Sesh Venugopal

CS 213 Spring 2023

Apr 17

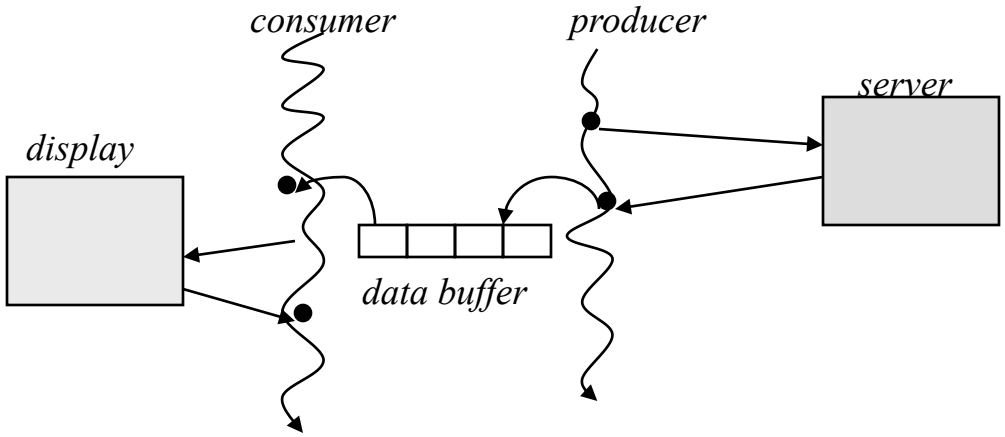
Multithreaded Programming II

Sesh Venugopal

Resource Sharing Between Threads

- A common design situation in multi-threaded programs is in the sharing of some data resource between two or more threads

- Typically, a buffer is shared between a producer thread which puts data in the buffer, and a consumer thread that takes data out of the buffer

