The synchronized block

The `synchronized` block is used to achieve atomicity. Code block after a `synchronized` call on an object `x` is never executed by two threads at the same time.

The synchronized block

The `synchronized` block is used to achieve atomicity. Code block after a `synchronized` call on an object `x` is never executed by two threads at the same time.

```
private val x = new AnyRef {}
private var uidCount = 0L
def getUniqueId(): Long = x.synchronized {
  uidCount = uidCount + 1
  uidCount
}
```

The synchronized block

The `synchronized` block is used to achieve atomicity. Code block after a `synchronized` call on an object `x` is never executed by two threads at the same time.

```
private val x = new AnyRef {}
private var uidCount = 0L
def getUniqueId(): Long = x.synchronized {
  uidCount = uidCount + 1
  uidCount
}
```

Different threads use the `synchronized` block to agree on unique values. The `synchronized` block is an example of a *synchronization primitive*.

Composition with the synchronized block

Invocations of the `synchronized` block can nest.

Composition with the synchronized block

Invocations of the `synchronized` block can nest.

```
class Account(private var amount: Int = 0) {
  def transfer(target: Account, n: Int) =
    this.synchronized {
      target.synchronized {
        this.amount -= n
        target.amount += n
      }
    }
}
```

Time for a demo!

Deadlocks

Deadlock is a scenario in which two or more threads compete for resources (such as monitor ownership), and wait for each to finish without releasing the already acquired resources.

Deadlocks

Deadlock is a scenario in which two or more threads compete for resources (such as monitor ownership), and wait for each to finish without releasing the already acquired resources.

```
val a = new Account(50)
val b = new Account(70)

// thread T1
a.transfer(b, 10)

// thread T2
b.transfer(a, 10)
```

Resolving deadlocks

One approach is to always acquire resources in the same order.

Resolving deadlocks

One approach is to always acquire resources in the same order.

This assumes an ordering relationship on the resources.

Resolving deadlocks

One approach is to always acquire resources in the same order.

This assumes an ordering relationship on the resources.

```
val uid = getUniqueUid()
private def lockAndTransfer(target: Account, n: Int) =
  this.synchronized {
    target.synchronized {
      this.amount -= n
      target.amount += n
    }
  }
def transfer(target: Account, n: Int) =
  if (this.uid < target.uid) this.lockAndTransfer(target, n)
  else target.lockAndTransfer(this, -n)
```

Memory model

Memory model is a set of rules that describes how threads interact when accessing shared memory.

Java Memory Model – the memory model for the JVM.

Memory model

Memory model is a set of rules that describes how threads interact when accessing shared memory.

Java Memory Model – the memory model for the JVM.

1. Two threads writing to separate locations in memory do not need synchronization.
2. A thread X that calls `join` on another thread Y is guaranteed to observe all the writes by thread Y after `join` returns.

Summary

The parallelism constructs in the remainder of the course are implemented in terms of:

- threads
- synchronization primitives such as `synchronized`

Summary

The parallelism constructs in the remainder of the course are implemented in terms of:

- threads
- synchronization primitives such as `synchronized`

It is important to know what's under the hood!