CSE201: Monsoon 2020
Advanced Programming

Lecture 20: Mutual Exclusion

Raghava Mutharaju (Section-B)

Vivek Kumar (Section-A)

CSE, IIIT-Delhi

raghava.mutharaju@iiitd.ac.in

Last Lecture

- Think tasks, not threads
  - Tasks are logical unit of work and are lightweight than thread

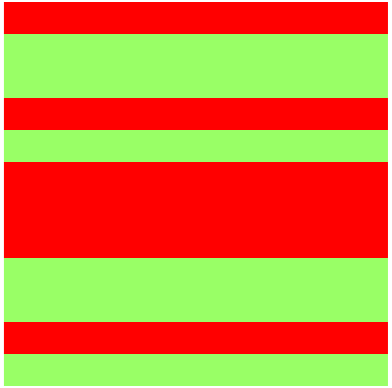
```java
public class ArraySum implements Runnable {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
                                throws InterruptedException {
      int size; int[] array; //allocated (size) & initialized
      ExecutorService exec = Executors.newFixedThreadPool(2);
      ArraySum left = new ArraySum(array, 0, size/2);
      ArraySum right = new ArraySum(array, size/2, size);
      exec.execute(left); exec.execute(right);
      if(!exec.isTerminated()) {
          exec.shutdown();
          exec.awaitTermination(5L, TimeUnit.SECONDS);
      }
      int result = left.getResult() + right.getResult();
    }
}
```

```java
import java.util.concurrent.*;

public class Fibonacci extends RecursiveTask<Integer> {
    int n;
    public Fibonacci(int _n) { n=_n; }

    public Integer compute() {
        if(n<2) return n;

        Fibonacci left = new Fibonacci(this.n-1);
        Fibonacci right = new Fibonacci(this.n-2);
        left.fork();
        return right.compute() + left.join();
    }
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool(2);
        Fibonacci task = new Fibonacci(40);
        int result = pool.invoke(task);
    }
}
```

Today’s Lecture

- Race conditions
- Mutual exclusion
- Monitor locks
- Memory consistency
- Producer consumer problem

Race Condition



Put green pieces

How can we have alternating colors?

Put red pieces

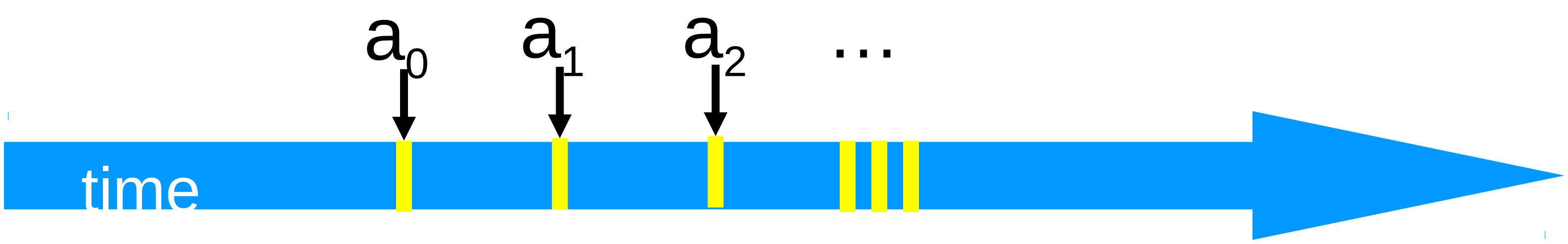
Mutual Exclusion

- ***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.
    - o *Otherwise it results in a race condition!*

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Threads

- A ***thread*** A is (formally) a sequence a_0, a_1, ... of events
  - Notation: $a_0 \rightarrow a_1$ indicates order



Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Example Thread Events

- Assign to shared variable
- Assign to local variable
- Invoke method
- Return from method
- Lots of other things …

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Concurrent Execution Over Multiple Threads

