# Operating Systems - Homework 1

## [Github repo](https://github.com/subata-naveen-khan/OS-Fall2024-HW) Subata Naveen Khan – 18119

1. File: subata\_q1.c:

#include "kernel/types.h"

#include "kernel/stat.h"

#include "user/user.h"

int **main** (void) {

    int p\_ptc[2], p\_ctp[2]; *// ptc = parent-to-child, ctp = child-to-parent*

    char byte[1]; *// char datatype has size of 1 byte*

    int n = 3; *// no. of times byte gets sent from each side*

    int i;

**pipe**(p\_ptc);*//child will read from p\_ptc[0], parent will write to p\_ptc[1]*

**pipe**(p\_ctp);*//parent will read from p\_ctp[0], child will write to p\_ctp[1]*

    if (**fork**() == 0)  { *// child*

**close**(p\_ptc[1]); *// won't write to ptc pipe*

**close** (p\_ctp[0]); *// won't read from ctp pipe*

        for (i = 0; i < n; i++)

        {

**read**(p\_ptc[0], byte, 1);

**printf**("child receives: %c\n", byte[0]);

            byte[0] = 'c';

**printf**("child sends: %c\n", byte[0]);

**write**(p\_ctp[1], byte, 1);

        }

**close**(p\_ctp[1]);

**close**(p\_ptc[0]);

**sleep**(1); *//making sure everything is finished printing before exiting*

**exit**(0);

    } else { *// parent*

**close**(p\_ptc[0]); *// won't read from ptc pipe*

**close**(p\_ctp[1]); *// won't write to ctp pipe*

        for (i = 0; i < n; i++)

        {

            byte[0] = 'p';

**printf**("parent sends: %c\n", byte[0]);

**write**(p\_ptc[1], byte, 1); *// byte sent to child*

**read**(p\_ctp[0], byte, 1);

**printf**("parent receives: %c\n", byte[0]);

        }

**close**(p\_ptc[1]); *// closing remaining pipe ends*

**close**(p\_ctp[0]);

**wait**(0); *// waiting for child*

    }

**exit**(0);

}

Output:   
A screen shot of a computer screen

Description automatically generated

1. File: subata\_q2.c:

#include "kernel/types.h"

#include "kernel/stat.h"

#include "user/user.h"

#include "kernel/fcntl.h"

int **main**(int argc, char \*argv[]) {

    int p[2];

    if (**pipe**(p) < 0) {

**printf**("pipe creation failed\n");

**exit**(0);

    }

    int pid = **fork**();

    if (pid == 0) { *// child*

**close**(p[0]); *// won't read from pipe*

**countsyscalls**(p[1]); *// passes p[1] to kernel to write to*

**exec**(argv[1], argv + 1); *// execute the target program*

**printf**("exec failed\n");

**exit**(0);

    } else { *// parent: read from pipe to count syscalls*

**close**(p[1]); *// won't write to pipe*

**uint64** buf[1]; *//*

        int syscall\_counter = 0;

        while (**read**(p[0], buf, sizeof(buf[0])) > 0) {

            syscall\_counter++; *// while pipe is being written to*

        }

**printf**("number of system calls made: %d\n", syscall\_counter);

**close**(p[0]);

**wait**(0);

    }

**exit**(0);

}

File: syscall.c:

void

**syscall**(void)

{

  int num;

  struct **proc** \*p = **myproc**();

  num = p->trapframe->a7;

  if(num > 0 && num < **NELEM**(syscalls) && syscalls[num]) {

*// if calling program called countsyscalls*

    if(p->syscall\_pipe >= 0) {

        struct **file** \*f = p->ofile[p->syscall\_pipe];

        if (f) {

**uint64** counter\_signal = 1;

**filewrite**(f, counter\_signal, sizeof(counter\_signal));

        }

    }

*// 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",

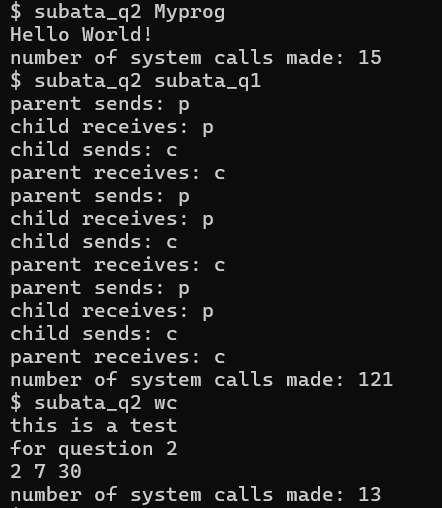
            p->pid, p->name, num);

    p->trapframe->a0 = -1;

  }

}

Output:



1. a. - Pipes automatically clean themselves up, as opposed to file redirection, where the

shell would need to remove the temorary file that is created.

* + Pipes can handle arbitrarily long streams of data. With file redirection, the disk must have enough free space to store all the data.
  + Pipes allow for parallel execution of the different pipeline stages. With file redirection, the first program needs to finish before the next can start.
  + Pipes provide efficient inter-process communication. The reading and writing processes can communicate directly and without delay. With file redirection, there is no real-time communication so it is much less efficient.

1. The main reason this code cannot be implemented in xv6 is that xv6 is a bare bones operating system with limited functionality, so it does not support the operations required.
   * xv6 has an ls command, but it doesn’t have options like -al
   * xv6 doesn’t have a tr command