Fall 2016

MIGRATING THREAD-BASED INTENTIONAL CONCURRENT PROGRAMMING TO A TASK-BASED PARADIGM

Seth Adam Hager
University of New Hampshire, Durham

MIGRATING THREAD-BASED INTENTIONAL CONCURRENT PROGRAMMING TO A TASK-BASED PARADIGM

BY

Seth Hager

B.M., University of Massachusetts Lowell, 2004

THESIS

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Master of Science

in

Computer Science

September 2016

This thesis has been examined and approved in partial fulfillment of the requirements for the degree of Master of Science in Computer Science by:

Thesis director, Philip J. Hatcher, Professor of Computer Science

Michel H. Charpentier, Associate Professor of Computer Science

R. Daniel Bergeron, Professor of Computer Science

August 16th, 2016

Original approval signatures are on file with the University of New Hampshire Graduate School.

DEDICATION

For Lily and Jacob.

ACKNOWLEDGMENTS

I would like to thank the members of my committee for all of their time and effort.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

MIGRATING THREAD-BASED INTENTIONAL CONCURRENT PROGRAMMING TO A TASK-BASED PARADIGM

by

Seth Hager
University of New Hampshire, September, 2016

Parallel programming is no longer restricted to supercomputers and academia. From the decline of Moore's law with respect to CPU clock speed, hardware designers have introduced multi-core CPU architectures in commodity hardware. Consequently software engineers can not expect to reap the performance benefits of such target hardware without exploiting parallelism. Unfortunately, writing multi-threaded software comes at a cost. The costs are two-fold: the amount of time invested to parallelize software is greater than that of writing single-threaded programs, and the safety of multi-threaded software is difficult to ensure. Multi-threaded software allows threads to operate in the same memory-address space. When more than one thread has access to the same memory address, an unexpected interleaving of memory accesses can lead to subtle concurrency bugs. Intentional Concurrent Programming is a model that requires programmers to explicitly state their intents with respect to threads sharing objects in memory. Concurrency bugs will be detected and reported as violations of the stated intents. The current implementation of this model, known as V3, is thread-based, meaning that users deal directly with individual threads. The goal for this thesis is to generalize the model to be task-based. This abstraction not only allows the user to view the program as a collection of tasks to be executed by a pool of threads, but also removes the need to deal directly with individual threads. The hypothesis is that intents can be added to a task-based system, and common task-based applications can be expressed with the resulting system.

CHAPTER 1

PROBLEM STATEMENT

"A class is *thread-safe* if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code." [1]

Writing and maintaining correct concurrent software is difficult due to the need to reason about synchronization. The coordination of threads and shared data is known as synchronization. Programming languages allow the programmer to write under-synchronized programs. A multi-threaded program that is incorrectly synchronized may appear to be intermittently broken. Furthermore, the difficulty is magnified because synchronization is implicit. Consider that there is some shared data and multiple threads. This data must be protected by a synchronization construct on all accesses by any thread. The relationship between the shared data and the synchronization construct is implicit. The programmer must ensure that the relationship is upheld throughout the lifetime of the shared data. Adding to these complications, this implicit relationship must be documented for the benefit of future code maintainers. However, this documentation does not always exist. Implicit relations are a major source of why finding concurrency bugs can be so difficult. Once a concurrency bug is found, it is often the case that it is not fixed on the first try [2]. These bugs have real costs associated with them: programmer time, company money, and the breakdown of large-scale, important systems [3, 4].

An explicitly expressed relationship between some shared data and a synchronization construct is an *intent*. In other words, intents are expressed with respect to how the data will be shared

among threads. Using an intent-based programming framework will assist programmers in writing correct code. Once explicit relationships are drawn by expressing intents, the framework will ensure that these relationships are upheld. If there is an attempted access to shared data that is contrary to the expressed intent, the framework will notify the programmer. Expressing intents not only upholds explicit relations between shared data and synchronizers, but also encodes a type of self-documentation in the program that is apparent for future maintainers [5]. It is important to be able to enforce concurrent access rules in the program, and not just document them [1].

In order to keep up with current programming patterns and models, it is desirable to incorporate task-based concurrency into an intentional concurrent framework. The task-based model uses a collection of threads known as a thread-pool. Potentially heterogeneous tasks, or units of work, are run by worker threads in the pool. This decouples the work from the thread, allowing the programmer to submit a number of tasks that is greater than the number of threads in the pool. Extra tasks will sit in a queue until there is an available thread. Tasks can be as coarse, or fine-grained, as the programmer decides. Thread-pools enable a system to process several requests without running out of resources due to excessive thread creation. This is known as degrading gracefully. Alternatively the programmer can still create a one-to-one relationship from task to thread, effectively recreating the thread-based model.

Intentional Concurrent Programming [6, 7] is an implementation of a thread-based, object-oriented, intentional programming framework. The contribution of this project is the migration of thread-based Intentional Concurrent Programming to task-based. The contribution includes an implementation of new constructs, as well as modification of the existing code base. In order to evaluate that the system is expressive enough, new applications and updates to existing applications have been carried out using the publicly exported Intentional Concurrent Programming API. Through this work, it is shown that a system exists which is expressive enough to support Intentional Concurrent Programming using a task-based model.

CHAPTER 2

RELATED WORK

2.1 Java Concurrency Library

The Java programming language offers a rich concurrency library in the java.util.concurrent package. There are several high-level thread-safe constructs that are available. The package offers synchronizers such as locks, semaphores, and barriers. It also contains data structures, from concurrent maps and sets, to blocking queues. The atomic portion of the package includes a collection of single variables that support atomic operations. Finally, the package offers thread-pools to run user-defined tasks. Due to widespread usage, and task-based support, the Java concurrency library is a good fit for Intentional Concurrent Programming.

An example is used throughout this document. The example uses a shared counter between threads. The shared counter is protected by a lock. Each user-defined work-load is called a Job, and uses its own counter. Each time the shared counter is increased the Job's counter is increased. When the Job's counter reaches 500, the Job is complete. Two Jobs are created, so the shared counter's total is expected to be 1,000. The Counter is shown in Figure 2-1. The Job is shown in Figure 2-2, and the application is shown in Figure 2-3. This example is written in pure Java.

Figure 2-1: Counter.

```
1  class Counter {
2
3      private int value;
4
5      void increment() { value += 1; }
6
7      int get() { return value; }
8  }
```

Figure 2-2: Job.

```
class Job implements Runnable{

    private final Counter myCount;
    private final Counter counter;
    private final ReentrantLock lock;
    private final CountDownLatch latch;

    Job(Counter counter, ReentrantLock lock, CountDownLatch latch){
        this.counter = counter; this.lock = lock; this.latch = latch;
        this.myCount = new Counter();
    }

    @Override
    public void run(){ work(); latch.countDown(); }

    private void work(){
        while(myCount.get() != 500) {
            lock.lock(); counter.increment(); lock.unlock();
            myCount.increment();
        }
    }

    public int getMyCount(){ return myCount.get(); }
}
```

Figure 2-3: Application.

```
final Counter counter = new Counter();
final ReentrantLock lock = new ReentrantLock();
final CountDownLatch latch = new CountDownLatch(2);

Job one = new Job(counter, lock, latch); Job two = new Job(counter, lock, latch);

// build thread-pool, submit jobs, then shutdown.
int nThreads = Runtime.getRuntime().availableProcessors() > 1 ? 2 : 1;
ExecutorService ex = Executors.newFixedThreadPool(nThreads);
ex.submit(one); ex.submit(two); ex.shutdown();

latch.await();

lock.lock();
int counterCount = counter.get();
lock.unlock();
int twoCount = two.getMyCount();
int oneCount = one.getMyCount();

assert oneCount == 500 && twoCount == 500 && counterCount == 1_000 :
"One: " + oneCount + " Two: " + twoCount + " Counter: " + counterCount;
```

Consider the synchronization policy in Table 2.1. Each description outlines a part of the policy. The example program upholds each part of the policy and produces correct results. Each code change, as outlined in the third column, represents a deviation from the policy. The result column demonstrates program behavior based on the deviation. If the result is acceptable a check mark is in the last column.

Part	Description	Code Change	Result	Acceptable Result
1	The *lock* must be held on all accesses to the shared *counter*.	Don't use *lock* in *Job*.	data race on *counter*	
2	Calling *Job.getMyCount* happens only after *latch* opens.	Call *Job.getMyCount* before *latch* opens.	data race on *myCount* of *Job*	
3	Only one thread ever runs a *Job* at a time, and it is run before *latch* opens.	Run *Job one* directly from main after submitting it to the thread-pool.	data race on *myCount* of *Job*	
4	Calling *get* on the shared *counter* happens only after *latch* opens.	Call *counter.get* before *latch* opens.	data race on *counter*	
5	Number of threads in the thread-pool can be 1 or 2.	Explicitly set *nThreads* to 1.	Works as expected	✓

Table 2.1: Synchronization Policy

2.2 Intentional Concurrent Programming

Intentional Concurrent Programming is currently in its third iteration, and is named V3 [8]. V3 is written in Java, and uses a third-party library called Javassist [9] for on-the-fly byte-code editing. In V3, all user-defined classes are instrumented using Javassist. The class files are edited in order to check that an intent is upheld when an object is used. An intent is expressed by explicitly associating an object with a synchronizer. This association is called *registration*. A synchronizer protects specific members of an object. These members are identified through annotations. When a user expresses their intent of how to share an object, that intent must be checked at runtime. In order to check the intent, the fulfillment of a contract is confirmed before access to a member

is granted. A contract is simply an agreement that states: in order for a thread to use an object, that thread must have some specific permission.

