COMP 322: Fundamentals of Parallel Programming

Lecture 27: Safety and Liveness Properties, Java Synchronizers, Dining Philosophers Problem

Instructors: Vivek Sarkar, Mack Joyner
Department of Computer Science, Rice University
{vsarkar, mjoyner}@rice.edu

http://comp322.rice.edu/

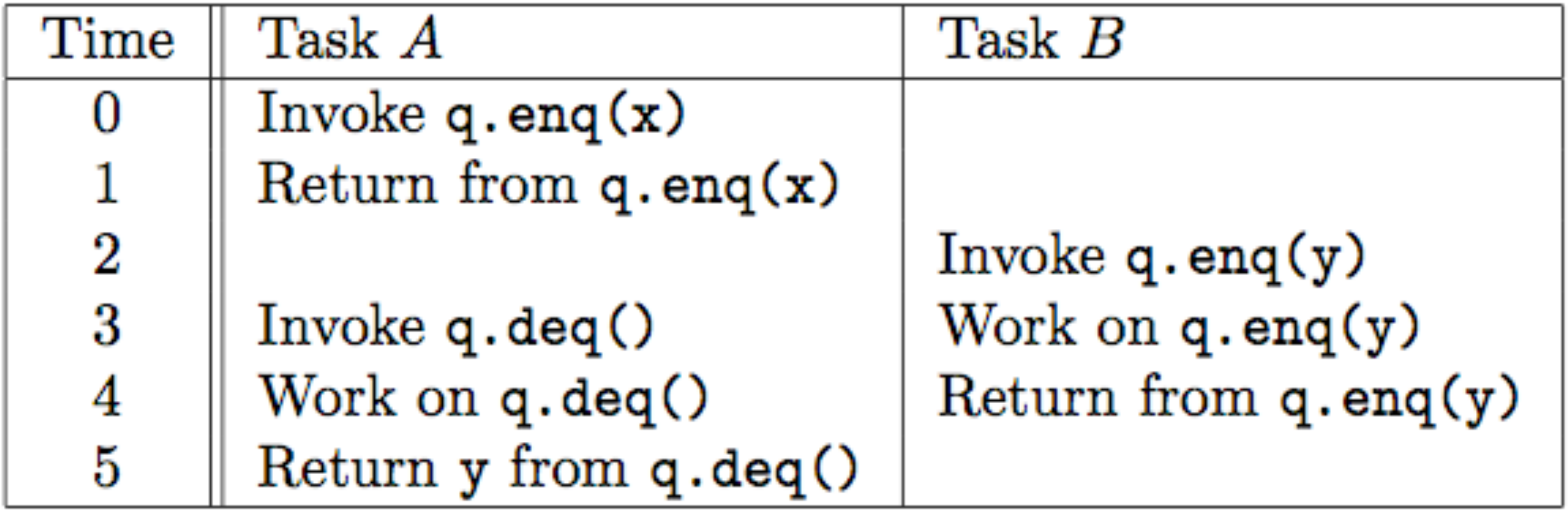
Worksheet #26a solution: use of tryLock()

Rewrite the transferFunds() method below to use j.u.c. locks with calls to tryLock (see slide 8) 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 below using pseudocode.

```
public void transferFunds(Account from, Account to, int amount) {
  while (true) {
    // assume that trylock() does not throw an exception
    boolean fromFlag = from.lock.trylock();
    if (!fromFlag) continue;
    boolean toFlag = to.lock.trylock();
    if (!toFlag) { from.lock.unlock(); continue; }
    try { from.subtractFromBalance(amount);
          to.addToBalance(amount); break; }
     finally { from.lock.unlock(); to.lock.unlock(); }
   } // while
}
```

Worksheet #26b solution: 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.