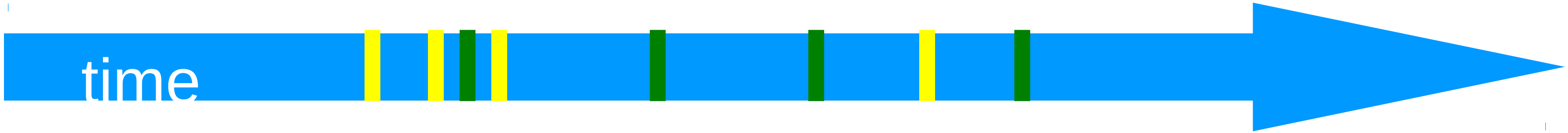
- Thread A

- Thread B

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Interleavings

- Events of two or more threads
  - Interleaved
  - Not necessarily independent (why?)



Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Question

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

    public void run() { counter++; }
    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);
    }
}
```

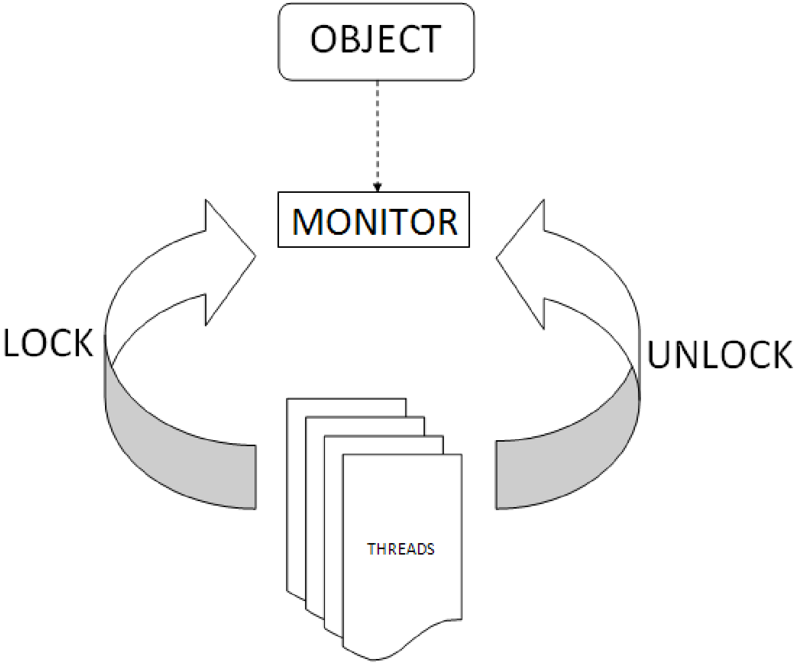
- What will be the output of this program?
  - Race on counter!
  - Buggy code and you will see different answers in different runs

Implementing Mutual Exclusion

