DA343A
Objektorienterad programutveckling, trådar och datakommunikation

Föreläsning 9 – Synchronization

Ben Blamey

Föreläsning 9 : Synchronization

- Revision
  - Callbacks
  - Communication between threads
- Why do we need synchronization? – Method Invocation Ordering
- Synchronization
  - The synchronized keyword
  - wait() / notifyAll()
  - Synchronized blocks
- Thread termination
- Queues & Buffers
- Multi-Threaded Producer/Consumer Example

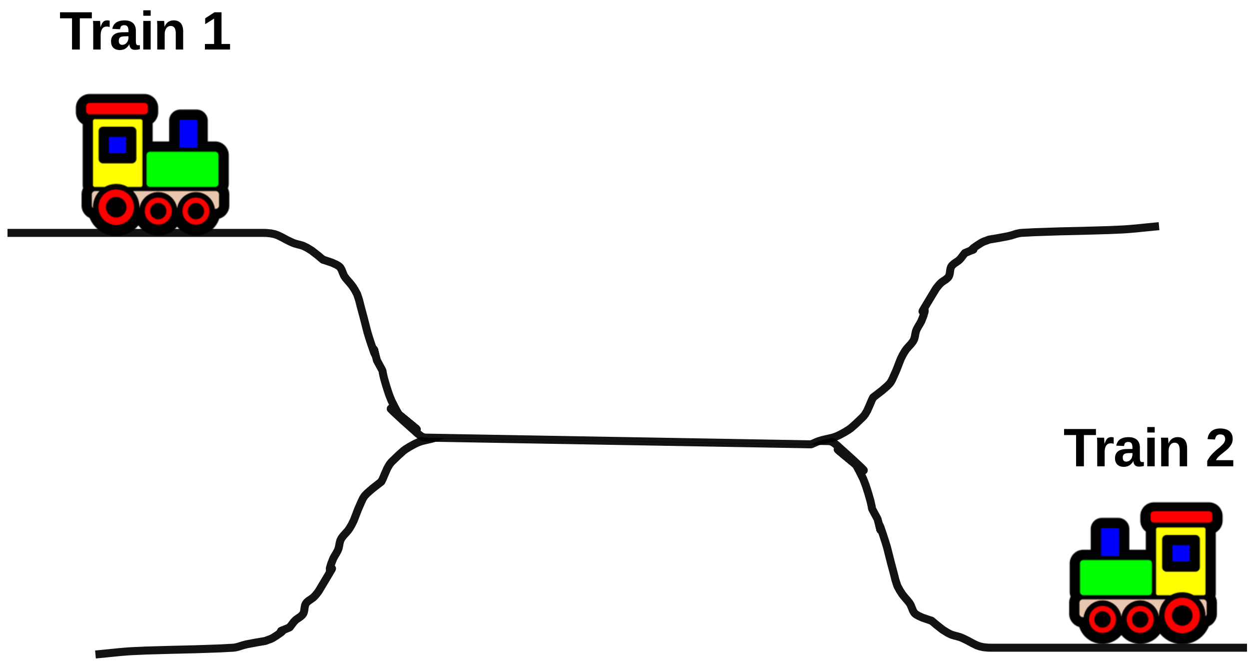
Revision: Callbacks

- A callback is a method that is registered with another object. The callback method can be invoked when:
  - A task is completed (or progress, error),
  - Something changes in the object.
- Callback can be invoked: immediately (synchronously) or later (asynchronously).
- We can use callbacks on a single thread, or with multiple threads.

Revision: Communication Between Threads

- **No communication**. “fire and forget”
- **Polling** (not recommended)
- **Callbacks** (can use arguments for extra information, ideally immutable types) (F8)
- **Shared State** – this requires some form of **synchronization**. Why?

Event Ordering Matters!



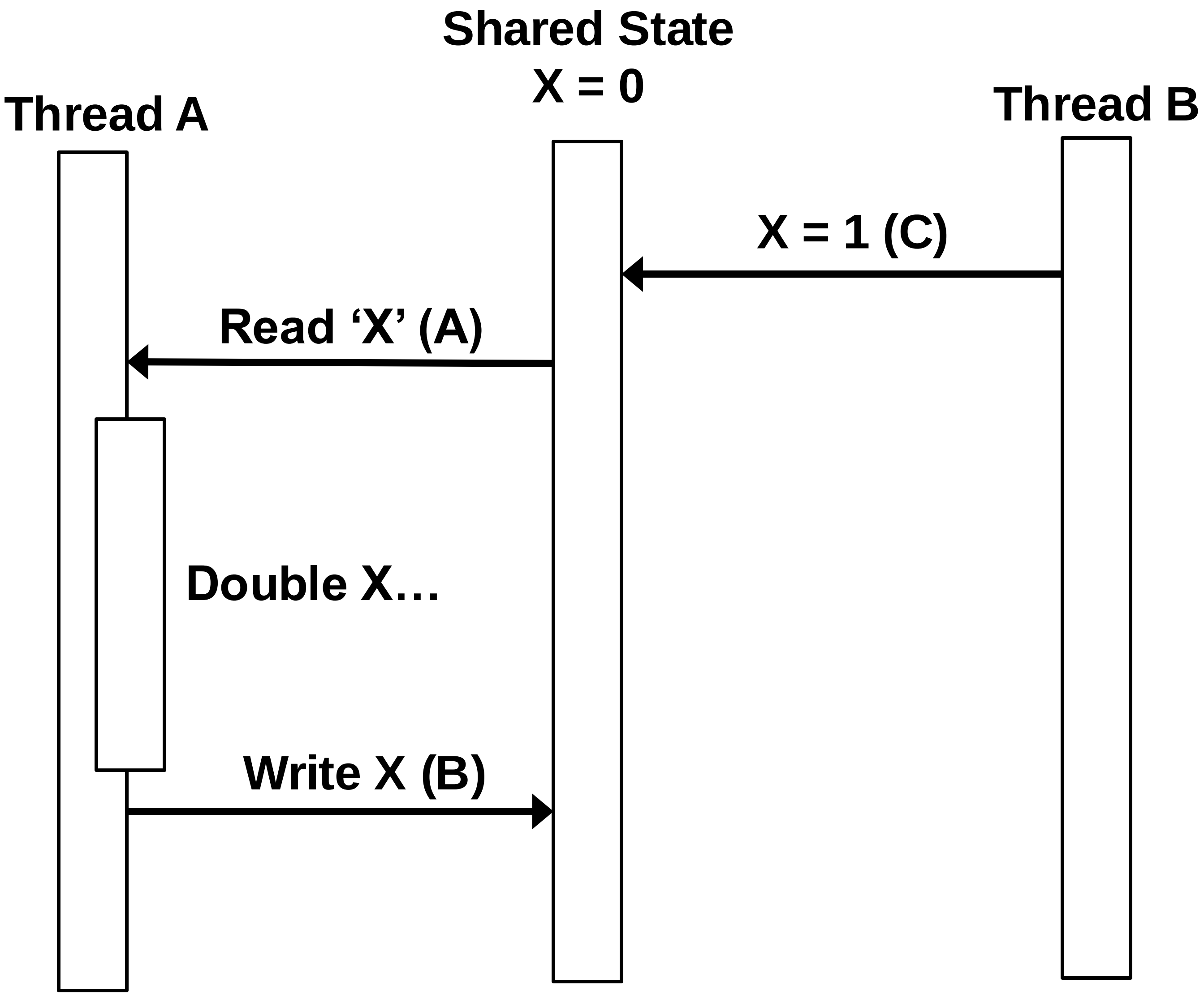
4 Events:

(A) Train 1 enters single track section.
(B) Train 1 leaves single track section.
(C) Train 2 enters single track section.
(D) Train 2 leaves single track section.

```
(A) ➡ (B) ➡ (C) ➡ (D)     OK
(C) ➡ (D) ➡ (A) ➡ (B)     OK
(A) ➡ (C) ➡ ( ) ➡ ( )     Collision!
(C) ➡ (A) ➡ ( ) ➡ ( )     Collision!
```

