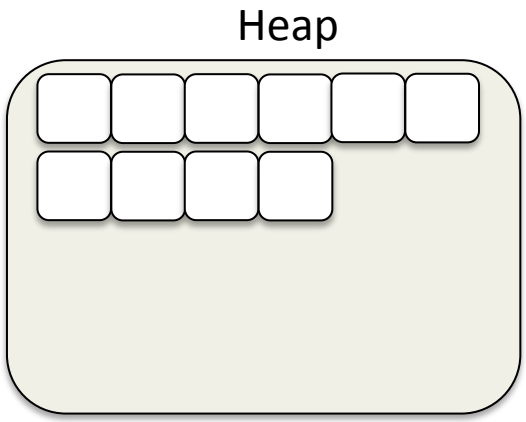
In the Last Week

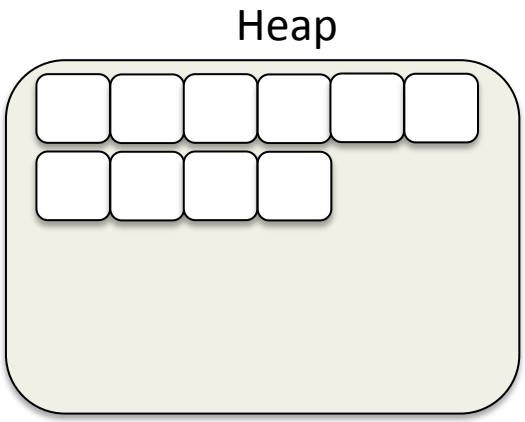
- Assignment 4 due yesterday
- Final project posted now
  - Due on December 7th
- Thanks for special topic and review suggestions
- GC debugging for fun and profit

```
createObject {

    ptr = allocate object

    call constructor(ptr)

    return ptr;
}
```

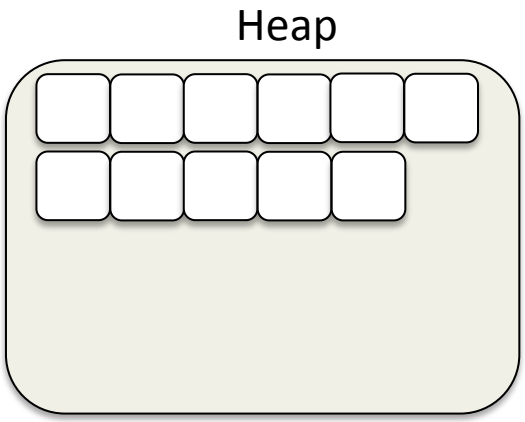
```
createObject {

    ptr = allocate object

    call constructor(ptr)

    return ptr;
}
```



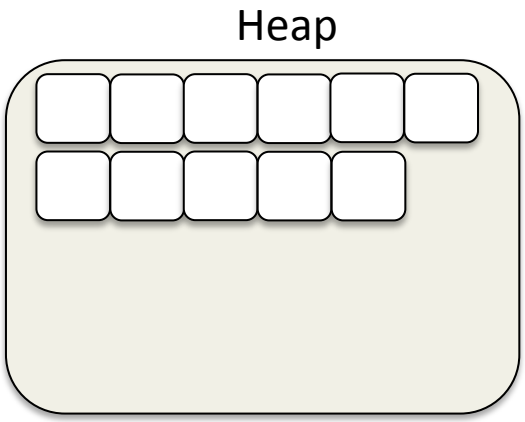
```
createObject {
    ptr = allocate object
    call constructor(ptr)
    return ptr;
}
```



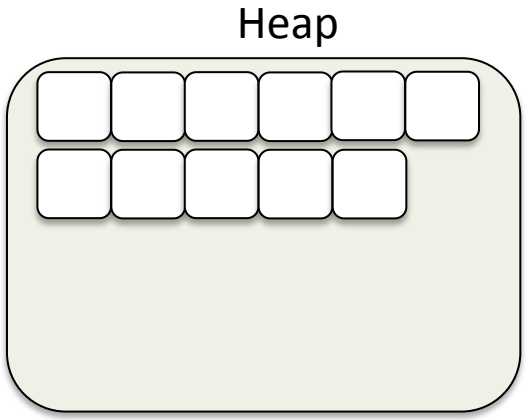
```
createObject {
    ptr = allocate object
    call constructor(ptr)
    return ptr;
}
```



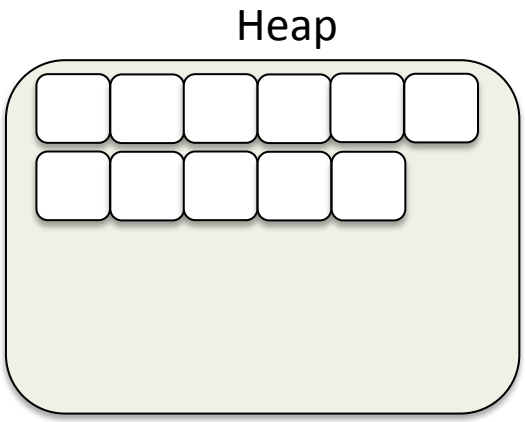
```
createObject {

    ptr = allocate object

    call constructor(ptr)

    return ptr;
}
```



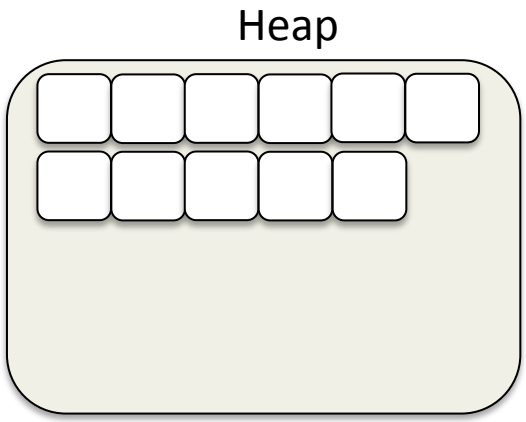
```
 constructor() {
    value = Cls.STATIC;
 }

createObject {
    ptr = allocate object
    call constructor(ptr)
    return ptr;
}
```



```
constructor() {
    value = Cls.STATIC;
}

createObject {
    ptr = allocate object
    call constructor(ptr)
    return ptr;
}
```

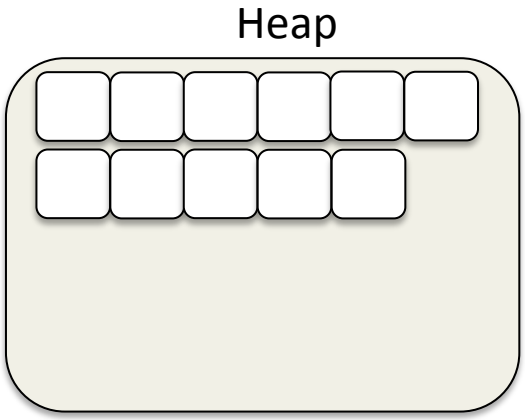


```
static {
    object = new Object();
}

constructor() {
    value = Cls.STATIC;
}

createObject {
    ptr = allocate object
    call constructor(ptr)
    return ptr;
}
```

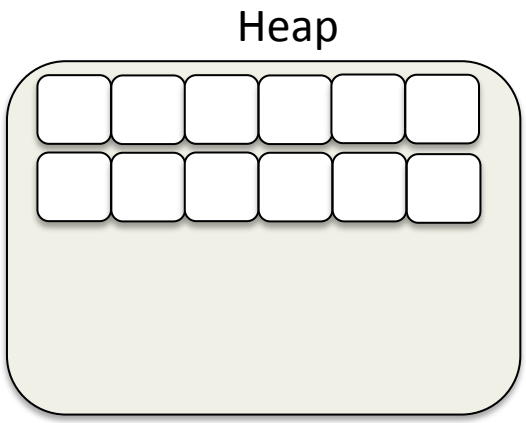


```
static {
    object = new Object();
}

constructor() {
    value = Cls.STATIC;
}

createObject {
    ptr = allocate object
    call constructor(ptr)
    return ptr;
}
```

Class Loader

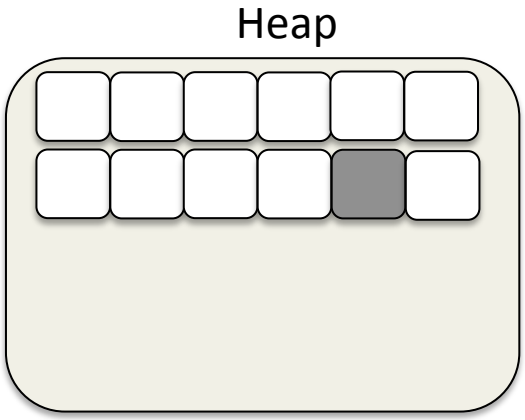


```
static {
    object = new Object();
}

constructor() {
    value = Cls.STATIC;
}

createObject {
    ptr = allocate object
    call constructor(ptr)
    return ptr;
}
```



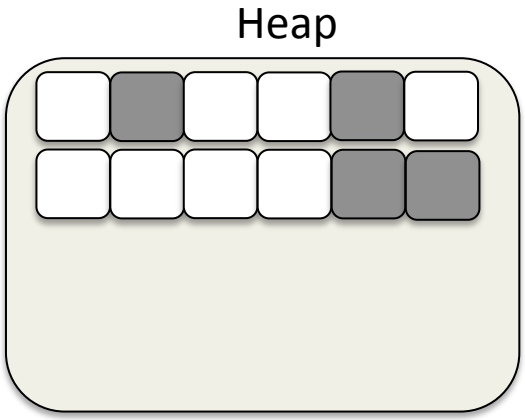
```
static {
    object = new Object();
}

constructor() {
    value = Cls.STATIC;
}

createObject {
    ptr = allocate object
    call constructor(ptr)
    return ptr;
}
```

Class Loader



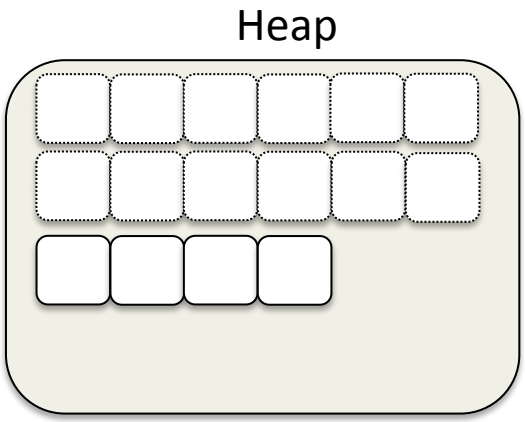
```
static {
    object = new Object();
}

constructor() {
    value = Cls.STATIC;
}

createObject {
    ptr = allocate object
    call constructor(ptr)
    return ptr;
}
```

Class Loader



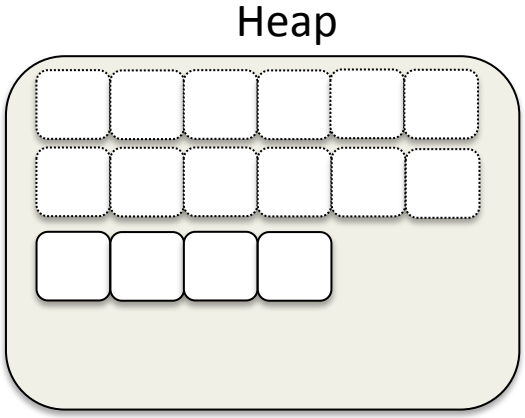
```
static {
    object = new Object();
}

constructor() {
    value = Cls.STATIC;
}

createObject {
    ptr = allocate object
    call constructor(ptr)
    return ptr;
}
```

Class Loader

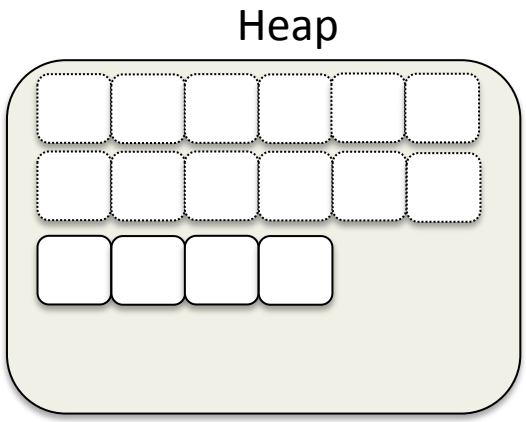


```
constructor() {
    value = Cls.STATIC;
}

createObject {
    ptr = allocate object
    call constructor(ptr)
    return ptr;
}
```

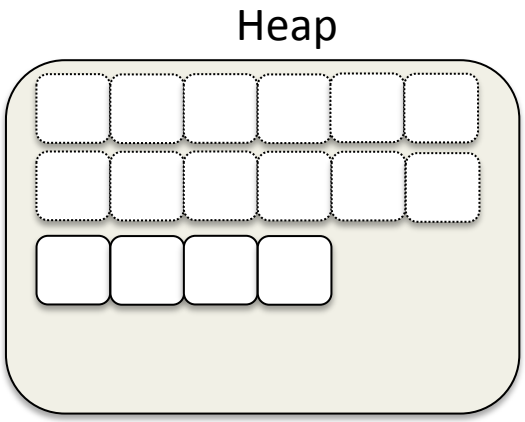
Class Loader



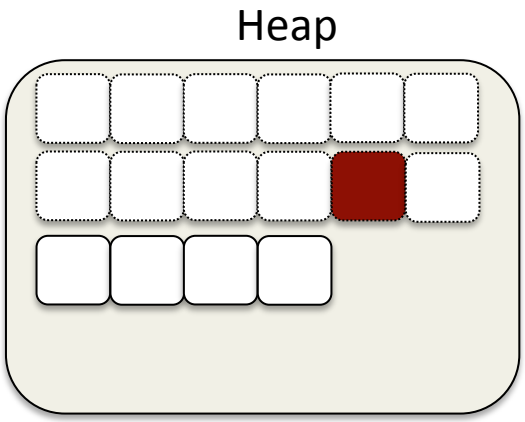
```
createObject {

    ptr = allocate object

    call constructor(ptr)

    return ptr;
}
```



```
createObject {
    ptr = allocate object
    call constructor(ptr)
    return ptr;
}
```



```
createObject {
    ptr = allocate object
    call constructor(ptr)
    return ptr;
}
```



Final Project

- The final project is now posted
  - No takers for the team version
- Implement generational garbage collector
  - See slide deck 14 for details
  - Lecture from Monday October 27th
- Three week timeframe
  - Non-trivial amount of work
  - Instructions assume good understanding of the code

Generational Collector Design

- Fixed-size nursery
  - Bump pointer allocation
  - Evacuate objects to the mature space
- Mark/Sweep mature space management
  - Allocation happens only during minor GC
  - Allocation uses first-fit algorithm
- Write barrier on pointer writes

Part I: Write Barrier

- Don't scan the full heap during minor collections
  - Major performance advantage
  - Need to track references from old to new space
- Use a write barrier to track pointer writes
  - Two bytecode instructions can write pointers
  - Instrument those instructions in the interpreter
- New interface and implementation class

Part I: Write Barrier

- Write barrier maintains list of old-new references
  - Actually, objects that may point from old to new
- Set should only contain relevant objects
  - Don't need to track other references
  - Correctly building the remembered set is important
- Remembered set allocated off to the side
  - Don't need to worry about space overhead

Part II: Non-Contiguous Region

- Mark/Sweep collector causes fragmentation
  - Objects are deleted without moving
- Need an allocator to allocate between objects
  - Current bump pointer region won't work
- New Region class for non-contiguous allocation
  - Method on Heap interface to get an instance

Part II: Non-Contiguous Region

- Allocate to the region using first-fit
  - May be best to track objects and holes
  - Metadata can be stored off to the side

- Support the Region interface's operations
  - `allocate`, `free`, `pointerInRegion`
  - `reset` operation not available
- Will need other operations for mark/sweep GC
  - Some way to scan through all objects
  - Implement on the class, not the interface

Part III: Mark/Sweep Collector

- Mature space managed using mark/sweep
  - Use a single non-contiguous region
  - New memory manager class skeleton
- Root set the same as in Assignment 3
  - Stack and local variables
  - Static fields
  - Interned strings

Part III: Mark/Sweep Collector

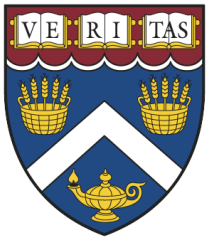
- Trigger garbage collection manually
  - Allocation only happens during minor collection
  - Heap in a strange state

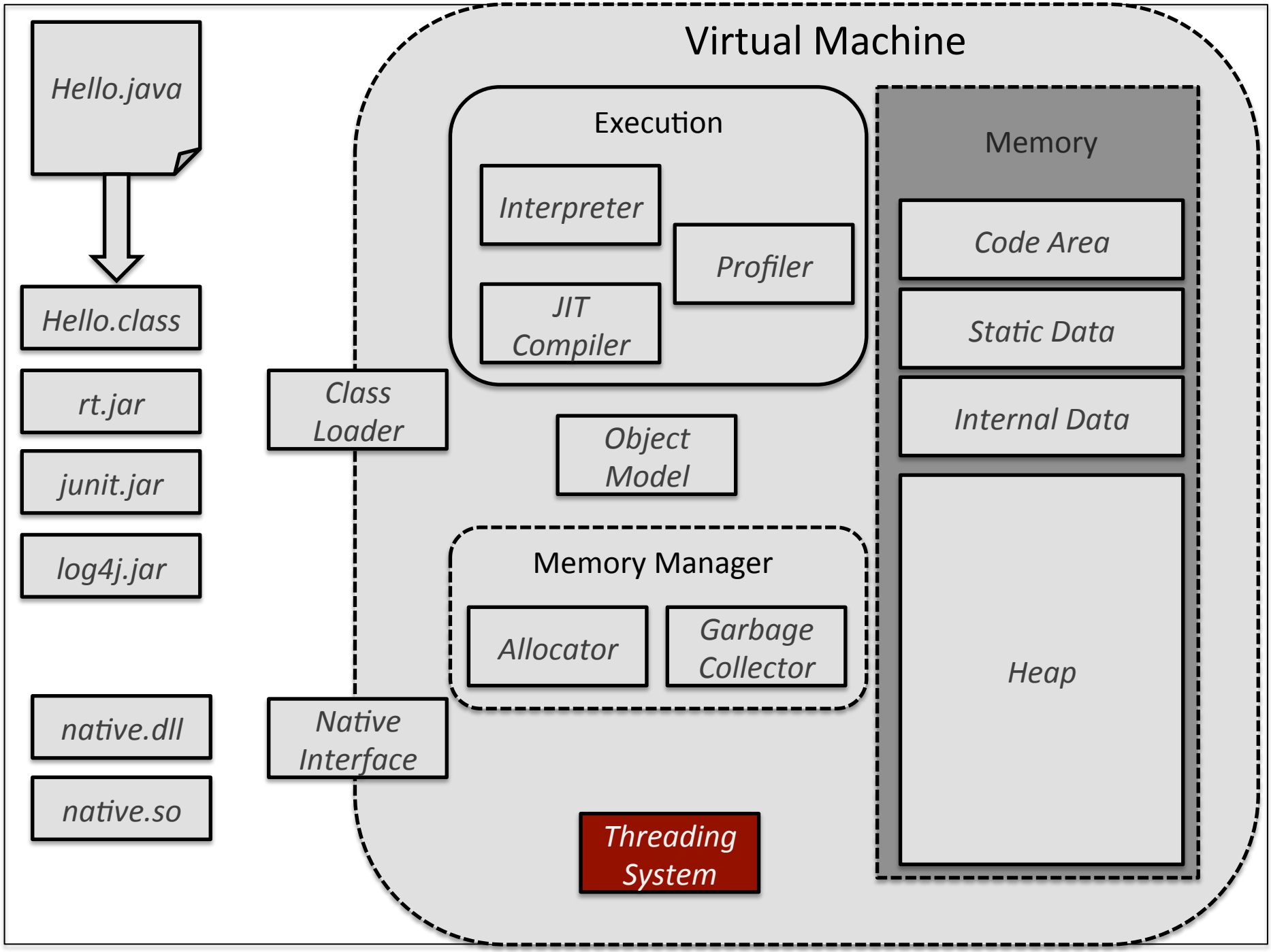
- May want to test this collector separately
  - Add temporary logic to trigger collection
  - Mark/Sweep will be a fully-functional GC
  - See `SimpleJavaVM.java` to switch collectors

Part IV: Nursery Collection

- Final part of the generational collector
- Roughly based on the semi-space collector
  - Root set includes the remembered set
  - Nursery is a single region, not a semi-space
  - Use an explicit gray stack for minor collection
- Trigger major collection after minor GC
  - Call when mature space is >= 2 * nursery size

Threading and Concurrency




Hello.java
Hello.class
rt.jar
junit.jar
log4j.jar
native.dll
native.so
Virtual Machine
Execution
Interpreter
Profiler
JIT Compiler
Class Loader
Object Model
Memory Manager
Allocator
Garbage Collector
Native Interface
Threading System
Memory
Code Area
Static Data
Internal Data
Heap

Concurrency

- Multiple operations occurring simultaneously
  - Possibly interacting with one another
- Concurrent systems are now commonplace
  - Offer major performance improvements
  - Make development far more difficult
- VM concurrency falls into two broad groups:
  - Application-level concurrency
  - VM internal concurrency

