Operating Systems
Constructor University
Dr. Jürgen Schönwälder

Module: CO-562
Date: 2023-09-29
Due: 2023-10-06

Problem Sheet #4

Problem 4.1: *multi-threaded coin flipping* (10 points)

20 coins are lying in a row on a table. P persons flip all 20 coins lying on the table N times. Write a program that simulates this by using threads, each person is represented by one thread. By default, there are $P = 100$ persons and each person flips all 20 coins $N = 10000$ times. Implement command line options to control the number of persons (-p) and the number of times a person flips all coins on the table (-n).

Use a global array of characters to represent the coins, where an `X` represents one side of a coin and a `0` represents the other side of a coin. Print the coins on the standard output before the coin flipping starts and print the coins when all persons have finished flipping coins. In addition, print the time it took for all persons to flip the coins.

You must ensure that no coin is flipped by another person, while one person has his turn. Implement three strategies:

1. In the first strategy, you use a global lock to protect the coins. A person first obtains the global lock covering all coins and then the person flips N times all 20 coins and finally, when done flipping coins, the person releases the lock.
2. In the second strategy, you also use a global lock but a person obtains the lock for each iteration, i.e., a person obtains the lock, flips the 20 coins, releases the lock, and then moves to the next iteration.
3. In the third strategy, there is a separate lock for each coin, i.e., a person obtains a lock for a coin, flips the coin, and releases the lock immediately after each coin flip.

Let your program measure the time it takes for all persons to flip all coins and write the results to the standard output. An example execution might look as follows (actual times replaced by underscores):

```
$ ./coins
coins: 00000000000XXXXXXXXXX (start - global lock)
coins: 00000000000XXXXXXXXXX (end - global lock)
100 threads x 10000 flips:   ____.___ ms

coins: 00000000000XXXXXXXXXX (start - iteration lock)
coins: 00000000000XXXXXXXXXX (end - table lock)
100 threads x 10000 flips:   ____.___ ms

coins: 00000000000XXXXXXXXXX (start - coin lock)
coins: 00000000000XXXXXXXXXX (end - coin lock)
100 threads x 10000 flips:   ____.___ ms
```

Submit the times you measured. What do you observe?

Time measurement can be done in a simple (although not very accurate) manner. The `timeit()` function shown below takes as arguments the number of threads and a pointer to a function implementing one of the three coin flipping strategies. The `run_threads()` function creates `n` threads (each one executing `proc`) and joins them again. The calls to `clock()` obtain the a timestamp before and a timestamp after the execution of `run_threads()`. The timestamps are then used calculate the execution time (in microseconds).

```
static double
timeit(int n, void* (*proc)(void *))
{
    clock_t t1, t2;
    t1 = clock();
    run_threads(n, proc);
    t2 = clock();
    return ((double) t2 - (double) t1) / CLOCKS_PER_SEC * 1000;
}
```

The execution times measured on a MacBook Pro (2017) with an 2,3 GHz Intel Core i5 Processor and 8 GB 2133 MHz LPDDR3 RAM are shown below:

```
$ ./coins
coins: 0000000000XXXXXXXXXX (start - global lock)
coins: 0000000000XXXXXXXXXX (end - global lock)
100 threads x 10000 flips:     41.015 ms

coins: 0000000000XXXXXXXXXX (start - table lock)
coins: 0000000000XXXXXXXXXX (end - iteration lock)
100 threads x 10000 flips:    703.950 ms

coins: 0000000000XXXXXXXXXX (start - coin lock)
coins: 0000000000XXXXXXXXXX (end - coin lock)
100 threads x 10000 flips:   2366.441 ms
```

Apparently, the cost of obtaining locks frequently is much higher than the performance gain obtained by using multiple threads.