的地址是线性地址。

虚拟地址：CPU 运行中生成的一个虚拟地址，用于转换为物理地址来简化寻址。

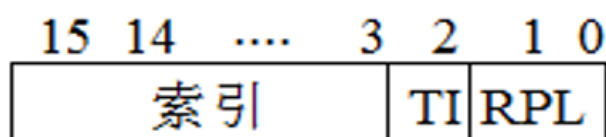
物理地址：CPU 通过地址总线的寻址，找到真实的物理内存的对应地址。CPU 对内存的访问是通过利用北桥芯片组的前端总线来完成的。在前端总线上传输的内存地址都是物理内存地址。

7.2 Intel 逻辑地址到线性地址的变换-段式管理

（以下格式自行编排，编辑时删除）

段式管理是从逻辑地址到线性地址（虚拟地址）

一个逻辑地址由两部份组成，段标识符和段内偏移量。段标识符是由一个 16 位长的字段组成，称为段选择符。其中前 13 位是索引。后面 3 位是描述符表类型和权限。



段描述符是一种数据结构，实际上就是段表项，分两类：

- 1：用户的代码段和数据段描述符
- 2：系统控制段描述符，又分两种：

特殊系统控制段描述符，包括：局部描述符表（LDT）描述符和任务状态段（TSS）描述符

控制转移类描述符，包括：调用门描述符、任务门描述符、中断门描述符

和陷阱门描述符

描述符表实际上就是段表，由段描述符(段表项)组成。有三种类型

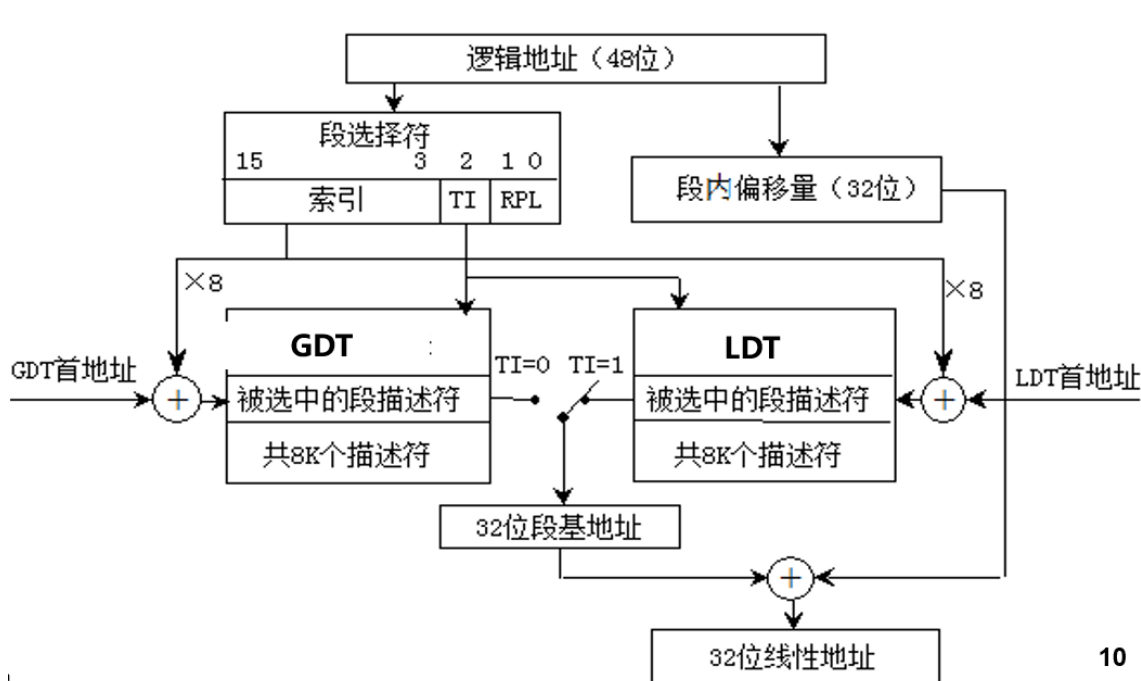
1: 全局描述符表 **GDT**: 只有一个，用来存放系统内每个任务都可能访问的描述符，例如，内核代码段、内核数据段、用户代码段、用户数据段以及 **TSS**（任务状态段）等都属于 **GDT** 中描述的段

2: 局部描述符表 **LDT**: 存放某任务（即用户进程）专用的描述符

3: 中断描述符表 **IDT**: 包含 256 个中断门、陷阱门和任务门描述符

逻辑地址向线性地址转换:

如图，被选中的段描述符先被送至描述符 cache，每次从描述符 cache 中取 32 位段基址，与 32 位段内偏移量（有效地址）相加得到线性地址:

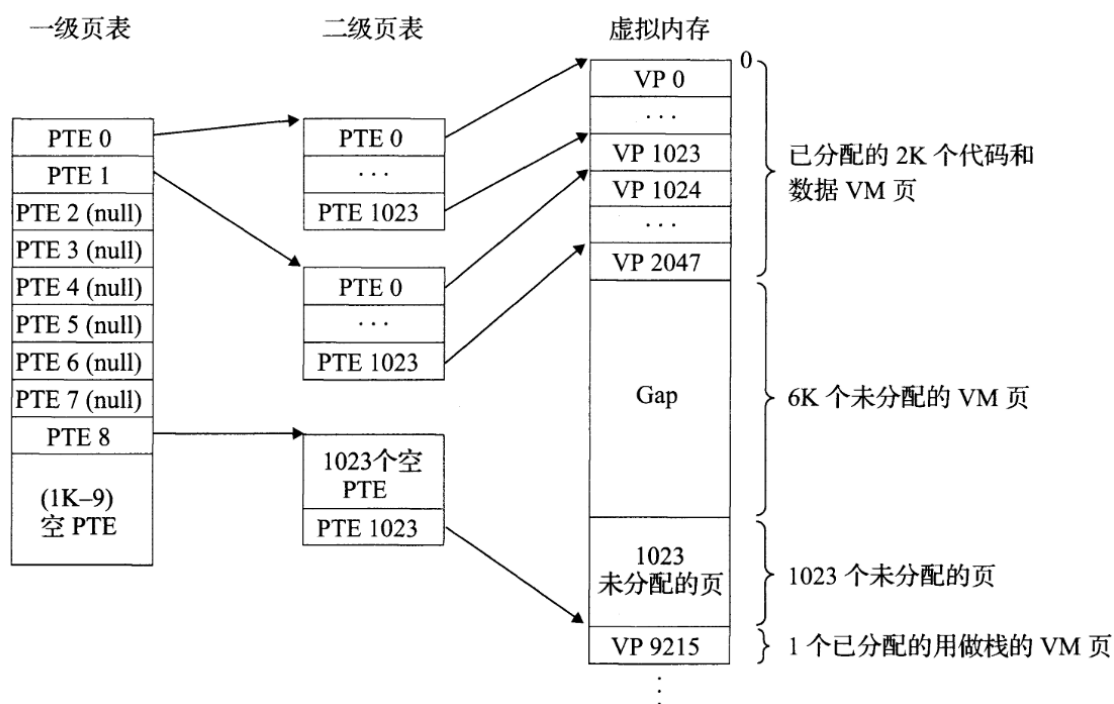


7.3 Hello 的线性地址到物理地址的变换-页式管理

页式管理是从虚拟地址到物理地址

线性地址被分为以固定长度为单位的组，称为页，例如一个 32 位的机器，线性地址最大可为 4G，可以用 4KB 为一个页来划分，这样，整个线性地址就被划分为一个被称为页目录的数组，共有 2^{20} 个次方个页。目录中的每一个目录项，就是一个地址——对应的页的地址。另一类“页”，也就是物理地址的页，我们称之为物理页。是分页单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。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。如图：使用 1K 个 PTE 的一级页表，每一个页面对应一个二级页表，再使用二级页表对内存进行划分，将虚拟内存分割为若干个 4K 大小的片，实现内存的划分。



进行访问时，虚拟地址先传给 MMU，MMU 使用虚拟地址中的部分作为 TLBT 和 TLBI 向一级页表进行匹配：找到对应的 PTE，作为二级页表的起始位置，再在二级页表中进行匹配，寻找对应的 PTE，作为虚拟内存中的起始位置，最后使用偏移找到相应的内存地址

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

下面使用教材上 Intel Core i7/Linux 内存系统来说明 VA 到 PA 的变换：

前提：虚拟地址空间 48 位，物理地址空间 52 位，页表大小 4Kb，4 级页表层次结构，TLB 为 4 路 16 组相联。即：一个页表大小为 4Kb，一个 PTE 条目为 8b，一共 512 个条目，使用 9 位二进制索引，一共 4 个页表共使用 36 位二进制索引，（VPN 为 36 位，VPO 为 12 位），TLB 共 16 组，TLBI 需要 4 位，所以 VPN35 位，TLBT32 位。

