Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Concurrent Programming with Chain Locking

Concurrent access to trees and lists requires carefully managed fine-grained locking. Here's a generic solution in C# that removes many of the typical problems.

February 26, 2013
URL:http://www.drdobbs.com/parallel/concurrent-programming-with-chain-lockin/240149442

In this article, I explain the many problems associated with concurrent access to hierarchical data structures. I explore several possible solutions, and describe a generic construct that encapsulates much of the complexity associated with locking hierarchical data structures in C#. The design, though, can be implemented in any programming language.

Let's start with a description of a problem with similar characteristics to my real project at work. I'm working on a scheduling service for online classes. The main data structure is a tree consisting of lectures. For each lecture, there are multiple class sessions (which can be represented as branches). Each class session contains slots that students can occupy. The invariants are: If the number of students exceeds a threshold, a new class session is added; and students can only attend one class at a time. (If a student joins another class, he is logged out of his current class.) So, the data structure that needs to be managed is a tree:

1. Lectures
2. Classes
3. Students

The code needs to support a small set of operations:

- Attend lecture (student wants to attend a lecture)
- Expel student (student is removed from current class)
- Cancel class (stop a particular running class)
- Cancel lecture (stop all running classes of this lecture)
- Get statistics (number of classes running and number of students overall)

When this service is being accessed by more than one user, clearly there will be a contention issue that needs intelligent management of mutual exclusion and locking. Let's quickly look at the locking strategies available to us:

Coarse-Grained Locking. Coarse-grained locking is relatively simple to implement. In every thread that needs to access some shared state, you lock the entire state, read/write to your heart's content, and then release the lock. The access to the shared state is serialized and only one thread can access it. The problem with coarse-grained locking is that it's slow. If you have hundreds of threads that need to access the shared state, they will all have to wait until the current thread that holds the lock finishes its work, and only then will another thread be allowed do its thing. Such a system will most likely perform much worse than a single-threaded system. Coarse-grained locking is only useful if your threads execute quickly and, as a result, don't create a lot of contention.

Fine-Grained Locking. With fine-grained locking, multiple locks are used in a set sequence to lock the smallest possible part of the data structure that the current thread needs to operate on. This gives other threads the opportunity to work in parallel on other parts of the data. Fine-grained locking is where most concurrency problems emerge: race conditions, dead locks, live locks, lock convoys, and so on.

Lock-Free Algorithms and Data Structures. Lock-free algorithms address the issues raised by locks, but bring their own set of problems. Their use in the industry is still fairly new. At their core, they rely on atomic operations at the hardware level. It is very hard to design and implement lock-free algorithms properly because the building blocks are very small; when you compose them, the emerging behavior is not trivial to analyze.

ChainLocker

In this article, I'll examine the most common solution — fine-grained locking. The approach I use here, called ChainLocker, encapsulates the concept of sequenced or hand-over-hand locking of a chain of locks. Hand-over-hand locking refers to locking a top-level item (lectures tree root), looking for a particular branch (a specific lecture), locking the lecture, and only then releasing the lectures tree root — so other threads can access other lectures in parallel. Once you have the lecture locked, you can look for a particular class, lock it, then release the lecture lock, so other threads can operate on other classes of this lecture. Here is pseudocode that demonstrates the concept:

```
lock(root)
lecture = root.find_lecture(...)
lock(lecture)
unlock(root)
class = lecture.find_class(...)
lock(class)
unlock(lecture)
process(class)
unlock(class)
```

To remove an element from the tree (such as a lecture), it's necessary to make sure nobody else has any class locked. You can do this by trying to lock all the classes in addition to the lecture, and only then remove the lecture; or you can mark the lecture as `CLOSED` and remove it later in a background task after a short delay to make sure all current threads finish processing (new requests for a `CLOSED lecture` will fail as though it had never existed).

Design and Implementation

`ChainLocker` is a generic class that can be parameterized by the types of objects you want to lock. In our example, the types are `IDictionary<int, Lecture>`, `Lecture`, and `Class`. A `ChainLocker` can lock arbitrarily long chains of objects. Here, I present a three-level deep `ChainLocker`, but I'll later introduce a `ChainLocker` generator that can generate any size `ChainLocker` you need. Listing One presents the code for the three-level `ChainLocker`.

