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　1170300505

班　　级　1703005

学　　生　叶成豪

指 导 教 师　吴锐

计算机科学与技术学院

2018 年 12 月

摘　要

摘要是论文内容的高度概括，应具有独立性和自含性，即不阅读论文的全文，就能获得必要的信息。摘要应包括本论文的目的、主要内容、方法、成果及其理论与实际意义。摘要中不宜使用公式、结构式、图表和非公知公用的符号与术语，不标注引用文献编号，同时避免将摘要写成目录式的内容介绍。

关键词：hello；总结；编译；执行；

本文介绍了 hello 程序从编写到运行再到回收的全部过程。编写本文章的目的主要是回顾本学期所学的知识，将所有的内容通过 hello 程序这个例子穿针引线融会贯通。在回顾的过程中发现新的认识和原先理解错误的地方。

（摘要 0 分，缺失-1 分，根据内容精彩称都酌情加分 0-1 分）

目　录

第 1 章 概述 ..... - 4 -

1.1 HELLO 简介 ..... - 4 -
1.2 环境与工具 ..... - 4 -
1.3 中间结果 ..... - 4 -
1.4 本章小结 ..... - 4 -

第 2 章 预处理 ..... - 6 -

2.1 预处理的概念与作用 ..... - 6 -
2.2 在 UBUNTU 下预处理的命令 ..... - 7 -
2.3 HELLO 的预处理结果解析 ..... - 7 -
2.4 本章小结 ..... - 8 -

第 3 章 编译 ..... - 9 -

3.1 编译的概念与作用 ..... - 9 -
3.2 在 UBUNTU 下编译的命令 ..... - 9 -
3.3 HELLO 的编译结果解析 ..... - 9 -
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 ..... - 22 -
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 ..... - 27 -
5.6 HELLO 的执行流程 ..... - 29 -
5.7 HELLO 的动态链接分析 ..... - 29 -
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 - 33 -
6.4 Hello 的 execve 过程 - 33 -
6.5 Hello 的进程执行 - 34 -
6.6 hello 的异常与信号处理 - 35 -
6.7 本章小结 - 38 -
第 7 章 HELLO 的存储管理 - 39 -
7.1 hello 的存储器地址空间 - 39 -
7.2 Intel 逻辑地址到线性地址的变换-段式管理 - 40 -
7.3 Hello 的线性地址到物理地址的变换-页式管理 - 41 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换 - 41 -
7.5 三级 Cache 支持下的物理内存访问 - 43 -
7.6 hello 进程 fork 时的内存映射 - 43 -
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 - 46 -
第 8 章 HELLO 的 IO 管理 - 48 -
8.1 Linux 的 IO 设备管理方法 - 48 -
8.2 简述 Unix IO 接口及其函数 - 48 -
8.3 printf 的实现分析 - 48 -
8.4 getchar 的实现分析 - 51 -
8.5 本章小结 - 51 -
结论 - 52 -
附件 - 53 -
参考文献 - 54 -

第 1 章 概述

1.1 Hello 简介

Hello 从无到有经历了许许多多，从键盘一个一个字符键入开始，**hello** 的生命从此孕育，**#include** 是生命的开端，**hello** 从这里开始，**return 0** 是生命的休止符，**hello** 的生命从这里结束。预处理使 **hello** 抛下了承重的“#”，编译使 **hello** 丢弃了高级语言的格式，回归了更接近本我的汇编代码，一行一行的汇编代码展示了无比详细的细节，但是 **hello** 的成长还未结束，汇编使 **hello** 丢下了文本的格式，来到机器代码的广阔世界中，在经过链接器的处理，**hello** 的细节被描摹完整，自此刻起，一个完整的生命就此诞生。

Hello 更加精彩的人生还在后面，**bash** 为 **hello** 创建新的进程，为 **hello** 腾出舞台和空间，在 **bash** 中 **execve** 将 **hello** 从磁盘加载到内存当中。在加载中，内存、**cache**、**TLB**、四级页表各显神通。**hello** 的人生即将迎来巅峰。**hello** 在激烈的争夺中抢占了 **CPU** 的一点青睐，在短暂的时间中高速的执行着一条条的指令，虽然短暂，但是 **hello** 在 **bash** 中留下了自己的足迹。表演虽然只有一瞬间，但是留下的却是永恒。**hello** 结束了它的一生，**OS** 和 **bash** 让他体面的离开，在谢幕之后回收了僵死的 **hello**。

1.2 环境与工具

X64 CPU；2GHz；2G RAM；256GHD Disk 以上

Windows7 64 位以上；VirtualBox/Vmware 11 以上；Ubuntu 16.04 LTS 64 位/优麒麟 64 位；

Visual Studio 2010 64 位以上；TestStudio；Gprof;Valgrind 等

1.3 中间结果

hello	经过链接后的可执行文件
hello.c	源代码
hello.i	经过预处理的源代码
hello.o	进过汇编得到的可重定位文件
hello.s	进过编译得到的汇编代码
hello.o_obj.txt	hello.o 文件经过反汇编得到的文本

hello_obj.txt	hello 文件经过反汇编得到的文本
hello_elf.txt	hello 的 elf 文件信息
hello_section_header.txt	hello 文件的节信息

1.4 本章小结

Hello 的一生经历了预处理、编译、汇编、链接才得到了最后的可执行文件，而这只是它的第一步。Bash 为它 fork 子进程，为它 execve，hello 的程序被加载到内存当中，hello 开始了它的表演。最后，Hello 在执行完最后一条指令后，hello 成为了一个僵尸进程，等待它的父进程的回收。

（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：在编译之前对源文件进行处理。

作用：

1. 宏定义：

宏定义又称为宏代换、宏替换，简称宏。

格式： #define 标识符 文本

预处理过程也叫宏展开:将宏名替换为文本(这个文本可以是字符串、可 是代码等）。例如#define PI 3.1415926 把程序中全部的标识符 PI 换成 3.1415926。

2. 文件包含：

一个文件包含另一个文件的内容

格式：#include "文件名" 或 #include <文件名>

编译时以包含处理以后的文件为编译单位，被包含的文件是源文件的一部分。编译以后只得到一个目标文件.obj 被包含的文件又被称为“标题文件”或“头部文件”、“头文件”，并且常用.h 作扩展名。修改头文件后所有包含该文件的文件都要重新编译。

头文件的内容除了函数原型和宏定义外，还可以有结构体定义，全局变量定义：

（1）一个#include 命令指定一个头文件；

（2）文件 1 包含文件 2，文件 2 用到文件 3，则文件 3 的包含命令#include 应放在文件 1 的头部第一行；

（3）包含可以嵌套；

（4）<文件名>称为标准方式，系统到头文件目录查找文件，

"文件名"则先在当前目录查找，而后到头文件目录查找；

（5）被包含文件中的静态全局变量不用在包含文件中声明。

3. 条件编译：

格式：（1）

#ifdef 标识符

程序段 1 #else 程序段 2 #endif

或 #ifdef 程序段 1 #endif

当标识符已经定义时，程序段 1 才参加编译。

2.2 在 Ubuntu 下预处理的命令

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

```
ych@ubuntu:~/dazuoye$ gcc -E hello.c -o hello.i
ych@ubuntu:~/dazuoye$ ls
hello.c  hello.i
```

2-1 预处理指令

2.3 Hello 的预处理结果解析

hello.c 文件经过预处理之后根据第一行的#include<stdio.h>命令告诉预处理器读取系统头文件 stdio.h 中的内容，并且直接插入程序文本中，除此之外，如果程序源代码中还有注释的话，经过预处理也会将所有的注释删除。最后得到的结果是另一个 C 程序，被命名为 hello.i。

hello.c

```
// 大作业的 hello.c 程序
// gcc -m64 -no-pie -fno-PIC hello.c
// 程序运行过程中可以按键盘，如不停乱
C等。
// 可以 运行 ps  jobs  pstree fg 等命

#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
\n");
                exit(1);
        }
        for(i=0;i<10;i++)
        {
                printf("Hello %s %s
                sleep(sleepsecs);
        }
        getchar();
        return 0;
}
```

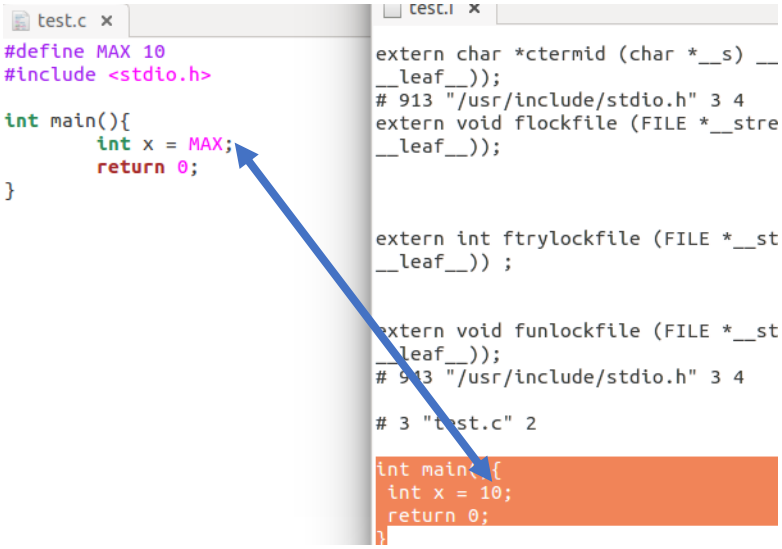
hello.i

```
extern int rpmatch (const char *__response) __attribute__
((__nothrow__ , __leaf__)) __attribute__ ((__nonnull__ (1)))
# 899 "/usr/include/stdlib.h" 3 4
extern int getsubopt (char **__restrict __optionp,
                      char *const *__restrict __tokens,
                      char **__restrict __valuep)
     __attribute__ ((__nothrow__ , __leaf__)) __attribute__
((__nonnull__ (1, 2, 3))) ;
# 951 "/usr/include/stdlib.h" 3 4
extern int getloadavg (double __loadavg[], int __nelem)
     __attribute__ ((__nothrow__ , __leaf__)) __attribute__
((__nonnull__ (1)));


# 1 "/usr/include/x86_64-linux-gnu/bits/stdlib-float.h" 1 3 4
# 956 "/usr/include/stdlib.h" 2 3 4
# 968 "/usr/include/stdlib.h" 3 4

# 9 "hello.c" 2

int sleepsecs=2.5;

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

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

2-2 预处理结果

用文本编辑器打开 hello.i 与 hello.c 进行比较，在 hello.i 的最后几行与 hello.c 的 main 函数是相同的，但是 hello.i 文件中没有#define 的预处理命令了，而且在主函数之前多了很多系统函数的调用。

如果在程序的开始有宏定义的话，那么预处理后所有宏都会被替换，例如下面用宏常量 MAX 对 x 进行赋值，在预处理后直接用 10 进行赋值。



2-3 预处理的常量的处理

2.4 本章小结

预处理是 hello.c 生命周期的第一个部分，通过预处理器，可以对 c 源文件中的预处理命令进行预处理，将头文件插入到程序中，执行一些宏替换等等。除此之外还会将注释全部消除掉。最后得到的结果是以.i 结尾的 c 程序。

（第 2 章 0.5 分）

第 3 章 编译

3.1 编译的概念与作用

概念：编译是指是指把用高级程序设计语言书写的源程序，翻译成等价的机器语言格式目标程序的翻译程序。

作用：把用高级程序设计语言书写的源程序，翻译成等价的机器语言格式目标程序的翻译程序。编译程序属于采用生成性实现途径实现的翻译程序。它以高级程序设计语言书写的源程序作为输入，而以汇编语言或机器语言表示的目标程序作为输出。除此之外，还具有语法检查、调试措施、修改手段、覆盖处理、目标程序优化、不同语言合用以及人-机联系等重要功能。

3.2 在 Ubuntu 下编译的命令

gcc -S hello.i -o hello.s

