

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>1163300916</u>
班	級	<u>1703001</u>
學	生	<u>李一鳴</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

本文回顾了 Hello 的一生：from program to process，从 hello.c 文件先后经历预处理、编译、汇编、链接生成可执行目标文件 hello，之后在 shell 中运行，shell 对进程进行管理，发送及接收信号；同时关注 hello 在内存中存储信息的变化，以及 I/O 的状态变化。hello 运行结束后，shell 恢复了未运行 hello 时的状态，悄悄的 hello 走了，正如 hello 悄悄来，看起来便是 from Zero-0 to Zero-0。全程运用了 Linux 下 gcc, objdump, readelf 等工具，纵向横向进行比较，对 hello 的一生进行较完整的阐述。

关键词：预处理；编译；汇编；链接；进程；存储；I/O；

（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 -
1.2.1 硬件环境.....	- 4 -
1.2.2 软件环境.....	- 5 -
1.2.3 开发与调试工具.....	- 5 -
1.3 中间结果.....	- 5 -
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.....	- 7 -
2.3 HELLO 的预处理结果解析.....	- 7 -
2.4 本章小结.....	- 9 -
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.....	- 19 -
第 4 章 汇编.....	- 20 -
4.1 汇编的概念与作用.....	- 20 -
4.2 在 UBUNTU 下汇编的命令.....	- 20 -
4.3 可重定位目标 ELF 格式.....	- 20 -
4.4 HELLO.O 的结果解析.....	- 24 -
4.5 本章小结.....	- 24 -
第 5 章 链接.....	- 25 -
5.1 链接的概念与作用.....	- 25 -
5.2 在 UBUNTU 下链接的命令.....	- 25 -
5.3 可执行目标文件 HELLO 的格式.....	- 26 -
5.4 HELLO 的虚拟地址空间.....	- 27 -
5.5 链接的重定位过程分析.....	- 28 -
5.6 HELLO 的执行流程.....	- 29 -
5.7 HELLO 的动态链接分析.....	- 30 -
5.8 本章小结.....	- 31 -
第 6 章 HELLO 进程管理.....	- 32 -

6.1 进程的概念与作用.....	32 -
6.2 简述壳 SHELL-BASH 的作用与处理流程.....	32 -
6.3 HELLO 的 FORK 进程创建过程.....	32 -
6.4 HELLO 的 EXECVE 过程.....	33 -
6.5 HELLO 的进程执行.....	33 -
6.6 HELLO 的异常与信号处理.....	34 -
6.7 本章小结.....	36 -
第 7 章 HELLO 的存储管理.....	37 -
7.1 HELLO 的存储器地址空间.....	37 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理.....	38 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理.....	39 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换.....	40 -
7.5 三级 CACHE 支持下的物理内存访问.....	41 -
7.6 HELLO 进程 FORK 时的内存映射.....	42 -
7.7 HELLO 进程 EXECVE 时的内存映射.....	43 -
7.8 缺页故障与缺页中断处理.....	44 -
7.9 动态存储分配管理.....	45 -
7.10 本章小结.....	45 -
第 8 章 HELLO 的 IO 管理.....	46 -
8.1 LINUX 的 IO 设备管理方法.....	46 -
8.2 简述 UNIX IO 接口及其函数.....	46 -
8.3 PRINTF 的实现分析.....	47 -
8.4 GETCHAR 的实现分析.....	49 -
8.5 本章小结.....	49 -
结论.....	49 -
附件.....	50 -
参考文献.....	51 -

第 1 章 概述

1.1 Hello 简介

1.1.1 P2P: From Program to Process

hello 程序的生命周期从一个高级 C 语言程序——源程序 `hello.c` 开始(Program)。先后经过 `cpp` 的预处理、`ccl` 的汇编、`as` 的编译、`ld` 的链接,生成可执行目标程序 `hello`。在命令行中输入“`gcc -o hello hello.c`”可执行上述过程。

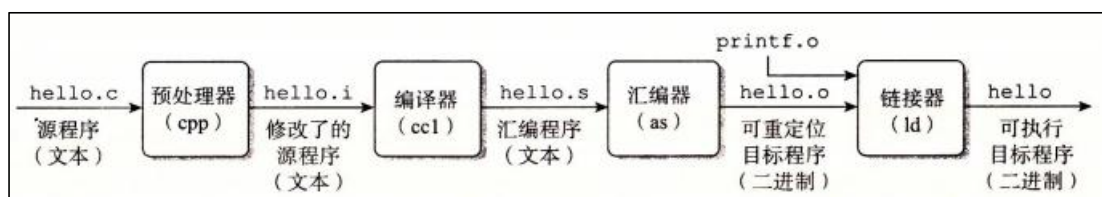


图 1.0 翻译过程

之后在 shell 中输入“`./hello`”，shell 会调用 `fork` 函数为其创建一个新的子进程(Process).....

以上便是 P2P 的过程。

1.1.2 020: From Zero-0 to Zero-0

接着 shell 调用 `execve` 函数,删除已存在的用户区域、映射私有区域、映射共享区域、设置程序计数器 PC。进入程序入口, Linux 根据需要换入代码和数据页面,执行目标代码。当 `hello` 进程运行结束后,内核会将其退出状态传递给父进程。当已终止的 `hello` 进程被父进程来回收后,内核会删除它的所有痕迹,于是 shell 再次变成 `hello` 执行之前的状态(Zero-0)。

以上便是 020 的过程。

1.2 环境与工具

1.2.1 硬件环境

DESKTOP-0DL5313

处理器:	Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz 2.59 GHz
已安装的内存(RAM):	8.00 GB (7.84 GB 可用)
系统类型:	64 位操作系统, 基于 x64 的处理器

1.2.2 软件环境

Windows10 64 位; VMware Workstation Pro; Ubuntu 18.04.1 64 位;

1.2.3 开发与调试工具

gdb, edb, IDA v7.0.180201,
HexEdit, notepad++,
gcc, objdump, readelf,
cpp, as, ld

1.3 中间结果

列出你为编写本论文，生成的中间结果文件的名称，文件的作用等。

hello.c 源程序(文本)

hello.i 修改了的源程序(文本)

hello.s 汇编程序(文本)

hello.o 可重定位目标程序(二进制)

hello 可执行目标程序(二进制)

1.4 本章小结

本章主要简述了 P2P, 020 的过程，并列出了实验的环境与工具、生成的中间文件。

(第 1 章 0.5 分)

第 2 章 预处理

2.1 预处理的概念与作用

2.1.1 概念：

预处理器 `cpp` 根据以字符`#`开头的命令（宏定义、条件编译），修改原始的 C 程序，将引用的所有库展开合并成为一个完整的文本文件。

GCC 编译器驱动程序把源文件翻译成可执行目标文件的过程中的第 1 个阶段(首阶段)。

2.1.2 作用：

1. 文件包含。`#include` 指令告诉预处理器(`cpp`)读取源程序所引用的系统源文件，并把源文件直接插入程序文本中。

2. 执行宏替代。`#define` 指令可以定义符号常量、函数功能、重新命名、字符串拼接等各种功能，预处理过程中预处理器会将源程序中的宏替换成实际值。

3. 删除注释。即删除`//`之后的内容，`/**/`之间的内容。

4. 条件编译。根据实际定义宏（某类条件）进行代码静态编译的手段。可根据表达式的值或某个特定宏是否被定义来确定编译条件。常见例子：

语句	含义
<code>#if</code>	编译预处理中的条件命令，相当于 C 语法中的 <code>if</code> 语句
<code>#ifdef</code>	判断某个宏是否被定义，若已定义，执行随后的语句
<code>#elif</code>	若 <code>#if</code> , <code>#ifdef</code> , <code>#ifndef</code> 或前面的 <code>#elif</code> 条件不满足，则执行 <code>#elif</code> 之后的语句，相当于 C 语法中的 <code>else-if</code>
<code>#else</code>	与 <code>#if</code> , <code>#ifdef</code> , <code>#ifndef</code> 对应，若这些条件不满足，则执行 <code>#else</code> 之后的语句，相当于 C 语法中的 <code>else</code>
<code>#endif</code>	<code>#if</code> , <code>#ifdef</code> , <code>#ifndef</code> 这些条件命令的结束标志.
<code>#ifndef</code>	与 <code>#ifdef</code> 相反，判断某个宏是否未被定义
<code>defined</code>	与 <code>#if</code> , <code>#elif</code> 配合使用，判断某个宏是否被定义

5. 布局控制。为编译程序提供非常规的控制流信息。

2.2 在 Ubuntu 下预处理的命令

在 hello.c 所在目录下打开 terminal

输入指令并回车：

gcc hello.c -E(通过-E 指令，只激活预处理指令)

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

或 cpp hello.c > hello.i

目录下生成 hello.i 文件

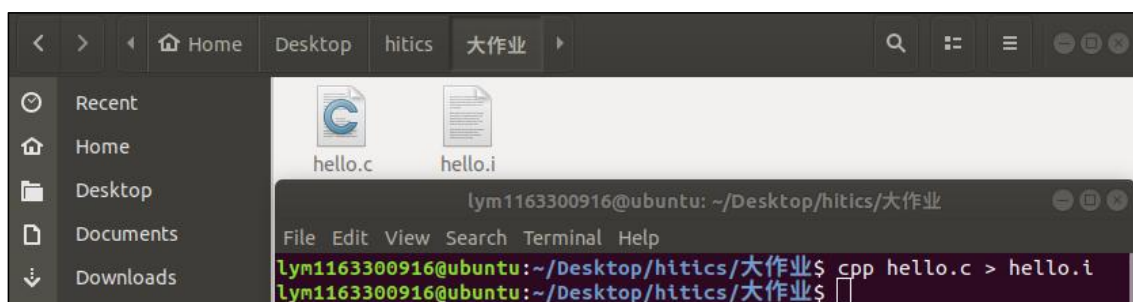


图 2.0 Ubuntu 下预处理的命令

2.3 Hello 的预处理结果解析

用 gedit 打开 hello.i，和原文件.c 相比，发现：

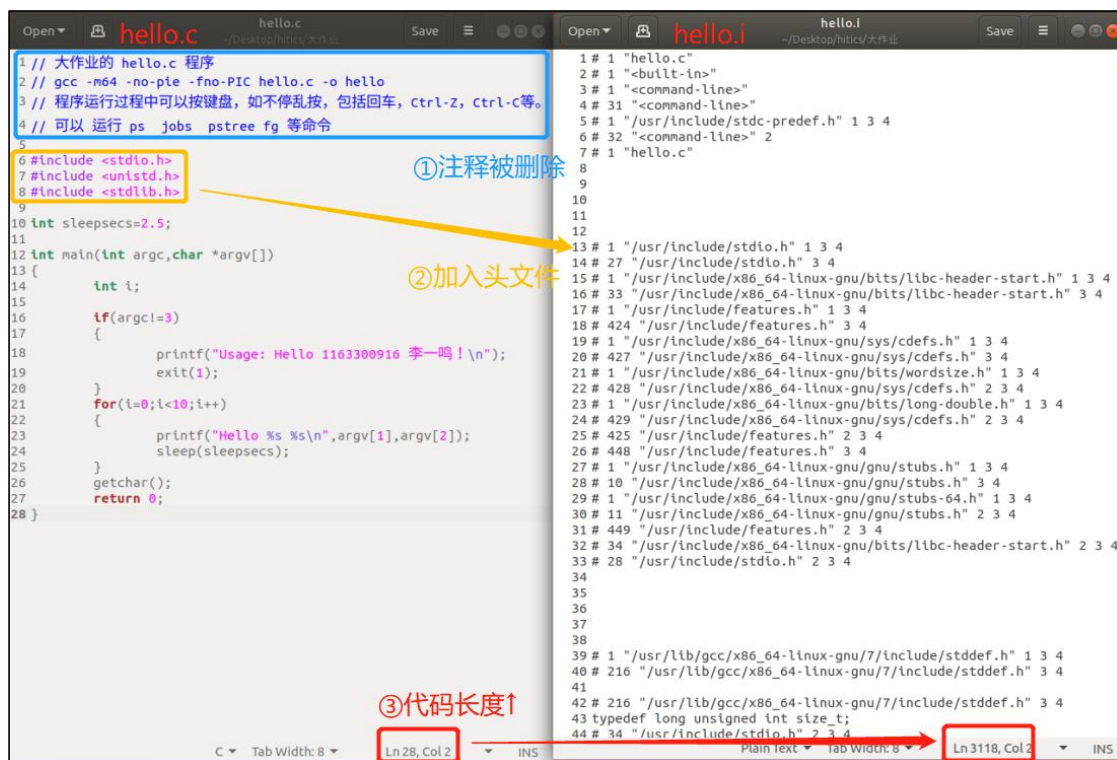


图 2.1 比较 hello.c 和 hello.i

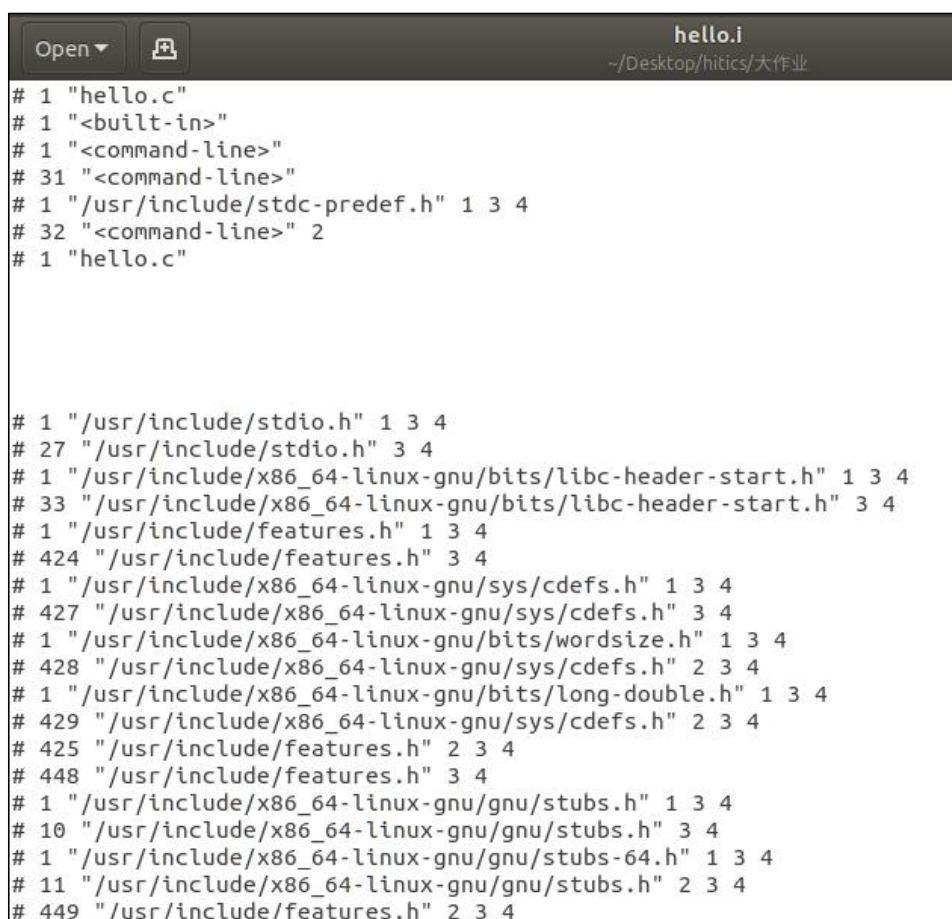
- ①注释被删除
- ②行数增加
- ③源代码未修改，在文末

预处理的文件包含的效果

hello.c 程序中包含三条预处理指令：

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
```

预处理将这三个系统头文件的内容引入(寻址和解析，描述使用的运行库在计算机中的位置)



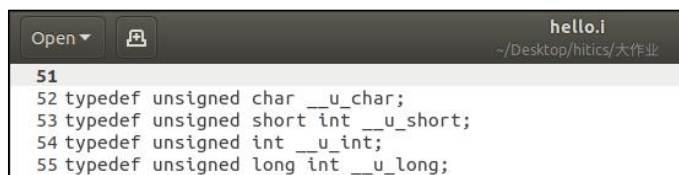
```
# 1 "hello.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "hello.c"

# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
# 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 424 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
# 427 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 428 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/long-double.h" 1 3 4
# 429 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 425 "/usr/include/features.h" 2 3 4
# 448 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 1 3 4
# 10 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs-64.h" 1 3 4
# 11 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 2 3 4
# 449 "/usr/include/features.h" 2 3 4
```

图 2.2 hello.i 内容

因为 hello.c 包含的头文件中还包含有其他头文件，系统递归式的寻址和展开，直到文件中不含宏定义且相关的头文件均已被引入。

同时引入了头文件中所有 typedef 关键字，结构体类型、枚举类型、通过 extern 关键字调用并声明外部的结构体及函数定义：



```

51
52 typedef unsigned char __u_char;
53 typedef unsigned short int __u_short;
54 typedef unsigned int __u_int;
55 typedef unsigned long int __u_long;

```

图 2.3 typedef 关键字



```

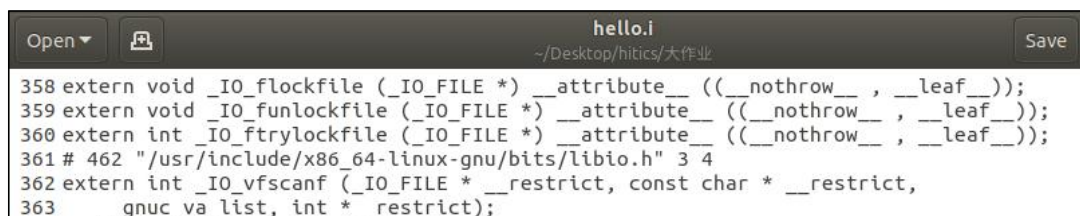
189 typedef struct
190 {
191     int __count;
192     union
193     {
194         unsigned int __wch;
195         char __wchb[4];
196     } __value;
197 } __mbstate_t;

242 enum __codecvt_result
243 {
244     __codecvt_ok,
245     __codecvt_partial,
246     __codecvt_error,
247     __codecvt_noconv
248 };

```

图 2.4 结构体类型

图 2.5 枚举类型



```

358 extern void __IO_flockfile (__IO_FILE *) __attribute__((__nothrow__ , __leaf__));
359 extern void __IO_funlockfile (__IO_FILE *) __attribute__((__nothrow__ , __leaf__));
360 extern int __IO_ftrylockfile (__IO_FILE *) __attribute__((__nothrow__ , __leaf__));
361 # 462 "/usr/include/x86_64-linux-gnu/bits/libio.h" 3 4
362 extern int __IO_vfscanf (__IO_FILE * __restrict, const char * __restrict,
363     __gnuc_va_list, int *__restrict);

```

图 2.6 extern 声明



```

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 1163300916 李一鸣!\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 }

```

图 2.7 hello.i 末尾是 main 函数的内容

2.4 本章小结

本章主要介绍了预处理的概念和作用、并对预处理的结果 hello.i 进行分析

(第 2 章 0.5 分)

第 3 章 编译

3.1 编译的概念与作用

3.1.1 概念：

编译是利用编译程序从预处理文本文件产生汇编程序（文本）的过程。

有五个阶段：词法分析、语法分析、语义检查、中间代码生成、目标代码生成。

GCC 编译器驱动程序把源文件翻译成可执行目标文件的过程中的第 2 个阶段，为下一步的汇编作准备。

3.1.2 作用：

编译器 `ccl` 将文本文件 `hello.i` 翻译成文本文件 `hello.s`，并在出现语法错误时给出提示信息。汇编语言为不同高级语言的不同编译器提供了通用的输出语言。

1. 词法分析。

逐个字符地对源程序进行扫描，将源代码的字符序列分割成一系列记号。

2. 语法分析。语法分析器以单词符号作为输入，生成语法树。

3. 语义检查。

语义分析器判断是否合法，即是否合乎逻辑结构和语法规则，不判断对错。

4. 目标代码生成和优化。

编译器会选择合适的寻址方式，左移右移代替乘除，删除多余指令等。

3.2 在 Ubuntu 下编译的命令

命令：`gcc hello.i -S` 或 `gcc -S hello.i -o hello.s`

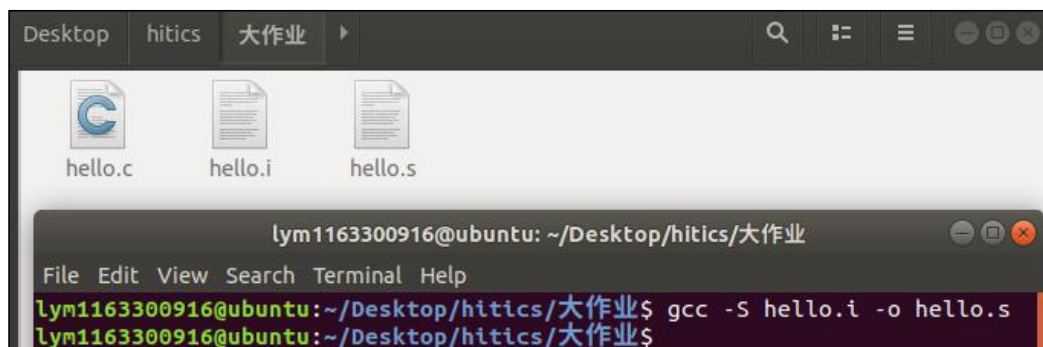


图 3.0 ubuntu 下编译命令

3.3 Hello 的编译结果解析

3.3.1 汇编文件伪指令

指令	含义
.file	声明源文件
.text	声明代码段
.data	声明数据段
.align	声明对齐方式
.type	指定类型
.size	声明大小
.section .rodata	只读数据
.globl	全局变量

3.3.2 数据

1、全局变量

hello.c 中定义了一个全局变量 sleepsecs，赋初值 2.5

hello.s 中声明 sleepsecs 为全局变量，存放在.data 节(4 字对齐)，

sleepsecs 类型为 object，占 4 字节，值为 long 型的 2(隐式类型转换)

```

1      .file    "hello.c"
2      .text
3      .globl   sleepsecs
4      .data
5      .align   4
6      .type    sleepsecs, @object
7      .size    sleepsecs, 4
8  sleepsecs:
9      .long    2
10     .section  .rodata
11     .align    8

```

图 3.1 hello.s 中的 sleepsecs

2、局部变量：

①int argc 记录从终端输入的参数个数，

是 main 的第一个参数，由寄存器%edi 保存，之后被存入-20(%rbp)栈中

```

27     subq     $32, %rsp
28     movl     %edi, -20(%rbp)
29     movq     %rsi, -32(%rbp)
30     cmpl     $3, -20(%rbp)
31     je       .L2

```

图 3.2 argc 部分

②int i 是循环中计数的局部变量。局部变量一般存储在寄存器或者栈中，由图可知，i 存储在-4(%rbp)中，初始值为 0，每次循环加 1，当 i>9 时跳出循环

```

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

```

图 3.3 for(i=0;i<10;i++) “i=0”

```

53      addl    $1, -4(%rbp)|
54      .L3:
55      cmpl    $9, -4(%rbp)|

```

图 3.4 “i++” 和 “i<10”

3、常量：

.c 文件的常量在汇编代码中以立即数\$形式显示。

```

27      subq    $32, %rsp
28      movl    %edi, -20(%rbp)
29      movq    %rsi, -32(%rbp)
30      cmpl    $3, -20(%rbp)
31      je      .L2

```

图 3.5 立即数表示

if(argc!=3) 的常数“3”在汇编中显示成“\$3”

4、数组

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

hello.c 中的数组是 main()函数的第二个参数 char *argv[] 被保存在寄存器%rsi 中，然后又被保存到栈中 32(%rbp)。

```

27      subq    $32, %rsp
28      movl    %edi, -20(%rbp)
29      movq    %rsi, -32(%rbp)
30      cmpl    $3, -20(%rbp)
31      je      .L2

```

图 3.6 数组操作 1

在循环体 .L4 内，函数printf("Hello %s %s\n", argv[1], argv[2]);:

数组首地址先被赋给%rax，

然后分两次读取了%rax+16 和%rax+8 地址的内容：argv[2]和 argv[1]，分别放入寄存器%rdx (printf 的第 3 个参数) 和%rsi (printf 的第 2 个参数)，再把 .LC1 放入%edi (printf 函数的第一个参数)，最后调用 printf 函数，

体现寄存器从前往后分配，调用前参数从后往前加载的过程：

```

39  .L4:
40      movq    -32(%rbp), %rax
41      addq    $16, %rax
42      movq    (%rax), %rdx
43      movq    -32(%rbp), %rax
44      addq    $8, %rax
45      movq    (%rax), %rax
46      movq    %rax, %rsi
47      leaq    .LC1(%rip), %rdi
48      movl    $0, %eax
49      call    printf@PLT

```

图 3.7 数组操作 2

5、字符串

hello.c 中共有两个字符串，分别是两个 printf 格式化输出的字符串

```

if(argc!=3)
{
    printf("Usage: Hello 1163300916 李一鸣! \n");
    exit(1);
}

```

图 3.8 c 语言字符串 1

```

for(i=0;i<10;i++)
{
    printf("Hello %s %s\n",argv[1],argv[2]);
    sleep(sleepsecs);
}

```

图 3.9 c 语言字符串 2

编译器一般会将字符串存放在.rodata 节

```

12  .LC0:
13      .string "Usage: Hello 1163300916
        \346\235\216\344\270\200\351\270\243\357\274\201"

```

图 3.10 汇编语言字符串 1

```

14  .LC1:
15      .string "Hello %s %s\n"

```

图 3.11 汇编语言字符串 2

第一个字符串 .LC0 包含汉字，每个汉字在 utf-8 编码中被编码为三字节 (全角的“!”感叹号也为 3 个字节)

第二个字符串的两个 “%s” 为用户在终端输入的两个参数

3.3.3 赋值

hello.c 的赋值操作有两处，

①将全局变量 sleepsecs 赋值为 2，

全局变量 sleepsecs 在.data 节中就已完成赋值，初始值为 long 型的 2。

```
8 sleepsecs:
9     .long 2
```

图 3.12 全局变量赋值

②局部变量 i 的赋值则由 mov 指令完成，在循环开始时将 i 赋值为 0。

```
37     movl $0, -4(%rbp)
```

图 3.13 局部变量赋值

3.3.3 类型转换

隐式类型转换

全局变量 sleepsecs 声明为 int 变量，所以赋值时，将 2.5 浮点数截断为 2。

```
3100 int sleepsecs=2.5;

8 sleepsecs:
9     .long 2
```

图 3.14 隐式类型转换的例子

3.3.4 算数运算

指令	效果	描述
leaq S, D	$D \leftarrow \&S$	加载有效地址
INC D	$D \leftarrow D + 1$	加1
DEC D	$D \leftarrow D - 1$	减1
NEG D	$D \leftarrow -D$	取负
NOT D	$D \leftarrow \sim D$	取补
ADD S, D	$D \leftarrow D + S$	加
SUB S, D	$D \leftarrow D - S$	减
IMUL S, D	$D \leftarrow D * S$	乘
XOR S, D	$D \leftarrow D \wedge S$	异或
OR S, D	$D \leftarrow D \vee S$	或
AND S, D	$D \leftarrow D \& S$	与
SAL k, D	$D \leftarrow D \ll k$	左移
SHL k, D	$D \leftarrow D \ll k$	左移（等同于SAL）
SAR k, D	$D \leftarrow D \gg_A k$	算术右移
SHR k, D	$D \leftarrow D \gg_L k$	逻辑右移

图 3.15 算数运算指令

hello.c 中的算术操作有：①循环内部变量 $i++$ ；②初始化分配栈空间 `sub`；编译器编译为 `addl $1,-4(%rbp)`，其中 1 是立即数， i 存放在 `-4(%rbp)`。

```

39      .L4:
40      movq    -32(%rbp), %rax
41      addq    $16, %rax
42      movq    (%rax), %rdx
43      movq    -32(%rbp), %rax
44      addq    $8, %rax
45      movq    (%rax), %rax
46      movq    %rax, %rsi
47      leaq    .LC1(%rip), %rdi
48      movl    $0, %eax
49      call    printf@PLT
50      movl    sleepsecs(%rip), %eax
51      movl    %eax, %edi
52      call    sleep@PLT
53      addl    $1, -4(%rbp)

```

图 3.16 `add: argv` 下标的获取； $i++$

