Why does unsafe multithreaded use of an std::unordered_map crash more often than unsafe multithreaded use of a std::map?

devblogs.microsoft.com/oldnewthing/20231103-00

November 3, 2023

Raymond Chen

A customer had some code that used a `std::map`, and they found that when they switched to `std::unordered_map` it began to crash a lot more than it did before. Eventually, they traced the problem back to unsafe multithreaded access to the collection, but they wondered why the rate of crashes shot up when they used `std::unordered_map`. What makes `std::unordered_map` more susceptible to multithreading problems?

First, let's reiterate that unsafe multithreaded access to C++ standard library containers is *undefined behavior*. Simultaneous read operations are safe for multithreading, but simultaneous writes or a read and write occurring simultaneously are unsafe for multithreading. The customer understood that their code was broken either way. They just were curious why `unordered_map` seems to demonstrate the problem more clearly.

Recall that `std::map` is typically implemented as a red-black tree. Inserting or erasing an element from a red-black tree is a sequence of up to $O(\log n)$ tree operations. Each tree operation affects a small number of nodes, so concurrent modifications have a relatively low chance of modifying the same node at the same time. Even if they don't modify the same node at the same time, they might still leave the tree in a bad state, but my gut tells me that the most likely consequence would be incorrect results, like failing to find an element that should be there, or having two entries with the same key.

On the other hand, `std::unordered_map` is typically implemented as a hash table with separate chaining. Inserting or erasing an element from a hash table is usually $O(1)$, and the operation affects only two nodes (the node being inserted/erased and its neighbor). Concurrent modifications have a relatively low chance of modifying the same node at the same time, with the note that concurrent modifications could still leave the hash table in a bad state.

I noted that inserting or erasing `std::unordered_map` is *usually* $O(1)$. But sometimes it's $O(n)$. This happens when the hash table needs to be rehashed and the buckets rebuilt. In this case, every element of the hash table is moved to a new bucket, and any attempt to

access the hash table during a rehash operation is going to see a hash table in disarray. This is the case that makes concurrent unsafe access `std::unordered_map` highly likely to crash.

In summary: The `std::map` has a uniformly low risk of crashing, whereas `std::unordered_map` has a generally low risk of crashing, but once in a while, it has a very high risk of crashing. And it's probably that "once in a while" that is causing the crash rates to spike.

Bonus chatter: I reiterate that the problem lies in the unsafe concurrent access, and that's what needs to be fixed. A low risk of crashing is not the same as zero risk of crashing. Even the old code was crashing. It was just crashing at a relatively low rate, and the switch to `std::unordered_map` caused the crash rate to skyrocket. The customer understood this and was just curious why a change in data structure led to a very noticeable change in failure rates.