Concurrency Building Blocks

Concurrent Programming

Martin Odersky and Viktor Kuncak

Recap: Concurrency Operations

- `thread(...)` to spawn a thread
- `t.join()` to wait for `t` to finish
- `synchronized(...)` for atomic execution
- `wait`, `notify`, `notifyAll` for signalling inside monitors
- @volatile for safe publishing of fields

Errata: Context Switch Times

- A context switch costs on the order of 1000ns.
- This is one micro-second (μs), not one milli-second (ms), as previously claimed.
- Even at one μs it's a relatively expensive operation.

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.

Recap: Sequential Consistency

> *The result of anyan execution is the same as if the operations of all threads were executed in some sequential order, and the operations of each individual thread appear in this sequence in the order specified by its program.*

This is a relatively strong guarantee

\+ Simple to reason about

\- Inefficient to implement

Recap: The Java Memory Model

Define 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.

Ctd: The Java Memory Model

Now,

- A program point of a thread t is *guaranteed* to see all actions that *happen_before* it.
- It *may* also see actions that can occur before it in the sequential consistency (interleaving) model.

The second point is the source of non-determinism of the model.

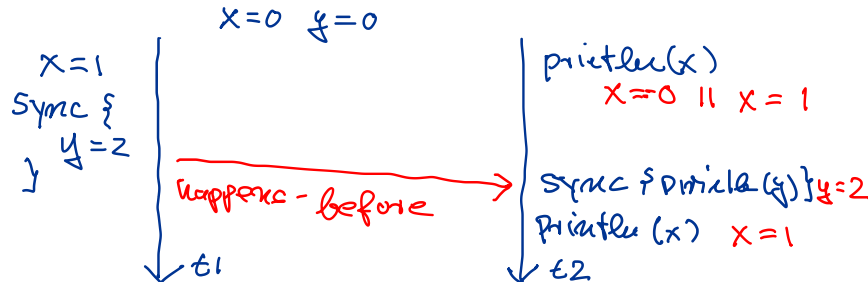
Example: The Java Memory Model

Consider:

```
val x = 0; val y = 0;
val t1 = thread {
  x = 1
  lock.synchronized {
    y = 2
  }
```

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

Then we have:



Executors

Thread creation is expensive.

Therefore, one often multiplexes threads to perform different tasks.

The JVM offers abstractions for that: `ThreadPools` and `Executors`.

- Threads become in essence the workhorses that perform various tasks presented to them.
- The number of available threads in a pool is typically some polynomial of the number of cores `$N$`. (e.g., N^2)
- That number is independent of the number of concurrent activities to perform.

Runnables

A task presented to an executor is encapsulated in a `Runnable` object.

Its signature is as follows.

```
trait Runnable {
  def run(): Unit // actions to per performed by the task
}
```

Forking a Task

Here's how a task gets passed to the `ForkJoinPool` executor:

```
import java.util.concurrent.ForkJoinPool
object ExecutorsCreate extends App {
  val executor = new ForkJoinPool
  executor.execute(new Runnable {
    def run() = log("This task is run asynchronously.")
  })
  Thread.sleep(500)
}
```

`ForkJoinPool` is a system provided thread-pool which also handles tasks spawned by Scala's implementation of parallel collections (and Java 8's implementation of streams).

Notes

1. Tasks are run by passing `Runnable` objects to an executor.
2. There is no way to await the end of a task, as we did with `t.join()` for threads.
3. Instead, we pause the main thread to give the executor threads time to finish.
4. Alternatively, we could do the following

```
import java.util.concurrent.TimeOut
executor.shutdown()
executor.awaitTermination(60, TimeUnit.SECONDS)
```

Methods `awaitTermination` and `shutdown` are defined in the Java interface `ExecutorService`, which is also inherited by `ForkJoinPool`.

Execution Contexts

The `scala.concurrent` package defines the `ExecutionContext` trait and object which is similar to `Executor` but more specific to Scala.

Execution contexts are passed as implicit parameters to many of Scala's concurrency abstractions.

Here's how one runs a task using the default execution context:

```
import scala.concurrent
object ExecutionContextCreate extends App {
  val ectx = ExecutionContext.global
  ectx.execute(new Runnable {
    def run() = log("This task is run asynchronously.")
  })
  Thread.sleep(500)
}
```

Hiding Boilerplate

To hide the boilerplate around executing tasks we define this utilitity method in the package object of the package where examples are run (i.e. in file `concurrent/package.scala`).

```
package object concurrent {
  ...
  def execute(body: => Unit) =
    scala.concurrent.ExecutionContext.global.execute(
      new Runnable { def run() = body })
```

Simplified Task Execution

With the helper method, we can spawn tasks as follows:

```
package concurrent
import scala.concurrent
object SimpleExecute extends App {
  execute{ log("This task is run asynchronously.") }
  Thread.sleep(500)
}
```

Atomic Primitives

So far, we have based concurrent operations on `synchronized`, `wait`, `notify` and `notifyAll`.

These are all complex operations which require support from the OS scheduler.

We now look at the primitives in terms of which these higher-level operations are implemented.

On the JVM these primitives are based on the notion of an *atomic variable*.

Atomic Variables

An atomic variable is a memory location that supports **linearizable** operations.

A **linearizable** operation is one that appears instantaneously with the rest of the system. We also say the operation is performed **atomically**.

Classes that create atomic variables are defined in package

```
java.util.concurrent.atomic
```

They include

```
AtomicInteger, AtomicLong, AtomicBoolean, AtomicReference
```

Using Atomic Variables

Here's how `getUniqueId` can be defined without `synchronized`:

```
import java.util.concurrent.atomic._

object AtomicUid extends App {
  private val uid = new AtomicLong(0L)
  def getUniqueId(): Long = uid.incrementAndGet()
  execute {
    log(s"Got a unique id asynchronously: ${getUniqueId()}")
  }
  log(s"Got a unique id: ${getUniqueId()}")
}
```

Atomic Operations

The `incrementAndGet` method is a complex, linearizable operation.

It reads the value `x` of `uid`, computes `x + 1`, stores the result back in `uid` and returns it.

No intermediate result can be observed by other threads.

Other linearizable method offered by `AtomicLong`:

```
def getAndSet(newValue: Long)
```

This reads the current value of the atomic variable, sets the new value, and returns the previous value.

Compare and Swap

Atomic Operations are usually all based on the **compare-and-swap (CAS)** primitive.

CAS is available as a `compareAndSet` method on atomic variables.

It is usally implemented by the underlying hardware as a machine instruction, but we can think of its implementation as follows:

```
class AtomicLong {
  ...
  def compareAndSet(expect: Long, update: Long) = this.synchronized {
    if (this.get == expect) { this.set(update); true }
    else false
  }
```

That is, `compareAndSet` atomically sets the value to the given updated value if the current value equals the expected value.

Using CAS

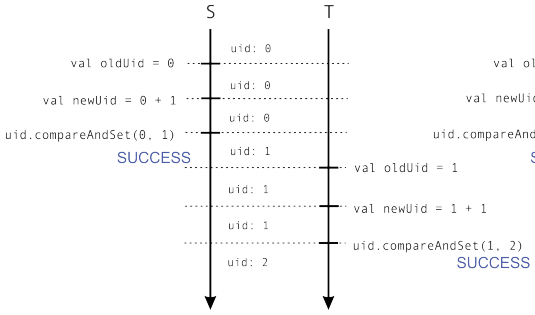
Let's implement `getUniqueId` using CAS directly:

```
@tailrec def getUniqueId(): Long = {
  val oldUid  = uid.get
  val newUid = oldUid + 1
  if (uid.compareAndSet(oldUid, newUid) newUid
  else getUniqueId()
}
```

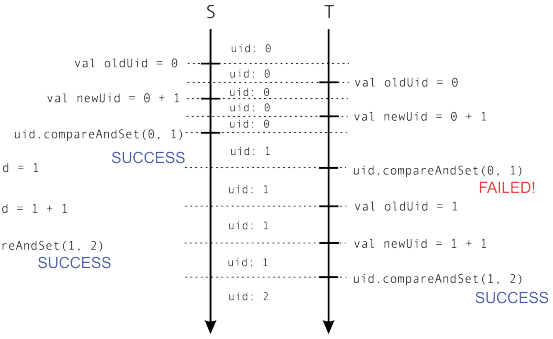
This uses the following general schema:

1. Read the old value from the atomic variable
2. Compute the new value
3. Try do do a CAS.
   - If succesful, the value was updated, and we are done.
   - If unsuccesful, a some other thread had stored a new value in the meantime. In that case, try the same operation again.

Execution Diagram



Execution without retries



Execution with retries

Programming Without Locks

Locks as implemented by `synchronized` are a convenient concurrency mechanism.

But they are also problematic

- possibility of deadlock
- possibility to arbitarily delay other threads if a thread executes a long-running operation in a `synchronized`, or if it gets pre-empted by the OS.

With atomic variables and their **lock-free operations**, we can avoid these problems.

A thread executing a lock-free operation cannot be pre-empted by the OS, so it cannot temporarily block other threads.

Simulating Locks

However, not all operations composed from atomic primitives are lock-free.

In fact, we can implement `synchronized` only from atomic operations:

```
private val lock = new AtomicBoolean(false)
def mySynchronized(body: => Unit): Unit = {
  while (!lock.compareAndSet(false, true)) {}
  try body
  finally lock.set(false)
}
```

So, atomic operations can be used to model locks and locking operations.

Lock-Free Operations

Here is a way to define lock-freedom without looking at implementation primitives:

> *An operation* op *is* lock-free *if whenever there is a set of threads executing* op *at least one thread completes the operation after a finite number of steps, regardless of the speed in which the different threads progress.*

Example

Is this operation lock-free?

```
@tailrec def getUniqueId(): Long = {
  val oldUid  = uid.get
  val newUid = oldUid + 1
  if (uid.compareAndSet(oldUid, newUid) newUid
  else getUniqueId()
}
```

Problem: It is (tail-)recursive, so could take arbitrarily long.

Example

Is this operation lock-free?

```
@tailrec def getUniqueId(): Long = {
  val oldUid  = uid.get
  val newUid = oldUid + 1
  if (uid.compareAndSet(oldUid, newUid) newUid
  else getUniqueId()
}
```

Problem: It is (tail-)recursive, so could take arbitrarily long.

However, the recursive call is made only if *some other thread* completed the operation (in a finite number of steps).

This observation proves lock-freedom.

In general, lock-freedom is quite hard to reason about.

Lazy Values

You know about lazy values in Scala.

```
lazy val x: T = E
```

We now look at how to implement them so that several threads can use a lazy value.

Would like to achieve the following:

- The first thread that demands a lezy value computes it.
- Other threads will block unitl the value is computed by the first thread.
- That way, every lazy value initializer is executed at most once.

Lazy Values (1)

Here is a first, naive implementation of the definition of `x` above:

```
private var x_defined = false
private var x_cached: T = _

def x: T = {
  if (!x_defined) {
    x_cached = E
    x_defined = true
  }
  x_cached
}
```

What is wrong with this implementation?

Lazy Values (2)

The previous implementation of `x` is not thread-safe.

- E might be evaluated several times
- We might access `x` without a value (possible reordering between `x_defined` and `x_cached` assignments).

Lazy Values (3)

Here's a safer implementation:

```
private var x_defined = false
private var x_cached: T = _

def x: T = this.sychronized {
  if (!x_defined) {
    x_cached = E
    x_defined = true
  }
  x_cached
}
```

What are potential problems with this implementation?

Lazy Values (4)

The `synchronized` call on every access is quite costly. Here is a faster implementation:

```
@volatile private var x_defined = false
private var x_cached: T = _
def x: T = {
  if (!x_defined) this.sychronized {
    if (!x_defined) { x_cached = E; x_defined = true }
  }
  x_cached
}
```

This pattern is called *double locking*. How does it work?

Lazy Values (4)

The `synchronized` call on every access is quite costly. Here is a faster implementation:

```
@volatile private var x_defined = false
private var x_cached: T = _
def x: T = {
  if (!x_defined) this.sychronized {
    if (!x_defined) { x_cached = E; x_defined = true }
  }
  x_cached
}
```

This pattern is called *double locking*. How does it work?

If `x_defined` is set, some thread must have completed the `synchronized` to do this.

Lazy Values (5)

The previous implementation of lazy values is what *scalac* currently implements.

It's not without problems though:

- it is not lock free
- it uses the current object as a lock, which might conflict with application-defined locking.
- it is prone to deadlocks.

Consider:

```
object A {
  lazy val x = B.y
}
```

```
object B {
  lazy val y = A.x
}
```

What does this do in a sequential setting? What can it do in a concurrent one?

Better Lazy Values

An alternative implementation of lazy values uses two flag bits per lazy value:

```
private var x_evaluating = false
private var x_defined = false
private var x_cached = _
```

Better Lazy Values (2)

The implementation of the getter `x` is as follows:

```
def x: T = {
  if (!x_defined) {
    this.synchronized {
      if (x_evaluating) wait() else x_evaluating = true
    }
    if (!x_defined) {
      x_cached = E
      this.sychronized {
        x_evaluating = false; x_defined = true; notifyAll()
      }
    }
  }
  x_cached
}
```

Better Lazy Values (3)

This is the implementation scheme used in the new Scala compiler, `dotty`, developed at LAMP.

Notes:

- The evaluation of expression `E` happens outside a monitor, therefore no arbitary slowdowns.
- Two short `synchronized` blocks instead of one arbitrary long one.
- No interference with user-defined locks.
- lazy val/lazy val deadlocks are still possible with the new implementation, but only in cases where sequential execution would give an infinite loop.

Exercise: Implement the new getter implementation using only atomic operations.

Using Collections Concurrently

Operations on mutable collections are usually not thread-safe.

Example:

```
import scala.collection._
object CollectionBad extends App {
  val buffer = mutable.ArrayBuffer[Int]()
  def add(numbers: Seq[Int]) = execute {
    buffer ++= numbers
    log(s"buffer = $buffer")
  }
  add(0 until 10)
  add(10 until 20)
  Thread.sleep(1000) }
```

This can give arbitrary interleavings of elements in `buffer` and it can also crash.

Using Synchronization

A safe way to deal with this is to use `synchronized`:

```
val buffer = mutable.ArrayBuffer[Int]()
def add(numbers: Seq[Int]) = execute {
  buffer.synchronized {
    buffer ++= numbers
    log(s"buffer = $buffer")
  }
}
```

Concurrent Collections

Using `sycnhronized` often leads to too much blocking because of *coarse-grained locking*.

To gain speed, we can use or implement special *concurrent collection* implementations.

Example: Concurrent queues, with operations `append`, `remove`.

- `Append` needs access to the end of the queue
- `Remove` needs access to the beginning of the queue

Concurrent Queues

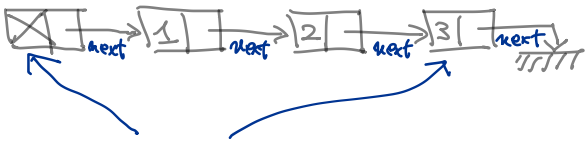
We now develop a lock-free concurrent queues implementation.

First step: A sequential implementation.

A queue has two fields:

- `head` points to a dummy node *before* the first element
- `last` points to the last element (or the dummy node if the queue is empty).

Empty Queue



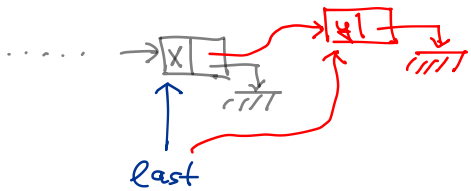
Non-empty queue

Sequential Queue Implementation

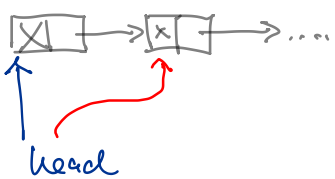
```
object SeqQueue {
  private class Node[T](var next: Node[T]) {
    var elem: T = _
  }
}
class SeqQueue[T] {
  import SeqQueue._
  private var last = new Node[T](null)
  private var head = last
```

Sequential Queue Implementation (ctd)

```
final def append(elem: T): Unit = {
  val last1 = new Node[T](null)
  last1.elem = elem
  val prev = last
  last = last1
  prev.next = last1
}
```




```
final def remove(): Option[T] =
  if (head eq last) None
  else {
    val first = head.next
    head = first
    Some(first.elem)
  }
```



Concurrent Queue Implementation

Idea: make `head` and `last` atomic (reference) variables.

Problem: `append` needs to atomically update *two* variables:

```
last = last1
prev.next = last1
```

But CAS can only work on one variable at a time.

Solution:

- Use one CAS (for `last = last1`) and fix the other assignment when succesful.
- This leaves a window of vulnerability where `prev.next == null` instead of `prev.next == last1`.
- We need to detect and compensate for this in `remove`.

Concurrent Queue: Setup

```
import java.util.concurrent.atomic._
import scala.annotation.tailrec

object ConcQueue {
  private class Node[T](@volatile var next: Node[T]) {
    var elem: T = _
  }
}

class ConcQueue[T] {
  import ConcQueue._
  private var last = new AtomicReference(new Node[T](null))
  private var head = new AtomicReference(last.get)
```

Concurrent Queue: Append

```
@tailrec final def append(elem: T): Unit = {
  val last1 = new Node[T](null)
  last1.elem = elem
  val prev = last.get
  if (last.compareAndSet(prev, last1)) prev.next = last1
  else append(elem)
}
```

Only last two lines are substantially different from the sequential implementation.

Concurrent Queue: Remove

```
@tailrec final def remove(): Option[T] =
  if (head eq last) None
  else {
    val hd = head.get
    val first = hd.next
    if (first != null && head.compareAndSet(hd, first)) Some(first.elem)
    else remove()
  }
```

Again, only the last two lines are substantially different from the sequential implementation.

Question

Are `append` and `remove` of `ConcQueue` lock-free operations?

Recall the definition:

> *An operation* op *is* lock-free *if whenever there is a set of threads executing* op *at least one thread completes the operation after a finite number of steps, regardless of the speed in which the different threads progress.*

Question (2)

What about if we change `remove` to the following implementation?

```
@tailrec final def remove(): Option[T] = {
  val hd = head.get
  val first = hd.next
  if (first == null) None
  else if (head.compareAndSet(hd, first)) Some(first.elem)
  else remove()
}
```

Concurrent Queues in the Standard Library

Multiple implementations in package java.util.concurrent implement interface `BlockingQueue`:

```
ArrayBlockingQueue   // bounded queue, similar to exercise last week

LinkedBlockingQueue  // unbounded queue, similar to implementation here
```

Concurrent Sets and Maps

The Scala library also provides implementations of concurrent sets and maps.

These make use of the underlying data structure for efficiency.

For instance for Hashmaps: