COMP 322: Fundamentals of Parallel Programming

Lecture 24: Actors (contd)

Vivek Sarkar, Eric Allen
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Worksheet #23 solution: Interaction between finish and actors

What would happen if the end-finish operation from slide 14 was moved from line 13 to line 11 as shown below?

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

Deadlock: the end-finish operation in line 11 waits for all the actors started in line 7 to terminate, but the actors are waiting for the message sequence initiated in line 13 before they call exit().

Actor Hello World Example (Recap)

```
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");
8.       actor.send(EchoActor.STOP_MSG);
9.     });
10.  }
11.  private static class EchoActor extends Actor<Object> {
12.    static final Object STOP_MSG = new Object();
13.    private int messageCount = 0;
14.    protected void process(final Object msg) {
15.      if (STOP_MSG.equals(msg)) {
16.         println("Message-" + messageCount + ": terminating.");
17.         exit(); // never forget to terminate an actor
18.       } else {
           messageCount += 1;
19.         println("Message-" + messageCount + ": " + msg);
20.  } } } }
```

HJ Actor library preserves order of messages between same sender and receiver

Summary of Actor API

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

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

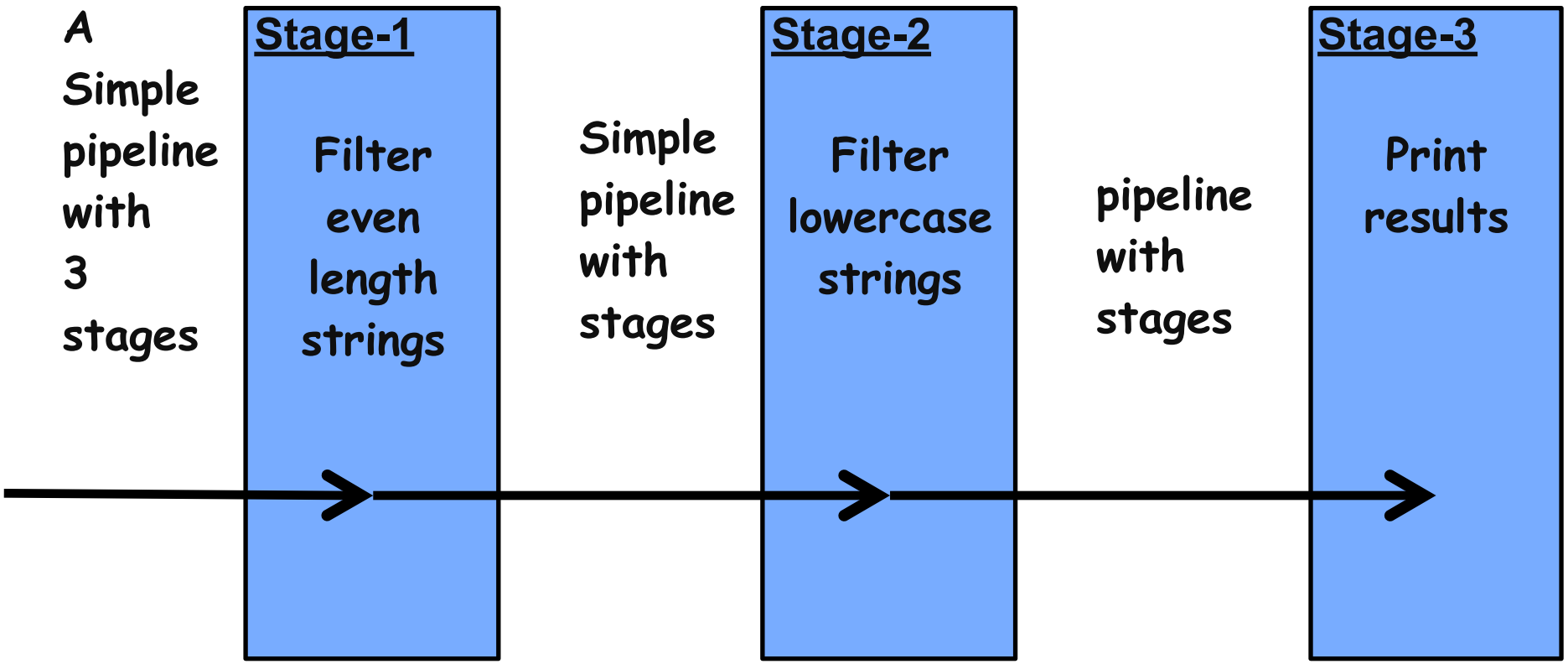
void start() // Cause the actor to start processing messages
void onPreStart() // Convenience: specify code to be executed before actor is started
void onPostStart() // Convenience: specify code to be executed after actor is started

