

操作系统实验二

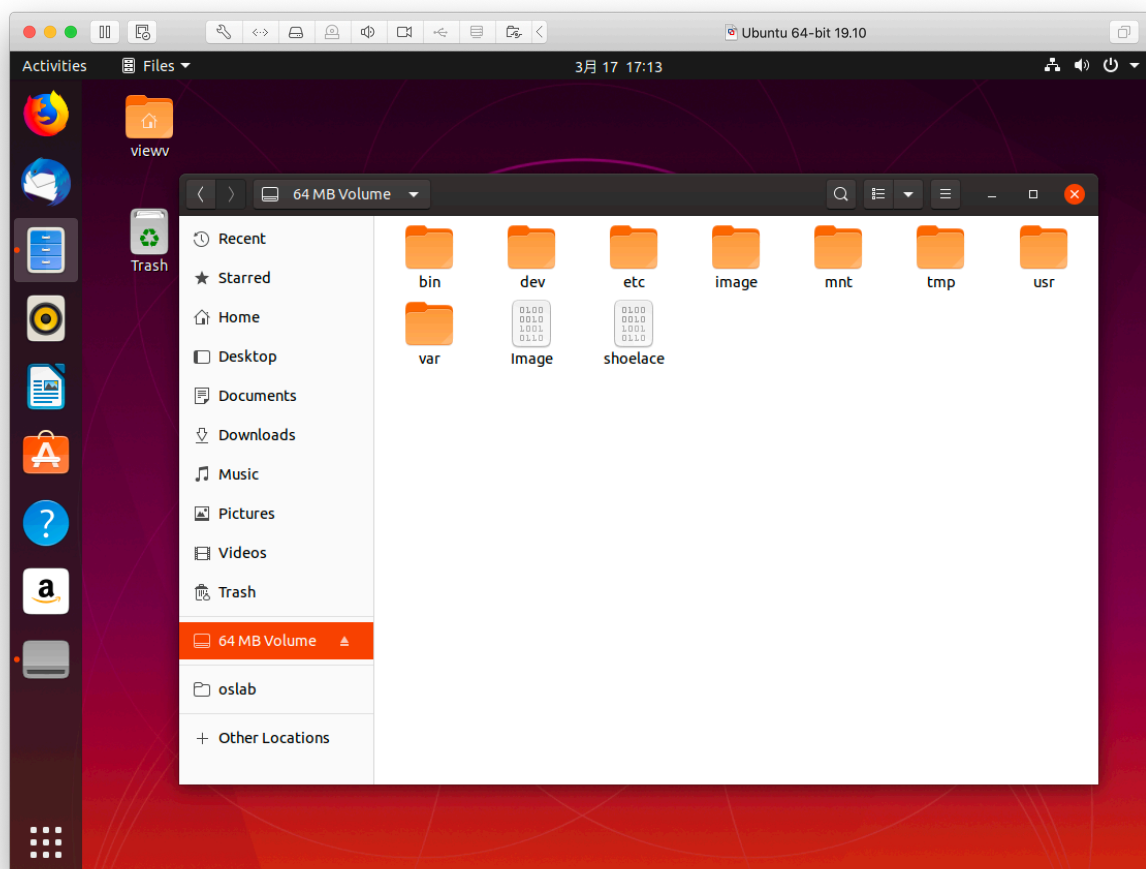
前言

这次实验，实验手册还是提示了很多东西的，可以说跟着实验手册加上测试一下内核这种层次的C语言简单怎么写就差不多可以实现了，但是还是有很多细节的东西需要讨论。

还是先来说明环境的问题，这次又遇到很麻烦的事情，这个看着很舒服的开发环境因为这个又不优雅了。

首先，针对这个老版本的Linux 0.11，其支持的文件系统应该是只有一种，也就是 MINIX 文件系统，这种文件系统虽然老，但是大概大家Linux是支持的，但是这时候我们会发现，默认的docker的ubuntu是不支持的，（这里还我还不确定是docker这种方式彻底无法内部挂载还是说不能挂载MINIX，因为docker本质上没有内核）也就是说，在docker上面，不能挂载hdc的img镜像文件，而在Mac上面，原生系统同样不支持MINIX文件系统，这也就是说，无法将这个img挂载到一个地方，方便的和内部交互。

为什么要挂载这个hdc-0.11.img呢，是因为其实这个img打开里面就是bochs那个虚拟机的内部文件：



在Ubuntu里面直接点开镜像文件显示的内容

这样我们其实可以再外部写C语言等等程序，之后拷贝到这个镜像内部，之后运行文件就在里面了，这对比在bochs系统里面对着黑框子vi写要好多了（不是说vi不好，而是这个Linux内部带的感觉不算好用，而且bochs毕竟是虚拟机，费资源而且每次我的电脑都会风扇转最快，发热严重）

我现在还没有找到方法，幸好我还有台装了Manjaro的电脑，没有的人也许就要回归虚拟机了，这样每次改测试都要换个电脑或者打开虚拟机，或者vi打，或者在那个manjaro（那个电脑环境我还没配置）上面写，似乎都不算很优雅。其实这里应该老师的本意也是方便挂载，所以在实验根目录下还有一个mount-hdc的脚本方便你挂载hdc，这里我也是用不到了。

实验二 系统调用

这次的实验目的算是给Linux加两个API函数，实验的目的是让我们知道，怎么实现添加API函数，首先我们先看要求：

iam

第一个系统调用是iam()，其原型为：

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

完成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。

在kernal/who.c中实现此系统调用。

总结下来，就有这么几个要求：

1. 字符串参数name拷贝到内核中保存下来，我们知道，系统是区分管态和目态的，也就是这里有内核状态和用户状态，这两个状态之间需要某种办法才能做到访问数据。
2. name长度不能超过23位，这算是实现的时候的逻辑要求了，实际上我们都知道，这样就相当于24，23个字符加终止符'\0'，这里就需要加以判断，如果大于24的话，其实就要报错了。

whoami

第二个系统调用是whoami()，其原型为：

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

它将内核中由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。

也是在kernal/who.c中实现。

总结下来，就有这么几个要求：

1. name是指向用户地址空间的，也就是数据要放到用户状态下的name指针那里，而且不能越界，也就是说，这里的size要大于或等于我们前面在iam存放的字符串长度。
2. 返回值是拷贝的字符数，这个地方我第一次写直接写执行成功return 0了，我还是不太细心。

这样，两个函数的要求就明朗了，之后还是讨论一个问题，系统API是怎么被调用的。

系统调用的过程

这个过程具体还是很复杂，但是要是增加一个API其实可以照猫画虎来完成的，首先找一个系统的API看一看是怎么实现的，之后我们就找到了第一个目的地，unistd.h头文件。

unistd.h

在这个头文件中，里面定义了一些宏，Linux 0.11最多支持三个参数的系统函数，所以就有下面这四种宏：

```
#define _syscall0(type,name) \
type name(void) \
{ \
    long __res; \
    __asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name)); \
    if (__res >= 0) \
        return (type) __res; \
    errno = -__res; \
    return -1; \
}

#define _syscall1(type,name,atype,a) \
type name(atype a) \
{ \
    long __res; \
    __asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name), "b" ((long)(a))); \
    if (__res >= 0) \
        return (type) __res; \
    errno = -__res; \
    return -1; \
}

#define _syscall2(type,name,atype,a,btype,b) \
type name(atype a,btype b) \
{ \
    long __res; \
    __asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name), "b" ((long)(a)), "c" ((long)(b))); \
    if (__res >= 0) \
        return (type) __res; \
    errno = -__res; \
    return -1; \
}
```

```
#define __syscall3(type,name,atype,a,btype,b,ctype,c) \
type name(atype a,btype b,ctype c) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
: "=a" (__res) \
: "0" (__NR_##name), "b" ((long)(a)), "c" ((long)(b)), "d" ((long)(c))); \
if (__res>=0) \
return (type) __res; \
errno=-__res; \
return -1; \
}
```

