# CS 10: Problem solving via Object Oriented Programming

Winter 2017

Tim Pierson
260 (255) Sudikoff

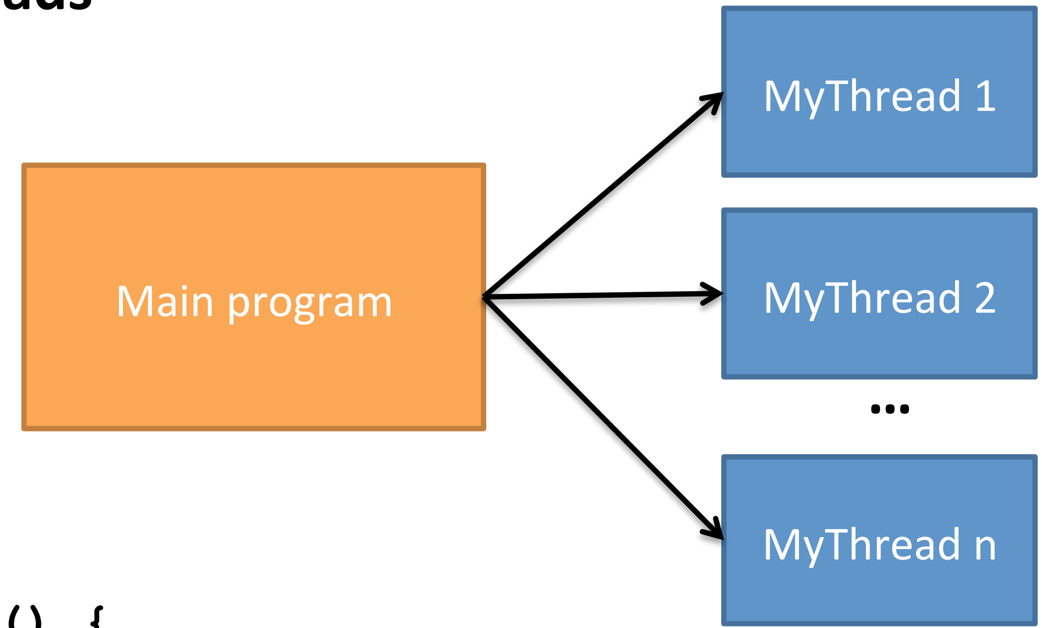
## Synchronization

# Agenda

1. Threads and interleaving execution
2. Producer/consumer
3. Deadlock, starvation

# Threads are a way for multiple processes to run concurrently

**Threads**




```
main() {
MyThreadClass t = new MyThreadClass();

//start thread at run method, main
thread keeps running
t.start()

//halt main until thread finishes
t.join
```

Assume MyThread is a class that extends Thread

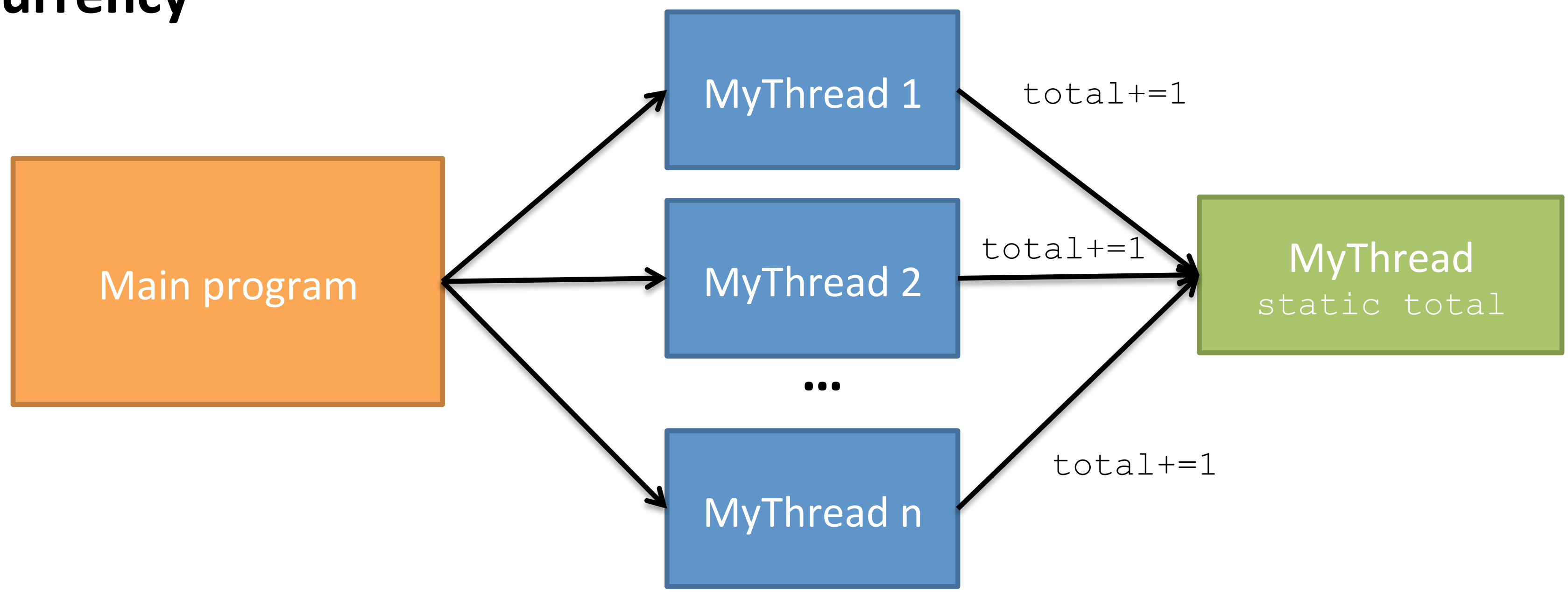
MyThread must a implement a `run` method

Execution begins by calling `start` on a MyThread object, `run` method then executes

Can call `join` to halt main program until thread finishes

# Concurrent threads can access the same resources, this can cause problems

**Concurrency**




- Threads can be interrupted at any time by the Operating System and another thread may be run
- When each thread tries to increment `total`, it gets a current copy of `total`, adds 1, then stores it back in memory
- What can go wrong?

# Threads can be interrupted at any point, this can cause unexpected behavior

**Incrementer.java**

- `total` is static (shared by all of same class)
- Two threads of same `Incrementer` class started
- Main program execution blocked with `joins`
- Each thread increments total 1 million times
- Each thread may be interrupted at any point
- Incrementing `total`
  - Get value of `total` from memory
  - Add 1
  - Store new value back in memory
- Another thread might get value from memory between time when first thread got value and time when first thread wrote new value back
- In that case, the value of `total` will only be incremented by 1 not 2

# Threads can be interrupted at any point, this can cause unexpected behavior

**IncrementerInterleaving.java**

- Almost the same as Incrementer.java
- Each thread now keeps track of its name
- Each thread now prints to console (causing more time for interruptions for other thread)
- Two threads try to increment `total` 5 times
- Sometimes it works, sometimes it doesn't, depends how threads were executed by Operating System
- Causes tricky debugging issues!
- Run several times

# Java provides the keyword synchronized to make some operations "atomic"

**IncrementerTotal.java**

```
public class IncrementerTotal {
    int total = 0;
    public synchronized void inc() {
        total++;
    }
}
```

- `synchronized` keyword in front of `inc` method means only one thread can be running this code at a time
- If multiple threads try to run `synchronized` code, one thread runs, all others are blocked until first one finishes
- Once first thread finishes, another thread is selected to run
- `synchronized` makes this code "atomic" (e.g., as if it were one instruction)
- This synchronized approach is called a "monitor" (or mutex)

# Java provides the keyword synchronized to make some operations “atomic”

**IncrementerTotal.java**

- Class that provides a synchronized method `inc` to ensure only one thread at a time can access `inc` method

**IncrementerSync.java**

- Uses synchronized code to make sure only one thread at a time can update total
- Total is 2 million at completion because threads don’t step on each other

# Agenda

1. Interleaving execution
2. Producer/consumer
3. Deadlock, starvation

# Producers tell Consumers when ready, Consumers tell Producers when done

**Main idea**

**Producer:**

- Tell Consumer when item is ready (`notify` or `notifyAll`)
- Block until woken up by Consumer that item handled (`wait`)
- Tell Consumer when next item is ready (`notify` or `notifyAll`)

**Consumer:**

- Block until woken up by Producer that item ready (`wait`)
- Process item and tell Producer when done (`notify` or `notifyAll`)
- Block until woken up by Producer (`wait`)

# Producers and Consumers synchronized with `wait`, `notify` or `notifyAll`

**`wait`**

- Removes thread from synchronized method
- Tells Operating System to put this thread into a list of waiting threads
- Allows another thread to enter synchronized method

**`notify`**

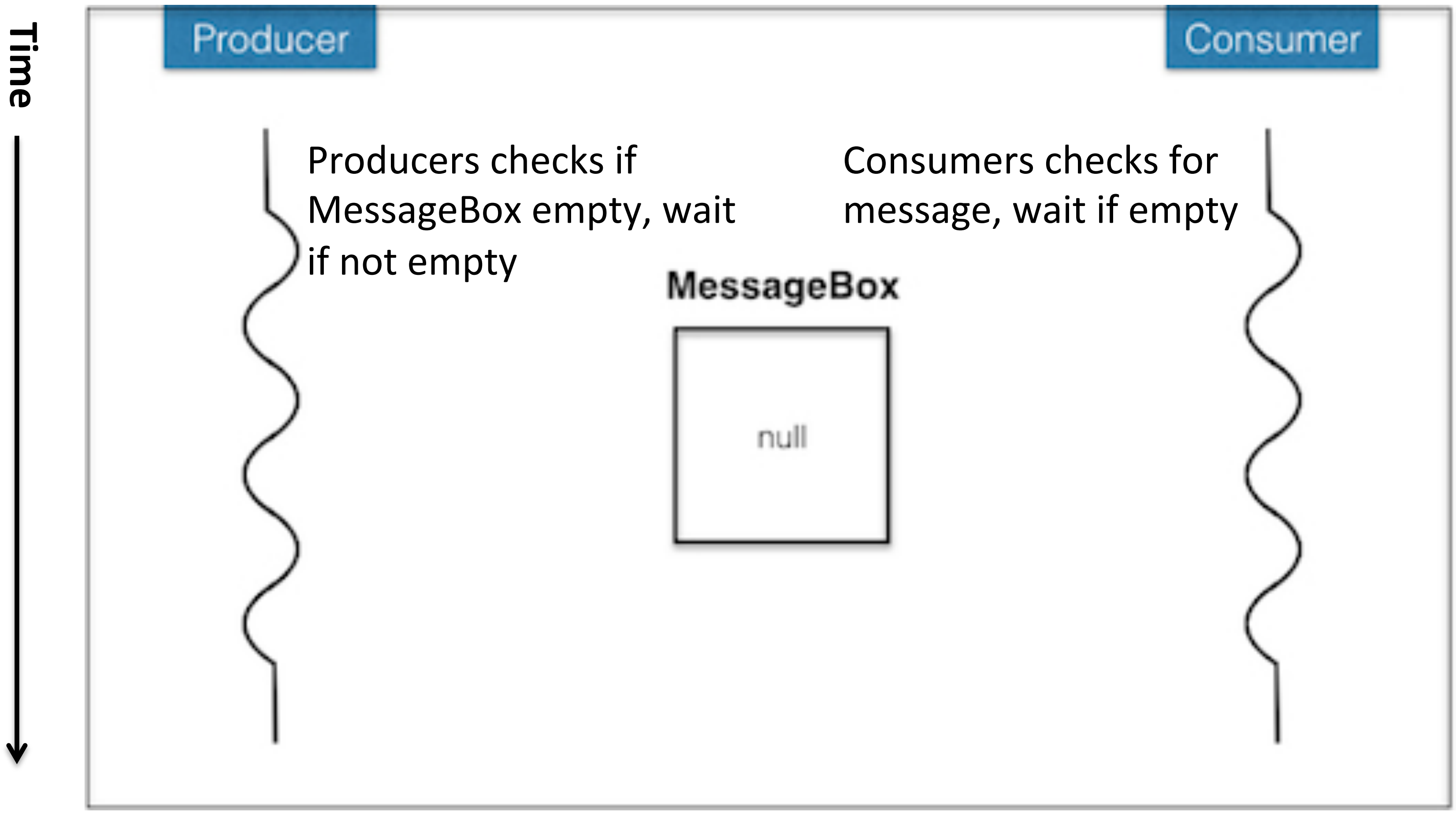
- Tells Operating System to pick a waiting thread and let it run again (not a FIFO queue, OS decides – take CS58 for more)
- Thread should check that conditions are met for it to continue