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

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

Simple Pipeline using Actors



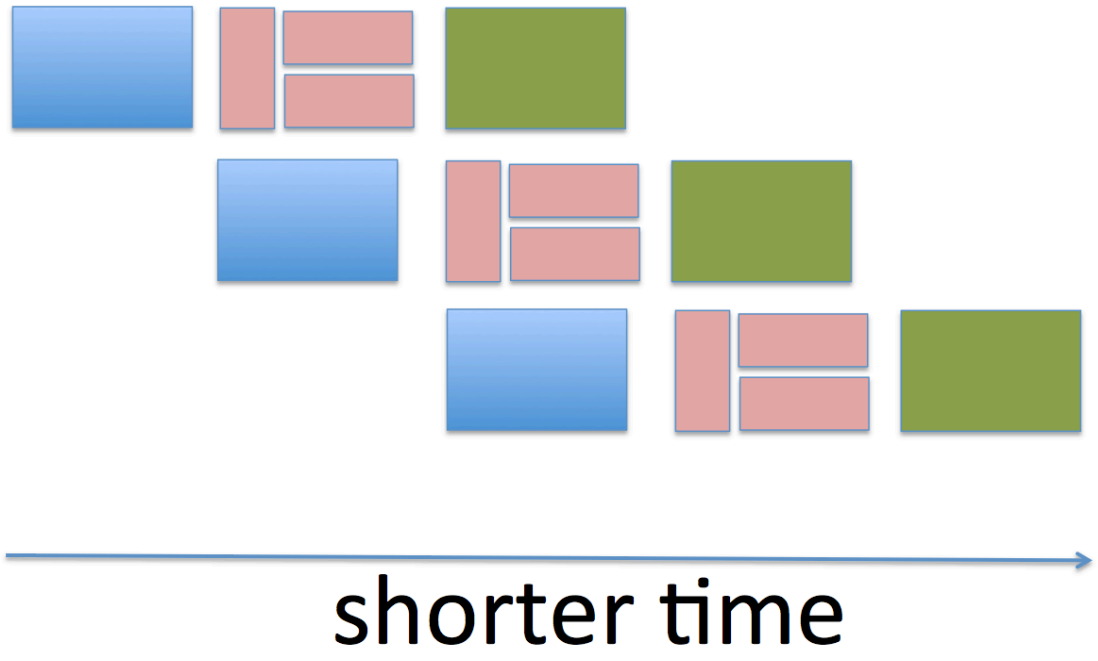
Parallelism with an Actor's process() method

- Use finish construct within process() body and spawn child tasks
- Take care not to introduce data races on local state!

```
1.  class ParallelActor1 extends Actor<Message> {
2.    void process(Message msg) {
3.      finish(() -> {
4.        async(() -> { S1; });
5.        async(() -> { S2; });
6.        async(() -> { S3; });
7.      });
8.    }
9.  }
```


Motivation for Parallelizing Actors

- Pipelined Parallelism
  - Reduce effects of slowest stage by introducing task parallelism.
  - Increases the throughput.



