CSE201: Monsoon 2017
Advanced Programming

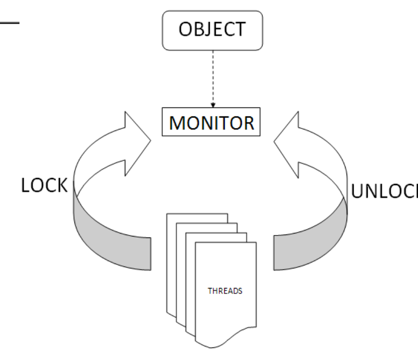
Lecture 31: Mutual Exclusion (Part 2)

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture

- ***Critical section:*** a block of code that access shared modifiable data or resource that should be operated on by only one thread at a time.
- ***Mutual exclusion:*** a property that ensures that a critical section is only executed by a thread at a time
- Each object has a "**monitor**" that is a token used to determine which application **thread** has control of a particular **object** instance
- Demerits of monitor lock
  - Does not guarantee fairness
    - Lock might not be given to the longest waiting thread
  - Might lead to starvation
    - A thread can indefinitely hold the monitor lock for doing some big computation while other threads keep waiting to get this monitor lock
    - Not possible to interrupt the waiting thread
    - Not possible for a thread to decline waiting for the lock if its unavailable
- java.util.concurrent.ReentrantLock overcomes the above limitations of monitor lock

```
class Counter implements Runnable {
    volatile int counter = 0;
    public synchronized int value() { return counter; }

    public synchronized void run() {
        if(value() < 100) {
            counter++;
        }
    }
    public void run() { increment(); }
    public static void main(String[] args)
                              throws InterruptedException {
        ExecutorService exec =
                    Executors.newFixedThreadPool(2);
        Counter task = new Counter();
        for(int i=0; i<1000; i++) {
            exec.execute(task);
        }
        if(!exec.isTerminated()) {
          exec.shutdown();
          exec.awaitTermination(5L,TimeUnit.SECONDS);
        }
        System.out.println(task.counter);
    }
}
```



Today's Lecture

- Deadlocks
- Producer consumer program
  - Communication across threads
- Lock free thread-safe programming

Acknowledgement: some slides from CS67633, Hebrew University


DEADLOCK
© CanStockPhoto.com - csp50107200

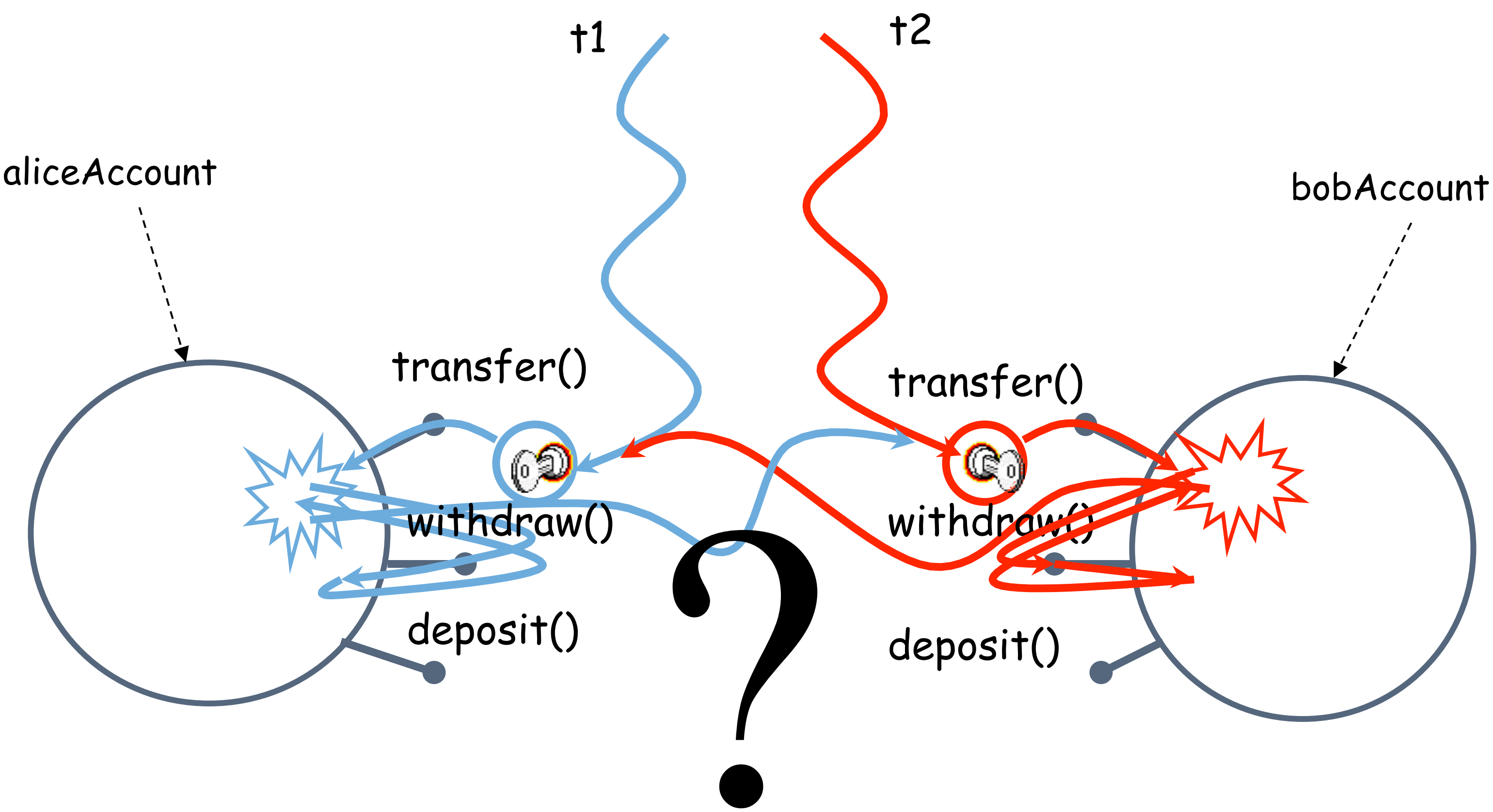
Let’s Code a Deadlock

```
public class BankAccount {

    private volatile float balance;

    public synchronized void deposit(float amount) {
        balance += amount;
    }

    public synchronized void withdraw(float amount) {
        balance -= amount;
    }

    public synchronized void transfer(float amount,
                                      BankAccount target) {
        withdraw(amount);
        target.deposit(amount);
    }
}
```

```
public class MoneyTransfer implements Runnable {
    private BankAccount source, target;
    private float amount;
    public MoneyTransfer(BankAccount from,
                         BankAccount to, float amount) {
        this.source = from;
        this.target = to;
        this.amount = amount;
    }
    public void run() {
        source.transfer(amount, target);
    }
}
```

```
BankAccount aliceAccount = new BankAccount();
BankAccount bobAccount = new BankAccount();
...
// At one place
Runnable transaction1 = new MoneyTransfer(aliceAccount, bobAccount, 1200);
Thread t1 = new Thread(transaction1);
t1.start();

// At another place
Runnable transaction2 = new MoneyTransfer(bobAccount, aliceAccount, 700);
Thread t2 = new Thread(transaction2);
t2.start();
```

