The University of Queensland
School of Electrical Engineering and Computer Science

CSSE7231 – Semester 2, 2024
Assignment 4 (Version 1.2)

Marks: 85
Weighting: 15%
Due: 3:00pm Friday 25 October, 2024

This specification was created for the use of Sizhe LI (s4876492) only.
Do not share this document. Sharing this document may result in a misconduct penalty.
Note that line numbers vary from those in the CSSE2310 specification.
Specification changes since version 1.0 are shown in red and are summarised at the end of the document.
Changes from v1.1 to v1.2 are shown in blue.

Introduction

The goal of this assignment is to further develop your C programming skills, and to demonstrate your understanding of networking and multithreaded programming. You are to create a server program (`uqchessserver`) that supports multiple simultaneously connected clients playing chess against another client or against a chess engine (Stockfish). The server will maintain pipe connections to and from a single chess engine process and mediate client requests for the engine's services. You should also create a client program (`uqchessclient`) that can send requests to the server and show responses.

Communication between the clients and `uqchessserver` is over TCP using a text based protocol defined later in this specification.

CSSE7231 students are expected to also implement some additional server functionality.

The assignment will also test your ability to code to a particular programming style guide and to use a revision control system appropriately.

It is not necessary that you know how to play chess to undertake this assignment, though knowledge of how pieces can move on a chess board may be useful in testing (i.e. knowing which moves are valid). See `https://en.wikipedia.org/wiki/Chess#Rules` to learn more about the rules of chess if you are interested.

Student Conduct

This section is unchanged from assignments one and three – but you should remind yourself of the referencing requirements. Remember that you can't copy code from websites and if you learn about how to use a library function from a resource other than course-provided material then you must reference it.

This is an individual assignment. You should feel free to discuss **general** aspects of C programming and the assignment specification with fellow students, including on the discussion forum. In general, questions like "How should the program behave if ⟨this happens⟩?" would be safe, if they are seeking clarification on the specification.

You must not actively help (or seek help from) other students or other people with the actual design, structure and/or coding of your assignment solution. It is **cheating to look at another person's assignment code and it is cheating to allow your code to be seen or shared in printed or electronic form by others**. All submitted code will be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion, formal misconduct actions will be initiated against you, and those you cheated with. That's right, if you share your code with a friend, even inadvertently, then **both of you are in trouble**. Do not post your code to a public place such as the course discussion forum or a public code repository. (Code in private posts to the discussion forum is permitted.) You must assume that some students in the course may have very long extensions so do not post your code to any public repository until at least three months after the result release date for the course (or check with the course coordinator if you wish to post it sooner). Do not allow others to access your computer – you must keep your code secure. Never leave your work unattended.

You must follow the following code usage and referencing rules for **all code committed to your SVN repository** (not just the version that you submit):

Code Origin	Usage/Referencing
Code provided by teaching staff this semester Code provided to you in writing **this semester** by CSSE7231 teaching staff (e.g., code hosted on Blackboard, found in `/local/courses/csse2310/resources` on `moss`, posted on the discussion forum by teaching staff, provided in Ed Lessons, or shown in class).	**Permitted** May be used freely without reference. (You must be able to point to the source if queried about it – so you may find it easier to reference the code.)
Code you wrote this semester for this course Code you have personally written this semester for CSSE7231 (e.g. code written for A1 reused in A3) – provided you have not shared or published it.	**Permitted** May be used freely without reference. (This assumes that no reference was required for the original use.)
Unpublished code you wrote earlier Code you have personally written in a previous enrolment in this course or in another UQ course or for other reasons and where that code has not been shared with any other person or published in any way. **Code from `man` pages on `moss`** Code examples found in `man` pages on `moss`. (This does not apply to code from `man` pages found on other systems or websites unless that code is also in the `moss` `man` page.)	**Conditions apply, references required** May be used provided you understand the code AND the source of the code is referenced in a comment adjacent to that code (in the required format – see the style guide). If such code is used without appropriate referencing then this will be considered misconduct.
Code and learning from AI tools Code written by, modified by, debugged by, explained by, obtained from, or based on the output of, an artificial intelligence tool or other code generation tool that you alone personally have interacted with, without the assistance of another person. This includes code you wrote yourself but then modified or debugged because of your interaction with such a tool. It also includes code you wrote where you learned about the concepts or library functions etc. because of your interaction with such a tool. It also includes where comments are written by such a tool – comments are part of your code.	**Conditions apply, references & documentation req'd** May be used provided you understand the code AND the source of the code or learning is referenced in a comment adjacent to that code (in the required format – see the style guide) AND an ASCII text file (named `toolHistory.txt`) is included in your repository and with your submission that describes in detail how the tool was used. (All of your interactions with the tool must be captured.) The file must be committed to the repository at the same time as any code derived from such a tool. If such code is used without appropriate referencing and without inclusion of the `toolHistory.txt` file then this will be considered misconduct. See the detailed AI tool use documentation requirements on Blackboard – this tells you what must be in the `toolHistory.txt` file.
Code copied from sources not mentioned above Code, in any programming language: • copied from any website or forum (including StackOverflow and CSDN); • copied from any public or private repositories; • copied from textbooks, publications, videos, apps; • copied from code provided by teaching staff only in a previous offering of this course (e.g. previous assignment one solution); • written by or partially written by someone else or written with the assistance of someone else (other than a teaching staff member); • written by an AI tool that you did not personally and solely interact with; • written by you and available to other students; or • from any other source besides those mentioned in earlier table rows above.	**Prohibited** May **not** be used. If the source of the code is referenced adjacent to the code then this will be considered code without academic merit (not misconduct) and will be removed from your assignment prior to marking (which may cause compilation to fail and zero marks to be awarded). Copied code without adjacent referencing will be considered misconduct and action will be taken. This prohibition includes code written in other programming languages that has been converted to C.
Code that you have learned from Examples, websites, discussions, videos, code (in any programming language), etc. that you have learned from or that you have taken inspiration from or based any part of your code on but have not copied or just converted from another programming language. This includes learning about the existence of and behaviour of library functions and system calls that are not covered in class.	**Conditions apply, references required** May be used provided you do not directly copy code AND you understand the code AND the source of the code or inspiration or learning is referenced in a comment adjacent to that code (in the required format – see the style guide). If such code is used without appropriate referencing then this will be considered misconduct.

You must not share this assignment specification with any person (other than course staff), organisation, website, etc. Uploading or otherwise providing the assignment specification or part of it to a third party including online tutorial and contract cheating websites is considered misconduct. The university is aware of many of these sites and many cooperate with us in misconduct investigations. You are permitted to post small extracts of this document to the course Ed Discussion forum for the purposes of seeking or providing clarification on this specification.

In short – **Don't risk it!** If you're having trouble, seek help early from a member of the teaching staff. Don't be tempted to copy another student's code or to use an online cheating service. Don't help another CSSE2310/7231 student with their code no matter how desperate they may be and no matter how close your relationship. You should read and understand the statements on student misconduct in the course profile and on the school website: `https://eecs.uq.edu.au/current-students/guidelines-and-policies-students/student-conduct`.

Specification - `uqchessclient`

The `uqchessclient` program provides a command line interface that allows you to interact with the server (`uqchessserver`) as a client – connecting for a session, playing one or more games in that session and then exiting. Because your client will need to listen for incoming messages on both `stdin` and the network connection, it will require two threads. Alternatively, you can implement your client using multiplexed IO as demonstrated in class.

Command Line Arguments

Your `uqchessclient` program is to accept command line arguments as follows:

`./uqchessclient` ***`portno`*** `[--col white|black] [--play machine|person]`

The square brackets (`[]`) indicate optional arguments. The pipe symbol (`|`) indicates a choice. *Italics* indicate a placeholder for a user-supplied arguments. The ***`portno`*** argument must always be the first argument. Option arguments can be in any order after the ***`portno`*** argument.

Some examples of how the program might be run include the following[1]:

```
./uqchessclient 1234
```

```
./uqchessclient 3456 --play machine
```

```
./uqchessclient mysql --col black
```

```
./uqchessclient 8978 --col white --play person
```

The meaning of the arguments is as follows:

- ***`portno`*** – this mandatory argument specifies which localhost port the server is listening on – either numerical or the name of a service.
- `--play` – if specified, this option is followed by either the string `machine` or `person` and indicates that, by default, the user wishes to play against the computer (i.e. the chess engine) or a "human" player respectively. If this argument is not specified, then the default is to play against the computer.
- `--col` – if specified, this option argument is followed by either the string `white` or `black` and indicates the colour (white or black) that the user wishes to be. If this argument is not specified, then the default is play white against a computer opponent and either colour against the first available human opponent. Note that white always plays first in chess.

Prior to doing anything else, your program must check the command line arguments for validity. If the program receives an invalid command line then it must print the (single line) message:

```
Usage: uqchessclient portno [--col white|black] [--play machine|person]
```

to standard error (with a following newline), and exit with an exit status of 2.

Invalid command lines include (but may not be limited to) any of the following:

- No arguments are present (i.e. there is no ***`portno`*** argument)
- An option argument is present but is not followed by a valid option value

[1]This is not an exhaustive list and does not show all possible combinations of arguments. The examples also assume that a `uqchessserver` server is running on the listed ports.

- Any of the option arguments is listed more than once.
- An unexpected argument is present.
- Any argument is the empty string.

Checking whether the *portno* is a valid port or service name is not part of the usage checking (other than checking that the value is not empty). The validity of the port or service name is checked after command line validity as described next.

Client Port Checking

If `uqchessclient` is unable to create a socket and connect to the server on the specified port (or service name) of host `localhost`, it shall print the following message (terminated by a newline) to `stderr` and exit with exit status 7:

```
uqchessclient: unable to make connection to port "N"
```

where *N* should be replaced by the argument given on the command line. (This may be a non-numerical string.) The double quotes must be present.

Client Runtime Behaviour

Assuming that the checks above are passed and your `uqchessclient` can successfully connect to the server, then your `uqchessclient` must print and flush the following message to `stdout` (terminated by a newline):

```
Welcome to UQChessClient - written by s4876492
```

Once this message is printed, the client will act as if a `newgame` command has been entered by the user as described below (i.e. send a "`start`" message to the server with appropriate argument(s)) and must then read lines of input from the user (via `stdin`) and respond to them as described below (e.g. usually by sending a message to the server), and will simultaneously read lines received from the server (via the connected socket) and respond to them as described below. Often the messages from the server will be responses to requests sent by the client but this is not always the case. Communication is asynchronous – either party can send a message at any time – so your client must be implemented to support this. You can do this by using two threads (one reading from `stdin` and one reading from the connected network socket) or by using multiplexed IO as demonstrated in class.

Note that `uqchessclient` does not have to understand chess or know the current position of pieces, etc. The server will keep track of the game state. The client must however keep track of whether a game is currently in progress or not and whose turn it currently is (the user or the opponent) and what colour the user is playing.

Client – Handling Standard Input

The user can enter the following commands on `stdin`. Commands are to be terminated by a single newline. No leading or trailing spaces are to be accepted. If an invalid command is entered, then `uqchessclient` must print the following message to `stderr` (terminated by a newline):

```
Try again - command invalid
```

Valid commands are listed below. In most cases, these will result in a message being sent to the server. See the Communication Protocol section on page 6 for details for how those messages are formatted and which arguments are mandatory and which are optional.

- **`newgame`**
  This command indicates that the user wishes to start a new game and will result in a "`start`" message being sent to the server (with appropriate argument(s)). This command is valid even if a current game is in progress (no matter whose turn it is). If playing against a computer, `uqchessclient` must send either "`black`" or "`white`" as the desired colour with the "`start`" message, not "`either`". (The default colour when playing the computer is "`white`" when no colour is specified on the command line.)
- **`print`**
  This command this indicates that the user wishes to have the current state of the board printed out. If a game is in progress or has finished then the command will result in a "`board`" message being sent to the

server. (The client does not keep track of the board state – it must ask the server for this.) If the first game has not been started yet then an error message must be printed (see below)[2].

- **hint**
  This command indicates that the user wishes to receive a hint from the server's chess engine about the best move to make next. If a game is in progress and it is the user's turn then the command will result in the "**hint best**" message being sent to the server. Otherwise, an error message must be printed (see below).
- **possible**
  This command indicates that the user wishes to know all possible moves that they could make given the current game state. If a game is in progress and it is the user's turn then the command will result in the "**hint all**" message being sent to the server. Otherwise, an error message must be printed (see below).
- **move** ***movestring***
  This command indicates that the user wishes to make the indicated move. The command "**move**" must be followed by a single space and then by 4 or 5 alphanumeric characters before the newline (the ***movestring***) to be considered valid. This associated ***movestring*** argument is a chess move expressed in standard chess long algebraic notation as used by the Universal Chess Interface (UCI), e.g. **e2e4** indicates that the piece on square e2 should be moved to square e4[3]. See `https://en.wikipedia.org/wiki/Algebraic_notation_(chess)` for details on chess notations. If the command is invalid then a message must be printed as described above. If the command is valid and a game is in progress and it is the user's turn then this will result in a "**move**" message being sent to the server. Otherwise, an error message must be printed (see below). (Note that a valid command may not contain a valid chess move. The server will report this if needed – it is not the client's job to know which moves are valid on the game board or not.)
- **resign**
  This command indicates that the user wishes to resign from the current game. If a game is in progress then this will result in a "**resign**" message being sent to the server. Otherwise, an error message must be printed (see below). A user can resign at any time in the game, not just when it is their turn.
- **quit**
  This command indicates that the user wishes to exit the program. The `uqchessclient` program should exit immediately, without sending anything to the server. The same behaviour is expected if the `stdin` of `uqchessclient` is closed even if the user has entered part of a command (e.g. `stdin` will be closed if a terminal user presses Ctrl-D at the start of a line or Ctrl-D Ctrl-D part way through a line). Partially entered lines are to be ignored. Your client must exit with status 0.

If a valid command above is entered (i.e. the line matches the syntax outlined above) but no action can be taken because a game is not in progress or hasn't started yet, then `uqchessclient` must print a message to `stderr` (with a terminating newline):

```
Invalid command - game is not in progress
```

The client should consider a game to be in progress from the time it receives ~~an "ok"~~ a "**started**" response in reply to a "**start**" message through until the time it receives a "**gameover**" message.

If a game is in progress, but a valid command can't be acted on because it is not the user's turn, then `uqchessclient` must print a message to `stderr` (with a terminating newline):

```
Command is not valid - not your turn
```

No messages are to be sent to the server in either of these cases. The user can enter another command if desired.

Client – Handling Messages from Server

The client may receive messages from the server at any time. These messages will have the form outlined in the Communication Protocol section below.

The client must just print all lines of text received from the server to the client's `stdout`, with the exception of the `startboard` and `endboard` lines that surround the depiction of the game board. (The lines between these keywords will be output – including any blank lines.) All messages must be flushed to `stdout` upon receipt[4].

[2]The only situation in which this error should happen is if the client wishes to play a human and no human opponent has yet been assigned.

[3]A *movestring* may be 5 characters for a pawn promotion move, e.g. `g7g8q` indicates a pawn moving to the final rank and being promoted to a queen.

[4]`uqchessclient` can not assume that `stdout` is line buffered.

There is no need for the client to check that messages comply with the protocol – they can just be output as described.

The client must also use the content of the messages (e.g. "`gameover`" messages and others) to keep track of the game state, specifically, whether a game is currently in progress, and if so, whether it is the client's turn to play or not. Specifically, a game will be in progress from when the client receives a "`started`" message from the server until it receives a "`gameover`" message. The player whose turn it is will change every time an "`ok`" or "`moved`" message is received from the server.

Note that a well implemented client should almost never receive some `error` responses from the server – it should never send an invalid command and it should never send commands that assume a game is in progress if it is not. It may receive an `error` message if the chess engine fails or if a move is invalid. Rarely, a game-not-in-progess error will be received in the event that a game-ending race condition occurs[5]. `error` messages from the server should just be printed to the client's `stdout` like other messages from the server.

Other Client Requirements

If your client detects that the connection to the server has been closed then it should print the following message to `stderr` (terminated by a newline):

```
uqchessclient: server connection closed
```

and exit with status 18.

There is no requirement that `uqchessclient` free all memory before exiting.

Your client must not exit due to SIGPIPE.

Communication Protocol

This section describes the incoming and outgoing messages that will be supported by `uqchessserver`. All messages are to be terminated with a single newline (`\n`) and no leading or trailing spaces are permitted. Each message is a command word that may be followed by arguments as described below. If a message accepts arguments then arguments must be separated using a single space character. In the messages below, square brackets (`[]`) indicate optional arguments, and *`italics`* indicate placeholders for specified values.

See the Specification - `uqchessserver` section below for precise details of how the server should behave.

Messages to server

The following incoming messages must be supported by `uqchessserver`.

- `start` *`against colour`*
  - A "`start`" message indicates to the server that the client is starting a new game. The argument *`against`* must be either "`computer`" or "`human`" (without the quotes) and indicates to the server which type of opponent the client will play against. The argument *`colour`* must be one of "`either`", "`black`" or "`white`" (without the quotes) and indicates the colour of pieces that the client will use. The "`either`" option will only matter if the client is playing against a human. If this option is specified with a "`computer`" opponent, then the client will be playing white pieces. See the Specification - `uqchessserver` section for details of how the server assigns colours to human players if "`either`" is specified.
  - The server's response to this message will be a "`started`" message – see below. This may take some time to come back if the client is requesting to play a human and no other player has connected to the server. This "`started`" response will specify which colour that this client will be playing (which will match the colour specified by this client if black or white was specified).
  - If a game is in progress for this client when this message is sent, then the server must treat this as a resignation and send a "`gameover`" response as described below, before sending the "`started`" message.
  - If the client is playing the "computer" and has chosen to be black, then the server (playing white) will also send a "`moved`" message to indicate the computer's first move in the game.

[5] A race condition can occur if the client believes that a game is in progress and sends a message on this basis, but the server has just received a resignation from their human opponent so finishes the game, but receives the client's message before sending out the `gameover` message to the client – followed by an `error 3` message in response to the client's message because the game is no longer in progress.