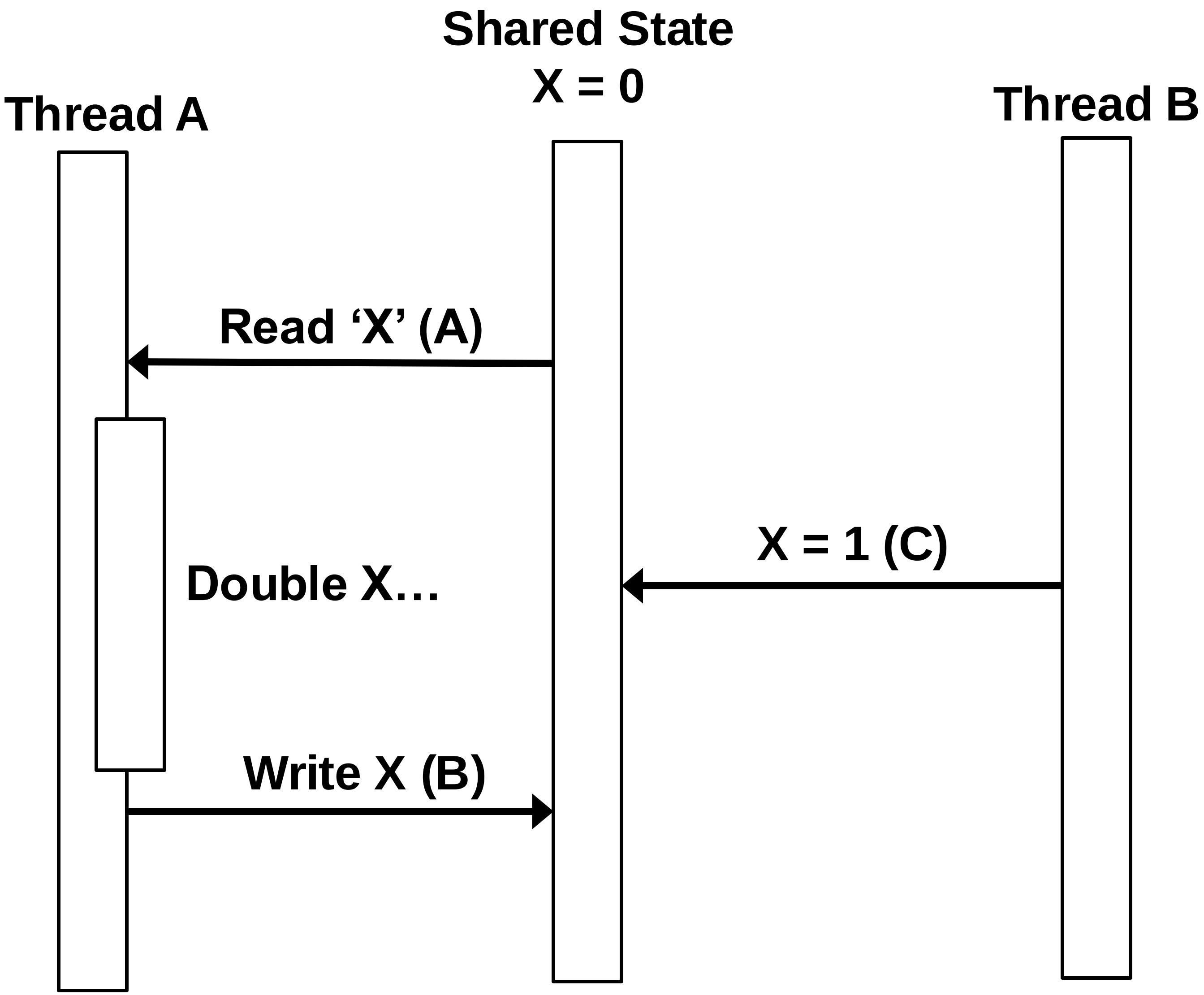
BrokenExample.java

```java
class SharedState {
    int x = 0;

    public void doubleX() {
        int x = this.x;
        this.x = x * 2;
    }

    public void setX1() {
        this.x = 1;
    }
}

private void run() throws InterruptedException {
    SharedState someSharedState = new SharedState();
    Thread threadA = new Thread(() -> {
        someSharedState.doubleX();
    });
    Thread threadB = new Thread(() -> {
        someSharedState.setX1();
    });

    threadA.start(); threadB.start();
    threadA.join(); threadB.join();
    System.out.print(String.format("X = %d . ", someSharedState.x));
}

public static void main(String args[]) throws InterruptedException {
    for (int i = 0; i < 100; i++) {
        new BrokenExample().run();
    }
}
```



In which order does it happen? What is the final value of X in each case?

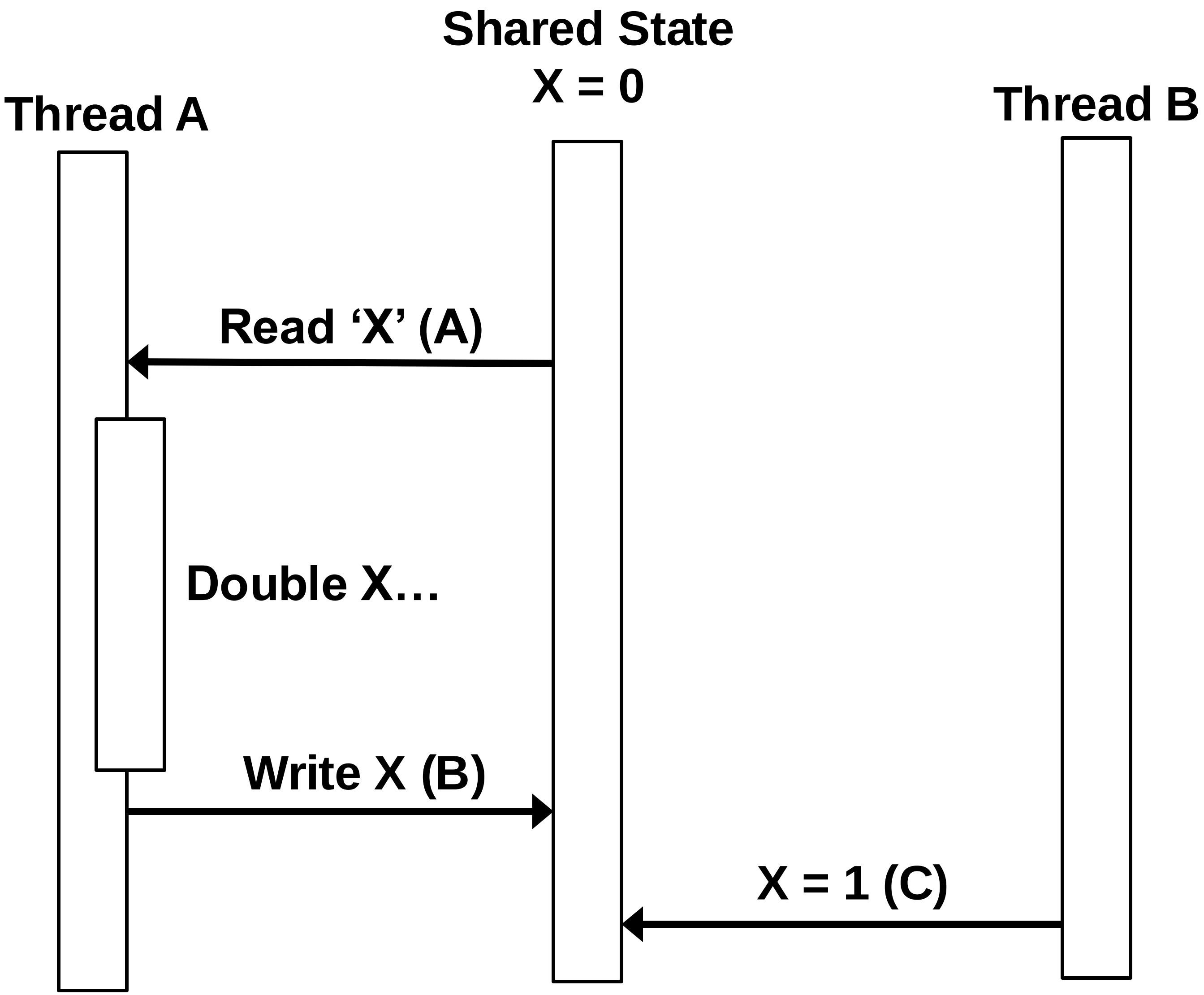
C ➡ A ➡ B
X = 2



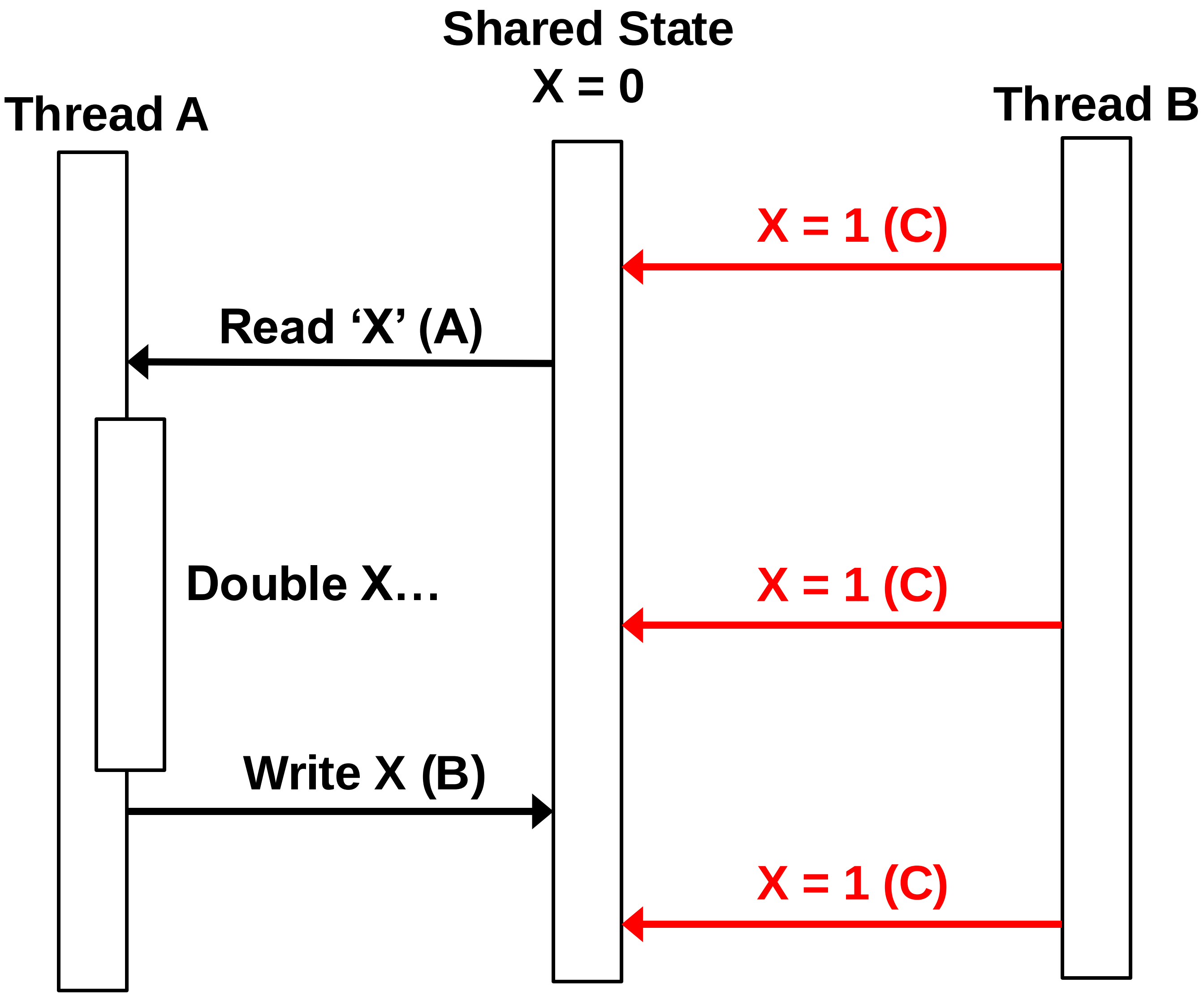
In which order does it happen? What is the final value of X in each case?

