
The University of Queensland
School of Information Technology and Electrical Engineering
Semester One, 2009

COMP2303 / COMP7306 – Assignment 4

Due: 11pm Friday June 5, 2009

Marks: 50

Weighting: 25% of your overall assignment mark (COMP2303)

Introduction

The goal of this assignment is to create a multithreaded network game server to play "noughts and crosses". You will be expected to program to the course style guide.

Note: Your server must follow the protocol as given in this specification but you are not required to write a game client – you can interact with your server using telnet.

This is an **individual assignment**. You should feel free to discuss aspects of C programming and the assignment specification with your fellow students, but you shouldn't 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 will 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 website:
http://www.itee.uq.edu.au/about_ITEE/policies/student-misconduct.html

Assignment Specification

A detailed specification is provided below.

Rules of the Game

Two players take turns placing marks (noughts 'O' or crosses 'X') into empty cells in a three by three grid. A player wins the game by forming a line of three of their marks (horizontally, vertically or diagonally). The game is a draw if the grid fills up but no line is formed.

Startup

Your submitted code should compile and link to a single executable called `ncserver`. Your program should accept two command line arguments: – the port number on which the server listens for incoming client connections; and the port number used to monitor the status of games in play. The second will be used in advanced functionality; but even if you do not attempt advanced functionality you must still accept the argument. For example, you might run the server as:

```
./ncserver 2303 7306
```

If the arguments aren't of the expected number (2 after the command name) and form (first and second arguments are integers from 1024 to 65535 inclusive), then your program should print the following message (with a following newline) to standard error and exit with an exit status of 1:

```
Usage: ncserver client-port-num board-port-num
```

Listening

Your game server should listen on the given client port number (on all of the machine's IP addresses) for incoming connection requests from clients. If your server is unable to listen on the given port number, it should print the following message to standard error (with a following newline) and exit with an exit status of 2:

```
ncserver: Unable to listen on port port-num
```

where *port-num* is replaced by the first port number given on the command line.

(Advanced functionality) Your game server should also listen on the given board port number (on all of the machine's IP addresses) for incoming connection requests.. If your server is unable to listen on the given port number, it should print the following message to standard error (with a following newline) and exit with an exit status of 3:

```
ncserver: Unable to listen on port port-num
```

where *port-num* is replaced by the second port number given on the command line.

Your program must check for the ability to listen on ports in the order in which they appear on the command line, i.e., if your program can't listen on the first port number, it should exit with the appropriate error message and never even check the second port number. The only text that your program may output to standard error is one of the error messages above.

Server Behaviour

Your server will listen for incoming TCP connections on the client port and "pair-up" clients to play games. The first player (client) to connect will play the second player to connect, the third will play the fourth etc. The first player of each pair will be assigned noughts (O), the second will be assigned crosses (X). If a player disconnects before being paired-up (can only be the first player of the pair) then the server should behave as if they never connected in the first place (i.e. the next player to connect will be the first player in the pair).

Protocol description

Communication between clients and the server uses an ASCII text line based protocol. All lines transmitted must end with a `\r\n` (carriage return and newline). Each line will contain no more than five(5) chars including line endings. The protocol is stateful, i.e. it relies on the client and server maintaining state – in this case, knowledge of the state of the game. Only changes to the board (i.e. moves) are transmitted.

Board positions (xy) are integers 0, 1, 2 in each direction and have the following meaning:

00	10	20
01	11	21
02	12	22

The following request message type is the only message type expected to be sent from a client to a server:

Message	Meaning	Comment
Mxy	Indicates position on the board that player (client) wishes to place their mark.	Client must be connected first.

The following messages are sent by the server to the clients.

Message	Meaning	Comment
WO	You are player O. Wait for a further message indicating that another player has joined your game.	Sent to new connections (first player).
WX	You are player X. Wait for another message.	Sent to new connections (second player).
T	It's your turn.	Sent to 'O' at the beginning of the game after the 'X' player connects. Otherwise, sent after an M message (see below)
Mxy	The other player placed their mark at x, y	Sent to one player immediately after receiving a valid move from the other player. Will be immediately followed by a T, G or D message (see below)
!	Message out of turn.	Sent to a player any time a message is received from them when it is not their turn.
?	Message not understood.	Sent to a player who sends a message on their turn but the message is incorrect (i.e. not one of M00, M01, M02, M10, M11, M12, M20, M21, M22.)
B	Bad move. The requested board location is occupied.	Sent to a player who sends a well formed move but the location is not available. It is still the player's turn and they should send another move.
O	Move accepted	Sent to a player when the server accepts their move. The player's turn is over when this message is sent. If the game is over, the server will immediately follow this with a G or D message (see below).
GO GX	The game is over and the winner was O/X.	This message will be sent to both clients when there is a winner. The server will immediately disconnect from both clients after sending this message.
D	The game was a draw. (Happens when all 9 squares are filled with no line of O's or X's; OR when one of the players disconnects before the game is over.	This message will be sent to both clients (or the remaining connected client) when the game is a draw. The server will immediately disconnect from both clients (or the remaining connected client) after sending this message.

