Data-Race Free Java

Tobias Wrigstad, PLISS 2026

An Introduction to Data-Race Free Java

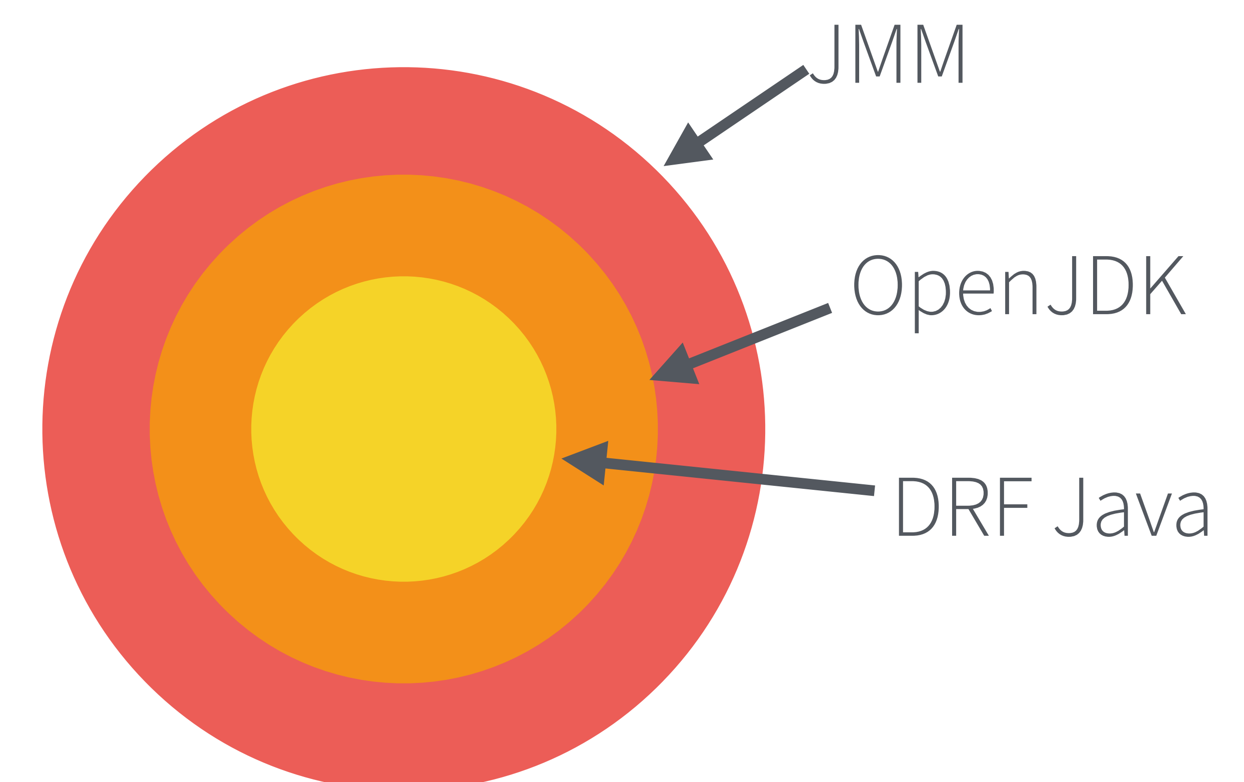
- What are we doing: **reinterpreting the JMM** (— changing the way Java programs execute!)

  1. We want to make it **simpler** to write a **correct** concurrent program
  2. We want to **data-races** to throw **exceptions**
  3. We want to **remove behaviour** not fully defined by the JMM

- Our goal is to be backwards-compatible

  Correctly synchronised programs should **behave exactly the same** as they do in Java today

  All **incorrectly** synchronised programs should exhibit behaviour that is **permitted** by the **current JMM**



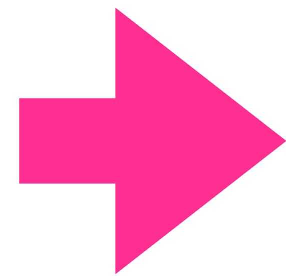
The core idea: enforced thread isolation + communication

- When a thread starts, it gets its own **private snapshot** of the current system
  - There is **no shared state**
- Synchronisation actions in the JMM are reinterpreted as **communication** with other threads
  - **Taking a lock** means **updating the private shapshot** with changes **published by other threads**
  - **Releasing a lock** means **publishing private changes** so that others **can update** their snapshots with these change
- Thread **isolation** gives us structural **freedom from read-write races**
  - **Communication** can detect **write-write races**
  - Write-write races are detected at **well-defined points in time**: synchronisation actions
  - Leakage

Very simple example:

Main's world

`cell = {0}`



```
1  final int[] cell = new int[1] = {0};
2  Thread t = Thread.ofVirtual().start(() -> {
3    cell[0] = 1;
4  });
5  System.out.println("Before " + cell[0]);
6  t.join();
7  System.out.println("After " + cell[0]);
```

Main thread

Line

Program order 1 → 2 →

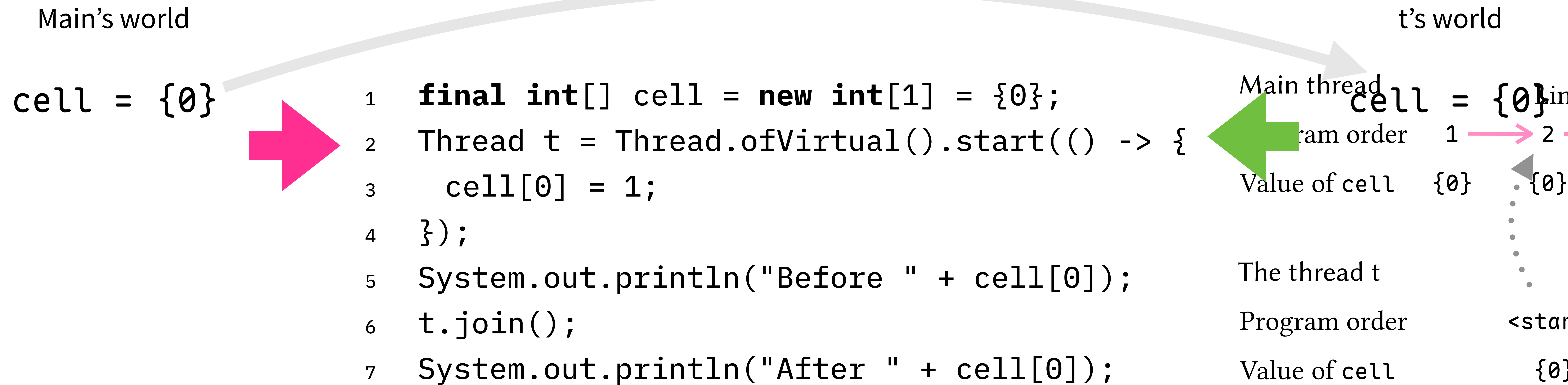
Value of `cell` {0} {0} {

The thread t

Program order <start> →

Value of `cell` {0} {

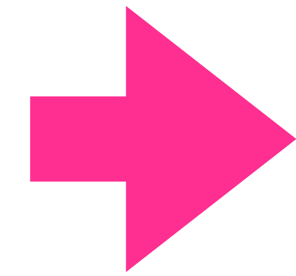
Very simple example:



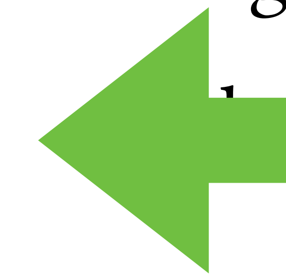
Very simple example:

Main's world

`cell = {0}`



```
1  final int[] cell = new int[1] = {0};
2  Thread t = Thread.ofVirtual().start(() -> {
3    cell[0] = 1;
4  });
5  System.out.println("Before " + cell[0]);
6  t.join();
7  System.out.println("After " + cell[0]);
```



t's world

`cell = {1}`

Main thread

Program order 1 → 2 → 5

Value of `cell` {0} {0} {0

The thread t

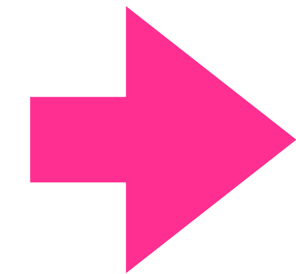
Program order `<start>` →

Value of `cell` {0} {1

Very simple example:

Main's world

`cell = {0}`



```
1  final int[] cell = new int[1] = {0};
2  Thread t = Thread.ofVirtual().start(() -> {
3    cell[0] = 1;
4  });
5  System.out.println("Before " + cell[0]);
6  t.join();
7  System.out.println("After " + cell[0]);
```

t's world

`cell = {1}`

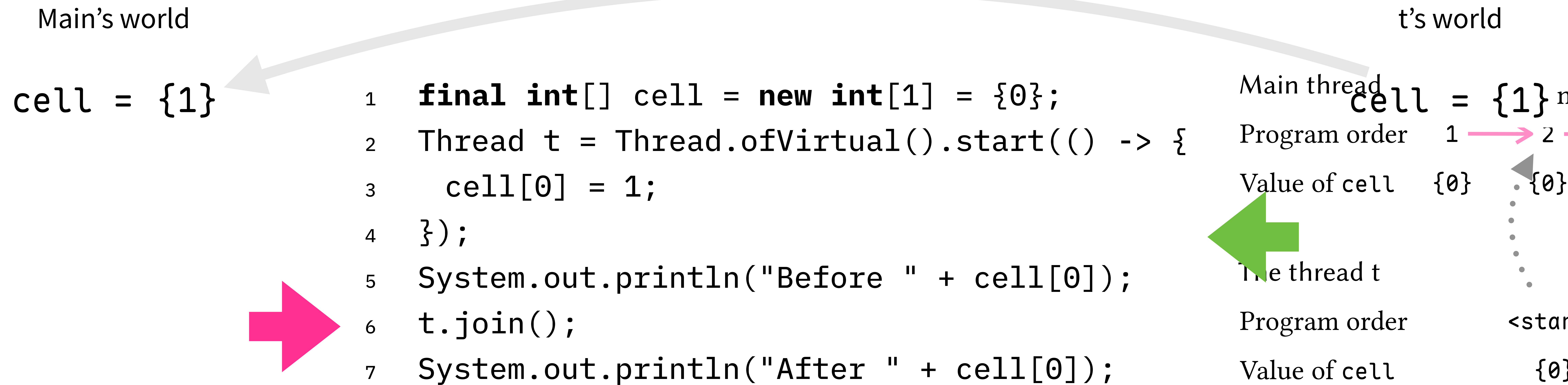
Main thread ne

Program order	1 →	2 →	5
Value of `cell`	{0}	{0}	{0

The thread t

Program order	<start> →	3
Value of `cell`	{0}	{1

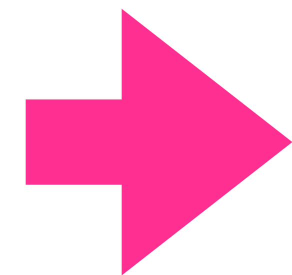
Very simple example:



Very simple example:

Main's world

```
cell = {1}
```



```
1  final int[] cell = new int[1] = {0};
2  Thread t = Thread.ofVirtual().start(() -> {
3    cell[0] = 1;
4  });
5  System.out.println("Before " + cell[0]);
6  t.join();
7  System.out.println("After " + cell[0]);
```

