The University of Queensland
School of Information Technology and Electrical Engineering
Semester Two, 2018
CSSE2310 / CSSE7231 - Assignment 4
**Due: 11:10pm 26th October, 2018**
Marks: 50
Weighting: 25% of your overall assignment mark (CSSE2310)
Revision 1.4

# Introduction

Your task is to write some **C99** programs that play, run and interact with networked versions of the game described in Assignment 3. In this assignment you will use `pthreads` and **not** `fork()`. This means instead of using pipes to communicate between your processes, you will use connections over the network.

*Your programs must not create any files on disk not mentioned in this specification or in command line arguments.* Your assignment submission must comply with the C style guide (version 2.0.4) available on the course blackboard area. This is an individual assignment. You should feel free to discuss aspects of C programming and the assignment specification with fellow students. You should not actively help (or seek help from) other students with the actual coding of your assignment solution. It is cheating to look at another student's code and it is cheating to allow your code to be seen or shared in printed or electronic form. You should note that all submitted code may be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion, formal misconduct proceedings will be initiated against you. A likely penalty for a first offence would be a mark of 0 for the assignment. Don't risk it! If you're having trouble, seek help from a member of the teaching staff. Don't be tempted to copy another student's code. You should read and understand the statements on student misconduct in the course profile and on the school web-site: `http://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism`

As with Assignment 1 and 3, we will use the subversion (svn) system to deal with assignment submissions. Do not commit any code to your repository unless it is your own work or it was given to you by teaching staff. **If you have questions about this, please ask.**

**While you are permitted to use sample code supplied by teaching staff this year in this course, code supplied for other courses or other offerings of this course is off limits — it may be deemed to be without academic merit and removed from your program before testing.**

# The Game

The game rules are the same as Assignment 3. The communication protocol and exit conditions have some additions to deal with network functionality. Additionally for this assignment you will use a course provided library to deal with game logic. Use of this library is **compulsory**.

## Game Library

The library provided for this assignment is designed to handle the majority of the game logic and protocol messages from Assignment 3. You are not allowed to use your version of the game logic from Assignment 3, nor are you allowed to write your own version for Assignment 4. Instead, your player and server **must** use the library to provide the game logic functionality. You should familiarise yourself with the game library by reading the header files, and if required, the library implementation.

## Key file

The key file is used by both players and servers as a form of authentication. Valid key files always contain at least 1 character, and no newlines. The entire contents of the key file makes up the key for the program.

# Programs

You will be required to write the following programs:

- `gopher` — A **client scores** program that will connect to the server and display information that it is sent from the server.
- `zazu` — A **client player** program that receives game moves from `stdin` and plays a game after connecting to a server. This program is similar to the players in Assignment 3, but requires a user to provide the moves instead of deciding automatically.
- `rafiki` — A **server** program which runs the games. Players and other clients will connect to this. This program serves the same role as the hub from Assignment 3, coordinating and running games.

## Program Behaviour — Scores

The scores client will be used to retrieve scores from a server and display them to the user. It will connect to a server on a port, sending a `scores` message to indicate this is a scores client. If the connection is accepted, the server will send a `yes` message to indicate success. If any other message is received apart from `yes`, the scores client will exit due to connecting to an invalid server. Upon success, the server will send a stream of game information, which the client should simply print to `stdout`. The scores client will remain connected until the server disconnects, at which point it will exit normally.

## Program Behaviour — Player

### Connect to a game

The most common use of the player will be to connect to a server to play a game. To do this, players will connect to `localhost` on the provided port. If a player successfully connects to the server they will attempt to authenticate with the server by sending a `play` message. The server will check the provided key against its key. The server will reply with a `yes` message if the key matches, or a `no` message if the keys do not match. Any message other than `yes` will cause the player to disconnect and exit with an authentication error.

After the authentication is confirmed, the players will send the following (in order):

- the game name, terminated with a newline.
- their player name, terminated with a newline.

Valid player names and game names do not include newline characters, nor do they include commas. There is no restriction on multiple games with the same name, or multiple players with the same name joining a game. Any players joining an existing game that is already at capacity will "overflow" into a new game. The newline on the end of these two messages should not be included in the actual player or game name.

### Reconnect after leaving

Alternatively, if the player is reconnecting they will send a `reconnect` message instead of the `play` message. If authentication is successful they will send the reconnect id via a `rid` message. If the reconnect id is valid, then the server will respond with `yes`. If the player receives `no` in the authentication or reconnect step then the player should disconnect.

Once the authentication is complete, the reconnected player will receive game state information from the server. It will then operate like a normal player, playing the game to completion.

## Program Behaviour — Server

The server will be responsible for running games, as well as sending information to scores clients. The server will listen on a set of ports on `localhost`.

Any errors/failures in a game or during a connection procedure should not cause deadlocks, errors or any other failures in any other parts of the server. E.g. if a client is non-responsive during the connection handshake, then this should not deadlock or cause any failures in any other part of the server.

Remember you are to not use any non-blocking IO or `fork()` in this assignment.

### Statfile

The statfile is a file with a list of newline separated entries describing ports to listen on and the games that run on them. An entry is in the form: *port*,*tokens*,*points*,*players*

- *port* — The port to listen on
- *tokens* — The number of tokens the game starts with (positive integer)
- *points* — The number of points to win the game (positive integer)
- *players* — The number of players in the game (2-26)

These values will dictate the parameters of games started through that port.

An example statfile:

```
2310,20,50,10
0,9,14,3
```

A valid port falls within the range of 1-65535. However if the value on a line is 0, then that port should be an ephemeral port (ie. one that is chosen by the kernel). A valid statfile has no extra whitespace, and must have at least one entry. A valid statfile does not have an newline character for the last entry. All entries should have a unique port (i.e. no duplicate ports), with the exception of multiple ephemeral ports (ie. port number 0) in the file.

Once the server starts it should read this file and begin listening for connections on the given ports. Once all listeners are set up, the server should print out a single-space separated list of all the ports it is listening on to `stderr`, in the order that they were read from the statfile[1]. This list should have a newline at the end, and no extra whitespace. For example: `2310 45894`

When the server receives a SIGINT signal, it should reload the statfile. This is done by rereading the statfile for a new list of entries to use, then setting up listeners for each of the ports in the new file. All previous listeners should be shutdown (even if they are identical) and cleaned up before the server starts listening on the new ports. Games currently in progress should continue until they are finished, even if the port they started on is no longer in the statfile. Note that this means your program should **not** lock the statfile from being modified during run-time. Any failures when rereading/opening listeners should exit with the appropriate error. This reread should also print out the ports again as described above.

[1] Any entries with a port number of 0 should be replaced by the ephemeral port the server is listening on.

### Playing a game

Following successful authentication, players will be put in games identified by the provided game name. Once a game has reached its capacity of players, the server will start the game.

The parameters of this game will be determined by the server, based on the port the game was started on. For example, if the first player connects to a game on port 3001, and the statfile used by the server includes the details "3001,5,20,10", then they will have connected to a game that will start with 5 tokens, 20 points to win and a capacity of 10 players. If there is an existing game with the provided game name, and that game is full of players (ie. it has already started), this will "overflow" and be treated as an entirely new game, following the same procedure as above. Note that game names are shared across all ports being listened on by the server. For example:

- Scenario 1:
  A server is running with two ports, 1234 (two player games) and 5678 (three player games). A player connects to port 1234 to play a game, using the game name `game1`. There are no current games with this name, so a new game is created, requiring two players to play (the game counter for the reconnect id will be 1). Another player connects to port 5678 to play a game, also using the game name `game1`. They are placed in the existing `game1` game. Now that there are two players in the game, the game will begin.

- Scenario 2:
  A server is running with two ports, 2345 (two player games) and 6789 (three player games). A player connects to port 2345 to play a game, using the game name `game2`. There are no current games with this name, so a new game is created, requiring two players to play (the game counter for the reconnect id will be 1). Another player connects to port 6789 to play a game, also using the game name `game2`. They are placed in the existing `game2` game. Now that there are two players in the game, the game will begin. A third player connects on port 6789 and asks to play a game, using game name `game2`. There is already a game with this name, however it is now full of players and in progress. The server will create a new game with name `game2`, which will this time require three players to play (the game counter for the reconnect id will be 2).

Players will be sorted by their names in lexicographic order before being assigned their player id. If there are multiple players in the game with the same name, these players will be sorted based on the order they are added to the game. For example, player "Bob" should come before player "Dan", no matter the connection order. However, two players with the name "Jane" will be sorted based on when they were added to the game, ie. after successfully authenticating and sending a valid game name and player name.

To begin, the server will send all players the following messages in this order:

1. `rid` — The Reconnect Id for the player, used for later reconnection to this game.
2. `playinfo` — Player information, including the letter for this player and the number of players in the game (from 2 to 26 inclusive).
3. `tokens` — Number of tokens in each non-wild pile (as per Assignment 3)

### Reconnect Id

A Reconnect Id is issued to every player at the start of a game. This is used by the player if they leave the game and want to reconnect to it.

A Reconnect Id ($R$) should be in the form: $G$,$GC$,$PID$

- G — Game name
- GC — Game counter. This is a positive integer that should increment each time a game with this name is played.
- PID — Player Id (0-25 inclusive)

### Reconnects

When a player reconnects correctly, they should be placed in their original game. If a player attempts to connect to a game that no longer exists (ie. due to the timeout passing), they should be sent a `no` message.

The server will then send the reconnected player the `playinfo` and `tokens` messages, as described above. Following these, the entire game state will be sent using the following format:

1. Send each card in the market (in order oldest to newest), using the `newcard` message.
2. Send details of each player (in order of player letter) using the `player` message.

Players should use these messages to "catch up" to the current state of the game. Once this has been done, the game should continue from where it was up to.

### Scores

The server must keep a high scores table which tracks information about all players, across all games, on all ports. The information that will be recorded includes:

- Player name
- Total tokens picked up
- Total points earned

The high scores table will include information from all completed games, as well as all games currently in progress. Each player name should only appear once. Note that player names can be shared across many games, as well as appearing multiple times within the same game, and all of these should be combined into a single table entry. The high scores table is sorted in the following order:

1. Total points, highest to lowest
2. Total tokens, lowest to highest (for players with the same total points)

For example:

- Scenario 1:
  Player "Fred" in Game 1 has 5 points, and a Player "Fred" in Game 2 has 9 points. The scores table will combine their scores to show a single entry for Player "Fred" having 14 points.
- Scenario 2:
  Player "Mary" has already player some games, and has a score of 6 points. There is currently a game in progress that has two players with the name "Mary", one on 1 point and the other on 3 points. The scores table will combine all of these scores to show a single entry for "Mary" having 10 points.

Once a scores client has connected to the server, the server will send it the current high scores table in a comma separated values (CSV) format. First it will send a header line describing each of the columns:
`Player Name,Total Tokens,Total Points`
Following this, the server will send all of the entries in the high scores table. Each entry will be sent on a new line, in the format:
*Name*,*TotalTokens*,*TotalPoints*
After this table is sent, the server should immediately disconnect from the scores client.

For example:

```
Player Name,Total Tokens,Total Points
Fred,7,14
Mary,4,10
Jane,6,10
```

The server will accept requests for scores on all of the ports it is listening on. Regardless of the port the scores request game from, the server will always respond with the entire high scores table, with results from all games on all ports.

# Invocation

## Invocation — Scores

The arguments (in order) to start a scores client are as follows:

1. Port — a valid port number (1-65535) to connect to

For example:
`./gopher 2310`
would start a scores client that connects to port 2310 to get the high scores table.

## Invocation — Player

The arguments (in order) to start a player are as follows:

1. Key file — A file to read the key from
2. Port — a valid port number (1-65535) to connect to
3. Game Name — the name of the game to join (a non-empty string, containing no newline or commas)
4. Player Name — the player name (a non-empty string, containing no newline or commas)

For example:
`./zazu shortkey 1234 Interesting_Game Alice`
would start a player using the *shortkey* file for the key and connects to port 1234. It then connects to a game with the name *Interesting_Game* with a player name of *Alice*. The word `reconnect` is not a valid game name, as it is reserved for the reconnect mode of the player.

Or if the player is reconnecting to a game then:

1. Key file — A file to read the key from
2. Port — a valid port number (1-65535) to connect to
3. "reconnect" — The literal word "reconnect"
4. reconnect_id — a string for the reconnection id

For example:
`./zazu shortkey 567 reconnect Interesting_Game,1,18`
would start a player using the *shortkey* file for the key and connects to port 567. It then reconnects to a game with a reconnect id of *Interesting_Game,1,18*.

## Invocation — Server

The arguments (in order) to start the server are as follows:

1. Key file — A file containing the key required when connecting to play a game on this server.
2. Deck file — The deck file that all games use.
3. Stat file — A file containing details about the ports the server will listen on and the games played on each of these.
4. Timeout — A non-negative integer for number of seconds to wait for a reconnect from a player if they disconnect. If this is zero then do not wait.

