实验二：

添加Linux系统调用及熟悉常见系统调用

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 （32位）
- Linux内核版本：Kernel 2.6.26
- **注意**：本次实验是以实验一为基础，如有不熟悉的步骤，请重新浏览实验一文档。**内核版本最好是2.6.26，否则代码可能会有所不同。**（***注意加黑斜体下划线的地方。***）

实验内容

一、添加Linux系统调用

1、分配系统调用号，修改系统调用表

- 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***类似***:

```
  void print_val(int a);    //通过printk在控制台打印如下信息（假设a是1234）：
                            //  in sys_print_val: 1234
  void str2num(char *str, int str_len, int *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
```

- 在Linux源代码根目录下，找到include/asm-x86/unistd_32.h文件，在文件末尾找到最大的已分配系统调用号，并***仿照原格式***在其后新增两个系统调用号。

```
......
#define __NR_utimensat      320
#define __NR_signalfd       321
#define __NR_timerfd_create 322
#define __NR_eventfd        323
#define __NR_fallocate      324
#define __NR_timerfd_settime    325
#define __NR_timerfd_gettime    326

//在此添加
```

- 为了让系统能根据系统调用号找到syscall_table中的相应表项，在arch/x86/kernel/syscall_table_32.S文件末尾中***仿照原格式***添加系统调用号和调用函数的对应关系。

```
    ......
    .long sys_timerfd_create
    .long sys_eventfd
    .long sys_fallocate
    .long sys_timerfd_settime   /* 325 */
    .long sys_timerfd_gettime

    //在此添加
```

- ***注意：系统调用函数名字应该以"sys "开头。***

2、实现系统调用函数

- 在include/linux/syscalls.h中声明新增的两个系统调用函数。注意sys_str2num的声明中指针参数需要有***" user"宏***。

```
......
asmlinkage long sys_timerfd_gettime(int ufd, struct itimerspec __user *otmr);
asmlinkage long sys_eventfd(unsigned int count);
asmlinkage long sys_fallocate(int fd, int mode, loff_t offset, loff_t len);

//在此添加
```

- 在kernel/sys.c中实现新增的两个系统调用函数，（***注意：函数实现的头部与函数声明保持一致！***）。
- **提示：**sys_str2num的实现中需要利用copy_from_user和copy_to_user函数。(***注意n的大小，不要复制过多数据，也不要复制过少的数据。***)

```
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);
```

3、编译内核

- 执行如下命令编译内核

```
make i386_defconfig
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：

```
        Give me a string:
        78234
        in sys_print_val: 78234
```

- **提示：**通过syscall函数执行系统调用。

```
long int syscall (long int sysno, ...); //sysno为系统调用号，后面跟若干个系统调用函数参数，参数个数与其原型一致
```

5、运行测试程序

- 使用GCC静态编译用户程序

```
gcc -static test.c -o test
```

- 并将可执行文件test复制到busybox下的_install目录下，重新打包该目录生成新的cpio.gz文件，运行qemu进入shell环境后，输入./test，执行用户程序。

二、熟悉Linux下常见的系统调用

1、熟悉以下系统调用的用法

```
pid_t fork();   //创建进程
pid_t waitpid(pid_t pid,int* status,int options);   //等待指定pid的子进程结束
int execv(const char *path, char *const argv[]);     //根据指定的文件名或目录名找到可执行文件，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号外，其他全部被新程序的内容替换了
int system(const char* command);    //调用fork()产生子进程，在子进程执行参数command字符串所代表的命令，此命令执行完后随即返回原调用的进程
FILE* popen(const char* command, const char* mode); //popen函数先执行fork，然后调用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的文件指针
int pclose(FILE* stream);   //关闭标准I/O流，等待命令执行结束
```

2、利用上面的系统调用函数实现一个简单的shell程序

- 要求如下：

  （1）每行命令可能由若干个子命令组成，如"ls -l; cat 1.txt; ps -a"，由“;”分隔每个子命令，需要按照顺序依次执行这些子命令。

  （2）每个子命令可能包含一个管道符号“|”，如"cat 1.txt | grep abcd"（这个命令的作用是打印出1.txt中包含abcd字符串的行），作用是先执行前一个命令，并将其标准输出传递给下一个命令，作为它的标准输入。

  （3）最终的shell程序应该有类似如下的输出：

```
OSLab2->echo abcd;date;uname -r
abcd
Mon Apr 8 09:35:57 CST 2019
2.6.26
OSLab2->cat 1.txt | grep abcd
123abcd
abcdefg
```

- 提示：程序的大致框架如下

```
int main(){
    char cmdline[256];
```

```
    while (1) {
        printf("OSLab2->");
        //读取一行字符串并根据“;”将其划分成若干个子命令
        for (i=0; i < cmd_num; i++) {
            if () { //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

            }
            else {  //通常的情况，利用fork创建子进程并执行命令

            }
        }
    }

    return 0;
}
```