实验报告(添加Linux系统调用及熟悉常见系统调用)

PB17000289 于佳睿

实验目的

- 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 14.04 i386（32位）
- Linux内核版本：Kernel 2.6.26

实验步骤

一、添加**Linux**系统调用

- 首先分配系统调用号，在Linux源代码根目录下，找到include/asm/unistd_32.h文件，在文件末尾找到最大的已分配系统调用号，这里为"print_val"分配327，为"str2num"分配328。

```
#define __NR_print_val 327
#define __NR_str2num   328
```

- 然后修改系统调用映射表

```
.long sys_print_val
.long sys_str2num
```

- 最后，声明并添加系统调用

1. 在syscalls.h文件中添加声明

```
asmlinkage void sys_print_val(int a);
asmlinkage void sys_str2num(char __user *str, int str_len, int __user *ret);
```

2. 在sys.c文件中添加函数

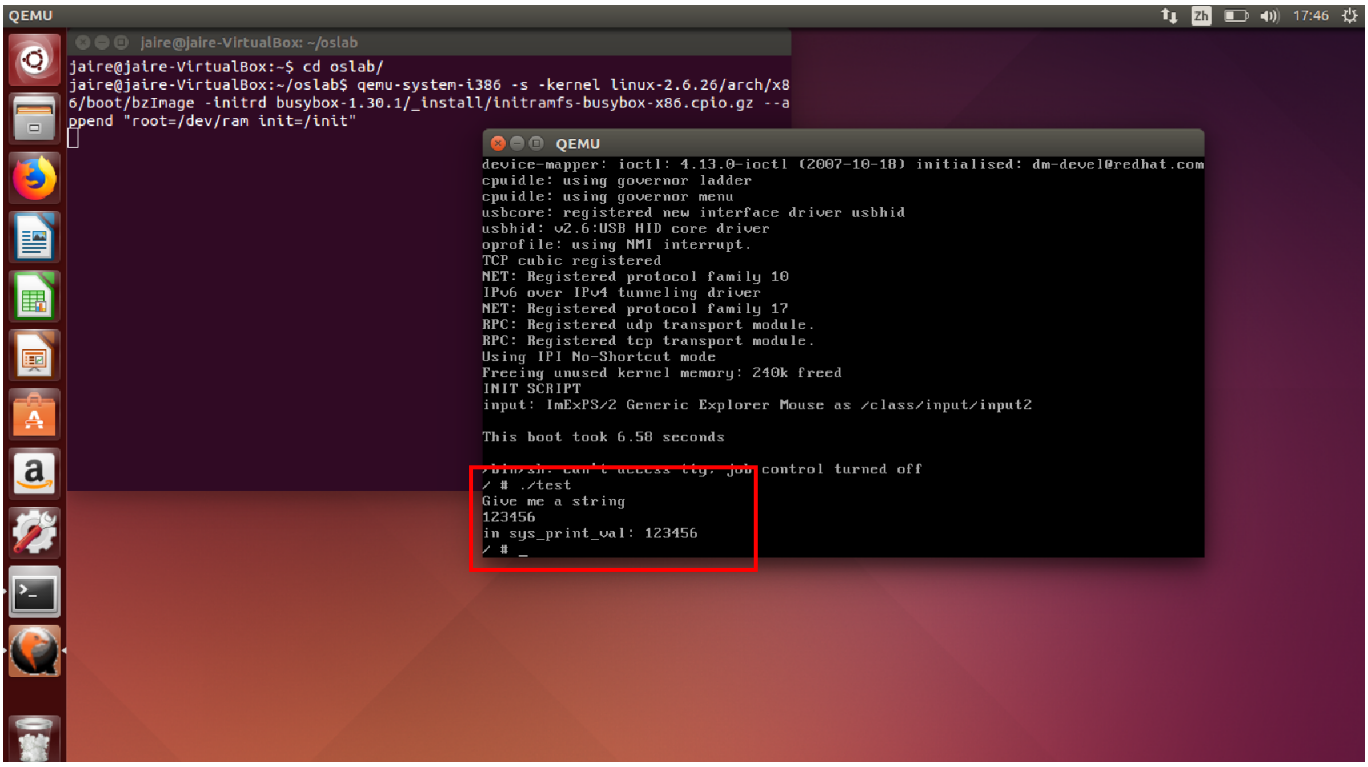
```
asmlinkage void sys_print_val(int a){
        printk("in sys_print_val: %d\n",a);
        return;
```

```
}
asmlinkage void sys_str2num(char __user *str, int str_len, int __user *ret){
        char strtmp[100];
        copy_from_user(strtmp, str, str_len*sizeof(char));
        int count = 0;
        if(str_len == 0)
                return;
        int i = 0;
        while(i != str_len){
                count = count*10 + strtmp[i]-'0';
                i ++;
        }
        copy_to_user(ret, &count, sizeof(int));
        return;
}
```

