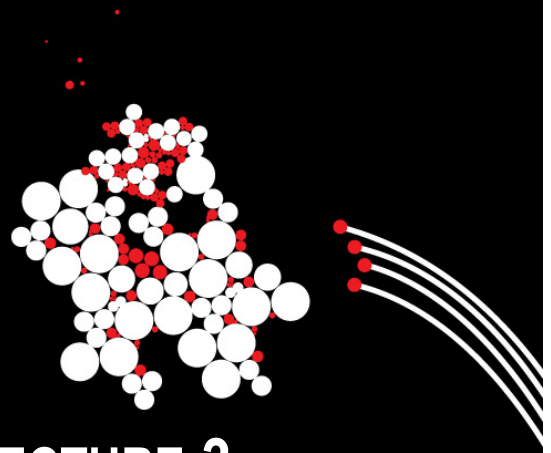
UNIVERSITY OF TWENTE.

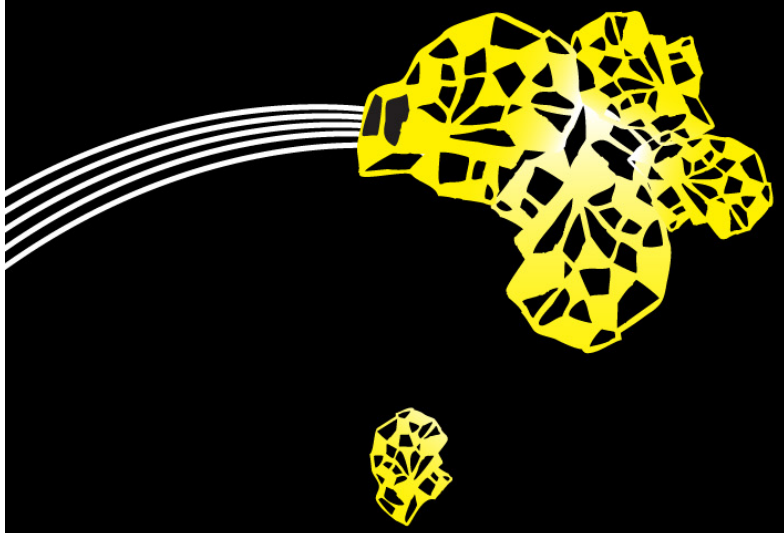


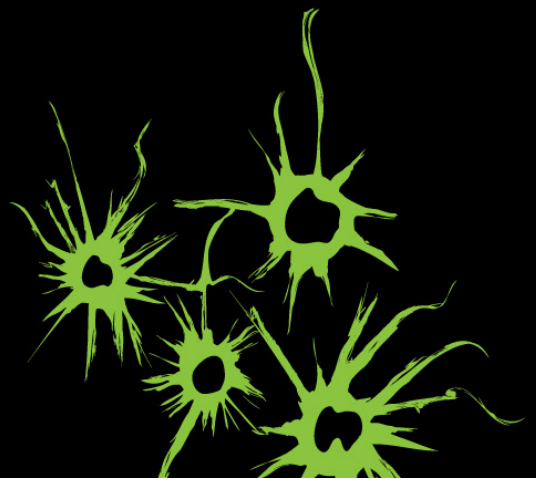
CONCURRENT PROGRAMMING – LECTURE 3

LIVENESS, PERFORMANCE AND FAIRNESS

MODULE 8: PROGRAMMING PARADIGMS

15 MAY 2019

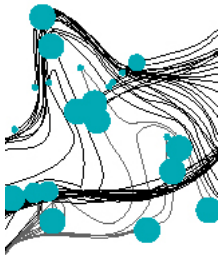




CONCURRENT PROGRAMMING

CONTENTS

2	Tue 30/4	Basics of concurrency, thread safety, testing concurrent systems	Ch. 1-3, 12 (+ SS material)
3	Tue 7/5	Synchronisation	Ch. 4, 5, 13, 14.1-14.4
4	Wed 15/5	Liveness, performance, and fairness	Ch. 10,11
5	Thu 23/5	Homogeneous threading (OpenMP, OpenCL)	Papers
6	Mon 3/6	Safe concurrency (Software Transactional Memory, Rust)	Papers
7	Fri 7/6	Fine-grained concurrency, memory models	Ch. 14.5-6, 15, 16



CONCURRENT PROGRAMMING

LECTURE 3

- Quiz on thread safety and synchronization
- Liveness
- Performance and Scalability
- Fairness

Literature:

JCIP: Chapters 10 & 11

QUESTION 1

```
final public class RGB {
    final private int red; final private int green;
    final private int blue;

    public RGB(int red, int green, int blue, String name) {// body }
    public RGB invert() {
        return new RGB(255 – red, 255 – green, 255 – blue);
    }
}
```

What is **NOT** a property of this class?

a. Immutable
b. Thread-safe
c. Data-race-free
d. Stateless

QUESTION 2

```
class Point {
    private int x; // @guardedby <A>
    private int y; // @guardedby <B>
    public synchronized void move(int newX, int newY) {x = newX; y = newY; }
    public synchronized int getX() { return x; }
    public synchronized int getY() { return y; }
    public synchronized void moveRight() { x = x + 1; }
}
```

What should be filled in at <A>
and <B>

a. <A> = this, <B> = this
b. <A> = Point, <B> = Point
c. <A> = x, <B> = y
d. <A> = x, y, <B> = x, y

QUESTION 3

```
public void enqueue(T x) {
  lock.lock();
  try {
    while (tail - head == items.length)
      {}

    items[tail % items.length]= x;
    tail++;

  } finally {
    lock.unlock();
  }
}
```

Suppose we have a bounded buffer with the following enqueue operation (for concurrent usage). Dequeue operation uses the same lock. What is the problem with this implementation of enqueue?

a. Starvation
b. Lock still held while queue full
c. Wasted CPU cycles while queue full
d. No mutual exclusion

QUESTION 4

```
class AltPoint {
    private final int x; // @guardedby <A>
    private int y; // @guardedby <B>
    public synchronized int getTotal() { return x + y; }
    public synchronized void moveUp() { y = y + 1; }
    public synchronized void jump() { y = x + y; }
    public int getX() { return x; }
}
```

