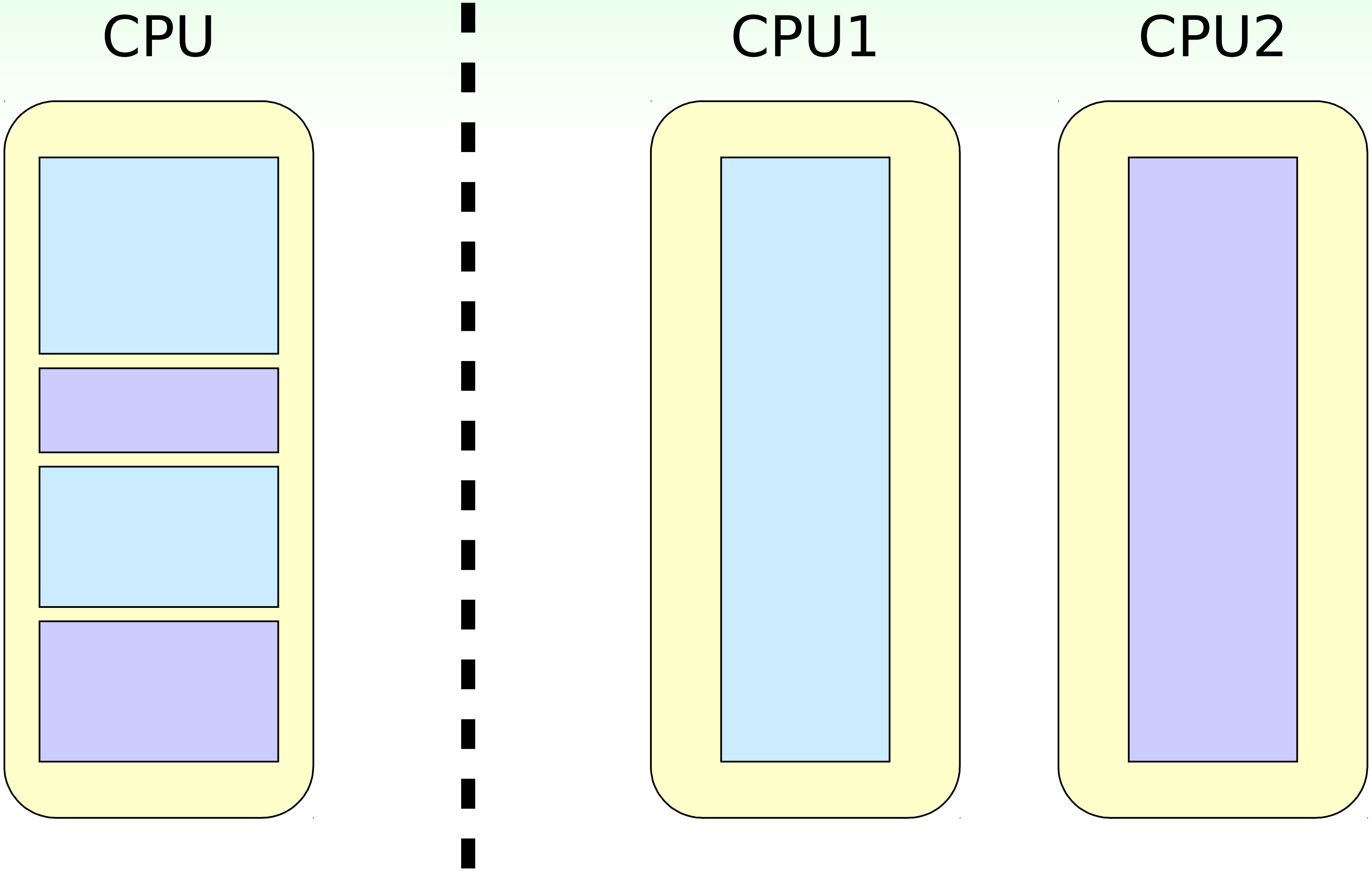
# Java Threads

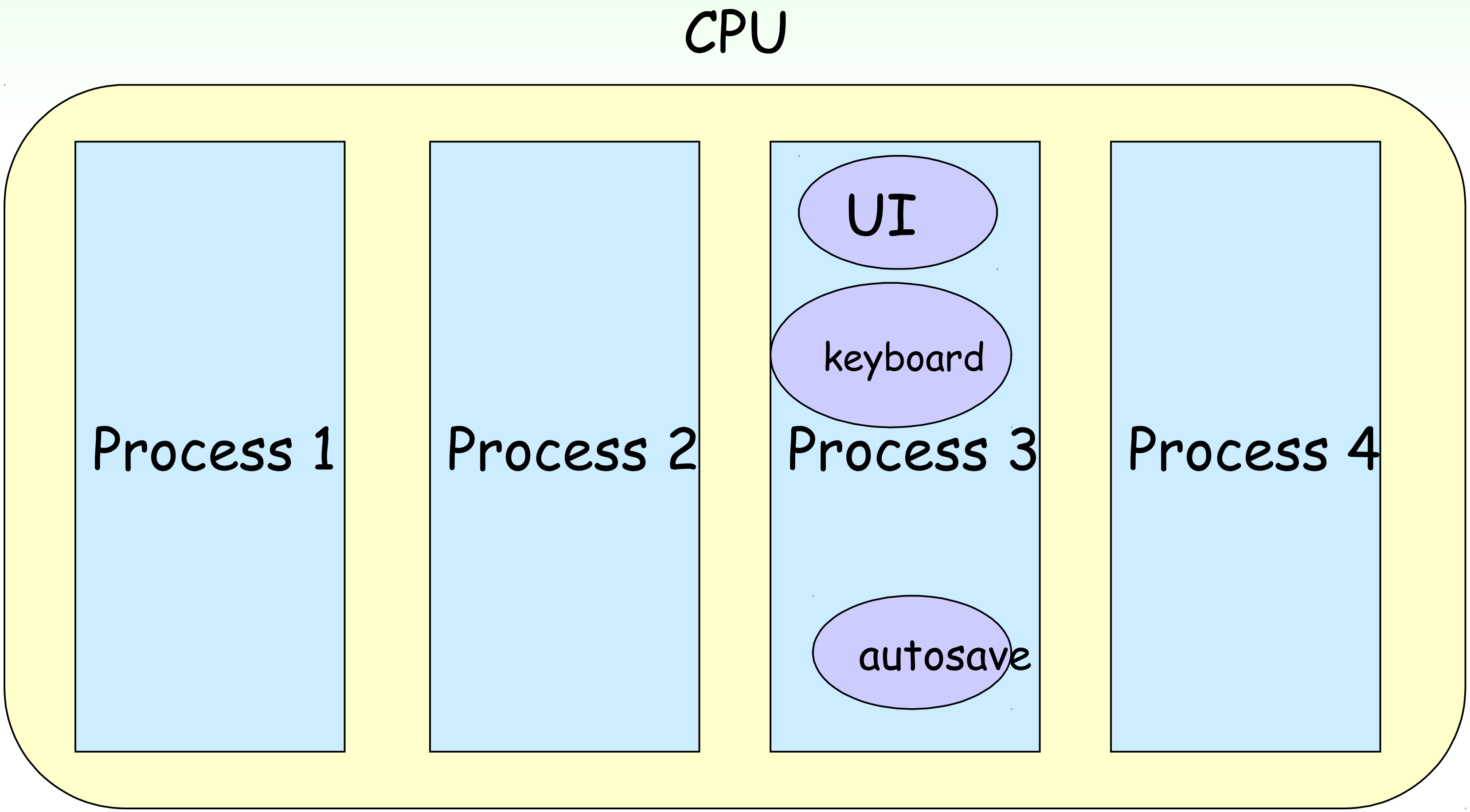
# Multitasking and Multithreading

- Multitasking refers to a computer's ability to perform multiple jobs concurrently
  - more than one program are running concurrently, e.g., UNIX
- A thread is a single sequence of execution within a program
- Multithreading refers to multiple threads of control within a single program
  - each program can run multiple threads of control within it, e.g., Web Browser

# Concurrency vs. Parallelism



# Threads and Processes



# What are Threads Good For?

- To maintain responsiveness of an application during a long running task.
- To enable cancellation of separable tasks.
- Some problems are intrinsically parallel.
- To monitor status of some resource (DB).
- Some APIs and systems demand it: Swing.

