

《操作系统原理与设计》 实验报告



实验题目： 添加 Linux系统调用
及熟悉常见系统调用

学生姓名： 王志强

学生学号： PB18051049

完成日期： 2020.04.20

计算机实验教学中心制

2019年9月

实验目的

- 了解系统调用的基本过程
- 学习如何添加Linux系统调用
- 熟悉Linux下常见的系统调用

实验环境

- OS: Ubuntu 18.04 LTS
- Linux内核版本: Kernel 0.11

实验内容

一、添加Linux系统调用

目标: 了解Linux是如何实现系统调用功能, 并在Linux 0.11中添加两个系统调用

1、分配系统调用号, 修改系统调用表

- 增加的两个系统调用对应函数的形式和功能要求如下:

```
1 | int print_val(int a);    //通过printf在控制台打印如下信息(假设a是1234): in
   | sys_print_val: 1234
2 | 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).
```

- 修改system_call.s文件中的系统调用数量(nr_system_call参数):

```
1 | nr_system_calls = 74    /*原先为72*/
```

- 增加系统调用编号并同步增加系统调用原型(也需挂载文件系统后在/hdc/usr中修改):

```
1 | #define __NR_print_val 72    /*增加系统调用编号*/
2 | #define __NR_str2num 73
3 |
4 | int print_val(int a);    /*添加*/
5 | int str2num(char *str, int str_len, long *ret);    /*添加*/
```

- 修改函数指针表并增加函数原型:

```
1 | extern int sys_print_val();    /*添加*/
2 | extern int sys_str2num();
3 | fn_ptr sys_call_table[] = { sys_setup, ..., sys_print_val, sys_str2num }; /*
   | 指针表*/
```

2、实现系统调用函数

- 新建kernel/EXP2.c, 完成两个系统调用的具体实现:

```
1 | #define __LIBRARY__
2 | #include<linux/kernel.h>
3 | #include<asm/segment.h>    /*定义了get_fs_byte()和set_fs_byte()*/
```

```

4  int sys_print_val(int a){
5      printk("in sys_print_val:%d\n",a);
6      return 0;
7  }
8  /*作为os实验，重在实现系统调用，不考虑负数情况*/
9  int sys_str2num(char *str,int str_len,long *ret){
10     if(str_len >=8)
11         return -1; /*超过10^8*/
12     int i;
13     int result = 0;
14     for(i=0;i<str_len;i++)
15         result = 10*result+(get_fs_byte(str+i)-'0');
16     put_fs_long(result,ret);
17     return 0;
18 }

```

- 修改Makefile文件

```

1  OBJS = sched.o system_call.o traps.o asm.o fork.o \
2      panic.o printk.o vsprintf.o sys.o exit.o \
3      signal.o mktime.o EXP2.o  #第一处，添加了EXP.o
4
5  EXP2.s EXP2.o: EXP2.c ../include/linux/kernel.h ../include/asm/segment.h
6      #第二处

```

3、编译内核

- 执行如下命令编译内核：

```

1  make clean
2  make

```

4、编写测试程序

- 创建一个简单的用户程序（文件名为test.c）验证已实现的系统调用正确性，要求能够从终端读取一串数字字符串，通过str2num系统调用将其转换成数字，然后通过print_val系统调用打印该数字。为避免混乱，执行用户程序后需要有如下输出：

```

1  Give me a string:
2  78234
3  in sys_print_val: 78234

```

- 编写的测试程序如下：

```

1  #define __LIBRARY__ /*_syscallx生效*/
2  #include <unistd.h>
3  #include<stdio.h>
4  #include<string.h>
5
6  _syscall1(int,print_val,int,a); /*用户空间的接口函数*/
7  _syscall3(int,str2num,char *,str,int,str_len,long *,ret);
8
9  int main(){
10     char str[10];
11     long a;

```

```

12     long *ret = &a;
13     printf("Give me a string:\n");
14     scanf("%s",str);
15     str2num(str,strlen(str),ret);    /*调用str2num*/
16     print_val((int)a);    /*调用print_val*/
17     return 0;
18 }

