系统调用-实验报告

2021113117 王宇轩

1. 修改unistd.h

添加系统调用时需要修改`include/unistd.h`文件，使其包含`__NR_whoami`和`__NR_iam`。所作的修改如下，添加了这两个系统调用的宏包含，分别分配系统调用号72和73。

```
129 #define __NR_ssetmask    69
130 #define __NR_setreuid    70
131 #define __NR_setregid    71
132 #define __NR_iam         72
133 #define __NR_whoami      73
134
135 #define _syscall0(type,name) \
136 type name(void) \
137 { \
138 long __res; \
139 __asm__ volatile ("int $0x80" \
```

2. 修改system_call.s

从`int 0x80`进入内核函数后，Linux 0.11会自动调用函数`system_call`，是一个汇编函数，见`kernel/system_call.s`，其中`nr_system_calls`表明了系统调用总数，我们将其改为74。

```
55 # offsets within sigaction
56 sa_handler = 0
57 sa_mask = 4
58 sa_flags = 8
59 sa_restorer = 12
60
61 nr_system_calls = 74
62
63 /*
64  * Ok, I get parallel prin
65  * strange reason. Urgel. I
66  */
67 .globl system_call,sys_fork
68 .globl hd_interrupt,floppy
```

3. 修改sys.h

`system_call`中，用如下指令获取处理函数的地址指针。`sys_call_table`就是函数指针数组的起始地址

```
call sys_call_table + 4 * %eax
```

`sys_call_table`就是函数指针数组的起始地址，在`include/linux/sys.h`中。增加实验要求的系统调用，需要在这个函数表中增加两个函数引用——`sys_iam`和`sys_whoami`。 该函数在`sys_call_table`数组中的位置必须和`__NR_xxxxxx`的值对应上。

```
72 extern int sys_setregid();
73
74 fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
75 sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
76 sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
77 sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
78 sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
79 sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
80 sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
81 sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
82 sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
83 sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
84 sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
85 sys_getpgrp, sys_setsid, sys_sigaction, sys_sgetmask, sys_ssetmask,
86 sys_setreuid,sys_setregid, sys_iam, sys_whoami };
```

C/ObjC Header Tab Width: 8 Ln 86, Col 28

同时还要仿照此文件中前面各个系统调用的写法，加上两个新系统调用，所作修改如下。

```
71 extern int sys_setreuid();
72 extern int sys_setregid();
73 extern int sys_iam();
74 extern int sys_whoami();
75
76 fn_ptr sys_call_table[] = { sys
77 sys_write, sys_open, sys_close,
78 sys_unlink, sys_execve, sys_chd
```

4. 实现`sys_iam()`和`sys_whoami()`

在`kernel`目录下新建`who.c`实现两个系统调用，如下。

```c
#include <asm/segment.h>
#include <errno.h>
#include <string.h>

char _myname[24];       // 23 characters + \0

int sys_iam(const char* name) {
    char temp[30];
    int i = 0;
    // get string
    for (i = 0; i < 30; i++) {
        temp[i] = get_fs_byte(name + i);
        if (temp[i] == '\0') {
            break;
        }
    }
    int len = strlen(temp);
```

```
    if (len > 23) {          // too long
        return -(EINVAL);    // error
    }
    strcpy(_myname, temp);
    return len;
}

int sys_whoami(char* name, unsigned int size) {
    int len = strlen(_myname) + 1;      // \0 at the end of the string

    if (size < len) {
        return -(EINVAL);
    }
    // save string
    int i = 0;
    for (i = 0; i < len; i++) {
        put_fs_byte(_myname[i], (name + i));
    }

    return len - 1;
}
```

