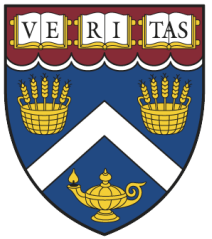
Threading and Concurrency II



Concurrent Memory

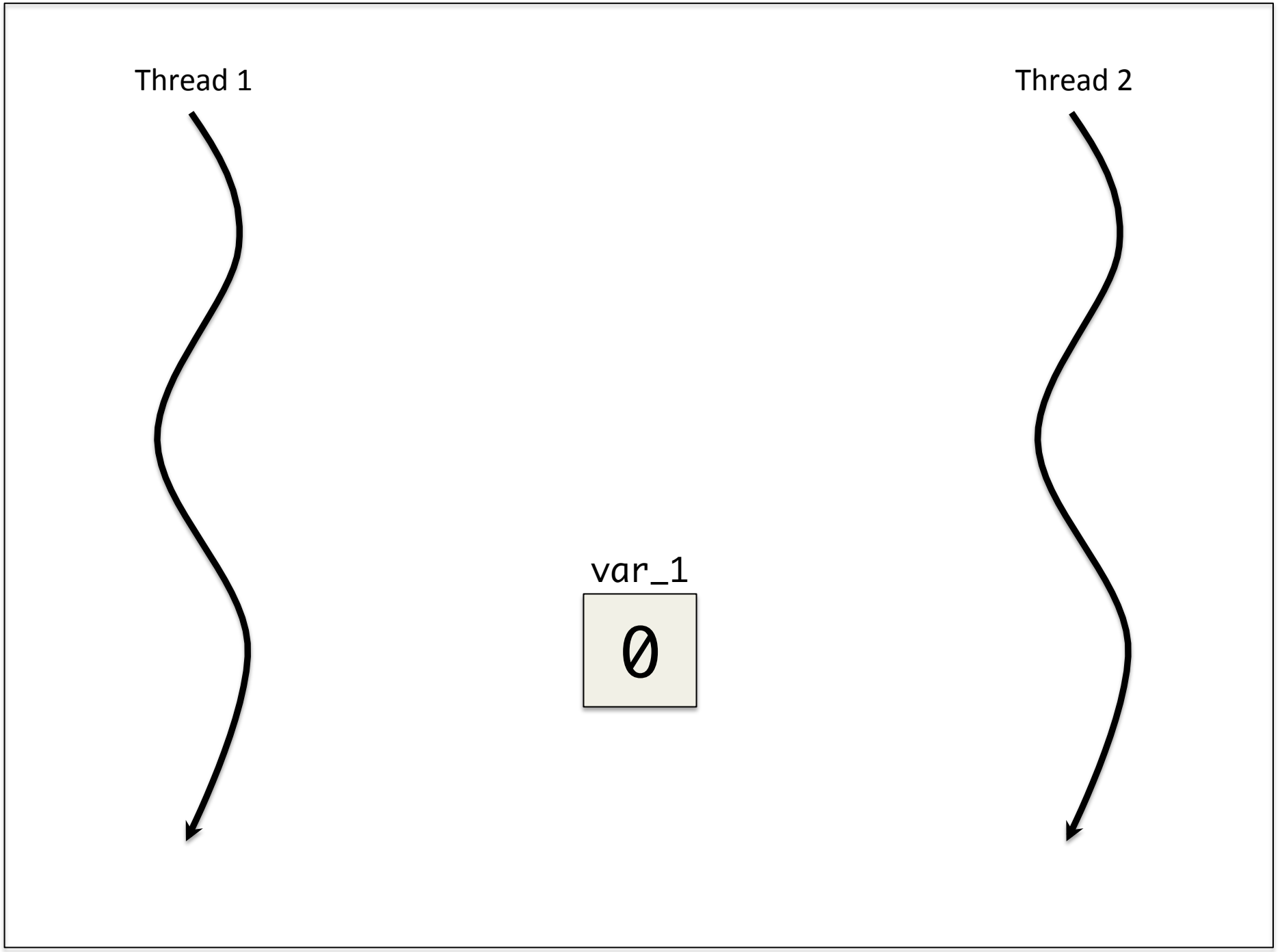
- Threading is simple if there are no side effects
  - Threads don't communicate with one another
  - Can run completely independently
- Much harder if threads influence one another
  - Frequently happens through the memory system
  - Could be any shared resource

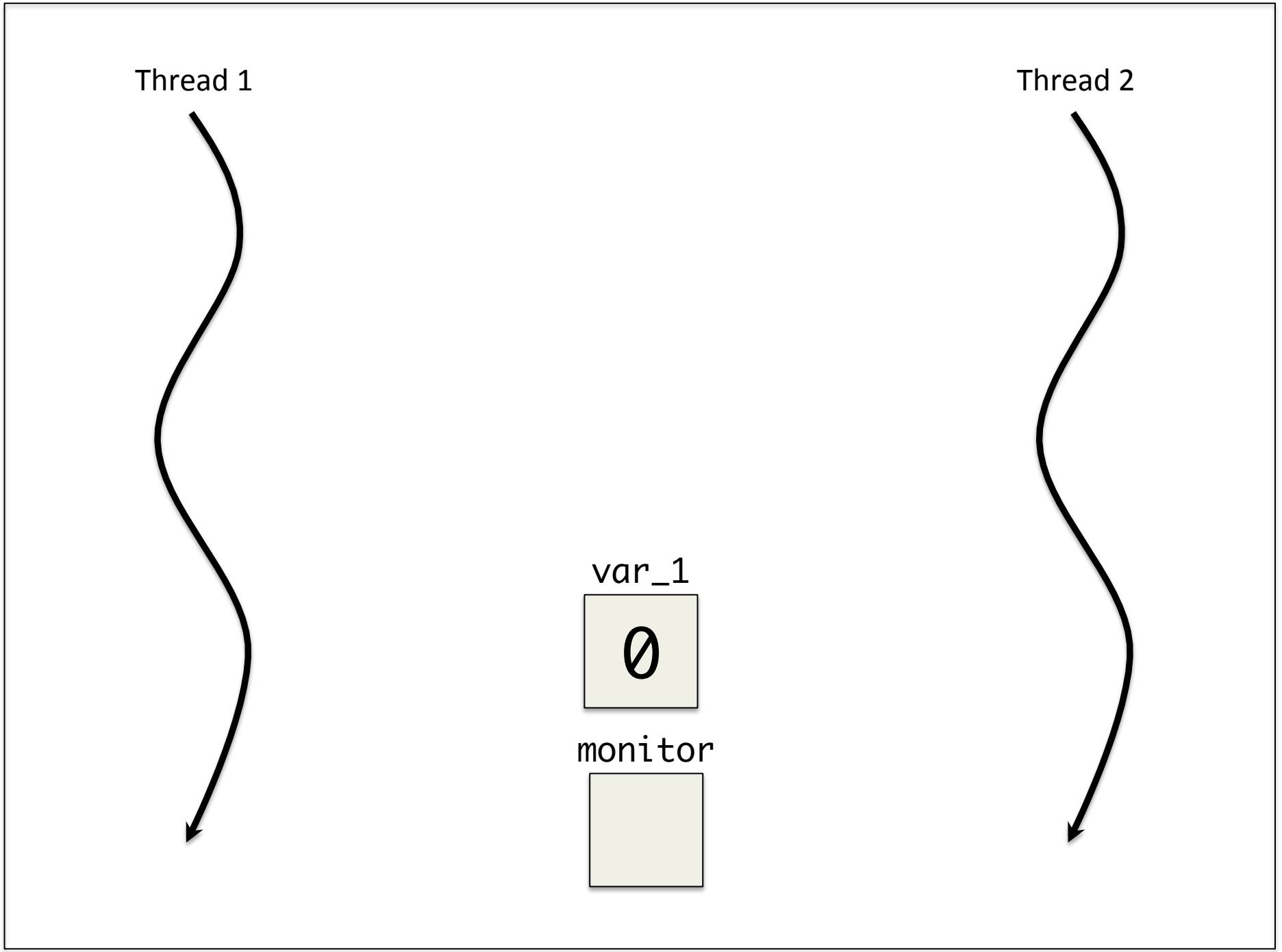
Synchronization

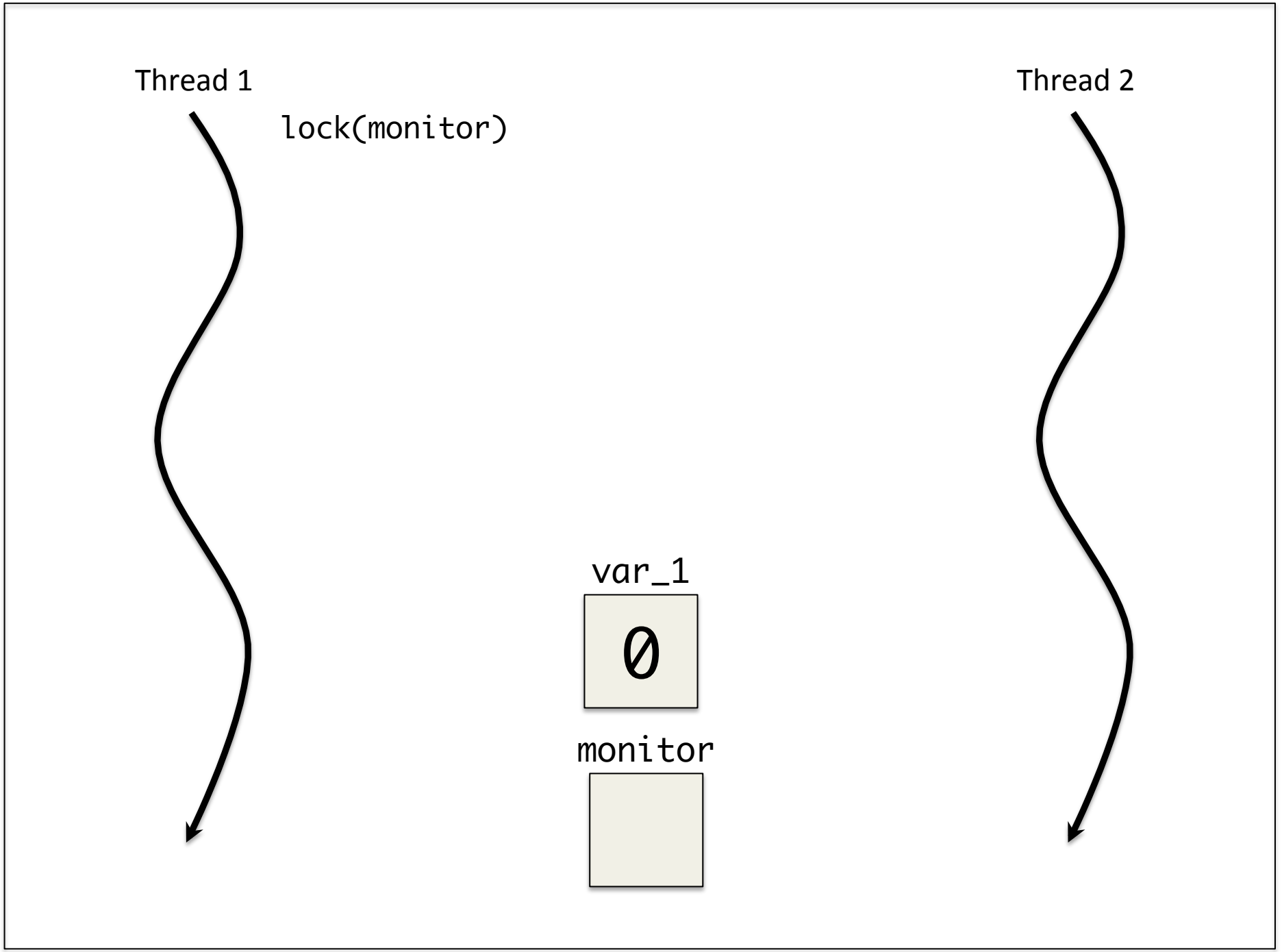
- Introduce communication between threads
- Two forms of inter-thread communication
  - Mutual exclusion
  - Cooperation
- Java handles both through monitors
  - Any object can be a monitor
  - Most objects are not monitors

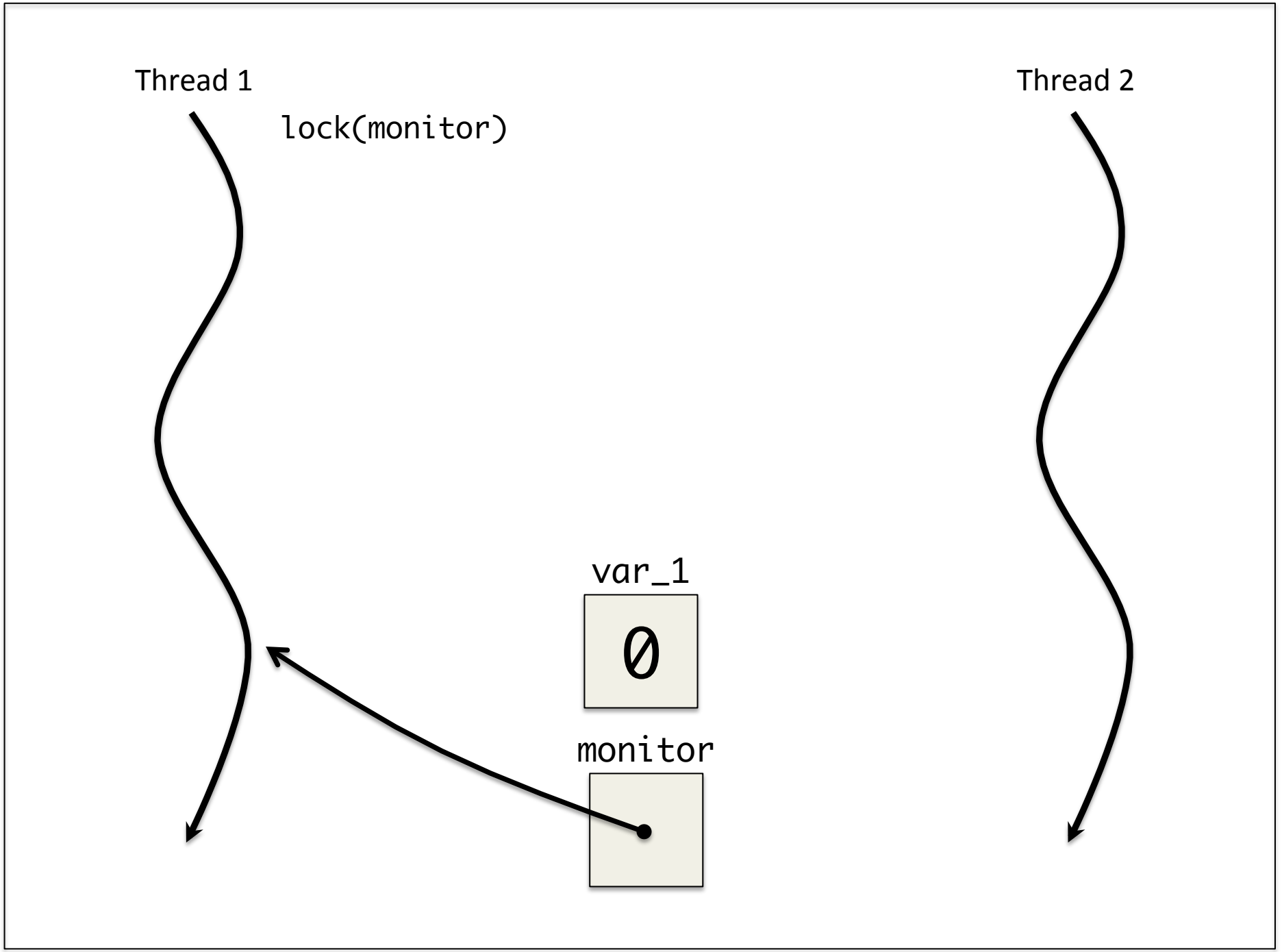
Mutual Exclusion

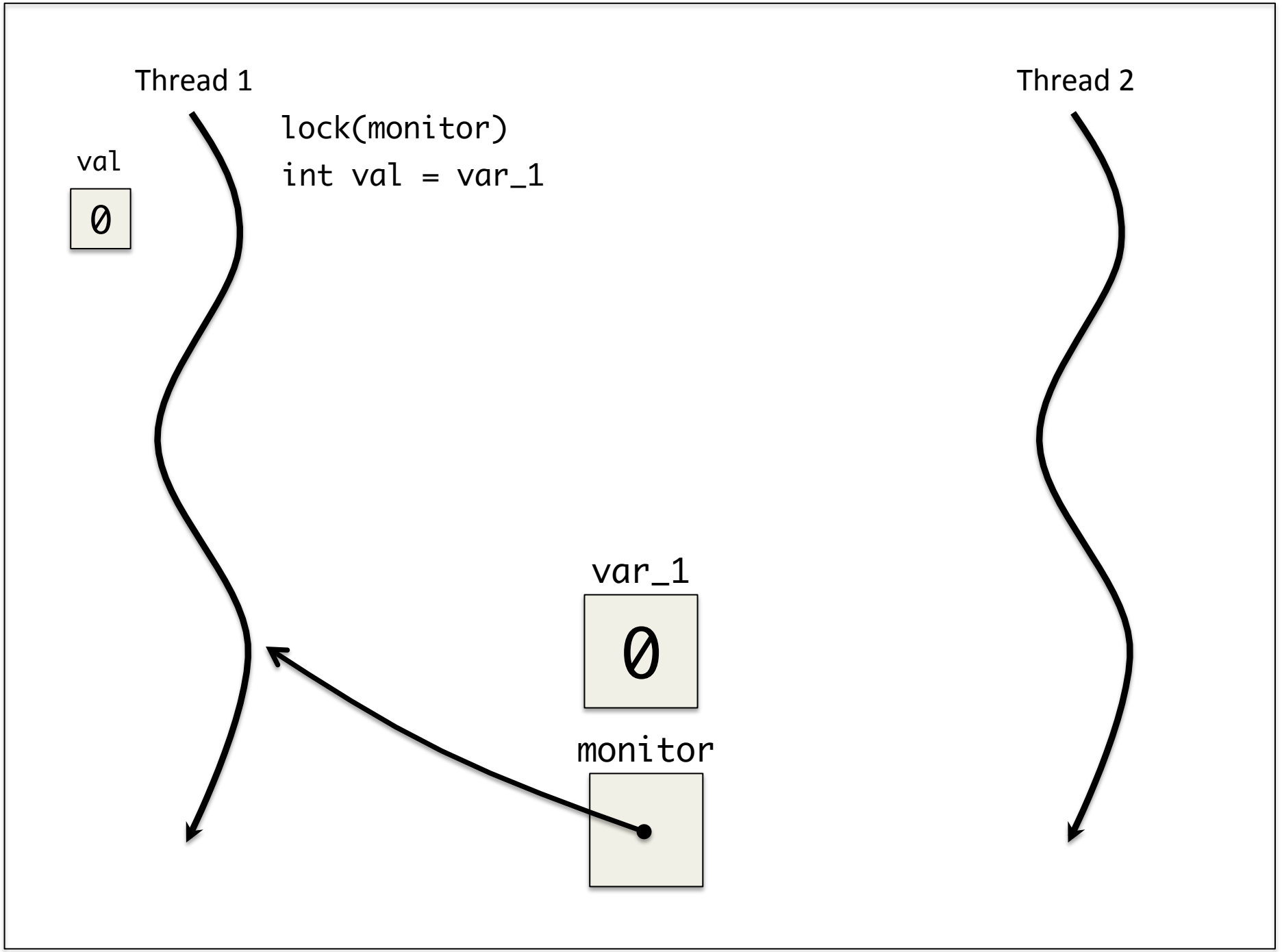
- Mark a section of code as guarded
  - Only one thread can be in it at a time
  - Java has two forms of synchronized code
- `synchronized` block
  - First thread to enter claims the lock
  - All other threads wait until it exits the block
- Method-level `synchronized` keyword
  - Claims monitor on `this` or class object

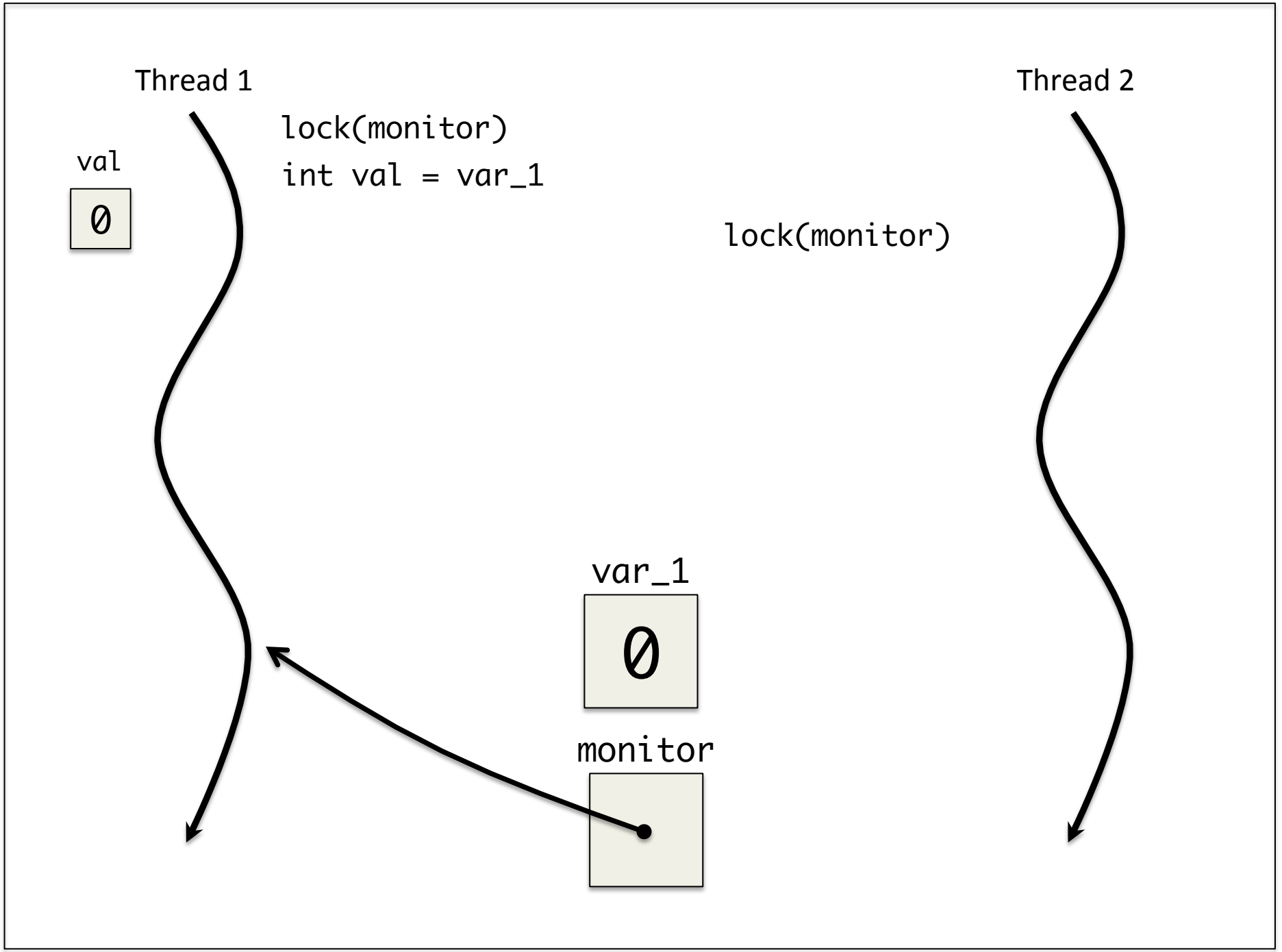

Thread 1
Thread 2
var_1
0

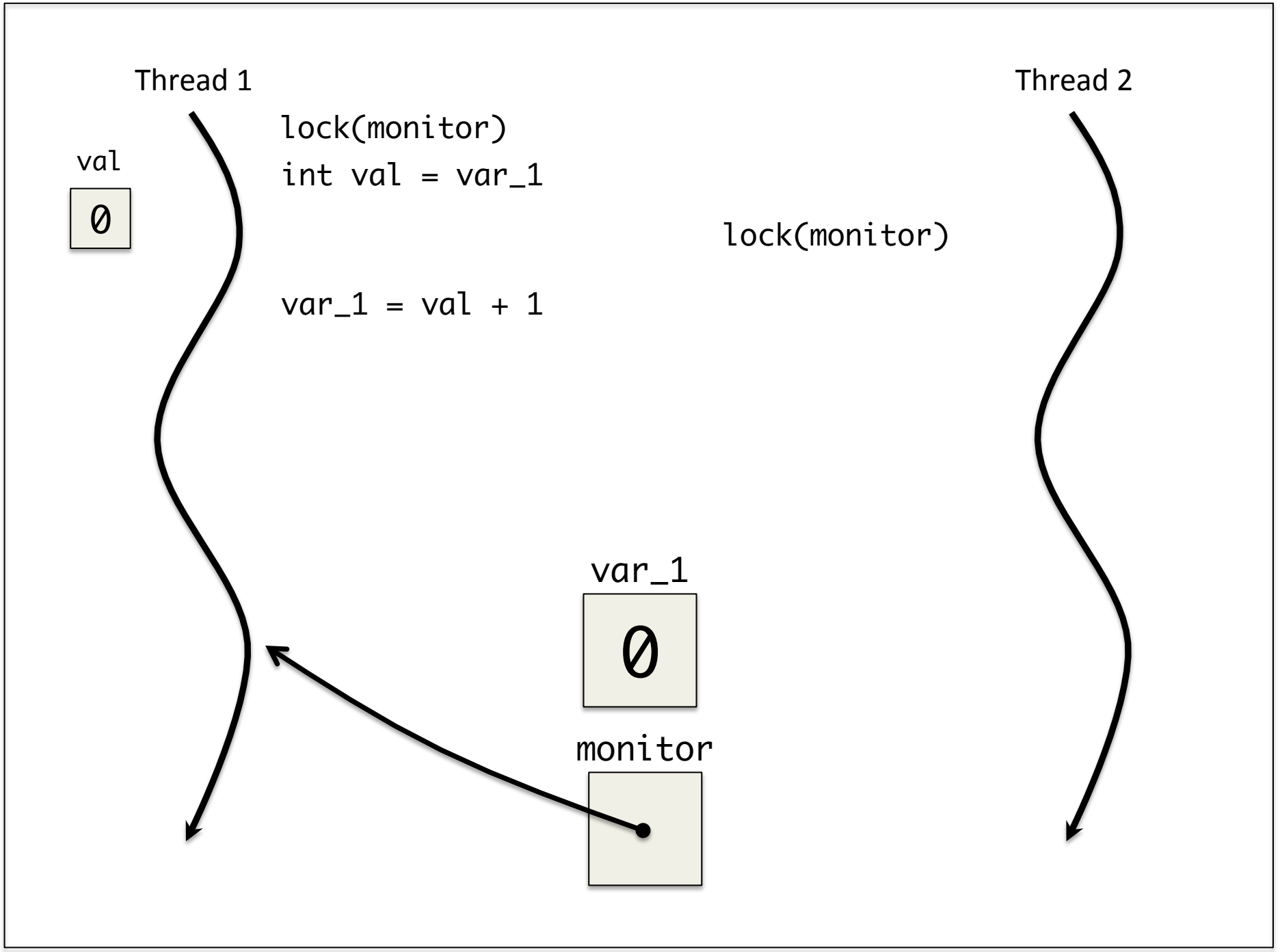

Thread 1
Thread 2
var_1
0
monitor

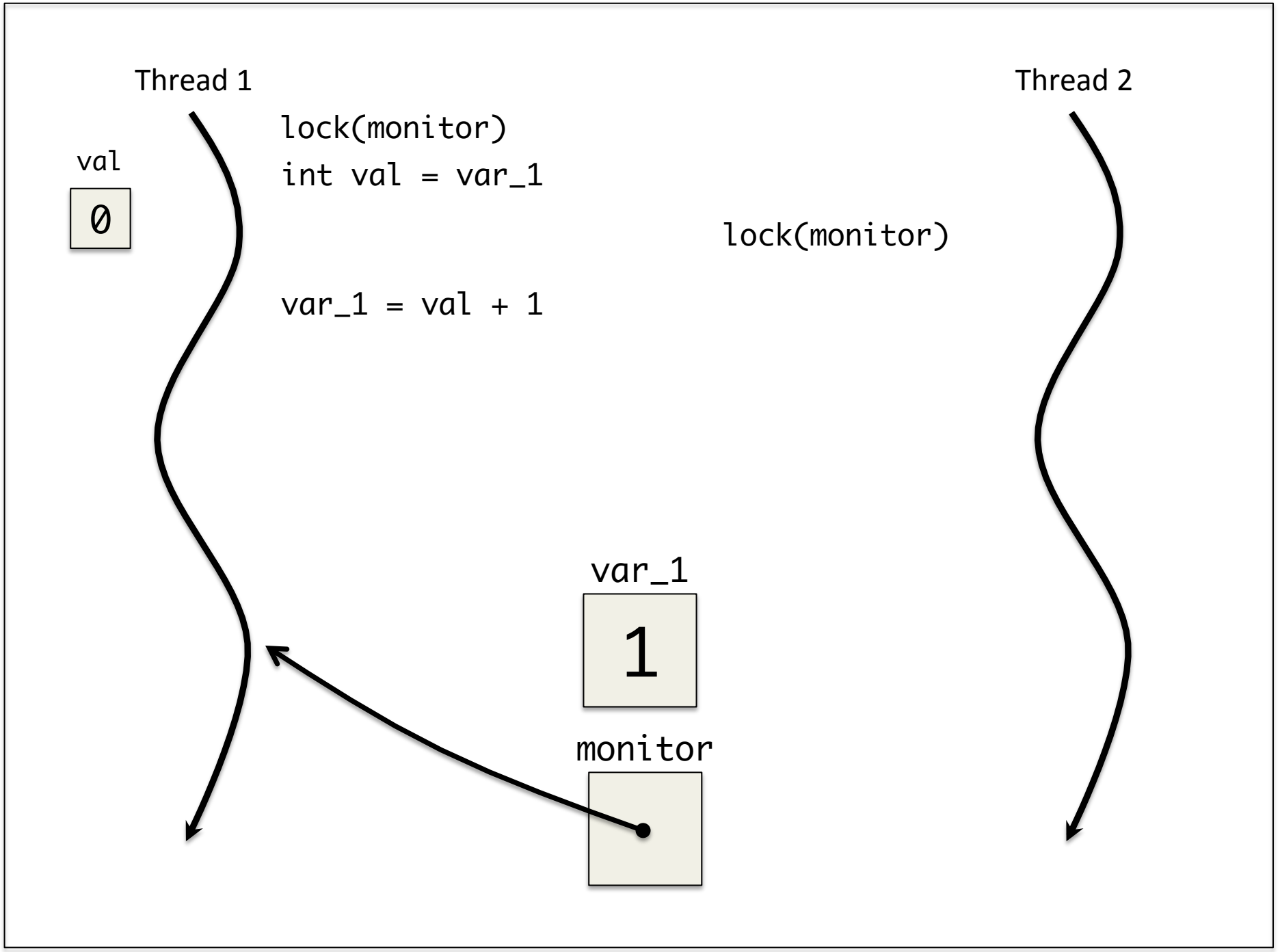

Thread 1
lock(monitor)
Thread 2
var_1
0
monitor

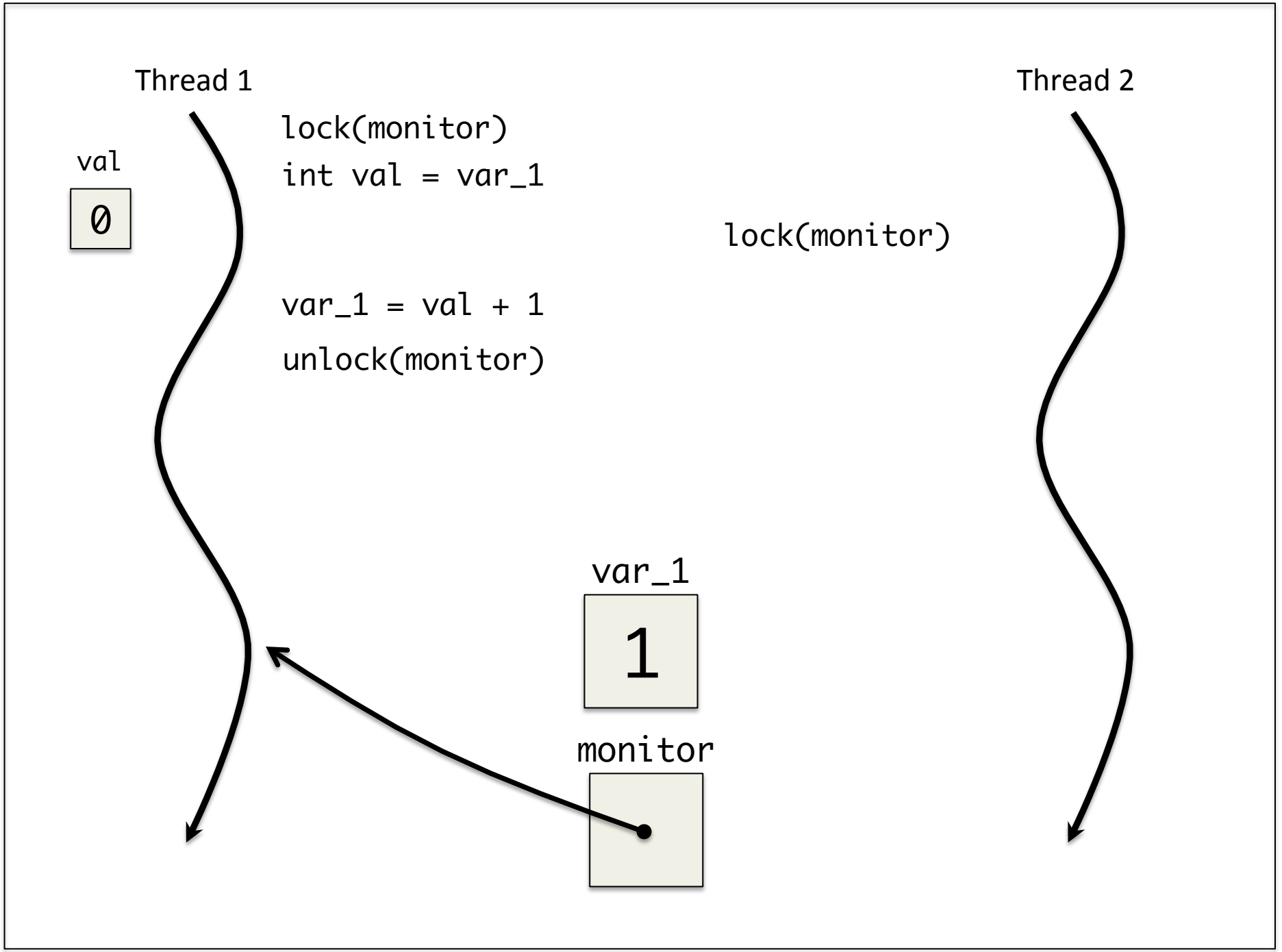

Thread 1
lock(monitor)
Thread 2
var_1
0
monitor

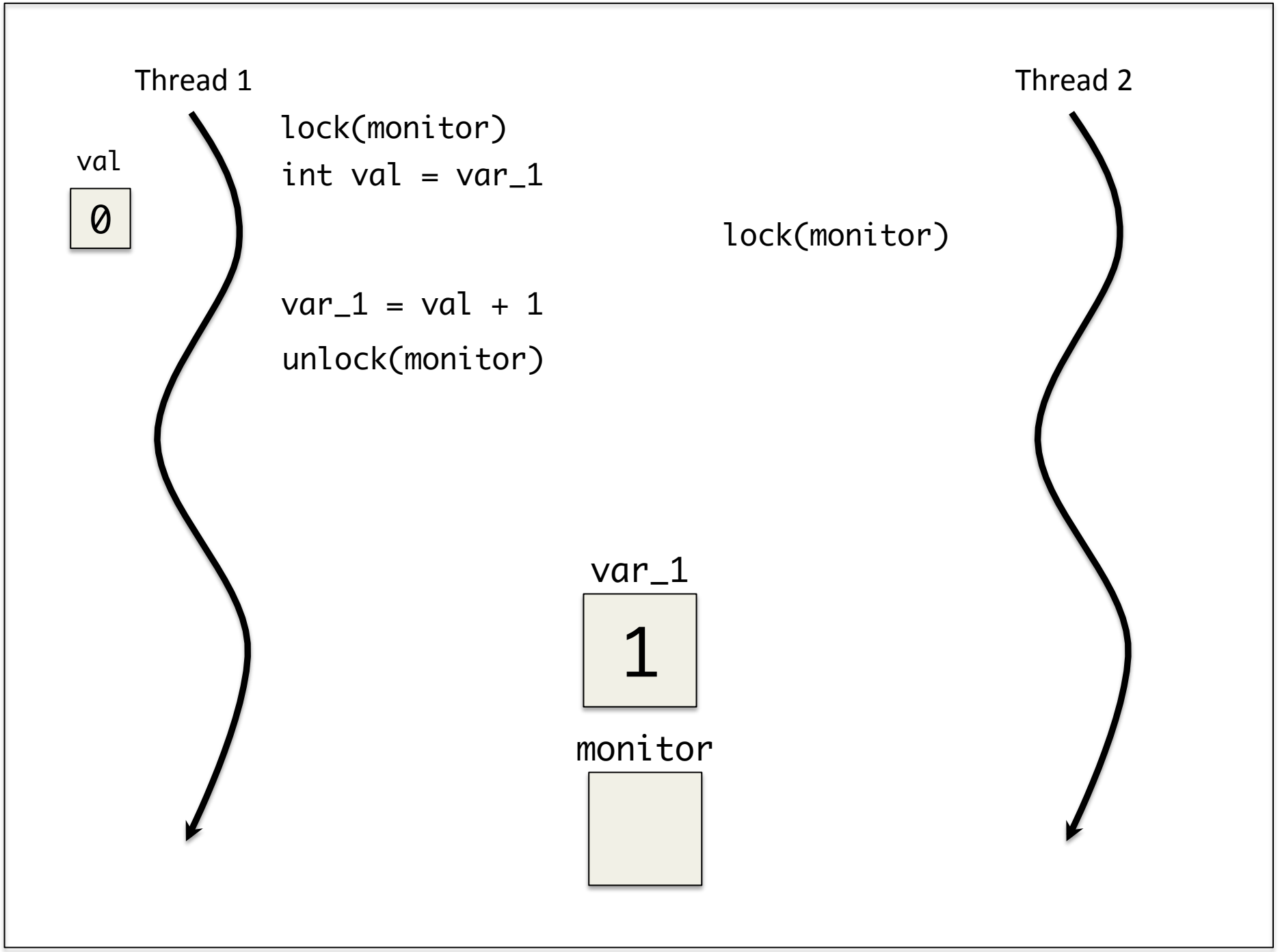

Thread 1
Thread 2
lock(monitor)
int val = var_1
val
0
var_1
0
monitor

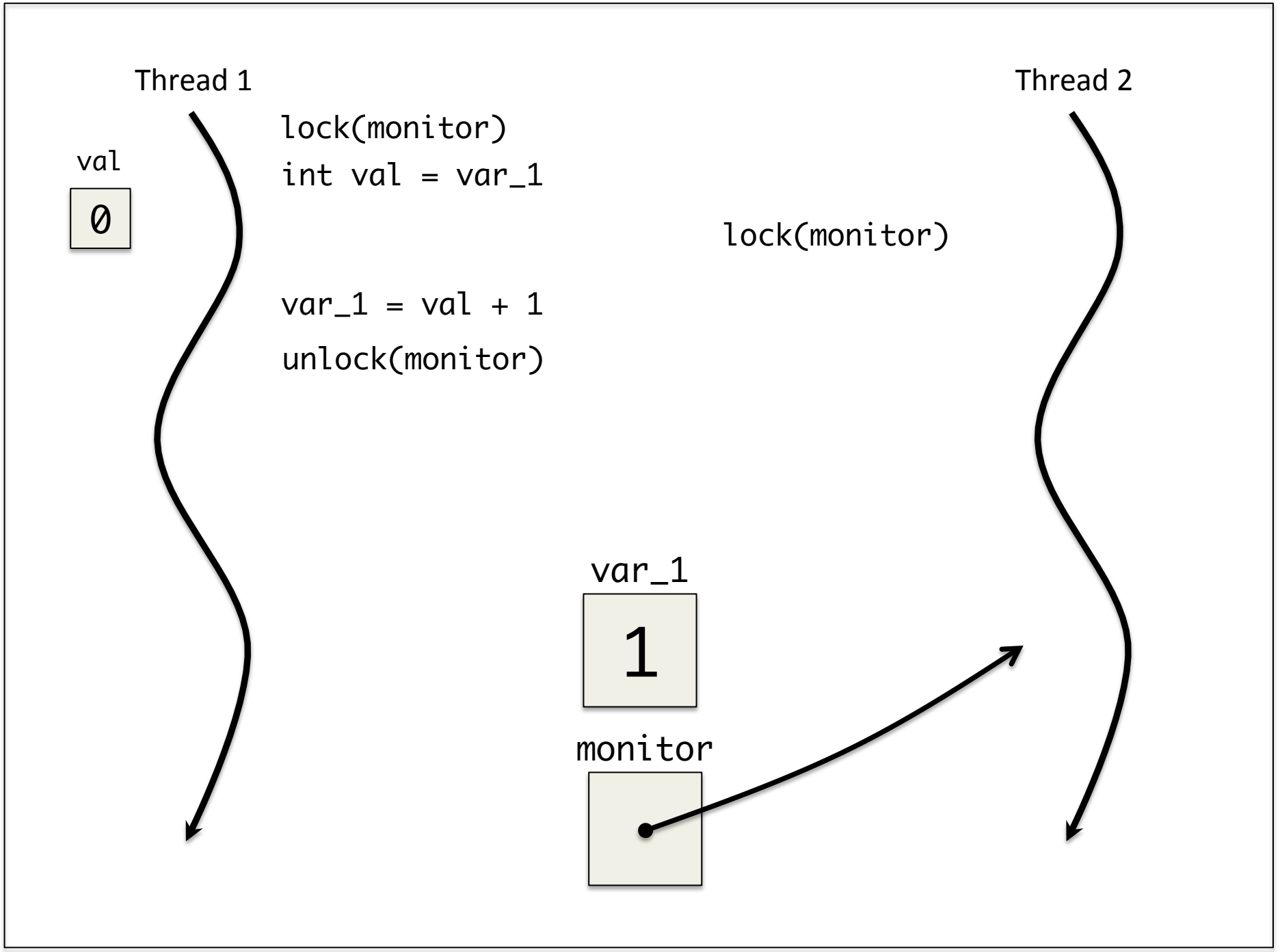

Thread 1
Thread 2
lock(monitor)
int val = var_1
val
0
lock(monitor)
var_1
0
monitor

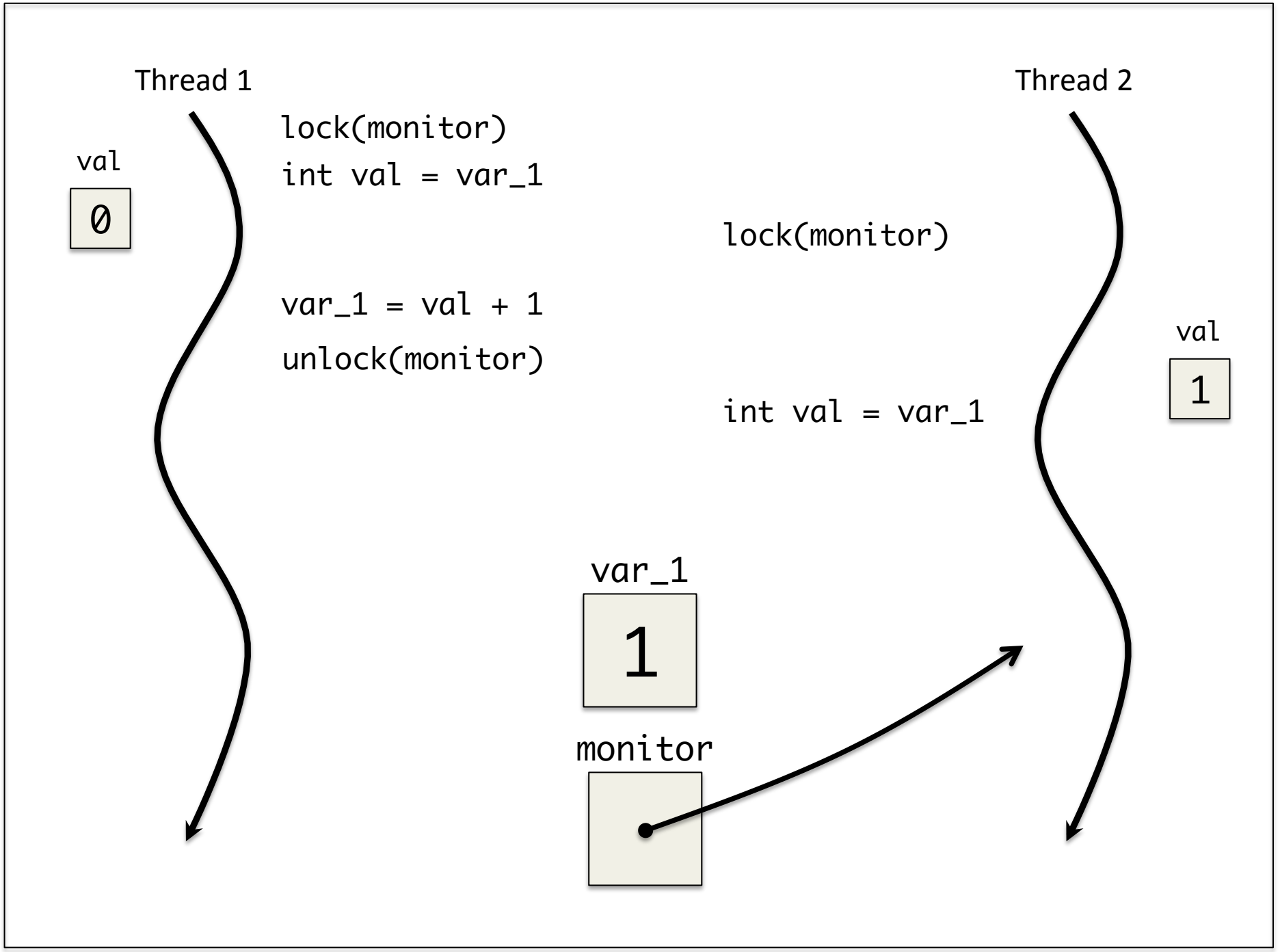

Thread 1
Thread 2
lock(monitor)
int val = var_1
val
0
lock(monitor)
var_1 = val + 1
var_1
0
monitor

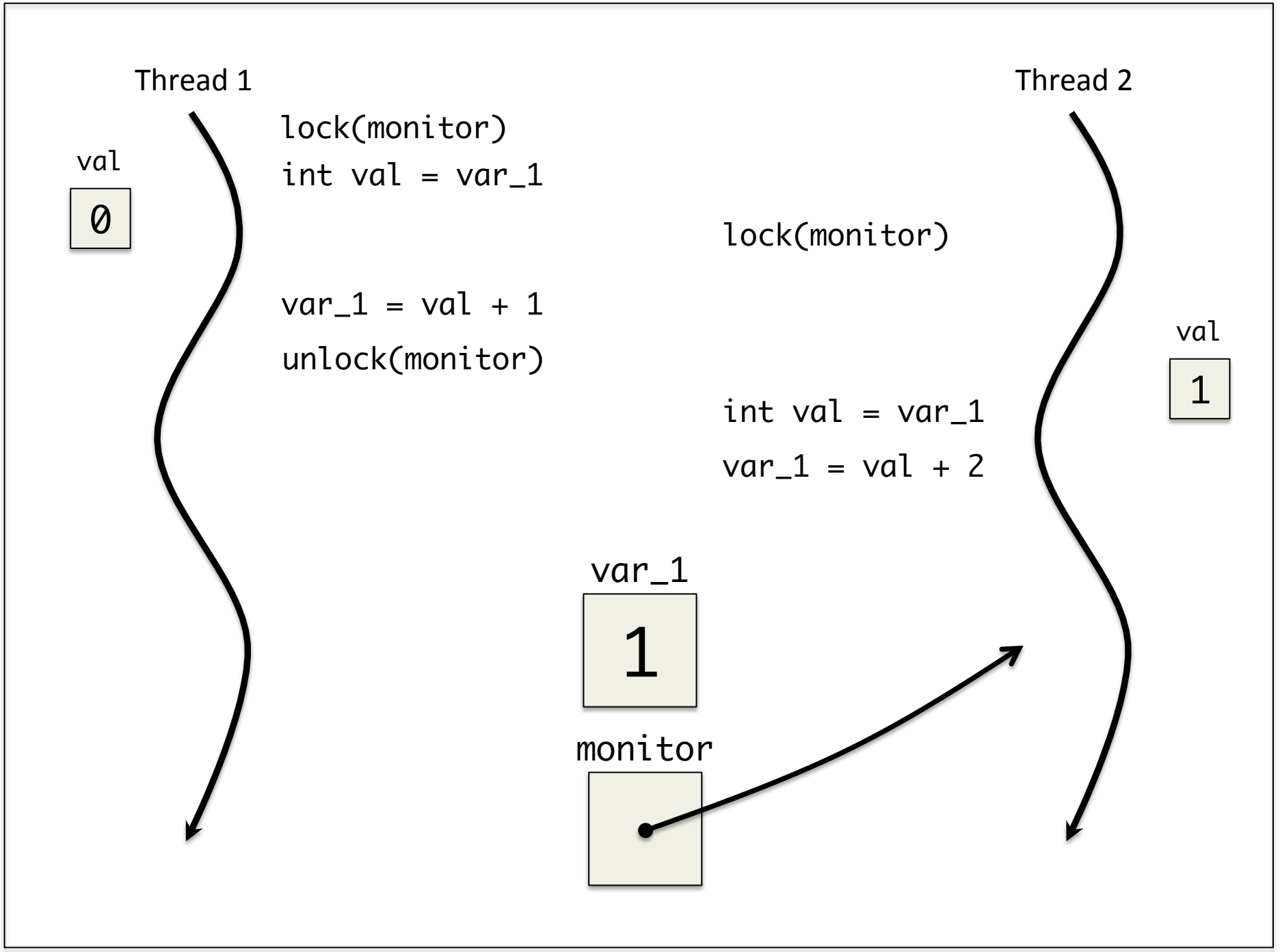

Thread 1
Thread 2
lock(monitor)
int val = var_1
val
0
lock(monitor)
var_1 = val + 1
var_1
1
monitor

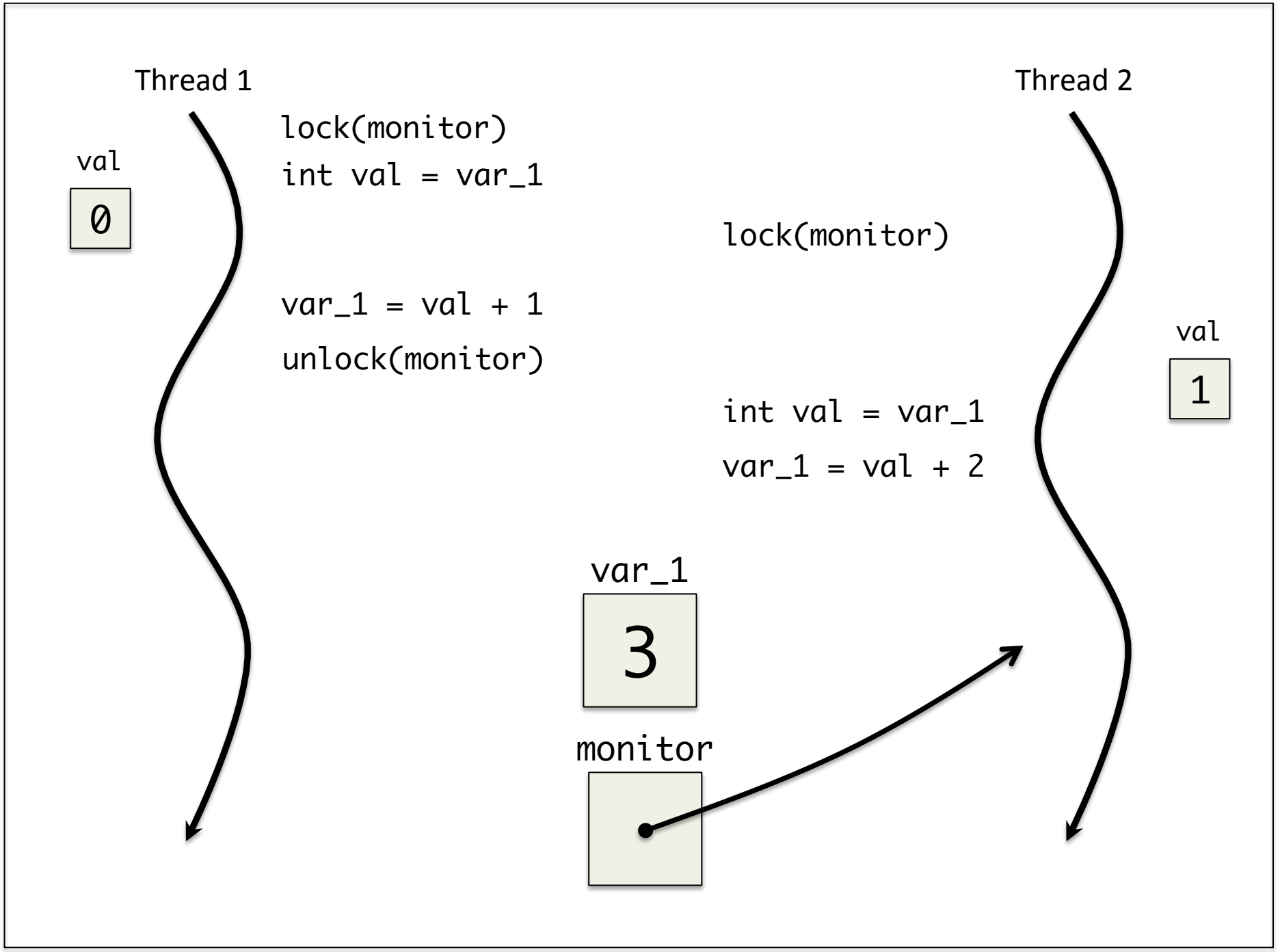

Thread 1
Thread 2
lock(monitor)
int val = var_1
val
0
lock(monitor)
var_1 = val + 1
unlock(monitor)
var_1
1
monitor

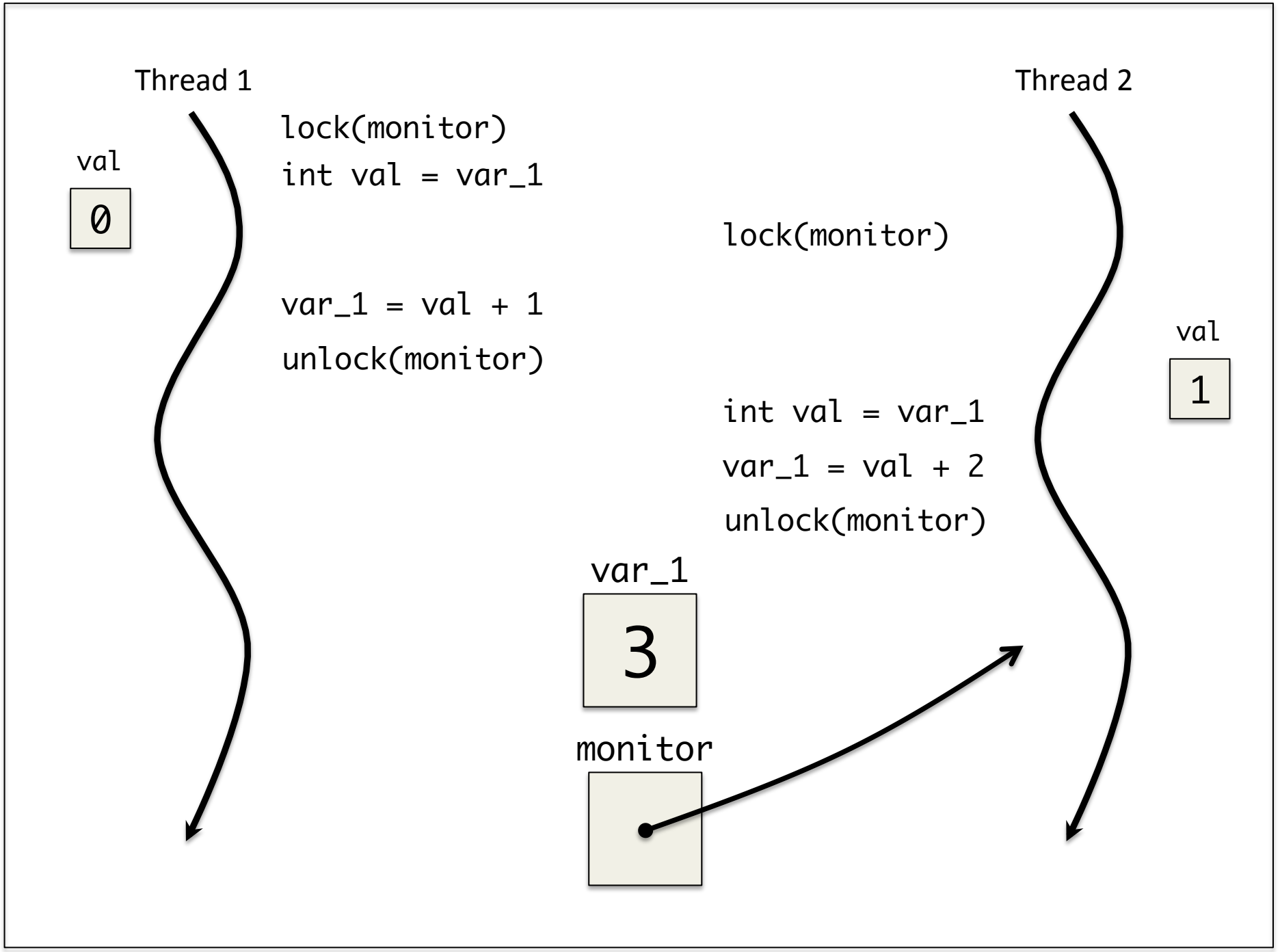

Thread 1
Thread 2
lock(monitor)
int val = var_1
val
0
lock(monitor)
var_1 = val + 1
unlock(monitor)
var_1
1
monitor

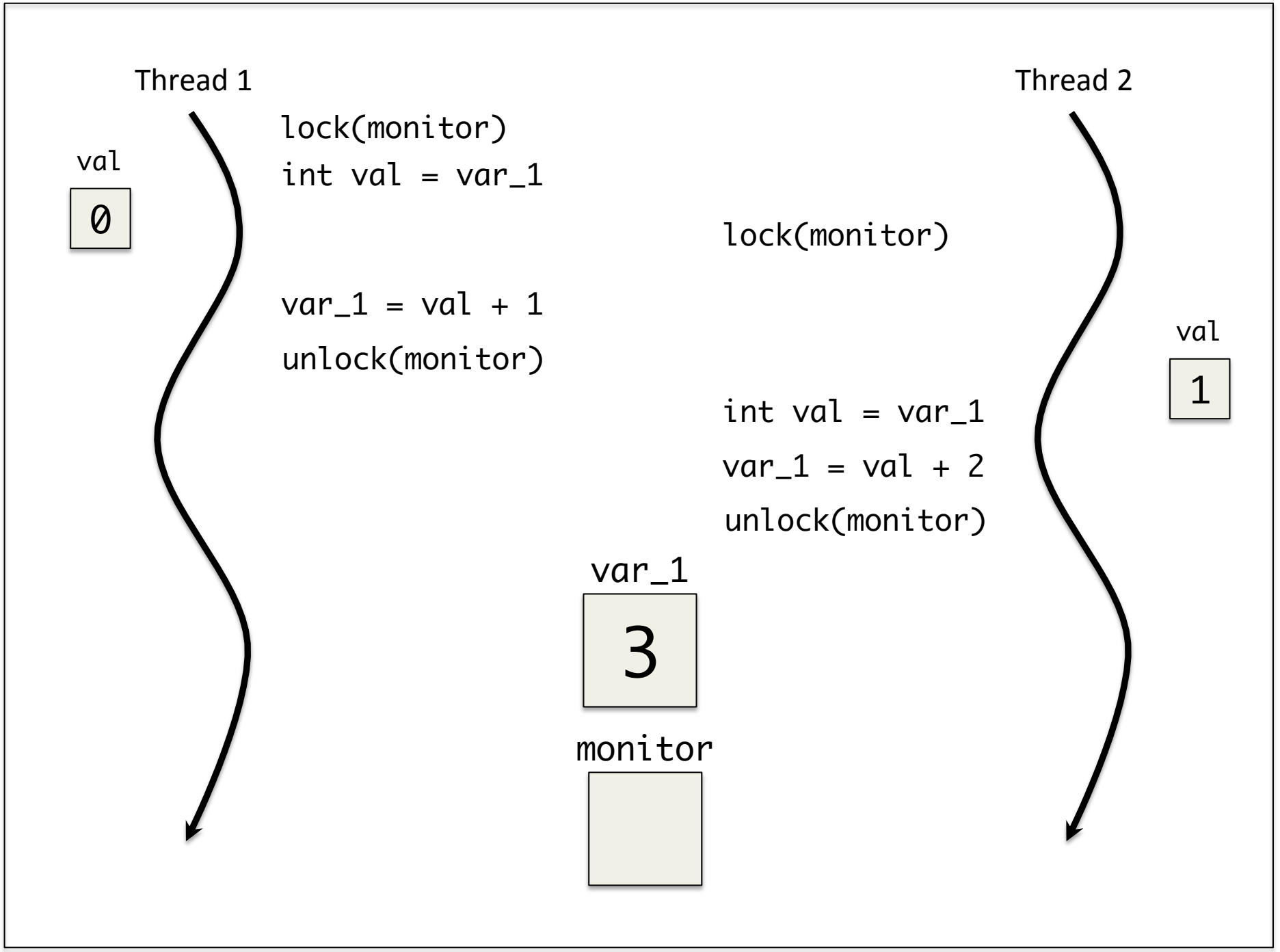

Thread 1
Thread 2
lock(monitor)
int val = var_1
val
0
lock(monitor)
var_1 = val + 1
unlock(monitor)
var_1
1
monitor


Thread 1
Thread 2
lock(monitor)
int val = var_1
val
0
lock(monitor)
var_1 = val + 1
unlock(monitor)
val
1
int val = var_1
var_1
1
monitor


Thread 1
Thread 2
lock(monitor)
int val = var_1
val
0
lock(monitor)
var_1 = val + 1
unlock(monitor)
val
1
int val = var_1
var_1 = val + 2
var_1
1
monitor


Thread 1
Thread 2
lock(monitor)
int val = var_1
val
0
lock(monitor)
var_1 = val + 1
unlock(monitor)
val
1
int val = var_1
var_1 = val + 2
var_1
3
monitor


Thread 1
Thread 2
lock(monitor)
int val = var_1
val
0
lock(monitor)
var_1 = val + 1
unlock(monitor)
val
1
int val = var_1
var_1 = val + 2
unlock(monitor)
var_1
3
monitor


Thread 1
lock(monitor)
int val = var_1
val
0
var_1 = val + 1
unlock(monitor)
Thread 2
lock(monitor)
val
1
int val = var_1
var_1 = val + 2
unlock(monitor)
var_1
3
monitor

Monitor Implementation

- Any object in the JVM can be a monitor
  - Suggests a per-object overhead
  - Store the monitor information in the object header
- Monitors in Java are reentrant
  - One thread can acquire a monitor multiple times
  - Need to keep a count of entries
- Store the thread identifier in the monitor
  - Easy to know whether a thread already owns it

Monitors in Bytecode

- Two synchronization bytecode instructions
  - `MonitorEnter`
  - `MonitorExit`
- Enter and exit must be matched on all paths
  - Checked by the verifier
- Operations are implicit for synchronized methods
  - Synchronized state determined by access flag
  - VM automatically wraps method in enter and exit

Contention

- Contention is when threads compete for a lock
  - Cost is always higher than uncontended lock
  - Actual overhead is implementation dependent
  - Can reduce through application design
- VM contains a contention manager component
  - Tracks threads waiting for a monitor
  - Uses some algorithm to decide which goes next
  - Wide variety of designs with different characteristics

Thin Locks

- Most monitors are uncontended
  - Actually, most objects are never locked
- Monitor entries are frequently actually reentries
  - The current thread owns the monitor

- Thin locks assume no contention
  - Use CAS to set the monitor word to the thread ID
  - Succeed if the monitor was set to zero
  - Slow path if the monitor was contended

Thin Lock Fallback

- The CAS operation fails if the monitor is held
- The current thread may already hold it
  - This is a re-entry there is no contention
  - Increase the recursive entry count
  - This doesn't have to be an atomic operation
- Some other thread has locked the monitor
  - Genuine contention

Handling Contention

- Can use spin lock for a short time
  - Repeatedly try to acquire the monitor
  - Wastes processor time
  - Context switching is expensive
- Eventually, we may need to stop spinning
  - How long to spin depends on the platform
  - At this point, the thin lock is no longer useful

Thick Locks

- Thin locks don't hold much information
  - Just the thread ID and the entry count
- We can inflate thin locks on contention
  - Allocate a new object
  - Reference from the lock word
  - Use stolen bits to indicate thick lock
- Store additional information in that object
  - OS-level mutex information
  - Overflow entry counts

Biased Locking

- CAS operations are relatively expensive
  - Involve synchronizing across hardware threads
- Most objects are locked by at most one thread
  - Contention is the uncommon case
- Biased locking is cheap for one thread
  - Bias the monitor towards that thread
  - Only that thread can acquire the monitor without CAS
  - Bias persists between monitor entries

Bias Revocation

- Bias removed if a second thread needs the monitor
  - More frequent than contention
  - Can happen when the owner doesn't hold the monitor
- Lock needs to be reset
  - Re-bias towards the new thread
  - Revert to non-biased mode
- Removing bias is good for contended monitors
  - Fewer revocation actions

CAS Overhead

- Biased locking is a response to CAS cost
  - Overhead of CAS must outweigh revocation cost
- CAS is a serializing operation
  - Memory accesses must be ordered
  - Interferes with processor instruction reordering
- Modern processors optimize CAS instruction
  - Biased locking is less valuable as a result

Lock Elision

- Remove unnecessary monitor operations
  - JIT compiler optimization
  - Based on static analysis
- Locks can be removed in several cases
  - Monitor does not escape the method
  - Protected operations are on local objects
  - Protected operations can be shown to be safe

Cooperation

- Mutual exclusion prevents threads from interfering
  - Protect critical code
  - Serialize the order of operations
- Cooperation allows threads to interact
  - Allows threads to work together
- Java implements both using monitors

Cooperation Primitives

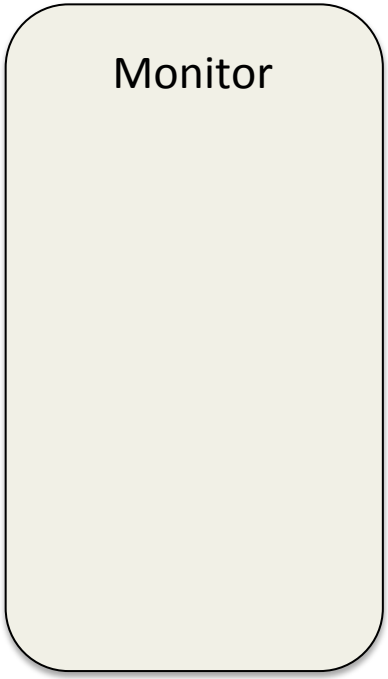
- Java provides a set of basic operations
  - `wait`, `notify`, `notifyAll`, `join`
  - Thread interruption
- Early versions included more explicit control
  - Thread `stop`, `suspend` and `restart`
- These methods are now deprecated
  - Thread death releases all monitors
  - Protected objects may be in a bad state
  - No way for a developer to handle the stop event

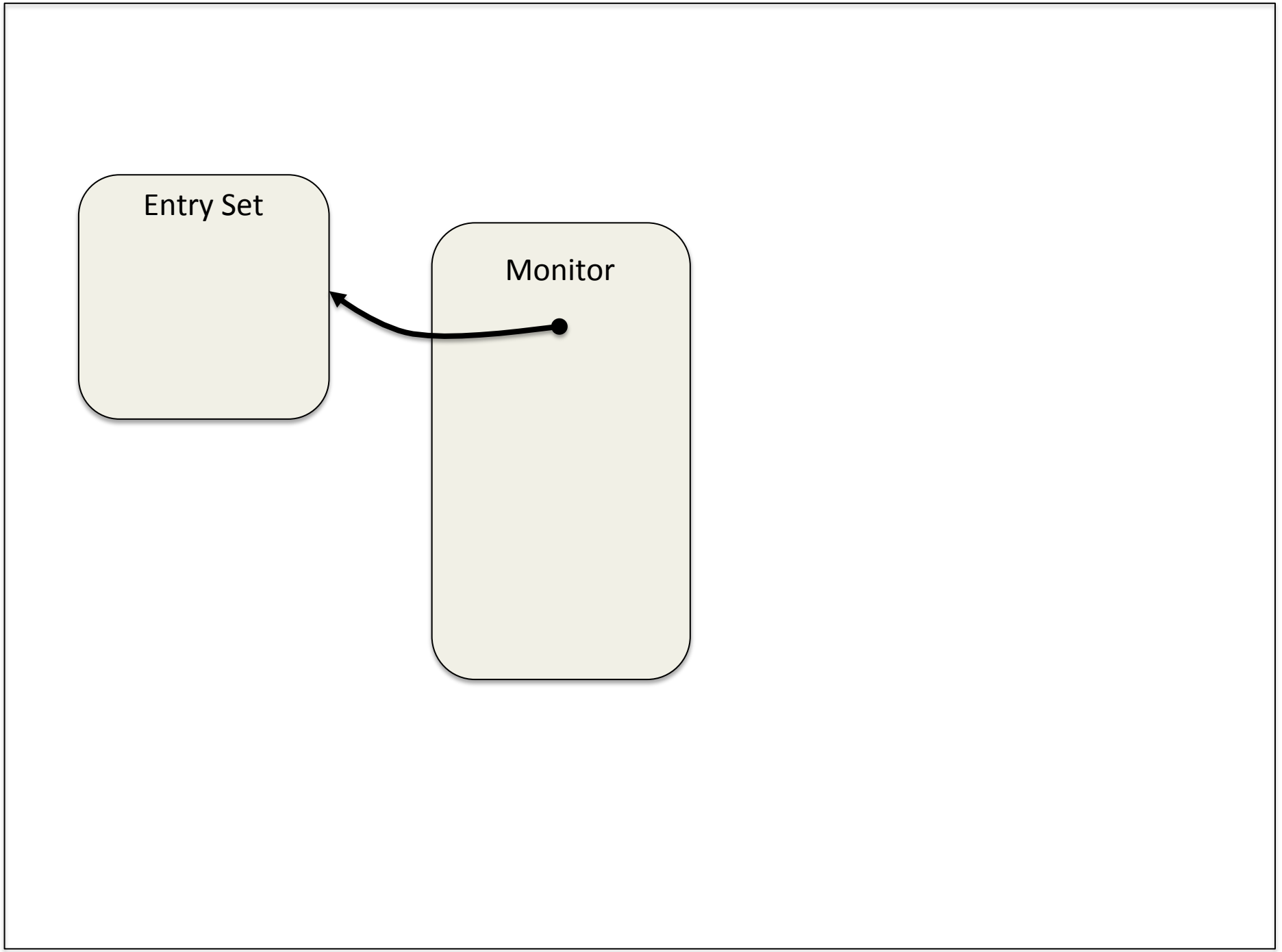
Waiting On A Monitor

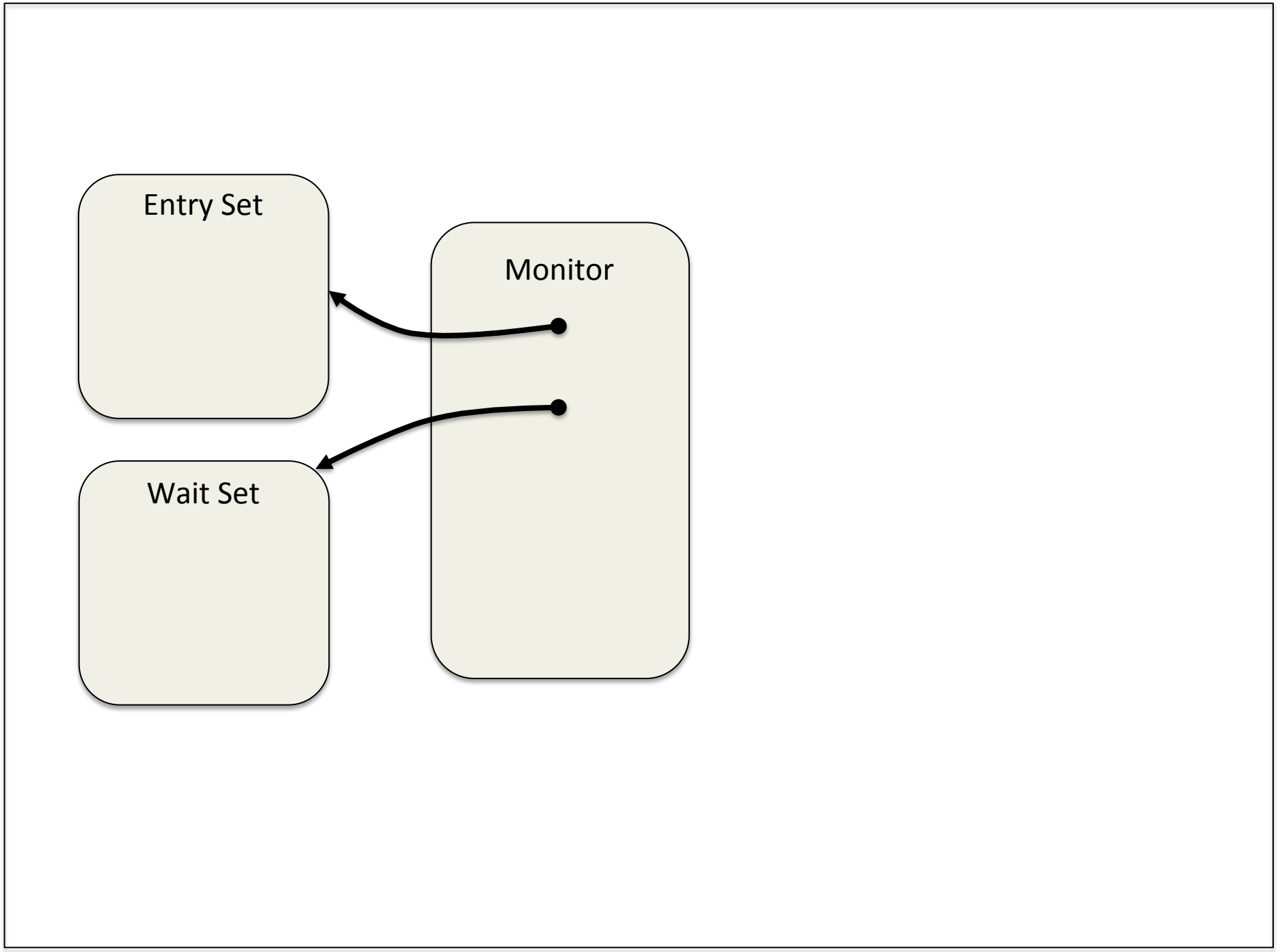
- Two sets of objects per monitor
  - Entry set contains threads trying to acquire lock
  - Wait set contains threads that have called wait
- Sets have similar functionality
  - Both contain suspended threads
- Threads are released in different circumstances

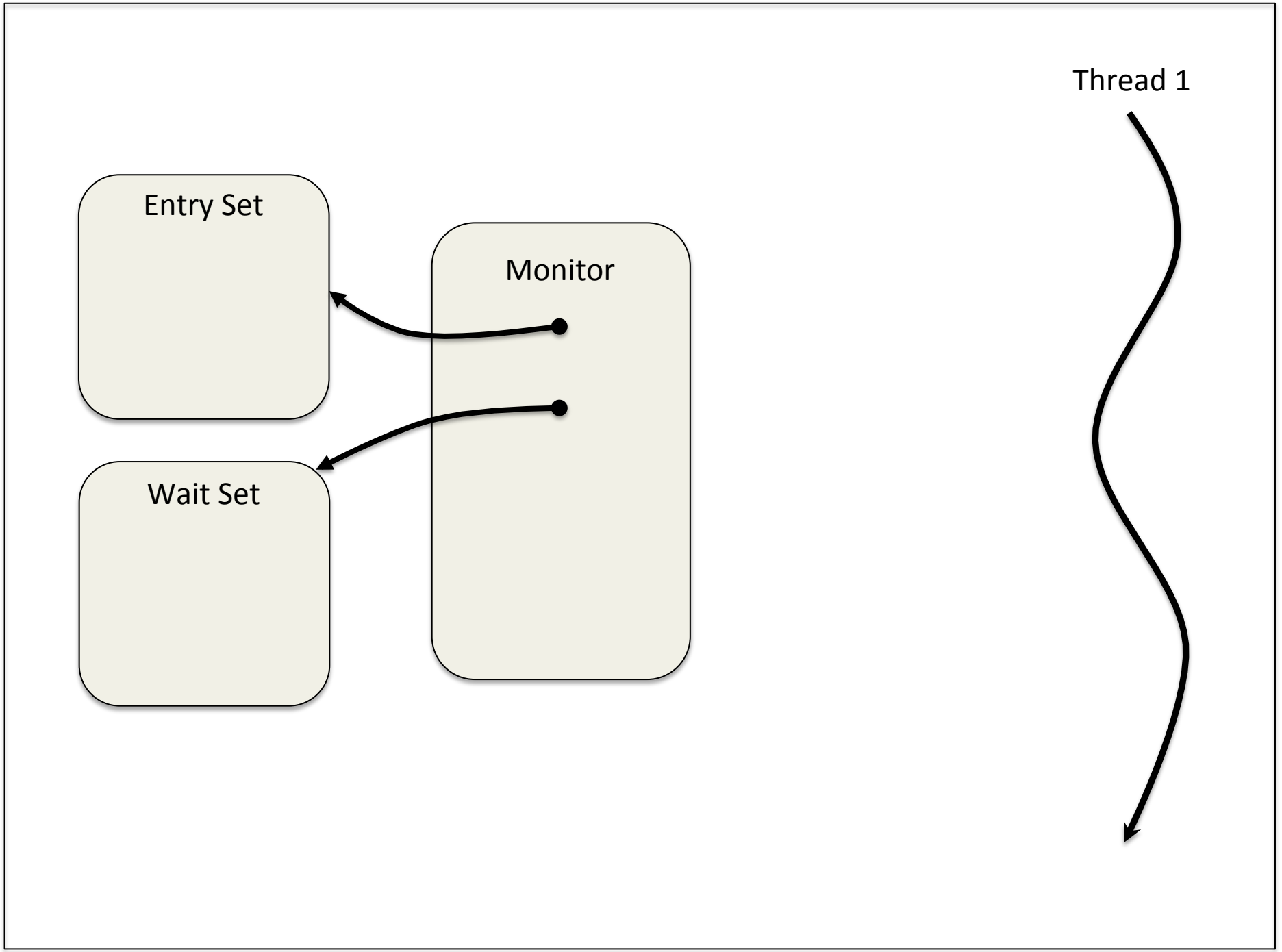
Waiting and Notification

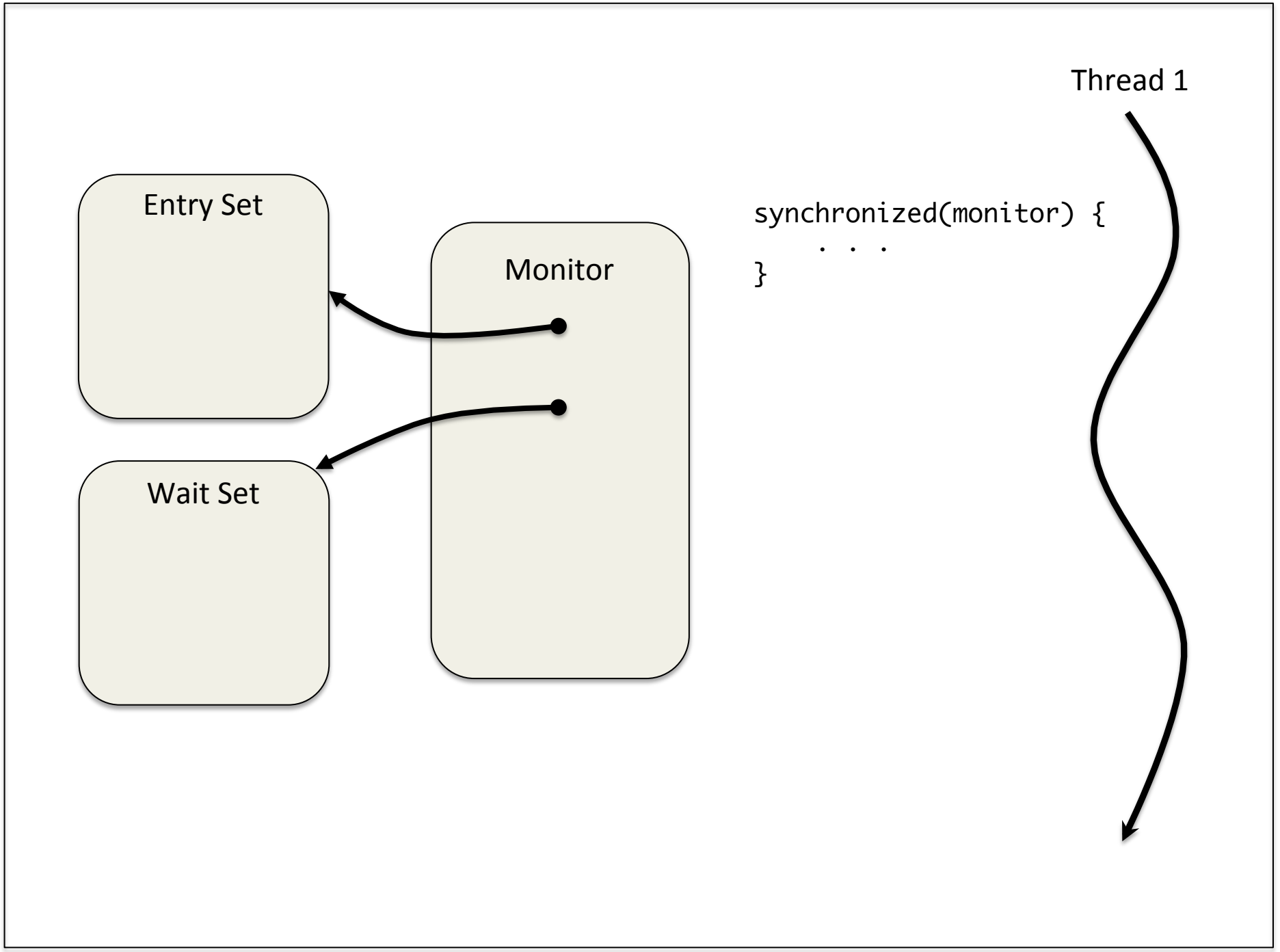
- Thread must hold the monitor before either action
  - Must have been scheduled by contention manager
- Waiting releases the monitor
  - Moves the thread to the wait set
  - Schedules some thread from the entry set
- Notification retains the monitor
  - Moves a waiting thread to the entry set
  - Can notify all threads at once