Listing One

```
using System;
using System.Linq;
using System.Threading;

namespace ChainLocker
{
    public class ChainLocker<T0, T1, T2>
        where T0 : class
        where T1 : class
        where T2 : class
    {
        public void Do(T0 state0,
                        Func<T0, T1> stage0,
                        Func<T1, T2> stage1,
                        Action<T2> stage2)
        {
            if (state0 == null)
            {
                throw new Exception("state0 can't be null");
            }

            var takenLocks = Enumerable.Repeat(false, 3).ToArray();
            var states = Enumerable.Repeat<object>(null, 3).ToArray();
            states[0] = state0;
            try
            {
                // Lock state0
                Monitor.Enter(states[0], ref takenLocks[0]);
                // Execute stage0
                states[1] = stage0((T0)(states[0]));
                // Bail out if returned null
                if (states[1] == null)
                {
                    return;
                }

                // Lock state1
                Monitor.Enter(states[1], ref takenLocks[1]);
                // Release the state0 lock so other threads can work on it
                Monitor.Exit(states[0]);
               // Execute stage1
                states[2] = stage1((T1)(states[1]));
                // Bail out if returned null
                if (states[2] == null)
                {
                    return;
                }

                // Lock state2
                Monitor.Enter(states[2], ref takenLocks[2]);
                // Release the state1 lock so other threads can work on it
                Monitor.Exit(states[1]);
                takenLocks[1] = false;
                // Execute stage2
                stage2((T2)(states[2]));
            }
            finally
            {
                // Release still held locks (If an exception was thrown)
                for (int i = 0; i < 3; ++i)
                {
                    if (states[i] != null && takenLocks[i])
                    {
                        Monitor.Exit(states[i]);
                    }
                }
            }
        }
    }
}
```

Here is the class definition:

```
public class ChainLocker<T0, T1, T2>
        where T0 : class
        where T1 : class
        where T2 : class
```

```
{
    public void Do(T0 state0,
                          Func<T0, T1> stage0,
                          Func<T1, T2> stage1,
                          Action<T2> stage2)
    {
       ...
    }
```

Before diving into the implementation, let's make sense of the class definition. `T0`, `T1`, and `T2` are the parameterized types in the order they should be locked. Note, the constraint `"where Tx : class."` In the management service (MS) example we are running, `T0` is the root (`IDictionary<int, Lecture>`), `T1` is `Lecture`, and `T2` is `Class`. `ChainLocker` has a single method called `Do()`. `Do()` has an interesting signature. It accepts an initial state of type `T0` called `state0`, and then three stages (`stage0`, `state1`, and `stage2`). The initial state `state0` is the root, which must be locked first. The `stage` parameters are delegates that specify what processing happens at each stage once the appropriate state is locked properly. If you are not familiar with the C#/.NET `Func` and `Action` delegates, they encapsulate an anonymous method with an arbitrary signature. `Action` is for `void` methods that don't do anything, and `Func` is for methods with a return type. Both can take [any number and type of arguments](#).

For example, `stage0` is defined as `Func<T0, T1>`, which means it's a method that takes an object of type `T0` and returns an object of type `T1`. The last argument to `Do() stage2` is `Action<T2>`, which translates as a method that takes a `T2` object and returns nothing (`void`).

What is the purpose of this strange signature and how does `ChainLocker` work? First off, an important concept of `ChainLocker` is quick exit. Remember that `ChainLocker` keeps scarce resources locked. It is often the case that a particular workflow will not go through the entire chain of `root`, `lecture`, `class`. For example, consider a student trying to join a nonexistent lecture. Once in `stage0`, it becomes clear that there is no such lecture and there is no point in going to `stage1` (processing the `lecture`) and `stage2` (processing the `class`). `ChainLocker` will bail out immediately at the end of `stage0`. How does the generic `ChainLocker` figure out that it should bail out early? That is the responsibility of the caller who provides the processing delegates. `ChainLocker` relies on an unwritten contract that if a `stage` delegate returns `null`, it will bail out immediately. Here is the relevant snippet for `stage0`:

```
// Execute state0
states[1] = stage0((T0)(states[0]));
// Bail out if returned null
if (states[1] == null)
{
    return;
}
```

The last stage (`Action<T2> state2`) returns nothing because there is no bailing out early after the last stage. An interesting consequence of this design is that entire `Do()` method returns nothing. As part of processing, you can assign a result to a property on one of the objects you process, but `ChainLocker.Do()` itself is always `void`. It is possible to extend `ChainLocker` to support a return type for `Do()`, but I decided to avoid it because it can make the semantics very confusing in cases of early bail out (you can only return `null`, but `null` may also be a valid return value if processing goes all the way and also if the processing involves launching asynchronous operations). I decided to keep it simple and let the caller manage the results processing.

