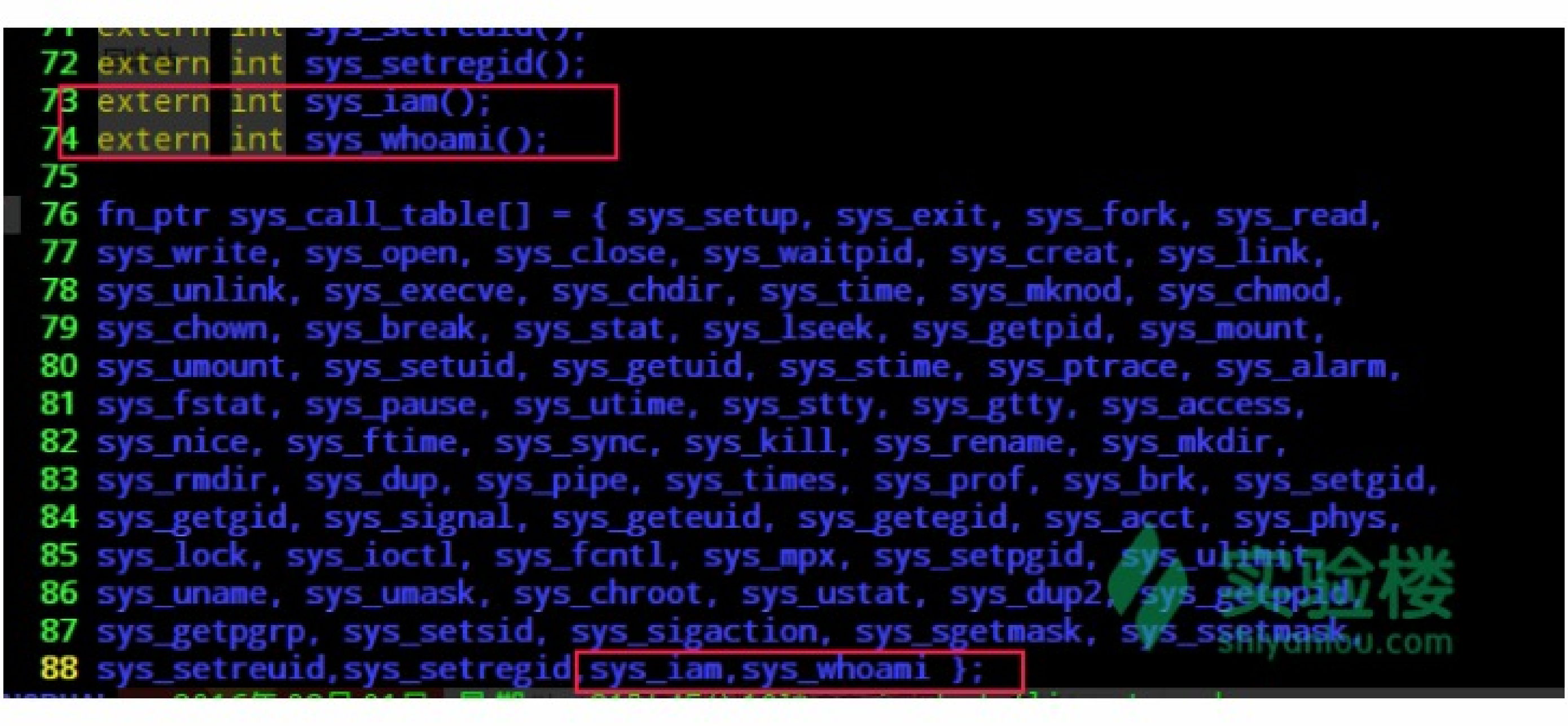
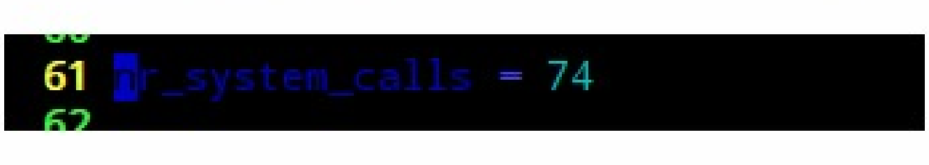
```c
#define __LIBRARY__
#include <unistd.h>
#include <errno.h>
#include <asm/segment.h>

static char myname[24] = {0};

int sys_iam(const char * name)
{
    int i = 0;
    while(get_fs_byte(name+i)!='\0')
        i++;
    if(i>23){
        return -EINVAL;
    }
    printk("%d]\n",i);
    i = 0;
    while(myname[i]= get_fs_byte(name+i)!= '\0'){
        i++;
    }
    return i;
}


int sys_whoami(char *name,unsigned int size){

    int i = 0;
    while(myname[i]!= '\0')
        i++;
    if(size<i)
        return -1;
    i = 0;
    while(myname[i] != '\0'){

        put_fs_byte(myname[i],(name+i));
        i++;
    }

    return i;
}
```

