

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>計算機科學與技術專業</u>
學 號	<u>1180300510</u>
班 級	<u>1803005</u>
學 生	<u>唐鵬程</u>
指 導 教 師	<u>鄭貴濱</u>

計算機科學與技術學院
2019 年 12 月

摘 要

一个简单的 hello 却拥有着不平凡的一生。本文以 hello 为例，讲述 Linux 下从 C 文件到可执行文件，从程序到进程，到最后程序回收结束的全过程，涉及到预处理、编译、汇编、链接、进程管理、存储管理、I/O 管理等，而这一切都依靠计算机系统软硬件的共同支持，在系统各组成部分的协调工作下，一个简单的应用程序，也显示出高效的系统运行机制。

关键词：计算机系统；hello，进程管理，存储管理，汇编，编译；

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	- 5 -
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	- 6 -
2.4 本章小结	- 7 -
第 3 章 编译	- 8 -
3.1 编译的概念与作用	- 8 -
3.2 在 UBUNTU 下编译的命令	- 8 -
3.3 HELLO 的编译结果解析	- 8 -
3.4 本章小结	- 11 -
第 4 章 汇编	- 13 -
4.1 汇编的概念与作用	- 13 -
4.2 在 UBUNTU 下汇编的命令	- 13 -
4.3 可重定位目标 ELF 格式	- 13 -
4.4 HELLO.O 的结果解析	- 15 -
4.5 本章小结	- 17 -
第 5 章 链接	- 18 -
5.1 链接的概念与作用	- 18 -
5.2 在 UBUNTU 下链接的命令	- 18 -
5.3 可执行目标文件 HELLO 的格式	- 18 -
5.4 HELLO 的虚拟地址空间	- 19 -
5.5 链接的重定位过程分析	- 20 -
5.6 HELLO 的执行流程	- 21 -
5.7 HELLO 的动态链接分析	- 22 -
5.8 本章小结	- 23 -
第 6 章 HELLO 进程管理	- 24 -
6.1 进程的概念与作用	- 24 -
6.2 简述壳 SHELL-BASH 的作用与处理流程	- 24 -
6.3 HELLO 的 FORK 进程创建过程	- 24 -

6.4 HELLO 的 EXECVE 过程	- 25 -
6.5 HELLO 的进程执行	- 25 -
6.6 HELLO 的异常与信号处理	- 26 -
6.7 本章小结	- 28 -
第 7 章 HELLO 的存储管理.....	- 29 -
7.1 HELLO 的存储器地址空间	- 29 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理	- 29 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理	- 29 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换	- 30 -
7.5 三级 CACHE 支持下的物理内存访问	- 31 -
7.6 HELLO 进程 FORK 时的内存映射.....	- 31 -
7.7 HELLO 进程 EXECVE 时的内存映射.....	- 32 -
7.8 缺页故障与缺页中断处理.....	- 32 -
7.9 动态存储分配管理.....	- 33 -
7.10 本章小结	- 34 -
第 8 章 HELLO 的 IO 管理	- 36 -
8.1 LINUX 的 IO 设备管理方法	- 36 -
8.2 简述 UNIX IO 接口及其函数	- 36 -
8.3 PRINTF 的实现分析	- 37 -
8.4 GETCHAR 的实现分析.....	- 38 -
8.5 本章小结	- 38 -
结论.....	- 38 -
附件.....	- 40 -
参考文献.....	- 41 -

第 1 章 概述

1.1 Hello 简介

首先介绍一下 Hello 的 P2P 过程。Program to process,首先产生 program, 先编写好一个 hello.c 文件, 然后经过预处理生成 hello.i 文件, 然后编译器将 hello.i 文件生成 hello.s 汇编文件, 编译器将 hello.s 文件转换为可重定位二进制文件 hello.o, 最后链接器将该文件与头文件中所需的库函数链接形成可执行 elf 文件。在命令行中输入 ./hello, Shell 解析命令为外部命令, 调用 fork 函数创建一个子进程, 由此将 program 变成一个 progress。调用 execve 函数执行程序、加载进程, sbrk 函数开辟内存空间, mmap 将其映射到内存中, 这就是 P2P 的过程。

在运行 hello 程序的过程中, CPU 也发挥着作用。CPU 为 hello 程序分配内存和时间片。CPU 通过取指、译码、执行、访存、写回、更新 PC 等执行 hello 程序。CPU 访问 hello 程序的数据, 首先需要将虚拟地址转换为物理地址, 通过 TLB 和页表加速地址的翻译, cache 加快数据的传输。IO 管理和 Shell 的进程处理和信号处理机制可以处理 hello 运行过程中的信号, 最终, 当进程结束时, shell 调用 waitpid 函数回收子进程, brk 函数回收内存。这就是 O2O 的过程。

1.2 环境与工具

硬件: Inter core i5, X64 CPU, 8GRAM

软件: Windows10, VMware 15, Ubuntu 18.04

调试工具: gcc, gdb, edb, readelf, objdump, vscode

1.3 中间结果

中间文件	作用
hello.i	预处理产生的文件
hello.s	将预处理文件变为汇编文件
hello.o	编译器将汇编文件变为可重定位文件
hello	ELF 格式的可执行文件
hello.l.d	hello 的反汇编文件, 查看链接
hello.d	hello 的反汇编文件, 查看汇编代码

1.4 本章小结

第一章对 hello 进行了简介，介绍了 hello 的 P2P,O2O 的过程，大致概括了从程序到进程，从编译执行到终止的过程，介绍了本次大作业的软硬件环境和调试工具，以及写了在 hello 的整个过程产生的中间文件和其作用。

(第 1 章 0.5 分)

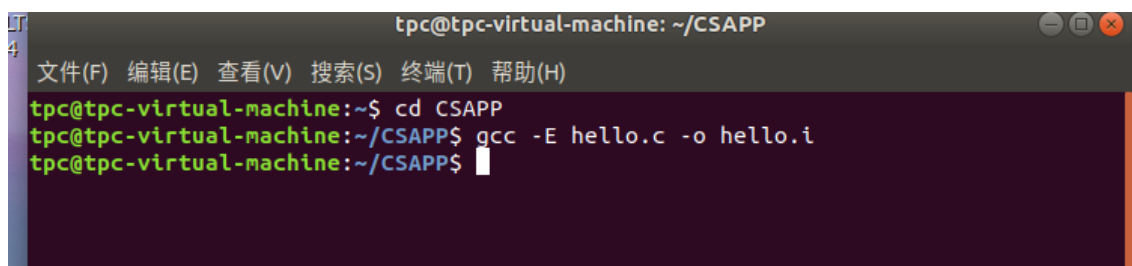
第 2 章 预处理

2.1 预处理的概念与作用

预处理的概念：预处理是 C 语言的一个重要功能，它由预处理程序负责完成，当对一个源文件进行编译时，系统将自动引用预处理程序对源程序中的预处理部分作处理。处理完毕后自动进入对源程序的编译。

预处理的作用：将源文件中以“include”格式包含的文件复制到编译的源文件中，用实际值替换用“#define”定义的字符串。根据“#if”后面的条件决定需要编译的代码。

2.2 在 Ubuntu 下预处理的命令



```
tpc@tpc-virtual-machine: ~/CSAPP
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
tpc@tpc-virtual-machine:~$ cd CSAPP
tpc@tpc-virtual-machine:~/CSAPP$ gcc -E hello.c -o hello.i
tpc@tpc-virtual-machine:~/CSAPP$
```

2.3 Hello 的预处理结果解析

用 vscode 查看 hello.i 的信息，发现在 hello.i 文件中把 include 格式包含的文件复制到了 main 函数之前，hello.i 的末尾是 main 函数，之前都是 include 文件的源代码，总共有三千多行。且可以发现头文件的引用是层层嵌套的。

```
13 # 1 "/usr/include/stdio.h" 1 3 4
14 # 27 "/usr/include/stdio.h" 3 4
15 # 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
16 # 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
17 # 1 "/usr/include/features.h" 1 3 4
18 # 424 "/usr/include/features.h" 3 4
19 # 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
20 # 427 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
21 # 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
22 # 428 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
23 # 1 "/usr/include/x86_64-linux-gnu/bits/long-double.h" 1 3 4
24 # 429 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
25 # 425 "/usr/include/features.h" 2 3 4
```

```
3093 # 1017 "/usr/include/stdlib.h" 2 3 4
3094 # 1026 "/usr/include/stdlib.h" 3 4
3095
3096 # 9 "hello.c" 2
3097
3098
3099 # 10 "hello.c"
3100 int sleepsecs=2.5;
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 }
3119
```

(main 函数截图)

2.4 本章小结

这一章首先介绍了预处理的概念和作用，在 Ubuntu 下输入预处理操作，对 hello.c 进行了预处理操作，且查看了预处理文件的具体内容。

(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：利用编译程序从源语言编写的源程序产生目标程序的过程，把预处理文件转换成汇编语言文件，把高级语言变成计算机可以识别的 2 进制语言，编译分为五个阶段：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。

编译的作用：将高级语言程序解释称为计算机所需的详细机器语言指令集。

3.2 在 Ubuntu 下编译的命令

```
tpc@tpc-virtual-machine:~/CSAPP$ gcc -S hello.i -o hello.s
tpc@tpc-virtual-machine:~/CSAPP$ ls
hello.c hello.i hello.s
```

3.3 Hello 的编译结果解析

3.3.1 数据

1、全局变量

hello.c 定义了一个全局变量 int 型 sleepsecs，且赋初值为 2，将 sleepsecs 存放在 .data 段，.data 段 4 字对齐，sleepsecs 变为 long 型，数值为 2，经过编译之后存放在 .rodata 段。

```
sleepsecs:
    .long 2
    .section .rodata
    .globl sleepsecs
    .data
    .align 4
    .type sleepsecs,@object
    .size sleepsecs,4
```

2、局部变量

main 函数中定义了 int 型局部变量 i，局部变量存储在栈上，从汇编代码中可见，i 存储在 -4 (%rbp) 的地址上，初始值为 0，每一次循环+1，跳出循环的条件为 i>9。

```

    addl $1,-4(%rbp)
.L2:
    movl $0,-4(%rbp)
    jmp .L3
.L3:
    cmpl $9,-4(%rbp)
    jle .L4

```

3、字符串

在 hello.c 的 main 函数中共有两个 printf 输出的字符串，查看汇编代码发现存储在 .rodata 段。

LC0 存储的是 “Usage: Hello 学号 姓名!”，其中汉字和! 一个字符使用三个 utf-8 编码，LC1 存储的是从终端输入的两个参数，argv[0],argv[1]，

```

11 .LC0:
12 .string "Usage: Hello \345\255\246\345\217\267 \345\247\223\345\220\215\357\274\201"
13 .LC1:
14 .string "Hello %s %s\n"
15 .text
16 .globl main
17 .type main,@function

```

3.3.2 赋值

hello.c 中将全局变量 sleepsecs 赋初值为 2.5，sleepsecs 一开始存储在 .data 段，而赋值完成在 .data 段，已经将 sleepsecs 赋值为 long 型的 2，还有对局部变量 i 的赋值，赋初值为 0，由 movl \$0,-4(%rbp) 完成。

3.3.3 类型转换

hello.c 中存在一个隐式类型转换，即 int 型全局变量 sleepsecs 的转换，sleepsecs 赋初值为 2.5，但 sleepsecs 为 int 型即向 0 舍入为 2。

3.3.4 函数及其参数

1、main 函数

main 函数存储在 .text 段中，程序运行时调用。main 函数的参数为 int 型的 argc 和 char** 的 argv，argc 为命令行输入的个数，argv[] 为输入的字符串，这两个参数都从命令行输入。其中 argc 存储在 %rdi，argv 存储在 %rsi。然后将 argc，argv 都存储在栈中，其中，argc 存储在 -20 (%rbp)，argv 存储在 -32 (%rbp)，在运算时，直接使用栈中存储的数据。

```

    movl %edi,-20(%rbp)
    movq %rsi,-32(%rbp)
    cmpl $3,-20(%rbp)

```

2、printf 函数

hello.c 中使用了两个 printf 函数，其中第一个 printf 函数打印常量字符串，该字符串存储在.LC0 中，当判断条件 `argc != 3` 满足时，将 LC0 的值赋给%rdi,其中%rip 是下一条指令的地址，然后调用 puts 函数输出字符串。第二个 printf 函数打印的是从命令行写入的字符串，分别为 `argv[0]`,`argv[1]`,还有.LC1。编译器使用多次 `movq` 指令,将 `argv[0]`存储在%rdx,`argv[1]`存储在%rsi,然后 `leaq` 指令将.LC1 存储在%rdi,最后调用 printf 函数输出字符串。

```
leaq .LC0(%rip), %rdi
call puts@PLT
movl $1, %edi
call exit@PLT
```

```
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
```

3、getchar 函数

getchar 函数没有参数，在循环结束之后直接调用。

```
call getchar@PLT
```

4、sleep 函数

sleep 函数的参数是 `sleepsecs(%rip)`,`movl` 将参数赋值给%eax,然后再赋值给%edi,最后调用 sleep 函数。

```
movl sleepsecs(%rip), %eax
movl %eax, %edi
call sleep@PLT
```

5、exit 函数

将 1 赋值给%edi,然后 `call exit` 函数，1 位 exit 的唯一参数，表示 exit 非正常运行退出程序。

```
movl $1,%edi
call exit@PLT
```

3.3.5、数组操作

hello.c 中使用了 `argv` 数组。数组通过地址访存，由首地址为基准进行地址的计算。从汇编代码中可以看到，编译器通过 `movq` 取内容指令和 `addq` 指令得到 `argv[0]`,`argv[1]`,其中 `argv` 的首地址存储在 `-32(%rbp)`,然后根据指针 8 字节大小+8,+16 得到地址。

```
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
```

3.3.6、关系运算 and 控制转移

hello.c 中使用到了比较运算和控制转移，比较运算分别为 `argc!=3` 和 `i<=9`，若 `argc!=3`，跳转到.L2 执行，若 `i<=9`,跳转到.L4 执行。

编译器通过 `cmp` 指令进行比较，然后调用 `je,jle` 等条件跳转指令（设置 CF,ZE,SF,OF 条件码，条件码不同，条件不同）跳转到某一地址执行，执行完之后返回下一条指令，这是控制转移，常常和关系运算一起使用。在这里只使用了 `cmp` 指令。其实还有 `test` 指令，和 `cmp` 指令类似。

常在 `if else,while,for` 等选择结构、循环结构中使用。

```
cmpl $3,-20(%rbp)    cmpl $9,-4(%rbp)
je .L2               jle .L4
```

3.4 本章小结

这一章主要讲述了编译器将预处理文件转换为汇编语言文件时，对于不同的数据类型和各种操作，编译器是如何处理的。以 `hello.c` 和 `hello.s` 为例，包含了变量、常量、函数、关系运算、数组、控制转移等操作，通过对这些操作汇编代码的

解读，了解了编译器的编译功能和汇编语言的基本操作。

(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：汇编器把汇编语言转换为相对应的机器语言，再把生成的机器语言变成可重定位目标文件，即由.s 文件生成.o 文件，其中，可重定位目标文件包含二进制代码和数据。

汇编的作用：把编译器产生的汇编语言翻译成二进制机器语言，由.s 文件生成.o 文件。

4.2 在 Ubuntu 下汇编的命令

```
tpc@tpc-virtual-machine:~/CSAPP$ gcc -c hello.s -o hello.o
tpc@tpc-virtual-machine:~/CSAPP$ ls
hello.c hello.i hello.o hello.s
```

4.3 可重定位目标 elf 格式

ELF 头
.text
.rodata
.data
.bss
.symtab
.rel.txt
.rel.data
.debug
.line
.strtab
节头部表

.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变量

.symtab:一个符号表，存放在程序中定义和引用的函数和全局变量的信息。

.rel.txt:一个.text 节中位置的列表

.rel.data:被模块引用或定义的所有全局变量的重定位信息。

.debug:一个调试符号表, 包括程序中定义的局部变量和类型定义, 程序中定义的全局变量, 以及原始的 C 源文件

.line:原始 C 源程序中的行号和.text 节中及其指令之间的映射

.strtab:一个字符串表, 包括.symtab,.debug 节中的符号表。

下面为 hello.o 各节的基本信息, 通过 readelf -a hello.o 指令得到

首先看到的是 ELF 头的相关信息, 包括 ELF 头的字节大小, 文件的类型, 节头的数量, 字符串表索引节头, 入口地址等信息, 这些信息帮助链接器解释目标文件的信息。

```
tpc@tpc-virtual-machine:~/CSAPP$ readelf -a hello.o
ELF 头:
  Magic:      7f 45 4c 46 02 01 01 00 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:                               REL (可重定位文件)
  系统架构:                               Advanced Micro Devices X86-64
  版本:                               0x1
  入口点地址:                               0x0
  程序头起点:                               0 (bytes into file)
  Start of section headers:               1152 (bytes into file)
  标志:                               0x0
  本头的大小:                               64 (字节)
  程序头大小:                               0 (字节)
  Number of program headers:               0
  节头大小:                               64 (字节)
  节头数量:                               13
  字符串表索引节头: 12
```

接下来可以看到节头部表, 节头部表记录了每个节的信息, 包括节的名称、地址、类型和偏移量。

节头: [号]	名称 大小	类型 全体大小	地址 标志	链接 信息	偏移量 对齐
[0]	0000000000000000	NULL	0000000000000000	0	0
[1]	.text 0000000000000081	PROGBITS	0000000000000000	AX	0
[2]	.rela.text 00000000000000c0	RELA	0000000000000000	0	0
[3]	.data 0000000000000004	PROGBITS	0000000000000000	I	1
[4]	.bss 0000000000000000	NOBITS	0000000000000000	WA	0
[5]	.rodata 000000000000002b	PROGBITS	0000000000000000	A	0
[6]	.comment 000000000000002c	PROGBITS	0000000000000000	MS	0
[7]	.note.GNU-stack 0000000000000000	PROGBITS	0000000000000000	0	0
[8]	.eh_frame 0000000000000038	PROGBITS	0000000000000000	A	0
[9]	.rela.eh_frame 0000000000000018	RELA	0000000000000000	I	10
[10]	.symtab 00000000000000198	SYMTAB	0000000000000000	11	9
[11]	.strtab 000000000000004d	STRTAB	0000000000000000	0	0
[12]	.shstrtab 0000000000000061	STRTAB	0000000000000000	0	0

