COMP 322: Fundamentals of Parallel Programming

Lecture 22: Introduction to the Actor Model

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

http://comp322.rice.edu/

Worksheet #21a solution:
bstract Metrics with Object-based Isolated Constructs

Q: Compute the WORK and CPL metrics for this program with a <u>global isolated</u> construct. Indicate if your answer depends on the execution order of isolated constructs.

```
finish(() -> {
        for (int i = 0; i < 5; i++) {
                async(() -> {
                        doWork(2);
                        isolated(() -> { doWork(1); });
                        doWork(2);
                }); // async
        } // for
 }); // 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

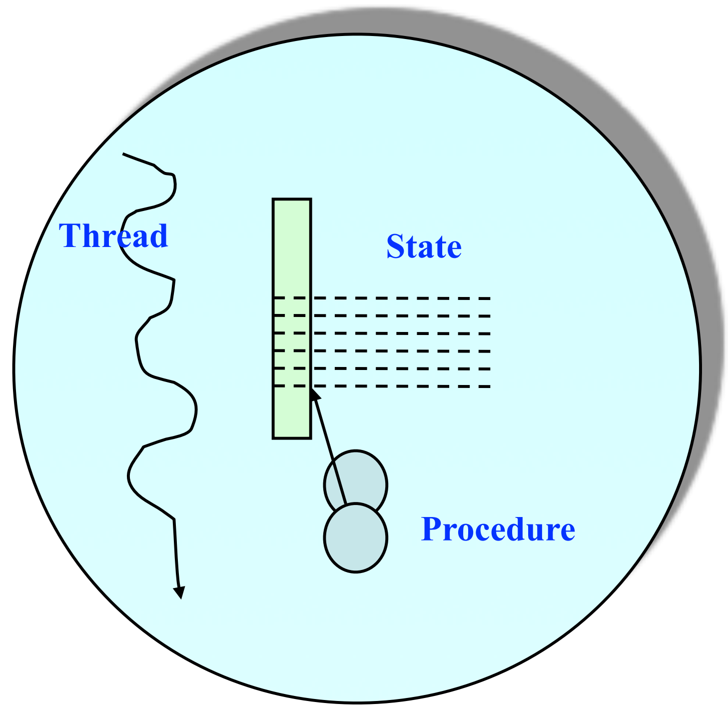
Q: Compute the WORK and CPL metrics for this program with an object-based isolated construct. Indicate if your answer depends on the execution order of isolated constructs.