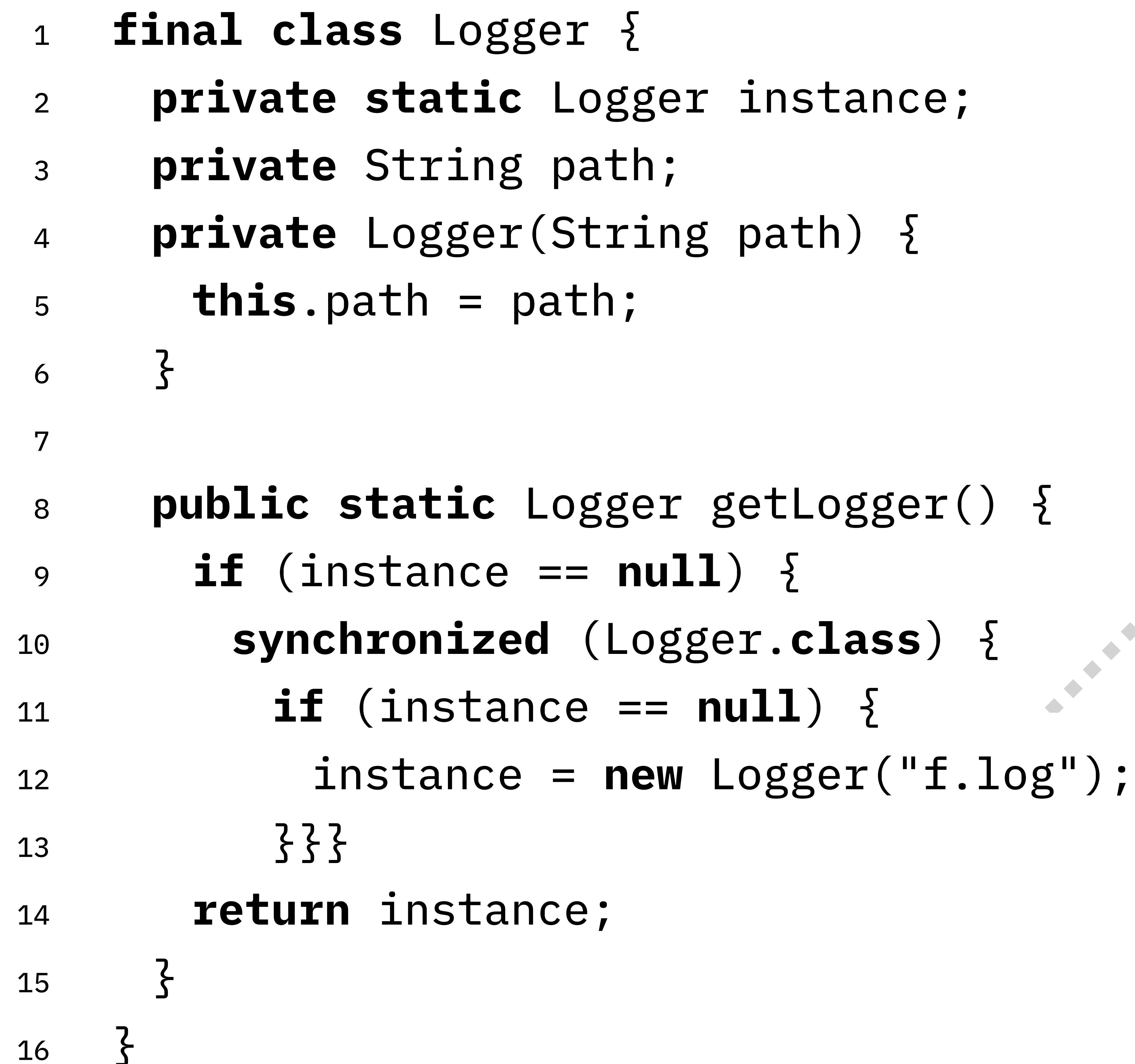
Main thread

Line			
Program order	1	2	5
Value of `cell`	{0}	{0}	{0

The thread t

Program order	<start>	3
Value of `cell`	{0}	{1

Simpler to write a correct concurrent program



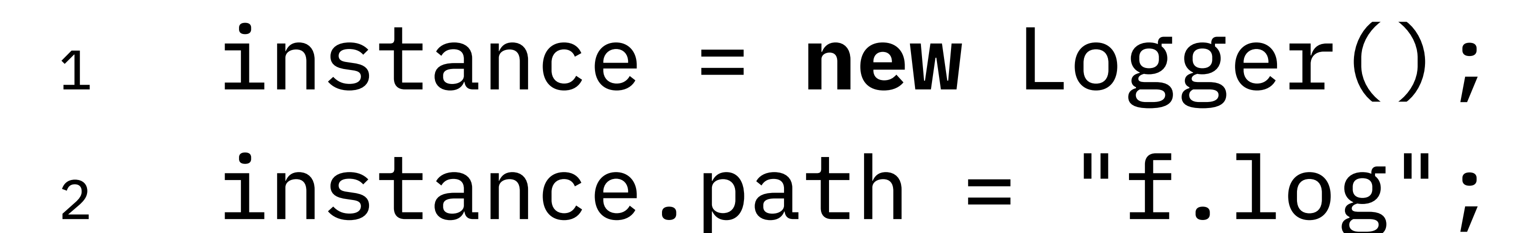
Original ⟶

```
1  instance =
2    new Logger("f.log");
```

After inlining ⟶

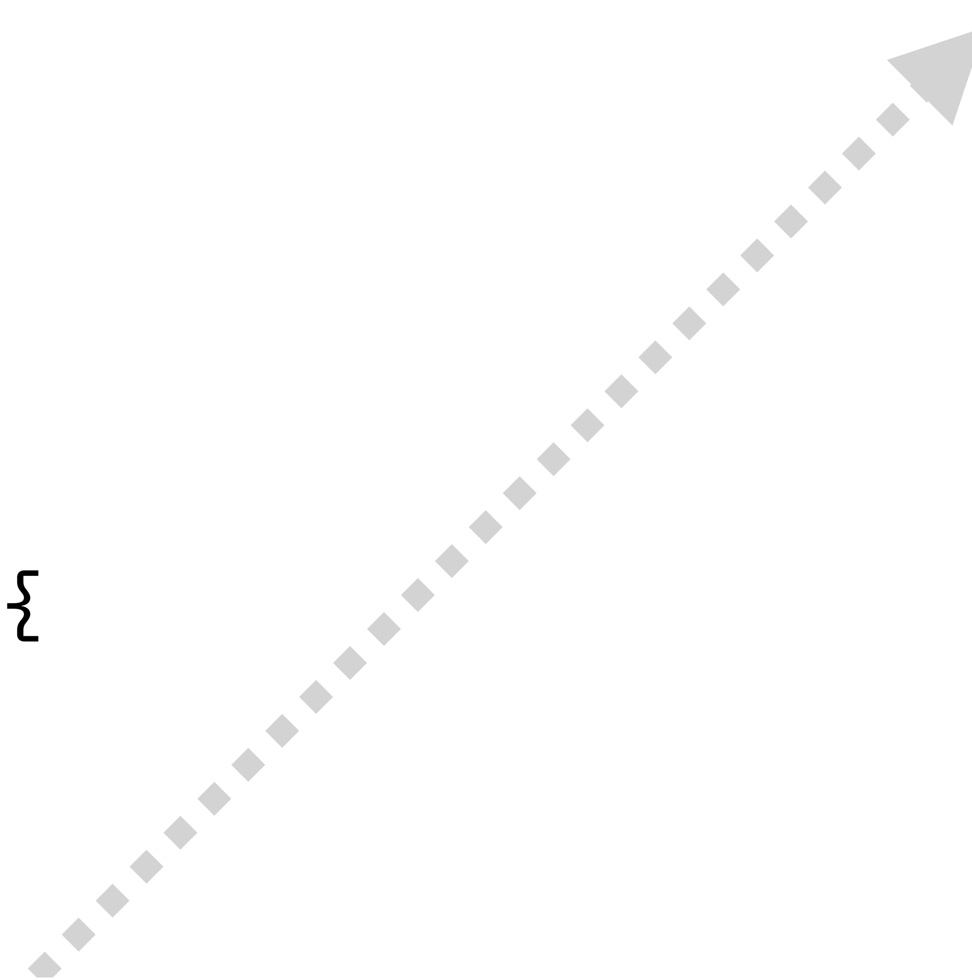
```
1  var tmp = new Logger();
2  tmp.path = "f.log";
3  instance = tmp;
```

After optimisation ⟶



Simpler to write a correct concurrent program

```
1  final class Logger {
2    private static Logger instance;
3    private String path;
4    private Logger(String path) {
5      this.path = path;
6    }
7
8    public static Logger getLogger() {
9      if (instance == null) {
10       synchronized (Logger.class) {
11         if (instance == null) {
12           instance = new Logger("f.log");
13         }}}
14     return instance;
15   }
16 }
```



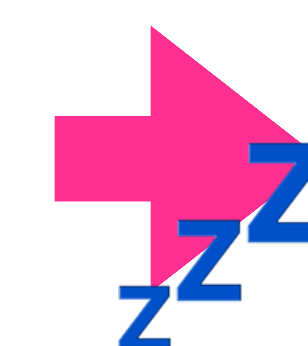
Original ⟶

```
1  instance =
2    new Logger("f.log");
```

After inlining ⟶

```
1  var tmp = new Logger();
2  tmp.path = "f.log";
3  instance = tmp;
```

After optimisation ⟶



```
1  instance = new Logger();
2  instance.path = "f.log";
```

Fixing the program

```java
1  final class Logger {
2    private static Logger instance;  volatile
3    private String path;  final
4    private Logger(String path) {
5      this.path = path;
6    }
7
8    public static Logger getLogger() {
9      if (instance == null) {
10       synchronized (Logger.class) {
11         if (instance == null) {
12           instance = new Logger("f.log");
13         }}}
14     return instance;
15   }
16 }
```

Original ⟶

```java
1  instance =
2    new Logger("f.log");
```

After inlining ⟶

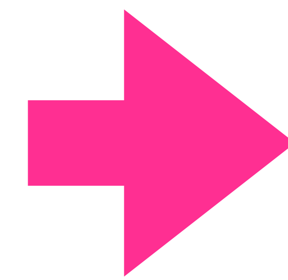
```java
1  var tmp = new Logger();
2  tmp.path = "f.log";
3  instance = tmp;
```

After optimisation ⟶

```java
1  instance = new Logger();
2  instance.path = "f.log";
```

Simpler to write a correct concurrent program

```
instance = null
```



```
1  final class Logger {
2    private static Logger instance;
3    private String path;
4    private Logger(String path) {
5      this.path = path;
6    }
7
8    public static Logger getLogger() {
9      if (instance == null) {
10       synchronized (Logger.class) {
11         if (instance == null) {
12           instance = new Logger("f.log");
13         }}}
14     return instance;
15   }
16 }
```

