

哈爾濱工業大學

# 计算机系统

## 大作业

题	目	程序人生-Hello's P2P
专	业	计算机类
学	号	1170300211
班	级	11703002
学	生	彭湃
指 导 教 师		史先俊

计算机科学与技术学院

2018 年 12 月

## 摘 要

在 ICS 的课堂中，深入介绍了计算机系统各个部分的原理，本文围绕 `hello.c` 的编译执行过程，详细介绍每一个步骤，及其生成的文件，以预处理，编译，汇编，链接，进程管理，存储管理和系统 I/O 为主题，阐释了计算机从底层硬件到操作系统到用户程序的具体实现。

关键词：hello；编译；进程；操作系统；内存管理

（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

<b>第 1 章 概述</b> .....	<b>4 -</b>
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 -
<b>第 2 章 预处理</b> .....	<b>6 -</b>
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.....	8 -
<b>第 3 章 编译</b> .....	<b>9 -</b>
3.1 编译的概念与作用.....	9 -
3.2 在 UBUNTU 下编译的命令 .....	10 -
3.3 HELLO 的编译结果解析.....	10 -
3.3.1 数据.....	10 -
3.3.2 赋值.....	11 -
3.3.3 算术操作.....	12 -
3.3.4 关系操作.....	12 -
3.3.5 数组操作.....	13 -
3.3.6 控制转移.....	13 -
3.3.7 函数操作.....	14 -
3.4 本章小结.....	15 -
<b>第 4 章 汇编</b> .....	<b>16 -</b>
4.1 汇编的概念与作用.....	16 -
4.2 在 UBUNTU 下汇编的命令 .....	16 -
4.3 可重定位目标 ELF 格式 .....	16 -
4.4 HELLO.O 的结果解析 .....	19 -
4.5 本章小结.....	20 -
<b>第 5 章 链接</b> .....	<b>21 -</b>
5.1 链接的概念与作用.....	21 -
5.2 在 UBUNTU 下链接的命令 .....	21 -
5.3 可执行目标文件 HELLO 的格式 .....	21 -
5.4 HELLO 的虚拟地址空间 .....	23 -
5.5 链接的重定位过程分析.....	24 -

5.6 HELLO 的执行流程 .....	- 27 -
5.7 HELLO 的动态链接分析 .....	- 28 -
5.8 本章小结 .....	- 30 -
<b>第 6 章 HELLO 进程管理 .....</b>	<b>- 31 -</b>
6.1 进程的概念与作用 .....	- 31 -
6.2 简述壳 SHELL-BASH 的作用与处理流程 .....	- 31 -
6.3 HELLO 的 FORK 进程创建过程 .....	- 31 -
6.4 HELLO 的 EXECVE 过程 .....	- 31 -
6.5 HELLO 的进程执行 .....	- 32 -
6.6 HELLO 的异常与信号处理 .....	- 32 -
6.7 本章小结 .....	- 34 -
<b>第 7 章 HELLO 的存储管理 .....</b>	<b>- 35 -</b>
7.1 HELLO 的存储器地址空间 .....	- 35 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理 .....	- 35 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理 .....	- 36 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换 .....	- 37 -
7.5 三级 CACHE 支持下的物理内存访问 .....	- 37 -
7.6 HELLO 进程 FORK 时的内存映射 .....	- 39 -
7.7 HELLO 进程 EXECVE 时的内存映射 .....	- 39 -
7.8 缺页故障与缺页中断处理 .....	- 39 -
7.9 动态存储分配管理 .....	- 40 -
7.10 本章小结 .....	- 41 -
<b>第 8 章 HELLO 的 IO 管理 .....</b>	<b>- 42 -</b>
8.1 LINUX 的 IO 设备管理方法 .....	- 42 -
8.2 简述 UNIX IO 接口及其函数 .....	- 43 -
8.3 PRINTF 的实现分析 .....	- 44 -
8.4 GETCHAR 的实现分析 .....	- 49 -
8.5 本章小结 .....	- 49 -
<b>结论 .....</b>	<b>- 50 -</b>
<b>附件 .....</b>	<b>- 51 -</b>
<b>参考文献 .....</b>	<b>- 52 -</b>

# 第 1 章 概述

## 1.1 Hello 简介

P2P 过程指的是 hello 从 .c 的源文件(program)到加载在内存中的进程(process)的过程, O2O 过程指的是 hello 从储存在磁盘上的代码(内存占用为 0), 到结束进程(内存占用被释放为 0)的过程。接下来将使用计算机的术语详细解释这一过程。

首先 hello.c 通过 I/O 设备如键盘等经过总线存入主存。完成后的 hello.c 文件依次经过编译器的预处理对源代码进行转换、编译得到汇编语言代码、汇编再将汇编语言转换为机器语言, 最后与库函数进行链接并进行重定位, 形成可执行文件。然后 shell 调用 fork 函数创建一个新运行的子进程, 在子进程中 hello 加载并运行。加载器删除子进程现有的虚拟内存段, 然后使用 mmap 函数创建新的内存区域, 并创建一组新的代码、数据、堆和栈段。hello 作为单独的进程, 享有抽象的虚拟地址空间和完整的 cpu 占用, 操作系统通过上下文切换为进程提供这种抽象。其中 TLB 和分级页表大大加速了地址翻译的过程, 计算机系统各抽象级中广泛存在的 cache 减少了读取数据所需要的时间。当需要结束 hello 进程的时候, 操作系统通过终端重新获得控制, 并向 hello 发送 SIGINT 信号, hello 进程结束, 释放内存空间。

## 1.2 环境与工具

### 1. 硬件环境:

Intel i5 CPU @2.8GHz; 8GB RAM; 256GB SSD

### 2. 软件环境:

Windows 10 64 位操作系统; VMware 14; Ubuntu 16.04 LTS;

### 3. 开发与调试工具:

gcc; edb; readelf; objdump; gedit;

## 1.3 中间结果

文件	注释
hello.i	hello
hello.s	hello 汇编代码文件
hello.o	hello 可重定位目标文件
hello	可执行文件
objectdump	hello.o 的反汇编文件
exedump	hello 的反汇编文件

objectelf	hello.o 的 elf 格式
exeelf	hello 的 elf 格式

## 1.4 本章小结

本章给出了 hello 的从源文件到运行到结束的全过程，描述了 P2P 和 020 的具体含义，并且给出了完成作业的硬件环境以及软件工具，并列出了完成本次实验所产生的全部中间文件。

**(第 1 章 0.5 分)**

## 第 2 章 预处理

### 2.1 预处理的概念与作用

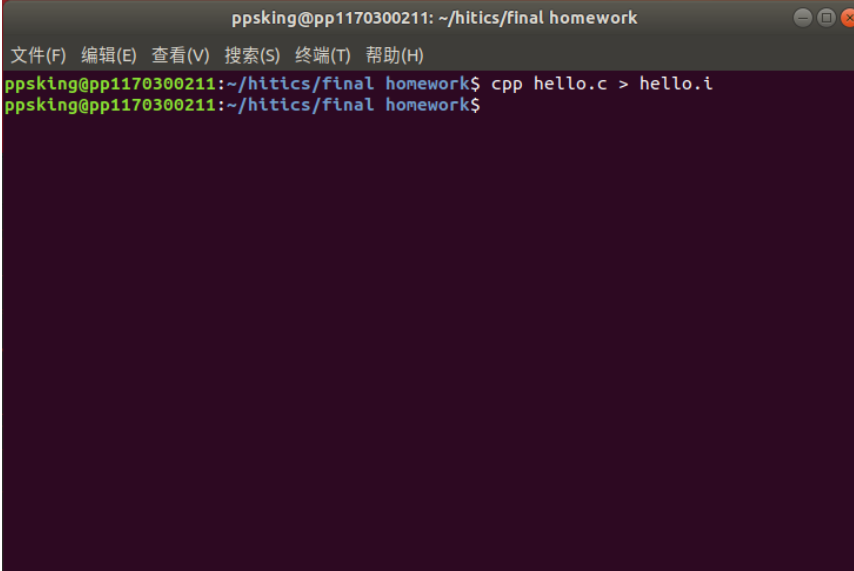
概念：在预处理阶段，预处理器(cpp)根据以#开头的命令，修改原始的 C 程序。

作用：

- a) 将所有的#define 删除，并且展开所有的宏定义。
- b) 处理所有条件编译指令，如#if, #ifdef 等。
- c) 处理#include 预编译指令，将被包含的文件插入到该预编译指令的位置。该过程递归进行，及被包含的文件可能还包含其他文件。
- d) 删除所有的注释//和 /\*\*/。
- e) 添加行号和文件标识，如#2 "hello.c" 2,以便于编译时编译器产生调试用的行号信息及用于编译时产生编译错误或警告时能够显示行号信息。
- f) 保留所有的#pragma 编译器指令，因为编译器须要使用它们。

### 2.2 在 Ubuntu 下预处理的命令

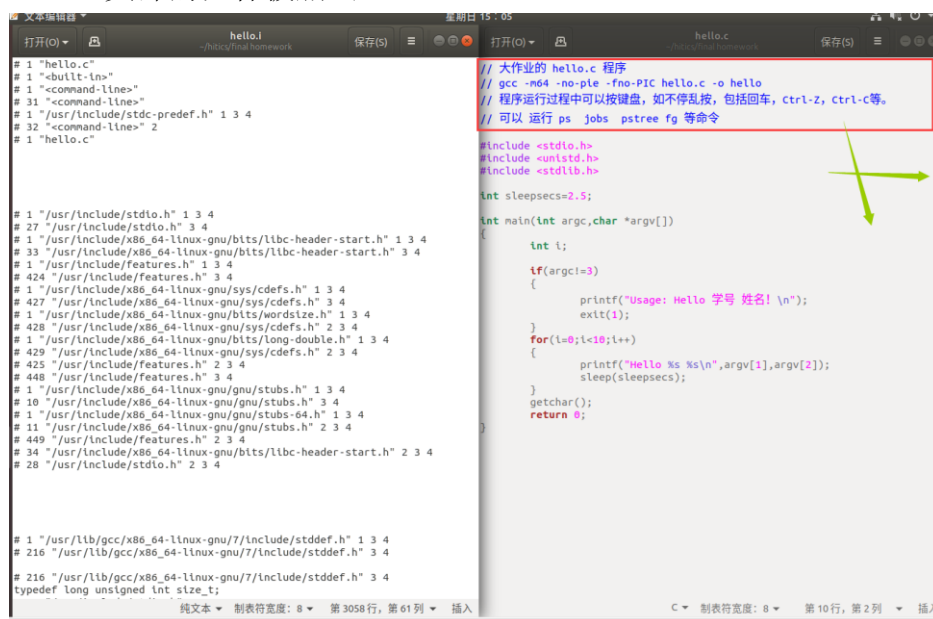
命令：cpp hello.c > hello.i



```
ppsking@pp1170300211: ~/hitics/final homework
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
ppsking@pp1170300211:~/hitics/final homework$ cpp hello.c > hello.i
ppsking@pp1170300211:~/hitics/final homework$
```

### 2.3 Hello 的预处理结果解析

## 1. hello.c 头部的注释被删去:



```
# 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
# 34 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 2 3 4
# 28 "/usr/include/stdio.h" 2 3 4

# 1 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 1 3 4
# 216 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 3 4

# 216 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 3 4
typedef long unsigned int size_t;

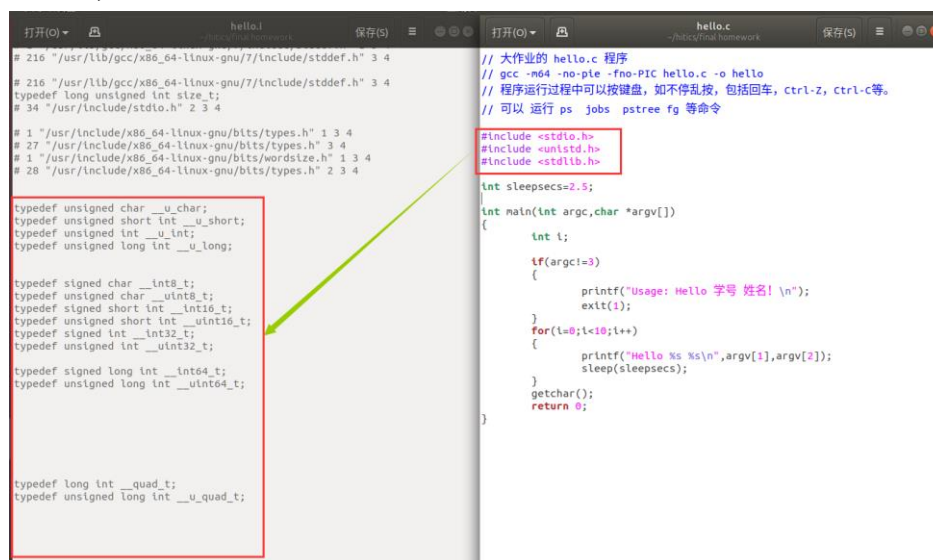
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;

typedef signed char __int8_t;
typedef unsigned char __uint8_t;
typedef signed short int __int16_t;
typedef unsigned short int __uint16_t;
typedef signed int __int32_t;
typedef unsigned int __uint32_t;

typedef signed long int __int64_t;
typedef unsigned long int __uint64_t;

typedef long int __quad_t;
typedef unsigned long int __u_quad_t;
```