Another important issue is making sure currently held locks are always released even if an exception is thrown. `ChainLocker` does it for you by managing a list of held locks and wrapping the processing in `try-finally` block, where all held locks are released in the `finally` block. The hand-over-hand locking strategy of `ChainLocker` means that up to two locks may be held at any point in the processing. Here is what it looks like:

```
if (state0 == null)
{
    throw new Exception("state0 can't be null");
}

var takenLocks = Enumerable.Repeat(false, 3).ToArray();
var states = Enumerable.Repeat<object>(null, 3).ToArray();
states[0] = state0;
try
{
    ...
}
finally
{
    // Release still held locks (If an exception was thrown)
    for (int i = 0; i < 3; ++i)
    {
        if (states[i] != null && takenLocks[i])
        {
            Monitor.Exit(states[i]);
        }
    }
}
```

(Please ignore the hard-coded number `3` in the code: This is generated code, so the [DRY principle](#) is not violated. When I write code manually, I don't hard-code the constant; but in generated code, it is acceptable and saves space.)

In the aforementioned code, the `takenLocks` variable is a Boolean flag array initialized to `[false, false, false]`. This means that, at this point, no lock is taken. The `states` variable is an array of objects that represent the objects that are locked/unlocked during processing. During normal processing, the locks will be released by `ChainLocker` in the hand-over-hand fashion; but if an exception is thrown, the `finally` block will unlock any remaining state objects that are still locked. It is crucial that the order in which the locks are taken by all threads is identical to avoid deadlocks.

Let's look at the actual processing done by `ChainLocker` inside the `try` block:

```
// Lock state0
Monitor.Enter(states[0], ref takenLocks[0]);
// Execute stage0
states[1] = stage0((T0)(states[0]));
```

```
    // Bail out if returned null
    if (states[1] == null)
    {
        return;
    }

    // Lock state1
    Monitor.Enter(states[1], ref takenLocks[1]);
    // Execute stage1
    states[2] = stage1((T1)(states[1]));
    // Bail out if returned null
    if (states[2] == null)
    {
        return;
    }

    // Lock state2
    Monitor.Enter(states[2], ref takenLocks[2]);
    // Release the state1 lock so other threads can work on it
    Monitor.Exit(states[1]);
    takenLocks[1] = false;
    // Execute stage2
    stage2((T2)(states[2]));
```

The processing for each `stage` is similar (identical except that the last `stage` doesn't do an early bail out check). In each `stage x`, the corresponding `state` object `states[x]` is locked and the Boolean flag `takenLocks[x]` is set by `Monitor.Enter()`. Then, the `stage x` itself is executed (casting the `state` object to its actual `Tx` type), and the result is stored in `states[x+1]`. If the result is `null`, `Do()` simply returns (the `finally` block will clear the held lock).

Concrete ChainLocker Subclass

`ChainLocker` is great, but if you misuse it, you can enter a deadlock. Consider the following two methods using a two-level deep `ChainLocker`:

```
void Foo(A a)
{
    ChainLocker<A, B>.Do(a, () => { return a.LookupChild() }, () => { ... });
}


void Bar(B b)
{
    ChainLocker<B, A>.Do(b, () => { return b.Parent() }, () => { ... });
}
```

Assume the hierarchical data structure is "`A` contains a list of `B` objects." The `Foo()` method works in the order `A -> B`. But the `Bar()` method works in the order `B -> A`. If `Foo()` and `Bar()` are called from two separate threads, you may easily get into a deadlock where each method is trying to lock an object currently locked by the other thread. The generic `ChainLocker` will not protect you from this situation; but a trivial subclass/specialization can do it. Consider the following subclass for the MS:

```
public class SchoolLocker : ChainLocker<IDictionary<int, Lecture>, Lecture, Class>
{
}
```

