
Project Report
System Programming
2nd Sem. 2019/20

# Distributed Pacman

Group 55
Vasco Candeias
nr. 84196
vascocandeias@tecnico.ulisboa.pt

Professor: João Nuno de Oliveira e Silva

June 4, 2020

# 1 Architecture

To have a multiplayer pacman game, two programs were developed, each with its own node – the client and the server. In figure 1, the architecture for both of these can be seen, where a simple client with two threads is shown along with a more complex server split into different modules – `players`, `fruits`, `communication`, `board` and `UI_library` –, all of them represented in dotted lines.

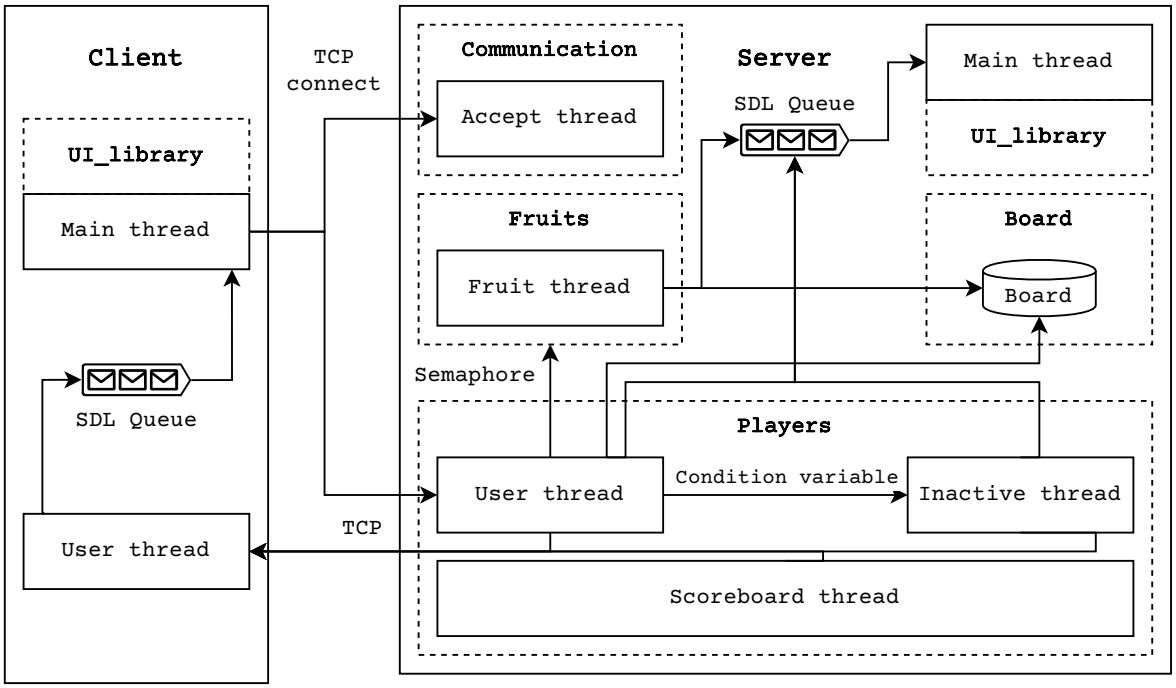


Figure 1: Project architecture.

Focusing on the server, in order to have multiple clients connecting to it and to handle each request in parallel, multiple threads were used. This section focus on the server threads, which are the following:

- `main` – main thread of the server, where the server and board are initialised and where the SDL functions are called. There is only one instance of this thread. This thread is also responsible to launch the accept and scoreboard threads.
- `accept` – this thread is responsible for listening to a socket and accepting any connection requests. When there is one, this thread will launch one user thread and, if the number of players is higher than one, will also launch two fruit threads.
- `user` – with one of these threads per user, this is the one responsible for waiting for the clients' requests, handling them and broadcasting the result to every client. Before doing so, this thread also starts two threads, responsible for handling the inactivity of each of the user's characters.

- `inactivity` – this thread implements the functionality of detecting that a user has been inactive for more than 30 seconds and its implementation is described in section 6. There is one of these threads for each character (that is, two per client).

- `fruit` – this thread implements the functionality of a fruit a placing it 2 seconds later and its implementation is described in section 6. There is one of these threads for each fruit.

