COMP 322: Fundamentals of Parallel Programming

Lecture 39: Course Review

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

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

Places in HJ (Lectures 17, 18)

here = place at which current task is executing

place.MAX_PLACES = total number of places (runtime constant)

Specified by value of `p` in runtime option, `-places p:w`

place.factory.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

async at(P) S

- Creates new task to execute statement S at place P
- **async S** is equivalent to **async at(here) S**
- Main program task starts at **place.factory.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

Distributions --- hj.lang.dist

- A distribution maps points in a rectangular index space (region) to places e.g.,
  - i → place.factory.place(i % place.MAX_PLACES-1)
- Programmers are free to create any data structure they choose to store and compute these mappings
- For convenience, the HJ language provides a predefined type, hj.lang.dist, to simplify working with distributions
- Some public members available in an instance d of hj.lang.dist are:
  - d.rank = number of dimensions in the input region for distribution d
  - d.get(p) = place for point p mapped by distribution d. It is an error to call d.get(p) if p.rank != d.rank.
  - d.places() = set of places in the range of distribution d
  - d.restrictToRegion(pl) = region of points mapped to place pl by distribution d

Block Distribution

- dist.factory.block([lo:hi]) creates a block distribution over the one-dimensional region, lo:hi.
- A block distribution splits the region into contiguous subregions, one per place, while trying to keep the subregions as close to equal in size as possible.
- Block distributions can improve the performance of parallel loops that exhibit spatial locality across contiguous iterations.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Place id	0				1				2				3			

Cyclic Distribution

- dist.factory.cyclic([lo:hi]) creates a cyclic distribution over the one-dimensional region, lo:hi.
- A cyclic distribution "cycles" through places 0 … place.MAX PLACES – 1 when spanning the input region
- Cyclic distributions can improve the performance of parallel loops that exhibit load imbalance
- Example in Table 3: dist.factory.cyclic([0:15]) for 4 places

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Place id	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

- Example in Table 4: dist.factory.cyclic([0:7,0:1]) for 4 places

Index	[0,0]	[0,1]	[1,0]	[1,1]	[2,0]	[2,1]	[3,0]	[3,1]	[4,0]	[4,1]	[5,0]	[5,1]	[6,0]	[6,1]	[7,0]	[7,1]
Place id	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

Homework 5: Solution to Problem 1a

```
dist d = dist.factory.block([1:N]);
  for (point [iter] : [0:M-1]) {
    finish for(int j=1; j<=N; j++)
      async at(d[j]) {
        myNew[j] = (myVal[j-1] + myVal[j+1]) / 2.0;
      } //finish-for-async-at
    double[] temp = myNew; myNew = myVal; myVal = temp;
  } // for
```

Number of remote reads for block distribution ~ 2*M*P

Number of remote reads for cyclic distribution ~ 2*M*N

HJ isolated statement (Lectures 20, 21, 37)

isolated <body>

- Two tasks executing isolated statements with interfering accesses must perform the isolated statement in mutual exclusion
  - Two instances of isolated statements, ⟨stmt1⟩ and ⟨stmt2⟩, are said to interfere with each other if both access a shared location, such that at least one of the accesses is a write.
  - ➔Weak isolation guarantee: no mutual exclusion applies to non-isolated statements i.e., to (isolated, non-isolated) and (non-isolated, non-isolated) pairs of statement instances
- Isolated statements may be nested (redundant)
- Isolated statements must not contain any other parallel statement that performs a blocking operation: finish, get, next
  - Non-blocking operations (e.g., async) are fine

Object-based isolation in HJ

`isolated(<object-list>) <body>`

- In this case, programmer specifies list of objects for which isolation is required
- Mutual exclusion is only guaranteed for instances of isolated statements that have a non-empty intersection in their object lists
  - Standard isolated is equivalent to isolated(*) by default i.e., isolation across all objects
- Implementation can choose to distinguish between read/write accesses for further parallelism
  - Current HJ implementation supports object-based isolation, does not exploit read/write distinction

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

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.

Parallel Spanning Tree Algorithm using isolated statement

```
class V  {
  V [] neighbors; // adjacency list for input graph
  AtomicReference parent; // output value of parent in spanning tree
  boolean tryLabeling(V n) {
    isolated if (parent == null) parent=n;
    return parent == n;
  } // tryLabeling
  void compute() {
    for (int i=0; i<neighbors.length; i++) {
      V child = neighbors[i];
      if (child.tryLabeling(this))
          async child.compute(); //escaping async
     }
  } // compute
} // class V
. . .
root.parent = root; // Use self-cycle to identify root
finish root.compute();
. . .
```

Parallel Spanning Tree Algorithm using object-based isolation

```
class V  {
  V [] neighbors; // adjacency list for input graph
  AtomicReference parent; // output value of parent in spanning tree
  boolean tryLabeling(V n) {
    isolated(this) if (parent == null) parent=n;
    return parent == n;
  } // tryLabeling
  void compute() {
    for (int i=0; i<neighbors.length; i++) {
      V child = neighbors[i];
      if (child.tryLabeling(this))
          async child.compute(); //escaping async
     }
  } // compute
} // class V
. . .
root.parent = root; // Use self-cycle to identify root
finish root.compute();
. . .
```

