CSN6214 Operating Systems

A Report on Tactical Token Race: A Hybrid Concurrent Multiplayer Game Server Architecture

Submitted to:

Mr Ng Hu

Lecture Section: TC3L
Tutorial Section: TT12L

Prepared by Group 4

Name	Student ID
Sathishkugan A/L Thangarajoo	1211103319
Thasayaini A/P Murthy	1211104170
Wan Hanani Iman Binti Wan Mohd Azidi @ Sapawi	242UC244CK
Jasmyne Yap	242UC244PT

Date of Submission: 5 February 2026

Game Description and Rules

1.1. Description

Tactical Token Race is a turn-based multiplayer board game where players compete to reach the final tile of a shared linear board. The game is managed entirely by the server, which enforces all rules, validates moves, and updates the game state. Clients act only as input/output terminals, sending simple commands and receiving updates.

The game supports up to 4 players per session. Each game round follows a deterministic turn order decided through an initial dice roll. Players take turns rolling a virtual die and moving their token forward on the board. The first player to reach the final tile wins the game.

1.2. Setup and Objective

- Players per game: 3–5.
- Host: The first player to connect becomes the Host and chooses the lobby size (3, 4, or 5 players).
- Start positions: All players start at position 0.
- Objective: Be the first player to reach position 30 or higher on the track.

1.3. Turn Order

1.3.1. Once the lobby is full, the server rolls a die for each player to determine the initial order.

1.3.2. Higher rolls act earlier; ties are broken with additional rolls.

1.3.3. The resulting order is stored and used in round robin fashion for the entire game:

- Player 1 → Player 2 → ... → Player N → back to Player 1

1.3.4. At any time, only the current player is allowed to act. The server:

- Announces whose turn it is.
- Sends a “YOUR TURN” message only to that player.
- Waits for their choice before proceeding

1.4. Actions Per Turn

1. **SAFE Move**
   - Server rolls a 6-sided die.
   - Movement based on the roll:
     - Roll 1–2 → move +1 step.
     - Roll 3–4 → move +2 steps.
     - Roll 5–6 → move +3 steps.
   - SAFE moves never move the player backward.
   - Any active Token bonus (see §1.5) is added to this movement.
2. **RISK Move**
   - Server rolls a 6-sided die.
   - Two cases:
     - Success (4–6): move forward by the exact roll (e.g., roll 5 → +5).
     - Failure (1–3): move 1 step backward (−1).SAFE moves never move the player backward.

        - Positions can never go below 0; if a backward move would go negative, the position is set to 0.
        - Any active Token bonus is added to this movement, which can partially or completely cancel the −1 on a failed roll. Any activ SAFE moves never move the player backward.
3. **EXIT Game**
    - The player voluntarily leaves the game.
    - They are immediately marked as inactive and cannot win that game.
    - The game continues with the remaining players.

1.5. Token Mechanic

- Token positions: 5, 10, 15, 20, and 25 on the track.
- When a player lands exactly on one of these positions after a move, they gain a Token.
- A Token provides a +2 step bonus on the next turn only:
    - Example SAFE: base move +2, Token +2 → total +4.
    - Example RISK success: roll 6 → +6, Token +2 → total +8.
    - Example RISK failure: base −1, Token +2 → net +1.
- Example RISK success: roll 6 → +6, Token +2 → total +8.
- Example RISK failure: base −1, Token +2 → net +1.
- After the bonus is applied on the next turn, the Token is consumed and removed.

1.6. Winning, Losing, and Disconnections

Standard Win

- After a move (including any Token bonus), if a player's position is 30 or higher, that player wins immediately.The server announces the winner and the game ends.

Forfeit Win

- If, during a game, all other players leave or disconnect, the last remaining active player wins by forfeit.

Loss

- Any player who is not the winner when the game ends is considered to have lost.
- A player who exits or disconnects before the game ends automatically loses that game.

Host Disconnection

- If the Host disconnects before choosing the lobby size, the lobby is cancelled and all waiting players are removed.
- If the Host disconnects after the game has started, the game continues with the remaining players; the Host's turn slot is simply skipped in the turn order.

Deployment Mode (IPC)

1.1. Mode Overview

- Selected Mode: Single-Machine Mode using IPC named pipes (FIFOs).
- Environment: Linux

1.2. Rationale for IPC Choice

- Easier to manage within a single host lab environment.
- Directly reinforces OS concepts:
    - POSIX named pipes
    - Shared memory via mmap
    - Process-shared mutexes and semaphores
- No network configuration overhead (ports, firewall).