To sum it up, there are always $1 + 5 \times$ `n_players` threads for `n_players` $> 0$ and three threads when no client is connected.

Finally, it is important to mention that a makefile is provided with the following commands:

- `make server` – compiles the server.

- `make client` – compiles the client.

- `make clean-server` – deletes every `*.o` file and the server executable.

- `make clean-client` – deletes every `*.o` file and the client executable.

To run the resulting programs, the commands are:

- `./server [filename.txt]` – run the server with the board configuration in `filename.txt`. If none is provided, the file used is named `board.txt`. If this does not exist, default dimensions are used.

- `./client server-ip server-port color-r color-g color-b` – run a client with the server at `server-ip:server-port` and with the color with the given rgb.

# 2 Code organisation

This section is dedicated to describing each module and listing their functions, along with the respective documentation.

## 2.1 Communication module

The communication module provides a function to initialise the server and a thread to accept connections, as explained in the previous section. The function declarations are done in `communication.h` and their implementation in `communication.c`.

```c
/*
 * description: initialize the server
 *
 * argument: port [int] - port to which the socket should be binded
 * returns: int - server socket file descriptor
 */
int init_server(int port);

/*
 * description: thread responsible for accepting new clients
 *
 * argument: arg [int*] - server socket file descriptor
 * returns: void* - NULL
 */
void* thread_accept(void* arg);
```

## 2.2 Board module

This module is responsible for every function that accesses or modifies the game board, being also responsible for the mutex board management. The function declarations are done in `board.h` and their implementation in `board.c`.

```c
/*
 * description: structure for each place in the board
 */
typedef struct place {
  character type;
  int id;
  int color[3];
  int kill_count;
} place;

/*
 * description: create a board structure, allocating the needed memory
 */
void create_board();

/*
 * description: initialize a board using a file or
 *
 * argument: filename [char*] - name of the txt file containing the initial
    board configuration. if it is NULL, default configuration is used
 * returns: int - number of free places available for the players
 */
int init_board(char* filename);

/*
 * description: delete and free the memory of the board structure
 */
void delete_board();

/*
 * description: determine if there is room for more players
 *
 * returns: true if there is still room for a new player and its fruits
 * returns: false otherwise
 */
bool has_room();

/*
```

```
 * description: generate a random position and place a character there
 *
 * argument: p [place*] - place to fill new coordinates
 * argument: position [int*] - coordinates of the position returned by
    reference
 */
void random_position(place* p, int position[2]);

/*
 * description: send the board structure to a player
 *
 * argument: fd [int] - file descriptor of the client
 * returns: 1 if successful
 * returns: -1 otherwise
 */
int send_board(int fd);

/*
 * description: get the pointer to the place with the given coordinates
 *
 * argument: position [int[2]] - coordinates of the desired place
 * returns: place* - the pointer to the place
 */
place* get_place(int position[2]);

/*
 * description: place a character in a random position on the board
 *
 * argument: p [place*] - place is returned by reference. cannot be NULL
 * argument: id [int] - character's id
 * argument: color [int[3]] - character's color
 * argument: c [character] - character's type
 * argument: position [int[2]] - position of the character returned by
    reference
 */
void random_character(place* p, int id, int color[3], character c, int
   position[2]);

/*
 * description: handle the logic of a client's request
 *
 * argument: m [message] - player's move
 * argument: id [int] - player's id
 */
void handle_request(message m, int id);

/*
 * description: delete a place
 *
 * argument: position [int[2]] - coordinates for the place to be deletes
 * argument: id [int] - id of the character to be deleted
 * argument: type [character] - type of character to be deleted
 * returns: true if the deletition was successful
 * returns: false otherwise, that is, the player with this id and type has
    moved while calling or executing the function
 */
bool delete_place(int position[2], int id, character type);

/*
 * description: get the number of columns
 *
 * returns: int - number of columns
 */
```

```
int get_columns();

/*
 * description: get the number of rows
 *
 * returns: int - number of rows
 */
int get_rows();

/*
 * description: render the swap of two characters
 *
 * argument: m [message] - already contains the message corresponding to
    the moving character, to be printed and sent
 * argument: new_place [place] - place now occupied by the moving character
 * argument: old_pos [int[2]] - final position of the moving character
 */
void draw_swap(message m, place new_place, int old_pos[2]);

/*
 * description: clears a position graphically, in the server and in the
    clients
 *
 * argument: pos [int[2]] - position to be cleared
 */
void clear_position(int pos[2]);
```

