Lab4 系统调用

一、实验目的

1. 建立对系统调用接口的深入认识
2. 掌握系统调用的基本过程
3. 能完成系统调用的全面控制
4. 为后续实验做准备

二、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。

2.1. `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。

2.2. `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。

2.3. 测试程序

运行添加过新系统调用的 `Linux 0.11`，在其环境下编写两个测试程序 `iam.c` 和 `whoami.c`。最终的运行结果是：

```
./iam 2022110829YangMingda          # 测试程序
./whoami                            # 测试程序
2022110829YangMingda                # 运行结果
```

三、实验过程

操作系统实现系统调用的基本过程是：

1. 应用程序调用库函数（ `API` ）；
2. `API` 将系统调用号存入 `EAX` ，然后通过中断调用使系统进入内核态；
3. 内核中的中断处理函数根据系统调用号，调用对应的内核函数（系统调用）；
4. 系统调用完成相应功能，将返回值存入 `EAX` ，返回到中断处理函数；
5. 中断处理函数返回到 `API` 中；
6. `API` 将 `EAX` 返回给应用程序。

3.1. 添加 `__NR_whoami` 宏和 `__NR_iam` 宏

3.1.1. 应用程序如何调用系统调用（原理分析）

在通常情况下，调用系统调用和调用一个普通的自定义函数在代码上并没有什么区别， 但调用后发生的事情有很大不同。 调用自定义函数是通过 `call` 指令直接跳转到该函数的地址，继续运行。 而调用系统调用，是调用系统库中为该系统调用编写的一个接口函数， 叫 `API` （Application Programming Interface）。 `API` 并不能完成系统调用的真正功能，它要做的是去调用真正的系统调用，过程是：

- 把系统调用的编号存入 `EAX`
- 把函数参数存入其它通用寄存器
- 触发 `0x80` 号中断（ `int 0x80` ）

`0.11` 的 `lib` 目录下有一些已经实现的 `API` 。 Linus编写它们的原因是在内核加载完毕后，会切换到用户模式下，做一些初始化工作，然后启动 `shell` 。 而用户模式下的很多工作需要依赖一些系统调用才能完成，因此在内核中实现了这些系统调用的 `API` 。 我们不妨看看 `lib/close.c` ，研究一下 `close()` 的 `API` ：

```
#define __LIBRARY__
#include <unistd.h>
_syscall1(int,close,int,fd)
```

其中 `_syscall1` 是一个宏，在 `include/unistd.h` 中定义。将 `_syscall1(int,close,int,fd)` 进行宏展开，可以得到：

```
int close(int fd)
{
	long __res;
	__asm__ volatile ("int $0x80"
		: "=a" (__res)
		: "0" (__NR_close),"b" ((long)(fd)));
	if (__res >= 0)
		return (int) __res;
	errno = -__res;
	return -1;
}
```

这就是 API 的定义。 它先将宏 __NR_close 存入 EAX ，将参数 fd 存入 EBX ，然后进行 0x80 中断调用。 调用返回后，从 EAX 取出返回值，存入 __res ，再通过对 __res 的判断决定传给 API 的调用者什么样的返回值。 其中 __NR_close 就是系统调用的编号，在 include/unistd.h 中定义：

```
#define __NR_close    6
```

所以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 。而在应用程序中，要有：

```
#define __LIBRARY__                                    /* 有它，_syscall1等才有效。详见
unistd.h */
#include <unistd.h>                                    /* 有它，编译器才能获知自定义的系统调用的
编号*/
_syscall1(int, iam, const char*, name);          /* iam()在用户空间的接口函数 */
_syscall2(int, whoami,char*,name,unsigned int,size);     /* whoami()在用户空间的接口
函数 */
```

在 0.11 环境下编译 C 程序，包含的头文件都在 /usr/include 目录下。 该目录下的 unistd.h 是标准头文件（它和 0.11 源码树中的 unistd.h 并不是同一个文件，虽然内容可能相同）， 没有 __NR_whoami 和 __NR_iam 两个宏，需要手工加上它们，也可以直接从修改过的 0.11 源码树中拷贝新的 unistd.h 过来。

