Concurrent Programming

Bernard Sufrin

Hilary Term 2016-17

4: Message Passing

Channels

* Inter-task communication using message passing
  - more intuitively understandable than ad-hoc shared-memory designs using monitors or semaphores
  - *essentially* compositional
  - close to the CSP model of concurrency $\therefore$ easier to reason about formally

"Share memory using communication instead of trying to communicate using shared memory!"

The CSO Programming Model

* Programs are composed of components ("processes"), which communicate using channels.
* Components may be composed in parallel or sequentially.
* Intuitive reasoning about processes is straightforward.
* This programming model was inherited and refined from occam
  (the first language based on Communicating Sequential Processes)

Why CSP?

* The theory encapsulates the fundamental principles of communication.
* Semantic models are mature and sufficiently expressive to enable reasoning about deadlock and livelock.
* There are robust tools available for formal verification of *designs* against specifications.
* Transliteration of (a large class of) CSP designs into programs can be straightforward.

* Think of a channel as a one-directional pipe

* At one end there is an output port, at the other an input port

```
OutPort --------------->---------------- InPort
  chan!v                                   chan?
```

* Values output to one end are input from the other in the same order

* Denoting the sequences of data so far read from the channel c by (terminated) read operations as $c^?$ and that written to it by (terminated) write operations as $c^!$ we can say, in general, that:

$$c^? \textbf{ prefixes } c^!$$

* A *synchronous* channel c satisfies the stronger constraint: $c^? = c^!$

* An *asynchronous* channel (a.k.a. $buffer$) satisfies the slacker constraint.

Channels in CSO

* If T is a type, then OneOne[T] is one type of channel that passes data of type T
* The declaration val chan = OneOne[T] defines chan to be such a channel.
  - Evaluating the command chan!v sends the value v on chan.
  - Evaluating the expression chan?() reads a value from chan and returns it.

  - Communication on a OneOne is *synchronous*:
    the command executed first waits for the other; both then proceed.

* So these two processes are essentially equivalent (if x and v are of type T):

```
{ val chan=OneOne[T];                          |
  proc { chan!v } || proc { x = chan?() }      |    proc { x = v }
}                                              |
```

Example: copy

* The *process-generator* `copy` is parameterised by the channels `in`, and `out`, and by the type T

```
import io.threadcso._

def copy[T](in: OneOne[T], out: OneOne[T]) = proc
{
  while (true) { val x = in?() ; out!x }
}
```

* `copy(i, o)` yields a process that, when run, repeatedly

  inputs a value from the channel `i`,

  then outputs it to the channel `o`

* The analogous CSP process is specified by

$$COPY(in, out) = in?x \rightarrow out!x \rightarrow COPY(in, out).$$

InPorts and OutPorts

* A channel comprises
  - an `InPort` (something from which processes can read) *together with*
  - an `OutPort` (something to which processes can write)

```
trait InPort[+T]  { def ?(): T                         // read
                    def ?[U]:(f: T⇒U): U = f(?()) // read&apply
                    ...
                  }

trait OutPort[-T] { def !(value: T): Unit             // write
                    ...
                  }

trait Chan[T]     extends InPort[T] with OutPort[T]{}
```

* "InPort" and "OutPort" refer to the point of view of the invoking *process*.
* The types `InPort[T]` and `OutPort[T]` are often abbreviated as `?[T]` and `![T]`.

* SyncChan is the "marker" type for a *synchronous* channel.

```
trait SyncChan[T] extends Chan[T]{}
```

* A OneOne is an implementation of a *synchronous* channel.

```
class OneOne[T] extends SyncChan[T] {···}
```

* `copy` uses only the `InPort` of `in` and the `OutPort` of `out`, so we could have written it as:

```
def copy[T] (in: ?[T], out: ![T]) = proc
{
  while(true){ val x = in?() ; out!x }
}
```

* Note how the type signature of *this version* of `copy` makes it clear how the process uses each parameter; whereas the previous version didn't.

* We can partially specify the behaviour of a running $copy$. Denoting the sequences of data input from in (and output to out) by $in^?$ (and $out^!$):

$$out^! \in in^? \overline{\downarrow}\{0, 1\}$$

i.e. the data that has been output to out at any point lags no more than one datum behind that input from in.

Synchronous Channel implementations in CSO

```
class   OneOne[T]
extends SyncChan[T] ···

class   N2N[T](writers: Int, readers: Int)
extends SyncChan[T]
with    SharedOutPort[T]
with    SharedInPort[T] ···
```

* Only one process may read from a `OneOne` channel, and only one process may write to it.
* Several different processes may write to an `N2N` channel.
  ◦ They compete for access to the shared output port.
* Several different processes may read from an `N2N` channel.
  ◦ They compete for access to the shared input port.
  ◦ Each value that is read is read by only one of the processes.

Misuse of unshared ports

* A compiler cannot *completely* enforce the restrictions on sharing at compile-time.

* For example, it will admit

```
val mid = OneOne[Int]
(producer(mid) || producer(mid) || consumer(mid))()
```

* Although CSO implementations are cautious, this and other infractions may go undetected at run-time.

* Here detection depends on the relative speeds of the three running processes
  - The `consumer` may be fast enough keep up with both `producer`s and an implementation will not notice the illegal sharing.
  - One `producer` may write before `consumer` has read the other `consumer`'s last output. and an implementation will notice and throw an `IllegalStateException`.

N2N channels

* N2N channels are synchronous, and can be used to distribute messages from one (or more) writers to one (or more) readers.
* Each written datum is read by exactly one process.
* For example, the script n2ndemo1.scalascript shares an N2N between 10 readers and 10 writers

```
val mid = N2N[Int](15, 15)
for (_ ← 0 until 3) {
    (  (|| (for (i←0 until 15) yield proc { mid!i })) ||
       (|| (for (j←0 until 15) yield proc { print ((j, mid?())) }))
    )()
    println
}
```

It will print something like:

```
(7,1)(2,2)(8,6)(1,3)(3,11)(4,4)(0,0)(11,5)(5,7)(6,9)(9,8)(10,10)(12,14)(13,13)(14,12)
(1,1)(2,12)(3,2)(4,3)(5,4)(6,5)(13,6)(7,7)(8,8)(9,9)(10,10)(11,11)(0,0)(14,14)(12,13)
(1,1)(2,2)(10,3)(3,0)(4,5)(5,6)(6,7)(7,8)(8,9)(9,10)(11,11)(0,12)(12,13)(13,14)(14,4)
```

* The reader-writer pairings are made nondeterministically, and depend on which reader and which writer happen to synchronize "in" the channel.

Naively-designed components

* We're going to look at some examples of concurrent programs built from small components.

* These aren't necessarily sensible concurrent programs: in most cases there are simpler equivalent sequential programs.

* The aim is to get used to thinking about the composition of concurrent components.

