

EXP2.1

实验过程

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

```
#define __NR_setregid    71
#define __NR_print_val  72
#define __NR_str2num     73

pid_t setsid(void);
void print_val(int a);
int str2num(char *str, int str_len, long *ret);
#endif
```

- ② 在 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。

```
nr_system_calls = 74
```

- ③ 在 linux0.11/include/linux/sys.h 中添加 extern 头，再在 sys_call_table[] 中加入系统调用的‘地址’。

```
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_ftime, 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 };
```

④ 在 linux-0.11/kernel 中实现该系统调用，并修改 Makefile 文件。

```
vmware@ubuntu:~/oslab/Linux-0.11/kernel$ ls
asm.o      exit.o      math        printk.c    sched.o     sys.c        traps.o
asm.s      fork.c      mktime.c    printk.o    signal.c     sys.o        vsprintf.c
blk_drv    fork.o      mktime.o    print_val.c signal.o     system_call.o vsprintf.o
chr_drv    kernel.o    panic.c     print_val.o str2num.c    system_call.s
exit.c     Makefile    panic.o     sched.c     str2num.o    traps.c
```

```
#include<asm/segment.h>
#include<unistd.h>
#include<errno.h>
#include <linux/sched.h>
#include <linux/kernel.h>
#include <sys/times.h>
#include <sys/utsname.h>

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

```
#include<asm/segment.h>
#include<unistd.h>
#include<errno.h>
#include <linux/sched.h>
#include <linux/kernel.h>
int sys_str2num(char *str, int str_len, long *ret)
{
    char ch[str_len];
    int i=0;
    for(;i<str_len;i++)
        ch[i]=get_fs_byte(str+i);
    while(i>=0)
    {
        put_fs_long(ch[i]-'0',ret+i);
        i--;
    }
    return 1;
}
```

```
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_val.o str2num.o
```

```
print_val.o: print_val.c ../include/asm/segment.h ../include/errno.h \
../include/linux/sched.h ../include/sys/times.h ../include/sys/utsname.h
str2num.o: str2num.c ../include/asm/segment.h ../include/errno.h \
../include/linux/sched.h ../include/linux/kernel.h \
../include/unistd.h
```

⑤编译内核

make: 为 gcc 无而做这件事。

```
vmware@ubuntu:~/oslab/Linux-0.11$ make clean
make[1]: 进入目录"/home/vmware/oslab/Linux-0.11/mm"
make[1]: 离开目录"/home/vmware/oslab/Linux-0.11/mm"
make[1]: 进入目录"/home/vmware/oslab/Linux-0.11/fs"
make[1]: 离开目录"/home/vmware/oslab/Linux-0.11/fs"
make[1]: 进入目录"/home/vmware/oslab/Linux-0.11/kernel"
make[2]: 进入目录"/home/vmware/oslab/Linux-0.11/kernel/chr_drv"
make[2]: 离开目录"/home/vmware/oslab/Linux-0.11/kernel/chr_drv"
make[2]: 进入目录"/home/vmware/oslab/Linux-0.11/kernel/blk_drv"
make[2]: 离开目录"/home/vmware/oslab/Linux-0.11/kernel/blk_drv"
make[2]: 进入目录"/home/vmware/oslab/Linux-0.11/kernel/math"
make[2]: 离开目录"/home/vmware/oslab/Linux-0.11/kernel/math"
make[1]: 离开目录"/home/vmware/oslab/Linux-0.11/kernel"
make[1]: 进入目录"/home/vmware/oslab/Linux-0.11/lib"
make[1]: 离开目录"/home/vmware/oslab/Linux-0.11/lib"
make[1]: 进入目录"/home/vmware/oslab/Linux-0.11/boot"
make[1]: 离开目录"/home/vmware/oslab/Linux-0.11/boot"
```

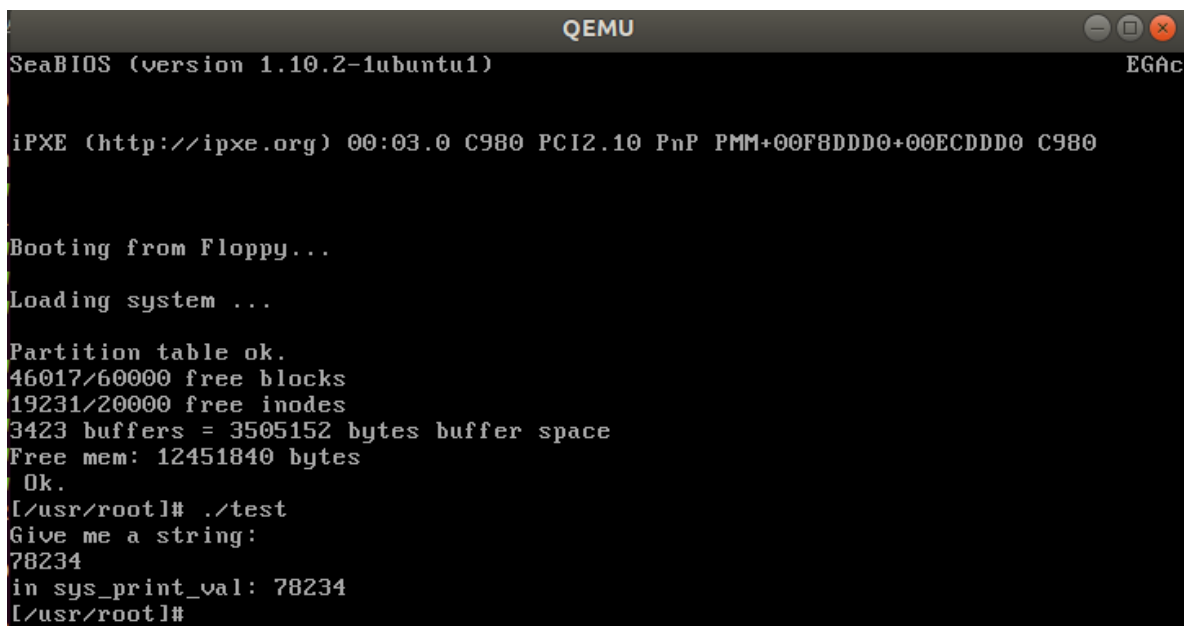
```
vmware@ubuntu:~/oslab/Linux-0.11$ make
make[1]: 进入目录"/home/vmware/oslab/Linux-0.11/boot"
make[1]: 离开目录"/home/vmware/oslab/Linux-0.11/boot"
make[1]: 进入目录"/home/vmware/oslab/Linux-0.11/boot"
make[1]: 离开目录"/home/vmware/oslab/Linux-0.11/boot"
make[1]: 进入目录"/home/vmware/oslab/Linux-0.11/boot"
make[1]: 离开目录"/home/vmware/oslab/Linux-0.11/boot"
init/main.c:23:15: warning: type defaults to 'int' in declaration of 'fork' [-Wimplicit-int]
    static inline fork(void) __attribute__((always_inline));
                ^~~~~~
init/main.c:24:15: warning: type defaults to 'int' in declaration of 'pause' [-Wimplicit-int]
    static inline pause(void) __attribute__((always_inline));
                ^~~~~~
make[1]: 进入目录"/home/vmware/oslab/Linux-0.11/kernel"
In file included from traps.c:13:0:
../include/string.h:46:22: warning: inline function 'memchr' declared but never defined
extern inline void * memchr(const void * cs,char c,int count);
                ^~~~~~
../include/string.h:45:22: warning: inline function 'memmove' declared but never defined
extern inline void * memmove(void * dest,const void * src, int n);
                ^~~~~~
```

```
make[1]: 离开目录"/home/vmware/oslab/Linux-0.11/kernel/blk_drv"
make[1]: 进入目录"/home/vmware/oslab/Linux-0.11/kernel/chr_drv"
sync
make[1]: 离开目录"/home/vmware/oslab/Linux-0.11/kernel/chr_drv"
make[1]: 进入目录"/home/vmware/oslab/Linux-0.11/kernel/math"
make[1]: 离开目录"/home/vmware/oslab/Linux-0.11/kernel/math"
make[1]: 进入目录"/home/vmware/oslab/Linux-0.11/lib"
make[1]: 离开目录"/home/vmware/oslab/Linux-0.11/lib"
记录了1+0 的读入
记录了1+0 的写出
512 bytes copied, 0.000199276 s, 2.6 MB/s
记录了0+1 的读入
记录了0+1 的写出
311 bytes copied, 0.000120705 s, 2.6 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, 0.000699381 s, 221 MB/s
记录了2+0 的读入
记录了2+0 的写出
2 bytes copied, 4.4965e-05 s, 44.5 kB/s
```

⑥编写测试程序