```
ych@ubuntu:~/dazuoye$ ls
hello.c  hello.i
ych@ubuntu:~/dazuoye$ gcc -S hello.i -o hello.s
ych@ubuntu:~/dazuoye$ ls
hello.c  hello.i  hello.s
```

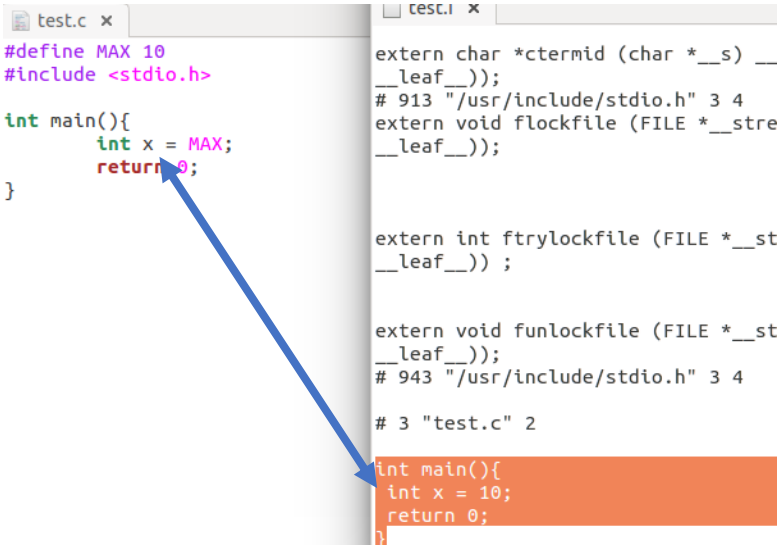
3-1 编译指令

3.3 Hello 的编译结果解析

3.3.1 数据

常量：在预处理过程中，预处理器已经将常量替换为对应的数值或语句。例如#define MAX 10，那么在源程序中出现的所有 MAX 都会被替换为 10。

截图中利用宏常量对 x 进行初始化，在预处理之后就直接用数字 10 对 x 进行初始化。

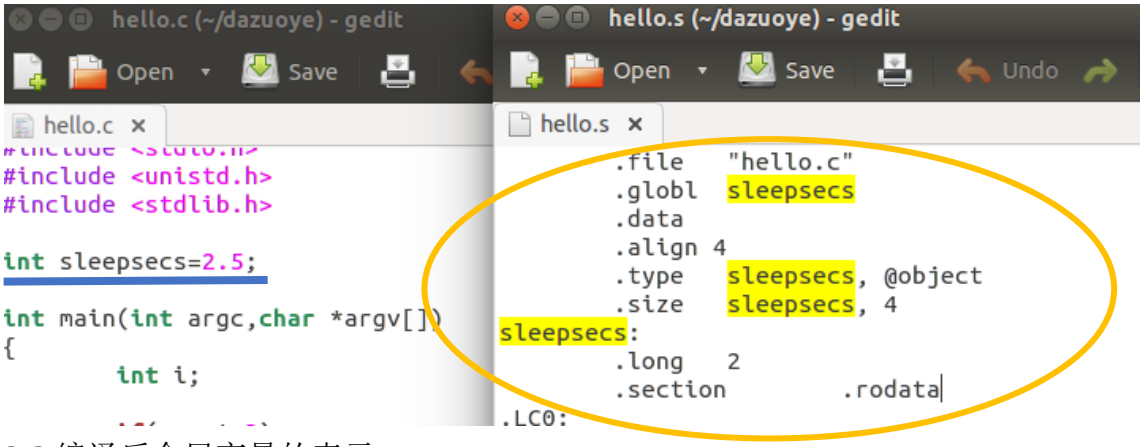


3-2 编译对常量的处理

变量：

1）全局变量

在 hello.c 中定义了一个 int 型全局变量，但是初始化的值为 2.5 因为是 int 型所以在经过预处理和编译后 sleepsecs 被赋值为 2.



3-3 编译后全局变量的表示

从截图中可以看到，在 hello.s 文件的开始出.globl 标识出了全局变量

.type sleepsecs, @object

.size sleepsecs, 4

sleepsecs:

.long 2

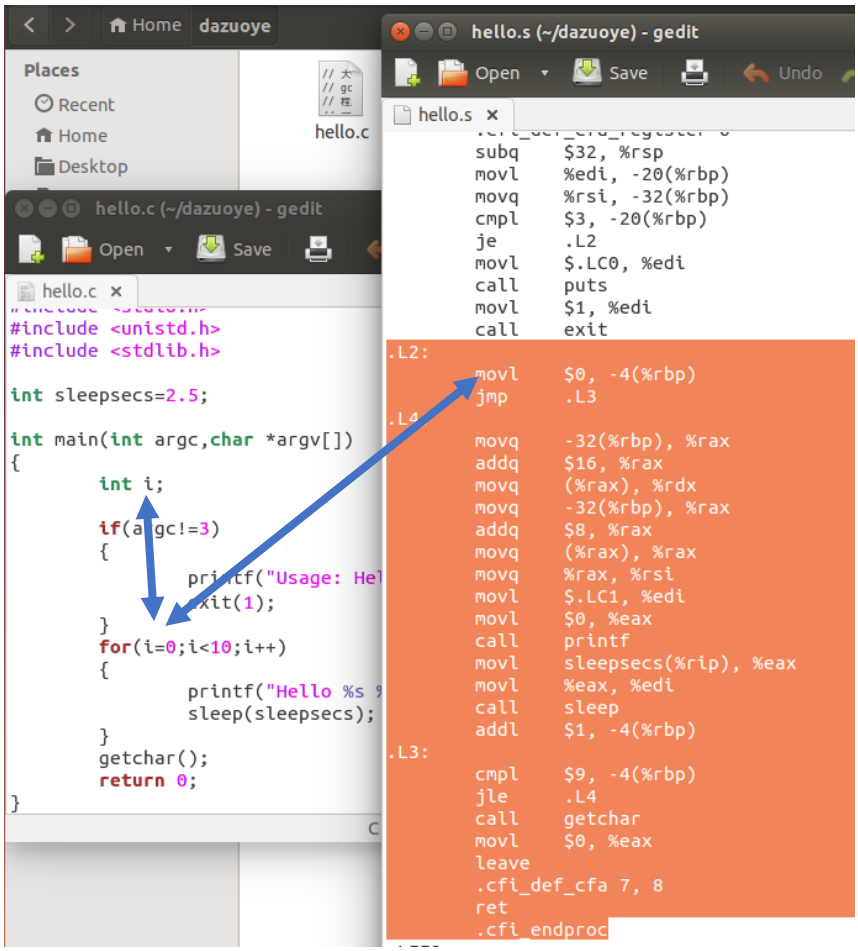
.section .rodata

从上述信息可以看出 sleepsecs 是大小为 4 字节的全局变量，大小为 2，存放在.rodata 节中

2）局部变量

编译器一般不会为局部变量分配专门的栈空间，只有在 c 源程序中使用到了‘&’时才会为相应的局部变量分配栈空间，一般来说，局部变量是直接存放在寄存器中的。如果局部变量是被初始化了的，那么这个局部变量的初始化值会直接体现在汇编代码中。

如下截图所示：



3-4 编译后局部变量的表示

虽然 i 是在 mian 函数开始时定义的，但是实际在汇编代码中是在 for 循环开始时才出现的，在.L2 中对 i 进行了初始化，i 的值存放在-4（%rbp）中。

为 i 赋值的汇编语句：

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

3）表达式

在 hello.c 中涉及的表达式主要是大于小于的判断。以 for 循环为例，比较的过程是由以下汇编代码实现的：

cmpl $9, -4(%rbp)

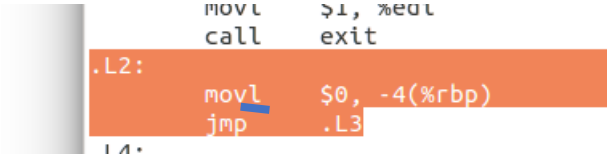
在执行完比较后，会设置相应的标志位 SF、ZF、OF 等，SF 代表符号位、ZF 代表是否为 0、OF 代表是否溢出等等。通过标志位来判断比较的结果，下一条汇编代码：

jle .L4

在 i 小于等于 9 时执行跳转。

4）类型

汇编代码并不会区分 c 程序中的数据类型，汇编代码通过汇编语句指定的数据大小对不同数据进行区分。在以 i 为例。



3-5 汇编代码数据没有类型

汇编代码通过汇编代码后缀区分大小，后缀的表示如下：

C 声明	Intel 数据类型	汇编代码后缀	大小(字节)
char	字节	b	1
short	字	w	2
int	双字	l	4
long	四字	q	8
char*	四字	q	8
float	单精度	s	4
double	双精度	l	8

图 3-1 C 语言数据类型在 x86-64 中的大小。在 64 位机器中，指针长 8 字节

3-6 汇编代码后缀

对 i 进行值的过程中用 l 作为后缀，代表指令操作的对象是 4 个字节，那么就向相应的地址写入 4 个字节的信息。

5）宏

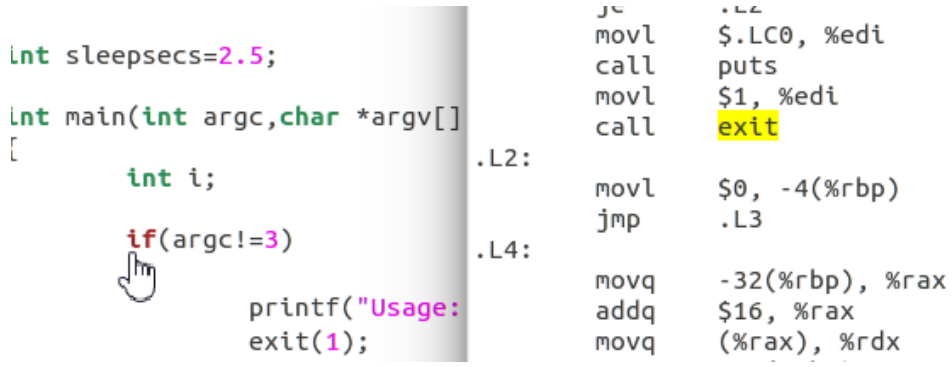
宏是在预处理阶段被处理的。

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

hello.c 文件引用了上面三个头文件，在经过预处理之后会将在程序中用到的库函数都直接插入到 hello.i 文件中，然后经过汇编则通过用指令 call 函数名来进行调用。在程序中应用了 exit（）函数，在经过编译之后，在对应的位置用指令 ***call exit*** 进行调用，而该函数的参数则通过%rdi 进行传递。

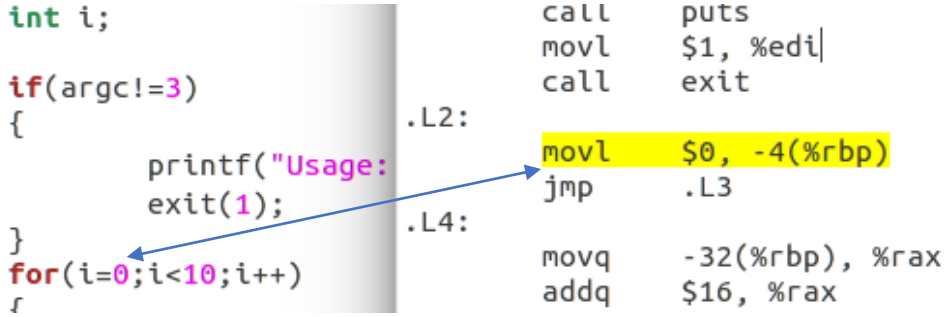


3-7 库文件的调用

3.3.2 赋值

1） ‘=’

C 语言的等号操作是通过 mov 指令进行操作的，因为 i 为 int 型所以 mov



3-8 赋值操作

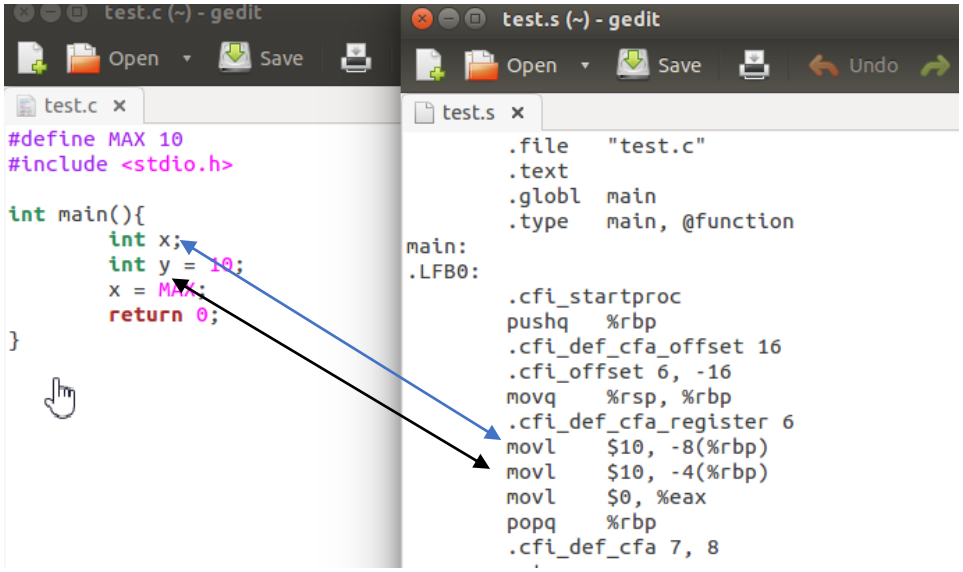
的后缀为 l，for 循环中 i 的初值被赋为 0，所以用进行初始化。其他的变量的

movl $0, -4(%rbp)

赋值都是类似的，通过 mov 指令进行。有些特殊情况，例如赋值为 0，则可以通过 xor 操作进行，这样做可以节省几个字节的空间。

2）赋初值

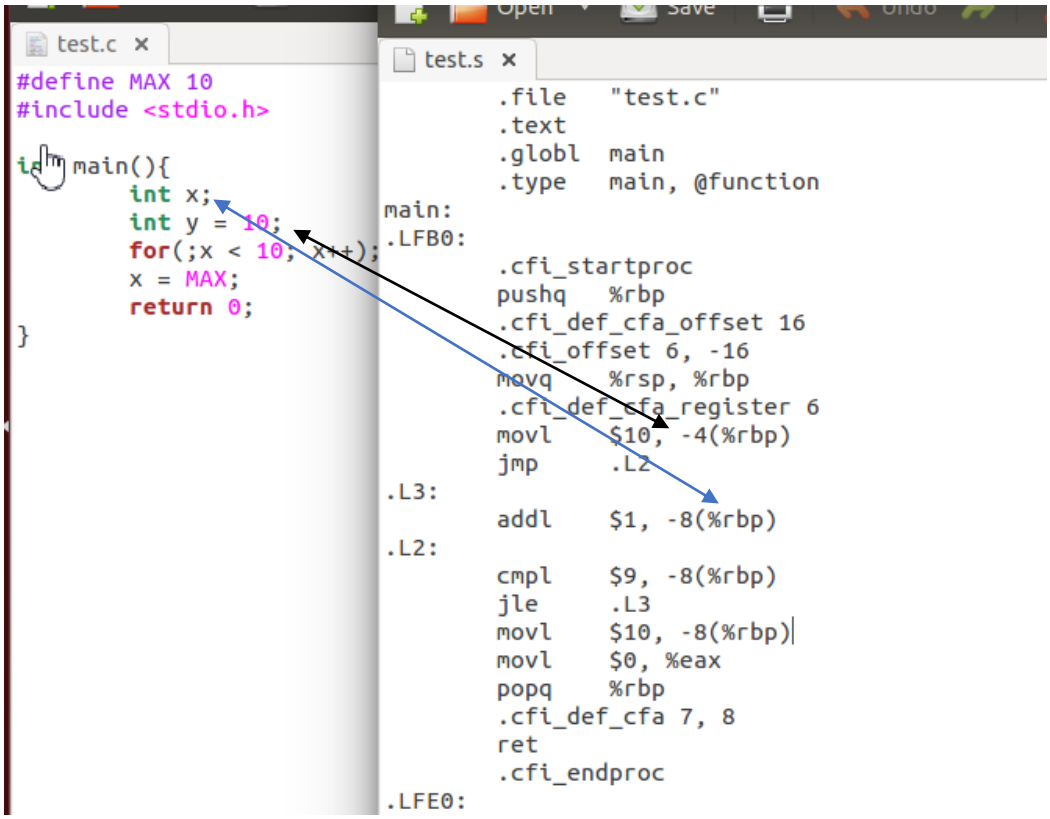
首先通过对寄存器%rsp 进行操作，在栈上分配相应的空间。然后直接通过 mov 操作对相应的栈上的空间进行赋值。如下图所示



3-9 对变量赋初值

3）未赋初值

经过测试发现，只要是在用到为赋初值的变量之前对这个变量进行赋值，其得到的结果都是一样的，都会在栈上分配空间是就赋值，而如果完全没有赋值操作，那么就只会在栈上分配空间，而不赋值，即其值是原来栈上的二进制数。是不确定的。测试的代码如下：



3-10 未赋初值的变量

3.3.3 算数操作

++：

++操作等价于 +1 操作，在汇编是中利用 ***add $1, 目的地址*** 实现的。如下图所示：

3-11++操作的实现

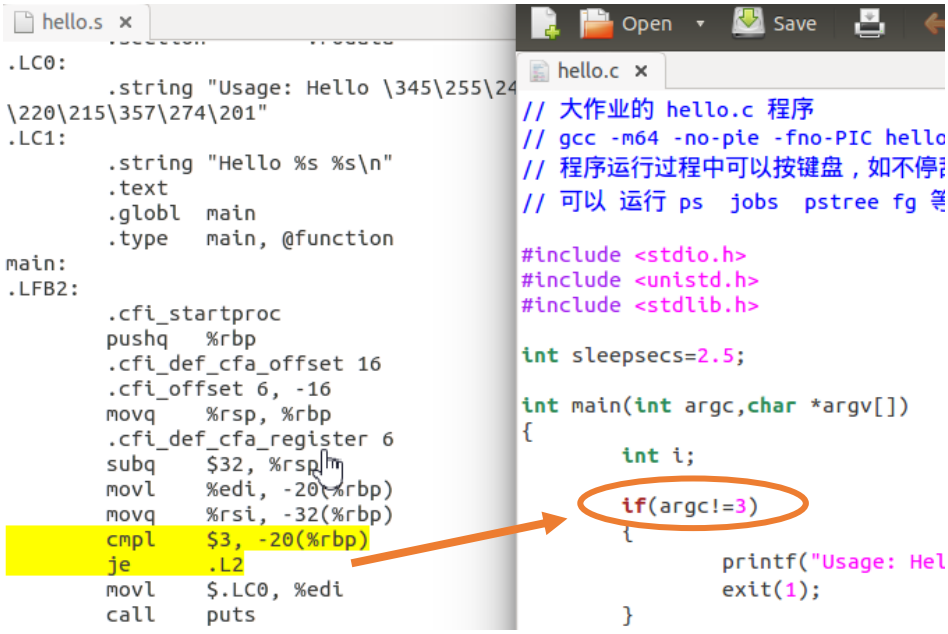
3.3.4 关系操作

1） ！=

关系操作是通过 cmp 这条指令来实现的，将两个要比较的数作为 cmp 的操作数，比较完成后 cmp 就会设置相应的标志位。如下截图所示,argv ！= 3 的操作通过指令实现

cmpl $3, -20(%rbp)

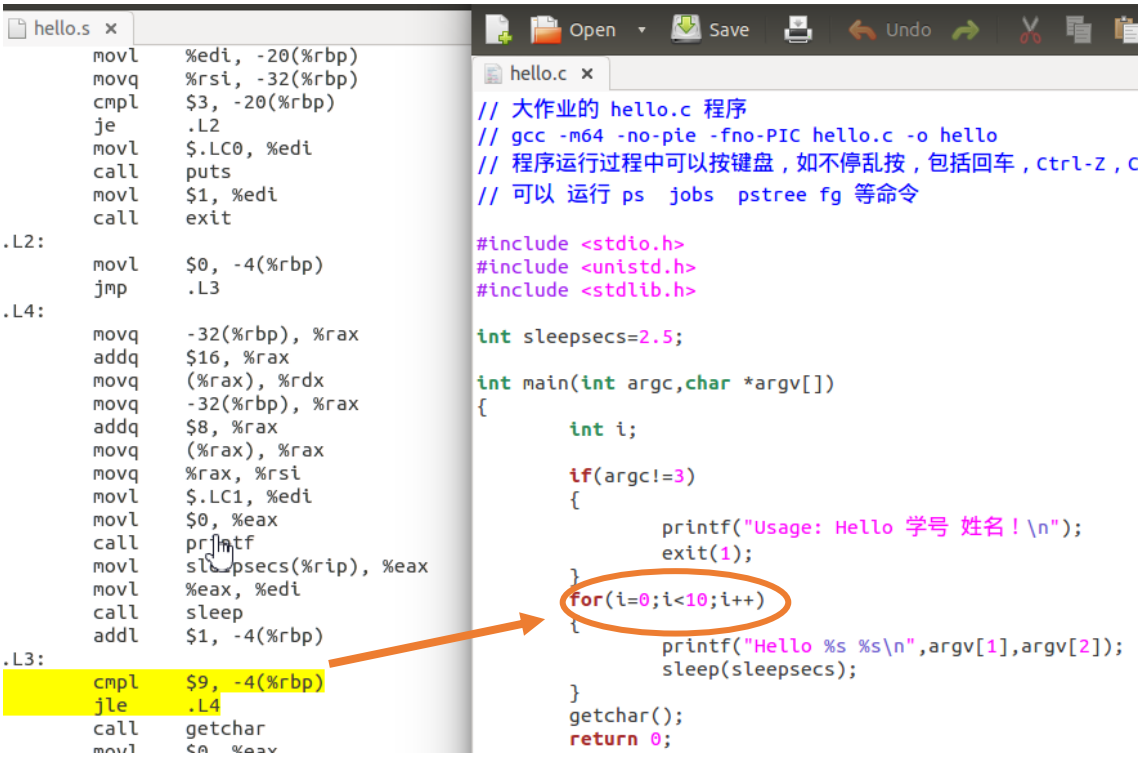
je .L2



3-12 ！=的实现

2） <

< 的实现和！=类似，也是通过 cmp 这条指令来实现的。但是在有些时候，编译器确定了数值不可能为负后可能会将比较的条件改为== 或者 ！= 。汇编代码的实现如下图所示。



3-13 < 符号的实现

3.3.5 数组操作

A[i]

数组的引用是通过指针的偏移来引用的，在 for 循环中调用了 printf 函数，参数为 argv[1]和 argv[2]在汇编代码中则是通过指针偏移将值传递到寄存器中。

