CS307 - Operating Systems — 2021-2022 Fall

Programming Assignment 2 Report

Doğukan Yıldırım — 28364

The Locking Algorithm

(a) Pseudocode

```
void* thread_procedure(void* arguments) {
    while (infinite loop):
        lock the mutex

        if there is a winner OR there are no more turns left:
            unlock the mutex
            break the loop

        if it is the thread's turn:
            select a random cell
            while the selected random cell is not an unmarked cell:
                select a random cell

            mark the randomly selected unmarked cell with the thread's mark
            increment turn count by one

            if the game has reached a winning state:
                set winner to thread's mark

            unlock the mutex

        else:
            unlock the mutex
}
```

(b) Code

```
1  void* thread_func(void* arg) {
2      struct Player* args = (struct Player*) arg;
3
4      while (1) {
5          pthread_mutex_lock(&mutex);
6
```

```
        if (winner != ' ' || turn >= N * N + 1) {
            pthread_mutex_unlock(&mutex);
            break;
        }

        if (turn % 2 == args->playerNo % 2) {
            int row, col;

            do {
                row = rand() % N;
                col = rand() % N;
            } while(matrix[row][col] != ' ');

            matrix[row][col] = args->mark;
            turn++;
            printf("Player %c played on: (%d,%d)\n", args->mark, row, col);

            if (winningState(args->mark, row, col) == 1) {
                winner = args->mark;
            }

            pthread_mutex_unlock(&mutex);
        }

        else {
            pthread_mutex_unlock(&mutex);
        }
    }
}
```

(c) Usage and Requirements

Considering the scope of this programming assignment and the fact that the two threads will not modify the shared matrix at the same time, my implementation of the programming assignment will make use of a single coarse-grained lock to access the matrix and other shared variables. Moreover, POSIX thread (pthread) library will be used for initializing and creating the threads; and initializing, locking and unlocking the mutex.

Specified shared variables will be N, the NxN matrix, the winner and the turn count. Each time a thread wants to manipulate or check the value of one of these variables inside the main loop, the thread will lock the mutex first, then follow along with the algorithm. After a thread is done with its turn, it will release the lock for the other thread to continue with its turn. N and turn count will be used to check whose turn it is to manipulate the variables and if the game is over due to completely full matrix with no winner. If a thread wins, it will manipulate the winner shared variable to declare itself as the winner. This variable will also be checked to see whether a game is over so the other thread can quit the loop.

Starting with **correctness** requirement, my lock implementation makes sure that each iteration of thread procedure is under isolation while modifying the shared matrix and variables. Also, in my implementation, there is no deadlock problem since:

- There is only one mutex/lock.
- No thread jumps or returns without releasing the mutex/lock.

Hence, my lock implementation is correct.

Secondly, my lock implementation is **fair** since it will be either the thread1 or thread2's turn to follow along with the algorithm. If either one follows along with the algorithm, the turn count will be incremented by one and the lock will be released, which will make it the other thread's turn to use the lock and execute the algorithm. Threads will execute the algorithm in turns, and no thread will starve because of the lock.

Finally, considering the **performance** requirement, there will be no busy waiting problems or spinning locks due to my implementation. Therefore, CPU cycles will not be wasted and there will not be any performance issues.