Programming Assignment (PA) -2

(Tic-Tac-Toe with Threads)

CS307 – Operating Systems

FALL 2021/2022

Submission Report

by

Furkan Kerim Yıldırım

28138

Problem Description

In this project assignment, we were expected to simulate a simple Tic-Tac-Toe game which will involve accessing shared resources and working with threads. In order to simulate this game, Linux PThreads had to be used as the player.

C Program Implementation

First of all, in tictactoe.c file, thread variables "thread1, thread2" are defined and value "N" for matrix size is received from the user. Afterwards, the matrix is started according to N and empty blank is assigned to each cell by default. Afterwards, a struct named “thread_d” is created to send data to the created threads. thread_d holds a player type (X or Y), a matrix, a matrix size, and a pointer to identify who the next player was. This pointer is processed between threads and allows the threads to understand the next player. Then, the values to be initialized to the threads are assigned using this struct structure. A char "ch" is created to determine who is the first player to start, and this char is sent to the threads with a pointer. After the structs to be sent to the threads are prepared, the thread functions are called and the game starts.

The working principle of thread_funcs is that checks that the game is over. This logic is used inside the while loop. If the game is not over, the mutex is locked and the player starts to play when the player and the order pointer match. After the mutex is locked and before the thread chooses an index, thread makes sure that the game is not over and the player is matched with the order pointer. If the player and the order pointer is not matched, mutex is unlocked. When it is understood that the game is not over yet and the player and the order pointer is matched,

the game is not over and the mutex is locked, the player starts to generate random numbers until it is understood that there is no value in the selected index and marks the index according to the numbers it produces. Player, that is the thread, after marking the index, prints which index it is marking, checks that the game is over and adjusts the order according to the other player. Finally, the thread opens the mutex again and checks the order until it realizes that it is its turn again. When it is understood that the game is over, both threads join the main thread. Then in the main thread it is determined whether there is a winner in the matrix. When it is determined that there is a winner, the winner is determined according to which player has the last turn. If there is no winner, it is printed as a draw. Finally, the final state of the matrix is printed, and the matrix is deallocated, and the program is terminated.

Locking Algorithm

In pseudo code;

```
Initialize isOver as false;
While isOver is not true:
    Lock mutex;
    If it's the player's turn in the thread and isOver is not true:
        Create random row and col index until matrix[row][col] is empty;
        Player in thread marks the matrix[row][col] according to its own sign;
        Print the player and the index the player marked;
        Check if the game is over and update the isOver variable accordingly;
        Update which player's turn it is;
    Unlock mutex;
```

In this algorithm, a single coarse-grained big lock was used to prevent mutual exclusion while threads are processing. Threads try to lock the mutex when they realize that the game is not over yet. The thread that manages to lock the mutex checks whether it is in the turn itself. If the mutex can verify that it is its turn to play, it will mark it according to the index it randomly generates and the turn passes to the other player. The thread that understands that its turn means that it has succeeded in locking the mutex. Based on this, it is understood that the other thread could not lock the mutex. Since the thread that cannot lock the mutex cannot access the values in the matrix, it is unnecessary to lock each cell in the matrix separately (fine grain). Here, when threads realize that the game is not over yet, they try to lock the mutex even if it is not in the turn itself. It is true that this situation creates an inefficiency problem. However, since only 2 threads are used, the inefficiency problem does not lead to bad results and can be neglected. To prevent the inefficiency problem here, atomic mutexes that work with one-time all-or-nothing principle can be used, but this will lead to undesirable situations.