Using `SchoolLocker` is less verbose than using the generic `ChainLocker` because you don't have to specify the parameterized types in each call and, of course, it guarantees that the locks are taken in the correct order.

```
public AttendLecture(int lectureId, int studentId)
{
    SchoolLocker.Do(
        _lectures,
        lectures =>
        {
            Lecture lecture = null;
            lectures.TryGetValue(lectureId, lecture);
            return lecture;
        },
        lecture => lecture.FindAvailableClass(),
        availableClass => availableClass.AddStudent(studentId));
}
```

Sharing State and Stage Isolation

In the `AttendLecture()` method shown in the last code snippet, I used the `SchoolLocker` subclass. It ensures only that the locks `ChainLocker` takes itself are taken in the right order. However, each `stage` delegate has access to its outer scope, and because it is defined inside a method, it also has access to the entire object state (including the root). This is very convenient if you want all stages to have access to some shared state or object, such as a logger. But it also allows the `stage` delegates to ignore all the nice machinations of the `ChainLocker` and potentially wreak havoc on the system. One way to minimize this risk is to define the `stage` delegates as static methods instead of in-place anonymous methods. This way, each delegate will have access only to the state variable passed to it by `ChainLocker` and to static variables of the hosting class. If you don't want to expose even static variables, you can define the delegates in a separate class altogether. Here is what it looks like:

```
private static Lecture LookupLecture(IDictionary<int, Lecture> lectures, int lectureId)
{
    Lecture lecture = null;
    lectures.TryGetValue(lectureId,  out lecture);
    return lecture;
}
```

```
private static Class FindAvailableClass(Lecture lecture)
{
    return lecture.FindAvailableClass();
}

private static void AddStudent(Class availableClass, int studentId)
{
    availableClass.AddStudent(studentId);
}


public void AttendLecture(int lectureId, int studentId)
{
    new SchoolLocker().Do(
        _lectures,
        lectures => LookupLecture(lectures, lectureId),
        lecture => FindAvailableClass(lecture),
        availableClass => AddStudent(studentId));
}
```

Another benefit of this approach is that stages that are used by several methods (like `LookupLecture()`) can be reused. What about sharing something between all stages (like a logger) without exposing the entire outer scope? There is a special version of `ChainLocker` that accommodates this scenario. It's called `StateLockerEx` and has an extra argument called `shared` that is passed to each `stage` function. Here is `ChainLockerEx` for a two-level deep hierarchy:

```
    public class ChainLockerEx<T, T0, T1>
        where T0 : class
        where T1 : class
    {
        T _sharedState;
        public ChainLockerEx(T sharedState)
        {
            _sharedState = sharedState;
        }

        public void Do(T0 state0,
                       Func<T, T0, T1> stage0,
                       Action<T, T1> stage1)
        {
            if (state0 == null)
            {
                throw new Exception("state0 can't be null");
            }

            var takenLocks = Enumerable.Repeat(false, 2).ToArray();
            var states = Enumerable.Repeat<object>(null, 2).ToArray();
            states[0] = state0;
            try
            {
                // Lock state0
                Monitor.Enter(states[0], ref takenLocks[0]);
                // Execute stage0
                states[1] = stage0(_sharedState, (T0)(states[0]));
                // Bail out if returned null
                if (states[1] == null)
                {
                    return;
                }

                // Lock state1
                Monitor.Enter(states[1], ref takenLocks[1]);
                // Release the state1 lock so other threads can work on it
                Monitor.Exit(states[0]);
                takenLocks[0] = false;
                // Execute stage1
                stage1(_sharedState, (T1)(states[1]));
            }
            finally
            {
                // Release still held locks (If an exception was thrown)
                for (int i = 0; i < 2; ++i)
                {
                    if (states[i] != null && takenLocks[i])
                    {
                        Monitor.Exit(states[i]);
                    }
                }
            }
        }
    }
```