## 2. stdio.h, unistd.h 和 stdlib.h 头文件通过#include 指令被插入到相应位置:



```
# 216 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 3 4
# 216 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 3 4
# 34 "/usr/include/stdio.h" 2 3 4

# 1 "/usr/include/x86_64-linux-gnu/bits/types.h" 1 3 4
# 27 "/usr/include/x86_64-linux-gnu/bits/types.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 28 "/usr/include/x86_64-linux-gnu/bits/types.h" 2 3 4

typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;

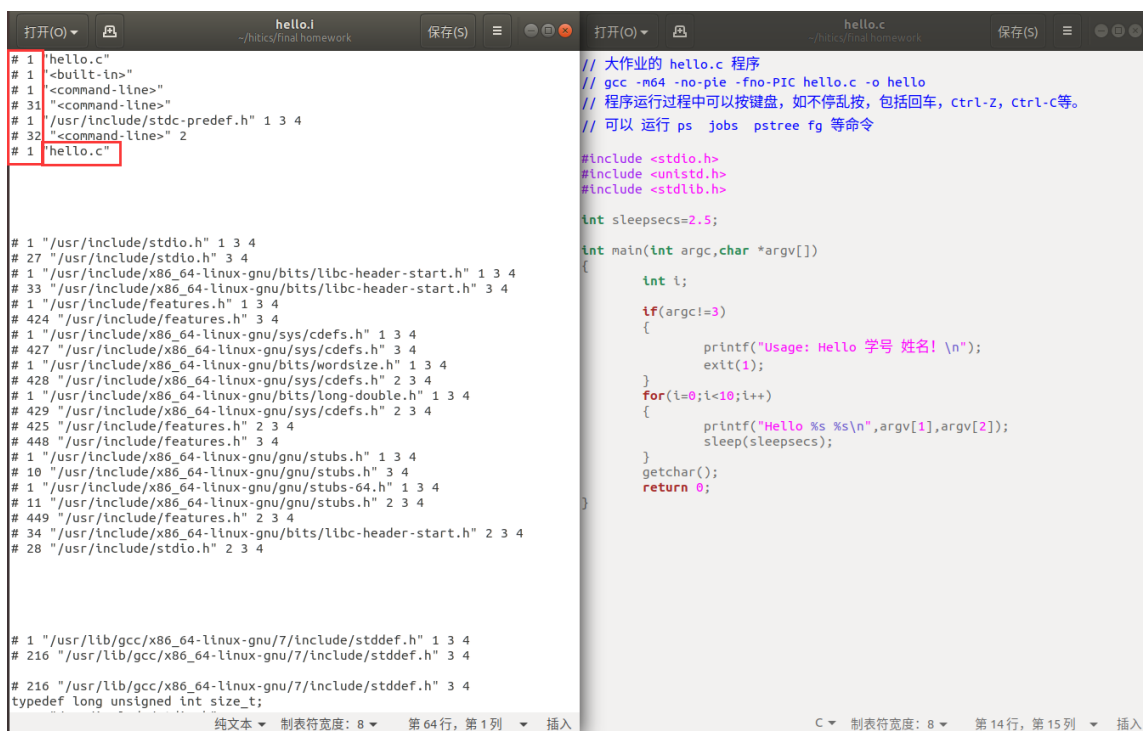
typedef signed char __int8_t;
typedef unsigned char __uint8_t;
typedef signed short int __int16_t;
typedef unsigned short int __uint16_t;
typedef signed int __int32_t;
typedef unsigned int __uint32_t;

typedef signed long int __int64_t;
typedef unsigned long int __uint64_t;

typedef long int __quad_t;
typedef unsigned long int __u_quad_t;
```

## 3. 添加行号和文件标识:





The image shows two side-by-side code editors. The left editor, titled 'hello.i', displays the preprocessed source code for 'hello.c'. It includes system headers like `<stdio.h>`, `<unistd.h>`, and `<stdlib.h>`, and shows the expansion of macros. The right editor, titled 'hello.c', shows the original source code. It includes the same headers and defines `sleepsecs=2.5`. The `main` function takes command-line arguments, checks if the first argument is 'Usage', and then prints 'Hello' followed by the second and third arguments, with a 2.5-second delay. Both editors have a status bar at the bottom showing 'C' and '制表符宽度: 8'.

```
# 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
# 34 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 2 3 4
# 28 "/usr/include/stdio.h" 2 3 4

# 1 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 1 3 4
# 216 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 3 4

# 216 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 3 4
typedef long unsigned int size_t;
```

```
// 大作业的 hello.c 程序
// gcc -m64 -no-pie -fno-PIC hello.c -o hello
// 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。
// 可以运行 ps jobs pstree fg 等命令

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

int sleepsecs=2.5;

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

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

## 2.4 本章小结

预处理器是 c 和 c++ 语言的宏处理器，它实现了头文件的引用，宏替换，条件编译和行控制等功能。

在 C 语言程序运行过程中，cpp 是一个独立的程序，在编译过程中被 gcc 调用，作为从.c 文件到可执行文件的第一个过程。

预处理器产生了.i 文件（预处理文件），为接下来的编译阶段做好准备。

**(第 2 章 0.5 分)**

## 第3章 编译

### 3.1 编译的概念与作用

编译的概念:

编译是通过调用编译器(ccl)将预处理文本文件翻译成汇编程序的过程。

编译的作用:

#### 1. 词法分析:

词法分析的任务是对由字符组成的单词进行处理,从左至右逐个字符地对源程序进行扫描,产生一个个的单词符号,把作为字符串的源程序改造成成为单词符号串的中间程序。执行词法分析的程序称为词法分析程序或扫描器。

#### 2. 语法分析:

编译程序的语法分析器以单词符号作为输入,分析单词符号串是否形成符合语法规则的语法单位,如表达式、赋值、循环等,最后看是否构成一个符合要求的程序,按该语言使用的语法规则分析检查每条语句是否有正确的逻辑结构,程序是最终的一个语法单位。编译程序的语法规则可用上下文无关文法来刻画。

#### 3. 中间代码:

中间代码是源程序的一种内部表示,或称中间语言。中间代码的作用是可使编译程序的结构在逻辑上更为简单明确,特别是可使目标代码的优化比较容易实现中间代码,即为中间语言程序,中间语言的复杂性介于源程序语言和机器语言之间。中间语言有多种形式,常见的有逆波兰记号、四元式、三元式和树。

#### 4. 代码优化:

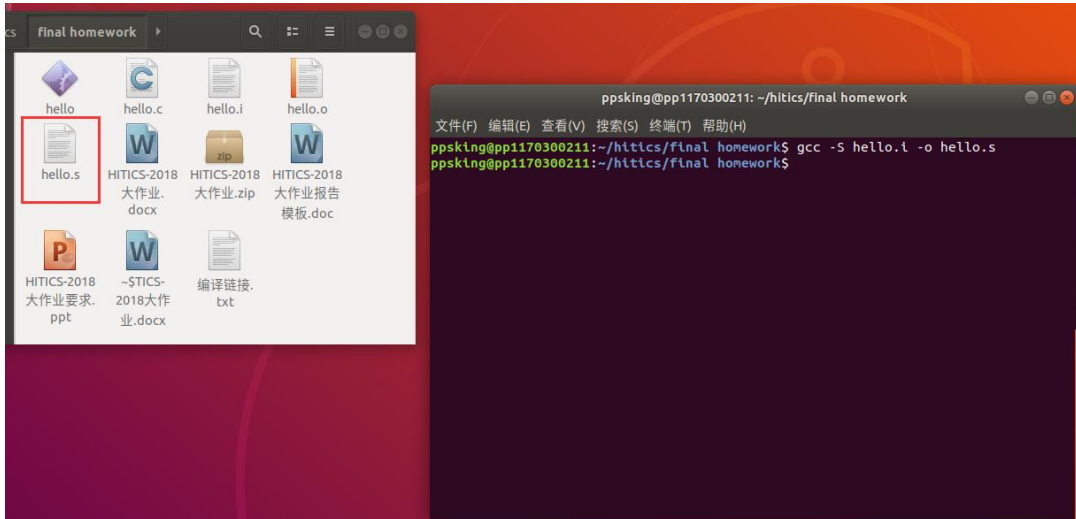
代码优化是指对程序进行多种等价变换,使得从变换后的程序出发,能生成更有效的目标代码。所谓等价,是指不改变程序的运行结果。所谓有效,主要指目标代码运行时间较短,以及占用的存储空间较小。这种变换称为优化。

#### 5. 目标代码:

目标代码生成是编译的最后一个阶段。目标代码生成器把语法分析后或优化后的中间代码变换成目标代码。

## 3.2 在 Ubuntu 下编译的命令

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



## 3.3 Hello 的编译结果解析

### 3.3.1 数据

1, 整形:

- a) `sleepsecs` 作为只读变量被存储在 `.data` 节中, 可以看到 `sleepsecs` 被赋值为 `long` 型的 2.

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

- b) `argc` 是从命令行传入的第一个参数, 保存在寄存器 `%edi` 中, 然后被存入 `-20(%rbp)`

```
27    movl    %edi, -20(%rbp)
28    movq    %rsi, -32(%rbp)
29    cmpl    $3, -20(%rbp)
```

- c) `i` 存储在 `-4(%rbp)` 中, 初始值为 0, 每次循环加一, 退出循环条件是 `i` 大于 9

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

2, 字符串:

a) hello 字符串:

定义在.data 节中, 是需要输出的字符串。

```
.LC0:
.string "Usage: Hello \345\255\246\345\217\267 \345\247\223\345\220\215\357\274\201"
```

b) 格式控制字符串:

```
.LC1:
.string "Hello %s %s\n"
.text
.globl main
.type main, @function
```

3, 数组:

argv[]开始被保存在寄存器%rsi 中, 然后又被保存到栈中 32(%rbp)的位置。

```
subq    $32, %rsp
movl    %edi, -20(%rbp)
movq    %rsi, -32(%rbp)
```

在循环体(.L4)内, 数组的首地址又被赋给了%rax, 然后 main 函数分两次读取了%rax+16 和%rax+8 的地址的内容, 分别放入了%rdx(sprintf 的第三个参数)和%rsi(sprintf 的第二个参数), 然后把.LC1 放入%edi(sprintf 的第一个参数), 然后调用 printf 函数。这样就实现了 printf 的格式化输出。

```
.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)
```

### 3.3.2 赋值

编译器对赋值操作的处理是将其编译为汇编指令 MOV。根据不同大小的数据类型有 movb、movw、movl、movq、movabsq。

hello 程序代码中的  $i = 0$  在汇编代码中体现为“movl \$0x0,-0x4(%rbp)”通过 mov 操作把常数 0 赋值给 i。int 数据类型的大小是 4 个字节。

### 3.3.3 算术操作

hello.s 中实现 i 在循环中递增的代码如下。

```

movl    $0, %eax
call    printf@PLT
movl    sleepsecs(%rip),
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.
.section .note.GNU-st

```

### 3.3.4 关系操作

以 hello 程序里的逻辑关系式 “ $i < 10$ ” 为例，编译器通常通过将其编译为 CMP 指令实现。根据不同的数据大小，有 cmpb、cmpw、cmpl 和 cmpq。在通过 CMP 指令比较后，在通过 jmp 指令跳转。例如 “小于等于则跳转” 的汇编指令是 jle。

```

.L3:
cmpl    $9, -4(%rbp)
jle     .L4
call    getchar@PLT
movl    $0, %eax
leave
.cfi_def_cfa 7, 8
ret
.cfi_endproc

```

C 语言中的关系操作有 ==、!=、<、> 等，这在汇编语言中主要由 cmp 指令和 test 指令实现。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。TEST 指令的行为与 AND 指令一样，除了它们只设置条件码而不改变目的寄存器的值。

### 3.3.5 数组操作

数组的索引实际上就是在第一个元素地址的基础上通过加索引值乘以数据大小来实现。例如整型数组 `a[0]` 的地址是 `address_1`，那么 `a[2]` 即 `address_1 + 4 * 2`。

有了这些分析，就能很好的理解数组在内存中是怎样操作的。以 `hello` 程序中的命令行参数数组的访问涉及到函数参数传递以及命令行参数的相关知识，较为特殊也比较复杂，但仍能看到相关数组操作的思想：

```

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)

```

可以看到，编译器先将 `%rax+16`，然后将对应地址的内容取出放在 `%rdx` 中，这样就实现了 `hello.c` 中的 `argv[2]`，后面的操作同理。

### 3.3.6 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。`hello.c` 中的控制转移操作见图。

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<=）

以 hello 程序中的 if 语句及 for 循环为例进行分析。

```

.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

```

编译器编译的执行逻辑首先和 3 进行比较，如果相等则不满足  $argc != 3$ ，那么通过 je 跳过下面的操作。如果不相等，则执行大括号内部的操作。

### 3.3.7 函数操作

a) 编译器将 `printf("Usage: Hello 学号 姓名! \n");` 编译为:

```

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

```

b) 对于第二个 `printf` 函数共有三个参数，分别是 `.LC1`, `argv[1]` 和 `argv[2]`(图 3.9)，编译器先用 `movq` 指令将对应的参数从内存中取出赋值给 `%rdi`, `%rsi` 和 `%rdx`，然后执行 `call printf` 调用 `printf` 函数。

c) 将 `sleep(sleepsecs);` 编译为:

```

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

```

### 3.4 本章小结

本章详细介绍了编译器处理预处理文件，分析逻辑，生成汇编代码的过程。通过对汇编代码的数据，数组，关系操作，赋值操作，函数操作进行分析，我们对汇编代码的有了全面的认识。编译是从.c 文件到可执行文件的第二步过程，编译为接下来的汇编以及链接打下了基础。

**(第3章2分)**



## 第 4 章 汇编

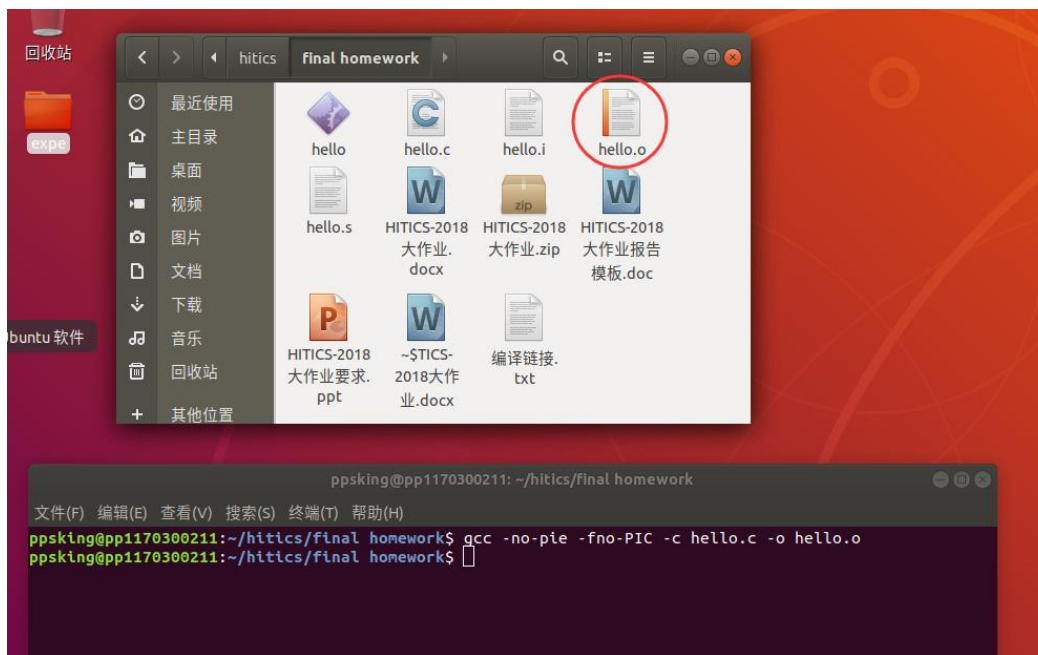
### 4.1 汇编的概念与作用

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

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

### 4.2 在 Ubuntu 下汇编的命令

命令：`as hello.s -o hello.o`



### 4.3 可重定位目标 elf 格式

ELF 文件的通用结构图如下。



接下来分析 hello 的 elf 文件。

### 1. elf 头:

```
ppsking@pp1170300211:~/hitics/final homework$ 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: 1104 (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
```

### 2. 节头:

节头: [号]	名称 大小	类型 全体大小	地址 旗标	链接	偏移量 信息	对齐
[ 0]	0000000000000000	NULL	0000000000000000	0	0	0
[ 1]	.text 000000000000007d	PROGBITS 0000000000000000	0000000000000000 AX	0	0	1
[ 2]	.rela.text 00000000000000c0	RELA 0000000000000018	0000000000000000 I	10	1	8
[ 3]	.data 0000000000000004	PROGBITS 0000000000000000	0000000000000000 WA	0	0	4
[ 4]	.bss 0000000000000000	NOBITS 0000000000000000	0000000000000000 WA	0	0	1
[ 5]	.rodata 000000000000002b	PROGBITS 0000000000000000	0000000000000000 A	0	0	1
[ 6]	.comment 000000000000002b	PROGBITS 0000000000000001	0000000000000000 MS	0	0	1
[ 7]	.note.GNU-stack 0000000000000000	PROGBITS 0000000000000000	0000000000000000	0	0	1
[ 8]	.eh_frame 0000000000000038	PROGBITS 0000000000000000	0000000000000000 A	0	0	8
[ 9]	.rela.eh_frame 0000000000000018	RELA 0000000000000018	0000000000000000 I	10	8	8
[10]	.symtab 00000000000000180	SYMTAB 0000000000000018	0000000000000000	11	9	8
[11]	.strtab 0000000000000037	STRTAB 0000000000000000	0000000000000000	0	0	1
[12]	.shstrtab 0000000000000061	STRTAB 0000000000000000	0000000000000000	0	0	1

Key to Flags:

## 3. 重定位节:

```

重定位节 '.rela.text' at offset 0x310 contains 8 entries:
偏移量 信息 类型 符号值 符号名称 + 加数
000000000016 00050000000a R_X86_64_32 0000000000000000 .rodata + 0
00000000001b 000b00000002 R_X86_64_PC32 0000000000000000 puts - 4
000000000025 000c00000002 R_X86_64_PC32 0000000000000000 exit - 4
00000000004c 00050000000a R_X86_64_32 0000000000000000 .rodata + 1e
000000000056 000d00000002 R_X86_64_PC32 0000000000000000 printf - 4
00000000005c 000900000002 R_X86_64_PC32 0000000000000000 sleepsecs - 4
000000000063 000e00000002 R_X86_64_PC32 0000000000000000 sleep - 4
000000000072 000f00000002 R_X86_64_PC32 0000000000000000 getchar - 4

重定位节 '.rela.eh_frame' at offset 0x3d0 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

The decoding of unwind sections for machine type Advanced Micro Devices X86-64 is not currently supported.

```

## 4. 符号表:

```

Symbol table '.symtab' contains 16 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    125 FUNC  GLOBAL DEFAULT 1 main
  11: 0000000000000000      0 NOTYPE GLOBAL DEFAULT UND puts
  12: 0000000000000000      0 NOTYPE GLOBAL DEFAULT UND exit
  13: 0000000000000000      0 NOTYPE GLOBAL DEFAULT UND printf
  14: 0000000000000000      0 NOTYPE GLOBAL DEFAULT UND sleep
  15: 0000000000000000      0 NOTYPE GLOBAL DEFAULT UND getchar

```

#### 4.4 Hello.o 的结果解析

```

main:                                0000000000000000 <main>:
.LFB5:                                0: 55                push    %rbp
    .cfi_startproc                    1: 48 89 e5            mov     %rsp,%rbp
    pushq %rbp                        4: 48 83 ec 20         sub     $0x20,%rsp
    .cfi_def_cfa_offset 16             8: 89 7d ec            mov     %edi,-0x14(%rbp)
    .cfi_offset 6, -16                b: 48 89 75 e0         mov     %rsi,-0x20(%rbp)
    movq %rsp, %rbp                   f: 83 7d ec 03         cmpl    $0x3,-0x14(%rbp)
    .cfi_def_cfa_register 6           13: 74 14              je      29 <main+0x29>
    subq $32, %rsp                    15: bf 00 00 00 00      mov     $0x0,%edi
    movl %edi, -20(%rbp)               1a: e8 00 00 00 00      callq   1f <main+0x1f>
    movq %rsi, -32(%rbp)              1f: bf 01 00 00 00      mov     $0x1,%edi
    cmpl $3, -20(%rbp)               24: e8 00 00 00 00      callq   29 <main+0x29>
    je .L2                           29: c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%rbp)
    leaq .LC0(%rip), %rdi             30: eb 39              jmp     6b <main+0x6b>
    call puts@PLT                    32: 48 8b 45 e0         mov     -0x20(%rbp),%rax
    movl $1, %edi                    36: 48 83 c0 10         add     $0x10,%rax
    call exit@PLT                   3a: 48 8b 10           mov     (%rax),%rdx
.L2:                                3d: 48 8b 45 e0         mov     -0x20(%rbp),%rax
    movl $0, -4(%rbp)                41: 48 83 c0 08         add     $0x8,%rax
    jmp .L3                          45: 48 8b 00           mov     (%rax),%rax
.L4:                                48: 48 89 c6           mov     %rax,%rsi
    movq -32(%rbp), %rax              4b: bf 00 00 00 00      mov     $0x0,%edi
    addq $16, %rax                    50: b8 00 00 00 00      mov     $0x0,%eax
    movq (%rax), %rdx                 55: e8 00 00 00 00      callq   5a <main+0x5a>
    movq -32(%rbp), %rax              5a: 8b 05 00 00 00 00      mov     0x0(%rip),%eax
    addq $8, %rax                     60: 89 c7              mov     %eax,%edi
    movq (%rax), %rax                 62: e8 00 00 00 00      callq   67 <main+0x67>
    movq %rax, %rsi                   67: 83 45 fc 01         addl    $0x1,-0x4(%rbp)
    leaq .LC1(%rip), %rdi             6b: 83 7d fc 09         cmpl    $0x9,-0x4(%rbp)
    movl $0, %eax                     6f: 7e c1              jle     32 <main+0x32>
    call printf@PLT                   71: e8 00 00 00 00      callq   76 <main+0x76>
    movl sleepsecs(%rip), %eax        76: b8 00 00 00 00      mov     $0x0,%eax
    call sleep@PLT                    7b: c9                leaveq
    addl $1, -4(%rbp)                 7c: c3                retq
.L3:                                7c: c3                retq
    cmpl $9, -4(%rbp)
    jle .L4
    call getchar@PLT
    movl $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret

```

通过对比汇编代码.s 文件和.o 文件反汇编的结果，可以发现几点不同。

1. 编译生成的代码跳转目标通过例如.L2 表示，而反汇编的代码通过一个地址表示。
2. 反汇编的结果中包含了供对照的来自可重定位目标文件中的机器语言代码及相关注释，包括一些相对寻址的信息与重定位信息，尽管没有经过链接不是最终结果，但是能很大程度上反应机器执行的过程。
3. 机器语言由二进制代码构成（图中反汇编结果用 16 进制表示），是计算机能够直接识别和执行的一种机器指令的集合，是与处理器紧密相关的。机器语言由操作码和操作数组成，操作码与汇编语言符号存在的对应关系。由于操作数类型、寻址方式等的不同，同一个汇编语言符号可能对应着不同的机器语言操作码。

## 4.5 本章小结

汇编是 C 语言程序执行过程中从汇编代码到可重定位文件的过程，编译器通过调用汇编器将文本代码翻译为二进制机器代码。通过以上对于 elf 文件的解读，以及对于汇编代码和目标文件的对比，我们可以分析汇编过程中发生的各种变化。汇编过程为接下来的链接打好了基础。

**（第 4 章 1 分）**

## 第 5 章 链接

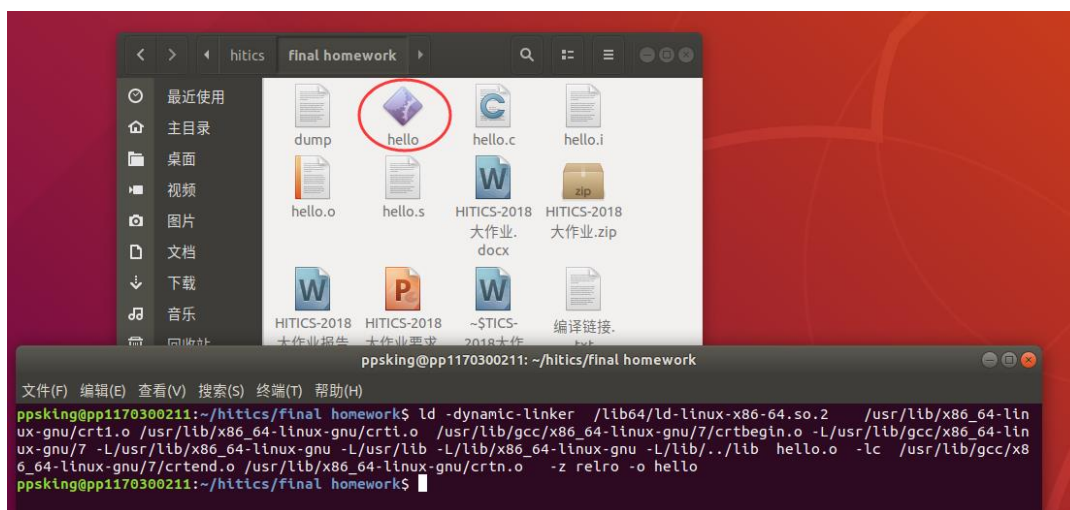
### 5.1 链接的概念与作用

链接的概念：链接是将各种代码和数据片段收集并组合成为一个单一文件的过程，这个文件可被加载到内存并执行。在现代系统中，链接是由叫做链接器的程序自动执行的。

链接的作用：链接使得分离编译成为可能。我们不用将一个大型的应用程序组织为一个巨大的源文件，而是可以把它分解为更小、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，只需简单地重新编译它，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

命令：`ld -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 /usr/lib/gcc/x86_64-linux-gnu/5/crtbegin.o hello.o -lc /usr/lib/gcc/x86_64-linux-gnu/5/crtend.o /usr/lib/x86_64-linux-gnu/crtn.o -z relro -o hello`



### 5.3 可执行目标文件 hello 的格式

使用 `readelf` 工具阅读 `hello.o` 的 `elf` 格式文件。



ppsking@pp1170300211: ~/hitics/final homework

文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

节头:

[号]	名称 大小	类型 全体大小	地址 旗标	链接 信息	偏移量 对齐
[ 0]	0000000000000000	NULL	0000000000000000	0 0	0
[ 1]	.interp 000000000000001c	PROGBITS	0000000000400200	A 0 0	00000200 1
[ 2]	.note.ABI-tag 0000000000000020	NOTE	000000000040021c	A 0 0	0000021c 4
[ 3]	.hash 0000000000000034	HASH	0000000000400240	A 5 0	00000240 8
[ 4]	.gnu.hash 000000000000001c	GNU_HASH	0000000000400278	A 5 0	00000278 8
[ 5]	.dynsym 00000000000000c0	DYNSYM	0000000000400298	A 6 1	00000298 8
[ 6]	.dynstr 0000000000000057	STRTAB	0000000000400358	A 0 0	00000358 1
[ 7]	.gnu.version 0000000000000010	VERSYM	00000000004003b0	A 5 0	000003b0 2
[ 8]	.gnu.version_r 0000000000000020	VERNEED	00000000004003c0	A 6 1	000003c0 8
[ 9]	.rela.dyn 0000000000000030	RELA	00000000004003e0	A 5 0	000003e0 8
[10]	.rela.plt 0000000000000078	RELA	0000000000400410	AI 5 21	00000410 8
[11]	.init 0000000000000017	PROGBITS	0000000000400488	AX 0 0	00000488 4
[12]	.plt 0000000000000060	PROGBITS	00000000004004a0	AX 0 0	000004a0 16
[13]	.text 00000000000001e2	PROGBITS	0000000000400500	AX 0 0	00000500 16
[14]	.fini 0000000000000009	PROGBITS	00000000004006e4	AX 0 0	000006e4 4
[15]	.rodata 000000000000002f	PROGBITS	00000000004006f0	A 0 0	000006f0 4
[16]	.eh_frame 0000000000000100	PROGBITS	0000000000400720	A 0 0	00000720 8
[17]	.init_array 0000000000000008	INIT_ARRAY	0000000000600e00	WA 0 0	00000e00 8
[18]	.fini_array 0000000000000008	FINI_ARRAY	0000000000600e08	WA 0 0	00000e08 8
[19]	.dynamic 00000000000001e0	DYNAMIC	0000000000600e10	WA 6 0	00000e10 8
[20]	.got 0000000000000010	PROGBITS	0000000000600ff0	WA 0 0	00000ff0 8
[21]	.got.plt 0000000000000040	PROGBITS	0000000000601000	WA 0 0	00001000 8
[22]	.data 0000000000000014	PROGBITS	0000000000601040	WA 0 0	00001040 8
[23]	.bss 0000000000000004	NOBITS	0000000000601054	WA 0 0	00001054 1
[24]	.comment 000000000000002a	PROGBITS	0000000000000000	MS 0 0	00001054 1
[25]	.symtab 0000000000000600	SYMTAB	0000000000000000	26 41	00001080 8
[26]	.strtab 000000000000020e	STRTAB	0000000000000000	0 0	00001680 1
[27]	.shstrtab 00000000000000e2	STRTAB	0000000000000000	0 0	0000188e 1

各段信息如图所示，可执行文件的 elf 格式和.o 文件的 elf 格式相像。

## 5.4 hello 的虚拟地址空间

1. 用 edb 加载 hello，打开 Data Dump 查看内容，可以发现开头是 ELF 头。

Data Dump	
+ 0x0000000000400000-0x0000000000401000	
00000000:00400000	7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00 .ELF.....
00000000:00400010	02 00 3e 00 01 00 00 00 05 40 00 00 00 00 00 ..>.....@
00000000:00400020	40 00 00 00 00 00 00 00 70 19 00 00 00 00 00 @.....p....
00000000:00400030	00 00 00 00 40 00 38 00 08 00 40 00 1c 00 1b 00 ...@.8.....@
00000000:00400040	06 00 00 00 04 00 00 00 40 00 00 00 00 00 00 .....@.....@
00000000:00400050	40 00 40 00 00 00 00 00 40 00 40 00 00 00 00 @.....@.....@
00000000:00400060	c0 01 00 00 00 00 00 00 c0 01 00 00 00 00 00 [].....
00000000:00400070	08 00 00 00 00 00 00 00 03 00 00 00 04 00 00 00 .....[].....
00000000:00400080	00 02 00 00 00 00 00 00 00 02 40 00 00 00 00 00 .....@.....
00000000:00400090	00 02 40 00 00 00 00 00 1c 00 00 00 00 00 00 00 .....@.....
00000000:004000a0	1c 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 .....@.....
00000000:004000b0	01 00 00 00 05 00 00 00 00 00 00 00 00 00 00 00 .....@.....@
00000000:004000c0	00 00 40 00 00 00 00 00 00 00 40 00 00 00 00 00 .....@.....@
00000000:004000d0	20 08 00 00 00 00 00 00 20 08 00 00 00 00 00 00 .....@.....@
00000000:004000e0	00 00 20 00 00 00 00 00 00 01 00 00 00 06 00 00 00 .....@.....@
00000000:004000f0	00 0e 00 00 00 00 00 00 00 0e 60 00 00 00 00 00 .....@.....@
00000000:00400100	00 0e 60 00 00 00 00 00 54 02 00 00 00 00 00 00 .....@.....@
00000000:00400110	58 02 00 00 00 00 00 00 00 00 20 00 00 00 00 00 .....@.....@
00000000:00400120	02 00 00 00 06 00 00 00 10 0e 00 00 00 00 00 00 .....@.....@
00000000:00400130	10 0e 60 00 00 00 00 00 10 0e 60 00 00 00 00 00 .....@.....@
00000000:00400140	e0 01 00 00 00 00 00 00 e0 01 00 00 00 00 00 00 .....@.....@
00000000:00400150	08 00 00 00 00 00 00 00 04 00 00 00 04 00 00 00 .....@.....@
00000000:00400160	1c 02 00 00 00 00 00 00 1c 02 40 00 00 00 00 00 .....@.....@
00000000:00400170	1c 02 40 00 00 00 00 00 20 00 00 00 00 00 00 00 .....@.....@
00000000:00400180	20 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 .....@.....@
00000000:00400190	51 e5 74 64 06 00 00 00 00 00 00 00 00 00 00 00 Q{td.....
00000000:004001a0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....@.....@
00000000:004001b0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....@.....@
00000000:004001c0	10 00 00 00 00 00 00 00 52 e5 74 64 04 00 00 00 .....@.....@
00000000:004001d0	00 0e 00 00 00 00 00 00 00 0e 60 00 00 00 00 00 .....@.....@
00000000:004001e0	00 0e 60 00 00 00 00 00 00 02 00 00 00 00 00 00 .....@.....@
00000000:004001f0	00 02 00 00 00 00 00 00 00 01 00 00 00 00 00 00 .....@.....@
00000000:00400200	2f 6c 69 62 36 34 2f 6c 64 2d 6c 69 6e 75 78 2d /lib64/ld-linux- x86-64.so.2.....
00000000:00400210	78 38 36 2d 36 34 2e 73 6f 2e 32 00 04 00 00 00 .....@.....@
00000000:00400220	10 00 00 00 01 00 00 00 47 4e 55 00 00 00 00 00 .....@.....@
00000000:00400230	03 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 .....@.....@
00000000:00400240	03 00 00 00 08 00 00 00 07 00 00 00 06 00 00 00 .....@.....@
00000000:00400250	04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....@.....@
00000000:00400260	01 00 00 00 02 00 00 00 00 00 00 00 03 00 00 00 .....@.....@
00000000:00400270	05 00 00 00 00 00 00 00 01 00 00 00 01 00 00 00 .....@.....@
00000000:00400280	01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....@.....@
00000000:00400290	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....@.....@
00000000:004002a0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....@.....@
00000000:004002b0	10 00 00 00 12 00 00 00 00 00 00 00 00 00 00 00 .....@.....@
00000000:004002c0	00 00 00 00 00 00 00 00 15 00 00 00 12 00 00 00 .....@.....@
00000000:004002d0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....@.....@
00000000:004002e0	2a 00 00 00 12 00 00 00 00 00 00 00 00 00 00 00 .....@.....@
00000000:004002f0	00 00 00 00 00 00 00 00 1c 00 00 00 12 00 00 00 .....@.....@
00000000:00400300	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....@.....@
00000000:00400310	48 00 00 00 20 00 00 00 00 00 00 00 00 00 00 00 .....@.....@
00000000:00400320	00 00 00 00 00 00 00 00 0b 00 00 00 12 00 00 00 .....@.....@
00000000:00400330	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....@.....@

2. 在 0x400200 位置是节头目表中的.interp 节，存储了 linux 动态共享库的路径。



```

00000000:00400200 2f 6c 69 62 36 34 2f 6c 64 2d 6c 69 6e 75 78 2d /lib64/ld-linux-
00000000:00400210 78 36 36 20 36 34 2e 73 61 2e 32 00 04 00 00 00 x86-64.so.2....
00000000:00400220 10 00 00 00 01 00 00 00 47 4e 55 00 00 00 00 00 .....GNU.....
00000000:00400230 03 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 .....
00000000:00400240 03 00 00 00 08 00 00 00 07 00 00 00 06 00 00 00 .....
00000000:00400250 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000000:00400260 01 00 00 00 02 00 00 00 00 00 00 00 03 00 00 00 .....
00000000:00400270 05 00 00 00 00 00 00 00 01 00 00 00 01 00 00 00 .....
00000000:00400280 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000000:00400290 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000000:004002a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000000:004002b0 10 00 00 00 12 00 00 00 00 00 00 00 00 00 00 00 .....
00000000:004002c0 00 00 00 00 00 00 00 00 15 00 00 00 12 00 00 00 .....
00000000:004002d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000000:004002e0 2a 00 00 00 12 00 00 00 00 00 00 00 00 00 00 00 *.
00000000:004002f0 00 00 00 00 00 00 00 00 1c 00 00 00 12 00 00 00 .....

```

3. 在 0x400358 位置是节头目标中的 dynstr 节，保存动态字符串。

```

00000000:00400358 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000000:00400360 2e 36 00 65 78 69 74 00 70 75 74 73 00 70 72 69 .....libc.so
00000000:00400370 6e 74 66 00 67 65 74 63 68 61 72 00 73 6c 65 65 .....6.exit.puts.pri
00000000:00400380 70 00 5f 5f 6c 69 62 63 5f 73 74 61 72 74 5f 6d .....ntf.getchar.slee
00000000:00400390 61 69 6e 00 47 4c 49 42 43 5f 32 2e 32 2e 35 00 .....p.__libc_start_m
00000000:004003a0 5f 5f 67 6d 6f 6e 5f 73 74 61 72 74 5f 5f 00 00 .....ain.GLIBC 2.2.5.
00000000:004003b0 00 00 02 00 02 00 02 00 02 00 00 00 02 00 02 00 .....__gmon_start__
00000000:004003c0 01 00 01 00 01 00 00 00 10 00 00 00 00 00 00 00 .....
00000000:004003d0 75 1a 69 09 00 00 02 00 3c 00 00 00 00 00 00 00 .....u.i...<.....
00000000:004003e0 f0 0f 60 00 00 00 00 00 06 00 00 00 03 00 00 00 .....
00000000:004003f0 00 00 00 00 00 00 00 00 00 f8 0f 60 00 00 00 00 .....
00000000:00400400 06 00 00 00 05 00 00 00 00 00 00 00 00 00 00 00 .....
00000000:00400410 18 10 60 00 00 00 00 00 07 00 00 00 01 00 00 00 .....
00000000:00400420 00 00 00 00 00 00 00 00 20 10 60 00 00 00 00 00 .....

```

4. 在 0x4006f0 位置是节头目标表中的 rodata 节，保存只读数据，比如打印的“hello”字符串和格式控制字符串“hello %s %s”。

```

00000000:004006e0 f3 c3 00 00 48 83 ec 08 48 83 c4 08 c3 00 00 00 .....H.L.H.L.H...
00000000:004006f0 01 00 02 00 55 73 61 67 65 3a 20 48 65 6c 6c 6f .....Usage: Hello
00000000:00400700 20 e5 ad a6 e5 8f b7 20 e5 a7 93 e5 90 8d ef bc .....àààà.ò àc.à.[]E
00000000:00400710 81 00 48 65 6c 6c 6f 20 25 73 20 25 73 0a 00 00 .....Hello %s %s
00000000:00400720 14 00 00 00 00 00 00 00 01 7a 52 00 01 78 10 01 .....zR..x...
00000000:00400730 1b 0c 07 08 90 01 07 10 10 00 00 00 1c 00 00 00 .....
00000000:00400740 c0 fd ff ff 2b 00 00 00 00 00 00 00 14 00 00 00 .....+
00000000:00400750 00 00 00 00 01 7a 52 00 01 78 10 01 1b 0c 07 08 .....zR..x...
00000000:00400760 90 01 00 00 10 00 00 00 1c 00 00 00 c4 fd ff ff .....+
00000000:00400770 02 00 00 00 00 00 00 00 24 00 00 00 30 00 00 00 .....$.0.
00000000:00400780 20 fd ff ff 60 00 00 00 0e 10 46 0e 18 4a 0f .....F.J.
00000000:00400790 0b 77 08 80 00 3f 1a 3b 2a 33 24 22 00 00 00 00 .....w...?;*3$"
00000000:004007a0 1c 00 00 00 58 00 00 00 3f fe ff ff 7d 00 00 00 .....X...?[]}...
00000000:004007b0 00 41 0e 10 86 02 43 0d 06 02 78 0c 07 08 00 00 .....A...C...x...
00000000:004007c0 44 00 00 00 78 00 00 00 a8 fe ff ff 65 00 00 00 .....D...x...[]e...
00000000:004007d0 00 42 0e 10 8f 02 42 0e 18 8e 03 45 0e 20 8d 04 .....B...B...E...
00000000:004007e0 42 0e 28 8c 05 48 0e 30 86 06 48 0e 38 83 07 4d .....B...H.0...H.8.M
00000000:004007f0 0e 40 72 0e 38 41 0e 30 41 0e 28 42 0e 20 42 0e .....@r.8A.0A.(B.B.

```

## 5.5 链接的重定位过程分析

使用 objdump 获得 hello 的反汇编代码。

```

ppsking@pp1170300211:~/hitcs/final homework$ objdump -d hello

hello:      文件格式 elf64-x86-64

Disassembly of section .init:

0000000000400488 <_init>:
 400488:    48 83 ec 08      sub    $0x8,%rsp
 40048c:    48 8b 05 65 0b 20 00 mov    0x200b65(%rip),%rax      # 600f
f8 <__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
Ubuntu 软件

Disassembly of section .plt:

00000000004004a0 <.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)

00000000004004b0 <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>

00000000004004c0 <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>

00000000004004d0 <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>

```

```

00000000004005e7 <main>:
4005e7:    55                push    %rbp
4005e8:    48 89 e5          mov     %rsp,%rbp
4005eb:    48 83 ec 20       sub     $0x20,%rsp
4005ef:    89 7d ec          mov     %edi,-0x14(%rbp)
4005f2:    48 89 75 e0       mov     %rsi,-0x20(%rbp)
4005f6:    83 7d ec 03       cmpl    $0x3,-0x14(%rbp)
4005fa:    74 14             je      400610 <main+0x29>
4005fc:    bf f4 06 40 00    mov     $0x4006f4,%edi
400601:    e8 aa fe ff ff    callq   4004b0 <puts@plt>
400606:    bf 01 00 00 00    mov     $0x1,%edi
40060b:    e8 d0 fe ff ff    callq   4004e0 <exit@plt>
400610:    c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%rbp)
400617:    eb 39            jmp     400652 <main+0x6b>
400619:    48 8b 45 e0       mov     -0x20(%rbp),%rax
40061d:    48 83 c0 10       add     $0x10,%rax
400621:    48 8b 10          mov     (%rax),%rdx
400624:    48 8b 45 e0       mov     -0x20(%rbp),%rax
400628:    48 83 c0 08       add     $0x8,%rax
40062c:    48 8b 00          mov     (%rax),%rax
40062f:    48 89 c6          mov     %rax,%rsi
400632:    bf 12 07 40 00    mov     $0x400712,%edi
400637:    b8 00 00 00 00    mov     $0x0,%eax
40063c:    e8 7f fe ff ff    callq   4004c0 <printf@plt>
400641:    8b 05 09 0a 20 00 mov     0x200a09(%rip),%eax    # 6010
50 <sleepsecs>
400647:    89 c7            mov     %eax,%edi
400649:    e8 a2 fe ff ff    callq   4004f0 <sleep@plt>
40064e:    83 45 fc 01       addl    $0x1,-0x4(%rbp)
400652:    83 7d fc 09       cmpl    $0x9,-0x4(%rbp)
400656:    7e c1            jle     400619 <main+0x32>
400658:    e8 73 fe ff ff    callq   4004d0 <getchar@plt>
40065d:    b8 00 00 00 00    mov     $0x0,%eax
400662:    c9              leaveq  %eax
400663:    c3              retq
400664:    66 2e 0f 1f 84 00 00 nopw    %cs:0x0(%rax,%rax,1)
40066b:    00 00 00
40066e:    66 90            xchg    %ax,%ax
0000000000400670 <__libc_csu_init>:
400670:    41 57            push    %r15
400672:    41 56            push    %r14
400674:    49 89 d7          mov     %rdx,%r15

```

链接器在完成符号解析以后，就把代码中的每个符号引用和正好一个符号定义（即它的一个输入目标模块中的一个符号表条目）关联起来。此时，链接器就知道它的输入目标模块中的代码节和数据节的确切大小。然后就可以开始重定位步骤了，在这个步骤中，将合并输入模块，并为每个符号分配运行时的地址。在 `hello` 到 `hello.o` 中，首先是重定位节和符号定义，链接器将所有输入到 `hello` 中相同类型的节合并为同一类型的新的聚合节。例如，来自所有的输入模块的 `.data` 节被全部合并成一个节，这个节成为 `hello` 的 `.data` 节。

对比 `hello` 和 `hello.o`，发现：

1. `hello.o` 的地址空间从 0 开始，而 `hello` 从 0x400000 开始。
2. `hello.o` 中不含有 `printf()`，`getchar()` 等函数的代码。

3. hello.o 含有重定位标记，用于链接生成 hello。
4. hello 中的地址是绝对地址，可以加载到内存中运行。
5. hello 中的相对寻址经过了重定位。

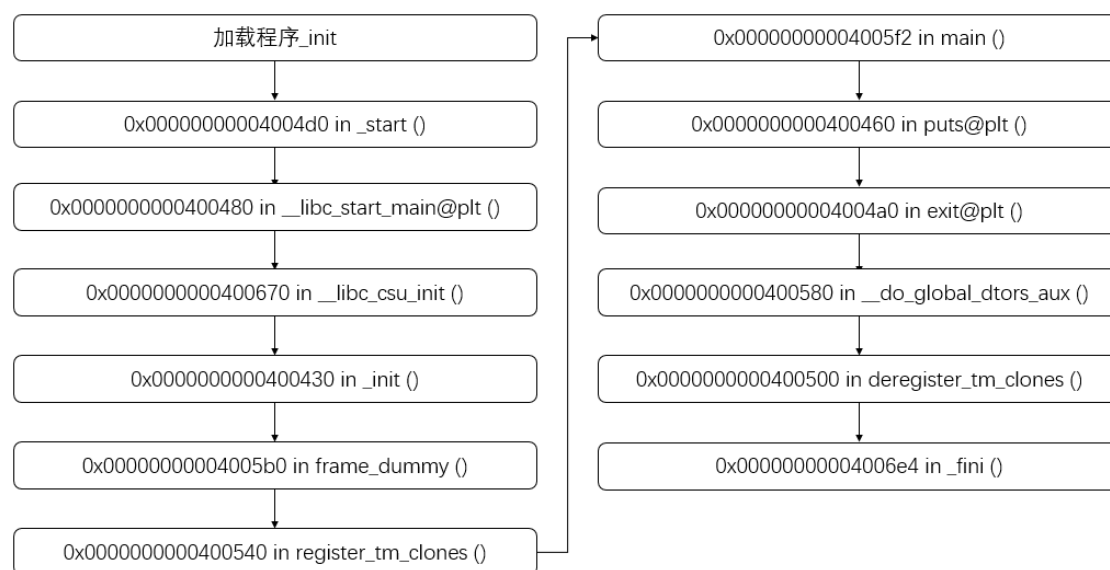
## 5.6 hello 的执行流程

通过 edb 逐步运行调试 hello 程序：

Address	Hex Data	Assembly Code
00000000:00400502	49 89 d1	xorl %ebp, %ebp
00000000:00400505	5e	movq %rdx, %r9
00000000:00400506	48 89 e2	popq %rsi
00000000:00400509	48 83 e4 f0	movq %rsp, %rdx
00000000:0040050d	50	andq \$0xffffffff0, %rsp
00000000:0040050e	54	pushq %rax
00000000:0040050f	49 c7 c0 e0 06 40 00	pushq %rsp
00000000:00400516	48 c7 c1 70 06 40 00	movq \$0x4006e0, %r8
00000000:0040051d	48 c7 c7 e7 05 40 00	movq \$0x400670, %rcx
00000000:00400524	ff 15 c6 0a 20 00	movq \$0x4005e7, %rdi
00000000:0040052a	f4	callq *0x200ac6(%rip)
00000000:0040052b	0f 1f 44 00 00	hlt
00000000:00400530	f3 c3	nopl (%rax, %rax)
00000000:00400532	66 2e 0f 1f 84 00 00 0...	retq
00000000:0040053c	0f 1f 40 00	nopl (%cs:(%rax, %rax))
00000000:00400540	55	nopl (%rax)
00000000:00400541	b8 58 10 60 00	pushq %rbp
00000000:00400546	48 3d 58 10 60 00	movl \$0x601058, %eax
00000000:0040054c	48 89 e5	cmpq \$0x601058, %rax
00000000:0040054f	74 17	movq %rsp, %rbp
00000000:00400551	b8 00 00 00 00	je 0x400568
00000000:00400556	48 85 c0	movl \$0, %eax
00000000:00400559	74 0d	testq %rax, %rax
00000000:0040055b	5d	je 0x400568
00000000:0040055c	bf 58 10 60 00	popq %rbp
00000000:00400561	ff e0	movl \$0x601058, %edi
00000000:00400563	0f 1f 44 00 00	jmpq *%rax
00000000:00400568	5d	nopl (%rax, %rax)
00000000:00400569	c3	popq %rbp
00000000:0040056a	66 0f 1f 44 00 00	retq
00000000:00400570	be 58 10 60 00	nopl (%rax, %rax)
00000000:00400575	ff	movl \$0x601058, %esi

ebp = 00000000

得到 hello 的执行流程如下图：

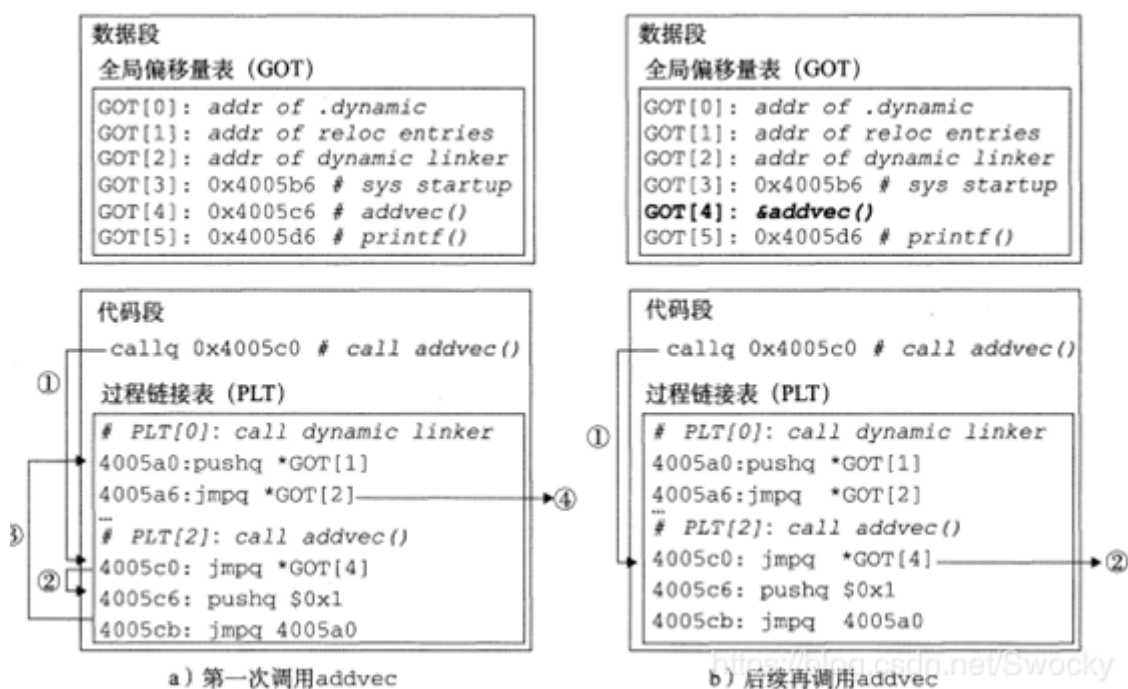


## 5.7 Hello 的动态链接分析

对于动态共享链接库中 PIC 函数，编译器没有办法预测函数的运行时地址，所以需要添加重定位记录，等待动态链接器处理，GNU 编译系统使用延迟绑定技术，将过程地址的绑定推迟到第一次调用该过程时。

使用延迟绑定的动机是对于一个像 `libc.so` 这样的共享库输出的成百上千个函数中，一个典型的应用程序只会使用其中很少的一部分。把函数地址的解析推迟到它实际被调用的地方，能避免动态链接器在加载时进行成百上千个其实并不需要的重定位。第一次调用过程的运行时开销很大，但是其后的每次调用都只会花费一条指令和一个间接的内存引用。

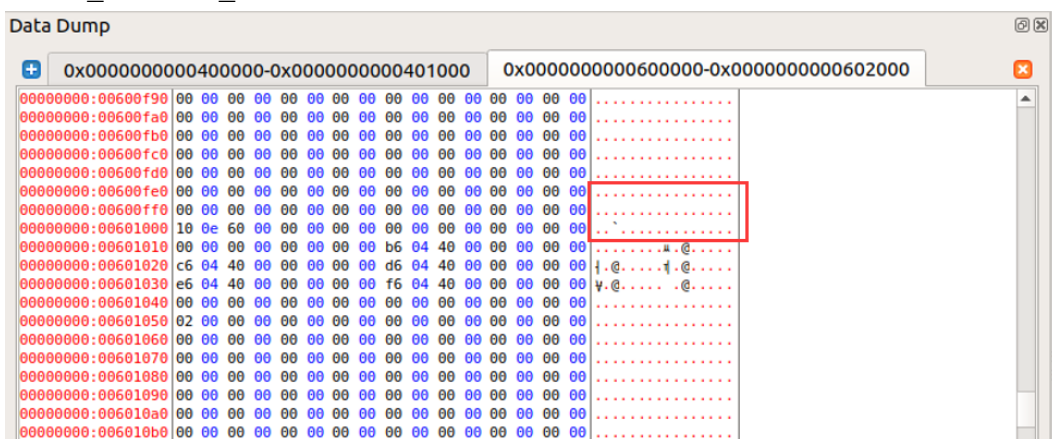
延迟绑定是通过两个数据结构的交互来实现的，这两个数据结构是 GOT(全局偏移量表)和 PLT(过程链接表)。如果一个目标模块调用定义在共享库中的任何函数，那么它就有自己的 GOT 和 PLT。GOT 是数据段的一部分，PLT 是代码段的一部分。下图介绍了动态链接库函数的延迟绑定。



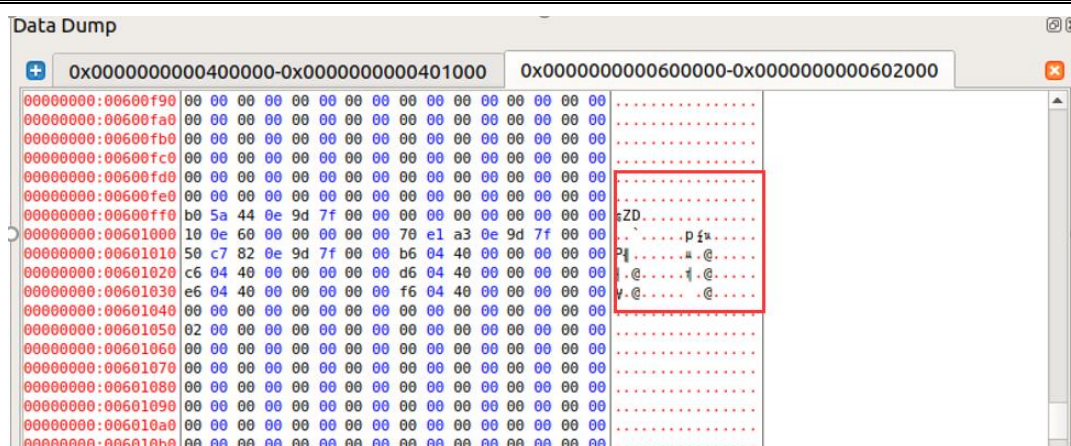
通过 edb 调试，在动态链接的过程中，`_GLOBAL_OFFSET_TABLE_` 发生了如下的变化。

调用 `dl_init` 前，动态库函数指向的地址。从图 5.3.3 中能够读取到 GOT 表的起始位置，即 `0x601000`。在 `dl_init` 调用之前可以查看其值，发现均为 0。调用 `dl_init` 后再次查看，根据 GOT 表的每一项为 8 字节，可以推得 GOT[2] 也就是存放动态链接器入口的地址为 `0x601010`。经过 `dl_init` 的调用，这里已经有了一段地址，为 `0x7f906bbf2870`。

查看 `hello` 的 `elf` 文件，`.got` 条目的地址是 `0xff0`，在 `dl_init` 前后，`_GLOBAL_OFFSET_TABLE_` 发生了变化。







## 5.8 本章小结

链接作为生成 C 语言可执行程序的最后一步，具有如下特点：模块化，代码可以写成好几个而不是写在一个源文件里面。；高性能：可以分离编译，改变一个文件，只需要对改变的文件进行重新编译。连接器在连接的过程中进行符号的确认：在源码中有符号定义，比如函数定义；也有函数索引。链接器就是把每个变量索引与唯一的一个变量定义进行关联，完成重定位的工作，将不同的部分链接成一个部分。把每个变量和函数的位置确定。

(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。系统中每个程序都运行在某个进程的上下文(context)中。

进程的作用：进程提供给应用程序两个关键的抽象：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。

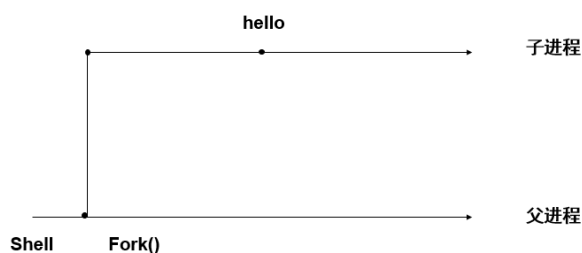
### 6.2 简述壳 Shell-bash 的作用与处理流程

作用：Shell 是一个提供访问操作系统服务的用户界面，它通过命令行接受用户的命令，然后执行相应的命令或者调用相应的应用程序。

处理流程：首先，Shell 解析用户输入的命令，翻译为相应的程序名+参数版本，然后判断该命令是否是内置命令，如果是则执行内置命令，如果不是则根据输入的程序名路径执行相应的程序。

### 6.3 Hello 的 fork 进程创建过程

Shell 通过调用 fork 函数创建一个子进程，提供 hello 执行。通过 fork 函数，子进程得到与父进程用户级虚拟地址空间相同的（但是独立的）一份副本。



### 6.4 Hello 的 execve 过程

Shell 通过 fork 产生子进程后，子进程调用 execve 函数。execve 在子进程上下文中加载并运行 hello 程序。execve 首先调用加载器，加载器删除原有的内存段，并将 hello 的代码的片复制到代码段和数据段。接下来，加载器跳转到程序的入口点，也就是\_start 函数的地址。\_start 函数调用系统启动函数\_\_libc\_start\_main，它初始化环境，调用 main 函数。这样就实现了 hello 的 execve 过程。



## 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。

hello 作为一个进程拥有自己的上下文，当没有异常中断的条件下，hello 正常运行。当有异常或系统中断时时，控制将返回给 linux 内核，在内核模式中完成上下文切换。当运行到 sleep()时，hello 调用系统函数，挂起进程，并在 2s 后，通过计时器产生一个终端，系统重新将控制返回给 hello。

## 6.6 hello 的异常与信号处理

### 1. 异常的种类

- a) 中断：中断是来自 I/O 设备的信号，异步发生，中断处理程序对其进行处理，返回后继续执行调用前待执行的下一条代码，就像没有发生过中断。
- b) 陷阱：陷阱是有意的异常，是执行一条指令的结果，调用后也会返回到下一条指令，用来调用内核的服务进行操作。帮助程序从用户模式切换到内核模式。
- c) 故障：故障是由错误情况引起的，它可能能够被故障处理程序修正。如果修正成功，则将控制返回到引起故障的指令，否则将终止程序。
- d) 终止：终止是不可恢复的致命错误造成的结果，通常是一些硬件的错误，处理程序会将控制返回给一个 abort 例程，该例程会终止这个应用程序。

### 2. 命令运行

#### a) ps:

```
ppsking@pp1170300211:~/hitics/final homework$ ps
  PID TTY          TIME CMD
 4622 pts/0        00:00:00 bash
 4734 pts/0        00:00:00 hello
 4737 pts/0        00:00:00 ps
```

#### b) jobs:

```
ppsking@pp1170300211:~/hitics/final homework$ jobs
[1]+  已停止                  ./hello 1170300211 彭湃
```

c) pstree:

```
ppsking@pp1170300211:~/hitics/final homework$ pstree
systemd├──ModemManager──2*[{ModemManager}]
      │├──NetworkManager──dhclient
      ││└──2*[{NetworkManager}]
      ├──VGAAuthService
      ├──accounts-daemon──2*[{accounts-daemon}]
      ├──acpid
      ├──avahi-daemon──avahi-daemon
      ├──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+
```

d) fg:发送 SIGCONT 信号继续执行停止的进程。

```
ppsking@pp1170300211:~/hitics/final homework$ fg
./hello 1170300211 彭湃
Hello 1170300211 彭湃
Hello 1170300211 彭湃
Hello 1170300211 彭湃
```

e) kill:向 PID 指定的进程发送特定信号。

```
ppsking@pp1170300211:~/hitics/final homework$ ps
  PID TTY          TIME CMD
  4622 pts/0        00:00:00 bash
  4734 pts/0        00:00:00 hello
  4772 pts/0        00:00:00 ps
ppsking@pp1170300211:~/hitics/final homework$ kill -9 4734
ppsking@pp1170300211:~/hitics/final homework$ ps
  PID TTY          TIME CMD
  4622 pts/0        00:00:00 bash
  4773 pts/0        00:00:00 ps
[1]+  已杀死                  ./hello 1170300211 彭湃
```

f) Ctrl-Z:向进程发送 SIGSTP 挂起进程。

```
ppsking@pp1170300211:~/hitics/final homework$ ./hello 1170300211 彭湃
Hello 1170300211 彭湃
Hello 1170300211 彭湃
Hello 1170300211 彭湃
^Z
[1]+  已停止                  ./hello 1170300211 彭湃
```

g) Ctrl-C:向进程发送 SIGINT 终止进程。

```
ppsking@pp1170300211:~/hitics/final homework$ ./hello 1170300211 彭湃
Hello 1170300211 彭湃
Hello 1170300211 彭湃
Hello 1170300211 彭湃
Hello 1170300211 彭湃
Hello 1170300211 彭湃
Hello 1170300211 彭湃
Hello 1170300211 彭湃
Hello 1170300211 彭湃
^C
ppsking@pp1170300211:~/hitics/final homework$ ps
  PID TTY          TIME CMD
  4622 pts/0        00:00:00 bash
  4782 pts/0        00:00:00 ps
```

## 6.7 本章小结

本章详细介绍了 `hello` 的运行和终止的过程，围绕着进程管理的思想，展示了上下文切换，进程挂起、终止等 linux 进程管理的细节。

**(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

1. 线性地址：地址空间是一个非负整数地址的有序集合，而如果此时地址空间中 的整数是连续的，则我们称这个地址空间为线性地址空间。
2. 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，其每一个字节都被给予一个唯一的地址，这个地址称为物理地址。物理 地址也是计算机的硬件中的电路进行操作的地址。
3. 逻辑地址：由程序产生的与段有关的偏移地址。分为两个部分，一个部分为段 基址，另一个部分为段偏移量。
4. 虚拟地址：与物理地址相似，虚拟内存被组织为一个存放在磁盘上的  $N$  个连续的字节大小的单元组成的数组，其每个字节对应的地址成为虚拟地址。

### 7.2 Intel 逻辑地址到线性地址的变换-段式管理

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 页的大小位  $P = 2^p$  字节。类似地，物理内存被分割为物理页，大小也为  $P$  字节。

数据段描述符

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32				
BASE(24-31)								G	B	O	A	V	L	LIMIT (16-19)				1	D	P	S	=	1	TYPE				BASE (16-23)							
BASE(0-15)																LIMIT (0-15)																			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				

代码段描述符

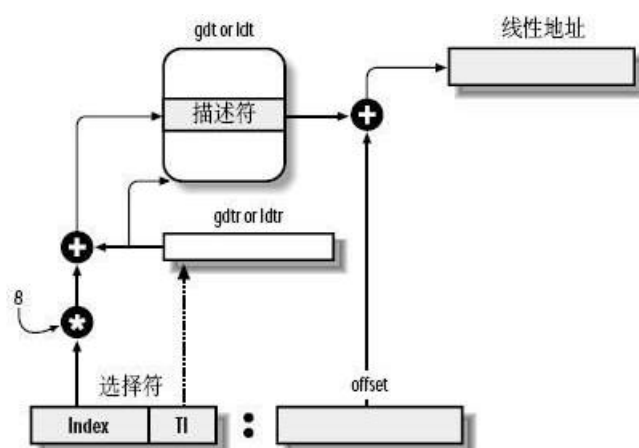
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
BASE(24-31)								G	D	O	A	LIMIT (16-19)				1	D	P	S	TYPE				BASE (16-23)							
BASE(0-15)																LIMIT (0-15)															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

系统段描述符

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32		
BASE(24-31)								G		O		LIMIT (16-19)				1	D	P	S	=	0	TYPE				BASE (16-23)							
BASE(0-15)																LIMIT (0-15)																	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		

其中 Base 字段，它描述了一个段的开始位置的线性地址，一些全局的段描述

符，就放在“全局段描述符表(GDT)”中，一些局部的，例如每个进程自己的，就放在所谓的“局部段描述符表(LDT)”中，由段选择符中的 T1 字段表示选择使用哪个，=0，表示用 GDT，=1 表示用 LDT。GDT 在内存中的地址和大小存放在 CPU 的 gdttr 控制寄存器中，而 LDT 则在 ldtr 寄存器中。如下图：

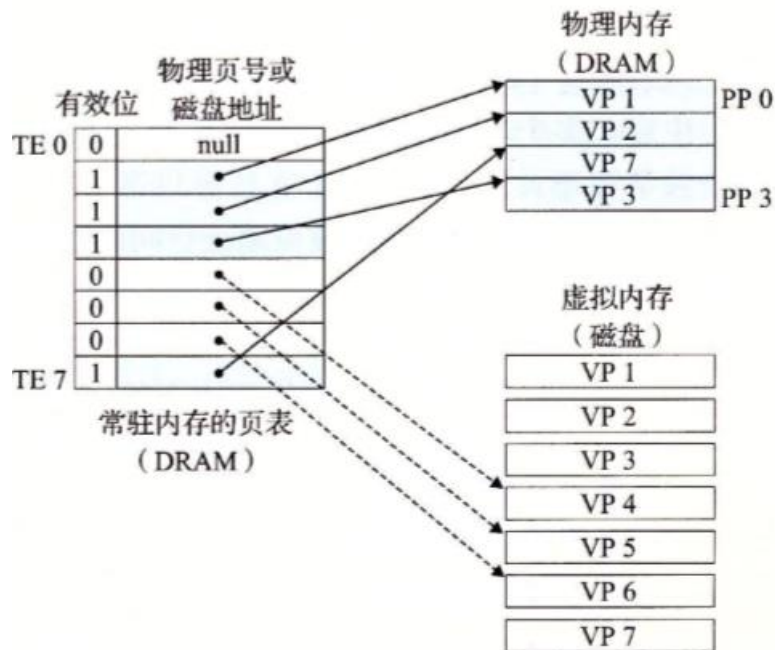


转换步骤：

1. 给定一个完整的逻辑地址[段选择符：段内偏移地址]。
2. 看段选择符的 T1=0 还是 1，知道当前要转换是 GDT 中的段，还是 LDT 中的段，再根据相应寄存器，得到其地址和大小。可以得到一个数组。
3. 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 Base，也就是基地址。
4. 线性地址 = Base + offset。

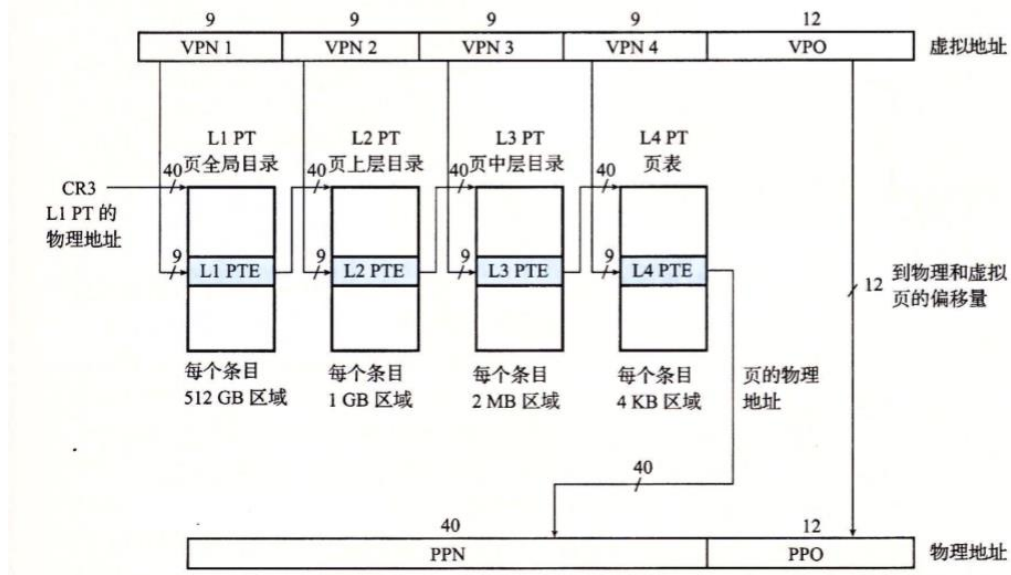
### 7.3 Hello 的线性地址到物理地址的变换-页式管理

如下图所示，页表中的各个页表条目（PTE）存储了页式管理的信息。有效位表明了该虚拟也当前是否被缓存在 DRAM 中，地址段记录了该地址对应的物理内存或者磁盘地址。



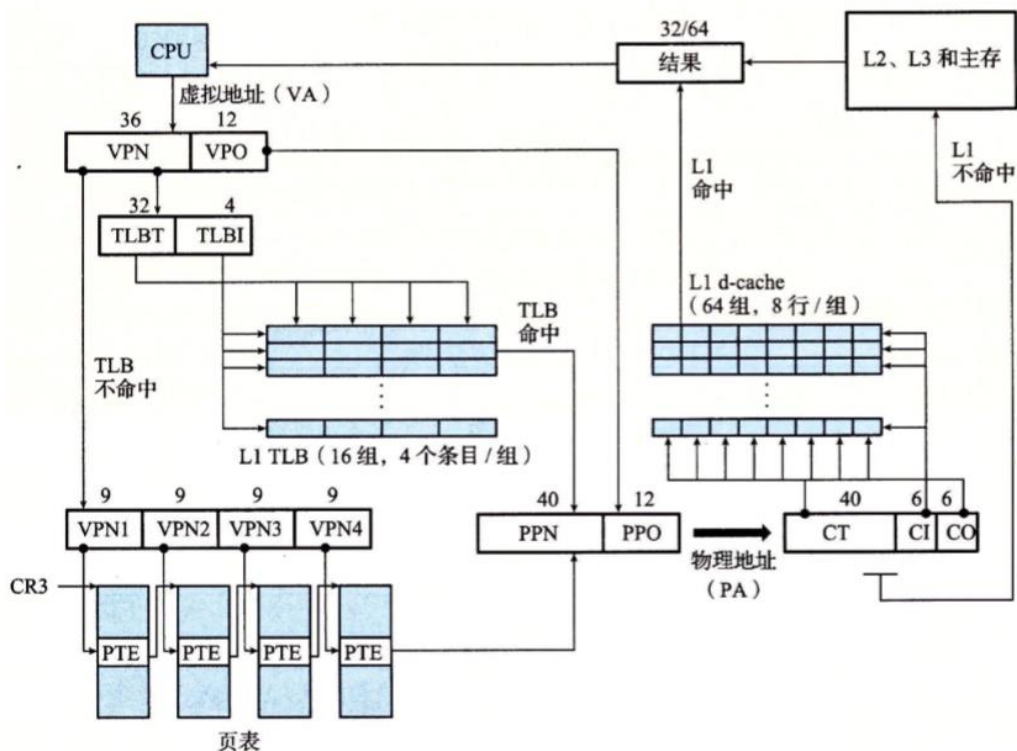
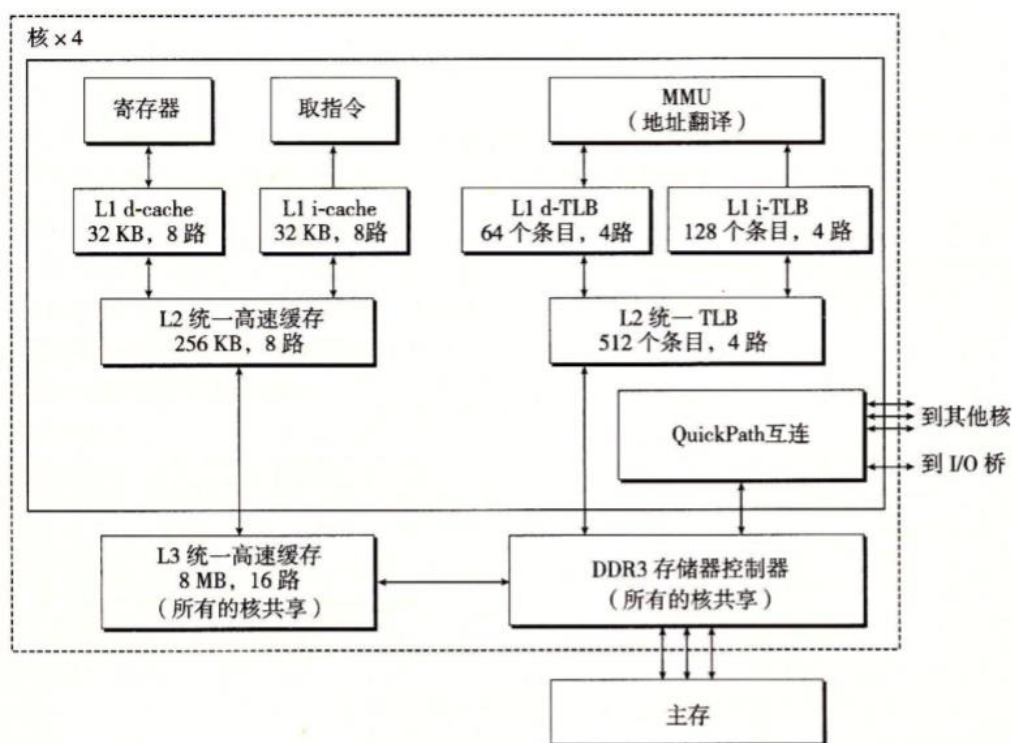
## 7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

如下图所示，MMU 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 提供一个 L1 PTE 的偏移量，这个 PTE 包含 L2 页表的基地址。VPN 2 提供一个 L2 PTE 的偏移量，以此类推。

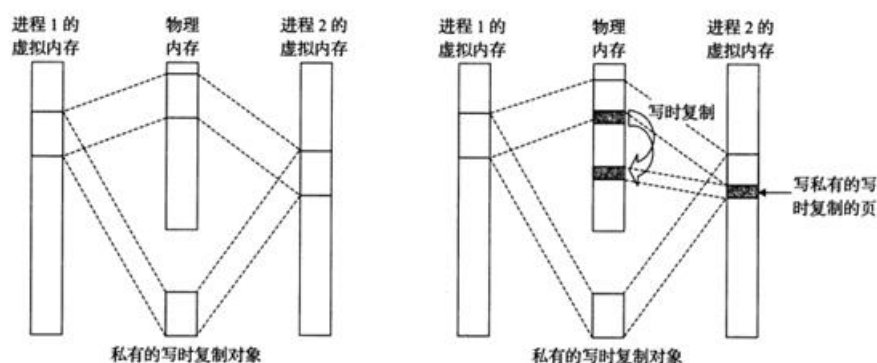


## 7.5 三级 Cache 支持下的物理内存访问

如下图所示，在三级缓存支持下能够在很大程度上减少 PTE 检索和获取主存或磁盘文件的耗时。Cache 中能够同时存储 PTE 文件或者缓存文件，通过这种机制再搭配上 TLB 就可以使得机器在翻译地址的时候的性能得以充分发挥。



## 7.6 hello 进程 fork 时的内存映射



shell 通过 fork 为 hello 创建新进程。当 fork 函数被当前进程调用时，内核为新进程创建各种数据结构，并分配给 hello 进程唯一的 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、区域结构和样表的原样副本。它将两个进程中的每个页面都标记为只读，并将每个进程中的每个区域结构都标记为写时复制。

当 fork 在新进程中返回时，新进程现在的虚拟内存刚好的和调用 fork 时存在的虚拟内存相同。当这两个进程中的任一个后来进行写操作时，写时复制机制就会创建新页面，因此，也就是为每个进程保持了私有地址空间的概念。

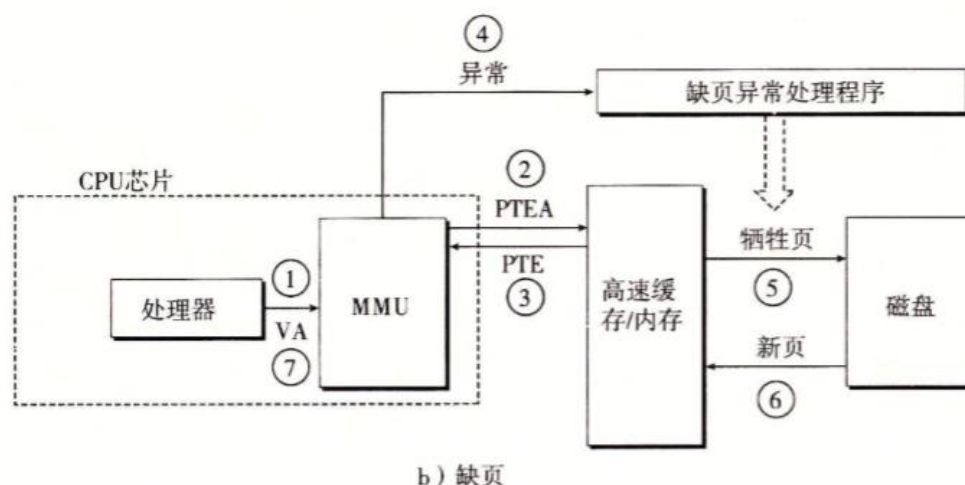
## 7.7 hello 进程 execve 时的内存映射

hello 进程时 execve 分为一下几个步骤：

1. 删除子进程中的用户区域，删除内存片和已有的区域结构。
2. 映射私有区域。为 hello 的代码、数据、bss 和栈区域创建新的区域结构。所有这些新的区域都是私有的、写时复制的。代码和数据区域被映射为 hello 文件中的.text 和.data 区。bss 区域是请求二进制零的，映射到匿名文件，其大小包含在 hello 中。栈和堆区域也是请求二进制零的，初始长度为零。
3. 映射共享区域。如果 hello 程序与共享对象（或目标）链接，比如标准 C 库 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。

## 7.8 缺页故障与缺页中断处理





如上图所示，是缺页故障与缺页中断流程的一个示意，下面分步解释。

1. 处理器生成一个虚拟地址，并传送给 MMU。
2. MMU 生成 PTE 地址，并从高速缓存/贮存请求得到它。
3. 高速缓存/贮存向 MMU 返回 PTE。
4. 返回的 PTE 中的有效位是零，MMU 出发了一次异常，传输给 CPU 中控制到操作系统内核中的缺页异常处理程序。（中断）
5. 缺页处理程序确定出物理内存中的牺牲页，如果这个页面已经被修改了，则把它换出到磁盘。（缺页处理）
6. 缺页处理程序调入新的页面，并更新内存中的 PTE。

## 7.9 动态存储分配管理

1. 动态内存分配器维护着一个进程的虚拟内存区域，称为堆(heap)。系统之间细节不同，但是不失通用性，假设堆是一个请求二进制零的区域，它紧接在未初始化的数据区域后开始，并向上生长（向更高的地址）。对于每个进程，内核维护着一个变量 brk，它指向堆的顶部。
2. 分配器将堆视为一组不同大小的块(block)的集合来维护。每个块就是一个连续的虚拟内存片(chunk)，要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可用来分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。
3. 基本方法：这里指的基本方法应该是在合并块的时候使用到的方法，有最佳适配和第二次适配还有首次适配方法，首次适配就是指的第一次遇到的就直接适配分配，第二次顾名思义就是第二次适配上的，最佳适配就是

搜索完以后最佳的方案，当然这种的会在搜索速度上大有降低。

4. 策略：这里的策略指的就是显式的链表的方式分配还是隐式的标签引脚的方式分配还是分离适配，带边界标签的隐式空闲链表分配器允许在常数时间内进行对前面块的合并。这种思想是在每个块的结尾处添加一个脚部，其中脚部就是头部的一个副本。如果每个块包括这样一个脚部，那么分配器就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，这个脚部总是在距当前块开始位置一个字的距离。显式空间链表就是将空闲块组织为某种形式的显式数据结构。因为根据定义，程序不需要一个空闲块的主体，所以实现这个数据结构的指针可以存放在这些空闲块的主体里面。例如，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，使首次适配的分配时间从块总数的线性时间减少到了空闲块数量的线性时间。为了分配一个块，必须确定请求的大小类，并且对适当的空闲链表做首次适配，查找一个合适的块。如果找到了一个，那么就（可选地）分割它，并将剩余的部分插入到适当的空闲链表中。如果找不到合适的块，那么就搜索下一个更大的大小类的空闲链表。如此重复，直到找到一个合适的块。如果空闲链表中没有合适的块，那么就向操作系统请求额外的堆内存，从这个新的堆内存中分配出一个块，将剩余部分放置在适当的大小类中。要释放一个块，我们执行合并，并将结果放置到相应的空闲链表中。

## 7.10 本章小结

本章围绕计算机的存储结构和 `hello` 程序涉及到的存储问题，涉及到了高速缓存，贮存和磁盘等结构，主要讨论了虚拟内存的思想和在 `hello` 程序中的实现。同时，本章讨论了关于如何通过 TLB 和三级 Cache 来加速虚拟内存寻址以及获取数据的过程，再次强调了多层次的存储体系结构对于现代计算机方方面面的影响。

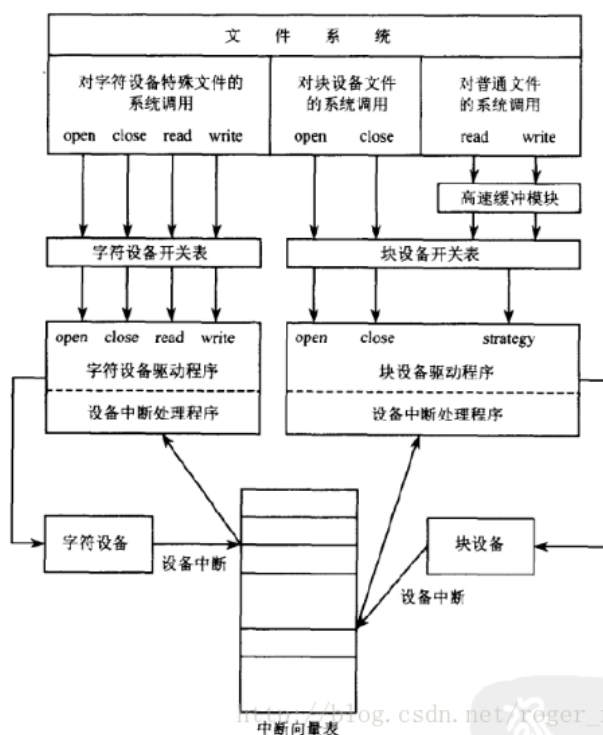
（第 7 章 2 分）

## 第8章 hello 的 IO 管理

### 8.1 Linux 的 IO 设备管理方法

#### 1. IO 设备管理的抽象层

分时并行多任务系统中，为了合理利用系统设备，达到一定的目标，不允许进程自行决定设备的使用，而是由系统按一定原则统一分配、管理。进程要进行 IO 操作时，需向操作系统提出 IO 请求，然后由操作系统根据系统当前的设备使用状况，按照一定的策略，决定对改进程的设备分配。设备的应用领域不同，其物理特性各异，但某些设备之间具有共性，为了简化对设备的管理，可对设备分类，或对同类设备采用相同的管理策略，比如Linux主要将外部 IO 设备分为字符设备和块设备(又被称为主设备)，而同类设备又可能同时存在多个，故而要定位具体设备还需提供“次设备号”。



#### 2. 设备分配使用方式

独占设备是指被分配给一个进程后，就被该进程独占使用，必须等到该进程退出后，其他进程才能进入，。

共享设备是指可以由多个进程交替使用的设备，如磁盘。

对于独占设备通常采用静态分配的方式，在一个作业开始执行前进程

独占设备的分配，一旦把某独占设备分配给作业，就被该作业独占（永久地分配给该作业），直到作业结束撤离时，才由操作系统将分配给作业的独占设备收回，静态分配方式实现简单，但是设备利用效率不高。

### 3. 设备无关性

设备无关性是指当在应用程序中使用某类设备时，不直接指定具体使用哪个设备，而只指定使用哪类设备，由操作系统为进程分配具体的一个该类设备。设备无关性功能可以使应用程序的运行不依赖于特定设备是否完好、是否空闲，而由系统合理地进行分配，从而保证程序的顺利进行。

为了便于描述设备无关性，引入逻辑设备和物理设备这两个概念。逻辑设备指示一类设备，物理设备指示一台具体的设备。类似于虚拟内存和物理内存的对应关系。这要求存储管理要具备地址变换的功能。

设备无关性显然也是操作系统关于 I/O 设备管理提供的一层抽象层，让上层程序无需关心使用 I/O 设备的细节情况。

### 4. 虚拟设备技术

系统中独占设备的数量有限，且对独占设备的分配往往采用静态分配方式，这样做不利于提高系统效率，这些设备只能分配给一个作业，且在作业的整个运行期间一直被占用，直至作业结束后才能被释放。但是一个作业往往不能充分利用该设备，在独占设备被某个作业占用期间，往往只有一部分时间在工作，其余时间处于不工作的空闲状态，因此设备利用率低，其他申请该设备的作业因得不到该设备而无法工作，降低了系统效率。

另一方面，独占设备往往是低速设备，因此，在作业执行过程中，如果直接使用该类设备会因为数据传输的低速大大延长作业的执行时间。

为了克服独占设备的这些缺点，可以采用虚拟设备技术，即用为每个独占设备配备相应的高速缓冲区（如高速磁盘、内存中专门划分的存储区域）来模拟该独占设备同时并存的多个虚拟接口。

## 8.2 简述 Unix I/O 接口及其函数

### 1. UNIX I/O 接口

a) 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。应用程序只需记住这个描述符。

b) 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）。头文件 `<unistd.h>` 定义了常量 `STDIN_FILENO`、`STDOUT_FILENO`

和 `STDERR_FILENO`，它们可用来代替显式的描述符值。

- c) 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` 操作，显式地设置文件的当前位置为 `K`。
- d) 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~m` 时执行读操作会触发一个称为 `end-of-file(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”。类似地，写操作就是从内存复制 `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`。
- e) 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，内核都会关闭所有打开的文件并释放它们的内存资源。

## 2. UNIX IO 函数:

- a) 打开文件: `int open(char *filename, int flags, mode_t mode);`
- b) 关闭文件: `int close(int fd);`
- c) 读文件: `ssize_t read(int fd, void *buf, size_t n);`
- d) 写文件: `ssize_t write(int fd, const void *buf, size_t n);`

## 8.3 printf 的实现分析

### 1. 分析 printf 函数:

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

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

## 2. 分析 vsprintf

```
int vsprintf(char *buf, const char *fmt, va_list args)
{
    int len;
    int i;
    char * str;
    char *s;
    int *ip;
    int flags;          /* flags to number() */
    int field_width;    /* width of output field */
    int precision;      /* min. # of digits for integers; max number of
chars for from string */
    int qualifier;      /* 'h', 'l', or 'L' for integer fields */

    for (str=buf ; *fmt ; ++fmt) {
        if (*fmt != '%') {
            *str++ = *fmt;
            continue;
        }

        /* process flags */
        flags = 0;

        repeat:
            ++fmt;      /* this also skips first '%' */
            switch (*fmt) {
                case '-': flags |= LEFT; goto repeat;
                case '+': flags |= PLUS; goto repeat;
                case ' ': flags |= SPACE; goto repeat;
                case '#': flags |= SPECIAL; goto repeat;
                case '0': flags |= ZEROPAD; goto repeat;
            }

            /* get field width */
            field_width = -1;

            if (is_digit(*fmt))
                field_width = skip_atoi(&fmt);
            else if (*fmt == '*') {
```

```
    /* it's the next argument */
    field_width = va_arg(args, int);
    if (field_width < 0) {
        field_width = -field_width;
        flags |= LEFT;
    }
}

/* get the precision */
precision = -1;

if (*fmt == '.') {
    ++fmt;
    if (is_digit(*fmt))
        precision = skip_atoi(&fmt);
    else if (*fmt == '*') {
        /* it's the next argument */
        precision = va_arg(args, int);
    }
    if (precision < 0)
        precision = 0;
}

/* get the conversion qualifier */
qualifier = -1;

if (*fmt == 'h' || *fmt == 'l' || *fmt == 'L') {
    qualifier = *fmt;
    ++fmt;
}

switch (*fmt) {
case 'c':
    if (!(flags & LEFT))
        while (--field_width > 0)
            *str++ = ' ';
    *str++ = (unsigned char) va_arg(args, int);
    while (--field_width > 0)
        *str++ = ' ';
    break;
```

```
case 's':
s = va_arg(args, char *);
len = strlen(s);
if (precision < 0)
    precision = len;
else if (len > precision)
    len = precision;
if (!(flags & LEFT))
    while (len < field_width--)
        *str++ = ' ';
for (i = 0; i < len; ++i)
    *str++ = *s++;
while (len < field_width--)
    *str++ = ' ';
break;

case 'o':
str = number(str, va_arg(args, unsigned long), 8,
    field_width, precision, flags);
break;

case 'p':
    if (field_width == -1) {
        field_width = 8;
        flags |= ZEROPAD;
    }
    str = number(str,
        (unsigned long) va_arg(args, void *), 16,
        field_width, precision, flags);
    break;

case 'x':
    flags |= SMALL;

case 'X':
    str = number(str, va_arg(args, unsigned long), 16,
        field_width, precision, flags);
    break;

case 'd':
case 'i':
    flags |= SIGN;
```



```

        case 'u':
            str = number(str, va_arg(args, unsigned long), 10,
                          field_width, precision, flags);
            break;

        case 'n':
            ip = va_arg(args, int *);
            *ip = (str - buf);
            break;

        default:
            if (*fmt != '%')
                *str++ = '%';
            if (*fmt)
                *str++ = *fmt;
            else
                --fmt;
            break;
        }
    }

    *str = '\0';

    return str-buf;
}

```

这个函数的作用是把后面的参数加到字符串里面然后输出字符串的长度。

### 3. write 函数

```

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

```

通过查询资料可以知道 `int INT_VECTOR_SYS_CALL` 的作用是调用 `sys_call` 函数，这个函数驱动的显示器。

### 4. sys\_call 函数

```
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
```

代码里面的 `call` 是访问字库模板并且获取每一个点的 RGB 信息最后放入到 `eax` 也就是输出返回的应该是显示 `vram` 的值，然后系统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

```
int getchar(void)
{
    char c;
    return (read(0, &c, 1) == 1) ? (unsigned char)c : EOF
}
```

通过分析代码，发现当用户按键时，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。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

输入 / 输出(I/O) 是在主存和外部设备（例如磁盘驱动器、终端和网络）之间复制数据的过程。输入操作是从 I/O 设备复制数据到主存，而输出操作是从主存复制数据到 I/O 设备。所有语言的运行时系统都提供执行 I/O 的较高级别的工具。例如，ANSI C 提供标准 I/O 库，包含像 `printf` 和 `scanf` 这样执行带缓冲区的 I/O 函数。C++ 语言用它的重载操作符 `<<`（输入）和 `>>`（输出）提供了类似的功能。在 Linux 系统中，是通过使用由内核提供的系统级 Unix I/O 函数来实现这些

较高级别的 I/O 函数的。

这章通过 hello 的 printf 和 getchar 详细分析了 LINUX 的 IO。

(第 8 章 1 分)

## 结论

1. 用计算机系统的语言，逐条总结 hello 所经历的过程：
  - a) 首先 hello.c 通过 I/O 设备如键盘等经过总线存入主存。
  - b) hello.c 通过预处理器生成预处理文本文件 hello.i。
  - c) 通过编译器生成汇编代码 hello.s。
  - d) 汇编代码通过汇编器生成可重定位的目标文件 hello.o。
  - e) 最后连接器根据可重定位目标文件，与库函数进行链接，生成可执行文件 hello。
  - f) 后 shell 调用 fork 函数创建一个新运行的子进程。
  - g) 加载器删除子进程现有的虚拟内存段，然后使用 mmap 函数创建新的内存区域，并创建一组新的代码、数据、堆和栈段。
  - h) TLB 和分级页表大大加速了 hello 地址翻译的过程，计算机系统各抽象级中广泛存在的 cache 减少了 hello 读取数据所需要的时间。
  - i) 当需要结束 hello 进程的时候，操作系统通过终端重新获得控制，并向 hello 发送 SIGINT 信号，hello 进程结束，释放内存空间。
2. 在完成大作业的过程中，我在计算机系统中进行了一次大漫游，从 cpu 到缓存到内存到磁盘，从 mmu 程序到操作系统到用户程序，我对整个计算机系统有了更深的了解。在这趟旅途中，给我影响很深的是 csapp 中反复提到的各层级抽象的精神，在这种精神的指导下，我能够在不同的抽象级别认识计算机系统的实现，这也给我的学习和思维带来了很大的帮助。计算机发展到现在，结合了人类最顶尖的智慧，这也激起了我的学习欲望，以及为计算机事业贡献自己智慧的决心。

(结论 0 分，缺失 -1 分，根据内容酌情加分)

## 附件

文件	注释
hello.c	hello 源文件
hello.i	hello
hello.s	hello 汇编代码文件
hello.o	hello 可重定位目标文件
hello	可执行文件
objectdump	hello.o 的反汇编文件
exedump	hello 的反汇编文件
objectelf	hello.o 的 elf 格式
exeelf	hello 的 elf 格式

(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] UNIX IO 函数:[https://blog.csdn.net/lingjun\\_love\\_tech/article/details/40706599](https://blog.csdn.net/lingjun_love_tech/article/details/40706599)
- [2] Linux 内核中的 printf 实现:  
<https://www.cnblogs.com/chenglei/archive/2009/08/06/1540702.html>
- [3] Linux 硬链接与软链接: <https://www.cnblogs.com/crazylqy/p/5821105.html>
- [4] Linux 虚拟地址和物理地址的映射: <https://www.cnblogs.com/big-devil/p/8590228.html>
- [5] 汇编语言指令集之条件转移指令:  
[https://blog.csdn.net/qq\\_36215315/article/details/79879391](https://blog.csdn.net/qq_36215315/article/details/79879391)
- [6] Linux 内核: IO 设备的抽象管理方式:  
[https://blog.csdn.net/roger\\_ranger/article/details/78473886](https://blog.csdn.net/roger_ranger/article/details/78473886)

**(参考文献 0 分, 缺失 -1 分)**