How to clone a Windows Runtime vector in the face of possible concurrent modification, part 1

devblogs.microsoft.com/oldnewthing/20230712-00

July 12, 2023

Raymond Chen

Some time ago, we looked at making a clone of a Windows Runtime vector in C++/WinRT. We ended up with

```
IVector<Thing> original = GetTheThings();
std::vector<Thing> temp(original.Size(), winrt_empty_value<Thing>());
original.GetMany(0, temp);
IVector<Thing> clone = multi_threaded_vector(std::move(temp));
```

I made two changes to the code:

- I used `winrt_empty_value` to fill the vector with emptiness instead of default-constructed `Thing`s.
- I upgraded the code we had last time from `single_threaded_vector` to `multi_threaded_vector`.

I noted at the time that this code assumes that the vector is not being mutated concurrently. If the vector size changes between the time we check the `Size()` and the time we call `GetMany()`, then we will either read too many or too few items.

I had left dealing with concurrency as an exercise. Let's solve the exercise.

To deal with the case that the vector shrinks unexpectedly, you can check how many items `GetMany` actually retrieved:

```
IVector<Thing> original = GetTheThings();
std::vector<Thing> temp(original.Size(), winrt_empty_value<Thing>());
temp.erase(temp.begin() + original.GetMany(0, temp), temp.end());
IVector<Thing> clone = multi_threaded_vector(std::move(temp));
```

The `GetMany()` method returns the actual number of elements retrieved. If the vector shrinks unexpectedly, then it will return the new smaller size of the vector, and we erase the extra elements. (We saw last time why we are using `erase` instead of `resize`.)

The trickier part is detecting whether the vector grew. In that case, the call to `GetMany()` will return, “Yup, I got all the elements you requested,” but it has no way of telling you, “But there are still more to go.” We’ll have to figure out some other way to get this information.

One thing that might occur to you is to recheck the size after the `GetMany()` call.

```
IVector<Thing> original = GetTheThings();
std::vector<Thing> temp{ original.Size() };
do {
    temp.erase(temp.begin() + original.GetMany(0, temp), temp.end());
} while (temp.size() != original.Size());
IVector<Thing> clone = multi_threaded_vector(std::move(temp));
```

However, this doesn’t work because it fails to detect the case where the vector changes size *twice*.

contents = `{ 3, 2 }` (size = 2)`temp.size() == original.Size()` (2 == 2)

Thread 1	**Thread 2**
contents = `{ 1, 2 }` (size = 2)	
`temp{ original.size() }; // 2`	
	contents = `{ 3, 1, 2 }` (size = 3)
`temp.resize(original.GetMany(0, temp))` `// temp = { 3, 1 }`	

When we re-check the size, we see that the size is still 2, and we think that means that we got all of the items. However, the values in `temp` are `{ 3, 1 }` which was never the contents of the original vector at any point.

The trick is to over-allocate the buffer by one element, and then ask for one too many elements. If we get more elements than we expected, then the vector grew by at least one (but possibly more), so we loop back and try again. Otherwise, we know that we got all the elements, and we can resize the vector to match the actual number.

The important thing is that we get the elements in a single `GetMany()` call, which is atomic.

```
IVector<Thing> original = GetTheThings();
std::vector<Thing> temp;
uint32_t expected;
uint32_t actual;
do {
    expected = original.Size();
    temp.resize(expected + 1, winrt_empty_value<Thing>());
    actual = original.GetMany(0, temp);
} while (actual > expected);
temp.resize(actual, winrt_empty_value<Thing>());
IVector<Thing> clone = multi_threaded_vector(std::move(temp));
```

Next time, we will try to encapsulate this pattern into a function.

Bonus chatter: You might decide to simply abandon the operation in the face of concurrent modification, the theory being that if somebody else is modifying the vector concurrently, we will just report the problem to the caller and let them decide how to proceed.

```
IVector<Thing> original = GetTheThings();
std::vector<Thing> temp;
auto expected = original.Size();
temp.resize(expected + 1, winrt_empty_value<Thing>());
auto actual = original.GetMany(0, temp);
if (actual > expected) {
    throw winrt::hresult_changed_state();
}
temp.erase(temp.begin() + actual, temp.end());
IVector<Thing> clone = multi_threaded_vector(std::move(temp));
```

There's an interesting question here: What should happen if we detect a concurrent modification that nevertheless did not prevent us from making an atomic clone of the vector? That would be the case if `actual < expected`.

One argument is that we should throw a state change exception, just like we do for a concurrent modification that we cannot recover from. The theory here is that we should report concurrent modifications consistently, especially if concurrent modification is not something the caller was expecting. In that case, we would throw when `actual != expected`. It would however fail to detect the case where the concurrent modification did not have a net change to the size of the vector.

Another school of thought is that we should report problems only if we can't fix them. In that case, we throw only if `actual > expected`.