注意：在函数实现中打印信息出来，以标识系统调用是否成功，这里是内核，printf是用不了的（CPU跳不过去），要使用内核中的打印函数printk 2)增加系统调用表项（sys_call_table) 代码文件： include /linux/sys.h



```c
extern int sys_setreuid();
extern int sys_setregid();
extern int sys_iam();
extern int sys_whoami();

fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
sys_getpgrp, sys_setsid, sys_sigaction, sys_sgetmask, sys_ssetmask,
sys_setreuid,sys_setregid,sys_iam,sys_whoami };
```

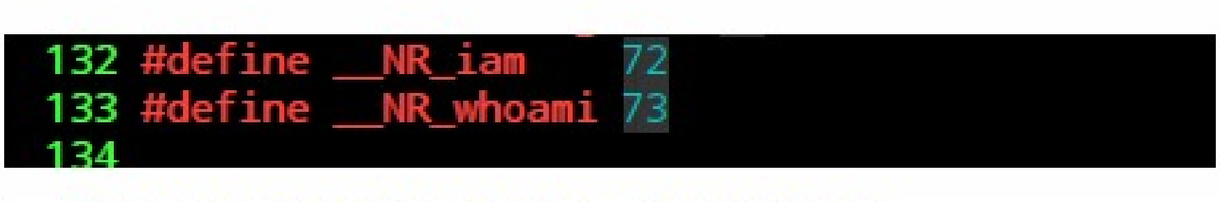