Processes and Threads

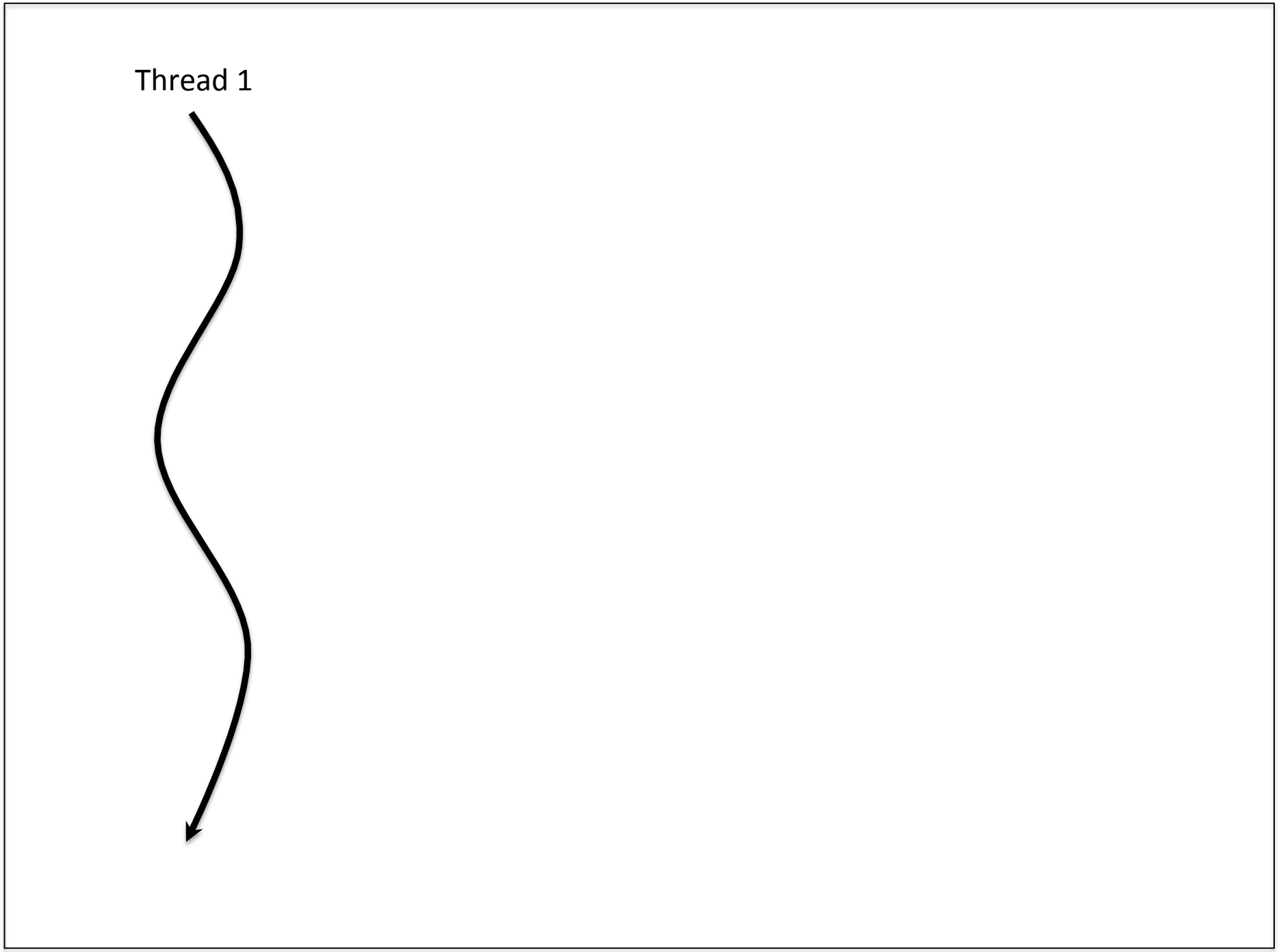
- Primary unit of concurrency is the process
  - Separate memory space
  - Isolated from other processes
- Threads allow concurrency within a process
  - Far more lightweight than processes
  - Shared address space
  - One thread's actions can affect another

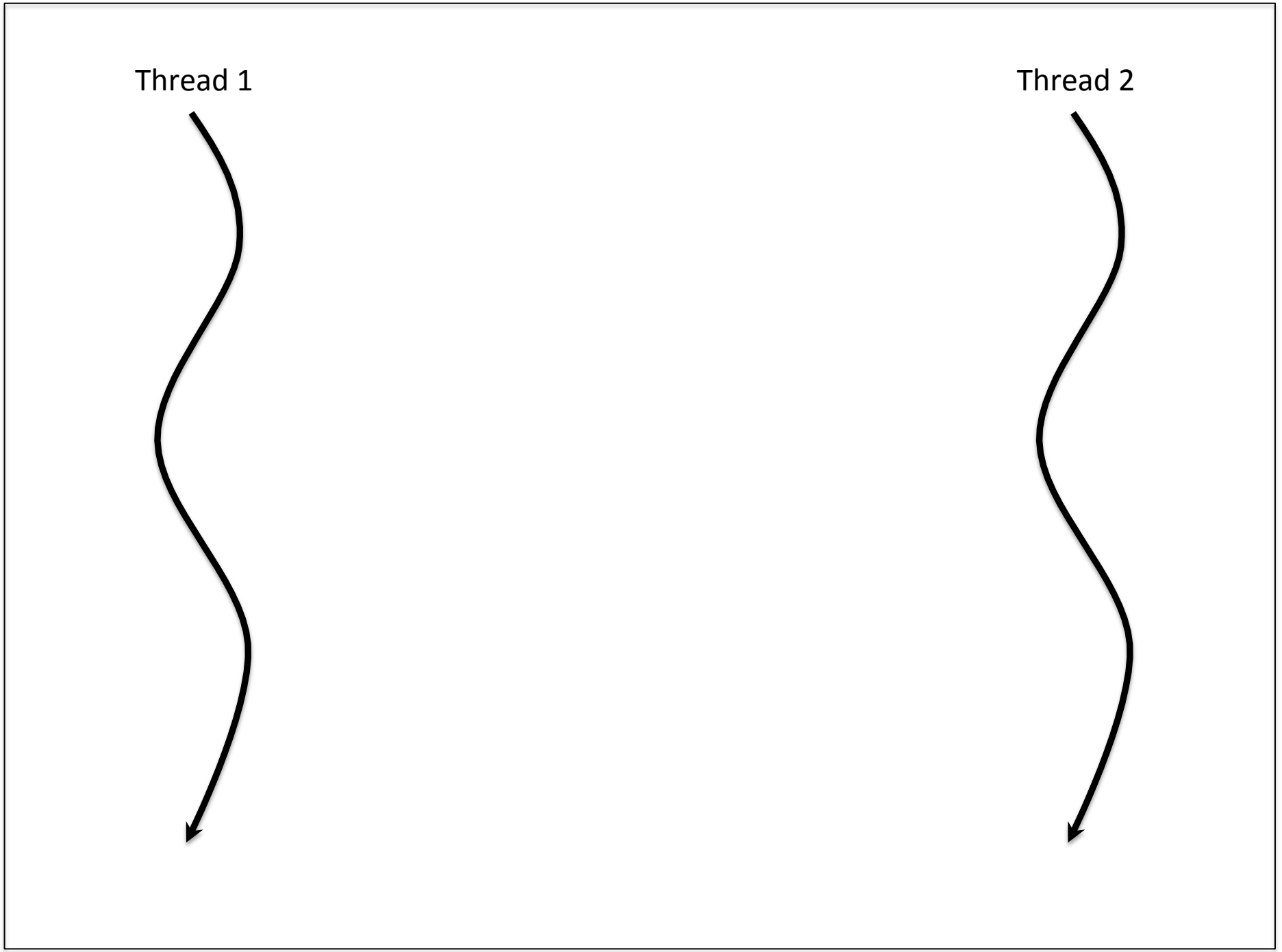
Moore's Law

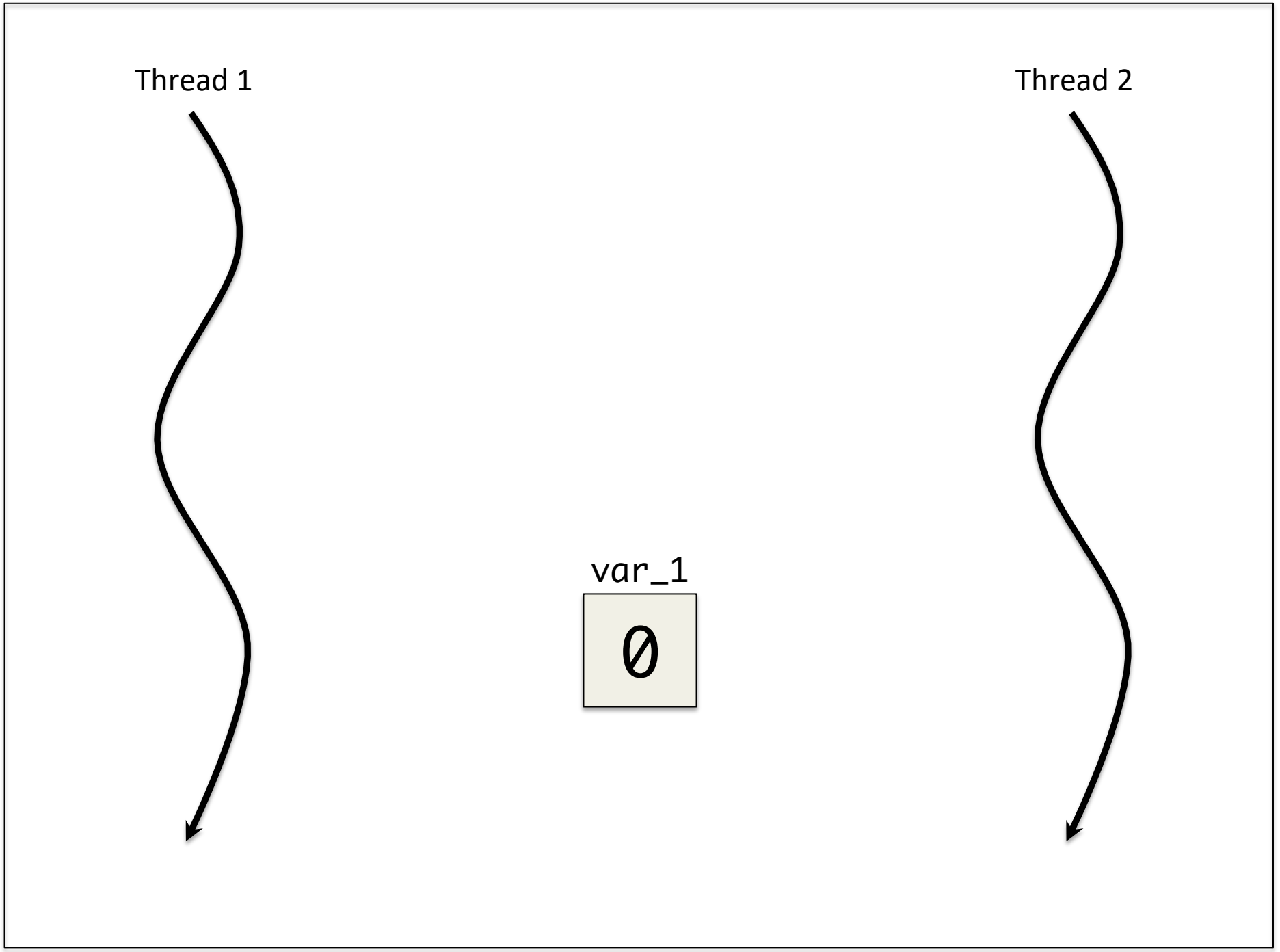
- Frequently misstated
  - The speed of processors doubles every two years
- Actually relates to processor manufacturing
  - Transistor density doubles every two years
- For a long time, the two were equivalent
- Complex circuitry means more heat
  - Modern designs use multiple simpler cores

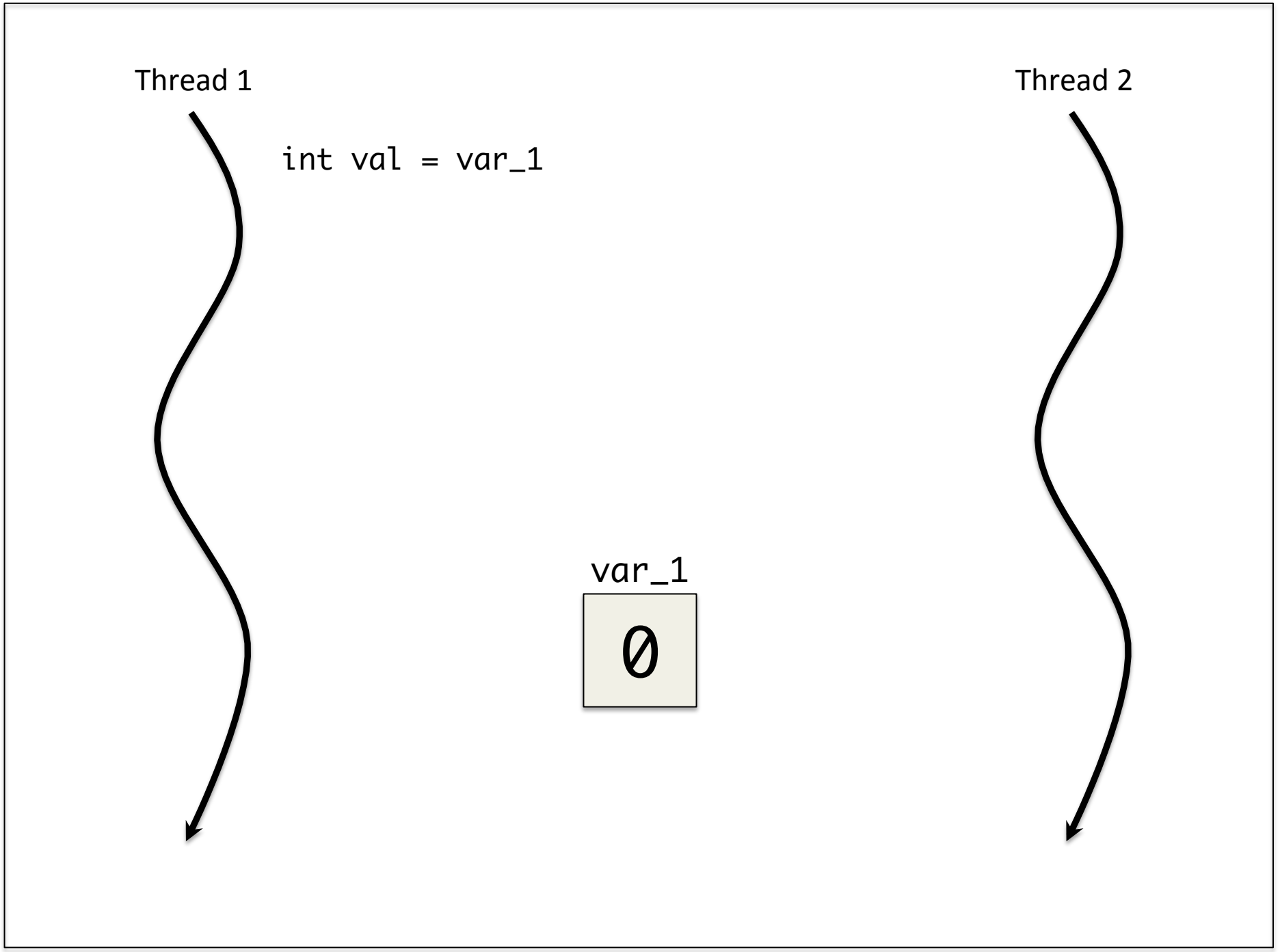
Hardware Concurrency Support

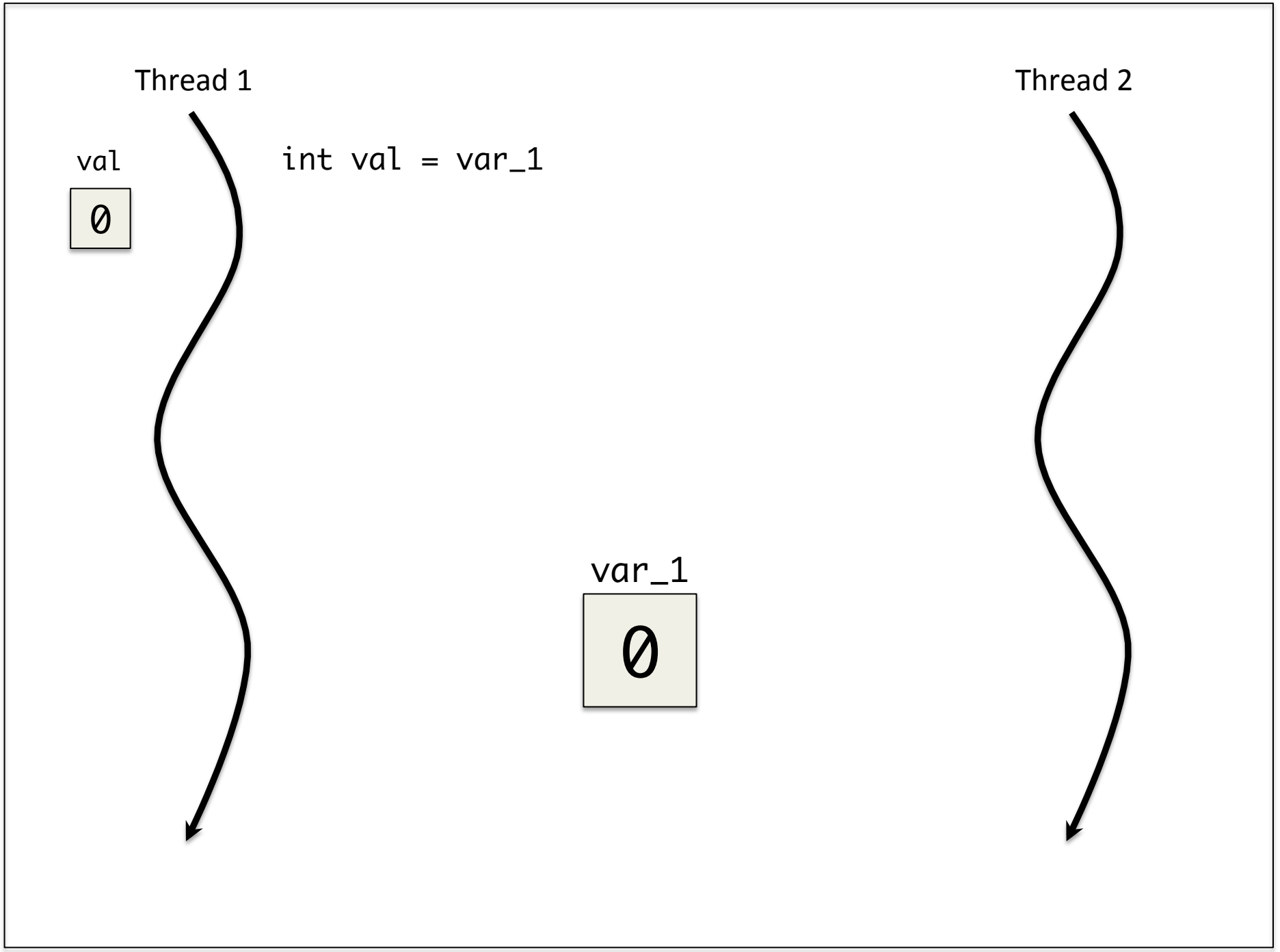
- Multicore processors allow genuine concurrency
  - Previously required multiple CPUs
  - Single core may also provide hardware multithreading
- CPU optimizes individual threads
  - Out of order execution
- Also provides instructions to handle concurrency
  - Memory fences
  - Atomic CAS instructions