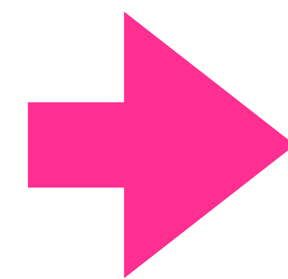
```
instance = null
```

After optimisation →

```
1  instance = new Logger();
2  instance.path = "f.log";
```

Simpler to write a correct concurrent program

instance = null



```
1  final class Logger {
2    private static Logger instance;
3    private String path;
4    private Logger(String path) {
5      this.path = path;
6    }
7
8    public static Logger getLogger() {
9      if (instance == null) {
10       synchronized (Logger.class) {
11         if (instance == null) {
12           instance = new Logger("f.log");
13         }}}
14     return instance;
15   }
16 }
```

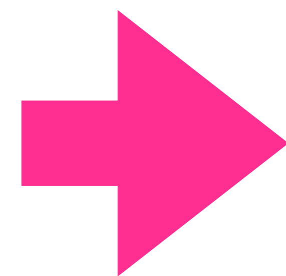
instance = null

After optimisation ⟶

```
1  instance = new Logger();
2  instance.path = "f.log";
```

Simpler to write a correct concurrent program

`instance = null`



```
1  final class Logger {
2    private static Logger instance;
3    private String path;
4    private Logger(String path) {
5      this.path = path;
6    }
7
8    public static Logger getLogger() {
9      if (instance == null) {
10       synchronized (Logger.class) {
11         if (instance == null) {
12           instance = new Logger("f.log");
13         }}}
14     return instance;
15   }
16 }
```

`instance = null`

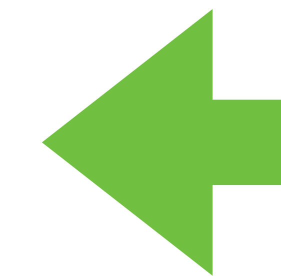
After optimisation →

```
1  instance = new Logger();
2  instance.path = "f.log";
```

Simpler to write a correct concurrent program

instance = null

```
1  final class Logger {
2    private static Logger instance;
3    private String path;
4    private Logger(String path) {
5      this.path = path;
6    }
7
8    public static Logger getLogger() {
9      if (instance == null) {
10       synchronized (Logger.class) {
11         if (instance == null) {
12           instance = new Logger("f.log");
13         }}}
14     return instance;
15   }
16 }
```



instance = null

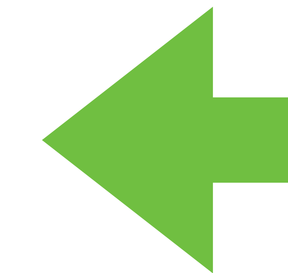
After optimisation ⟶

```
1  instance = new Logger();
2  instance.path = "f.log";
```

Simpler to write a correct concurrent program

```
instance = 0x…
```

```
1  final class Logger {
2    private static Logger instance;
3    private String path;
4    private Logger(String path) {
5      this.path = path;
6    }
7
8    public static Logger getLogger() {
9      if (instance == null) {
10       synchronized (Logger.class) {
11         if (instance == null) {
12           instance = new Logger("f.log");
13         }}}
14     return instance;
15   }
16 }
```



```
instance = null
```

After optimisation ⟶

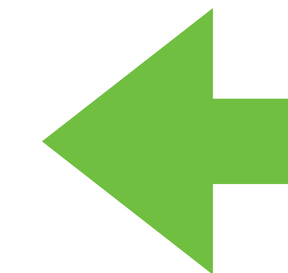
```
1  instance = new Logger();
2  instance.path = "f.log";
```

Simpler to write a correct concurrent program

instance = 0x…

```
1  final class Logger {
2    private static Logger instance;
3    private String path;
4    private Logger(String path) {
5      this.path = path;
6    }
7
8    public static Logger getLogger() {
9      if (instance == null) {
10       synchronized (Logger.class) {
11         if (instance == null) {
12           instance = new Logger("f.log");
13         }}}
14     return instance;
15   }
16 }
```

instance = null



After optimisation →

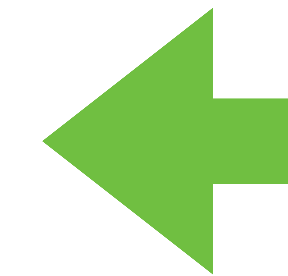
```
1  instance = new Logger();
2  instance.path = "f.log";
```

Simpler to write a correct concurrent program

instance = 0x…

```
1  final class Logger {
2    private static Logger instance;
3    private String path;
4    private Logger(String path) {
5      this.path = path;
6    }
7
8    public static Logger getLogger() {
9      if (instance == null) {
10       synchronized (Logger.class) {
11         if (instance == null) {
12           instance = new Logger("f.log");
13         }}}
14     return instance;
15   }
16 }
```

instance = null



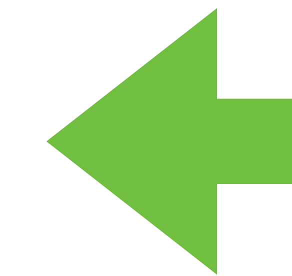
After optimisation →

```
1  instance = new Logger();
2  instance.path = "f.log";
```

Simpler to write a correct concurrent program

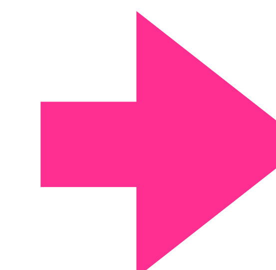
instance = 0x…

```
1  final class Logger {
2    private static Logger instance;
3    private String path;
4    private Logger(String path) {
5      this.path = path;
6    }
7
8    public static Logger getLogger() {
9      if (instance == null) {
10       synchronized (Logger.class) {
11         if (instance == null) {
12           instance = new Logger("f.log");
13         }}}
14     return instance;
15   }
16 }
```



instance = null

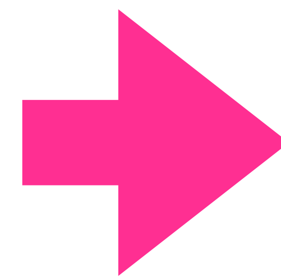
After optimisation →



```
1  instance = new Logger();
2  instance.path = "f.log";
```

Simpler to write a correct concurrent program

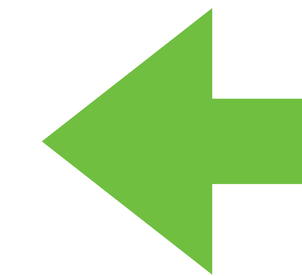
instance = 0x…



```
final class Logger {
  private static Logger instance;
  private String path;
  private Logger(String path) {
    this.path = path;
  }

  public static Logger getLogger() {
    if (instance == null) {
      synchronized (Logger.class) {
        if (instance == null) {
          instance = new Logger("f.log");
        }}}
    return instance;
  }
}
```

instance = null



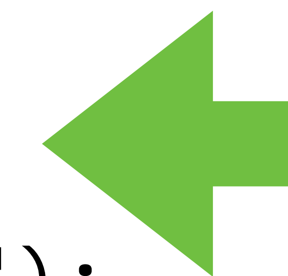
After optimisation ⟶

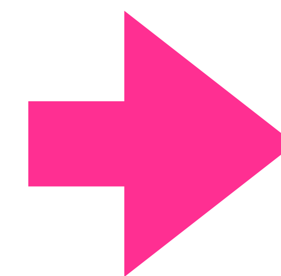
```
instance = new Logger();
instance.path = "f.log";
```

Simpler to write a correct concurrent program

instance = 0x…

```
1  final class Logger {
2    private static Logger instance;
3    private String path;
4    private Logger(String path) {
5      this.path = path;
6    }
7
8    public static Logger getLogger() {
9      if (instance == null) {
10       synchronized (Logger.class) {
11         if (instance == null) {
12           instance = new Logger("f.log");
13         }}}
14     return instance;
15   }
16 }
```





instance = 0x… / null

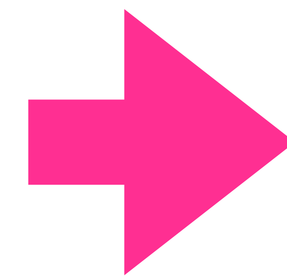
After optimisation ⟶

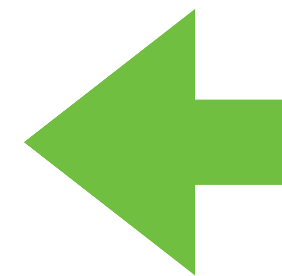
```
1  instance = new Logger();
2  instance.path = "f.log";
```

Simpler to write a correct concurrent program

```
instance = 0x…
```

```
1  final class Logger {
2    private static Logger instance;
3    private String path;
4    private Logger(String path) {
5      this.path = path;
6    }
7
8    public static Logger getLogger() {
9      if (instance == null) {
10       synchronized (Logger.class) {
11         if (instance == null) {
12           instance = new Logger("f.log");
13       }}}
14     return instance;
15   }
16 }
```





```
instance = 0x…
```

After optimisation ⟶

```
1  instance = new Logger();
2  instance.path = "f.log";
```

What DRF Java Is and Is Not

DRF Java is not ”just” dynamic data-race detection

Structurally guaranteed to be free from read-write races

Several programs that are considered incorrectly synchronised under the JMM are correctly synchronised in DRF Java

– As exemplified by the singleton example (double-checked locking without volatiles)

DRF Java is not software transactional memory

No imposed nesting

We can do e.g. hand-over-hand locking

There is no rollback

. . .

Semantic Overview

We are exploring two different models in parallel that differ in how threads communicate changes with each other

Linear model

- All synchronisation interacts with a ”global store”
- Faster convergence — aka fails faster
- More leakage

Branching model

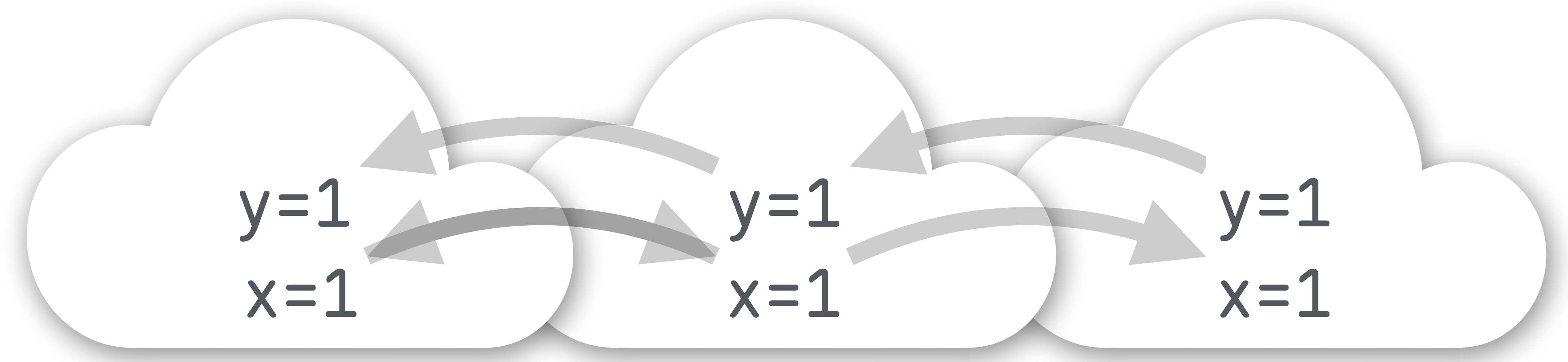
- Locks, volatile fields, and threads have their own stores
- Permits more divergence
- Less leakage