What should be filled in at <A>
and <B>

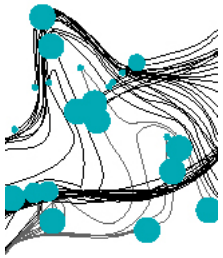
a. <A> = lock, <B> = this
b. <A> = Point, <B> = Point
c. <A> = nothing, <B> = this
d. <A> = nothing, <B> = nothing

QUESTION 5: LOCK COUPLING LIST

```
public void insert(int pos, T val) {
    LockNode<T> current = sentinel;
    LockNode<T> tmp;
    current.lock();
    while (current.next != null && pos > 0) {
        tmp = current; current = current.next; pos--;
        current.lock();
        tmp.unlock();
    }
    tmp = new LockNode<T>(val, current.next);
    current.next = tmp;
    current.unlock();
}
```

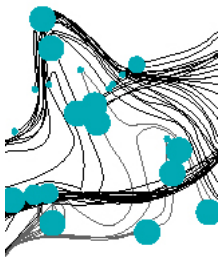
Suppose we swap the two green lines in the code.
What can go wrong?

a. Nothing
b. Data races
c. Dangling pointer
d. Values stored in the list can be corrupted



AVOIDING LIVENESS HAZARDS

CHAPTER 10



CORRECTNESS

SAFETY & LIVENESS

```
public class T extends Thread {
    public void run() {}

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            (new T()).start();
        }
    }
}
```

Is this program thread-safe?

Does it do anything useful?

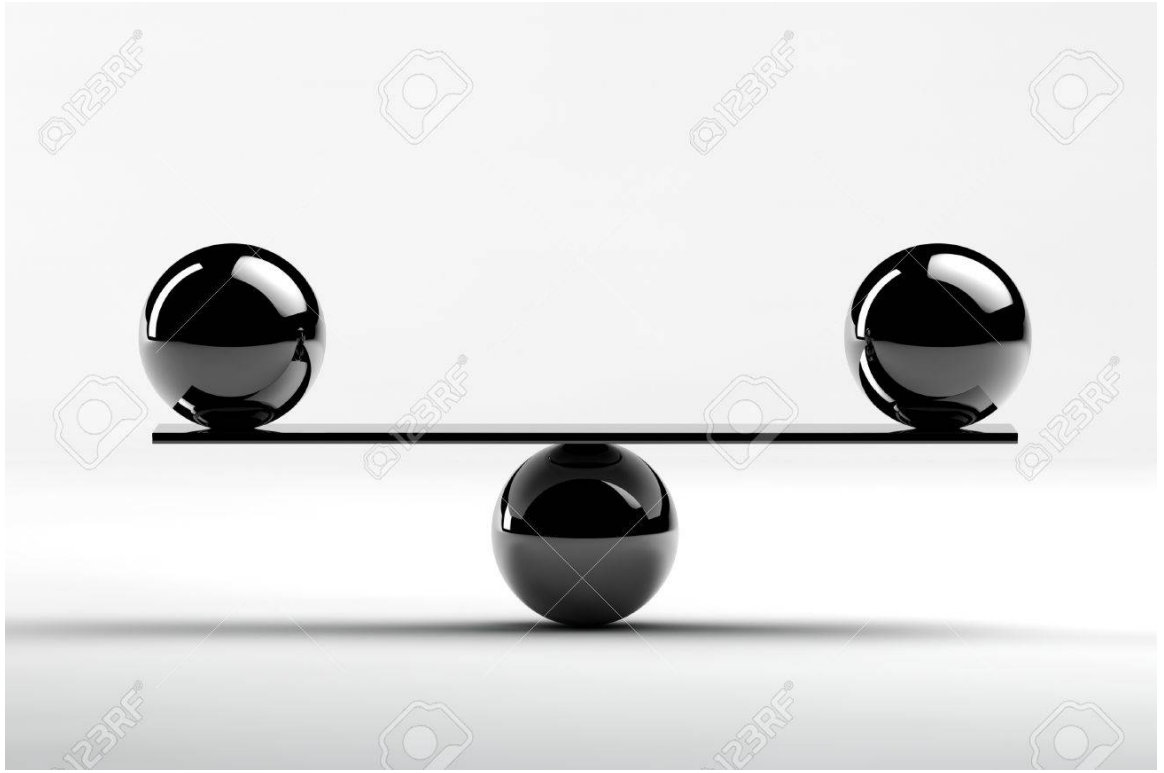
Safety: nothing bad will ever happen!

Liveness: something good will eventually happen!

Leslie Lamport. Proving the correctness of Multiprocess Programs (1977).

SAFETY VS. LIVENESS

Increasing liveness often introduces safety problems.
E.g. fine-grained locking increases the chances of deadlock.



LIVENESS HAZARDS

- Deadlock
- Starvation
- Livelock

DEADLOCK

(IN)FAMOUS DINING PHILOSOPHERS PROBLEM

- Alternate in thinking and eating
- Philosopher needs two forks to eat

What happens if all philosophers take left fork first?

How can deadlock be avoided?

DEADLOCK DEFINITION

A state in which each member of a group is waiting for some other member to take action

- releasing a lock
- sending a message

General solution: break the symmetry

Always use the same lock order!

… or a single lock!

LOCK ORDER

HOW TO SOLVE THIS DEADLOCK?

```
private Object lockX = new Object();
private Object lockY = new Object();
private int x; //@ guardedby lockX
private int y; //@ guardedby lockY
```

```
public void moveX(int n) {
   synchronized (lockX) {
      synchronized (lockY) {
         if (x + n != y) {
            x = x + n;
         }
      }
   }
}
```

```
public void moveY(int n) {
   synchronized (lockY) {
      synchronized (lockX) {
         if (y + n != x) {
            y = y + n;
         }
      }
   }
}
```

DYNAMIC LOCK ORDER

```
class Point {
   Object lock = new Object();
   ...
}
```

```
public void addPoints(Point p1, Point p2) {
   synchronized (p1.lock) {
      synchronized (p2.lock) {
         p1.setX(p1.getX() + p2.getX());
         p1.setY(p1.getY() + p2.getY());
      }
   }
}
```

```
Thread 1: addPoints(p1, p2);
Thread 2: addPoints(p2, p1);
```

dynamic lock order!

How can deadlock be avoided?

fix static order, for example:

- hashcode of objects determines lock order
- tie lock (extra lock) if hashcodes are equal

AVOIDING DEADLOCK CAUSED BY LOCKING

LOCK ORDER

- Single lock

  drawback: performance

- Static lock order

  drawback: cannot always be done statically

- Timed lock attempts

- Use dynamic order (e.g., hash code), if necessary combined with tie lock

COOPERATING OBJECTS

```
public class Hotel {
  private List<Guest> guests = ...
  private String hName = …
  public synchronized
         String getName() {
    return hName;
  }
  public synchronized
         String getGuestList() {
    res = "";
    for (Guest g: guests) {
      res = res + g.getName();
    }
    return res;
  }
}
```

- lock on Hotel
- lock on Guest

```
public class Guest {
  private Hotel accomodation = …
  private String gName = …
  public synchronized
         String getName() {
    return gName;
  }
  public synchronized
         String travInfo() {
    return gName + "stays in" +
        accomodation.getName();
  }
}
```

- lock on Guest
- lock on Hotel

ALIEN CALLS

General rule: invoking alien methods while holding lock is dangerous

- Alien call: call to any method whose behaviour is not fully specified in the class
  - Method in other classes
  - Overrideable method

- Strive for open calls: calling a method with no locks held
- Local (and minimal) locking
- Increases well-behavedness and composability of classes
- Makes liveness analysis easier

```
public class Test {

    public static void main(String[] args) throws InterruptedException {
        PrintChar a = new PrintChar('a');
        PrintChar b = new PrintChar('b');
        Thread ta = new Thread(a);
        Thread tb = new Thread(b);
        ta.start();
        tb.start();
    }
}

class PrintChar implements Runnable {
    final Object o = new Object();
    char ch;
    public PrintChar(char a) {
        ch = a;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            synchronized (o) {
                System.out.print(ch);
                try {
                    o.wait();
                    o.notifyAll();
                } catch (InterruptedException ex) {
                }
            }
        }
    }
}
```

From Stackoverflow:
I want to run two threads one after the other, without using sleep() or locks, but a deadlock happens! What's wrong with my code?

Problems:

- Each thread own lock
- Wait before notify

Note: wait should be called within the synchronized block!

RESOURCE DEADLOCK EXAMPLE

Circular resource dependency

- A
  - holds connection to database D1
  - waits for connection to database D2
- B
  - holds connection to database D2
  - waits for connection to database D1

WAIT-NOTIFY DEADLOCK

- All threads that could reach a notify are waiting
- Missing notify(All)

General rule:

- Notify before waiting
- Design your system carefully

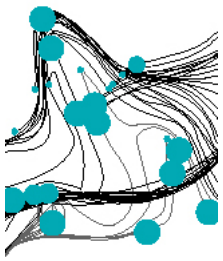
THREAD DUMPS

Exercise 1

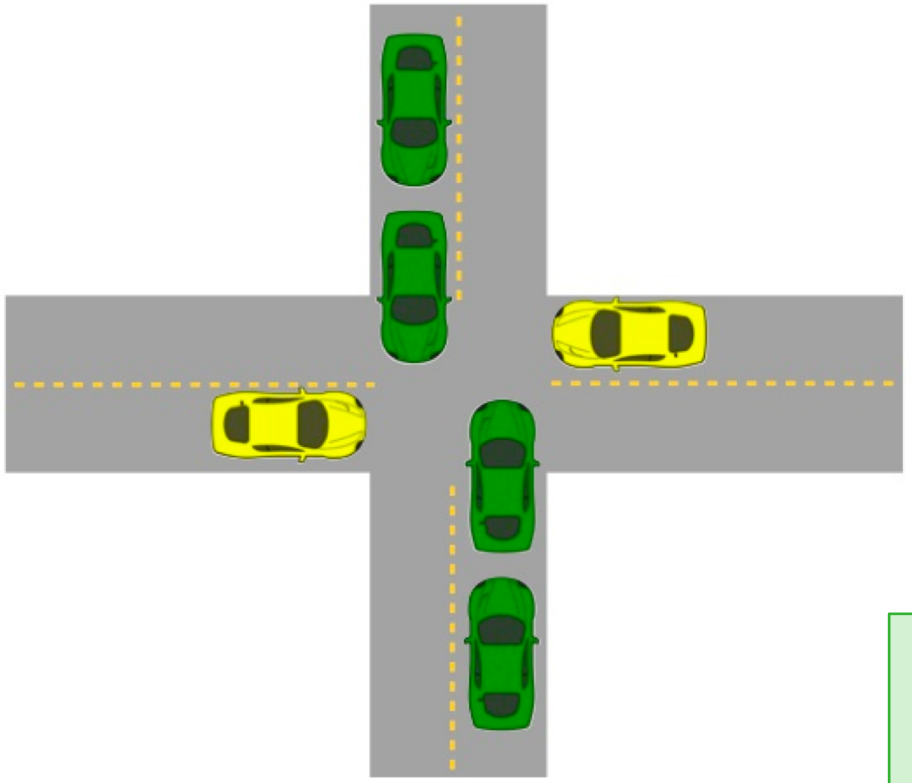
```
"Thread-0" #9 prio=5 os_prio=31 tid=0x00007fe94206d800 nid=0x4b03 waiting for monitor entry [0x000]
  java.lang.Thread.State: BLOCKED (on object monitor)
    at pp.block3.cp.threaddumps.LeftRightDeadlock.rightLeft(LeftRightDeadlock.java:26)
    - waiting to lock <0x000000076ab76168> (a java.lang.Object)
    - locked <0x000000076ab76178> (a java.lang.Object)
    at pp.block3.cp.threaddumps.LeftRightDeadlockTester$Task.run(LeftRightDeadlockTester.java:37)
    at java.lang.Thread.run(Thread.java:745)

...

Found one Java-level deadlock:
=============================
"Thread-1":
    waiting to lock monitor 0x00007fe94205b6a8 (object 0x000000076ab76178, a java.lang.Object),
    which is held by "Thread-0"
"Thread-0":
    waiting to lock monit
    which is held by "Thr

Java stack information fo
=========================
"Thread-1":
    at pp.block3.cp.threa
    - waiting to lock <0x000000076ab76178> (a java.lang.Object)
    - locked <0x000000076ab76168> (a java.lang.Object)
    at pp.block3.cp.threaddumps.LeftRightDeadlockTester$Task.run(LeftRightDeadlockTester.java:36)
    at java.lang.Thread.run(Thread.java:745)
"Thread-0":
    at pp.block3.cp.threaddumps.LeftRightDeadlock.rightLeft(LeftRightDeadlock.java:26)
    - waiting to lock <0x000000076ab76168> (a java.lang.Object)
    - locked <0x000000076ab76178> (a java.lang.Object)
    at pp.block3.cp.threaddumps.LeftRightDeadlockTester$Task.run(LeftRightDeadlockTester.java:37)
    at java.lang.Thread.run(Thread.java:745)

Found 1 deadlock.
```