```

- 编译测试程序并运行:

testcase:

```

1 78243
2 520
3 0001

```

```

[/usr]# ls
bin      include  root      shell.c   test      test.txt  var
docs     local    shell     src       test.c    tmp
[/usr]# gcc test.c -o test
[/usr]# ./test
Give me a string:
78234
in sys_print_val:78234
[/usr]# ./test
Give me a string:
520
in sys_print_val:520
[/usr]# ./test
Give me a string:
0001
in sys_print_val:1

```

5、回答问题

- Q: 简要描述如何在Linux 0.11添加一个系统调用

A: 首先, 应当分配系统调用号并修改系统调用表, 其中包括修改系统调用数量(nr_system_call参数)、增加系统调用编号并同步函数原型(挂载后重复进行一次)、修改函数指针表并增加函数原型; 其次, 创建文件具体实现系统调用函数并修改Makefile文件, 重新编译内核; 最后, 编写测试程序检验所实现的系统调用。

- Q: 系统是如何通过系统调用号索引到具体的调用函数的?

A: 通过_syscallx宏, 将传入的参数替换成了一个函数定义, 将系统调用索引号存入EAX寄存器, 然后触发x86中断调用, 之后自动调用函数system_call, 根据 `call sys_call_table(,%eax,4)` 去sys_call_table索引到具体的调用函数。

- Q: 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?

A: 由系统调用宏_syscall3可知, Linux 0.11中最多支持三个参数; 超过限制的办法: 如果寄存数量足够, 那么继续编写调用宏_syscall4、_syscall5...即可; 或者, 可以将多个参数打包为一个结构体, 将结构体传入即可达到传入多个参数的效果。

二、熟悉Linux下常见的系统调用函数

目标：利用Linux提供的系统调用，实现一个简单的shell程序

1、熟悉系统调用函数的用法

```
1  /*本实验中可用到的系统调用:*/
2  int read(int fildes, char * buf, off_t count); /*从fildes对应的文件中读取count
   个字符
3  到buf内（fildes为文件描述符）*/
4  int write(int fildes, const char * buf, off_t count); /*从buf写count个字符到
   fildes
5  对应的文件中*/
6  pid_t fork(); /*创建进程*/
7  pid_t waitpid(pid_t pid,int* status,int options); /*等待指定pid的子进程结束*/
8  int execl(const char *path, const char *arg, ...); /*根据指定的文件名或目录名找
   到可
9  执行文件，并用它来取代原调用进程的数据段、代码段和堆栈段，在执行完之后，原调用进程的内容除
   了进程
10  号外，其他全部被新程序的内容替换了*/
```

2、使用上述系统调用实现三个进程创建相关的函数(设计思路见注释)

```
1  int os_system(const char* command); /*调用fork()产生子进程，在子进程执行参数
   command字
2  符串所代表的命令，此命令**执行完后**随即返回原调用的进程*/
3  int os_popen(const char* command, const char mode); /*popen函数先执行fork，然后
   调用
4  exec以执行command，并且根据mode的值（'r'或'w'）返回一个指向子进程的stdout或指向stdin的
   文件
5  指针*/
6  int os_pclose(const int pno); /*关闭标准I/O流，等待命令执行结束，与popen对应*/
```

