Konzepte objektorientierter Programmierung
– Lecture 8 –

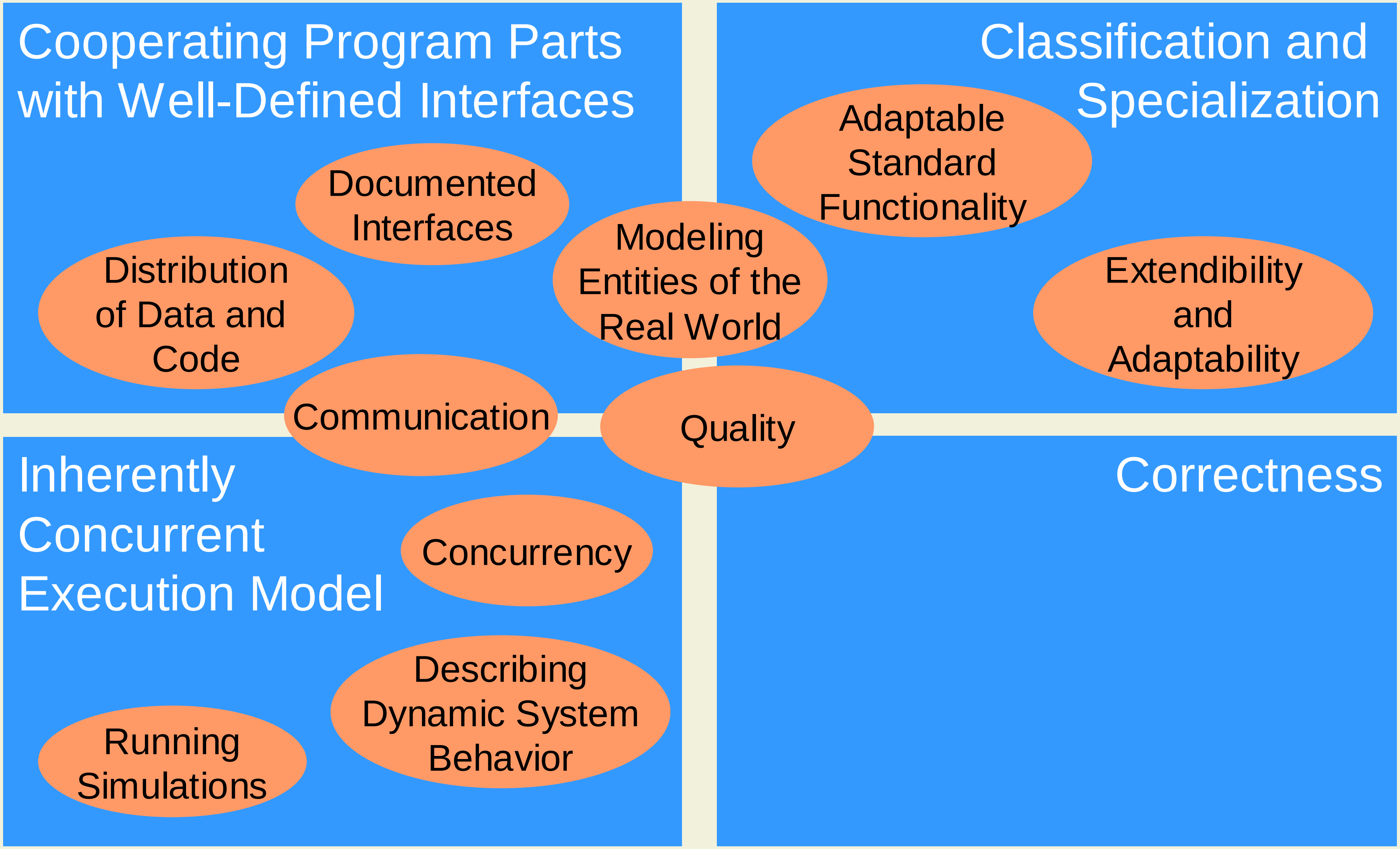
Prof. Dr. Peter Müller

Software Component Technology

Wintersemester 06/07

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

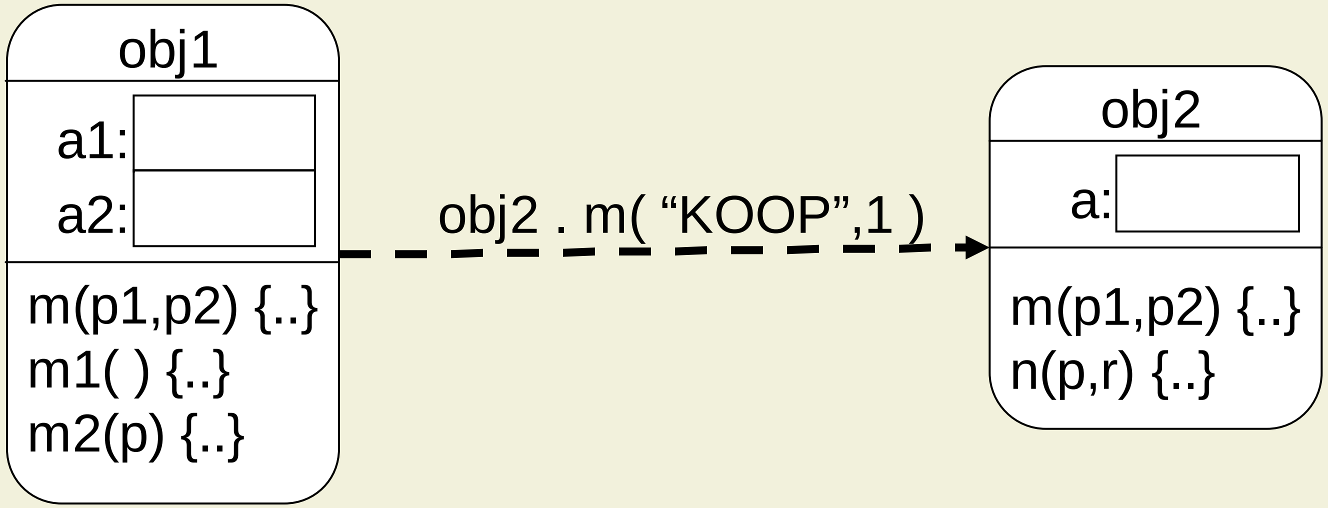
Correctness

→ Interfaces
→ Encapsulation
→ Simple, powerful concepts

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

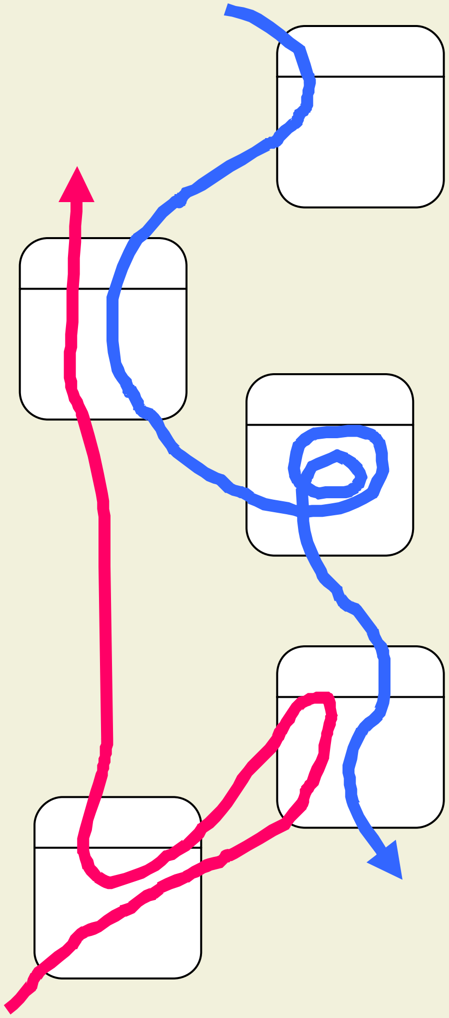
Objectives

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

8. Concurrency

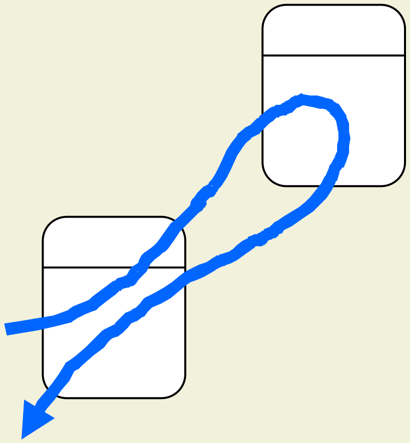
Threads

- 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

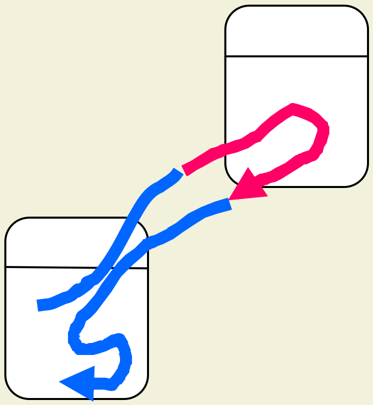


Concurrency in OO-Programs

- 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



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

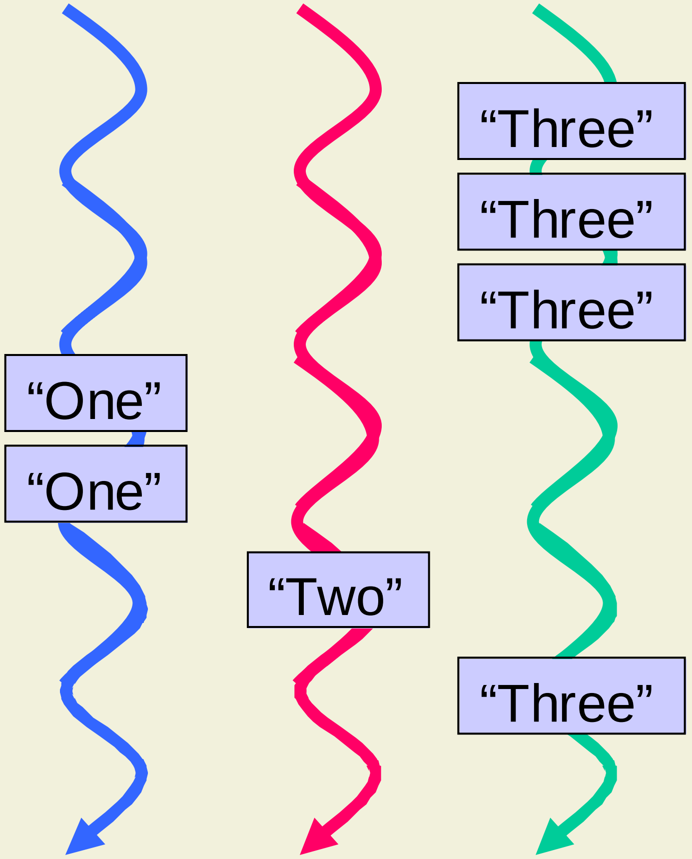
```
interface Runnable {
  void run( );
}
```

```
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( );
```



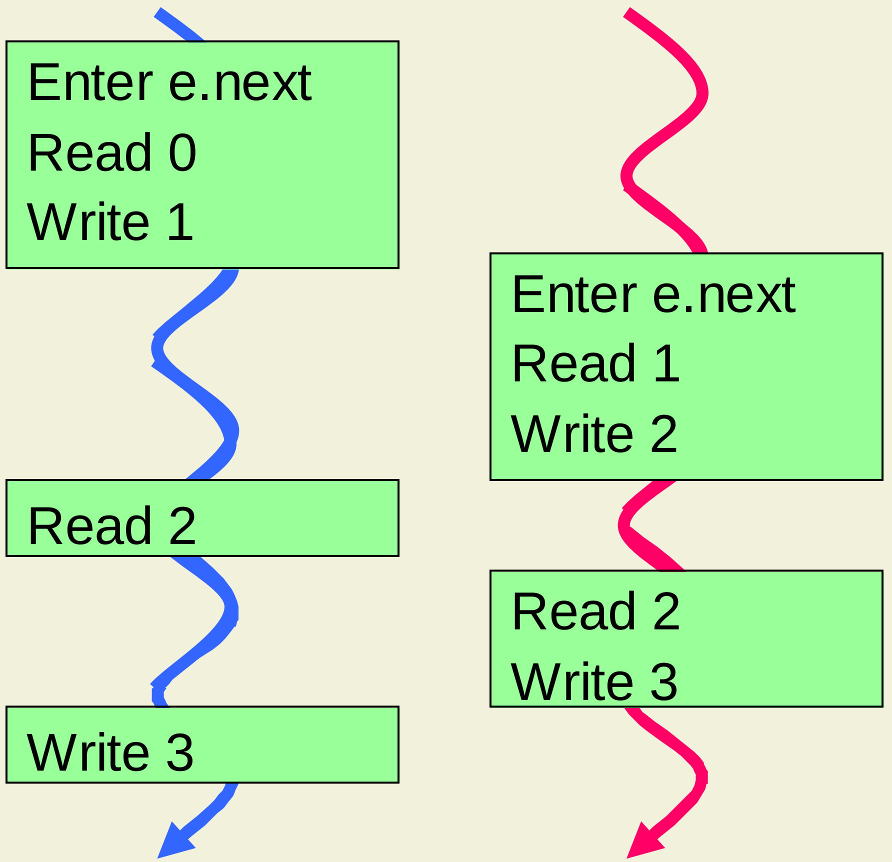
ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

8. Concurrency

Data Races

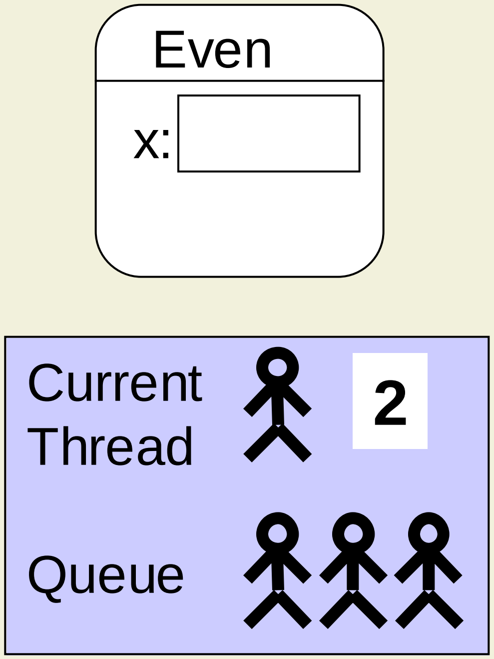
- Access to **common resources** (e.g., variables) can lead to unwanted behavior
- Execution is divided into **critical** and non-critical **sections**
- Execution of **critical sections** should be **mutually exclusive**

