SideTrack: Generalizing Dynamic Atomicity Analysis

Caitlin Sadowski
Jaeheon Yi
Cormac Flanagan

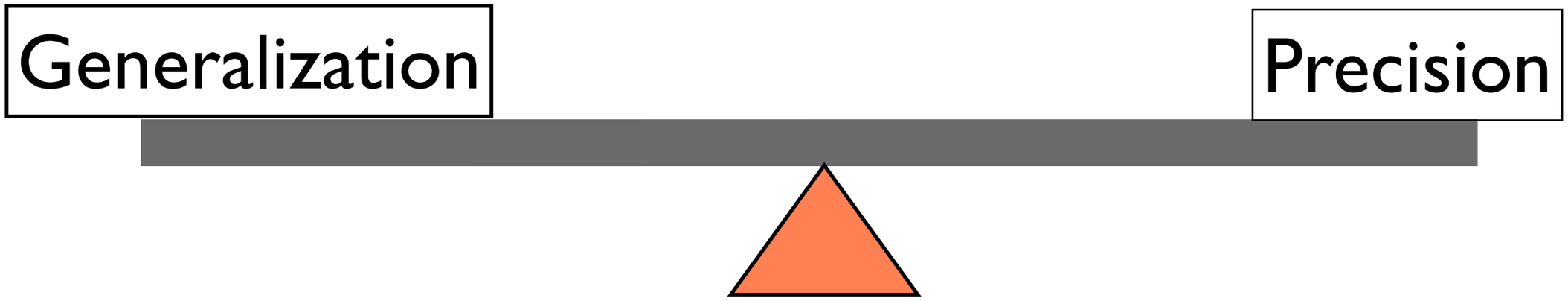
University of California, Santa Cruz

Atomicity

The effect of an atomic code block can be considered in isolation from the rest of a running program.

- enables sequential reasoning
- atomicity violations often represent synchronization errors
- most methods are atomic

Analyzing for Atomicity



- online/dynamic
- generalizes
- no false alarms

Thread 1

```
atomic{
  synchronized(n){
      tmp = bal;
  }
  synchronized(n){
      bal = tmp + 1;
  }
}
```

Thread 2

```
synchronized(m){
  newVar = 0;
}
```

Thread 1

```
begin
acquire(n)
t1 = bal
release(n)
acquire(n)
bal = t + 1
release(n)
end
```

Thread 2

```
acquire(m)
newVar = 0
release(m)
```

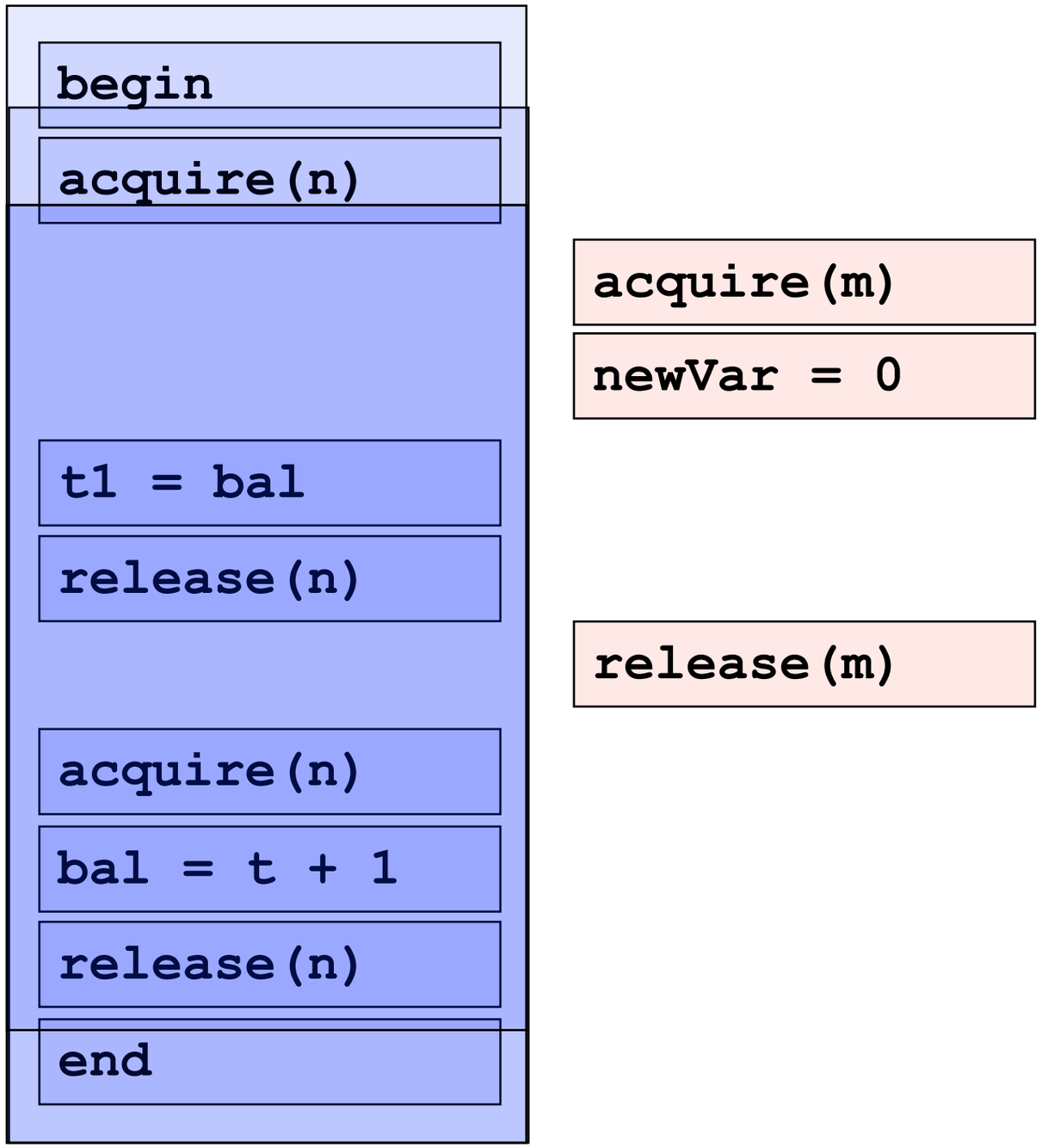
Serial Trace: Each atomic block executes contiguously

Thread 1

```
atomic{
   synchronized(n){
       tmp = bal;
   }
   synchronized(n){
       bal = tmp + 1;
   }
}
```

Thread 2

```
   synchronized(m){
      newVar = 0;
   }
```



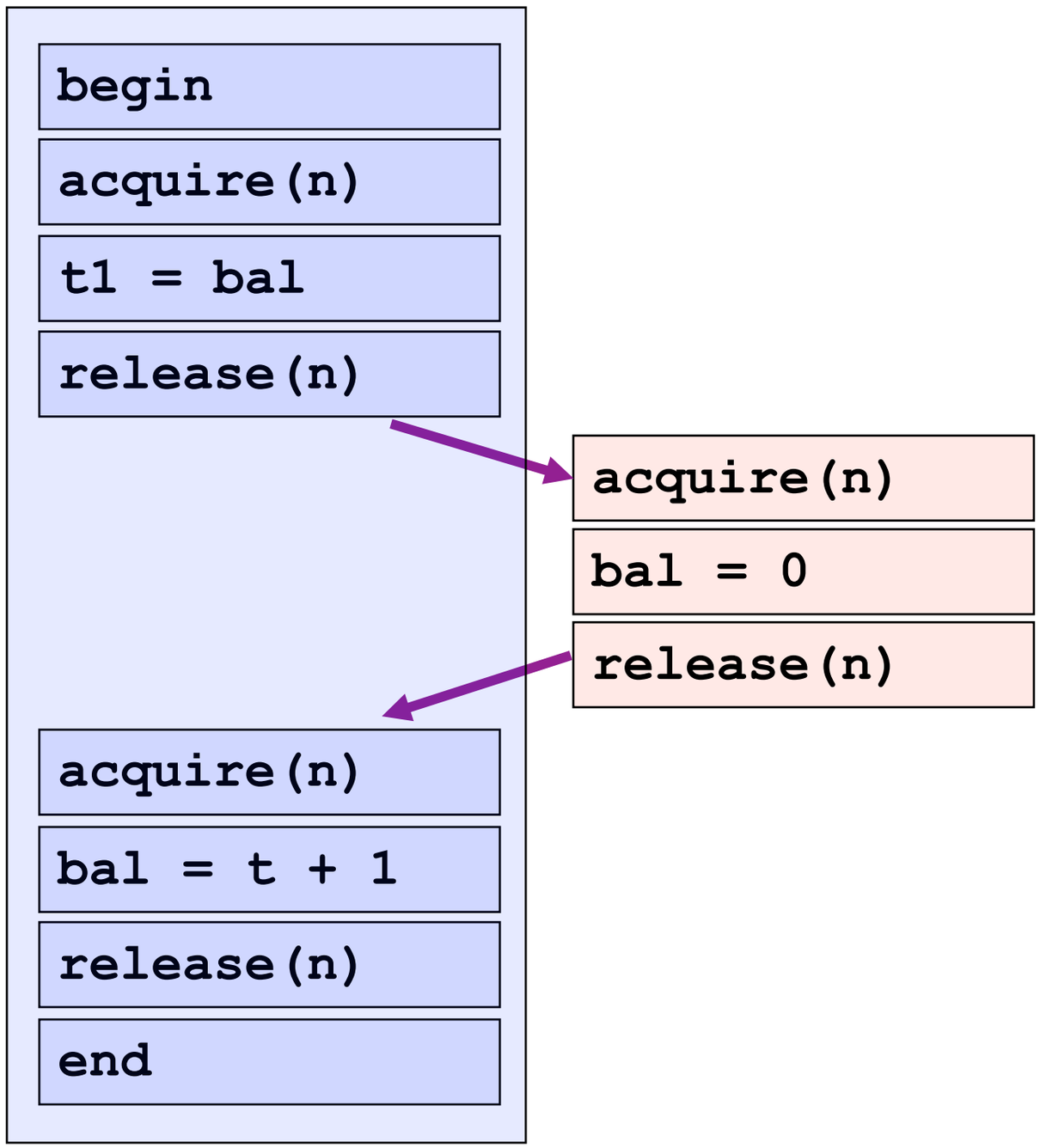
Atomicity = Serializability

Thread 1

```
atomic{
  synchronized(n){
      tmp = bal;
  }
  synchronized(n){
      bal = tmp + 1;
  }
}
```