接下来可以看到.text 节重定位的信息。重定位节中包括所有需要重定位的符

号的信息，包括偏移量、信息、类型、符号值和符号名称+加数。当链接器把这个可重定位目标文件与其他文件相结合时，需要修改这些符号的位置。

其中，这些符号有两种类型。第一种类型为 `R_X86_64_PC32`，表示重定位 PC 相对引用。链接器首先计算出引用的运行时地址，`refaddr=ADDR(.text)+偏移量`，然后更新该引用，使得它在运行时指向符号。这里的加数就是 `r.append`，偏移量就是 `r.offset`。故 `*refptr=ADDR(运行时地址)+r.append-refaddr`，因此 PC 的值为 `PC+*refptr`，之后 CPU 调用 `call` 指令。

第二种类型为 `R_X86_64_PLT32`，为位置无关代码，即无需重定位的代码。

另外，还有一种类型在这里没有显示，`R_X86_64_32`，表示重定位 PC 绝对引用。地址计算为 `*refptr=ADDR(运行时地址)+r.append`

在 `.rela.text` 表之后是 `.rela.eh_frame`，保存 `eh_frame` 的重定位信息。

重定位节 '.rela.text' at offset 0x340 contains 8 entries:					
偏移量	信息	类型	符号值	符号名称 + 加数	
000000000018	000500000002	R_X86_64_PC32	0000000000000000	.rodata - 4	
00000000001d	000c00000004	R_X86_64_PLT32	0000000000000000	puts - 4	
000000000027	000d00000004	R_X86_64_PLT32	0000000000000000	exit - 4	
000000000050	000500000002	R_X86_64_PC32	0000000000000000	.rodata + 1a	
00000000005a	000e00000004	R_X86_64_PLT32	0000000000000000	printf - 4	
000000000060	000900000002	R_X86_64_PC32	0000000000000000	sleepsecs - 4	
000000000067	000f00000004	R_X86_64_PLT32	0000000000000000	sleep - 4	
000000000076	001000000004	R_X86_64_PLT32	0000000000000000	getchar - 4	

重定位节 '.rela.eh_frame' at offset 0x400 contains 1 entry:					
偏移量	信息	类型	符号值	符号名称 + 加数	
000000000020	000200000002	R_X86_64_PC32	0000000000000000	.text + 0	

接下来我们可以看到 `.symtab` 表，记录了 `hello.c` 中调用的函数和全局变量的的名称，类型，地址等信息，`value` 地址信息，在可重定位文件中是起始位置的偏移量。

`bind` 表示符号是全局的还是本地的。UND 表示为在本文件中定义的符号。

Symbol table '.symtab' contains 17 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	6	
9:	0000000000000000	4	OBJECT	GLOBAL	DEFAULT	3	sleepsecs
10:	0000000000000000	129	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	sleep
16:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	getchar

4.4 Hello.o 的结果解析


```

0000000000000000 <main>:
0: 55                push    %rbp
1: 48 89 e5          mov     %rsp,%rbp
4: 48 83 ec 20       sub     $0x20,%rsp
8: 89 7d ec          mov     %edi,-0x14(%rbp)
b: 48 89 75 e0       mov     %rsi,-0x20(%rbp)
f: 83 7d ec 03       cmpl   $0x3,-0x14(%rbp)
13: 74 16             je      2b <main+0x2b>
15: 48 8d 3d 00 00 00 lea     0x0(%rip),%rdi    # 1c <main+0x1c>
18: R_X86_64_PC32    .rodata-0x4
1c: e8 00 00 00 00    callq  21 <main+0x21>
1d: R_X86_64_PLT32    puts-0x4
21: bf 01 00 00 00    mov     $0x1,%edi
26: e8 00 00 00 00    callq  2b <main+0x2b>
27: R_X86_64_PLT32    exit-0x4
2b: c7 45 fc 00 00 00 movl   $0x0,-0x4(%rbp)
32: eb 3b            jmp     6f <main+0x6f>
34: 48 8b 45 e0       mov     -0x20(%rbp),%rax
38: 48 83 c0 10       add     $0x10,%rax
3c: 48 8b 10          mov     (%rax),%rdx
3f: 48 8b 45 e0       mov     -0x20(%rbp),%rax
43: 48 83 c0 08       add     $0x8,%rax
47: 48 8b 00          mov     (%rax),%rax
4a: 48 89 c6          mov     %rax,%rsi
4d: 48 8d 3d 00 00 00 lea     0x0(%rip),%rdi    # 54 <main+0x54>
50: R_X86_64_PC32    .rodata+0x1a
54: b8 00 00 00 00    mov     $0x0,%eax
59: e8 00 00 00 00    callq  5e <main+0x5e>
5a: R_X86_64_PLT32    printf-0x4
5e: 8b 05 00 00 00 00 mov     0x0(%rip),%eax    # 64 <main+0x64>
60: R_X86_64_PC32    sleepsecs-0x4
64: 89 c7            mov     %eax,%edi
66: e8 00 00 00 00    callq  6b <main+0x6b>
67: R_X86_64_PLT32    sleep-0x4
6b: 83 45 fc 01       addl   $0x1,-0x4(%rbp)
6f: 83 7d fc 09       cmpl   $0x9,-0x4(%rbp)
73: 7e bf            jle     34 <main+0x34>
75: e8 00 00 00 00    callq  7a <main+0x7a>
76: R_X86_64_PLT32    getchar-0x4
7a: b8 00 00 00 00    mov     $0x0,%eax
7f: c9              leaveq  %eax
80: c3              retq

```

(.s 见第三章)

1、机器语言的构成

机器语言是计算机能够直接识别的二进制代码，在 `hello.o` 的反汇编文件中，使用 16 进制表示，由多个字节表示，每个字节由两个十六进制数表示。

2、机器语言与汇编语言的映射

机器语言对应的反汇编的每一条语句由多个字节表示，每一组字节都是一条汇编指令，不过由 `.o` 文件生成的反汇编文件与 `hello.s` 的汇编语言存在差别。

(1)、操作数的差别

`hello.s` 中立即数表示使用十进制，而生成的反汇编立即数使用 16 进制表示。

(2)、函数调用

`hello.s` 中函数调用直接 `call+函数名` 即可，`call printf` 等，而在反汇编中则不同，链接器为函数调用找到匹配的函数的可执行代码的位置，故在 `call` 指令调用函数写的是地址，地址是符号重定位之后的地址，即运行的地址，同时也要记下 PC 下一条指令的地址。如图中的 `getchar`、`sleep` 函数，会加载可重定位信息以计算重定位后的地址。

(3)、分支转移

`hello.s` 中的跳转指令是像 `jmp .L1` 一样的格式，`.L1` 是代码段。而反汇编中的跳转指令是直接跳转到某个地址，或是直接的地址，或是函数相对偏移地址。

(4)、全局变量

hello.s 中调用全局变量, 如将全局变量值直接赋给寄存器, 全局变量的值即为.LC0。而在反汇编代码中, 将可重定位文件重定位之后计算出全局变量的地址, 然后将该全局变量地址的值赋值给寄存器。如 `lea 0x0(%rip) %rdi`。

4.5 本章小结

这一章将.s 文件转换成.o 可重定位文件。分析了 ELF 格式下可重定位文件的组成和各个节存储的内容, 使用 `readelf` 看具体查看。用 `objdump` 工具对可重定位文件进行了反汇编, 并将反汇编代码与汇编代码进行比较, 比较两者之间的不同之处, 得出机器语言和汇编代码之间的映射关系。

(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：链接是将各种代码和数据片段收集并组合成一个单一文件的过程，这个文件可被加载到内存中并执行，也就是从.o 文件到可执行文件的过程。链接是由链接器的程序自动执行的。链接包含符号解析和重定位两步。链接器将每个符号引用与符号的定义相关联，将符号在可重定位文件的位置重定位至可执行文件的位置。

链接的作用：可以将一个大型的应用程序分解成更小、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，只需要简单地重新编译，并重新链接应用，而不必重新编译其他文件。

5.2 在 Ubuntu 下链接的命令

```
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
```

```
tpc@tpc-virtual-machine:~/CSAPP$ 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
tpc@tpc-virtual-machine:~/CSAPP$ ls
hello hello.c hello.i hello.o hello.s
tpc@tpc-virtual-machine:~/CSAPP$
```

5.3 可执行目标文件 hello 的格式

[19]	.got.plt	PROGBITS	0000000000601000	00001000
	0000000000000040	0000000000000008	WA 0 0	8
[20]	.data	PROGBITS	0000000000601040	00001040
	0000000000000008	0000000000000000	WA 0 0	4
[21]	.comment	PROGBITS	0000000000000000	00001048
	000000000000002b	0000000000000001	MS 0 0	1
[22]	.symtab	SYMTAB	0000000000000000	00001078
	0000000000000498	0000000000000018	23 28	8
[23]	.strtab	STRTAB	0000000000000000	00001510
	0000000000000150	0000000000000000	0 0	1
[24]	.shstrtab	STRTAB	0000000000000000	00001660
	00000000000000c5	0000000000000000	0 0	1