3.1.2. 修改unistd.h文件（实验操作）

由于linux-0.11的unistd.h文件中没有 __NR_whoami 和 __NR_iam 两个宏，所以要手动添加。

1. 挂载文件系统镜像

在oslab目录下，运行命令切换到挂载的文件系统hdc目录，即Linux-0.11使用的文件系统。

```
sudo ./mount-hdc
```

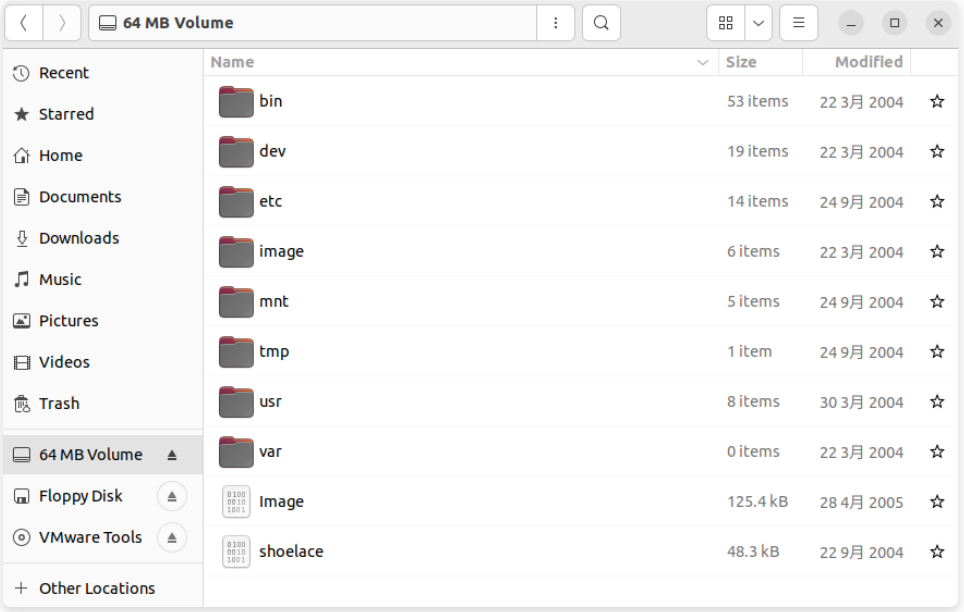

图3-1-1 linux-0.11文件系统

2. 修改unistd.h文件

在 Linux-0.11 文件系统下，进入 usr/include 目录，在 unistd.h 头文件中添加 __NR_whoami 和 __NR_iam 两个宏。

```
// ...
#define __NR_ssetmask   69
#define __NR_setreuid   70
#define __NR_setregid   71
// 添加__NR_whoami和__NR_iam两个宏
#define __NR_iam        72
#define __NR_whoami     73
```

3. 卸载文件系统

在 oslab 目录下执行以下命令，进行文件系统的卸载。

```
sudo umount hdc
```

3.2. 更改系统调用总数

3.2.1. 从 int 0x80 进入内核函数(上)（原理分析）

int 0x80 触发后，接下来就是内核的中断处理了。 先了解一下 0.11 处理 0x80 号中断的过程。

在内核初始化时，主函数（在 init/main.c 中，Linux 实验环境下是 main()， Windows 下因编译器兼容性问题被换名为 start() ）调用了 sched_init() 初始化函数：

```
void main(void)
{
    // ……
    time_init();
    sched_init();
    buffer_init(buffer_memory_end);
    // ……
}
```

sched_init() 在 kernel/sched.c 中定义为：

```
void sched_init(void)
{
    // ……
    set_system_gate(0x80,&system_call);
}
```

set_system_gate 是个宏，在 include/asm/system.h 中定义为：

```
#define set_system_gate(n,addr) \
    _set_gate(&idt[n],15,3,addr)
```

_set_gate 的定义是：

```
#define _set_gate(gate_addr,type,dpl,addr) \
__asm__ ("movw %%dx,%%ax\n\t" \
    "movw %0,%%dx\n\t" \
    "movl %%eax,%1\n\t" \
    "movl %%edx,%2" \
    : \
    : "i" ((short) (0x8000+(dpl<<13)+(type<<8))), \
    "o" (*((char *) (gate_addr))), \
    "o" (*(4+(char *) (gate_addr))), \
    "d" ((char *) (addr)),"a" (0x00080000))
```

虽然看起来挺麻烦，但实际上很简单，就是填写 `IDT` （中断描述符表），将 `system_call` 函数地址写到 `0x80` 对应的中断描述符中，也就是在中断 `0x80` 发生后，自动调用函数 `system_call` 。`system_call` 函数分析如下。

```
# ……
nr_system_calls = 72                # 这是系统调用总数。如果增删了系统调用，必须做相应修改
# ……
.globl system_call
.align 2
system_call:
    cmpl $nr_system_calls-1,%eax    # 检查系统调用编号是否在合法范围内
    ja bad_sys_call
    push %ds
    push %es
    push %fs
    pushl %edx
    pushl %ecx
    pushl %ebx                      # push %ebx,%ecx,%edx，是传递给系统调用的参数
    movl $0x10,%edx                 # 让ds,es指向GDT，内核地址空间
    mov %dx,%ds
    mov %dx,%es
    movl $0x17,%edx                 # 让fs指向LDT，用户地址空间
    mov %dx,%fs
    call sys_call_table(,%eax,4)
    pushl %eax
    movl current,%eax
    cmpl $0,state(%eax)
    jne reschedule
    cmpl $0,counter(%eax)
    je reschedule
```

`system_call` 用 `.globl` 修饰为其他函数可见。 `call sys_call_table(,%eax,4)` 之前是一些压栈保护，修改段选择子为内核段， `call sys_call_table(,%eax,4)` 后是看是否需要重新调度。

3.2.2. 修改 `system_call.s` 文件（实验操作）

1. 计算系统调用总数

根据上一节，观察 `unistd.h` ，发现头文件中添加 `__NR_whoami` 和 `__NR_iam` 两个宏后，宏的序号从 `0` 依次到 `73` ，总共 `74` 个宏。所以系统调用总数为 `74` 。

