# Introduction: Concurrent Programming

Principles of Functional Programming

Martin Odersky and Aleksandar Prokopec

# A Surprising Program

```
var a, b = false
var x, y = -1
val t1 = thread {
    Thread.sleep(1)
    a = true
    y = if b then 0 else 1
}
val t2 = thread {
    Thread.sleep(1)
    b = true
    x = if a then 0 else 1
}
t1.join(); t2.join()
assert(!(x == 1 && y == 1))
```

# Enumerating Executions Intuitively

- Possibility I:
  - `t1` writes `true` to `a`
  - `t1` reads `b` and sees `false` – writes `1` to `y`
  - `t2` writes `true` to `b`
  - `t2` reads `a` and sees `true` – writes `0` to `x`

# Enumerating Executions Intuitively

- Possibility I:
  - `t1` writes `true` to `a`
  - `t1` reads `b` and sees `false` – writes `1` to `y`
  - `t2` writes `true` to `b`
  - `t2` reads `a` and sees `true` – writes `0` to `x`
- Possibility II: same as I, with `t1` and `t2` reversed $(\texttt{x} = 1, \texttt{y} = \texttt{0})$

# Enumerating Executions Intuitively

- Possibility I:
  - `t1` writes `true` to `a`
  - `t1` reads `b` and sees `false` – writes `1` to `y`
  - `t2` writes `true` to `b`
  - `t2` reads `a` and sees `true` – writes `0` to `x`
- Possibility II: same as I, with `t1` and `t2` reversed ($\texttt{x} = \texttt{1}$, $\texttt{y} = \texttt{0}$)
- Possibility III:
  - `t1` writes `true` to `a`
  - `t2` writes `true` to `b`
  - `t1` reads `b` and sees `true` – writes `0` to `y`
  - `t2` reads `a` and sees `true` – writes `0` to `x`

# Enumerating Executions Intuitively

- Possibility I:
  - t1 writes true to a
  - t1 reads b and sees false – writes 1 to y
  - t2 writes true to b
  - t2 reads a and sees true – writes 0 to x
- Possibility II: same as I, with t1 and t2 reversed ($x = 1$, $y = 0$)
- Possibility III:
  - t1 writes true to a
  - t2 writes true to b
  - t1 reads b and sees true – writes 0 to y
  - t2 reads a and sees true – writes 0 to x
- Conclusion: there is no execution in which $x = 1$ *and* $y = 1$

# Surprising Result

However, upon running the previous program many times, we get occasional executions in which `x` $= 1$ and `y` $= 1$ – the `assert` statement sometimes crashes the program.

# Surprising Result

However, upon running the previous program many times, we get occasional executions in which `x` = 1 and `y` = 1 – the `assert` statement sometimes crashes the program.

Something is wrong with our intuition of concurrent programming.

Let's rebuild our intuition about concurrency, starting from the basic principles.

# Goals of a Concurrent Programming Model

Every concurrent programming model must answer two questions:

1. How to express that two executions are concurrent?
2. Given a set of concurrent executions, how can they exchange information (i.e. synchronize)?

In what follows, we will answer these two questions in the context of the JVM concurrency model.

# Threads

The `thread` notation starts a new *thread* – a concurrent execution.

```
thread {
    a = true
    y = if b then 0 else 1
}
```

The `thread` function is implemented as follows:

```
def thread(body: => Unit): Thread =
  val t = new Thread:
      override def run() = body
  t.start()
  t
```

# Why Threads?

Why do we need threads? Why not just program with CPUs directly?

```
at (CPU1) {
    a = true
    y = if b then 0 else 1
}
```

# Why Threads?

Why do we need threads? Why not just program with CPUs directly?

```
at (CPU1) {
    a = true
    y = if b then 0 else 1
}
```

Several reasons:

- portability – number of available processors varies (not every computer has a `CPU1`)
- number of concurrent entities in a program can be much larger than the number of CPUs

# Thread as an Abstraction

*Principle: threads express opportunities for concurrency in the program.*

# Thread as an Abstraction

> *Principle: threads express opportunities for concurrency in the program.*

To start a thread:

1. Define what a thread does (on the JVM, create a `Thread` object and override `run`).
2. Start a thread instance (on the JVM, call `start` on the `Thread` object).

