

哈爾濱工業大學

計算機系統

大作業

題 目 程序人生-Hello's P2P

專 業 計算機科學與技術

學 號 1170300901

班 級 1703009

學 生 侯欣宇

指 導 教 師 史先俊

計算機科學與技術學院

2018 年 12 月

摘 要

本文通过一个简单的例程 `hello.c`，介绍了 Linux 下 `hello` 从 C 语言文件到可执行程序、从程序到进程再到被回收的全过程。本文编写过程中用到了 `edb`、`gcc`、`objdump`、`readelf` 等工具。

关键词：`hello`；计算机系统；编译系统

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	- 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	- 12 -
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	- 23 -
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	- 29 -
第 7 章 HELLO 的存储管理.....	- 30 -
7.1 HELLO 的存储器地址空间	- 30 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理.....	- 30 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理	- 30 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换.....	- 31 -
7.5 三级 CACHE 支持下的物理内存访问	- 33 -
7.6 HELLO 进程 FORK 时的内存映射	- 34 -
7.7 HELLO 进程 EXECVE 时的内存映射	- 34 -
7.8 缺页故障与缺页中断处理.....	- 35 -
7.9 动态存储分配管理	- 35 -
7.10 本章小结	- 36 -
第 8 章 HELLO 的 IO 管理	- 37 -
8.1 LINUX 的 IO 设备管理方法	- 37 -
8.2 简述 UNIX IO 接口及其函数	- 37 -
8.3 PRINTF 的实现分析.....	- 38 -
8.4 GETCHAR 的实现分析.....	- 38 -
8.5 本章小结	- 39 -
结论	- 39 -
附件	- 40 -
参考文献.....	- 41 -

第 1 章 概述

1.1 Hello 简介

P2P: From Program to Process

使用 C 语言编写的到 `hello.c` 代码文件, C 语言是高级语言, 所以这个形式的代码能让人读懂, 但是系统不认识, 为了让系统能够读懂代码, 需要将 `hello.c` 转化成一系列机器能够读懂的语言指令, 然后将这些指令按照一种称为可执行目标程序的格式进行打包, 并将以二进制磁盘文件形式存放, 目标程序也可以称为执行文件。

使用 GCC 编译器编译解析 `hello.c`, 依次经历预处理阶段, 编译阶段, 汇编阶段, 链接阶段(这几个阶段将会在下面论述中详细展开), 最后生成了可执行目标文件 `hello`。

在 shell 中建立 `./hello` 的命令后, shell 将自动为起 `fork` 一个进程, 这就实现了 P2P。

O2O: From Zero-0 to Zero -0

在执行 `hello` 这个目标文件中, 系统 `fork` 了一个子进程。之后 `execve` 函数加载进程, 创建新的内存区域以及西南的数据、堆、栈等, 映射虚拟内存, 进入程序入口后程序开始加载物理内存, 然后从 `main` 函数执行目标代码, CPU 为 `hello` 分配时间片执行逻辑控制流。`Hello` 通过 I/O 管理来控制设备的输入和输出, 实现软硬件结合。当整个程序运行完成之后, 进程结束, 父进程回收结束的子进程, 防止资源的浪费, 实现 O2O。

1.2 环境与工具

硬件环境: Intel Core i7-7700HQ x64CPU, 16G RAM, 128G SSD + 1T HDD.

软件环境: Ubuntu 18.04.1 LTS

开发与调试工具: `edb`, `cb`, `vim`, `gcc`, `as`, `ld`, `readelf`, `HexEdit`

1.3 中间结果

文件作用

`hello.i` 预处理之后文本文件

`hello.s` 编译之后的汇编文件

hello.o	汇编之后的可重定位目标执行
hello	链接之后的可执行目标文件
hello.elf	hello.o 的 ELF 格式
hello1	hello 的 ELF 格式

1.4 本章小结

本章介绍了 `hello` 从编译生成、到执行、再到终止的全过程，从整体上大致介绍了 `hello` 的一生，并且列出了做本次作业的软硬件环境以及工具，最后列出了本次作业从 `hello.c` 到 `hello` 的过程中产生的中间文件。

第 2 章 预处理

2.1 预处理的概念与作用

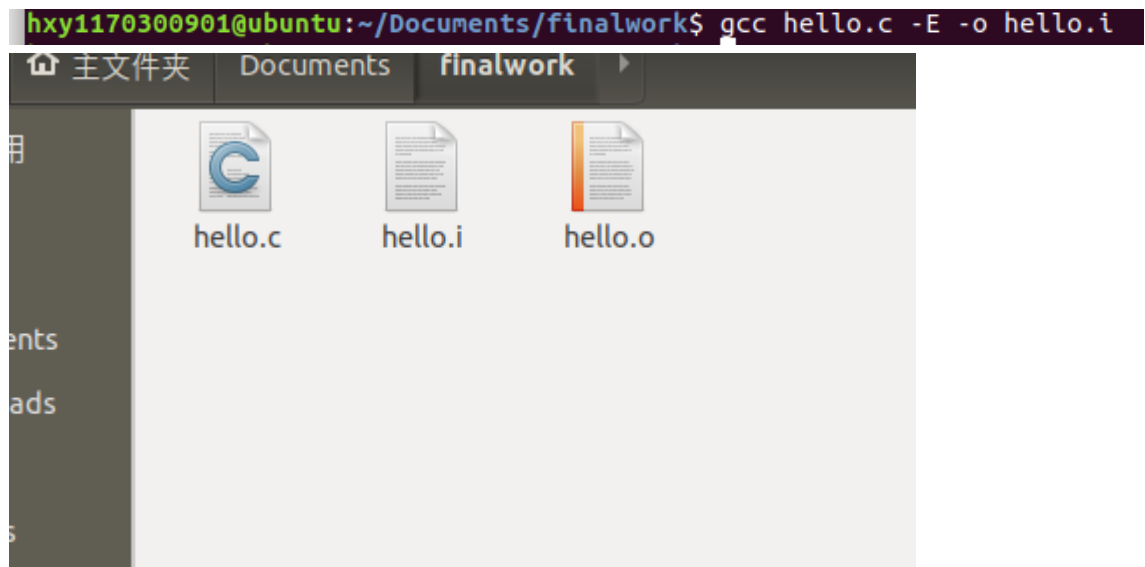
概念: 预处理器(cpp)主要处理根据以字符#开头的命令,修改原始的 C 程序。比如 hello.c 中第一行的 `#include <stdio.h>` 命令会告诉预处理其读区系统头文件 `stdio.h` 的内容,并把它直接插入到程序文本中。此过程,会得到以.i 作为扩展名。

作用:

- ①加载头文件
- ②进行宏替换
- ③条件编译

2.2 在 Ubuntu 下预处理的命令

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



2.2.1hello 预处理

2.3 Hello 的预处理结果解析

```
int main(int argc, char *argv[])
{
    int i;

    if(argc != 3)
    {
        printf("Usage: Hello 学号 姓名! \n");
        exit(1);
    }
    for(i=0; i<10; i++)
    {
        printf("Hello %s %s\n", argv[1], argv[2]);
        sleep(sleepsecs);
    }
    getchar();
    return 0;
}
```

2.3.1 hello.i

在 `hello.c` 中只存在库函数调用需要进行预处理，预处理将库函数直接插入到原代码上面。

2.4 本章小结

Hello.C 变为 `hello.i` 的过程就是将各种宏引入到函数的内容中，这和你引入多少的头文件（各种宏）有关。

第 3 章 编译

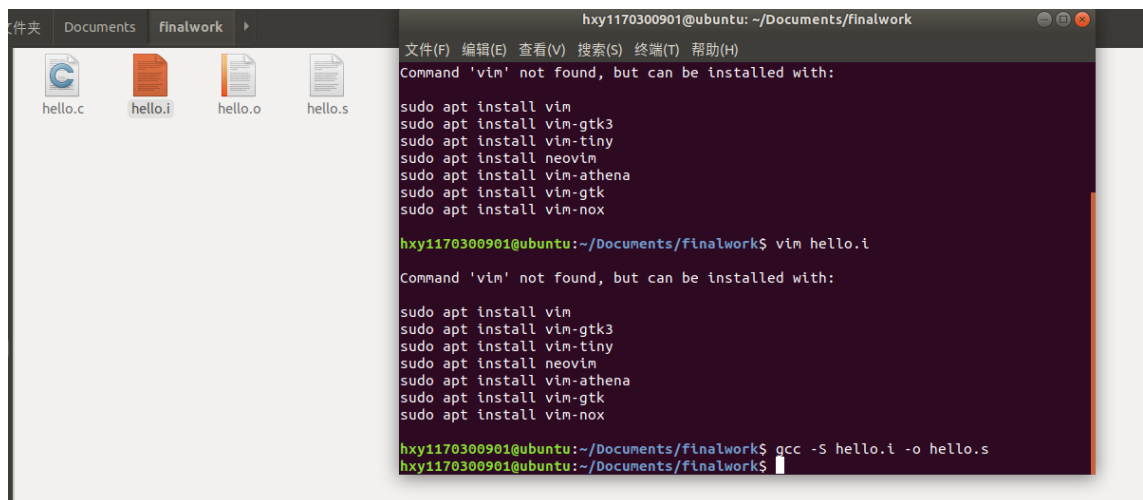
3.1 编译的概念与作用

概念：编译器(ccl)将 hello.i 翻译成文件 hello.s 文件，它包含一个汇编语言程序,汇编程序中的每条语句都以一种标准的文本格式确切地描述了一条条低级机器语言指令.所以该过程会检查代码规范，语法,词法分析,具体如下图.只有编译成功之后，才能生成具体的汇编代码。