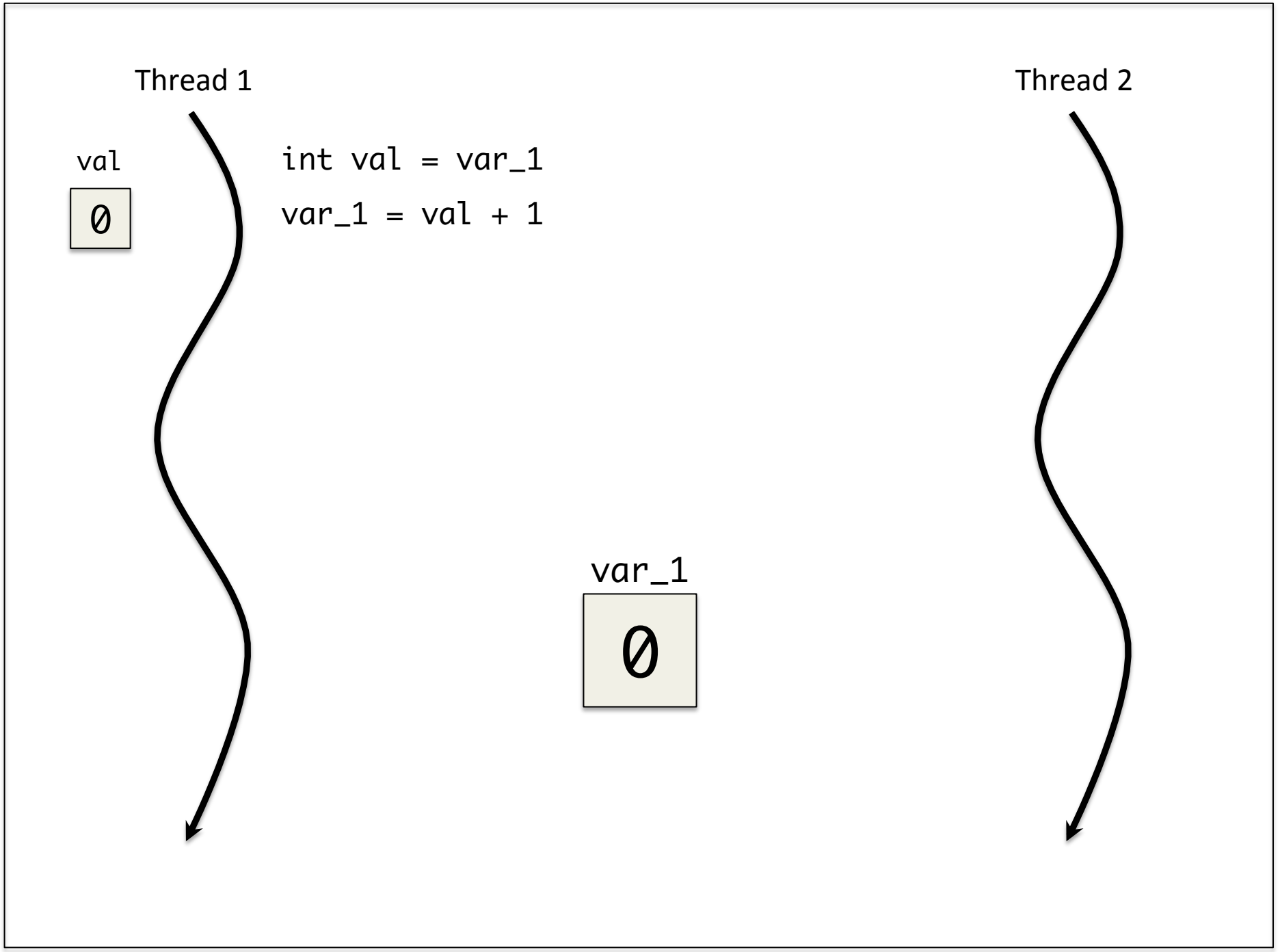

Thread 1

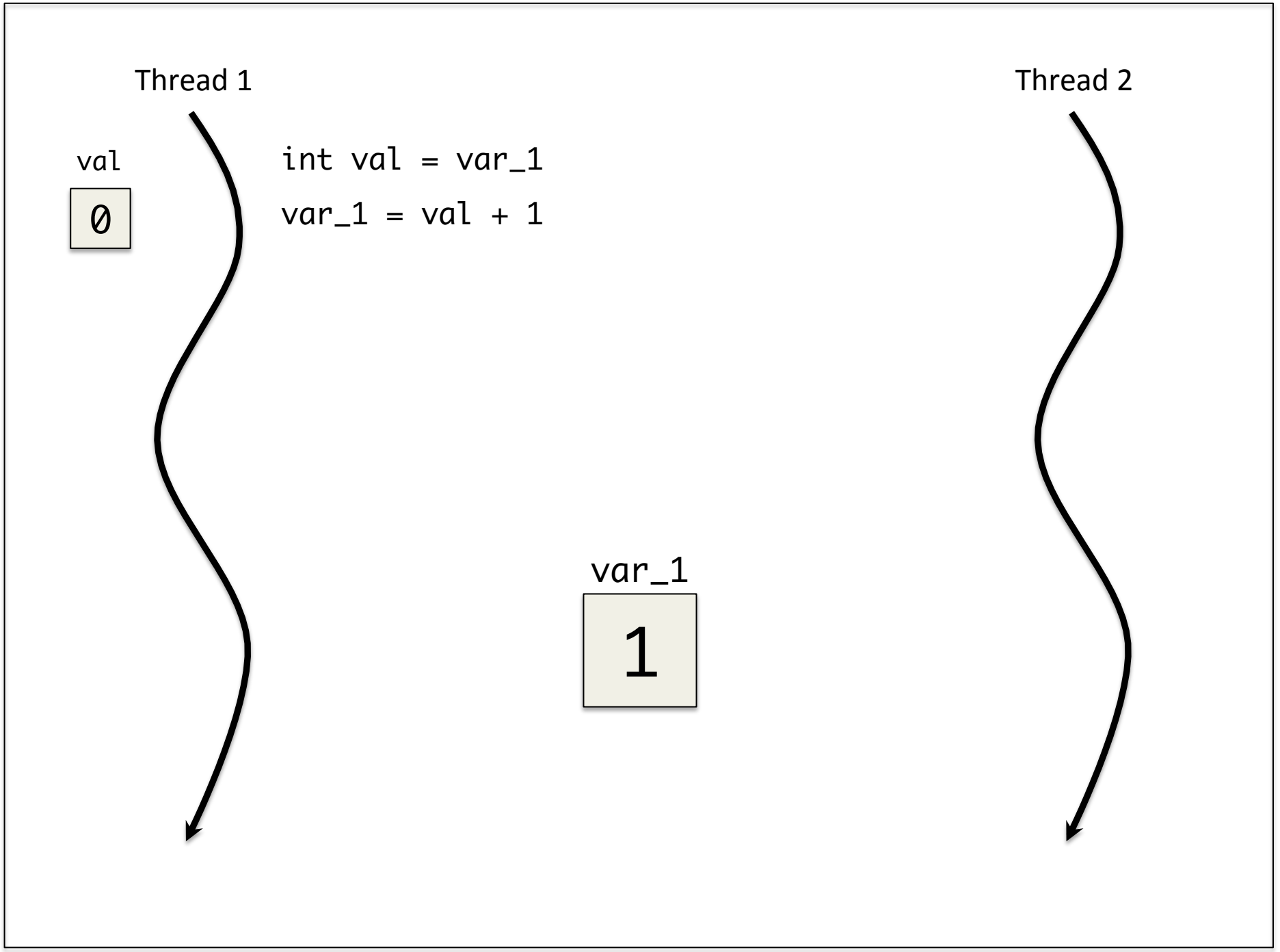

Thread 1
Thread 2

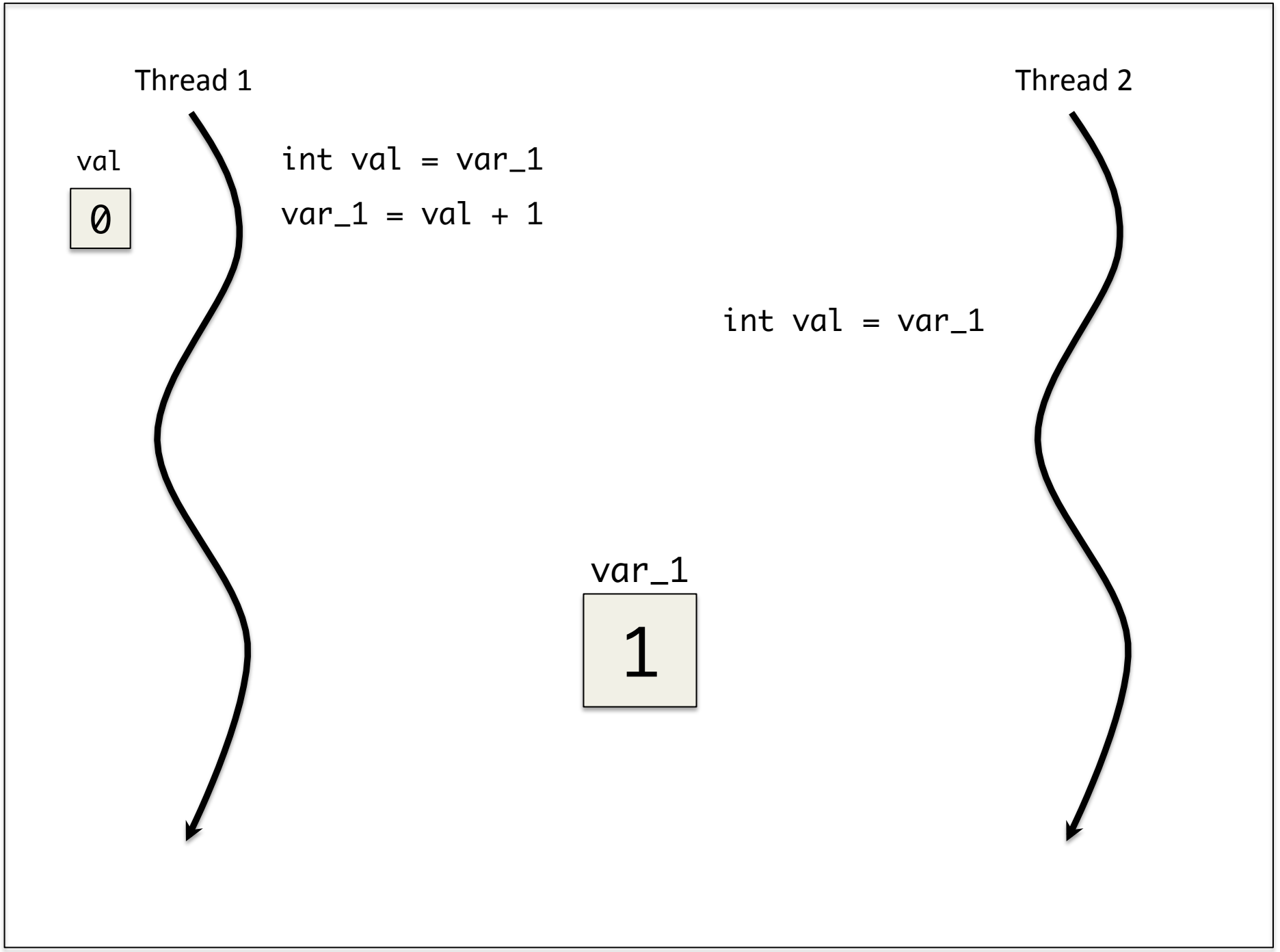

Thread 1
Thread 2
var_1
0

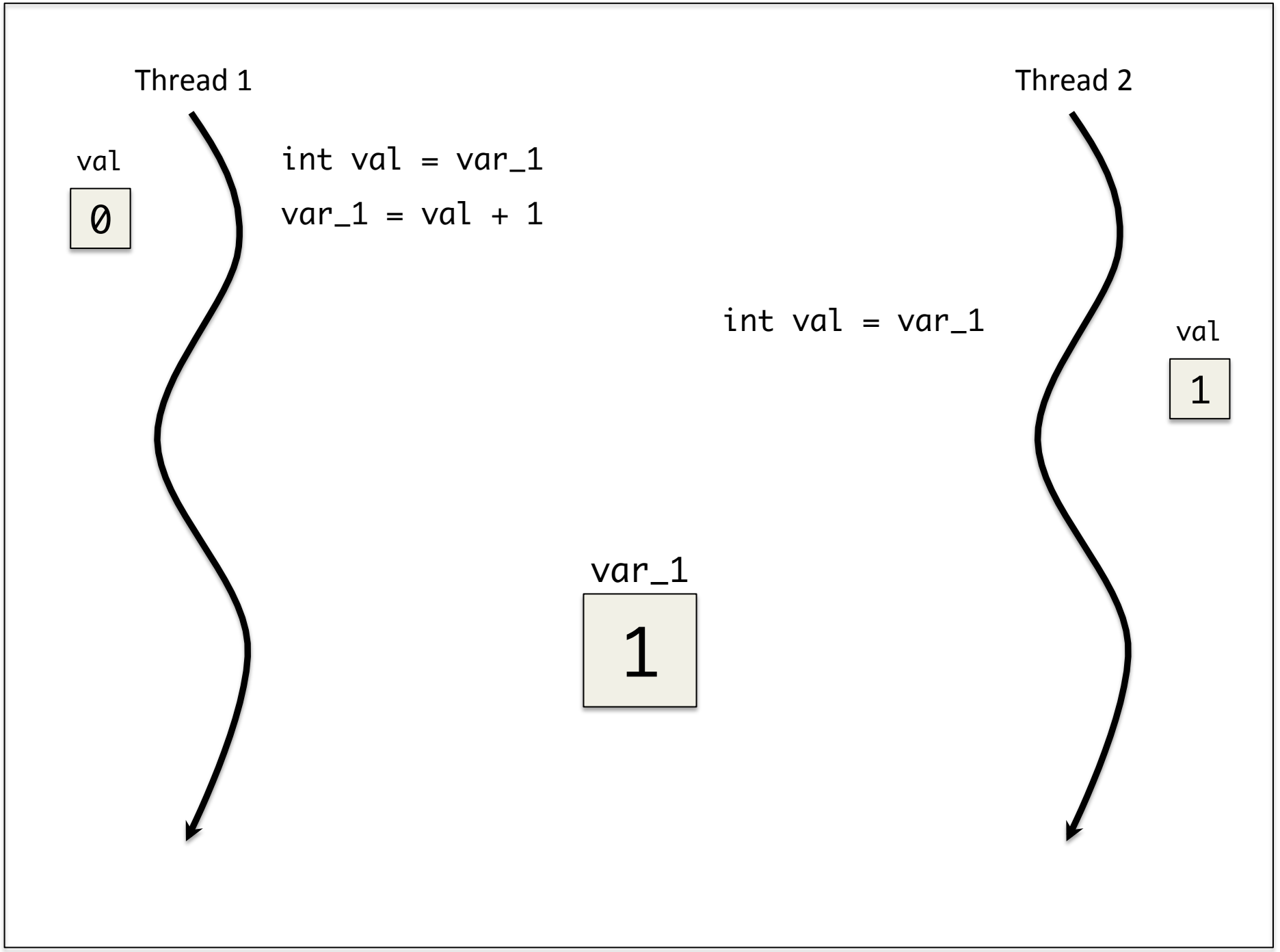

Thread 1
int val = var_1
Thread 2
var_1
0

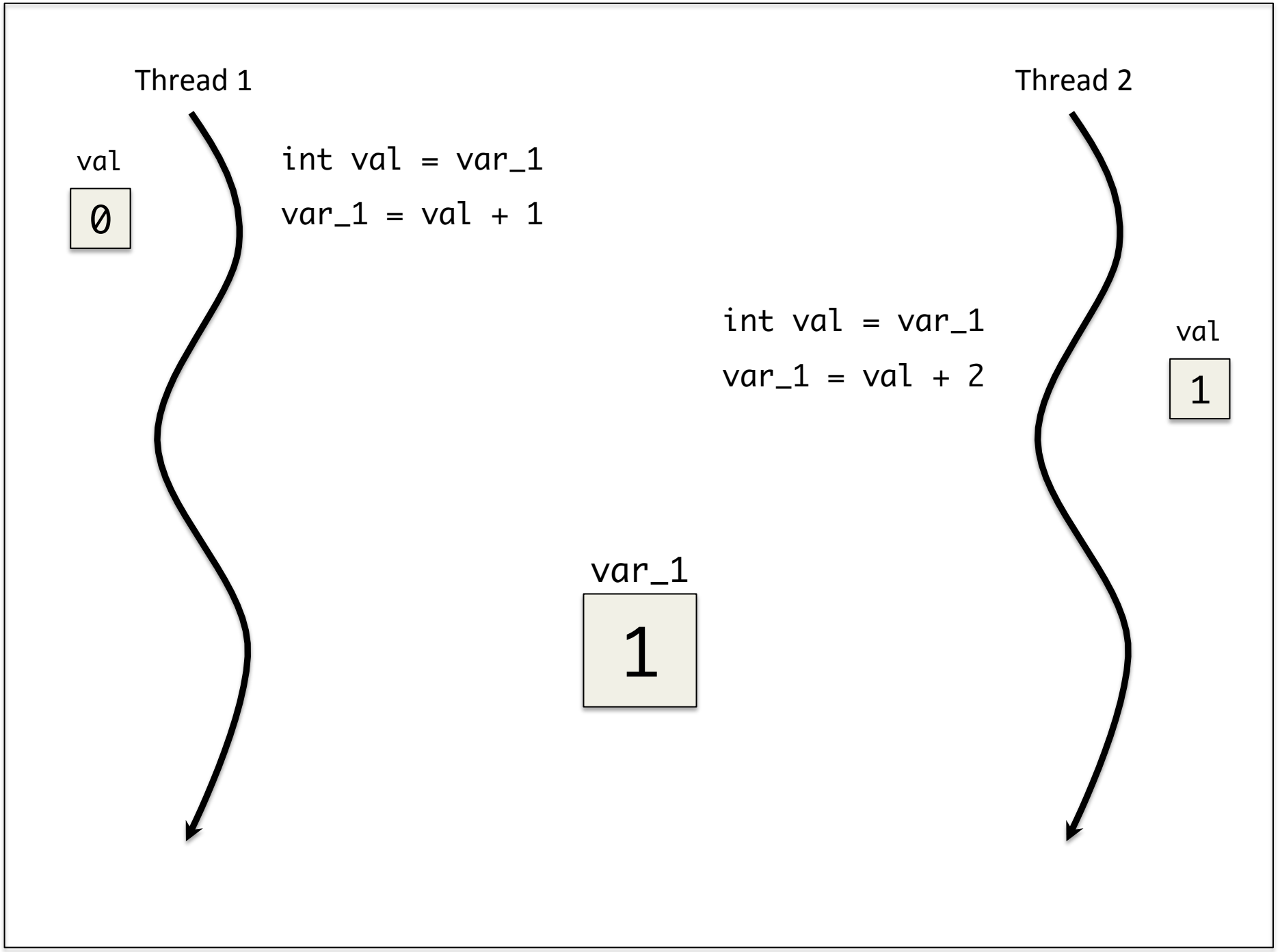

Thread 1
Thread 2
val
0
int val = var_1
var_1
0

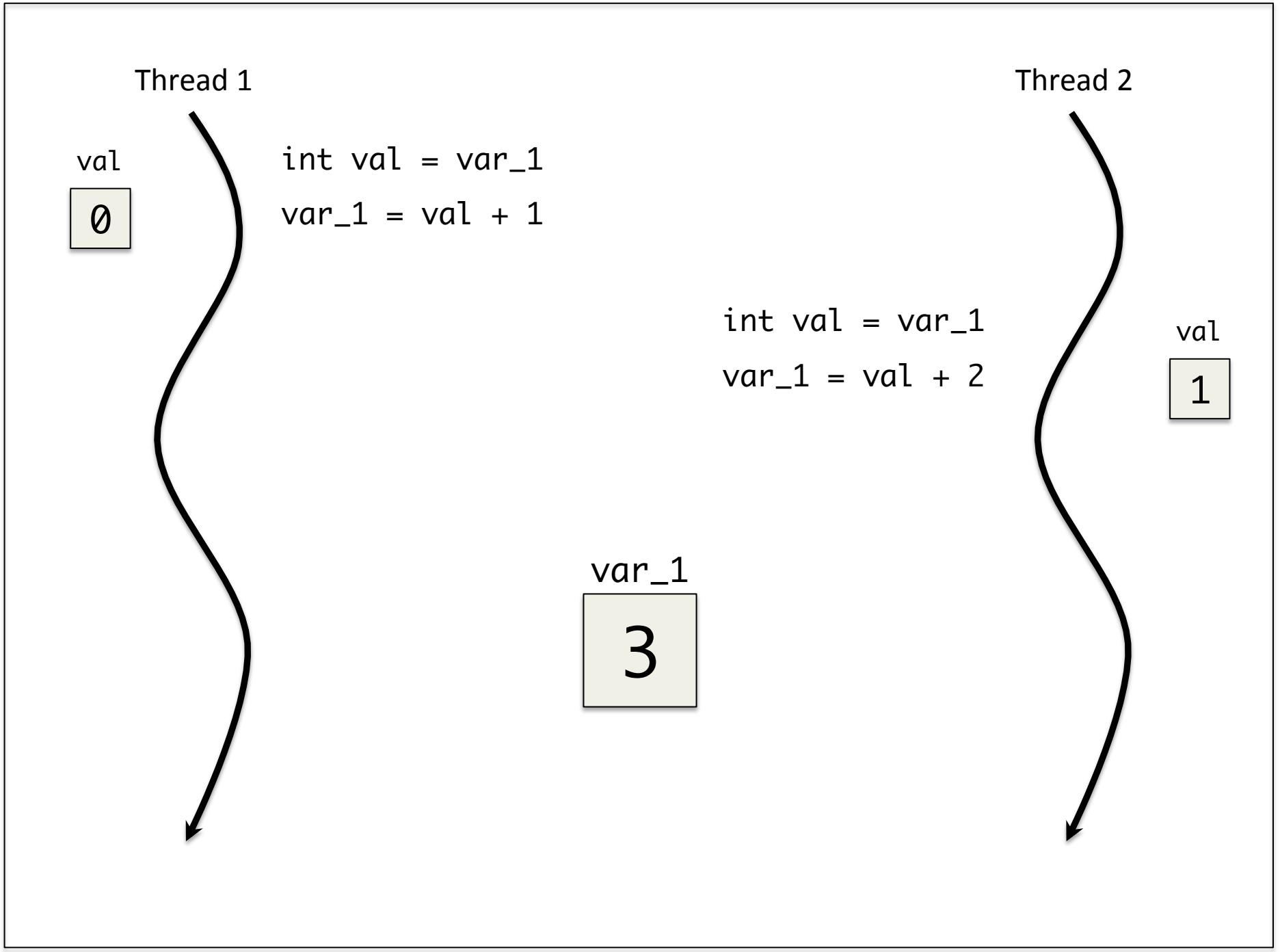

Thread 1
Thread 2
val
0
int val = var_1
var_1 = val + 1
var_1
0

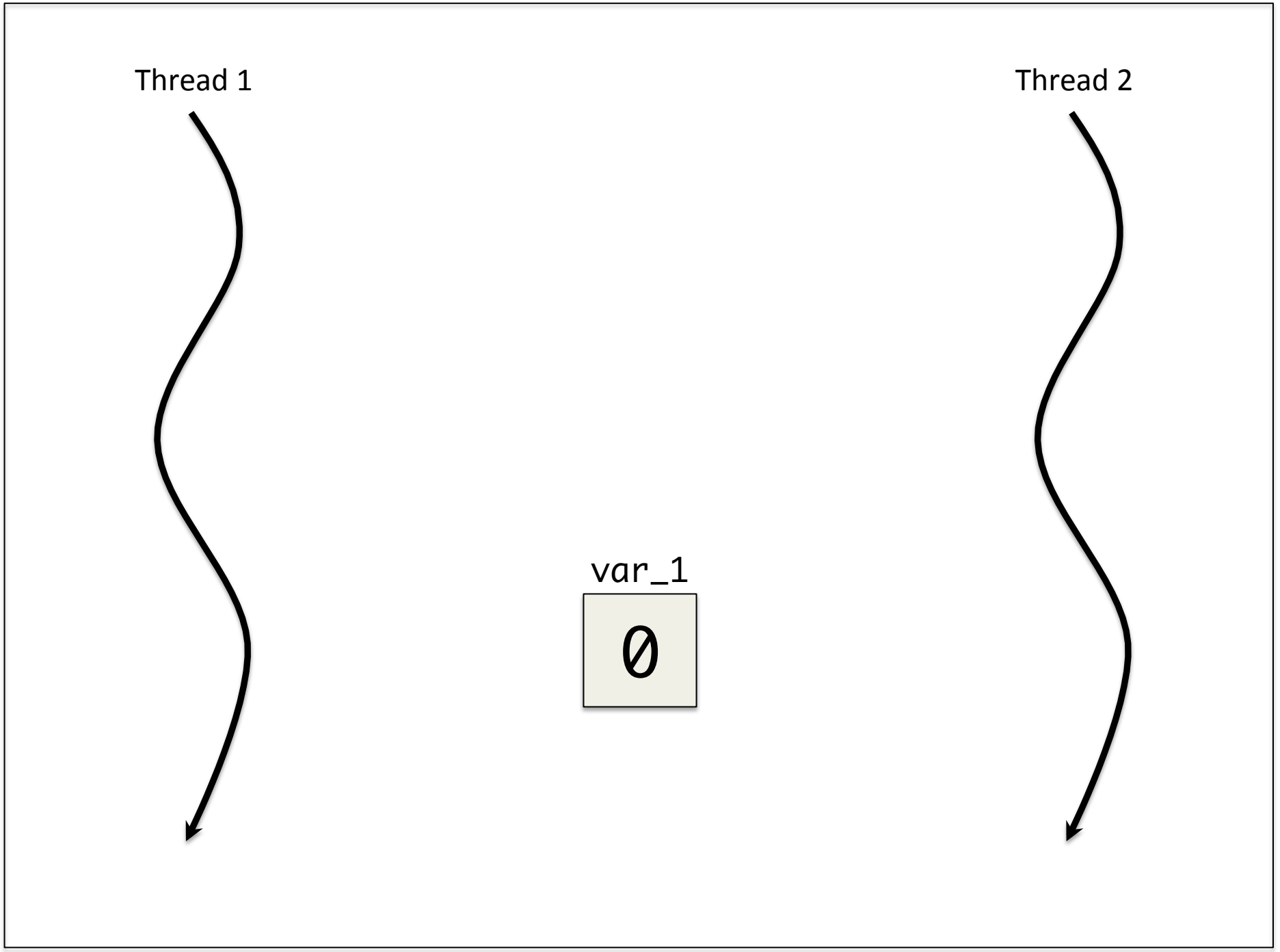

Thread 1
Thread 2
val
0
int val = var_1
var_1 = val + 1
var_1
1

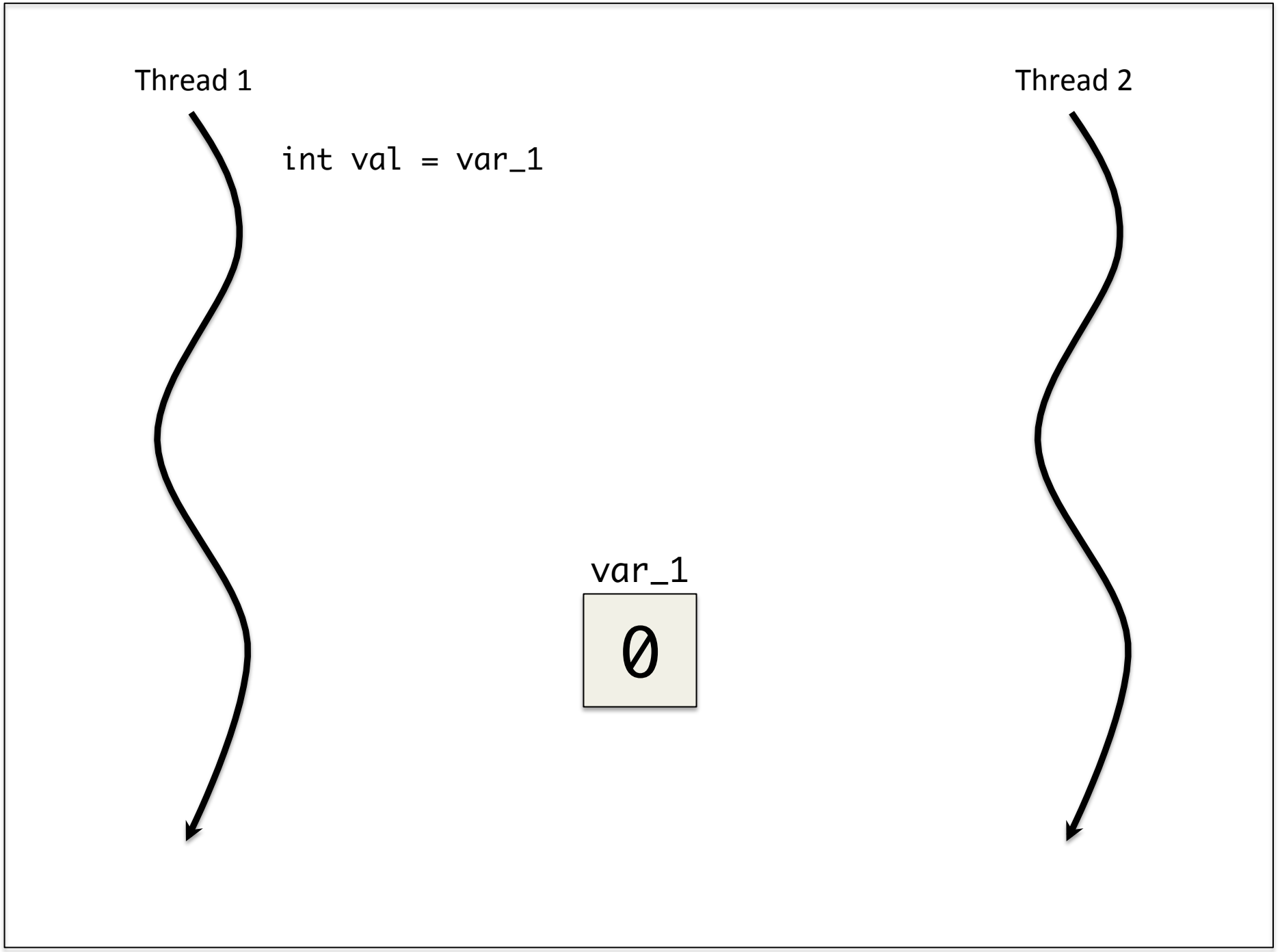

Thread 1
Thread 2
val
0
int val = var_1
var_1 = val + 1
int val = var_1
var_1
1

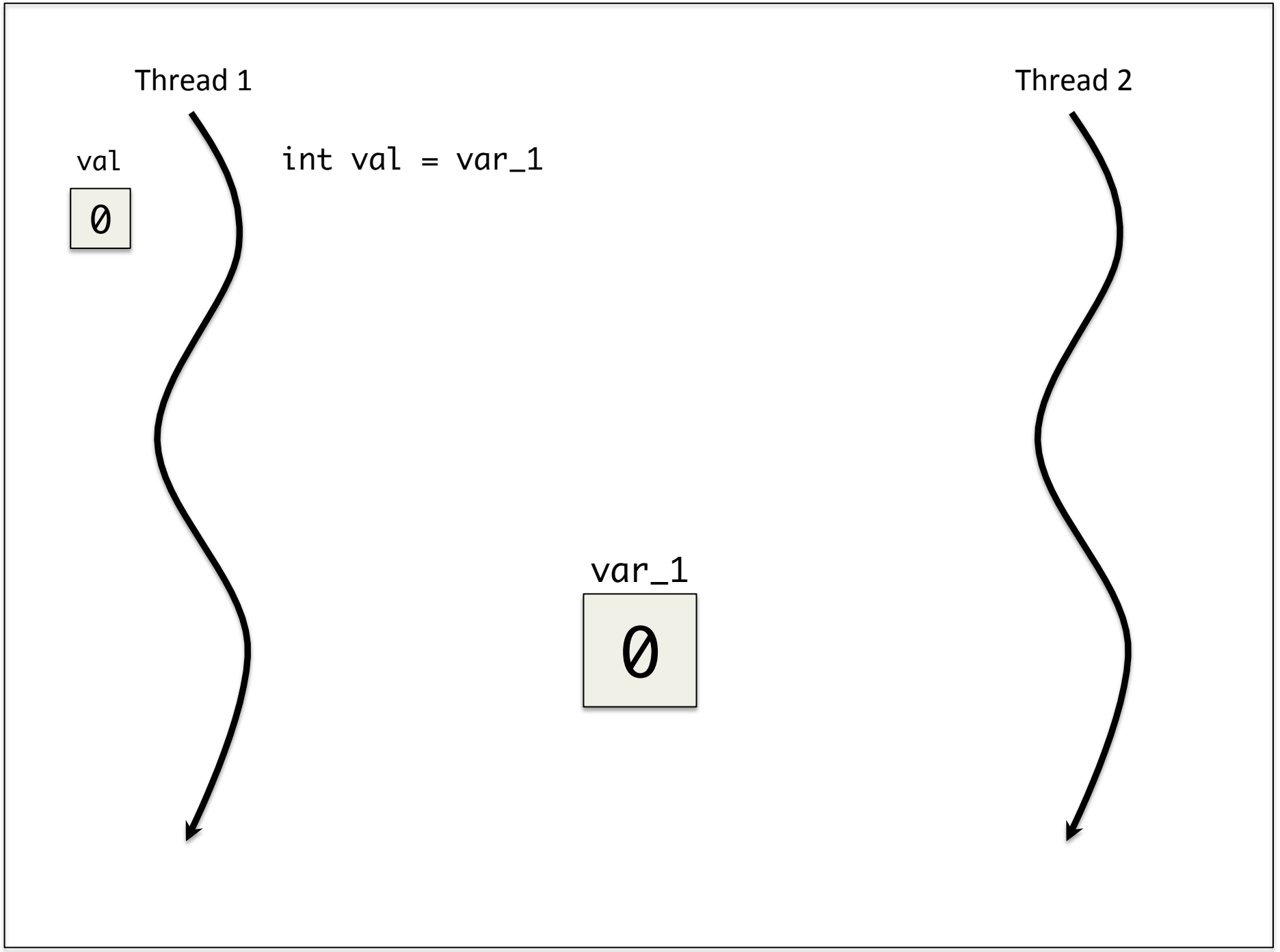

Thread 1
Thread 2
val
0
int val = var_1
var_1 = val + 1
int val = var_1
val
1
var_1
1

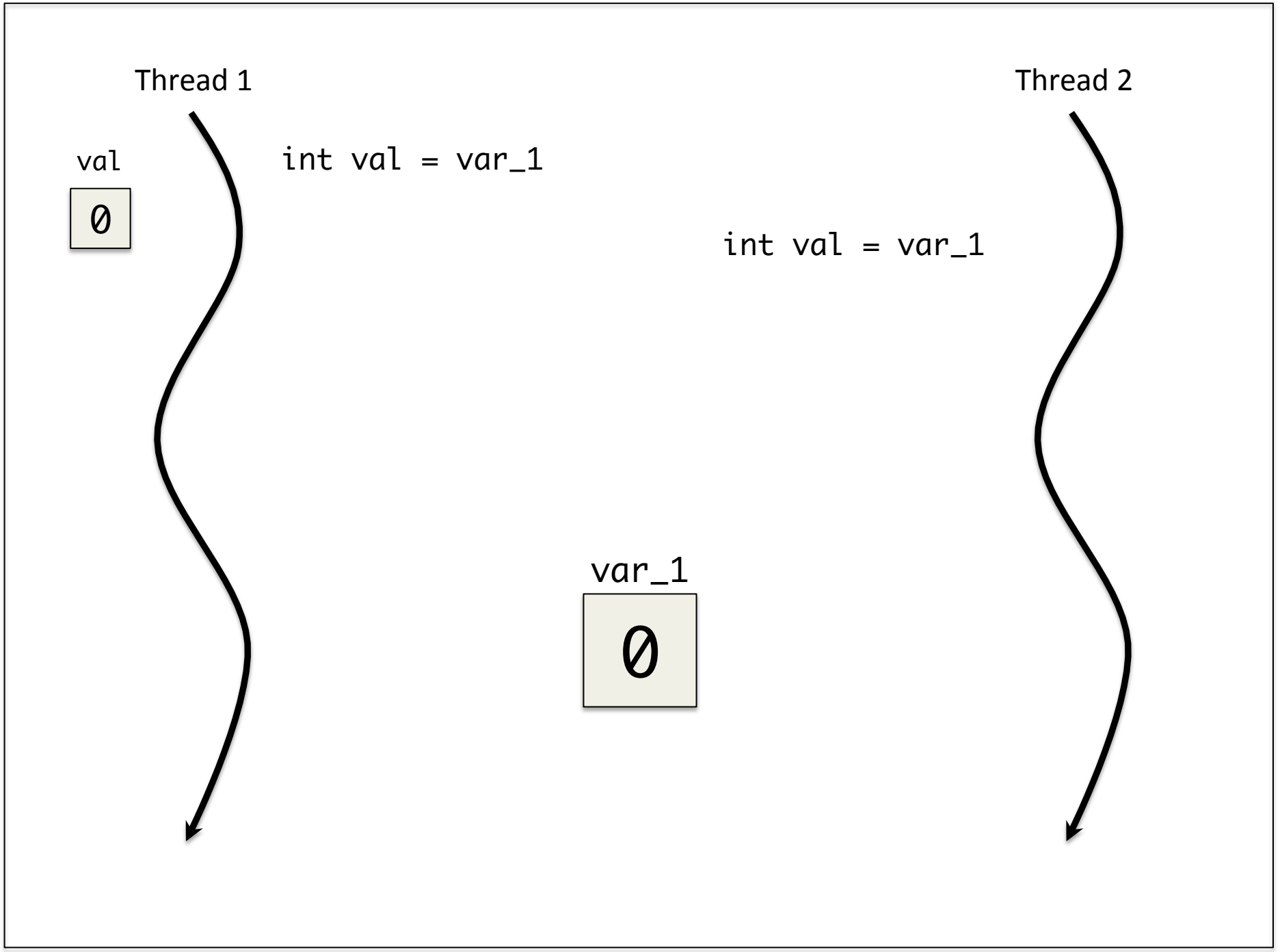

Thread 1
val
0
int val = var_1
var_1 = val + 1
Thread 2
int val = var_1
var_1 = val + 2
val
1
var_1
1

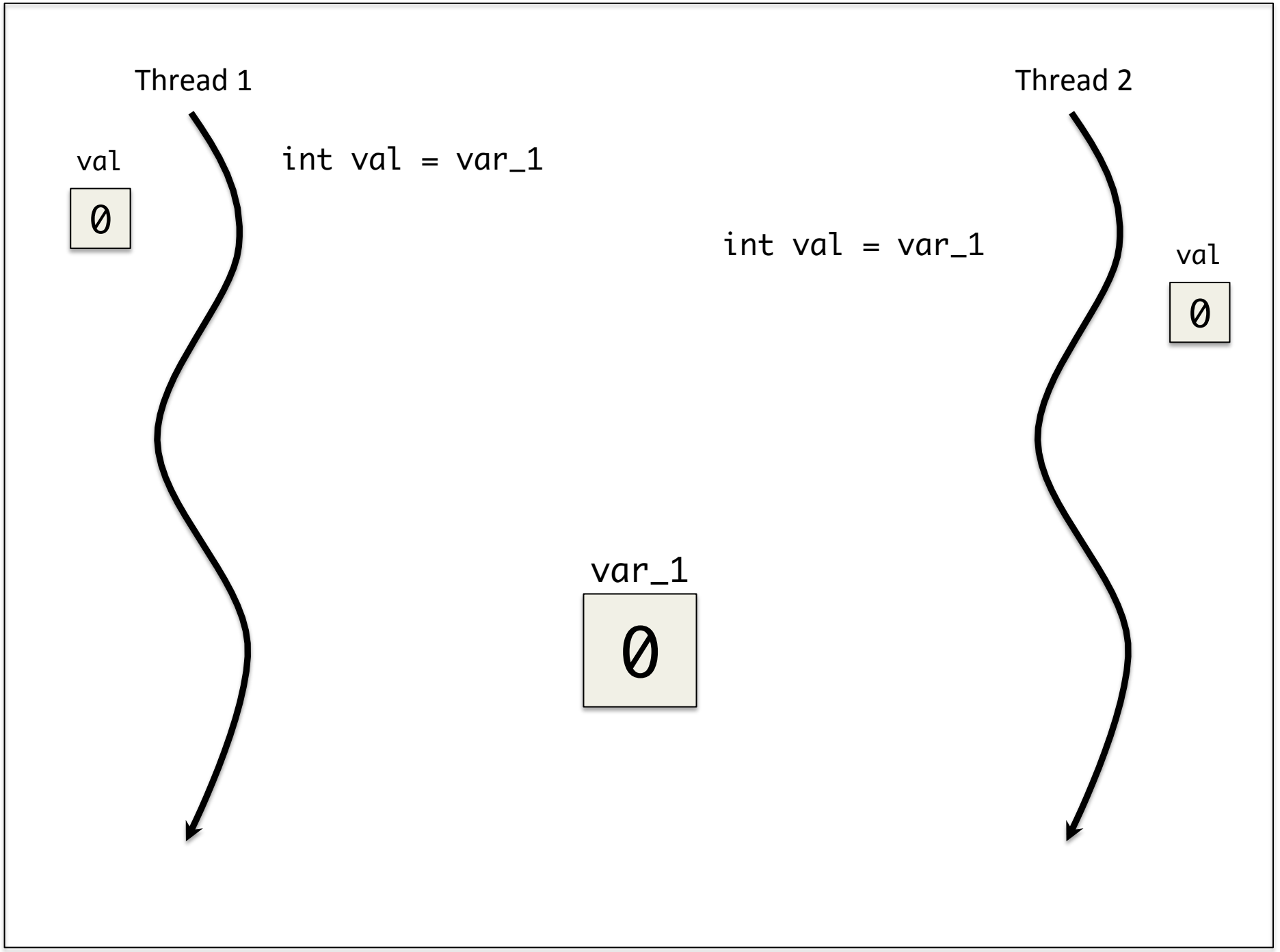

Thread 1
val
0
int val = var_1
var_1 = val + 1
Thread 2
int val = var_1
var_1 = val + 2
val
1
var_1
3

