

操作系统实验二

PB18151866 龚小航

实验内容一：添加 Linux 系统调用

实验目标：了解系统调用如何实现，并在 Linux 0.11 中添加两个系统调用。

实验步骤：

①在 os_lab/Linux-0.11/include/unistd.h 下定义系统调用号，并声明系统调用函数的形式。

打开(O) 未命名

unistd.h

admin:///home/virtual/os_lab/Linux-0.11/include

```
#define __NRuname 59
#define __NRumask 60
#define __NRchroot 61
#define __NRustat 62
#define __NRdup2 63
#define __NRgetppid 64
#define __NRgetpgrp 65
#define __NRsetsid 66
#define __NRsigaction 67
#define __NRsgetmask 68
#define __NRssetmask 69
#define __NRsetreuid 70
#define __NRsetregid 71

#define __NR_print_val 72
#define __NR_str2num 73

#define _syscall0(type,name) \
    type name(void) \
{ \
    long __res; \
    __asm__ volatile ("int $0x80" \
```

打开(O) 未命名

unistd.h

admin:///home/virtual/os_lab/Linux-0.11/include

```
int setpgid(pid_t pid, pid_t pgid);
int setuid(uid_t uid);
int setgid(gid_t gid);
void (*signal(int sig, void (*fn)(int)))(int);
int stat(const char * filename, struct stat * stat_buf);
int fstat(int fildes, struct stat * stat_buf);
int stime(time_t * tptr);
static int sync(void);
time_t time(time_t * tloc);
time_t times(struct tms * tbuf);
int ulimit(int cmd, long limit);
mode_t umask(mode_t mask);
int umount(const char * specialfile);
int uname(struct utsname * name);
int unlink(const char * filename);
int ustat(dev_t dev, struct ustat * ubuf);
int utime(const char * filename, struct utimbuf * times);
pid_t waitpid(pid_t pid, int * wait_stat, int options);
pid_t wait(int * wait_stat);
int write(int fildes, const char * buf, off_t count);
int dup2(int oldfd, int newfd);
int getppid(void);
pid_t getpgrp(void);
pid_t setsid(void);

int print_val(int a);
int str2num(char *str, int str_len, long *ret);

#endif
```

C/C++/ObjC 头文件 制表符宽度: 8

②在 os_lab/Linux-0.11/kernel/system_call.s 中修改系统调用的个数，以使此系统调用被调用时，可以识别到。

打开(O) 未命名

system_call.s

admin:///home/virtual/os_lab/Linux-0.11/kernel

保存(S)

```
sigaction = 16 # MUST be 16 (=len of sigaction)
blocked = (33*16)

# offsets within sigaction
sa_handler = 0
sa_mask = 4
sa_flags = 8
sa_restorer = 12

nr_system_calls = 74

/*
 * Ok, I get parallel printer interrupts while using the floppy for some
 * strange reason. Urgel. Now I just ignore them.
 */
.globl system_call, sys_fork, timer_interrupt, sys_execve
.globl hd_interrupt, floppy_interrupt, parallel_interrupt
.globl device_not_available, coprocessor_error

.align 2
bad_sys_call:
    movl $-1, %eax
    iret

.align 2
reschedule:
    pushl $ret from sys call
```

C 制表符宽度: 8 第 65 行, 第 43 列

③在 os_lab/Linux-0.11/include/linux/sys.h 中添加 extern 头，再在 sys_call_table[] 中加入系统调用的‘地址’。

打开(O) 未命名

sys.h

admin:///home/virtual/os_lab/Linux-0.11/include/linux

保存(S)

```
extern int sys_getpgrp();
extern int sys_setsid();
extern int sys_sigaction();
extern int sys_sgetmask();
extern int sys_ssetmask();
extern int sys_setreuid();
extern int sys_setregid();

extern int sys_print_val();
extern int sys_str2num();

fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_set
sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_ph
sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
sys_getpgrp, sys_setsid, sys_sigaction, sys_sgetmask, sys_ssetmask,
sys_setreuid, sys_setregid,
sys_print_val, sys_str2num
};
```