**`notifyAll`**

- Wake up all waiting threads
- Each thread should check that conditions are met for it to continue

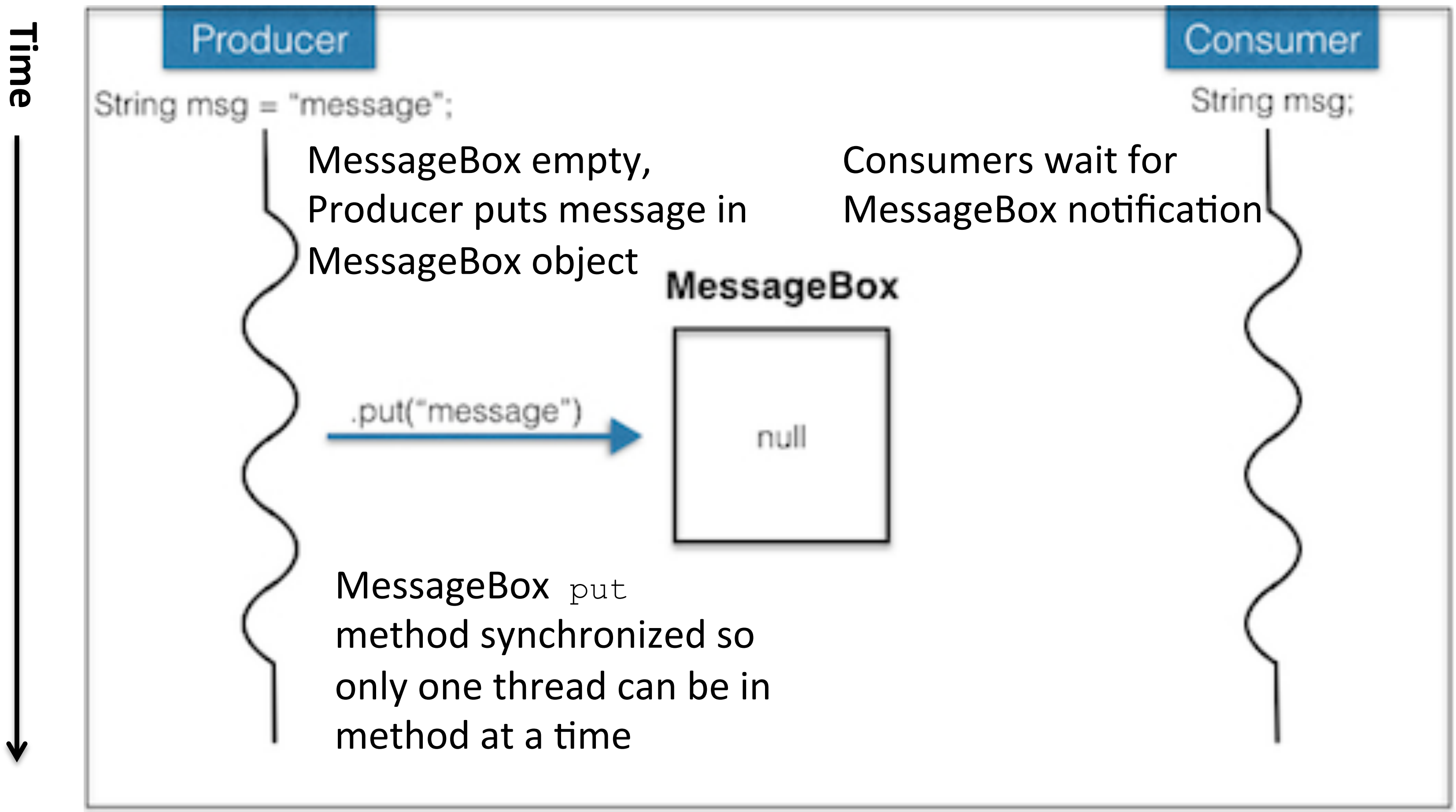
# Producer passing messages to Consumer using semaphore

