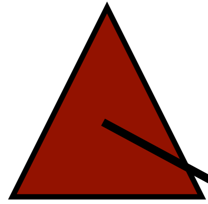
Concurrent Paradigms

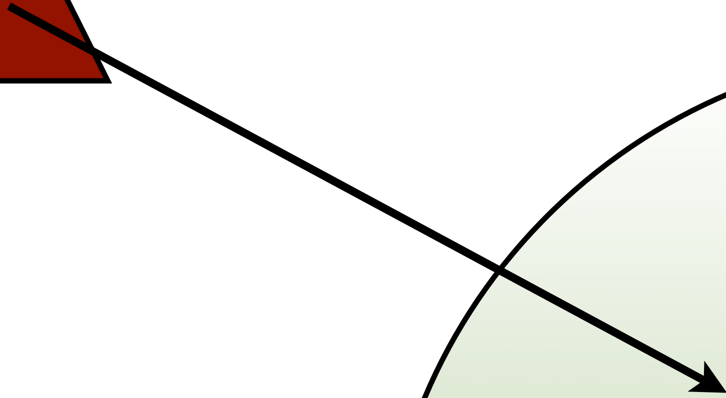
Jorge Ortiz
CS94SI

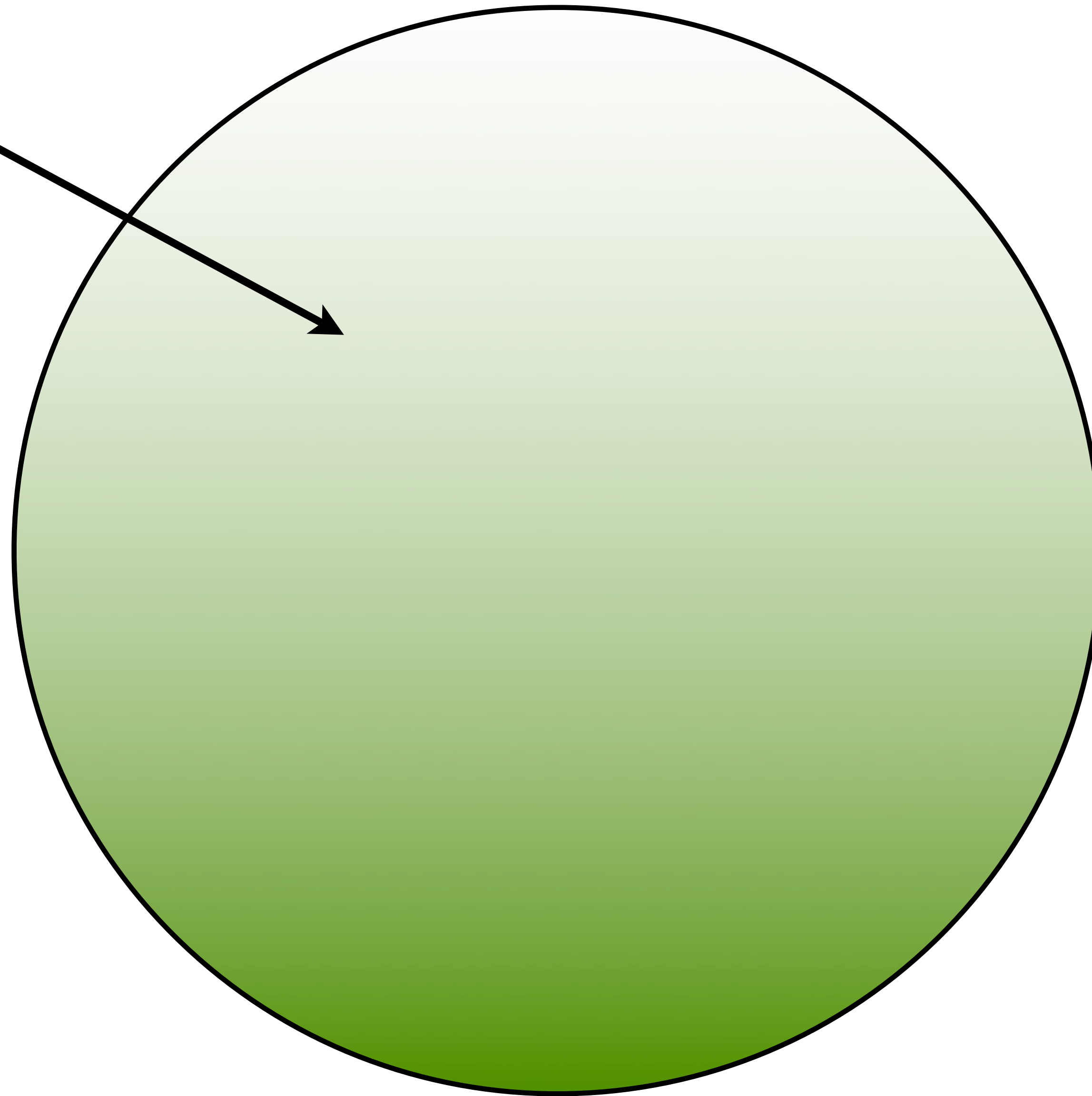
Concurrency is hard

Paradigm:
Shared Memory & Locks

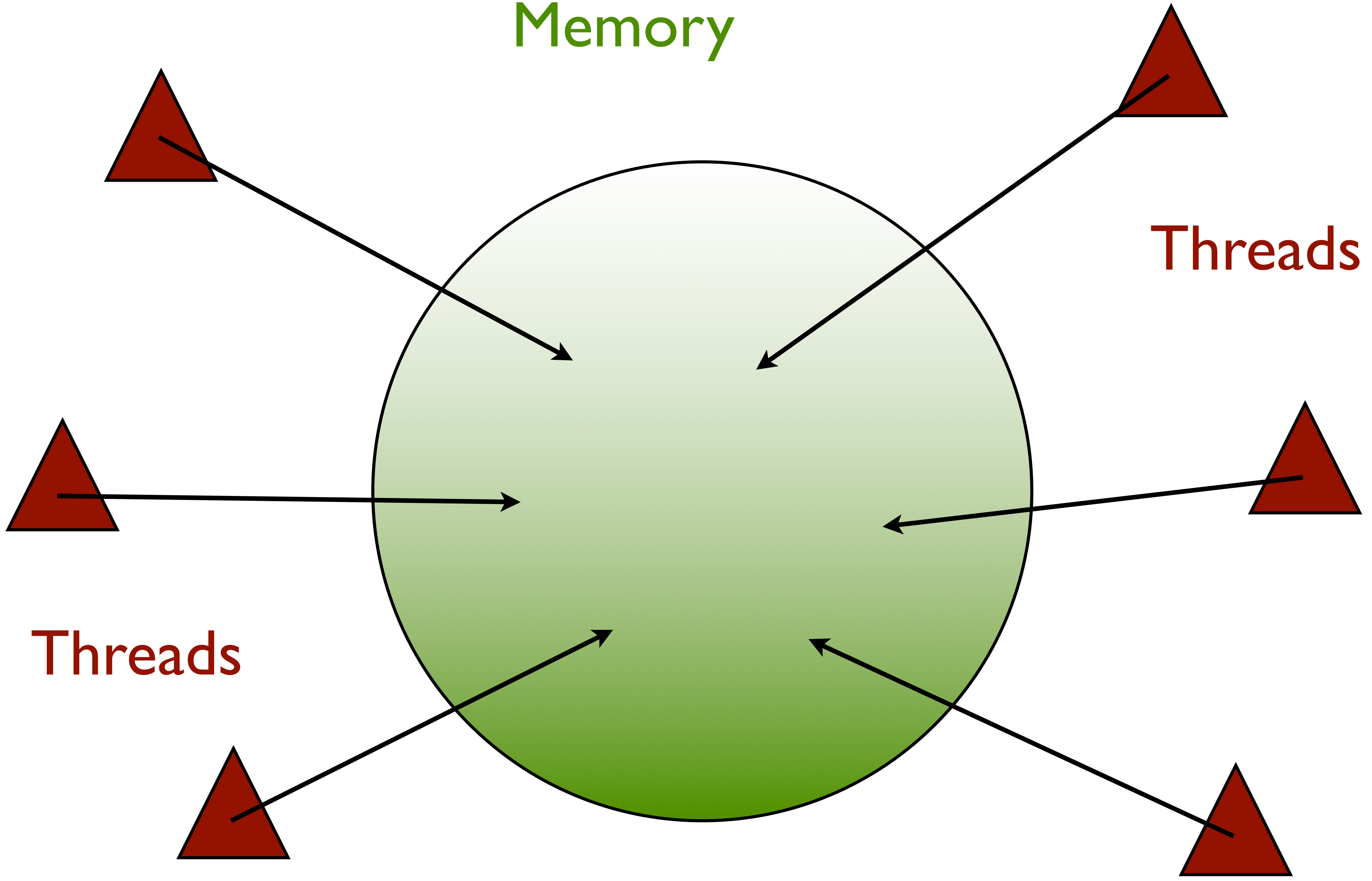
Memory