ChainLockerGenerator: A Python Script to Generate a Customized ChainLocker

`ChainLocker[Ex]` is generic, but does require some customization in regard to the depth of the hierarchy it needs to support and to address whether you want `ChainLocker` (no shared state) or `ChainLockerEx` (with shared state). You may opt create one file with many variations or just the one particular configuration you need. I decided to create a little Python program that can generate any combination of `ChainLockers` based on a few text templates. The added benefit is that if I decide to add a new feature or modify the design, I don't have to go and edit all the instances manually. I can edit just the templates and regenerate everything. For example, if I decide that the first level lock should be `ReaderWriterLockSlim` instead of the standard `Monitor`, I can add this option to the script and generate any combination of locks for my `ChainLocker` instances. The code and templates for ChainLockerGenerator are available on GitHub. Here is the usage message that explains how to use it:

```
"""
Usage: python ChainLockerGenerator.py <namespace> <N> [kind]

namespace - The C# namespace of your project
N - the maximal number of stages to generate
Kind - one of standard, extended, both

ChainLockerGenerator generates a C# file that contains multiple generic ChainLocker classes.
If you don't know what that is you have no business running this script :-)
There are two variants of chain lockers: standard and extended. The extended one provides
a shared state that is not locked to the stage operations.

Each generated instance has a certain number of stages that are locked. All instances
from 2 to N will be generated. For example, if you specified N=4 then 3 instances will be
generated with 2, 3 and 4 stages.

If you specified Kind=standard (or omitted it) only the standard instances will be generated.
If you specified Kind=extended only the extended instances will be generated.
If you specified both then you get both standard and extended. Everything is printed out
to standard output as a single C# module with the namespace you chose. The standard instances
are named ChainLocker<T1,...,Tn>. The extended instances are named ChainLockerEx<T, T1,...,Tn>
"""
```

Implementing with ChainLocker

Let's implement a few operations of the online school with `ChainLocker` and discuss it from a concurrent programming point of view. Before I go on, remember that the purpose of this code is to demonstrate how to use `ChainLocker`. It is not industrial strength and is not part of any real-world system.

Listing Two contains the abstract object model of the school's central scheduling service. It consists of an `ILecture` interface that contains `classes`, an `IClass` interface that contains `students`, a `Student` class with and associated `ID` and the `IClass` it attends, and a `Status enum` shared by `lectures` and `classes` (`LIVE` or `CANCELLED`).

Listing Two

```
using System.Collections.Generic;

namespace LectureManager
{
    public enum Status
    {
        LIVE,
        CANCELLED,
        REMOVED
    }

    public interface ILecture
    {
        int ID { get; set; }
        Status Status { get; set; }
        IEnumerable<IClass> Classes { get; }

        IClass FindAvailableClass();
        IClass LookupClass(int classId);
        void RemoveClass(int classId);
        void AddClass(IClass theClass);
    }

    public interface IClass
    {
        int LectureID { get; set; }
        int ID { get; set; }
        Status Status { get; set; }
        IEnumerable<int> Students { get; }

        void AddStudent(int studentId);
        void ExpelStudent(int studentId);
    }

    public class Student
    {
        public int ID { get; set; }
        public int ClassID { get; set; } // attending class
    }

    public class Stats
    {
        public int TotalClasses;
        public int TotalStudents;
    }
}
```

Removing a Student

A student may be removed from a class for one of several reasons: The lecturer decides to expel the student, the entire class is cancelled, or the student joins a new lecture/class. The service doesn't really care. From concurrency point of view, it's important to lock the class that the student is removed from because the class manages the student list. If multiple students need to be removed from the same class (say, if the class is cancelled), it makes sense to lock the class once and remove all the students instead of locking out each student. The `IClass` interface has an `ExpelStudent()` that handles all the mundane details like removing the student from the student list and notifying other interested parties. This method can be called by several other methods that are responsible for locking (via `SchoolLocker`, of course) at the right granularity. Here is the code for the external `ExpelStudent()` method. Notice its concision:

```
public void ExpelStudent(int lectureId, int classId, int studentId)
{
    new SchoolLocker().Do(
        _lectures,
        lectures => LookupLecture(lectures, lectureId),
        lecture => lecture.LookupClass(classId),
        theClass => theClass.ExpelStudent(studentId));
}
```

The first delegate uses the `LookupLecture()` method to look for the proper lecture, locking the entire lectures tree just for the brief moment it takes to look up the class. The second delegate finds the proper class (again locking the lecture just for the brief moment needed to find the class). Finally, the third delegate actually expels the student.

