

操作系统实验四 系统调用

一、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。

二、实验环境

Vmware 17.5.1, Ubuntu 20.04, Bochs 2.4.6

三、实验内容

在 Linux 0.11 上添加两个系统调用 `iam()` 和 `whoami()`，并编写两个简单的应用程序 `iam.c` 和 `whoami.c` 测试它们。

3.1 `iam()`

第一个系统调用是 `iam()`，其原型为：`int iam(const char * name);`，它将字符串参数 `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。

3.2 `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` 中实现此系统调用。

3.3 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`。执行 `iam` 时输入要保存的字符串，执行 `whoami` 时在屏幕输出该字符串。

四、实验过程

4.1 修改 Linux 0.11 源代码文件

4.1.1 修改 `include/unistd.h`

在 `include/unistd.h` 中添加宏定义 `__NR_whoami` 和 `__NR_iam`，它们是系统调用的编号。




```
123 #define __NR_dup2      63
124 #define __NR_getppid   64
125 #define __NR_getpggrp   65
126 #define __NR_setsid     66
127 #define __NR_sigaction   67
128 #define __NR_sgetmask    68
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" \
140         : "=a" (__res) \
141         : "0" (__NR_#name)); \
142     if (__res >= 0) \
143         return (type) __res; \
144     errno = -__res; \
145     return -1; \
146 }
147
```

注意在 Linux 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 Linux 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 Linux 0.11 源码树中拷贝新的 unistd.h 过来。

4.1.2 修改 kernel/system_call.s

在 system_call.s 中将 nr_system_calls 修改为 74，它是系统调用总数。如果增删了系统调用，必须做相应修改。



```
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 = 74
62
63 /*
64  * Ok, I get parallel printer interrupts while using the floppy for some
65  * strange reason. Urgel. Now I just ignore them.
66  */
67 .globl system_call,sys_fork,timer_interrupt,sys_execve
68 .globl hd_interrupt,floppy_interrupt,parallel_interrupt
69 .globl device_not_available, coprocessor_error
70
71 .align 2
72 bad_sys_call:
73     movl $-1,%eax
74     tret
```

system_call 中 call sys_call_table(%eax,4) 使用偏移量寻址并执行系统调用，其中 eax 中放的就是系统调用号，即上面提到的 __NR_xxxxxx，而 sys_call_table 是一个函数指针数组的起始地址，将在下一步进行修改。

4.1.3 修改 include/linux/sys.h

在 sys.h 中添加外部变量 sys_iam(); 和 sys_whoami();，并在系统调用函数表 sys_call_table 中添加 sys_iam 和 sys_whoami，便于 system_call 由偏移量寻址。注意函数在 sys_call_table 数组中的位置必须和 __NR_xxxxxx 的值对应上。

```
sys.h
~/OS/linux011/oslab4/linux-0.11/include/linux

64 extern int sys_dup2();
65 extern int sys_getppid();
66 extern int sys_getpgrp();
67 extern int sys_setsid();
68 extern int sys_sigaction();
69 extern int sys_sgetmask();
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_ssetmask, sys_ssetmask,
88 sys_setreuid, sys_setregid, sys_iam, sys_whoami};
C/C++/ObjC 头文件 制表符宽度: 8 第 1 行, 第 1 列 插入
```

4.2 实现 sys_iam()和 sys_whoami()

在 kernel/ 目录中创建 who.c 文件，编写 sys_iam()和 sys_whoami()的具体内容。定义全局变量 char msg[24]，用于存放要保存的字符串。在用户态和内核态之间传递数据，需要用到 get_fs_byte()和 put_fs_byte()函数，它们位于 asm/segment.h 中，单位为字节。

在 sys_iam()中，首先定义一个空字符数组 tmp[25]，用于临时存放用户态向内核态传递的字符串；len 表示字符串总长度（包括 '\0'）。接着使用 get_fs_byte()函数逐个字节获取用户空间中的数据（即 1 个字符），存入 tmp 数组并使 len 加 1，直到读取到结束字符 '\0' 或 len 超过 24。然后判断字符串长度，若 len 不大于 24，则将 tmp 拷贝给 msg 保存，并返回 len-1；否则返回 -(EINVAL)。

在 sys_whoami()中，首先获取保存的字符串长度 strlen(msg)，然后比较用户地址空间大小 size 和字符串长度 len，若 size 大于 len 则使用 put_fs_byte()函数将字符串逐个字符由内核传递到用户空间中，并返回 len；否则返回 -(EINVAL)。

```
who.c
~/OS/linux011/oslab4/linux-0.11/kernel

1 /*
2  * linux/kernel/who.c
3  *
4  * (C) 2024 Liu Zikang
5  */
6
7 #include <errno.h>
8 #include <string.h>
9 #include <asm/segment.h>
10
11 char msg[24];
12
13 int sys_iam(const char * name)
14 {
15     char tmp[25] = "";
16     int len = 0;
17
18     do {
19         tmp[len] = get_fs_byte(&name[len]);
20     } while(tmp[len++] != '\0' && len < 25);
21
22     if (len > 24) {
23         return -(EINVAL);
24     }
25
26     strcpy(msg, tmp);
27     return len - 1;
28 }
29
30 int sys_whoami(char* name, unsigned int size)
31 {
32     int len = strlen(msg), i = 0;
33
34     if (size < len) {
35         return -(EINVAL);
36     }
37
38     for (; i < len; i++) {
39         put_fs_byte(msg[i], name+i);
40     }
41
42     return len;
43 }
44
45
C 制表符宽度: 8 第 45 行, 第 1 列 插入
```

4.3 修改 MakeFile 文件

为使得添加的 kernel/who.c 可以和其它 Linux 0.11 代码编译链接到一起，需要修改 kernel/MakeFile 文件，在 OBJS 中添加 who.o，在依赖中添加 who.s who.o: who.c ../include/linux/kernel.h ../include/unistd.h。之后在 linux-0.11/目录下执行 make 即可正确编译。



```
22 $(AS) -o $*.o $<
23 .c.o:
24 $(CC) $(CFLAGS) \
25 -c -o $*.o $<
26
27 OBJS = sched.o system_call.o traps.o asm.o fork.o \
28 panic.o printk.o vsprintf.o sys.o exit.o \
29 signal.o mktime.o who.o
30
31 kernel.o: $(OBJS)
32 $(LD) -m elf_i386 -r -o kernel.o $(OBJS)
33 sync
34
35 clean:
36 rm -f core *.o *.a tmp_make keyboard.s
37 for i in *.c;do rm -f `basename $$i .c`.s;done
38 (cd chr_drv; make clean)
39 (cd blk_drv; make clean)
40 (cd math; make clean)
41
42 dep:
43 sed '/\#/# Dependencies/q' < Makefile > tmp_make
44 (for i in *.c;do echo -n 'echo $$i | sed 's,\.C,\.S,' ; \
45 $(CPP) -M $$i;done) >> tmp_make
46 cp tmp_make Makefile
47 (cd chr_drv; make dep)
48 (cd blk_drv; make dep)
49
50 ### Dependencies:
51 who.s who.o: who.c ../include/linux/kernel.h ../include/unistd.h
52 exit.s exit.o: exit.c ../include/errno.h ../include/signal.h \
53 ../include/sys/types.h ../include/sys/wait.h ../include/linux/sched.h \
54 ../include/linux/head.h ../include/linux/fs.h ../include/linux/mm.h \
55 ../include/linux/kernel.h ../include/linux/tty.h ../include/termios.h \
56 ../include/asm/segment.h
```

4.4 运行测试程序

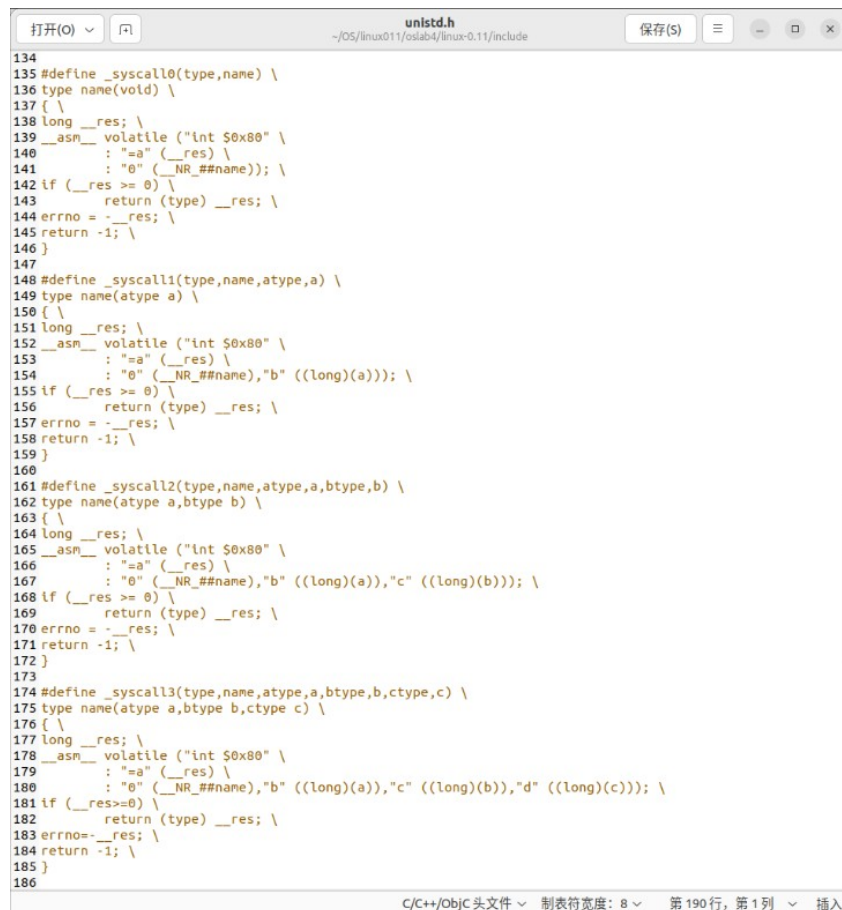
编写测试程序 iam.c 和 whoami.c，需要用到 unistd.h 头文件（使编译器获知自定义的系统调用编号）和 __LIBRARY__ 宏（使 _syscall1 等有效）。

