1 Question 1

This is beneficial for multiple reasons:

- First, it probably will lead to lower latency in situations where two different tabs simultaneously require work to be done.
- Second, it is essential if we take sound into account: often people have music playing in one tab while reading something in another tab. This is only possible if the tabs are run in parallel, or are multiplexed to an extent to which they are essentially running in parallel.
- Third, it is probably conceptually cleaner to implement each tab as it's own process, rather than have a single large process that runs all tabs simultaneously.

All of these are still valid in a uni-processor computer.
The main problem is that concurrent programs are relatively harder to get right. This can be dealt with through careful design.

2 Question 2

Combinatorics tells us that there are $m + n$ take n i.e. $\frac{(m+n)!}{m!n!}$ ways of interleaving two sequences, one of size n and one of size m.

3 Question 3

Suppose that process `p` executes `LD x; INC x; ST x; LD x; ADD x 2; ST x` and `q` executes `LD x; ADD x 4; ST x`. The time at which `INC`'s and `ADD`'s happen does not affect the result of the interleaving, so I omit them. Also, I consider interleavings that differ only by a permutation of adjacent loads in different threads as being the equivalent. Thus, the following distinct interleavings are possible:

p()			LD x	ST x	LD x	ST x
q()	LD x	ST x				
value of x:	0	4	4	5	5	7
p()		LD x		ST x	LD x	ST x
q()	LD x		ST x			
value of x:	0	0	4	1	1	3
p()			LD x	ST x	LD x	ST x
q()	LD x	ST x				
value of x:	0	4	4	5	5	7
p()		LD x	ST x		LD x	ST x
q()	LD x			ST x		
value of x:	0	0	1	4	4	6
p()	LD x	ST x			LD x	ST x
q()			LD x	ST x		
value of x:	0	1	1	5	5	7
p()		LD x	ST x	LD x		ST x
q()	LD x				ST x	
value of x:	0	0	1	1	4	3
p()	LD x	ST x	LD x			ST x
q()				LD x	ST x	
value of x:	0	1	1	1	5	3
p()		LD x	ST x	LD x	ST x	
q()	LD x					ST x
value of x:	0	0	1	1	3	4
p()	LD x	ST x		LD x	ST x	
q()			LD x			ST x
value of x:	0	1	1	1	3	5
p()	LD x	ST x	LD x	ST x		
q()					LD x	ST x
value of x:	0	1	1	3	3	7

Thus there are 10 different interleavings, which can give 4 different values: 4, 5, 6, 7.

4 Question 4

Not necessarily. Consider the following execution path:

p	q	Value of x	Value of y
Test if (x = y), set x = x+1		1	10
Test if (x = y), set x = x+1		2	10
Test if (x = y), set x = x+1		3	10
Test if (x = y), set x = x+1		4	10
Test if (x = y), set x = x+1		5	10
Test if (x = y), set x = x+1		6	10
Test if (x = y), set x = x+1		7	10
Test if (x = y), set x = x+1		8	10
Test if (x = y), set x = x+1		9	10
Test if (x = y), get x		9	10
	Test if (x = y), get y	9	10
Set x to x+1		10	10
	Set y to y-1	10	9
...	...	...	...

From here execution will continue at least until overflow occurs. When that happens, a sequence like the one above can still occur again, making it possible for the process to execute forever.

5 Question 5

I assume `doDebit` is the second code listing shown. The problem appears in the following kind of situation: suppose initially our balance is 1. We then simultaneously debit 1 twice in two different threads. The threads may interleave themselves in such a way that both threads check if the account can debit before any one of them actually debits the account. They would then both debit the account by one unit. This would leave us with an overdrawn account, which was not expected (implicitly, I assume that we want to maintain a safety condition that guarantees a non-negative balance at all times). A way to fix this would be to make `doDebit` atomic as well. This would not allow the `canDebit` check to be separated from the call of `debit` in any potential interleaving, thus insuring that only valid `debit`'s are done.

6 Question 6

The code for the sorter, and the testing rig, follow:

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

object Sorter{
  /* This is the component specified by the question.
   * Out represents a channel through which the
   * output is written, and left/right have the
   * meaning defined in the question. */
  def sorting_component(out: ![Int], left: ?[Int], right: ![Int])
      = proc {

    // buffer represents the maximal value that we have seen so far.
    // It is set to None if and only if we have seen no
    // integers so far. Otherwise it is set to Some x, where
    // x is the maximal value seen so far.
    var buffer: Option[Int] = None

    // Until left is closed:
    repeat{
      // Read a value x from left
      val x = left?()

      // Now, based on the state of buffer:
      buffer match {
        // If we have seen nothing, then store x in buffer
        case None => buffer = Some (x)
        // Otherwise, if the largest value seen so far is y
        case Some(y) => {
          // Store max(x, y) in buffer
          buffer = Some (max(x, y))
          // And transmit min(x, y) through right
          right!min(x, y)
        }
      }
    }

    // Once left is closed, we close right
    right.closeOut()

    // and output the largest value we have seen.
    out!buffer.get
  }

  // This procedure sorts the Array it is given,
  // returning a sorted copy of the input,
  // using the method described in the question.
  def parallel_sort(a: Array[Int]) = {
    val n = a.size

    // These are the channels between components
    val inner_channels = Array.fill(n + 1)(OneOne[Int])

    // These are the channels through which the components
    // communicate with the testing function.
    val output_channels = Array.fill(n)(OneOne[Int])

    // This will contain the result of sorting.
    val results = ArrayBuffer.fill(n)(0)

    // This process copies a into the first inner channel, then closes it:
    val input_process = proc {
      for(x <- a)
        inner_channels(0)!x
      inner_channels(0).closeOut
    }

    // This is a process that runs all the sorting components
    // in parallel.
    val components = || (for(i <- 0 until n) yield
      sorting_component(
        output_channels(n - i - 1),
        inner_channels(i),
        inner_channels(i+1)))

    // This process will write the values in the
    // output channels to results.
    val output_process = || (for(i <- 0 until n) yield
      proc { results(i) = output_channels(i)?() })

    // the overall system
    val system = input_process ||
      components ||
      output_process

    run(system)

    results
  }
}

//This will test Sorter.
object Q6 extends App{
  var nr_tests = 0
  var test_size = 0
  try{
    println("Input number of tests: ")
    nr_tests = readInt
    println("Input test size: ")
    test_size = readInt
  }
  catch{
    case _ : Throwable => {
      println("Expected integer input, defaulting to 100 tests of size 100")
      nr_tests = 100
      test_size = 100
    }
  }

  val t0 = System.nanoTime()
  for(i <- 1 to nr_tests){
    val input = Array.fill(test_size)(scala.util.Random.nextInt)
    val output = Sorter.parallel_sort(input)
    assert(input.sorted sameElements output, "Test " + i + " failed")
    println("Test " + i + " succeeded")
  }
  val t1 = System.nanoTime()
  println("Average time: " + (((t1 - t0).toDouble / nr_tests.toDouble) * 1e-6) + " ms")
  exit()
}
```

6.1 Finding the number of sequentially ordered messages

Claim 1. *If we sort n integers, then the largest set of messages through* `inner_channels` *that must be sequentially ordered is of size n.*

Proof. Let $M(i, j)$ denote the i'th message passed through `inner_channels(j)`.
First, there exist n messages that must happen in a particular order; more precisely: $M(1, 0) \prec M(2, 0) \prec M(3, 0) \prec \ldots \prec M(n, 0)$, where $a \prec b$ asserts that a must happen strictly before b.
Second, there exists no set of more than n messages that must be ordered sequentially, as the following ordering of messages is possible:

- First $M(1, 0)$.
- Then $M(2, 0), M(1, 1)$ simultaneously.
- Then $M(3, 0), M(2, 1), M(1, 2)$ simultaneously.
- ...
- Then $M(n, 0), M(n - 1, 1), \ldots, M(1, n - 1)$ simultaneously.

And the longest sequentially ordered chain of messages in this ordering has length n.
So, overall, the largest set of sequentially ordered messages is of size n. □

If we take into account the messages through `out`, this gives us $n + 1$ sequentially ordered messages overall.

7 Question 7

The required code:

```
import io.threadcso._
import scala.math._
import scala.collection.mutable._
import scala.language.postfixOps

/* A trait that specifies what a matrix multiplier is */
trait Multiplier{
  def multiply(n: Int,
      a: Array[Array[Int]],
      b: Array[Array[Int]]) : Array[Array[Int]]
}

/* A multiplier that operates sequentially */
object Sequential_multiplier extends Multiplier{
  def multiply(n: Int,
      a: Array[Array[Int]],
      b: Array[Array[Int]]) = {
    require(a.size == n && b.size == n &&
      a(0).size == n && b(0).size == n)
    val c = Array.fill(n)(Array.fill(n)(0))
    for(i <- 0 until n)
      for(k <- 0 until n){
        c(i)(k) = 0
        for(j <- 0 until n)
          c(i)(k) += a(i)(j) * b(j)(k)
      }
    c
  }
}

/* This is a Gen-eric Par-allel Mult-iplier.
 * This multiplier considers computing the inner
 * products of some rows and some columns a task.
 * I have not defined the collector here, as there are two
 * different ways of collecting, with different performance
 * benefits. */
abstract class Gen_par_mult(nChunks: Int, nWorkers: Int)
        extends Multiplier{
  /* Task(ileft, iright, kleft, kright) represents that we should
   * calculate the submatrix that contains positions (i, k) that satisfy
   * ileft <= i < iright && kleft <= k < kright */
  case class Task(ileft: Int, iright:Int, kleft:Int, kright:Int)

  /* TaskAnswer(i, j, r) represents that
   * c(i)(j) should be r */
  case class TaskAnswer(i: Int, j: Int, res: Int)

