Konzepte objektorientierter Programmierung
– Lecture 8 –

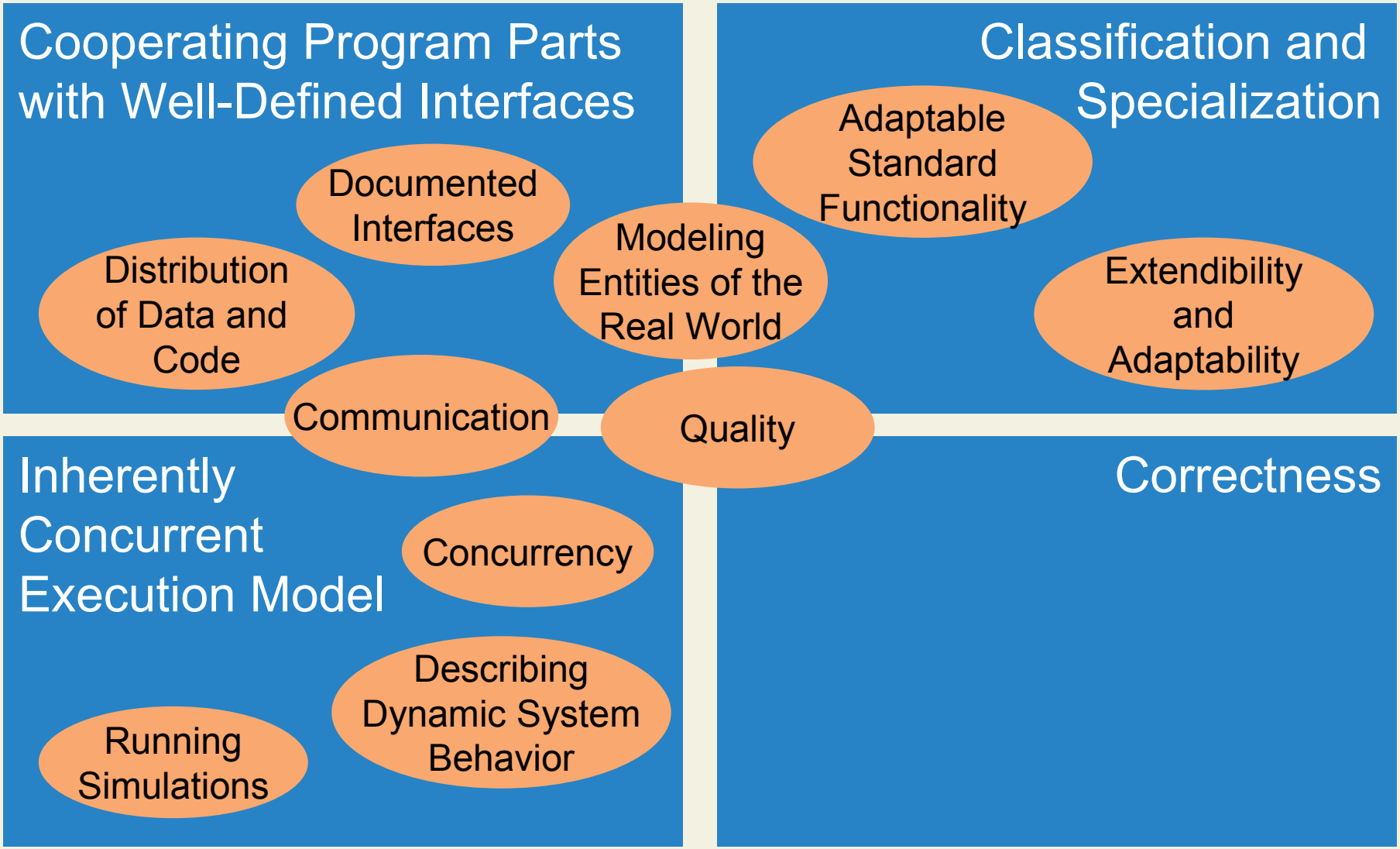
Prof. Dr. Peter Müller

Software Component Technology

Wintersemester 03/04

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Meeting the Requirements



Meeting the Requirements

Cooperating Program Parts with Well-Defined Interfaces

→ Objects (data + code)
→ Interfaces
→ Encapsulation

Classification and Specialization

→ Classification, subtyping
→ Polymorphism
→ Substitution principle

Inherently Concurrent Execution Model

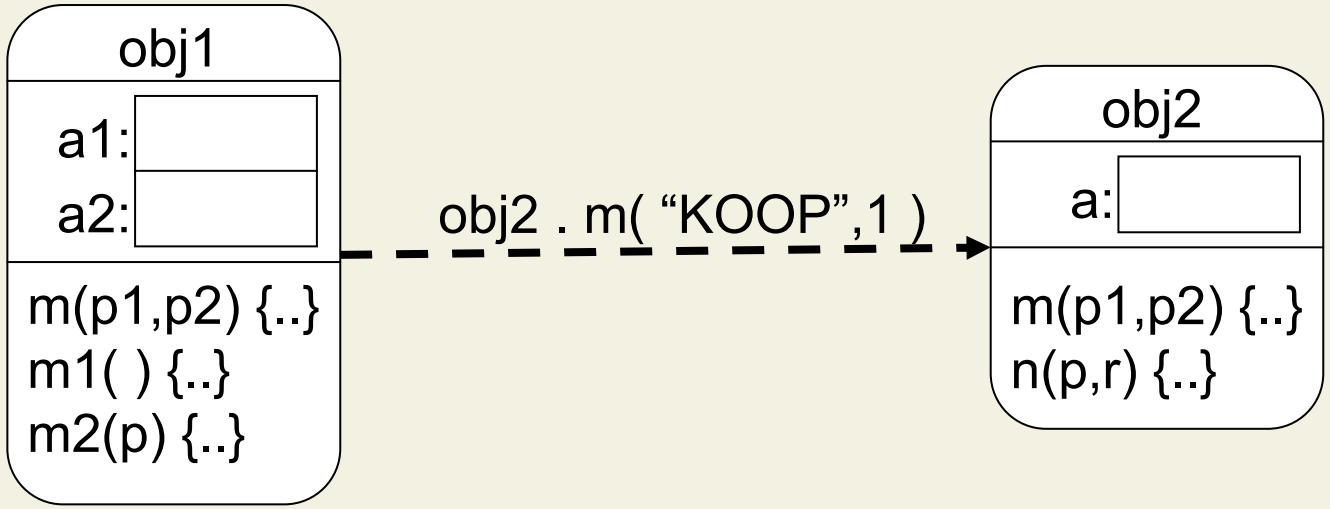
→ Active objects
→ Message passing

Correctness

→ Interfaces
→ Encapsulation
→ Simple, powerful concepts

The Object Model

- A software system is a set of cooperating objects
- Objects have state and processing ability
- Objects exchange messages



Agenda for Today

8. Concurrency

- 8.1 Threads
- 8.2 Synchronization
- 8.3 Active Objects

Objectives

- Object-oriented concurrency model
- Synchronization of objects and object structures

8. Concurrency

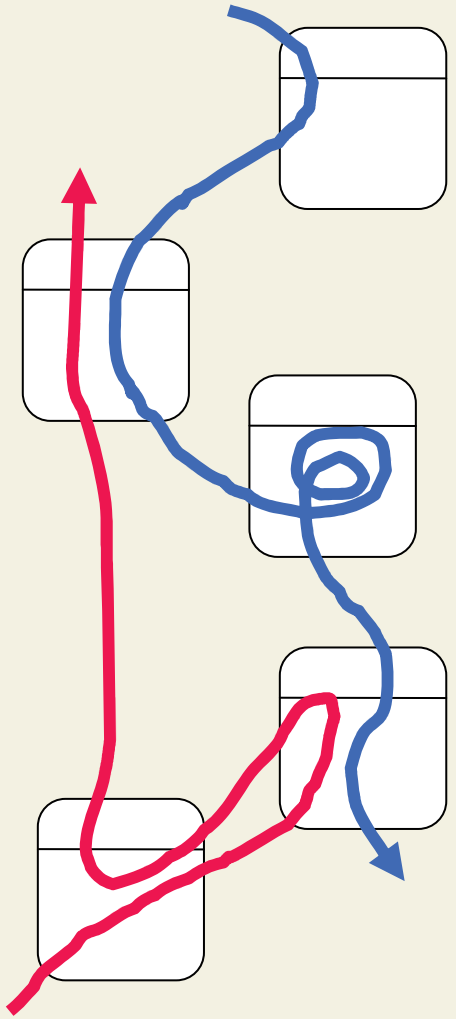
8.1 Threads

8.2 Synchronization

8.3 Active Objects

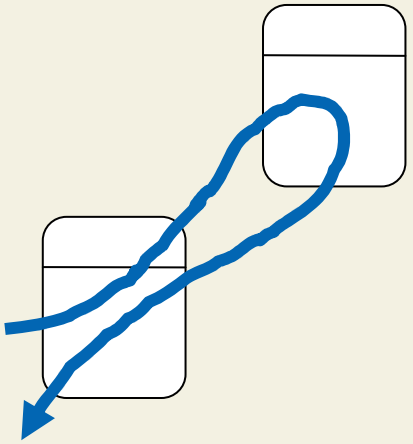
Threads

- Execution threads are **sequences of atomic actions** during a program execution
- Concurrent programs can have **more than one thread**
- Execution of threads can be **parallel** (on several processors) or **virtually parallel** (on one processor)
- A **scheduler** maps threads to processors

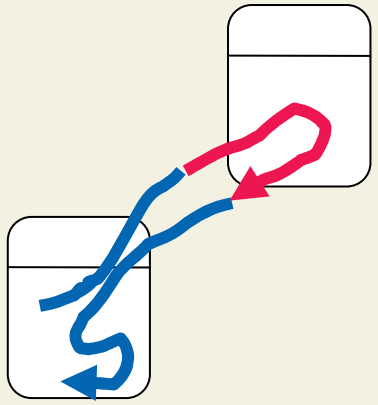


Concurrency in OO-Programs

- Passive objects
  - Threads **pass through different objects** (by method invocations)
  - **Several threads** executed **on one object** possible



- Active objects
  - **Each object has** its own **thread**
  - Upon method invocation, the thread of the target object serves the request
  - **At most one thread** executed on one object



Threads and Passive Objects

- Threads have to be created, started, synchronized, and controlled
- Threads are represented by special objects
- Method “start” starts new thread and returns immediately

