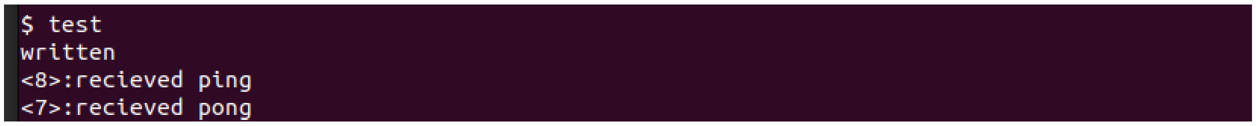
HOMEWORK 1 NOTES
Ikhlas Ahmed Khan
27096

Q1: **PING PONG**

```
int main(void) {
        int p1[2]; // parent-pipe
        pipe(p1);

        int p2[2]; // child-pipe
        pipe(p2);
        int pid=fork();

        /*1*/
        if(pid>0){
        write(p1[1],"P",1); // writes on the pipe p[1]
        printf("written\n");
        wait((int *)0); // waits for child to complete

        /*3*/
        char *bufp[1];
        read(p2[0],bufp,1); // parent reads from pipe p2 on which child has written
        printf("<%d>:recieved pong\n",getpid() );
        }

        /*2*/
        else if(pid==0){//child
        char *buf[1];
        read(p1[0],buf,1); // child reads the byte in p1 which the parent has written
        printf("<%d>:recieved ping\n",getpid() );

        //2nd pipe process
        write(p2[1],"C",1); //child writes on pipe p2[1]
        exit(0);
        }
        return 0;
}
```

OUTPUT on xv6:

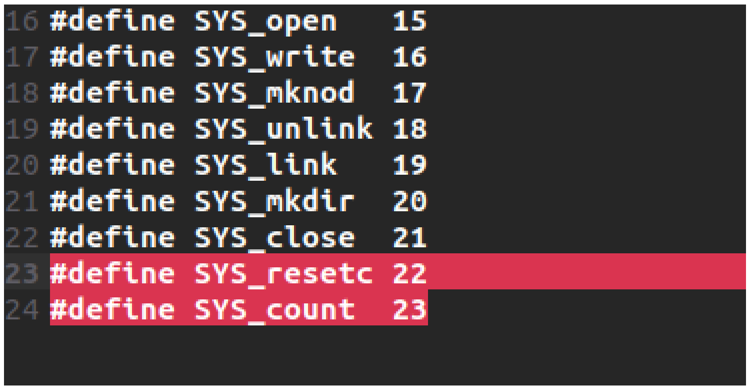

```
$ test
written
<8>:recieved ping
<7>:recieved pong
```

The received ping indicates that the child has read the contents in pipe p1
The received pong indicates that the parent has read the content in pipe p2

