

---

# COMP 322: Fundamentals of Parallel Programming

## Lecture 23: Actors (contd)

Instructors: Vivek Sarkar, Mack Joyner  
Department of Computer Science, Rice University  
{vsarkar, mjoyner}@rice.edu

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



# Worksheet #22 solution:

## Interaction between `finish` and actors

---

What output will be printed if the end-finish operation from slide 13 is moved from line 13 to line 11 as shown below?

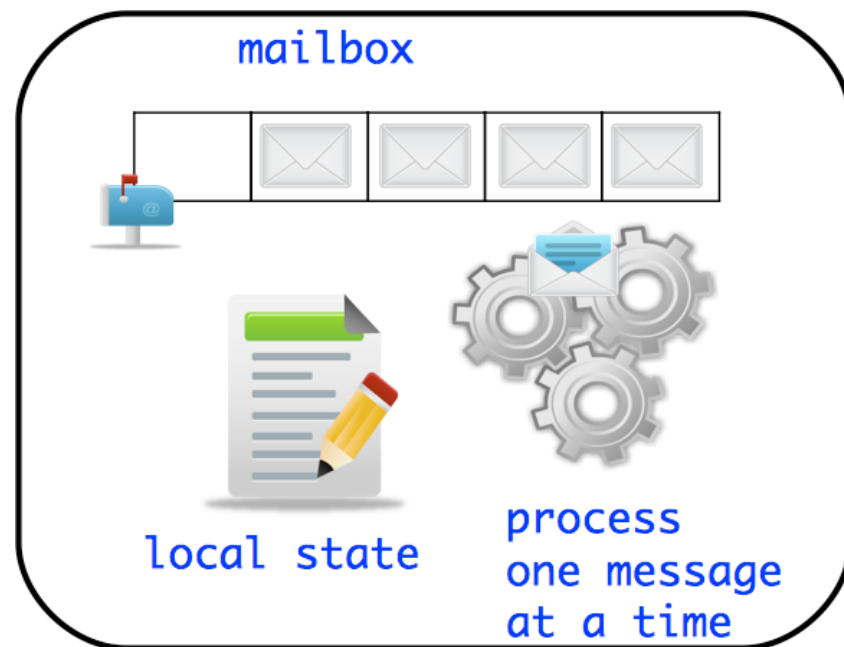
```
1. finish(() -> {
2.     int numThreads = 4;