Hybrid Architecture

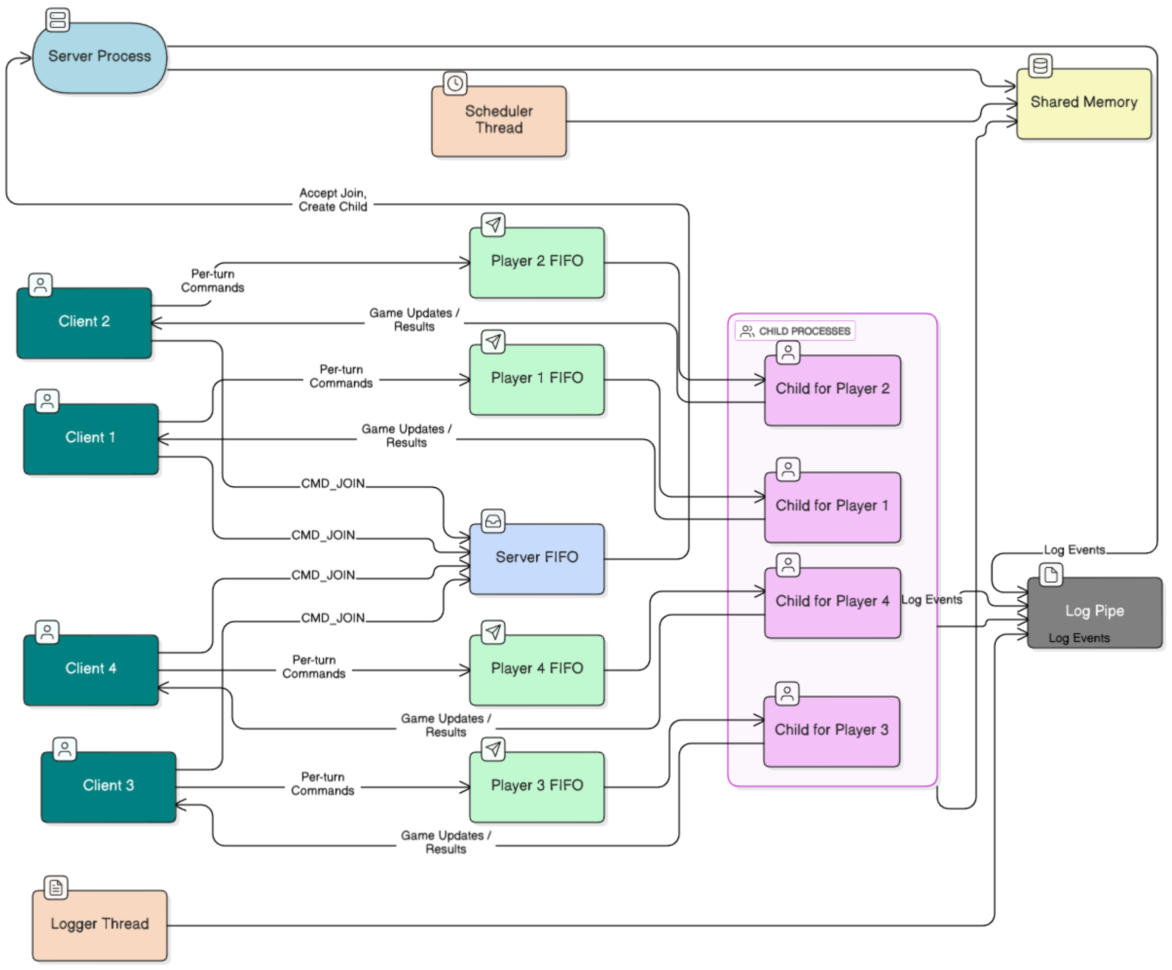


Figure 1 Hybrid Architecture Diagram

Figure 1 Hybrid Architecture Diagram shows the hybrid architecture of Tactical Token Race. The server runs as a parent process that creates a Scheduler thread and a Logger thread and forks one child process per connected player. All server processes and threads share a common

game_state_t structure in POSIX shared memory. Clients communicate with the server via named pipes: a shared SERVER_FIFO for join requests and per-player FIFOs for per-turn commands and game updates. All processes write log messages to a pipe consumed by the Logger thread, which serializes them into data/game.log.

IPC Mechanism and Shared Memory Layout

In this project, Inter-Process Communication (IPC) is required so the main controller process and multiple terminal processes can exchange information. Since each terminal runs as a separate process with its own unique PID, IPC ensures they communicate in a consistent and controlled manner.

IPC Mechanism Used

The project uses shared memory because it allows fast, direct data exchange between processes without needing to copy data through pipes or files. Shared memory lets all processes access the same memory region simultaneously. To prevent conflicts when multiple processes try to read or write at the same time, semaphores are used to synchronize access. This avoids race conditions and ensures data integrity.

Communication Flow

- The main process creates the shared memory and initializes the semaphore.
- Each child process attaches to the shared memory when it starts.
- Terminals write commands or updates into shared memory.
- The main process monitors the shared region and reacts to the commands.

Shared Memory Layout (Simplified)

Field	Description
active_terminal_count	Number of running terminals
last_command_pid	PID of the terminal that sent the latest command
command_buffer	Stores the most recent command
message_buffer	General message are

Synchronization Strategy

1.1. Process-Shared Mutex

- game_mutex is initialized with PTHREAD_PROCESS_SHARED.
- It protects all modifications to shared game state:
  - Joining/leaving players
  - Updating positions, tokens, winner
  - Changing players_connected, lobby_ready, target_players

- Reading/updating leaderboard

1.2. Semaphores

1.2.1. Per-player semaphores turn_sems[]:

- Child blocks on its entry.
- Scheduler posts when it is that player's turn, or when the game ends and it needs to send final status.

1.2.2. Scheduler semaphore scheduler_sem:

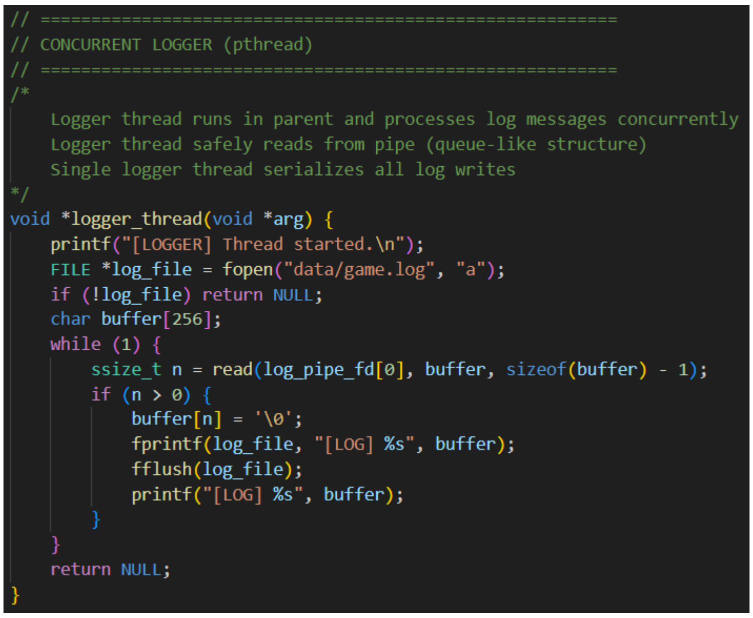
- Child posts when it has finished processing its turn or when a disconnect occurs.
- Scheduler waits on this to know when it can safely choose the next player and modify turn state.

1.3. Critical Sections and Invariants

- All code sections that modify shared data are inside pthread_mutex_lock(&game->game_mutex) / unlock.
- Invariants maintained, e.g.:
  - 0 <= players_connected <= MAX_PLAYERS
  - winner_id == -1 except when a winner is declared
  - active[i] == 1 only if the player slot is valid and in use

Logger Design

The logging system is designed to be concurrent, non-blocking, and thread-safe, ensuring that game events are recorded chronologically without interrupting the flow of gameplay.



```c
// ==========================================================
// CONCURRENT LOGGER (pthread)
// ==========================================================
/*
    Logger thread runs in parent and processes log messages concurrently
    Logger thread safely reads from pipe (queue-like structure)
    Single logger thread serializes all log writes
*/
void *logger_thread(void *arg) {
    printf("[LOGGER] Thread started.\n");
    FILE *log_file = fopen("data/game.log", "a");
    if (!log_file) return NULL;
    char buffer[256];
    while (1) {
        ssize_t n = read(log_pipe_fd[0], buffer, sizeof(buffer) - 1);
        if (n > 0) {
            buffer[n] = '\0';
            fprintf(log_file, "[LOG] %s", buffer);
            fflush(log_file);
            printf("[LOG] %s", buffer);
        }
    }
    return NULL;
}
```

1.1 Architecture

The logger runs as a dedicated thread (logger_thread) within the parent server process. It operates on a Producer-Consumer model where the server and child processes act as producers, and the logger thread acts as the single consumer.

1.2 Communication Mechanism (Pipe)

The system uses a standard UNIX unnamed pipe (log_pipe_fd) for communication.

- **Producers:** The Scheduler thread and all forked Child Processes write log messages to the write-end of the pipe (log_pipe_fd[1]). Because the size of the log messages is well within the system's PIPE_BUF limit, these writes are atomic, preventing interleaved (garbled) output from different processes.
- **Consumers:** The logger_thread sits in a continuous loop reading from the read-end of the pipe (log_pipe_fd[0]).

