

Konzepte objektorientierter Programmierung

Prof. Dr. Peter Müller

Werner Dietl

Software Component Technology

Exercises 10: Concurrency

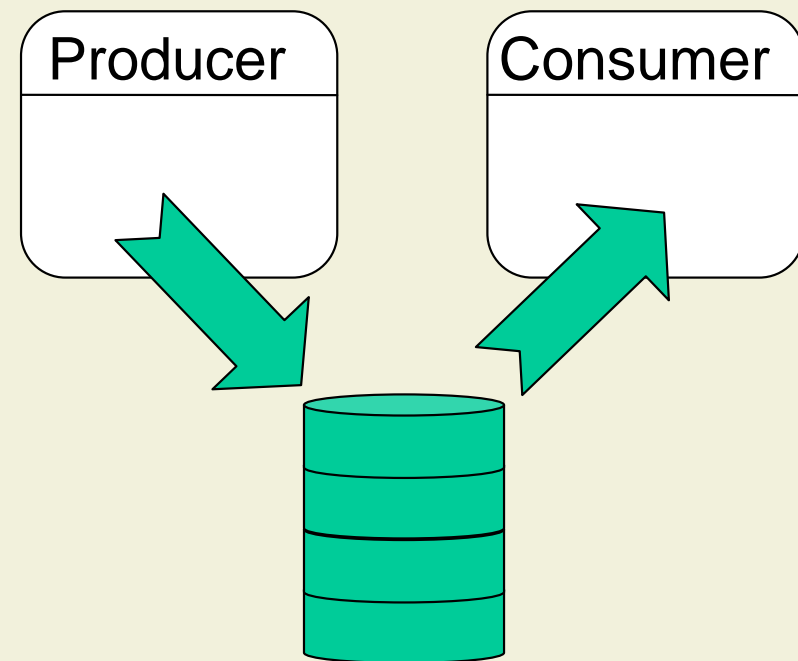
Wintersemester 05/06

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

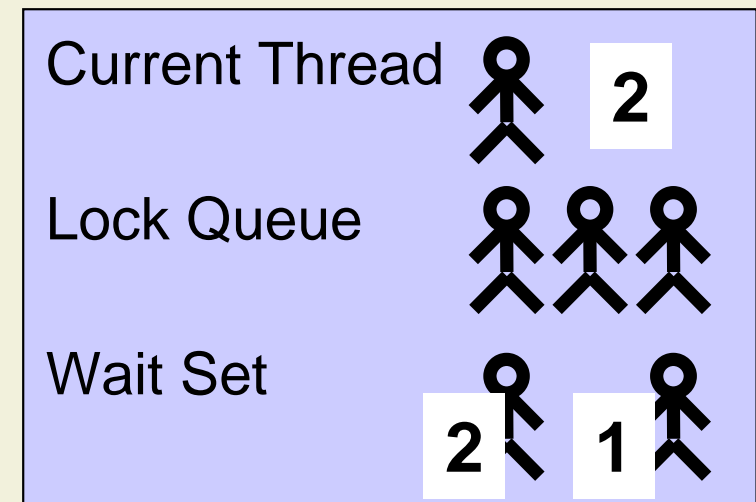
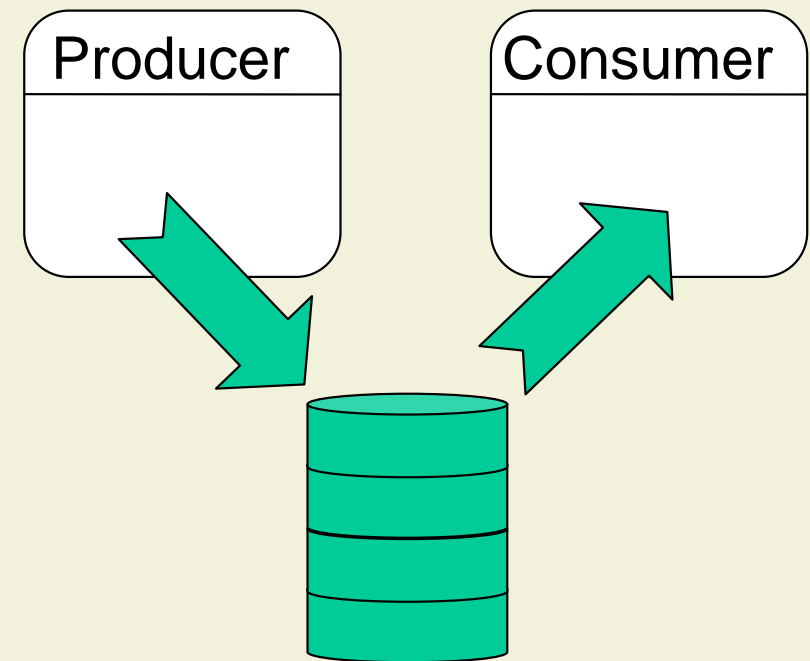
Cooperating Threads

