Simon Fraser University
School of Computing Science
CMPT 300: Assignment #3

IPC and Concurrency

Reminder: The rules of academic conduct apply as described in the course outline. All coding is to be done individually or in pairs. We will be using software to compare all assignments handed in by the class to each other, as well as to assignments handed in for previous terms.

Be sure that this assignment compiles on the Linux computers in the CSIL lab using the gcc compiler. You can access the Linux server remotely as detailed on the course discussion forum.

What to Hand In: Include your source code file(s) (.h and .c) and a makefile (with a default target to make all, and a *clean* target).

There is a bit of work to do here and you should begin right away. Trust me - you will not finish this if you leave it for the last week. Do yourself a huge favour and begin NOW.

This is a great assignment. Why is it great you ask? It is great for two reasons:

1. you are going to learn many things:
   - how to use UNIX UDP IPC
   - increased detail regarding the joy (and pain) of programming concurrent processes
   - how to program using the client/server model
   - how to write a multi-threaded program using *pthreads* (which is representative of most other threads packages)
   - how to solve the critical section problem between threads
2. it is going to be fun
   - you are left to judge the truth of this statement

Assignment Overview

For this assignment you are going to create a simple "chat"-like facility that enables someone at one terminal (or Xterm) to communicate with someone at another terminal. The interface will not be pretty, but it will be functional.

This program will be called "s-talk" for simple-talk. To initiate an s-talk session two users will first agree on two things:

- the machine that each will be running on
- the port number (explained later) each will use

Say that Fred and Barney want to talk. Fred is on machine "csil-cpu1" and will use port number 6060. Barney is on machine "csil-cpu3" and will use port number 6001.

To initiate s-talk, Fred must type:

```
s-talk 6060 csil-cpu3 6001
```

And Barney must type:

```
s-talk 6001 csil-cpu1 6060.
```

So, (in case you haven't figured it out) the general format is:

```
s-talk [my port number] [remote machine name] [remote port number]
```

The result will be that every line typed at each terminal will appear on BOTH terminals: it will appear character-by-character on the sender's terminal as they type it, and it will appear on the receiver's terminal once the sender presses enter.

If you want to learn about "curses" and "cbreak" on your own, you can alter this slightly so that every character typed appears on both screens as it is typed rather than having to wait for each [return]. If you are interested look in the man pages under "*curses*" (this is NOT a requirement of the assignment; it is not for marks).

An s-talk session is terminated when either user enters the complete line of text which is just one '!' and presses enter.

Expected Process Structure for this Assignment

This assignment will be done using pthreads, a kernel-level thread implementation for LINUX. As you may or may not know, pthreads allows you to create any number of threads inside one UNIX process. All threads running in the same UNIX process share memory (which means pointers valid for one thread are valid in another thread) and also have access to semaphores (mutexes) and the ability to use conditional signal/wait to synchronize their actions in relation to each other. UNIX itself also allows you to create multiple processes. Communication between UNIX processes can be done using something called "datagram sockets" which use a protocol called UDP (universal datagram protocol).

In this assignment, you will be dealing with processes/threads on two levels. First, you will have two UNIX processes. Each one is started by one of the people who want to talk (as a result of executing s-talk). Second, within each s-talk process a pthreads environment will be running four threads that you will write.

Required threads (in each of the processes):

- One of the threads does nothing other than await input from the keyboard.
- The other thread does nothing other than await a UDP datagram.
- There will also be a thread which prints characters to the screen.
- Finally a thread which sends data to the remote UNIX process over the network using UDP.

All four threads will share access to a list ADT (the one you wrote for assignment #1).

- The keyboard input thread, on receipt of input, adds the input to the list of messages that need to be sent to the remote s-talk client.
- The UDP output thread will take each message off this list and send it over the network to the remote client.
- The UDP input thread, on receipt of input from the remote s-talk client, will put the message onto the list of messages that need to be printed to the local screen.
- The screen output thread will take each message off this list and output it to the screen.

Clearly the lists that are shared between the threads will need to be carefully controlled to prevent concurrent access. This will be done by using mutexes and signalling/waiting on condition variables.[1]

How do you go about starting?

There are several things you are going to have to figure out - and you'd best start NOW! I think you will appreciate all the knowledge you gain from doing this assignment, but there is a lot to learn - this is not one you can leave until the last few days. There will be no extensions.

First, try out some of the keyboard/screen I/O primitives supplied by UNIX without using pthreads. Check the section 2 man pages under "read", "write" and (optionally) "curses", "cbreak".

Next (and this is a big one) you'll need to experiment with UNIX UDP sockets. Do this by experimenting with code that sends a message (without pthreads) from one UNIX process to another.

Check the man pages for:

- socket
- bind
- sendto
- recvfrom

[1] It would be easy to modify the process structure to have the keyboard input thread send its characters directly out over the network, and to have the UDP input thread print its characters directly to the screen. **Please don't do it this way** - I'd like you to get more experience writing client/server applications. Plus, there's a deduction for doing it the wrong way. ☺

- gethostname
- gethostbyname

Some of these will be complicated, but don't despair. You can ask the TAs and me questions in the discussion forum, though we won't answer any questions that can be easily obtained from the man pages. I also strongly suggest you use the web as a resource here. Try a search on "UNIX and SOCKET" and see what you come up with. If you find something good, please share it with others using the discussion forum. There are a couple of good web-pages that I will point you to:

- Socket programming: http://beej.us/guide/bgnet/
- Pthreads documentation: https://computing.llnl.gov/tutorials/pthreads/

Finally, bring them all together in a pthreads application and two UNIX processes.

Marking

To test, we will not only run your program as a normal user might, but we will also

- Pipe a text file to s-talk (contents of the text file appears to your program as standard keyboard input):
  `cat someTestData.txt | ./s-talk <args> <go> <here>`
    - The contents of the *someTestData.txt* file must be sent over the network to the attached other s-talk client.
- Pipe the output of your program to a text file
  `./s-talk <args> <go> <here> >> someOutputData.txt`
    - The contents of *someOutputData.txt* must be the data that was sent from the other s-talk client.
    - Your s-talk program may print out additional messages when it starts up, and/or when it terminates.
- We will not intercept or analyze the UDP packets; therefore, you may use whatever message structure you like in your network packets.

Submit to CourSys a ZIP file of the following:

- all your source code (.c and .h files)
- a makefile that compiles your .c files and links the objects to any necessary libraries. The makefile should produce the executable *s-talk* (all lower case) which behaves according to the description above. The makefile must have a default rule to build *s-talk*, and must also have a 'clean' rule.
- (optionally) README file documenting any errors, like things you didn't get working, or things we need to know to mark your work
- Do NOT hand in any executables or .o files.

Have fun and good luck.