```
class Counter implements Runnable {
    int counter = 0;
    // Both the versions of run method below is
correct
    public synchronized void run() { counter++; }
 /* public void run() { synchronized(this) {counter+
+;} } */
    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);
    }
```

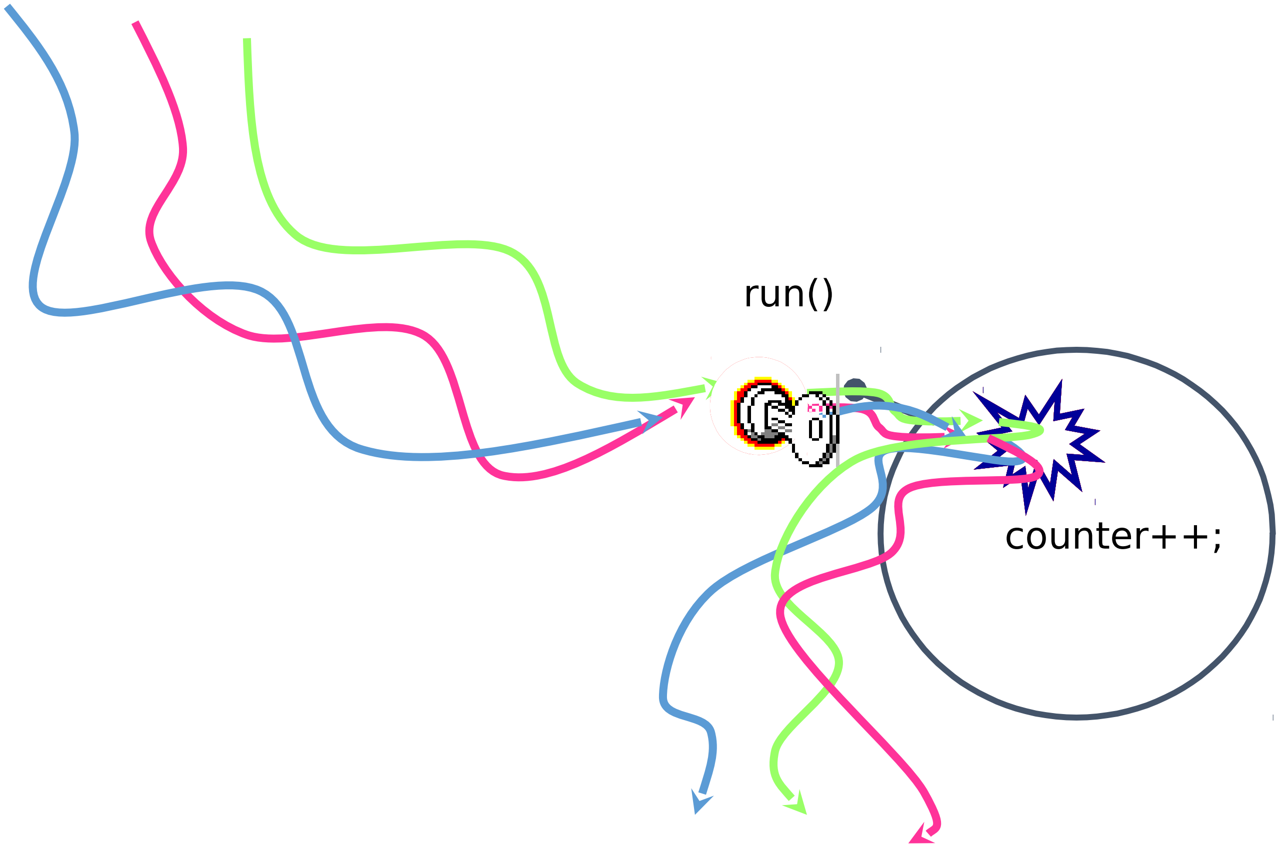
- Critical section
  - The `synchronized` methods (or block) define the critical sections
  - By using `synchronized` keyword we achieved mutual exclusion
    - Now let's analyze this

Monitors



- Each object has a "**monitor**" that is a token used to determine which application **thread** has control of a particular **object** instance
- In execution of a synchronized method (or block), access to the object monitor must be gained before the execution
- Access to the object monitor is queued
- Entering a monitor is also referred to as **locking** the monitor, or **acquiring ownership** of the monitor
- If a thread *A* tries to acquire ownership of a monitor and a different thread has already entered the monitor, the current thread (*A*) must wait until the other thread leaves the monitor

Analyzing our Counter Increment Example



- Only one thread can get the “key” to enter the “run” method i.e., take a lock on monitor
- Rest all threads will be queued to get the lock on monitor
- **Note**: There is no guarantee for fairness, i.e. longest waiting thread need not always get the lock first

Static Synchronized Methods

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

    public synchronized static void increment()
{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(Counter.counter);
    }
```

- Marking a static method as `synchronized`, **associates a monitor with the class** itself
- The execution of synchronized static methods of the same class is mutually exclusive