教材截图如下，CPU 生成一个虚拟地址（VA）之后，VA 传给 MMU，MMU 使用前 36 位 VPN 作为 TLBT 和 TLBI 向 TLB 进行匹配：

如果命中，得到 40 位 PPN，与 VPO（PPO）进行组合为 52 为 PA。

如果不命中，MMU 向页表中进行查询，CR3（指向一级页表的起始位置）

和 VPN1 偏移量找到相应 PTE，如果命中且符合权限，则确定第二级页表的起始地址，以此类推，在四级页表中得到相应 PPN，和 PPO 组合成 PA；如果查询 PTE 时发现不在物理内存中，触发缺页故障，进入缺页处理程序。

7-1 Intel Core i7/Linux 内存系统

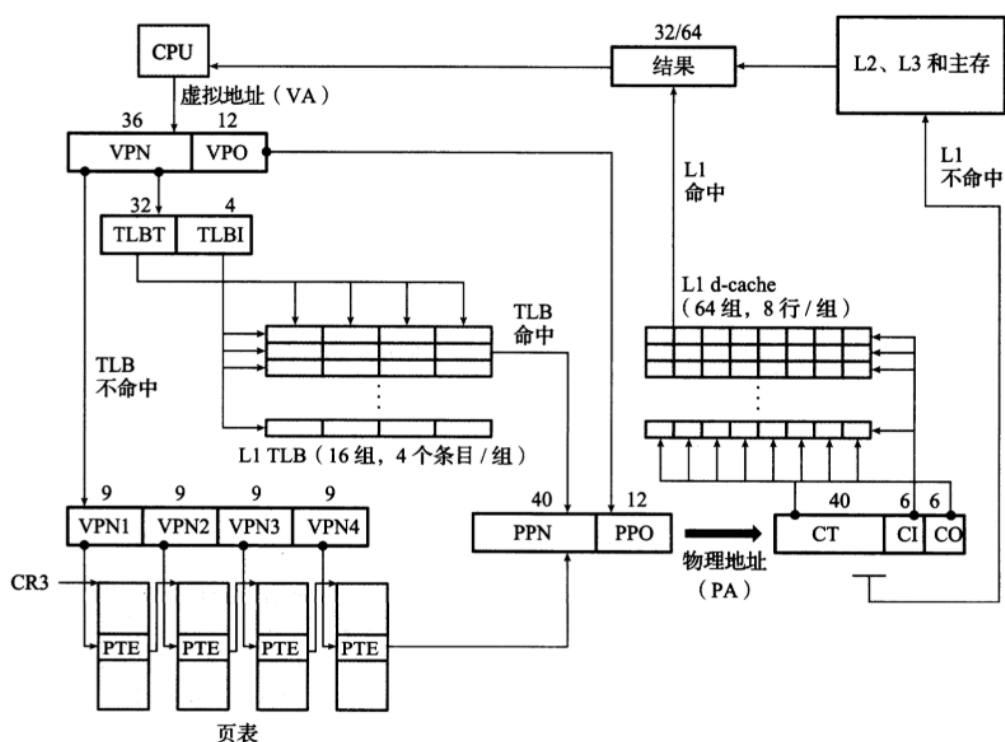


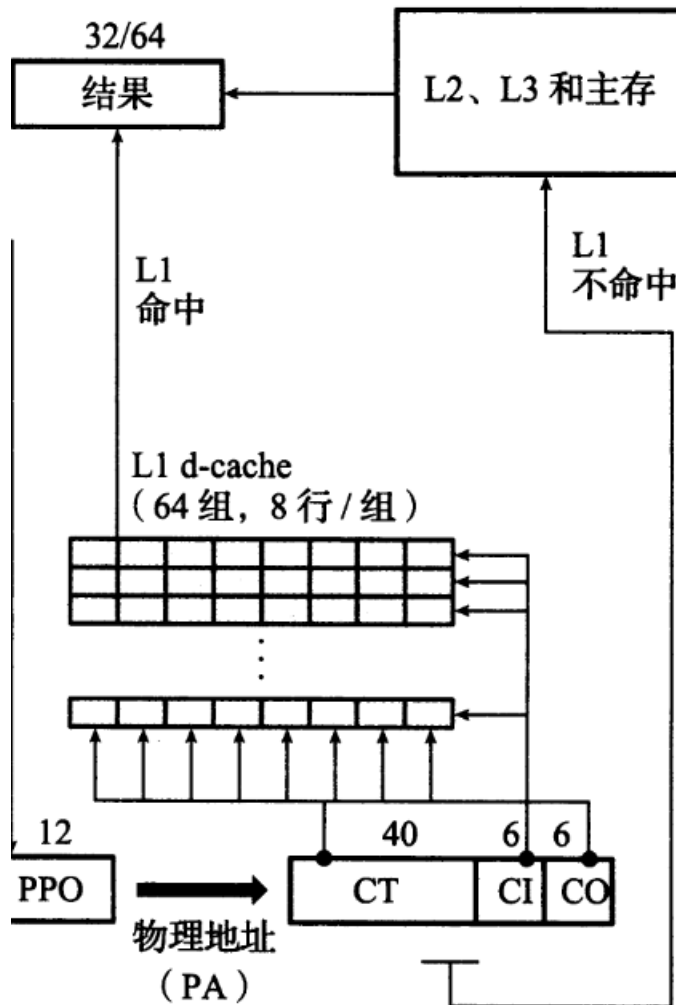
图 9-22 Core i7 地址翻译的概况。为了简化，没有显示 i-cache、i-TLB 和 L2 统一 TLB

7.5 三级 Cache 支持下的物理内存访问

(以下格式自行编排，编辑时删除)

使用教材上 Intel Core i7/Linux 的 L1 一级 Cache: 8 路 64 组相联 Cache，块大小为 64b，进行讲解。L2，L3 二级三级 Cache 实现类似：