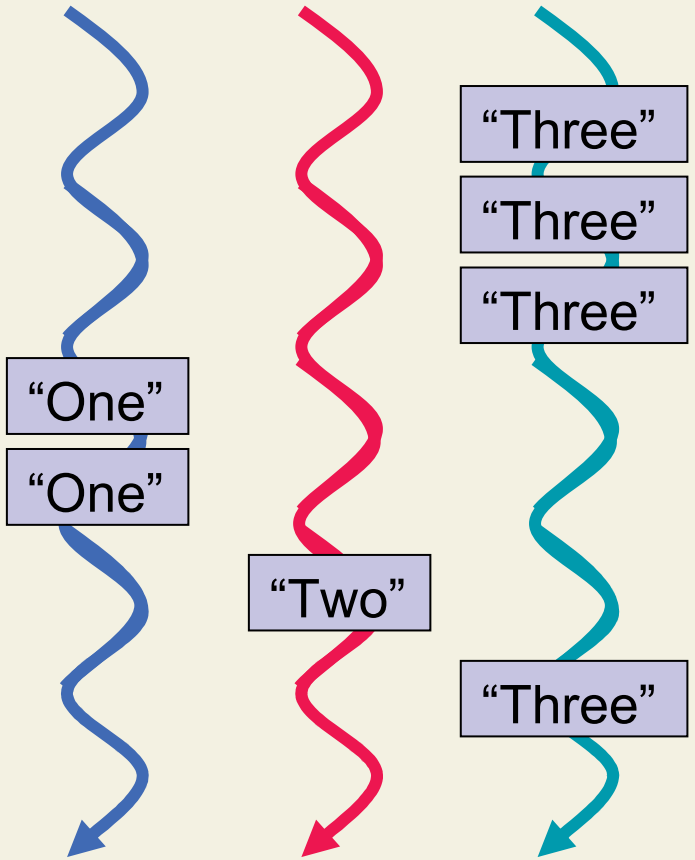
```
interface Runnable {
  void run( );
}
```

```
class Thread
          implements  Runnable {
  Thread( Runnable target )   { … }
  void run( )                 { … }
  native void start( );
  void interrupt( )           { … }
  …
}
```

Example

```
class Printer implements Runnable {
  String val;
  Printer( String s ) { val = s; }
  void run( ) {
    while( true )
      System.out.println( val );
  }
}
```

```
new Thread( new Printer( “One” ) ).start( );
new Thread( new Printer( “Two” ) ).start( );
new Thread( new Printer( “Three” ) ).start( );
```



8. Concurrency

8.1 Threads

8.2 Synchronization

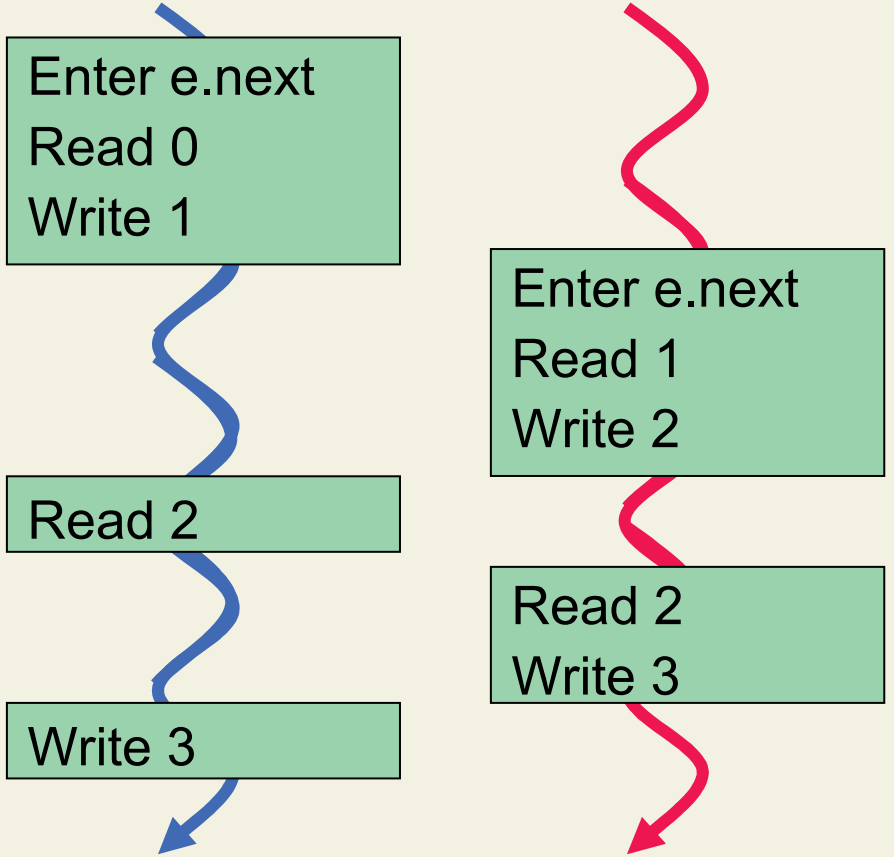
8.3 Active Objects

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Data Races

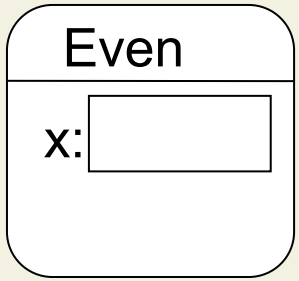
- Access to **common resources** (e.g., variables) can lead to unwanted behavior
- Execution is divided into **critical** and non-critical **sections**
- Execution of **critical sections** should be **mutually exclusive**

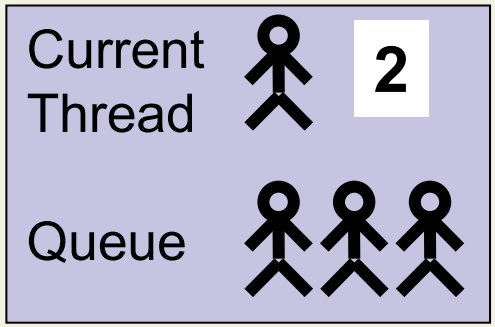
```
class Even {
  private int x;
  void next( ) { x++; x++; }
}
```



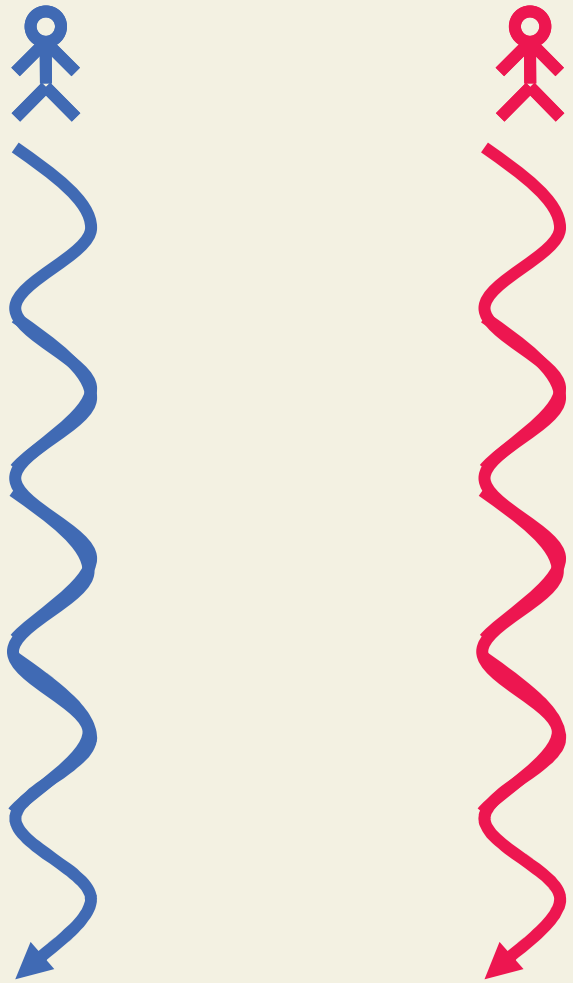
Object-Oriented Monitors

- Each object has a monitor
- Execution of synchronized methods requires lock of monitor
  - Lock is obtained upon invocation
  - Lock is released upon termination
  - Other threads have to wait
- Monitor keeps track of
  - Thread that has locked the monitor
  - Number of locks of this thread
  - Queue of blocked threads

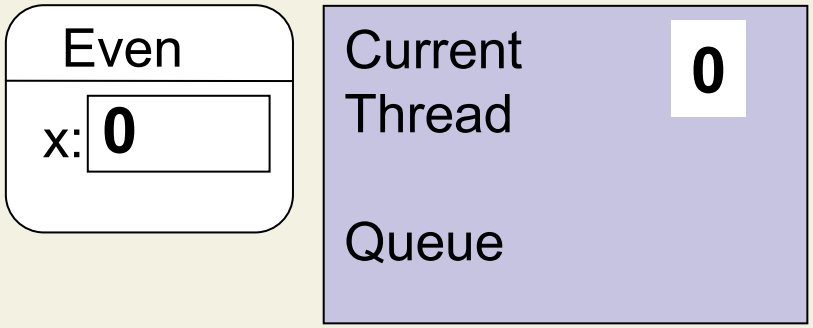




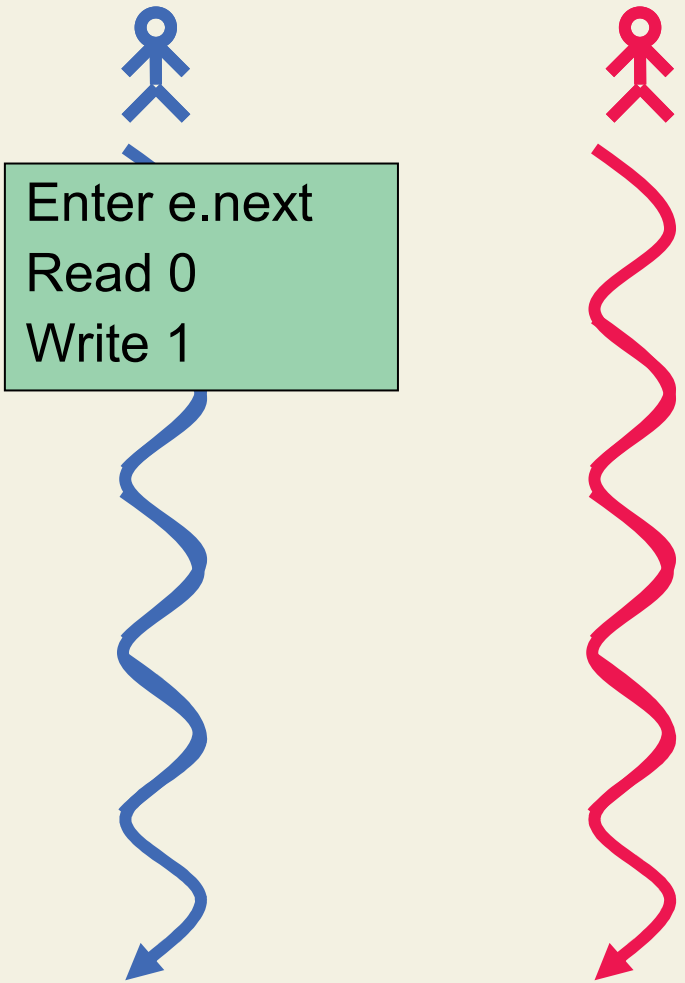
Preventing Data Races



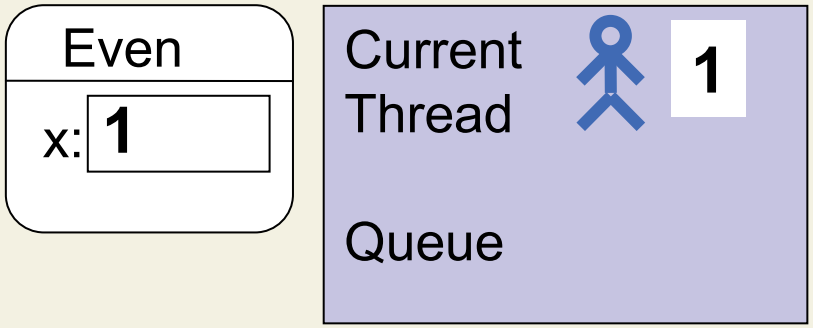
```
class Even {
  private int x;
  void next( ) { x++; x++; }
}
```