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();
    }}
```

- ~~This program will never terminate~~
- 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

Memory Consistency Issue (1/2)

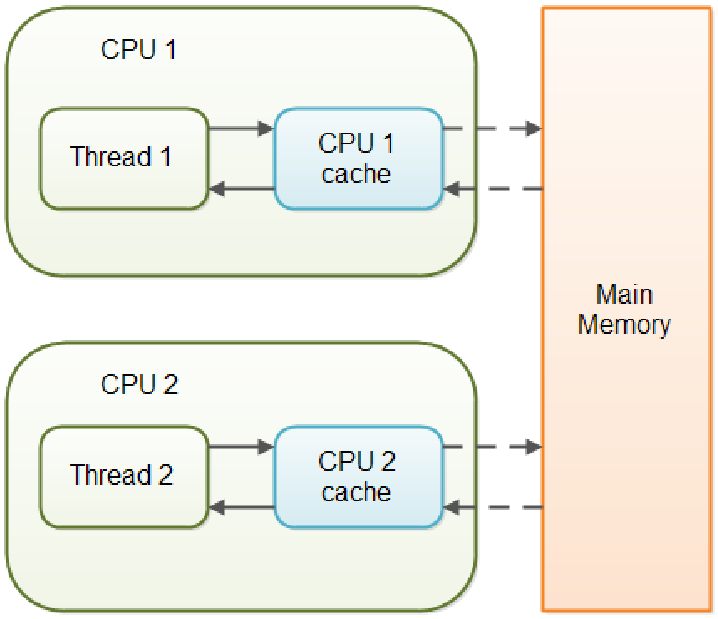
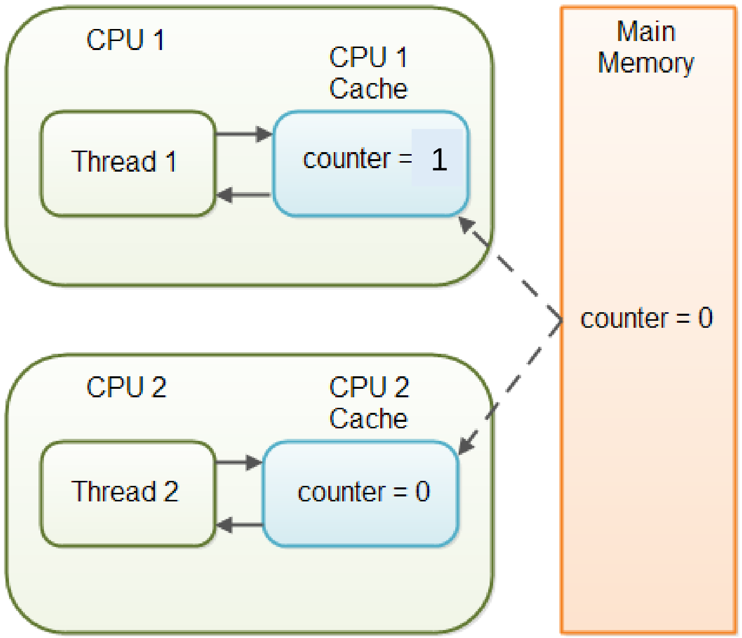


Fig. source: http://tutorials.jenkov.com/java-concurrency/volatile.html