- `board`
  - This message asks the server to return the current game board or the game board of a game that has just finished (anytime before a "`start`" request is made). If a game is in progress or has just finished, the server will respond with a multi-line response starting with `startboard` and finishing with `endboard` - see below. If the first game has not started yet, the server will respond with an `error` response.
- `hint best|all`
  - Either "`best`" or "`all`" (without the quotes) must be given as an argument to the "`hint`" command.
  - With the "`best`" argument, this message asks the server to return a hint to the client about the next move that the client can make. If a game is in progress and it is this client's turn then the server will respond with a `moves` response (see below), otherwise the server will respond with an `error` response.
  - With the "`all`" argument, this message asks the server to return a list of all the possible moves a client can make. If a game is in progress and it is this client's turn then the server will respond with a `moves` response (see below), otherwise the server will respond with an `error` response.
- `move` *`move-to-make`*
  - This message indicates the move that the client wishes to make. The required argument *`move-to-make`* must be a 4 or 5 character alphanumeric string that indicates the move in long chess algebraic notation (e.g. `e2e4`[6]). The server will respond with either an "`ok`" or "`error`" response depending on whether the move is valid in the current game state or not. A "`move`" command will be considered valid (i.e. will not result in an "`error command`" response from the server) if the *`move-to-make`* argument consists of 4 or 5 letters and or numbers (alphanumeric characters). The server should not check that the letters/numbers form a valid move – this is left to the chess engine.
  - If the move was accepted ("`ok`" response received) then the message might also be followed by a "`check`" or "`gameover`" response from the server if the move resulted in check or checkmate or stalemate. (See the Specification - `uqchessserver` section for details.)
  - If the client is playing the "computer" (and the client's move did not finish the game) then the server will also send a `moved` message to indicate the move that the "computer" is making.
- `resign`
  - This message indicates that the client is resigning from the current game. The server will respond with either a `gameover` or `error` response – see below.

Messages from server to client

The following messages may be sent by `uqchessserver`.

- `started` *`colour`*
  - The server will send this response to a "`start`" message, when the game can be started.
  - The argument *`colour`* confirms the colour that the client will be playing. (This will match the colour the user requested, but is present for those situations where the user requested either colour against a human opponent.)
- `ok`
  - The server will send this response to a "`move`" message if the move is a valid move on the game board.
- `error` *`reason`*
  - The server will send this response to the client when it can not accept or process a message sent by the client or some other error has happened. The required argument (*`reason`*) will be a single word and must be one of:
    - `engine`
      Chess engine failure. If the chess engine fails (terminates for some reason) then the message "`error engine`" must be sent to all clients before the server exits. See the Specification - `uqchessserver` section for more details.

[6] Castling can be represented by showing the movement of the king, e.g. `e1g1`. Pawn promotion can be indicated by appending a letter indicating which piece the pawn is promoted to, e.g. `h7h8q`.

  - `command`
    Invalid command – the server does not understand this command (which could include invalid formatting, e.g. unknown command, additional characters, etc.). A properly implemented client will never send invalid commands but a server must handle them gracefully if received.
  - `game`
    Game not in progress – this command can not be accepted because no game is currently in progress.
  - `turn`
    Not your turn – this command can not be accepted because it is not currently the client's turn to play (but a game currently is in progress).
  - `move`
    Invalid move – the attempted move is not valid for this game

- `startboard`
  `*lines of text*`
  `endboard`
  - This multiline message returns the current state of the game board. The lines between the `startboard` line and the `endboard` line will contain an ASCII art representation of the board. These lines may contain spaces and blank lines.

- `moves` *`move-details [...]`*
  - This message will be sent in response to a valid "`hint`" message and will contain one or more moves (each 4 or 5 character alphanumeric strings), space separated, each in chess long algebraic notation (e.g. `e2e4`). Only one move will be sent in response to a "`hint best`" message. One or more moves will be sent in response to a "`hint all`" message.

- `moved` *`move-made`*
  - This message will tell the client the move made by their opponent (human or computer). The *`move-made`* argument will be a 4 or 5 character alphanumeric string in chess long algebraic notation (e.g. `e7e5`). If playing a human, the *`move-made`* string sent back to a client will be exactly the string provided by their opponent.

- `check`
  - This message will be sent by the server when it detects that the most recent move has resulted in a king being in a "check" position. This message will be sent in response to a `move` message from the client to the server ~~in place of~~ after an `ok` message if the client's move has resulted in the opponent's king being in check (but not checkmate). It will immediately follow a `moved` message from the server to the client if the opponent's move has resulted in the client's king being in check.

- `gameover` *`how [winner]`*
  - This message tells the client that the current game is now over. The required argument *`how`* is one of "`checkmate`" or "`stalemate`" or "`resignation`" (without the quotes). If the *`how`* argument is "`checkmate`" or "`resignation`" then the *`winner`* argument must also be supplied and will be either "`white`" or "`black`" (without the quotes) – indicating which colour won the game. This message will be sent ~~in place of~~ after an "`ok`" message if the client's move has resulted in checkmate or stalemate. This message may arrive to a client at any time if their human opponent resigns (the "computer" never resigns). If two clients are playing (i.e. two humans) then the same message is sent to both clients.

Specification - uqchessserver

`uqchessserver` is a networked, multithreaded server that manages chess games for connected clients. It will start and communicate with a single chess engine process (Stockfish) and use that engine to support all of the games that are in progress. Games can be between one client and another client or between a client and the "computer" (i.e. the chess engine).

All communication between clients and the server is via a ASCII text line-based protocol over TCP. See the Communication Protocol section on 6 for details.

Command Line Arguments

Your `uqchessserver` program is to accept command line arguments as follows:

```
./uqchessserver [--max num] [--port portnum]
```

The square brackets (`[]`) indicate optional arguments. *Italics* indicate a placeholder for a user-supplied argument. None, either or both of the options can be specified (at most once each). Option arguments can be in either order if both are specified.

Some examples of how the program might be run include the following[7]:

```
./uqchessserver

./uqchessserver --port 2310

./uqchessserver --max 5

./uqchessserver --port 1234 --max 10
```

The meaning of the arguments is as follows:

- `--max` – if specified, this option argument is followed by a non-negative integer less than or equal to 800 specifying the maximum number of simultaneous client connections to be permitted. If this is zero or this argument pair is absent, then there is no limit to how many clients may connect (other than operating system limits which we will not test).
- `--port` – if specified, this option argument is followed by a string which specifies which localhost port the server is to listen on. This can be either numerical or the name of a service. If this is zero or this argument pair is absent, then `uqchessserver` is to use an ephemeral port.

Important: Even if you do not implement the connection limiting functionality, your program must correctly handle command lines which include this argument and value (after which it can ignore any provided values – you will simply not receive any marks for that feature).

Prior to doing anything else, your program must check the command line arguments for validity. If the program receives an invalid command line then it must print the (single line) message:

```
./uqchessserver [--max num] [--port portnum]
```

to standard error (with a following newline), and exit with an exit status of 15.

Invalid command lines include (but may not be limited to) any of the following:

- The `--max` option argument is given but it is not followed by a non-negative integer less than or equal to 800. A leading `+` sign is permitted (optional). Numbers with leading zeroes will not be tested – i.e. may be accepted or rejected.
- Any of the option arguments is listed more than once.
- The `--port` option argument is given but it is not followed by a non-empty string argument
- An unexpected argument is present.
- Any argument is the empty string.

Checking whether the *portnum* argument is a valid port or service name is not part of the usage checking (other than checking that the value is not empty). The validity of this value is checked after command line validity as described below.

Port Checking

If `uqchessserver` is unable to listen on the given port (or service name) of `localhost`, it shall output the following message (terminated by a newline) to `stderr` and exit with status 6:

```
uqchessserver: unable to listen on port "N"
```

where *N* should be replaced by the argument given on the command line. (This may be a non-numerical string.) The double quotes must be present. Being "unable to listen on a given port" includes the cases where the socket can't be created, the port string is invalid, the socket can't be bound to the address, and the socket can't be listened on. Note that we will not test the situation where `uqchessserver` is unable to listen on an ephemeral port.

[7]This is not an exhaustive list and does not show all possible combinations of arguments.

Starting the Chess Engine

If the checks above succeed, then `uqchessserver` is to start the chess engine (Stockfish). A single chess engine is used by `uqchessserver` no matter how many clients are being serviced. Stockfish (which is run using the `stockfish` executable on moss) is a Universal Chess Interface (UCI) compliant chess engine[8] which means that it listens for text commands on its `stdin` and provides text responses on its `stdout`. Details of UCI can be found at `https://backscattering.de/chess/uci/`, however, the necessary information for this assignment can be found in this specification in the UCI – Universal Chess Interface section below.

To start the chess engine, your `uqchessserver` program must:

- create two pipes – one will be for `uqchessserver` to write to the Stockfish process's `stdin` and one will be for the server to read from the Stockfish process's `stdout`;
- fork to create a child process;
- redirect/close file descriptors as appropriate;
- in the child process, run the program `stockfish` – which is to be found in the user's path (no command line arguments are to be provided);
- send the "`isready`" message to `stockfish` and wait for the "`readyok`" response (another line will be sent before this also – this can be ignored); and
- after the "`readyok`" response has been received, send the "`uci`" message to `stockfish` and wait for the "`uciok`" response (other lines will be sent before this also – these can be ignored).

Note that `stockfish`'s `stderr` should just be inherited from that of `uqchessserver`.

If `uqchessserver` is unable to start the Stockfish process, e.g. SIGPIPE or EPIPE or other error is detected when writing to `stockfish` or EOF is detected when reading from `stockfish` then `uqchessserver` must print the following message to `stderr` (with a terminating newline):

```
uqchessserver: unable to start communication with chess engine
```

and exit with status 12. No client connections are ever to be accepted.

Your server can assume that the chess engine is well behaved, i.e. that if you are expecting a response in accordance with the UCI protocol then that response will be forthcoming, i.e. reads from `stockfish` looking for that response won't block forever.