作用：

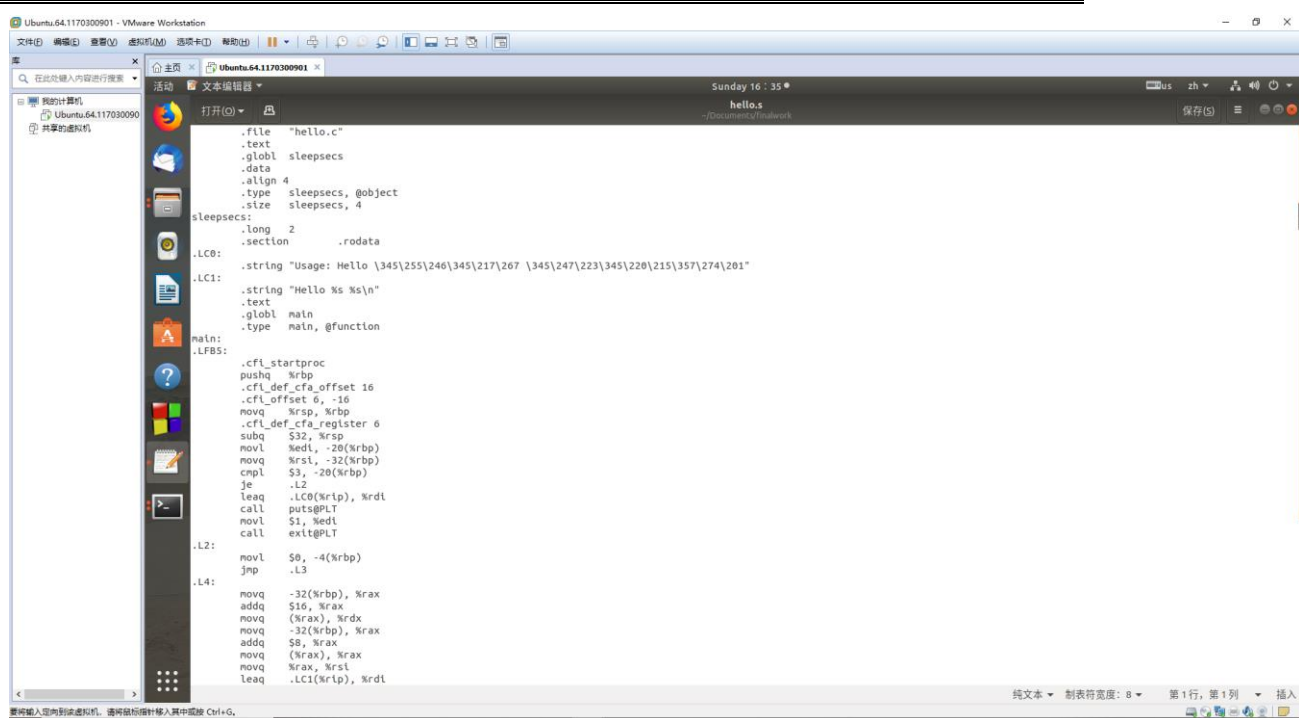
- ①将每条语句描述成一条条低级机器语言指令
- ②检查代码规范

3.2 在 Ubuntu 下编译的命令



3.2.1hello.i 编译生成 hello.s

3.3 Hello 的编译结果解析



3.3.1hello.s

3.3.1 数据

有整数，字符串，数组。

字符串：

1. “Usage: Hello 学号 姓名！\n”，可以发现字符串被编码成 UTF-8 格式，一个汉字在 utf-8 编码中占三个字节，一个\代表一个字节。
2. “Hello %s %s\n”，第二个 printf 传入的输出格式化参数。

```

.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

```

3.3.1.1 hello.s 中声明在.LC0 和.LC1 段中的字符串

整数：

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

```

.file    hello.c
.text
.globl   sleepsecs
.data
.align 4
.type    sleepsecs, @object
.size    sleepsecs, 4
sleepsecs:
.long    2

```

3.3.1.2 hello.s 中声明的全局变量 sleepsecs

2. **int i:** 编译器将局部变量存储在寄存器或者栈空间中，在 hello.s 中编译器将 i 存储在栈上空间-4(%rbp)中，可以看出 i 占据了栈中的 4B。

3. **int argc:** 第一个参数传入。

4. **立即数:** 其他整形数据的出现都是以立即数的形式出现的，直接硬编码在汇编代码中。

数组:

程序中涉及数组的是: `char *argv[]` main, 函数执行时输入的命令行, argv 作为存放 char 指针的数组同时是第二个参数传入。

3.3.2 赋值与类型转换

这两点一起说的原因是他们两个操作出现在同一个操作中: `Int sleepsecs = 2.5` (单股赋值操作还有个 `i=0`, 因为同样可以概括到, 这里不加讨论)

赋值操作在汇编语言中是以 `mov` 的形式来体现的, 例如对 `int` 型进行的赋值操作, 一般用 `movl`

而对于第一个赋值操作, 还涉及到强制类型转换, 将浮点型数据转换为 `int` 性数据数据转换时采用值向 0 舍入原则, 所以此时 2.5 会被舍入为 2。

3.3.3 算术操作

加法: <code>x=x+y</code>	<code>addq y,x</code>
减法: <code>x=x-y</code>	<code>subq y,x</code>
乘法: <code>x=x*y</code>	<code>imulq y,x</code>
除法: <code>z=x/y</code>	<code>movq x,z</code> <code>Cqto</code> <code>idivq y</code>

3.3.5 关系操作与控制转移

指令	效果	描述
<code>CMP S1,S2</code>	<code>S2-S1</code>	比较-设置条件码
<code>TEST S1,S2</code>	<code>S1&S2</code>	测试-设置条件码
<code>SET** D</code>	<code>D=**</code>	按照**将条件码设置D

3.3.5.1 关系操作的汇编指令

jX指令	条件	描述
jmp	1	无条件
je	ZF	相等 / 结果为0
jne	~ZF	不相等 / 结果不为0
js	SF	结果为负数
jns	~SF	结果为非负数
jg	~(SF^OF)&~ZF	大于 (符号数)
jge	~(SF^OF)	大于等于 (符号数)
jl	(SF^OF)	小于 (符号数)
jle	(SF^OF) ZF	小于等于 (符号数)
ja	~CF&~ZF	大于 (无符号数)
jb	CF	小于 (无符号数)

3.3.5.2 控制转移的汇编指令

1. `argc!=3`: 判断 `argc` 不等于 3。hello.s 中使用 `cmpl $3,-20(%rbp)`, 计算 `argc-3` 然后设置条件码, 为下一步 `je` 利用条件码进行跳转作准备。
2. `i<10`: 判断 `i` 小于 10。hello.s 中使用 `cmpl $9,-4(%rbp)`, 计算 `i-9` 然后设置条件码, 为下一步 `jle` 利用条件码进行跳转做准备。

3.3.6 函数操作

返回值: 一个函数的返回值一般存在寄存器 `eax` 中, 如果要设定返回值的话, 那就先将返回值传入 `eax`, 然后再用 `ret` 语句返回。以 `hello.c` 为例子, 具体操作如下:

```
movl    $0, %eax
leave
.cfi_def_cfa 7, 8
ret
.cfi_endproc
```

函数调用及参数传递: 如果你需要调用一个函数并且向其中传入参数的话, 你需要先找几个寄存器, 将参数传给这些寄存器, 选择哪些寄存器要看你调用的函数的具体实现, 然后再执行一个 `call` 跳转语句, 跳转到你想要调用的函数的开头位置, 此时程序就会从那个位置开始继续执行。具体操作如下:

```
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
```

(printf(“Hello %s %s\n”,argv[1],argv[2])操作，其中的 rdx,rsi,edi 这些都是参数)

3.4 本章小结

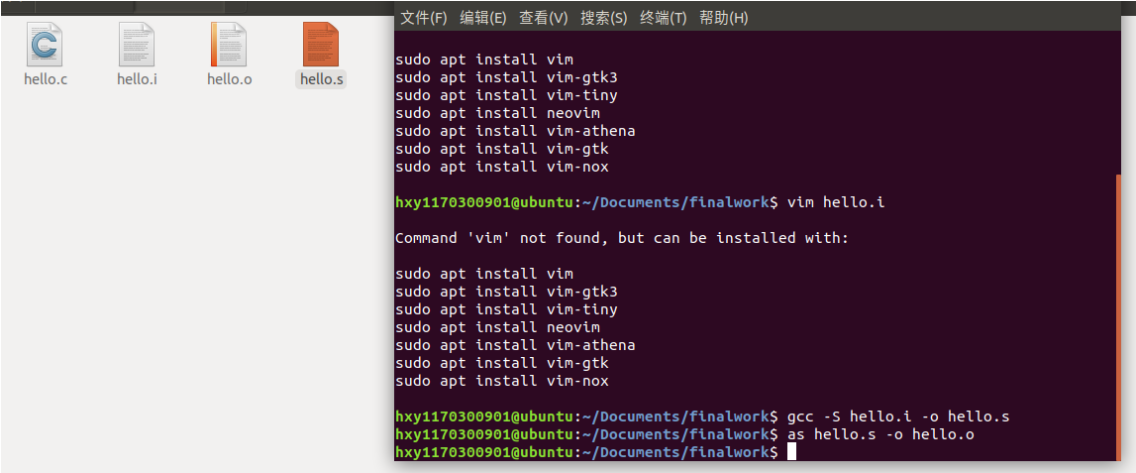
本章简述了编译的概念和作用，并用 `hello` 这个程序做了示范，具体的话就是 `hello.i` 到 `hello.s` 的过程，相关操作都是由汇编代码表示的。

第 4 章 汇编

4.1 汇编的概念与作用

汇编的概念是指的将汇编语言(`xxx.s`)翻译成机器指令,并将这些指令打包成一种叫做可重定位目标程序,并将这个结果保留在(`xxx.o`)中。这里的 `xxx.o` 是二进制文件。汇编过程的作用是将汇编指令转换成一条条机器可以直接读取分析的机器指令。

4.2 在 Ubuntu 下汇编的命令



```
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

hello.c  hello.i  hello.o  hello.s

hxy1170300901@ubuntu:~/Documents/finalwork$ sudo apt install vim
hxy1170300901@ubuntu:~/Documents/finalwork$ sudo apt install vim-gtk3
hxy1170300901@ubuntu:~/Documents/finalwork$ sudo apt install vim-tiny
hxy1170300901@ubuntu:~/Documents/finalwork$ sudo apt install neovim
hxy1170300901@ubuntu:~/Documents/finalwork$ sudo apt install vim-athena
hxy1170300901@ubuntu:~/Documents/finalwork$ sudo apt install vim-gtk
hxy1170300901@ubuntu:~/Documents/finalwork$ sudo apt install vim-nox

hxy1170300901@ubuntu:~/Documents/finalwork$ vim hello.i

Command 'vim' not found, but can be installed with:

sudo apt install vim
sudo apt install vim-gtk3
sudo apt install vim-tiny
sudo apt install neovim
sudo apt install vim-athena
sudo apt install vim-gtk
sudo apt install vim-nox

hxy1170300901@ubuntu:~/Documents/finalwork$ gcc -S hello.i -o hello.s
hxy1170300901@ubuntu:~/Documents/finalwork$ as hello.s -o hello.o
hxy1170300901@ubuntu:~/Documents/finalwork$
```

4.2.1 使用 `as` 指令生成 `hello.o` 文件

4.3 可重定位目标 `elf` 格式

```

hxy1170300901@ubuntu:~/Documents/finalwork$ readelf -a hello.o
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:      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

```

4.3.1 elf 头

ELF 头:: 用于总的描述 ELF 文件各个信息的段。

```

节头:
[号] 名称      类型      地址      偏移量
     大小      全体大小  旗标  链接  信息  对齐
[ 0]                NULL      0000000000000000 00000000
     0000000000000000 0000000000000000 0 0 0
[ 1] .text      PROGBITS 0000000000000000 00000040
     00000000000000081 0000000000000000 AX 0 0 1
[ 2] .rela.text RELA      0000000000000000 00000340
     000000000000000c0 0000000000000018 I 10 1 8
[ 3] .data      PROGBITS 0000000000000000 000000c4
     00000000000000004 0000000000000000 WA 0 0 4
[ 4] .bss       NOBITS   0000000000000000 000000c8
     00000000000000000 0000000000000000 WA 0 0 1
[ 5] .rodata    PROGBITS 0000000000000000 000000c8
     0000000000000002b 0000000000000000 A 0 0 1
[ 6] .comment   PROGBITS 0000000000000000 000000f3
     0000000000000002b 0000000000000001 MS 0 0 1
[ 7] .note.GNU-stack PROGBITS 0000000000000000 0000011e
     00000000000000000 0000000000000000 0 0 1
[ 8] .eh_frame   PROGBITS 0000000000000000 00000120
     00000000000000038 0000000000000000 A 0 0 8
[ 9] .rela.eh_frame RELA      0000000000000000 00000400
     00000000000000018 0000000000000018 I 10 8 8
[10] .symtab     SYMTAB   0000000000000000 00000158
     00000000000000198 0000000000000018 11 9 8
[11] .strtab     STRTAB   0000000000000000 000002f0
     0000000000000004d 0000000000000000 0 0 1
[12] .shstrtab   STRTAB   0000000000000000 00000418
     00000000000000061 0000000000000000 0 0 1

```

4.3.2 节头

节头: 描述了.o 文件中出现的各个节的类型、位置、所占空间大小等信息

```

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

```

4.3.3 重定位节

重定位节：这个节包含了.text（具体指令）节中需要进行重定位的信息。这些信息描述的位置，在由.o 文件生成可执行文件的时候需要被修改（重定位）。在这个hello.o 里面需要被重定位的有 printf , puts , exit , sleepsecs , getchar , sleep , rodata 里面的两个元素（.L0 和.L1 字符串）

.rela.eh_frame : eh_frame 节的重定位信息。

.symtab: 符号表，用来存放程序中定义和引用的函数和全局变量的信息。重定位需要引用的符号都在其中声明。

4.4 Hello.o 的结果解析


```

.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
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
movl    sleepsecs(%rip), %eax
movl    %eax, %edi
call    sleep@PLT
addl    $1, -4(%rbp)
.L3:
cmpl    $9, -4(%rbp)
jle     .L4
call    getchar@PLT
movl    $0, %eax
leave
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE5:
.size   main, .-main
.ident  "GCC: (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0"
.section .note.GNU-stack,"",@progbits

```

4.4.1hello.s

```

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

```

4.4.2 反汇编代码

1. 分支转移：反汇编用的不是.L3 这种的段名称，而是具体地址。
2. 函数调用：在.s 文件中，函数调用之后直接跟着函数名称，而在反汇编程序中，call 的目标地址是当前下一条指令。（涉及到链接与重定位）
3. 全局变量访问：在.s 文件中，访问 rodata（printf 中的字符串），使用段名称+%rip，在反汇编代码中 0+%rip，因为 rodata 中数据地址也是在运行时确定，故访问也需要重定位。所以在汇编成为机器语言时，将操作数设置为全 0 并添加重定位条目。

4.5 本章小结

本章介绍了 hello.s 到 hell.o 的汇编过程，了解到汇编语言和机器语言的不同。

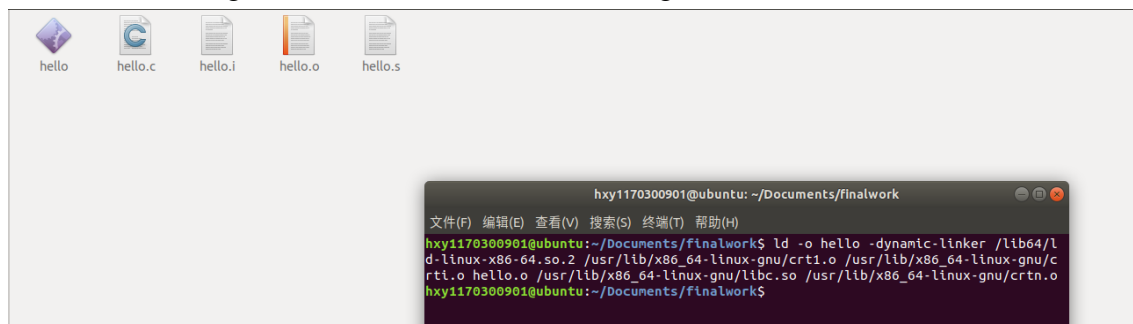
第 5 章 链接

5.1 链接的概念与作用

链接程序将分别在不同的目标文件中编译或汇编的代码收集到一个可直接执行的文件中。它还连接目标程序和用于标准库函数的代码，以及连接目标程序和由计算机的操作系统提供的资源（例如，存储分配程序及输入与输出设备）。链接工作大致包含两个步骤，一是符号解析，二是重定位。在符号解析步骤中，链接器将每个符号引用与一个确定的符号定义关联起来。将多个单独的代码节和数据节合并为单个节。将符号从它们的在.o 文件的相对位置重新定位到可执行文件的最终绝对内存位置。更新所有对这些符号的引用来反映它们的新位置。

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
```



5.2.1 链接指令

5.3 可执行目标文件 hello 的格式

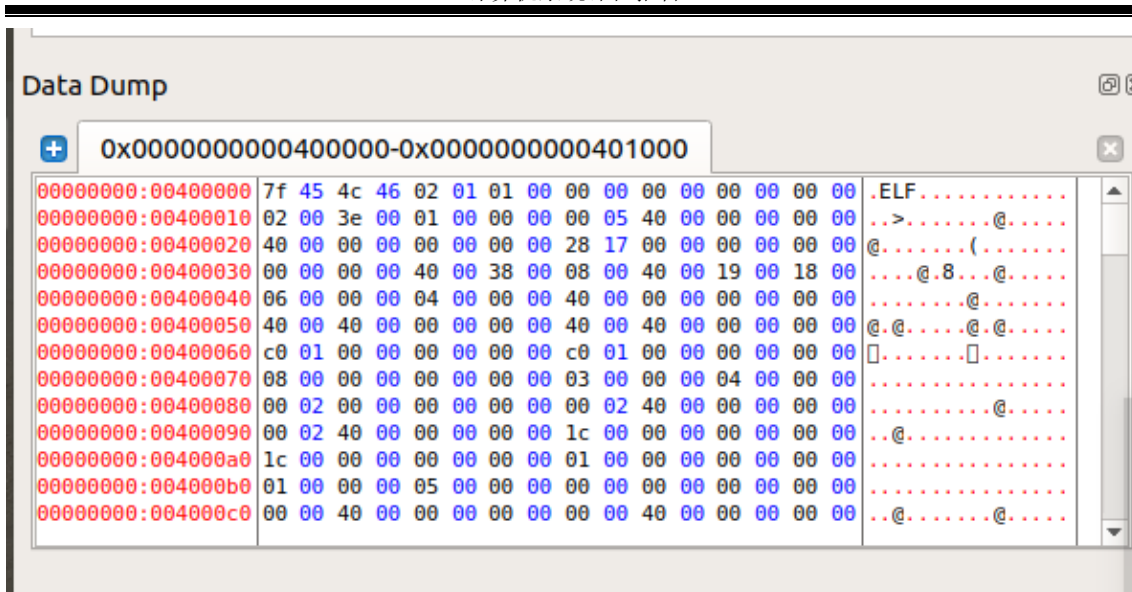
在 ELF 格式文件中，节头对 `hello` 中所有的节信息进行了声明，其中包括大小以及在程序中的偏移量，因此根据节头中的信息我们就可以用 HexEdit 定位各个节所占的区间（起始位置，大小）。其中地址是程序被载入到虚拟地址的起始地址。

节头:

[号]	名称	类型	地址		偏移量	
	大小	全体大小	旗标	链接	信息	对齐
[0]	0000000000000000	NULL	0000000000000000	0	0	0
[1]	.interp 000000000000001c	PROGBITS	0000000000400200	A	0	1
[2]	.note.ABI-tag 0000000000000020	NOTE	000000000040021c	A	0	4
[3]	.hash 0000000000000034	HASH	0000000000400240	A	5	8
[4]	.gnu.hash 000000000000001c	GNU_HASH	0000000000400278	A	5	8
[5]	.dynsym 00000000000000c0	DYNSYM	0000000000400298	A	6	1
[6]	.dynstr 0000000000000057	STRTAB	0000000000400358	A	0	1
[7]	.gnu.version 0000000000000010	VERSYM	00000000004003b0	A	5	2
[8]	.gnu.version_r 0000000000000020	VERNEED	00000000004003c0	A	6	1
[9]	.rela.dyn 0000000000000030	RELA	00000000004003e0	A	5	8
[10]	.rela.plt 0000000000000078	RELA	0000000000400410	AI	5	19
[11]	.init 0000000000000017	PROGBITS	0000000000400488	AX	0	4
[12]	.plt 0000000000000060	PROGBITS	00000000004004a0	AX	0	16
[13]	.text 0000000000000132	PROGBITS	0000000000400500	AX	0	16
[14]	.fini 0000000000000009	PROGBITS	0000000000400634	AX	0	4
[15]	.rodata 000000000000002f	PROGBITS	0000000000400640	A	0	4
[16]	.eh_frame 00000000000000fc	PROGBITS	0000000000400670	A	0	8
[17]	.dynamic 00000000000001a0	DYNAMIC	0000000000600e50	WA	6	8
[18]	.got 0000000000000010	PROGBITS	0000000000600ff0	WA	0	8
[19]	.got.plt 0000000000000040	PROGBITS	0000000000601000	WA	0	8

5.3.1hello 的节头

5.4 hello 的虚拟地址空间



5.4.1 hello 的虚拟地址空间

可以看出程序是在 0x00400000 地址开始加载的, 结束的地址大约是 0x00400fff。

程序头:

Type	Offset FileSiz	VirtAddr MemSiz	PhysAddr Flags Align
PHDR	0x0000000000000040 0x00000000000001c0	0x0000000000400040 0x00000000000001c0	0x0000000000400040 R 0x8
INTERP	0x0000000000000200 0x000000000000001c	0x0000000000400200 0x000000000000001c	0x0000000000400200 R 0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]			
LOAD	0x0000000000000000 0x000000000000076c	0x0000000000400000 0x000000000000076c	0x0000000000400000 R E 0x200000
LOAD	0x0000000000000e50 0x00000000000001f8	0x0000000000600e50 0x00000000000001f8	0x0000000000600e50 RW 0x200000
DYNAMIC	0x0000000000000e50 0x00000000000001a0	0x0000000000600e50 0x00000000000001a0	0x0000000000600e50 RW 0x8
NOTE	0x000000000000021c 0x0000000000000020	0x000000000040021c 0x0000000000000020	0x000000000040021c R 0x4
GNU_STACK	0x0000000000000000 0x0000000000000000	0x0000000000000000 0x0000000000000000	0x0000000000000000 RW 0x10
GNU_RELRO	0x0000000000000e50 0x00000000000001b0	0x0000000000600e50 0x00000000000001b0	0x0000000000600e50 R 0x1

5.4.2 hello ELF 中的程序头

PHDR: 程序头表

INTERP: 程序执行前需要调用的解释器

LOAD: 程序目标代码和常量信息

DYNAMIC: 动态链接器所使用的信息

NOTE:: 辅助信息

GNU_EH_FRAME: 保存异常信息

GNU_STACK: 使用系统栈所需要的权限信息

GNU_RELRO: 保存在重定位之后只读信息的位置

5.5 链接的重定位过程分析

```

hello:    文件格式 elf64-x86-64

Disassembly of section .init:

000000000400488 <.init>:
400488: 48 83 ec 08      sub    $0x8,%rsp
40048c: 48 8b 05 65 20 00 mov     0x200b65(%rip),%rax      # 600ff8 <__gmon_start__>
400493: 48 85 c0         test   %rax,%rax
400496: 74 02          je     40049a <_init+0x12>
400498: ff d0         callq  *%rax
40049a: 48 83 c4 08      add     $0x8,%rsp
40049e: c3             retq

Disassembly of section .plt:

0000000004004a0 <.plt>:
4004a0: ff 35 62 0b 20 00 pushq   0x200b62(%rip)          # 601008 <_GLOBAL_OFFSET_TABLE_+0x8>
4004a6: ff 25 64 0b 20 00 jmpq     *0x200b64(%rip)        # 601010 <_GLOBAL_OFFSET_TABLE_+0x10>
4004ac: 0f 1f 40 00      nopl    0x0(%rax)

0000000004004b0 <puts@plt>:
4004b0: ff 25 62 0b 20 00 jmpq     *0x200b62(%rip)        # 601018 <puts@GLIBC_2.2.5>
4004b6: 68 00 00 00 00 00 pushq   $0x0
4004bb: e9 e0 ff ff ff  jmpq     4004a0 <_.plt>

0000000004004c0 <printf@plt>:
4004c0: ff 25 5a 0b 20 00 jmpq     *0x200b5a(%rip)        # 601020 <printf@GLIBC_2.2.5>
4004c6: 68 01 00 00 00 00 pushq   $0x1
4004cb: e9 d0 ff ff ff  jmpq     4004a0 <_.plt>

0000000004004d0 <getchar@plt>:
4004d0: ff 25 52 0b 20 00 jmpq     *0x200b52(%rip)        # 601028 <getchar@GLIBC_2.2.5>
4004d6: 68 02 00 00 00 00 pushq   $0x2
4004db: e9 c0 ff ff ff  jmpq     4004a0 <_.plt>

0000000004004e0 <exit@plt>:
4004e0: ff 25 4a 0b 20 00 jmpq     *0x200b4a(%rip)        # 601030 <exit@GLIBC_2.2.5>
4004e6: 68 03 00 00 00 00 pushq   $0x3
4004eb: e9 b0 ff ff ff  jmpq     4004a0 <_.plt>

```

5.5.1 Hello 的反汇编代码

1.函数个数：在使用 ld 命令链接的时候，指定了动态链接器为 64 的 /lib64/ld-linux-x86-64.so.2，crt1.o、crti.o、crtn.o 中主要定义了程序入口_start、初始化函数_init，_start 程序调用 hello.c 中的 main 函数，libc.so 是动态链接共享库，其中定义了 hello.c 中用到的 printf、sleep、getchar、exit 函数和_start 中调用的 __libc_csu_init，__libc_csu_fini，__libc_start_main。链接器将上述函数加入。

2.函数调用：链接器解析重定条目时发现对外部函数调用的类型为 R_X86_64_PLT32 的重定位，此时动态链接库中的函数已经加入到了 PLT 中，.text 与.plt 节相对距离已经确定，链接器计算相对距离，将对动态链接库中函数的调用值改为 PLT 中相应函数与下条指令的相对地址，指向对应函数。对于此类重定位链接器为其构造.plt 与.got.plt。

3..rodata 引用：链接器解析重定条目时发现两个类型为 R_X86_64_PC32 的对.rodata 的重定位（printf 中的两个字符串），.rodata 与.text 节之间的相对距离确定，因此链接器直接修改 call 之后的值为目标地址与下一条指令的地址之差，指向相应的字符串。

5.6 hello 的执行流程

00007f80:65df2c30	48 89 e7	movq %rsp, %rdi
00007f80:65df2c33	e8 78 0d 00 00	callq ld-2.23.so!_dl_start
● 00007f80:65df2c38	49 89 c4	movq %rax, %r12
00007f80:65df2c3b	8b 05 37 50 22 00	movl 0x225037(%rip), %eax
00007f80:65df2c41	5a	popq %rdx
00007f80:65df2c42	48 8d 24 c4	leaq (%rsp, %rax, 8), %rsp
00007f80:65df2c46	29 c2	subl %eax, %edx
00007f80:65df2c48	52	pushq %rdx
00007f80:65df2c49	48 89 d6	movq %rdx, %rsi
00007f80:65df2c4c	49 89 e5	movq %rsp, %r13
00007f80:65df2c4f	48 83 e4 f0	andq \$0xfffffffff0, %rsp
00007f80:65df2c53	48 8b 3d e6 53 22 00	movq 0x2253e6(%rip), %rdi
00007f80:65df2c5a	49 8d 4c d5 10	leaq 0x10(%r13, %rdx, 8), %rcx
00007f80:65df2c5f	49 8d 55 08	leaq 8(%r13), %rdx
00007f80:65df2c63	31 ed	xorl %ebp, %ebp
00007f80:65df2c65	e8 d6 fa 00 00	callq ld-2.23.so!_dl_init
● 00007f80:65df2c6a	48 8d 15 3f fe 00 00	leaq 0xfe3f(%rip), %rdx
00007f80:65df2c71	4c 89 ec	movq %r13, %rsp
➔ 00007f80:65df2c74	41 ff e4	jmpq *%r12
00007f80:65df2c77	66 0f 1f 84 00 00 00 00..	nopw (%rax, %rax)
00007f80:65df2c80	48 8d 05 39 63 22 00	leaq 0x226339(%rip), %rax
00007f80:65df2c87	c3	retq
00007f80:65df2c88	0f 1f 84 00 00 00 00 00	nopl (%rax, %rax)
00007f80:65df2c90	83 47 04 01	addl \$1, 4(%rdi)
00007f80:65df2c94	c3	retq

·12 = 0x0000000000400510 <hello!_start+0x0>

5.6.1 edb 加载 hello

载入:

_dl_start

_dl_init

开始执行:

_start

_libc_start_main

_init

执行

main:

_main

_printf

_exit

_slee

_getchar

_dl_runtime_resolve_xsave

_dl_fixup

_dl_lookup_symbol_x

退出:

exit

5.7 Hello 的动态链接分析

在 edb 调试之后我们发现原先 0x00600a10 开始的 global_offset 表是全 0 的状态, 在执行过 _dl_init 之后被赋上了相应的偏移量的值。这说明 dl_init 操作是给程序赋上当前执行的内存地址偏移量, 这是初始化 hello 程序的一步。

0x0000000000400000-0x0000000000401000																0x0000000000600000-0x0000000000602000															
00000000:00600fe0		00	00	00	00	00	00	00	00	00	00	00	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	00	00	00	00	00	00	00	00	00	00	00			
00000000:00601000		28	0e	60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
00000000:00601010		00	00	00	00	00	00	00	00	00	00	00	00	e6	04	40	00	00	00	00	00	00	00	00	00	00	00	00			
00000000:00601020		f6	04	40	00	00	00	00	00	00	00	00	06	05	40	00	00	00	00	00	00	00	00	00	00	00	00	00			
00000000:00601030		16	05	40	00	00	00	00	00	00	00	00	26	05	40	00	00	00	00	00	00	00	00	00	00	00	00	00			
00000000:00601040		36	05	40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
00000000:00601050		00	00	00	00	00	00	00	00	00	00	00	02	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
00000000:00601060		00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
00000000:00601070		00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			

5.7.1dl init 前

00000000:00601000	28	0e	60	00	00	00	00	00	00	00	00	00	00	00	00	68	81	11	84	85	7f	00	00	00	00	00	00	00
00000000:00601010	70	88	f0	83	85	7f	00	00	00	00	00	00	00	00	00	e6	04	40	00	00	00	00	00	00	00	00	00	00
00000000:00601020	f6	04	40	00	00	00	00	00	00	00	00	00	00	00	00	06	05	40	00	00	00	00	00	00	00	00	00	
00000000:00601030	16	05	40	00	00	00	00	00	00	00	00	00	00	00	00	26	05	40	00	00	00	00	00	00	00	00	00	
00000000:00601040	36	05	40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	

5.7.2dl init 后

5.8 本章小结

本章介绍了链接过程中对 hello 的处理以及对链接后生成的可执行文件 hello 的分析, 包括 hello 的 elf 格式、hello 的虚拟地址空间、hello 的重定位以及动态链接的过程。链接完成后, hello 成为了一个可以运行的程序。

第 6 章 hello 进程管理

6.1 进程的概念与作用

进程的概念：

一个执行中的程序的实例，同时也是系统进行资源分配和调度的基本单位。一般情况下，包括文本区域、数据区域和堆栈。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储着活动过程调用的指令和本地变量。进程的作用：给予应用程序关键抽象：一个独立的逻辑流，它提供一个假象，好像我们的程序独占地使用处理器；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。

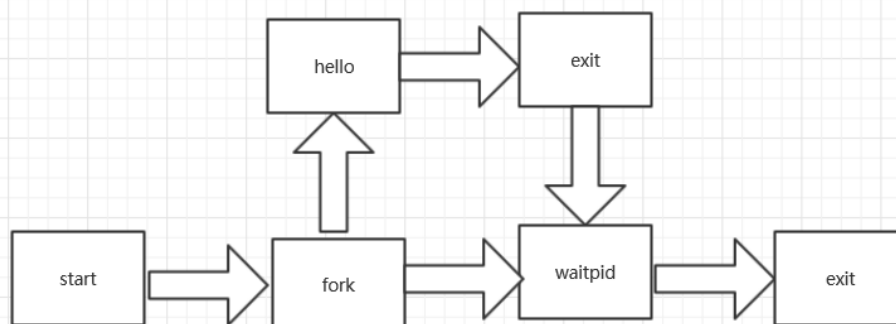
6.2 简述壳 Shell-bash 的作用与处理流程

Shell 的作用：Shell 是一个用 C 语言编写的程序，他是用户使用 Linux 的桥梁。Shell 是指一种应用程序，Shell 应用程序提供了一个界面，用户通过这个界面访问操作系统内核的服务。

shell-bash 的处理流程： 1. 从终端或控制台获取用户输入的命令 2. 对读入的命令进行分割并重构命令参数 3. 如果是内部命令则调用内部函数来执行 4. 否则执行外部程序 5. 判断程序的执行状态是前台还是后台，若为前台进程则等待进程结束；否则直接将进程放入后台执行，继续等待用户的下一次输入。

6.3 Hello 的 fork 进程创建过程

首先，要运行 hello 程序，需要在 shell 输入 ./hello 1170300901 侯欣宇。接下来 shell 会分析这一串命令： 1. 先判断 ./hello 是否是内置命令，结果是它不是内置命令 2. 然后 shell 调用 fork() 函数，创建一个子进程，这个子进程与父进程几乎没有差别，子进程的虚拟地址空间均与父进程的映射关系一致，是父进程虚拟地址空间的一份副本，包括代码和数据段、堆、共享库以及用户栈。同时，子进程还获得与父进程任何打开文件描述符相同的副本，故此时子进程可以读写父进程打开的任何文件。子进程与父进程的最大差别在于他们有不同的 PID。 3. 接下来 hello 将在 fork 创建的子进程中执行。流程图如下：



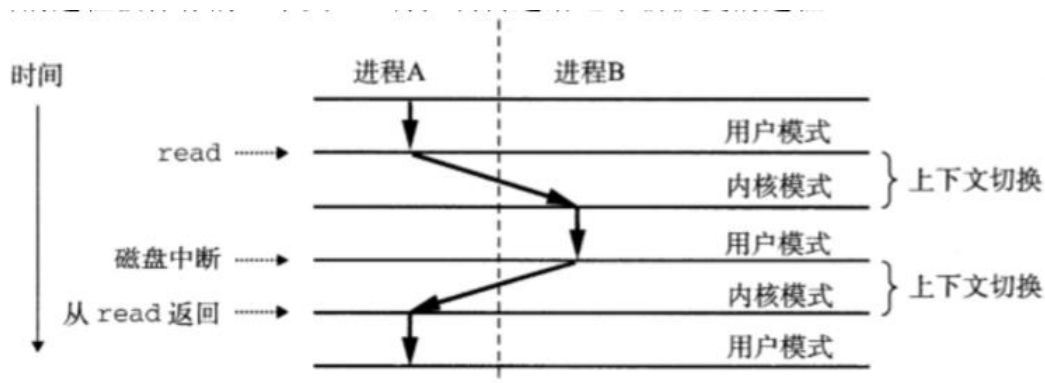
6.3.1 流程图

6.4 Hello 的 execve 过程

fork 之后, shell 在子进程中调用 `execve` 函数, 在当前进程的上下文中加载并运行 `hello` 程序, `execve` 调用驻留在内存中的被称为启动加载器的操作系统代码来执行 `hello` 程序, 加载器删除子进程现有的虚拟内存段, 并创建一组新的代码、数据、堆和栈段。新的栈和堆段被初始化为零, 通过将虚拟地址空间中的页映射到可执行文件的页大小的片, 新的代码和数据段被初始化为可执行文件中的内容, 然后跳转到 `_start`, `_start` 函数调用系统启动函数 `__libc_start_main` 来初始化环境, 调用用户层中 `hello` 的 `main` 函数, 并在需要的时候将控制返回给内核。

6.5 Hello 的进程执行

操作系统内核使用一中称为上下文切换的较高层形式的异常控制流来实现多任务: 内核为每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, 它由一些对象的值组成, 这些对象包括通用目的寄存器、浮点寄存器、程序计数器、用户栈、状态寄存器、内核栈和各种内核数据结构, 比如描述地址空间的页表、包含有关当前进程信息的进程表, 以及包含进程一打开文件的信息的文件表。上下文切换的流程是: 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。



6.5.1 进程上下文切换

为了使操作系统内核提供一个无懈可击的进程抽象，处理器必须提供一种机制，限制一个应用可以执行的指令以及它可以访问的地址空间范围。处理器通常使用某个控制寄存器的一个模式位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。接下来分析 `hello` 的进程调度，`hello` 在刚开始运行时内核为其保存一个上下文，进程在用户状态下运行。如果没有异常或中断信号的产生，`hello` 将继续正常地执行。如果有异常或系统中断，那么内核将启用调度器休眠当前进程，并在内核模式中完成上下文切换，将控制传递给其他进程。当 `hello` 运行到 `sleep(sleepsecs)` 时，`hello` 显式地请求休眠，并发生上下文切换，控制转移给另一个进程，此时计时器开始计时，当计时器到达 2s 时，它会产生一个中断信号，中断当前正在进行的进程，进行上下文切换，恢复 `hello` 在休眠前的上下文信息，控制权回到 `hello` 继续执行。当循环结束后，`hello` 调用 `getchar` 函数，之前 `hello` 运行在用户模式下，在调用 `getchar` 时进入内核模式，内核中的陷阱处理程序请求来自键盘缓冲区的 DMA 传输，并执行上下文切换，并把控制转移给其他进程。当完成键盘缓冲区到内存的数据传输后，引发一个中断信号，此时内核从其他进程切换回 `hello` 进程，然后 `hello` 执行 `return`，进程终止。

结合进程上下文信息、进程时间片，阐述进程调度的过程，用户态与核心态转换等等。

6.6 `hello` 的异常与信号处理

```

hxy1170300901@ubuntu:~/Documents/finalwork$ ./hello 1170300901 侯欣宇
Hello 1170300901 侯欣宇
haskdjklksjHello 1170300901 侯欣宇
sawueherm,bHello 1170300901 侯欣宇
/'f[ahqeHello 1170300901 侯欣宇
ppjsfklHello 1170300901 侯欣宇
n././kHello 1170300901 侯欣宇
jbskjagddqqpHello 1170300901 侯欣宇
nx,vHello 1170300901 侯欣宇
sdaqHello 1170300901 侯欣宇
Hello 1170300901 侯欣宇

```

6.6.1 瞎输入

瞎输入一些字母，不影响。

```

hxy1170300901@ubuntu:~/Documents/finalwork$ ./hello 1170300901 侯欣宇
Hello 1170300901 侯欣宇

Hello 1170300901 侯欣宇
Hello 1170300901 侯欣宇
Hello 1170300901 侯欣宇
Hello 1170300901 侯欣宇
^C

```

6.6.2 ctrl+c

回车也不影响，但 ctrl+c 程序结束了，shell 父进程收到 SIGINT 信号，信号处理函数的逻辑是结束 hello，并回收 hello 进程。

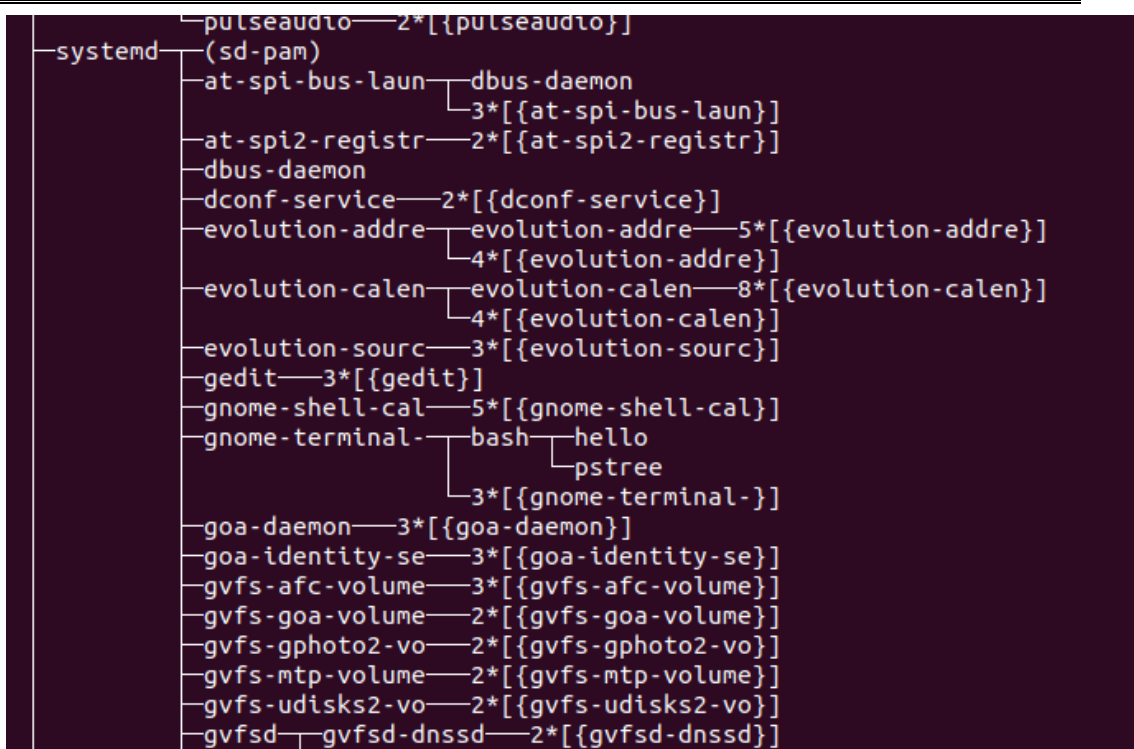
```

hxy1170300901@ubuntu:~/Documents/finalwork$ ./hello 1170300901 侯欣宇
Hello 1170300901 侯欣宇
^Z
[1]+ 已停止                  ./hello 1170300901 侯欣宇
hxy1170300901@ubuntu:~/Documents/finalwork$ ps
  PID TTY          TIME CMD
  5763 pts/0        00:00:00 bash
  5847 pts/0        00:00:00 hello
  5848 pts/0        00:00:00 ps
hxy1170300901@ubuntu:~/Documents/finalwork$ jobs
[1]+ 已停止                  ./hello 1170300901 侯欣宇

```

6.6.3 ctrl+z

在 hello 运行中按下 Ctrl+Z，hello 被挂起，通过 ps 命令可以看到 hello 程序的 pid，通过 jobs 命令可以看到 hello 进程被挂起。



6.6.4pstree 命令

用 pstree 命令可以在进程树中找到 bash 下的 hello 进程。

```

└─wpa_supplicant
hxy1170300901@ubuntu:~/Documents/finalwork$ fg 1
./hello 1170300901 侯欣宇
Hello 1170300901 侯欣宇
Hello 1170300901 侯欣宇
^Z
[1]+  已停止                  ./hello 1170300901 侯欣宇
hxy1170300901@ubuntu:~/Documents/finalwork$ ps
  PID TTY          TIME CMD
  5763 pts/0        00:00:00 bash
  5847 pts/0        00:00:00 hello
  5869 pts/0        00:00:00 ps
hxy1170300901@ubuntu:~/Documents/finalwork$ kill 5847
hxy1170300901@ubuntu:~/Documents/finalwork$ kill -9 5847
hxy1170300901@ubuntu:~/Documents/finalwork$ ps
  PID TTY          TIME CMD
  5763 pts/0        00:00:00 bash
  5870 pts/0        00:00:00 ps
[1]+  已杀死                  ./hello 1170300901 侯欣宇

```

6.6.5fg 命令和 kill

用 fg 1 命令可以让 hello 重新在前台运行。用 ps 可以看到 hello 的 PID 是 5847，故可以用 kill -9 5847 终止 hello 进程。

6.7 本章小结

本章介绍了进程的概念和作用，和 `hello` 由 `Program` 变身为 `Process` 后，调用 `fork` 创建新进程，调用 `execve` 执行 `hello`，`hello` 的进程执行，`hello` 的异常与信号处理。

第 7 章 hello 的存储管理

7.1 hello 的存储器地址空间

物理地址：CPU 通过地址总线的寻址，找到真实的物理内存对应地址。CPU 对内存的访问是通过连接着 CPU 和北桥芯片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。

逻辑地址：程序代码经过编译后出现在汇编程序中地址。逻辑地址由选择符（在实模式下是描述符，在保护模式下是用来选择描述符的选择符）和偏移量（偏移部分）组成。

线性地址：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。

至于虚拟地址，只关注 CSAPP 课本中提到的虚拟地址，实际上就是这里的线性地址。

7.2 Intel 逻辑地址到线性地址的变换-段式管理

最初 8086 处理器的寄存器是 16 位的，为了能够访问更多的地址空间但不改变寄存器和指令的位宽，所以引入段寄存器，8086 共设计了 20 位宽的地址总线，通过将段寄存器左移 4 位加上偏移地址得到 20 位地址，这个地址就是逻辑地址。将内存分为不同的段，段有段寄存器对应，段寄存器有一个栈、一个代码、两个数据寄存器。分段功能在实模式和保护模式下有所不同。实模式，即不设防，也就是说逻辑地址=线性地址=实际的物理地址。段寄存器存放真实段基址，同时给出 32 位地址偏移量，则可以访问真实物理内存。在保护模式下，线性地址还需要经过分页机制才能够得到物理地址，线性地址也需要逻辑地址通过段机制来得到。段寄存器无法放下 32 位段基址，所以它们被称作选择符，用于引用段描述符表中的表项来获得描述符。描述符表中的一个条目描述一个段。

Base：基地址，32 位线性地址指向段的开始。**Limit**：段界限，段的大小。**DPL**：描述符的特权级 0（内核模式）-3（用户模式）。

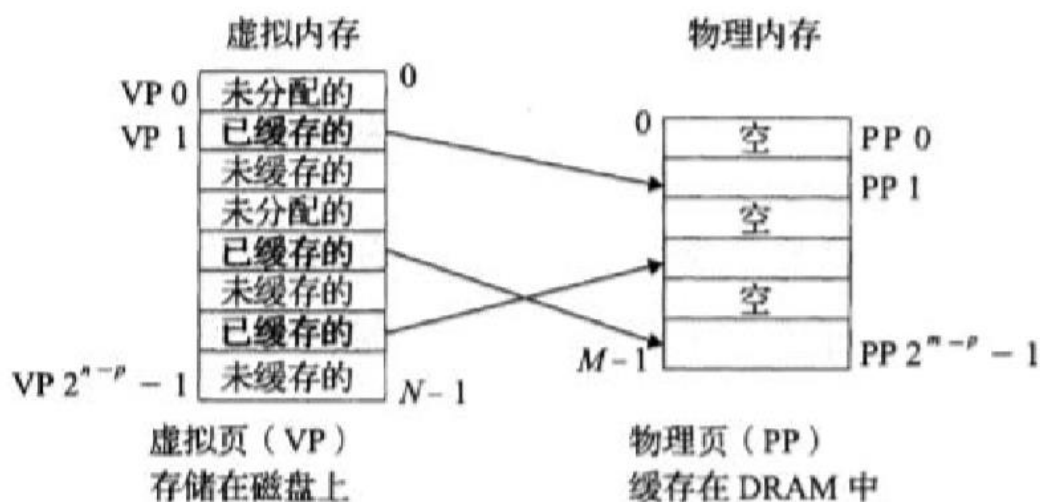
所有的段描述符被保存在两个表中：全局描述符表 GDT 和局部描述符表 LDT。gdt 寄存器指向 GDT 表基址。

在保护模式下，分段机制就可以描述为：通过解析段寄存器中的段选择符在段描述符表中根据 Index 选择目标描述符条目 Segment Descriptor，从目标描述符中提取出目标段的基地址 Base address，最后加上偏移量 offset 共同构成线性地址 Linear Address。

当 CPU 位于 32 位模式时，内存 4GB，寄存器和指令都可以寻址整个线性地址空间，所以这时候不再需要使用基地址，将基地址设置为 0，此时逻辑地址=描述符=线性地址，Intel 的文档中将其称为扁平模型（flat model），现代的 x86 系统内核使用的是基本扁平模型，等价于转换地址时关闭了分段功能。在 CPU 64 位模式中强制使用扁平的线性空间。逻辑地址与线性地址就合二为一了。

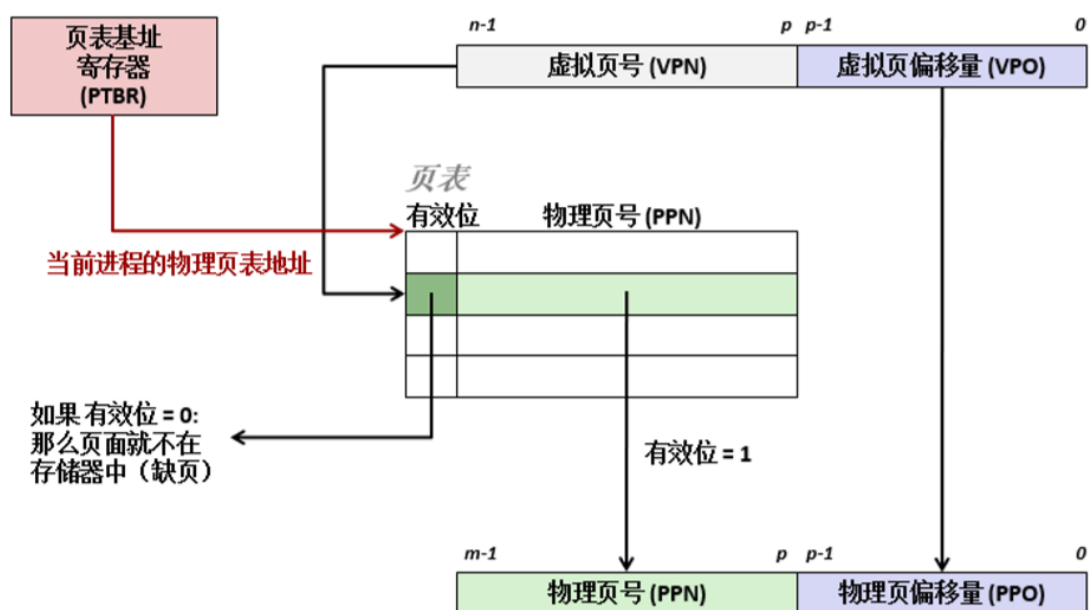
7.3 Hello 的线性地址到物理地址的变换-页式管理

概念上而言，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。磁盘上数组的内容被缓存在主存中。和存储器层次结构中其他缓存一样，磁盘（较低层）上的数据被分割成块，这些块作为磁盘和主存（较高层）之间的传输单元。VM 系统通过将虚拟内存分割位称为虚拟页的大小固定的块来处理这个问题。每个虚拟页的大小为 $P = 2^p$ 字节。类似地，物理内存被分割为物理页，大小也为 P 字节。虚拟页面集合被分为三个不相交的子集：已缓存、未缓存和未分配。



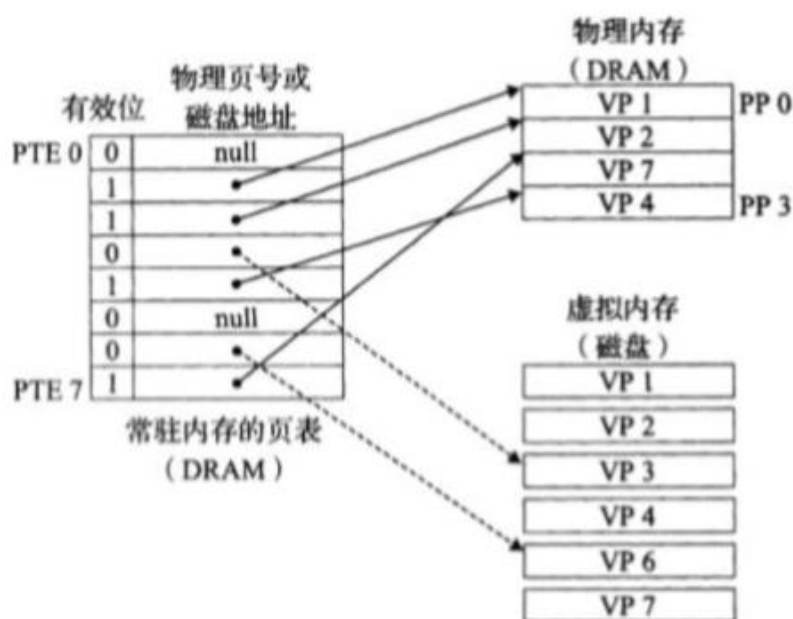
7.2.1 虚拟内存与物理内存的对应

下图展示了页式管理中虚拟地址到物理地址的转换：



7.2.2 页式管理中虚拟地址到物理地址的转换

虚拟地址分为两部分：前一部分为虚拟页号，可以索引到当前进程的物理页表地址，后一部分为虚拟页偏移量，将来可以直接作为物理页偏移量，页表是一个存放在物理内存中的数据结构，页表将虚拟页映射到物理页。每次地址翻译硬件将一个虚拟地址转换为物理地址时，都会读取页表。



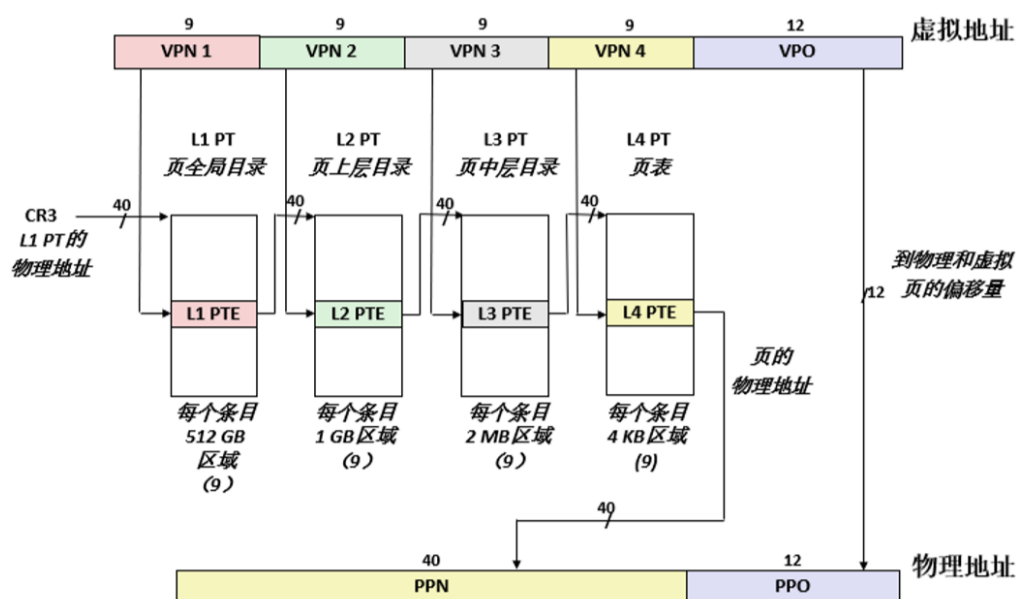
7.2.3 页表的基本组织结构

展示了一个页表的基本组织结构。虚拟地址空间中的每个页在页表中一个固定偏移量的位置都有一个 PTE(页表条目)，而每个 PTE 是由一个有效位和一个 n 位的地址字段组成的。页表 PTE 分为三种情况：1. 已分配：PTE 有效位为 1 且地址部分不为 null，即页面已被分配，将一个虚拟地址映射到了一个对应的物理地址 2. 未缓冲：PTE 有效位为 0 且地址部分不为 null，即页面已经对应了一个

虚拟地址，但虚拟内存内容还未缓存到物理内存中 3. 未分配：PTE 有效位为 0 且地址部分为 null，即页面还未分配，没有建立映射关系 现在根据图 7.4 介绍虚拟地址转换为物理地址的过程：首先根据虚拟页号在当前进程的物理页表中找到对应的页面，若符号位设置为 1，则表示命中，从页面中取出物理页号+虚拟页偏移量即组成了一个物理地址；否则表示不命中，产生一个缺页异常，需要从磁盘中读取相应的物理页到内存。

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

首先还是按照 7.3 所说的那样将 VPN 分成三段，对于 TLBT 和 TLBI 来说，如果可以在 TLB 中找到对应的 PPN 的话那肯定是最好不过的了，但是还有可能出现缺页的情况，这时候就需要到页表中去找。此时，VPN 被分成了更多段（这里是 4 段）CR3 是对应的 L1PT 的物理地址，然后一步步递进往下寻址，越往下一层每个条目对应的区域越小，寻址越细致，在经过 4 层寻址之后找到相应的 PPN 让你和 VPO 拼接起来。



7.4.1va 到 pa 的变换

7.5 三级 Cache 支持下的物理内存访问

前提：只讨论 L1 Cache 的寻址细节，L2 与 L3Cache 原理相同。L1 Cache 是 8 路 64 组相联。块大小为 64B。解析前提条件：因为共 64 组，所以需要 6bit CI 进行组寻址，因为共有 8 路，因为块大小为 64B 所以需要 6bit CO 表示数据偏移位置，因为 VA 共 52bit，所以 CT 共 40bit。在上一步中我们已经获得了物理地址 VA，使用 CI（后六位再后六位）进行组索引，每组 8 路，对 8 路的块分别匹配 CT（前

40 位) 如果匹配成功且块的 valid 标志位为 1, 则命中 (hit), 根据数据偏移量 CO (后六位) 取出数据返回。

如果没有匹配成功或者匹配成功但是标志位是 1, 则不命中 (miss), 向下一级缓存中查询数据 (L2 Cache->L3 Cache->主存)。查询到数据之后, 一种简单的放置策略如下: 如果映射到的组内有空闲块, 则直接放置, 否则组内都是有效块, 产生冲突 (evict), 则采用最近最少使用策略 LRU 进行替换。

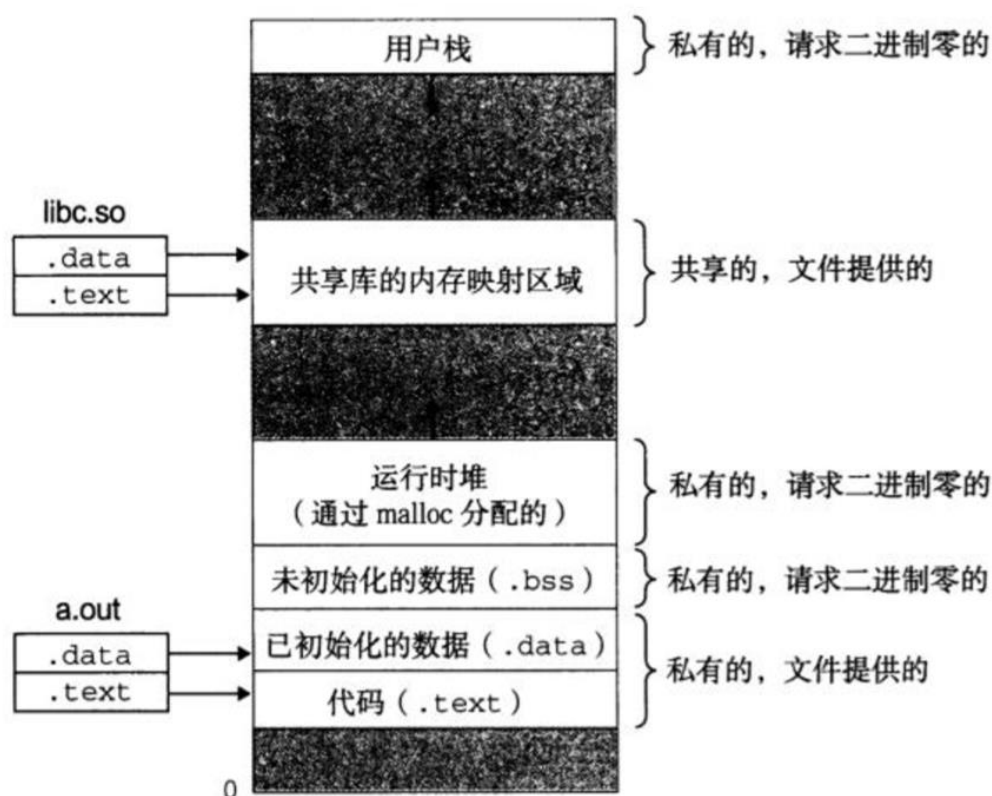
7.6 hello 进程 fork 时的内存映射

当 fork 函数被 shell 进程调用时, 内核为新进程创建各种数据结构, 并分配给它一个唯一的 PID, 为了给这个新进程创建虚拟内存, 它创建了当前进程的 mm_struct、区域结构和页表的原样副本。它将这两个进程的每个页面都标记为只读, 并将两个进程中的每个区域结构都标记为私有的写时复制。

7.7 hello 进程 execve 时的内存映射

execve 函数调用驻留在内核区域的启动加载器代码, 在当前进程中加载并运行包含在可执行目标文件 hello 中的程序, 用 hello 程序有效地替代了当前程序。

加载并运行 hello 需要以下几个步骤: 1. 删除已存在的用户区域, 删除当前进程虚拟地址的用户部分中的已存在的区域结构。2. 映射私有区域, 为新程序的代码、数据、bss 和栈区域创建新的区域结构, 所有这些新的区域都是私有的、写时复制的。代码和数据区域被映射为 hello 文件中的.text 和.data 区, bss 区域是请求二进制零的, 映射到匿名文件, 其大小包含在 hello 中, 栈和堆地址也是请求二进制零的, 初始长度为零。3. 映射共享区域, hello 程序与共享对象 libc.so 链接, libc.so 是动态链接到这个程序中的, 然后再映射到用户虚拟地址空间中的共享区域内。4. 设置程序计数器 (PC), execve 做的最后一件事情就是设置当前进程上下文中的程序计数器, 使之指向代码区域的入口点。



7.7.1 加载器是如何映射用户地址空间区域的

7.8 缺页故障与缺页中断处理

分为以下三种类型:

1. 段错误: 首先, 先判断这个缺页的虚拟地址是否合法, 那么遍历所有的合法区域结构, 如果这个虚拟地址对所有的区域结构都无法匹配, 那么就返回一个段错误 (segment fault)
2. 非法访问: 接着查看这个地址的权限, 判断一下进程是否有读写改这个地址的权限。
3. 如果不是上面两种情况那就是正常缺页, 那就选择一个页面牺牲然后换入新的页面并更新到页表。

7.9 动态存储分配管理

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

分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

分配器分为两种基本风格：显式分配器、隐式分配器。

- 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，自动释放未使用的已经分配的块的过程叫做垃圾收集。

其中涉及的组织结构知识还有显式空间链表，隐式空间链表，空间块的合并

7.10 本章小结

本章介绍了本章主要介绍了 `hello` 的存储器地址空间、intel 的段式管理、`hello` 的页式管理，以 intel Core7 在指定环境下介绍了 VA 到 PA 的变换、物理内存访问，还介绍了 `hello` 进程 `fork` 时的内存映射、`execve` 时的内存映射、缺页故障与缺页中断处理、动态存储分配管理。虽然 `hello` 很小，但无数个 `hello` 放在一起管理起来就变得非常棘手，计算机一定有条理清晰的存储和访问机制才能保证访问的速度。

第 8 章 hello 的 IO 管理

8.1 Linux 的 IO 设备管理方法

设备的模型化：所有的 IO 设备都被模型化为文件，而所有的输入和输出都被当做对相应文件的读和写来执行，这种将设备优雅地映射为文件的方式，允许 Linux 内核引出一个简单低级的应用接口，称为 Unix I/O。

8.2 简述 Unix IO 接口及其函数

Unix I/O 接口统一操作：打开文件。一个应用程序通过要求内核打开相应的文件，来宣告它想要访问一个 I/O 设备，内核返回一个小的非负整数，叫做描述符，它在后续对此文件的所有操作中标识这个文件，内核记录有关这个打开文件的所有信息。

Shell 创建的每个进程都有三个打开的文件：标准输入，标准输出，标准错误。

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

读写文件：一个读操作就是从文件复制 $n>0$ 个字节到内存，从当前文件位置 k 开始，然后将 k 增加到 $k+n$ ，给定一个大小为 m 字节的而文件，当 $k\geq m$ 时，触发 EOF。类似一个写操作就是从内存中复制 $n>0$ 个字节到一个文件，从当前文件位置 k 开始，然后更新 k 。

关闭文件，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去。

Unix I/O 函数：

`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。

`int close(fd)`，`fd` 是需要关闭的文件的描述符，`close` 返回操作结果。

`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

返回值表示的是实际传送的字节数量。

`ssize_t write(int fd, const void *buf, size_t n)`, `write` 函数从内存位置 `buf` 复制至多 `n` 个字节到描述符为 `fd` 的当前文件位置。

8.3 printf 的实现分析