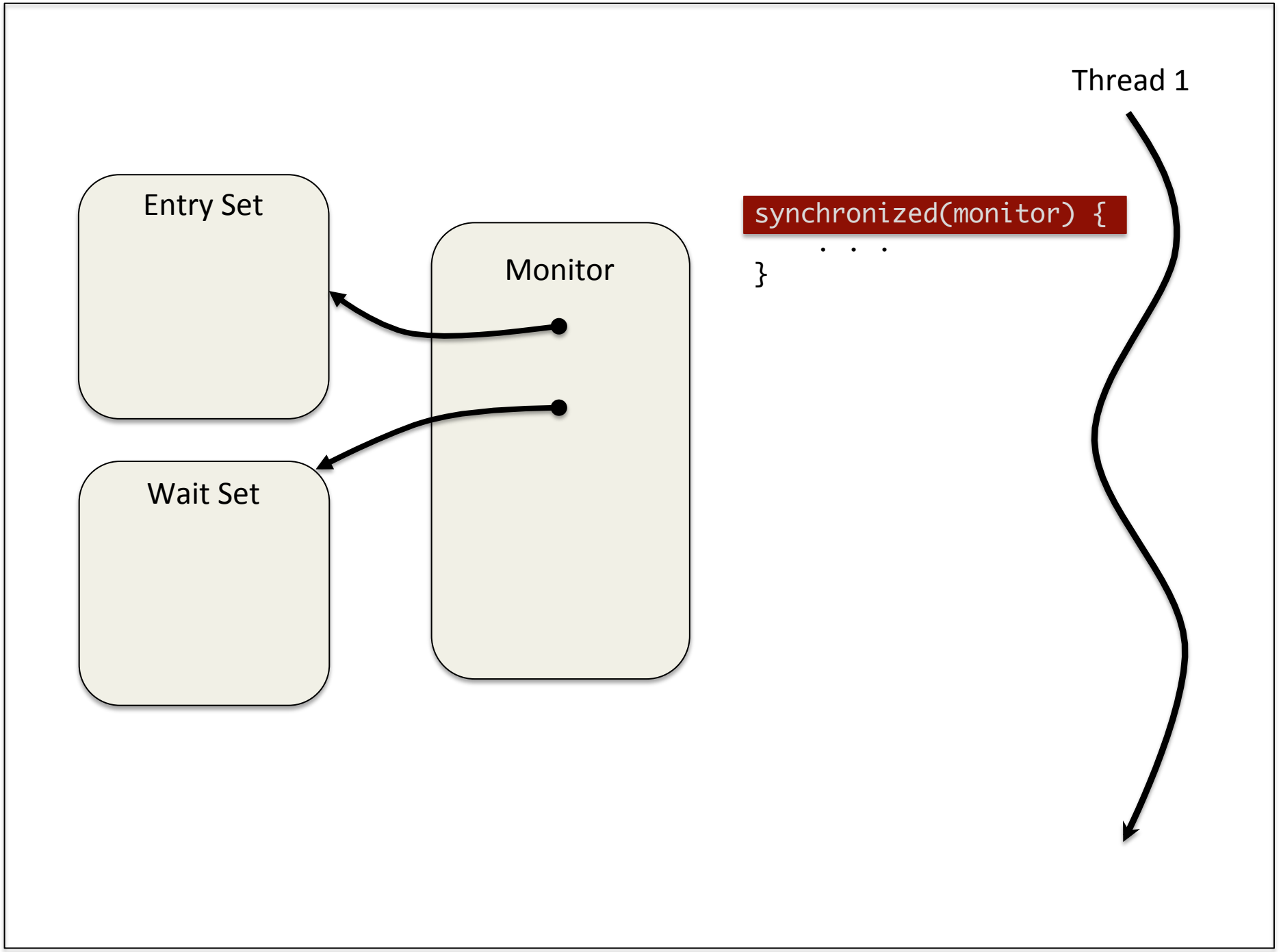

Monitor

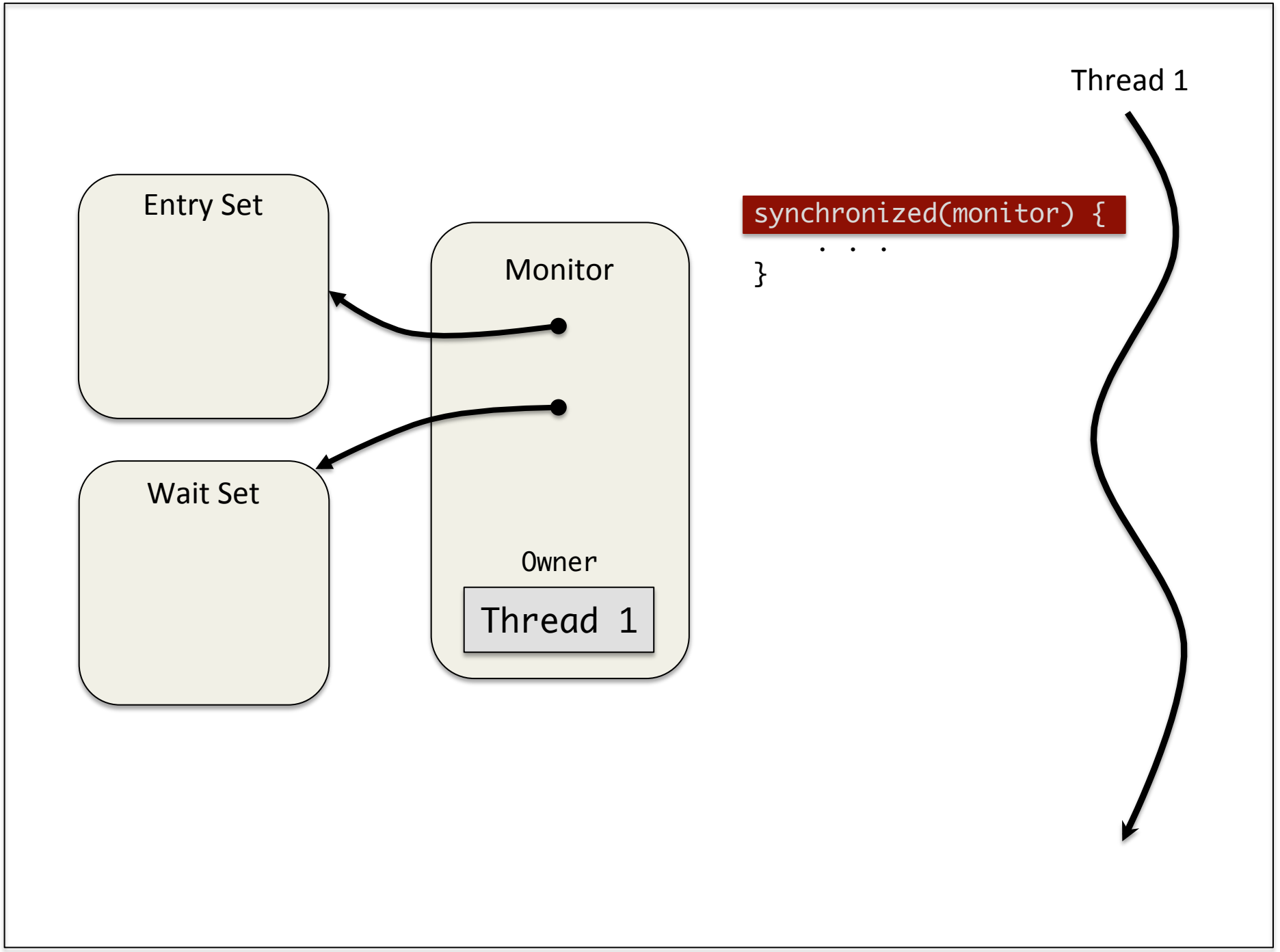

Entry Set
Monitor

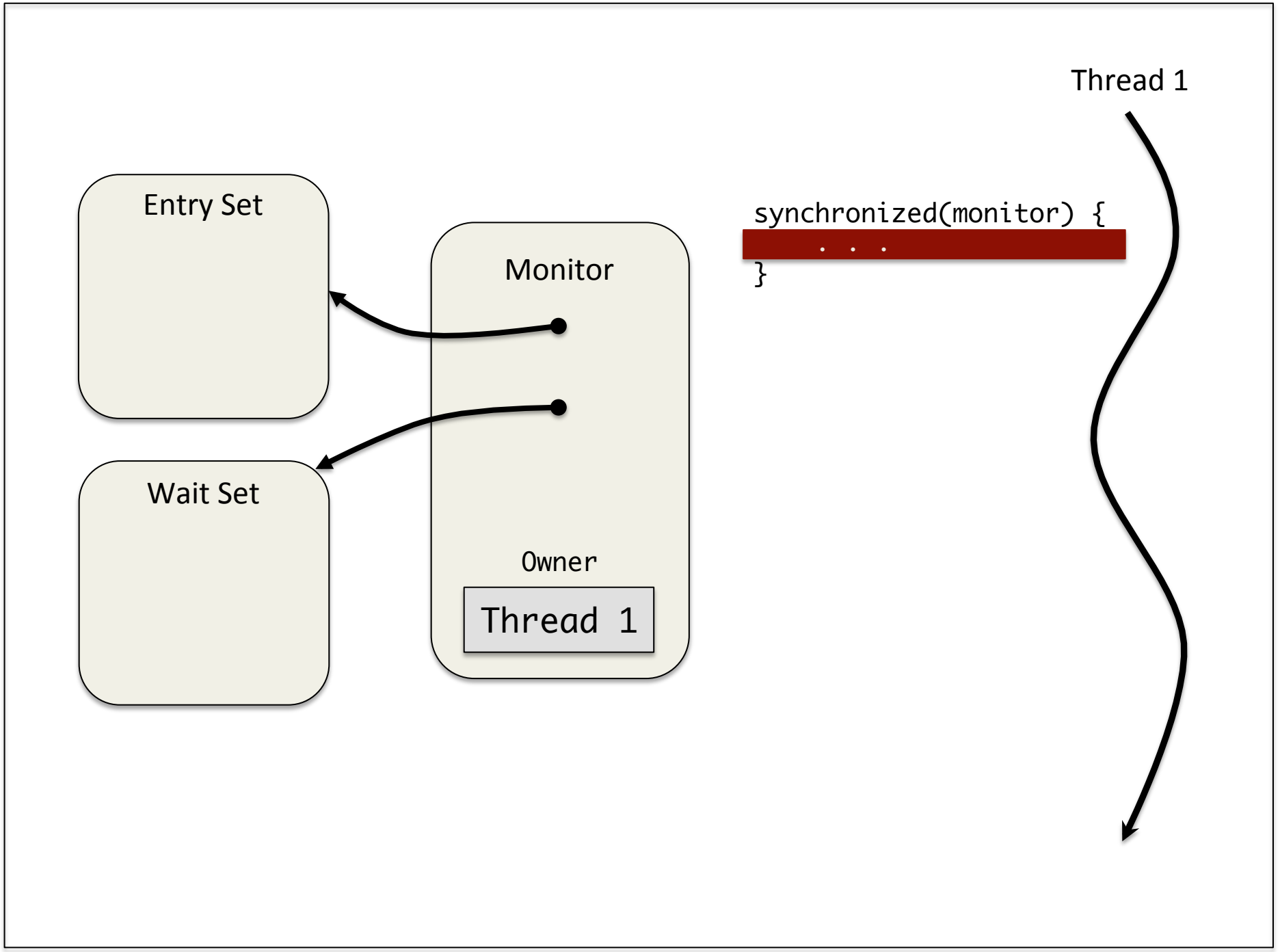

Entry Set
Monitor
Wait Set

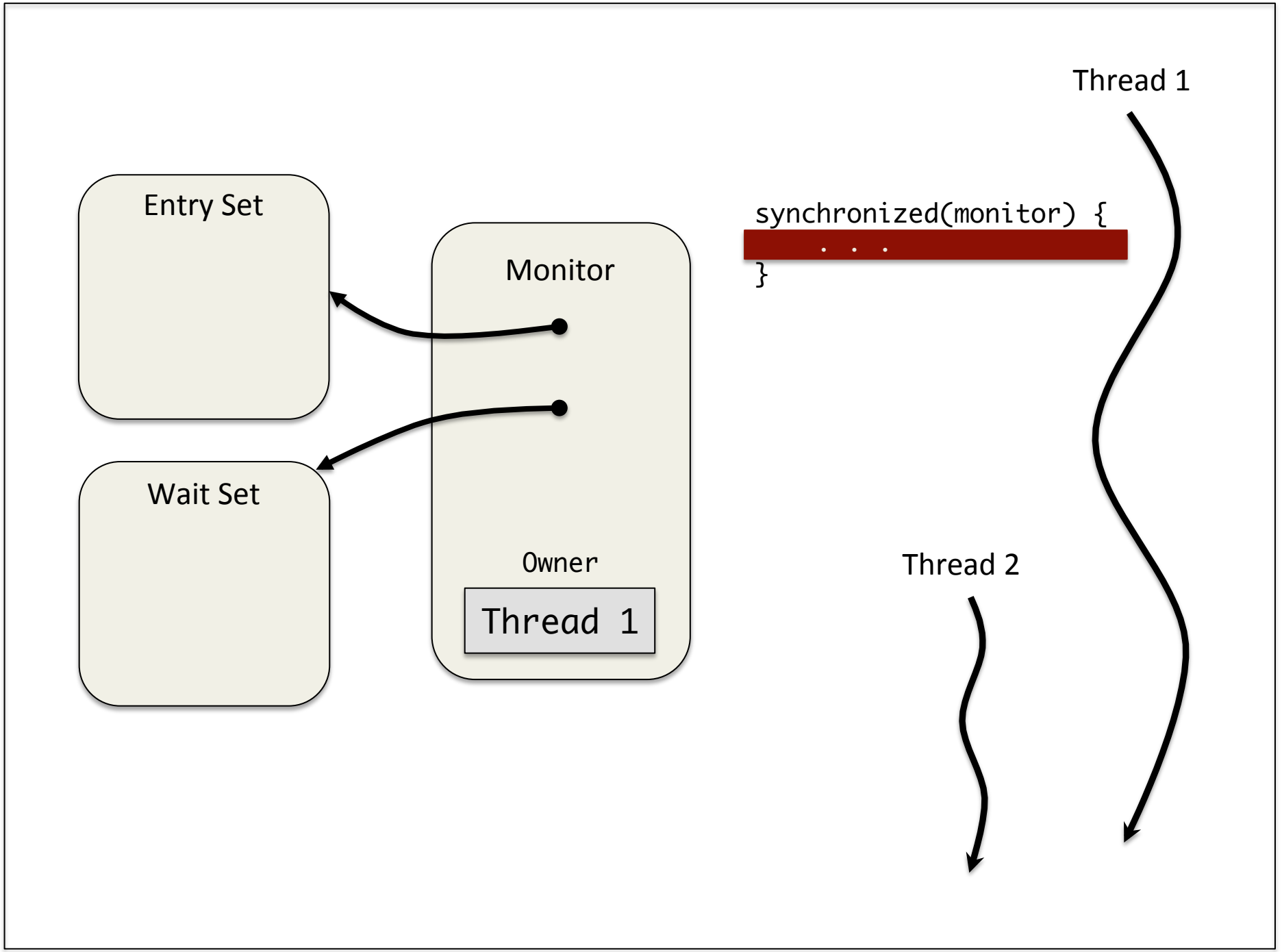

Thread 1
Entry Set
Monitor
Wait Set

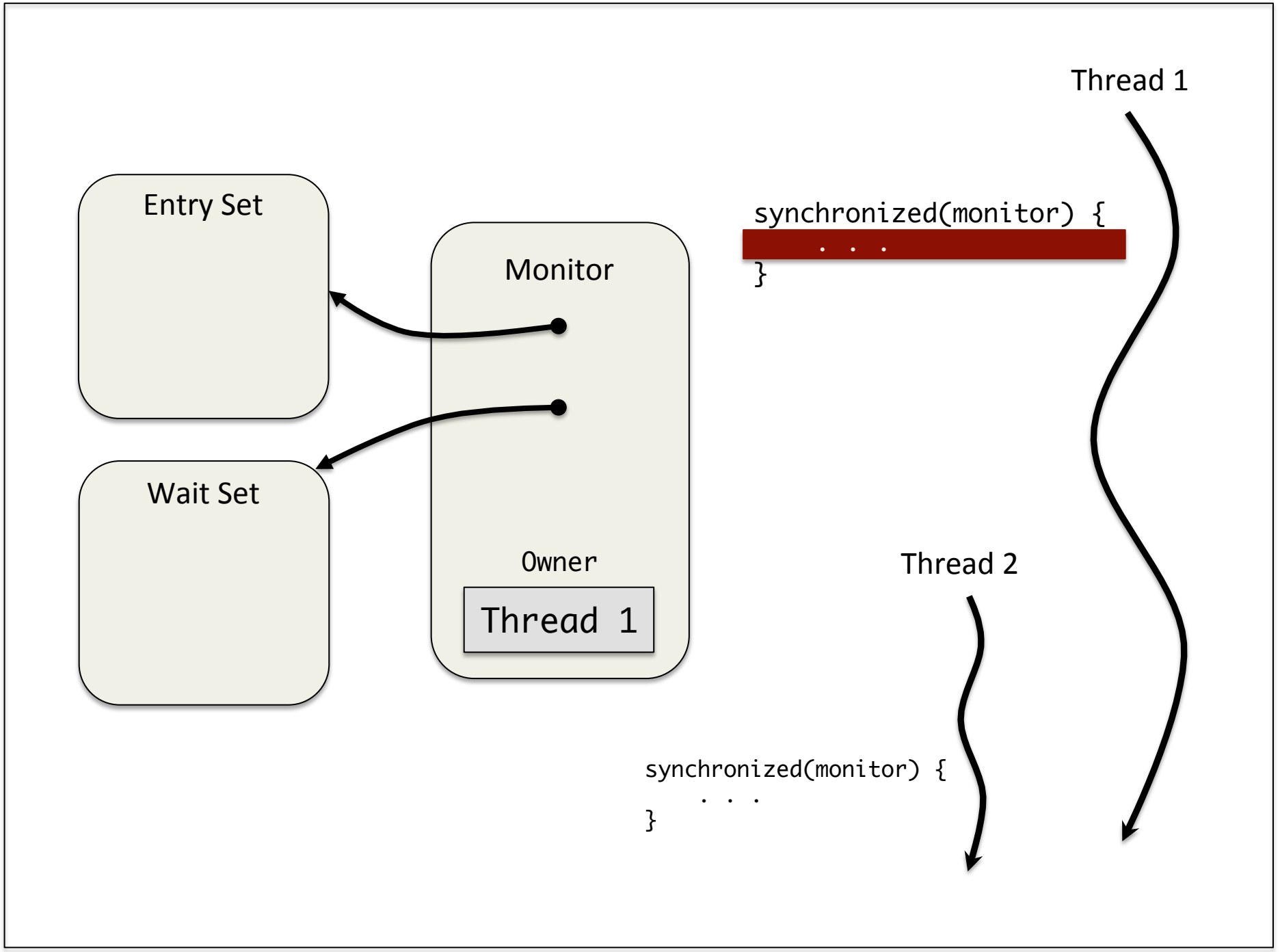

Thread 1
Entry Set
Monitor
Wait Set
synchronized(monitor) {
. . .
}

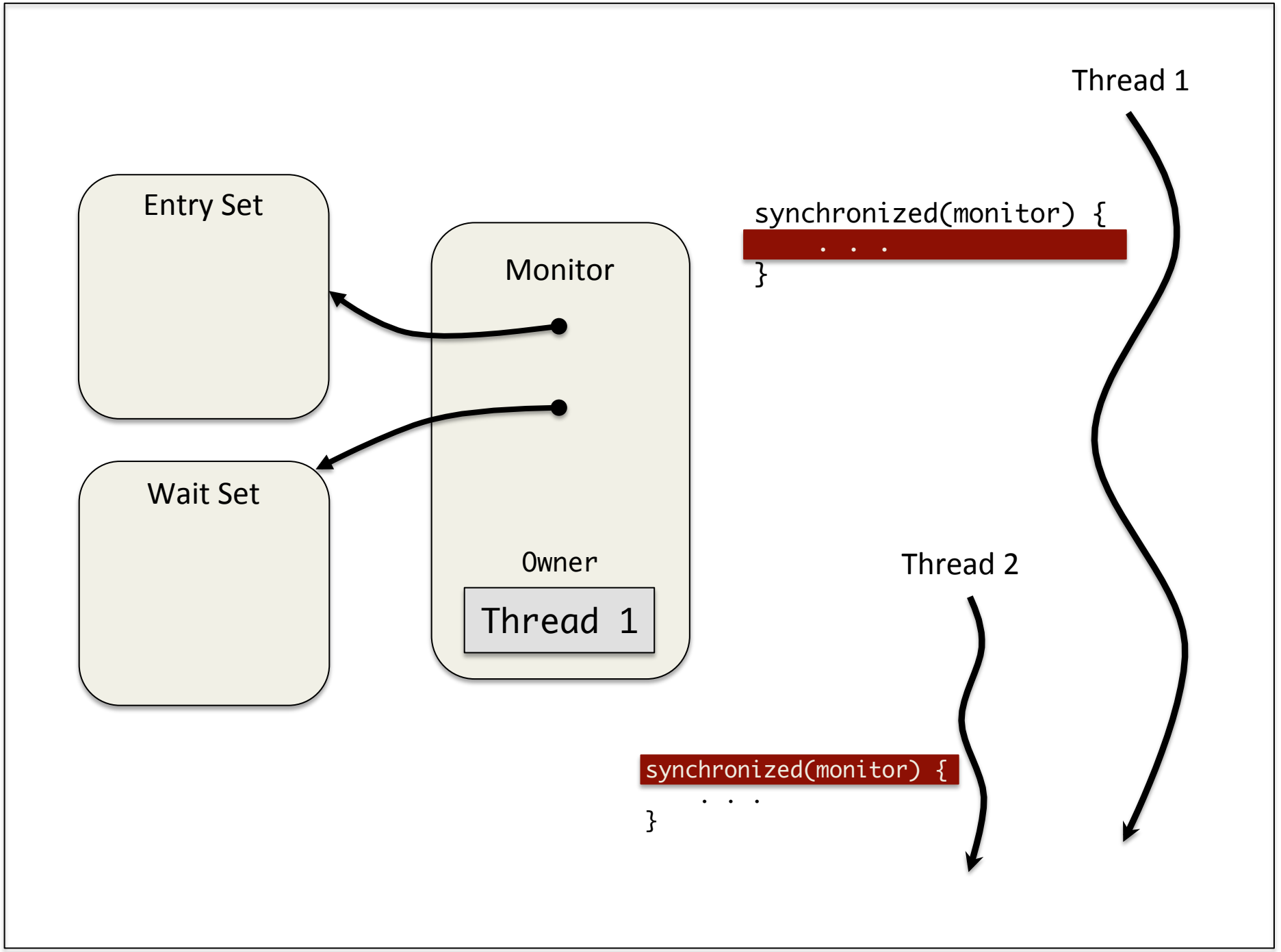

Thread 1
Entry Set
Monitor
Wait Set
synchronized(monitor) {
. . .
}

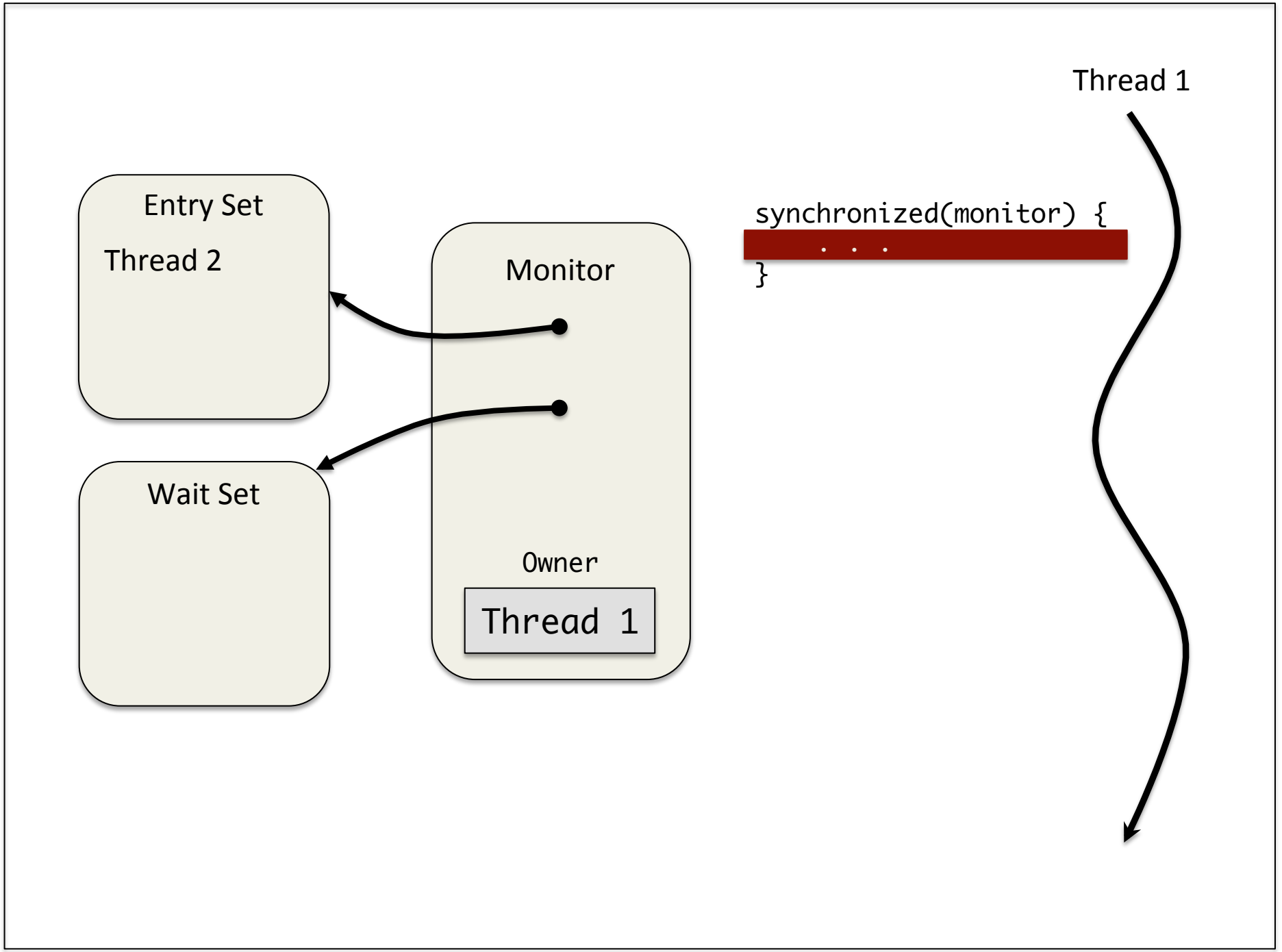

Thread 1
Entry Set
Monitor
synchronized(monitor) {
. . .
}
Wait Set
Owner
Thread 1

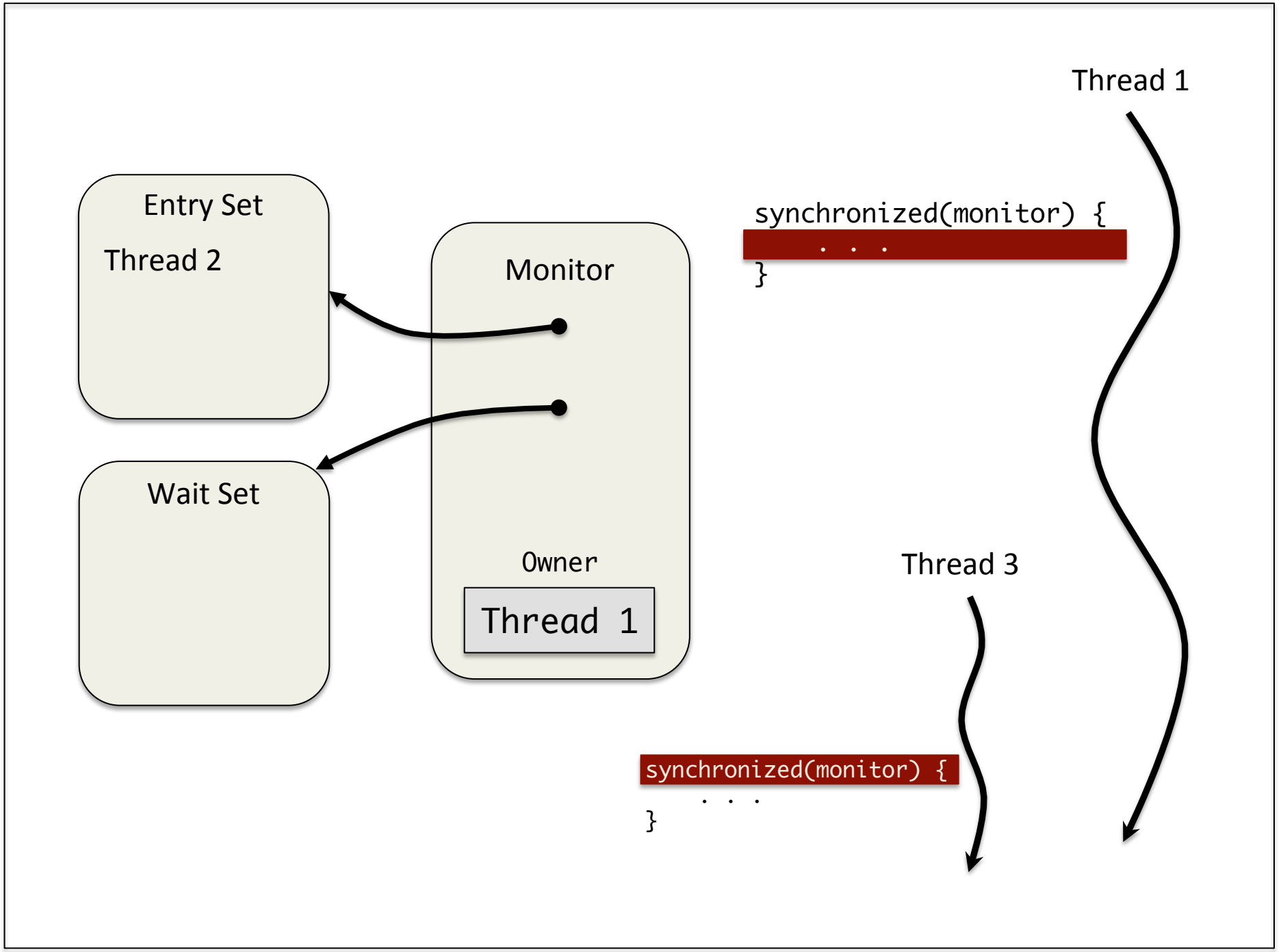

Thread 1
Entry Set
Monitor
synchronized(monitor) {
. . .
}
Wait Set
Owner
Thread 1

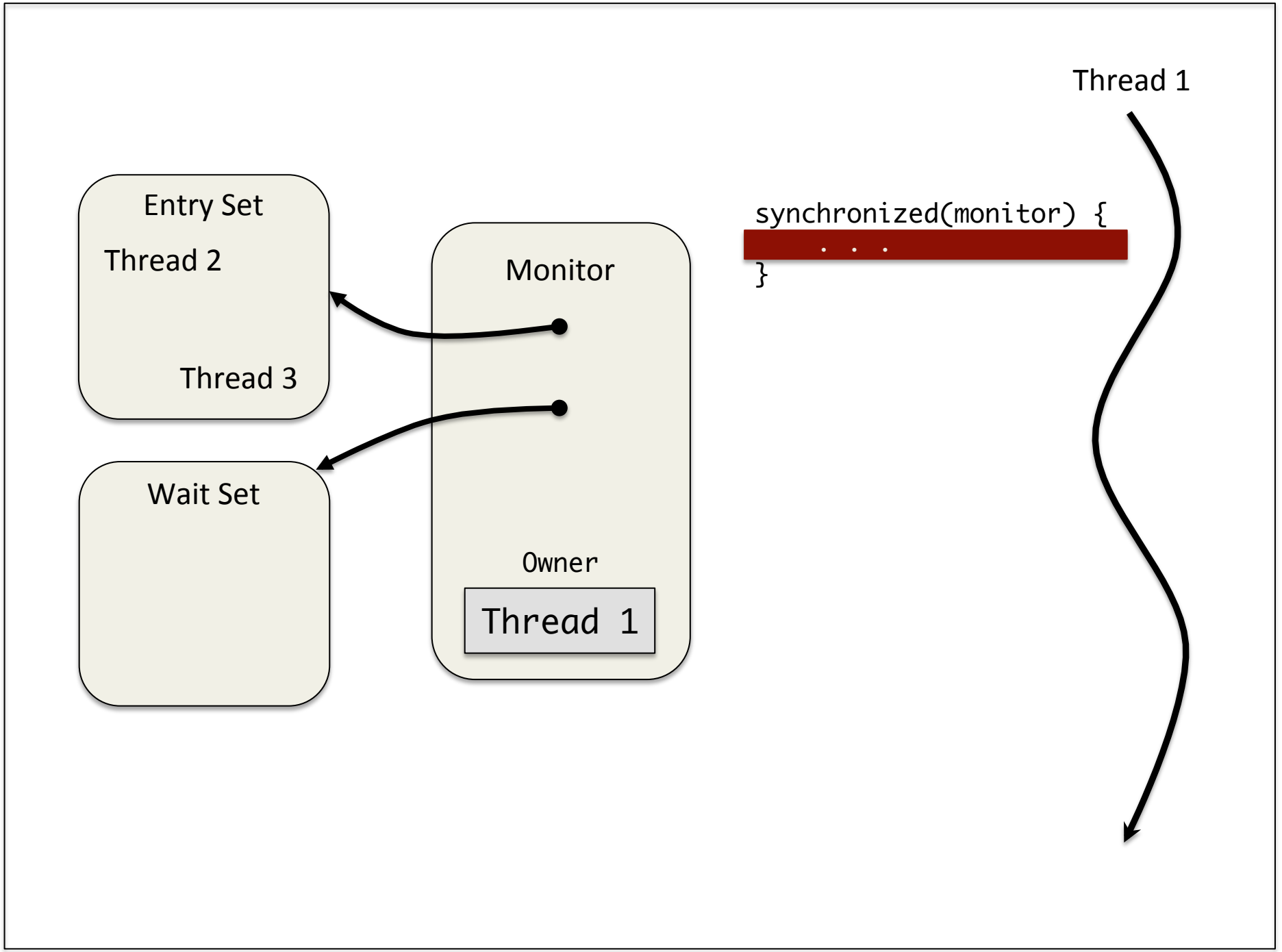

Thread 1
Entry Set
Monitor
synchronized(monitor) {
. . .
}
Wait Set
Owner
Thread 1
Thread 2

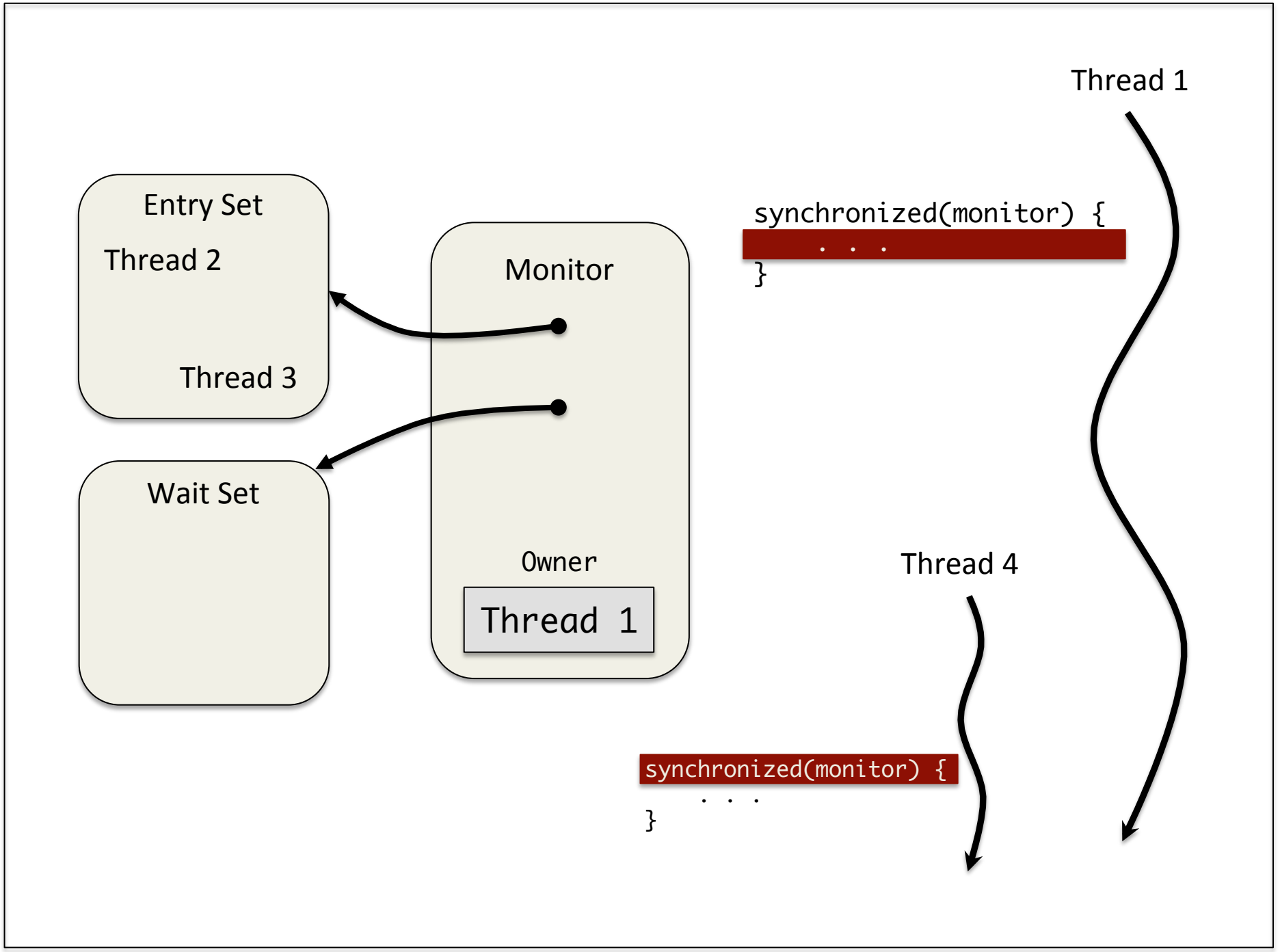

Thread 1
Entry Set
Monitor
synchronized(monitor) {
. . .
}
Wait Set
Owner
Thread 1
Thread 2
synchronized(monitor) {
. . .
}

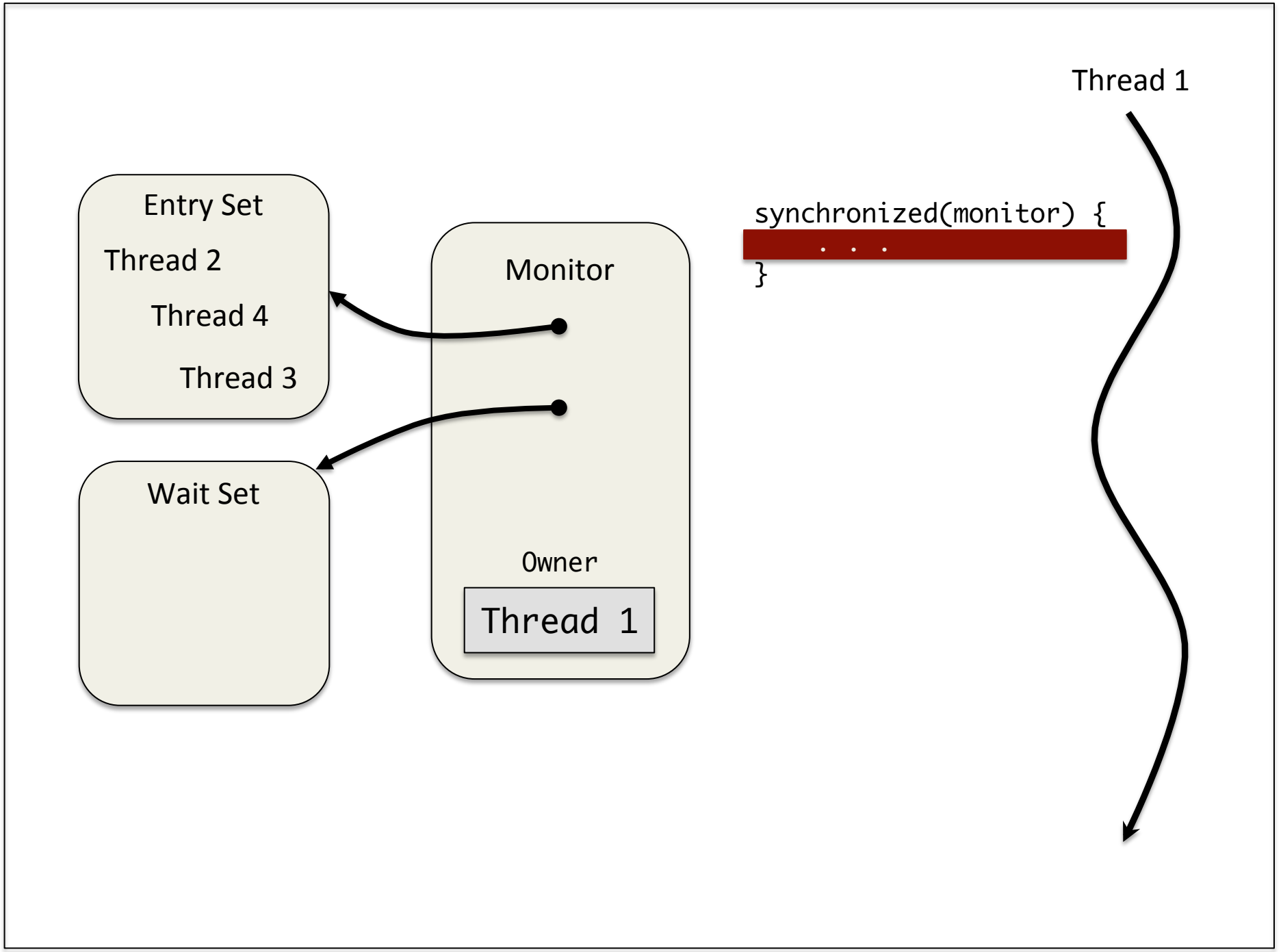

Thread 1
Entry Set
Monitor
synchronized(monitor) {
. . .
}
Wait Set
Owner
Thread 1
Thread 2
synchronized(monitor) {
. . .
}

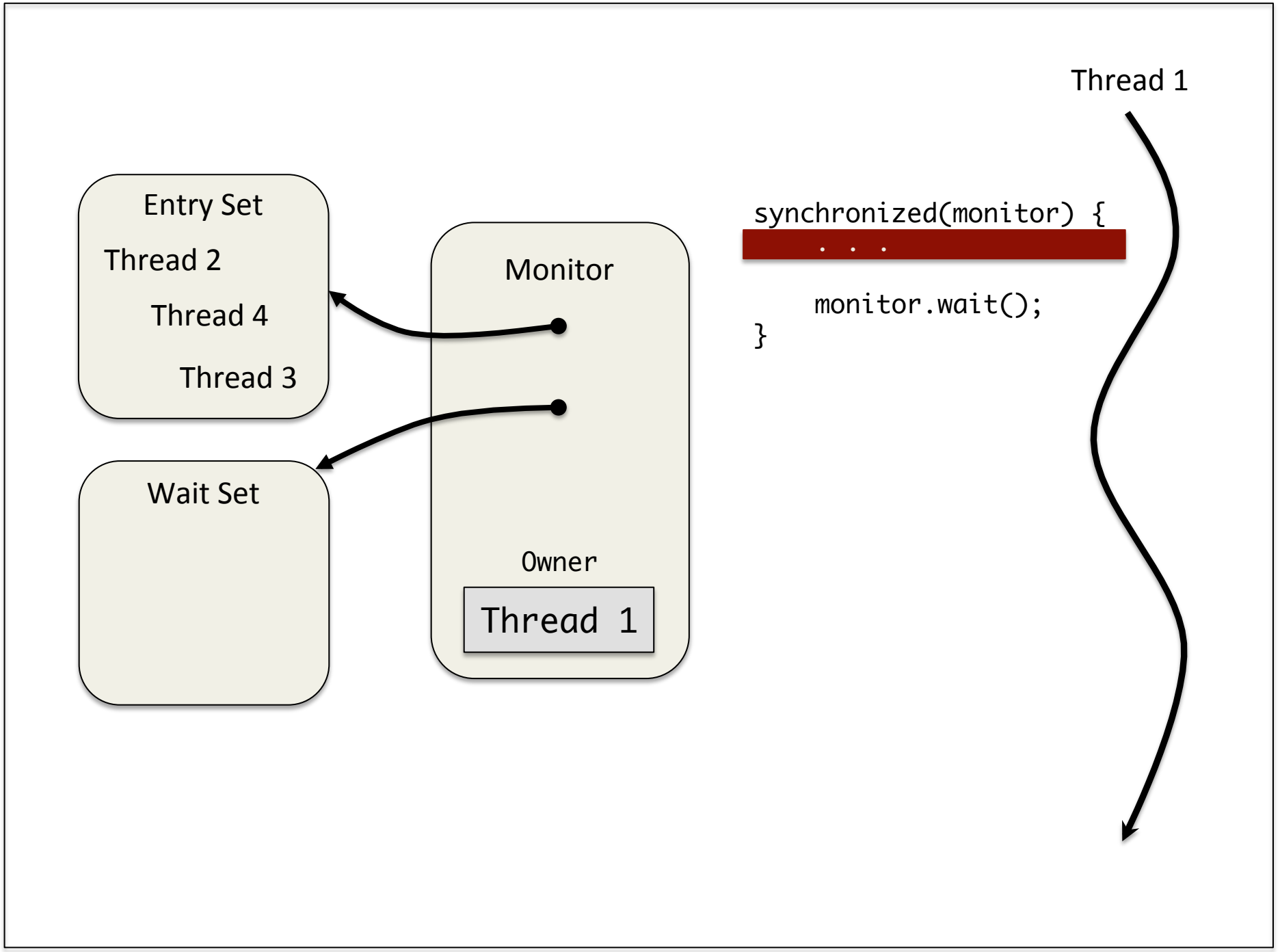

Thread 1
Entry Set
Thread 2
Monitor
synchronized(monitor) {
. . .
}
Wait Set
Owner
Thread 1

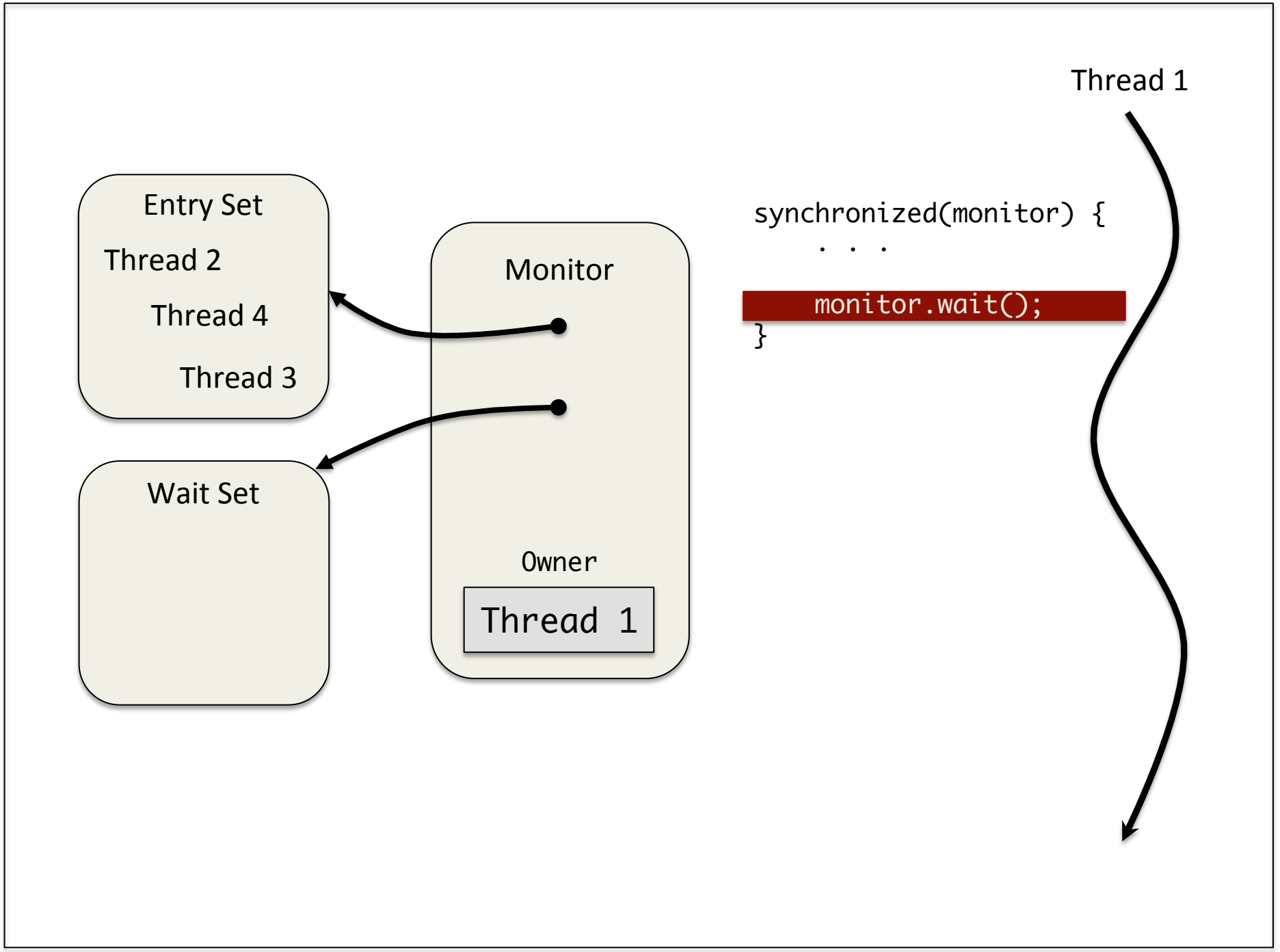

Thread 1
Entry Set
Thread 2
Monitor
synchronized(monitor) {
. . .
}
Wait Set
Owner
Thread 1
Thread 3
synchronized(monitor) {
. . .
}

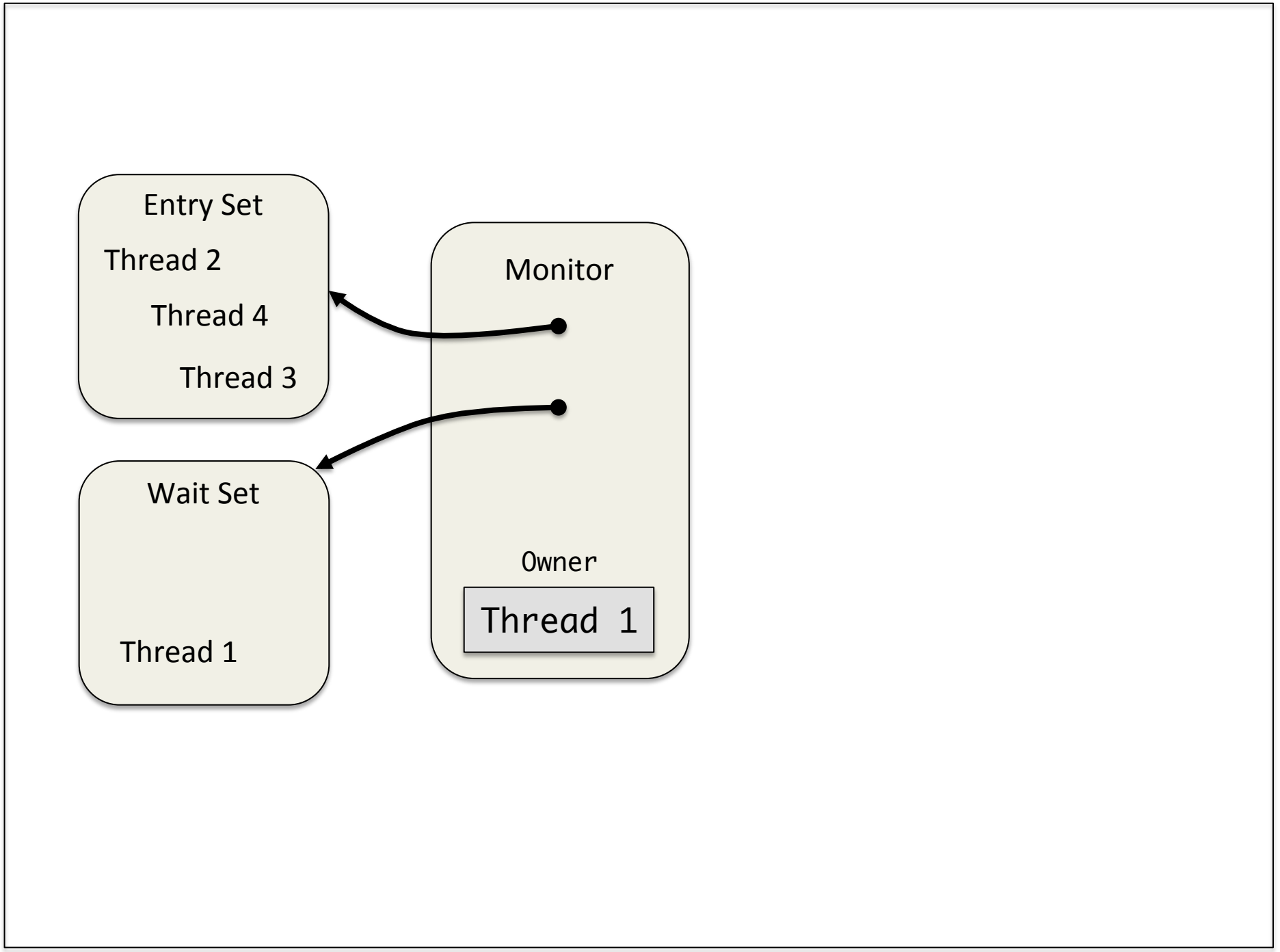

Thread 1
Entry Set
Thread 2
Thread 3
Monitor
synchronized(monitor) {
. . .
}
Wait Set
Owner
Thread 1

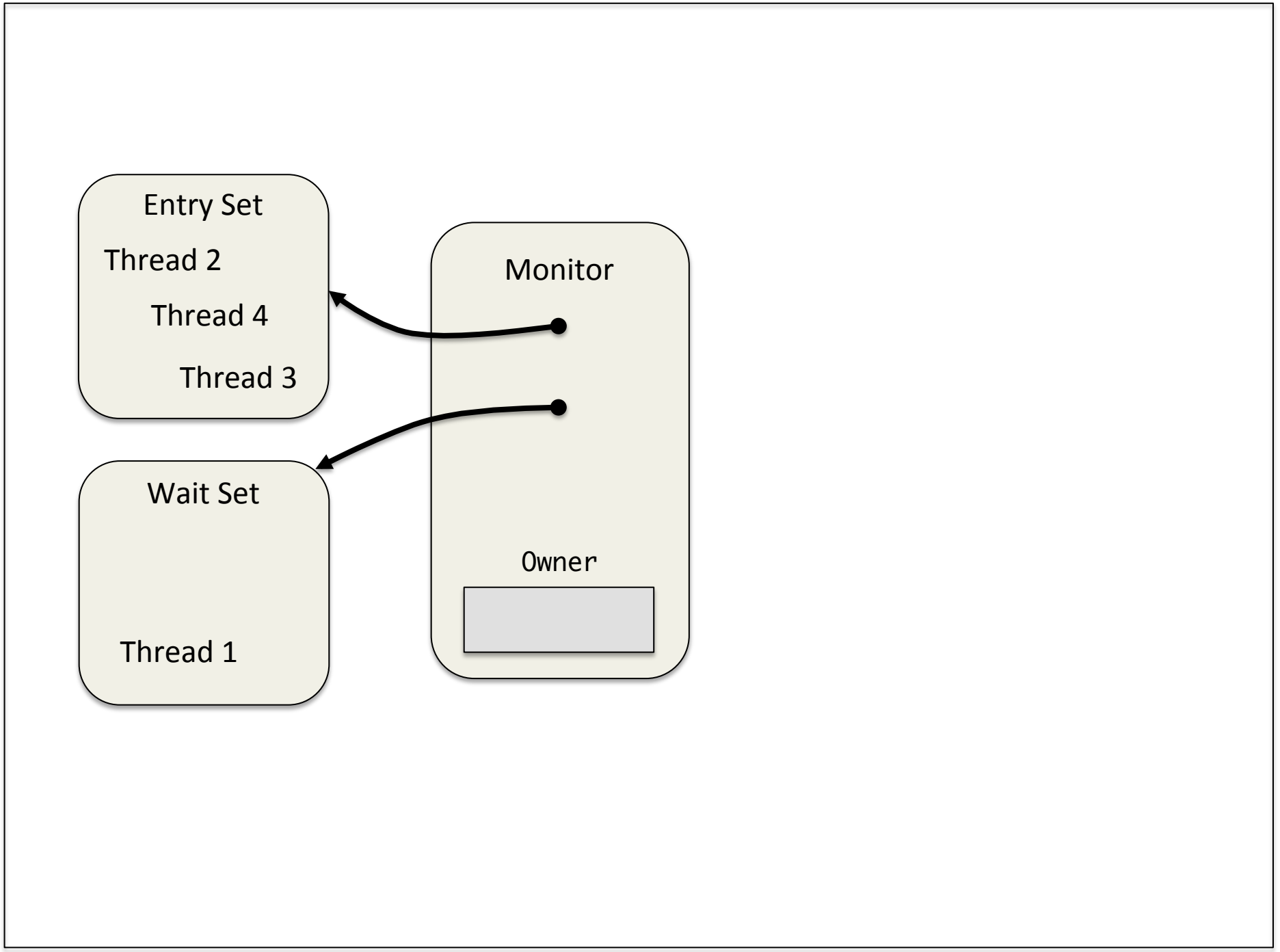

Thread 1
Entry Set
Thread 2
Thread 3
Monitor
synchronized(monitor) {
. . .
}
Wait Set
Owner
Thread 1
Thread 4
synchronized(monitor) {
. . .
}

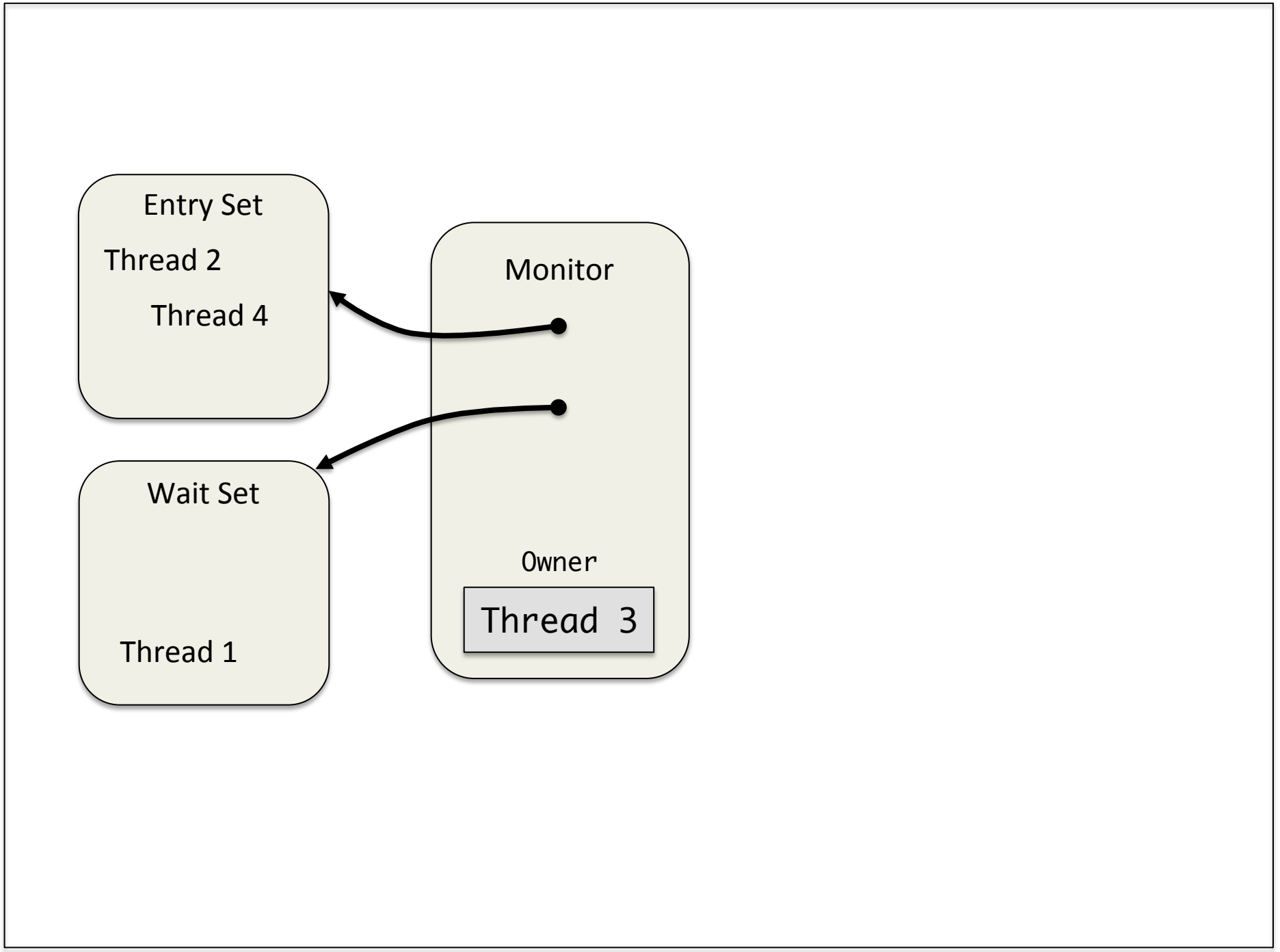

Thread 1
Entry Set
Thread 2
Thread 4
Thread 3
Wait Set
Monitor
Owner
Thread 1
synchronized(monitor) {
. . .
}

