COMP 322: Fundamentals of Parallel Programming

Lecture 28: Introduction to the Actor Model

Mack Joyner and Zoran Budimlić
{mjoyner, zoran}@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, 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`.
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

```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<
16. `protected void process(Message msg) {
17. `if (msg instanceof IncMessage) {
18. `counter++;
19. `} else if (msg instanceof DecMessage){
20. `counter--;`
21. `}
22. `}
23. `Counter counter = new Counter();
24. `counter.start();
25. `public void foo() {
26. `counter.send(new IncrementMessage(1));
27. `}
28. `public void bar() {
29. `counter.send(new DecrementMessage(1));
30. `}
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) {...}
7. `  protected void process(Integer n) {
8.   if (n > 0) {
9.     println("Thread-" + id +
10.       "active, remaining = " + n);
11.     nextActor.send(n - 1);
12.   } else {
13.     println("Exiting Thread-"+ id);
14.     nextActor.send(-1);
15.     exit();
16.   } }
17. } }`
Announcements & Reminders

- Quiz #6 is due Wednesday, March 30th at 11:59pm
- Hw #4 is due Friday, Apr. 1st at 11:59pm (expected speedups have changed in handout)