A ➡ C ➡ B
X = 0

In which order does it happen? What is the final value of X in each case?



A ➡ B ➡ C
X = 1



In which order does it happen? What is the final value of X in each case?

Any of the them?
X = (could be 0, 1 or 2)

BrokenExample.java

```java
class SharedState {
    int x = 0;

    public void doubleX() {
        int x = this.x;
        this.x = x * 2;
    }

    public void setX1() {
        this.x = 1;
    }
}

private void run() throws InterruptedException {
    SharedState someSharedState = new SharedState();
    Thread threadA = new Thread(() -> {
        someSharedState.doubleX();
    });
    Thread threadB = new Thread(() -> {
        someSharedState.setX1();
    });

    threadA.start(); threadB.start();
    threadA.join(); threadB.join();
    System.out.print(String.format("X = %d . ", someSharedState.x));
}

public static void main(String args[]) throws InterruptedException {
    for (int i = 0; i < 100; i++) {
        new BrokenExample().run();
    }
}
```

BrokenExample.java

```
class SharedState {
    int x = 0;

    public void doubleX() {
        int x = this.x;
        this.x = x * 2;
    }

    public void setX1() {
        this.x = 1;
```

```
Output:

X = 2 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 0 . X = 1 . X = 1 .
X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 .
X = 1 . X = 1 . X = 2 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 .
X = 1 . X = 1 . X = 1 . X = 0 . X = 1 . X = 2 . X = 1 . X = 1 . X = 2 . X = 1 .
X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 0 . X = 1 . X = 1 .
X = 1 . X = 0 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 .
X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 .
X = 1 . X = 2 . X = 1 . X = 1 . X = 1 . X = 0 . X = 1 . X = 1 . X = 1 . X = 1 .
X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 .
X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 .
```

```
        for (int i = 0; i < 100; i++) {
            new BrokenExample().run();
        }
}
```

BrokenExample.java

```
class SharedState {
    int x = 0;

    public void doubleX() {
        int x = this.x;
        this.x = x * 2;
    }

    public void setX1() {
        this.x = 1;
```

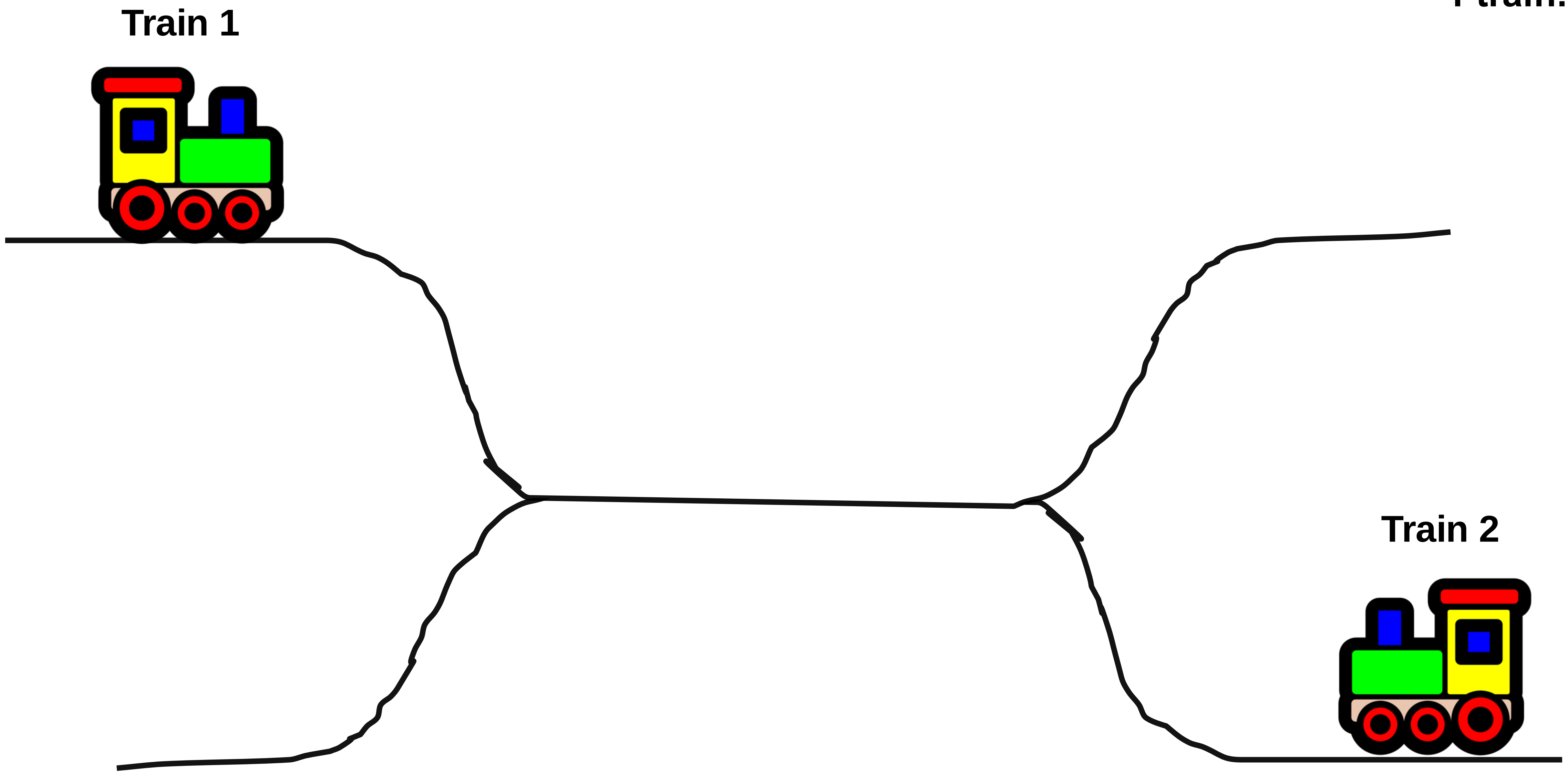
```
for (int i = 0; i < 100; i++) {
        new BrokenExample().run();
    }
}
```

```
Output:

X = 2 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 0 . X = 1 . X = 1 .
X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 .
X = 1 . X = 1 . X = 2 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 .
X = 1 . X = 1 . X = 1 . X = 0 . X = 1 . X = 2 . X = 1 . X = 1 . X = 2 . X = 1 .
X = 1 . X = 1 . X = 1 . X = 1 . X = = 1 .
X = 1 . X = 0 . X = 1 . X = 1 . X = = 1 .
X = 1 . X = 1 . X = 1 . X = 1 . X = = 1 .
X = 1 . X = 2 . X = 1 . X = 1 . X = = 1 .
X = 1 . X = 1 . X = 1 . X = 1 . X = = 1 .
X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 . X = 1 .
```

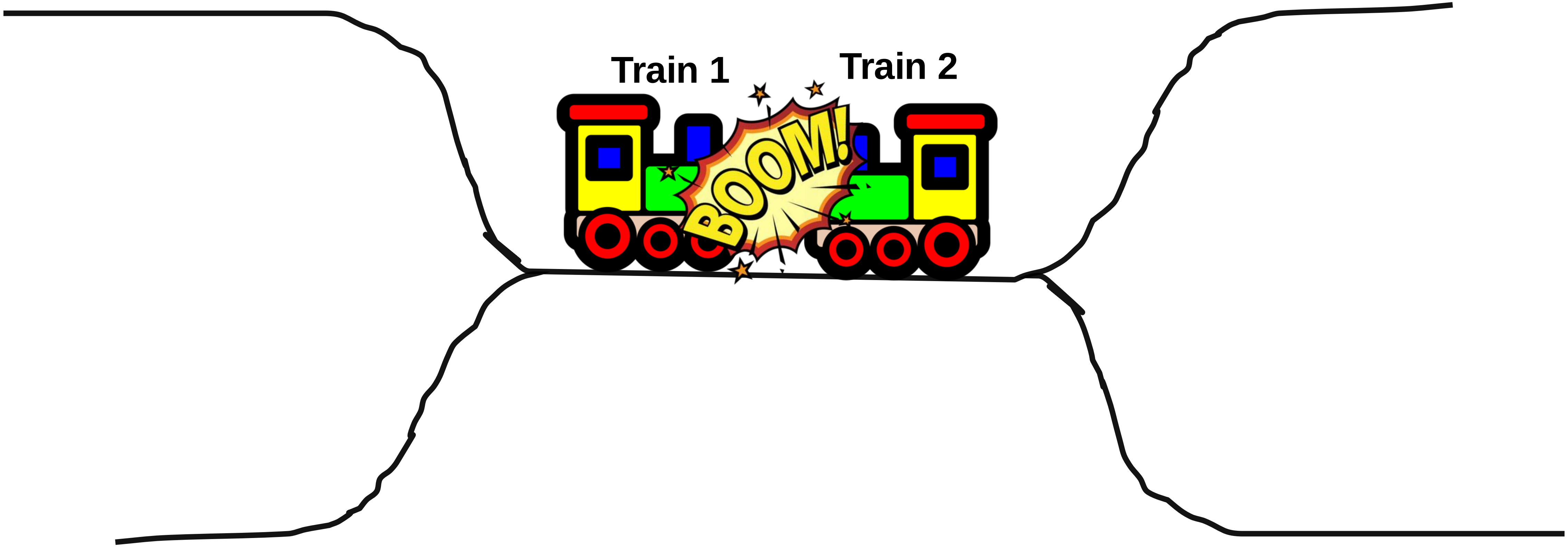
Program Behaviour is undefined!

