Recitation 8

Ammar Karkour

September 29, 2022

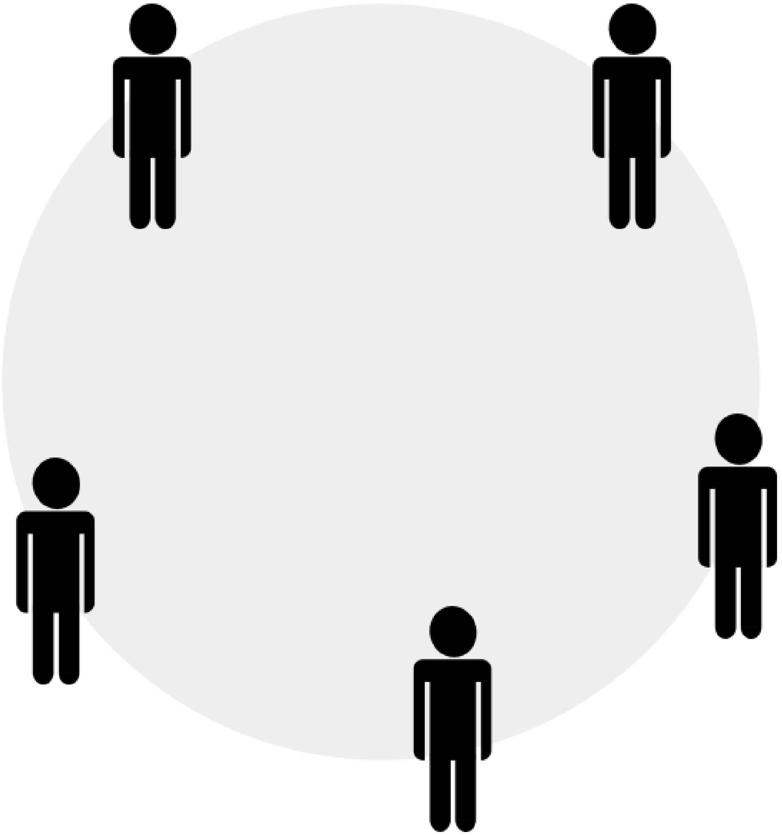
Announcements

- How was the midterm?
- P2 due next Tuesday

Agenda

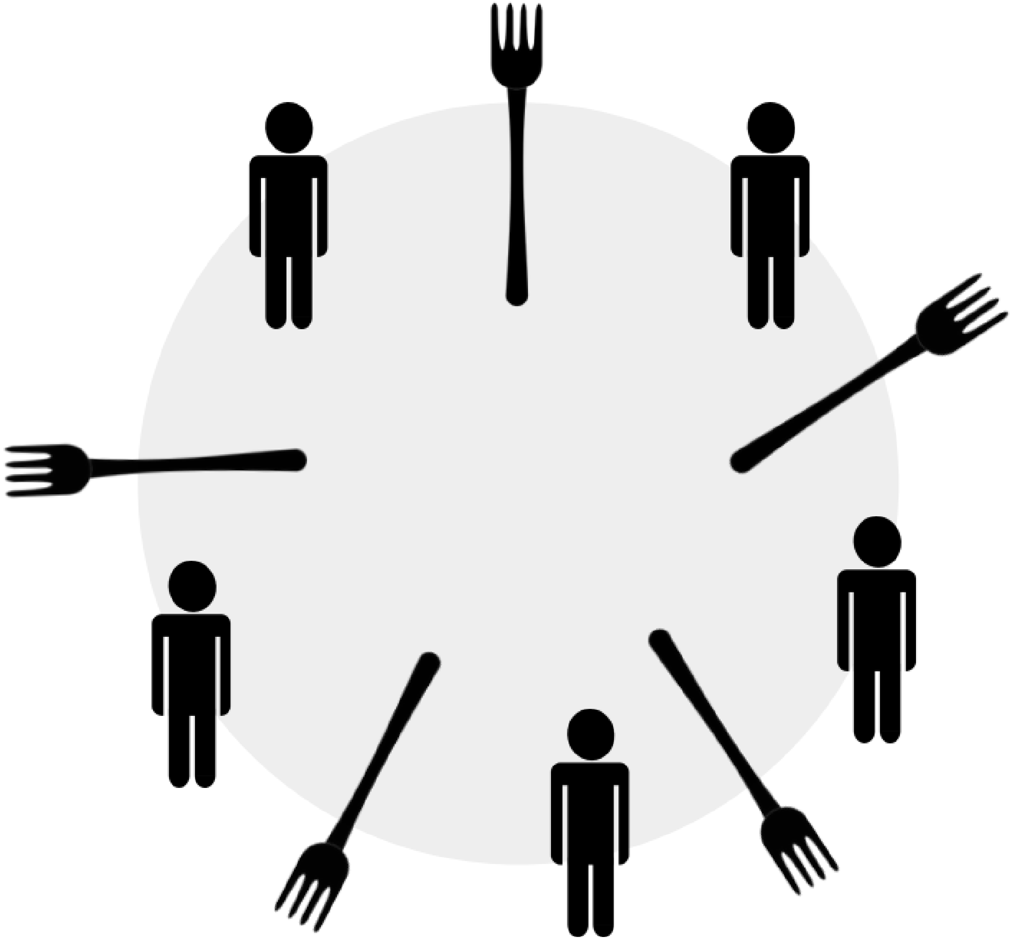
- Dining Philosophers
- Locks in Java

Dining Philosophers



Dining Philosophers

- *Actions: Thinking and Eating*
- *Each P needs a pair of forks*
- *When P is done eating, he is back to thinking and puts back his forks*



Dining Philosophers

Step 1: think until the left chopstick is available; when it is, pick up;

Step 2: think until the right chopstick is available; when it is, pick up;

Step 3: when both chopsticks are held, eat for some time;

Step 4: then, put the right chopstick down;

Step 5: then, put the left chopstick down;

Step 6: repeat from the beginning

Dining Philosophers

A concurrent system with a need for synchronization, should ensure

Correctness

Dining Philosophers

A concurrent system with a need for synchronization, should ensure

Correctness

No two philosophers should be using the same chopsticks at the same time.

Dining Philosophers

A concurrent system with a need for synchronization, should ensure

Correctness

No two philosophers should be using the same chopsticks at the same time.

Efficiency

Dining Philosophers

A concurrent system with a need for synchronization, should ensure

Correctness

No two philosophers should be using the same chopsticks at the same time.

Efficiency

Philosophers do not wait too long to pick-up chopsticks when they want to eat.

Dining Philosophers

A concurrent system with a need for synchronization, should ensure

Correctness

No two philosophers should be using the same chopsticks at the same time.

Efficiency

Philosophers do not wait too long to pick-up chopsticks when they want to eat.

Fairness

Dining Philosophers

A concurrent system with a need for synchronization, should ensure

Correctness

No two philosophers should be using the same chopsticks at the same time.

Efficiency

Philosophers do not wait too long to pick-up chopsticks when they want to eat.

Fairness

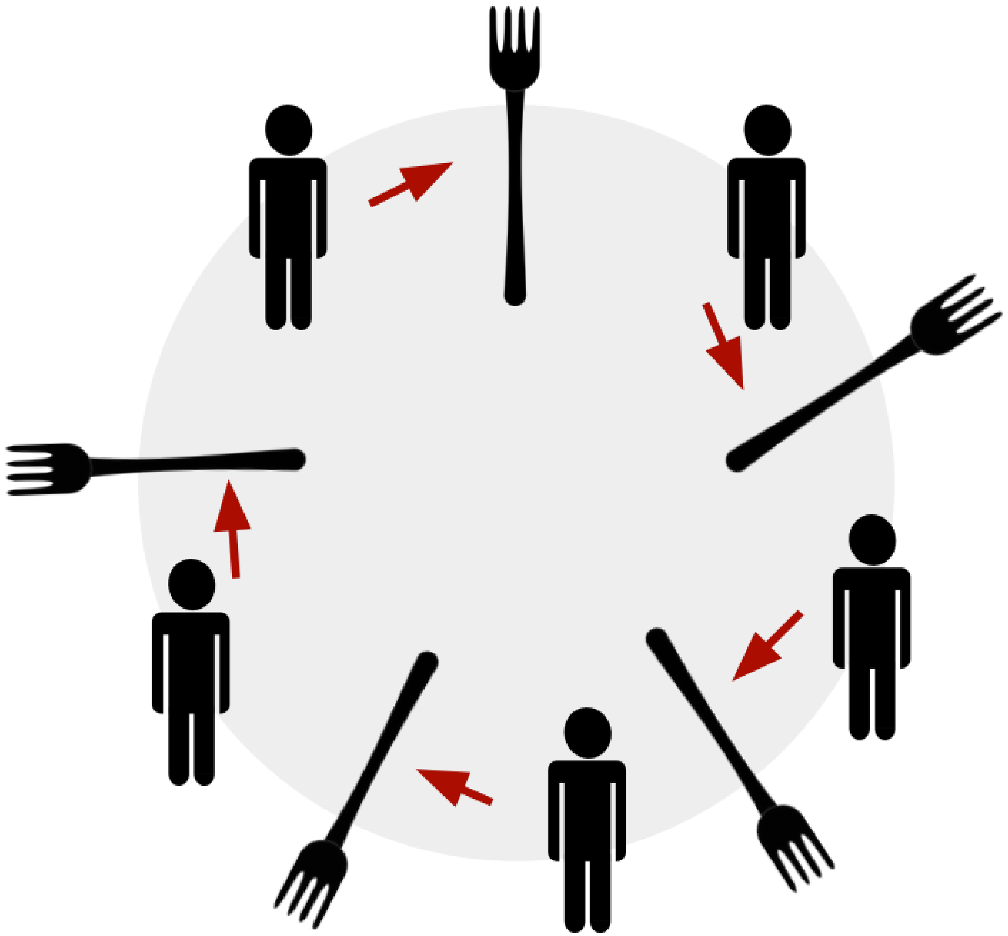
No philosopher should be unable to pick up chopsticks forever and starve

