Concurrent Programming - Sheet 1

George Andrews

February 5, 2019

1

2

3

Input and output run at the same time, with two threads, one continuously enqueueing inputs and the other emptying the queue on the out channel while it isn't empty.

```
def buff[T](in: ?[T], out: ![T]) = proc{
    var queue = Queue[T]()
    def input=proc{
        while(true){
            queue.enqueue(in?())
        }
    }
    def output=proc{
        while(true){
            if(queue.isEmpty()==false){
                out!queue.dequeue()
            }
        }
    }
    run(input || output)
}
```

4

```
import io.threadcso._
import scala.collection.mutable.Queue

object CPSheet2Q4{

val epsilon=0.1
val numWorkers=100
val ins=OneMany[(Double,Double)]
val outs=ManyOne[Option[Double]]
val intervals=Queue[(Double,Double)]()
var numIntervals=0
var estimate=0.0
def estimate(f: Double => Double, a:Double, b:Double){

    def mainThread() = proc{
        var system = taskThread || addingThread
        for(i<-0 until numWorkers){
            var w = estimateWorker(ins,outs)
```

```
            system = system || w
        }
        numIntervals=1
        run(system)
        intervals.enqueue((a,b))
    }

    def taskThread = proc{
        while(numIntervals>0) if(intervals.isEmpty==false) ins!intervals.dequeue
        println(estimate)
        exit()
    }

    def addingThread = proc{
        while(true){
            val out=outs?()
            out match {
              case Some(b) =>
                estimate+=b
                numIntervals-=1
              case default =>
                numIntervals-=1
            }
        }
    }

    def estimateWorker(rangeIn: ?[(Double,Double)], out: ![Option[Double]]) = proc{
        while(true){
            val (ca,cb)=rangeIn?()
            val mid=(ca+cb)/2.0
            val fa=f(ca); val fb=f(cb); val fmid=f(mid)
            val lArea=(fa+fmid)*(mid-a)/2; val rArea = (fmid+fb)*(b-mid)/2
            val area=(fa+fb)*(b-a)/2
            if(Math.abs(lArea+rArea-area) < epsilon) out!Some(area)
            else {
                numIntervals+=2
                intervals.enqueue((ca,mid))
                intervals.enqueue((mid,cb))
                out!None
            }
        }
    }

}
}
```

5

```
import io.threadcso._
import scala.collection.mutable.Queue

object CPSheet2Q5{
    class Synchronisation {

        //val men=Queue[String]()
        //val women=Queue[String]()

        val inMen=ManyOne[String]
        val inWomen=ManyOne[String]
        val outMen=OneMany[String]
        val outWomen=OneMany[String]

        private def pairer() = proc{
            while(true){
                val m = inMen?()
                val w = inWomen?()
                outMen!(w)
                outWomen!(m)
            }
        }

        def manSync(me: String):String = {
            inMen!(me)
            outMen?()
        }

        def womanSync(me: String):String = {
            inWomen!(me)
            outWomen?()
        }

        run(pairer)

    }

    def main(args:Array[String])= {
        val synchro = new Synchronisation
        run(proc{Assert(synchro.manSync("Dave")=="Emma")}||
            proc{Assert(synchro.manSync("Steve")=="Jess")}||
            proc{Assert(synchro.womanSync("Emma")=="Dave")}||
            proc{Assert(synchro.womanSync("Jess")=="Steve")})
    }

}
```

