
COMP 322: Fundamentals of Parallel Programming

Lecture 22: Introduction to the Actor Model

Srđan Milaković

Department of Computer Science, Rice University
{srdan.milakovic}@rice.edu

<http://comp322.rice.edu/>



Worksheet #21a solution:

Abstract Metrics with Object-based Isolated Constructs

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.

```
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.



Worksheet #21b solution:

Abstract Metrics with 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.

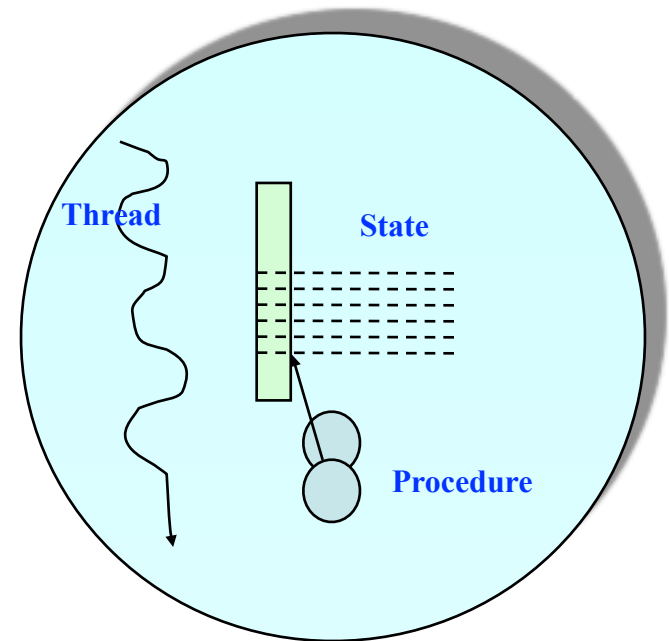
```
1.  finish(() -> {
2.      // Assume X is an array of distinct objects
3.      for (int i = 0; i < 5; i++) {
4.          async(() -> {
5.              doWork(2);
6.              isolated(X[i], X[i+1],
7.                  () -> { doWork(1); });
8.              doWork(2);
9.          }); // async
10.     } // for
11. }); // finish
```

Answer: *WORK* = 25, *CPL* = 7. These metrics do not depend on the execution order of object-based isolated constructs.



Actors: an alternative approach to isolation

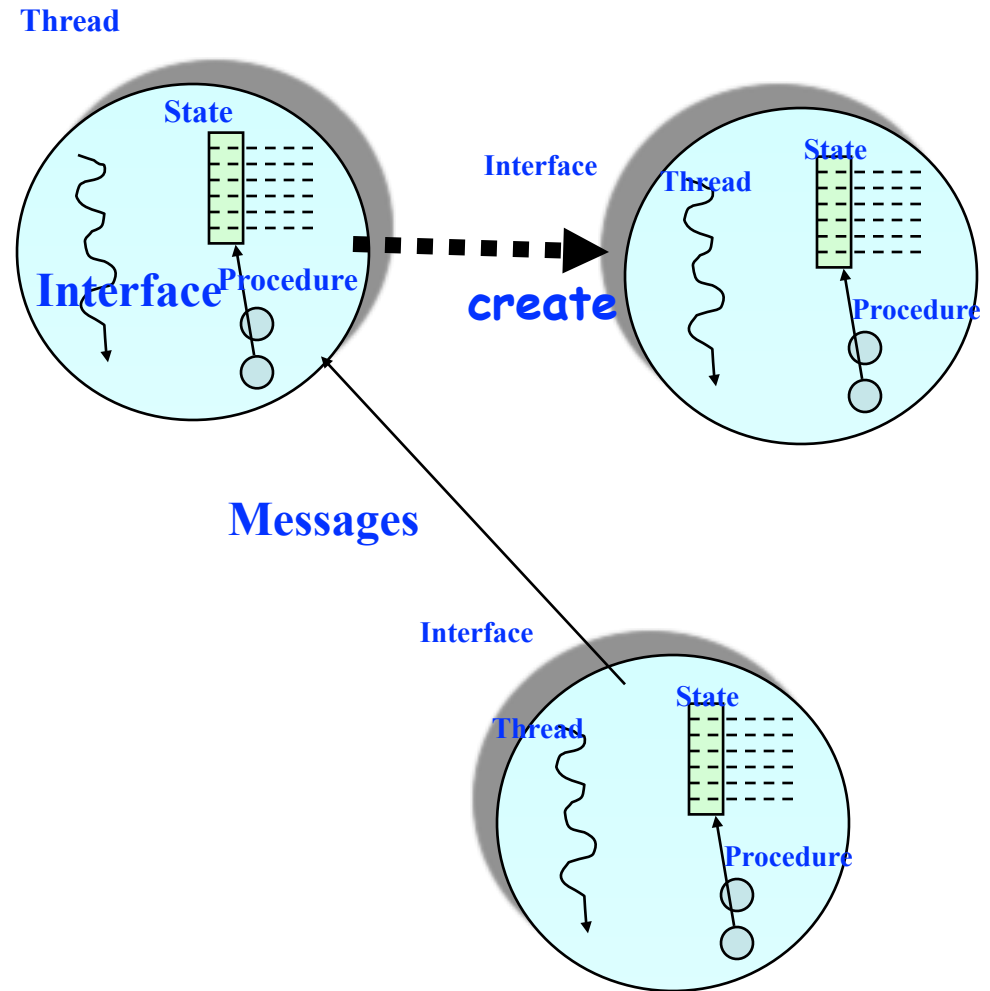
- An actor is an autonomous, interacting component of a parallel system.
- An actor has:
 - an immutable identity (global reference)
 - *a single logical thread of control*
 - mutable local state (isolated by default)
 - procedures to manipulate local state (interface)



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

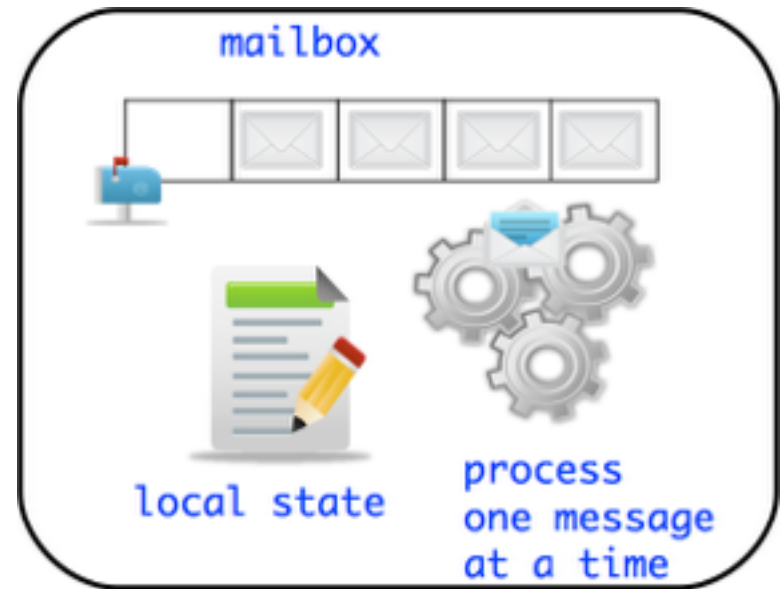


Actor Life Cycle



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**
- **Actor creation (stretching the analogy)**
 - **Create a new email account that can send/receive messages**



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)

```
class MyActor extends Actor<T> {  
    protected void process(T message) {  
        println("Processing " + message);  
    }  
}
```

- Instantiate and start your actor

```
Actor<Object> anActor = new MyActor();  
anActor.start();
```

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

- Use a special message to terminate an actor

```
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

// Later today

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

See <http://www.cs.rice.edu/~vs3/hjlib/doc/edu/rice/hj/runtime/actors/Actor.html> for details



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.    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.    }
25. }
```

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. }
```

- Can also use atomic variables instead of isolated construct

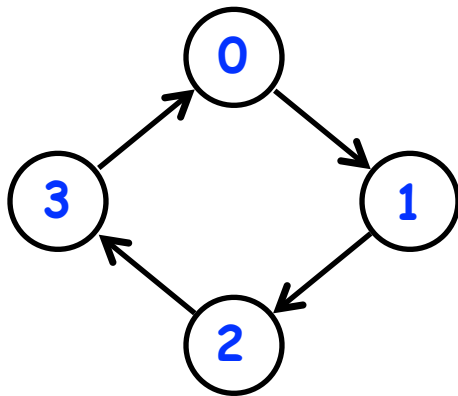
With Actors:

```
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. 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.         if (i < threads - 1) {
10.            ring[i].nextActor(ring[i + 1]);
11.        } }
12.     ring[threads-1].nextActor(ring[0]);
13.     ring[0].send(numberOfHops);
14. }); // finish
```



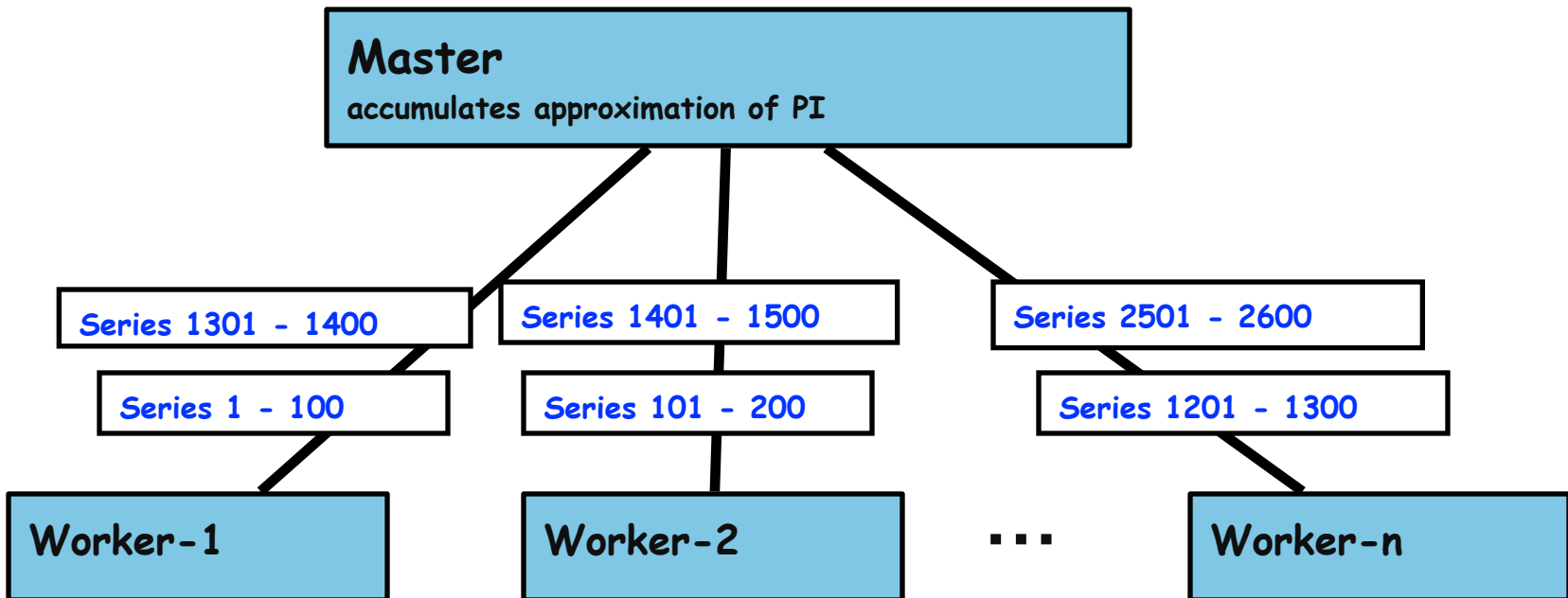
```
14. class ThreadRingActor
15.     extends Actor<Integer> {
16.     private Actor<Integer> nextActor;
17.     private final int id;
18.     ...
19.     public void nextActor(
20.         Actor<Object> nextActor) {...}
21.
22.     protected void process(Integer n) {
23.         if (n > 0) {
24.             println("Thread-" + id +
25.                 " active, remaining = " + n);
26.             nextActor.send(n - 1);
27.         } else {
28.             println("Exiting Thread-" + id);
29.             nextActor.send(-1);
30.             exit();
31.         } } }
```



Pi Computation Example

$$\pi = 4 \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \dots$$

- Use Master-Worker technique:



Source: <http://www.enotes.com/topic/Pi>



Pi Calculation --- Master Actor

```
1. class Master extends Actor<Object> {
2.     private double result = 0; private int nrMsgsReceived = 0;
3.     private Worker[] workers;
4.     Master(nrWrkrs, nrEls, nrMsgs) {...} // constructor
5.     protected void onPostStart() {
6.         // Create and start workers
7.         workers = new Worker[nrWrkrs];
8.         for (int i = 0; i < nrwrkrs; i++) {
9.             workers[i] = new Worker();
10.            workers[i].start();
11.        }
12.        // Send messages to workers
13.        for (int j = 0; j < nrMsgs; j++) {
14.            someWrkr = ... ; // Select worker for message j
15.            someWrkr.send(new Work(...));
16.        }
17.    } // start()
```



Pi Calculation --- Master Actor (contd)

```
19.     protected void onPostExit() {
20.         for (int i = 0; i < nrWrkrs; i++)
21.             workers[i].send(new Stop());
22.     } // post-exit()
23.     protected void process(final Object msg) {
24.         if (msg instanceof Result) {
25.             result += ((Result) msg).result;
26.             nrMsgsReceived += 1;
27.             if (nrMsgsReceived == nrMsgs) exit();
28.         }
29.         // Handle other message cases here
30.     } // process()
31. } // Master
32. . . .
33. // Main program
34. Master master = new Master(w, e, m);
35. finish(() -> { master.start(); });
36. println("PI = " + master.getResult());
```



Pi Calculation --- Worker Actor

```
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.             master.send(new ResultMessage(result));
9.         } } // process()
10.
11.     private double calculatePiFor(int start, int end) {
12.         double acc = 0.0;
13.         for (int k = start; k < end; k++) {
14.             acc += 4.0 * (1 - (k % 2) * 2) / (2 * k + 1);
15.         }
16.         return acc;
17.     }
18. } // Worker
```

$$4 \sum_{k=S}^{e-1} \frac{(-1)^k}{2k+1}$$



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!)



Worksheet #22:

Interaction between finish and actors

Name: _____

Net ID: _____

What output will be printed if the end-finish operation from slide 13 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.
```