Let’s Analyze Our Bank Transaction



Deadlock Avoidance

- Deadlock occurs when multiple threads need the same locks but obtain them in different order
- Not so easy to avoid deadlocks
- It's an active research area
    - If you are interested to know more about it then you can read the following conference paper
        - *Kumar et. al., "Integrating asynchronous task parallelism and data-centric atomicity", PPPJ 2016, Switzerland*

Let's try two simple remedies to fix our Bank Transaction program

Deadlock Avoidance

- Lock ordering
  - Ensure that all locks are taken in same order by any thread
- Lock timeout
  - Put a timeout on lock attempts
    - Use ReentrantLock.tryLock instead of monitor lock

Now Let’s Resolve the Deadlock

```java
public class BankAccount {

    private volatile float balance;
    final int account_id;

    public BankAccount(int i) { account_id = i; }

    public synchronized void deposit(float amount) {
        balance += amount;
    }

    public synchronized void withdraw(float amount) {
        balance -= amount;
    }

    public synchronized void transfer(float amount,
                                      BankAccount target) {
        withdraw(amount);
        target.deposit(amount);
    }
}
```

```java
public class MoneyTransfer implements Runnable {
    private BankAccount source, target;
    private float amount;
    public MoneyTransfer(BankAccount from,
                         BankAccount to, float amount) {
        this.source = from;
        this.target = to;
        this.amount = amount;
    }
    public void run() {
        Object obj1 = null, obj2 = null;
        if(source.account_id > target.account_id) {
            obj1=target; obj2=source;
        }
        else { obj1=source; obj2=target; }
        synchronized(obj1) { synchronized(obj2) {
                source.transfer(amount, target);
        } }
    }
}
```

```java
BankAccount aliceAccount = new BankAccount(1); // account_id = 1;
BankAccount bobAccount = new BankAccount(2);   // account_id = 2;
...
// At one place
Runnable transaction1 = new MoneyTransfer(aliceAccount, bobAccount, 1200);
Thread t1 = new Thread(transaction1);
t1.start();

// At another place
Runnable transaction2 = new MoneyTransfer(bobAccount, aliceAccount, 700);
Thread t2 = new Thread(transaction2);
t2.start();
```

- We are using lock ordering technique here to resolve the deadlock
- Lock on BankAccount objects are taken in run() method as per the ascending order value of the account_id
    - Recall monitor locks are reentrant

The Producer Consumer Problem

- We need to synchronized between transactions, for example, the consumer-producer scenario

Wait and Notify

- Allows two threads to cooperate
- Based on a single shared lock object
    - Marge put a cookie wait and notify Homer
    - Homer eat a cookie wait and notify Marge
    - Marge put a cookie wait and notify Homer
    - Homer eat a cookie wait and notify Marge