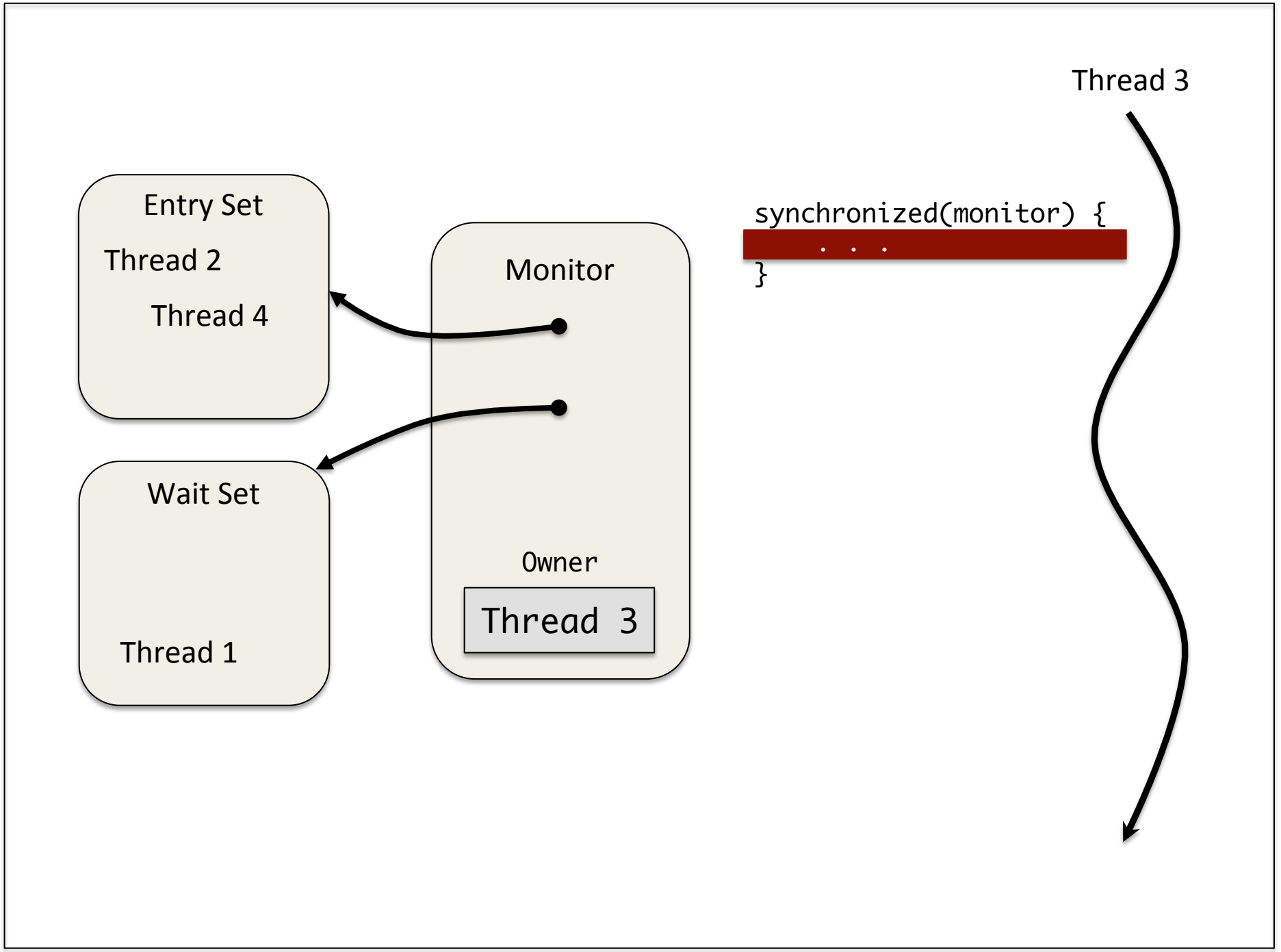

Thread 1
Entry Set
Thread 2
Thread 4
Thread 3
Monitor
synchronized(monitor) {
. . .
monitor.wait();
}
Wait Set
Owner
Thread 1

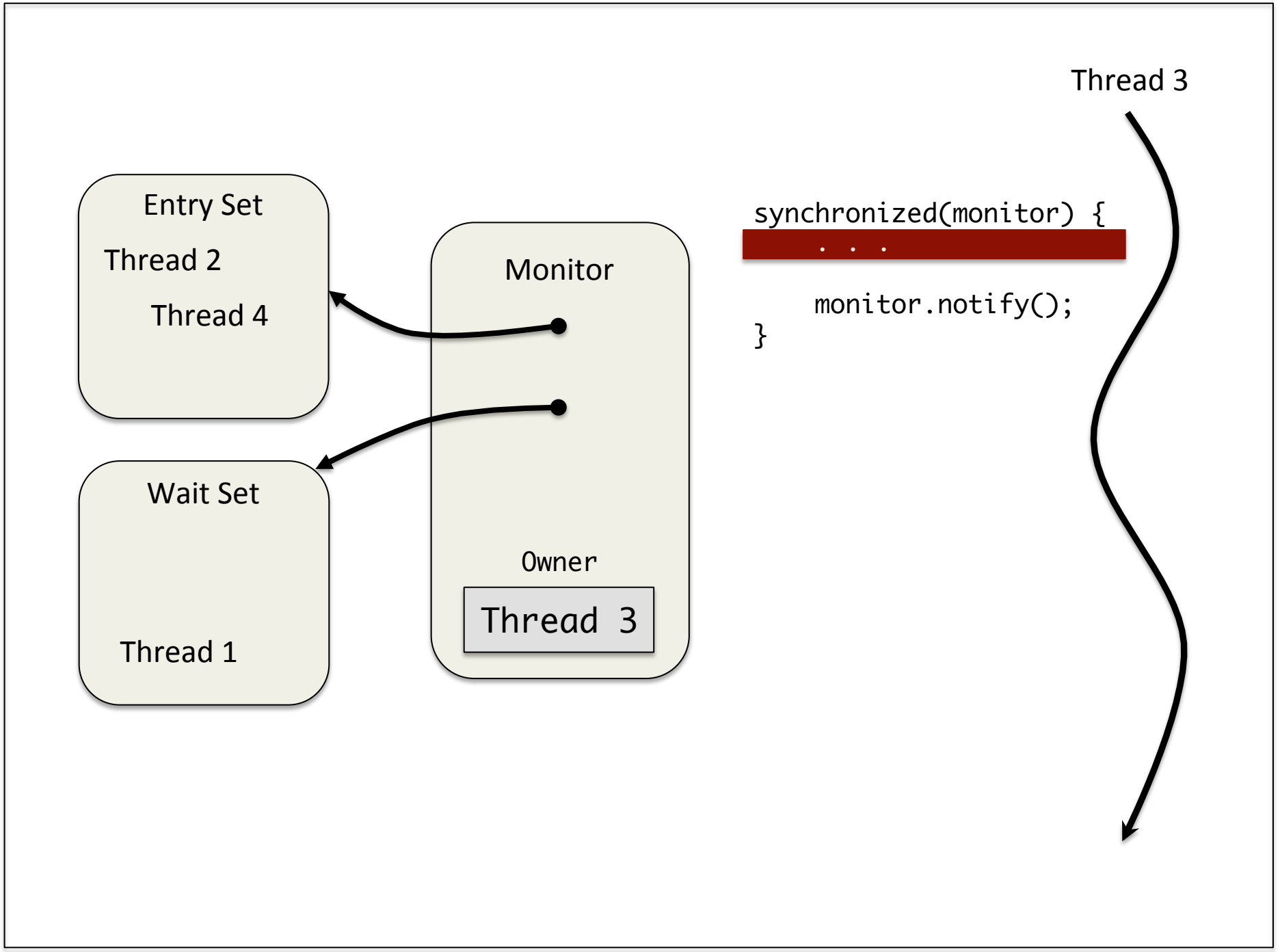

Thread 1
Entry Set
Thread 2
Thread 4
Thread 3
Monitor
synchronized(monitor) {
. . .
monitor.wait();
}
Wait Set
Owner
Thread 1

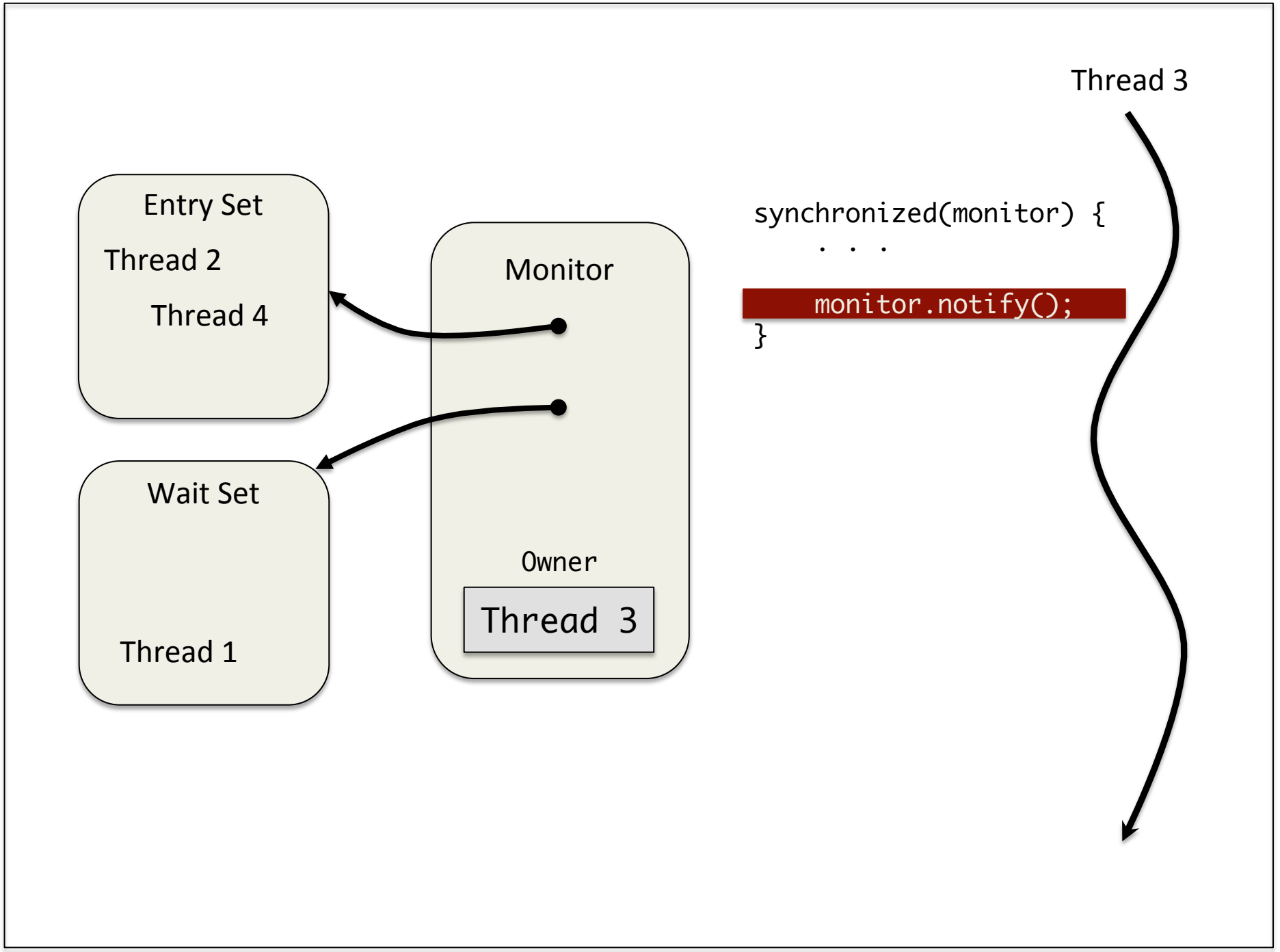

Entry Set
Thread 2
Thread 4
Thread 3
Monitor
Wait Set
Thread 1
Owner
Thread 1

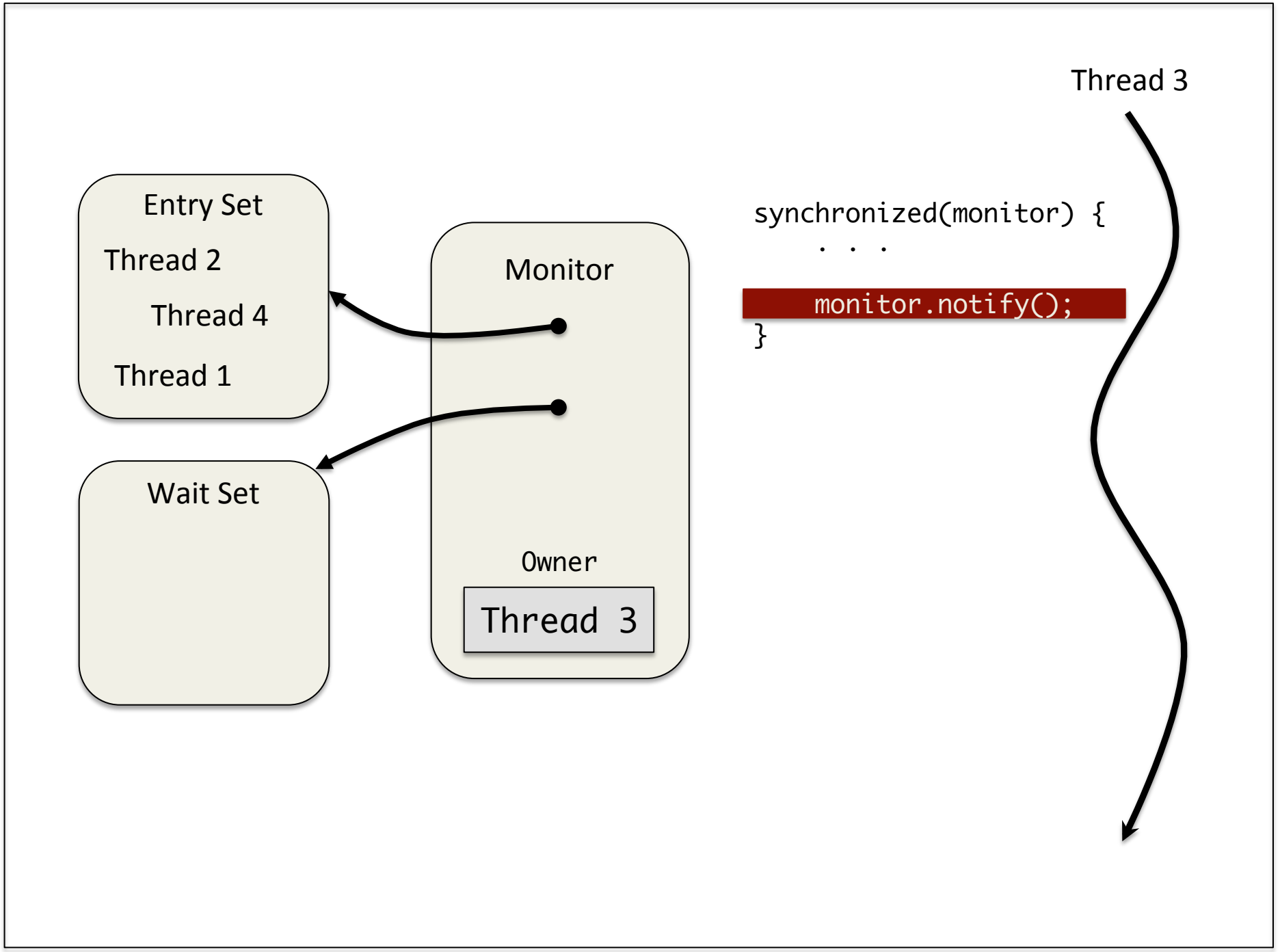

Entry Set
Thread 2
Thread 4
Thread 3
Monitor
Wait Set
Thread 1
Owner

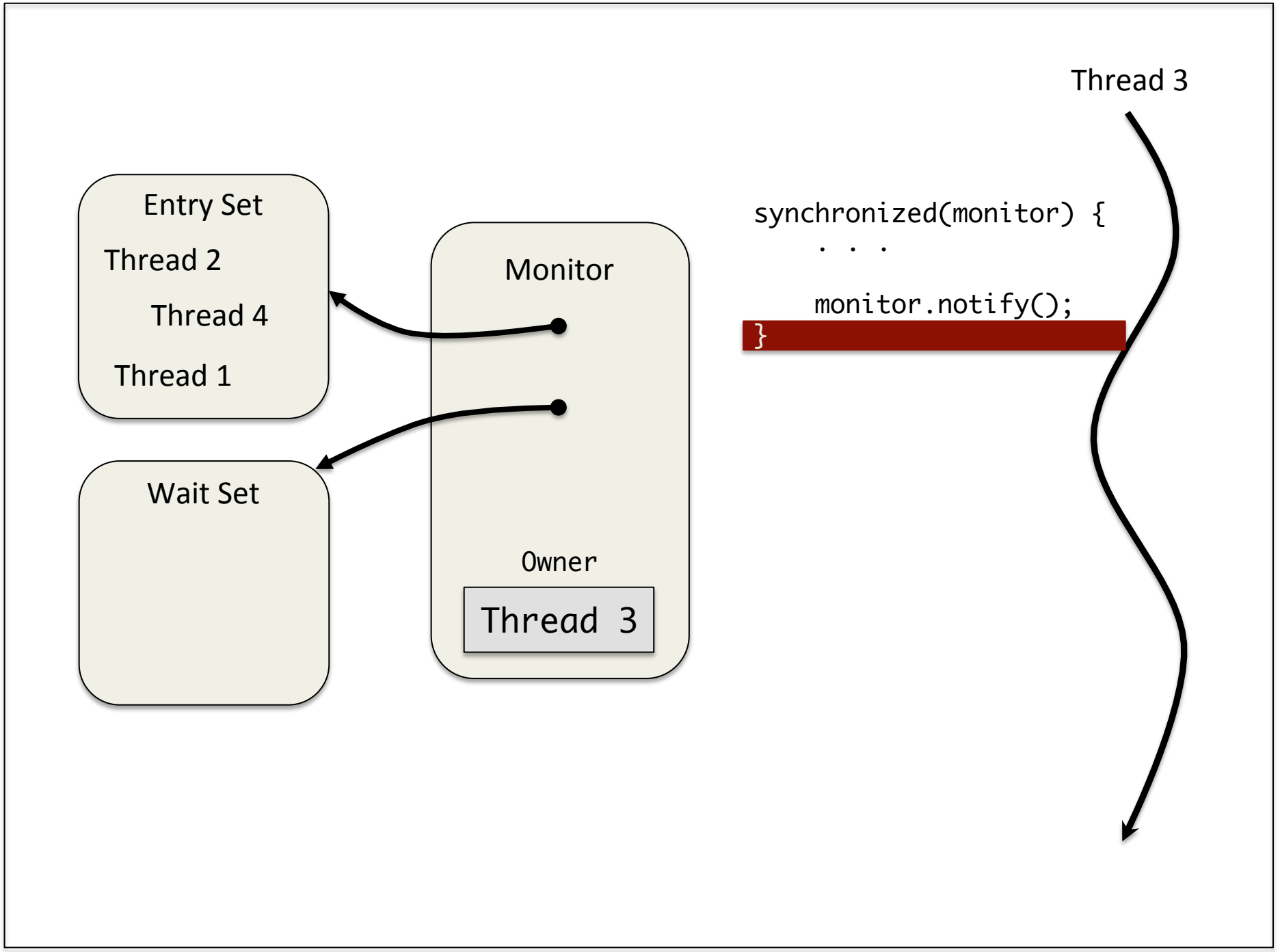

Entry Set
Thread 2
Thread 4
Monitor
Wait Set
Thread 1
Owner
Thread 3

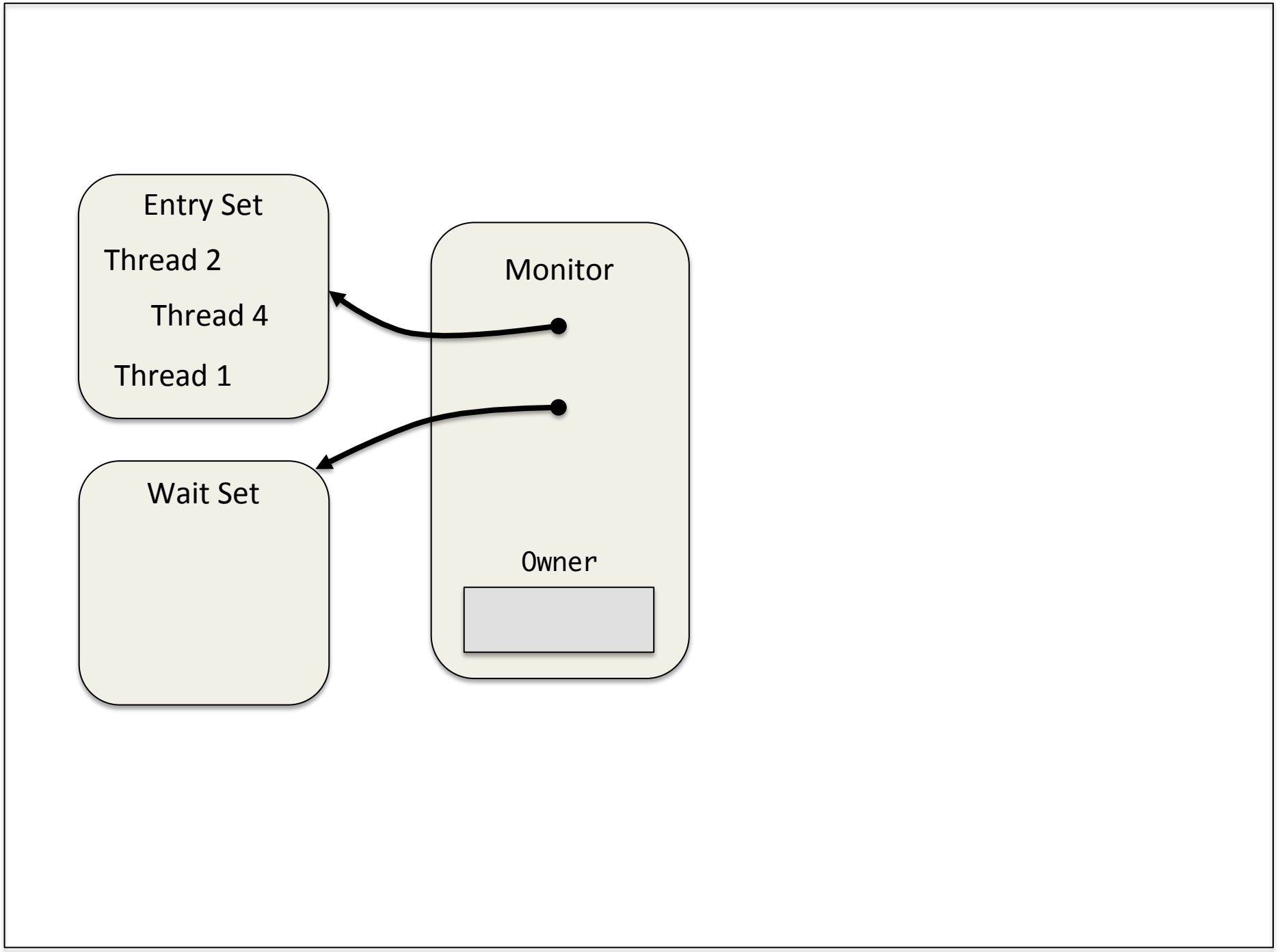

Thread 3
Entry Set
Thread 2
Thread 4
Monitor
synchronized(monitor) {
. . .
}
Wait Set
Owner
Thread 3
Thread 1

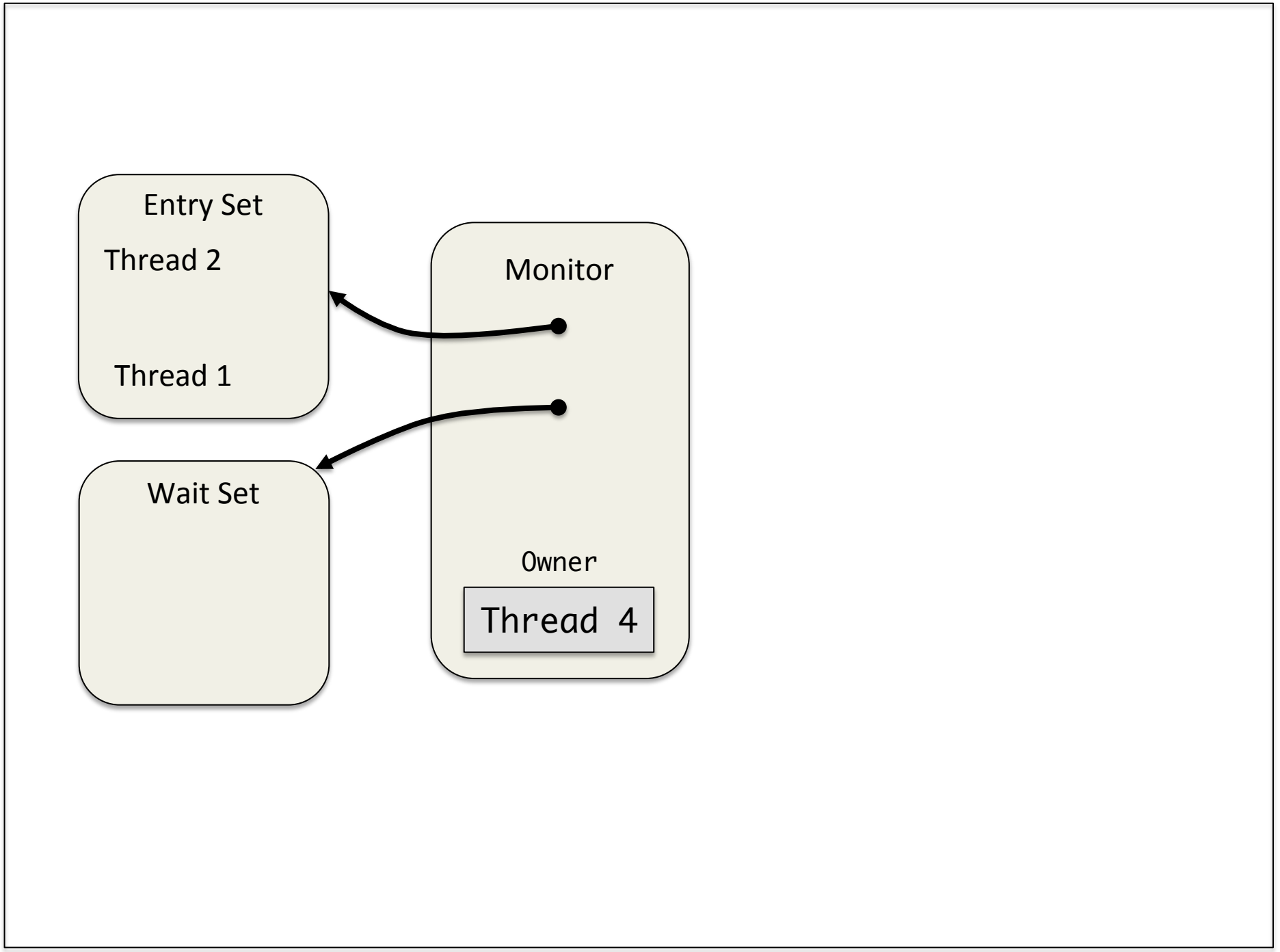

Thread 3
Entry Set
Thread 2
Thread 4
Monitor
synchronized(monitor) {
. . .
monitor.notify();
}
Wait Set
Thread 1
Owner
Thread 3

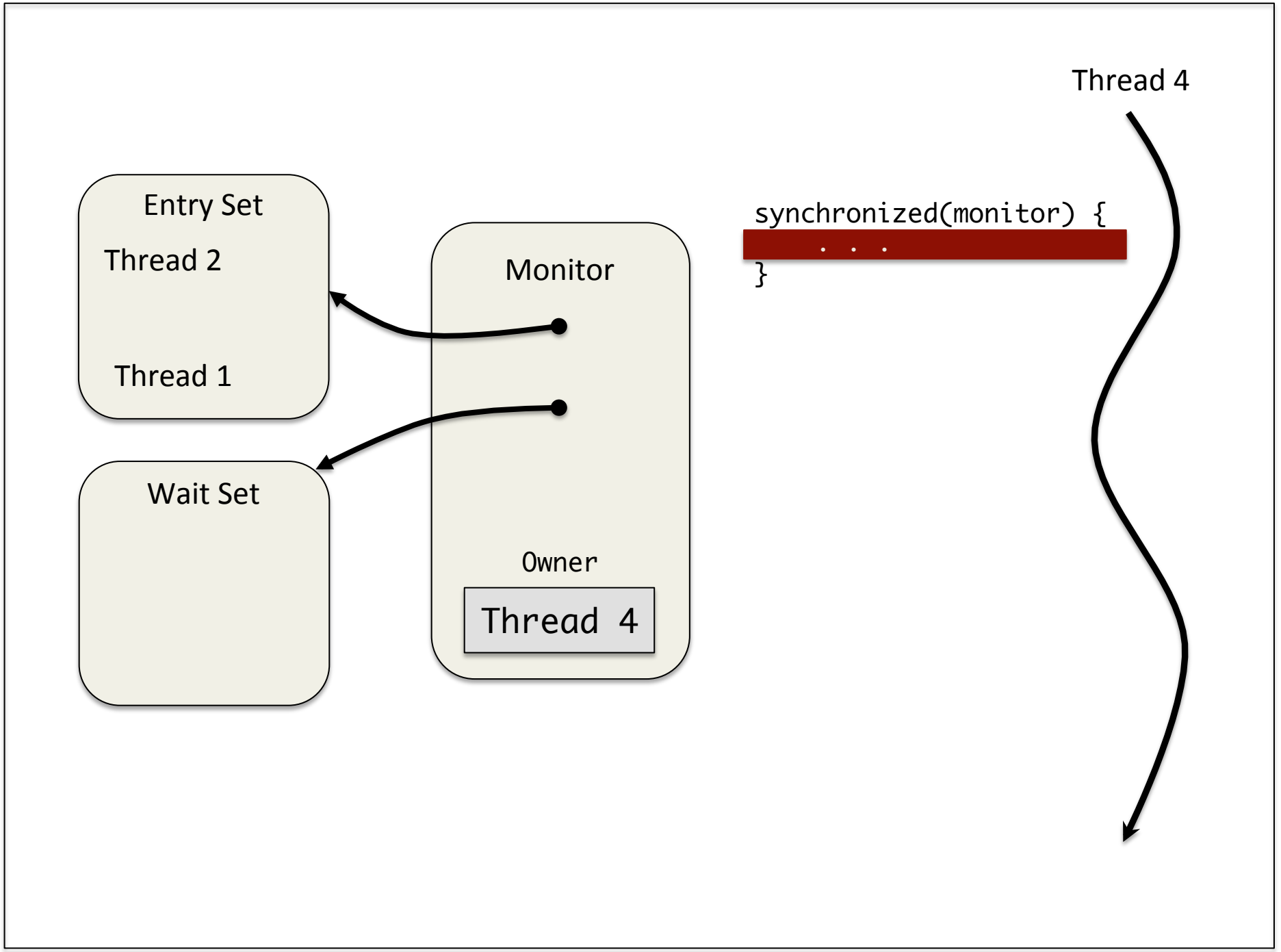

Thread 3
Entry Set
Thread 2
Thread 4
Monitor
synchronized(monitor) {
. . .
monitor.notify();
}
Wait Set
Thread 1
Owner
Thread 3


Thread 3
Entry Set
Thread 2
Thread 4
Thread 1
Monitor
synchronized(monitor) {
. . .
monitor.notify();
}
Wait Set
Owner
Thread 3


Thread 3
Entry Set
Thread 2
Thread 4
Thread 1
Monitor
synchronized(monitor) {
. . .
monitor.notify();
}
Wait Set
Owner
Thread 3


Entry Set
Thread 2
Thread 4
Thread 1
Monitor
Wait Set
Owner


Entry Set
Thread 2
Thread 1
Monitor
Wait Set
Owner
Thread 4


Thread 4
Entry Set
Thread 2
Thread 1
Monitor
synchronized(monitor) {
. . .
}
Wait Set
Owner
Thread 4

Wait Set Implementation

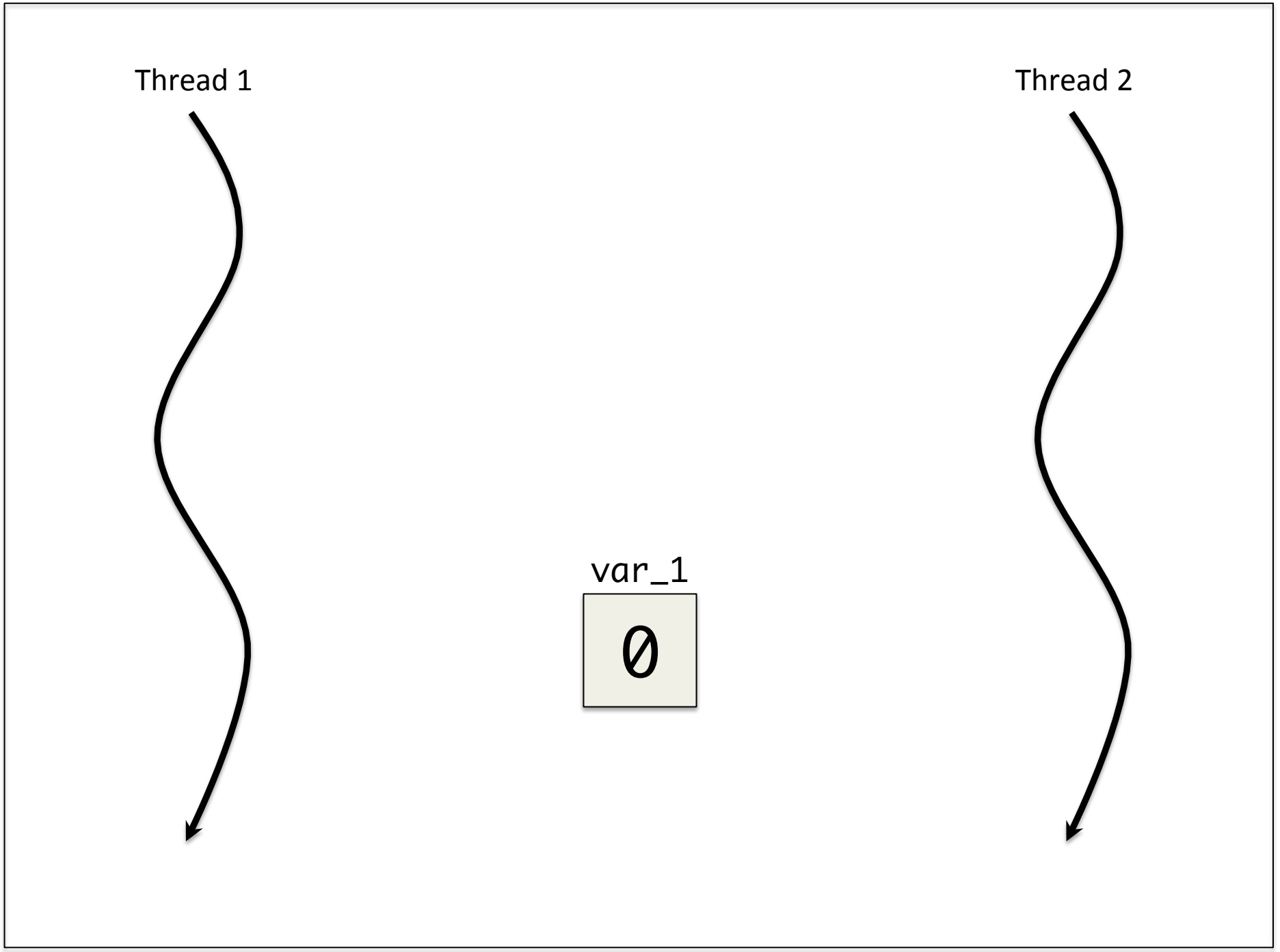
- Wait set management largely at the Java level
  - `wait`, `notify` and `notifyAll` methods on `Object`
- Methods implemented as intrinsics
  - Implementation has knowledge of VM structures
- We don't create a wait set for all objects
  - Wait sets linked to thickened locks
  - Monitors used for waiting likely to be contended

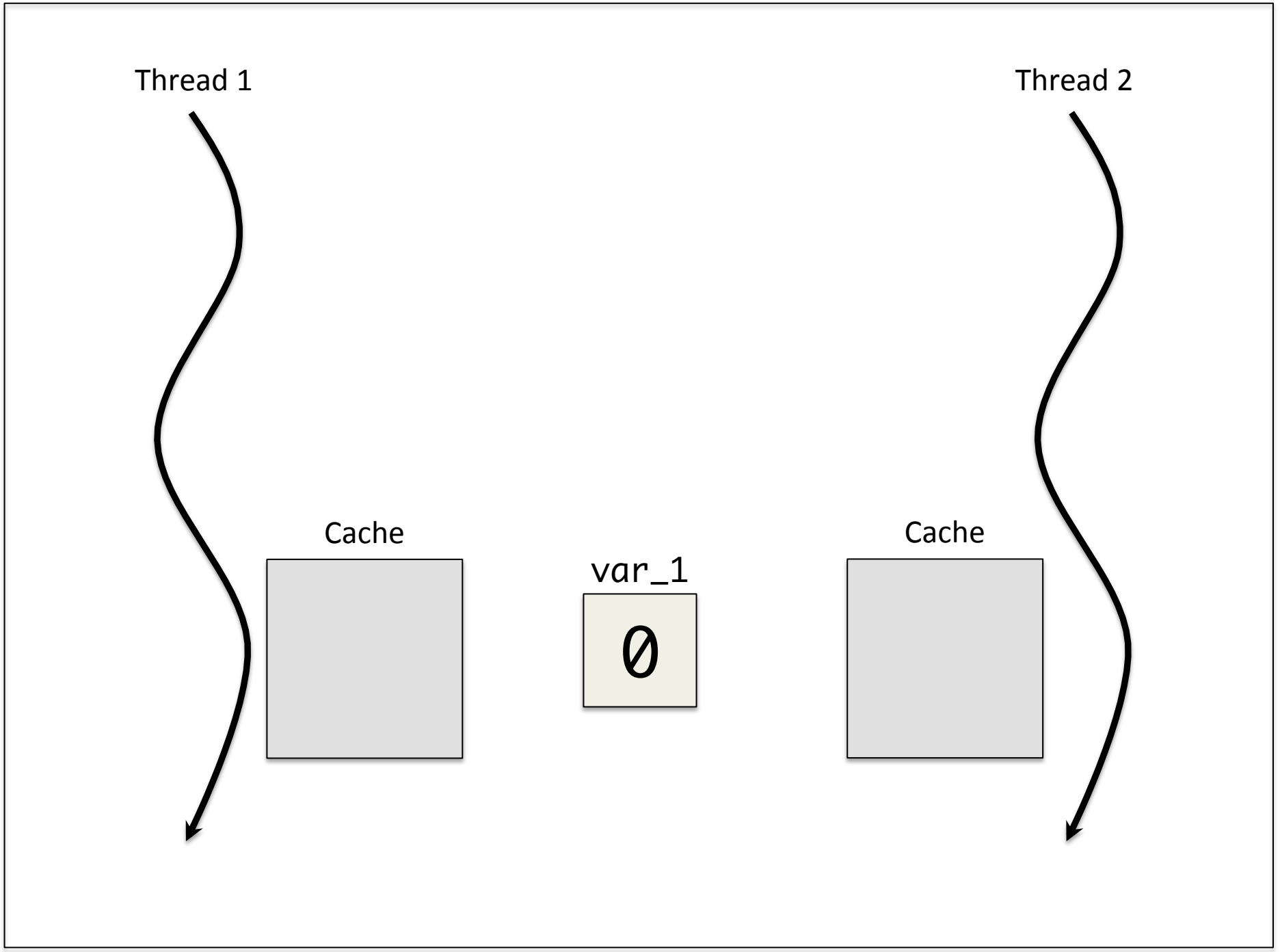
Memory Consistency

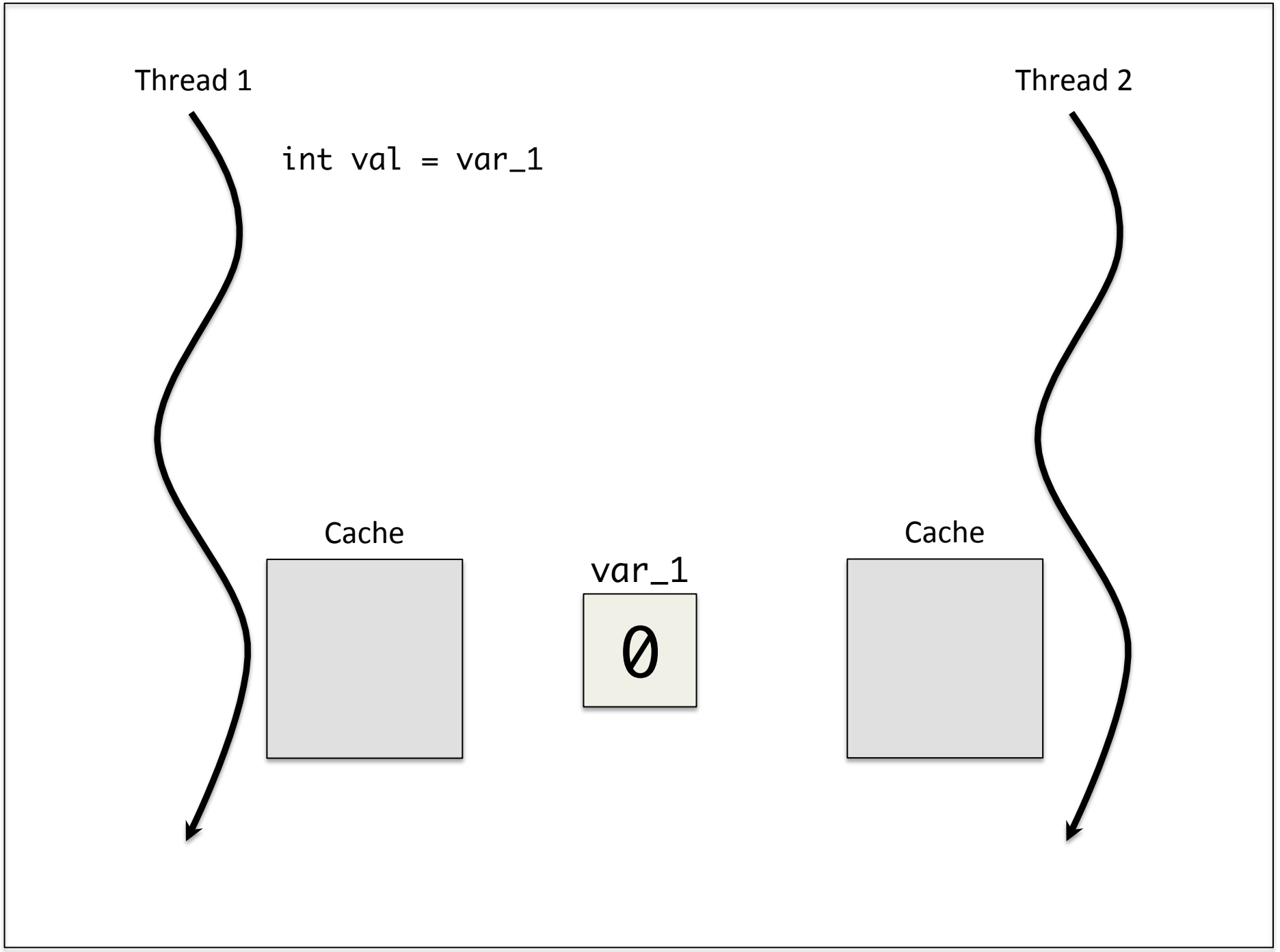
- Consistency refers to threads views of data
  - Sequential means that all threads see the same values
  - Relaxed if values can differ at times
- Behavior set by the memory model
  - All computer systems have a memory model
  - Some are better defined than others
- Sequential consistency easiest to understand
  - A global view of data
  - Simple extension of single-threaded semantics

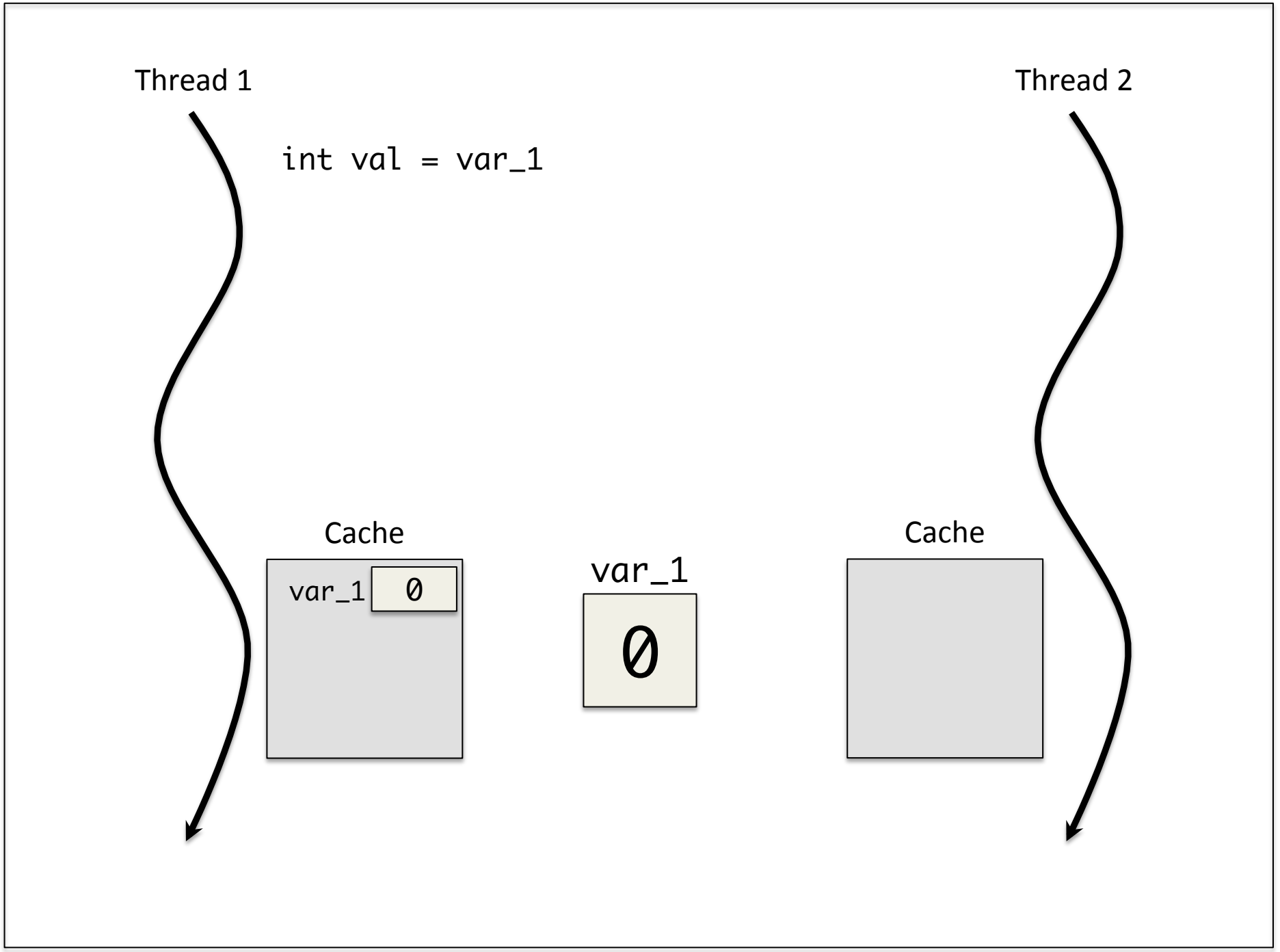
Relaxed Consistency

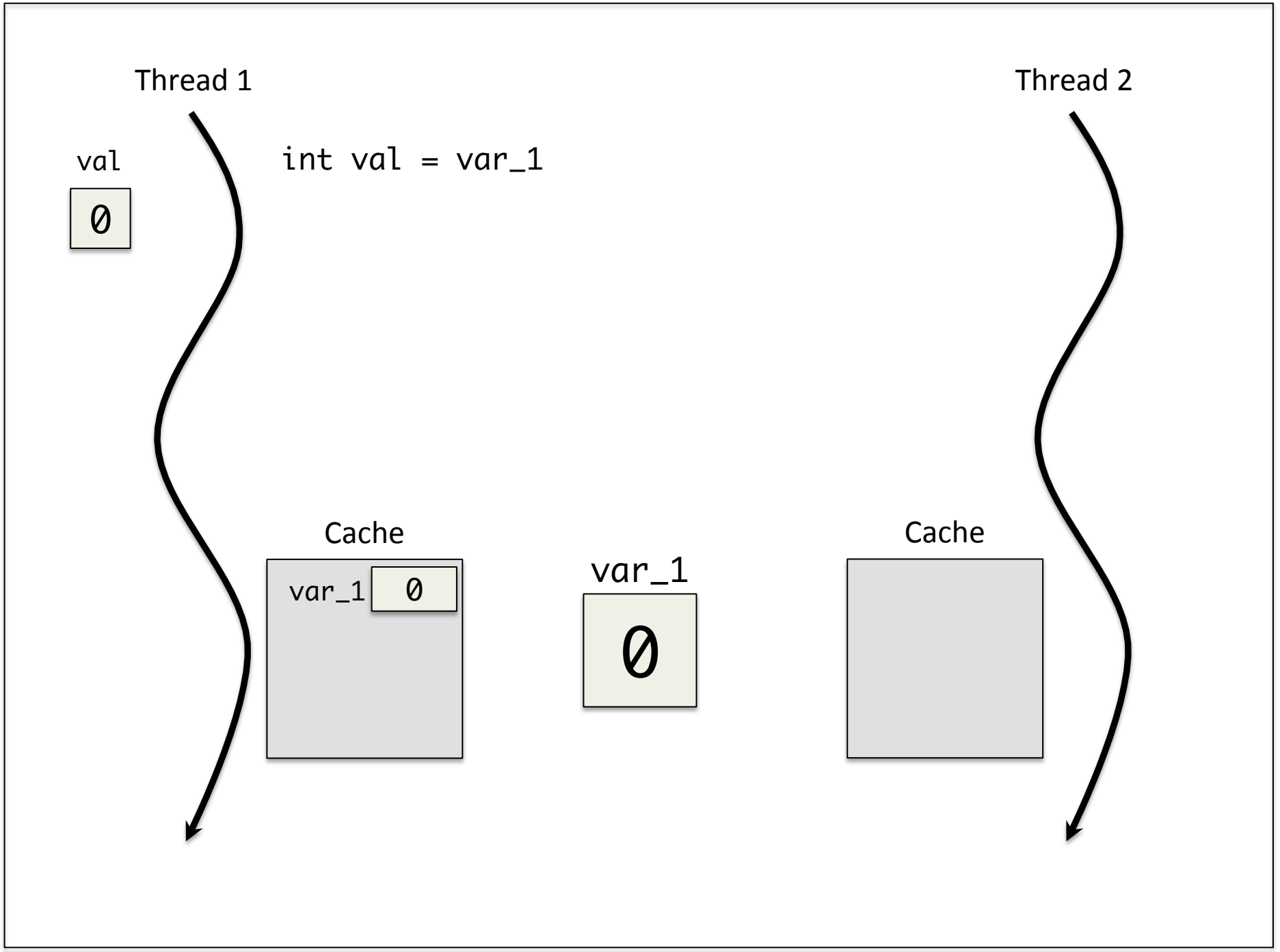
- Memory accesses are a major overhead
  - We've seen how the memory hierarchy helps
- Caches bring data close to the processor
  - Separate L1, L2 and possibly L3 cache per core
- Things get worse with multi-processor systems
  - Not to mention multi-machine distributed systems
- Sequential consistency is unfeasible

