Parallel Programming
Exercise Session 7

Spring 2025

Schedule

Post-Discussion Ex. 6

Theory Recap

Pre-Discussion Ex. 7

Kahoot

(For group 9: hard wait/notify exam task)

Theory that you have not yet seen in the lectures

Evaluation

Please fill in the evaluation (~5min)

It is anonymous

(you don’t need to be logged in)

It helps us improving the exercise sessions

Post-Discussion Exercise 6

Merge Sort

Discussion of solution

Longest Sequence

Given a sequence of numbers:

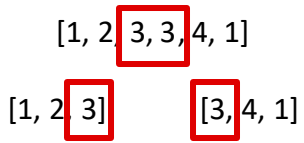
[1, 9, 4, 3, 3, 8, 7, 7, 7, 0]

find the longest sequence of the same consecutive number

Longest Sequence

```
public class LongestSequenceMulti extends RecursiveTask<Sequence> {

  protected Sequence compute() {
    if (// work is small)
      // do the work directly

    else {
      // split work into pieces

      // invoke the pieces and wait for the results

      // check that result is not in between the pieces


      // return the longest result
    }
  }
}
```

Outline almost as before, except:



Longest Sequence

Discussion of solution

Theory Recap

Thread Safe Counter

```
public class Counter {
  private int value;
  // returns a unique value

  public int getNext() {
    return value++;
  }
}
```

How to implement a thread safe Counter?

Thread Safe Counter

```
public class SyncCounter {
  private int value;

  public synchronized int getNext() {
    return value++;
  }
}
```

```
public class AtomicCounter {
  private AtomicInteger value;

  public int getNext() {
    return value.incrementAndGet();
  }
}
```

```
public class LockCounter {
  private int value;
  private Lock = new ReentrantLock();

  public int getNext() {
    lock.lock();
    try {
      return value++;
    } finally {
      lock.unlock()
    }
  }
}
```

How to implement a thread safe Counter?

Thread Safe Counter

```
public class SyncCounter {
  private int value;

  public synchronized int getNext() {
    return value++;
  }
}
```

```
public class AtomicCounter {
  private AtomicInteger value;

  public int getNext() {
    return value.incrementAndGet();
  }
}
```

```
public class LockCounter {
  private int value;
  private Lock = new ReentrantLock();

  public int getNext() {
    lock.lock();
    try {
      return value++;
    } finally {
      lock.unlock()
    }
  }
}
```

What is the difference between synchronized and a Lock?

Java: The **synchronized** keyword

Synchronization is built around an internal entity
known as the intrinsic lock or monitor lock

Every intrinsic lock has an object (or class) associated with it

A thread that needs exclusive access to an object's field has to acquire
the object's intrinsic lock before accessing them

java.util.concurrent.Lock Interface

More low-level primitive than synchronized.

Clients need to implement:

lock(): Acquires the lock, blocks until it is acquired
trylock(): Acquire lock only if its lock is free when function is called
unlock(): Release the lock

Allows more flexible structuring than synchronized blocks

What does it mean to be more flexible?
Why is this useful?

Lock Flexibility

Synchronized forces all lock acquisition and release to occur in a block-structured way

```
synchronized (A) {
  synchronized (B) {
  }
}
```

```
A.lock();
B.lock();
B.unlock();
A.unlock();
```

The following lock order cannot be expressed using synchronized blocks

```
A.lock();
B.lock();
A.unlock();
B.unlock();
```

As we will see later in the course, such order is useful for implementing concurred data structures and referred to as “hand-over-hand” locking (or “chain-locking”)

Lock Flexibility

Consider a list of locks that you should acquire

```
public int getNext(List<Lock> locks) {
            // acquire all locks
            // critical section
            // release all locks
}
```

Can this be achieved using synchronized?

Lock Flexibility

Is the Lock acquired?

```
lock.isLocked()
```

Is the Lock acquired by current thread?

```
lock.isHeldByCurrentThread()
```

Try acquire the Lock without blocking

```
lock.tryLock()
```

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/locks/ReentrantLock.html

Implementing Classes of java.util.concurrent.Lock

ReentrantLock
ReentrantReadWriteLock.ReadLock
ReentrantReadWriteLock.WriteLock

Readers/Writers Lock will be covered in detail in 3 weeks

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/locks/Lock.html

Conclusion: Synchronized vs. External Locks

Synchronized

Acquire / release

Automatic (start / end of the synchronized block)

Scope

Inside the synchronized block

Reentrant

Yes

External locks

Acquire / release

lockobject.lock(), lockobject.unlock()

Scope

From .lock() to .unlock()

Reentrant

Only if you use ReentrantLock class

More flexible compared to synchronized

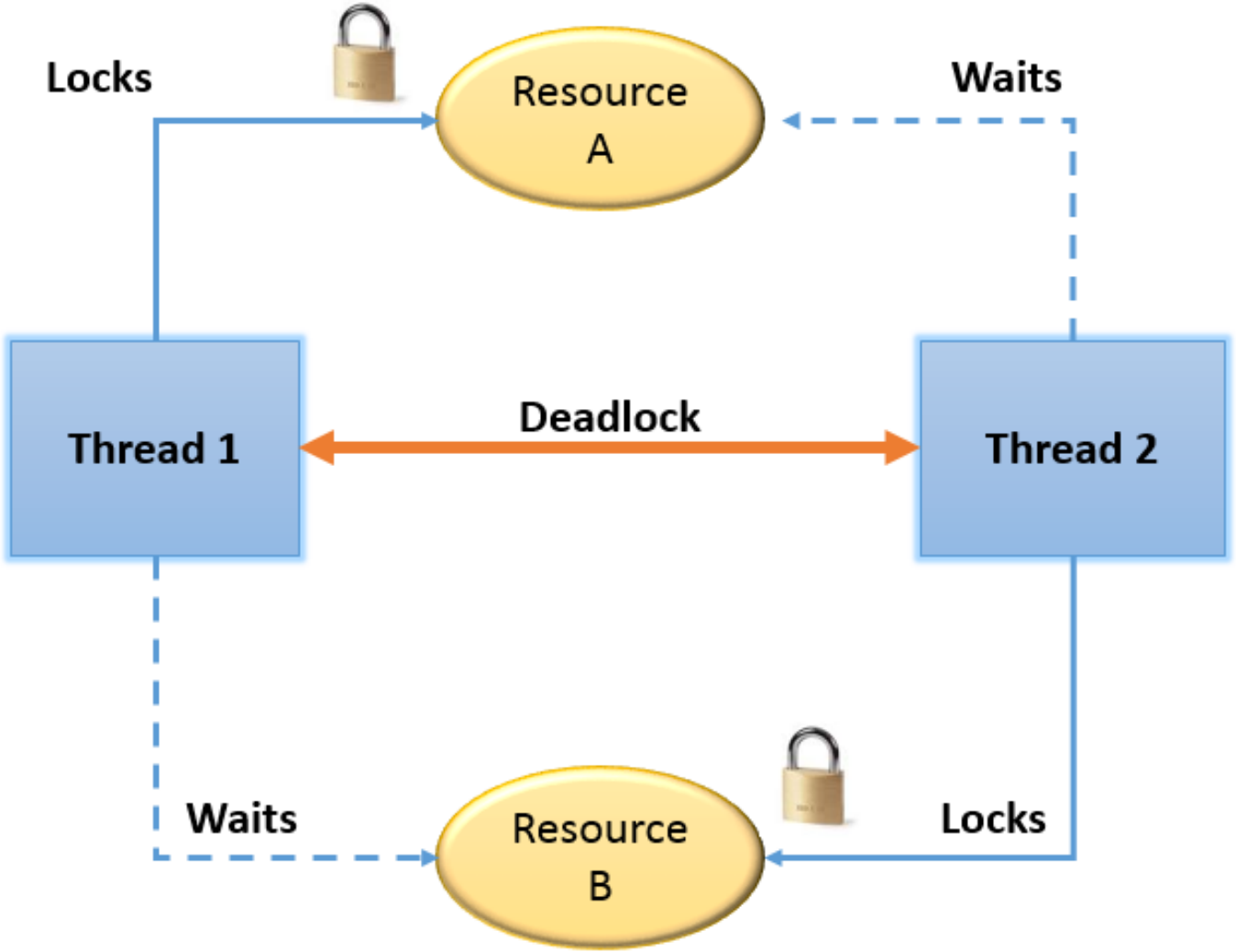
Basic Synchronization Rules

Access to **shared** and **mutable state** needs to be **always protected**!

Synchronization Issues

Data Race: A program has a data race if, during any possible execution, a memory location could be written from one thread, while concurrently being read or written from another thread.

Deadlock: Circular waiting/blocking (no instructions are executed and CPU time may be used) between threads, so that the system (union of all threads) cannot make any progress anymore.


Locks
Resource A
Waits
Thread 1
Deadlock
Thread 2
Waits
Resource B
Locks

Deadlock vs Livelock

Deadlock: Circular waiting/blocking (no instructions are executed and CPU time may be used) between threads, so that the system (union of all threads) cannot make any progress anymore.

Livelock: Livelocks are similar to deadlocks, but in a livelock, the threads are not blocked (they are still executing and consuming resources). However, they are unable to make progress towards completing their tasks due to their interactions with each other. **(all of them change state but no one makes it into CS)**

Quiz: What is wrong with this code?

```
void exchangeSecret(Person a, Person b) {
    a.getLock().lock();
    b.getLock().lock();
    Secret s = a.getSecret();
    b.setSecret(s);
    a.getLock().unlock();
    b.getLock().unlock()
}
```

```
public class Person {
  private ReentrantLock mLock = new ReentrantLock();
  private String mName;

  public ReentrantLock getLock() {
    return mLock;
  }

  ...
}
```

Quiz: What is wrong with this code?

```
void exchangeSecret(Person a, Person b) {
    a.getLock().lock();
    b.getLock().lock();
    Secret s = a.getSecret();
    b.setSecret(s);
    a.getLock().unlock();
    b.getLock().unlock()
}
```

```
public class Person {
  private ReentrantLock mLock = new ReentrantLock();
  private String mName;

  public ReentrantLock getLock() {
    return mLock;
  }

  ...
}
```

Thread 1:
exchangeSecret(p1, p2)

Deadlock

Thread 2:
exchangeSecret(p2, p1)

Possible solution

```
void exchangeSecret(Person a, Person b) {
            ReentrantLock first, second;
            if (a.getName().compareTo(b.getName()) < 0) {
                        first = a.getLock(); second = b.getLock();
            } else if (a.getName().compareTo(b.getName()) > 0) {
                        first = b.getLock(); second = a.getLock();
            } else { throw new UnsupportedOperationException(); }
            first.lock();
            second.lock();
            Secret s = a.getSecret();
            b.setSecret(s);
            first.unlock();
            second.unlock();
}
```

Always acquire and release the Locks in the same order

Deadlocks and Race conditions

Not easy to spot

Hard to debug

- ➔ Might happen only very rarely
- ➔ Testing usually not good enough

  Reasoning about code is required

Lesson learned: Need to be careful when programming with locks

Pre-Discussion Exercise 7

Exercise 7

Banking System

- Multi-Threaded Implementation
- Coding exercise: Use **synchronized** and/or **Locks**
  - Might have to make additions to existing classes
- Reason about Performance
- Reason about Deadlocks
- Run Tests

Multi-threaded Implementation

Task 1 – Problem Identification:

The methods of the classes **Account** and **BankingSystem** must be thread-safe.

You should understand why the current implementation does not work for more than one thread.

Thread-Safe – transferMoney()

Task 2 – Synchronized:

A simple solution to make the *transferMoney()* thread-safe is to use the **synchronized** keyword:
public synchronized boolean transferMoney(…)

Even though the code works as expected, the performance is poor.

The performance of the multi-threaded implementation is worse than the single-threaded. Why does this happen?

Performance of transferMoney()

Task 3 – Locking:

Since the solution with the synchronized keyword does not perform well, you should find a better strategy to achieve the thread-safe implementation.

- *Does your proposed solution work if a transaction happens from and to the same account?*
- *How do you know that your proposed solution does not suffer from deadlocks?*

ThreadSafe - sumAccounts()

Task 4 – Summing Up

With a fine-grained synchronization on the transfer method, the method sumAccounts() may return incorrect results when a transaction takes place at the same time.

- Explain why the current implementation of the sumAccounts() method is not thread-safe any more.
- You should provide a thread-safe implementation.
- Is there any way to parallelize this method?

Testing

You should run the provided tests for your implementation.

If the test succeeds, your code is not necessarily correct.

It is hard to reproduce a bad interleaving.

Old Exam Task (FS 2023)

5. (a) Erklären Sie den Begriff “Deadlock” im Kontext von gegenseitigem Ausschluss mehrerer Threads.

Explain the term ”deadlock” in the context of mutual exclusion in a multi-threaded environment. (2)

(b) Was ist der Unterschied zwischen einem “Deadlock” und einem “Livelock”?

What is the difference between a deadlock and a livelock? (2)

Old Exam Task (FS 2023)

5. (a) Erklären Sie den Begriff "Deadlock" im Kontext von gegenseitigem Ausschluss mehrerer Threads. *Explain the term "deadlock" in the context of mutual exclusion in a multi-threaded environment.* (2)

Solution: A deadlock occurs when no progress can happen in a multi-threaded environment because threads wait for each other's actions.

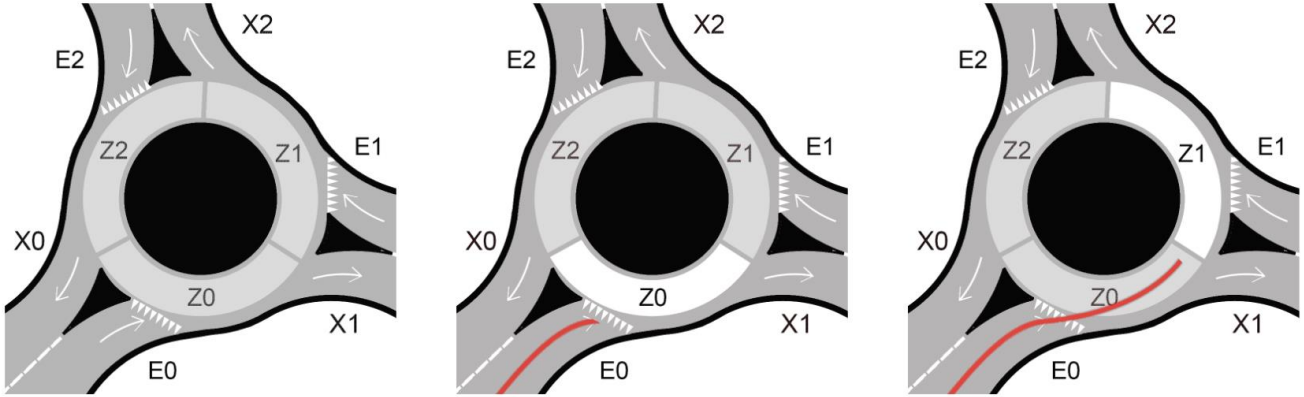
For mentioning no change in state or the idea thereof with other words (1pt). For mentioning the idea of waiting on each other/circular wait (1pt). If an example is provided but no definition is given (1 pt).

(b) Was ist der Unterschied zwischen einem "Deadlock" und einem "Livelock"? *What is the difference between a deadlock and a livelock?* (2)

Solution: In a deadlock the state of the system does not change. In a livelock, the state of the system changes continuously but without progress being made. (1+1pts)


QUIZ

Wait/notify exam task

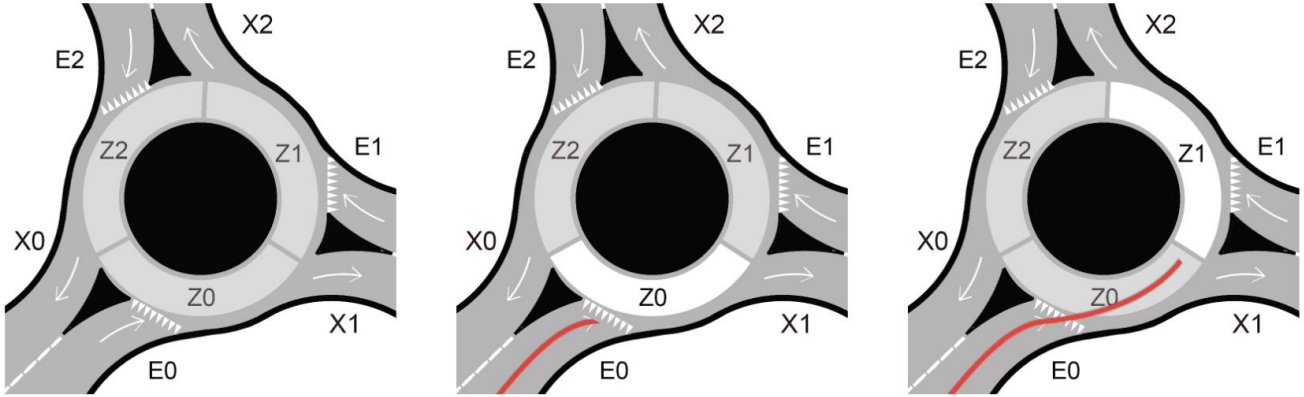


Wir betrachten 3-Wege-Kreisel (siehe Abbildung 1a). Der Kreisverkehr kann in 3 Zonen unterteilt werden: Z0, Z1 und Z2. Die Einfahrten in den Kreisverkehr sind mit E0, E1 und E2 gekennzeichnet. Die Ausfahrten sind mit X0, X1 und X2 gekennzeichnet.

Beispiel: Ein an E0 ankommendes Auto, welches den Kreisel bei Ausfahrt X2 verlassen will, muss zunächst anstehen, bis die zuvor an E0 angekommenen Autos abgefahren sind. Das Auto muss dann warten, bis die Zone Z0 frei ist, um in den Kreisverkehr einfahren zu können (siehe Abbildung 1b). Vor dem Überqueren der Zone Z1 muss das Auto warten, bis Z1 frei ist (siehe Abbildung 1c). Das Auto kann dann bei X2 den Kreisel verlassen. **Um Deadlocks zu vermeiden, dürfen sich im Kreisverkehr nie mehr als 2 Autos gleichzeitig befinden!**

We consider 3-way roundabouts (see Figure 1a). The roundabout can be segmented into 3 zones: Z0, Z1, and Z2. The entries into the roundabout are labeled E0, E1, and E2. The exits are labeled X0, X1, and X2.

Example: A car arriving at E0, intending to leave the roundabout at exit X2, must first wait in line until the cars that have arrived earlier at E0 have left. The car then has to wait for zone Z0 to be free to enter the roundabout (see Figure 1b). Before crossing zone Z1, the car has to wait for Z1 to be free (see Figure 1c). The car can then leave at X2. ***To avoid deadlocks, no more than 2 cars can be in the roundabout at any time!***



Wir betrachten 3-Wege-Kreisel (siehe Abbildung 1a). Der Kreisverkehr kann in 3 Zonen unterteilt werden: Z0, Z1 und Z2. Die Einfahrten in den Kreisverkehr sind mit E0, E1 und E2 gekennzeichnet. Die Ausfahrten sind mit X0, X1 und X2 gekennzeichnet.

Beispiel: Ein an E0 ankommendes Auto, welches den Kreisel bei Ausfahrt X2 verlassen will, muss zunächst anstehen, bis die zuvor an E0 angekommenen Autos abgefahren sind. Das Auto muss dann warten, bis die Zone Z0 frei ist, um in den Kreisverkehr einfahren zu können (siehe Abbildung 1b). Vor dem Überqueren der Zone Z1 muss das Auto warten, bis Z1 frei ist (siehe Abbildung 1c). Das Auto kann dann bei X2 den Kreisel verlassen. **Um Deadlocks zu vermeiden, dürfen sich im Kreisverkehr nie mehr als 2 Autos gleichzeitig befinden!**

We consider 3-way roundabouts (see Figure 1a). The roundabout can be segmented into 3 zones: Z0, Z1, and Z2. The entries into the roundabout are labeled E0, E1, and E2. The exits are labeled X0, X1, and X2.

Example: A car arriving at E0, intending to leave the roundabout at exit X2, must first wait in line until the cars that have arrived earlier at E0 have left. The car then has to wait for zone Z0 to be free to enter the roundabout (see Figure 1b). Before crossing zone Z1, the car has to wait for Z1 to be free (see Figure 1c). The car can then leave at X2. ***To avoid deadlocks, no more than 2 cars can be in the roundabout at any time!***

Warum?

Vervollständigen Sie das Code-Skelett der Klasse `Car` gemäss der Beschreibung in den Kommentaren. Ihre Lösung sollte **notify()**, **notifyAll()**, **wait()**, das **synchronized**-Schlüsselwort und die vorhandenen Locks und Atomic Integers verwenden, um das Programm korrekt zu synchronisieren, aber Sie sollen **keine** unnötige Synchronisation einführen. In dieser Aufgabe müssen Sie **Exceptions nicht behandeln**.

Complete the code skeleton of the class ***`Car`*** *according to the description in the comments. Make sure that your solution is properly synchronized by using* ***notify()***, ***notifyAll()***, ***wait()***, *the* ***synchronized*** *keyword, and the provided locks and atomic integers, where required, but* ***do not*** *introduce unnecessary synchronization. For this task,* ***you do not need to handle exceptions****.*

```java
public class Entry {
    private int index;
    // Die Ticketnummer, die dem Auto gegeben wird,
    // welches sich als nächstes hinten an der Warteschlange anstellt.
    private AtomicInteger next_ticket = new AtomicInteger(initialValue:0);
    // Die Ticketnummer des Autos am Anfang der Warteschlange.
    private AtomicInteger next_car_in_line = new AtomicInteger(initialValue:0);
    Entry(int index) {
        this.index = index;
    }
    public int get_index() {
        return index;
    }
    public int get_ticket() {
        return next_ticket.getAndIncrement();
    }
    public int get_next_car_in_line() {
        return next_car_in_line.get();
    }
    public void car_clears_entry(int car_ticket) {
        int expected_ticket = get_next_car_in_line();
        assert car_ticket == expected_ticket : "Cars must wait their turn.";
        next_car_in_line.incrementAndGet();
    }
}
```

```
class Zone {
    private int index;
    public final Lock mutex = new ReentrantLock();

    Zone(int index) {
        this.index = index;
    }

    public int get_index() {
        return index;
    }
}
```

```java
class Car implements Runnable {
    // this.zones enthält alle Zonen, welche das Auto passieren will, geordnet.
    private final List<Zone> zones = new ArrayList<>();
    private final Entry entry, exit;
    private final int ticket_number;
    private final AtomicInteger n_cars_on_roundabout;
    Car(Zone[] zones, Entry entry, Entry exit, AtomicInteger n_cars_on_roundabout) {
        this.entry = entry;
        this.exit = exit;
        this.ticket_number = entry.get_ticket();
        int zone_index = this.entry.get_index();
        this.n_cars_on_roundabout = n_cars_on_roundabout;
        while (zone_index != this.exit.get_index()) {
            this.zones.add(zones[zone_index]);
            zone_index = (zone_index + 1) % zones.length;
        }
    }
    @Override
    public void run() {
```

```java
@Override
public void run() {
    while (true) {
        synchronized (/* TODO */) {
            if (this.ticket_number > this.entry.get_next_car_in_line()) {
                // Vorrang für Autos, die früher bei derselben Einfahrt ankamen.
            } else {
                break;
            }
        }
    }
    /*
     * Bevor das Auto in den Kreisverkehr einfährt,
     * darf höchstens ein Auto im Kreisverkehr sein.
     */
    while (true){
    }
    Zone current_zone = this.zones.get(index:0);
    // Stellen Sie sicher, dass die entsprechende Zone frei ist...
    // ...und dass das nächste wartende Auto an dieser Einfahrt passieren kann.
    for (int i = 1; i < this.zones.size(); ++i) {
        Zone zone = this.zones.get(i);
        // Stellen Sie sicher, dass die nächste Zone frei ist,
        // und machen Sie die vorherige Zone frei.
        current_zone = zone;
    }
    // Das Auto verlässt den Kreisverkehr
}
```

```java
@Override
public void run() {
    while (true) {
        synchronized (/* TODO */) {
            if (this.ticket_number > this.entry.get_next_car_in_line()) {
                // Vorrang für Autos, die früher bei derselben Einfahrt ankamen.
            } else {
                break;
            }
        }
    }
    /*
     * Bevor das Auto in den Kreisverkehr einfährt,
     * darf höchstens ein Auto im Kreisverkehr sein.
     */
    while (true){
    }
    Zone current_zone = this.zones.get(index:0);
    // Stellen Sie sicher, dass die entsprechende Zone frei ist...
    // ...und dass das nächste wartende Auto an dieser Einfahrt passieren kann.
    for (int i = 1; i < this.zones.size(); ++i) {
        Zone zone = this.zones.get(i);
        // Stellen Sie sicher, dass die nächste Zone frei ist,
        // und machen Sie die vorherige Zone frei.
        current_zone = zone;
    }
    // Das Auto verlässt den Kreisverkehr
}
```

this.entry

entry.wait();

```java
@Override
public void run() {
    while (true) {
        synchronized (/* TODO */) {
            if (this.ticket_number > this.entry.get_next_car_in_line()) {
                // Vorrang für Autos, die früher bei derselben Einfahrt ankamen.
            } else {
                break;
            }
        }
    }
    /*
     * Bevor das Auto in den Kreisverkehr einfährt,
     * darf höchstens ein Auto im Kreisverkehr sein.
     */
    while (true){
    }
    Zone current_zone = this.zones.get(index:0);
    // Stellen Sie sicher, dass die entsprechende Zone frei ist...
    // ...und dass das nächste wartende Auto an dieser Einfahrt passieren kann.
    for (int i = 1; i < this.zones.size(); ++i) {
        Zone zone = this.zones.get(i);
        // Stellen Sie sicher, dass die nächste Zone frei ist,
        // und machen Sie die vorherige Zone frei.
        current_zone = zone;
    }
    // Das Auto verlässt den Kreisverkehr
}
```

this.entry

entry.wait();

int t = n_cars_on_roundabout.get();
if (t < 2 && n_cars_on_roundabout.compareAndSet(t, t + 1)) break;

```java
@Override
public void run() {
    while (true) {
        synchronized (/* TODO */) {
            if (this.ticket_number > this.entry.get_next_car_in_line()) {
                // Vorrang für Autos, die früher bei derselben Einfahrt ankamen.
            } else {
                break;
            }
        }
    }
    /*
     * Bevor das Auto in den Kreisverkehr einfährt,
     * darf höchstens ein Auto im Kreisverkehr sein.
     */
    while (true){
    }
    Zone current_zone = this.zones.get(index:0);
    // Stellen Sie sicher, dass die entsprechende Zone frei ist...
    // ...und dass das nächste wartende Auto an dieser Einfahrt passieren kann.
    for (int i = 1; i < this.zones.size(); ++i) {
        Zone zone = this.zones.get(i);
        // Stellen Sie sicher, dass die nächste Zone frei ist,
        // und machen Sie die vorherige Zone frei.
        current_zone = zone;
    }
    // Das Auto verlässt den Kreisverkehr
}
```

this.entry

entry.wait();

int t = n_cars_on_roundabout.get();
if (t < 2 && n_cars_on_roundabout.compareAndSet(t, t + 1)) break;

current_zone.mutex.lock();

```java
@Override
public void run() {
    while (true) {
        synchronized (/* TODO */) {
            if (this.ticket_number > this.entry.get_next_car_in_line()) {
                // Vorrang für Autos, die früher bei derselben Einfahrt ankamen.
            } else {
                break;
            }
        }
    }
    /*
     * Bevor das Auto in den Kreisverkehr einfährt,
     * darf höchstens ein Auto im Kreisverkehr sein.
     */
    while (true){
    }
    Zone current_zone = this.zones.get(index:0);
    // Stellen Sie sicher, dass die entsprechende Zone frei ist...
    // ...und dass das nächste wartende Auto an dieser Einfahrt passieren kann.
    for (int i = 1; i < this.zones.size(); ++i) {
        Zone zone = this.zones.get(i);
        // Stellen Sie sicher, dass die nächste Zone frei ist,
        // und machen Sie die vorherige Zone frei.
        current_zone = zone;
    }
    // Das Auto verlässt den Kreisverkehr
}
```

this.entry

entry.wait();

int t = n_cars_on_roundabout.get();
if (t < 2 && n_cars_on_roundabout.compareAndSet(t, t + 1)) break;

current_zone.mutex.lock();

synchronize(entry){
entry.car_clears_entry(this.ticket_number);
entry.notifyAll();
}

```java
@Override
public void run() {
    while (true) {
        synchronized (/* TODO */) {
            if (this.ticket_number > this.entry.get_next_car_in_line()) {
                // Vorrang für Autos, die früher bei derselben Einfahrt ankamen.
            } else {
                break;
            }
        }
    }
    /*
     * Bevor das Auto in den Kreisverkehr einfährt,
     * darf höchstens ein Auto im Kreisverkehr sein.
     */
    while (true){
    }
    Zone current_zone = this.zones.get(index:0);
    // Stellen Sie sicher, dass die entsprechende Zone frei ist...
    // ...und dass das nächste wartende Auto an dieser Einfahrt passieren kann.
    for (int i = 1; i < this.zones.size(); ++i) {
        Zone zone = this.zones.get(i);
        // Stellen Sie sicher, dass die nächste Zone frei ist,
        // und machen Sie die vorherige Zone frei.
        current_zone = zone;
    }
    // Das Auto verlässt den Kreisverkehr
}
```

this.entry

entry.wait();

int t = n_cars_on_roundabout.get();
if (t < 2 && n_cars_on_roundabout.compareAndSet(t, t + 1)) break;

current_zone.mutex.lock();

synchronize(entry){
entry.car_clears_entry(this.ticket_number);
entry.notifyAll();
}

zone.mutex.lock();
current_zone.mutex.unlock();

```java
@Override
public void run() {
    while (true) {
        synchronized (/* TODO */) {
            if (this.ticket_number > this.entry.get_next_car_in_line()) {
                // Vorrang für Autos, die früher bei derselben Einfahrt ankamen.
            } else {
                break;
            }
        }
    }
    /*
     * Bevor das Auto in den Kreisverkehr einfährt,
     * darf höchstens ein Auto im Kreisverkehr sein.
     */
    while (true){
    }
    Zone current_zone = this.zones.get(index:0);
    // Stellen Sie sicher, dass die entsprechende Zone frei ist...
    // ...und dass das nächste wartende Auto an dieser Einfahrt passieren kann.
    for (int i = 1; i < this.zones.size(); ++i) {
        Zone zone = this.zones.get(i);
        // Stellen Sie sicher, dass die nächste Zone frei ist,
        // und machen Sie die vorherige Zone frei.
        current_zone = zone;
    }
    // Das Auto verlässt den Kreisverkehr
}
```

this.entry

entry.wait();

int t = n_cars_on_roundabout.get();
if (t < 2 && n_cars_on_roundabout.compareAndSet(t, t + 1)) break;

current_zone.mutex.lock();

zone.mutex.lock();
current_zone.mutex.unlock();

synchronize(entry){
entry.car_clears_entry(this.ticket_number);
entry.notifyAll();
}

current_zone.mutex.unlock();
n_cars_on_roundabout.decrementAndGet();

Not yet seen theory

What will you see soon in the lectures?

- Memory reordering and optimizations
- Orders:
  Program Order, Synchronizes-with, Synchronization Order, Happens-before
- State Space Diagrams
- Dekker‘s Algorithm, Peterson Lock, Filter Lock (generalization of Peterson Lock), Bakery Lock
- Test-and-set (TAS), compare-and-swap (CAS), Test and Test-and-set (TTAS), Exponential Backoff

We will take a look at

- **Memory reordering and optimizations**
- **Orders:**
  Program Order, Synchronizes-with, Synchronization Order, Happens-before
- **State Space Diagrams**
- **Dekker‘s Algorithm, Peterson Lock, Filter Lock (generalization of Peterson Lock),** Bakery Lock
- **Test-and-set (TAS), compare-and-swap (CAS), Test and Test-and-set (TTAS), Exponential Backoff**

Motivation

```
class C {
  private int x = 0;
  private int y = 0;
  Thread 1
    x = 1;
    y = 1;
  }
  Thread 2
    int a = y;
    int b = x;
    assert(b >= a);
  }
}
```

Can this fail?

Another proof

```
class C {
  private int x = 0;
  private int y = 0;
  Thread 1
    x = 1;
    y = 1;
  }
  Thread 2
    int a = y;
    int b = x;
    assert(b >= a);
  }
}
```

There is no interleaving of `f` and `g` causing the assertion to fail

Another proof

```
class C {
  private int x = 0;
  private int y = 0;
  Thread 1
    x = 1;
    y = 1;
  }
  Thread 2
    int a = y;
    int b = x;
    assert(b >= a);
  }
}
```

There is no interleaving of `f` and `g` causing the assertion to fail

Another proof (by contradiction):

Assume `b<a` ⇒ `a==1` and `b==0`.

But if `a==1` ⇒ `y=1` *happened before* `a=y`.
And if `b==0` ⇒ `b=x` *happened before* `x=1`.

Because we assume that programs execute in order:

`a=y` *happened before* `b=x`
`x=1` *happened before* `y=1`

So by transitivity,
`a=y` happened before `b=x` happened before `x=1` happened before `y=1` happened before `a=y` ⇒ Contradiction ↯

But does this really work?

No

Because of:
Optimizations by Compiler
Optimizations by Hardware
(basically Memory Reordering)

Why it still can fail: Memory reordering

Rule of thumb: Compiler and hardware allowed to make changes that do not affect the *semantics* of a *sequentially* executed program

```
void f() {
      x = 1;
      y = x+1;
      z = x+1;
}
```

semantically equivalent?

```
void f() {
      x = 1;
      z = x+1;
      y = x+1;
}
```

semantically equivalent?

```
void f() {
      x = 1;
      z = 2;
      y = 2;
}
```

Are these semantically equivalent?

Example: Fail with self-made rendezvous (C / GCC)

```
int x;

void wait() {
  x = 1;
  while(x==1);
}
```

Assembly without optimization

```
movl   $0x1, x
test:
mov    x, %eax
cmp    $0x1, %eax
je     test
```

```
void arrive(){
  x = 2;
}
```

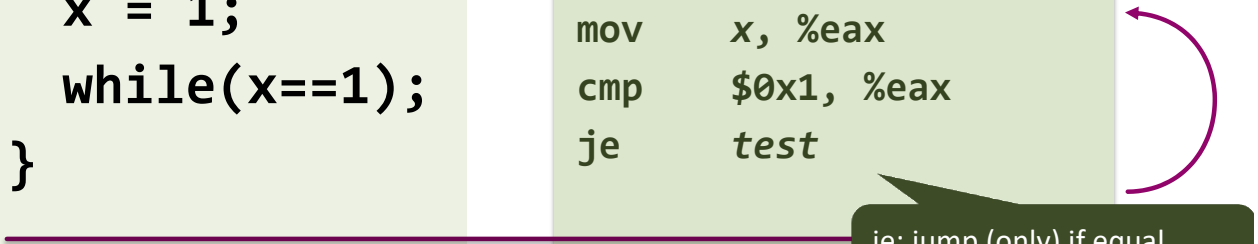
```
movl $0x2, x
```

Example: Fail with self-made rendezvous (C / GCC)

```
int x;

void wait() {
  x = 1;
  while(x==1);
}

void arrive(){
  x = 2;
}
```

Assembly without optimization

```
movl   $0x1, x
test:
mov    x, %eax
cmp    $0x1, %eax
je     test

movl $0x2, x
```



Example: Fail with self-made rendezvous (C / GCC)

```
int x;

void wait() {
  x = 1;
  while(x==1);
}

void arrive(){
  x = 2;
}
```

Assembly without optimization

```
movl   $0x1, x
test:
mov    x, %eax
cmp    $0x1, %eax
je     test

movl $0x2, x
```

je: jump (only) if equal, i.e., if cmp yields true

Assembly with optimization

```
movl   $0x1, x
test:
jmp    test

movl $0x2, x
```

jmp: jump always

Memory hierachy (one core)

0.5ns

1 ns

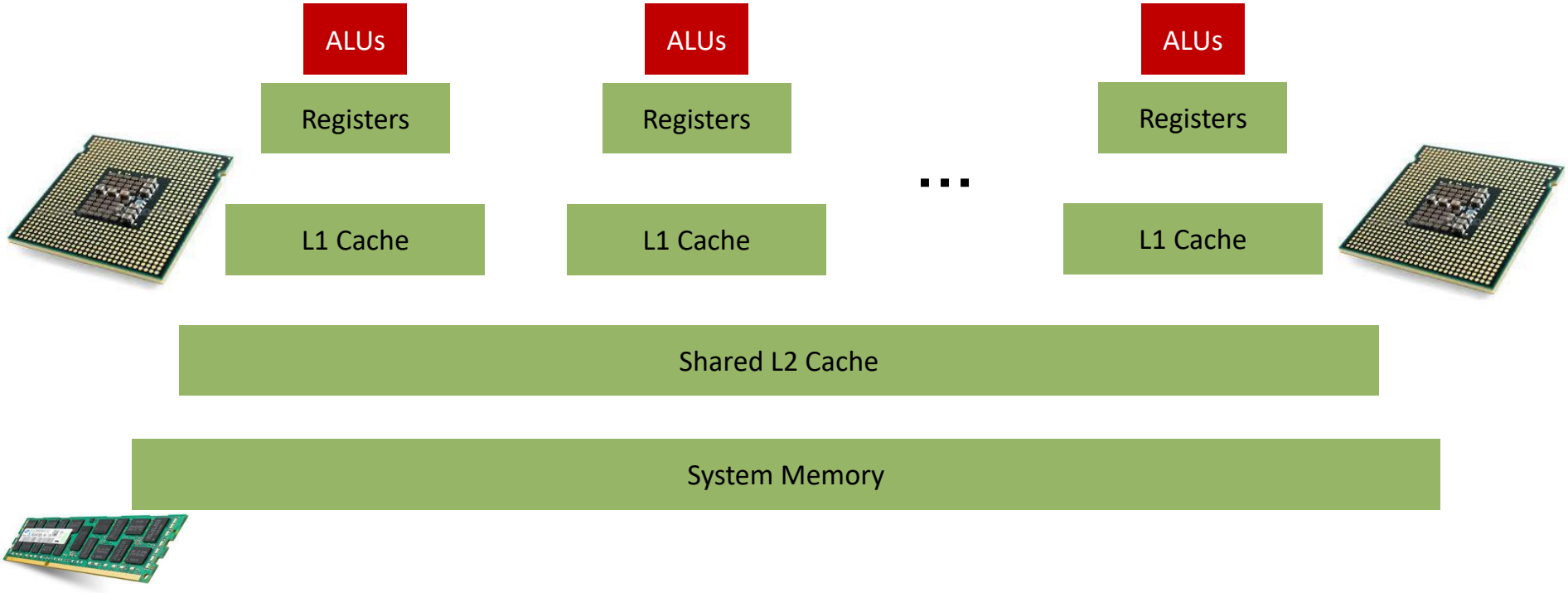
7 ns

100 ns

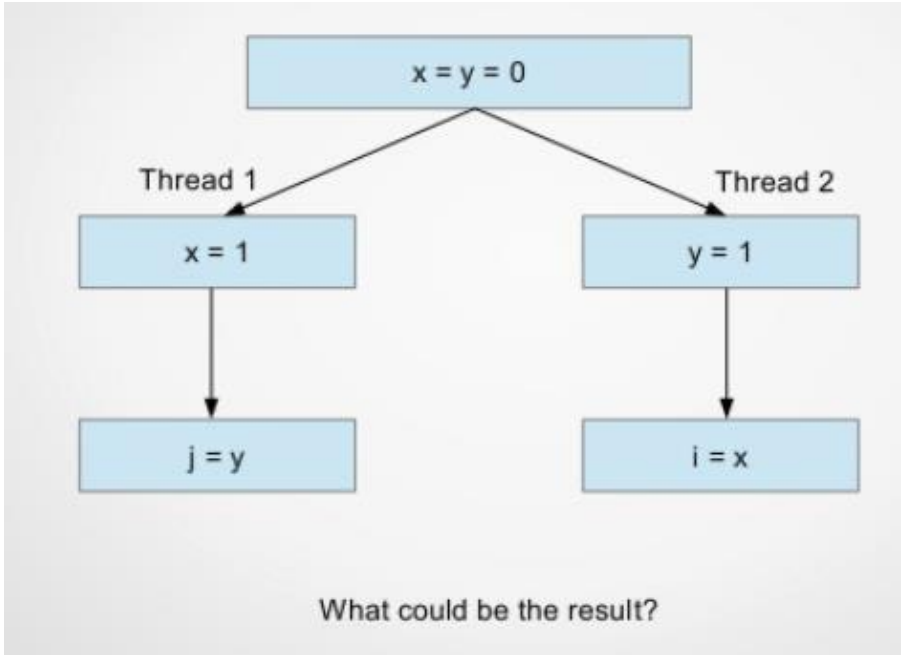
fast, low latency, high cost, low capacity

slow, high latency latency, low cost, high capacity

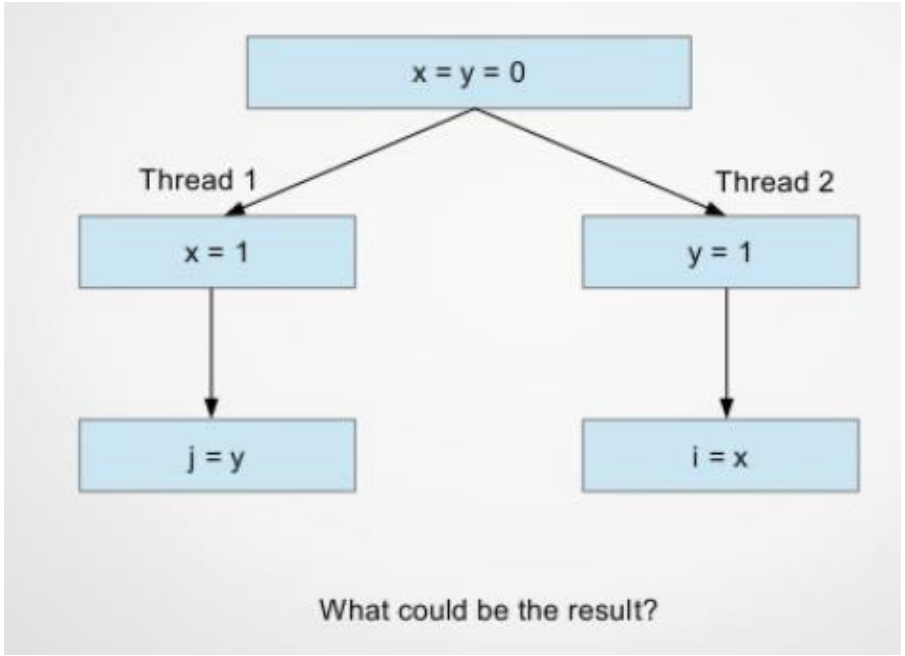
Memory hierachy (many cores)



Why memory models, x86 example



Why memory models, x86 example



Answer:
i=1, j=1
i=0, j=1
i=1, j=0
i=0, j=0 (but why?)

Visibility not guaranteed

And even if an action has been executed, we do not have guarantees that other threads see them (in the correct order).

In other words, actions that were performed by one thread may not be **visible** to another thread!

We want to make sure that the actions become visible. And we want some guarantees on the ordering.

How? Java Memory Model!

Java Memory Model (JMM): Necessary basics

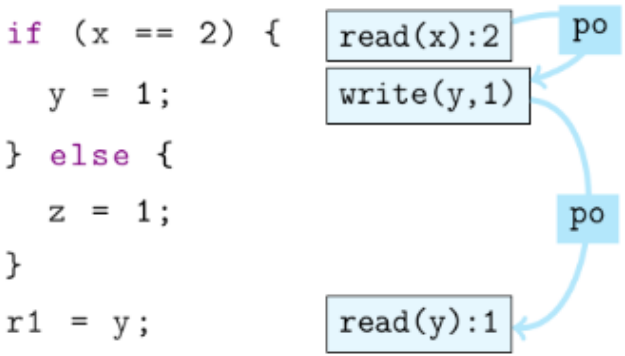
- **JMM restricts allowable outcomes of programs**
  - You saw that if we don't have these operations (volatile, synchronized etc.) – outcome can be "arbitrary" (not quite correct, say unexpected)

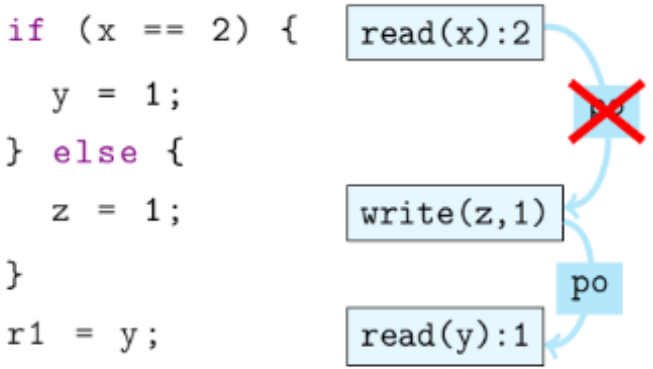
- **JMM defines *Actions:* `read(x):1` "read variable x, the value read is 1"**

- ***Executions* combine *actions* with *ordering:***
  - *Program Order*
  - *Synchronizes-with*
  - *Synchronization Order*
  - *Happens-before*

JMM: Program Order (PO)

- **Program order is a total order of intra-thread actions**
  - Program statements are NOT a total order across threads!
- **Program order does not provide an ordering guarantee for memory accesses!**
  - The only reason it exists is to provide the link between possible executions and the original program.
- **Intra-thread consistency: Per thread, the PO order is consistent with the thread's isolated execution**





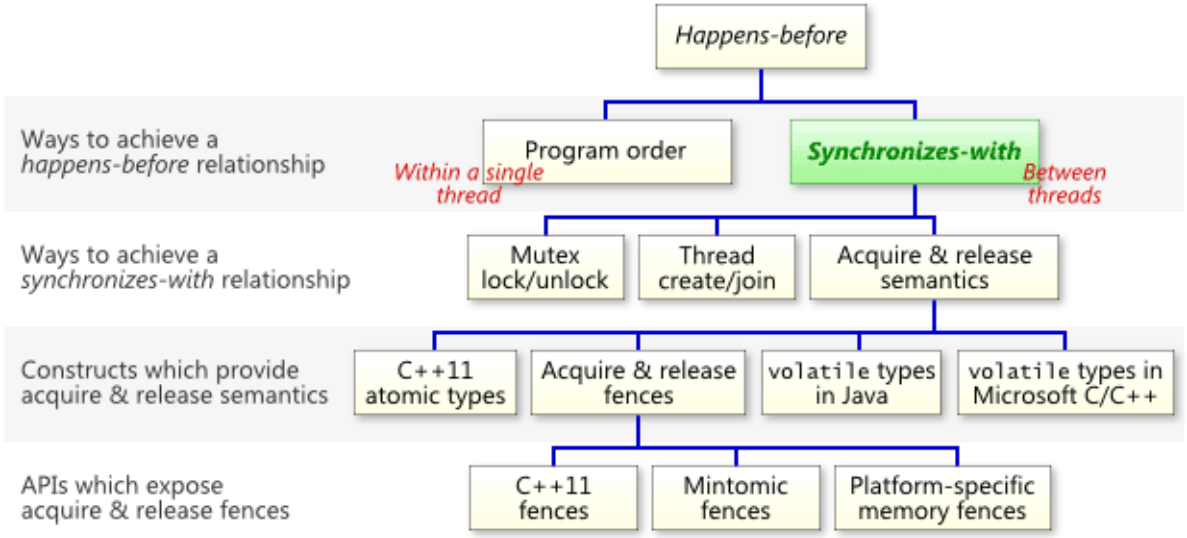
JMM: Synchronization Actions (SA) and Synchronization Order (SO)

- **Synchronization actions are:**
  - Read/write of a volatile variable
  - Lock monitor, unlock monitor
  - First/last action of a thread (synthetic)
  - Actions which start a thread
  - Actions which determine if a thread has terminated

- **Synchronization Actions form the Synchronization Order (SO)**
  - SO is a total order
  - All threads see SA in the same order
  - SA within a thread are in PO
  - SO is consistent: all reads in SO see the last writes in SO

JMM: Synchronizes-With (SW) / Happens-Before (HB) orders

- **SW only pairs the specific actions which "see" each other**
- **A volatile write to x synchronizes with subsequent read of x (subsequent in SO)**
- **The transitive closure of PO and SW forms HB**
- **HB consistency: When reading a variable, we see either the last write (in HB) or any other unordered write.**
  - This means races are allowed!



Problem: How do we implement locks?

- For two threads: Dekker‘s Algorithm, Peterson Lock
- For n threads: Filter Lock, Bakery Lock

Why do we need locks again?
Critical Sections

Critical sections

Pieces of code with the following conditions

1. **Mutual exclusion: statements from critical sections of two or more processes must not be interleaved**
2. **Freedom from deadlock: if some processes are trying to enter a critical section then one of them must eventually succeed**
3. **Freedom from starvation: if *any* process tries to enter its critical section, then that process must eventually succeed**

According to M. Ben Ari, Principles of Concurrent and Distributed Programming

Critical section problem

global (shared) variables

Process P

local variables

loop

- **non-critical section**
- **preprotocol**
- **critical section**
- **postprotocol**

Process Q

local variables

loop

- **non-critical section**
- **preprotocol**
- **critical section**
- **postprotocol**

Mutual exclusion for 2 processes -- 1st Try

```
volatile boolean wantp=false, wantq=false
```

Process P

local variables

loop

p1	non-critical section
p2	while(wantq);
p3	wantp = true
p4	critical section
p5	wantp = false

Process Q

local variables

loop

q1	non-critical section
q2	while(wantp);
q3	wantq = true
q4	critical section
q5	wantq = false

State Space Diagram

- When dealing with mutual exclusion problems, we should focus on:
  - the structure of the underlying state space, and
  - the state transitions that occur

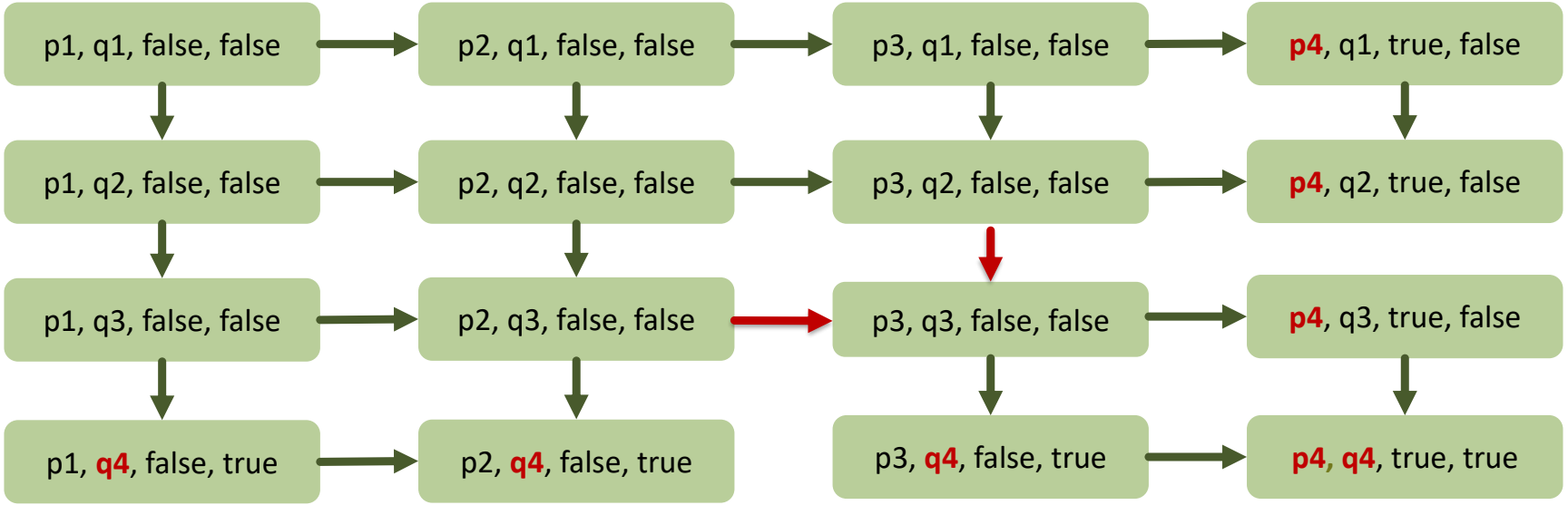
- State diagram captures the entire state space and all possible computations (execution paths a program may take)

- A good solution will have a state space with no bad states

State space diagram [p, q, wantp, wantq]

1 non-critical section **2** while(wantp) while(wantq) **3** wantp = true wantq = true **4** critical section **5** wantp = false wantq = false

p1	non-critical section
p2	while(wantq);
p3	wantp = true
p4	critical section
p5	wantp = false



Correctness of Mutual exclusion

- *"Statements from the critical sections of two or more processes must **not** be interleaved."*

- We can see that there is no state in which the program counters of both P and Q point to statements in their critical sections

Freedom from deadlock

- *“If some processes are trying to enter their critical sections then one of them must eventually succeed.”*

- We don’t have a situation when the processes aren’t making any progress anymore

Freedom from deadlock

- Since the behaviour of processes P and Q is symmetrical, we only have to check what happens for one of the processes, say P.

- Freedom from deadlock means that from any state where a process wishes to enter its CS (by awaiting its turn), there is *always a path* (sequence of transitions) leading to it entering its CS.

Freedom from deadlock

- Typically, a deadlocked state has no transitions leading from it, i.e. no statement is able to be executed.

- Sometimes a cycle of transitions may exist from a state for each process, from which no useful progress in the parallel program can be made. We call this a **Livelock**. Everyone is ‘busy doing nothing’.

Freedom from individual starvation

- *"If any process tries to enter its critical section then that process must eventually succeed."*
- If a process is wishing to enter its CS (awaiting its turn) and another process refuses to set the turn, the first process is said to be starved.
- Possible starvation reveals itself as cycles in the state diagram.
- Because the definition of the critical section problem allows for a process to not make progress from its Non-critical section, starvation is, in general, possible in this example

Dekker‘s Algorithm

volatile boolean wantp=false, wantq=false, integer turn= 1

```
Process P
loop
        non-critical section
        wantp = true
        while (wantq) {
                if (turn == 2) {
                        wantp = false;
                        while(turn != 1);
                        wantp = true; }}
        critical section
        turn = 2
        wantp = false
```

only when q tries to get lock

and q has preference

let q proceed

and wait

and try again

```
Process Q
loop
        non-critical section
        wantq = true
        while (wantp) {
                if (turn == 1) {
                        wantq = false
                        while(turn != 2);
                        wantq = true; }}
        critical section
        turn = 1
        wantq = false
```

Peterson Lock

let P=1, Q=2; volatile boolean array flag[1..2] = [false, false];
volatile integer victim = 1

Process P (1)

```
loop
    non-critical section
    flag[P] = true
    victim = P
    while(flag[Q] && victim == P);
    critical section
    flag[P] = false
```

I am interested

but you go first

We both are interested

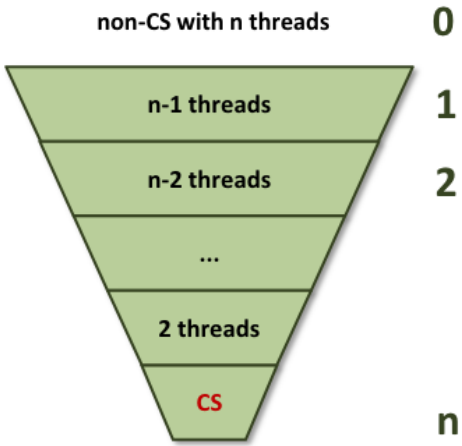
And you go first

Process Q (2)

```
loop
    non-critical section
    flag[Q] = true
    victim = Q
    while(flag[P] && victim == Q);
    critical section
    flag[Q] = false
```

Filter Lock

```
int[] level(#threads), int[] victim(#threads)
```




```
lock(me) {
  for (int i=1; i<n; ++i) {
    level[me] = i;
    victim[i] = me;
    while (∃k ≠ me: level[k] >= i && victim[i] == me) {};
  }
}

unlock(me) {
  level[me] = 0;
}
```

Other threads are at same or higher level

And I have to wait

Atomic Registers

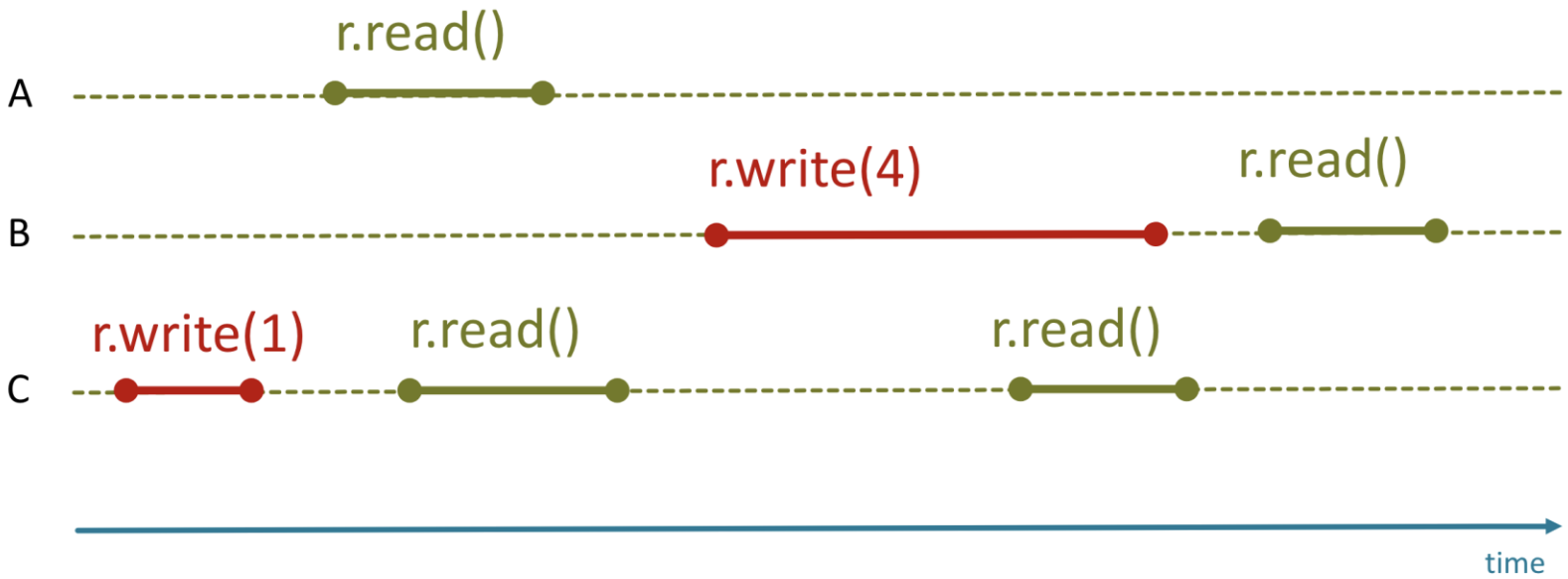
Register: basic memory object, can be shared or not
i.e., in this context register ≠ register of a CPU

Register *r* : operations *r.read()* and *r.write(v)*

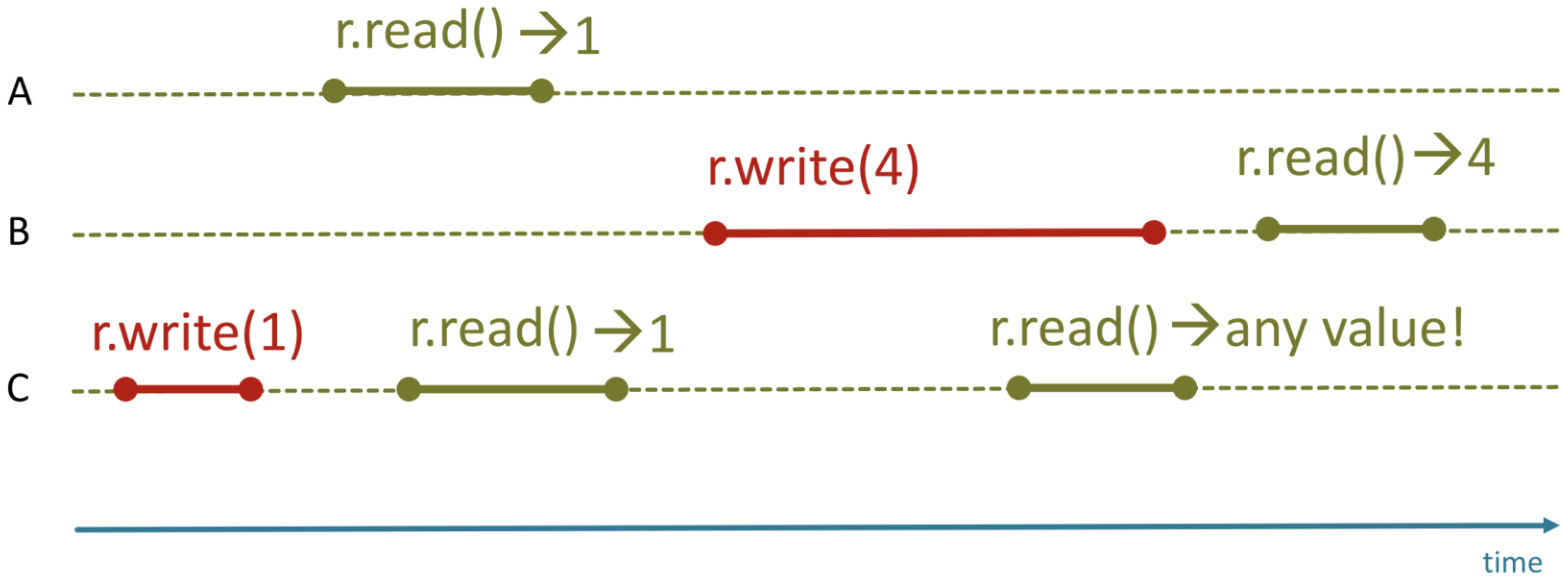
Atomic Register:

- **An invocation *J* of *r.read* or *r.write* takes effect at a single point $\tau(J)$ in time**
- **$\tau(J)$ always lies between start and end of the operation *J***
- **Two operations *J* and *K* on the same register always have a different effect time $\tau(J) \neq \tau(K)$**
- **An invocation *J* of *r.read()* returns the value *v* written by the invocation *K* of *r.write(v)* with closest preceding effect time $\tau(K)$**

Atomic Registers



Atomic Registers



Atomic operations

- An atomic action is one that effectively **happens at once** i.e. this action cannot stop in the middle nor be interleaved
- It either happens completely, or it doesn't happen at all.
- No side effects of an atomic action are visible until the action is complete

This essentially means that other Threads think that the change happened in an instant

Hardware support for atomic operations

- Test-And-Set (TAS)
- Compare-And-Swap (CAS)

TAS and CAS

boolean TAS(*memref* s)

atomic

```
if (mem[s] == 0) {
        mem[s] = 1;
        return true;
} else

return false;
```

int CAS (*memref* a, int old, int new)

atomic

```
oldval = mem[a];

if (old == oldval)

        mem[a] = new;

return oldval;
```

Lets build a spinlock using RMW operations

Test and Set (TAS)

```
Init (lock)
    lock = 0;
```

```
Acquire (lock)
    while !TAS(lock); // wait
```

```
Release (lock)
    lock = 0;
```

Compare and Swap (CAS)

```
Init (lock)
    lock = 0;
```

```
Acquire (lock)
    while (CAS(lock, 0, 1) != 0);
```

```
Release (lock)
    CAS(lock, 1, 0);
```

In Java…

```
public class TASLock implements Lock {
  AtomicBoolean state = new AtomicBoolean(false);

  public void lock() {
    while(state.getAndSet(true)) {
      //do nothing
    }
  }

  public void unlock() {
    state.set(false);
  }

}
```

TAS Spinlock scales horribly, why?

TAS

n = 1, elapsed= 224, normalized= 224
n = 2, elapsed= 719, normalized= 359
n = 3, elapsed= 1914, normalized= 638
n = 4, elapsed= 3373, normalized= 843
n = 5, elapsed= 4330, normalized= 866
n = 6, elapsed= 6075, normalized= 1012
n = 7, elapsed= 8089, normalized= 1155
n = 8, elapsed= 10369, normalized= 1296
n = 16, elapsed= 41051, normalized= 2565
n = 32, elapsed= 156207, normalized= 4881
n = 64, elapsed= 619197, normalized= 9674

Cache Coherency Protocol ☹

We have a sequential bottleneck!

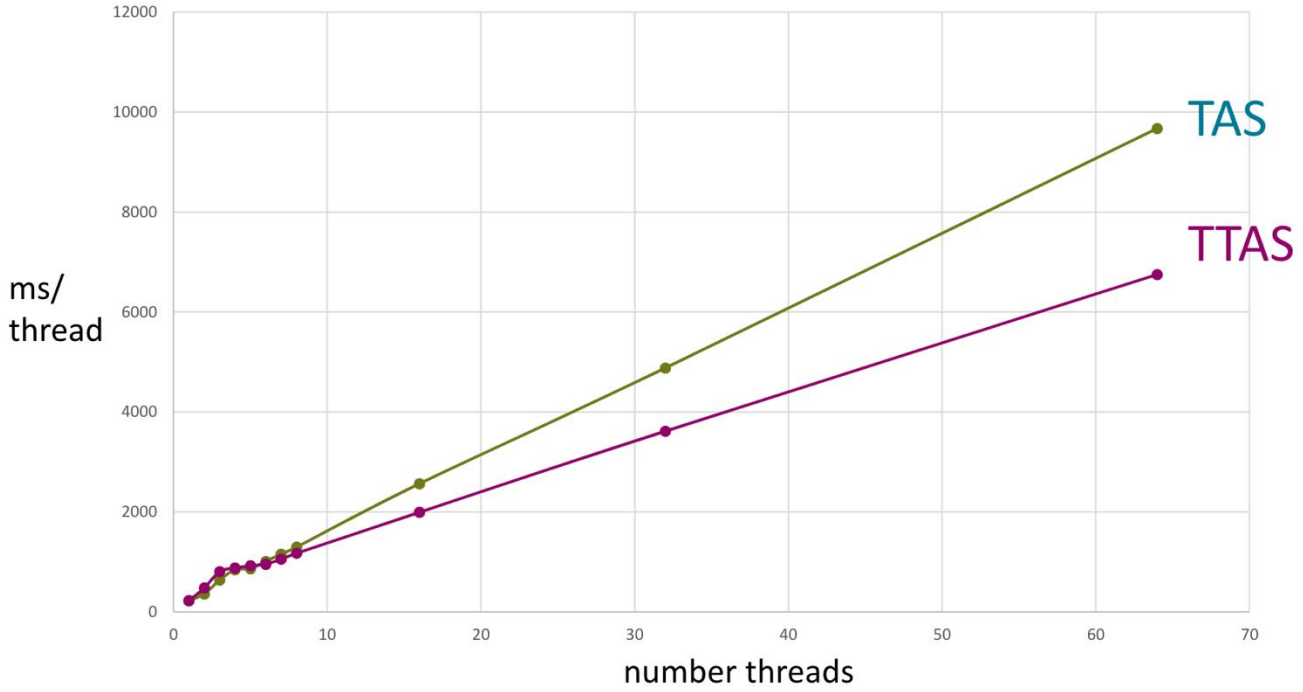
Each call to getAndSet() invalidates cached copies! => Threads need to access memory via Bus => Bus Contention!

“[...] the getAndSet() call forces other processors to discard their own cached copies of the lock, so every spinning thread encounters a cache miss almost every time, and must use the bus to fetch the new, but unchanged value.” - The Art of Multiprocessor Programming

Lets try spinning on local cache

```
public class TASLock implements Lock {
  AtomicBoolean state = new AtomicBoolean(false);

  public void lock() {
    do
      while (state.get() == true) //spins on local cache
    while(!state.compareAndSet(false, true)) {}
  }

  public void unlock() {
    state.set(false);
  }

}
```

It only helped a little bit



What we learned

- (too) many threads fight for access to the same resource
- slows down progress globally and locally
- **CAS/TAS: Processor assumes we modify the value even if we fail!**

Solution? Exponential Backoff

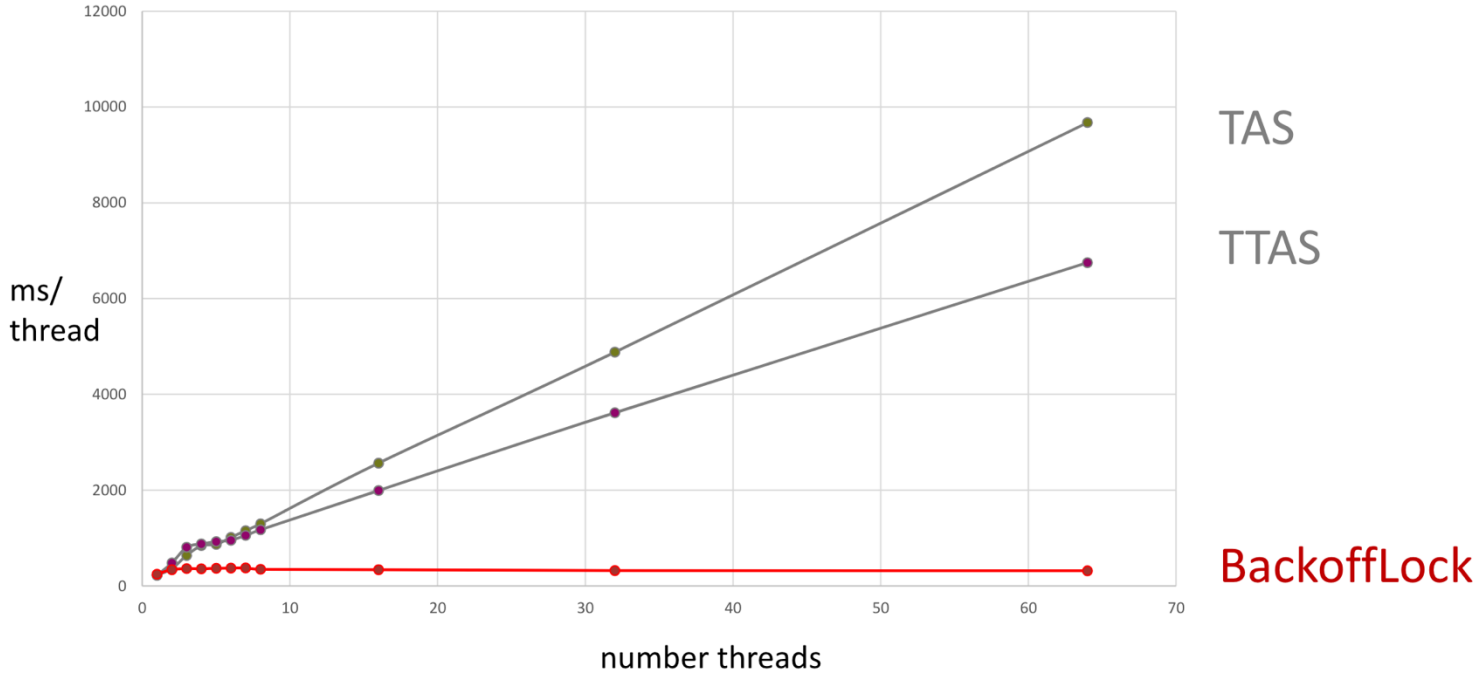
Idea: Each time TAS fails, wait longer until you re-try

- Backoff must be random!

Exponential backoff

```
acquire(lock):
  while True:
    while lock == 1: pass
    if CAS(lock, 0, 1) == 0:
      return True
    else:
      backoff *= 2
      sleep(backoff)

unlock(lock):
  lock = 0
```

Nice!


12000
10000
8000
6000
4000
2000
0
ms/
thread
0
10
20
30
40
50
60
70
number threads
TAS
TTAS
BackoffLock