The University of Queensland
School of Information Technology and Electrical Engineering
Semester One, 2012
CSSE2310 / CSSE7231 - Assignment 4

Due: 11:10pm 1 June, 2012

While I have recieved a request to honour the previous code and its students by setting the deadline at 23 : 03, *I think that could cause confusion.*

Marks: 50
Weighting: 25% of your overall assignment mark (CSSE2310)
Revision 1.4

Introduction

Your task is to write a networked trivia game in c99. This will require three programs: a client (called `trivial`) a server (called `serv`) and a program to query the scores (`scores`).

This assignment will require the use of threads, tcp networking and thread-safety. Your assignment submission must comply with the C style guide available on the course website. As with Assignment 3, tabs are allowed but all source code will be run through `expand` before marking.

This assignment is designed to be completed in stages. Good design and thinking about the final target would help but it is important to *design*, implement and *test* each stage before moving on to the next one. Do not attempt the whole thing in one go.

This is an individual assignment. You should feel free to discuss aspects of C programming and the assignment specification with fellow students. You should not actively help (or seek help from) other students with the actual coding of your assignment solution. It is cheating to look at another student's code and it is cheating to allow your code to be seen or shared in printed or electronic form. You should note that all submitted code may be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion, formal misconduct proceedings will be initiated against you. A likely penalty for a first offence would be a mark of 0 for the assignment. Don't risk it! If you're having trouble, seek help from a member of the teaching staff. Don't be tempted to copy another student's code. You should read and understand the statements on student misconduct in the course profile and on the school web-site:
`http://www.itee.uq.edu.au/itee-student-misconduct-i`

In this course we will use the subversion (svn) system to deal with assignment submissions. Do not commit any

code to your repository unless it is your own work or it was given to you by teaching staff. **If you have questions about this, please ask.**

Invocation

The client ("`trivial`" from now on) takes the following arguments.

1. The name of the player.
2. The port the game server is listening on.
3. The hostname to connect to. This parameter is optional and if not given, it should default to the local machine.

There is no requirement that player names be unique. Two clients could connect to a game using the same name. If an incorrect number of arguments is given, then the following message should be printed to standard error:

```
Usage: trivial name port [host]
```

The exit status is given in table later in this specification.

The server ("`serv`" from now on) takes the following arguments:

1. Time allowed to answer each question (in seconds)
2. The minimum number of players required for a game to start/continue.

3. The maximum number of players allowed in a game.
4. port number to listen on
5. the name of a file containing questions to ask
6. (optional) additional pairs of parameters giving a port and a question file

The time to answer questions must be > 0. All ports must be integers 1 <=port<= 65535. If an incorrect number of arguments is given, then the following message should be printed to standard error:

```
Usage: serv round_time minplayers maxplayers port q
```

The exit status is given in table later in this specification.

The `scores` program takes the following arguments:

1. the port to connect to
2. hostname [optional]

The usage message is:

```
Usage: scores port [host]
```

trivial output

Before each question, the server will send a scores line. After printing this line and a blank line, the lines which make up the question are printed. [Note that the terminating `.` is

not displayed]. The options block is preceded by ===== and followed by ++++. Each option is formatted as "%d: %s", numbered from 1. For example:

```
Hello Player 1/1
Scores: T2:0 Joel:0

My life for:
=====
1: chocolate
2: aiur
3: spiders
4: ire
5: cats on the internet
++++
```

After the time for the question has run out, display the results and scores lines sent by the server.

```
Results: T2:TimedOut Joel:TimedOut
Scores: T2:0 Joel:0
```

If the game is over the client will then display the winners line from the server. (Note that the server, does not send the lines in this order.)

```
Winner(s): T2 Joel
```

If the game is not over, print a blank line and display the next question.

Formats and protocols

Question file format

Questions in the question file are stored in the following form:

```
<One or more lines which make up the question.>\n
----\n
number_of_options correct_option\n
<options, one per line>\n
\n
```

Options are numbered contiguously starting at 1. Note that the question description ends in a blank line.

