Technische Hochschule **Rosenheim**
Technical University of Applied Sciences

Object-oriented programming
Chapter 13 – Parallel execution

Prof. Dr Kai Höfig

Programmes are processes

- Until now, our programmes were stand-alone processes in which the instructions were executed individually, one after the other, and in the order in which they were written. The programme below

```
public class MyProgramme {
    String name;
  MyProgramme(String name) {
    this.name = name;
    System.out.println("Created MyProgramme: " + name);
    }
  void printNum(int n) {
    System.out.println(name + ": " + n);
    }
  public static void main(String[] args) {
    MyProgramme mp = new MyProgramme("Test");
    for (int i = 0; i < 3; i++)
      mp.printNum(i);
    }
}
```

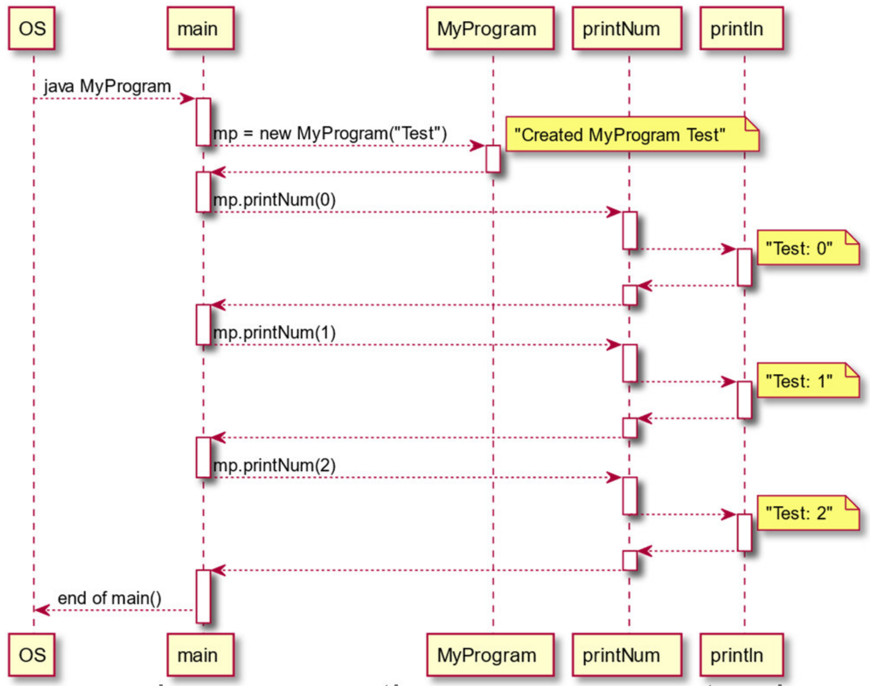
- produces the following output:

```
Created MyProgramme Test
    Test: 0
  Test: 1
  Test: 2
```

Processes

- A sequence diagram shows the programme sequence; the methods are plotted as columns:

```
public class MyProgramme {
    String name;
  MyProgramme(String name) {
    this.name = name;
    System.out.println("Created MyProgramme: " + name);
    }
  void printNum(int n) {
    System.out.println(name + ": " + n);
    }
  public static void main(String[] args) {
    MyProgramme mp = new MyProgramme("Test");
    for (int i = 0; i < 3; i++)
      mp.printNum(i);
    }
}
```



- This also corresponds with the order in the debugger when we run the programme step by step (with the step-into statement). A process has an isolated and independent environment. It has its own full set of runtime resources; this particularly applies to the memory: Each process has its own address space for variables.

Runnables: BeanCounter

- The following BeanCounter class models a bean counter. In the constructor, an instance gets a name and allocates a large array of numbers. The call of `run()` (of the `Runnable` interface) sorts this array.