```
movq    -32(%rbp), %rax
addq    $8, %rax
movq    (%rax), %rax
movq    %rax, %rsi
movl    $.LC1, %edi
movl    $0, %eax
```

%rax 原来指向的时 argv[0]，然后通过指令 addq $8, %rax 使%rax 指向 argv[1],然后将值传递给%rdi。对数组的引用就是基地址加上索引乘以类型的大小。

目标地址：a[0]+ sizeof(datatype)* index

3.3.6 控制转移

if

if 语句在汇编中是通过 jmp 指令完成的。首先 if 中会有一个表达式，这个表达式的求值的过程会设置标志位，而跳转是通过标志位来区分是否跳转的。以下是 for 循环的汇编代码：

```
.L2:
        movl    $0, -4(%rbp)
        jmp     .L3
.L4:
        movq    -32(%rbp), %rax
        addq    $16, %rax
        movq    (%rax), %rdx
        movq    -32(%rbp), %rax
        addq    $8, %rax
        movq    (%rax), %rax
        movq    %rax, %rsi
        movl    $.LC1, %edi
        movl    $0, %eax
        call    printf
        movl    sleepsecs(%rip), %eax
        movl    %eax, %edi
        call    sleep
        addl    $1, -4(%rbp)
.L3:
        cmpl    $9, -4(%rbp)
        jle     .L4
        call    getchar
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

3-14 if 语句在汇编中的实现

在 for 循环中每次都会对 i 的值进行更新，然后利用 cmpl 语句对 i 和 9 的值进行比较，比较的结果会影响标志位，然后 jle 则通过标志位判断 i 是否小于等于 9 如果 i 小于等于 9 那么就跳转到.L4 所代表的语句，如果不是，就继续顺序执行接下来的语句。

cmpl $9, -4(%rbp)
jle .L4

3.3.7 函数操作

函数的调用分为几个步骤，首先传递参数，第一个参数存放在%rdi 中，第二个参数存放在%rsi 中，前 6 个参数都可以存放在寄存器中传递，如果有 7 个及以上的参数就要通过栈来存放参数值。以调用 sleep 为例，通过如下语句调用 sleep。

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

首先向 sleep 传参，参数被传送到%edi 中，然后是 call 指令，call 首先将当前的 PC 压入栈中，然后将 PC 值更新为 sleep 函数所在的地方。然后执行下一条指令，程

序就会跳转到 sleep 中，sleep 可以利用调用者存放在寄存器中的参数。咱被调用的函数执行完一段程序之后，它会恢复栈指针的值，使它指向返回地址，也就是 call 指令压入的 PC 值，随后调用 ret 指令，ret 指令有两个操作，将 PC 设置为返回地址然后是%rsp 的值减 8，在 ret 执行完毕后，就重新回到了调用函数中。

3.4 本章小结

编译过程会将经过预处理的 c 程序编译为汇编代码，汇编代码是严格按照 c 程序的原意进行转换的，在汇编中没有数据类型这个概念，所以数据都是通过后缀来表示大小的。赋值操作大多是通过 mov 指令来实现的。算数操作在汇编中有一系列的操作来实现例如 add、sub 等等。关系操作是通过设置标志位来表示的。数组、指针、结构操作都是通过指针的偏移来完成的。而控制的转移与关系操作相类似，都是利用 cmp 这条指令设置标志位，然后利用 jmp 指令进行跳转。函数操作则有 call 和 ret 指令完成。

（第 3 章 2 分）

第 4 章 汇编

4.1 汇编的概念与作用

概念：把汇编语言翻译成机器语言的过程称为汇编

作用：汇编是将汇编语言翻译为机器语言的过程。一般而言，汇编生成的是目标代码，需要经链接器生成可执行代码才可以执行。

汇编语言是一种以处理器指令系统为基础的低级语言，采用助记符表达指令操作码，采用标识符表示指令操作数。作为一门语言，对应于高级语言的编译器，需要一个“汇编器”来把汇编语言原文件汇编成机器可执行的代码。

4.2 在 Ubuntu 下汇编的命令

gcc -c hello.s -o hello.o

```
ych@ubuntu:~/dazuoye$ gcc -c hello.s -o hello.o
ych@ubuntu:~/dazuoye$ ls
hello.c  hello.i  hello.o  hello.s
ych@ubuntu:~/dazuoye$
```

4-1 汇编的指令

4.3 可重定位目标 elf 格式

1）

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          448 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         13
  Section header string table index: 10
```

4-2 elf 头

ELF 头包括了该可重定位的一些基本信息，描述了生成该文件的系统的字的大小和字节序列等。包括 ELF 头的大小、目标文件的类型、机器类型、节头部表的文件偏移等。

2）

```
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
       000000000000007d  0000000000000000  AX       0     0     1
  [ 2] .rela.text        RELA             0000000000000000  000006b8
       00000000000000c0  0000000000000018          11     1     8
  [ 3] .data             PROGBITS         0000000000000000  000000c0
       0000000000000004  0000000000000000  WA       0     0     4
  [ 4] .bss              NOBITS           0000000000000000  000000c4
       0000000000000000  0000000000000000  WA       0     0     1
  [ 5] .rodata           PROGBITS         0000000000000000  000000c4
       000000000000002b  0000000000000000   A       0     0     1
  [ 6] .comment          PROGBITS         0000000000000000  000000ef
       000000000000002c  0000000000000001  MS       0     0     1
  [ 7] .note.GNU-stack   PROGBITS         0000000000000000  0000011b
       0000000000000000  0000000000000000           0     0     1
  [ 8] .eh_frame         PROGBITS         0000000000000000  00000120
       0000000000000038  0000000000000000   A       0     0     8
  [ 9] .rela.eh_frame    RELA             0000000000000000  00000778
       0000000000000018  0000000000000018          11     8     8
  [10] .shstrtab         STRTAB           0000000000000000  00000158
       0000000000000061  0000000000000000           0     0     1
  [11] .symtab           SYMTAB           0000000000000000  00000500
       0000000000000180  0000000000000018          12     9     8
  [12] .strtab           STRTAB           0000000000000000  00000680
       0000000000000037  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

4-3 节头部表

这一部分是节头部表，节头部表的位置有 ELF 头中的内容指出，在节头部表中指出各个节的大小偏移和位置，每个节都会有一个固定大小的条目。size 指出了每个节的大小，align 指出了每个节的对齐方式。Address 则指出每个节所在的地址。

3）

```
Offset          Info           Type           Sym. Value        Sym. Name + Addend
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

Relocation section '.rela.eh_frame' at offset 0x778 contains 1 entries:
  Offset          Info           Type           Sym. Value        Sym. Name + Addend
000000000020  000200000002 R_X86_64_PC32     0000000000000000 .text + 0
```

4-4 可重定位信息

这一段是可重定位信息。Offest 指的是在相应的节中的位置，例如第一行指的是在 rodata 接中偏移 000000000016 的地方，type 指出了是 PC 相对寻址，即相对于当前 PC 值的偏移。还是绝对地址引用。Name 则说明了该可重定位条目是存放在哪一个节中的。Addend 是一个有符号常数，一些类型的重定位条目使用它对被修改的引用的值做偏移调整。

4）

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

4-5symtab 节

这一部分是 symtab 节，这是一个符号表，它存放在程序中定义和引用的函数的和全局变量的信息。Value 这一列代表它的偏移大小，size 这一列代表这个函数或者全局函数的大小，如果为 0 说明这个全局变量或者函数不是在当前可重定位文件中定义的。需要在链接过程中在进行重定位。Bind 则说明了是一个是局部的还是全局的。利用 NDX 来标识每个节，例如 NDX = 1 标识.text 节, NDX = 3 标识.data 节。最后的 name 则标识这个符号的名字。

4.4 Hello.o 的结果解析

通过反汇编得到的汇编代码与编译得到的汇编代码相比主要有三个地方不同，首先是 hello.s 文件中会有文件的基本信息，如对齐方式，全局变量的符号，文件的文字等，除此之外，hello.s 中还有在程序中定义的字符串的信息，例如 hello.c 中 printf(“hello %s %s”)；就会在 hello.s 中显示。两个汇编代码表示也有所不同，通过编译生成的汇编代码都有后缀，以表示操作数的大小，而反汇编产生的代码则没有，编译产生的返回指令是 ret 而反汇编产生的则是 retq。但是这些不同没有本质上的区别，只是书写方式有所不同。除了以上两点外，hello.s 中对未重定位的函数或者变量都是用符号名进行表示,而反汇编产生的代码则直接用 00 进行填充。这是两者最大的不同之处。

```
ych@ubuntu:~/dazuoye$ objdump -d hello.o

hello.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 83 ec 20             sub    $0x20,%rsp
   8:   89 7d ec                mov    %edi,-0x14(%rbp)
   b:   48 89 75 e0             mov    %rsi,-0x20(%rbp)
   f:   83 7d ec 03             cmpl   $0x3,-0x14(%rbp)
  13:   74 14                   je     29 <main+0x29>
  15:   bf 00 00 00 00          mov    $0x0,%edi
  1a:   e8 00 00 00 00          callq  1f <main+0x1f>
  1f:   bf 01 00 00 00          mov    $0x1,%edi
  24:   e8 00 00 00 00          callq  29 <main+0x29>
  29:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  30:   eb 39                   jmp    6b <main+0x6b>
  32:   48 8b 45 e0             mov    -0x20(%rbp),%rax
  36:   48 83 c0 10             add    $0x10,%rax
  3a:   48 8b 10                mov    (%rax),%rdx
  3d:   48 8b 45 e0             mov    -0x20(%rbp),%rax
  41:   48 83 c0 08             add    $0x8,%rax
  45:   48 8b 00                mov    (%rax),%rax
  48:   48 89 c6                mov    %rax,%rsi
  4b:   bf 00 00 00 00          mov    $0x0,%edi
  50:   b8 00 00 00 00          mov    $0x0,%eax
  55:   e8 00 00 00 00          callq  5a <main+0x5a>
  5a:   8b 05 00 00 00 00       mov    0x0(%rip),%eax
  60:   89 c7                   mov    %eax,%edi
  62:   e8 00 00 00 00          callq  67 <main+0x67>
  67:   83 45 fc 01             addl   $0x1,-0x4(%rbp)
  6b:   83 7d fc 09             cmpl   $0x9,-0x4(%rbp)
  6f:   7e c1                   jle    32 <main+0x32>
  71:   e8 00 00 00 00          callq  76 <main+0x76>
  76:   b8 00 00 00 00          mov    $0x0,%eax
  7b:   c9                      leaveq
  7c:   c3                      retq
```

4-6 汇编语言与机器语言的对应关系

机器语言与汇编语言是一一对应的，在知道了机器语言是从哪个字节开始的，就可以映射出一串唯一的汇编指令。机器语言可以被唯一的解码为汇编代码。例如只有 pushq %rbx 是以字节值 53 开头。

在汇编语言中分支转移和函数调用都是通过绝对地址引用的，如下图所示

40054e: e8 55 fe ff ff callq 4003a8 <_init>

400553: 48 85 ed test %rbp,%rbp

400556: 74 1e je 400576 <__libc_csu_init+0x56>

调用函数和分支跳转都是采用的绝对寻址方式，但是观察机器代码，call 代表的机器代码时 e8，而 55 fe ff ff 显然与 4003a8 不同，55 fe ff ff 是 0x400553 – 0x4003a8 的值的补码表示。在机器码中，调用函数和分支转移都是通过相对寻址的方式，在当前 PC 值上进行加减操作。Jmp 以及一系列的跳转操作也是通过 PC 相对寻址来实现的。

4.5 本章小结

汇编的过程主要是将汇编代码转换成机器代码，然后添加一些重定位信息，生成可重定位文件。编译器会读取相应的重定位信息，为未定义的函数和全局变量添加条目，生成的可充定位中只有占位符，并没有实质的值。并且添加一些必要的信息，等待链接器将这些可重定位条目重定位。汇编器将汇编代码转换为机器码的过程并不是完全一一对应的，在机器代码中大多数的函数调用以及分支跳转都是通过 PC 相对寻址来实现的，而在汇编代码中则是直接通过使用 00 作为占位符来表示。

可以用 readelf 这个工具读取可重定位文件的头文件信息。头文件信息中包含了 ELF 头，节头部表和各个节。ELF 头指出了生成该文件的系统的字的大小和字节序列，还说明了节头部表的位置，在节头部表中则说明了各个节的位置和偏移量

（第 4 章 1 分）

第 5 章 链接

5.1 链接的概念与作用

概念：

链接器（Linker）是一个程序，将一个或多个由编译器或汇编器生成的目标文件外加库链接为一个可执行文件。

作用：简单的讲，链接器的工作就是解析未定义的符号引用，将目标文件中的占位符替换为符号的地址。链接器还要完成程序中各目标文件的地址空间的组织，这可能涉及重定位工作。

链接器使用每个目标模块中的重定位信息和符号表，来解析所有未定义标签。这种引用发生在分支指令、跳转指令和数据寻址处，所以这个程序的工作非常像一个编辑器：它寻找所有旧地址并用新地址取代它们：编辑是“链接编辑器”或链接器名字的简称。采用链接器的原因是修补代码比重新编译和汇编要快得多。

如果所有外部引用都解析完，链接器接着决定每个模块将要占用的内存位置。MlIPS 在内存中为程序和数据分配空间的方式。因为文件是单独汇编的，所以汇编器不可能知道该模块的指令和数据相对于其他模块而言将会被放到哪里。当链接器将一个模块放到内存中的时候，所有绝对引用，即与寄存器无关的内存地址必须重定位以反映它的真实地址。

链接器产生一个可执行文件，它可以在一台计算机上运行。通常，这个文件与目标文件具有相同的格式，但是它不包含未解决的引用。具有部分链接的文件是可能的，如库程序，在目标文件中仍含有未解决的地址。

5.2 在 Ubuntu 下链接的命令

ld -plugin /usr/lib/gcc/x86_64-linux-gnu/4.8/collect2 --sysroot=/ --build-id --eh-frame-hdr -m elf_x86_64 --hash-style=gnu --as-needed -dynamic-linker /lib64/ld-linux-x86-64.so.2 -z relro -o hello /usr/lib/gcc/x86_64-linux-gnu/4.8/../../../x86_64-linux-gnu/crt1.o /usr/lib/gcc/x86_64-linux-gnu/4.8/../../../x86_64-linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-gnu/4.8/crtbegin.o -L/usr/lib/gcc/x86_64-linux-gnu/4.8 -L/usr/lib/gcc/x86_64-linux-gnu/4.8/../../../x86_64-linux-gnu -L/usr/lib/gcc/x86_64-linux-gnu/4.8/../../../../lib -L/lib/x86_64-linux-gnu -L/lib/../lib -L/usr/lib/x86_64-linux-gnu -L/usr/lib/../lib -L/usr/lib/gcc/x86_64-linux-gnu/4.8/../../.. hello.o -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed /usr/lib/gcc/x86_64-linux-gnu/4.8/crtend.o /usr/lib/gcc/x86_64-linux-gnu/4.8/../../../x86_64-linux-gnu/crtn.o

ld: /usr/lib/gcc/x86_64-linux-gnu/4.8/collect2: error loading plugin: /usr/lib/gcc/x86_64-linux-gnu/4.8/collect2: cannot dynamically load executable