④在 linux-0.11/kernel 中实现该系统调用，源代码如下。 并修改 Makefile 文件。修改如下：

打开(O) 图标

os_lab2.c
admin:///home/virtual/os_lab/Linux-0.11/kernel

```
#define __LIBRARY__

#include <errno.h>

#include <linux/sched.h>
#include <linux/kernel.h>
#include <asm/segment.h>
#include <asm/system.h>

int sys_print_val(int a){
    printk("in sys_print_val: %d \n",a);
    return 0;
}

int sys_str2num(char *str, int str_len, long *ret){
    int i;
    long result=0;
    long pow=1;
    char c;
    if(str_len>8){
        printk("sys_str2num overflow \n");
        *ret=0;
        return -1;
    }
    for(i=str_len-1;i>=0;i--){
        c=get_fs_byte((char*)&str[i]);
        result+=pow*(c-'0');
        pow*=10;
    }

    put_fs_long(result,ret);

    return 0;
}
```

打开(O) 图标

Makefile
admin:///home/virtual/os_lab/Linux-0.11/kernel

```
.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 os_lab2.o

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

    ../include/asm/system.h ../include/asm/segment.h ../include/asm/io.h
vsprintf.s vsprintf.o: vsprintf.c ../include/stdarg.h ../include/string.h

os_lab2.s os_lab2.o: os_lab2.c ../include/asm/segment.h ../include/unistd.h
```

⑤将 os_lab/Linux-0.11/hdc/usr/include/unistd.h 文件中添加系统调用号和系统调用的声明。

这一步操作需要挂载 img 镜像文件。这一步和第一步更改的内容相同。这里略去。

⑥写测试程序进行测试。测试代码如下，将其放入镜像文件的 root 目录下。

打开(O) 图标

test.c
admin:///home/virtual/os_lab/Linux-0.11/hdc/usr/root

保存(S)

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

_syscall1(int, print_val,int, a); /* print_val()在用户空间的接口函数,
_syscall3(int, str2num,char*,str,int,str_len, long*, ret);

int main(){
    char input [10];
    long ret;
    int len=0;
    printf("Give me a string:\n");
    scanf("%s",input);
    while(input[len]!='\0'){
        len++;
    }
    str2num(input,len,&ret);
    print_val((int)ret);

    return 0;
}
```

实验结果：

在主机终端中依次输入命令 make clean; make; make start，编译内核并进入 Linux-0.11 系统。

使用 gcc 编译刚才的测试文件，运行，实现实验要求的输出输入。至此完成第一部分实验。

virtual@L-virtual-machine: ~/os_lab/Linux-0.11

文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

记录了1+0 的写出
512 bytes copied, 0.000131534 s, 3.9 MB/s
记录了0+1 的读入
记录了0+1 的写出
311 bytes copied, 9.226e-05 s, 3.4 MB/s
记录了301+1 的读入
记录了301+1 的写出
154401 bytes (154 kB, 151 KiB) copied, 0.00125696 s, 123 MB/s
记录了2+0 的读入
记录了2+0 的写出
2 bytes copied, 0.000199547 s, 10.0 kB/s
virtual@L-virtual-machine:~/os_lab/Linux-0.11\$ make start
WARNING: Image format was not specified for 'Image' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
WARNING: Image format was not specified for 'hdc-0.11.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.01H:ECX.vmx [bit 5]
[]

QEMU

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+00F8DD0+00ECDD0

Booting from Floppy...

Loading system ...

Partition table ok.
46066/60000 free blocks
19232/20000 free inodes
3423 buffers = 3505152 bytes buffer space
Free mem: 12451840 bytes
Ok.
r[/usr/root]# ls
OS_exp1 gcclib140 hello.c shell test.c
README hello mtools.howto shell.c
r[/usr/root]# gcc -o test test.c
r[/usr/root]# ./test
Give me a string:
6654321
in sys_print_val: 6654321
r[/usr/root]# _

实验第一部分问答：

问题1：简要描述如何在Linux 0.11添加一个系统调用

- ① 定义系统调用号，并声明系统调用函数的形式。
- ② 修改系统调用的个数，以使此系统调用被调用时，可以识别到。
- ③ 添加 **extern** 头，并在 **sys_call_table[]**中加入系统调用的‘地址’。
- ④ 写调用的具体代码，实现该系统调用，并修改 **Makefile** 文件。
- ⑤ 将挂载目录下的 **unistd.h** 文件中添加系统调用号和系统调用的声明。（和第一步修改内容一样）
- ⑥写测试程序进行测试。

问题2：系统是如何通过系统调用号索引到具体的调用函数的？

- 1、API把系统调用的编号存入EAX中，把函数参数存入其他通用寄存器中并触发0x80号中断；
- 2、调用system_call函数，函数中语句call sys_call_table(%eax,4)会在中断向量表中找到相应的系统调用的函数地址；
- 3、通过这个地址执行真正的系统调用函数。

问题3：在Linux-0.11中，系统调用最多支持几个参数？有什么方法可以超过这个限制吗？

在Linux-0.11中，系统调用最多支持3个参数。

可以通过写一个新的宏来支持4个参数的系统调用，也可以将多余的参数传入其他的通用寄存器中。

实验内容二：熟悉 Linux 下常见的系统调用函数

实验目标：利用 Linux 提供的系统调用，实现一个简单 shell 程序

实验步骤：在给出的代码框架上，补全 shell 代码。

第一个添加部分，使用 `fork()` 创建一个新的进程。

