操作系统实验二

添加Linux系统调用及熟悉常见系统调用

主要步骤几核心代码

添加Linux系统调用

源代码

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>


int main(){
    char str[20];
    printf("Give me a string:\n");
    fgets(str,20,stdin);
    int num;
    int str_len = strlen(str);
    int a = syscall(328, str, str_len-1,&num);//调用新增的str2num系统调用

    int b = syscall(327, num);//调用新增的print_val系统调用
    return 0;
}
```

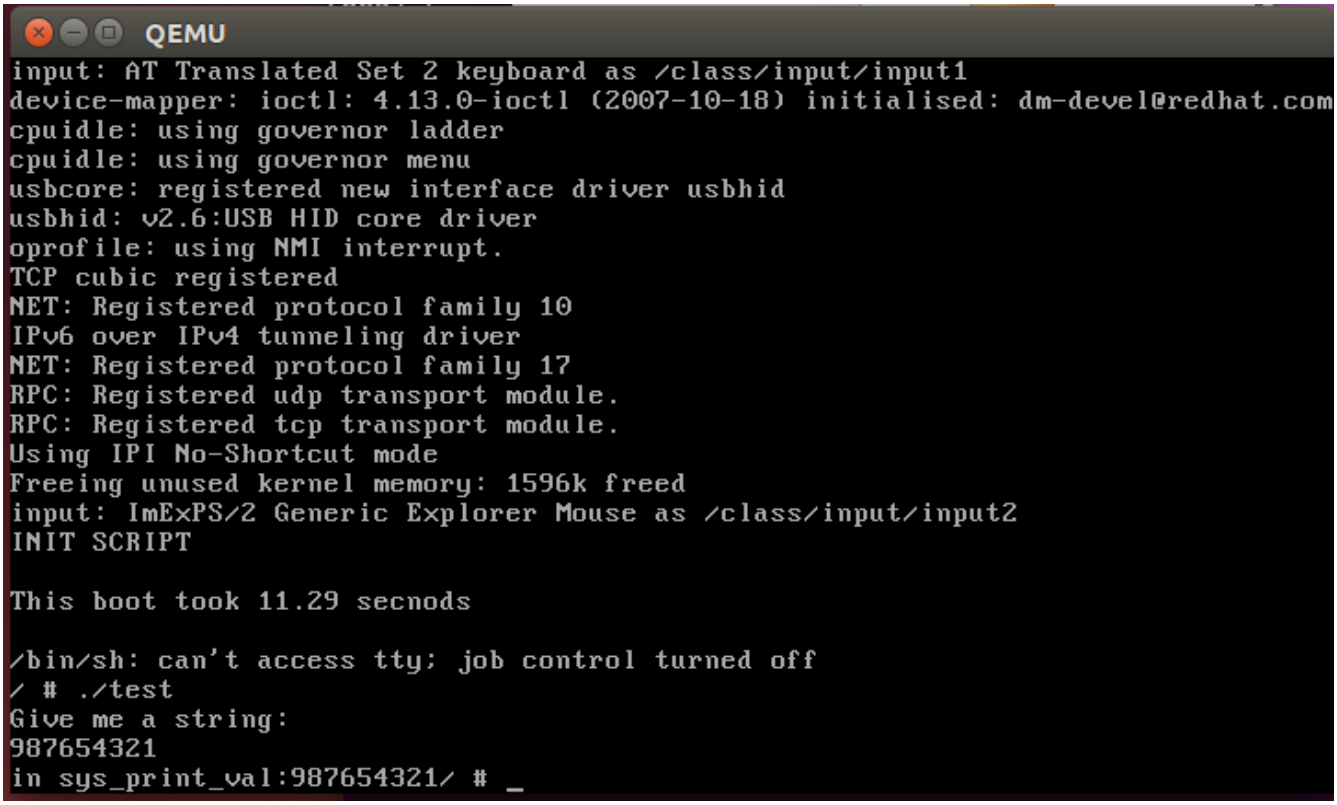
str2num的实现