# Thread Under-the-Hood

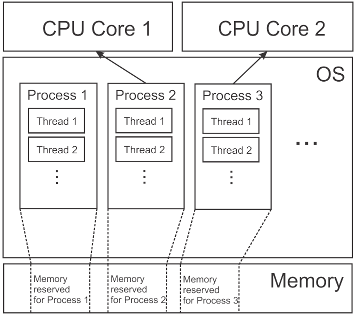
A thread image in memory contains:

- copies of processor registers
- the call stack (default size: ~2MB)

Hence, we cannot have more than a couple of thousand threads per VM.

# Cores, Processes, Threads

How do the threads get assigned to CPUs under-the-hood?



User program instances are separated into processes, which have separate memory spaces.

Each process can have multiple threads, and starts with one *main thread*.

# Role of the OS

The operating system *eventually* assigns threads to processes (the OS guarantees liveness).

# Role of the OS

The operating system *eventually* assigns threads to processes (the OS guarantees liveness).

Two approaches:

- cooperative multitasking – a program has to explicitly give control (yield) back to the OS (think Windows 3.1)
- preemptive multitasking – the OS has a hardware timer that periodically interrupts the running thread, and assigns different thread to the CPU (*time slices* usually ∼10 ms)

# Example Thread Program

```
def log(msg: String) = println(s”${Thread.currentThread}: $msg”)

log(”Creating a new thread.”)
val t = thread {
    log(”New thread still running.”)
    Thread.sleep(1000)
    log(”Completed.”)
}
```

The `Thread.sleep` statement pauses the thread, and revives after the specified period in milliseconds.

# Example Thread Program

```
def log(msg: String) = println(s”${Thread.currentThread}: $msg”)

log(”Creating a new thread.”)
val t = thread {
    log(”New thread still running.”)
    Thread.sleep(1000)
    log(”Completed.”)
}
```

The `Thread.sleep` statement pauses the thread, and revives after the specified period in milliseconds.

The program above is deterministic, but this is not generally so.

# Non-Deterministic Programs

Given the same input, the program output is not unique between multiple runs.

```
@main def ThreadsNonDeterminism =
    val t = thread {
      log(”New thread running”)
    }
    log(”...”)
    log(”...”)
}
```

We call such programs *non-deterministic*.

# Synchronization

The second requirement of a concurrent programming model is synchronization. On JVM, threads synchronize through shared memory.

# Synchronization

The second requirement of a concurrent programming model is synchronization. On JVM, threads synchronize through shared memory.

Some problems with shared memory synchronization:

- Interleaving and race conditions
- Deadlocks
- Data races

# Thread Joining

One of the basic forms of synchronization.

The call `t.join()` lets the calling thread wait until thread `t` has terminated.

```
@main def ThreadsStart =
    class MyThread extends Thread:
        override def run(): Unit = println(s”New thread running”)
    val t = new MyThread()
    t.start()
    t.join()
    println(s”New thread joined”)
```

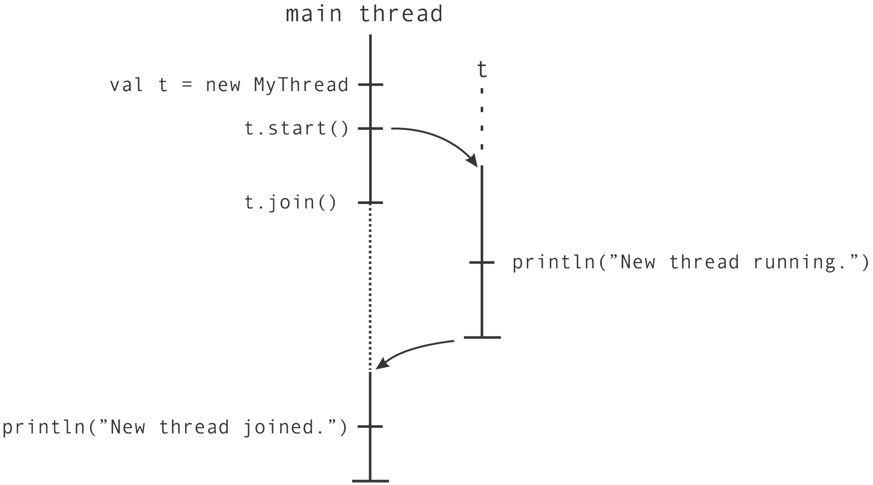
# Thread Joining

One of the basic forms of synchronization.