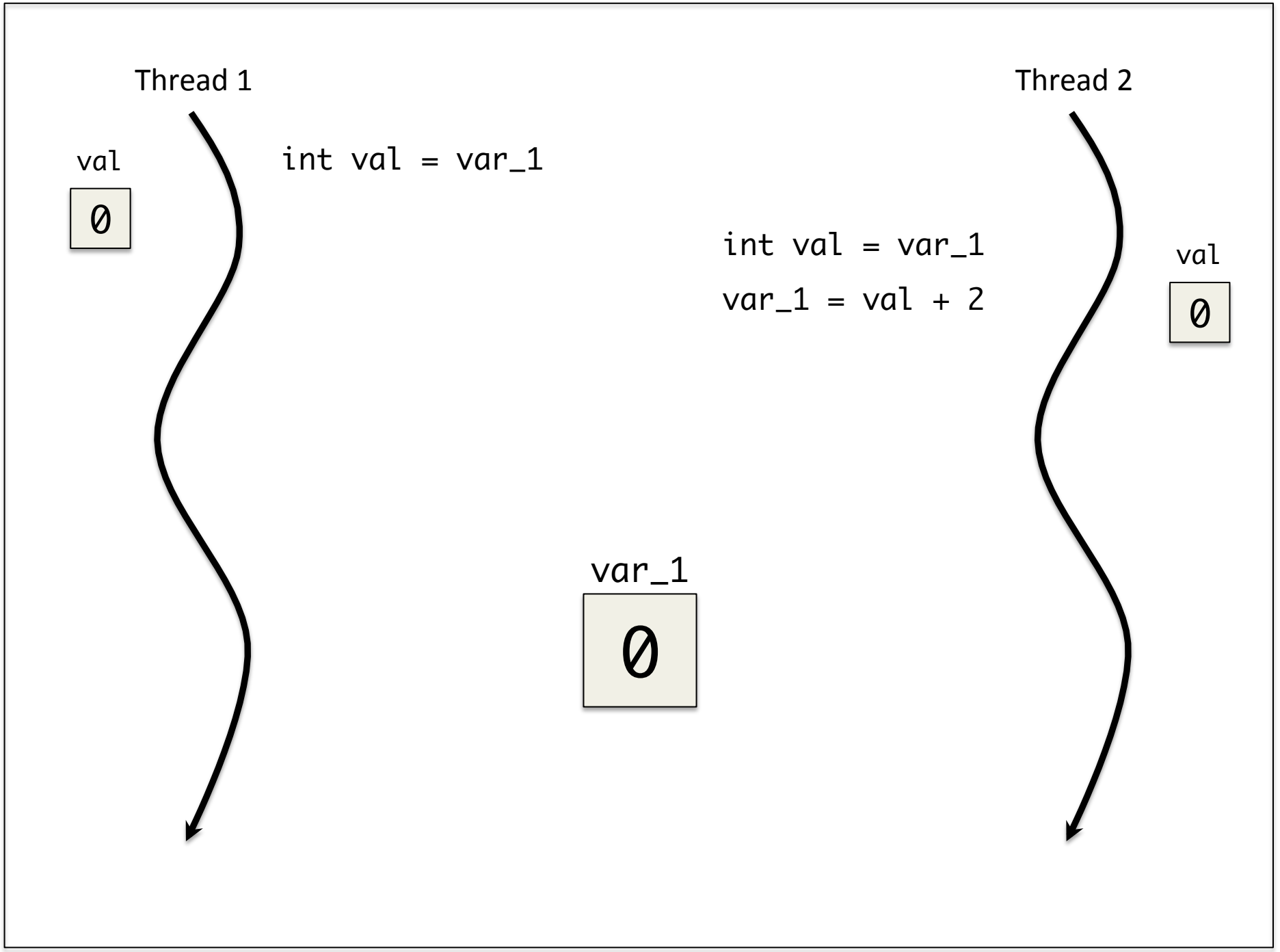

Thread 1
Thread 2
var_1
0

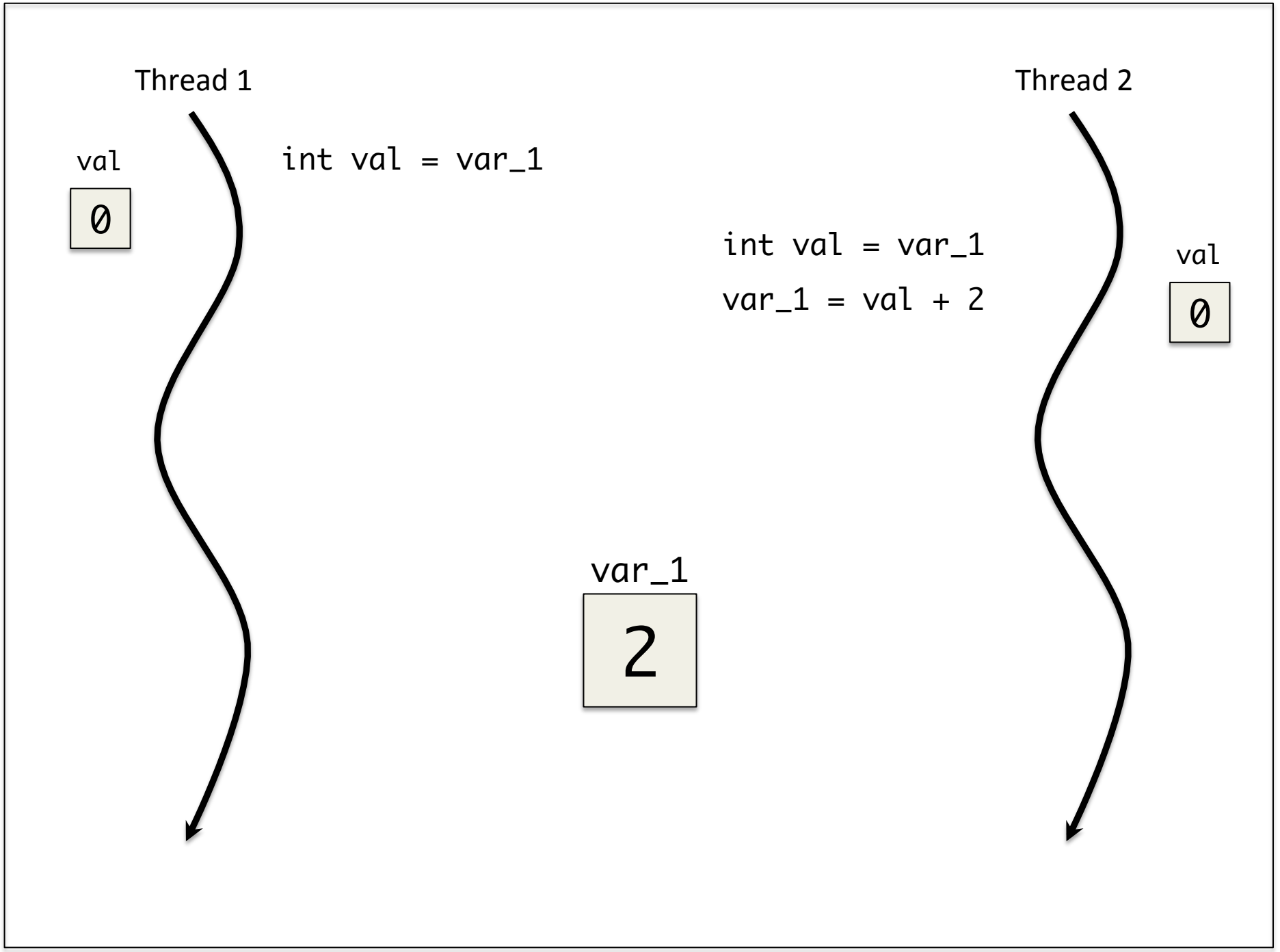

Thread 1
int val = var_1
Thread 2
var_1
0

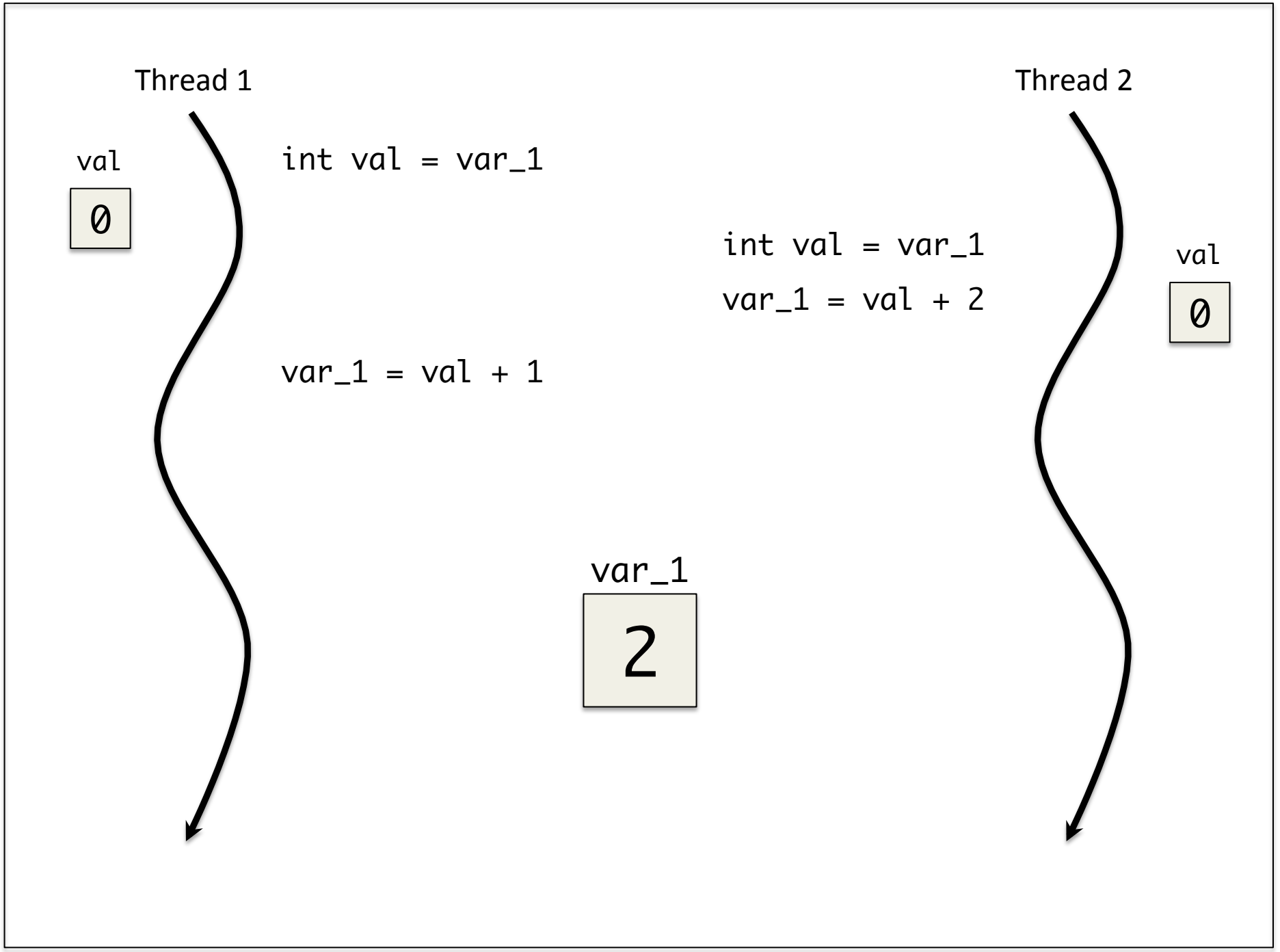

Thread 1
Thread 2
val
0
int val = var_1
var_1
0

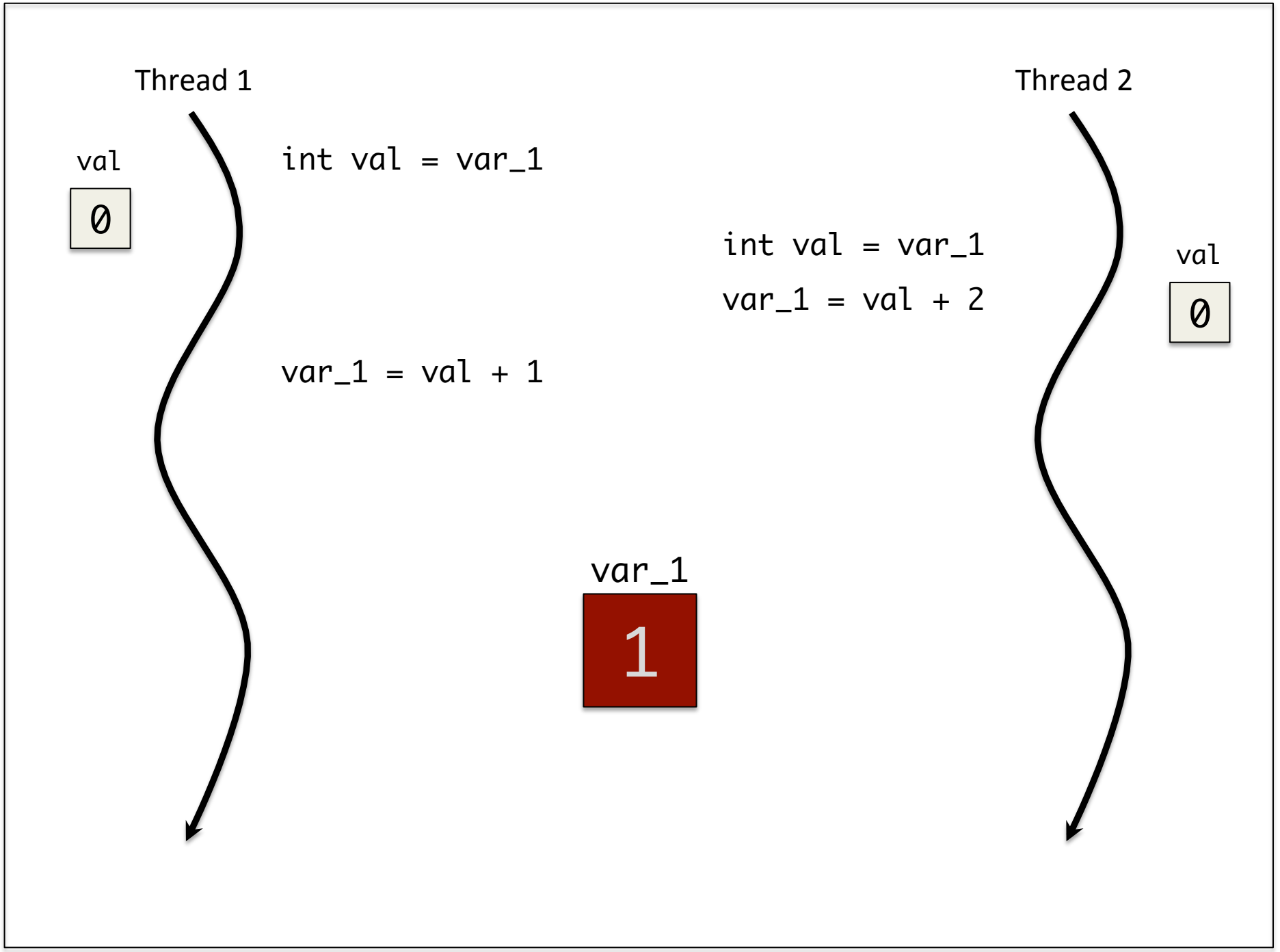

Thread 1
Thread 2
val
0
int val = var_1
int val = var_1
var_1
0

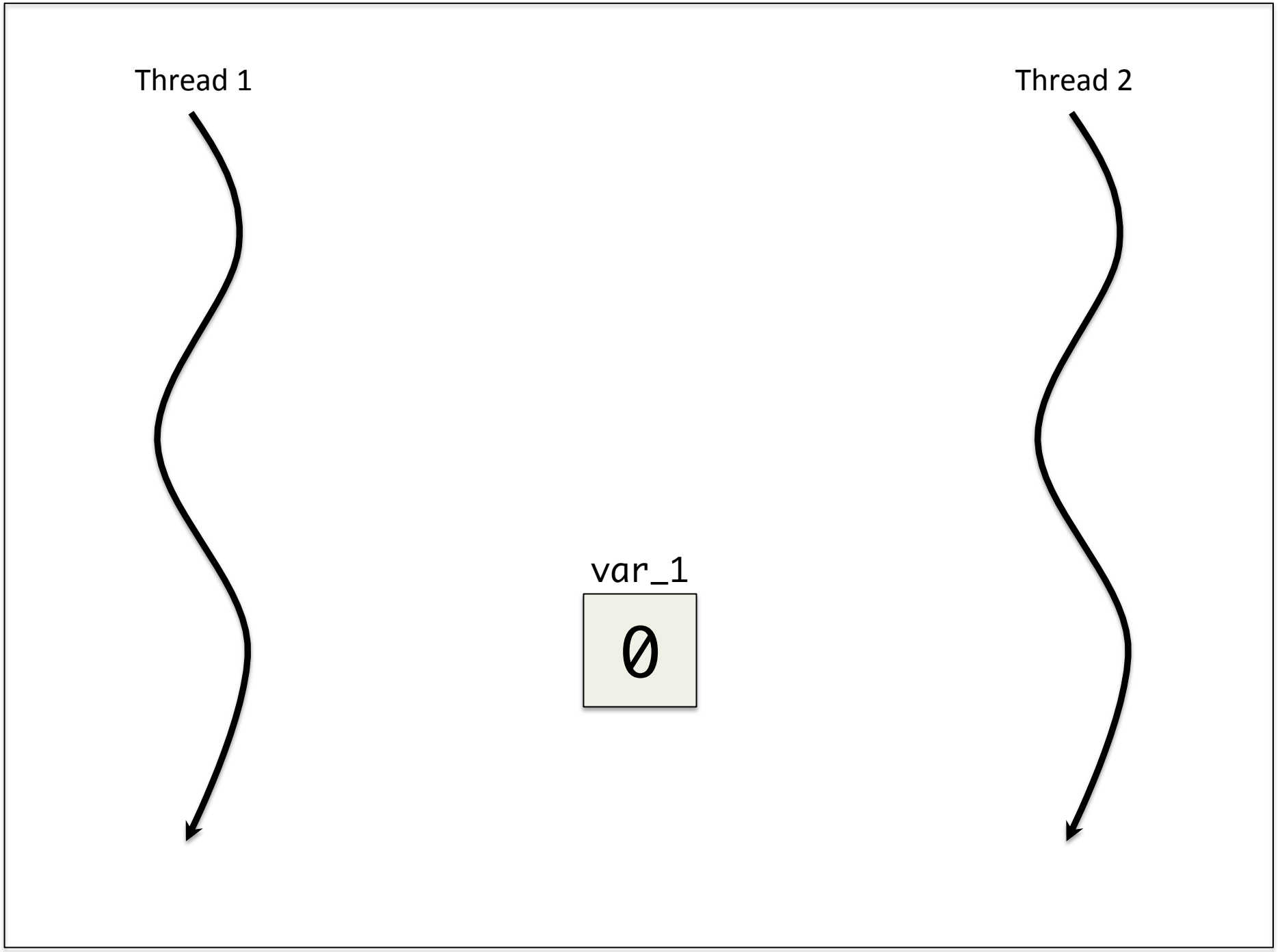

Thread 1
Thread 2
val
0
int val = var_1
int val = var_1
val
0
var_1
0


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


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

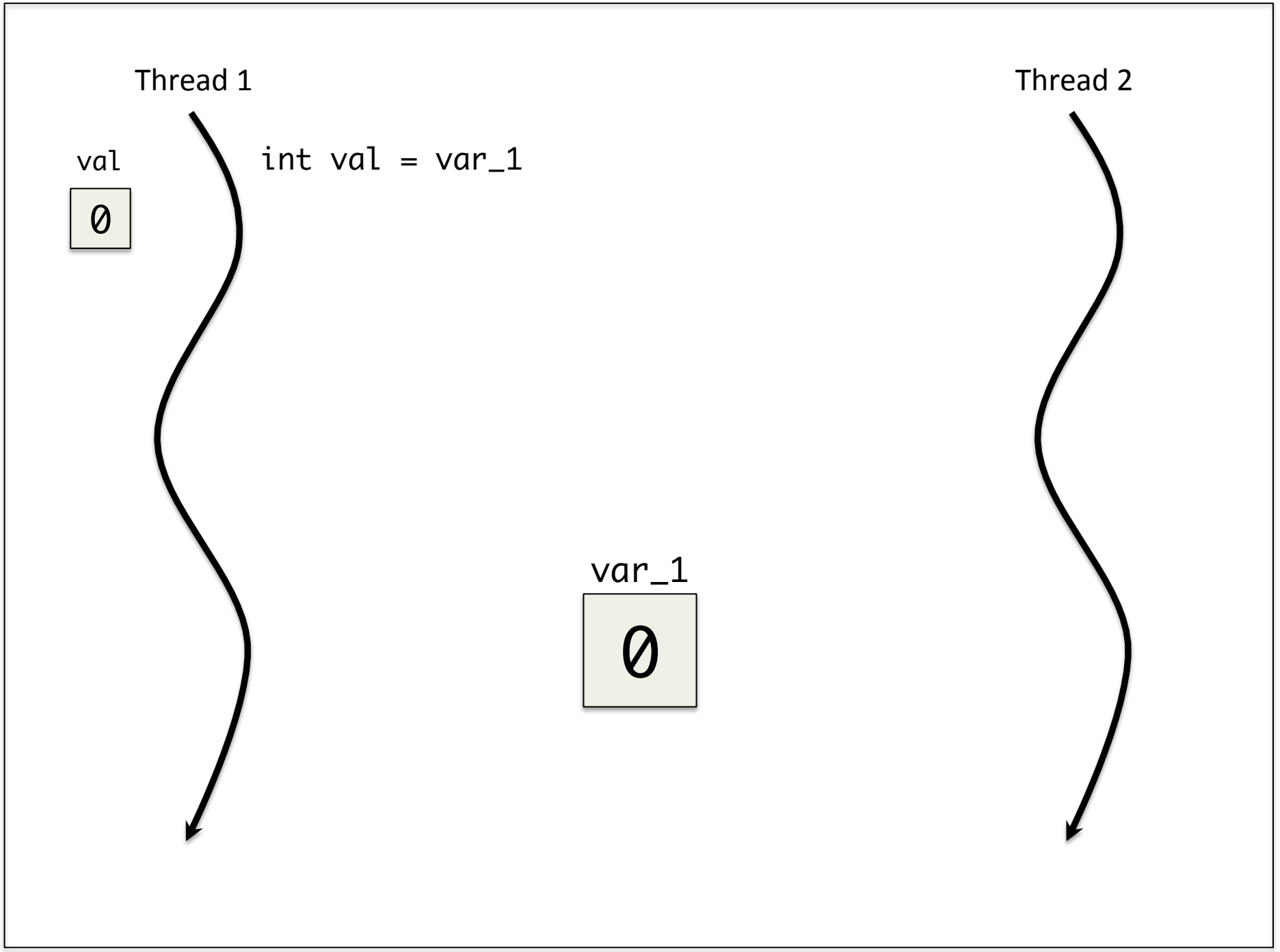

Thread 1
val
0
int val = var_1
var_1 = val + 1
Thread 2
int val = var_1
var_1 = val + 2
val
0
var_1
2