We need to control the ordering of (i.e. *synchronize*) the events…

Single-Track Rails – for max 1 train…

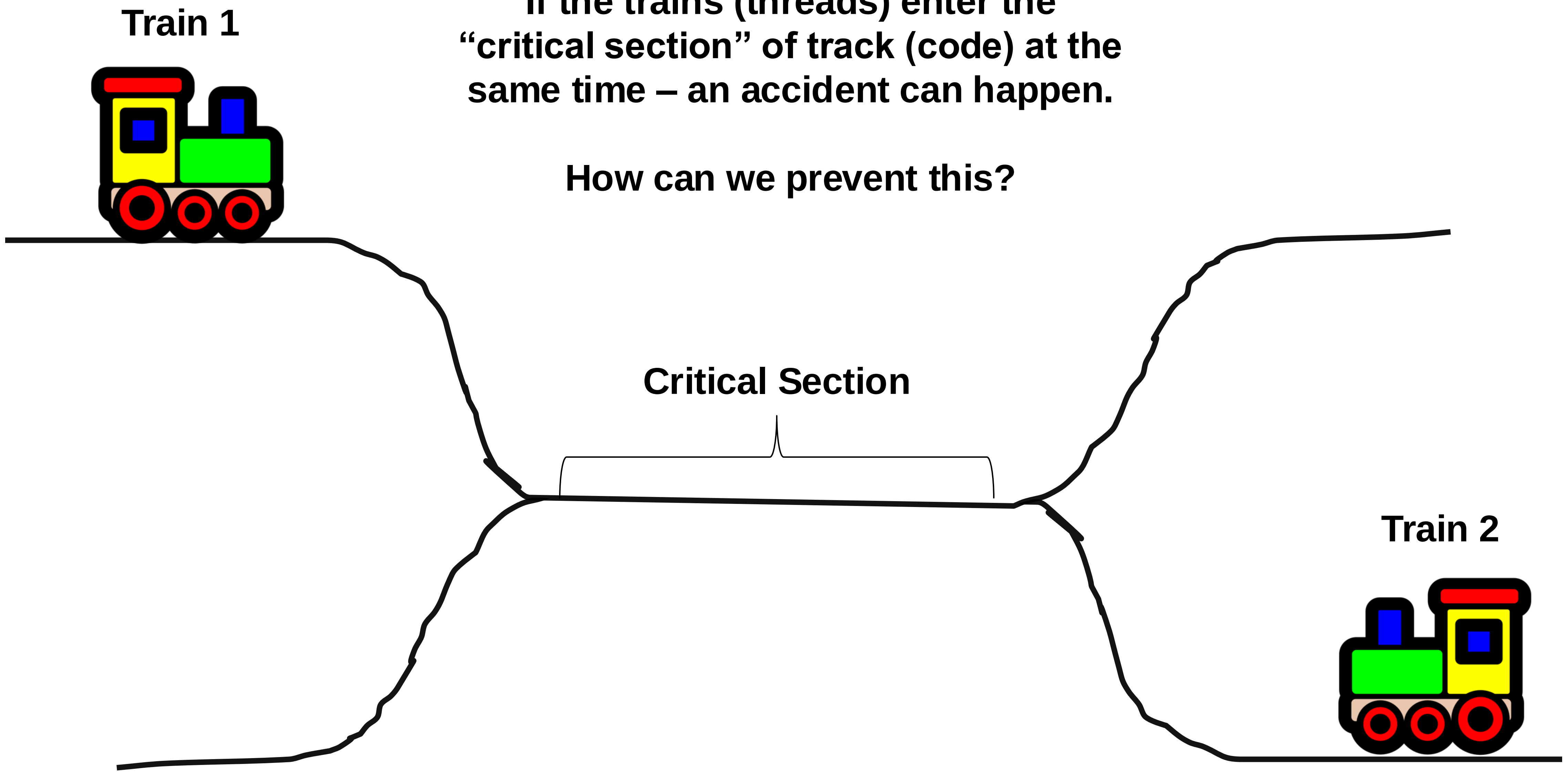


Depending on the speed of the trains, when they set off etc. etc….



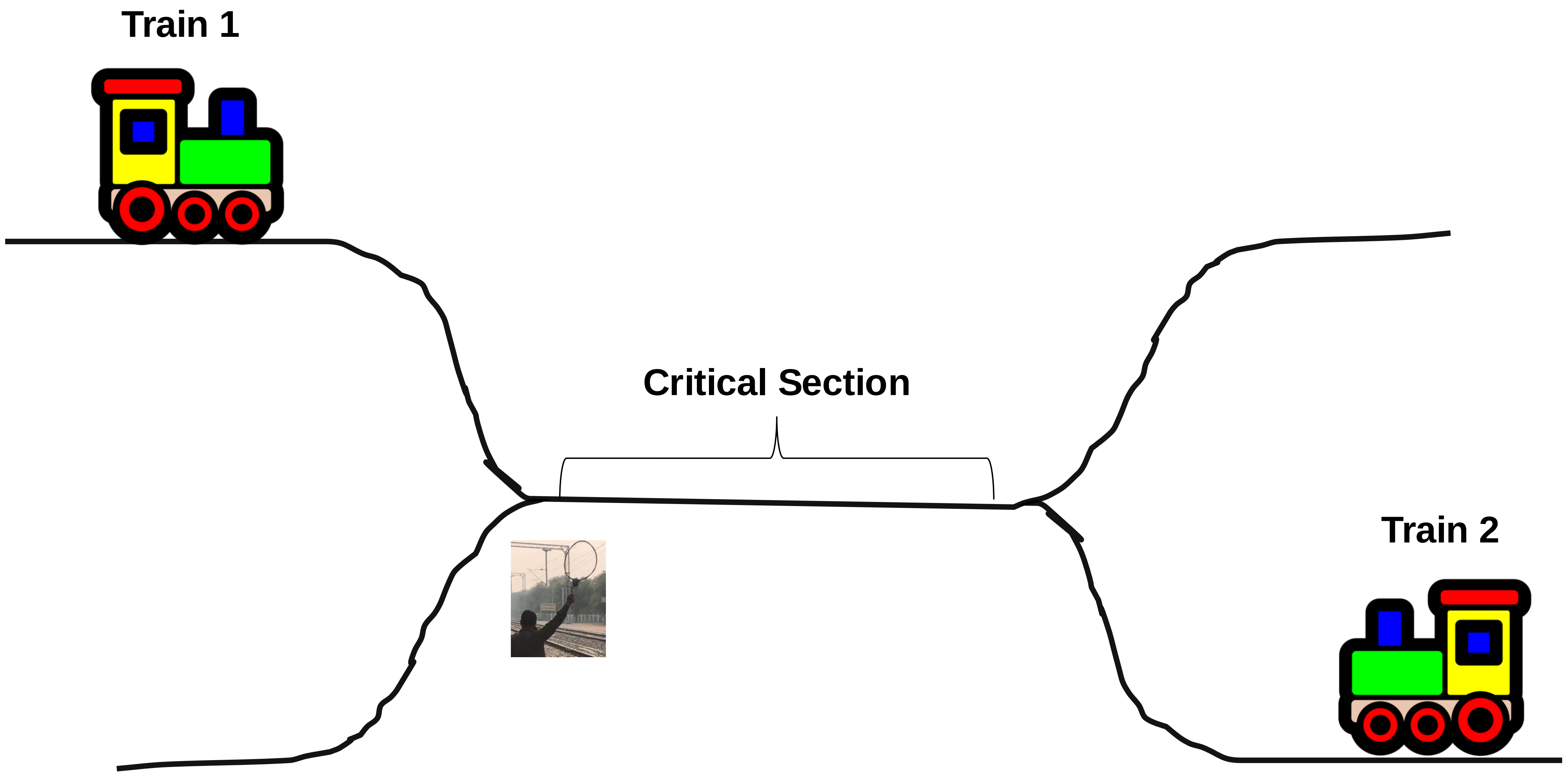
If the trains (threads) enter the “critical section” of track (code) at the same time – an accident can happen.

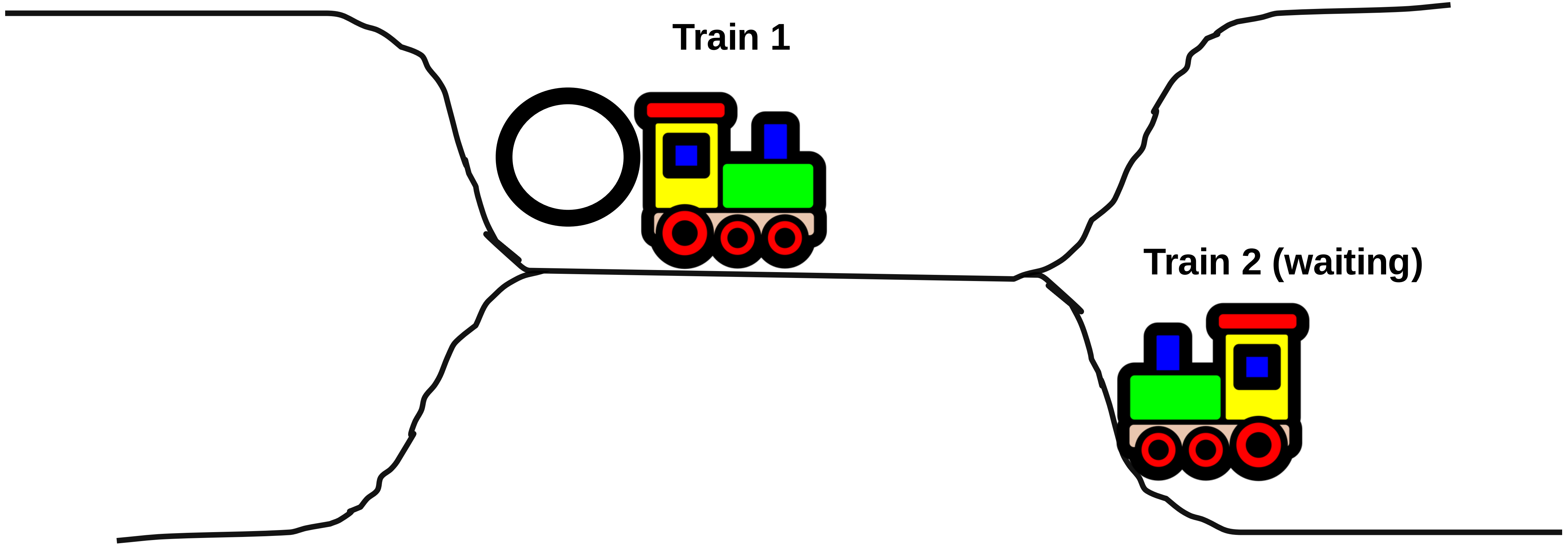
How can we prevent this?