* console repeatedly reads a value from in and writes it to standard output:

```
def console[T](in: ?[T]) = proc {
  while (true) { println(in?) }
}
```

* nats sends the natural numbers to out

```
def nats(out: ![Int]) = proc {
  var n=0; while (true) { out!n; n+=1 }
}
```

* alts copies alternate values read from in to out:

```
def alts[T](in: ?[T], out: ![T]) = proc {
  while (true) { out!(in?); in? }
}
```

Example: printing multiples of four

```
import io.threadcso._

object Mults4
{
  def console(in: ?[T])              = proc ...
  def nats(out: ![Int])              = proc ...
  def alts[T](in: ?[T], out: ![T]) = proc ...

  val x1, x2, x4 = OneOne[Int]

  def main(args: Array[String]) =
  (  nats(x1)
  || alts(x1, x2)
  || alts(x2, x4)
  || console(x4)
  )()
}
```

Closing Channels

* The conventional way of cleanly terminating networks of processes is to close the channels between them

* A channel may be closed at any time, using the `close` command. Its output port can be closed using `closeOut` and its input port can be closed using `closeIn`.

* The informal contracts of these methods are:
  - `close` – asserts that no process will write to or read from this channel ever again
  - `closeOut` – asserts that *this* process will not write to this port ever again
  - `closeIn` – asserts that *this* process will not read from this port ever again

* Channel types behave appropriately in response to closing.

* If a process tries to read or write a closed channel, a `Chan.closed` exception is thrown.

* When a producer of data wants to signal the end of the data it is producing, it can close the appropriate port for output, which will close the associated channel if it is unshared. Other components in the system should detect when the channel has been closed and "do the right thing".

* The following version of `alts` detects when either of the channels associated with its ports has been closed; then closes its ports:

```
def alts[T](in: ?[T], out: ![T]) = proc {
  try {
    while (true) { out!(in?); in? }
  } catch {
    case Chan.closed(_) ⇒ { in.closeIn; out.closeOut }
  }
}
```

* The close actions are idempotent on channels; so although *at least one* of `in`, `out` must already have closed for the exception to be thrown: closing the associated port after this does no harm.

Repetition

* The pattern used in `alts` is very common, and there is an equivalent CSO construction

```
repeat{ <command> }
```

that behaves much like

```
while(true){ <command> }
```

but terminates cleanly if `<command>` throws an `io.threadcso.Stopped` exception.

* `Chan.Closed` is such an exception; so an equivalent implementation of `alts` is:

```
def alts[T](in: ?[T], out: ![T]) = proc{
  repeat { out!(in?); in? }
  in.closeIn; out.closeOut
}
```

* The CSO expression

```
attempt { expression₁ } { expression₂ }
```

yields the value of `expression`$_1$ unless its evaluation fails with a `Stopped` exception; in which case it yields the value of `expression`$_2$.

Termination of Process Networks

* Networks of repetitively communicating processes can usually be designed to terminate cleanly:
  - A data source indicates the end of the data stream(s) it is producing by closing its output port(s). If an output port isn't shared then the channel it is associated with is closed by this action.
  - Every component that discovers that one of its channels is closed does the right thing: normally closing each of its ports in the appropriate direction and terminating.

    The closing of an unshared output port for output closes the associated channel, as does the closing of an unshared input port for input.
* When a data source closes as described above, termination generally propagates "downstream" through the network as components discover that their channels are closed.
* Analogously, when a data sink decides that it no longer wishes to consume data from the network it closes its input port(s). Termination generally propagates "upstream" .

canInput, canOutput and attempting reads and writes

* `inport.canInput` returns **`false`** if the given input port cannot ever again be read from.
  - Once the method returns **`false`** for this port, it will never again return **`true`** for it.
  - If the method returns **`true`** it simply asserts that (the channel associated with) the port has not yet been closed: it is no guarantee that input is available and/or a subsequent read from the port will succeed

* `outport.canOutput` returns **`false`** if the given output port cannot ever again be written to.
  - Once the method returns **`false`** for this port, it will never again return **`true`** for it.
  - If the method returns **`true`** it simply asserts that (the channel associated with) the port has not yet been closed: it is no guarantee that a subsequent write to the port will succeed

Avoiding potential deadlock with concurrent reads and writes

* A process generated by `tee` repeatedly inputs values on `in`, then outputs them on both $\mathtt{out}_1$ and $\mathtt{out}_2$.

```
def tee[T](in: ?[T], out₁: ![T], out₂: ![T]) = proc
{
    repeat { val v = in?(); out₁!v; out₂!v }
    in.closeIn; out₁.closeOut; out₂.closeOut
}
```

* In fact it always outputs on the $\mathtt{out}_1$ port before outputting on the $\mathtt{out}_2$ port.

* What happens if we use a `tee` in the context of some larger system that inputs on $\texttt{out}_2$ before inputting on $\texttt{out}_1$? For example:

```
val mid, out₁, out₂, sums = OneOne[Int]

val summer = proc
{ repeat
  { val x = out₂?
    val y = out₁?
    sums!(x+y)
  }
  ...
}

( nats(mid) || tee(mid, out₁, out₂) || summer )()
```

* Deadlock!

Deadlock-avoiding redesign of tee

* Generic components should place as few assumptions as possible upon the network in which they are placed.
* The following version of `tee` performs the outputs concurrently, *i.e.* in either order or simultaneously.

```
                    out₁
                 /|--------> x
x >---in--->{ |
                 \|--------> x
                    out₂
```

```
def tee[T](in: ?[T], out₁: ![T], out₂: ![T]) = proc {
    repeat { val v = in?();
             (proc{out₁!v} || proc{out₂!v})()
           }
    in·closeIn; out₁·closeOut; out₂·closeOut
}
```

* What happens if one of the `out_i` closes?

Termination of Concurrent Compositions

* The concurrent composition p_1 `||` p_2 terminates when both its component processes have terminated: either normally or with exceptions.

  - If neither component terminates with an exception then the composite terminates normally.

  - If one component terminates with an exception and the other terminates normally then the composite re-throws that exception when they have both terminated.

  - If both components terminate with `Stop` exceptions then the composite re-throws the exception that terminated the left component.

  - If both components terminate with exceptions and at least one of them is a non-`Stop` exception then a `ParException` which embodies both exceptions is thrown when they have both terminated.

Case Study: the sameLeaves problem

* We have two trees, annotated with values V at their nodes.
* We wish to see if the leaves of the trees (taken in depth-first order) are annotated with identical values, even though the tree structures may be different (and, indeed, may be generated by different computations).
* Generating subtrees may be computationally expensive, so we want to *terminate as soon as possible*.

* Our solution will be factored into reuseable tree-traversal and stream-equality components.
  - Coupling of components by channels yields laziness.
  - Tree structure doesn't need reifying beforehand.
  - Easily adaptable to comparisons of differing traversals.
  - Very few neurons need to die in the programming of either component!