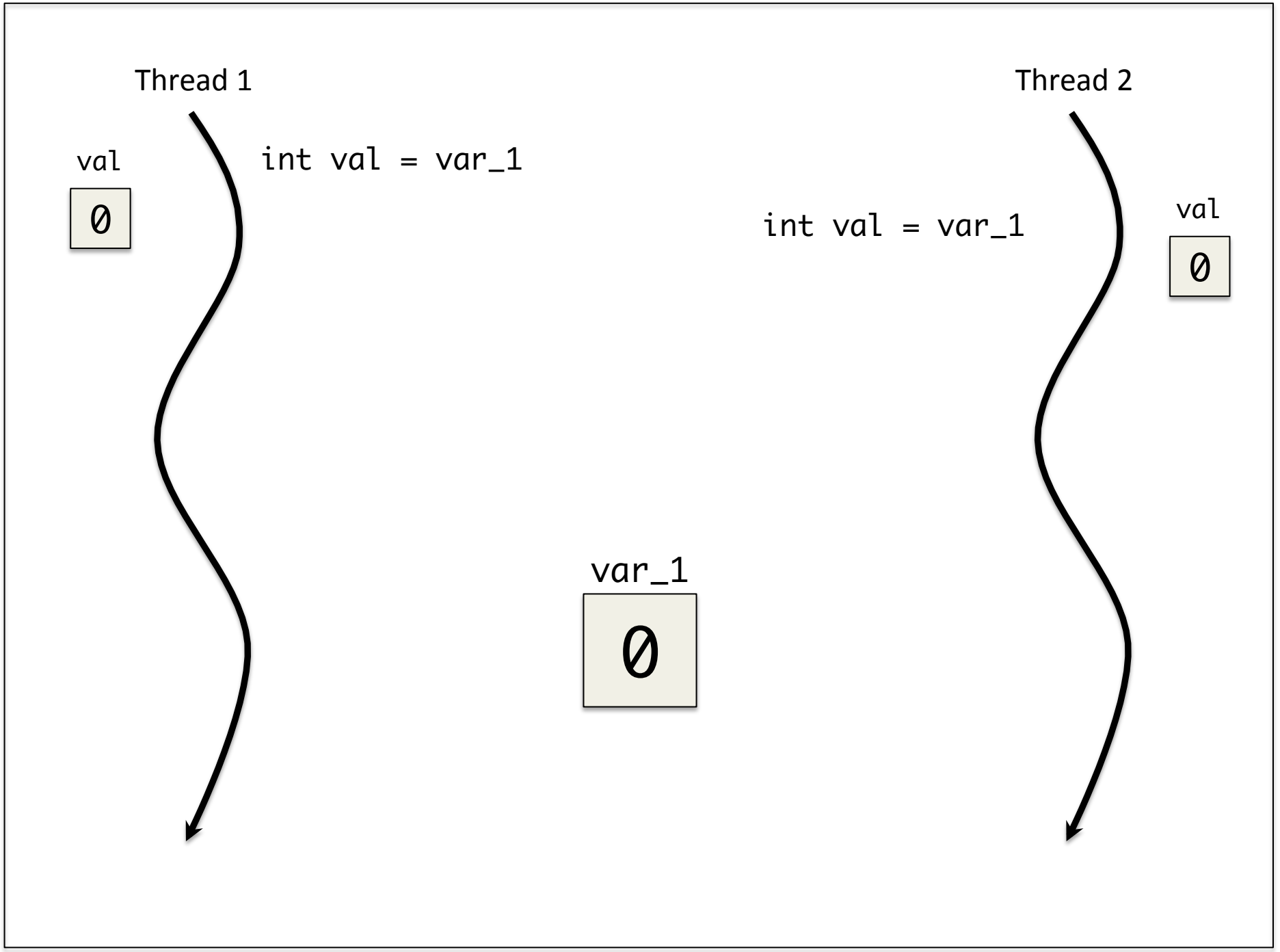

Thread 1
val
0
int val = var_1
var_1 = val + 1
Thread 2
int val = var_1
var_1 = val + 2
val
0
var_1
1

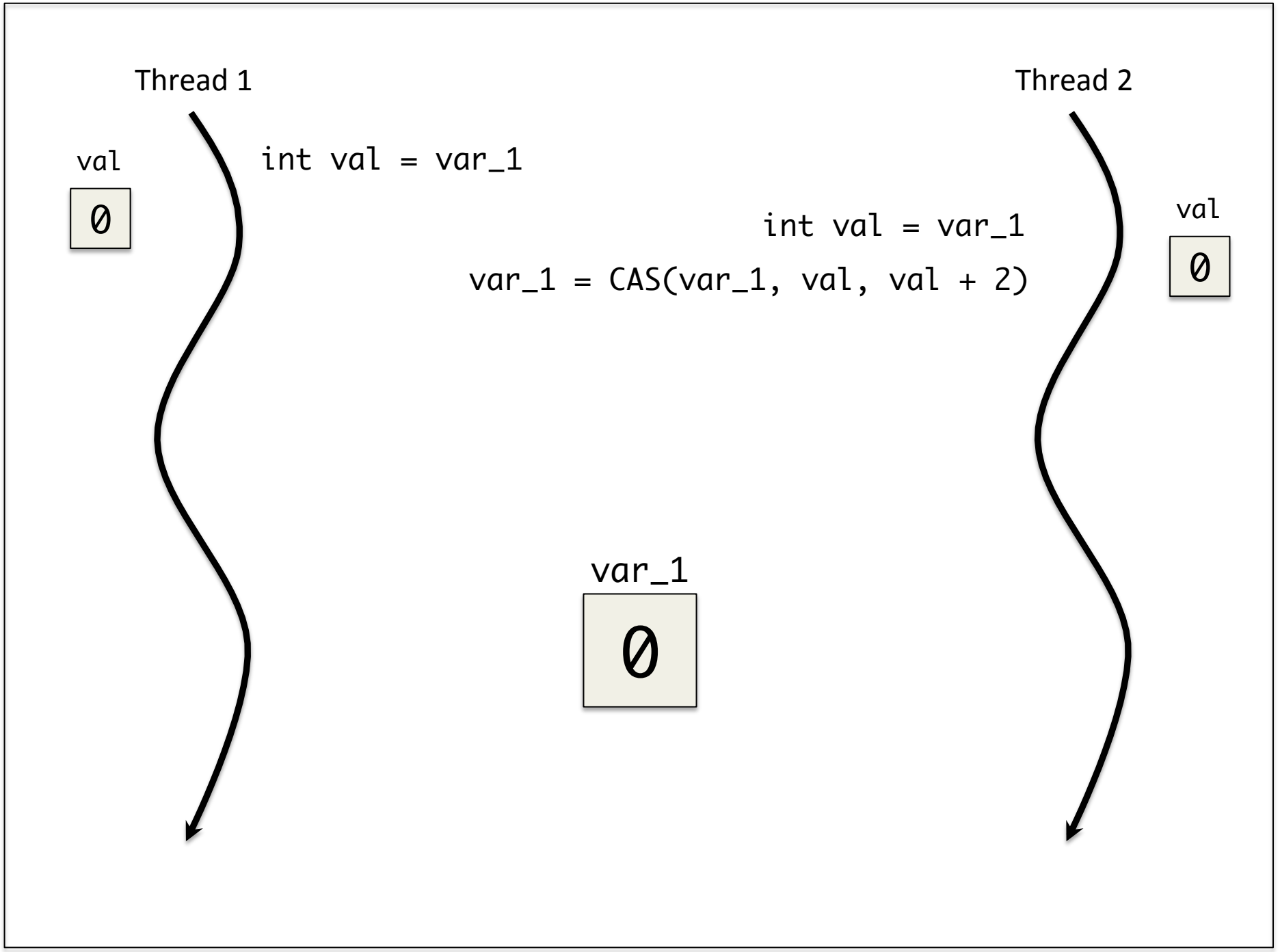

Thread 1
Thread 2
var_1
0

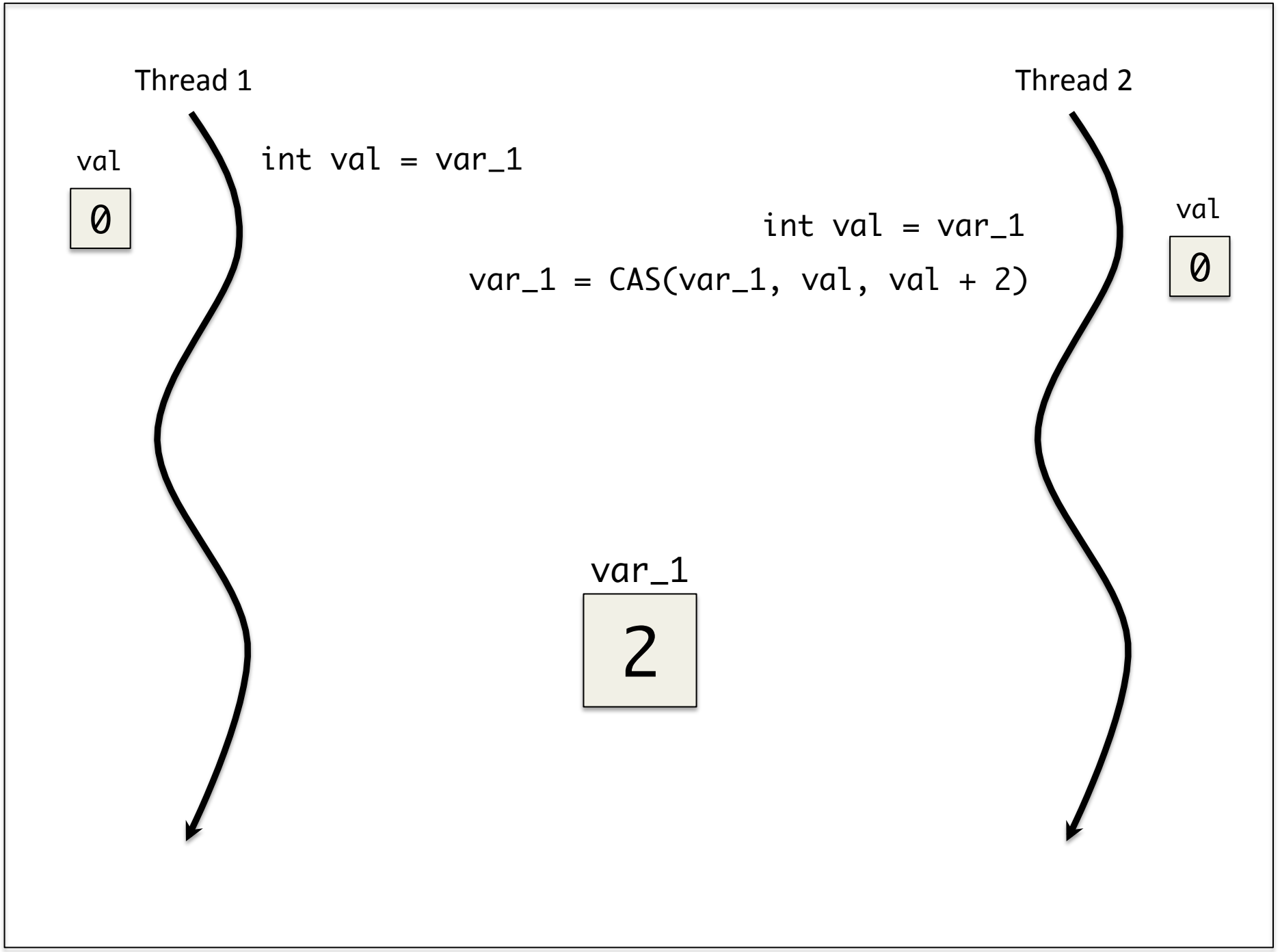
Compare and Swap

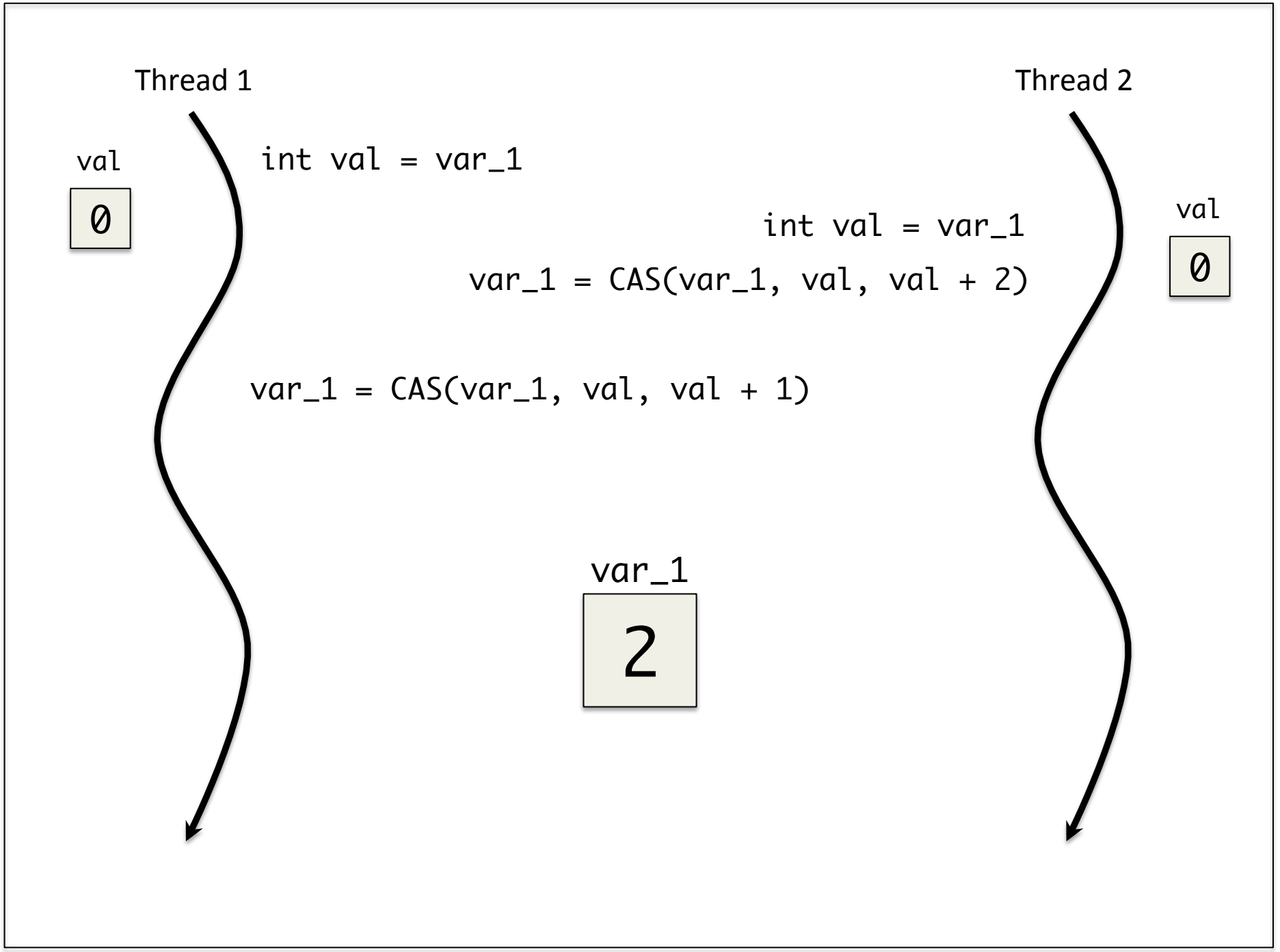
- Atomic instruction to update a value
  - Takes an original value
  - Compares with the current contents of memory
- If the test succeeds, update memory immediately
  - No other thread can pre-empt the update
- If the test fails, return the current value
  - Don't perform the update