# Application Thread

- When we execute an application:
  - The JVM creates a Thread object whose task is defined by the **main()** method
  - It starts the thread
  - The thread executes the statements of the program one by one until the method returns and the thread dies

# Multiple Threads in an Application

- Each thread has its private run-time stack
- If two threads execute the same method, each will have its own copy of the local variables the methods uses
- However, all threads see the same dynamic memory (heap)
- Two different threads can act on the same object and same static fields concurrently

# Thread Methods

**void start()**

- Creates a new thread and makes it runnable
- This method can be called only once

**void run()**

- The new thread begins its life inside this method

**void stop()** (deprecated)

- The thread is being terminated

# Thread Methods

- **yield()**
  - Causes the currently executing thread object to temporarily pause and allow other threads to execute
  - Allow only threads of the same priority to run
- **sleep(int *m*)/sleep(int *m*,int *n*)**
  - The thread sleeps for *m* milliseconds, plus *n* nanoseconds

# Creating Threads

- There are two ways to create our own **Thread** object
    1. Subclassing the **Thread** class and instantiating a new object of that class
    2. Implementing the **Runnable** interface
- In both cases the **run()** method should be implemented

# Extending Thread

```
public class ThreadExample extends Thread {
   public void run () {
      for (int i = 1; i <= 100; i++) {
         System.out.println(“Thread: ” + i);
      }
   }
}
```

# Implementing Runnable

```
public class RunnableExample implements Runnable {
    public void run () {
        for (int i = 1; i <= 100; i++) {
            System.out.println ("Runnable: " + i);
        }
    }
}
```

# A Runnable Object

- The Thread object's **run()** method calls the Runnable object's **run()** method
- Allows threads to run inside any object, regardless of inheritance

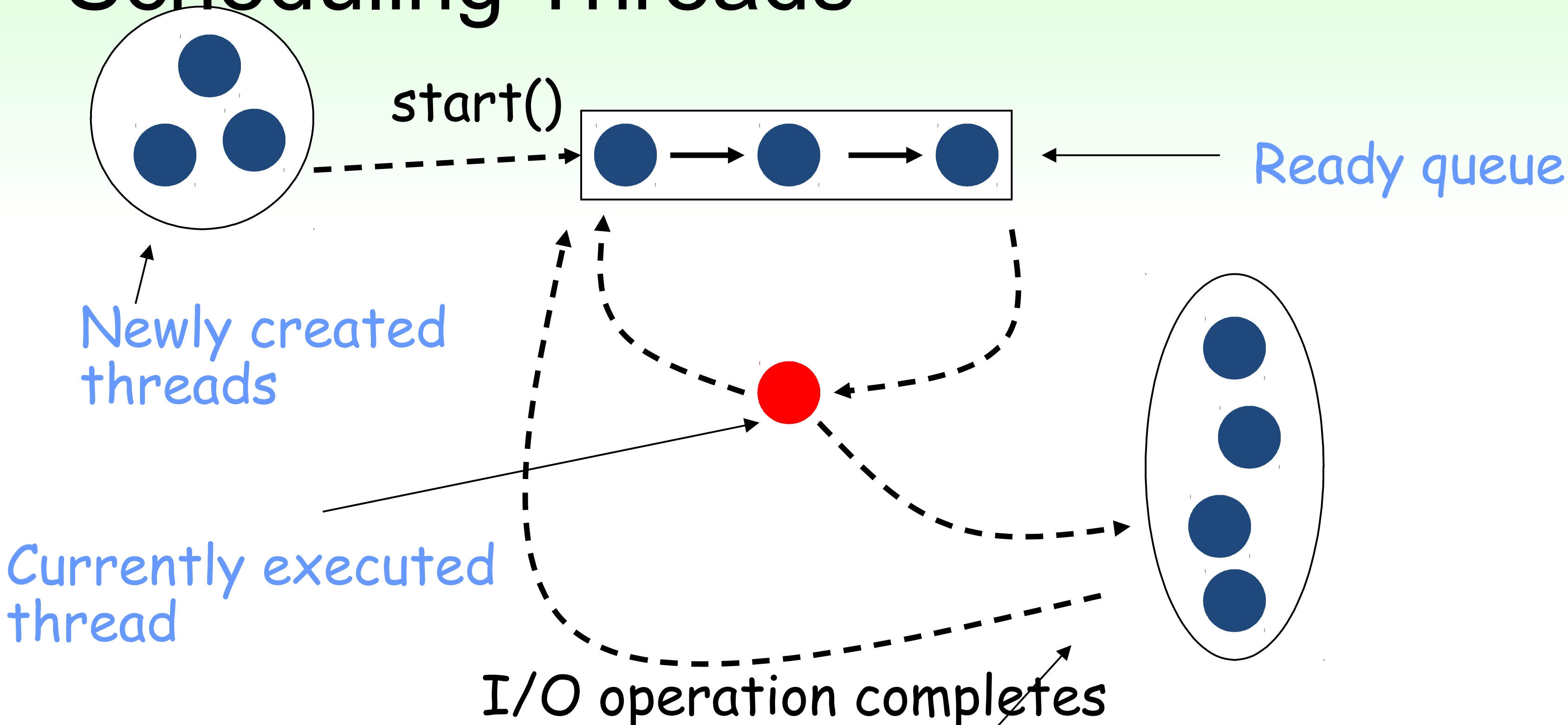
Example – a servlet that is also a thread

# Starting the Threads

```
public class ThreadsStartExample {
    public static void main (String argv[]) {
        new ThreadExample ().start ();
        new Thread(new RunnableExample ()).start ();
    }
}
```