并修改系统调用总个数： 代码文件：kernel/system_call.s



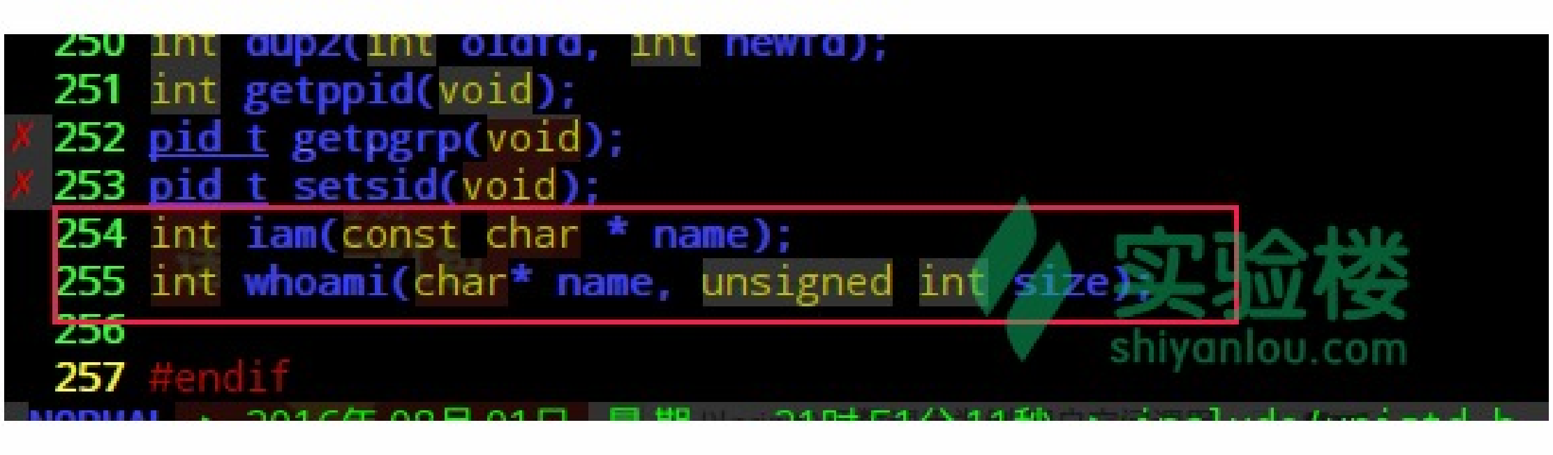
```
nr_system_calls = 74
```

以上，就是内核部分的实现，下面定义用户空间的调用

3）include /unistd.h (应用层调用） 增加



```c
#define __NR_iam     72
#define __NR_whoami  73
```

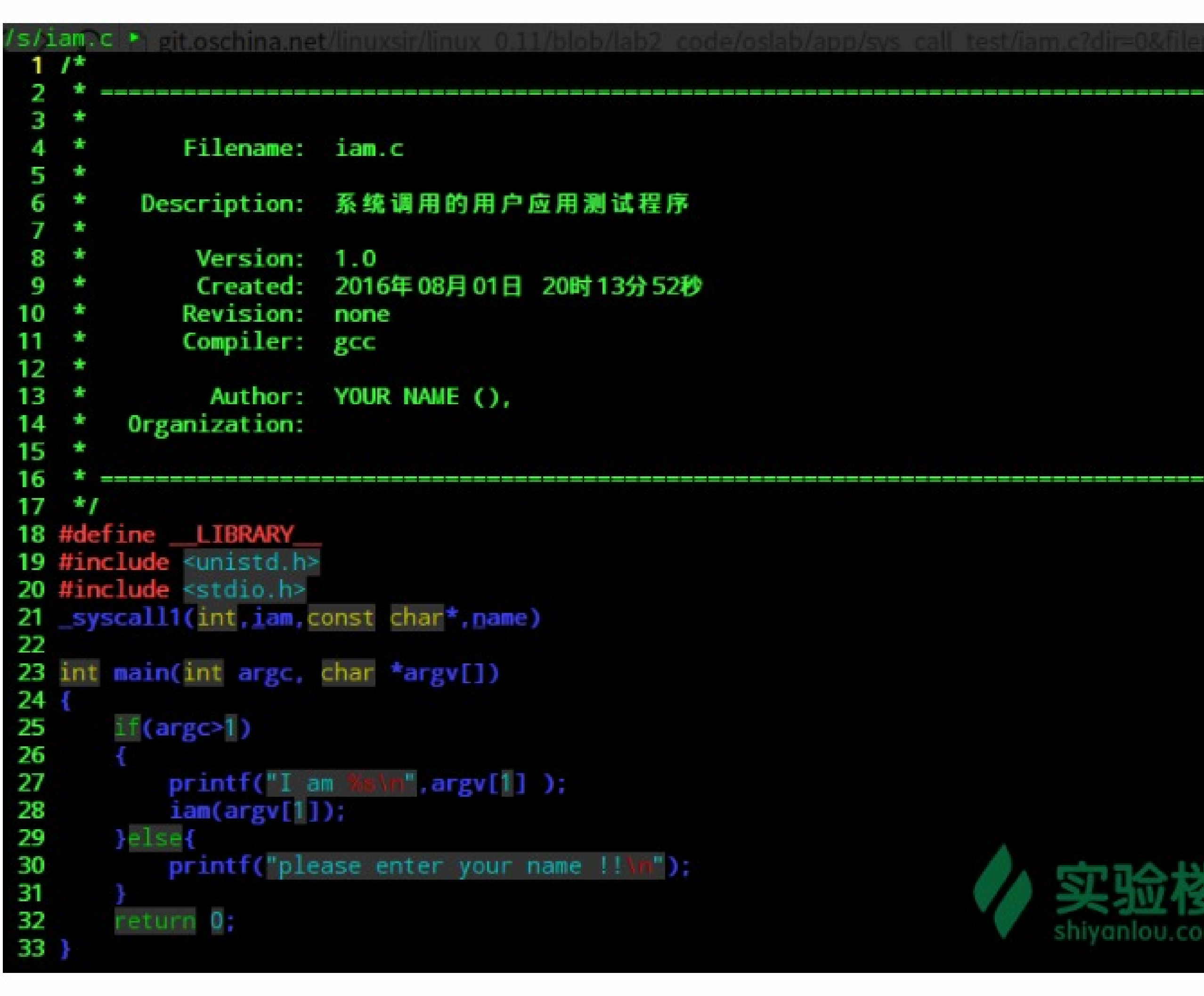


```c
int dup2(int oldfd, int newfd);
int getppid(void);
pid_t getpgrp(void);
pid_t setsid(void);
int iam(const char * name);
int whoami(char* name, unsigned int size);

#endif
```

注：上面这段定义有一个开关：#define **LIBRARY** ,使用 时要注意定义它，不然就会找不到_syscall1,_syscall2等，编译报错。 另外，还要能看穿这里函数iam() 和whoami()的面目，表面上看只有函数原型的声明，其实它是借用_syscall1 , _syscall2两个宏定义把自己隐藏了，通过这两个宏展开，便能看到函数的定义了。

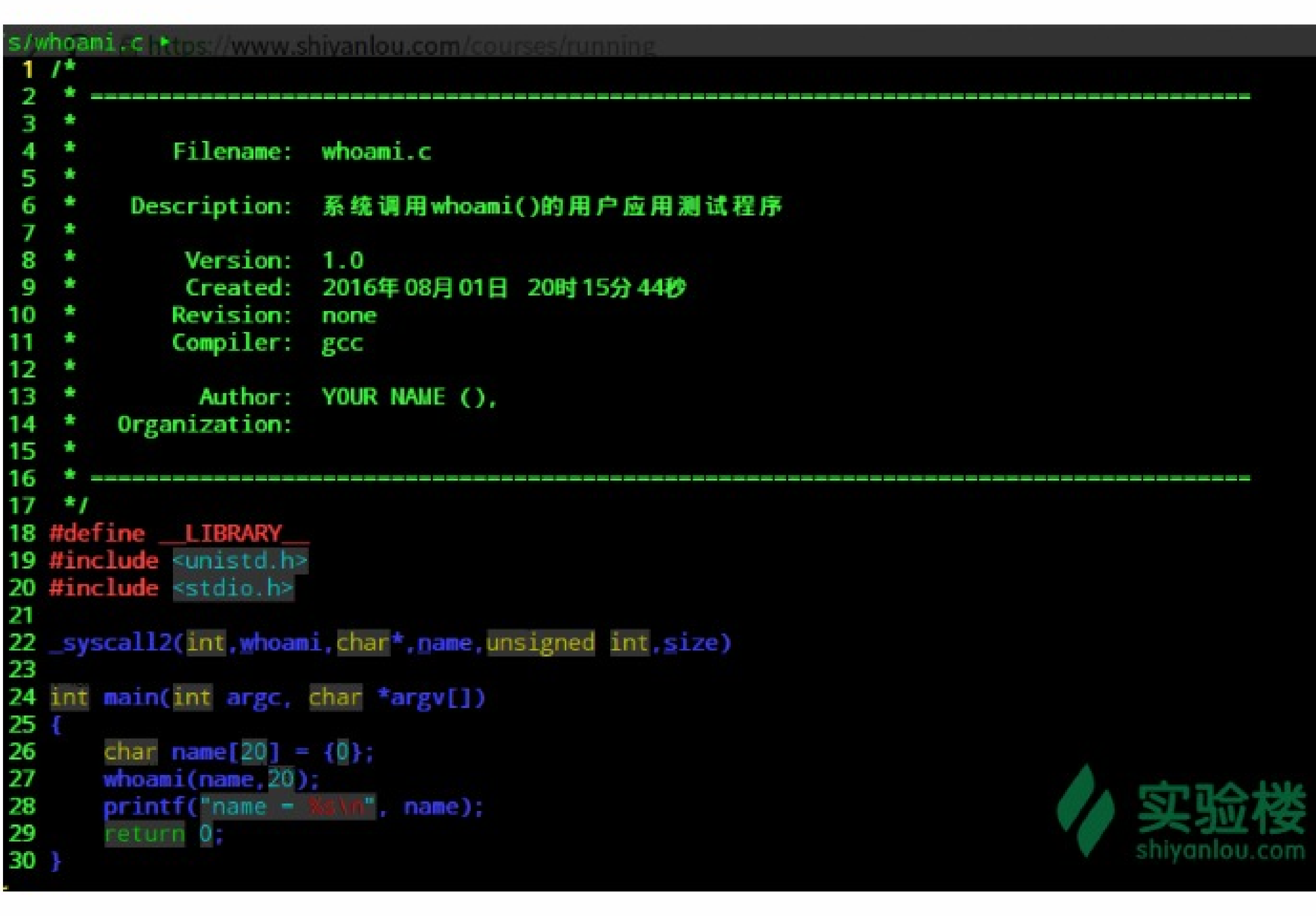
4）测试应用程序 将测试程序通过挂载(sudo ./mount-hdc)后，放到文件系统(/usr/root)下，供启动虚拟机后编译运行。