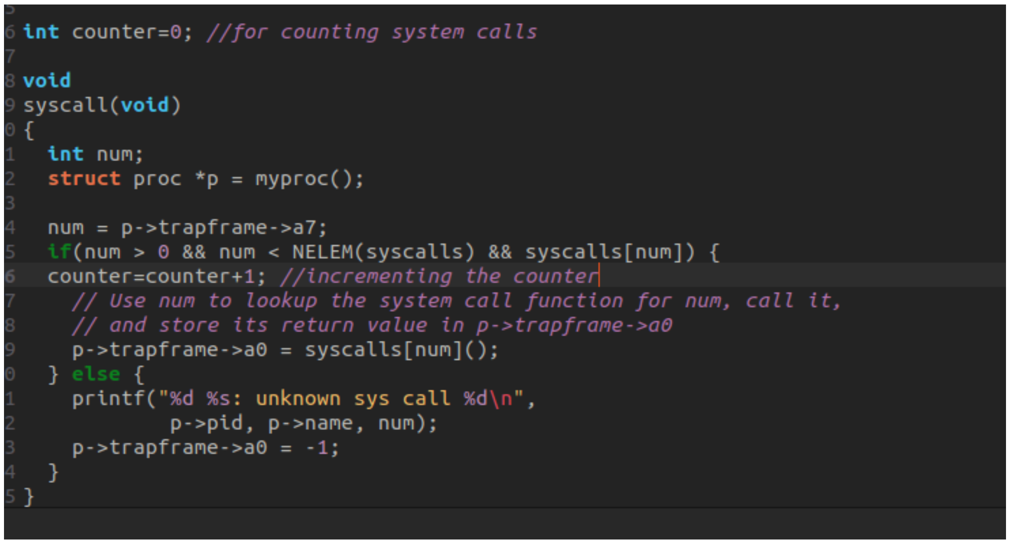
Q2: **System calls count**

Before you understand the code you need to see the link below:
https://medium.com/@mahi12/adding-system-call-in-xv6-a5468ce1b463

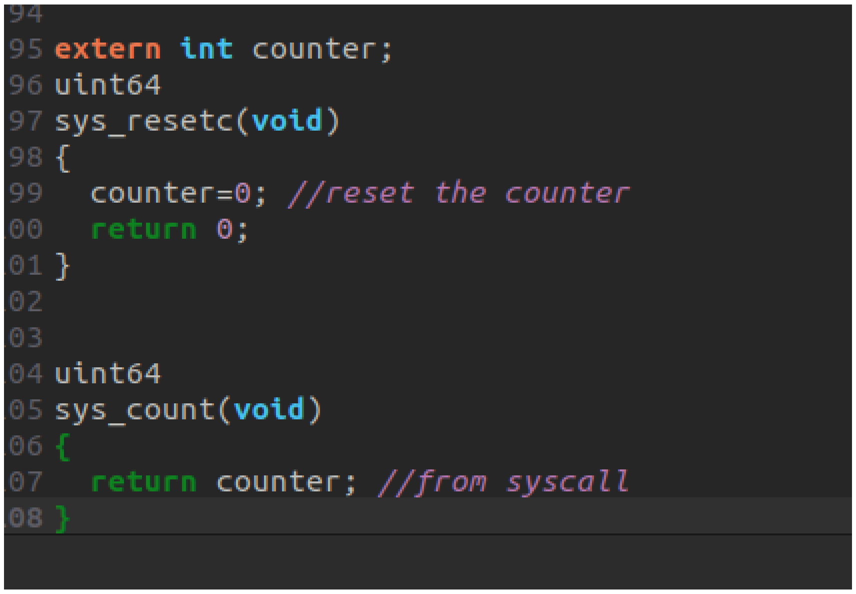
1. So first i went to **syscall.h** file in kernel and added the following lines;


```
#define SYS_open    15
#define SYS_write   16
#define SYS_mknod   17
#define SYS_unlink  18
#define SYS_link    19
#define SYS_mkdir   20
#define SYS_close   21
#define SYS_resetc  22
#define SYS_count   23
```

2. Then i headed to **syscall.c file** in kernel and added a variable of counter and increment it each time a system call is invoked
Problem: This would involve the startup system calls as well this is why i will be introducing resetc() systemcall to reset the counter