```
finish(() -> {
     // Assume X is an array of distinct objects
     for (int i = 0; i < 5; i++) {
       async(() -> { // Async task A_i
         doWork(2);
         isolated(X[i], X[i+1],
                  () -> { doWork(1); });
         doWork(2);
       }); // async
     } // for
}); // finish
```

Answer: WORK = 25, worst-case CPL = 7 (e.g., if A_1, A_4 execute in parallel first, then the isolated sections in A_2, A_3 must be serialized thereafter.)

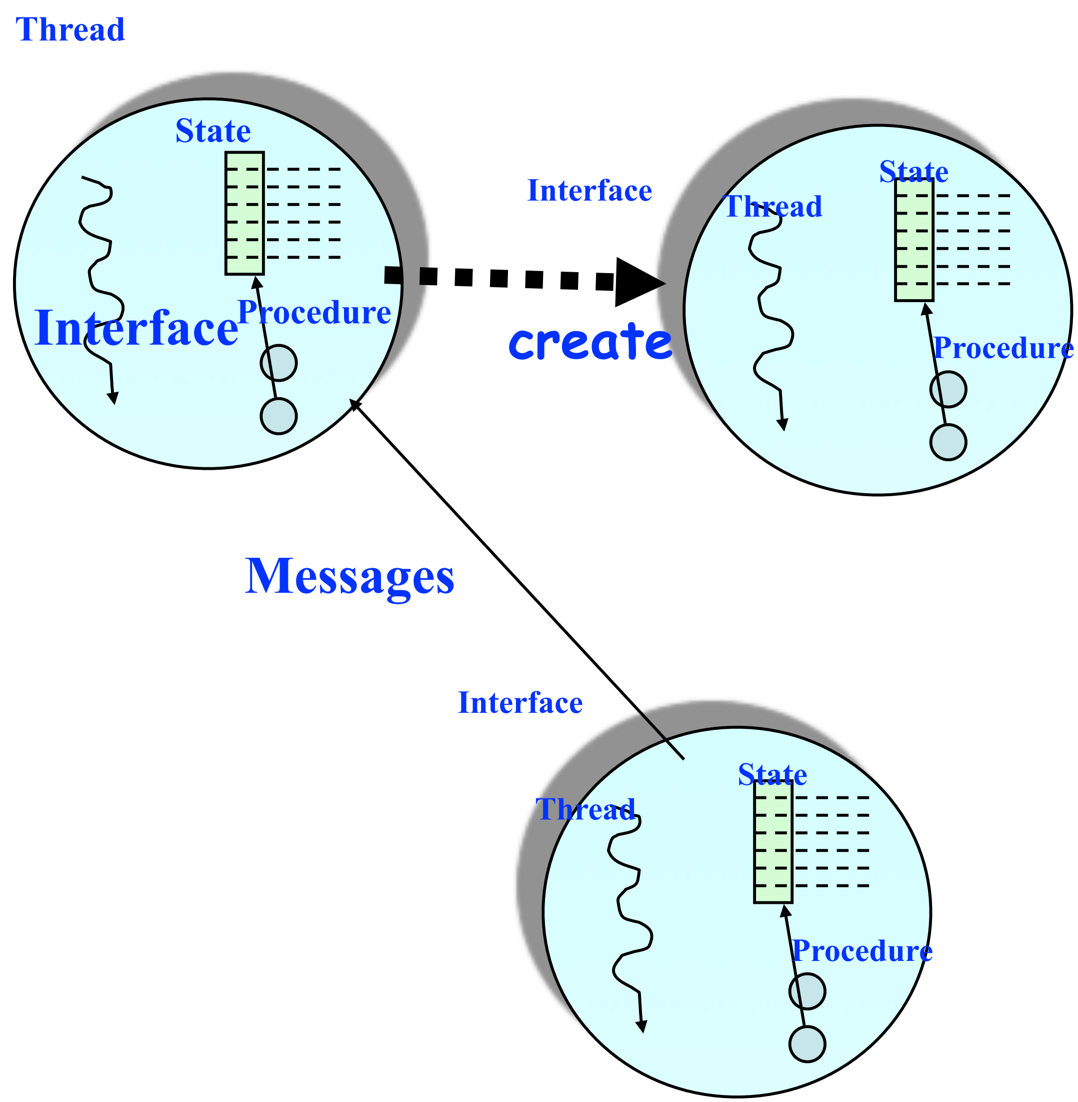
Actors: an alternative approach to isolation

- 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



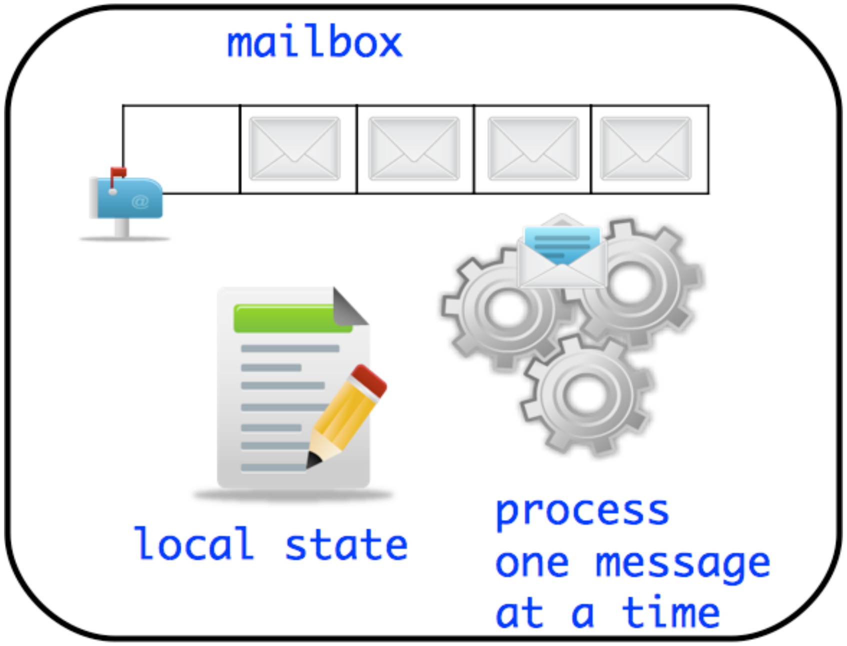
Actor Model

- 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 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 <u>**process**</u>**(**<u>**MessageType**</u> **theMsg)** // Specification of actor's "behavior" when processing messages

void <u>**send**</u>**(**<u>**MessageType**</u> **msg)** // Send a message to the actor

void <u>**start**</u>**()** // Cause the actor to start processing messages

void <u>**onPreStart**</u>**()** // Convenience: specify code to be executed before actor is started

void <u>**onPostStart**</u>**()** // Convenience: specify code to be executed after actor is started

void <u>**exit**</u>**()** // Actor calls exit() to terminate itself

void <u>**onPreExit**</u>**()** // Convenience: specify code to be executed before actor is terminated

void <u>**onPostExit**</u>**()** // Convenience: specify code to be executed after actor is terminated

// In Lecture 23

void <u>**pause**</u>**()** // Pause the actor, i.e. the actors stops processing messages in its mailbox

void <u>**resume**</u>**()** // Resume a paused actor, i.e. actor resumes processing messages in mailbox

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

Hello World Example