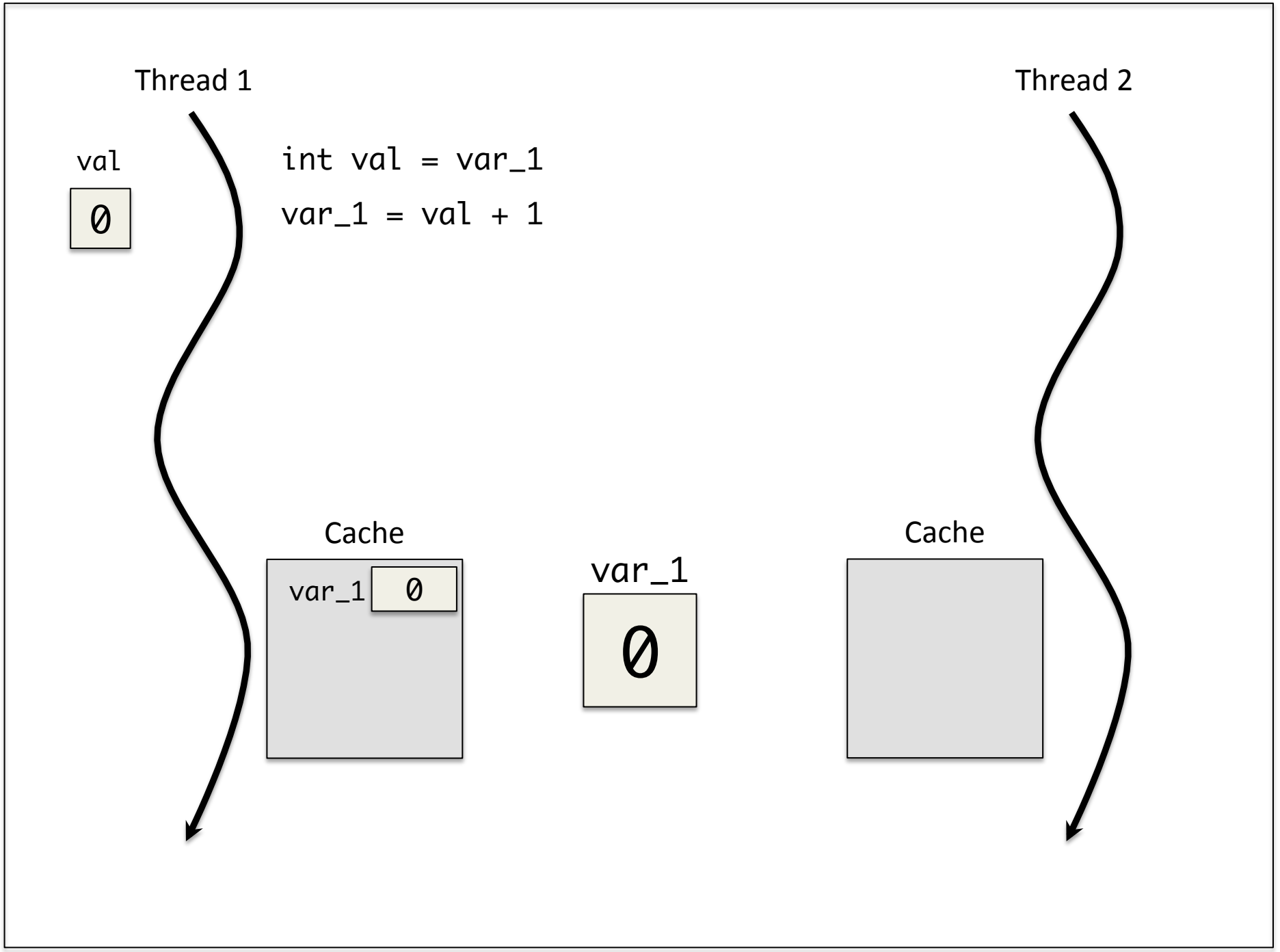

Thread 1
Thread 2
var_1
0

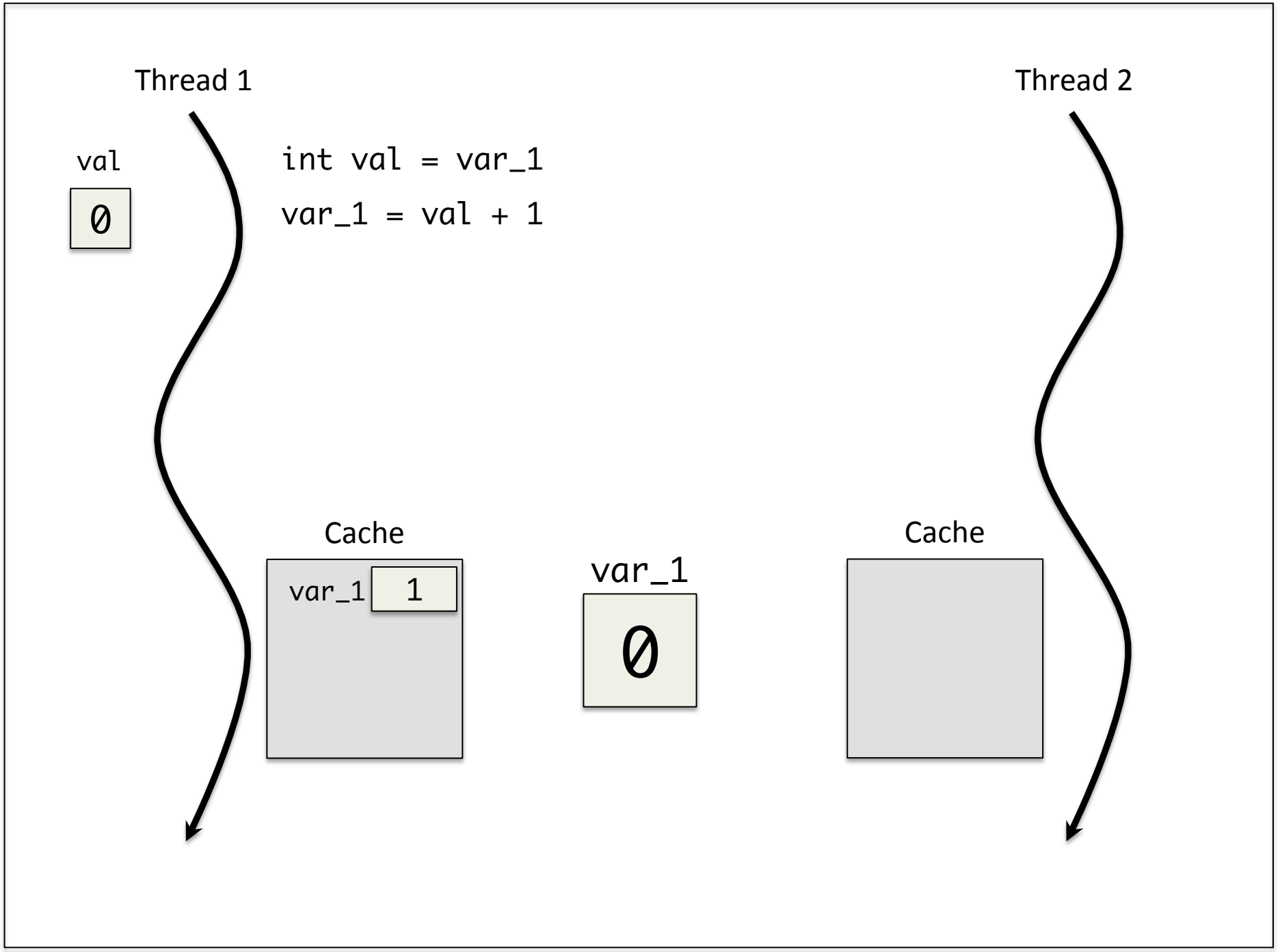

Thread 1
Thread 2
Cache
var_1
0
Cache

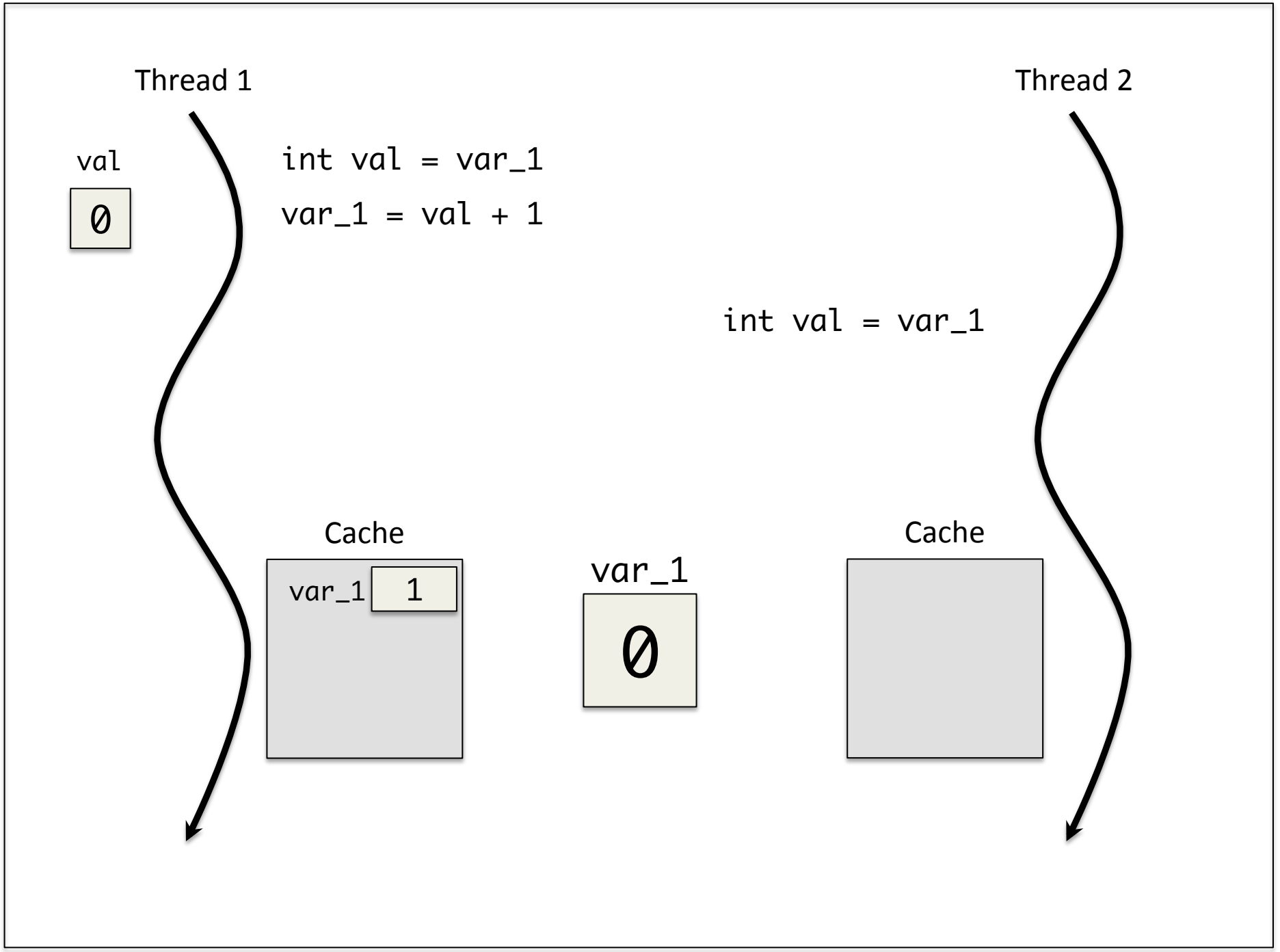

Thread 1
Thread 2
int val = var_1
Cache
var_1
0
Cache

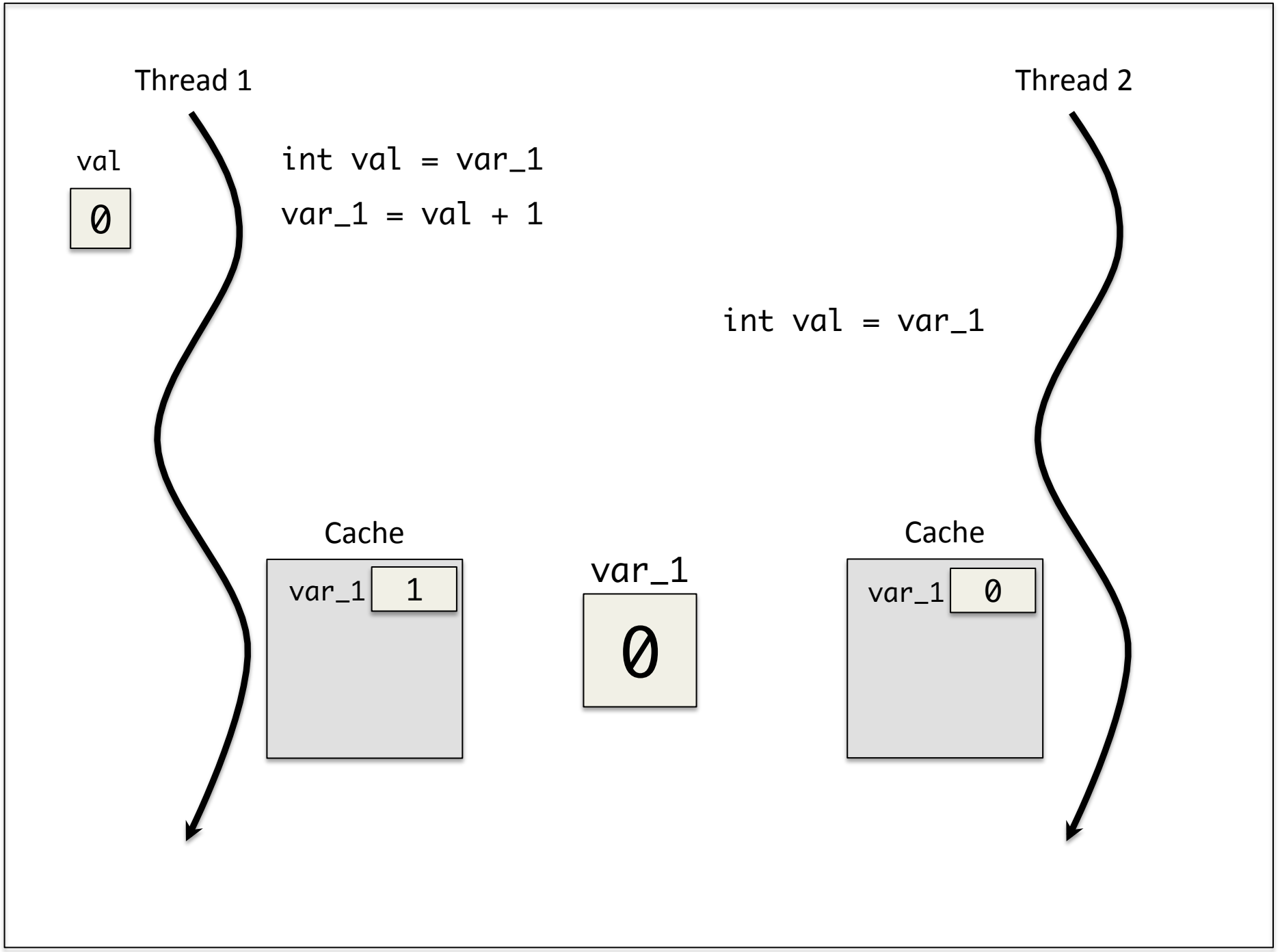

Thread 1
Thread 2
int val = var_1
Cache
var_1
0
var_1
0
Cache

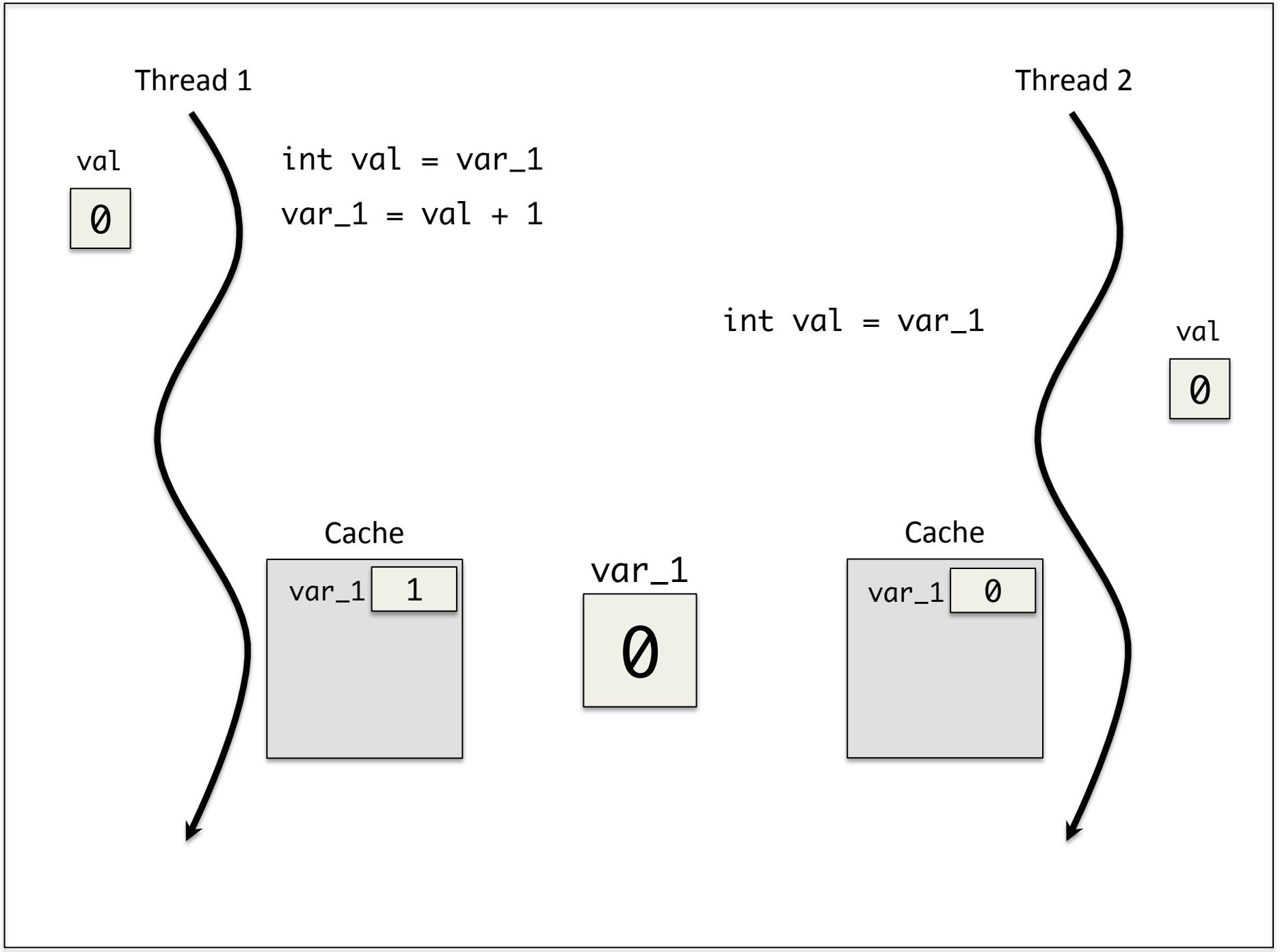

Thread 1
Thread 2
val
0
int val = var_1
Cache
var_1
0
var_1
0
Cache

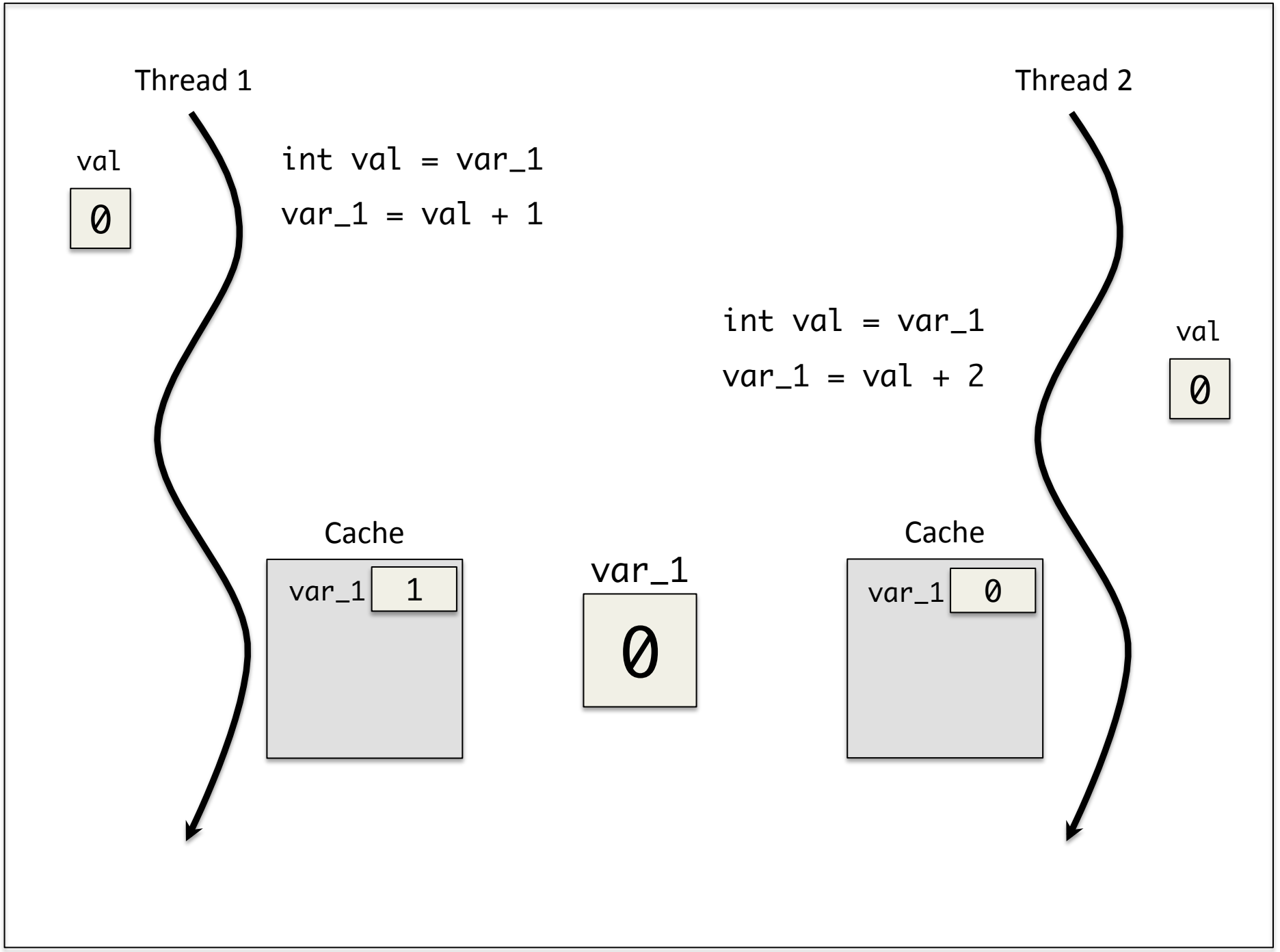

Thread 1
Thread 2
val
0
int val = var_1
var_1 = val + 1
Cache
var_1
0
var_1
0
Cache

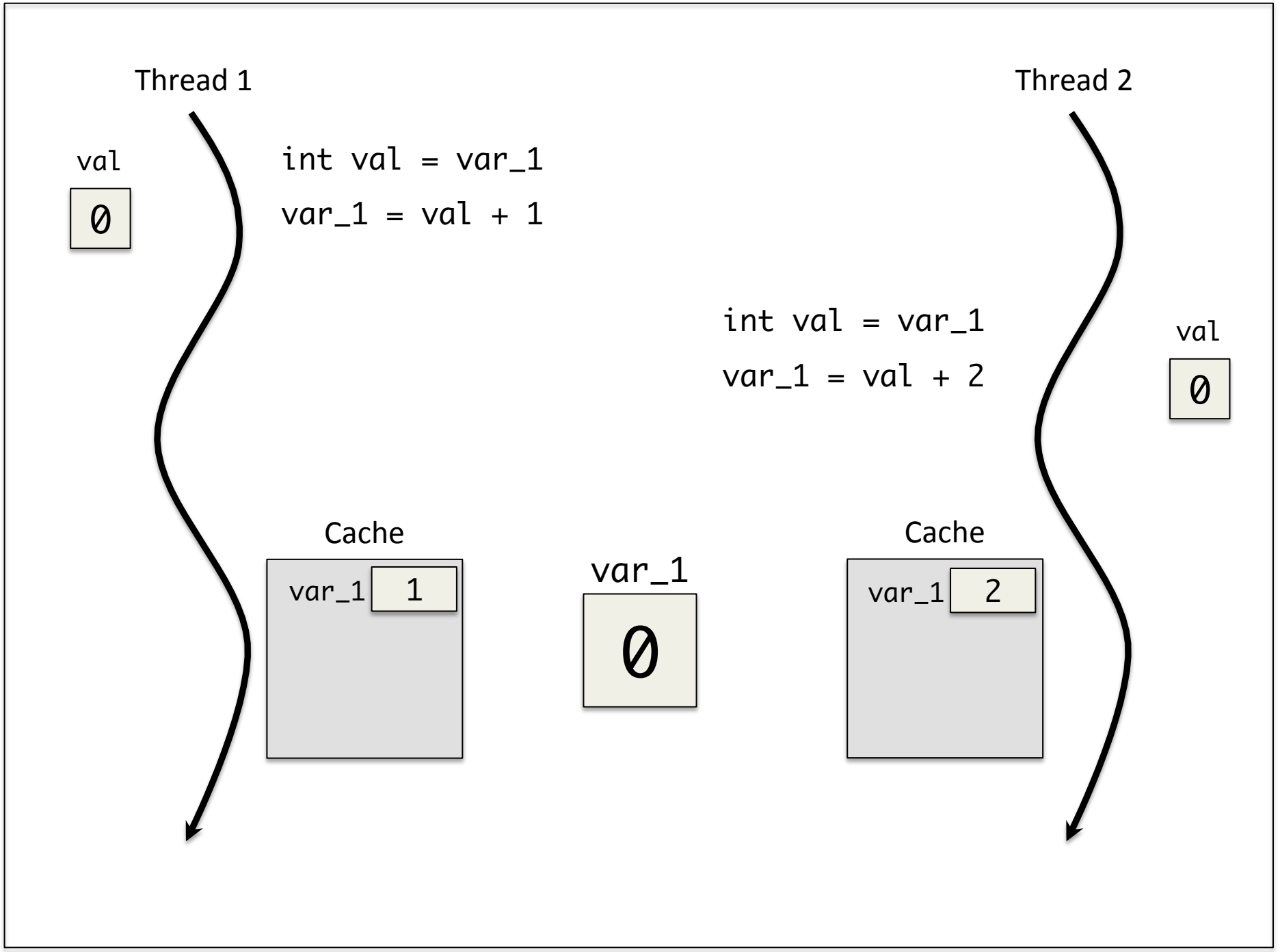

Thread 1
Thread 2
val
0
int val = var_1
var_1 = val + 1
Cache
var_1
1
var_1
0
Cache

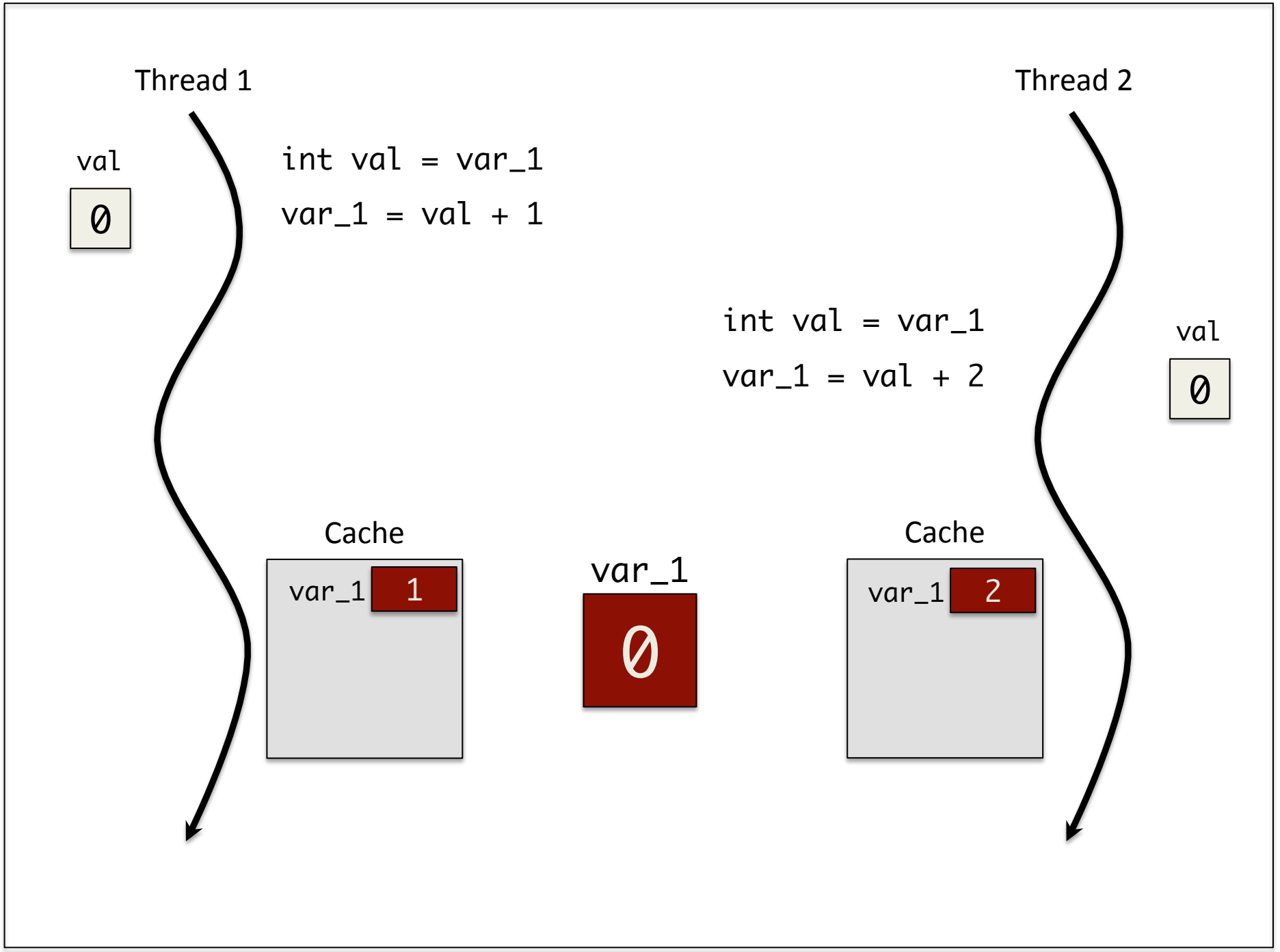

Thread 1
Thread 2
val
0
int val = var_1
var_1 = val + 1
int val = var_1
Cache
var_1
1
var_1
0
Cache


Thread 1
Thread 2
val
0
int val = var_1
var_1 = val + 1
int val = var_1
Cache
var_1
1
var_1
0
Cache
var_1
0


Thread 1
val
0
int val = var_1
var_1 = val + 1
Cache
var_1
1
var_1
0
Thread 2
int val = var_1
val
0
Cache
var_1
0


Thread 1
val
0
int val = var_1
var_1 = val + 1
Cache
var_1
1
var_1
0
Thread 2
int val = var_1
var_1 = val + 2
val
0
Cache
var_1
0


Thread 1
val
0
int val = var_1
var_1 = val + 1
Cache
var_1
1
var_1
0
Thread 2
int val = var_1
var_1 = val + 2
val
0
Cache
var_1
2


Thread 1
val
0
int val = var_1
var_1 = val + 1
Thread 2
int val = var_1
var_1 = val + 2
val
0
Cache
var_1
1
var_1
0
Cache
var_1
2

Race Condition

- The previous example showed a data race
  - Write by one thread
  - Read by another thread
  - No explicit synchronization in between
- Memory model determines program behavior
  - Guarantees semantics in a data race
  - What ordering of operations are allowable
  - When we can assume that memory is consistent

Memory Models

- Defining a memory model is tricky
- Developers need code to be reasonably consistent
  - Want to be able to reason about program behavior
- Too much synchronization hurts performance
  - Leads to inter-processor communication
  - Rules out many common optimizations
- Key is to be predictable

The Java Memory Model

- Originally defined by the Java Language Spec
  - Some flaws were discovered over time
  - Revised by JSR 133 in Java 1.5
- First attempt to make a cross-platform model
  - Has to work regardless of architecture
  - Many other models have followed since
- Correctly synchronized code should be simple
  - Poorly synchronized code should be predictable

Java Memory Model Guarantees

- The Java Memory Model makes five guarantees

Java Memory Model Guarantees

- The Java Memory Model makes five guarantees
  - Each action in a single thread happens before the next

Java Memory Model Guarantees

- The Java Memory Model makes five guarantees
  - Each action in a single thread happens before the next
  - Unlock happens before subsequent lock on an object

Java Memory Model Guarantees

- The Java Memory Model makes five guarantees
  - Each action in a single thread happens before the next
  - Unlock happens before subsequent lock on an object
  - Write to a volatile field happens before read of that field

Java Memory Model Guarantees

- The Java Memory Model makes five guarantees
  - Each action in a single thread happens before the next
  - Unlock happens before subsequent lock on an object
  - Write to a volatile field happens before read of that field
  - Call to `Thread.start` happens before actions in that thread

Java Memory Model Guarantees

- The Java Memory Model makes five guarantees
  - Each action in a single thread happens before the next
  - Unlock happens before subsequent lock on an object
  - Write to a volatile field happens before read of that field
  - Call to `Thread.start` happens before actions in that thread
  - Thread actions happen before `join` returns

Volatile

- `volatile` variables are never cached locally
  - All writes conceptually go to main memory
  - All threads see the same value immediately
- Useful as lightweight synchronization
  - Fewer features than full monitors
  - May perform better than critical sections
- Some semantic traps for the unwary
  - Incrementing a variable in two threads really two actions

Memory Model and JIT Compiler

- Memory model has implications for JIT
  - Limits code reordering
  - Limits memory prefetching
- `final` fields treated specially
  - Initial write in constructor comes before object is shared
  - Could be reordered in inlined code
  - Afterwards, don't have to reload the final field

Memory Fences

- Processor-level synchronization instruction
  - One of the most expensive in the ISA
  - Most processors have finer-grained instructions
- Guarantees memory ordering before and after
  - Use fences to synchronize on monitor operations
- Flushes cached versions of variables