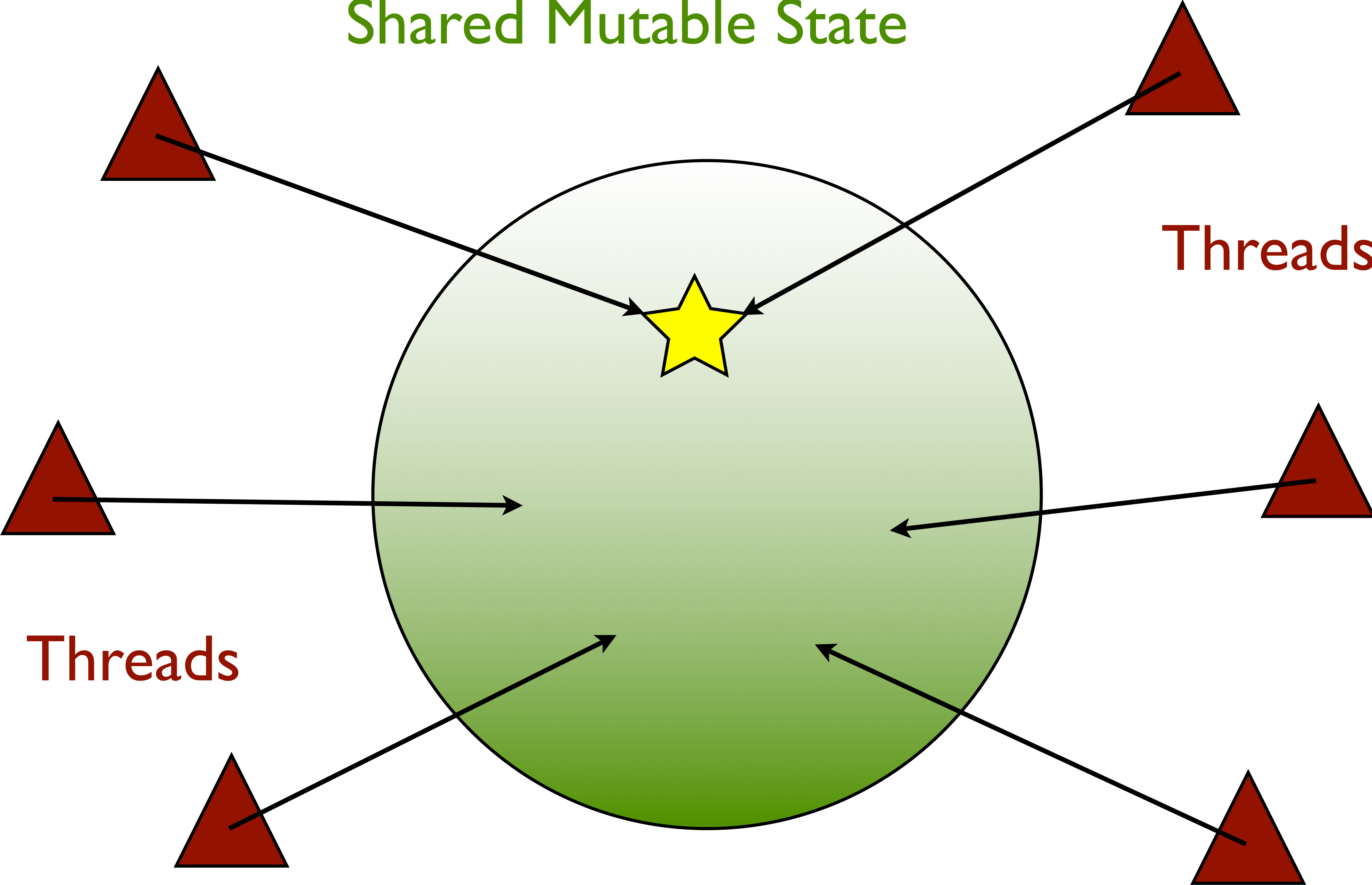


Execution Thread

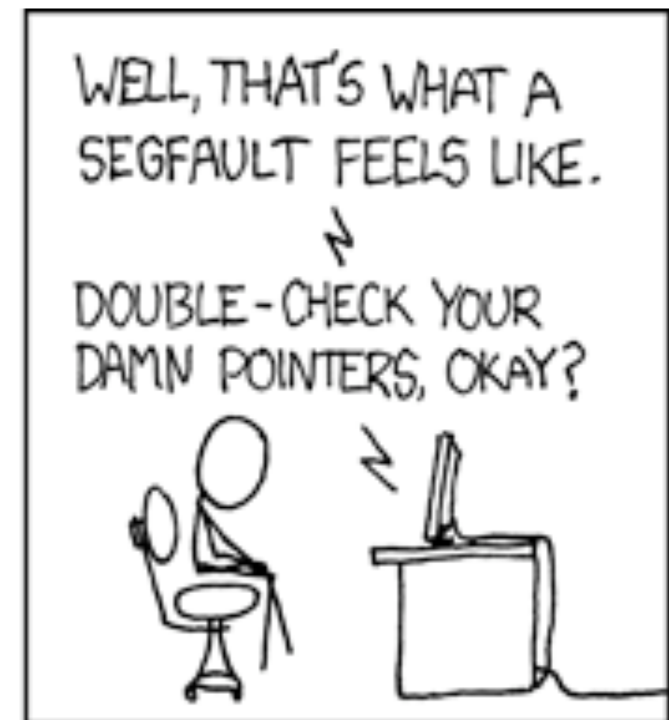

Memory
Threads
Threads

Problems?

