Announcements

• How was the midterm?
• P2 due next Wednesday

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 goes 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

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
- 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

```java
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!
}
```
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

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();
}

Locks in Java

for (int i = 0; i < philosophers.length; i++) {
    Object leftFork = forks[i];
    Object rightFork = forks[(i + 1) % forks.length];
    if (i == philosophers.length - 1) {
        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 vs. Synchronized

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

ReadWriteLock Example

```java
public class SynchronizedHashMapWithReadWriteLock {
    Map<String, String> syncHashMap = new HashMap<>();
    ReadWriteLock lock = new ReentrantReadWriteLock();

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

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

Locks with Conditions Example

```java
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.

```
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: 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

<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:
- The Dining Philosophers Problem
- Locks in Java
- Semaphores in Java