If, after starting the chess engine (the first `readyok` and `uciok` responses are received) your `uqchessserver` ever detects that the Stockfish process has exited (e.g. by detecting an error when writing to `stockfish`, or detecting EOF when reading the pipe from `stockfish`), then the server must reap the child process, send an error message to all connected clients (if any) (see the Communication Protocol) section), and print the following message to `stderr` (with a terminating newline):

```
uqchessserver: chess engine terminated
```

and then exit with exit status 20.

This is the only circumstance under which the server is expected to exit after it has completed the start-up checks (on the command line arguments and whether it can listen and whether it can start the chess engine). It is not necessary for the server to free all memory when it exits (under this or any circumstance).

Runtime Behaviour

Once the port is opened for listening and the chess engine started, `uqchessserver` shall print to `stderr` the port number that it is listening on (not the service name) followed by a single newline character and then flush the output. **In the case of ephemeral ports, the actual port number obtained shall be printed, not zero.**

Upon receiving an incoming client connection on the port, `uqchessserver` shall spawn a new thread to handle that client (see below for client thread handling).

If the `--max` argument is specified on the command line (along with a value argument) and connection limiting behaviour is implemented, then `uqchessserver` must keep track of how many active client connections exist, and must not let that number exceed the *`num`* parameter. See below for more details on how this limit is to be implemented.

The `uqchessserver` program should not terminate under normal circumstances (unless the chess engine terminates – see above), nor should it block or otherwise attempt to handle `SIGINT`.

Note that your `uqchessserver` must be able to deal with any clients, not just the client programs specified for the assignment. Testing with `netcat` is highly recommended.

[8]Stockfish has some additional functionality beyond that required in UCI. We take advantage of that in this assignment, e.g. the presence of the "`d`" command.

Client handling threads

A single client handler thread is spawned for each incoming connection. This client thread must then wait for a message from the client, and act on it – which in many cases will involve sending a message to the chess engine, awaiting a response and then sending a response to the client. The exact format of client requests is described in the Communication Protocol section on page 6.

Due to the simultaneous nature of multiple client connections, your `uqchessserver` will need to ensure mutual exclusion around accesses to any shared data structure(s) and resources (e.g. the chess engine) so as to ensure that data/messages do not get corrupted.

Once a client disconnects or there is a communication error on the socket (e.g. a `read()` or equivalent from the client returns `EOF`, or a `write()` or equivalent fails) then the client handler thread is to close the connection, clean up as appropriate (e.g. send a "`gameover`" message to a human opponent – a disconnection is to be treated as a resignation) and terminate. Other client threads and the `uqchessserver` program itself must continue uninterrupted. `uqchessserver` must not exit in response to a `SIGPIPE`.

Managing Chess Games

Your client thread must handle messages from the client as described in the Communication Protocol section. Additional information on server behaviour in response to each message type is provided below. Note that each client message will result in the specified behaviour, usually resulting in a response to the client. Once the specified behaviour is undertaken (and a response sent to the client if appropriate), then the client thread must await the next message from the client. Note that if messages must be sent to the chess engine, they will need to be preceded by other messages that specify the game state - see the UCI – Universal Chess Interface section below (page 14).

- `start` ***`against colour`***
  When the server receives a "`start`" message from a client, it must do the following:
  - Conclude any game in progress as if a resignation message had been received.
  - If the client wishes to play against a computer opponent, start the game and send back an appropriate "`started`" response. If the computer is playing white, then the server must also generate a move (see below) and send back a "`moved`" response.
  - If the client wishes to play against a human opponent, then check to see if another client is awaiting a human opponent and their colour choices are compatible (i.e. white with black or a specified colour preference with no colour preference or both have no colour preference). If neither has a colour preference then the first client waiting will be assigned the white pieces (i.e. the client that sent the "`start`" message last will be assigned black). If multiple clients are waiting with a particular colour preference then the one which was the first to connect to the server must be chosen. Appropriate "`started`" messages must be sent to both clients[9]. If there is no colour-compatible human opponent waiting to play, then this will be noted (so that if a colour-compatible client does connect then a game can be started – by the other client's thread). The client thread does not send a response to the client in this case, but will wait for further messages from the client.
- `board`
  When the server receives a "`board`" message from the client, it must do the following:
  - If this client has never started a game (no "`start`" request has been received) or a "`start`" request to play a human has been received and no colour compatible opponent is available, then the client must send a game-not-in-progress "`error`" response.
  - If a game is in progress (even if no moves have been made yet) then the server must issue a "`d`" UCI request to the engine and return the current game board to the client between `startboard` and `endboard` messages (see details of the "`d`" message in the UCI – Universal Chess Interface section).
  - If a game is not in progress but one has been played (i.e. it is over) and no new "`start`" request has been received, then the server must must issue a "`d`" UCI request to the engine and return the last state of the game board (i.e. as it was when the last game finished) to the client between `startboard` and `endboard` messages.
- `hint [best|all]`
  When the server receives a "`hint`" message from the client, it must do the following:
  - If a game is not in progress or it is not the client's turn then issue an appropriate "`error`" response.

[9]Note that this one client thread will be sending messages to two different connected clients.

  - If it is a "`hint best`" message, then the client thread must issue the UCI command "`go movetime 500 depth 15`" to the chess engine and await a "`bestmove`" line in the response. (This request will limit the move search to 500 milliseconds or a depth of 15 moves, whichever happens sooner.). The client thread must then send a `moves` response to the client with the move extracted from the "`bestmove`" response from the chess engine. Other lines of text emitted by the chess engine (`info` lines) are to be ignored.
  - If it is a "`hint all`" message, then the client thread must issue the UCI command "`go perft 1`" to the chess engine and collect the moves in the response (over multiple lines). The client thread must then send a "`moves`" response to the client with the moves extracted from the response. Other lines of text emitted by the chess engine ("`info`" lines, blank lines and the count of "Nodes searched") are to be ignored.

- `move` *`move-to-make`*
  When the server receives a "`move`" message from the client, it must do the following:
  - If a game is not in progress or it is not the client's turn then issue an appropriate "`error`" response.
  - If it is the client's turn, then the client thread must issue a "`position`" UCI command with the current board state (see below) and this specified move. Following this it must issue a "`d`" command to get the board state to determine the outcome of the move.
    - If the board state has not changed (the "`Fen`" line in the response is the same as before the attempted move) then the move is not valid and the client thread must send the appropriate "`error`" response to the client.
    - If the board state has changed (the move was accepted) then an "`ok`" response must be sent to the client. If the opponent is human then the "`moved`" message must be sent to the client of the opponent.
    - If the move was accepted, the client thread must also send the "`go perft 1`" command to the chess engine (to determine the number of valid moves for the opponent – from the "Nodes searched" line in the response) and use the output of this and the output of the "`d`" command to determine the consequences of the move as outlined next.
    - If the move was accepted and the "`Checkers`" line in the "`d`" response is not empty then the player is in check or checkmate. If there are zero possible moves for the opponent then this is checkmate and the game is over. The client thread must send a "`gameover`" message to the client (and the opponent's client if the opponent is human). If there are possible moves then the move has resulted in a check position and the client thread must send a "`check`" message to the client (and the opponent's client if the opponent is human).
    - If the move was accepted and the "`Checkers`" line in the "`d`" response was empty and there are no valid moves for the opponent then this is a stalemate (the game is over, but no player wins). The client thread must send an appropriate "`gameover`" message to the client (and the opponent's client if the opponent is human).
  - If the opponent is the "`computer`" and the game was not finished by the provided move, then the server must do the following:
    - Generate a move that the computer will make in response. It does this is the same way that hints are generated (see above), by sending the UCI command "`go movetime 500 depth 15`" to the chess engine and waiting for the "`bestmove`" line (the last line) in the response. The best move is then sent to the client with a "`moved`" message.
    - Check whether the computer's move resulted in check, checkmate or stalemate – using the same approach described above. If so, this will result in the server sending a "`check`" or "`gameover`" message to the client following the "`moved`" message.

- `resign`
  When the server receives a "`resign`" message from the client, it must do the following:
  - If a game is not in progress then issue an appropriate "`error`" response.
  - If a game is in progress, then this indicates that this client has resigned (i.e. lost the game). The client thread must send an appropriate "`gameover`" message back to the client. If the opponent is human, then an appropriate "`gameover`" message must be sent the opponent's client also.

Game State

For each game, `uqchessserver` must maintain some game state:

- whether the game is in progress or finished;
- who is playing white (i.e. which client, or the computer);
- who is playing black (i.e. which client, or the computer – noting that at least one of the colours must be played by a client); and
- the current or last game board state – maintained as a FEN string (provided by the chess engine – see below). This string includes information about whose turn it is to play next.

Note that recording which client is playing which colour does not involve recording which thread is managing that client – more likely it will involve recording the file descriptor or `FILE*` handle(s) associated with the client so that messages can be sent to the client.

Note that the state of completed games must be maintained until the client(s) who played the game has/have started another game or disconnected from the server. This is so that an appropriate response can be given to a "`board`" message, i.e. what the board looked like at the end of the game.

Client connection limiting (CSSE7231 students only)

If the `--max` *`num`* feature is implemented and a non-zero command line argument is provided, then `uqchessserver` must not permit more than that number of simultaneous client connections to the server. If a client beyond that limit attempts to connect, `uqchessserver` shall block, indefinitely if required, until another client leaves and this new client's connection request can be `accept()`ed. Clients in this waiting state are not to be counted in statistics reporting – they are only counted once they have properly connected. Existing clients must continue to be served.