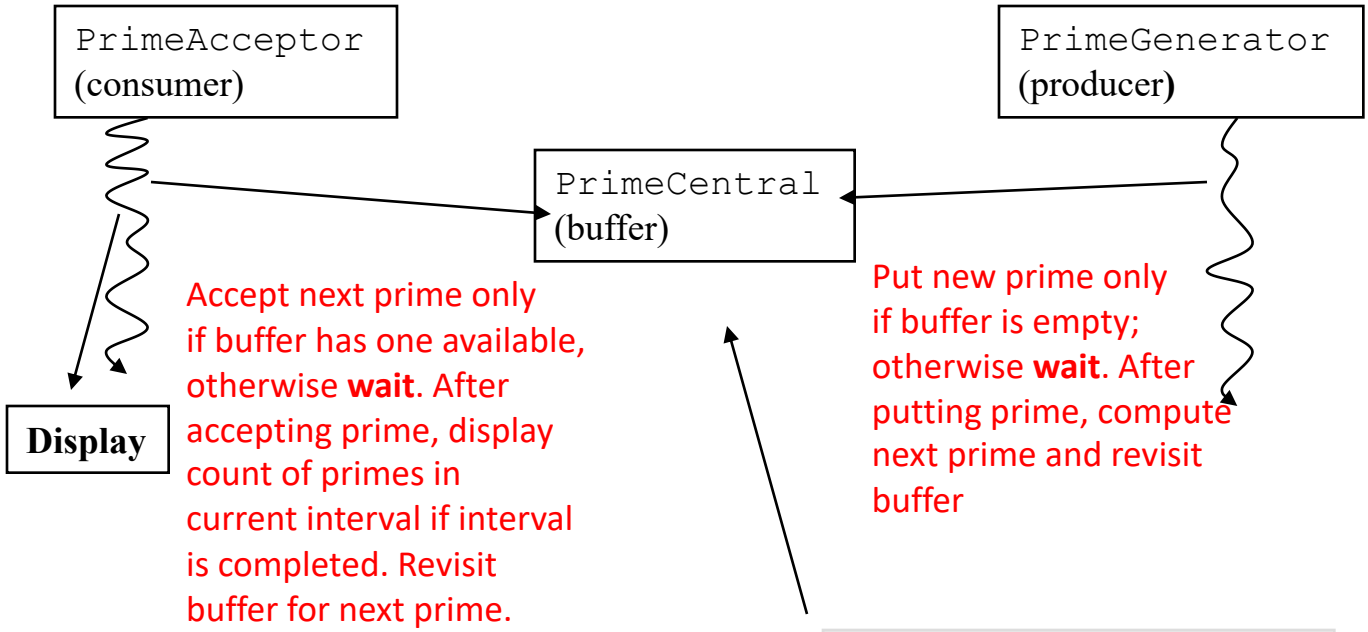


- While this is an excellent opportunity for asynchronous data filling and retrieval, care has to be taken to ensure that the producer and consumer "talk" to each other so they don't work in ways that result in inconsistent or incorrect results

- Example: A shared array into which a producer puts data one element at a time, first to last, and from which a consumer takes data out, one element at a time, first to last. The consumer must ensure there is at least one unretrieved element available before it can take anything out

Sesh Venugopal

Prime Counter: Producer/Consumer



The shared object, which is a `PrimeCentral` instance, is responsible for ensuring that `PrimeGenerator` and `PrimeAcceptor` coordinate their accesses to it.

```
> java PrimeDriver
Enter integer bound => 100000
Enter number of intervals => 10
#Primes in range [90001,100000] :879
#Primes in range [80001,90000] :876
#Primes in range [70001,80000] :902
#Primes in range [60001,70000] :878
#Primes in range [50001,60000] :924
#Primes in range [40001,50000] :930
#Primes in range [30001,40000] :958
#Primes in range [20001,30000] :983
#Primes in range [10001,20000] :1033
#Primes in range [2,10000] :1229
Total number of primes <= 100000 : 9592
```

Sesh Venugopal

Prime Counter: Producer

```
package primesync;

public class PrimeGenerator implements Runnable {

    private PrimeCentral primeCentral;
    private int bound;

    public PrimeGenerator(PrimeCentral primeCentral,
                          int bound) {
        this.primeCentral = primeCentral;
        this.bound = bound;
        new Thread(this).start();
    }

    public void run() {
        int n=bound;
        while (n > 1) {
            int d;
            for (d=2; d <= n/2; d++) {
              if ((n % d) == 0) {
                 break;
              }
            }
            if (d > n/2) {
              primeCentral.put(n);
            }
            n--;
        }
    }
}
```

data buffer that is shared by producer and consumer

This thread generates prime numbers between 2 and bound, for a given bound. Every time a prime number is identified, it is written into a buffer called primeCentral.