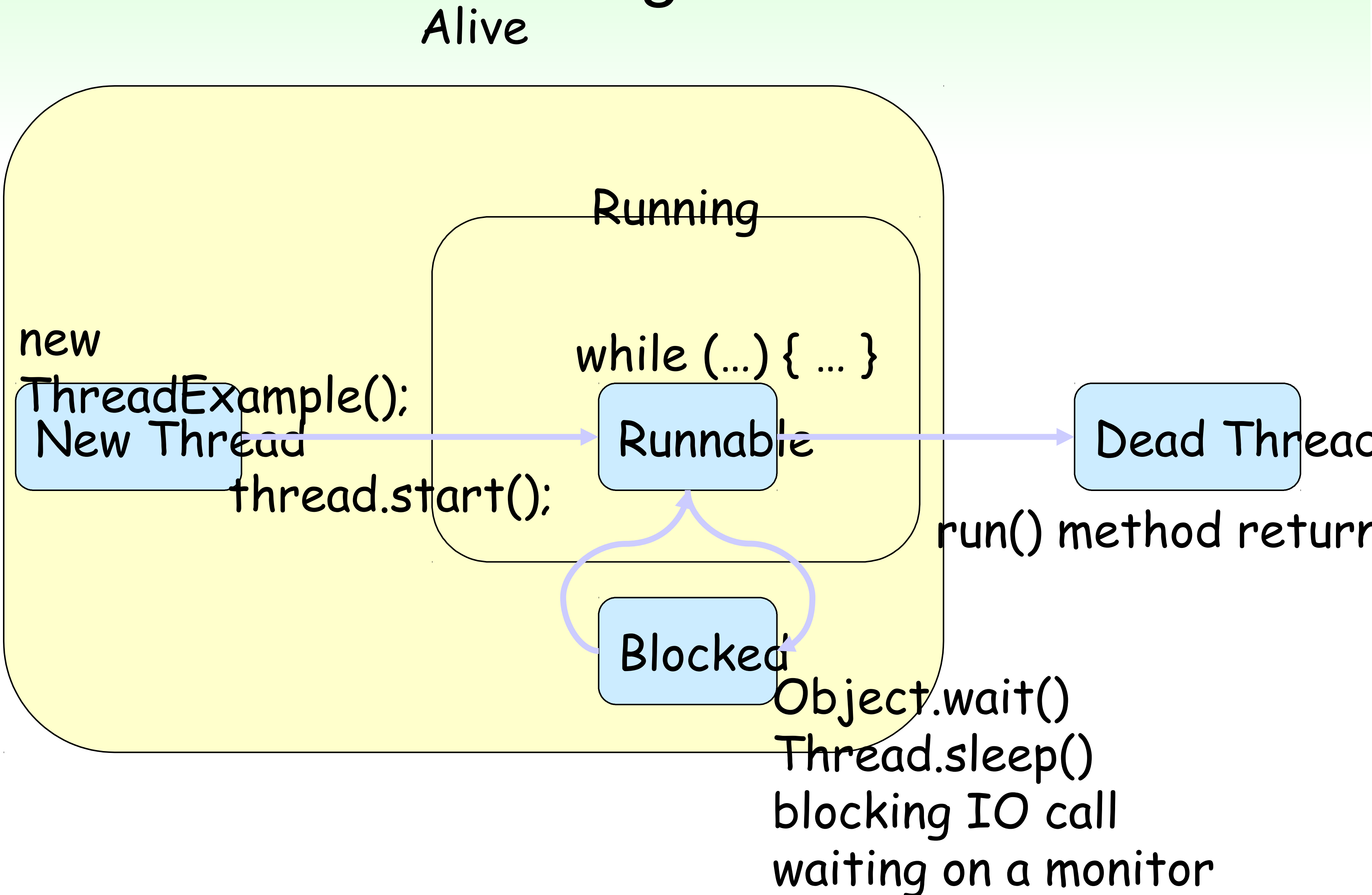
RESULT

# Scheduling Threads




- Waiting for I/O operation to be completed
- Waiting to be notified
- Sleeping
- Waiting to enter a synchronized section

# Thread State Diagram



# Example

```
public class PrintThread1 extends Thread {
    String name;
    public PrintThread1(String name) {
        this.name = name;
    }
    public void run() {
        for (int i=1; i<500 ; i++) {
            try {
                sleep((long)(Math.random() * 100));
            } catch (InterruptedException ie) { }
            System.out.print(name);
    } }
```

# Example (cont)

```
public static void main(String args[]) {
    PrintThread1 a = new PrintThread1("*");
    PrintThread1 b = new PrintThread1("-");
    PrintThread1 c = new PrintThread1("=");
    a.start();
    b.start();
    c.start();
  }
}
```

RESULT

# Scheduling

- Thread scheduling is the mechanism used to determine how runnable threads are allocated CPU time
- A thread-scheduling mechanism is either preemptive or nonpreemptive

# Preemptive Scheduling

- Preemptive scheduling – the thread scheduler preempts (pauses) a running thread to allow different threads to execute
- Nonpreemptive scheduling – the scheduler never interrupts a running thread
- The nonpreemptive scheduler relies on the running thread to yield control of the CPU so that other threads may execute

# Starvation

- A nonpreemptive scheduler may cause starvation (runnable threads, ready to be executed, wait to be executed in the CPU a lot of time, maybe even forever)
- Sometimes, starvation is also called a livelock

# Time-Sliced Scheduling

- Time-sliced scheduling – the scheduler allocates a period of time that each thread can use the CPU
  - when that amount of time has elapsed, the scheduler preempts the thread and switches to a different thread
- Nontime-sliced scheduler – the scheduler does not use elapsed time to determine when to preempt a thread
  - it uses other criteria such as priority or I/O status

# Java Scheduling

- Scheduler is preemptive and based on priority of threads
- Uses fixed-priority scheduling:
  - Threads are scheduled according to their priority w.r.t. other threads in the ready queue

# Java Scheduling

- The highest priority runnable thread is always selected for execution above lower priority threads
- When multiple threads have equally high priorities, only one of those threads is guaranteed to be executing
- Java threads are guaranteed to be preemptive-but not time sliced
- Q: Why can’t we guarantee time-sliced scheduling?

# Thread Priority

- Every thread has a priority
- When a thread is created, it inherits the priority of the thread that created it
- The priority values range from 1 to 10, in increasing priority

# Thread Priority (cont.)

- The priority can be adjusted subsequently using the **setPriority()** method
- The priority of a thread may be obtained using **getPriority()**
- Priority constants are defined:
  - MIN_PRIORITY=1
  - MAX_PRIORITY=10
  - NORM_PRIORITY=5

# Some Notes

- Thread implementation in Java is actually based on operating system support
- Some Windows operating systems support only 7 priority levels, so different levels in Java may actually be mapped to the same operating system level
- What should we do about this?

# Daemon Threads

- Daemon threads are “background” threads, that provide services to other threads, e.g., the garbage collection thread
- The Java VM will not exit if non-Daemon threads are executing
- The Java VM will exit if only Daemon threads are executing
- Daemon threads die when the Java VM exits

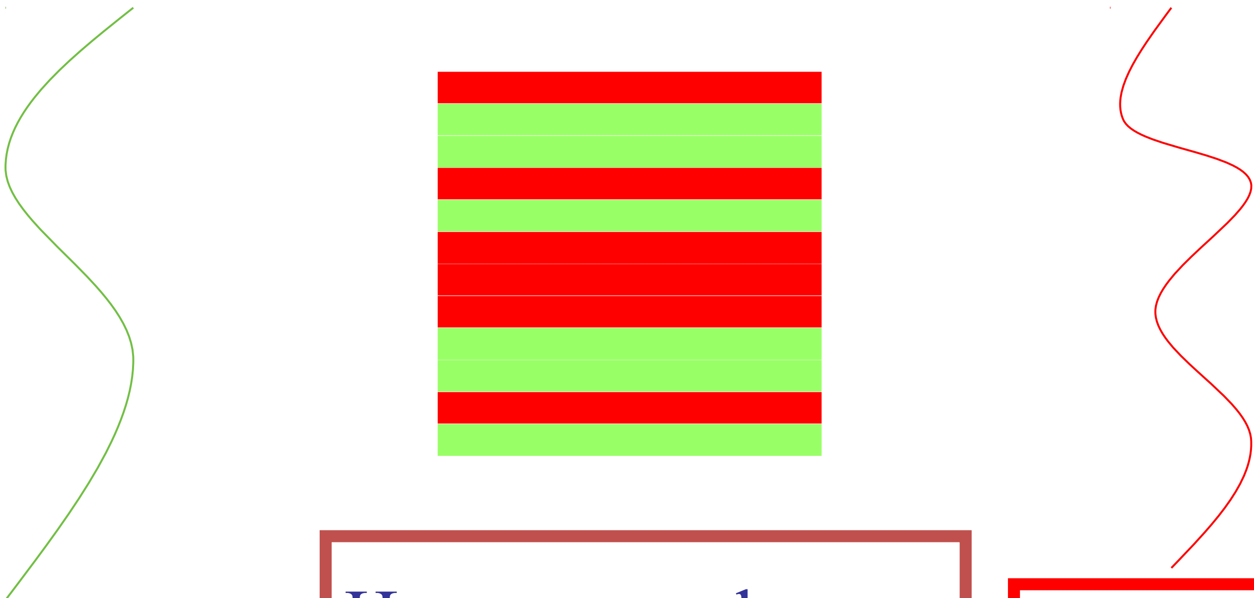
# Concurrency

- An object in a program can be changed by more than one thread
- Q: Is the order of changes that were preformed on the object important?

# Race Condition

- A race condition – the outcome of a program is affected by the order in which the program's threads are allocated CPU time
- Two threads are simultaneously modifying a single object
- Both threads “race” to store their value

# Race Condition Example



Put green pieces

How can we have alternating colors?