Cache 介绍：一共 64 组，组索引需要 6 位二进制位，块大小为 64b，需要 6 位二进制位作为块偏移，剩下的 40 位作为标记位：



在上一步中我们获得了物理地址 PA，使用 CI，PPO 的后 6 位寻找相应的组，如果匹配成功，CT 标识位相同且使用 CO 块偏移找到相应的块，块的有效位为 1，则命中，将内容取出返回；反之不命中，在下一级存储中寻找

7.6 hello 进程 fork 时的内存映射

Hello 进程使用 fork 创建子进程时，内存系统会分配给子进程一个唯一的 PID 并创建其父进程 mm_struct，区域结构和页表的副本，此时父子进程共享同样的页面，系统将两个进程的页面全部标记为只读，将两个进程的区域结构改为写时复制。

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进程。

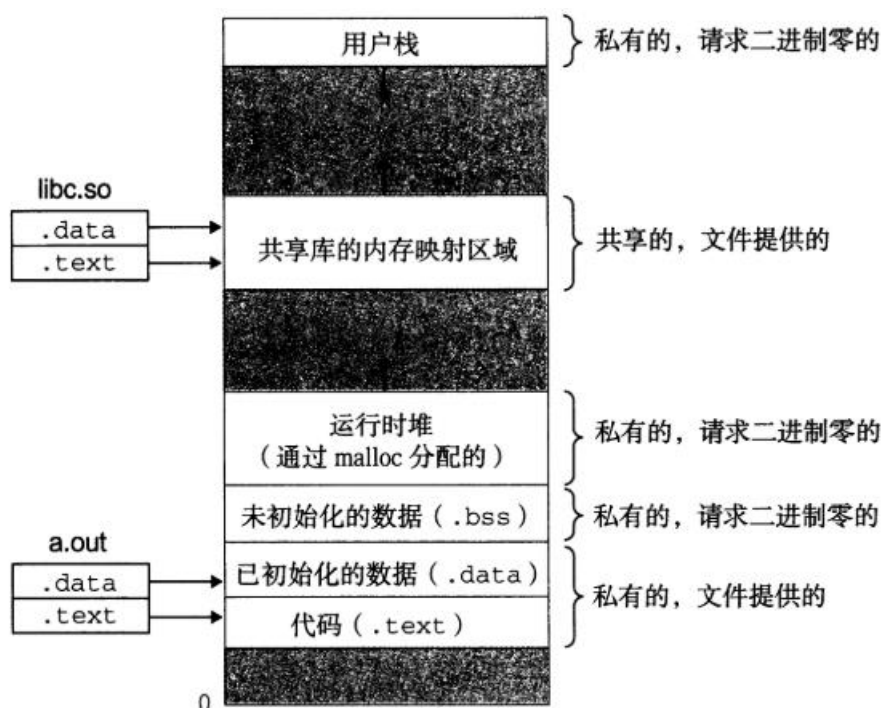
`Execve` 加载并运行 `hello` 的步骤：

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`（图中的 `a.out`）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`（图中的 `a.out`）中。栈和堆区域也是请求二进制零的，初始长度为零。下图中包含了私有区域的不同映射。