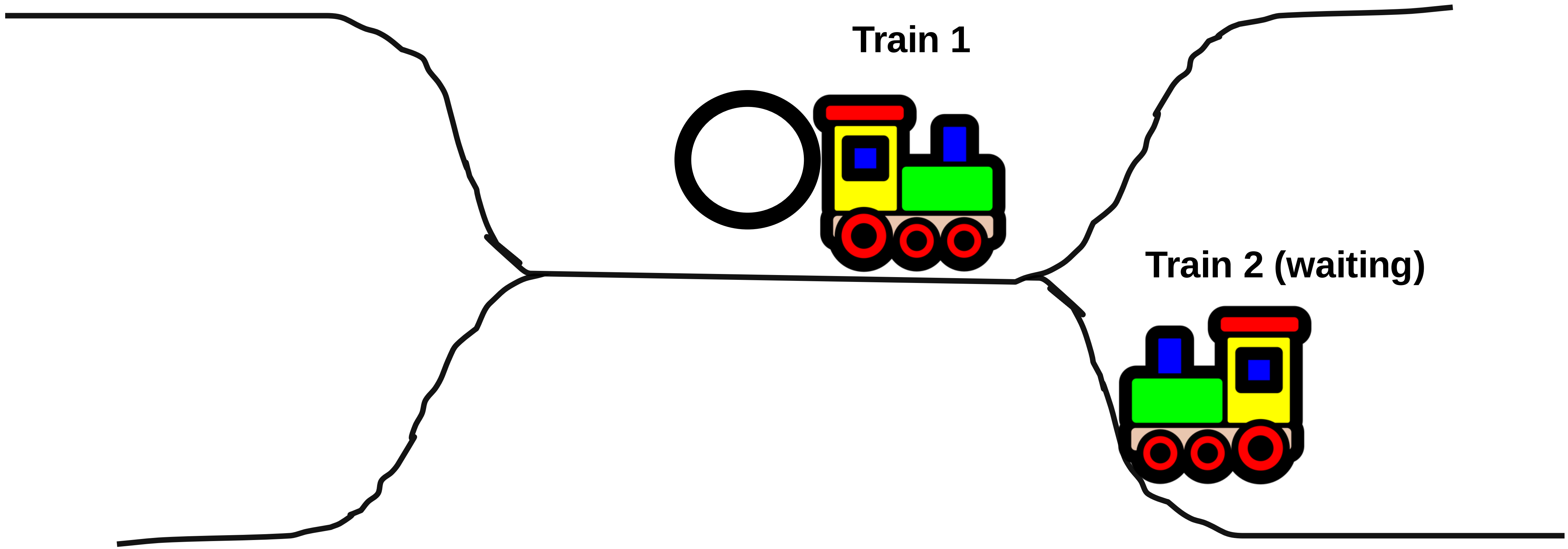


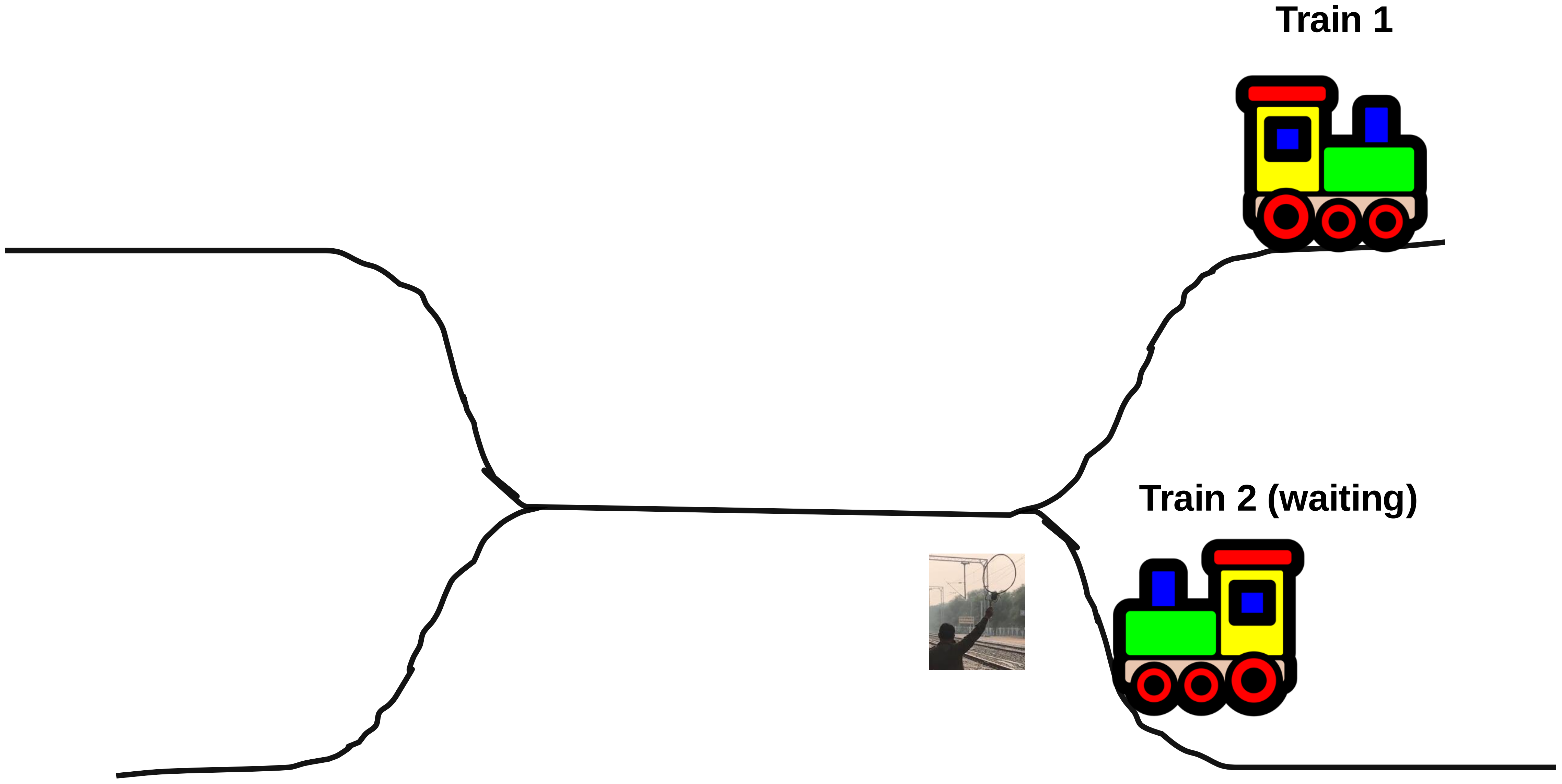
See: https://en.wikipedia.org/wiki/Token_(railway_signalling)

