Destructing outside the lock when removing items from C++ standard containers

devblogs.microsoft.com/oldnewthing/20201112-00

November 12, 2020

Raymond Chen

Last time, we saw that object destruction is a hidden callout that can cause you to violate the rule against calling out to external code while holding an internal lock. To recap, here's the class we were using as our example:

```
class ThingManager
{
private:
  std::mutex things_lock_;
  std::vector<std::shared_ptr<Thing>> things_;

public:
   void AddThing(std::shared_ptr<Thing> thing)
   {
      std::lock_guard guard(things_lock_);
      things_.push_back(std::move(thing));
   }

   void RemoveThingById(int32_t id)
   {
      std::lock_guard guard(things_lock_);
      auto it = std::find_if(things_.begin(), things_.end(),
        [&](auto&& thing)
        {
          return thing->id() == id;
        });
      if (it != things_.end()) {
        things_.erase(it);
      }
   }
};
```

We saw that there was a problem at the call to `things_. erase()` because the erasure will result in the destruction of the `shared_ptr`, which could in turn trigger the destruction of an object you don't control. And that uncontrolled code could try to call back into the `ThingManager`, resulting in a mess.

Solving this problem can be tricky in general, although things are a bit easier for us in this case because we know some properties of `shared_ptr`, most importantly that it is noexcept movable.

What we want to do is to remove the element from the vector without destructing it immediately. We want to extend the object's lifetime until after the lock has been released.

Here's one way of doing it that works specifically for `shared_ptr`, taking advantage of its inexpensive default constructor.

```
void RemoveThingById(int32_t id)
{
  std::shared_ptr<Thing> removed;
  std::lock_guard guard(things_lock_);
  auto it = std::find_if(things_.begin(), things_.end(),
    [&](auto&& thing)
    {
      return thing->id() == id;
    });
  if (it != things_.end()) {
    removed = std::move(*it);
    things_.erase(it);
  }
}
```

Before entering the lock, we create an empty `shared_ptr` and use that to hold the element we intend to remove. C++ destructs objects in reverse order of construction, so declaring it before the `guard` means that the `removed` variable destructs after the guard is destructed; in other words, the `removed` variable destructs outside the lock.

If the container is a `std::map` or `std::unordered_map`, you can accomplish this delayed destruction in general by using the `extract` method to extract the entire node, and then destruct the node outside the lock. For `std::list` and `std::forward_list`, you can `splice` the element out of the container into a temporary initially-empty list. But for containers like vectors, you'll need to move the object out of the container, in which case you're counting on the move operation not doing anything scary.

Mind you, you're already assuming that the move operation isn't doing anything scary, because your vector mutation operations are going to move objects around, too.

Fortunately, for things like `shared_ptr`, `unique_ptr`, and (in Windows) COM wrappers, the move operations are indeed not scary. They just shuffle pointers around. (I'm assuming you're moving into an empty object. Obviously, if the destination of the move is not empty, then you're going to run scary code when the previous contents of the destination are released.)

Here are some ways of extracting elements from a container for later destruction:

```
// map, multimap, set, multiset,
// and unordered versions of same
auto removed = source.extract(it);

// vector, dequeue
auto removed = std::move(*it);
source.erase(*it);

// list, forward_list
std::list<T> removed; // or std::forward_list
removed.splice(removed.begin(), source, it);
return removed;
```

If you need to delay-destruct multiple items, you could extract them into a temporary vector.

```
template<typename FwdIt, typename OutIt, typename Pr>
FwdIt extract_if(FwdIt first, const FwdIt last, OutIt out, Pr pred)
{
    first = std::find_if_not(first, last, std::ref(pred));
    auto next = first;
    for (; first != last; ++first) {
            if (pred(*first)) {
                *out = std::move(*first);
                ++out;
            } else {
                *next = std::move(*first);
                ++next;
            }
    }

    return next;
}

std::vector<T> removed;
v.erase(extract_if(v.begin(), v.end(), std::back_inserter(removed), filter),
v.end());
```

Keep the `removed` object alive beyond the release of the lock, so that the objects are destructed outside the lock.

Bonus chatter: It would be convenient if `vector`, `list`, and `forward_list` also had `extract` methods which returned the removed object, similar to the node extractors for the map and set types. Until then, we'll have to write our own:

```
template<typename T>
T extract(
    std::vector<T>& v,
    typename std::vector<T>::iterator it)
{
    auto cleanup = wil::scope_exit([&] { v.erase(it); });
    return std::move(*it);
}

template<typename T>
std::list<T> extract(
    std::list<T>& source,
    typename std::list<T>::const_iterator it)
{
    std::list<T> removed;
    removed.splice(removed.begin(), source, it);
    return removed;
}

template<typename T>
std::forward_list<T> extract_after(
    std::forward_list<T>& source,
    typename std::forward_list<T>::const_iterator it)
{
    std::forward_list<T> removed;
    removed.splice_after(removed.before_begin(), source, it);
    return removed;
}
```

Exercise: Why did I use a `scope_exit` in the vector extractor? Why not

```
template<typename T>
T extract(std::vector<T>& v,
    std::vector<T>::iterator it)
{
    auto result = std::move(*it);
    v.erase(it);
    return result;
}
```

Update: Wrote `extract_if` because the previous version used `remove_if` incorrectly.

Raymond Chen

Follow