A question file consisting of 2 could look like:

```
1+2*3=
----
6 4
one
three
five
seven
nine
eleven

What operator is missing from the following express
1+5 ? 3=3
----
6 4
?:
*
+
%
^
/
```

Questions sent to clients

When questions are sent to clients they are sent in the following form:

```
<one or more lines containing the text of the quest
.\n
number_of_options\n
<options, one per line>\n
```

Note that the end of the question text is indicated by a lone '.' and that there is no blank line following the question.

`trivial` ⇒ `serv`

On connecting, the client should send their name followed by a newline. Apart from this, the only things the client should send are integers representing the correct answers to questions. Each integer should be followed by a newline.

Note that `trivial` should not (knowingly) send an invalid guess to the server. If the user enters an invalid guess (< 1, greater than the biggest option, not an integer), then the client should print `Invalid guess` followed by a newline to stdout and read another guess.

What should the server accept from the client? Remember that the server might be running against a buggy client. Do not assume that just because your client behaves correctly that every client does. The server should only accept input consisting of digits and '\n'. If any other chars are recieved the server will disconnect.

It is still possible to have bad inputs that only consist of permitted chars. For example, blank lines or a long string of digits, or "0". These will always be wrong but will not cause a disconnect. If it would help, you can assume that no question will ever have more than 999 possible answers.

What should the client accept from the user? For example, is "1batman" valid or not? You can decide this yourselves[1]? That is, we will not expect sensible answers from your programs with client input like that. However, we will expect your client to keep processing further questions. It must not shutdown or loop when fed input like that.

`scores` ⇔ `serv`

`scores` should connect to **`serv`**, send the string "**`scores`**" followed by a newline. It should then print whatever the server sends back to **`stdout`**.

[1] My version accepts anything that atoi will process.

`serv` must allow score queries even if the game running on the port is full. For each player (name) who has connected since the server started send back a row containing the following text:
name played:? won:? disc:? score:?
Where the **?** should be replaced with the number of games that name has connected to, won, disconnected early from and their total score over all games. (If a player connects, then disconnects and reconnects this counts as two games)

The "connected" counter should be updated as soon as a player connects to a game (connecting to a full game doesn't count). The other counters should be updated when the game ends or the player disconnects.

Note: there is a single score table for all ports that the server is listening on. Connecting to any of the servers ports should produce the same results.

Communication protocols

If a game is full when `trivial` attempts to connect, the server must send a line beginning with $. Then disconnect that client.

See Figure 1 for an illustration. Each of the send steps shown in the figure consists of sending the same text to all connected clients. (The hello message is only sent to the