Example of Parallelizing Actors

```
1.    class ConsumerActor extends Actor<Object> {
2.      private double resultSoFar = 0;
3.      @Override
4.      protected void process(final Object theMsg) {
5.        if (theMsg != null) {
6.          final double[] dataArray = (double[]) theMsg;
7.          final double localRes = doComputation(dataArray);
8.          resultSoFar += localRes;
9.        } else { ... }
10.     }
11.     private double doComputation(final double[] dataArray) {
12.       final double[] localSum = new double[2];
13.       finish(() -> {
14.         final int length = dataArray.length;
15.         final int limit1 = length / 2;
16.         async(() -> {
17.           localSum[0] = doComputation(dataArray, 0, limit1);
18.         });
19.         localSum[1] = doComputation(dataArray, limit1, length);
20.       });
21.       return localSum[0] + localSum[1];
22.     }
23.   }
```


Parallelizing Actors in HJlib

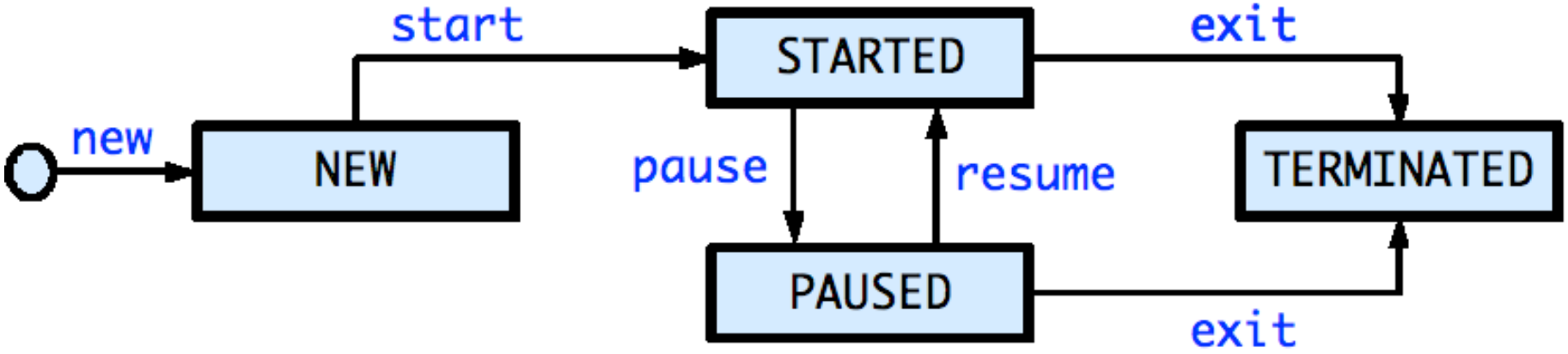
- Two techniques:
  - Use finish construct to wrap asyncs in message processing body
    - Finish ensures all spawned asyncs complete before next message returning from process()
  - Allow escaping asyncs inside process() method
    - **WAIT!** **Won't escaping asyncs violate the one-message-at-a-time rule in actors**
    - Solution: Use pause and resume

Parallelizing Actors in HJ

- Allow escaping asyncs inside process()

```
1.   class ParallelActor2 extends Actor<Message> {
2.     void process(Message msg) {
3.       pause(); // process() will not be called until a resume() occurs
4.       async(() -> { S1; }); // escaping async
5.       async(() -> { S2; }); // escaping async
6.       async(() -> {
7.         // This async must be completed before next message
8.         // Can also use async-await if you want S3 to wait for S1 & S2
9.         S3;
10.        resume();
11.      });
12.    }
13.  }
```


Hybrid Actors in HJ-Lib



- Paused state: actor will not process subsequent messages until it is resumed
- Resume actor when it is safe to process the next message
- Messages can accumulate in mailbox when actor is in PAUSED state (s in NEW state)

Actors: pause and resume

- pause() operation:
  - Is a non-blocking operation, i.e. allows the next statement to be executed.
  - Calling pause() when the actor is already paused is a no-op.
  - Once paused, the state of the actor changes and it will no longer process messages sent (i.e. call process(message)) to it until it is resumed.
- resume() operation:
  - Is a non-blocking operation.
  - Calling resume() when the actor is not paused is an error, the HJ runtime will throw a runtime exception.
  - Moves the actor back to the STARTED state
    - the actor runtime spawns a new asynchronous thread to start processing messages from its mailbox.