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



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



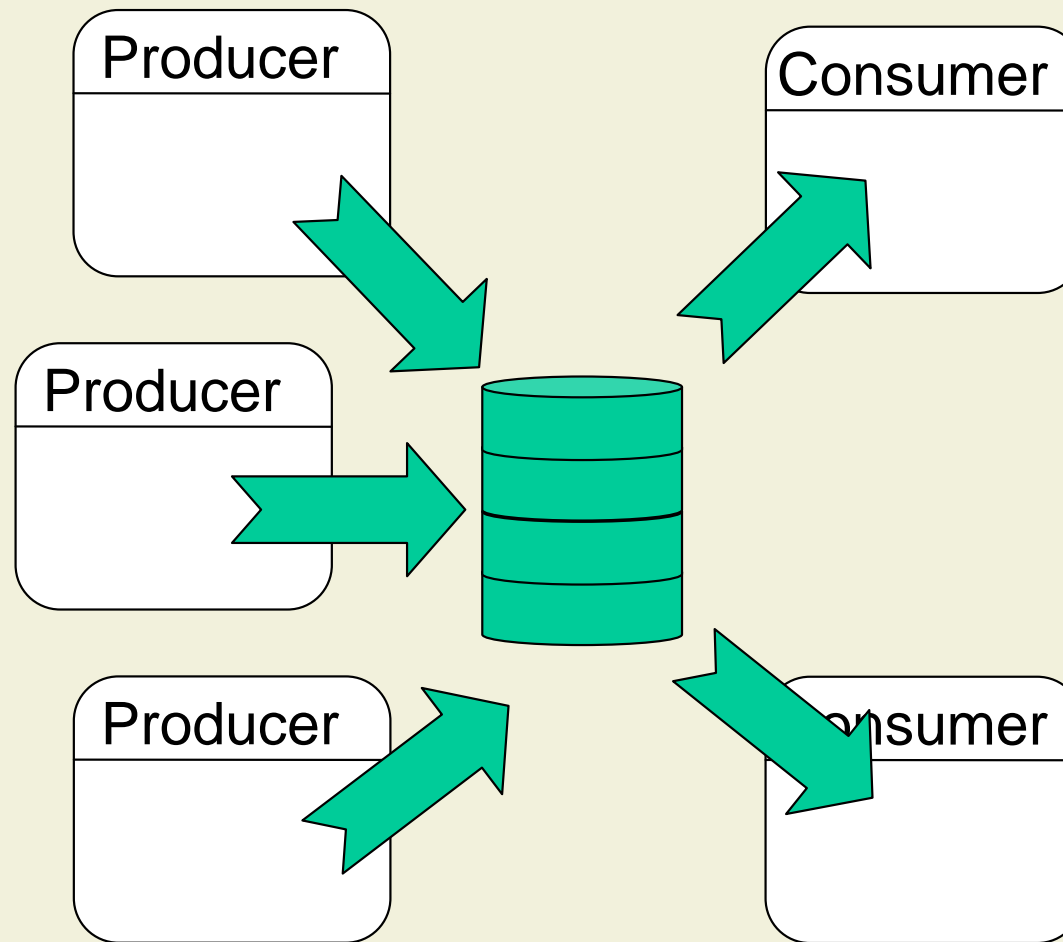
Producer-Consumer Example Revisited

```
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( );  
    }  
}
```