Outline

- <u>Safety and Liveness</u>
- Java Synchronizers
- Dining Philosophers Problem

Safety vs. Liveness

- 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)
- Examples of safety
  - Data race freedom is a desirable safety property for parallel programs (Module 1)
  - Linearizability is a desirable safety property for concurrent objects (Module 2)

Liveness

- Liveness = a program's ability to make progress in a timely manner
- Termination ("no infinite loop") is not necessarily a requirement for liveness
  - some applications are designed to be non-terminating
- Different levels of liveness guarantees (from weaker to stronger) for tasks/threads in a concurrent program
  1. Deadlock freedom
  2. Livelock freedom
  3. Starvation freedom
  4. Bounded wait

1. 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

```
// Thread T1
public void leftHand() {
  synchronized(obj1) {
    synchronized(obj2) {
      // work with obj1 & obj2
      . . .
    }
  }
}
```

```
// Thread T2
public void leftHand() {
  synchronized(obj2) {
    synchronized(obj1) {
      // work with obj2 & obj1
      . . .
    }
  }
}
```

- In this case, Task1 and Task2 are in a deadlock cycle.
  - Three constructs that can lead to deadlock in HJlib: async await, finish w/ actors, explicit phaser wait (instead of next)
  - There are many constructs that can lead to deadlock cycles in other programming models (e.g., thread join, synchronized, locks in Java)

2. 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 T1
incrToTwo(AtomicInteger ai) {
  // increment ai till it reaches 2
  while (ai.incrementAndGet() < 2);
}
```

```
// Task T2
decrToNegTwo(AtomicInteger ai) {
  // decrement ai till it reaches -2
  while (a.decrementAndGet() > -2);
}
```

- Many well-intended approaches to avoid deadlock result in livelock instead
- Any HJlib program that uses only Module 1 features, and is data-race-free, is guaranteed to be livelock-free (may be nonterminating in a single task, however)

3. 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 be non-terminating (infinite loop)
- Classic source of starvation for OS threads: "Priority Inversion"
  - 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 (because its priority is lower than B's priority), 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

Related Concept: Progress Conditions for shared resources

- A resource is said to be *obstruction*-free if it is deadlock-free
- A resource is said to be *lock-free* if it is livelock-free and deadlock-free
- A resource is said to be *wait-free* if it is starvation-free, livelock-free, and deadlock-free
  - Wait-free $\Rightarrow$ every thread/task will eventually get an opportunity to make progress, i.e., to access the shared resource
  - Question: how to bound the wait duration?

4. Bounded Wait

- A parallel program execution exhibits bounded wait if each task requesting a resource should only have to wait for a bounded number of other tasks to “cut in line” i.e., to gain access to the resource after its request has been registered.
- If bound = 0, then the program execution is fair

Outline

- **Safety and Liveness**
- **Java Synchronizers**
- **Dining Philosophers Problem**

Key Functional Groups in java.util.concurrent (j.u.c.)

- Atomic variables
  - The key to writing lock-free algorithms
- Concurrent Collections:
  - Queues, blocking queues, concurrent hash map, …
  - Data structures designed for concurrent environments
- Locks and Conditions
  - More flexible synchronization control
  - Read/write locks
- Executors, Thread pools and Futures
  - Execution frameworks for asynchronous tasking
- Synchronizers: Semaphore, Latch, Barrier, Exchanger
  - Ready made tools for thread coordination

j.u.c Synchronizers --- common patterns in HJ's phaser construct

- Class library includes several state-dependent synchronizer classes
  - `CountDownLatch` – waits until latch reaches terminal state
  - `Semaphore` – waits until permit is available
  - `CyclicBarrier` – like barriers in HJlib forall loops
  - `Phaser` – inspired by Habanero phasers
  - `FutureTask` – like futures in HJlib
  - `Exchanger` – waits until two threads rendezvous (special synchronization)
- These typically have three main groups of methods
  - Methods that block until the object has reached the right state
    - Timed versions will fail if the timeout expired
    - Many versions can be cancelled via interruption
  - Polling methods that allow non-blocking interactions
  - State change methods that may release a blocked method

— WARNING: synchronizers should only be used in Java threads, not HJlib tasks, since they can cause the HJlib runtime system to deadlock

`CountDownLatch`

- A counter that releases waiting threads when it reaches zero
  - —Allows one or more threads to wait for one or more events
  - —Initial value of 1 gives a simple gate or latch

`CountDownLatch(int initialValue)`

- `await()`: wait until the counter is zero
  - —await() is what differentiates a CountDownLatch from an AtomicInteger
- `countDown()`: decrement the counter if > 0
- Query: `getCount()`
- Very simple but widely useful
  - Replaces error-prone attempts with data races

Example: using j.u.c.CountDownLatch to implement finish for Java threads

- Problem: Run N tasks concurrently in N threads and wait until all are complete
  - Use a `CountDownLatch` initialized to the number of threads

```
public static void runTask(int numThreads, final Runnable task)
        throws InterruptedException {
  final CountDownLatch done = new CountDownLatch(numThreads);
  for (int i=0; i<numThreads; i++) {
      Thread t = new Thread() {
          public void run() {
              try {
               task.run();
              }
              finally { done.countDown();}
          }};
      t.start();
  }
  done.await();   // wait for all threads to finish
}
```

Old-fashioned way of specifying lambdas in Java!

Semaphores

- Conceptually serve as “permit” holders
  - Construct with an initial number of permits
  - `acquire():` waits for permit to be available, then “takes” one, i.e., decrements the count of available permits
  - `release():` “returns” a permit, i.e., increments the count of available permits
  - But no actual permits change hands
    - The semaphore just maintains the current count
    - Thread performing release() can be different from the thread performing acquire()
- “fair” variant hands out permits in FIFO order
- Useful for managing bounded access to a shared resource

Bounded Blocking Concurrent List using Semaphores

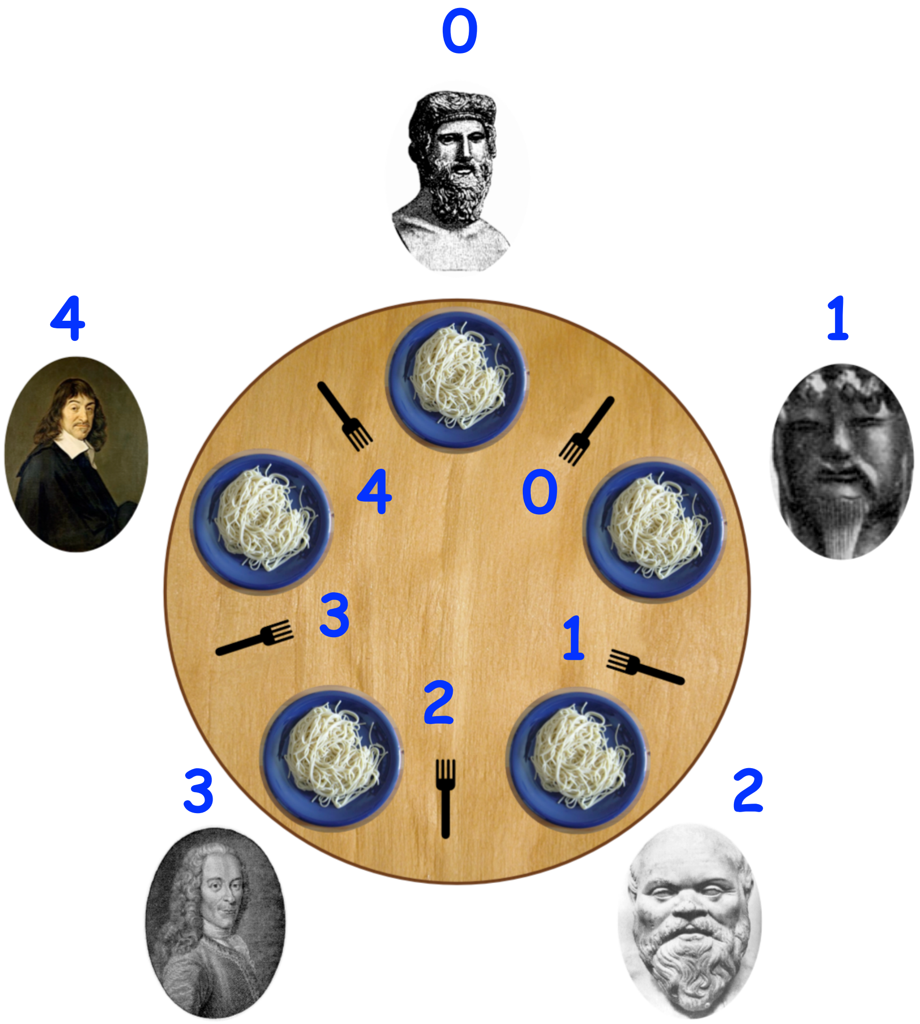
```
public class BoundedBlockingList {
 final int capacity;
 final ConcurrentLinkedList list = new ConcurrentLinkedList();
 final Semaphore sem;
 public BoundedBlockingList(int capacity) {
  this.capacity = capacity;
  sem = new Semaphore(capacity);
}
public void addFirst(Object x) throws InterruptedException {
   sem.acquire(); // blocks until a permit is available
   try { list.addFirst(x); }
   catch (Throwable t){ sem.release(); rethrow(t); } // only performed on exception
 }
 public boolean remove(Object x) {
   if (list.remove(x)) {  sem.release(); return true; }
   return false;
 }
 … } // BoundedBlockingList
```

Outline

- Safety and Liveness
- Java Synchronizers
- Dining Philosophers Problem
  - Acknowledgments
    - CMSC 330 course notes, U. Maryland

      http://www.cs.umd.edu/~lam/cmsc330/summer2008/lectures/class20-threads_classicprobs.ppt
    - Dave Johnson (COMP 421 instructor)

The Dining Philosophers Problem



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)

General Structure of Dining Philosophers Problem: PseudoCode