Statistics reporting (CSSE7231 students only)

Upon receiving `SIGHUP`, `uqchessserver` is to output (and flush) to `stderr` statistics reflecting the program's operation to-date, specifically

- Total number of clients connected (at this instant)
- The total number of clients that have connected and disconnected since program start
- Number of games in progress (at this instant)
- The total number of games that have been started and completed up to now

The required statistics format is illustrated below. Each of the four lines is terminated by a single newline. You can assume that all numbers will fit in a 32-bit unsigned integer, i.e. you do not have to consider numbers larger than 4,294,967,295.

Example 1: `uqchessserver` SIGHUP `stderr` output sample

```
Number of connected clients: 4
Number of completed clients: 20
Games in progress: 3
Games completed: 35
```

Note that to accurately gather these statistics and avoid race conditions, you will need some sort of mutual exclusion protecting the variables holding these statistics.

Global variables are NOT to be used to implement signal handling (or for any other purposes in this assignment). See the Hints section below for how you can implement signal handling.

Other Requirements

Other than the listening port number, `SIGHUP`-initiated statistics output, and error messages, `uqchessserver` is not to emit any output to `stdout` or `stderr`.

Your server must not leak memory. Your server is never expected to exit after startup (unless the chess engine terminates for some reason) so your program never has to free all memory but it's memory consumption should not grow significantly over time. It is expected that the memory consumption of your program will be the same after a client disconnects as it was just before the client connected (assuming no other clients connected and remained connected in that time).

Your server must not busy wait. If a thread has nothing to do then it must be blocking, e.g. in `accept()` or a reading call (such as `read()`, `fread()`, `fgets()`, or `getline()`), and not sleeping or busy waiting.

We will not test for unexpected system call or library failures in an otherwise correctly-implemented program (e.g. if `fork()`, `malloc()`, `pipe()`, `pthread_create()`, etc. fails due to insufficient available resources). Your program can behave however it likes in these cases, including crashing.

UCI – Universal Chess Interface

Full details about how Stockfish implements the Universal Chess Interface (UCI) can be found at `https://github.com/official-stockfish/Stockfish/wiki/UCI-&-Commands`. Descriptions of the commands you need to send to Stockfish, and the expected responses, are found below. In the examples below, green text indicates commands sent to Stockfish via its `stdin`. Black text indicates response text from Stockfish via its `stdout`. Commands and responses are terminated by a single newline.

- `isready`
  This command must be sent prior to any other communication with the chess engine to check whether the engine is ready or not. No other commands should be sent until the engine responds with a "`readyok`"

Example 2: Example interaction with Stockfish showing the `isready` command and response

```
Stockfish dev-20240917-60351b9d by the Stockfish developers (see AUTHORS file)
isready
readyok
```

- `uci`
  This command tells the engine to use UCI (Universal Chess Interface). The engine will return a number of `id` and `option` lines identifying the engine and which options it supports, concluded by a `uciok` message. None of the commands below should be sent until after this `uciok` response is received.

Example 3: Example interaction with Stockfish showing the `uci` command and response

```
uci
id name Stockfish dev-20240917-60351b9d
id author the Stockfish developers (see AUTHORS file)

option name Debug Log File type string default <empty>
option name NumaPolicy type string default auto
option name Threads type spin default 1 min 1 max 1024
option name Hash type spin default 16 min 1 max 33554432
option name Clear Hash type button
option name Ponder type check default false
option name MultiPV type spin default 1 min 1 max 256
option name Skill Level type spin default 20 min 0 max 20
option name Move Overhead type spin default 10 min 0 max 5000
option name nodestime type spin default 0 min 0 max 10000
option name UCI_Chess960 type check default false
option name UCI_LimitStrength type check default false
option name UCI_Elo type spin default 1320 min 1320 max 3190
option name UCI_ShowWDL type check default false
option name SyzygyPath type string default <empty>
option name SyzygyProbeDepth type spin default 1 min 1 max 100
option name Syzygy50MoveRule type check default true
option name SyzygyProbeLimit type spin default 7 min 0 max 7
option name EvalFile type string default nn-1111cefa1111.nnue
option name EvalFileSmall type string default nn-37f18f62d772.nnue
uciok
```

- `ucinewgame`
  This command tells the engine that the interactions that follow are associated with a different game. You can think of this as a context switch – the engine will now be doing calculations for a different game – not necessarily from the starting position. This command must always be followed by an `isready` command

and no further commands should be sent until a `readyok` response is received. (The engine will not send a response to the `ucinewgame` command.) A `ucinewgame` command and subsequent `isready` command must only be sent just before it is necessary to send some other UCI command – usually a `position` command (followed by a `go` or `d` command). A `ucinewgame` command (and subsequent `isready` command) can be sent prior to every `position` command if desired – even if the subsequent `position` command is for the same game. Alternatively, you can just send them when a context (i.e. game) switch is necessary.

Example 4: Example interaction with Stockfish showing the `ucinewgame` command

```
ucinewgame
isready
readyok
```

- **`position startpos`**
  `position startpos moves` ***`movestring`***
  `position fen` ***`fenstring`***
  `position fen` ***`fenstring`*** **`moves`** ***`movestring`***
  The `position` command specifies a given game state and optionally a move to make in that game state. The game state can be `startpos` which indicates a chess board at the start of the game when no moves have been made (white to play first). Alternatively, the game state can be specified as `fen` *`fenstring`* which specifies the game state using a string in Forsyth Edwards Notation (FEN) – with *`fenstring`* made up of six space-separated fields. If a move is specified (with the `moves` keyword), the following *`movestring`* is a 4 or 5 character string in long algebraic notation, e.g. `e2e4`. The engine will not send a response to the position command. Note that the second of six fields in the FEN string (either the letter '`w`' or '`b`') indicates the colour of the next player to move. A `position` command must be immediately followed by a `go` or `depth` command – i.e. don't send a `position` command if you don't have something else to ask the server.

Example 5: Examples of the `position` command

```
position startpos
position startpos moves e2e4
position fen rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
position fen rnbqkbnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR b KQkq - 0 1 moves e7e5
```

- **`go perft 1`**
  `go movetime` ***`time`*** **`depth`** ***`depth`***
  The `go` command specifies that the engine should evaluate the current position (as set with the `position` command) to calculate a move or possible moves. The first variant, "`go perft 1`" will list all of the possible moves that can be made from the current position and how many different moves are possible (listed against the heading "`Nodes searched:`"). The second variant ("`go movetime` *`time`* `depth` *`depth`*") searches for the best possible move from the current position. The *`time`* value is the maximum search time (in milliseconds). The *`depth`* value is the maximum depth of the search (how many moves to look ahead). The line of interest in the output is the "`bestmove`" line. The "`ponder`" element (if present) is the move that the engine thinks is most likely in response – this value is ignored. A `go` command must always be preceded by a `position` command (possibly with an intervening `d` command between the `position` and `go` commands).

Example 6: Examples of the `go` command. The 20 possible moves are listed on lines 7 to 26. The best move to make is listed on line ~~50~~ 51.

```
1 position startpos
2 go perft 1
3 info string Available processors: 0-11
4 info string Using 1 thread
5 info string NNUE evaluation using nn-1111cefa1111.nnue (133MiB, (22528, 3072, 15, 32,
     1))
6 info string NNUE evaluation using nn-37f18f62d772.nnue (6MiB, (22528, 128, 15, 32, 1))
7 a2a3: 1
8 b2b3: 1
9 c2c3: 1
```

```
d2d3: 1
e2e3: 1
f2f3: 1
g2g3: 1
h2h3: 1
a2a4: 1
b2b4: 1
c2c4: 1
d2d4: 1
e2e4: 1
f2f4: 1
g2g4: 1
h2h4: 1
b1a3: 1
b1c3: 1
g1f3: 1
g1h3: 1

Nodes searched: 20

position startpos
go movetime 500 depth 15
info string Available processors: 0-11
info string Using 1 thread
info string NNUE evaluation using nn-1111cefa1111.nnue (133MiB, (22528, 3072, 15, 32, 1))
info string NNUE evaluation using nn-37f18f62d772.nnue (6MiB, (22528, 128, 15, 32, 1))
info depth 1 seldepth 3 multipv 1 score cp 33 nodes 23 nps 23000 hashfull 0 tbhits 0 time 1 pv e2e4
info depth 2 seldepth 8 multipv 1 score cp 33 nodes 62 nps 62000 hashfull 0 tbhits 0 time 1 pv e2e4 e7e5 g1f3 d7d5 e4d5 e5e4
info depth 3 seldepth 10 multipv 1 score cp 33 nodes 95 nps 47500 hashfull 0 tbhits 0 time 2 pv e2e4 e7e5 g1f3 d7d5 e4d5 e5e4 f3e5 d8d5
info depth 4 seldepth 12 multipv 1 score cp 33 nodes 138 nps 69000 hashfull 0 tbhits 0 time 2 pv e2e4 e7e5 g1f3 d7d5 e4d5 e5e4 f3e5 d8d5 e5c4 f8e7
info depth 5 seldepth 13 multipv 1 score cp 33 nodes 182 nps 91000 hashfull 0 tbhits 0 time 2 pv e2e4 e7e5 g1f3 d7d5 e4d5 e5e4 f3e5 d8d5 e5c4 f8e7 b1c3 d5c6
info depth 6 seldepth 13 multipv 1 score cp 33 nodes 236 nps 118000 hashfull 0 tbhits 0 time 2 pv e2e4 e7e5 g1f3 d7d5 e4d5 e5e4 f3e5 d8d5 e5c4 f8e7 b1c3 d5c6
info depth 7 seldepth 13 multipv 1 score cp 33 nodes 292 nps 146000 hashfull 0 tbhits 0 time 2 pv e2e4 e7e5 g1f3 d7d5 e4d5 e5e4 f3e5 d8d5 e5c4 f8e7 b1c3 d5c6
info depth 8 seldepth 13 multipv 1 score cp 33 nodes 374 nps 187000 hashfull 0 tbhits 0 time 2 pv e2e4 e7e5 g1f3 d7d5 e4d5 e5e4 f3e5 d8d5 e5c4 f8e7 b1c3 d5c6
info depth 9 seldepth 13 multipv 1 score cp 33 nodes 608 nps 304000 hashfull 0 tbhits 0 time 2 pv e2e4 e7e5 g1f3 d7d5 e4d5 e5e4 f3e5 d8d5 e5c4 f8e7 b1c3 d5c6
info depth 10 seldepth 17 multipv 1 score cp 32 nodes 2040 nps 408000 hashfull 0 tbhits 0 time 5 pv e2e4 e7e5 g1f3 g8f6 d2d4 d7d5 e4d5 e5d4 f1b5 c7c6 d5c6 b7c6 b5e2
info depth 11 seldepth 17 multipv 1 score cp 31 nodes 2846 nps 569200 hashfull 0 tbhits 0 time 5 pv e2e4 e7e5 g1f3 g8f6 d2d4 f6e4 f3e5 d7d5 f1d3 f8d6 e1g1 e8g8 b1c3 e4c3 b2c3 d8h4
info depth 12 seldepth 17 multipv 1 score cp 30 nodes 3896 nps 556571 hashfull 0 tbhits 0 time 7 pv e2e4 e7e5 g1f3 b8c6 f1b5 g8f6 d2d4 f6e4 e1g1
info depth 13 seldepth 21 multipv 1 score cp 35 nodes 12002 nps 631684 hashfull 4 tbhits 0 time 19 pv e2e4 e7e5 g1f3 g8f6 d2d4 f6e4 f3e5 d7d5 d1f3 c8e6 f1d3 f8d6 d3e4 d5e4 f3e4
info depth 14 seldepth 18 multipv 1 score cp 33 nodes 15415 nps 670217 hashfull 5 tbhits 0 time 23 pv e2e4 e7e5 g1f3 g8f6 d2d4 f6e4 f3e5 d7d5 f1d3 b8c6 e1g1 f8d6 f2f4 e8g8 b1c3
info depth 15 seldepth 21 multipv 1 score cp 35 nodes 28782 nps 702000 hashfull 9
    tbhits 0 time 41 pv e2e4 e7e5 g1f3 b8c6 d2d4 e5d4 f3d4 g8f6 d4c6 b7c6 f1d3 d7d5
    e4e5 f6d7 e1g1 d7c5
bestmove e2e4 ponder e7e5
```