Sesh Venugopal

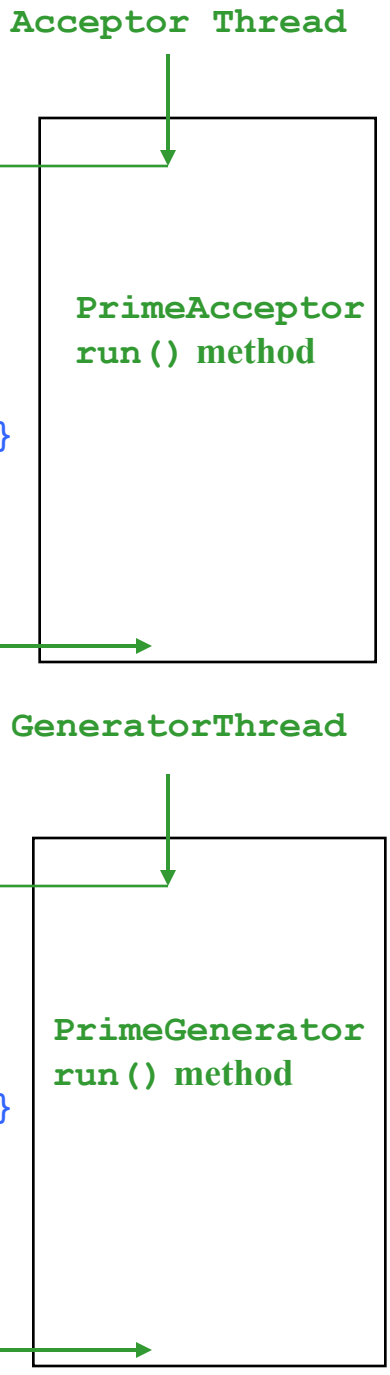
Prime Counter: Consumer

```
package primesync;

public class PrimeAcceptor implements Runnable {
   private PrimeCentral primeCentral;
   private int bound, numIntervals;

   public PrimeAcceptor(PrimeCentral primeCentral,
                        int bound, int numIntervals) {
      this.primeCentral = primeCentral;
      this.numIntervals = numIntervals; this.bound = bound;
      new Thread(this).start();
   }

   public void run() {
      int range = bound/numIntervals;
      int lo = bound-(bound%numIntervals+range-1);
      int rangeCount=0, totalCount=0, hi=bound;
      while (true) {
         int prime = primeCentral.get();
         if (prime == 2) {
             rangecount++; break;
         }
         if (prime < lo) {
             System.out.println("#Primes in range [" + lo +
                           "," + hi + "] :\t" + rangeCount);
             totalCount += rangeCount;
             rangeCount=0; hi=lo-1; lo=hi-range+1;