```
1  #define __LIBRARY__
2  #include <unistd.h>
3  #include<stdio.h>
4  #include<errno.h>
5  #include<stdlib.h>
6  _syscall1(void, print_val, int, a);
7  _syscall3(int, str2num, char*, str, int, str_len, long*, ret);
8  int main()
9  {
10     int c;
11     char a[20];
12     long int b[20];
13     int i, x;
14     printf("Give me a string:\n");
15     scanf("%s", a);
16     i = strlen(a);
17     if (str2num(a, i, b) != 1)
18         printf("error!\n");
19     else
20     {
21         c = 0;
22         for (x = 0; x < i; x++)
23             c = (b[x] - 0) + c * 10;
24         print_val(c);
25     }
26     return 0;
27 }
```

⑦运行测试程序



```
QEMU
SeaBIOS (version 1.10.2-1ubuntu1)

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

Booting from Floppy...

Loading system ...

Partition table ok.
46017/60000 free blocks
19231/20000 free inodes
3423 buffers = 3505152 bytes buffer space
Free mem: 12451840 bytes
Ok.
[/usr/root]# ./test
Give me a string:
78234
in sys_print_val: 78234
[/usr/root]#
```

相关问题

1. 简要描述如何在 Linux 0.11 添加一个系统调用。

- ①在 `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。
- ②在 `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。
- ③在 `linux0.11/hdc/usr/include/sys.h` 中添加 `extern` 头，再在 `sys_call_table[]` 中加入系统调用的‘地址’。
- ⑤ 在 `linux-0.11/kernel` 中实现该系统调用，并修改 `Makefile` 文件。

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应用程序调用库函数（API）；
- ②API 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中断处理函数 `system_call` 把参数入栈(`kernel/system_call.s`)

```
call sys_call_table(,%eax,4)
```

- ④根据索引在 `sys_call_table` 中查找函数地址(`include/linux/sys.h`)
- ⑤执行真正的系统调用函数。

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

直接能传递的参数至多有 3 个。在 `Linux-0.11` 中，程序使用 `ebx`、`ecx`、`edx` 这三个通用寄存器保存参数，可以直接向系统调用服务过程传递至多三个参数（不包括放在 `eax` 寄存器中的系统调用号）。如果使用指向用户数据空间的指

针，将指针信息通过寄存器传递给系统调用服务，然后系统调用就可以通过该指针访问用户数据空间中预置的更多数据，就可以达到传递更多参数的目的。

EXP2.2

首先添加 `os_popen` 函数的代码

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

使用 `fork()` 函数创建子进程。

```
/* 2. 子进程部分 */  
if(pid==0)  
{  
    if (type == 'r') {  
        /* 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]);  
        }  
    } else {  
        /* 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]);  
        }  
    }  
}
```

子进程其本身无子进程，返回的 `pid` 为 0

当 `type` 为 `r` 时，此时应该输出其内容，关闭管道读端 `pipe_fd[0]`，并将其对应写端文件描述符传给标准输出 `STDOUT_FILENO`。

当前类型为 `w` 时，此时应向其输入内容，关闭管道写端 `pipe_fd[1]`，并将其对应读端文件描述符传给标准输入 `STDIN_FILENO`。

```
/* 2.3 通过execl系统调用运行命令 */  
if (type == 'r')  
{  
    execl(SHELL, SHELL, "-c", cmd, NULL);  
}  
else  
{  
    execl(SHELL, SHELL, "-c", cmd, STDIN_FILENO, NULL);  
}
```

当为 r 类型时，调用 SHELL 并执行相应命令。

当为 w 类型时，调用 SHELL 执行相应命令，并传入相关输入。

```
/* 3. 父进程部分 */
if (type == 'r') {
    close(pipe_fd[1]);
    proc_fd = pipe_fd[0];
} else {
    close(pipe_fd[0]);
    proc_fd = pipe_fd[1];
}
child_pid[proc_fd] = pid;
return proc_fd;
```

与子进程相反，类型为 r 时关闭写端，函数返回读端；类型为 w 时关闭读端，函数返回写端。

然后添加 os_system 函数。

```
/* 4.1 创建一个新进程 */
pid = fork();

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

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

类似的，先创建子进程，然后子进程使用 execl 调用 SHELL，执行相关命令，父进程等待子进程结束，返回相关状态。


```

/* 5.1 运行cmd1, 并将cmd1标准输出存入buf中 */
fd1 = os_popen(cmd1, 'r');
read(fd1, buffer, 4096);
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, buffer, strlen(buffer));
os_pclose(fd2);

```

实现管道功能，将管道命令的第一个命令的输出利用系统调用 read 读入 buffer，并关闭相应管道；将第二个命令的输入利用系统调用 write 从 buffer 写入，并关闭相应管道。

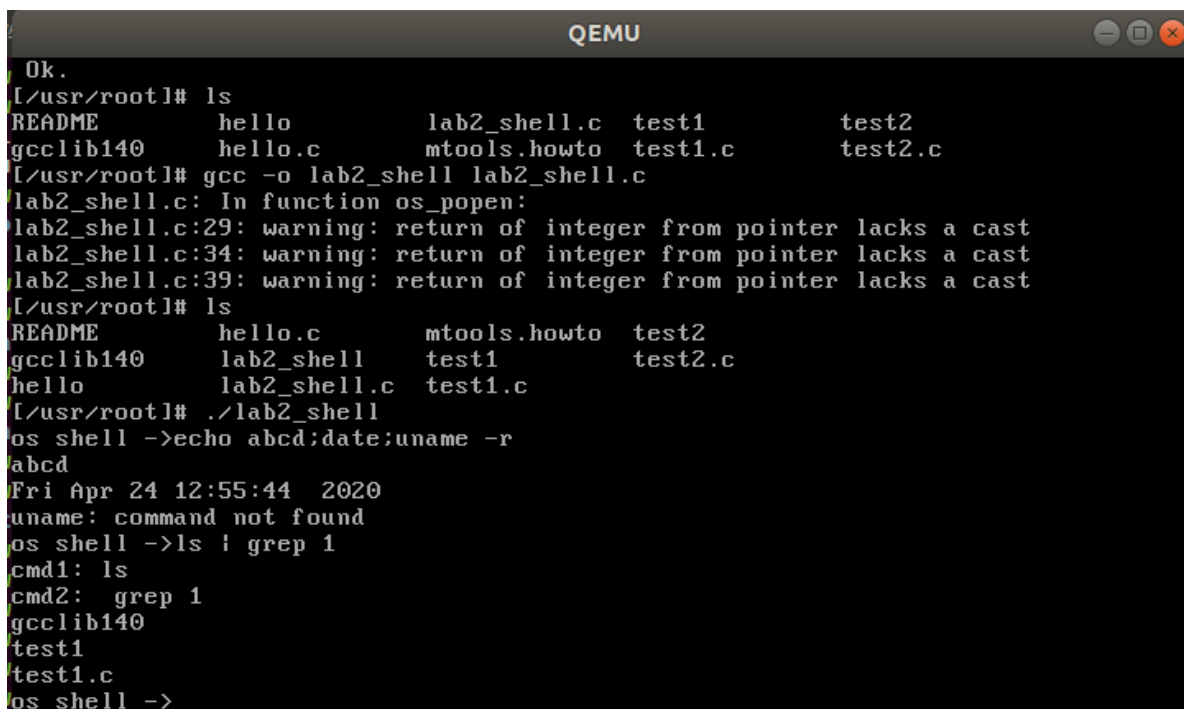
一般的命令，使用 os_system 实现，如果子进程运行错误，输出 ERROR。

```

else {
    /* 6 一般命令的运行 */
    status=os_system(cmds[i]);
    if(status!=0)
        printf("ERROR!\n");
}

```

再 Linux-0.11 中的实现效果：



```

Ok.
[/usr/root]# ls
README      hello      lab2_shell.c  test1      test2
gcclib140   hello.c    mtools.howto test1.c     test2.c
[/usr/root]# gcc -o lab2_shell lab2_shell.c
lab2_shell.c: In function os_popen:
lab2_shell.c:29: warning: return of integer from pointer lacks a cast
lab2_shell.c:34: warning: return of integer from pointer lacks a cast
lab2_shell.c:39: warning: return of integer from pointer lacks a cast
[/usr/root]# ls
README      hello.c    mtools.howto test2
gcclib140   lab2_shell test1      test2.c
hello      lab2_shell.c test1.c
[/usr/root]# ./lab2_shell
os shell ->echo abcd:date:uname -r
abcd
Fri Apr 24 12:55:44 2020
uname: command not found
os shell ->ls | grep 1
cmd1: ls
cmd2:  grep 1
gcclib140
test1
test1.c
os shell ->

```