- 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

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

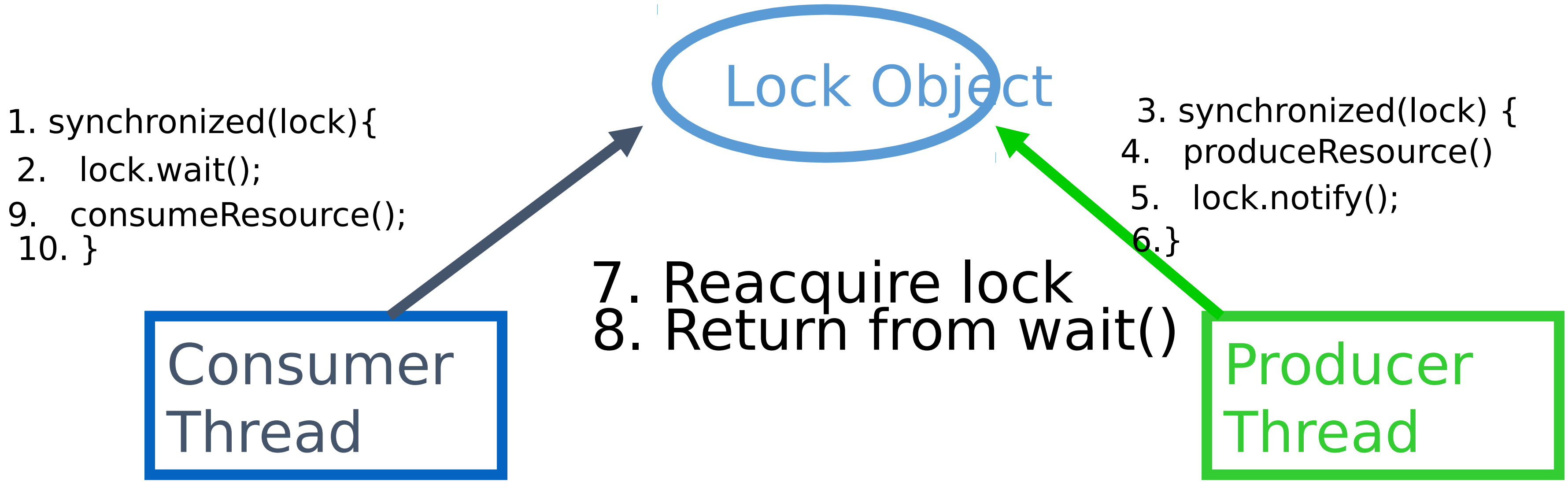
```
1. synchronized(lock){
 2.   lock.wait();
9.   consumeResource();
 10. }
```

Consumer

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

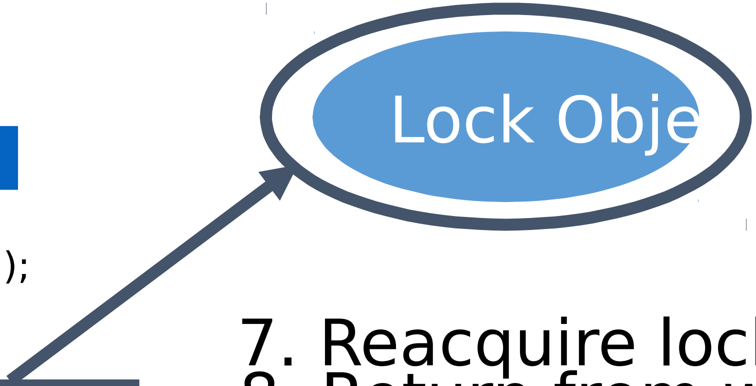
Producer

Wait/Notify Sequence