```c
asmlinkage void sys_str2num(char __user *str, int str_len, int __user *ret){
    char string[str_len];
    //将字符串从用户空间copy到kernel空间
    copy_from_user(string,str,str_len);//第一个参数是要拷贝到kernel空间中的地址，第二个参数是用户空间中字符串的首地址，第三个参数是字符串的长度（多少个byte）
    int num=0;//num存放转换之后的数字，初始化为0
    int i;
    for(i=0;i<str_len;i++){//对字符串中的每个位上的字符进行计算
        num+=((*(str+i))-'0') * pow(10,str_len-1-i);//pow是自己实现的计算次方的函数
    }
    //将数字从kernel空间copy到用户空间
    copy_to_user(ret,&num,4);//第一个参数是用户空间中要被拷贝到的地址，第二个参数是kernel空间中存放数字的地址，第三个是被拷贝的数据类型的大小（即多少个byte）
    return;
}
```

print_val的实现

```c
asmlinkage void sys_print_val(int a){
    printk(KERN_EMERG "in sys_print_val:%d",a);
    return;
}
```

运行截图



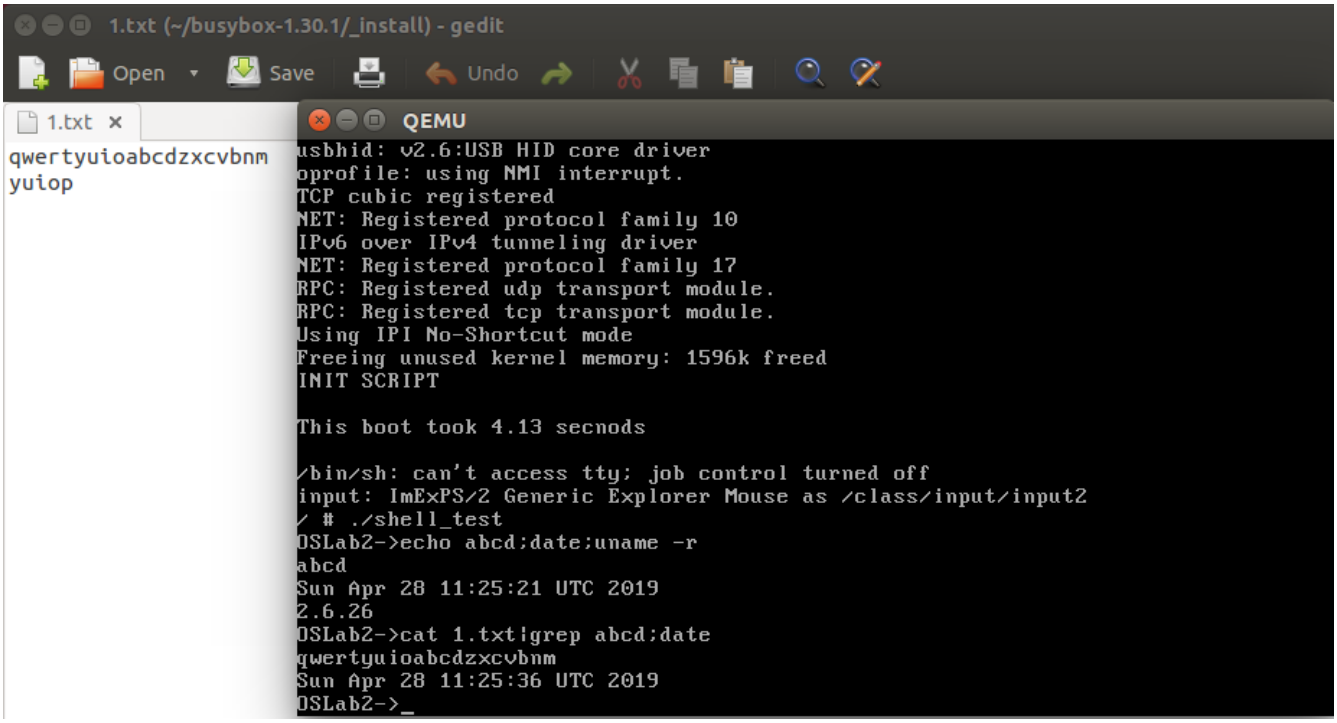
熟悉常见的系统调用

源代码

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

void STRCPY(char* cmd, char cmdline[256], int a, int b){//将cmdline的第a位到第b位的指令复制到cmd中
    int i;
    for(i=a;i<b;i++){
        *(cmd+i-a)=cmdline[i];
    }
    *(cmd+i-a)='\0';
    return;
}

int main(){
    char cmdline[256];//存放多条指令
    int cmd_num;
    int cmd_start,cmd_end;//cmd_start和cmd_end之间标记一条指令，cmd_start是指令的第一位，cmd_end是指令最后一位的下一位（通常是分号）
    int pipe=0;
    int findcmd=0;
    int i,p;
    FILE* fp1;
    FILE* fp2;
    while (1) {
        printf("OSLab2->");
        fgets(cmdline,256,stdin);
```

```
        cmd_num=strlen(cmdline);
        cmd_start=0;

        for (i=0; i<cmd_num; i++) {
            if(cmdline[i]=='|')pipe=i; //若读到了‘|’，说明需要管道，将管道标志位置为当前数组地址
            if(cmdline[i]==';' || i==cmd_num-1){ //若读到了‘; ’或cmdline的末尾，说明又找到了一条指令，将end置为当前地址，findcmd作为标志位置为1
                findcmd=1;
                cmd_end=i;
