COMP 322: Fundamentals of Parallel Programming

Lecture 39: Review of Modules 2 & 3 (Lectures 20-37)

Vivek Sarkar, Eric Allen
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

HJ isolated construct (Lecture 20)

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) can lead to a deadlock, if used incorrectly

Object-based isolation

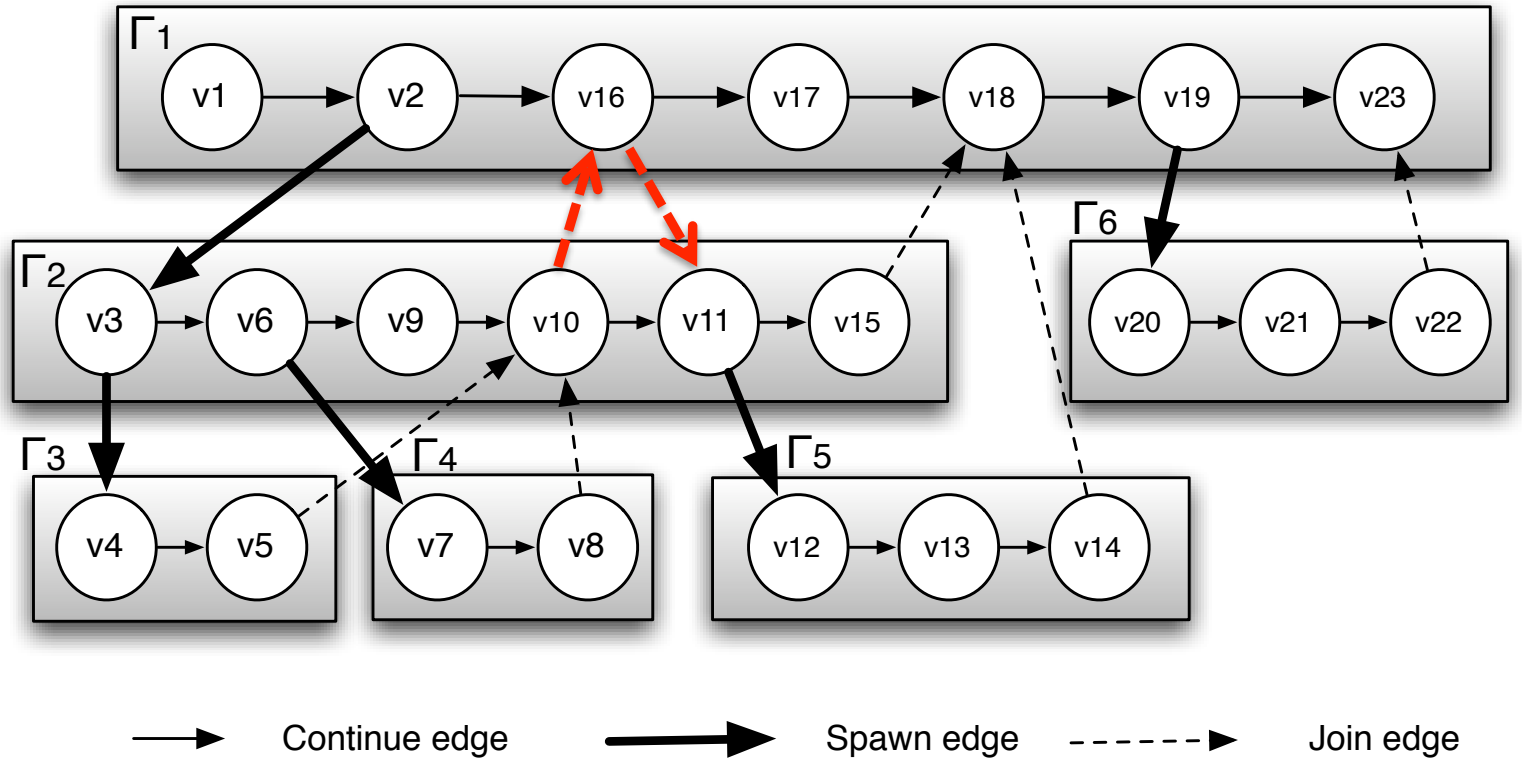
```
isolated(obj1, obj2, …, () -> <body>)
```

- 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 objstec 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 (Lab 9)

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


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



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

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

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

Worksheet #20 solution: Insertion of isolated for correctness

The goal of IsolatedPRNG is to implement a single Pseudo Random Number Generator object that can be shared by multiple tasks. Show the isolated statement(s) that you can insert in method nextSeed() to avoid data races and guarantee proper semantics.

```
1.class IsolatedPRNG {
2. private int seed;
3. public int nextSeed() {
4. return isolatedWithReturn(this, ()->{
5.    int retVal = seed;
6.    seed = nextInt(retVal);
7.    return retVal;
8.  });
9.} // nextSeed()
10.  . . .
11.} // IsolatedPRNG
```

```
12.main() { // Pseudocode
13.  // Initial seed = 1
14.  IsolatedPRNG r = new IsolatedPRNG(1);
15.  async(() -> { print r.nextSeed(); ... });
16.  async(() -> { print r.nextSeed(); ... });
17.} // main()
```

Note that enclosing line 5 and line 6 in separate isolated constructs will avoid data races, but it will not guarantee the semantics of a sequential Pseudo Random Number Generator for a given PRNG object.

Eureka construct (Lecture 21)

1. `eureka = eurekaFactory()`
2. `finish (eureka) S1`
   - Multiple `finish`'es can register on same Eureka
   - Wait for all tasks to finish as before
     - Except that some tasks may terminate early when eureka is resolved
3. `async`
   - Inherits eureka registrations from immediately-enclosing finish
4. `offer()`
   - Triggers eureka event on registered eureka
5. `check()`
   - Causes task to terminate if eureka resolved

Eureka Variants

```
def eurekaFactory() {
  val initValue = [-1, -1]
  return new SearchEureka(initValue)
}
```

```
def eurekaFactory() {
  val K = 4
  return new CountEureka(K)
}
```

```
def eurekaFactory() {
  // comparator to compare indices
  val comparator = (a, b) -> {
    ((a.x - b.x) == 0) ? (a.y - b.y) : (a.x - b.x)
  }
  val initValue = [INFINITY, INFINITY]
  return new MinimaEureka(initValue, comparator)
}
```

```
def eurekaFactory() {
  val time = 4.seconds
  return new TimerEureka(time)
}
```

```
def eurekaFactory() {
  val units = 400
  return new EngineEureka(units)
}
```


java.util.concurrent.atomic.AtomicReference (Lecture 22)

- Constructors
  - `new AtomicReference()`
    - Creates a new AtomicReference with initial value 0
  - `new AtomicReference(Object init)`
    - Creates a new AtomicReference with the given initial value
- Selected methods
  - `int getAndSet(Object newRef)`
    - Atomically get current value of the atomic variable, and set value to newRef
  - `int compareAndSet(Object expect, Object update)`
    - Atomically check if current value = expect. If so, replace the value of the atomic variable by update and return true. Otherwise, return false.

java.util.concurrent. AtomicReference methods and their equivalent isolated statements

j.u.c.atomic Class and Constructors	j.u.c.atomic Methods	Equivalent HJ isolated statements
AtomicReference	Object o = v.**get**();	Object o; isolated (v) o = v.ref;
	v.**set**(newRef);	isolated (v) v.ref = newRef;
AtomicReference() // init = null	Object o = v.**getAndSet**(newRef);	Object o; isolated (v) { o = v.ref; v.ref = newRef; }

AtomicReference(init)	boolean b =
	v.**compareAndSet**
	(expect,update);
		boolean b;
		isolated (v)
		if (v.ref==expect) {v.ref=update; b=true;}
		else b = false;

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

AtomicReference<T> can be used to specify a type parameter.

Parallel Spanning Tree Algorithm using AtomicReference

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


Worksheet #22 solution: Abstract Metrics with Isolated Constructs

Q: Compute the WORK and CPL metrics for this program. Indicate if your answer depends on the execution order of isolated constructs.