When you think that (a part of your) program) is deadlocked, you can kill the JVM, and the JVM will generate a ‘thread dump’.

And the JVM will even try to point out the deadlock!



STARVATION



Starvation occurs when a thread is perpetually denied access to resources it needs to make progress!

sometimes caused by thread priorities

Avoid (the temptation) to use thread priorities, since they increase platform dependencies and can cause liveness problems.

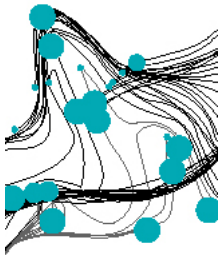
STARVATION EXAMPLE

What happens if instances of ThreadA and ThreadB run in parallel?

```
public class ThreadA {
    private Object lock;
    public ThreadA(Object o) {
        lock = o;
    }

    public void run() {
        synchronized(lock) {
            lock.wait();
        }
    }
}
```

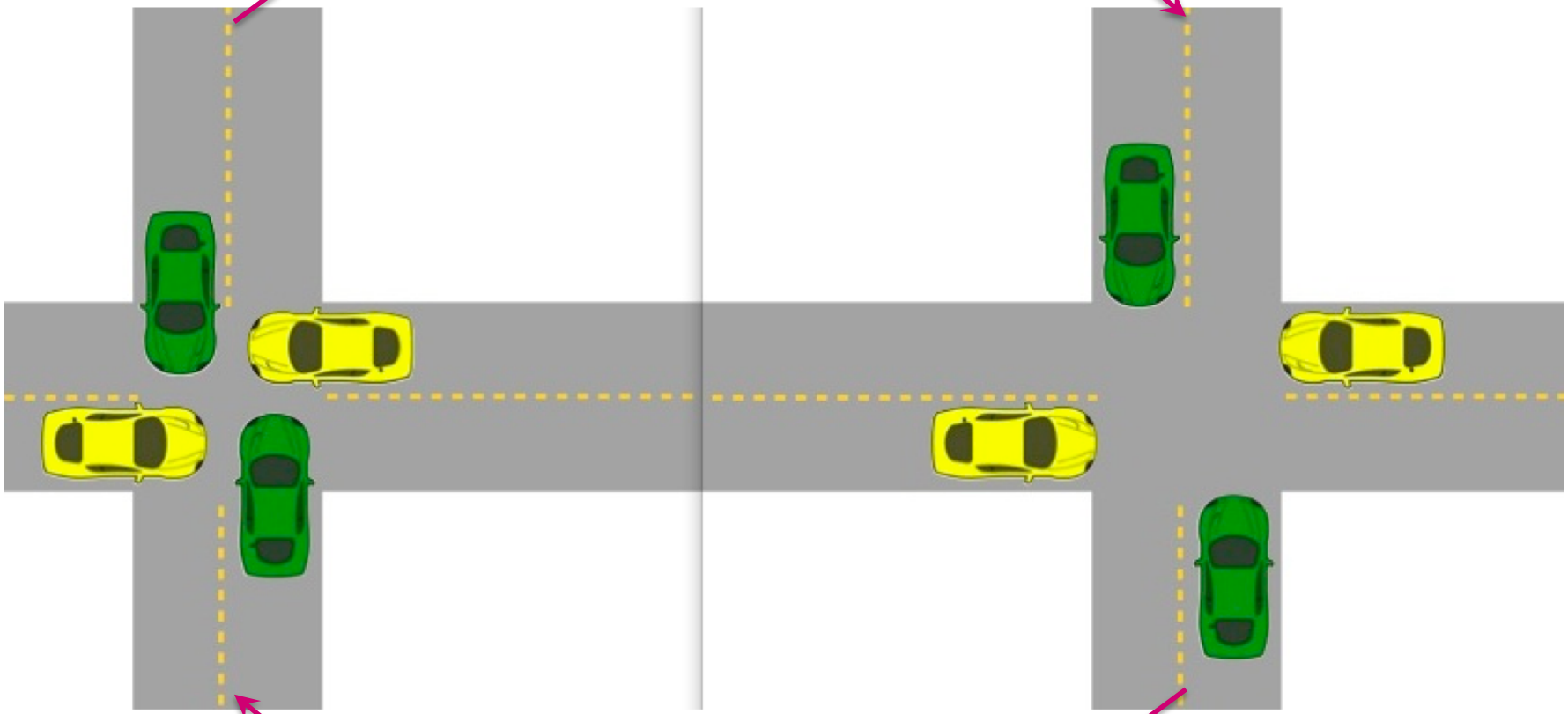
```
public class ThreadB {
    private Object lock;
    public ThreadB(Object o) {
        lock = o;
    }

    public void run() {
        synchronized(lock) {
            lock.notify();
        }
    }
}
```

Notify might come to early!
ThreadA will starve



LIVELOCK

Keep on retrying a failing operation, and never any progress



LIVELOCK DEFINITION

- Processes involved in *livelock* are repeatedly changing their states with regard to each other, but are not making any progress
- Effect similar to a deadlock
- Special case of resource starvation: multiple processes are not progressing

General solution: pick a random thread to continue

LIVELOCK

Husband and wife eat soup with shared spoon
If spouse is hungry, let him/her go first

```
public void eatWith(Spoon spoon, Diner spouse) {
    while (isHungry) {
        // Don't have the spoon, so wait patiently for spouse
        if (spoon.owner != this) {
            // pass time
            continue; }
            // If spouse is hungry, insist upon passing the spoon
        if (spouse.isHungry()) {
            spoon.setOwner(spouse);
            continue; }
        // Spouse wasn't hungry, so finally eat
        spoon.use(); isHungry = false; spoon.setOwner(spouse); }
}
```

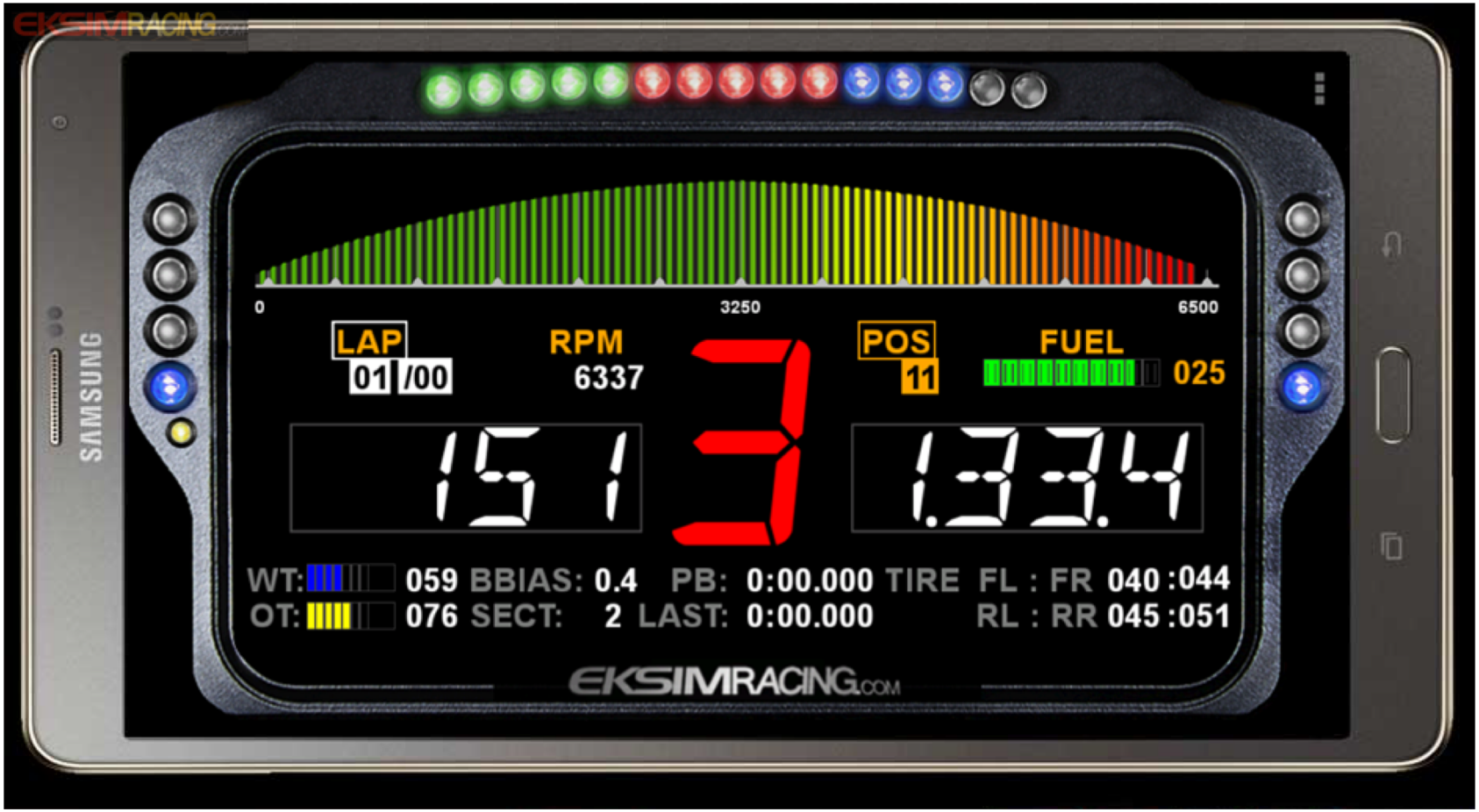
CLASSICAL EXAMPLE: AVOID DEADLOCKS

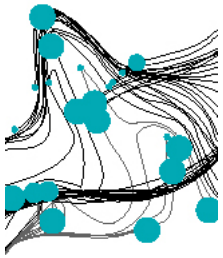
```
// thread 1
getLocks12(lock1, lock2) {
    lock1.lock();
    while (lock2.locked()) {
        // attempt to step aside
        lock1.unlock();
        wait();
        lock1.lock();
    }
    lock2.lock();
}
```

```
// thread 2
getLocks21(lock1, lock2) {
    lock2.lock();
    while (lock1.locked()) {
    // attempt to step aside
        lock2.unlock();
        wait();
        lock2.lock();
    }
    lock1.lock();
}
```

PERFORMANCE AND SCALABILITY

CHAPTER 11





PERFORMANCE

- Increase in performance is often the major reason for using concurrency
- Search for performance also often leads to greater complexity

- Increasing performance means:
  - More work done in same time/using same resources
  - Get job done faster, with less resources
- How much is done is more important question than how fast one thing is done (throughput vs. latency)

Using multiple threads always leads to some overhead

But there is also a theoretical limit

MAXIMAL SPEEDUP

AMDAHL'S LAW

- Given a job, that is executed on N processors. Let $p \in [0, 1]$ be the fraction of the job that can be parallelized (over N processors).
- Let sequential execution of the job take 1 time unit.
- Then parallel execution of the job takes

$$(1 - p) + \frac{p}{N}$$ time units

- So the maximal speedup is

$$\frac{1}{(1 - p) + \frac{p}{N}}$$

Amdahl, Gene (1967). *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*".
AFIPS Conference Proceedings (30): 483–485.

EXAMPLES

numbers of processers

fraction that can be parallelized

$N = 10$

$p = 0.6$ gives speed-up of $\frac{1}{0.4 + \frac{0.6}{10}} = 2.2$

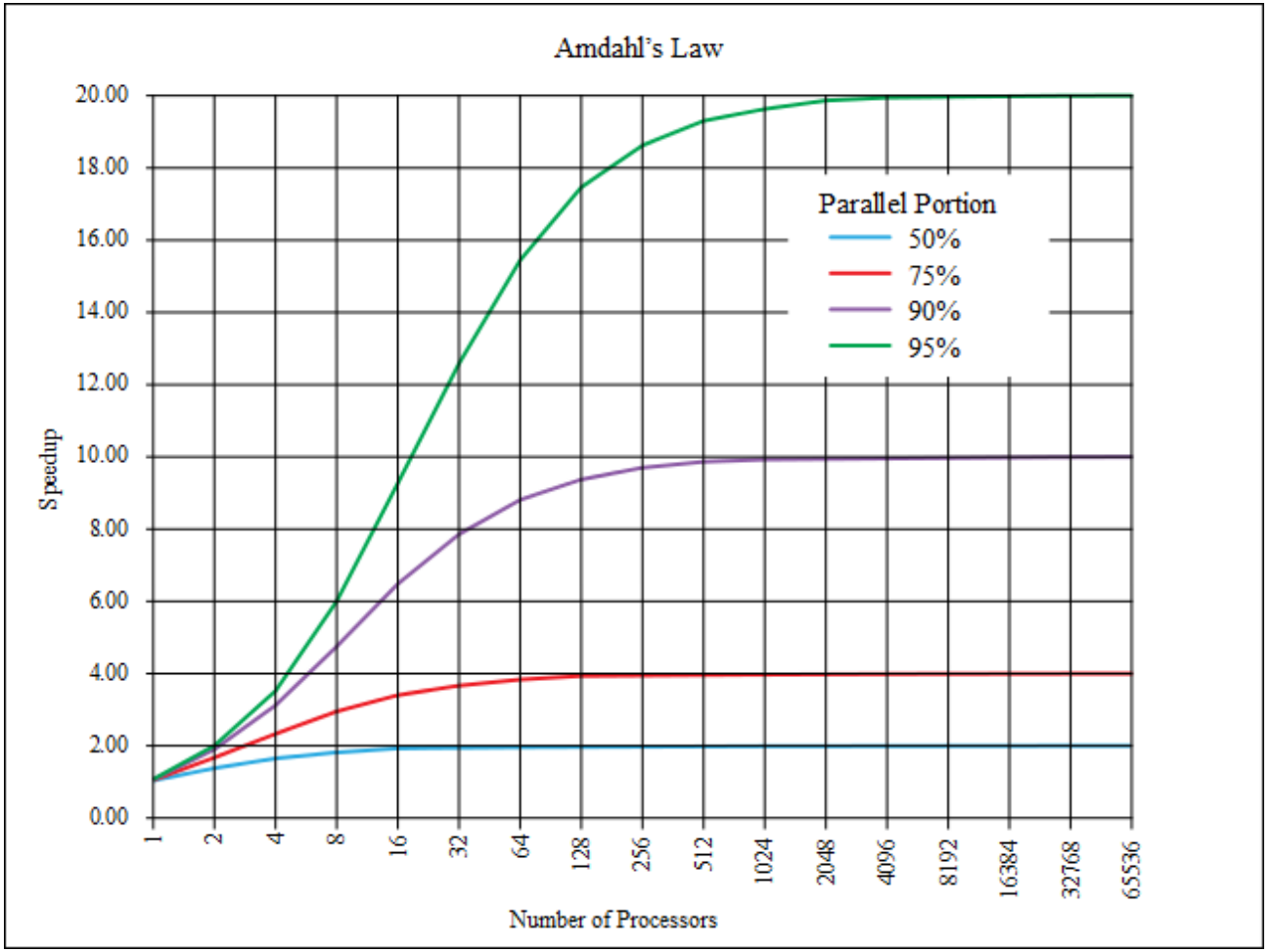
$p = 0.9$ gives speed-up of $\frac{1}{0.1 + \frac{0.9}{10}} = 5.3$