         }
         rangeCount++;
      }
      System.out.println("#Primes in range [2," + hi +
                         "] :\t" + rangeCount);
      totalCount += rangeCount;
      System.out.println("Total number of primes <= " +
                          bound + " : " + totalCount);
   }
}
```

Sesh Venugopal

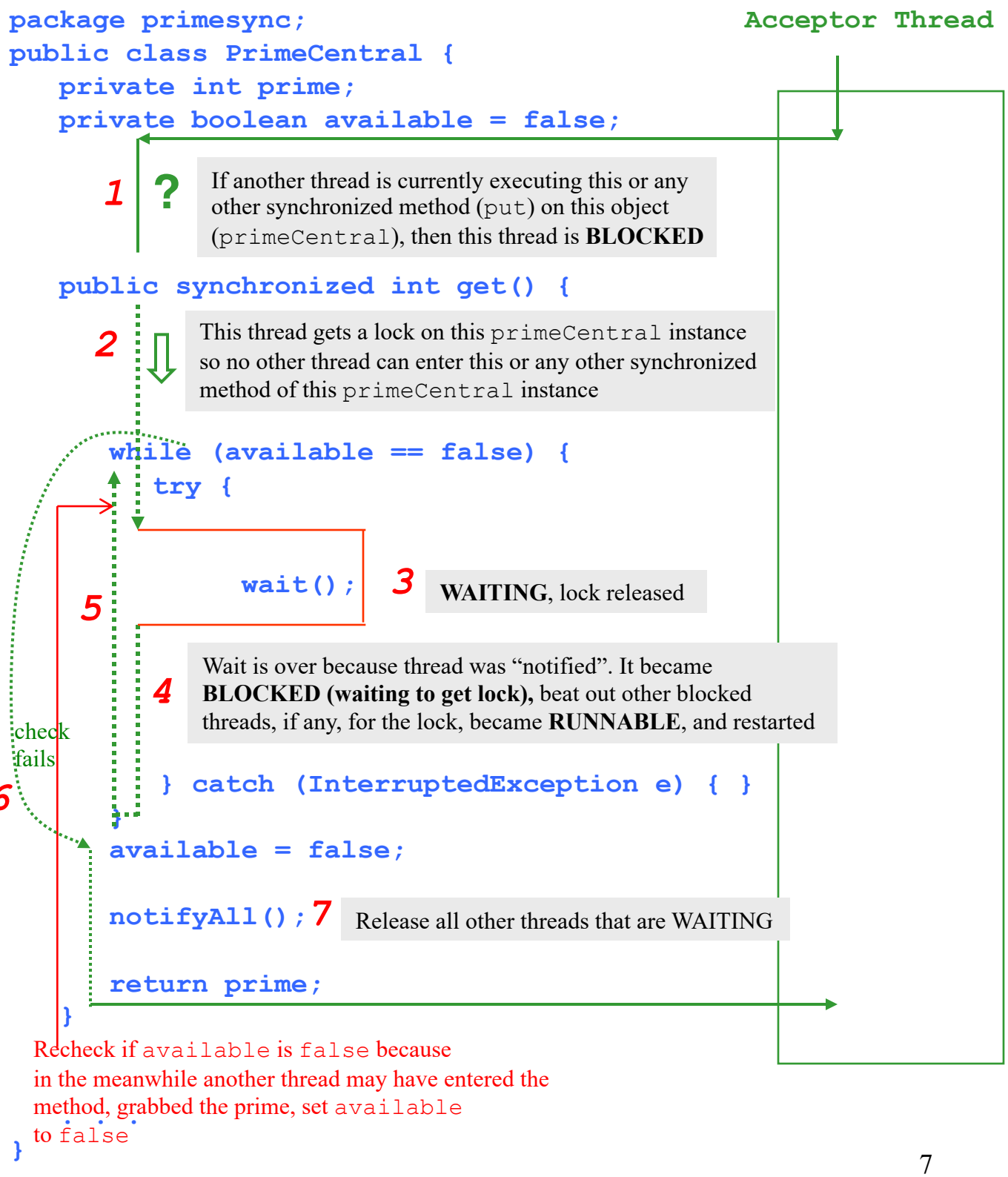
Prime Counter: Shared Resource

```
package primesync;

public class PrimeCentral {
   private int prime;
   private boolean available = false;

   public synchronized int get() {
      while (available == false) {
         try {
            wait();
         } catch (InterruptedException e) { }
      }
      available = false;
      notifyAll();
      return prime;
   }

   public synchronized void put(int prime) {
      while (available == true) {
         try {
            wait();
         } catch (InterruptedException e) { }
      }
      this.prime = prime;
      available = true;
      notifyAll();
   }
}
```