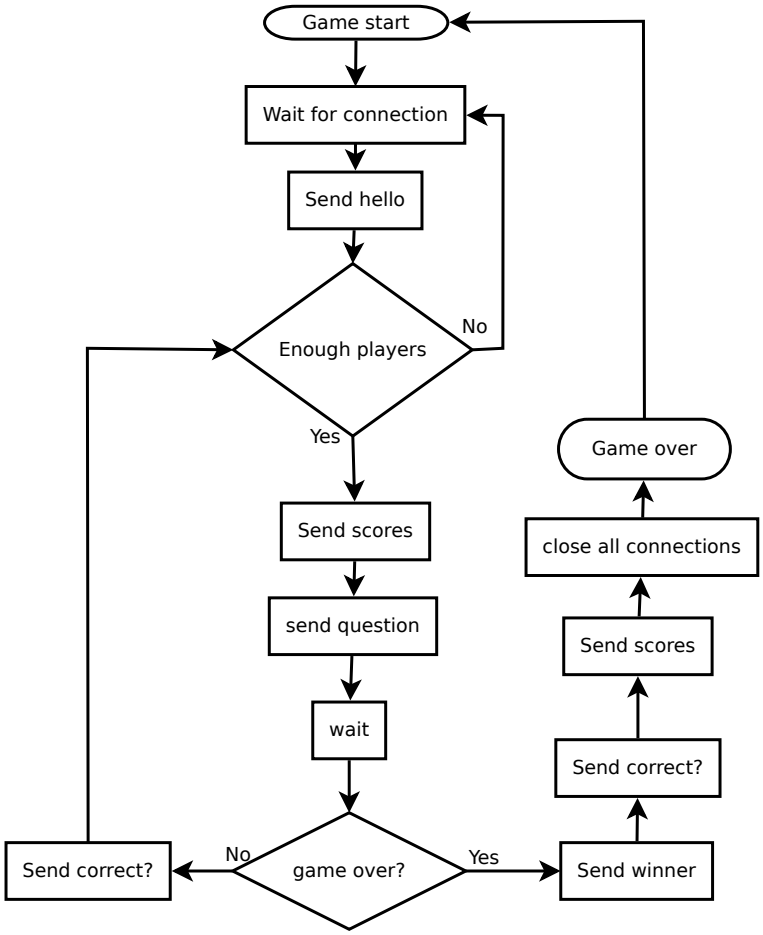

Figure 1: Main steps for the server playing a game

new player).

- Send scores — An 'S' followed by *name*:*score* for each player (space separated).
- Send winner — A 'W' followed by the names of all players with the maximal score (in the current game).
- Send correct? — A 'C' followed by *name*:*result* for each player (space separated). Where *result* is one of "TimedOut", "Correct", "Incorrect". TimedOut should be sent when the client does not send an answer in time.

These leading characters should be stripped before the client displays them.

If the client receives a line which does not follow this protocol, it should exit with an error (see table later).

Compilation

Your code (all three programs) must compile with command:

`make`

Each individual file must compile with at least `-Wall -pedantic -std=gnu99`. You may of course use additional flags (eg `-pthread`) but you must not use them to try to disable or hide warnings. You must also not use pragmas to achieve the same goal.

If any errors result from the make command (ie no executables can be created), then you will receive 0 marks for functionality. Any code without academic merit will be removed from your program before compilation is attempted (and if compilation fails, you will receive 0 marks for functionality). If your code produces warnings (as opposed to errors), then you will lose style marks (see later).

Your solution must not invoke other programs. Your solution must not use non-standard headers/libraries.

Stages

The following are suggestions not requirements:

1. Write `scores` first, use netcat as a server and test against that.
2. `trivial` client
3. `serv` running on one port handling one game then exit.
4. `serv` starting a new game when the previous one finishes.
5. `serv` running games on multiple ports.
6. `serv` correctly handling disconnecting players.[2]

[2]Changes in players should only take affect when a new question is sent. For example, if the game was full and a player disconnected, no more players can be admitted to the game until the next question. Implementing these features will require modifications to the states shown in Fig. 1

7. `serv` allowing players to connect to games which have room.[1]

Exit status

scores

Condition	**Status**	**Message to stderr**
Insufficient arguments	1	Usage: scores port [host]
Invalid port#	4	Invalid Port
Unable to connect to server	5	Bad Server
System error	8	System Error
It's all good	0	

trivial

Condition	**Status**	**Message to stderr**
Insufficient arguments	1	Usage: trivial name port [host]
Invalid port#	4	Invalid Port
Unable to connect to server	5	Bad Server
System error	8	System Error
End of input on client before the end of the game	9	Client EOF
Server disconnected at an unexpected time	10	Server Disconnected
The game on the specified port is currently full	11	Server Full
Server does not follow protocol	12	Protocol Error
It's all good	0	

serv

Condition	**Status**	**Message to stderr**
Insufficient arguments	1	Usage: serv round_time minplayers maxplayers port qfile [port qfile ...]
Round time, max or min players is not a positive integer	2	Bad Number
Can't open question files or fail to read a valid question from file	3	Bad File
Invalid port#	4	Invalid Port
Unable to listen on a port	6	Bad Listen
System error processing an incoming connection	7	Bad Client
System error	8	System Error
It's all good	0	

Style

You must follow version 1.6 of the C programming style guide found at:
`http://courses.itee.uq.edu.au/csse2310/2012s1/resources/c_resources.html` All tab characters will be replaced using the `expand` tool before assignment are marked.