Put red pieces

# 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

# Monitor (cont.)

- 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

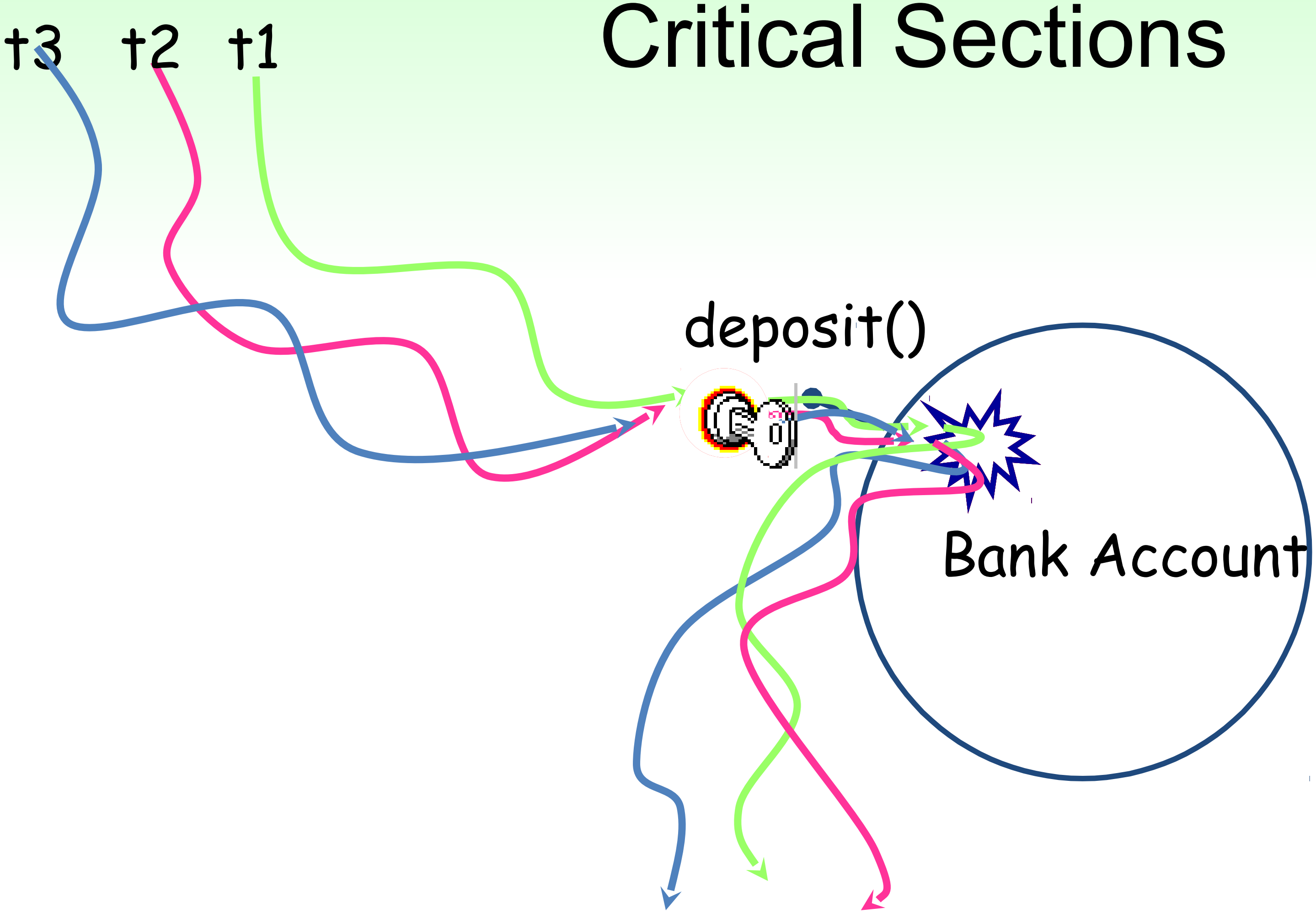
# Critical Section

- The synchronized methods define critical sections
- Execution of critical sections is mutually exclusive. Why?

# Example

```
public class BankAccount {

    private float balance;

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

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

# Critical Sections



# Static Synchronized Methods

- 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. Why?

# Example

```
public class PrintThread2 extends Thread {
    String name;
    public PrintThread2(String name) {
        this.name = name;
    }
    public static synchronized void print(String name) {
        for (int i=1; i<500 ; i++) {
            try {
                Thread.sleep((long)(Math.random() * 100));
            } catch (InterruptedException ie) { }
            System.out.print(str);
        }
    }
```

# Example (cont)

```
public void run() {
    print(name);
 }

public static void main(String args[]) {
    PrintThread2 a = new PrintThread2("*“);
    PrintThread2 b = new PrintThread2("-“);
    PrintThread2 c = new PrintThread2("=“);
    a.start();
    b.start();
    c.start();
  }
}
```

RESULT

```
public class BankAccount {

    private 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);
    }
}
```

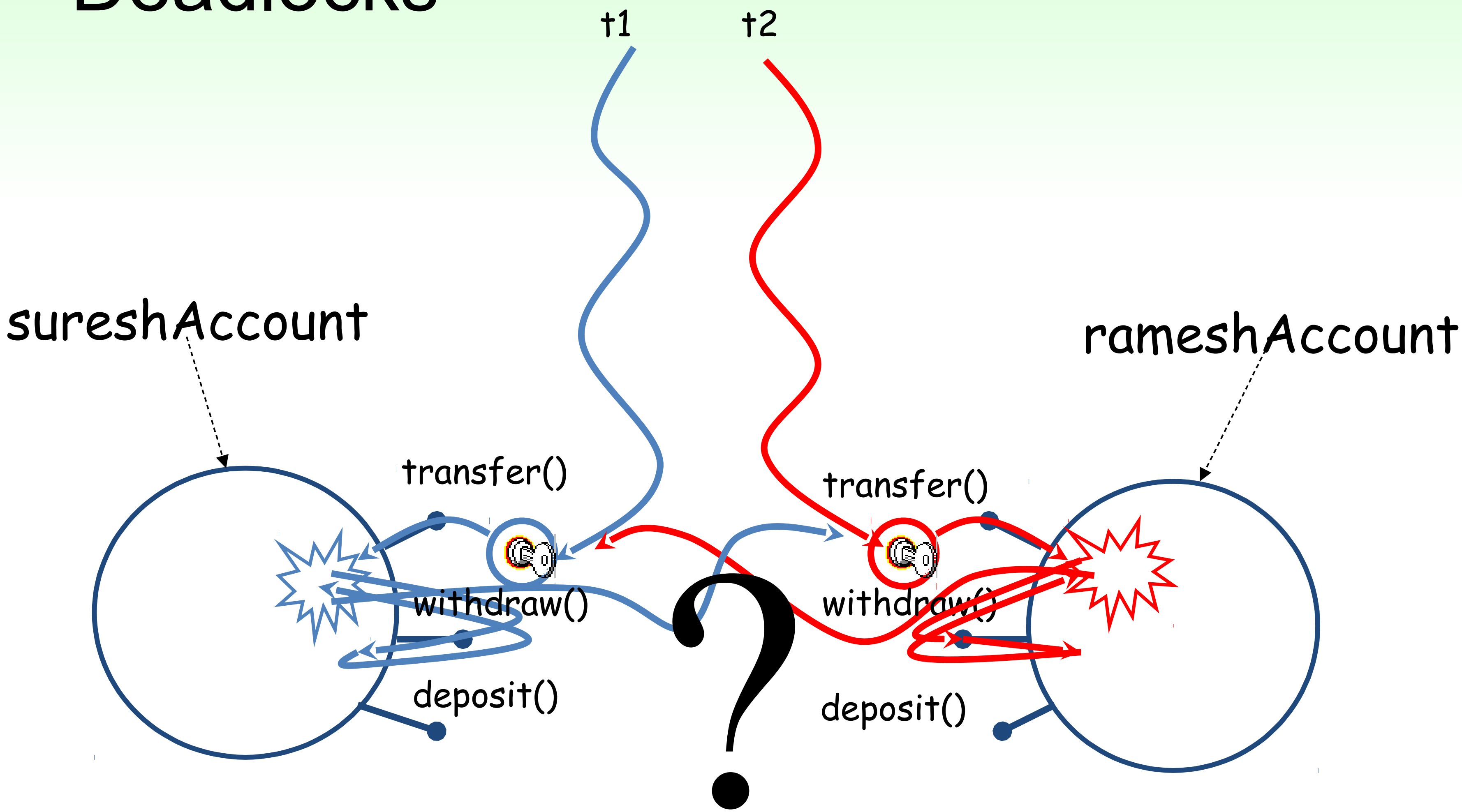
```
public class MoneyTransfer implements Runnable {

  private BankAccount from, to;
  private float amount;

  public MoneyTransfer(
     BankAccount from, BankAccount to, float amount)  {
     this.from = from;
     this.to = to;
     this.amount = amount;
  }

  public void run() {
     from.transfer(amount, target);
  }
}
```

```
BankAccount sureshAccount = new BankAccount();
BankAccount rameshAccount = new BankAccount();
...
```

```
// At one place
Runnable transaction1 =
   new MoneyTransfer(sureshAccount, rameshAccount, 1200);
Thread t1 = new Thread(transaction1);
t1.start();
```

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

# Deadlocks



t1
t2
sureshAccount
rameshAccount
transfer()
withdraw()
deposit()
transfer()
withdraw()
deposit()
?

# Java Locks are Reentrant

- Is there a problem with the following code?

