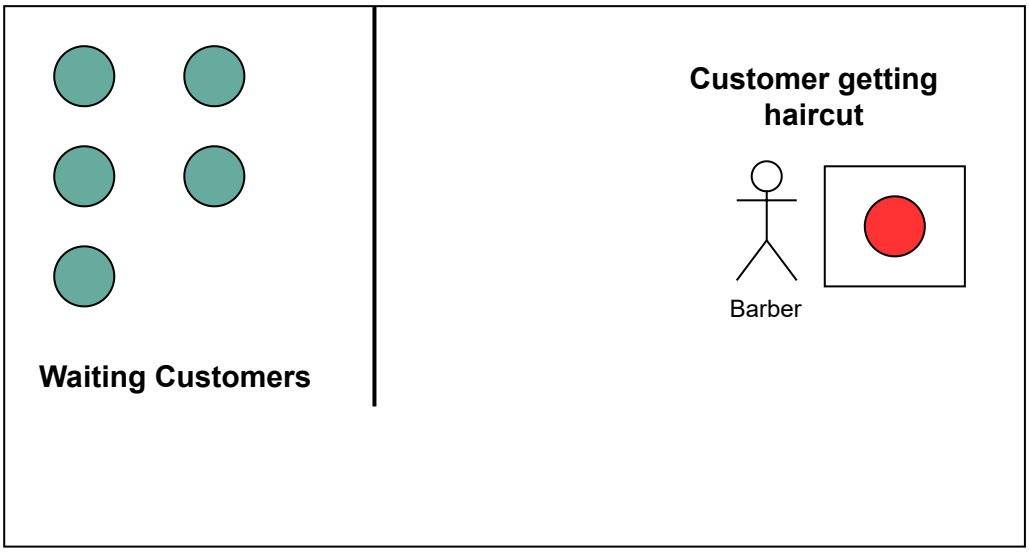
Barber Shop

This lesson visits the synchronization issues when programmatically modeling a hypothetical barber shop and how they are solved using Java's concurrency primitives.

Problem

A similar problem appears in Silberschatz and Galvin's OS book, and variations of this problem exist in the wild.

A barbershop consists of a waiting room with n chairs, and a barber chair for giving haircuts. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the interaction between the barber and the customers.



Barber Shop Problem

Solution

First of all, we need to understand the different state transitions for this problem before we devise a solution. Let's look at them piecemeal:

- A customer enters the shop and if all ***N*** chairs are occupied, he leaves. This hints at maintaining a count of the waiting customers.
- If any of the ***N*** chairs is free, the customer takes up the chair to wait for his turn. Note this translates to using a semaphore on which threads that have found a free chair wait on before being called in by the barber for a haircut.
- If a customer enters the shop and the barber is asleep it implies there are no customers in the shop. The just-entered customer thread wakes up the barber thread. This sounds like using a signaling construct to wake up the barber thread.

We'll have a class which will expose two APIs one for the barber thread to execute and the other for customers. The skeleton of the class would look like the following:

```
public class BarberShopProblem {

   final int CHAIRS = 3;
   int waitingCustomers = 0;
   ReentrantLock lock = new ReentrantLock();

   void customerWalksIn() throws InterruptedException {

   }

   void barber() throws InterruptedException {
   }
}
```

Now let's think about the customer thread. It enters the shop, acquires a lock to test the value of the counter `waitingCustomers`. We must test the value of this variable while no other thread can modify its value, hinting

that we'll wrap the test under a lock. If the value equals all the chairs available, then the customer thread gives up the lock and returns from the method. If a chair is available the customer thread increments the variable `waitingCustomers`. Remember, the barber might be asleep which can be modeled as the barber thread waiting on a semaphore `waitForCustomerToEnter`. The customer thread must signal the semaphore `waitForCustomerToEnter` in case the barber is asleep.

Next, the customer thread itself needs to wait on a semaphore before the barber comes over, greets the customer and leads him to the salon chair. Let's call this semaphore `waitForBarberToGetReady`. This is the same semaphore the barber signals as soon as it wakes up. All customer threads waiting for a haircut will block on this `waitForBarberToGetReady` semaphore. The barber signaling this semaphore is akin to letting one customer come through and sit on the barber chair for a haircut. This logic when coded looks like the following:

```
void customerWalksIn() throws InterruptedException {

        lock.lock();
        if (waitingCustomers == CHAIRS) {
            System.out.println("Customer walks out, all chairs occupi
ed.");
            // Remember to unlock before leaving
            lock.unlock();
            return;
        }
        waitingCustomers++;
        lock.unlock();

        // Let the barber know you are here, in case he's asleep
        waitForCustomerToEnter.release();
        // Wait for the barber to come take you to the salon chair wh
en its your turn
        waitForBarberToGetReady.acquire();
        // TODO: complete the rest of the logic.
}
```