The call `t.join()` lets the calling thread wait until thread `t` has terminated.

```
@main def ThreadsStart =
    val t = thread { println(s”New thread running”) }
    t.join()
    println(s”New thread joined”)
```

# Start and Join - Diagrammatically



# Guarantees When Joining Threads

When `join` returns, the effects of the terminated thread are visible to the thread that called `join`.

```
@main def ThreadsStart =
    var a = false
    val t = thread {
        a = true
    }
    t.join()
    assert(a)
```

# Guarantees When Joining Threads

When `join` returns, the effects of the terminated thread are visible to the thread that called `join`.

```
@main def ThreadsStart =
    var a = false
    val t = thread {
        a = true
    }
    t.join()
    assert(a)
```

However, `join` is not useful enough – it always demands that the target thread terminates.

How to see the effects of another thread that did not yet terminate?

# Interleaving

Consider the problem of implementing a concurrent UID generator.

# Interleaving

Consider the problem of implementing a concurrent UID generator.

```
object GetUID:
    var uidCount = 0
    def getUniqueId() =
        val freshUID = uidCount + 1
        uidCount = freshUID
        freshUID
```

Is it true that every call to `getUniqueUID` will yield a *unique* number?

# Interleaving

Let's put it to the test:

```
@main def ThreadsGetUID =
    def printUniqueIds(n: Int): Unit =
        val uids = for i <- 0 until n yield GetUID.getUniqueId()
        log(s”Generated uids: $uids”)

    val t = thread { printUniqueIds(5) }
    printUniqueIds(5)
    t.join()
```

# Sample Test Output

```
> java ThreadsGetUID
Thread[main,5,main]: Generated uids: Vector(1, 3, 4, 6, 8)
Thread[Thread-0,5,main]: Generated uids: Vector(2, 5, 7, 9, 10)
> java ThreadsGetUID
Thread[Thread-0,5,main]: Generated uids: Vector(1, 2, 3, 4, 5)
Thread[main,5,main]: Generated uids: Vector(6, 7, 8, 9, 10)
> java ThreadsGetUID
Thread[Thread-0,5,main]: Generated uids: Vector(1, 2, 3, 4, 5)
Thread[main,5,main]: Generated uids: Vector(6, 7, 8, 9, 10)
s> java ThreadsGetUID
Thread[main,5,main]: Generated uids: Vector(1, 2, 3, 4, 5)
Thread[Thread-0,5,main]: Generated uids: Vector(6, 7, 8, 9, 10)
> java ThreadsGetUID
Thread[Thread-0,5,main]: Generated uids: Vector(1, 2, 4, 6, 8)
Thread[main,5,main]: Generated uids: Vector(1, 3, 5, 7, 9)
```

# Interleaving

We observe:

- Most runs produce different sequences of IDs.
- In most runs, threads share some IDs but not others.

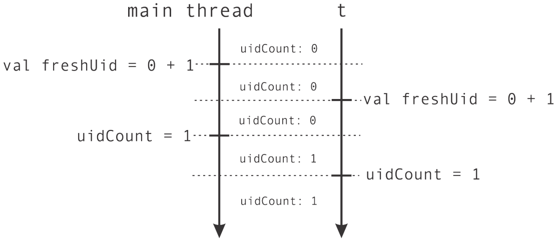
How can we explain this?

# Interleaving

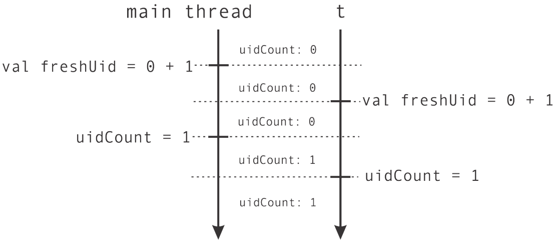
We observe:

- Most runs produce different sequences of IDs.
- In most runs, threads share some IDs but not others.