3.     int numberOfHops = 10;
4.     ThreadRingActor[] ring = new ThreadRingActor[numThreads];
5.     for(int i=numThreads-1;i>=0; i--) {
6.         ring[i] = new ThreadRingActor(i);
7.         ring[i].start(); // like an async
8.         if (i < numThreads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

**Deadlock (no output):** 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()`.

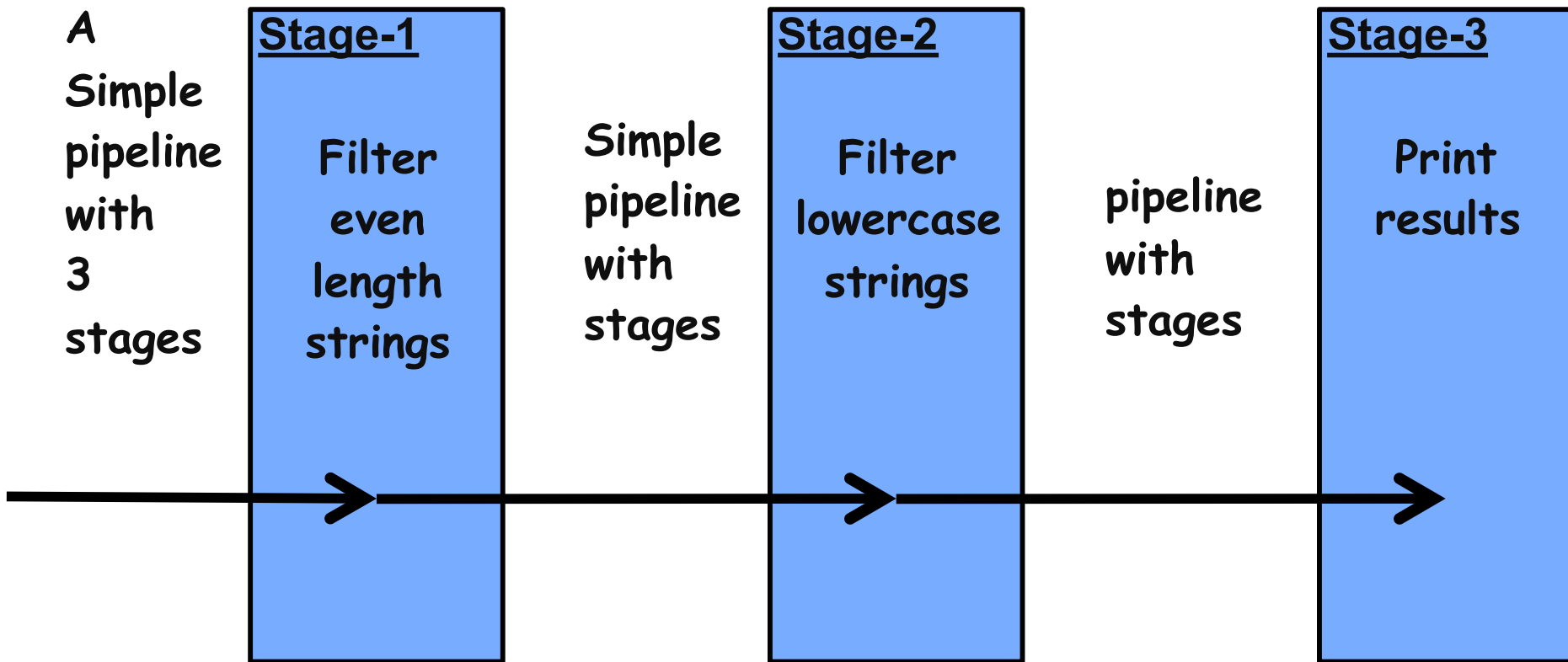


# Recap of Actors

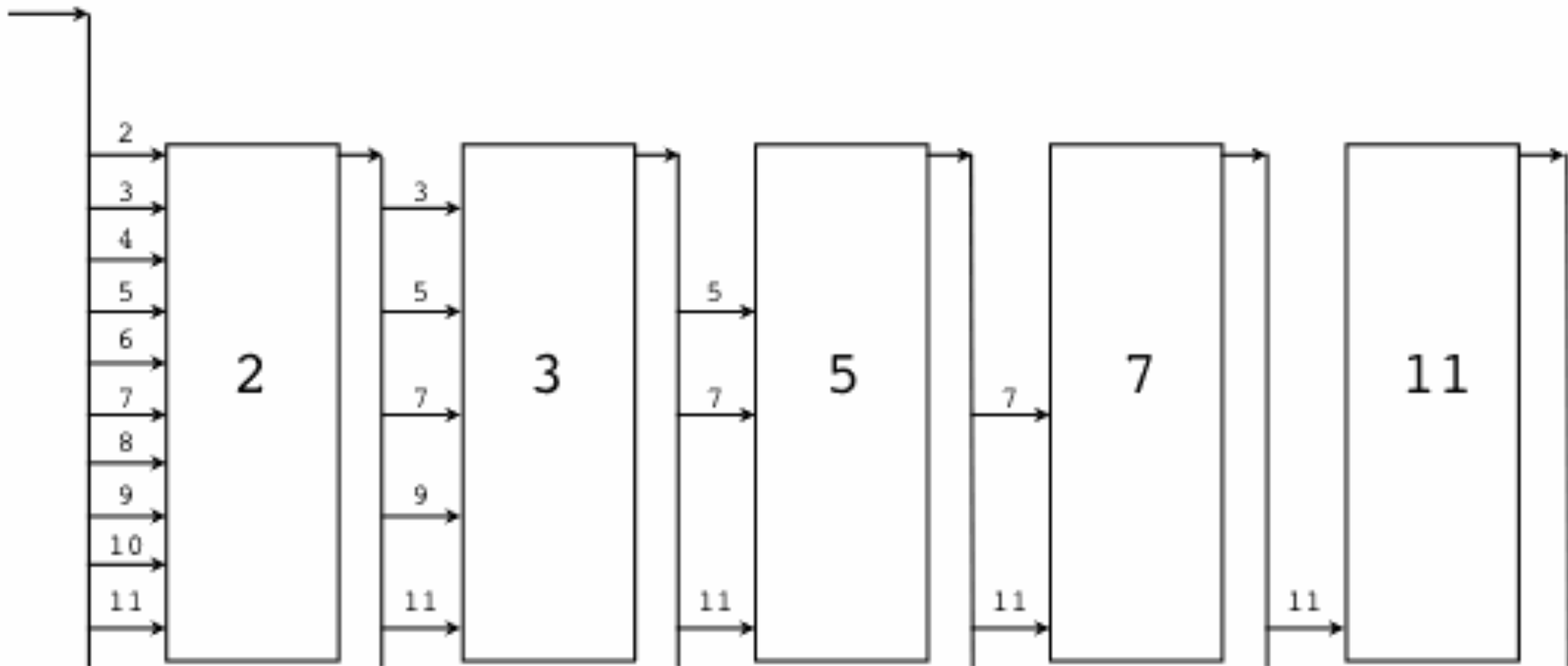
- Rely on asynchronous messaging