• os_open函数:

```
1  /* popen, 输入为命令和类型("r""w"), 输出执行命令进程的I/O文件描述符 */
2  int os_popen(const char* cmd, const char type){
3      int i, pipe_fd[2], proc_fd;
4      pid_t pid;
5
6      if (type != 'r' && type != 'w') {
7          printf("popen() flag error\n");
8          return NULL;
9      }
10     if(child_pid == NULL) {
11         if ((child_pid = (int *)calloc(NR_TASKS, sizeof(int))) == NULL)
12         {
13             printf("what's matter?\n");
14             return NULL;
15         }
16     }
17     if (pipe(pipe_fd) < 0) {
18         printf("popen() pipe create error\n");
19         return NULL;
20     }
21     /* 1. 使用系统调用创建新进程 */
```

```

21     if((pid = fork())<0){ /*思路: fork函数创建子进程, 判断是否成功创建*/
22     printf("child process creat error\n");
23     return NULL; /*创建失败*/
24     }
25     /* 2. 子进程部分 */
26     else if(pid == 0){
27         if (type == 'r') {
28             /* 2.1 关闭pipe无用的一端, 将I/O输出发送到父进程 */
29             close(pipe_fd[0]); /*思路: 端口0为读, 端口1为写*/
30             if (pipe_fd[1] != STDOUT_FILENO) { /*'r'时, 子写父读*/
31                 dup2(pipe_fd[1], STDOUT_FILENO); /*写入端口定向到标准输出
32             */
33                 close(pipe_fd[1]);
34             }
35         } else {
36             /* 2.2 关闭pipe无用的一端, 接收父进程提供的I/O输入 */
37             close(pipe_fd[1]); /*思路: 端口0为读, 端口1为写*/
38             if (pipe_fd[0] != STDIN_FILENO) { /*'w'时, 子读父写*/
39                 dup2(pipe_fd[0], STDIN_FILENO); /*读出端口定向到标准输入*/
40                 close(pipe_fd[0]);
41             }
42         }
43         /* 关闭所有未关闭的子进程文件描述符 (无需修改) */
44         for (i=0;i<NR_TASKS;i++)
45             if(child_pid[i]>0)
46                 close(i);
47         /* 2.3 通过exec1系统调用运行命令 */
48         exec1(SHELL,"sh","-c",cmd,NULL);/*思路: 成功创建, 用exec1函数覆盖子
49         进程, 实现shell命令*/
50         _exit(127);
51     }
52     /* 3. 父进程部分 */
53     if (type == 'r') { /*思路: 和子进程行为对应*/
54         close(pipe_fd[1]); /*子写父读*/
55         proc_fd = pipe_fd[0];
56     } else {
57         close(pipe_fd[0]); /*子读父写*/
58         proc_fd = pipe_fd[1];
59     }
60     child_pid[proc_fd] = pid;
61     return proc_fd;
62 }

```

- **os_pclose函数:**

框架中已给出完整的实现, 无需修改、补充

- **os_system函数:**

```

1  int os_system(const char* cmdstring) {
2      pid_t pid;
3      int stat;
4      if(cmdstring == NULL) {
5          printf("nothing to do\n");
6          return 1;
7      }

```

```

8      /* 4.1 创建一个新进程 */
9      if ((pid = fork()) < 0) /*思路: fork函数创建子进程, 判断是否成功创建*/
10     {
11         printf("child process create error\n"); /*创建失败*/
12         return -1;
13     }
14     /* 4.2 子进程部分 */
15     else if(pid == 0) { /*思路: 成功创建, 用exec1函数覆盖子进程, 实现
shell命令*/
16         exec1(SHELL, "sh", "-c", cmdstring, NULL);
17         _exit(127);
18     }
19     /* 4.3 父进程部分: 等待子进程运行结束 */
20     /*思路: 当waitpid收集到已退出的子进程时waitpid返回被等待进程的ID*/
21     if(waitpid(pid, &stat, 0) != pid)
22         stat = -1; /*异常返回*/
23     return stat;
24 }

```

3、利用上面的函数实现一个简单的shell程序

- 结合上述函数, shell.c的main函数如下:

```

1  /*设计思路见注释*/
2  int main() {
3      int      cmd_num, i, j, fd1, fd2, status; /*删除了count变量*/
4      pid_t    pids[MAX_CMD_NUM];
5      char      cmdline[MAX_CMDLINE_LENGTH];
6      char      cmds[MAX_CMD_NUM][MAX_CMD_LENGTH];
7      char      buf[BUFF_SIZE]; /*修改buf为字符数组, 而不是指针数组*/
8      char      *div = NULL; /*在gcc1.4下编译, 变量定义放在开头*/
9      char      cmd1[MAX_CMD_LENGTH], cmd2[MAX_CMD_LENGTH];
10     int      len;
11     while(1){
12         /* 将标准输出文件描述符作为参数传入write, 即可实现print */
13         write(STDOUT_FILENO, "os shell ->", 11);
14         gets(cmdline);
15         if(strcmp(cmdline, "goodbye")==0){ /*补充的退出shell命令*/
16             printf("Thank you for using!\n");
17             break; /*退出shell*/
18         }
19         cmd_num = parseCmd(cmdline, cmds); /*划分命令字符串*/
20         for(i=0; i<cmd_num; i++){
21             div = strchr(cmds[i], '|');
22             if (div) {
23                 /* 如果需要用到管道功能 */
24                 len = div - cmds[i];
25                 memcpy(cmd1, cmds[i], len);
26                 cmd1[len] = '\0';
27                 len = (cmds[i] + strlen(cmds[i])) - div - 1;
28                 memcpy(cmd2, div + 1, len);
29                 cmd2[len] = '\0';
30                 printf("cmd1: %s\n", cmd1);
31                 printf("cmd2: %s\n", cmd2);
32                 /* 5.1 运行cmd1, 并将cmd1标准输出存入buf中 */
33                 zeroBuff(buf, BUFF_SIZE); /*buf清0*/
34                 fd1 = os_popen(cmd1, 'r');

