CAB403 Assignment – Semester 2 2017

Nicholas Gosney – n9176004
Tristan Low – n9217126

a) Statement of Completeness:

Tasks Completed

- Task 1
- Task 2
- Task 3

Deviations from assignment specification:

- None that are known

Problems and deficiencies in solution

- None that are known

b) Team Info

Nicholas Gosney – n9176004
Tristan Low – n9217126

c) Statement of Contributions:

Both team members contributed equally to this assignment.

d) Data Structure for Leader Board

An ordered linked list was used for the Leader Board in our implementation. Each item contains Username, Games Won, Total Games played, Percentage game won (it's easier than recalculating each time you want to check it). Whenever a new player is added to the Leader Board, we search for the correct spot to insert the player to keep the list ordered. Likewise, if an existing player plays another game, we search for that player in the Leader Board, and update their position if necessary.

e) Critical Section Problem

The critical section problem was handled the same way as the reader-writer problem from Prac 5. Multiple users can read the Leader Board, but only one can write to it at the same time. This is accomplished through several Mutex locks. When a user goes to read the Leader Board, they lock a "Read Count" mutex to ensure they can check how many users are currently reading safely. If they are the first, they lock the "Write" mutex to prevent anyone from writing to the Leader Board whilst the user is reading. When a user stops reading the Leader Board, they lock the "Read Count" mutex again in order to check if they are the last reader. If they are, they unlock the "Write" mutext to allow writers to write again.

When a user tries to write to the Leader Board, the “Read” and “Write” mutexs are locked to prevent any other readers or writers trying to access the Leader Board. When the writer is finished, it unlocks both mutexs to allow other readers and writers to access the Leader Board. This ensures only one writer can access the Leader Board at once.

If any errors occur during reading the Leader Board somehow (e.g. client closes connection and fails to receive messages) then the mutexs are affected as if the reader finished reading in order to prevent infinite lock. The only real error possible in writing to the leader board is an “Out of Memory” error when trying to allocate memory for a new item. In this case, the entire server exits as gracefully as possible.

f) Threadpool Implementation

The threadpool implementation is essentially the same as that used in Prac 5. When the server is first run, MAX_USERS (10) number of threads are created that run the “handle_requests_loop” function and are passed a “threadId” (their sequential number from 0-9 inclusive). This function essentially has each thread looping and trying to get a request from a linked list (requests are created when main() receives a connection from a client). A mutex lock is used to prevent multiple threads trying to grab requests from the linked list at the same time. Once a thread has a request, it will use the fileDescriptor from that request for communication with the client and then when it’s finished it will close the connection and begin the loop to try and handle requests again. This requests structure essentially means that if more connections come in than threads can handle, the requests are queued up so that they can be handled as soon as a thread is free.

g) How to compile and run the program

Simply run the make file included in submission and run the files as specified in the assignment spec. i.e.

```
./server 12345
./client server_IP_address 12345
```

The port number for server is optional. If no port is specified, 12345 will be used by default.

In case somehow something goes wrong in submission, the make file is extremely simple and just compiles each program as follows:

```
gcc server.c -std=c11 -g -lpthread -Wall -pedantic -o server
gcc client.c -std=c11 -g -lpthread -Wall -pedantic -o client
```