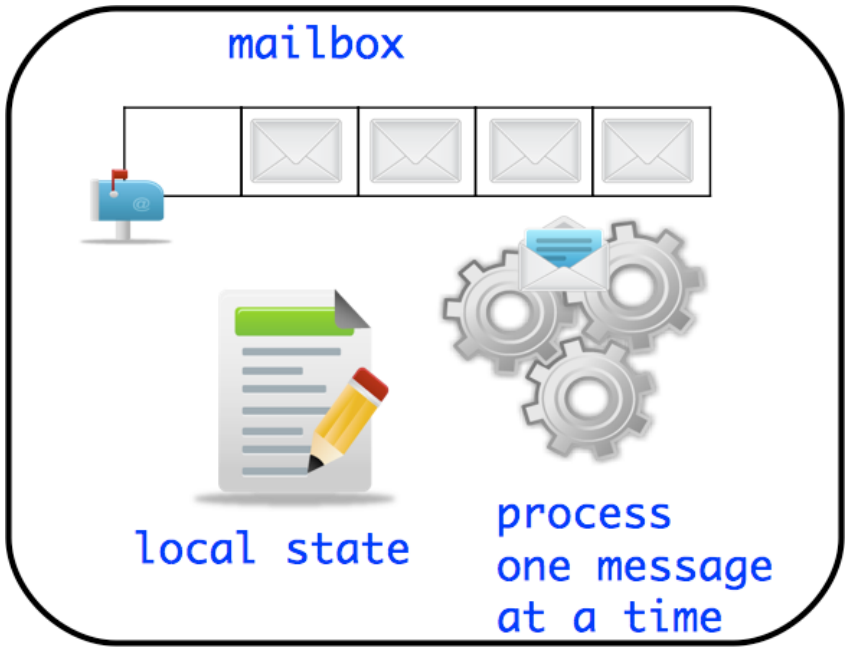
```
1.    finish(() -> {
2.          for (int i = 0; i < 5; i++) {
3.              async(() -> {
4.                        doWork(2);
5.                        isolated(() -> { doWork(1); });
6.                        doWork(2);
7.                    }); // async
8.          } // for
9.     }); // finish
```

Answer: WORK = 25, CPL = 9. These metrics do not depend on the execution order of isolated constructs.

The Actor Model (Lectures 23-25)

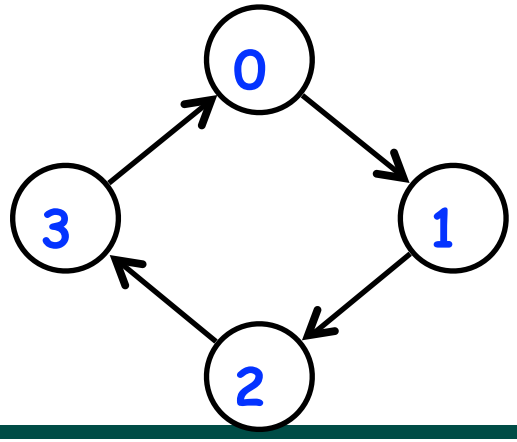
Actor states

- 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



ThreadRing (Coordination) Example

```
1.  finish(() -> {
2.    int numThreads = 4;
3.    int numberOfHops = 10;
4.    ThreadRingActor[] ring =
        new ThreadRingActor[numThreads];
5.    for(int i=numThreads-1;i>=0; i--) {
6.      ring[i] = new ThreadRingActor(i);
7.      ring[i].start();
8.      if (i < numThreads - 1) {
9.        ring[i].nextActor(ring[i + 1]);
10.   } }
11.   ring[numThreads-1].nextActor(ring[0]);
12.   ring[0].send(numberOfHops);
13. }); // finish
```




```
14. class ThreadRingActor
15.     extends Actor<Object> {
16.   private Actor<Object> nextActor;
17.   private final int id;
18.   ...
19.   public void nextActor(
        Actor<Object> nextActor) {...}
20.   void process(Object theMsg) {
21.     if (theMsg instanceof Integer) {
22.       Integer n = (Integer) theMsg;
23.       if (n > 0) {
24.         println("Thread-" + id +
25.           " active, remaining = " + n);
26.         nextActor.send(n - 1);
27.       } else {
28.         println("Exiting Thread-"+ id);
29.         nextActor.send(-1);
30.         exit();
31.       }
32.     } else {
33.     /* ERROR - handle appropriately */
34. } } }
```


Worksheet #23 solution: Interaction between finish and actors

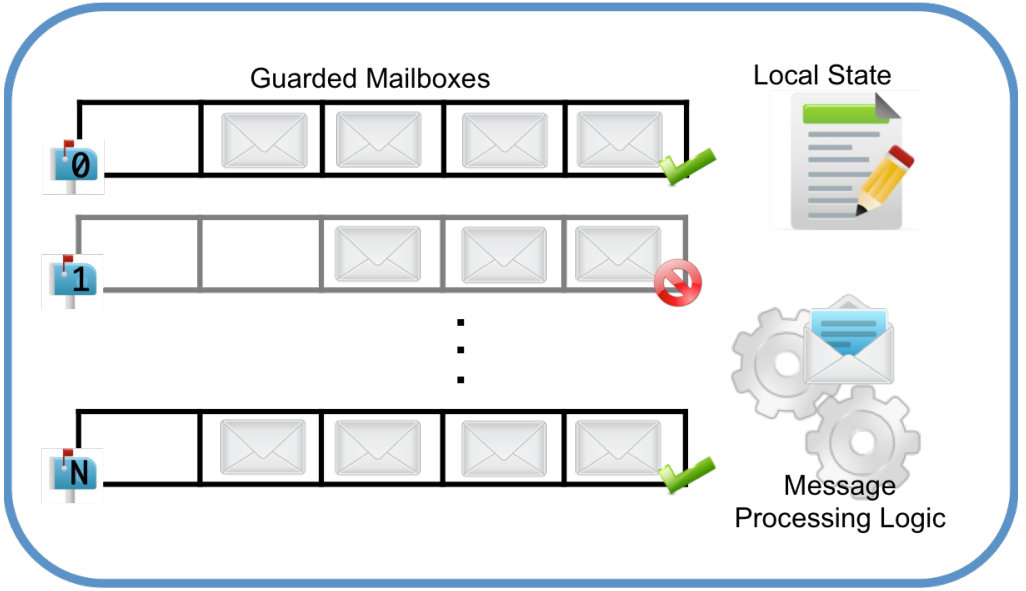
What would happen if the end-finish operation from slide 14 was moved from line 13 to line 11 as shown below?

```
1.  finish(() -> {
2.     int numThreads = 4;
3.     int numberOfHops = 10;
4.     ThreadRingActor[] ring = new ThreadRingActor[numThreads];
5.     for(int i=numThreads-1;i>=0; i--) {
6.       ring[i] = new ThreadRingActor(i);
7.       ring[i].start(); // like an async
8.       if (i < numThreads - 1) {
9.         ring[i].nextActor(ring[i + 1]);
10.      } }
11. }); // finish
12.ring[numThreads-1].nextActor(ring[0]);
13.ring[0].send(numberOfHops);
```

Deadlock: 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().

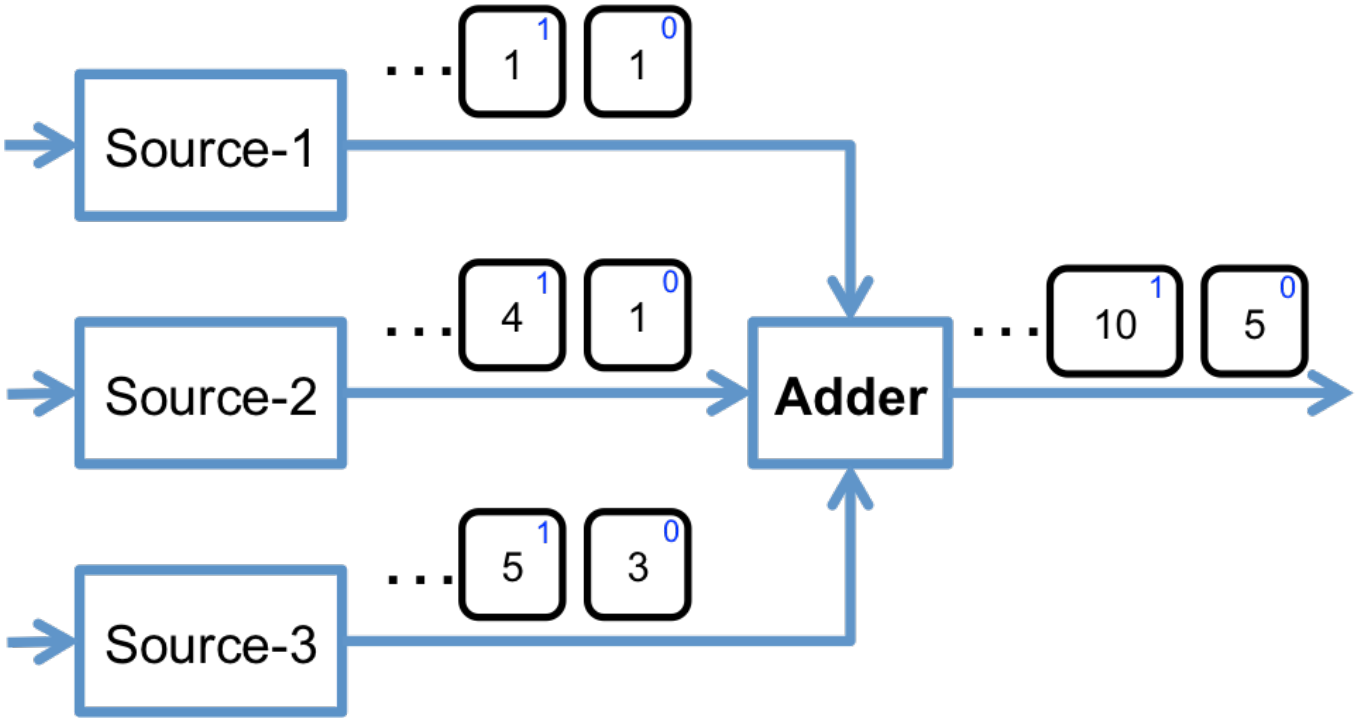
From Actors to Selectors (Actors with multiple mailboxes)

- The basic idea behind pause() and resume() is to enable/disable processing of messages in an actor's mailbox
- This idea can be extended to *selectors*
  - —a selector is an actor with multiple mailboxes numbered 0…n-1
  - —s.send(i,msg) sends msg to mailbox i of selector s
  - —disable(i) disables mailbox i (like "pausing" mailbox i)
  - —enable(i) enables mailbox i (like "resuming" mailbox i)
  - —enableAll() enables all the mailboxes



Join Patterns in Streaming Applications

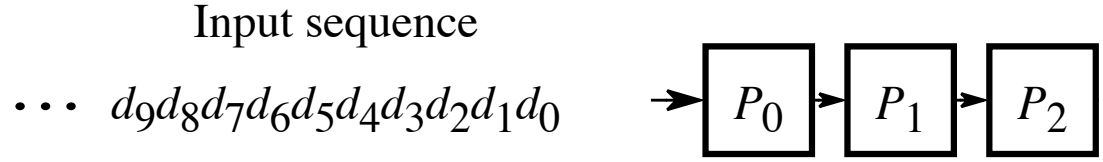
- **Selectors can be used to implement an adder for 3 input streams using actors (see slide 16 in Lecture 24)**
- Messages from two or more data streams are combined together into a single message
- Joins need to match inputs from each source



Solution to Worksheet #24: Ideal Parallelism in 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.

Solution: WORK = 300, CPL = 102




```
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.         }
```


Linearizability of Concurrent Objects (Lecture 25)

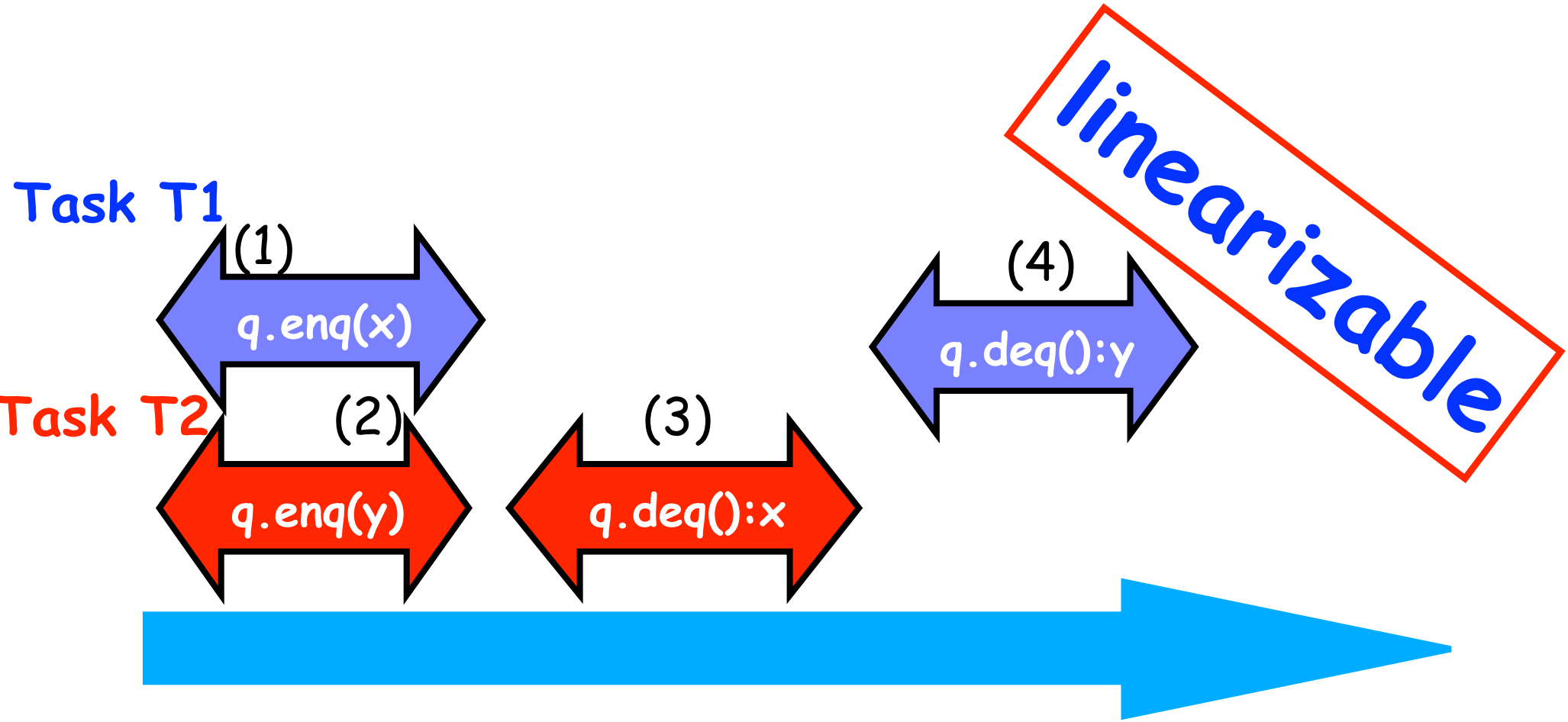
Concurrent object

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

Linearizability

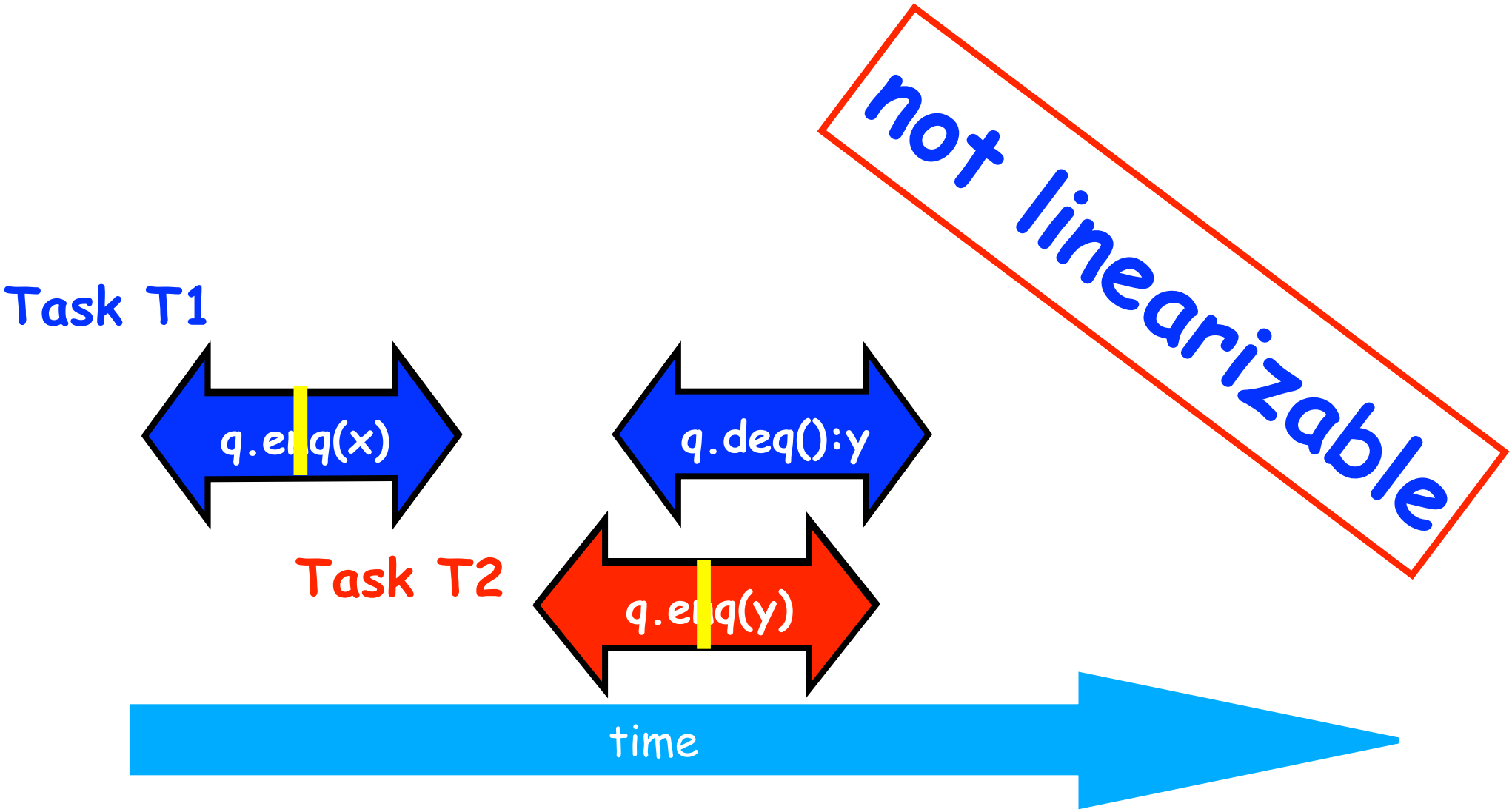
- Assume that each method call takes effect "instantaneously" at some distinct point in time between its invocation and return.
- An execution is linearizable if we can choose instantaneous points that are consistent with a sequential execution in which methods are executed at those points
- An object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

Example 1



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

Example 2: is this execution linearizable?



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

Solution to Worksheet #25: Linearizability of method calls on a concurrent object

Is this a linearizable execution for a FIFO queue, q?

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

No! q.enq(x) must precede q.enq(y) in all linear sequences of method calls invoked on q. It is illegal for the q.deq() operation to return y.

Two-way Parallel Array Sum using Java Threads (Lecture 26)

```
1.  // Start of main thread
2.  sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
3.  Thread t1 = new Thread(() -> {
4.      // Child task computes sum of lower half of array
5.      for(int i=0; i < X.length/2; i++) sum1 += X[i];
6.    });
7.  t1.start();
8.  // Parent task computes sum of upper half of array
9.  for(int i=X.length/2; i < X.length; i++) sum2 += X[i];
10. // Parent task waits for child task to complete (join)
11. t1.join();
12. return sum1 + sum2;
```


Objects and Locks in Java --- synchronized statements and methods

- 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
- Java language does not enforce any relationship between object used for locking and objects accessed in isolated code
  - If same object is used for locking and data access, then the object behaves like a monitor
- 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

Avoiding Dynamic Order Deadlocks

- The solution is to induce a lock ordering
  - In this example, use an existing unique numeric key, acctId, to establish an order

```
public class SafeTransfer {

    public void transferFunds(Account from, Account to, int amount) {

        Account firstLock, secondLock;
        if (fromAccount.acctId == toAccount.acctId)
            throw new Exception("Cannot self-transfer");
        else if (fromAccount.acctId < toAccount.acctId) {
            firstLock = fromAccount;
            secondLock = toAccount;
        }
        else {
            firstLock = toAccount;
            secondLock = fromAccount;
        }
        synchronized (firstLock) {
          synchronized (secondLock) {
            from.subtractFromBalance(amount);
            to.addToBalance(amount);
          }
        }
    }
}
```


Solution to Worksheet #26: Java Threads

1) Write a sketch of the pseudocode for a Java threads program that exhibits a data race using start() and join() operations.

```
1.  // Start of thread t0 (main program)
2.  sum1 = 0; sum2 = 0; // Assume that sum1 & sum2 are fields
3.  // Compute sum1 (lower half) and sum2 (upper half) in parallel
4.  final int len = X.length;
5.  Thread t1 = new Thread(() -> {
6.                      for(int i=0 ; i < len/2 ; i++) sum1+=X[i];});
7.  t1.start();
8.  Thread t2 = new Thread(() -> {
9.                      for(int i=len/2 ; i < len ; i++)
    sum2+=X[i];});
10. t2.start();
11. int sum = sum1 + sum2; // data race between t0 & t1, and t0 & t2
12. t1.join(); t2.join();
```

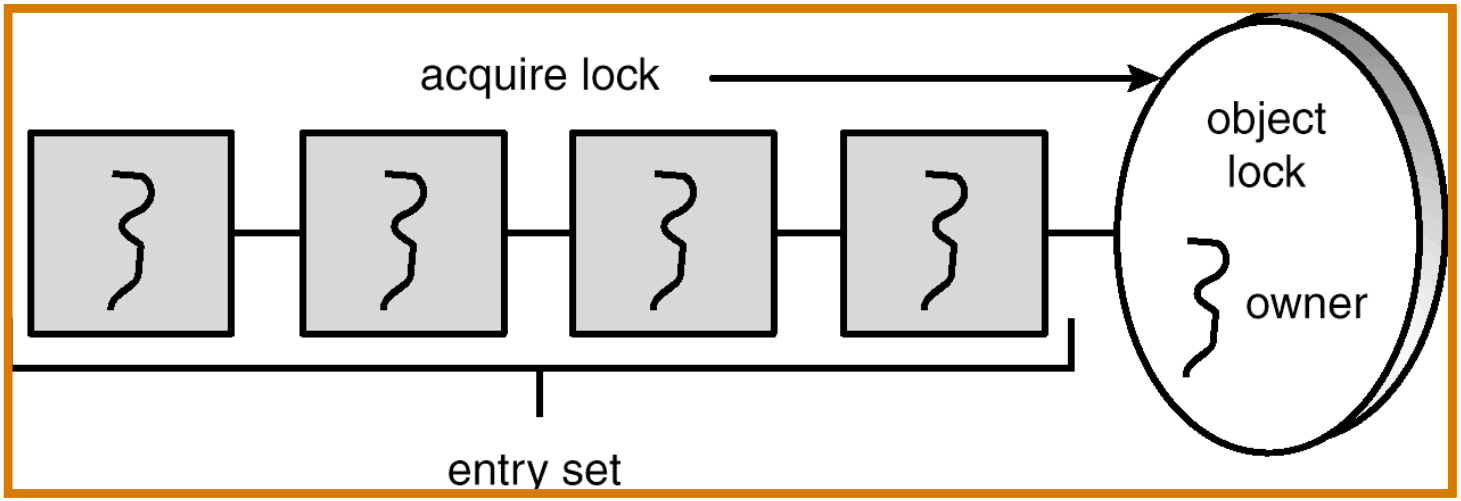

Solution to Worksheet #26: Java Threads (contd)

2) Write a sketch of the pseudocode for a Java threads program that exhibits a data race using synchronized statements.

```
1.  // Start of thread t0 (main program)
2.  sum = 0; // static int field
3.  Object a = new ... ;
4.  Object b = new ... ;
5.  Thread t1 = new Thread(() -> { synchronized(a) { sum++; } });
6.  Thread t2 = new Thread(() -> { synchronized(b) { sum++; } });
1.  t1.start();
7.  t2.start(); // data race between t1 & t2
8.  t1.join(); t2.join();
```


Implementation of Java synchronized statements/methods (Lecture 27)

- Every object has an associated lock
- "synchronized" is translated to matching monitorenter and monitorexit bytecode instructions for the Java virtual machine
  - monitorenter requests "ownership" of the object's lock
  - monitorexit releases "ownership" of the object's lock
- If a thread performing monitorenter does not gain ownership of the lock (because another thread already owns it), it is placed in an unordered "entry set" for the object's lock



java.util.concurrent.locks.Lock interface (Lecture 28)

```
interface Lock {
   void lock();
   void lockInterruptibly() throws InterruptedException;
   boolean tryLock(); // return false if lock is not obtained
   boolean tryLock(long timeout, TimeUnit unit)
                                throws InterruptedException;
   void unlock();
   Condition newCondition();
   // can associate multiple condition vars with lock
}
```

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

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

Example code

```
class Hashtable<K,V> {
  …
  // coarse-grained, one lock for table
  ReadWriteLock 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();
  }
}
```


Worksheet #28 solution: use of tryLock()

Extend the transferFunds() method from Lecture 26 (shown below) to use j.u.c. locks with tryLock() instead of synchronized, and to return a boolean value --- true if it succeeds in obtaining both locks and performing the transfer, and false otherwise. *Assume that each Account object contains a reference to a dedicated ReentrantLock object.* Sketch your answer below using pseudocode. Can you create a deadlock with multiple calls to transferFunds() in parallel?

```
1.  public boolean transferFunds(Account from, Account to,
2.                                   int amount) {
3.    // Assume that each Account object has a lock field of
4.    // a type/class that implements java.util.concurrent.locks.Lock
5.    // Assume that no exception can be thrown in this code
6.    // Calls to this method can never lead to a deadlock
7.    if (! from.lock.trylock()) return false;
8.    if (! to.lock.trylock()) { from.lock.unlock(); return false; }
9.    from.subtractFromBalance(amount); to.addToBalance(amount);
10.   // NOTE: unlock() should be in try-catch-finally for robustness
11.   from.lock.unlock(); to.lock.unlock();
12.   return true;
13. }
```


Safety vs. Liveness (Lecture 29)

- In a concurrent setting, we need to specify both the safety and the liveness properties of an object
- Need a way to define
  - Safety: when an implementation is functionally correct (does not produce a wrong answer)
  - Liveness: the conditions under which it guarantees progress (completes execution successfully)

- Data race freedom is a desirable safety property for most parallel programs
- Linearizability is a desirable safety property for most concurrent objects
- What about liveness properties?

Liveness

- Liveness = a program's ability to make progress in a timely manner
- Is termination a requirement for liveness?
  - But some applications are designed to be non-terminating
- Different levels of liveness guarantees (from weaker to stronger)
  1. Deadlock freedom
  2. Livelock freedom
  3. Starvation freedom
  4. Bounded wait

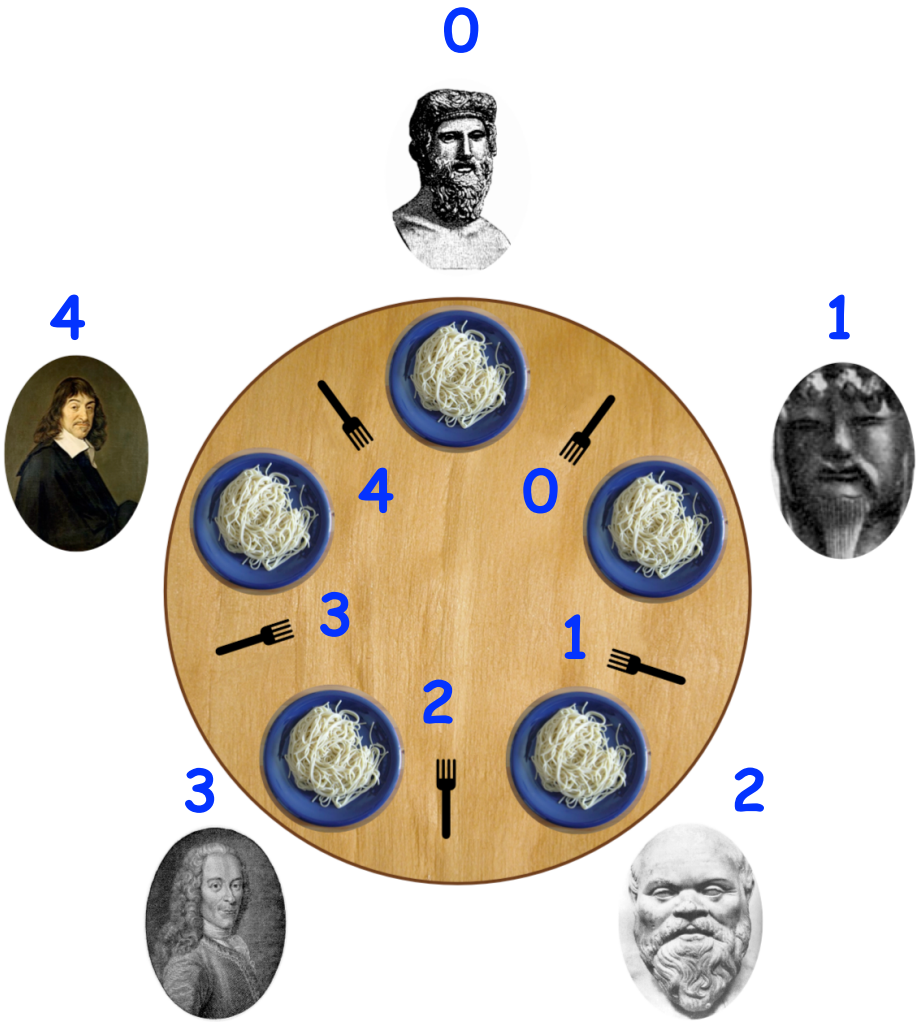
Worksheet #29: Liveness Guarantees

```
/** 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))
              // commit
              return current;
      }
  }
```

Assume that multiple tasks call getAndAdd() repeatedly in parallel. Can this implementation of getAndAdd() lead to executions with a) deadlock, b) livelock, c) starvation, or d) unbounded wait? Write and explain your answer below.

c) starvation and d) unbounded wait are both possible
NOTE: a parallel program execution that terminates exhibits none of a), b), or c).

The Dining Philosophers Problem (Lecture 30)



Constraints

- Five philosophers either eat or think
- They must have two forks to eat (chopsticks are a better motivation!)
- Can only use forks on either side of their plate
- No talking permitted

Goals

- Progress guarantees
  - Deadlock freedom
  - Livelock freedom
  - Starvation freedom
  - Maximum concurrency (no one should starve if there are available forks for them)

Worksheet #30: Characterizing Solutions to the Dining Philosophers Problem

For the five solutions studied in Lecture #29, indicate in the table below which of the following conditions are possible and why:

1. Deadlock: when all philosopher tasks are blocked
2. Livelock: when all philosopher tasks are executing (i.e., no philosopher is blocked) but ALL philosophers are starved (never get to eat)
3. Starvation: when one or more philosophers are starved (never get to eat)
4. Non-Concurrency: when more than one philosopher cannot eat at the same time, even when resources are available i.e., not being used

NOTES:

- Deadlock implies Starvation and Non-Concurrency
- Livelock implies Starvation and Non-Concurrency

	Deadlock	Livelock	Starvation	Non-concurrency
Solution 1: synchronized	Yes	No	Yes	Yes
Solution 2: tryLock/ unLock	No	Yes	Yes	Yes
Solution 3: isolated	No	No	Yes	Yes
Solution 4: object-based isolation	No	No	Yes	No
Solution 5: semaphores	No	No	No	No

Places (Lecture 31)

here() = place at which current task is executing

numPlaces() = total number of places (runtime constant)

Specified by value of p in runtime option:

```
HjSystemProperty.numPlaces.set(p);
```

place(i) = place corresponding to index i

<place-expr>.toString() returns a string of the form "place(id=0)"

<place-expr>.id() returns the id of the place as an int

asyncAt(P, () -> S)

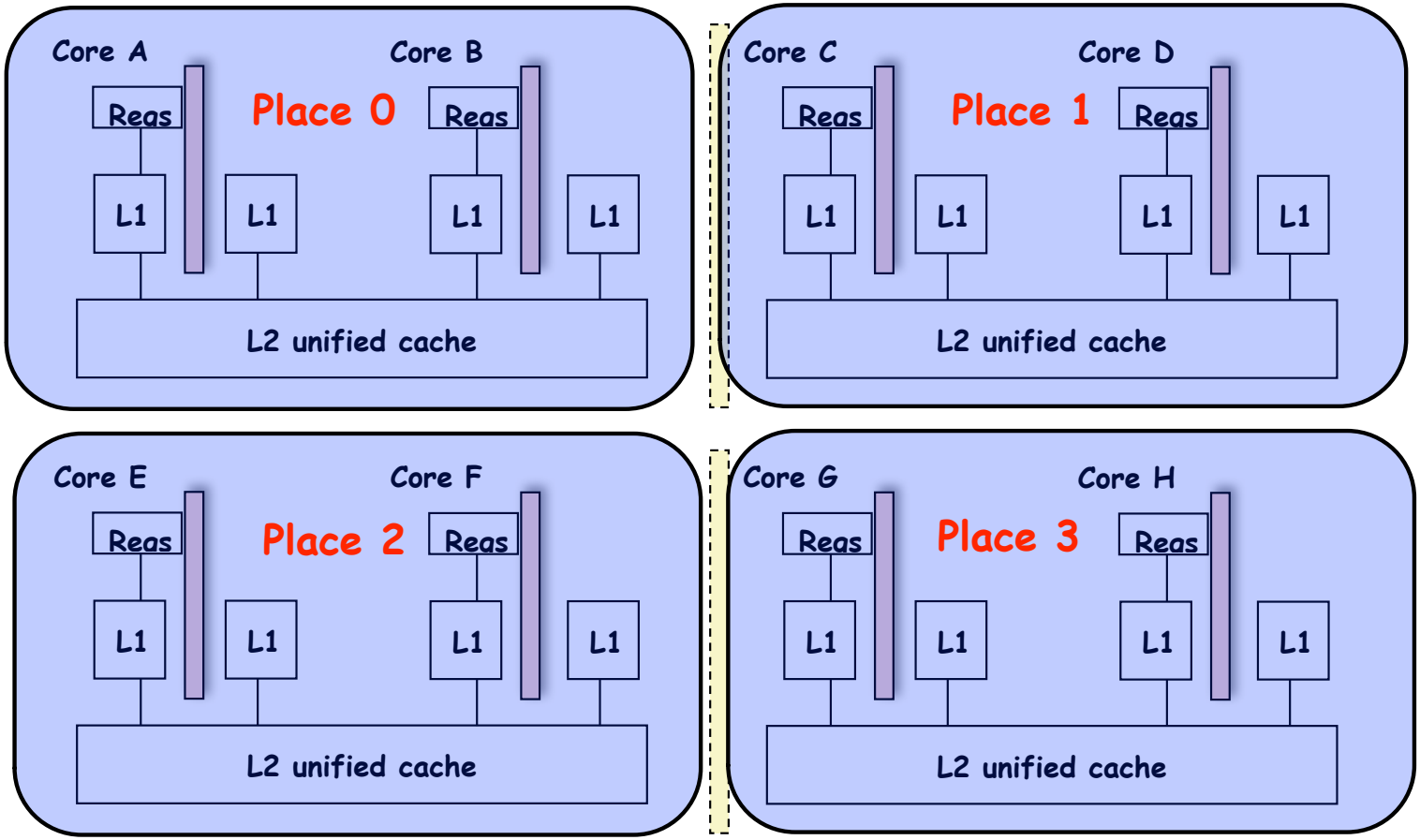
- Creates new task to execute statement S at place P
- async(() -> S) is equivalent to asyncAt(here(), () -> S)
- Main program task starts at place(0)

Note that here() in a child task refers to the place P at which the child task is executing, not the place where the parent task is executing

Example of 4:2 option on an 8-core node (4 places w/ 2 workers per place)

```
// 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 #32 solution: Spark and Map-Reduce

```
val points = spark.textFile(…).map(parsePoint).cache()

var w = Vector.random(D) // current separating plane

for (i <- 1 to ITERATIONS) {
  val gradient = points.map(doWork(1)).reduce(_ + _)

  w -= gradient
}

println("Final separating plane: " + w)
```

There are ITERATIONS sequential iterations, each mapping doWork in parallel over every value in points, which is immediately forced by a reduce. So,
work = ITERATIONS * |points|
CPL = ITERATIONS.

Introduction to MPI (Lectures 33-34)

main() is enclosed in an implicit "forall" --- each process runs a separate instance of main() with "index variable" = myrank

```
1.import mpi.*;
2.class Hello {
3.    static public void main(String[] args) {
4.       // Init() be called before other MPI calls
5.       MPI.Init(args); /
6.       int npes = MPI.COMM_WORLD.Size()
7.       int myrank = MPI.COMM_WORLD.Rank() ;
8.       System.out.println("My process number is " + myrank);
9.       MPI.Finalize(); // Shutdown and clean-up
10.   }
11.}
```


Worksheet #33 solution: MPI send and receive

```
1. int a[], b[];
2. ...
3. if (MPI.COMM_WORLD.rank() == 0) {
4.     MPI.COMM_WORLD.Send(a, 0, 10, MPI.INT, 1, 1);
5.     MPI.COMM_WORLD.Send(b, 0, 10, MPI.INT, 1, 2);
6. }
7. else {
8.     Status s2 = MPI.COMM_WORLD.Recv(b, 0, 10, MPI.INT, 0, 2);
9.     Status s1 = MPI.COMM_WORLD.Recv(a, 0, 10, MPI_INT, 0, 1);
10.    System.out.println("a = " + a + " ; b = " + b);
11.}
12. ...
```

Question: In the space below, indicate what values you expect the print statement in line 10 to output (assuming the program is invoked with 2 processes).

Answer: Nothing! The program will deadlock due to mismatched tags, with process 0 blocked at line 4, and process 1 blocked at line 8.

Collective Communications

- A popular feature of MPI is its family of collective communication operations.
- Each collective operation is defined over a communicator (most often, MPI.COMM_WORLD)

— Each collective operation contains an *implicit barrier*. The operation completes and execution continues when all processes in the communicator perform the *same* collective operation.

— A mismatch in operations results in *deadlock* e.g.,

Process 0: .... MPI.Bcast(...) ....

Process 1: .... MPI.Bcast(...) ....

Process 2: .... MPI.Gather(...) ....

- A simple example is the broadcast operation: all processes invoke the operation, all agreeing on one root process. Data is broadcast from that root.

  void Bcast(Object buf, int offset, int count, Datatype type, int root)

  - Broadcast a message from the process with rank root to all processes of the group

Worksheet #34 solution: MPI Gather

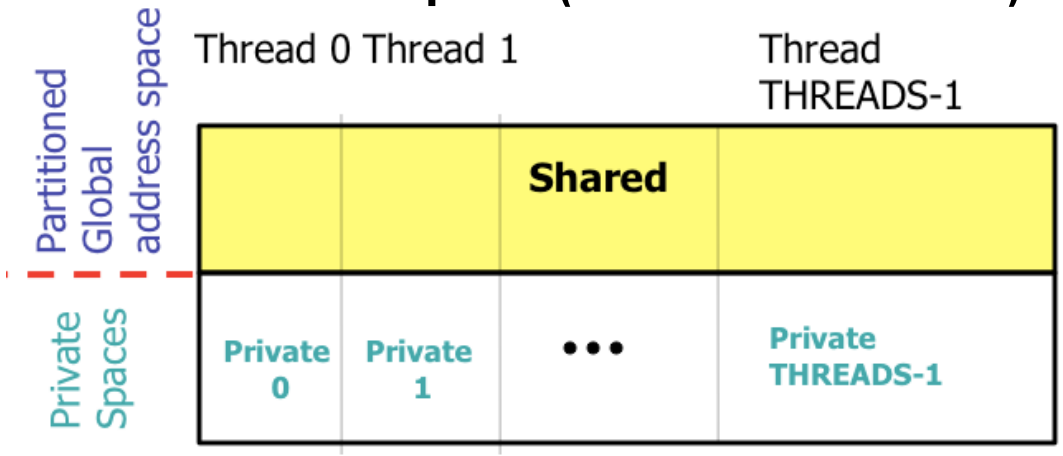
```
1.    MPI.Init(args) ;
2.    int myrank = MPI.COMM_WORLD.Rank() ;
3.    int numProcs = MPI.COMM_WORLD.Size() ;
4.    int size = ...;
5.    int[] sendbuf = new int[size];
6.    int[] recvbuf = new int[???];
7.    . . . // Each process initializes sendbuf
8.    MPI.COMM_WORLD.Gather(sendbuf, 0, size, MPI.INT,
9.                          recvbuf, 0, size, MPI.INT,
10.                         0/*root*/);
11.   . . .
12.   MPI.Finalize();
```

In the space below, indicate what value should be provided instead of ??? in line 6, and how it should depend on myrank.

Answer: it should be numProcs*size for rank 0, and zero for all other ranks.

UPC Execution Model (Lecture 35)

- Multiple threads working independently in a SPMD fashion
  - —MYTHREAD specifies thread index (0..THREADS-1)
    - – Like MPI processes and ranks
  - —# threads specified at compile-time or program launch
- Partitioned Global Address Space (different from MPI)



- Threads synchronize as necessary using
  - —synchronization primitives
  - —shared variables

Worksheet #35 solution: UPC data distributions

In the following example from slide 22, assume that each UPC array is distributed by default across threads with a cyclic distribution. In the space below, identify an iteration of the upc_forall construct for which all array accesses are local, and an iteration for which all array accesses are non-local (remote). Explain your answer in each case.

```
shared int a[100],b[100], c[100];
int i;
upc_forall (i=0; i<100; i++; (i*THREADS)/100)
   a[i] = b[i] * c[i];
```

Solution:

- Iteration 0 has affinity with thread 0, and accesses a[0], b[0], c[0], all of which are located locally at thread 0
- Iteration 1 has affinity with thread 0, and accesses a[1], b[1], c[1], all of which are located remotely at thread 1

Example use of volatile declarations (Lecture 36)

```
1. public class NoVisibility {
2.  private static volatile boolean ready;
3.  private static volatile int number;
4.
5.  private static class ReaderThread extends Thread {
6.    public void run() {
7.      while (!ready) Thread.yield()
8.      System.out.println(number)
9.    }
10.  }
11.
12.  public static void main(String[] args) {
13.    new ReaderThread().start();
14.    number = 42;
15.    ready = true;
16.  }
17. }
```

Declaring number and ready as volatile ensures happens-before-edges: 14-->15-->7-->8, thereby ensuring that only 42 will be printed

Worksheet #36 solution: Double Checked Locking Idiom in Java

Consider two threads calling the getHelper() method in parallel:

1) Can you construct a possible data race if they call the unoptimized version of getHelper() in lines 3-8?

No race possible (monitor-based synchronization)

2) Can you construct a possible data race if they call the optimized version of getHelper() in lines 12-21?

Yes, thread T1 can assign helper in line 16 while thread T2 reads helper in line 13

3) How will your answer to 2) change if the helper field in line 11 was declared as volatile?

Technically, no data race since volatile declaration causes read and write of helper to be (semantically) enclosed in isolated blocks. But there can be nondeterminism.

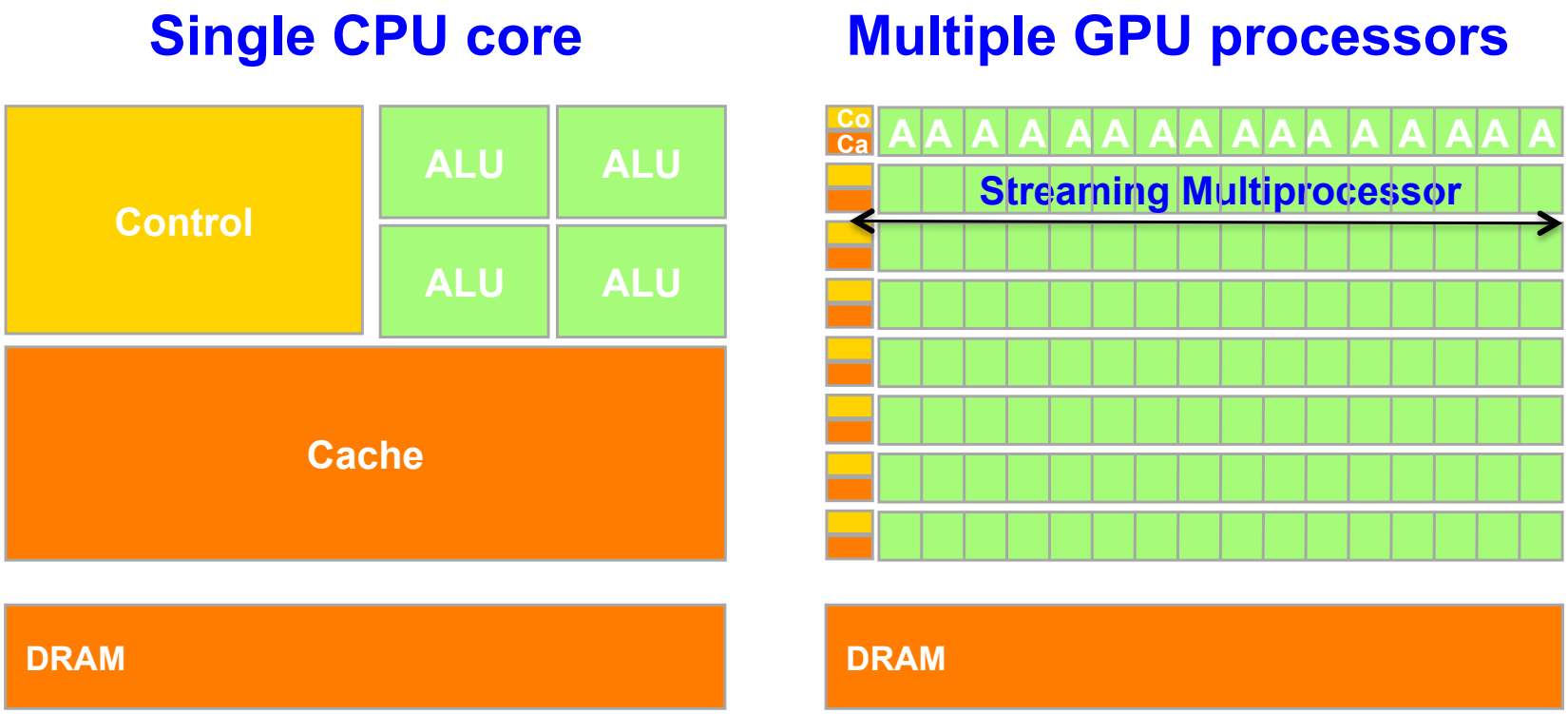
Worksheet #36 (contd)

```
1. class Foo { //unoptimized version
2.     private Helper helper; // Singleton pattern
3.     public synchronized Helper getHelper() {
4.         if (helper == null) {
5.             helper = new Helper();
6.         }
7.         return helper;
8.     }
9.     . . .

10.class Foo { //Optimized version
11.     private Helper helper; // Singleton pattern
12.     public Helper getHelper() {
13.         if (helper == null) {
14.             synchronized(this) {
15.                 if (helper == null) {
16.                     helper = new Helper();
17.                 }
18.             }
19.         }
20.         return helper;
21.     }
22.     . . .
```

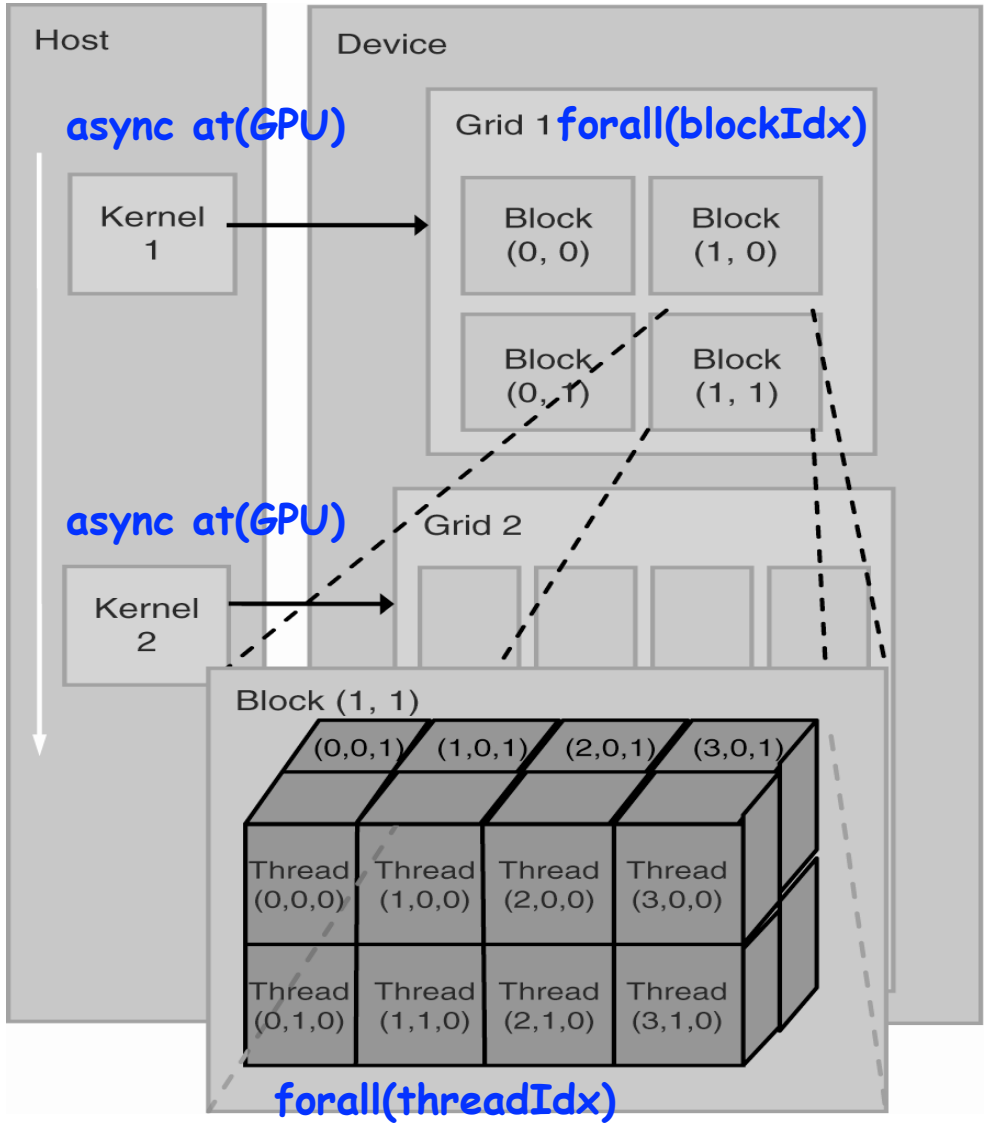

CPUs and GPUs have fundamentally different design philosophies (Lecture 37)

GPU = Graphics Processing Unit



GPUs are provided to accelerate graphics, but they can also be used for non-graphics applications that exhibit large amounts of data parallelism and require large amounts of "streaming" throughput
⇒ SIMD parallelism within an SM, and SPMD parallelism across SMs

HJ abstraction of a CUDA kernel invocation: async-at-gpu + block-forall + thread-forall



Worksheet #37: Branching in SIMD code

Consider SIMD execution of the following pseudocode with 8 threads. Assume that each call to doWork(x) takes x units of time, and ignore all other costs. How long will this program take when executed on 8 GPU cores, taking into consideration the branching issues discussed in Slide 13?

```
1. int tx = threadIdx.x; // ranges from 0 to 7
2. if (tx % 2 = 0) {
3.   S1: doWork(1); // Computation S1 takes 1 unit of time
4. }
5. else {
6.   S2: doWork(2); // Computation S2 takes 2 units of time
7. }
```

Solution: 3 units of time (WORK=12, CPL=3)

How did COMP 322 work out this semester?

- What worked (relatively) well
  - Course software: Java 8, HJlib, HJ-viz, Abstract Metrics
  - Worksheets, labs, videos, quizzes, lecture handouts
  - Piazza, in-class demonstrations (but more are needed)
- What did not work so well
  - Performance complexities for Java on STIC
  - Grading delays
- Help us improve COMP 322 in the future
  - Send us your suggestions for improvement
  - Serve as a TA next year
  - Sign up to work on improving course material and software

Announcements (Recap)

- Graded midterms can be picked up from Bel Martinez in Duncan Hall room 3122 (bellem@rice.edu)
- Homework 5 due by 11:55pm on April 24th, penalty-free extension till May 1st
  - Slip days can be applied past May 1st
- Exam 2 is a scheduled final exam to be held during 9am - 12noon on Tuesday, May 5th, in Hertzstein Amphitheatre
  - Final exam will cover material from Lectures 20 - 37

COMP 322 A01	20754	Scheduled	Tue, May 05, 2015	9:00am - Noon	Vivek Sarkar	HRZ AMP
COMP 322 A02	24087	^Scheduled	Tue, May 05, 2015	9:00am - Noon	Vivek Sarkar	HRZ AMP

- Today is the last lecture!

Acknowledgments

- Co-instructor
  - Eric Allen
- Graduate TAs
  - Prasanth Chatarasi, Peng Du,
  - Xian Fan, Max Grossman
- Undergraduate TAs
  - Matthew Bernhard, Nicholas Hanson-Holtry,
  - Yi Hua, Yoko Li, Ayush Narayan, Derek Peirce,
  - Maggie Tang, Wei Zeng, Glenn Zhu
- HJlib consultant
  - Shams Imam
- Administrative Staff
  - Bel Martinez

ion is
vives
at has
been learned
has been
forgotten"
B.F. Skinner