这种方式来实现函数我其实还是头一次看见，这里面其实重要的是中间这里有这么一个东西：

```
long __res; \
__asm__ volatile ("int $0x80" \
: "=a" (__res) \
: "0" (__NR_##name), "b" ((long)(a)), "c" ((long)(b)), "d" ((long)(c))); \
if (__res>=0) \
return (type) __res; \
errno=-__res; \
return -1; \
```

这里的 `__NR_##name` 就是调用号，这个地方会通过 `int $0x80` 实现一种说过的中断格式来调用上面的号，之后我们在上面就可以发现这些调用号是在这个地方声明的，所以需要在这里添加这次要实现两个多两个函数的调用号。

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

仿照写法，就是前面加这个前缀，之后号码顺延，写成了72，73

system_call.s

这里又需要对汇编代码进行修改，但是这里我觉得先得看这个地方：

```
system_call:
    cmpl $nr_system_calls-1,%eax
    ja bad_sys_call
    push %ds
    push %es
    push %fs
    pushl %edx
```

```

pushl %ecx    # push %ebx,%ecx,%edx as parameters
pushl %ebx    # to the system call
movl $0x10,%edx # set up ds,es to kernel space
mov %dx,%ds
mov %dx,%es
movl $0x17,%edx # fs points to local data space
mov %dx,%fs
call sys_call_table(,%eax,4)
pushl %eax
movl current,%eax
cmpl $0,state(%eax) # state
jne reschedule
cmpl $0,counter(%eax) # counter
je reschedule

```

这个地方可以看到注释 `push %ebx,%ecx,%edx as parameters`，而 `ebx`, `ecx`, `edx` 实际上应该是通用寄存器，这样就知道了为什么支持最多三个参数的调用，这样大致上如果需要修改支持更多参数的话，应该可以在这里增加寄存器个数，之后估计可以增加更多的函数调用，难度颇大。在这部分还可以看到 `eax` 的作用，其实 `eax` 就是在找调用系统函数的位置，这个函数就在另一个地方定义。

同时要注意前面的一行对实验很关键的代码：

```
nr_system_calls = 74
```

这个是我已经修改过后的代码，这个地方应该是偏移量，也就是说又多少个系统调用，如果不增加两个的话，是肯定找不到我们新定义的系统调用的，这是很小的一点，但是非常重要。

sys.h

有函数肯定要有声明，那么声明在什么地方呢，就是在这个 `sys.h` 中声明，这是非常重要的一点，不过这个地方较为简单，只需要在最后加入：

```

extern int sys_whoami();
extern int sys_iam();

```

之后在下面的数组中也加入这两个函数引用，这个时候就知道了前面哪些号码的意思，也就是在这里其实是这个函数数组的位置，所以一定要写对。

实现

之后就是实现了，这个地方还是有很多需要注意的地方：

printk

在内核态中，是无法使用用户态的 `printf` 函数的，所以在 `debug` 或者内核态需要输出信息的时候，不能使用 `printf`，而是需要使用类似的 `printk` 函数来输出信息。

