COMP 322: Fundamentals of Parallel Programming

Lecture 21: Introduction to the Actor Model

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu

Worksheet #20: Sequential->Parallel Spanning Tree Algorithm

Insert finish, async, and atomic (includes a compareAndSet) constructs (pseudocode is fine) to convert the sequential spanning tree algorithm to a parallel algorithm

```
class V  {
  V [] neighbors; // adjacency list for input graph
  V parent; // output value of parent in spanning tree

  boolean makeParent(V n) {
    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))
        child.compute(); // recursive call
     }
  } // compute
} // class V
. . . // main program
root.parent = root; // Use self-cycle to identify root
root.compute();
. . .
```

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(); });
. . .
```

Work-Sharing Pattern using AtomicInteger

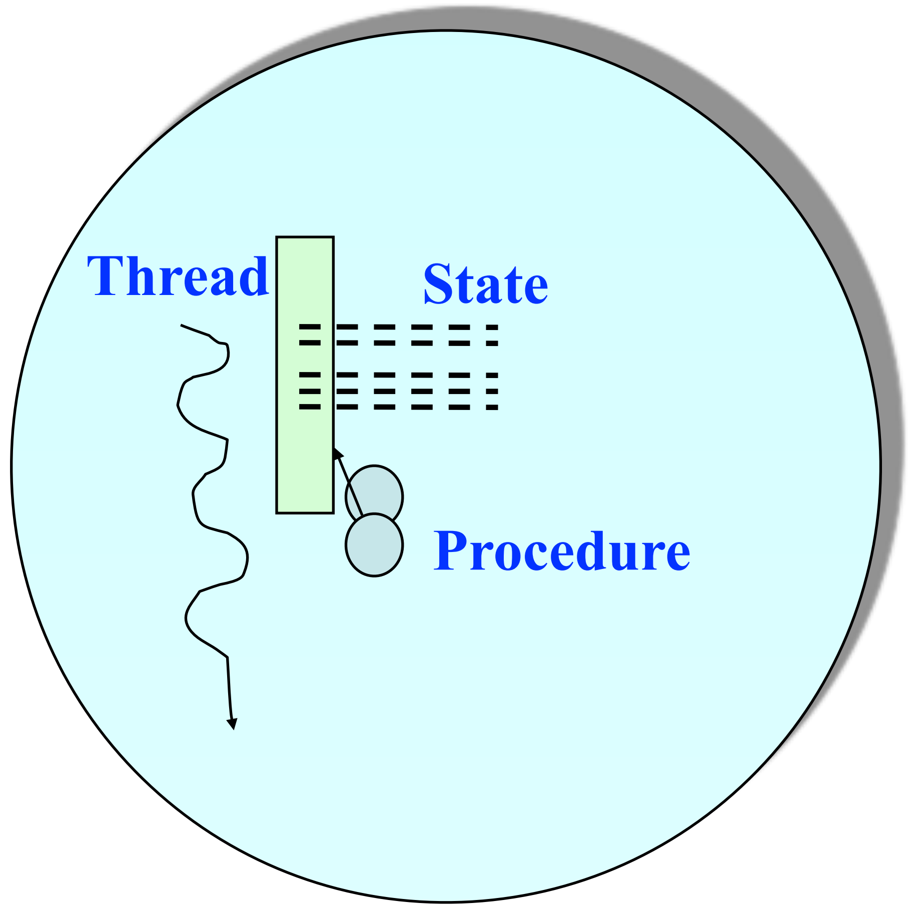
```
import java.util.concurrent.atomic.AtomicInteger;
. . .
String[] X = ... ; int numTasks = …; int j;
int[] taskId = new int[X.length];
AtomicInteger a = new AtomicInteger();
. . .
finish(() -> {
  for (int i=0; i<numTasks; i++ )
    async(() -> {
     do {
        j = a.getAndAdd(1);
        // can also use a.getAndIncrement()
        if (j >= X.length) break;
        taskId[j] = i; // Task i processes string X[j]
        . . .
      } while (true);
    });
}); // finish-for-async
```

Work-Sharing Pattern using AtomicInteger

```
import java.util.concurrent.atomic.AtomicInteger;
. . .
String[] X = ... ; int numTasks = …;
int[] taskId = new int[X.length];
AtomicInteger a = new AtomicInteger();
. . .
finish(() -> {
  for (int i=0; i<numTasks; i++ )
    async(() -> {
     do {
        int j = a.getAndAdd(1);
        // can also use a.getAndIncrement()
        if (j >= X.length) break;
        taskId[j] = i; // Task i processes string X[j]
        . . .
      } while (true);
    });
}); // finish-for-async
```

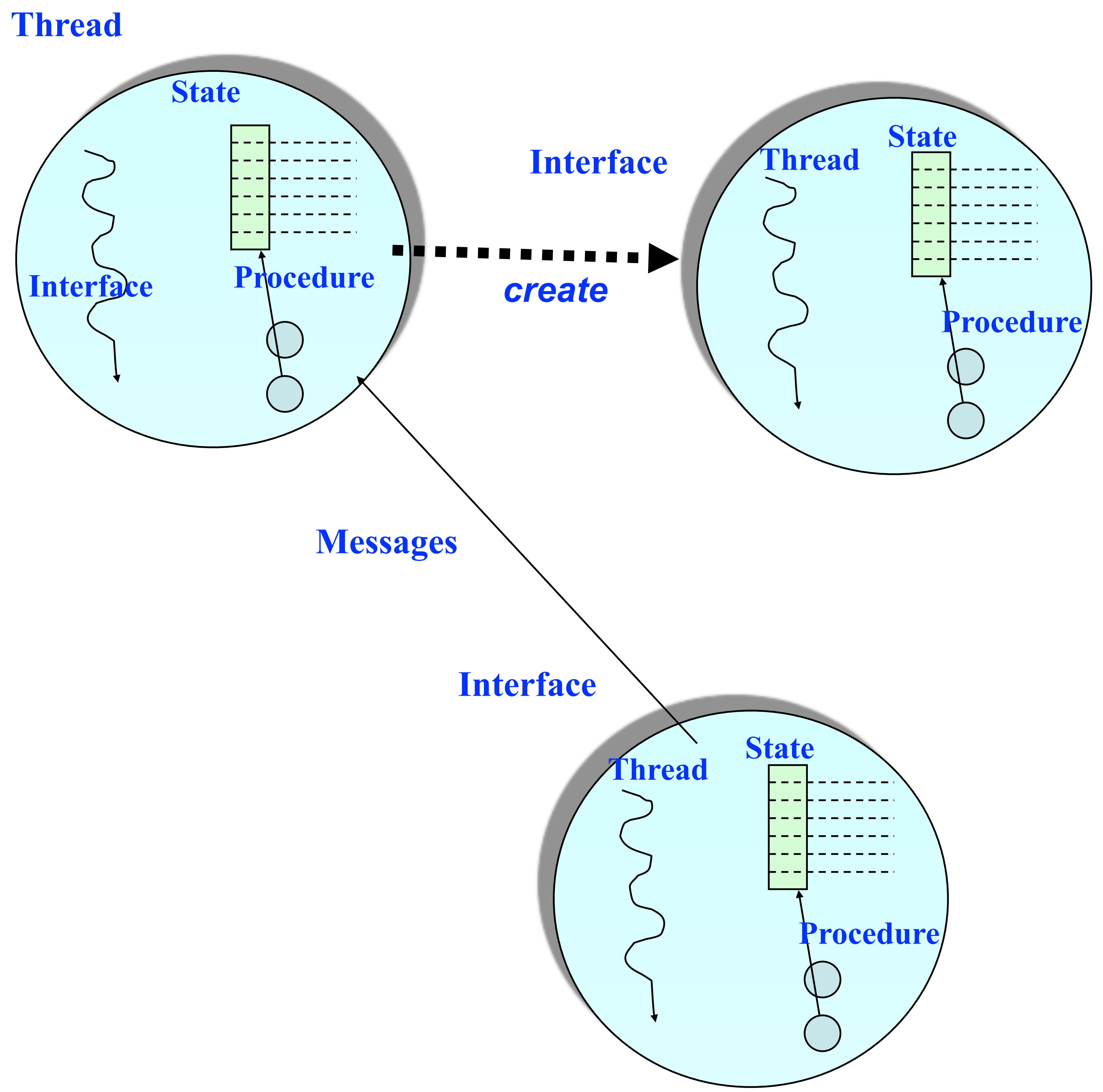
Actors: an alternative approach to isolation, atomics

- 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



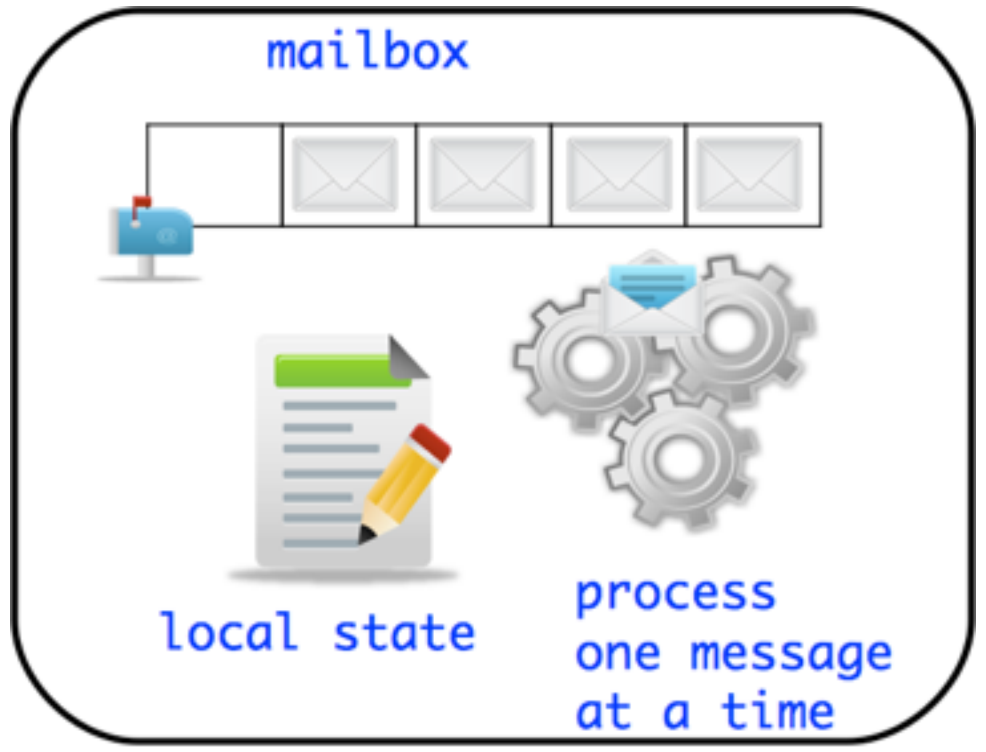
Actor Model

- 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

Using Actors in HJ-Lib

- 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 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

// Next lecture
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

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) <u>preserve the order of messages between the same sender actor/task and the same receiver actor</u>

Integer Counter Example

Without Actors:

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

With Actors:

```
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();
} } }
```

Announcements & Reminders

- Quiz for Unit 4 is due Friday, March 6th at 11:59pm
- Lab 5 is tomorrow (setup before lab, try logging into NOTS)
- Quiz for Unit 5 will be in class on Wednesday, March 11th