```
/* 1. 使用系统调用创建新进程 */
pid=fork();
```

子进程 `pid=0`，这也是区分子进程与父进程的标志，它们执行不同的代码。

```
/* 2. 子进程部分 */
if(pid==0)
{
```

接下来的 2.1 修改处，`pipe` 无用端是读端口，因此关闭 0 端，将 1 端指向标准输出。完成后关闭 1 端。

```
45  | if (type == 'r') {
46  |     /* 2.1 关闭pipe无用的一端，将I/o输出发送到父进程 管道0端读，1端口写*/
47  |     close(pipe_fd[0]);
48  |     if (pipe_fd[1] != STDOUT_FILENO) {
49  |         dup2(pipe_fd[1], STDOUT_FILENO);
50  |         close(pipe_fd[1]);
51  |     }
```

2.2 修改处与上一处相似，`pipe` 无用端是写端口，因此关闭 1 将 0 端指向标准输入。完成后关闭。

```
53  | /* 2.2 关闭pipe无用的一端，接收父进程提供的I/o输入 */
54  | close(pipe_fd[1]);
55  | if (pipe_fd[0] != STDIN_FILENO) {
56  |     dup2(pipe_fd[0], STDIN_FILENO);
57  |     close(pipe_fd[0]);
58  | }
```

下面的修改处补全 `execl` 函数，通过该系统调用来执行 `shell` 命令。

```
64      /* 2.3 通过exec系统调用运行命令 */
65      execl(SHELL, "sh", "-c", cmd, NULL);
```

接下来是父进程部分，和子进程部分类似。Type==r 时关写开读，反之亦然。

```
69      /* 3. 父进程部分 */
70      else {
71          if (type == 'r') {
72              close(pipe_fd[1]);
73              proc_fd = pipe_fd[0];
74          } else {
75              close(pipe_fd[0]);
76              proc_fd = pipe_fd[1];
77          }
78      }
79      child_pid[proc_fd] = pid;
80      return proc_fd;
81  }
```

4.1 处，通过 `fork()` 创建新进程，`if` 判断创建是否成功，如果失败标志变量 `stat` 值即为-1。

```
108      /* 4.1 创建一个新进程 */
109      if((pid=fork())<0) stat=-1;
```

4.2 处，判断子进程，并使 `execl` 系统调用来执行命令：

```
111      /* 4.2 子进程部分 */
112      else if(pid == 0){
113          execl(SHELL, "sh", "-c", cmdstring, NULL);
114          _exit(127);
115      }
```


4.3 处，父进程部分，通过 `waitpid()` 等待子进程运行结束后继续运行，如果出错则将标志变量 `stat` 赋为-1.

```
117      /* 4.3 父进程部分：等待子进程运行结束 */
118      else{
119          while(waitpid(pid,&stat,0) < 0)
120              if(errno != EINTR){
121                  stat = -1;
122                  break;
123              }
124      }
125      return stat;
126  }
```

5.1 处，这部分命令需要管道。先通过 `os_popen()` 打开管道，然后用 `read()` 将 `cmd1` 标准输出存入 `buf` 中，存入字的数量返回赋值给 `count`，做完以后，利用 `os_pclose()` 关闭管道，最后再判断关闭是否成功。

