Favoring Isolated Mutability
The Actor Model of Concurrency

CSCI 5828: Foundations of Software Engineering
Lecture 24 — 04/11/2012
Goals

• Review the material in Chapter 8 of the Concurrency textbook
  • that deals with the fundamentals of the actor approach
  • and the use of UntypedActors in Akka

• We’ll review the use of
  • TypedActors, and
  • Mixing STM with the Actor Model

• in our next lecture
“If it hurts, stop doing it”

- With concurrency, we have learned
  - that the shared mutability approach to concurrency leads to lots of problems and difficulty
- As a result, we’ve been looking at alternatives
  - With the STM model, we looked at managed mutability
    - where all values were immutable
    - but a “ref” or an “identity” could be associated with different values over time; with change occurring inside of transactions
- Now, we’ll look at the isolated mutability approach
  - as embodied in the Actor Model of Concurrency
Isolated Mutability and the Actor Model

• Isolated Mutability is a design approach in which
  • we can have mutable values
  • but we make sure that for each mutable value
    • only one thread has access to it
• The Actor Model of concurrency is one in which
  • we have multiple lightweight processes, known as actors
    • each actor can have mutable state if it wants
      • because no other actor has access to that state
  • actors instead pass immutable messages to each other if they need to communicate or coordinate
    • these messages are passed asynchronously and are processed in the order that they arrive
The Actor Model: Background

• This model has been around for a long time
  • It is built into the functional programming language, Erlang
• And was also built into Scala, a hybrid functional/OO language
  • Scala is built on top of the JVM
    • it can call Java code and Java can call Scala code
      • as we saw when working with the STM
        • The Akka framework is written in Scala
Actor Model

• To emphasize
  • Programs that use the Actor model are multithreaded
    • but each individual actor is single threaded
  • they each have access to mutable state
    • but an actor cannot access the mutable state of another actor
  • all one actor can do to another actor is send an immutable message
    • since the messages are immutable, it is safe to share them between actors
  • likewise, all an actor can do is sit and wait for messages to arrive
Actor Model == OOP++?

• The book asserts that the actor model can be seen as taking OOP to the next level
  • We have objects and they can have mutable state
    • but they each run on their own thread
    • and all we can do is send messages to them
      • we can’t call their methods directly
Actor Qualities (I)

• Each actor is an independent activity
  • it can receive messages
  • process them
  • and send messages

• Each actor has a built in message queue
  • it can receive multiple messages at once
  • it can send a message at the same time that other actors are sending them
• As a result, there is plenty of opportunity for concurrency!
Actor Qualities (II)

- Actor does not equal Thread
  - Instead, think of an actor as a task
- Recall how we separated task decomposition from thread allocation?
  - Allowing us to create, for instance, a thread pool with 20 threads
    - And then allocate 100 tasks to be processed by the thread pool
- The same thing happens with actors
  - We will likely have X actors being managed by Y threads where
    - $X >> Y$
- We can get away with this because of the actor lifecycle
Actor Life Cycle

If an actor is active but has no messages, then it is essentially blocked; we can swap it out and let some other actor run.

© Kenneth M. Anderson, 2012
Creating Actors (I)

- Support for the Actor Model is built into Scala
  - as a result, Scala’s syntax makes it easy to create actors and send messages to them
    - and to process messages as well
- I’m not going to cover the details of Scala in this class
  - so we are going to use the Akka framework to implement actors in Java
- As a result, the information discussed in Lecture 19 on slides 34 and 35 is relevant here
  - you will need to follow those instructions in order to compile our example programs
Creating Actors (II)

• The simplest actor in Akka has a class name of UntypedActor
  • That class is located in the akka.actor package

• We can think of it as an abstract class that has one method we need to implement:
  • public void onReceive(final Object message)

• We implement this method to indicate how our actor will handle its messages
  • Note: the parameter type for message is java.lang.Object
  • In practice, only immutable types can be sent to us
    • and we do need to perform checks at run-time to figure out what message was sent to us
Creating Actors (III)

- Our increment program lives again!
  - public class Counter extends UntypedActor {
    - private int count = 0;
    - public void onReceive(final Object message) {
      - if (message instanceof Integer) {
        - count += (Integer)message;
        - System.out.println("Count: " + count);
      - }
    - }
  - }

© Kenneth M. Anderson, 2012
Creating Actors (IV)

• Our increment program lives again!