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

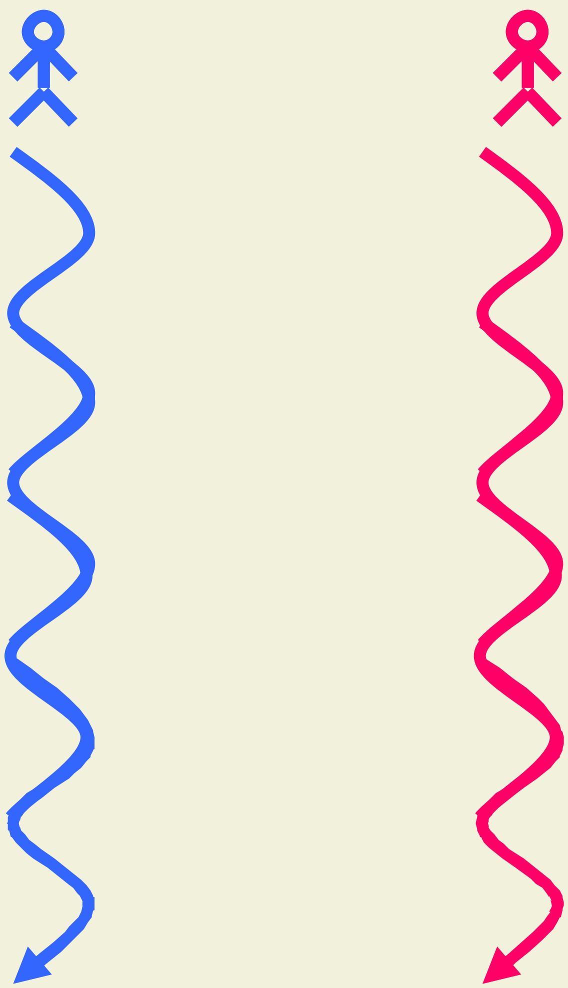


Object-Oriented Monitors

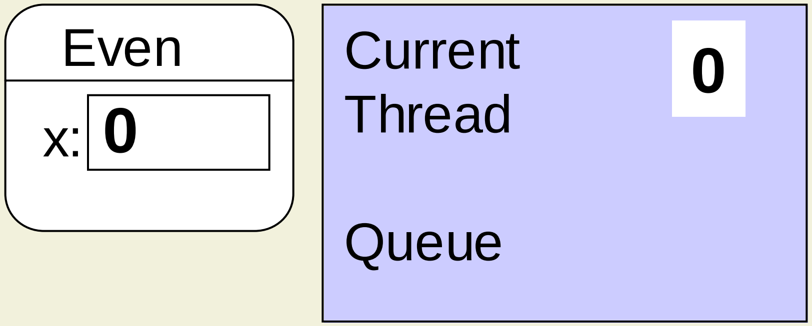
- Each object has a monitor
- Execution of synchronized methods requires lock of monitor
  - Lock is obtained upon invocation
  - Lock is released upon termination
  - Other threads have to wait
- Monitor keeps track of
  - Thread that has locked the monitor
  - Number of locks of this thread
  - Queue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



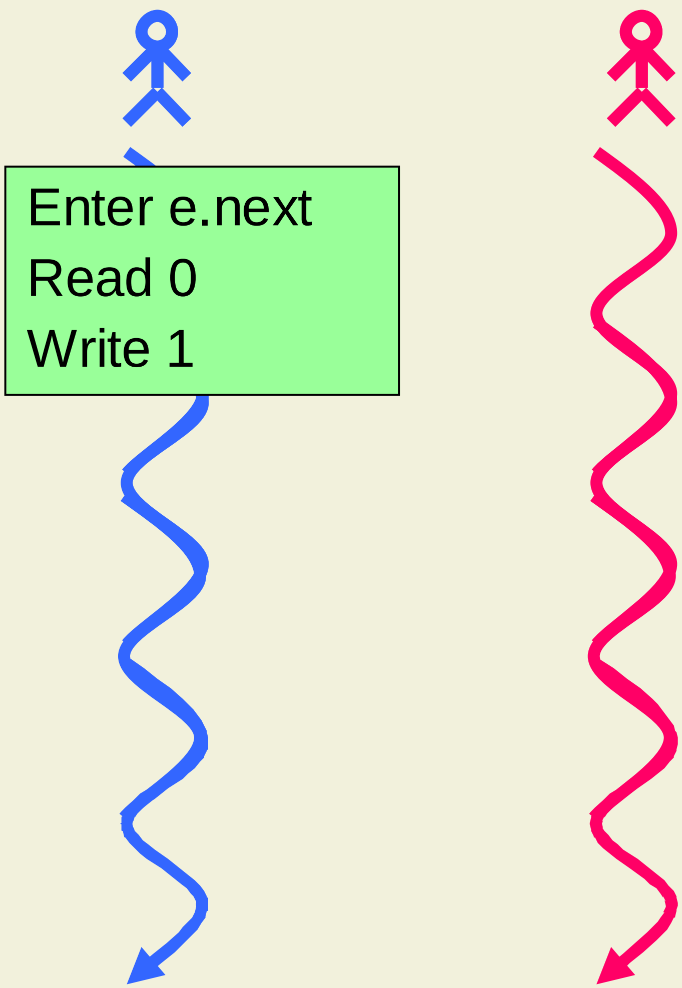
Preventing Data Races



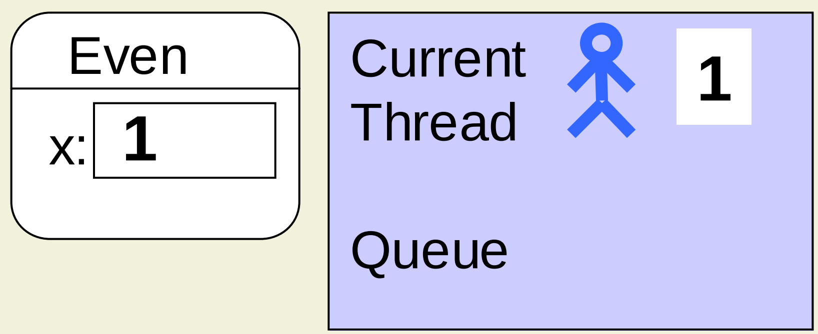
```
class Even {
  private int x;
  synchronized void next( )
     { x++; x++; }
}
```



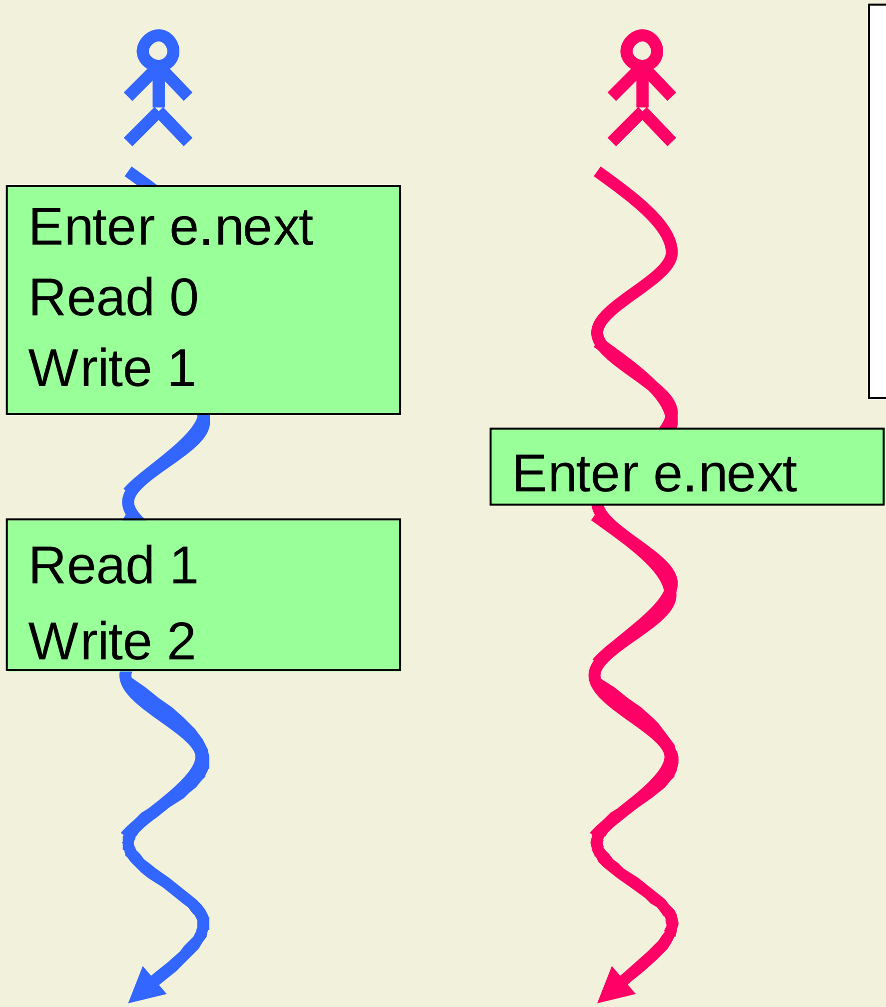
Preventing Data Races



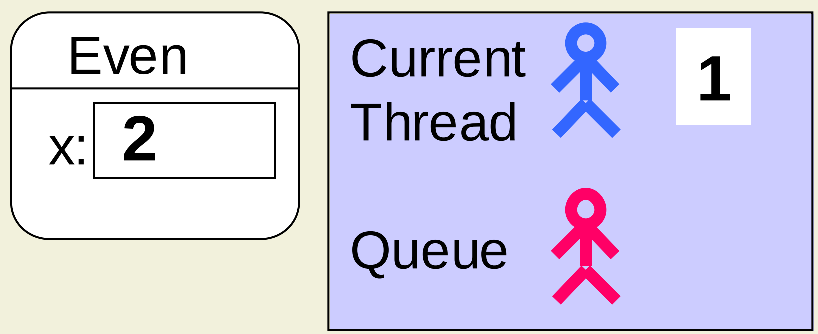

```
class Even {
  private int x;
  synchronized void next( )
      { x++; x++; }
}
```



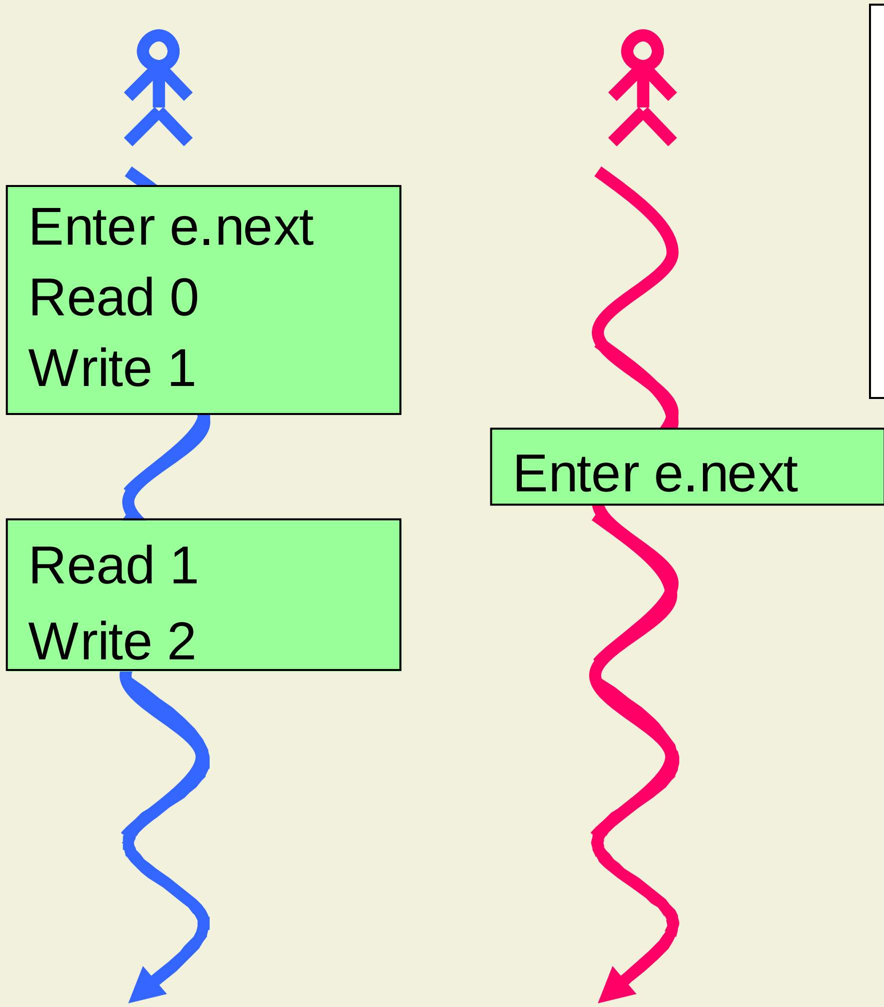
Preventing Data Races



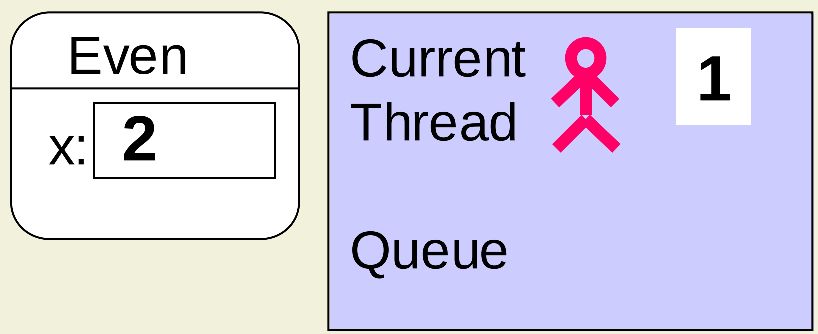

```
class Even {
  private int x;
  synchronized void next( )
      { x++; x++; }
}
```



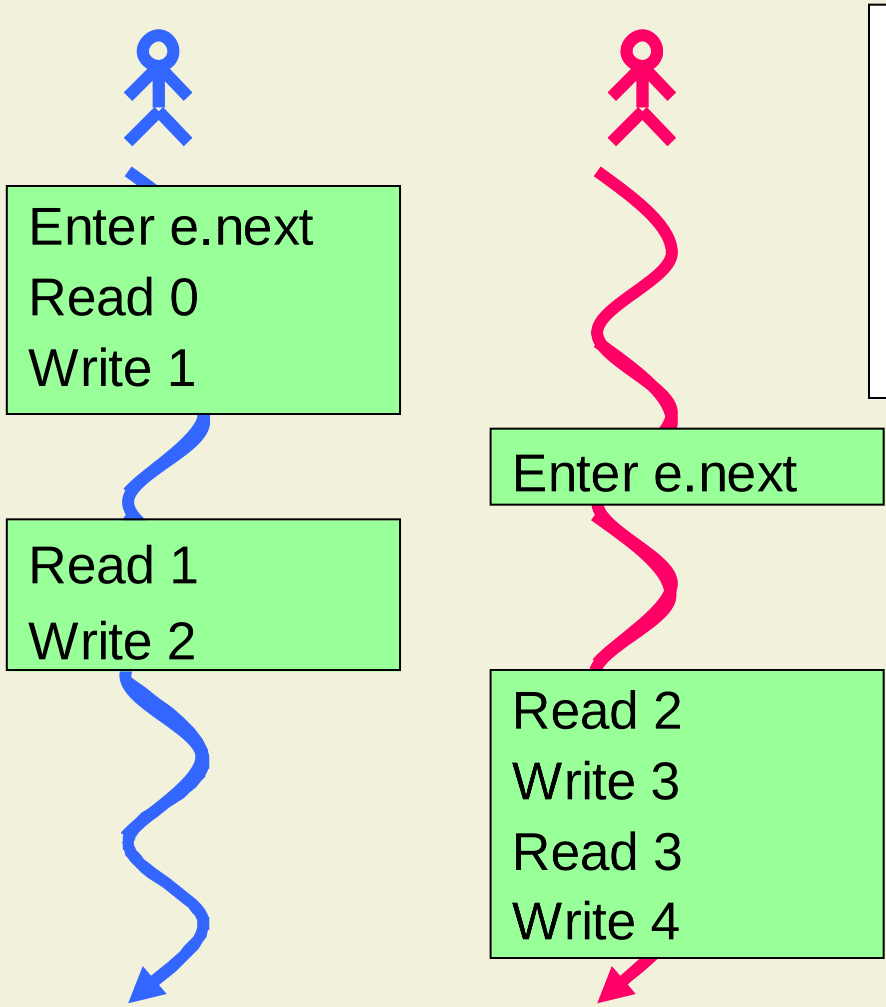
Preventing Data Races



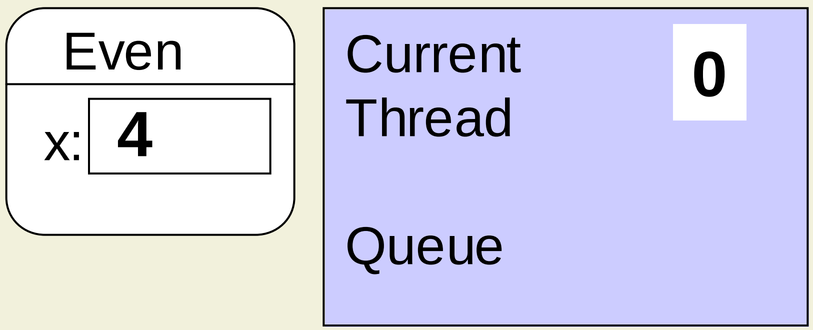

```
class Even {
  private int x;
  synchronized void next( )
      { x++; x++; }
}
```



Preventing Data Races

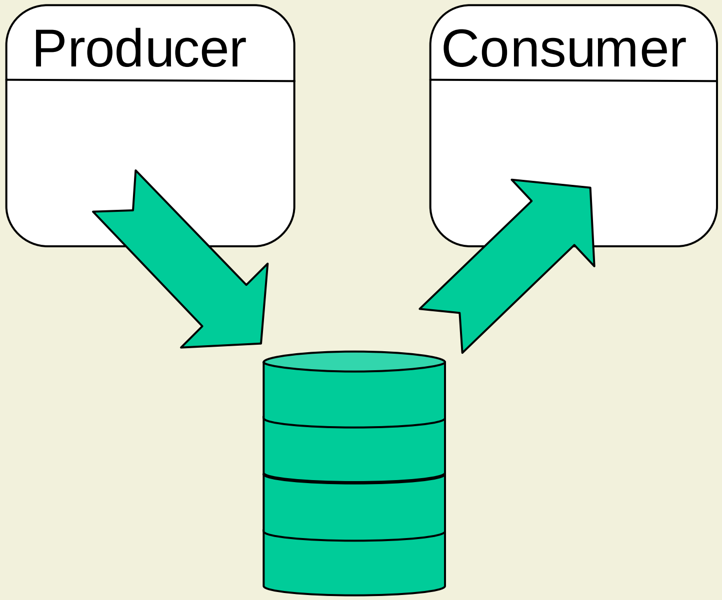



```
class Even {
  private int x;
  synchronized void next( )
      { x++; x++; }
}
```



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

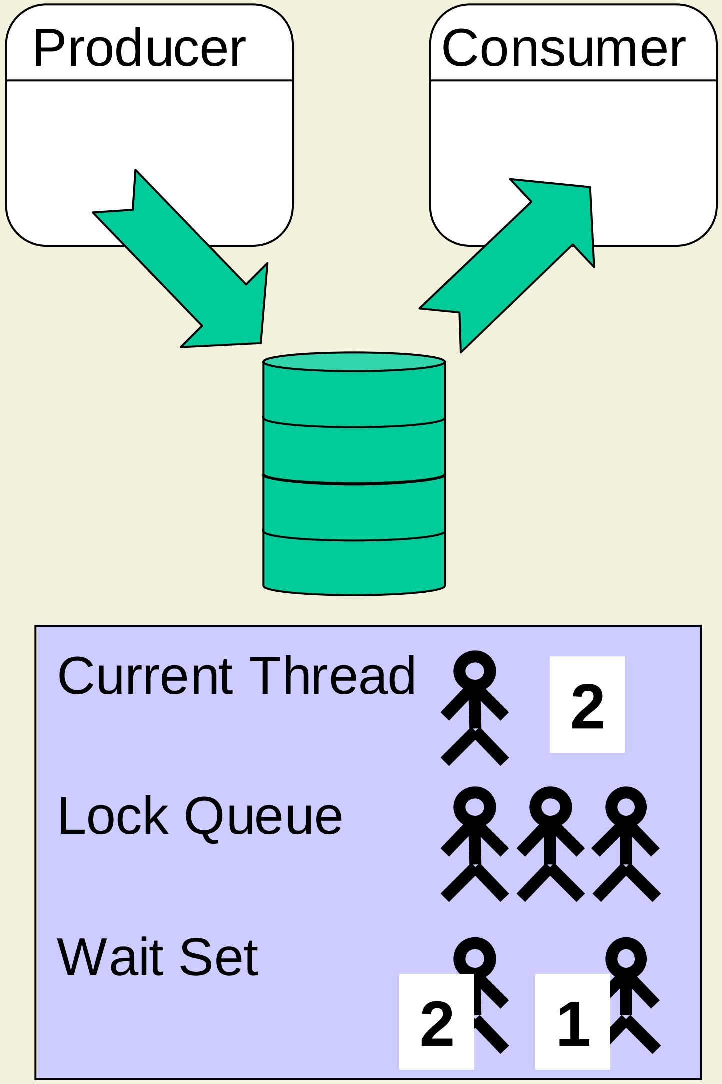
```
class Buffer {
  …
  synchronized boolean put( Prd p ) {
    if ( isFull( ) )        return false;
    …
    return true;
  }

 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



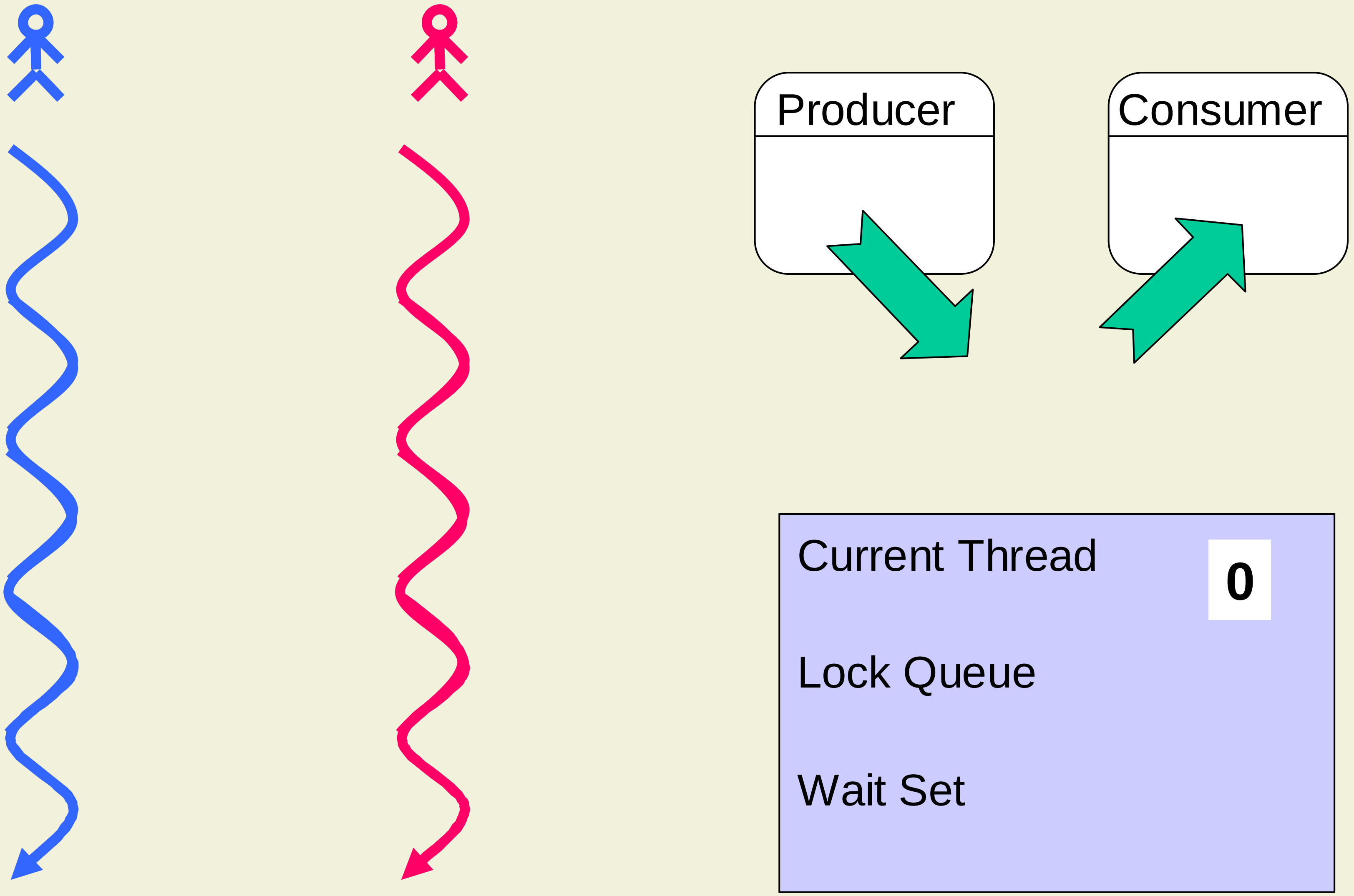
Producer-Consumer Example Revisited

```
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( ) );
  }
}
```

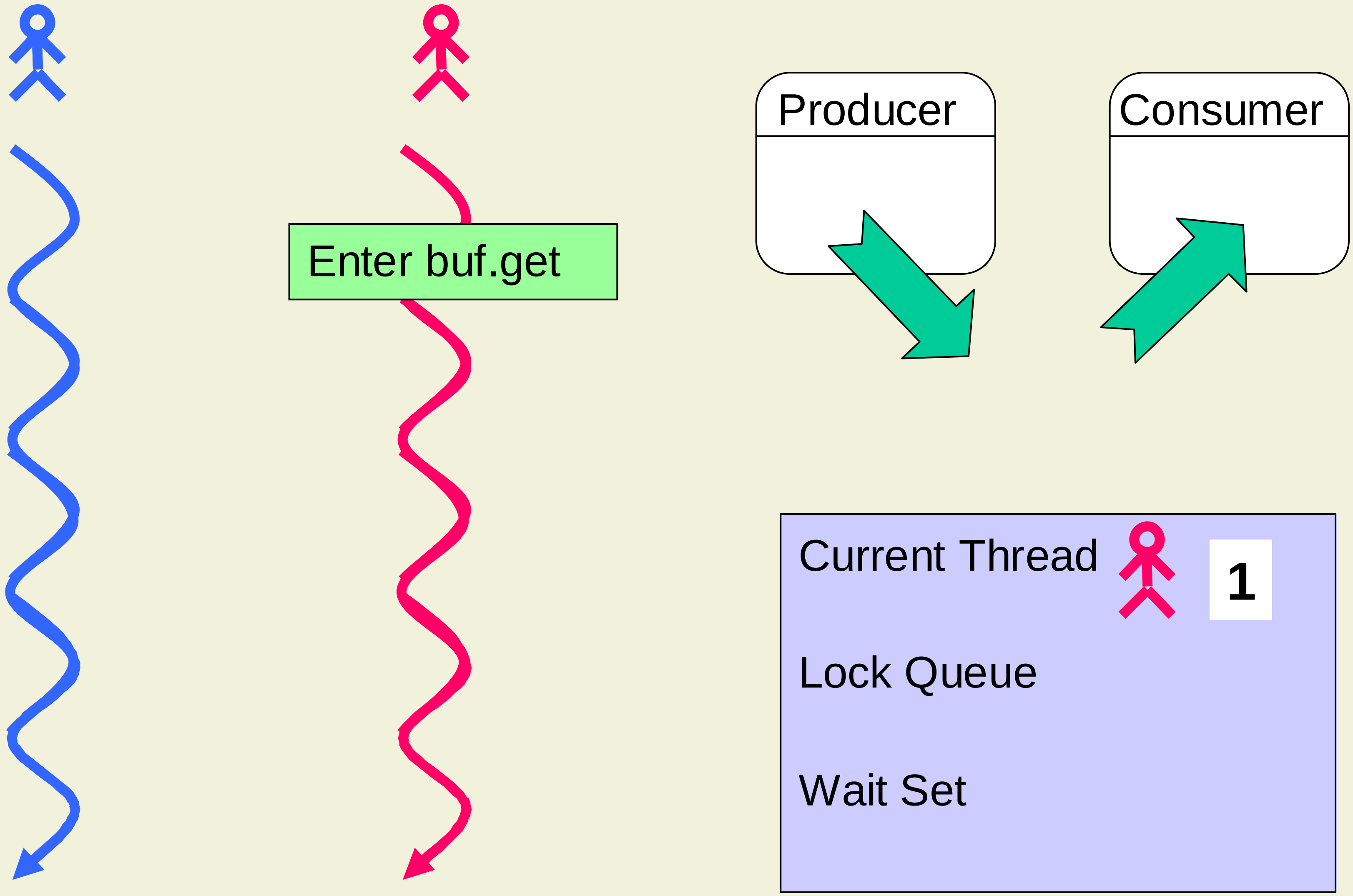
```
class Consumer extends Thread {
  Buffer buf;
  void run( ) {
    while ( true )
    buf.get( );
  }
}
```

Producer-Consumer Example Revisited

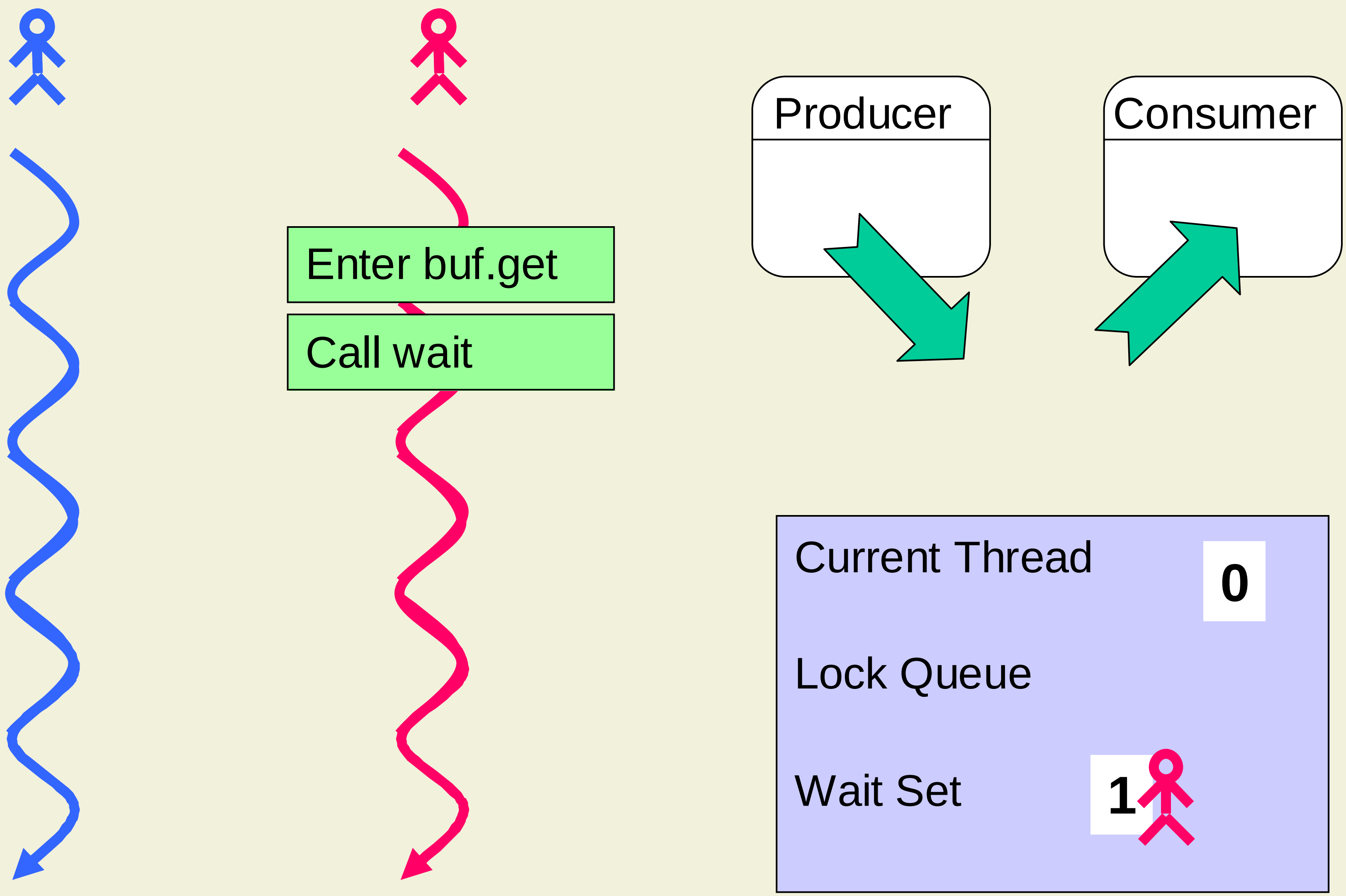


Current Thread	**0**
Lock Queue	
Wait Set	

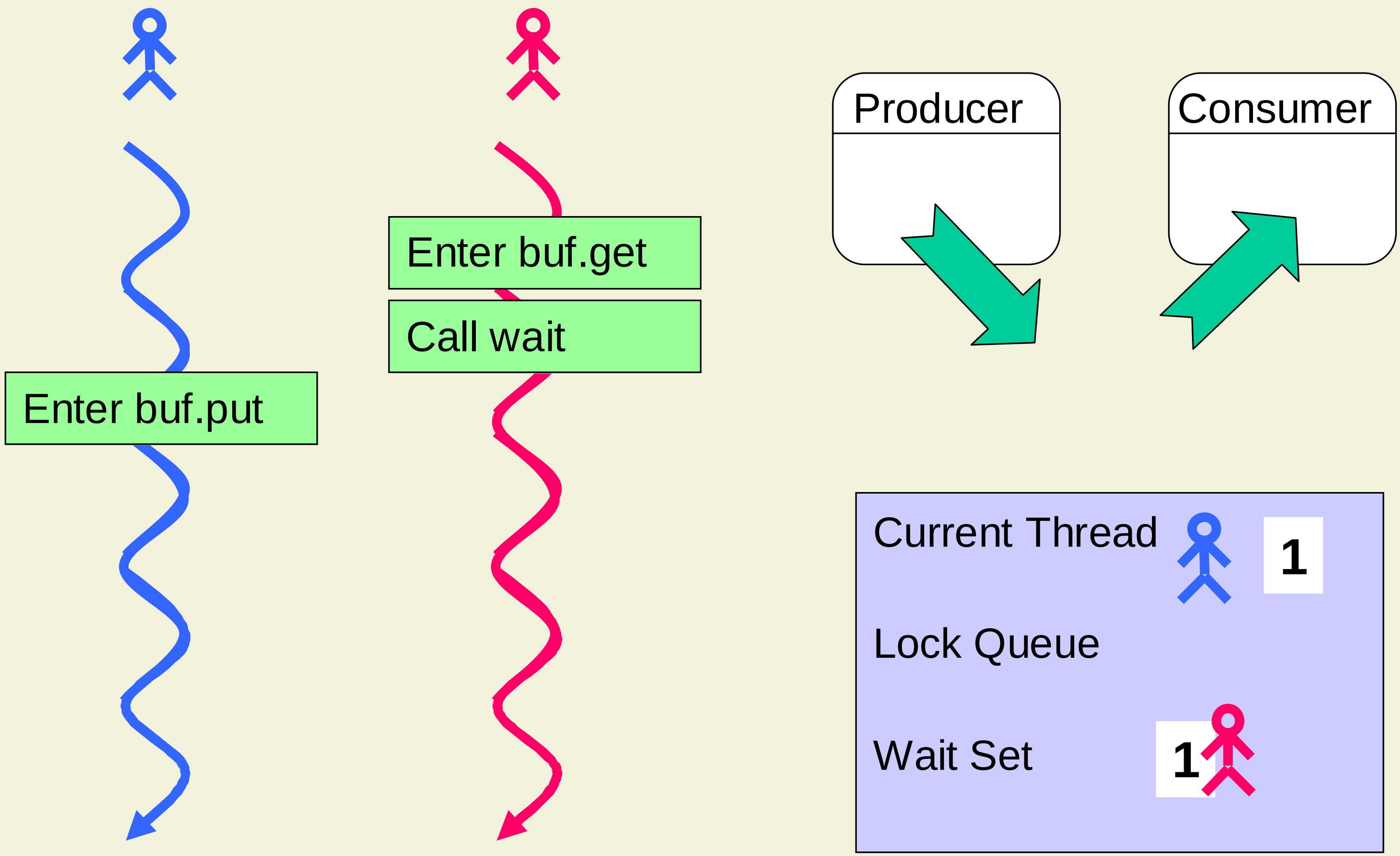
Producer-Consumer Example Revisited



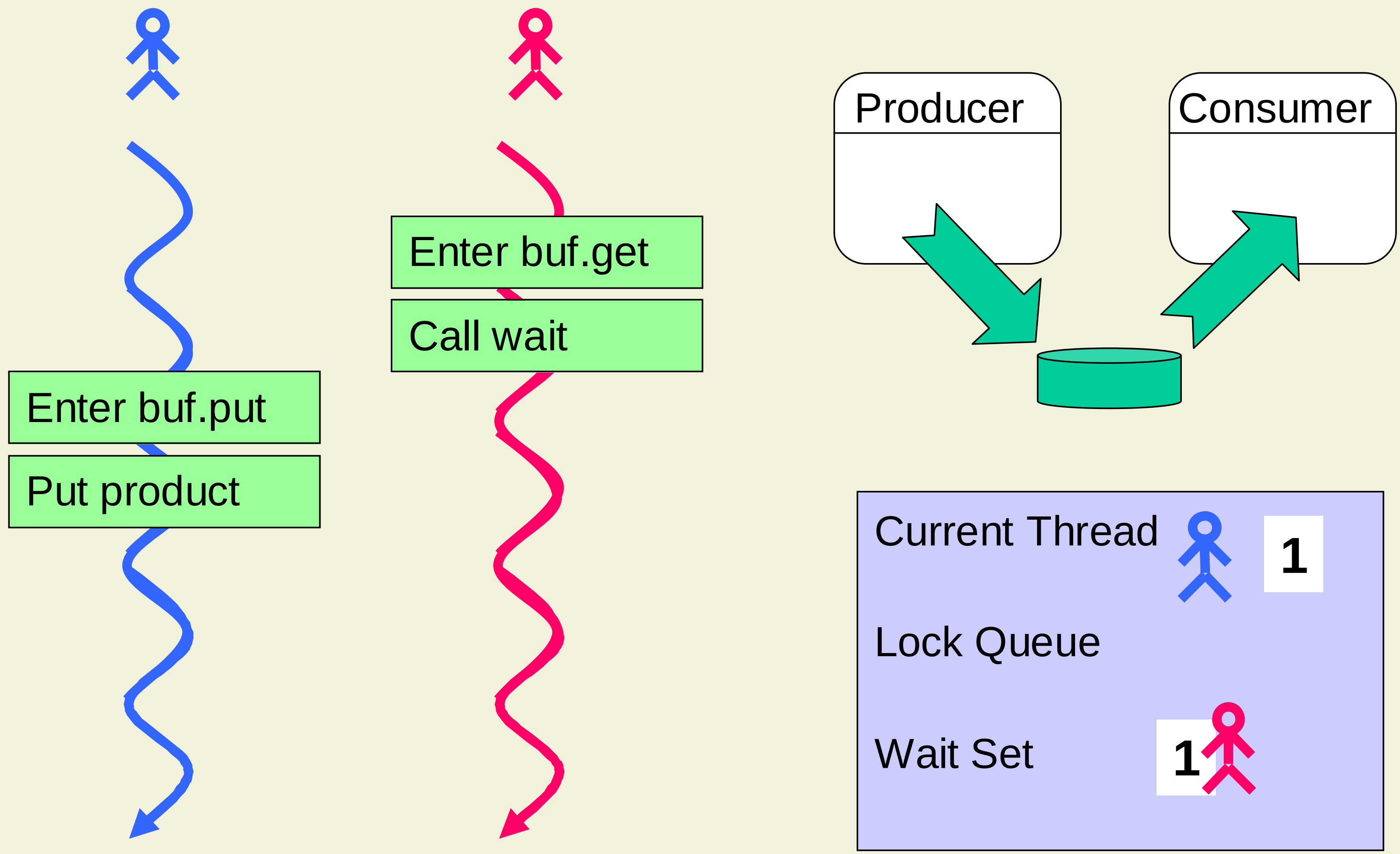
Producer-Consumer Example Revisited



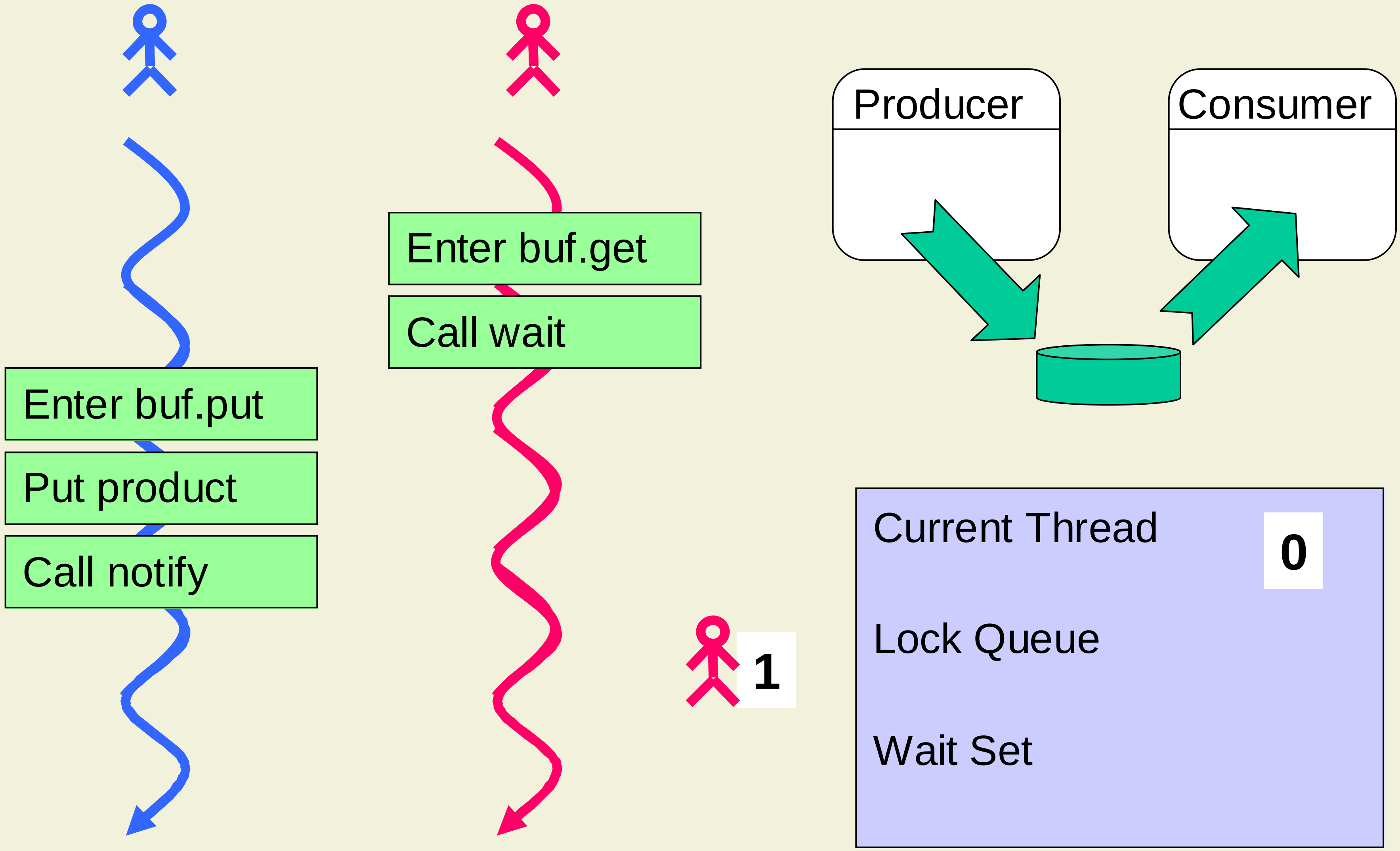
Producer-Consumer Example Revisited



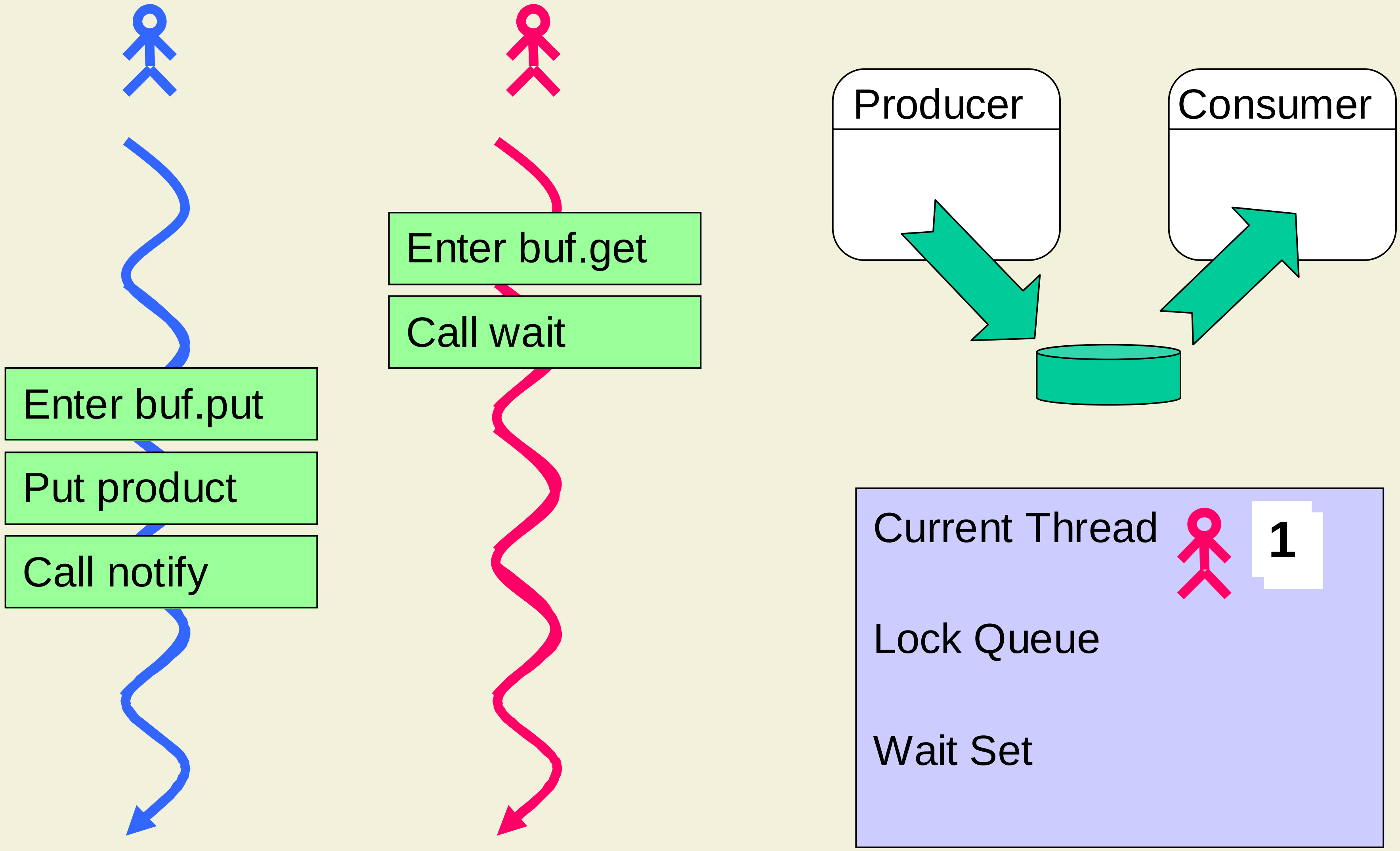
Producer-Consumer Example Revisited



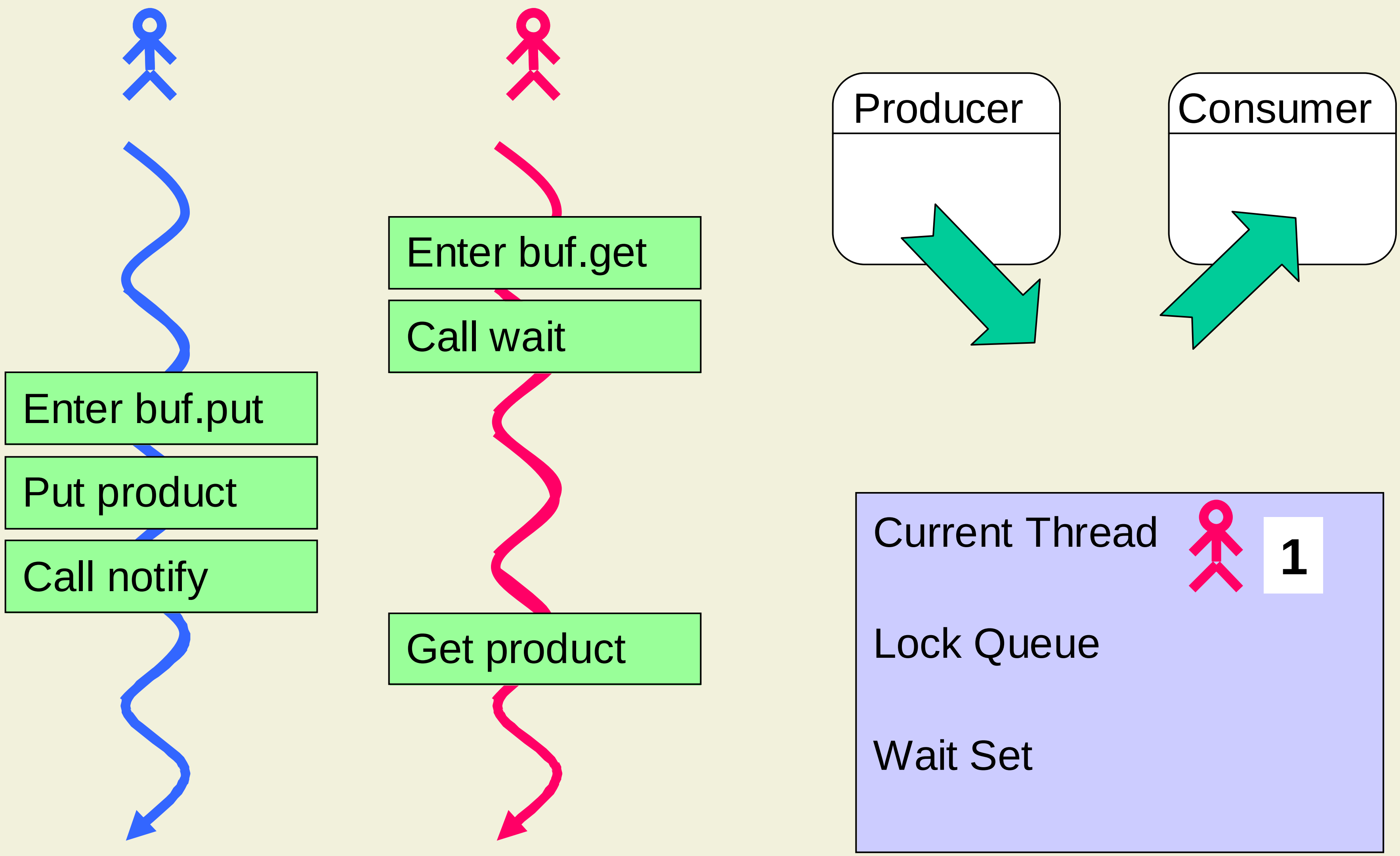
Producer-Consumer Example Revisited



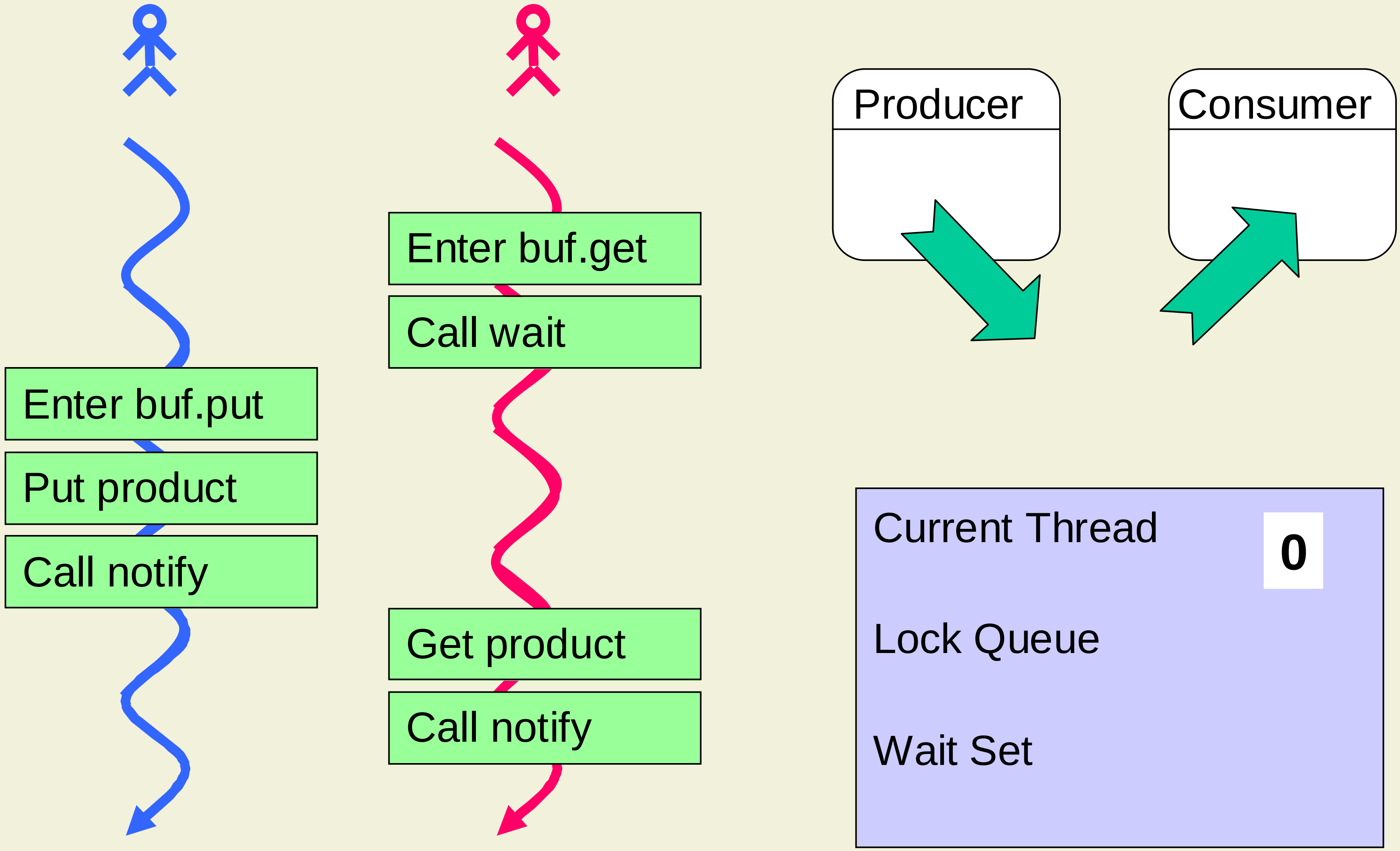
Producer-Consumer Example Revisited



Producer-Consumer Example Revisited



Producer-Consumer Example Revisited

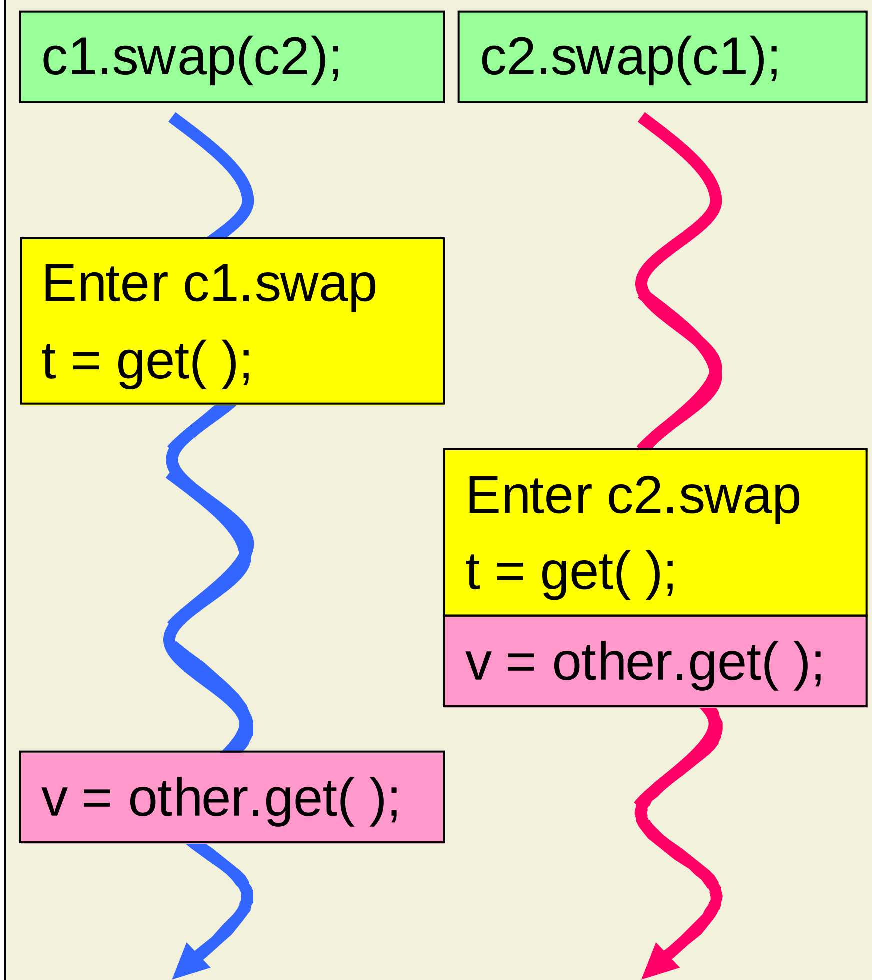


Safety and Liveness

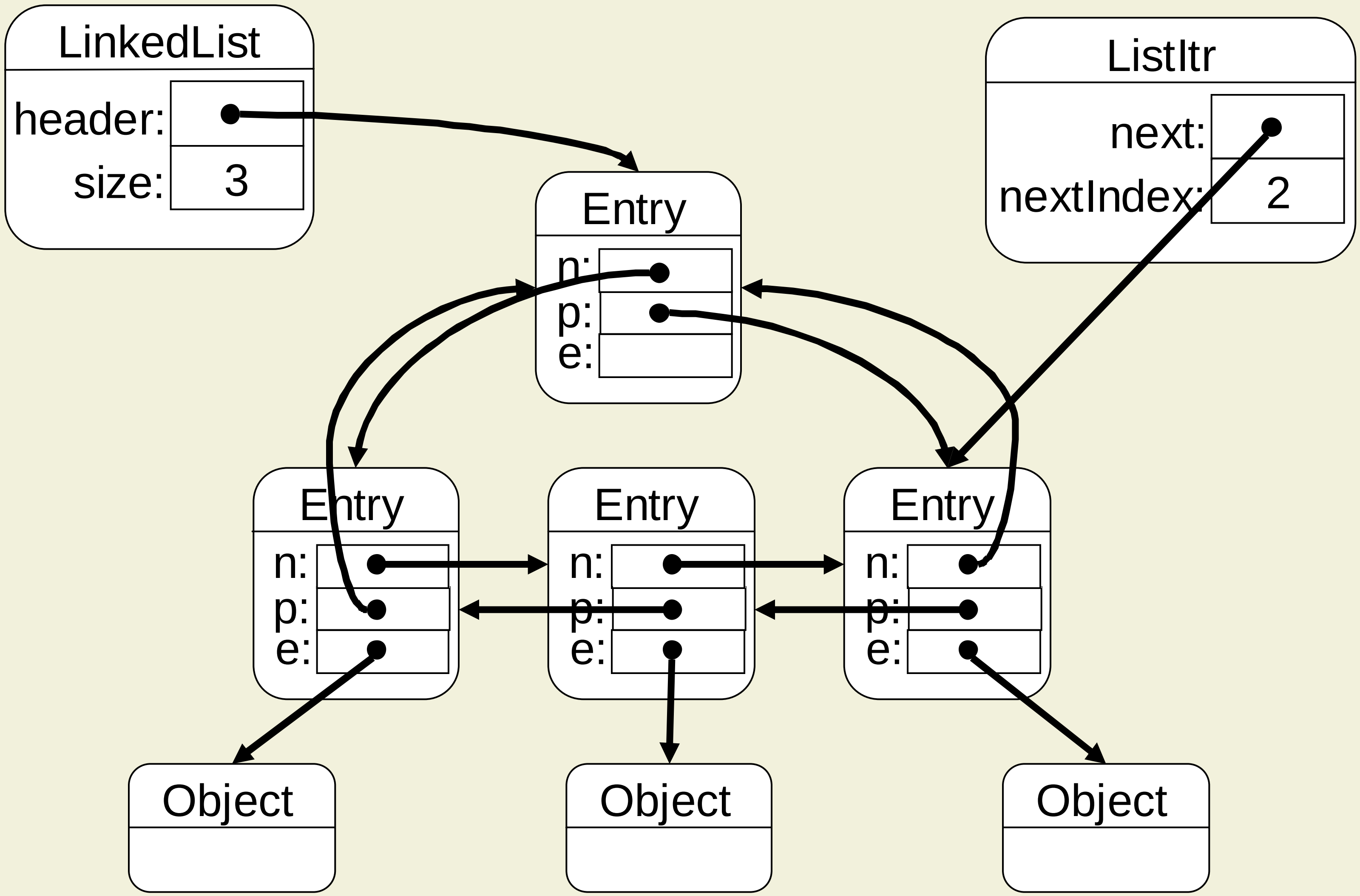
- Safety
  - **“Nothing bad ever happens”**
  - To perform method actions only when in consistent states
  - Achieved by mutual exclusion
- Liveness
  - **“Something eventually happens”**
  - Every called method should eventually execute
  - Avoiding deadlocks
  - Avoiding unfair scheduling (not guaranteed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

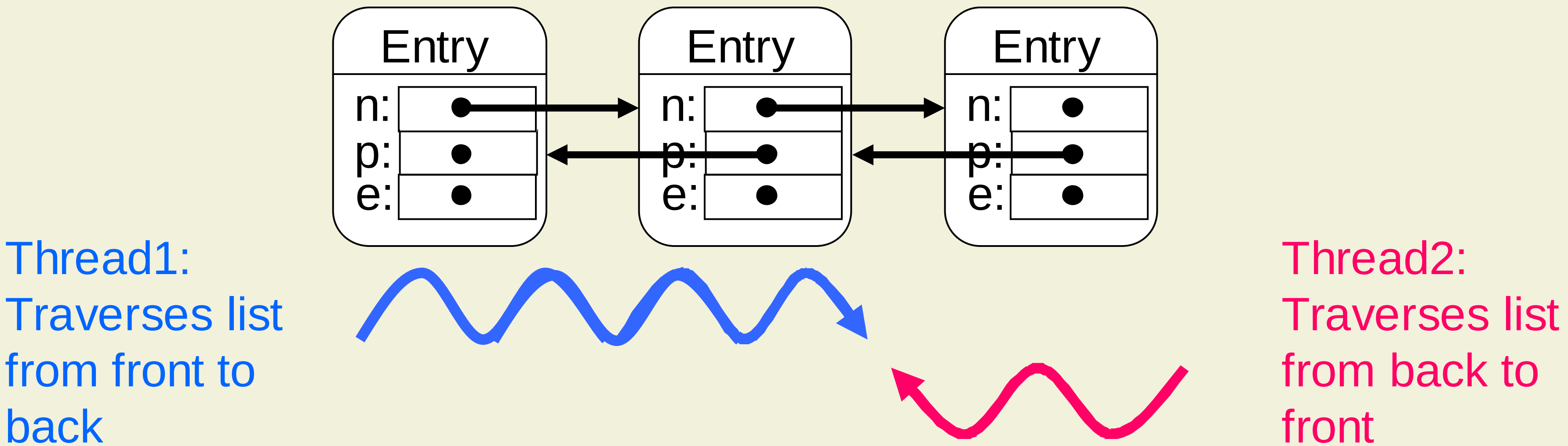


Synchronization with Object Structures



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



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 ) { … }
  }
}
```

Splitting Locks and Behavior

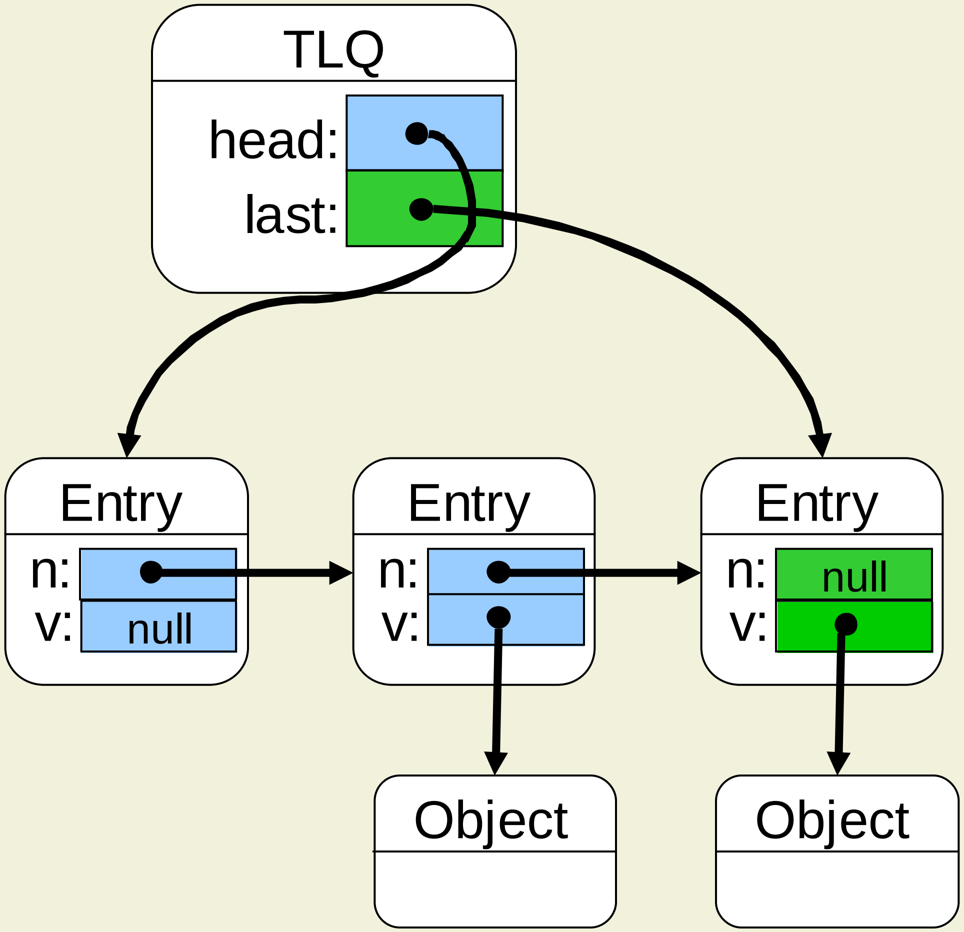
- Associate a helper object with an **independent subset of state and functionality**
- Delegate actions to helper via pass-through method
- grow and shift can execute simultaneously

```
class Shape {
  // size & location are independent
  int height = 0;
  int width = 0;
  synchronized void grow() {
     height++; width++;
  }

  Location l = new Location(0,0);
         // fully synchronized
  void shift() { l.moveBy( 1 1 ); }
         // Use l's synchronization
}
```