Until now, V3 has been a thread-based system. This means that contracts were thread-based, checking to see if the calling thread is granted access to the member. For example, consider that a programmer has expressed an intent that a lock, L, must be held before accessing a field, F. When any thread, T, tries to access F, a contract must be fulfilled before access is granted. The contract agreement is: if T has a hold on L, then let T access F, otherwise enter an error state.

Figures 2-4, 2-5, and 2-6 show the example program using thread-based Intentional Concurrent Programming. The thread-pool is a construct from the Java concurrency library, as in Figure 2-3. The reader can draw from the example that a contract can be associated with a synchronizer, and a member of an object. It is worth noting that creating a new object implicitly expresses an intent. This initial intent is upheld by a "private" contract. The agreement is: in order to access a member of the new object, the calling thread must be the creator of the object. In the example, when the intent is expressed that a specific lock must protect access to a specific field, the "private" contract is replaced with a new "lock" contract. Replacement is allowed if the current contract is a "private" contract that can be fulfilled by the calling thread. Thread-based Intentional Concurrent Programming is the starting point for the work described in this document.

Figure 2-4: Thread-Based Intentional Concurrent Programming Counter.

```
1  class Counter {
2      // Annotations are used to identify object members.
3      // `@GetShared' is used for field reads `@PutShared' is used for field writes.
4      @GetShared("ThreadSafe") @PutShared("ThreadSafe")
5      protected int value;
6      // `@Shared' is used for methods.
7      @Shared("Lock") void increment() { value += 1; }
8      // forward call to `getHelp' in order to add a synchronization constraint.
9      @Shared("Lock") int get() { return getHelp(); }
10
11     @Shared("After") private int getHelp() { return value; }
12 }
```

Figure 2-5: Thread-Based Intentional Concurrent Programming Job.

```
class Job implements Runnable{

    // Override methods of Counter to change their annotations.
    @GetShared("ThreadSafe")
    private final Counter myCount = new Counter(){
        @Override @Shared("ThreadSafe")
        void increment() { sharedData += 1; }

        @Override @Shared("ThreadSafe")
        int get() { return sharedData; }
    };

    @GetShared("Lock") private final Counter counter;

    @GetShared("ThreadSafe") private final ReentrantLock lock;

    @GetShared("ThreadSafe") private final CountDownLatch latch;

    Job(Counter counter, ReentrantLock lock, CountDownLatch latch){
        this.counter = counter; this.lock = lock; this.latch = latch;
        ThreadSafe.get().register(myCount);
    }

    @Override
    @Shared("ThreadSafe")
    public void run(){
        // latch.start expresses an intent that the caller is ready
        // to access members that are enabled before the latch is open.
        latch.start(); work(); latch.countDown();
    }

    @Shared("Before")
    private void work(){
        while(myCount.get() != 500) {
            lock.lock(); counter.increment(); lock.unlock();
            myCount.increment();
        }
    }

    @Shared("After")
    int getMyCount(){ return myCount.get(); }
}
```

Figure 2-6: Thread-Based Intentional Concurrent Programming Application.

```
final Counter counter = new Counter();
final ReentrantLock lock = new ReentrantLock();
final CountDownLatch latch = new CountDownLatch(2);

// Notice all paths to the shared counter's mutable state
// are protected by synchronization constructs.

// Express intent to protect members of `counter' that are
// annotated with "Lock" using the `lock' instance of ReentrantLock.
lock.register(counter, "Lock");

// Express intent to allow access by any thread to members of `counter'
// that are annotated with "ThreadSafe"
ThreadSafe.get().register(counter);

// Express intent to restrict access for all threads to members of `counter'
// that are annotated with "After" until after `latch' has opened.
latch.register(counter, null, "After");

Job one = new Job(counter, lock, latch); Job two = new Job(counter, lock, latch);
ThreadSafe.get().register(one); ThreadSafe.get().register(two);
lock.register(one, "Lock"); lock.register(two, "Lock");
// Express intent to enable members of `one' and `two' annotated with
// "Before" only before the latch is open, and annotated with "After" only after the latch is open.
latch.register(one, "Before", "After"); latch.register(two, "Before", "After");

Runtime.getRuntime().availableProcessors() > 1 ? 2 : 1;
ExecutorService ex = Executors.newFixedThreadPool(nThreads);
ex.submit(one); ex.submit(two);
ex.shutdown();

latch.await();

lock.lock();
int counterCount = counter.get();
lock.unlock();

int oneCount = one.getMyCount();
int twoCount = two.getMyCount();

assert oneCount == 500 && twoCount == 500 && counterCount == 1_000 :
"One: " + oneCount + " Two: " + twoCount + " Counter: " + counterCount;
```

Table 2.2 revisits the synchronization policy. Policy parts 1, 2 and 4 have an acceptable outcome. Since the code change is contrary to the synchronization policy, there is an exception thrown. The policy is embedded in the program as opposed to just residing in documentation. This is not successful for part 3, and 5. Part 3 does not have an acceptable result because there is a data race on the *myCount* variable of *Job one*. In order to solve this problem there needs to

be a way for one thread to safely create an object and delegate its usage to only one other thread. Part 5 does not have an acceptable result either. This is because when *nThreads* has a value of one an unexpected exception is thrown. The exception is thrown on line 27 of Figure 2-5. This is due to contracts being thread-based. The contract of CountDownLatch uses a Java ThreadLocal to keep track of valid state transitions. When execution of the *Job* instances is serialized, the state information from *Job one* is left over in the pooled thread when *Job two* is started. We need a way to translate contracts to per task as opposed to per thread.

Part	Description	Code Change	Result	Acceptable Result
1	The *lock* must be held on all accesses to the shared *counter*.	Don't use *lock* in *Job*.	edu.unh.cs.v3.IntentError: current thread does not own lock	✓
2	Calling *Job.getMyCount* happens only after *latch* opens.	Call *Job.getMyCount* before *latch* opens.	edu.unh.cs.v3.IntentError: latch contract violation	✓
3	Only one thread ever runs a *Job* at a time, and it is run before *latch* opens.	Run *Job one* directly from main after submitting it to the thread-pool.	data race on *myCount* of *Job*	
4	Calling *get* on the shared *counter* happens only after *latch* opens.	Call *counter.get* before *latch* opens.	edu.unh.cs.v3.IntentError: latch contract violation	✓
5	Number of threads in the thread-pool can be 1 or 2.	Explicitly set *nThreads* to 1.	edu.unh.cs.v3.IntentError: start method cannot be called after countdown	

Table 2.2: Thread-Based Intentional Concurrent Programming Synchronization Policy

2.3 Other Concurrency Libraries

Thread-pools, tasks, and rich APIs that support high-level concurrency are not exclusive to Java, or languages implemented on top of the JVM. Python 3, has its own concurrent package with thread-pools [10]. C# and .NET depend on concurrency and offer very similar constructs [11]. The Boost library for C++ offers thread-pools, while C++11 offers constructs for high-level concurrent programming [12]. The task and thread-pool pattern is language independent, and common to

concurrent programming.

2.4 Actors

Actors is an alternate model for concurrent programming [13]. Akka is an implementation of the model, and offers an implementation in Java [14]. This implementation uses the Java ForkJoinPool, a thread-pool, under the hood for Actors that are run on a single machine. Akka is a higher level of abstraction for concurrent programming that can sit on top of a thread-pool. Actors is a conceptually "clean" model, but there is still the requirement of synchronization in an implementation. The model requires actors to encapsulate their internal, potentially mutable state. An actor must send a message to another actor in order to communicate. If a message includes a reference to an actor's internal mutable state, the system can break down without synchronization. The Akka implementation requires that a message be immutable. Sharing immutable objects among threads requires no synchronization [1]. This requirement is by convention only. There is no compile-time or runtime mechanism that enforces this property. Actors models like Akka may benefit from the task-based Intentional Concurrent Programming framework. These potential benefits are discussed in Chapter 6.

CHAPTER 3

DESIGN

This section describes the major design components of the task-based Intentional Concurrent Programming system. Each sub-section describes what the component is, and why it is part of the design. Components of the thread-based implementation were updated in order to work with the task-based system. These components are described in the last sub-section.

3.1 Task

A task is the central construct in the task-based system. As mentioned in Chapter 1, it is a unit of work. The main purpose of a task is to decouple a unit of work from a thread. This allows a thread-pool to run a group of tasks when there are more tasks than threads. Each thread takes a task off a queue, runs the task, then returns to get another task from the queue. Threads in the pool repeat this process until the pool is shut down.

Using a thread-pool helps a system manage resources. Consider a system that takes unrelated requests and gives some response to them. If the system efficiently uses resources to keep up with requests, it processes the requests in parallel, provided the proper hardware is available. Consider that the system spawns a new thread for each request. When the request volume reaches some threshold, there will be a point of diminishing returns in terms of response time. In fact the system may make no progress due to being saturated with requests. Saturation results in resource exhaustion. A way to mediate the problem of over-subscription is to build tasks from requests. These tasks sit in a queue until there is an available thread to work on the request. This lets the response time of the system degrade gracefully.

In the context of the task-based system, intents are expressed per task, as opposed to per thread. As part of the task-based Intentional Concurrent Programming framework, a task fulfills a contract as opposed to a thread. The agreement of a contract now is: in order for a task to use an object, that task must have some specific permission. Tasks have a set of properties:

- A task can start another task, allowing it to temporarily take on a new behavior. When complete, it returns to the previous task.
- Tasks can only be started once by one other task.
- A task can be blocked, waiting for access to a member of an object.
- When blocked, the underlying thread that is running the task is also blocked.
- Tasks can produce a result that can only be returned to the parent task.
- Nothing bars a task from emitting side effects.

