Introduction: Concurrent Programming

Concurrent Programming

Martin Odersky

Concurrent Programming

Concurrent programming expresses a program as

- a set of concurrent computations
- that execute during overlapping time intervals
- and that need to coordinate in some way.

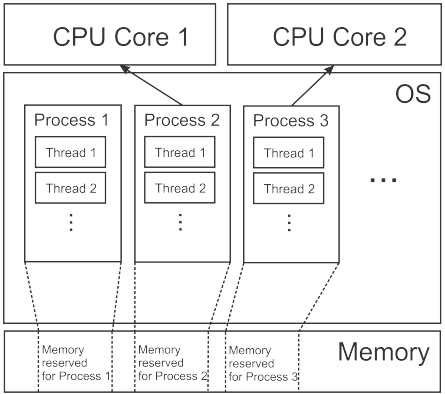
Concurrent is difficult; we get all the problems of sequential programming, plus

- non-determinism (behaviors change from one run to the next)
- race conditions (time-dependent behavior)
- deadlocks (program lockups)

Benefits of Concurrent Programming

1. Performance
   - exploit parallelism in hardware
   - parallel programming is implemented by means of concurrent programming
2. Responsiveness
   - faster I/O without blocking or polling
3. Distribution
   - Processes used in concurrent programming can be spread over several nodes.

Cores, Processes, and Threads



- OS schedules *cores* to run *threads*.
- Threads in the same process *share memory*.

Concurrency on the JVM

Every time a JVM process starts, it creates several threads, including the **main thread**, which runs the program.

You can find this out by running this little program:

```
object ThreadsMain extends App {
  val t: Thread = Thread.currentThread
  println(s"I am the thread ${t.getName}")
}
```

Threads on the JVM

A thread image in memory contains:

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

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

Context switch is a moderately complex operation

- ~ `1ms` per switch
- much more expensive than calls or objects creation
- less expensive than (blocking I/O)

Creating Threads

Threads are defined as subclasses of `java.lang.Thread`.

```
object ThreadsStart extends App {
  class MyThread extends Thread {
    override def run(): Unit = {
      println(s"I am ${Thread.currentThread.getName}")
    }
  }
  val t = new MyThread()
  t.start()
  println(s"I am ${Thread.currentThread.getName}")
}
```

Two steps to start a thread: (1) create an instance of a `Thread` subclass, (2) invoke its `start` method.

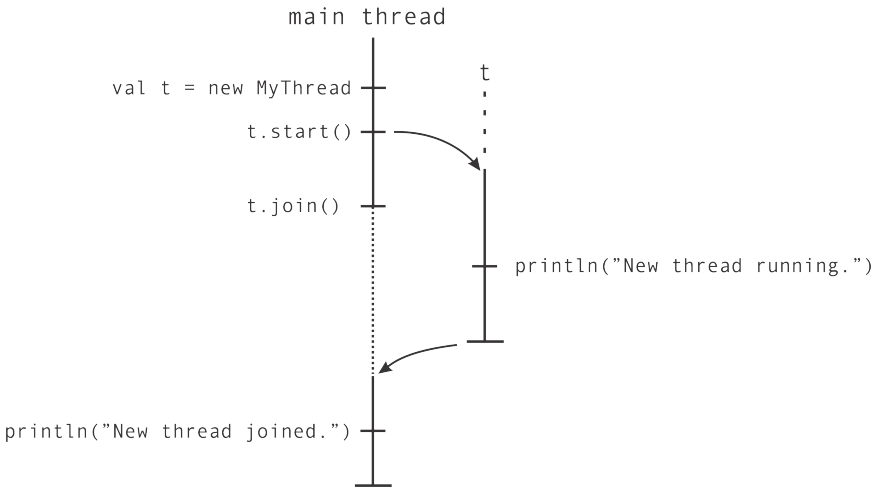
Waiting for Thread Termination

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

```
object ThreadsStart extends App {
  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")
}
```

Start and Join - Diagrammatically



Starting threads in Scala

In Scala, we can define a helper method to simplify thread creation:

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

Then the previous example becomes:

```
val t = thread { println(s"New thread running") }
t.join()
println(s"New thread joined")
```

Thread Execution

- The operating system contains a *scheduler* that runs all active threads on all cores.
- If there are more available threads than cores, threads are *time-sliced*.
- Typically, a thread is run for 10 to 100ms on a core, or until it pauses or waits.

Pausing Thread Execution

We can pause the currently running thread by calling its `sleep` method.

```
object ThreadsSleep extends App {

  val t = thread {
    Thread.sleep(1000)
    log("New thread running.")
    Thread.sleep(1000)
    log("Still running.")
    Thread.sleep(1000)
    log("Completed.")
  }
  t.join()
  log("New thread joined.")
}
```

Non-Determinism

The previous program was deterministic. It always printed

```
New thread running.
Still running.
Completed.
New thread joined.
```

in that order.