**Example**



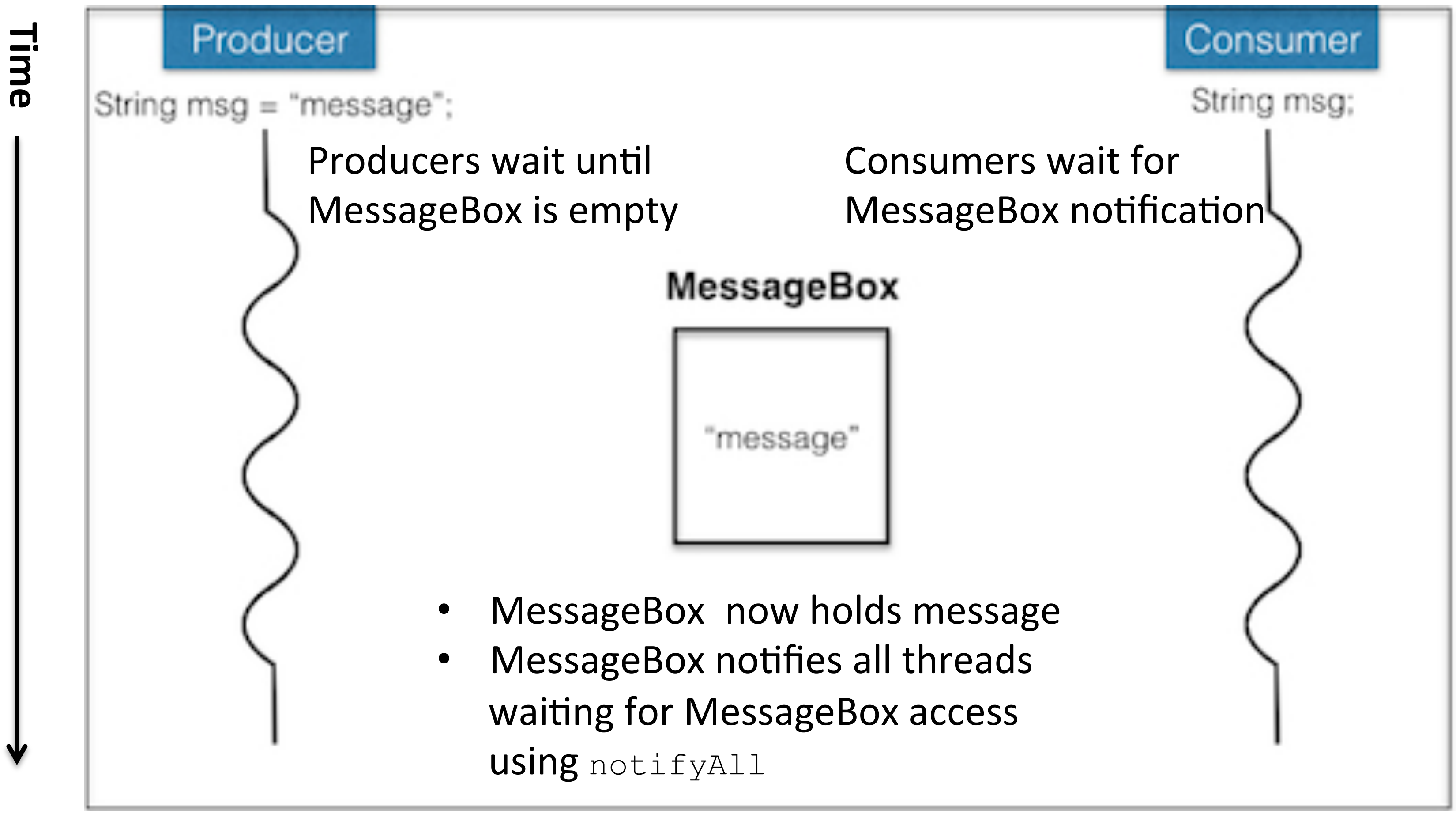
# Producer passing messages to Consumer using semaphore