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	00000000
[1]	.interp 000000000000001c	PROGBITS 0000000000000000	0000000000400200 A 0 0	00000200	00000200
[2]	.note.ABI-tag 0000000000000020	NOTE 0000000000000000	000000000040021c A 0 0	0000021c	0000021c
[3]	.hash 0000000000000034	HASH 0000000000000004	0000000000400240 A 5 0	00000240	00000240
[4]	.gnu.hash 000000000000001c	GNU_HASH 0000000000000000	0000000000400278 A 5 0	00000278	00000278
[5]	.dynsym 00000000000000c0	DYNSYM 0000000000000018	0000000000400298 A 6 1	00000298	00000298
[6]	.dynstr 0000000000000057	STRTAB 0000000000000000	0000000000400358 A 0 0	00000358	00000358
[7]	.gnu.version 0000000000000010	VERSYM 0000000000000002	00000000004003b0 A 5 0	000003b0	000003b0
[8]	.gnu.version_r 0000000000000020	VERNEED 0000000000000000	00000000004003c0 A 6 1	000003c0	000003c0
[9]	.rela.dyn 0000000000000030	RELA 0000000000000018	00000000004003e0 A 5 0	000003e0	000003e0
[10]	.rela.plt 0000000000000078	RELA 0000000000000018	0000000000400410 AI 5 19	00000410	00000410
[11]	.init 0000000000000017	PROGBITS 0000000000000000	0000000000400488 AX 0 0	00000488	00000488
[12]	.plt 0000000000000060	PROGBITS 0000000000000010	00000000004004a0 AX 0 0	000004a0	000004a0
[13]	.text 0000000000000132	PROGBITS 0000000000000000	0000000000400500 AX 0 0	00000500	00000500
[14]	.fini 0000000000000009	PROGBITS 0000000000000000	0000000000400634 AX 0 0	00000634	00000634
[15]	.rodata 000000000000002f	PROGBITS 0000000000000000	0000000000400640 A 0 0	00000640	00000640
[16]	.eh_frame 00000000000000fc	PROGBITS 0000000000000000	0000000000400670 A 0 0	00000670	00000670
[17]	.dynamic 00000000000001a0	DYNAMIC 0000000000000010	0000000000600e50 WA 6 0	00000e50	00000e50
[18]	.got 0000000000000010	PROGBITS 0000000000000008	0000000000600ff0 WA 0 0	00000ff0	00000ff0

通过 `readelf -a hello` 得到了 hello 的 ELF 格式，hello 共由 24 个段组成，从 .interp 到 .shstrtab。在地址这一栏中我们可以看到各段的起始地址，在大小一栏中即可得到各段的大小。

5.4 hello 的虚拟地址空间

打开 edb，加载 hello，打开 symbolviewer 查看虚拟地址各段信息。

Loaded symbols:	
Filter	
0x0000000000400200:	hello!.interp
0x000000000040021c:	hello!.interp+0x1c
0x0000000000400240:	hello!.hash
0x0000000000400278:	hello!.gnu.hash
0x0000000000400298:	hello!.dynsym
0x0000000000400358:	hello!.dynsym+0xc0
0x00000000004003b0:	hello!.gnu.version
0x00000000004003c0:	hello!.gnu.version+0x10
0x00000000004003e0:	hello!.gnu.version_r+0x20
0x0000000000400410:	hello!.rela.dyn+0x30
0x0000000000400488:	hello!.rela.plt+0x78
0x0000000000400488:	hello!_init
0x00000000004004a0:	hello!.plt
0x00000000004004b0:	hello!puts@plt
0x00000000004004c0:	hello!printf@plt
0x00000000004004d0:	hello!getchar@plt
0x00000000004004e0:	hello!exit@plt
0x00000000004004f0:	hello!sleep@plt
0x0000000000400500:	hello!.plt+0x60

```

0x000000000400500: hello! start
0x000000000400530: hello! dl_relocate_static_pie
0x000000000400532: hello! main
0x0000000004005c0: hello! __libc_csu_init
0x000000000400630: hello! __libc_csu_fini
0x000000000400634: hello! .fini
0x000000000400634: hello! .fini
0x000000000400640: hello! .rodata
0x000000000400640: hello! _IO_stdin_used
0x000000000400670: hello! .eh_frame
0x000000000600e50: hello! .dynamic
0x000000000600e50: hello! DYNAMIC
0x000000000600e50: hello! __init_array_end
0x000000000600e50: hello! __init_array_start
0x000000000600ff0: hello! .dynamic+0x1a0
0x000000000601000: hello! .got+0x10
0x000000000601000: hello! GLOBAL_OFFSET_TABLE_
0x000000000601040: hello! .got.plt+0x40
0x000000000601040: hello! _data_start
0x000000000601040: hello! data_start
0x000000000601044: hello! sleepsecs
0x000000000601048: hello! _bss_start
0x000000000601048: hello! .edata
0x000000000601048: hello! .end

```

从这里我们看出各段的起始地址，和 5.3 中使用 `readelf` 指令看到的是一样的，如 `.interp` 都是 `0x0400200`，另外，这里按照地址从小到大的顺序写出了在各段之中含有的其他信息。如 `.interp` 之后是 `.interp+0x1C`

5.5 链接的重定位过程分析

```

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      # 400644 <_IO_stdin_used+0x4>
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
400579: 48 8b 00          mov     (%rax),%rax
40057c: 48 89 c6          mov     %rax,%rsi
40057f: 48 8d 3d dc 00 00 00 lea     0xdc(%rip),%rdi      # 400662 <_IO_stdin_used+0x22>
400586: b8 00 00 00 00    mov     $0x0,%eax
40058b: e8 30 ff ff ff    callq   4004c0 <printf@plt>
400590: 8b 05 ae 0a 20 00 mov     0x200aae(%rip),%eax   # 601044 <sleepsecs>
400596: 89 c7            mov     %eax,%edi
400598: e8 53 ff ff ff    callq   4004f0 <sleep@plt>
40059d: 83 45 fc 01       addl    $0x1,-0x4(%rbp)
4005a1: 83 7d fc 09       cmpl    $0x9,-0x4(%rbp)
4005a5: 7e bf            jle     400566 <main+0x34>
4005a7: e8 24 ff ff ff    callq   4004d0 <getchar@plt>
4005ac: b8 00 00 00 00    mov     $0x0,%eax
4005b1: c9              leaveq  %eax,%rsi
4005b2: c3              retq
4005b3: 66 2e 0f 1f 84 00 00 nopw    %cs:0x0(%rax,%rax,1)
4005ba: 00 00 00          nopb
4005bd: 0f 1f 00          nopl    (%rax)

```

观察 `hello` 和 `hello.o`，可以发现在汇编代码上没有发生实质性的变化，主要是地址发生了改变，链接之前，`.o` 文件中 `main` 函数的反汇编代码从地址 0 开始往下，可以认为是相对偏移地址，而在链接之后，在 `main` 函数之前还链接上了其他的库文件，因此 `hello` 的 `main` 函数是从地址 `0x400532` 开始的，这时，在 `main` 函数中每一条指令的地址，每一个函数的地址都可认为是绝对地址，是 CPU 可以直接访问的地址。在 `hello` `main` 函数中的绝对地址是通过可重定位文件中地址的偏移量加上起始地址得到的。

另外，`hello` 和 `hello.o` 在 ELF 格式下的段的个数种类发生了变化，`hello` 增加了如 `.interp`, `.hash` 等段。

接下来以 `hello.o` 为例，说明链接的过程。链接主要包括解析符号和重定位两步。在重定位之前，汇编器在 `hello.o` 文件的重定位段记录了需要重定位的符号和相应的类型和偏移量。链接器通过对符号的解析（包括局部符号和全局符号），将每个符号的引用和符号的定义相关联。这之后还需要将命令行输入的静态库链接，然后就开始重定位，在重定位过程中，将合并输入模块，并为每个符号分配运行时的地址。

首先需要对符号和节进行重定位。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，赋给定义的每个节和符号，此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。

然后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。

运行时地址的计算和重定位引用的计算在第四章中做了介绍。

5.6 `hello` 的执行流程

从加载 `hello` 到 `_start`

程序名
<code>ld-2.27.so!_dl_start</code>
<code>ld-2.27.so!_dl_init</code>
<code>hello!start</code>

到 `call main`

程序名
<code>libc-2.27.so! libc_start_main</code>
<code>libc-2.27.so! _dl_fixup</code>
<code>libc-2.27.so! libc_csu_init</code>
<code>libc-2.27.so! setjmp</code>
<code>hello!main</code>

到结束

程序名
hello!puts@plt
hello!exit@plt
ld-2.27.so! dl_fixup
ld-2.27.so!exit

5.7 Hello 的动态链接分析

延迟绑定（将过程地址的绑定推迟到第一次调用该过程时）通过两个数据结构之间简洁但又有些复杂的交互来实现，这两个数据结构是：GOT 和过程链接表（PLT）。GOT 是数据段的一部分，而 PLT 是代码段的一部分。