The `wait()` Method

- The `wait()` method is part of the class **java.lang.Object**
- It requires a lock on the object's monitor to execute
- It must be called from a synchronized method, or from a synchronized segment of code
- `wait()` causes the current thread to relinquish the CPU and wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object
- Upon call for wait(), the thread releases ownership of this monitor and waits until another thread notifies the waiting threads of the object

Wait/Notify Sequence

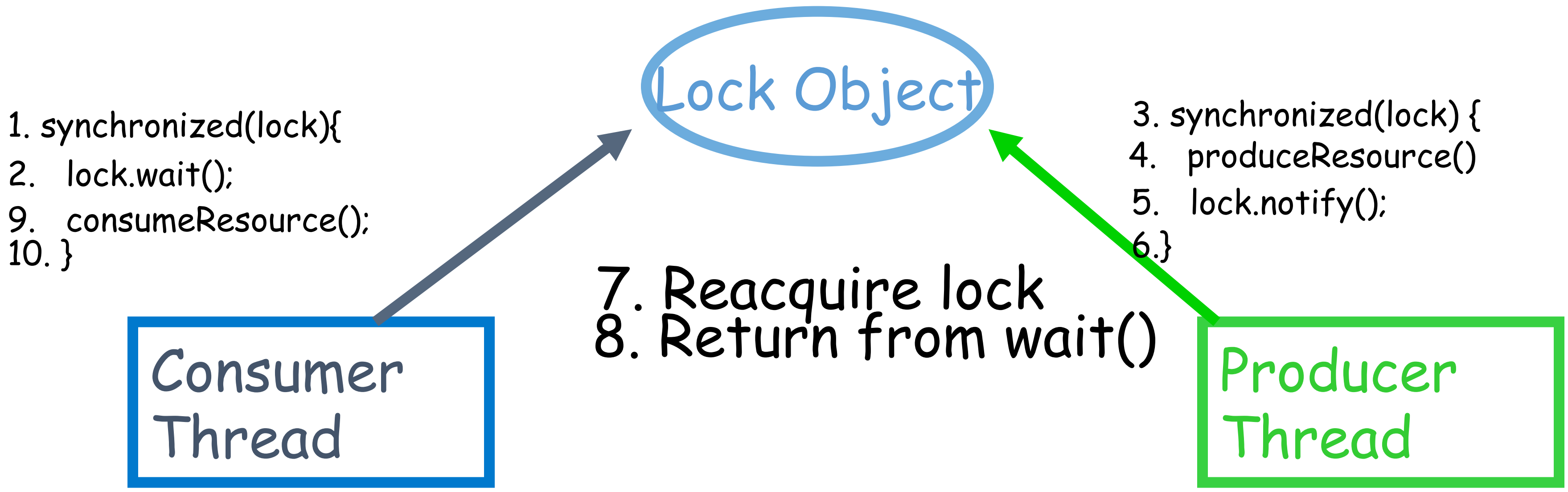
```
1. synchronized(lock){
2.   lock.wait();
9.   consumeResource();
10. }
```

Consumer

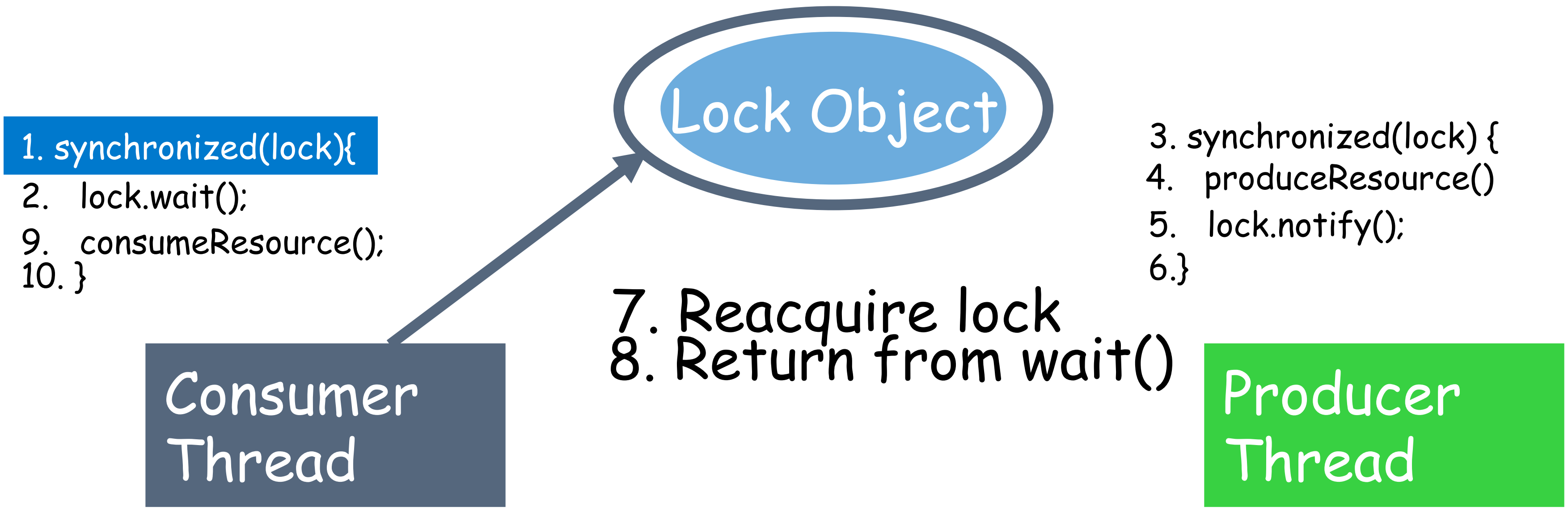
```
3. synchronized(lock) {
4.   produceResource()
5.   lock.notify();
6.}
```