            }
            if(findcmd){//若findcmd为1，说明找打了一条指令，cmd_start和cmd_end将该指令夹住
                if (pipe) {//处理管道
                    char *cmd1;
                    char *cmd2;
                    cmd1 = malloc((pipe-cmd_start)*sizeof(char));
                    cmd2 = malloc((cmd_end-pipe-1)*sizeof(char));
                    STRCPY(cmd1,cmdline,cmd_start,pipe);//cmd1存第一条指令
                    STRCPY(cmd2,cmdline,pipe+1,cmd_end);//cmd2存第二条指令

                    fp1 = popen(cmd1,"r");//cmd1的文件以读模式打开
                    if(fp1==NULL){printf("file open failed!");break;}

                    fp2 = popen(cmd2, "w");//cmd2的文件以写模式打开
                    if(fp2==NULL){printf("file open failed!");break;}

                    char *buffer;//缓冲区暂存cmd1打开的文件中的内容
                    buffer=(char*)malloc(200*sizeof(char));
                    //fgets每次读一行，每读一行，将buffer中的内容写到fp2中，直到文件末尾
                    while(fgets(buffer,200,fp1)!=NULL)fputs(buffer,fp2);
                    //关闭管道，释放内存
                    pclose(fp1);
                    pclose(fp2);
                    free(buffer);
                    free(cmd1);
                    free(cmd2);
                }
                else {//处理非管道
                    char *cmd;
                    cmd = (char*)malloc((cmd_end-cmd_start)*sizeof(char));
                    STRCPY(cmd,cmdline,cmd_start,cmd_end);//cmd存此条指令
                    system(cmd);//调用system执行此条指令
                    free(cmd);
                }
                //将各个标志位恢复/更新
                findcmd=0;
                cmd_start=cmd_end+1;
                pipe=0;
            }
        }
    }
    return 0;}
```