3.2 TaskLocal

If an object is only used by one thread for that object's life-time, the object is known to be thread-confined. "When an object is confined to a thread, usage is automatically thread-safe even if the object itself is not" [1]. Java offers a class called ThreadLocal that provides thread-local variables. These thread-local variables are thread-confined. Taking this a step further, only letting a single task use an object makes it task-confined. TaskLocal provides task-local variables in much the same way that ThreadLocal provides thread-local variables. This finer-grained mechanism is useful when per-task mutable state needs to be maintained. When a task is complete, all of its TaskLocal variables can be garbage-collected. Using TaskLocal with thread-pools is more amenable than ThreadLocal. Consider the system that takes unrelated requests and provides a response to them. If ThreadLocal is used to maintain a per-thread state, at the end of each request all of the ThreadLocal variables must be explicitly cleaned up. If they are not cleaned up, and a new request is made, state information is left in the ThreadLocal from a prior request.

3.3 Transfer

There is a necessary idiom in multi-threaded programming that is referred to as *create-and-delegate*. The responsibility of creating objects and using them is split between two threads. An example of create-and-delegate comes from the producer-consumer design pattern [15]. There can be any number of producers and consumers in this pattern. A producer creates objects and delegates their usage to another thread. This is done by the producer adding the newly created objects to a shared queue. The consumer takes the objects off the queue and works on them. Race conditions can occur if the shared objects are not thread-safe. A race can happen if the producer keeps and uses a reference to the newly created object after it has been added to the queue. The consumer could take the object off the queue, and start to work on it while the producer is still working on the same object.

A thread-safe way to share objects that are not thread-safe is needed to create-and-delegate. The object is delegated to exactly one other thread. If accesses to all members of an object are disabled when the object is in the queue, then no thread can access any member. A consumer thread takes the object off the queue enabling all members of the object. These members must now be accessible by only that consumer thread. This is the purpose of the Transfer object.

A Transfer is a container that keeps a reference to one object. Since it is a container, it requires a type parameter when declared, which is the type of the object that the Transfer will hold. When an instance of Transfer is created, members of the contained object that have a specified annotation are disabled. In other words the Transfer is closed. The Transfer is like a box: when the box is closed no one can use its contents; when it is open, someone can use the contents. In the task-based Intentional Concurrent Programming framework this means that no task can use any of these disabled members when the Transfer is closed. Once a task opens the Transfer, the content's specified members become usable by only the opening task.

Although this works well for simple objects, consider the use of a composite object that may have several layers of references. The root of the object graph is the object that the Transfer must contain. There is a facility to register other objects with the Transfer. This allows intents

to be expressed about children of the root object. The children's members with the corresponding annotation are then disabled after registration and re-enabled once the Transfer is open.

An extension to create-and-delegate continues to delegate as many more times as needed, similar to an assembly line. Consider a factory that makes wall clocks. The first worker creates clock frames and puts them on the assembly line for the next worker. The next worker adds all of the gears to the clock, then puts the clock back on the line for the following worker. Finally, the last worker puts the face and hands on the clock, finishing its construction. This style of work is made safe by using a new Transfer at each stage. A Transfer has a limit of one usage. It is created in a closed state and can only transition to an open state once. The clock factory example needs two Transfers, one to transition from the frame-maker to the gear-installer, and one to transition from the gear-installer to the finisher. A Transfer gives the guarantee that there is one worker making changes to the clock at a time. There is also a guarantee that each worker sees the clock in a consistent state. In other words, the state of the clock is the same for the worker that opens the Transfer as it was for the worker that built that Transfer.

There is one other benefit to using a Transfer to create-and-delegate. The programmer can express intents for the contents of the Transfer after it has been opened. It is as if the opener of the Transfer is the creator of the object. This is only for members that were associated with the Transfer in the first place.

3.4 ThreadPoolExecutor

Migrating from a thread-based system to a task-based system requires that a thread-pool be readily available to users. Java's concurrency library offers a thread-pool called ThreadPoolExecutor. This implementation is used as the underlying thread-pool in the task-based Intentional Concurrent Programming framework's ThreadPoolExecutor. This underlying thread-pool makes use of a producer-consumer design pattern. Consider a typical usage of the task-based framework: a main task creates instances of user-defined jobs, then submits them to a ThreadPoolExecutor. By submitting a job to the ThreadPoolExecutor the job will be placed in a queue for another task

to remove and start. Since this follows the create-and-delegate pattern, it is a requirement to put user-defined jobs into a Transfer before submission.

The task-based framework's ThreadPoolExecutor implements an ExecutorService interface, which is a limited, and slightly modified version of the one provided by the Java concurrency library. The task-based framework's interface enforces the requirement of submitting only Transfers. All Transfers that are submitted must be parameterized by a Callable of some type. Callable is an interface that is defined in the Java concurrency library. It is accepted as a task by the concurrency library's thread-pools and returns a value. The type of that value is the type parameter of the Callable. The interface also requires methods to shutdown, and terminate the thread-pool.

3.5 Executors

In the task-based Intentional Concurrent Programming framework a new thread-pool must be created via static factory methods of the Executors class. There are three available methods: *newFixedThreadPool(int n), newCachedThreadPool(), newSingleThreadExecutor()*. Each method returns a newly created and configured ThreadPoolExecutor. These methods are a subset of what is available in the Java concurrency library's version of Executors [1].

- The *newFixedThreadPool* static method takes an integer parameter, n, and returns a ThreadPoolExecutor that will use at most n Threads.
- The *newCachedThreadPool* static method returns a new ThreadPoolExecutor that creates new threads as needed, reusing threads when possible. There is no bound on the number of threads that can be created with a *newCachedThreadPool*.
- The *newSingleThreadExecutor* static method returns a new ThreadPoolExecutor that has at most one thread.

3.6 New Synchronizer

The submit method of an ExecutorService returns a Future of some parameterized type. A Future is a synchronizer that returns a result once it is ready. The *get* method of the Future blocks until the result is ready, then returns it. This is all based on how ExecutorService and Future work in the Java concurrency library.

The task-based Intentional Concurrent Programming framework provides a Future that offers two methods: *get* and *isDone*. The *isDone* method returns true if the Future has a result ready, and false otherwise. The *get* method returns the result, blocking the current task if needed. This method may only be called once. The result that is returned by the Future's *get* method substitutes "private" contracts as if the calling task is the creator of the result. This is another create-and-delegate pattern, and is further explained in Chapter 4. A Future may only be used by the task that is the creator of the Future.

3.7 Existing Synchronizers

Thread-based Intentional Concurrent Programming has several existing synchronizers: Disabled, ThreadSafe, ReentrantLock, ReentrantReadWriteLock, CountDownLatch, and Synchronized. All of these synchronizers are derived from existing Java synchronizers [1], except for ThreadSafe and Disabled. Each of these synchronizers has an associated contract that has been updated in order to migrate to the task-based framework. A short description of each follows.

ThreadSafe has a non-restrictive agreement; any task may access the member protected by the contract. Conversely, Disabled is completely restrictive. It does not let any task access the protected member under any conditions. It is an error to attempt to access a "disabled" member. A non-final field may be registered with Disabled after an initialization routine is complete. This is useful to render an object immutable. The migration for ThreadSafe and Disabled is trivial.

Next, is the ReentrantLock. If a member protected by the ReentrantLock's associated contract is to be accessed, the calling task must hold the lock. Using the calling task, as opposed to the calling thread, avoids a boundary case. If the calling thread is used, there is potential that one

task could hold the lock and terminate with the lock held. The same thread could run another task that is able to access the protected member without explicitly holding the lock, then release it.

ReentrantReadWriteLock, a solution to the reader-writer problem [16], uses two contracts. One contract is for the read side of the lock, and the other for the write side of the lock. In both cases, similar to the ReentrantLock, a task must hold the lock in order to access a member protected by this contract.

CountDownLatch uses two contracts as well. One protects members that should be accessible when the latch is in an open state, while the other protects members that should be accessible when the latch is in a closed state. In other words, some members of an object should be enabled only when the latch is closed, while others should be enabled only when the latch is open. There are additional intents that are expressed when using a latch. These intents are coupled with the state of the latch. A task calling the *start* method on the latch expresses an intent that closed members will be accessed by the calling task. Similarly, a task calling the *await* method of the latch expresses an intent that open members will be accessed. These constraints are all per task as opposed to per thread.

The Synchronized synchronizer uses intrinsic locks [1]. The contract is conceptually the same as the ReentrantLock. The monitor on the specified object must be held in order to access a member protected by the contract. Using synchronized methods and synchronized blocks in Java is thread-based. This complicates the migration of the Synchronized synchronizer to the task-based system. There is another boundary case that must be handled in the task-based system. Consider a task holds the specified monitor, then starts another task. The new task could access a member that is protected by this contract without first acquiring the monitor. The solution to this problem is discussed in Chapter 4.

Figures 3-1, 3-2, and 3-3 show the example program using task-based Intentional Concurrent Programming. This version of the example uses constructs as described in Chapter 3. All parts of the synchronization policy have an acceptable result as shown in Table 3.1. Parts 1, 2 and 4 still work as expected. What is most interesting is Part 3. The Transfer has provided the vehicle to

properly create-and-delegate Job instances from the main task to workers in the pool. Part 5 has shown the benefit of updating contracts to task-based. This is further discussed in Section 4.2. With all parts of the synchronization policy successfully embedded into the program, task-based Intentional Concurrent Programming has done its job.

