COMP 322: Fundamentals of Parallel Programming

Lecture 30: Java Synchronizers, Dining Philosophers Problem

Vivek Sarkar, Shams Imam
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu, shams.imam@twosigma.com

http://comp322.rice.edu/
Worksheet #29: Analyzing Parallelism in an Actor Pipeline

Consider a three-stage pipeline of actors (as in slide 5), 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

Input sequence

\[ ... d_9 d_8 d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0 \]

1. protected void process(final Object msg) {
2.     if (msg == null) {
3.         exit(); //actor will exit after returning from process()
4.     } else {
5.         doWork(1); // unit work
6.     }
7.     if (nextStage != null) {
8.         nextStage.send(msg);
9.     }
10. } // process()
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** – 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
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

```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
}
```
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**  
  - `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`

**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
      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)
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.   } // while
12.   Eat ;
13.   fork[p].lock.unlock();fork[(p-1)%numForks].lock.unlock();
14. } // 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
Course Reminders

• Homework 4, Checkpoint 1 is due by 12noon on Monday, April 4th

• Location of final exam is confirmed: Herzstein Hall Auditorium

• Data and time for final exam unchanged from what was announced earlier: Tuesday, 3-MAY-2016, 9:00AM - 12:00PM

• Have a great recess!
For the five solutions studied in today’s lecture, indicate in the table below which of the following conditions are possible and why:

1. Deadlock: when all philosopher tasks are blocked (neither thinking nor eating)
2. Livelock: when all philosopher tasks are executing 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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Possible</th>
<th>Why</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadlock</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Livelock</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Starvation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-Concurrency</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Deadlock</td>
<td>Livelock</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>Solution 1:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>synchronized</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Solution 2:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tryLock/unLock</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Solution 3:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>isolated</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Solution 4:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>object-based isolation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Solution 5:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>semaphores</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>