Programming Assignment (PA) - 2
Tic-Tac-Toe with Threads

CS307 - Operating Systems (Spring 2021)

5 April 2021
DEADLINE: 12 April 2021, 23:55

1 Introduction

For this PA, you will simulate a simple Tic-Tac-Toe game which will involve accessing shared resources and working with threads. Tic-Tac-Toe is a turn based game which can be played between two players on a board ($N \times N$ matrix). In each turn, a player marks a cell with an `X` or `O` and gives the turn to the other player. Having N consecutive diagonal, horizontal or vertical marks of the same type is accepted as a winning state. The first player reaching to a winning state wins the game. If there are no cells left to mark on the board and the board is not in a winning state, the result will be a tie. One can assume that square shaped boards will be used for the game.
In order to simulate this game, Linux PThreads will be used as players.

2 Threads

2.1 Parent Thread (Main)

The main thread initiates 2 other player threads together with a 2-D array (matrix) of fixed size representing the board of the game. Size of the matrix should be given to the main method as an argument. Since we are assuming square matrices, a single integer argument is sufficient for determining its size in both dimensions.

The memory required for keeping the matrix should be allocated from the heap. So, it is a shared data structure and the player threads can access and update it concurrently. Accessing a shared data structure may cause a race condition. Therefore, player threads must be properly synchronized to prevent data races using the locking mechanisms discussed before.

The main thread should wait until both children threads terminate. The children terminate either when the board reaches to a winning state or it is fully marked. After children threads join, the main thread prints the string `Game ends` to the console followed by the result and the final board state. See sample runs (Section 4) to understand the format better.

2.2 Children Threads (Players)

Two children threads execute the same procedure. So, they are replica of each other. The only difference is the mark they will be using to modify the board. The mark should be an argument

to the running procedures. There are two marks to be given as inputs: "X" and "O" signs.

The procedure children threads execute consists of a main loop. Inside the loop, the child first tries to acquire a lock. The lock serves to two purposes. First, it ensures synchronized access to the shared table and prevents race conditions. Second, it organizes turns of the players. Each player should wait until other's turn finishes before acquiring the lock. In order to ensure these, the lock should satisfy some guarantees. Details of the lock specification is provided in Section 3. You can directly use or adapt one of the locks we have seen in the lectures to meet the specifications.

After holding the lock, a player (child) thread generates random numbers to find an empty cell in the board and marks it. Then, it releases the lock and gives both the turn and the board to the other player.

The main loop iterates until the board becomes full or reaches to a winning state. 0

3 Implementation Details

3.1 Main Thread

- The matrix size N will be an argument to the main method. Hence, it will be entered as an argument to the program when it is called from the command line (console).
- The main thread generates an $N \times N$ matrix from the heap. All cells must be initialized to a default value different than the marks that will be used by the player threads.
- The main thread creates the lock that will be utilized by the players. Note that the main thread must carefully initiate the lock so that the player with the mark "X" starts the game first every time.
- Both children (player) threads are created from the main thread. If at least one creation fails, the main thread must terminate immediately.
- If both children creation is successful, then the main waits until the both children terminates and then prints game result related information to the console.

3.2 X & O Threads

They will run concurrently in a loop, executing the same procedure. In each iteration, they will:

(a) Acquire the lock.

(b) Select a random cell on the board.

(c) Check if that cell is marked or not.

- If the cell is marked, Go to (b).
- If the cell is empty, mark it with X or O depending on their predetermined input symbol.

(d) Release the lock.

3.3 Synchronization

When there is a shared resource and multiple threads which are trying to access this shared resource, a synchronization mechanism is needed.
Consider the case,

1. Thread X is scheduled.
2. Thread X checks the board (the shared resource).
3. Thread X recognizes cell (0,0) as an empty cell.
4. A context switch occurs and Thread O is scheduled.
5. Thread O checks the board (the shared resource).
6. Thread O recognizes cell (0,0) as an empty cell.

   At this moment both threads will see cell (0,0) as an empty cell.

7. Thread O marks the cell (0,0) with an 'O'.
8. Context wwitch occurs and Thread X is scheduled.
9. Thread X marks the cell (0,0) with an 'X'.

   After these steps, the mark of Thread 0 will be overwritten by Thread X.

As it is stated in the above case, the result is depended on the execution order of the threads. This is an important mutual exclusion problem and should be avoided by using a synchronization mechanism. In order to avert such read/write problems, a possible solution is using locks. There are different ways to use locks on shared objects. One can use individual locks for each cell (fine-grained) or use a one big lock to control access to the whole matrix (coarse-grained). Both of these methods will achieve mutual exclusion if implemented correctly, however the efficiency changes. For the scope of this PA, It is suggested to use a coarse-grained lock to protect the whole object. Since the game is turn based, no two threads are expected to modify disjoint parts of the board at the same time. Hence, using fine-grained locks will be an overkill costing more memory and time wasting synchronization instructions.

In the lectures, we evaluated locks by three parameters: correctness, fairness and performance. Your lock implementation must be correct. Each iteration of the thread procedure described above is a critical section. Hence your lock must ensure its execution under isolation.

Moreover, acquire and release methods should never cause any deadlocks. Here, deadlock means that all threads wait for each other on some condition, their dependencies form a cycle and as a result of that all of them are blocked forever. Aside from an incorrect lock implementation, deadlock can occur due to a misuse of locks. For instance, if a thread holding the lock returns or jumps without releasing the lock, it might cause a deadlock because the same thread might call the acquire method again never releasing it.

Your lock implementation must be a fair one. In the lectures, we have seen a fair lock never starves a thread waiting for the lock. Your lock implementation must be more than fair. It must automatically transfer the control, turn or privilege to the other thread even though the other thread has not requested it yet. We have seen such implementations in the lectures. You might directly use them or adapt other locks to be fair. Using your locks, threads must execute their turns in an alternating manner.

In terms of performance, we have seen that the major cause of the bad performance of locks is the busy-waiting. If a thread requesting the lock iteratively checks the lock's value until it is released, it causes waste of CPU cycles. Solution to this problem requires complicated OS help like `park - unpark` or `wait - signal` methods. Since we have only two threads in our simple program, you do not have to worry about the busy waiting problem and using spinning locks causes no problems.

4 Useful Information & Tips

- You must generate different random numbers at each execution of your program. To ensure this, you might add **srand (time(NULL))** to the beginning of the main. So your random generator will generate different seeds at each execution, thus you will have different random numbers. If you add it inside of a loop, it will try to generate random seed every iteration. Thus, execution time will be very slow. Please do not do that.
- Your program should terminate properly, you will lose points if your program can not exit infinite loop or missing outputs. At the end, all three threads must have terminated.
- You have to use pthreads (POSIX Threads) library and submit a C file, any other thread library will not be accepted as a solution.
- Do not forget to use *-lpthread*, command while compiling.

Submission

You are expected to submit a zip file named *<YourSUUserName>_PA2.zip* until 12 April 2021, 23h55.

The content of the zip file is as follows:

- **report.pdf**: In your report, you must present your locking algorithm as a pseudo code. You discuss which locking mechanism you have chosen, how you adapted it to suit your needs and provide formal arguments on why it satisfies the requirements described above.
- **tictactoe.c**: Your C implementation file.

Grading

Some parts of the grading will be automated. If automated tests fail, your code will not be manually inspected for partial points. Some students might be randomly called for an oral exam to explain their implementations and reports.

Submissions will be graded over 100 points:

1. **Compilation (10 pts):** Your program compiles without an error. When we run the program, we do not get any runtime errors.
2. **Child Creation (10 pts):** Your program is concurrent. The game is played between two children threads. You successfully create children threads.
3. **Dynamic Board Size (10 pts):** Your program takes the board size as a command line argument. If it is a fixed, hard-coded value into the source code, you get zero from this part. Please do not take the board size inside the program using `scanf` method. This will ruin the automated testing and cause you lose points from these parts.
4. **Report (20 pts):** Your report explains the locking algorithm and how it adheres to the specifications (-5 pts if your report is not in pdf format).
5. **No Race Conditions (10 pts):** Accesses to the board cells are synchronized by a coarse-grained global lock. No data race occurs on the board object.
6. **Turn Structure (20 pts):** The game proceeds in turns. The game starts with the turn of X Thread. Turns continue in an alternating manner.
7. **No Deadlock (10 pts):** A set of threads do not get blocked together indefinitely due to cyclic dependencies. Your lock and game implementation avoids deadlocks.
8. **Termination (10 pts):** Both threads terminate when the board reaches to a winning state or it is full. The main thread waits for their termination and prints the result information afterwards.

Item 1 is a precondition for items 2 and 3. Item 2 is a precondition for items 5, 6 and 7. Lastly, Item 7 is a precondition for Item 8.

Sample Output

Due to random nature of threads and random generator, your output could not exactly be same but the output format should be similar with the below examples.

Sample Run 1

```
Board Size: 3x3
Player x played on: (0,1)
Player o played on: (0,2)
Player x played on: (2,2)
Player o played on: (1,1)
Player x played on: (1,2)
Player o played on: (1,0)
Player x played on: (2,1)
Player o played on: (0,0)
Player x played on: (2,0)
Game end
Winner is X
[o][x][o]
[o][o][x]
[x][x][x]
```

Sample Run 2

```
Board Size: 4x4
Player x played on: (0,0)
Player o played on: (1,3)
Player x played on: (2,1)
Player o played on: (3,0)
Player x played on: (0,2)
Player o played on: (0,3)
Player x played on: (3,3)
Player o played on: (1,2)
Player x played on: (2,0)
Player o played on: (0,1)
Player x played on: (3,2)
Player o played on: (2,3)
Player x played on: (1,1)
Player o played on: (1,0)
Player x played on: (3,1)
```

```
Player o played on: (2,2)
Game end
It is a tie
[x][o][x][o]
[o][x][o][o]
[x][x][o][o]
[o][x][x][x]
```

Sample Run 3

```
Board Size: 3x3
Player x played on: (0,0)
Player o played on: (0,2)
Player x played on: (1,1)
Player o played on: (1,2)
Player x played on: (2,0)
Player o played on: (2,2)
Game end
Winner is O
[x][ ][o]
[ ][x][o]
[x][ ][o]
```