Concurrent Queue

- put works at the end of the list
- get works at the front of the list

- Operations can be synchronized on different locks



Concurrent Queue: Code

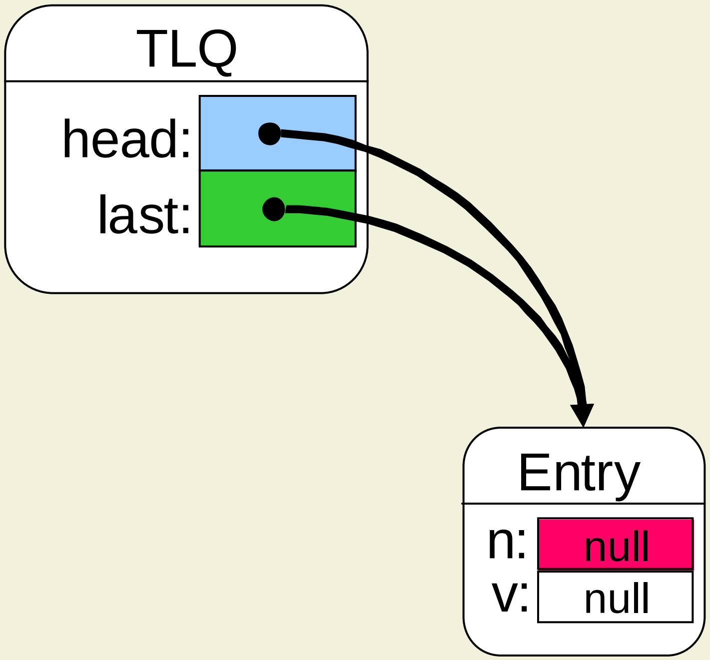
```
class Entry {
  Object value;
  Entry next = null;
  Entry( Object x ) { value = x; }
}
```

```
class TwoLockQueue {
  private Entry head =
              new Entry( null );
  private Entry last = head;
  private Object lastLock =
              new Object( );
```

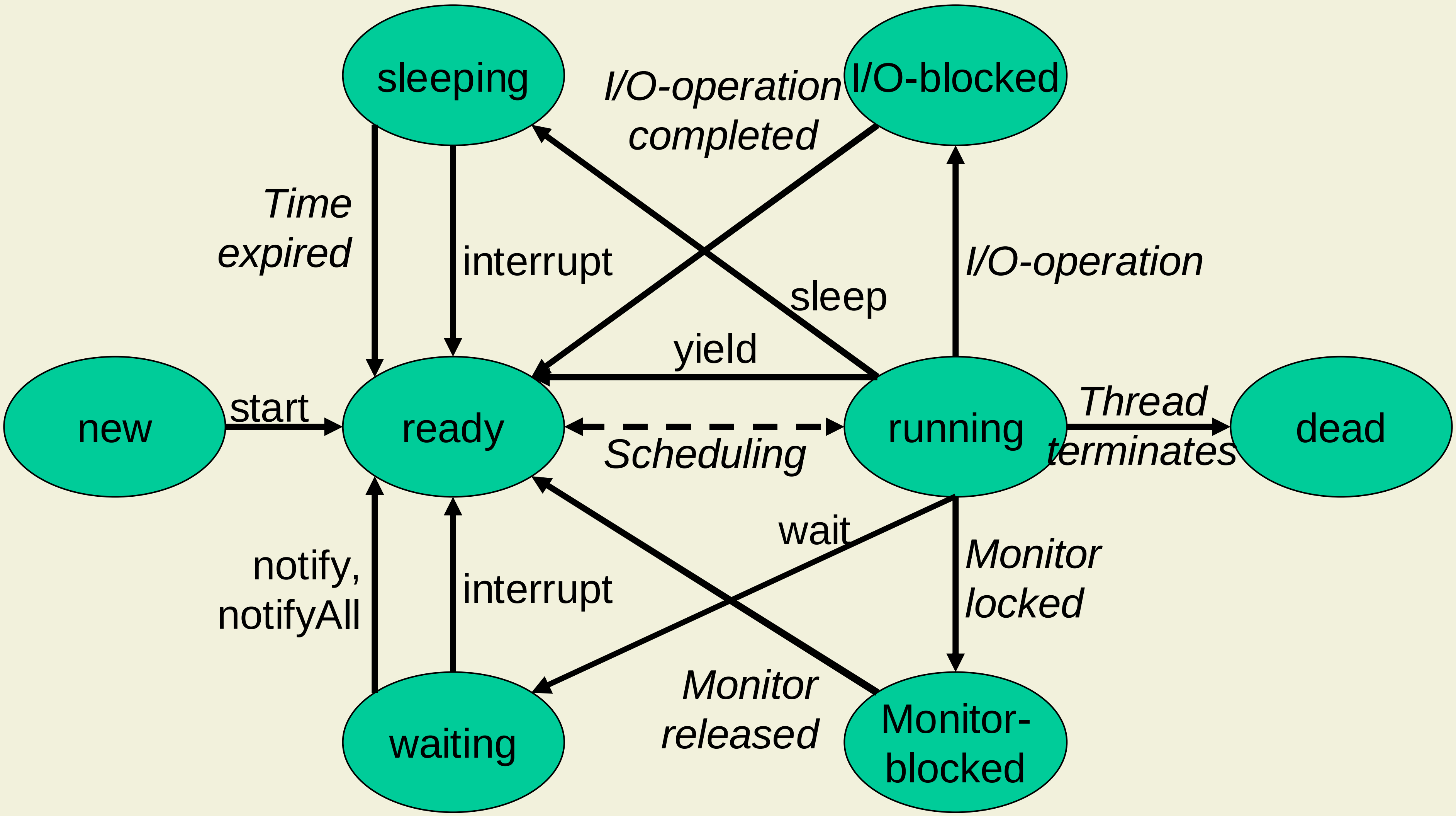
```
  void put( Object x ) {
    synchronized ( lastLock )
      { last = last.next = new Entry(x); }
  }
  synchronized Object get( ) {
    Object x = null;
    Entry first = head.next;
    if (first != null) {
      x = first.value; first.value = null;
      head = first;
    }
    return x;
} }
```

Concurrent Queue: Discussion

- put and get can run concurrently
- Java atomicity guarantees at only potential contention point
- Multiple puts and multiple gets disallowed



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

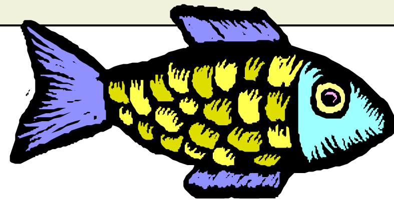
8. Concurrency

[This part is based on a guest lecture by Emil Sekerinski, McMaster University]

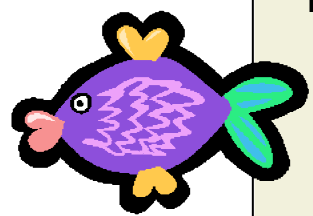
Overview

- Objects are a natural “unit” of concurrency
    - Objects can evolve independently and thus concurrently
    - Method calls allow for communication and synchronization
    - Creating an object potentially initiates a concurrent activity
- Model for active (autonomous) objects
    - Autonomous activity expressed by actions
    - Synchronization expressed by guards
- No threads

Actions: Fish Screen Saver



```
class Fish
  attr x, y: integer
  attr up, right: boolean
  initialization x, y, up, right := 0, 0, true, true
  action moveUp
    when y < H and up do y := y + 1
  action moveDown
    when y > 0 and ¬up do y := y – 1
  …
  action bounceUp
    when y = H and up do up := false
  action bounceDown
    when y = 0 and ¬up do up := true
end
```



```
var f: Fish ;
for I := 1 to 10 do
  f := new Fish
```

- Actions are executed autonomously
- An action is executed only if its guard is true

Guarded Methods: Bounded Buffer

```
class BoundedBuffer
  attr b: array of Object
  attr in, out, n, max: integer
  initialization (m: integer)  in, out, n, max, b := 0, 0, 0, m, new Object[ m ]
  method put (x: Object)
    when n < max do in, b[ in ], n := ( in + 1 ) mod max, x, n + 1
  method get: Object
    when n > 0 do out, result, n := ( out + 1 ) mod max, b[ out ], n – 1
end
```

- Filtering from buffer in into out

```
x := in.get; if f( x ) then out.put( x )
```

- Execution may block at calls to guarded methods

Semaphore

```
class Semaphore
  attr n: integer
  initialization (m: integer)
    n := m
  method acquire
    when n > 0 do n := n – 1
  method release
    n := n + 1
end
```

- A semaphore s that allows m concurrent users of a resource

```
var s: Semaphore; s :=
new Semaphore (m)
```

- A user requiring semaphores s and t for a critical section

```
s.acquire; t.acquire;
… critical section …;
s.release; t.release
```

Eager versus Lazy Computation

```
class Doubler
  attr x: integer
  method store (u: integer)
    x := 2 * u
  method retrieve: integer
    result := x
end
```

```
class LazyDoubler
  attr x: integer
  attr d: boolean
  initialization d := true
  method store (u: integer)
    x, d := u, false
  method retrieve: integer
    if ¬d then x, d := 2 * x, true;
    result := x
end
```

- Objects of class LazyDoubler can be used wherever objects of class Doubler are expected

Introducing Concurrency

```
class LazyDoubler
  attr x: integer
  attr d: boolean
  initialization d := true
  method store (u: integer)
    x, d := u, false
  method retrieve: integer
    if ¬d then x, d := 2 * x, true;
    result := x
end
```

```
class DelayedDoubler
  attr x: integer
  attr d: boolean
  initialization d := true
  method store (u: integer)
    x, d := u, false
  method retrieve: integer
    when d do result := x
  action double
    when ¬d do x, d := 2 * x, true
end
```

- Objects of class DelayedDoubler can be used wherever objects of class Doubler are expected

Introducing Concurrency in Subclasses

```
class Doubler
  attr x: integer
  method store (u: integer)
    x := 2 * u
  method retrieve: integer
    result := x
end
```

```
class DelayedDoubler
                inherit Doubler
  attr d: boolean
  initialization d := true
  method store (u: integer)
    x, d := u, false
  method retrieve: integer
    when d do result := x
  action double
    when ¬d do x, d := 2 * x, true
end
```


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Concurrent Priority Queue: Code

```
class PriorityQueue
  attr m, p: integer
  attr l: PriorityQueue
  attr a: boolean
  initialization l, a := nil, false
  method add (e: integer)
    when ¬a do
      if l = nil then
        begin
          m := e; l := new PriorityQueue
        end
      else
        begin p := e ; a := true end
  action doAdd
    when a do
      begin
        if m < p then l.add (p)
        else
          begin
            l.add (m) ; m := p
          end;
        a := false
  method empty: boolean
    result := l = nil
end
```

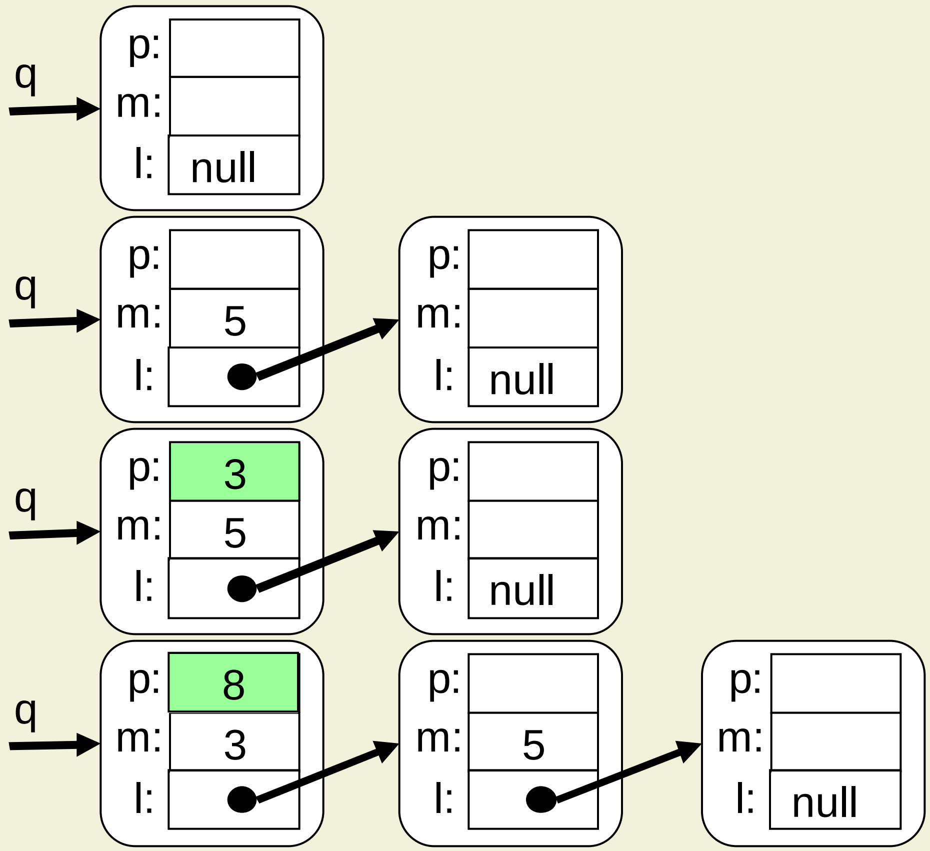
Concurrent Priority Queue

q := **new** PQ

q.add(5)

q.add(3)

q.add(8)



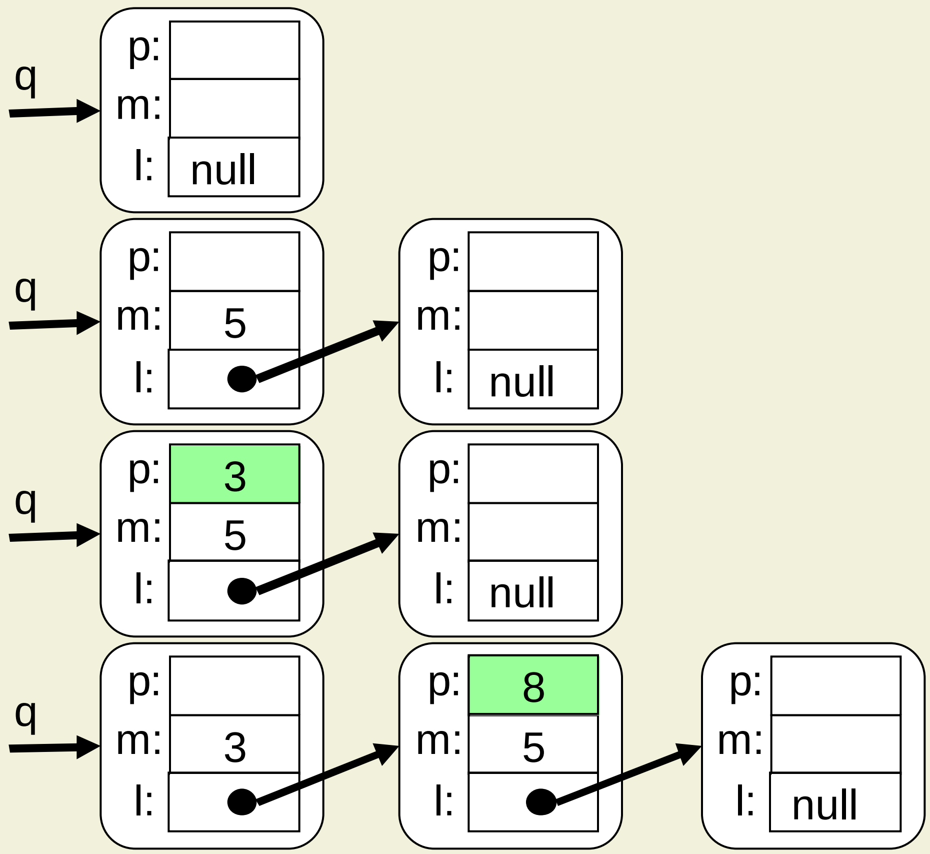
Concurrent Priority Queue

q := **new** PQ

q.add(5)

q.add(3)

q.add(8)



