CSE201: Monsoon 2020
Advanced Programming

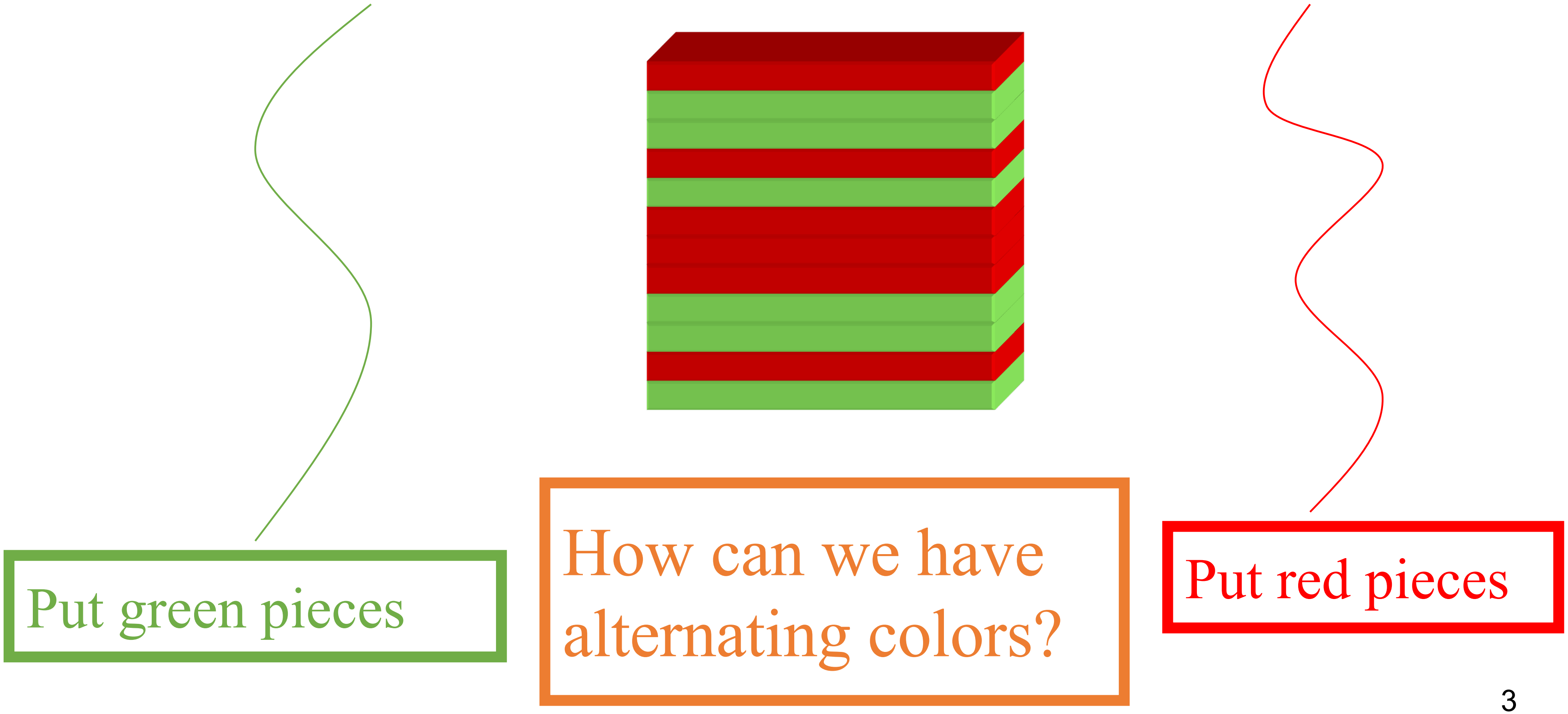
Lecture 21: Mutual Exclusion (Part-2)

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

Today’s Lecture

- Java memory model
  - Memory consistency
- Producer consumer problem

Race Condition



We are Still Missing Something…

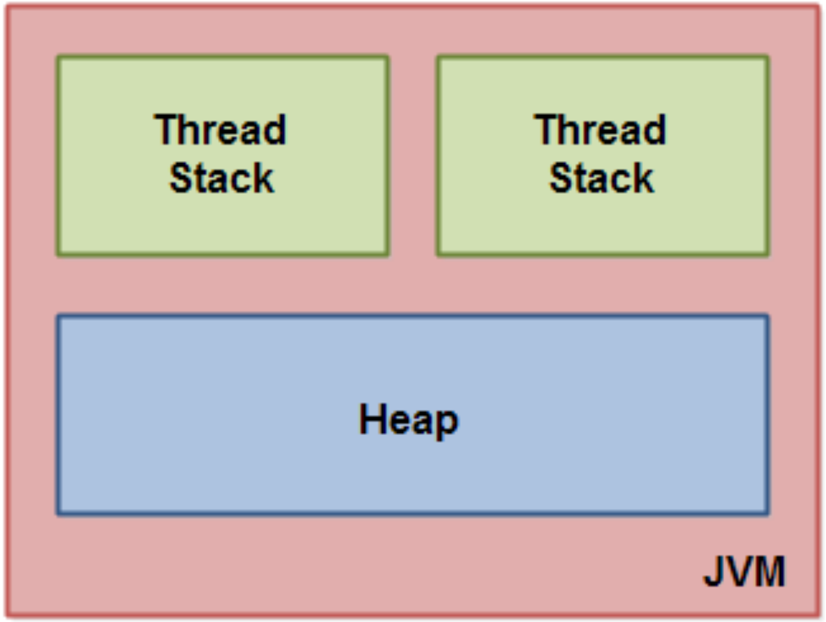
```
class Counter implements Runnable {
    static int counter = 0;
    static int turn = RED; //finals RED=0 and GREEN=1
    int me, other;
    public Counter(int c1, int c2) { me=c1; other=c2; }
    synchronized static void update(int me, int other) {
        if(counter<MAX && turn==me) {
            counter++; turn=other;
        }
    }
    public void run() {
        while(counter < MAX) {
            if(turn == me) {
                update(me, other);
            }
        }
    }
    public static void main(String args[])throws InterruptedException{
        Counter task1 = new Counter(RED, GREEN);
        Counter task2 = new Counter(GREEN, RED);
        Thread t1 = new Thread(task1); Thread t2 = new Thread(task2);
        t1.start(); t2.start(); t1.join(); t2.join();
    }
}
```