2. 修改系统调用总数

通过分析 `system_call` 函数，发现系统调用函数在 `kernel/system_call.s` 中定义：

```
nr_system_calls = 72                    # 这是系统调用总数。如果增删了系统调用，必须做相应修改
```

所以要修改系统调用总数，修改`system_call.s`代码为：

```
nr_system_calls = 74                    # 这是系统调用总数。如果增删了系统调用，必须做相应修改
```

```
48 state   = 0              # these are offsets into the task-struct.
49 counter = 4
50 priority = 8
51 signal  = 12
52 sigaction = 16           # MUST be 16 (=len of sigaction)
53 blocked = (33*16)
54
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 = 72
62 nr_system_calls = 74
63
64 /*
65  * Ok, I get parallel printer interrupts while using the floppy for some
66  * strange reason. Urgel. Now I just ignore them.
67  */
68 .globl system_call,sys_fork,timer_interrupt,sys_execve
69 .globl hd_interrupt,floppy_interrupt,parallel_interrupt
70 .globl device_not_available, coprocessor_error
71
72 .align 2
73 bad_sys_call:
74         movl $-1,%eax
75         iret
76 .align 2
```

图3-2-1 在system_call.s中修改系统调用总数

3.3. 添加系统调用`sys_iam`和`sys_whoami`

3.3.1. 从`int 0x80`进入内核函数(下)（原理分析）

`system_call`函数中的`call sys_call_table(,%eax,4)`根据汇编寻址方法实际为：

```
call sys_call_table + 4 * %eax    # 其中eax中放的是系统调用号，即__NR_xxxxxx
```

`sys_call_table`是函数指针数组的起始地址，它定义在`include/linux/sys.h`中：

```
fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read, // ……
```

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`。当然该函数在`sys_call_table`数组中的位置必须和`__NR_xxxxxx`的值对应上。同时还要仿照此文件中前面各个系统调用的写法，加上下面代码，以免编译出错。

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

3.3.2. 修改`sys.h`文件

1. 在函数表中增加函数引用

在`include/linux/sys.h`中的函数表中增加`sys_iam`和`sys_whoami`两个函数引用：

```
// ...
extern int sys_ssetmask();
extern int sys_setreuid();
extern int sys_setregid();
// 添加函数引用
extern int sys_iam();
extern int sys_whoami();
```

2. 扩展`sys_call_table`数组

```
fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read, // ……
                            sys_iam, sys_whoami // 添加函数引用名
```

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

图3-3-1 在sys.h中添加系统调用

3.4. 实现`sys_iam()`和`sys_whoami()`函数

3.4.1. 原理分析

添加系统调用的最后一步，是在内核中实现函数`sys_iam()`和`sys_whoami()`。

每个系统调用都有一个`sys_xxxxxx()`与之对应，它们都是我们学习和模仿的好对象。比如在`fs/open.c`中的`sys_close(int fd)`：

```
int sys_close(unsigned int fd)
{
    // ……
    return (0);
}
```

它没有什么特别的，都是实实在在地做`close()`该做的事情。所以只要自己创建一个文件：`kernel/who.c`，然后实现两个函数就万事大吉了。

3.4.2. `sys_iam()`分析

`sys_iam()`函数具体代码及其详细分析如下所示。

```
int sys_iam(const char * name) {
    char tmp[26];                           // 声明一个字符数组 tmp，大小为 26
    short break_flag = 0, i = 0;            // 声明短整型整数 break_flag 为0，i 为0
    for (i = 0; i < 26; ++i) {              // 进入循环
        tmp[i] = get_fs_byte(name + i);     // 从用户空间读取一个字节并保存到 tmp[i]
        if (tmp[i] == '\0') {               // 如果遇到字符串结束符（'\0'）
```

```
            break_flag = 1;                 // 设置 break_flag 标志为 1，表示字符串结束
            break;                          // 跳出循环
        }
    }
    if (!break_flag || i > 23) {            // 如果没有遇到字符串结束符或字符串长度大于23
        return -(EINVAL);                   // 返回无效参数错误，EINVAL 错误码
    }
    char* dest = username;                  // 定义一个指向全局 username 数组的指针
    strcpy(dest, tmp);                      // 将 tmp 中的内容复制到 username 中
    return i;                               // 返回读取的字符数（即字符串长度）
}
```

3.4.3. `sys_whoami()`分析

`sys_whoami()`函数具体代码及其详细分析如下所示。

```
int sys_whoami(char* name, unsigned int size) {
    short length = strlen(username);            // 计算全局 username 字符串的长度
    if (length > size) {                        // 如果 username 的长度大于传入的size 参数
        return -(EINVAL);                       // 返回无效参数错误
    }
    short i = 0;                                // 声明短整型整数 i 为0
    for (i; i < size; ++i) {                    // 进入循环
        put_fs_byte(username[i], name + i);     // 将 username 中的字符写入用户空间
        if (username[i] == '\0') {              // 如果遇到字符串结束符（'\0'）
            break;                              // 跳出循环
        }
    }
    return i;                                   // 返回写入的字符数（即字符串长度）
}
```