6

```
import io.threadcso._
import scala.collection.mutable.Queue
import scala.util.Random

object CPSheet2Q6{

    def f(x:Double,y:Double)={
        math.max(x,y)
    }

    class Ring(n:Int){
        require(n >=2)
        private val chan=Array.fill(n)(OneOne[Double])

        def apply(me:Int,x:Double):Double= {
            val in=chan(me); val out=chan((me+1)%n)
            if(me==0){
                out!x
                val y=in?()
                out!y
                in?()
                y
            } else{
                val z=in?()
                out!f(z,x)
                val y=in?()
                out!y
                y
            }
        }

    }

    var xs:Array[Double]=null
    var results:Array[Double]=null
    def thread(me:Int,r:Ring)=proc("Thread"+me){
        val x=Random.nextDouble(); xs(me)=x
        val result=r(me,x)
        results(me)=result
    }

    def runTest= {
        val n=1+Random.nextInt(20)
        val r = new Ring(n)
        xs= new Array[Double](n); results= new Array[Double](n)

        run(|| (for (i <- 0 until n) yield thread(i,r)))
            //Checkresults
        var fx=xs(0)
        var i=1
        while(i < n){
            fx=f(fx,xs(i))
            i+=1
        }
        i=0
        while(i < n){
```

```
                assert(results(i)==fx, "xs="+xs.mkString(",")
                    +"\nresults="+results.mkString(","))
                i+=1
            }

        }
}
```

Invariant properties: After i ($i < n$) sequential operations, thread i holds $f(f(f(...f(x_0, x_1), x_2)...), x_i)$.
After $n + i$ ($i < n$) operations, threads 0 to i each contain $f(f(f(...f(x_0, x_1), x_2)...), x_{n-1})$
If f has certain properties, the second ring topology can be used instead of the first. This will do the same, in n (half as many) sequential operations. For this, $f(f(f(...f(x_0, x_1), x_2)...), x_{n-1})$ must equal $f(f(f(...f(x_1, x_2), x_3)...), x_0)$ and other cycles of these. Associativity is not sufficient, and commutativity would not necessarily be enough, but both together will ensure these are equal. This process can be done with n processes, each running the 'apply f and pass to the next process' part of apply n times. At the end of this all processes will have the correct value.

7

7.1

While the token is circulating, a previously passive message may become active before the token is passed back to process 0. Say all processes are passive except 10 when 0 goes passive. The message is sent to process 1 and then process 2, and then process 10 tells 1 to become active, but becomes passive itself, and process 1 remains active indefinitely without activating any other processes. The token is still true, so if the other processes remain passive and 1 remains active, 1 will still be active when the true token returns to process 0.
Essentially, since the scheme does not take into account what happens to the processes after they pass on the token, this does not guarantee total passivity at once.

7.2

A simple, though perhaps inefficient fix is as follows: Every token is given a timestamp that does not change as it is passed around, that records the time this started being sent in some way. Whenever a process becomes active, it sends a token with a timestamp to process 0. If process 0 recieves any passivity token with a timestamp from before the most recent activity timestamp (process 0 keeps a track of the most recent activity) it is discarded as unreliable. If process 0 is still passive when this token returns, it sends another, as with the original scheme. With this, the only way the system can terminate is if all processes have been confirmed as passive, and none have become active since the confirmation began. This satisfies safety, as the above must be satisfied for the process to terminate - if any process is active, the system cannot terminate; and liveness is satisfied (though is not as useful as it perhaps could be) as if all processes are passive and remain passive for the duration of the check the system will terminate.

7.3

With all messages sent in a ring topology this is slightly more difficult. However, we are given the property that ALL messages must follow the ring topology, not just the ones for determining termination. In this case, every process will send a true token (with a timestamp) to its next process when it goes passive. Before doing this, it will send on (and count) every token of this type it has received. If the count reaches n and is passive, it terminates. If it receives a false token it will reset its count to 0, updates its 'active time' and discards any true tokens it receives stamped after that time. It then re-sends its own true token with an updated timestamp. If a process becomes active it will send out a false token with the appropriate time.
This works because of the following: If n true tokens are encountered timed after the last activity, that means all processes have reported being passive since the last activity was detected. If there was some active process, that must've become active at some point, taking a message from another process. However, if that process became active, it would've sent out a false token, and at most $n-1$ true tokens could exist after that timestamp, as processes only create one token per new false timestamp. The process in question must recieve this false token before all n true tokens, as the active process must've received a message from an active (presumably now inactive) process before it. This means safety holds.
Liveness holds, as quite trivially, if all processes are active and become passive one by one (without activating any other processes) n tokens will eventually be passed around the ring, terminating all processes.