Problem #1
Race conditions


Shared Mutable State
Threads
Threads
Race condition

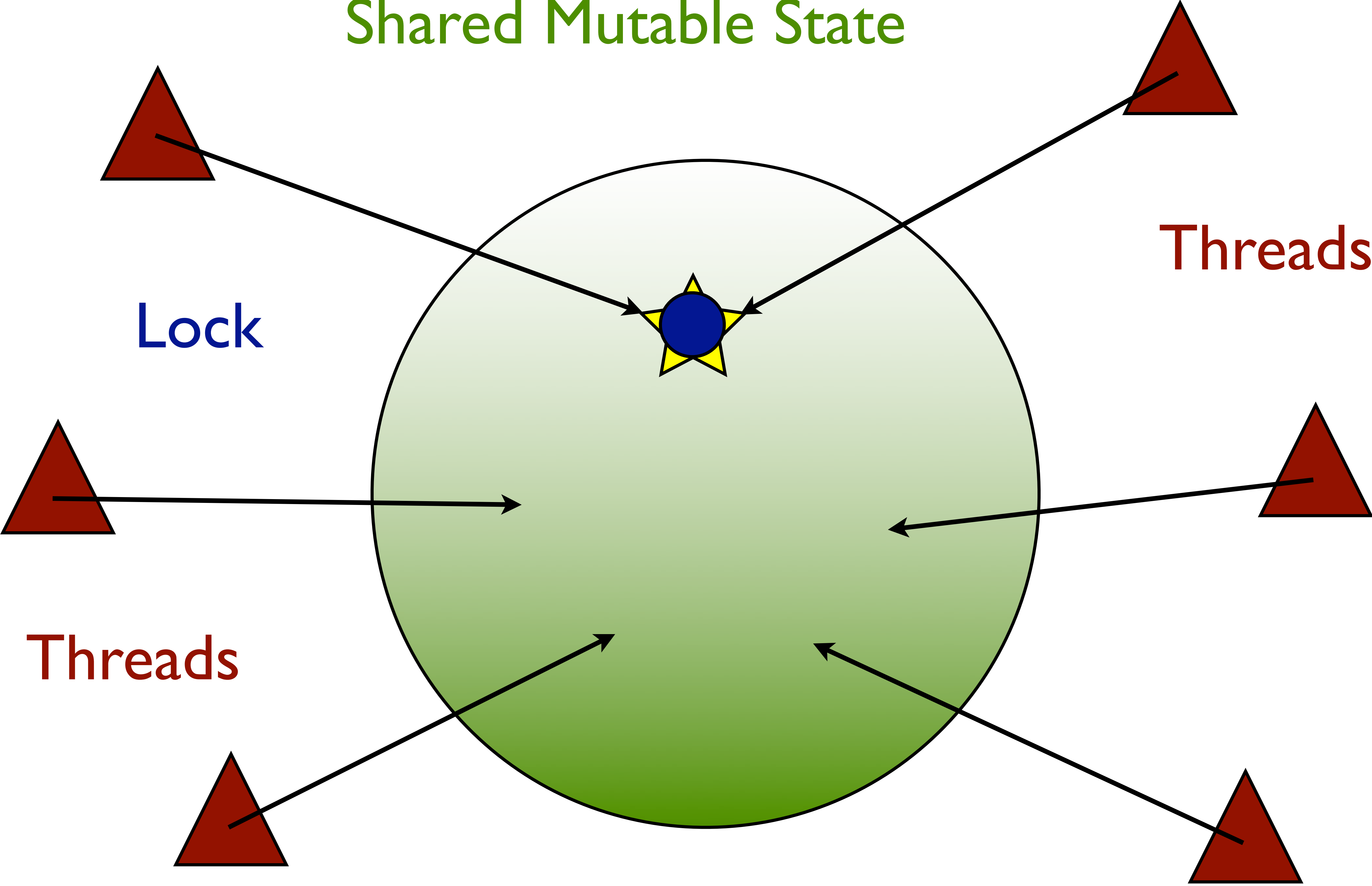
Segfault

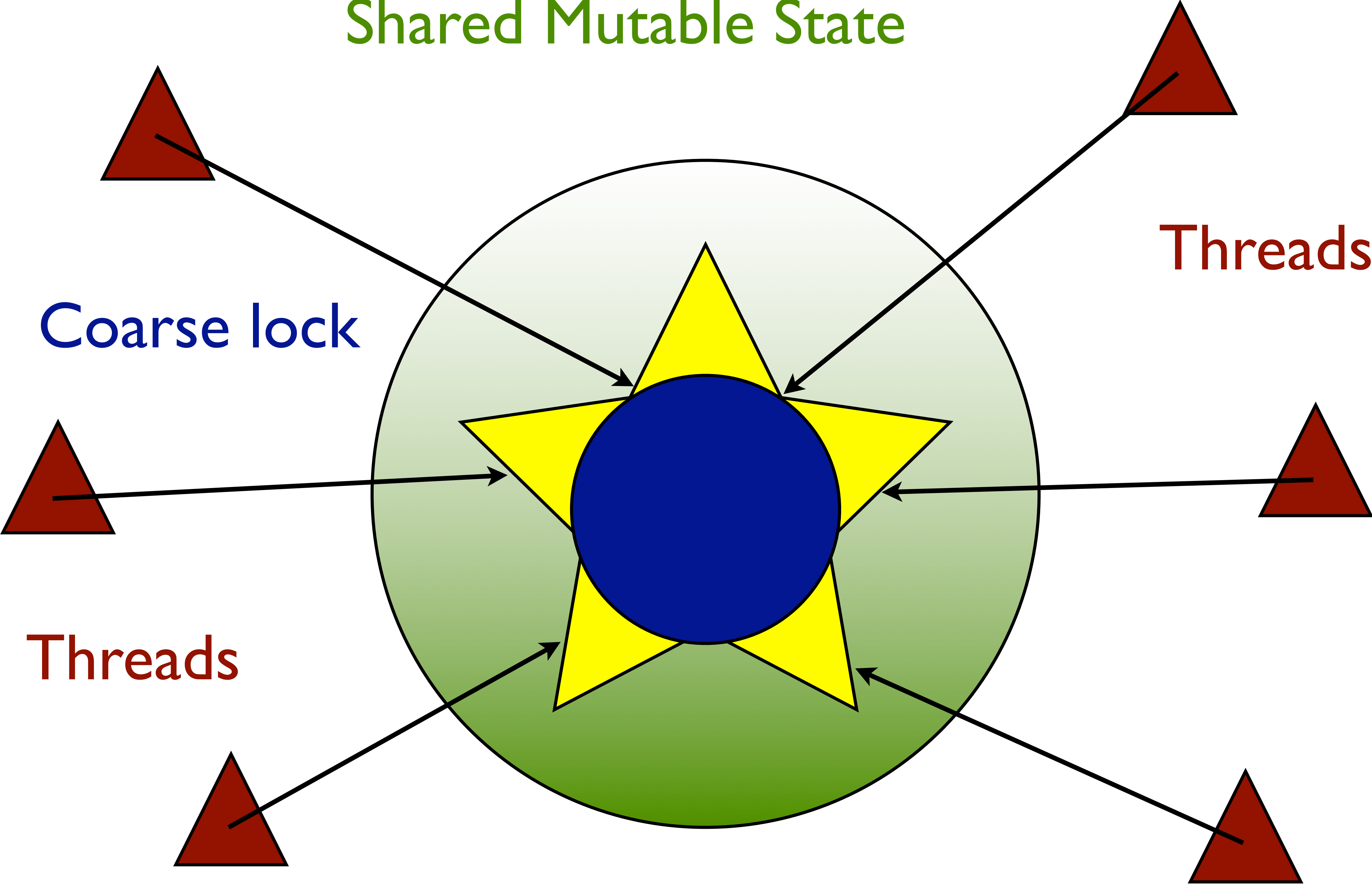


Segfault

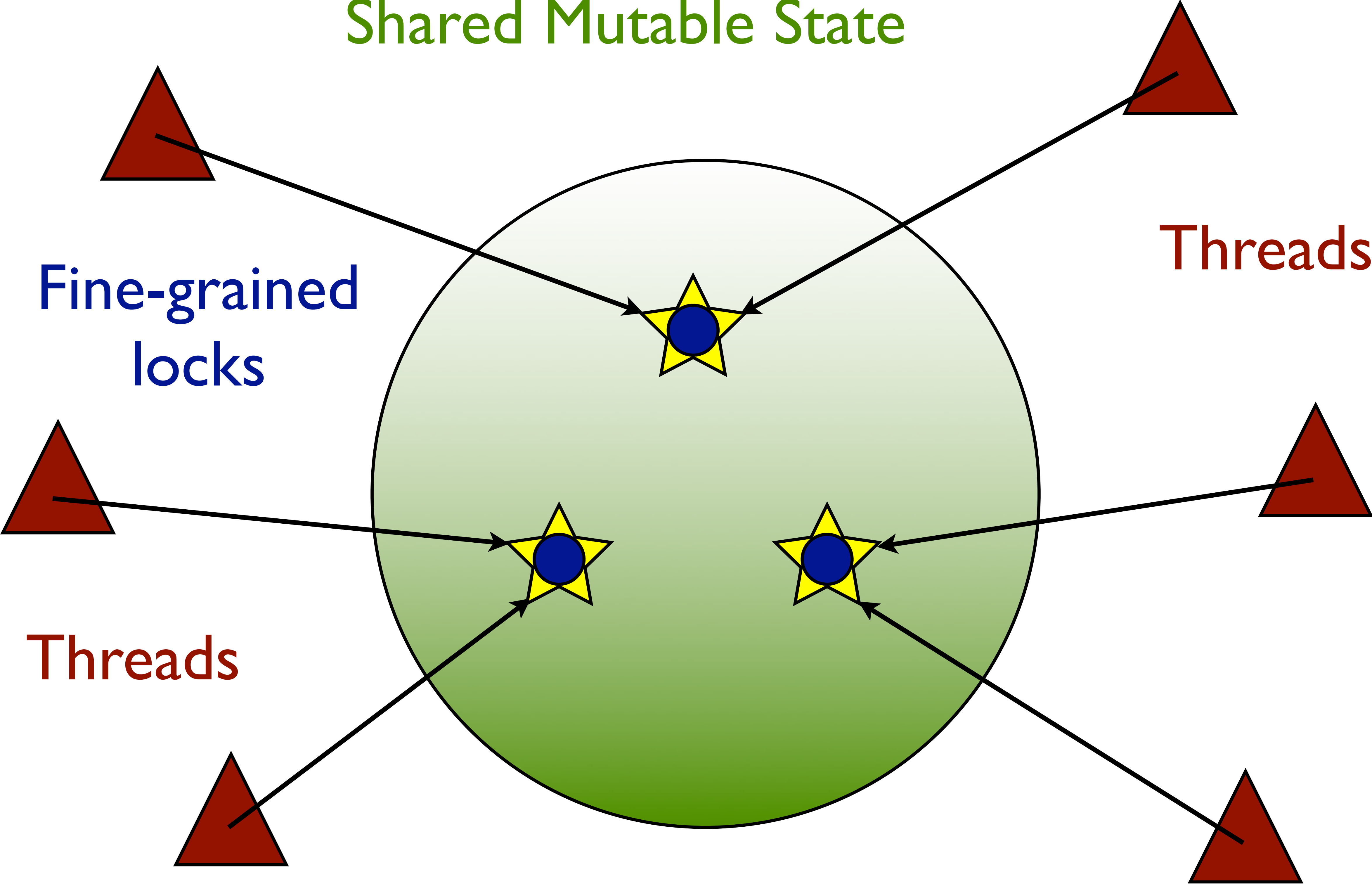
Checking whether build environment is sane ... build environment is grinning and holding a spatula. Guess not.

