

# COMP 322: Parallel and Concurrent Programming

## Lecture 28: Dining Philosophers

Mack Joyner  
mjoyner@rice.edu

<http://comp322.rice.edu>

Acknowledgments: CMSC 330 U. Maryland, CS 444 (Clarkson), Dave Johnson (COMP 421), Ken Birman (Cornell)



# Liveness Recap

- **Deadlock**: task's execution remains incomplete due to it being blocked awaiting some condition
- **Livelock**: two or more tasks repeat the same interactions without making any progress
- **Starvation**: some task is repeatedly denied the opportunity to make progress
- **Bounded wait (fairness)**: each task requesting a resource should only have to wait for a bounded number of other tasks to “cut in line”
- **Non-concurrency**: a task is prevented from making progress due to overly restrictive resource management

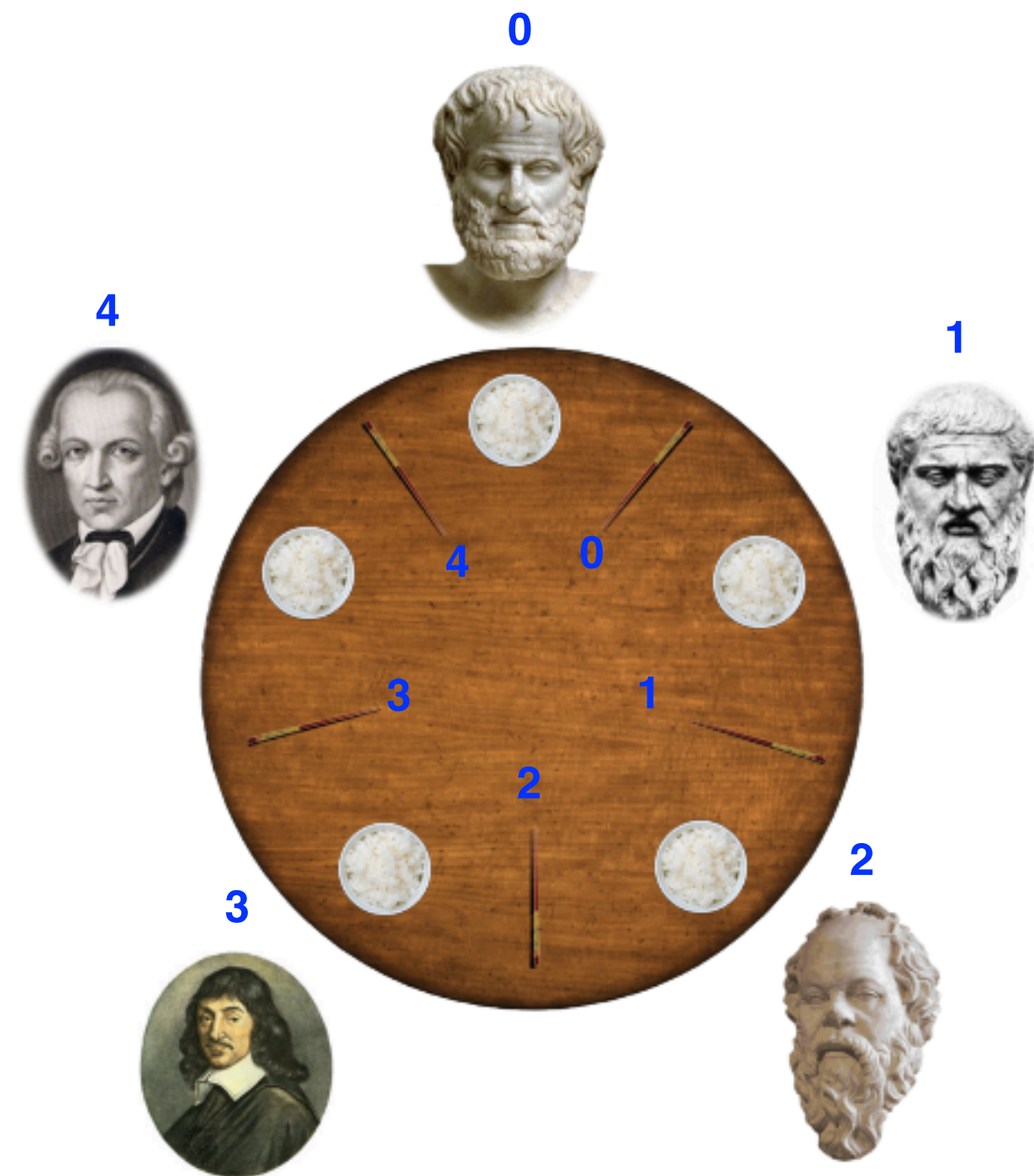


# Deadlock Conditions

- Mutual Exclusion
  - At least one resource that must be held is in non-shareable mode
- Hold and wait
  - There exists a task holding a resource, and waiting for another
- No preemption
  - Resources cannot be preempted
- Circular wait
  - There exists a set of tasks  $\{T_1, T_2, \dots, T_N\}$ , such that
    - $T_1$  is waiting for  $T_2$ ,  $T_2$  for  $T_3$ , .... and  $T_N$  for  $T_1$
- All four conditions must hold for deadlock to occur