For example:
`./rafiki servkey mydeck mystats 3`
would start a server using the *servkey* file for the key, the *mydeck* file for the deck, the *mystats* file for the stats and having a timeout of 3 seconds.

The timeout value should be used with the reconnection logic for players. If players do not reconnect after being disconnected for the timeout duration, all clients should receive the appropriate `disco` message to end the game.

# Messages

## Messages — Scores to Server

| Encoded form | Parameters | Purpose |
|---|---|---|
| scores | | Ask the server to provide scores information |

## Messages — Player to Server

The messages sent by the player to the hub in Assignment 3 will be used when the player communicates with the server. There are additional messages to support the authentication and reconnect functionality of the server.

| Encoded form | Parameters | Purpose |
|---|---|---|
| play$K$ | K — Key for the server | Ask the server to play a game |
| reconnect$K$ | K — Key for the server | Ask the server to reconnect to a game |
| rid$R$ | R — Reconnect Id | Inform the server what to reconnect to |

## Messages — Server to Players

The server will use all messages that the hub sends to players, as defined in Assignment 3. There are some new additions to the messages to inform players of different situations within the game, as described in the table below.

| Encoded form | Parameters | Purpose |
|---|---|---|
| yes | | Accept the initial connection message (`play`, `reconnect`, `scores`) |
| no | | Reject the initial connection message (`play`, `reconnect`), if authentication is unsuccessful |
| rid$R$ | R — Reconnect id | Inform the player of their reconnect id |
| playinfo$P/NP$ | P — Letter for this player<br>NP — Number of players in the game (2-26 inclusive) | Inform player of their player letter and how many players are in the game |
| player$P$:$TP$:d=$D_P, D_B, D_Y, D_R$ :t=$T_P, T_B, T_Y, T_R, T_W$<br><br>Note: Actual message has no line break in the middle. Only done here for display purposes. | P — The player this state is for<br>TP — The total points the player has<br>$D_?$ — Number of each colour of discount the player has<br>$T_?$ — Number of each colour of token the player has | Inform a reconnected player of the current state of a player |
| disco$P$ | P — Player that disconnected early | Inform players that a player has disconnected early, causing the game to end |
| invalid$P$ | P — Player that sent two invalid messages in a row (as described in Assignment 3) | Inform players that a player sent invalid messages, causing the game to end |

Note that the `disco` and `invalid` messages are sent instead of the `eog` message for those conditions. Players should exit early as appropriate when they receive these messages.

# Output

## Output — Scores

Upon successful connection (receiving a `yes` from the server), the scores client should print all information sent to it to standard out. Once the server closes the connection, the scores client should exit normally. The initial `yes` message should not be printed.

## Output — Player

All players should continue to output the same information as described for players in Assignment 3. **However**, instead of printing this to standard error, the player should print its output to **standard out**.

In addition, the player should send the following to standard out when these messages are received (do not print the full game state for these).

| Message | Output to stdout | Parameters |
|---|---|---|
| rid$R$ | R | R — The reconnect id for the player. |
| player$P$:$TP$:d=$D_P, D_B, D_Y, D_R$ :t=$T_P, T_B, T_Y, T_R, T_W$ | Player *letter*:*TotalPoints* :Discounts=$D_P, D_B, D_Y, D_R$ :Tokens=$T_P, T_B, T_Y, T_R, T_W$ | As per Assignment 3 game state |

### Prompting the user

After receiving a `dowhat` message and printing out *Received dowhat*, the player will prompt the user for an action. Prompts will be printed to standard out, and the user will respond via standard in. Please note that all prompts described have a single space following the `>`.

First, the user will be asked what action they wish to use. This will be done by printing a prompt:

```
Action>
```

The user will then enter one of: `purchase`, `take` or `wild`. After this, additional prompting will be done based on the action.

For a `purchase` action, the user will then be prompted for the card they wish to buy:

```
Card>
```

The user must then enter a number from 0-7, inclusive. After this, the user will be prompted for the tokens to use. The player will then prompt the user for how many tokens of each colour to use for this purchase, in the order: P, B, Y, R, W. The prompt for each of these will be in the form:

```
Token-?>
```

where `?` is replaced with the appropriate token colour letter from above. The player will not prompt a user for a colour if they do not have any tokens of that colour. A valid response from the user is a non-negative number of tokens, up to the number of tokens of that colour currently held by the player. After all information is received, the player will send a `purchase` message to the server.