  • public class Drone extends UntypedActor {
    •   public void onReceive(final Object message) {
      •     if (message instanceof ActorRef) {
        •       ActorRef counter = (ActorRef)message;
        •       for (int i = 0; i < 5; i++) {
          •         counter.tell(new Integer(1));
        •       }
      •     }
    •   }
  • }
Creating Actors (V)

• To instantiate one of our actors, we make use of the Actors factory in the package akka.actors.
  • We pass the factory the class of the instance, we want created
  • We get back an ActorRef that points at our newly created Actor

• So, when we create an instance of our Counter actor
  • we do not get back a reference to Counter
  • we get back a reference to ActorRef

• We can use that reference to send messages to the Counter
  • Why? We are not supposed to have access to instances of Counter directly; if we did, Counter’s mutable variables might escape!
Creating Actors (VI)

- Our increment program lives again!
  - The full program is in Increment.java

- Creating an ActorRef looks like this
  - final ActorRef counter = Actors.actorOf(Counter.class);

- To start an Actor, you call start() on the ActorRef
  - counter.start();

- To send an asynchronous message, use the tell() method
  - counter.tell(new Integer(100));

- DEMO
Creating Actors (VII)

• The book had several examples of creating Actors

  • DEMO

• One of its examples touches on creating Actors that have constructors

  • Since you are not allowed to directly instantiate an Actor class, it is difficult to pass values to an Actor’s constructor

  • To do that, you need to create an anonymous instance of the UntypedActorFactory class

    • that factory has a create() method that returns instances of UntypedActor and you can pass constructor arguments there

  • You then pass the UntypedActorFactory to the actorOf() method

    • It uses the factory to create an instance and return an ActorRef
Sending Messages

• You can send messages in two ways
  
  • `tell(final Object message)` -- sends an asynchronous, immutable message
  
  • `Future ask(final Object message)` -- sends a message, provides future
  
  • Future is NOT `java.util.concurrent.Future` but it operates in the same way
  
  • Once you get back a future
    
    • you call `Future.await()` to block until a response is available
  
  • You then call `Future.result().get()` to acquire the immutable response sent to you by the other actor
    
    • The call to `result().get()` can fail; you need to call `Future.result().isDefined()` and only call `get()` when `isDefined()` returns true

DEMO
Repying

• If you receive a message from another actor, how do you reply?
  • If you are within the onReceive() method of an actor, you can simply call
    • getContext().channel() to get access to an ActorRef
    • Once you have the ActorRef, you can call tell() and ask() as normal
  • So, replying is really the same thing as just sending!
Handling Multiple Actors

- We already saw multiple actors in action with our simple Increment program
  - The book returns to the PrimeFinder example
- It has a simple design
  - We are provided the upper bound of our search and the number of partitions
    - Our main program creates one Actor per partition and sends it a range using ask(), which returns a Future
      - Each actor calculates the number of primes in that range and sends it back
    - Our main program loops through Future objects and calculates the total number of primes
- As you will see, the program maxes out the cores of my machine when I run the program
Coordinating Actors (I)

• To show how Actors can coordinate with each other, the book returns to the FileSize program

  • Our previous versions of this program made use of coordination mechanisms (locks) and executors

• With the isolated mutability approach of the Actor model

  • we can get a simpler solution and avoid some of the problems we encountered earlier

  • such as when we locked up the thread pool with a poor design related to spawning tasks while traversing the file hierarchy
Coordinating Actors (II)

• The design of this system depends on two types of actors
  
  • FileProcessors
    
    • actors which process the size of a single directory
    
    • we will create 100 of these to simulate the 100 threads we used earlier
  
  • SizeCollector
    
    • an actor who coordinates the FileProcessors and maintains a counter to keep track of the total size of the directory
Coordinating Actors (III)

• The design of this system also depends on three messages
  • RequestAFile:
    • sent by a FileProcessor to the SizeCollector
    • The effect is to tell the SizeCollector, “I’m ready!”
  • FileToProcess:
    • sent by SizeCollector to FileProcessor or vice versa
    • provides a pointer to a file or directory that needs to be processed
  • FileSize
    • sent by FileProcessor to SizeCollector
    • returns the size of a file/directory that was processed
Coordinating Actors (IV)

• Two key design points

  • When a FileProcessor starts up, it needs to tell the SizeCollector that it is available

    • It overrides a lifecycle method preStart() to do that

      • That method ensures that we send a RequestAFile message to the SizeCollector

  • A reference to the SizeCollector is passed in via FileProcessor’s constructor

• When a FileProcessor is given a directory, it does not recursively work its way through all subdirectories. Instead, as it finds subdirectories, it sends FileToProcess messages to SizeCollector
Results

• In both cases (PrimeFinder and FileSizer), with the Actor Model, we get
  • comparable performance to the previous solutions
  • much simpler code
    • no locks
  • all code written from single threaded standpoint
    • allows for use of mutable variables with predictable behavior
Summary

• Reviewed the basics of the Actor model
  • Independent actors (which can be assigned to threads like tasks)
    • with mutable state that is NOT shared
      • with predictable semantics since the actor is single threaded
    • communicating with other actors by passing immutable messages
  • These constructs enable the isolated mutability approach to concurrency

• You get great performance with a very simple and straightforward model
  • no thread allocation, no task decomposition, no locks
Coming Up Next

• Lecture 25: Advanced Actor Model
• Lecture 26: Creating Agile Software