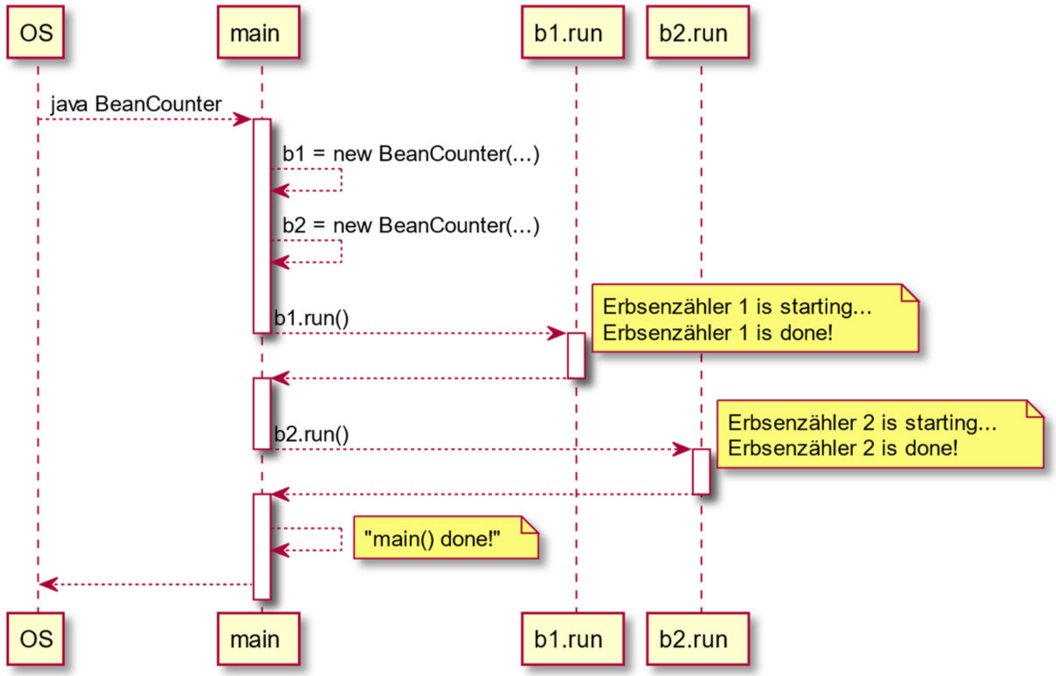
```java
class BeanCounter implements Runnable {
  private final String name;
  private final double[] data;
  BeanCounter(String name, int n) {
    this.name = name;
    this.data = new double [n];
    }
  @Override
  public void run() {
    System.out.println(name + " is starting...");
    Arrays.sort(data);
    System.out.println(name + " is done!");
    }
}
```

Callers of BeanCounter

- And here's an example where two BeanCounters are first created and then set to work; afterwards an end message is output.

```
public class CallBeanCounter {
  public static void main(String... args) {
    BeanCounter b1 = new BeanCounter("BeanCounter 1", 10000);
    BeanCounter b2 = new BeanCounter("BeanCounter 2", 1000);
        b1.run();
        b2.run();
    System.out.println("main() done!");
    }
}
```

```
          BeanCounter 1 is starting...
        BeanCounter 1 is done!
        BeanCounter 2 is starting...
        BeanCounter 2 is done!
    main() done!
```



Hard-working BeanCounters with threads

- If we now want these BeanCounters to work simultaneously (concurrently, in parallel), we can use the `Thread` class. This takes an instance of `Runnable`, whose `run()` method runs in a separate thread (thread of execution), as soon as the thread's `start()` method is called.

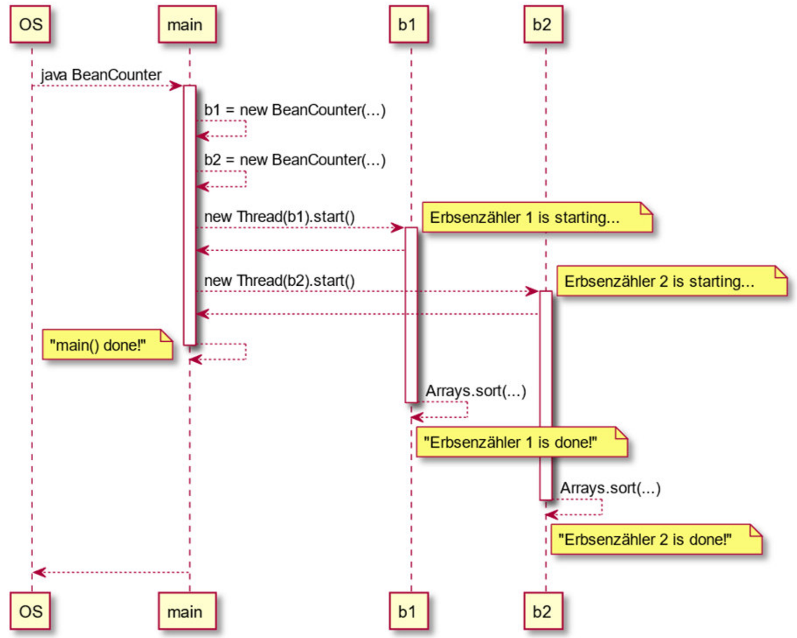
```
public class CallBeanCounterAsThreads {

     public static void main(String[] args) {
        BeanCounter b1 = new BeanCounter("BeanCounter 1", 10000);
        BeanCounter b2 = new BeanCounter("BeanCounter 2", 1000);

          new Thread(b1).start();

          new Thread(b2).start();
        System.out.println("main() done!");
    }
}
```

```
main() done!
BeanCounter 1 is starting...
BeanCounter 2 is starting...
BeanCounter 2 is done!
BeanCounter 1 is done!
```



OS
main
b1
b2
java BeanCounter
b1 = new BeanCounter(...)
b2 = new BeanCounter(...)
new Thread(b1).start()
Erbsenzähler 1 is starting...
new Thread(b2).start()
Erbsenzähler 2 is starting...
"main() done!"
Arrays.sort(...)
"Erbsenzähler 1 is done!"
Arrays.sort(...)
"Erbsenzähler 2 is done!"
OS
main
b1
b2

Threads: examples

The three methods `main`, `b1.run` and `b2.run` were processed in parallel. Depending on the number of processors and the load, a different order of the lines can also be output. This is because only one thread can write to `System.out` at any time. Instead of giving the `Thread` a `Runnable`, we can also inherit from `Thread` and overwrite the `run` method. Threads are sometimes called lightweight processes; they exist in processes and have shared resources. On the one hand, this enables communication by the threads, but on the other hand, it also has risks.

Parallel programming with threads can be seen in all modern applications:

- Browser: simultaneously loading and rendering resources on a web page
- Simultaneous rendering of multiple animations
- Handling user interactions such as clicks or swiping
- Sorting with divide-and-conquer procedures
- Simultaneous database, network and file operations
- Controllability of long-running processes

Joining

- The previous code example has a disadvantage: the `main` method is finished before the actual work (sorting) is completed. We can recognise this from the fact that the corresponding output appears before the "is done" output.

- Put in real life terms, this means that we delegate work to the team, but immediately report to the top that everything is completed (even though the team is still working).

- One (very bad) solution for this is to actively wait until a thread has finished by repeatedly calling the `isAlive` method.

- This active waiting works, but is a terrible idea: the status query with `isAlive` is very fast, so the `main` thread "runs hot" without really doing anything (useful).

Active wait

```
public class Joining implements Runnable {
  @Override
  public void run() {
    System.out.println("Sleeping for 15 seconds");
    try {
      Thread.sleep(15000);
        } catch (InterruptedException e) {
      e.printStackTrace();
        }
    }
  public static void main(String[] args) {
        Thread t = new Thread(new Joining());
    t.start();
    while (t.isAlive())
            ; // do nothing, but really fast...
    System.out.println("Done!");
    }
}
```

- Here, *active wait* in the `main` method repeatedly calls the thread's `isAlive` method. This happens essentially as fast as the computer can manage, repeatedly in succession. This consumes resources, both in the `main` and the thread itself.

The `join` method

- A much more elegant solution is to wait for this using the thread's `join` method:

```
public static void main(String[] args) throws
InterruptedException {
    Thread t = new Thread(new Joining());
  t.start();
  t.join(); // block/sleep until t is done
  System.out.println("Done!");
}
```

- The `join` and `sleep` methods can throw exceptions of the `InterruptedException` type here, which must be handled accordingly.

- `join` can be used whenever there is access to the thread reference (here `t`). For example, we can give a thread reference to another thread, so that this thread can only start after the end of the other thread.

Shared resources

1. Now let’s give our BeanCounters access to a common resource, a counter:

```
class Counter {
  private int c = 0;
  int getCount() {
    return c;
    }
  void increment() {
    c = c + 1;
    }
}
```

2. We now pass the same counter to four BeanCounters via constructor:

```
public static void main(String[] args) {
    Counter c = new Counter();
  new Thread(new TeamBeanCounter(c)).start();
  new Thread(new TeamBeanCounter(c)).start();
  new Thread(new TeamBeanCounter(c)).start();
  new Thread(new TeamBeanCounter(c)).start();
    System.out.println("Main beans: "+c.getCount());
}
```

3. The BeanCounters do the following with it:

```
public class TeamBeanCounter implements Runnable {
    Counter c;
  TeamBeanCounter(Counter c) {
    this.c = c;
    }
  @Override
  public void run() {
    for (int i = 0; i < 100_000; i++) {
      c.increment();
        }
    System.out.println("Total beans: " + c.getCount());
    }
}
```

