COMP 322: Fundamentals of Parallel Programming

Lecture 29: Dining Philosophers problem

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

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
Worksheet #28 solution: Relating j.u.c. libraries to HJ constructs

For each functional group of j.u.c. libraries included below, indicate one of the following choices: a) can be used in HJ programs, b) can be substituted by equivalent HJ constructs in some cases (give examples), c) cannot be substituted by equivalent HJ constructs in some cases (give examples).

1. Atomic variables
   a), b) Can be used freely in HJ programs. A method on atomic variable V is equivalent to (but more efficient than) an “isolated (V)” statement

2. Concurrent Collections
   a) Nonblocking APIs can be used freely e.g., ConcurrentHashMap
   c) Blocking APIs cannot be used in HJ programs

3. Locks
   b) Can be substituted by isolated or actor constructs in HJ in some cases
   c) Cannot be used in HJ programs in general

4. Executors
   b) Can be substituted by HJ async tasks in most cases (except when priority/timeout is needed)
   c) Cannot be used in HJ programs in general

5. Synchronizers
   b) Can be substituted by finish, futures, phasers in many cases
   c) Cannot be used in HJ programs in general
Summary: Relating j.u.c. libraries to HJ constructs

- **Atomics:** java.util.concurrent.atomic
  
  Can be used as is in HJ programs

- **Concurrent Collections**
  
  Can be used as is in HJ programs

- **Locks:** java.util.concurrent.locks
  
  Many uses of j.u.c.locks & synchronized can be replaced by HJ isolated

- **Synchronizers**
  
  Many uses can be replaced by finish, phasers, and data-driven futures

- **Executors**
  
  Many uses can be replaced by async, finish, futures, forall

- **Queues**
  
  Do not use BlockingQueue in HJ programs, and take care to avoid infinite loops on retrieval operations on non-blocking queues
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 (don’t ask why)
- Can only use forks on either side of their plate
- No talking permitted

Goals
- Progress guarantees
  - Deadlock freedom
  - Livelock freedom
  - Starvation freedom
  - Bounded wait
- Maximize concurrency when eating
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 statement

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 statement

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);
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
Worksheet #29: Characterizing Solutions to the Dining Philosophers Problem

Name 1: ___________________          Name 2: ___________________

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

• No lab quiz this week because of midterm recess
  —You are still encouraged to do this week’s lab and submit your work using turnin

• HW5 will be posted soon