代码文件 iam.c



```c
/*
 * =====================================================================================
 *
 *       Filename:  iam.c
 *
 *    Description:  系统调用的用户应用测试程序
 *
 *        Version:  1.0
 *        Created:  2016年08月01日 20时13分52秒
 *       Revision:  none
 *       Compiler:  gcc
 *
 *         Author:  YOUR NAME (),
 *   Organization:
 *
 * =====================================================================================
 */
#define __LIBRARY__
#include <unistd.h>
#include <stdio.h>
_syscall1(int,iam,const char*,name)

int main(int argc, char *argv[])
{
    if(argc>1)
    {
        printf("I am %s\n",argv[1] );
        iam(argv[1]);
    }else{
        printf("please enter your name !!\n");
    }
    return 0;
}
```

代码文件 whoami.c

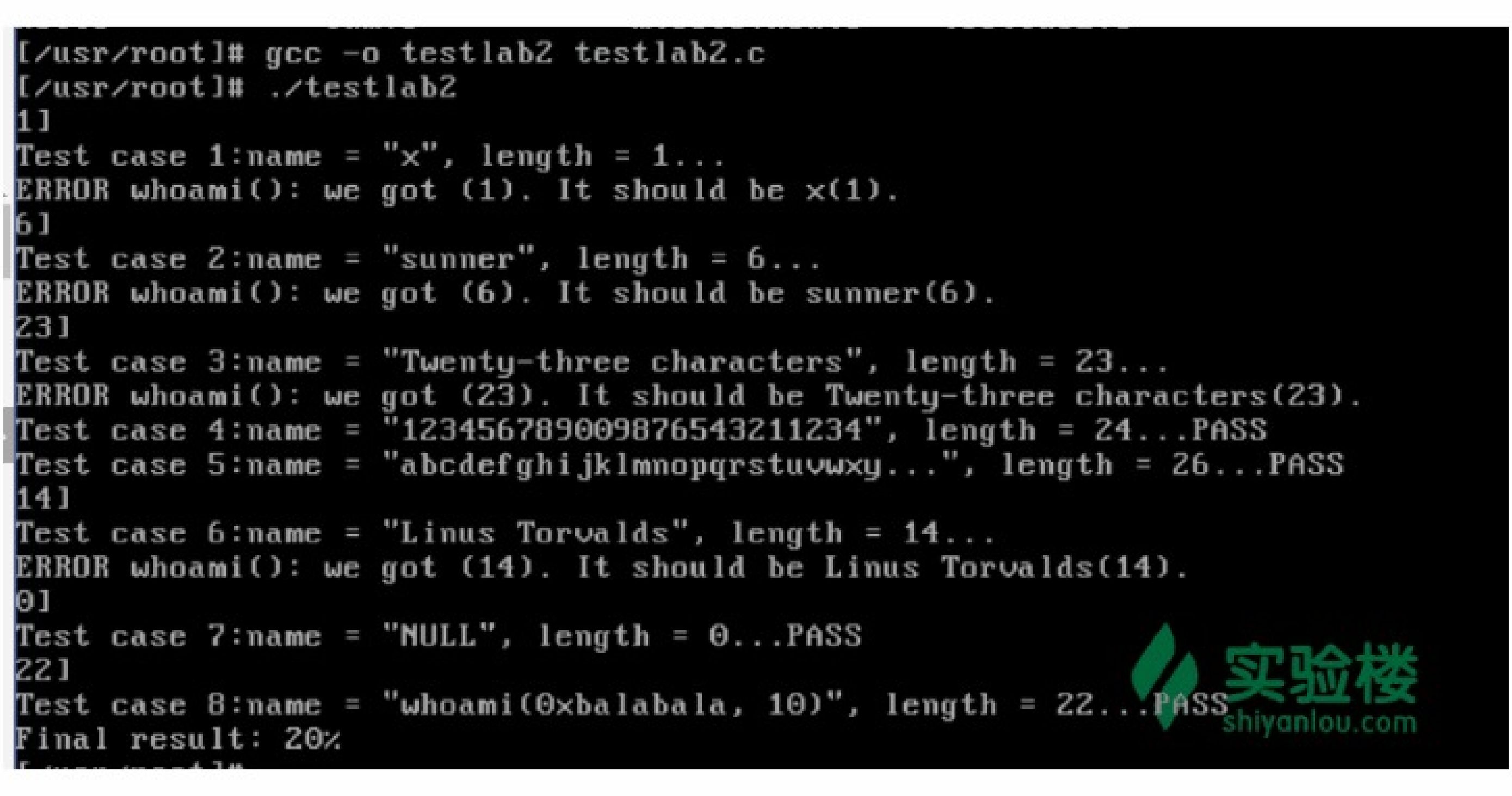


```c
/*
 * =====================================================================================
 *
 *       Filename:  whoami.c
 *
 *    Description:  系统调用whoami()的用户应用测试程序
 *
 *        Version:  1.0
 *        Created:  2016年08月01日 20时15分44秒
 *       Revision:  none
 *       Compiler:  gcc
 *
 *         Author:  YOUR NAME (),
 *   Organization:
 *
 * =====================================================================================