A sameStreams Component

* A process generated by `sameStreams` terminates when it detects the first difference between the elements arriving on its two input streams, or when one or both streams are closed.

```
def sameStreams[T](inl: ?[T], inr: ?[T], ans: ![Boolean]) = proc
{   var l,  r           = inl·nothing  // last-read value
    var ln, rn          = 0            // number of values read
    val nextPair: PROC = proc {l=inl?(); ln=ln+1} || proc {r=inr?(); rn=rn+1}
    var same            = true
    repeat (same) { nextPair(); same=l==r }
    ans!(same && ln==rn)
    ans·closeOut; inl·closeIn; inr·closeIn
}
```

* Correctness: ln is the number of inputs so far read from inl, likewise rn is the number of inputs so far read from inr. If both inl and inr become unreadable (because both streams have been closed) on the same round of $nextPair$ then the **repeat** will terminate with $ln == rn$. If only one becomes unreadable, then the **repeat** will terminate with $ln \neq rn$.

The Tree Trait

```
trait Tree[V]
{  def subtrees(): Seq[Tree[V]]
   def value:      V
   def isLeaf:     Boolean

   private def  depthFirstTo(out: ![V]) =
           if   (isLeaf) out!value
           else for (t←subtrees()) t·depthFirstTo(out)

   def depthFirst(out: ![V]) =
   { attempt { this·depthFirstTo(out) } {}
     out·closeOut
   }
}
```

* $depthFirst(out)$ outputs leaf values depth-first on port out, then closes the port
* A channel-closed exception generated while it runs will be caught in the **attempt** construct.

* $SameLeaves(tl, tr)$ can now be implemented as a network of three processes

```
def sameLeaves[V](tl: Tree[V], tr: Tree[V]): Boolean =
{   val left, right = OneOne[V]
    val answer      = OneOneBuf[Boolean](1)
    (  proc { tl.depthFirst(left)  }
    || proc { tr.depthFirst(right) }
    || sameStreams(left, right, answer)
    )()                                     // Start the network
    return answer?()                        // Collect the answer
}
```

Discussion: The `answer` channel is buffered; why can't it be synchronous?

Discussion: What happens when two trees with distinct leaf sequences are compared?

Buffers

Sometimes we want to use a channel, say `c`, to pass data between processes, but have no need for the `c!` to be synchronized with the `c?`. In other words, we want an *asynchronous* channel or a *buffer*, where the writer can send data, even if the reader is not ready.

The declaration

```
val c = OneOneBuf[T](size)
```

// $size > 0 \Rightarrow 0 \leq \#c^{!} - \#c^{?} \leq size$

defines `c` to be an asynchronous channel, with capacity `size`, passing data of type `T`. Data can be sent and received over `c` in the same way as for a `OneOne` channel except that the writer can at any time have sent up to `size` values more than the reading process has received.

Because `OneOneBuf` is a subtype of `Chan`, `c` can be used as a parameter for any component defined using `InPort`s and/or `OutPort`s.

`OneOneBuf` has a `OneOne` sharing discipline (but its implementation may not be dynamically checked).

Component generators

* Many forms of concurrent process network use the same patterns of interprocess communication again and again.

* In this section we introduce and demonstrate a handful of the "plumbing component" generators provided by CSO to facilitate the construction of such process networks.

* Some of the components will reminiscent of those that arise in functional programs when lists are used to communicate between functional modules.

* Warning: The demonstrations shown below will result in rather inefficient programs for doing rather simple tasks. They are intended to help us get to grip with issues such as network termination, not as a pattern for building small programs.

zipWith

* A `zipwith` processes a pair of input streams in tandem

```
            inl    __
         x >----->|  \  out
            inr   | f }-----> f(x,y)
         y >----->|__/
```

```
def zipwith[L,R,O](f:(L, R)⇒O) (inl: ?[L], inr: ?[R], out: ![O]) = proc
{ var l = inl·nothing // null value of type L
  var r = inr·nothing // null value of type R
  repeat {   (proc { l = inl? } || proc { r = inr? })();
             out!f(l, r)
         }
  inl·closeIn; inr·closeIn; out·closeOut
}
```

* Note that the inputs are read in parallel. Why?

Discussion: what exactly happen if an `in`$_x$ closes?

map

* The following process applies the function f to its inputs, before outputting the result:

```
x >---[f]---> f(x)
```

```
def map[I,O](f: I ⇒ O)(in: ?[I], out: ![O]) = proc
{ repeat { out!(f(in?)) }
  out·closeOut; in·closeIn
}
```

Demonstration: Generating naturals

* Here's a *circuit* to generate the natural numbers, based upon the identity

```
nats = 0 :  map (+1) nats
```

```
                                out
                  nats  /|-------->
+-->prefix 0----->{ |
|                       \|--------+
|  succs                     mid  |
+----------[+1]<-------------+
```

* `prefix` copies data, prefixing it with the given value:

```
def prefix[T] (v: T)(i: ?[T], o: ![T]) = proc
{ attempt { o!v; repeat { o!(i?) } } {}
  i·closeIn; o·closeOut
}
```

* Here's the realization of the circuit as a network of CSO processes using stock components.

```
object Nats
{ import io.threadcso._
  import io.threadcso.component.{prefix,map,console,tee}

  val mid, nats, succs, out = OneOne[Int]

  val circuit = (  prefix(0)(succs, nats)          || tee(nats, out, mid)
                || map((x:Int) ⇒ x+1)(mid,succs) || console(out)
                )

  def main(args: Array[String]) = circuit()
}
```

* Exercise/experiment: what would happen if the `out` channel were closed after console had printed a few dozen numbers?

Aside: zipwith revisited for efficiency

* The previous version of `zipwith` creates a new concurrent process *on every iteration* (to do the inputs) .

```
repeat {  (proc { l = inl? } || proc { r = inr? })()          //**
           out!f(l, r)                                         //**
       }
```

* In CSO it is a little more efficient to create and name just one process, and re-use it

```
def zipwith[L,R,O](f:(L, R)⇒O) (inl: ?[L], inr: ?[R], out: ![O]) = proc {
    var l = inl.nothing
    var r = inr.nothing
    val doInputs = proc { l = inl? } || proc { r = inr? }  //**
    repeat { doInputs(); out!f(l, r) }                     //**
    inl.close; inr.closeIn; out.closeOut
}
```

* Such transformations pay dividends when there are *many* concurrent components in an iterated process.

Demonstration: an integrator component

```
               in                        out
[x,y,z,...] >------>|\    mid     /|-------> [x,x+y,x+y+z,...]
                    |+}--------->{ |
                +-->|/            \|--+
            addl|                     |back
                +------<prefix 0<-----+
```

* A call of `integrator(···)` composes and returns a network of processes, with hidden internal connections that is started in the usual way – *i.e.* by running it.

```
type Num = ···

def integrator(in: ?[Num], out: ![Num]): PROC =
{ val mid, back, addl = OneOne[Num] // internal connections
  (  zipwith ((x: Num, y: Num)⇒x+y) (in, addl, mid)
  || tee (mid, out, back)
  || prefix(0)(back, addl)
  )
}
```

Aside: process-generation v. process generation and start

* Don't confuse the `integrator` process generator with the following procedure that composes and immediately starts an integrator network, terminating when the network's components have terminated.

```
def runIntegrator(in: ?[Numeric], out: ![Numeric]): Unit =
{ val mid, back, addl = OneOne[Numeric] // internal connections
  (  zipwith ((x:Int, y:Int)⇒x+y) (in, addl, mid)
  || tee (mid, out, back)
  || prefix(0)(back, addl)
  )()
}
```

* The two are, of course, related: when executed and observed via i and o and termination:
  ◦ the expression $runIntegrator(i, o)$ is indistinguishable from $integrator(i, o)()$
  ◦ the expression $integrator(i, o)()$ is indistinguishable from $\mathbf{proc}\{runIntegrator(i, o)\}()$

Demonstration: Sorting networks

An n-channel sorting network reorders the data on its channels.