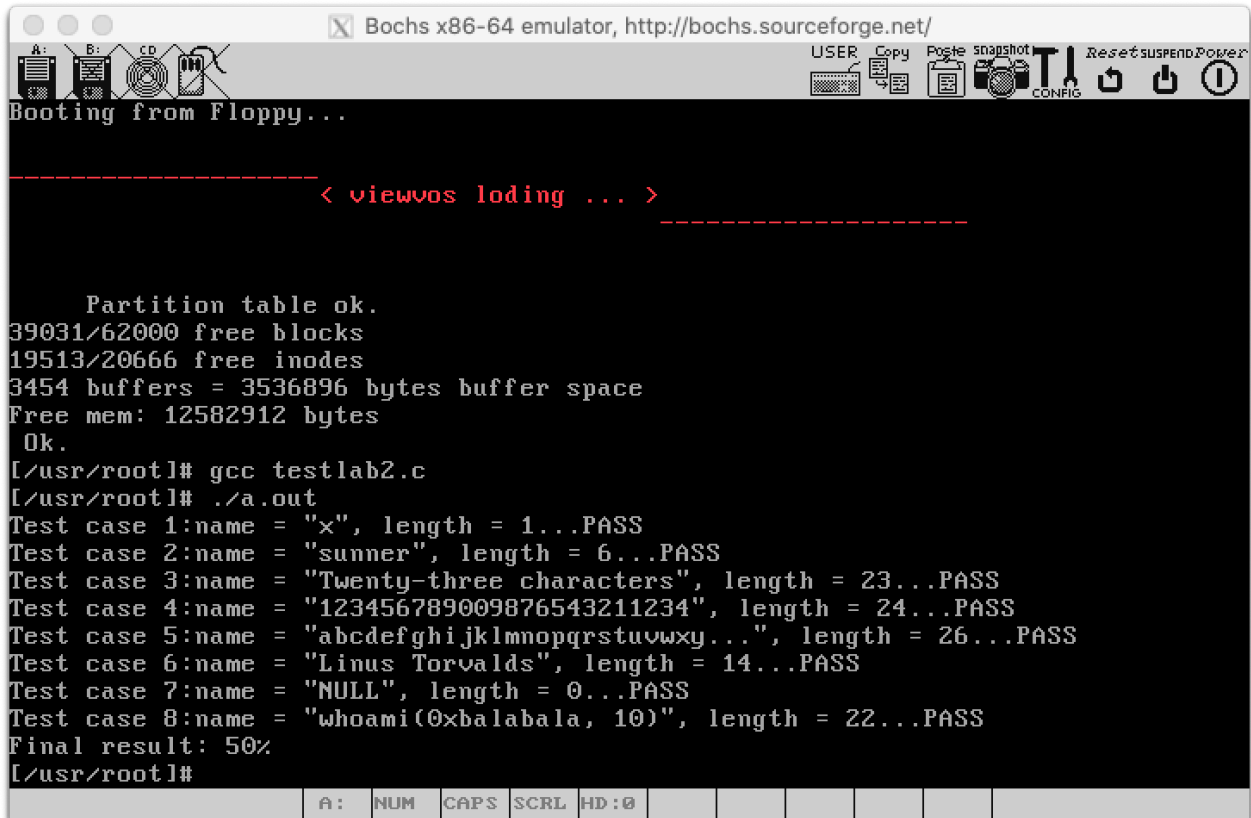
put_fs_byte, get_fs_byte

这两个函数负责内核态和用户态之间交互数据，其中get就是从用户态获取数据，每次提取一个字节，正好就是一个char，put就是放入数据，有了这两个函数，就可以实现这次实验了。

具体的最后修改出来的文件，都放在文件夹下面了，在这里按照实验手册也需要对kernel的makefile进行修改，我还写了一个miao.sh脚本用来编译iam.c 和 whoami.c，要不然每次输入那么多太累了。