编译器在数据段开始的地方创建全局偏移量表（GOT），在加载时，动态链接器会重定位 GOT 中的每个条目，使得它包含目标的正确的绝对地址。

每个被可执行程序调用的库函数都有它自己的 PLT 条目，每个条目负责调用一个具体的函数。

接下来通过 edb 调试，观察在 dl_init 前后，动态链接项目的变化。

.got.plt 的起始地址为 0x601000,在 datadump 中找到该位置。

在 dl_init 之前：

```
00000000:00600ff0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00601000 50 0e 60 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
```

edb 单步调试至 dl_init 时，可以发现.got.plt 发生了改变

```
00000000:00600ff0 b0 7a 48 a6 9e 7f 00 00 00 00 00 00 00 00 00 00 z
00000000:00601000 50 0e 60 00 00 00 00 00 70 01 a8 a6 9e 7f 00 00 P
00000000:00601010 50 e7 86 a6 9e 7f 00 00 b6 04 40 00 00 00 00 P
```

可以看到 dl.init 之后出现了两个地址，为 0x7f9ea6a80170 和 0x7f9e4686e750 而这两个就是 GOT[1]和 GOT[2]，GOT[1]包含动态链接器在解析函数地址时会使用的信息,GOT[2]是动态链接器在 ld-linux.so 模块的入口点。查看该地址的内容发现是动态链接函数。

```
00007f9e:a686e750 53                push rbx
00007f9e:a686e751 48 89 e3          mov rbx, rsp
00007f9e:a686e754 48 83 e4 c0       and rsp, 0xffffffffffffc0
00007f9e:a686e758 48 2b 25 a9 00 21 00 sub rsp, [rel 0x7f9ea6a7e808]
00007f9e:a686e75f 48 89 04 24       mov [rsp], rax
00007f9e:a686e763 48 89 4c 24 08    mov [rsp+8], rcx
00007f9e:a686e768 48 89 54 24 10    mov [rsp+0x10], rdx
00007f9e:a686e76d 48 89 74 24 18    mov [rsp+0x18], rsi
00007f9e:a686e772 48 89 7c 24 20    mov [rsp+0x20], rdi
00007f9e:a686e777 4c 89 44 24 28    mov [rsp+0x28], r8
00007f9e:a686e77c 4c 89 4c 24 30    mov [rsp+0x30], r9
00007f9e:a686e781 b8 ee 00 00 00    mov eax, 0xee
00007f9e:a686e786 31 d2            xor edx, edx
00007f9e:a686e788 48 89 94 24 50 02 00 mov [rsp+0x250], rdx
00007f9e:a686e790 48 89 94 24 58 02 00 mov [rsp+0x258], rdx
00007f9e:a686e798 48 89 94 24 60 02 00 mov [rsp+0x260], rdx
00007f9e:a686e7a0 48 89 94 24 68 02 00 mov [rsp+0x268], rdx
00007f9e:a686e7a8 48 89 94 24 70 02 00 mov [rsp+0x270], rdx
00007f9e:a686e7b0 48 89 94 24 78 02 00 mov [rsp+0x278], rdx
00007f9e:a686e7b8 0f                db 0xf
00007f9e:a686e7b9 c7                db 0xc7
00007f9e:a686e7ba 64 24 40         and al, 0x40
```

5.8 本章小结

这一章介绍了链接的过程和重定位的过程，以 `hello.o` 链接静态库形成 `hello` 可执行文件为例，对 `hello` 可执行文件的格式进行分析，并使用 `edb` 调试工具和 `objdump` 工具分析 `hello` 的虚拟地址空间，`hello` 的可重定位的过程（包括地址的计算方式，`hello.o` 和 `hello` 的比较等），还有对动态链接的分析。

（第 5 章 1 分）

第 6 章 hello 进程管理

6.1 进程的概念与作用

进程的概念：进程是一个执行中程序的实例，系统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的状态组成的。包括存放在内存中的程序的代码和数据，它的栈、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。

进程的作用：提供给应用程序一个独立的逻辑控制流，它提供一个假象，好像我们的程序独占地使用处理器；提供给应用程序一个私有的地址空间，它提供一个假象，好像我们的程序独占地使用内存系统。

6.2 简述壳 Shell-bash 的作用与处理流程

Shell-bash 是一个交互型的应用级程序，它代表用户运行其他程序。Shell 执行一系列的读/求值步骤，然后终止。度步骤读取来自用户的一个命令行，求值步骤解析命令行，并代表用户运行程序。

对命令行求值的函数解析以空格分隔的命令行参数，并构造最终会传递给 `execve` 的 `argv` 向量。第一个参数被假设为要么是一个内置的 shell 命令名，马上会解释这个命令，要么是一个可执行目标文件，会在一个新的子进程的上下文中加载并运行这个文件。

在解析命令行之后，调用函数检查第一个命令行参数是否是一个内置的 shell 命令。如果是，就立即解释。否则，shell 创建一个子进程在子进程中执行所请求的程序，如果用户要求在后台运行该程序，那么 shell 返回到循环的顶部，等待下一个命令行。Shell 使用 `waitpid` 函数等待作业终止，当作业终止时，shell 就开始下一轮迭代。

6.3 Hello 的 fork 进程创建过程

在命令行输入 `./hello 学号 姓名`，shell 检查该命令是否为内置命令，显然这不是内置命令。于是，shell 调用 `fork` 函数创建一个新的子进程，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，这意味着父进程调用 `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区别在于有不同的 PID。

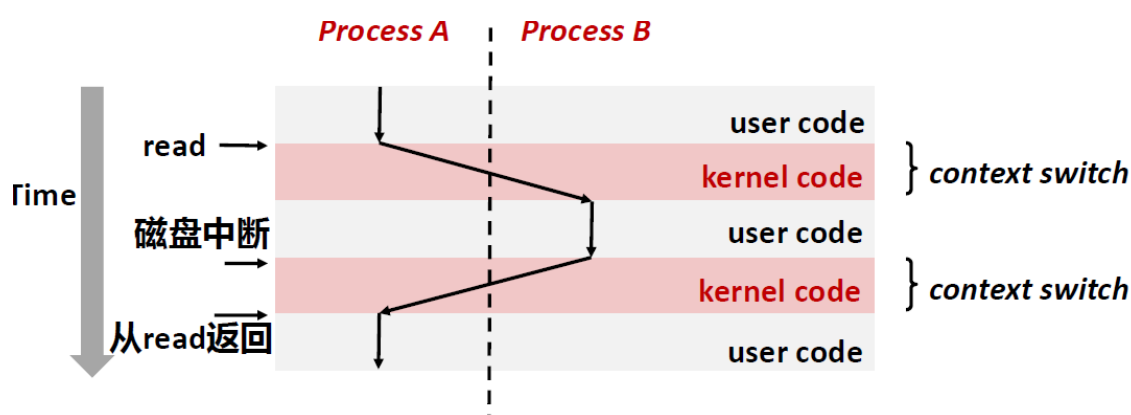
之后，更改进程组编号，准备 hello 的 `execve`。

6.4 Hello 的 execve 过程

在 shell 新创建的子进程中，`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`，只有出现错误是，`execve` 才会返回到调用程序。

在 `execve` 加载 `hello` 之后，调用启动代码来执行 `hello`，新的代码和数据段初始化为可执行文件的内容，跳转到 `_start` 调用 `libc_start_main` 设置栈，并将控制传递给新程序的主函数，

6.5 Hello 的进程执行



在执行 `hello` 程序之后，`hello` 进程一开始运行在用户模式，进程从用户模式变为内核模式的唯一方法是通过中断、故障等异常的调用，当进程处于内核模式时，可以访问任何内存位置，调用任何指令。当处理程序返回到应用程序代码时，从内核模式改为用户模式。

内核为每一个进程维持一个上下文，上下文就是内核重新启动一个被抢占的进程所需的状态。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。

当内核选择一个新的进程运行时，即内核调度这个进程，内核使用上下文切换的机制来控制转移到新的进程：保存当前进程的上下文，恢复某个先前被强占的进程被保存的上下文，将控制传递给这个新恢复的进程。

在 `hello` 程序运行时，会有其他进程并发地运行，这些进程时间与 `hello` 重叠，为并发流，这些进程轮流运行，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。

接下来根据上述知识分析一下 `hello` 的进程调度。`hello` 一开始运行在用户模式，内核保存一个上下文，继续运行调用 `printf` 函数，系统调用使得进程从用户模式变成内核模式，在 `printf` 函数执行完之后又返回到用户模式，继续运行调用 `sleep`

函数，此时会有些不同，由于该进程进行休眠，内核进行上下文切换，调用其他进程运行，同时计数器记录休眠的时间，等到休眠的时间到时，系统发生中断，再次进行上下文切换，转换到 `hello` 进程原先运行的位置。继续运行，遇到循环之后，`hello` 进程会多次进行用户模式和内核模式的转变。之后调用 `getchar` 函数，进入内核模式，需要完成从键盘缓冲区到内存的数据传输，故上下文切换，运行其它进程，当数据传输结束，发生中断，再次上下文切换，回到 `hello` 进程，此时 `hello` 进程就运行结束了，`return`，`hello` 进程运行终止。

6.6 `hello` 的异常与信号处理

6.6.1、`hello` 的异常

(1)、中断

中断时异步发生的，是来自处理器外部的 I/O 设备的信号的结果，中断处理程序运行之后，返回到下一条指令。

(2)、陷阱和系统调用

陷阱是有意的异常，是执行一条指令的结果，就像中断处理程序一样，陷阱处理程序将控制返回到下一条指令，在用户程序和内核之间提供一个像过程一样的接口，即系统调用。如读一个文件、创建一个进程、加载一个新的程序等。

(3)、故障

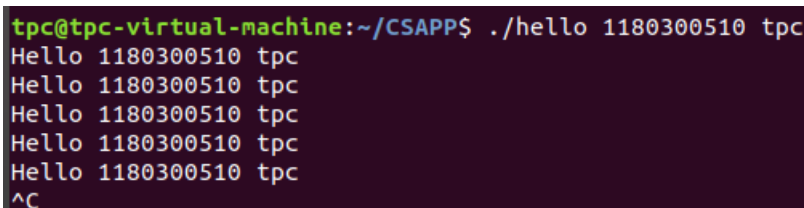
故障是由错误情况引起的，当故障发生时，是利器将控制转移给故障处理程序，如果错误情况可以修正，则将控制返回到引起故障指令，重新执行，否则处理程序返回到内核 `abort`，终止故障的应用程序。

(4)、终止

终止是不可恢复的致命错误造成的结果，会终止应用程序。

6.6.2、`hello` 的信号

SIGINT 中断信号



```
tpc@tpc-virtual-machine:~/CSAPP$ ./hello 1180300510 tpc
Hello 1180300510 tpc
Hello 1180300510 tpc
Hello 1180300510 tpc
Hello 1180300510 tpc
Hello 1180300510 tpc
^C
```

当用户输入 `ctrl-c` 时产生中断信号，导致内核发送一个 `SIGINT` 信号到前台工作组中的每个进程，默认终止前台作业，在这里，`hello` 被终止。

SIGTSTP 信号

```

tpc@tpc-virtual-machine:~/CSAPP$ ./hello 1180300510 tpc
Hello 1180300510 tpc
Hello 1180300510 tpc
Hello 1180300510 tpc
Hello 1180300510 tpc
Hello 1180300510 tpc
Hello 1180300510 tpc
Hello 1180300510 tpc
^Z
[1]+  已停止                  ./hello 1180300510 tpc

```

输入 `ctrl-z` 会发送一个 SIGTSTP 信号到前台进程组中的每个进程，默认情况下，停止（挂起）前台作业。

```

tpc@tpc-virtual-machine:~/CSAPP$ ./hello 1180300510 tpc
Hello 1180300510 tpc
cscnHello 1180300510 tpc
cascmlmmHello 1180300510 tpc

Hello 1180300510 tpc
^C

```

在程序执行过程中，可以在命令行中乱按，包括回车，对于程序运行来说没有影响。通过 `ps` 命令我们可以看到 `hello` 进程的 `pid`，`ctrl-z` 后 `hello` 进程被挂起

```

tpc@tpc-virtual-machine:~/CSAPP$ ps
  PID TTY          TIME CMD
 10288 pts/1        00:00:00 bash
 11153 pts/1        00:00:00 hello
 11154 pts/1        00:00:00 ps

```

`jobs` 命令看到 `hello` 进程的状态

```

tpc@tpc-virtual-machine:~/CSAPP$ jobs
[1]+  已停止                  ./hello 1180300510 tpc

```

通过 `pstree` 命令在进程树中找到 `bash` 的 `hello` 进程

```

  -gnome-shell-cal--5*[{gnome-shell-cal}]
  -gnome-terminal--bash
                    |
                    | bash
                    |   |
                    |   | hello
                    |   | pstree
                    |   |
                    |   | 3*[{gnome-terminal-}]

```

`fg` 命令可以让 `hello` 进程重新运行

```

tpc@tpc-virtual-machine:~/CSAPP$ fg 1
./hello 1180300510 tpc
Hello 1180300510 tpc
Hello 1180300510 tpc
Hello 1180300510 tpc
Hello 1180300510 tpc
Hello 1180300510 tpc
^Z
[1]+  已停止                  ./hello 1180300510 tpc

```

使用 `kill -9 PID` 杀死 `hello` 进程，在这里，`hello` 的 `PID` 为 11153

```
tpc@tpc-virtual-machine:~/CSAPP$ ps
  PID TTY          TIME CMD
 10288 pts/1        00:00:00 bash
 11153 pts/1        00:00:00 hello
 11163 pts/1        00:00:00 ps
tpc@tpc-virtual-machine:~/CSAPP$ kill -9 11153
tpc@tpc-virtual-machine:~/CSAPP$ ps
  PID TTY          TIME CMD
 10288 pts/1        00:00:00 bash
 11164 pts/1        00:00:00 ps
[1]+  已杀死                  ./hello 1180300510 tpc
```

6.7 本章小结

这一章介绍了 `hello` 可执行文件在进程中的执行过程。介绍了 `shell-bash` 的工作流程，`shell` 利用 `fork` 和 `execve` 运行 `hello` 程序的过程，用户模式和内核模式，上下文的切换。最后，通过在命令行的各种命令的演示，讲述了 `hello` 进程的异常处理和信号机制。

(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：由程序产生的段偏移地址。由段标识和段偏移量组成。以段标识为下标到 GDT/LDT 查表获得段地址。段地址+端偏移量=线性地址

线性地址：一个非负整数地址的有序集合，如果此时地址是连续的，则称这个空间为线性地址空间。

虚拟地址：在保护模式下，程序运行在虚拟内存中。虚拟内存被组织为一个由存放在磁盘上的 N 个连续的字节大小的单元组成的数组。每字节都有一个唯一的虚拟地址，作为到数组的索引。虚拟地址由 VPO(虚拟页面偏移量)、VPN(虚拟页号)、TLBI(TLB 索引)、TLBT(TLB 标记)组成。

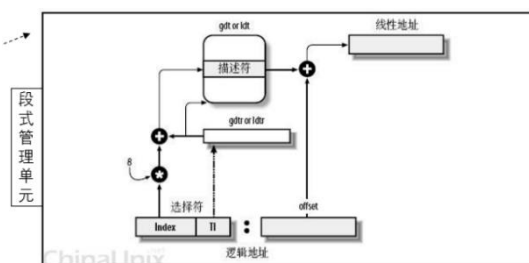
物理地址：计算机系统的主存被组织成一个由 M 个连续的字节大小的单元组成的数组，每字节都有一个唯一的物理地址，比如第一个字节地址为 0，第二个地址为 1，以此类推。物理地址空间对应于系统中物理内存的 M 个字节： $\{0, 1, 2, \dots, M-1\}$ 。

在 hello 中，main 函数的地址为 0x400532，这是逻辑地址中的段偏移量，加上段地址就是 main 函数的虚拟地址，虚拟地址与物理地址之间存在一种映射关系，MMU 利用页表实现这种映射，可得到实际的物理地址。

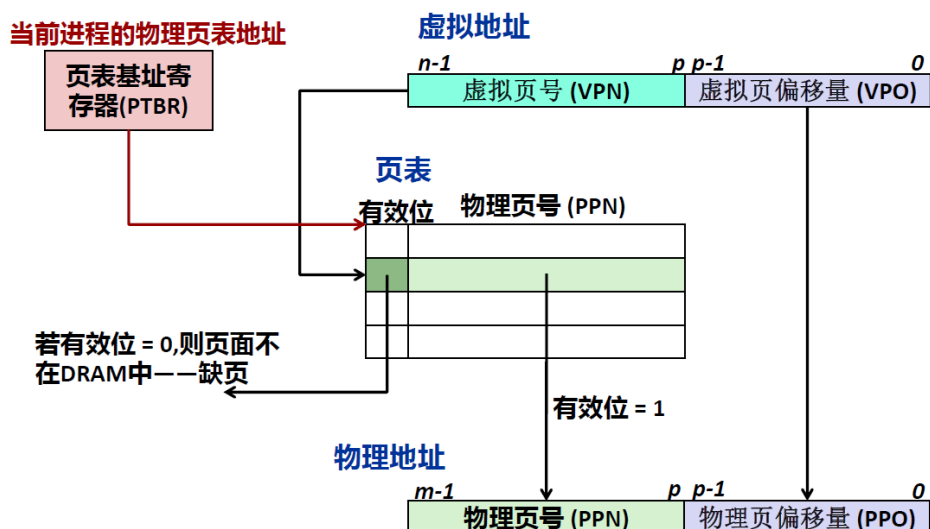
7.2 Intel 逻辑地址到线性地址的变换-段式管理

实模式：逻辑地址 $CS:EA = EA + 16 * CS$

保护模式：逻辑地址由段标识和段偏移量组成。以段标识为下标，去索引段描述符表，若 $T1=0$ ，索引全局段描述符表（GDT），若 $T1=1$ ，索引局部段描述符表（LDT）。将段描述符表中的段地址（base 字段）加上段偏移量，即为线性地址。



7.3 Hello 的线性地址到物理地址的变换-页式管理



hello 的线性地址到物理地址的变换，也就是从虚拟地址寻址物理地址，在虚拟地址和物理地址之间存在一种映射，MMU 通过页表实现这种映射。

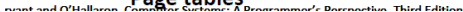
虚拟地址由虚拟页号（VPN）和虚拟页偏移量（VPO）组成，页表中由有效位和物理页号组成，VPN 作为到页表的索引，去页表中寻找相应的 PTE，其中 PTE 有三种情况，分别为已分配，未缓存，未分配。已分配表示已经将虚拟地址对应到物理地址，有效位为 1，物理页号不为空。未缓存表示还未将虚拟内容缓存到物理页表中，有效位为 0，物理页号不为空。未分配表示未建立映射关系，有效位为 0，物理页号为空。

如果有效位为 0，表示缺页，进行缺页处理，从磁盘读取物理页到内存，若有效位为 1，则可以查询到相对应的 PPN，物理页偏移量和 VPO 相同，PPN 和 PPO 组成物理地址。

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

为减少内存读取数据的次数，在 MMU 中包括了一个关于 PTE 的小的缓存，即 TLB，每一行都保存着一个由单个 PTE 组成的块。TLB 索引由 VPN 的 t 个最低位组成，剩余的为 TLB 标记。

CPU 产生一个虚拟地址，当 TLB 命中时，MMU 从 TLB 中取出相应的 PTE，MMU 将这个虚拟地址翻译成一个物理地址，并且将它发送到高速缓存/主存中，高速缓存/主存将所请求的数据字返回给 CPU。若 TLB 不命中，MMU 必须从页表中的 PTE 取出 PPN 复制到 PTE，而在得到 PTE 还会发生缺页或者是缓存不命中的情况。



7.7 hello 进程 execve 时的内存映射

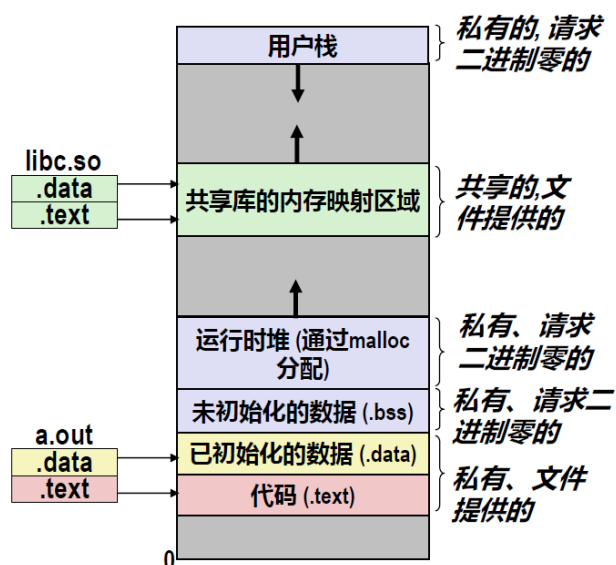
`execve` 函数在当前进程中加载并运行包含在可执行目标文件 `hello` 中的程序，用 `hello` 有效替代当前程序。加载并运行 `hello` 需要以下几个步骤：

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 `hello` 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` 段，`bss` 是请求二进制零的，映射到匿名文件，大小包含在 `hello` 中，栈和堆也是请求二进制零的，初始长度为 0。

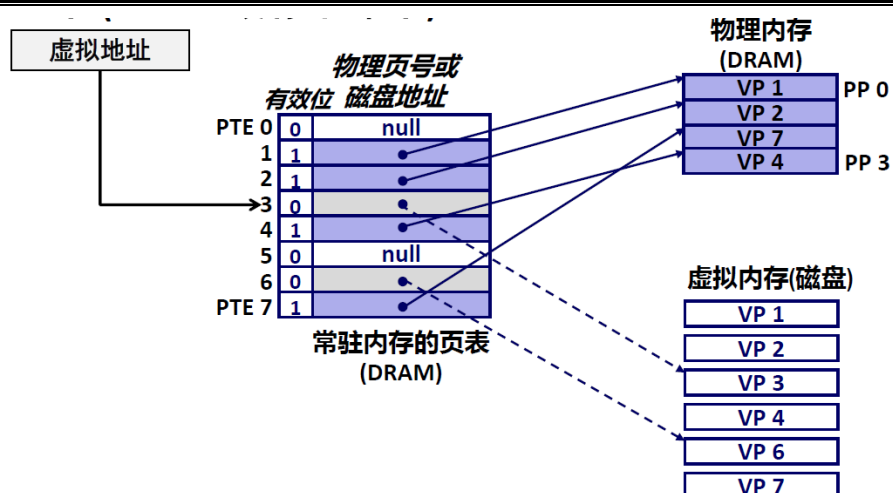
映射共享区域。如果 `hello` 程序与共享对象链接，如 `libc.so`，那么这些对象都是动态链接到这个程序的，然后再映射到用户虚拟地址空间中的共享区域内。

设置程序计数器。`execve` 做的最后一件事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。下一次调度这个进程时将从这个入口点开始执行。

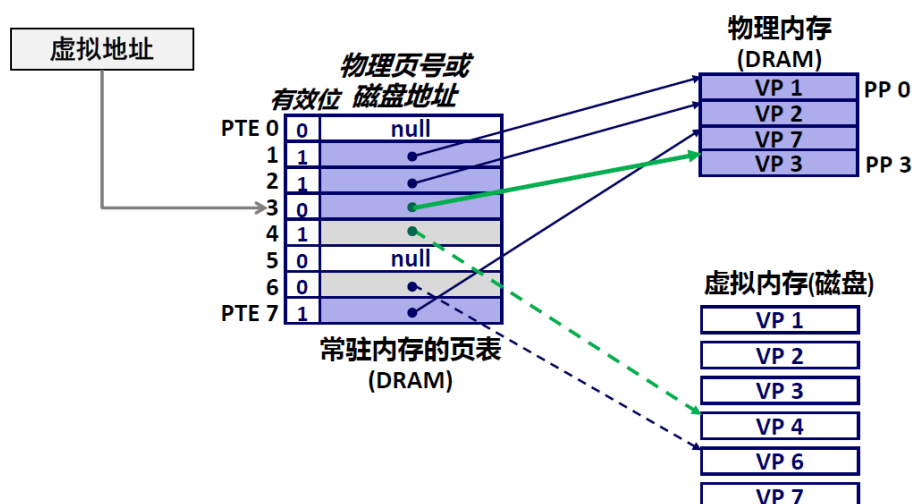


7.8 缺页故障与缺页中断处理

缺页是指 DRAM 缓存不命中。如下图，CPU 引用 `VP3` 中的一个字，`VP3` 并未缓存在 DRAM 中。地址翻译硬件从内存中读取 `PTE3`，从有效位推断 `VP3` 未被缓存，出发一个缺页异常。缺页异常调用内核中的缺页异常处理程序，该程序会选择一个牺牲页，在这里假设是 `VP4`。如果 `VP4` 已经被修改了，那么内核就会将它复制回磁盘。



接下来，内核从磁盘复制 VP3 到内存中的 PP 3，更新 PTE 3，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新发送到地址翻译硬件，现在 VP3 已经在缓存中了，那么页命中也可以正常处理了。



7.9 动态存储分配管理

动态内存分配器维护着一个进程的虚拟内存区域，称为堆，分配器将堆视为一组不同大小的块的集合来维护，每个块就是一个连续的虚拟内存片，要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可以用来分配。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。

动态内存分配器分为显式分配器和隐式分配器两种。

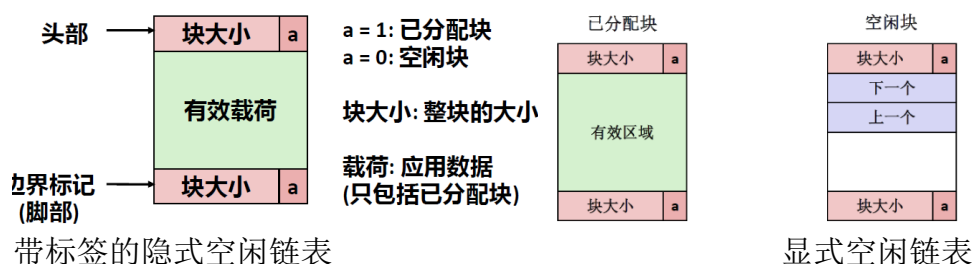
7.9.1、带标签的隐式空闲链表

将堆组织为一个连续的已分配块和空闲块的序列的结构是隐式空闲链表，空闲块通过头部的大小字段隐含地链接。而带边界标签的隐式空闲链表则在每个块的结尾处添加一个脚部——头部的副本，脚部总是在距当前块开始位置一个字的距离。分配器可以通过检查它的脚部，判断前面一个块的起始位置和状态。

当一个应用请求一个 k 字节的块时，分配器搜索空闲链表，查找一个足够大可以防止所请求块的空闲块。这种搜索方式由放置策略确定，包括首次适配、下一次适配和最佳适配。

一旦分配器找到匹配的空闲块之后，作出另一个决定——分配这个空闲块多少空间。通常选择将空闲块分割成两个部分，剩下的一部分变成新的空闲块。

当分配器释放一个已分配块之时，可能有其他空闲块与新释放的空闲块相邻，可能会导致假碎片问题，因此需要合并相邻的空闲块。而带有边界标签的隐式空闲链表分配器就可以在常数时间内完成对前面块的合并。简单来说，就是双向合并。