3.4.4. 完整实现代码

在`kernel`目录下创建`who.c`文件，编写完整代码为：

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

char username[24];

int sys_iam(const char * name) {
    char tmp[26];
    short break_flag = 0, i = 0;
    for (i = 0; i < 26; ++i) {
        tmp[i] = get_fs_byte(name + i);
        if (tmp[i] == '\0') {
            break_flag = 1;
            break;
        }
    }
```

```
    if (!break_flag || i > 23) {
        return -(EINVAL);
    }
    char* dest = username;
    strcpy(dest, tmp);
    return i;
}

int sys_whoami(char* name, unsigned int size) {
    short length = strlen(username);
    if (length > size) {
        return -(EINVAL);
    }
    short i = 0;
    for (i; i < size; ++i) {
        put_fs_byte(username[i], name + i);
        if (username[i] == '\0') {
            break;
        }
    }
    return i;
}
```

3.5. 修改`Makefile`文件

3.5.1. 原理分析

要想让我们添加的`kernel/who.c`可以和其它Linux代码编译链接到一起，必须要修改`Makefile`文件。`Makefile`里记录的是所有源程序文件的编译、链接规则，《注释》`3.6`节有简略介绍。我们之所以简单地运行`make`就可以编译整个代码树，是因为`make`完全按照`Makefile`里的指示工作。

`Makefile`在代码树中有很多，分别负责不同模块的编译工作。我们要修改的是`kernel/Makefile`。

3.5.2. 修改`Makefile`

1. 修改`OBJS`

在`kernel/Makefile`中修改`OBJS`部分从：

```
OBJS  = sched.o system_call.o traps.o asm.o fork.o \
        panic.o printk.o vsprintf.o sys.o exit.o \
        signal.o mktime.o
```

改为：

```
OBJS  = sched.o system_call.o traps.o asm.o fork.o \
        panic.o printk.o vsprintf.o sys.o exit.o \
        signal.o mktime.o who.o
```

```
.c.s:
        @$(CC) $(CFLAGS) \
        -S -o $*.s $<
.s.o:
        @$(AS) -o $*.o $<
.c.o:
        @$(CC) $(CFLAGS) \
        -c -o $*.o $<

OBJS  = sched.o system_call.o traps.o asm.o fork.o \
        panic.o printk.o vsprintf.o sys.o exit.o \
        signal.o mktime.o who.o

kernel.o: $(OBJS)
        @$(LD) $(LDFLAGS) -o kernel.o $(OBJS)
        @sync
```

图3-5-1 修改OBJS

2. 修改`Dependencies`

在`kernel/Makefile`中修改`Dependencies`部分从：

```
### Dependencies:
exit.s exit.o: exit.c ../include/errno.h ../include/signal.h \
  ../include/sys/types.h ../include/sys/wait.h ../include/linux/sched.h \
  ../include/linux/head.h ../include/linux/fs.h ../include/linux/mm.h \
  ../include/linux/kernel.h ../include/linux/tty.h ../include/termios.h \
  ../include/asm/segment.h
### ...
vsprintf.s vsprintf.o: vsprintf.c ../include/stdarg.h ../include/string.h
```

改为：

```
### Dependencies:
exit.s exit.o: exit.c ../include/errno.h ../include/signal.h \
  ../include/sys/types.h ../include/sys/wait.h ../include/linux/sched.h \
  ../include/linux/head.h ../include/linux/fs.h ../include/linux/mm.h \
  ../include/linux/kernel.h ../include/linux/tty.h ../include/termios.h \
  ../include/asm/segment.h
### ...
vsprintf.s vsprintf.o: vsprintf.c ../include/stdarg.h ../include/string.h
who.s who.o: who.c ../include/linux/kernel.h ../include/unistd.h
```

```
### Dependencies:
exit.s exit.o: exit.c ../include/errno.h ../include/signal.h \
  ../include/sys/types.h ../include/sys/wait.h ../include/linux/sched.h \
  ../include/linux/head.h ../include/linux/fs.h ../include/linux/mm.h \
  ../include/linux/kernel.h ../include/linux/tty.h ../include/termios.h \
  ../include/asm/segment.h
fork.s fork.o: fork.c ../include/errno.h ../include/linux/sched.h \
  ../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \
  ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h \
  ../include/asm/segment.h ../include/asm/system.h
mktime.s mktime.o: mktime.c ../include/time.h
panic.s panic.o: panic.c ../include/linux/kernel.h ../include/linux/sched.h \
  ../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \
  ../include/linux/mm.h ../include/signal.h
printk.s printk.o: printk.c ../include/stdarg.h ../include/stddef.h \
  ../include/linux/kernel.h
sched.s sched.o: sched.c ../include/linux/sched.h ../include/linux/head.h \
  ../include/linux/fs.h ../include/sys/types.h ../include/linux/mm.h \
  ../include/signal.h ../include/linux/kernel.h ../include/linux/sys.h \
  ../include/linux/fdreg.h ../include/asm/system.h ../include/asm/io.h \
  ../include/asm/segment.h
signal.s signal.o: signal.c ../include/linux/sched.h ../include/linux/head.h \
  ../include/linux/fs.h ../include/sys/types.h ../include/linux/mm.h \
  ../include/signal.h ../include/linux/kernel.h ../include/asm/segment.h
sys.s sys.o: sys.c ../include/errno.h ../include/linux/sched.h \
  ../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \
  ../include/linux/mm.h ../include/signal.h ../include/linux/tty.h \
  ../include/termios.h ../include/linux/kernel.h ../include/asm/segment.h \
  ../include/sys/times.h ../include/sys/utsname.h
traps.s traps.o: traps.c ../include/string.h ../include/linux/head.h \
  ../include/linux/sched.h ../include/linux/fs.h ../include/sys/types.h \
  ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h \
  ../include/asm/system.h ../include/asm/segment.h ../include/asm/io.h
vsprintf.s vsprintf.o: vsprintf.c ../include/stdarg.h ../include/string.h
who.s who.o: who.c ../include/linux/kernel.h ../include/unistd.h
```

图3-5-2 修改Dependencies

`Makefile` 修改后，和往常一样 `make all` 就能自动把 `who.c` 加入到内核中了。 如果编译时提示 `who.c` 有错误，就说明修改生效了。 所以，有意或无意地制造一两个错误也不完全是坏事，至少能证明 `Makefile` 是对的。

3.6. 编译`Linux-0.11`源码

经过上述代码修改之后，切换到`Linux-0.11`目录下执行以下命令，生成`Image`镜像文件。