Sesh Venugopal

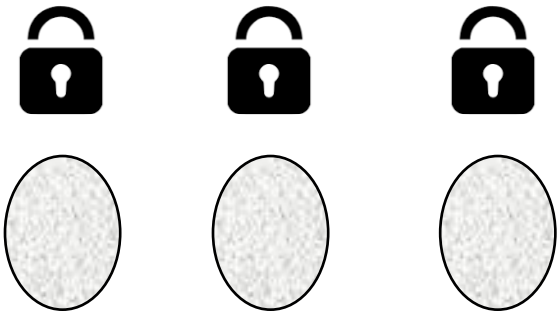
Prime Counter: Shared Resource



Thread Synchronization

- A **synchronized** method implements mutual exclusion: i.e. only one thread can execute the method at any time. A synchronized method is said to implement a *critical section*, i.e. code that can be executed by several concurrent threads on the same object.

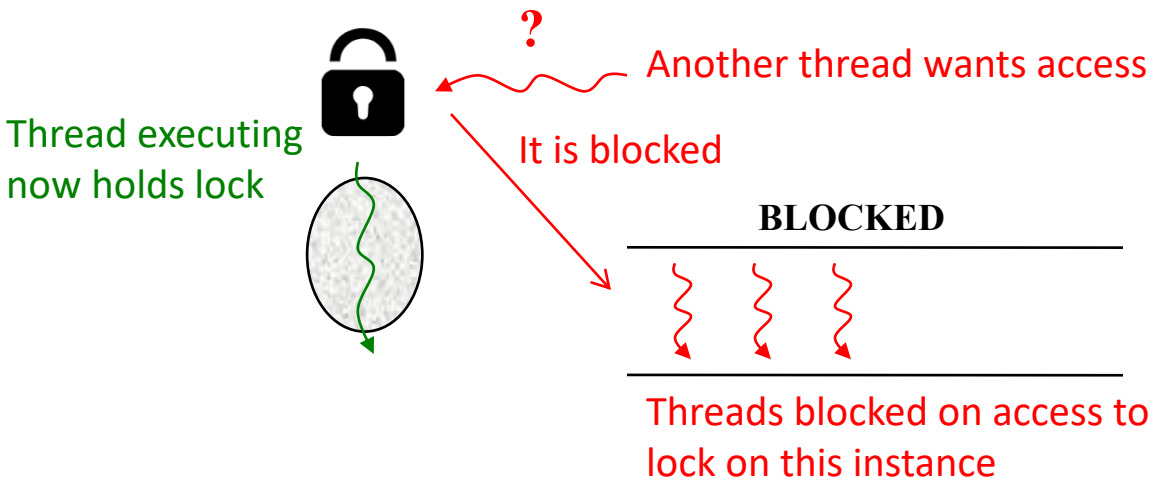
- To enter a **synchronized** method, a thread must acquire a **lock** on the object on which this method is invoked



EVERY INSTANCE of a class with a synchronized method has its own personal lock

Thread Synchronization

- If a lock on the object is already held by another thread (because it is executing this, or some other synchronized method on the same object), the *requesting* thread will be **blocked**

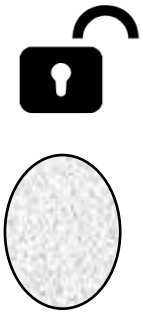


Sesh Venugopal

Thread Synchronization

- A blocked thread *may* be unblocked when the thread that is currently *holding* the object lock finishes executing the synchronized method/block (the blocked thread will have to compete with other blocked threads to reacquire the lock)

Lock released



ANY one of these threads will be unblocked – NOT FIFO

Thread Synchronization

- Two or more threads may concurrently execute different methods *on the same object* as long as at most one of these methods is synchronized. (For instance, one thread may be executing a synchronized method while another may be executing a non-synchronized method at the same time.)

Waiting and Notification

- `wait()` is an `Object` class method: only a thread that is holding a lock on an object may issue a `wait()`

- A thread that issues a `wait()` relinquishes its lock on the object and is taken out of contention for execution (not *runnable*)

- A *waiting* thread may be released from its wait when the thread that is currently *holding* the object lock calls the `notify` or `notifyAll` method on that object

- If several threads are *waiting* on a lock for the same object:
  - A `notify` call by the *holding* thread will release one of the (arbitrarily chosen) waiting threads, which will then become **blocked** (in contention with other **blocked** threads to acquire lock)
  - A `notifyAll` call by the holding thread will release all the waiting threads, which will become **blocked**, in contention with other **blocked** threads for acquiring a lock on the object

Sesh Venugopal

Thread Life Cycle

- Invoking the **start** method on a thread puts it in the *runnable* state
- A thread being in the runnable state only means that it will *share* cpu time with other runnable threads – a runnable thread is not actually *running* when another runnable thread is executing on the cpu
- A thread terminates (safely) by completing its run method, and goes into the *terminated* state

Sesh Venugopal

Thread Life Cycle

A thread becomes *not runnable* when one of the following occurs:

1. It is in a blocked state because another thread is holding a lock on an object (target) on which this thread is trying to acquire a lock

2. It is in timed_waiting state, because the sleep method is invoked on it. In this case, the thread becomes runnable once the sleep time has elapsed (Note: sleep is independent of synchronization. If the sleep is invoked within a method that is synchronized, the lock is *not given up* by the sleeping thread.)