7.9.2、显式空闲链表

将空闲块组织成某种形式的显式数据结构，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。堆可以组织成一个双向空闲链表，在每个空闲块中，都包含一个前驱和后继指针。

使用双向链表可以使首次适配的分配时间减少到空闲块数量的线性时间，释放块的时间取决于选择的空闲链表中块的排序策略。

一种方法是后进先出（LIFO）的顺序维护链表，将新释放的块放置在链表的开始处，分配器会最先检查最近使用过的块。

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定位合适的前驱。

7.10 本章小结

这一章介绍了 hello 的存储器地址空间的概念和相关的地址计算方法，缺页和缺页处理，重点介绍了虚拟地址转换成物理地址的过程，包括四级页表、TLB 加

速、三级 cache 等。除此以外，介绍了内存映射，以及 fork 创建进程和 execve 函数运行 hello 时的具体过程。最后讲述了动态内存分配管理的不同结构的链表的操作。

(第 7 章 2 分)

第 8 章 hello 的 IO 管理

8.1 Linux 的 IO 设备管理方法

一个 Linux 文件就是一个 m 个字节的序列： $B_0, B_1, \dots, B_k, \dots, B_{m-1}$ ，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做对应文件的读和写来执行。这种将设备映射为文件的方式，允许 Linux 内核引出一个简单、低级的应用接口，称为 Unix I/O，这使得所有输入和输出都能以一种统一且一致的方式来执行。

设备的模型化：文件

设备管理：unix io 接口

8.2 简述 Unix IO 接口及其函数

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。内核返回一个叫做描述符的小的非负整数，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。

对应的函数为 `int open(char *filename, int flags, mode_t mode)`; `filename` 为文件名，`flags` 参数指明了进程打算如何访问这个文件，可以是只读、只写、可读可写。`mode` 参数指定了新文件的访问权限位。若 `open` 成功则返回新文件描述符，若失败则返回 -1。

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

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

读写文件：一个读操作就是从文件复制 n 个字节到内存，从当前文件位置 k 开始，然后 k 增加到 $k+n$ ，给定一个大小为 m 字节的文件，当 $k \geq m$ 时执行读操作会触发一个 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地 EOF 符号。

类似的，写操作就是从内存复制 n 个文件到一个文件，从当前文件位置 k 开始，更新 k 。

读文件对应的函数为 `ssize_t read(int fd, void *buf, size_t n)`; `fd` 为当前文件的描

述符, buf 是内存位置, n 是复制最多 n 个字节。若成功则为读的字节数, 若 EOF 则为 0, 若出错-1。

写文件对应的函数是 `ssize_t write(int fd, const void *buf, size_t n)`; 若成功则为写的字节数, 若出错则为-1。

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, 内核丢回关闭所有打开的文件并释放他们的内存资源。

关闭文件的函数为 `int close(int fd)`; 若成功为 0, 若出错为-1。关闭一个已经关闭的描述符会出错。

I/O 重定向: Linux shell 提供了 I/O 重定向操作符, 允许用户将磁盘文件和标准输入输出联系起来。

函数为 `int dup2(int oldfd, int newfd)`; `dup2` 函数复制描述符表象项 `oldfd` 到描述符表项 `newfd`, 覆盖 `newfd` 之前的内容, 如果 `newfd` 已经打开了, `dup2` 会在复制 `oldfd` 之前关闭 `newfd`。

8.3 printf 的实现分析

首先观察一下 Linux 下 `printf` 的函数体

```
static int printf(const char *fmt, ...)
{
    va_list args;
    int i;

    va_start(args, fmt);
    write(1, printbuf, i = vsprintf(printbuf, fmt, args));
    va_end(args);
    return i;
}
```

定义 `va_list` 型变量 `args`, 指向参数的指针。`va_start` 和 `va_end` 是获取可变长度参数的函数, 首先调用 `va_start` 函数初始化 `args` 指针, 通过对 `va_arg` 返回可变的参数, 然后 `va_end` 结束可变参数的获取。

重点需要看 `write` 函数和 `vsprintf` 函数。

`vsprintf` 函数的作用是以 `fmt` 为格式字符串, 根据 `args` 中的参数, 向 `printbuf` 输出格式化后的字符串。然后调用 `write` 函数, `write` 函数是 Unix I/O 函数, 用以在屏

幕上输出长度为 `i` 的在 `printfbuf` 处的内容。查看 `write` 函数的汇编代码可以看出它将栈中参数存入寄存器，然后执行 `INT_VECTOR_SYS_CALL`,代表通过系统调用 `syscall`，`syscall` 将寄存器中存储的字符串通过总线复制到显卡的内存中，字符显示驱动子程序通过 ASCII 码在字模库中找到点阵信息并将其存储到 `vram` 中。接下来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）。此时在屏幕上显示一个已经格式化的字符串。

8.4 getchar 的实现分析

当程序调用 `getchar` 函数，程序等待用户的按键。用户按键时，键盘接口会得到一个键盘扫描码，同时产生一个中断请求，进行上下文切换，运行键盘中断子程序，该程序将按键扫描码转换为 ASCII 码，保存到键盘缓存区。直到用户按回车为止，然后将用户输入的字符显示到屏幕。

`getchar` 的代码：

```
int getchar(void)
{
    static char buf[BUFSIZ];
    static char* bb=buf;
    static int n=0;
    if(n==0)
    {
        n=read(0,buf,BUFSIZ);
        bb=buf;
    }
    return (--n>=0)?(unsigned char)*bb++:EOF;
}
```

`getchar` 函数通过调用 `read` 函数来读取字符，`read` 函数的返回值是读入字符的个数，若出错则返回-1。`read` 函数通过调用内核中的系统函数，读取键盘缓冲区的 ASCII 码，直到读到回车为止，然后将整个字符串返回。

8.5 本章小结

这一章介绍了Linux的I/O管理方法、I/O接口及其函数，以及通过阅读 `printf` 函数和 `getchar` 函数的代码了解如何通过Unix I/O实现功能。总的说，Unix I/O使得所有的输入和输出都能以一种统一且一致的方式来执行。

（第8章1分）

结论

hello from program to process, from zero to zero, this is hello's life:

- 1、用户从键盘输入，得到 `hello.c` 的 C 源文件。
- 2、预处理器对 `hello.c` 进行预处理，得到 `hello.i`，编译器将 `hello.i` 翻译成汇编文件 `hello.s`，汇编器将 `hello.s` 翻译成机器指令得到 `hello.o` 可重定位文件。
- 3、链接器对 `hello.o` 进行链接，得到可执行文件 `hello`，此时，`hello` 已经可以被系统加载和运行。
- 4、在终端中输入命令，`shell-bash` 调用 `fork` 函数创建一个新的子进程，并在新的子进程中调用 `execve` 函数，加载并运行 `hello`。
- 5、`hello` 会与多个进程并行运行，当发生中断或异常时，发生上下文切换，转到内核模式，内核调度另一个进程运行。
- 6、`hello` 在运行过程中可能会遇到各种信号和键盘输入，`shell` 为其提供信号处理程序。
- 7、CPU 为 `hello` 分配内存空间，`hello` 从磁盘加载到内存。
- 8、`hello` 访存时，请求一个虚拟地址，通过 MMU、TLB、四级页表得到虚拟地址对应的物理地址，在三级 cache 中进行访存。
- 9、`hello` 输出信息调用 `printf` 函数和 `getchar` 函数，这两个函数需要调用 Unix I/O 接口函数实现。
- 10、`hello` 执行 `return 0`，结束进程，`bash` 会回收 `hello` 子进程。

我的感悟：

一个简单的 `hello.c` 要真正变成可执行文件被系统加载和运行，需要有软件和硬件的共同支持，也需要经过大量的流程。`hello.c` 变成 `hello` 就需要预处理器、编译器、汇编器、链接器等。而让 `hello` 真正运行更是需要 `shell` 的函数调用、信号处理、并发处理、访存机制等，在 `hello` 运行过程中，进程管理、存储管理更是起到了重要的作用。可以说，一个程序的运行依靠系统各个组成部分的协调运行，缺一不可。

`hello` 可以说囊括了这一个月学习 CSAPP 的大部分内容，用一个大作业的形式将这些内容很好的串联在一起。

(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。

中间文件	作用
hello.i	预处理产生的文件
hello.s	将预处理文件变为汇编文件
hello.o	汇编器将汇编文件变为可重定位文件
hello	ELF 格式的可执行文件
hello l.d	hello 的反汇编文件，查看链接
hello .d	hello 的反汇编文件，查看汇编代码

(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] 深入理解计算机系统
- [2] printf 函数实现的深入剖析 <https://www.cnblogs.com/pianist/p/3315801.html>
- [3] Linux 逻辑地址、线性地址、虚拟地址、物理地址
https://blog.csdn.net/baidu_35679960/article/details/80463445

(参考文献 0 分，缺失 -1 分)