```
make clean
make all
```

```
make[1]: Leaving directory '/home/guojun/oslab/linux-0.11/kernel/blk_drv'
make[1]: Entering directory '/home/guojun/oslab/linux-0.11/kernel/chr_drv'
sync
make[1]: Leaving directory '/home/guojun/oslab/linux-0.11/kernel/chr_drv'
make[1]: Entering directory '/home/guojun/oslab/linux-0.11/kernel/math'
make[1]: Leaving directory '/home/guojun/oslab/linux-0.11/kernel/math'
make[1]: Entering directory '/home/guojun/oslab/linux-0.11/lib'
make[1]: Leaving directory '/home/guojun/oslab/linux-0.11/lib'
1+0 records in
1+0 records out
512 bytes copied, 0.000498899 s, 1.0 MB/s
0+1 records in
0+1 records out
311 bytes copied, 0.000202214 s, 1.5 MB/s
309+1 records in
309+1 records out
158497 bytes (158 kB, 155 KiB) copied, 0.00114823 s, 138 MB/s
2+0 records in
2+0 records out
2 bytes copied, 7.922e-05 s, 25.2 kB/s
guojun@guojunos:~/oslab/linux-0.11$
```

图3-6-1 编译Linux-0.11源码

3.7. 添加程序验证系统调用

在运行的`Linux-0.11`上编写`iam.c`和`whoami.c`文件，添加程序验证系统调用。

3.7.1. `iam.c`分析

`iam.c`函数具体代码及其详细分析如下所示。

```
#include <errno.h>                          // 包含定义了错误码的头文件
#define __LIBRARY__                         // 用于指示这是一个系统调用的程序（内核与用户之间的接口）
#include <unistd.h>                         // 包含标准的 Unix 系统调用接口
#include <stdio.h>                          // 包含标准输入输出的库，用于输出打印等功能

// 定义了一个宏，用于创建一个系统调用接口
// _syscall1：是一个宏，用于定义系统调用，表示该系统调用有一个参数（第一个参数是 name）
// int: 表示系统调用返回的类型是 int，即返回一个整数（可能是系统调用的返回值）
// iam: 系统调用的名称,与内核中的 sys_iam 系统调用对应
// const char*: 表示系统调用的参数类型是一个指向常量字符的指针
// name: 这是传递给系统调用的参数，表示用户名
_syscall1(int, iam, const char*, name);
// int main(int argc, char **argv): 这是标准的 main 函数入口，它有两个参数：
// argc: 表示命令行参数的个数
// argv: 是一个字符串数组，其中包含传递给程序的命令行参数
int main(int argc,char ** argv)
{
    iam(argv[1]);       // 调用自定义的 iam 系统调用，传入命令行参数中的第一个字符串作为参数
    return 0;           // 正常退出程序
}
```

3.7.2. `whoami.c`分析

`whoami.c`函数具体代码及其详细分析如下所示。

```
#include <errno.h>                          // 包含定义错误码的头文件
#define __LIBRARY__                         // 用于指示这是一个系统调用的程序（内核与用户之间的接口）
#include <unistd.h>                         // 包含标准的 Unix 系统调用接口

// 定义了一个宏，用于创建一个用户空间的接口
// _syscall2: 是一个宏，用来定义带有两个参数的系统调用。系统调用 whoami 接受两个参数：char* name 和 unsigned int size，返回一个整数（通常是系统调用成功时返回的结果，或者是出错时的错误码）
// int: 系统调用的返回类型，表示返回一个整数值，通常用于返回系统调用执行的结果或错误码
// whoami: 系统调用的名字，它与内核中的 sys_whoami 系统调用相对应，用户程序通过调用这个名字来与内核进行交互
// char* name: 系统调用的第一个参数，表示一个字符指针，用来存储获取到的用户名
// unsigned int size: 系统调用的第二个参数，表示用户提供的缓冲区的大小，用来确保目标缓冲区足够大以存储用户名
_syscall2(int, whoami,char*,name,unsigned int,size);

int main()
{
    char s[30];     // 声明一个字符数组 s，大小为 30，用来存储从内核返回的用户名
    whoami(s,30);   // 调用定义的 whoami 系统调用，将结果存储在字符数组 s 中，最多读取 30 字节
    printf("%s",s); // 打印存储在 s 中的字符串，即用户名
    return 0;       // 正常退出程序
```

