# Lab #3: Synchronization

**Lab 3 is due Monday, Nov. 6.** Your lab report and source code must be submitted by **1:50PM** before class. The late policy applies to this lab project.



---

When you measure the performance of the programs you write for this lab, make sure the measurement is stable and consistent. A simple way to do that is to repeat the execution multiple times and use the average.

## Problem 1: Synchronization

Assume we have a *binary tree* with the following data structure declaration. Every node has an integer key "v". The keys on the left subtree are always smaller than or equal to "v", and the keys

| | |
|---|---|
| struct p {<br><br>int v;<br><br>struct p * left;<br><br>struct p * right;<br><br>} | struct p * add (int v, struct p * somewhere);<br><br>struct p * remove(int v, struct p * somewhere);<br><br>int size();<br><br>int checkIntegrity(); |

on the right subtree are always larger than "v".

**Task 1:**

Implement four methods based on the tree. The tree methods are (1) “add”, that adds a key to the tree. The newly added value might be equal to some existing keys in the tree. (2) “remove”, will remove ***a*** node that has key value “v”, if “v” is currently present in the tree, and hoist a child node to its position. If “v” is not appearing in the tree, the method returns NULL. If multiple nodes in a tree contain key “v”, only ONE of them is removed. (3) “size”, which simply returns the total number of nodes in the tree. And (4) “checkIntegrity” that checks the tree is properly formed. If yes, return “1”, otherwise, return “0”. Also when a new node is needed, just malloc the memory for that node.

**Task 2:**

Implement the following workload. Spawn 16 threads, each of which executes one instance of the workload. The goal of this task is to minimize the overall execution time of the 16 threads. N is a constant value, i.e., can’t be changed during execution, but can be set for different program versions. For each subtasks 2.1 and 2.2, two different values for N should be used, N=64 or N=1048576.

```
For(i=0; i<1K; i++){
    int key = random_value_between_1_and_N.
    Add(key, root);
}
For(i=0; i<100K; i++){
    int key = random_value_between_1_and_N.
    Add(key, root);
    remove(key, root);
}
Print size().
Print checkIntegrity();
```

**How to generate a random integer between 1 and N:**

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

In “main” function:

```
srand(time(NULL));
```

When generating:

```
key = rand() % N+1;
```

## Subtask 2.1

First make sure your implementation of the methods in a. is thread-safe, i.e., multiple threads can call them concurrently. The simplest way is to use one lock to protect the whole tree. This is the baseline version for this task. Measure the execution time of the 16 threads based on this baseline version.

### Subtask 2.2

Design and implement a better synchronization schemes at method level (i.e, not protecting the whole loop)., in pthread, for the “add” and “remove” methods, specifically for the example workload. Very importantly, the scheme should provide the best tradeoff between maximum concurrency and lock overhead. The maximum concurrency means that as long as the invocations of the methods from multiple threads change different locations of the tree, the multiple invocation should run concurrently, i.e., not waiting for each other. For example, if n new values end up being inserted into different places of a tree and at the same time multiple existing values are being removed, the n insertions and the removals should happen concurrently. The lock overhead is incurred when a lot of locks are used in a synchronization scheme.

Measure the execution time of the 16 threads based on the new synchronization scheme. Very importantly, you should verify that your implementation is correct at the end of the program: the tree is properly formed (checkIntegrity should return 1), and tree_size = 16K. ***Explain why you think your scheme works better for each value of N***.

## What to submit:

The package you submit for this lab should include your source code, performance results and analysis, and brief description that you think might help the instructor understand your code, e.g., about any design choices you make in your program. The instructor will compile, run and measure the performance of your code. Therefore, you should also describe how to compile and run your code in your submitted documentation.

---

## How to Submit:

Copy your lab report, which is a .pdf, a .doc, or a .html file, and all your source code into an empty directory. Assuming the directory is "submission", make a tar ball of the directory using the following command:

tar czvf [your_first_name]_[your_last_name]_lab3.tar.gz submission.

Replace [your_first_name] and [your_last_name] with your first name and your last name.

Submit the tar ball. The submission time will be used as the time-stamp of your submission.

---