Choose Concurrency-Friendly Data Structures

By Herb Sutter, June 27, 2008

Post a Comment

Linked Lists and Balanced Search Trees are familiar data structures, but can they make the leap to parallelized environments?

Linked Lists

Linked lists are wonderfully concurrency-friendly data structures because they support highly localized updates. In particular, as illustrated in Figure 1, to insert a new node into a doubly linked list, you only need to touch two existing nodes; namely, the ones immediately adjacent to the position the new node will occupy to splice the new node into the list. To erase a node, you only need to touch three nodes: the one that is being erased, and its two immediately adjacent nodes.

This locality enables the option of using fine-grained locking: We can allow a potentially large number of threads to be actively working inside the same list, knowing that they won't conflict as long as they are manipulating different parts of the list. Each operation only needs to lock enough of the list to cover the nodes it actually uses.

For example, consider Figure 2, which illustrates the technique of hand-over-hand locking. The basic idea is this: Each segment of the list, or even each individual node, is protected by its own mutex. Each thread that may add or remove nodes from the list takes a lock on the first node, then while still holding that, takes a lock on the next node; then it lets go of the first node and while still holding a lock on the second node, it takes a lock on the third node; and so on. (To delete a node requires locking three nodes.) While traversing the list, each such thread always holds at least two locks—and the locks are always taken in the same order.

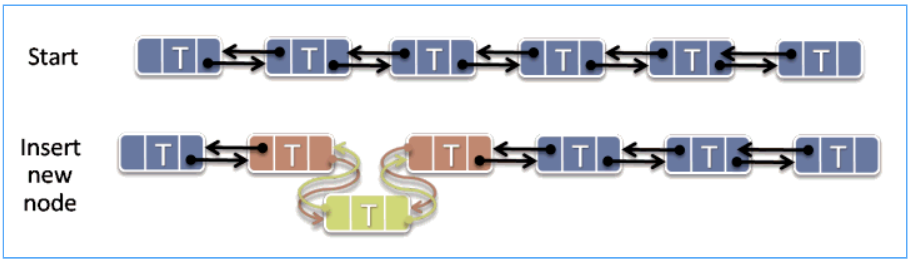

Figure 1: Localized insertion into a linked list.

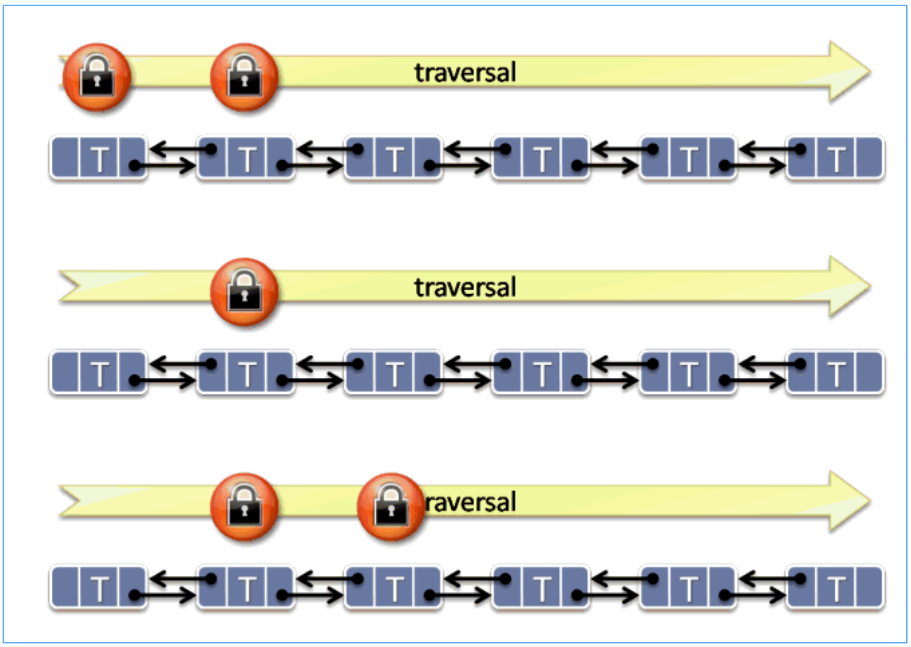

Figure 2: Hand-over-hand locking in a linked list.

This technique delivers a number of benefits, including the following:

- Multiple readers and writers can be actively doing work in the same list.
- Readers and writers that are traversing the list in the same order will not pass each other. This can be useful to get deterministic results in concurrent code. In particular, the list's semantics will be the same as if each thread acquired complete exclusion on the list and performed its complete pass in isolation, which is easy to reason about.
- The locks taken on parts of the list won't deadlock with each other, because multiple locks are acquired in the same order.
- We can readily tune the code for better concurrency vs. lower locking overhead by choosing a suitable locking granularity: one lock for the whole list (no concurrency), a lock for each node in the list (maximum concurrency), or a lock for each chunk of some fixed or variable length (something in between).

Aside: If we always traverse the list in the same order, why does the figure show a doubly linked list? Because not all operations need to take multiple locks; those that use individual segments or nodes in-place one at a time without taking more than one node's or chunk's lock at a time can traverse the list in any order without deadlock. (For more on avoiding deadlock, see [1].)

Besides being well suited for concurrent traversal and update, linked lists also are cache-friendly on parallel hardware. When one thread removes a node, for example, the only memory that needs to be transferred to every other core that subsequently reads the list is the memory containing the two adjacent nodes. If the rest of the list hasn't been changed, multiple cores can happily store read-only copies of the list in their caches without expensive memory fetches and synchronization. (Remember, writes are always more expensive than reads because writes need to be broadcast. In turn, "lots of writes" are always more expensive than "limited writes.")

Clearly, one benefit lists enjoy is that they are node-based containers: Each element is stored in its own node, unlike an array or vector where elements are contiguous and inserting or erasing typically involves copying an arbitrary number of elements to one side or the other of the inserted or erased value. We might therefore anticipate that perhaps all node-based containers will be good for concurrency. Unfortunately, we would be wrong.