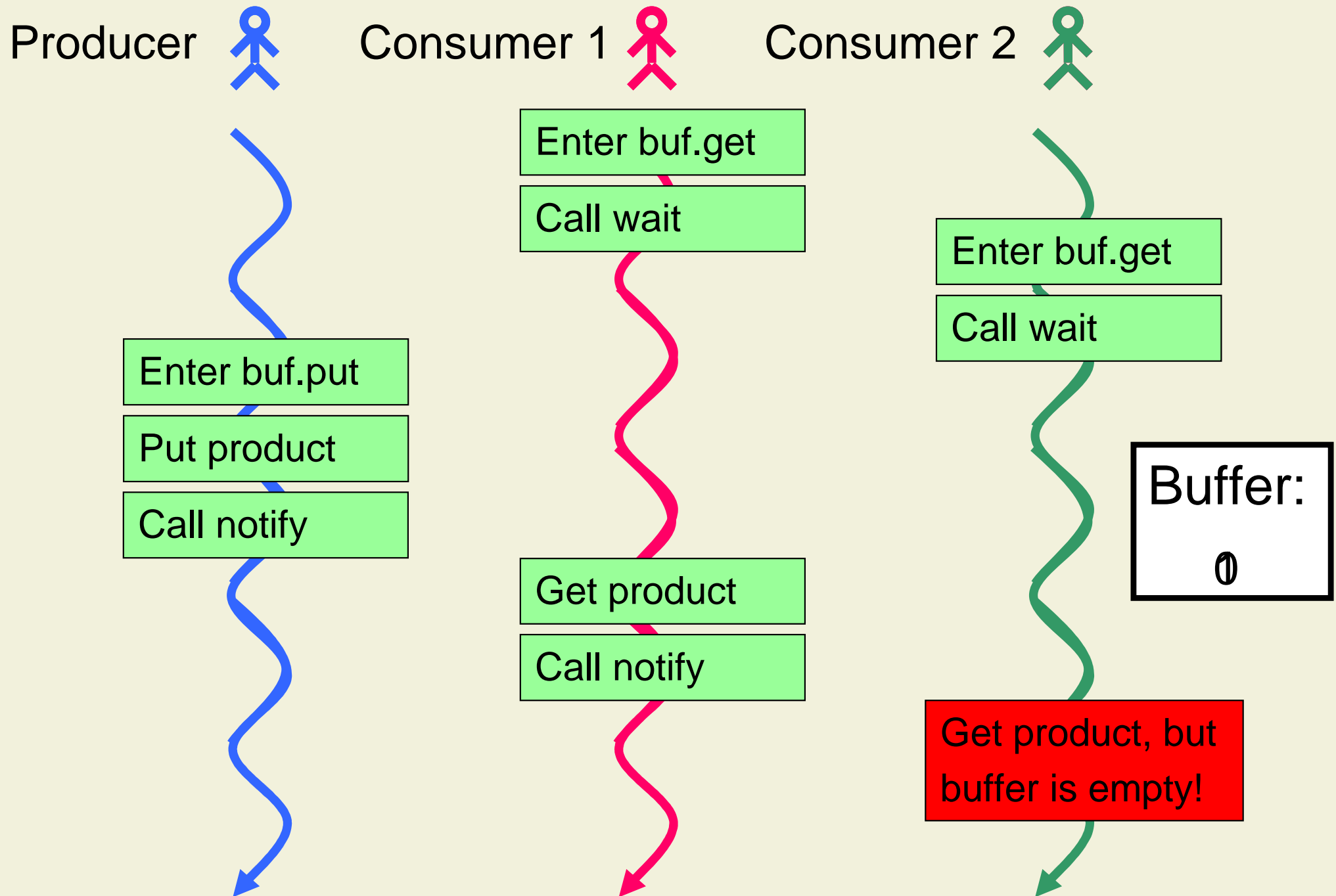
Multiple Producers & Consumers



Lazy Attempt: Reuse the Program

```
public synchronized void put( Prd p ) {  
    if( isFull() ) {  
        wait();  
    }  
    ...  
    notify();  
}
```