## 2.3 Fruits module

The fruits module manages the fruits array and has every function related to this structure. The function declarations are done in `fruits.h` and their implementation in `fruits.c`.

```
/*
 * description: initialize the array of fruits
 *
 * argument: size [int] - maximum number of fruits
 */
void init_fruits(int size);

/*
 * description: create a fruit
 *
 * argument: type [character] - either LEMON or CHERRY
 * argument: id [int*] - fruit id, passed by reference
 * returns: fruit* - newly created fruit
 */
fruit* create_fruit(character type, int* id);

/*
 * description: thread responsible to manage a fruit. waits for fruit to be
    eaten, then waits for two seconds, renders the fruit again and starts
    over
 *
 * argument: arg [character*] - type of the fruit
 * returns: void* - NULL
 */
void* thread_fruit(void* arg);

/*
```

```
 * description: to call when a character eats a fruit. tells fruit the
    thread to sleep
 *
 * argument: id [int] - id of the eaten fruit
 */
void eat_fruit(int id);

/*
 * description: close and deletes every fruit semaphore
 */
void delete_fruits();

/*
 * description: delete two fruits if there are any
 */
void delete_two_fruits();
```

## 2.4 Players module

The players module manages the players array and has every function related to this structure. The function declarations are done in `players.h` and their implementation in `players.c`.

```
/*
 * description: initialize player list
 *
 * argument: max_players [int] - max number of players
 * argument: event [Uint32] - event to be used by SDLPushEvent
 */
void init_players(int max_players, Uint32 event);

/*
 * description: insert a new player into the
 *
 * argument: fd [int] - file descriptor of the player's socket
 * returns: int - new player's id
 */
int insert_player(int fd);

/*
 * description: send message to every player
 *
 * argument: msg [message] - message to be sent
 */
void send_messages(message msg);

/*
 * description: get the position of the "id" player's monster
 *
 * argument: id [int] - player's id
 * returns: int* - position of the monster. might be undefined if the
    monster is not placed or the player does not exist or NULL if id is
    greater than n_players
 * returns: NULL if id > max number of players
 */
int* get_monster(int id);

/*
 * description: get the position of the "id" player's pacman
 *
```

```
 * argument: id [int] - player's id
 * returns: int* - position of the monster. might be undefined if the
    pacman is not placed or the player does not exist or NULL if id is
    greater than n_players
 * returns: NULL if id > max number of players
 */
int* get_pacman(int id);

/*
 * description: set a player's monster position
 *
 * argument: id [int] - player's id
 * argument: position [int[2]] - position where the monster was placed
 */
void set_monster(int id, int* position);

/*
 * description: set a player's pacman position
 *
 * argument: id [int] - player's id
 * argument: position [int[2]] - position where the pacman was placed
 */
void set_pacman(int id, int* position);

/*
 * description: calls one of the previous two functions, depending on the
    character
 *
 * argument: type [character] - either MONSTER, PACMAN or POWER
 * argument: id [int] - player's id
 * argument: position [int*] - position where the character was placed
 */
void set_character(character type, int id, int pos[2]);

/*
 * description: draw a character and send it to every player
 *
 * argument: msg [message] - contains the drawing information
 */
void draw_character(message msg);

/*
 * description: initialize a player
 *
 * argument: fd [int] - file descriptor of the player's socket
 * argument: pac [place*] - empty place structure passed by reference
 * argument: mon [place*] - empty place structure passed by reference
 * returns: int - player's id
 */
int init_player(int fd, place* pac, place* mon);

/*
 * description: delete a player
 *
 * argument: id [int] - player's id
 */
void delete_player(int id);

/*
 * description: delete every player and free the memory
 */
void delete_players();
```

```
/*
 * description: thread to handle user requests
 *
 * argument: arg [int*] - player's socket file descriptor
 * returns: void* - NULL
 */
void* thread_user(void* arg);

/*
 * description: thread to handle each player's character inactivity
 *
 * argument: args [conditional*] - structure with information regarding the
    conditional variable handling and the character's place pointer
 * returns: void* - NULL
 */
void* thread_inactive(void* args);

/*
 * description: return the number of players
 *
 * returns: int - number of players in the game
 */
int get_n_players();

/*
 * description: thread to periodically send the scoreboard to every player
 *
 * argument: arg [void*] - NULL
 * returns: void* - NULL
 */
void* thread_scoreboard(void* arg);

/*
 * description: increase a player's monster or pacman score
 *
 * argument: id [int] - player's id
 * argument: type [character] - character's type, either MONSTER, PACMAN or
    POWER
 */
void increase_score(int id, character type);
```

