

COMP 322: Fundamentals of Parallel Programming

Lecture 32: Actors cont.

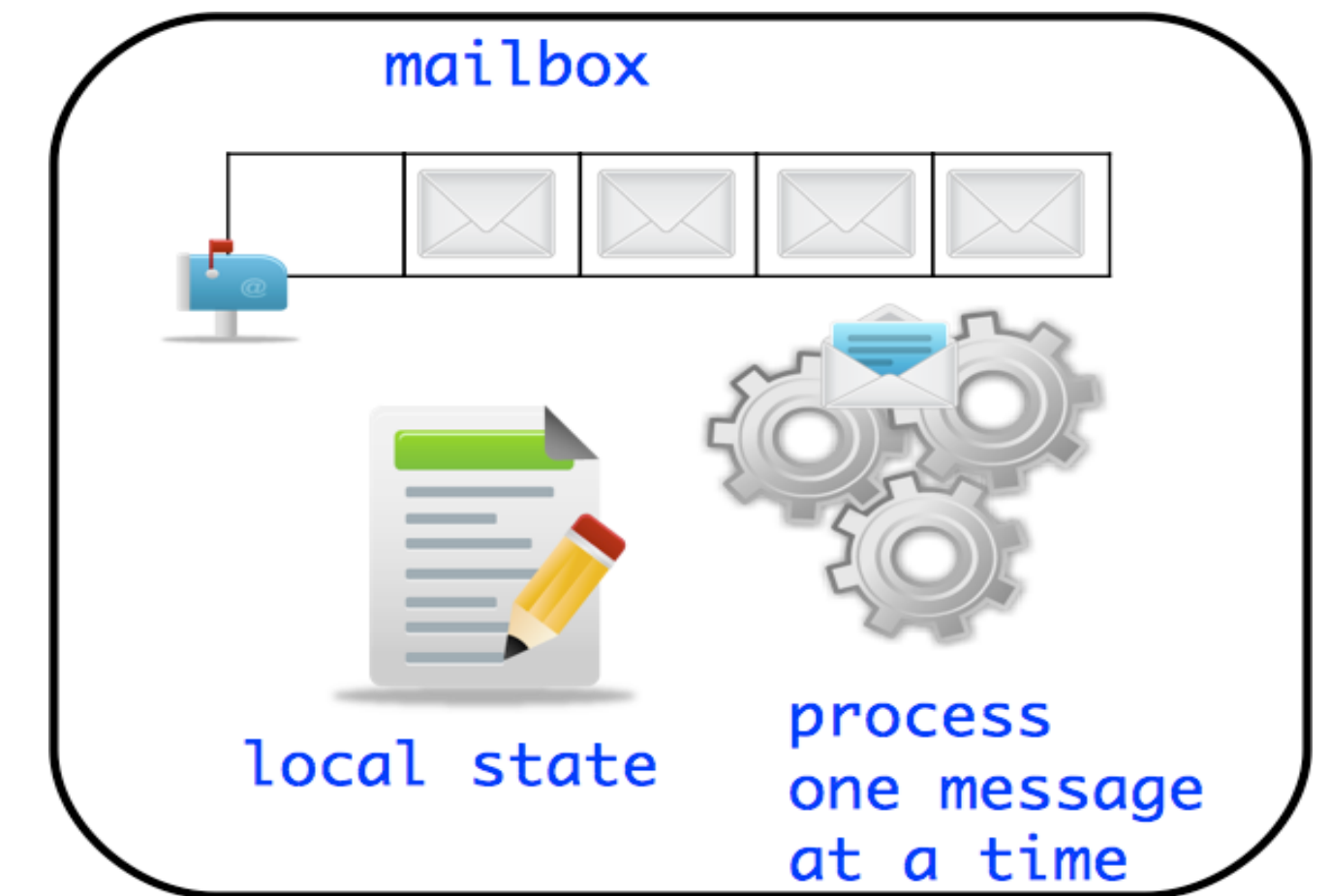
Mack Joyner
mjoyner@rice.edu

<http://comp322.rice.edu>

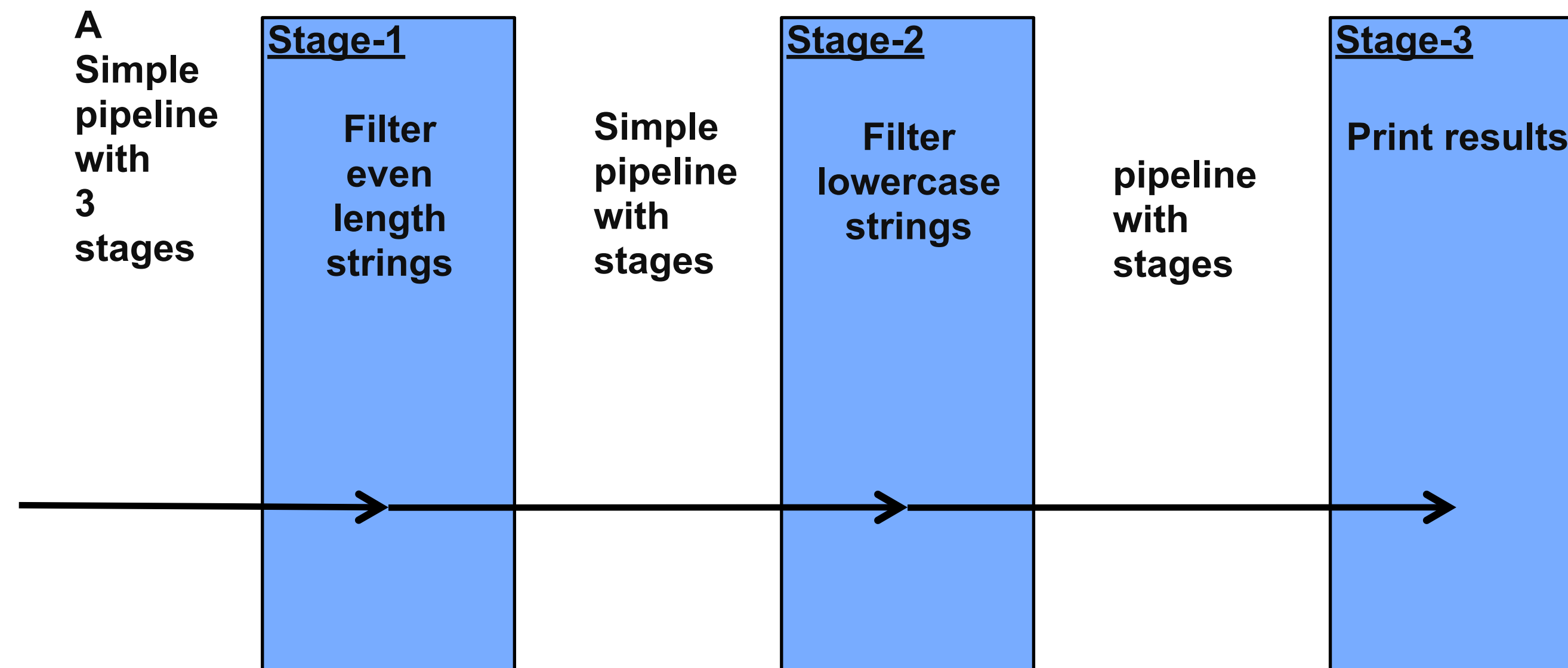


Recap of Actors

- Rely on asynchronous messaging
- Messages are sent to an actor using its `send()` method
- Messages queue up in the mailbox
- Messages are processed by an actor after it is started
- Messages are processed asynchronously
 - one at a time
 - using the body of `process()`