How can we explain this?



# Race condition



The scenario shown in the previous figure is called a *race condition*.

A race condition occurs when the behavior of the system depends on the interleaving of executions.

# Atomic Execution

We would like to ensure that all operations of `getUniqueId` are performed atomically, without another thread reading or writing intermediate results.

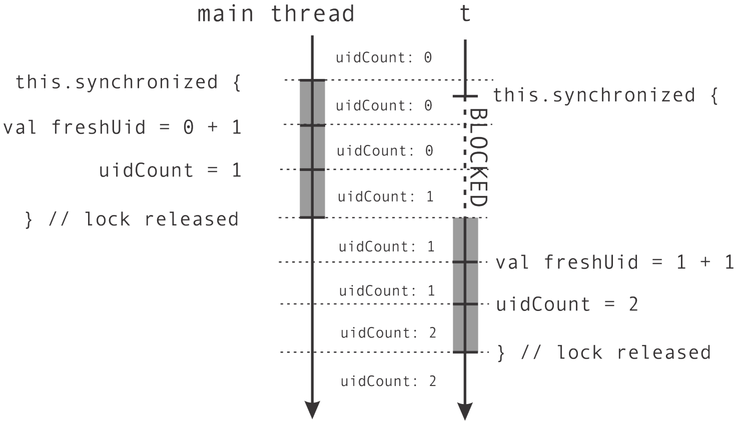
# Atomic Execution

We would like to ensure that all operations of `getUniqueId` are performed atomically, without another thread reading or writing intermediate results.

This can be achieved by wrapping a block in a `synchronized` call:

```
object GetUID extends Monitor:
    var uidCount = 0
    def getUniqueId() = synchronized {
        val freshUID = uidCount + 1
        uidCount = freshUID
        freshUID
    }
  ...
```

# Atomic Execution



main thread

t

uidCount: 0

```
this.synchronized {
val freshUid = 0 + 1
uidCount = 1
} // lock released
```

uidCount: 0

uidCount: 0

uidCount: 1

uidCount: 1

uidCount: 1

uidCount: 2

uidCount: 2

```
this.synchronized {
```

BLOCKED

```
val freshUid = 1 + 1
uidCount = 2
} // lock released
```

# Synchronized statement

In Scala, `synchronized` is a member method of `AnyRef`. In the call:

```
obj.synchronized { block }
```

`obj` serves as a *lock*.

- `block` can be executed only by thread `t` holds the lock.
- At most one thread can hold a lock at any one time.

# Synchronized statement

In Scala, `synchronized` is a member method of `AnyRef`. In the call:

```
obj.synchronized { block }
```

`obj` serves as a *lock*.

- `block` can be executed only by thread `t` holds the lock.
- At most one thread can hold a lock at any one time.

Consequently, if another thread is already running `synchronized` on the same object (i.e. it holds the lock), then a thread calling `synchronized` gets temporarily blocked, until the lock is released.

# Monitors

- Having `synchronized` defined on any object imposes a significant runtime cost.
- So that's generally considered a design mistake of Java.
- Typically, we want provide synchronized (and associated operations such as `wait`, `notify`) only for instances of a special class `Monitor`.
- That's what the examples in this course segment do.
- Our testing framework also requires to extend from `Monitor` since it overrides `synchronized`.

# Example: Money Transfers

The `synchronized` statements can nest, which allows composition.

# Example: Money Transfers

The synchronized statements can nest, which allows composition.

Let's design an online banking system in which we want to log money transfers.

First, here is the code to collect log messages:

```scala
import scala.collection._
private val transfers = mutable.ArrayBuffer[String]()
private val log = new Monitor {}
def logTransfer(name: String, n: Int) = log.synchronized {
    transfers += s”transfer to account $name = $n”
}
```

Note the synchronized, which is needed because += is not by itself atomic.