Fix #1
Locks


Shared Mutable State
Threads
Lock
Threads
Race condition


Shared Mutable State
Threads
Coarse lock
Threads
Race condition

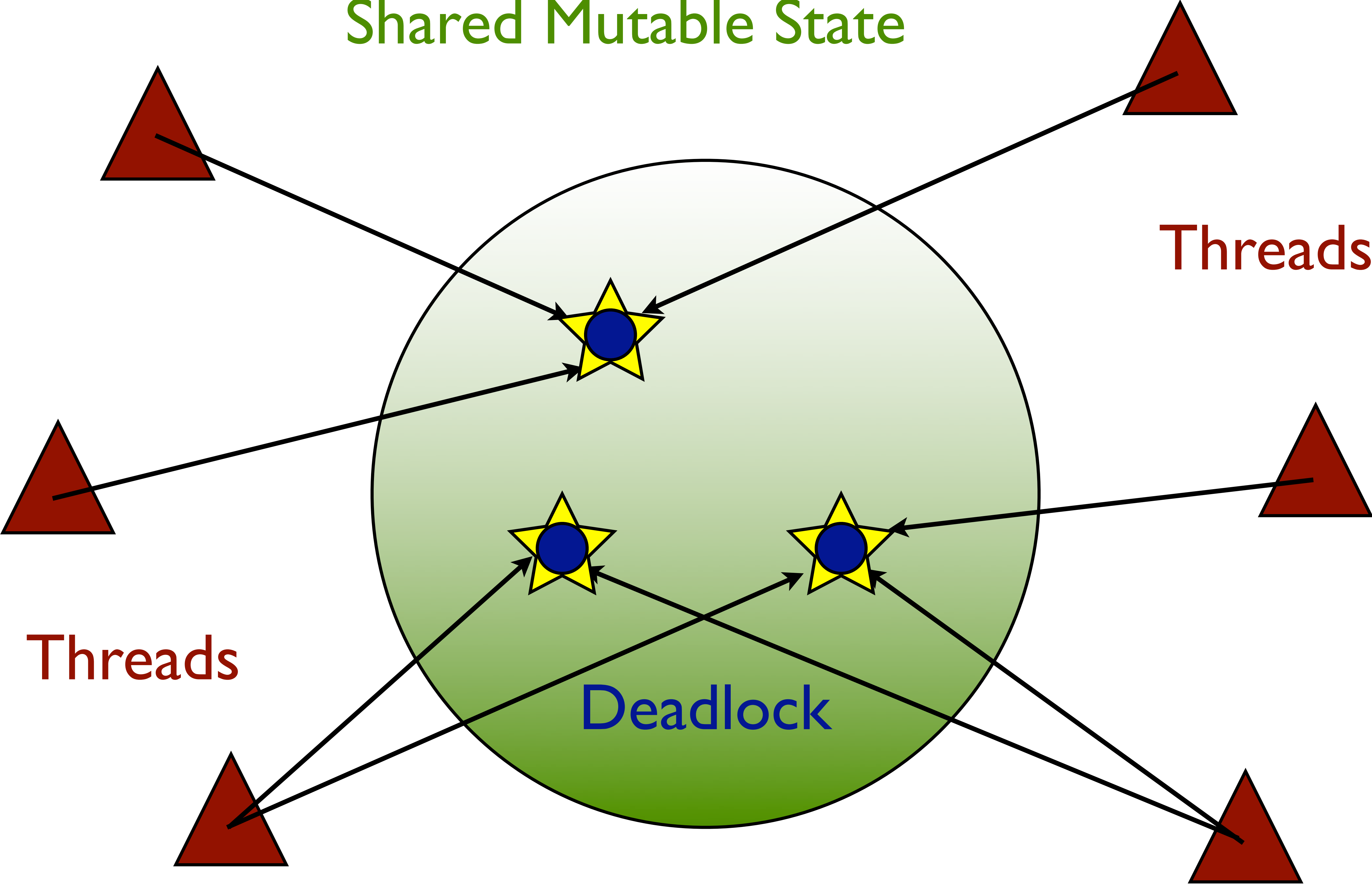
Problem #2
Blocking/contention


Shared Mutable State
Threads
Fine-grained locks
Threads
Race condition

Problem #3
Overhead

Solutions #2 & #3
??? ... Nothing, really
It’s a trade-off

Problem #4
Deadlock


Shared Mutable State
Threads
Threads
Deadlock

Solution #4
Rigorous adherence to complicated lock order

Problem #5
Doesn’t compose

Problem #6