5. 修改Makefile

按照实验手册中的提示修改，时who.c能正确编译、加载，不再赘述。

6. 修改Linux-0.11的文件系统

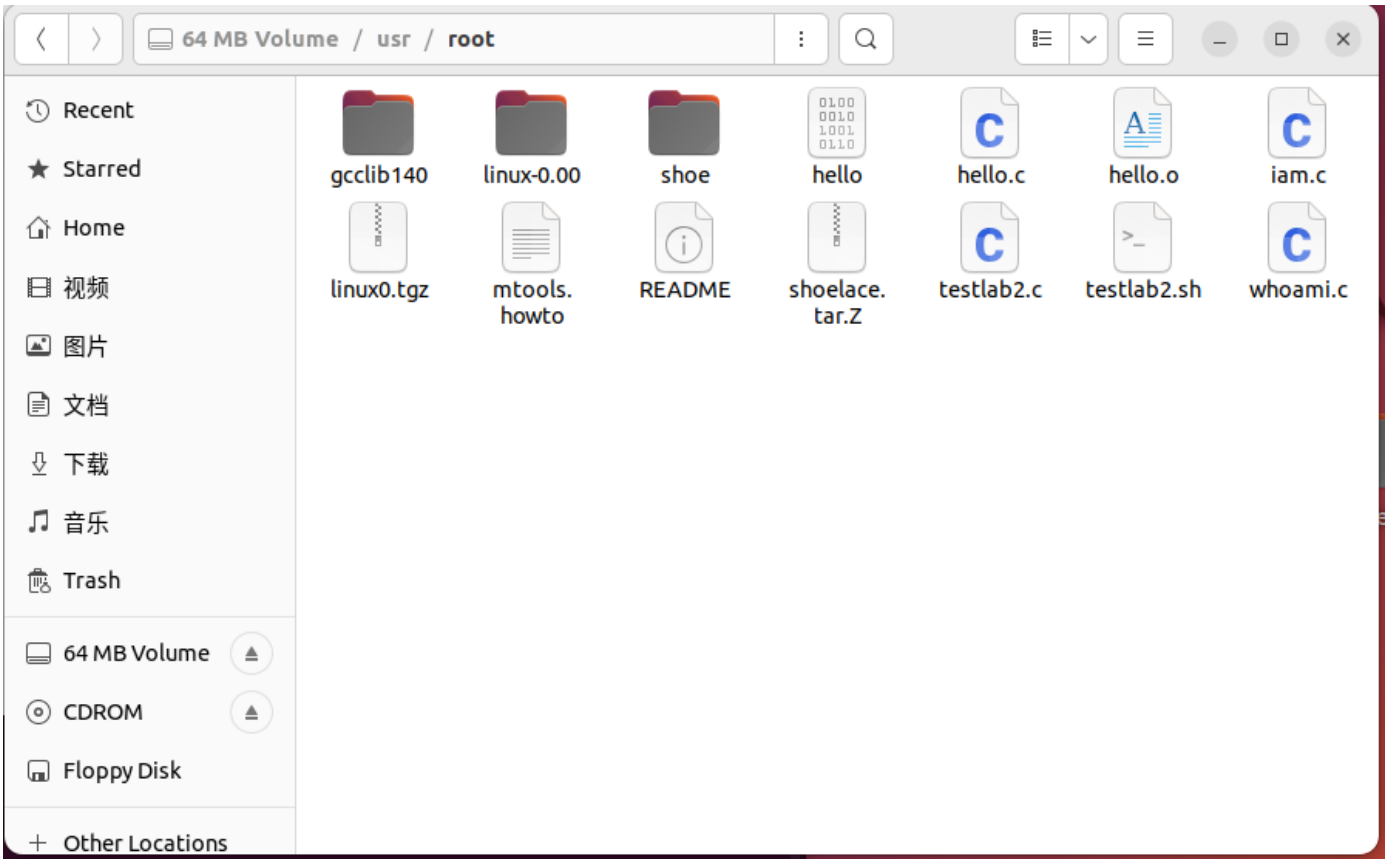
按照课程网站上对Linux-0.11的文件系统的说明，用`mount_hdc`修改文件，首先，编写调用程序`iam.c`和`whoami.c`，内容分别如下。

```
#include <errno.h>
#define __LIBRARY__
#include <unistd.h>
#include <stdio.h>

_syscall1(int, iam, const char*, name);

int main(int argc, char *argv[]) {
    iam(argv[1]);
    return 0;
}
```

```
#include <errno.h>
#define __LIBRARY__
#include <unistd.h>
```