```
public class Test {
    public synchronized void a() {
        b();
        System.out.println("I am at a");
    }
    public synchronized void b() {
        System.out.println("I am at b");
    }
}
```

# Synchronized Statements

- A monitor can be assigned to a block
- It can be used to monitor access to a data element that is not an object, e.g., array
- Example:

```
void arrayShift(byte[] array, int count) {
        synchronized(array) {
            System.arraycopy (array, count,array,                0,
    array.size - count);
          }
}
```

# Thread Synchronization

- 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
  - Ramesh put a cookie wait and notify Suresh
  - Suresh eat a cookie wait and notify Ramesh
    - Ramesh put a cookie wait and notify Suresh
    - Suresh eat a cookie wait and notify Ramesh

# The wait() Method

- The **wait()** method is part of the **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. Why?

# The wait() Method

- wait() causes the current thread to 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

# The wait() Method

- **wait()** is also similar to **yield()**
  - Both take the current thread off the execution stack and force it to be rescheduled
- However, **wait()** is not automatically put back into the scheduler queue
  - **notify()** must be called in order to get a thread back into the scheduler's queue

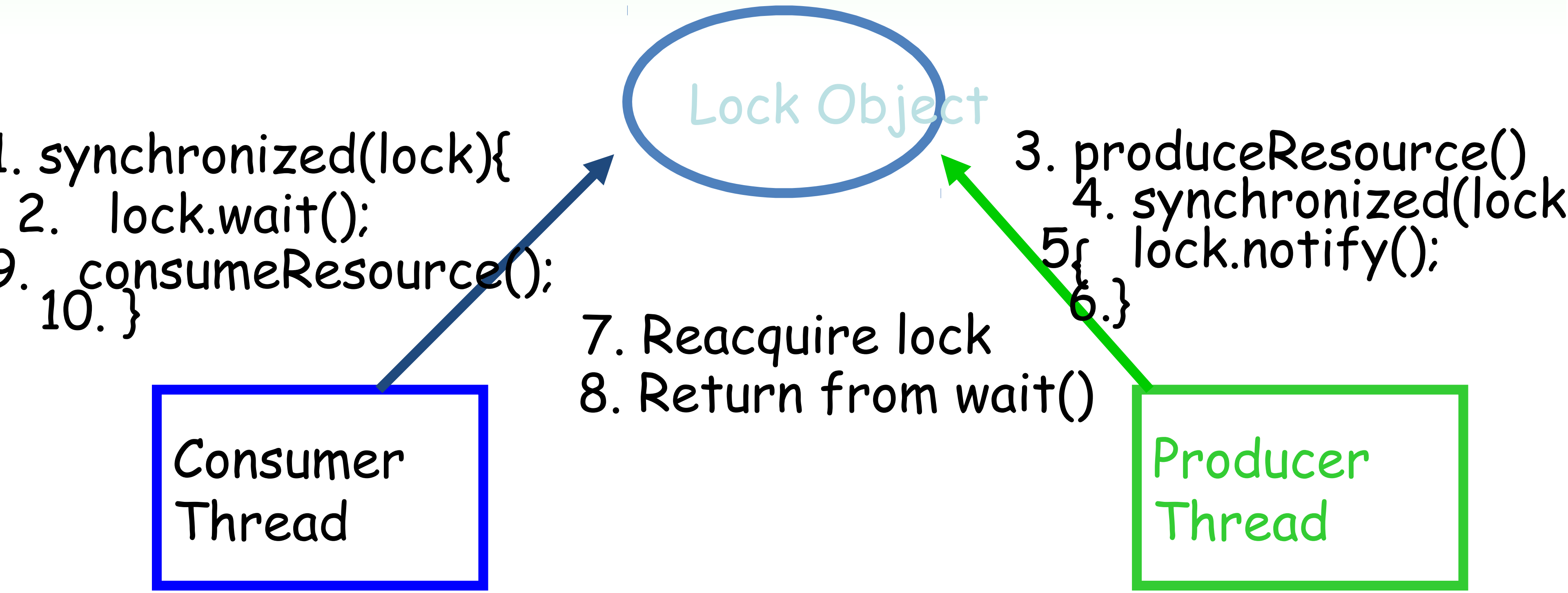
# Consumer

```
synchronized (lock) {
    while (!resourceAvailable()) {
        lock.wait();
    }
    consumeResource();
}
```

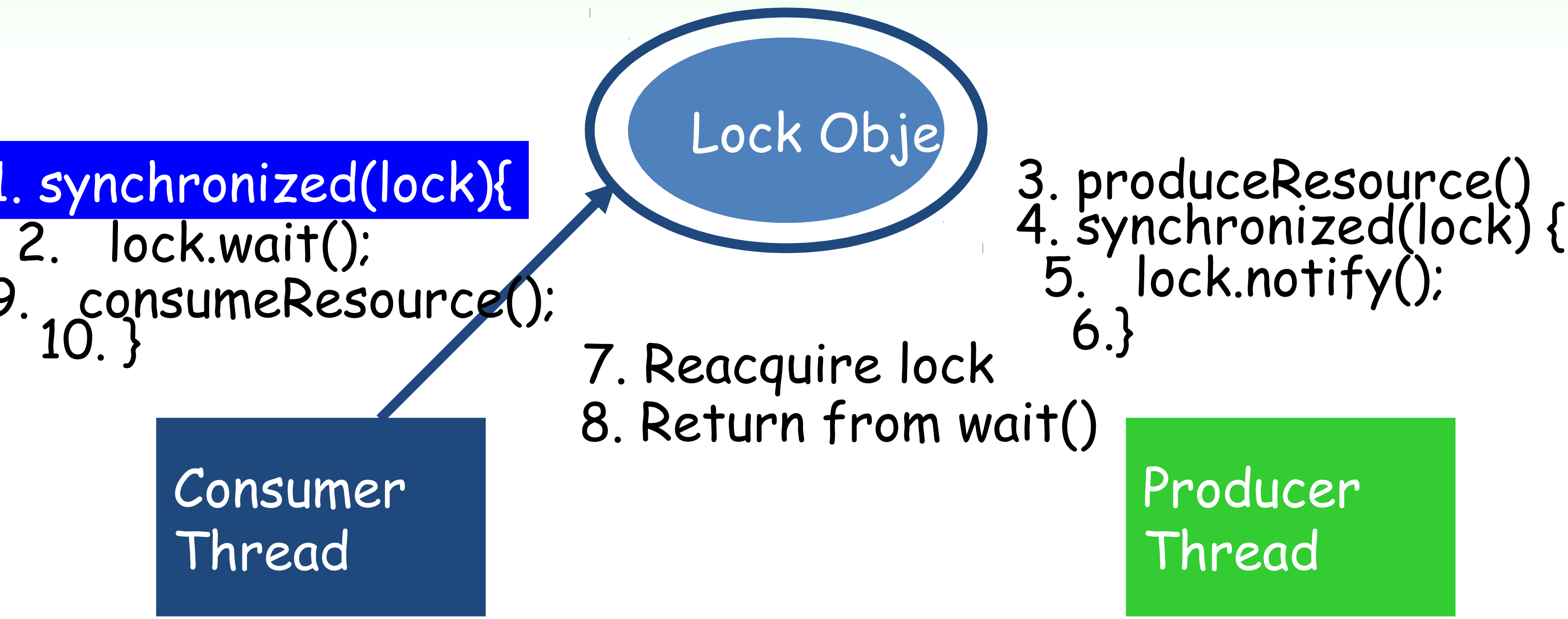
# Producer

