

Lab 10: Actors

Instructor: Vivek Sarkar

Course wiki : <https://wiki.rice.edu/confluence/display/PARPROG/COMP322>

Staff Email : comp322-staff@mailman.rice.edu

Importants tips and links

edX site : <https://edge.edx.org/courses/RiceX/COMP322/1T2014R>

Piazza site : <https://piazza.com/rice/spring2015/comp322/home>

Java 8 Download : <https://jdk8.java.net/download.html>

Maven Download : <http://maven.apache.org/download.cgi>

IntelliJ IDEA : <http://www.jetbrains.com/idea/download/>

HJlib Jar File : <http://www.cs.rice.edu/~vs3/hjlib/code/maven-repo/edu/rice/hjlib-cooperative/0.1.5-SNAPSHOT/hjlib-cooperative-0.1.5-SNAPSHOT.jar>

HJlib API Documentation : <https://wiki.rice.edu/confluence/display/PARPROG/API+Documentation>

HelloWorld Project : <https://wiki.rice.edu/confluence/pages/viewpage.action?pageId=14433124>

1 HJlib Actors

HJlib actors were introduced in Lectures 23–24. An actor class is defined by extending the `edu.rice.hj.runtime.actors.Actor` class. Concrete sub-classes of `Actor` are required to implement the `process()` method.

The following code snippet shows the schema for defining an actor class:

```
1 import edu.rice.hj.runtime.actors.Actor;
2 public class EchoActor extends Actor<Object> {
3     protected void process(Object aMessage) {
4         ...
5     } }
```

Method calls can be invoked on actor objects, and they work just like method calls on any other HJlib objects. However, what distinguishes actors from normal objects is that they can be activated by the `start()` method, after which the HJlib runtime ensures that the actor's `process()` method is called in sequence for each message sent to the actor's mailbox. The actor can terminate itself by calling `exit()` in a `process()` call.

Messages can be sent to actors from actor code or non-actor code by invoking the actor's `send()` method using a call as follows, “`someActor.send(aMessage)`”. A `send()` operation is *non-blocking* and *asynchronous*. The HJlib Actor library preserves the order of messages with the same sender and receiver, but messages from different senders may be interleaved in an arbitrary order.

As mentioned in the lectures, there are three basic states for an actor:

- **new**: when an instance of an actor is created, it is in the new state. In this state, an HJlib actor will receive messages sent to its mailbox but will not process them.
- **started**: in this state, the actor will process all messages in its mailbox, one at a time. It will keep doing so until it decides to terminate. In HJlib, an actor is started by invoking its `start()` method: e.g., `myActor.start()`.
- **terminated**: in this state the actor has decided it will no longer process any more messages. Once terminated, an actor cannot be restarted. An actor requests termination by calling its `exit()` method, which changes the actor's state to *terminated* after the `process()` call containing `exit()` returns. Note that the `exit()` call does not itself result in an immediate termination of the `process()` call; it just ensures that no subsequent `process()` calls will be processed.

All async tasks created internally within an actor are registered on the `finish` scope that contained the actor's `start()` operation. The `finish` scope will block until all actors started within it terminate. This is similar to the `finish` semantics while dealing with `asyncs`.

The following HelloWorld example was discussed in Lecture 23, and is also available in `HelloWorld.java`:

```
1  import edu.rice.hj.runtime.actors.Actor;
2
3  public class HelloWorld {
4      public static void main(final String[] args) {
5          EchoActor actor = new EchoActor();
6          finish(() -> {
7              actor.start(); // actor attaches itself to finish scope
8                          // we are guaranteed ordered sends, i.e. though Hello and ↵
9                          // World will be
10                         // processed asynchronously, they will be processed in that ↵
11                         order
12              actor.send("Hello");
13              actor.send("World");
14              actor.send(EchoActor.STOP_MSG);
15          }); // wait until actor terminates
16          System.out.println("EchoActor has terminated");
17      } }
18
19  class EchoActor extends Actor<Object> {
20      static final Object STOP_MSG = new Object();
21      protected void process(final Object msg) {
22          if (STOP_MSG.equals(msg)) {
23              exit();
24          } else {
25              System.out.println(msg);
26          }
27      }
28  }
```

Other examples that were discussed in Lecture 21 include `Pipeline.java`, and `ThreadRingMain.java`.

1.1 Tips and Pitfalls

- Use an actor-first approach when designing programs that use actors *i.e.*, think about which actors need to be created and how they will communicate with each other. This step will also require you to think about the communication objects used as messages.
- If possible, use immutable objects for messages, since doing so avoids data races and simplifies debugging of parallel programs.

- When overriding the `start()` or `exit()` methods in actor classes, remember to make the appropriate calls to the parent's implementation with `super.start()` or `super.exit()`, respectively,
- The HJlib actor `start()` method is not idempotent. Take care to ensure you do not invoke `start()` on the same actor instance more than once. The `exit()` method on the other hand is idempotent, invoking `exit()` multiple times is safe within the same call to `process()`.
- *Always remember to terminate a started actor* using the `exit()` method. If an actor that has been started is not terminated, the enclosing `finish` will wait forever (deadlock).

2 Exercises for today

2.1 Pi Computation using Bailey-Borwein-Plouffe formula

Our first exercise involves computing π to a specified precision in HJlib. The following formula can be used to compute π :

$$\pi = \sum_{n=0}^{\infty} \left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right) \left(\frac{1}{16} \right)^n$$

The `PiSerial1.java` file contains a simple sequential algorithm for computing π using Java's `BigDecimal` data type, that runs for a fixed number of iterations. The `PiActor1.java` file contains a parallel version of `PiSerial1.java` using Master-Worker style actors, as explained in Lecture 23.

In contrast, the `PiSerial2.java` file contains a more realistic sequential algorithm that uses a `while` loop to compute more and more terms of the series until a desired precision is reached.

We have already provided a version of `PiActor2.java` with `TODO` comments. For this section, your assignment is to convert the sequential program in `PiSerial2.java` (for computing π to a desired precision) to an actor-based parallel program in `PiActor2.java` by filling in code at the `TODO` segments. Next, you will need to evaluate the performance of the serial and parallel versions, `PiSerial2.java` and `PiActor2.java`, on a STIC compute node. Also, attempt to run with higher precision values while evaluating the performance. Our implementation achieved over $5.7\times$ speedup over the sequential implementation (`edu.rice.comp322.SieveSerial`) on STIC while using 8 worker threads.

2.2 Primes Sieves using a Pipeline

The `SieveSerial.java` file contains a sequential version of the Sieve of Eratosthenes algorithm for generating prime numbers. You already studied a similar example in Lab 1. For this section, your assignment is to convert the sequential program in `SieveSerial.java` (for computing the number of primes in a given range) to an actor-based parallel program in `Sieve.java` (by filling in code at the `TODO` segments), and to evaluate the performance of the serial and parallel versions on a STIC compute node. Also, attempt to run with higher limit values while evaluating the performance. Our implementation achieved over $6.5\times$ speedup over the sequential implementation (`edu.rice.comp322.PiSerial2`) on STIC while using 8 worker threads.

The basic idea is to create multiple stages of the pipeline that forward a candidate prime number to the next stage only if the stage determines the candidate is locally prime. When the candidate reaches the end of the pipeline, the pipeline may need to be extended. Thus, this is also an example of a dynamic pipeline where the number of stages is not necessarily known in advance. A simple diagrammatic explanation of how the pipeline would work is shown in Figure 1. Note that to reduce the relative overhead, you will need to increase the amount of work done in each stage by having it store and process multiple prime numbers as a batch.

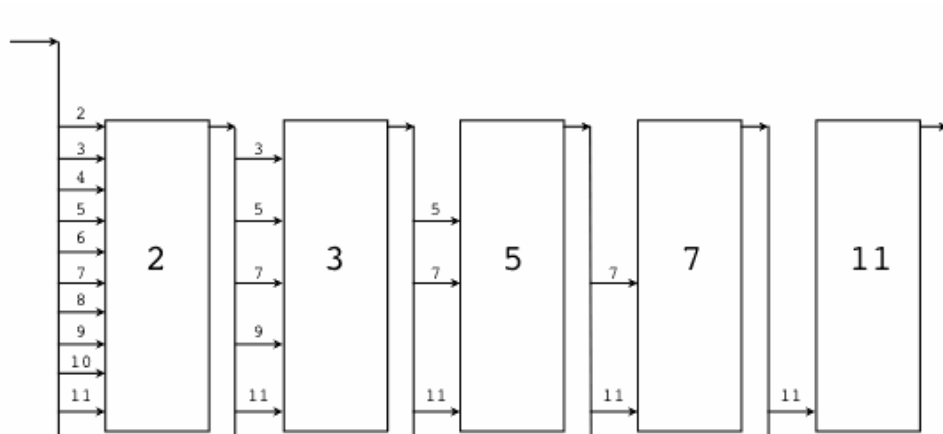


Figure 1: Illustration of Sieve of Eratosthenes algorithm (source: <http://golang.org/doc/sieve.gif>)

3 Turning in your lab work

For each lab, you will need to turn in your work before leaving, as follows.

1. Show your work to an instructor or TA to get credit for this lab. Be prepared to explain the lab at a high level.
2. Check that all the work for today's lab is in the `lab_10` directory. If not, make a copy of any missing files/folders there. It's fine if you include more rather than fewer files — don't worry about cleaning up intermediate/temporary files.
3. Use the `svn` command script to submit the `lab_10` directory to your turnin directory as explained in [the first handout](#). Note that you should *not* turn in a zip file.