# Accounts

Next, here is the class Account:

```
class Account(val name: String, initialBalance: Int) extends Monitor:
    private var myBalance = initialBalance
    def balance: Int = this.synchronized { myBalance }
    def add(n: Int): Unit = this.synchronized {
        myBalance += n
        if n > 10 then logTransfer(name, n)
    }
    val getUID = ThreadsGetUID.getUniqueId()
```

(we will need getUID later)

# Test Code

Finally, here is some code that simulates account movements:

```
val jane = new Account(”Jane”, 100)
val john = new Account(”John”, 200)
val t1 = thread { jane.add(5) }
val t2 = thread { john.add(50) }
val t3 = thread { jane.add(70) }
t1.join(); t2.join(); t3.join()
log(s”--- transfers ---\n$transfers”)
```

Note the nested `synchronized` calls: First on `add`, then on `logTransfer`.

# Transfers

Let's add a method that transfers money from one account to another (as an atomic action):

```
def transfer(a: Account, b: Account, n: Int) =
    a.synchronized {
        b.synchronized {
            a.add(n)
            b.add(-n)
        }
    }
```

# Testing Transfers

Test it as follows:

```
val jane = new Account(”Jane”, 1000)
val john = new Account(”John”, 2000)
log(”started...”)
val t1 = thread { for i <- 0 until 100 do transfer(jane, john, 1) }
val t2 = thread { for i <- 0 until 100 do transfer(john, jane, 1) }
t1.join(); t2.join();
log(s”john = ${john.balance}, jane = ${jane.balance}”)
```

# Testing Transfers

Test it as follows:

```
val jane = new Account(”Jane”, 1000)
val john = new Account(”John”, 2000)
log(”started...”)
val t1 = thread { for i <- 0 until 100 do transfer(jane, john, 1) }
val t2 = thread { for i <- 0 until 100 do transfer(john, jane, 1) }
t1.join(); t2.join();
log(s”john = ${john.balance}, jane = ${jane.balance}”)
```

What behavior do you expect to see?

1. The program terminates with both accounts having the same balance at the end as at the beginning.
2. The program terminates with accounts having a different balance.
3. The program crashes.
4. The program hangs.

# Deadlocks

Here's a possible sequence of events:

## Deadlocks

Here's a possible sequence of events:

A situation like this where no thread can make progress because each thread waits on some lock is called a **deadlock**.

# Preventing Deadlocks

In the previous example, a deadlock arose because two threads tried to grab two locks in different order.

To prevent the deadlock, we can enforce that locks are always taken in the same order by all threads.

Here's a way to do this:

```scala
def transfer(a: Account, b: Account, n: Int) =
    def adjust() { a.add(n); b.add(-n) }
    if a.getUID < b.getUID then
        a.synchronized { b.synchronized { adjust() } }
    else
        b.synchronized { a.synchronized { adjust() } }
```

# Monitors

The `fork` and the `synchronized` statement allow *serializing* the execution of different threads, but sometimes a thread needs to wait for a specific condition (for example, a particular value of some variable).

In such cases, we need a synchronization primitive called a *monitor*.

# Monitors

The `fork` and the `synchronized` statement allow *serializing* the execution of different threads, but sometimes a thread needs to wait for a specific condition (for example, a particular value of some variable).

In such cases, we need a synchronization primitive called a *monitor*.

In Scala and Java, the `synchronized` statement has a dual purpose – it can be used both as a lock and as a monitor.

# Monitors

The `fork` and the `synchronized` statement allow *serializing* the execution of different threads, but sometimes a thread needs to wait for a specific condition (for example, a particular value of some variable).

In such cases, we need a synchronization primitive called a *monitor*.

In Scala and Java, the `synchronized` statement has a dual purpose – it can be used both as a lock and as a monitor.

Let's implement a commonly used synchronization data structure – a one-place buffer.

# One-Place Buffer

When using a one-place buffer, we will distinguish two roles a thread can have, a *producer*, or a *consumer*.

# One-Place Buffer

When using a one-place buffer, we will distinguish two roles a thread can have, a *producer*, or a *consumer*.

The one-place buffer has the following invariants:

- *producers* send an element to the buffer.
- *consumers* take an element from the buffer.
- at most one element can be in the buffer at any one time.
- If buffer is full, producers have to wait.
- If buffer is empty, consumers have to wait.

# Implementation Schema

Here's an outline of class `OnePlaceBuffer`

```
import compiletime.unintialized
class OnePlaceBuffer[Elem] extends Monitor:
    private var elem: Elem = uninitialized
    private var full: Boolean = false
    def put(e: Elem): Unit = synchronized {
        if full then ???
        else { elem = e; full = true }
    }
    def get(): Elem = synchronized {
        if !full then ???
        else { full = false; elem }
    }
```

# Implementation Schema

Here's an outline of class `OnePlaceBuffer`

```
import compiletime.unintialized
class OnePlaceBuffer[Elem] extends Monitor:
    private var elem: Elem = uninitialized
    private var full: Boolean = false
    def put(e: Elem): Unit = synchronized {
        if full then ???
        else { elem = e; full = true }
    }
    def get(): Elem = synchronized {
        if !full then ???
        else { full = false; elem }
    }
```

**Question**: How to implement the `???`s?

# Busy Waiting

We could wait as follows:

```
class OnePlaceBuffer[Elem] extends Monitor:
    private var elem: Elem = uninitialized
    private var full: Boolean = false
    def put(e: Elem) = while !tryToPut(e) do {}
    def tryToPut(e: Elem): Boolean = synchronized {
        if full then false
        else { elem = e; full = true; true }
    }
  // similarly for get
```

This technique is called *polling* or *busy waiting*.

Problem: Consumes compute time while waiting.

# Wait and Notify

A monitor can be used for more than just locking. Every monitor object has the following methods:

# Wait and Notify

A monitor can be used for more than just locking. Every monitor object has the following methods:

`wait()` suspends the current thread,

`notify()` wakes up one other thread waiting on the current object,

`notifyAll()` wakes up all other thread waiting on the current object.

# Wait and Notify

A monitor can be used for more than just locking. Every monitor object has the following methods:

| | |
|---|---|
| `wait()` | suspends the current thread, |
| `notify()` | wakes up one other thread waiting on the current object, |
| `notifyAll()` | wakes up all other thread waiting on the current object. |

**Note:** these methods can only be called from inside a `synchronized` statement.

# Blocking Implementation

```
class OnePlaceBuffer[Elem] extends Monitor:
  var elem: Elem = uninitialized; var full = false
  def put(e: Elem): Unit = synchronized {
      while full do wait()
      elem = e; full = true; notifyAll()
  }
  def get(): Elem = synchronized {
      while !full do wait()
      full = false; notifyAll(); elem
  }
```

# Blocking Implementation

```
class OnePlaceBuffer[Elem] extends Monitor:
  var elem: Elem = uninitialized; var full = false
  def put(e: Elem): Unit = synchronized {
      while full do wait()
      elem = e; full = true; notifyAll()
  }
  def get(): Elem = synchronized {
      while !full do wait()
      full = false; notifyAll(); elem
  }
```

**Questions:**

1. Why `notifyAll()` instead of `notify()`?
2. Why `while full do wait()` instead of `if full then wait()`?

# The Fine Print

- `wait`, `notify` and `notifyAll` should only be called from within a `synchronized` on `this`

# The Fine Print

- `wait`, `notify` and `notifyAll` should only be called from within a `synchronized` on `this`
- `wait` will release the lock, so other threads can enter the monitor

# The Fine Print

- `wait`, `notify` and `notifyAll` should only be called from within a `synchronized` on `this`
- `wait` will release the lock, so other threads can enter the monitor
- `notify` and `notifyAll` schedule other threads for execution after the calling thread has released the lock (has left the monitor)

# The Fine Print

- `wait`, `notify` and `notifyAll` should only be called from within a `synchronized` on `this`
- `wait` will release the lock, so other threads can enter the monitor
- `notify` and `notifyAll` schedule other threads for execution after the calling thread has released the lock (has left the monitor)
- on the JVM runtime, it is possible that a thread calling `wait` sometimes wakes up even if nobody called `notify` or `notifyAll`

# Visibility

While discussing `fork` and `synchronized`, we did not examine how they affect the visibility of memory writes.

# Visibility

While discussing `fork` and `synchronized`, we did not examine how they affect the visibility of memory writes.

The `join` method ensures that all the writes of the joined thread are visible to the thread that called `join`.

