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
/** 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.

  \[\Rightarrow\] 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

  \[ \text{CountDownLatch}(\text{int } \text{initialValue}) \]

- \text{await}: wait (if needed) until the counter is zero
  - Timeout version returns false on timeout

- \text{countDown}: decrement the counter if > 0

- Query: \text{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

```java
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

1. public class BoundedBlockingList {
2.   final int capacity;
3.   final ConcurrentLinkedLIst list = new ConcurrentLinkedLIst();
4.   final Semaphore sem;
5.   public BoundedBlockingList(int capacity) {
6.     this.capacity = capacity;
7.     sem = new Semaphore(capacity);
8.   }
9.   public void addFirst(Object x) throws InterruptedException {
10.      sem.acquire();
11.      try { list.addFirst(x); }
12.      catch (Throwable t) { sem.release(); rethrow(t); }
13.   }
14.   public boolean remove(Object x) {
15.      if (list.remove(x)) { sem.release(); return true; }
16.      return false;
17.   }
18.   ... } // 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**
  - 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
Outline

• Java Synchronizers

• Dining Philosophers Problem
  — Acknowledgments
    – CMSC 330 course notes, U. Maryland
    – 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

1. int numPhilosophers = 5;
2. int numForks = numPhilosophers;
3. Fork[] fork = ... ; // Initialize array of forks
4. forall(point [p] : [0:numPhilosophers-1]) {
5.   while(true) {
6.     Think ;
7.     Acquire forks;
8.     // Left fork = fork[p]
9.     // Right fork = fork[(p-1)%numForks]
10.    Eat ;
11.   } // while
12.} // forall
Solution 1: using Java’s synchronized

1. int numPhilosophers = 5;
2. int numForks = numPhilosophers;
3. Fork[] fork = ... ; // Initialize array of forks
4. forall(point [p] : [0:numPhilosophers-1]) {
5.   while(true) {
6.     Think ;
7.     synchronized(fork[p])
8.     synchronized(fork[(p-1)%numForks]) {
9.       Eat ;
10.     }
11.   }
12. } // while
13. } // forall
Solution 2: using Java’s Lock library

1. int numPhilosophers = 5;
2. int numForks = numPhilosophers;
3. Fork[] fork = ... ; // Initialize array of forks
4. forall(point [p] : [0:numPhilosophers-1]) {
5.     while(true) {
6.         Think ;
7.         if (!fork[p].lock.tryLock()) continue;
8.         if (!fork[(p-1)%numForks].lock.tryLock()) {
9.             fork[p].lock.unlock(); continue;
10.         }
11.     }
12.     Eat ;
13.     fork[p].lock.unlock();fork[(p-1)%numForks].lock.unlock();
14. } // while
15. } // forall
Solution 3: using HJ's isolated

1. int numPhilosophers = 5;
2. int numForks = numPhilosophers;
3. Fork[] fork = ... ; // Initialize array of forks
4. forall(point [p] : [0:numPhilosophers-1]) {
5.   while(true) {
6.     Think ;
7.     isolated {
8.       Pick up left and right forks;
9.       Eat ;
10.     }
11.   } // while
12.} // forall
Solution 4: using HJ’s object-based isolation

1. int numPhilosophers = 5;
2. int numForks = numPhilosophers;
3. Fork[] fork = ... ; // Initialize array of forks
4. forall(point [p] : [0:numPhilosophers-1]) {
   5.   while(true) {
      6.      Think;
      7.      isolated(fork[p], fork[(p-1)%numForks]) {
      8.         Eat;
      9.      }
   10. } // while
11. } // forall
Solution 5: using Java’s Semaphores

1. int numPhilosophers = 5;
2. int numForks = numPhilosophers;
3. Fork[] fork = ... ; // Initialize array of forks
4. Semaphore table = new Semaphore(4); // assume semaphores are fair
5. for (i=0;i<numForks;i++) fork[i].sem = new Semaphore(1);
6. forall(point [p] : [0:numPhilosophers-1]) {
7.     while(true) {
8.         Think ;
9.         table.acquire(); // At most 4 philosophers at table
10.        fork[p].sem.acquire(); // Acquire left fork
11.        fork[(p-1)%numForks].sem.acquire(); // Acquire right fork
12.        Eat ;
13.        fork[p].sem.release(); fork[(p-1)%numForks].sem.release();
14.        table.release();
15.     } // while
16. } // forall