## 2.5 UI_library module

This module has the required functions to handle the graphical interface. The function declarations are done in `UI_library.h` and their implementation in `UI_library.c`.

```
/*
 * description: get the board place based on the mouse position
 *
 * argument: mouse_x [int] - mouse x position on the window
 * argument: mouse_y [int] - mouse y position on the window
 * argument: board_x [int*] - board x position returned by reference
 * argument: board_y [int*] - board y position returned by reference
 */
void get_board_place(int mouse_x, int mouse_y, int *board_x, int *board_y);

/*
 * description: create a board window
 *
 * argument: dim_x [int] - width of the board
```

```
 * argument: dim_y [int] - height of the board
 * returns: int - always 0
 */
int create_board_window(int dim_x, int dim_y);

/*
 * description: close the board windows
 */
void close_board_windows();

/*
 * description: paint colored pacman
 *
 * argument: board_x [int] - board x position to paint pacman
 * argument: board_y [int] - board y position to paint pacman
 * argument: r [int] - color's red component
 * argument: g [int] - color's green component
 * argument: b [int] - color's blue component
 */
void paint_pacman(int board_x, int board_y, int r, int g, int b);

/*
 * description: paint colored superpowered pacman
 *
 * argument: board_x [int] - board x position to paint pacman
 * argument: board_y [int] - board y position to paint pacman
 * argument: r [int] - color's red component
 * argument: g [int] - color's green component
 * argument: b [int] - color's blue component
 */
void paint_powerpacman(int board_x, int board_y, int r, int g, int b);

/*
 * description: paint colored monster
 *
 * argument: board_x [int] - board x position to paint monster
 * argument: board_y [int] - board y position to paint monster
 * argument: r [int] - color's red component
 * argument: g [int] - color's green component
 * argument: b [int] - color's blue component
 */
void paint_monster(int board_x, int board_y, int r, int g, int b);

/*
 * description: paint a lemon
 *
 * argument: board_x [int] - board x position to paint lemon
 * argument: board_y [int] - board y position to paint lemon
 */
void paint_lemon(int board_x, int board_y);

/*
 * description: paint a cherry
 *
 * argument: board_x [int] - board x position to paint cherry
 * argument: board_y [int] - board y position to paint cherry
 */
void paint_cherry(int board_x, int board_y);

/*
 * description: paint a brick
 *
 * argument: board_x [int] - board x position to paint brick
```

```
 * argument: board_y [int] - board y position to paint brick
 */
void paint_brick(int board_x, int board_y);

/*
 * description: clear a place in the board
 *
 * argument: board_x [int] - board x position to clear
 * argument: board_y [int] - board y position to clear
 */
void clear_place(int board_x, int board_y);

/*
 * description: force the window to be rendered
 */
void render();
```

# 3 Data structures

Since it is used by multiple modules, the `character` enumeration is presented before all other in structure 1. Here, every constant is self-explanatory, but it should be noted that `PACMAN` must come first and `POWER` lastly, making it easier to preform a check for normal and superpowered pacmen by simply evaluating `type % POWER == PACMAN`.

```
enum character {
  PACMAN,
  MONSTER,
  BRICK,
  LEMON,
  CHERRY,
  CLEAR,
  POWER,
};
```

Structure 1: `character` enumeration.

Regarding the board module, only one data structure is needed – the `place` (structure 2).

```
struct place {
  character type;
  int id;
  int color[3];
  int kill_count;
};
```

Structure 2: `place` structure.

This structure is allocated to a single character when he is created and its pointer is the one moving around between board positions. This means that the board is a matrix of `place` pointers – `place***`. Hence, the `id` always corresponds to the id of

the character and the `kill_count` to the number of superpowered pacman kills or to `-1` for the fruits.