Now let's work with the barber code. This should be a perpetual loop, where the barber initially waits on the semaphore `waitForCustomerToEnter` to simulate no customers in the shop. If woken up, then it implies that there's at least one customer in the shop who

needs a hair-cut and the barber gets up, greets the customer and leads

him to his chair before starting the haircut. This sequence is translated into code as the barber thread signaling the `waitForBarberToGetReady` semaphore. Next, the barber simulates a haircut by sleeping for 50 milliseconds

Once the haircut is done. The barber needs to inform the customer thread too; it does so by signaling the `waitForBarberToCutHair` semaphore. The customer thread should already be waiting on this semaphore.

Finally, to make the barber thread know that the current customer thread has left the barber chair and the barber can bring in the next customer, we make the barber thread wait on yet another semaphore `waitForCustomerToLeave`. This is the same semaphore the customer thread needs to signal before exiting. The barber thread's implementation appears below:

```
    void barber() throws InterruptedException {

        while (true) {
            // wait till a customer enters a shop
            waitForCustomerToEnter.acquire();
            // let the customer know barber is ready
            waitForBarberToGetReady.release();

            System.out.println("Barber cutting hair...");
            Thread.sleep(50);

            // let customer thread know, haircut is done
            waitForBarberToCutHair.release();
            // wait for customer to leave the barber chair
            waitForCustomerToLeave.acquire();
        }
    }
```

The complete customer thread code appears below:

```
    void customerWalksIn() throws InterruptedException {

        lock.lock();
        if (waitingCustomers == CHAIRS) {
            System.out.println("Customer walks out, all chairs occupi
```

```
            System.out.println("Customer walks out, all chairs occupied");
            lock.unlock();
            return;
        }
        waitingCustomers++;
        lock.unlock();

        // Let the barber know, there's atleast 1 customer
        waitForCustomerToEnter.release();
        // Wait for barber to greet you and lead you to barber chair
        waitForBarberToGetReady.acquire();

        // This is where the customer gets the haircut

        // Wait for haircut to complete
        waitForBarberToCutHair.acquire();
        // Leave the barber chair and let barber thread know chair is vacant
        waitForCustomerToLeave.release();

        lock.lock();
        waitingCustomers--;
        lock.unlock();
    }
```

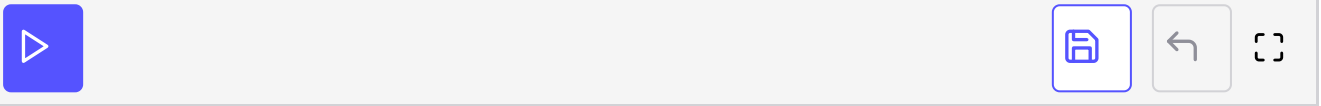
Complete Code

The entire code alongwith the test appears below. Since the barber thread is perpetual, the widget execution would time-out.

```
import java.util.HashSet;
import java.util.concurrent.Semaphore;
import java.util.concurrent.locks.ReentrantLock;

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        BarberShopProblem.runTest();
    }
}

class BarberShopProblem {

    final int CHAIRS = 3;
```

```
Semaphore waitForCustomerToEnter = new Semaphore(0);
Semaphore waitForBarberToGetReady = new Semaphore(0);
Semaphore waitForCustomerToLeave = new Semaphore(0);

Semaphore waitForBarberToCutHair = new Semaphore(0);
int waitingCustomers = 0;
ReentrantLock lock = new ReentrantLock();
int hairCutsGiven = 0;

void customerWalksIn() throws InterruptedException {

    lock.lock();
    if (waitingCustomers == CHAIRS) {
        System.out.println("Customer walks out, all chairs occupied");
        lock.unlock();
        return;
    }
    waitingCustomers++;
    lock.unlock();

    waitForCustomerToEnter.release();
    waitForBarberToGetReady.acquire();

    waitForBarberToCutHair.acquire();
    waitForCustomerToLeave.release();

    lock.lock();
    waitingCustomers--;
    lock.unlock();
}

void barber() throws InterruptedException {

    while (true) {
        waitForCustomerToEnter.acquire();
        waitForBarberToGetReady.release();
        hairCutsGiven++;
        System.out.println("Barber cutting hair..." + hairCutsGiven);
        Thread.sleep(50);
        waitForBarberToCutHair.release();
        waitForCustomerToLeave.acquire();
    }
}

public static void runTest() throws InterruptedException {

    HashSet<Thread> set = new HashSet<Thread>();
    final BarberShopProblem barberShopProblem = new BarberShopProblem();

    Thread barberThread = new Thread(new Runnable() {
        public void run() {
            try {
                barberShopProblem.barber();
            } catch (InterruptedException ie) {

            }
        }
    });
    barberThread.start();

    for (int i = 0; i < 10; i++) {
        Thread t = new Thread(new Runnable() {
            public void run() {
```

```
                    try {
                        barberShopProblem.customerWalksIn();
                    } catch (InterruptedException ie) {

                    }
                }
            });
            set.add(t);
        }

        for (Thread t : set) {
            t.start();
        }

        for (Thread t : set) {
            t.join();
        }

        set.clear();
        Thread.sleep(800);

        for (int i = 0; i < 5; i++) {
            Thread t = new Thread(new Runnable() {
                public void run() {
                    try {
                        barberShopProblem.customerWalksIn();
                    } catch (InterruptedException ie) {

                    }
                }
            });
            set.add(t);
        }
        for (Thread t : set) {
            t.start();
        }

        barberThread.join();
    }
}
```



The execution output would show 6 customers getting a haircut and the rest walking out since there are only three chairs available at the barber shop.

Note we only decrement `waitingCustomers` ***after*** the customer thread has received a haircut. However, you may argue that since the customer getting the haircut has left one spot open in the waiting area, it should be possible to have one more thread come in the shop and wait for a total of

four threads. Three threads wait on chairs in the waiting area and one thread occupies the barber chair where it undergoes a hair-cut. If we tweak our implementation to compensate for this change (**lines 37, 38, 39** in the below widget) then we'll see the above test give eight haircuts instead of six. The change entails we decrement the `waitingCustomers` variable right after the barber seats a customer. The code with the change appears below. If you run the widget, you'll see eight threads getting haircut.

```
import java.util.HashSet;
import java.util.concurrent.Semaphore;
import java.util.concurrent.locks.ReentrantLock;

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        BarberShopProblem.runTest();
    }
}

class BarberShopProblem {

    final int CHAIRS = 3;
    Semaphore waitForCustomerToEnter = new Semaphore(0);
    Semaphore waitForBarberToGetReady = new Semaphore(0);
    Semaphore waitForCustomerToLeave = new Semaphore(0);
    Semaphore waitForBarberToCutHair = new Semaphore(0);
    int waitingCustomers = 0;
    ReentrantLock lock = new ReentrantLock();
    int hairCutsGiven = 0;

    void customerWalksIn() throws InterruptedException {

        lock.lock();
        if (waitingCustomers == CHAIRS) {
            System.out.println("Customer walks out, all chairs occupied");
            lock.unlock();
            return;
        }
        waitingCustomers++;
        lock.unlock();

        waitForCustomerToEnter.release();
        waitForBarberToGetReady.acquire();

        // The chair in the waiting area becomes available
        lock.lock();
        waitingCustomers--;
        lock.unlock();

        waitForBarberToCutHair.acquire();
        waitForCustomerToLeave.release();
    }

    void barber() throws InterruptedException {
```

```
    void barber() throws InterruptedException {

        while (true) {
            waitForCustomerToEnter.acquire();
            waitForBarberToGetReady.release();
            hairCutsGiven++;
            System.out.println("Barber cutting hair..." + hairCutsGiven);
            Thread.sleep(50);
            waitForBarberToCutHair.release();
            waitForCustomerToLeave.acquire();
        }
    }

    public static void runTest() throws InterruptedException {

        HashSet<Thread> set = new HashSet<Thread>();
        final BarberShopProblem barberShopProblem = new BarberShopProblem();

        Thread barberThread = new Thread(new Runnable() {
            public void run() {
                try {
                    barberShopProblem.barber();
                } catch (InterruptedException ie) {

                }
            }
        });
        barberThread.start();

        for (int i = 0; i < 10; i++) {
            Thread t = new Thread(new Runnable() {
                public void run() {
                    try {
                        barberShopProblem.customerWalksIn();
                    } catch (InterruptedException ie) {

                    }
                }
            });
            set.add(t);
        }

        for (Thread t : set) {
            t.start();
        }

        for (Thread t : set) {
            t.join();
        }

        set.clear();
        Thread.sleep(500);

        for (int i = 0; i < 5; i++) {
            Thread t = new Thread(new Runnable() {
                public void run() {
                    try {
                        barberShopProblem.customerWalksIn();
                    } catch (InterruptedException ie) {

                    }
                }
            });
```

```
            set.add(t);
        }
        for (Thread t : set) {
            t.start();
            Thread.sleep(5);
        }

        barberThread.join();
    }
}
```