```

27      subq    $32, %rsp
28      movl    %edi, -20(%rbp)
29      movq    %rsi, -32(%rbp)
30      cmpl    $3, -20(%rbp)
31      je     .L2

```

图 3.17 `sub: %rsp-0x32` 开辟栈空间

3.3.5 数组/指针/结构操作

hello.c 中对数组的操作是地址计算和取值

- ① `%rax+16` 地址对应的内容 `argv[2]` 放在 `%rdx` 中，作为 `printf` 的第 3 个参数，
- ② 再把 `%rax+8` 地址对应的 `argv[1]` 取出放在 `%rsi` 中，是 `printf` 的第 2 个参数

```

39      .L4:
40      movq    -32(%rbp), %rax
41      addq    $16, %rax
42      movq    (%rax), %rdx
43      movq    -32(%rbp), %rax
44      addq    $8, %rax
45      movq    (%rax), %rax
46      movq    %rax, %rsi

```

图 3.18 数组操作

3.3.6 关系操作

C 语言中的关系操作有==、!=、<、>、<=、>=等，
在汇编语言中主要由 `cmp` 指令和 `test` 指令实现

指令	基于	描述
CMP S_1, S_2	$S_2 - S_1$	比较
<code>cmpb</code>		比较字节
<code>cmpw</code>		比较字
<code>cmpl</code>		比较双字
<code>cmpq</code>		比较四字
TEST S_1, S_2	$S_1 \& S_2$	测试
<code>testb</code>		测试字节
<code>testw</code>		测试字
<code>testl</code>		测试双字
<code>testq</code>		测试四字

图 3.19 关系操作指令

- ① `CMP` 指令根据两个操作数之差来设置条件码。除了只设置条件码而不更新目的寄存器之外, `CMP` 指令与 `SUB` 指令的行为是一样的。
- ② `TEST` 指令根据两操作数的并来设置条件码, 除了它们只设置条件码而不改变目的寄存器的值之外, `TEST` 指令和 `AND` 指令一样。

例如:

在 `hello.s` 中的两处关系操作:

```
30      cmpl    $3, -20(%rbp)
```

1、对应 `hello.c` 中 “`argc!=3`”

```
55      cmpl    $9, -4(%rbp)
```

2、对应 `hello.c` 中 “`i<10`”

3.3.7 控制转移

控制转移往往与关系操作配合进行，如果满足某个条件，则跳转至某个位置。

指令	同义名	跳转条件	描述
<code>jmp Label</code>		1	直接跳转
<code>jmp *Operand</code>		1	间接跳转
<code>je Label</code>	<code>jz</code>	ZF	相等/零
<code>jne Label</code>	<code>jnz</code>	\sim ZF	不相等/非零
<code>js Label</code>		SF	负数
<code>jns Label</code>		\sim SF	非负数
<code>jg Label</code>	<code>jnle</code>	\sim (SF ^ OF) & \sim ZF	大于（有符号>）
<code>jge Label</code>	<code>jnl</code>	\sim (SF ^ OF)	大于或等于（有符号>=）
<code>jl Label</code>	<code>jnge</code>	SF ^ OF	小于（有符号<）
<code>jle Label</code>	<code>jng</code>	(SF ^ OF) ZF	小于或等于（有符号<=）
<code>ja Label</code>	<code>jnbe</code>	\sim CF & \sim ZF	超过（无符号>）
<code>jae Label</code>	<code>jnb</code>	\sim CF	超过或相等（无符号>=）
<code>jb Label</code>	<code>jnae</code>	CF	低于（无符号<）
<code>jbe Label</code>	<code>jna</code>	CF ZF	低于或相等（无符号<=）

图 3.20 控制操作

例如：

```
30    cmpl    $3, -20(%rbp)
31    jle     .L2
```

1、对应 hello.c 中 “if(argc!=3)” 的跳转

```
54    .L3:
55    cmpl    $9, -4(%rbp)
56    jle     .L4
```

2、对应 hello.c 中 “for(i=0;i<10;i++)” 的跳转

3.3.8 函数操作

C 语言函数操作包含参数传递、函数调用和函数返回功能。

①参数传递：当 过程 P 调用过程 Q 时，P 有可能会传入参数，

64 位程序，传递参数的寄存器：

参数数量						
第 1 个	第 2 个	第 3 个	第 4 个	第 5 个	第 6 个	超过 6 个
%rdi	%rsi	%rdx	%rcx	%r8	%r9	栈

②函数调用：由 `call` 指令实现

③函数返回：过程 `Q` 把返回信息存入 `%rax` 中，由 `ret` 指令返回

1. main 函数

```
16      .text
17      .globl main
18      .type main, @function
```

`main` 函数被存在 `.text` 节，标记类型为 `function`，
有两个参数，分别为 `argc` 和 `argv[]`，
由命令行输入，存储在 `%rdi` 和 `%rsi` 中

2. printf 函数

`hello.c` 中有两个 `printf` 函数，

①第一个 `printf` 的参数只有一个，被优化为 `puts`

```
30      cmpl    $3, -20(%rbp)
31      je     .L2
32      leaq    .LC0(%rip), %rdi
33      call   puts@PLT
```

图 3.21 `printf1` 的汇编

②对于第二个 `printf` 函数，共有三个参数，分别是 `.LC1`, `argv[1]` 和 `argv[2]`，
`call` 调用之前分别加载参数

```
39      .L4:
40      movq    -32(%rbp), %rax
41      addq    $16, %rax
42      movq    (%rax), %rdx
43      movq    -32(%rbp), %rax
44      addq    $8, %rax
45      movq    (%rax), %rax
46      movq    %rax, %rsi
47      leaq    .LC1(%rip), %rdi
48      movl    $0, %eax
49      call   printf@PLT
```

图 3.22 `printf2` 的汇编

3. exit 函数

若输入的参数不是 3 个，调用 `exit` 函数，
`exit` 只有一个参数，表示退出状态

4. sleep 函数

传入 sleep 函数的参数是%rip+sleepsecs,

其中%rip 是指令指针寄存器, 保存当前指令的下一条指令的地址

50	movl	sleepsecs(%rip), %eax
51	movl	%eax, %edi
52	call	sleep@PLT

图 3.23 sleep 的汇编

5. getchar 函数

getchar 函数没有任何参数,

在相应位置直接调用即可

57	call	getchar@PLT
----	------	-------------

图 3.24 getchar 的汇编

6. ret 函数

在 main 函数结束之前, 把 0 赋给%eax 然后返回。

3.4 本章小结

本章主要介绍了编译器是如何处理 C 语言的各种数据类型和各种操作的, 对比结合 C 代码与汇编代码进行解释。

(第 3 章 2 分)

第 4 章 汇编

4.1 汇编的概念与作用

4.1.1 概念:

汇编器(as)将.s 文件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的格式, 并将结果保存在目标文件(这里是 hello.o)中。

4.1.2 作用:

将编译器产生的汇编语言进一步翻译为计算机可以理解的二进制机器语言, 并产生.o 文件。GCC 编译器驱动程序把源文件翻译成可执行目标文件的过程中的第 3 个阶段

4.2 在 Ubuntu 下汇编的命令

命令: `as hello.s -o hello.o`

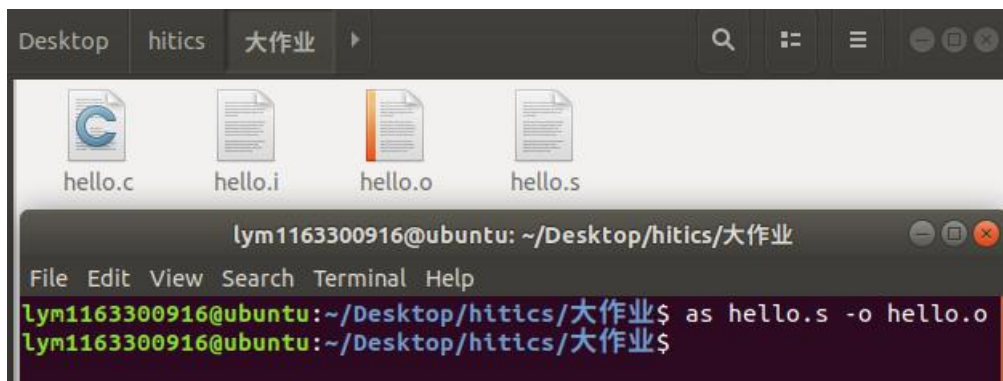


图 4.0 在 Ubuntu 下汇编的命令

4.3 可重定位目标 elf 格式

1、典型的 ELF 可重定位目标文件:

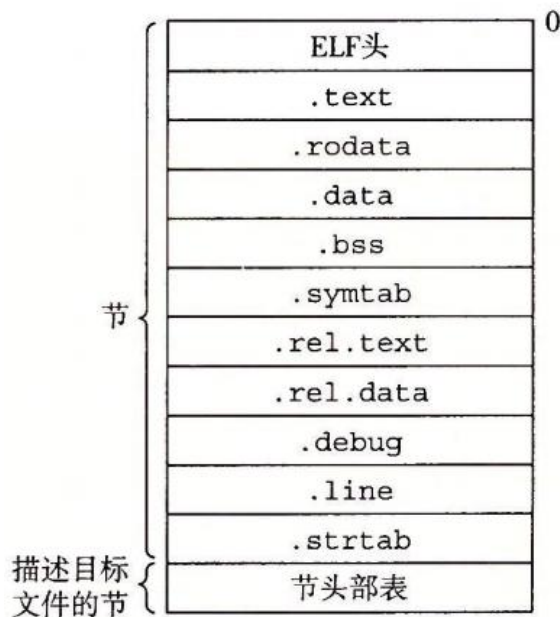


图 4.1 典型的 ELF 可重定位目标文件

2、命令：readelf -a hello.o > hello.elf。

```

1  ELF Header:
2  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
3  Class:                               ELF64
4  Data:                                   2's complement, little endian
5  Version:                               1 (current)
6  OS/ABI:                                UNIX - System V
7  ABI Version:                           0
8  Type:                                  REL (Relocatable file)
9  Machine:                                Advanced Micro Devices X86-64
10 Version:                                0x1
11 Entry point address:                    0x0
12 Start of program headers:                0 (bytes into file)
13 Start of section headers:                1160 (bytes into file)
14 Flags:                                  0x0
15 Size of this header:                     64 (bytes)
16 Size of program headers:                 0 (bytes)
17 Number of program headers:               0
18 Size of section headers:                 64 (bytes)
19 Number of section headers:               13
20 Section header string table index:       12

```

图 4.2 ELF 头

ELF 头以一个 16 字节的序列开始，这个序列描述了生成该文件的系统的大小和字节顺序。剩下的部分包含帮助连接器进行语法分析和解释目标文件的信息。不同节的位置和大小是由节头部表描述的。

3、节头部表:

Section Headers:							
[Nr]	Name	Type	Address	Offset			
	Size	EntSize	Flags	Link	Info	Align	
[0]		NULL	0000000000000000	00000000			
	0000000000000000	0000000000000000		0	0	0	
[1]	.text	PROGBITS	0000000000000000	00000040			
	0000000000000081	0000000000000000	AX	0	0	1	
[2]	.rela.text	RELA	0000000000000000	00000348			
	00000000000000c0	0000000000000018	I	10	1	8	
[3]	.data	PROGBITS	0000000000000000	000000c4			
	0000000000000004	0000000000000000	WA	0	0	4	
[4]	.bss	NOBITS	0000000000000000	000000c8			
	0000000000000000	0000000000000000	WA	0	0	1	
[5]	.rodata	PROGBITS	0000000000000000	000000c8			
	0000000000000032	0000000000000000	A	0	0	8	
[6]	.comment	PROGBITS	0000000000000000	000000fa			
	000000000000002b	0000000000000001	MS	0	0	1	
[7]	.note.GNU-stack	PROGBITS	0000000000000000	00000125			
	0000000000000000	0000000000000000		0	0	1	
[8]	.eh_frame	PROGBITS	0000000000000000	00000128			
	0000000000000038	0000000000000000	A	0	0	8	
[9]	.rela.eh_frame	RELA	0000000000000000	00000408			
	0000000000000018	0000000000000018	I	10	8	8	
[10]	.symtab	SYMTAB	0000000000000000	00000160			
	00000000000000198	0000000000000018		11	9	8	
[11]	.strtab	STRTAB	0000000000000000	000002f8			
	000000000000004d	0000000000000000		0	0	1	
[12]	.shstrtab	STRTAB	0000000000000000	00000420			
	0000000000000061	0000000000000000		0	0	1	

图 4.3 节头部表

节头部表描述了 hello.o 文件各个节的信息: name, type, address, offset, size, entsize, flags, link, info, align, 比如 .data 节的地址是 0, 偏移量是 0xc4, 大小是 0x04。

4、重定位节.rela.text:

Relocation section '.rela.text' at offset 0x348 contains 8 entries:					
Offset	Info	Type	Sym. Value	Sym. Name + A	
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 + 21	
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	

图 4.4 重定位节

5、.symtab