- Once in a while you would notice that this program will never terminate
    - Hint: launch this command in an infinite loop
- Using `synchronized` is just one part of the perfect solution
- Although there is no race on shared variables **`counter`** and **`color`**, the value of **`counter`** and **`color`** that a thread begins with may not be its last updated value

Java Memory Model (Quick Tour)

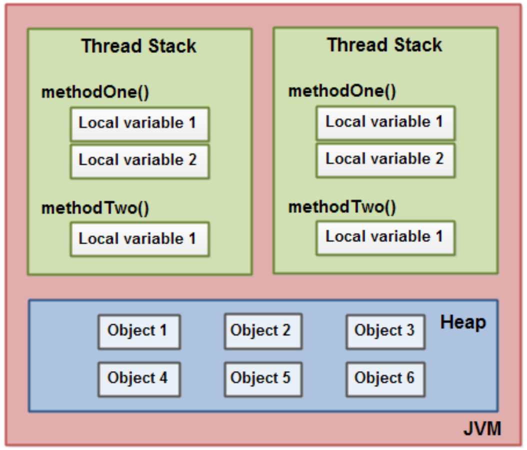
- What is it?
  - Defines how threads can interpret read/write from/to shared variables
- Why it is there?
  - Modern multicore processors have different layers of memory hierarchy
    - Caches and RAM
      - Improves the performance of memory operations (why?)
- Why you should care about it?
  - Hard to write correct multithreaded programs otherwise

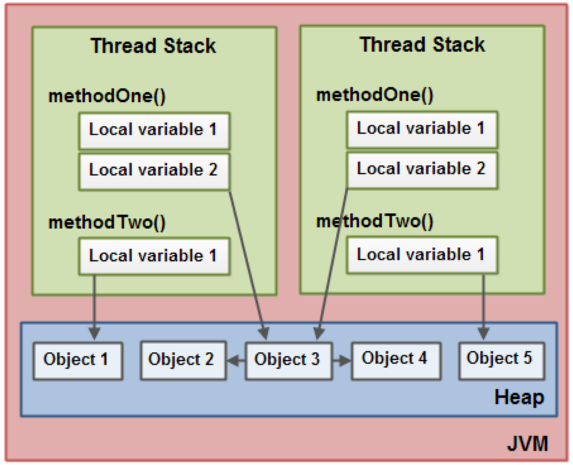
JVM Memory (1/2)



- Where are the thread local variables allocated?
  - Primitive types
    - Allocated on thread local stack
  - Objects (including object version of primitive types)
    - Allocated on heap

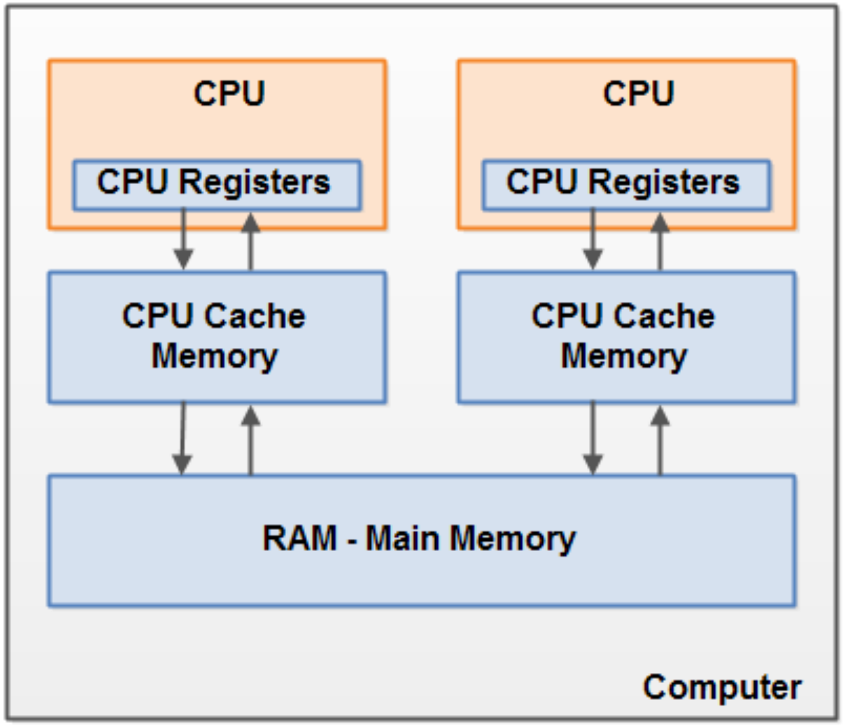
JVM Memory (2/2)