Pseudocode

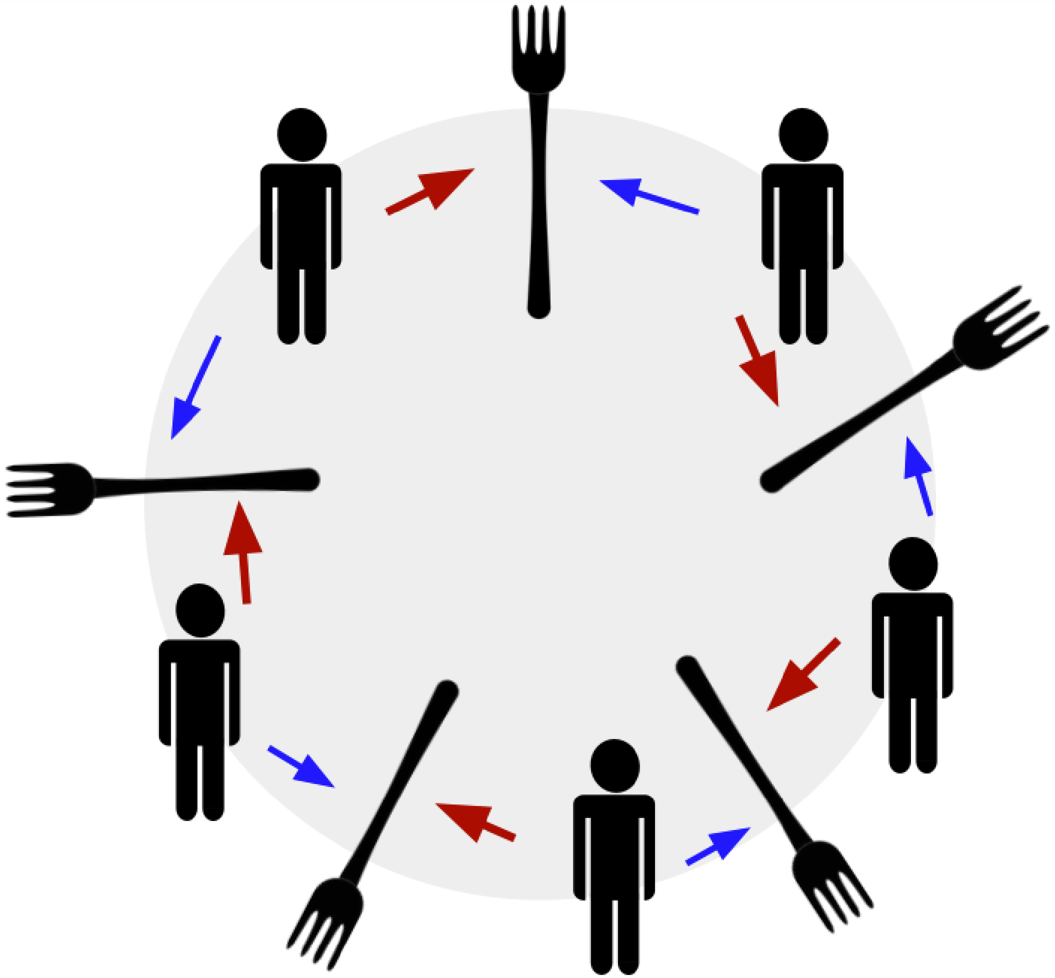
```
while(true) {
    // Initially, thinking about life, universe, and everything
    think();
    // Take a break from thinking, hungry now
    pick_up_left_fork();
    pick_up_right_fork();
    eat();
    put_down_right_fork();
    put_down_left_fork();

    // Not hungry anymore. Back to thinking!
}
```