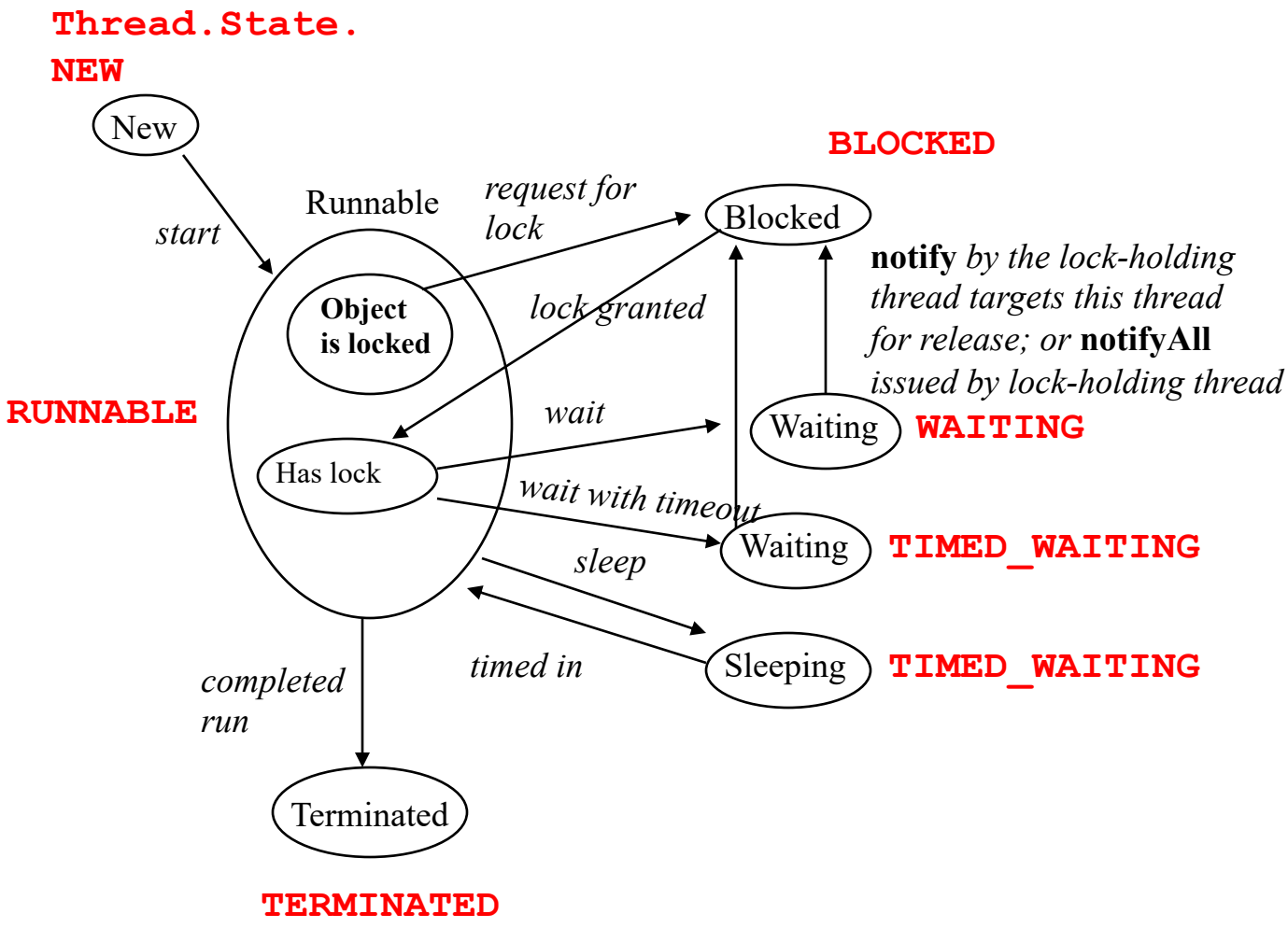
3. It is in the timed_waiting state because the wait method was invoked on it with timeout.

4. It is in the waiting state, because it invoked the wait method on the target object

5. It is blocking on I/O

Sesh Venugopal

Thread Life Cycle



Sesh Venugopal

Statement/Block level Synchronization

• Sometimes it is necessary to synchronize a single statement or block of statements instead of an entire method

• Example: A linked list that has been implemented without consideration of multiple threads using it

• If an application uses instances of this linked list, and finds a need to run multiple threads through them, it needs to enforce *thread safety* in its code if the linked list code comes in a library that is not modifiable. (What could go wrong if the application did not do anything different – how could multiple threads lead to incorrect data in the list?)