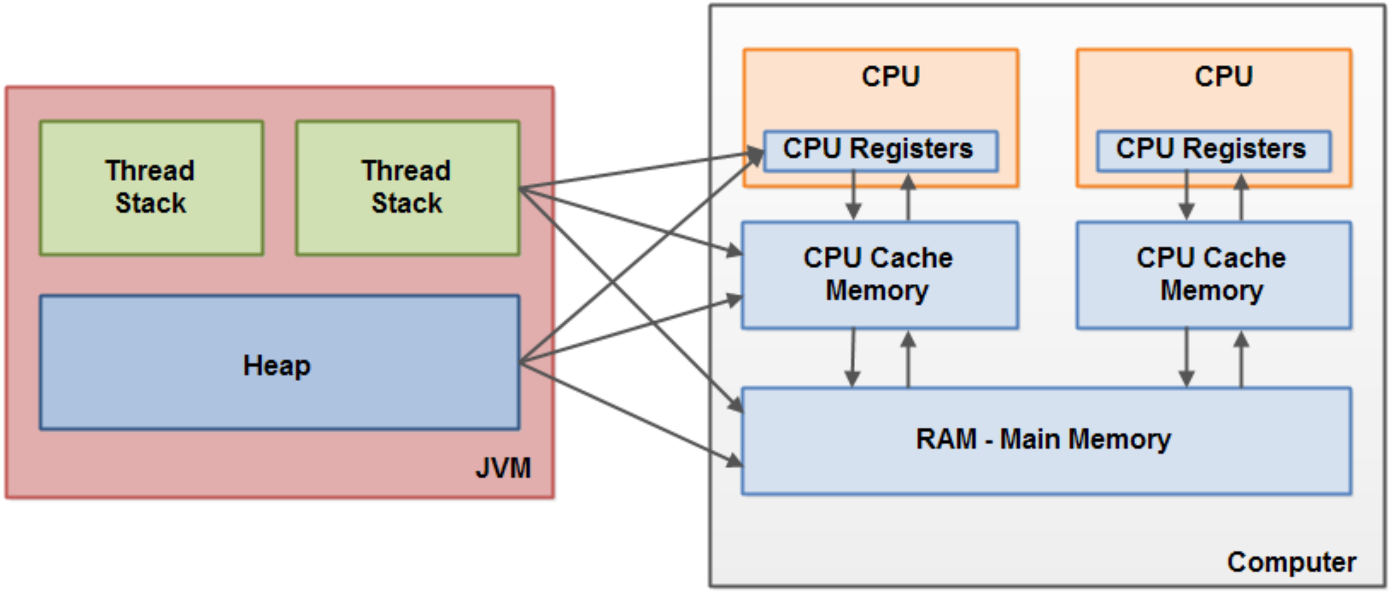


- Where are the thread local variables allocated?
  - Primitive types
    - Allocated on thread local stack
    - Thread T1 can pass it to thread T2, but modifications by one thread won't be visible to the other one
  - Objects (including object version of primitive types)
    - Allocated on heap
    - Thread T1 passes reference to thread T2
      - Both threads access object using thread local variables pointing to heap object
      - Changes done by either of the threads visible to others

Hardware Memory Architecture



Stack and Heap in Memory



- Java memory model is different than hardware memory architecture
  - Java understands the difference between threads, stack and heap
  - Hardware doesn't know anything about it
    - Treats stack and heap the same way
    - Problem:
      - Visibility of updates on shared variables by the threads?
      - Race condition during read/write

Statement Reordering

Thread-1	Thread-2
1: R2 = A;	3: R1 = B;
2: B = 1;	4: A = 2;

Compiler *is allowed to reorder the instructions in either thread, when this does not affect the execution of that thread in isolation*

- Initially A=B=0
  - Possible outputs
    - R2=0, R1=0
    - R2=2, R1=0
    - R2=0, R1=1
    - R2=2, R1=1

How to fix?

- "synchronize" missing for both these operations!
- In that case each thread will already read the updated value available on RAM instead of cache

Memory Consistency Issue (1/2)

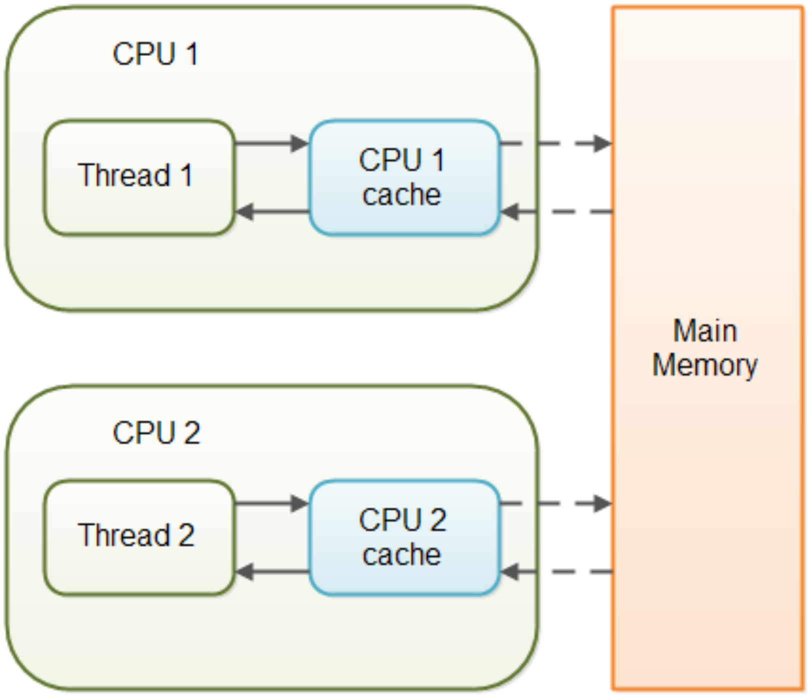
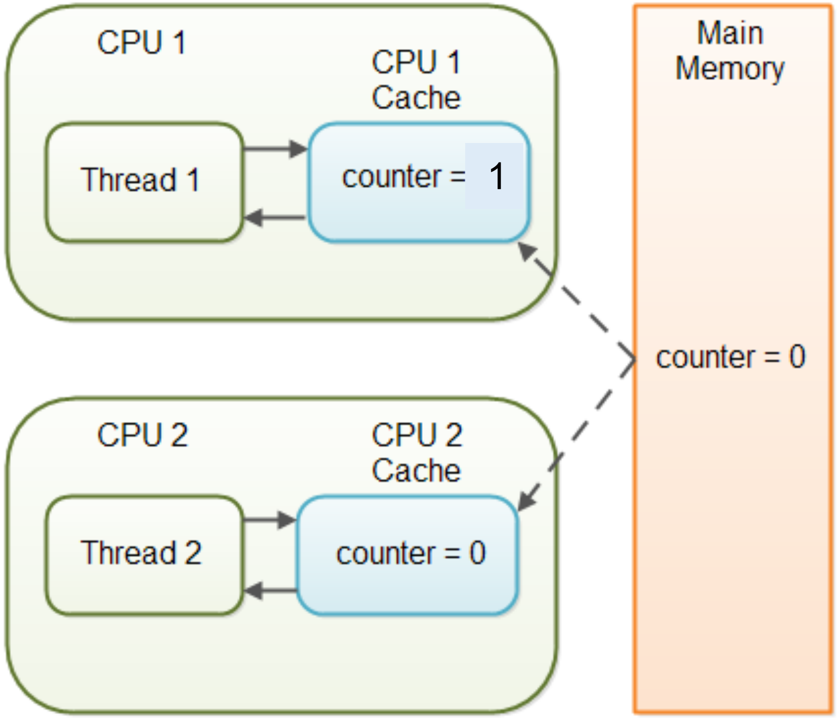