- 编写test文件,并编译

```
#include<stdio.h>
#include<unistd.h>
#include<sys/syscall.h>
int main(){
        char str[100];
        int num;
        int len;
        printf("Give me a string\n");
        scanf("%s", str);
        for(len = 0; str[len] !='\0'; len ++){
        }
        syscall(328, str, len, &num);
        syscall(327, num);
}
```

- 重新编译内核，重新打包_install，运行qemu观察测试结果



二、熟悉Linux下常见的系统调用

- 思路设计

1. 首先对命令根据";"分成子串，并对子串计数
2. 顺序执行子串：首先判断有没有管道符"|"，若有则执行3，否则执行4
3. 有管道的情况，执行左面命令，将执行结果读入buffer，将buffer写入右面指令执行时的参数中
4. 无管道的情况，根据空格分词，形成字符串数组，fork()一个子进程，在子进程中用execvp根据字符串数组参数来执行

- 实验代码

```
#include<string.h>
#include<stdlib.h>
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/syscall.h>
#include<sys/wait.h>
void sepspace(char* str, char **cmd){
    int i = 0;
    cmd[0] = strtok(str, " ");
    for(i = 1; i < 10; i ++){
        cmd[i] = strtok(NULL, " ");
    }
}
int sepsemicolon(char *cmdline, char **cmd){
    int i = 0;
    int num = 0;
    cmd[0] = strtok(cmdline, ";");
    num = num + 1;
```