**2 Problems if there are
more than 2 threads!**

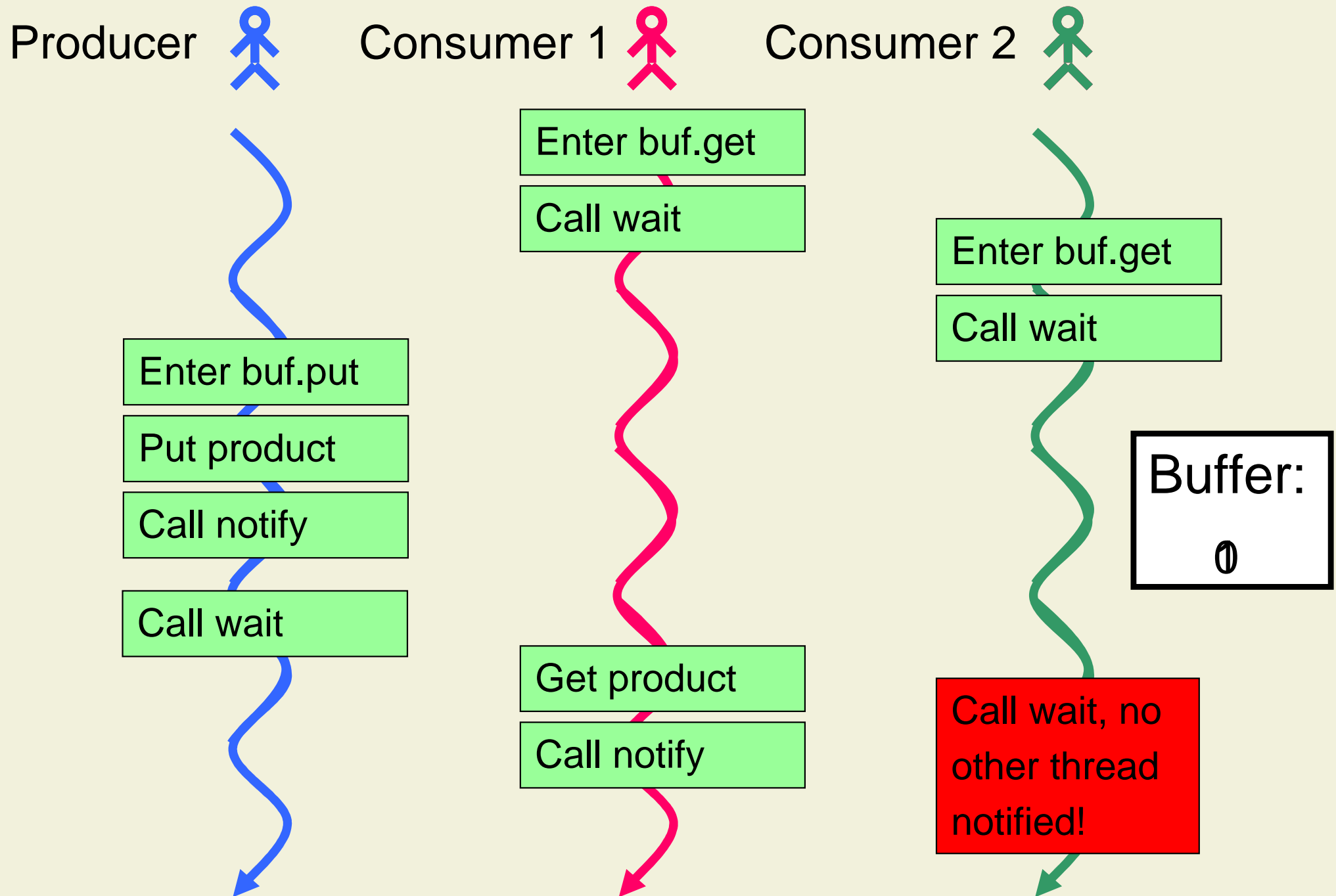


Lazy Attempt: Reuse the Program

```
public synchronized void put( Prd p ) {  
    if( isFull() ) {  
        wait();  
    }  
    ...  
    notify();  
}
```