3：映射共享区域。`libc.so` 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。

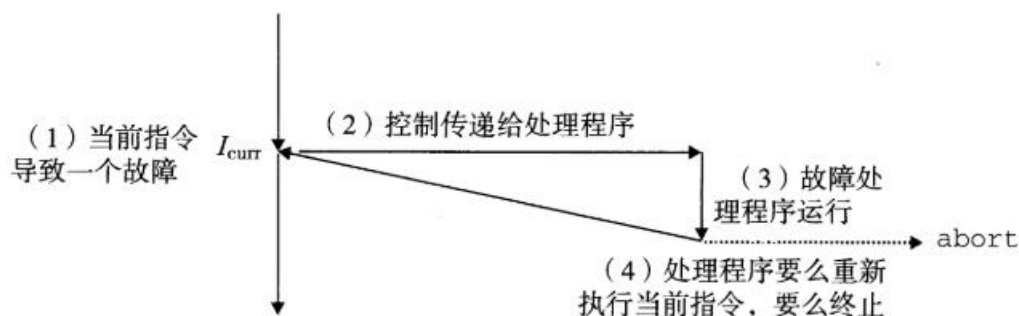
4：设置程序计数器（PC）。`execve` 做的最后一件事情就是设置当前进程上下文中的程序计数器，使之指向代码区域的入口点。



7.8 缺页故障与缺页中断处理

缺页故障是常见的故障之一，即：DRAM 缓存不命中时发生，标志寻找内容不在内存中，需要从硬盘中取。缺页故障遵循下面的故障处理逻辑：

7-2缺页故障处理



缺页中断处理：这个是内核代码，从以缓存的页表中选取一个牺牲页，如果改牺牲页更改过，那么就将它交换出去，换入新的页面并更新相应页表。之后 CPU 重新执行引发却也的指令，由原来的虚拟地址重新寻址。

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(heap)。系统之间细节不同，但是不失通用性，假设堆是一个请求二进制零的区域，它紧接在未初始化的数据区域后开始，并向上生长(向更高的地址)。

对于每个进程，内核维护着一个变量 `brk`(读做“break”),它指向堆的顶部。分配器将堆视为一组不同大小的块(block)的集合来维护。每个块就是一个连续的虚拟内存片(chunk),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。

根据分配释放的方式，分配器分为显式隐式分配器两种：

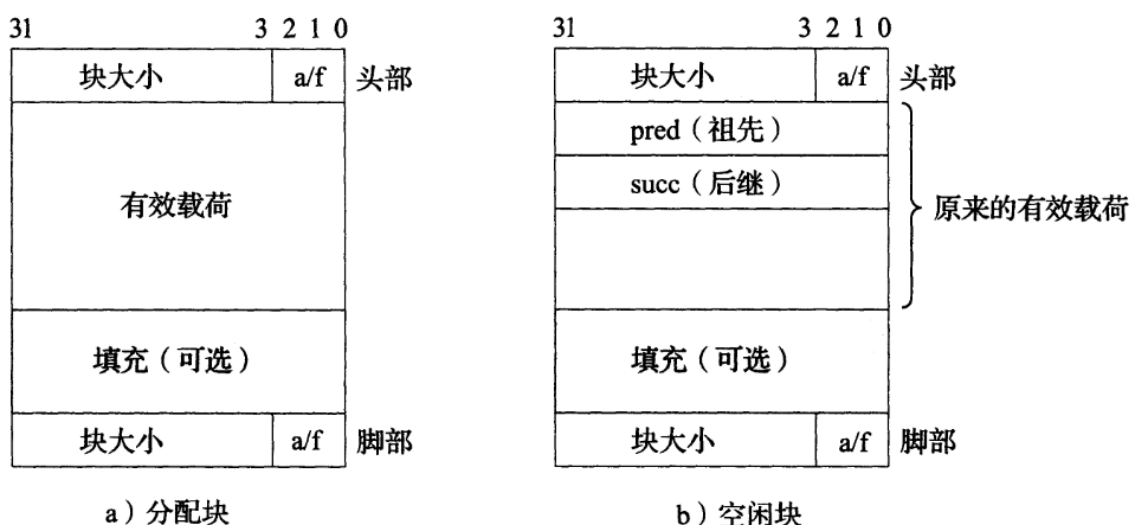
- 1：显式分配器：要求应用显式地释放任何已分配的块。
- 2：隐式分配器：要求分配器检测一个已分配块何时不再使用，那么就释放这个块