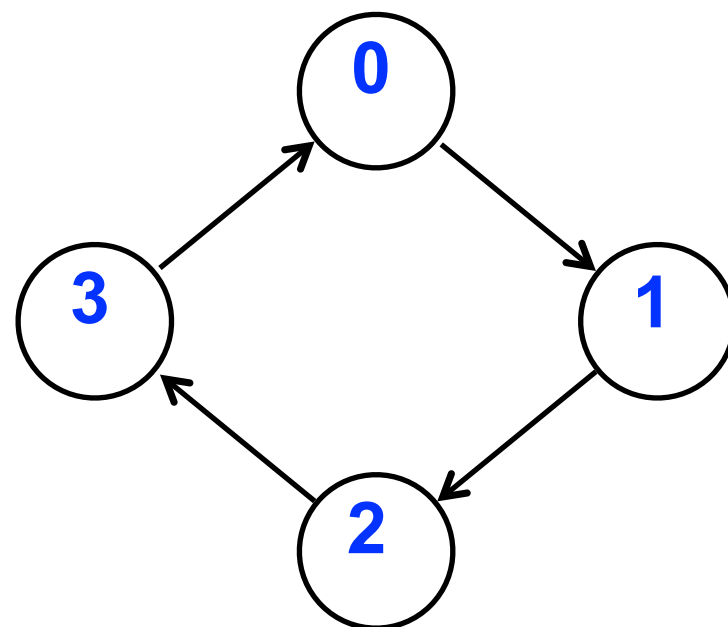


Simple Pipeline using Actors



ThreadRing (Coordination) Example

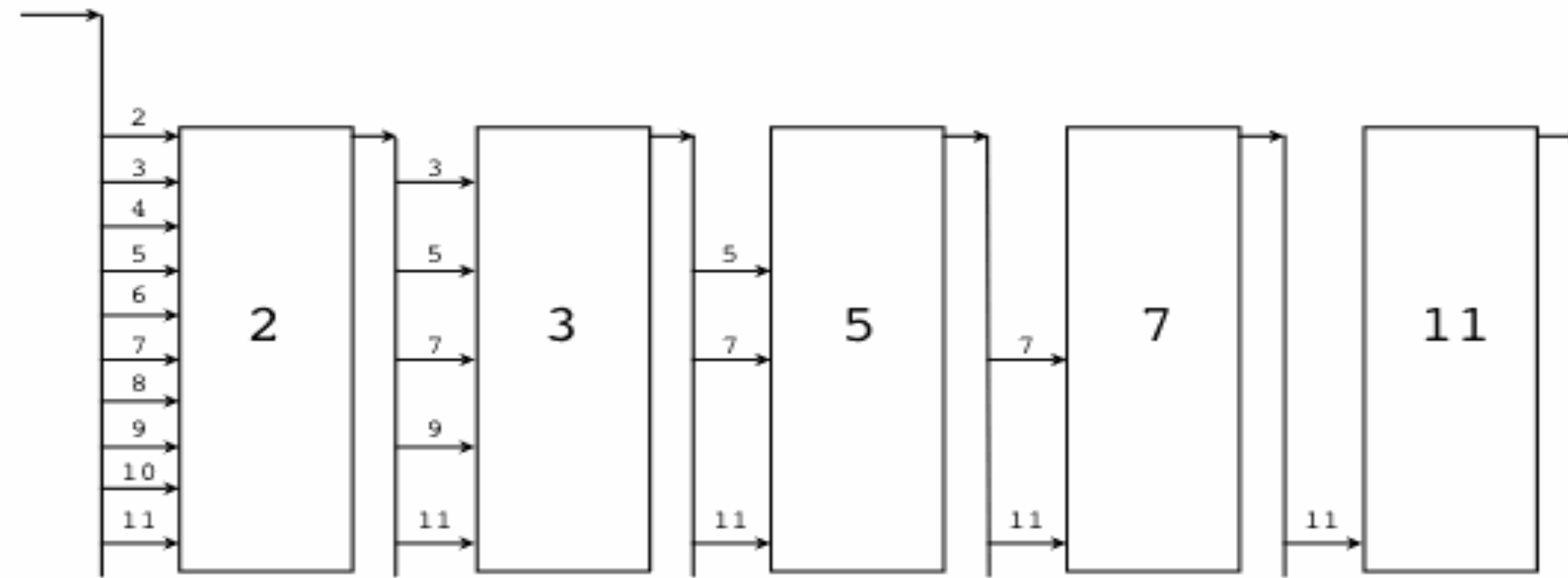
```
1. finish(() -> {
2.   int threads = 4;
3.   int numberOfHops = 10;
4.   ThreadRingActor[] ring =
       new ThreadRingActor[threads];
5.   for(int i=threads-1;i>=0; i--) {
6.     ring[i] = new ThreadRingActor(i);
7.     ring[i].start();
8.     if (i < threads - 1) {
9.       ring[i].nextActor(ring[i + 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.
8.   protected void process(Integer n) {
9.     if (n > 0) {
10.      println("Thread-" + id +
11.        " active, remaining = " + n);
12.      nextActor.send(n - 1);
13.    } else {
14.      println("Exiting Thread-" + id);
15.      nextActor.send(-1);
16.      exit();
17.    } } }
```



