操作系统第二次实验实验报告

肖桐 PB18000037

一、添加Linux系统调用

1. 实验过程

我们需要增加两个系统调用，分别称为print_val和str2num。对应的函数与如下形式 .

```
1  int print_val(int a);    //通过printk在控制台打印如下信息（假设a是1234）：
2  // in sys_print_val: 1234
3  int str2num(char *str, int str_len, long *ret);    //将一个有str_len个数字的字符串str转换成十进制数字，然后将结果写到ret指向的地址中，其中数字大小要合适，应当小于100000000(1*e^8).
```

首先进入kernel/system_call.s中将参数nr_system_calls的值改为74，意为在我们添加以上两个系统调用之后Linux-0.11中存在74个系统调用。

```
nr_system_calls = 74
```

第二，在include/unistd.h文件中添加两个参数：**__NR_print_val以及__NR_str2num**

```
#define __NR_print_val  72
#define __NR_str2num    73
```

同时增加两个函数原型：

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

然后通过lab1的文件交换操作进入Linux-0.11文件系统，再次更改unistd.h文件，修改方式于上面相同。

第三，进入include/linux/sys.h文件中增加系统调用函数原型，同时在sys_call_table[ ]数组中也增加系统调用函数原型：

```
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_setgid,
sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
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 };
```

第四，实现这两个系统调用函数，再kernel文件夹中新建文件print_str2num.c文件，在该文件中实现函数sys_print_val()以及sys_str2num()：

```c
#define __LIBRARY__

#include<unistd.h>
#include<asm/segment.h>
#include <linux/kernel.h>

char num_to_char(int i) /* i is a single number */
{
    return (char)(i + 48);
}
long char_to_num(char c)
{
    return (long)(c - 48);
}

int sys_print_val(int a)
{
    int num_len = 0;
    char str_num[64];
    for(int num = a; num != 0; num_len++)   //计算数字num的位数
    {
        num /= 10;
    }
    str_num[num_len] = '\0';
    for(int i = num_len - 1, num = a; num != 0; i--)    //将num转化为字符串
    {
        str_num[i] = num_to_char(num % 10);
        num /= 10;
    }
    printk(str_num);
    return num_len;
}

int sys_str2num(char* str, int str_len, long* ret)
{
    long num = 0;
    int i = 0;
    for(i = 0; i < str_len; i++)
    {
        num = num * 10 + char_to_num(get_fs_byte(&str[i]));
    }
    put_fs_long(num, ret);
    return 0;
}
```

最后修改MakeFile：

```
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 print_str2num.o
```

```
print_str2num.s print_str2num.o: print_str2num.c ../include/asm/segment.h
```

至此，在Linux-0.11中添加系统调用已完成，现在只需编写测试程序test.c：

```c
#define __LIBRARY__

```

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

_syscall1(int,print_val,int,a);
_syscall3(int,str2num,char*,str,int,str_len,long*,ret); //调用system call

int main()
{
    char str[81];
    long* num_addr;
    int str_len;
    int num;
    printf("Give me a String:\n");
    scanf("%s", &str);
    for(str_len = 0; str[str_len] != '\0'; str_len++);  /* get the length of the string */
    str2num(str, str_len, num_addr);
    num = (int)(*num_addr);
    printf("in sys_print_val:\n");
    print_val(num);
    printf("\n");
    return 0;
}
```

在Linux-0.11根目录中执行如下命令编译内核：

```
make clean
make
```

然后使用lab1中文件交换的方法将测试文件复制到Linux-0.11中，使用gcc编译之后运行，其结果如下：

```
[/usr]# gcc test.c -o test
[/usr]# ls
bin             include         local           test            var
docs            lab2_shell      root            test.c
hello.txt       lab2_shell.c    src             tmp
[/usr]# ./test
Give me a String:
12345678
in sys_print_val:
12345678
[/usr]# _
```

至此，本次实验的第一部分所有内容已经完成。

2. 回答问题

(1). 简要描述如何在Linux 0.11添加一个系统调用 &

(2). 系统是如何通过系统调用号索引到具体的调用函数的?

首先我们先观察系统调用的宏定义原型**syscall**. 该宏定义在include/unistd.h文件中。以**syscall1()**为例。其在unistd.h文件中原型为：

```
#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; \
}
```

我们可以先使用该宏定义对**_syscall1(int,print_val,int,a)**进行宏展开：

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

可以看出，该宏将传入的参数展开为了一个汇编函数。从第二行可以看出形参**type**展开为该函数的返回值类型，形参**name**展开为该函数的函数名。形参**atype**和形参**a**分别为传入参数的类型和值。
注意到在第6行出现了一个参数**__NR_pirnt_val**，这便是系统调用函数**print_val()**的系统调用，因为宏定义的原因，**print_val()函数的系统调用号必须取名为__NR_print_val**，否则二者之间无法建立联系。

而关于**__NR_print_val**参数的用处，在该宏定义中参数**__NR_print_val**被传入寄存器%eax中，而将这个参数传入寄存器%eax到底有什么用，我们还得进入system_calls.s文件中查看。现摘抄一段如下：

```
system_call:
    cmpl $nr_system_calls-1,%eax
    ja bad_sys_call
    push %ds
    push %es
    push %fs
    pushl %edx
    pushl %ecx      # push %ebx,%ecx,%edx as parameters
    pushl %ebx      # to the system call
    movl $0x10,%edx     # set up ds,es to kernel space
    mov %dx,%ds
    mov %dx,%es
    movl $0x17,%edx     # fs points to local data space
    mov %dx,%fs
    call *sys_call_table(,%eax,4)
    pushl %eax
    movl current,%eax
    cmpl $0,state(%eax)     # state
    jne reschedule
    cmpl $0,counter(%eax)       # counter
    je reschedule
```

在第2行，首先判断了参数**nr_system_calls - 1**与寄存器%eax中的值——即之前传入的**__NR_print_val**大小关系，若**nr_system_calls - 1 < %eax**，则会执行第三行语句**ja bad_sys_call**，直接跳出，不再执行system_call。这也是为什么我们之前要改变参数**nr_system_calls**的值与现有系统调用个数相同。

然后最重要的就是第15行语句：**call *sys_call_table(,%eax,4)**。这段语句实际上是在将寄存器%eax的值传给include/linux/sys.h文件中的数组**sys_call_table[ ]**，调用对应下标的函数。

```
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_setgid,
sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
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 };
```

到了这里，添加系统调用的步骤，这些步骤的原因，以及为什么这么做计算机可以识别用户添加的系统调用的原因也逐渐明了了。以添加系统调用**int print_val(int a)**为例，原因如下：

在使用print_val()之前(比如在test.c文件中)，我们应当先使用_syscall1(int,print_val,int,a)函数，该函数的目的是调用真正实现print_val()系统调用功能的函数(本例中为在print_str2num.c文件中定义的sys_print_val()函数)，然后根据_syscall1()函数的宏定义，有关系统调用print_val()的各种参数被传入寄存器，其中参数__NR_print_val被传入寄存器%eax。然后进行0x80中断调用，此时来到了kernel/system_calls.s文件中的system_call部分。执行到语句call *sys_call_table(,%eax,4)时，系统再次跳转到了include/linux/sys.h文件中的sys_call_table[ ]数组中，并将寄存器%eax的值传给该数组，寻找对应的真正实现系统调用功能的函数sys_print_val()。可知我们之前定义的参数__NR_print_val的值便是数组sys_call_table[ ]中真正实现系统调用功能的函数sys_print_val()的下标，然后操作系统便会到其他的文件中去寻找实验该函数的源代码(本例中为print_str2num.c文件)。在这一切的调用都结束后，我们回到test.c文件中，每当我们调用print_val()函数，实际上操作系统都在通过_syscall1()函数调用sys_print_cal()函数来实现相应的功能。

通过以上的调用、对应关系，可以知道真正实现print_val()功能的sys_print_val()函数也可以取别的函数名，并非一定要在前面加上sys_才可以。真正起到对应关系的是参数**__NR_print_val**的值与该函数在**sys_call_table[ ]**数组中的位置关系，二者在这个数组中是一个下标与元素的对应关系。只要这个不改变，不论sys_print_val()取什么函数名，**_syscall1()**都能够找到对应的函数。但是由于宏定义的原因，**print_val()函数与参数__NR_print_val之间的联系必须通过取名来建立。即若系统调用函数名为str(str为一个任意的满足取名规则的字符串)，其对应的参数名必须为字符串：__NR_str。**

(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，因为能够传递最多参数的函数为_syscall3(...)。

超过这个限制的方法：参考main函数的传参方式：**main(int argc, char* argv[ ])**，可以传入一个char**型数据，在这个字符串数组中写入其他更多的参数，再在系统调用函数的具体实现中将这些参数取出并使用。

二、熟悉Linux下常见的系统调用函数

os_popen()函数实现：

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

/* 2. 子进程部分 */
if(pid == 0) {
    if (type == 'r') {  /* parent process read */
        /* 2.1 关闭pipe无用的一端，将I/O输出发送到父进程 */
        close(pipe_fd[0]);
        if (pipe_fd[1] != STDOUT_FILENO) {
            dup2(pipe_fd[1], STDOUT_FILENO);
            close(pipe_fd[1]);              /* 关闭多余的写端 */
        }
    }
    else {  /* parent process write */
        /* 2.2 关闭pipe无用的一端，接收父进程提供的I/O输入 */
        close(pipe_fd[1]);
        if (pipe_fd[0] != STDIN_FILENO) {
            dup2(pipe_fd[0], STDIN_FILENO);
            close(pipe_fd[0]);
        }
    }
    /* 关闭所有未关闭的子进程文件描述符（无需修改） */
    for (i = 0; i < NR_TASKS; i++)
        if(child_pid[i] > 0)
            close(i);
    /* 2.3 通过execl系统调用运行命令 */
    execl(SHELL, "sh", "-c", cmd, NULL);
    _exit(127);
}
/* 3. 父进程部分 */
if (type == 'r') {  /* read */
    close(pipe_fd[1]);
    proc_fd = pipe_fd[0];
}
else {                          /* write */
    close(pipe_fd[0]);
    proc_fd = pipe_fd[1];
}
child_pid[proc_fd] = pid;
return proc_fd;
```

1.使用**fork()**创建新进程。

2.1.令pipe_fd[0]为管道读端，pipe_fd[1]为管道写端。当父进程为读时，子进程为写。故先将子进程的读端关闭，然后将子进程的写端重定向到STDOUT_FILENO之后，将多于的写端关闭。2.2.同理。

2.3.子进程调用**execl()**函数后使用"/bin/sh"执行指令，若失败则退出。

3.父进程为读时，将父进程的写端关闭，反之将父进程的读端关闭。然后将proc_fd指向当前未关闭的一段，并最后返回proc_fd，用于之后将管道关闭。

os_system()函数实现：

```c
int os_system(const char* cmdstring) {
    pid_t pid;
    int stat;
    int cmd_num;

    if(cmdstring == NULL) {
        printf("nothing to do\n");
        return 1;
    }

    /* 4.1 创建一个新进程 */
    pid = fork();

    /* 4.2 子进程部分 */
    if(pid == 0) {
        execl(SHELL, "sh", "-c", cmdstring, NULL);
        _exit(127);
    }

    /* 4.3 父进程部分：等待子进程运行结束 */
    else if (pid > 0) {
        waitpid(pid, &stat, 0);
    }

    return stat;
}
```

4.1.使用**fork()**创建新进程。

4.2.子进程调用**execl()**函数之后使用"/bin/sh"执行指令，若失败则退出。

4.3.父进程等待子进程结束。

main()函数实现：

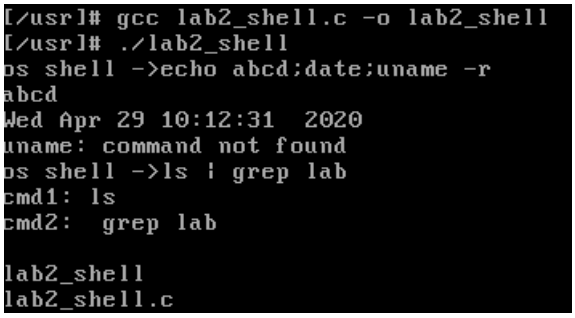
```c
for(i = 0; i < cmd_num; i++) {
    char *div = strchr(cmds[i], '|');
    if (div) {
        /* 如果需要用到管道功能 */
        char cmd1[MAX_CMD_LENGTH], cmd2[MAX_CMD_LENGTH];
        int len = div - cmds[i];
        memcpy(cmd1, cmds[i], len);     /* 获取管道符'|'之前的命令 */
        cmd1[len] = '\0';
        len = (cmds[i] + strlen(cmds[i])) - div - 1;
        memcpy(cmd2, div + 1, len);     /* 获取管道符'|'之后的命令 */
        cmd2[len] = '\0';
        printf("cmd1: %s\n", cmd1);
        printf("cmd2: %s\n", cmd2);
        /* 5.1 运行cmd1，并将cmd1标准输出存入buf中 */
        zeroBuff(buf, 4096);
        fd1 = os_popen(cmd1, 'r');
        count = read(fd1, buf, 4096);
        status = os_pclose(fd1);

        /* 5.2 运行cmd2，并将buf内容写入到cmd2输入中 */
        fd2 = os_popen(cmd2, 'w');
        write(fd2, buf, count);
        status = os_pclose(fd2);
        zeroBuff(buf, 4096);

    }
    else {
        /* 6 一般命令的运行 */
        os_system(cmds[i]);
    }
    zeroBuff(cmds[i], MAX_CMDLINE_LENGTH);
}
```

5.1.在运行**os_popen()**之前先将数组buf清零。之后使用**os_popen()**函数创建管道同时将cmd1的运行结果输出到STDOUT_FILENO。然后使用**read()**将cmd1的运行结果写入buf，最后将管道关闭。

5.2.与上面同理，先使用**os_popen()**函数创建管道，之后用**write()**函数将buf内的数据写入STDIN_FILENO作为cmd2的输入。最后关闭管道，再次将buf清零。

6.若不需要用到管道，则直接使用**os_system()**执行命令。

至此，lab2_shell.c文件已填写完成。使用lab1中文件交换的方法将lab2_shell.c复制到Linux-0.11中，使用gcc编译后运行如下：



至此，本次实验的第二部分所有内容已经完成。