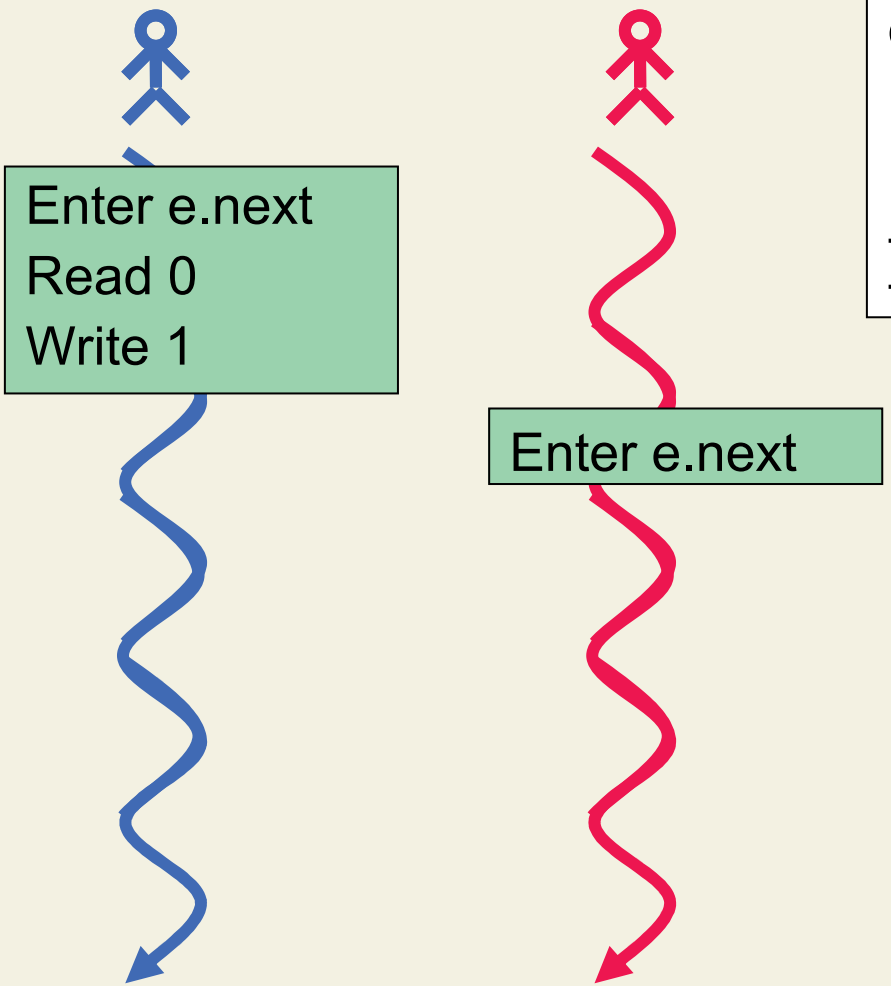
Preventing Data Races



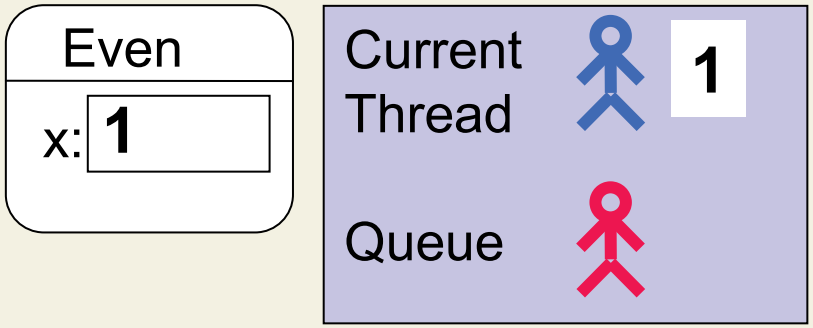

```
class Even {
  private int x;
  void next( ) { x++; x++; }
}
```



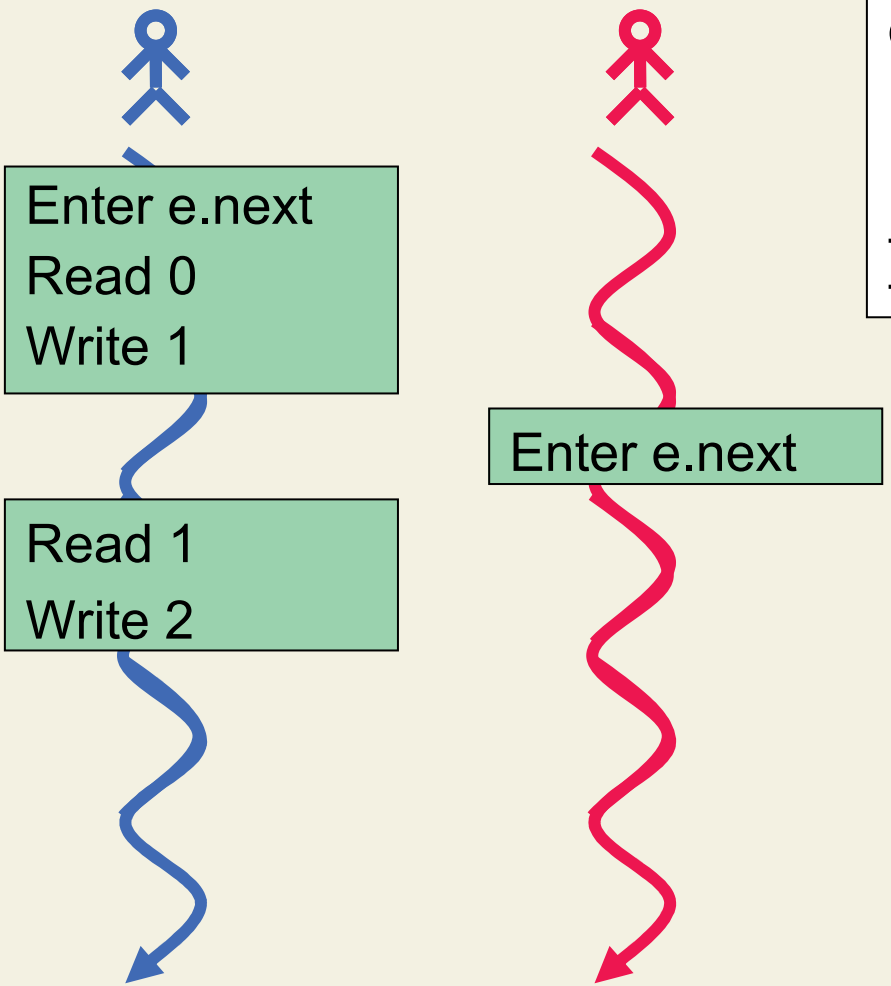
Preventing Data Races



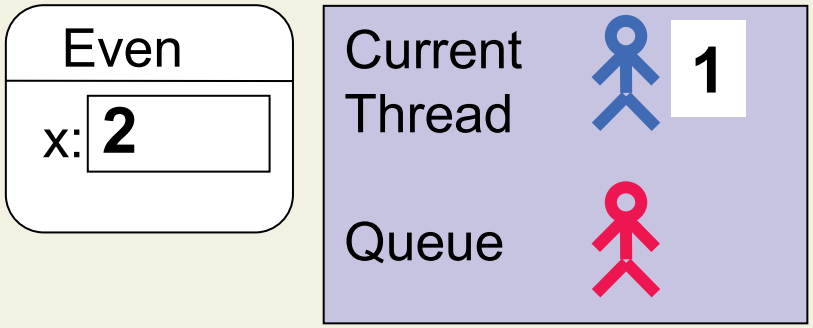

```
class Even {
  private int x;
  void next( ) { x++; x++; }
}
```



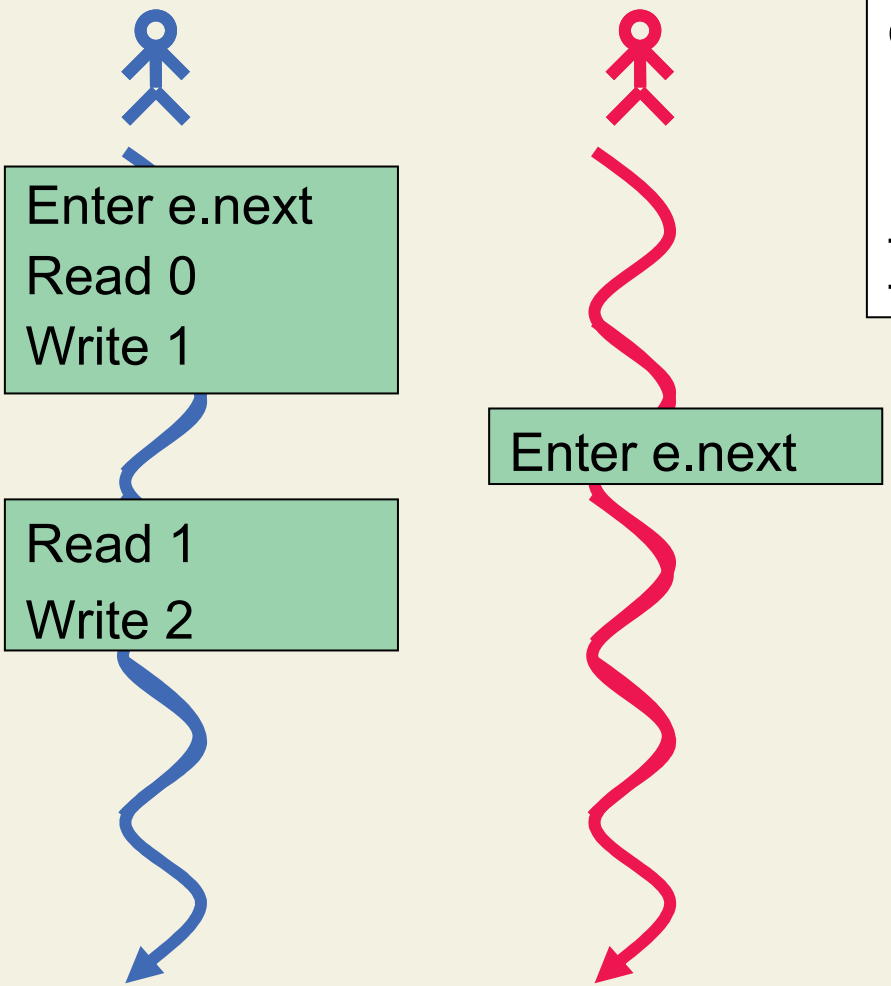
Preventing Data Races



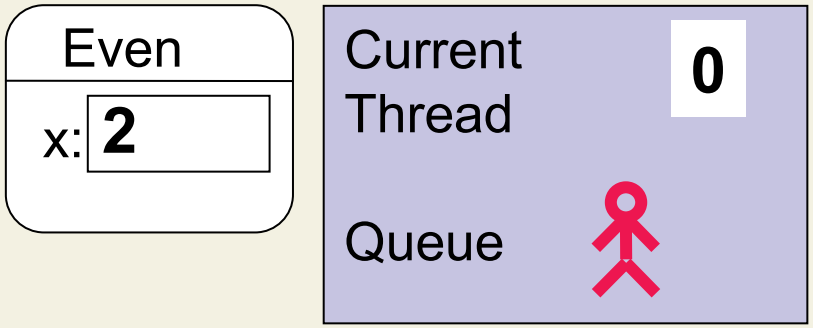

```
class Even {
  private int x;
  void next( ) { x++; x++; }
}
```