Producer

Wait/Notify Sequence



Wait/Notify Sequence



Wait/Notify Sequence

Lock Object

```
1. synchronized(lock){
2.   lock.wait();
9.   consumeResource();
10. }
```

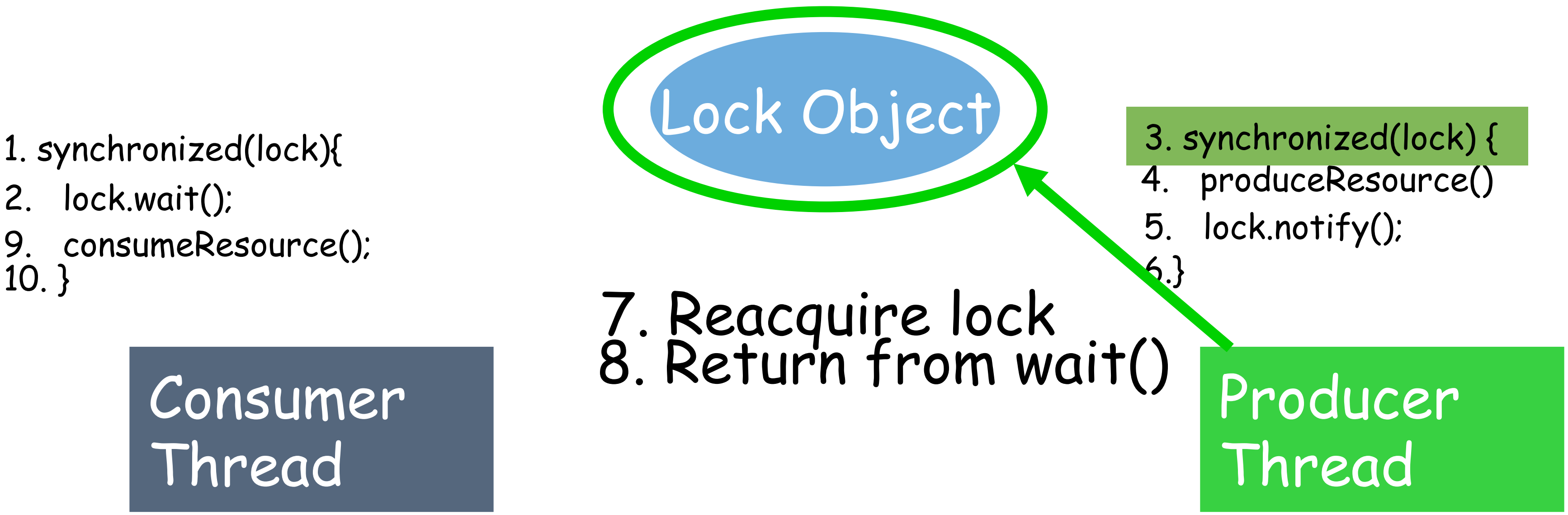
```
3. synchronized(lock) {
4.   produceResource()
5.   lock.notify();
6.}
```