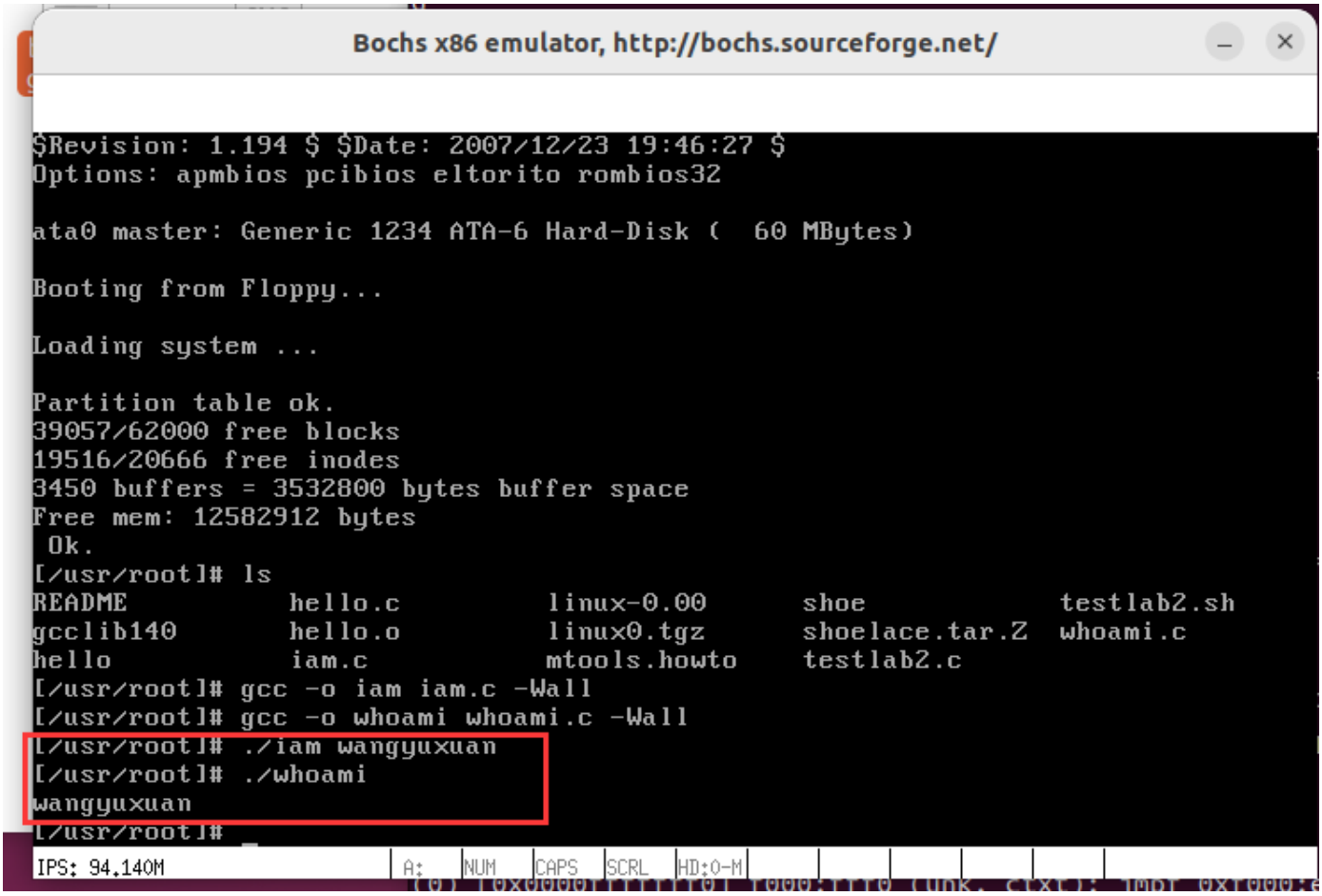
```c
#include <stdio.h>

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

int main(int argc, char *argv[]) {
    char name[30] = {0};

    // 调用库函数 API
    whoami(name, 30);
    printf("%s\n", name);

    return 0;
}
```

