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

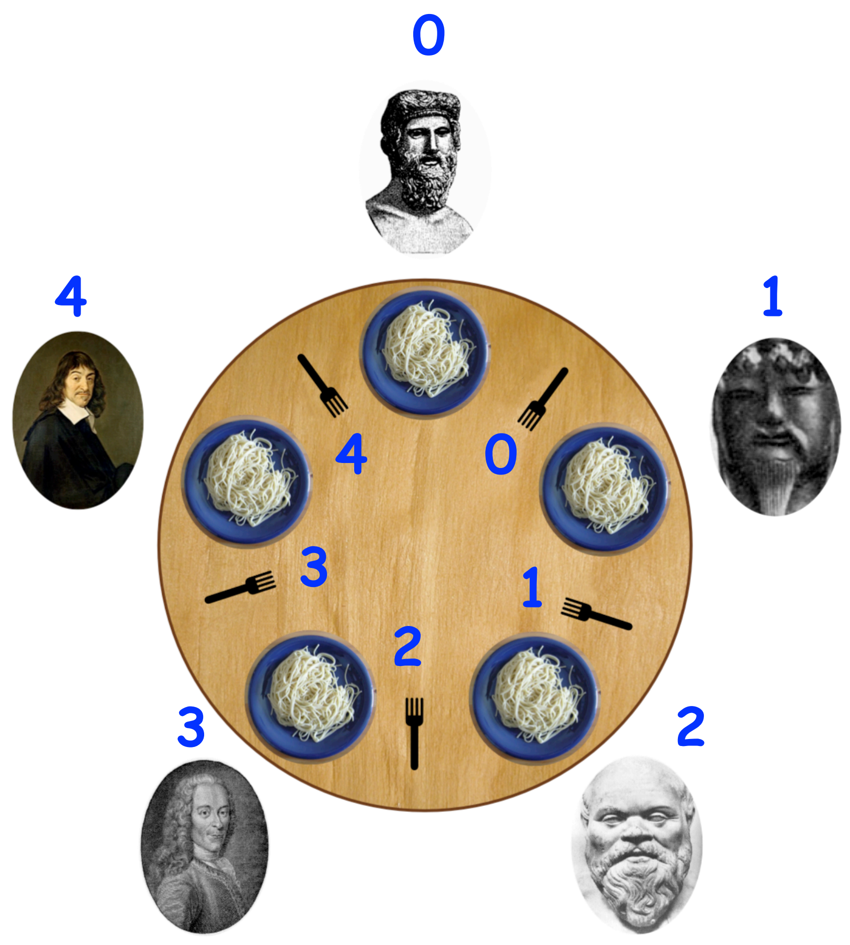
- Queues

  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
  - 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 (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

```
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);
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
```

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

	Deadlock	Livelock	Starvation	Non-concurrency
Solution 1: synchronized				
Solution 2: tryLock/ unLock				
Solution 3: isolated				
Solution 4: object-based isolation				
Solution 5: semaphores				

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