Lecture 21: Introduction to the Actor Model

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu
Worksheet #20: Sequential->Parallel Spanning Tree Algorithm

Insert finish, async, and atomic (includes a compareAndSet) constructs (pseudocode is fine) to convert the sequential spanning tree algorithm to a parallel algorithm

```java
1. class V {
2.     V [] neighbors; // adjacency list for input graph
3.     V parent; // output value of parent in spanning tree
4.     
5.     boolean makeParent(V n) {
6.         if (parent == null) { parent = n; return true; }
7.         else return false; // return true if n became parent
8.     } // makeParent
9.     
10.    void compute() {
11.        for (int i=0; i<neighbors.length; i++) {
12.            final V child = neighbors[i];
13.            if (child.makeParent(this))
14.                child.compute(); // recursive call
15.        }
16.    } // compute
17. } // class V
18. . . . // main program
19. root.parent = root; // Use self-cycle to identify root
20. root.compute();
21. . . .
```
Atomic Variables represent a special (and more efficient) case of object-based isolation

1. class V {
2.     V [] neighbors; // adjacency list for input graph
3.     AtomicReference<V> parent; // output value of parent in spanning tree
4.     boolean makeParent(final V n) {
5.         // compareAndSet() is a more efficient implementation of
6.         // object-based isolation
7.         return parent.compareAndSet(null, n);
8.     } // makeParent
9.     void compute() {
10.        for (int i=0; i<neighbors.length; i++) {
11.            final V child = neighbors[i];
12.            if (child.makeParent(this))
13.                async(() -> { child.compute(); }); // escaping async
14.        }
15.     } // compute
16. } // class V
17. ...
18. root.parent = root; // Use self-cycle to identify root
19. finish(() -> { root.compute(); });
20. ...
A C T O R S: an alternative approach to isolation, atomics

• An actor is an autonomous, interacting component of a parallel system.
• An actor has:
  — an immutable identity (global reference)
  — a single logical thread of control
  — mutable local state (isolated by default)
  — procedures to manipulate local state (interface)
The Actor Model: Fundamentals

• An actor may:
  — process messages
  — change local state
  — create new actors
  — send messages
Actor Model

• A message-based concurrency model to manage mutable shared state
  — First defined in 1973 by Carl Hewitt
  — Further theoretical development by Henry Baker and Gul Agha

• Key Ideas:
  — Everything is an Actor!
  — Analogous to “everything is an object” in OOP
  — Encapsulate shared state in Actors
  — Mutable state is not shared - i.e., no data races

• Other important features
  — Asynchronous message passing
  — Non-deterministic ordering of messages
Actor Life Cycle

**Actor states**

- **New**: Actor has been created
  — e.g., email account has been created, messages can be received

- **Started**: Actor can process messages
  — e.g., email account has been activated

- **Terminated**: Actor will no longer process messages
  — e.g., termination of email account after graduation
Actor Analogy - Email

• Email accounts are a good simple analogy to Actors

• Account A2 can send information to account A1 via an email message

• A1 has a mailbox to store all incoming messages

• A1 can read (i.e. process) one email at a time
  — At least that is what normal people do :) 

• Reading an email can change how you respond to a subsequent email
  — e.g. receiving pleasant news while reading current email can affect the response to a subsequent email
Using Actors in HJ-Lib

- Create your custom class which extends `edu.rice.hj.runtime.actors.Actor<T>`, and implement the `void process()` method (type parameter `T` specifies message type)
  ```java
class MyActor extends Actor<T> {
    protected void process(T message) {
      println("Processing " + message);
    }
  }
```

- Instantiate and start your actor
  ```java
  Actor<Object> anActor = new MyActor();
anActor.start();
  ```

- Send messages to the actor (can be performed by actor or non-actor)
  ```java
  anActor.send(aMessage); // aMessage can be any object in general
  ```

- Use a special message to terminate an actor
  ```java
  protected void process(Object message) {
    if (message.someCondition()) exit();
  }
  ```

- Actor execution implemented as async tasks
  Can use `finish` to await completion of an actor, if the actor is start-ed inside the `finish`. 
void \texttt{process(MessageType theMsg)} // Specification of actor’s "behavior" when processing messages

void \texttt{send(MessageType msg)} // Send a message to the actor

void \texttt{start()} // Cause the actor to start processing messages
void \texttt{onPreStart()} // Convenience: specify code to be executed before actor is started
void \texttt{onPostStart()} // Convenience: specify code to be executed after actor is started

void \texttt{exit()} // Actor calls \texttt{exit()} to terminate itself
void \texttt{onPreExit()} // Convenience: specify code to be executed before actor is terminated
void \texttt{onPostExit()} // Convenience: specify code to be executed after actor is terminated