分配方式示例：显示空间链表：

由于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，即实现显式空闲链表：使用一个显示数据结构组织空闲块

例如：使用双向空闲链表：在每个空闲块中，都包含一个 `pred`（前驱）和 `succ`

（后继）指针（如图）



维护方法:

一种方法是用后进先出(LIFO)的顺序维护链表,将新释放的块位置放在链表的开始出。使用 LIFO 的顺序和首次适配的放置策略,分配器会最先检查最近使用过的块。在这种情况下,释放一个块可以在常数时间内完成。如果使用了边界标记,那么合并也可以在常数时间内完成。

另一种方法是按照地址顺序来维护链表,其中链表中每个块的地址都小于它后继的地址。在这种情况下,释放一个块需要线性时间的搜索来定位合适的前驱。平衡点在于,按照地址排序的首次适配比 LIFO 排序的首次适配有更高的存储器利用率,接近最佳适配的利用率。一般而言,显示链表的缺点是空闲块必须足够大,以包含所有需要的指针,以及头部和可能的脚部。这就导致了更大的最小块大小。也潜在地提高了内部碎片的程度。

7.10 本章小结

本章主要介绍了 hello 的储存器地址空间,和 intel 的段式管理,页式管理,以及 TLB 和四级页表,三级 Cache 支持下的 VA 到 PA 的变换及内存访问,还有 hello 进程 fork 和 execve 的内存映射,以及缺页故障和处理,还有最后的动态存储分配管理。

(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：采用模型化方法，将设备映射为文件，达到允许 Unix 内核引出一个简单、低级的应用接口的目的。

8.2 简述 Unix IO 接口及其函数

- 1: 打开文件: `int open(char *filename, int flags, mode_t mode)`

Open 函数将 filename 转换为一个文件描述符，并返回描述符数字。

- 2: 关闭文件: `int close(int fd)`

- 3: 读写文件:

`ssize_t read(int fd, void *buf, size_t n)`

`ssize_t write(int fd, const void *buf, size_t n)`

8.3 printf 的实现分析

Printf 源码如下:

```
1. int printf(const char *fmt, ...)
2. {
3.     int i;
4.     char buf[256];
5.
6.     va_list arg = (va_list)((char*)&fmt + 4);
7.     i = vsprintf(buf, fmt, arg);
8.     write(buf, i);
9.
10.    return i;
11. }
```

Printf 中使用 arg 获得一个不定长参数，作为格式化串的输出内容。

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

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)。

8.4 getchar 的实现分析

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

本章讲述了 hello 程序运行时, LinuxIO 设备的管理思想方法和一些 IO 接口及其函数。

(第 8 章 1 分)

结论

到这里, hello 程序和计算机系统这门课都走到了尾声, hello 一开始是一个被当成最常见的程序, 经过相关知识的改造和不同工具不同形式的运行, 一个更加完善的程序就出现了

Emmmmm, 这个 hello 程序, 见过的人谁也不会忘吧

(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。

hello.i	预处理文件
hello.s	汇编语言文件
hello.o	可重定位目标文件
hello	可执行程序文件
hello.elf	hello.o 的 elf 格式文件
hello_objdump.txt	objdump 的反汇编文件



hello.i



hello.o



hello.s



hello.elf



hello_objdump.t
xt



hello

(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] C 预处理器. 维基百科
<https://zh.wikipedia.org/wiki/C%E9%A2%84%E5%A4%84%E7%90%86%E5%99%A8>
- [2] <https://www.cnblogs.com/pianist/p/3315801.html>
- [3] 赵耀东. 新时代的工业工程师[M/OL]. 台北：天下文化出版社，1998
[1998-09-26]. <http://www.ie.nthu.edu.tw/info/ie.newie.htm> (Big5) .
- [4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨：哈尔滨工业大学，1992：
8-13.
- [5] KANAMORI H. Shaking Without Quaking[J]. Science, 1998, 279 (5359) :
2063-2064.
- [6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL].
Science , 1998 , 281 : 331-332[1998-09-23]. <http://www.sciencemag.org/cgi/collection/anatmorp>.

(参考文献 0 分，缺失 -1 分)