

# SCL for CSO Programmers

Gavin Lowe

April 13, 2022

This document describes SCL (Scala Concurrency Library) for those familiar with CSO (Concurrent Scala Objects).

SCL is heavily influenced by Bernard Sufrin's CSO. Most of the names of classes and functions are unchanged. However, a few changes have been made, for example to provide a simpler interface.

The emphasis in SCL is on pedagogy, rather than out-and-out performance. SCL aims to provide a fairly minimal interface, essentially just enough for the Concurrent Programming course. In particular, SCL makes much less use of factory methods than CSO.

The classes and top-level functions described below can be obtained by including the line `import ox.scl._` at the top of the file.

## Threads

In SCL, a `ThreadGroup` represents a collection of threads. The declaration `thread{ comp }` creates a computation that, when run, executes `comp`. As with CSO, threads can be combined in parallel using `||`.

SCL syntax	CSO syntax	Comments
<code>thread{comp}</code>	<code>proc{comp}</code>	A thread that, when run, executes <code>comp</code> .
<code>ThreadGroup</code>	<code>PROC</code>	Type of a collection of threads.
<code>p    q</code>	<code>p    q</code>	Parallel composition of <code>p</code> and <code>q</code> .
<code>   (ps)</code>	<code>   (ps)</code>	Parallel composition of the collection of Computations <code>ps</code> .
<code>run(p)</code> or <code>p.run</code>	<code>run(p)</code> or <code>p()</code>	Execute the threads <code>p</code> .
<code>fork(p)</code> or <code>p.fork</code>	<code>p.fork</code>	Executes <code>p</code> in a new JVM thread.

Exceptions are treated slightly differently from in CSO. When `run` is used, if a thread throws a non-`Stopped` exception, then it is caught, all the other threads in the computation are interrupted, and the exception is re-thrown. If a thread throws a `Stopped` exception, then all the other threads in the computation are allowed to terminate, at which point the `Stopped` exception is re-thrown. When `fork` is used, if any exception is thrown, the program exits.

SCL has the same constructs as CSO for handling `Stopped` errors.

SCL syntax	CSO syntax	Comments
<code>repeat{comp}</code>	<code>repeat{comp}</code>	Repeatedly execute <code>comp</code> , exiting cleanly if it throws a <code>Stopped</code> exception.
<code>repeat(guard){comp}</code>	<code>repeat(guard){comp}</code>	Repeatedly execute <code>comp</code> while <code>guard</code> is true, exiting cleanly if it throws a <code>Stopped</code> exception.
<code>attempt{comp}{handler}</code>	<code>attempt{comp}{handler}</code>	Execute <code>comp</code> , but if it throws a <code>Stopped</code> exception, execute <code>handler</code> .

## Channels

SCL makes no distinction between shared and unshared ports. There are just two types of channels: synchronous (`SyncChan`) and buffered (`BuffChan`). SCL does not provide factory methods for channels, so a channel can be constructed with, for example, `new BuffChan[A](size)`<sup>1</sup>.

As with CSO, a channel comprises an `InPort` and an `OutPort`. The syntax for standard sends and receives is unchanged. The syntax for time-bounded sends and receives is changed slightly (see below).

The syntax for closing channels is mostly unchanged, except the `closeIn` operation has been removed (it was equivalent to `close`). `closeOut` has been renamed to `endOfStream` (this operation normally signals the end of a stream). In addition, operations `isClosed` and `reopen` have been added.

SCL syntax	CSO syntax	Comments
<code>new SyncChan[A]</code>	<code>OneOne[A]()</code> , <code>OneMany[A]()</code> , <code>ManyOne[A]()</code> , <code>ManyMany[A]()</code> , <code>N2N[A](m,n)</code>	Creation of synchronous channel.
<code>new BuffChan[A](size)</code>	<code>OneOneBuf[A](size)</code> , <code>ManyManyBuf[A](size)</code> , <code>N2NBuf[A](m,n,size)</code>	Creation of buffered channel with capacity <code>size</code> .
<code>in?()</code>	<code>in?()</code>	Receive from <code>InPort in</code> .
<code>out!x</code>	<code>out!x</code>	Send <code>x</code> on <code>OutPort out</code> .
<code>c.close</code> , <code>in.close</code>	<code>c.close</code> , <code>in.closeIn</code>	Fully close the channel.
<code>out.endOfStream</code>	<code>out.closeOut</code>	Close the channel for sending, normally signalling the end of the stream.
<code>c.isClosed</code>	—	Is the channel closed?
<code>c.reopen</code>	—	Re-open a closed channel.
<code>in.receiveWithin(millis)</code>	<code>in.readBefore(nanos)</code>	Receive from <code>in</code> , or timeout after <code>millis ms/nanos ns</code> , returning an <code>Option</code> value.
<code>out.sendWithin(millis)(x)</code>	<code>out.writeBefore(nanos)(x)</code>	Send <code>x</code> on <code>out</code> , or timeout after <code>millis ms/nanos ns</code> , returning a boolean value.

<sup>1</sup>The type `A` of data for a `BuffChan` must have an associated `ClassTag`. When `A` is a parametric type parameter, it is enough to give the type bound `A: scala.reflect.ClassTag`. If the parameter `A` is subsequently instantiated with a concrete type `T`, it may be necessary to provide an additional argument `scala.reflect.ClassTag(classOf[T])`.

## Alternation

The syntax for alternation (`alt` and `serve`) is largely unchanged. Unlike CSO, parentheses must *not* be placed around the guard and port. The following example (a two-place buffer) illustrates most of the syntax.

```
var x = -1; var empty = true
serve(
  !empty && out !=> {x} ==> { empty = true }
  | empty && in ==> { v ==> x = v; empty = false }
  | !empty && in ==> { v ==> out!x; x = v }
)
```

Alternations in SCL have slightly fewer restrictions than in CSO. It is possible for a port to be shared between an `alt` and a non-`alt`. A port may be simultaneously enabled in several branches of the same `alt` (all but one instance will be ignored). However, the following restrictions remain:

- A port may not be simultaneously enabled in two `alts`. This restriction could be removed without too much difficulty.
- A channel may not have both of its ports simultaneously enabled in `alts`. Such usage could lead to a deadlock.

The implementation checks the above restrictions, throwing an exception if they are broken.

Unlike CSO, the expressions defining the ports are re-evaluated on each iteration of a `serve`. This means that a definition like the following has the expected effect.

```
serve( (if(flag) c1 else c2) ==> { x ==> flag = !flag; ... } | ... )
```

## Monitors/locks

JVM monitors are outside CSO/SCL, so are unchanged.

In SCL, the class `Monitor` is replaced by a class `Lock`, although each `Lock` supports the functionality of a CSO `Monitor`. A `Lock` can be acquired or released. A computation can be protected by a `Lock` using `lock.mutex{comp}`: this ensures `comp` is executed under mutual exclusion on the lock.

Conditions can be created, associated with a `Lock`, and used as with CSO monitors.

SCL syntax	CSO syntax	Comments
<code>new Lock</code>	<code>new Monitor</code>	Creation of lock or monitor.
<code>lock.acquire</code>	—	Acquire the lock.
<code>lock.release</code>	—	Release the lock.
<code>lock.mutex{comp}</code>	<code>monitor.withLock{comp}</code>	Execute <code>comp</code> under mutual exclusion on the lock/monitor.
<code>lock.newCondition</code>	<code>monitor.newCondition</code>	Obtain a new condition on the lock/monitor.
<code>cond.await</code>	<code>cond.await</code>	Wait for a signal on <code>cond</code> .
<code>cond.await(test)</code>	<code>cond.await(test)</code>	Wait for <code>test</code> to become true, rechecking when a signal on <code>cond</code> is received.
<code>cond.signal</code>	<code>cond.signal</code>	Send a signal to a thread waiting on <code>cond</code> .
<code>cond.signalAll</code>	<code>cond.signalAll</code>	Send a signal to each thread waiting on <code>cond</code> .

## Barrier synchronisations

The implementation of a barrier for  $n$  threads runs in time  $O(\log n)$  (whereas the CSO implementation was  $O(n)$ ). The downside of this is that each call to `sync` requires an identity parameter in the range  $[0..n)$ , with different threads providing different identities.

Combining barriers, unlike with CSO, do not require the starting value for the accumulation.

SCL syntax	CSO syntax	Comments
<b>new</b> Barrier( $n$ )	<b>new</b> Barrier( $n$ )	Creation of barrier object for $n$ threads.
barrier.sync( $me$ )	barrier.sync	Synchronisation (by thread with identity $me$ ).
<b>new</b> Combining-Barrier( $n$ , $f$ )	<b>new</b> Combining-Barrier( $n$ , $f$ , $e$ )	Creation of combining barrier for $n$ threads, with combining function $f$ (and starting value $e$ ).
barrier.sync( $me$ , $x$ )	barrier.sync( $x$ )	Synchronisation (by thread with identity $me$ ) providing input $x$ .
<b>new</b> AndBarrier( $n$ )	<b>new</b> lock.AndBarrier( $n$ )	Creation of conjunctive combining barrier.
<b>new</b> OrBarrier( $n$ )	<b>new</b> lock.OrBarrier( $n$ )	Creation of disjunctive combining barrier.

## Semaphores

Semaphores in SCL are very similar to as in CSO. An exception is that the `up` operation *requires* that the semaphore is in the down state (a call of `up` when the semaphore is already up is normally a programming error).

SCL syntax	CSO syntax	Comments
<b>new</b> Semaphore(isUp)	Semaphore(isUp)	Creation of semaphore with state given by isUp.
<b>new</b> MutexSemaphore	MutexSemaphore()	Creation of semaphore in the up state, e.g. for mutual exclusion.
<b>new</b> Signalling-Semaphore	SignallingSemaphore()	Creation of semaphore in the down state, e.g. for signalling.
<b>new</b> Counting-Semaphore(permits)	Counting-Semaphore(permits)	Counting semaphore, with <code>permits</code> permits available initially.
sem.up	sem.up	Raise the semaphore.
sem.down	sem.down	Lower the semaphore.

## Linearizability testing

The linearizability testing framework is incorporated within SCL. (It was a separate package from CSO.) The interface has been slightly simplified from previously.

Each worker operating on the concrete datatype should have signature

```
def worker(me: Int, log: LinearizabilityLog[S, C]) = ...
```

where  $S$  is the type of the sequential specification object, and  $C$  is the type of the concurrent object being tested. Each worker performs and logs operations using commands of the form

```
log(concOp, string, seqOp)
```

where `concOp: C => A` is the operation performed on the concurrent object, `seqOp: S => (A,S)` is the corresponding operation on the specification object, and `string` is a `String` that describes the operation (with different strings for semantically different operations).

The linearizability tester is created and run using commands of the form

```
val tester = LinearizabilityTester[S,C](seqObj, concObj, p, worker _)
assert(tester() > 0)
```

where `seqObj` is the sequential specification object, `concObj` is the concurrent object being tested, `p` is the number of workers to run, and `worker` is as above.