4. And our five threads now produce the following output:

```
Main beans: 730
Total beans: 284429
Total beans: 310467
Total beans: 335498
Total beans: 400000
```

A synchronisation problem

- What has happened? All BeanCounters share the same `Counter` instance, but use them in their own respective threads of execution. Let's take a closer look at the `increment` method:

```
void increment() {
    c = c + 1;
}
```

- Here it is important to understand how such commands are actually executed by the CPU or in the JVM: first, the value of `c` must be loaded, then 1 added, and then finally the result must be assigned to `c`. Broken down into individual instructions, the statement block now looks like this:

```
void increment_brokenDown() {
    int tmp = c;
     ++tmp;
    c = tmp;
}
```

Shared resources: inconsistency!

- However, since each thread has its own thread of execution, it may be that two threads have their instruction pointer (essentially the line in the programme) at different positions, but share the memory as discussed above. Thus, interactions can occur between two or more threads:

```
void increment_brokenDown() {
    int tmp = c;
     ++tmp;
    c = tmp;
}
```

	Thread 1	Thread 2	*result*
1	tmp1 = c		tmp1 = 0
2		tmp2 = c	tmp2 = 0
3	++tmp1		tmp1 = 1
4		++tmp2	tmp2 = 1
5	c = tmp1		c = 1
6		c = tmp2	**c = 1 !**

"dirty read"

Too low, c should be 2!

The keyword `synchronized`

- In order to prevent this, we as programmers need to tell the JVM which programme sections can only be accessed by one thread at a time. This so-called locking is achieved in Java using the keyword `synchronized`, either as a modifier for methods, or as a block statement with a key object.

```
synchronized void
increment_sync() {
   c = c + 1;
}
```

or alternatively

```
void increment_sync_alt() {
   synchronized (this) {
      c = c + 1;
   }
}
```

- The output of our teamwork is then as follows:

```
   Main beans: 2636
Total beans: 298960
Total beans: 310526
Total beans: 338668
Total beans: 400000
```

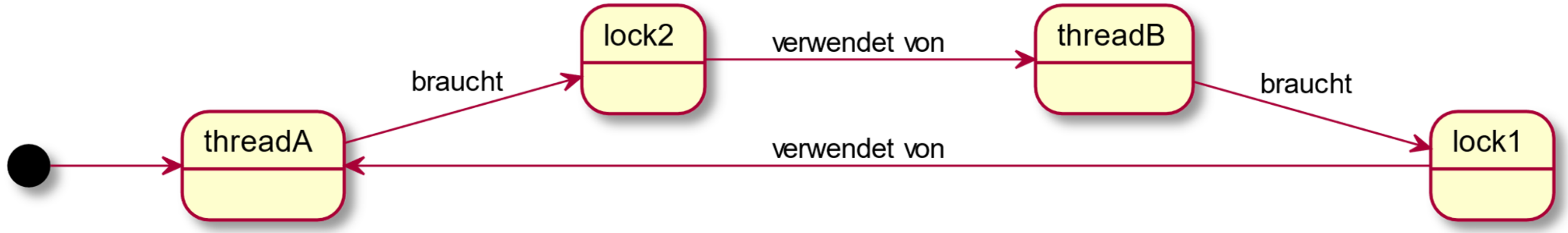
Main is finished, the hard-working BeanCounters have already counted up to 2636. Thread 1 is finished counting its 100k beans, the counter has already been increased to 298960, etc.

Synchronized methods vs. key objects

- The advantage of `synchronized` methods is that they are often a very simple solution to a synchronisation problem. The disadvantage is that the entire method block is locked.
- However, the advantage of block notation (`synchronized (lock) { ... }`) is that the locking is only applied when it is really necessary, e.g. for a block in a method. Although any object can be used as a key, the same instance must be used in all relevant places.
  - With `synchronized(this)`, it is ensured that the block section is only executed when `this` is released and is not locked by another process.
  - With `synchronized(lock)`, it is ensured that the block section is only executed when the `lock` object is released and is not locked by another process. In our counter example, this could be the variable `c` for example, but this can only be done with elements of the type `Object`.
  - For further information, please refer to the documentation: https://docs.oracle.com/javase/specs/jls/se9/html/jls-17.html

Deadlocks

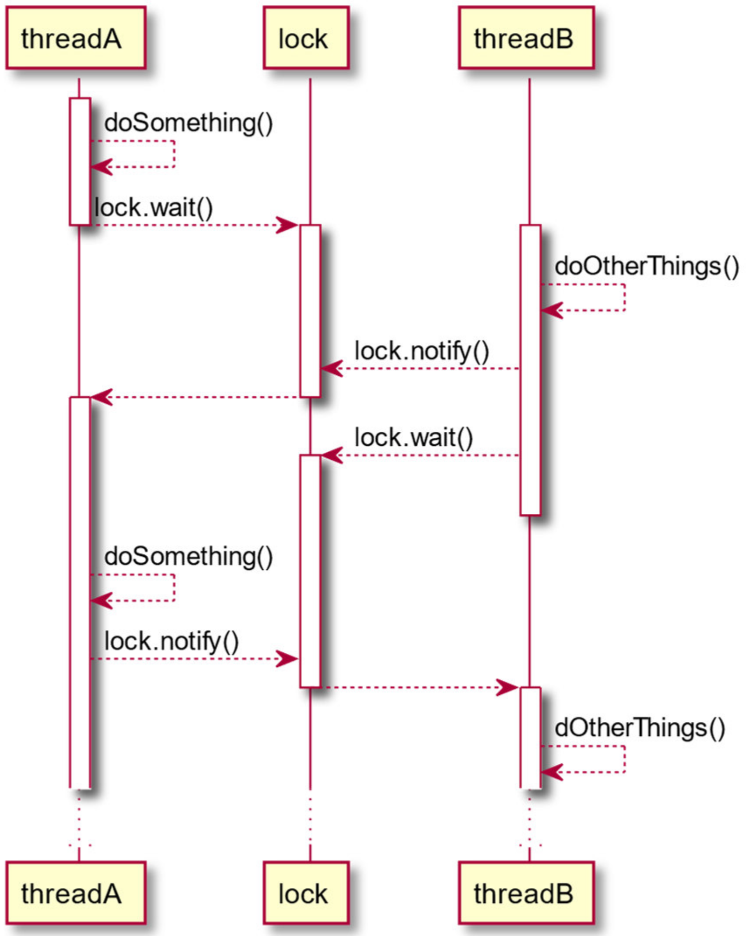
- The `synchronized` keyword allows us to safely change values that are used by multiple threads. However, sometimes simple locking is not enough, as the following diagram illustrates:



- This is a classic deadlock situation, as is sometimes also the case with government agencies: `threadA` needs `lock2`, but this is used by `threadB` ; `threadB` needs `lock1`, but this is used by `threadA` . The result is that nothing moves forward, the situation is stuck (deadlocked).

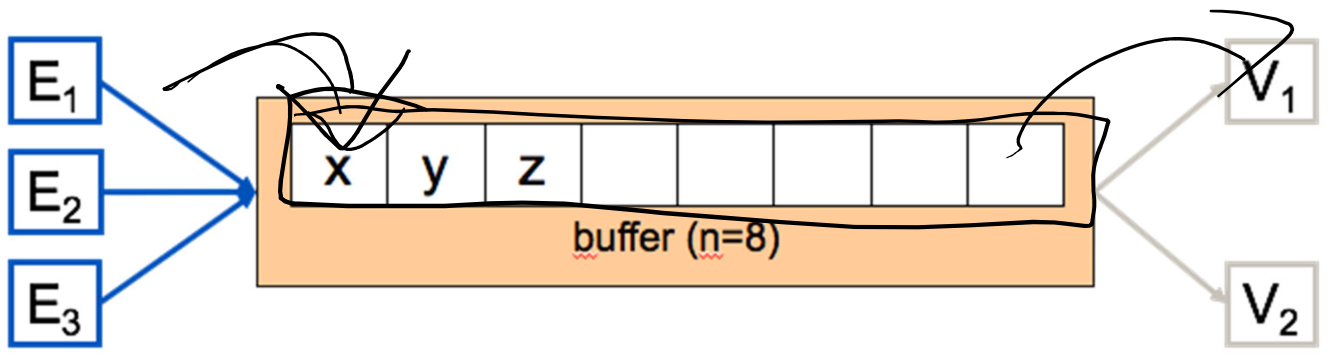
`wait`, `notify` and `notifyAll`

- To prevent such deadlocks, we can use the `wait()`, `notify()` and `notifyAll()` methods of the key object within `synchronized` sections, to wait at certain points until there is a notification from another thread. This mechanism can be used to let threads only work when there is really something to do.
- These methods are implemented for all objects.
- `notify` wakes (any) one thread, `notifyAll` correspondingly wakes all.
- *The application is explained below*



The producer-consumer problem

- One or more producers store data in a queue, and one or more consumers process data in the order in which it is made available.