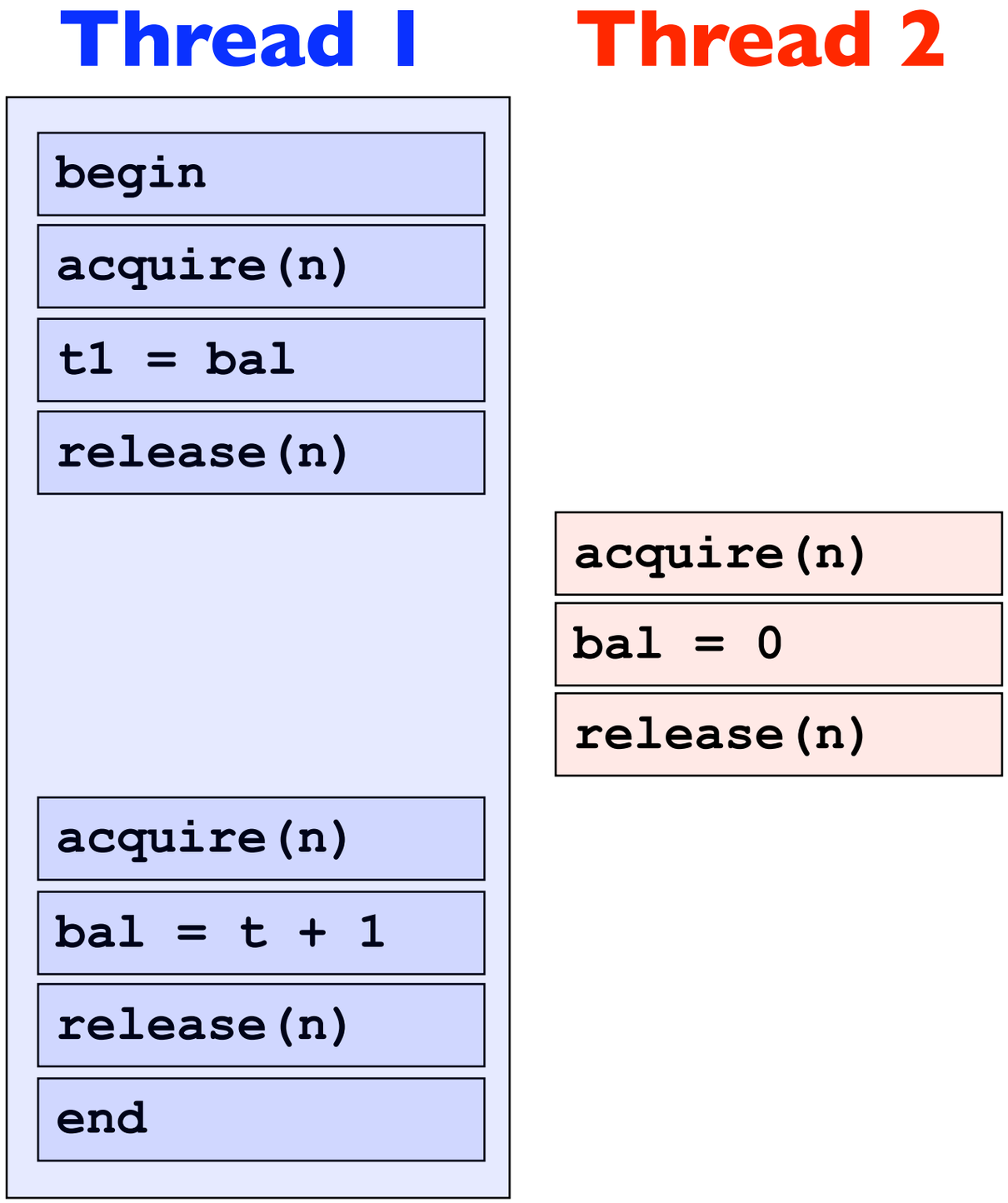
Thread 2

```
synchronized(n){
  bal = 0;
}
```