```
produceResource();
synchronized (lock) {
    lock.notifyAll();
}
```

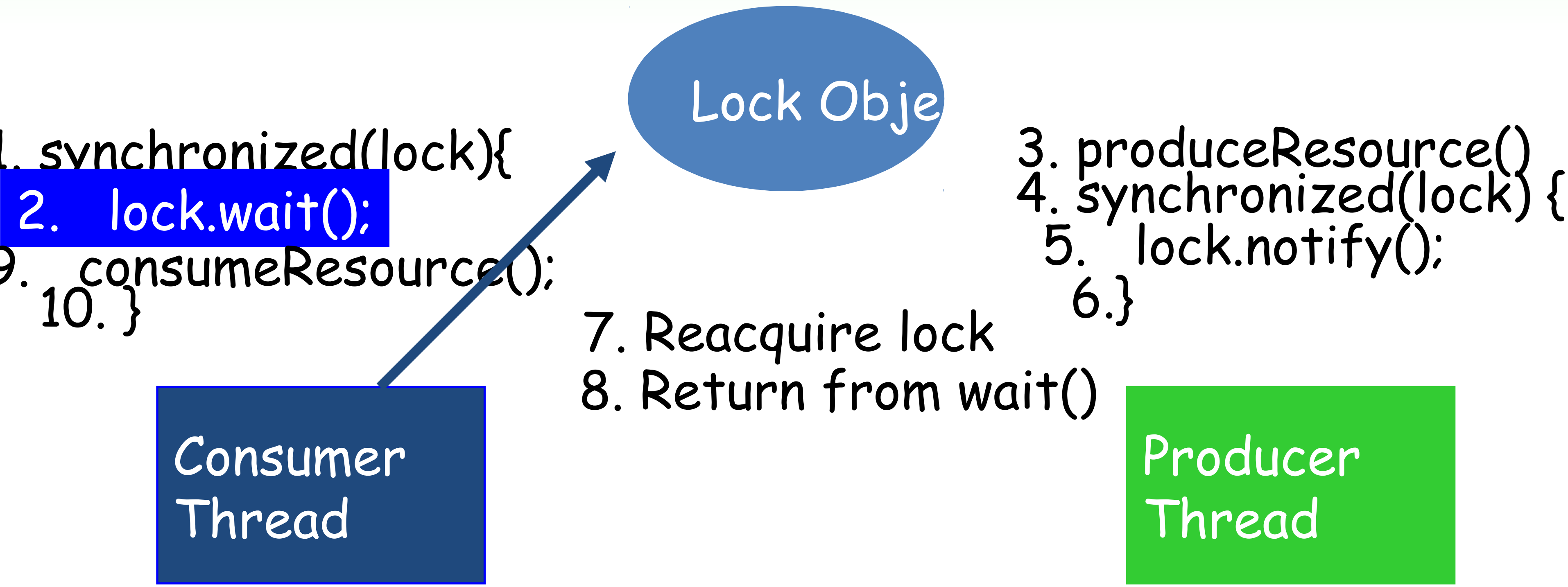
# Wait/Notify Sequence



# Wait/Notify Sequence



# Wait/Notify Sequence



# Wait/Notify Sequence

Lock Obje

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

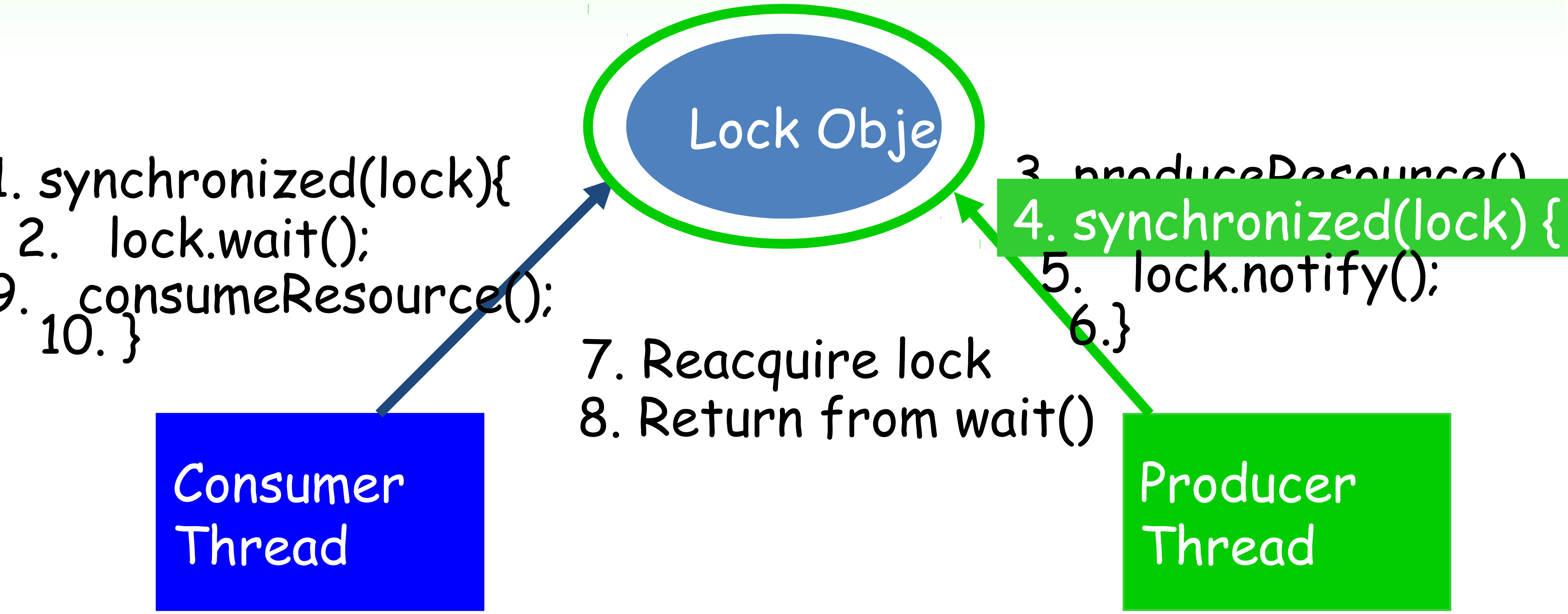
Consumer Thread

7. Reacquire lock
8. Return from wait()

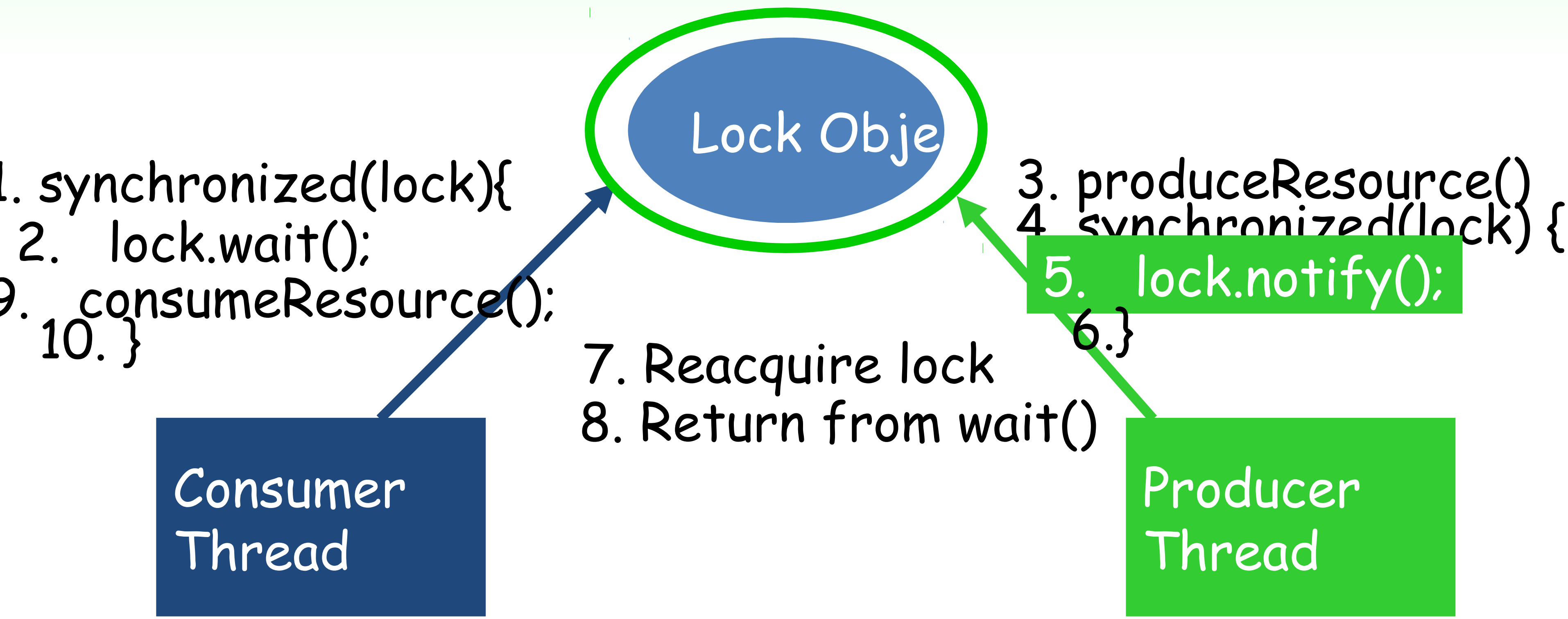
3. produceResource()
4. synchronized(lock) {
5. lock.notify();
6.}

Producer Thread

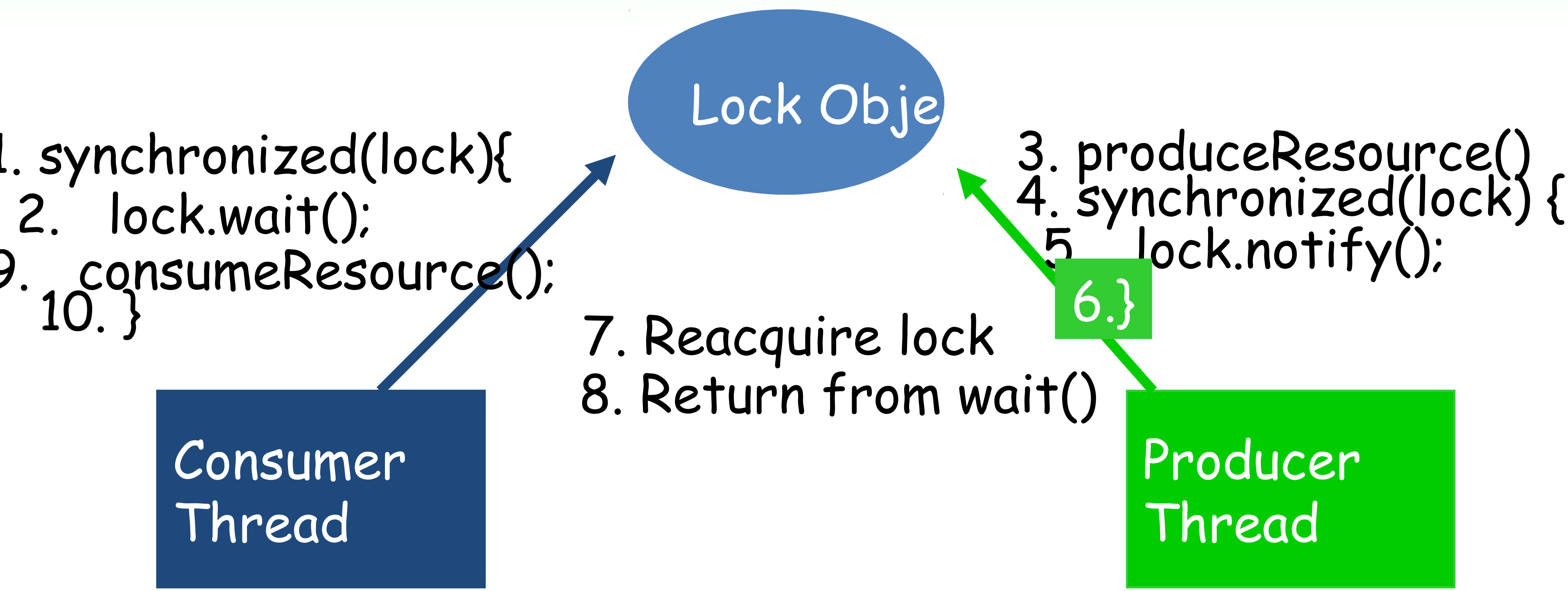