7. Reacquire lock
8. Return from wait()

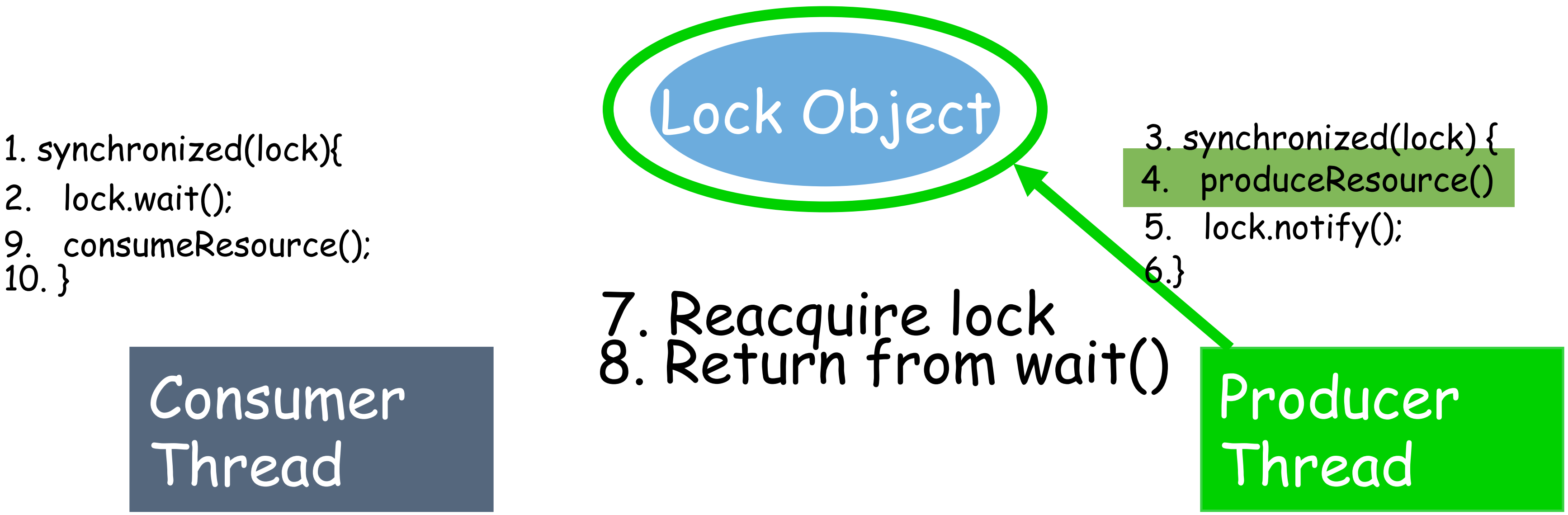
Consumer Thread

Producer Thread

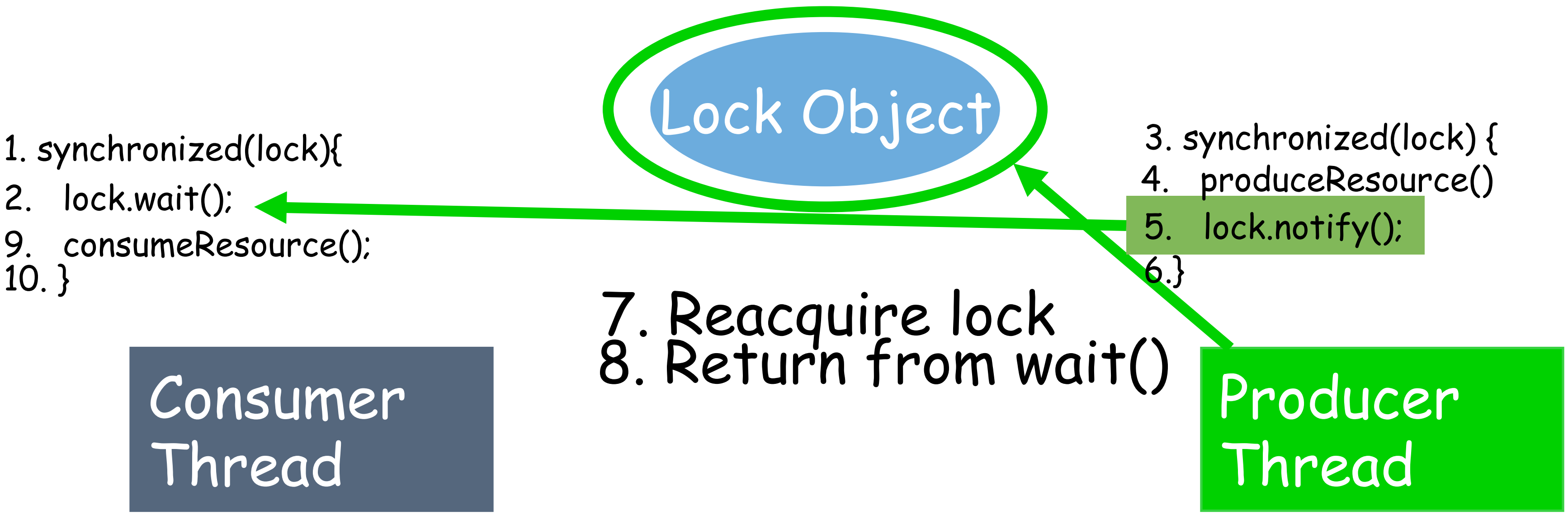
Wait/Notify Sequence



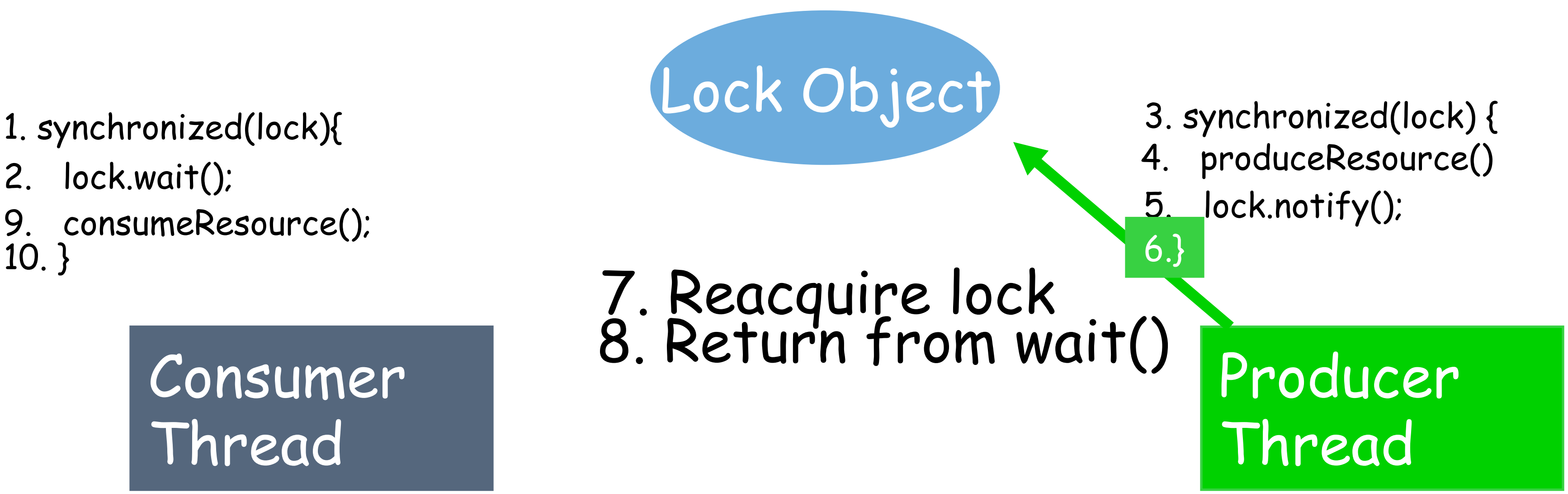
Wait/Notify Sequence



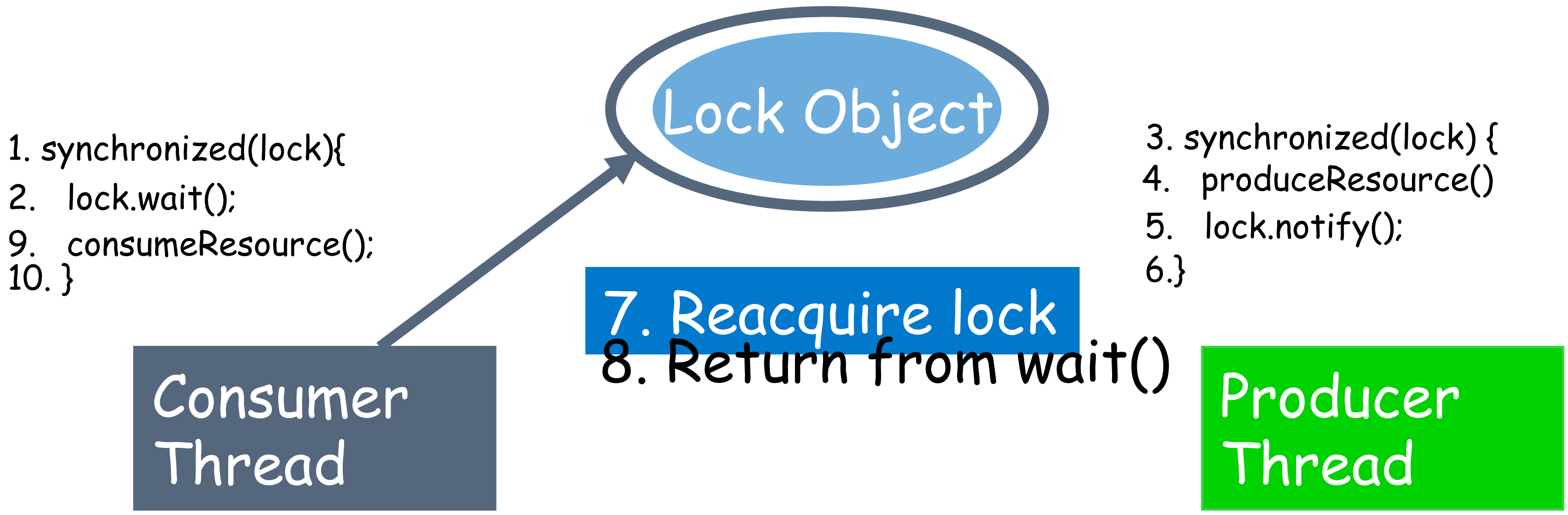
Wait/Notify Sequence



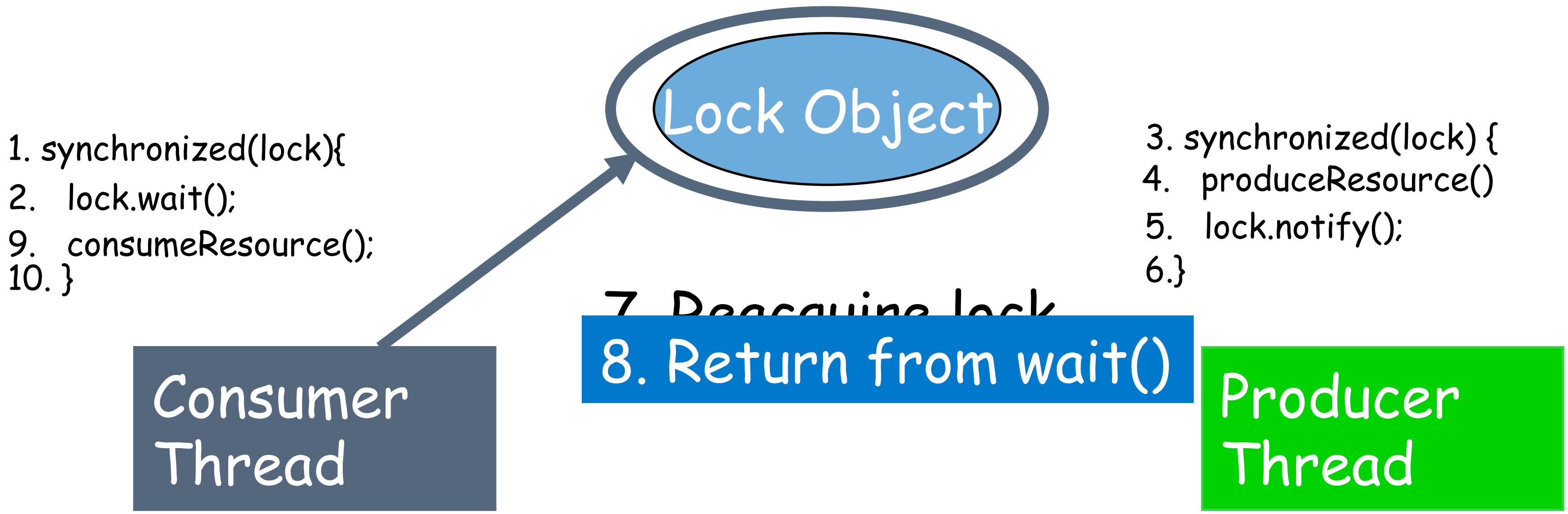