Part	Description	Code Change	Result	Acceptable Result
1	The *lock* must be held on all accesses to the shared *counter*.	Don't use *lock* in *Job*.	edu.unh.cs.v3.lib.IntentError: current Task does not satisfy contract	✓
2	Calling *Job.getMyCount* happens only after *latch* opens.	Call *Job.getMyCount* before *latch* opens.	edu.unh.cs.v3.lib.IntentError: current Task does not satisfy contract	✓
3	Only one thread ever runs a *Job* at a time, and it is run before *latch* opens.	Run *Job one* directly from main after submitting it to the thread-pool.	edu.unh.cs.v3.lib.IntentError: running task does not have access to object	✓
4	Calling *get* on the shared *counter* happens only after *latch* opens.	Call *counter.get* before *latch* opens.	edu.unh.cs.v3.lib.IntentError: current Task does not satisfy contract	✓
5	Number of threads in the thread-pool can be 1 or 2.	Explicitly set *nThreads* to 1.	Works as expected	✓

Table 3.1: Task-Based Intentional Concurrent Programming Synchronization Policy

Figure 3-1: Task-Based Intentional Concurrent Programming Counter.

```
1  class Counter {
2
3      @GetShared("ThreadSafe") @PutShared("ThreadSafe")
4      private int value;
5
6      // Annotations are only used to identify members.
7      @Shared("increment") void increment() { value += 1; }
8
9      @Shared("get") int get() { return getHelp(); }
10
11     @Shared("getHelp") private int getHelp() { return value; }
12 }
```

Figure 3-2: Task-Based Intentional Concurrent Programming Job.

```
class Job implements Callable<Transfer<Job>>{

    @GetShared("myCount")  private final Counter myCount;
    @GetShared("Transfer") private final Counter counter;
    @GetShared("Transfer") private final ReentrantLock lock;
    @GetShared("Transfer") private final CountDownLatch latch;

    Job(Counter counter, ReentrantLock lock, CountDownLatch latch){
        this.counter = counter; this.lock = lock; this.latch = latch;
        this.myCount = new Counter();
    }

    // Register the individual Job counter with the same Transfer that the
    // Job will be boxed in. Note, all members of `myCount' are registered
    // with the Transfer.
    @Shared("myCount")
    public void register(Transfer<?> transfer){
        transfer.register(myCount, "increment"); transfer.register(myCount, "get");
        transfer.register(myCount, "getHelp"); transfer.register(myCount, "ThreadSafe");
        transfer.register(this, "myCount");
    }

    // Once work is complete return the `Job' instance back to
    // the creator of the `Job'. This is done by returning the
    // `Job' in a `Transfer'
    @Override @Shared("Transfer")
    public Transfer<Job> call(){
        latch.start(); work(); latch.countDown();
        Transfer<Job> toRtn = Transfer.newInstance(this, "Transfer");
        register(toRtn);
        return toRtn;
    }

    @Shared("Before")
    private void work(){
        while(myCount.get != 500) {
            lock.lock(); counter.increment(); lock.unlock();
            myCount.increment();
        }
    }

    // Another example of forwarding method calls to add
    // synchronization constraints.
    @Shared("Transfer")
    public int getMyCount(){ return getMyCountHelp(); }

    @Shared("After")
    private int getMyCountHelp(){ return myCount.get(); }
}
```

Figure 3-3: Task-Based Intentional Concurrent Programming Application.

```
final Counter counter = new Counter();
final ReentrantLock lock = ReentrantLock.newInstance();
lock.register(counter, "increment");
lock.register(counter, "get");
ThreadSafe.get().register(counter);

final CountDownLatch latch = CountDownLatch.newInstance(2);
Job one = new Job(counter, lock, latch); Job two = new Job(counter, lock, latch);
latch.register(one, "Before", "After"); latch.register(two, "Before", "After");
latch.register(counter, null, "getHelp");

// Build two `Transfer' instances using the `Job' instances.
// Both `Job' instances will have all members related to the
// `Transfer' disabled after construction and registration.
Transfer<Job> boxOne = Transfer.newInstance(one, "Transfer");
Transfer<Job> boxTwo = Transfer.newInstance(two, "Transfer");
one.register(boxOne); two.register(boxTwo);

int nThreads = Runtime.getRuntime().availableProcessors() > 1 ? 2 : 1;
ExecutorService ex = Executors.newFixedThreadPool(nThreads);

// Submit `Transfer' instances to the thread-pool, and get back a Future for each.
Future<Transfer<Job>> futureOne = ex.submit(boxOne);
Future<Transfer<Job>> futureTwo = ex.submit(boxTwo);
ex.shutdown();

lock.lock(); int counterCount = counter.get(); lock.unlock();
latch.await();

// After calling `get' on the a Future a `Transfer' is returned.
// The `Transfer' is subsequently opened, transfering ownership
// of the two `Job' references back to the main task.
futureOne.get().open(); futureTwo.get().open();

int oneCount = one.getMyCount();
int twoCount = two.getMyCount();

assert oneCount == 500 && twoCount == 500 && counterCount == 1_000 :
"One: " + oneCount + " Two: " + twoCount + " Counter: " + counterCount;
```

CHAPTER 4

IMPLEMENTATION

Major portions of the implementation of task-based V3 are discussed in this chapter. The chapter is broken into sections that complement Chapter 3.

4.1 Task

In order for any thread to use any user-defined object in task-based V3, the thread must have a Task attached to it. It is an error for a thread without an attached Task to access any members of any user-defined objects. The root of this comes from the "private" contract. Recall that the "private" contract allows only the creating Task to access the protected member. This means that if a user-defined object is built by a thread that does not have an attached Task, the system throws an IntentError, and alerts the programmer that the current Thread does not have a Task.

The public static *currentTask* method of the Task class is used to assign a current task to a private contract; it returns a reference to the currently calling task. This is implemented using a *ThreadLocal<Task>*. If any calling thread does not have a Task, an IntentError is thrown.

The lower-level mechanism, *ThreadLocal<Task>* in the Task class, uses the default access modifier, which allows only internal use. The first purpose for the ThreadLocal is to implement a child class of Task, InitialTask. This class has no more functionality than the Task class, but is defined as a marker. An InitialTask is attached to each thread in order for the thread to participate in the task-based framework. This includes the main thread. Without an InitialTask, threads fail when calling *currentTask*. InitialTask instances are only created internally. The user never needs to directly deal with attaching an InitialTask to a thread. There is a special contract that protects the

members of an InitialTask, an InitialTaskContract. This contract is associated with the members of the InitialTask instead of a "private" contract when it is created. The contract's agreement is: if the current task, which is retrieved from the lower-level mechanism, exists then go to an error state, otherwise do nothing and return. The contract protects against attaching any other InitialTask to a thread.

There is another derivation of Task called RunnableTask. RunnableTask is the implementation that safely runs user jobs. A RunnableTask takes a user job in the form of *Callable<T>* on construction. In order to work with thread-pools the RunnableTask implements the Java *Runnable* interface. The algorithm that RunnableTask implements in its *run* method is shown in Figure 4-1. Preconditions are shown by the ***Require*** statement.

Figure 4-1: RunnableTask.run Algorithm

```
1:  ThreadLocal variables: task, monitorsHeld
2:  Instance variables: state ← INIT, result ← null, userJob ← userDefinedJob
3:  procedure RunnableTask.run()
4:  Local variables: task′, monitorsHeld′
5:      Require state = INIT
6:      state ← RUNNING
7:      Require task ≠ null
8:      task′ ← task
9:      task ← this
10:     monitorsHeld′ ← monitorsHeld
11:     monitorsHeld ← Task.getLockedMonitors()
12:     result ← userJob.call()
13:     if exception is thrown then
14:         print "TaskDeath"
15:     if error is thrown then
16:         print "TaskDeath"
17:         propagate error up the call stack
18:     task ← task′
19:     monitorsHeld ← monitorsHeld′
20:     Require state = RUNNING
21:     state ← COMPLETE
```

A RunnableTask uses an explicit state transition system. The state transition system requires through atomic "compare-and-set" operations that only one thread will run the Task. The transition system consists of three states: *INIT*, *RUNNING* and *COMPLETE*. There are two valid transitions: *INIT* $\rightarrow$ *RUNNING*, and *RUNNING* $\rightarrow$ *COMPLETE*. This set of transitions does not allow a RunnableTask to be run more than once. A new instance of RunnableTask is created in the *INIT* state. When the Task is run, it transitions to the *RUNNING* state. Once it has completed running, the final transition to *COMPLETE* is made.

Recall that a Task can start another task. It is also imperative that the reference returned by *Task.currentTask* is in fact the one that is running. In order to move from a parent Task to a child Task and back to the parent, there is an implicit stack that RunnableTask maintains. If at any point the user job throws an exception, that exception is caught and the programmer is notified of the exception. The notification, "TaskDeath", is printed to the standard error stream. In the event of an error, the programmer receives the same notification, but the error is propagated up the call stack.

In order to implement the Synchronized synchronizer an additional static method of Task is needed. The method *holdsLock(Object o)* returns true if the current task holds the intrinsic lock on the supplied Object, and false otherwise. The current task needs to keep track of the intrinsic locks that its running thread holds. That value is kept in a ThreadLocal of type MonitorInfo[] called *monitorInfo*. This value is maintained by RunnableTask in the same way that the current task value is maintained. Finding the intrinsic locks that the running thread holds is done by the protected static *getLockedMonitors* method. This method leverages the Java ThreadMXBean, which is the thread management interface of the JVM. In order to tell if the current Task holds the intrinsic lock on an object four properties must be satisfied:

1. the current thread that is running the current task must hold the lock;

2. the number of locked monitors returned by *getLockedMonitors* must be greater than the number of monitors in the array returned by calling *get* on the *monitorInfo* ThreadLocal;

3. the value returned by *System.identityHashCode(o)* must be equal to the value returned by

calling *getIdentityHashCode* on the MonitorInfo object in question;

4. the class name given by *o.getClass().getName()* must be equal to the value returned by calling *getClassName()* on the MonitorInfo object in question.

The Synchronized synchronizer is further discussed in Section 4.6.

4.2 TaskLocal

In order to implement contracts in Intentional Concurrent Programming, it is reasonable to keep a per-task, integer-valued variable, which is associated with an instance of a contract. The value of the variable can be updated under certain conditions with respect to the task's usage of a synchronizer. In other words the state of the synchronizer from a task's point of view can be represented by this integer value. Now, when a task attempts to access a member of an object, the value of the variable is tested to determine whether access is granted.

Consider an example very similar to the example in Chapter 2. A task, T, tries to access a field, F. There is a lock, L, that T must hold in order to access F. T keeps a TaskLocal variable, I, of type Integer. Upon encountering the contract, C: if T holds L then the value of I is greater than zero, otherwise it is zero. So when T tries to access F, if I is greater than zero, C allows access, otherwise C moves T into an error state.

TaskLocal variables are an easy way to create task-confined objects. The usage makes implementing many contracts fall into a common pattern. It is simpler to use TaskLocal than a massive shared data structure to map which tasks are currently allowed to access what members of which objects.

The implementation of TaskLocal is derived from the implementation of Java ThreadLocal. TaskLocal is guaranteed to be thread-safe because it is a stateless class. It contains a package private static class called TaskLocalMap, which is a Map from TaskLocal to Object. All Task objects contain one instance field, a TaskLocalMap that is initialized to null. Because there are no means to share the TaskLocalMap, it is thread-safe. The *currentTask* method of Task returns a

reference in order to aid the implementation of TaskLocal. TaskLocal is a publicly accessible bridge to access the current task's TaskLocalMap. There are four methods of interest in TaskLocal:

- *T initialValue()* is a protected method that returns null by default. If the user chooses they may override this method to return an initial value the first time *get()* is called.
- *T get()* is a public method that first gets the current task. It then gets the TaskLocalMap reference from the current task, initializing it if needed. If there is no entry in the TaskLocalMap for *this*, then an entry is added mapping *this* to the result of calling *initialValue*. Otherwise the result of the lookup in the TaskLocalMap is returned.
- *set(T value)* is a public method that gets the current task and gets its TaskLocalMap, initializing it if necessary. An entry is entered into the TaskLocalMap that maps *this* to the *value*.
- *remove()* is a public method that gets the current task's TaskLocalMap and removes the entry associated with *this* from it.

An example usage of TaskLocal is shown in Figure 4-2. Consider a gambling service that needs a random number generator to implement a dice game. The service has many requests, and contention on one generator has proven to be a bottleneck in performance. The system is task-based and an easy solution to alleviate the bottleneck is to give each task its own *TaskLocal<Random>*.

Figure 4-2: TaskLocal Usage

```
private static class DiceRoller{

    private static TaskLocal<Random> random;

    // Need a static initializer block here to override initialValue then register
    // with ThreadSafe because the instance method, initialValue, will be used
    // by several Tasks.
    static{
        random = new TaskLocal<Random>(){

            @Override
            @Shared("ThreadSafe")
            protected Random initialValue(){
                return new Random();
            }
        };
        ThreadSafe.get().register(random);
    }

    public static int rollDice(){

        // The die is 6 sided so the range needs to be [1, 6]
        return random.get().nextInt(6) + 1;
    }
}
```

4.3 Transfer

Recall that all members for each newly created object use a "private" contract. This contract's purpose in task-based Intentional Concurrent Programming is to ensure that only the creating task may access any member of the newly created object. When a Transfer is opened by a task the "private" contract is replaced with a new "private" contract. The new "private" contract substitutes the opener of the box for the creating task.

A Transfer maintains a simple state transition system. The Transfer is constructed in a *CLOSED* state and can only transition to an *OPEN* state. The user can obtain an instance of a Transfer using the static method *newInstance* of the Transfer class. This method takes two parameters: an Object, called *contents*, whose members are disabled, and a String that is the annotation value that specifies what members to disable. On construction, the *contents* members that are annotated with the specified String are disabled by substituting their "private" contract

with a new one that has a null creating task.

A *register* method is offered in order to affect dependent instances of a composite object that is used as the *contents* of a Transfer. These dependent instances are children of the *contents* in an object graph. The parameters to this methods are: a child of the *contents*, and a String that is the annotation value that specifies what members of the child to disable. This method requires that the current state of the Transfer is *CLOSED*, before adding the child object to a collection. The method then disables the specified members of the Object.

When a Task opens the Transfer to retrieve the *contents*, all the members of the *contents* and its explicitly registered children that were disabled are re-enabled. The null owner "private" contracts are replaced with new "private" contracts that see the opener of the Transfer as the creator. In order to retrieve the *contents* of the Transfer, the *open* method must be called. The method requires that the state of the Transfer is *CLOSED* before returning the *contents*.

There is a subtle detail about Transfer that is interesting. The *register* method can only be called by the creator of the Transfer. The *open* method can be called by any task. There is a common lock that must be held in order to access either method. This lock keeps the state variable of the Transfer consistent, and disallows any race between a call to *register* and a call to *open*.

4.4 ThreadPoolExecutor

The ThreadPoolExecutor is the only implementation of a thread-pool in task-based V3. It simply wraps a ThreadPoolExecutor instance from the Java concurrency library. All of the methods of ThreadPoolExecutor are thread-safe. The following methods are all public and forwarded directly to the underlying ThreadPoolExecutor: *shutdown()*, *List<Runnable> shutdownNow()*, *boolean isShutdown()*, *boolean isTerminated()*, *boolean awaitTermination(long timeOut, TimeUnit unit)*. The *submit* method is not directly forwarded to the underlying ThreadPoolExecutor; it takes a single parameter of *Transfer<? extends Callable<T>>*. This is the Transfer that would hold a user job. The transfer is wrapped in a *RunnableTask<T>*. All the Task does is open the Transfer and call the user job, storing its result. Next, the RunnableTask is wrapped in a

thin *Callable<Transfer<T>>*, which calls *run* on the RunnableTask and gets the result from the RunnableTask. As long as there is a result, it is placed into a Transfer with all of its members disabled. The *submit* method returns a *Future<T>* to the user. To boil this down, the user submits a Transfer with their user job as the contents, then they get a Future back that returns the result of the user job as if the caller to *submit* is the creator of the result.

4.5 New Synchronizer

The new synchronizer added to V3 is the Future. The Future has a simple public API consisting of two public methods: *T get()* and *boolean isDone()*. These methods are only accessible by the caller of *submit* to a ThreadPoolExecutor. On construction a Future instance takes a Future from the Java concurrency library. This Future is wrapped by the task-based V3 Future. The *isDone* method is simply forwarded to the underlying Future. The *get* method first calls the *get* method of the underlying Future. This call blocks if necessary. The underlying Future returns a *Transfer<T>*. If a Transfer exists, it is opened. The result that is returned by the Future's *get* method substitutes "private" contracts as if the calling task is the creator of the result.

4.6 Existing Synchronizers

The existing synchronizers discussed in this section are: ReentrantReadWriteLock, ReentrantLock, Synchronized, and CountDownLatch. ThreadSafe and Disabled are left out because the implementations are trivial.

As previously mentioned, the use of TaskLocal is convenient when implementing contracts. Contracts are an integral part of the implementation of synchronizers under the Intentional Concurrent Programming model. It is reasonable to implement a CommonContract that may be used by as many synchronizers as possible. The CommonContract contains a per instance TaskLocal named *state*, and a functional interface of Java 8 called Predicate of type Integer named *acquired*. The Predicate is passed to the CommonContract on construction, its purpose is to test the *state*, adding any needed constraints. This test is done in the *use* method of the CommonContract,

which is inherited from Contract. This method is called any time that a Contract is used. The calls to the *use* method are maintained by the V3 system internally. The CommonContract has two package private methods: *set(int transition)* and *int get()*. The *set* method sets the *state*, and *get* returns the current value of the *state*. If any calling Task does not yet have a mapping for this particular ThreadLocal, the initial value of the *state* is zero.

There are two lock implementations that were migrated to the task-based system. A common lock class was implemented called V3Lock. The V3Lock wraps a Lock from the Java concurrency library. It also keeps a CommonContract. Both these elements are passed in on construction. The V3Lock offer two public methods: *lock* and *unlock*. The *lock* method forwards the call to the underlying Lock; this call blocks if necessary. Next, it updates the *state* of the contract. It adds one to the *state* of the contract for each successful call to lock. The *unlock* method tests the *state* of the contract, throwing an IntentError if the contract's *state* is less than one. This means that the calling task does not own the lock. Next, it decrements the *state* of the contract and calls *unlock* on the underlying Lock.

ReentrantLock and ReentrantReadWriteLock both use the V3Lock. ReentrantLock extends V3Lock. It is built with a Predicate that returns true if the *state* of the CommonContract is greater than zero, and false otherwise. There is a *register* method that takes an Object and String for the user to express intents. The ReentrantReadWriteLock wraps two V3Locks. Both of the Predicates for these locks are the same as for the ReentrantLock. One V3Lock is for the read side of the ReentrantReadWriteLock and the other is for the write side. The ReentrantReadWriteLock maintains two public methods for acquiring the read and write sides of the ReentrantReadWriteLock: *readLock*, and *writeLock* respectively. The *register* method takes an Object and two Strings: one String to specify members to associate with the read side of the lock and the other to specify members to associate with the write side of the lock.