- Message are sent to an actor using its `send()` method
- Messages queue up in the mailbox
- Messages are processed by an actor after it is started
- Messages are processed asynchronously
  - one at a time
  - using the body of `process()`



# Simple Pipeline using Actors

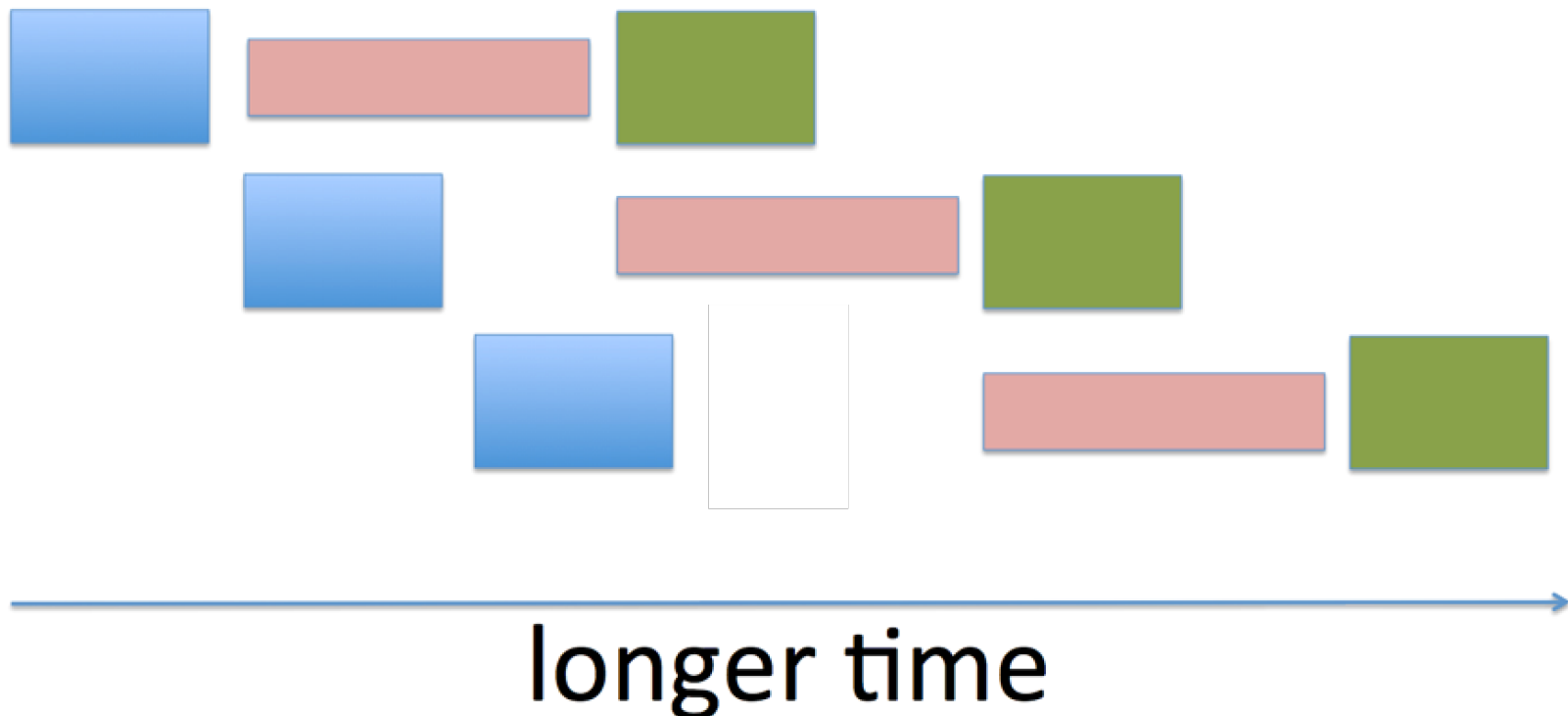


# Sieve of Eratosthenes using Actors



# 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**



# Limitations of Actor Model

---

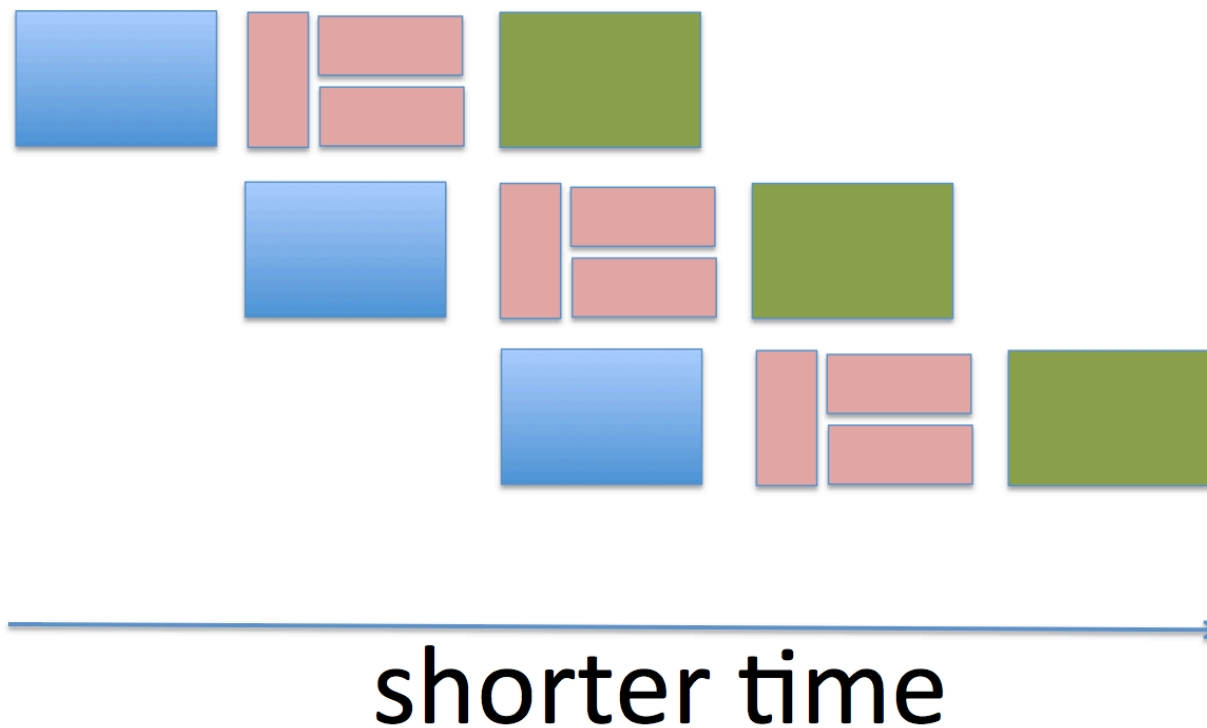
- **Deadlocks possible**
  - **Deadlock occurs when all started (but non-terminated) actors have empty mailboxes**
- **Data races possible when messages include shared objects**
- **Simulating synchronous replies requires some effort**
  - **e.g., does not support synchronous `get()` or `addAndGet()`**
- **Implementing truly concurrent data structures is hard**
  - **No support for parallel reads (as in read-write isolation), or for parallel implementations of 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**



# 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. }
```