 */
#define __LIBRARY__
#include <unistd.h>
#include <stdio.h>

_syscall2(int,whoami,char*,name,unsigned int,size)

int main(int argc, char *argv[])
{
    char name[20] = {0};
    whoami(name,20);
    printf("name = %s\n", name);
    return 0;
}
```

注：上面两个应用调用iam()和whoami()时，需要#defin **LIBRARY** _syscall1() 这两 行，不然编译会报错，这里与API和调用合在一个文件里了（正常系统下API定义在标准库里了）。

任务2：运行老师的测试程序：

测试结果：



```
[/usr/root]# gcc -o testlab2 testlab2.c
[/usr/root]# ./testlab2
1]
Test case 1:name = "x", length = 1...
ERROR whoami(): we got (1). It should be x(1).
6]
Test case 2:name = "sunner", length = 6...
ERROR whoami(): we got (6). It should be sunner(6).
23]
Test case 3:name = "Twenty-three characters", length = 23...
ERROR whoami(): we got (23). It should be Twenty-three characters(23).
Test case 4:name = "123456789009876543211234", length = 24...PASS
Test case 5:name = "abcdefghijklmnopqrstuvwxy...", length = 26...PASS
14]
Test case 6:name = "Linus Torvalds", length = 14...
ERROR whoami(): we got (14). It should be Linus Torvalds(14).
0]
Test case 7:name = "NULL", length = 0...PASS
22]
Test case 8:name = "whoami(0xbalabala, 10)", length = 22...PASS
Final result: 20%
```