Sync. Action		Re-interpretation
Object	Action	
thread	spawn t' join t'	
lock	lock l unlock l	
volatile	read v write v	

Sync. Action	Re-interpretation
start end	

(Ignoring thread interrupt for now)

Diagrammatic Notation Primer



g

t_1	t_2
<start> x = 1 spawn t_2 join t_2	<start> y = 1 <end>

t_1

<start>

x=1

spawn t_2

join t_2

t_2

<start>

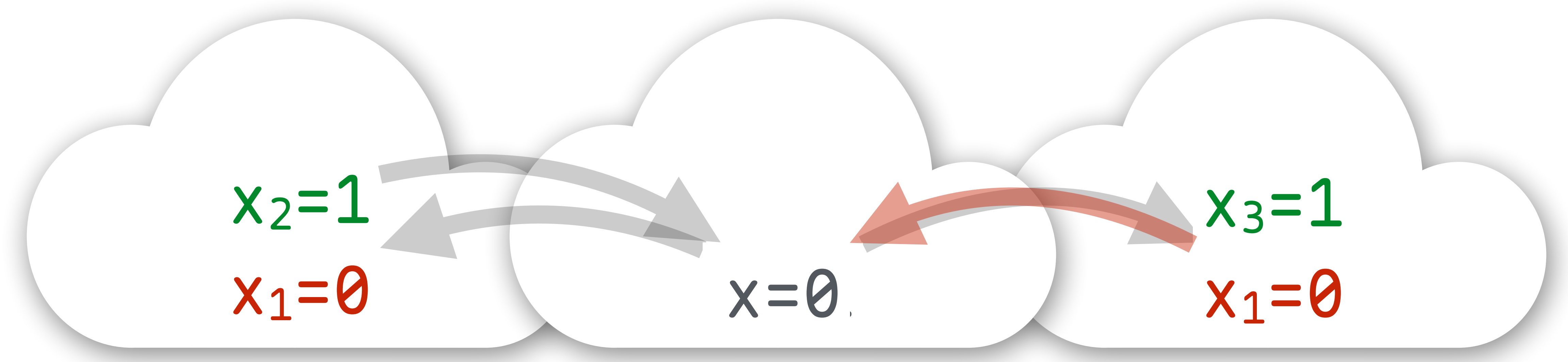
y=1

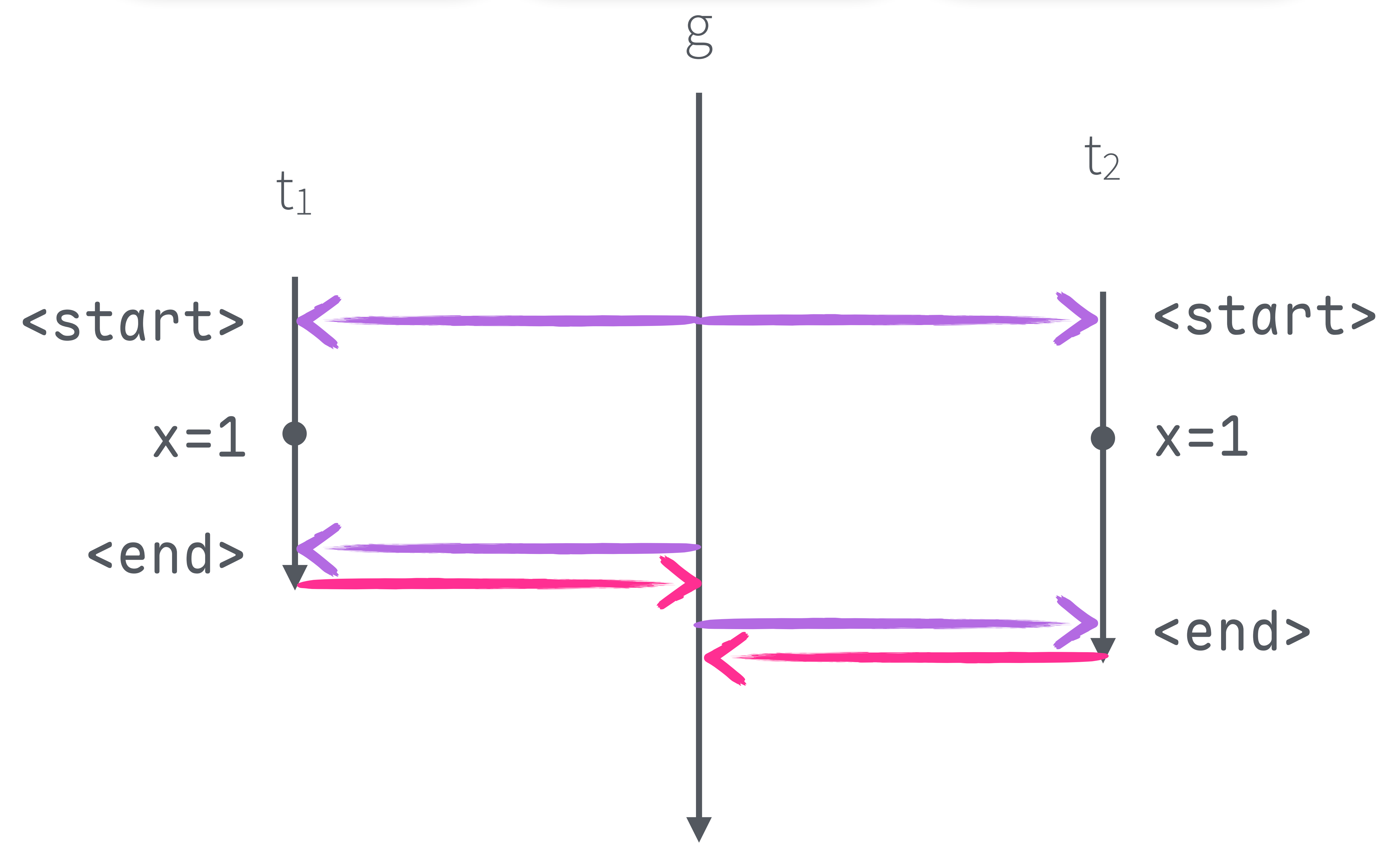
<end>

Detecting Conflicts

Counter

3





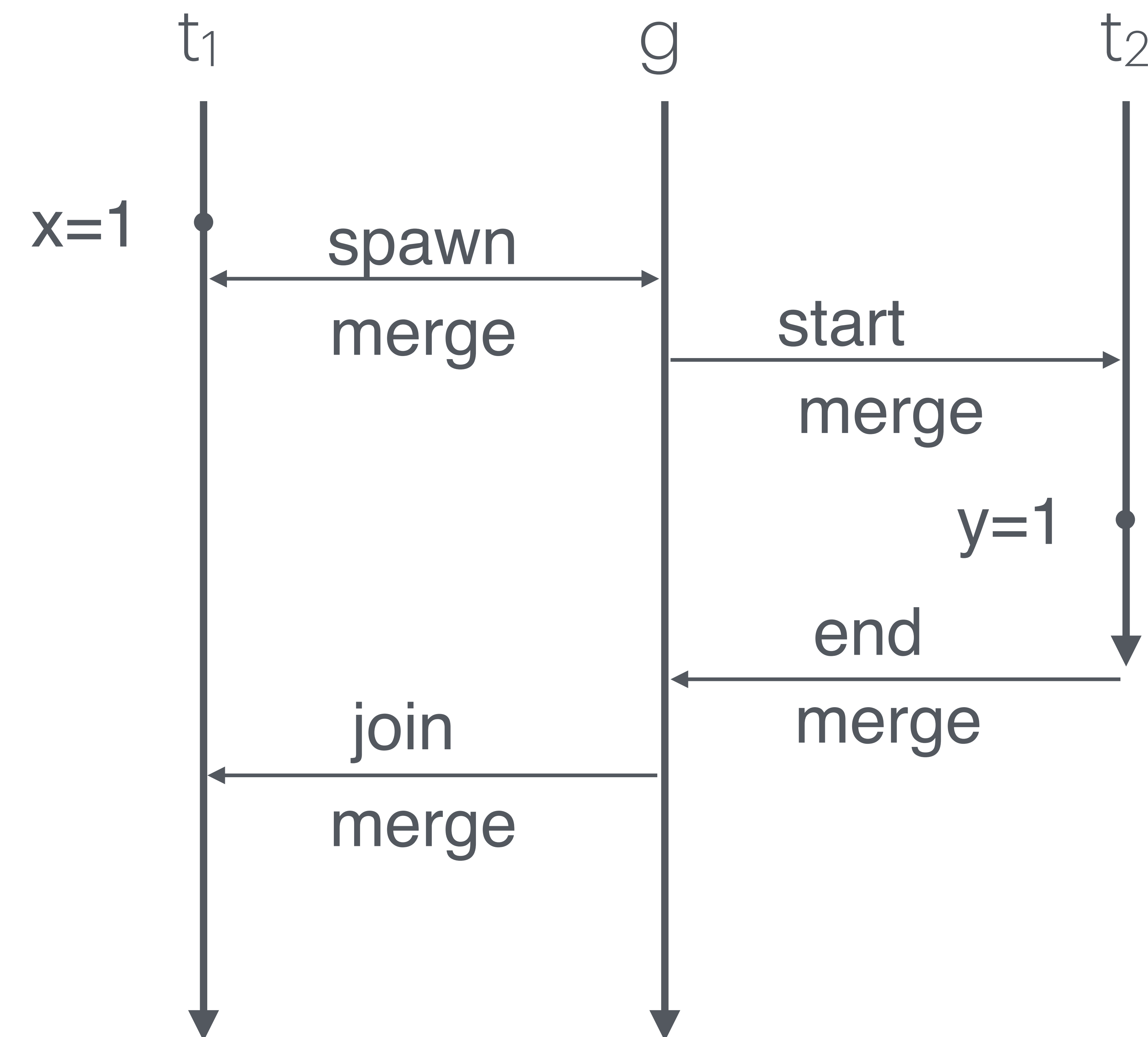
t_1	t_2
<start> x = 1 <end>	<start> x = 1 <end>