// Next lecture
void \texttt{pause()} // Pause the actor, i.e. the actors stops processing messages in its mailbox
void \texttt{resume()} // Resume a paused actor, i.e. actor resumes processing messages in mailbox

See \url{http://www.cs.rice.edu/~vs3/hjlib/doc/edu/rice/hj/runtime/actors/Actor.html} for details
Hello World Example

```java
1. public class HelloWorld {
2.   public static void main(final String[] args) {
3.       finish(() -> {
4.         EchoActor actor = new EchoActor();
5.         actor.start(); // don’t forget to start the actor
6.         actor.send("Hello"); // asynchronous send (returns immediately)
7.         actor.send("World"); // Non-actors can send messages to actors
8.         actor.send(EchoActor.STOP_MSG);
9.       });
10.      println("EchoActor terminated.")
11.   }
12. private static class EchoActor extends Actor<Object> {
13.     static final Object STOP_MSG = new Object();
14.     private int messageCount = 0;
15.     protected void process(final Object msg) {
16.         if (STOP_MSG.equals(msg)) {
17.             println("Message-" + messageCount + ": terminating.");
18.             exit(); // never forget to terminate an actor
19.         } else {
20.             messageCount += 1;
21.             println("Message-" + messageCount + ": " + msg);
22.         }
23.     } } }
```

Though sends are asynchronous, many actor libraries (including HJLib) preserve the order of messages between the same sender actor/task and the same receiver actor.
Integer Counter Example

Without Actors:
1. int counter = 0;
2. public void foo() {
3.   // do something
4.   isolated(() -> {
5.     counter++;
6.   });
7.   // do something else
8. }
9. public void bar() {
10.   // do something
11.   isolated(() -> {
12.     counter--;
13.   });
14. }

With Actors:
15. class Counter extends Actor<Message> {
16.   private int counter = 0; // local state
17.   protected void process(Message msg) {
18.     if (msg instanceof IncMessage) {
19.       counter++;
20.     } else if (msg instanceof DecMessage){
21.       counter--;
22.   }
23.   }
24.   Counter counter = new Counter();
25.   counter.start();
26.   public void foo() {
27.     // do something
28.     counter.send(new IncrementMessage(1));
29.     // do something else
30.   }
31.   public void bar() {
32.     // do something
33.     counter.send(new DecrementMessage(1));
34.   }

12
ThreadRing (Coordination) Example

1. `finish()` -> {
2.   int threads = 4;
3.   int numberOfHops = 10;
4.   ThreadRingActor[] ring =
          new ThreadRingActor[threads];
5.   for(int i=threads-1; i>=0; i--) {
6.     ring[i] = new ThreadRingActor(i);
7.     ring[i].start();
8.     if (i < threads - 1) {
9.       ring[i].nextActor(ring[i + 1]);
10.    } }
11.   ring[threads-1].nextActor(ring[0]);
12.   ring[0].send(numberOfHops);
13. } // finish

1. class ThreadRingActor
2.   extends Actor<Integer> {
3.     private Actor<Integer> nextActor;
4.     private final int id;
5.     ...
6.   public void nextActor(Actor<Object> nextActor) {...}
8.   protected void process(Integer n) {
9.     if (n > 0) {
10.        println("Thread-" + id +
11.            " active, remaining = " + n);
12.        nextActor.send(n - 1);
13.    } else {
14.        println("Exiting Thread-" + id);
15.        nextActor.send(-1);
16.        exit();
17.    } } }
Announcements & Reminders

• Lab 5 is tomorrow (setup before lab, try logging into NOTS)
• HW #3 CP 1 is due Wednesday, March 24th at 11:59pm
• Quiz for Unit 5 will be due Monday, March 29th at 11:59pm
Worksheet #21: Interaction between finish and actors

What output will be printed if the end-finish operation from slide 15 is moved from line 13 to line 11 as shown below?

```
1. finish() -> {
2.     int threads = 4;
3.     int numberOfHops = 10;
4.     ThreadRingActor[] ring = new ThreadRingActor[threads];
5.     for(int i=threads-1;i>=0; i--) {
6.         ring[i] = new ThreadRingActor(i);
7.         ring[i].start();
8.         if (i < threads - 1) {
9.             ring[i].nextActor(ring[i + 1]);
10.        }
11.    } } // finish
12. ring[threads-1].nextActor(ring[0]);
13. ring[0].send(numberOfHops);
14.
```