```
183      /* 5.1 运行cmd1，并将cmd1标准输出存入buf中 */
184      fd1 = os_popen(cmd1,'r');
185      count = read(fd1,buf,sizeof(buf));
186      status = os_pclose(fd1);
187      if(status == -1) printf("%s is error!\n",cmd1);
```

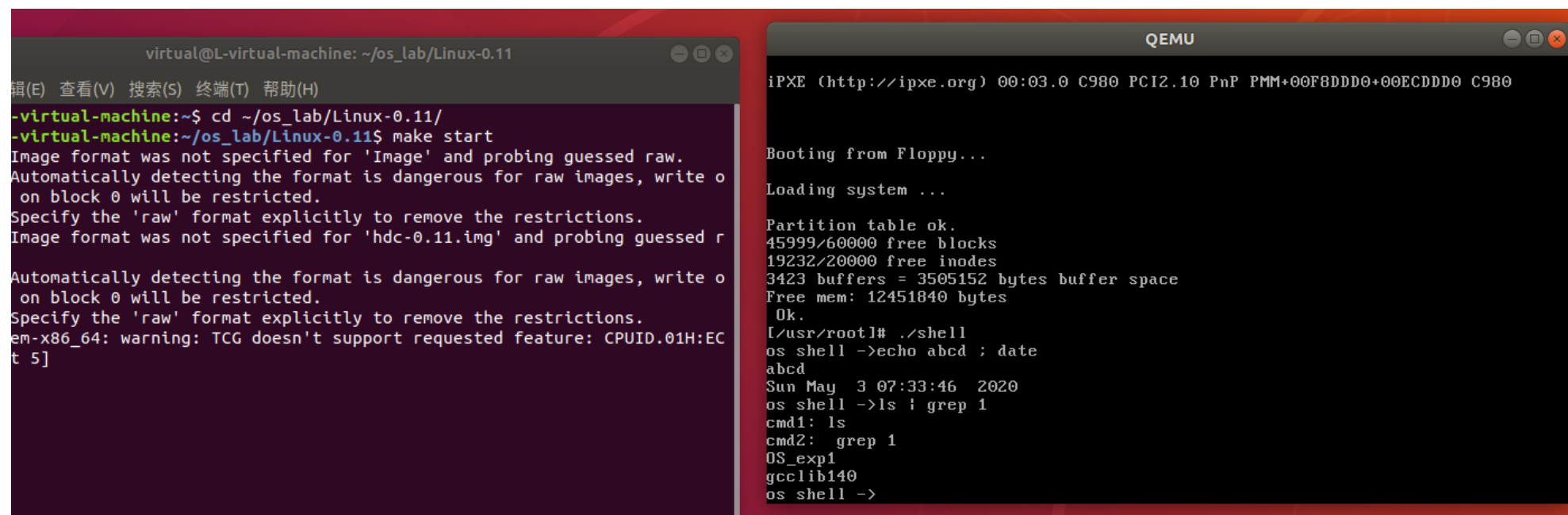
5.2，与上一处完全相同，只不过是将 `buf` 的内容写入到 `cmd2` 的输入中：

```
189      /* 5.2 运行cmd2，并将buf内容写入到cmd2输入中 */
190      fd2 = os_popen(cmd2,'w');
191      write(fd2,buf,count);
192      status = os_pclose(fd2);
193      if(status == -1) printf("%s is error!\n",cmd2);
```

最后一处需要补充普通命令的运行，即不需要管道。用 `os_system` `system()` 函数运行命令即可。

```
197      /* 6 一般命令的运行 */
198      status = os_system(cmds[i]);
199      if(status == -1) printf("%s is error!\n",cmds[i]);
```

第二部分实验结果：



```
virtual@L-virtual-machine: ~/os_lab/Linux-0.11
编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
~virtual-machine:~$ cd ~/os_lab/Linux-0.11/
~virtual-machine:~/os_lab/Linux-0.11$ make start
Image format was not specified for 'Image' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write o
on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
Image format was not specified for 'hdc-0.11.img' and probing guessed r
Automatically detecting the format is dangerous for raw images, write o
on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
em-x86_64: warning: TCG doesn't support requested feature: CPUID.01H:EC
t 5]

QEMU
iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+00F8DDDD+00ECDDDD C980

Booting from Floppy...

Loading system ...

Partition table ok.
45999/60000 free blocks
19232/20000 free inodes
3423 buffers = 3505152 bytes buffer space
Free mem: 12451840 bytes
Ok.
[/usr/root]# ./shell
os shell ->echo abcd ; date
abcd
Sun May 3 07:33:46 2020
os shell ->ls | grep 1
cmd1: ls
cmd2: grep 1
OS_exp1
gcclib140
os shell ->
```

输入输出符合预期，第二部分实验结束。