First Step

```
public synchronized void put( Prd p ) {  
    while( isFull() ) {  
        wait();  
    }  
    ...  
    notify();  
}
```



Solution

```
public synchronized void put( Prd p ) {  
    while( isFull() ) {  
        wait();  
    }  
    ...  
    notifyAll();  
}
```

Problem with notify

If there are multiple producers and consumers:

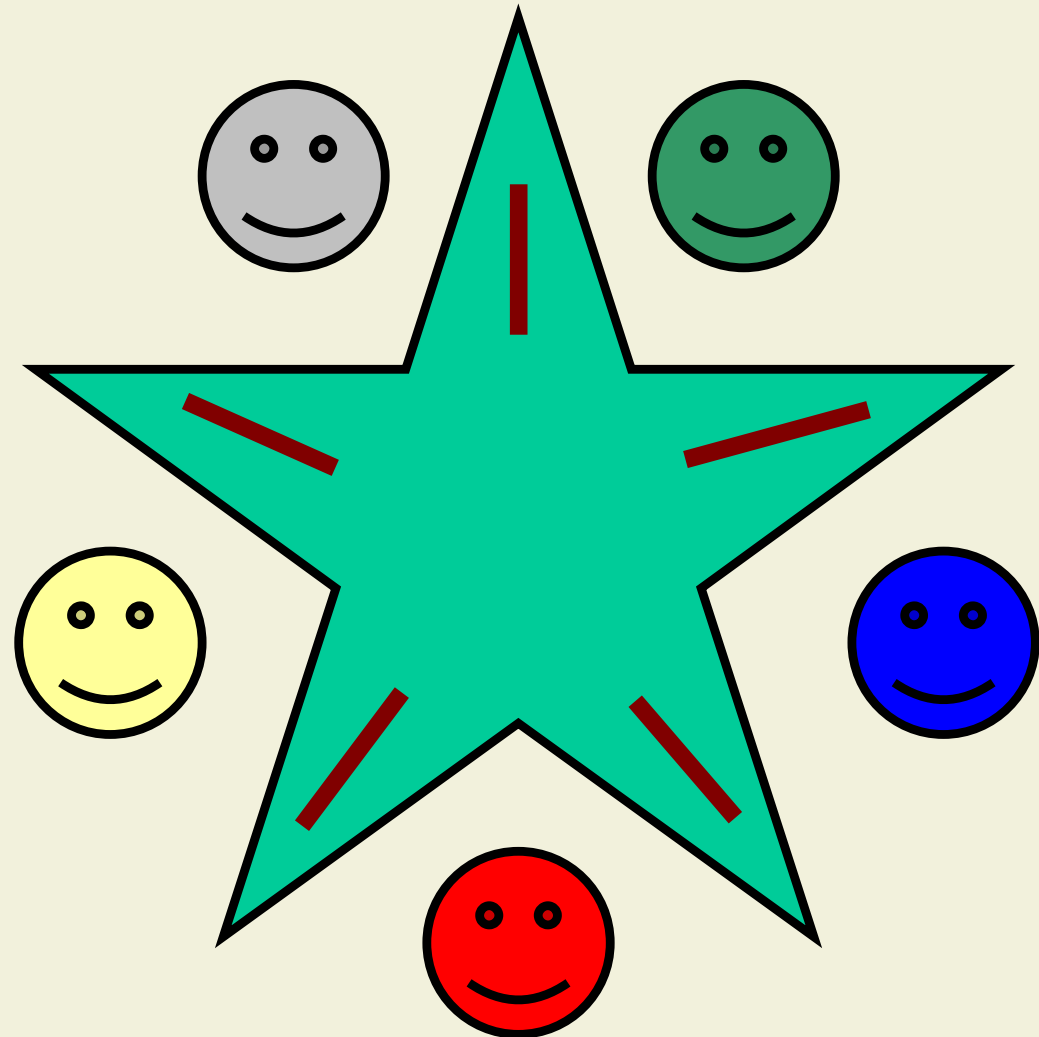
- All producers wait, b/c the buffer is full
- The buffer becomes empty and all consumers start to wait
- Some first producer A fills the buffer and then notifies another producer B
- That producer B sees a full buffer and waits
- Producer A sees a full buffer and waits