```
public class HelloWorld {
  public static void main(final String[] args) {
    finish(()-> {
      EchoActor actor = new EchoActor();
      actor.start(); // don't forget to start the actor
      actor.send("Hello"); // asynchronous send (returns immediately)
      actor.send("World"); // Non-actors can send messages to actors
      actor.send(EchoActor.STOP_MSG);
    });
    println("EchoActor terminated.")
  }
  private static class EchoActor extends Actor<Object> {
    static final Object STOP_MSG = new Object();
    private int messageCount = 0;
    protected void process(final Object msg) {
      if (STOP_MSG.equals(msg)) {
        println("Message-" + messageCount + ": terminating.");
        exit(); // never forget to terminate an actor
      } else {
        messageCount += 1;
        println("Message-" + messageCount + ": " + msg);
} } } }
```

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:

```java
int counter = 0;
public void foo() {
  // do something
  isolated(() -> {
    counter++;
  });
  // do something else
}
public void bar() {
  // do something
  isolated(() -> {
    counter--;
  });
}
```

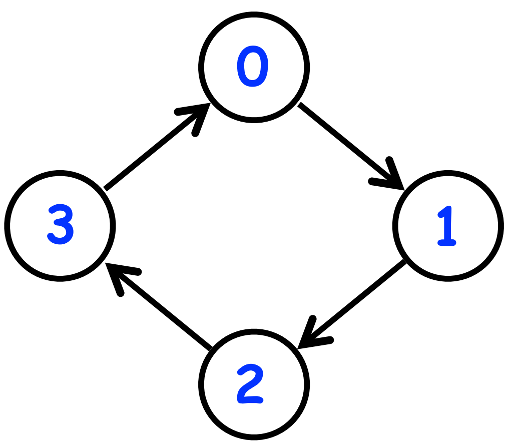
- Can also use atomic variables instead of `isolated` construct

With Actors:

```java
class Counter extends Actor<Message> {
  private int counter = 0; // local state
  protected void process(Message msg) {
    if (msg instanceof IncMessage) {
      counter++;
    } else if (msg instanceof DecMessage){
      counter--;
} } }
  . . .
Counter counter = new Counter();
counter.start();
   public void foo() {
     // do something
     counter.send(new IncrementMessage(1));
     // do something else
   }
   public void bar() {
     // do something
     counter.send(new DecrementMessage(1));
   }
```

ThreadRing (Coordination) Example

```
finish(() -> {
  int threads = 4;
  int numberOfHops = 10;
  ThreadRingActor[] ring =
    new ThreadRingActor[threads];
  for(int i=threads-1;i>=0; i--) {
    ring[i] = new ThreadRingActor(i);
    ring[i].start();
    if (i < threads - 1) {
      ring[i].nextActor(ring[i + 1]);
  } }
  ring[threads-1].nextActor(ring[0]);
  ring[0].send(numberOfHops);
}); // finish
```



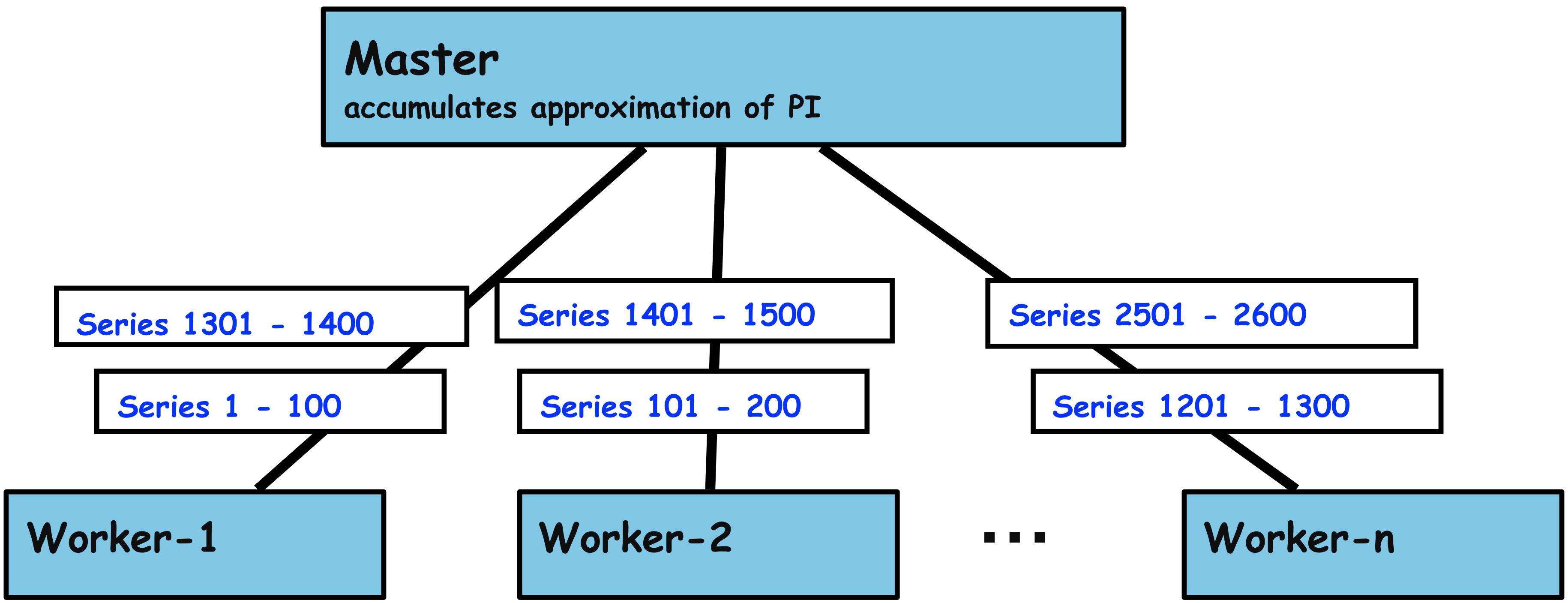

```
class ThreadRingActor
    extends Actor<Integer> {
  private Actor<Integer> nextActor;
  private final int id;
  ...
  public void nextActor(
    Actor<Object> nextActor) {...}

  protected void process(Integer n) {
    if (n > 0) {
      println("Thread-" + id +
        " active, remaining = " + n);
      nextActor.send(n - 1);
    } else {
      println("Exiting Thread-"+ id);
      nextActor.send(-1);
      exit();
} } }
```

Pi Computation Example

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

- **Use Master-Worker technique:**



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

Pi Calculation --- Master Actor

```
class Master extends Actor<Object> {
  private double result = 0; private int nrMsgsReceived = 0;
  private Worker[] workers;
  Master(nrWrkrs, nrEls, nrMsgs) {...} // constructor
  protected void onPostStart() {
    // Create and start workers
    workers = new Worker[nrWrkrs];
    for (int i = 0; i < nrwrkrs; i++) {
      workers[i] = new Worker();
      workers[i].start();
    }
    // Send messages to workers
    for (int j = 0; j < nrMsgs; j++) {
      someWrkr = ... ; // Select worker for message j
      someWrkr.send(new Work(...));
    }
  } // start()
```

Pi Calculation --- Master Actor (contd)

```
  protected void onPostExit() {
    for (int i = 0; i < nrWrkrs; i++)
      workers[i].send(new Stop());
  } // post-exit()
  protected void process(final Object msg) {
    if (msg instanceof Result) {
      result += ((Result) msg).result;
      nrMsgsReceived += 1;
      if (nrMsgsReceived == nrMsgs) exit();
    }
    // Handle other message cases here
  } // process()
} // Master
 . . .
 // Main program
 Master master = new Master(w, e, m);
 finish(() -> { master.start(); });
 println("PI = " + master.getResult());
```

Pi Calculation --- Worker Actor

```
class Worker extends Actor<Object> {
  protected void process(final Object msg) {
    if (msg instanceof Stop)
      exit();
    else if (msg instanceof Work) {
      Work wm = (Work) msg;
      double result = calculatePiFor(wm.start, wm.end)
      master.send(new ResultMessage(result));
  } } // process()

  private double calculatePiFor(int start, int end) {
    double acc = 0.0;
    for (int k = start; k < end; k++) {
      acc += 4.0 * (1 - (k % 2) * 2) / (2 * k + 1);
    }
    return acc;
  }
} // Worker
```

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

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? (Hint: see rule re. finish and actor start operations at bottom of slide 9.)

```
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.
```

BACKUP SLIDES START HERE

Limitations of Actor Model (move to Lec 23)

- 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 #20 solution: Parallel Spanning Tree Algorithm

1. Insert finish, async, and isolated constructs (pseudocode is fine) to convert the sequential spanning tree algorithm below into a parallel algorithm

   See slide 3, as well as the isolatedWithReturn() API in slide 4 for convenience in implementing the pseudocode.

2. Is it better to use a global isolated or an object-based isolated construct for the parallelization in question 1? If object-based is better, which object(s) should be included in the isolated list?

   Object-based isolation should be better with a singleton object list containing the "this" object for the makeParent() method.

Parallel Spanning Tree Algorithm using object-based isolated construct

```
class V  {
  V [] neighbors; // adjacency list for input graph
  V parent; // output value of parent in spanning tree
  boolean makeParent(final V n) {
    return isolatedWithReturn(this, () -> {
      if (parent == null) { parent = n; return true; }
      else return false; // return true if n became parent
    });
  } // makeParent
  void compute() {
    for (int i=0; i<neighbors.length; i++) {
      final V child = neighbors[i];
      if (child.makeParent(this))
        async(() -> { child.compute(); });
     }
  } // compute
} // class V
. . .
root.parent = root; // Use self-cycle to identify root
finish(() -> { root.compute(); });
. . .
```

HJ isolatedWithReturn construct

```
// <body> must contain return statement
isolatedWithReturn (obj1, obj2, …, () -> <body> );
```

Motivation: isolated() construct cannot modify local variables due to restrictions imposed by Java 8 lambdas

- Workaround 1: use isolated() and modify objects rather than local variables
  - Pro: code can be easier to understand than modifying local variables
  - Con: source of errors if multiple tasks read/write same object
- Workaround 2: use isolatedWithReturn()
  - Pro: cleaner than modifying local variables
  - Con: can only return one value

java.util.concurrent.AtomicInteger methods and their equivalent object-based isolated constructs (Lecture 20)

j.u.c.atomic Class and Constructors	j.u.c.atomic Methods	Equivalent HJ isolated statements
AtomicInteger	int j = v.**get**();	int j; `isolated` (v) j = v.val;
	v.**set**(newVal);	`isolated` (v) v.val = newVal;
AtomicInteger()	int j = v.**getAndSet**(newVal);	int j; `isolated` (v) { j = v.val; v.val = newVal; }
// init = 0	int j = v.**addAndGet**(delta);	`isolated` (v) { v.val += delta; j = v.val; }
	int j = v.**getAndAdd**(delta);	`isolated` (v) { j = v.val; v.val += delta; }
AtomicInteger(init)	boolean b = v.**compareAndSet**(expect,update);	boolean b; `isolated` (v) if (v.val==expect) {v.val=update; b=true;} else b = false;

Methods in java.util.concurrent.AtomicInteger class and their equivalent HJ isolated statements. Variable v refers to an AtomicInteger object in column 2 and to a standard non-atomic Java object in column 3. val refers to a field of type int.

Atomic Variables represent a special (and more efficient) case of Object-based isolation

```
class V  {
  V [] neighbors; // adjacency list for input graph
  AtomicReference<V> parent; // output value of parent in spanning tree
  boolean makeParent(final V n) {
    // compareAndSet() is a more efficient implementation of
    // object-based isolation
    return parent.compareAndSet(null, n);
  } // makeParent
  void compute() {
    for (int i=0; i<neighbors.length; i++) {
      final V child = neighbors[i];
      if (child.makeParent(this))
        async(() -> { child.compute(); }); // escaping async
     }
  } // compute
} // class V
. . .
root.parent = root; // Use self-cycle to identify root
finish(() -> { root.compute(); });
. . .
```

Motivation for Read-Write Object-based isolation

Sorted List example

```
public boolean contains(Object object) {
    // Observation: multiple calls to contains() should not
    // interfere with each other
    return isolatedWithReturn(this, () -> {
      Entry pred, curr;
      ...
      return (key == curr.key);
   });
}

 public int add(Object object) {
  return isolatedWithReturn(this, () -> {
      Entry pred, curr;
      ...
      if (...) return 1; else return 0;
  });
}
```

Read-Write Object-based isolation in HJ

```
isolated(readMode(obj1),writeMode(obj2), …, () -> <body> );
```

- Programmer specifies list of objects as well as their read-write modes for which isolation is required
- Not specifying a mode is the same as specifying a write mode (default mode = read + write)
- Mutual exclusion is only guaranteed for instances of isolated statements that have a non-empty intersection in their object lists such that one of the accesses is in writeMode
- <u>Sorted List example</u>

```
public boolean contains(Object object) {
   return isolatedWithReturn( readMode(this), () -> {
     Entry pred, curr;
     ...
     return (key == curr.key);
  });
}

 public int add(Object object) {
   return isolatedWithReturn( writeMode(this), () -> {
     Entry pred, curr;
     ...
     if (...) return 1; else return 0;
   });
 }
```

The world according to Module 1 without & with Phasers

- All the non-phaser parallel constructs that we learned focused on task creation and termination
  - —async creates a task
    - – forasync creates a set of tasks specified by an iteration region
  - —finish waits for a set of tasks to terminate
    - – forall (like "finish forasync") creates and waits for a set of tasks specified by an iteration region
  - —future get() waits for a specific task to terminate
  - —asyncAwait() waits for a set of DataDrivenFuture values before starting
- Motivation for phasers
  - —Deterministic directed synchronization within tasks for barriers, point-to-point synchronization, pipelining
  - —Separate from synchronization associated with task creation and termination
  - —next operations are much more efficient than task creation/termination (async/finish), but they *only help reduce overhead if you perform multiple next operations in a task*

Pipeline Parallelism: Another Example of Point-to-point Synchronization (Recap)

- **Medical imaging pipeline with three stages**
  1. **Denoising stage generates a sequence of results, one per image.**
  2. **Registration stage's input is Denoising stage's output.**
  3. **Segmentation stage's input is Registration stage's output.**
- **Even though the processing is sequential for a single image, *pipeline parallelism* can be exploited via point-to-point synchronization between neighboring stages**

Implementation of Medical Imaging Pipeline

```
final List<PhaserPair> phList1 = Arrays.asList(ph0.inMode(PhaserMode.SIG));
final List<PhaserPair> phList2 = Arrays.asList(ph0.inMode(PhaserMode.WAIT), ph1.inMode(PhaserMode.SIG));
final List<PhaserPair> phList3 = Arrays.asList(ph1.inMode(PhaserMode.WAIT));

asyncPhased(phList1, () -> { // DENOISE stage
    for (int i = 0; i < n; i++) {
          doWork(1);
          signal(); // same as ph0.signal(); as only ph0 is registered in this async
    }
});

asyncPhased(phList2, () -> { // REGISTER stage
    for (int i = 0; i < n; i++) {
          ph0.doWait();  // WARNING: Explicit calls to doWait() can lead to deadlock in general
          doWork(1);
          ph1.signal();
    }
});

