Overview of the scalable co

February 17, 2015

This article was contributed by Paul McKenney

We all learned about the comm
the same value as Y + X, for a
commutativity rule uses a mo
encompasses the order of exe
arguments to a single operato
atomically adds the value X to variable A at about the same time that another CPU atomically adds the value Y to this same variable. Because addition is commutative in this more concurrent sense, we know that regardless of the order of execution, the overall effect will be to add X + Y to A.

The key insight behind the scalable commutativity rule is that if the ordering of a pair of operations is irrelevant according to the API, then it should be possible to implement those operations so as to avoid scalability bottlenecks. In contrast, if the ordering of the two operations is critically important, then it is most likely impossible to avoid the bottlenecks between these operations.

For example, suppose that the two operations are inserting two objects, each with its own key, into a search data structure not already containing elements with those keys. Suppose further that this structure does not allow duplicate keys. No matter which of the two operations executes first, the result is the same: both objects are successfully inserted. Therefore, the scalable commutativity rule holds that a scalable implementation is possible, and there is, in fact, a wealth of scalable implementations of search structures, ranging from hash tables to radix trees to dense arrays.