# Wait/Notify Sequence



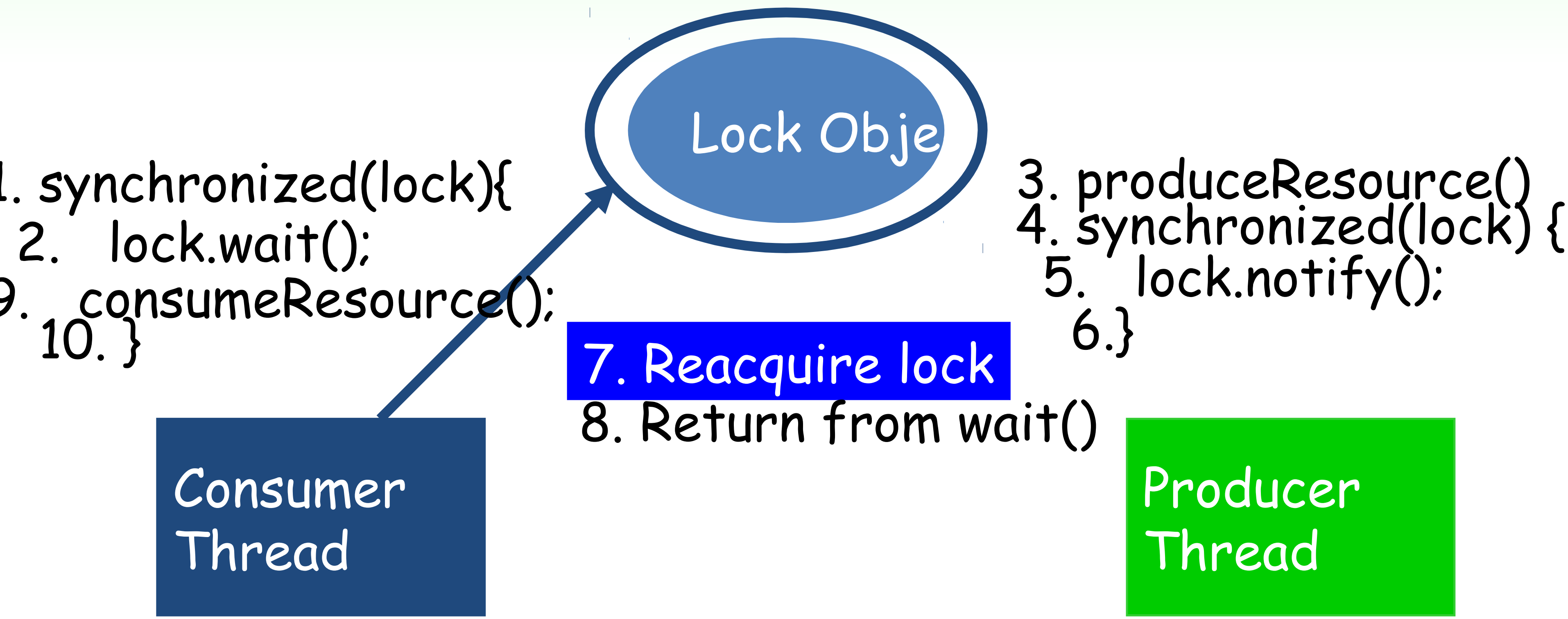
# Wait/Notify Sequence



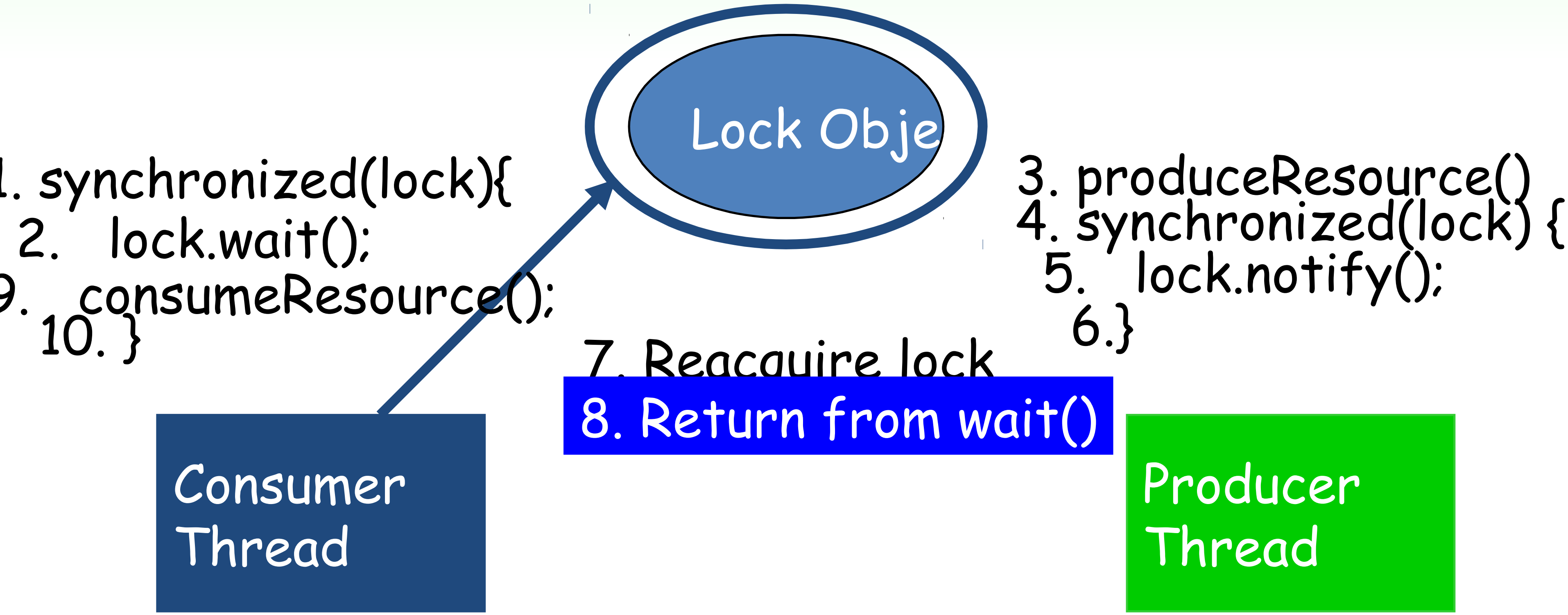
# Wait/Notify Sequence



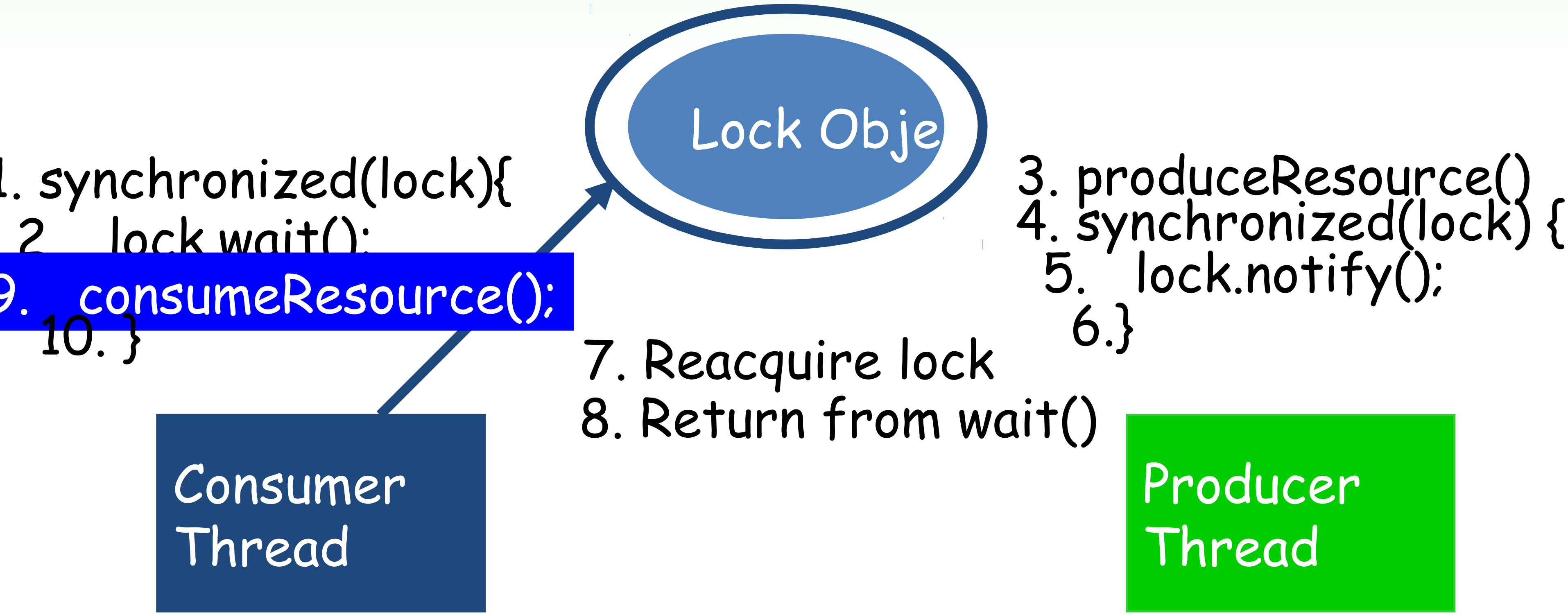
# Wait/Notify Sequence



# Wait/Notify Sequence



# Wait/Notify Sequence



# Wait/Notify Sequence

Lock Obje

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

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

7. Reacquire lock
8. Return from wait()

Consumer Thread

Producer Thread

# The Producer-Consumer Scenario:Cookies