```

```

35         if(fd1 == 0)    /*os_popen执行失败*/
36             printf("%s:os_popen excute failed!\n",cmd1);
37
38     else{
39         read(fd1,buf,BUFF_SIZE);    /*标准输出写入到buf*/
40         os_pclose(fd1); /*关闭标准I/O流*/
41         /* 5.2 运行cmd2, 并将buf内容写入到cmd2输入中 */
42         fd2 = os_popen(cmd2,'w');
43         if(fd2 == 0)    /*os_popen执行失败*/
44             printf("%s:os_popen excute failed!\n",cmd2);
45         else
46             write(fd2,buf,BUFF_SIZE); /*buf内容写到cmd2输入
47
48     */
49         os_pclose(fd2); /*关闭标准I/O流*/
50     }
51 }
52 else {
53     /* 6 一般命令的运行 */
54     if(status = os_system(cmds[i]) < 0) /*直接调用os_system即可*/
55         printf("%s:excute failed!",cmds[i]);
56 }
57 }
58 return 0;
59 }

```

4、Linux 0.11下的编译与运行 (主机和Linux 0.11均运行成功>)

- 首先, 在Ubuntu主机上进行测试, 运行结果如下:

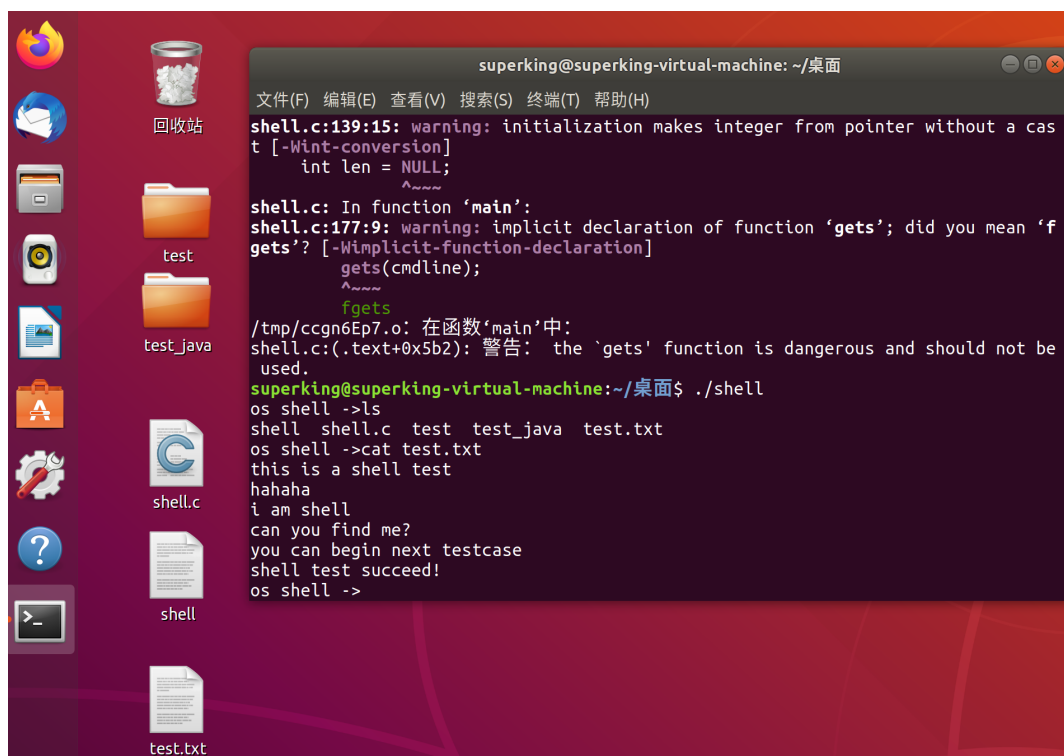
- 单命令测试

testcase:

```

1  ls
2  cat test.txt

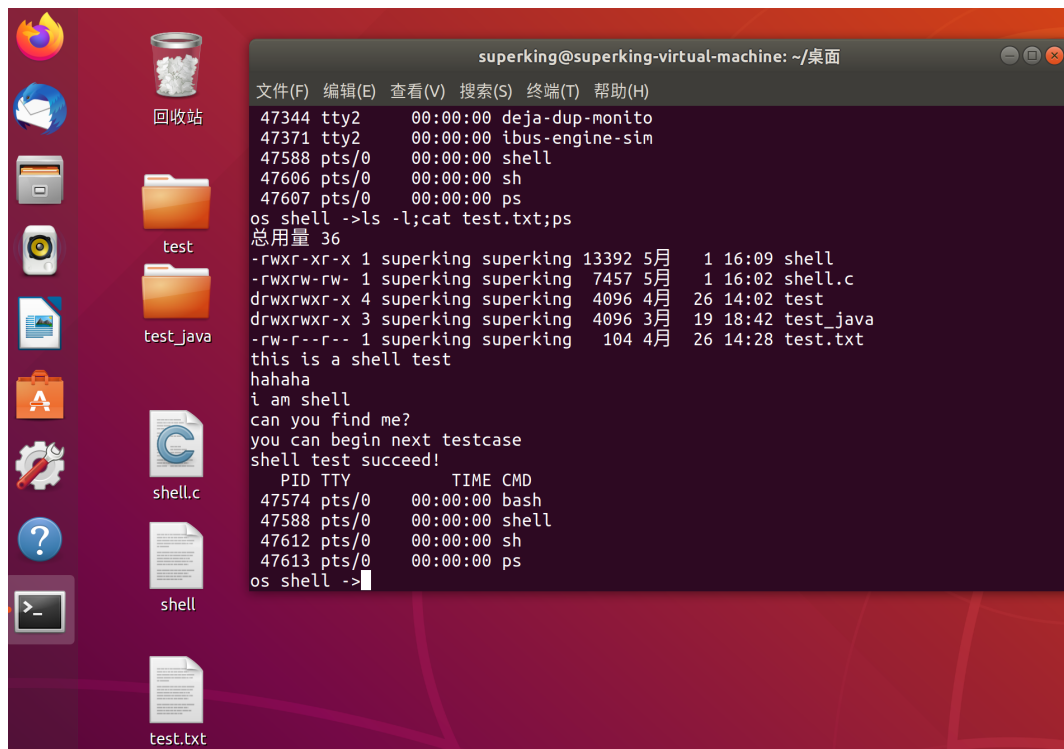
```



○ 子命令测试

testcase:

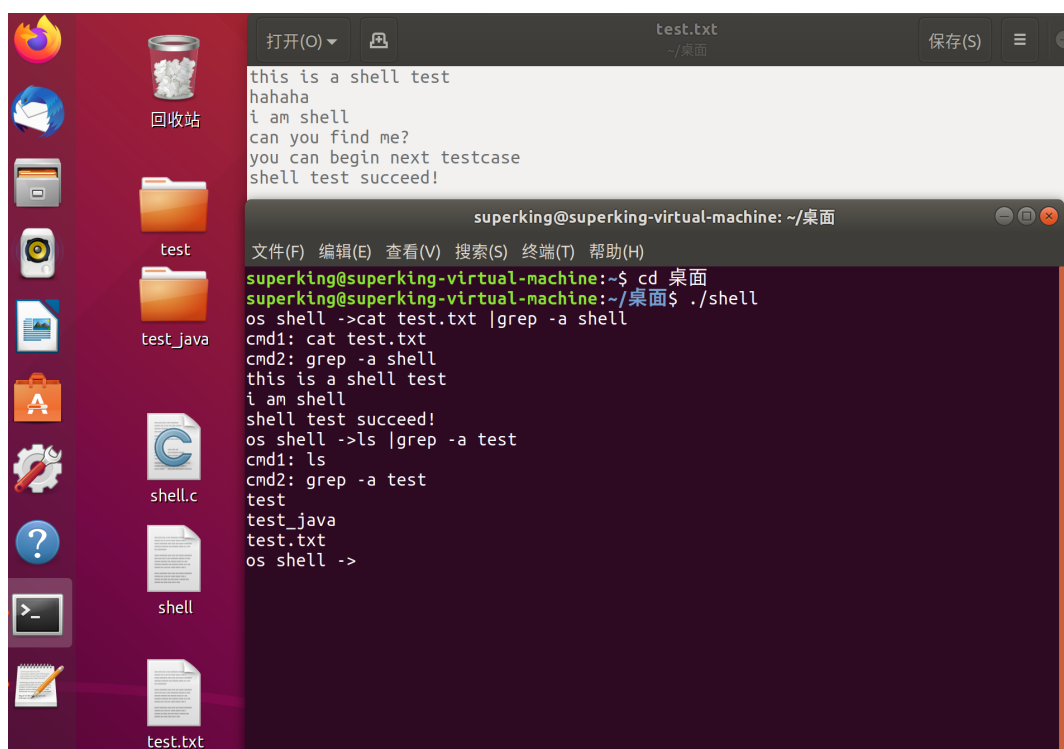
```
1 | ls -l; cat test.txt; ps
```



○ 管道测试:

testcase:

```
1 | cat test.txt | grep -a shell
2 | ls | grep -a test
```

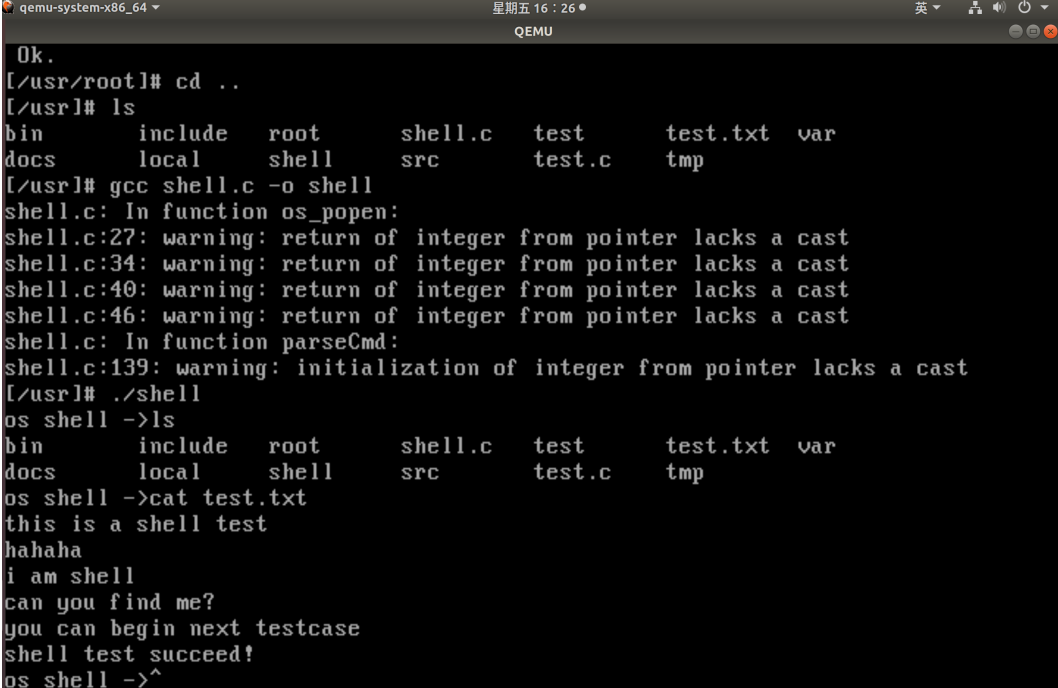


- 在Linux 0.11中测试shell程序，运行结果如下：

- 单命令测试

testcase:

```
1 | ls
2 | cat test.txt
```

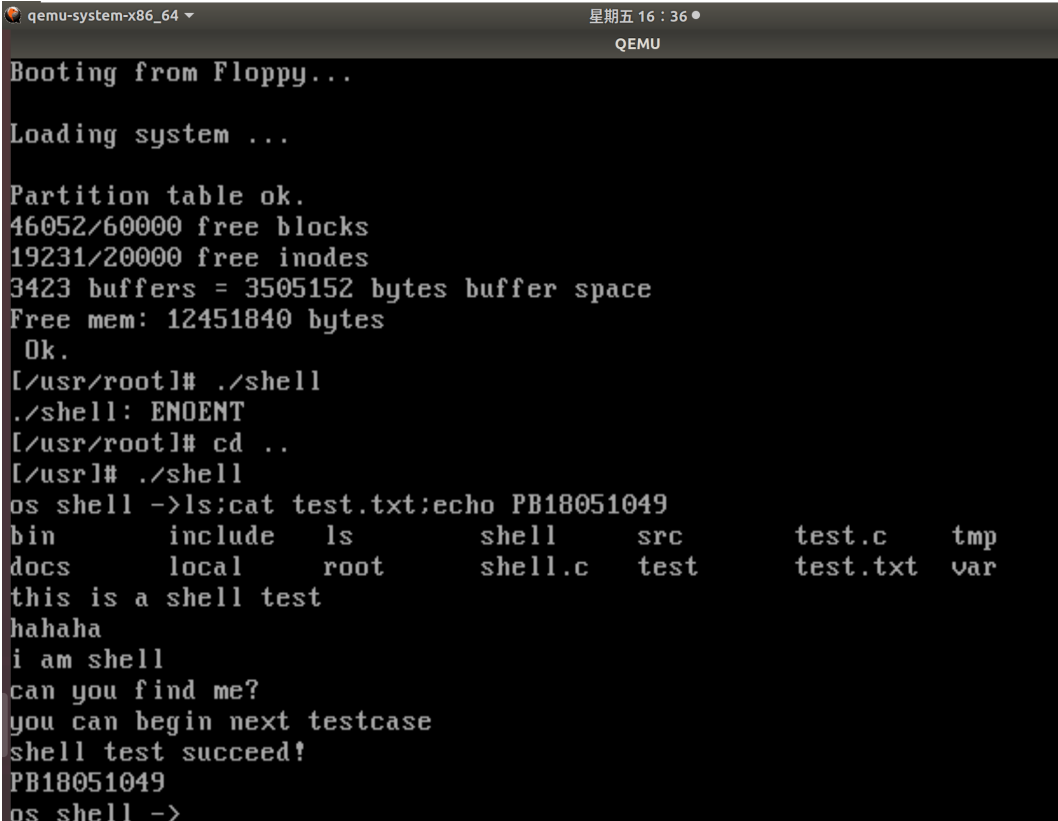


```
Ok.
[/usr/root]# cd ..
[/usr]# ls
bin      include  root      shell.c   test      test.txt  var
docs     local    shell     src       test.c    tmp
[/usr]# gcc shell.c -o shell
shell.c: In function os_popen:
shell.c:27: warning: return of integer from pointer lacks a cast
shell.c:34: warning: return of integer from pointer lacks a cast
shell.c:40: warning: return of integer from pointer lacks a cast
shell.c:46: warning: return of integer from pointer lacks a cast
shell.c: In function parseCmd:
shell.c:139: warning: initialization of integer from pointer lacks a cast
[/usr]# ./shell
os shell ->ls
bin      include  root      shell.c   test      test.txt  var
docs     local    shell     src       test.c    tmp
os shell ->cat test.txt
this is a shell test
hahaha
i am shell
can you find me?
you can begin next testcase
shell test succeed!
os shell ->^_
```

