COMP 3430 Winter 2013 Assignment 1

Due Date: Wednesday, Jan 30, 11:59 PM

Notes and Style

- must be written in C
- provide a signed Honesty Declaration to your instructor by the due date.
- your assignment code must be handed in electronically using the D2L drop box. Try early and re-submit later. Include a Makefile for all questions. Confusion over D2L at the last minute is your own fault.
- Use common-sense commenting. Provide function headers, comment regions, etc. No particular format or style is required.
- Error checking is extremely important in real world OS work. However, it is complex and can get in the way when trying to learn these concepts. You can assume extreme cases will not happen such as lack of memory or disk failure (and thus can assume that, e.g., read and write will never fail). Error check for things that you should report to the user or recover from, e.g., missing file.

Q1 – processes and anonymous pipes (30%)

Implement a program that takes the name of a file to run as an argument (note that a correct path is required). Your program will create an anonymous `pipe` for communication and `fork` a child process for executing the specified program. Prior to performing an `execv`, the child will link the write end of the pipe to its standard output using `dup2` (note that you will need to pass a file number, which is defined in `STDOUT_FILENO` for standard output). Your use of `execv` should pass the remaining command line arguments (including the name of the program run!) to the child. Note that, in linux, the `argv` passed to your program is already a NULL terminated array. The parent process will read the child's output through the pipe, modify it to replace all instances of the text "cat" with "dog", and put it out to standard out. Make this case insensitive for checking but don't worry about the case of output.

For example, try the following invocation of your program: "`./q1 \`which lynx\` http://lolcats.com`" as demonstrated in class.

In addition, your parent process will dump a copy of all unmodified data received from the child through the pipe to a file on disk (e.g., "raw.log").

hint: read a character at a time to test for "cat". Buffer characters until you are sure it's not "cat", but otherwise output as you get them

hint: use the low level `open/close/read/write` commands instead of the buffered `FILE` ones

Q2 – a prime-finding server and client using named pipes (FIFO files) (70%)

You will implement a client-server system using FIFOs (named pipes). You will create a server which is searching for ever-larger prime numbers. Rather than doing the work itself, however, it farms the work out to clients.

Implement a client-server system using FIFOs. There will be a maximum of 10 clients, numbered 0-9. The server will create a FIFO called `./primeserver` and each client will create a FIFO called `./primeclientx` where `x` is the client number (from 0-9): the client identifier should be given to the client program as a command-line option, using

`-c` – see `man 3 getopt` for example code. Each client will send requests to the server using the `primeserver` FIFO to ask for work, and the server will send responses back using each client's FIFO. Each program is responsible for creating their own FIFO and removing it prior to exiting.

The flow is as follows:

a) a client connects to the server
b) the client sends a request for a prime
c) the server sends back a number *n* which may or may not be prime
d) the client checks *n*: if it is prime, it sends a "prime found" message to the server
e) loop a) (note that no negative confirmation is sent).

Protocol: the server and client needs to communicate so we will define a simple protocol (which is quite poor; you should take the Networks course for more info on protocol design). To implement do not worry about rigorous error checking beyond basics (complain if protocol is broken, etc.) which will help you debug. All messages will be ascii text and will have the following format:

"!#C<DATA>?"

- '!' denotes start of command
- '#' is an ascii number representing sender: 0-9 for client #, or 'S' for server
- 'C' is a single letter command:
  - 'R' requests a prime to check, used by the client. This command has an empty data field.
  - 'P' tells the server that a prime was found. The prime number is contained in ascii format in the data field.
  - 'Q' is a query. Server uses this to send a prime to the client, with an un-checked number in the data field.
- <DATA> - this region contains an ascii representation of the number. Note that this is varying length depending on the number size
- '?' denotes end of command

An example is "!8P13?" which is a message from client #8 saying that 13 is a prime number. **Hint:** for debugging, you can send messages through the FIFO to the server by echoing them into the FIFO file directly:

```
echo '!8P13?' > primeserver
```

To implement this, you may find it useful to make a state machine using your own "packet" data structure (including, e.g., source, message, int data, string data) and enumerated type for states and a byte-by-byte parsing of input until a complete packet is received.

It is recommended to put the protocol code in a separate module (.h/.c) to be used by both the client and the server for parsing and generating messages.

Server: The server will start, create its FIFO, and listen for messages from clients. It should keep track of the last checked number (incrementing as queries are sent) and currently largest prime, and print the largest prime to standard out as new ones are found. Make sure your program runs with small numbers (thus a lot of traffic as prime-checking is instant) and large numbers such as starting at 5,000,000.

Client: The client will start, and check if the server exists (by if its FIFO exits). If so, create its FIFO, and start sending requests to the server. Prime checking is easy, don't waste your time on optimizations (slower solutions may be better to help manage output for debugging):

```
factors = 0
for i = 1..number
      if (number%i==0) factors++
isPrime = (factors==2)
```

Signals: Your client and server will only quit when signaled to do so. Both should respond to the `SIGINT` signal (ctrl-C) and the `SIGTERM` signal (default kill), clean up by closing files and removing their FIFOs, print a message to screen, and exit. You can use `exit` in the signal handler – graceful quitting of your loop is doable but getting it right requires proper error checking of all system calls to prevent blocking. Both should also respond to the `SIGPIPE` signal, but in this case, raise a flag and panic (via an informative fprintf to stderr) in addition to cleanup.

Running the system: It may be useful to work in two terminals simultaneously (if you are sshing in, make sure both are on the same system or FIFOs are unlikely to work). Run the server on one, and run ten simultaneous clients on the other. A bash command for doing this is: "`for x in ``seq 0 9``; do ./client -c $x & done`"

Misc: Your programs should only keep FIFOs open as long as needed, and close them immediately after use. This is considered to be "polite," minimizes system resources, and helps keep the system robust. The exception is that, as the server is always listening, it should keep its FIFO open all the time (why?? why won't it just buffer until the server opens for read?).

Bonus (5%): Your server should use barely any CPU resources, but with most solutions you will see the server soon using as much as or more than a client. Why is this? Fix it, and provide a detailed written explanation (1/4 page) of what is happening and why your solution is safe.