[图片]

注：第一次做这个试验，得分低了点，待日后再战！！

任务3：回答问题

1）从Linux 0.11现在的机制看，它的系统调用最多能传递几个参数？你能想出办法来扩大这个限制吗？ 从当前0.11版本来看（_syscall3的宏定义），每个系统调用使用 中断时最多只使用 了eax放系统调用号及返回值，ebx,ecx,edx三个寄存器来存放参数，所以最多能传递三个参数，若通过复杂结构的指针可以扩展这个限制，只是传递数据时比较麻烦，特别是内核态和用户态之间使用指针时。

2）用文字简要描述向Linux 0.11添加一个系统调用foo()的步骤。

从里到外的顺序： 1）在内核目录里实现sys_xxx()函数，作为系统调用在内核中的处理过程。 2）定义好系统调用号(include/unistd.h)，以及更新系统调用总数(kernel/system_call.s) ,与中断_syscall 系列宏配合好。 3）定义系统调用的API，是供用户应用来调用的。

小结：

通过这个系统调用的添加过程，更真切地感受到了系统调用从定义到实现，再到调用的整个过程，从用户空间通过中断（几个寄存器）陷入到内核空间里去执行，最后通过内核态和用户态的数据交换，更深刻理解了每个应用程序都有自己的逻辑地址空间，每个程序里的指针只是内部的逻辑地址，只在当前应用程序地址空间里有效，有效就是意味着通过此逻辑地睛能找到正确的物理内存。

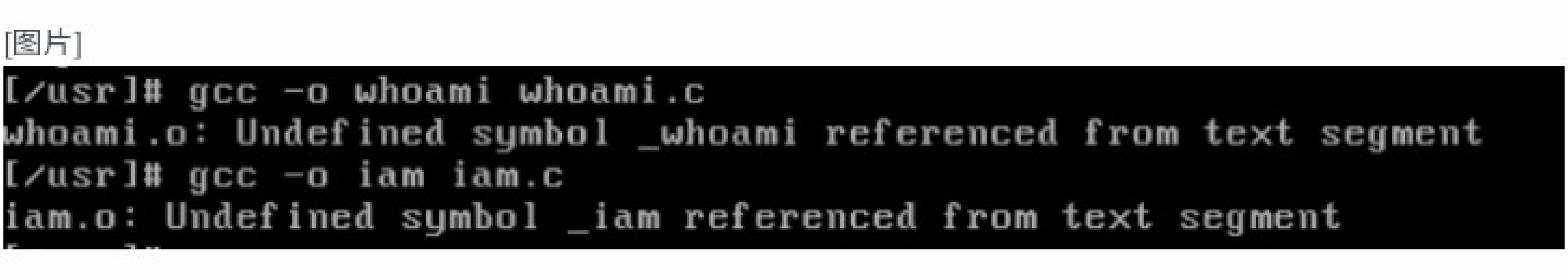
虽然在一个系统中添加新的系统调用的机会很少，但此实现的意图在于理解整个系统的架构设计，以及出于安全考虑，需要分内核态和用户态，基于这种设计，通过程序具体实现出来，又用到了逻辑地址和物理地址的隔离和映射，中断的巧妙使用，跨越用户态和内核态等，针对多个系统调用，只共用一个中断（0x80号），通过不同的中断号，通过查表(sys_call_table),使用 相应的实现函数对应起来，实现“里应外合”的效果，既实现安全，又能达到目标。