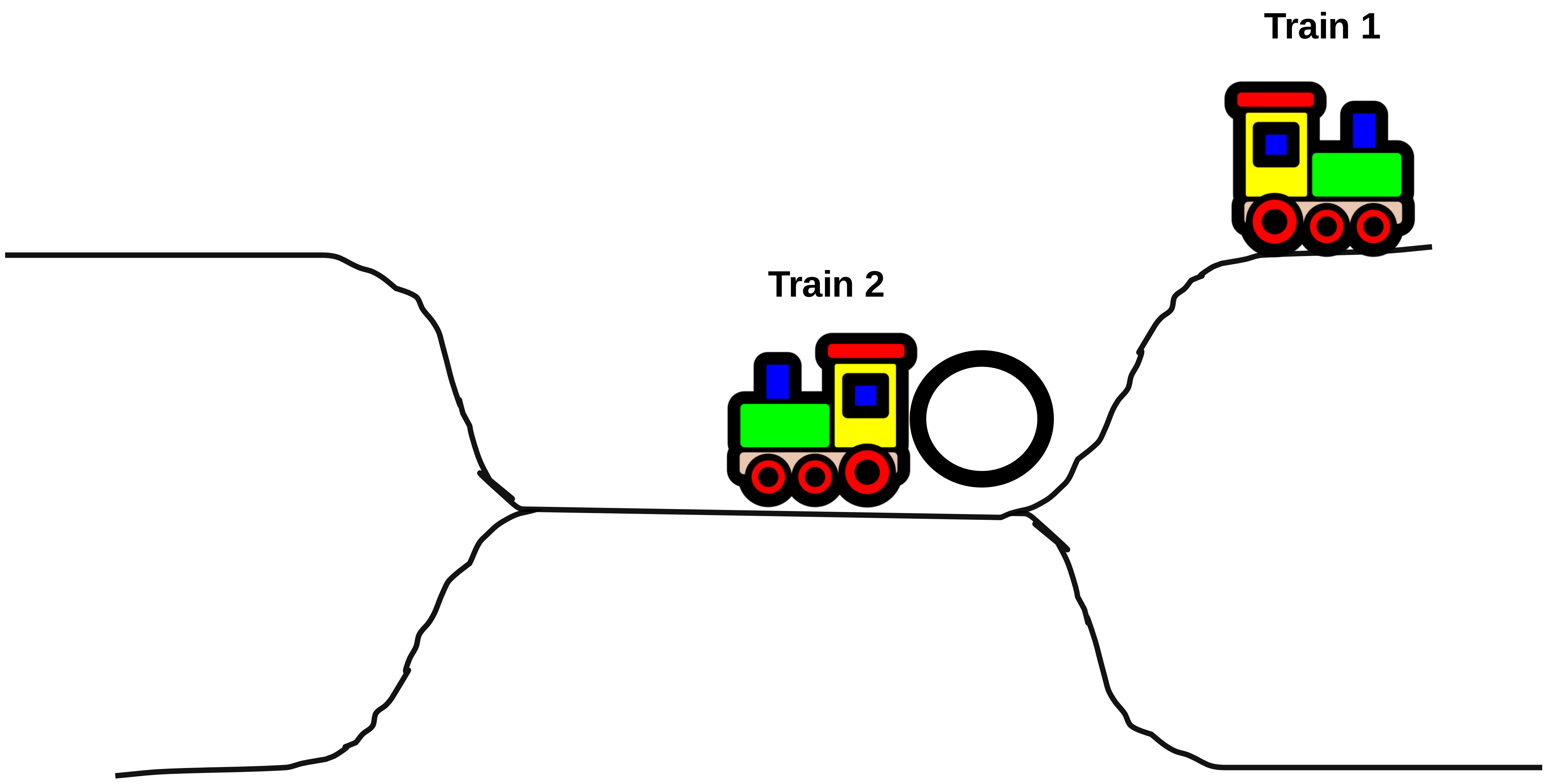

In Programming:
“Monitor” or “Lock”

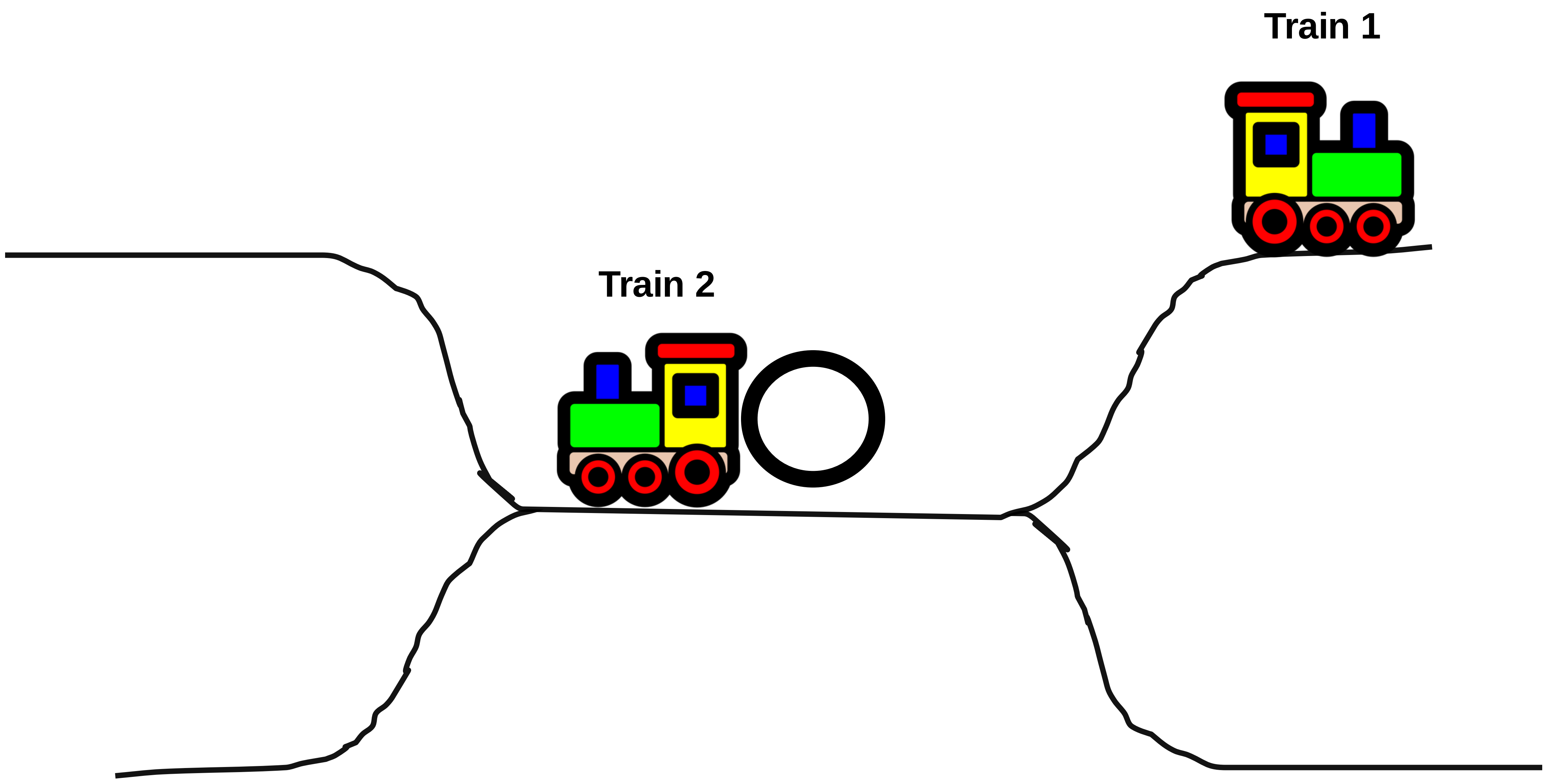

Train 1
Critical Section
Train 2

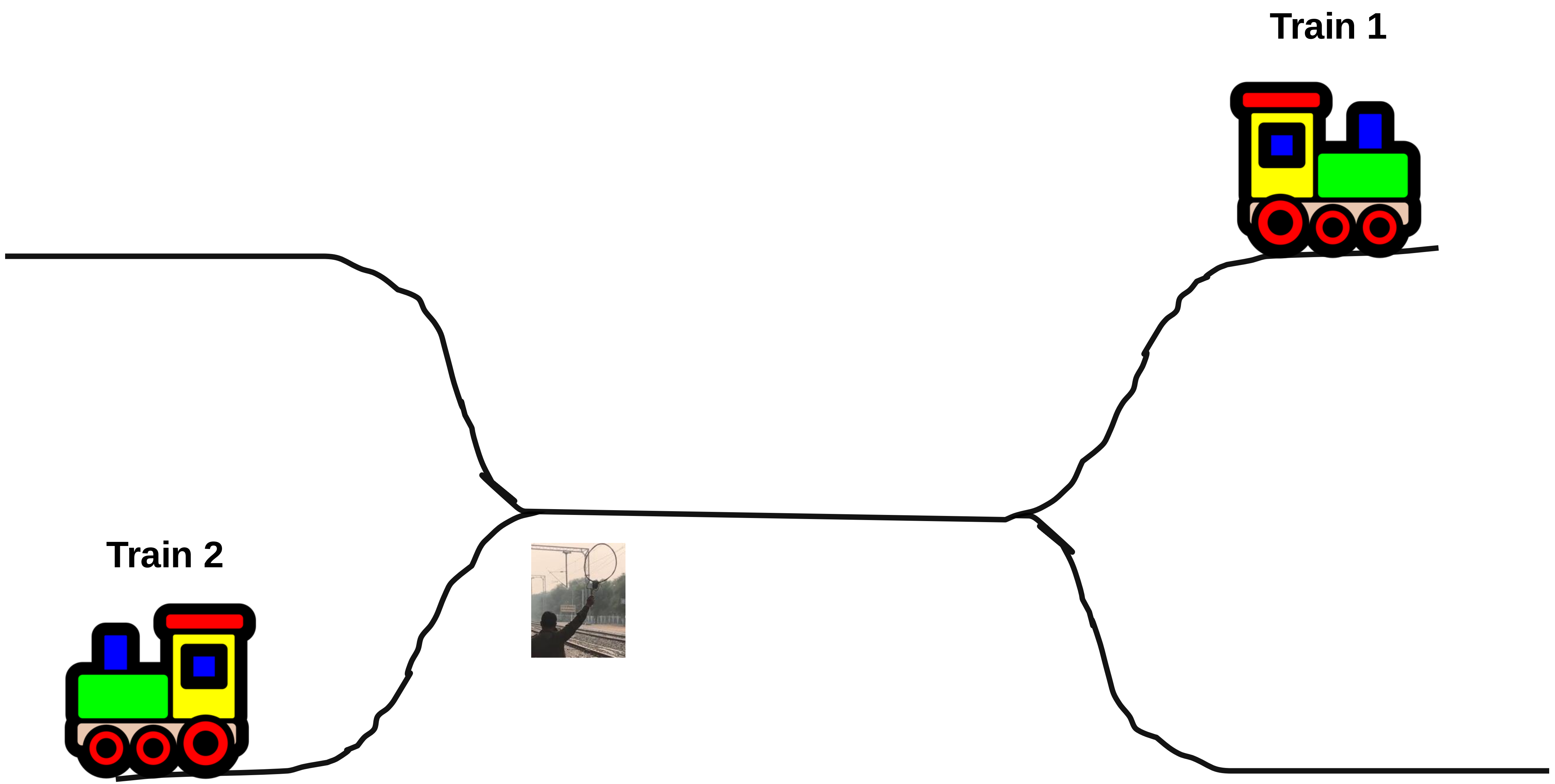

Train 1
Train 2 (waiting)


Train 1
Train 2 (waiting)


Train 1
Train 2 (waiting)


Train 1
Train 2


Train 1
Train 2


Train 1
Train 2

BrokenExample2.java

The synchronized keyword

```
class SharedState {
    int x = 0;

    public synchronized void doubleX() {
        int x = this.x;
        this.x = x * 2;
    }

    public synchronized void setX1(){
        this.x = 1;
    }
}
```

We can add the **synchronized** keyword to methods, as shown.