As for the players module, it contains an array of `player` – structure 3 – where `id` is the socket file descriptor, `pacman` and `monster` are the pacman and monster positions and `pacman_score` and `monster_score` their scores. The true id of the player corresponds to its index in the array.

```
struct player {
  int id;
  int pacman[2];
  int monster[2];
  int pacman_score;
  int monster_score;
};
```

Structure 3: `player` structure.

To implement the character inactivity, a structure with a condition variable, a mutex to allow its usage, the character information and a flag to signal that the thread should be stopped. This is the `conditional` structure, defined in structure 4.

```
struct conditional {
  pthread_mutex_t mutex;
  pthread_cond_t cond;
  place* p;
  bool stop;
}
```

Structure 4: `conditional` structure.

Finally, the fruit module only defines a data structure which contains a semaphore for eating the fruit, its type and a flag to indicate its termination. Note that since the development OS was macOS, the semaphores had to be named, thus the usage of a name as well, which is used for deleting the semaphore. This `fruit` structure is described in structure 5 and is used in an array of pointers, implementing a list – `fruits`.

```
struct fruit {
  char name[100];
  sem_t *semaphore;
  character type;
  bool stop;
};
```

Structure 5: `fruit` structure.

# 4 Communication protocols

During the game, the messages serve multiple different purposes but always have the same structure (structure 6). To differentiate between them, the `score` field is used.

```
struct message {
  int x, y;
  int id;
  character type;
  int color[3];
  int score;
}
```

Structure 6: `message` structure.

During initialisation, the client starts by sending a message containing only its color and receives an id, as well as the width and height of the board in the `id`, `x` and `y` fields, respectively. This message sent by the server should have a `score` equal to `-2`.

From then on, the movement requests made by the client only need to have the `type` as `PACMAN`, `POWER` or `MONSTER` and the `x` or `y` as either `1` or `-1`, depending on which direction and orientation of the client wants the character to move.

As for the messages sent by the server during the game, they are all broadcasts and can either be board updates (`score` equal to `-1`) or score updates (`score` equal to `0` or `1`).

When the server messages relate to the game board, the sctructure is self-explanatory: `x` and `y` are the cordinates, `id` is the id of the moving player `type` is the type of character to be printed, as already explained, and `color` is to color of the character.

On the other hand, for the score related messages, `x` corresponds as the player's id and `y` as the score.

# 5 Validation

**Messages exchanged between processes**

When the client receives an initialisation message, the verification needed is to confirm the type of the message (if it is not `-2`, the messages are broadcasts and should be ignored) and that both the `x` and `y` are positive. If they are not, something must have gone wrong and the process exits. When the message is a game message, its `score` field is checked and if it corresponds to a move, than the `x` and `y` should be possible positions (positive and smaller than the respective dimension) and each element of the `color` array is compared to 0 and 255 and, if it falls out of the range, its value will be one of these values. In both cases, if the client reads 0 bytes or an error occurs, it assumes the server stopped and also exits the process.

When the server reads initialisation messages, he preforms the same color verification as the client. If, however, there is a problem reading the initialisation message, this function fails, returning `-1`, and the player is not inserted. For movement messages, if the absolute value for the sum of `x` and `y` is different than 1, the move is invalid and therefore ignored. The same happens if the type of character is neither `MONSTER`, `PACMAN` or `POWER`. If the server reads 0 bytes or an error occurs, the player is assumed to have left and is therefore cleared from the board and player list, along with two fruits (if he was not the only player left).

#### Messages exchanged between processes

As for the functions, a list of the possible problems and their handlers follows:

- `send_board()` – in case this function fails, returning `-1`, the initialisation is reverted and the error is propagated back to the player thread who ensures the fruits created are deleted and finally exits.
- `delete_place` – this is a very important return value to be checked, since the deletion can fail due to a player movement. As such, if this function returns `false`, the position should be retrieved again and the deletion repeated, until it succeeds or a maximum number of attempts is reached.
- `accept()` – when the server fails to accept a client, the `errno` is checked. If it is equal to `ECONNABORTED`, the socket was closed and the thread returns `NULL`. Otherwise, a more serious problem may have occured and the process is killed.
- `write()` – every write error is ignored except for when the client is connecting, in which case the same happens as if the read had failed.

