Homework 1: Critical Sections

Locks, Barriers, and Condition Variables

Alex Lindberg
alex5@kth.se

February 4, 2022

Introduction

This is a short report detailing the first homework — *Critical Sections*. Implementations for task 1, 2 and 4 are included. All implementations were done in C using the pthread library.

1 Matrix Sum, Min and Max

This assignment consists of 3 parts.

a. Max and Min

The first part consists of computing the minimum and maximum values in a matrix and printing them, as well as initializing all elements to random values.

The values can be randomized by using the `rand()` function modulo some value. `rand()` was seeded using `srand()`.

Finding the min and max was done by setting min and max to the first number in the matrix and updating them if needed when we compute the total value of a strip. To be sure every Worker has the same max and min we lock a mutex before updating a global struct containing the values, and then unlock to continue.

```
    :
if (element > max) ->
    lock mutex;
    if(element > max)
        save id;
        max = element
    unlock mutex;
    :
```

b. No Barrier

For this part the struct containing the min and max was enhanced to also contain the total value making it global. Then, by simply locking the value with a mutex it can be updated from any thread. Once the sum-loop is finished we can join the thread, and once all threads have been joined we can print from `main`.

c. Bag of Tasks

In this part the summation was done row-by-row by tracking the next row to be computed using an index. This is all done in a while-loop where we lock a mutex to grab a row index, then set the next row and unlock the mutex.

```
    :
lock mutex;
row = nextRow;
nextRow++;
unlock mutex;
if(row >= size) -> break;
    :
```

Quicksort

To complete this assignment I re-used some code from a previous assignment where a sequential Quicksort was implemented (course ID1021). It took a while to understand recursive parallelism and how to efficiently implement it.

To put it simply, we track how many workers are currently active using a global variable and calling `pthread_create` for **one** of the partitions when sorting and letting the current worker continue on the other partition. If we already have a maximum amount of workers running we simply let the current thread make the recursive call to both partitions.

```
lock mutex;
if (activeWorkers < maxWorkers) ->
    activeWorkers += 1;
    unlock mutex;
    pthread_t worker;
    pthread_create(&worker, NULL, quicksort, part 1);
    quicksort(part 2);
else
    quicksort(part 1);
    quicksort(part 2);
```

Performance Evaluation

As seen in 1 the difference between number of workers seem almost negligible. The laptop running the program has a low-end CPU with 4 cores and no hyper-threading. Unfortunately my desktop runs Windows so I could not get it to work on there. In any case, the sample size could also have affected the performance as i manually ran tests and averaged the results.

n	Workers	Time
1000	1	4.5ms
1000	2	3.4ms
10000	1	32ms
10000	2	26ms
10000	4	30ms
1000000	1	2034ms
1000000	2	1940ms
1000000	4	1850ms

Table 1: Execution time of Quicksort over an array of varying size using different amounts of worker threads.

Tee

The solution to this task was fairly painless. Three threads are launched (as instructed), then the `stdin` reader thread locks a buffer which is filled with the terminal input, then unlocks and locks a second mutex before entering a barrier. This solution was inspired from Task 1. The two writer threads then perform their task and enter the barrier. The process is then repeated.

```
// in readStdIn()
lock bufferLock;
if ((bytes read = getline -> &buffer) > 0)
    unlock bufferLock
    lock readLock
    Barrier();
```

```
// in writeStdout() and writeFile()
lock bufferLock;
if (bytes read > 0)
    unlock bufferLock
    ... perform task ...
    Barrier();
else
    unlock bufferLock
```