COMP 322: Fundamentals of Parallel Programming

Lecture 31: Introduction to the Actor Model

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu
How to prevent data races when accessing shared data?

• Preventing data races on shared mutable data
  — Future, get
  — Async, finish
  — DDTs, asyncAwait
  — Atomics

• The predominant approach to ensure mutual exclusion for concurrent data structures is to enclose the code region in a critical section.
  — Global and object-based isolated statements
  — Java synchronized methods and statements
  — Java unstructured locks
Actors: 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

- **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`. 
Summary of HJlib Actor API

```c
void process(MessageType theMsg) // Specification of actor's "behavior" when processing messages

void send(MessageType msg) // Send a message to the actor

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

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

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

```
Hello World Example

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. }
13. private static class EchoActor extends Actor<Object> {
14.     static final Object STOP_MSG = new Object();
15.     private int messageCount = 0;
16.     protected void process(final Object msg) {
17.         if (STOP_MSG.equals(msg)) {
18.             println("Message-" + messageCount + ": terminating.");
19.             exit(); // never forget to terminate an actor
20.         } else {
21.             messageCount += 1;
22.             println("Message-" + messageCount + ": " + msg);
23.         }
24.     }
25. } }
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. `}`
ThreadRing (Coordination) Example

1. \texttt{finish()} \rightarrow 
2. \hspace{0.5em} int \texttt{threads} = 4;
3. \hspace{0.5em} int \texttt{numberOfHops} = 10;
4. \hspace{0.5em} \texttt{ThreadRingActor[]} \texttt{ring} = 
5. \hspace{2.5em} new \ \texttt{ThreadRingActor[\texttt{threads}]};
6. \hspace{0.5em} for(int \texttt{i=threads-1;}\texttt{i>=0;} \texttt{i--}) \{ 
7. \hspace{1.5em} \texttt{ring[i]} = new \ \texttt{ThreadRingActor(i)};
8. \hspace{1.5em} \texttt{ring[i].start();}
9. \hspace{1.5em} \texttt{if (i < \texttt{threads - 1}) \{ 
10. \hspace{2.5em} \texttt{ring[i].nextActor(ring[i + 1]);}
11. \hspace{1.5em} \}} }
12. \hspace{0.5em} \texttt{ring[\texttt{threads-1]}.nextActor(ring[0]);}
13. \hspace{0.5em} \texttt{ring[0].send(numberOfHops);}
14. \hspace{0.5em} \} // \texttt{finish}

\begin{center}
\begin{tikzpicture}
\node (0) at (0,0) {0};
\node (1) at (1.5,0) {1};
\node (2) at (0,-1.5) {2};
\node (3) at (1.5,-1.5) {3};
\draw[->] (0) to (1);
\draw[->] (1) to (2);
\draw[->] (2) to (3);
\draw[->] (3) to (0);
\end{tikzpicture}
\end{center}

1. \texttt{class ThreadRingActor}
2. \hspace{0.5em} extends \ \texttt{Actor<Integer> \{ 
3. \hspace{1.5em} private \ \texttt{Actor<Integer> \texttt{nextActor;}}
4. \hspace{1.5em} private \ \texttt{final int \texttt{id;}}
5. \hspace{1.5em} \hspace{0.5em} ... 
6. \hspace{1.5em} \}} \texttt{public \ void \texttt{nextActor(}
7. \hspace{2.5em} \ \texttt{Actor<Object> \texttt{nextActor)} \{ ... \}}
8. \hspace{0.5em} \}} \texttt{protected \ void \texttt{process(Integer \texttt{n)} \{ 
9. \hspace{1.5em} \texttt{if (n > 0) \{ 
10. \hspace{2.5em} \texttt{println(}"Thread-" + \texttt{id} + 
11. \hspace{2.5em} " active, remaining = " + \texttt{n});
12. \hspace{2.5em} \texttt{nextActor.send(\texttt{n - 1});}
13. \hspace{2.5em} \}} \texttt{else \{ 
14. \hspace{3.5em} \texttt{println(}"Exiting Thread-"+ \texttt{id});
15. \hspace{3.5em} \texttt{nextActor.send(-1);}
16. \hspace{3.5em} \texttt{exit();}
17. \hspace{2.5em} \}} \} \} \}