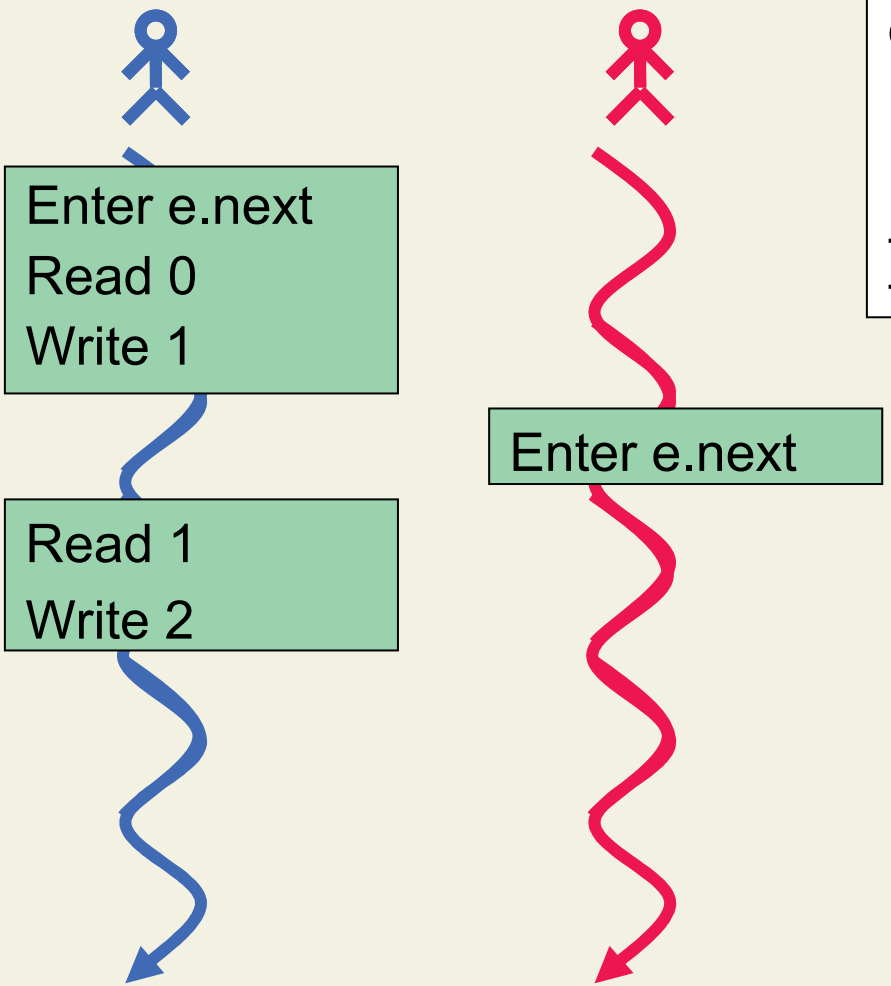
Preventing Data Races



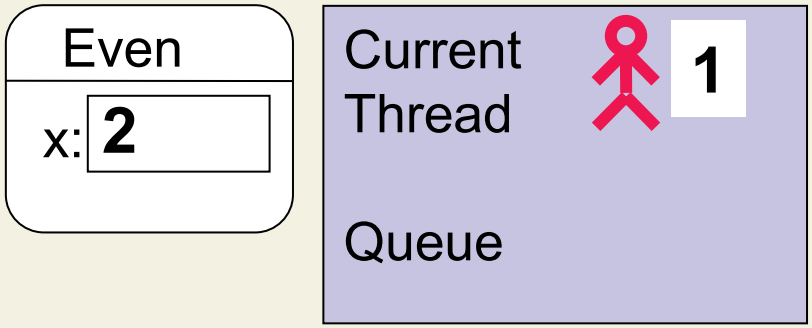

```
class Even {
  private int x;
  void next( ) { x++; x++; }
}
```



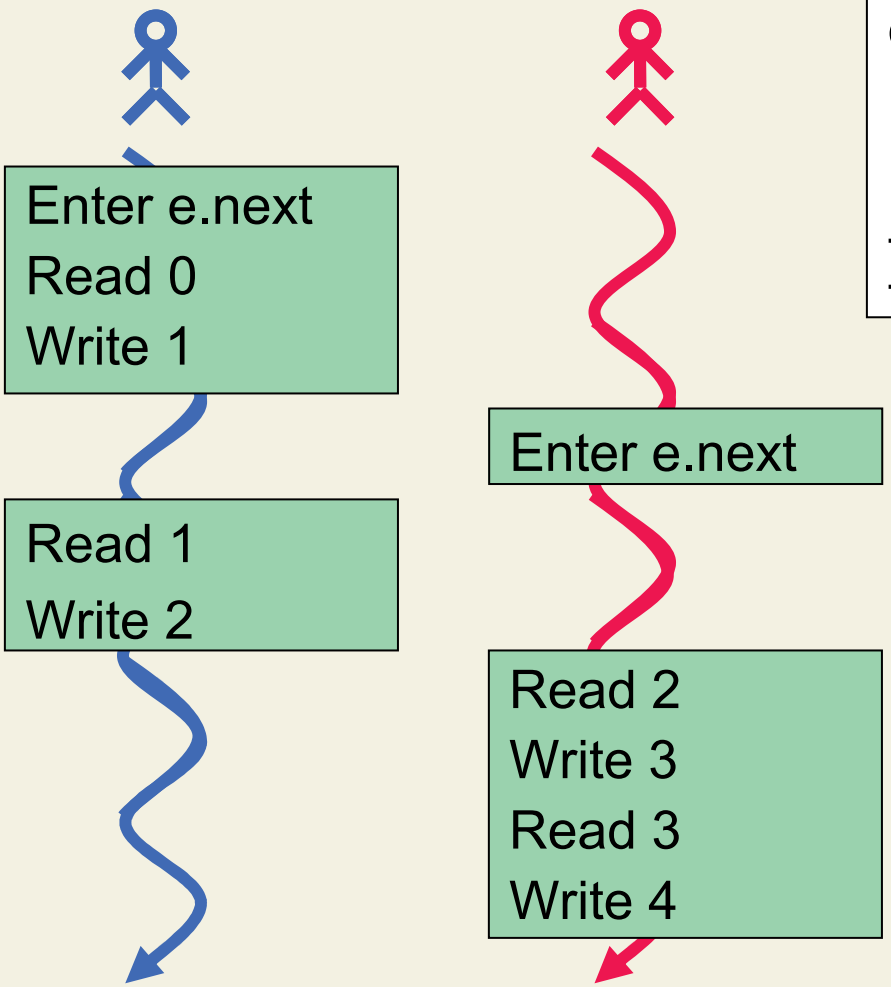
Preventing Data Races



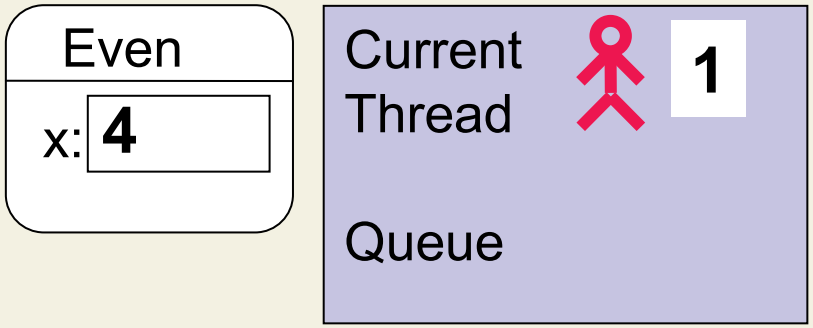

```
class Even {
  private int x;
  void next( ) { x++; x++; }
}
```



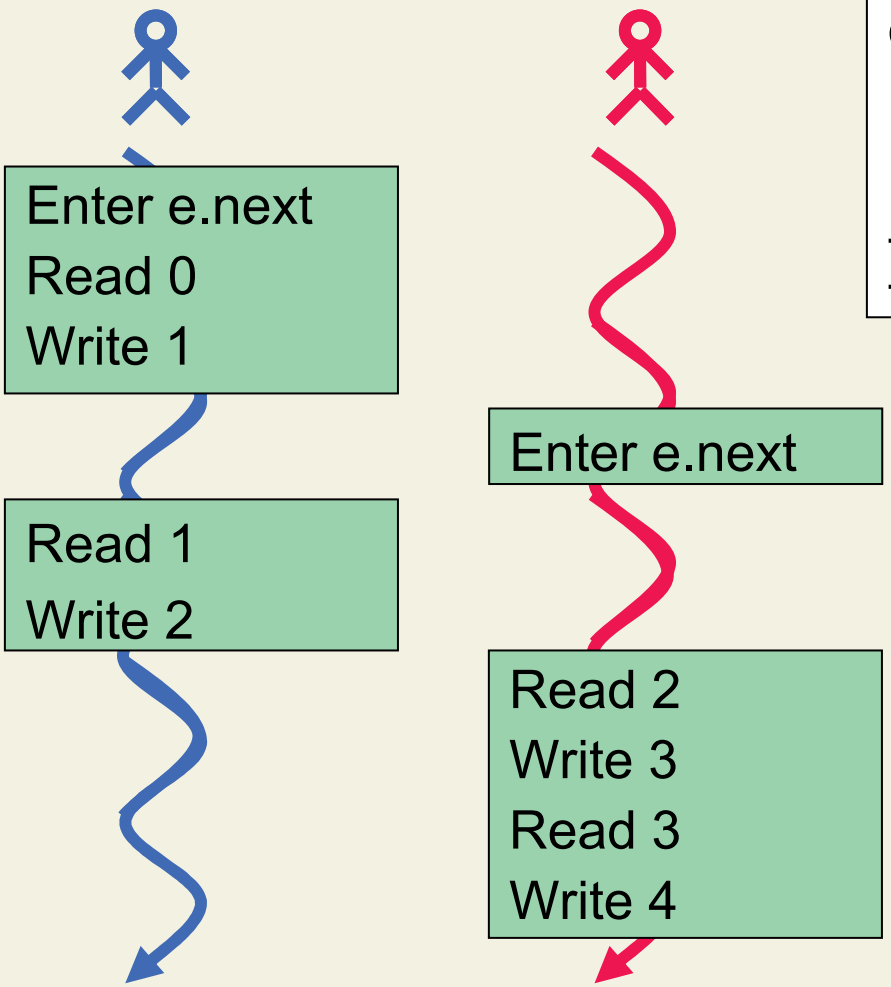
Preventing Data Races



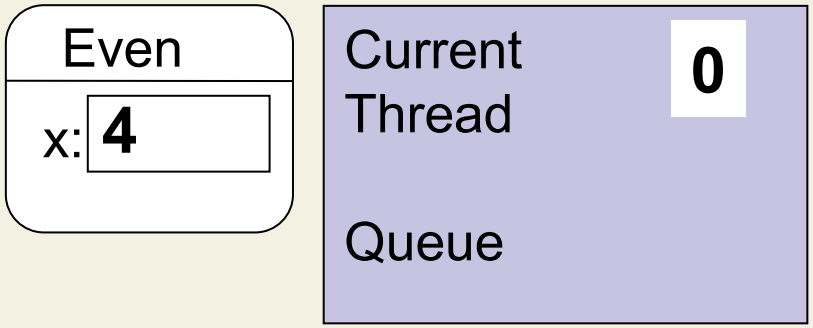

```
class Even {
  private int x;
  void next( ) { x++; x++; }
}
```



