4. 系统调用

4.1. 实验目的

建立对系统调用接口的深入认识

掌握系统调用的基本过程

能完成系统调用的全面控制

为后续实验做准备

4.2. 实验内容

此次实验的基本内容是：在 Linux 0.11 上添加两个系统调用，并编写两个简单的应用程序测试它们。

4.2.1. iam()

4.2.2. whoami()

```
int iam(const char * name);
```

iam 的功能是：将字符串参数 name 的内容拷贝到内核中保存下来。 要求 name 的长度不能超过 23 个字符。返回值是拷贝的字符数。 如果 name 的字符个数超过了 23 ，则返回 -1 ，并置 errno 为 EINVAL。

```
int whoami(char* name, unsigned int size);
```

whoami 的功能是：将内核中由 iam() 保存的名字拷贝到 name 指向的用户地址空间中，同时确保不会对 name 越界访存（ name 的大小由 size 说明）。 返回值是拷贝的字符数。如果 size 小于需要的空间，则返回 -1 ，并置 errno 为 EINVAL 。

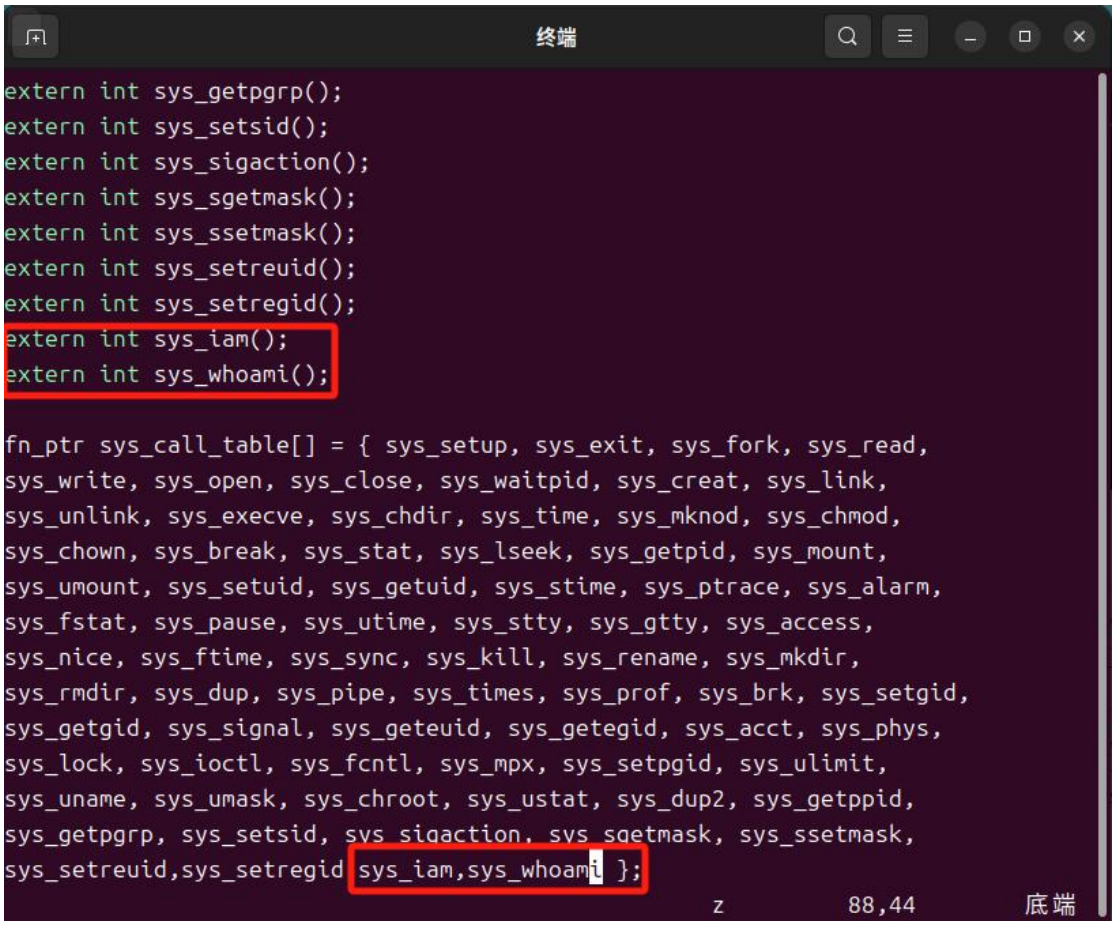
因为具体操作文件重复度高，故一起实现：

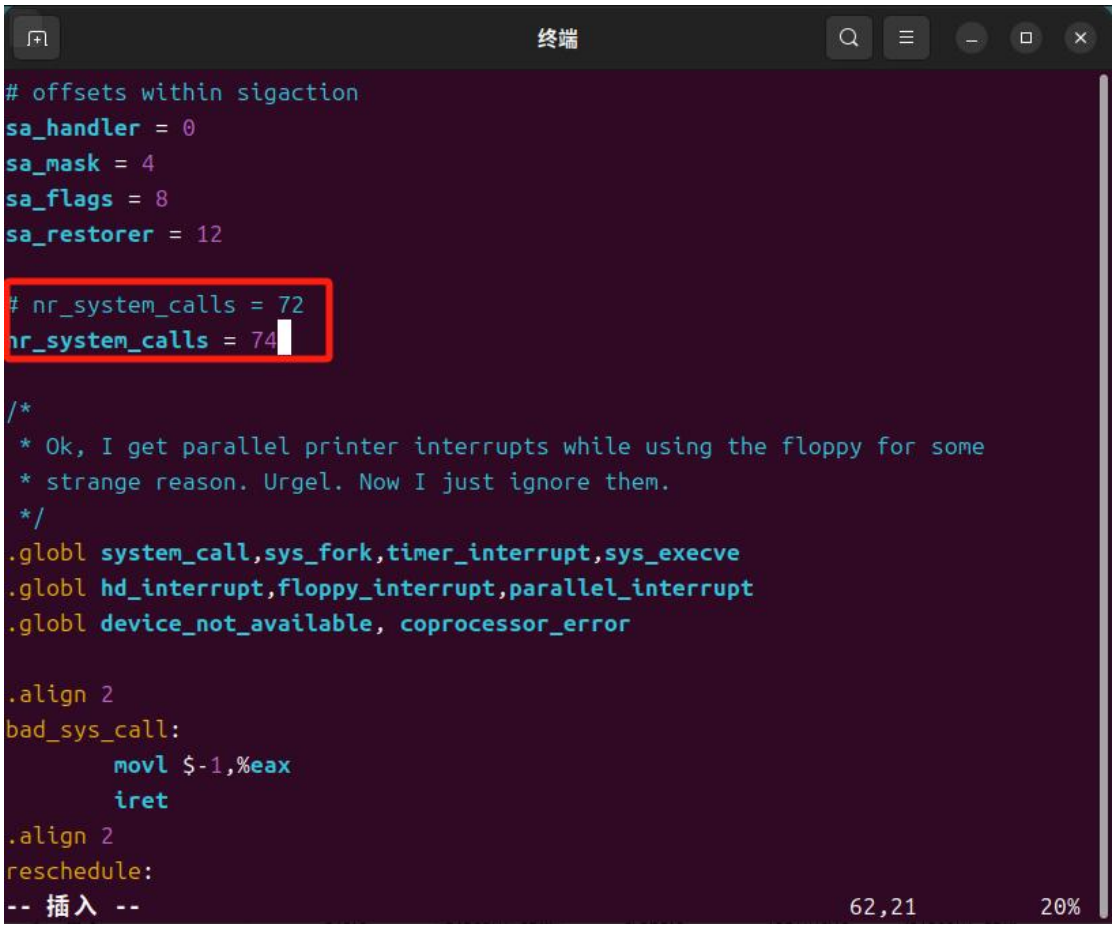
实现方式：

首先打开 unistd.h 添加 iam（为了方便把 whoami 也提前加上了）的定义，也就是添加一个能够被指令调用的宏定义，详见下图红标部分：

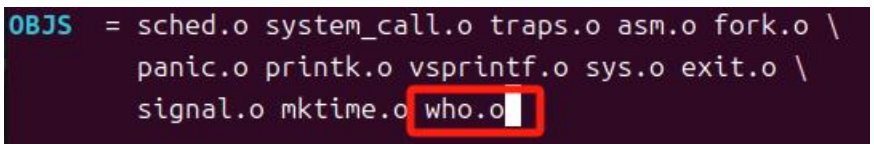
```
#define __NR_sgetmask   68
#define __NR_ssetmask   69
#define __NR_setreuid   70
#define __NR_setregid   71
#define __NR_iam        72
#define __NR_whoami     73

#define _syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name)); \
                                133,22-26     50%
```