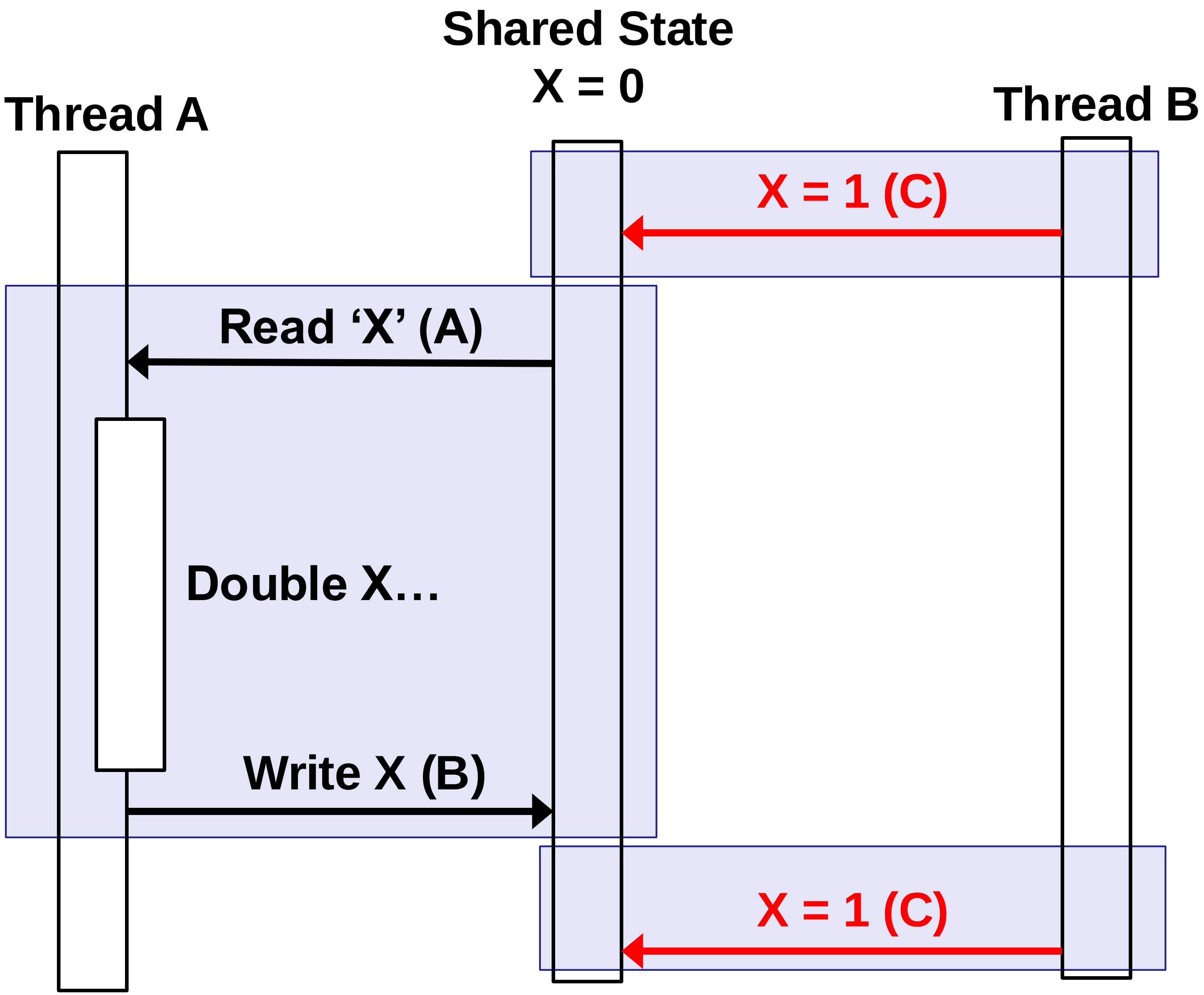
=> A thread can only proceed with the method when it holds the monitor or lock ().

"it is <u>not possible for two invocations</u> of synchronized methods on the same object <u>to interleave</u>.

When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object."

Source: https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

With the synchronized keyword:



Now in what order can it happen?

We can have:

(C) ➡ (A ➡ B)
Result: 2

-or-

(A ➡ B) ➡ (C)
Result: 1

(C) cannot occur between (A) and (B)!

But we still have a 'race condition' (its still broken)

The synchronized keyword

- The synchronized keyword controls access to all method innovocations on each **object independently.**
- Other threads can invoke synchronized methods **on other objects of the same class**.
- We can use the synchronized keyword when we are doing a complex operation, such as:
  - reading attributes, performing computation, and writing back a new value.
  - this includes **increment**/**decrement** operations i++/i– (these are really combined read & write operations!)
  - modifying more than one attribute at the same time.
  - or, e.g., operations on lists of objects stored in attributes.
- It prevents multiple threads from interfering with each other, potentially loosing data, overwriting values, leaving attributes inconsistent with eachother, or causing an exception.
- Sometimes, using synchronized on the necessary methods is enough to ensure **thread safety**.
- But, the synchronized keyword does not give any guarantees the order in which threads can execute synchronized methods – its ***först till kvarn!***
  - ***In our example, the behaviour is still undefined. It's still broken!***

wait() / notifyAll()

- We do not have control over the order in which different threads can invoke snychronized methods.
- For example, with queues, we cannot remove an item from the queue until at least one item is on the queue.
- Sometimes we need a thread to **wait()** for some **condition** so it can complete execution of the synchronized method**.**
- We can call **notifyAll()** so that other threads to re-check their condition.

wait() / notifyAll()

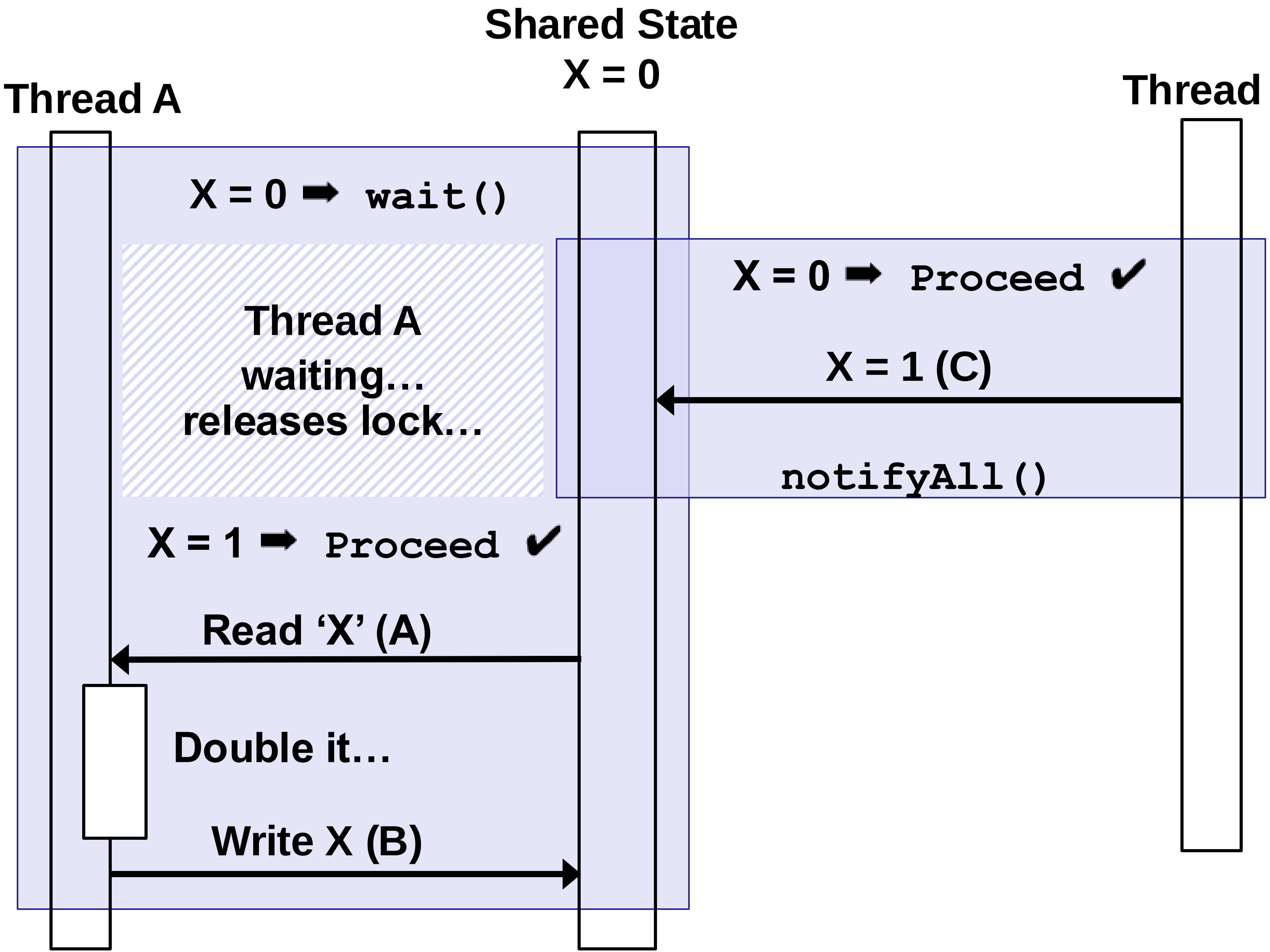
BrokenExample3.java

Now, the doubleX() and setX1() methods will wait() for their conditions to hold.

We allow the two threads to enter the synchronized blocks, but they **still cannot execute concurrently**.