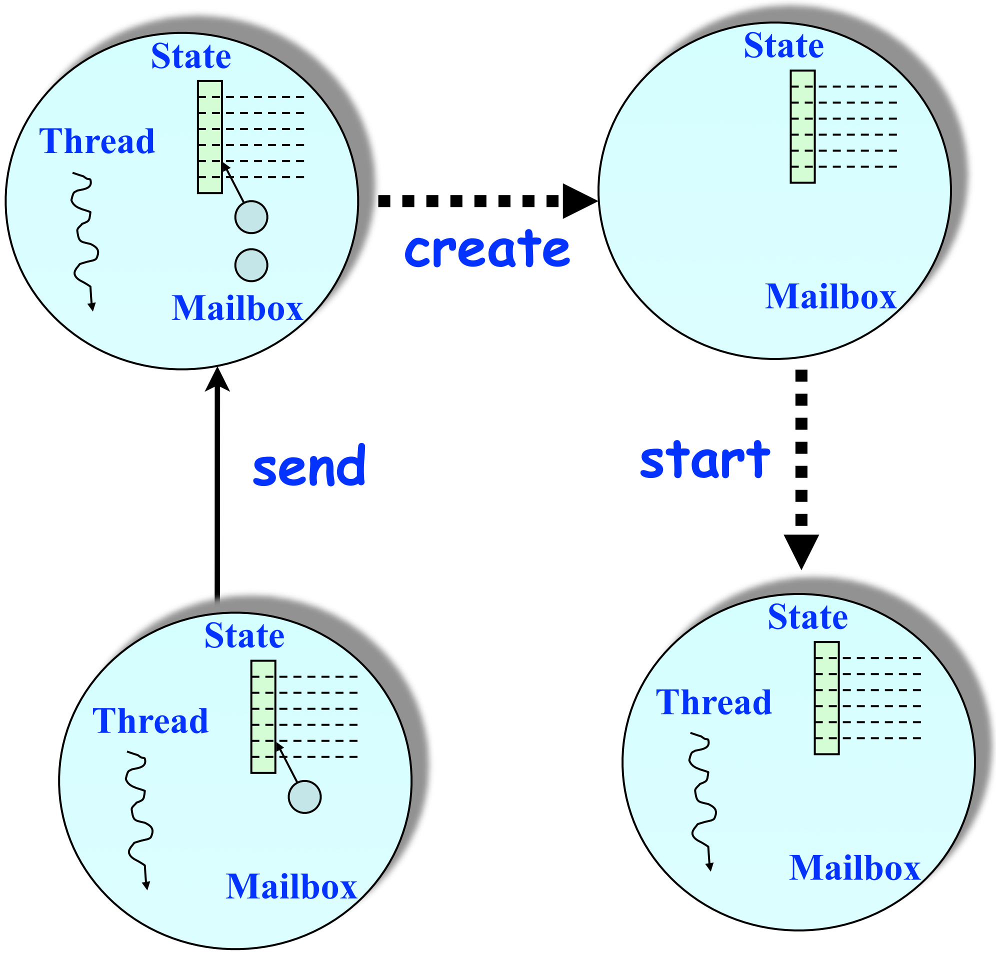
Parallel Spanning Tree Algorithm using java.util.concurrent.atomic.AtomicReference

```
class V  {
  V [] neighbors; // adjacency list for input graph
  AtomicReference parent; // output value of parent in spanning tree
  boolean tryLabeling(V n) {
    return parent.compareAndSet(null ,n);

  } // tryLabeling
  void compute() {
    for (int i=0; i<neighbors.length; i++) {
      V child = neighbors[i];
      if (child.tryLabeling(this))
          async child.compute(); //escaping async
     }
  } // compute
} // class V
. . .
root.parent = root; // Use self-cycle to identify root
finish root.compute();
. . .
```

The Actor Model (Lectures 21, 22, 23)

- An actor may:
  - process messages
  - read/write local state
  - create a new actor
  - start a new actor
  - send messages to other actors
  - terminate
- An actor processes messages sequentially
  - guaranteed mutual exclusion on accesses to local state



Actor Life Cycle

Actor states

- New: Actor has been created
  - e.g., email account has been created
- Started: Actor can receive and process messages
  - e.g., email account has been activated
- Terminated: Actor will no longer processes messages
  - e.g., termination of email account after graduation

Using Actors in HJ

- Create your custom class which extends hj.lang.Actor<Object> ,and implement the void process() method

```
class MyActor extends Actor<Object> {
  protected void process(Object message) {
    System.out.println("Processing " + message);
} }
```