## Example



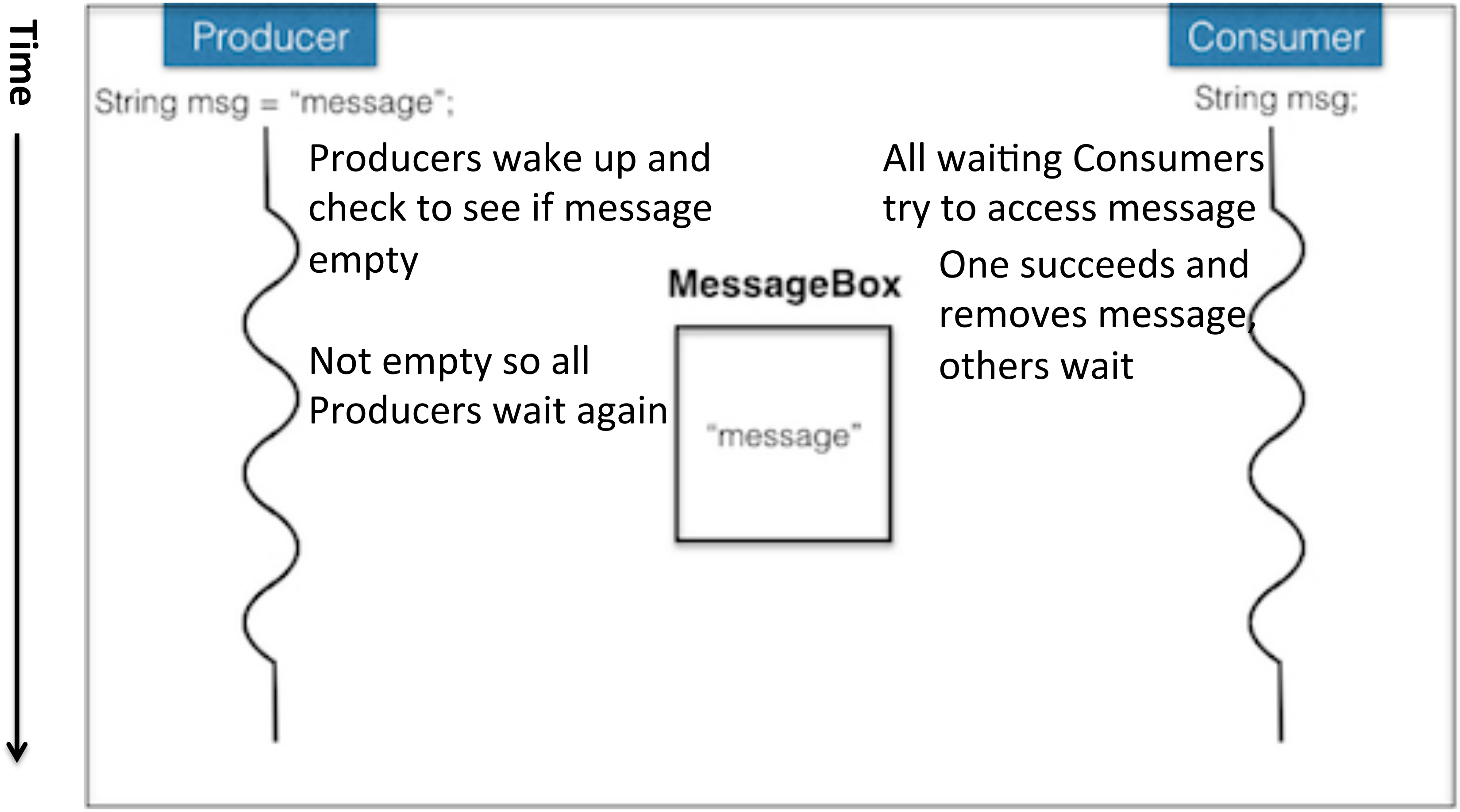
# Producer passing messages to Consumer using semaphore

**Example**



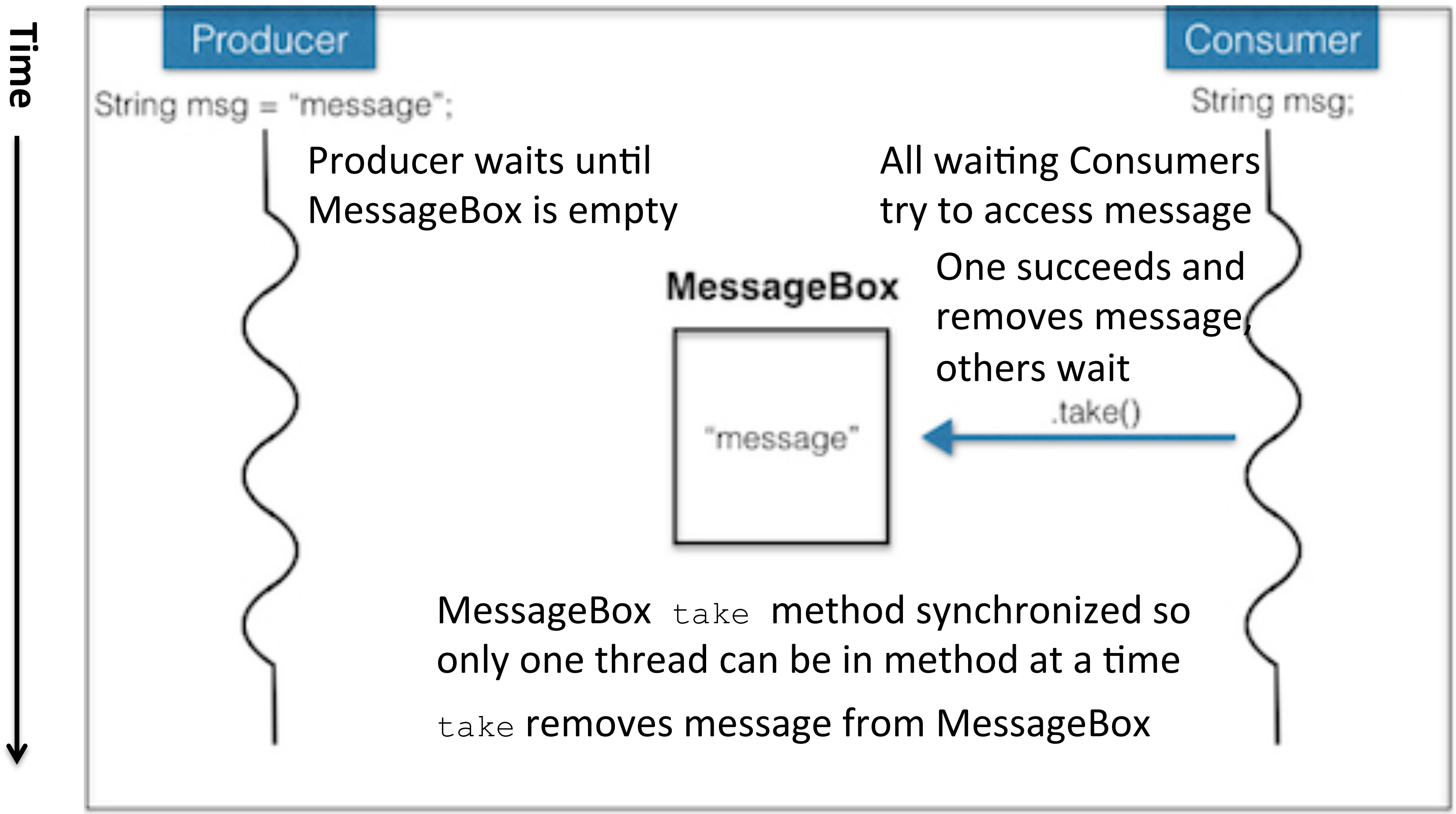
# Producer passing messages to Consumer using semaphore