notify VS. notifyAll

- `java.lang.Object: final void notify()`
Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, **one** of them is chosen to be awakened. The choice is **arbitrary** and occurs at the discretion of the implementation. A thread waits on an object's monitor by calling one of the wait methods.
- `java.lang.Object: final void notifyAll()`
Wakes up **all** threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods.

Dining Philosophers Problem

- n Philosophers
- n Chopsticks
- 2 Chopsticks needed for eating



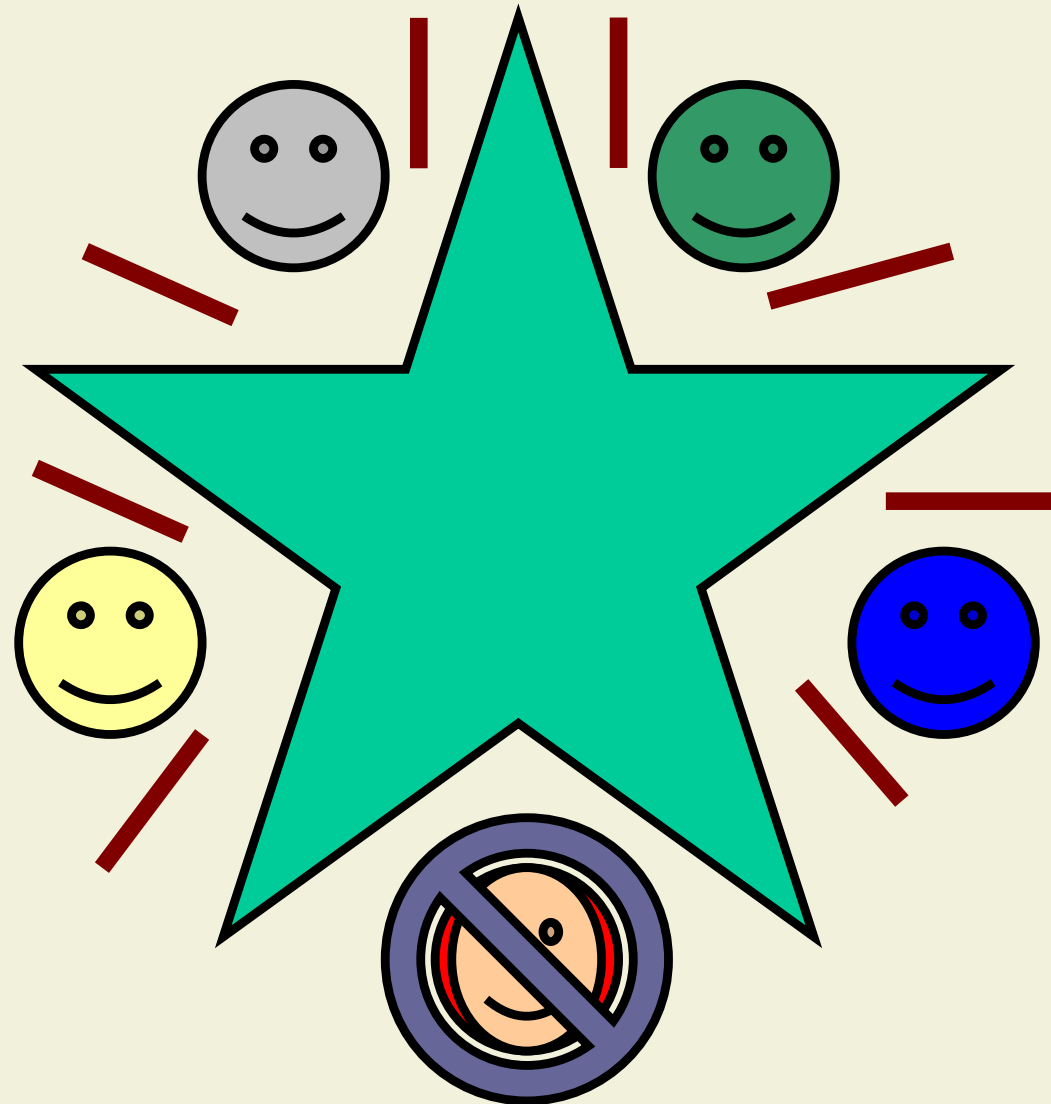
Problem: Deadlock

Everybody picks
the left chopstick
and then waits
forever to get
the right chopstick.