Sieve of Eratosthenes using Actors



Limitations of Actor Model

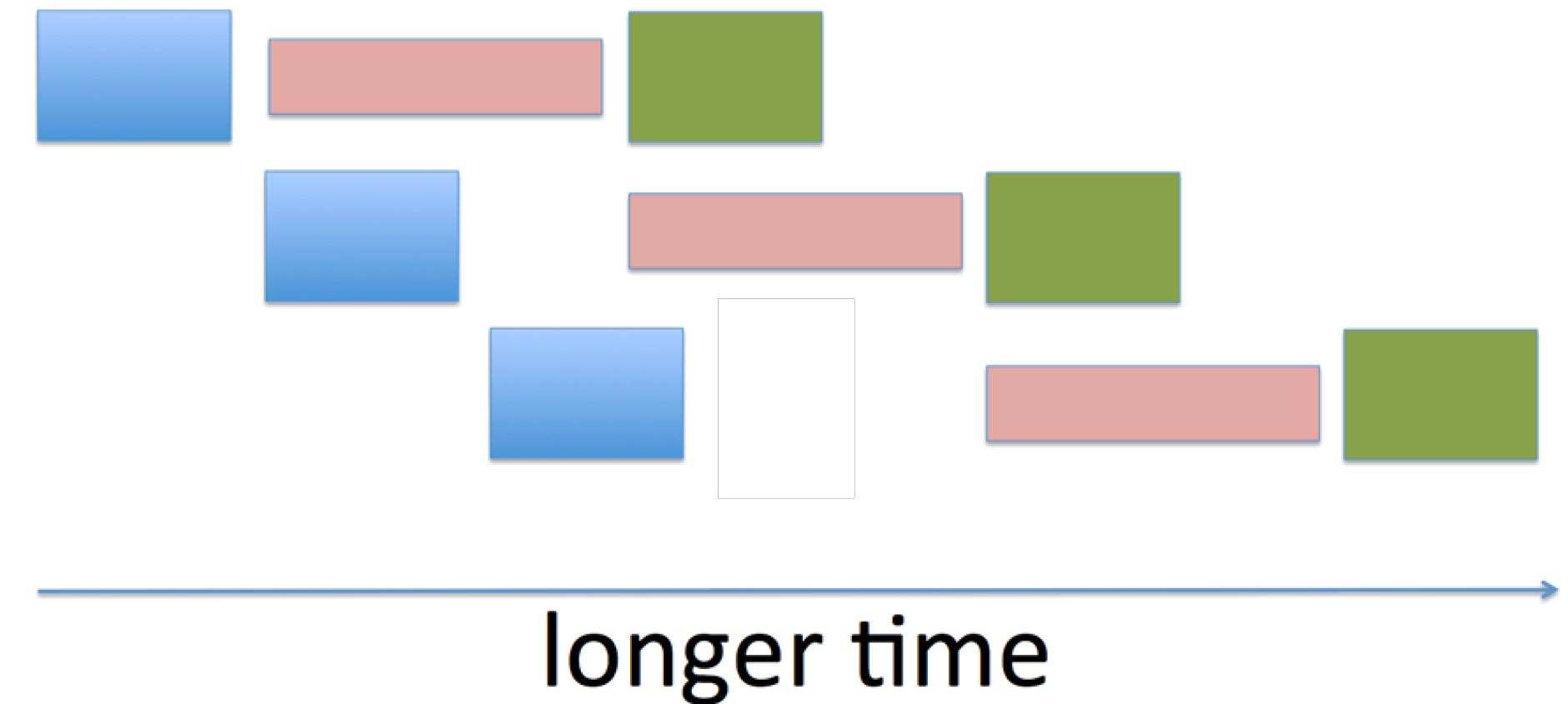
- Deadlocks possible
 - Occurs when all started (but non-terminated) actors have empty mailboxes
- Data races possible when messages include shared objects
- Simulating synchronous replies requires some effort
 - e.g., does not support `addAndGet()`
- Difficult to achieve global consensus
 - Finish not supported as first-class primitive