❌ 2 ≠ 1 ❌

Example 1

Linear — correctly synchronised, changes different parts of the state

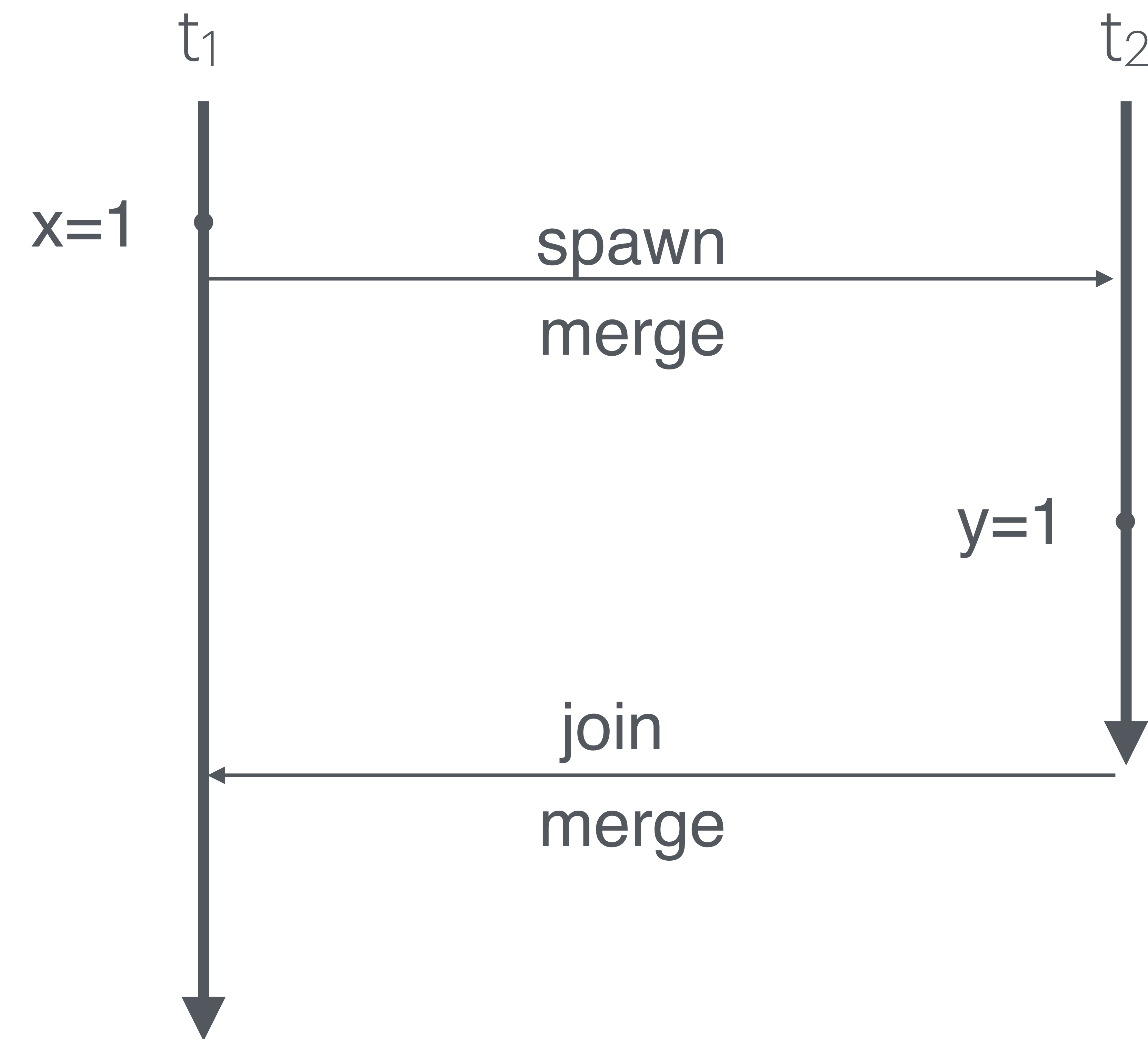
t_1	t_2
x = 1 spawn t_2 join t_2	y = 1



Example 1

Branching — correctly synchronised, changes different parts of the state

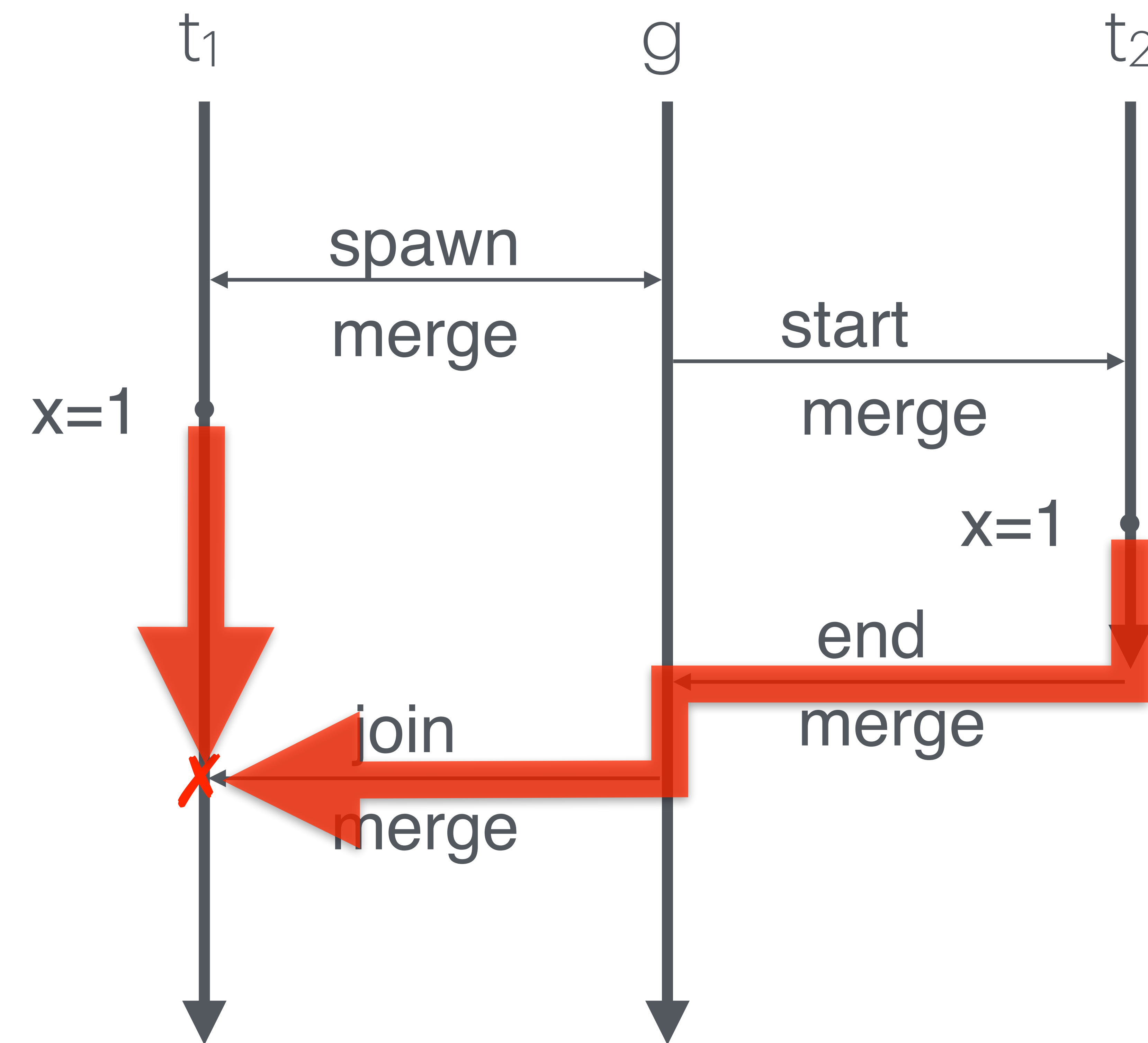
t_1	t_2
x = 1 spawn t_2 join t_2	y = 1



Example 2

Linear — **incorrectly** synchronised, write-write race on x

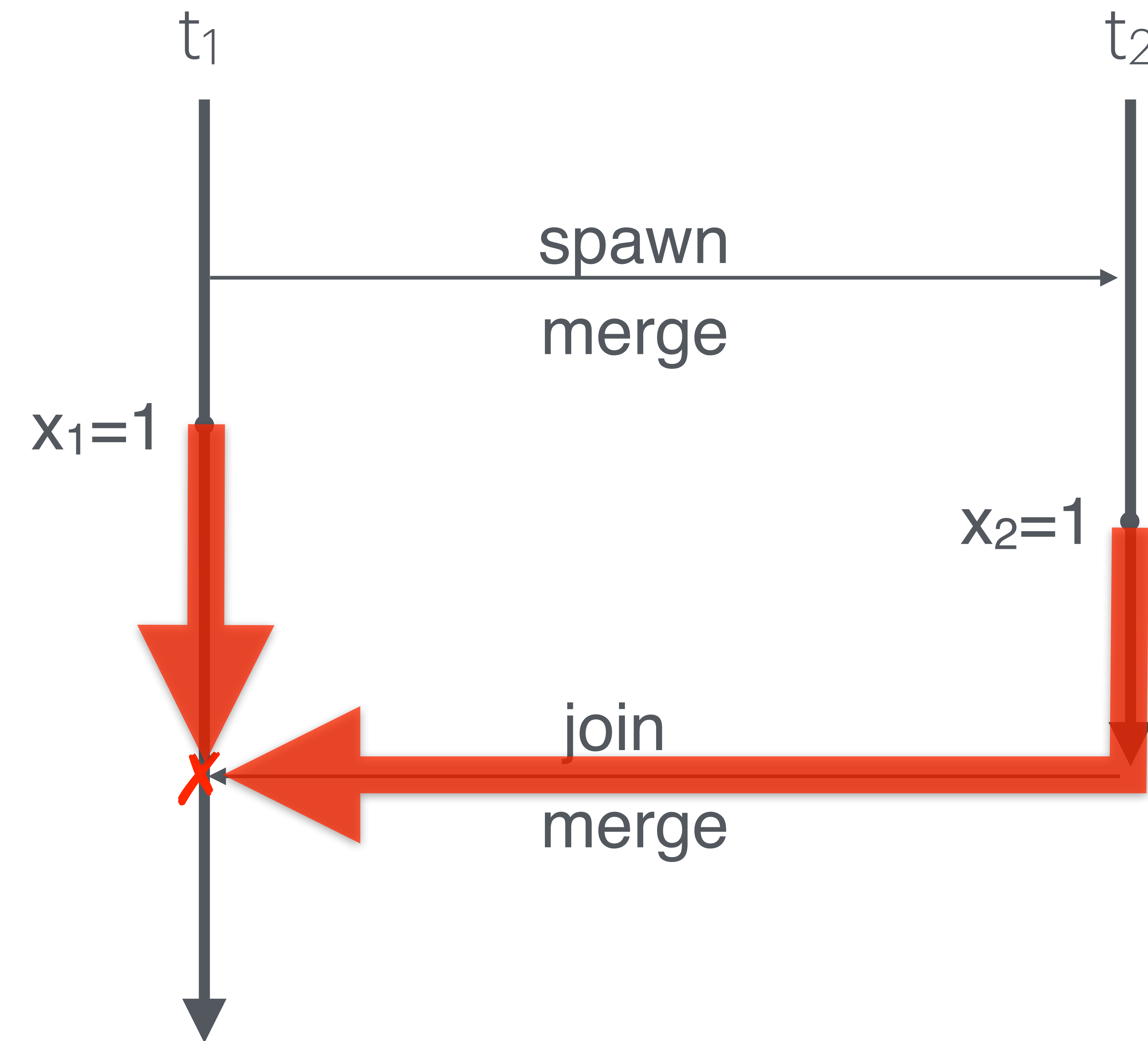
t_1	t_2
spawn t_2 x = 1 join t_2	x = 1