Wait/Notify Sequence

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

Lock Obje

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

Lock Obje

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

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

7. Reacquire lock
8. Return from wait()

Consumer Thread

Producer Thread

Wait/Notify Sequence

Lock Obje

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

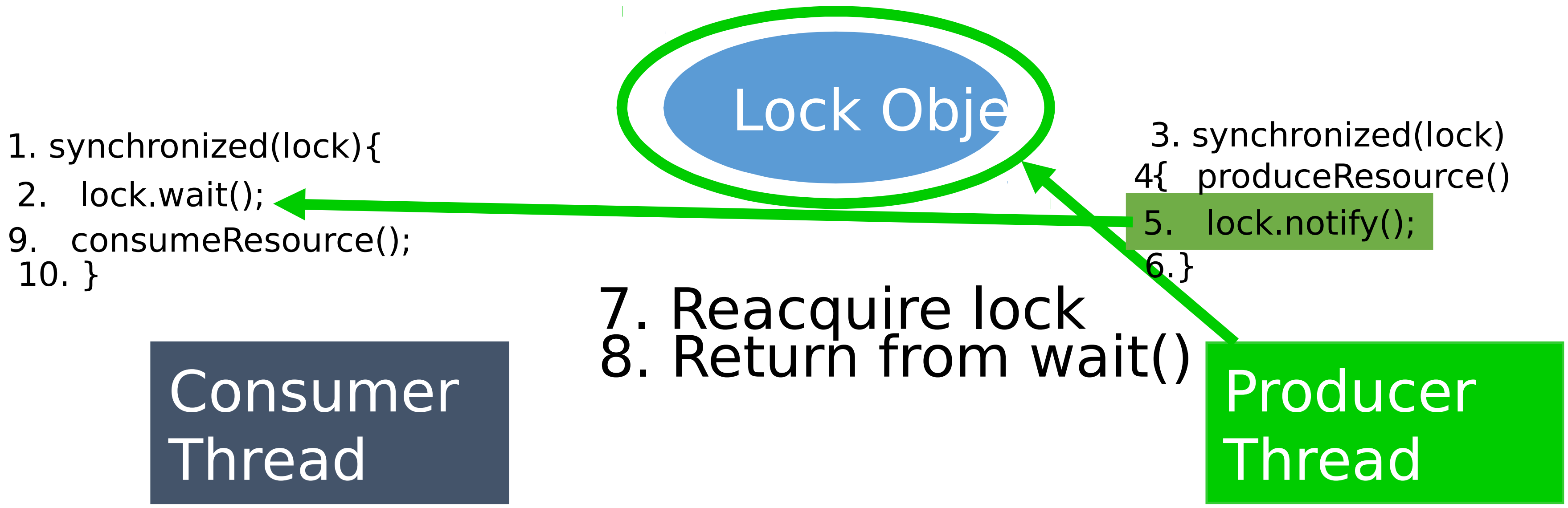
3. synchronized(lock)
4{ produceResource()
5. lock.notify();
6.}

7. Reacquire lock
8. Return from wait()

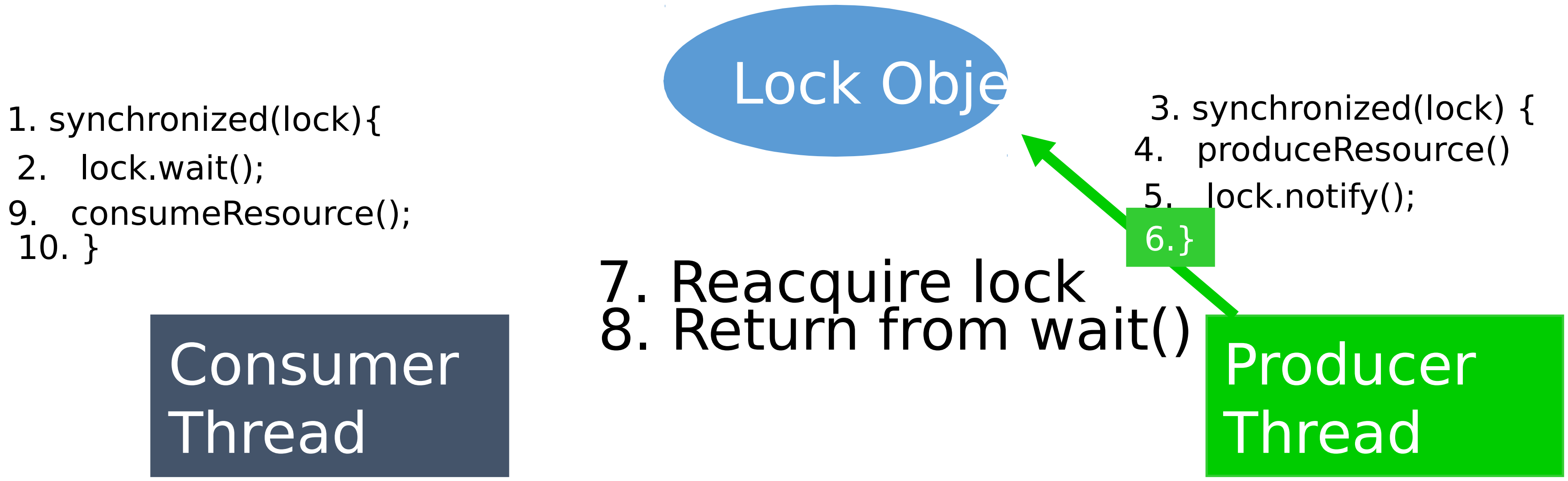
Consumer Thread

Producer Thread

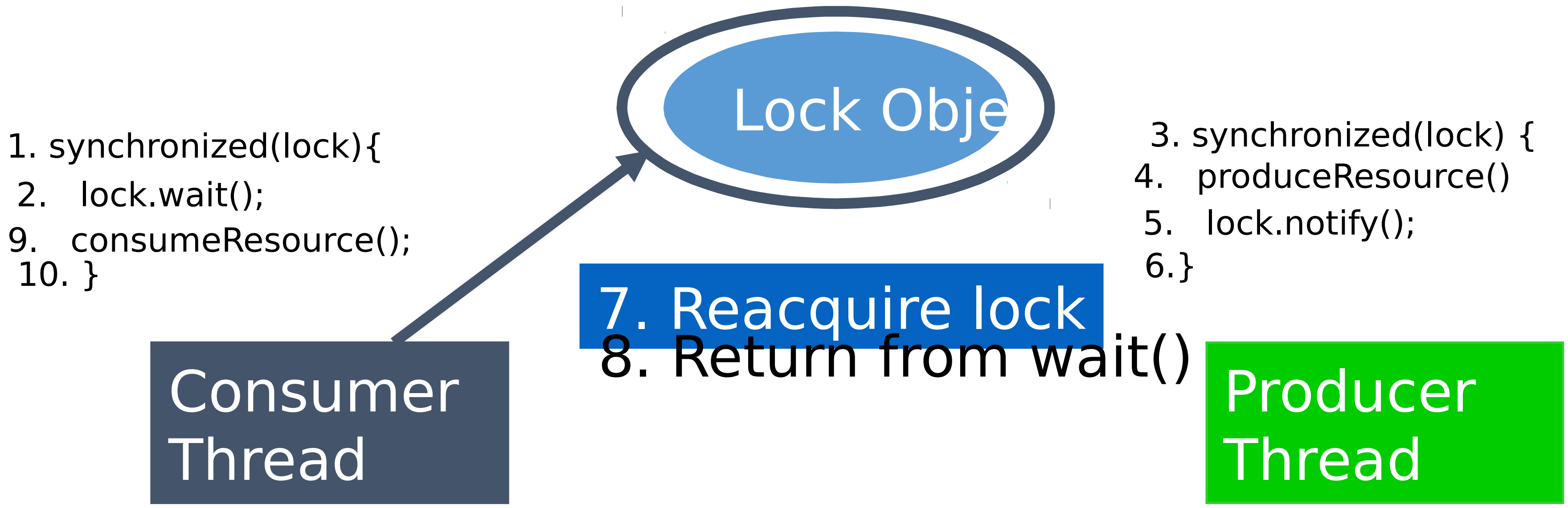
Wait/Notify Sequence