CountDownLatch originally used a LatchContract that was specific to it. The CountDownLatch contained a ThreadLocal of type Integer. The LatchContract kept a reference onto its associated CountDownLatch, and was able to access the ThreadLocal to determine per-thread state. To migrate to task-based V3 the LatchContract was replaced by the CommonContract. This removes the

ThreadLocal from the CountDownLatch because of the use of TaskLocal in the CommonContract. Finally, the task-based CountDownLatch wraps a CountDownLatch from the Java concurrency library, while the thread-based version extends it.

Synchronized is a special case; the CommonContract is not used here. Synchronized uses its own special Contract called a SynchronizedContract. This contract calls the static *holdsLock* method of Task. If the current task holds the lock then access to the protected member is allowed, otherwise the system throws an IntentError. Synchronized offers two *register* methods. The first takes two Objects: the object to be prepared for sharing, and the object whose intrinsic lock must be held. This method is used when synchronized blocks that specify another object than *this* are part of the program. The second method takes the Object that will be prepared to be shared and whose intrinsic lock must be held. This method is used for synchronized methods. The String value that annotations must have in order to use Synchronized is "Synchronized".

Unfortunately, the *holdsLock* implementation has a limitation. Consider the following case. A task, T, must hold the intrinsic lock on an Object, O, in order to access a field F. T synchronizes on O with a synchronized block, then starts a new task, T_1. The programmer's intent for the new task, T_1, is to synchronize on O and access F. By mistake the programmer synchronizes on a different Object, O_1. An IntentError should be thrown, and almost always will be. However, the value returned by *System.identityHashCode* is not guaranteed to always be unique. The Java documentation from Object states:

> *"As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java programming language.)"*

This is the same value that is returned by *System.identityHashCode*. This means that there is a very small chance that when T_1 attempts to access F, it will be allowed. The slim chance for a false positives negates the Intentional Concurrent Programming model. An implementation should not return any false positives. Since a MonitorInfo object does keep a reference to the

object whose intrinsic lock is held, there is some potential for ambiguity raised from using its available descriptions of that object. It is worth noting that none of the current applications in the V3 repository exercises a case similar to the one described above. A second potential for error is that an implementation of the JVM is not required to support monitoring of intrinsic locks. A lower-level implementation is needed to acquire the actual list of object references whose intrinsic locks are held. This avoids the potentials of false positives and lack of platform support.

CHAPTER 5

EVALUATION

5.1 Overview

This chapter discusses the evaluation of task-based V3. Unit-tests demonstrate correct semantics of updates and additions to the system. Positive and negative unit-testing shows that classes act as expected in isolation.

Existing thread-based V3 applications were migrated to task-based V3. There is also an addition to the application repository. The evaluation shows that intents are successfully embedded into a task-based system. Table 5.1 lists all applications, and the features of task-based V3 that are used. The applications are: Pi, Jacobi, and LongestWords. Pi approximates the value of π by using numerical integration. Jacobi simulates the transfer of heat on a two-dimensional square plate. LongestWords reads in books of text from disk. It finds the twenty longest words that appear in all of the books. LongestWords has a "Locks" version and a "Future" version.

Application	Synchronized	ReentrantLock	ReentrantReadWriteLock	CountDownLatch	Future
Jacobi			✓	✓	
Pi		✓		✓	
L.W. Locks	✓	✓	✓	✓	
L.W. Future					✓

Table 5.1: Synchronization Features Used in Applications

5.2 Pi

This application is migrated to task-based V3 by editing the class that represented each thread's work. The work is changed to be framed as a user job. Each of the user jobs become the contents of

an associated Transfer. All of the Transfer instances are submitted to a thread-pool. The thread-pool is built with one thread per available processor. Now the command line client can request a number of tasks without needing to worry about spawning too many threads. When the program is run on a machine with more available processors, it can now scale to appropriately use the hardware. The command-line client can now request larger numbers of tasks as compared to the thread-based version. A closer approximation of π results from using more tasks. This application is illustrated in Figure 5-1 and Figure 5-2. After the migration to task-based V3, solutions are still adequate approximate values of π.

5.3 Jacobi

The square plate is modeled as a two-dimensional array of type double. The application uses the iterative Jacobi method to converge on a solution. One thread is allocated per row of the two-dimensional array. The thread writes to its assigned row. It reads its assigned row, the row above it, and the row below it. This application was migrated to task-based V3 in much the same way as Pi, except there is one thread per task. All of the Transfer instances are submitted to a thread-pool that keeps the same number of threads as the original application. The solution that the application reports has not changed after the migration to task-based V3.

5.4 LongestWords Locks

The Locks version of LongestWords allows the command-line client to specify what type of lock to use. The options are: Synchronized, ReentrantLock, and ReentrantReadWriteLock. The specified lock protects a shared data structure. There is a main task that creates one task per book, submitting each task to a thread-pool. The main task also creates a CountDownLatch with the count initialized to the number of books. After submitting all of the book-reading tasks, the main task calls *await* on the CountDownLatch instance. Each book-reading task reads one line from the text file, converting it into valid words. The data structure stores the words, and the books in which the words occur. Each task keeps a reference to the CountDownLatch instance created

by the main task. The CountDownLatch is counted down as each task finishes its work. After the last task has finished with the last book, the main task is unblocked from the CountDownLatch. This allows the main task to print the results to the standard output stream.

5.5 LongestWords Future

The Future version does not use any locks. It uses Futures to coordinate steps of work. There is one task built per book by the main task. These book-reading tasks are submitted to a thread-pool. The main task keeps a Future for each task that it submitted. The book-reading task reads several lines of text, and creates a new task that transforms the text to a set of valid words. These new tasks are submitted to a thread-pool and Futures are kept for each of the new set-building tasks. Once a book-reading task has read all of the lines of its associated book, it waits on the Futures for each of its submitted set-building tasks. Each Future returns a resulting set of words that are combined into a set that represents all of the valid words in the book. The main task waits on all of the Futures returned by submitting the book-reading tasks. The main task uses each of the per-book sets of words to determine the longest twenty words that appear in all of the books. Both implementations return the same results when tested with the same 25 books as input.

Figure 5-1: Pi.java

```
1  public class Pi{
2
3    private static class GlobalSum{
4      // threads will add their partial sums here
5      @PutShared("Lock") @GetShared("Lock")
6      private double sum;
7
8      @Shared("LatchClosed")
9      public void addToSum(double partial){ sum += partial; }
10
11     @Shared("LatchOpen")
12     public double getSum(){ return sum; }
13
14     public GlobalSum(){ sum = 0.0; }
15   }
16
17   public static void main(String[] args){
18
19     if (args.length == 0)
20     {
21       throw new RuntimeException("usage: Pi numberOfTasks");
22     }
23     // get the number of tasks to use from the command line
24     final int n;
25     try {
26       n = Integer.parseInt(args[0]);
27     }
28     catch (NumberFormatException nfe)
29     {
30       throw new RuntimeException("usage: Pi numberOfTasks");
31     }
32
33     final int INTERVALS = 50000000;
34     final double width = 1.0 / INTERVALS;
35     // control for distributing the intervals
36     int ch = INTERVALS / n;
37     int sp = INTERVALS % n;
38     if (sp == 0)
39     {
40       sp = n;
41       ch -= 1;
42     }
43     final int chunk = ch;
44     final int split = sp;
45
46     // create a shared object to capture the partial sums
47     final GlobalSum sum = new GlobalSum();
48     ThreadSafe.get().register(sum);
49     final ReentrantLock lock = ReentrantLock.newInstance();
50     lock.register(sum, "Lock");
51
52     // create latch to synchronize the tasks (and control sharing of the
53     // sum object)
54     final CountDownLatch latch = CountDownLatch.newInstance(n);
55     latch.register(sum, "LatchClosed", "LatchOpen");
```

Figure 5-2: Pi.java continued

```
58      // build thread-pool with number of threads equal to number of processors.
59      int processors = Runtime.getRuntime().availableProcessors();
60      ExecutorService exec = Executors.newFixedThreadPool(processors);
61      // now create the user jobs to do the computation
62      for (int i = 0; i < n; i++) {
63        final int id = i;
64        Callable<Void> job = new Callable<Void>() {
65          @Override
66          @Shared("Transfer")
67          public Void call() {
68            // tasks will update global sum before latch opens
69            latch.start();
70
71            int low;
72            int high;
73            if (id < split) {
74              low = (id * (chunk + 1));
75              high = low + (chunk + 1);
76            } else {
77              low = (split * (chunk + 1)) + ((id - split) * chunk);
78              high = low + chunk;
79            }
80            double localSum = 0.0;
81            double x = (low + 0.5) * width;
82            for (int j = low; j < high; j++) {
83              localSum += (4.0 / (1.0 + x * x));
84              x += width;
85            }
86            lock.lock();
87            sum.addToSum(localSum);
88            lock.unlock();
89
90            latch.countDown();
91            return null;
92          }
93        };
94        // build Transfer with user job as contents, then submit to thread-pool
95        Transfer<Callable<Void>> box = Transfer.newInstance(job, "Transfer");
96        exec.submit(box);
97      }
98      // wait for the latch to open and shutdown the thread-pool.
99      try {
100       latch.await();
101       exec.shutdown();
102     } catch (InterruptedException ie) {
103       exec.shutdownNow();
104       throw new RuntimeException("caught InterruptedException");
105     }
106     double globalSum;
107     lock.lock();
108     globalSum = sum.getSum();
109     lock.unlock();
110     assert(String.format("%14.12f", globalSum * width).startsWith("3.141592653"));
111   }
112 }
```

CHAPTER 6

FUTURE WORK AND CONCLUSION

6.1 Future Work

Future should have a *register* method that creates "before" and "after" relationships with a registered object's annotated members. "Before" members would be enabled only before *get* is called, and "after" members would be enabled only after *get* has returned. This is similar to the idea of registering "closed" and "open" intents between a CountDownLatch instance and an object. The only drawback is that this only works for one specific Future and one specific object; it is a common pattern to submit many tasks and receive many Futures. The addition of contract combinators could help in this instance. This would be a kind of meta-contract that would use several other contracts. These contracts would be related with logical connectives, and access to a protected member would be granted based on the outcome.