```
}
```

3.7.3. 实验操作

1. 挂载文件系统镜像

在`oslab`目录下，运行命令切换到挂载的文件系统`hdc`目录，即`Linux-0.11`使用的文件系统。

```
sudo ./mount-hdc
```

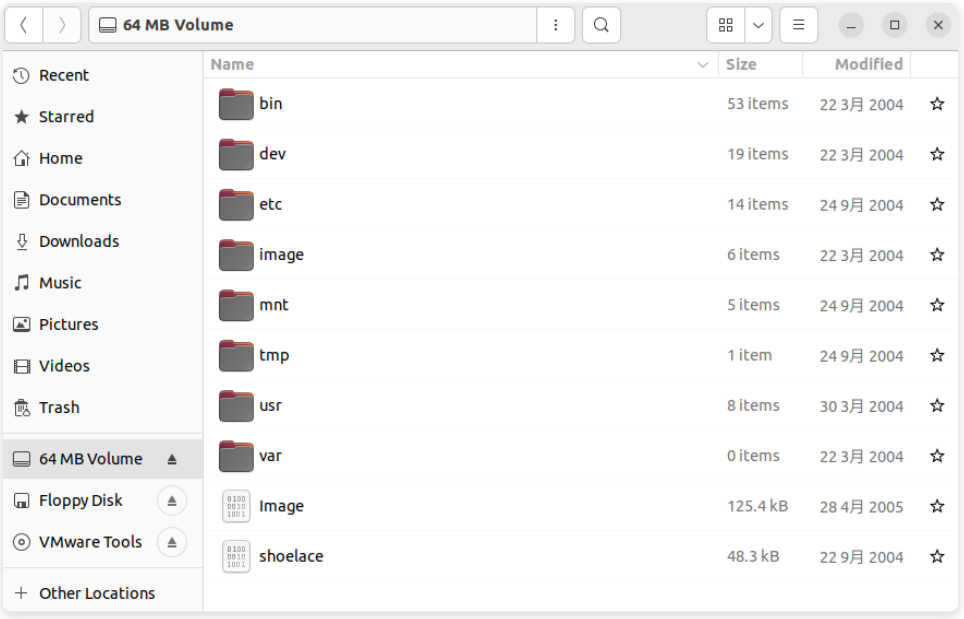

图3-7-1 linux-0.11文件系统

2. 创建并编写`iam.c`和`whoami.c`

在`~/oslab/hdc/usr/root`目录下创建`iam.c`和`whoami.c`两个文件。

```
touch iam.c
touch whoami.c
```

接着将上述分析的`iam.c`和`whoami.c`两段代码复制到文件中。

3.8. 测试程序

在完成上面所有的代码编写，接下来就可以进行程序的测试环境。

1. 在`~/oslab`目录下运行以下命令，在终端输入`c`后进入`Bochs`界面。

```
./dbg-bochs
```

```
guojun@guojunos:~/oslab$ ./dbg-bochs
========================================================================
                        Bochs x86 Emulator 2.8
            Built from GitHub snapshot on March 10, 2024
               Timestamp: Sun Mar 10 08:00:00 CET 2024
                    Compiled by GuoJun, Dec 3 2024
========================================================================
00000000000i[      ] BXSHARE is set to '/usr/local/bochs28/share/bochs'
00000000000i[      ] reading configuration from ./bochs/linux-0.11.bxrc
00000000000i[      ] installing x module as the Bochs GUI
00000000000i[      ] using log file ./bochsout.txt
Bochs internal debugger, type 'help' for help or 'c' to continue
Switching to CPU0
Next at t=0
(0) [0x0000fffffff0] f000:fff0 (unk. ctxt): jmpf 0xf000:e05b          ; ea5be000f0
<bochs:1> c
```

图3-8-1 终端输入

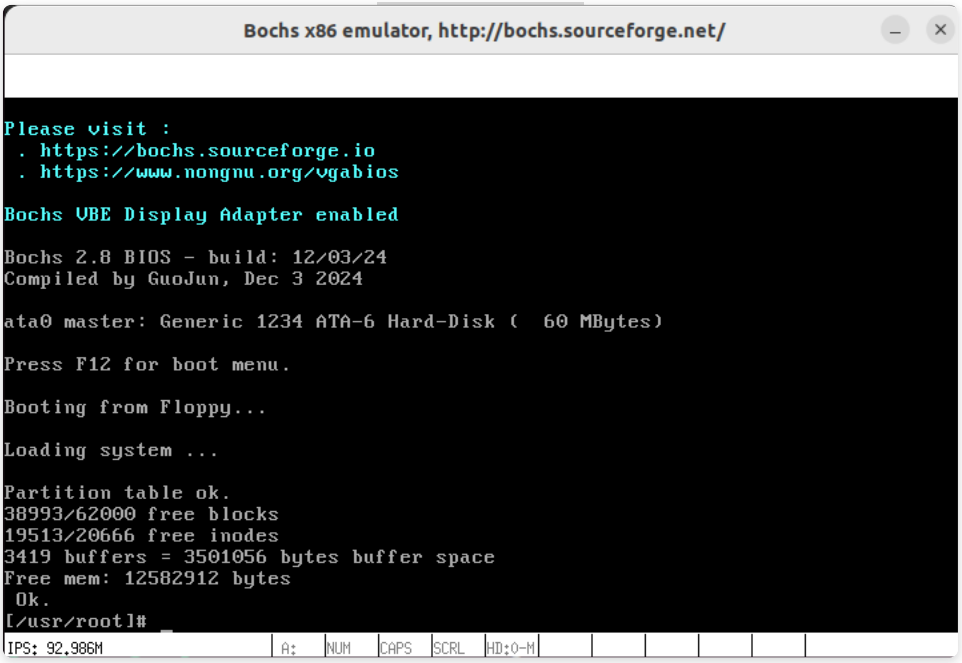

图3-8-2 进入Bochs界面

2. 编译`iam.c`和`whoami.c`

在`Bochs`界面下，使用以下命令编译`iam.c`和`whoami.c`。

