**Ghulam Ishaq Khan Institute of Engineering Sciences and Technology, Topi Swabi. KP.**

A circular logo with text and symbols

Description automatically generated

**Project Name: TIC-TAC-TOE Game implementation**

**Topic: Semester Project Proposal**

**Subject: CS-311 OPERATING SYSTEM LAB**

**Zarish Asim**

**2022640**

**Adina Kamran**

**2022044**

**1. Introduction**

This project is a **Tic-Tac-Toe** game implemented in **C** using **multithreading** to simulate two players (Player X and Player O) playing the game concurrently. The game board is synchronized with a mutex to ensure thread safety during the game. The players are prompted for their moves, and the game checks for a win or draw condition after each move.

**Objectives**

* To implement a Tic-Tac-Toe game where two players interact concurrently via threads.
* To use **mutex locks** to manage the synchronization of the game board while players make their moves.
* To ensure thread safety when updating shared resources like the game board.

**2. System Design**

**2.1. Overview**

The game is designed to have two players ('X' and 'O') playing a 3x3 grid (Tic-Tac-Toe board). The core components of the system are:

* **Game Board**: A 3x3 grid where the players make their moves.
* **Player Threads**: Each player has a separate thread that takes their input and updates the board.
* **Mutex**: To ensure that only one thread updates the board at a time, preventing race conditions.

**2.2. Data Structures**

* **Game Board**: A 2D array of characters, char board[SIZE][SIZE], where each element represents a cell on the game board.
* **PlayerData**: A struct to hold information about each player such as their symbol ('X' or 'O'), the move they want to make, and a reference to the game board and mutex for synchronization.

**2.3. Flow of Execution**

1. **Initialization**: The game board is initialized, and the mutex is created.
2. **Game Loop**: The game alternates between Player X and Player O. Each player, represented by a separate thread, makes a move. The game checks after each move whether a player has won or if the game is a draw.
3. **Thread Synchronization**: The game board is protected by a mutex to ensure that only one player can modify it at a time, preventing concurrent write conflicts.

**3. Code Breakdown**

**3.1. Header Files**

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#include <stdbool.h>

* **stdio.h**: Provides input/output functions like printf and scanf.
* **pthread.h**: Contains the necessary functions for creating and managing threads.
* **stdlib.h**: For memory allocation using malloc and free.
* **stdbool.h**: For using boolean types (true, false).

**3.2. Game Board and Player Data Structure**

#define SIZE 3

typedef struct {

char player;  
    int move;  
    char (\*board)[SIZE];  
    pthread\_mutex\_t \*mutex;  
} PlayerData;

* **SIZE**: Defines the board size (3x3).
* **PlayerData struct**: Stores information about each player, including:
  + player: The symbol of the player ('X' or 'O').
  + move: The player's move (input from 1 to 9).
  + board: A pointer to the game board.
  + mutex: A pointer to the mutex used for synchronizing access to the board.

**3.3. Memory Allocation for the Game Board**

char(\*board)[SIZE] = malloc(sizeof(char[SIZE][SIZE]));

if (board == NULL) {  
    printf("Memory allocation failed!\n");  
    return 1;  
}

* Dynamically allocates memory for the game board.
* If the memory allocation fails, the program will terminate with an error message.

**3.4. Board Initialization**

for(int i = 0; i < SIZE; i++) {  
    for (int j = 0; j < SIZE; j++) {  
        board[i][j] = '1' + i \* SIZE + j;  
 }  
}

* Initializes the game board with numbers from '1' to '9' for easy identification of available spots.

**3.5. Mutex Initialization**

pthread\_mutex\_t mutex;  
pthread\_mutex\_init(&mutex, NULL);

* Initializes the mutex used to protect the game board from concurrent access by multiple threads.

**3.6. Game Loop and Thread Creation**

while (!gameWon && !gameDraw) {  
    if (currentPlayer == 'X') {  
        pthread\_create(&thread1, NULL, getPlayerMove, &player1);  
        pthread\_join(thread1, NULL);  
        gameWon = checkWin(board, 'X');  
    } else {  
        pthread\_create(&thread2, NULL, getPlayerMove, &player2);  
        pthread\_join(thread2, NULL);  
        gameWon = checkWin(board, 'O');  
    }    displayBoard(board);    if (gameWon) {  
        printf("Player %c wins!\n", currentPlayer);  
 break;  
 }  
 gameDraw = checkDraw(board);  
 if (gameDraw) {  
 printf("It's a draw!\n");  
 break;  
 }  
 currentPlayer = (currentPlayer == 'X') ? 'O' : 'X';  
}

* **Game loop**: Alternates turns between Player X and Player O, creating a new thread for each player's move and ensuring that the board is updated in sequence.
* **pthread\_create**: Creates a thread for each player's move.
* **pthread\_join**: Waits for the thread to finish before proceeding with the next player's turn.

**3.7. Freeing Resources and Mutex Destruction**

free(board);  
pthread\_mutex\_destroy(&mutex);

* Frees the dynamically allocated memory for the board.
* Destroys the mutex after the game ends to release system resources.

**3.8. Displaying the Board**

void displayBoard(char board[SIZE][SIZE]) {  
    for (int i = 0; i < SIZE; i++) {  
        for (int j = 0; j < SIZE; j++) {  
            printf(" %c ", board[i][j]);  
            if (j < SIZE - 1) printf("|");  
        }        printf("\n");  
        if (i < SIZE - 1) printf("---|---|---\n");  
 }  
}

* **displayBoard**: Prints the current state of the board to the console, showing the moves of players ('X' or 'O') in each cell.

**3.9. Win Condition Check**

Bool checkWin(char board[SIZE][SIZE], char player) {  
    for (int i = 0; i < SIZE; i++) {  
        if ((board[i][0] == player && board[i][1] == player && board[i][2] == player) ||  
            (board[0][i] == player && board[1][i] == player && board[2][i] == player)) {  
            return true;  
        }    }    if ((board[0][0] == player && board[1][1] == player && board[2][2] == player) ||  
        (board[0][2] == player && board[1][1] == player && board[2][0] == player)) {  
 return true;  
 }  
 return false;  
}

* **checkWin**: Checks all rows, columns, and diagonals to see if the current player has won the game.

**3.10. Draw Condition Check**

bool checkDraw(char board[SIZE][SIZE]) {  
    for (int i = 0; i < SIZE; i++) {  
        for (int j = 0; j < SIZE; j++) {  
            if (board[i][j] != 'X' && board[i][j] != 'O') {  
                return false;  
            }  
        }  
    }  
    return true;  
}

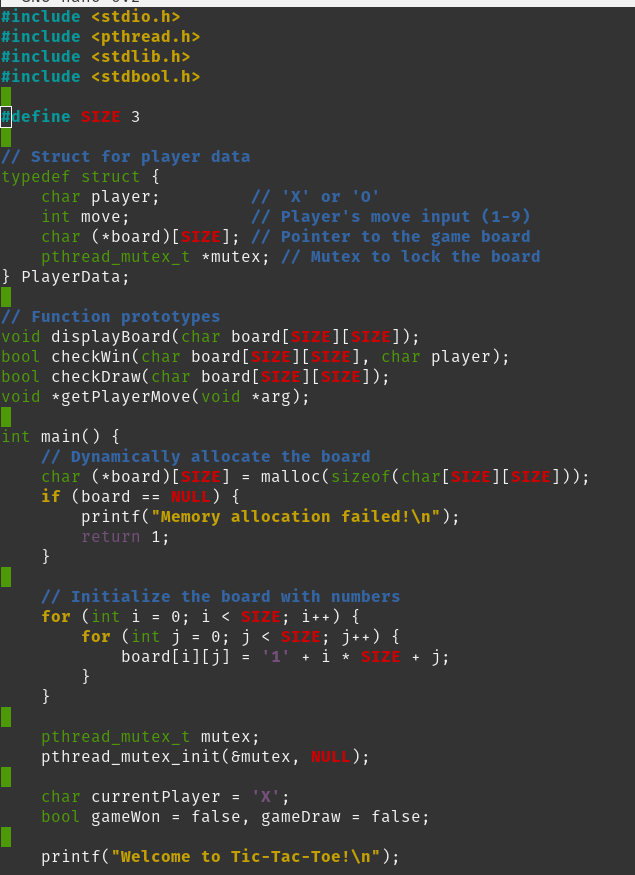
* **checkDraw**: Checks if the board is full, meaning no moves are left and the game is a draw.

**3.11. Getting Player Move**

void \*getPlayerMove(void \*arg) {  
    PlayerData \*playerData = (PlayerData \*)arg;    int move;  
    bool validMove = false;  
    while (!validMove) {  
        pthread\_mutex\_lock(playerData->mutex);        printf("Player %c, enter your move (1-9): ", playerData->player);  
        scanf("%d", &move);  
        int row = (move - 1) / SIZE;  
        int col = (move - 1) % SIZE;  
        if (playerData->board[row][col] != 'X' && playerData->board[row][col] != 'O') {  
            playerData->board[row][col] = playerData->player;  
            validMove = true;  
 } else {  
 printf("Spot already taken! Choose another.\n");  
 }  
 pthread\_mutex\_unlock(playerData->mutex);  
 }  
 return NULL;  
}

* **getPlayerMove**: Receives the player's move input and updates the board. The mutex is locked before modifying the board to ensure thread safety.

1. **Code implementation:**

A screenshot of a computer program

Description automatically generated

A computer screen shot of a program

Description automatically generated

A screenshot of a computer game

Description automatically generatedA screenshot of a game

Description automatically generated

These are the ones run in terminal with actual players playing

**5. Conclusion**

The Tic-Tac-Toe game utilizes **multithreading** to simulate a game between two players, with each player running in their own thread. **Mutex locks** are used to ensure that the game board is updated safely when both players attempt to make a move concurrently. The program handles both win and draw conditions and outputs the game result after each turn.

**Future Improvements**

* **Error Handling**: Improve the validation of player input (e.g., handle invalid input more robustly).
* **Graphical Interface**: Create a graphical user interface (GUI) for better user interaction.
* **AI Player**: Add an AI player for single-player mode.

This project demonstrates the application of **multithreading** and **mutexes** in a simple game scenario, showcasing basic concurrency control in C.