Lab 5

CC3100 WIFI COMMUNICATION, MULTIPLAYER PONG GAME, DESIGNING A FULL GAME PROTOCOL

OBJECTIVES

- Use priority-based multi-threading to implement a multiplayer game over WiFi.
- No library development. But many many threads.
- The game structure and an API is given to you. You need to fill in the body of the functions.

REQUIRED

- More Hardware
  - C3100 Booster Pack
- Software
  - Lab 4 G8RTOS
  - Lab 4 LCDLib
  - Board Support Package
  - C3100 Support Package
- You will need at least two boards for testing. For that you can pair up with a classmate.
- Note that the two players should be running the same code.

Game Description

- Game starts with no balls in the arena. Balls are added to the game randomly although with a low probability.
- Initially, the ball color is white which means no one owns it. Once it is hit it will take on a color and get owned by a player.
- Only if a ball that is owned by one player passes the other player's side the score is incremented.

Game Description

- The max number of balls allowed in the game is defined in Game.h. To make the game even more interesting, the time interval for how often a new ball is created will be proportional to the current number of balls currently in play.
- You will be lighting the LEDs to keep track of the score.
- Once a new game is played, the winner's overall score will be incremented. The number of games won by each player will be displayed on the left side of the screen.

Game.h

- The API for this lab is given in the Game.h header:

```
void SendData(_u8 *data, _u32 IP, _u16 BUF_SIZE);
_i32 ReceiveData(_u8 *data, _u16 BUF_SIZE);
void initCC3100(playerType playerRole);
```

- Host IP address is fixed. Client IP address is decided through DHCP.

Game.h

```
/*
 * Struct to be sent from the
client to the host
 */
typedef struct
{
    uint32_t IP_address;
    int16_t displacement;
    uint8_t playerNumber;
    bool ready;
    bool joined;
    bool acknowledge;
} SpecificPlayerInfo_t;
```

```
/*
 * General player info to be used
by both host and client
 * Client responsible for
translation
 */
typedef struct
{
    int16_t currentCenter;
    uint16_t color;
    playerPosition position;
} GeneralPlayerInfo_t;
```

Game.h


```
/*
 * Struct of all the balls, only
changed by the host
 */
typedef struct
{
    int16_t currentCenterX;
    int16_t currentCenterY;
    uint16_t color;
    bool alive;
} Ball_t;
```

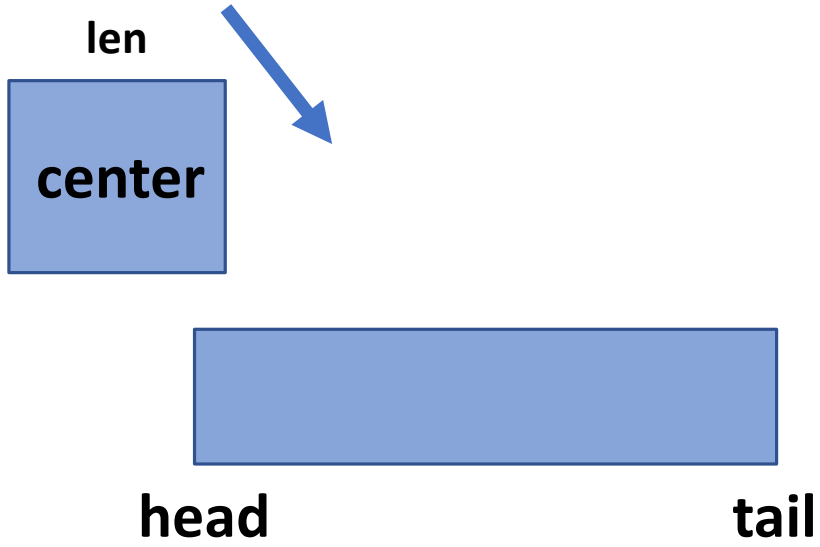
```
/*
 * Struct to be sent from the host to the
client
 */
typedef struct
{
    SpecificPlayerInfo_t player;
    GeneralPlayerInfo_t
players[MAX_NUM_OF_PLAYERS];
    Ball_t balls[MAX_NUM_OF_BALLS];
    uint16_t numberOfBalls;
    bool winner;
    bool gameDone;
    uint8_t LEDScores[2];
    uint8_t overallScores[2];
} GameState_t;
```

Offset Drawing of Objects

- Draw only the difference of objects to save time:

Geometric Collision Detection

- Use the geometry of the shapes to detect collision rather than quadratic search over their area:



- The speed of the balls can be dynamic. You decide their dynamics.

