COMP 322: Fundamentals of Parallel Programming

Lecture 40: Review of Lectures 19-35 (Scope of Exam 2)

Mack Joyner and Zoran Budimlić
{mjoyner, zoran}@rice.edu

http://comp322.rice.edu

Announcements & Reminders

The Final exam (in Canvas) is Saturday, April 30th from 7pm - 10pm

- You may reschedule the exam time if you have a conflict
- Exam is open notes, slides, handouts, canvas videos
- Closed IntelliJ or any other executable platform

HJ isolated construct (start of Module 2, Concurrency)

isolated (() -> <body>);

- Isolated construct identifies a critical section
- Two tasks executing isolated constructs are guaranteed to perform them in mutual exclusion

  ➔Isolation guarantee applies to (isolated, isolated) pairs of constructs, not to (isolated, non-isolated) pairs of constructs
- Isolated constructs may be nested

  — An inner isolated construct is redundant
- Blocking parallel constructs are forbidden inside isolated constructs

  —Isolated constructs must not contain any parallel construct that performs a blocking operation e.g., finish, future get, next

  —Non-blocking async operations are permitted, but isolation guarantee only applies to creation of async, not to its execution
- Isolated constructs can never cause a deadlock

  — Other techniques used to enforce mutual exclusion (e.g., locks — which we will learn later) can lead to a deadlock, if used incorrectly

Object-based isolation

isolated(obj1, obj2, …, () -> <body>)

- In this case, programmer specifies list of objects for which isolation is required
- Mutual exclusion is only guaranteed for instances of isolated constructs that have a common object in their object lists
  - Serialization edges are only added between isolated steps with at least one common object (non-empty intersection of object lists)
  - Standard isolated is equivalent to “isolated(*)” by default i.e., isolation across all objects
- Inner isolated constructs are redundant — they are not allowed to “add” new objects

Parallel Spanning Tree Algorithm using object-based isolated construct

```
1.  class V  {
2.    V [] neighbors; // adjacency list for input graph
3.    V parent; // output value of parent in spanning tree
4.    boolean makeParent(final V n) {
5.      return isolatedWithReturn(this, () -> {
6.        if (parent == null) { parent = n; return true; }
7.        else return false; // return true if n became parent
8.      });
9.    } // makeParent
10.   void compute() {
11.     for (int i=0; i<neighbors.length; i++) {
12.       final V child = neighbors[i];
13.       if (child.makeParent(this))
14.         async(() -> { child.compute(); });
15.     }
16.   } // compute
17. } // class V
18. . . .
19. root.parent = root; // Use self-cycle to identify root
20. finish(() -> { root.compute(); });
21. . . .
```


Worksheet: Abstract Metrics with Object-based Isolated Constructs

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. Since there may be multiple possible computation graphs (based on serialization edges), try and pick the worst-case CPL value across all computation graphs.

Answer: WORK = 25, CPL = 7.

```
1.    finish(() -> {
2.        // Assume X is an array of distinct objects
3.        for (int i = 0; i < 5; i++) {
4.          async(() -> {
5.            doWork(2);
6.            isolated(X[i], X[i+1],
7.                    () -> { doWork(1); });
8.            doWork(2);
9.          }); // async
10.       } // for
11.   }); // finish
```


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
- Sorted List example

```
1.  public boolean contains(Object object) {
2.     return isolatedWithReturn( readMode(this), () -> {
3.       Entry pred, curr;
4.       ...
5.       return (key == curr.key);
6.    });
7.  }
8.
9.   public int add(Object object) {
10.   return isolatedWithReturn( writeMode(this), () -> {
11.     Entry pred, curr;
12.     ...
13.     if (...) return 1; else return 0;
14.   });
15. }
```


java.util.concurrent.AtomicInteger methods and their equivalent isolated constructs (pseudocode)

`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.

Worksheet: Atomic Variables represent a special (and more efficient) case of object-based isolation

```
1.  class V  {
2.    V [] neighbors; // adjacency list for input graph
3.    AtomicReference<V> parent; // output value of parent in spanning tree
4.    boolean makeParent(final V n) {
5.      // compareAndSet() is a more efficient implementation of
6.      // object-based isolation
7.      return parent.compareAndSet(null, n);
8.    } // makeParent
9.    void compute() {
10.     for (int i=0; i<neighbors.length; i++) {
11.       final V child = neighbors[i];
12.       if (child.makeParent(this))
13.         async(() -> { child.compute(); });
14.     }
15.   } // compute
16.} // class V
17.. . .
18.root.parent = root; // Use self-cycle to identify root
19.finish(() -> { root.compute(); });
20.. . .
```


Synchronized statements and methods in Java (Lecture 21)

- Every Java object has an associated lock acquired via:
  - synchronized statements
    - synchronized(foo) { // acquire foo's lock
      // execute code while holding foo's lock
      } // release foo's lock
  - synchronized methods
    - public synchronized void op1() { // acquire 'this' lock
      // execute method while holding 'this' lock
      } // release 'this' lock

- Locking and unlocking are automatic
  - Locks are released when a synchronized block exits
    - By normal means: end of block reached, return, break
    - When an exception is thrown and not caught

Deadlock example with Java synchronized statement

- The code below can deadlock if leftHand() and rightHand() are called concurrently from different threads
  - Because the locks are not acquired in the same order

```
public class ObviousDeadlock {
    . . .
    public void leftHand() {
      synchronized(lock1) {
        synchronized(lock2) {
          for (int i=0; i<10000; i++)
            sum += random.nextInt(100);
        }
      }
    }
}
```

```
    public void rightHand() {
      synchronized(lock2) {
        synchronized(lock1) {
          for (int i=0; i<10000; i++)
            sum += random.nextInt(100);
        }
      }
    }
}
```