```
gcc -o iam iam.c -Wall
gcc -o whoami whoami.c -Wall
```

3. 运行`iam`和`whoami`进行测试

在`Bochs`界面下，使用以下命令运行`iam`和`whoami`进行测试，得到正确的结果。

```
./iam 2022110829YangMingda          # 测试程序
./whoami                            # 测试程序
2022110829YangMingda                # 运行结果
```

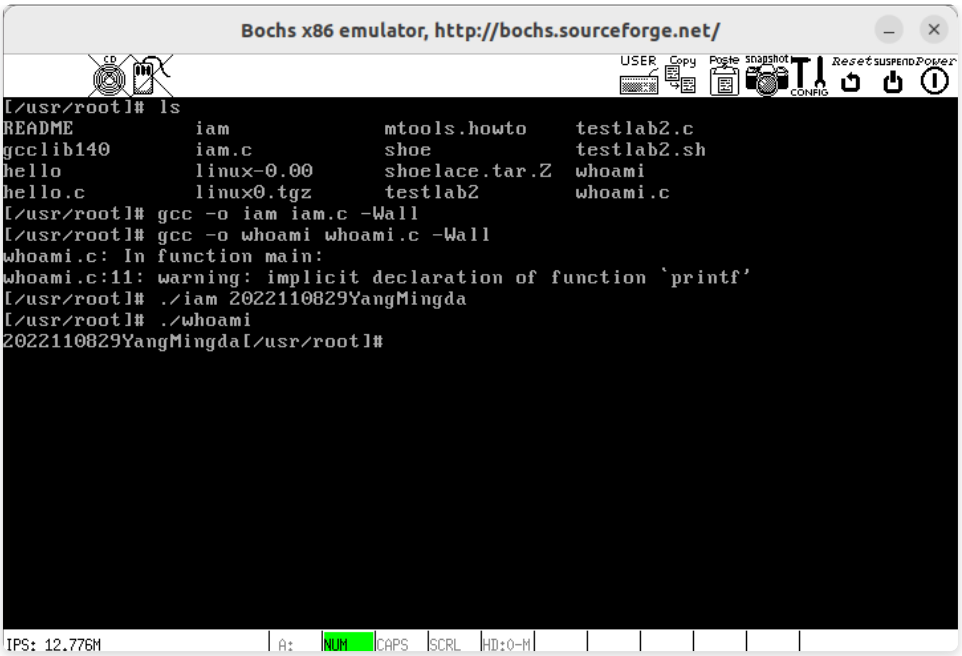

图3-8-3 运行iam和whoami进行测试

4. 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。

```
gcc -o testlab2 testlab2.c -Wall
./testlab2
```

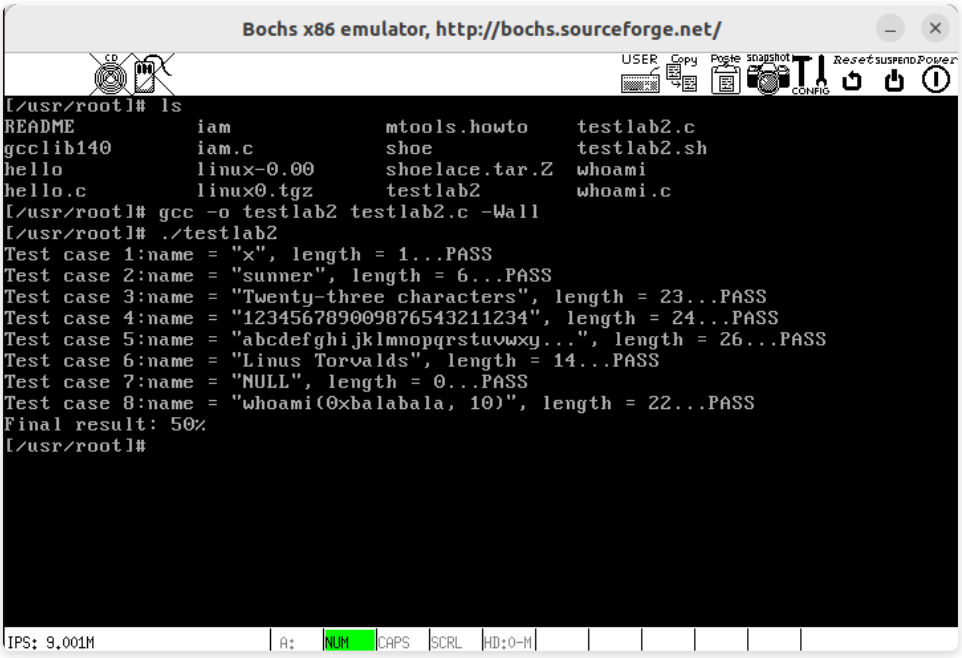

图3-8-4 运行testlab2.c

5. 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。