Pipeline and Actors

Pipelined Parallelism:

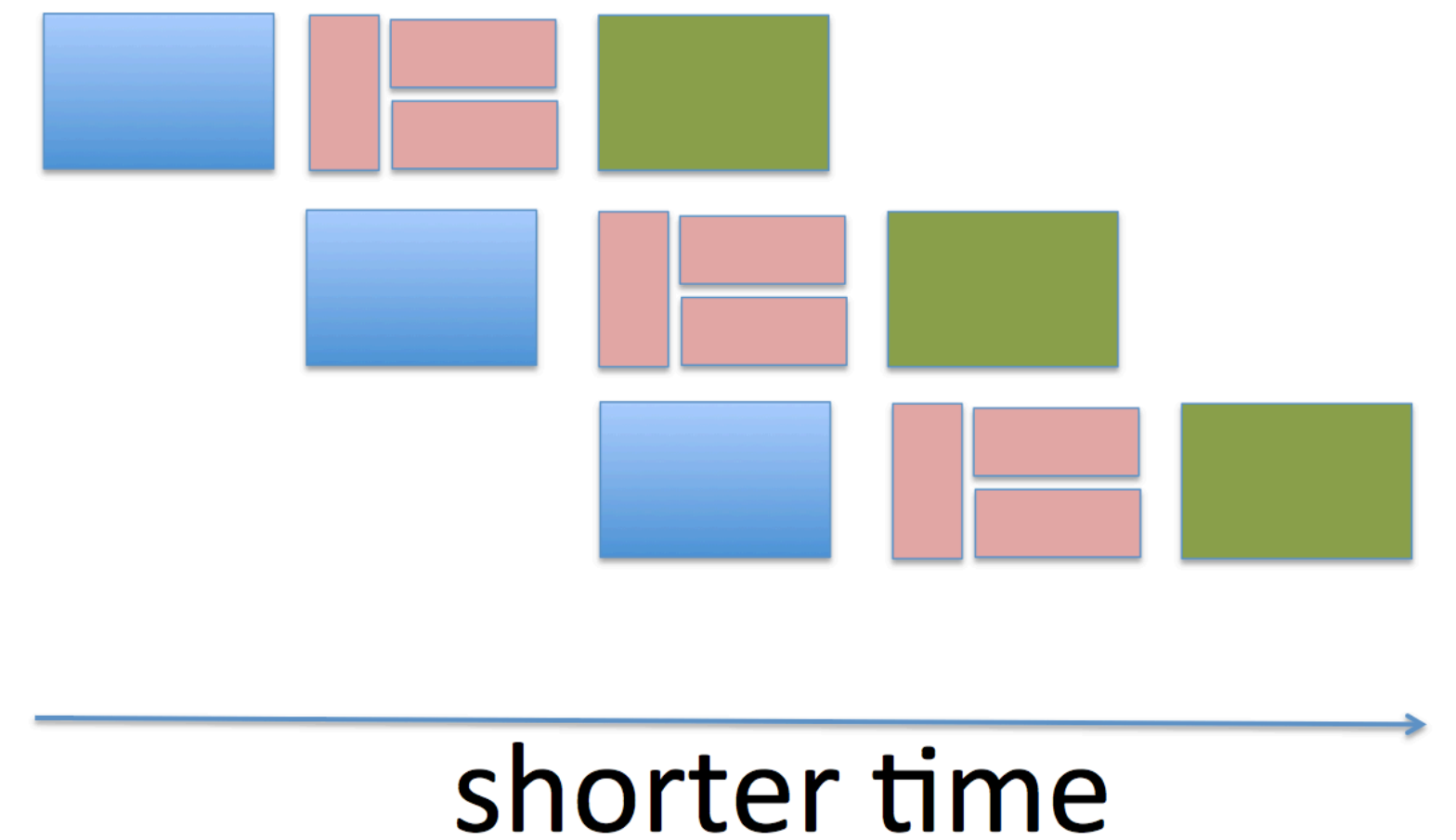
- Each stage can be represented as an actor
- Stages need to ensure ordering of messages while processing them
- Slowest stage is a **throughput bottleneck**



