Washington University
Department of Computer Science

CS422: Operating Systems Organization
Fall 1997

Programming Project
Due September 18^{th}, 1997

The purpose of this programming project is to familiarize you with basic UNIX operating system interprocess communication (IPC) system calls. You will be using C++ to implement your programs. Note that part of your grade will depend on the clarity and style of your program, so make sure to write "readable" code...

1 UNIX Local Interprocess Communication (IPC)

The purpose of this project is to learn about the UNIX system calls that perform IPC between separate processes on a local host machine. Many traditional UNIX applications (such as the C shell, compilers, and line printer daemons) utilize local IPC. You will write two programs: one using unnamed pipes and one using named pipes (*a.k.a.* FIFOs).

1.1 Pipe Implementation

- Take as the only program argument, the name of an input file (*e.g.,* `"/etc/passwd"` or `"/usr/dict/words"`).
- Set up an unnamed pipe [`pipe(2)`], create a child process [`fork(2)`], set up the stdin of the child to read from the pipe and the stdout of the parent to write to the pipe [`close(2)`/`dup(2)` or `dup2(2)`].
- In the child, exec the program `/usr/bin/more` [`execl(2)`]. In the parent, open the named input file [`open(2)`] and read the input file (in BUFSIZ chunks), and write the data to the input end of the pipe.
- When the parent is finished sending the file through the pipe to the child, it should wait for the child to exit [`wait(2)`]. After the child has exited the parent should exit [`exit(2)`].

1.2 FIFO Implementation

- Reimplement the first part of the assignment, using UNIX *named pipes*, (*i.e.,* FIFOs) instead of unnamed pipes. This will require a few minor changes, in particular, you will need to create the FIFO using a "well-known" pathname [`mknod(2)` or mkfifo(2)].[1] However, the program should behave identically with the pipe-based version you implemented earlier. Note that when you are finished, you should remove the FIFO [unlink(2)].

Your programs should simply print out the appropriate error message [perror(3c)] and exit with a return status of 1 if any of the system calls fail to work properly (*e.g.*, due to input errors from the user). If everything works correctly, the program should exit with a return status of 0.

2 Points to Ponder

Note that both of these programs are somewhat tricker than they appear at first glance. In particular, be very careful to close down I/O descriptors when they are no longer needed. Otherwise, you're program will hang mysteriously.

In addition, to prevent a "broken pipe" error when performing a write(2) on a pipe or FIFO, you may want to have the parent process ignore the SIGPIPE signal (sigaction(2)). You'll then need to check if errno (perror(3f)) is equal to EPIPE when a write(2) to a pipe or FIFO returns -1. If this happens, don't print an error message in the parent process, just exit normally.

3 Getting Started

You can get the "shells" and Makefile for part one of the program from your account on cec. These files are stored in /project/adaptive/cs422/assignment-2/. Here's a script that shows you how to set everything up and get these files:

```
% cd ~/cs422
% mkdir assignment-2
% cd assignment-2
% cp -r /project/adaptive/cs422/assignment-2/* .
% make
```

The Makefile and various header files are written for you. All you need to do is edit the *.C files to add the code to implement the programs.

[1] Chapter 3 in the Richard Stevens UNIX Network Programming book contains more information about the subtleties creating FIFOs.