The University of Queensland
School of Information Technology and Electrical Engineering
Semester Two, 2014
CSSE2310 / CSSE7231 - Assignment 4
Due: 11:10pm 31 October, 2014
Marks: 50
Weighting: 25% of your overall assignment mark (CSSE2310)
Revision 4.0

Introduction

Your task is to write a networked card game in c99. This will require two programs: a client (called `2310client`) and a server (called `2310serv`).

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 Blackboard area.

Your programs must not create any files on disk not mentioned in this specification. 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-including-plagiarism`

As with Assignment 1, 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.**

Organisation

The game in this assignment uses the same game rules as Assignment 3 (the user interface will be different). However, instead of computer players connected to a hub via pipes, the players will be TCP clients connecting to a server. Also, the client will read its moves from `stdin` rather than choosing moves automatically.

To organise players into games,when a client connects it will tell the server the name of the game it wants to join. If there is a game with that name with room, the player joins that game. If there is no game with the requested name and room for the new player, a new game will be created for them to join. A game does not begin until it is "full". The number of players needed to fill a game is determined by the game's name. A game with a name starting with:

- '2' — will be a two player game.
- '3' — will be a three player game.
- anything else — will be a four player game.

The server will be able to listen for player connections on multiple ports. Games on different ports will be kept separate (that is, players connecting to one port will not join games on another port).

Within individual games, players will be arranged in increasing order (as determined by `strcmp`). For example: if players called Henri, Albert, William and Frances join the same four player game (in that order); then Albert will be player A, Frances will be B, Henri will be C and William will be D.

Code from Assignment 3

Depending on how it was written, it may be possible that code from Assignment 3 will be useful in this assignment as well(particularly the game logic). You have two options:

a. You may use the code from **your** Assignment 3 as the basis for Assignment 4.

b. You may use the lecturer's sample solution for Assignment 3 as the basis for Assignment 4.

Pick exactly **ONE** of the above. Using someone else's Assignment 3 code will be regarded as plagiarism. Put the Assignment 3 source code you are basing your Assignment 4 on into your repository at `/trunk/ass4/previous`. Then copy it into `/trunk/ass4` and start editing.

Server Invocation

The parameters to start the server are:

- A port number to use for administration commands. (This will be explained later).
- Pairs of arguments specifying:
  - Port to listen on.
  - Decks file to use for games on that port.

For example:
`./2310serv 4000 4001 ex.decks`
Would start a server listening for clients to connect on port 4001. All games played from that port will use the decks from ex.decks.
`./2310serv 4000 4001 ex.decks 4002 ex2.decks`
Will listen for clients on ports 4001 and 4002. Clients connecting to 4001 will play games using ex.decks while those connecting to 4002 will play games using ex2.decks. The format of the decks files is the same as in Assignment 3.

Client Invocation

The parameters to start a client are:

- The name of the player — there are no checks to ensure that player names are unique.
- The name of the game the player wishes to join.
- The port to connect to.
- The name of the host to connect to. [optional defaults to `localhost`].

For example:
`./2310client timmy 3player 4001 localhost`

Communications

The messages between the server and clients will be the same as the ones between hub and players with the following changes:

- The client will not send the initial '-' which the player sent.
- When a client connects to the server it will send (each name is newline terminated)
  - the player name
  - the game name
- When a game begins, the server will send each player:
  - The number of player in the game, the letter for the player (space separated and newline terminated).
  - The names of all players (in order, newline terminated).
- After each move has been sent by a client, the server will respond with one of two messages (newline terminated):
  - YES
  - NO

  In the case of NO, the server has rejected the move because it is invalid for some reason. The client must prompt its user for a new move.
- When sending the message to the hub when playing a `1`, send either both target and guess or neither. For example: `1B4` or `1--` but not `1-4`.

Client interface

When a client's turn comes, the following information will be displayed:

- A row for each player: ?(???)?:????? — where the missing information is, the player letter, the player's name, a character indicating the player's status (see Assignment 3), the cards which that player has discarded.
- `You are holding:??` — The card(s) the player is holding.
- A prompt: `card>` — no trailing space.

For example:

```
A(bob) :562
B(tom) :1
You are holding:73
card>
```

If the card chosen is legal (see rules from Assignment 3) and requires a target, then prompt:
`target>` (no tailing space).

If the card needs a guess, then prompt:
`guess>` (no trailing space).

If both a target and a guess are required, then get the target first. If the input is not valid at a particular prompt, the prompt is repeated (so an invalid target will result in another target prompt, not a new card

prompt). If the server rejects a move, the client should prompt for a card again (but not display the status information). Do not prompt for a target or a guess, if there are no valid targets. For example if the card is '6' but all other players have played '4' or have been eliminated. If the player wishes to target themselves with '5', then they should enter their own letter as the target.

When a `thishappened` message arrives from the server, the player should display it in the same way the hub did in Assignment 3. For example:

```
Player B discarded 5 aimed at D. This forced D to discard 8.  D was out.
```

When a `scores` message arrives, the client should display:
`Scores:` followed by *playername=score*
Items to be space separated (names in player order). For example:

```
Scores: nicola=2 schmidt=3
```

When a `gameover` message arrives, the client should display: `Game over`

Client error messages and exit status

Conditions should be tested in the order given in tables.

Condition	**Status**	**Message**
It's all good	0	
Incorrect number of arguments	1	Usage: client name game_name port host
Player name is empty (or contains a newline char)	2	Invalid player name
Game name is empty (or contains a newline char)	3	Invalid game name
Port is not an integer > 0 and ≤ 65535	4	Invalid server port
Unable to connect to specified host and port	5	Server connection failed
Failure to read game setup information	6	Invalid game information received from server
Badly formed message received	7	Bad message from server
Connection to server was broken without a `gameover` message	8	Unexpected loss of server
Encountered end of input reading from the user	9	End of player input
Unexpected system call failure (not tested)	20	Unexpected system call failure

Server interface

The server should not produce any output to `stdout` or `stderr` apart from messages given in the following table.

Condition	Status	Message
It's all good	0	
Incorrect number of arguments	1	Usage: 2310serv adminport [[port deck]...]
Unable to open decks file for reading	2	Unable to access deckfile
Problem with the contents of the decks file	3	Error reading deck
Port number is invalid (bad number or port is known to be in use)	4	Invalid port number
Problem listening on port	5	Unable to listen on port
Other system call failure	9	System error

Exit status 9 is not something we will test for, but it is provided to give people something to return in the case of unusual failure.

If these problems occur as a result of dealing with command line arguments, they should be printed to `stderr` and the server exits with the specified status.

Note there is no error message for a client disconnecting early. In such cases, the game is finished early (send `gameover` to all players). There will be no game winner recorded in such cases but rounds won still count.

Admin port

The first command line argument gives a (TCP) port to listen on for administration commands. The admin port will only take one connection at a time. It prints no prompt, and any invalid commands are silently ignored. The two commands it accepts are:

- P — start listening for client connections on an additional port. Eg: `P4001 ex3.decks`
- S — (no arguments), sends the client a table of statistics (sorted by player name, items separated by commas):
  - Number of games played (this includes incomplete games)
  - Number of rounds won
  - Number of games won (including ties).

For example (responses by server are in indented — this is for clarity only, your code should not indent its responses):

```
nc localhost 4000
S
   Alfonz,2,1,0
   Harry,3,5,1
   Willem,3,9,2
   OK
P4002 ex.decks
   OK
P4002 ex.decks
```

```
   Invalid portnumber
hello
hello
```

Note that after a successful command execution, `OK` is printed. If the command caused an error in the table above, the message should be sent back down the connection to the client instead of to the server's `stderr`, but the server will not exit. The statistics should be updated after each game has completed.

Stages of this assignment

Do not attempt to write the whole assignment in one go. The following is a suggestion as to a sane path. If you use modular coding and data structures, you will find later stages easier. But be sure to code and test each stage before moving on to the next one.

1. Client error checking. You can validate the command line args up to and including opening a connection without any server code at all (use `nc -l` to act as a server).
2. Server command line args — listening on the admin port and one game port. Note that you don't need to actually do anything with the admin port until much later, just make sure you can listen on it.
3. Get the client to play a complete game connected to a netcat server. Don't forget the extra messages the server needs to send.
4. Write the server to take the first two players it gets and play a game with them.
5. Extend server to determine the number of players from the game name.
6. Extend the server to match players by game name.
7. Extend the server to play multiple games simultaneously.
8. Enable the `S` command on the admin port.
9. Extend server to be able to listen for connections on multiple ports.
10. Enable the `P` command on the admin port.

Compilation

Your code must compile (on a clean checkout) with the 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 the make command does not produce one or more of the required programs, then those programs will not be marked. If none of the required programs are produced, then you will receive 0 marks for functionality. Any code without academic merit will be removed from your program before compilation is attempted [This will be done even if it prevents the code from compiling]. 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.

Submission

Submission must be made electronically by committing using subversion. In order to mark your assignment, the markers will check out `/trunk/ass4/` from your repository on `source.eait.uq.edu.au`. Code checked in to any other part of your repository will not be marked. Your starting point from Assignment 3 must be checked in to `/trunk/ass4/previous` as described earlier.

The due date for this assignment is given on the front page of this specification. Note that no submissions can be made more than 96 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.

Marks

Marks will be awarded for both functionality and style.

Functionality (42 marks)

Provided that your code compiles (see above), you will earn functionality marks based on the number of features your programs correctly implement, as outlined below. Partial marks may 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. Your programs should not run for unreasonably long times.

- Bad client args checking (4 marks)
- Bad server args checking (4 marks)
- Run a single game on one port (10 marks)
- Run multiple concurrent games on one port (8 marks)
- Run concurrent games on multiple ports (8 marks)
- Admin port supports **P** command. (4 marks)
- Admin port supports **S** command. (4 marks)

Style (8 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:

$$8 \times 0.9^{g+w}$$

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 the current 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.

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.

Test Data

Test data and scripts for this assignment will be made available. The idea is to help clarify some areas of the specification and to provide a basic sanity check of code which you have committed. *They are not guaranteed to check all possible problems nor are they guaranteed to resemble the tests which will be used to mark your assignments.* Testing that your assignment complies with this specification is still *your* responsibility.

Notes and Addenda:

1. Your programs must not invoke any other programs.
2. Be sure to test on moss.
3. You should not assume that system calls always succeed.
4. You may assume that only 8bit ASCII characters are in use[no unicode].
5. Tab stops are assumed to be 8 characters wide.
6. You may not use any `#pragma` in this assignment.
7. Client disconnection should be registered when the player is expected to supply input and the server or client registers `EOF` instead. Incomplete messages will trigger the invalid message code rather than immediate disconnect.
8. Where your program needs to parse numbers (as opposed to characters) from strings, the whole string must be a valid number. e.g. `"3biscuit"` is not a valid number, nor is `"3 "`

9. Neither program should assume that the other will be well behaved.
10. We will not be performing tests with more than one command line error for the server. That is, the server needs to check messages from the players and the players need to check messages from the server.
11. When you need network ports please use the ones assigned to you (run `2310port` - you may use that port and the two next ports).
12. Invalid messages which you need to check for:
    (a) Badly formatted messages (including trailing space).
    (b) Messages which refer to non-existant card types or players.
    (c) Messages which indicate a non-standard deck is in use.
    (d) Messages which arrive out of order. Please note that `gameover` could arrive at any time (due to one of the other players leaving early).

Preliminary→ 4.0

1. The following functions are banned for this assignment:
    - `fork()`
    - `select()`, `poll()` and anything similar.