Common Threads

- DrawObjects:
  - Should hold arrays of previous players and ball positions
  - Draw and/or update balls (you'll need a way to tell whether to draw a new ball, or update its position (i.e. if a new ball has just been created – hence the alive attribute in the Ball_t struct.
  - Update players
  - Sleep for 20ms (reasonable refresh rate)

Common Threads

- MoveLEDs:
- Responsible for updating the LED array with current scores

Host Threads

- CreateGame:
  - Only thread created before launching the OS
  - Initializes the players
  - Establish connection with client (use an LED on the Launchpad to indicate Wi-Fi connection)
  - Should be trying to receive a packet from the client
  - Should acknowledge client once client has joined
  - Initialize the board (draw arena, players, and scores)
  - Add the following threads:
    - GenerateBall, DrawObjects, ReadJoystickHost, SendDataToClient ,ReceiveDataFromClient , MoveLEDs (lower priority), Idle
  - Kill self

Host Threads

- GenerateBall:
- Adds another MoveBall thread if the number of balls is less than the max
- Sleeps proportional to the number of balls currently in play

Host Threads

- MoveBall:
  - Go through array of balls and find one that's not alive
  - Once found, initialize random position and X and Y velocities, as well as color and alive attributes
  - Checking for collision given the current center and the velocity
  - If collision occurs, adjust velocity and color accordingly
  - If the ball passes the boundary edge, adjust score, account for the game possibly ending, and kill self
  - Otherwise, just move the ball in its current direction according to its velocity
  - Sleep for 35ms

Host Threads

- ReadJoystickhost:
  - You can read the joystick ADC values by calling GetJoystickCoordinates
  - You'll need to add a bias to the values (found experimentally) since every joystick is offset by some small amount displacement and noise
  - Change Self.displacement accordingly (you can experiment with how much you want to scale the ADC value)
  - Sleep for 10ms
  - Then add the displacement to the bottom player in the list of players (general list that's sent to the client and used for drawing)
  - i.e. players[0].position += self.displacement

Host Threads

- SendDataToClient:
  - Fill packet for client
  - Send packet
  - Check if game is done
    - If done, Add EndOfGameHost thread with highest priority
  - Sleep for 5ms (found experimentally to be a good amount of time for synchronization)

Host Threads

- ReceiveDataFromClient:
  - Continually receive data until a return value greater than zero is returned (meaning valid data has been read)
  - Note: Remember to release and take the semaphore again so you're still able to send data
  - Sleeping here for 1ms would avoid a deadlock
  - Update the player's current center with the displacement received from the client
  - Sleep for 2ms (again found experimentally)

Host Threads

- EndOfGameHost:
  - Wait for all the semaphores to be released
  - Kill all other threads (you'll need to make a new function in the scheduler for this)
  - Re-initialize semaphores
  - Clear screen with the winner's color
  - Print some message that waits for the host's action to start a new game
  - Create an aperiodic thread that waits for the host's button press (the client will just be waiting on the host to start a new game
  - Once ready, send notification to client, reinitialize the game and objects, add back all the threads, and kill self

Client Threads

- JoinGame:
  - Only thread to run after launching the OS
  - Set initial SpecificPlayerInfo_t strict attributes (you can get the IP address by calling getLocalIP()
  - Send player into to the host
  - Wait for server response
  - If you've joined the game, acknowledge you've joined to the host and show connection with an LED
  - Initialize the board state, semaphores, and add the following threads
    - ReadJoystickClient, SendDataToHost, ReceiveDataFromHost, DrawObjects, MoveLEDs, Idle
  - Kill self

Client Threads

- ReadJoystickClient:
  - Read joystick and add offset
  - Add Displacement to Self accordingly
  - Sleep 10ms

Client Threads

- SendDataTohost:
  - Send player info
  - Sleep for 2ms

Client Threads

- ReceiveDataFromHost:
  - Continually receive data until a return value greater than zero is returned (meaning valid data has been read)
    - Note: Remember to release and take the semaphore again so you're still able to send data
    - Sleeping here for 1ms would avoid a deadlock
  - Empty the received packet
  - If the game is done, add EndOfGameClient thread with the highest priority
  - Sleep for 5ms

Client Threads

- EndOfGameClient:
  - Wait for all semaphores to be released
  - Kill all other threads
  - Re-initialize semaphores
  - Clear screen with winner's color
  - Wait for host to restart game
  - Add all threads back and restart game variables
  - Kill Self