```
./testlab2.sh
```

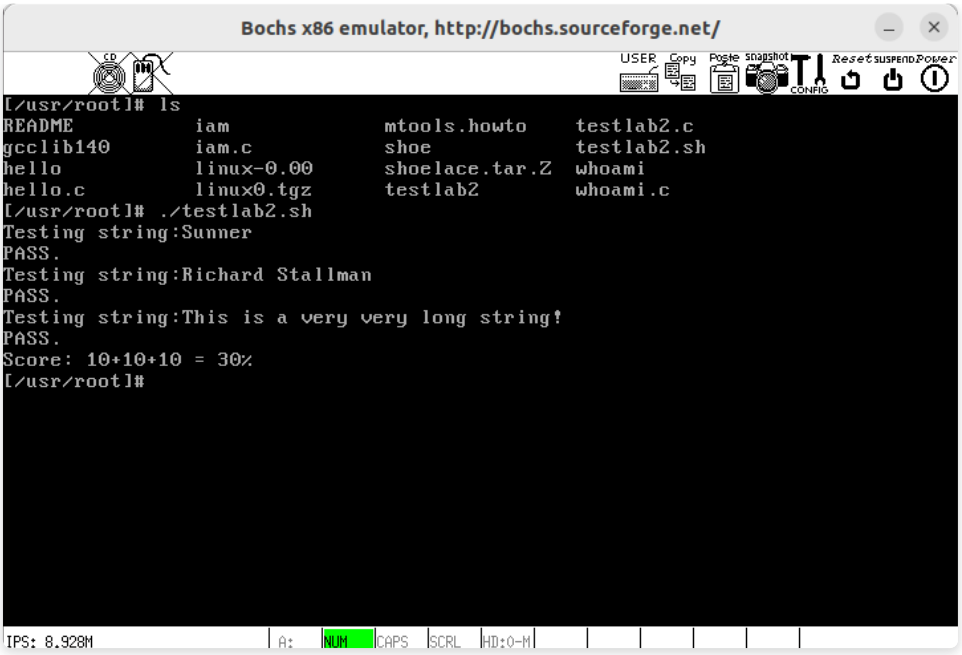

图3-8-5 运行testlab2.sh

四、实验报告

- 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?

最多传递3个参数。通过`unistd.h`中的 `_syscall`，可得知应用函数经过库函数向内核发送了一个`0x80`的中断，然后开始执行系统调用。而`Linux-0.11`使用`ebx、ecx、edx`传递参数，所以从 `Linux 0.11` 现在的机制看，它的系统调用最多能传递3个参数。

- 你能想出办法来扩大这个限制吗?

可以通过使用一个寄存器保存指向进程的用户态栈中的一块内存区域的地址，该内存中保存参数的值。也可以采用间接寻址的办法，将这三个参数全部改成指针，然后将待传递的参数都存放在指针对应的地址处。

- 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 `hdc/usr/include/unistd.h`， 添加`#define __NR_foo num`， `num`为接下来使用的系统调用号
2. 修改 `include/linux/sys.h`，添加`extern rettype sys_foo();`， 在`sys_call_table`数组对应位置加入`sys_foo`
3. 修改 `kernel/system_call.s`，修改 `nr_system_calls = num`（`num`为系统调用总数目）
4. 在 `kernel` 目录中添加 `foo.c` ，如果需要支持内核态与用户态数据交互，则包含`include/asm/segment.h`，其中有 `put_fs_XXX` 和 `get_fs_XXX` 函数
5. 在系统类库中编写 `foo.c` ，实现系统调用`sys_foo()`
6. 修改 `kernel` 的Makefile，将 `foo.c` 与内核其它代码编译链接到一起
7. 添加程序验证系统调用

五、总结

通过本次实验，在`Linux-0.11`下通过添加两个系统调用，并编写两个简单的应用程序测试它们，我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，并且为后续实验做准备。在实验过程中，我还不断学习操作系统`linux0.11`中实现系统调用的基本过程，了解其中的作用和含义。通过本次实验还锻炼了我解决问题的能力。