The members of Future currently have no annotations. A Future is only usable by its creator. This is a safe way to ensure that the semantics of Future are not broken. A Future can not be the contents of a Transfer, because there is no annotation to identify a relation to a Future's members. This case brings up a potential extension to Transfer. Transfer could offer a "FullyDisabled" subtype. This type would disable all members of the contents, without regard to the annotations. Disabling would be done on construction. When a "FullyDisabled" is opened, all of the members of the contents would be re-enabled as if the opener of the "FullyDisabled" was the creator.

The potential race condition that is avoided by using a lock in Transfer could be completely avoided. There should be one state added to the Transfer state transition system. An instance of Transfer needs to start in an *INIT* state, then move to *CLOSED* by a new method called *close*. Calling *register* should only be allowed in the *INIT* state. If it is only valid to open a Transfer that is in the *CLOSED* state, then there is never a chance to race on *register* and *open*. This would eliminate the need for the lock instance in the Transfer.

Another potential change is in TaskLocal. When overriding the protected *initialValue* method of TaskLocal, the user must confirm the method is thread-safe. This problem is directly inherited from ThreadLocal, the class that the TaskLocal implementation is based on. Task-based V3 does not help here. A different method that users override could be offered, *safeInitialValue*. That method would be protected by an internal lock, per instance of TaskLocal. The *initialValue* method would first acquire the internal lock, then call the *safeInitalValue* method. After returning, the lock would be released and execution would carry on as normal. This will force serialization when accessing a potentially shared state in the call to *sharedInitialValue*. The trade-off here is that serialization may be forced when it is not needed.

The Actors section of Chapter 2 discussed some potential benefits of using task-based V3 with Akka. Akka specifies by convention only that messages be immutable. This is to avoid race conditions that would be caused by accidentally sharing an Actors internal mutable state. If Akka used Transfers, it could ensure that it would not be possible to have race conditions on messages. Consider the case that some internal mutable state of an Actor is shared in a message, but it is now the contents of a Transfer. Once the Transfer is built by the mistaken Actor, the shared state will no longer be enabled for that Actor to use. Once the Actor tries to use the accidentally shared state, an IntentError would be thrown.

6.2 Conclusion

The current V3 implementation described here demonstrates that the migration to task-based Intentional Concurrent Programming is possible. A thread-pool is added to the system to make

task-based programming possible. Task and its derivatives are integral components, giving threads membership to use the system, and the ability to run user jobs. Future is a new synchronization construct that holds the result returned by a Task that is run asynchronously. Transfer is the highlight of the project, providing the vehicle that allows users to safely share mutable state. Transfer facilitates safely submitting jobs to a thread-pool, and retrieving results from a Future.

Existing synchronization constructs are updated in order to migrate to task-based V3. The biggest difficulty in the migration is the Synchronized synchronizer. However, this difficulty is due to the high-level implementation of task-based V3, and does not prohibit embedding intents in a task-based system. ReentrantLock, ReentrantReadWriteLock, and CountDownLatch all benefit from a unified contract, CommonContract. This is possible as a result of the addition of TaskLocal to the system. Unit testing showed that each of the additions, and updates work as expected in isolation. The successful changes to the application repository demonstrate that the system is expressive enough to support the Intentional Concurrent Programming model. In the current state of the digital age, the safety of concurrent programming is a priority. Task-based V3 is a step that positively supports research in concurrency.

BIBLIOGRAPHY

[1] Brian Goetz, David Holmes, Doug Lea, Tim Peierls, Joshua Bloch, Joseph Bowbeer. *Java Concurrency in Practice*. Pearson, 2006.

[2] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM.

[3] Kevin Poulsen. Tracking the blackout bug. `http://www.securityfocus.com/news/8412`, April 2004.

[4] Nancy Leveson, et al. Medical devices: The Therac-25. *Appendix of: Safeware: System Safety and Computers*, 1995.

[5] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[6] Michel Charpentier and Phil Hatcher. Intentional Concurrent Programming. Working paper, 2014. Department of Computer Science, University of New Hampshire.

[7] Michaela Tremblay. Throwing Exceptions for Concurrency Errors, May 2015. B.S. Honors Thesis, University of New Hampshire.

[8] Nicholas Craycraft. A System for Intentional, Multithreaded Java, May 2016. B.S. Honors Thesis, University of New Hampshire.

[9] Shigeru Chiba. Java bytecode engineering toolkit. `http://jboss-javassist.github.io/javassist/`, 2016.

[10] Python Standard Library. Launching parallel tasks. https://docs.python.org/3/library/concurrent.futures.html.

[11] Microsoft. Task Parallelism. https://msdn.microsoft.com/en-us/library/dd537609(v=vs.110).aspx.

[12] Philipp Henkel. Threadpool. http://threadpool.sourceforge.net/.

[13] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[14] Lightbend Inc. Actors. http://doc.akka.io/docs/akka/current/java/untyped-actors.html.

[15] Wikipedia. Producer-consumer problem. https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem.

[16] Wikipedia. Readers-writers problem. https://en.wikipedia.org/wiki/Readers%E2%80%93writers_problem.

Appendix A

The public API of task-based V3 follows. In some cases descriptions and names are largely derived from the underlying implementation. These derivations are from the Java SE runtime library. In some cases descriptions and names are derived from the thread-based implementation of V3.

A.1 Task

Provide public access to Task.currentTask. Default access for inheritance and lower-level static methods.

public static Task currentTask()
Returns the current Task.
Returns:
A reference to the currently running Task.
Throws:
IntentError - if the Current thread has no Task.

A.2 InitialTask

Class used to bootstrap threads.

A.3 RunnableTask<V>

Used to wrap Callable user code. A Task implements Runnable in order to not pass values or exceptions to other tasks.

public RunnableTask(java.util.concurrent.Callable<V> userCode)
Returns a RunnableTask<V> for use with the V3 Task-Based system.
Parameters:
userCode - a user object intended to be run by a thread in the V3 task-based system.

public RunnableTask(java.util.concurrent.Callable<V> userCode, java.lang.String taskName)
Returns a RunnableTask<V> for use with the V3 task-based system.
Parameters:
userCode - a user object intended to be run by a thread in the V3 task-based system.
taskName - string to represent this task.

public void run()
RunnableTask run method. Run the user code storing the result if it exists

public V getResult()
Get the result of the user code. This method can only be called after the Task has completed running.

Returns:
the result of running the user code.

public java.lang.String toString()
Return a string to represent this task. It follows the format: If taskName is not null: "Task 'taskName' run by Thread 'currentThread'" If taskName is null: "Task 'super.toString' run by Thread 'currentThread'"
Returns:
a String representation of this

A.4 TaskLocal<T>

This class provides task-local variables. It is largely derived from java.lang.ThreadLocal.

public TaskLocal()
Creates a task-local variable

protected T initialValue()
Returns the current task's "initial value" for this task-local variable. This method will be invoked the first time a task accesses the variable with the get() method, unless the task previously invoked the set(T) method, in which case the initialValue method will not be invoked for the task. Normally, this method is invoked at most once per task, but it may be invoked again in case of subsequent invocations of remove() followed by get(). This implementation simply returns null; if the programmer desires task-local variables to have an initial value other than null, TaskLocal must be subclassed, and this method overridden. Typically, an anonymous inner class will be used. In this case, initialValue will need to be annotated with @Shared("ThreadSafe"), and registered with the ThreadSafe pseudo-synchronizer. If the task-local variable is a static member of a class, registration with ThreadSafe must be done in a static initializer block.
Returns:
the initial value for this task-local

public T get()
Returns the value in the current task's copy of this task-local variable. If the variable has no value for the current thread, it is first initialized to the value returned by an invocation of the initialValue() method.
Returns:
the current task's value of this task-local

public void set(T value)
Sets the current task's copy of this task-local variable to the specified value
Parameters:
value - the value to be stored in the current task's copy of this task-local

public void remove()
Removes the current task's value for this task-local variable. If this task-local variable is subsequently read by the current task, its value will be reinitialized by invoking its initialValue() method, unless its value is set by the current task in the interim. This may result in multiple invocations of the initialValue method in the current task.

A.5 Transfer<T>

A Transfer provides a mechanism to safely share mutable state with one other task. The creating task, builds the Transfer putting contents into it. At the end of construction the Transfer is in a "closed" state. The

creating task may register additional objects with the Transfer. Exceptions will occur in the the following cases: 1. Attempt to open a Transfer that is not closed. 2. Attempt to register an object with this Transfer when it is in an open state.

public static <T> Transfer<T> newInstance(T contents, java.lang.String openSync)
Static factory method to produce a new instance of a Transfer<T>
Type Parameters:
T - type of contents
Parameters:
contents - object to be held by the Transfer
openSync - annotation value for members that will be disabled when Transfer is CLOSED and enabled when Transfer is OPEN.
Returns:
a new instance of a Transfer<T>

public T open()
Open the Transfer to get the contained object out. Opening the Transfer allows the calling task to access the members registered with the Transfer. These members are enabled when the Transfer is open. These members should be reachable from the returned object.
Returns:
The object contained in the the Transfer.
Throws:
IntentError - if this Transfer is not closed when open is called, or if the opening task was the last to close the Transfer.

public void register(java.lang.Object child, java.lang.String childOpenSync)
Register children of the contents of the Transfer. Only the creator of the Transfer can register children with the Transfer. Registration can only be done when the Transfer is CLOSED.
Parameters:
child - a child of the object that is the contents of the Transfer.
childOpenSync - Annotation value for members that are enabled when the Transfer is OPEN and disabled when the Transfer is CLOSED.