$p = 0.99$ gives speed-up of $\frac{1}{0.01 + \frac{0.99}{10}} = 9.2$

Conclusion: to make efficient use of multiprocessors, it is important to

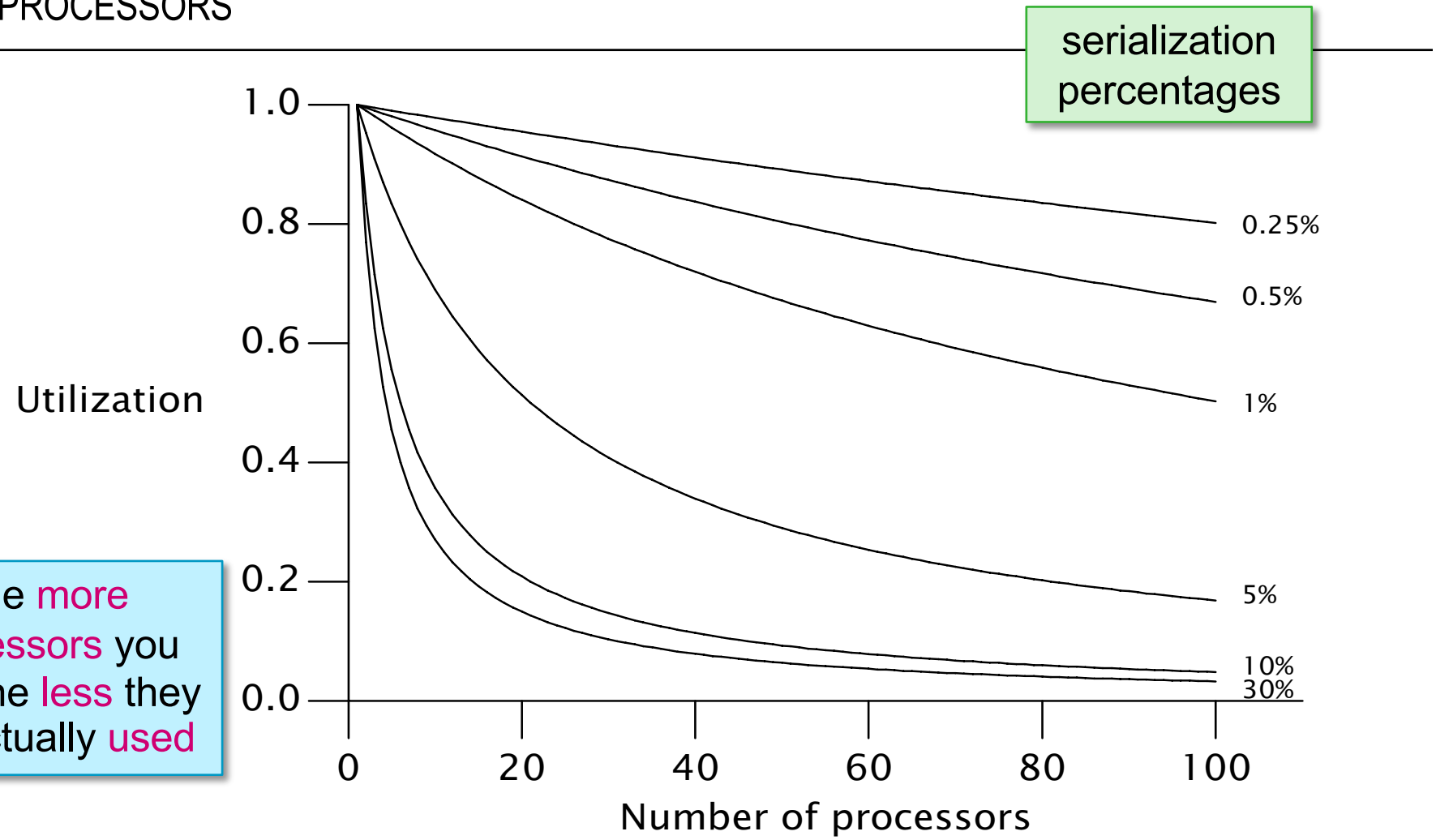
- minimize sequential parts, and
- reduce idle time in which threads wait

MAXIMUM SPEEDUP



UTILIZATION

PROCESSORS



The more processors you add, the less they are actually used

OVERHEAD: CONCURRENCY-INDUCED COSTS

- Context switching
  - saving/restoring execution context
  - more cache misses
- Memory synchronisation
  - To maintain visibility guarantees, additional barriers are necessary
- Blocking when locks are contending
  - Spin-waiting
  - Suspending

Synchronisation optimised for the uncontended case

These costs prevent you from reaching the maximal speedup

PERFORMANCE OPTIMIZATIONS

- Avoid premature optimization.
  - *First make it right, then make it fast!*
- The quest for performance is probably the single greatest source of concurrency bugs!
- Measure, do not guess!
  - Use profiling tools: activity monitor, `vmstat`, `mpstat`
- All concurrent applications have some sources of serialization!

LOCK COARSENING

Do not worry excessively about uncontended synchronization; focus optimization efforts on areas where lock contention actually occurs.

```
public String getNephews() {
    List<String> nephews
        = new Vector<String>();
    nephews.add("Huey");
    nephews.add("Dewey");
    nephews.add("Louie");
    return nephews.toString();
}
```

thread-local, and thus safe (using escape analysis)

4x acquire+release lock

```
public String getNephews() {
    List<String> nephews
        = new Vector<String>();
    synchronized (nephews) {
        nephews.add("Huey");
        nephews.add("Dewey");
        nephews.add("Louie");
        return nephews.toString();
    }
}
```

Smart JVM might perform lock coarsening automatically: the merging of adjacent synchronized blocks.

LOCK SPLITTING

```
public class Point {
  public void synchronized
                  update(int arg) {
    x = x + longComputation(arg);
    y = y + longComputation(arg);
  }
}
```

Only correct when x and y are independent!

```
public class Point {
  public void update(int arg) {
    synchronized(lockX) {
      x = x + longComputation(arg);
    }
    synchronized(lockY) {
      y = y + longComputation(arg);
    }
  }
}
```

REDUCING LOCK CONTENTION

- Reduce lock contention:
  - reduce duration lock held
  - reduce frequency with which locks are requested
  - replace exclusive locks by non-exclusive mechanisms
- Reduce scope of synchronised block:
  - delegate thread safety
  - use thread safe data structures
  - if data independent, use lock splitting

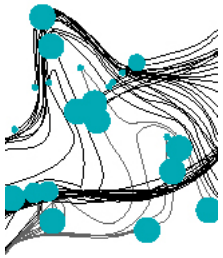
BEWARE OF HOT FIELDS

Every mutable operation needs access to it

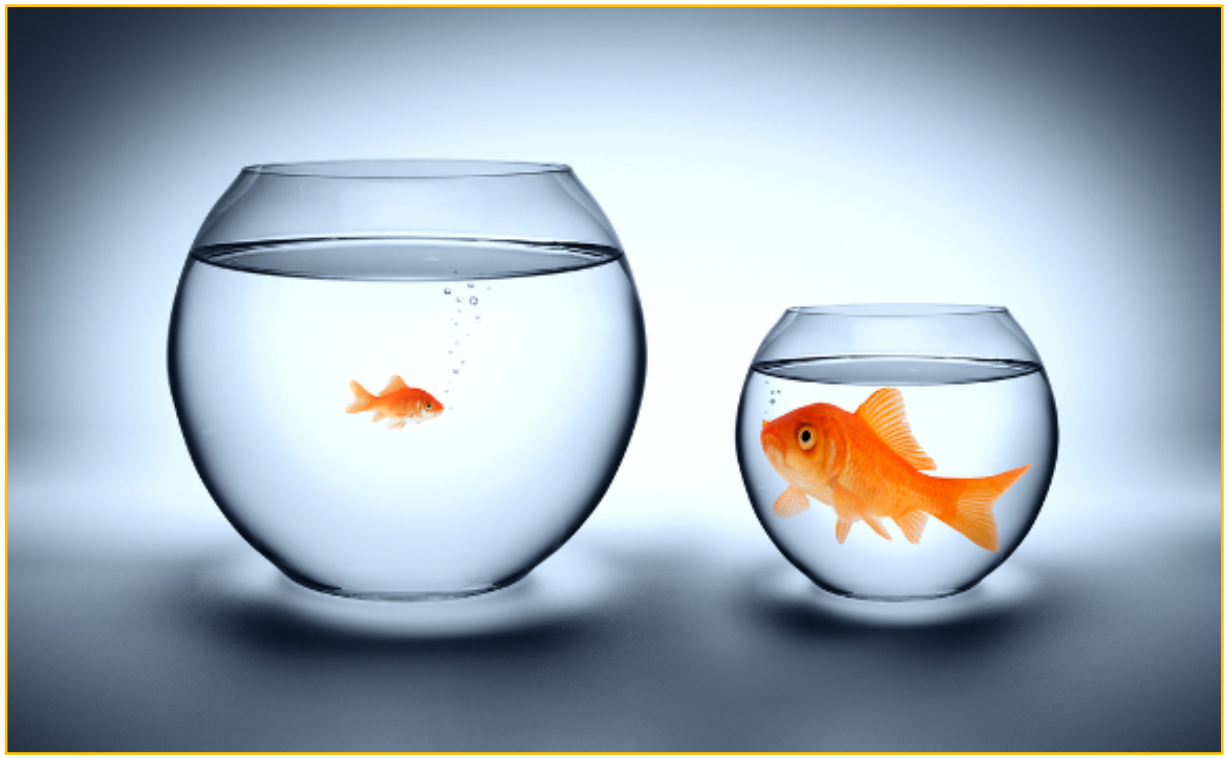
Example: caching size for set as explicit field
(//@ invariant size == this.size();)

- Advantage: quick result for size and isEmpty
- Disadvantage: every update operation needs to update size

Reintroduces scalability problem of exclusive locking



FAIRNESS



THREAD SCHEDULERS

- Scheduler decides which thread will execute when (and for how long)

- Many possible scheduler algorithms
  - Random (mainly theoretical)
  - Preemptive (run until blocked, or until higher priority task comes)
  - Time-slicing
- Selection of next thread
  - Round robin
  - Priority-based
  - FIFO
  - Shortest job first

Some schedulers guarantee fairness

FAIRNESS DEFINITION

- All computation paths must be fair
  - if the machine enters a state infinitely often, it must take every possible transition from that state
- Every enabled execution step must occur eventually in an infinite computation
- Although it may take an unbounded amount of time for the transition to occur

This is sometimes called weak fairness

STRONG FAIRNESS

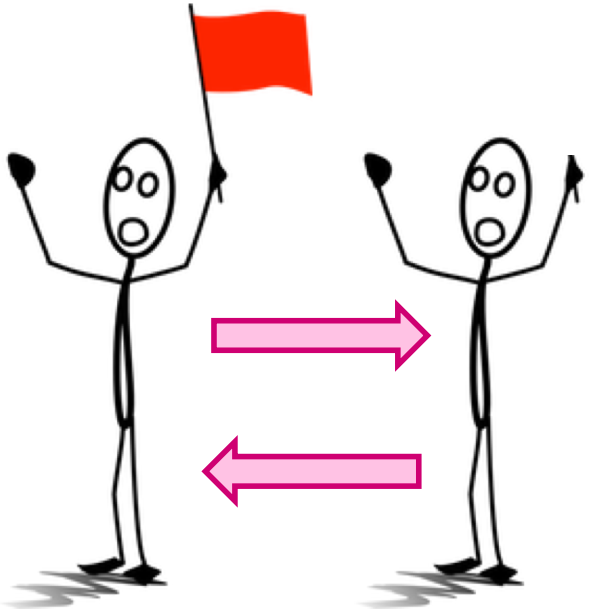
- If a thread is enabled infinitely often, it eventually will execute
- Mainly theoretical notion
- Example:
  - Unreliable channel, where message might be lost
  - Sender continues sending message until receiving acknowledgement
  - Strong fairness guarantees that message will eventually be read
    - There is an infinite number of states where the message may be read

WEAK VS. STRONG FAIRNESS

Weak Fairness

If the flag is raised, and remains raised, other process will react

Strong Fairness



If the flag is raised infinitely often, other process will react

EXAMPLE

Thread 1:

```
while (true) {
    x = x + 1;
}
```

Thread 2:

```
wait_for(even(x)){
    print("YEP");
}
```

Should this always print "YEP"?

Weak fairness: no
Strong fairness: yes

FAIR LOCKS

EXPLICIT LOCKS

- Fair locking: threads are queued in the order that they arrive.
  - has impact on performance.
- Often statistical fairness (weak fairness) sufficient:

  If you ask access, eventually you will get it
- Non-fair (default) locking permits barging
  - Threads requesting the lock can jump ahead of the queue if the lock happens to be available when requested.
- Application-specific whether you need fairness.

FAIR LOCKS IN JAVA

- Create ReentrantLock with fair policy set to true
- If lock is fair, it favors granting access to the longest-waiting thread
- May result in lower overall throughput
- Advantages:
  - Smaller variances in time to obtain locks
  - Guarantees lack of starvation
- Does not guarantee fairness of thread scheduling
- Untimed tryLock method ignores the fairness setting

SUMMARY

- Liveness
  - Deadlock
  - Starvation
  - Livelock
- Performance
  - Amdahl's law: maximal speedup determined by sequential part of the program
  - Synchronisation costs
  - Measure, don't guess!
- Fairness