Preventing Data Races

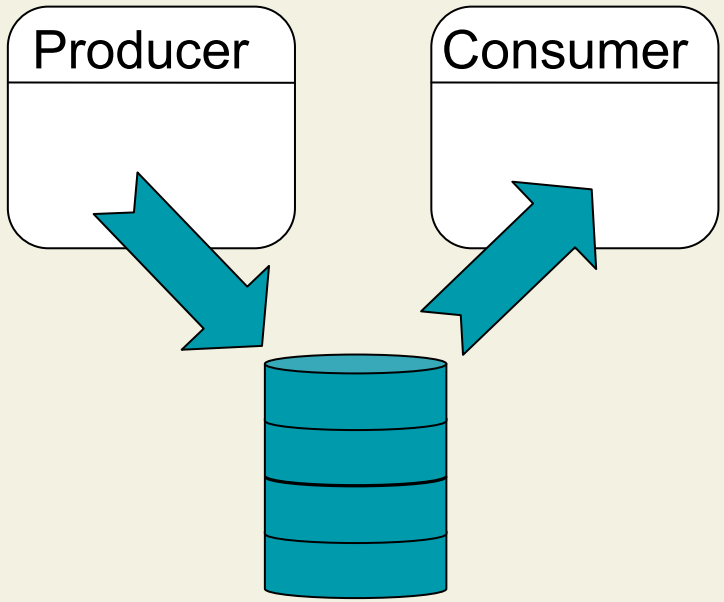



```
class Even {
  private int x;
  void next( ) { x++; x++; }
}
```



Cooperating Threads

- One thread has to wait until another thread has performed an action
- Wait condition usually depends on commonly used variables (occurs inside synchronized methods)



Producer-Consumer Example

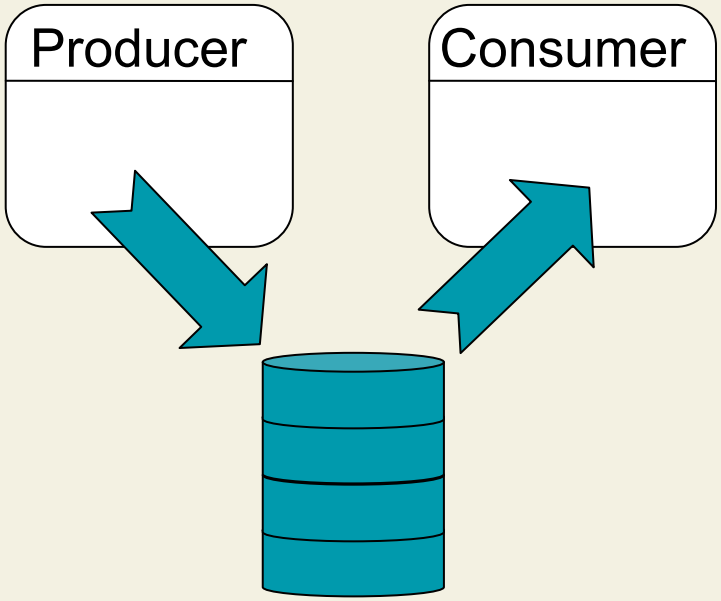
```
class Buffer {
  …
  synchronized boolean put( Prd p ) {
    if ( isFull( ) )        return false;
    …
    return true;
  }

  synchronized Prd get( ) {
    if ( isEmpty( ) )       return null;
    …
  }
}
```

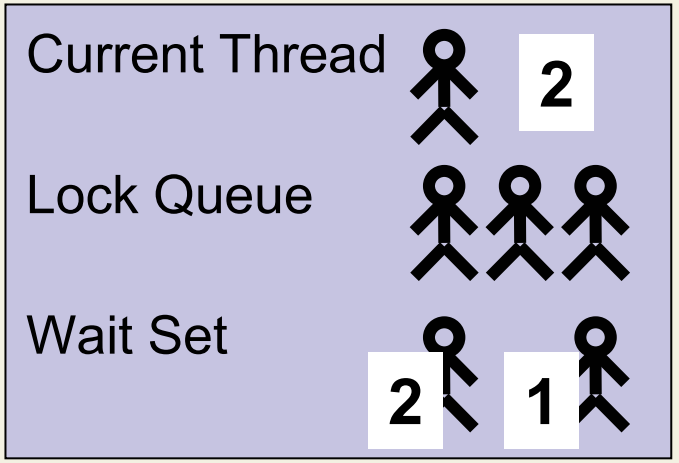
```
class Producer extends Thread {
  Buffer buf;
  void run( ) {
    while ( true ) {
      Prd p = new Prd( );
      while( buf.put( p ) == false )
        sleep( 1000 );
  }
}
```

```
class Consumer extends Thread {
  Buffer buf;
  void run( )
    { // analogous }
}
```

Wait and Notify

- Wait operation
  - Can be applied if a thread has locked a monitor
  - Puts thread into wait state and adds thread to wait set
  - Releases lock
- Notify operation
  - Can be applied if a thread has locked a monitor
  - Chooses one thread from wait set and re-enables it for scheduling





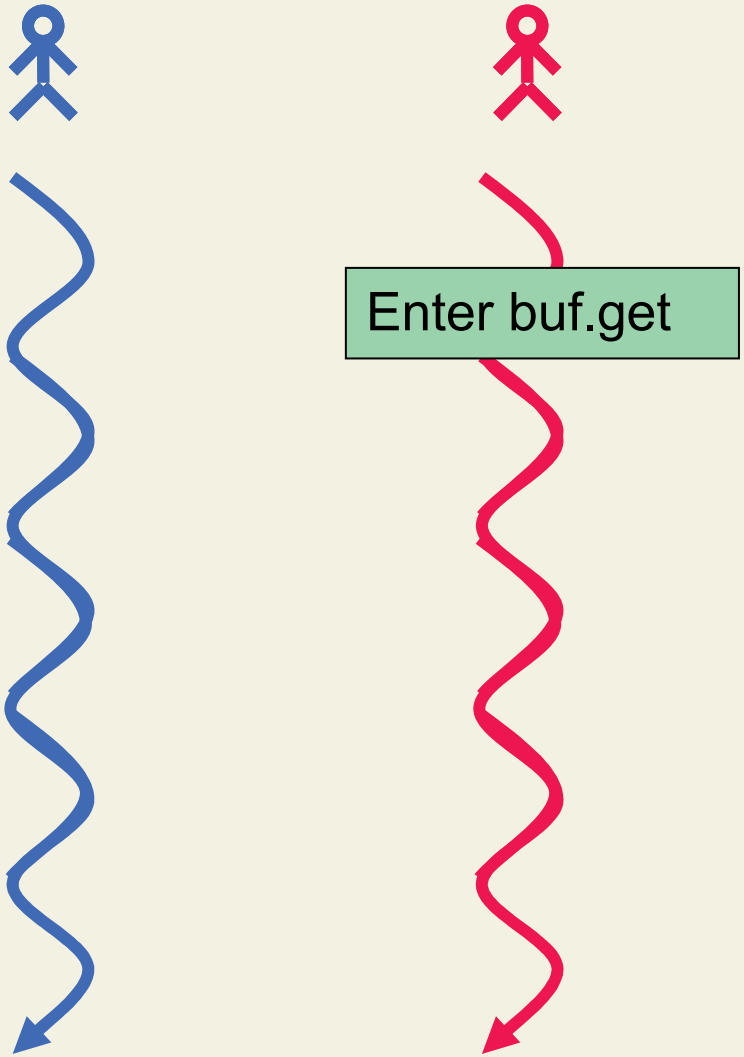
Producer-Consumer Example Revisited

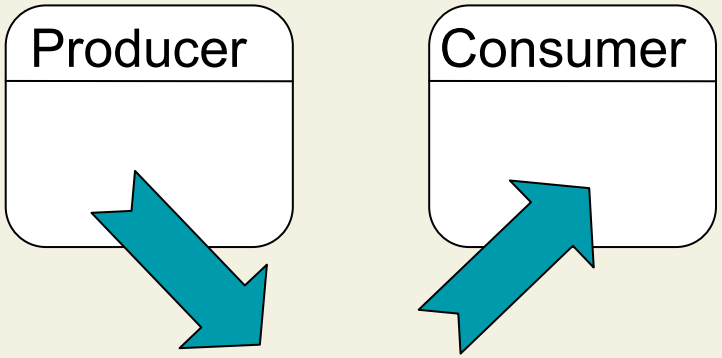
```
class Buffer {
  …
  synchronized void put( Prd p ) {
    if ( isFull( ) )       wait( );
    …
    notify( );
  }

  synchronized Prd get ( ) {
    if ( isEmpty( ) )   wait( );
    …
    notify( );
  }
}
```

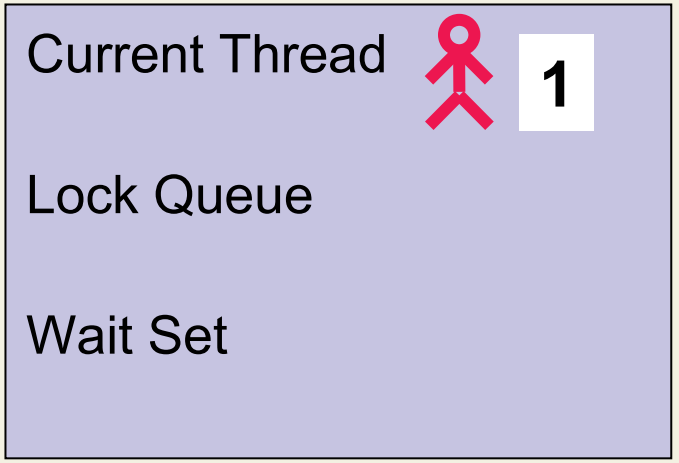
```
class Producer extends Thread {
  Buffer buf;
  void run( ) {
    while ( true )
      buf.put( new Prd( ) );
  }
}
```

```
class Consumer extends Thread {
  Buffer buf;
  void run( ) {
    while ( true )
    buf.get( );
  }
}
```