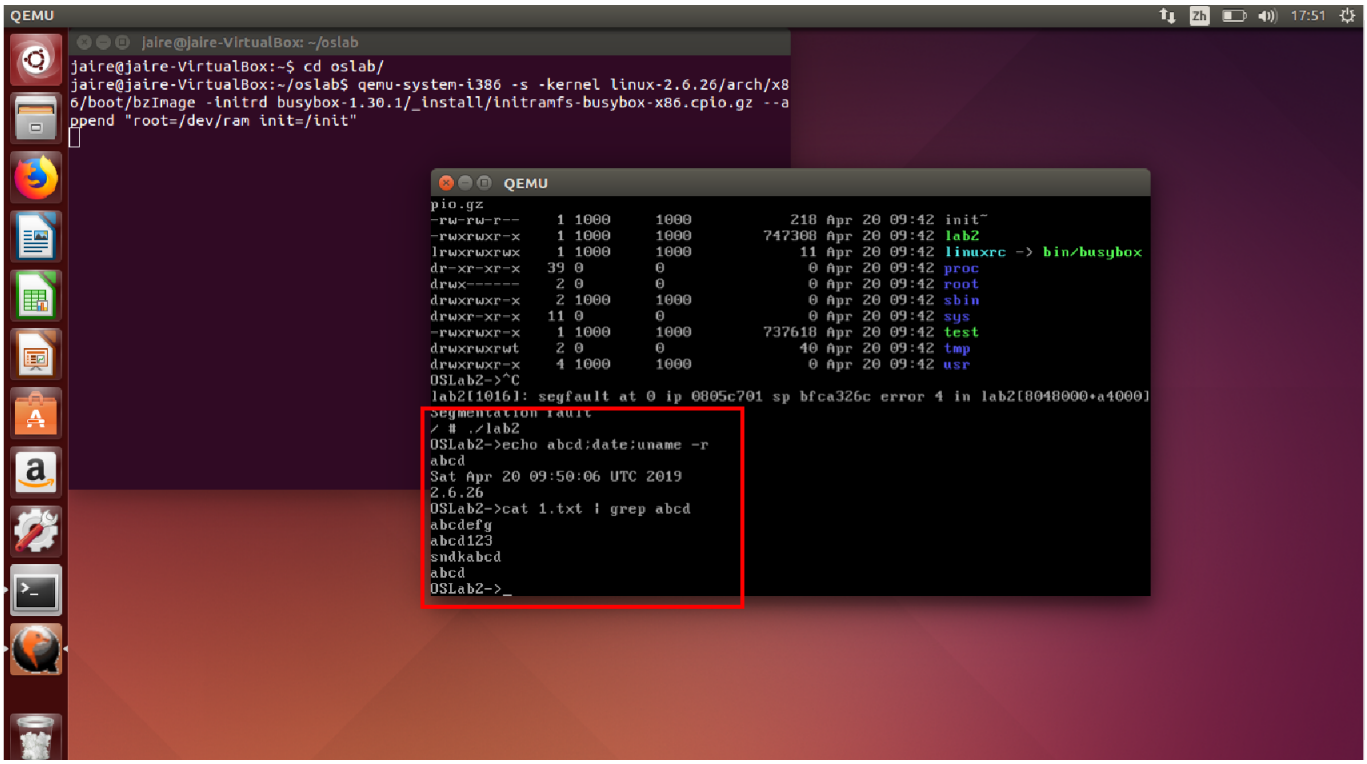
```
    for(i = 1; i < 10; i++){
        cmd[i] = strtok(NULL, ";");
        if(cmd[i] != NULL)
            num = num + 1;
    }
    return num;
}
void seppipe(char* str, char **cmd){
    int i = 0;
    cmd[0] = strtok(str, "|");
    cmd[1] = strtok(NULL, "|");
}
int main(){
    char cmdline[256];
    while (1) {
        printf("OSLab2->");
        gets(cmdline);
        int cmd_num = 0;
        char *cmd[10];
        int i = 0;
        int j = 0;
        int k = 0;
        cmd_num = sepsemicolon(cmdline, cmd);
        for (i=0; i < cmd_num; i++) {
            int pipe_index = 0;
            int pipe = 0;
            for(j = 0; j != strlen(cmd[i]); j ++){
                if(cmd[i][j] == '|'){
                    pipe = 1;
                    break;
                }
            }
            if (pipe) { //处理包含一个管道符号“|”的情况,利用popen处理命令的输入输出转
换
                char *pipe_cmd[2];
                seppipe(cmd[i], pipe_cmd);
                char buffer[10000];
                FILE *fr = popen(pipe_cmd[0], "r");
                FILE *fw = popen(pipe_cmd[1], "w");
                if(fr != NULL && fw != NULL){
                    int successnum = 0;
                    successnum = fread(buffer, sizeof(char), 1000, fr);
                    if(successnum == 10000){
                        printf("Buffer overflow...Cancel the command\n");
                        pclose(fr);
                        pclose(fw);
                    }
                    else if(fw != NULL){
                        fwrite(buffer, sizeof(char), 1000, fw);
                        pclose(fr);
                        pclose(fw);
                    }
                }
                else{
```

```
                    printf("right or left command error!\n");
                    if(fr != NULL)
                        pclose(fr);
                    if(fw != NULL)
                        pclose(fw);
                }
            }
            else {
                char *cmd_nopipe[10];
                int count = 0;
                sepspace(cmd[i], cmd_nopipe);
                pid_t child_pid = fork();
                if(child_pid == -1){
                    printf("Failed to fork a child process\n");
                }
                else if(child_pid == 0){
                    //child_process
                    if(execvp(cmd_nopipe[0], cmd_nopipe)<0)
                        printf("Cannot execute the %d command\n", i);
                    exit(0);
                }
                else{
                    //parent_process
                    int status;
                    status = waitpid(child_pid, &status, 0);
                    if(status == 0)
                        printf("Failed to execute child process\n");
                    else if(status == -1)
                        printf("There are errors while executing child
process\n");
                    else if(status != child_pid)
                        printf("Executing wrong child...please try again\n");
                }
            }

        }
    }

    return 0;
}
```