asyncPhased(phList3, () -> { // SEGMENT stage
    for (int i = 0; i < n; i++) {
          ph1.doWait();
          doWork(1);
    }
});
```

Announcements

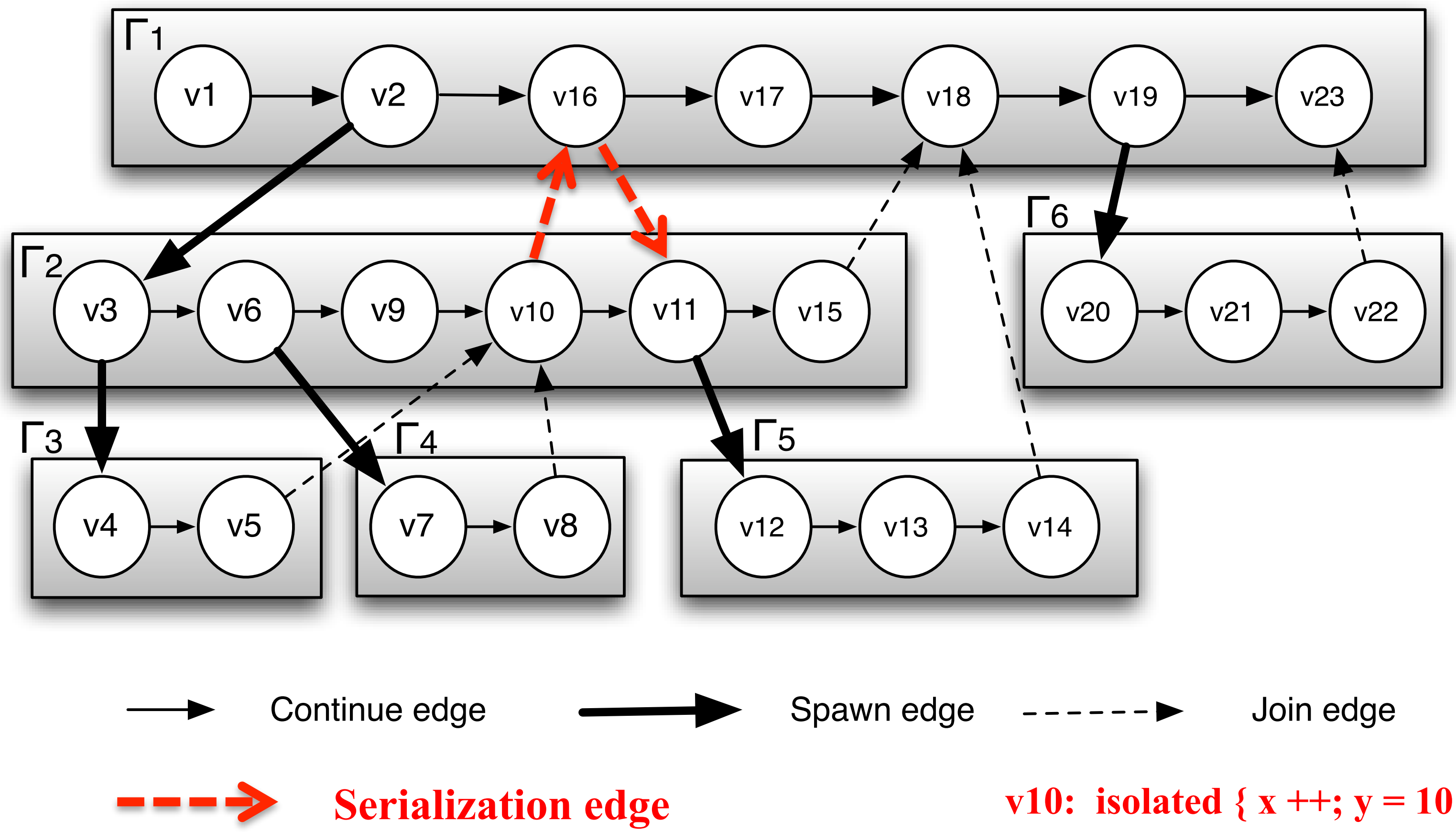
- Reminder: Quiz for Unit 4 is due today
- Reminder: Checkpoint #2 for Homework 3 is due by Wednesday, March 8th, and the entire homework is due by March 22nd
- The registrar has announced the schedule for the COMP 322 final exam:
  - 2-MAY-2017
  - 9:00AM - 12:00PM
  - Location TBD
- Scope of final exam (Exam 2) will be limited to Lectures 19 - 38

Serialized Computation Graph for Isolated Constructs (Recap)

- Model each instance of an isolated construct as a distinct step (node) in the CG.
- Need to reason about the *order* in which interfering isolated constructs are executed
  - Complicated because the order of isolated constructs may vary from execution to execution
- Introduce Serialized Computation Graph (SCG) that includes a specific ordering of all interfering isolated constructs.
  - SCG consists of a CG with additional serialization edges.
  - Each time an isolated step, S', is executed, we add a serialization edge from S to S' for each prior "interfering" isolated step, S
    - Two isolated constructs always interfere with each other
    - Interference of "object-based isolated" constructs depends on intersection of object sets
    - Serialization edge is not needed if S and S' are already ordered in CG
  - An SCG represents a set of schedules in which all interfering isolated constructs execute in the same order.

Example of Serialized Computation Graph with Serialization Edges for v10-v16-v11 order (Recap)

Data race definition can be applied to Serialized Computation Graphs (SCGs) just like regular CGs



v10: isolated { x ++; y = 10; }
v11: isolated { x++; y = 11; }
v16: isolated { x++; y = 16; }

- Need to consider all possible orderings of interfering isolated constructs to establish data race freedom

BACKUP SLIDES START HERE