For a `take` action, the user will then be prompted how many of each colour token they would like to take, in the order: P, B, Y, R. The prompt for each of these will be in the form:

```
Token-?>
```

where `?` is replaced with the appropriate token colour letter from above. A valid response from the user is a non-negative number of tokens up to the maximum number available in the game. After all information is received, the player will send a `take` message to the server.

No additional prompting will be done if the user performs a `wild` action.

At any of the prompts above, if the user input is invalid, the player should display **that** prompt again to obtain a valid value. There is no limit to how many times the user will be reprompted by the player. The player should not send an action message to the server until a valid message (in terms of structure) has been constructed from the user input. The server may still reprompt the player for a move as per Assignment 3, by sending a `dowhat` message again.

## Output — Server

After setting up all listeners for the ports in the statfile, the server will print out all of the ports it is listening on to standard error. The ports will be sorted in the order they were in the statfile. This will be in the form of a space separated list on a single line, with no extra whitespace. For example: `2310 45894`

# Program Exit

## Exit — Scores

Upon any exit the scores client should ensure all allocated memory is freed, and any connections/sockets are cleaned up.

| Condition | Exit | Message to stderr |
|---|---|---|
| Normal exit due to server disconnect | 0 | |
| Incorrect number of arguments | 1 | `Usage: gopher port` |
| Connection/Port Error | 3 | Failed to connect |
| Invalid server | 4 | Invalid server |

## Exit – Player

Upon any exit the player should ensure all allocated memory is freed, and any connections/sockets are cleaned up.

| Condition | Exit | Message to stderr |
|---|---|---|
| Normal exit due to `eog` | 0 | |
| Incorrect number of arguments | 1 | `Usage: zazu keyfile port game pname` |
| Invalid/Missing keyfile | 2 | Bad key file |
| Invalid game/player name | 3 | Bad name |
| Connection/Port error | 5 | Failed to connect |
| Authentication error | 6 | Bad auth |
| Reconnect id was invalid | 7 | Bad reconnect id |
| Server communication error (Bad message format or socket closed before end of game) | 8 | Communication Error |
| Other player disconnected (after `disco`) | 9 | Player $P$ disconnected<br><br>NOTE: Replace $P$ with the player letter |
| Other player sent invalid messages (after `invalid`) | 10 | Player $P$ sent invalid message<br><br>NOTE: Replace $P$ with the player letter |

## Exit — Server

The server will continue to run until it receives the `SIGTERM` signal or encounters an error condition, as described in the Exit Statuses section. Before the server exits the server should "cleanly" shutdown all games, including sending `eog` to all clients, freeing any allocated memory and ensuring any threads/connections/listeners are appropriately cleaned up.

| Condition | Exit | Message to stderr |
|---|---|---|
| Normal exit due to SIGTERM | 0 | |
| Incorrect number of arguments | 1 | `Usage: rafiki keyfile deckfile statfile timeout` |
| Invalid/Missing keyfile | 2 | Bad keyfile |
| Invalid/Missing deckfile | 3 | Bad deckfile |
| Invalid/Missing statfile | 4 | Bad statfile |
| Bad timeout | 5 | Bad timeout |
| Couldn't listen on port | 6 | Failed listen |
| System call failure (malloc failed, etc) | 10 | System error |

## Memory Usage

It is expected that your programs will allocate memory for use while operating. Your programs should operate without leaking memory. This means that any memory allocated by your programs will be freed before the programs exit. Memory usage and leakage will be checked for both the server and clients.

## Suggested Stages of this Assignment

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

1. Scores client. Remember you can use `netcat` to act as the server to test this, so you do not need any server code for this stage.
2. Player client — argument checking and connecting to a port.
3. Server — argument checking, statfile and deckfile reading. Setup a port to listen for and accept connections.
4. Write the player to play a game against a `netcat` server.
5. Write the server to play a single game on a single port.
6. Write the server to play a multiple games on a single port.
7. Write the server to play multiple games on multiple ports.
8. Other server functionality, such as scores, reconnect and reloading the statfile.

## Compilation

Your code must compile (on a clean checkout) with the command:
`make`

Each individual file must compile with at least
`-Wall -pedantic -std=gnu99 -I/local/courses/csse2310/include`
and link with
`-L/local/courses/csse2310/lib -la4`

You may of course use additional flags but you must not use them to try to disable or hide warnings. You must also not use pragmas to achieve the same goal. Your code must be compiled with the `gcc` compiler.

