COMP 322: Fundamentals of Parallel Programming

Lecture 30: Java Synchronizers, Dining Philosophers Problem

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

Contact email: vsarkar@rice.edu

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

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 terminating parallel program execution exhibits none of a), b), or c).

Starvation vs. Bounded Wait

- Starvation: A parallel program execution exhibits starvation if some task is repeatedly denied the opportunity to make progress

- 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.

  ⇒ Unbounded Wait is the same as Starvation, in practice

- Many implementations of critical sections exhibit Starvation

Outline

- Java Synchronizers

- Dining Philosophers Problem

Key Functional Groups in java.util.concurrent

- 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` – waits until N threads rendezvous
  - `Phaser` – extension of CyclicBarrier with dynamic parallelism
  - `Exchanger` – waits until 2 threads rendezvous
  - `FutureTask` – waits until a computation has completed
- 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

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 (if needed) until the counter is zero
  - Timeout version returns false on timeout
- `countDown`: decrement the counter if > 0
- Query: `getCount()`
- Very simple but widely useful:
  - Replaces error-prone constructions ensuring that a group of threads all wait for a common signal

Example: using j.u.c.CountDownLatch to implement finish

- 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
  - `release`: "returns" a permit
  - But no actual permits change hands
    - The semaphore just maintains the current count
    - No need to acquire a permit before you release it
- "fair" variant hands out permits in FIFO order
- Supports balking and timed versions of `acquire`
- Applications:
  - Resource controllers
  - Designs that otherwise encounter missed signals
    - Semaphores 'remember' how often they were signalled

Bounded Blocking Concurrent List Example

- Concurrent list with fixed capacity
  - Insertion blocks until space is available
- Tracking free space, or available items, can be done using a Semaphore
- Demonstrates composition of data structures with library synchronizers
  - Easier than modifying implementation of concurrent list directly

Bounded Blocking Concurrent List

```
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();
   try { list.addFirst(x); }
   catch (Throwable t){ sem.release(); rethrow(t); }
 }
 public boolean remove(Object x) {
   if (list.remove(x)) {  sem.release(); return true; }
   return false;
 }
 … } // BoundedBlockingList
```

Summary of j.u.c. libraries

- Atomics: java.util.concurrent.atomic
  - Atomic[Type]
  - Atomic[Type]Array
  - Atomic[Type]FieldUpdater
  - Atomic{Markable,Stampable}Reference
- Concurrent Collections
  - ConcurrentMap
  - ConcurrentHashMap
  - CopyOnWriteArray{List,Set}
- Locks: java.util.concurrent.locks
  - Lock
  - Condition
  - ReadWriteLock
  - AbstractQueuedSynchronizer
  - LockSupport
  - ReentrantLock
  - ReentrantReadWriteLock
- *Executors*
  - *ExecutorService*
  - *ScheduledExecutorService*
  - *Callable*
  - *Future*
  - *ScheduledFuture*
  - *Delayed*
  - *CompletionService*
  - *ThreadPoolExecutor*
  - *ScheduledThreadPoolExecutor*
  - *AbstractExecutorService*
  - *FutureTask*
  - *ExecutorCompletionService*
- Synchronizers
  - CountDownLatch
  - Semaphore
  - Exchanger
  - CyclicBarrier

Executors are the only class that we haven't studied as yet

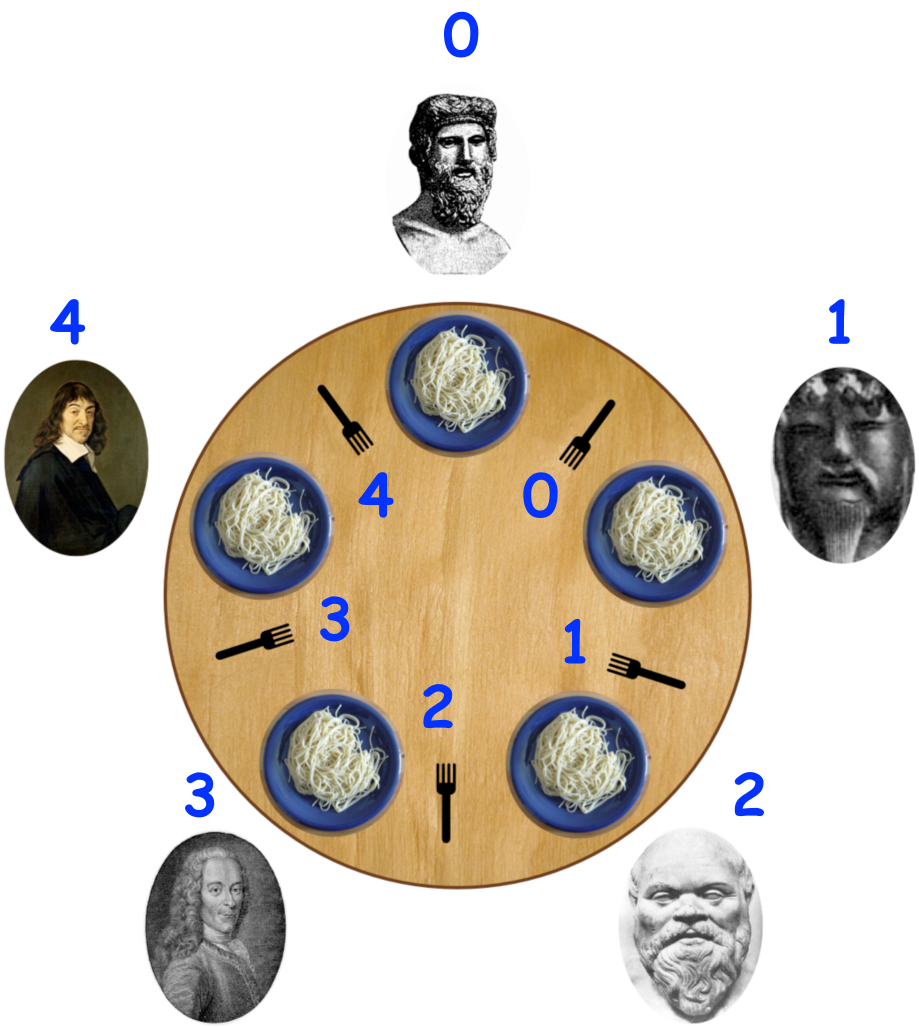
Outline

- **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

```
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

```
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); // assume semaphores are fair
for (i=0;i<numForks;i++) fork[i].sem = new Semaphore(1);
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
```