- Instantiate and start your actor

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

- Send messages to the actor

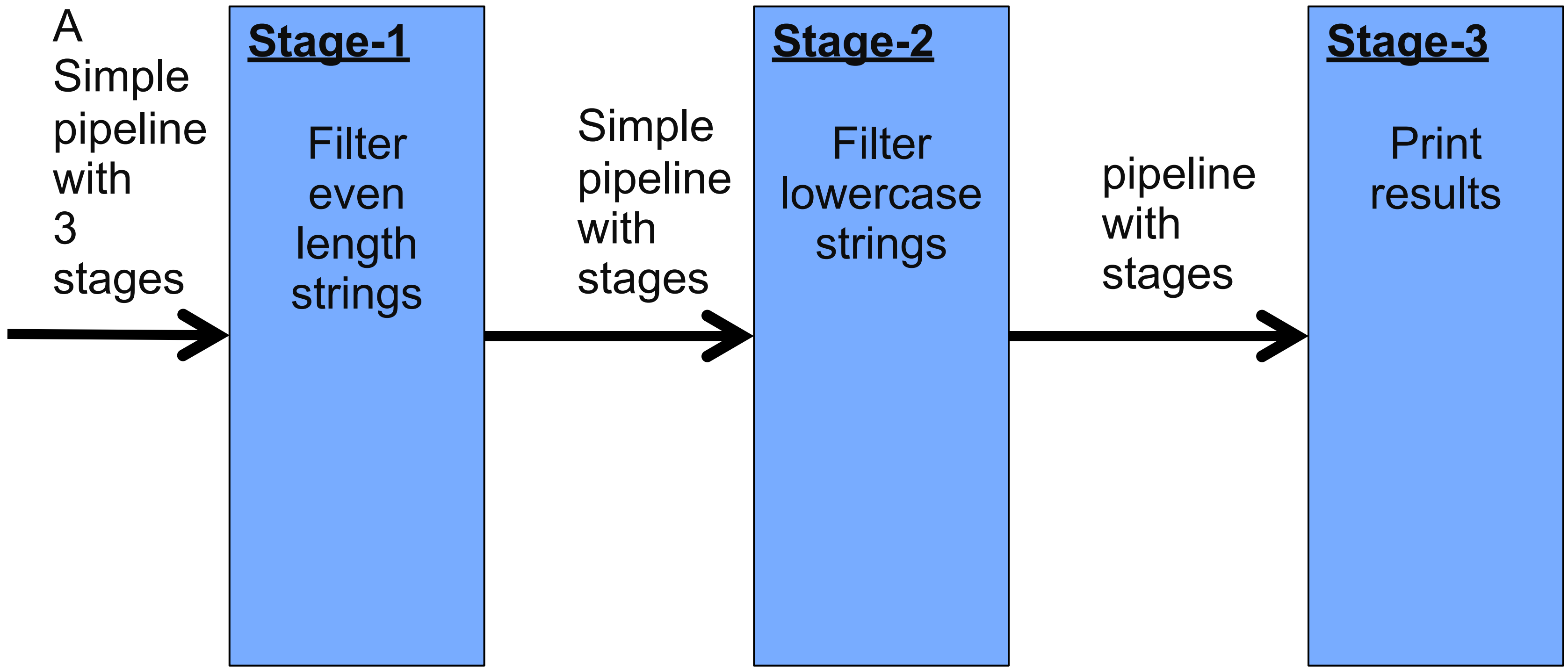
```
anActor.send(aMessage); //aMessage can be any object in general
```

- Use a special message to terminate an actor

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

- Actor execution implemented as async tasks in HJ
  - Can use finish to await their completion

Simple Pipeline



Simple Pipeline using HJ Actors

```
  // Main program
  finish {
    Actor<Object> firstStage =
      new EvenLengthFilter(
        new LowerCaseFilter(
          new LastStage()));
    firstStage.start(); // starts others
    firstStage.send("pipeline");
    firstStage.send(new StopMessage());
  }

class LastStage extends Actor {
  protected void process(Object msg) {
    if (msg instanceof StopMessage) {
      exit();
    } else if (msg instanceof String) {
      System.out.println(msg);
} } }
```

Sends are asynchronous in actor model, but HJ Actor library preserves order of messages between same sender and receiver

Simple Pipeline using HJ Actors (contd)

```
class LowerCaseFilter extends Actor {
  protected void process(Object msg) {
    if (msg instanceof StopMessage) {
      exit(); nextStage.send(msg);
    } else if (msg instanceof String) {
      String str = (String) msg;
      if (str.toLowerCase().equals(str)) {
        nextStage.send(str);
} } } }
class EvenLengthFilter extends Actor {
  protected void process(Object msg) {
    if (msg instanceof StopMessage) {
      nextStage.send(msg);
      exit();
    } else if (msg instanceof String) {
      String msgStr = (String) msg;
      if (msgStr.length() % 2 == 0) {
        nextStage.send(msgStr);
} } } }
```

Adding support for places in HJ actors

- Basic approach: include an optional place parameter in the start() method

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

 anActor.start(p);    // Start actor at place p
```

- Example:

```
SievePlaceActor nextActor = new SievePlaceActor(...);