If the make command does not produce one or more of the required programs, then those programs will not be marked. If none of the required programs are produced, then you will receive 0 marks for functionality. Any code without academic merit will be removed from your program before compilation is attempted (This will be done even if it prevents the code from compiling). If your code produces warnings (as opposed to errors), then you will lose style marks (see later).

Your solution must not invoke other programs apart from those listed in the command line arguments for the hub. Your programs are not permitted to create additional files on the filesystem. Your solution must not use non-standard headers/libraries. Further, you are not permitted to make use of gnu language extensions or any of the following: `__attribute__`, `getdelim`, `getline`, `setjump`, `longjump`, `goto`, `select`. You are not permitted to use posix regex features. You are also not to use non-blocking IO nor `setbuff()` or equivalents to switch off buffering on `FILE*`.

## Submission

Submission must be made electronically by committing using subversion. In order to mark your assignment, the markers will check out `/trunk/ass4/` from your repository on `source.eait.uq.edu.au` [2]. Code checked in to any other part of your repository will not be marked.

[2] That is, `https://source.eait.uq.edu.au/svn/csse2310-$USER/trunk/ass4`

The due date for this assignment is given on the front page of this specification. (Only the contents of the `trunk/ass4` directory at the deadline will be marked).

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

**Note:** Any .h or .c files in your `trunk/ass4` directory will be marked for style *even if they are not linked by the makefile*. If you need help moving/removing files in svn, then ask. Consult the style guide for other restrictions.

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

# Marks

Marks will be awarded for both functionality and style.

## Functionality (42 marks)

Provided that your code compiles (see above), you will earn functionality marks based on the number of features your program correctly implements, as outlined below. Partial marks may be awarded for partially meeting the functionality requirements. Not all features are of equal difficulty. If your program does not allow a feature to be tested then you will receive 0 marks for that feature, even if you claim to have implemented it. For example, if your program can never open a file, we can not determine if your program would have loaded input from it. The markers will make no alterations to your code (other than to remove code without academic merit). Your programs should not crash or lock up/loop indefinitely. Your programs should not run for unreasonably long times.

- For *gopher* scores client
  - Argument checking (1 mark)
  - Detects invalid start (1 mark)
  - Correctly handles scores output from server (2 marks)
- For *zazu* player client
  - Argument checking (1 mark)
  - Correctly connects to server (2 marks)
  - Correctly prompts user (1 mark)
  - Correctly handles game play (2 marks)
  - Correctly handles invalid moves by user (2 marks)
  - Correctly handles reconnect (2 marks)
- For *rafiki* server
  - Argument checking (1 mark)
  - Correctly loads statfile and prints ports if successful (2 marks)
  - Correctly handles connection sequence for 1 game (2 marks)
  - Run a single game on one port (2 marks)
  - Run multiple concurrent games on one port (4 marks)
  - Run multiple concurrent games on multiple ports (5 marks)
  - Correctly reload statfile, including game cleanup (3 marks)
  - ~~Scores correctly printed for 1 game (2 marks)~~
  - ~~Scores correctly printed for many games (3 marks)~~
- Memory usage
  - *gopher* does not leak memory under normal operation (1 mark)
  - *zazu* does not leak memory under normal operation and shutdown (1 mark)
  - *rafiki* does not leak memory after normal operation and shutdown (1 mark)
  - *rafiki* does not leak memory after reloading statfile (SIGINT) and shutdown (1 mark)

### Style (8 marks)

Style marks will be calculated as follows:
Let $A$ be the number of style violations detected by simpatico plus the number of build warnings. Let $H$ be the number of style violations detected by human markers. Let $F$ be the functionality mark for your assignment.

- If $A > 10$, then your style mark will be zero and $M$ will not be calculated.
- Otherwise, let $M_A = 4 \times 0.8^A$ and $M_H = M_A - 0.5 \times H$. Your style mark $S$ will be $M_A + \max\{0, M_H\}$.

Your total mark for the assignment will be $F + \min\{F, S\}$.

A maximum of 5 violations will be penalised for each broad guideline area. The broad guideline areas are Naming, Comments, Braces, Whitespace, Indentation, Line Length and Overall. For naming violations, the penalty will be one violation per offending name (not per use of the name) up to the maximum of five. You should pay particular attention to commenting so that others can understand your code. The marker's decision with respect to commenting violations is final — it is the marker who has to understand your code. To satisfy layout related guidelines, you may wish to consider the `indent(1)` tool. Your style mark can never be more than your functionality mark — this prevents the submission of well styled programs which don't meet at least a minimum level of required functionality.

## Late Penalties

Late penalties will apply as outlined in the course profile.

## Specification Updates

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

## Test Data

Test data and scripts for this assignment will be made available (`testa4.sh`, `reptesta4.sh`). The idea is to help clarify some areas of the specification and to provide a basic sanity check of code which you have committed. *They are not guaranteed to check all possible problems nor are they guaranteed to resemble the tests which will be used to mark your assignments.* Testing that your assignment complies with this specification is still *your* responsibility.

## Notes

1. Start early
2. Write simple programs to experiment/learn how to use pthreads, as well as networking.
3. Netcat will help you test your programs.
4. Your programs must run and work on moss.
5. You are not permitted to use non-blocking IO.
6. You are not permitted to use any of the following functions:
   - `select()`, `pselect()`, `poll()`, `epoll()`, `system()`, `fork()`, `exec()`, `pipe()`, `popen()`, `prctl()`, `setjmp()`, `asprintf()`
7. You may not use any `#pragma` or `goto` in this assignment.
8. You may not use libevent or kqueue in this assignment.

9. You may not use regex in this assignment.
10. The server should validate that the players are not cheating/attempting to do thing that would be impossible in the given game state. The players do not have to do the same thing for the server.
11. You will need to do something with SIGPIPE.
12. All messages are sent as strings, terminated by a new line. There are no null bytes within a valid message.
13. Valid messages contain no leading, trailing or embedded spaces. Messages must be the exactly the correct length.
14. `structs` and `enums` are your friends, use them!
15. All programs detect failure of other programs by **read failures**. Write failures must be silently ignored.
16. `valgrind` will help you find memory leaks and other memory issues.
17. For the purposes of this assignment, the positive integers do not include 0 (it's 1, 2, 3, ...). The non-negative integers do include 0 (it's 0, 1, 2, 3, ...).
18. After a `no` message to a client, the server can just disconnect. There is no need to wait for the client to disconnect.
19. The player letter (often referred to as $P$) is the letter corresponding to the player id, ie. player 0 is indicated by 'A', 1 by 'B' and so on.
20. The server should use `SOMAXCONN` for its backlog when listening on a port.
21. All networking will be done using IPv4.
22. If you are using `fdopen` on your connected sockets — which we strongly recommend — you will want to have two separate `FILE*` (one for read and one for write). To avoid the possibility of a race condition, consider doing the following:

```
FILE* i = fdopen(s, "r"); // variable names deliberately awful
int s2 = dup(s);
FILE* j = fdopen(s2, "w");
```

23. In your server you should ensure that sockets being listened on can be reused immediately after closing them. To do this, set the socket to be reusable via the `SO_REUSEADDR` flag. This should be done after creating the socket, and before binding to a port.

```
// socketFd is the file descriptor for your open socket
int optVal = 1;
if (setsockopt(socketFd, SOL_SOCKET, SO_REUSEADDR, &optVal, sizeof(int)) < 0) {
    error("Failed to set socket option");
}
```

# Updates

- 1.3 → 1.4
  - Clarify procedure for reloading statfile.
  - Clarify player output for messages from server (ie. display player state after receiving a `player` message).
  - Clarify user reprompting — only the prompt at which an invalid input was received will be shown again.
  - Add note about socket creation.
- 1.2 → 1.3
  - Clarify player authentication error.
  - Clarify that statfile can contain multiple ephemeral ports (ie. port number 0).
  - Add scenarios for server playing a game, to better clarify game name and game counter.
  - Clarify server scores behaviour — accept scores requests on all ports.
  - Clarify player exit status for server communication error.
  - Add *Suggested Stages of this Assignment* section as suggestion for how to approach the assignment.
  - Updated banned functions list.
  - Add note about `fdopen`.
- 1.1 → 1.2
  - Minor wording and grammatical fixes
- 1.0 → 1.1
  - Fixed due date to be 11:10pm Fri 26 Oct 2018
  - Add player exit status for invalid game/player name
  - Add note about server listen backlog
  - Add note about networking protocol
  - Adjust marking scheme to clarify memory usage after SIGINT (reload statfile)
  - Updated banned functions list