Finally, every memory allocation is checked. Since this only happens for the board and mutex board in the beginning and for the fruits – a small structure – a failure results in the process exiting.

# 6 Functionalities

In this section, most of the rules, as well as their implementation, are explained.

#### Control of the pacman and monster on the client

As the client should only send the direction in which the character is moving, the monster implementation is straightforward: the arrow keys directly map to the direction. On the other hand, the pacman moves to an adjacent position only when a mouse movement is detected in said board space.

#### Validation of the maximum number of players

Whenever the server accepts a client connection, he checks whether the number of players times 4 (number of pacmen, monsters and fruits, including the new one) plus 2 is greater than the total number of places minus the bricks. If it is not, the connection will be closed.

#### Placement of new players

After the new client has connected and has been initialised, its pacman and monster are randomly placed on the board, by generating a random column and row, locking that place, checking if it is empty and, if so, occupying it. If the place is already occupied, the mutex is unlocked and other coordinates are generated. This process is repeated until an empty place is found.

#### Client disconnect

When the client disconnects, the server reads 0 bytes and proceeds with closing his socket and replacing in the array by a `-1`, meaning that it has already been closed. Then, its pacman and monster are removed from the board, by getting their positions from the players array and deleting them. Since the positions are not protected, they might be wrong when reading. As such, it is necessary to guarantee that the id and character type are right when deleting them and, if not, the position is retrieved again and the process is run again, until it succeeds or a max number of retries have been made. Finally, two fruits are removed, if there are any, and if there is ønly one player, its score is set to `0`.

#### Character movement

Firstly, if the character moves into a brick or out of bounds, he is bounced back. Then, the following rules apply:

- Empty position – if the place is `NULL`, the character's pointer is moved to the new place and the old one is set to `NULL`.
- Against wall – nothing is done, as bounces have already been taken care of.
- Against fruit – the fruit is eaten, as explained later, and the corresponding character's score is increased. There is a score for each character instead of each player so that it is guarded by the board position mutex. If the player eating the fruit is a pacman, he also turns superpowered. In any case, the character moves into the fruit's old place and the character's old place is set to `NULL`.

- Change position – when two characters switch positions, a simple swap of pointers is done.
- Monster eating – the pacman is sent to a random location, the monster's score is increased and the monster moves into the new position. If it was the pacman that moved into the monster, then the last oen does not move.
- Superpowered pacman eating – same thing as when a monster eats, but his kill counter is incremented and, if it reaches 2, the pacman turns normal again.

After every movement, the relevant messages are composed and sent both to the main thread as SDL events and to every client as messages, where they will be pushed as SDL events as well.

Finally, it is also worth mentioning that a character's position is always retrieved and locked in the same way as described for the deletion in the previous functionality and that whenever a character moves, his position is also updated in the players' array.

#### Guarantee of two movements per second maximum

To limit the characters to move at a maximum speed of two places per second, the player thread has a matrix with the times for the last two movements of both of its characters, as well as an index to which is the oldest one. Whenever this thread receives a request, it checks if it has been more than one second since that oldest move. If it has not, nothing happens and the request is discarded. Otherwise, the time and index are updated, the inactivity condition variable (discussed below) is signalled and the request is handled.

#### Fruit management(eating and placement)

Fruits are placed randomly, in the same way that the characters are. Then, they wait for a semaphore to be posted, which will happen when the fruit is eaten. When this happens, a flag is checked to ensure that the fruit was eaten and not deleted. Finally, in case it was eaten, the thread sleeps for two seconds and then randomly places the fruit again.

#### Character inactivity

The character inactivity is implemented using a `timedwait` on a condition variable. By signalling this variable whenever the character moves, it is possible to know that if the wait times out, it can only mean that the character should be removed from the board and placed in some random place.

#### Superpowered pacman

Whenever a pacman eats a fruit, its type is changed to `POWER`. Afterwards, if he kills a monster, his kill counter is incremented, until it reaches two. When this happens, the type goes back to `PACMAN` and the counter is reset.

#### Game score board

To send the score board, another thread is used, consisting of a simple while loop that sleeps for one minute before sending the score boar to every player. To do so, it cycles through every player, sums its monster and pacman scores and send that information to every client. These messages can be distinguished from the rest by looking at the `score` field, which is `-1` if the message is normal, `0` if it is the first score board message (so that the client prints the header) or `1` for the following.