Happens-Before

Enables Relation

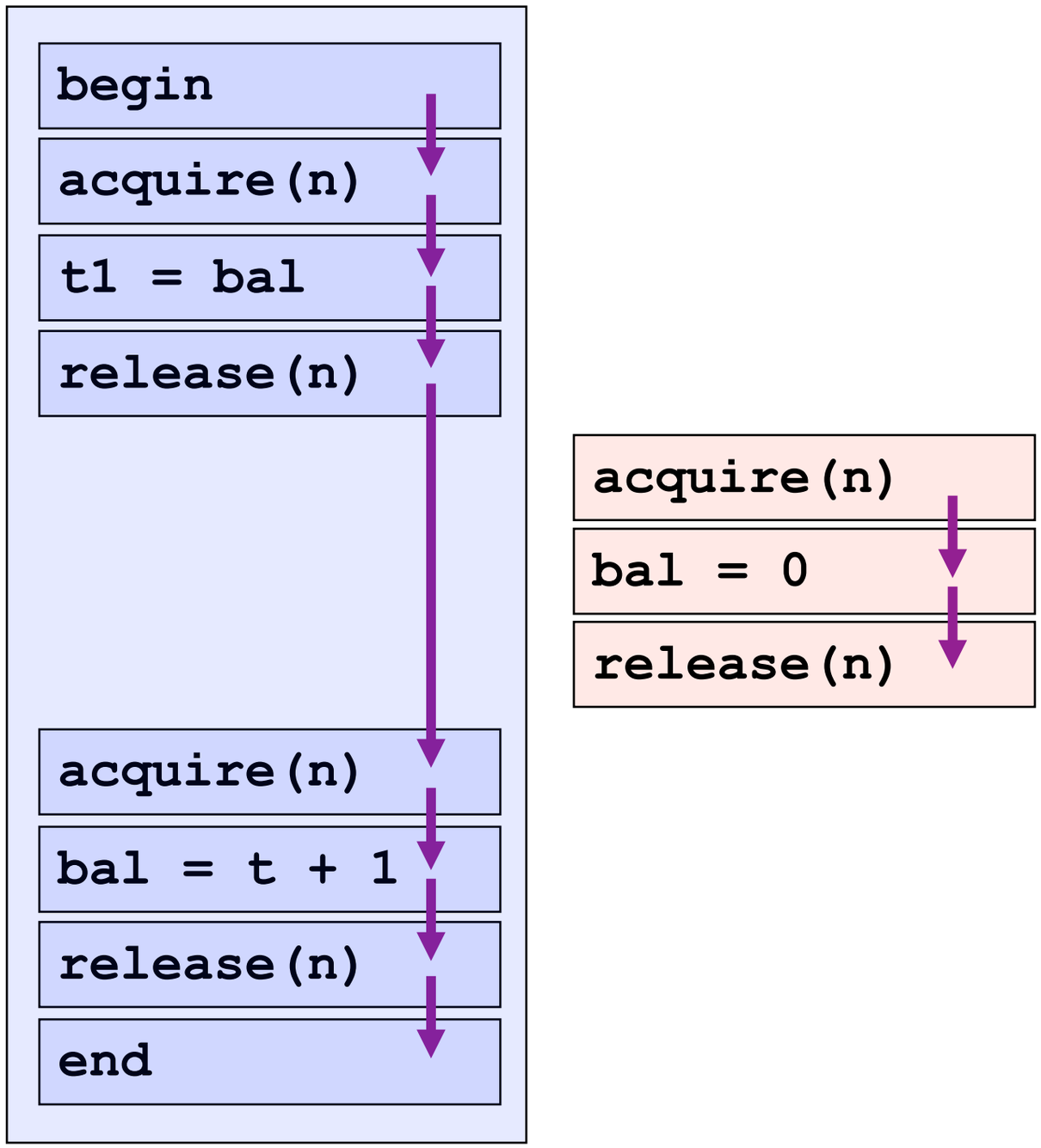


Happens-Before

- program order

Enables Relation

- program order

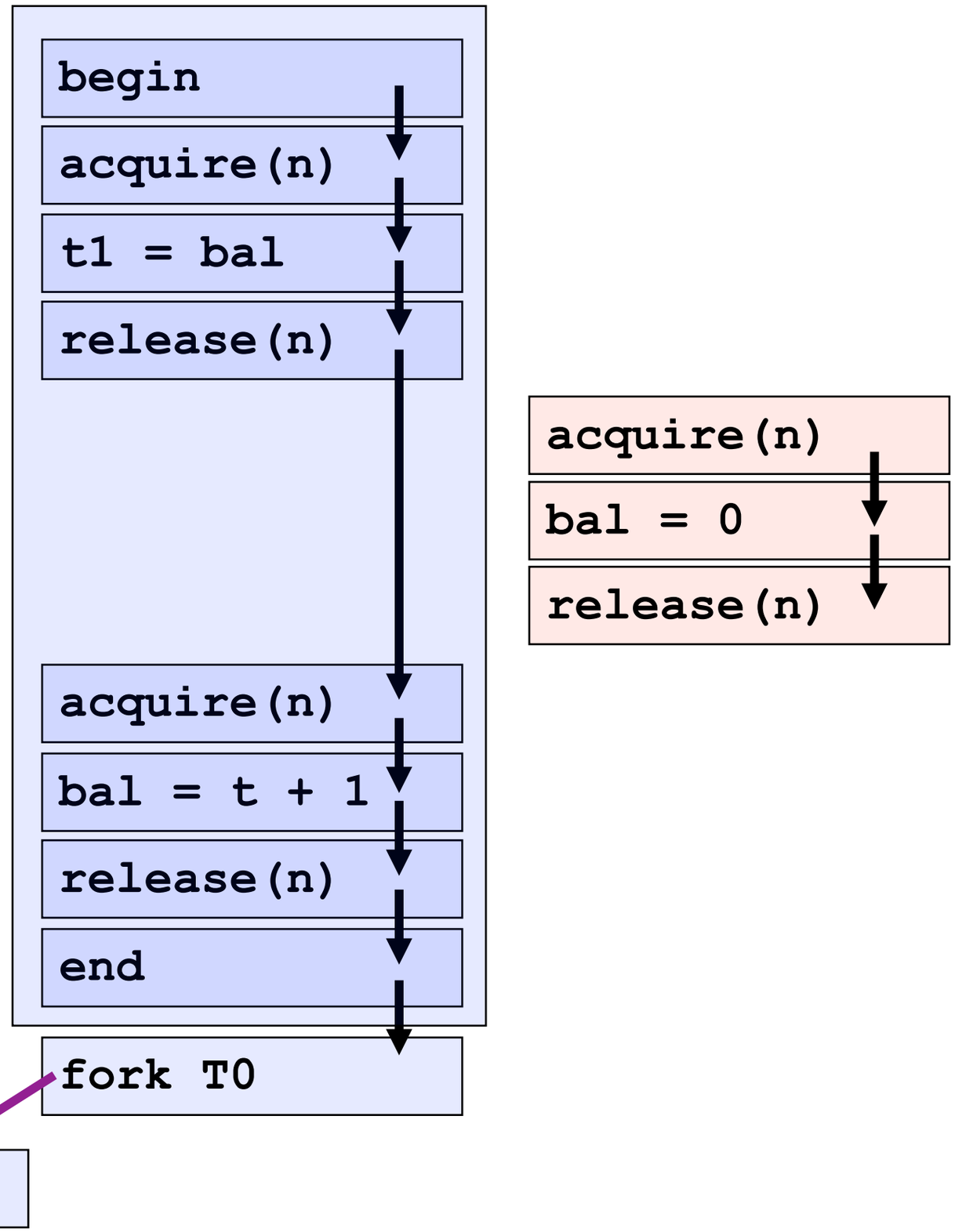


Happens-Before

- program order
- fork/join order

Enables Relation

- program order
- fork/join order

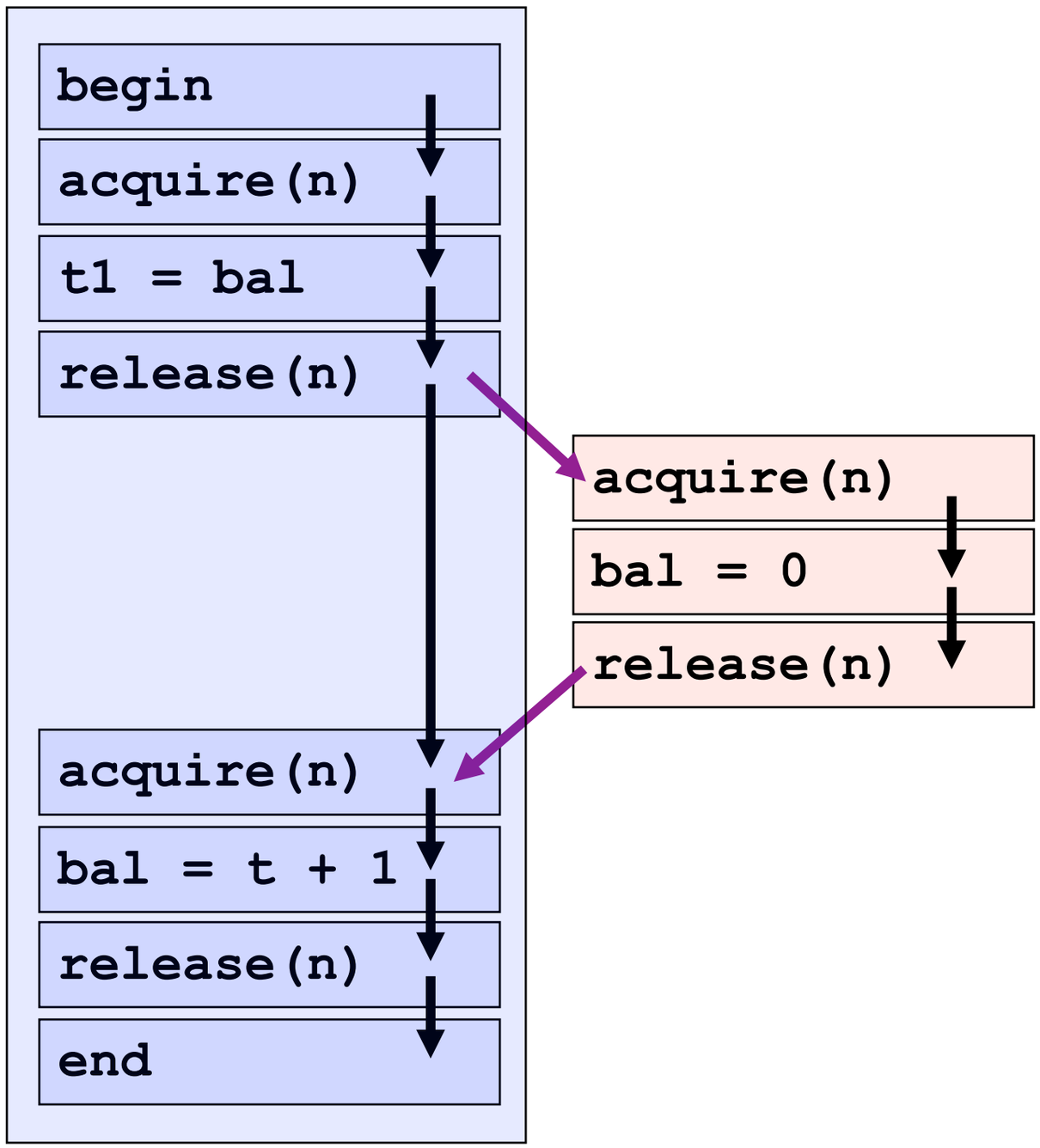


Happens-Before

- program order
- fork/join order
- synchronization order

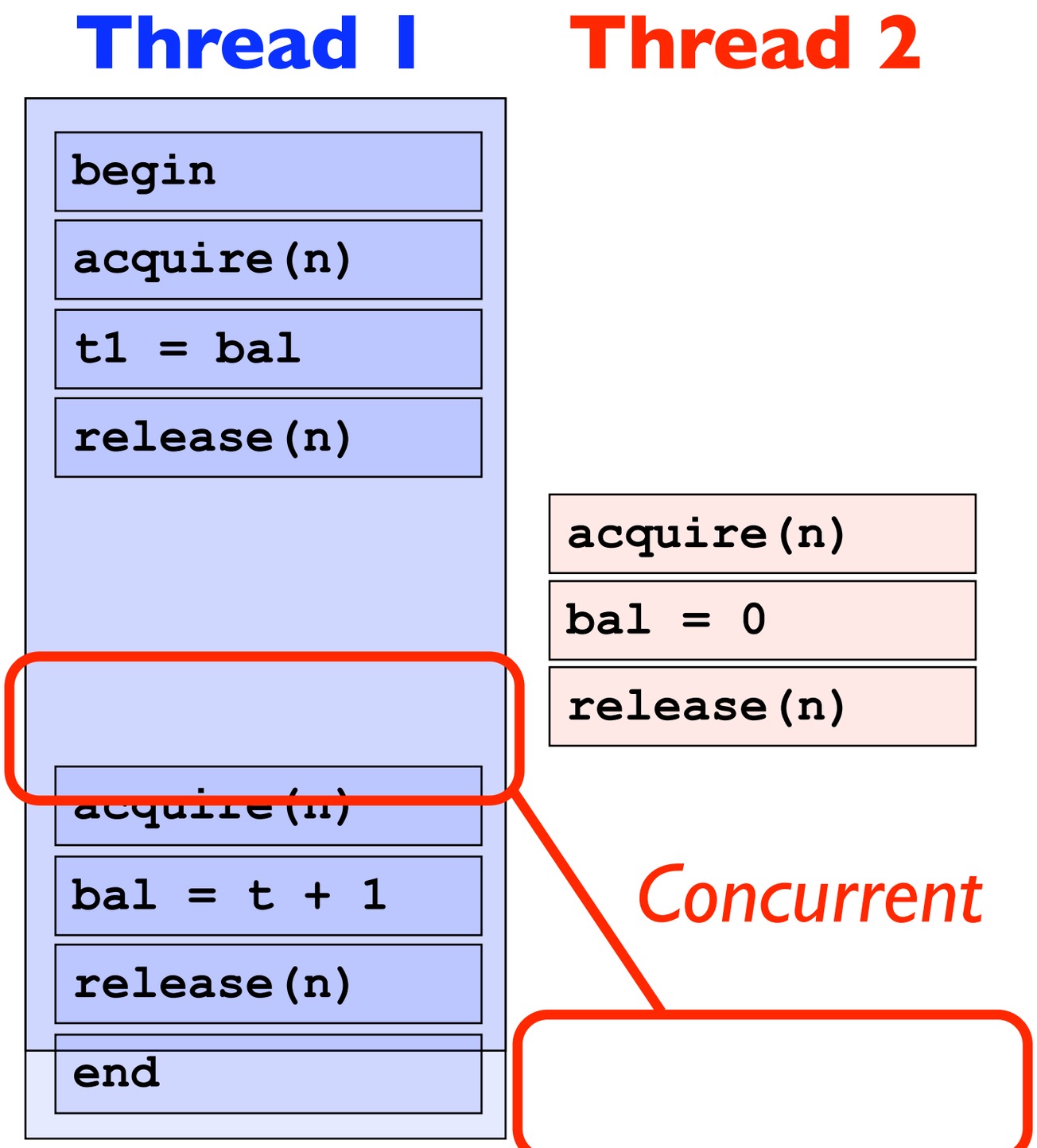
Enables Relation

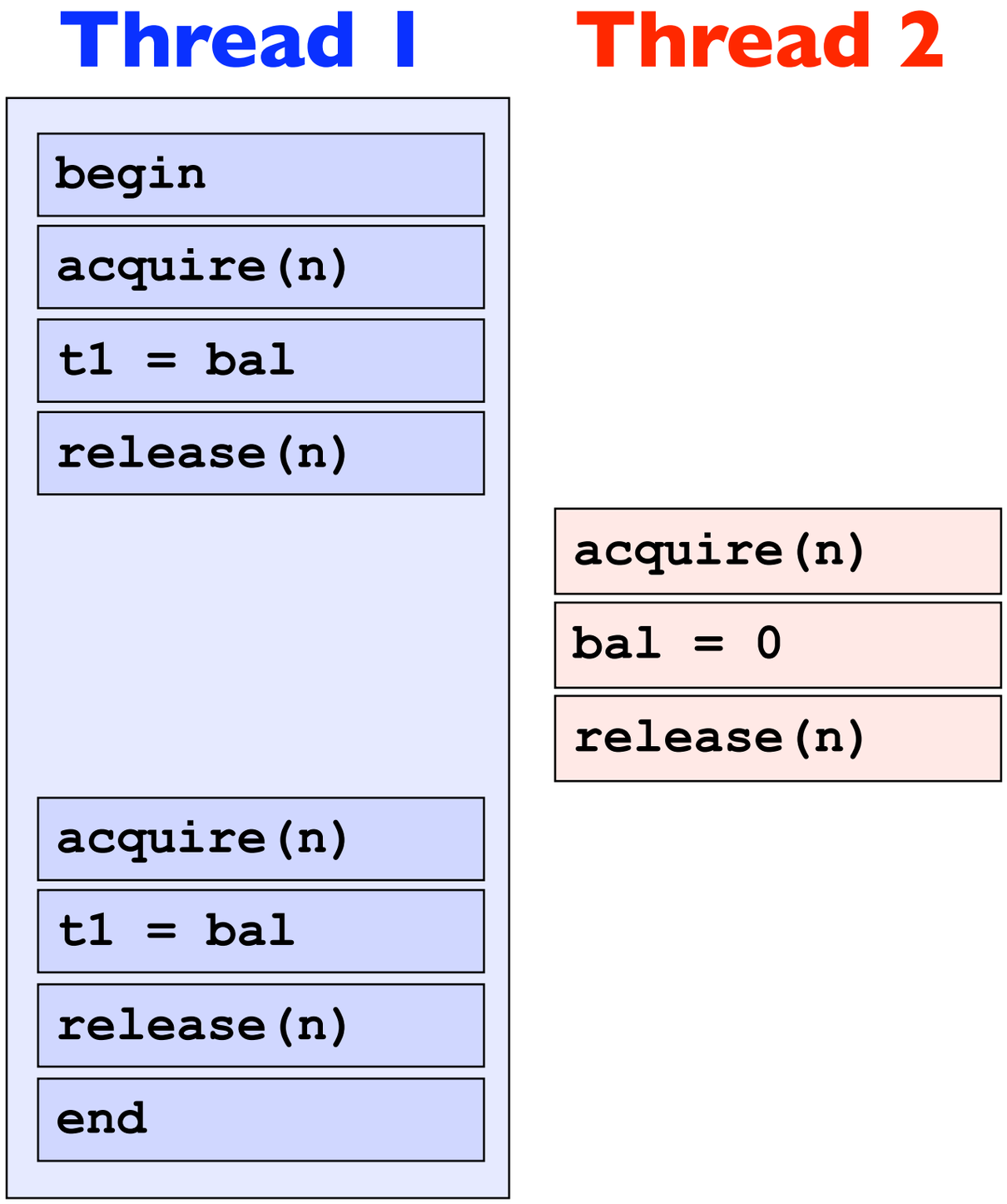
- program order
- fork/join order

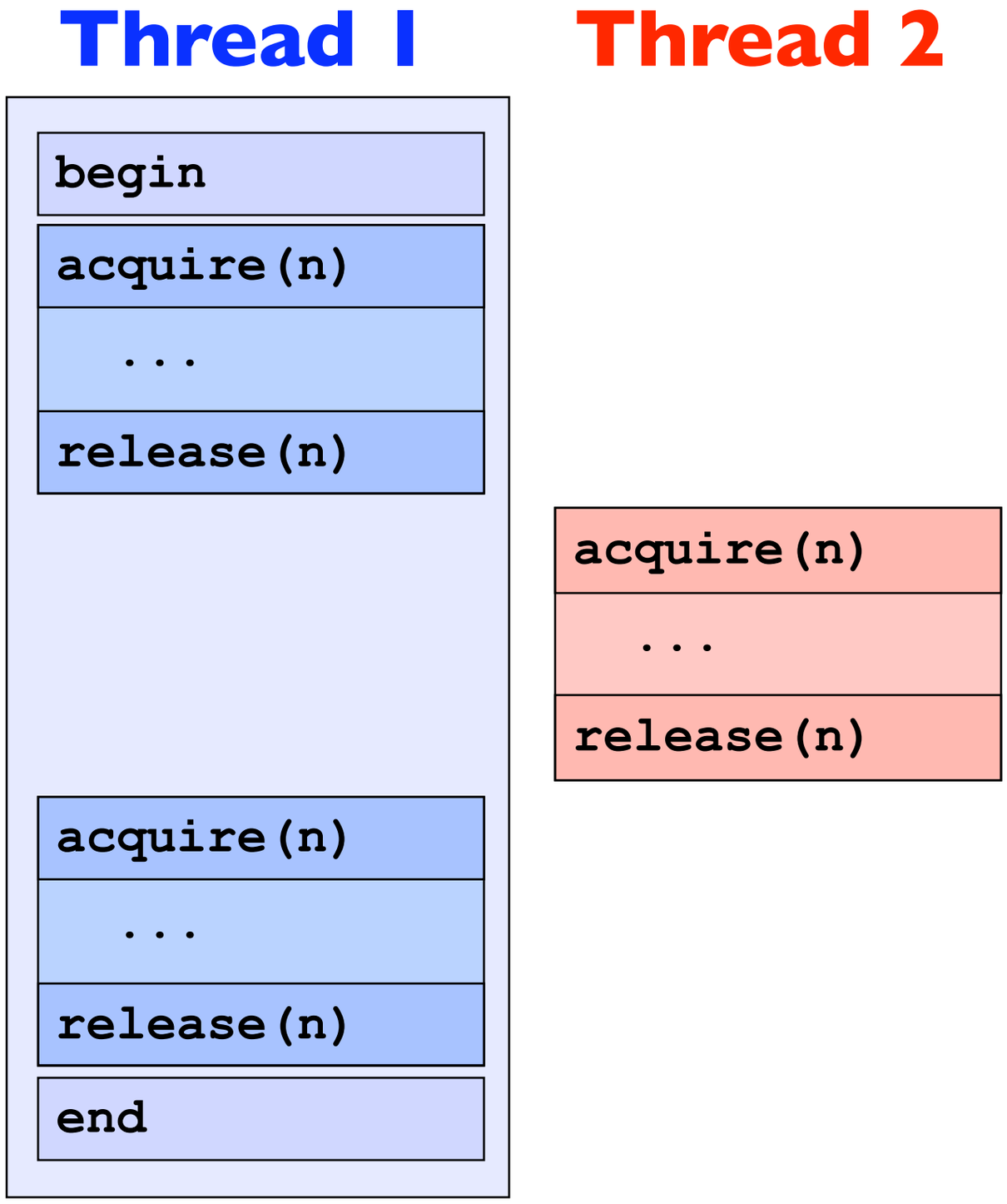


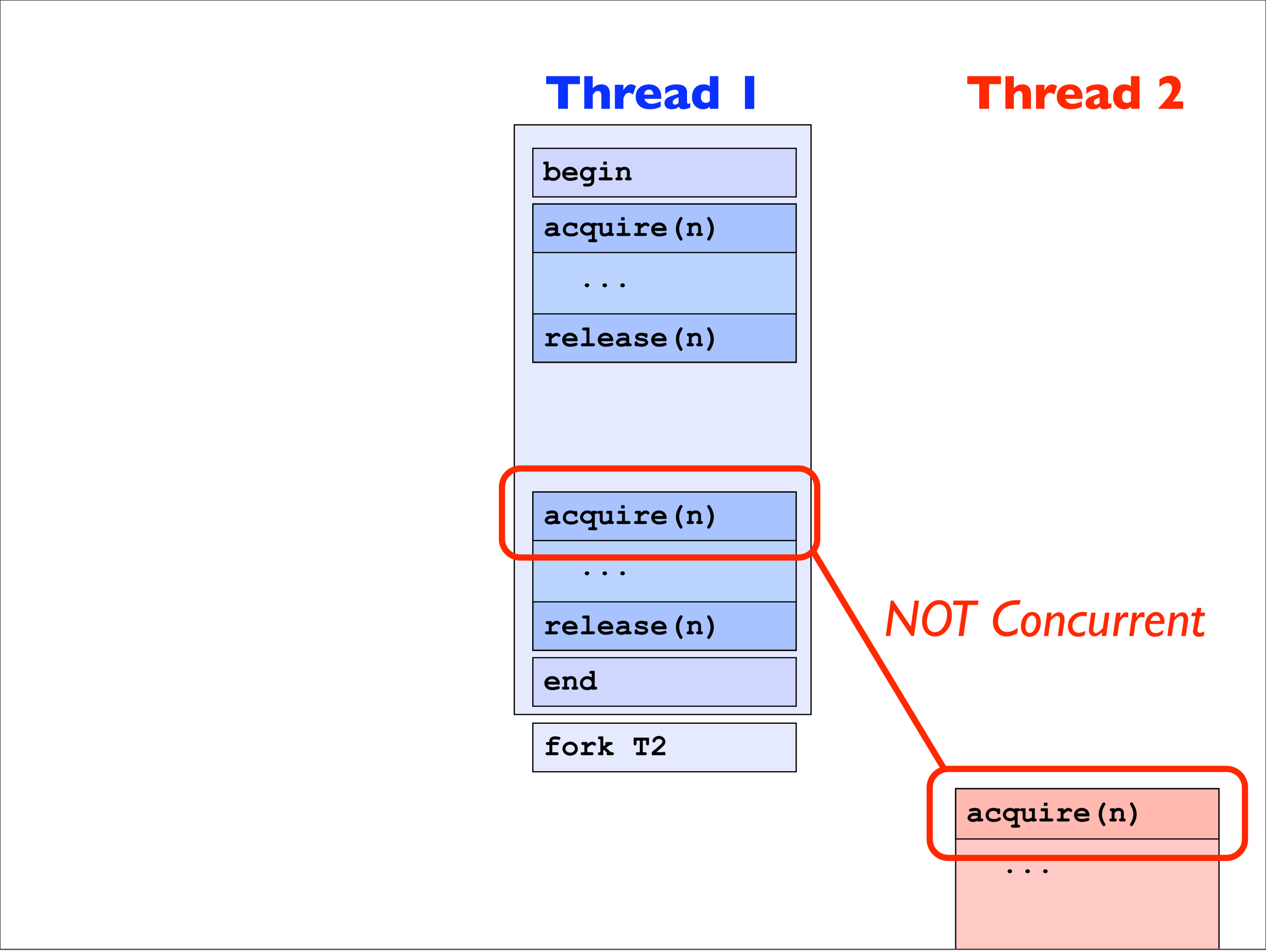
happens-before
atomicity violation

feasible
atomicity violation

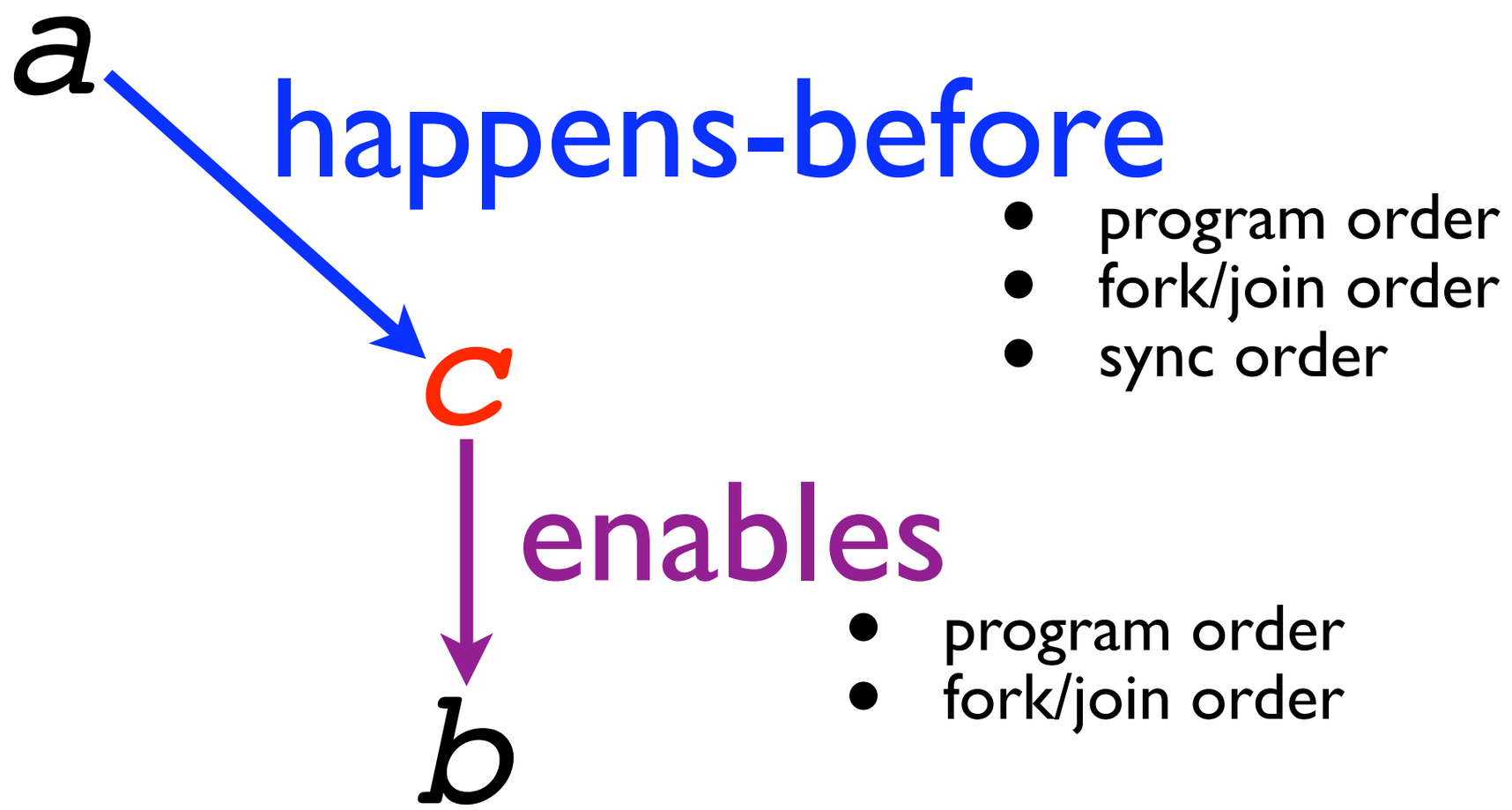



Thread 1
Thread 2
begin
acquire(n)
t1 = bal
release(n)
acquire(n)
bal = 0
release(n)
acquire(n)
t1 = bal
release(n)
end


Thread 1
Thread 2
begin
acquire(n)
...
release(n)
acquire(n)
...
release(n)
acquire(n)
...
release(n)
end

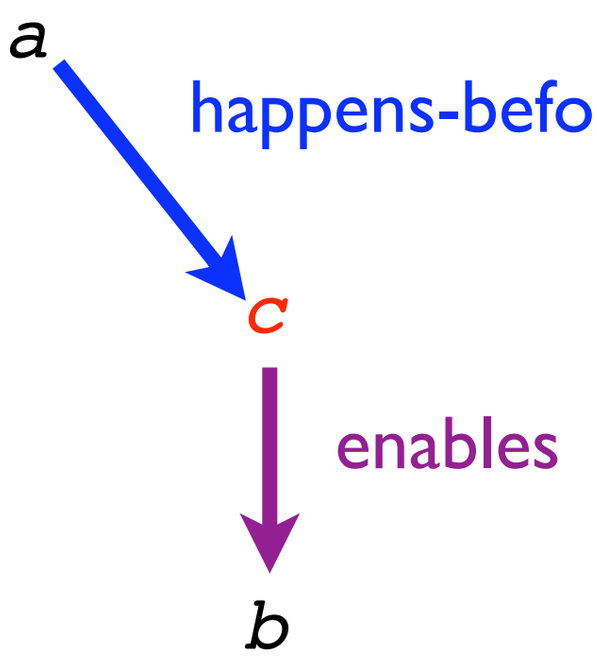

Thread 1
Thread 2
begin
acquire(n)
...
release(n)
acquire(n)
...
release(n)
end
fork T2
NOT Concurrent
acquire(n)
...

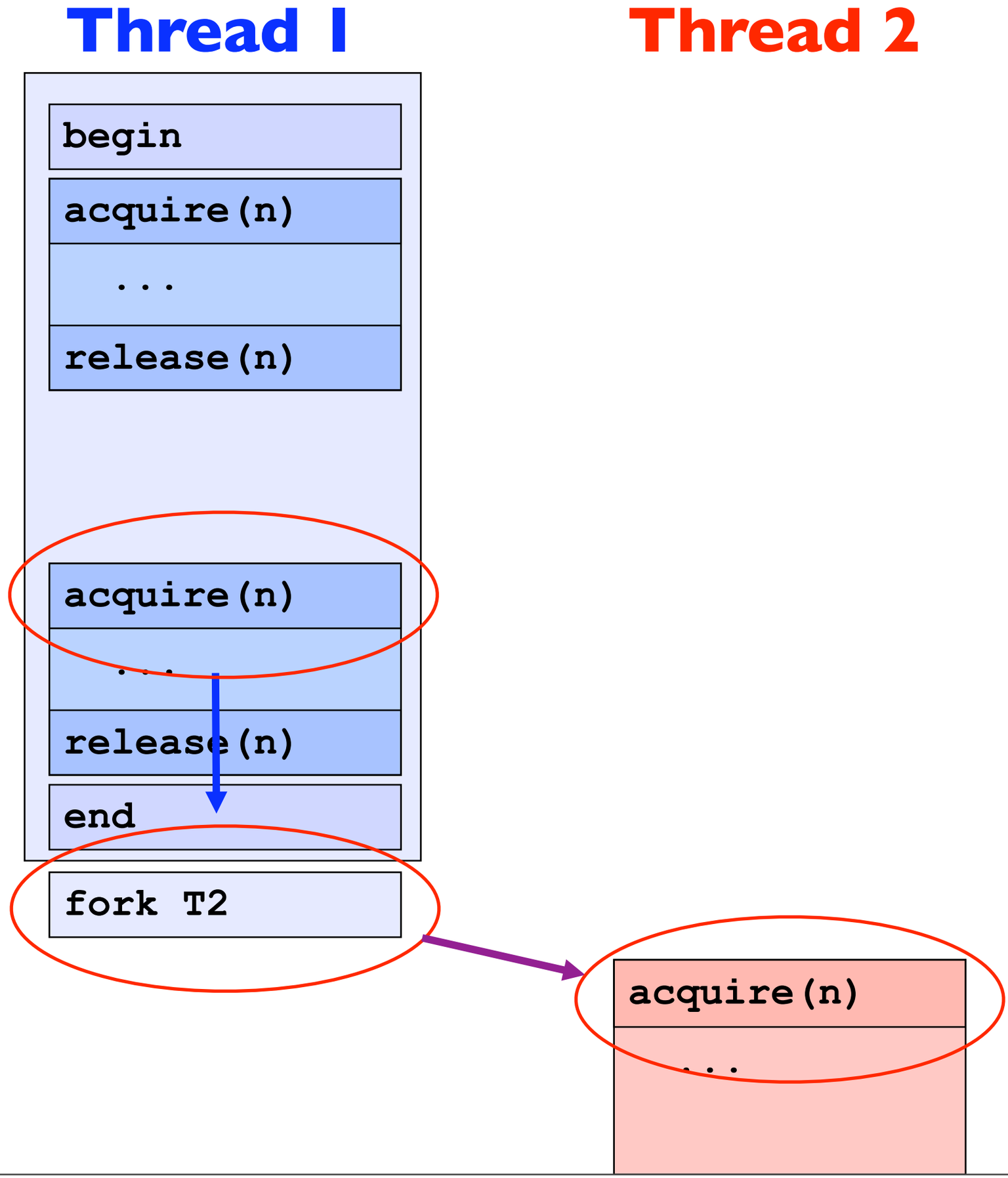
- In a trace, a lock operation *a* is *concurrent* with a later lock operation *b* if there are no intermediate operations which both enable *b* and happen-after *a*.



a and *b* not concurrent

NOT Concurrent:





no c

a

happens-before

c

enables

b

Thread 1

Thread 2

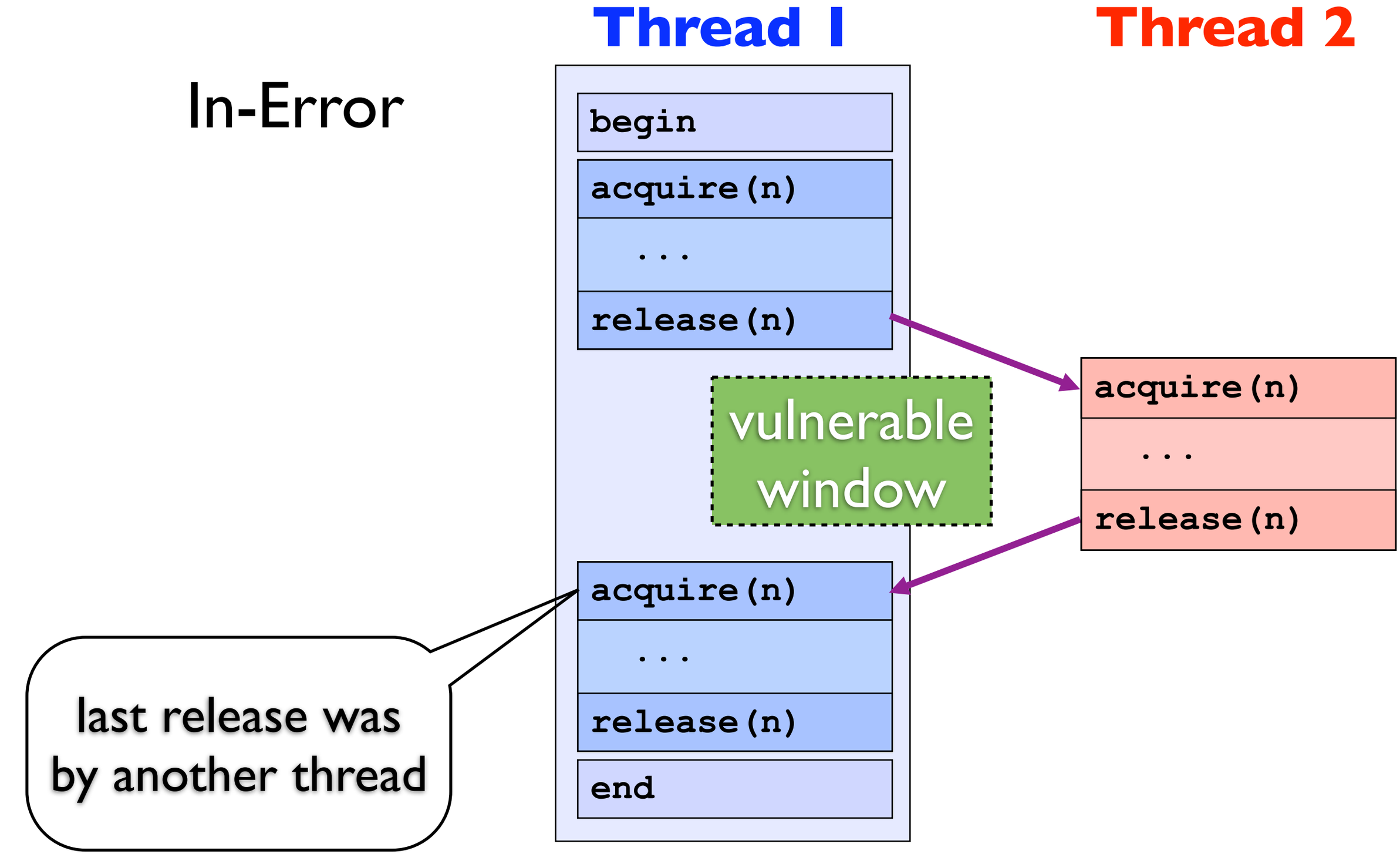
```
begin
acquire(n)
   ...
release(n)
```

```
acquire(n)
   ...
release(n)
end
```

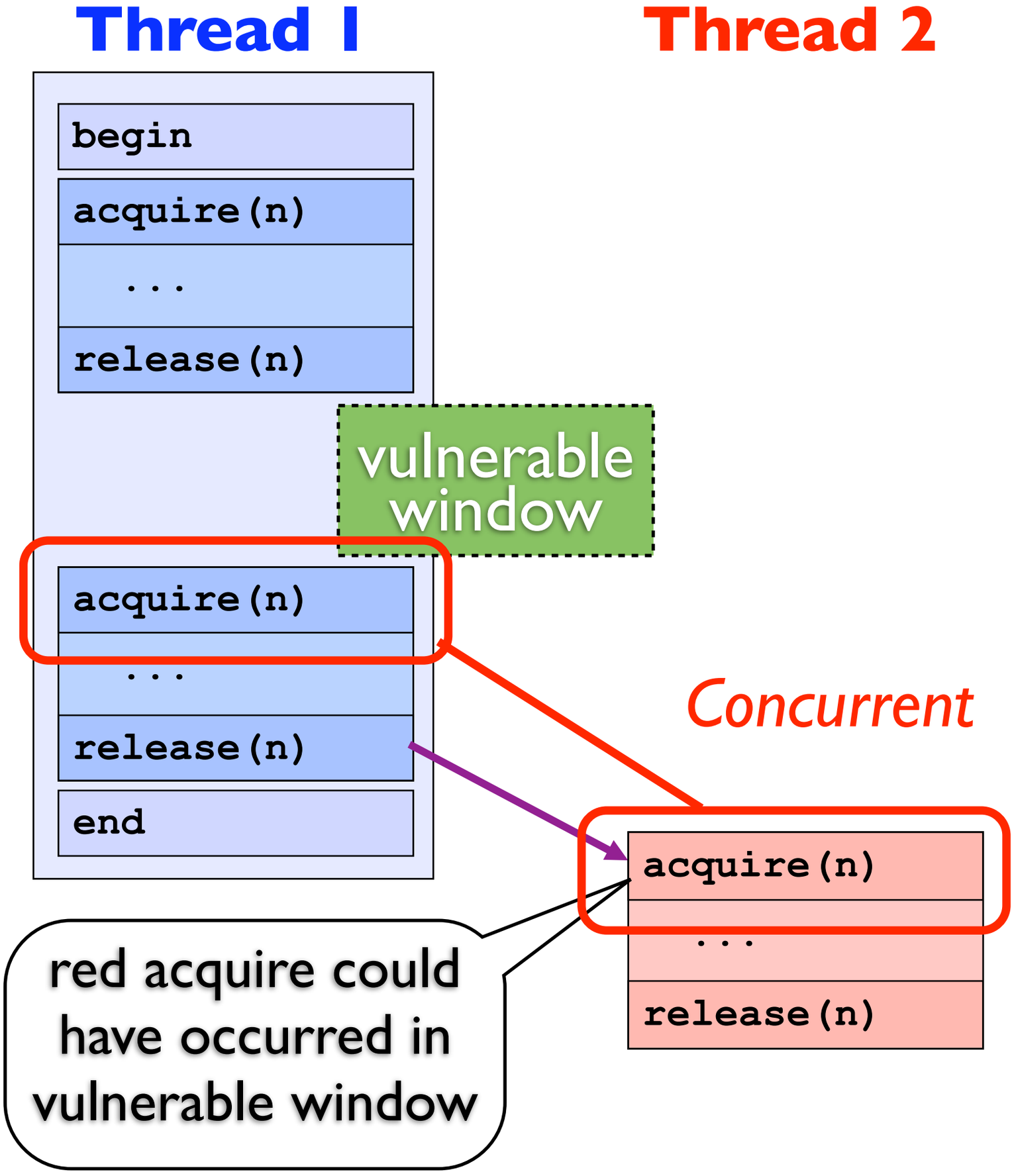
Concurrent

```
acquire(n)
   ...
release(n)
```

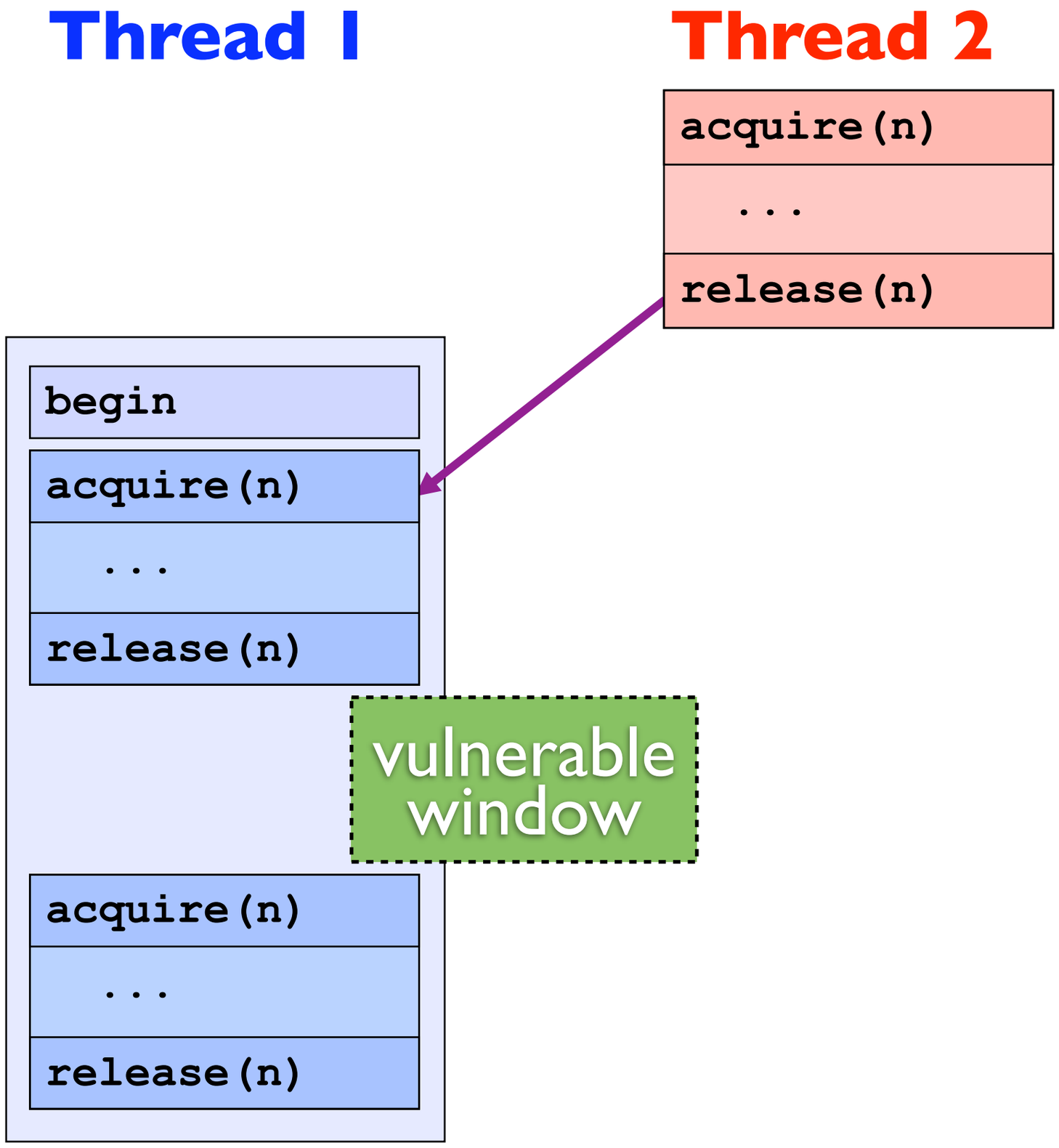
In-Error


Thread 1
Thread 2
begin
acquire(n)
...
release(n)
vulnerable window
acquire(n)
...
release(n)
acquire(n)
...
release(n)
end
last release was by another thread