```
public class Cookies {

  public static void main(String[] args) {

    CookyJar jar = new CookyJar();

    Suresh suresh = new Suresh(jar);
    Ramesh ramesh = new Ramesh(jar);

    new Thread(Suresh).start();
    new Thread(Ramesh).start();
  }
}
```

# The Producer-Consumer Scenario: Suresh

```
public class Suresh implements Runnable {
    CookyJar jar;

    public Suresh(CookyJar jar) {
        this.jar = jar;
    }

    public void eat() {
        jar.getCooky("Suresh");
        try {
            Thread.sleep((int)Math.random() * 1000);
        } catch (InterruptedException ie) {}
    }

    public void run() {
        for (int i = 1 ; i <= 10 ; i++) eat();
    }
}
```

# The Producer-Consumer Scenario: Ramesh

```
public class Ramesh implements Runnable {
    CookyJar jar;

    public Ramesh(CookyJar jar) {
        this.jar = jar;
    }

    public void bake(int cookyNumber) {
        jar.putCooky("Ramesh", cookyNumber);
        try {
            Thread.sleep((int)Math.random() * 500);
        } catch (InterruptedException ie) {}
    }

    public void run() {
        for (int i = 0 ; i < 10 ; i++) bake(i);
    }
}
```

# The Producer-Consumer Scenario: CookieJar

```
public class CookyJar {
    private int contents;
    private boolean available = false;

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

# The Producer-Consumer Scenario: CookieJar

```
public synchronized void putCooky(String who, int value) {
    while (available) {
        try {
            wait();
        } catch (InterruptedException e) { }
    }
    contents = value;
    available = true;
    System.out.println(who + " put cooky " + contents +
                                   " in the jar");
    notifyAll();
}}
```