# COMP 322: Parallel and Concurrent Programming

# Lecture 25: Read/Write Pattern, Dining Philosophers

Zoran Budimlić and Mack Joyner
{zoran, mjoyner}@rice.edu

http://comp322.rice.edu

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

# Motivation for Read-Write Object-based isolation

Sorted List example

```
public boolean contains(Object object) {
    // Observation: multiple calls to contains() should not
    // interfere with each other
    return isolatedWithReturn(this, () -> {
      Entry pred, curr;
      ...
      return (key == curr.key);
   });
}

 public int add(Object object) {
  return isolatedWithReturn(this, () -> {
    Entry pred, curr;
    ...
    if (...) return 1; else return 0;
  });
}
```

# Read-Write Object-Based Isolation

`isolated(readMode(obj1),writeMode(obj2), …, () -> <body> );`

- Programmer specifies list of objects as well as their read-write modes for which isolation is required
- Mutual exclusion is only guaranteed for instances of isolated statements that have a non-empty intersection in their object lists such that one of the accesses is in writeMode
- <u>Sorted List example</u>

```
public boolean contains(Object object) {
  return isolatedWithReturn( readMode(this), () -> {
     Entry pred, curr;
     ...
     return (key == curr.key);
  });
}

 public int add(Object object) {
  return isolatedWithReturn( writeMode(this), () -> {
    Entry pred, curr;
    ...
    if (...) return 1; else return 0;
  });
}
```

# Read-Write Concurrency Pattern

- Common pattern in concurrency
- HJLib Read-Write Object Isolation, Java ReentrantReadWriteLock, C++ Boost UpgradeLockable, sync.RWMutex in Go
- Upgradeable/downgradeable
  - Can upgrade Read access to Write access
    - Could be tricky to implement and avoid deadlock
  - Downgrade Write access to Read access
- Priority policies
  - Read-preferring
    - Max concurrency
    - Could starve writers
  - Write-preferring
    - Less concurrency
    - More overhead

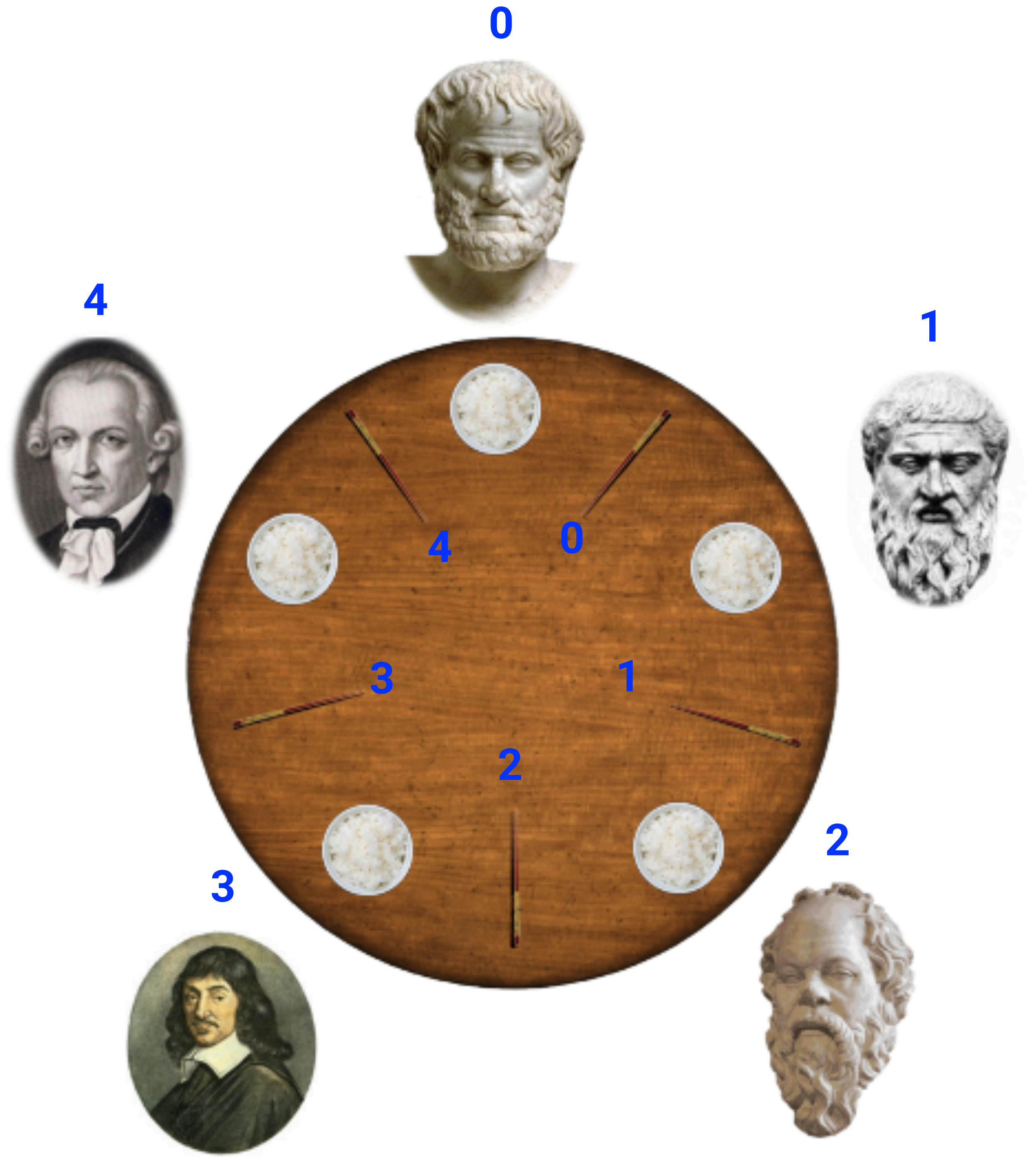
# Liveness Recap

- ***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 4a: Impose Order

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

# Preventing Deadlock by Ordering

It is not possible for all philosophers to have a chopstick

1. Two philosophers, A and B, must share a chopstick, X, that is “smaller” than all other chopsticks
2. One of them, A, has to pick up X first
3. B can’t pick up X at this point
4. B can’t pick up the “bigger” chopstick until X is released
5. SO, 4 philosophers left, 5 chopsticks total
6. One philosopher must be able to have two chopsticks!

# Solution 4b: Using tryLock

```
int numPhilosophers = 5;
int numChops = numPhilosophers;
Chop[] chop = ... ; // Initialize array of chopsticks
for(p in 0 .. numPhilosophers-1) {
  async(() -> {
    int first = (p == 0)? (p - 1) % numChops : p
    int second = (p == 0)? p : (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
```

# Solution 4c: 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 for 4a, 4b and 4c?

- 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.
- Concurrency
  - 4a: still have a non-concurrency problem. If philosopher 0 is eating, philosophers 1-4 could all be holding their left chopstick waiting
  - 4b and 4c: 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(3, 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 3 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 fairness