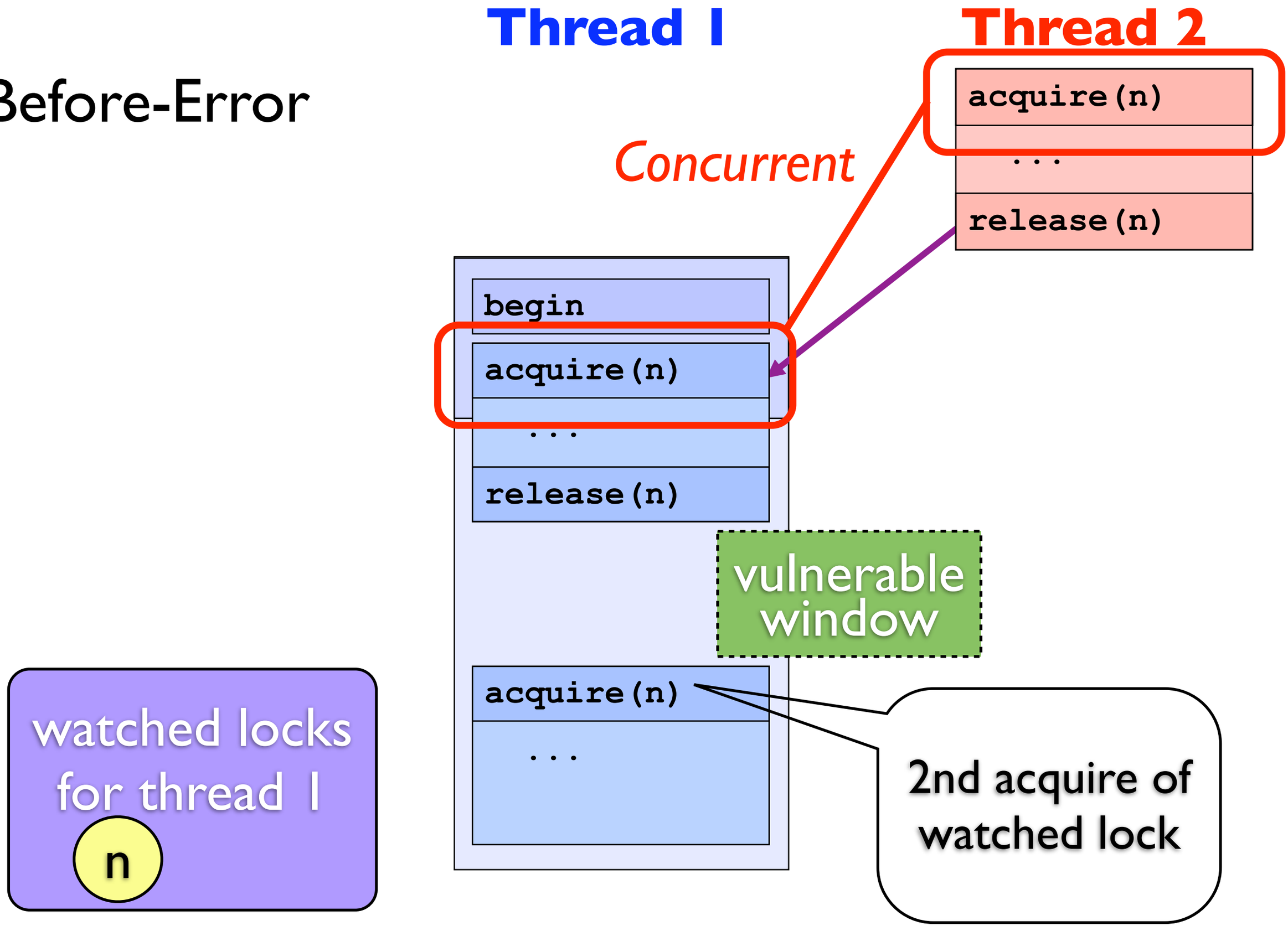
After-Error


Thread 1
Thread 2
begin
acquire(n)
...
release(n)
vulnerable window
acquire(n)
...
release(n)
end
Concurrent
acquire(n)
...
release(n)
red acquire could have occurred in vulnerable window

Before-Error



Before-Error



Blame Assignment

```
atomic b(){
  d();
  c();
}
```

Not Atomic!

```
atomic c(){
  d();
}
```

Atomic!

```
atomic d(){
  sync(m) {...}
}
```

Atomic!

Call Stack

b

b:d

b

b:c

b:c:d

Thread 1

Thread 2

Concurrent

```
enter d
acq/rel(m)
exit d
```

```
begin
enter b
enter d
acq/rel(m)
exit d
enter c
enter d
acquire(m)
```

vulnerable window

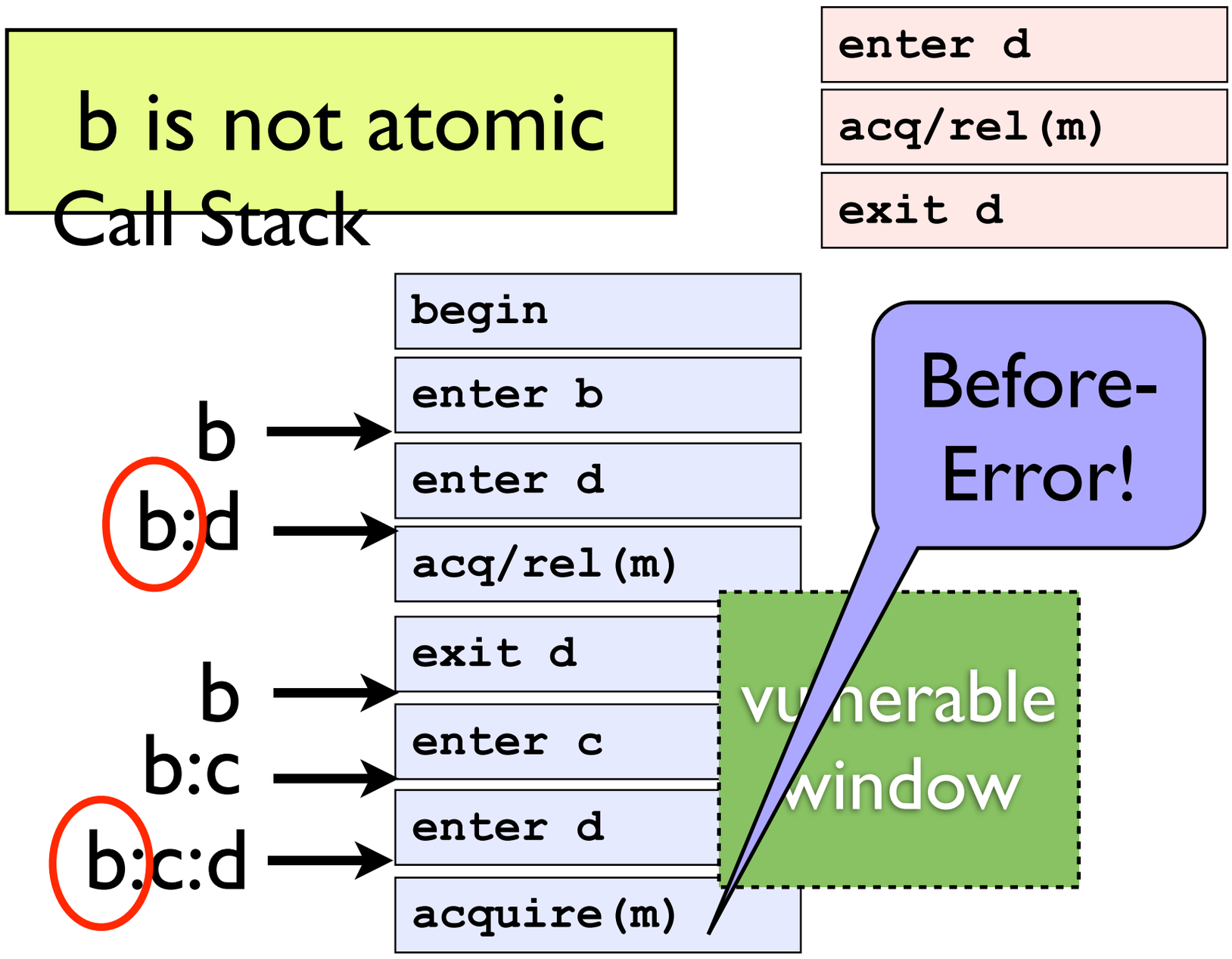
Before-Error!

Blame Assignment

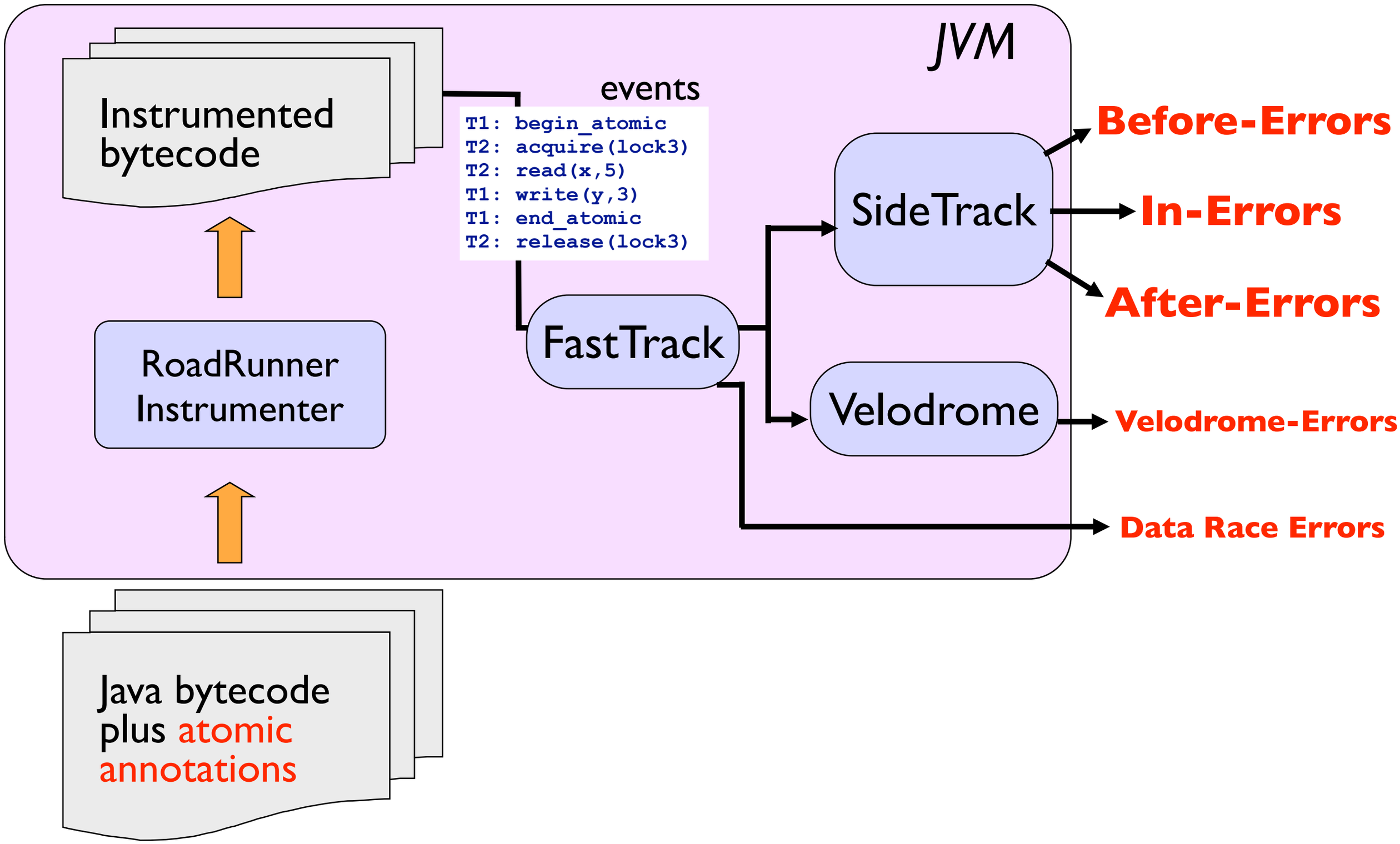
```
atomic b(){
  d();
  c();
}

atomic c(){
  d();
}

atomic d(){
  sync(m) {...}
}
```



