Concurrent Programming (Sheet #2)

Marius Gavrilescu

1.
```
class FCFSLock2 extends Lock {
  var locked = 0
  val mutex = new BooleanSemaphore(available=true, fair=true)
  val lock = new BooleanSemaphore(available=false, fair=true)

  def lock = {
    mutex.down
    if(locked == 0) {
      locked = 1
      mutex.up
    } else {
      locked = locked + 1
      mutex.up
      lock.down
    }
  }

  def unlock = {
    mutex.down
    locked = locked - 1
    if(locked > 0) lock.up
    mutex.up
  }
}
```

   For locks with priorities, we can use the same technique as for `FCFSLock` but with a priority queue instead of a queue. To ensure liveness, we can use `Thread.currentThread.getPriority - K * N` as the priority of every incoming thread, where `K` is a small factor and `N` is the number of locks requested so far. This way the earlier a lock request was submitted the higher its priority. This ensures liveness because for any pending lock request there is a maximum number of future locks that can preempt it.

```
class H2OCS extends H2O {
    var hs = 0

    val mutex = new BooleanSemaphore(true)
    val oWait = new CountingSemaphore(0)
    val hWait = new CountingSemaphore(0)

    def O: Unit = {
        oWait.down
        mutex.down
```

```
        hs -= 2
        hWait.up
        hWait.up
        mutex.up

    def H: Unit = {
        mutex.down
        hs += 1
        if(hs % 2 == 0)
          oWait.up
        mutex.up
        hWait.down
    }
}

class CountingSemaphoreFQ(private var available: Int) {
  private val queue = new scala.collection.mutable.Queue[Flag]
  private val mutex = new BooleanSemaphore(true)

  def up = {
    mutex.down
    if(queue.isEmpty)
      available += 1
    else {
      queue.dequeue.release
    }
    mutex.up
  }

  def down = {
    mutex.down
    if(available) {
      available -= 1
      mutex.up
    } else {
      val flag = new Flag
      queue.enqueue(flag)
      mutex.up
      flag.down
    }
  }
}

import scala.collection.mutable.Queue
import scala.collection.mutable.HashMap

class SemaphoreTuteArranger {
  private val donFlagQ = new Queue[Flag]
  private val donIdQ   = new Queue[Int]
  private val donMap   = new HashMap[Int, Int]
```

```
  private val studentFlagQ = new Queue[Flag]
  private val studentIdQ   = new Queue[Int]
  private val studentMap   = new HashMap[Int, Int]
  private val mutex = new BooleanSemaphore(true)

  def don(id: Int) = {
    mutex.down
    if(studentFlagQ.isEmpty) {
      val f = new Flag
      donIdQ.enqueue(id)
      donFlagQ.enqueue(f)
      mutex.up
      f.down
      mutex.down
      val ans = donMap(id)
      donMap -= id
      mutex.up
      ans
    } else {
      val ans = studentIdQ.dequeue
      studentFlagQ.dequeue.up
      studentMap(ans) = id
      mutex.up
      ans
    }
  }

  def student(id: Int) = {
    mutex.down
    if(donFlagQ.isEmpty) {
      val f = new Flag
      studentIdQ.enqueue(id)
      studentFlagQ.enqueue(f)
      mutex.up
      f.down
      mutex.down
      val ans = studentMap(id)
      studentMap -= id
      mutex.up
      ans
    } else {
      val ans = donIdQ.dequeue
      donFlagQ.dequeue.up
      donMap(ans) = id
      mutex.up
      ans
    }
  }
}
```

2.
```
object Monitor3 {
  var menWaiting, womenWaiting = 0
  var menCommited = 0

  val mutex = new BooleanSemaphore(true)
  val womanWait = new BooleanSemaphore(false)
  val manWait = new BooleanSemaphore(false)

  def ManEnter = {
    mutex.down
    menWaiting += 1
    while(womenWaiting == 0) {
      mutex.up
      manWait.down
      mutex.down
    }
    womenWaiting -= 1
    menWaiting -= 1
    menCommited += 1
    if(womenWaiting > 0) womanWait.up
    mutex.up
  }

  def WomanEnter = {
    mutex.down
    womenWaiting += 1
    if(menWaiting > 0) manWait.up
    while(menCommited == 0) {
      mutex.up
      womanWait.down
      mutex.down
    }
    menCommited -= 1
    mutex.up
  }
}
```

3.
```
class Smoothing(val N: Int, image: Array[Array[Boolean]]) {
  val WORKERS = 10

  var source = image
  var dest = Array.ofDim[Boolean](N, N)

  val computed = new Barrier(WORKERS)
  val updated = new CombiningBarrier[Boolean](WORKERS, true, (_ && _))

  def valOf(i: Int, j: Int) = {
    if(i < 0 || j < 0 || i >= N || j >= N) 0
    else if(source(i)(j)) 1
    else -1
```

```
    }

    def compute(i: Int, j: Int) = {
      var sum = 0
      for(a <- i-1 to i+1)
        for(b <- j-1 to j+1)
          if(a != i || b != j)
            sum += valOf(a, b)
      sum > 0
    }

    def worker(start: Int, cnt: Int) = proc {
      var finished = false
      var stable = true

      while(!finished) {
        for(i <- start until start + cnt)
          for(j <- 0 until N) {
            dest(i)(j) = compute(i, j)
            stable = stable && dest(i)(j) == source(i)(j)
          }
      }

      computed.sync()

      if(start == 0) {
        val t = source
        source = dest
        dest = t
      }

      finished = updated.sync(stable)
      stable = true
    }

    def run = {
      val slice = N/WORKERS
      var bigproc = worker(slice * (WORKERS-1), N - slice * (WORKERS-1))
      for(i <- 0 until WORKERS-1) {
        bigproc = bigproc || worker(slice * i, slice)
      }
      bigproc()
    }
  }
```

4.

5.
```
class MonitorCoordinator extends Coordinator {
    var sum = 0

    def Enter(id: Int) = synchronized {
```

```
    while(sum % 3 > 0) wait
    sum = sum + id
    if(sum % 3 == 0)
      notify
    sum
  }

  def Exit(id: Int) = synchronized {
    sum = sum - id
    if(sum % 3 == 0)
      notify
    sum
  }
}

class SemaphoreCoordinator extends Coordinator {
  var sum = 0
  var waitingNr = 0

  val mutex    = new BooleanSemaphore(true)
  val waitHere = new BooleanSemaphore(false)

  def Enter(id: Int) = {
    mutex.down
    if(sum % 3 > 0) {
      waitingNr = waitingNr + 1
      mutex.up
      waitHere.down
      mutex.down
    }
    sum = sum + id
    if(waitingNr > 0 && sum % 3 == 0) {
      waitingNr = waitingNr - 1
      waitHere.up
    }
    mutex.up
    sum
  }

  def Exit(id: Int) = {
    mutex.down
    sum = sum - id
    if(waitingNr > 0 && sum % 3 == 0) {
      waitingNr = waitingNr - 1
      waitHere.up
    }
    mutex.up
    sum
  }
}
```

```
import util.Random

class CoordinatorTest(c : Coordinator) {
  def nextInt = { Math.abs(Random.nextInt) }

  def myBody : Unit = {
    val id = Random.nextInt % 10
    sleep(seconds(nextInt * seconds(.0001)))
    println("id " + id + " got " + c.Enter(id))
    sleep(seconds(nextInt * seconds(.0001)))
    c.Exit(id)
  }

  def makeProc : PROC = proc { myBody }

  def apply = {
    var bigproc = makeProc
    for(i <- 1 to 100)
      bigproc = bigproc || makeProc
    bigproc()
  }
}
```

6.
```
import semaphore._
object Buffer {
    val N = 10
    val arrays = Array.ofDim[Int](2, N)
    var target = 0
    var size = 0
    var waiting = 0

    val mutex   = new BooleanSemaphore(true)
    val putWait = new BooleanSemaphore(false)
    val getWait = new BooleanSemaphore(false)

    def Put(item: Int) = {
        mutex.down
        if(size == N) {
          waiting = waiting + 1
          mutex.up
          putWait.down
          mutex.down
          waiting = waiting - 1
          if(size < N - 1 && waiting > 0) putWait.up
        }
        arrays(target)(size) = item
        size = size + 1
        if(size == N) getWait.up
        mutex.up
```

```
    }

    def Get : Array[Int] = {
        mutex.down
        if(size < N) {
            mutex.up
            getWait.down
            mutex.down
        }
        target = 1 - target
        size = 0
        val ans = arrays(1 - target)
        if(waiting > 0) putWait.up
        mutex.up
        ans
    }
}
```

7.
```
class SemaphoreController extends ReadWriteController {
    private val mutex    = new BooleanSemaphore(true)
    private val readable = new BooleanSemaphore(false)
    private val writable = new BooleanSemaphore(false)

    private var readers, writers, rwaiting, wwaiting = 0

    def startRead = {
        mutex.down
        if(writers != 0) {
          rwaiting += 1
          mutex.up
          readable.down
          mutex.down
        }
        readers += 1
        mutex.up
    }

    def startWrite = {
        mutex.down
        if(writers != 0 || readers != 0) {
          wwaiting += 1
          mutex.up
          writable.down
          mutex.down
        }
        writers = 1
        mutex.up
    }

    def endRead = {
```

```
        mutex.down
        readers -= 1
        if(readers == 0 && wwaiting > 0) {
          wwaiting -= 1
          writable.up
        }
        mutex.up
    }

    def endWrite = {
        mutex.down
        writers = 0
        if(wwaiting > 0) {
          wwaiting -= 1
          writable.up
        }
        else if(rwaiting > 0) {
          rwaiting -= 1
          readable.up
        }
        mutex.up
    }
}
```