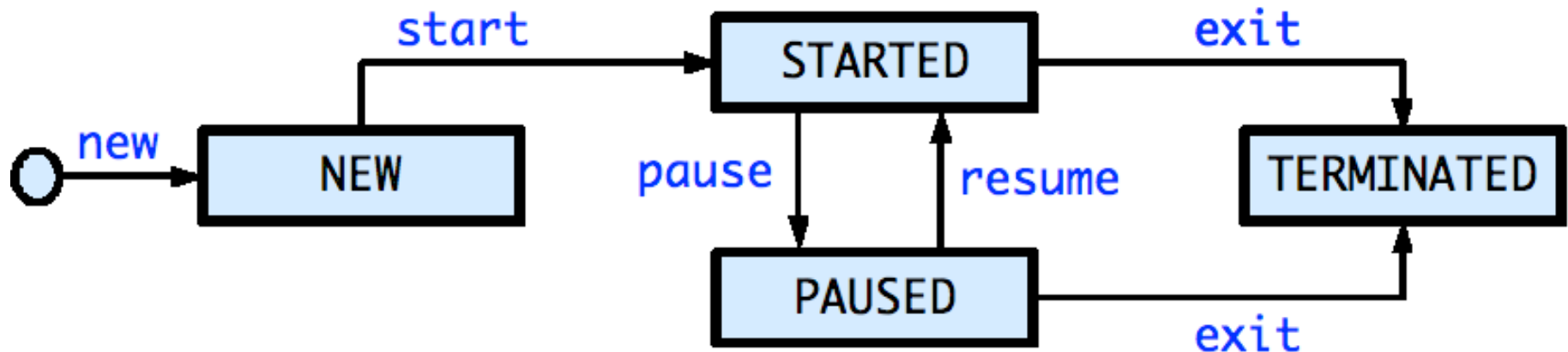
# Parallelizing Actors in HJlib

---

- **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`**



# 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 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.



# Parallelizing Actors in HJlib

---

- 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. }
```



# Synchronous Reply using Pause/Resume

- Actors are asynchronous, sync. replies require blocking operations
- We need notifications from recipient actor on when to resume
- Resumption needs to be triggered on sender actor
- Use DDFs and `asyncAwait`

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

```
1. class SynchronousReplyActor
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.    } }
```



# Actors in the Real World

---

- **Erlang - uses actors for high availability**
  - Facebook chat service backend
  - Whatsapp messaging servers
  - Ericsson, Motorola, T-Mobile - call processing/SMS
  - RabbitMQ - high-performance enterprise messaging
- **Akka - distributed Actor library in Scala**
  - TwoSigma - customized realtime Dashboards on huge datasets
  - ResearchGate - distributed event/data propagation system
  - NBC - election reporting and analysis system
  - eBay - scalable web server monitoring and management





# Announcements

---

- **Reminder: Checkpoint #2 for Homework 3 is due by 11:59pm tonight, and the entire written + programming homework (Checkpoint #3) is due by March 22nd**
- **Reminder: Quiz for Unit 5 is due by this Friday (March 10th)**
- **The registrar has announced the schedule for the COMP 322 final exam:**
  - 2-MAY-2017**
  - 9:00AM - 12:00PM**
  - Location TBD**
- **Scope of final exam (Exam 2) will be limited to Lectures 19 - 38**