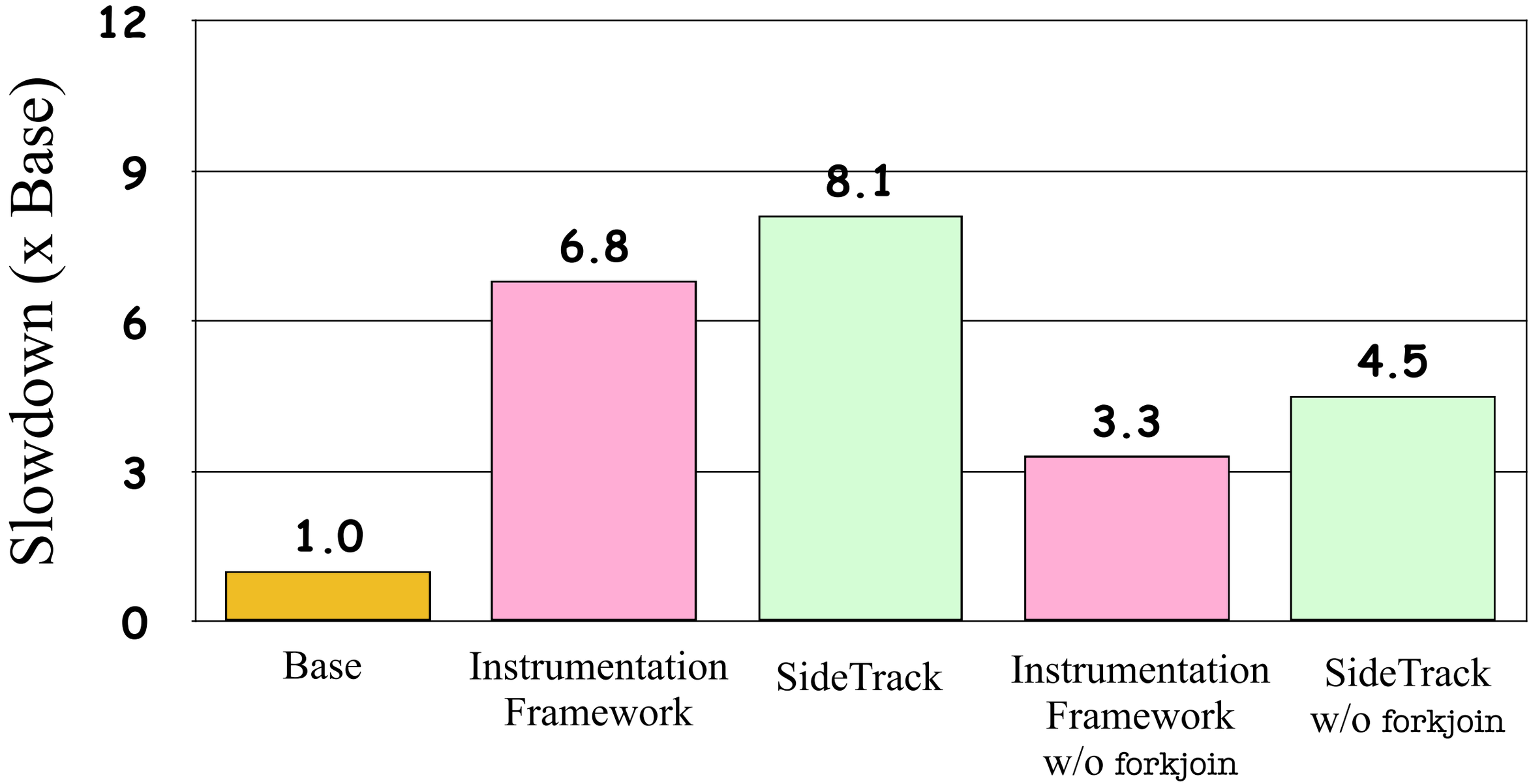
SideTrack Implementation



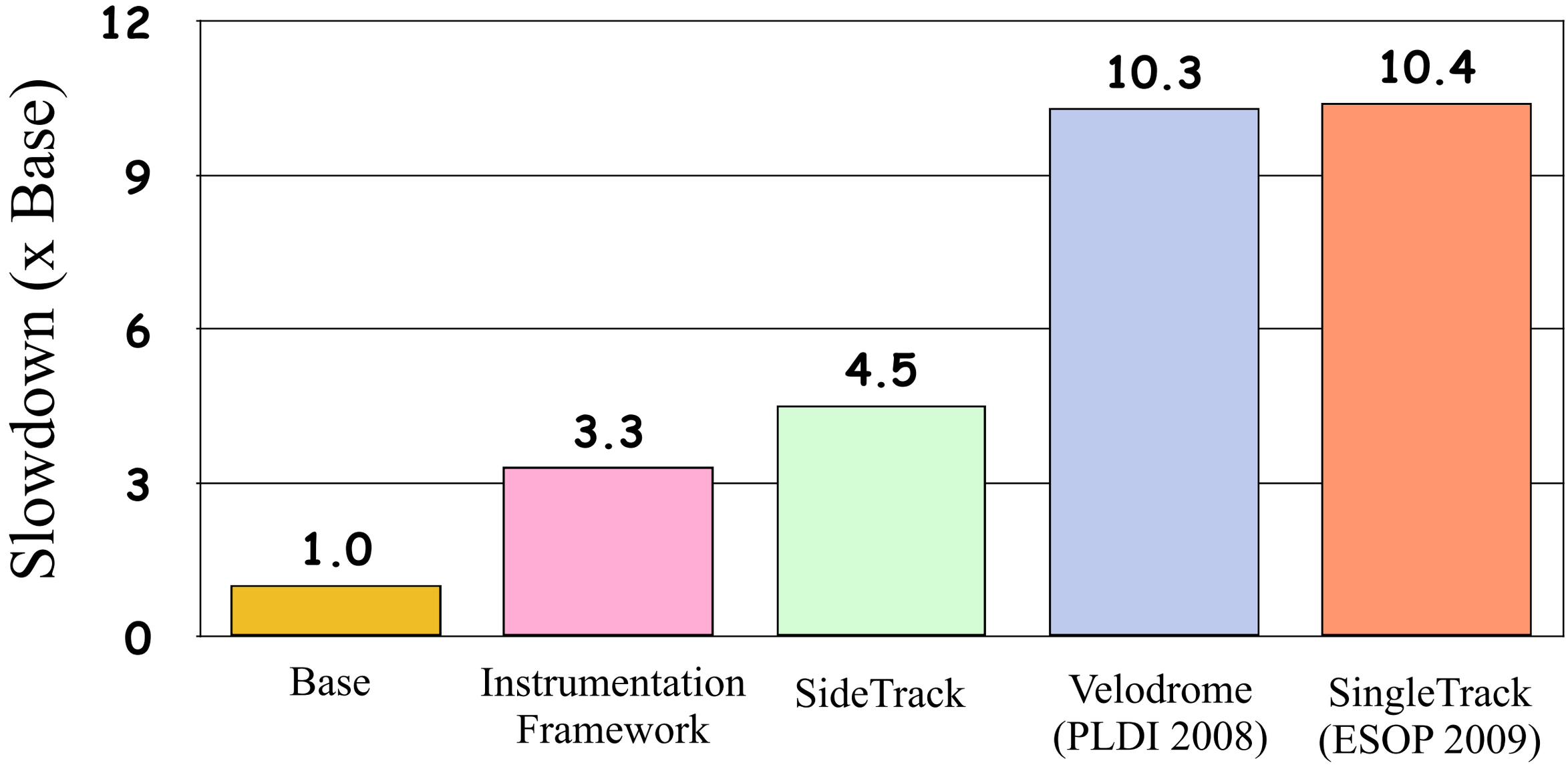
Errors Found:
40% improvement with prediction

	In-Errors	Before-Errors	After-Errors	Predicted Errors (Before ∪ After)/In
elevator	1	3	5	4
colt	7	4	9	2
jbb	5	7	10	5
hedc	4	1	4	0
barrier	1	1	1	0
philo	1	1	1	0
tsp	4	4	4	0
sync	4	4	4	0
	27	25	38	11

Experimental Results: Performance



Experimental Results: Performance



Related Work:

Predictive Approaches

- Wang & Stoller et al. (2006, 2009)
  - analyze traces offline; add static info (HAVE)
- JPredictor (Chen et al. 2008)
  - offline causality slicing, violation patterns
- Farzan & Madhusudan (2009)
  - time bounds & algorithms, no implementation
- AtomFuzzer (Sen & Park 2008)
  - drive scheduler to produce violation, probabilistic

Conclusion: SideTrack

- **no false alarms**
- predicts **feasible** atomicity violations
- **40% increase** in atomicity violations detected
- competitive performance
- chain with other tools (Velodrome, FastTrack)

SideTrack:

- no false alarms
- predicts feasible atomicity violations
- 40% increase in atomicity violations detected
- competitive performance
- chain with other tools

Future Work

- volatiles, wait/notify, barriers, etc.
- direct comparison with other tools
- more benchmarks
- formal proofs