Sesh Venugopal

Statement/Block level Synchronization

- Here's a possible way to enforce thread safety:

```
List ll = new LinkedList();
SomeRunnable r = new SomeRunnable(ll);
SomeThread t1 = new Thread(r).start();
SomeThread t2 = new Thread(r).start();
```

Set up threads that would do stuff on a linked list instance

```
public void run() {
    ...
    synchronized(ll) {
       ll.addToFront(5);
    }
    ...
}
```

In the run method of SomeRunnable, sychronize on the linked list object in order to execute the add front statement

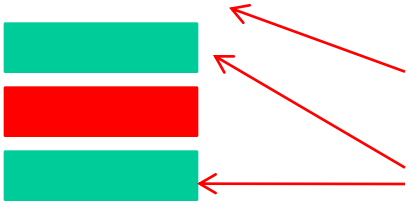
- An add is done to the front of the linked list only after acquiring a lock on the linked list instance so that the two threads don't get into a conflict while trying to add at the same time

Sesh Venugopal

Statement/Block level Synchronization

• Another example: suppose a method in an object contains a block of code that needs to run with mutual exclusion (one thread at a time), but there is a significant amount of other code that CAN be run by multiple threads at the same time

```
public synchronized void m() {
```



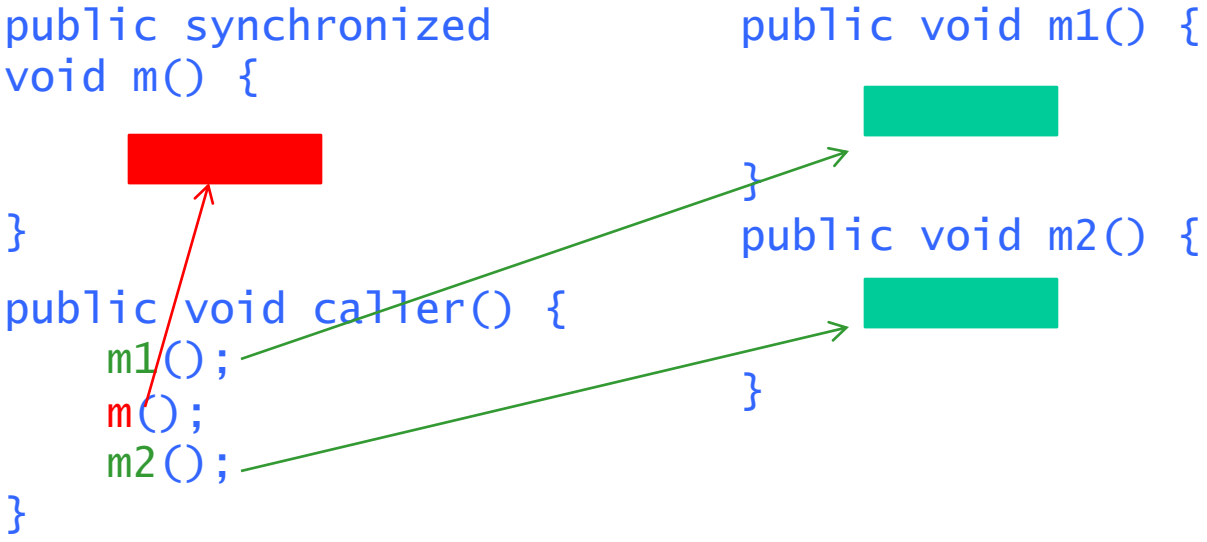
Synchonizing entire method forces execution to be unnecessarily sequentialized over the non-critical parts

```
}
```

Sesh Venugopal

Statement/Block level Synchronization

- One way to solve the problem is to separate out the non-critical blocks into their own non-synchronized methods:



- Another way is to do this: