第2节 系统调用

前言:

一定要对系统调用的整个流程熟悉,里面涉及到的知识非常综合,前期如果没学到后面的东西,你可能对某些代码非常懵逼,比如一些段寄存器的修改,以及寻址,用户态的数据从哪里搬到内核态,怎么搬的,从用户态切换到内核态,背后发生了哪些事情,等等一些列问题,都不是单独学本章内容就能搞懂的,整个执行流程我在os部分笔记里面用画图的方式概要写了出来,具体还要结合哈工大os课程反复观看,不会的地方需要到处查资料,然后才能逐步把核心的细节搞明白。

系统调用这个lab,我们实现一个简化版的输出字符串的功能,比如用户态api叫zyw(char *),然后内核中使用sys_zyw(char *)来具体实现输出字符串的功能,当这些都设计好了之后,我们再写一个c程序调用这个用户api,并启动我们修改后的linux-0.11系统,在这个系统中运行这个c程序,看看修改后的这个linux-0.11是否正常运行,我们设计的系统调用是否能正常工作。

参考课程和lab教程:

https://www.bilibili.com/video/BV19r4y1b7Aw/?p=5

操作系统原理与实践 - 系统调用 - 蓝桥云课 (langiao.cn)

HIT-Linux-0.11/2-syscall/2-syscall.md at master · Wangzhike/HIT-Linux-0.11 (github.com)

哈工大操作系统之基础(Lab2添加系统调用实验全过程+跳坑解析) testlab2-CSDN博客

Linux操作系统(哈工大李治军老师)实验楼实验2-系统调用(1)哔哩哔哩bilibili

https://www.bilibili.com/video/BV16L411n7pa/?p=19

用户API——zyw ()

参考: linux-0.11/lib/open.c

```
"d" (va_arg(arg,int)));
if (res>=0)
    return res;
errno = -res;
return -1;
}
```

直接打开上面这个,参考它的写法即可

使用 touch linux-0.11/lib/zyw.c 创建文件,并使用gedit linux-0.11/lib/zyw.c 把下面的内容拷贝进去保存即可。

注意,在system_call中,在call sys_call_table之前,有一个压栈动作

```
pushl %edx
pushl %ecx  # push %ebx,%ecx,%edx as parameters
pushl %ebx  # to the system call
```

因此sys_zyw()函数前三个参数就是%ebx, %ecx, %edx, 不过我们用不上这么多参数,

只需要第一个参数,也就是%ebx。假如只需要%edx,那么前两个参数必须占位。

关于参数传递,这里就不多说了。

内核系统调用函数——sys_zyw()

系统调用号_NR_zyw

修改位置: linux-0.11/include/unistd.h

直接gedit打开添加定义即可

```
105 #define __NR_brk
106 #define __NR_setgid
107 #define __NR_getgid
                                                                                 47
  108 #define __NR_signal
108 #define NR_signal
109 #define NR_geteuid
110 #define NR_geteuid
111 #define NR_acct
112 #define NR_lock
113 #define NR_lock
114 #define NR_ioctl
115 #define NR_fcntl
116 #define NR_setpgid
118 #define NR_ulimit
119 #define NR_uname
120 #define NR_unaws
121 #define NR_unstat
122 #define NR_ustat
123 #define NR_dup2
                                                                                 49
                                                                                 50
51
                                                                                 52
                                                                                 54
                                                                                 58
 122 #define __nR_getpgr 65
124 #define __nR_getpgr 65
126 #define __nR_setsid 66
127 #define __nR_sigaction 66
128 #define NR_sgetmask
129 #define NR_ssetmask
130 #define NR_setreuid
131 #define NR_setregid
132 #define NR_zyw
 128 #define
                                        NR sgetmask
 133
 134 #define _syscall@(type,name)
135 type name(void) \
 136 { \
137 long __res; \
 138 __asm__ volatile ("int $0x80" \
139 __ : "=a" (__res) \
140 __ : "0" (__NR_##name)); \
141 if (__res >= 0) \
                                                                                                                                 C/C++/ObjC头文件 ▼ 制表符宽度: 8 ▼ 第132行,第1列 ▼ 插入
```

函数调用表, sys_call_table[]

位置: linux-0.11/include/linux/sys.h

```
保存(S)
51 extern int sys_getegid();
52 extern int sys_acct();
53 extern int sys_phys();
54 extern int sys_lock();
55 extern int sys_loctl();
56 extern int sys fcntl();
57 extern int sys_mpx();
58 extern int sys_setpgid();
59 extern int sys_ulimit();
60 extern int sys_uname();
61 extern int sys_umask();
62 extern int sys_chroot();
63 extern int sys_ustat();
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_zyw();
75 fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
76 sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
77 sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
78 sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
79 sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
80 sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
81 sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir, 82 sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
83 sys_getgid, sys_signal, sys_geteuld, sys_getegid, sys_acct, sys_phys,
84 sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
85 sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
86 sys_getpgrp, sys_setsid sys_sigaction, sys_sgetmask, sys_ssetmask,
87 sys_setreuid.sys_setreadd.sys_zyw }:
 正在保存文件"/home/amadeus/oslab/linux-0.11/incl... C/C++/ObjC 头文件 ▼ 制表符宽度: 8 ▼
                                                                                         第87行,第34列 ▼
                                                                                         📭 touch linux-0.11/lib/
```