Fig. source: http://tutorials.jenkov.com/java-concurrency/volatile.html

- Modern computing systems uses multicore processors
- Each core has its own local cache
- For faster data access, memory referenced by a CPU is first copied from main memory (RAM) onto its local cache
- The updated memory content on cache is not immediately written back to RAM
    - This memory address might be referenced again in near future, hence immediately writing the cache content to RAM can hamper performance

Memory Consistency Issue (2/2)



- Imagine Counter example has two threads in its thread pool – Thread 1 on CPU1 and Thread 2 on CPU2
- Thread 1 increments counter from 0 to 1. This updated value resides on the cache of CPU1 and might not be immediately written back to the RAM
- Thread 2 now gets the chance to update the counter. It fetches the counter content from RAM but this is the old value (=0) and not the last updated value (=1)
- This is memory consistency error!

Fig. source: http://tutorials.jenkov.com/java-concurrency/volatile.html

The Correct Version of Counter Code

```
class Counter implements Runnable {
    volatile static int counter = 0;
    volatile static int turn = RED;

    ............
    ............
}
```

- Declare the counter as "**volatile**"
- Indication to JVM for storing the value of `counter` & color on RAM after every update to it
- With this each thread will always get the latest value of the `counter` & color
- Note that updates to variables inside synchronized block is anyway updated directly on the main memory at the end of synchronized block
  - Java memory model!

Creating an Object Lock

```
class Counter implements Runnable {
    volatile int counter = 0;

    private Object lock = new Object();
    public void run() {
        synchronized(lock) {
            counter++;
        }
    }

    ............
    ............
}
```

- We can also pass any object instance to synchronized

Monitor Locks are Reentrant

```
class Counter implements Runnable {
    volatile int counter = 0;

    public synchronized int value() { return counter; }

    public synchronized void run() {
        if(value() < 100) {
            counter++;
        }
    }

    ............
    ............
}
```

- Both value() and run() are synchronized methods
- Monitor locks are **reentrant** in Java
    - Same thread can recursively take the same lock
- Once a thread has taken a monitor lock then any further request by this same thread to reacquire the same monitor lock is redundant
- Monitor lock is released only after exiting the oldest synchronized block

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 thread who owns the lock
  - Not possible for a thread to decline waiting for the lock if its unavailable

The Producer Consumer Problem

- We need to synchronize 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

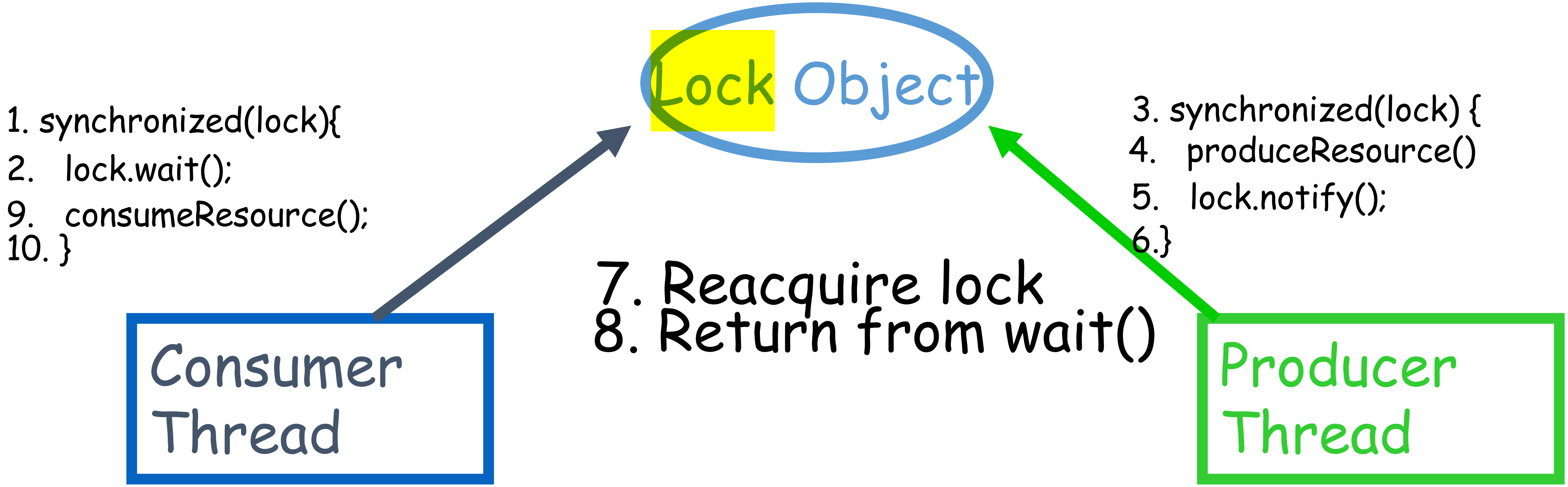
```
1. synchronized(lock){
2.   lock.wait();
9.   consumeResource();
10. }
```