运行测试

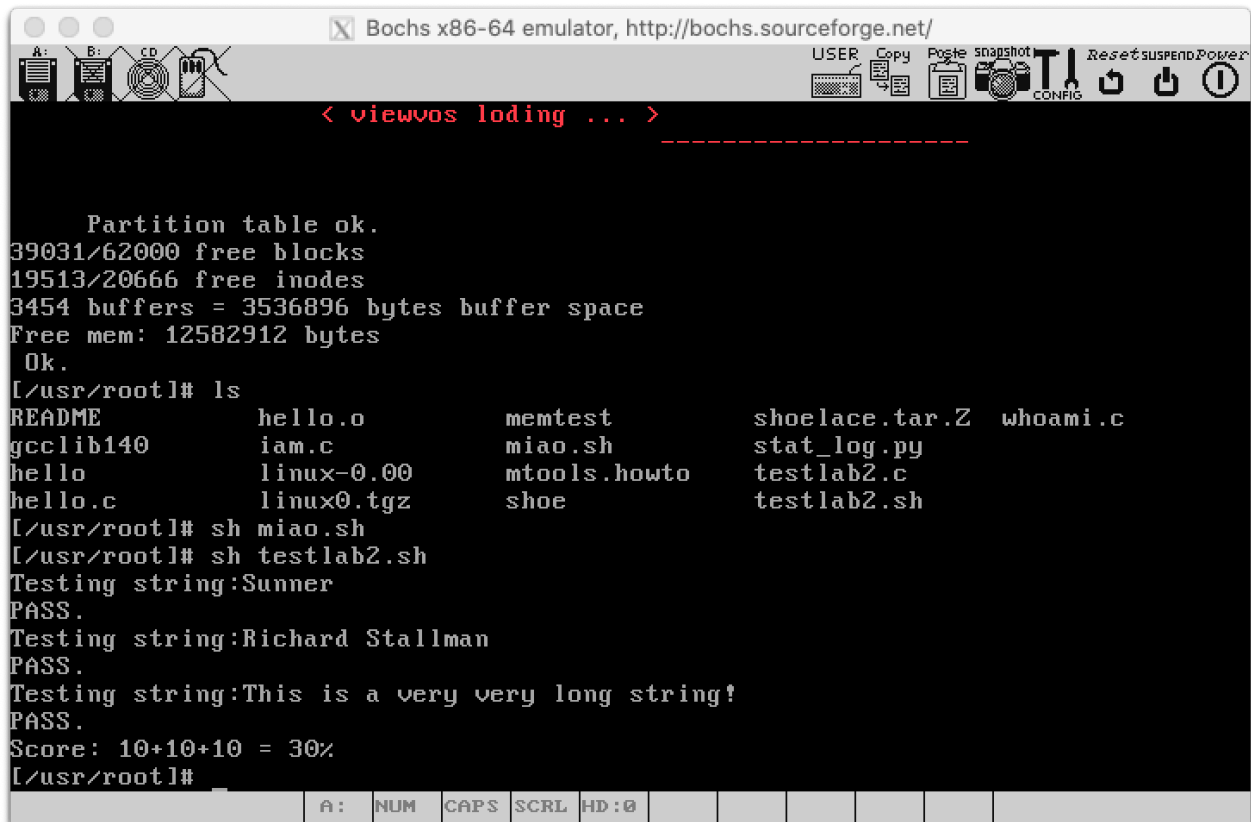
这次实验，提供了两个测试文件，也就是testlab2.c和testlab2.sh，针对C程序要编译通过，最后显示出来的必须都是pass，第二个sh脚本文件，需要先写好iam和whoami的用户程序，之后运行脚本文件检验结果，下面是我这里成功的结果：



```
Bochs x86-64 emulator, http://bochs.sourceforge.net/
Booting from Floppy...
-----
< viewvos loding ... >
-----

Partition table ok.
39031/62000 free blocks
19513/20666 free inodes
3454 buffers = 3536896 bytes buffer space
Free mem: 12582912 bytes
Ok.
[/usr/root]# gcc testlab2.c
[/usr/root]# ./a.out
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 = "abcdefghijklmnopqrstuvwxyz...", 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.c编译之后执行全部pass，这部分占最后结果的50%



```
Bochs x86-64 emulator, http://bochs.sourceforge.net/
< viewvcs loding ... >
-----
Partition table ok.
39031/62000 free blocks
19513/20666 free inodes
3454 buffers = 3536896 bytes buffer space
Free mem: 12582912 bytes
Ok.
[/usr/root]# ls
README          hello.o          memtest          shoelace.tar.Z   whoami.c
gcclib140       iam.c            miao.sh          stat_log.py
hello           linux-0.00       mtools.howto     testlab2.c
hello.c         linux0.tgz       shoe             testlab2.sh
[/usr/root]# sh miao.sh
[/usr/root]# sh 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]#
```

testlab2.sh的运行结果，全部pass，这部分占最后结果的30%

加上实验报告的20%，这也就是最后的结果，实现了编写一个系统API的工作流程。

最后其实还是想说，这里面学问还是很大，这样最后是实现了结果，但是感觉最后还是在Linus给的更底层的接口上面，跳了一次舞。