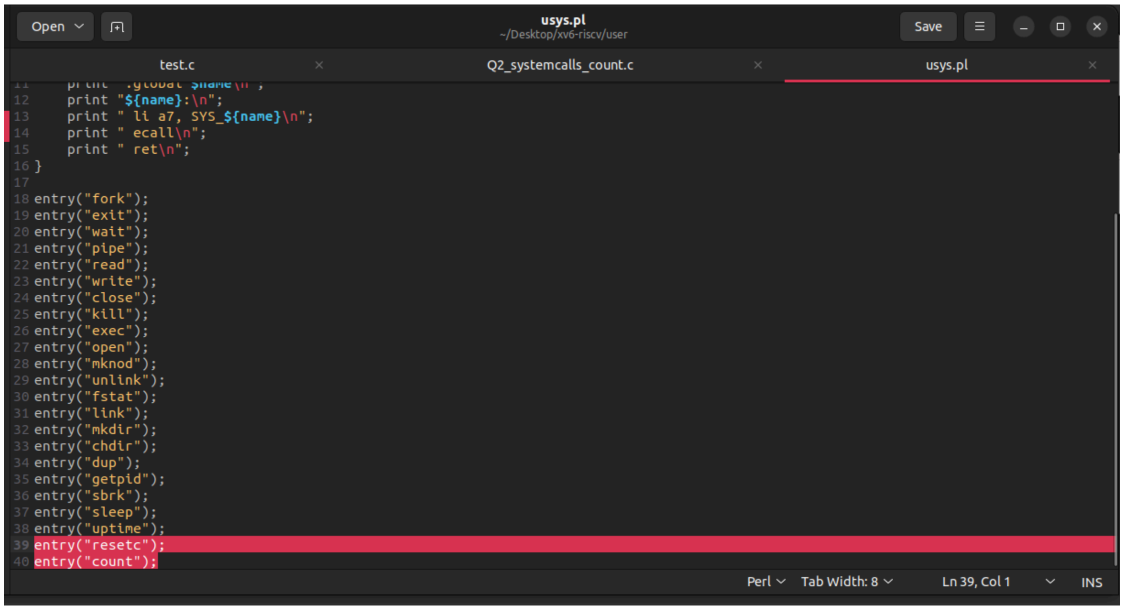

```
int counter=0; //for counting system calls

void
syscall(void)
{
  int num;
  struct proc *p = myproc();

  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
  counter=counter+1; //incrementing the counter
    // Use num to lookup the system call function for num, call it,
    // and store its return value in p->trapframe->a0
    p->trapframe->a0 = syscalls[num]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

3. Then i implemented both the functions inside **sysproc.c**



```c

extern int counter;
uint64
sys_resetc(void)
{
  counter=0; //reset the counter
  return 0;
}


uint64
sys_count(void)
{
  return counter; //from syscall
}
```

4. (NOTE there is no such thing as Usys.S in this version of xv6)
   So i edited the **Usys.pl** file in users to finish adding my system call



```perl
    print ".global $name\n";
    print "${name}:\n";
    print " li a7, SYS_${name}\n";
    print " ecall\n";
    print " ret\n";
}