Hard to debug

Languages

- Well... very popular

Paradigm:
Actors

Actor Principles

Actor Principles

- No shared, mutable state

Actor Principles

- No shared, mutable state
  - All mutable state is private

Actor Principles

- No shared, mutable state
  - All mutable state is private
  - All shared state is immutable

Actor Principles

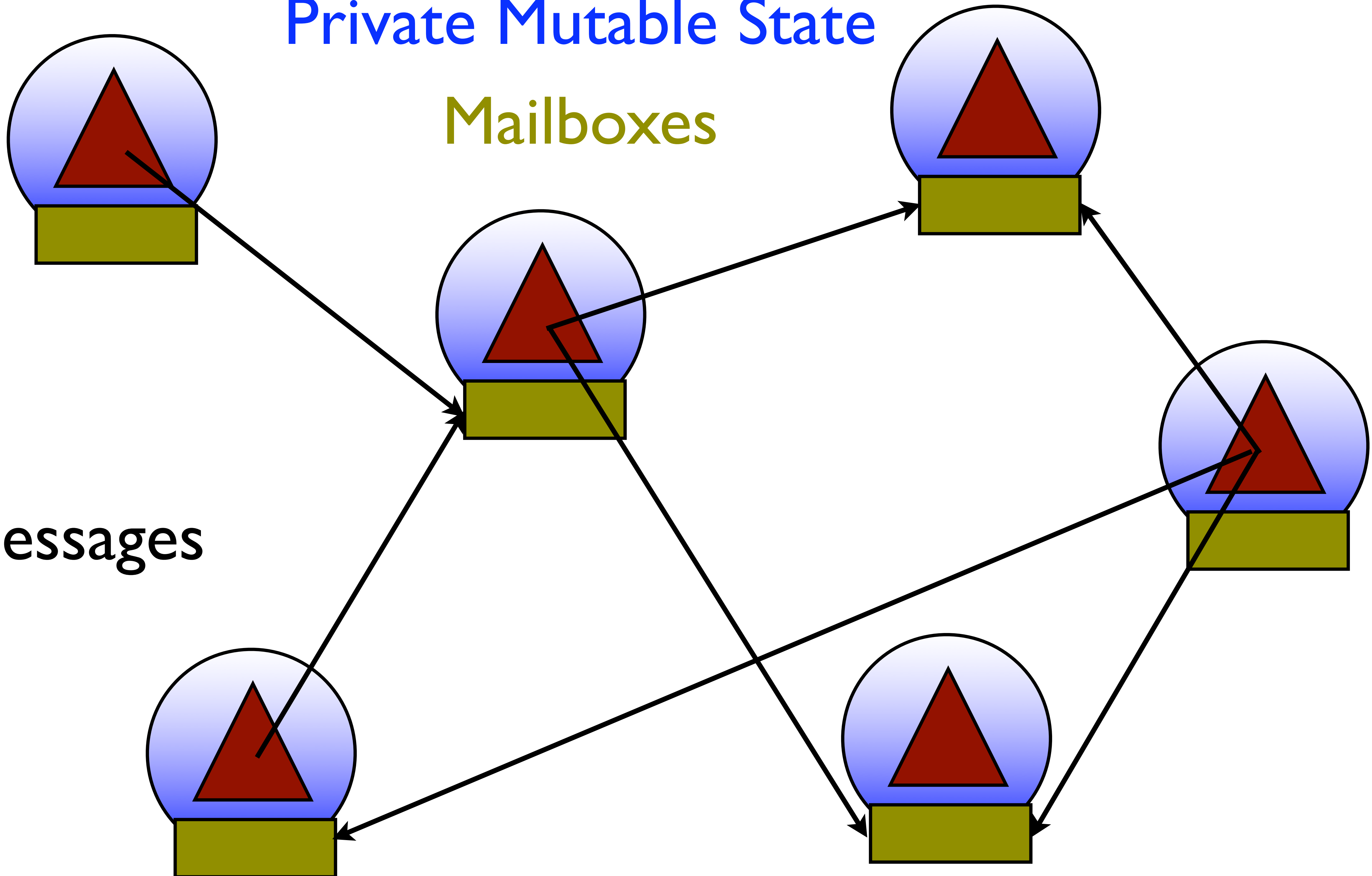
- No shared, mutable state
  - All mutable state is private
  - All shared state is immutable
- Communicate via immutable, asynchronous message-passing

