Department of Computing and Information Systems
COMP30023: Computer Systems

Project 2A

Due date: No later than 5:00pm on Friday, May 29.

Weight: 10%

Project Overview

The aims of this project are:

- To provide you with experience in writing programs that interact with each other over a network (socket programming).
- To provide you with experience in writing multi-threaded applications and process synchronization.

Your task consists of two components:

- To implement a server capable of handling multiple concurrent requests for the game 'Connect 4'.

  See `http://en.wikipedia.org/wiki/Connect_Four` for an overview of the game.
- To implement a client program that can be used to test your game server.

Your programs must be written in C.

The executables programs must be called `server` and `client` respectively. Your programs must run on the School of Engineering Linux server `digitalis.eng.unimelb.edu.au`. And, if you elect to deploy your `server` on your NeCTAR cloud VM, obviously your executable must run on the OS configuration adopted (with an appropriately accessible port).

Getting Started

Under the Project 2A LMS menu item, you will be able to access the source code (`connect4.c`) for a text-based version of the game Connect 4.

Download, compile and run this version of the game.

Now that you have played the Connect 4 game a few times – some questions to think about:

- How could you re-design the source code to extract the key steps for the 'computer player' (or the game server) and the 'human player' (or the client)?
- How could threads be used to process the 'human' player's moves?
- What information must be transferred between the game server and the 'human player'? How does the 'human player' know the status of the game?
- How could a client-server architecture be used?
  - How might a multi-threaded game server for the Connect 4 be designed and implemented?
  - What steps are necessary to implement a client to play the game?

Requirements

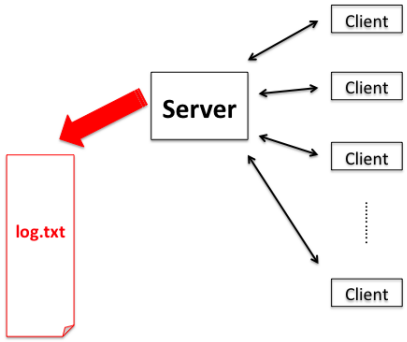

Figure 1: *A simple illustration of the client-server architecture. Your server must be able to handle concurrent requests. The server is responsible for maintaining a log file* (`log.txt`) *of interactions between each client and the server.*

We strongly suggest that you use `connect4.c` as the starting point for the design of both your `server` and `client` programs. We are not imposing tight constraints on design/requirements – you are free to design your `server` and `client` programs as you see fit. However, your implementation must meet the following criteria:

- When you implement sockets, you must use TCP (SOCK_STREAM for AF_INET).
- Your `server` program should take a port number as an argument (see details below).
- Your `client` program requires two command line arguments: an IP address (or host name) of the `server` and the corresponding port number (see details below).
- Your `server` program must be able to handle multiple/concurrent requests from individual `clients` (typically running on hosts with different IP addresses to the server).
- Your `client` program should display a text-based version of the Connect 4 board (with appropriate messages) for each step of the game, so that the 'human player' has the necessary information to make a move decision.
- The `client` program (used by the human player) moves first.
- To keep things simple, the only input you collect from the human player is an `int` value representing the column in which they wish to move. The `client` then transfer this `int` value (or nominated move) to the `server` for processing.
- You must make use of Pthreads to process each `client`'s moves (interactions with the `server`).
- It is important to note that `clients` do not play games against each other. ie. a client plays a game against the `server`.

- You must continually write to the log file `log.txt` detailing interactions between the `server` and individual `client`s (ie. the moves made by the player in the game). Each log file entry will include:
  - a time-stamp
  - IP address (or 'localhost' value) for the `client` or `0.0.0.0` for the `server`
  - a socket id (or file descriptor) for the `client` connection
  - details of the move played

  When a `client` connects to the `server`, the interaction should also be logged (before logging any moves).

  The first 6 lines from a sample log file are listed below. You can download the corresponding log file `log.txt` from the LMS.

```
[12 05 2015 14:27:37](128.250.26.183)(soc_id 5) client connected
[12 05 2015 14:27:39](128.250.26.183)(soc_id 5) client's move = 1
[12 05 2015 14:27:39](0.0.0.0) server's move = 2
[12 05 2015 14:27:41](127.0.0.1)(soc_id 6) client connected
[12 05 2015 14:27:42](127.0.0.1)(soc_id 6) client's move = 3
[12 05 2015 14:27:42](0.0.0.0) server's move = 5
```

- Care must be taken to make sure that the concurrent processing of interactions between the `server` and individual `client`s does not cause incorrect entries in the log file.