entry("fork");
entry("exit");
entry("wait");
entry("pipe");
entry("read");
entry("write");
entry("close");
entry("kill");
entry("exec");
entry("open");
entry("mknod");
entry("unlink");
entry("fstat");
entry("link");
entry("mkdir");
entry("chdir");
entry("dup");
entry("getpid");
entry("sbrk");
entry("sleep");
entry("uptime");
entry("resetc");
entry("count");
```

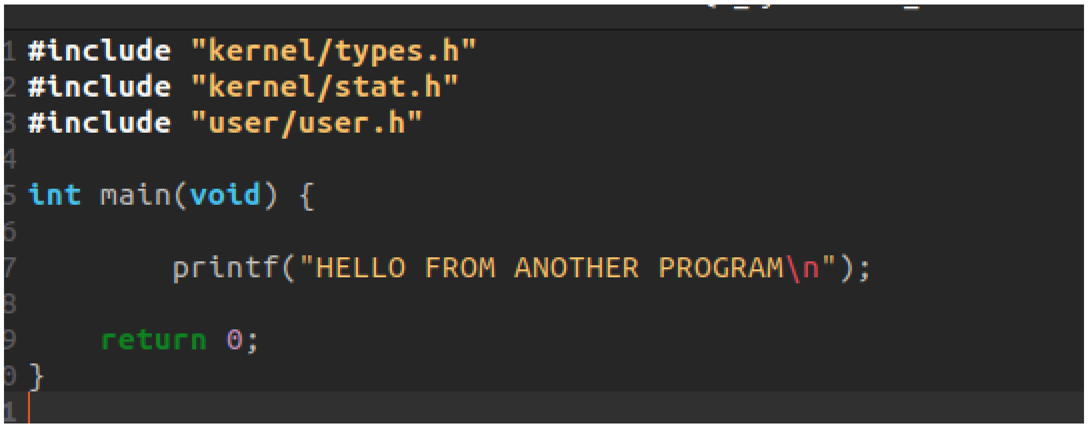
With this i have successfully introduced 2 new system calls,

1) **resetc();** which basically just resets the counter for system calls.
2) **count()**; which returns,in integer, the number of system calls.

Their implementation is in **systemcalls_count.c** file:

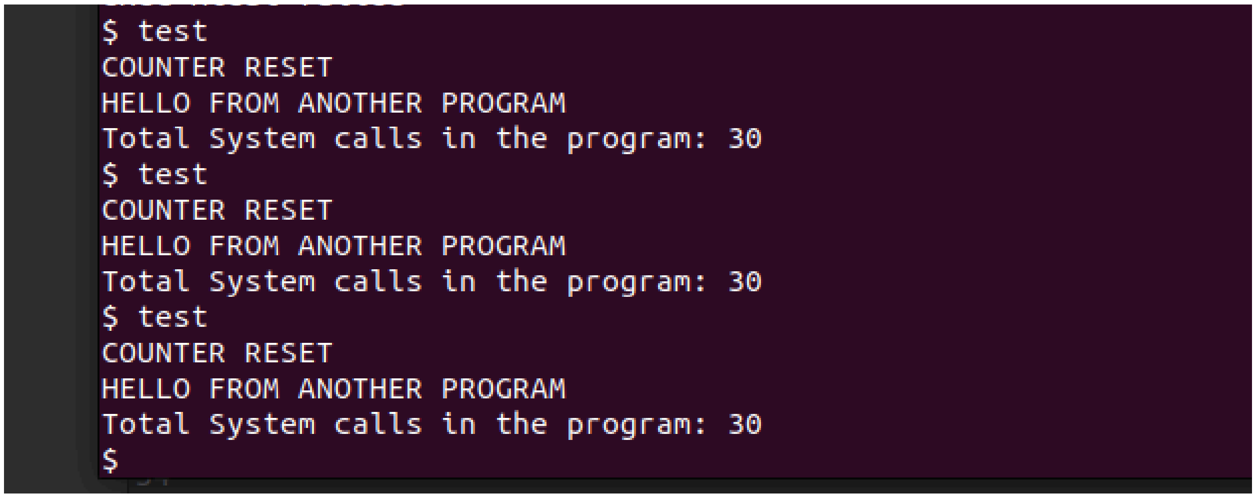
```
if (pid == 0) { // child
   char *args[] = {"Simpleprog", "Hello", "from", "ikhlas", 0}; //file-name
// can replace args[0] with any file/program name you want to run
   // or you can capture from the command line
   printf("COUNTER RESET\n");
   resetc(); // RESET COUNTER /*1*/
   exec(args[0], args); // EXECUTE PROGRAM IN CHILD /*2*/
   exit(0);
} else { // parent
   wait(0);
   // gave total system calls /*3*/
   printf("Total System calls in the program: %d\n", count());
}
return 0;
```

***Simpleprog is a c file placed in users folder in xv6**



```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main(void) {

        printf("HELLO FROM ANOTHER PROGRAM\n");

    return 0;
}
```

OUTPUT IN XV6:



```
$ test
COUNTER RESET
HELLO FROM ANOTHER PROGRAM
Total System calls in the program: 30
$ test
COUNTER RESET
HELLO FROM ANOTHER PROGRAM
Total System calls in the program: 30
$ test
COUNTER RESET
HELLO FROM ANOTHER PROGRAM
Total System calls in the program: 30
$
```

A point to note here is that the number of system calls for the program (Simpleprog) remains the same (30) at each call, which means that the counter is successfully reset before the program call and it counts the system calls in the program correctly.

Q3: a)

Benefit of pipes over other file Redirection method

- Pipes automatically clean themselves up. No need to explicitly delete any resources in the pipe.
  However with file redirection, there are temporary files created which ~~are~~ have to be carefully removed by the shell (in OS).

- Pipes can pass data between processes without intermediate storage.
  However file redirection writes data to a file in disk before passing data.

- Pipes enable parallel execution of pipeline stages, by allowing ~~one process~~ simultaneous read and write operations on the pipe. (Processes can run concurrently and effeciently).
  However file redirection allows only one operation (read or write) at a time
  e.g ~~one can program cannot start until the previous program~~
  e.g First program has to finish before second starts, leading to sequential execution.

Q3)c) **Potential Reasons: Why Q3's Code Fails to Run on xv6**

- There were a few errors since some functions of C are not available in xv6 or are available with some different name so I had to modify all of them. So i have added **/*ERROR*/** in those lines which wont run in xv6 but will in C.
- The code below runs in xv6 but isn't able to run **exec command** on **ls** and **tr** files as ls and tr are unix based utilities and are not available in xv6
- **A proposed solution is to make your own custom LS and TR files,add them in user files in xv6, and then run it.**
- Note that since **exec** is unsuccessful so the lines after **exec** runs by default

Secondary Issues:

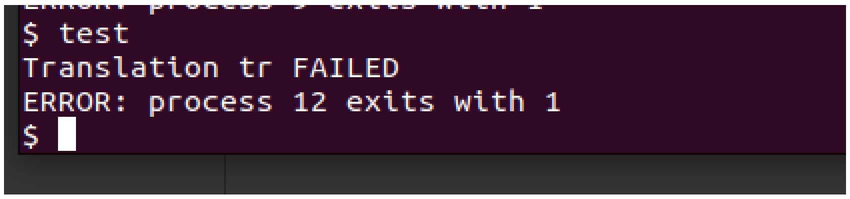
- **fprintf** doesnt exist in xv6 only printf does.
- **execvp** does not exist in xv6,instead exec does.
- **WEXITSTATUS** not in XV6.
- **perror** not in xv6 so i just replaced it with a simple printf.
- **dup2** not in xv6 but its equivalent is **close()** and **dup()**.
  E.g **dup2(p[0],0)** means redirect pipe p’s read end to standard-input(0)
  This is equivalent to writing **close(0)**, and then **dup(p[0])** in xv6.

```
int main(int argc, char **argv)
{
        int pid, status;
        int fd[2]; //file decsriptor of pipe always set to 2
        pipe(fd); //piping


        switch (pid = fork()) {
        case 0:
        runpipe(fd);
        exit(0);



        default:
        while ((pid = wait(&status)) != -1)
/*ERROR*/     //fprintf(stderr, "process %d exits with %d\n",pid,WEXITSTATUS(status)); //
        //WEXITSTATUS NOT IN XV6
        printf("ERROR: process %d exits with %d\n",pid,status);
        break;
        case -1:
/*ERROR*/    //perror("fork");
        printf("FORK FAILED\n");
        exit(1);
        }
        exit(0);

}
```

**ls and tr not available in xv6

```
        char *cmd1[] = { "/bin/ls", "-al", "/", 0 };
        char *cmd2[] = { "/usr/bin/tr", "a-z", "A-Z", 0 };

void runpipe(int pfd[])
{
        int pid;

        switch (pid = fork()) {
        case 0:
/*ERROR*/    //dup2(pfd[0], 0);
        close(0);
        dup(pfd[0]);
        close(pfd[1]); //close write end of pipe to avoid data leakage

/*ERROR*/   //execvp(cmd2[0], cmd2);
//altho exec is a correct command in xv6
        exec(cmd2[0], cmd2); //will not execute properly as TR doesnt exist in xv6
        printf("Translation tr FAILED\n");
/*ERROR*/    //perror(cmd2[0]);
        exit(0);

        default:
/*ERROR*/    //dup2(pfd[1], 1);
        close(1); //close stdout
        dup(pfd[1]); //redirect pipe pfd's write end  to stdout
        close(pfd[0]);
/*ERROR*/    //execvp(cmd1[0], cmd1);
        exec(cmd1[0],cmd1); //will not execute properly as LS does not exist in xv6
        printf("Listing FAILED\n");
        wait(0);
/*ERROR*/     //perror(cmd1[0]);

        case -1:
/*ERROR*/     //perror("fork");
        printf("FORK FAILED\n");
        exit(1);

        }
}
```

Output on xv6 (indicating it's running but not producing the expected result):



```
$ test
Translation tr FAILED
ERROR: process 12 exits with 1
$
```

This shows that the printf function prints the message Translation tr failed as the exec command did not run successfully in both the parent and the child.