- `d`
  The `d` command asks the engine to display details of the current game position and state. This includes an ASCII art representation of the game board (white pieces are shown with upper case letters, black pieces are shown with lower case letters[10]). The ASCII art lines, including the blank line immediately before the art and the blank line immediately after the art, are the lines to be sent back to clients when requested (with "`startboard`" and "`endboard`" delimiters). The FEN string (in the line with the "`Fen:`" prefix) will be line saved as part of the game state – this contains full details about the game. The "`Checkers:`" line lists the positions of all the pieces that have the current player's king in check. A `d` command must always be immediately preceded by a `position` command.

  Example 7: Examples of the `d` command. Lines 3 to 22 (inclusive) or lines 28 to 47 (inclusive) would be those sent back in response to a "`board`" message.

```
1  position startpos
2  d
3
4   +---+---+---+---+---+---+---+---+
5   | r | n | b | q | k | b | n | r | 8
6   +---+---+---+---+---+---+---+---+
7   | p | p | p | p | p | p | p | p | 7
8   +---+---+---+---+---+---+---+---+
9   |   |   |   |   |   |   |   |   | 6
10  +---+---+---+---+---+---+---+---+
11  |   |   |   |   |   |   |   |   | 5
12  +---+---+---+---+---+---+---+---+
13  |   |   |   |   |   |   |   |   | 4
14  +---+---+---+---+---+---+---+---+
15  |   |   |   |   |   |   |   |   | 3
16  +---+---+---+---+---+---+---+---+
17  | P | P | P | P | P | P | P | P | 2
18  +---+---+---+---+---+---+---+---+
19  | R | N | B | Q | K | B | N | R | 1
20  +---+---+---+---+---+---+---+---+
21    a   b   c   d   e   f   g   h
22
23 Fen: rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
24 Key: 8F8F01D4562F59FB
25 Checkers:
26 position fen 8/k7/8/2QK4/8/8/8/8 b - - 4 20
27 d
28
29  +---+---+---+---+---+---+---+---+
30  |   |   |   |   |   |   |   |   | 8
31  +---+---+---+---+---+---+---+---+
32  | k |   |   |   |   |   |   |   | 7
33  +---+---+---+---+---+---+---+---+
34  |   |   |   |   |   |   |   |   | 6
35  +---+---+---+---+---+---+---+---+
36  |   |   | Q | K |   |   |   |   | 5
37  +---+---+---+---+---+---+---+---+
38  |   |   |   |   |   |   |   |   | 4
39  +---+---+---+---+---+---+---+---+
40  |   |   |   |   |   |   |   |   | 3
```

[10] The piece notation used is: k/K = king, q/Q = queen, b/B = bishop, n/N = knight, r/R = rook, p/P = pawn.

```
 +---+---+---+---+---+---+---+---+
 |   |   |   |   |   |   |   |   | 2
 +---+---+---+---+---+---+---+---+
 |   |   |   |   |   |   |   |   | 1
 +---+---+---+---+---+---+---+---+
   a   b   c   d   e   f   g   h

Fen: 8/k7/8/2QK4/8/8/8/8 b - - 4 20
Key: B7AE661E3C37B0A7
Checkers: c5
```

Provided Libraries

libcsse2310a4

A `split_by_char()` function is available to break a string up into multiple parts, e.g. based on spaces. Several functions that operate on Stockfish's output are also available to simplify the parsing of this output. The functions available in this library are:

```
char** split_by_char(char* str, char split, unsigned int maxFields);
```

```
StockfishGameState* read_stockfish_d_output(FILE* stream);
```

```
void free_stockfish_game_state(StockfishGameState* state);
```

```
ChessMoves* read_stockfish_go_perft_1_output(FILE* stream);
```

```
ChessMoves* read_stockfish_bestmove_output(FILE* stream);
```

```
void free_chess_moves(ChessMoves* moves);
```

```
char next_player_from_fen_string(const char* fen);
```

These functions and the associated types are declared in `/local/courses/csse2310/include/csse2310a4.h` on moss and their behaviour and required compiler flags are described in man pages on moss.

Testing

You are responsible for ensuring that your program operates according to the specification. You are encouraged to test your program on a variety of scenarios. A variety of programs will be provided to help you in testing:

- Two demonstration programs (called `demo-uqchessserver` and `demo-uqchessclient`) that implement the correct behaviour will be made available on `moss`. You can test your client with the demo server and vice-versa. You can also use `demo-uqchessserver` and `demo-uqchessclient` to check the expected behaviour of the programs if some part of this specification is unclear.
- Remember that you can use netcat (`nc`) to do testing also – you can use netcat as a client to communicate with your server, or as a server that your client can communicate with. This will allow you to simulate and capture requests and responses between the server and client.
- A test script will be provided on `moss` that will test your programs against a subset of the functionality requirements – approximately 50% of the available functionality marks. The script will be made available about 7 to 10 days before the assignment deadline and can be used to give you some confidence that you're on the right track. The "public tests" in this test script will not test all functionality and you should be sure to conduct your own tests based on this specification. The "public tests" will be used in marking, along with a set of "private tests" that you will not see.

- The Gradescope submission site will also be made available about 7 to 10 days prior to the assignment deadline. Gradescope will run the test suite immediately after you submit. When this is complete[11] you will be able to see the results of the "public tests". You should check these test results to make sure your program is working as expected. Behaviour differences between `moss` and Gradescope may be due to memory initialisation assumptions in your code, so you should allow enough time to check (and possibly fix) any issues after submission.

Style

Your program must follow version 3 of the CSSE2310/CSSE7231 C programming style guide available on the course Blackboard site. Your submission must also comply with the *Documentation required for the use of AI tools* if applicable.

Hints

1. The multithreaded network server example from the lectures can form the basis of `uqchessserver`.
2. Review the lectures and sample code related to network clients, threads and synchronisation (semaphores), and multi-threaded network servers. This assignment builds on all of these concepts. You should also review the content on pipes, redirection and forking if this is not fresh in your memory.
3. Use the provided library functions (see above).
4. Remember to `fflush()` output that you `printf()` or `fprintf()` or otherwise output via `FILE*` streams. Output to network sockets is not newline buffered. (Flushing is not necessary when using `write()` to output directly to a file descriptor.)
5. You will need to use appropriate mutual exclusion (e.g. implemented with semaphores) in your server to avoid race conditions when accessing common data structures and shared resources (e.g. the chess engine).
6. (Read this in conjunction with the next hint.) Your server will probably need to maintain several types of data structures. For example, each client thread might have an associated `Client` struct that records details of how to communicate with the client, details of the game in progress (which might be a pointer to a `Game` struct possibly shared between two clients, which points back to the client(s) involved so that one client thread can send messages to another client). The server will also have to maintain some array or list data structures, e.g. a list of all clients and a list of clients who are awaiting a human opponent with a compatible colour choice.
7. The implementation of the server and associated data structures is much simpler when clients can only play the computer and not other clients (humans) – all the game information is specific to one client and not shared between multiple clients and a client thread will only ever communicate with one client. You also don't have to manage queues of clients waiting for a colour-compatible opponent. It is possible to obtain 60% of the marks for the server with an implementation that only allows clients to play the computer.
8. Consider a dedicated signal handling thread for `SIGHUP`. `pthread_sigmask()` can be used to mask signal delivery to threads, and `sigwait()` can be used in a thread to block until a signal is received. You will need to do some research and experimentation to get this working. Be sure to properly reference any code samples or inspiration you use.
9. The `isalnum()` function can be used to check if a character is alphanumeric (i.e. is either a letter or a number).