之后，将两个测试也文件导入`usr/root`目录，修改后的文件镜像如下。

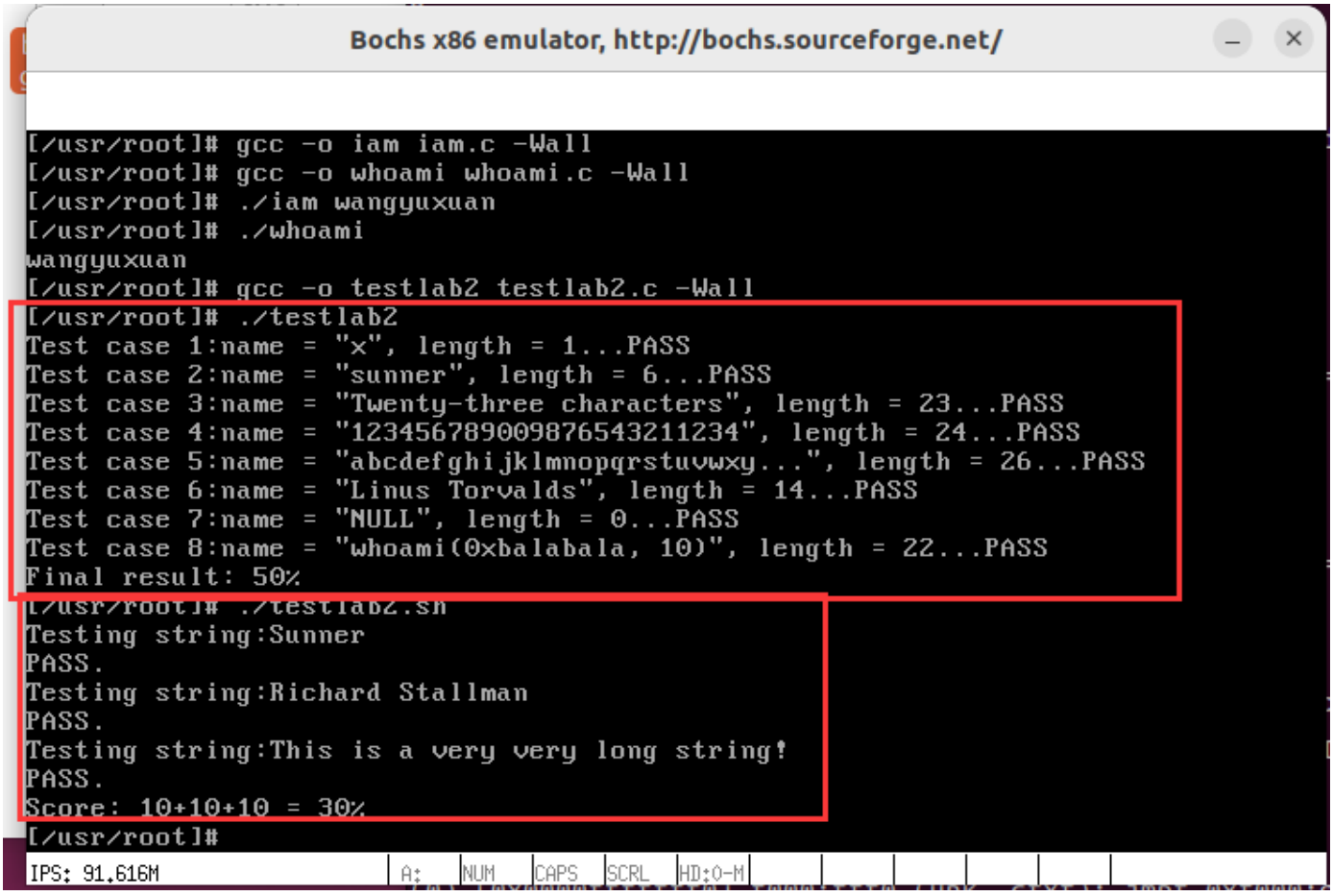


7. 运行结果

两个调用程序的运行结果如下，能够正常运行。



测试程序运行结果如下，结果满分。



8. 问题回答

8.1 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?

三个。对应于`include/unistd.h`中的如下宏定义内容：

```
#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; \
}

#endif /* __LIBRARY__ */
```

8.2 你能想出办法来扩大这个限制吗?

将多个参数集合为结构体，传递的参数为其在用户态地址空间的首地址（以及大小界限等），通过间接寻址方式依次读取参数。

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

1. 修改 `include/unistd.h` 中的宏定义，添加系统调用的宏编号，如 `#define __NR_foo XX` 。
2. 修改 `kernel/system_call.s` 中系统调用总数 `nr_system_calls` 。
3. 修改 `include/linux/sys.h` ，引用增加的系统调用，如 `extern int sys_foo();` ，此外， 在中断处理函数所调用的函数表 `sys_call_table[]` 中增加两个函数引用，其位置与 `__NR_foo XX` 的值对应。
4. 在 `kernel` 下编写 `foo.c` 文件实现该调用。
5. 修改 `kernel/Makefile` ，添加 OBJS 变量后面的依赖文件 foo.o以及依赖 foo.s 和 foo.o 的生成规则。
6. 系统调用用户界面要添加 `#define __LIBRARY__ #include <unistd.h>` 和 `_syscallN` 宏展开系统调用，提供用户态的系统调用接口。