**Example**



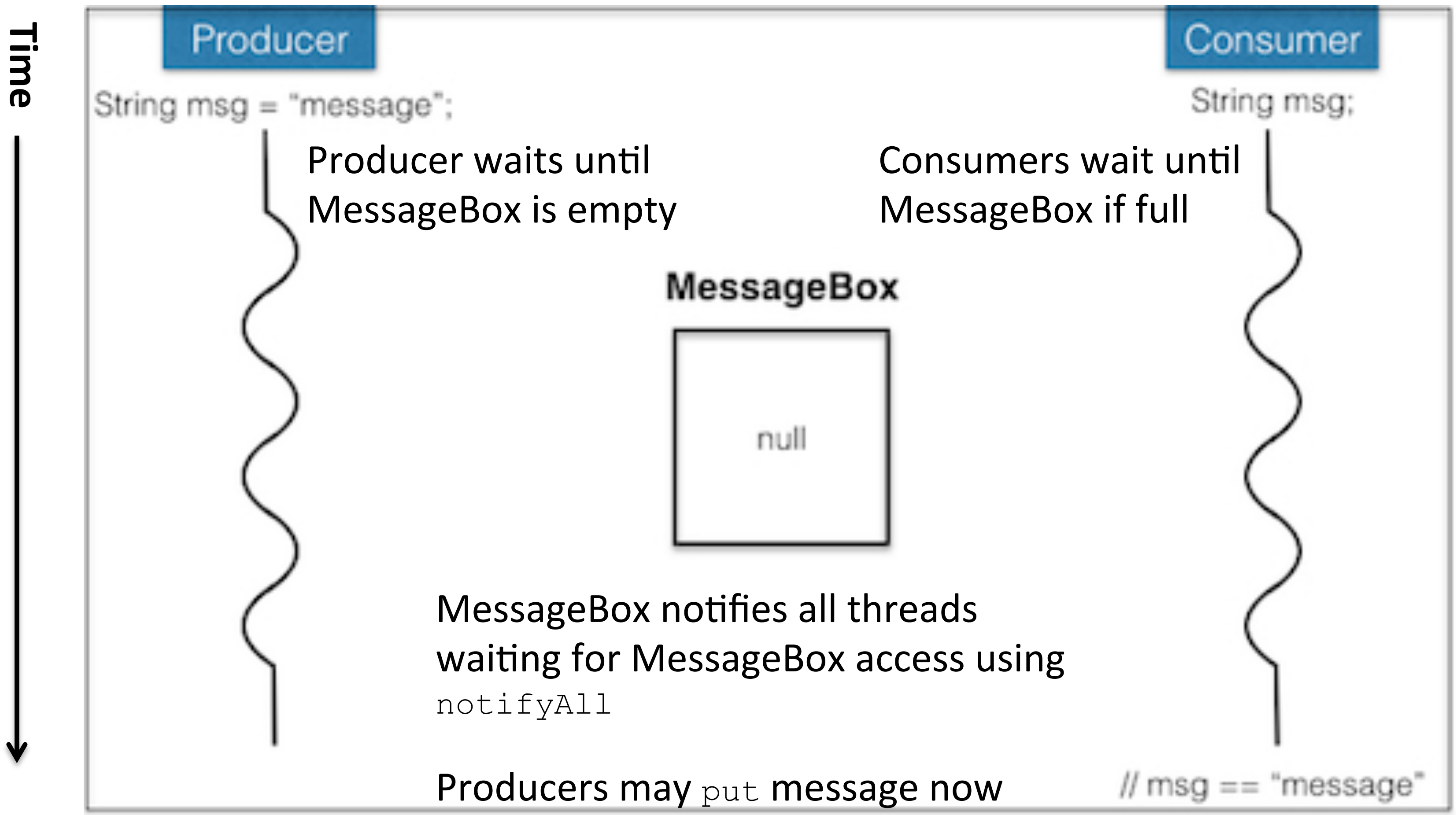
# Producer passing messages to Consumer using semaphore

**Example**



# Producer passing messages to Consumer using semaphore

**Example**



# Producer/Consumer example shows how to use object as semaphore

## ProducerConsumer.java

- Create a `MessageBox, Producer` and `Consumer`
- Start `Producer` and `Consumer` running on different threads
- NOTE: no join, so main thread ends, while threads run

## Producer.java

- `run` method tries to `put` 5 messages into `MessageBox`
- Sleeps for random time between `puts`