Submission

Submission must be made electronically by committing using subversion. In order to mark your assignment the markers will check out `/ass4/trunk` from your repository on `svn.eait.uq.edu.au`. Code checked in to any other part of your repository will not be marked.

The due date for this assignment is given on the front page of this specification. Note that no submissions can be made more than 120 hours past the deadline under any circumstances.

Test scripts will be provided to test the code on the trunk. Students are strongly advised to make use of this facility after committing.

Note: Any .h or .c files in your trunk will be marked for style even if they are not linked by the makefile. If you

need help moving/removing files in svn, then ask.

You must submit a `Makefile` or we will not be able to compile your assignment. Remember that your assignment will be marked electronically and strict adherance to the specification is critical.

Additional requirements

Marks

Marks will be awarded for both functionality and style.

Style (6 marks)

If g is the number of style guide violations and w is the number of compilation warnings, your style mark will be the minimum of your functionality mark and:

$$6 \times 0.9^{g+w}$$

The number of compilation warnings will be the total number of distinct warning lines reported during the compilation process described above. The number of style guide violations refers to the number of violations of version 1.6 of the C Programming Style Guide. A maximum of 5 violations will be penalised for each broad guideline area. The broad guideline areas are Naming, Comments, Braces,

Whitespace, Indentation, Line Length and Overall. For naming violations, the penalty will be one violation per offending name (not per use of the name) up to the maximum of five. You should pay particular attention to commenting so that others can understand your code. The marker's decision with respect to commenting violations is final — it is the marker who has to understand your code. To satisfy layout related guidelines, you may wish to consider the `indent(1)` and `expand(1)` tools. Your style mark can never be more than your functionality mark — this prevents the submission of well styled programs which don't meet at least a minimum level of required functionality.

Functionality (44 marks)

Provided that your code compiles (see above), you will earn functionality marks based on the number of features your program correctly implements, as outlined below. Partial marks will be awarded for partially meeting the functionality requirements. Not all features are of equal difficulty. If your program does not allow a feature to be tested then you will receive 0 marks for that feature, even if you claim to have implemented it. For example, if your program can never open a file, we can not determine if your program would have loaded input from it. The markers will make

no alterations to your code (other than to remove code without academic merit). Your programs should not crash or lock up/loop indefinitely.

- `scores`
  - Exit status and messages (2 marks)
  - Correct operation (3 marks)
- `trivial`
  - Arg processing and exit status (2 marks)
  - Correctly handle attempt to connect to a full game (ie one which already has the maximum number of players.) (2 marks)
  - Correctly handle a game with one question (2 marks)
  - Correctly detect invalid guesses (2 marks)
  - Play games consisting of multiple questions, invalid guesses, badly behaved servers, early end-of-input ... (4 marks)
- `serv`
  - One game, one player, one question (2 marks)
  - One game, multiple players, one question (2 marks)
  - One game, multiple players (2 marks)
  - Correctly handle attempts to connect to full game (2 marks)
  - Game with disconnecting players (2 marks)
  - A sequence of games on the same port (4 marks)
  - Serve multiple ports (4 marks)
  - Allow players to join non-full games (4 marks)
  - correctly produce scores (3 marks)
  - No memory leaks (according to valgrind[3]) (2 marks)

Some of the marking items depend on other items. For example, Playing a full game on the client relies on being

[3]In order to be eligable for these marks, you must at least support a sequence of games on the same port.

able to handle a game containing only one question. In some cases, valgrind may report memory as leaked when it is unavoidable or out of the programmers control. To deal with such cases we will provide a supressions file to exclude those cases. We are only concerned with leaks which valgrind reports as "definitely lost".