Example 2

Branching — **incorrectly** synchronised, write-write race on x

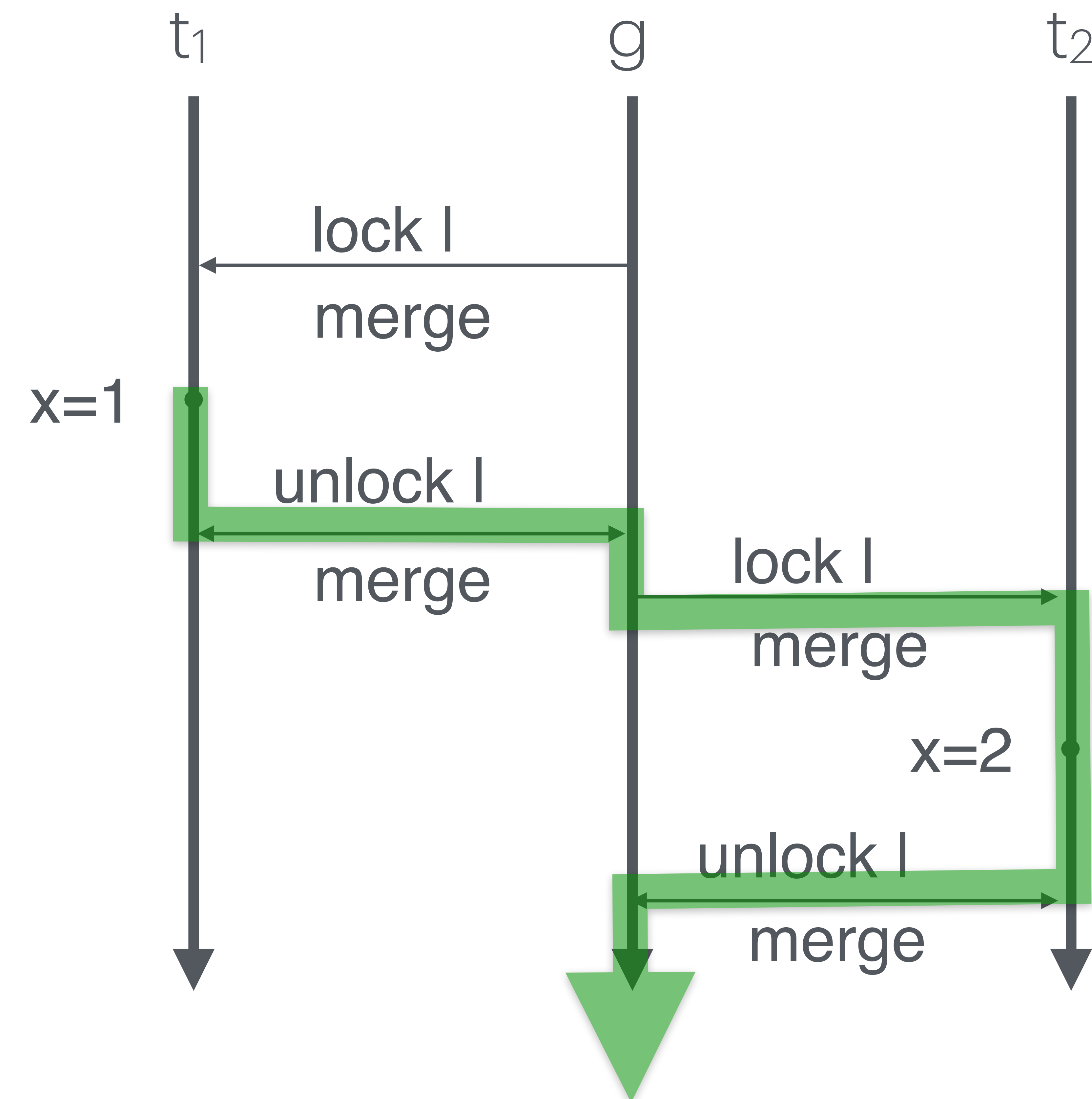
t_1	t_2
spawn t_2 x = 1 join t_2	x = 1



Example 3

Linear — correctly synchronised, lock l orders writes to x

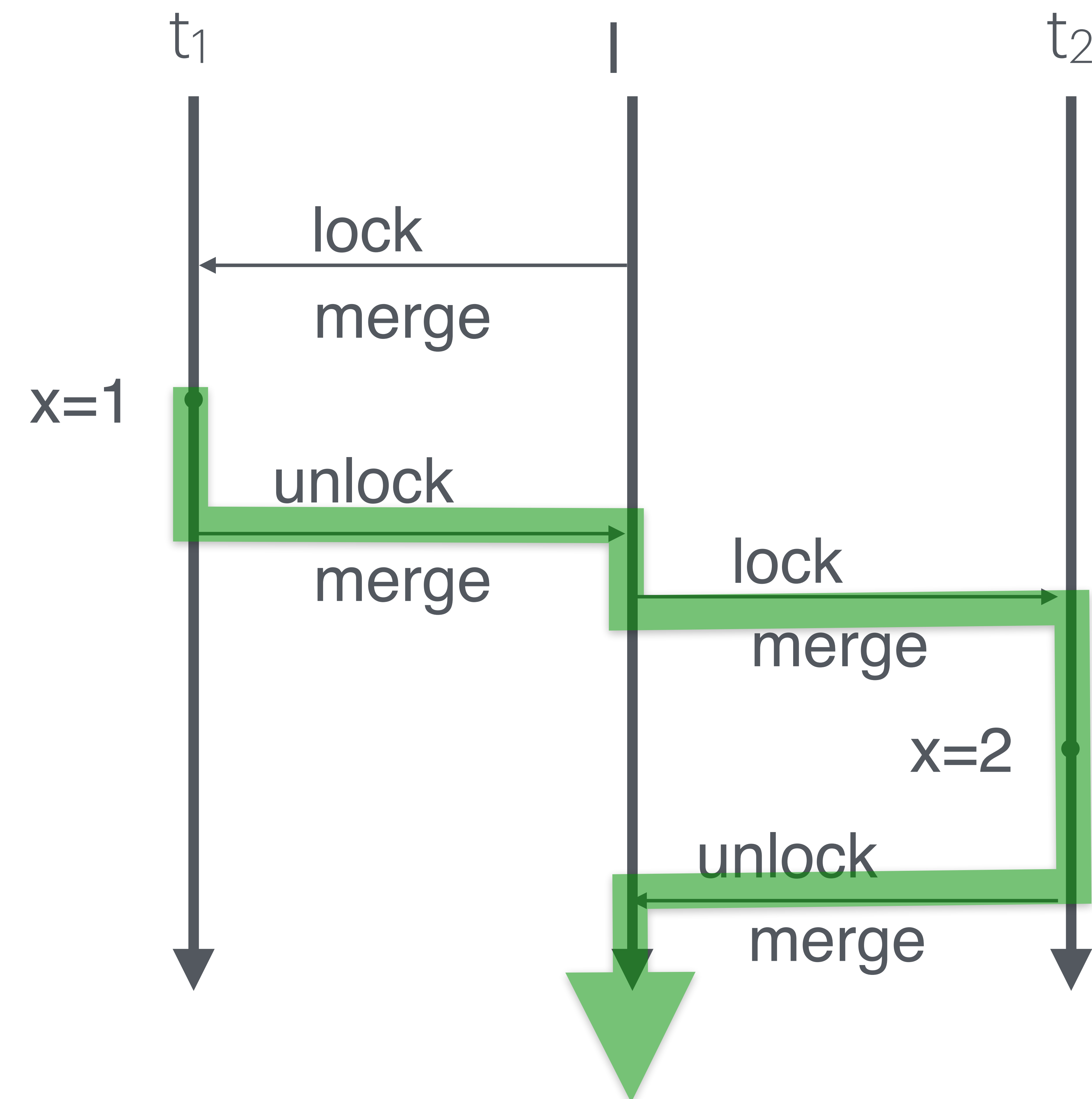
t_1	t_2
lock l x = 1 unlock l	lock l x = 2 unlock l



Example 3

Branching — correctly synchronised, lock l orders writes to x

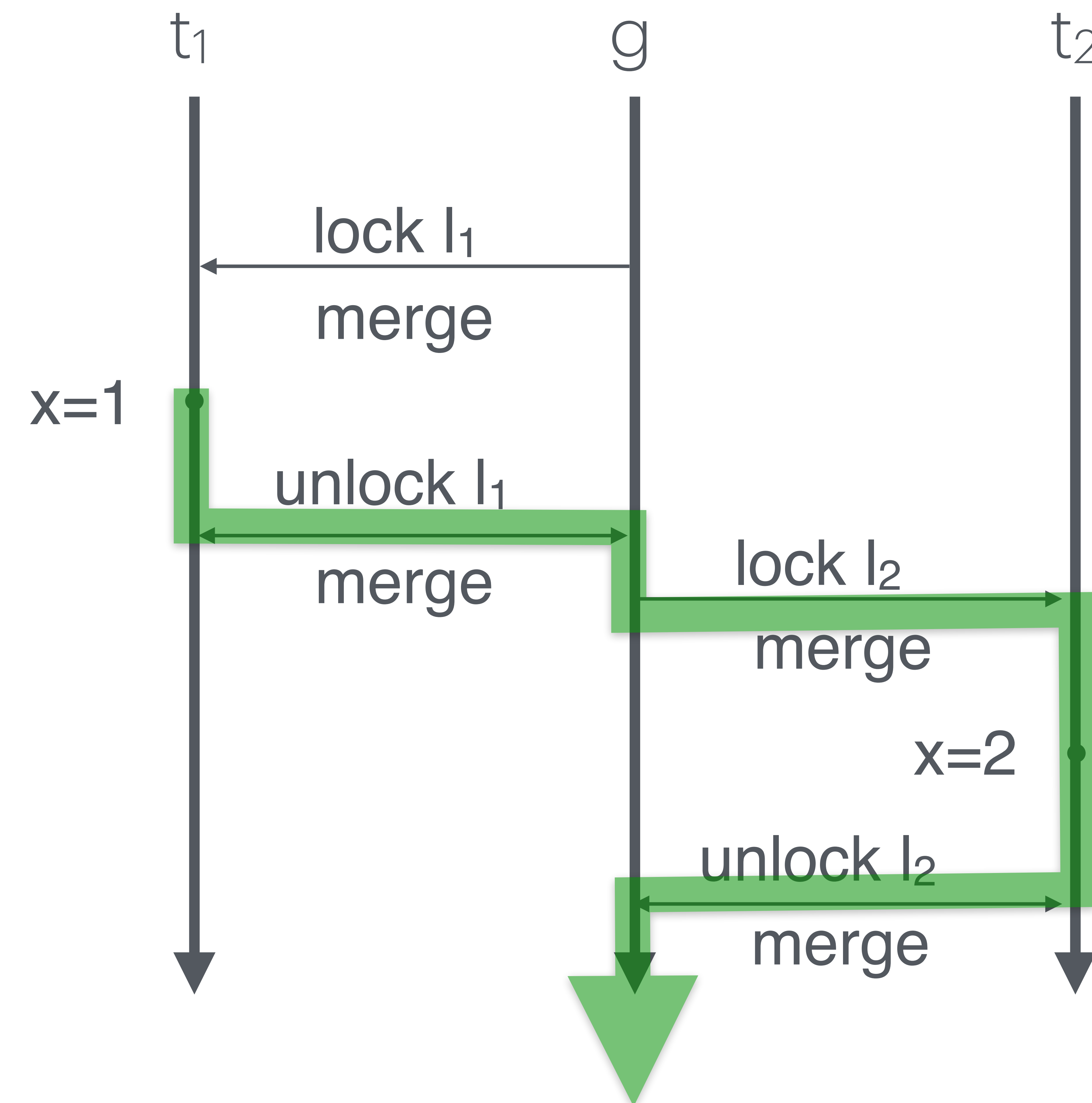
t_1	t_2
lock l x = 1 unlock l	lock l x = 2 unlock l