Consumer

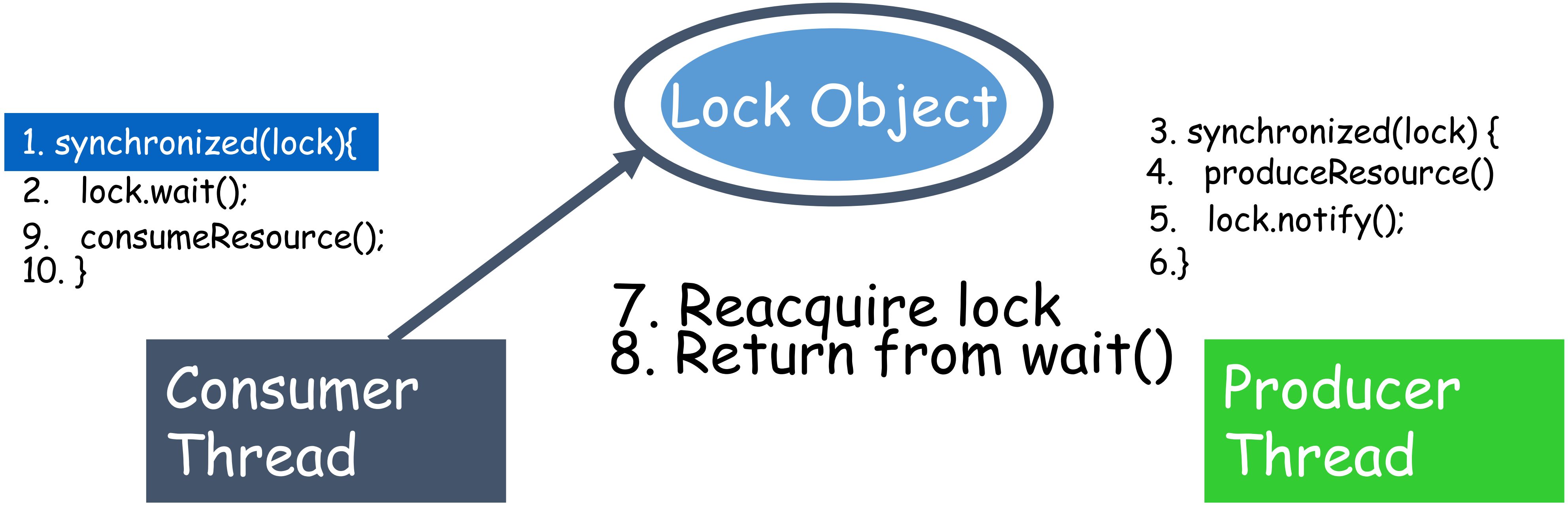
```
3. synchronized(lock) {
4.   produceResource()
5.   lock.notify();
6.}
```

Producer

Wait/Notify Sequence



Wait/Notify Sequence



Wait/Notify Sequence

Lock Object

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

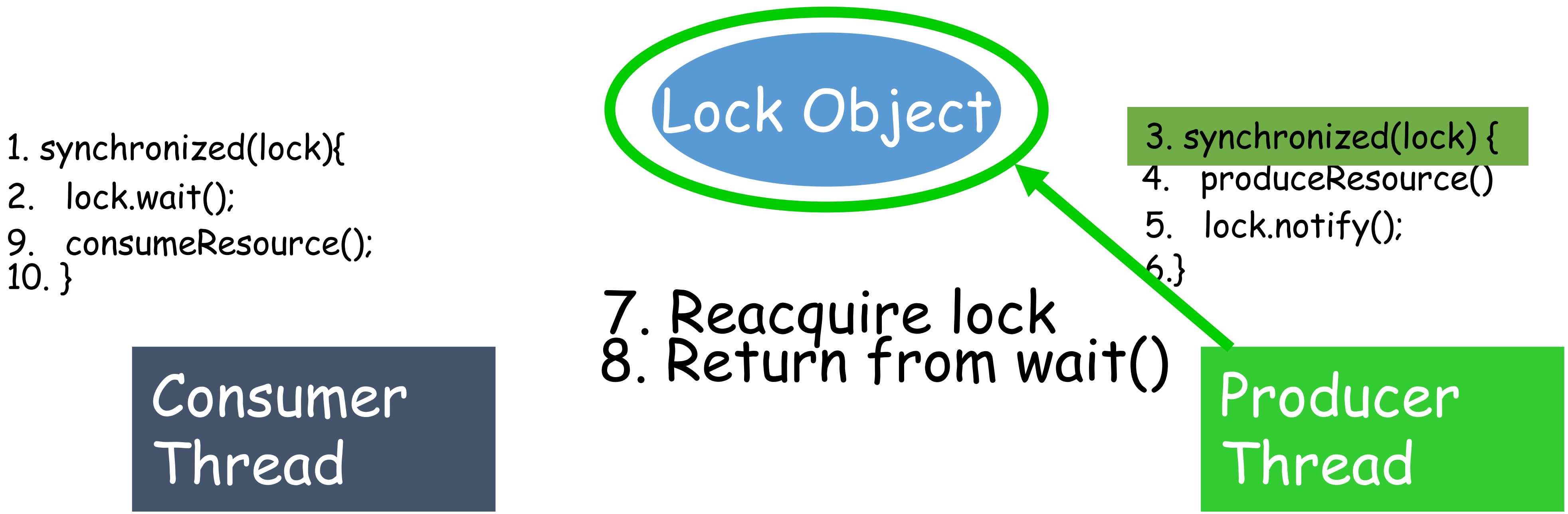
```
3. synchronized(lock) {
4.   produceResource()
5.   lock.notify();
6.}
```