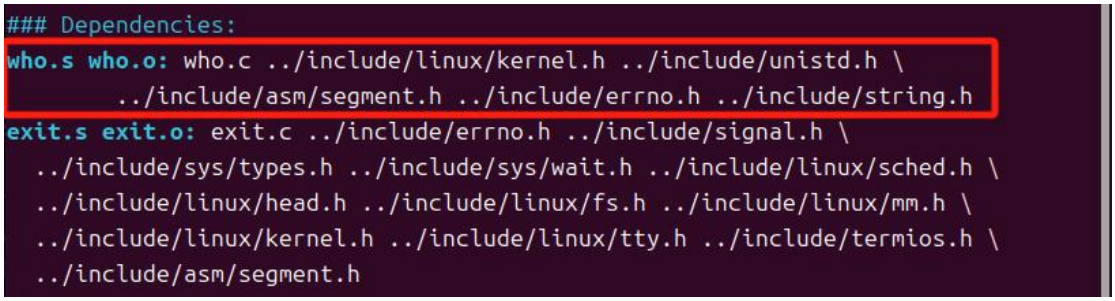
之后，在子目录的 sys.h 中也加上这两条指令，并且，将../kernel/system_call.s 中的 nr_system_calls 加 2，具体操作实现如下图：


终端
extern int sys_getpgrp();
extern int sys_setsid();
extern int sys_sigaction();
extern int sys_sgetmask();
extern int sys_ssetmask();
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 };
z
88,44
底端



然后，修改./kernel 目录下的 Makefile，我要在该目录创建的文件名是 who.c，所以修改如下：





之后，在./kernel 目录下创建 who.c 文件，用于实现 whoami 和 im 之间的功能，虽然这两个功能的目的都是将 root 的名称返回，但是为了实现用户态和内核态之间的数据传输，两者都使用了字符串指针作为参数调用，需要用到 include/asm/segment.h 中 get_fs_byte()(获得一个字节的用户空间中的数据)函数和 put_fs_byte()(将一个字节的数据传到用户空间)，两个函数的具体结构如下图：

```
static inline unsigned char get_fs_byte(const char * addr)
{
        unsigned register char _v;

        __asm__ ("movb %%fs:%1,%0":"=r" (_v):"m" (*addr));
        return _v;
}
```

```
static inline void put_fs_byte(char val,char *addr)
{
__asm__ ("movb %0,%%fs:%1"::"r" (val),"m" (*addr));
}
```

