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)

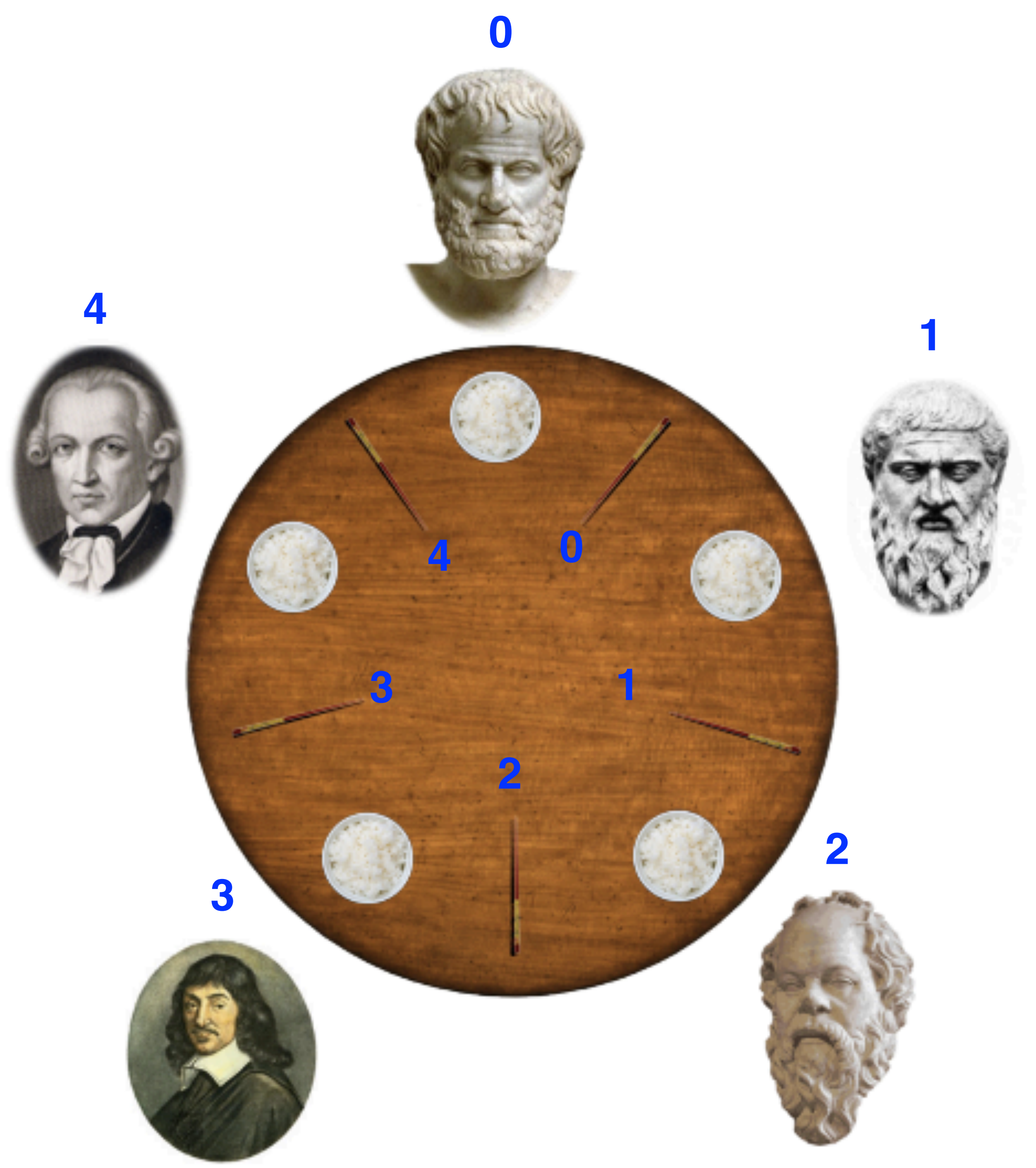
Liveness Recap

- ***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 $\{T_1, T_2, \ldots T_N\}$, 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

```
int numPhilosophers = 5;
int numChops = numPhilosophers;
Chop[] chop = ... ; // Initialize array of chopsticks
for(p in 0 .. numPhilosophers-1) {
  async(() -> {
    while(true) {
      Think ;
      Acquire chopsticks;
      // Left chopstick = chop[p]
      // Right chopstick = chop[(p-1)%numChops]
      Eat ;
    } // while
  }); // async
} // for
```

Solution 1: Using Java’s Synchronized Statement

```
int numPhilosophers = 5;
int numChops = numPhilosophers;
Chop[] chop = ... ; // Initialize array of chopsticks
for(p in 0 .. numPhilosophers-1) {
  async(() -> {
    while(true) {
      Think ;
      synchronized(chop[p]) { // get the left chopstick
        synchronized(chop[(p-1)%numChops]) { // get the right chopstick
          Eat ;
        }
      }
    } // while
  }); // async
} // 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

```
int numPhilosophers = 5;
int numChops = numPhilosophers;
Chop[] chop = ... ; // Initialize array of chopsticks
for(p in 0 .. numPhilosophers-1) {
  async(() -> {
    int first = p; int second = (p - 1) % numChops;
    while(true) {
      Think ;
      if (!chop[first].lock.tryLock()) continue;
      if (!chop[second].lock.tryLock()) {
        chop[first].lock.unLock(); continue;
      }
      Eat ;
      chop[first].lock.unlock();chop[second].lock.unlock();
    } // while
  }); // async
} // 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

```
int numPhilosophers = 5;
int numChops = numPhilosophers;
Chop[] chop = ... ; // Initialize array of chopsticks
for(p in 0 .. numPhilosophers-1) {
  async(() -> {
    while(true) {
      Think ;
      isolated {
        Pick up left and right chopsticks;
        Eat ;
      }
    } // while
  }); // async
} // for
```

Problems?

- No deadlock or lovelock 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

```
int numPhilosophers = 5;
int numChops = numPhilosophers;
Chop[] chop = ... ; // Initialize array of chopsticks
for(p in 0 .. numPhilosophers-1) {
  async(() -> {
    while(true) {
      Think ;
      isolated (chop[p], chop[(p-1)%numChops){
        Eat ;
      }
    } // while
  }); // async
} // 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

```
int numPhilosophers = 5;
int numChops = numPhilosophers;
Chop[] chop = ... ; // Initialize array of chopsticks
Semaphore table = new Semaphore(4, true);
for (i=0;i<numChops;i++) chop[i].sem = new Semaphore(1, true);
for(p in 0 .. numPhilosophers-1) {
  async(() -> {
    while(true) {
      Think ;
      table.acquire(); // At most 4 philosophers at table
      p = empty place at the table that has nobody on the left
      chop[p].sem.acquire(); // Acquire left chopstick
      chop[(p-1)%numChops].sem.acquire(); // Acquire right chopstick
      Eat ;
      chop[p].sem.release(); chop[(p-1)%numChops].sem.release();
      table.release();
    } // while
  }); // async
} // for
```

"true" parameter creates a semaphore that guarantees

Problems?