Two Projects in Multi-Core Synchronization

Offered by Prof. Hagit Attiya

**Required background:** operating systems; parallel and distributed programming is a plus.

To register, please contact Prof. Hagit Attiya at hagit@cs.technion.ac.il.

**Project 1. Tree-Based Combining for Concurrent Data Structures**

Combining is an approach for reducing the synchronization overhead when implementing concurrent data structure. **This project aims to implement a new combining mechanism, called TreeCombine, and evaluate its performance on several data structures.**

A concurrent data structure is like a regular data structure, but it allows several threads to apply operations to it, concurrently. An implementation of a concurrent data structure gives procedures (code) for executing these operations in terms of primitive operations, like reads, writes, Compare&Swap, Fetch&Add, Test&Set, and possibly, lock acquisition and release.

The implementation must be linearizable, which is a condition similar to (strict) serialization, namely, the operations must appear to execute in some sequential order. Specifically, there is a serial execution of the same operations with the same results, in which operations that do not overlap preserve their order. (The formal definition is more complicated, but is not needed for the project.)

There are many ways to design such implementations: Some rely on coarse locking, i.e., protecting the whole data structure with a lock, while others use fine-grain locking, i.e., protecting smaller parts of the data structure with locks, and acquiring / releasing them carefully.

This project considers a different approach to implementing concurrent data structures, called combining: The data structure is protected as in coarse locking, however, the thread that acquires the lock on the whole data structure, also performs operations of other waiting threads, and then notifies them. This technique is beneficial when operations are simple, and the cost of applying them is small in comparison with the cost of obtaining the lock.

The key to good combining is (efficiently) knowing which threads have pending operations, and (efficiently) notifying them when their operations are done. In this project you will check a new mechanism that is inspired by a new (mutex) lock implementation, TreeMutex.

The project consists of two tasks:

1. Implement TreeCombine in C, including easily configurable options to pick operations to combine.

2. Provide a framework to compare with alternative combining approaches, and run benchmarks.

The final results should be obtained by running on a multi-core machine in the computer lab which we will provide access to. For coding and debugging, you can use your own machine.

**Required background:** operating systems; parallel and distributed programming is a plus.
Project 2. Implementing a Tree-Based Mutex Lock

This project aims to implement a new (mutex) lock, called TreeMutex, and evaluate its performance within a new sandbox for mutex algorithms. TreeMutex, a recent tree-based lock, is depicted in the figure, and outlined next.

The algorithm uses a binary tree so that leaf $L_p$, for $p \in \{0, \ldots, n - 1\}$, is statically assigned to thread $p$. To obtain the lock, thread $p$ traverses up the path from $L_p$ to the root, writing its identifier to each node along this path. Then, $p$ performs a single memory barrier to ensure that these writes become visible, and then attempts to capture the lock using a CAS operation. If the CAS succeeds, then $p$ has obtained the lock.

The algorithm uses a promotion mechanism. This mechanism allows a thread performing its exit section to facilitate the entry of other threads whose identifiers it reads along the path from the root to its leaf. Identifiers of promoted threads that did not yet enter the critical section are stored in a FIFO queue named promQ. Threads apply operations to promQ before they release the lock, hence its implementation is not required to support concurrent access.

When exiting the critical section, a thread descends down the path from the root to its leaf, reading the identifiers written at every internal node along this path and its child nodes. Thread $p$ promotes any process whose identifier it reads, by adding it to the promotion queue if it is not already there. Finally, $p$ checks the promotion queue: if empty, $p$ releases the lock; otherwise, $p$ dequeues the first process from the promotion queue, and hands over the lock to it.

The project includes three tasks:

1. Implement TreeMutex in C.
2. Install the mutex evaluation sandbox, and compare TreeMutex within it.
3. Find performance optimizations for TreeMutex.

The final results should be obtained by running on a multi-core machine in the computer lab which we will provide access to. For coding and debugging, you can use your own machine.