Multithreaded Programming

Multithreaded Programming is really difficult.

Problems

- race conditions
- non-determinism
- deadlock
- termination / livelock


“From a fundamental perspective, threads are seriously flawed as a computation model because they are wildly non-deterministic.”
How difficult?

Sahoo et al., ICSE 2010:
Empirical study of bugs in the Apache web server, Squid HTTP proxy server, MySQL database server, Tomcat servlet container, and OpenSSH server.

“Concurrency bugs (e.g., data races or deadlocks) form a small fraction (< 2%) of all bugs (even in highly concurrent applications)”
How difficult?

Sahoo et al., ICSE 2010:
Empirical study of bugs in the Apache web server, Squid HTTP proxy server, MySQL database server, Tomcat servlet container, and OpenSSH server.

“Concurrency bugs (e.g., data races or deadlocks) form a small fraction (< 2%) of all bugs (even in highly concurrent applications) but they have very different, more complex characteristics:

▶ nearly all are non-deterministic,
How difficult?

Sahoo et al., ICSE 2010:
Empirical study of bugs in the Apache web server, Squid HTTP proxy server, MySQL database server, Tomcat servlet container, and OpenSSH server.

“Concurrency bugs (e.g., data races or deadlocks) form a small fraction (< 2%) of all bugs (even in highly concurrent applications) but they have very different, more complex characteristics:

- nearly all are non-deterministic,
- they usually require more inputs to trigger,
How difficult?

Sahoo et al., ICSE 2010:
Empirical study of bugs in the Apache web server, Squid HTTP proxy server, MySQL database server, Tomcat servlet container, and OpenSSH server.

“Concurrency bugs (e.g., data races or deadlocks) form a small fraction (< 2%) of all bugs (even in highly concurrent applications) but they have very different, more complex characteristics:

- nearly all are non-deterministic,
- they usually require more inputs to trigger,
- they have more hangs/crashes and fewer incorrect output symptoms,
How difficult?

Sahoo et al., ICSE 2010:
Empirical study of bugs in the Apache web server, Squid HTTP proxy server, MySQL database server, Tomcat servlet container, and OpenSSH server.

“Concurrency bugs (e.g., data races or deadlocks) form a small fraction (< 2%) of all bugs (even in highly concurrent applications) but they have very different, more complex characteristics:

- nearly all are non-deterministic,
- they usually require more inputs to trigger,
- they have more hangs/crashes and fewer incorrect output symptoms,
- about 17% of them show different failure symptoms in different executions for the same inputs.”
How difficult?

Yin et al., ESEC/FSE 2011

Study on incorrect fixes in a commercial OS, FreeBSD, Linux, and OpenSolaris.

“Based on our samples, concurrency bugs are the most difficult ... to fix right.”

<table>
<thead>
<tr>
<th>type of bug</th>
<th>incorrect fix ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>concurrency</td>
<td>39%</td>
</tr>
<tr>
<td>semantic</td>
<td>17%</td>
</tr>
<tr>
<td>memory</td>
<td>14%</td>
</tr>
</tbody>
</table>
How difficult?

Lu et al., ASPLOS 2008:
Study on concurrency bugs in MySQL database server, Apache web server, Mozilla browser suite, and OpenOffice office suite.

- 17 out of 57 Mozilla bugs have at least one buggy patch
- On average, 0.4 buggy patches were released before the final correct patch
- 23 buggy patches
  - 6 only decrease the probability of the original bug
  - 5 introduce new concurrency bugs
  - 12 introduce new non-concurrency bugs
How difficult?

Fonseca et al., OSDI 2014:
Studies a systematic approach to trigger kernel concurrency bugs.

“In particular, the kernel developers provided simple stress tests, which continuously execute the same operations in a tight loop, waiting until a buggy interleaving occurs.”

“Despite the fact that we gave each stress test up to 24 hours to complete, bug A and bug D were not triggered at all by their corresponding stress tests. While the stress tests for bugs B and C did eventually trigger their corresponding bugs, they required significantly more executions (and time) [than our systematic approach].”
A sensitive operation in my lab today went completely wrong. An actuator on an electron microscope went over its boundary, and after a chain of events I lost $12 million of equipment. I’ve narrowed down over 40K lines in the faulty module to this:

```java
import java.util.*;

class A {
    static Point currentPos = new Point(1,2);
    static class Point {
        int x;
        int y;
        Point(int x, int y) {
            this.x = x;
            this.y = y;
        }
    }
    public static void main(String[] args) {
        new Thread() {
            void f(Point p) {
                synchronized(this) {
                    if (p.x+1 != p.y) {
                        System.out.println(p.x+" ~p.y");
                        System.exit(1);
                    }
                }
                @Override
                public void run() {
                    while (currentPos != null) {
                        while (true)
                            f(currentPos);
                    }
                }
            }
            currentPos = new Point(currentPos.x+1, currentPos.y+1);
            }.start();
            while (true)
                System.out.println(currentPos.x + " ~ currentPos.y");
        }
    }
```

Some samples of the output I’m getting:

```
$ java A
145281 145282
```

Why does this Java program terminate despite that apparently it shouldn’t (and didn’t)?
Aims

- How to approach multithreaded programming
- Show several multithreaded intricacies
- Provide lessons or rule of thumbs
Approach

- Show several bad code examples
- Show an implementation of synchronization constructs with its problems
- Show a problem in Java’s Thread library
Suppose you have an application you want to make concurrent.

steps

1. analyze data structures
   ▶ which data belongs logically together
2. determine granularity
   ▶ how can data best be protected
3. reason about concurrency
   ▶ use schedules
Running Example: Producer/Consumer

Producer/Consumer is a very well known pattern.

Typical usage:
  ▶ streaming multimedia
  ▶ task queues
steps

1. analyze data structures
   ▶ which data belongs logically together
2. determine granularity
   ▶ how can data best be protected
3. reason about concurrency
   ▶ use schedules

Let’s start with a sequential Buffer implementation that is to be shared between threads.
Buffer: implementation

class Buffer {

    private byte[] elements;
    private int nrElements;

    Buffer(int maxNrElements) {
        this.elements = new byte[maxNrElements];
        this.nrElements = 0;
    }

    void add(byte b) {
        if (isFull()) {
            throw new Error("The buffer is full");
        }
        elements[nrElements] = b;
        nrElements++;
    }

    byte remove() {
        if (isEmpty()) {
            throw new Error("The buffer is empty");
        }
        byte element = elements[nrElements - 1];
        nrElements--;
        return element;
    }
}
Buffer: implementation, continued

class Buffer {

    private byte[] elements;
    private int nrElements;

    int nrElements() {
        return nrElements;
    }

    boolean isFull() {
        return nrElements == elements.length;
    }

    boolean isEmpty() {
        return nrElements == 0;
    }

    int capacity() {
        return elements.length;
    }
}
Buffer

steps

1. analyze data structures
   elements and nrElements are logically ‘connected’

2. determine granularity
   every operation needs to be atomic
   Let’s make every method synchronized

3. reason about concurrency
Buffer: implementation

class Buffer {

    private byte[] elements;
    private int nrElements;

    Buffer(int maxNrElements) {
        this.elements = new byte[maxNrElements];
        this.nrElements = 0;
    }

    synchronized void add(byte b) {
        if (isFull()) {
            throw new Error("The buffer is full");
        }
        elements[nrElements] = b;
        nrElements++;
    }

    synchronized byte remove() {
        if (isEmpty()) {
            throw new Error("The buffer is empty");
        }
        byte element = elements[nrElements - 1];
        nrElements--;
        return element;
    }
}
Buffer: implementation, continued

class Buffer {

    private byte[] elements;
    private int nrElements;

    synchronized int nrElements() {
        return nrElements;
    }

    synchronized boolean isFull() {
        return nrElements == elements.length;
    }

    synchronized boolean isEmpty() {
        return nrElements == 0;
    }

    synchronized int capacity() {
        return elements.length;
    }
}
class Producer extends Thread {

    Buffer buffer;

    Producer(Buffer buffer) {
        this.buffer = buffer;
    }

    public void run() {
        while (true) {
            if (buffer.isFull()) {  
                buffer.wait();
            }

            buffer.add(produce());

            buffer.notify();
        }
    }

    byte produce() {
        return (byte) 0;
    }
}
Consumer implementation

```java
class Consumer extends Thread {

    Buffer buffer;

    Consumer(Buffer buffer) {
        this.buffer = buffer;
    }

    public void run() {
        while (true) {
            if (buffer.isEmpty()) {
                buffer.wait();
            }
            consume(buffer.remove());
            buffer.notify();
        }
    }

    void consume(byte b) {
    }
}
```
Conditional dependency

Within the application there is a conditional dependency between calls to the synchronized buffer.

