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

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.

How do we fix this?
Dining Philosophers

```java
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

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

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

```java
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
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();
    }
}

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.

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

```java
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: \textit{wait} and \textit{signal}. These operations are sometimes referred to as \textit{P} and \textit{V}, or \textit{down} and \textit{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

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

This recitation was inspired by multiple Baeldung tutorials:

Readers-writers problem
The Dining Philosophers Problem
Locks in Java
Semaphores in Java