COMP 322: Fundamentals of Parallel Programming

Lecture 22: Actors (contd)

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

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

Worksheet #22 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();
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 created 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.  } } } }
```


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

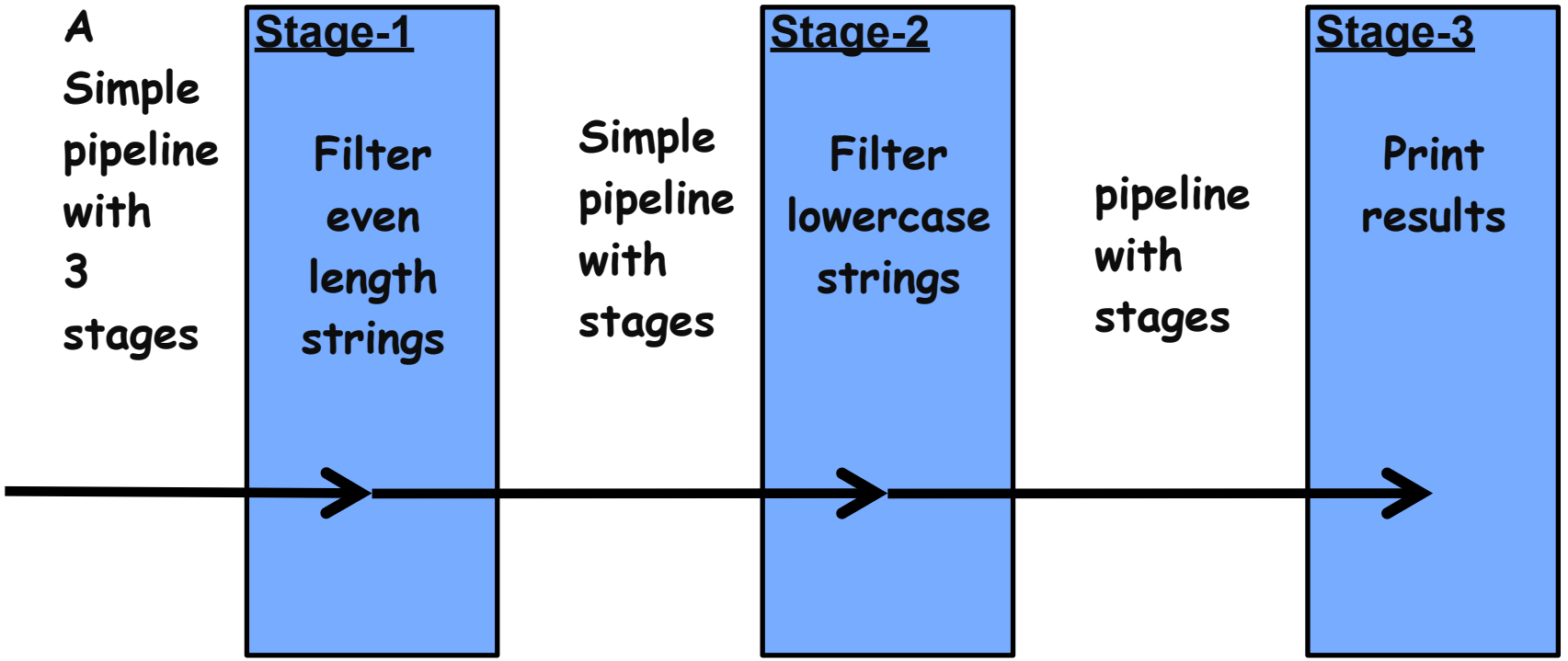
- **void start()** // Cause the actor to start processing messages
- **void onPreStart()** // Convenience: specify code to be executed before actor is started
- **void onPostStart()** // Convenience: specify code to be executed after actor is started

- **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



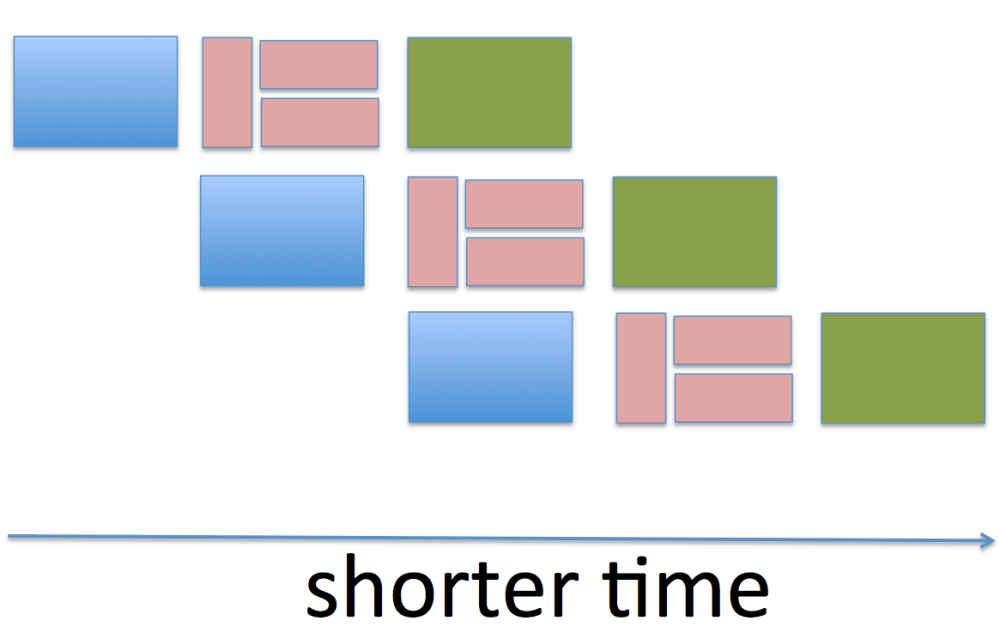
Parallelizing Actors

- 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.   }
```