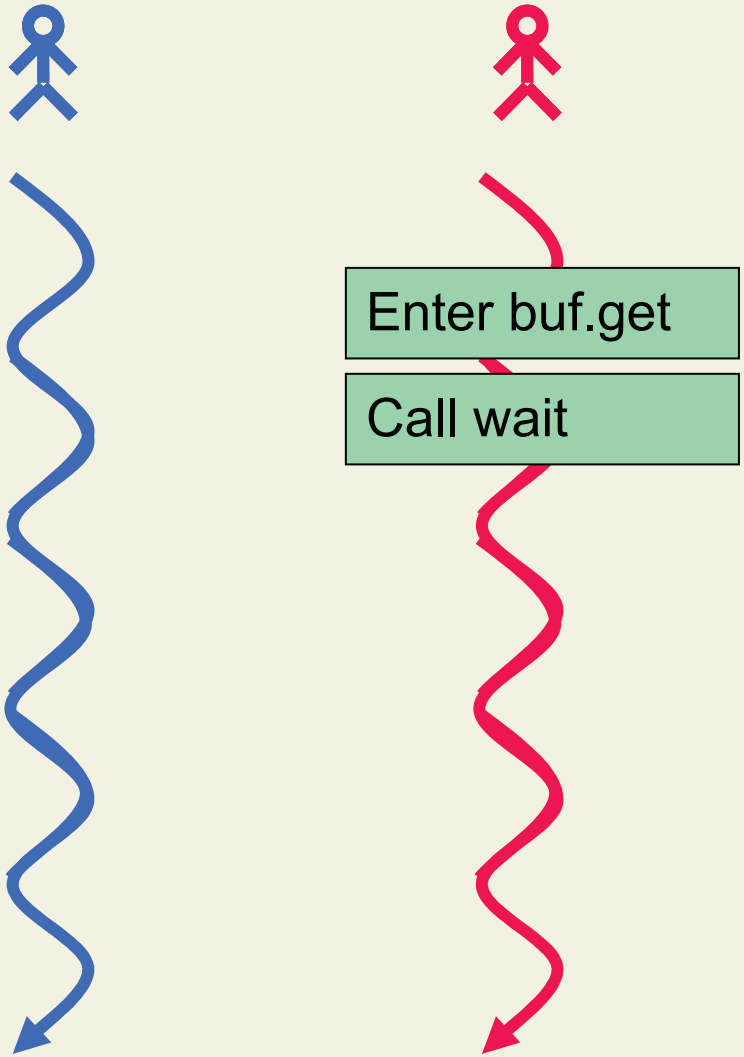
Producer-Consumer Example Revisited

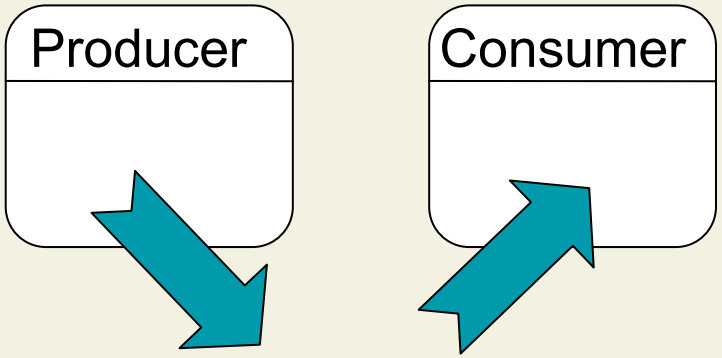






Producer-Consumer Example Revisited



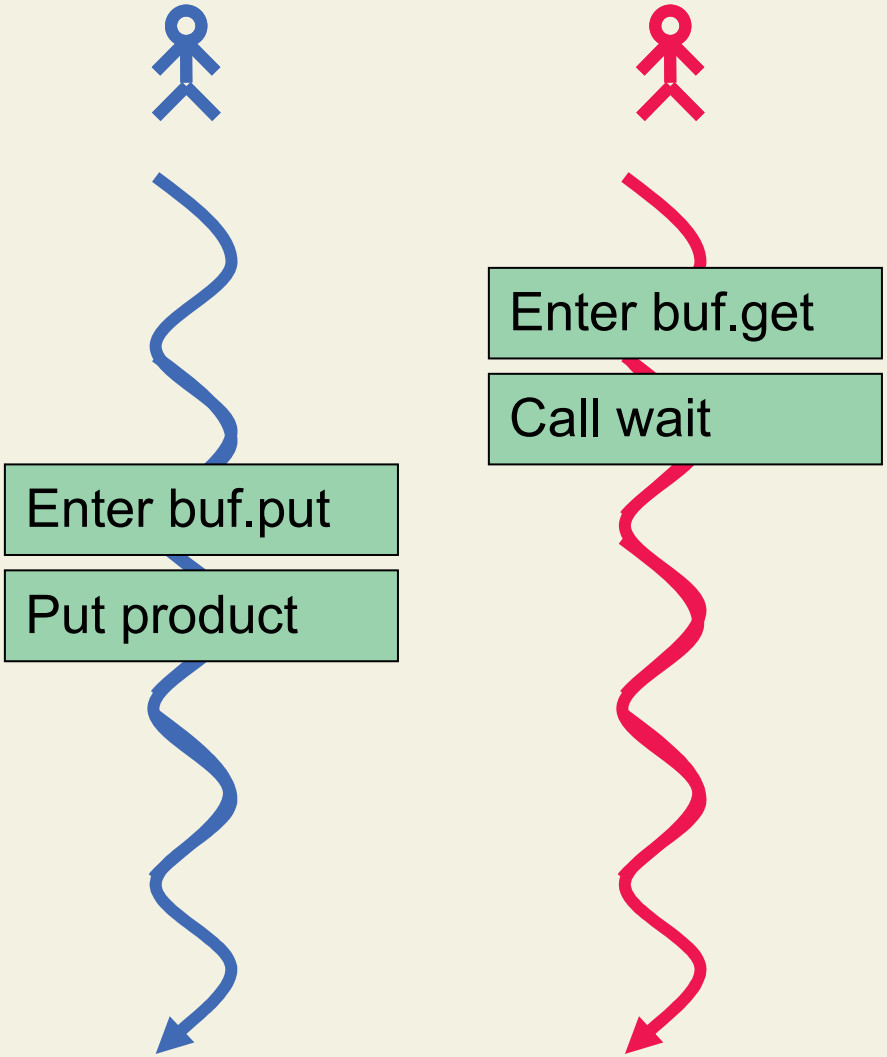


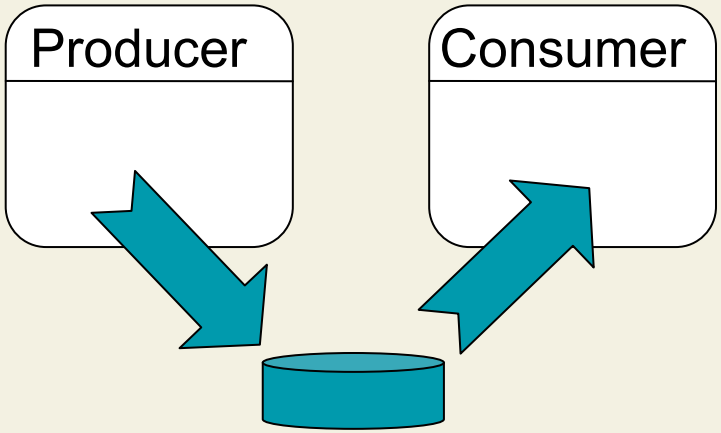
Current Thread 0

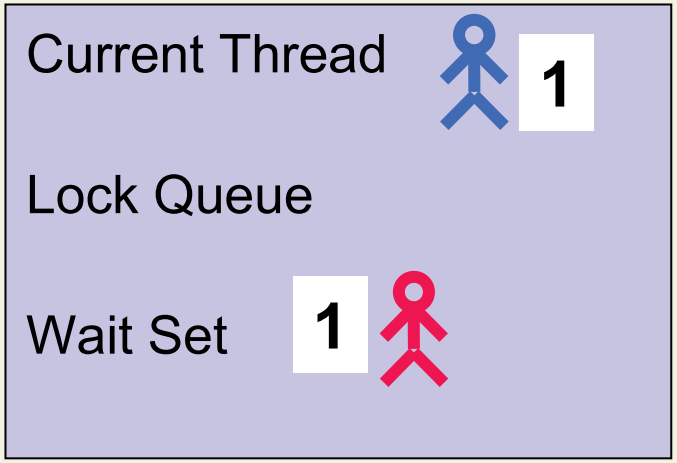
Lock Queue

Wait Set 1

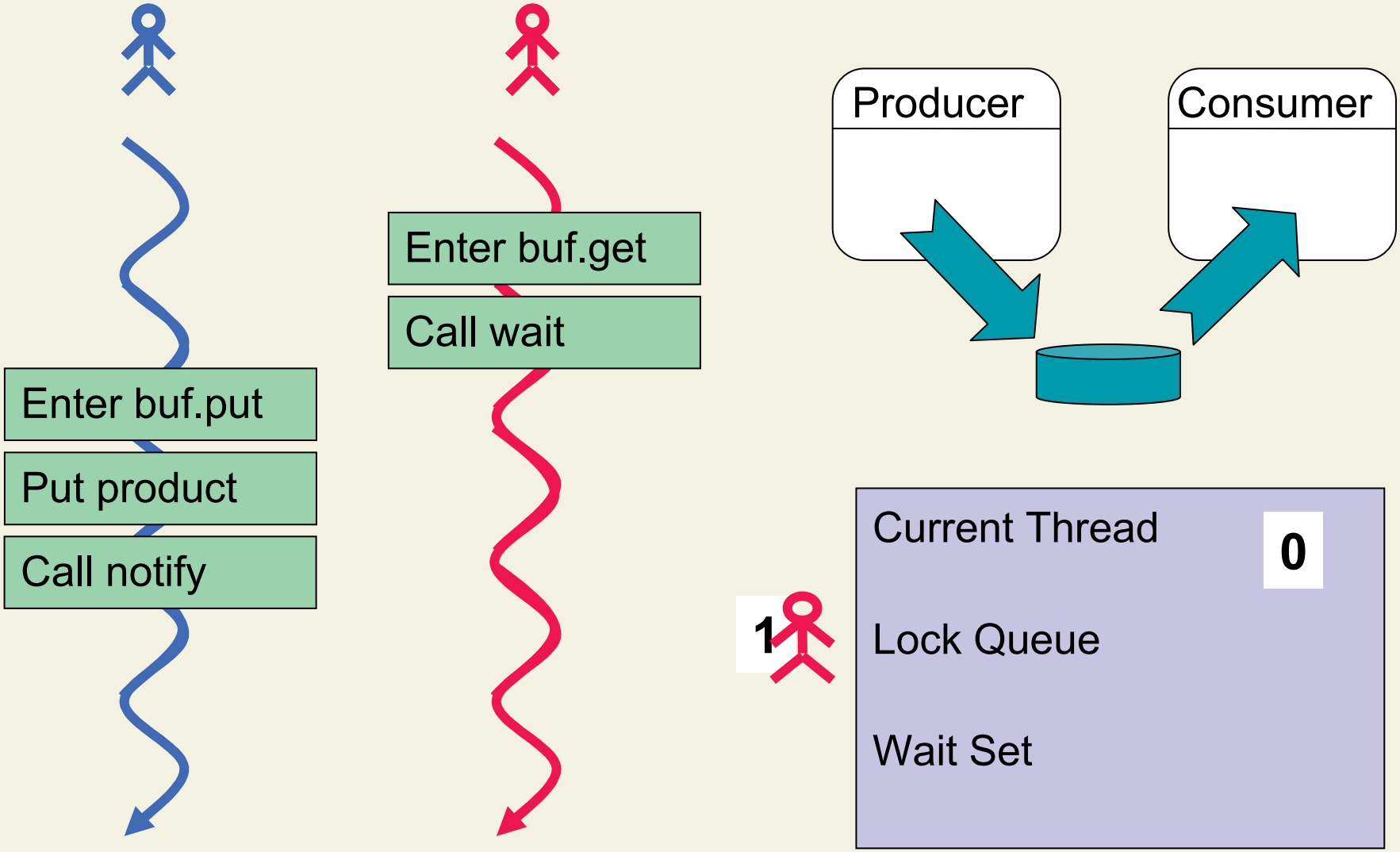
Producer-Consumer Example Revisited



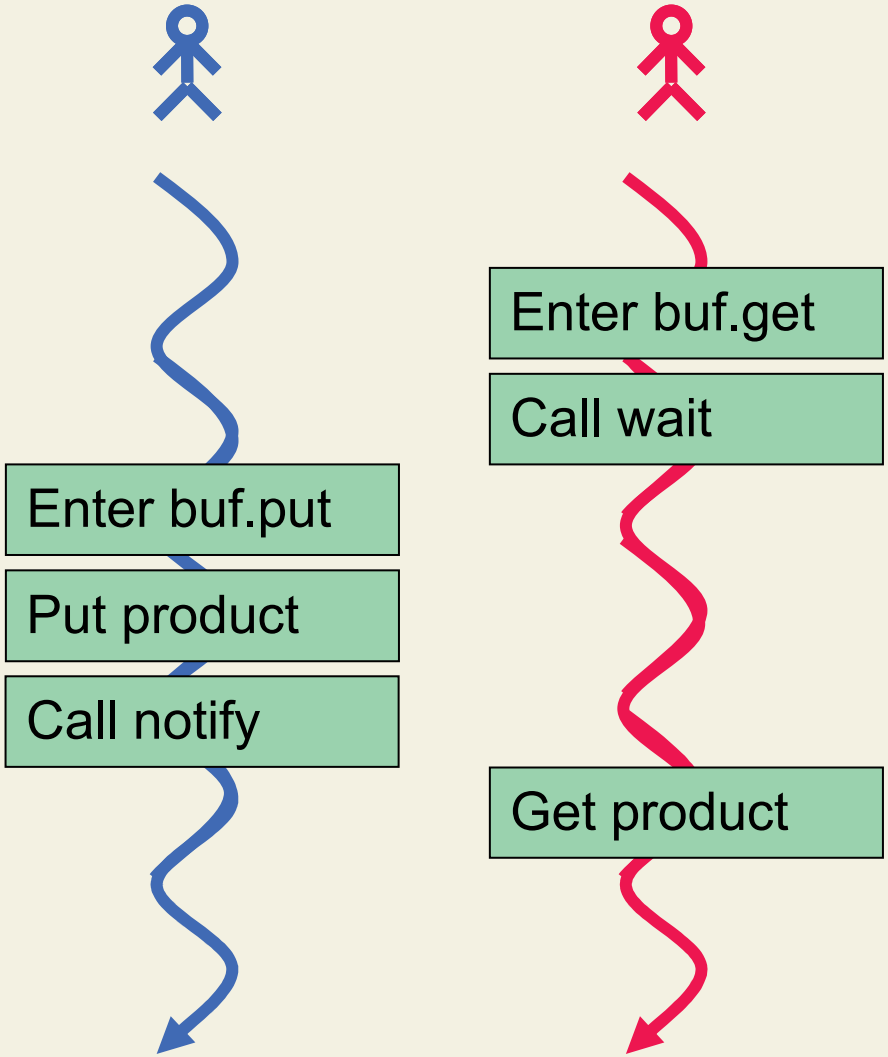


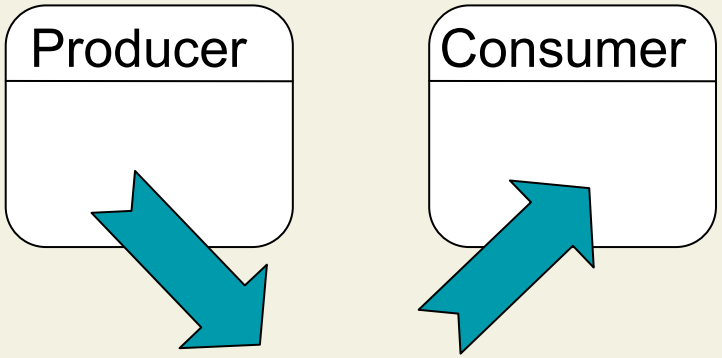


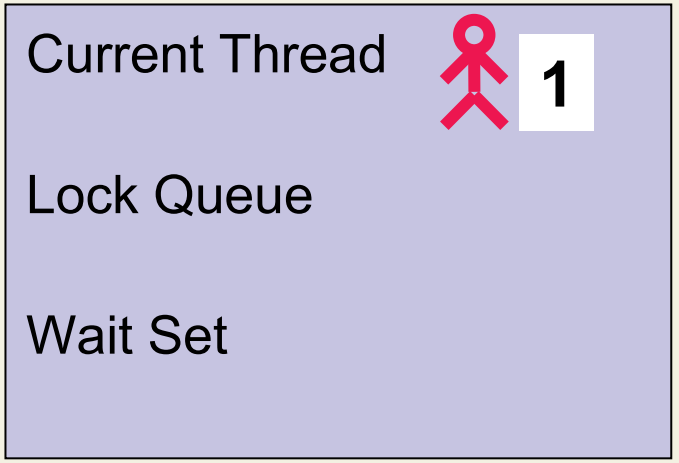
Producer-Consumer Example Revisited



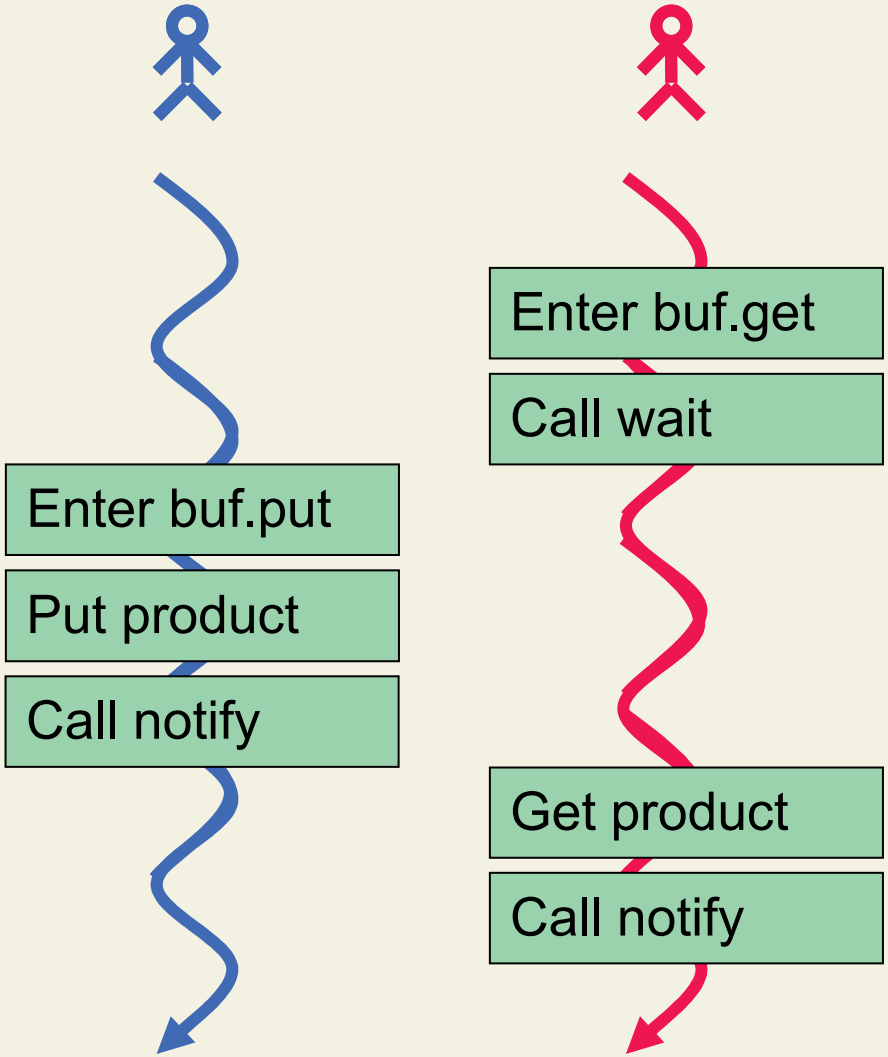
Producer-Consumer Example Revisited

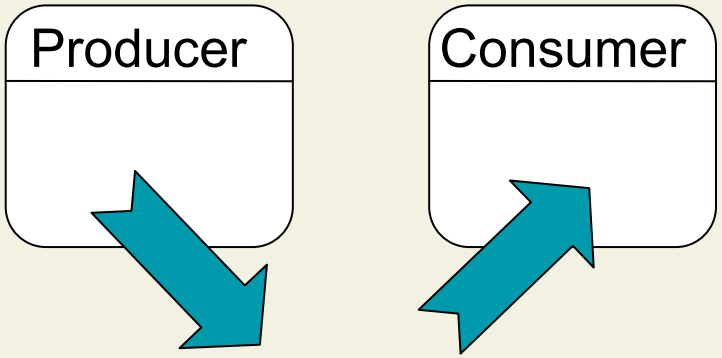






Producer-Consumer Example Revisited





Current Thread **0**

Lock Queue

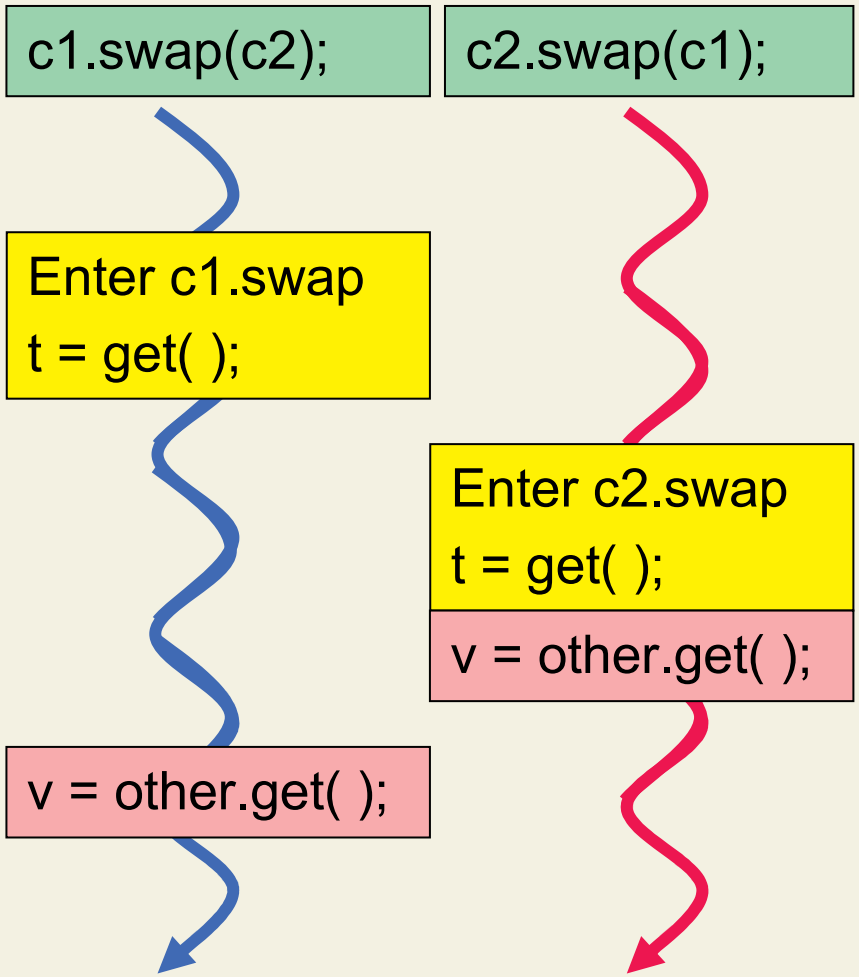
Wait Set

Safety and Liveness

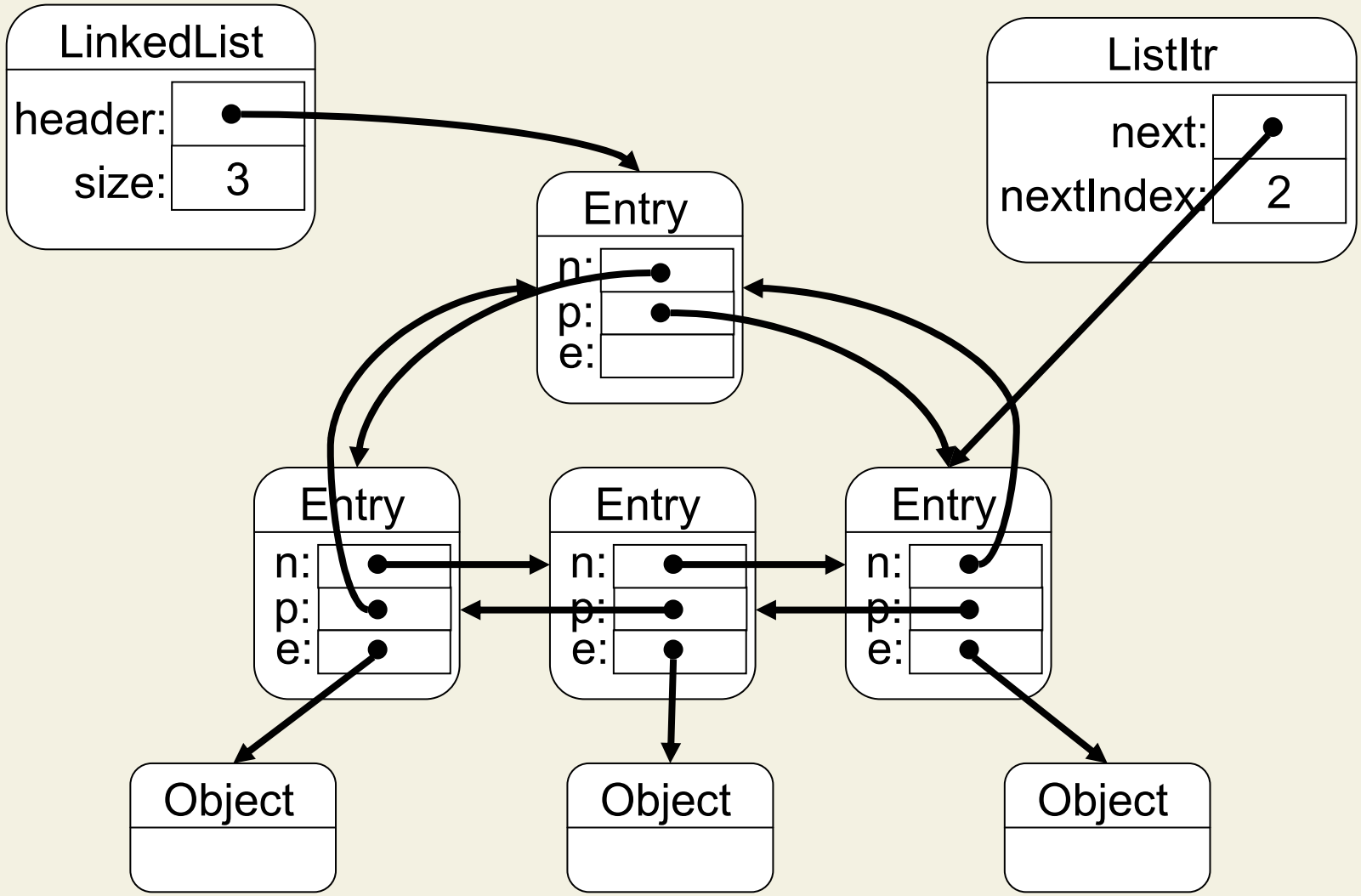
- Safety
  - **“Nothing bad ever happens”**
  - To perform method actions only when in consistent states