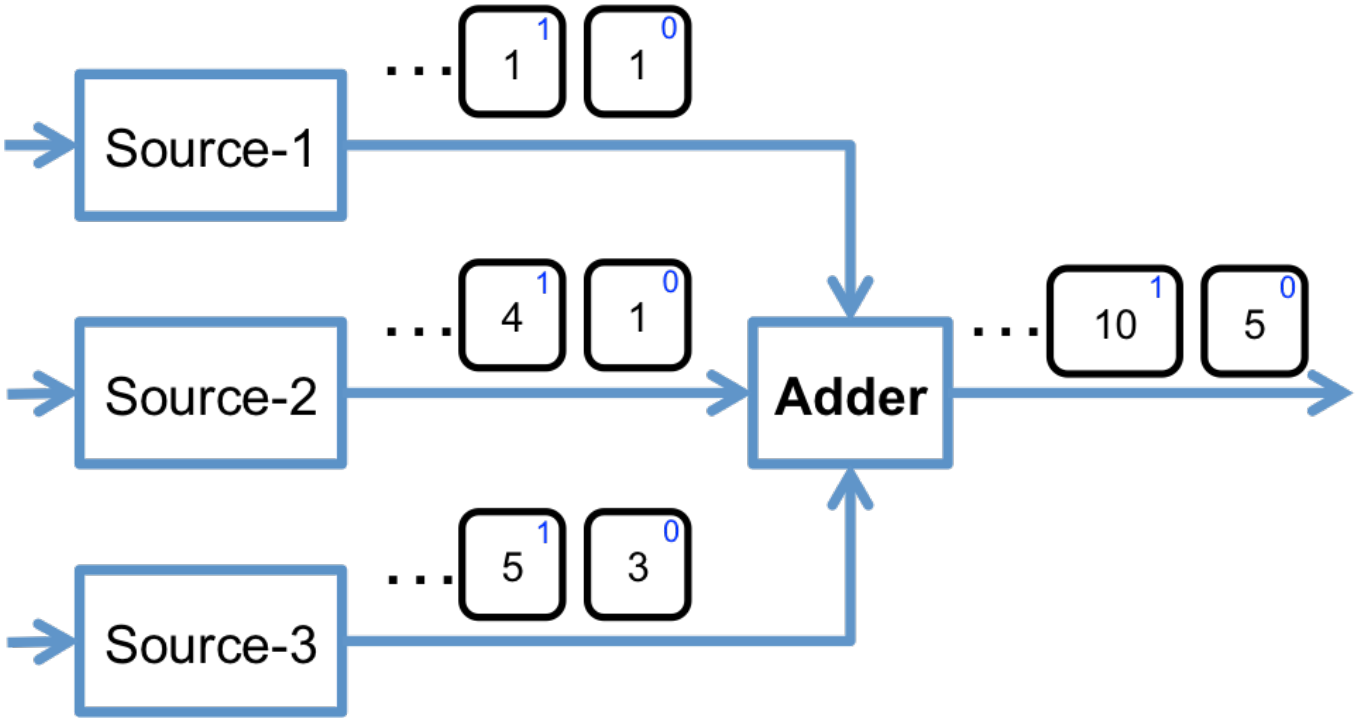
Synchronous Reply using Pause/Resume

- Actors are inherently asynchronous
- Synchronous replies require blocking operations e.g., asyncAwait

```
1.   class SynchronousReplyActor2 extends Actor<Message> {
2.     void process(Message msg) {
3.       if (msg instanceof Ping) {
4.         HjDataDrivenFuture<T> ddf = newDataDrivenFuture();
5.         otherActor.send(ddf);
6.         pause(); // when paused, the actor doesn't process messages
7.         asyncAwait(ddf, () -> { // processes synchronous reply
8.            T synchronousReply = ddf.get();
9.            // do some processing with synchronous reply
10.           resume(); // allow actor to process next message
11.        });
12.    } else if (msg instanceof ...) { ... } } }
```