## Consumer.java

- Takes messages from `MessageBox`
- Prints `message`

# Producer/Consumer example shows how to use object as semaphore

## MessageBox.java

- Acts as a semaphore
- `put`
  - `Synchronized` so only one thread runs method at a time
  - Causes threads to block with `wait` if message not empty
  - NOTE: empty check in a while loop, just because notified, doesn't mean another thread hasn't already put a message, must make this check!
  - `notifyAll` after setting `message` to wake up all Producers and Consumers (see note above)
- `take`
  - Causes all Consumers to block with `wait` if `message` is null
  - Makes check in while loop like `put`
  - Nulls and returns `message`
  - `notifyAll` to wake up all Producers and Consumers

# Agenda

1. Interleaving execution
2. Producer/consumer
3. Deadlock, starvation

# Synchronization can lead to two problems: deadlocks and starvations

**Deadlock**

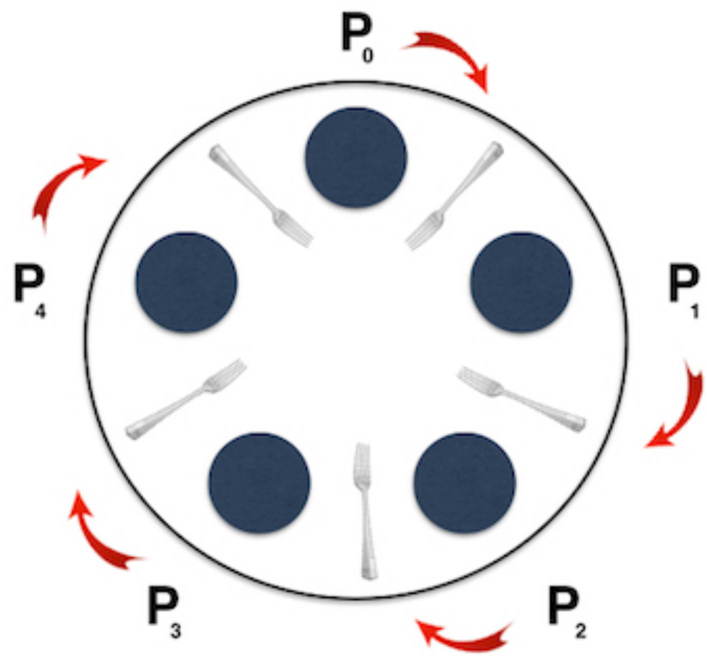
- Objects lock resources
- Execution cannot proceed because object need a resource another locked
- Object A locks resource 1
- Object B locks resource 2
- A needs resource 2 to proceed but B has it locked
- B needs resources 1 to proceed but A has it locked
- A and B are deadlocked

**Starvation**

- Thread never gets resource it needs
- Thread A needs resource 1 to complete
- Other threads always take resource 1 before A can get it
- A is starved

# Dinning Philosophers explains deadlock and starvation

## Dinning Philosophers

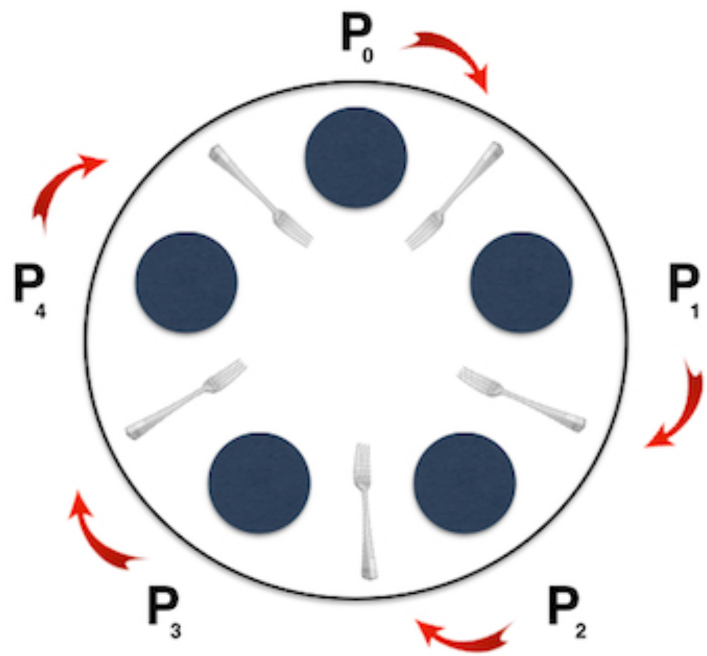



**Problem set up**

- Five philosophers ($P_0$-$P_4$) sit at a table to eat spaghetti
- There are forks between each of them (five total forks)
- Each philosopher needs two forks to eat
- After acquiring two forks, philosopher eats, then puts both forks down
- Another philosopher can then pick up and use fork previously put down (gross!)

# Dinning Philosophers explains deadlock and starvation

**Dinning Philosophers**




**Naïve approach**

- Each philosopher picks up fork on left
- Then picks up fork on right
- Deadlock occurs if all philosophers get left fork, none can get right fork

