Concurrent Programming

Bernard Sufrin and Gavin Lowe

Hilary Term 2016-17

8a: Server Elimination

Server Elimination: Replumbing for efficiency

Interprocess communication through channels without the intervention of a mediating (server) process is (in general) more efficient because there are fewer context switches.

It is usually possible to restructure programs that use "active plumbing" components implemented by processes (*e.g.* `zip, merge, tee, probe, prefix, map`) that connect input and/or output channels so as to eliminate mediating (server) processes.

For example, we could build a `merge2Gen` class such that the following behave identically when `···` only reads from `out` and only writes to `l, r`

```
val m = new merge2Gen[T]
val out = m·out
val l = m·in1
val r = m·in2
···
```

```
val l, r, out = OneOne[T]
(  merge(List(l,r), out)
|| ···
)()
```

The restructuring components can be built using low-level concurrency constructs, can be harder to understand, and may have subtly different synchronization or termination characteristics.

Example: Tee2Gen

Here we *sketch*[1] a component designed so that the following behave identically when `···` only reads from `l, r` (in separate processes) and only writes to `out`

```
val t = new Tee2Gen[T]
val out = t.out
val l = t.in1
val r = t.in2
...
```

```
val out, l, r = OneOne[T]
(  tee(out, List(l, r))
|| ···
)()
```

THE CODE IS UNTESTED!

Our solution is structured as follows:

```
class Tee2Gen[T](name: String)
{ var obj: T = _                  // the most recent out!value
  val in1, in2 = new Reader       // InPorts that yield obj on reads

  // out !  value synchronises the assignment of value to obj
  object out extends OutPort[T] { ··· }

  // aReader ?  () synchronizes the returning of the
  // current value of obj
  class Reader extends InPort[T] { ··· }

  def atomically[T](mutex: BooleanSemaphore){body: ⇒ T} =
      try { mutex·acquire; body } finally { mutex·release }
}
```

The readers are controlled using `permit···, await···` to operate signalling semaphores.

Reads must terminate before a write returns (because !, ? are synchronized in `OneOne`)

```
object out extends OutPort[T]
{  _isOpenForWrite = true

   def !(value: T): Unit =
   { if (! _isOpenForWrite) throw new Closed(name)
     obj = value
     // permit both readers to proceed
     in1·permitRead
     in2·permitRead
     // await (possibly forced) synchronization with read terminations
     in1·awaitReadFinished
     // synchronization may have been due to a close
     if (! _isOpenForWrite) throw new Closed(name)
     in2·awaitReadFinished
     // synchronization may have been due to a close
     if (! _isOpenForWrite) throw new Closed(name)
   }
   ...
```

The input ports must be read by different processes else this implementation could deadlock. (Does this *really* make the component useless?)

When the out port is closed, the in ports are forced to close.

```
    override def close: Unit = {
      // once-only:  mark closed and tell both inports
      if (_isOpenForWrite) {
         _isOpenForWrite = false
          in1·close
          in2·close
      }
    }
  }
```

`permit`··· synchronizes the start of a read; `await`··· synchronizes its termination.

```
class Reader extends InPort[T]
{  val canRead      = BooleanSemaphore(false)
   val readFinished = BooleanSemaphore(false)
   def permitRead         = canRead·release
   def awaitReadFinished  = readFinished·acquire
   ...
```

When an in port is closed it forces the out port to close, then pretends that any outstanding reads/writes have terminated.

```
   _isOpen = true

   override def close: Unit =  {
     // once-only:  mark closed and tell the out port
     if (_isOpen) {
        _isOpen = false
        out·close
        readFinished·release       // simulate termination of ?  so !  proceeds
        canRead·release            // simulate termination of !  so ?  proceeds
     }
   }
```

Reading is straightforward

```
def ?(): T =
 { if (! _isOpen) throw new Closed(name)
   canRead·acquire          // synchronize with permit...  from the out port
   if (! _isOpen) throw new Closed(name) // closed DURING read wait
   val v = obj
   readFinished·release     // synchronize with await...  from the out port
   return v
 }
```

Extended rendezvous `in ? f` synchronizes with `out!v` *after* the rendezvous `f(v)` has terminated

```
def ??[U](f: T ⇒ U): U =
{ if (! _isOpen) throw new Closed(name)
  canRead·acquire         // synchronize with permit...  from the out port
   if (! _isOpen) throw new Closed(name)
  val v = f(obj)
  readFinished·release    // synchronize with await...  from the out port
  return v
}
}
```

(in this case, the rendezvous is a three-way affair: both readers and the writer synchronize)

Contents