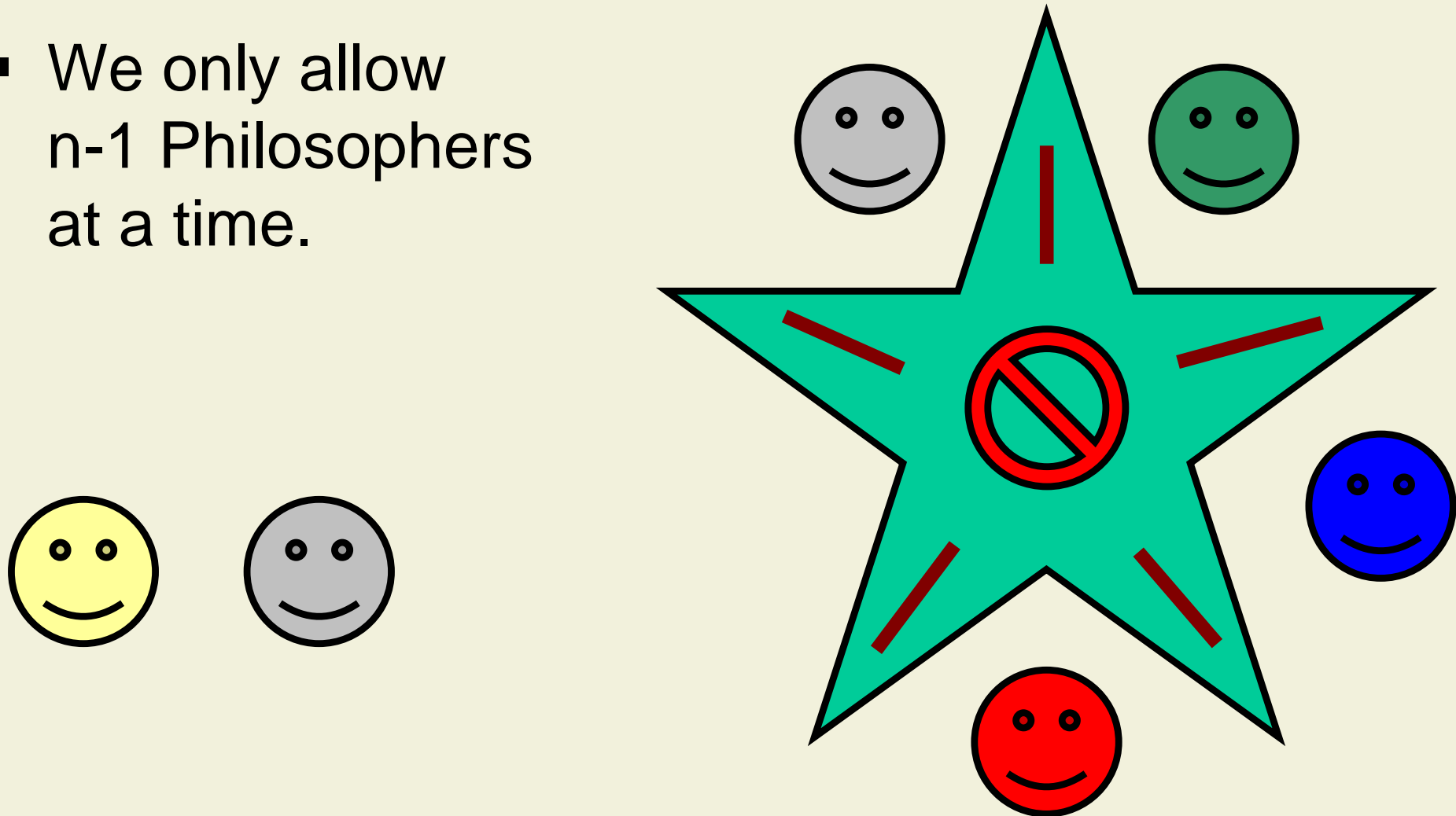
Problem: Starvation

Some philosophers
Never get a chance
to pick up both
chopsticks and
“starve”.



Solution 1: Central Control

- We only allow $n-1$ Philosophers at a time.



Solution 1: Der Philosoph

```
public class Philosoph extends Thread {  
    public void run() {  
        while( true ) {  
            denken();  
  
            t.anmelden(id);  
            t.gabel[id].belegen(id);  
            t.gabel[(id+1)%t.n].belegen((id+1)%t.n);  
  
            essen();  
  
            t.gabel[id].freigeben(id);  
            t.gabel[(id+1)%t.n].freigeben((id+1)%t.n);  
            t.abmelden(id);  
        }  
    }  
}
```

Solution 1: Die Gabel

```
public class Gabel {  
    public boolean belegt;  
  
    public synchronized void belegen(int id) {  
        if( belegt ) try { wait(); }  
            catch( InterruptedException e ) {};  
        belegt = true;  
    }  
  
    public synchronized void freigeben(int id) {  
        belegt = false;  
        notify();  
    }  
}
```

Solution 1: Der Tisch

```
public class Tisch {  
    public Gabel[] gabel;  
    private int phil;  
    Philosoph[] phils;  
  
    public synchronized void anmelden(int id) {  
        if( phil >= n ) try { wait(); }  
            catch( InterruptedException e ) {}  
        phil = phil + 1;  
    }  
  
    public synchronized void abmelden(int id) {  
        phil = phil - 1;  
        notify();  
    }  
}
```

Solution 2: Ordered Resources

- We demand that the chopsticks are picked up in a specific order
- For one philosopher this means that he will pick up the right chopstick first