```

80 Symbol table '.symtab' contains 17 entries:
81
82 Num:      Value      Size Type   Bind   Vis      Ndx Name
83 0: 0000000000000000 0 NOTYPE LOCAL DEFAULT UND
84 1: 0000000000000000 0 FILE  LOCAL DEFAULT ABS hello.c
85 2: 0000000000000000 0 SECTION LOCAL DEFAULT 1
86 3: 0000000000000000 0 SECTION LOCAL DEFAULT 3
87 4: 0000000000000000 0 SECTION LOCAL DEFAULT 4
88 5: 0000000000000000 0 SECTION LOCAL DEFAULT 5
89 6: 0000000000000000 0 SECTION LOCAL DEFAULT 7
90 7: 0000000000000000 0 SECTION LOCAL DEFAULT 8
91 8: 0000000000000000 0 SECTION LOCAL DEFAULT 6
92 9: 0000000000000000 4 OBJECT GLOBAL DEFAULT 3 sleepsecs
93 10: 0000000000000000 129 FUNC  GLOBAL DEFAULT 1 main
94 11: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND _GLOBAL_OFFSET_TABLE_
95 12: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND puts
96 13: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND exit
97 14: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND printf
98 15: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND sleep
99 16: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND getchar

```

图 4.5 符号表

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。

- ①name：字符串表中的字节偏移，指向符号的以 null 结尾的字符串名字，
- ②value：符号的地址，对于可重定位的模块来说，value 是距定义目标的节的起始位置的偏移；对于可执行目标文件来说，value 是一个绝对运行时地址，
- ③size：目标的大小，type 通常要么是数据，要么是函数，
- ④binding：表示符号是本地的还是全局的。
- ⑤ABS：代表不该被重定位的符号，
- ⑥UNDEF：代表未定义的符号，在本目标模块中引用，但在其他地方定义
- ⑦COMMON：表示还未被分配位置的未初始化的数据目标。

4.4 Hello.o 的结果解析

命令: `objdump -d -r hello.o > hello.bjdump`

hello.s	hello.bjdump
19 main:	7 0000000000000000 <main>:
20 .LFB5:	8 0: 55 push %rbp
21 .cfi_startproc	9 1: 48 89 e5 mov %rsp,%rbp
22 pushq %rbp	10 4: 48 83 ec 20 sub \$0x20,%rsp
23 .cfi_def_cfa_offset 16	11 8: 89 7d ec mov %edi,-0x14(%rbp)
24 .cfi_offset 6, -16	12 b: 48 89 75 e0 mov %rsi,-0x20(%rbp)
25 movq %rsp, %rbp	13 f: 83 7d ec 03 cmpl \$0x3,-0x14(%rbp)
26 .cfi_def_cfa_register 6	14 13: 74 16 je 2b <main+0x2b>
27 subq \$32, %rsp	15 15: 48 8d 3d 00 00 00 00 lea 0x0(%rip),%rdi # 1c <main+0x1c>
28 movl %edi, -20(%rbp)	16 18: R_X86_64_PC32 .rodata-0x4
29 movq %rsi, -32(%rbp)	17 1c: e8 00 00 00 00 callq 21 <main+0x21>
30 cmpl \$3, -20(%rbp)	18 1d: R_X86_64_PLT32 puts-0x4
31 je .L2	19 21: bf 01 00 00 00 mov \$0x1,%edi
32 leaq .LC0(%rip), %rdi	20 26: e8 00 00 00 00 callq 2b <main+0x2b>
33 call puts@PLT	21 27: R_X86_64_PLT32 exit-0x4
34 movl \$1, %edi	22 2b: c7 45 fc 00 00 00 00 movl \$0x0,-0x4(%rbp)
35 call exit@PLT	23 32: eb 3b jmp 6f <main+0x6f>
36 .L2:	24 34: 48 8b 45 e0 mov -0x20(%rbp),%rax
37 movl \$0, -4(%rbp)	25 38: 48 83 c0 10 add \$0x10,%rax
38 jmp .L3	26 3c: 48 8b 10 mov (%rax),%rdx
39 .L4:	27 3f: 48 8b 45 e0 mov -0x20(%rbp),%rax
40 movq -32(%rbp), %rax	28 43: 48 83 c0 08 add \$0x8,%rax
41 addq \$16, %rax	29 47: 48 8b 00 mov (%rax),%rax
42 movq (%rax), %rdx	30 4a: 48 89 c6 mov %rax,%rsi
43 movq -32(%rbp), %rax	31 4d: 48 8d 3d 00 00 00 00 lea 0x0(%rip),%rdi # 54 <main+0x54>
44 addq \$8, %rax	32 50: R_X86_64_PC32 .rodata+0x21
45 movq (%rax), %rax	33 54: b8 00 00 00 00 mov \$0x0,%eax
46 movq %rax, %rsi	34 59: e8 00 00 00 00 callq 5e <main+0x5e>
47 leaq .LC1(%rip), %rdi	35 5a: R_X86_64_PLT32 printf-0x4
48 movl \$0, %eax	36 5e: 8b 05 00 00 00 00 mov 0x0(%rip),%eax # 64 <main+0x64>
49 call printf@PLT	37 60: R_X86_64_PC32 sleepsecs-0x4
50 movl sleepsecs(%rip), %eax	38 64: 89 c7 mov %eax,%edi
51 movl %eax, %edi	39 66: e8 00 00 00 00 callq 6b <main+0x6b>
52 call sleep@PLT	40 67: R_X86_64_PLT32 sleep-0x4
53 addl \$1, -4(%rbp)	41 6b: 83 45 fc 01 addl \$0x1,-0x4(%rbp)
54 .L3:	42 6f: 83 7d fc 09 cmpl \$0x9,-0x4(%rbp)
55 cmpl \$9, -4(%rbp)	43 73: 7e bf jle 34 <main+0x34>
56 jle .L4	44 75: e8 00 00 00 00 callq 7a <main+0x7a>

图 4.6 hello.s 与 hello.objdump 的对比

差异分析:

- 1、前奏多了一串数字: 冒号前面的是运行时候的机器指令的位置, 冒号后面, 是每一行汇编语句所对应的机器指令;
- 2、操作数: 在 hello.s 里面是十进制→在 hello.o 里面的是十六进制;
- 3、条件分支: 原来对应的符号→相对偏移地址。
- 4、函数调用: 原来的函数名字→函数的相对偏移地址。

4.5 本章小结

简要介绍汇编, 用 `readelf` 看 elf、`objdump` 看反汇编、对比分析汇编特点

(第 4 章 1 分)

第 5 章 链接

5.1 链接的概念与作用

5.1.1 概念:

链接是将各种代码和数据片段收集并组合成为一个单一文件的过程，这个文件可被加载到内存并执行。现代系统中，链接由链接器程序自动执行。

5.1.2 作用:

链接器使得分离编译成为可能，把大化小，减少重复工作，极大提高了效率。GCC 编译器驱动程序把源文件翻译成可执行目标文件的过程中的第 4 个阶段(末阶段)

5.2 在 Ubuntu 下链接的命令

命令:

```
ld -o hello -dynamic-linker /lib64/ld-linux-x86-64.so.2
/usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o hello.o
/usr/lib/x86_64-linux-gnu/libc.so /usr/lib/x86_64-linux-gnu/crtn.o
```

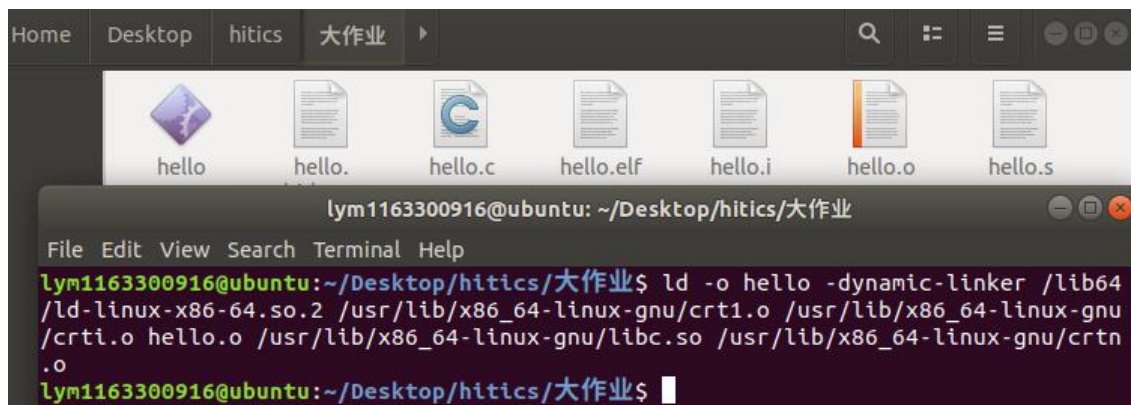


图 5.0 在 Ubuntu 下链接的命令

5.3 可执行目标文件 hello 的格式

命令: readelf -a hello > hello.elf2

hello.elf2							
22	Section Headers:						
23	[Nr]	Name	Type	Address	Offset		
24		Size	EntSize	Flags Link Info	Align		
25	[0]		NULL	0000000000000000	00000000		
26		0000000000000000	0000000000000000	0	0	0	
27	[1]	.interp	PROGBITS	0000000000400200	00000200		
28		000000000000001c	0000000000000000	A	0	0	1
29	[2]	.note.ABI-tag	NOTE	000000000040021c	0000021c		
30		0000000000000020	0000000000000000	A	0	0	4
31	[3]	.hash	HASH	0000000000400240	00000240		
32		0000000000000034	0000000000000004	A	5	0	8
33	[4]	.gnu.hash	GNU_HASH	0000000000400278	00000278		
34		000000000000001c	0000000000000000	A	5	0	8
35	[5]	.dynsym	DYNSYM	0000000000400298	00000298		
36		00000000000000c0	0000000000000018	A	6	1	8
37	[6]	.dynstr	STRTAB	0000000000400358	00000358		
38		0000000000000057	0000000000000000	A	0	0	1
39	[7]	.gnu.version	VERSYM	00000000004003b0	000003b0		
40		0000000000000010	0000000000000002	A	5	0	2
41	[8]	.gnu.version_r	VERNEED	00000000004003c0	000003c0		
42		0000000000000020	0000000000000000	A	6	1	8
43	[9]	.rela.dyn	RELA	00000000004003e0	000003e0		
44		0000000000000030	0000000000000018	A	5	0	8
45	[10]	.rela.plt	RELA	0000000000400410	00000410		
46		0000000000000078	0000000000000018	AI	5	19	8
47	[11]	.init	PROGBITS	0000000000400488	00000488		
48		0000000000000017	0000000000000000	AX	0	0	4
49	[12]	.plt	PROGBITS	00000000004004a0	000004a0		
50		0000000000000060	0000000000000010	AX	0	0	16
51	[13]	.text	PROGBITS	0000000000400500	00000500		
52		0000000000000132	0000000000000000	AX	0	0	16
53	[14]	.fini	PROGBITS	0000000000400634	00000634		
54		0000000000000009	0000000000000000	AX	0	0	4
55	[15]	.rodata	PROGBITS	0000000000400640	00000640		
56		000000000000003a	0000000000000000	A	0	0	8
57	[16]	.eh_frame	PROGBITS	0000000000400680	00000680		
58		00000000000000fc	0000000000000000	A	0	0	8
59	[17]	.dynamic	DYNAMIC	0000000000600e50	00000e50		

图 5.1 elf 表各段基本信息

5.4 hello 的虚拟地址空间

通过 elf 表中的 Address 值, 在 edb 的 Data Dump 可以找到相应区域:

Data Dump

+ 0x0000000000400000-0x0000000000401000

00000000:00400000	7f 45 4c 46 02 01 01 00	..ELF....
00000000:00400008	00 00 00 00 00 00 00 00
00000000:00400010	02 00 3e 00 01 00 00 00	...>....
00000000:00400018	00 05 40 00 00 00 00 00	..@.....
00000000:00400020	40 00 00 00 00 00 00 00	@.....
00000000:00400028	28 17 00 00 00 00 00 00	(.....
00000000:00400030	00 00 00 00 40 00 38 00@.8.
00000000:00400038	08 00 40 00 19 00 18 00	..@.....
00000000:00400040	06 00 00 00 04 00 00 00
00000000:00400048	40 00 00 00 00 00 00 00	@.....

图 5.2 ELF 头(起始地址 0x400000)

Data Dump

+ 0x0000000000400000-0x0000000000401000

00000000:00400200	2f 6c 69 62 36 34 2f 6c	/lib64/l
00000000:00400208	64 2d 6c 69 6e 75 78 2d	d-linux-
00000000:00400210	78 38 36 2d 36 34 2e 73	x86-64.s
00000000:00400218	6f 2e 32 00 04 00 00 00	o.2....
00000000:00400220	10 00 00 00 01 00 00 00
00000000:00400228	47 4e 55 00 00 00 00 00	GNU.....
00000000:00400230	03 00 00 00 02 00 00 00
00000000:00400238	00 00 00 00 00 00 00 00
00000000:00400240	03 00 00 00 08 00 00 00

图 5.3 .interp 节(起始地址 0x400200)

Data Dump

+ 0x0000000000400000-0x0000000000401000

00000000:00400358	00 6c 69 62 63 2e 73 6f	.libc.so
00000000:00400360	2e 36 00 65 78 69 74 00	.6.exit.
00000000:00400368	70 75 74 73 00 70 72 69	puts.pri
00000000:00400370	6e 74 66 00 67 65 74 63	ntf.getc
00000000:00400378	68 61 72 00 73 6c 65 65	har.slee
00000000:00400380	70 00 5f 5f 6c 69 62 63	p.__libc
00000000:00400388	5f 73 74 61 72 74 5f 6d	__start_m
00000000:00400390	61 69 6e 00 47 4c 49 42	ain.GLIB
00000000:00400398	43 5f 32 2e 32 2e 35 00	C.2.2.5.
00000000:004003a0	5f 5f 67 6d 6f 6e 5f 73	__gmon_s
00000000:004003a8	74 61 72 74 5f 5f 00 00	tart__...
00000000:004003b0	00 00 02 00 02 00 02 00

图 5.4 .dynstr 节(起始地址 0x400358)

Data Dump

+ 0x0000000000400000-0x0000000000401000

00000000:00400640	01 00 02 00 00 00 00 00
00000000:00400648	55 73 61 67 65 3a 20 48	Usage: H
00000000:00400650	65 6c 6c 6f 20 31 31 36	ello 116
00000000:00400658	33 33 30 30 39 31 36 20	3300916
00000000:00400660	e6 9d 8e e4 b8 80 e9 b8	%. x.+z
00000000:00400668	a3 ef bc 81 00 48 65 6c	u[]E..Hel
00000000:00400670	6c 6f 20 25 73 20 25 73	lo %s %s
00000000:00400678	0a 00 00 00 00 00 00 00

图 5.5 .rodata(起始地址 0x400640)

5.5 链接的重定位过程分析

1、命令：objdump -d -r hello > hello.objdump2

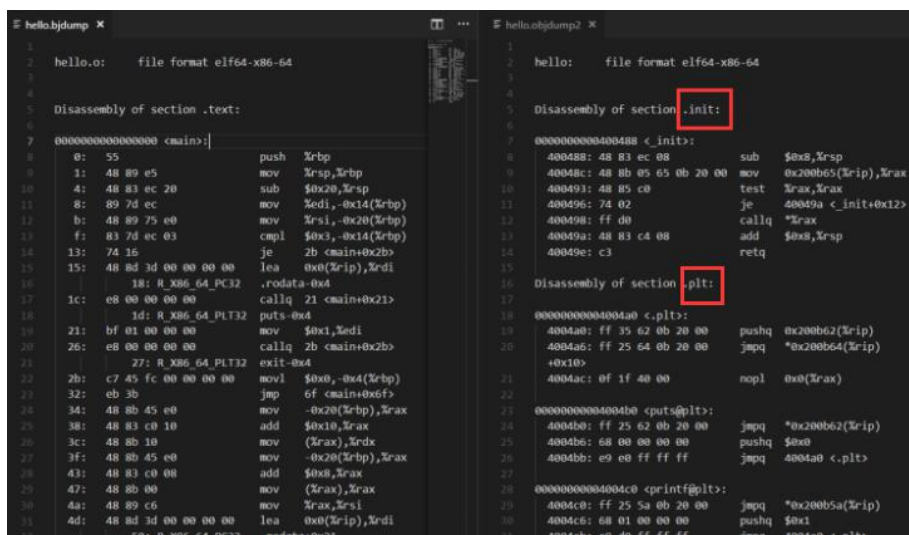


图 5.6 链接前后反汇编的变化 1

变化：①相比之前，加入了新节 .init、.plt 等

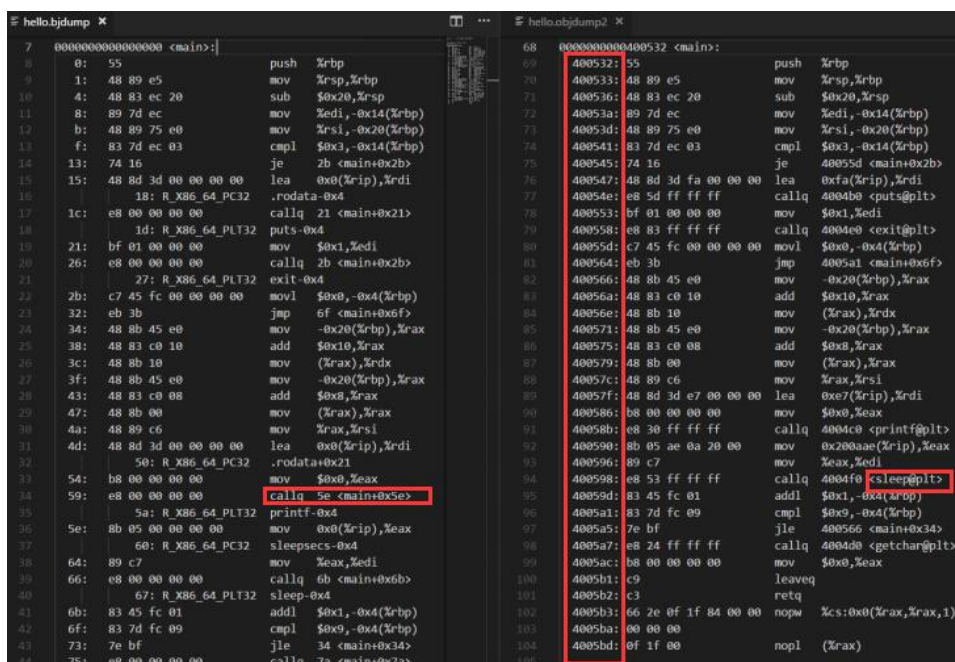


图 5.7 链接前后反汇编的变化 2

其他变化：

②相对偏移地址→虚拟内存地址

③增加了许多的外部链接来的函数。

④原文件中跳转以及函数调用的地址被更换成了虚拟内存地址。

2、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` 的过程中：

①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。

②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。

③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

使用 `edb` 执行 `hello`，说明从加载 `hello` 到 `_start`，到 `call main`，以及程序终止的所有过程。请列出其调用与跳转的各个子程序名 或 程序地址。