1.3 File I/O

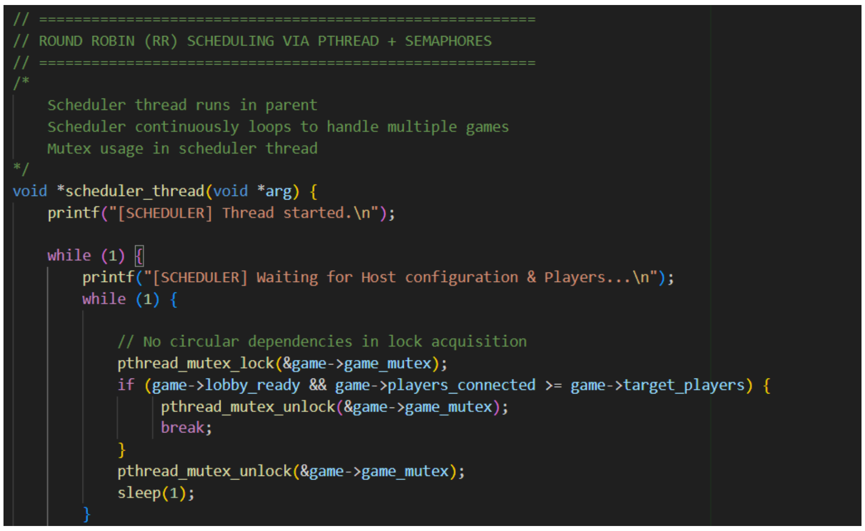
Upon reading a message from the pipe, the logger thread appends it to **data/game.log** and flushes the stream immediately to ensure real-time persistence.

1.4 Concurrency Safety

By decoupling file I/O from the game logic, the child processes do not need to lock the log file themselves. They simply write to the pipe and continue execution immediately, preventing I/O bottlenecks from slowing down the game.

Round Robin (RR) Scheduler

The Round Robin Scheduler is implemented as a dedicated thread (scheduler_thread) in the parent process, responsible for orchestrating the flow of the game and managing turn states across isolated child processes.



```c
// ========================================================
// ROUND ROBIN (RR) SCHEDULING VIA PTHREAD + SEMAPHORES
// ========================================================
/*
    Scheduler thread runs in parent
    Scheduler continuously loops to handle multiple games
    Mutex usage in scheduler thread
*/
void *scheduler_thread(void *arg) {
    printf("[SCHEDULER] Thread started.\n");

    while (1) {
        printf("[SCHEDULER] Waiting for Host configuration & Players...\n");
        while (1) {

            // No circular dependencies in lock acquisition
            pthread_mutex_lock(&game->game_mutex);
            if (game->lobby_ready && game->players_connected >= game->target_players) {
                pthread_mutex_unlock(&game->game_mutex);
                break;
            }
            pthread_mutex_unlock(&game->game_mutex);
            sleep(1);
        }
```

1.1 Turn Order Determination

- At the start of a session, the scheduler performs a "Roll for Initiative." It generates random values for each player, sorts them to determine the turn_order array, and handles ties by triggering re-rolls.
- This order is stored in shared memory (game->turn_order[]) so it remains consistent throughout the match.

1.2 Turn Cycling Logic

- The scheduler maintains an index (turn_array_index) to track the current player.
- It uses a Request-Reply synchronization pattern using POSIX semaphores:
  1. **Granting the Turn:** The scheduler signals the specific semaphore for the current player (sem_post(&game->turn_sems[id])). This wakes up the corresponding child process, which is blocked waiting for this signal.
  2. **Waiting for Completion:** The scheduler then blocks on a separate semaphore (sem_wait(&game->scheduler_sem)), waiting for the child process to finish its move.

1.3 Inactive Player Handling

Before granting a turn, the scheduler checks the game->active[id] flag in shared memory. If a player has disconnected (flag is 0), the scheduler automatically increments the turn index to skip them, ensuring the game continues uninterrupted for remaining players.

1.4 Synchronization

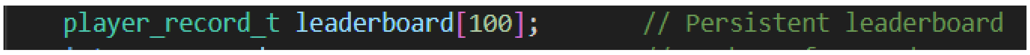
All access to shared turn state variables is protected by the process-shared mutex (game->game_mutex) to prevent race conditions between the scheduler and child processes.

Persistence Strategy

The server implements a persistent scoring system that tracks player victories across server restarts using a flat-file database approach.

1.1 Data Structures

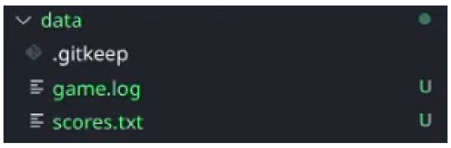
- Player statistics are loaded into a fixed-size array (leaderboard[100]) located in Shared Memory. This allows all child processes to read player history instantly without disk I/O during the game.
- The structure player_record_t holds the player's nickname and their total win count.



```
    player_record_t leaderboard[100];       // Persistent leaderboard
```

1.2 File Format

- Data is stored in data/scores.txt in a simple text format: Nickname Wins (e.g., ALEX 5).

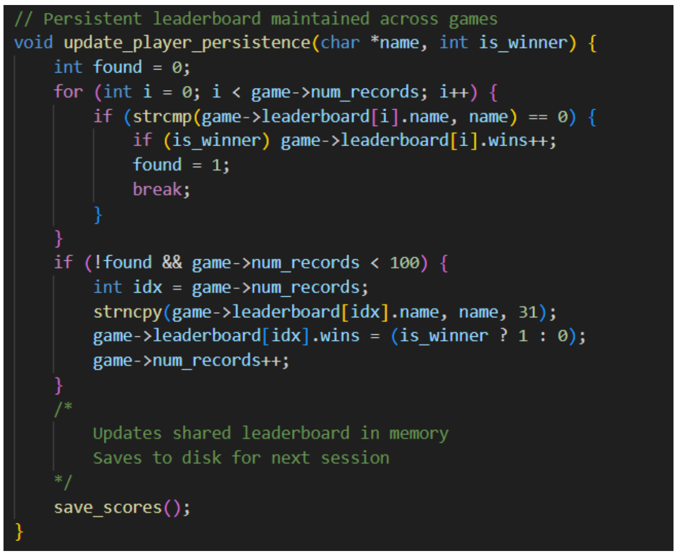


```
v data
   .gitkeep
   game.log      U
   scores.txt    U
```

1.3 Loading and Saving Mechanism

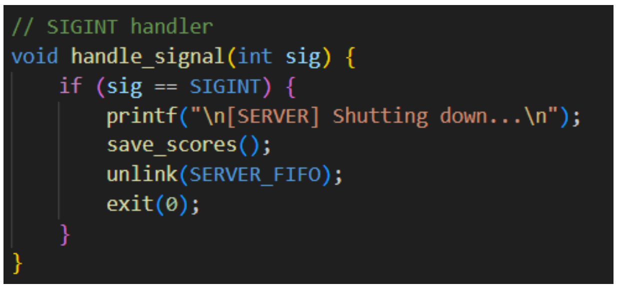
- **Startup:** When the server initializes, it calls load_scores(), which reads the text file and populates the shared memory structure.