在 iam.c 中，添加 iam() 在用户空间的接口函数 _syscall1(int, iam, const char*, name);，_syscall1 表示传递 1 个参数。调用 sys_iam() 的 API，传递用户输入。



```
1 /*
2  * linux/lib/iam.c
3  *
4  * (C) 2024 Liu Zikang
5  */
6
7 #define __LIBRARY__
8 #include <unistd.h>
9
10 _syscall1(int, iam, const char*, name);
11
12 int main(int argc, char **argv)
13 {
14     iam(argv[1]);
15     return 0;
16 }
17 |
```

在 whoami.c 中，添加 whoami() 在用户空间的接口函数，调用 _syscall2(int, whoami, char*, name, unsigned int, size);，_syscall2 表示传递 2 个参数。调用 sys_whoami() 的 API，将获取的字符串存放到 output 字符数组中，在屏幕上打印 output。



```
134
135 #define _syscall0(type,name) \
136 type name(void) \
137 { \
138     long __res; \
139     __asm__ volatile ("int $0x80" \
140         : "=a" (__res) \
141         : "0" (__NR_#name)); \
142     if (__res >= 0) \
143         return (type) __res; \
144     errno = -__res; \
145     return -1; \
146 }
147
148 #define _syscall1(type,name,atype,a) \
149 type name(atype a) \
150 { \
151     long __res; \
152     __asm__ volatile ("int $0x80" \
153         : "=a" (__res) \
154         : "0" (__NR_#name), "b" ((long)(a))); \
155     if (__res >= 0) \
156         return (type) __res; \
157     errno = -__res; \
158     return -1; \
159 }
160
161 #define _syscall2(type,name,atype,a,btype,b) \
162 type name(atype a,btype b) \
163 { \
164     long __res; \
165     __asm__ volatile ("int $0x80" \
166         : "=a" (__res) \
167         : "0" (__NR_#name), "b" ((long)(a)), "c" ((long)(b))); \
168     if (__res >= 0) \
169         return (type) __res; \
170     errno = -__res; \
171     return -1; \
172 }
173
174 #define _syscall3(type,name,atype,a,btype,b,ctype,c) \
175 type name(atype a,btype b,ctype c) \
176 { \
177     long __res; \
178     __asm__ volatile ("int $0x80" \
179         : "=a" (__res) \
180         : "0" (__NR_#name), "b" ((long)(a)), "c" ((long)(b)), "d" ((long)(c))); \
181     if (__res >= 0) \
182         return (type) __res; \
183     errno = -__res; \
184     return -1; \
185 }
186
```

·你能想出办法来扩大这个限制吗？

(1) 使用堆栈传递额外参数：将额外参数通过堆栈传递，使用 push 指令将参数推入堆栈，然后在内核中通过 esp 访问堆栈来获取参数。

(2) 通过指针传递数据结构：将多个参数封装成一个单独的对象，将该结构体或数组的首地址作为一个参数传递到内核，然后在内核中使用寄存器间接寻址来访问所有参数。

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

(1) 修改 include/unistd.h：添加#define __NR_foo xxx，xxx 为自定义系统调用编号；

(2) 修改 kernel/system_call.s：将系统调用总数 nr_system_calls 加 1；

(3) 修改 include/linux/sys.h：添加外部变量 extern int sys_foo()，并在系统调用函数表 sys_call_table 中添加 sys_foo；

(4) 实现 sys_foo() 函数：在 kernel/ 目录创建 foo.c 文件，编写 sys_foo() 的具体内容，注意引用需要的头文件和宏；

(5) 修改 kernel/Makefile：在 OBJS 中添加 foo.o，在依赖中添加 foo.s foo.o: foo.c ../include/linux/kernel.h ../include/unistd.h，使得添加的 kernel/foo.c 可以和其它 Linux 0.11 代码编译链接到一起；

(6) 在应用程序添加 foo() 在用户空间的接口函数，并通过 API 实现系统调用。