Acquiring Locks without Deadlocks

Previous Next

For our example, it is fairly easy to avoid the deadlock caused by cyclic dependencies: instead of holding the lock for the duration of `destroy`, we set the `_destroyed` flag under protection of the lock and unlock `_m` again before calling `remove` on the factory:

C++11

```
void
PhoneEntryI::destroy(const Current& c)
{
    {
        lock_guard<mutex> lock(_m);

        if(_destroyed)
        {
            throw ObjectNotExistException(__FILE__, __LINE__);
        }

        _destroyed = true;

    } // _m is unlocked here.

    _factory->remove(_name, c.adapter);
}
```

Now deadlock is impossible because no function holds more than one lock, and no function calls another function while it holds a lock. However, rearranging locks in this fashion can be quite difficult for complex applications. In particular, if an application uses callbacks that do complex things involving several objects, it can be next to impossible to prove that the code is free of deadlocks. The same is true for applications that use condition variables and suspend threads until a condition becomes true.

At the core of the problem is that concurrency can create circular locking dependencies: an operation on the factory (such as `getDetails`) can require the same locks as a concurrent call to `destroy`. This is one reason why threaded code is harder to write than sequential code — the interactions among operations require locks, but dependencies among these locks are not obvious. In effect, locks set up an entirely separate and largely invisible set of dependencies. For example, it was easy to spot the mutual dependency between the factory and the servants due to the presence of `remove`; in contrast, it was much harder to spot the lurking deadlock in `destroy`. Worse, deadlocks may not be found during testing and discovered only after deployment, when it is much more expensive to rectify the problem.

Back to Top ^

See Also

- Removing Cyclic Dependencies

Previous Next