- **Atomic Updates:** When a player wins, the server calls update_player_persistence(). This function updates the in-memory leaderboard and immediately calls save_scores() to write the entire array back to disk.



```c
// Persistent leaderboard maintained across games
void update_player_persistence(char *name, int is_winner) {
    int found = 0;
    for (int i = 0; i < game->num_records; i++) {
        if (strcmp(game->leaderboard[i].name, name) == 0) {
            if (is_winner) game->leaderboard[i].wins++;
            found = 1;
            break;
        }
    }
    if (!found && game->num_records < 100) {
        int idx = game->num_records;
        strncpy(game->leaderboard[idx].name, name, 31);
        game->leaderboard[idx].wins = (is_winner ? 1 : 0);
        game->num_records++;
    }
    /*
        Updates shared leaderboard in memory
        Saves to disk for next session
    */
    save_scores();
}
```

- **Shutdown:** A signal handler for SIGINT (Ctrl+C) ensures that save_scores() is called one last time before the server terminates, preventing data loss during manual shutdowns.



```c
// SIGINT handler
void handle_signal(int sig) {
    if (sig == SIGINT) {
        printf("\n[SERVER] Shutting down...\n");
        save_scores();
        unlink(SERVER_FIFO);
        exit(0);
    }
}
```

1.4 Synchronization

Because multiple child processes (handling different players) could theoretically trigger game-end conditions simultaneously, the update_player_persistence() function is guarded by the global game->game_mutex. This ensures that file writes and memory updates are mutually exclusive.

Multi-Game Handling

The scheduler_thread() function runs an infinite outer loop. The reset process starts by clearing all players states such as positions, token flags, and all players are marked inactive. The lobby is then configured to it's initial state :

```
game->players_connected = 0;
game->target_players = MAX_PLAYERS;
game->lobby_ready = 0;
game->winner_id = -1;
game->game_running = 1;
```

This line of code :

```
pthread_mutex_unlock(&game->game_mutex);
break;
```

Unlocks the mutex so other threads can access shared memory. 'break' exits the current game session loop and returns to the scheduler's outer loop that waits for players in the new lobby. There are 2 ways in which a game could end and the session restarts :

i. When all players leave the game

```
// If everyone left, RESET
if (game->players_connected == 0) {
printf("[SCHEDULER] All players left. Resetting...\n");
// Reset Session
for(int i=0; i<MAX_PLAYERS; i++) {
game->positions[i] = 0;
game->token_flag[i] = 0;
game->active[i] = 0;
sem_init(&game->turn_sems[i], 1, 0);
}
game->players_connected = 0;
game->target_players = MAX_PLAYERS;
game->lobby_ready = 0;
game->winner_id = -1;
game->game_running = 1;

pthread_mutex_unlock(&game->game_mutex);
break; // Go back to lobby wait
}
```

This occurs when all players disconnect from the game, and the scheduler detects players_connected ==0. All game states will reset to initial values. 'Break' exits the inner game loop and returns to the outer loop.

ii. When a game names a winner

```
pthread_mutex_lock(&game->game_mutex);
for(int i=0; i<MAX_PLAYERS; i++) {
game->positions[i] = 0;
game->token_flag[i] = 0;
game->active[i] = 0;
sem_init(&game->turn_sems[i], 1, 0);
}
game->players_connected = 0;
game->target_players = MAX_PLAYERS;
game->lobby_ready = 0;
game->winner_id = -1;
```

```c
game->game_running = 1;
printf("[SCHEDULER] Session Ended. Lobby Reset.\n");
pthread_mutex_unlock(&game->game_mutex);
break;}
```

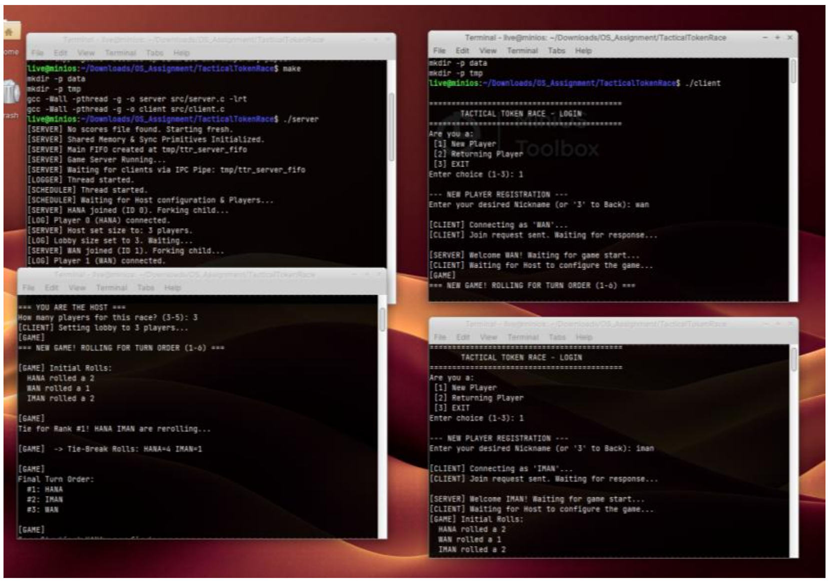
A winner is named when a player reaches or exceeds the target score which is 30. The winner_id is set to that player's name

iii. Host leaves during setup

```c
if (bytes <= 0) {
pthread_mutex_lock(&game->game_mutex);
printf("[SERVER] Host disconnected during setup (EOF).\n");
// Mark Host as gone
game->active[0] = 0;
game->players_connected--;
game->player_pids[0] = 0;
memset(game->player_names[0], 0, 32);
// Nuke the lobby for everyone else
printf("[SERVER] Aborting lobby. Kicking %d waiting players...\n", game->players_connected);
for(int i=1; i<MAX_PLAYERS; i++) {
if (game->active[i]) {
char target_fifo[64];
snprintf(target_fifo, sizeof(target_fifo), CLIENT_FIFO_TEMPLATE, game->player_pids[i]);
int t_fd = open(target_fifo, O_WRONLY | O_NONBLOCK);
if (t_fd != -1) {
ipc_msg_t kill_msg;
kill_msg.command = MSG_LOSE;
snprintf(kill_msg.payload, sizeof(kill_msg.payload), "Host disconnected. Lobby Closed.");
write(t_fd, &kill_msg, sizeof(kill_msg));
close(t_fd);}
game->active[i] = 0;
game->players_connected--;}}
```