# The Dining Philosophers Problem



A classical Synchronization Problem devised by Dijkstra in 1965

## Constraints

- Five philosophers either eat or think
- They must have two chopsticks to eat
- Can only use chopsticks 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 numChops = numPhilosophers;
3. Chop[] chop = ... ; // Initialize array of chopsticks
4. for(p in 0 .. numPhilosophers-1) {
5.   async() -> {
6.     while(true) {
7.       Think ;
8.       Acquire chopsticks;
9.       // Left chopstick = chop[p]
10.      // Right chopstick = chop[(p-1)%numChops]
11.      Eat ;
12.    } // while
13.  }; // async
14.} // for
```



# Solution 1: Using Java's Synchronized Statement

```
1. int numPhilosophers = 5;
2. int numChops = numPhilosophers;
3. Chop[] chop = ... ; // Initialize array of chopsticks
4. for(p in 0 .. numPhilosophers-1) {
5.   async(() -> {
6.     while(true) {
7.       Think ;
8.       synchronized(chop[p]) { // get the left chopstick
9.         synchronized(chop[(p-1)%numChops]) { // get the right chopstick
10.        Eat ;
11.      }
12.    }
13.  } // while
14. }); // async
15.} // for
```



# Problems?

---

- What if everyone picks up the left chopstick at the same time?
- Deadlock!
- Starvation due to deadlock
- No livelock
- Non-concurrency due to deadlock



# Solution 2: Using Java's tryLock

```
1. int numPhilosophers = 5;
2. int numChops = numPhilosophers;
3. Chop[] chop = ... ; // Initialize array of chopsticks
4. for(p in 0 .. numPhilosophers-1) {
5.   async(() -> {
6.     int first = p; int second = (p - 1) % numChops;
7.     while(true) {
8.       Think ;
9.       if (!chop[first].lock.tryLock()) continue;
10.      if (!chop[second].lock.tryLock()) {
11.        chop[first].lock.unlock(); continue;
12.      }
13.      Eat ;
14.      chop[first].lock.unlock(); chop[second].lock.unlock();
15.    } // while
16.  }); // async
17.} // for
```





# Problems?

---

- Everyone picks up the left chopstick at the same time, tries to pick up the right one, gives up, puts down the left one, and repeat
- Livelock!
- Starvation due to livelock!
- No deadlock
- Non-concurrency due to livelock



# Solution 3: Using Global Isolated

```
1. int numPhilosophers = 5;
2. int numChops = numPhilosophers;
3. Chop[] chop = ... ; // Initialize array of chopsticks
4. for(p in 0 .. numPhilosophers-1) {
5.   async() -> {
6.     while(true) {
7.       Think ;
8.       isolated {
9.         Pick up left and right chopsticks;
10.        Eat ;
11.       }
12.    } // while
13. }); // async
14.} // for
```



# Problems?

---

- No deadlock or livelock possible
- Starvation!
  - No guarantee that a philosopher will ever get to eat, if others are very hungry and “cut in line” all the time.
- Non-concurrency
  - Only one philosopher can eat at any time



# Solution 4: Using Object-Based Isolation

```
1. int numPhilosophers = 5;
2. int numChops = numPhilosophers;
3. Chop[] chop = ... ; // Initialize array of chopsticks
4. for(p in 0 .. numPhilosophers-1) {
5.   async() -> {
6.     while(true) {
7.       Think ;
8.       isolated (chop[p], chop[(p-1)%numChops]){
9.         Eat ;
10.      }
11.    } // while
12.  }); // async
13.} // for
```



# Problems?

---

- No deadlock or livelock possible
- Starvation! No guarantee that a philosopher will ever get to eat, if others are very hungry and “cut in line” all the time.
- Concurrency. If a philosopher is hungry, and his chopsticks are not used for eating, he’ll get to eat



# Solution 5: Using Semaphores

```
1. int numPhilosophers = 5;
2. int numChops = numPhilosophers;
3. Chop[] chop = ... ; // Initialize array of chopsticks
4. Semaphore table = new Semaphore(4, true);
5. for (i=0;i<numChops;i++) chop[i].sem = new Semaphore(1, true);
6. for(p in 0 .. numPhilosophers-1) {
7.   async()-> {
8.     while(true) {
9.       Think ;
10.      table.acquire(); // At most 4 philosophers at table
11.      p = empty place at the table that has nobody on the left
12.      chop[p].sem.acquire(); // Acquire left chopstick
13.      chop[(p-1)%numChops].sem.acquire(); // Acquire right chopstick
14.      Eat ;
15.      chop[p].sem.release(); chop[(p-1)%numChops].sem.release();
16.      table.release();
17.    } // while
18.  }; // async
19.} // for
```

“true” parameter creates a semaphore that guarantees



# Problems?

---