7. Reacquire lock
8. Return from wait()

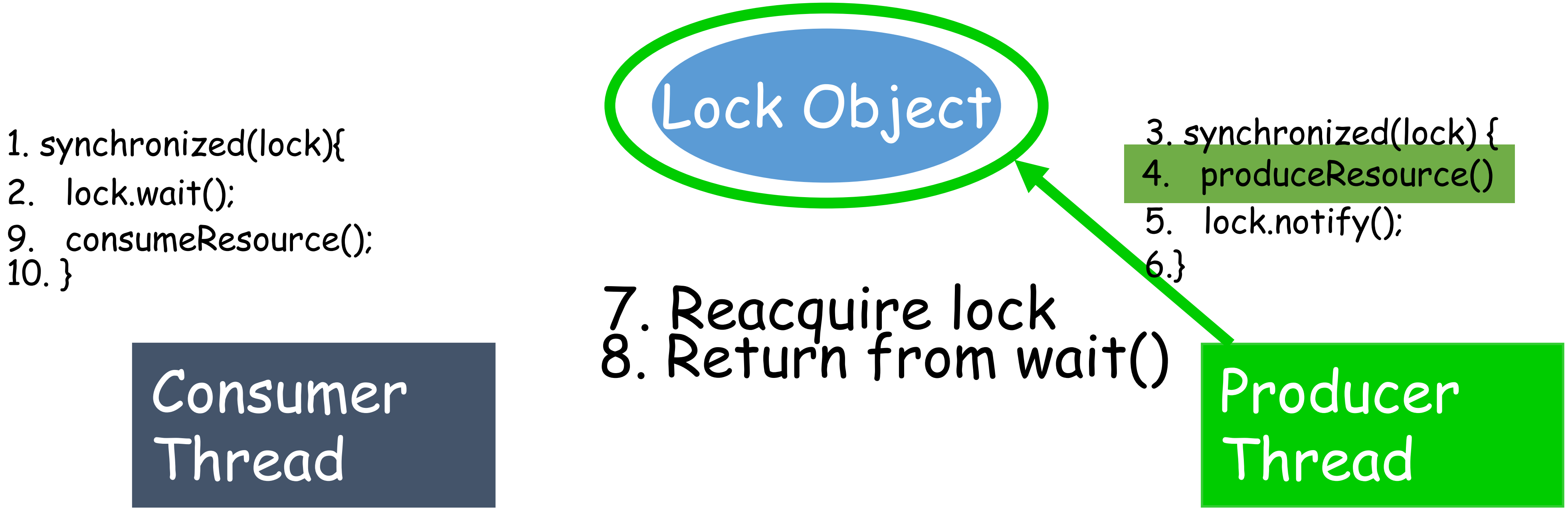
Consumer Thread

Producer Thread

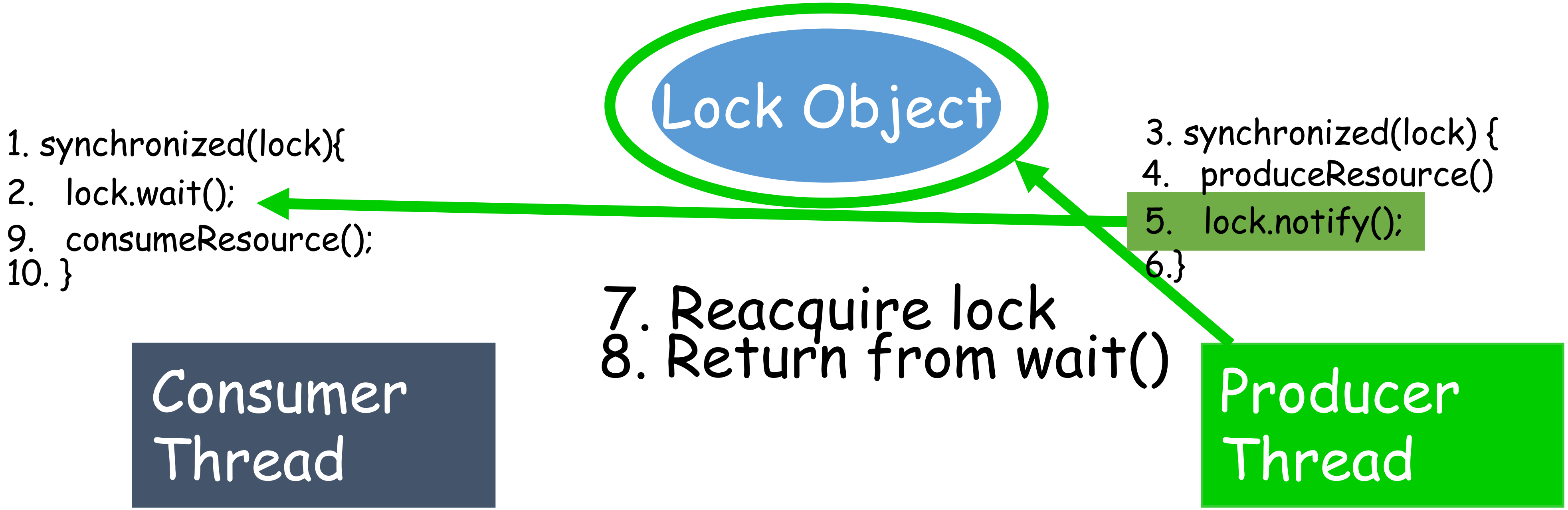
Wait/Notify Sequence



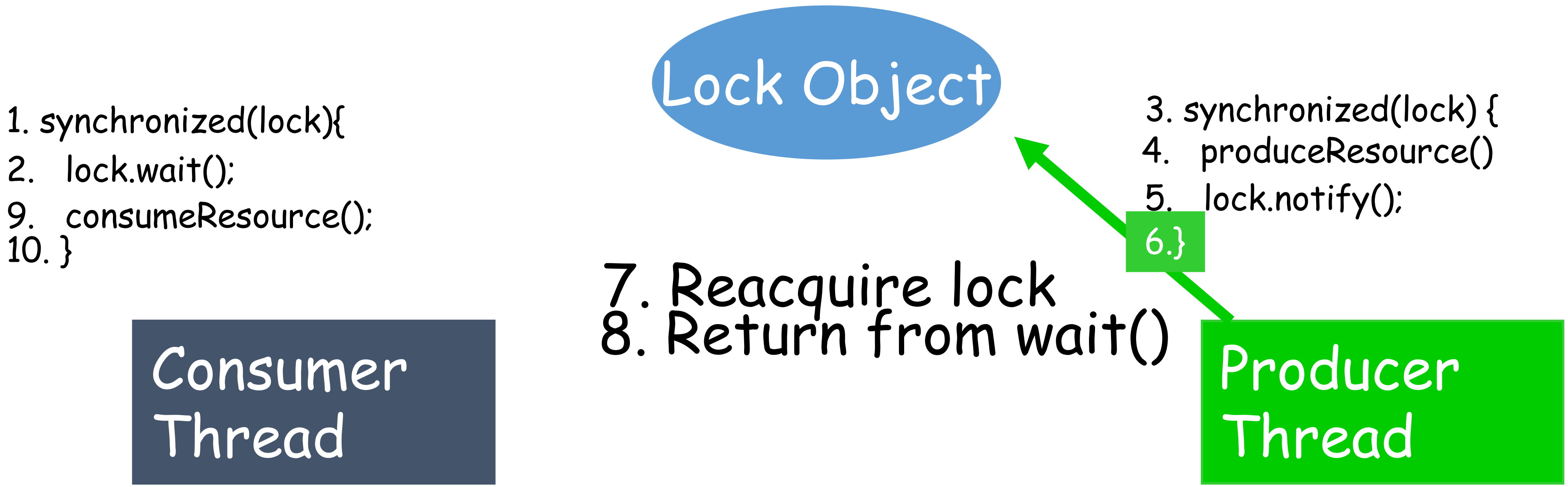
Wait/Notify Sequence



Wait/Notify Sequence