同时，观察到函数需要 errno 作为失败返回值，所以也要调用 include 目录下的 errno.h。

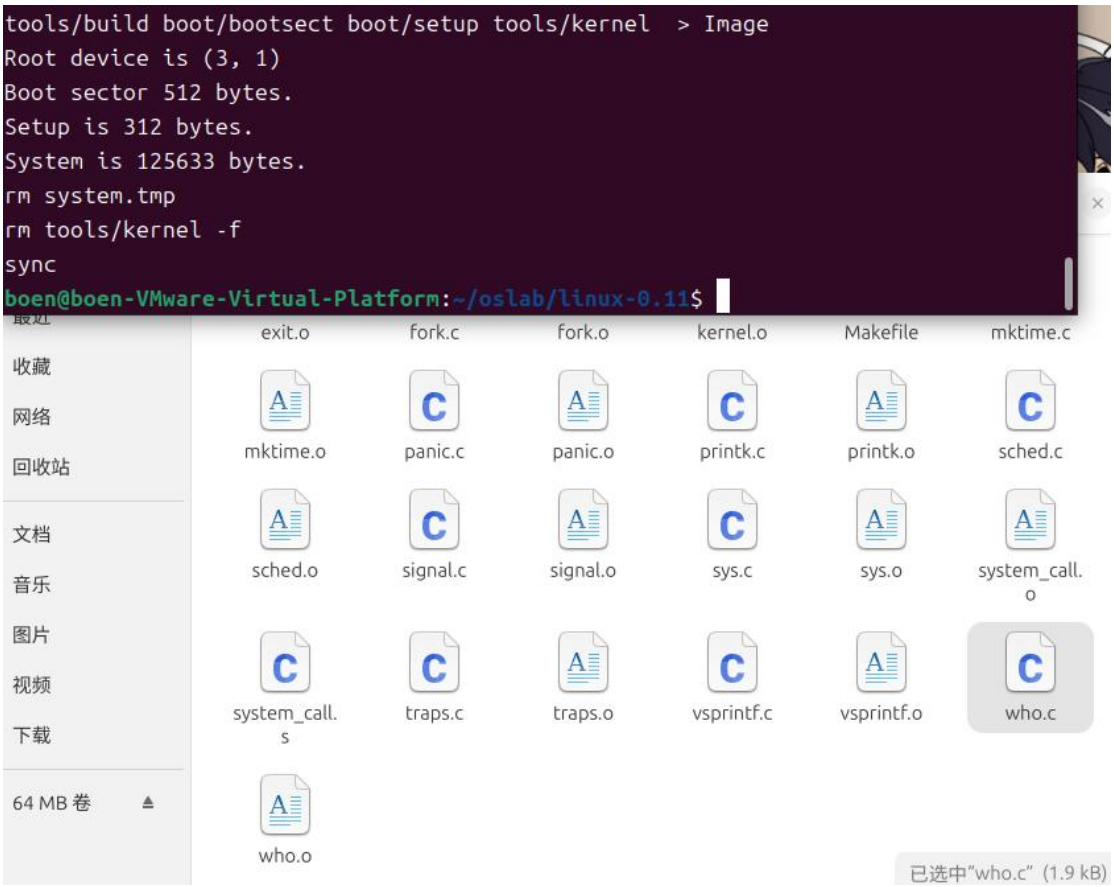
然后，实现对 who.c 的编写，实际上是在 win 上面写的，cv 到了虚拟机中，具体内容：

```
#include <string.h>
#include <linux/kernel.h>
#include <unistd.h>
#include <errno.h>
#include <asm/segment.h>
```

```
char msg[24];       //定义一个存储字符串内容的字符串数组,23个字符内容加一个\0

int sys_iam(const char * name)
{
    int len = 0;
    char buffer[24];
    for(len = 0;len < 24;len++)
    {
        buffer[len] = get_fs_byte(name + len);
        if(buffer[len] == '\0')              //如果在24次读入中存在\0则存入msg
        {
            break;
        }
    }
    if(len == 24 && buffer[23] != '\0')
        return -(EINVAL);
    else
    {
        strcpy(msg,buffer);
        return len;
    }
}
```

```c
int sys_whoami(char* name, unsigned int size)
{
    int len = strlen(msg);                    //strlen统计全局变量msg的长度
    if(len > size)                            //如果 size 小于需要的空间，返回异常
        return -(EINVAL);
    else
    {
        int i;
        for(i = 0; i < len;i++)
        {
            put_fs_byte(msg[i],name + i);
        }
        return len;                           //返回长度
    }
}
```

然后，在该目录下 make，生成.o 文件，之后在../(也就是 0.11 目录）下 make，(或者直接 make all）之后进行下一步的任务,make 结果如下:



4.2.3. 测试程序

首先，在 oslab 目录下执行./mount-hdc 将生成的镜像文件移动到 hdc 目录下，然后对 hdc 目录进行检查：

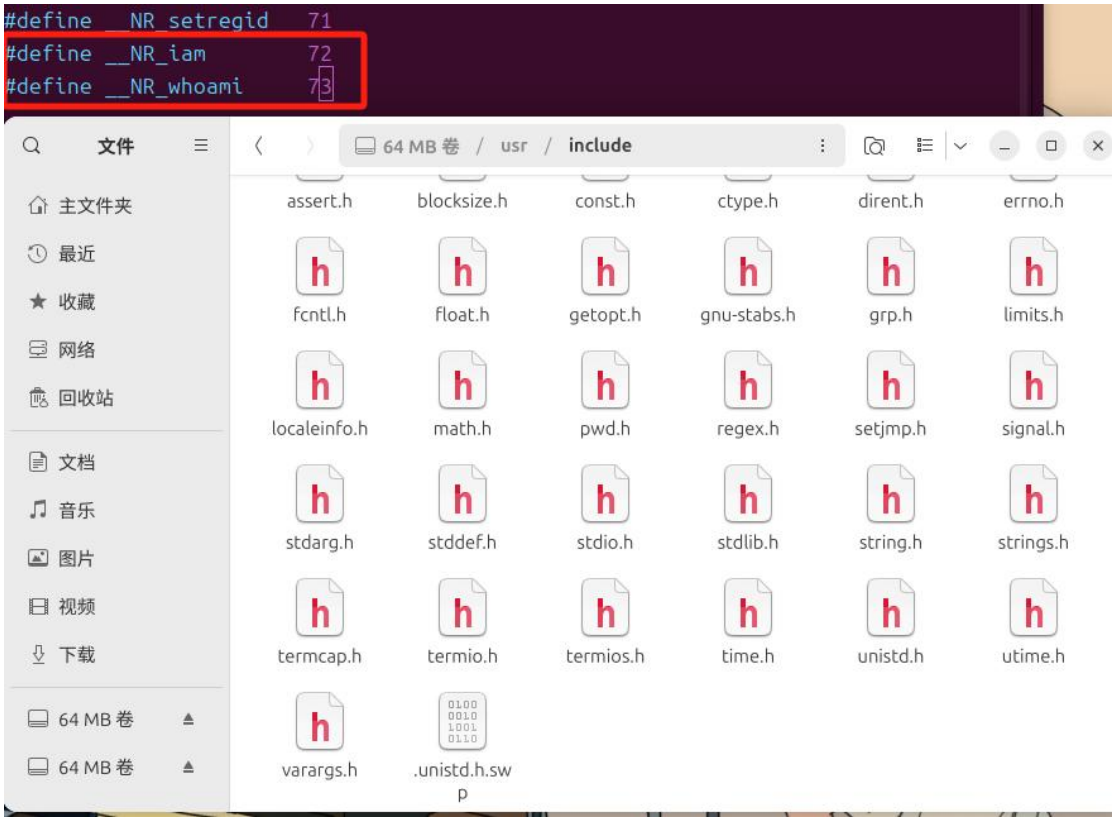
```
boen@boen-VMware-Virtual-Platform:~/oslab/linux-0.11$ cd ../
boen@boen-VMware-Virtual-Platform:~/oslab$ ./mount-hdc
[sudo] boen 的密码：
boen@boen-VMware-Virtual-Platform:~/oslab$ cd hdc
boen@boen-VMware-Virtual-Platform:~/oslab/hdc$ ls -a
.  ..  bin  dev  etc  image  Image  mnt  shoelace  tmp  usr  var
boen@boen-VMware-Virtual-Platform:~/oslab/hdc$ cd usr
boen@boen-VMware-Virtual-Platform:~/oslab/hdc/usr$ ls -a
.  ..  bin  docs  include  local  root  src  tmp  var
boen@boen-VMware-Virtual-Platform:~/oslab/hdc/usr$
```

之后，编写 iam.c whoami.c，将其与 testlab2.c testlab2.sh 移动到 hdc/usr/root 目录下，两个文件分别内容如下：

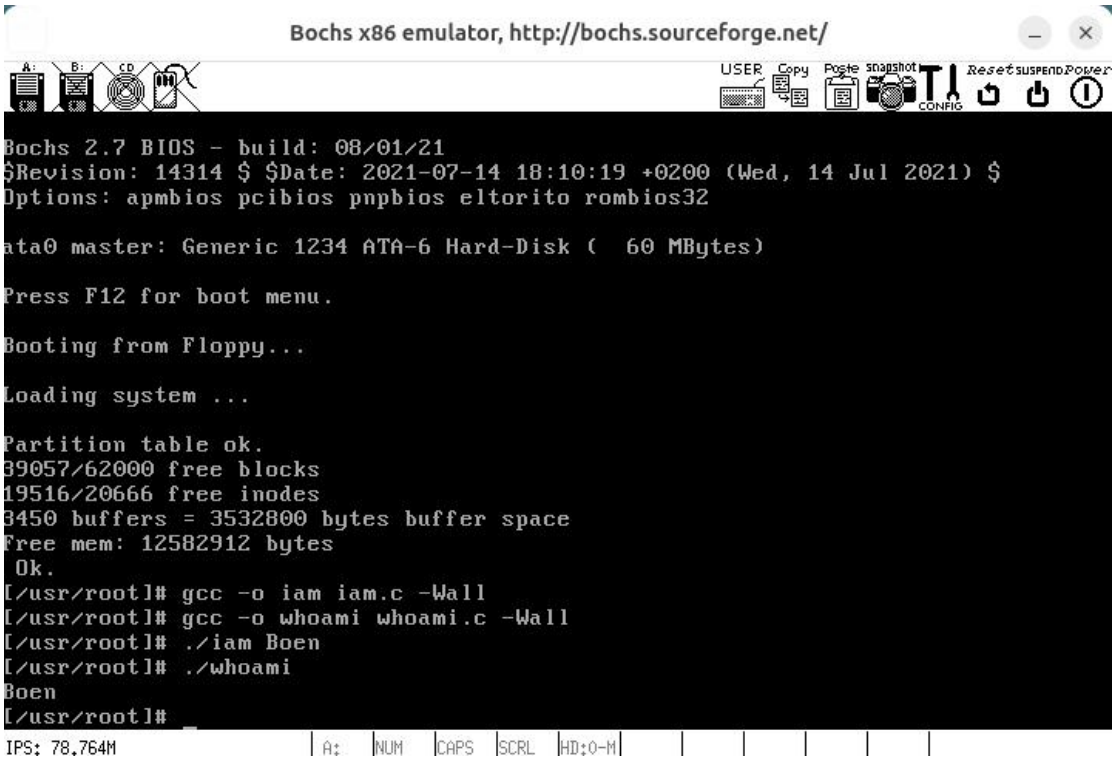
```c
#define __LIBRARY__
#include <unistd.h>
#include <stdio.h>
_syscall1(int,iam,const char*,name);

int main(int argc, char ** argv)
{
        int len = 0;
        if(argc < 1){
                printf("not enough arguments!\n");
                return -1;
        }
        len = iam(argv[1]);
        return len;
}
```

```c
#define __LIBRARY__
#include <unistd.h>
#include <stdio.h>
_syscall2(int, whoami, char*, name, unsigned int, size);

int main(){
        char msg[24];
        int len = 0;
        len = whoami(msg, 24);
        printf("%s\n",msg);
        return len;
}
```