Example 4

Linear — **incorrectly** synchronised, write-write race on x under **different** locks ("lucky" with timing)

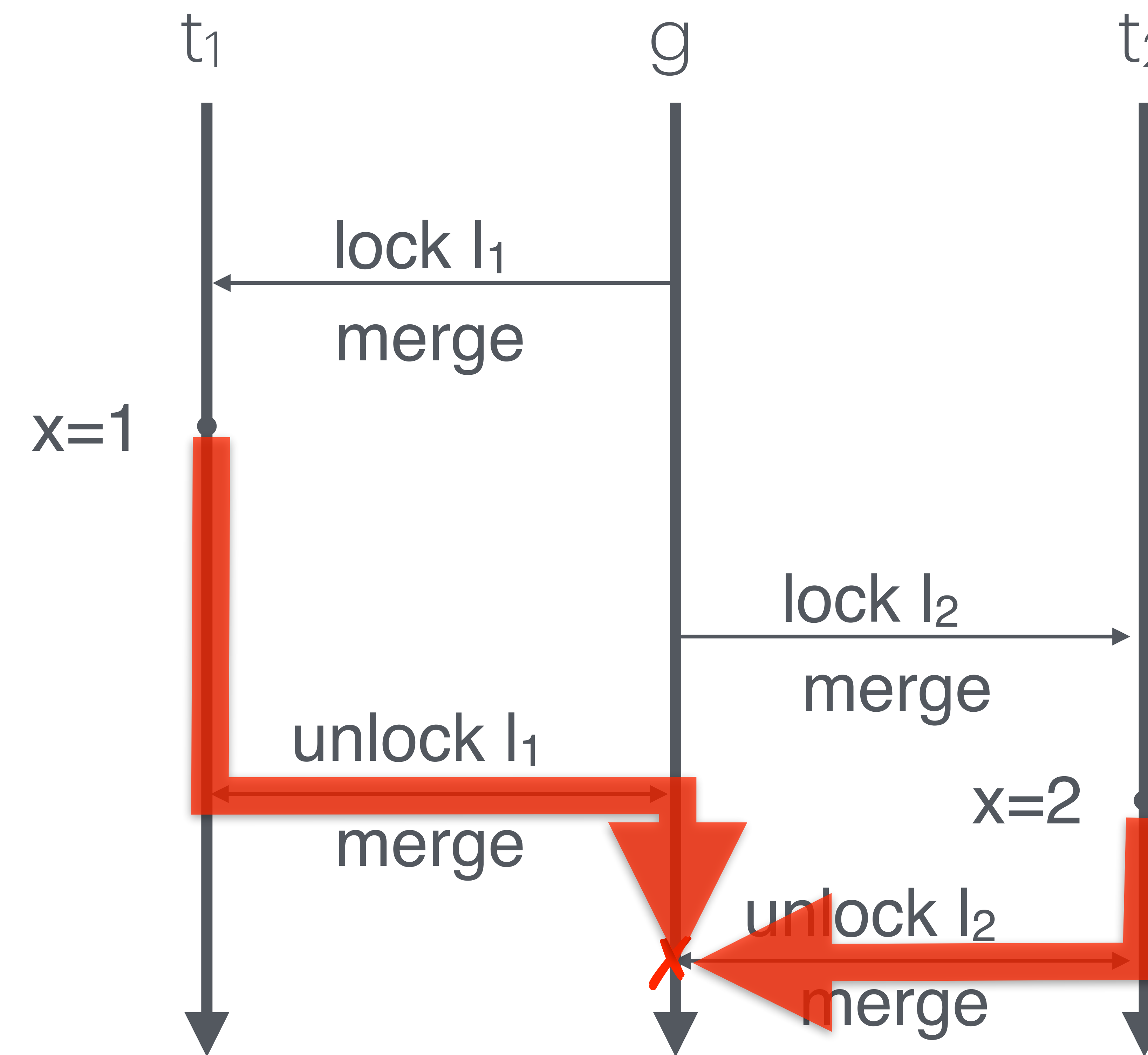
t_1	t_2
lock l_1 $x = 1$ unlock l_1	lock l_2 $x = 2$ unlock l_2



Example 4

Linear — **incorrectly** synchronised, write-write race on x under **different** locks (not "lucky")

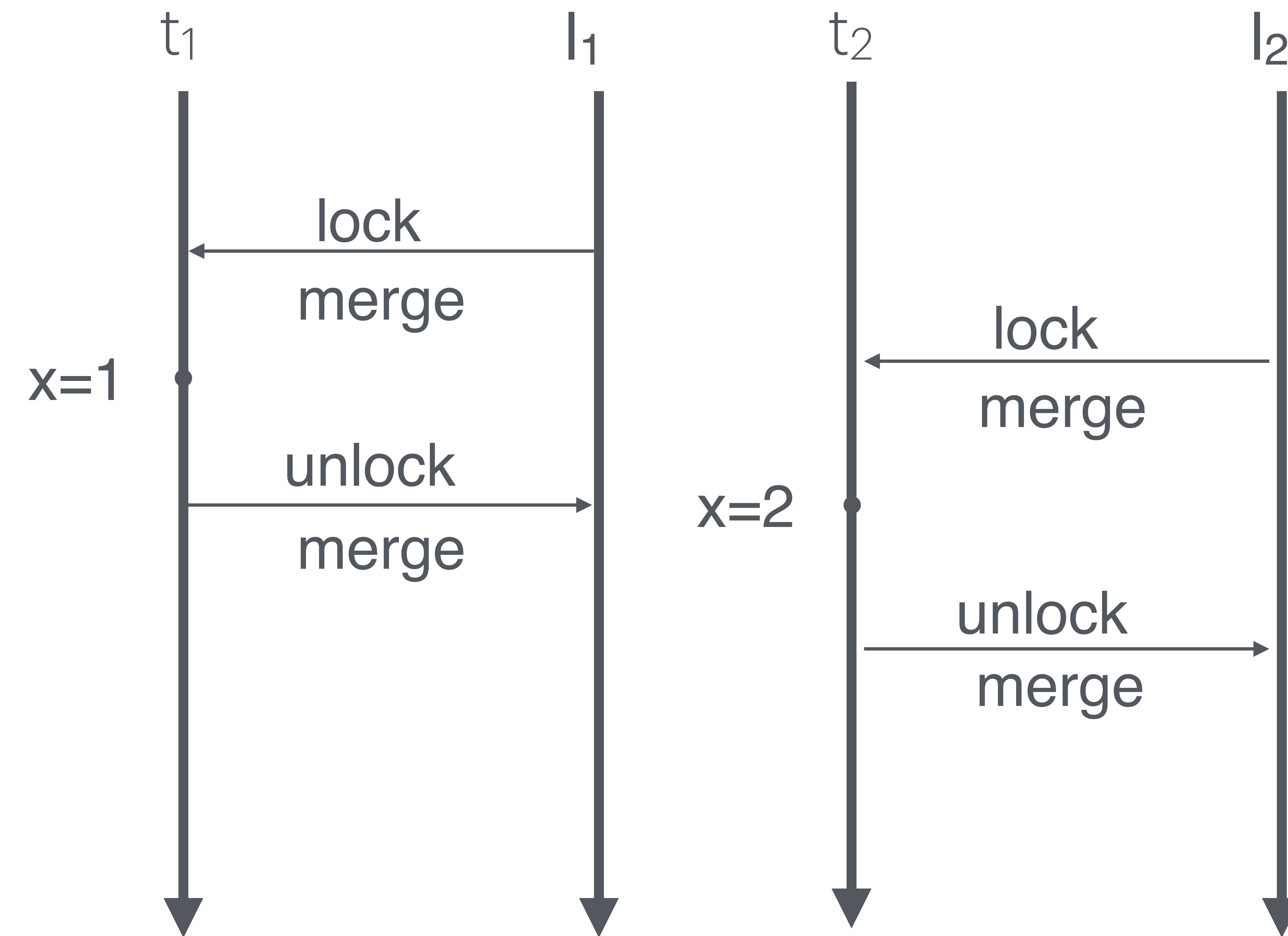
t_1	t_2
lock l_1 x = 1 unlock l_1	lock l_2 x = 2 unlock l_2



Example 4

Branching — **incorrectly** synchronised, write-write race on x under **different** locks (thread isolated)

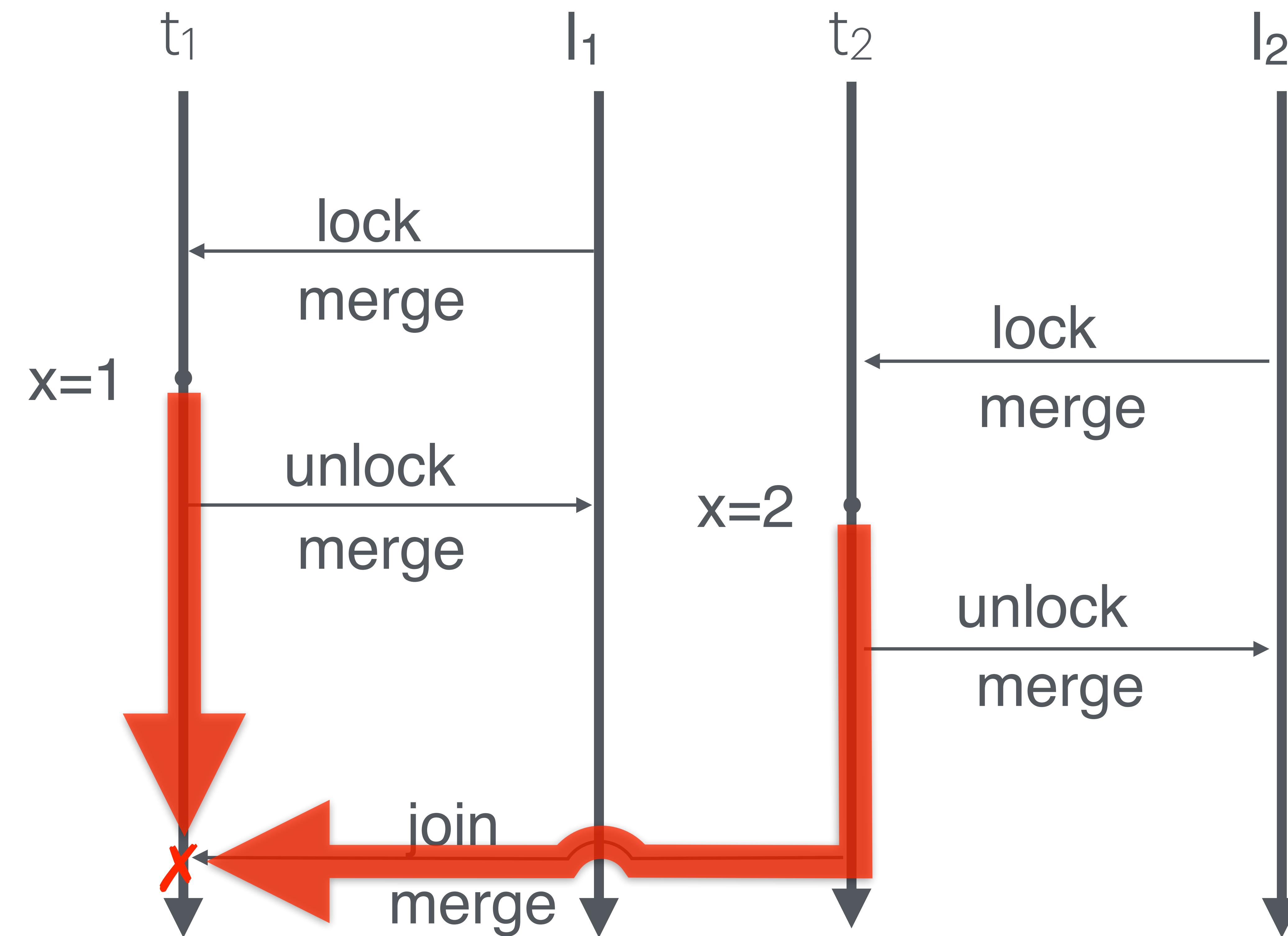
t_1	t_2
lock l_1 $x = 1$ unlock l_1	lock l_2 $x = 2$ unlock l_2