#### Data cleanup

The data cleanup when a client disconnects is straightforward because instead of allocating memory for every structure, most of them are passed by reference by the parent thread. This means that they will be cleaned when the thread stops.

One structure that is constantly being allocated and freed, however, is the SDL event data, which is allocated before being pushed to the main thread and freed after being read. This is also the case for the client application.

One the other hand, there is one exceptions, such as the fruits, that are freed whenever the corresponding thread is signalled to stop.

Finally, the board, mutex board, players array, fruits array and brick structure are freed by the OS when the process terminates. Nonetheless, they are all cleaned when the server is exiting.

# 7 Synchronisation

To prevent memory corruption and undefined variables when having parallel threads modifying the same variable, multiple synchronisation mechanisms were used.

## 7.1 Board module

**`pthread_mutex_t** mutex_board`**

The most important mechanisms are the board mutexes, with one being created for each position. The decision of creating a matrix instead of using one mutex for the entire board was made on the basis that multiple players can be moving at the exact

same time in different, unrelated places. In this case, locking the entire board for each of them would prevent them from moving at the same time, slowing down the game substantially with an increased number of players.

So when are they used? Suppose that two user threads try to move a character to a place at the same time and that no locks are used. In this case, they might verify the occupancy state of the new position at the same time and, based on that assumption, both will move, but only one will in fact occupy the position. In this cases, there is a race condition to the old and new positions that must be avoided.

As to where they are used, it is in any place that access or modifies the board. This means there are three critical regions in `board.c` solved by these mutexes:

- In `random_position()` it is necessary to lock the new position before verifying if it is empty. If it is not, the mutex is unlocked and a new position is generated, cycling through the same steps. Otherwise, the character is placed and only then is the lock released.
- In `send_board()` the lock should be made before checking if the the place is empty and release after getting a copy of its occupant.
- In `handle_request` both the old and the new positions in the board must be locked, which happens between lines 234 and 264 (because of the logic later explain for the deadlock avoidance and to check that the old position is correct). The critical region end when both places have been changed, which depends on the movement, happening throughout the rest of the function.

Notice that the board is a matrix of pointers instead of structure so that the critical regions can be much smaller since the movements are atomic.

To avoid a deadlock, however, it is extremely important that the locks are ordered. Let's suppose that the lock is first made on the origin place and only then on the destination place. If two adjacent characters (A and B) move at the exact same time to the other's position (positions 1 and 2), the following might happen:

1. Player A locks position 1.
2. Player B locks position 2.
3. Player A tries to lock position 2, but it fails.
4. Player B tries to lock position 1, but it fails.

This problem is avoided by making sure that the locks are always done first on the position with the lower x coordinate and, if both are the same, than the first lock is done on the position with the lower y coordinate. The above scenario would then be the following:

1. Player A locks position 1.
2. Player B tries locks position 1, but it fails.
3. Player A locks position 2.
4. Player A executes the logic and unlocks both mutexes.
5. Player B locks position 1.
6. Player B locks position 2.
7. ...

Finally, these mutexes not only protect the character in the board, but also the character in the player's list. This means that when modifying the character's position, it is necessary to still be holding the lock, preventing another thread to win the race and modify the position in advance, leaving an incorrect value. To get the position, this is not an issue, because after locking the board mutex there should be a validation to confirm that the position does indeed correspond to the desired character.

## 7.2 Players module

`pthread_rwlock_t rwlock`

To prevent the player list to be incoherent when sending messages, fetching scores, positions or the number of players, there should be a write lock whenever a new player is added or a player removed (in functions `insert_player()`, `delete_player()` and `delete_players()`), and a read lock surrounding the calls for `send_messages()` and inside the `increase_score()` function.

## 7.3 Fruits module

`pthread_rwlock_t rwlock`

Much like the previous lock, this one is used to make sure that a fruit is not eaten while it is being deleted or that a fruit is not created while another is being deleted, which is mandatory for protecting the variable holding the number of fruits and, since the list is implemented with an array, to ensure that no gaps exist in the array or that no two fruits are created in the same index. To do so, a read lock is done when eating a fruit (`eat_fruit()`) and a write lock whenever fruits are deleted (`delete_two_fruits()`) or created (`create_fruit()`).