- A typical application of this pattern is a video stream player: the producer is the decoder, which converts the data stream into image sequences; the consumer is the graphics driver, which then actually displays the images.

Example: problem

- A simple example is an extension of the counter already considered, here modelled as the class `CounterBuffer`. Producers now call `produce()` in order to increase the internal counter; consumers call `consume()` in order to decrease it:

```java
public class CounterBuffer {
  // must never be below 0!
  private int c = 0;

  // c <= max must always be true
  private final int max;

  CounterBuffer(int max) {
    this.max = max;
    }
  int getAvailable() {
    return c;
    }
  void produce() {
    c = c + 1;
    }
  void consume() {
    c = c - 1;
    }
}
```

- How is it now ensured that c<=max?

- For example, it's possible through exceptions:

```java
void produce_exc() {
  if (c == max)
    throw new RuntimeException("No space anymore!!");
  c = c + 1;
}
```

- However, the threads would then have to provide exception handling and actively wait again.

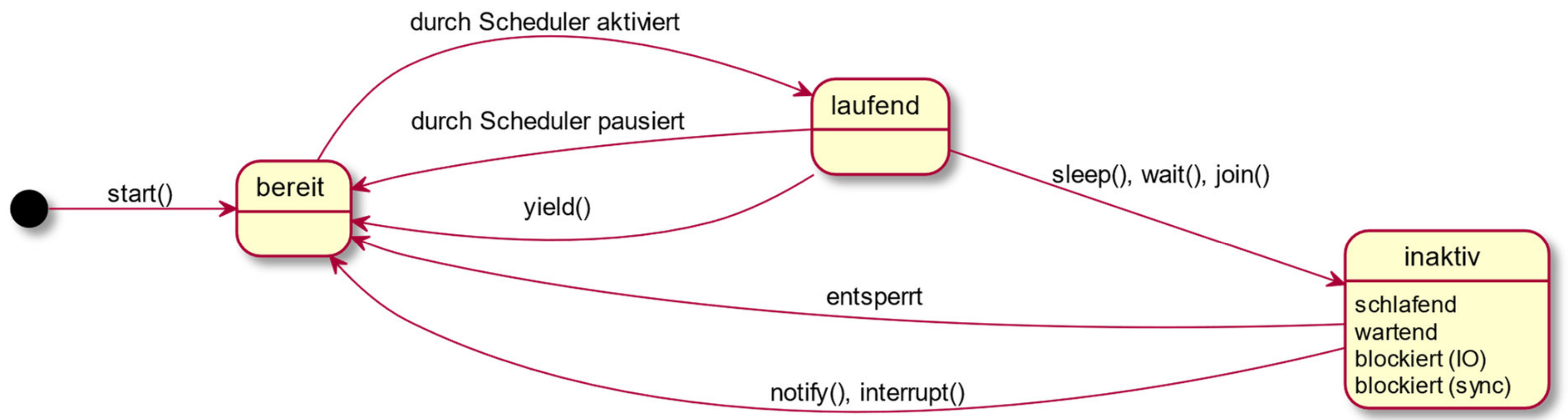
Example: solution

- The `CounterBuffer` class (as the resource to be locked) itself controls how the threads are notified.
- If a thread calls consume_new(), but nothing is available, it will wait (wait()) until something is made available. If a thread calls `produce_new()`, but there is no space, it will wait until space is available again.

```java
synchronized void produce_new() throws InterruptedException {
  // wait until something fits in again
  while (c >= max)
    wait();
  // now something fits in again!
  c = c + 1;
  // notify other threads which might be waiting
  notifyAll();
}
synchronized void consume_new() throws InterruptedException {
  // wait until at least 1 element is there!
  while (c < 0)
    wait();
  // now there is at least 1 element
  c = c - 1;
  // notify other threads which might be waiting
  notifyAll();
}
```

Life cycle of threads

- Finally, the full life cycle of a thread that illustrates the effect of start, wait, sleep and join.
- Further literature on the topic: https://docs.oracle.com/javase/specs/jls/se9/html/jls-17.html



durch Scheduler aktiviert
laufend
durch Scheduler pausiert
start()
bereit
yield()
sleep(), wait(), join()
inaktiv
schlafend
wartend
blockiert (IO)
blockiert (sync)
entsperrt
notify(), interrupt()