Motivation for Parallelizing Actors

Pipelined Parallelism:

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



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



Example of Parallelizing Actors

```
1. class ArraySumActor 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(() -> { // Two-way parallel sum snippet
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. }
```

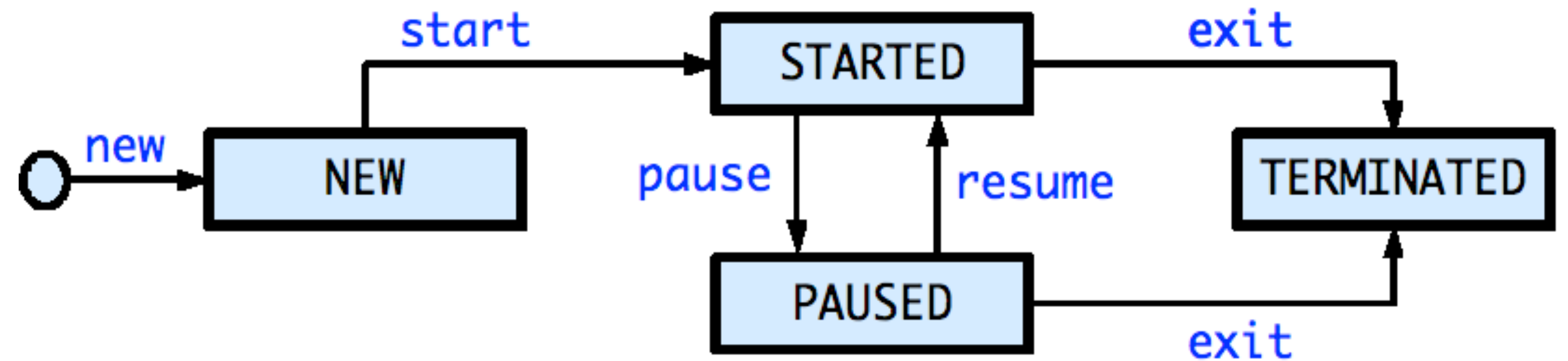


Parallelizing Actors in HJ-Lib

- 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`



State Diagram for Extended Actors with Pause-Resume



- 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

NOTE: Calls to `exit()`, `pause()`, `resume()` only impact the processing of the next message, and not the processing of the current message. These calls should just be viewed as “state change” operations.



Actors: `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.



Actors: 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.



Parallelizing Actors in HJ-Lib

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. }
```



Synchronized Reply using Pause/Resume

Actors don't normally require synchronization with other actors. However, sometimes we might want actors to be in synch with one another.

```
1.class SynchSenderActor
2.     extends Actor<Message> {
3.     private Actor otherActor = ...
4.     void process(Msg msg) {
5.         ...
6.         DDF<T> ddf = newDDF();
7.         otherActor.send(ddf);
8.         println("Response received");
9.         ...
10.} }
```

```
1.class SynchReplyActor
2.     extends Actor<DDF> {
3.     void process(DDF msg) {
4.         ...
5.         println("Message received");
6.         // process message
7.         T responseResult = ...;
8.         ...
9.} }
```



Synchronized Reply using Pause/Resume

Actors don't normally require synchronization with other actors. However, sometimes we might want actors to be in synch with one another.

```
1.class SynchSenderActor
2.     extends Actor<Message> {
3. private Actor otherActor = ...
4. void process(Msg msg) {
5.     ...
6.     DDF<T> ddf = newDDF();
7.     otherActor.send(ddf);
8.     pause(); // non-blocking
9.     asyncAwait(ddf, () -> {
10.         T synchronousReply = ddf.safeGet();
11.         println("Response received");
12.         resume(); // non-blocking
13.     });
14.     ...
15.} }
```

```
1.class SynchReplyActor
2.     extends Actor<DDF> {
3. void process(DDF msg) {
4.     ...
5.     println("Message received");
6.     // process message
7.     T responseResult = ...;
8.     msg.put(responseResult);
9.     ...
10.} }
```