One final comment: Your game `server` does not have to implement advanced AI game playing techniques (such as using heuristics, minimax, alpha beta pruning etc, although you are welcome to do so if you wish).

Program execution / command line arguments

To run your `sever` program on `digitalis.eng.unimelb.edu.au`

`prompt: ./server [port_number]`

where `[port_number]` is a valid port number (eg. 6543) entered via a command line argument

To run your `client` program on `digitalis.eng.unimelb.edu.au`

`prompt: ./client localhost [port_number]`

where `localhost` corresponds to 'this computer'

and `[port_number]` is the valid port number (eg. 6543).

To run your `client` program on a different host machine

`prompt: ./client [host_name/IP_address] [port_number]`

where `[host_name/IP_address]` corresponds to host of your `sever`

and `[port_number]` is the valid port number (eg. 6543).

Note: A new log file `log.txt` file is created each time the `sever` program is started.

Submission details

Please include your *name* and *login_id* in a comment at the top of each file.

Our plan is to directly harvest your submissions on the due date from your SVN repository.

`https://svn.eng.unimelb.edu.au/comp30023/username/project2`

You must submit program file(s), including a `Makefile`. Make sure that your makefile, header files and source files are added/committed. Do not add/commit object files or executables. Anything you want to mention about your submission, write a text file called README.

If you do not use your SVN repository for the project you will NOT have a submission and may be awarded zero marks.

It should be possible to "checkout" the SVN repository, then type `make server` to produce the executable `server` and `make client` to produce the executable `client`.

Late submissions will incur a deduction of 2 mark per day (or part thereof).

If you submit late, you MUST email the lecturer, Michael Kirley <`mkirley@unimelb.edu.au`>. Failure to do will result in our request to sysadmin for a copy of your repository to be denied.

Extension policy: If you believe you have a valid reason to require an extension you must contact the lecturer, Michael Kirley <`mkirley@unimelb.edu.au`> at the earliest opportunity, which in most instances should be well before the submission deadline.

Requests for extensions are not automatic and are considered on a case by case basis. You will be required to supply supporting evidence such as a medical certificate.

Plagiarism policy: You are reminded that all submitted project work in this subject is to be your own individual work. Automated similarity checking software will be used to compare submissions. It is University policy that cheating by students in any form is not permitted, and that work submitted for assessment purposes must be the independent work of the student concerned.

Using SVN is an important step in the verification of authorship.

Assessment

This project is worth 10% of your final mark for the subject. Your submission will be tested and marked with the following criteria:

- **Client-Server interactions (4 marks)**
  - `server` and `client` programs run on `digitalis.eng.unimelb.edu.au`
  - `client` program displays appropriate messages (and the game board is displayed to *stdout*) at each stage of the game
  - it is possible to use the `client` program to play a complete game
  - `server` programs writes data to the log file `log.txt`
- **Concurrency (2 marks)**
  - `server` program is able to process two concurrent games, and record interactions in the log file; this test will be done manually
  - `server` program is able to process a large number of concurrent games (from a variety of IP addresses; up to 100 `client`s) and record interactions in the log file; this test will be done automatically using a script, thus it it essential that your client program only accepts an `int` value representing the nominated move at each stage of the game
- **Log file (2 marks)**
  - correctly documents interactions between the `server` and multiple `client`s using the format shown in the sample log file available on the LMS
- **Quality of code (2 marks)**
  - elegant code
  - coding standards followed (and consistent), Makefile works
  - use of header files, separate source files as appropriate
  - detailed documentation where necessary

Bonus marks available

- **NeCTAR cloud deployment (2 marks)**
  - `server` running on your VM functions correctly when tested
  - you must upload to your SVN repository a text file `ip.txt` that contains the IP number for your VM and port number.
  - Note: the total mark you achieved in this subject cannot exceed 100 marks.

Questions about the project specification should be directed to the LMS discussion forum.