Parallelizing Actors Example

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



Parallelizing Actors Example

```
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 HJ

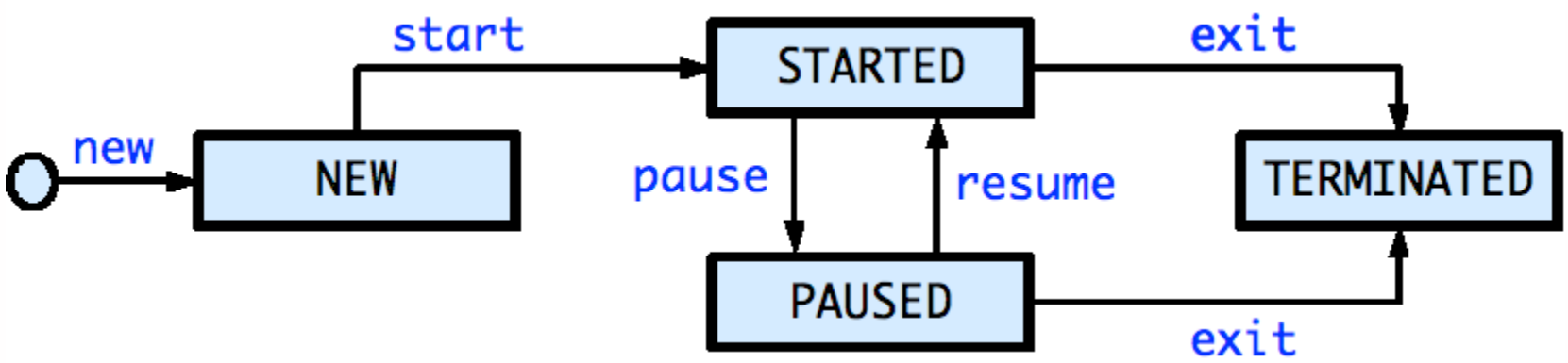
- Two techniques:
  - Use finish construct to wrap asyncs in message processing body
    - Finish ensures all spawned asyncs complete before next message returning from process()
  - Allow escaping asyncs inside process() method
    - WAIT! Won't escaping asyncs violate the one-message-at-a-time rule 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();
4.       async(() -> { S1; }); // escaping async
5.       async(() -> { S2; }); // escaping async
6.       async(() -> {
7.         // async that must be executed before next message
8.         // can use async-await if you want S3 to wait for S1 and 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
- Akin to Java's wait/notify operations with locks
- Messages can accumulate in mailbox when actor is in PAUSED state (analogous to NEW state)

Actors: pause and resume (contd)

- 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.

Actors - Simulating synchronous replies

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

```
class CountMessage {
  ... ddf = newDataDrivenFuture();
  int localCount = 0;

  static int getAnIncrement() {
    ... msg = new CountMessage();
    counterActor.send(msg);
    // use ddf to wai for response
    // THREAD-BLOCKING
    finish(() -> {
      asyncAwait(msg.ddf, () -> {});
    });
    // return count from the message
    return msg.localCount;
  }
}
```

```
class CounterActor extends
    Actor<Object> {
  int counter = 0;
  void process(Object m) {
    if (m instanceof CountMessage) {
     CountMessage cm = (CountMessage)
m;
      counter++;
      msg.localCount = counter;
      msg.ddf.put(true);
    }
  }
}
```


Synchronous Reply using Async-Await (without pause/resume)

```
1.  class SynchronousReplyActor1 extends Actor<Message> {
2.    void process(Message msg) {
3.      if (msg instanceof Ping) {
4.        finish(() -> {
5.          HjDataDrivenFuture<T> ddf = newDataDrivenFuture();
6.          otherActor.send(ddf);
7.          finish(() -> {
8.            asyncAwait(ddf, () -> {
9.              T synchronousReply = ddf.get();
10.             // do some processing with synchronous reply
11.           });
12.         });
13.       });
14.     } else if (msg instanceof ...) { ... } } }
```


Synchronous Reply using Pause/Resume

```
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 ...) { ... } } }
```

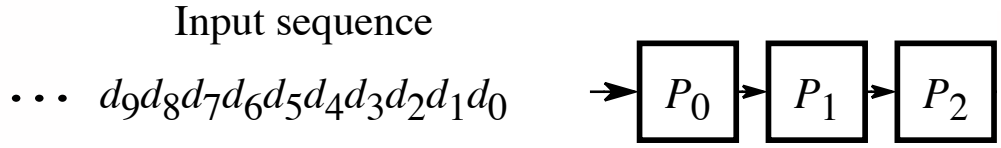

Uses of hybrid actor+task parallelism

- Can use finish to detect actor termination
- Event-driven tasks
- Stateless Actors
  - If an actor has no state, it can process multiple messages in parallelism
- Pipeline Parallelism
  - Actors represent pipeline stages
  - Use tasks to balance pipeline by parallelizing slower stages

Worksheet #23: Ideal Parallelism in Actor Pipeline

Name: _____ Netid: _____

Consider a three-stage pipeline of actors set up so that P0.nextStage = P1, P1.nextStage = P2, and P2.nextStage = null. The process() method for each actor is shown below. Assume that 100 non-null messages are sent to actor P0 after all three actors are started, followed by a null message. What will the total WORK and CPL be for this execution? Recall that each actor has a sequential thread.




```
1.          protected void process(final Object msg) {
2.              if (msg == null) {
3.                  exit();
4.              } else {
5.                  doWork(1); // unit work
6.              }
7.              if (nextStage != null) {
8.                  nextStage.send(msg);
9.              }
10.         }
```