5-1 链接的指令

5.3 可执行目标文件 hello 的格式

分析 hello 的 ELF 格式，用 readelf 等列出其各段的基本信息，包括各段的起始地址，大小等信息。

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R E    8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000904 0x0000000000000904  R E    200000
  LOAD           0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
                 0x0000000000000254 0x0000000000000258  RW     200000
  DYNAMIC        0x0000000000000e28 0x0000000000600e28 0x0000000000600e28
                 0x00000000000001d0 0x00000000000001d0  RW     8
  NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254
                 0x0000000000000044 0x0000000000000044  R      4
  GNU_EH_FRAME   0x00000000000007b4 0x00000000004007b4 0x00000000004007b4
                 0x000000000000003c 0x000000000000003c  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
                 0x00000000000001f0 0x00000000000001f0  R      1
```

5-2 段信息

Offset：目标文件中的偏　　vaddr/paddr：内存地址

Filesize：目标文件中的段大小　　memsiz：内存中的段大小

Flags：运行时访问权限　　align：对齐要求

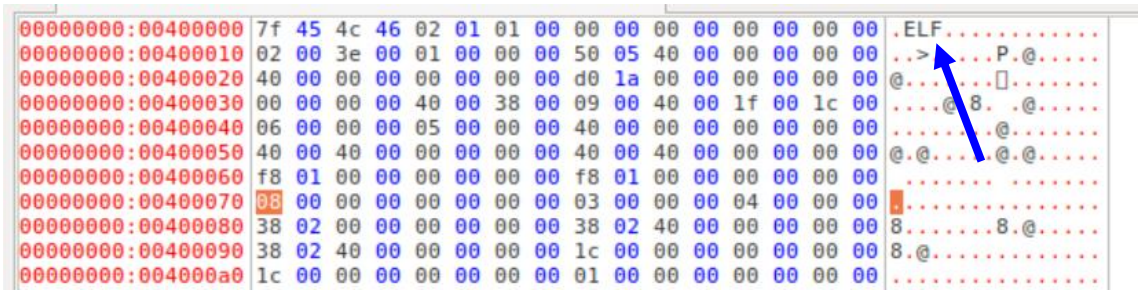
该段信息中有两个 load，其中第一个 load 是指只读信息段，第二个 load 值得是数据段。

表中信息说明第一个 load（代码段）有读和执行访问权限，开始于内存地址 0x400000 处，总共内存大小是 0x904 个字节。

第二个 load(数据段)有读和写权限。开始于内存地址 0x600e10 处，总共内存大小是 0x258 个字节。

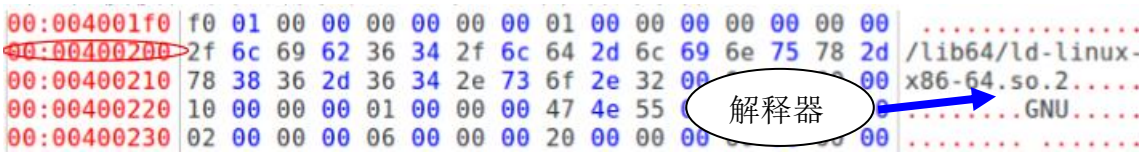
5.4 hello 的虚拟地址空间

使用 edb 加载 hello，查看本进程的虚拟地址空间各段信息，并与 5.3 对照分析说明。



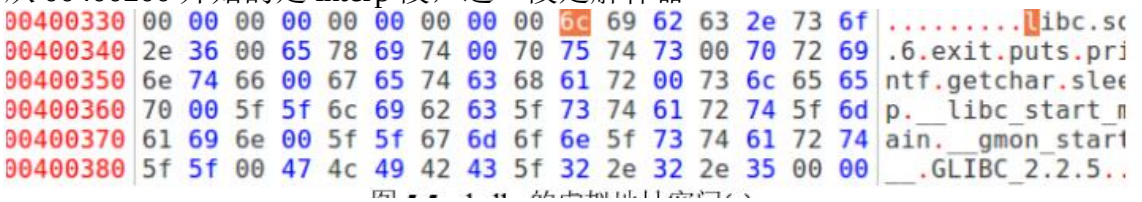
5-3 虚拟地址空间（1）

PHDR 程序头表是从 00400000 开始的可以看到一开始有 ELF 的文本信息



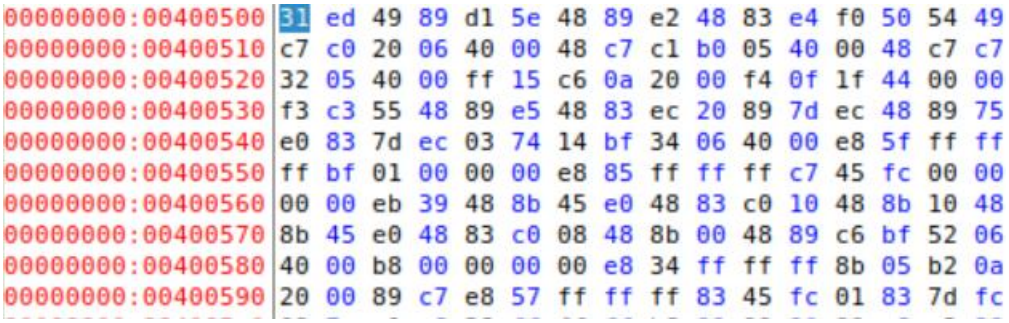
5-4 虚拟地址空间（2）

从 00400200 开始的是 interp 段，这一段是解释器



5-5 虚拟地址空间（3）

这一段右侧是库函数的信息，是节头部表中的.dynsym 节，保存的是与动态链接相关的符号。



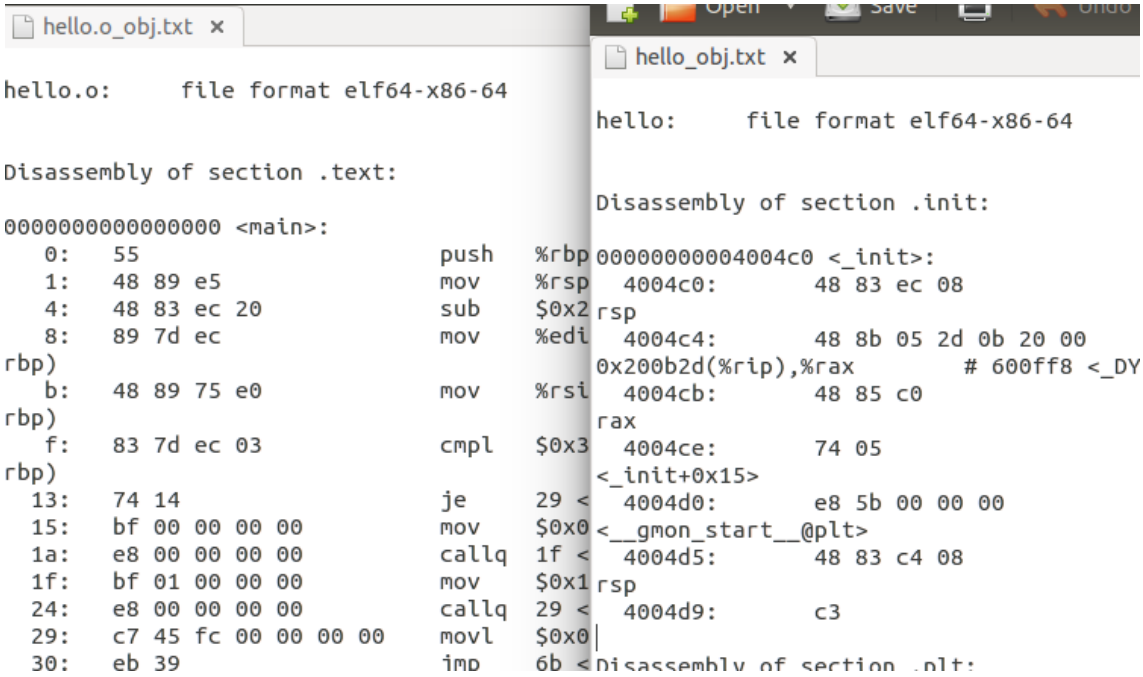
5-6 虚拟地址空间（4）

这一段是.text 节的内容。

5.5 链接的重定位过程分析

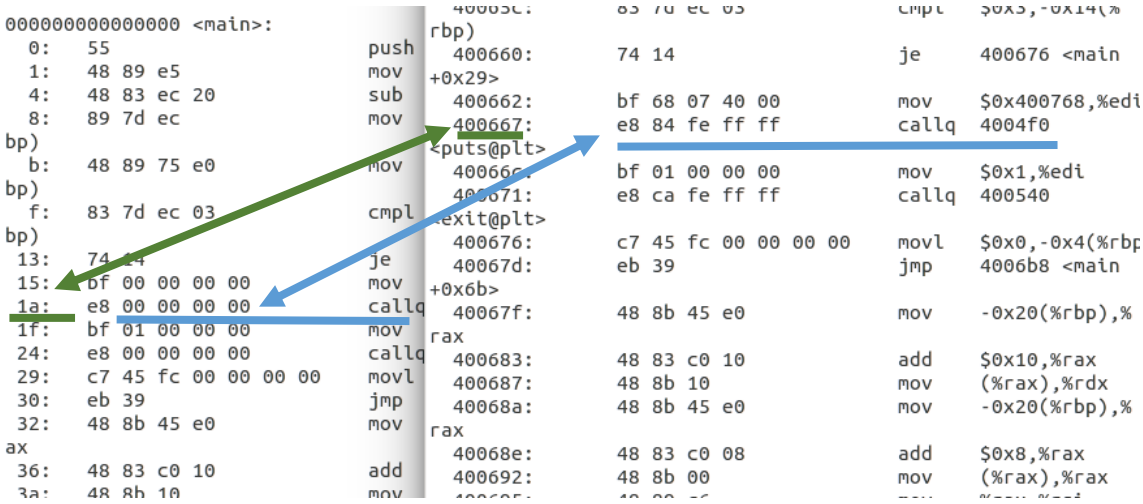
objdump -d -r hello 分析 hello 与 hello.o 的不同，说明链接的过程。

结合 hello.o 的重定位项目，分析 hello 中对其怎么重定位的。



5-7 链接前后的反汇编代码

与 hello.o 相比，hello 的反汇编程序多了很多标准库函数的汇编代码，在 hello.o 中只有 main 函数的汇编代码，在 main 函数中如果有调用标准库函数则会用地址 00 来站位，并补充相应信息指导链接器进行重定位。除此之外，在 hello 的反汇编文件中，call 调用库函数的地址不在是 00 的占位符，而是实际的内存地址。



5-8 汇编语句的对比

还有就是在每条指令左侧的地址是实际的内存地址，例如上图右侧的划出 callq 指令左侧地址为 0x400667，那么在加载 hello 到内存中后，这条指令的实际虚拟内存的地址就是 0x400667.而 hello.o 反汇编的文件左侧地址只显示与文件开头的差。

重定位过程：

在汇编器生成.o 文件的过程中，main 函数中调用了外部的库函数，汇编器不知道数据和代码最终存放在内存中的什么位置，所以当汇编器遇到对最终位置未知的目标引用时，就会生成一个重定位条目，告诉链接器在将目标文件合并成可执行文件时如何修改这个引用。链接器会将所有相同类型的节合并成为同一类型的新的聚合节。然后根据汇编器产生的重定位条目修改代码节和数据节中对每个符号的引用，使得他们指向正确的地址。

以 hello.o 为例，hello.o 并没有外部的变量引用，只引用了外部的函数，因此汇编器在生成可重定位文件时遇到未定义的函数时会用占位符 00 来占位，然后生成重定位条目，告诉链接器，这个目标需要重新定位。随后链接时，链接器将所有的函数放在同一个段中，此时就能确定每个函数在代码段中的位置了，链接器就通过重定位条目修改符号引用。

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。

ld-2.27.so! dl_start	0x7ffee5aca680
ld-2.27.so! dl_init	0x7f9f48629630
hello!_start	0x0x400500
ld-2.27.so!_libc_start_main	0x7f9f48249ab0
libc-2.27.so! cxa_atexit	0x7f4523fd6af7
libc-2.27.so! lll_look_wait_private	0x7f4523ff8471
libc-2.27.so!_new_exitfn	0x7f87ff534220
hello!_libc_csu_init	0x7f87ff512b26
libc-2.27.so!_setjmp	0x7f87ff512b4a
libc-2.27.so!_sigsetjmp	0x7f87ff52fc12
libc-2.27.so!__sigjmp_save	0x7f87ff52fbc3
hello_main	0x400532
hello!puts@plt	0x4004b0
hello!exit@plt	0x4004e0
hello!printf@plt	0x400587
hello!sleep@plt	0x400594
hello!getchar@plt	0x4005a3
dl_runtime_resolve	0x7f169ad84750
libc-2.27.so!exit	0x7fce 8c889128

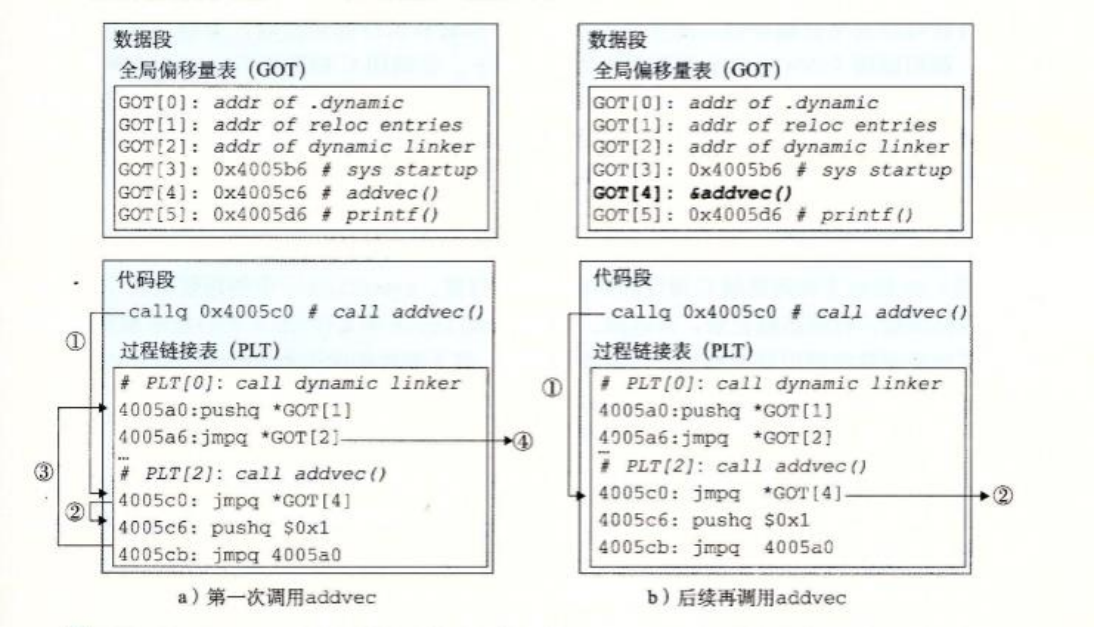
5.7 Hello 的动态链接分析

分析 hello 程序的动态链接项目，通过 edb 调试，分析在 dl_init 前后，这些项目的内容变化。要截图标识说明。

假设程序调用一个由共享库定义的函数。编译器没有办法预测这个函数的运行时地址,因为定义它的共享模块在运行时可以加载到任意位置。正常的方法是为该引用生成一条重定位记录,然后动态链接器在程序加载的时候再解析它。不过,这种方法并不是 PC,因为它需要链接器修改调用模块的代码段,GNU 编译系统使用了一种很有趣的技术来解决这个问题,称为延迟绑定,将过程地址的绑定推迟到第一次调用该过程时。

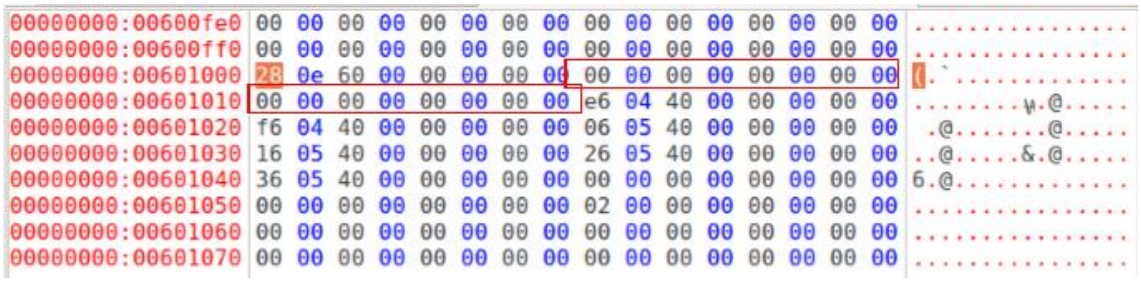
使用延迟绑定的动机是对于一个像 1ibc.so 这样的共享库输出的成百上千个函数中个典型的应用程序只会使用其中很少的一部分。把函数地址的解析推迟到它实际被调用的地方,能避免动态链接器在加载时进行成百上千个其实并不需要的重定位。第一次调用过程的运行时开销很大,但是其后的每次调用都只会花费一条指令和一个间接的内存引用

延迟绑定是通过两个数据结构之间简洁但又有些复杂的交互来实现的,这两个数据结构是:GOT 和过程链接表。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,而 PLT 是代码段的一部分。

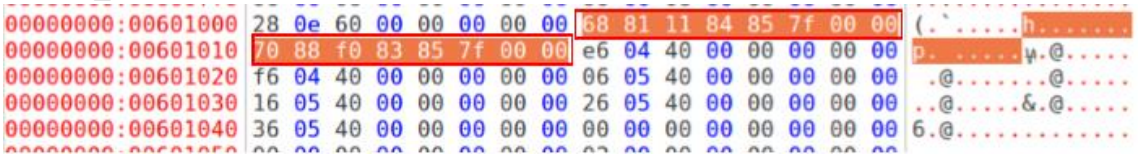


5-9addvec 修改示意图

在 dl_init 调用之前，都不会直接调用 addvec，而是通过 GOT[4]进行间接跳转。因为每个 GOT 条目都指向它对应的第二条指令，这个间接跳转只是简单地把控制传送回 PLT[2]中的下一条指令，在将 ID 压栈之后挑战到 PLT[0]，再压入一个参数后，跳转进动态链接器中动态链接器根据栈条目重写 addvec 的运行时地址。



5-10 dl_init 前的 PLT



5-11 dl_init 后的 PLT

在修改完 PLT 后，有两个地址被加入到 PLT 中了，分别是 0x7f8584118168 和 0x7f8583f08870。这两个地址就是 GOT[1]以及 GOT[2]。当修改完成 PLT 后，以后调用 addvec 时，就可以直接跳转到 addvec 了。

5.8 本章小结

本章介绍了链接的命令、概念以及段头部表的信息。通过链接，在可重定位文件中的外部函数以及外部全局变量都能得到重新定位，在程序中对这些符号的引用也能够正确解析。在链接完成之后，hello 正式成为了一个可以运行的目标文件。

（第 5 章 1 分）

第 6 章 hello 进程管理

6.1 进程的概念与作用

概念：

进程就是一个执行中的程序的实例。系统中的每个进程都运行在某个进程的上下文中。上下文是由程序正确运行所需的状态组成的。这个状态包括存放在内存中的代码和数据。它的栈、通用目的寄存器的内容、程序寄存器、环境变量以及打开文件描述符的集合。

进程是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。进程（Process）是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。在早期面向进程设计的计算机结构中，进程是程序的基本执行实体；在当代面向线程设计的计算机结构中，进程是线程的容器。程序是指令、数据及其组织形式的描述，进程是程序的实体。

作用：

在现代系统上运行一个程序时，我们会得到一个假象，就好像我们的程序是系统中当前运行的唯一的程序一样。我们的程序好像是独占地使用处理器和内存。处理器就好像是无间断的一条接一条地执行我们程序中的指令。最后，我们程序中的代码和数据好像是系统内存中唯一的对象，这些假象都是通过进程的概念提供给我们的。

6.2 简述壳 Shell-bash 的作用与处理流程

作用：

在计算中，命令行解释器或命令语言解释器是用于读取用户输入的文本行的某类程序的总称，从而实现命令行界面。Shell 俗称壳（用来区别于核），是指“为使用者提供操作界面”的软件（命令解析器）它接收用户命令，然后调用相应的应用程序。它交互式解释和执行用户输入的命令或者自动地解释和执行预先设定好的一连串的命令。

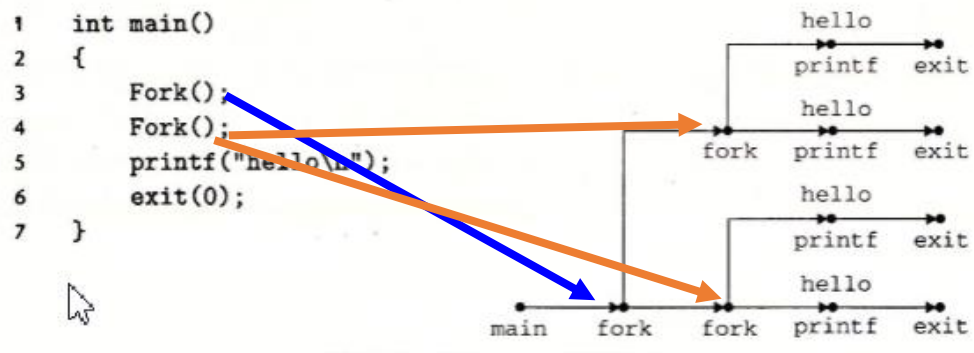
1. 执行执行预先置入的命令，比如说 quit、bg、cd 等，通过这些预先置入的命令可以完成一系列的操作，比如说目录的切换，退出当前运行的程序、或者打印相应的信息。
2. 执行可执行文件。当输入的命令不是内置的命令时，shell 会将输入的命令解析

为可执行文件，并且根据输入的内容构造参数和环境变量，执行可执行文件。
处理流程：

首先根据输入的内容来判断是否是内置的命令，如果是内置的命令再根据不同 的参数执行不同的操作，不是内置的命令，那么就将该命令当做可执行文件执行， 首先在当前目录下查找是否有这样的可执行文件，如果没有则输出错误。如果存 在这样的可执行文件，那么根据输入的命令参数，以不同的 argv[] 和环境变量执 行该可执行文件，并且更具命令行结尾是否有'&'符号来判断是否在前台执行，如 果在前台执行，那么就要等待该程序执行完成，才能执行下一条命令。如果是后 台执行，那么就可以在后台运行，在运行的同时可以执行其他的命令。

6.3 Hello 的 fork 进程创建过程

在 shell 中键入 ./hello 的命令后，shell 会调用 fork 函数，创建另外一个进程在执行 fork 函数后，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。在根据键入的命令不同，hello 程序可能在后台执行，也可以在前台执行。在创建一个另外一个相同的 shell 进程之后，子进程会调用 execve 函数，并将环境变量当做参数传递，加载可执行文件 hello。



6-1fork 创建进程

6.4 Hello 的 execve 过程

execve 函数加载并运行可执行目标文件 filename，且带参数列表 argv 和环境变量列表 envp。只有当出现错误时，例如找不到 filename，execve 才会返回到调用程序。

execve 函数更具参数和环境变量将 hello 文件复制到内存当中，在程序头部表的引导下，加载器将可执行文件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。系统启动函数初始化执行环境，调用用户层的 main 函数，处理 main 函数的返回值，并在需要的时候讲控制返回给内核。

6.5 Hello 的进程执行

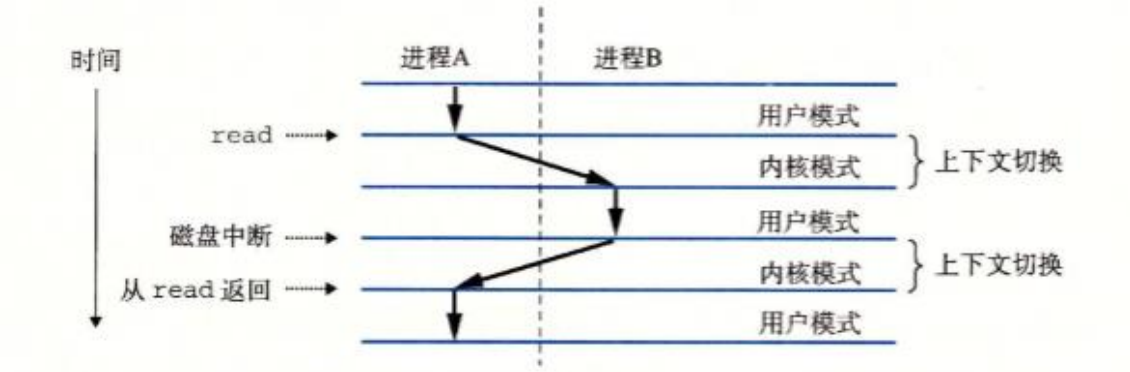
结合进程上下文信息、进程时间片，阐述进程调度的过程，用户态与核心态转换等等。

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强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。

在内核执行的某些时刻，内核可以决定抢占当前进程，并重新开始一个先前被强占了的进程。当内核选择一个新的进程运行时，我们说内核调度了这个进程。在内核调度了一个先前被强占了的进程后，它就强占当前进程，并用一种称为上下文切换的机制来将控制转移到新的进程。

6-2 并发控制流

多个流并发的执行的一般现象被称为并发。一个进程和其他进程轮流运行的概念称为多任务。一个进程执行它的控制流的一部分的每一时间段叫做时间片。



6-3 并发中内核与用户模式的切换

如上图所示，进程 A 初始运行在用户模式中，直到它通过执行系统调用 read 陷入到内核。控制流进入到内核中，在内核处理完从磁盘到内存的数据传输后，内核将控制流交还给用户程序，于是控制流重新回到了用户。

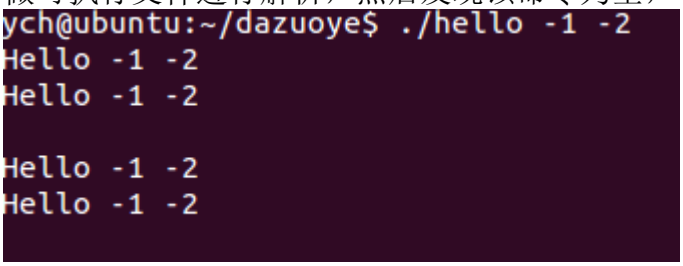
6.6 hello 的异常与信号处理

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结截屏，说明异常与信号的处理。

回车：

键入回车键，shell 首先会判断是否为内置命令，发现不是内置命令。然后将其当做可执行文件进行解析，然后发现该命令为空，于是不执行任何操作。



6-4 执行 hello 文件

Crtl-Z：

键入 Crtl-Z 会导致内核发送一个 SIGTSTP 信号到前台进程组中的每个进程。在默认情况下，将会挂起前台进程。

6-5hello 执行中键入 Crtl-Z

Crtl-Z ps：

在键入 Crtl-Z 后，hello 进程被挂起，键入 ps，shell 会判断这是一个内置命令，ps 的作用是将这次登入的 pid 与相关的信息列出来。于是，shell 会将自己的 pid 一个 hello 进程的 pid 和为了执行 ps 命令创建的子进程的 pid 全部列出来。