Concurrent Priority Queue

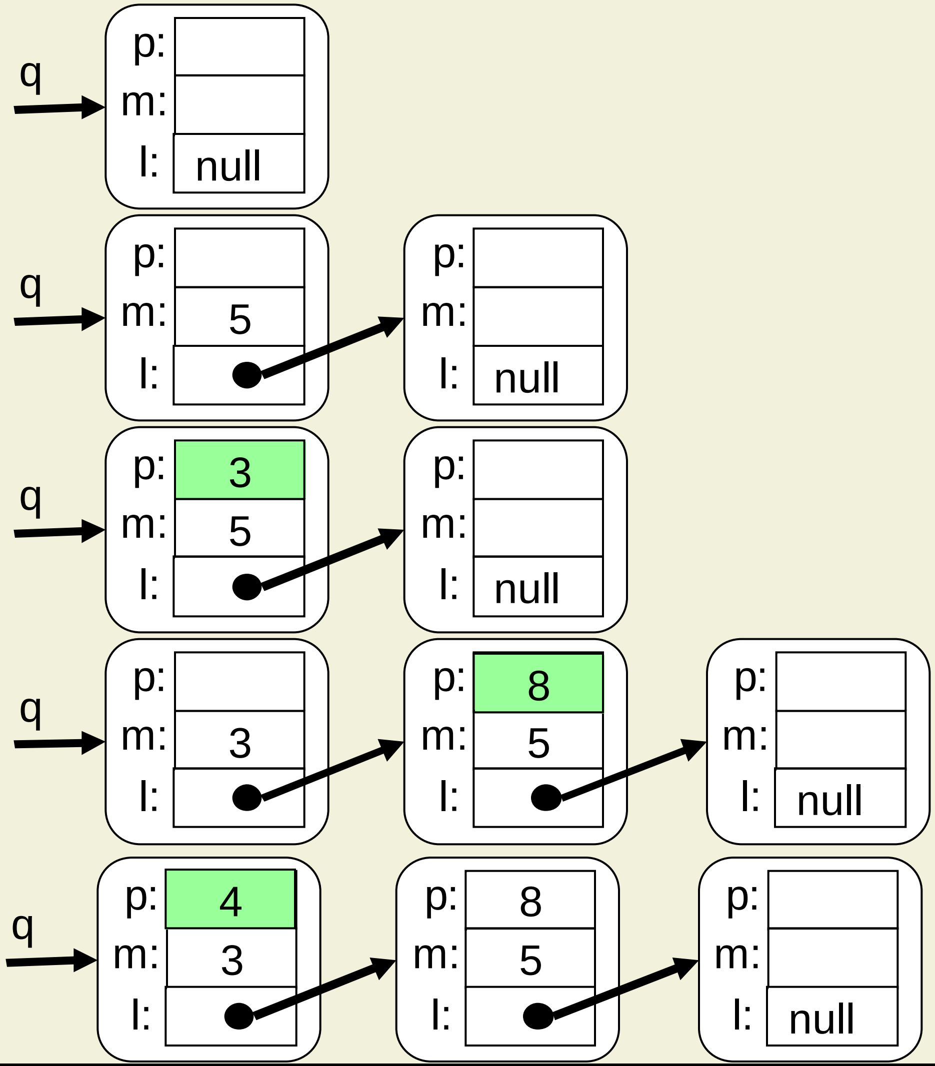
q := **new** PQ

q.add(5)

q.add(3)

q.add(8)

q.add(4)



Concurrent Priority Queue

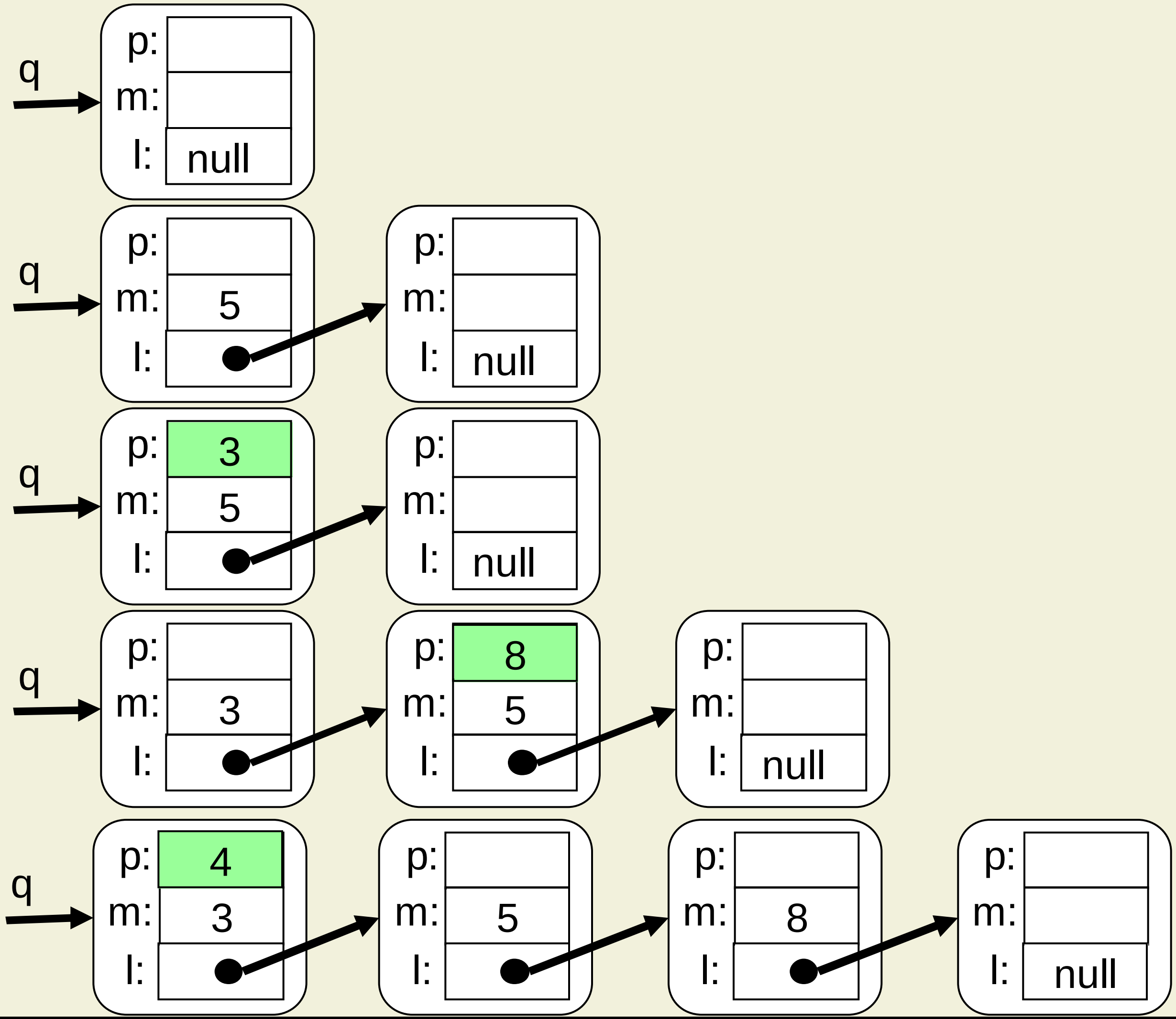
q := **new** PQ

q.add(5)

q.add(3)

q.add(8)

q.add(4)



Concurrent Priority Queue

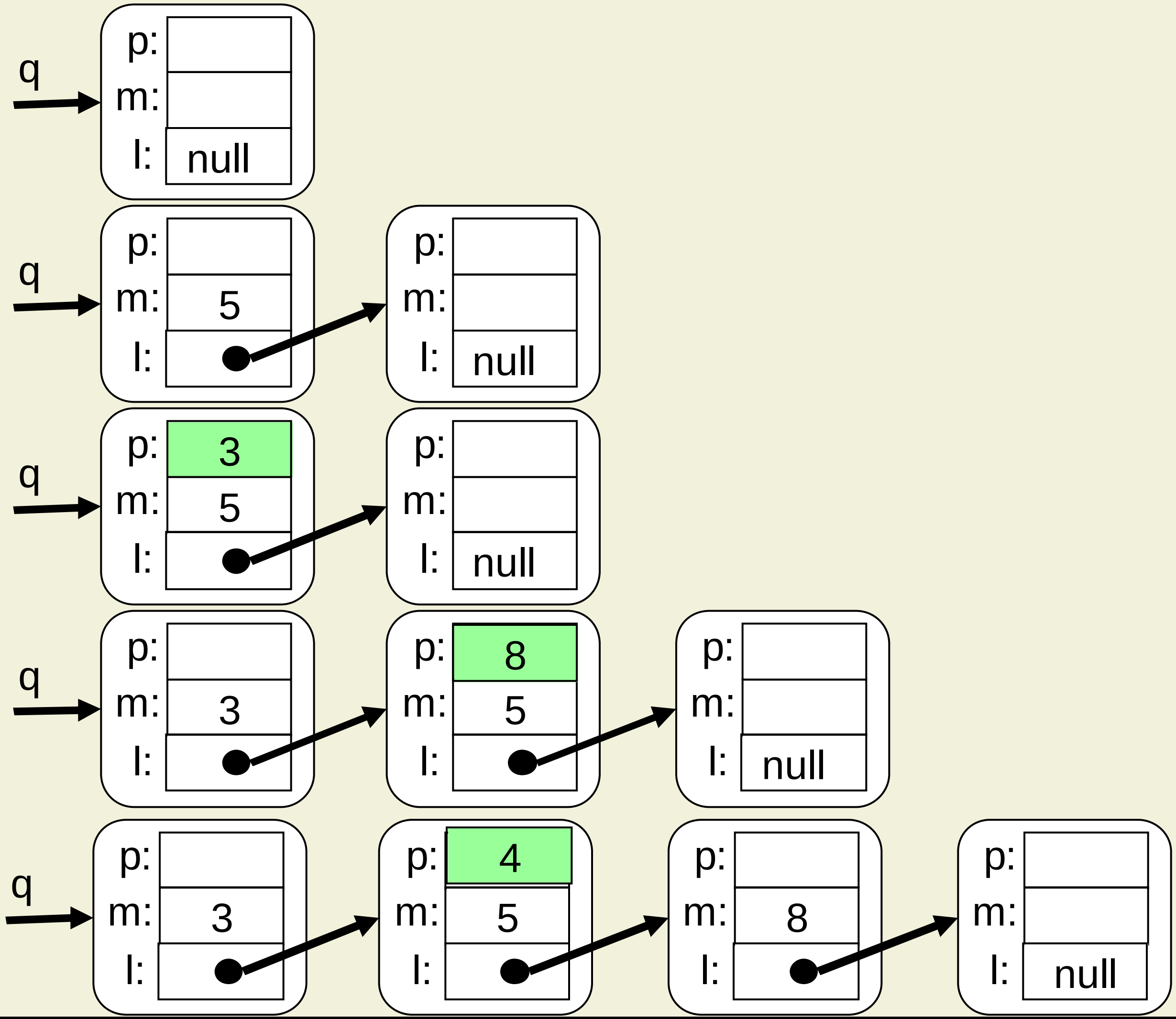
q := **new** PQ

q.add(5)

q.add(3)

q.add(8)

q.add(4)



Dining Philosophers

```
class Fork
  attr available: boolean
  initialization available := true
  method pickUp when available do available := false
  method putDown available := true
end
```

```
class Philosopher
  attr state: (thinking, hungry, eating, full)
  attr left, right: Fork
  initialization (l, r : Fork) state, left, right := thinking, l, r
  action gettingHungry when state = thinking do
    begin state := hungry; left.pickUp ; right.pickUp; state := eating end
  action gettingFull when state = eating do
    begin state := full; left.putDown ; right. putDown; state := thinking end
end
```

Dining Philosophers: Main Program

```
var fork := new Fork [ 5 ];
var philosopher := new Philosopher [ 5 ];
for i := 0 to 4 do
  fork [ i ] := new Fork ;
for j := 0 to 4 do
  philosopher [ j ] := new Philosopher ( fork [ j ], fork [ (j + 1) mod 5 ] )
```

Fairness through Strong Semaphores

```
class WeakBinarySemaphore
  attr a: boolean
  initialization a := true
  method acquire
    when a do a := false
  method release
    a := true
end
```

- If continuously several users try to acquire a weak semaphore, some may be delayed indefinitely

```
class StrongBinarySemaphore
  attr a: boolean
  attr q: seq of Object
  initialization a, q := true, <>
  method acquire( u: Object )
    begin q := q ° <u>;
    when a and u = head( q ) do
      a, q := false, tail( q )
  method release a := true
end
```

- The strong semaphore ensures a first-in first-out policy

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Summary of Active Objects

- Language extensions
  - **No** construct for **threads**
  - Classes: attributes, methods, **actions**
  - **Guards** for synchronization: **when** b **do** S = **await** b S

- Goal: Bring the practice of concurrent object-oriented programming as close as possible to a simple model with a sound theory