```

edb - /mnt/hgfs/hitcs-2/大作业/h
File View Debug Plugins Options Help
No Analysis Fo

00007f1f:c381a090 48 89 e7 movq %rsp, %rdi
00007f1f:c381a093 e8 08 0e 00 00 callq ld-2.27.so!_dl_start
00007f1f:c381a098 49 89 c4 movq %rax, %r12
00007f1f:c381a09b 8b 05 97 66 22 00 movl 0x226697(%rip), %eax
00007f1f:c381a0a1 5a popq %rdx
00007f1f:c381a0a2 48 8d 24 c4 leaq (%rsp, %rax, 8), %rsp
00007f1f:c381a0a6 29 c2 subl %eax, %edx
00007f1f:c381a0a8 52 pushq %rdx
00007f1f:c381a0a9 48 89 d6 movq %rdx, %rsi
00007f1f:c381a0ac 49 89 e5 movq %rsp, %r13
00007f1f:c381a0af 48 83 e4 f0 andq $0xffffffff0, %rsp
00007f1f:c381a0b3 48 8b 3d a6 6f 22 00 movq 0x226fa6(%rip), %rdi
00007f1f:c381a0ba 49 8d 4c d5 10 leaq 0x10(%r13, %rdx, 8), %rcx
00007f1f:c381a0bf 49 8d 55 08 leaq 8(%r13), %rdx
00007f1f:c381a0c3 31 ed xorl %ebp, %ebp
00007f1f:c381a0c5 e8 66 f5 00 00 callq ld-2.27.so!_dl_init
00007f1f:c381a0ca 48 8d 15 cf f8 00 00 leaq 0xf8cf(%rip), %rdx
00007f1f:c381a0d1 4c 89 ec movq %r13, %rsp
00007f1f:c381a0d4 41 ff e4 jmpq *%r12

0x7f1fc381aea0 = 0x00007f1fc381aea0 <ld-2.27.so!_dl_start+0>

```

图 5.8 用 `edb` 进行调试，记录调整信息

加载程序

```
ld-2.23.so!_dl_start
ld-2.23.so!_dl_init
LinkAddress!_start
ld-2.23.so!_libc_start_main
ld-2.23.so!_cxa_atexit
LinkAddress!_libc_csu.init
ld-2.23.so!_setjmp
```

运行

```
LinkAddress!main
```

程序终止

```
ld-2.23.so!exit
```

5.7 Hello 的动态链接分析

初始时，地址 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。

说明 dl_init 操作是给程序赋上当前执行的内存地址偏移量

以下是运行截图：

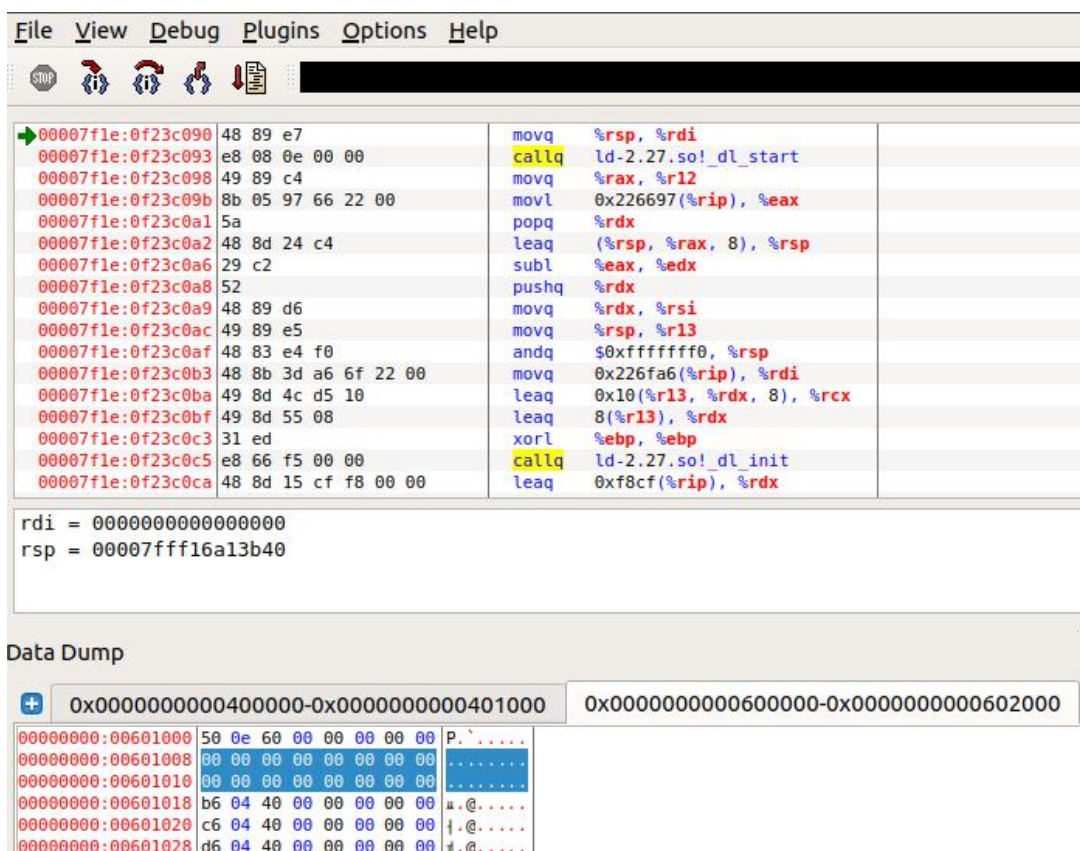


图 5.9 动态链接前的 Data Dump，全 0

The screenshot shows a debugger window with the following components:

- Assembly View:** A list of instructions with their addresses and hex values. The instruction at address 00007f1e:0f23c0ca is highlighted with a green arrow.

Address	Hex	Assembly
00007f1e:0f23c090	48 89 e7	movq %rsp, %rdi
00007f1e:0f23c093	e8 08 0e 00 00	callq ld-2.27.so!_dl_start
00007f1e:0f23c098	49 89 c4	movq %rax, %r12
00007f1e:0f23c09b	8b 05 97 66 22 00	movl 0x226697(%rip), %eax
00007f1e:0f23c0a1	5a	popq %rdx
00007f1e:0f23c0a2	48 8d 24 c4	leaq (%rsp, %rax, 8), %rsp
00007f1e:0f23c0a6	29 c2	subl %eax, %edx
00007f1e:0f23c0a8	52	pushq %rdx
00007f1e:0f23c0a9	48 89 d6	movq %rdx, %rsi
00007f1e:0f23c0ac	49 89 e5	movq %rsp, %r13
00007f1e:0f23c0af	48 83 e4 f0	andq \$0xfffffffff0, %rsp
00007f1e:0f23c0b3	48 8b 3d a6 6f 22 00	movq 0x226fa6(%rip), %rdi
00007f1e:0f23c0ba	49 8d 4c d5 10	leaq 0x10(%r13, %rdx, 8), %rcx
00007f1e:0f23c0bf	49 8d 55 08	leaq 8(%r13), %rdx
00007f1e:0f23c0c3	31 ed	xorl %ebp, %ebp
00007f1e:0f23c0c5	e8 66 f5 00 00	callq ld-2.27.so!_dl_init
00007f1e:0f23c0ca	48 8d 15 cf f8 00 00	leaq 0xf8cf(%rip), %rdx
- Registers:** Below the assembly view, it shows the current values of registers:


```
0xf8cf(%rip) = [0x00007f1e0f24b9a0] = 0x56415741e5894855
rdx = 0000000000000000
```
- Data Dump:** A section showing memory dump with two selected ranges:
 - 0x0000000000040000-0x00000000000401000
 - 0x0000000000060000-0x00000000000602000
 The dump shows several lines of memory data with their addresses and hex values.

图 5.10 动态链接后的 Data Dump，内容变化

5.8 本章小结

介绍了链接、elf 表、虚拟地址空间、重定位过程、hello 执行流程、动态链接

(第 5 章 1 分)

第 6 章 hello 进程管理

6.1 进程的概念与作用

6.1.1 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。

6.1.2 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，好像这个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

6.2.1 作用：

shell-bash 是一个 C 语言程序，是用户使用 Unix/Linux 的桥梁，它交互性地解释和执行用户输入的命令。

bash 还提供了一个图形化界面，提升交互的速度。

6.2.2 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

在 shell 输入 ./hello 1163300916 李一鸣

shell 会分析这一串命令：

1、先判断 ./hello 不是内置命令，

2、然后 shell 调用 fork() 函数，创建一个子进程，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。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。

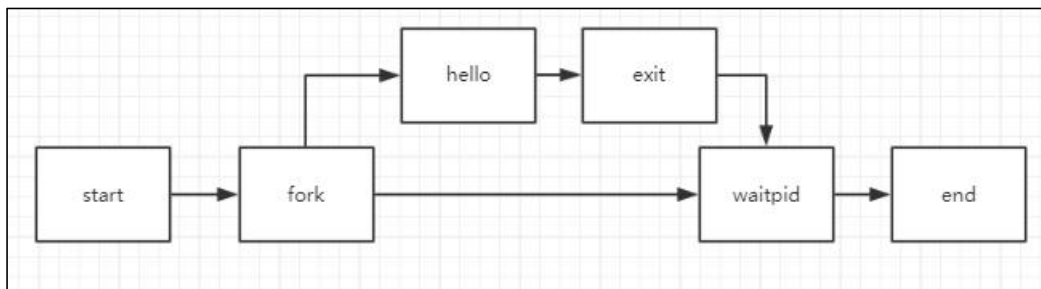


图 6.0 fork 流程

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。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。execve 有 3 个参数，第一个参数为文件名，这里就是 Hello 文件，第二个和第三个是传入 Hello 的 main 函数的参数以及参数个数。

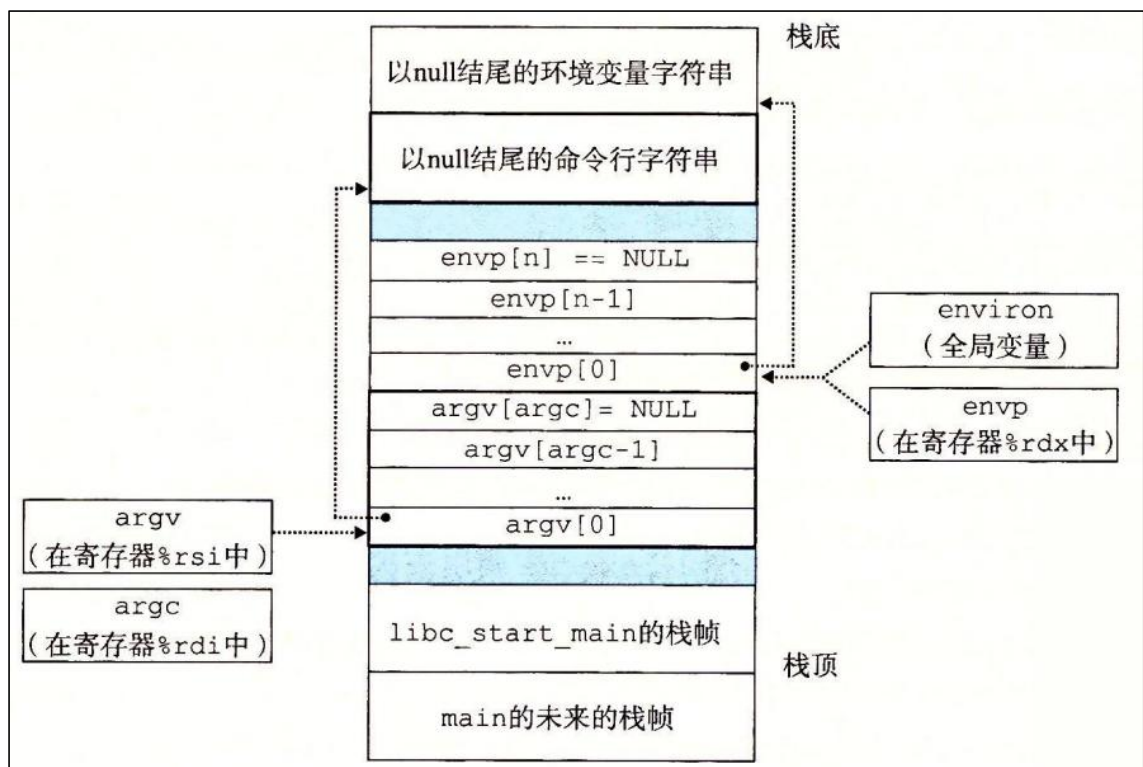


图 6.1 execve 开始时的用户栈

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。

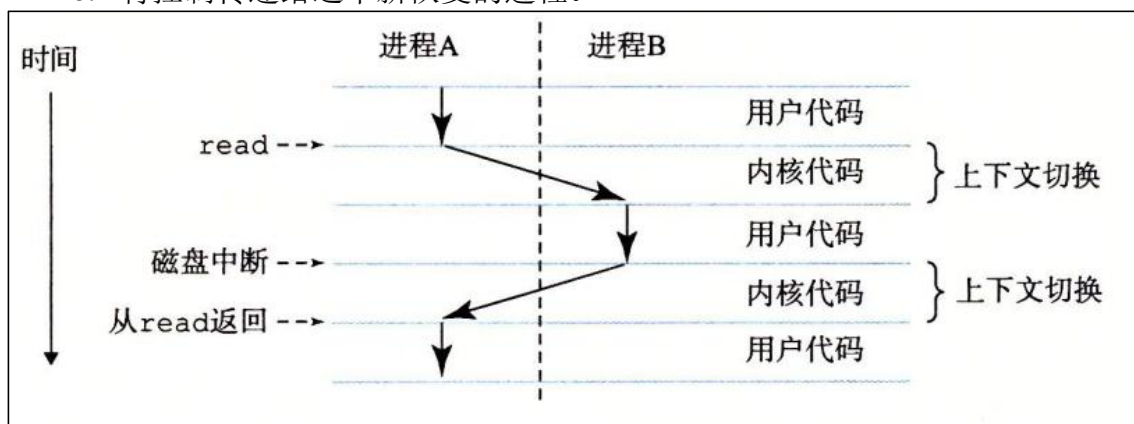


图 6.2 上下文切换

Hello 执行过程中，调用 printf 时，会调用系统的 write 函数，内核把控制从用户态传到核心态，输出结束之后控制交还给 main，从核心态恢复到用户态

6.6 hello 的异常与信号处理

hello 执行过程中会出现哪几类异常，会产生哪些信号，又怎么处理的。

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

```
File Edit View Search Terminal Help
lym1163300916@ubuntu:~/Desktop/hitics/大作业$ ./hello
Usage: Hello 1163300916 李一鸣！
lym1163300916@ubuntu:~/Desktop/hitics/大作业$ ./hello 1163300916 李一鸣
Hello 1163300916 李一鸣
Hello 1163300916 李一鸣
kajdf
Hello 1163300916 李一鸣
sd,mn
Hello 1163300916 李一鸣
555
Hello 1163300916 李一鸣
233
Hello 1163300916 李一鸣
9284rujmcx^[l Hello 1163300916 李一鸣
Hello 1163300916 李一鸣
^C
lym1163300916@ubuntu:~/Desktop/hitics/大作业$
```

图 6.3 乱按，不影响程序执行

```

lym1163300916@ubuntu:~/Desktop/hitcs/大作业$ ./hello 1163300916 李一鸣
Hello 1163300916 李一鸣
Hello 1163300916 李一鸣
^Z
[1]+  Stopped                  ./hello 1163300916 李一鸣
lym1163300916@ubuntu:~/Desktop/hitcs/大作业$ ps
  PID TTY          TIME CMD
  8154 pts/0        00:00:00 bash
  8180 pts/0        00:00:00 hello
  8217 pts/0        00:00:00 ps
lym1163300916@ubuntu:~/Desktop/hitcs/大作业$ jobs
[1]+  Stopped                  ./hello 1163300916 李一鸣
lym1163300916@ubuntu:~/Desktop/hitcs/大作业$ fg
./hello 1163300916 李一鸣
Hello 1163300916 李一鸣
Hello 1163300916 李一鸣
Hello 1163300916 李一鸣

Hello 1163300916 李一鸣
Hello 1163300916 李一鸣

```

图 6.4 ps jobs fg 恢复前台

```

lym1163300916@ubuntu:~/Desktop/hitcs/大作业$ fg
./hello 1163300916 李一鸣
Hello 1163300916 李一鸣
^Z
[1]+  Stopped                  ./hello 1163300916 李一鸣
lym1163300916@ubuntu:~/Desktop/hitcs/大作业$ pstree
systemd--ModemManager--2*[{ModemManager}]
        |
        |--NetworkManager--dhclient
        |                   2*[{NetworkManager}]
        |--VGAAuthService
        |--accounts-daemon--2*[{accounts-daemon}]
        |--acpid
        |--avahi-daemon--avahi-daemon
        |--bluetoothd
        |--boltd--2*[{boltd}]
        |--colord--2*[{colord}]
        |--cron
        |--cups-browsed--2*[{cups-browsed}]
        |--cupsd
        |--2*[dbus-daemon]
        |--fcitx--{fcitx}
        |--fcitx-dbus-watc
        |--fwupd--4*[{fwupd}]
        |--gdm3--gdm-session-wor--gdm-wayland-ses--gnome-session-b--gnome-sh+
        |                                     |
        |                                     |--gsd-a11y+
        |                                     |--gsd-clip+
        |                                     |--gsd-colo+
        |                                     |--gsd-date+
        |                                     |--gsd-hous+
        |                                     |--gsd-keyb+
        |                                     |--gsd-medi+
        |                                     |--gsd-mous+
        |                                     |--gsd-powe+
        |                                     |--gsd-prin+
        |                                     |--gsd-rfki+
        |                                     |--gsd-scre+

```

图 6.5 pstree

```
lym1163300916@ubuntu: ~/Desktop/hitcs/大作业
File Edit View Search Terminal Help
lym1163300916@ubuntu:~/Desktop/hitcs/大作业$ ./hello 1163300916 李一鸣
Hello 1163300916 李一鸣
^Z
[1]+  Stopped                  ./hello 1163300916 李一鸣
lym1163300916@ubuntu:~/Desktop/hitcs/大作业$ ps
  PID TTY          TIME CMD
  8278 pts/0        00:00:00 bash
  8304 pts/0        00:00:00 hello
  8323 pts/0        00:00:00 ps
lym1163300916@ubuntu:~/Desktop/hitcs/大作业$ kill -9 8304
lym1163300916@ubuntu:~/Desktop/hitcs/大作业$ ps
  PID TTY          TIME CMD
  8278 pts/0        00:00:00 bash
  8324 pts/0        00:00:00 ps
[1]+  Killed                  ./hello 1163300916 李一鸣
lym1163300916@ubuntu:~/Desktop/hitcs/大作业$ fg
bash: fg: current: no such job
lym1163300916@ubuntu:~/Desktop/hitcs/大作业$ █
```

图 6.6 kill

6.7 本章小结

本章介绍了进程概念，包括 `fork`、`execve` 等系统函数的调用，以及父进程与子进程之间的关系。

(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

7.1.1 逻辑地址：逻辑地址(LogicalAddress)是指由程序产生的与段相关的偏移地址部分。如 hello.o 里面的相对偏移地址。

7.1.2 线性地址：地址空间(address space) 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(linear address space) 。如 hello 里面的虚拟内存地址。

7.1.3 虚拟地址：CPU 通过生成一个虚拟地址(Virtual Address, VA) 。

如 hello 里面的虚拟内存地址。

7.1.4 物理地址：用于内存芯片级的单元寻址，与处理器和 CPU 连接的地址总线相对应。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。如 hello 在运行时虚拟内存地址对应的物理地址。

■ 逻辑地址空间：段地址：偏移地址

23: 8048000 段寄存器 (CS等16位)：偏移地址 (16/32/64)

实模式下： 逻辑地址CS: EA \Rightarrow 物理地址CS*16+EA

保护模式下：以段描述符作为下标，到GDT/LDT表查表获得段地址，
段地址+偏移地址=线性地址。

■ 线性地址空间：非负整数地址的有序集合：

{0, 1, 2, 3 ... }

■ 虚拟地址空间：N = 2^n 个虚拟地址的集合 ===线性地址空间

{0, 1, 2, 3, ..., N-1}

■ 物理地址空间：M = 2^m 个物理地址的集合

{0, 1, 2, 3, ..., M-1}

■ Intel采用段页式存储管理 (MMU实现)

■ 段式管理： 逻辑地址 \rightarrow 线性地址==虚拟地址

■ 页式管理： 虚拟地址 \rightarrow 物理地址

图 7.0 课程 PPT 中的虚拟空间

7.2 Intel 逻辑地址到线性地址的变换-段式管理

给定一个完整的逻辑地址[段选择符: 段内偏移地址], 步骤:

- ①看段选择符, 再根据相应寄存器, 得到其地址和大小。可以得到一个数组。
- ②取出段选择符中前 13 位, 在数组中查找到对应的段描述符, 得到基地址。
- ③线性地址 = 基址 + 偏移。

每个段有不同的作用:

- 1.SS 栈段寄存器
- 2.ES/GS/FS 辅助段寄存器
- 3.DS 数据段寄存器
- 4.CS 代码段寄存器

原理

 编辑

为了进行段式管理, 每道程序在系统中都有一个段(映象)表来存放该道程序各段装入主存的情况信息。段表中的每一项(对应表中的每一行)描述该道程序一个段的基本状况, 由若干个字段提供。段名字段用于存放段的名称, 段名一般是有其逻辑意义的, 也可以转换成用段号指明。由于段号从0开始顺序编号, 正好与段表中的行号对应, 如2段必是段表中的第3行, 这样, 段表中就可不设段号(名)字段。装入位字段用来指示该段是否已经调入主存, “1”表示已装入, “0”表示未装入。在程序的执行过程中, 各段的装入位随该段是否活跃而动态变化。当装入位为“1”时, 地址字段用于表示该段装入主存中起始(绝对)地址, 当装入位为“0”时, 则无效(有时机器用它表示该段在辅存中的起始地址)。段长字段指明该段的大小, 一般以字数或字节数为单位, 取决于所用的编址方式。段长字段是用来判断所访问的地址是否越出段界的界限保护检查用的。访问方式字段用来标记该段允许的访问方式, 如只读、可写、只能执行等, 以提供段的访问方式保护。除此之外, 段表中还可以根据需要设置其它的字段。段表本身也是一个段, 一般常驻在主存中, 也可以存在辅存中, 需要时再调入主存。假设系统在主存中最多可同时有N道程序, 可设N个段表基址寄存器。对应于每道程序, 由基号(程序号)指明使用哪个段表基址寄存器。段表基址寄存器中的段表基址字段指向该道程序的段表在主存中的起始地址。段表长度字段指明该道程序所用段表的行数, 即程序的段数。

图 7.1 段式管理原理(摘自百度百科)

7.3 Hello 的线性地址到物理地址的变换-页式管理

页式管理的基本原理；数据结构；地址变换：

(1)页式存储管理

1)基本原理。将程序的逻辑地址空间划分为固定大小的页(page)，而物理内存划分为同样大小的页框(pageframe)。程序加载时，可将任意一页放入内存中任意一个页框，这些页框不必连续，从而实现了离散分配。该方法需要CPU的硬件支持，来实现逻辑地址和物理地址之间的映射。在页式存储管理方式中地址结构由两部分构成，前一部分是页号，后一部分为页内地址。

这种管理方式的优点是，没有外碎片，每个内碎片不超过页大小比前面所讨论的几种管理方式的最大进步是，一个程序不必连续存放。这样就便于改变程序占用空间的大小(主要指随着程序运行，动态生成的数据增多，所要求的地址空间相应增长)。缺点是仍旧要求程序全部装入内存，没有足够的内存，程序就不能执行。

2)页式管理的数据结构。在页式系统中进程建立时，操作系统为进程中所有的页分配页框。当进程撤销时收回所有分配给它的页框。在程序的运行期间，如果允许进程动态地申请空间，操作系统还要为进程申请的空间分配物理页框。操作系统为了完成这些功能，必须记录系统内存中

实际的页框使用情况。操作系统还要在进程切换时，正确地切换两个不同的进程地址空间到物理内存空间的映射。这就要求操作系统要记录每个进程页表的相关信息。为了完成上述的功能，一个页式系统中，一般要采用如下的数据结构。

进程页表：完成逻辑页号(本进程的地址空间)到物理页面号(实际内存空间)的映射。

每个进程有一个页表，描述该进程占用的物理页面及逻辑排列顺序。

物理页面表：整个系统有一个物理页面表，描述物理内存空间的分配使用状况，其数据结构可采用位示图和空闲页链表。

请求表：整个系统有一个请求表，描述系统内各个进程页表的位置和大小，用于地址转换也可以结合到各进程的PCB(进程控制块)里。

3) 页式管理地址变换

在页式系统中，指令所给出的地址分为两部分：逻辑页号和页内地址。CPU中的内存管理单元(MMU)按逻辑页号通过查进程页表得到物理页框号，将物理页框号与页内地址相加形成物理地址。上述过程通常由处理器的硬件直接完成，不需要软件参与。通常，操作系统只需在进程切换时，把进程页表的首地址装入处理器特定的寄存器中即可。一般来说，页表存储在主存之中。这样处理器每访问一个在内存中的操作数，就要访问两次内存。第一次用来查找页表将操作数的逻辑地址变换为物理地址；第二次完成真正的读写操作。这样做时间上耗费严重。为缩短查找时间，可以将页表从内存装入CPU内部的关联存储器(例如，快表)中，实现按内容查找。此时的地址变换过程是：在CPU给出有效地址后，由地址变换机构自动将页号送入快表，并将此页号与快表中的所有页号进行比较，而且这种比较是同时进行的。若其中有与此相匹配的页号，表示要访问的页的页表项在快表中。于是可直接读出该页所对应的物理页号，这样就无需访问内存中的页表。由于关联存储器的访问速度比内存的访问速度快得多。

图 7.2 页式管理

网址：<http://www.cnblogs.com/xavierlee/p/6400230.html>

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

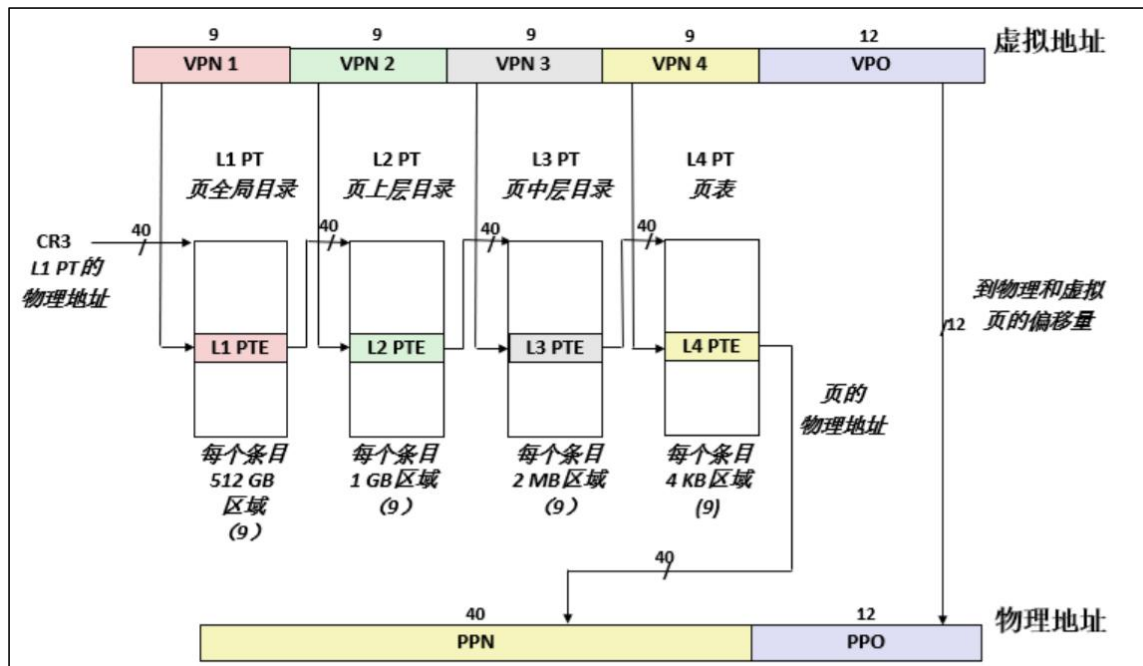


图 7.3 四级页表

Core i7 MMU 如何使用四级的页表来将虚拟地址翻译成物理地址。36 位 VPN 被划分成四个 9 位的片，每个片被用作到一个页表的偏移量。CR3 寄存器包含 L1 页表的物理地址。VPN 1 提供到一个 L1 PTE 的偏移量，这个 PTE 包含 L2 页表的基地址。VPN 2 提供到一个 L2 PTE 的偏移量，以此类推

7.5 三级 Cache 支持下的物理内存访问

得到物理地址之后，先将物理地址拆分成 CT（标记）+CI（索引）+CO（偏移量），然后在一级 cache 内部找，如果未能寻找到标记位为有效的字节（miss）的话就去二级和三级 cache 中寻找对应的字节，找到之后返回结果。

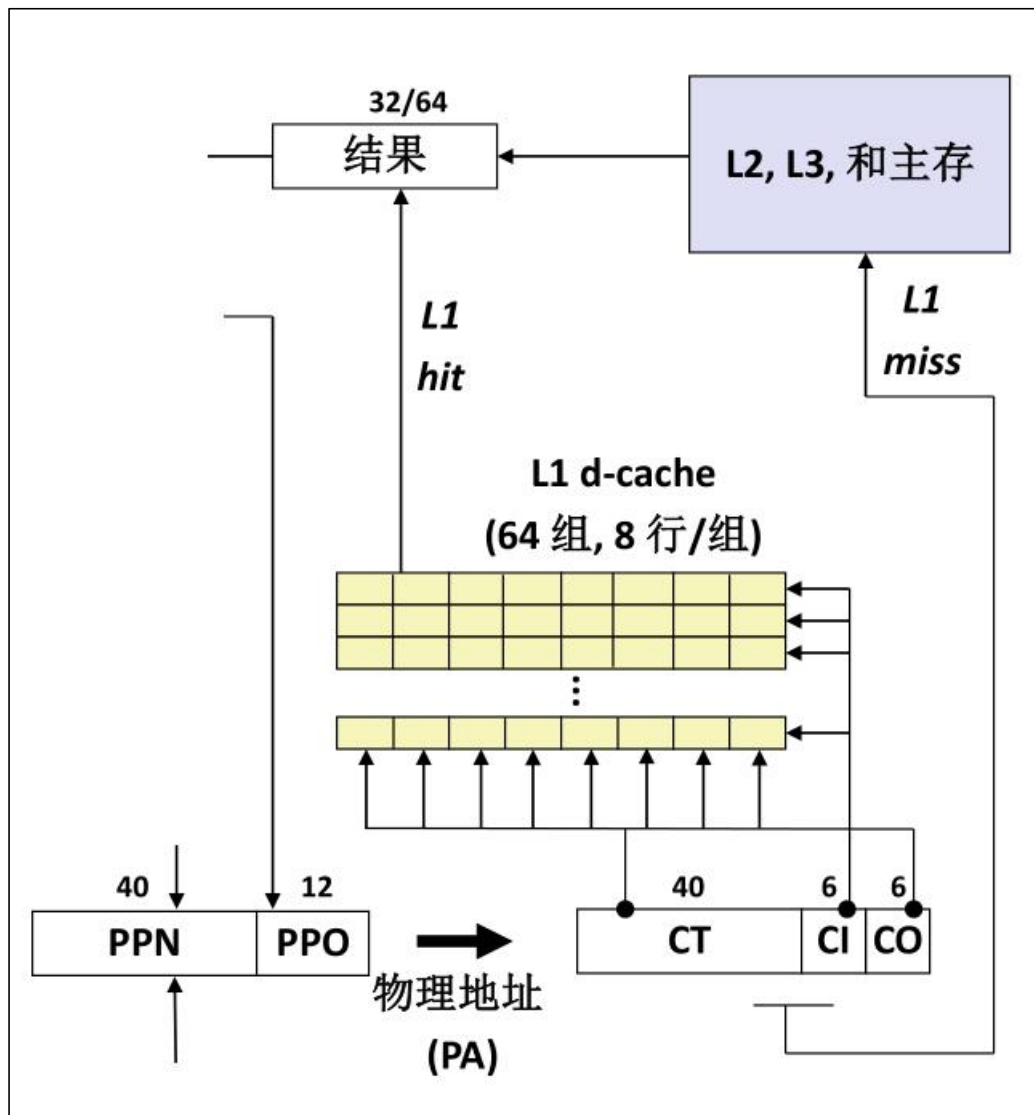


图 7.4 三级 cache

7.6 hello 进程 fork 时的内存映射

当 fork 函数被 shell 调用时，内核为 hello 进程创建各种数据结构，并分配给它一个唯一的 PID。创建了 hello 进程的 mm_struct、区域结构和页表的原样副本。

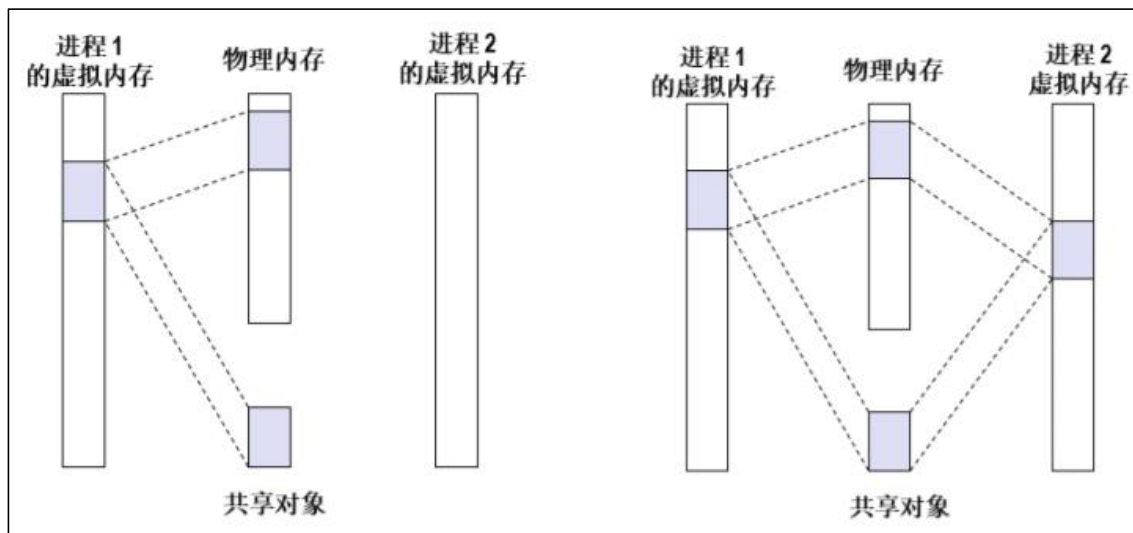


图 7.5 共享对象

将两个进程中的每个页面都标记为只读，并将两个进程中的每个区域结构都标记为私有的写时复制。当这两个进程中的任一个后来进行写操作时，写时复制机制就会创建新页面，因此，也就为每个进程保持了私有地址空间的抽象概念

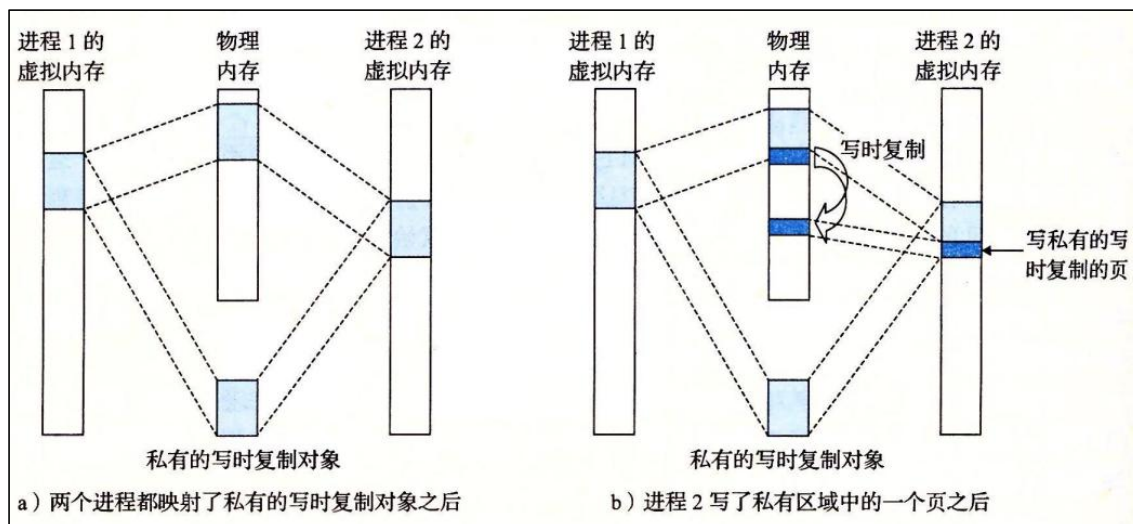


图 7.6 写时复制

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的程序计数器，使之指向代码区域的入口点

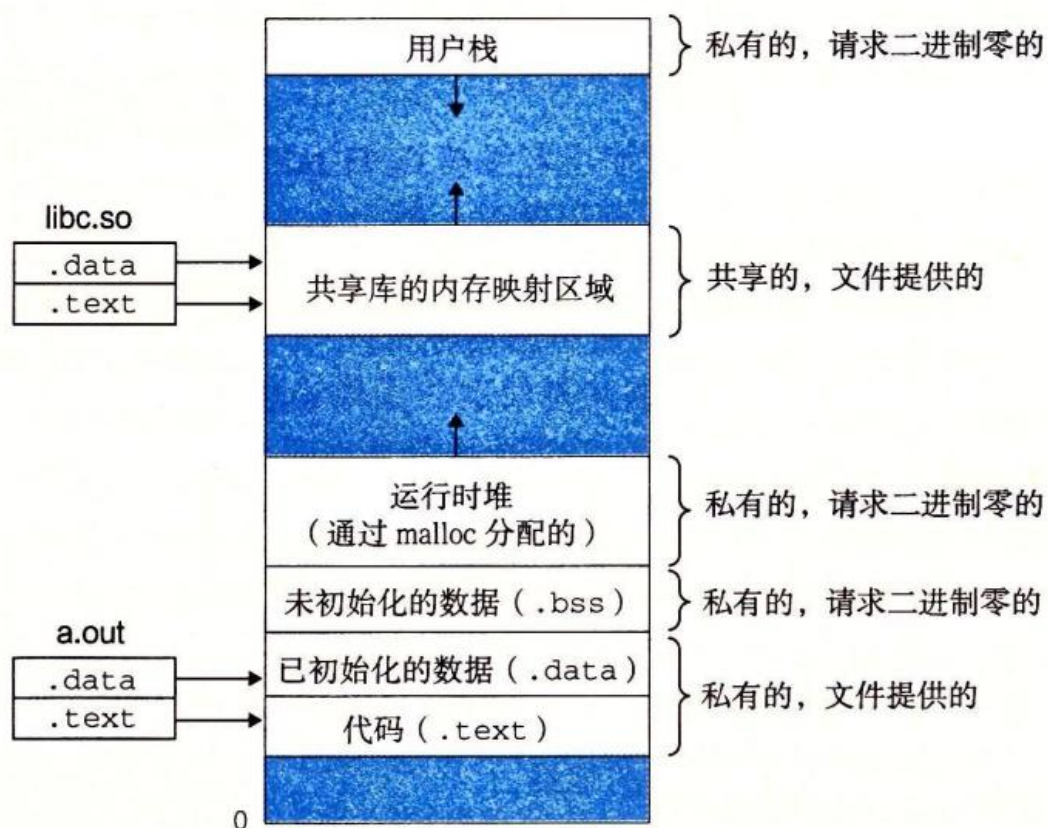


图 7.7 内存情况

7.8 缺页故障与缺页中断处理

缺页概念: DRAM 缓存不命中称为缺页,即虚拟内存中的字不在物理内存中。

缺页故障: 缺页异常触发后, 内核中的缺页异常处理程序会选择一个牺牲页, 然后将其用目标页替换掉, 最后处理程序返回, 重启导致缺页的指令, 该指令会再次访问目标地址, 此时目标地址出就已经有了需要的数据了。

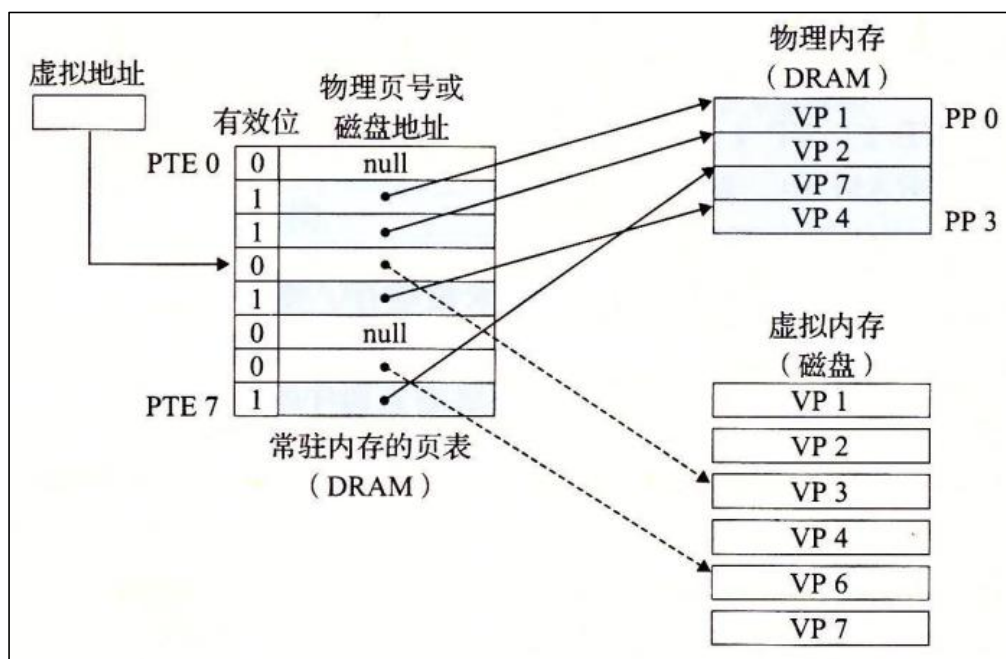


图 7.9 缺页故障

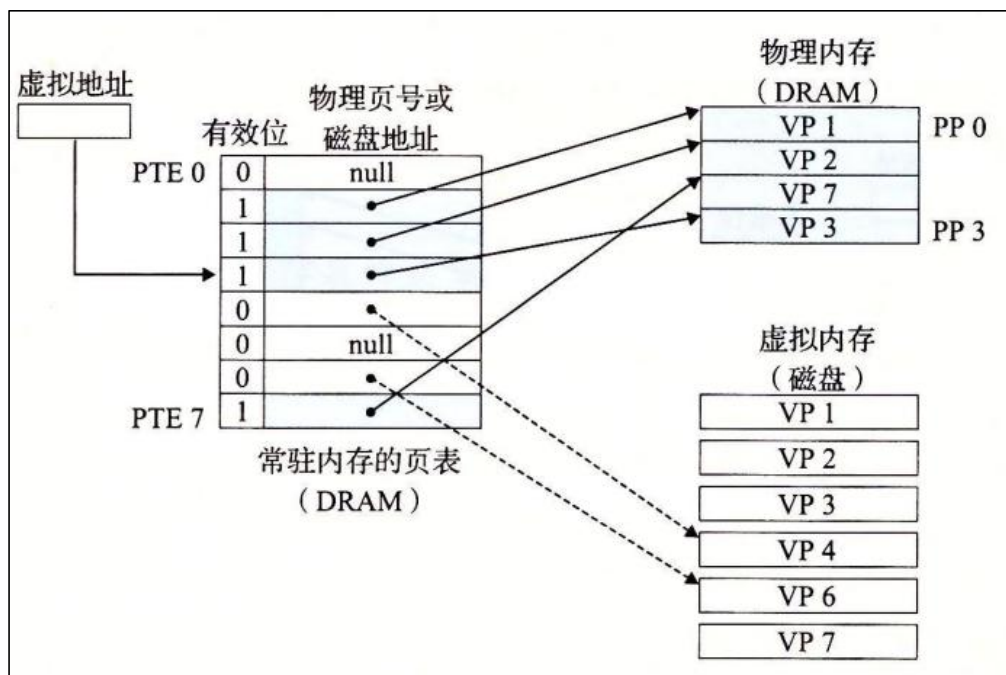


图 7.10 替换牺牲页

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。

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。

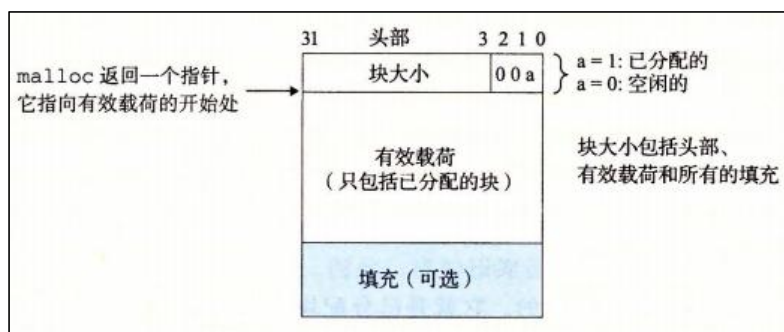


图 7.11 隐式模型

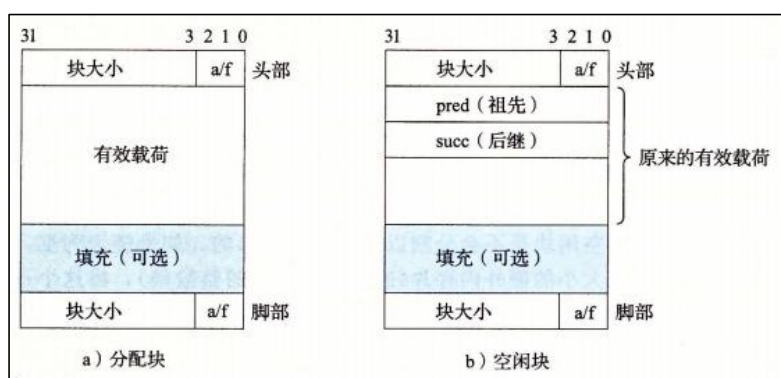


图 7.12 显式模型

7.10 本章小结

简单介绍了段式管理、页式管理、多级页表、内存映射、缺页处理、动态内存分配的知识。在虚拟内存的视角下，重新分析了系统函数调用时 fork、execve 的操作，观测虚拟内存空间的变化。

第 8 章 hello 的 IO 管理

8.1 Linux 的 IO 设备管理方法

设备的模型化：文件

设备管理：unix i/o 接口

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 接口的几种操作：

1. 打开文件：程序要求内核打开文件，内核返回一个小的非负整数（描述符），用于标识这个文件。程序在只要记录这个描述符便能记录打开文件的所有信息。
2. shell 在进程的开始为其打开三个文件：标准输入、标准输出和标准错误。
3. 改变当前文件的位置：对于每个打开的文件，内核保存着一个文件位置 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 。
4. 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 字节的文件，当 $k \geq m$ 时执行读操作会出发一个称为 EOF 的条件，应用程序能检测到这个条件，在文件结尾处并没有明确的 EOF 符号。
5. 关闭文件：内核释放打开文件时创建的数据结构以及占用的内存资源，并将描述符恢复到可用的描述符池中。无论一个进程因为何种原因终止时，内核都会关闭所有打开的文件并释放它们的内存资源。

Unix I/O 函数：

1. `int open(char *filename, int flags, mode_t mode);`

open 函数将 filename 转换为一个文件描述符，并且返回描述符数字。

返回的描述符总是在进程中当前没有打开的最小描述符，flags 参数指

明了进程打算如何访问这个文件，mode 参数指定了新文件的访问权限位。

2. `int close(int fd);`

关闭一个打开的文件。

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

`read` 函数从描述符为 `fd` 的当前文件位置赋值最多 `n` 个字节到内存位置 `buf`。返回值-1 表示一个错误，0 表示 EOF，否则返回值表示的是实际传送的字节数量。

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

`write` 函数从内存位置 `buf` 复制至多 `n` 字节到描述符 `fd` 当前文件位置。

8.3 printf 的实现分析

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

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）。

1、`printf` 的实现：定义一个 `buf`，调用 `vsprintf`

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

2、`vsprintf` 实现：