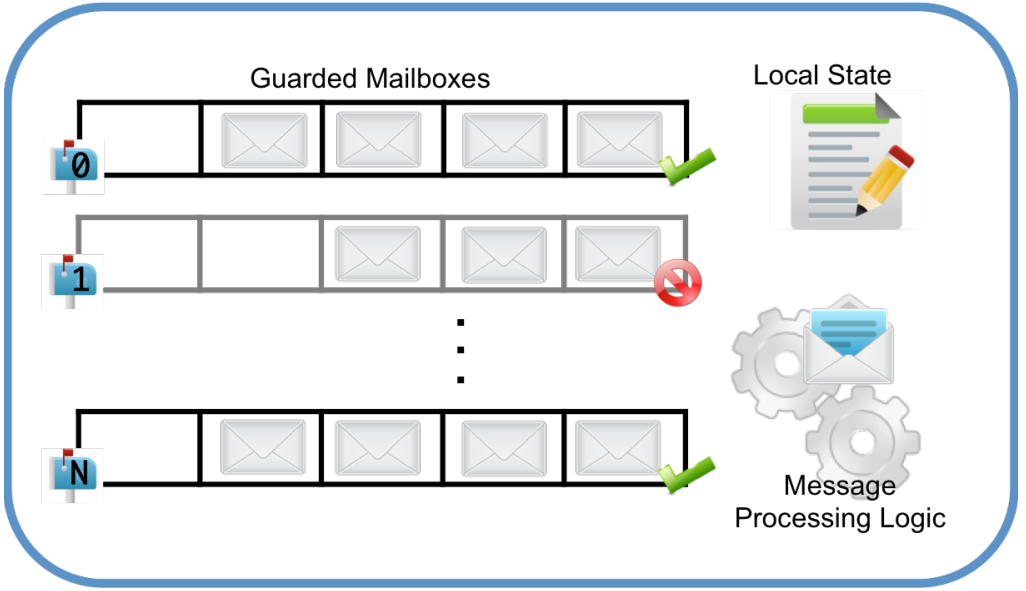

Join Patterns in Streaming Applications

- **Open question: how to implement an adder for 3 input streams using actors?**
- Messages from two or more data streams are combined together into a single message
- Joins need to match inputs from each source



From Actors to Selectors (Actors with multiple mailboxes)

- The basic idea behind pause() and resume() is to enable/disable processing of messages in an actor's mailbox
- This idea can be extended to *selectors*
  - —a selector is an actor with multiple mailboxes numbered 0…n-1
  - —s.send(i,msg) sends msg to mailbox i of selector s
  - —disable(i) disables mailbox i (like "pausing" mailbox i)
  - —enable(i) enables mailbox i (like "resuming" mailbox i)
  - —enableAll() enables all the mailboxes



Selector-based solution to Join Pattern

```
1.private static final class JoinSelector
2.   extends Selector<IntMessage> {
3.   . . .
4.   protected void process(final IntMessage theMsg) {
5.     messagesProcessed++;
6.     partialSum += theMsg.value;
7.     disable(theMsg.sourceIndex);
8.
9.     if (IntMessage.END == theMsg.value) exit();
10.     else if (messagesProcessed % numJoiners == 0) {
11.       nextInChain.send(partialSum);
12.       enableAll();
13.       partialSum = 0;
14.       seriesProcessed++;
15.     } // else if
16. }}
```