Can we make a schedule in which an error occurs?
Step 3: Reasoning about concurrency

Pragmatic approach

Create a schedule and find an interleaving of threads which leads to an error.

- find the crucial shared state
- pick a thread and write down actions
- interleave threads with each other
Schedule

Go to producer

producer: if (buffer.isFull()) -> false
buffer.add(produce())
consumer 0: if (buffer.isEmpty()) -> false
consumer 1: if (buffer.isEmpty()) -> false
consume(buffer.remove())
consumer 0: consume(buffer.remove()) -> Error
producer: if (buffer.isFull()) -> false
producer: if (buffer.isFull()) -> false
buffer.add(produce())
producer: if (buffer.isFull()) -> false
buffer.add(produce())

consumer 0: if (buffer.isEmpty()) -> false
consumer 1: if (buffer.isEmpty()) -> false
consume(buffer.remove())
consumer 0: consume(buffer.remove()) -> Error
producer: if (buffer.isFull()) -> false
    buffer.add(produce())

consumer 0: if (buffer.isEmpty()) -> false

consumer 1: if (buffer.isEmpty()) -> false
producer: if (buffer.isFull()) -> false
buffer.add(produce())

consumer 0: if (buffer.isEmpty()) -> false
consumer 1: if (buffer.isEmpty()) -> false
consume(buffer.remove())
producer: if (buffer.isFull()) -> false
       buffer.add(produce())

consumer 0: if (buffer.isEmpty()) -> false

consumer 1: if (buffer.isEmpty()) -> false
           consume(buffer.remove())

consumer 0: consume(buffer.remove()) -> Error
Lesson 1

Be very careful with synchronized datastructures

There may be conditional dependencies between operations within the application.

Examples from the Java Collections Framework:
  ▶ Vector (sort of deprecated)
  ▶ Collections.synchronizedCollection()
      makes every collection a synchronized collection

Example

```java
if (vector.size() < 100) {
    vector.add(element);
}
```
Producer: incorrect implementation

Exception:
IllegalMonitorStateException because we do not own the monitor of the buffer.

class Producer extends Thread {

    Buffer buffer;

    public void run() {
        while (true) {
            if (buffer.isFull()) {
                buffer.wait();
            }
        }

        buffer.add(produce());

        buffer.notify();
    }

    byte produce() {
        return (byte) 0;
    }
}
Producer: improved implementation

class Producer extends Thread {

    public void run() {
        while (true) {
            synchronized (buffer) {
                if (buffer.isFull()) {
                    buffer.wait();
                }

                buffer.add(produce());

                buffer.notify();
            }
        }
    }

    private byte produce() {
        return (byte) 0;
    }
}
Schedule

consumer 0: synchronized (buffer)
if (buffer.isEmpty()) -> true
buffer.wait(); // releases lock

producer: synchronized (buffer)
if (buffer.isFull()) -> false
buffer.add(produce())
buffer.notify();
(releases lock)

consumer 0: (woken up)

consumer 1: synchronized (buffer)
if (buffer.isEmpty()) -> false
consume(buffer.remove());
buffer.notify();
(releases lock)

consumer 0: (returns from buffer.wait() and acquires lock)
consume(buffer.remove()); -> Error
Schedule

consumer 0: synchronized (buffer)
Schedule

consumer 0: synchronized (buffer)
    if (buffer.isEmpty()) -> true

consumer 0: (woken up)

consumer 1: synchronized (buffer)
    if (buffer.isEmpty()) -> false
    consume(buffer.remove());
    buffer.notify();
    (releases lock)

consumer 0: (returns from buffer.wait() and acquires lock)
    consume(buffer.remove()); -> Error
Schedule

consumer 0: synchronized (buffer)
if (buffer.isEmpty()) -> true
  buffer.wait(); // releases lock

producer: synchronized (buffer)
if (buffer.isFull()) -> false
  buffer.add(produce())
  buffer.notify(); (releases lock)

consumer 0: (woken up)
consumer 1: synchronized (buffer)
if (buffer.isEmpty()) -> false
  consume(buffer.remove())
  buffer.notify(); (releases lock)
consumer 0: (returns from buffer.wait() and acquires lock)
  consume(buffer.remove()); -> Error
Schedule

consumer 0: synchronized (buffer)
   if (buffer.isEmpty()) -> true
   buffer.wait(); // releases lock

producer: synchronized (buffer)

consumer 1: synchronized (buffer)
   if (buffer.isFull()) -> false
   buffer.add(produce())
   buffer.notify(); // releases lock

consumer 0: (woken up)

consumer 0: (returns from buffer.wait() and acquires lock)
   consume(buffer.remove()); -> Error
Schedule

consumer 0: synchronized (buffer)
    if (buffer.isEmpty()) -> true
        buffer.wait(); // releases lock

producer: synchronized (buffer)
    if (buffer.isFull()) -> false

consumer 0: (woken up)
consumer 1: synchronized (buffer)
    if (buffer.isEmpty()) -> false
        consume(buffer.remove());
    buffer.notify(); // releases lock

consumer 0: (returns from buffer.wait() and acquires lock)
    consume(buffer.remove()); -> Error
Schedule

consumer 0: synchronized (buffer)
    if (buffer.isEmpty()) -> true
        buffer.wait(); // releases lock

producer: synchronized (buffer)
    if (buffer.isFull()) -> false
        buffer.add(produce())
Schedule

consumer 0: synchronized (buffer)
  if (buffer.isEmpty()) -> true
  buffer.wait(); // releases lock

producer: synchronized (buffer)
  if (buffer.isFull()) -> false
  buffer.add(produce())
  buffer.notify();

consumer 0: (woken up)
consumer 1: synchronized (buffer)
  if (buffer.isEmpty()) -> false
  consume(buffer.remove())
  buffer.notify();

consumer 0: (returns from buffer.wait() and acquires lock)
  consume(buffer.remove()); -> Error
Schedule

consumer 0: synchronized (buffer)
    if (buffer.isEmpty()) -> true
        buffer.wait(); // releases lock

producer:     synchronized (buffer)
    if (buffer.isFull()) -> false
        buffer.add(produce())
        buffer.notify();
    (releases lock)

consumer 0: (woken up)
consumer 1: synchronized (buffer)
    if (buffer.isEmpty()) -> false
        consume(buffer.remove());
    buffer.notify();
    (returns from buffer.wait() and acquires lock)
    consume(buffer.remove()); -> Error
Schedule

consumer 0: synchronized (buffer)
    if (buffer.isEmpty()) -> true
        buffer.wait(); // releases lock

producer:    synchronized (buffer)
    if (buffer.isFull()) -> false
        buffer.add(produce())
        buffer.notify();
    (releases lock)

consumer 0: (woken up)
Schedule

consumer 0: synchronized (buffer)
    if (buffer.isEmpty()) -> true
    buffer.wait(); // releases lock

producer:  synchronized (buffer)
    if (buffer.isFull()) -> false
    buffer.add(produce())
    buffer.notify();
    (releases lock)

consumer 0: (woken up)

consumer 1: synchronized (buffer)
Schedule

consumer 0: synchronized (buffer)
    if (buffer.isEmpty()) -> true
    buffer.wait(); // releases lock

producer: synchronized (buffer)
    if (buffer.isFull()) -> false
    buffer.add(produce())
    buffer.notify();
    (releases lock)

consumer 0: (woken up)

consumer 1: synchronized (buffer)
    if (buffer.isEmpty()) -> false
Schedule

consumer 0: synchronized (buffer)
    if (buffer.isEmpty()) -> true
        buffer.wait(); // releases lock

producer: synchronized (buffer)
    if (buffer.isFull()) -> false
        buffer.add(produce());
        buffer.notify();
        (releases lock)

consumer 0: (woken up)

consumer 1: synchronized (buffer)
    if (buffer.isEmpty()) -> false
        consume(buffer.remove());
Schedule

consumer 0: synchronized (buffer)
    if (buffer.isEmpty()) -> true
    buffer.wait(); // releases lock

producer: synchronized (buffer)
    if (buffer.isFull()) -> false
    buffer.add(produce())
    buffer.notify();
    (releases lock)

consumer 0: (woken up)

consumer 1: synchronized (buffer)
    if (buffer.isEmpty()) -> false
    consume(buffer.remove());
    buffer.notify();
Schedule

consumer 0: synchronized (buffer)
  if (buffer.isEmpty()) \rightarrow true
  buffer.wait(); // releases lock

producer: synchronized (buffer)
  if (buffer.isFull()) \rightarrow false
  buffer.add(produce())
  buffer.notify();
  (releases lock)

consumer 0: (woken up)

consumer 1: synchronized (buffer)
  if (buffer.isEmpty()) \rightarrow false
  consume(buffer.remove());
  buffer.notify();
  (releases lock)
Schedule

consumer 0: synchronized (buffer)
    if (buffer.isEmpty()) -> true
    buffer.wait(); // releases lock

producer: synchronized (buffer)
    if (buffer.isFull()) -> false
    buffer.add(produce())
    buffer.notify();
    (releases lock)

consumer 0: (woken up)

consumer 1: synchronized (buffer)
    if (buffer.isEmpty()) -> false
    consume(buffer.remove());
    buffer.notify();
    (releases lock)

consumer 0: (returns from buffer.wait() and acquires lock)
consumer 0: synchronized (buffer)
    if (buffer.isEmpty()) -> true
    buffer.wait(); // releases lock

producer: synchronized (buffer)
    if (buffer.isFull()) -> false
    buffer.add(produce());
    buffer.notify();
    (releases lock)

customer 0: (woken up)

customer 1: synchronized (buffer)
    if (buffer.isEmpty()) -> false
    consume(buffer.remove());
    buffer.notify();
    (releases lock)

customer 0: (returns from buffer.wait() and acquires lock)
    consume(buffer.remove()); -> Error
Lesson 2

Always re-check the condition when you wake-up from a signal.

- What causes a thread to go to sleep?
- When a thread wakes-up, has the situation changed?
Producer: improved implementation

class Producer extends Thread {

    public void run() {
        while (true) {
            synchronized (buffer) {
                while (buffer.isFull()) {
                    buffer.wait();
                }
                buffer.add(produce());
                buffer.notify();
            }
            buffer.add(produce());
            buffer.notify();
        }
    }

    private byte produce() {
        return (byte) 0;
    }
}
Schedule

Go to producer

consumer 0: synchronized (buffer) // acquires lock
while (buffer.isEmpty()) -> true
buffer.wait(); // releases lock

consumer 1: synchronized (buffer) // acquires lock
while (buffer.isEmpty()) -> true
buffer.wait(); // releases lock

producer 0: synchronized (buffer) // acquires lock
while (buffer.isFull()) -> false
producer 1: synchronized (buffer) // blocks
producer 0: buffer.add(produce())
buffer.notify(); // releases lock

producer 1: synchronized (buffer) // acquires lock
consumer 0: (woken up)
synchronized (buffer) // blocks
producer 0: synchronized (buffer) // blocks
Schedule

consumer 0: synchronized (buffer) // acquires lock
consumer 0: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true

consumer 1: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true

producer 0: synchronized (buffer) // acquires lock
    while (buffer.isFull()) -> false
producer 0: buffer.add(produce())
    buffer.notify(); // releases lock

producer 1: synchronized (buffer) // blocks

producer 0: synchronized (buffer) // blocks
consumer 0: synchronized (buffer) // acquires lock
  while (buffer.isEmpty()) -> true
  buffer.wait(); // releases lock
Schedule

consumer 0: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true
    buffer.wait(); // releases lock

consumer 1: synchronized (buffer) // acquires lock

producer 0: synchronized (buffer) // acquires lock
    while (buffer.isFull()) -> false
    producer 0: buffer.add(produce())
    buffer.notify(); // releases lock

producer 1: synchronized (buffer) // blocks

consumer 0: (woken up)
    synchronized (buffer) // blocks

producer 0: synchronized (buffer) // blocks
consumer 0: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true
    buffer.wait(); // releases lock

consumer 1: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true
consumer 0: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true
    buffer.wait(); // releases lock

counter 1: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true
    buffer.wait(); // releases lock
consumer 0: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true
    buffer.wait(); // releases lock

consumer 1: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true
    buffer.wait(); // releases lock

producer 0: synchronized (buffer) // acquires lock
    producer 0: buffer.add(produce())
    buffer.notify(); (releases lock)

producer 1: synchronized (buffer) // blocks

consumer 0: (woken up)
    synchronized (buffer) // blocks

producer 0: synchronized (buffer) // blocks
consumer 0: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true
    buffer.wait(); // releases lock

consumer 1: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true
    buffer.wait(); // releases lock

producer 0: synchronized (buffer) // acquires lock
    while (buffer.isFull()) -> false
    producer 0: buffer.add(produce())
    buffer.notify(); (releases lock)

producer 1: synchronized (buffer) // blocks
consumer 0: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true
        buffer.wait(); // releases lock

consumer 1: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true
        buffer.wait(); // releases lock

producer 0: synchronized (buffer) // acquires lock
    while (buffer.isFull()) -> false

producer 1: synchronized (buffer) // blocks
consumer 0: synchronized (buffer) // acquires lock
   while (buffer.isEmpty()) -> true
      buffer.wait(); // releases lock

consumer 1: synchronized (buffer) // acquires lock
   while (buffer.isEmpty()) -> true
      buffer.wait(); // releases lock

producer 0: synchronized (buffer) // acquires lock
   while (buffer.isFull()) -> false

producer 1: synchronized (buffer) // blocks

producer 0: buffer.add(produce())
consumer 0: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true
    buffer.wait(); // releases lock

consumer 1: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true
    buffer.wait(); // releases lock

producer 0: synchronized (buffer) // acquires lock
    while (buffer.isFull()) -> false

producer 1: synchronized (buffer) // blocks

producer 0: buffer.add(produce())
    buffer.notify();
consumer 0: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true
    buffer.wait(); // releases lock

customer 1: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true
    buffer.wait(); // releases lock

producer 0: synchronized (buffer) // acquires lock
    while (buffer.isFull()) -> false

producer 1: synchronized (buffer) // blocks

producer 0: buffer.add(produce())
    buffer.notify();
    (releases lock)
consumer 0: synchronized (buffer) // acquires lock
       while (buffer.isEmpty()) -> true
       buffer.wait(); // releases lock

consumer 1: synchronized (buffer) // acquires lock
       while (buffer.isEmpty()) -> true
       buffer.wait(); // releases lock

producer 0: synchronized (buffer) // acquires lock
       while (buffer.isFull()) -> false

producer 1: synchronized (buffer) // blocks

producer 0: buffer.add(produce())
       buffer.notify();
       (releases lock)

producer 1: synchronized (buffer) // acquires lock
consumer 0: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true
    buffer.wait(); // releases lock

consumer 1: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true
    buffer.wait(); // releases lock

producer 0: synchronized (buffer) // acquires lock
    while (buffer.isFull()) -> false

producer 1: synchronized (buffer) // blocks

producer 0: buffer.add(produce())
    buffer.notify();
    (releases lock)

producer 1: synchronized (buffer) // acquires lock

consumer 0: (woken up)
consumer 0: synchronized (buffer) // acquires lock
   while (buffer.isEmpty()) -> true
       buffer.wait(); // releases lock

consumer 1: synchronized (buffer) // acquires lock
   while (buffer.isEmpty()) -> true
       buffer.wait(); // releases lock

producer 0: synchronized (buffer) // acquires lock
   while (buffer.isFull()) -> false

producer 1: synchronized (buffer) // blocks

producer 0: buffer.add(produce())
       buffer.notify();
       (releases lock)

producer 1: synchronized (buffer) // acquires lock

consumer 0: (woken up)
   synchronized (buffer) // blocks
consumer 0: synchronized (buffer) // acquires lock
   while (buffer.isEmpty()) -> true
      buffer.wait(); // releases lock

consumer 1: synchronized (buffer) // acquires lock
   while (buffer.isEmpty()) -> true
      buffer.wait(); // releases lock

producer 0: synchronized (buffer) // acquires lock
   while (buffer.isFull()) -> false

producer 1: synchronized (buffer) // blocks

producer 0:    buffer.add(produce())
   buffer.notify();
      (releases lock)

producer 1: synchronized (buffer) // acquires lock

consumer 0: (woken up)
   synchronized (buffer) // blocks

producer 0: synchronized (buffer) // blocks
Go to producer

producer 1: while (buffer.isFull())
            -> true
            buffer.wait(); // releases the lock

producer 0: synchronized (buffer) // acquires lock
            while (buffer.isFull())
                -> true
                buffer.wait(); // releases the lock

consumer 0: synchronized (buffer) // acquires lock
            while (buffer.isEmpty())
                -> false
                consume(buffer.remove());
                buffer.notify() // releases lock

consumer 1: (woken up)
            synchronized (buffer) // acquires lock
            while (buffer.isEmpty())
                -> true
                buffer.wait(); // releases the lock

consumer 0: synchronized (buffer) // acquires lock
            while (buffer.isEmpty())
                -> true
                buffer.wait(); // releases the lock
            -> Deadlock
Schedule, continued

producer 1: while (buffer.isFull()) -> true
Schedule, continued

producer 1: while (buffer.isFull()) -> true
  buffer.wait(); // releases the lock
producer 1: while (buffer.isFull()) -> true
     buffer.wait(); // releases the lock

producer 0: synchronized (buffer) // acquires lock
Schedule, continued

producer 1: while (buffer.isFull()) -> true
    buffer.wait(); // releases the lock

producer 0: synchronized (buffer) // acquires lock
    while (buffer.isFull()) -> true

producer 0: synchronized (buffer) // acquires lock
    while (buffer.isFull()) -> true

consumer 0: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> false
    consume(buffer.remove());
    buffer.notify() (releases lock)

consumer 1: (woken up)
    synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true
    buffer.wait(); // releases the lock

Deadlock
producer 1: while (buffer.isFull()) -> true
    buffer.wait(); // releases the lock

producer 0: synchronized (buffer) // acquires lock
    while (buffer.isFull()) -> true
    buffer.wait(); // releases the lock

producer 0: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> false
    consume(buffer.remove());
    buffer.notify() (releases lock)

consumer 1: (woken up)
    synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true
    buffer.wait(); // releases the lock

consumer 0: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true
    buffer.wait(); // releases the lock
Schedule, continued

producer 1: while (buffer.isFull()) -> true
        buffer.wait(); // releases the lock

producer 0: synchronized (buffer) // acquires lock
            while (buffer.isFull()) -> true
            buffer.wait(); // releases the lock

consumer 0: synchronized (buffer) // acquires lock
Schedule, continued

producer 1: while (buffer.isFull()) -> true
  buffer.wait(); // releases the lock

producer 0: synchronized (buffer) // acquires lock
  while (buffer.isFull()) -> true
  buffer.wait(); // releases the lock

consumer 0: synchronized (buffer) // acquires lock
  while (buffer.isEmpty()) -> false

consumer 1: (woken up)
  synchronized (buffer) // acquires lock
  while (buffer.isEmpty()) -> true
  buffer.wait(); // releases the lock

  consumer 0: synchronized (buffer) // acquires lock
  while (buffer.isEmpty()) -> true
  buffer.wait(); // releases the lock -> Deadlock
producer 1: while (buffer.isFull()) -> true
    buffer.wait(); // releases the lock

producer 0: synchronized (buffer) // acquires lock
    while (buffer.isFull()) -> true
    buffer.wait(); // releases the lock

consumer 0: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> false
    consume(buffer.remove());

consumer 1: (woken up)
Schedule, continued

producer 1:
while (buffer.isFull()) -> true
    buffer.wait(); // releases the lock

producer 0:
synchronized (buffer) // acquires lock
    while (buffer.isFull()) -> true
        buffer.wait(); // releases the lock

consumer 0:
synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> false
    consume(buffer.remove());
    buffer.notify()
producer 1: while (buffer.isFull()) -> true
    buffer.wait(); // releases the lock

producer 0: synchronized (buffer) // acquires lock
    while (buffer.isFull()) -> true
    buffer.wait(); // releases the lock

consumer 0: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> false
    consume(buffer.remove());
    buffer.notify() (releases lock)
producer 1: while (buffer.isFull()) -> true
    buffer.wait(); // releases the lock

producer 0: synchronized (buffer) // acquires lock
    while (buffer.isFull()) -> true
        buffer.wait(); // releases the lock

consumer 0: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> false
    consume(buffer.remove());
    buffer.notify();
    (releases lock)

consumer 1: (woken up)
producer 1: while (buffer.isFull()) \(\rightarrow\) true
    buffer.wait(); // releases the lock

producer 0: synchronized (buffer) // acquires lock
    while (buffer.isFull()) \(\rightarrow\) true
        buffer.wait(); // releases the lock

consumer 0: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) \(\rightarrow\) false
        consume(buffer.remove());
        buffer.notify();
        buffer.notify();
        (releases lock)

consumer 1: (woken up)
    synchronized (buffer) // acquires lock
producer 1: while (buffer.isFull()) -> true
  buffer.wait(); // releases the lock

producer 0: synchronized (buffer) // acquires lock
  while (buffer.isFull()) -> true
  buffer.wait(); // releases the lock

consumer 0: synchronized (buffer) // acquires lock
  while (buffer.isEmpty()) -> false
  consume(buffer.remove());
  buffer.notify()
  (releases lock)

consumer 1: (woken up)
  synchronized (buffer) // acquires lock
  while (buffer.isEmpty()) -> true
Schedule, continued

producer 1: while (buffer.isFull()) -> true
    buffer.wait(); // releases the lock

producer 0: synchronized (buffer) // acquires lock
    while (buffer.isFull()) -> true
        buffer.wait(); // releases the lock

consumer 0: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> false
        consume(buffer.remove());
        buffer.notify()
        (releases lock)

consumer 1: (woken up)
    synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true
        buffer.wait(); // releases the lock
producer 1: while (buffer.isFull()) -> true
    buffer.wait(); // releases the lock

producer 0: synchronized (buffer) // acquires lock
    while (buffer.isFull()) -> true
    buffer.wait(); // releases the lock

consumer 0: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> false
    consume(buffer.remove());
    buffer.notify()
    (releases lock)

consumer 1: (woken up)
    synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true
    buffer.wait(); // releases the lock