A 2-channel `exchanger` repeatedly inputs pairs of data from its two input channels, `a,b` and outputs an ordered permutation of them on its two output channels `a',b'`.

```
a-->[\/]-->a'
b-->[/\]-->b'
```

Here is a four-channel sorting network, composed of 5 (2-channel) `exchangers`.

```
a-->[\/]--------->[\/]------------>a'''
b-->[/\]---+  +-->[/\]--+
           |  |         |
           |  |         +-->[\/]-->b'''
c-->[\/]------+         +-->[/\]-->c'''
d-->[/\]-+ |            |
         | +---->[\/]---+
         +------>[/\]------------->d'''
```

Exercise (??) Show how to implement an exchanger

Case Study: Generating Primes

* We will construct a network of processes that generates primes

* The construction is inspired by the "Sieve of Eratosthenes"
  - Generate all the natural numbers starting from 2
  - Repeatedly:
    * Output the first remaining number, as a prime;
    * Delete all multiples of that number.

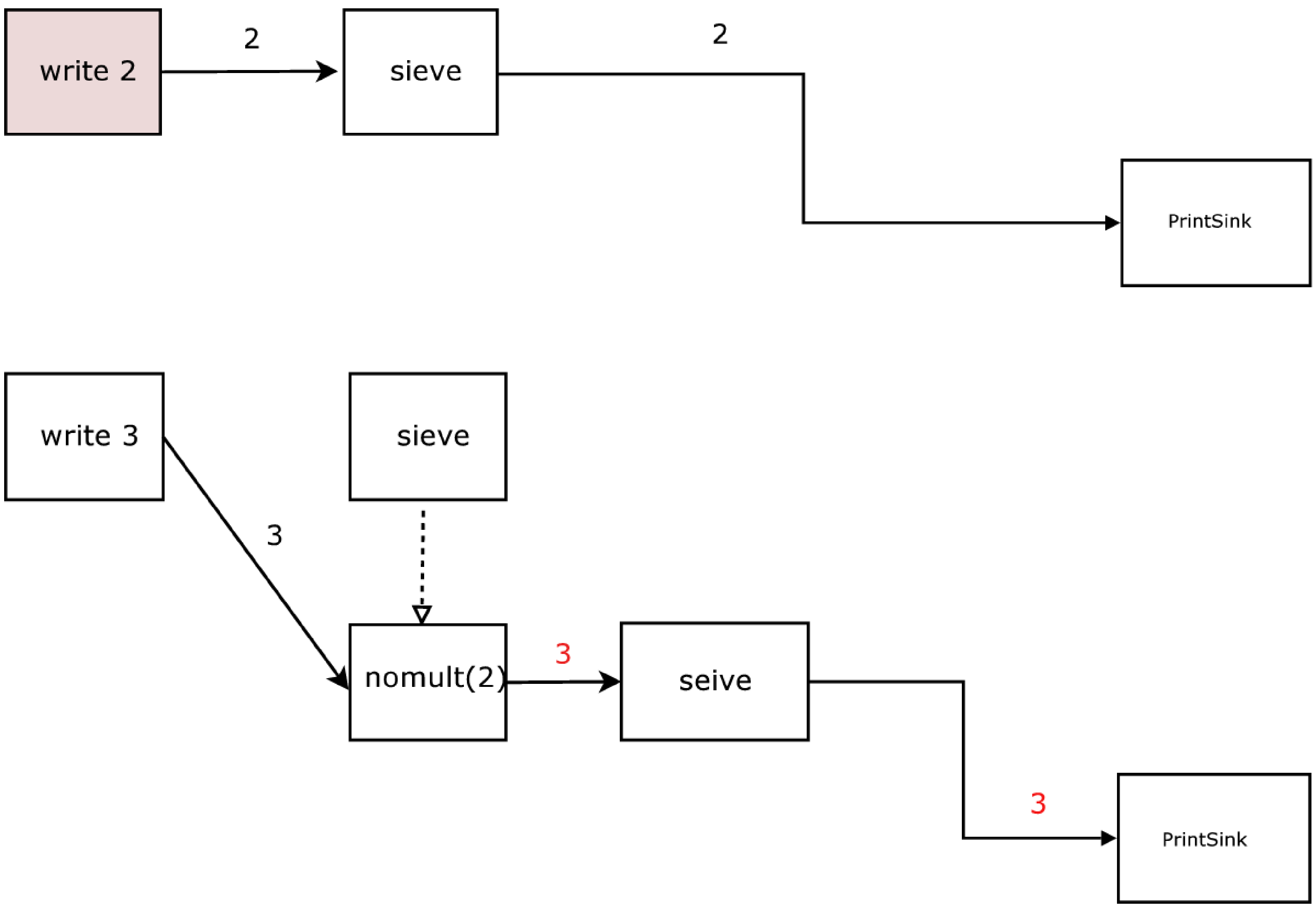
* Most of the work will be done by the process sieve, which inputs a stream of numbers, outputs the first number n, deletes all multiples of n, and (recursively) continues sieving the remainder:

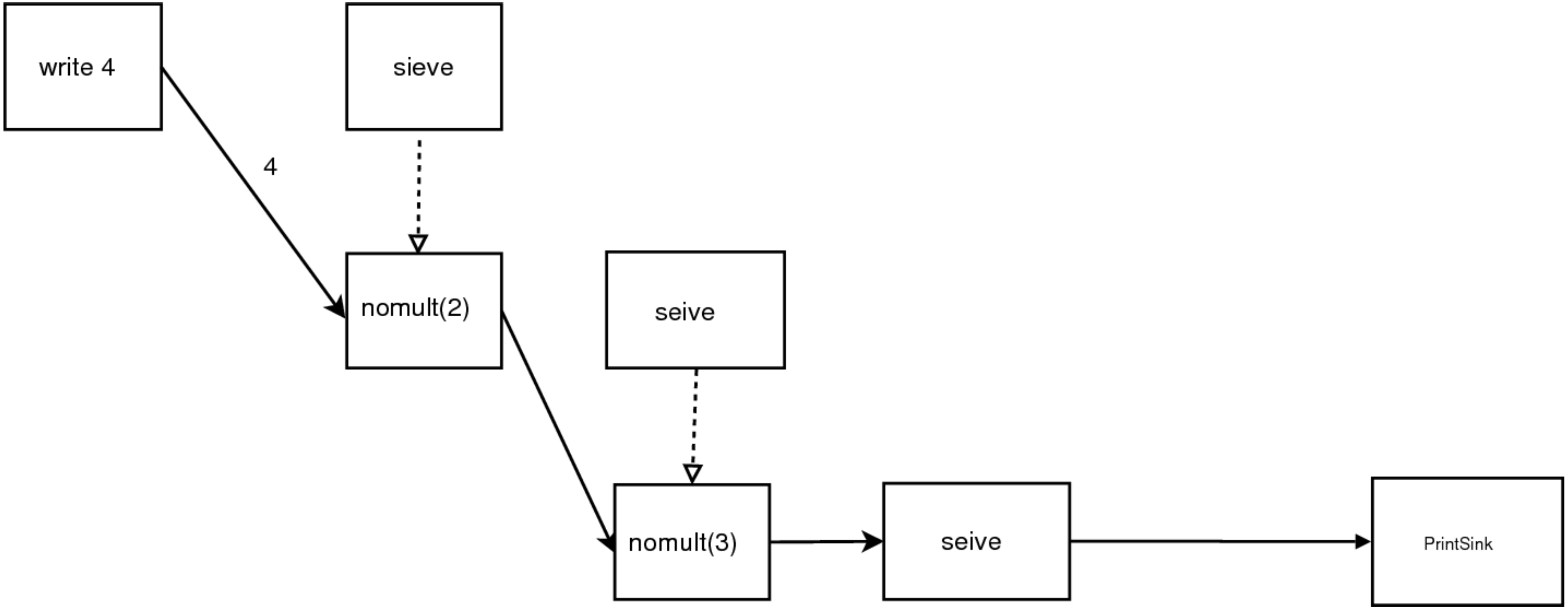
```
def sieve(in: ?[Int], out: ![Int]): PROC = proc {
  val n   = in?()
  val mid = OneOne[Int]
  out!n
  (noMult(n, in, mid) || sieve(mid, out))()
}

def noMult(n: Int, in: ?[Int], out: ![Int]) = proc {
  repeat { val m = in?(); if (m%n != 0) out!m }
}
```

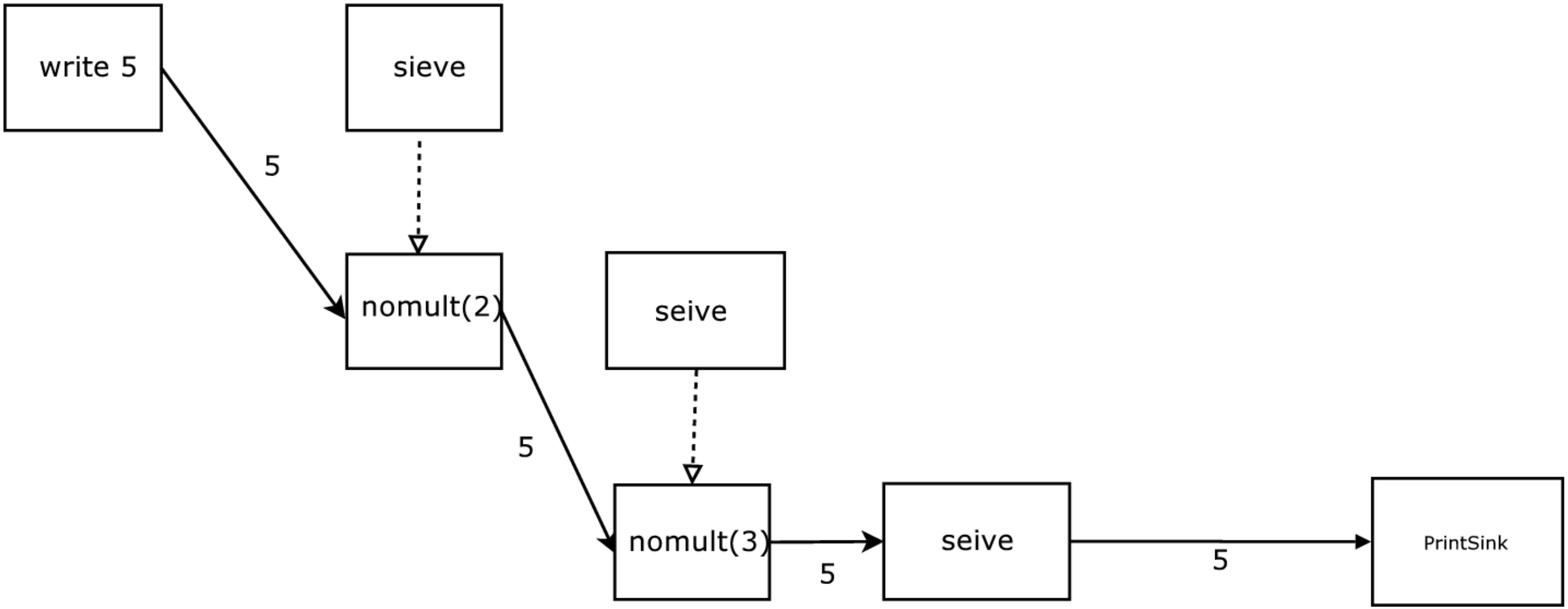
* Note the recursion within sieve.

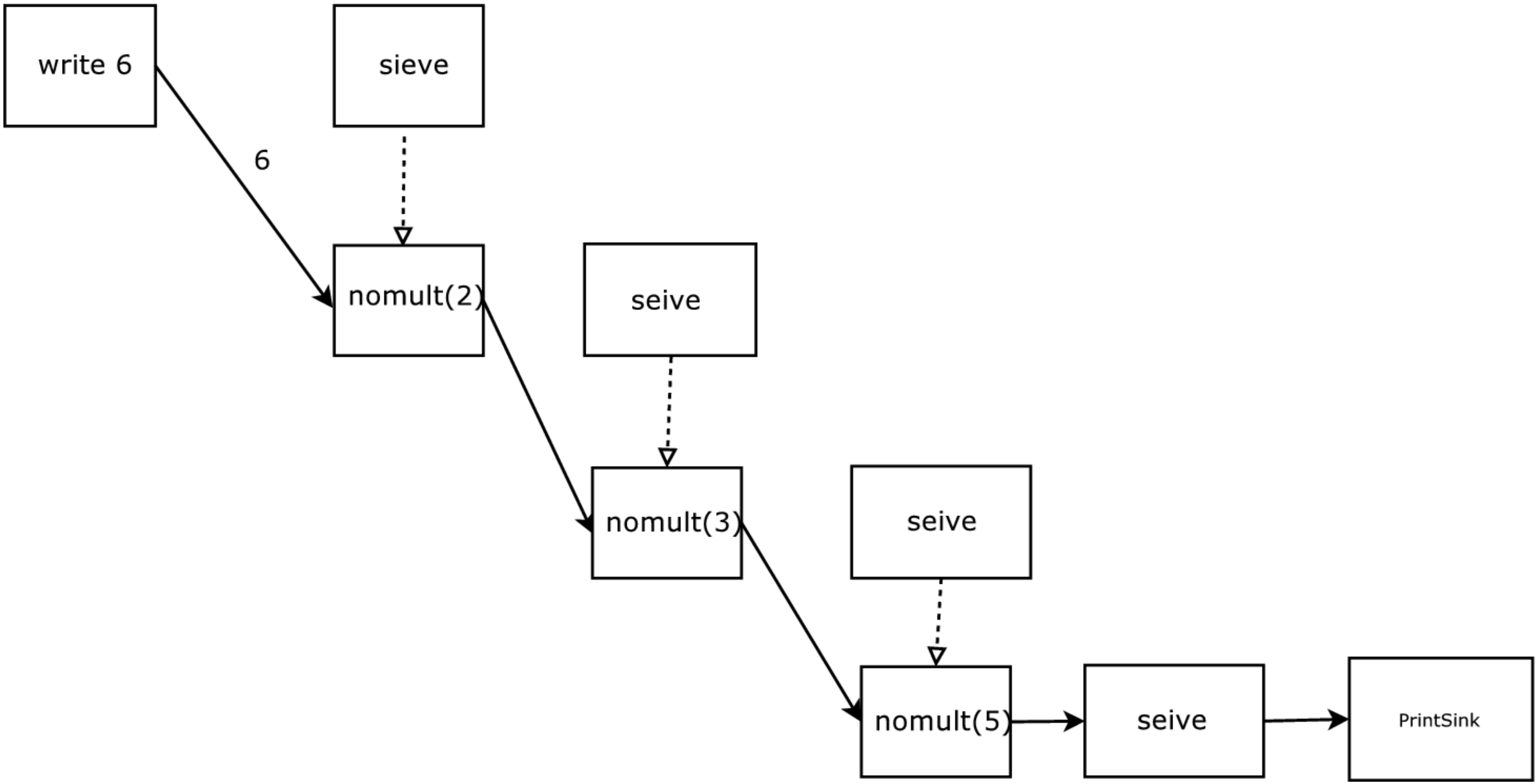
```
object Primes
{ import io.threadcso._
  import io.threadcso.component.console

  ... Sieve and NoMult

  def nats2(out: ![Int]) = proc {
    var i=2; repeat { out!i; i+=1 }
  }

  val mid, res = OneOne[Int]

  val network = nats2(mid) || sieve(mid, res) || console(res)

  def main(args: Array[String]) = network()
}
```

Exercise: How many new threads are needed to run each new call of `sieve` to termination?


write 2
2
sieve
2
PrintSink
write 3
sieve
3
nomult(2)
3
seive
3
PrintSink


write 4
sieve
4
nomult(2)
seive
nomult(3)
seive
PrintSink


write 5
sieve
5
nomult(2)
seive
5
nomult(3)
5
seive
5
PrintSink


write 6
6
sieve
nomult(2)
seive
nomult(3)
seive
nomult(5)
seive
PrintSink

Sharing variables

Earlier we said that concurrent processes should use *disjoint* sets of variables. We can weaken this slightly, to allow processes to share variables, as long as they do so in a disciplined way.

If two concurrently-running processes both access a variable v (other than both reading it) then there should be a (direct or indirect) synchronisation between the running processes after the first finishes accessing v, and before the second starts accessing it. This can be achieved by sending a (dataless) message from the first to the second.

```
var v : T = ···
val c = OneOne[Unit]
proc{··· v = expr; c!(()); ···}  || proc{···; c?(); ··· v ···}
```

We discourage this style of variable-sharing in channel-based programs, because

* it breaks the rule "share by communicating, don't communicate by sharing";
* it makes it impossible to restructure the program to run on separate processors; and
* `proc{··· c!expr; ···} || proc{···; val v=c?(); ···v ···}` is almost always better.

Transmitting mutable objects over channels

Objects can be passed across channels in the same way as values of scalar type.

If two processes share a single *mutable* object there is a potential for race conditions, just as with standard shared variables. This is because in a single running JVM program objects are communicated by reference rather than by value, so object-sharing arises from communication.

For example: consider `P(in, mid) || Q(mid, out)` where

```
def P(in : ?[String], mid : ![Person]) = proc{
  val pers = new Person();
  repeat{ val n = in?; pers.name = n; mid!pers; }
}

def Q(mid : ?[Person], out : ![String]) = proc{
  repeat{ val pers = mid?; out!(pers.name); }
}
```

The single `Person` object created in `P` is shared by `P, Q`.

Cacheing and Channels

Recall that multiprocessor machines may cache variables, and that the JVM does not guarantee that the caches will be kept coherent, so two concurrent processes may operate independently on their own cached copies of the same variable! And further, the compiler is allowed to optimise the code to something "semantically equivalent".

However, when a process reads or write a channel, the updates in its cache are flushed to main memory, and cached values re-read from main memory. Further compiler optimisations may not reorder a read or write of a variable with a channel read or write.[1]

[1] http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html.

For example, in the code

```
val c = OneOne[Unit];
def P = proc{x = ···; c!(()); ···};
def Q = proc{···; c?(); <use x>};
(P || Q)()
```

Q is guaranteed to use the value for x written by P because:

* P finishes writing before the synchronisation;
* The synchronisation ensures the caches are correctly updated;
* The compiler is not allowed to perform optimisations that reorder the accesses to x with those to c.

Contents

Note 1: Trace notations 3 ☞

In these notes we use a simplified notation for traces and the algebraic operations on them.

* If i is an input port, then $i^?$ denotes the (finite) sequence of data so far read from the port by *terminated* read operations.
* If o is an output port, then $o^!$ denotes the (finite) sequence of data so far written to the port by *terminated* write operations.
* The most useful operator on finite sequences of data that we shall use here is $t \,\overline{\downarrow}\, n$. For $0 \leq n \leq \#t$ it denotes the prefix of t comprising all but the last n elements of t. For $n > \#t$ it denotes the empty sequence.
* If ns is a set of natural numbers then $t \,\overline{\downarrow}\, ns$ denotes $\{t \,\overline{\downarrow}\, n \mid n \in ns\}$
* More formally:
    - If t is a sequence of data, and $0 \leq n \leq \#t$ then $t \uparrow n$ ("take n from t") denotes the prefix of t of length n, and $t \downarrow n$ ("drop n from t) denotes the suffix of t obtained by removing its prefix $t \uparrow n$. If $\#t < n$ then $t \uparrow n = t$ and $t \downarrow n = []$.

      Remark: $t \uparrow n \mathbin{+\!\!+} t \downarrow n = t$
    - The "complementary" expressions $t \,\overline{\downarrow}\, n$, and $t \,\overline{\uparrow}\, n$ (respectively meaning "all but the last n elements of t", and "the last n elements of t") can be most concisely defined by $t \,\overline{\downarrow}\, n = reverse((reverse\ t) \downarrow n)$, and $t \,\overline{\uparrow}\, n = reverse((reverse\ t) \uparrow n)$.

      Remark: $t \,\overline{\downarrow}\, n \mathbin{+\!\!+} t \,\overline{\uparrow}\, n = t$

Note 2: Read and write operations in synchronous channels 3 ☞
Implementations of synchronous channels are obliged to delay termination of a write operation until the corresponding read operation has terminated.

Note 3: Using ? as a postfix operator 4 ☞
The expressions `chan?()` and `chan?` mean the same in CSO; and you will see both used in these notes and in CSO source text. Unless postfix operators have been enabled, the latter causes the Scala compiler to complain as follows:

```
warning: postfix operator ? should be enabled
by making the implicit value scala.language.postfixOps visible.
This can be achieved by adding the import clause 'import scala.language.postfixOps'
or by setting the compiler option -language:postfixOps.
```

Note 4: 4 ☞
The two processes are essentially indistinguishable, though not operationally equivalent: on termination they both leave the variable x with value v.

Note 5: 4 ☞
val chan = OneOne[T] is a call to a factory method.

`OneOne` means one sender and one receiver.

Note 6: 5 ☞
We could have written the body of `copy` as:

```
while (true) { out!(in?()) }
```

or (using the read-then-apply notation as)

```
while (true) { in ? { x ⇒ out!x } }
```

Note 7: 8 ☞
The (local) sequential loop invariant is that the traces are identical: $out^! \in in^? \overline{\downarrow}\{0\}$. After the input ("at" the semicolon) $out^! \in in^? \overline{\downarrow}\{1\}$, and the invariant is re-established by the output. Reasoning using local invariants can be useful, but is not sufficient, to deal with networks of communicating processes.

Note 8: 9 ☞

Earlier variants of CSO implemented additional channel types. These days they

```
class OneMany[T]  extends SyncChan[T] with SharedInPort[T] ···
class ManyOne[T]  extends SyncChan[T] with SharedOutPort[T] ···
class ManyMany[T] extends SyncChan[T] with SharedOutPort[T]
                                      with SharedInPort[T] ···
```

* With `OneMany` and `ManyMany` channels, several different processes may read from the same channel — but each value that is read is read by only one of them.
* With `ManyOne` and `ManyMany` channels, several different processes may write to the same channel.

`Go` channels are all, effectively, `ManyMany`. This makes them somewhat (perhaps a lot) more efficient to implement; but it can also can be a source of hard-to-find errors.

Note 9: N2N channels 11 ☞

The parameters given in the construction of an `N2N(writers, readers)` are not limits on the numbers of writing (reading) processes that may share the respective output (input) ports, but on the number of times those ports may be closed for output (input) before the channel is considered to have closed. We deal with channel closure and the termination of networks of processes later in these notes.

Note 10: 11 ☞

The following program (n2ndemo2.scala) also demonstrates the nondeterministic choice of reader-writer pairs synchronised in `N2N` channels.

```
import io.threadcso._
object n2ndemo2 {
   def main (args: Array[String])
   { def copy10[T] (in: ?[T], out: ![T]) = proc { for (_←0 until 10) { out!(in?()) } }
     val mid = N2N[(String, Long)](2, 1)
     val out = N2N[(String, Long)](1, 2)
     val experiment =
         (  proc { for (i←0 until 5)  { mid!("W0", i) }} ||
            proc { for (i←0 until 5)  { mid!("W1", i) }} ||
            copy10(mid, out)                               ||
            proc { for (_ ←0 until 5) out?{ case (w, t) ⇒ print (("R0",w, t)) } } ||
            proc { for (_ ←0 until 5) out?{ case (w, t) ⇒ print (("R1",w, t)) } }
         )

     for (_ ← 0 until 4) { experiment();  println }
     exit
   }
}
```

It output the following four lines the last time I compiled and ran it:

```
(R1,W0,0)(R1,W0,1)(R1,W0,2)(R1,W0,3)(R1,W0,4)(R0,W1,0)(R0,W1,1)(R0,W1,2)(R0,W1,3)(R0,W1,4)
(R0,W0,0)(R0,W0,1)(R0,W0,2)(R0,W0,3)(R0,W0,4)(R1,W1,0)(R1,W1,1)(R1,W1,2)(R1,W1,3)(R1,W1,4)
(R0,W0,0)(R0,W0,1)(R0,W0,2)(R0,W0,3)(R0,W0,4)(R1,W1,0)(R1,W1,1)(R1,W1,2)(R1,W1,3)(R1,W1,4)
(R1,W1,0)(R1,W1,1)(R1,W1,2)(R1,W1,3)(R1,W1,4)(R0,W0,0)(R0,W0,1)(R0,W0,2)(R0,W0,3)(R0,W0,4)
```

Note 11: 13 ☞
A more comprehensive version of `console` that deals properly with the closing of its `in` channel is defined in `io.threadcso.component`.

Note 12: 15 ☞
For example:

* `OneOne.closeOut`=`OneOne.closeIn`=`OneOne.close` – pending `?/!` requests are aborted.
* `OneOneBuf.closeOut` – should close after all currently-buffered values have been read.

* `OneOneBuf.closeIn` – closes immediately, and pending `?/!` requests are aborted.
* `UnsharedOutPort.closeOut` – closes immediately, and pending `?/!` requests are aborted.
* An `N2N[T](writers, readers)` will close when either `closeOut` has been invoked on it *writers* times, or `closeIn` has been invoked on it *readers* times. If *writers* is 0, then `closeOut` will never close the channel; likewise if *readers* is 0, then `closeIn` will never close the channel.

Note 13: tee 23 ☞

`io.threadcso.component` defines a version of `tee` with signature

`tee[T](in: ?[T], outs: Seq[![T]])`.

Note 14: 24 ☞

Generalized concurrent compositions behave analogously: if all components terminate normally then the composition terminates normally; if all terminate either normally or with a `Stop` exception then the composite terminates with a `Stop` exception; otherwise a `ParException` is thrown, embodying the reasons for termination of all the components in the sequence they were composed, representing normal termination by **`null`**.

The point of this apparent complexity is to enable "disorderly" terminations of composite processes to be diagnosed by catching the `ParException`.

Note 15: Incorrectness of an earlier version of *sameStreams* 26 ☞

In an earlier version of this case study we effectively defined *sameStreams* as follows:

```
def sameStreams[T](inl: ?[T], inr: ?[T], ans: ![Boolean]) =
proc {
    var l, r           = inl.nothing
    val nextPair: PROC = { l=inl?(); } || { r=inr?(); }
    var same           = true
    repeat (same && inl.canInput && inr.canInput)
           { nextPair(); same=l==r }
    ans!(same && !inl.canInput && !inr.canInput) // **
    ans.closeOut; inl.closeIn; inr.closeIn
   }
```

We had placed too much reliance on the "obvious" correspondence between streams and lists, and the obvious sameLists function over lists.

```
sameLists []     []     = True                     // both at an end
sameLists (x:xs) (y:ys) = x==y && sameLists xs ys  // neither at an end
sameLists _      _      = False                    // one at an end
```

Following an observation made by a student we realized that

* The second and third conjuncts of the guard `(same && inl·canInput && inr·canInput)` are redundant. Even if they both yield true at the moment the guard is evaluated, either or both streams' writers may have already decided to close the channels. The only definitive test we have (at this point) of whether a port is readable is to read it.
* Without loss of generality, suppose that $same$ is still true, and inl has been closed at the point $nextPair$ is started, but inr has not yet been. The left hand component of $nextPair$ will terminate with a $Closed$ exception; the right hand component will terminate normally; so $nextPair$ will re-throw the $Closed$ exception; thus terminating the **`repeat`**.

  By the time the computation of the answer `(same && !inl·canInput && !inr·canInput)` takes place, inr may or may not be closed, and may or may not be going to be closed – the time of closing of this stream depends on the time it takes the right-hand producer to decide whether it has another value to produce. *In short, we had introduced a race into the program.*
* The results of the methods `canInput, canOutput` can only be relied on indefinitely *if they are false*.

Note 16: 27 ☞
The construct **`attempt`** is a one-shot form of the construct **`repeat`**

Note 17: 28 ☞
This problem is also known as the "same fringe" problem. It has a long history, which is summarised exhaustively (and somewhat tediously) in `http://wiki.c2.com/?SameFringeProblem`.

The technique of separating structure traversal from computation over fringes can easily be adapted: for example to find the longest common prefix of the breadth-first traversal of one tree and the depth-first traversal of another.

Note 18: Buffer specification 29 ☞
The trace specification of `b: OneOneBuf[T](size)` states that the number of `T` data that have been written to the channel *at any moment* can never exceed $size$ more than the number of data that have been read from it *at that moment*, providing $size > 0$.

$$size > 0 \Rightarrow 0 \leq \#c^! - \#c^? \leq size$$

If the excess is strictly less than $size$ we say the buffer is nonfull; if it is strictly greater than 0 we say that the buffer is nonempty. Reading from a nonempty buffer cannot be delayed indefinitely, and writing to a nonfull buffer cannot be delayed indefinitely.

In our implementation an unbounded buffer is constructed by giving a nonpositive $size$.

Note 19: "Upstream" closing 34 ☞

With this program (CountNats.scala) we can investigate what happens when the nats circuit's output is closed at its downstream end. In fact it outputs the specified number of naturals, then terminates *immediately* because the components of the parallel composition have themselves all terminated.

```
object CountNats
{ import io.threadcso._
  import io.threadcso.component.{prefix,map,console,tee}

  /** Yield a process that will copy ‘count‘ inputs to out, then close both ports. */
  def copier[T](count: Int, in: ?[T], out: ![T]): PROC = proc
  { var n = count
    repeat (n>0) { out!(in?); n-=1 }
    out.closeOut; in.closeIn
  }

  def main(args: Array[String]) =
  {  val mid, nats, succs, counted, out = OneOne[Int]
     val N = if (args.size>0) args(0).toInt else 500
     (  prefix(0)(succs, nats)         || tee(nats, out, mid)
     || map((x:Int) ⇒ x+1)(mid,succs) || copier(N, out, counted)
     || console(counted)
     )()
     exit
  }
}
```

Note 20: New threads used by sieve 41 ☞

The CSO implementation uses only one additional thread per terminating invocation of `sieve`. THis is because each invocation of `sieve` that starts running using a thread t continues after its execution of `out!n` by running the parallel composition `(noMult(n, in, mid) || sieve(mid, out))`. As explained in the CSO paper, this is done by running the first component in t, and acquiring a new thread to run the second component.

Note 21: Advantages of the Process/Channel model 46 ☞

Supports Modular Design

In modular program design, various components of a program are developed separately, as independent modules/components, and then combined to obtain a complete program. The goal is to reduce program complexity and facilitate module/component reuse.

It can be particularly straightforward to reason locally about the safety (correctness) aspects of a module whose only interaction with other modules is via channels.

It can be harder to reason about liveness properties *in general*, although methods of analysing inter-module communication are well-understood, and many patterns of deadlock-free communication have been developed.

If modules are specified as interfaces relating their channel histories (traces) then their implementations can be changed without modifying other modules. The properties of a composite can be understood by understanding the interface specifications of its components together with the way in which they are plugged together.

Mapping Independence

The collection of communicating processes that constitutes a program can be mapped to different numbers of CPUs and/or to different numbers of machines.

Because tasks interact using the same mechanism (channels) regardless of task location, the result computed by a program does not depend on where tasks execute. So algorithms can be designed and implemented without regard for the number of processors on which they will execute.

In fact, algorithms are frequently designed that create many more tasks than the expected number of processors.

This is a straightforward way of achieving scalability: as the number of processors increases, the number of tasks per processor is reduced but the algorithm itself need not be modified.

The creation of more tasks than processors can also serve to mask communication delays, by providing other computations that can be performed while communication is performed that accesses "remote" data.

Performance

Sequential programming abstractions such as procedures and data structures, or objects, are effective because they can be mapped simply and efficiently to the von Neumann computer.

Tasks and channels have a similarly direct mapping to multiprocessor machines. A task represents a piece of code that can be executed sequentially on a single processor. If two tasks that share a channel are mapped to different processors the channel connection is implemented as interprocessor communication; if they are mapped to the same processor, a more efficient mechanism can be used.

No longer seen as arcane

The Go language is a serious, and *very seriously-resourced* attempt to realize the conceptual and the performance advantages of communicating concurrent components in a practical mainstream setting.

Those of us who have been long-term enthusiasts for the Process/Channel model hope that the fact that Google is behind Go means that mainstream operating systems designers will (at last) pay attention to mechanisms that facilitate shared-responsibility[2] scheduling and memory management.

Other languages and APIs

* occam is a programming language, with channel communication. It was inspired by CSP, and developed by INMOS in the early 1980s to program Transputers (small multicore machines with hardware channels between cores). The KROC (Kent retargetable occam compiler) is still available from Github.
* XMOS revitalised the ideas behind the Transputer a few years ago. Its XC language has channel communication and is used to program its XCore processors – these consist of very powerful multicore "tiles" with hardware channels between cores, and between tiles. See `https://en.wikipedia.org/wiki/XCore_Architecture`.
* ECSP (`https://www.cs.ox.ac.uk/people/bernard.sufrin/personal/ECSP/ecsp.pdf`), designed in the late 1990s, was a precursor of CSO inspired by occam.
* JCSP (`http://www.cs.kent.ac.uk/projects/ofa/jcsp/`) is a Java API based on occam like semantics.
* Various APIs hosted within Python, C, and C++ have occam-style channels and communication.
* The MPI (Message Passing Interface) (see `https://computing.llnl.gov/tutorials/mpi/`) is an API for C that is well-known in the high-performance computing world.
* Scala and Erlang provide implementations of "Actors" – process-like entitities that communicate using messages. The Erlang virtual machine is reputed to be very efficient, though the language is dynamically typed.

[2] Responsibility shared between kernel and user-level language run-times.

Note 22: 46 ☞

Recent versions of Scala give spurious warnings about terminating an argument list if one tries to communicate the unit value $()$ on a Unit channel c using the command $c!()$. We avoid the spurious warning by using $c!(())$