Worksheet #27 solution: Liveness Guarantees

```java
/** Atomically adds delta to the current value.
 * @param delta the value to add
 * @return the previous value
 */
public final int getAndAdd(int delta) {
    for (;;) {
        int current = get();
        int next = current + delta;
        if (compareAndSet(current, next))
            return current;
    }
}
```

Assume that multiple tasks call `getAndAdd()` repeatedly in parallel. Can this implementation of `getAndAdd()` lead to a) deadlock, b) livelock, or c) starvation?

Write and explain your answer below.

SOLUTION: c) starvation is possible, but a) deadlock and b) livelock are not possible

NOTE 1: a terminating parallel program execution exhibits none of a), b), or c).
NOTE 2: the original worksheet had option d) unbounded wait, but that's the same as c) starvation.
Actors: an alternative approach to isolation

- An actor is an autonomous, interacting component of a parallel system.

- An actor has:
  - an immutable identity (name, global id)
  - 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 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)

  ```java
  class MyActor extends Actor<T> { 
  protected void process(T message) { 
  println("Processing “ + message); 
  } 
  }
  ```

- Instantiate and start your actor

  ```java
  Actor<Object> anActor = new MyActor();
anActor.start();
  ```

- Send messages to the actor (can be performed by actor or non-actor)

  ```java
  anActor.send(aMessage); //aMessage can be any object in general
  ```

- Use a special message to terminate an actor

  ```java
  protected void process(Object message) {
  if (message.someCondition()) exit();
  }
  ```

- Actor execution implemented as async tasks in HJ

  Can use `finish` to await completion of an actor!
  The actor must be start-ed inside a `finish`.
Hello World Example

```java
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.         println("EchoActor terminated.");
11.     }
12. }
13. private static class EchoActor extends Actor<Object> {
14.     static final Object STOP_MSG = new Object();
15.     private int messageCount = 0;
16.     protected void process(final Object msg) {
17.         if (STOP_MSG.equals(msg)) {
18.             println("Message-" + messageCount + ": terminating.");
19.             exit(); // never forget to terminate an actor
20.         } else {
21.             messageCount += 1;
22.             println("Message-" + messageCount + ": " + msg);
23.         }
24.     }
25. }
```

COMP 322, Spring 2016 (V. Sarkar, S. Imam)

Integer Counter Example

**Without Actors:**

```java
1. int counter = 0;
2. public void foo() {
3.     // do something
4.     isolated(() -> {
5.         counter++;
6.     });
7.     // do something else
8. }
9. public void bar() {
10.    // do something
11.   isolated(() -> {
12.      counter--;
13.   });
14. }
```

**With Actors:**

```java
15. class Counter extends Actor<Message> {
16.     private int counter = 0; // local state
17.     protected void process(Message msg) {
18.         if (msg instanceof IncMessage) {
19.             counter++;
20.         } else if (msg instanceof DecMessage){
21.             counter--;
22.     }
23. }
24. Counter counter = new Counter();
25. counter.start();
26. public void foo() {
27.     // do something
28.     counter.send(new IncrementMessage(1));
29.     // do something else
30. }
31. public void bar() {
32.     // do something
33.     counter.send(new DecrementMessage(1));
34. }
```

- Can also use atomic variables instead of isolated construct
ThreadRing (Coordination) Example

```java
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
```

```java
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:

```
Master
accumulates approximation of PI
```

- Series 1301 - 1400
  - Worker-1
  - Series 1 - 100

- Series 1401 - 1500
  - Worker-2
  - Series 101 - 200

- Series 2501 - 2600
  - Worker-n
  - Series 1201 - 1300

Source: [http://www.enotes.com/topic/Pi](http://www.enotes.com/topic/Pi)
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()

    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();
        }
    } // process()

    // Main program
    Master master = new Master(w, e, m);
    finish() -> { master.start(); }
    println("PI = " + master.getResult());
}
```java
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));
        }
    }

    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;
    }
}
```

**Pi Calculation --- Worker Actor**

### 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 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!)