- The deadlock situation of all philosophers picking up the left chopstick at the same time is prevented

Solution 2: Der Philosoph

```
public void run() {
    int first, second;

    first = id < ((id+1)%t.n) ? id : (id+1)%t.n;
    second = id < ((id+1)%t.n) ? (id+1)%t.n : id;

    while (true) {
        denken();
        t.gabel[first].belegen(first);
        t.gabel[second].belegen(second);

        essen();
        t.gabel[first].freigeben(first);
        t.gabel[second].freigeben(second);
    }
}
```

Solution 3: Atomic Operation

- We only allow both chopsticks to be picked up together
- Needs a critical section for the atomic operation
- Careful with what objects are used for synchronization

Solution 3: Die Gabel

```
public class Gabel {  
    public boolean belegt;  
  
    public Gabel() { belegt = false; }  
  
    public synchronized void belegen(int id) {  
        belegt = true;  
    }  
  
    public synchronized void freigeben(int id) {  
        belegt = false;  
    }  
}
```


Solution 3: Der Philosoph

```
public void run() {  
    while( true ) {  
        denken( );  
  
        synchronized( t ) {  
            while( t.gabel[id].belegt ||  
                   t.gabel[(id+1)%t.n].belegt ) {  
                try { t.wait(); }  
                catch( InterruptedException e ) { };  
            }  
            t.gabel[id].belegen(id);  
            t.gabel[(id+1)%t.n].belegen( (id+1)%t.n );  
        }  
    }  
}
```

Solution 3: Der Philosoph

...

essen() ;

```
synchronized( t ) {  
    t.gabel[id].freigeben(id) ;  
    t.gabel[(id+1)%t.n].freigeben((id+1)%t.n) ;  
  
    t.notifyAll() ;  
}  
}  
}
```