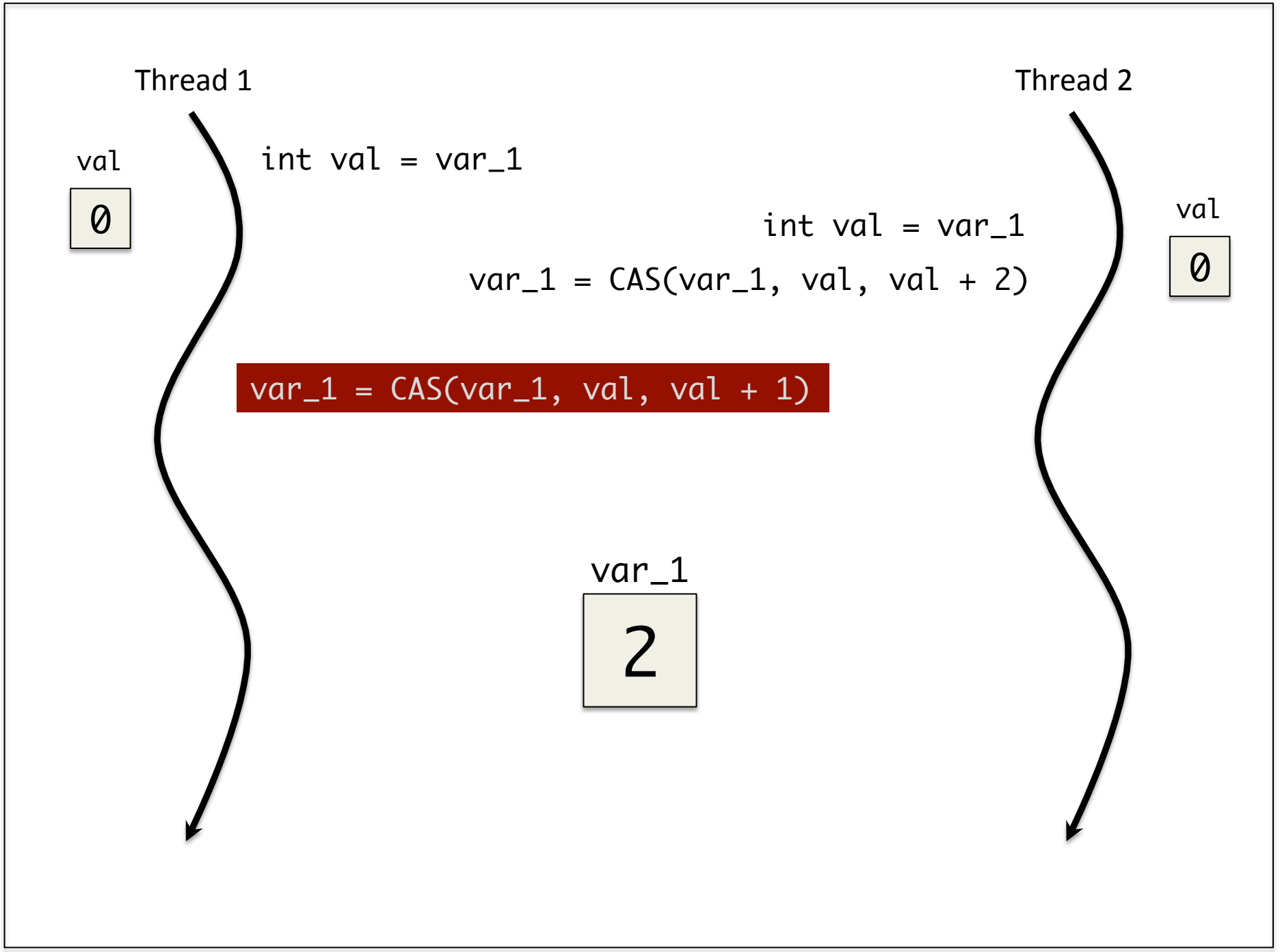

Thread 1
Thread 2
val
0
int val = var_1
var_1
0

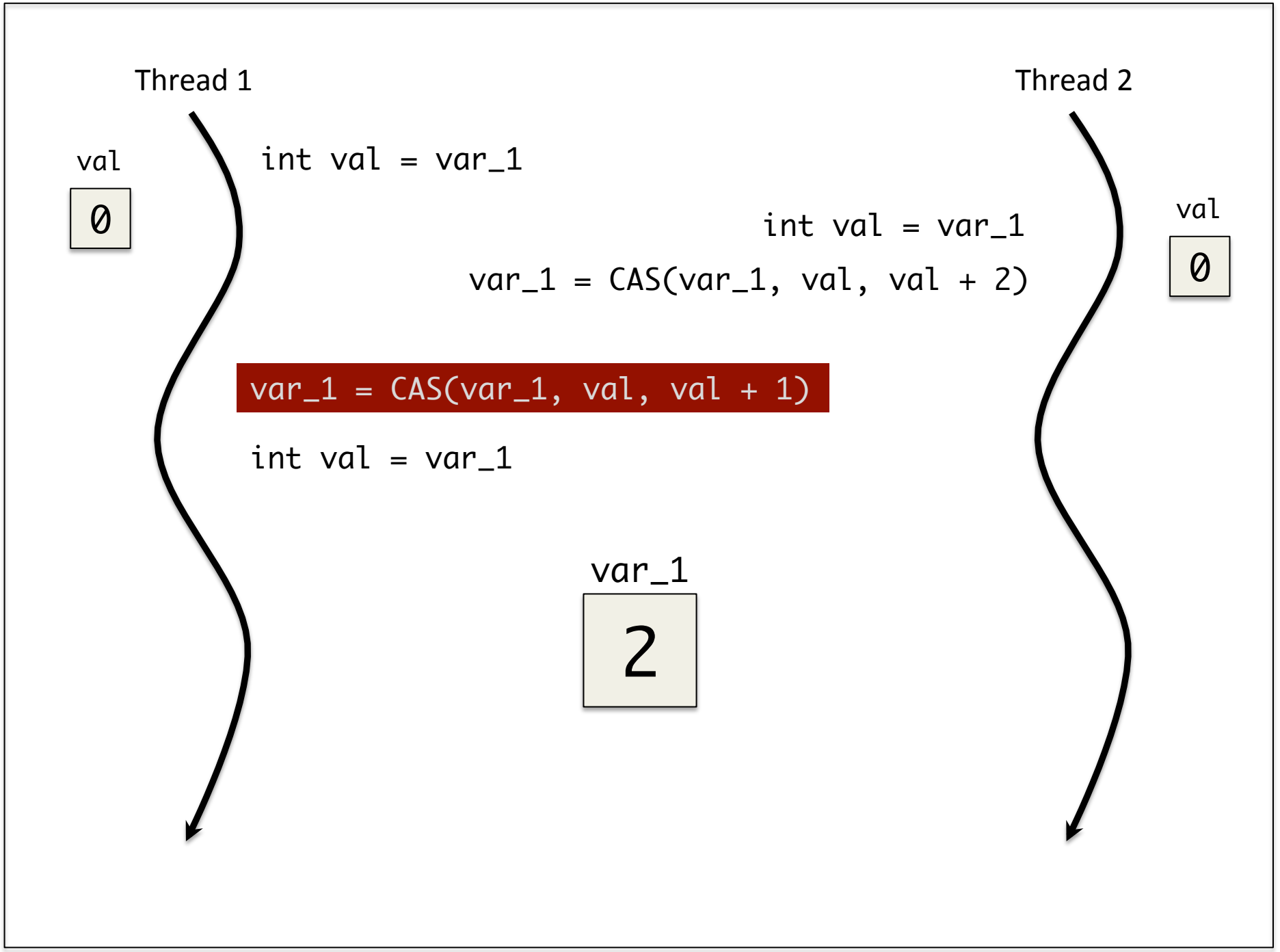

Thread 1
val
0
int val = var_1
Thread 2
int val = var_1
val
0
var_1
0

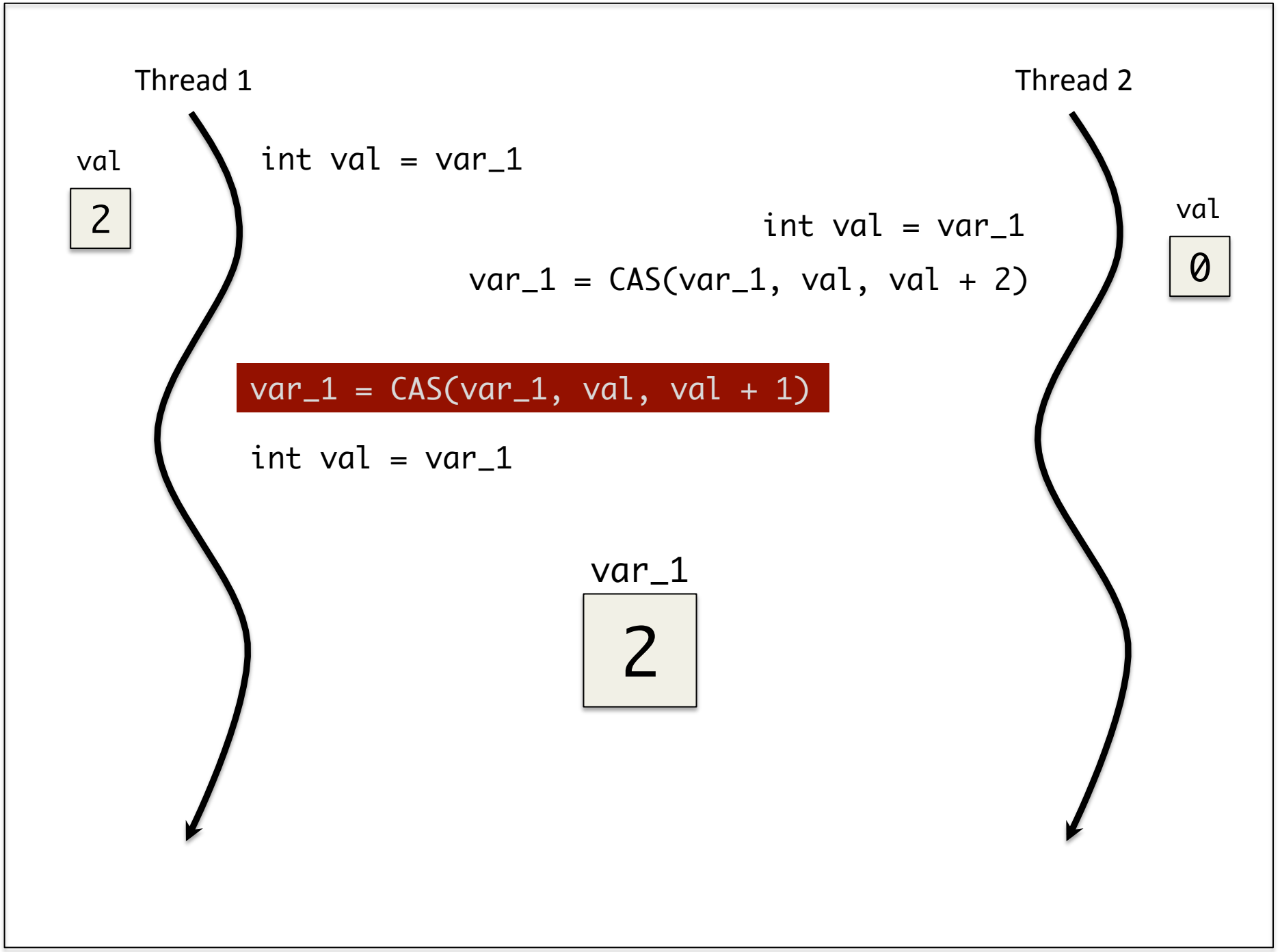

Thread 1
Thread 2
val
0
int val = var_1
int val = var_1
var_1 = CAS(var_1, val, val + 2)
val
0
var_1
0

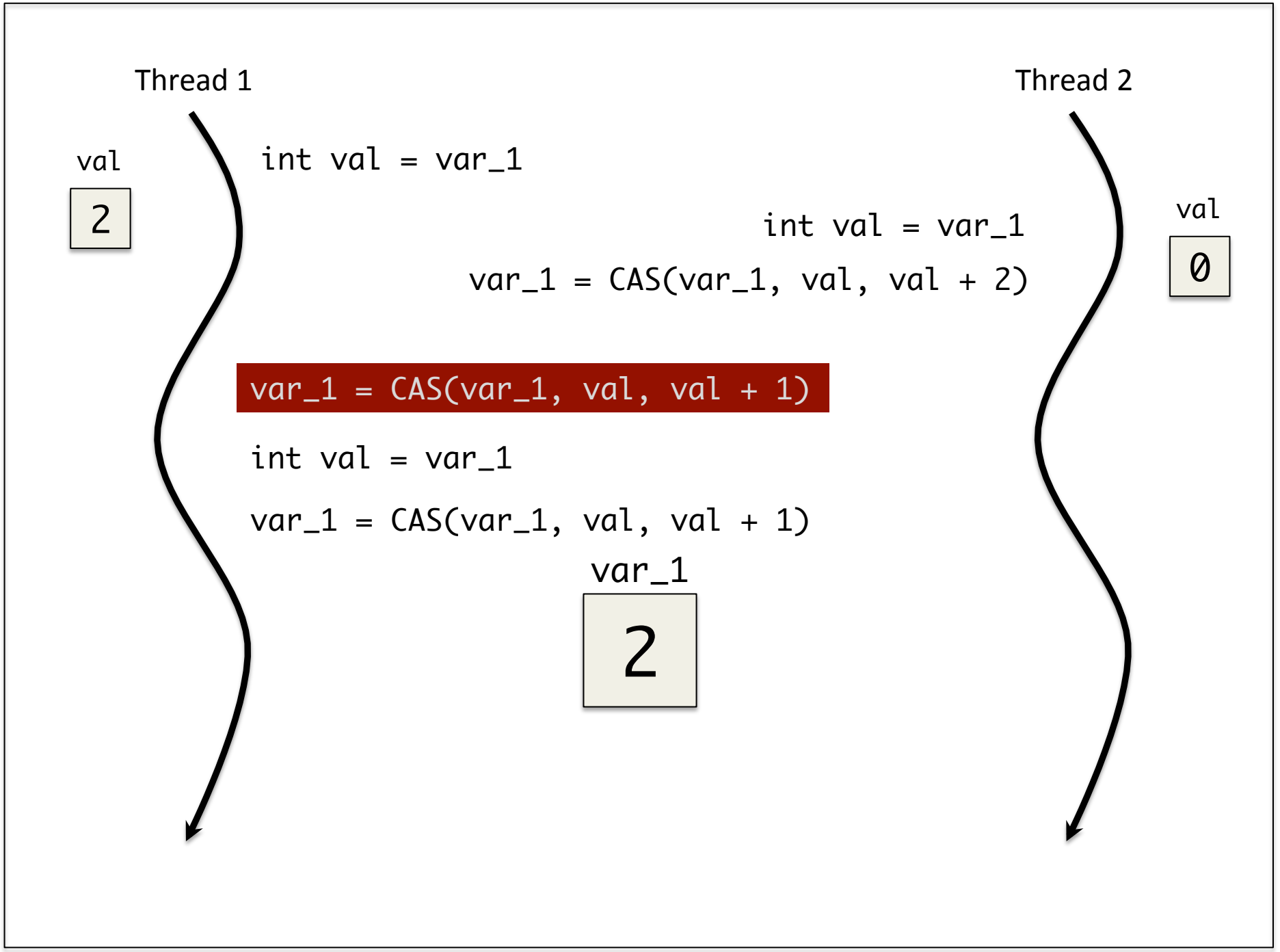

Thread 1
Thread 2
val
0
int val = var_1
int val = var_1
var_1 = CAS(var_1, val, val + 2)
val
0
var_1
2

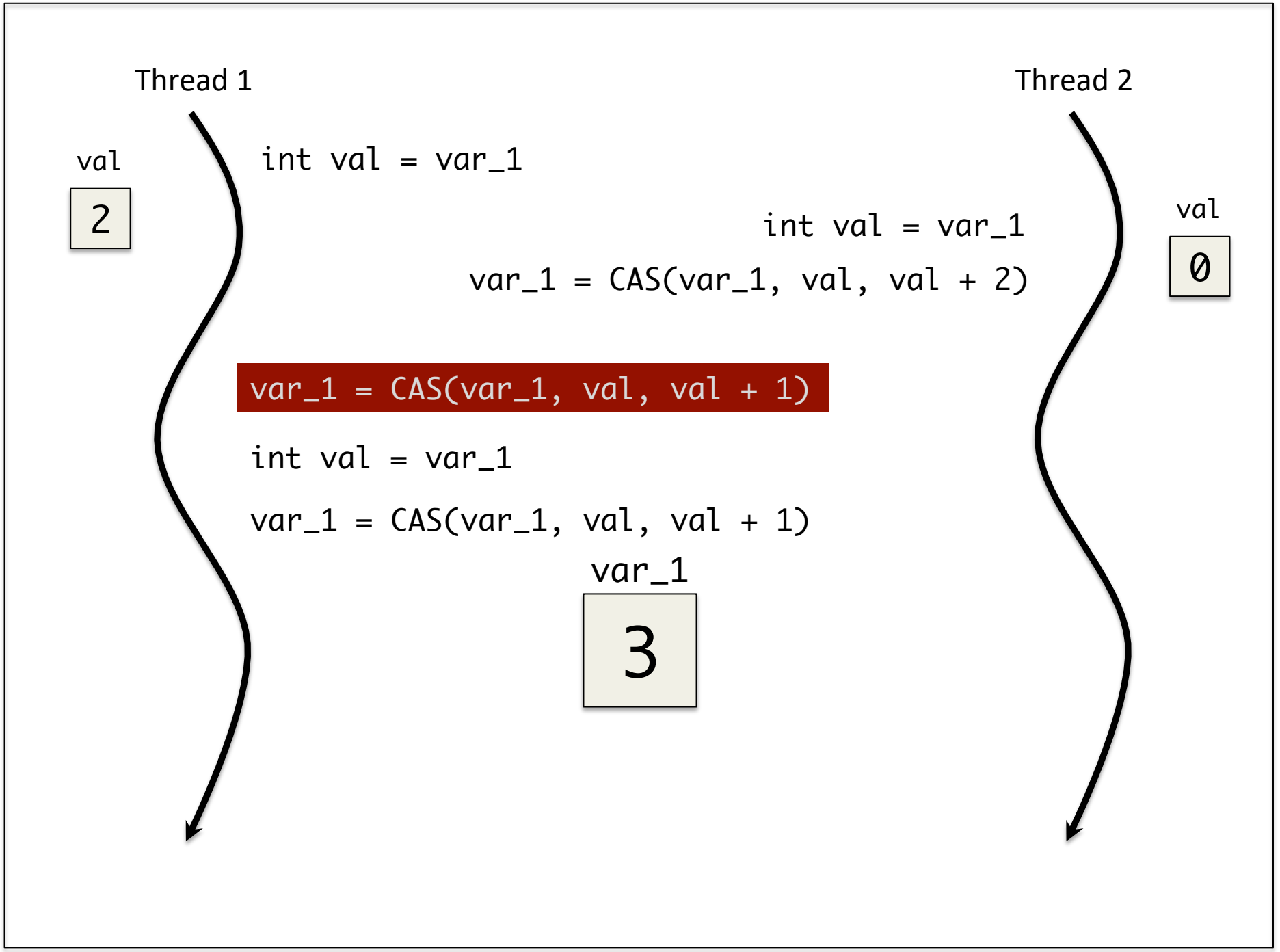

Thread 1
Thread 2
val
0
int val = var_1
int val = var_1
val
0
var_1 = CAS(var_1, val, val + 2)
var_1 = CAS(var_1, val, val + 1)
var_1
2


Thread 1
Thread 2
val
0
int val = var_1
int val = var_1
var_1 = CAS(var_1, val, val + 2)
val
0
var_1 = CAS(var_1, val, val + 1)
var_1
2


Thread 1
Thread 2
val
0
int val = var_1
int val = var_1
var_1 = CAS(var_1, val, val + 2)
val
0
var_1 = CAS(var_1, val, val + 1)
int val = var_1
var_1
2


Thread 1
Thread 2
val
2
int val = var_1
int val = var_1
val
0
var_1 = CAS(var_1, val, val + 2)
var_1 = CAS(var_1, val, val + 1)
int val = var_1
var_1
2


Thread 1
Thread 2
val
2
int val = var_1
int val = var_1
var_1 = CAS(var_1, val, val + 2)
val
0
var_1 = CAS(var_1, val, val + 1)
int val = var_1
var_1 = CAS(var_1, val, val + 1)
var_1
2


Thread 1
Thread 2
val
2
int val = var_1
int val = var_1
var_1 = CAS(var_1, val, val + 2)
val
0
var_1 = CAS(var_1, val, val + 1)
int val = var_1
var_1 = CAS(var_1, val, val + 1)
var_1
3

Operating System Threading

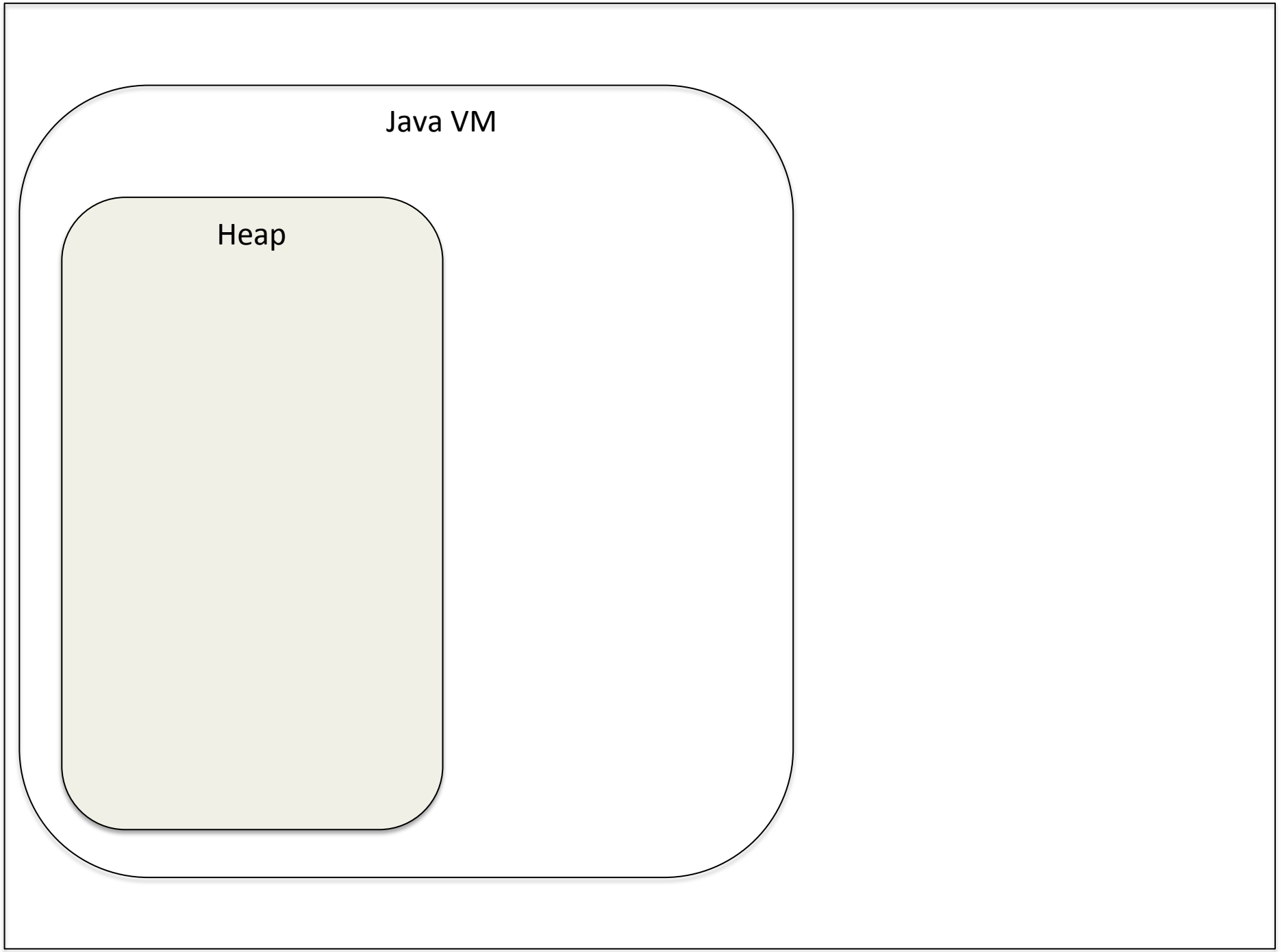
- Various native threading libraries
  - `pthreads` available on most platforms today
  - Windows, Solaris, OS X, iOS have native packages
- Some standard features
  - Atomic operations
  - Mutual exclusion locks
  - Semaphores
  - Thread-local storage