What's wrong with this code

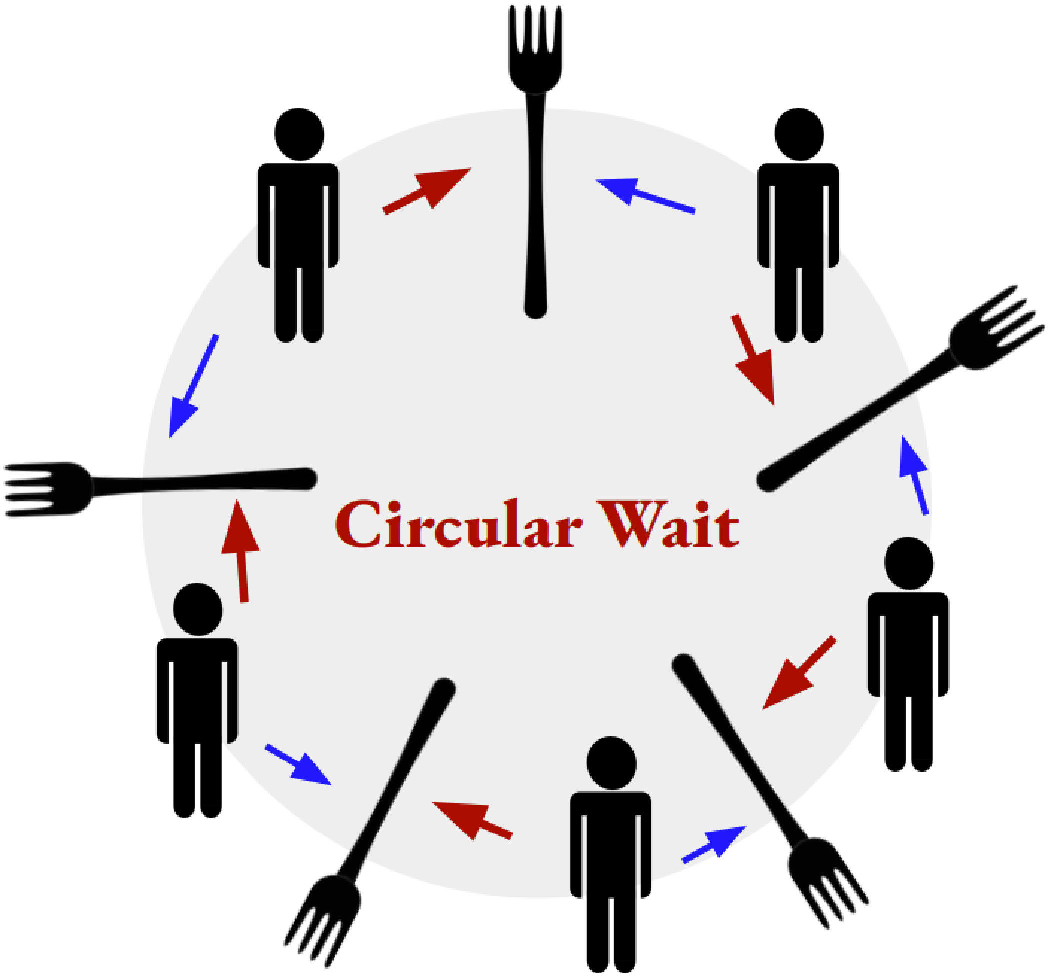
Dining Philosophers



Dining Philosophers



Dining Philosophers



Dining Philosophers

How do we fix this?

A concurrent system with a need for synchronization, should ensure

Correctness

No two philosophers should be using the same chopsticks at the same time.

Efficiency

Philosophers do not wait too long to pick-up chopsticks when they want to eat.

Fairness

No philosopher should be unable to pick up chopsticks forever and starve

Dining Philosophers

```
for (int i = 0; i < philosophers.length; i++) {
    Object leftFork = forks[i];
    Object rightFork = forks[(i+1) % forks.length];
    philosophers[i] = new Philosopher(leftFork, rightFork);
    Thread t = new Thread(philosophers[i], "Philosopher " + (i+1));
    t.start();
}
```

Dining Philosophers

```
for (int i = 0; i < philosophers.length; i++) {
            Object leftFork = forks[i];
            Object rightFork = forks[(i + 1) % forks.length];
            if (i == philosophers.length - 1) {
                // The last philosopher picks up the right fork first
                philosophers[i] = new Philosopher(rightFork, leftFork);
            } else {
                philosophers[i] = new Philosopher(leftFork, rightFork);
            }

            Thread t = new Thread(philosophers[i], "Philosopher " + (i + 1));
            t.start();
        }
```

Locks in Java

Locks vs. Synchronized

Synchronized	Locks
Fully contained within a method	Can have lock() and unlock() operation in separate methods
Rigid, any thread can acquire the lock once released, no preference can be specified	Flexible; we can prioritize waiting threads for example
A thread always gets blocked if it can't get an access to the synchronized block	The Lock API provides tryLock() method. The thread acquires lock only if it's available and not held by any other thread.
A thread which is in “waiting” state to acquire the access to synchronized block, can't be interrupted	The Lock API provides a method lockInterruptibly() which can be used to interrupt the thread when it's waiting for the lock

Lock API

Method	Description
`void lock()`	Acquire the lock if it's available; if the lock isn't available a thread gets blocked until the lock is released
`void lockInterruptibly()`	similar to the *lock(),* but it allows the blocked thread to be interrupted and resume the execution through a thrown *java.lang.InterruptedException*
`boolean tryLock()`	non-blocking version of *lock()* method; it attempts to acquire the lock immediately, return true if locking succeeds
`boolean tryLock(long timeout, TimeUnit timeUnit)`	similar to *tryLock(),* except it waits up the given timeout before giving up trying to acquire the *Lock*
`void unlock()`	unlocks the *Lock* instance