// Start actor at next place, relative to current place
nextActor.start(here.next());
```

Summary of Mutual Exclusion approaches in HJ

- Isolated --- analogous to critical sections
- Object-based isolation, isolated(a, b, …)
  - Single object in list --- like monitor operations on object
  - Multiple objects in list --- deadlock-free mutual exclusion on sets of objects
- Java atomic variables --- optimized implementation of object-based isolation
- Java concurrent collections --- optimized implementation of monitors
- Actors --- different paradigm from task parallelism (mutual exclusion by default)

Linearizability of Concurrent Objects (Lectures 23, 24)

Concurrent object

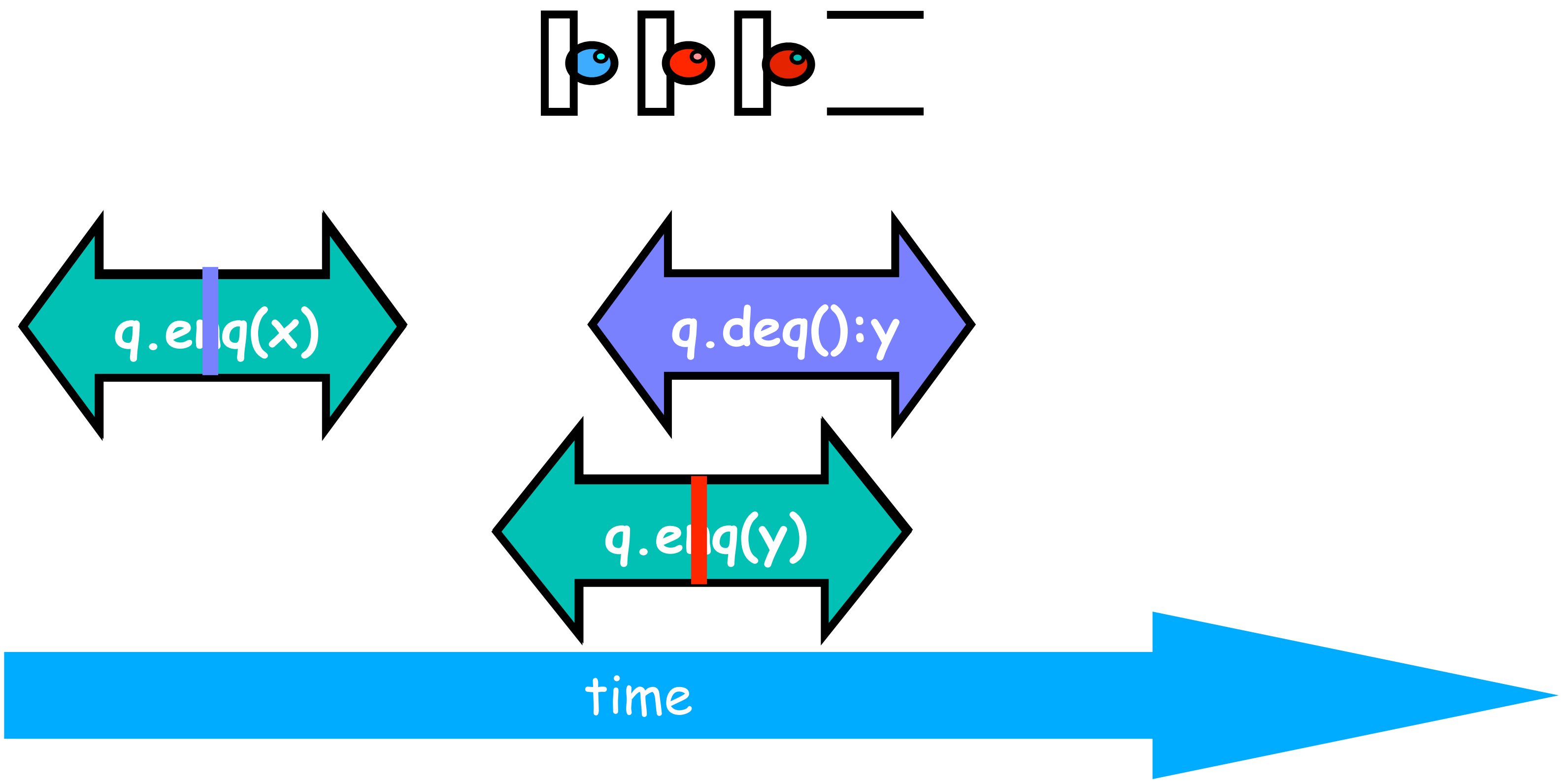
- A concurrent object is an object that can correctly handle methods invoked in parallel by different tasks or threads
  - Examples: concurrent queue, AtomicInteger

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

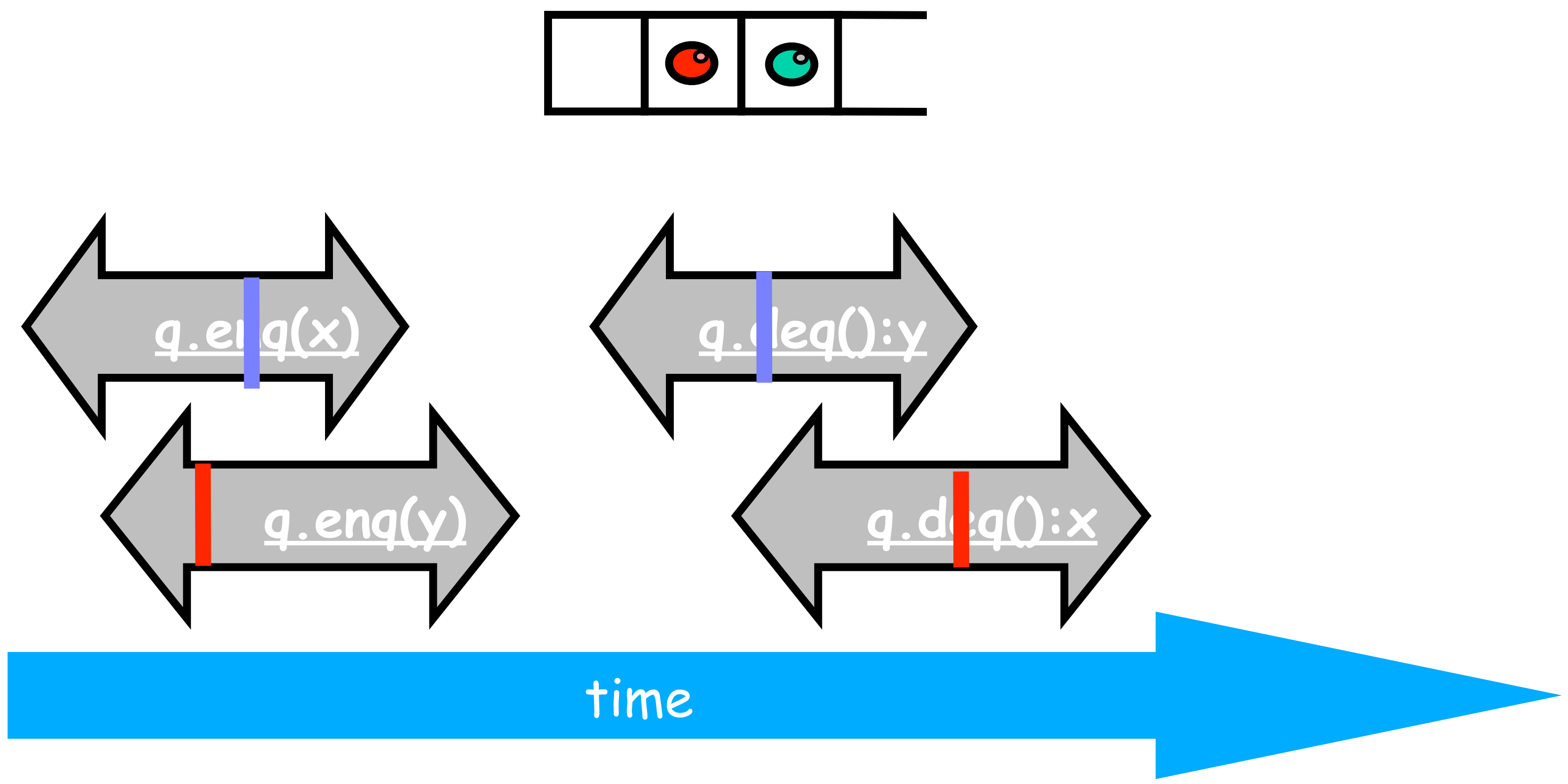
Is this execution linearizable?



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

Example 2

Is this execution linearizable?



Homework 5: Solution to Problem 2b

```
     import java.util.concurrent.atomic.*;
1.class IQueue {
2.  AtomicInteger head = new AtomicInteger(0);
3.  AtomicInteger tail = new AtomicInteger(0);
4.  Object[] items = new Object[Integer.MAX_VALUE];
5.  public void enq(Object x) {
6.    int slot ;
7.    // Loop till enqueue slot is found
8.    do slot = tail.get();
9.    while (!tail.compareAndSet(slot,slot +1));
10.    items[slot] = x;
11.  } // enq
12.  public Object deq() throws EmptyException {
13.    Object value; int slot;
14.    // Loop till dequeue slot is found
15.    do {
16.      slot = head.get(); value = items[slot];
17.      if (value == null) throw new EmptyException();
18.    } while (!head.compareAndSet(slot,slot+1));
19.    return value;
20.  } // deq
21.} // Iqueue
```

Not linearizable. Consider { async enq(A); enq(B); deq(); }

Assume that enq(A) pauses between lines 9 and 10

Safety vs. Liveness (Lecture 25)

- 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 correct
  - Liveness: the conditions under which it guarantees progress
- Linearizability is a safety property for concurrent objects

Desirable Properties of Parallel Program Executions

- Data-race freedom
- Termination
  - But some applications are designed to be non-terminating
- Liveness = a program's ability to make progress in a timely manner
- Different levels of liveness guarantees (from weaker to stronger)
  - Deadlock freedom
  - Livelock freedom
  - Starvation freedom
- Today's lecture discusses progress guarantees for HJ programs
  - We will revisit progress guarantees for Java concurrency later

Deadlock-Free Parallel Program Executions

- A parallel program execution is deadlock-free if no task's execution remains incomplete due to it being blocked awaiting some condition
- Example of a program with a deadlocking execution

```
DataDrivenFuture left = new DataDrivenFuture();
DataDrivenFuture right = new DataDrivenFuture();
finish {
  async await ( left ) right.put(rightBuilder()); // Task1
  async await ( right ) left.put(leftBuilder()); // Task2
}
```

- In this case, Task1 and Task2 are in a deadlock cycle.
  - Only two constructs can lead to deadlock in HJ: async await or explicit phaser wait (instead of next)
  - There are many mechanisms that can lead to deadlock cycles in other programming models (e.g., locks)

Livelock-Free Parallel Program Executions

- A parallel program execution exhibits livelock if two or more tasks repeat the same interactions without making any progress (special case of nontermination)
- Livelock example:

```
// Task 1
incrToTwo(AtomicInteger ai) {
  // increment ai till it reaches 2
  while (ai.incrementAndGet() < 2);
}
```

```
// Task 2
decrToNegativeTwo(AtomicInteger ai) {
  // decrement ai till it reaches -2
  while (a.decrementAndGet() > -2);
}
```

- Many well-intended approaches to avoid deadlock result in livelock instead
- Any data-race-free HJ program without isolated/atomic-variables/ actors is guaranteed to be livelock-free (may be nonterminating in a single task, however)

Starvation-Free Parallel Program Executions

- A parallel program execution exhibits starvation if some task is repeatedly denied the opportunity to make progress
  - Starvation-freedom is sometimes referred to as "lock-out freedom"
  - Starvation is possible in HJ programs, since all tasks in the same program are assumed to be cooperating, rather than competing
    - If starvation occurs in a deadlock-free HJ program, the "equivalent" sequential program must have been non-terminating
- Classic source of starvation: "Priority Inversion" problem for OS threads (usually from different processes)
  - Thread A is at high priority, waiting for result or resource from Thread C at low priority
  - Thread B at intermediate priority is CPU-bound
  - Thread C never runs, hence thread A never runs
  - Fix: when a high priority thread waits for a low priority thread, boost the priority of the low-priority thread

Selecting the Right Pattern (Lecture 25)
(adapted from page 9, Parallel Programming w/ Microsoft .Net)

Application characteristics	Algorithmic pattern	Relevant HJ constructs
Sequential loop with independent iterations	1) Parallel Loop	forall, forasync
Independent operations with well-defined control flow	2) Parallel Task	async, finish
Aggregating data from independent tasks/iterations	3) Parallel Aggregation (reductions)	finish accumulators, atomic variables
Ordering of steps based on data flow constraints	4) Futures	futures, data-driven tasks
Divide-and-conquer algorithms with recursive data structures	5) Dynamic Task Parallelism	async, finish
Repetitive operations on data streams	6) Pipelines	streaming phasers (deterministic), actors (non-deterministic)

Supporting Patterns

1) **Master-worker**

— A process or thread (the master) sets up a task queue and manages other threads (the workers) as they grab a task from the queue, carry out the computation, and then return for their next task. This continues until the master detects that a termination condition has been met, at which point the master ends the computation.

2) **Single Instruction Multiple Data (SIMD)**

— A supporting pattern for data parallelism, in which a single instruction stream is applied to multiple data elements in parallel.

3) **Single Program Multiple Data (SPMD)**

— Multiple copies of a single program are launched typically with their own view of the data. The path through the program for each copy is determined in part based on a unique ID (a rank).

3) SPMD Supporting Pattern

- SPMD: Single Program Multiple Data
- Run the same program on P processing elements (PEs)
- Use the "rank" … an ID ranging from 0 to (P-1) … to determine what computation is performed on what data by a given PE
- Different PEs can follow different paths through the same code (unlike the SIMD pattern)
- Convenient pattern for hardware platforms that are not amenable to efficient forms of dynamic task parallelism
  - General-Purpose Graphics Processing Units (GPGPUs)
  - Distributed-memory parallel machines
- Key design decisions --- what data and computation should be replicated or partitioned across PEs?

SPMD Example #2: Iterative Averaging Example (Slide 9, Lecture 13)

```
double[] gVal=new double[n+2]; double[] gNew=new double[n+2];
gVal[n+1] = 1; // Boundary condition
int Cj = Runtime.getNumOfWorkers();
forall (point [jj]:[0:Cj-1]) { // SPMD computation
  double[] myVal = gVal; double[] myNew = gNew; // Local copy
  for (point [iter] : [0:numIters-1]) {
    // Compute MyNew as function of input array MyVal
    for (point [j]:getChunk([1:n],[Cj],[jj]))
      myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
    next; // Barrier before executing next iteration of iter loop
    // Swap myVal and myNew (replicated computation)
    double[] temp=myVal; myVal=myNew; myNew=temp;
    // myNew becomes input array for next iter
  } // for
} // forall
```

java.lang.Thread class (Lecture 27)

- **Execution of a Java program begins with an instance of Thread created by the Java Virtual Machine (JVM) that executes the program's main() method.**
- **Parallelism can be introduced by creating additional instances of class Thread that execute as parallel threads.**

```java
public class Thread extends Object implements Runnable {
  Thread() { ... } // Creates a new Thread
  Thread(Runnable r) { ... } // Creates a new Thread with Runnable object r
  void  run() { ... } // Code to be executed by thread
   // Case 1: If this thread was created using a Runnable object,
   //         then that object's run method is called
   // Case 2: If this class is subclassed, then the run() method
   //         in the subclass is called
  void start() { ... } // Causes this thread to start execution
  void join() { ... } // Wait for this thread to die
  void  join(long m) // Wait at most m milliseconds for thread to die
  static Thread currentThread() // Returns currently executing thread
  . . .
}
```

Listing 3: java.lang.Thread class

Listing 4: Two-way Parallel ArraySum using Java threads

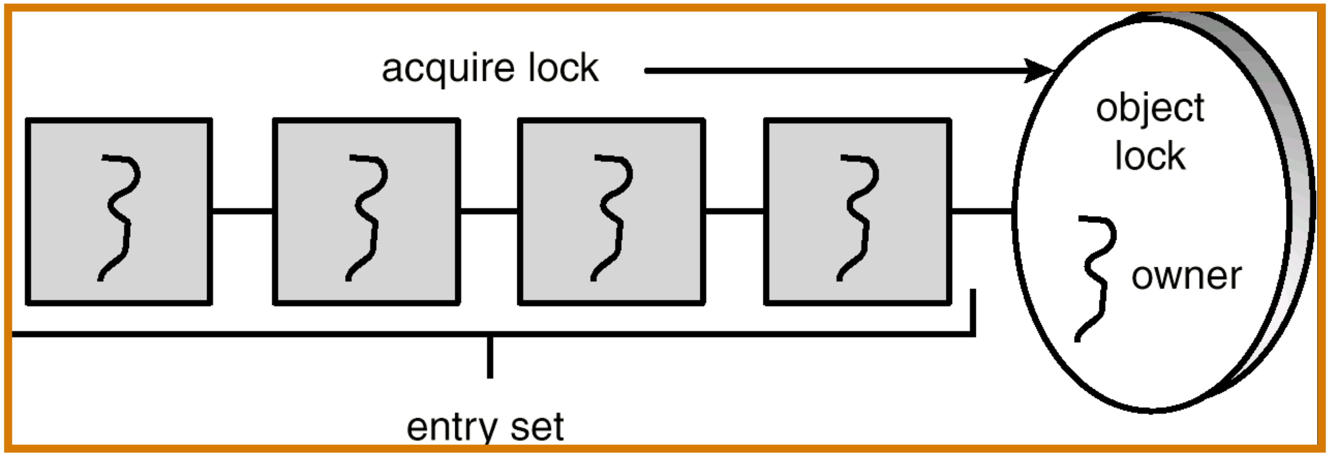
```
// Start of Task T1 (main program)
sum1 = 0; sum2 = 0; // Assume that sum1 & sum2 are fields (not local vars)
// Compute sum1 (lower half) and sum2 (upper half) in parallel
final int len = X.length;
Runnable r1 = new Runnable() {
  public void run(){ for(int i=0 ; i < len/2 ; i++) sum1 += X[i];}
};
Thread t1 = new Thread(r1);
t1.start();
Runnable r2 = new Runnable() {
  public void run(){ for(int i=len/2 ; i < len ; i++) sum2 += X[i];}
};
Thread t2 = new Thread(r2);
t2.start();
// Wait for threads t1 and t2 to complete
t1.join(); t2.join();
int sum = sum1 + sum2;
```

Objects and Locks in Java --- synchronized statements and methods (Lecture 29)

- 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
- Java's synchronized is related to "mutex" locks in POSIX thread library

Implementation of Java synchronized statements/methods

- 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 own the lock (because another thread already owns it), it is placed in an unordered "entry set" for the object's lock



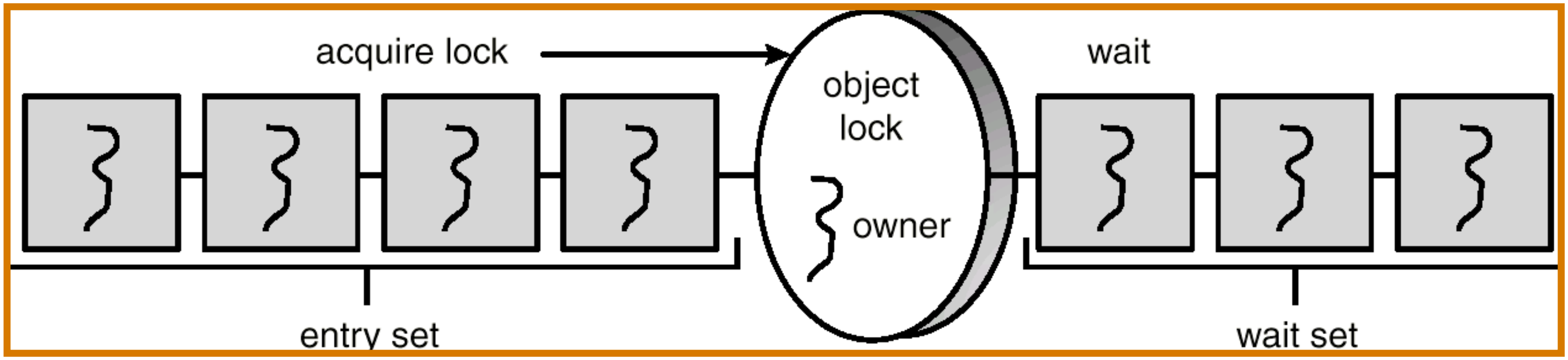
The Java wait() Method

- A thread can perform a wait() method on an object that it owns:
  1. the thread releases the object lock
  2. thread state is set to blocked
  3. thread is placed in the wait set
- Causes thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
- Since interrupts and spurious wake-ups are possible, this method should always be used in a loop e.g.,