函数调用表大小,nr_system_calls

位置: linux-0.11/kernel/system_call.s

```
44 EFLAGS
                  = 0x24
45 OLDESP
                  = 0x28
46 OLDSS
                   = 0x2C
47
48 state = 0
                           # these are offsets into the task-struct.
49 counter
50 priority = 8
51 \text{ 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 \text{ sa_mask} = 4
58 \text{ sa\_flags} = 8
59 sa_restorer = 12
61 nr_system_calls = 73
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
71 .align 2
72 bad_sys_call:
73
           movl $-1,%eax
74
           iret
75 .align 2
76 reschedule:
77
           pushl $ret_from_sys_call
78
           jmp schedule
79 .align 2
80 system call:
正在保存文件"/home/amadeus/oslab/linux-0.11/kernel/system_call.s... C▼ 制表符宽度: 8▼ 第61行,第21列 ▼ 插入
```

函数本体, sys_zyw

位置: linux-0.11/kernel/zyw.c

```
#include <errno.h>
#include <asm/segment.h>
#include <stdarg.h>
#include <linux/kernel.h>
int sys_zyw(const char *str) {
   int i;
   char tmp[100]:
   for(i = 0; i < 100; i++) {
       // 将用户态数据搬到内核态
       tmp[i] = get_fs_byte(str + i);
       if(tmp[i] == '\0')
           break;
   }
   printk("%s",tmp);
   return i; // 简单返回显示的字符串数量,不作各种判断了
}
```

可以看到用 get_fs_byte() 获得一个字节的用户空间中的数据。get_fs_byte() 在/include/asm/segment.h中实现:

```
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;
}
```

movb %%fs:%1,%0 就是把1号寄存器中的内容,也就是*addr,其实就是用户态的数据,搬到0号寄存器 "=r" (_v) 就是把0号寄存器的内容,搬到内核态的内存变量_v中 经过上面就实现了将用户态的一个字符搬到内核态中去。

注意,寻址的时候,fs=0x17,二进制就是10111,TI=1,会查ldt段表,而不是gdt段表,这是能找到用户态的关键,

而addr就是一开始从用户态传来的偏移地址,就是我们的zyw()里面的参数。

理解这个代码,需要你理解mmu的工作原理。

c程序

前面都是操作系统层面的,最后提供给应用程序一个zyw()接口,现在我们编写应用程序,直接使用这个接口。

另外需要注意,linux-0.11操作系统部分前面已经全部写完了,后面这个c程序写在linux-0.11挂载的的hdc文件系统中,当然,我们在ubuntu下也能挂载,为了编写程序方便,我们先用Ubuntu挂载,把c程序写进去,然后卸载这个文件系统,最后编译运行linux-0.11系统,在linux-0.11中来编译我们的c程序,并运行,看看是否work。

1. 挂载命令

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

2. 然后在hdc/usr/root目录下创建test.c文件,这就是我们的c语言主程序

```
touch hdc/usr/root/test.c
```

"test.c"文件内容如下

```
return -1;
}

int main() {
    const char *message = "Hello, ZYW system call!";
    zyw(message);
    return 0;
}
```

注意,用户API这里重写一遍了,因为前面写的在这里没有#include到,到底怎么才能#include进来,后面有空再研究,现在直接抄过来

3. 然后修改hdc/usr/include/unistd.h,添加一行。

这个其实前面在linux-0.11系统里面已经写了,但是在主程序中却看不到,不清楚为啥

```
#define __NR_zyw 72
```

4. 最后卸载hdc

```
sudo umount hdc
```

假如没有卸载hdc就run了,会导致文件系统破坏,删除oslab,重新来一遍。

Makefile文件修改

(1) 第一处

```
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 zyw.o
```

添加了 zyw.o。

(2) 第二处

```
### 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
```

改为:

添加了 who.s who.o: who.c ../include/linux/kernel.h ../include/unistd.h。

Makefile 修改后,和往常一样 make all 就能自动把 zyw.c 加入到内核中了。

开始编译运行

首先进入linux-0.11目录下进行编译

```
cd linux-0.11
make clean
make all
```

然后启动linux-0.11

1s

```
cd ~/oslab
./run
```

在linux-0.11下, ls查看文件

你会发现hdc里面的test.c就在里面

然后,我们编译test.c,注意,我们还是在linux-0.11系统下,不是在ubuntu下

```
gcc -o test test.c -Wall
```

-Wall是可选项,可提供编译错误提示

然后就正常编译成功,最后我们执行这个程序

```
./test
```

你会在屏幕上看见我们在test.c主程序中定义的message字符串,这个字符串就是通过zyw()系统调用,复制到内核中,然后由printk进行输出的。