Example 5

Branching — **incorrectly** synchronised, write-write race on x under **different** locks (thread join)

t_1	t_2
lock l_1 $x = 1$ unlock l_1 join t_2	lock l_2 $x = 2$ unlock l_2



Some Open Design Choices

- Example 4 shows that two threads that never synchronise can hold diverging world views
- Example 5 showed that two such threads joining (or synchronising) will eventually raise a conflict
- In Java, there is a synthetic join of all threads when the VM tears down

  Should we raised a conflict exception here or not?

- Our approach: a ”ConcurrencyManager”

  Gives programmer some control over what should happen on a conflict

  Not intended for fine-grained conflict resolution

The Leakage Problem

(Initially $x = 0$)

t_1	t_2
lock l_1 $x = 1$ unlock l_1	lock l_2 print(x) unlock l_2

(Initially $x = 0$, $y = 0$)

t_1	t_2
lock l_1 $x = 1$ lock l_2 $y = 2$ unlock l_2	lock l_2 print(x) unlock l_2

Status

Completed

Informal model

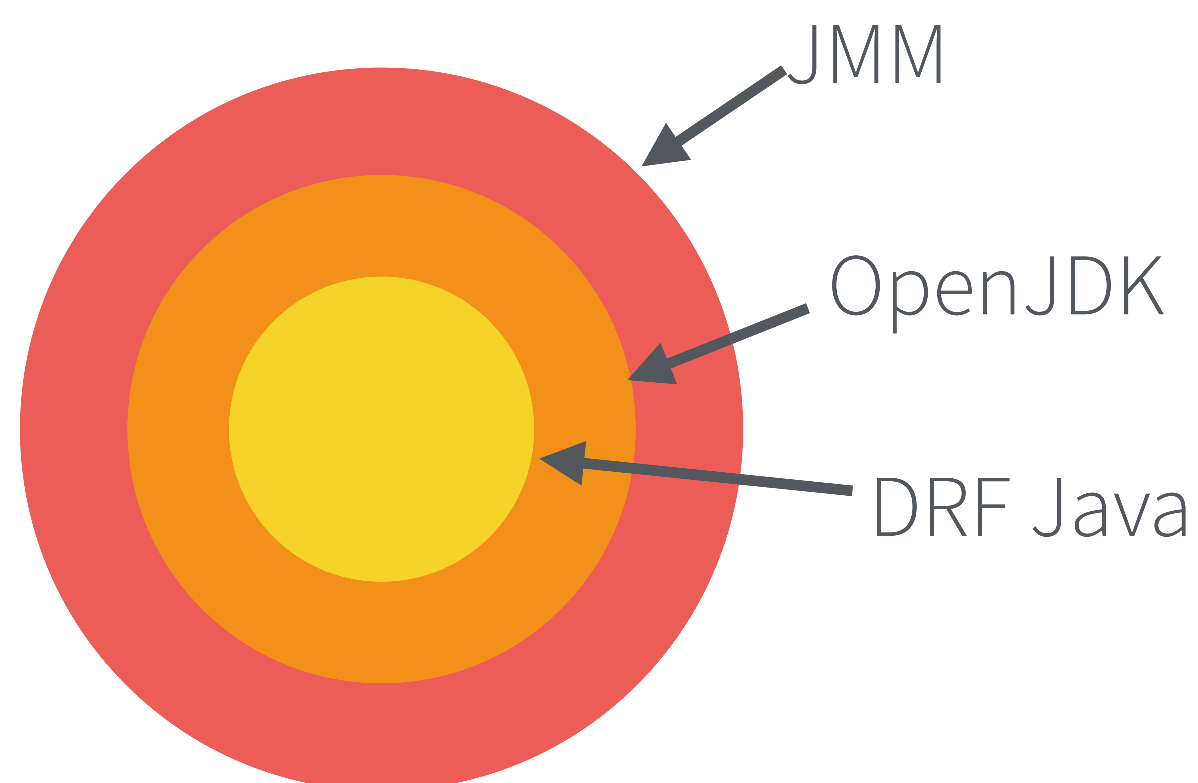
Diagrams for communication

Toy tool for writing litmus test programs that simulate execution

Pen & Paper + Mechanised formalism of our models

On-going

Implementation in the OpenJDK 26 interpreter



$$\frac{T(t) = (\mathcal{N}, e, \emptyset, c)}{\langle T\,;H\,;L\rangle \longrightarrow \langle T[\,t \mapsto_{ts} \mathcal{R}\,]\,;H\,;L\rangle}\ \textsc{Start}$$

$$\frac{T(t) = (\mathcal{R}, r, \delta, c) \qquad \langle T\,;H\,;L\rangle \xRightarrow{\text{commit}_t} \langle T'\,;H'\,;L\rangle}{\langle T\,;H\,;L\rangle \longrightarrow \langle T'[\,t \mapsto_{ts} \mathcal{T}\,]\,;H'\,;L\rangle}\ \textsc{End}$$

$$\frac{T(t) = (\mathcal{R}, e, \delta, c) \qquad e \xRightarrow{\text{read } i} K \qquad v = \mathsf{read}(H, \delta, c, i)}{\langle T\,;H\,;L\rangle \longrightarrow \langle T[\,t \mapsto_{e} K\langle v\rangle\,]\,;H\,;L\rangle}\ \textsc{Read}$$

$$\frac{T(t) = (\mathcal{R}, e, \delta, c) \qquad e \xRightarrow{\text{write } i\ v} K \qquad (H', \delta') = \mathsf{write}(H, \delta, i, v) \qquad T' = T[\,t \mapsto (\mathcal{R}, K\langle\epsilon\rangle, \delta', c)\,]}{\langle T\,;H\,;L\rangle \longrightarrow \langle T'\,;H'\,;L\rangle}\ \textsc{Write}$$

$$\frac{T(t) = (\mathcal{R}, e, \delta, c) \qquad e \xRightarrow{\text{spawn } e_s} K \qquad T' = T[\,t \mapsto_e K\langle t'\rangle\,][\,t' \mapsto (\mathcal{N}, e_s, \emptyset, 0)\,] \qquad \langle T'\,;H\,;L\rangle \xRightarrow{\text{push}_{t,t'}} \langle T''\,;H'\,;L\rangle \qquad t' \notin \mathsf{dom}(T)}{\langle T\,;H\,;L\rangle \longrightarrow \langle T''\,;H'\,;L\rangle}\ \textsc{Spawn}$$

$$\frac{T(t) = (\mathcal{R}, e, \delta, c) \qquad e \xRightarrow{\text{join } t'} K \qquad T(t') = (\mathcal{T}, r, _, _) \qquad \langle T\,;H\,;L\rangle \xRightarrow{\text{pull}_{t,t'}} \langle T'\,;H'\,;L\rangle}{\langle T\,;H\,;L\rangle \longrightarrow \langle T'[\,t \mapsto_e K\langle r\rangle\,]\,;H'\,;L\rangle}\ \textsc{Join}$$

$$\frac{T(t) = (\mathcal{R}, e, \delta, c) \qquad e \xRightarrow{\text{lock } \ell} K \qquad L(\ell).ls = \mathsf{free} \qquad \langle T\,;H\,;L\rangle \xRightarrow{\text{pull}_{t,\ell}} \langle T'\,;H'\,;L\rangle}{\langle T\,;H\,;L\rangle \longrightarrow \langle T'[\,t \mapsto_e K\langle\epsilon\rangle\,]\,;H'\,;L[\,\ell \mapsto_{ls} \mathsf{held}(t)\,]\rangle}\ \textsc{Lock}$$

$$\frac{T(t) = (\mathcal{R}, e, \delta, c) \qquad e \xRightarrow{\text{unlock } \ell} K \qquad L(\ell).ls = \mathsf{held}(t) \qquad \langle T\,;H\,;L\rangle \xRightarrow{\text{push}_{t,\ell}} \langle T'\,;H'\,;L'\rangle}{\langle T\,;H\,;L\rangle \longrightarrow \langle T'[\,t \mapsto_e K\langle\epsilon\rangle\,]\,;H'\,;L'[\,\ell \mapsto_{ls} \mathsf{free}\,]\rangle}\ \textsc{Unlock}$$

(b) Concurrency Semantics.

```
// Construct the singleton pattern:
// …

instance = 0;

$t1 = spawn {
  if (instance == 0) {
    lock l;
    if (instance == 0) {
      instance = 100;
    }
    unlock l;
  }
  assert(instance == 100);
};


if (instance == 0) {
  lock l;
  if (instance == 0) {
    instance = 100;
  }
  unlock l;
}
assert(instance == 100);

join $t1;
assert(instance == 100);
```

Thread #0

Winstance = 0

Spawn 1

Rinstance = 0

Lock l

Rinstance = 0

Thread #1

Rinstance = 0

Lock l

Rinstance = 100

Unlock l

Rinstance = 100

rf

sync

rf

rf

rf

rf

sync

Can this be sufficiently performant?

- We do not know!
- Next step: instrument OpenJDK to measure various events to give an idea of the overhead costs
- This also enables some optimisations that can be used to off-set the costs

141

Fixing the program

```
1  final class Logger {
2    private static Logger instance; volatile
3    private String path; final
4    private Logger(String path) {
5      this.path = path;
6    }
7
8    public static Logger getLogger() {
9      if (instance == null) {
10       synchronized (Logger.class) {
11         if (instance == null) {
12           instance = new Logger("f.log");
13         }}}
14     return instance;
15   }
16 }
```

Original ⟶

```
1  instance =
2    new Logger("f.log");
```

After inlining ⟶

```
1  var tmp = new Logger();
2  tmp.path = "f.log";
3  instance = tmp;
```

After reordering

```
1  instance = new Logger();
2  instance.path = "f.log";
```