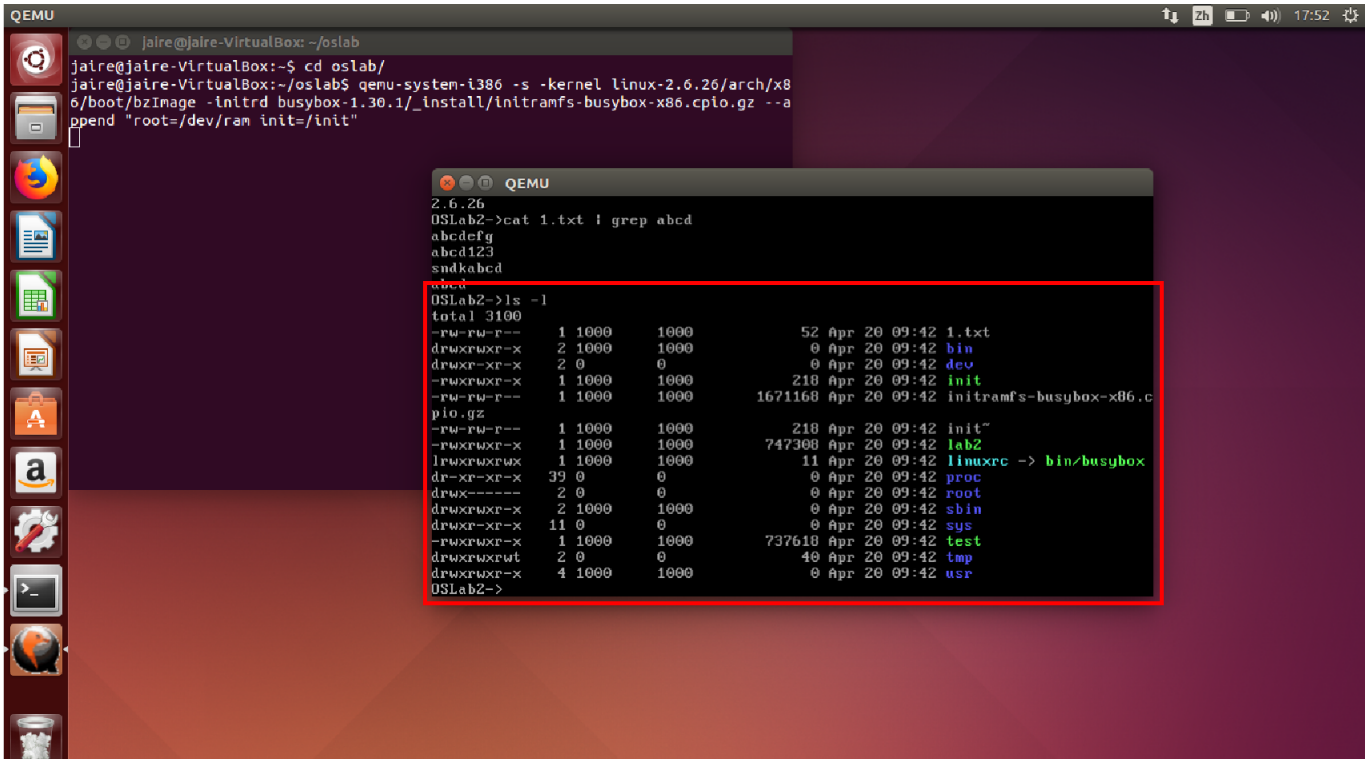
- 实验代码解释

1. 字符串分割是通过"strtok"函数完成的
2. 在循环中加入了很多执行失败的判断，有的是通过返回值，有的是通过指针是否为NULL
3. 使用了popen,pclose,exec,fork,waitpid等系统调用

- 实验结果展示



多命令和管道测试



单命令测试

实验总结

- 通过本次实验我了解了kernel寻找系统调用的流程，和添加系统调用的方式

> 一个函数声明加上asmlinkage,则表示这是一个系统调用,然后会通过宏定义中的系统调用号来寻找syscall_table文件中的相应部分来实现跳转

- 通过本次shell的编写，我理解了fork，exec系统调用的重要性和主要的用处，学习了exec族各个调用的区别
- 我还了解了静态编译与动态编译的区别

> 动态编译的可执行文件需要附带一个的动态链接库，在执行时，需要调用其对应动态链接库中的命令。所以其优点一方面是缩小了执行文件本身的体积，另一方面是加快了编译速度，节省了系统资源。缺点一是哪怕是很简单的程序，只用到了链接库中的一两条命令，也需要附带一个相对庞大的链接库；二是如果其他计算机上没有安装对应的运行库，则用动态编译的可执行文件就不能运行。
>
> 静态编译就是编译器在编译可执行文件的时候，将可执行文件需要调用的对应动态链接库(.so)中的部分提取出来，链接到可执行文件中去，使可执行文件在运行的时候不依赖于动态链接库。所以其优缺点与动态编译的可执行文件正好互补。
>
> 来源：CSDN