以下是做实验过程中的问题 记录：

问题1：编译出错： make[1]:正在离开目录 `/home/shiyanlou/oslab/oslab/linux_0.11/oslab/linux-0.11' ld -m elf_i386 -Ttext 0 -e startup_32 boot/head.o init/main.o \ kernel/kernel.o mm/mm.o fs/fs.o \ kernel/blk_drv/blk_drv.a kernel/chr_drv/chr_drv.a \ kernel/math/math.a \ lib/lib.a \ -o tools/system kernel/kernel.o:(.data+0x120): undefined reference to `sys_iam' kernel/kernel.o:(.data+0x124): undefined reference to `sys_whoami' make: *[tools/system] 错误 1 解决办法：修改Makefile，使who可以与其它文件一起编译进kernel.o中去，这样最终才能使用到它。

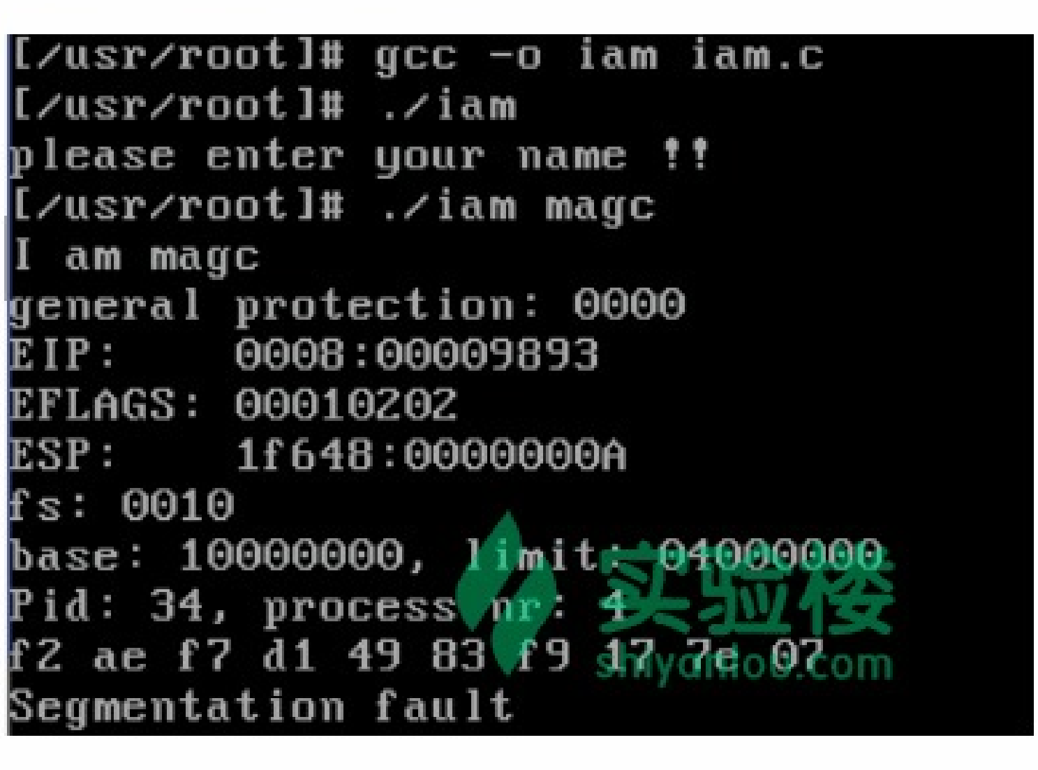
问题 2 ： 编译测试程序时报错：

[图片]



```
[/usr]# gcc -o whoami whoami.c
whoami.o: Undefined symbol _whoami referenced from text segment
[/usr]# gcc -o iam iam.c
iam.o: Undefined symbol _iam referenced from text segment
```

问题 3：运行 测试程序段错误



```
[/usr/root]# gcc -o iam iam.c
[/usr/root]# ./iam
please enter your name !!
[/usr/root]# ./iam magc
I am magc
general protection: 0000
EIP:     0008:00009893
EFLAGS: 00010202
ESP:     1f648:0000000A
fs: 0010
base: 10000000, limit: 04000000
Pid: 34, process nr: 4
f2 ae f7 d1 49 83 f9 17 7e 07
Segmentation fault
```