The hidden callout: The destructor

devblogs.microsoft.com/oldnewthing/20201111-00

November 11, 2020

Raymond Chen

This is a general problem, but I'm going to give a specific example.

C++ standard library collections do not support concurrent mutation. It is the caller's responsibility to serialize mutation operations, and to ensure that no mutation occurs at the same time as any other operation. And the usual way of accomplishing this is to use a mutex of some kind.

```
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);
    }
  }
};
```

The idea here is that you give the `ThingManager` a bunch of things, and then you can later remove them by providing the `Thing` 's ID. Presumably there are also methods to search for `Thing` s or to perform some operation across all `Thing` s, but those are just distractions from the exercise.

This particular object wants to support concurrent operation, so it internally uses a mutex to ensure safe operation.

Now, you can quibble about the use of `find_if` instead of `remove_if`, or using a `std::vector` instead of a `std::map`, but let's set that aside.

The question is: What's wrong with this code?

I sort of gave it away in the title: We are calling out to external code while holding our lock.

You probably know not to call out to external code when holding an internal lock, and the act of invoking a method on an object may remind you of that fact. But destructors just run by themselves. You don't typically write code the trigger the destruction of an object. The object just destructs when it destructs.

Removing the `shared_ptr<Thing>` from our vector could result in the `Thing`'s destruction if this was the last strong reference to the `Thing`. And that destructor runs while the `things_lock_` is still locked.

Now things get interesting.

You may not know all that happens inside the `Thing` destructor. It may have been written by another team, or by you, six months ago. Or somebody could have derived from `Thing` and given you a shared pointer to the derived object.[1] Or you might be given a shared pointer to a `Thing` embedded inside a larger object.

Let's do that thing with the derived type:

```
class SuperThing : Thing
{
private:
  ThingManager& manager_;
  int32_t helper_id_ = 0;

public:
  SuperThing(ThingManager& manager) :
    manager_(manager)
  {
    auto helper = std::make_shared<Thing>();
    helper_id_ = helper->id();
    manager_.AddThing(helper);
  }

  ~SuperThing()
  {
    manager_.RemoveThingById(helper_id_);
  }
};
```

The `SuperThing` object is itself a `Thing`, but it also uses a helper thing. At construction, it creates a helper thing and registers it with the manager, retaining the ID. And at destruction, it removes its helper thing.

And then this happens:

```
void test(ThingManager& manager)
{
  auto s = std::make_shared<SuperThing>();
  auto id = s->id();
  manager.AddThing(s);
  s = nullptr;

  manager.RemoveThingById(id);
}
```

This little test function creates a `SuperThing`, adds it to the thing manager, and then immediately removes it.

The `RemoveThingById` function looks for a matching Id and finds it, so it erases the corresponding `Thing` from the vector. That erasure destroys the `shared_ ptr`, and since this is the last strong reference, the underlying `Thing` is also destroyed.

This runs the destructor of our `SuperThing`, which tries to remove its helper `Thing`. And that calls back into the `ThingManager`, which gets stuck trying to acquire a mutex that is already held (unwittingly, by itself).

This is not a purely theoretical exercise. This sort of thing happens, and it's a source of bugs.

Next time, we'll look at how to address these types of problems.

[1] If you work in Windows, a common scenario for this is that the `shared_ptr` in the example above takes the form of a COM reference-counted pointer.

Raymond Chen

Follow