Testing Evidence

1. Starting a game session by initializing the server (top left), and registering the host and players



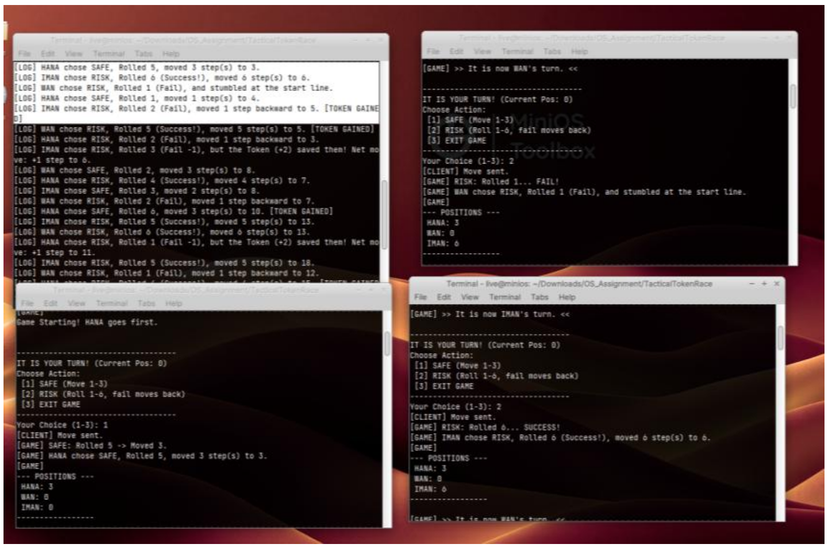
```
live@minios:~/Downloads/OS_Assignment/TacticalTokenRace$ make
mkdir -p data
mkdir -p tmp
gcc -Wall -pthread -g -o server src/server.c -lrt
gcc -Wall -pthread -g -o client src/client.c
live@minios:~/Downloads/OS_Assignment/TacticalTokenRace$ ./server
[SERVER] No scores file found. Starting fresh.
[SERVER] Shared Memory & Sync Primitives Initialized.
[SERVER] Main FIFO created at tmp/ttr_server_fifo
[SERVER] Game Server Running...
[SERVER] Waiting for clients via IPC Pipe: tmp/ttr_server_fifo
[LOGGER] Thread started.
[SCHEDULER] Thread started.
[SCHEDULER] Waiting for Host configuration & Players...
[SERVER] HANA joined (ID 0). Forking child...
[LOG] Player 0 (HANA) connected.
[SERVER] Host set size to: 3 players.
[LOG] Lobby size set to 3. Waiting...
[SERVER] WAN joined (ID 1). Forking child...
[LOG] Player 1 (WAN) connected.
```

```
=== YOU ARE THE HOST ===
How many players for this race? (3-5): 3
[CLIENT] Setting lobby to 3 players...
[GAME]
=== NEW GAME! ROLLING FOR TURN ORDER (1-6) ===

[GAME] Initial Rolls:
  HANA rolled a 2
  WAN rolled a 1
  IMAN rolled a 2

[GAME]
Tie for Rank #1! HANA IMAN are rerolling...

[GAME]  -> Tie-Break Rolls: HANA=4 IMAN=1

[GAME]
Final Turn Order:
  #1: HANA
  #2: IMAN
  #3: WAN

[GAME]
```

```
mkdir -p data
mkdir -p tmp
live@minios:~/Downloads/OS_Assignment/TacticalTokenRace$ ./client

==========================================
      TACTICAL TOKEN RACE - LOGIN
==========================================
Are you a:
 [1] New Player
 [2] Returning Player
 [3] EXIT
Enter choice (1-3): 1

--- NEW PLAYER REGISTRATION ---
Enter your desired Nickname (or '3' to Back): wan

[CLIENT] Connecting as 'WAN'...
[CLIENT] Join request sent. Waiting for response...

[SERVER] Welcome WAN! Waiting for game start...
[CLIENT] Waiting for Host to configure the game...
[GAME]
=== NEW GAME! ROLLING FOR TURN ORDER (1-6) ===
```

```
==========================================
      TACTICAL TOKEN RACE - LOGIN
==========================================
Are you a:
 [1] New Player
 [2] Returning Player
 [3] EXIT
Enter choice (1-3): 1

--- NEW PLAYER REGISTRATION ---
Enter your desired Nickname (or '3' to Back): iman

[CLIENT] Connecting as 'IMAN'...
[CLIENT] Join request sent. Waiting for response...

[SERVER] Welcome IMAN! Waiting for game start...
[CLIENT] Waiting for Host to configure the game...
[GAME] Initial Rolls:
  HANA rolled a 2
  WAN rolled a 1
  IMAN rolled a 2
```

2. Gameplay and game logging tracked in the server terminal



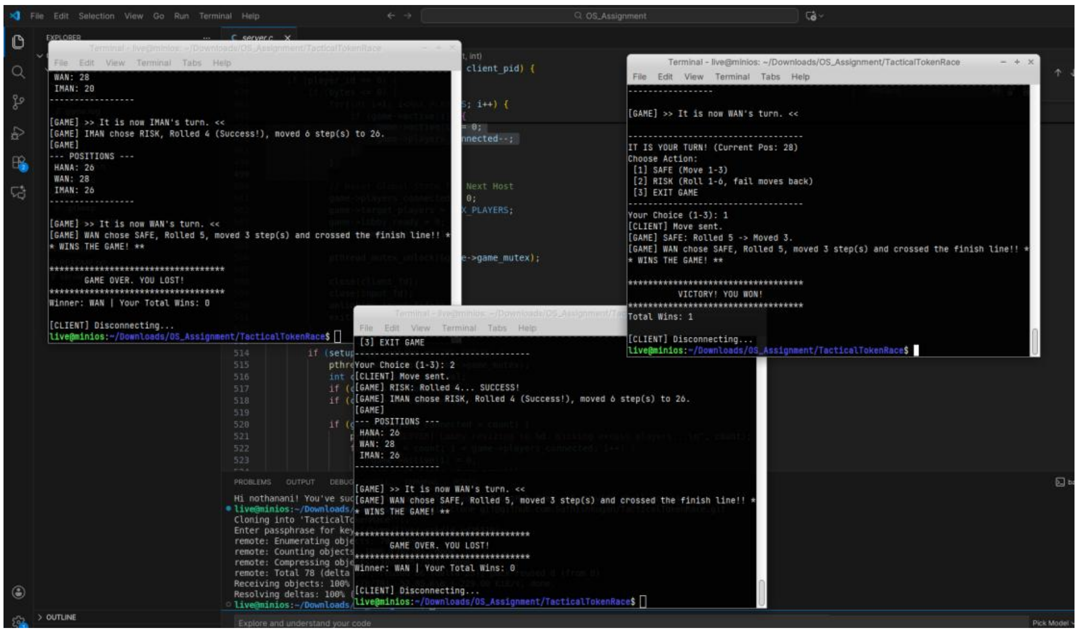
```
[LOG] HANA chose SAFE, Rolled 5, moved 3 step(s) to 3.
[LOG] IMAN chose RISK, Rolled 6 (Success!), moved 6 step(s) to 6.
[LOG] WAN chose RISK, Rolled 1 (Fail), and stumbled at the start line.
[LOG] HANA chose SAFE, Rolled 1, moved 1 step(s) to 4.
[LOG] IMAN chose RISK, Rolled 2 (Fail), moved 1 step backward to 5. [TOKEN GAINE
D]
[LOG] WAN chose RISK, Rolled 5 (Success!), moved 5 step(s) to 5. [TOKEN GAINED]
[LOG] HANA chose RISK, Rolled 2 (Fail), moved 1 step backward to 3.
[LOG] IMAN chose RISK, Rolled 3 (Fail -1), but the Token (+2) saved them! Net mo
ve: +1 step to 6.
[LOG] WAN chose SAFE, Rolled 2, moved 3 step(s) to 8.
[LOG] HANA chose RISK, Rolled 4 (Success!), moved 4 step(s) to 7.
[LOG] IMAN chose SAFE, Rolled 3, moved 2 step(s) to 8.
[LOG] WAN chose RISK, Rolled 2 (Fail), moved 1 step backward to 7.
[LOG] HANA chose SAFE, Rolled 6, moved 3 step(s) to 10. [TOKEN GAINED]
[LOG] IMAN chose RISK, Rolled 5 (Success!), moved 5 step(s) to 13.
[LOG] WAN chose RISK, Rolled 6 (Success!), moved 6 step(s) to 13.
[LOG] HANA chose RISK, Rolled 1 (Fail -1), but the Token (+2) saved them! Net mo
ve: +1 step to 11.
[LOG] IMAN chose RISK, Rolled 5 (Success!), moved 5 step(s) to 18.
[LOG] WAN chose RISK, Rolled 1 (Fail), moved 1 step backward to 12.
```

```
Game Starting! HANA goes first.


-----------------------------------
IT IS YOUR TURN! (Current Pos: 0)
Choose Action:
 [1] SAFE (Move 1-3)
 [2] RISK (Roll 1-6, fail moves back)
 [3] EXIT GAME
-----------------------------------
Your Choice (1-3): 1
[CLIENT] Move sent.
[GAME] SAFE: Rolled 5 -> Moved 3.
[GAME] HANA chose SAFE, Rolled 5, moved 3 step(s) to 3.
[GAME]
--- POSITIONS ---
 HANA: 3
 WAN: 0
 IMAN: 0
----------------
```

```
[GAME] >> It is now WAN's turn. <<


-----------------------------------
IT IS YOUR TURN! (Current Pos: 0)
Choose Action:
 [1] SAFE (Move 1-3)
 [2] RISK (Roll 1-6, fail moves back)
 [3] EXIT GAME
-----------------------------------
Your Choice (1-3): 2
[CLIENT] Move sent.
[GAME] RISK: Rolled 1... FAIL!
[GAME] WAN chose RISK, Rolled 1 (Fail), and stumbled at the start line.
[GAME]
--- POSITIONS ---
 HANA: 3
 WAN: 0
 IMAN: 6
----------------
```

```
[GAME] >> It is now IMAN's turn. <<


-----------------------------------
IT IS YOUR TURN! (Current Pos: 0)
Choose Action:
 [1] SAFE (Move 1-3)
 [2] RISK (Roll 1-6, fail moves back)
 [3] EXIT GAME
-----------------------------------
Your Choice (1-3): 2
[CLIENT] Move sent.
[GAME] RISK: Rolled 6... SUCCESS!
[GAME] IMAN chose RISK, Rolled 6 (Success!), moved 6 step(s) to 6.
[GAME]
--- POSITIONS ---
 HANA: 3
 WAN: 0
 IMAN: 6
----------------
```