- 子命令测试

testcase:

```
1 | ls; cat test.txt; echo PB18051049
```



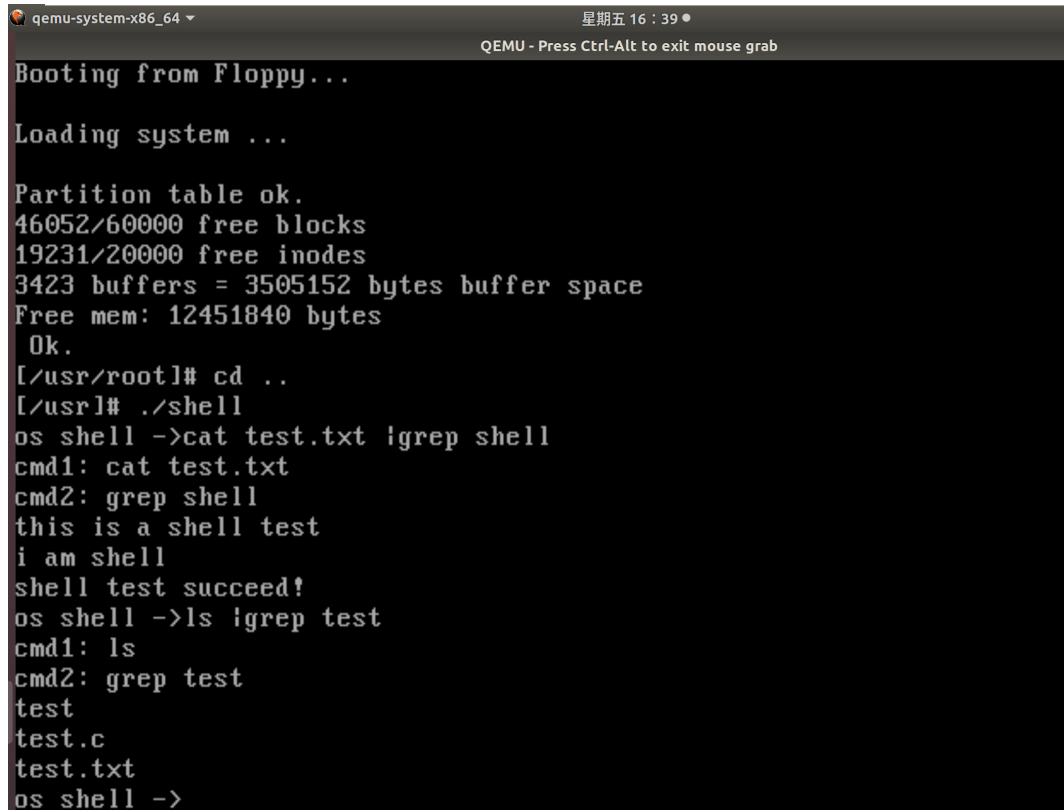
```
Booting from Floppy...
Loading system ...

Partition table ok.
46052/60000 free blocks
19231/20000 free inodes
3423 buffers = 3505152 bytes buffer space
Free mem: 12451840 bytes
Ok.
[/usr/root]# ./shell
./shell: ENOENT
[/usr/root]# cd ..
[/usr]# ./shell
os shell ->ls;cat test.txt;echo PB18051049
bin      include  ls        shell     src       test.c    tmp
docs     local    root      shell.c   test      test.txt  var
this is a shell test
hahaha
i am shell
can you find me?
you can begin next testcase
shell test succeed!
PB18051049
os shell ->
```

○ 管道测试

testcase:

```
1 | cat test.txt | grep shell
2 | ls | grep test
```



The screenshot shows a QEMU virtual machine window titled 'qemu-system-x86_64'. The terminal output is as follows:

```
Booting from Floppy...
Loading system ...

Partition table ok.
46052/60000 free blocks
19231/20000 free inodes
3423 buffers = 3505152 bytes buffer space
Free mem: 12451840 bytes
Ok.
[/usr/root]# cd ..
[/usr]# ./shell
os shell ->cat test.txt |grep shell
cmd1: cat test.txt
cmd2: grep shell
this is a shell test
i am shell
shell test succeed!
os shell ->ls |grep test
cmd1: ls
cmd2: grep test
test
test.c
test.txt
os shell ->
```

5、总结

- 第二部分实验所有内容均顺利完成，测试结果如上
- 熟悉了进程创建相关内容，掌握了简单shell的架构方法
- pipe的使用花费较多时间和精力，深刻地理解了操作系统的管道通信