Disconnect behaviour

When this specification says that a message is sent to a player but they have disconnected already, the program should continue. No error message should be printed.

Sample game

A sample session is given below. Arrows indicate the direction of the information flow (to or from the server). For clarity, only one client is shown communicating at a time but this should not be assumed by the implementation. You must support simultaneous communication by both clients (and indeed, multiple simultaneous games being played by multiple pairs of players).

First Player (O)	Server	Second Player (X)	Board State									
Connects	→											
	← WO											
		← Connects										
	← WX	→										
	← T											
		← M00										
	← !	→										
Help	→											
	← ?											
M00	→											
	← O		<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>O</td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr></table>	O								
O												
	M00 →											
	T →											
		← M11										
	O →		<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>O</td><td></td><td></td></tr><tr><td></td><td>X</td><td></td></tr><tr><td></td><td></td><td></td></tr></table>	O				X				
O												
	X											
	← M11											
	← T											
M11	→											
	← B											
		← M01										
	← !	→										
M01	→											
	← O		<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>O</td><td></td><td></td></tr><tr><td>O</td><td>X</td><td></td></tr><tr><td></td><td></td><td></td></tr></table>	O			O	X				
O												
O	X											
	M01 →											
	T →											
		← M22										
	O →		<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>O</td><td></td><td></td></tr><tr><td>O</td><td>X</td><td></td></tr><tr><td></td><td></td><td>X</td></tr></table>	O			O	X				X
O												
O	X											
		X										
	← M22											
	← T											
M02	→											
	← O		<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>O</td><td></td><td></td></tr><tr><td>O</td><td>X</td><td></td></tr><tr><td>O</td><td></td><td>X</td></tr></table>	O			O	X		O		X
O												
O	X											
O		X										
	M02 →											
	← GO											
	← GO	→										

Threads

Your server should be multi-threaded using the pthreads API, with the thread functionality described below. You should NOT use multiple processes (i.e. you may not use `fork()` or `system()` or similar) and you must not use `select()`.

You will require a thread for each connection and you may require other threads, e.g. one per game or one that keeps track of all games. You will need to use mutual exclusion mechanisms where appropriate, e.g. pthread mutexes or semaphores, in order to prevent multiple threads corrupting shared data structures.

Error handling

Your server should handle communication errors gracefully. If an error occurs when transmitting a message to a client (e.g. because the client has closed in the mean time) then your server should treat this the same as a client disconnecting. If your server is able to start up and listen on the given ports, then your server is not expected to exit unless it encounters problems unrelated to communication (e.g. running out of memory). These circumstances will not be tested.

Your server should not print anything to standard out or standard error apart from the messages given in the startup section.

New games

At the end of a game, the server should close connections to the players and end/cleanup the relevant threads. Your server must not leak memory, threads, file descriptors or other system resources.

Advanced Functionality

The functionality described above is worth 38 out of the 42 functionality marks for the assignment. The following functionality is worth 4 marks.

Your program listens on the second port given on the command line. When a client (e.g. web browser) connects, the server will send a HTTP response containing a representation of the games currently in progress and then disconnects. The incoming HTTP request, if any, can be ignored. The status message should take the following form:

```
HTTP/1.0 200 OK
Content-Type: text/plain

Games in progress: 2

...
...
...

O..
OX.
.X.
```

Each line is to be terminated `\r\n` with dots used to represent spaces. The first line of the body of the response (after the blank line) reports the number of games currently in progress. A game is only considered to be in progress if there are two players connected to the server. After this line, the current state of the board for each game in progress is reported – with a blank line separating the games in progress line and the first board, and a blank line separating each of the boards (but no blank line after the last). The game board representations must be listed in order of creation (i.e. client connection). After sending this text the server will disconnect.

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 (i.e. by 11pm Sunday May 31, 2009) will form part of the assignment specification. If you have any doubts about any parts of the specification, you should ask.

Style

You must follow **version 1.4** of the COMP2303/COMP7306 C programming style guide found at http://www.itee.uq.edu.au/~comp2303/resources/COMP2303_C_Style_Guide_v1.4.pdf.

Hints

You should review the thread and network-related example programs on the course website (<http://www.itee.uq.edu.au/~comp2303/lectures/>) and ensure you understand how these work before commencing. You may freely use any elements of these programs. You may freely use or modify any elements of the code supplied for this assignment also.

You should note that data read from a socket may not arrive in single block – it may require multiple `read()` calls to retrieve the data (even for a single line). You may therefore need to buffer data from several `read()` calls (possibly growing a buffer using `realloc()`) before examining it. You may receive more than one line (request) in a single `read()`. You may find the supplied buffer code helpful. As an alternative, you may use the higher level `fscanf` function.

It is important to note that **you are not required to write a client**. You may use telnet, netcat or nc to make your connections (and a web browser to test the advanced functionality).

Submission

Your assignment must be checked into your Subversion repository in the `comp2303a4` directory (at the top level) – which you will need to create yourself. (The contents of any subdirectories within your repository `comp2303a4` directory will be ignored. You can create subdirectories if you wish, e.g. for test files, but none of those files will be considered when assessing your work – they will be deleted.) **Your submission must include a Makefile which builds your server using the default make on agave.**

You do NOT need to submit standard C header files (e.g. `stdio.h`) – only those header files that you have created (if any). If you use any of the supplied files (e.g. `buffer.h`, `buffer.c`), then you should add them to your repository (and you may edit them if you wish).

The due date for this assignment is 11pm Friday June 5, 2009. The policy on “grace days” and exceptional circumstances (e.g. illness) is as outlined in the course profile. No grace days will be earned for early submissions. Any remaining grace days at the time of last submission are irrelevant. Note that no submissions can be made more than 120 hours past the deadline (i.e. after 11pm Wednesday June 10, 2009) under any circumstances.

Code Compilation

Your submitted code will be compiled on `agave.students.itee.uq.edu.au` as follows:

1. The `comp2303a4` directory within your repository will be checked out to a testing directory on `agave`.
2. All subdirectories, executables and object files in your `comp2303a4` directory (if any) will be removed. (This should leave only `.c` files, `.h` files and a `Makefile`.)
3. For the purposes of counting the warnings generated, each C file in your submission will be compiled with the command:

```
gcc -pthreads -Wall -c filename.c
```

where *filename* is replaced as appropriate. (The object files created with this method will then be removed.)

4. The following command will then be executed:

```
make ncserver
```

If any errors result from the command above (i.e. the `ncserver` executable cannot be created) then you will receive 0 marks for the relevant functionality (see below). 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 breaks any of the rules outlined in the specification above (e.g. uses multiple processes or `select()`) then you will receive 0 marks.

Marks

Marks will be awarded for both functionality and style. A maximum of 50 marks will be awarded.

Functionality (42 marks)

Provided your code compiles (see above), you will earn functionality marks based on the number of features your program correctly implements. 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 server can never accept a connection from a client then it won't be possible to demonstrate full functionality for status clients (web browsers). If your server does not support multiple simultaneous connections, it will not be possible to test whether your server leaks resources or to fully test advanced functionality. The markers will make no alterations or corrections to your code (other than to remove code without academic merit), although minor corrections may be possible (see below). You are advised against making last minute changes to your code without fully testing your complete program before submission. Marking will be automated – it is important that you follow the specification precisely, e.g., correctly spelling all error messages.

- A. Makefile supplied and is able to build the `ncserver` program (2 marks)
- B. Server handles invalid command lines (2 marks)
- C. Server handles not being able to listen on the given client port (2 marks)
- D. Server handles not being able to listen on the given board port (2 marks)
- E. Server listens on the correct client port and is able to accept a single connection and sends the correct message. (2 marks)
- F. Server accepts a second client connection and sends the correct messages to both clients. (3 marks)
- G. Server correctly accepts valid moves from the current player and sends appropriate messages. (3 marks)
- H. Server correctly rejects out of turn messages and sends appropriate replies. (2 marks)
- I. Server correctly rejects invalid messages and sends appropriate replies. (2 marks)
- J. Server correctly rejects invalid moves from the current player. (2 marks)
- K. Server correctly handles game over (draw) due to disconnections. (2 marks)
- L. Server correctly detects when a client has won and ends game correctly. (3 marks)
- M. Server detects a full board (draw) and ends the game appropriately (2 marks)
- N. Server correctly handles disconnections at any time (e.g. before game starts). (2 marks)
- O. Server supports multiple simultaneous games (4 marks)
- P. Server does not leak system resources over-time (or on exit) and does not busy-wait (3 marks)

Advanced Functionality

- Q. Server correctly implements advanced functionality (reporting board status via a HTTP response). (4 marks)

Style (8 marks)

Your style mark will be the minimum of:

$$8 \times 0.9^{(\text{Number of style guide violations} + \text{number of compilation warnings})}$$

and

your functionality mark.

The number of compilation warnings will be the total number of warning lines reported during the compilation process described above. The number of style guide violations refers to the number of violations of version 1.4 of the COMP2303/COMP7306 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)` tool. 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.

Penalties

In cases where submitted code does not compile due to a minor error (e.g. failure to submit a required .h file), the course coordinator may make minor source code corrections and mark the “corrected” code. If this occurs, a penalty between -5 and -15 marks will be applied (at the discretion of the course coordinator) depending on the amount of code correction required. If changes are only required to one line of code, the penalty will be -5.

Late Penalties

Late penalties will apply as outlined in the course profile if no grace days remain at the time of submission.