consumer 0: synchronized (buffer) // acquires lock
producer 1:  while (buffer.isFull()) -> true
           buffer.wait(); // releases the lock

producer 0: synchronized (buffer) // acquires lock
           while (buffer.isFull()) -> true
             buffer.wait(); // releases the lock

consumer 0: synchronized (buffer) // acquires lock
           while (buffer.isEmpty()) -> false
             consume(buffer.remove());
             buffer.notify()  
               (releases lock)

consumer 1: (woken up)
            synchronized (buffer) // acquires lock
            while (buffer.isEmpty()) -> true
              buffer.wait(); // releases the lock

consumer 0: synchronized (buffer) // acquires lock
            while (buffer.isEmpty()) -> true
Schedule, continued

producer 1: while (buffer.isFull()) -> true
    buffer.wait(); // releases the lock

producer 0: synchronized (buffer) // acquires lock
    while (buffer.isFull()) -> true
        buffer.wait(); // releases the lock

consumer 0: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> false
    consume(buffer.remove());
    buffer.notify()
    (releases lock)

consumer 1: (woken up)
    synchronized (buffer) // acquires lock
        while (buffer.isEmpty()) -> true
            buffer.wait(); // releases the lock

consumer 0: synchronized (buffer) // acquires lock
    while (buffer.isEmpty()) -> true
        buffer.wait(); // releases the lock -> Deadlock
Lesson 3

Almost always use `notifyAll()` or `signalAll()`

`notify()` chooses a thread non-deterministically to continue executing. Common false assumptions (pitfalls) are:

- the expected thread is woken up
- a thread that wakes up, immediately acquires the lock

**Overhead**

In a correct implementation, the overhead is not much, because most threads go to sleep immediately.
Producer: improved implementation

class Producer extends Thread {

    public void run() {
        while (true) {
            synchronized (buffer) {
                while (buffer.isFull()) {
                    buffer.wait();
                }

                buffer.add(produce());

                buffer.notifyAll();
            }

            buffer.add(produce());
        }
    }

    private byte produce() {
        return (byte) 0;
    }
}
Producer: improved implementation

Problem
This implementation is inefficient. Too many signals are sent.
Producer: correct implementation

class Producer extends Thread {

    public void run() {
        while (true) {
            synchronized (buffer) {
                while (buffer.isFull()) {
                    buffer.wait();
                }
            }

            buffer.add(produce());

            if (buffer.nrElements() == 1) {
                buffer.notifyAll();
            }
        }
    }

    private byte produce() {
        return (byte) 0;
    }
}
Deadlock example

ordering of locks

thread $t_0$ acquires lock $L_0$
thread $t_1$ acquires lock $L_1$
thread $t_0$ tries to acquire lock $L_1$
thread $t_1$ tries to acquire lock $L_0$
Observer pattern, sequential:

```java
public class ValueHolder {

    private List listeners = new LinkedList();
    private int value;

    public interface Listener {
        public void valueChanged(int newValue);
    }

    public void addListener(Listener listener) {
        listeners.add(listener);
    }

    public void setValue(int newValue) {
        value = newValue;
        Iterator i = listeners.iterator();
        while (i.hasNext()) {
            (Listener i.next()).valueChanged(newValue);
        }
    }
}
```
Observer pattern, synchronized:

```java
public class ValueHolder {

    private List listeners = new LinkedList();
    private int value;

    public interface Listener {
        public void valueChanged(int newValue);
    }

    public synchronized void addListener(Listener listener) {
        listeners.add(listener);
    }

    public synchronized void setValue(int newValue) {
        value = newValue;
        Iterator i = listeners.iterator();
        while (i.hasNext()) {
            ((Listener i.next()).valueChanged(newValue);
        }
    }
}
```
Suppose we have:

- ValueHolder $v$
- another object $l$ that implements Listener

We do not know the implementation of $l$.valueChanged(), $l$ might block on a lock that a thread $t$ holds.

This thread might try to set the value on $v$ → Deadlock.
Example:

```java
Lock lock;
ValueHolder v;
Listener l;

in l:
void valueChanged(int newValue) {
  lock.lock();
  process(newValue);
  lock.unlock();
}

thread 0: lock.lock();

thread 1: v.setValue(); // synchronized method
  l.valueChanged(...);
  lock.lock(); // blocks on the lock

thread 0: v.setValue(); // blocks on the synchronized method
```
Underlying problem:
Locking convention not in code.
For instance, locking scheme is not encoded in the signature of methods.
Locking is often communicated through comments in code.

Implication:
You cannot call third party’s code while holding a lock.
Observer pattern, improved:

```java
public class ValueHolder {

    private List listeners = new LinkedList();
    private int value;

    public interface Listener {
        public void valueChanged(int newValue);
    }

    public synchronized void addListener(Listener listener) {
        listeners.add(listener);
    }

    public void setValue(int newValue) {
        List copyOfListeners;
        synchronized (this) {
            value = newValue;
            copyOfListeners = new LinkedList(listeners);
        }
        Iterator i = copyOfListeners.iterator();
        while (i.hasNext()) {
            ((Listener i.next()).valueChanged(newValue);
        }
    }
}
```
Problem

Order in which `valueChanged()` is executed is non-deterministic.

A listener might conclude that the last value of the valueHolder is different than in reality.
Lesson 4

Be careful when calling third party’s code while holding a lock.

Watch ordering of locks closely.
Let’s build our own synchronization primitives with a few minimal assumptions:

- A spinning filter lock, c.f. Fig 2.7
- A ThreadManager with
  - resumeThreads() → (resume all threads)
  - suspendThread() → (suspend this thread)
Semaphores

- introduced by Edsger Dijkstra
- Dutch term: seinpaal
- one of the most basic synchronization primitives

- goal
  - obtain blocking semantics
  - do not forget signals
Semaphores: implementation

internal counter
if the counter is zero, a thread suspends

accessed by two operations

▶  p()
from the Dutch made up word prolagen, a contraction from
‘proberen te verlagen’ or ‘try to decrement’

▶  v()
from the Dutch word verhogen or ‘increment’
Semaphore

Pseudo code

shared counter

p() {
    while counter == 0 {
        suspend the thread
    }
    counter = counter - 1
}

v() {
    counter = counter + 1
    resume all threads
}
Semaphore

Steps

1. What data needs to be protected?

2. Granularity?

3. Reason about the concurrency?
Semaphore

Steps

1. What data needs to be protected?  
   counter

2. Granularity?

3. Reason about the concurrency?
Semaphore

Steps

1. What data needs to be protected?
   counter

2. Granularity?
   FilterLock is enough
   time spent inside the critical section is short

3. Reason about the concurrency?
Semaphore: real implementation

class Semaphore {

    FilterLock filterLock;
    int capacity;
    int counter;

    Semaphore(int capacity, int numberThreads) {
        this.filterLock = new FilterLock(numberThreads);
        this.counter = capacity;
    }

    void p() {
        try {
            filterLock.lock();
            while (counter == 0) {
                filterLock.unlock();
                ThreadManager.suspendThread();
                filterLock.lock();
            }
            counter -= 1;
        } finally {
            filterLock.unlock();
        }
    }
}
Semaphore: real implementation, continued

class Semaphore {

    FilterLock filterLock;
    int capacity;
    int counter;

    void v() {
        try {
            filterLock.lock();
            counter++;
        }
        finally {
            filterLock.unlock();
        }
        ThreadManager.resumeThreads();
    }
}
BlockingLock: pseudo code

With a semaphore, it is now very easy to build a blocking lock
BlockingLock: pseudo code

With a semaphore, it is now very easy to build a blocking lock

Pseudo code:

```java
semaphore = new Semaphore(1)

lock() {
    p()
}

unlock() {
    v()
}
```
class BlockingLock {

    Semaphore semaphore;
    int nrThreads;

    BlockingLock(int nrThreads) {
        this.semaphore = new Semaphore(1, nrThreads);
        this.nrThreads = nrThreads;
    }

    void lock() {
        semaphore.p();
    }

    void unlock() {
        semaphore.v();
    }
}

Running Example: Producer/Consumer

Besides a blocking lock, we also need condition variables

Pseudo code:
shared buffer

produce() {
    while (true) {
        lock()
        while buffer full
            wait()
            add element to buffer
            signal other threads
        unlock()
    }
}
Condition variables: pseudo code

With a semaphore, it is also very easy to build condition variables.
Condition variables: pseudo code

With a semaphore, it is also very easy to build condition variables

A condition variable is associated with a specific lock

Pseudo code:

```java
semaphore = new Semaphore(0)

lock

await() {
    lock.unlock()
    p()
    lock.lock()
}

signalAll() {
    for i in nrThreads
        v()
}
```
Notice obtaining the lock after a wake-up from a signal!!

Similar to the semantics of Java, with `wait()` and `notifyAll()`, it is not guaranteed that the awakened thread regains the lock!

**Pseudo code:**

```java
Semaphore = new Semaphore(0)
lock

await() {
    lock.unlock()
    p()
    lock.lock()
}

signalAll() {
    for i in nrThreads
        v()
}
```
SemaphoreCondition: real implementation

class SemaphoreCondition {

    Semaphore semaphore;
    BlockingLock blockingLock;
    int nrThreads;

    SemaphoreCondition(BlockingLock blockingLock, int nrThreads) {
        this.semaphore = new Semaphore(0, nrThreads);
        this.blockingLock = blockingLock;
        this.nrThreads = nrThreads;
    }

    void await() {
        blockingLock.unlock();
        semaphore.p();
        blockingLock.lock();
    }

    void signalAll() {
        for (int i = 0; i < nrThreads; i++) {
            semaphore.v();
        }
    }
}
Because a condition variable is associated with a lock, we also need to modify the BlockingLock:

```java
class BlockingLock {
    Semaphore semaphore;
    int nrThreads;

    BlockingLock(int nrThreads) {
        this.semaphore = new Semaphore(1, nrThreads);
        this.nrThreads = nrThreads;
    }

    SemaphoreCondition newCondition() {
        return new SemaphoreCondition(this, nrThreads);
    }
}
```
Overview

Building Blocks
- SpinningLock
- Thread resume/suspend
Overview

Building Blocks

- SpinningLock
- Thread resume/suspend

Semaphore

- SpinningLock
- Thread resume/suspend
Overview

Building Blocks

- SpinningLock
- Thread resume/suspend

Semaphore

- SpinningLock
- Thread resume/suspend

BlockingLock

- Semaphore(1)
Overview

Building Blocks

- SpinningLock
- Thread resume/suspend

Semaphore

- SpinningLock
- Thread resume/suspend

BlockingLock

- Semaphore(1)
- Condition

Condition

- Semaphore(0)
- BlockingLock
Buffer: implementation

class Buffer {

    private byte[] elements;
    private int nrElements;

    Buffer(int maxNrElements) {
        this.elements = new byte[maxNrElements];
        this.nrElements = 0;
    }

    void add(byte b) {
        if (isFull()) {
            throw new Error("The buffer is full");
        }
        elements[nrElements] = b;
        nrElements++;
    }

    byte remove() {
        if (isEmpty()) {
            throw new Error("The buffer is empty");
        }
        byte element = elements[nrElements - 1];
        nrElements--;
        return element;
    }
}
Buffer: implementation, continued

class Buffer {

    private byte[] elements;
    private int nrElements;

    int nrElements() {
        return nrElements;
    }

    boolean isFull() {
        return nrElements == elements.length;
    }

    boolean isEmpty() {
        return nrElements == 0;
    }

    int capacity() {
        return elements.length;
    }
}

Producer

Pseudo code:

shared buffer

produce() {
    while (true) {
        lock()
        while buffer full
            wait()
        add element to buffer
        signal other threads
        unlock()
    }
}
Producer, real implementation

```java
class Producer extends Thread {

    BlockingLock blockingLock;
    SemaphoreCondition isFullCondition;
    SemaphoreCondition isEmptyCondition;

    Buffer buffer;

    Producer(Buffer buffer, BlockingLock blockingLock, 
              SemaphoreCondition isFullCondition, 
              SemaphoreCondition isEmptyCondition) {

        this.buffer = buffer;
        this.blockingLock = blockingLock;
        this.isFullCondition = isFullCondition;
        this.isEmptyCondition = isEmptyCondition;
    }
```
Producer, real implementation, continued

class Producer extends Thread {

    public void run() {
        while (true) {
            blockingLock.lock();
            while (buffer.isFull()) {
                isFullCondition.await();
            }

            buffer.add(produce());

            if (buffer.nrElements() == 1) {
                isEmptyCondition.signalAll();
            }

            blockingLock.unlock();
        }
    }

    private byte produce() {
        return (byte) 0;
    }
}
class Consumer extends Thread {

    public void run() {
        while (true) {
            blockingLock.lock();
            while (buffer.isEmpty()) {
                isEmptyCondition.await();
            }
            consume(buffer.remove());
            if (buffer.nrElements() == buffer.capacity() - 1) {
                isFullCondition.signalAll();
            }
            blockingLock.unlock();
        }
    }

    private void consume(byte b) {
    }
}
Semaphores

Building blocks

- general
- flexible

Examples

- blocking lock
- condition variables
- barriers
- multi lock (multiple threads acquire critical section)

For more information: Allen B. Downey, “The Little Book of Semaphores”
I am certain that the program contains a deadlock, but I am not able to reproduce it...

Any ideas?
Producer, implementation

class Producer extends Thread {

    public void run() {
        while (true) {
            blockingLock.lock();
            while (buffer.isFull()) {
                isFullCondition.await();
            }

            buffer.add(produce());

            if (buffer.nrElements() == 1) {
                isEmptyCondition.signalAll();
            }

            blockingLock.unlock();
        }
    }

    private byte produce() {
        return (byte) 0;
    }
}
SemaphoreCondition, implementation

class SemaphoreCondition {

    Semaphore semaphore;
    BlockingLock blockingLock;
    int nrThreads;

    SemaphoreCondition(BlockingLock blockingLock, int nrThreads) {
        this.semaphore = new Semaphore(0, nrThreads);
        this.blockingLock = blockingLock;
        this.nrThreads = nrThreads;
    }

    void await() {
        blockingLock.unlock();
        semaphore.p();
        blockingLock.lock();
    }

    void signalAll() {
        for (int i = 0; i < nrThreads; i++) {
            semaphore.v();
        }
    }
}

Semaphore, implementation

class Semaphore {

    FilterLock filterLock;
    int capacity;
    int counter;

    Semaphore(int capacity, int numberThreads) {
        this.filterLock = new FilterLock(numberThreads);
        this.counter = capacity;
    }

    void p() {
        try {
            filterLock.lock();
            while (counter == 0) {
                filterLock.unlock();
                ThreadManager.suspendThread();
                filterLock.lock();
            }

            counter -= 1;
        }
        finally {
            filterLock.unlock();
        }
    }
}

Semaphore, implementation, continued

class Semaphore {

    FilterLock filterLock;
    int capacity;
    int counter;

    void v() {
        try {
            filterLock.lock();
            counter++;
        }
        finally {
            filterLock.unlock();
        }
        ThreadManager.resumeThreads();
    }
}
Schedule
producer: blockingLock.lock()
Schedule

producer: blockingLock.lock()
  semaphore.p();
producer: blockingLock.lock()
    semaphore.p();
    filterLock.lock();
producer: blockingLock.lock()
    semaphore.p();
    filterLock.lock();
    while (counter == 0) \(\rightarrow\) false

consumer: blockingLock.lock()
    semaphore.p();
    filterLock.lock();
    while (counter == 0) \(\rightarrow\) true
(is about to execute suspendThread(),
but becomes really slow)
producer: blockingLock.lock()
    semaphore.p();
    filterLock.lock();
    while (counter == 0) -> false
    counter -= 1;

(is about to execute suspendThread(),
but becomes really slow)
producer: blockingLock.lock()
    semaphore.p();
    filterLock.lock();
    while (counter == 0) -> false
    counter -= 1;
    filterLock.unlock();
Schedule

producer: blockingLock.lock()
    semaphore.p();
    filterLock.lock();
    while (counter == 0) -> false
    counter -= 1;
    filterLock.unlock();
    while (buffer.isFull()) -> false

Schedule

producer: blockingLock.lock()
    semaphore.p();
    filterLock.lock();
    while (counter == 0) -> false
    counter -= 1;
    filterLock.unlock();
    while (buffer.isFull()) -> false
    buffer.add(produce());
Schedule

producer: blockingLock.lock()
   semaphore.p();
   filterLock.lock();
   while (counter == 0) -> false
   counter -= 1;
   filterLock.unlock();
   while (buffer.isFull()) -> false
   buffer.add(produce());

consumer: blockingLock.lock()
producer: blockingLock.lock()
    semaphore.p();
    filterLock.lock();
    while (counter == 0) -> false
    counter -= 1;
    filterLock.unlock();
    while (buffer.isFull()) -> false
    buffer.add(produce());

consumer: blockingLock.lock()
    semaphore.p();
Schedule

producer: blockingLock.lock()
    semaphore.p();
    filterLock.lock();
    while (counter == 0) -> false
    counter -= 1;
    filterLock.unlock();
    while (buffer.isFull()) -> false
    buffer.add(produce());

    consumer: blockingLock.lock()
        semaphore.p();
        filterLock.lock();
Schedule

producer: blockingLock.lock()
    semaphore.p();
    filterLock.lock();
    while (counter == 0) -> false
    counter -= 1;
    filterLock.unlock();
    while (buffer.isFull()) -> false
    buffer.add(produce());

consumer: blockingLock.lock()
    semaphore.p();
    filterLock.lock();
    while (counter == 0) -> true
producer: blockingLock.lock()
    semaphore.p();
    filterLock.lock();
    while (counter == 0) -> false
    counter -= 1;
    filterLock.unlock();
    while (buffer.isFull()) -> false
    buffer.add(produce());

consumer: blockingLock.lock()
    semaphore.p();
    filterLock.lock();
    while (counter == 0) -> true
    filterLock.unlock();
Schedule

producer: blockingLock.lock()
    semaphore.p();
    filterLock.lock();
    while (counter == 0) -> false
    counter -= 1;
    filterLock.unlock();
    while (buffer.isFull()) -> false
    buffer.add(produce());

c consumer: blockingLock.lock()
    semaphore.p();
    filterLock.lock();
    while (counter == 0) -> true
    filterLock.unlock();
    (is about to execute suspendThread(),
    but becomes really slow)
producer: if (buffer.nrElements() == 1) -> true
isEmptyCondition.signalAll()
semaphore.v(); // 2 times
filterLock.lock();
counter++;
filterLock.unlock();
ThreadManager.resumeThreads();
blockingLock.unlock();
semaphore.v();
filterLock.lock();
counter++;
filterLock.unlock();
ThreadManager.resumeThreads();
blockingLock.lock();
semaphore.p();
filterLock.lock();
while (counter == 0) -> false
counter -= 1;
filterLock.unlock();
while (buffer.isFull()) -> true
isFullCondition.await(); // goes to sleep
producer: if (buffer.nrElements() == 1) -> true
Schedule, continued

producer: if (buffer.nrElements() == 1) -> true
    isEmptyCondition.signalAll()
Schedule, continued

producer: if (buffer.nrElements() == 1) -> true
    isEmptyCondition.signalAll()
    semaphore.v();  // 2 times
Schedule, continued

producer: if (buffer.nrElements() == 1) -> true
    isEmptyCondition.signalAll()
    semaphore.v(); // 2 times
    filterLock.lock();
    counter++;
    filterLock.unlock();
    ThreadManager.resumeThreads();
    blockingLock.unlock();
    semaphore.p();
    filterLock.lock();
    while (counter == 0) -> false
        counter -= 1;
    filterLock.unlock();
    while (buffer.isFull()) -> true
        isFullCondition.await(); // goes to sleep
Schedule, continued

producer: if (buffer.nrElements() == 1) -> true
    isEmptyCondition.signalAll()
    semaphore.v();  // 2 times
    filterLock.lock();
    counter++;

while (counter == 0) -> false
    counter -= 1;
    filterLock.unlock();

while (buffer.isFull()) -> true
    isFullCondition.await(); // goes to sleep
Schedule, continued

producer: if (buffer.nrElements() == 1) -> true
    isEmptyCondition.signalAll()
    semaphore.v(); // 2 times
    filterLock.lock();
    counter++;
    filterLock.unlock();
    ThreadManager.resumeThreads();

    while (counter == 0) -> false
        counter -= 1;
        filterLock.unlock();
        while (buffer.isFull()) -> true
            isFullCondition.await(); // goes to sleep
            blockingLock.lock();
            semaphore.p();
            filterLock.lock();
            ThreadManager.resumeThreads();
    }
producer: if (buffer.nrElements() == 1) -> true
  isEmptyCondition.signalAll()
  semaphore.v(); // 2 times
  filterLock.lock();
  counter++;
  filterLock.unlock();
  ThreadManager.resumeThreads();
producer: if (buffer.nrElements() == 1) -> true
    isEmptyCondition.signalAll()
    semaphore.v();  // 2 times
    filterLock.lock();
    counter++;
    filterLock.unlock();
    ThreadManager.resumeThreads();
    blockingLock.unlock();
producer: if (buffer.nrElements() == 1) -> true
    isEmptyCondition.signalAll()
    semaphore.v(); // 2 times
    filterLock.lock();
    counter++;
    filterLock.unlock();
    ThreadManager.resumeThreads();
    blockingLock.unlock();
    semaphore.v();

producer: if (buffer.nrElements() == 1) -> true
    isEmptyCondition.signalAll()
    semaphore.v(); // 2 times
    filterLock.lock();
    counter++;
    filterLock.unlock();
    ThreadManager.resumeThreads();
blockingLock.unlock();
    semaphore.v();
    semaphore.v();
    filterLock.lock();
Schedule, continued

producer: if (buffer.nrElements() == 1) -> true
    isEmptyCondition.signalAll()
    semaphore.v(); // 2 times
    filterLock.lock();
    counter++;
    filterLock.unlock();
    ThreadManager.resumeThreads();
blockingLock.unlock();
semaphore.v();
filterLock.lock();
counter++;
producer: if (buffer.nrElements() == 1) -> true
    isEmptyCondition.signalAll()
    semaphore.v(); // 2 times
    filterLock.lock();
    counter++;
    filterLock.unlock();
    ThreadManager.resumeThreads();
blockingLock.unlock();
    semaphore.v();
    filterLock.lock();
    counter++;
    filterLock.unlock();
producer: if (buffer.nrElements() == 1) -> true
  isEmptyCondition.signalAll()
  semaphore.v(); // 2 times
  filterLock.lock();
  counter++;
  filterLock.unlock();
  ThreadManager.resumeThreads();

blockingLock.unlock();
semaphore.v();
filterLock.lock();
counter++;
filterLock.unlock();
ThreadManager.resumeThreads();
producer: if (buffer.nrElements() == 1) -> true
    isEmptyCondition.signalAll()
    semaphore.v();  // 2 times
    filterLock.lock();
    counter++;
    filterLock.unlock();
    ThreadManager.resumeThreads();
blockingLock.unlock();
semaphore.v();
    filterLock.lock();
    counter++;
    filterLock.unlock();
    ThreadManager.resumeThreads();
blockingLock.lock()
Schedule, continued

```
producer: if (buffer.nrElements() == 1) -> true
    isEmptyCondition.signalAll()
    semaphore.v(); // 2 times
    filterLock.lock();
    counter++;
    filterLock.unlock();
    ThreadManager.resumeThreads();
    blockingLock.unlock();
    semaphore.v();
    filterLock.lock();
    counter++;
    filterLock.unlock();
    ThreadManager.resumeThreads();
    blockingLock.lock()
    semaphore.p();
```
producer: if (buffer.nrElements() == 1) -> true
    isEmptyCondition.signalAll()
    semaphore.v();  // 2 times
    filterLock.lock();
    counter++;
    filterLock.unlock();
    ThreadManager.resumeThreads();
blockingLock.unlock();
    semaphore.v();
    filterLock.lock();
    counter++;
    filterLock.unlock();
    ThreadManager.resumeThreads();
blockingLock.lock()
    semaphore.p();
    filterLock.lock();
Schedule, continued

```java
producer: if (buffer.nrElements() == 1) -> true
  isEmptyCondition.signalAll()
  semaphore.v(); // 2 times
  filterLock.lock();
  counter++;
  filterLock.unlock();
  ThreadManager.resumeThreads();
blockingLock.unlock();
  semaphore.v();
  filterLock.lock();
  counter++;
  filterLock.unlock();
  ThreadManager.resumeThreads();
blockingLock.lock()
  semaphore.p();
  filterLock.lock();
  while (counter == 0) -> false
```
producer: if (buffer.nrElements() == 1) -> true
    isEmptyCondition.signalAll()
    semaphore.v(); // 2 times
    filterLock.lock();
    counter++;
    filterLock.unlock();
    ThreadManager.resumeThreads();

blockingLock.unlock();
semaphore.v();
    filterLock.lock();
    counter++;
    filterLock.unlock();
    ThreadManager.resumeThreads();

blockingLock.lock()
semaphore.p();
    filterLock.lock();
while (counter == 0) -> false
    counter -= 1;
Schedule, continued

producer: if (buffer.nrElements() == 1) -> true
  isEmptyCondition.signalAll()
  semaphore.v(); // 2 times
  filterLock.lock();
  counter++;
  filterLock.unlock();
  ThreadManager.resumeThreads();
blockingLock.unlock();
  semaphore.v();
  filterLock.lock();
  counter++;
  filterLock.unlock();
  ThreadManager.resumeThreads();
blockingLock.lock()
  semaphore.p();
  filterLock.lock();
  while (counter == 0) -> false
  counter -= 1;
  filterLock.unlock();
Schedule, continued

producer: if (buffer.nrElements() == 1) -> true
    isEmptyCondition.signalAll()
    semaphore.v(); // 2 times
    filterLock.lock();
    counter++;
    filterLock.unlock();
    ThreadManager.resumeThreads();
blockingLock.unlock();
    semaphore.v();
    filterLock.lock();
    counter++;
    filterLock.unlock();
    ThreadManager.resumeThreads();
blockingLock.lock()
    semaphore.p();
    filterLock.lock();
    while (counter == 0) -> false
    counter -= 1;
    filterLock.unlock();
    while (buffer.isFull()) -> true
Schedule, continued

producer: if (buffer.nrElements() == 1) -> true
  isEmptyCondition.signalAll()
  semaphore.v(); // 2 times
  filterLock.lock();
  counter++;
  filterLock.unlock();
  ThreadManager.resumeThreads();
blockingLock.unlock();
  semaphore.v();
  filterLock.lock();
  counter++;
  filterLock.unlock();
  ThreadManager.resumeThreads();
blockingLock.lock()
  semaphore.p();
  filterLock.lock();
  while (counter == 0) -> false
  counter -= 1;
  filterLock.unlock();
while (buffer.isFull()) -> true
  isFullCondition.await(); // goes to sleep
What is the real problem?

The time between unlocking and going to sleep.

**consumer**: acquire the lock
- buffer is empty
- unlock the lock
  
  **Become slow now, before going to sleep**

**producer**: acquire the lock
- fill the buffer and send signals
- ...
- buffer is full
- unlock the lock
- go to sleep

**consumer**: go to sleep
Java has language support for unlocking monitors and going to sleep.

`Object.wait()` unlocks the monitor and starts waiting atomically. `Thread.suspend()` is deprecated, because it is deadlock prone in two situations:

```java
synchronized (this) {
    // while condition is true
    Thread.suspend(); // instead of Object.wait();
} // The thread still owns the monitor
```

```java
synchronized (this) {
    // while condition is true
} 
Thread.suspend();
// The thread does not own the monitor anymore,
// but it is unclear when the thread is going to suspend
```
Lesson 5

Let go of all assumptions regarding execution of threads.

Try to find the extreme boundaries of an application, for example:
- nrElements set to 1
- make a thread very slow, another thread very fast
Why does this Java program terminate despite that apparently it shouldn't (and didn't)?

```java
import java.util.*;

class A {
    static Point currentPos = new Point(1,2);
    static class Point {
        int x;
        int y;
        Point(int x, int y) {
            this.x = x;
            this.y = y;
        }
    }
    public static void main(String[] args) {
        new Thread() {
            void f(Point p) {
                synchronized(this) {
                    if (p.x==1 && p.y) {
                        System.out.println(p.x + "p.y");
                        System.exit(1);
                    }
                }
                @Override
                public void run() {
                    while (currentPos == null);
                    while (true)
                        f(currentPos);
                    }.start();
                    while (true)
                        currentPos = new Point(currentPos.x+1, currentPos.y+1);
                }
        }
    }
}
```

Some samples of the output I'm getting:

```
$ java A
145281 145282
```

12 million dollar concurrency bug
class A {
    static Point currentPos = new Point(1,2);

    public static void main(String[] args) {
        new Thread() {
            void f(Point p) {
                if (p.x+1 != p.y) {
                    System.out.println(p.x + " "+p.y);
                    System.exit(1);
                }
            }
            public void run() {
                while (true)
                    f(currentPos);
            }
        }.start();

        while (true)
            currentPos = new Point(currentPos.x+1, currentPos.y+1);
    }
}

class Point {
    int x;
    int y;

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
Output

$ java A
145281 145282
$ java A
141373 141374
$ java A
49251 49252
$ java A
47007 47008
$ java A
47427 47428
$ java A
154800 154801
$ java A
34822 34823
$ java A
127271 127272
$ java A
63650 63651
Suppose you have an application you want to make concurrent.

steps

1. analyze data structures
   - which data belongs logically together

2. determine granularity
   - how can data best be protected

3. reason about concurrency
   - use schedules
Lesson 1

Be very careful with synchronized datastructures

There may be conditional dependencies between operations within the application.

Examples from the Java Collections Framework:

- Vector (sort of deprecated)
- Collections.synchronizedCollection()
  makes every collection a synchronized collection

Example

```java
if (vector.size() < 100) {
    vector.add(element);
}
```
Always re-check the condition when you wake-up from a signal.

- What causes a thread to go to sleep?
- When a thread wakes-up, has the situation changed?
Lesson 3

Almost always use `notifyAll()` or `signalAll()`

`notify()` chooses a thread non-deterministically to continue executing. Common false assumptions (pitfalls) are:
- the right thread is woken up
- a thread that wakes up, immediately acquires the lock

Overhead
In a correct implementation, the overhead is not much, because most threads go to sleep immediately.
Lesson 4

Be careful when calling third party’s code while holding a lock.

Watch ordering of locks closely.
Lesson 5

Let go of all assumptions regarding execution of threads.

Try to find the extreme boundaries of an application, for example:
- nrElements set to 1
- make a thread very slow, another thread very fast