```
int numPhilosophers = 5;
int numForks = numPhilosophers;
Fork[] fork = ... ; // Initialize array of forks
forall(point [p] : [0:numPhilosophers-1]) {
  while(true) {
    Think ;
    Acquire forks;
      // Left fork = fork[p]
      // Right fork = fork[(p-1)%numForks]
    Eat ;
  } // while
} // forall
```

Solution 1: using Java’s synchronized statement

```
int numPhilosophers = 5;
int numForks = numPhilosophers;
Fork[] fork = ... ; // Initialize array of forks
forall(point [p] : [0:numPhilosophers-1]) {
  while(true) {
    Think ;
    synchronized(fork[p])
      synchronized(fork[(p-1)%numForks]) {
        Eat ;
      }
    }
  } // while
} // forall
```

Solution 2: using Java’s Lock library

```
int numPhilosophers = 5;
int numForks = numPhilosophers;
Fork[] fork = ... ; // Initialize array of forks
forall(point [p] : [0:numPhilosophers-1]) {
  while(true) {
    Think ;
    if (!fork[p].lock.tryLock()) continue;
    if (!fork[(p-1)%numForks].lock.tryLock()) {
      fork[p].lock.unLock(); continue;
    }
    Eat ;
    fork[p].lock.unlock();fork[(p-1)%numForks].lock.unlock();
  } // while
} // forall
```

Solution 3: using HJ’s isolated statement

```
int numPhilosophers = 5;
int numForks = numPhilosophers;
Fork[] fork = ... ; // Initialize array of forks
forall(point [p] : [0:numPhilosophers-1]) {
  while(true) {
    Think ;
    isolated {
      Pick up left and right forks;
      Eat ;
    }
  } // while
} // forall
```

Solution 4: using HJ’s object-based isolation

```
int numPhilosophers = 5;
int numForks = numPhilosophers;
Fork[] fork = ... ; // Initialize array of forks
forall(point [p] : [0:numPhilosophers-1]) {
  while(true) {
    Think ;
    isolated(fork[p], fork[(p-1)%numForks]) {
      Eat ;
    }
  } // while
} // forall
```

Solution 5: using Java's Semaphores

```
int numPhilosophers = 5;
int numForks = numPhilosophers;
Fork[] fork = ... ; // Initialize array of forks
Semaphore table = new Semaphore(4, true);
for (i=0;i<numForks;i++) fork[i].sem = new Semaphore(1, true);
forall(point [p] : [0:numPhilosophers-1]) {
  while(true) {
    Think ;
    table.acquire(); // At most 4 philosophers at table
    fork[p].sem.acquire(); // Acquire left fork
    fork[(p-1)%numForks].sem.acquire(); // Acquire right fork
    Eat ;
    fork[p].sem.release(); fork[(p-1)%numForks].sem.release();
    table.release();
  } // while
} // forall
```

"true" parameter creates a semaphore that guarantees fairness