COMP 322: Fundamentals of Parallel Programming

Lecture 22: Introduction to the Actor Model

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

http://comp322.rice.edu
Q: Compute the WORK and CPL metrics for this program with a **global isolated** construct. Indicate if your answer depends on the execution order of isolated constructs.

```java
1.   finish(() -> { 
2.       for (int i = 0; i < 5; i++) { 
3.           async(() -> { 
4.               doWork(2); 
5.           isolated(() -> { doWork(1); }); 
6.               doWork(2); 
7.           }); // async 
8.       } // for 
9.   }); // finish
```

Answer: WORK = 25, CPL = 9. These metrics do not depend on the execution order of isolated constructs.
Q: Compute the WORK and CPL metrics for this program with an object-based isolated construct. Indicate if your answer depends on the execution order of isolated constructs.

Answer: WORK = 25, CPL = 7. These metrics depend on the execution order of isolated constructs.
Actors: an alternative approach to isolation

• 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 processes 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 HJlib

• 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

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. }
24. }
```

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:

```java
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:

```java
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. }
```

- Can also use atomic variables instead of isolated construct.
ThreadRing (Coordination) Example

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

13. class ThreadRingActor
14.     extends Actor<Integer> {
15.         private Actor<Integer> nextActor;
16.         private final int id;
17.         ...
18.         public void nextActor(
19.             Actor<Object> nextActor) {...}
20.       protected void process(Integer n) {
21.           if (n > 0) {
22.             println("Thread-" + id +
23.                 " active, remaining = " + n);
24.             nextActor.send(n - 1);
25.           } else {
26.             println("Exiting Thread-" + id);
27.             nextActor.send(-1);
28.             exit();
29.           } }
30.   }

![Diagram of ThreadRing Actor structure]
\[ \pi = 4 \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \cdots. \]

- Use Main-Worker technique:

\[\text{Main} \quad \text{accumulates approximation of PI}\]

- Series 1 - 100
- Series 101 - 200
- Series 1201 - 1300
- Series 1301 - 1400
- Series 1401 - 1500
- Series 2501 - 2600
- Series 2601 - 2700

Source: http://www.enotes.com/topic/Pi
class Main extends Actor<Object> {
    private double result = 0; private int nrMsgsReceived = 0;
    private Worker[] workers;
    Main(nrWrkrs, nrEls, nrMsgs) {...} // constructor
    protected void onPostStart() {
      // Create and start workers
      workers = new Worker[nrWrkrs];
      for (int i = 0; i < nrwrkrs; i++) {
        workers[i] = new Worker();
        workers[i].start();
      }
      // Send messages to workers
      for (int j = 0; j < nrMsgs; j++) {
        someWrkr = ... ; // Select worker for message j
        someWrkr.send(new Work(...));
      }
    } // start()
protected void `onPostExit()` {
    for (int i = 0; i < nrWrkrs; i++)
        workers[i].send(new Stop());
} // post-exit()

protected void `process(final Object msg)` {
    if (msg instanceof Result) {
        result += ((Result) msg).result;
        nrMsgsReceived += 1;
        if (nrMsgsReceived == nrMsgs) exit();
    }
    // Handle other message cases here
} // process()
} // Main actor

... 

// Start of program
Main star = new Main(w, e, m);
finish(() -> { star.start(); });
println("PI = "+ star.getResult());
Pi Calculation --- Worker Actor

```java
1. class Worker extends Actor<Object> {
2.     protected void process(final Object msg) {
3.         if (msg instanceof Stop)
4.             exit();
5.         else if (msg instanceof Work) {
6.             Work wm = (Work) msg;
7.             double result = calculatePiFor(wm.start, wm.end)
8.                 .sender().send(new ResultMessage(result));
9.         } } // process()
10. }
11. 
12. private double calculatePiFor(int start, int end) {
13.     double acc = 0.0;
14.     for (int k = start; k < end; k++) {
15.         acc += 4.0 * (1 - (k % 2) * 2) / (2 * k + 1);
16.     }
17.     return acc;
18. } // Worker
```

\[
\pi = 4 \sum_{k=0}^{e-1} \frac{(-1)^k}{2k + 1}
\]
Limitations of Actor Model

- Deadlocks possible
  - Deadlock occurs when all started (but non-terminated) actors have empty mailboxes
- Data races possible when messages include shared objects
- Simulating synchronous replies requires some effort
  - e.g., does not support `addAndGet()`
- Implementing truly concurrent data structures is hard
  - No parallel reads, no reductions/accumulators
- Difficult to achieve global consensus
  - Finish and barriers not supported as first-class primitives

==> Some of these limitations can be overcome by using a hybrid model that combines task parallelism with actors (more on this in the next lecture!)