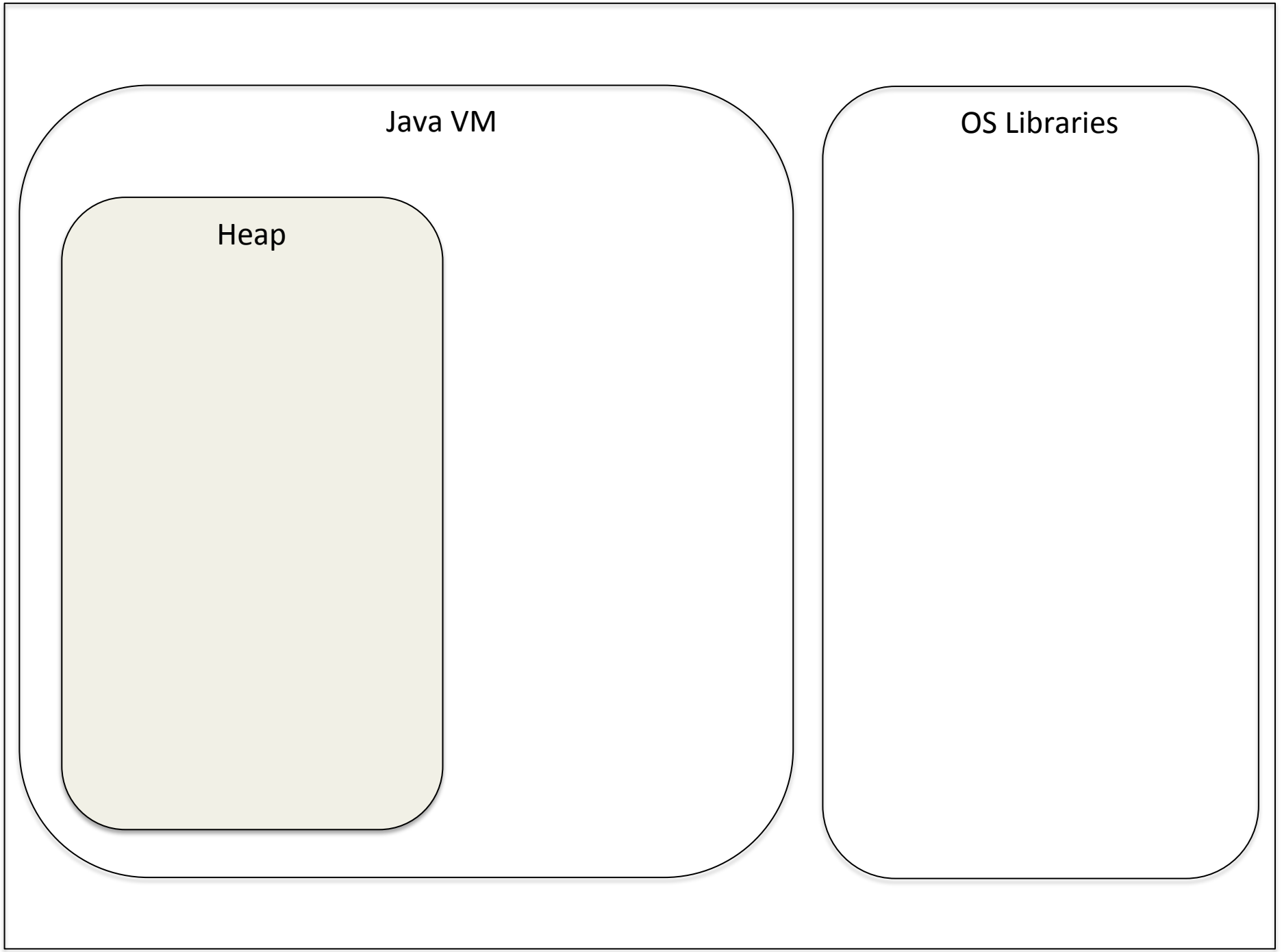
Language-Level Threads

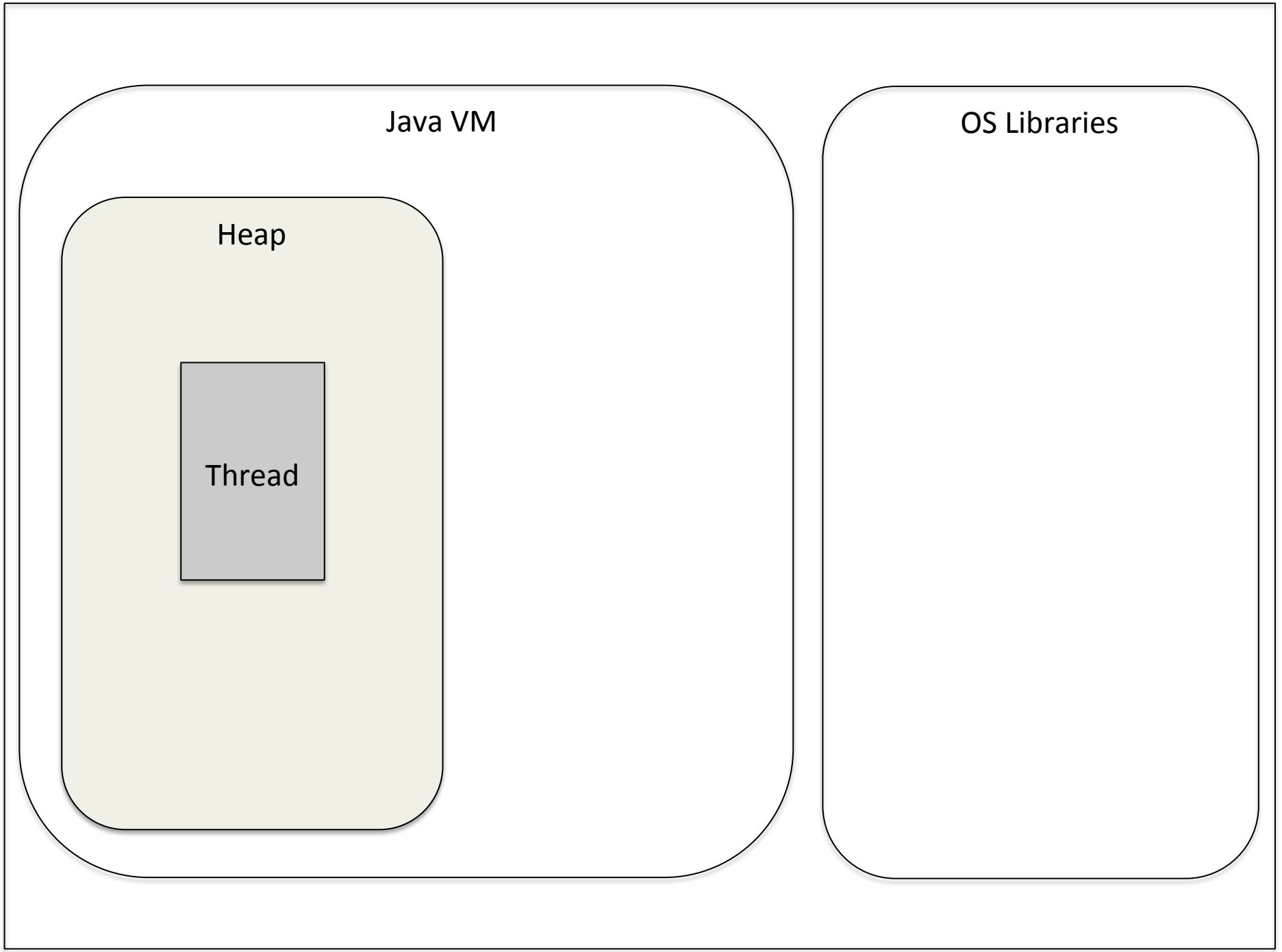
- Higher level abstraction than OS-level
  - Boundary can be fuzzy
- Describes concurrency abstraction
  - More concerned with meaning than mechanism
- Degree of abstraction varies by language
  - Fine control harder to manage but more powerful
  - Very abstract is easy to work with, but limited

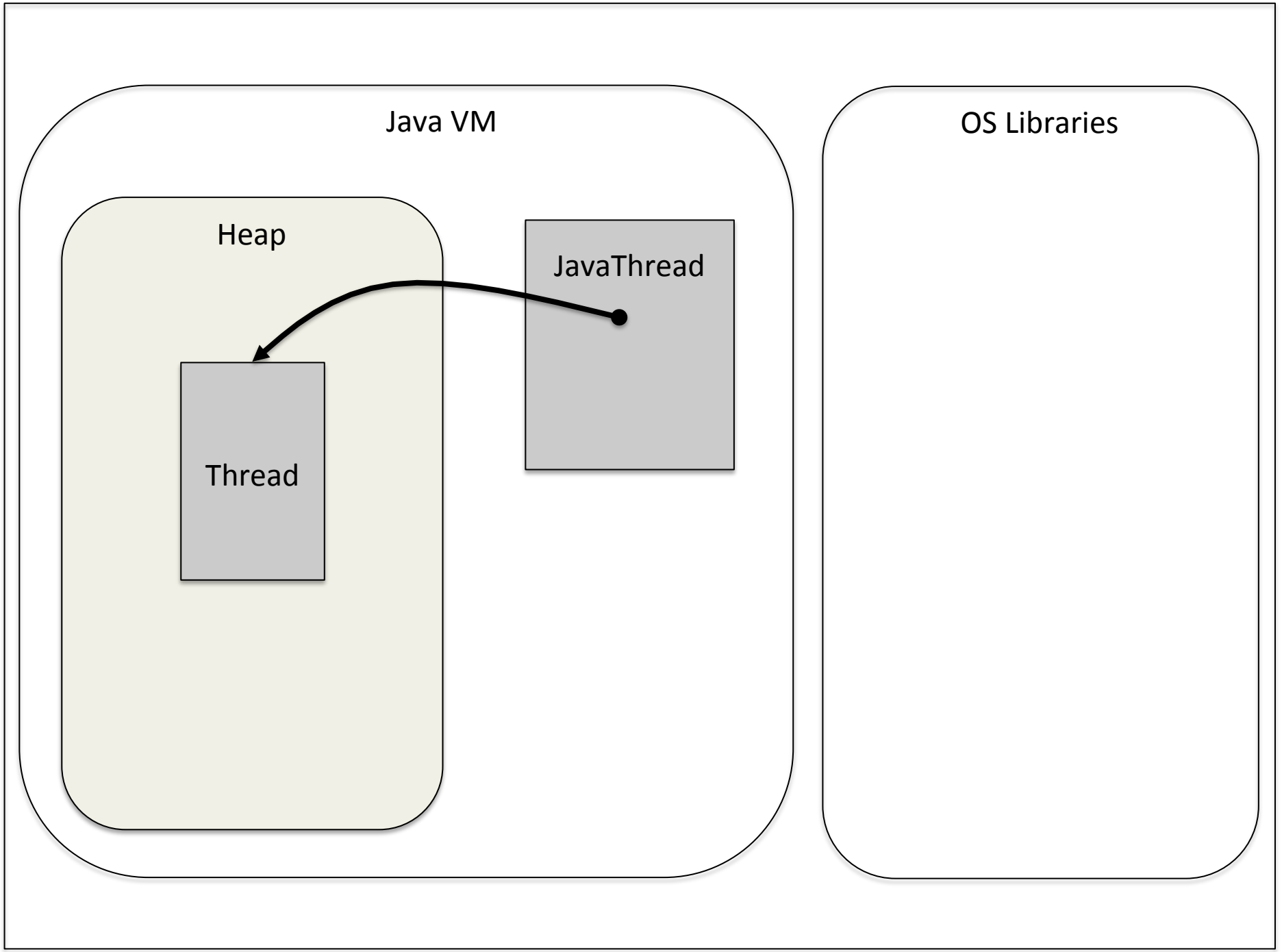
Java Threading

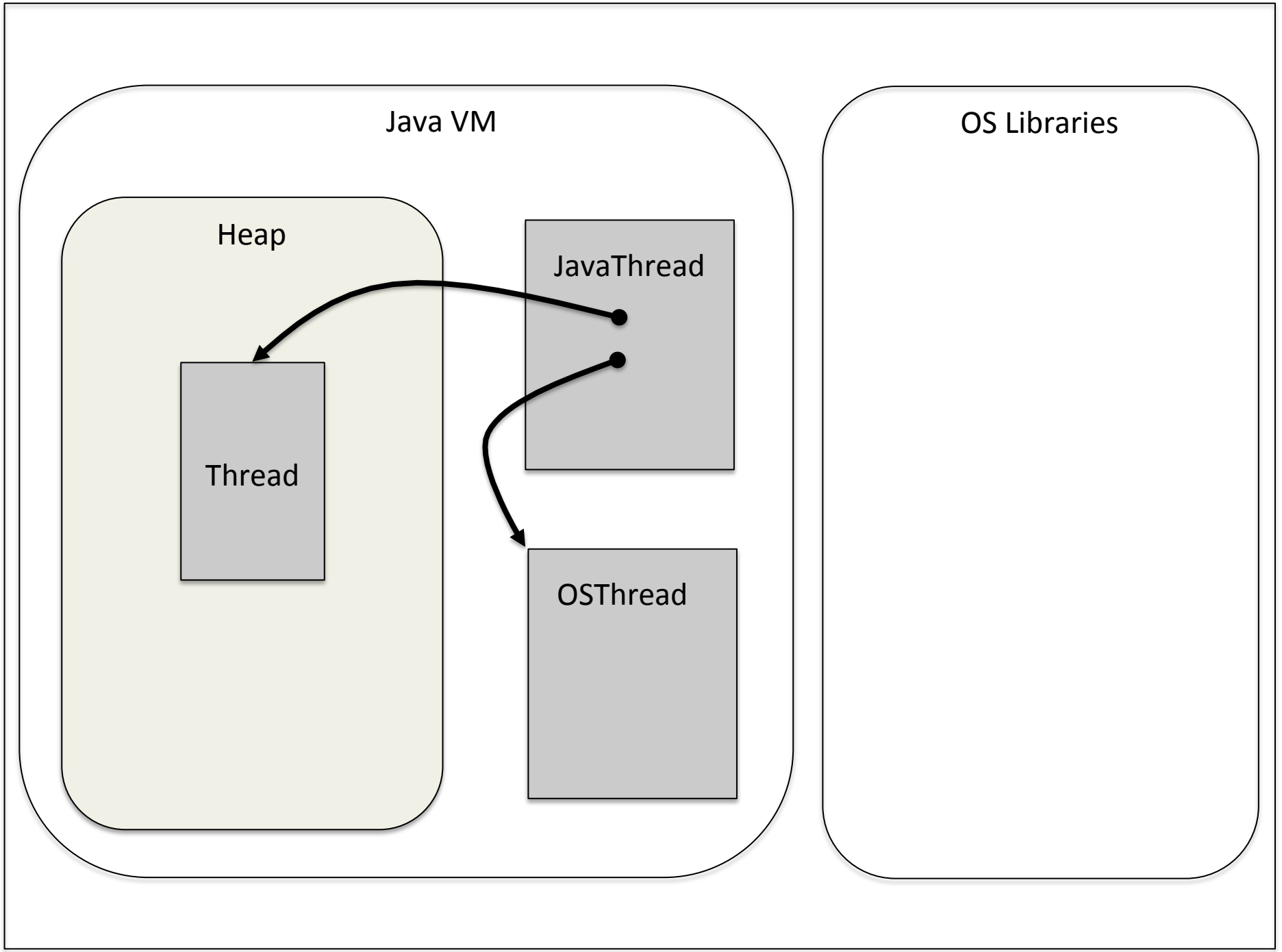
- Basic model
  - `Thread` class
  - `Runnable` and `Callable` interfaces
- `Thread` created like any other Java object
  - Call `start` method to launch
- Various other thread operations
  - `join`, `yield`, `sleep`
  - All described in the language spec

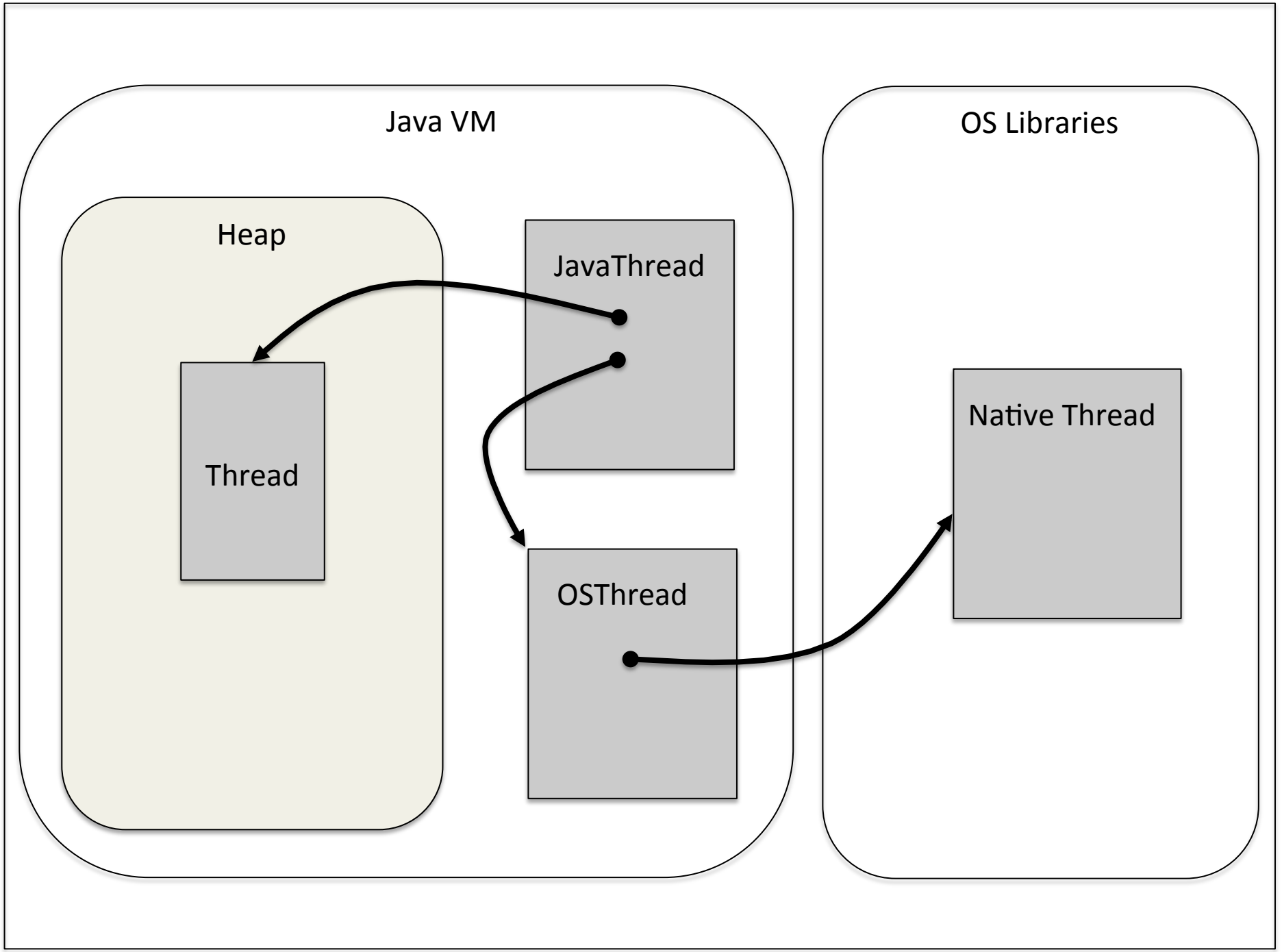

Java VM
Heap


Java VM
Heap
OS Libraries


Java VM
Heap
Thread
OS Libraries


Java VM
OS Libraries
Heap
JavaThread
Thread


Java VM
OS Libraries
Heap
JavaThread
Thread
OSThread


Java VM
OS Libraries
Heap
JavaThread
Thread
Native Thread
OSThread

Scheduling

- Scheduling algorithm is implementation specific
  - Depends greatly on the platform
  - Goal is to present a unified model
- Scheduling can be delegated to the operating system
  - Hotspot maps Java threads directly to OS threads
- VM may alternatively manage threads itself
  - More work for the VM implementer
  - Finer control of scheduling

Green Threads

- Some systems don't support multithreading
  - Mostly embedded devices these days
- VM can use a single OS thread
  - Manually schedule VM threads
- Generally uses straight priority scheduling
  - Highest priority thread runs to completion
  - Thread can be interrupted by higher priority thread
  - Possibility of starvation
- Can't rely on priorities for mutual exclusion

Native Threads

- VM maps `JavaThread` to `OSThread` directly
  - Uses OS-level priority mechanism
- OS has better knowledge of the platform
  - Can optimize scheduling around system events
  - May predict long-latency events
- VM has no control over thread switching

Time Slicing

- OS multiplexes threads to CPU cores
  - There are normally more threads than cores
- Scheduler uses time slicing to share CPU
  - Thread runs for a defined quantum of time
  - Scheduling decision made at the end of that time
  - Handles threads with equal priorities
- More time slicing, worse cache behavior

Hybrid Approach

- VM can map OS to Java threads as one-to-many
  - Run green thread scheduling on each OS thread
- Can group related threads on an OS thread
  - Application thread on one, VM on another
- Green threads can optionally use time slicing
  - Implementation details trickier
  - May have to account for OS time slicing as well

Safepointing

- Time slicing can cause problems with safepointing
  - All non-native threads need to be at a safepoint
  - Can't guarantee when threads were preempted
- Adds an additional step to safepointing
  - Let a thread run to a safepoint
  - Set it as suspended and yield the processor
  - Next highest priority thread takes over
- Need to restore suspended status afterwards
  - Track thread's original status