Concurrent Programming (Sheet #4)

Marius Gavrilescu

1. There is a possibility of deadlock in this case, if the two alts select different processes. Then each alt will try to read from its channel (making it writable), but noone will write.

2. ```
   def Interleave(left: ?[Int], right: ?[Int], out: ![Int]) = proc {
     serve {
       (left)  =?=> (x => out!x) |
       (right) =?=> (x => out!x)
     }
     out.closeOut
     left.closeIn
     right.closeIn
   }
   ```

4. ```
   def Changer(inPound: ?[Unit], out5p: ![Unit], out10p: ![Unit], out20p: ![Unit]) = pro
     val balance = 0
     serve {
       (balance == 0 && inPound) =?=> (x => balance += 100) |
       (balance > 20 && out20p)  =!=> (balance -= 20;) |
       (balance > 10 && out10p)  =!=> (balance -= 10;) |
       (balance > 5  && out5p )  =!=> (balance -= 5;)
    }
   }
   ```

5. ```
   def untagger[T](shared: ?[(Int, T)], l: ![T], r: ![T]) = proc {
     repeat {
       val (i, t) = shared?()
       if(i == 0) {
         l!t
       } else {
         r!t
       }
     }
     shared.closeIn
     l.closeOut
     r.closeOut
   }
   ```

   In `Independent` the maximum rates are `lR`, `rR` respectively. In `Shared` the whole system runs at the speed of the slowest channel, so the maximum rate of writing to the input channels is the lesser of `lR` and `rR`.

   In `Independent` if one channel closes its pair will also close but the other two channels will remain open and functional. If one channel closes in `Shared` the shared channel will close and everything else will be closed.
   ```

7.
```
class ChanBridgeController {
  val enterN, leaveS, enterS, leaveN = ManyOne[Unit]

  val server = proc {
    var cntNS, cntSN = false
    serve {
      (cntSN == 0 && enterN) =?=> (x => cntNS += 1) |
      (cntNS == 0 && enterS) =?=> (x => cntSN += 1) |
      leaveS =?=> (x => cntNS -= 1) |
      leaveN =?=> (x => cntSN -= 1)
    }
  }

  def ArriveNorth = enterN?()
  def LeaveSouth  = leaveS?()
  def ArriveSouth = enterS?()
  def LeaveNorth  = leaveN?()
}
```

We can achieve fairness if after K cars from one side entered we wait for them all to leave until releasing any more cars.

You can limit the number of cars by also guarding the enter channels on cntNS and cntSN being smaller than the limit.

We might be able to detect a car being stuck on the bridge with a timeout in the serve.

Traffic signals would make the solution significantly more complex, as Arrive* and Leave* methods would no longer map directly to RPCs.

8.
```
def f(a: Int, b: Int) = { a + b }

def Node(x: Int, i: Int, in: ?[Int], out: ![Int]) = proc {
  if(i == 0) {
    out!x
  } else {
    val acc = in?()
    out!(f(acc, x))
  }

  val result = in?()
  println("Result is" + result)
  out!result
  if(i == 0)
    in?()

  ()
}

def runit = {
  val c1, c2, c3, c4 = OneOne[Int]
  val n1 = Node(1, 0, c1, c4)
  val n2 = Node(5, 1, c2, c1)
```

```
  val n3 = Node(60, 2, c3, c2)
  val n4 = Node(100, 3, c4, c3)

  run(n1 || n2 || n3 || n4)
}
```

9.

10.
```
class FastBarrier(val N: Int) {
  val up   = for (i <- 0 until N) yield OneOne[Unit](s"up$i")
  val down = for (i <- 0 until N) yield OneOne[Unit](s"down$i")

  def sync(me: Int) = {
    val child1 = 2 * me + 1
    val child2 = 2 * me + 2
    if(child1 < N) up(child1)?;
    if(child2 < N) up(child2)?;

    if(me != 0) {
      up(me)!(())
      down(me)?;
    }

    if(child1<N) down(child1)!(())
    if(child2<N) down(child2)!(())
  }
}
```

11. x and y are points in the square with corners $0, 0$ and $1, 1$. $x^2 + y^2 < 1$ means x and y are also in the unit circle. This means x and y must be in the first quadrant (which has area $\frac{\pi}{4}$). The square has area 1, and since the coordinates are uniformly distributed the probability the point lies in the quadrant is $\frac{\pi}{4}$.

```
import scala.util.Random
var totalIn, total = 0
val monitor = new Object

def worker(cnt: Int) = proc {
  var in, out = 0
  for(i <- 0 until cnt) {
    val x = Random.nextDouble
    val y = Random.nextDouble
    if(x * x + y * y < 1)
      in += 1
    else
      out += 1
  }

  monitor synchronized {
    totalIn += in
    total += in + out
```

```
  }
}

val trials = 100000000

def computePI(N: Int) = {
  var bigproc = worker(trials / N)
  for(i <- 1 until N)
    bigproc = bigproc || worker(trials / N)
  run(bigproc)
  println(4.0 * totalIn / total)
}
```

13.
```
def ListBuilder(in: ?[Int], out: ![List[Int]]) = proc {
  var xs: List[Int] = Nil
  serve {
    in =?=> (x => xs = x :: xs) |
    out =!=> (xs) ==> (xs = Nil)
  }
  out.closeOut
  in.closeIn
}

val nodes: Array[Node] = ...
val p = ...
val height = ...

val ins  = for(i <- 0 until p) yield OneOne[Int]
val outs = for(i <- 0 until p) yield OneOne[Int]
val lbs  = for(i <- 0 until p) yield new ListBuilder(ins(i), outs(i))

import scala.collection.mutable.HashSet

def worker(me: Int) = proc {
  val visited = new HashSet[Int]
  repeat {
    val xs = outs(me)?()
    val next: List[Int] = Nil
    for(a <- xs) {
      if(!visited(a)) {
        visited.add(a)
        nodes(a).Visit
      }
      next = next ++ nodes(a).successors
      ins(me)!next
    }
  }
}
```