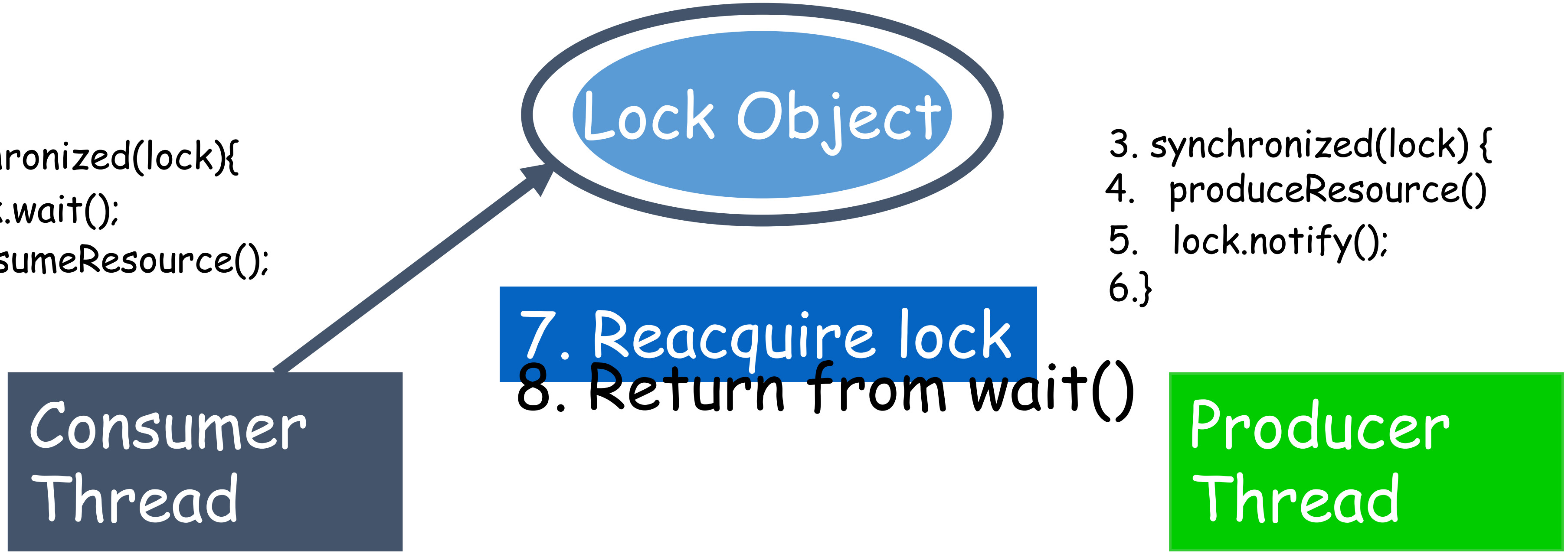
Wait/Notify Sequence



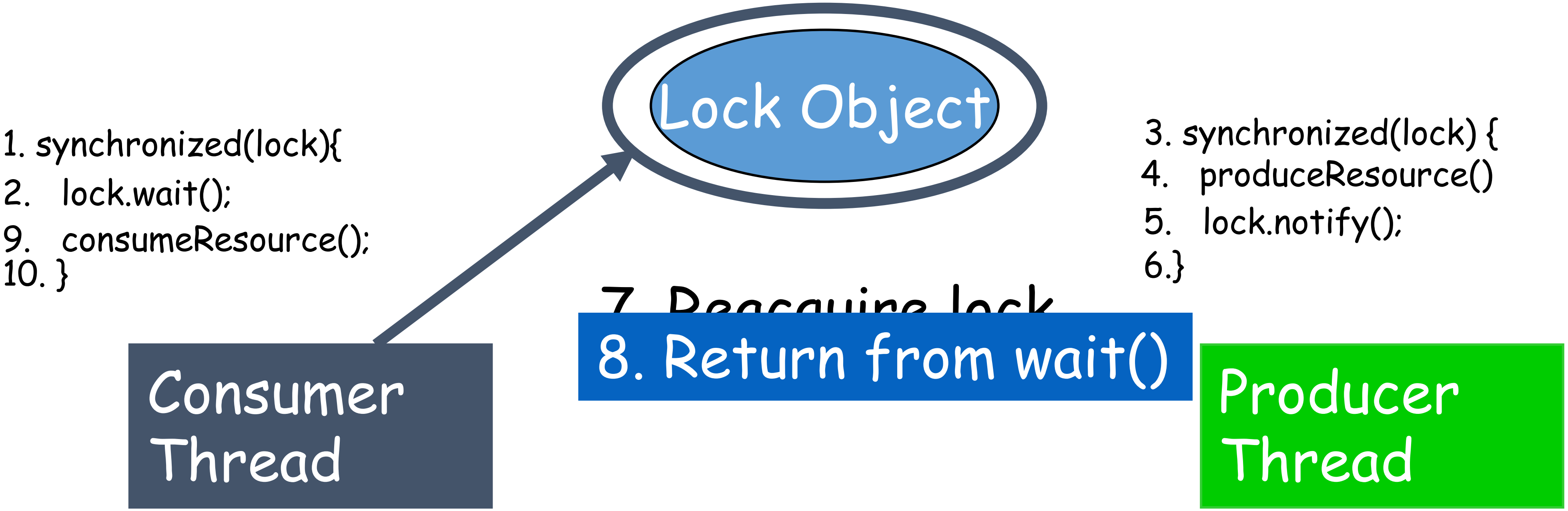
Wait/Notify Sequence

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

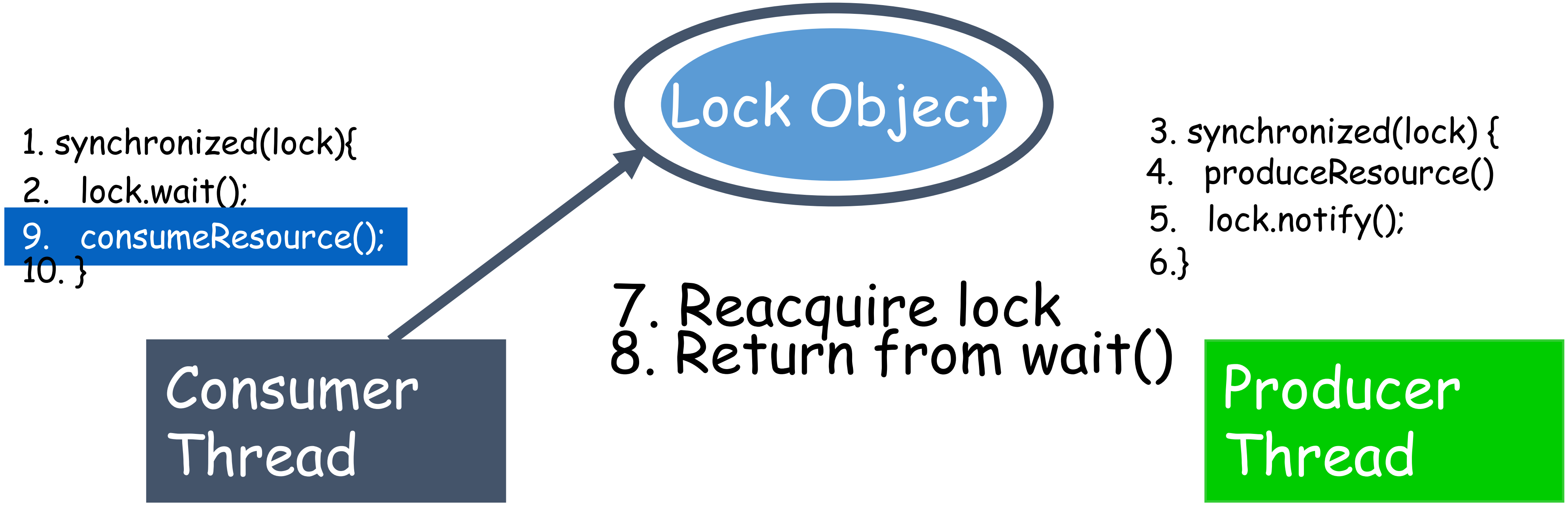
3. synchronized(lock) {
4. produceResource()
5. lock.notify();
6.}



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

Next Lecture

- Introduction to design patterns
  - *Beginning of last remaining topic in CSE201*
- Quiz next Friday at 4.15pm
- Assignment on multithreading next week