But this is not always the case.

Non-Determinism

Here is an example of a program that has different behaviors in different runs.

```
object ThreadsNonDeterminism extends App {
  val t = thread {
    log("New thread running")
  }
  log("...")
  log("...")
  t.join()
  log("New thread joined")
}
```

We call such programs *non-deterministic*.

Interleaving

Thread operations are executed concurrently or interleaved.

This invalidates many assumptions we have on sequential programs.

For instance, consider this object:

```
object ThreadsGetUID extends App {
  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:

```
  ...
  def printUniqueIds(n: Int): Unit = {
    val uids = for (i <- 0 until n) yield getUniqueId()
    log(s"Generated uids: $uids")
  }
  val t = thread { printUniqueIds(5) }
  printUniqueIds(5)
  t.join()
}
```

Interleaving

We observe:

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

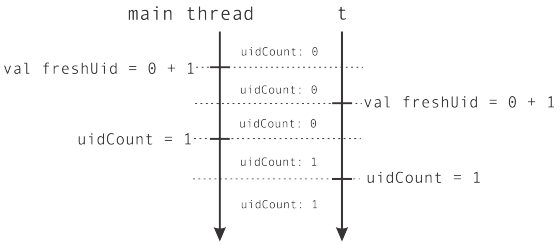
How can we explain this?

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?



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 ThreadsGetUID extends App {
  var uidCount = 0
  def getUniqueId() = synchronized {
    val freshUID = uidCount + 1
    uidCount = freshUID
    freshUID
  }
  ...
}
```

Synchronized

`synchronized` translates directly to Java bytecodes.

The Scala typechecker treats it as if it was a method of type `AnyRef`:

```
def synchronized[T](block: => T): T
```

A call of

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

executes `block` and makes sure that no other thread executes inside a `synchronized` method of the same `obj`.

Locking

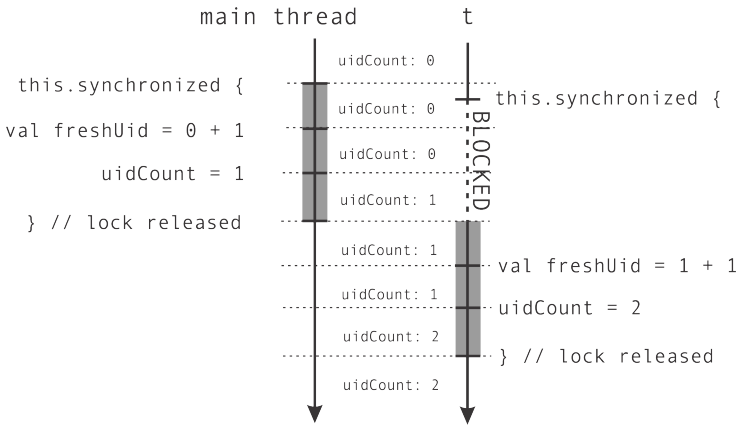
In the call

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

`obj` serves as a *lock*.

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

Locking, Diagrammatically



Monitors

- On the JVM, *every* object can serve as a lock.
- This is needlessly general and expensive on non JVM platforms.
- So future Scala will treat `synchronized` as a method of a trait `Monitor`.
- Only instances of `Monitor` can serve as locks.

In the exercises you will use different implementations of `Monitor`, which test your programs under many different thread schedules.

Revised GetUniqueId

```
object ThreadsGetUID extends App with Monitor {
  var uidCount = 0
  def getUniqueId() = this.synchronized {
    val freshUID = uidCount + 1
    uidCount = freshUID
    freshUID
  }
  ...
}
```

A Memory Model

A simple way to think about concurrent executions is in terms of *interleavings*.

Assume:

- Primitive operations: single-word loads, stores.
- Need to read the fine-print on multi-word values, e.g. `Long`s on a 32 bit architecture.

Then:

- The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

This is called the **Sequential Consistency** memory model.

Reorderings

Unfortunately, modern multicore processors do **not** implement the sequential consistency model.

One problem is caches.

- Each core might have a different copy of shared memory in its private cache.
- Write-back of caches to main-memory happens at unpredictable times.

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

Observing Reorderings

```
object ThreadsReordering extends App {
  for (i <- 9 until 100000) {
    var a, b = false
    var x, y = -1
    val t1 = thread { a = true; y = if (b) 0 else 1 }
    val t2 = thread { b = true; x = if (a) 0 else 1 }
    t1.join()
    t2.join()
    assert(!(x == 1) && (y == 1), s"x = $x, y = $y, i = $i")
  }
}
```

The assertion will fail every once in a while :-(.