Wait/Notify Sequence

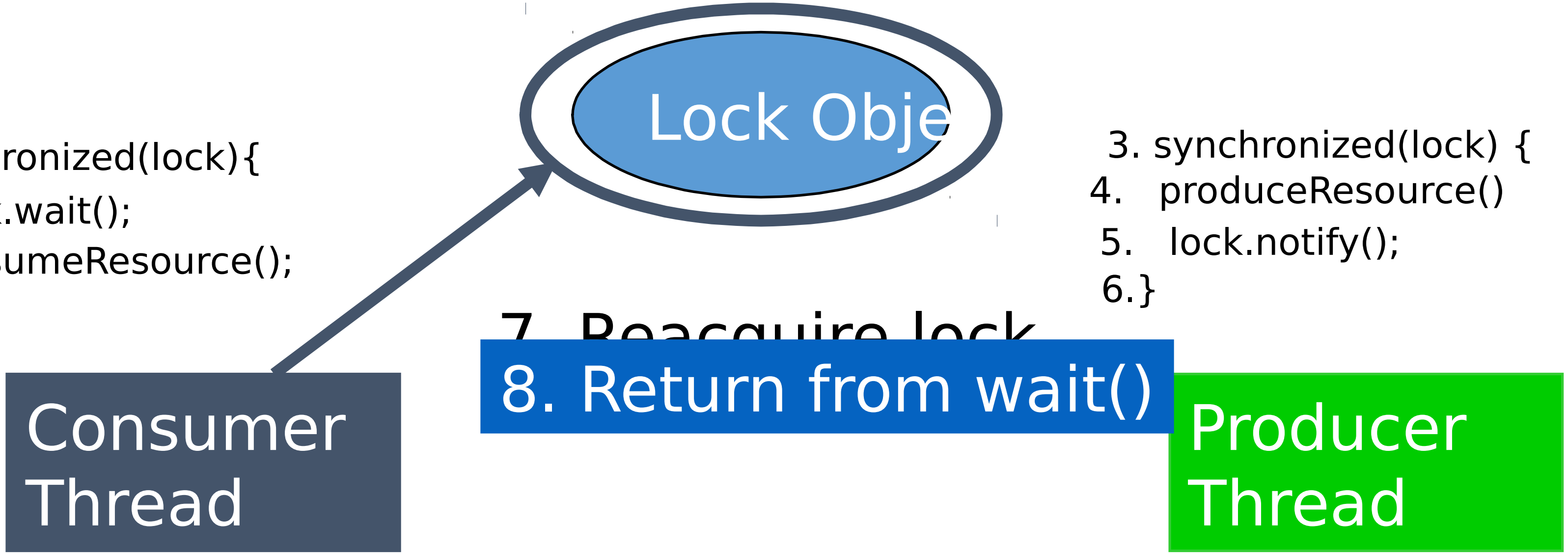


Wait/Notify Sequence



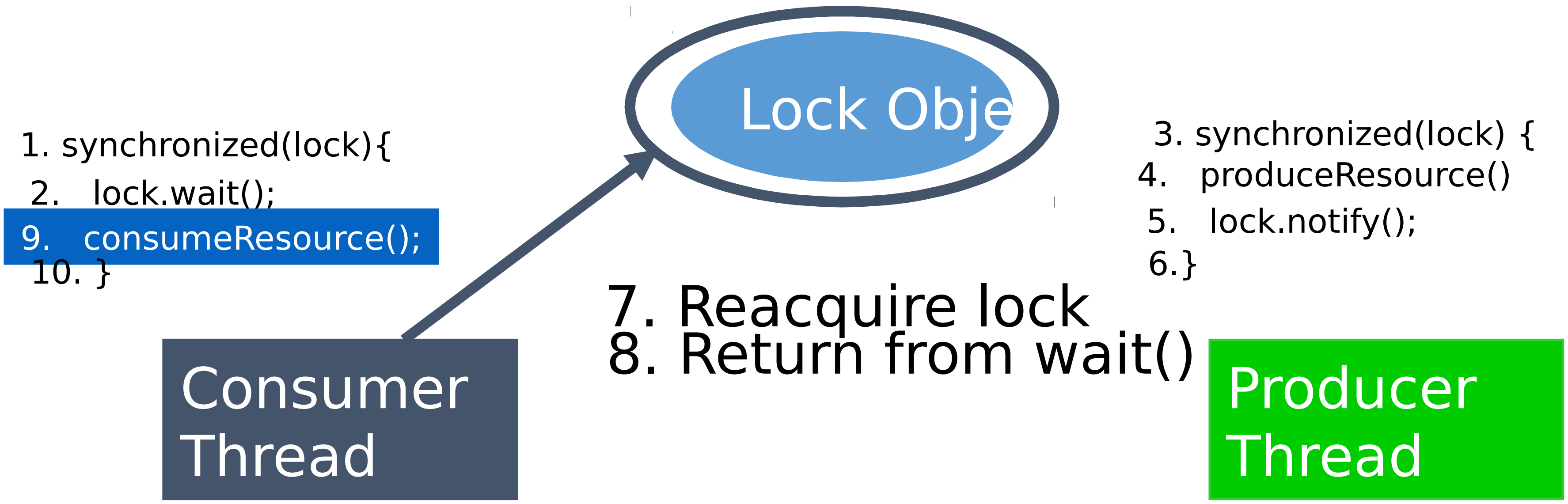
Wait/Notify Sequence

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



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

Wait/Notify Sequence



Wait/Notify Sequence

Lock Obje

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*