运行截图



其中1.txt的内容如左所示

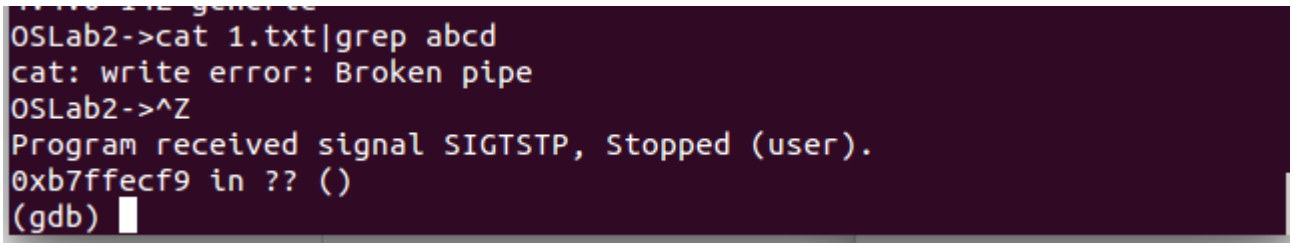
结果分析

如图所示，实验结果均正确。

其中我遇到了一个问题，就是buffer缓冲区的大小如何设置。因为fgets一次读一行，我担心如果一行内字符过多，buffer我设置的200太小会发生错误。然后我试了一下故意将buffer设置很小，设成5，再用上面的1.txt进行实验，发现实验结果还是正确的，说明fgets如果一行内的字符数把buffer要大的话会停止读取，下次从停止的位置继续读取。因此buffer的大小变成了无关紧要的问题。

技术问题

1.



调试过程中出现了上面的错误，说Broken pipe。上网搜了一下说是因为管道在还需要从cat中读内容时被grep关闭了。我检查了我的代码发现我提前关闭了cat打开的文件。

问题解答：

5

The reason is because the pipe is closed by `grep` when it still has some data to be read from `cat`. The signal `SIGPIPE` is caught by cat and it exits.

What usually happens in a pipeline is the shell runs `cat` in one process and `grep` in another. The stdout of `cat` is connected to the write-end of the pipe and stdin of `grep` to the read end. What happened was `grep` hit a pattern search that did not exist and exited immediately causing the read end of the pipe to be closed, which `cat` does not like since it has some more data to be write out to the pipe. Since the write actions happens to an other which has been closed other end, `SIGPIPE` is caught by the `cat` on which it immediately exits.

For such a trivial case, you could remove the pipeline usage altogether and run it as `grep "pattern" file.txt` when the file's contents are made available over the stdin of `grep` on which it could read from.

我的错误代码：

```
fp1 = popen(cmd1,"r");
if(fp1==NULL){printf("file open failed!");break;}
FILE* fp_temp=fp1;
fseek(fp_temp,0,SEEK_END);
int len = ftell(fp_temp);
char buffer[len];
fgets(buffer,len,fp1);
fclose(fp1);

fp2 = popen(cmd2, "w");
if(fp2==NULL){printf("file open failed!");break;}
fputs(buffer,fp2);
```

（但fp1是被我关掉的而不是grep，我感觉没问题，但是改掉代码后就好了。）

2.我本来想用feek和ftell判断文件长度来规定缓冲区大小，但是始终调用不成功，于是改了代码。

实验总结

本次试验中，我学会了如何添加系统调用，需要更改四个文件

- include/asm-x86/unistd_32.h（增加系统调用号）
- arch/x86/kernel/syscall_table_32.s（增加调用号和函数的关系）
- include/linux/syscalls.h（增加函数声明）
- kernel/sys.c（函数具体实现）

注1：sys.c中若函数参数中需要用到指向用户空间内的指针，需要添加__user宏，如：

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

注2：从用户空间读数据好写数据时，用copy_from_user和copy_to_user函数，它们的函数声明：

```
unsigned long copy_from_user(void * to, const void __user * from, unsigned long n);
unsigned long copy_to_user(void __user *to, const void *from, unsigned long n);
```

他们的第一个参数都是目标的地址destination，第二个都是原地址sourc，第三个都是要拷贝的字符长度，以八个字节为单位。