```
int printf(const char fmt, ...)
{
    int i;
    char buf[256];
    va_list arg = (va_list)((char*)&fmt + 4);
    i = vsprintf(buf, fmt, arg);
    write(buf, i);
    return i;
}
```

其中, `vsprintf` 的作用是将所有的参数内容格式化之后存入 `buf` 数组, 然后返回格式化数组的长度。`write` 函数是将 `buf` 中的 `i` 个元素写到终端的函数。

`Printf` 的运行过程:

从 `vsprintf` 生成显示信息, 显示信息传送到 `write` 系统函数, `write` 函数再陷入系统调用 `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

当用户按键时, 键盘接口会得到一个代表该按键的键盘扫描码, 同时产生一个中断请求, 中断请求抢占当前进程运行键盘中断子程序, 键盘中断子程序先从键盘接口取得该按键的扫描码, 然后将该按键扫描码转换成 ASCII 码, 保存到系统的键盘缓冲区之中 再看 `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;
}
```

可以看到, `getchar` 调用了 `read` 函数, `read` 函数也通过 `sys_call` 调用内核中的

系统函数，将读取存储在键盘缓冲区中的 ASCII 码，直到读到回车符，然后返回整个字符串，`getchar` 函数只从中读取第一个字符，其他的字符被缓存在输入缓冲区。

8.5 本章小结

本章介绍了 Linux 中 I/O 设备的管理方法，Unix I/O 接口和函数，并且分析了 `printf` 和 `getchar` 函数是如何通过 Unix I/O 函数实现其功能的。

结论

Hello 的一生

1. `cpp` 预处理，处理以 `#` 开头的，得到 `hello.i`
2. 编译器将 `hello.i` 变成 `hello.s`
3. 汇编器将 `hello.s` 翻译成机器语言指令得到可重定位目标文件 `hello.o`
4. 链接器将 `hello.o` 与动态链接库链接生成可执行目标文件 `hello`，此时 `hello` 可以运行了
5. 运行：在 shell 中输入 `./hello 1170300901 侯欣宇`，shell 为 `hello` fork 一个子进程，并在子进程中调用 `execve`，加载运行 `hello`
6. CPU 为 `hello` 分配内存空间，`hello` 从磁盘被加载到内存。
7. 当 CPU 访问 `hello` 时，请求一个虚拟地址，MMU 把虚拟地址转换成物理地址并通过三级 cache 访存。
8. Shell 处理各种信号
9. Unix I/O 连接了输入输出硬件
10. 最后 `return 0`，安全结束

Hello 的一生如此精妙复杂，但展现给我们的只是屏幕输出 `hello` 等字符串的一瞬，不深入了解它（计算机系统），可能就丢生了一座宝库。

附件

hello.i	预处理之后文本文件
hello.s	编译之后的汇编文件
hello.o	汇编之后的可重定位目标执行
hello	链接之后的可执行目标文件
hello.elf	hello.o 的 ELF 格式
hello1	hello 的 ELF 格式

参考文献

为完成本次大作业你翻阅的书籍与网站等

- [1] 林来兴. 空间控制技术[M]. 北京：中国宇航出版社，1992：25-42.
- [2] 辛希孟. 信息技术与信息服务国际研讨会论文集：A 集[C]. 北京：中国科学出版社，1999.
- [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）