```
1.  int vsprintf(char *buf, const char *fmt, va_list args)
2.  {
3.      char* p;
4.      char tmp[256];
5.      va_list p_next_arg = args;
6.      for (p=buf; *fmt; fmt++) {
7.          if (*fmt != '%') {
8.              *p++ = *fmt;
9.              continue;
```

```

10.     }
11.
12.     fmt++;
13.     switch (*fmt) {
14.     case 'x':
15.         itoa(tmp, *((int*)p_next_arg));
16.         strcpy(p, tmp);
17.         p_next_arg += 4;
18.         p += strlen(tmp);
19.         break;
20.     case 's':
21.         break;
22.     default:
23.         break;
24.     }
25.     }
26.     return (p - buf);
27. }

```

vsprintf 函数作用是格式化，接受格式化字符串，确定最终输出的字符串，存在 buf 中，最后 write 输出。实际上是在 printf 中，用户传入的内容里含有%d 等内容，要转换成对应的值。

3、write 的调用，会将 buf 内容全部输出。

```

mov eax, _NR_write
mov ebx, [esp + 4]
mov ecx, [esp + 8]
int INT_VECTOR_SYS_CALL

```

参数被放入寄存器，然后调用 syscall

4、调用 sys_call:

```

call save
push dword [p_proc_ready]
sti
push ecx
push ebx
call [sys_call_table + eax * 4]
add esp, 4 * 3
mov [esi + EAXREG - P_STACKBASE], eax
cli
ret

```

①将字符串通过总线复制到显卡的显存中，然后在显存中存储相应编码；

②字符驱动程序在字模库中找到点阵信息，存储到 `vram` 中。

③显示芯片按照刷新频率逐行读取 `vram`，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

介绍了操作系统相关知识：I/O 设备的管理方法，Unix I/O 接口和函数，`printf`、`getchar` 函数通过 Unix I/O 实现的分析

（第 8 章 1 分）

结论

`hello.c` 自身并不能运行，需要操作系统、Unix I/O 的协助，才使 `hello` 有了生命，它的一生如下：

1. 预处理：生成 `hello.i`。
2. 编译：将 `hello.i` 编译成汇编文件 `hello.s`。
3. 汇编：将 `hello.s` 翻译成可重定位目标文件 `hello.o`
4. 链接：将 `hello.o` 与动态链接库链接生成可执行目标文件 `hello`，
`hello` 的运行还需要操作系统的平台支持
5. `shell` 调用 `fork`，为 `hello` 生成子进程，在子进程调用 `execve`，加载运行 `hello`。
6. CPU 为 `hello` 分配内存空间，`hello` 从磁盘中加载到内存。
7. 当 CPU 访问 `hello` 时，请求虚拟地址，
MMU 将虚拟地址转换成物理地址并通过多级 `cache` 访存。
8. `hello` 运行时可能遇到各种信号，`shell` 为其提供了相应的信号处理程序
9. Unix I/O 使 `hello` 可以从键盘输入和输出到屏幕。
10. 最后 `hello` 程序结束；父进程回收，内核对其删除。故事完结。

计算机系统是一门大学问，精密有条理，从 `hello` 程序的运行中可见一斑。`hello` 麻雀虽小，五脏俱全，方便用来理解计算机系统的内部机制。要多花功夫

（结论 0 分，缺失 -1 分，根据内容酌情加分）

附件

 hello.c	源代码文件
 hello.i	预处理过的文件： 解释预处理器的作用
 hello.s	编译过的文件： 解释编译器
 hello.o	汇编生成的可重定位目标文件： 解释汇编器的作用
 hello	最终编译出来的可执行目标文件： 解释链接器的作用以及分析链接过程等

(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] 兰德尔 E.布莱恩特. 深入理解计算机系统. 北京: 机械工业出版社, 2016
- [2] <http://www.cnblogs.com/xavierlee/p/6400230.html>
- [3] 虚拟地址 、 逻辑地址 、 线性地址 、 物理地址 :
https://blog.csdn.net/rabbit_in_android/article/details/49976101
- [4] printf 函数实现的剖析:
https://blog.csdn.net/zhengqijun_/article/details/72454714

(参考文献 0 分, 缺失 -1 分)