4.1 Lecture Summary

4.1 Optimistic Concurrency

Lecture Summary: In this lecture, we studied the *optimistic concurrency* pattern, which can be used to improve the performance of concurrent data structures. In practice, this pattern is most often used by experts who implement components of concurrent libraries, such as *AtomicInteger* and *ConcurrentHashMap*, but it is useful for all programmers to understand the underpinnings of this approach. As an example, we considered how the *getAndAdd()* method is typically implemented for a shared *AtomicInteger* object. The basic idea is to allow multiple threads to read the existing value of the shared object *(curVal)* without any synchronization, and also to compute its new value after the addition *(newVal)* without synchronization. These computations are performed *optimistically* under the assumption that no interference will occur with other threads during the period between reading *curVal* and computing *newVal*. However, it is necessary for each thread to confirm this assumption by using the *compareAndSet()* method as follows. (c*ompareAndSet()* is used as an important building block for optimistic concurrency because it is implemented very efficiently on many hardware platforms.)

The method call *A.compareAndSet(curVal, newVal)* invoked on *AtomicInteger A* checks that the value in *A* still equals *curVal*, and, if so, updates *A*'s value to *newVal* before returning *true*; otherwise, the method simply returns *false* without updating *A*. Further, the *compareAndSet()* method is guaranteed to be performed atomically, as if it was in an object-based isolated statement with respect to object *A*. Thus, if two threads, T_1 and T_2 call *compareAndSet()* with the same *curVal* that matches *A*'s current value, only one of them will succeed in updating *A* with their *newVal*. Furthermore, each thread will invoke an operation like *compareAndSet()* repeatedly in a loop until the operation succeeds. This approach is guaranteed to never result in a deadlock since there are no blocking operations. Also, since each call *compareAndSet()* is guaranteed to eventually succeed, there cannot be a livelock either. In general, so long as the contention on a single shared object like *A* is not high, the number of calls to *compareAndSet()* that return *false* will be very small, and the optimistic concurrency approach can perform much better in practice (but at the cost of more complex code logic) than using locks, isolation, or actors.

Optional Reading:

1. Wikipedia article on Optimistic concurrency control

2. Documentation on Java's AtomicInteger class