Assignment 2: Distributed Communication

Due Date: **21st October 2014**

Weighting: **30%**

Group or Individual: **Individual/Group of Two**

1 Tasks

Task 1:

You only need to attempt this task if you are aiming to achieve a mark up to 74% for this assignment.

You need to develop a client/server system for professional dieticians. Dieticians can query statistics for various food items. The statistics of each type of food are stored in a text file, *calories.csv*, which is stored on the server. The information stored in the text file includes Food, Measure, Weight (g), kCal, Fat (g), Carbo (g), Protein (g).

The Client and Server will be implemented in the C programming language using BSD sockets on the Linux operating system. This is the same environment used during the weekly practicals. You will have acquired all the necessary knowledge and skills to complete the assignment by attending all the lectures and practicals, and completing the associated reading from the text book.

These programs are to run in a terminal reading input from the keyboard and writing output to the screen.

Server

The server will automatically load *calories.csv* file which is located in the same directory as the server binary file. The server listens on port 12345 by default, but can be configured to listen to a different port via an optional command line argument.

The server will take one command line parameter that indicates the port to listen on. The following command will run the server on port 12345.

```
./server 12345
```

The server will exit upon receiving a SIGINT from the operating system. SIGINT is a type of signal found on POSIX systems. On Linux this means ctrl+c has been pressed and the user wants to exit the program. The INT in SIGINT stands for interrupt. You will need configure a signal handler and ensure the server exits cleanly. In the 8th edition of the textbook, Section 4.4.3 Signal Handling on page 167 has an introduction to signal handlers and Section 21.9.1 Synchronisation and Signals on page 837 has a detailed description of signals on Linux. In the 9th edition of the textbook, Section 4.6.2 Signal Handling on page 183 has an introduction to

signal handlers and Section 18.9.1 Synchronisation and Signals on page 818 has a detailed description of signals on Linux.

Client

The client will take two command line parameters: hostname and port.
The following command will run the client connecting to a server on port 12345.

```
./client server's_IP_address 12345
```

The client program can be used to query information about a particular type of food. When a user enters a food name, the client program sends a request to the server and the server retrieves the information about the food and sends it back to the client program. The information about the food is then displayed on the client's console.

Below is an example:

```
Enter the food name to search for, or 'q' to quit:
>salmon

3 food items found.

Food: Salmon, Baked, Red
Measure: 3 oz
Weight (g): 85
kCal: 140
Fat (g): 5
Carbo (g): 0
Protein (g): 21

Food: Salmon, Canned, Pink, W/Bones
Measure: 3 oz
Weight (g): 85
kCal: 120
Fat (g): 5
Carbo (g): 0
Protein (g): 17

Food: Salmon, Smoked
Measure: 3 oz
Weight (g): 85
kCal: 150
Fat (g): 8
Carbo (g): 0
Protein (g): 18

Enter the food name to search for, or 'q' to quit:
>
```

If the name that the user enters is not in *calories.csv*, the system should display the error information as below:

```
Enter the food name to search for, or ‘q’ to quit:
>Basketball

No food item found.
Please check your spelling and try again.
```

Search Algorithm

Search terms are case-insensitive, so typing “ice cream” or “IcE cREAM” should return the same results. Matching should always start at the beginning of food name, i.e. searching for “Cake” should return “Cake or Pastry Flour” but not entries like “Carrot Cake”. Matching should also match the whole word, i.e. searching for “Butter” should return all the “Butter”, but not entries like “Buttermilk”.

If the user wants to terminate the client program, the user enters ‘q’.

The client/server system will be console based. No bonuses will be given for fancy or complex user interfaces.

The implementation for Task 1 does not need to handle concurrent connections and therefore does not need to be multi-threaded.

Note: Food name may or may not contain commas, which can be an issue when parsing the database. Please implement your parser to work around this issue.

Task 2:

<u>You need to attempt this task in addition to Task 1 if you are aiming to achieve a mark up to 84% for this assignment.</u>

The server you have implemented in Task 1 can only handle a single request at a time. You are required to extend the server to accept multiple concurrent connections using a thread pool.

The server will allow up to 10 clients to use the system at the same time. You will need to implement a thread pool where each incoming connection is handled in a separate thread. A thread pool reduces the cost of constantly creating and destroying threads to handle requests. Instead, threads are reused to handle connections. It also limits the number of incoming connections to guarantee the performance of the server.

You can reuse the producer consumer pattern from the first assignment, where the main thread accepts incoming sockets and places them in a queue (the producer) and 10 threads remove sockets from the queue and process the connection (the consumer).

Task 3:

You need to attempt this task in addition to Tasks 1&2 if you are aiming to achieve a mark up to 100% for this assignment.

The client/server programs in Task 2 do not allow the clients to add a new food item to the database. You are required to extend the client/server program in Task 2 to allow *any* of the clients to add a new food item to the database in addition to querying the database. When the server is terminated by pressing ctrl+c, the server should have saved the new food(s) into the text file *calories.csv*. In the original text file, *calories.csv* the food items are sorted in alphabetical order. If there are new food items added by the clients, all the food items, including those newly added food items, should be saved in the text file *calories.csv* and the food items in the text file should still be sorted in alphabetical order before the server is terminated; otherwise, there is no need to save the text file.

When adding a new food the implementation method is your design choice. For example, you may instruct the user:

```
Enter the food name to search, 'a' to add a new food item, or 'q' to quit:
```

If the user types 'a' then presses <enter>, the user is then prompted to enter the food name, as well as the attributes (Measure, Weight, Kcal, Fat, Carbo, Protein) for that food item.

How you elect to implement this is completely your own design choice. However, you should keep it simple and easy for the user to understand. All information must be added via the console using the keyboard.

Hints: Each of the clients can be a Reader or Write at any time. Thus, the solution to the first readers-writers problem introduced in the textbook (Section 6.6.2 on page 241 in the 8th edition or Section 5.72. on page 220 in the 9th edition) and in Lecture 5 Process Synchronisation can be used in Task 3.

2 Submission

Your assignment solution will be submitted electronically via Blackboard before 11:59pm on Tuesday, 21st October 2014. If you work in a group, you only need to submit one copy. Your submission should be a zip file and the file name should contain the student number of the contributors and include the following items:

- All source code of your solution.
- **README.txt** file with instructions on how to compile and run your program.
- The ***make*** file that you used for generating the executable file, if any.
- **A report in Word Document Format or PDF which includes:**
    a. A statement of completeness indicating the tasks you attempted, any deviations from the assignment specification, and problems and deficiencies in your solution.
    b. Information about your team, including student names and student numbers, if applicable.