Deadlock avoidance in HJ with object-based isolation

- HJ implementation ensures that all locks are acquired in the same order
- ==> no deadlock

```
public class ObviousDeadlock {
    . . .
    public void leftHand() {
      isolated(lock1,lock2) {
        for (int i=0; i<10000; i++)
          sum += random.nextInt(100);
      }
    }
}
```

```
    public void rightHand() {
      isolated(lock2, lock1) {
        for (int i=0; i<10000; i++)
          sum += random.nextInt(100);
      }
    }
}
```


java.util.concurrent.locks.Lock interface

```
1. interface Lock {
2.     // key methods
3.     void lock(); // acquire lock
4.     void unlock(); // release lock
5.     boolean tryLock(); // Either acquire lock (returns true), or return false if lock is not obtained.
6.                                 // A call to tryLock() never blocks!
7.
8.     Condition newCondition();  // associate a new condition
9. }
```

java.util.concurrent.locks.Lock interface is implemented by java.util.concurrent.locks.ReentrantLock class

Worksheet: Use of trylock()

Rewrite the transferFunds() method below to use j.u.c. locks with calls to tryLock instead of synchronized.

Your goal is to write a correct implementation that never deadlocks, unlike the buggy version below (which can deadlock).

Assume that each Account object already contains a reference to a ReentrantLock object dedicated to that object e.g., from.lock() returns the lock for the from object. Sketch your answer using pseudocode.

```
1.  public void transferFunds(Account from, Account to, int amount) {
2.        synchronized (from) {
3.            synchronized (to) {
4.                    from.subtractFromBalance(amount);
5.                    to.addToBalance(amount);
6.            }
7.        }
8.  }
```


Worksheet: Use of trylock()

Rewrite the transferFunds() method below to use j.u.c. locks with calls to tryLock instead of synchronized.

Your goal is to write a correct implementation that never deadlocks, unlike the buggy version below (which can deadlock).

Assume that each Account object already contains a reference to a ReentrantLock object dedicated to that object e.g., from.lock() returns the lock for the from object. Sketch your answer using pseudocode.

```
1.  public void transferFunds(Account from, Account to, int amount) {
2.      while (true) {
3.          // assume that trylock() does not throw an exception
4.          boolean fromFlag = from.lock.trylock();
5.          if (!fromFlag) continue;
6.          boolean toFlag = to.lock.trylock();
7.          if (!toFlag) { from.lock.unlock(); continue; }
8.          try { from.subtractFromBalance(amount);
9.                to.addToBalance(amount); break; }
10.         finally { from.lock.unlock(); to.lock.unlock(); }
11.     } // while
12.  }
```


java.util.concurrent.locks.ReadWriteLock interface

```
interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
  }
```

- Even though the interface appears to just define a pair of locks, the semantics of the pair of locks is coupled as follows
  - —Case 1: a thread has successfully acquired writeLock().lock()
    - No other thread can acquire readLock() or writeLock()
  - —Case 2: no thread has acquired writeLock().lock()
    - Multiple threads can acquire readLock()
    - No other thread can acquire writeLock()
- java.util.concurrent.locks.ReadWriteLock interface is implemented by java.util.concurrent.locks.ReadWriteReentrantLock class

Hashtable Example

```
class Hashtable<K,V> {
  …
  // coarse-grained, one lock for table
  ReentrantReadWriteLock lk = new ReentrantReadWriteLock();
  V lookup(K key) {
    int bucket = hasher(key);
    lk.readLock().lock(); // only blocks writers
    … read array[bucket] …
    lk.readLock().unlock();
  }
  void insert(K key, V val) {
    int bucket = hasher(key);
    lk.writeLock().lock(); // blocks readers and writers
     … write array[bucket] …
    lk.writeLock().unlock();
  }
}
```


Linearizability of Concurrent Objects (Lecture 27)

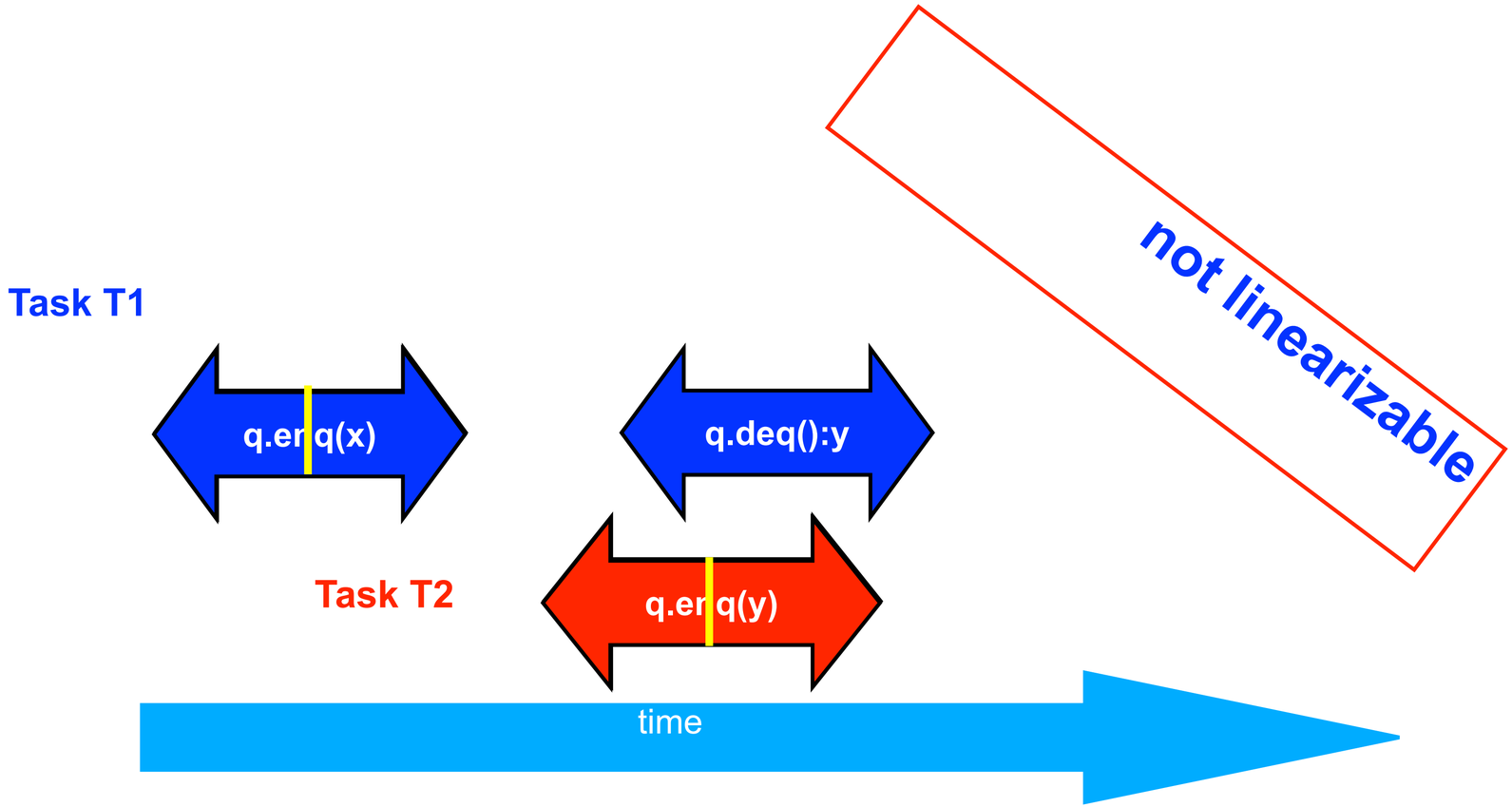
Concurrent object

- A concurrent object is an object that can correctly handle methods invoked in parallel by different tasks or threads
  - —Examples: Concurrent Queue, AtomicInteger

Linearizability

- Assume that each method call takes effect "instantaneously" at some distinct point in time between its invocation and return.
- An execution is linearizable if we can choose instantaneous points that are consistent with a sequential execution in which methods are executed at those points
- An object is linearizable if all its possible executions are linearizable

Example 2: is this execution linearizable?



Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

Worksheet: Execution of concurrent implementation of FIFO queue q

Is this a linearizable execution?

Time	Task A	Task B
0	Invoke `q.enq(x)`	
1	Work on `q.enq(x)`	Invoke `q.enq(y)`
2	Work on `q.enq(x)`	Return from `q.enq(y)`
3	Return from `q.enq(x)`	
4		Invoke `q.deq()`
5		Return `x` from `q.deq()`

Yes! Can be linearized as "q.enq(x) ; q.enq(y) ; q.deq():x"

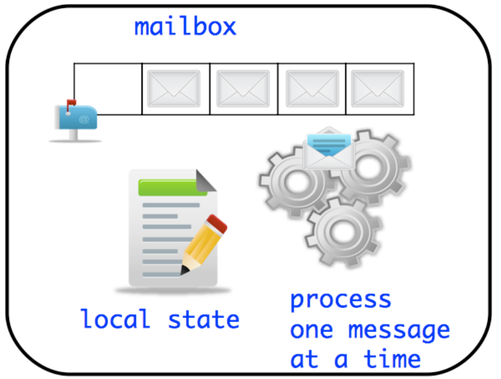
Actor Life Cycle (Lecture 28)

Actor states

- New: Actor has been created
  —e.g., email account has been created

- 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



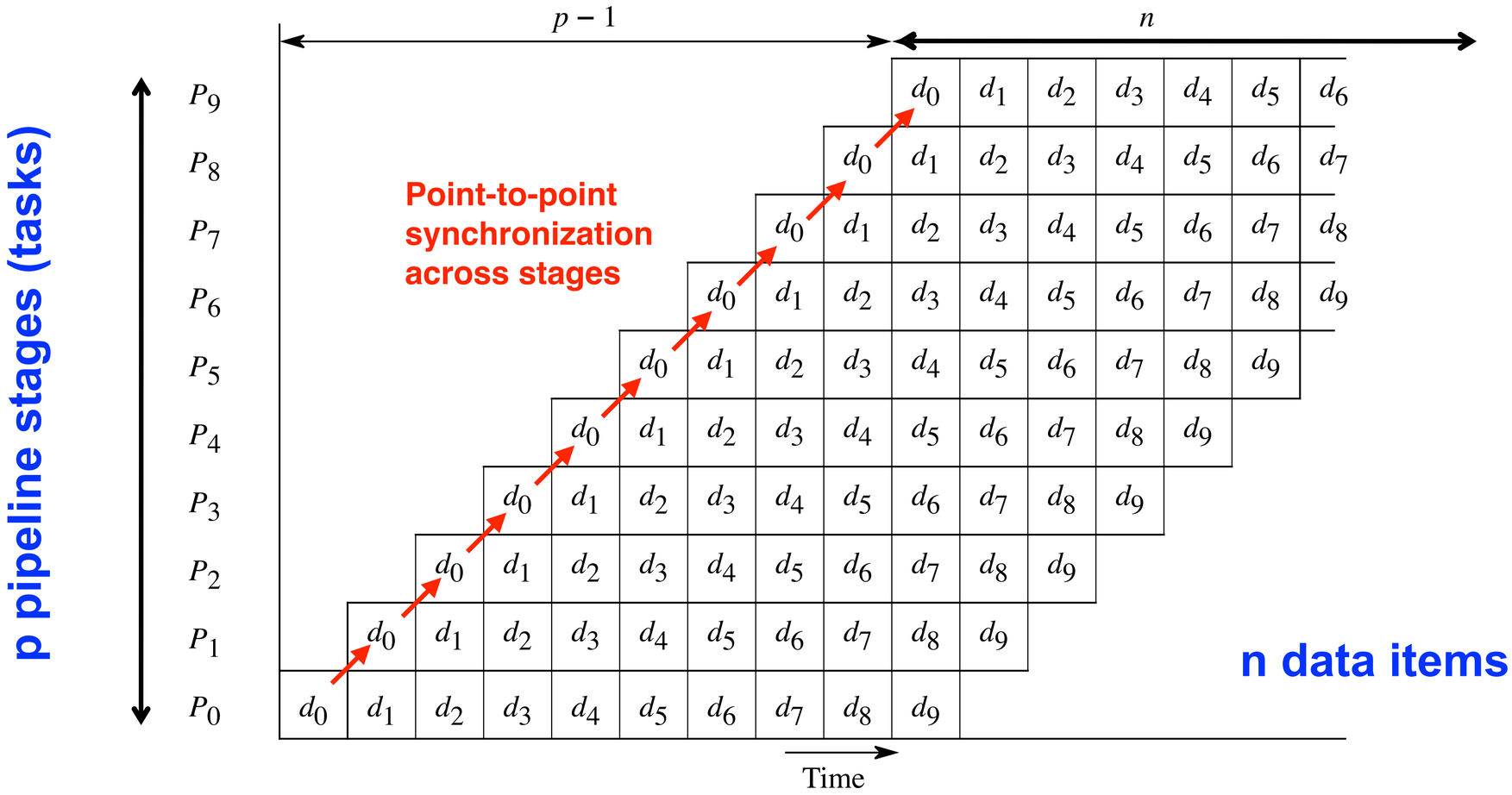
Worksheet: Interaction between finish and actors

What output will be printed if the end-finish operation is moved from line 13 to line 11 as shown below?

```
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(); // like an async
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.
```

Deadlock (no output): the end-finish operation in line 11 waits for all the actors started in line 7 to terminate, but the actors are waiting for the message sequence initiated in line 13 before they call exit().

Timing Diagram for One-Dimensional Pipeline

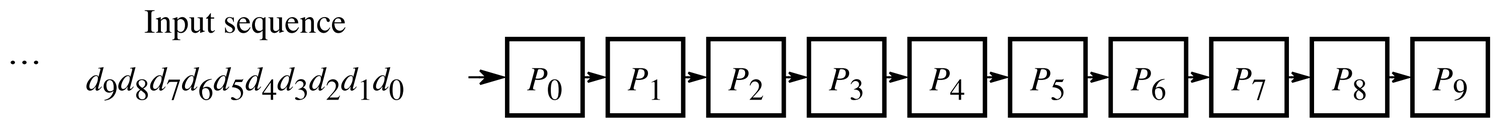


- Horizontal axis shows progress of time from left to right, and vertical axis shows which data item is being processed by which pipeline stage at a given time.

Worksheet: Analyzing Parallelism in an Actor Pipeline

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.

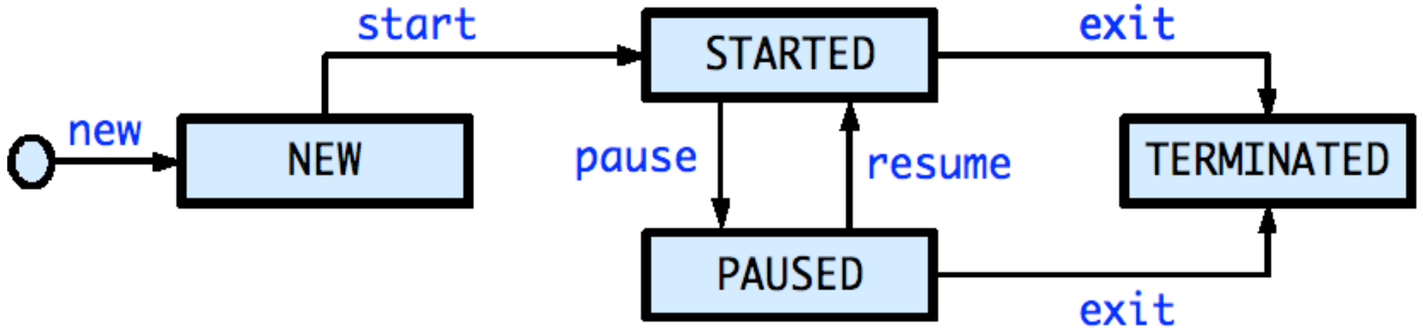
Assume that 100 non-null messages are sent to actor P0 after all three actors are started, followed by a null message. What will the total WORK and CPL be for this execution? Recall that each actor has a sequential thread.




```
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.    }
```

WORK = 300, CPL = 102

State Diagram for Extended Actors with Pause-Resume



- Paused state: actor will not process subsequent messages until it is resumed
- Resume actor when it is safe to process the next message
- Messages can accumulate in mailbox when actor is in PAUSED state

NOTE: Calls to exit(), pause(), resume() only impact the processing of the next message, and not the processing of the current message. These calls should just be viewed as "state change" operations.

Synchronized Reply using Pause/Resume

Actors don't normally require synchronization with other actors. However, sometimes we might want actors to be in synch with one another.

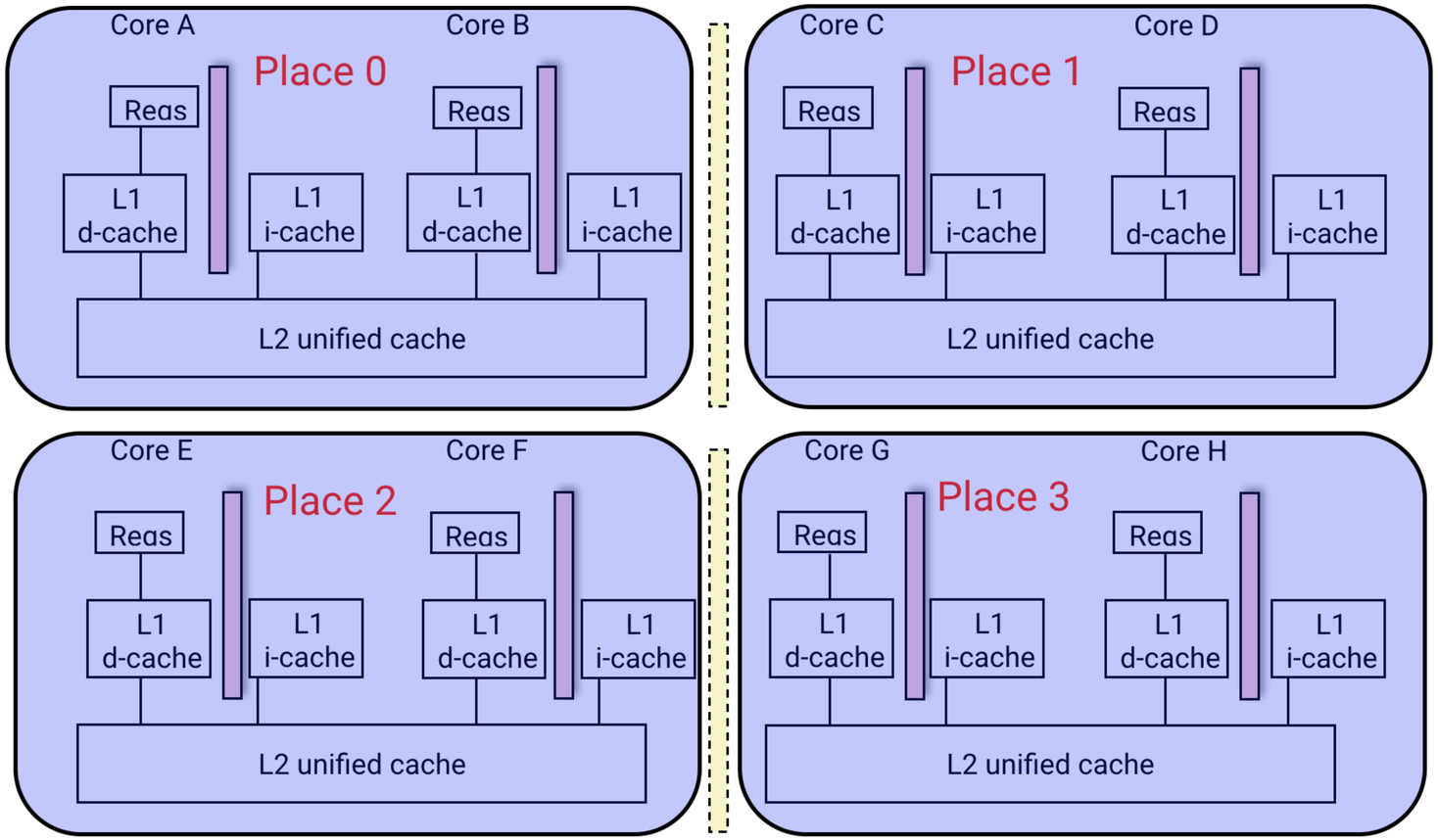
```
1.class SynchSenderActor
2.     extends Actor<Message> {
3.  private Actor otherActor = …
4.  void process(Msg msg) {
5.     ...
6.     DDF<T> ddf = newDDF();
7.     otherActor.send(ddf);
8.     pause(); // non-blocking
9.     asyncAwait(ddf, () -> {
10.      T synchronousReply = ddf.safeGet();
11.       println("Response received");
12.       resume(); // non-blocking
13.    });
14.    ...
15.} }
```

```
1.class SynchReplyActor
2.     extends Actor<DDF> {
3.  void process(DDF msg) {
4.     ...
5.     println("Message received");
6.     // process message
7.     T responseResult = ...;
8.     msg.put(responseResult);
9.     ...
10.} }
```


Co-locating async tasks in "places" (Lecture 30)

```
// Main program starts at place 0
asyncAt(place(0), () -> S1);
asyncAt(place(0), () -> S2);
```

```
asyncAt(place(1), () -> S3);
asyncAt(place(1), () -> S4);
asyncAt(place(1), () -> S5);
```




```
asyncAt(place(2), () -> S6);
asyncAt(place(2), () -> S7);
asyncAt(place(2), () -> S8);
```

```
asyncAt(place(3), () -> S9);
asyncAt(place(3), () -> S10);
```

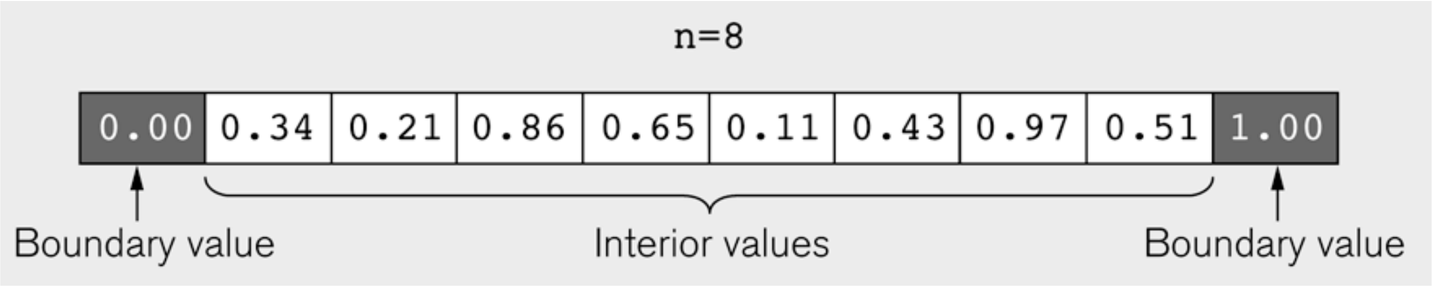

Worksheet: impact of distribution on parallel completion time

```
1.  public void sampleKernel(
2.      int iterations, int numChunks, Distribution dist) {
3.    for (int iter = 0; iter < iterations; iter++) {
4.      finish(() -> {
5.        forseq (0, numChunks - 1, (jj) -> {
6.          asyncAt(dist.get(jj), () -> {
7.            doWork(jj);
8.            // Assume that time to process chunk jj = jj units
9.          });
10.       });
11.     });
12.   } // for iter
13. } // sample kernel
```

- Assume an execution with n places, each place with one worker thread
- Will a block or cyclic distribution for dist have a smaller abstract completion time, assuming that all tasks on the same place are serialized with one worker per place?
- Answer: Cyclic distribution because it leads to better load balance (locality was not a consideration in this problem)

One-Dimensional Iterative Averaging Example

- Initialize a one-dimensional array of (n+2) double's with boundary conditions, myVal[0] = 0 and myVal[n+1] = 1.
- In each iteration, each interior element myVal[i] in 1..n is replaced by the average of its left and right neighbors.
  —Two separate arrays are used in each iteration, one for old values and the other for the new values
- After a sufficient number of iterations, we expect each element of the array to converge to myVal[i] = (myVal[i-1]+myVal[i+1])/2, for all i in 1..n



Ilustration of an intermediate step for n = 8 (source: Figure 6.19 in Lin-Snyder book)

HJ code for One-Dimensional Iterative Averaging

```
1.// Intialize m, n, myVal, newVal
2.m = … ; n = … ;
3.float[] myVal = new float[n+2];
4.float[] myNew = new float[n+2];
5.forseq(0, m-1, (iter) -> {
6.  // Compute MyNew as function of input array MyVal
7.    forall(1, n, (j) -> { // Create n tasks
8.        myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
9.    }); // forall
10.  // What is the purpose of line 11 below?
11.  float[] temp=myVal; myVal=myNew; myNew=temp;
12.}); // forseq
```


HJ code for One-Dimensional Iterative Averaging

```
1.// Intialize m, n, myVal, newVal
2.m = … ; n = … ; nc = … ;
3.float[] myVal = new float[n+2];
4.float[] myNew = new float[n+2];
5.forseq(0, m-1, (iter) -> {
6.  // Compute MyNew as function of input array MyVal
7.    forallChunked(1, n, n/nc, (j) -> { // Create nc tasks
8.        myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
9.    }); // forall
10.  // What is the purpose of line 11 below?
11.  float[] temp=myVal; myVal=myNew; myNew=temp;
12.}); // forseq
```


Worksheet: One-dimensional Iterative Averaging Example

1) Assuming n=9 and the input array below, perform a "half-iteration" of the iterative averaging example by only filling in the blanks for odd values of j in the myNew[] array (different from the real algorithm). Recall that the computation is "myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;"

index, j	0	1	2	3	4	5	6	7	8	9	10
myVal	0	0	0.2	0	0.4	0	0.6	0	0.8	0	1
myNew	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1

2) Will the contents of myVal[] and myNew[] change in further iterations?

No, this represents the converged value (equilibrium/fixpoint).

3) Write the formula for the final value of myNew[i] as a function of i and n. In general, this is the value that we will get if m (= #iterations in sequential for-iter loop) is large enough.

After a sufficiently large number of iterations, the iterated averaging code will converge with myNew[i] = myVal[i] = i / (n+1)

Barriers

- Question: how can we transform this code so as to ensure that all tasks say hello before any tasks say goodbye, without having to change the local variable?
- Approach 2: insert a "barrier" ("next" statement) between the hello's and goodbye's

```
1. // APPROACH 2
2. forallPhased (0, m - 1, (i) -> {
3.   int sq = i*i;
4.   System.out.println("Hello from task with square = " + sq);
5.   next(); // Barrier
6.   System.out.println("Goodbye from task with square = " + sq);
7. });
```

Phase 0 (lines 3–4)

Phase 1 (lines 6)

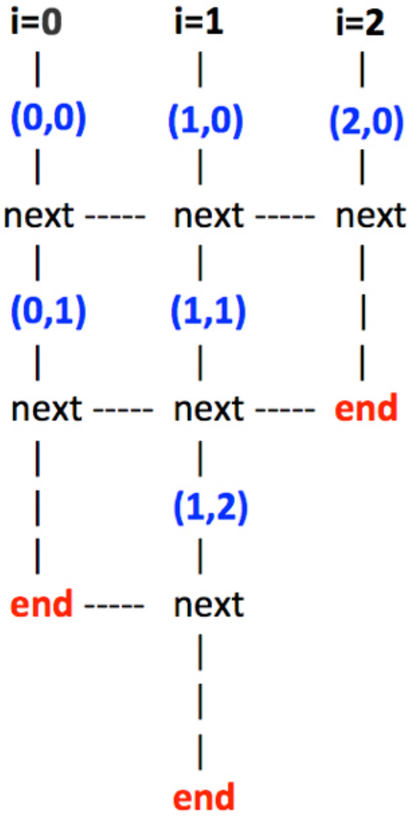
- next -> each forallPhased iteration waits at barrier until all iterations arrive (previous phase is completed), after which the next phase can start
  - Scope of next is the closest enclosing forallPhased statement
  - If a forallPhased iteration terminates before executing "next", then the other iterations don't wait for it

Worksheet: Forall Loops and Barriers

Draw a “barrier matching” figure similar to lecture 12 slide 11 for the code fragment below.

```
1.  String[] a = { “ab”, “cde”, “f” };
2.  . . . int m = a.length; . . .
3.  forallPhased (0, m-1, (i) -> {
4.    for (int j = 0; j < a[i].length(); j++) {
5.      // forall iteration i is executing phase j
6.      System.out.println("(" + i + "," + j + ")");
7.      next();
8.    }
9.  });
```

Solution



Summary of Phaser Construct (Lecture 33)

- Phaser allocation
  - HjPhaser ph = newPhaser(mode);
    - Phaser ph is allocated with registration mode
    - Phaser lifetime is limited to scope of Immediately Enclosing Finish (IEF)
- Registration Modes
  - HjPhaserMode.SIG, HjPhaserMode.WAIT, HjPhaserMode.SIG_WAIT, HjPhaserMode.SIG_WAIT_SINGLE
    - NOTE: phaser WAIT is unrelated to Java wait/notify (which we will study later)
- Phaser registration
  - asyncPhased (ph_1.inMode(<$mode_1$>), ph_2.inMode(<$mode_2$>), … () -> <stmt>)
    - Spawned task is registered with ph_1 in $mode_1$, ph_2 in $mode_2$, …
    - Child task's capabilities must be subset of parent's
    - asyncPhased <stmt> propagates all of parent's phaser registrations to child
- Synchronization
  - next();
    - Advance each phaser that current task is registered on to its next phase
    - Semantics depends on registration mode
    - Barrier is a special case of phaser, which is why next is used for both

Worksheet: Reordered Asyncs with One Phaser

Task A4 has been moved up to line 6. Does this change the computation graph in slide 9? If so, draw the new computation graph. If not, explain why the computation graph is the same.

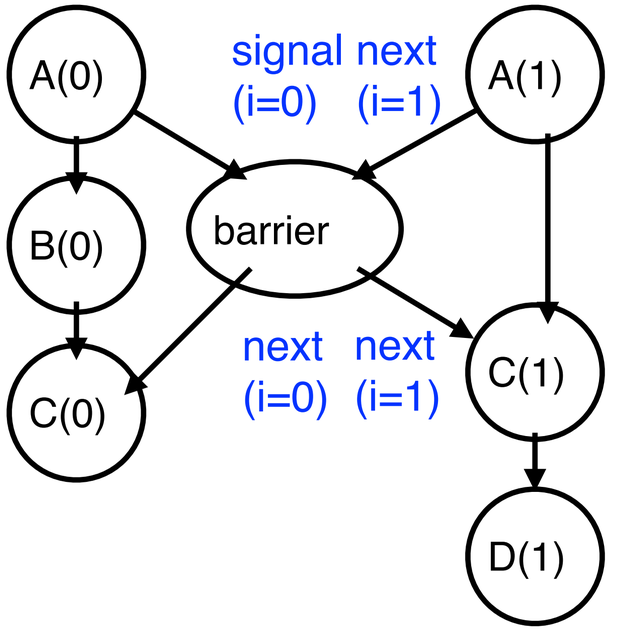
No, A4 still needs to wait on A2 and A3 to signal before it can start doA4Phase2().

```
1.  finish (() -> {
2.      ph = newPhaser(SIG_WAIT); // mode is SIG_WAIT
3.      asyncPhased(ph.inMode(SIG), () -> {
4.        // A1 (SIG mode)
5.        doA1Phase1(); next(); doA1Phase2(); });
6.      asyncPhased(ph.inMode(HjPhaserMode.WAIT), () -> {
7.       // A4 (WAIT mode)
8.       doA4Phase1(); next(); doA4Phase2(); });
9.      asyncPhased(ph.inMode(SIG_WAIT), () -> {
10.       // A2 (SIG_WAIT mode)
11.       doA2Phase1(); next(); doA2Phase2(); });
12.     asyncPhased(ph.inMode(HjPhaserMode.SIG_WAIT), () -> {
13.      // A3 (SIG_WAIT mode)
14.      doA3Phase1(); next(); doA3Phase2(); });
15.   });
```


Signal statement & Fuzzy barriers

- When a task T performs a signal operation, it notifies all the phasers it is registered on that it has completed all the work expected by other tasks ("shared" work) in the current phase.
- Later, when T performs a next operation, the next degenerates to a wait since a signal has already been performed in the current phase.
- The execution of "local work" between signal and next is overlapped with the phase transition (referred to as a "split-phase barrier" or "fuzzy barrier")

```
1. forallPhased(point[i] : [0:1]) {
2.   A(i); // Phase 0
3.   if (i==0) { signal; B(i); }
4.   next; // Barrier
5.   C(i); // Phase 1
6.   if (i==1) { D(i); }
7. }
```



Worksheet: Critical Path Length for Computation with Signal Statement

Compute the WORK and CPL values for the program shown below. How would they be different if the signal() statement was removed? (Hint: draw a computation graph as in slide 11)

WORK = 204, CPL = 102. If the signal() is removed, CPL = 202.

```
1.finish(() -> {
2.  final HjPhaser ph = newPhaser(SIG_WAIT);
3.  asyncPhased(ph.inMode(SIG_WAIT), () -> { // Task T1
4.    A(0); doWork(1);   // Shared work in phase 0
5.    signal();
6.    B(0); doWork(100); // Local work in phase 0
7.    next(); // Wait for T2 to complete shared work in phase 0
8.    C(0); doWork(1);
9.  });
10.  asyncPhased(ph.inMode(SIG_WAIT), () -> { // Task T2
11.    A(1); doWork(1);   // Shared work in phase 0
12.    next(); // Wait for T1 to complete shared work in phase 0
13.    C(1); doWork(1);
14.    D(1); doWork(100); // Local work in phase 0
15.  });
16.}); // finish
```


What is “Eureka Style” Computation? (Lecture 35)

- Many optimization and search problems attempts to find a result with a certain property or cost
- Announce when a result has been found
  - An "aha!" moment – **Eureka** event
  - Can make rest of the computation unnecessary

==> Opportunities for “speculative parallelism”, e.g., Parallel Search, Branch and Bound Optimization, Soft Real-Time Deadlines, Convergence Iterations, . . .

Image source: http://www.netstate.com/states/mottoes/images/ca_eureka.jpg

Tree Min Index Search Example

```
HjExtremaEureka<Integer> eureka = newExtremaEureka(
  Integer.MAX_VALUE, (Integer i, Integer j) -> j.compareTo(i));
finish(eureka, () -> {
    async(() -> {
        minIndexSearchBody(eureka, rootNode, elemToSearch);
    });
});

private static void minIndexSearchBody(
  HjExtremaEureka<Integer> eureka, Node rootNode,
  int elemToSearch) throws SuspendableException {
    eureka.check(rootNode.id);
    if (rootNode.value == elemToSearch) {
        eureka.offer(rootNode.id);
    }
    if (rootNode.left != null) {
        async(() -> {
            minIndexSearchBody(eureka, rootNode.left, elemToSearch);
        });
    }
    if (rootNode.right != null) {
        async(() -> {
            minIndexSearchBody(eureka, rootNode.right, elemToSearch);
        });
    }
}
```

Inputs:

- binary tree, T
- id for each node in T, in breadth-first order e.g., root.id = 0, root.left.id = 1, root.right.id = 2, …
- value for each node in T that is the search target

Outputs:

- calls to offer() update eureka with minimum id found so far (among those that match)
- calls to check() can lead to early termination if the argument is >= than current minimum in eureka
- final value of eureka contains minimum id of node with value == elemToSearch

Worksheet: Finding maximal index of goal in matrix

Below is a code fragment intended to find the maximal (largest) index of a goal value that occurs multiple times in the input matrix. Could we make the code more efficient? If so, explain why?

```
1.  class AsyncFinishEurekaSearchMaxIndexOfGoal {
2.    HjEureka eurekaFactory() {
3.      comparator = (cur, newVal) -> { // cur is initially [-1, -1]
          (cur.x==newVal.x) ? (newVal.y - cur.y) : (newVal.x - cur.x) }
4.      return new MaximaEureka([-1, -1], comparator)
5.    }
6.    int[] doWork(matrix, goal) {
7.      val eu = eurekaFactory()
8.      finish (eu, () -> { // eureka registration
9.        forasync (0, matrix.length - 1, (r) ->
10.         procRow(matrix(r), r, goal));
11.     });
12.     return eu.get()
13.   }
14.   void procRow(array, r, goal) {
15.     for (int c = 0; c < array.length(); c++)
16.       check([r, c]) // terminate if comparator returns negative
17.       if goal.match(array(c)) offer([r, c]) // updates cur in eureka
18.   } }
```

This code is inefficient due to starting c at 0 instead of array.length() -1. We could also use forasyncChunked to reduce the number of tasks created.