[11] Gradescope marking may take only a few minutes or more than 30 minutes depending on the functionality and efficiency of your code.

Possible Approach

1. Try implementing `uqchessclient` first. (The programs are independent so this is not a requirement, but when you test it with `demo-uqchessserver` it may give you a better understanding of how `uqchessserver` works.)
2. For `uqchessserver`, start with the multithreaded network server example from the lectures, gradually adding functionality for supported message types and operations.

Forbidden functions

You must not use any of the following C statements/directives/etc. If you do so, you will get zero (0) marks for the assignment.

- `goto`
- `#pragma`
- `gcc` attributes (other than the possible use of `__attribute__((unused))` as described in the style guide)

You must not use any of the following C functions. If you do so, you will get zero (0) marks for any test case that calls the function.

- `longjmp()` and equivalent functions
- `system()`
- `mkfifo()` or `mkfifoat()`
- `pthread_cancel()`
- `signal(2)`, `sigpending(2)`, `sigqueue(3)`, `sigtimedwait(2)`
- `sleep()`, `usleep()`, `nanosleep()` or any other function that involves a sleep, alarm or timeout.
- Functions described in the man page as non standard, e.g. `strcasestr()`. Standard functions will conform to a POSIX standard – often listed in the "CONFORMING TO" section of a man page. Note that `getopt_long()` and `getopt_long_only()` are an exception to this – these functions are permitted if desired.

The use of comments to control the behaviour of `clang-format` and/or `clang-tidy` (e.g., to disable style checking) will result in zero marks for automatically marked style.

Submission

Your submission must include all source and any other required files (in particular you must submit a `Makefile`). Do not submit compiled files (eg `.o`, compiled programs) or test files.

Your programs (named `uqchessclient` and `uqchessserver`) must build on `moss.labs.eait.uq.edu.au` and in the Gradescope environment with:

```
make
```

If you only implement one of the programs then it is acceptable for `make` to just build that program – and we will only test that program.

Your program must be compiled with `gcc` with at least the following options:

```
-Wall -Wextra -pedantic -std=gnu99
```

You are not permitted to disable warnings or use pragmas to hide them. You may not use source files other than `.c` and `.h` files as part of the build process – such files will be removed before building your program.

If any errors result from the `make` command (e.g. an executable can not be created) then you will receive zero marks for 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 zero marks for functionality).

Your program must not invoke other programs or use non-standard headers/libraries (besides those we have provided for you to use).

Your assignment submission must be committed to your Subversion repository under

```
svn+ssh://source.eait.uq.edu.au/csse2310-2024-sem2/csse2310-s4876492/trunk/a4
```

Only files at this top level will be marked so **do not put source files in subdirectories**. You may create subdirectories for other purposes (e.g. your own test files) but these will not be considered in marking – they will not be checked out of your repository.

You must ensure that all files needed to compile and use your assignment (including a Makefile) are committed and within the `trunk/a4` directory in your repository (and not within a subdirectory) and not just sitting in your working directory. Do not commit compiled files or binaries. You are strongly encouraged to check out a clean copy for testing purposes.

To submit your assignment, you must run the command

```
2310createzip a4
```

on `moss` and then submit the resulting zip file on Blackboard (a GradeScope submission link will be made available in the Assessment area on the CSSE2310/7231 Blackboard site)[12]. The zip file will be named

`s4876492_csse2310_a4_`*`timestamp`*`.zip`

where *`timestamp`* is replaced by a timestamp indicating the time that the zip file was created.

The `2310createzip` tool will check out the latest version of your assignment from the Subversion repository, ensure it builds with the command '`make`', and if so, will create a zip file that contains those files and your Subversion commit history and a checksum of the zip file contents. You may be asked for your password as part of this process in order to check out your submission from your repository. You will be asked to confirm references in your code and also to confirm your use (or not) of AI tools to help you.

You must not create the zip file using some other mechanism and you must not modify the zip file prior to submission. If you do so, you will receive zero marks. Your submission time will be the time that the file is submitted via GradeScope on Blackboard, and **not** the time of your last repository commit nor the time of creation of your submission zip file.

Multiple submissions to Gradescope are permitted. We will mark whichever submission you choose to "activate" – which by default will be your last submission, even if that is after the deadline and you made submissions before the deadline. Any submissions after the deadline[13] will incur a late penalty – see the CSSE7231 course profile for details.

Note that Gradescope will run the test suite immediately after you submit. When complete[14] you will be able to see the results of the "public" tests. You should check these test results to make sure your programs are working as expected.

Marks

Marks will be awarded for functionality and style and documentation. Marks may be reduced if you attend an interview about your assignment and you are unable to adequately respond to questions – see the CSSE7231 Student Interviews section below.

Functionality (70 marks for CSSE7231)

Provided your code compiles (see above) and does not use any prohibited statements/functions (see above), and your zip file has been generated correctly and has not been modified prior to submission, then you will earn functionality marks based on the number of features your program correctly implements, as outlined below. Not all features are of equal difficulty.

Partial marks will be awarded for partially meeting the functionality requirements. A number of tests will be run for each marking category listed below, testing a variety of scenarios. Your mark in each category will be proportional (or approximately proportional) to the number of tests passed in that category.

If your program does not allow a feature to be tested then you will receive zero marks for that feature, even if you claim to have implemented it. For example, if your client can never create a connection to a server, then we can not determine whether it can send the correct requests or not. If your server cannot establish and use a connection to the chess engine then many tests will fail.

Your tests must run in a reasonable time frame, which could be as short as a few seconds for usage checking to many tens of seconds in some cases. If your program takes too long to respond, then it will be terminated and you will earn no marks for the functionality associated with that test.

[12] You may need to use scp or a graphical equivalent such as WinSCP, Filezilla or Cyberduck in order to download the zip file to your local computer and then upload it to the submission site.

[13] or your extended deadline if you are granted an extension.

[14] Gradescope marking may take only a few minutes or more than 30 minutes depending on the functionality and efficiency of your code.

Exact text matching of output (`stdout` and `stderr`) and communication messages is used for functionality marking. Strict adherence to the formats in this specification is critical to earn functionality marks.

The markers will make no alterations to your code (other than to remove code without academic merit). Note that your client and server will be tested independently.

Marks will be assigned in the following categories. There are 20 marks for `uqchessclient` and 50 marks for `uqchessserver`.

1. `uqchessclient` correctly handles invalid command lines (2 marks)
2. `uqchessclient` connects to server (includes sending correct `start` request) and also handles inability to connect to server (2.5 marks)
3. `uqchessclient` correctly prints responses received from the server (~~1.5 marks~~ 1 mark)
4. `uqchessclient` correctly handles `newgame` commands (1 mark)
5. `uqchessclient` correctly handles `print` commands (1.5 marks)
6. `uqchessclient` correctly handles `hint` commands (1.5 marks)
7. `uqchessclient` correctly handles `possible` commands (1.5 marks)
8. `uqchessclient` correctly handles `move` commands, including `move` commands with invalid arguments (2.5 marks)
9. `uqchessclient` correctly handles `resign` commands (1.5 marks)
10. `uqchessclient` correctly handles `quit` commands and EOF on `stdin` (2 marks)
11. `uqchessclient` correctly handles invalid commands (2 marks)
12. `uqchessclient` correctly handles communication failure with server (includes handling SIGPIPE when writing to the socket) (1 mark)

13. `uqchessserver` correctly handles invalid command lines (2 marks)
14. `uqchessserver` establishes communication with the chess engine and handles failure to do so (2 marks)
15. `uqchessserver` correctly listens for connections and reports the port (including inability to listen for connections) (2 marks)
16. `uqchessserver` correctly handles `start` requests, including colour matching of human players, and initial move if playing computer as black. (4 marks)
17. `uqchessserver` correctly handles `board` requests, including in various game states (3 marks)
18. `uqchessserver` correctly handles `hint` requests, including in various game states (4 marks)
19. `uqchessserver` correctly handles `move` requests when playing the computer (including invalid move requests, but excluding those that result in check, checkmate or stalemate) (4 marks)
20. `uqchessserver` correctly handles `move` requests that result in check, checkmate or stalemate when playing the computer (including checking the result of subsequent requests of other types to ensure games are not considered to be in progress, if applicable) (4 marks)
21. `uqchessserver` correctly handles `move` requests when playing a human (in all scenarios, including checks described in the previous item) (5 marks)
22. `uqchessserver` correctly handles `resign` requests (3 marks)
23. `uqchessserver` correctly handles disconnecting clients and communication failure (including not exiting due to SIGPIPE, and receiving invalid commands from clients) (3 marks)
24. `uqchessserver` handles chess engine failure after startup (2 marks)
25. `uqchessserver` does not leak memory and does not busy wait (2 marks)

26. (CSSE7231 only) `uqchessserver` correctly implements client connection limiting (3 marks)
27. (CSSE7231 only) `uqchessserver` correctly implements SIGHUP statistics reporting (including protecting data structures with mutexes or semaphores) (7 marks)

Some functionality may be assessed in multiple categories. The ability to support multiple simultaneous clients will be covered in multiple categories. Multiple categories will include checks that the correct number of threads are created in handling clients (one additional thread per connected client). Multiple categories will test playing against both humans and computers – these are categories 16 to 18 and 22 to 27. At most 20 marks total will relate to playing against humans.