Be careful with your resource usage and make sure to clean up While it is true that resources are (generally) cleaned up when the process exits, servers are expected to be able to run for long periods of time.

Late Penalties

Late penalties will apply as outlined in the course profile.

Specification Updates

It is possible that this specification contains errors or inconsistencies or missing information. It is possible that clarifications will be issued via the course website. Any such clarifications posted 5 days (120 hours) or more before the due date will form part of the assignment specification. If you find any inconsistencies or omissions, please notify the teaching staff.

Notes and tips

- Start early.
- If any aspect of the spec is unclear, inconsistant or incomplete then **ask questions**.
- netcat (`nc`) can function as a TCP client or server. It is your friend, learn to use it.
- Think about how many threads you will need and what they will do. Do this **before** you start coding. Writing code to experiment with the concepts is a good idea **but** you must be able to explain to tutors what your threads are supposed to be doing when you ask for help. For example, the client will need to listen to both the user and the server at the same time.
- The server must be multi-threaded (no `select()`).
- Clients disconnecting unexpectedly must not crash your server.
- The server does not clear input streams before each question is sent so if the client sends multiple guesses in succession or sends a guess late, they will be counted as the answers to future questions.
- `scores`

  - Once a connection has been established, any read/write errors should be interpreted as the server disconnecting.
  - Any system call failures, prior to the connection being established are grouped under — couldn't connect.
  - Even if the server is running games on multiple ports, there is still only one score table. Connecting to any valid server port should display scores from all games on all ports.

- Some useful functions to look at:
  - sigwait
  - pthread_sigmask

- Memory leak marks: If you wish to attempt the no leaked memory part of the assignment, you will need to add an additional signal handler for SIG_HUP. After your server receives SIG_HUP it should not start any new games. It should allow any games currently running to finish (waiting for extra connections if there are not enough players). Once all games have finished, the server should close down. To get the marks, valgrind must not report any memory as definitely lost.

  Please do not spend time on this part unless you have a resonable portion of the server implemented successfully.

Updates

1. 1.0.1 → 1.2

   - As soon as `trivial` connects and gives their name, the server sends:

     `Hello Player x/y.`

     Where x is the current number of players (after adding the new player) and y is the minimum number of players required to continue the game.
   - The "Insufficient arguments" error also applies if `argc` for `serv` is an odd number greater than 6. (This would happen if a port is given with no qfile).
   - Moved exit status 8 and 0 into the table for clarity. So when to use System Error vs Bad Server/Bad Client? In a more thorough version of this type of program we would distinguish different types of failures for each operation. In this assignment though, failed system calls which relate to establishing the network connections will be reported as Bad Server/Client. These calls could be expected to fail under normal operation. For example if the port you are attempting to connect to does not have a server on it.

     System Error is reserved for calls which would not

be expected to fail under normal circumstances. For example, fdopen or pthread_create failing would be a System Error.

2. 1.2 → 1.3
   (a) All communication between the client and server is as text. Do not send integers as raw bytes.
   (b) Emphasised the single score table for all ports in the scores section.
   (c) Consistant usage instructions for serv
   (d) More explanation about sending a guess before knowing whether it was too large in the `trivial` ⇒ `serv` section. There are also some changes here about what answers the server will accept.
   (e) ~~The records returned by the `scores` program should be in order that the players first connected in.~~
   (f) We will not test situations where the minimum players required is greater than the maximum allowed.
3. 1.3 → 1.4
   (a) We will not check the order that records appear in the score table.
   (b) `trivial` now has a new error message and exit code for when the server doesn't follow the protocol.

(c) Clarified the marking criteria (what does “full” game mean?)