```
synchronized (obj) {
    while (<condition does not hold>)
        obj.wait();
    ... // Perform action appropriate to condition
}
```

- Java's wait-notify is related to "condition variables" in POSIX threads

Entry and Wait Sets



The notify() Method

When a thread calls notify(), the following occurs:

1. selects an arbitrary thread T from the wait set
2. moves T to the entry set
3. sets T to Runnable

T can now compete for the object's lock again

java.util.concurrent.locks.Lock interface (Lecture 30)

```
interface Lock {
   void lock();
   void lockInterruptibly() throws InterruptedException;
   boolean tryLock();
   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

Simple ReentrantLock() example

- Used extensively within `java.util.concurrent`

```
final Lock lock = new ReentrantLock();
 ...
 lock.lock();
 try {
   // perform operations protected by lock
 }
 catch(Exception ex) {
   // restore invariants & rethrow
 }
 finally {
   lock.unlock();
 }
```

- **Must manually ensure lock is released**

Reading vs. writing

- Recall that the use of synchronization is to protect interfering accesses
  - Multiple concurrent reads of same memory: Not a problem
  - Multiple concurrent writes of same memory: Problem
  - Multiple concurrent read & write of same memory: Problem

So far:

- If concurrent write/write or read/write might occur, use synchronization to ensure one-thread-at-a-time

But:

- This is unnecessarily conservative: we could still allow multiple simultaneous readers

Consider a hashtable with one coarse-grained lock

- So only one thread can perform operations at a time

But suppose:

- There are many simultaneous `lookup` operations
- `insert` operations are very rare

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

Our First MPI Program (mpiJava version, Lecture 33)

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

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

Example of Send and Recv

```
import mpi.*;

class myProg {
  public static void main( String[] args ) {
    int tag0 = 0;
    MPI.Init( args );                    // Start MPI computation
    if ( MPI.COMM_WORLD.rank() == 0 ) { // rank 0 = sender
      int loop[] = new int[1]; loop[0] = 3;
      MPI.COMM_WORLD.Send( "Hello World!", 0, 12, MPI.CHAR, 1, tag0 );
      MPI.COMM_WORLD.Send( loop, 0, 1, MPI.INT, 1, tag0 );
    } else {                             // rank 1 = receiver
      int loop[] = new int[1]; char msg[] = new char[12];
      MPI.COMM_WORLD.Recv( msg, 0, 12, MPI.CHAR, 0, tag0 );
      MPI.COMM_WORLD.Recv( loop, 0, 1, MPI.INT, 0, tag0 );
      for ( int i = 0; i < loop[0]; i++ ) System.out.println( msg );
    }
    MPI.Finalize( );                     // Finish MPI computation
  }
}
```

Send() and Recv() calls are blocking operations by default

Announcements

- **Homework 6 due due by 11:55pm today**
  - An automatic 7-day penalty-free extension can be used till April 27th
- **Homeworks 4 and 5 will be returned by end of Monday, April 23rd**
- **Exam 2 is a take-home exam**
  - Maximum duration = 2 hours
  - Closed-book, closed-notes, closed-computer
  - Pick up exam from Amanda Nokleby's office (Duncan Hall 3137) any time starting 2pm today
  - Return exam to Amanda's office by 4pm on Friday, April 27th
  - Written exam --- no penalty for minor syntactic errors in program text, so long as the meaning of the program is unambiguous.
  - If you believe there is any ambiguity or inconsistency in a question, you should state the ambiguity or inconsistency that you see, and any assumptions that you make to resolve it.
  - Scope of exam includes Lectures 17-34, excluding Lecture 19 (midterm review) and Lecture 28 (guest lecture)

Acknowledgments

- Graduate TAs
  - Sanjay Chatterjee
  - Deepak Majeti
  - Dragos Sbirlea
- Undergraduate TAs
  - Max Grossman
  - Damien Stone
  - Yunming Zhang
- Research Programmer
  - Vincent Cave
- Additional HJ expert
  - Shams Imam
- Administrative assistant
  - Amanda Nokleby

Have a great summer!!