Actor Principles

- No shared, mutable state
  - All mutable state is private
  - All shared state is immutable
- Communicate via immutable, asynchronous message-passing
- Messages, threads are extremely lightweight

Actor Components

- Actors: Very lightweight threads
- Messages: Immutable (hence, lightweight)
- Mailboxes: Asynchronous, non-blocking queues


Actors
Private Mutable State
Mailboxes
Messages

In Erlang

```
start() -> loop(0).

loop(Sum) ->
  receive
    {increment, Count} ->
      loop(Sum + Count);
    reset ->
      loop(0);
    {counter, Pid} ->
      Pid ! {counter, Sum},
      loop(Sum);
  end.
```

In Scala

```
// We need to define our messages first

case class Increment(count: Int)
case object Reset
case class ReportTotal(a: Actor)
case class ReportingTotal(sum: Int)
```

In Scala

```
object Counter extends Actor {
  var sum = 0

  def act() = while(true) {
    receive {
      case Increment(cnt) =>
        sum += cnt
      case Reset =>
        sum = 0
      case ReportTotal(actor) =>
        actor ! ReportingTotal(sum)
    }
  }
}
```

In Scala

```
// One hundred actors are defined,
// each sends one message
for (i <- (1 to 100)) {
  actor {
    Counter ! Increment(i)
  }
}

// Ask for the total and print it
Counter ! ReportTotal(this)
receive {
  case ReportingTotal(sum) =>
    println(sum)
}
```

In Scala

```
object Counter extends Actor {
  var sum = 0

  def act() = while(true) {
    receive {
      case Increment(cnt) =>
        sum += cnt
      case Reset =>
        sum = 0
      case ReportTotal =>
        reply(sum)
    }
  }
}
```

In Scala

```
// One hundreds actors are defined,
// each sends one message
for (i <- (1 to 100)) {
  actor {
    Counter ! Increment(i)
  }
}

// Ask for the total and print it
Counter !? ReportTotal match {
  case sum: Int => println(sum)
}
```

Implementation

```
// Subclasses must implement the abstract
// “act” method
class Actor {
  def !(msg: Any): Unit = ...
  def !?(msg: Any): Any = ...
  def act(): Unit
}
```

Implementation

- Java threads are too heavy (only scales to a few thousand threads)
- Scala actors are much lighter weight (scales to millions of actors)
- See: “Actors that Unify Threads and Events”, Haller and Odersky

Lessons

- Scala’s flexible syntax allows for a remarkably faithful implementation of Erlang’s concurrency primitives as a Scala library and not a language primitive
- We lose some type safety
  - We can use typed Channels to partly remedy this

Advantages

- Much easier to reason about than locks and shared memory
- Proven to be reliable and scale to large systems
- Don’t care if actors are local or remote
- Implemented as a library, not core part of language

Disadvantages

- Overhead
- Implemented as a library, not core part of VM

Languages

- Erlang
- Actalk, Actra (Smalltalk)
- SALSA (Java extension)
- Scala

Paradigm: Fork/Join

Fork/Join

```
// PSEUDOCODE
def solve(p: Problem): Result {
  if (problem.size < THRESHOLD)
    solveSequentially(p)
  else {
    INVOKE-IN-PARALLEL {
      left = solve(extractLeft(p));
      right = solve(extractRight(p));
    }
    combine(left, right);
  }
}
```

Advantages

- Easy to reason about
- Easy to implement

Disadvantages

- Primitive framework
- Not all tasks are trivially parallelizable

Languages

- Java 7
- Almost trivial in Scala

In Scala

```
def future[A](p: => A): Unit => A = {
  val result = new SyncVar[A]
  spawn { result.set(p) }
  () => result.get
}
```

In Scala

```
def solve(p: Problem): Result {
  if (problem.size < THRESHOLD)
    solveSequentially(p)
  else {
    left = future(solve(extractLeft(p)))
    right = solve(extractRight(p))
    combine(left(), right)
  }
}
```

Paradigm: Transactional Memory

Transactional Memory

- Hardware Transactional Memory
- Software Transactional Memory

Transactional Memory

```
// insert a node into a
// doubly-linked list
atomically {
  newNode.prev = node
  newNode.next = node.next
  node.next.prev = newNode
  node.next = newNode
}
```

Advantages

- Easier to reason about
- Much more composable

Disadvantages

- Doesn’t support irreversible operations
- Tricky to implement

Languages

- Gemstone (Smalltalk)
- Haskell
- Clojure
- coThreads (OCaml)
- Scala (unofficial, experimental)
- (various C, C++, Java, C# implementations)

Paradigm:
Join Calculus

Join Calculus

```
// Unbounded buffer
class Buffer {
  def get(): String
    & def put(s: String): Async = {
      s
    }
}
```

Join Calculus

```
// One-place bounded buffer
class OnePlaceBuffer {
  empty()

  def put(s: String): Unit
    & private def empty(): Async = {
      has(s)
    }
  }

  def get(): String
    & private def has(s: String): Async = {
      s
    }
}
```

Languages

- JoCaml
- Polyphonic C#, Cω
- Join Java
- Boost.Join
- scala.concurrent.jolib? .pilib?