A.6 ThreadPoolExecutor

ThreadPoolExecutor that is specific to the V3 project. ThreadPoolExecutor is a wrapper around java.util.concurrent.ThreadPoolExecutor. This is the only implementation of ExecutorService that is currently offered by the V3 system. Submission of a Transfer<? extends Callable<T>> is the only way access the underlying thread-pool.

public void shutdown()
Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted. Invocation has no additional effect if already shut down. This method does not wait for previously submitted tasks to complete execution. Use awaitTermination to do that.
Throws:
java.lang.SecurityException - if a security manager exists and shutting down this ExecutorService may manipulate threads that the caller is not permitted to modify because it does not hold RuntimePermission("modifyThread"), or the security manager's checkAccess method denies access.

public java.util.List<java.lang.Runnable> shutdownNow()
Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution. This method does not wait for actively executing tasks to terminate. Use awaitTermination to do that. There are no guarantees beyond best-effort attempts to stop processing actively executing tasks. For example, typical implementations will cancel via Thread.interrupt(), so any task that fails to respond to interrupts may never terminate.
Returns:
list of tasks that never commenced execution
Throws:
java.lang.SecurityException - if a security manager exists and shutting down this ExecutorService may manipulate threads that the caller is not permitted to modify because it does not hold RuntimePermission("modifyThread"), or the security manager's checkAccess method denies access.

public boolean isShutdown()
Returns true if this executor has been shut down.
Returns:
true if this executor has been shut down

public boolean isTerminated()
Returns true if all tasks have completed following shut down. Note that isTerminated is never true unless either shutdown or shutdownNow was called first.
Returns:
true if all tasks have completed following shut down

public boolean awaitTermination(long timeout, java.util.concurrent.TimeUnit unit) throws java.lang.InterruptedException
Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.
Parameters:
timeout - the maximum time to wait unit - the time unit of the timeout argument
Returns:
true if this executor terminated and false if the timeout elapsed before termination
Throws:
java.lang.InterruptedException - if interrupted while waiting

public <T> Future<T> submit(Transfer<? extends java.util.concurrent.Callable<T>> box)
Submits a value-returning task for execution and returns a Future representing the pending results of the task. The Future's get method will return the task's result upon successful completion. The task's result can be null. If you would like to immediately block waiting for a task, you can use constructions of the form result = exec.submit(aCallable).get();
Type Parameters:
T - the type of the task's result
Parameters:
box - the Transfer to submit
Returns:
a Future representing pending completion of the task

A.7 Executors

Built to mimic java.util.concurrent.Executors. This class is a collection of static factories that will return an ExecutorService. The ExecutorService implementation is a limited implementation based on an

edu.unh.cs.v3.lib.ThreadPoolExecutor. There are three static factory methods:
newFixedThreadPool(nThreads), newSingleThreadExecutor(), and newCachedThreadPool()

public static ExecutorService newFixedThreadPool(int nThreads)
Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue. At any point, at most nThreads threads will be active processing tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available. If any thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks. The threads in the pool will exist until it is explicitly shutdown.
Parameters:
nThreads - the number of threads in the pool
Returns:
the newly created thread pool
Throws:
java.lang.IllegalArgumentException - if nThreads <= 0

public static ExecutorService newCachedThreadPool()
Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. These pools will typically improve the performance of programs that execute many short-lived asynchronous tasks. Calls to execute will reuse previously constructed threads if available. If no existing thread is available, a new thread will be created and added to the pool. Threads that have not been used for sixty seconds are terminated and removed from the cache. Thus, a pool that remains idle for long enough will not consume any resources.
Returns:
the newly created thread pool

public static ExecutorService newSingleThreadExecutor()
Creates an Executor that uses a single worker thread operating off an unbounded queue. (Note however that if this single thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.) Tasks are guaranteed to execute sequentially, and no more than one task will be active at any given time.
Returns:
the newly created single-threaded Executor

A.8 Future<V>

Concrete class that is private to the submitter of an ExecutorService obtained from Executors. get can only be called once or an IntentError is thrown.

public boolean isDone()
Returns true if this task completed. Completion may be due to normal termination, an exception, or cancellation – in all of these cases, this method will return true.
Returns:
true if this task completed

public V get() throws java.lang.InterruptedException, java.util.concurrent.ExecutionException
Waits if necessary for the computation to complete, and then retrieves its result. Result can be null.
Returns:
the computed result
Throws:
java.util.concurrent.CancellationException - if the computation was canceled

java.util.concurrent.ExecutionException - if the computation threw an exception
java.lang.InterruptedException - if the current thread was interrupted while waiting
IntentError - if there is more than one call to get

A.9 ThreadSafe

Pseudo-synchronizer for marking members as thread-safe.

public static ThreadSafe get()
Get a ThreadSafe mechanism.
Returns:
a ThreadSafe mechanism.

public void register(java.lang.Object obj)
Put in place contracts that allow members annotated with the synchronizer name "ThreadSafe" to be accessed by any thread.
Parameters:
obj - object to be prepared for sharing

A.10 Disabled

Pseudo-synchronizer for marking members as not accessible by any task.

public static Disabled get()
Get a Disabled mechanism.
Returns:
a Disabled mechanism.

public void register(java.lang.Object obj)
Put in place contracts that disallow members annotated with the synchronizer name "Disabled" from being accessed by any task.
Parameters:
obj - object to be processed

A.11 Synchronized

Pseudo-synchronizer for sharing objects controlled by the synchronized keyword. WARNING - using this class has a small potential to result in false positives.

public static Synchronized get()
Get a Synchronized sharing mechanism.
Returns:
a Synchronized sharing mechanism.

public void register(java.lang.Object obj, java.lang.Object syncObj)
Facilitate the sharing of an object by requiring members annotated with Synchronized to be accessible only if the running task holds the internal lock of an object, which might be a different object than the first object.

Parameters:
obj - object to be prepared for sharing
syncObj - object whose internal lock must be held

public void register(java.lang.Object obj)
Facilitate the sharing of an object by requiring members annotated with Synchronized to be accessible only if the running task holds the internal lock of the object.
Parameters:
obj - object to be prepared for sharing

A.12 ReentrantLock

Synchronizer for V3 ReentrantLock. ReentrantLock extends V3Lock, this is where the lock and unlock methods are implemented.

public static ReentrantLock newInstance()
Produce a new ReentrantLock.
Returns:
A new instance of ReentrantLock that is properly registered and ready for use.

public void register(java.lang.Object obj, java.lang.String sync)
Register an object. Nobody can access the matching annotated members of the object without obtaining the lock.
Parameters:
obj - object to be registered with the lock.
sync - annotation name to be matched.

public void lock()
Acquires the lock.

public void unlock()
Releases the lock.

A.13 ReentrantReadWriteLock

Synchronizer for ReentrantReadWriteLock. The lock and unlock methods for both the read and write sides of the lock are implemented in V3Lock.

public static ReentrantReadWriteLock newInstance()
Get a new instance of a V3 ReentrantReadWriteLock, properly registered and ready to use.
Returns:
A new ReentrantReadWriteLock.

public Lock readLock()
Get the ReadLock for this ReentrantReadWriteLock.
Returns:
The ReadLock of this.

public Lock writeLock()
Get the WriteLock for this ReentrantReadWriteLock.

Returns:
The WriteLock of this.

public void register(java.lang.Object obj, java.lang.String readSync, java.lang.String writeSync)
Register the object. Nobody can access Read members of the object without obtaining the read lock. Nobody can access Write members of the object without obtaining the write lock.
Parameters:
obj - object to be registered with the lock.
readSync - annotation name to be matched for read members.
writeSync - annotation name to be matched for write members.

A.14 CountDownLatch

CountDownLatch with two CommonContract instances. Contracts are used to control whether a task can access a member of a registered object before or after the opening of the latch. Each task is in one of three states: latch is closed, the task has called start but has not yet called countdown; latch is closed but the task has called countdown; and, latch is open and the task has called await. In the first case the task is allowed to call "closed" methods. In the second case the task cannot call the "closed" methods or the "open" methods. In the third case the task can call the "open" methods.

public static CountDownLatch newInstance(int count)
Produce a new V3 CountDownLatch that is properly registered and ready for use.
Type Parameters:
count - the count for the latch.
Returns:
A new instance of a V3 CountDownLatch.

public void register(java.lang.Object obj, java.lang.String closedSync, java.lang.String openSync)
Register an object with the latch. Contracts are set up for the members based upon whether their annotation indicates whether they are to be called when the latch is open or closed.
Parameters:
obj - object to be registered with the latch.
closedSync - name marking members accessible when latch is closed.
openSync - name marking members accessible when latch is open.

public void start()
Calling task declares its intent to access the "closed" members.

public void countDown()
Calling task counts down the latch.

public void await() throws java.lang.InterruptedException

Calling task awaits the latch opening.

Throws:

java.lang.InterruptedException - if the current Thread is interrupted while waiting.