Wait/Notify Sequence



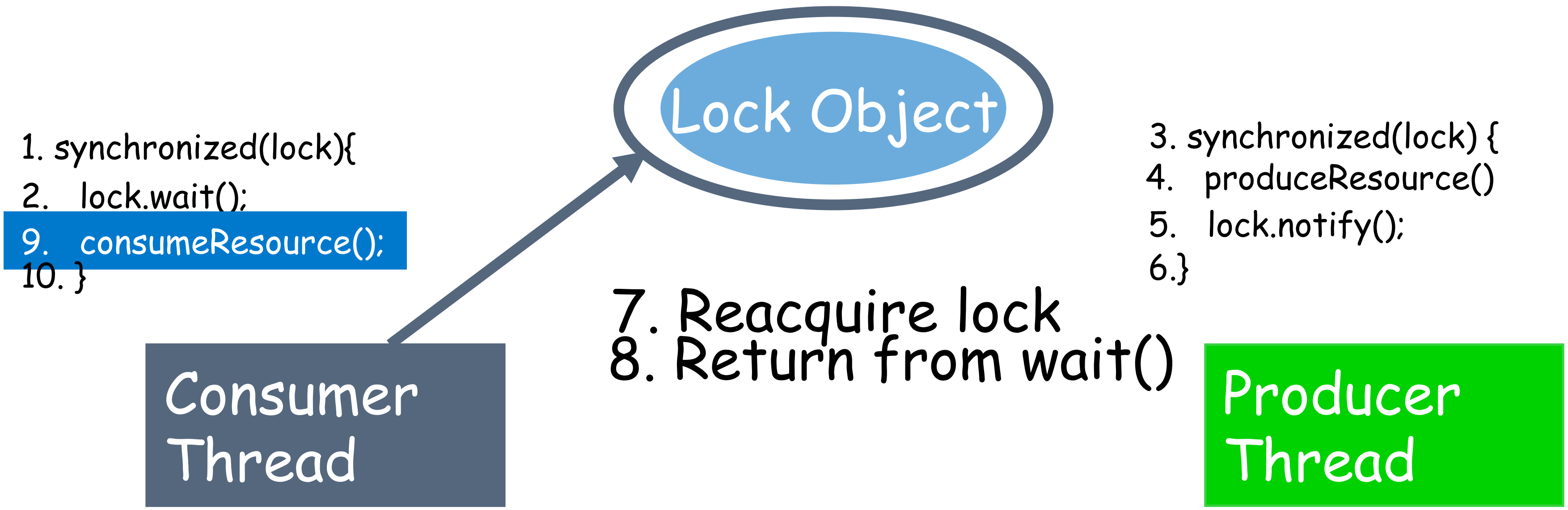
Wait/Notify Sequence



Wait/Notify Sequence



Wait/Notify Sequence



Wait/Notify Sequence

Lock Object

1. synchronized(lock){
2. lock.wait();
9. consumeResource();
10. }

3. synchronized(lock) {
4. produceResource()
5. lock.notify();
6.}

7. Reacquire lock
8. Return from wait()

Consumer Thread

Producer Thread

The Simpsons: Main Method

```
public class SimpsonsTest {

    public static void main(String[] args) {

        CookieJar jar = new CookieJar();

        Homer homer = new Homer(jar);
        Marge marge = new Marge(jar);

        new Thread(homer).start();
        new Thread(marge).start();
    }
}
```