# Visibility

While discussing `fork` and `synchronized`, we did not examine how they affect the visibility of memory writes.

The `join` method ensures that all the writes of the joined thread are visible to the thread that called `join`.

The `synchronized` statement ensures that all the writes by thread A preceding the release of the lock by that thread A are visible to any thread B that subsequently acquires the lock.

# A Memory Model

A *memory model* is a set of rules that defines how and when the writes to memory by one thread become visible to other threads.

# A Memory Model

A *memory model* is a set of rules that defines how and when the writes to memory by one thread become visible to other threads.

Consider our introductory example:

```
var a, b = false; var x, y = -1
val t1 = thread {
    a = true
    y = if b then 0 else 1
}
val t2 = thread {
    b = true
    x = if a then 0 else 1
}
t1.join(); t2.join()
assert(!(x == 1 && y == 1))
```

# Sequential Consistency Model

When we initially analyzed the introductory example, we assumed that every read and write happens in the program order, and that every read and write goes to main memory.

# Sequential Consistency Model

When we initially analyzed the introductory example, we assumed that every read and write happens in the program order, and that every read and write goes to main memory.

That specific memory model is called the *sequential consistency* model.

# Sequential Consistency Model

When we initially analyzed the introductory example, we assumed that every read and write happens in the program order, and that every read and write goes to main memory.

That specific memory model is called the *sequential consistency* model.

More formally:

> *Consider all the reads and writes to program variables. If the result of the execution is the same as if the read and write operations were executed in some sequential order, and the operations of each individual processor appear in the program order, then the model is sequentially consistent.*

# Reorderings

Unfortunately, as we saw in our experiment, multicore processors and compilers do **not** implement the sequential consistency model.

# Reorderings

Unfortunately, as we saw in our experiment, multicore processors and compilers do **not** implement the sequential consistency model.

One reason are CPU registers, which hold local copies of memory values.

- Each core might have a different copy of shared memory in its registers.
- Writing the registers back to main-memory happens at unpredictable times.

Another reason are optimizing compilers: they are generally allowed to reorder instructions as if no other thread was watching.

# The Java Memory Model

The Java Memory Model (JMM) defines a "*happens-before*" relationship as follows.

- **Program order**: Each action in a thread *happens-before* every subsequent action in the same thread.

# The Java Memory Model

The Java Memory Model (JMM) defines a “*happens-before*” relationship as follows.

- **Program order**: Each action in a thread *happens-before* every subsequent action in the same thread.
- **Monitor locking**: Unlocking a monitor *happens-before* every subsequent locking of that monitor.

# The Java Memory Model

The Java Memory Model (JMM) defines a "*happens-before*" relationship as follows.

- **Program order**: Each action in a thread *happens-before* every subsequent action in the same thread.
- **Monitor locking**: Unlocking a monitor *happens-before* every subsequent locking of that monitor.
- **Volatile fields**: A write to a volatile field *happens-before* every subsequent read of that field.

# The Java Memory Model

The Java Memory Model (JMM) defines a “*happens-before*” relationship as follows.

- **Program order**: Each action in a thread *happens-before* every subsequent action in the same thread.
- **Monitor locking**: Unlocking a monitor *happens-before* every subsequent locking of that monitor.
- **Volatile fields**: A write to a volatile field *happens-before* every subsequent read of that field.
- **Thread start**: A call to `start()` on a thread *happens-before* all actions of that thread.

# The Java Memory Model

The Java Memory Model (JMM) defines a "*happens-before*" relationship as follows.

- **Program order**: Each action in a thread *happens-before* every subsequent action in the same thread.
- **Monitor locking**: Unlocking a monitor *happens-before* every subsequent locking of that monitor.
- **Volatile fields**: A write to a volatile field *happens-before* every subsequent read of that field.
- **Thread start**: A call to `start()` on a thread *happens-before* all actions of that thread.
- **Thread termination**. An action in a thread *happens-before* another thread completes a `join` on that thread.

# The Java Memory Model

The Java Memory Model (JMM) defines a "*happens-before*" relationship as follows.