然后，将../usr/include 中的 unistd.h 进行修改，方法于上面相同：



之后通过 bochs 运行并评分，具体流程以及评分结果如下：



```
[/usr/root]# gcc -o testlab2 testlab2.c -Wall
[/usr/root]# ./testlab2
Test case 1:name = "x", length = 1...PASS
Test case 2:name = "sunner", length = 6...PASS
Test case 3:name = "Twenty-three characters", length = 23...PASS
Test case 4:name = "123456789009876543211234", length = 24...PASS
Test case 5:name = "abcdefghijklmnopqrstuvwxy...", length = 26...PASS
Test case 6:name = "Linus Torvalds", length = 14...PASS
Test case 7:name = "NULL", length = 0...PASS
Test case 8:name = "whoami(0xbalabala, 10)", length = 22...PASS
Final result: 50%
[/usr/root]# ./testlab2.sh
Testing string:Sunner
PASS.
Testing string:Richard Stallman
PASS.
Testing string:This is a very very long string!
PASS.
Score: 10+10+10 = 30%
[/usr/root]#
```

如图，全部满昏！

4.3. 实验报告

在实验报告中回答如下问题：

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？

Linux 中（Linux-0.11/inlcude/unistd.h）有四个关于系统调用的方法：_syscall0、_syscall1、_syscall2、_syscall03。

其中#define _syscall3(type,name,atype,a,btype,b,ctype,c)包含三个参数。系统调用号存放在 eax 寄存器，而各个参数则分别存放在 ebx、ecx 和 edx 寄存器中。通过执行 `int $0x80` 汇编指令触发系统调用。

因此 Linux 0.11 中最多支持系统调用传递 3 个参数。

你能想出办法来扩大这个限制吗？

在硬件上，可以多设置几个参与传递参数的寄存器供操作系统使用，以扩展_syscall 的范围，或者，可以利用堆栈的方式将所需要的参数压入栈中，通过用户栈和内核栈的切换来获取更多参数，

用文字简要描述向 Linux 0.11 添加一个系统调用 foo() 的步骤。

与上面添加 whoami 和 iam 的操作相同，只是名称不同：

首先，在 linux-0.11/include/unistd.h 中：

添加系统调用号宏定义'#define __NR_foo 调用号+1

然后在./linux/sys.h 中：

1.添加 extern int sys_foo();

2.在 fn_ptr sys_call_table[]中加入 sys_foo

接着在../kernel/system_call.s 中：

修改 nr_system_calls 的值，使其加一

同时，在 linux-0.11/kernel 中：

创建 foo.c，并在其中实现供用户调用的系统调用函数 sys_foo()

在/kernel/Makefile 中：

1.OBJS 中加入 foo.o

2.Dependencie 下添加 foo.s foo.o:foo.c + 所需要的头文件

最后在用户程序中选择相应的参数调用方式_syscallN（N 是参数个数，回答第一问提及）完成用户程序 foo()的调用

4.3.1. 评分标准

将 testlab2.c 在修改过的 Linux 0.11 上编译运行，显示的结果即内核程序的得分。满分 50%
只要至少一个新增的系统调用被成功调用，并且能和用户空间交换参数，可得满分
将脚本 testlab2.sh 在修改过的 Linux 0.11 上运行，显示的结果即应用程序的得分。满分 30%
实验报告，20%