The Simpsons: Homer

```
class Homer implements Runnable {
    CookieJar jar;

    public Homer(CookieJar jar) {
        this.jar = jar;
    }

    public void eat() {
        jar.getCookie("Homer");
        try {
            Thread.sleep((int)Math.random() * 500);
        } catch (InterruptedException ie) {}
    }

    public void run() {
        for (int i = 0 ; i < 5 ; i++) eat();
    }
}
```

The Simpsons: Marge

```
class Marge implements Runnable {
    CookieJar jar;

    public Marge(CookieJar jar) {
        this.jar = jar;
    }

    public void bake(int cookieNumber) {
        jar.putCookie("Marge", cookieNumber);
        try {
            Thread.sleep((int)Math.random() * 500);
        } catch (InterruptedException ie) {}
    }

    public void run() {
        for (int i = 0 ; i < 5 ; i++) bake(i);
    }
}
```

The Simpsons: CookieJar

```
class CookieJar {
    private volatile int contents;
    private volatile boolean available = false;

   public synchronized void getCookie(String who) {
        while (!available) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        available = false;
        notifyAll();
        System.out.println( who + " ate cookie "
                                   + contents);
   }
```

```
    public synchronized void putCookie(String who,
                                     int value) {
       while (available) {
            try {
                wait();
            } catch (InterruptedException e) { }
       }
       contents = value;
       available = true;
       System.out.println(who + " put cookie " +
            contents + ” in the jar");
       notifyAll();
    }
} /* end of class CookieJar */
```

The Simpsons: Output

Marge put cookie 0 in the jar
Homer ate cookie 0
Marge put cookie 1 in the jar
Homer ate cookie 1
Marge put cookie 2 in the jar
Homer ate cookie 2
Marge put cookie 3 in the jar
Homer ate cookie 3
Marge put cookie 4 in the jar
Homer ate cookie 4

Package `java.util.concurrent.Atomic`

- A small toolkit of classes that support lock-free thread-safe programming **on single variable**
- Very handy when we have perform thread-safe operations on Java primitive types **but without using any locks**
  - `counter ++`
  - `counter --`
  - `if(counter == expectedValue) { counter = newValue; }`
    - A.K.A Compare-and-Swap operation (CAS)
- Some classes in this package
  - `AtomicInteger`
  - `AtomicBoolean`
  - `AtomicLong`
  - .....

AtomicInteger Example

```
class Counter implements Runnable {
    AtomicInteger counter = new AtomicInteger(0);

    public void run() { counter.getAndIncrement(); }
    public static void main(String[] args)
                         throws InterruptedException {
        ExecutorService exec =
                    Executors.newFixedThreadPool(2);

        Counter task = new Counter();
        for(int i=0; i<1000; i++) {
            exec.execute(task);
        }

        if(!exec.isTerminated()) {
          exec.shutdown();
          exec.awaitTermination(5L,TimeUnit.SECONDS);
        }

        System.out.println(task.counter.get());
    }
}
```

- No need to use any locks
- Never deadlocks
- No need to mark counter as **`volatile`**
- We may use these too:
  - `addAndGet(int delta)`
  - `getAndAdd(int delta)`
  - `getAndIncrement()`
  - `incrementAndGet()`
  - Methods for decrementing:
    - `decrementAndGet()`
    - `getAndDecrement()`

Compare-and-Swap (CAS) Operations

- `if(counter==expectedValue) {counter = newValue;}`
  - A.K.A Compare-and-Swap operation (CAS)
- Modern CPUs have built-in support for atomic compare and swap operations
- Supported by classes in package java.util.concurrent.atomic
- E.g., AtomicInteger:
  - ```
    public final boolean compareAndSet(int expectedValue,
                                                            newValue)
    ```
    - Atomically sets the value to newValue if the current value == expectedValue
    - true if successful
    - false return indicates that the actual value was not equal to the expected value
    ```

Still Hungry for Multithreading Concepts?

- See you in Foundations of Parallel Programming (CSE502)
- Few interesting reads until you take CSE502 course
  - Kumar et. al., “Work-Stealing Without the Baggage”, OOPSLA 2012, Tucson, Arizona
  - Kumar et. al., “Friendly Barriers: Efficient Work-Stealing with Return Barriers”, VEE 2014, Salt Lake City, Utah
  - Kumar et. al., “Integrating asynchronous task parallelism and data-centric atomicity”, PPPJ 2016, Lugano, Switzerland