  - Achieved by mutual exclusion
- Liveness
  - **“Something eventually happens”**
  - Every called method should eventually execute
  - Avoiding deadlocks
  - Avoiding unfair scheduling (not guaranteed in Java)

Deadlock Example

```
class Cell {
  private long value;
  synchronized long get( )
    { return value; }
  synchronized void set( long v )
    { value = v; }
  synchronized void
          swap( Cell other ) {
    long t = get( );
    long v = other.get( );
    set( v );
    other.set( t );
  }
}
```

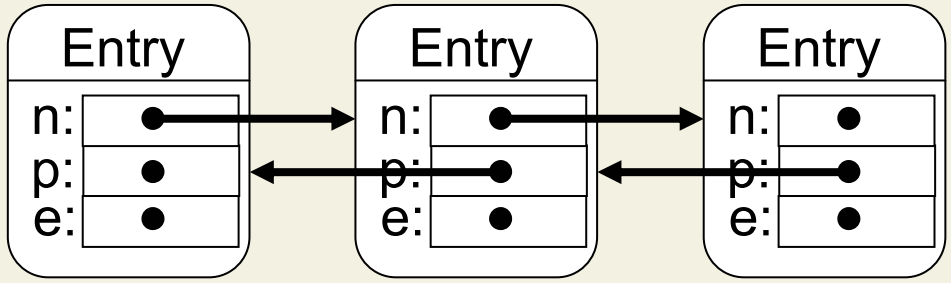


Synchronization with Object Structures



Synchronization on all Objects

- Possible solution: Make the methods of all representation objects **synchronized**
- Disadvantages
  - Direct attribute access must be synchronized separately
  - Might easily lead to deadlocks



Thread1: Traverses list from front to back

Thread2: Traverses list from back to front

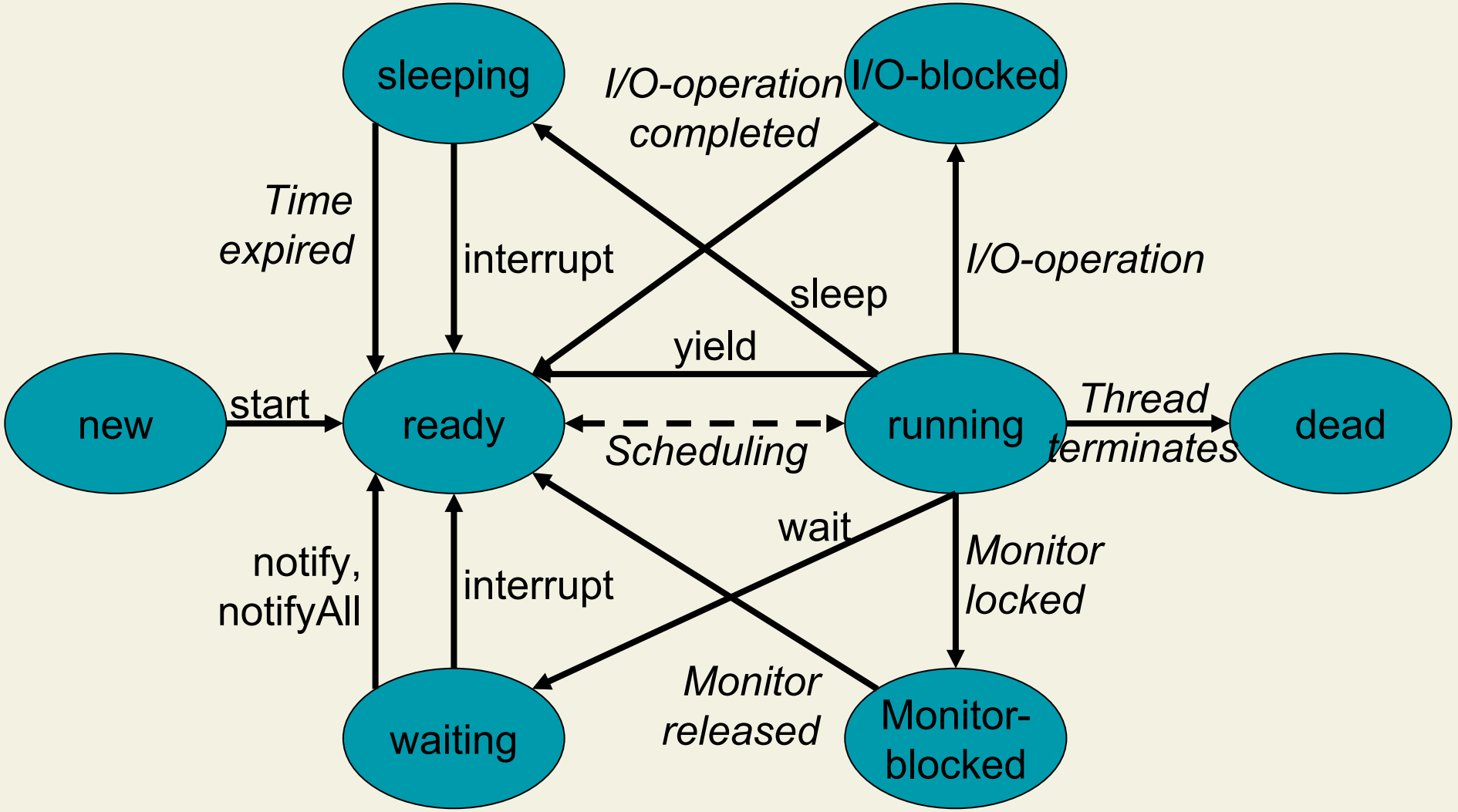
Central Objects of Synchronization

- Possible guideline: To access data structure, **lock on one designated object** (e.g., the owner object) must be obtained
- Disadvantages
  - No concurrent operations on data structure
  - Clients might not follow the guideline (encapsulation!)

```
class LinkedList {
  Entry header;
  int size;
  synchronized void
          add(Object o ) { … }
}
```

```
class ListItr {
  Entry next;   int nextIndex;
  LinkedList theList;
  synchronized Object getVal( ) {
    synchronized( theList ) { … }
  }
}
```

States of Java Threads



Summary: Object-Oriented Threads

- Threads are objects
  - Threads can be controlled by method invocations
  - Threads can be specialized by inheritance
- Each object has an associated monitor
  - Operations are inherited from Object (wait, notify, etc.)
  - Synchronization works well for individual objects, but is especially difficult for object structures
- Objects are passive
  - No real support for inherently concurrent object model