Style Marking

Text below this point is unchanged from assignment three (other than any specification updates at the end). You should still make sure that you are familiar with all of the requirements below.

Style marking is based on the number of style guide violations, i.e. the number of violations of version 3 of the CSSE2310/CSSE7231 C Programming Style Guide (found on Blackboard). Style marks will be made up of two components – automated style marks and human style marks. These are detailed below. Your style marks 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.

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.

You are encouraged to use the `2310reformat.sh` and `2310stylecheck.sh` tools installed on `moss` to correct and/or check your code style before submission. The `2310stylecheck.sh` tool does not check all style requirements, but it will determine your automated style mark (see below). Other elements of the style guide are checked by humans.

All `.c` and `.h` files in your submission will be subject to style marking. This applies whether they are compiled/linked into your executable or not[15].

Automated Style Marking (5 marks)

Automated style marks will be calculated over all of your `.c` and `.h` files as follows. If any of your submitted `.c` and/or `.h` files are unable to be compiled by themselves then your automated style mark will be zero (0). If your code uses comments to control the behaviour of `clang-format` and/or `clang-tidy` then your automated style mark will be zero. If any of your source files contain C functions longer than 100 lines of code[16] then your automated and human style marks will both be zero. If you use any global variables then your automated and human style marks will both be zero.

If your code does compile and does not contain any C functions longer than 100 lines and does not use any global variables and does not interfere with the expected behaviour of `clang-format` and/or `clang-tidy` then your automated style mark will be determined as follows: Let

- W be the total number of distinct compilation warnings recorded when your `.c` files are individually built (using the correct compiler arguments)
- A be the total number of style violations detected by `2310stylecheck.sh` when it is run over each of your `.c` and `.h` files individually[17].

Your automated style mark S will be

$$S = 5 - (W + A)$$

If $W + A \geq 5$ then S will be zero (0) – no negative marks will be awarded. If you believe that `2310stylecheck.sh` is behaving incorrectly or inconsistently then please bring this to the attention of the course coordinator prior to submission, e.g., it is possible the style checker may report different issues on moss than it does in the Gradescope environment. Your automated style mark can be updated if this is deemed to be appropriate. You can check the result of Gradescope style marking soon after your Gradescope submission – when the test suite completes running.

[15]Make sure you remove any unneeded files from your repository, or they will be subject to style marking.

[16]Note that the style guide requires functions to be 50 lines of code or fewer. Code that contains functions whose length is 51 to 100 lines will be penalised somewhat – one style violation (i.e. one mark) per function. Code that contains functions longer than 100 lines will be penalised very heavily – no marks will be awarded for human style or automatically marked style.

[17]Every `.h` file in your submission must make sense without reference to any other files, e.g., it must `#include` any `.h` files that contain declarations or definitions used in that `.h` file. You can check that a header file compiles by itself by running `gcc -c` *`filename.h`* – with any other `gcc` arguments as required.

Human Style Marking (5 marks)

The human style mark (out of 5 marks) will be based on the criteria/standards below for "comments", "naming" and "modularity". Note that if your code contains any functions longer than 100 lines or uses a global variable then your human style mark is zero and the criteria/standards below are not relevant.

The meanings of words like *appropriate* and *required* are determined by the requirements in the style guide.

Comments (3 marks)

Mark	Description
0	25% or more of the comments that are present are inappropriate AND/OR at least 50% of the required comments are missing
1	At least 50% of the required comments are present AND the vast majority (75%+) of comments present are appropriate AND the requirements for a higher mark are not met
2	All or almost all required comments are present AND all or almost all comments present are appropriate AND the requirements for a mark of 3 are not met
3	All required comments are present AND all comments present are appropriate AND additional comments are present as appropriate to ensure clarity

Naming (1 mark)

Mark	Description
0	At least a few names used are inappropriate
0.5	Almost all names used are appropriate
1	All names used are appropriate

Modularity (1 mark)

Mark	Description
0	There are two or more instances of poor modularity (e.g. repeated code blocks)
0.5	There is one instance of poor modularity (e.g. a block of code repeated once)
1	There are no instances of poor modularity

SVN Commit History Marking (5 marks)

Markers will review your SVN commit history for your assignment up to your zip file creation time. This element will be graded according to the following principles:

- Appropriate use and frequency of commits (e.g. a single monolithic commit of your entire assignment will yield a score of zero for this section). Progressive development is expected, i.e., no large commits with multiple features in them.
- Appropriate use of log messages to capture the changes represented by each commit. (Meaningful messages explain briefly what has changed in the commit (e.g. in terms of functionality, not in terms of specific numbered test cases in the test suite) and/or why the change has been made and will be usually be more detailed for significant changes.).

The standards expected are outlined in the following rubric. The mark awarded will be the highest for which the relevant standard is met.

Mark (out of 5)	Description
0	Minimal commit history – only one or two commits OR all commit messages are meaningless.
1	Some progressive development evident (three or more commits) AND at least one commit message is meaningful.
2	Progressive development is evident (multiple commits) AND at least half the commit messages are meaningful
3	Multiple commits that show progressive development of almost all or all functionality AND at least two-thirds of the commit messages are meaningful.
4	Multiple commits that show progressive development of ALL functionality AND meaningful messages for all but one or two of the commits.
5	Multiple commits that show progressive development of ALL functionality AND meaningful messages for ALL commits.

Total Mark

Let

- F be the functionality mark for your assignment (out of 70 for CSSE7231 students).
- S be the automated style mark for your assignment (out of 5).
- H be the human style mark for your assignment (out of 5).
- C be the SVN commit history mark (out of 5).
- V be the scaling factor (0 to 1) determined after interview (see the CSSE7231 Student Interviews section below) – or 0 if you fail to attend a scheduled interview without having evidence of exceptional circumstances impacting your ability to attend.

Your total mark for the assignment will be:

$$M = (F + \min\{F, S + H\} + \min\{F, C\}) \times V$$

out of 85 (for CSSE7231 students).

In other words, you can't get more marks for style or SVN commit history than you do for functionality. Pretty code that doesn't work will not be rewarded!

Late Penalties

Late penalties will apply as outlined in the course profile.

CSSE7231 Student Interviews

This section has been changed from assignments one and three – only a subset of students will be interviewed.

The teaching staff will conduct interviews with a subset of CSSE7231 students about their submissions, for the purposes of establishing genuine authorship. If you write your own code, you have nothing to fear from this process. If you legitimately use code from other sources (following the usage/referencing requirements outlined in this assignment, the style guide, and the AI tool use documentation requirements) then you are expected to understand that code. If you are not able to adequately explain the design of your solution and/or adequately explain your submitted code (and/or earlier versions in your repository) and/or be able to make simple modifications to it as requested at the interview, then your assignment mark will be scaled down based on the level of understanding you are able to demonstrate and/or your submission may be subject to a misconduct investigation where your interview responses form part of the evidence. Failure to attend a scheduled interview will result in zero marks for the assignment unless there are documented exceptional circumstances that prevent you from attending. Students will be selected for interview based on a number of factors that may include (but are not limited to):

- Feedback from course staff based on observations in class, on the discussion forum, and during marking;
- An unusual commit history (versions and/or messages), e.g. limited evidence of progressive development;

- Variation of student performance, code style, etc. over time;
- Use of unusual or uncommon code structure/functions etc.;
- Referencing, or lack of referencing, present in code;
- Use of, or suspicion of undocumented use of, artificial intelligence or other code generation tools; and
- Reports from students or others about student work.

Specification Updates

Any errors or omissions discovered in the assignment specification will be added here, and new versions released with adequate time for students to respond prior to due date. Potential specification errors or omissions can be discussed on the discussion forum.

Version 1.1

- Clarify that a client disconnection is to be treated as a resignation (line 429).
- Clarify the *`colour`* argument with a "`start`" command being sent by `uqchessclient` when playing the computer (line 122).
- Fixed mistake on line 164.
- Adjusted marking scheme to account for client sending initial `start` requests (lines 793 to 795).
- Clarify choice of players and assignment of colours when two humans are playing (lines 449 and 450).
- Added `signal(2)` and related functions to the list of forbidden functions (line 710)
- Corrected mistake in protocol description - an "`ok`" message is always sent after a valid move (lines 302 and 310).

Version 1.2

- Clarified when a `ucinewgame` UCI command may be used (lines 588 to 592).
- Clarified when a `position` UCI command may be used (lines 604 to 606).
- Clarified that a `go` UCI command must be preceded by a `position` command (line 616 and subsequent lines and the example) – possibly with an intervening `d` command.
- Clarified that a `d` UCI command must always be preceded by a `position` command (lines 626 to 627).
- Clarified checking of the argument to a `move` command. It must be 4 to 5 alphanumeric characters but must not be further checked by the server (lines 241 to 244).
- Added hint about the function to use to check that characters are alphanumeric (line 690).
- Clarified that the argument associated with a `moved` response must be unchanged from that supplied by a human opponent (line 297).
- Noted that marking category 23 includes the server handling invalid commands from the client (line 825).