An Explanation

Safe Publication

`synchronized` not only ensures atomic execution; it also ensures that writes are visible. Specifically:

- After `obj.synchronized { ... }` was executed by thread `t`, all writes of `t` up to that point are visible to every thread that subsequently executes a `obj.synchronized { ... }`.

Example: Money Transfers

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

First, here is the code to collect log messages:

```
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 class `Account`:

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

(`getUID` will be needed 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) transfer(jane, john, 1) }
val t2 = thread { for (i <- 0 until 100) 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:

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:

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

Thread Coordination

Here's another common task: Implement a (one-place) buffer.

Two classes of threads, *producers*, and *consumers*.

- *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`

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

Question: How to implement the ???s?

Busy Waiting

We could wait by

```
class OnePlaceBuffer[Elem] extends Monitor {
  private var elem: Elem = _
  private var full: Boolean = false
  def put(e: Elem) = while (!tryToPut(e)) {}
  def tryToPut(e: Elem): Boolean = synchronized {
    if (full) 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

Monitors can do more than just locking with `synchronized`.

They also offer 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.

Blocking Implementation

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

Questions:

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

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).

Signals

The Java implementation is not quite the same as the original model of *Monitors* by Per Brinch Hansen, as implemented in the languages Modula and Modula-2.

The original model has `synchronized` but not `wait` and `notify` and `notifyAll`.

Instead, it introduces *signals*.

A signal can be thought of implementing the following class:

```
class Signal {
  def wait(): Unit   // wait for someone to send
  def send(): Unit   // wakes up the first waiting thread
}
```

Buffer Implementation with Signals

```
class OnePlaceBuffer[Elem] extends Monitor {
  var elem: Elem = _
  var full = false
  val isEmpty, isFull = new Signal
  def put(e: Elem): Unit = synchronized {
    while (full) isEmpty.wait()
    isFull.send()
    full = true; elem = e
  }
  def get(): Elem = synchronized {
    while (!full) wait()
    isEmpty.send()
    full = false; elem
  }
}
```

Can Signals be Implemented in a Library?

Let's try:

```
class Signal {
  def send(): Unit = synchronized { notify() }
  def wait(): Unit = synchronized { wait() }
}
```

(We'd have to rename `wait` because it is `final` in `AnyRef`, but never mind for now).

What do you expect to see when combining this with one-place buffers?

1. correct and faster implementation
2. wrong behavior
3. deadlocks

Stopping Threads

- The JVM contains a method `stop()` on class `Thread`, but it is deprecated, and should not be used.
- [Why is Thread#stop() bad, yet killing a process is acceptable?]
- The best thing to do instead is to notify threads that they should stop by setting a global variable, which needs to be polled regularly.

Volatile Variables

Sometimes we need only safe publication instead of atomic publication.

In these situations there's a cheaper solution than using `synchronized`: we can use a *volatile field*:

Example:

```
@volatile var found = _
val t1 = thread { ... ; found = true }
val t2 = thread { while (!found) ... }
```

Guarantees of Volatile

Making a variable `@volatile` has two effects:

- Assignments to the variable will not be reordered wrt to other statements in the thread.
- Assignments to the variable are visible immediately to all other threads.

Using Volatile: Parallel Search

Let's assume we want to search in parallel a number of pages for an occurrence of a pattern.

Once a pattern is found, all threads should stop searching.

This can be achieved as follows:

class Page(val txt: String, var position: Int)

```
object Volatile extends App {
  val pages = for (i <- 1 to 5) yield
    new Page("Na" * (100 - 20*i) + " BatMan!", -1)

...
```

Parallel Search ctd

```
    @volatile var found = false
    for (p <- pages) yield thread {
      var i = 0
      while (i < p.txt.length && !found)
        if (p.txt(i) == '!') {
          p.position = i
          found = true
        }
        else i += 1
    }
    while (!found) Thread.`yield`()
    log(s"results: ${pages.map(_.position)}")
  }
```

What happens if the `@volatile` is left out?

Memory Models in General

A memory model is an abstraction of the hardware capabilities of different computer systems.

It essentially abstracts over the underlying systems *cache coherence protocol*.

Memory models are non-deterministic, to allow some freedom of implementation in compiler and hardware.

Every memory model is a compromise, since it has to trade off between:

- more guarantees = easier to write concurrent programs,
- fewer guarantees = more capabilities for optimizations.

The Java Memory Model

The 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

Consider:

```
val t1 = thread {
  x = 1
  lock.synchronized {
    y = 2
  }
```

```
val t2 = thread {
  println(x)
  lock.synchronized { println(y) }
  println(x)
}
```