```
ych@ubuntu:~/dazuoye$ ./hello -1 -2
Hello -1 -2
Hello -1 -2
^Z
[1]+  Stopped                 ./hello -1 -2
ych@ubuntu:~/dazuoye$ ps
   PID TTY          TIME CMD
  7051 pts/0    00:00:00 bash
  7069 pts/0    00:00:00 hello
  7070 pts/0    00:00:00 ps
ych@ubuntu:~/dazuoye$
```

6-6hello 执行中键入 Crtl-Z 和 ps

Crtl-Z　　jobs：

在键入 Crtl-Z 后，hello 进程被挂起，键入 jobs，shell 判断出 jobs 是内置命令，而 jobs 的作用是查看 shell 当前正在处理的作业。于是 shell 会输出 hello 的信息，hello 当前的状态是被挂起，于是输出 stopped。

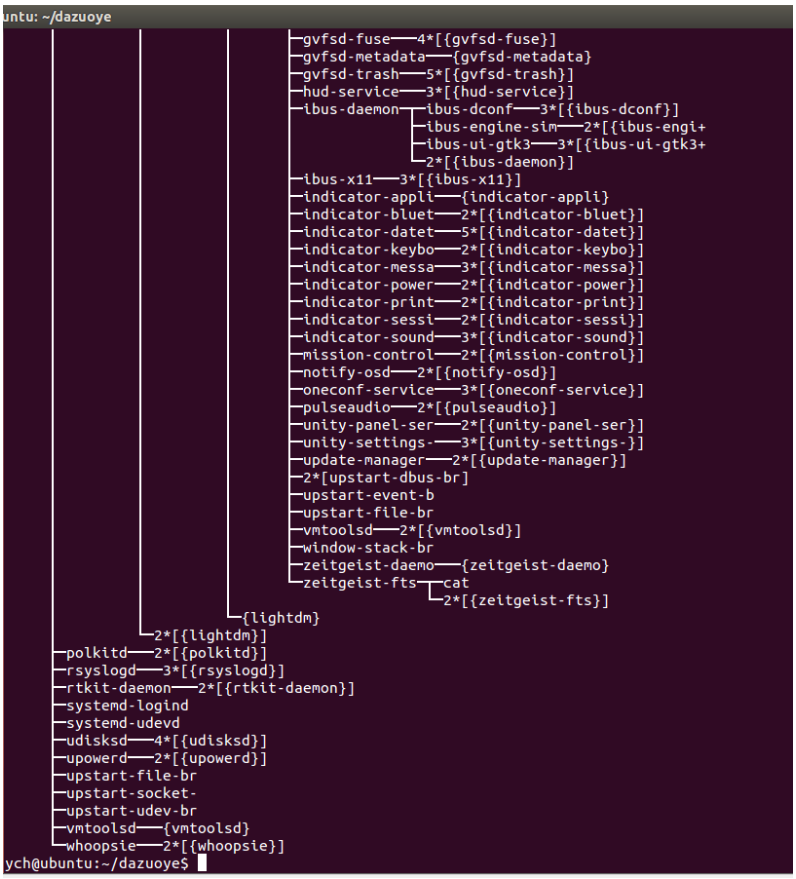
```
ych@ubuntu:~/dazuoye$ jobs
[1]+  Stopped                 ./hello -1 -2
ych@ubuntu:~/dazuoye$
```

6-7hello 执行中键入 Crtl-Z 和 jobs

Crtl-Z　　pstree：

键入 pstree 命令会以树状图的形式显示进程间的关系。

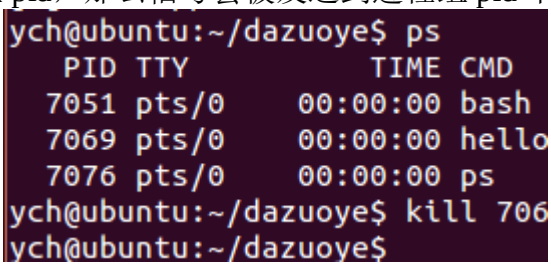
```
ych@ubuntu:~/dazuoye$ pstree
init─┬─ModemManager───2*[{ModemManager}]
     ├─NetworkManager─┬─dhclient
     │                ├─dnsmasq
     │                └─3*[{NetworkManager}]
     ├─accounts-daemon───2*[{accounts-daemon}]
     ├─acpid
     ├─avahi-daemon───avahi-daemon
     ├─bluetoothd
     ├─colord───2*[{colord}]
     ├─cron
     ├─cups-browsed
     ├─cupsd───dbus
     ├─dbus-daemon
     ├─6*[getty]
     ├─gnome-keyring-d───5*[{gnome-keyring-d}]
     ├─kerneloops
     ├─lightdm─┬─Xorg
     │         └─lightdm─┬─init─┬─at-spi-bus-laun─┬─dbus-daemon
     │                   │      │                 └─3*[{at-spi-bus-laun}]
     │                   │      ├─at-spi2-registr───{at-spi2-registr}
     │                   │      ├─bamfdaemon───3*[{bamfdaemon}]
     │                   │      ├─dbus-daemon
     │                   │      ├─dconf-service───2*[{dconf-service}]
     │                   │      ├─evolution-calen───4*[{evolution-calen}]
     │                   │      ├─evolution-sourc───2*[{evolution-sourc}]
     │                   │      ├─gconfd-2
     │                   │      ├─gnome-session─┬─compiz───4*[{compiz}]
     │                   │      │               ├─deja-dup-monito───2*[{deja-du+
     │                   │      │               ├─nautilus───3*[{nautilus}]
     │                   │      │               ├─nm-applet───2*[{nm-applet}]
     │                   │      │               ├─polkit-gnome-au───2*[{polkit-+
     │                   │      │               ├─telepathy-indic───2*[{telepat+
     │                   │      │               ├─unity-fallback-───2*[{unity-f+
     │                   │      │               ├─update-notifier───3*[{update-+
     │                   │      │               ├─zeitgeist-datah───3*[{zeitgei+
     │                   │      │               └─3*[{gnome-session}]
     │                   │      ├─gnome-terminal─┬─bash─┬─hello
     │                   │      │                │      └─pstree
     │                   │      │                ├─gnome-pty-helpe
     │                   │      │                └─3*[{gnome-terminal}]
     │                   │      ├─gvfs-afc-volume───2*[{gvfs-afc-volume}]
     │                   │      ├─gvfs-gphoto2-vo───{gvfs-gphoto2-vo}
     │                   │      ├─gvfs-mtp-volume───{gvfs-mtp-volume}
     │                   │      ├─gvfs-udisks2-vo───2*[{gvfs-udisks2-vo}]
```



6-8hello 执行中键入 Crtl-Z 和 pstree

Crtl-Z　　kill：

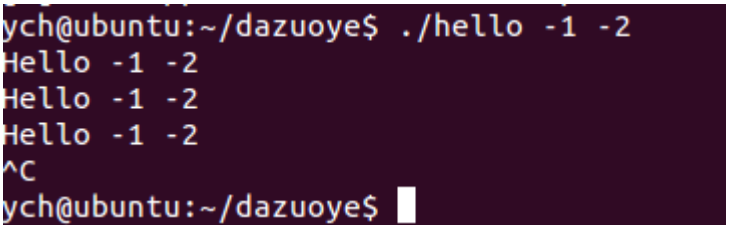
键入 kill 加上一个 pid 号，如果 pid 为负，该命令会发送一个 SIGKILL 信号给进程 pid，那么信号会被发送到进程组 pid 中的每个进程。



6-9hello 执行中键入 Crtl-Z 和 kill

Ctrl-C：

键入 Ctrl-C 命令后，内核会发送一个 SIGINT 信号到前台进程组中的每个进程，默认情况下结果是终止前台作业。



6-10 键入 Crtl-C

6.7 本章小结

Hello 从被加载到内存中到被进程被终止，到最后被回收，它的生命不可谓不坎坷。Hello 首先通过加载器被加载到内存当中，通过一系列的步骤确保能够顺利执行 hello 文件。进程的概念为 hello 创造了一种假象，hello 天真的以为 CPU 和内存是由自己独占。内核通过不断的在不同的进程之间调度，通过上下文机制保证进程的信息不会被破坏，进而确保资源的合理分配。在 hello 执行的过程中还有可能遭到各种各样的意外。CRTL+C 直接杀死 hello 而不给它一点机会，CRTL+Z 则会将 hello 挂起，直到有人愿意解救它。除此之外，还有各种各样的风险“信号”会导致 hello 经历这样那样的磨难。最后，在 hello 结束了它的一生后，还会遭人唾弃，因为它还在占用宝贵的内存资源，甚至还给它取了一个悲伤的名字“僵尸进程”，直到僵尸进程被回收，hello 的一生才算完整的画上了句号。

（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

结合 hello 说明逻辑地址、线性地址、虚拟地址、物理地址的概念。

逻辑地址：

CPU 所生成的地址。逻辑地址是内部和编程使用的、并不唯一。例如，在进行 C 语言指针编程中，可以读取指针变量本身值(&操作)，实际上这个值就是逻辑地址，它是相对于你当前进程数据段的地址（偏移地址）。

线性地址：

线性地址是逻辑地址到物理地址变换之间的中间层。在分段部件中逻辑地址是段中的偏移地址，然后加上基地址就是线性地址。

Hello 程序的各类信息存放在不同的段中，基地址加上逻辑地址就是线性地址。

虚拟地址：

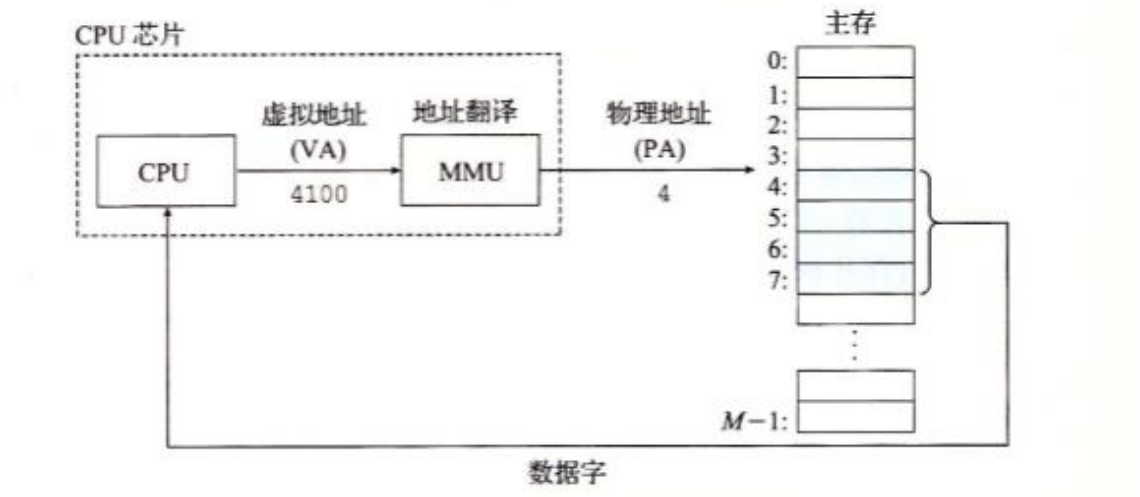
虚拟地址是相对与虚拟内存而言的。虚拟地址与虚拟内存上的每个字节一一对应。虚拟内存为每个进程提供了一个大的、一致的和私有的地址空间。通过一个很清晰的机制，虚拟内存提供了三个重要的能力。

1）它将主存看成是一个存储在磁盘上的地址空间的高速缓存，在主存中只保护活动区域，并根据需要在磁盘和主存之间来回传送数据，通过这种方式，高效的使用了主存。

2）它为每个进程提供了一致的地址空间，从而简化了内存管理。

3）它保护了每个进程的地址空间不被其他进程破坏。

Hello 程序每次被加载到内存总是被加载到相同的地址空间，从 0x40000000 开始向上增长，而所有的进程都有看似相同的内存空间，但是实际上是不同的。CPU 和操作系统协同将虚拟地址转化为物理地址，于是就可以将信息写入或读取内存。虽然每个进程的虚拟内存看似很大，但实际上占用的物理内存空间可能只有一小部分。



7-1 虚拟地址到物理地址的翻译

物理地址：

在存储器里以字节为单位存储信息，为正确地存放或取得信息，每一个字节单元给以一个唯一的存储器地址，称为物理地址。

地址从 0 开始编号，顺序地每次加 1，因此存储器的物理地址空间是呈线性增长的。它是用二进制数来表示的，是无符号整数，书写格式为十六进制数。它是出现在 CPU 外部地址总线上的寻址物理内存的地址信号，是地址变换的最终结果。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。

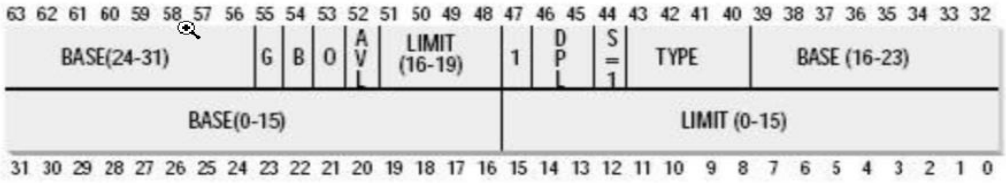
Hello 程序的二进制代码存放在内存中的一小块上，物理地址与内存上的每个字节一一对应，通过物理地址可以寻找到内存上存储的信息。而物理地址是通过虚拟地址转化而来的。

7.2 Intel 逻辑地址到线性地址的变换-段式管理

段式内存管理方式就是直接将逻辑地址转换成物理地址，也就是 CPU 不支持分页机制。其地址的基本组成方式是段号+段内偏移地址。

在介绍段式内存管理方式之前首先介绍逻辑空间。逻辑空间分为若干个段，其中每一个段都定义了一组具有完整意义的信息，逻辑地址对应于逻辑空间，如（主程序的 main()）函数。

数据段描述符



7-2 数据段描述符

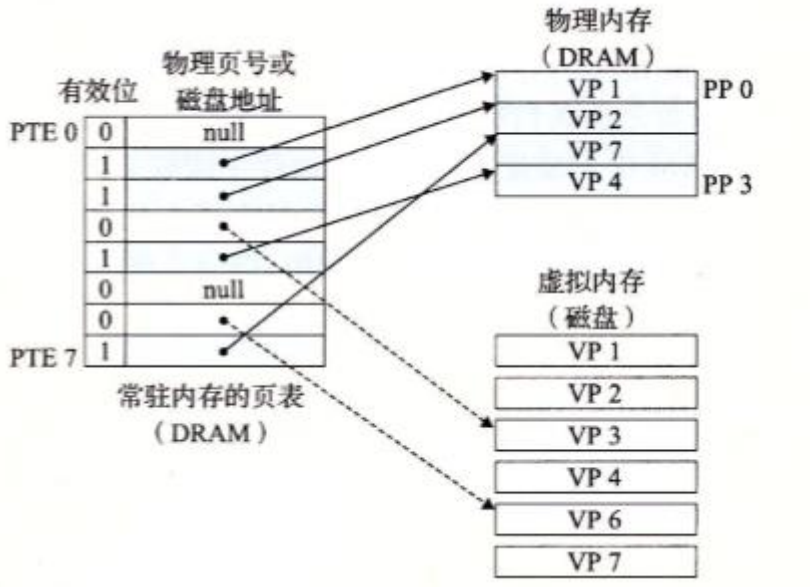
段是对程序逻辑意义上的一种划分，一组完整逻辑意义的程序被划分成一段，所以段的长度是不确定的。

段式内存管理方式经过段表映射到内存空间。先说明一下段表的概念，可以将段表抽象成一个大的数组集合，数组中的元素是什么呢？就是“段描述符”----用于描述一个段的详细信息的结构。段描述符一般是由 8 个字节组成，也就是 64 位。操作系统使用的不同的段描述符如图所示。

7.3 Hello 的线性地址到物理地址的变换-页式管理

磁盘上的数据被分割成块，这些块作为磁盘和主存之间的传输单元。VM 系统通过将虚拟内存分割为称为虚拟页的大小固定的块来处理这个问题。

虚拟内存系统通过软硬件结合的方式判断一个虚拟页是否缓存在 DRAM 中。这些软硬件包括操作系统、MMU 中的地址翻译硬件和一个存放在物理内存中的叫做页表的数据结构，页表将虚拟页映射到物理页。每次地址翻译硬件将一个虚拟地址转换为物理地址时，都会读取页表。操作系统负责维护页表的内容，以及在磁盘和 DRAM 之间来回传送页。

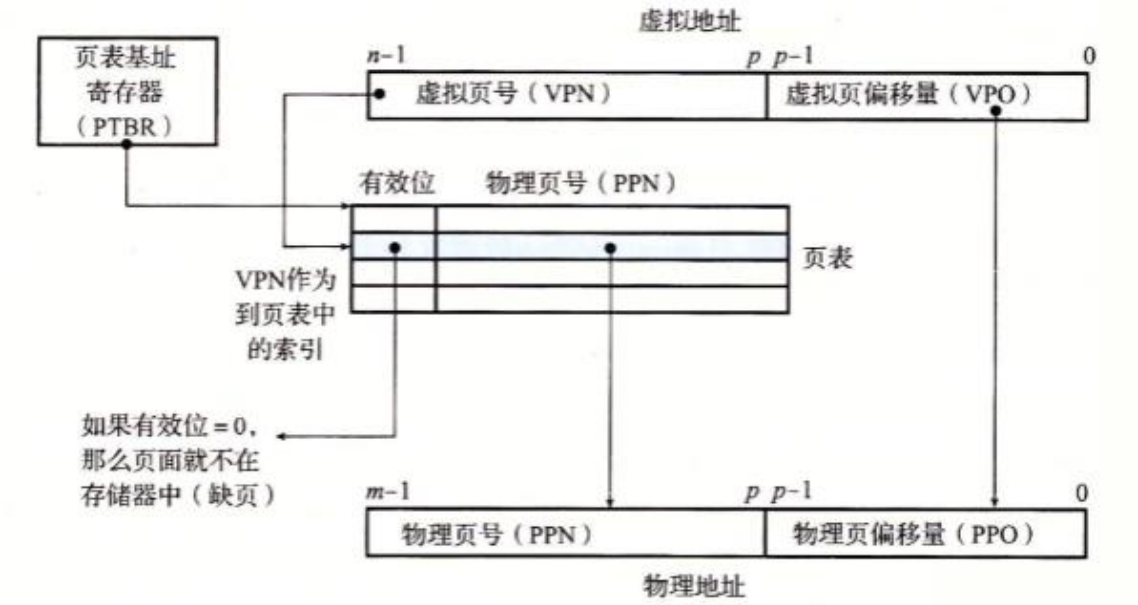


7-3 页表的映射

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

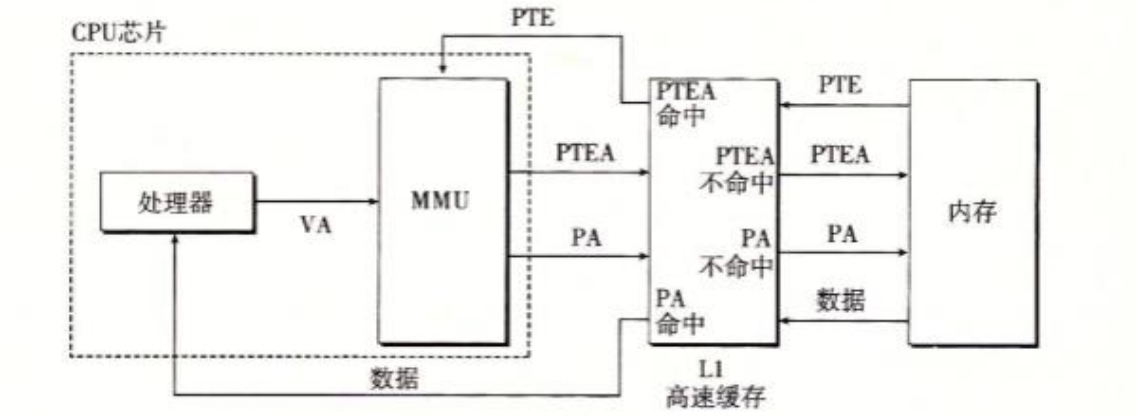
地址翻译是一个 N 元素的虚拟空间地址中的元素和一个 M 元素的物理地址空间中元素的映射。

页表基址寄存器指向当前页表。N 位的虚拟地址包含两个部分：一个 p 位的虚拟页面偏移和一个（n-p）位的虚拟页号。MMU 利用 VPN 选择适当的 PTE。



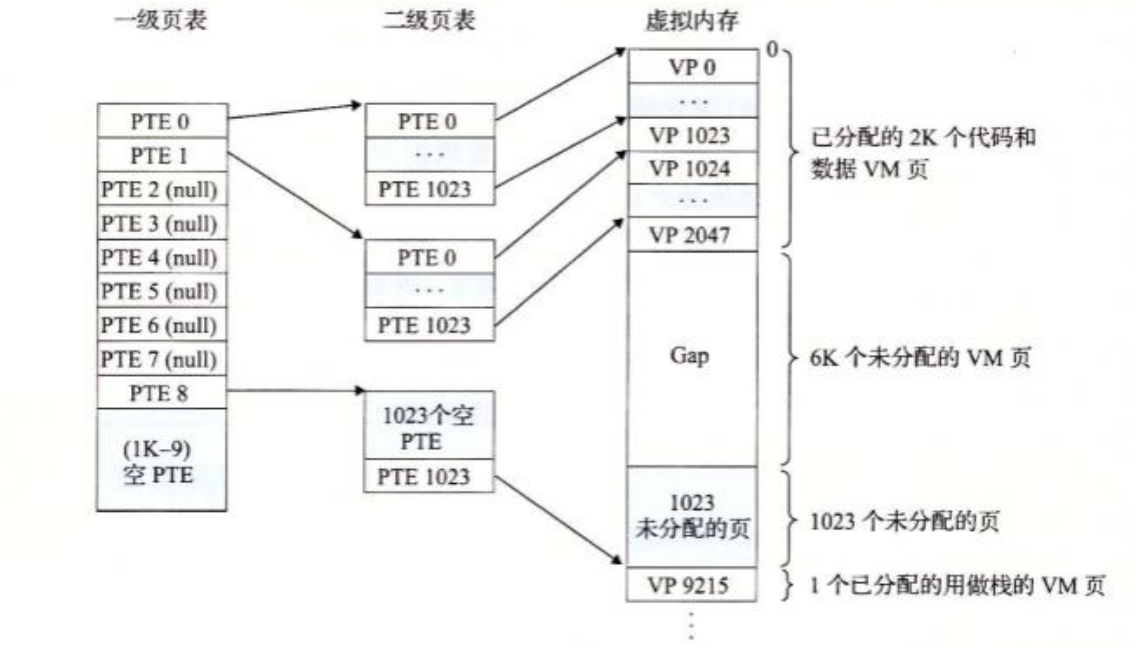
7-4 虚拟地址像物理地址的转换

在 MMU 中包括了一个关于 PTE 的小的缓存，称为翻译后备缓冲器。TLB 是一个小的、虚拟寻址的缓存，其中每一行都保存着一个由单个 PTE 组成的块。含有 TLB 的地址翻译过程如下图所示。



7-5 带 cache 的地址翻译过程

多级页表中，虚拟地址被划分为 4 个 VPN 和 1 个 VPO。每个 VPNi 都是一个到第 i 级页表的缩影，其中 i<=4.第 j 级页表中的每个 PTE 包含某个物理页面 dePPN，或者一个磁盘块的地址。为了过早物理地址，在能够确定 PPN 之前，MMU 必须访问 4 个 PTE。



7-6 多级页表的结构

7.5 三级 Cache 支持下的物理内存访问

通过硬件和操作系统的结合，我们已经得到了与虚拟地址相对应的物理地址，然后根据物理地址先从 L1cache 查找该字节是否缓存在 L1 中，更具组号和标志位，如果命中，那么直接将字节信息从 L1 中取走，如果不命中那么就从下一级 cache 中寻找，如果命中则直接取走，不命中则从 L3 中查找，依次类推。

7.6 hello 进程 fork 时的内存映射

既然我们理解了虚拟内存和内存映射,那么我们可以清晰地知道 fork 函数是如何创建一个带有自己独立虚拟地址空间的新进程的。

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,并分配给它一个唯一的 PID。为了给这个新进程创建虚拟内存,它创建了当前进程的 mm struct、区域结构和页表的原样副本。它将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 fork 在新进程中返回时,新进程现在的虚拟内存刚好和调用 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,也就为每个进程保持了私有地址空间的抽象概念。

7.7 hello 进程 execve 时的内存映射

假设运行在当前进程中的程序执行了如下的 execve 调用:

```
execve("a out", NULL, NULL);
```

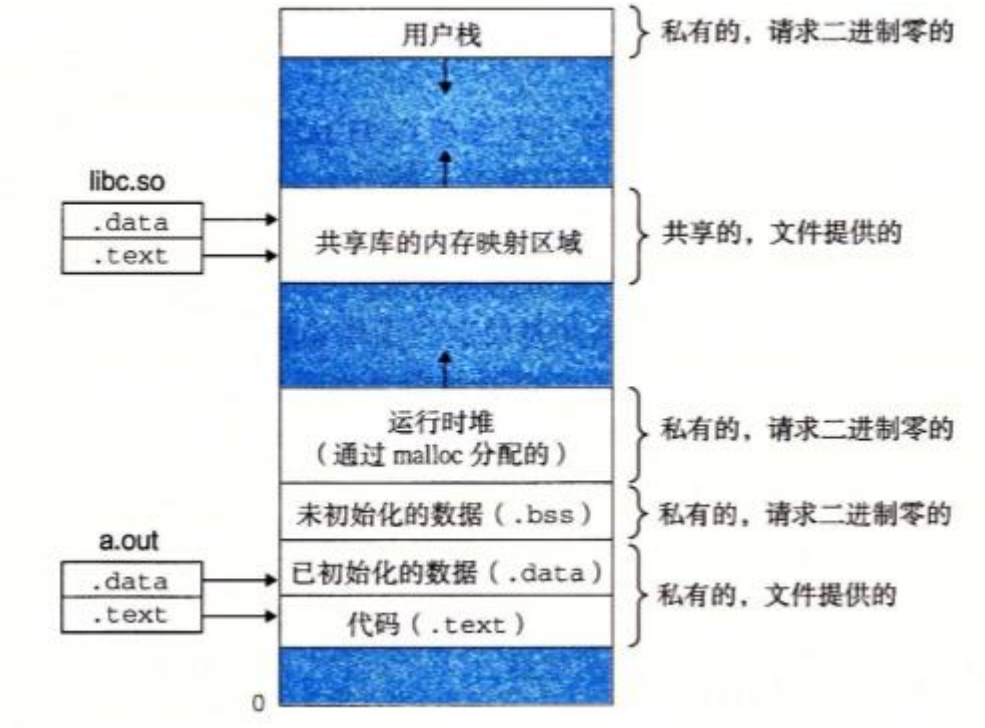
execve 函数在当前进程中加载并运行包含在可执行目标文件 a.out 中的程序,用 a.out 程序有效地替代了当前程序。加载并运行 aout 需要以下几个步骤

• 删除已存在的用户区域。删除当前进程虚拟地址的用户部分中的已存在的区域结构。

• 映射私有区域。为新程序的代码、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 a.out 文件中的.text 和.data 区。bss 区域是请求二进制零的,映射到匿名文件,其大小包含在 a.out 中。栈和堆区域也是请求二进制零的,初始长度为零。图 9-31 概括了私有区域的不同映射。

• 映射共享区域。如果 a.out 程序与共享对象(或目标)链接,比如标准 C 库 1ibc.so,那么这些对象都是动态链接到这个程序的,然后再映射到用户虚拟地址空间中的共享区域内。

•设置程序计数器(PC)。 execve 做的最后一件事情就是设置当前进程上下文中的程序计数器,使之指向代码区域的入口点。下一次调度这个进程时,它将从这个人口点开始执行。 Linux 将根据需要换入代码和数据页面。

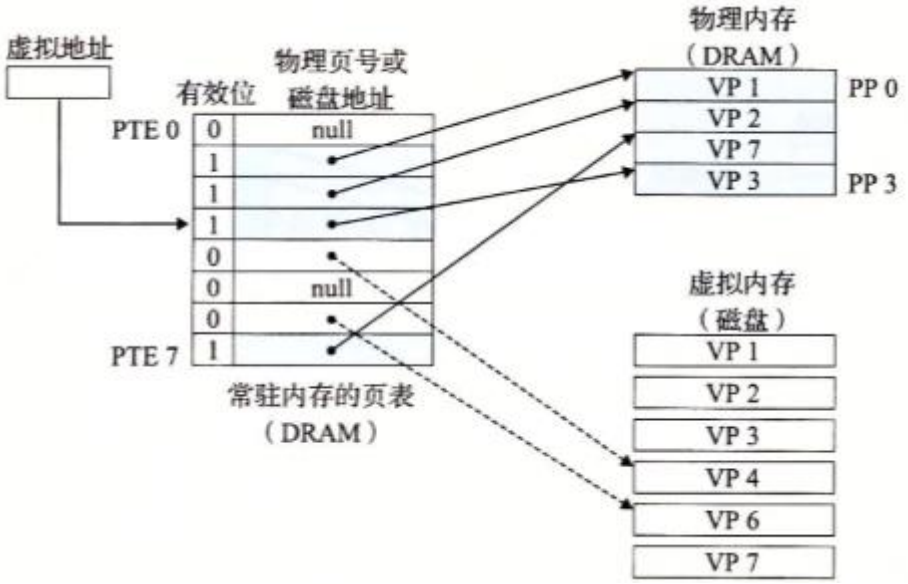


7-7 程序运行时内存的分配情况

7.8 缺页故障与缺页中断处理

如图所示为缺页之前我们的示例页表的状态。CPU 引用了 VP3 中的一个

字,VP3 并未缓存在 DRAM 中。地址翻译硬件从内存中读取 PTE3,从有效位推断出 VP3 未被缓存,并且触发一个缺页异常。缺页异常调用内核中的缺页异常处理程序,该程序会选择一个牺牲页,在此例中就是存放在 PP3 中的 VP4。如果 VP4 已经被修改了,那么内核就会将它复制回磁盘。无论哪种情况,内核都会修改 VP4 的页表条目,反映出 VP4 不再缓存在主存中这一事实。



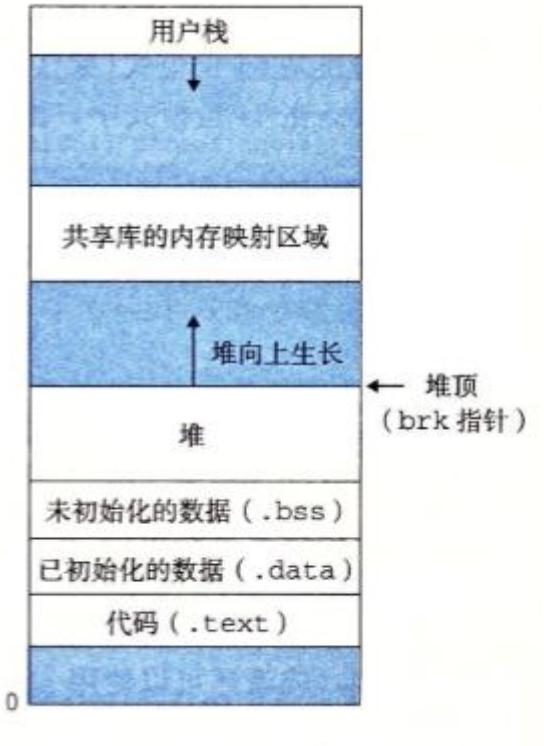
7-8 页表的映射

接下来,内核从磁盘复制 VP3 到内存中的 PP3,更新 PTE3,随后返回。当异常处理程序返回时,它会重新启动导致缺页的指令,该指令会把导致缺页的虚拟地址重发送到地址翻译硬件。但是现在,VP3 已经缓存在主存中了,那么页命中也能由地址翻译硬件正常处理了。图 9-7 展示了在缺页之后我们的示例页表的状态

7.9 动态存储分配管理

Printf 会调用 malloc，请简述动态内存管理的基本方法与策略。

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。



7-9 堆

分配器将堆视为一组不同大小的块的集合来维护。每个块就是一个连续的虚拟内存片,要么是已分配的,要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可用来分配。空闲块保持空闲,直到它显式地被应用所分配。一个已分配的块保持已分配状态,直到它被释放,这种释放要么是应用程序显式执行的,要么是内存分配器自身隐式执行的。分配器有两种基本风格。两种风格都要求应用显式地分配块。它们的不同之处在于由哪个实体来负责释放已分配的块。

•显式分配器(explicit allocator),要求应用显式地释放任何已分配的块。例如,C标准库提供一种叫做 ma1loc 程序包的显式分配器。C 程序通过调用 ma11oc 函数来分配一个块,并通过调用 free 函数来释放一个块。C++中的 new 和 delete 操作符与 C 中的 ma110c 和 free 相当。

• 隐式分配器(implicit allocator),另一方面,要求分配器检测一个已分配块何时不再被程序所使用,那么就释放这个块。隐式分配器也叫做垃圾收集器(garbage collec-tor),而自动释放未使用的已分配的块的过程叫做垃圾收集(garbage collection)例如,诸如 Lisp、ML 以及 Java 之类的高级语言就依赖垃圾收集来释放已分配的块。

7.10 本章小结

Hello 程序在加载到内存后是被虚拟内存系统所管理，在 hello 刚刚被加载时，

只是在新的进程中分配了虚拟地址，将各个段都加在到虚拟内存中，但这些虚拟地址都是出于未缓存的状态，在第一次需要用到这些数据时，系统通过缺页处理将信息加载到物理内存中。此外在 CPU 中还有一个 TLB 高速缓存，用于更快的进行地址翻译过程。将虚拟地址翻译为物理地址的过程首先是在 TLB 中查找是否有该页表条目，如果命中则直接得到物理地址，如果没有命中，则到高速缓存中查找 PTE 页表，得到物理地址后到 cache 和主存中获取字节的信息。

（第 7 章 2 分）

第 8 章 hello 的 I0 管理

8.1 Linux 的 I0 设备管理方法

设备的模型化：文件

设备管理：unix io 接口

所有的 I/O 设备(例如网络、磁盘和终端)都被模型化为文件,而所有的输入和输出都被当作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 Lnux 内核引出一个简单、低级的应用接口,称为 Unix I/O,这使得所有的输入和输出都能以一种统一且一致的方式来执行

8.2 简述 Unix I0 接口及其函数

• 打开文件。一个应用程序通过要求内核打开相应的文件,来宣告它想要访问 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。

• Linux shell t 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。

• 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

• 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 十 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”。

• 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。

函数：

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

返回：若成功则为新文件描述符，若出错为-1

进程通过调用 open 函数来打开一个已存在的文件或者创建一个新文件的。

open 函数将 filename 转换为一个文件描述符,并且返回描述符数字。返回的描述符总是在进程中当前没有打开的最小描述符。f1ags 参数指明了进程打算如何访问这个文件:

- O_RDONLY:只读
- O_WRONLY:只写。
- O_RDWR:可读可写。

int close(int fd);

返回：若成功则为 1，若出错为-1

进程通过调用 close 函数关闭一个已打开的文件。关闭一个已关闭的描述符会出错。

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

返回：若成功则为读的字节数，若 EOF 则为 0，若出错为-1

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

返回：若成功则为写的字节数，若出错则为-1

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

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

8.3 printf 的实现分析

Printf 的函数代码如下：