- **Program order**: Each action in a thread *happens-before* every subsequent action in the same thread.
- **Monitor locking**: Unlocking a monitor *happens-before* every subsequent locking of that monitor.
- **Volatile fields**: A write to a volatile field *happens-before* every subsequent read of that field.
- **Thread start**: A call to `start()` on a thread *happens-before* all actions of that thread.
- **Thread termination**. An action in a thread *happens-before* another thread completes a `join` on that thread.
- **Transitivity**. If A happens before B and B *happens-before* C, then A *happens-before* C.

# The Java Memory Model ctd

Rule: if any two operations A and B are in a happens-before relationship, then B is guaranteed to see the memory effects of A.

# The Java Memory Model ctd

Rule: if any two operations A and B are in a happens-before relationship, then B is guaranteed to see the memory effects of A.

Example:

```
var a = 0
var set = false
thread {
  a = 1
  synchronized { set = true }
}
thread {
 synchronized { if set then println(a) }
}
```

Can the second thread print `0`?

# Observing Reorderings

```
var a, b = false
var x, y = -1
val t1 = thread {
    synchronized { a = true }
    synchronized { y = if b then 0 else 1 }
}
val t2 = thread {
    synchronized { b = true }
    synchronized { x = if a then 0 else 1 }
}
t1.join()
t2.join()
assert(!(x == 1) && (y == 1))
```

Will the assertion fail now?

# Extra: Volatile Variables

One last synchronization primitive that we examine are *volatile variables*.

Sometimes we need only safe publication instead of atomic execution. In these cases, `synchronized` can be too heavyweight.

# Extra: Volatile Variables

One last synchronization primitive that we examine are *volatile variables*.

Sometimes we need only safe publication instead of atomic execution. In these cases, `synchronized` can be too heavyweight.

There's a cheaper solution than using `synchronized` – we can use a *volatile field*.

# Guarantees of Volatile

Making a variable `@volatile` has several effects.

First, reads and writes to volatile variables are never reordered by the compiler.

# Guarantees of Volatile

Making a variable `@volatile` has several effects.

First, reads and writes to volatile variables are never reordered by the compiler.

Second, volatile reads and writes are never cached in CPU registers – they go directly to the main memory.

## Guarantees of Volatile

Making a variable `@volatile` has several effects.

First, reads and writes to volatile variables are never reordered by the compiler.

Second, volatile reads and writes are never cached in CPU registers – they go directly to the main memory.

Third, writes to normal variables that in the program precede a volatile write W cannot be moved by the compiler after W.

# Guarantees of Volatile

Making a variable `@volatile` has several effects.

First, reads and writes to volatile variables are never reordered by the compiler.

Second, volatile reads and writes are never cached in CPU registers – they go directly to the main memory.

Third, writes to normal variables that in the program precede a volatile write W cannot be moved by the compiler after W.

Fourth, reads from normal variables that in the program appear after a volatile read R cannot be moved by the compiler before R.

## Guarantees of Volatile

Making a variable `@volatile` has several effects.

First, reads and writes to volatile variables are never reordered by the compiler.

Second, volatile reads and writes are never cached in CPU registers – they go directly to the main memory.

Third, writes to normal variables that in the program precede a volatile write W cannot be moved by the compiler after W.

Fourth, reads from normal variables that in the program appear after a volatile read R cannot be moved by the compiler before R.

Fifth, before a volatile write, values cached in registers must be written back to main memory.

# Guarantees of Volatile

Making a variable `@volatile` has several effects.

First, reads and writes to volatile variables are never reordered by the compiler.

Second, volatile reads and writes are never cached in CPU registers – they go directly to the main memory.

Third, writes to normal variables that in the program precede a volatile write W cannot be moved by the compiler after W.

Fourth, reads from normal variables that in the program appear after a volatile read R cannot be moved by the compiler before R.

Fifth, before a volatile write, values cached in registers must be written back to main memory.

Sixth, after a volatile read, values cached in registers must be re-read from the main memory.

# Conclusion About Volatiles

For almost all practical purposes, you should keep your programs simple, and avoid using `@volatile` variables.

# Conclusion About Volatiles

For almost all practical purposes, you should keep your programs simple, and avoid using `@volatile` variables.

Instead, as a rule of the thumb, stick with `synchronized` when you need to make your writes visible to other threads.