3. Winner is named (Wan), and server starts again



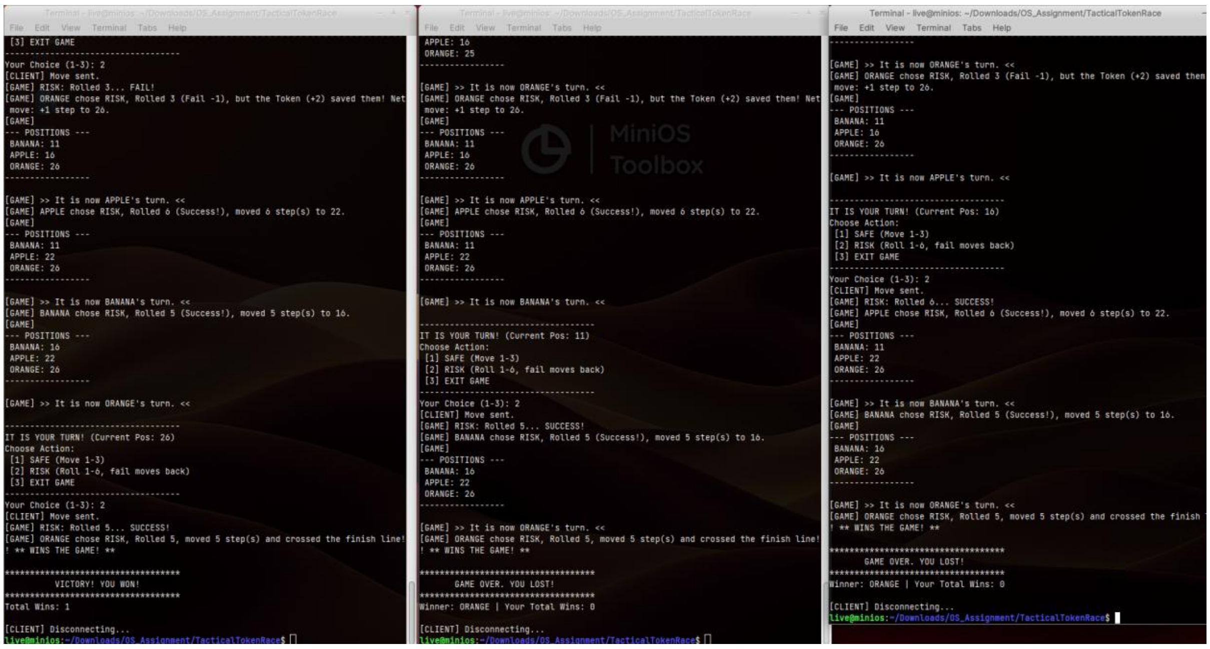
```
 WAN: 28
 IMAN: 20
----------------

[GAME] >> It is now IMAN's turn. <<
[GAME] IMAN chose RISK, Rolled 4 (Success!), moved 6 step(s) to 26.
[GAME]
--- POSITIONS ---
 HANA: 26
 WAN: 28
 IMAN: 26
----------------

[GAME] >> It is now WAN's turn. <<
[GAME] WAN chose SAFE, Rolled 5, moved 3 step(s) and crossed the finish line!! *
* WINS THE GAME! **

***********************************
      GAME OVER. YOU LOST!
***********************************
Winner: WAN | Your Total Wins: 0

[CLIENT] Disconnecting...
live@minios:~/Downloads/OS_Assignment/TacticalTokenRace$
```

```
----------------

[GAME] >> It is now WAN's turn. <<


-----------------------------------
IT IS YOUR TURN! (Current Pos: 28)
Choose Action:
 [1] SAFE (Move 1-3)
 [2] RISK (Roll 1-6, fail moves back)
 [3] EXIT GAME
-----------------------------------
Your Choice (1-3): 1
[CLIENT] Move sent.
[GAME] SAFE: Rolled 5 -> Moved 3.
[GAME] WAN chose SAFE, Rolled 5, moved 3 step(s) and crossed the finish line!! *
* WINS THE GAME! **

***********************************
          VICTORY! YOU WON!
***********************************
Total Wins: 1

[CLIENT] Disconnecting...
live@minios:~/Downloads/OS_Assignment/TacticalTokenRace$
```

```
 [3] EXIT GAME
-----------------------------------
Your Choice (1-3): 2
[CLIENT] Move sent.
[GAME] RISK: Rolled 4... SUCCESS!
[GAME] IMAN chose RISK, Rolled 4 (Success!), moved 6 step(s) to 26.
[GAME]
--- POSITIONS ---
 HANA: 26
 WAN: 28
 IMAN: 26
----------------

[GAME] >> It is now WAN's turn. <<
[GAME] WAN chose SAFE, Rolled 5, moved 3 step(s) and crossed the finish line!! *
* WINS THE GAME! **

***********************************
      GAME OVER. YOU LOST!
***********************************
Winner: WAN | Your Total Wins: 0

[CLIENT] Disconnecting...
live@minios:~/Downloads/OS_Assignment/TacticalTokenRace$
```