```
int printf(const char *fmt, ...){
    int i;
    char buf[256];

    va_list arg = (va_list)((char*)(&fmt) + 4);
    i = vsprintf(buf, fmt, arg);
    write(buf, i);

     return i;
}
```

其中“...”表示不定长参数。va_list 指的是 char*为字符指针。`(char*)(&fmt) + 4` 指的是不定长参数中的第一个参数，&fmt 代表第一个参数的位置，参数在栈中

传递。&fmt + 4 就代表第一个参数，之后依次加 4 就能获得之后的参数了。

第二个函数 valprintf 代码如下：

```
int vsprintf(char *buf, const char *fmt, va_list args) {
    char* p;
    char tmp[256];
    va_list p_next_arg = args;

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

        fmt++;

        switch (*fmt) {
            case 'x':
                itoa(tmp, *((int*)p_next_arg));
                strcpy(p, tmp);
                p_next_arg += 4;
                p += strlen(tmp);
                break;
            case 's':
                break;
            default:
                break;
            }
    }

    return (p - buf);
}
```

Valprintf 的作用就是解析 buf 字符串中的内容，如果遇到%就会进行处理，如果没有遇到%就会继续处理下一个字符。

在调用完 valprintf 后就会将修改之后的字符串 buf 作为参数调用 write，而 write 的作用就是讲 buf 的内容写入终端。

而 write 的汇编代码如下：

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

最后的 int INT_VECTOR_SYS_CALL 表示通过系统调用 sys_call 函数。

而 sys_call 的代码如下：

```
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
```

call save 作用是保存中断前的进程状态。sys_call 的作用是显示格式化了的字符串。

从 vsprintf 生成显示信息，到 write 系统函数，到陷阱-系统调用 int 0x80 或 syscall.

字符显示驱动子程序：从 ASCII 到字模库到显示 vram（存储每一个点的 RGB 颜色信息）。

显示芯片按照刷新频率逐行读取 vram，并通过信号线向液晶显示器传输每一个点（RGB 分量）。

8.4 getchar 的实现分析

异步异常-键盘中断的处理：键盘中断处理子程序，例如用户键入 crtl+c 时，系统会收到中断进程的信号，然后内核就会中断子程序。接受按键扫描码转成 ascii 码，保存到系统的键盘缓冲区。getchar 函数会将从缓冲区读取的字符转化为 ascii 码返回。

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

Unix 系统将 IO 设备都模型化为文件，所有的输入输出都被当做相应文件的读和写来执行。Unix 为读写操作提供了几组系统级的 IO，类似于 printf 和 scanf 这样的标准输入输出函数都是基于 Unix I/O 来实现的。

I/O 是操作系统中不可或缺的一部分，比如说，在进程的创建和执行过程中就需要 I/O 来执行对应的功能，除此之外打开文件，进行读写都需要用到 I/O。

Printf 的实现其中出现了“…”不定长参数，而不定长参数的实现则是通过指针来完成的，参数在栈中传递，通过指针的偏移就可以不需要知道参数的个数直接进行引用。Printf 函数的实现是系统内核和外部设备共同完成的，从进程调用 sys_call，再到通过总线将信息传输给显示器最后将字符显示在屏幕上。

（第 8 章 1 分）

结论

用计算机系统的语言，逐条总结 hello 所经历的过程。

你对计算机系统的设计与实现的深切感悟，你的创新理念，如新的设计与实现方法。

1. 预处理：处理预处理命令，将库文件展开，同时替换宏常量。
2. 编译：将 hello.i 文件转换为汇编文件。
3. 汇编：将汇编文件转换为机器代码。
4. 链接：将 hello.o 可重定位文件与库文件重定位，生成可执行文件。
5. Fork：在 shell 中为 hello 的执行创建一个子进程。
6. Execve：在 shell 中调用 execve 函数加载 hello 程序到内存中，同时开始执行 hello 程序。
7. Mmap：为 hello 程序进行内存分配。
8. 执行指令：CPU 加载指令，然后在对应的单元中执行指令。同时 OS 负责调度不同的进程之间的时间片的分配。
9. 地址翻译：hello 程序执行过程中需要从内存中读取信息是，mmu 单元负责将虚拟地址翻译为物理地址。
10. IO：printf 函数需要使用 I/O 函数来将信息显示到显示器上。
11. 信号：如果在 hello 执行过程中键入 Crtl+c 或者 Crtl+z 那么这些信号会被响应，hello 进程被中断或者被挂起。
12. 回收：在 hello 执行完成后，hello 进程会被父进程回收。

（结论 0 分，缺少 -1 分，根据内容酌情加分）

附件

列出所有的中间产物的文件名，并予以说明起作用。

hello	经过链接后的可执行文件
hello.c	源代码
hello.i	经过预处理的源代码
hello.o	进过汇编得到的可重定位文件
hello.s	进过编译得到的汇编代码
hello.o_obj.txt	hello.o 文件经过反汇编得到的文本
hello_obj.txt	hello 文件经过反汇编得到的文本
hello_elf.txt	hello 的 elf 文件信息
hello_section_header.txt	hello 文件的节信息

（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] https://www.cnblogs.com/pianist/p/3315801.html[转]printf 函数实现的深入剖析

[2] https://blog.csdn.net/shenhuan1104/article/details/76059647 gcc 预处理、编译、汇编和链接详解

（参考文献 0 分，确实 -1 分）