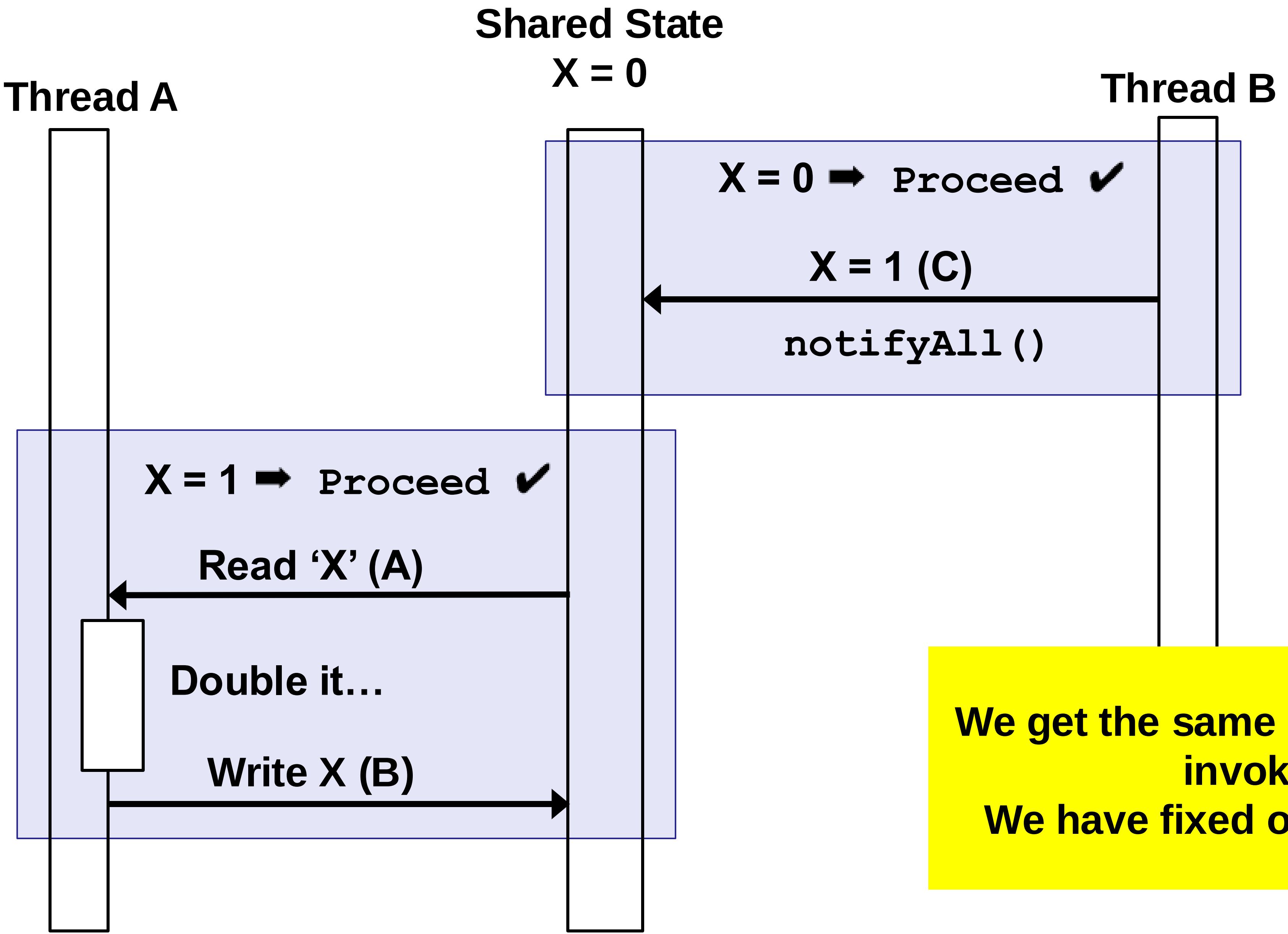
```
class SharedState {
    int x = 0;

    public synchronized void doubleX() throws InterruptedException {
        // We only want to continue with this method if x == 1.
        while (x != 1) {
            // allow another thread to proceed, and wait for them to call notifyAll()
            this.wait();
        }

        // Now x == 1, we can proceed.
        int x = this.x;
        this.x = x * 2;
        this.notifyAll(); // 'wake up' the other thread when we exit the sync block.
    }

    public synchronized void setX1() throws InterruptedException {
        // We only want to continue with this method if x == 0.
        while (x != 0) {
            // allow another thread to proceed, and wait for them to call notifyAll()
            this.wait();
        }

        // Now x == 0, we can proceed.
        this.x = 1;
        this.notifyAll(); // 'wake up' the other thread when we exit the sync block.
    }
}
```

DoubleX() invoked first…



Final Result:
X = 2

SetX1() invoked first…



Final Result:
X = 2

We get the same result, whichever is invoked first.
We have fixed our race condition!

wait() / notifyAll()

- wait() and notifyAll() can only be used in **synchronized blocks** (otherwise we get an exception).
- wait() and notifyAll() are implemented on the **Object class**.
- We should always **check the condition again after the wait()** – we should not assume the condition now holds just because of the notifyAll().
  - It not, we simply wait() again and check the next time.
  - For this reason, we should always use wait() **inside a loop**.
  - This is an example of a **design pattern**.
- The **notifyAll()** actually only happens when the thread exits the synchronized block (i.e. the method returns).
  - By convention, we put it at the end.
- **notifyAll()** notifies all waiting threads. Still only one thread at a time!
- There is also **notify ()** we can use, in some advanced scenarios this might give better performance.

Design Pattern for wait()/notifyAll()

```
public synchronized void someMethod() throws InterruptedException {

    while (<<condition is false>>) {
        // release lock, allow another thread to proceed
        // and wait for them to call notifyAll()
        this.wait();
    }

    // Now, <<condition is true>>, and we have the lock.
    // We can safely make our changes to the state…

    this.notifyAll(); // 'wake up' the other thread when we exit the method.
}
```

Synchronized Blocks

- Sometimes, we don't need the whole method to be a synchronized, just part of it.
- We can create a **synchronized block** within a method:

```
synchronized(this) {
        ….
}
```

- We can use any object as the lock!
  - Typically we use 'this'.
- The same as putting synchronized on the method.
  - Which implicitly uses 'this' as the lock.
  - We can still use notifyAll() / wait()

Break?

Classes in libraries ar often not synchronized (i.e. not thread safe)

java.util.HashMap<K,V>

Note that this implementation is not synchronized. If multiple threads access a hash map concurrently, and at least one of the threads modifies the map structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural

Source: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

Sychronization is expensive – we only use it when necessary.

One Solution: Facade Design Pattern

To synchronize such a class, you can:

1. Set an instance of the class to be private attribute of a new class
2. write synchronized methods that use the instance.

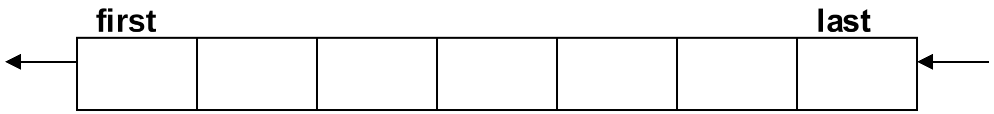
```
public class SynchronizedHashMap<K,V> {
    private HashMap<K,V> map = new HashMap<K,V>();

    public synchronized V put(K key, V value) {
        return map.put(key, value);
    }

    public synchronized V remove(K key) {
        return map.remove(key);
    }

    public synchronized void clear() {
        map.clear();
    }

    public synchronized V get(K key) {
        return map.get(key);
    }
}
```

Thread Termination

- To terminate the application, **all the threads** need to terminate.
- We want to do this **cleanly** so that all important tasks finish before the application exits – for example saving the user's work.
- We can wait for a thread to finish by calling **join()** from another thread.
  - This call 'blocks' until the other thread is terminated.
- We can also call **interrupt()** on the thread object.
- See: https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html
- A common design pattern for worker threads is to use a **loop**:
  - A queue of incoming tasks or items to process.
  - If the queue is empty, we **wait()** for more tasks.
  - Also check the status of the **interrupted()** flag, to break the loop and terminate the thread.
  - Can also have a boolean "closing" flag, to break the loop and terminate the thread.

Queue (kö) Design Pattern

- A **queue (kö)** is a familiar concept.
- A queue is usually **FIFO** ("first-in-first-out")
- **LIFO** ("last in first out"), we call a **stack.**



- Components which **add** (or **put**) items to the queue are sometimes called a **producers**
- Components which **remove** (or **get, pop**) items from the queue are sometimes called a **consumers**
- There might be **more than one** producer/consumer.
- We often use queues to send data between threads. This is a form of **buffer.**
- This pattern allows the producer thread(s) to continue working while consumer thread(s) process items.

Buffer<T>

```
public class Buffer<T> {

    // LinkedList<T> is not synchronized (thread safe)
    private final LinkedList<T> buffer = new LinkedList<T>();

    public synchronized void put(T o) {
        buffer.addLast(o);
        notifyAll();
    }

    public synchronized T get() throws InterruptedException {
        while(buffer.isEmpty()) {
            wait();
        }
        return buffer.removeFirst();
    }

    public int size() {
        return buffer.size();
    }
}
```

Buffer<T> Example

See:

- **Buffer.java**
- **Consumer.java**
- **Producer.java**
- **BufferDemo.java**

Exercise: adapt the example there is a queue of *Runnable* objects, and have the consumers execute them.

Can you adapt your solution to use callbacks with the results?

Writing Multi-Threaded Code

- With **single-threaded code**, if it works (and we have tests) – we feel like we're done!
- With **multi-threaded code**, just because it works, it might still be broken.
  - Even if we run our program several times, and we get the correct output, this does not necessarily mean we have thread-safe code.
  - Running the same code later on, or on a different machine, may produce different results.
  - Maybe there will only be a problem .01% of the time!
  - It could be that future modifications to the code will expose existing problems with thread safety, which were just avoided with good luck.
- Nor can we easily write unit tests for thread safety.
- There are tools for code analysis (outside the scope of the course).
- We imagine how different threads can interact with our objects – and call methods in all possible different orders.
- Use the debugger!
- Print out the name of the current thread. (can also **assert** which thread we are on)

Questions?