Screenshots

Client window: Orange, Banana, Apple (left to right)



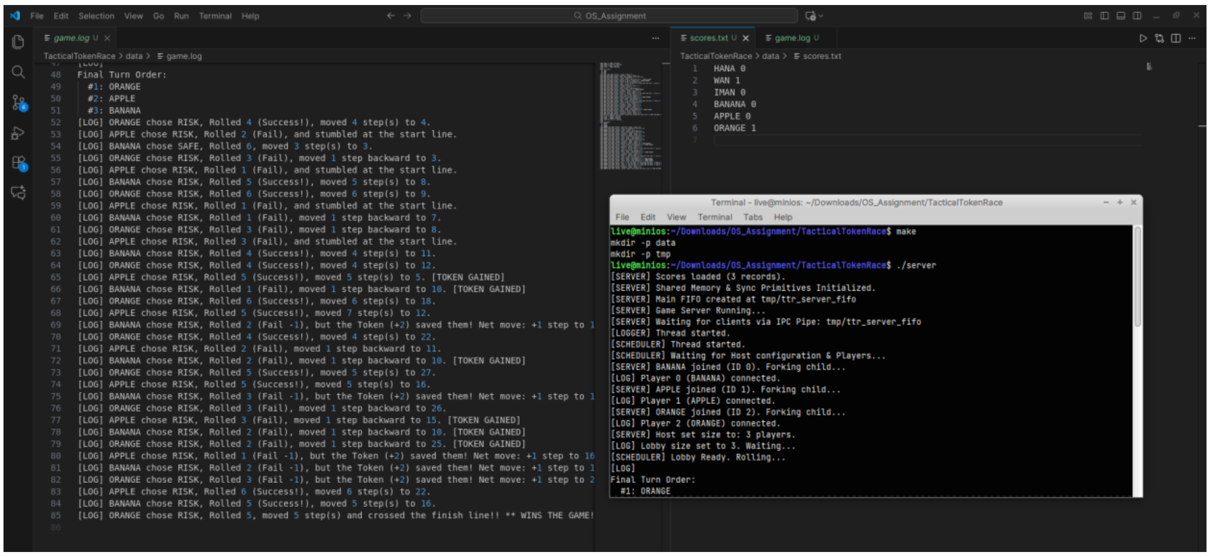
Terminal - live@minios: ~/Downloads/OS_Assignment/TacticalTokenRace
File Edit View Terminal Tabs Help
MiniOS Toolbox

```
 [3] EXIT GAME
-----------------------------------
Your Choice (1-3): 2
[CLIENT] Move sent.
[GAME] RISK: Rolled 3... FAIL!
[GAME] ORANGE chose RISK, Rolled 3 (Fail -1), but the Token (+2) saved them! Net
 move: +1 step to 26.
[GAME]
--- POSITIONS ---
 BANANA: 11
 APPLE: 16
 ORANGE: 26
-----------------

[GAME] >> It is now APPLE's turn. <<
[GAME] APPLE chose RISK, Rolled 6 (Success!), moved 6 step(s) to 22.
[GAME]
--- POSITIONS ---
 BANANA: 11
 APPLE: 22
 ORANGE: 26
-----------------

[GAME] >> It is now BANANA's turn. <<
[GAME] BANANA chose RISK, Rolled 5 (Success!), moved 5 step(s) to 16.
[GAME]
--- POSITIONS ---
 BANANA: 16
 APPLE: 22
 ORANGE: 26
-----------------

[GAME] >> It is now ORANGE's turn. <<

-----------------------------------
IT IS YOUR TURN! (Current Pos: 26)
Choose Action:
 [1] SAFE (Move 1-3)
 [2] RISK (Roll 1-6, fail moves back)
 [3] EXIT GAME
-----------------------------------
Your Choice (1-3): 2
[CLIENT] Move sent.
[GAME] RISK: Rolled 5... SUCCESS!
[GAME] ORANGE chose RISK, Rolled 5, moved 5 step(s) and crossed the finish line!
! ** WINS THE GAME! **

***********************************
         VICTORY! YOU WON!
***********************************
Total Wins: 1

[CLIENT] Disconnecting...
live@minios:~/Downloads/OS_Assignment/TacticalTokenRace$
```

```
 APPLE: 16
 ORANGE: 25
-----------------

[GAME] >> It is now ORANGE's turn. <<
[GAME] ORANGE chose RISK, Rolled 3 (Fail -1), but the Token (+2) saved them! Net
 move: +1 step to 26.
[GAME]
--- POSITIONS ---
 BANANA: 11
 APPLE: 16
 ORANGE: 26
-----------------

[GAME] >> It is now APPLE's turn. <<
[GAME] APPLE chose RISK, Rolled 6 (Success!), moved 6 step(s) to 22.
[GAME]
--- POSITIONS ---
 BANANA: 11
 APPLE: 22
 ORANGE: 26
-----------------

[GAME] >> It is now BANANA's turn. <<

-----------------------------------
IT IS YOUR TURN! (Current Pos: 11)
Choose Action:
 [1] SAFE (Move 1-3)
 [2] RISK (Roll 1-6, fail moves back)
 [3] EXIT GAME
-----------------------------------
Your Choice (1-3): 2
[CLIENT] Move sent.
[GAME] RISK: Rolled 5... SUCCESS!
[GAME] BANANA chose RISK, Rolled 5 (Success!), moved 5 step(s) to 16.
[GAME]
--- POSITIONS ---
 BANANA: 16
 APPLE: 22
 ORANGE: 26
-----------------

[GAME] >> It is now ORANGE's turn. <<
[GAME] ORANGE chose RISK, Rolled 5, moved 5 step(s) and crossed the finish line!
! ** WINS THE GAME! **

***********************************
       GAME OVER. YOU LOST!
***********************************
Winner: ORANGE | Your Total Wins: 0

[CLIENT] Disconnecting...
live@minios:~/Downloads/OS_Assignment/TacticalTokenRace$
```

```
-----------------

[GAME] >> It is now ORANGE's turn. <<
[GAME] ORANGE chose RISK, Rolled 3 (Fail -1), but the Token (+2) saved them
 move: +1 step to 26.
[GAME]
--- POSITIONS ---
 BANANA: 11
 APPLE: 16
 ORANGE: 26
-----------------

[GAME] >> It is now APPLE's turn. <<

-----------------------------------
IT IS YOUR TURN! (Current Pos: 16)
Choose Action:
 [1] SAFE (Move 1-3)
 [2] RISK (Roll 1-6, fail moves back)
 [3] EXIT GAME
-----------------------------------
Your Choice (1-3): 2
[CLIENT] Move sent.
[GAME] RISK: Rolled 6... SUCCESS!
[GAME] APPLE chose RISK, Rolled 6 (Success!), moved 6 step(s) to 22.
[GAME]
--- POSITIONS ---
 BANANA: 11
 APPLE: 22
 ORANGE: 26
-----------------

[GAME] >> It is now BANANA's turn. <<
[GAME] BANANA chose RISK, Rolled 5 (Success!), moved 5 step(s) to 16.
[GAME]
--- POSITIONS ---
 BANANA: 16
 APPLE: 22
 ORANGE: 26
-----------------

[GAME] >> It is now ORANGE's turn. <<
[GAME] ORANGE chose RISK, Rolled 5, moved 5 step(s) and crossed the finish
! ** WINS THE GAME! **

***********************************
       GAME OVER. YOU LOST!
***********************************
Winner: ORANGE | Your Total Wins: 0

[CLIENT] Disconnecting...
live@minios:~/Downloads/OS_Assignment/TacticalTokenRace$
```