Canceling a Class

Canceling a class is a little more complicated. The service needs to expel all the students and then get rid of the class itself (remove it from the `class` list managed by the `lecture`). This means that it needs to lock the `lecture` when removing the `class`. One way to do it is to lock the `lecture`, expel all the `students`, and finally remove the `class`. But that means locking the `lecture` for a relatively long time, which will block all threads that may want to access other classes. Another approach is to find the `class`, release the `lecture`, and then set the `class` status to `CANCELLED` first (thus unlocking the `lecture`), and then proceed to expel all the students. Other threads may work with other `classes` of this `lecture`. Once all the students have been expelled, the schedule service can start a new `SchoolLocker` instance and (in its `Do()` method) remove the class. Note that once the first `SchoolLocker.Do()` method completes, the cancelled class will be unlocked, and other threads may try to access the class before it is removed. This is OK because its status is CANCELLED, so it will not be available to other threads.

```
public void CancelClass(int lectureId, int classId)
{
    // Set the class status to CANCELLED
    new SchoolLocker().Do(
        _lectures,
        lectures => LookupLecture(lectures, lectureId),
        lecture => lecture.LookupClass(classId),
        theClass =>
        {
            theClass.Status = Status.CANCELLED;
            foreach (var studentId in theClass.Students)
            {
                theClass.ExpelStudent(studentId);
            }
        });

    // Remove the cancelled class from its lecture
    new SchoolLocker().Do(
        _lectures,
        lectures => LookupLecture(lectures, lectureId),
        lecture =>
            {
                lecture.RemoveClass(classId);
                return null;
            },
        null);
}
```

Canceling a Lecture

Canceling a `lecture` is very much like canceling a class except that in order to remove the lecture, we need to cancel all the classes properly and expel all the students because there may be other parts of the system that depend on orderly cancellation. We must ensure that once the lecture is cancelled, each of its classes is locked directly (to avoid conflicts with other threads that might have started working on a class before the lecture was cancelled) and all its students are expelled. Finally, the lecture is removed from the lectures tree (with a simple lock):

```
public void CancelLecture(int lectureId)
{
    ILecture cancelledLecture = null;

    // Set the lecture status to CANCELLED
    new SchoolLocker().Do(
        _lectures,
        lectures => LookupLecture(lectures, lectureId),
        lecture =>
        {
            lecture.Status = Status.CANCELLED;
            cancelledLecture = lecture;
            return null;
        },
        null);

    // Now cancel all the classes and expel all their students
    foreach (var theClass in cancelledLecture.Classes)
```

```
        {
            lock (theClass)
            {
                theClass.Status = Status.CANCELLED;
            }

            foreach (var studentId in theClass.Students)
            {
                theClass.ExpelStudent(studentId);
            }
        }

        // Finally, remove the lecture from the lectures dictionary
        lock (_lectures)
        {
            _lectures.Remove(lectureId);
        }
    }
```

Getting Statistics

Getting statistics out of a highly multithreaded service involves trade-offs. You can lock the whole system and accumulate your statistics: This will give you an exact snapshot, but will freeze your system for the duration of statistics collection. Alternatively, you can iterate over your data structures without locking and know that you collect data from a system in flux, and by the time you finish, some of the statistics will already be stale. For example, suppose you just want to count how many students attend classes at a given moment. If you just iterate over all the classes and count their students without locking, you may count students that were expelled by the time you finish your count, but also include students that attended a class after you started your count. You may even count the same student twice if that individual switched classes while you were counting. There are other approaches, such as copying your entire state and calculating your statistics off the copy. That makes sense if your data structures don't take much space, but is inefficient if your statistics computations are sophisticated and take a relatively long time to compute.

Here, I implement the simplest approach of just locking everything and counting classes and students. There is no need for `SchoolLocker` in this scenario because we lock the entire lectures tree, so a simple lock will suffice:

```
public Stats GetStatistics()
{
    var stats = new Stats();

    lock (_lectures)
    {
        foreach (var lecture in _lectures.Values)
        {
            foreach (var theClass in lecture.Classes)
            {
                stats.TotalClasses += 1;
                stats.TotalStudents += theClass.Students.Count();
            }
        }
    }

    return stats;
}
```

Conclusion

Concurrent programming will become ubiquitous as the number of cores continues to increase and more threads are expected to execute in parallel. To take advantage of all these cores, you'll have to make sure your code is thread-safe and doesn't use too coarse a locking strategy. The `ChainLocker` construct can assist you by providing an abstraction of safe fine-grained locking for hierarchical data structures that hides most of the gnarly locking details.

Gigi Sayfan specializes in cross-platform object-oriented programming in C/C++/ C#/Python/Java with emphasis on large-scale distributed systems, and is a long-time contributor to *Dr. Dobb's*.

More on Coarse-Grained versus Fine-Grained Parallelism

Will Parallel Code Ever Be Embraced?

Parallel Evolution, Not Revolution