ReadWriteLock

```
ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

readWriteLock.readLock().lock();

    // multiple readers can enter this section
    // if not locked for writing, and not writers waiting
    // to lock for writing.

readWriteLock.readLock().unlock();

readWriteLock.writeLock().lock();

    // only one writer can enter this section,
    // and only if no threads are currently reading.

readWriteLock.writeLock().unlock();
```

- **Read Lock** – *if no thread acquired the write lock or requested for it then multiple threads can acquire the read lock*
- **Write Lock** – *if no threads are reading or writing then only one thread can acquire the write lock*

ReadWriteLock Example

```
    public class SynchronizedHashMapWithReadWriteLock {

        Map<String,String> syncHashMap = new HashMap<>();
        ReadWriteLock lock = new ReentrantReadWriteLock();
```

```
Lock writeLock = lock.writeLock();

        public void put(String key, String value) {
            try {
                writeLock.lock();
                syncHashMap.put(key, value);
            } finally {
                writeLock.unlock();
            }
        }
```

ReadWriteLock Example

```java
public String remove(String key){
    try {
        writeLock.lock();
        return syncHashMap.remove(key);
    } finally {
        writeLock.unlock();
    }
}
```

```java
Lock readLock = lock.readLock();
//...
public String get(String key){
    try {
        readLock.lock();
        return syncHashMap.get(key);
    } finally {
        readLock.unlock();
    }
}
```

Locks with Conditions

- The *Condition* class provides the ability for a thread to wait for some condition to occur while executing the critical section.
- This can occur when a thread acquires the access to the critical section but doesn't have the necessary condition to perform its operation

Example?

- Traditionally Java provides *wait(), notify() and notifyAll()* methods for thread intercommunication. *Conditions* have similar mechanisms, but in addition, we can specify multiple conditions

Locks with Conditions Example

```
public class ReentrantLockWithCondition {

    Stack<String> stack = new Stack<>();
    int CAPACITY = 5;

    ReentrantLock lock = new ReentrantLock();
    Condition stackEmptyCondition = lock.newCondition();
    Condition stackFullCondition = lock.newCondition();
```

Locks with Conditions Example

```
public void pushToStack(String item){
        try {
            lock.lock();
            while(stack.size() == CAPACITY) {
                stackFullCondition.await();
            }
            stack.push(item);
            stackEmptyCondition.signalAll();
        } finally {
            lock.unlock();
        }
    }
```

Locks with Conditions Example

```
 public String popFromStack() {
        try {
            lock.lock();
            while(stack.size() == 0) {
                stackEmptyCondition.await();
            }
            return stack.pop();
        } finally {
            stackFullCondition.signalAll();
            lock.unlock();
        }
    }
}
```

Semaphores

- An integer variable, shared among multiple processes
- A semaphore has two indivisible (atomic) operations, namely: `wait` and `signal`. These operations are sometimes referred to as `P` and `V`, or `down` and `up`.
- The initial value of a semaphore depends on the problem at hand.
- Usually, we use the number of resources available as the initial value.

Semaphores API

Method/Constructor	Description
Semaphore(int permits, boolean fair)	Creates a Semaphore with the given number of permits and the given fairness setting
acquire()	Acquires a permit; blocks until one is available
acquire(int permits)	Acquires the given number of permits from this semaphore, blocking until all are available
tryAcquire()	Return true if a permit is available immediately and acquire it; otherwise return false
availablePermits()	Return number of current permits available
drainPermits()	Acquires and returns all permits that are immediately available

Credits

This recitation was inspired by multiple Baeldung tutorials:

Readers-writers problem

The Dining Philosophers Problem

Locks in Java

Semaphores in Java