# For deadlock to occur four conditions must be met

**Deadlock conditions**

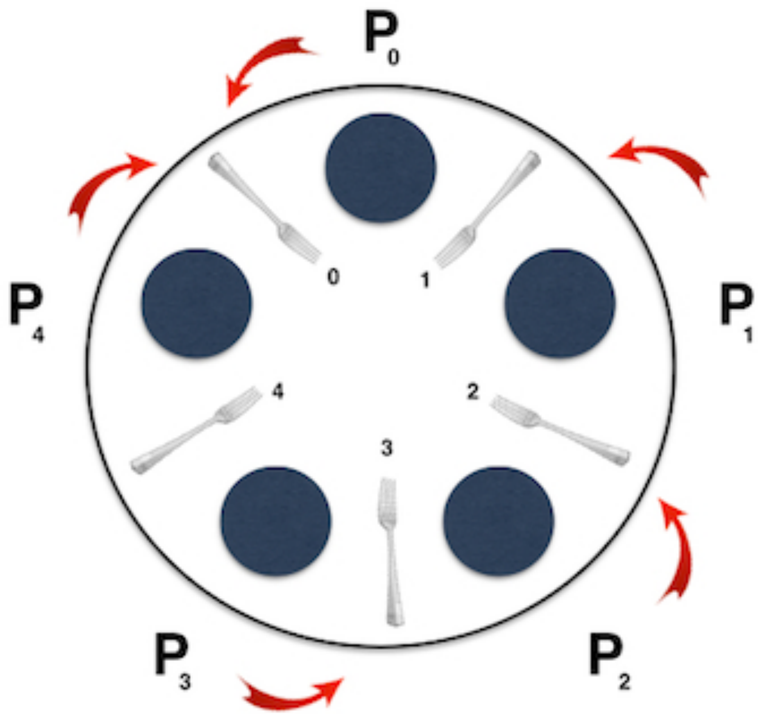
1. **Mutual exclusion**
   - At least one resource class must have non-sharable access. That is,
     - Either one process is using that instance (and others wait), or
     - that instance is free
2. **Hold and wait**
   - At least one process is holding a resource instance, while also waiting to be granted another resource instance. (e.g., Each philosopher is holding on to their left fork, while waiting to pick up their right fork.)
3. **No preemption**
   - Granted resources cannot be pre-empted; a resource can be released only voluntarily by the process holding it (e.g., you can't force the philosophers to drop their forks.)
4. **Circular wait**
   - There must exist a circular chain of at least two processes, each of whom is waiting for a resource held by the previous one. (e.g., each Philosopher[i] is waiting for Philosopher[(i+1) mod 5] to drop its fork.)



From Coffman, 1971

# We can break the deadlock by ensuring the "circular wait" does not occur

**Dinning Philosophers**




Could also force one of the Philosophers to wait at first

**Eliminate circular wait**

- Number each fork in circular fashion
- Make each philosopher pick up lowest numbered fork first
- All pick up right fork, except $P_4$ who tries to pick up left fork 0
- Either $P_0$ or $P_4$ get fork 0
- If $P_0$ gets it, $P_4$ waits for fork 0 before picking up fork 4, so $P_3$ eats
- $P_3$ eventually releases both forks and $P_4$ and $P_2$ eat
- Others eat after $P_4$ and $P_2$
- Cannot deadlock

# Dining Philosophers demonstration

**DiningPhilosphers.java**

- Create 5 philosophers and 5 forks
- Each philosopher has a left and right fork

**Philosopher.java**

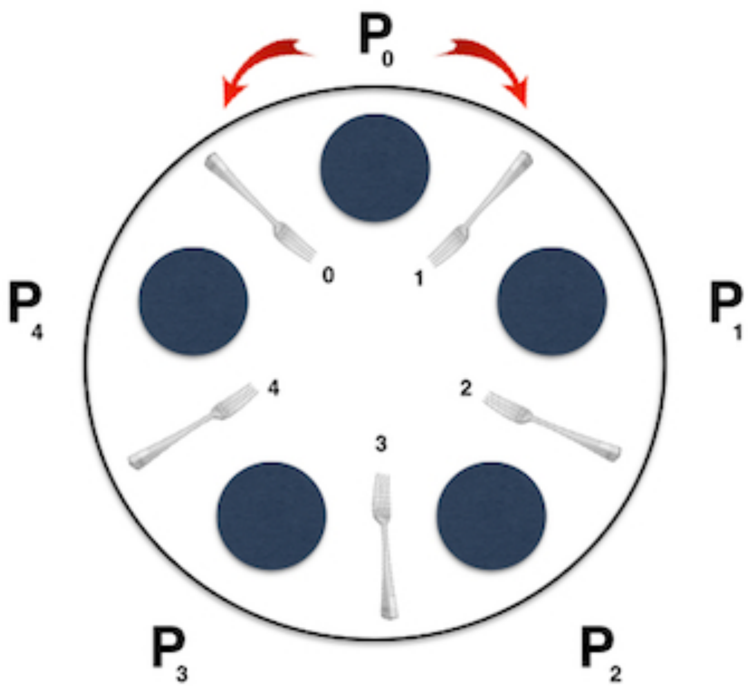
- Each philosopher tries to eat three meals
  - Work up appetite (random pause)
  - `acquire` left (random pause)
  - `acquire` right (random pause)
  - `eat` (random pause)
  - `release` right and left forks

**Fork.java**

- Keep record if `available` (not already acquired)
- Make philosopher `wait` if already acquired
- If not `acquired`, mark fork as `acquired`
- Release – mark as not `acquired` and `notifyAll` waiting

# Another approach is to prevent "hold and wait" by picking up both forks atomically

**Dinning Philosophers**




**Eliminate hold and wait**

- Make picking up both forks an atomic operations
- Forks no longer control their destiny as in prior code
- Now we lock both with a monitor
- Could lead to starvation if one philosopher always picks up before another

# Monitored version avoids deadlocks by picking up both forks atomically

**MonitoredDiningPhilosphers.java**

- `acquire` and `release` moved here to get or release both forks

**Philosopher.java**

- Each philosopher tries to eat three meals
- Uses monitor to `acquire` and `release` both forks

**Fork.java**

- Simply tracks if `available`