Logging into text file, storing score boards, and server backlogging



File Edit Selection View Go Run Terminal Help
OS_Assignment
game.log U
TacticalTokenRace > data > game.log
scores.txt U
TacticalTokenRace > data > scores.txt
Terminal - live@minios: ~/Downloads/OS_Assignment/TacticalTokenRace
File Edit View Terminal Tabs Help

```
Final Turn Order:
  #1: ORANGE
  #2: APPLE
  #3: BANANA
[LOG] ORANGE chose RISK, Rolled 4 (Success!), moved 4 step(s) to 4.
[LOG] APPLE chose RISK, Rolled 2 (Fail), and stumbled at the start line.
[LOG] BANANA chose SAFE, Rolled 6, moved 3 step(s) to 3.
[LOG] ORANGE chose RISK, Rolled 3 (Fail), moved 1 step backward to 3.
[LOG] APPLE chose RISK, Rolled 1 (Fail), and stumbled at the start line.
[LOG] BANANA chose RISK, Rolled 5 (Success!), moved 5 step(s) to 8.
[LOG] ORANGE chose RISK, Rolled 6 (Success!), moved 6 step(s) to 9.
[LOG] APPLE chose RISK, Rolled 1 (Fail), and stumbled at the start line.
[LOG] BANANA chose RISK, Rolled 1 (Fail), moved 1 step backward to 7.
[LOG] ORANGE chose RISK, Rolled 3 (Fail), moved 1 step backward to 8.
[LOG] APPLE chose RISK, Rolled 3 (Fail), and stumbled at the start line.
[LOG] BANANA chose RISK, Rolled 4 (Success!), moved 4 step(s) to 11.
[LOG] ORANGE chose RISK, Rolled 4 (Success!), moved 4 step(s) to 12.
[LOG] APPLE chose RISK, Rolled 5 (Success!), moved 5 step(s) to 5. [TOKEN GAINED]
[LOG] BANANA chose RISK, Rolled 1 (Fail), moved 1 step backward to 10. [TOKEN GAINED]
[LOG] ORANGE chose RISK, Rolled 6 (Success!), moved 6 step(s) to 18.
[LOG] APPLE chose RISK, Rolled 5 (Success!), moved 7 step(s) to 12.
[LOG] BANANA chose RISK, Rolled 2 (Fail -1), but the Token (+2) saved them! Net move: +1 step to 1
[LOG] ORANGE chose RISK, Rolled 4 (Success!), moved 4 step(s) to 22.
[LOG] APPLE chose RISK, Rolled 2 (Fail), moved 1 step backward to 11.
[LOG] BANANA chose RISK, Rolled 2 (Fail), moved 1 step backward to 10. [TOKEN GAINED]
[LOG] ORANGE chose RISK, Rolled 5 (Success!), moved 5 step(s) to 27.
[LOG] APPLE chose RISK, Rolled 5 (Success!), moved 5 step(s) to 16.
[LOG] BANANA chose RISK, Rolled 3 (Fail -1), but the Token (+2) saved them! Net move: +1 step to 1
[LOG] ORANGE chose RISK, Rolled 3 (Fail), moved 1 step backward to 26.
[LOG] APPLE chose RISK, Rolled 3 (Fail), moved 1 step backward to 15. [TOKEN GAINED]
[LOG] BANANA chose RISK, Rolled 2 (Fail), moved 1 step backward to 10. [TOKEN GAINED]
[LOG] ORANGE chose RISK, Rolled 2 (Fail), moved 1 step backward to 25. [TOKEN GAINED]
[LOG] APPLE chose RISK, Rolled 1 (Fail -1), but the Token (+2) saved them! Net move: +1 step to 16
[LOG] BANANA chose RISK, Rolled 2 (Fail -1), but the Token (+2) saved them! Net move: +1 step to 1
[LOG] ORANGE chose RISK, Rolled 3 (Fail -1), but the Token (+2) saved them! Net move: +1 step to 2
[LOG] APPLE chose RISK, Rolled 6 (Success!), moved 6 step(s) to 22.
[LOG] BANANA chose RISK, Rolled 5 (Success!), moved 5 step(s) to 16.
[LOG] ORANGE chose RISK, Rolled 5, moved 5 step(s) and crossed the finish line!! ** WINS THE GAME!
```

```
HANA 0
WAN 1
IMAN 0
BANANA 0
APPLE 0
ORANGE 1
```

```
live@minios:~/Downloads/OS_Assignment/TacticalTokenRace$ make
mkdir -p data
mkdir -p tmp
live@minios:~/Downloads/OS_Assignment/TacticalTokenRace$ ./server
[SERVER] Scores loaded (3 records).
[SERVER] Shared Memory & Sync Primitives Initialized.
[SERVER] Main FIFO created at tmp/ttr_server_fifo
[SERVER] Game Server Running...
[SERVER] Waiting for clients via IPC Pipe: tmp/ttr_server_fifo
[LOGGER] Thread started.
[SCHEDULER] Thread started.
[SCHEDULER] Waiting for Host configuration & Players...
[SERVER] BANANA joined (ID 0). Forking child...
[LOG] Player 0 (BANANA) connected.
[SERVER] APPLE joined (ID 1). Forking child...
[LOG] Player 1 (APPLE) connected.
[SERVER] ORANGE joined (ID 2). Forking child...
[LOG] Player 2 (ORANGE) connected.
[SERVER] Host set size to: 3 players.
[LOG] Lobby size set to 3. Waiting...
[SCHEDULER] Lobby Ready. Rolling...
[LOG]
Final Turn Order:
  #1: ORANGE
```