  /* A bag of tasks. It splits the lines and columns into sets of chunks,
   * computes the cartesian product of these, and assigns each
   * pair of line chunk and column chunk to a task. */
  private class BagOfTasks(n: Int, nChunks: Int){
    private val toWorkers = OneMany[Task]

    // This process spawns the tasks
    private def server = proc {
      val chunkSz = (n + nChunks - 1) / nChunks

      // I'll divide the rows / columns into chunks of size "chunkSz".
      val chunks = Array.fill(nChunks)(List[Int]())

      for(i <- 0 until n by chunkSz)
        for(k <- 0 until n by chunkSz){
          /* This bucket contains lines from i to min(i + chunkSz, n)
           * and columns from k to min(k + chunkSz, n) */
          toWorkers!(Task(i, min(i+chunkSz, n), k, min(k+chunkSz, n)))
        }

      toWorkers.closeOut()
    }

    // get a task from the bag
    def getTask: Task = toWorkers?

    server.fork
  }

  /* This represents a generic collector */
  trait Collector{
    // add a TaskAnswer to the result
    def add(ta: TaskAnswer)

    // get the result matrix
    def get: Array[Array[Int]]
  }

  /* This worker will repeatedly find a task that consist of a range
   * of lines and columns, compute their inner products,
   * then tell the collector to set the corresponding
   * values in the output matrix to their correct values.
```

```
    * I allow it to see a and b as these are constant
    * throughout, and thus introduce no races */
  private def worker(n: Int, a: Array[Array[Int]], b: Array[Array[Int]],
        bag: BagOfTasks, col: Collector) = proc {
    repeat{
      val Task(ileft, iright, kleft, kright) = bag.getTask
      /* For all lines in our range */
      for(i <- ileft until iright)
        /* and all columns in our range */
        for(k <- kleft until kright){
          /* compute the inner product, storing it in res */
          var res = 0
          for(j <- 0 until n)
            res += a(i)(j) * b(j)(k)
          /* and output it to the collector */
          col.add(TaskAnswer(i, k, res))
        }
    }
  }

  // This abstract function will return a collector for an n by n matrix
  def mk_collector(n: Int): Collector

  // This simply puts all the components defined so far together,
  // in order to multiply two n by n matrices.
  def multiply(n: Int,

      a: Array[Array[Int]],
      b: Array[Array[Int]]) = {
    require(a.size == n && b.size == n &&
      a(0).size == n && b(0).size == n)

    val bag = new BagOfTasks(n, nChunks)
    val col = mk_collector(n)
    val workers = || (for (i <- 0 until nWorkers) yield worker(n, a, b, bag, col))
    workers()
    col.get
  }
}


/* This indirect parallel multiplier will use a collector
 * that accumulates the information from the workers
 * via messages (i.e. indirectly), writing them one by one
 * to the output matrix. */
class Indirect_par_mult(nChunks: Int, nWorkers: Int)
    extends Gen_par_mult(nChunks, nWorkers){
  /* This collector accumulates the information from the various
   * workers, then relays it to the client process. */
  class MessageCollector(n: Int) extends Collector{
    // This stores the comunication from the workers to the collector
    private val toCollector = ManyOne[TaskAnswer]

    // This stores the communication from the collector to the
    // client process
    private val result_chan = OneOne[Array[Array[Int]]]

    // Through this, a worker can signal that is has completed its task,
    // yielding a certain result.
    def add(ta: TaskAnswer) = toCollector!ta

    // This lets the client process find the result.
    def get = result_chan?

    // This process waits until all tasks are done, then signals the result
    // through result_chan.
    private def server = proc{
      val c = Array.fill(n)(Array.fill(n)(0))
      for(i <- 0 until n * n){
        val TaskAnswer(i, j, r) = toCollector?()
```

```scala
        c(i)(j) = r
      }
      result_chan!c
    }

    server.fork
  }

  def mk_collector(n: Int) = new MessageCollector(n)
}


/* This direct parallel multiplier will use a collector
 * that allows workers to directly write their results
 * without the use of messages. This is quite faster,
 * and does not lead to race conditions, since the
 * vaious tasks are always output-disjoint. */
class Direct_par_mult(nChunks: Int, nWorkers: Int)
    extends Gen_par_mult(nChunks, nWorkers){
  /* This collector allows direct access to the matrix c.
   * This does not lead to race conditions, as each task
   * is output-disjoint, and because the main function
   * waits for all workers to terminate before calling
   * get. */
  class DirectCollector(n: Int) extends Collector{
    private val c = Array.fill(n)(Array.fill(n)(0))
    def add(ta: TaskAnswer) = {
      val TaskAnswer(i, j, res) = ta
      c(i)(j) = res
    }

    def get = c
  }

  def mk_collector(n: Int) = new DirectCollector(n)
}


object Q7 extends App{
  val n = 2000
  val nWorkers = 4
  var nChunks = 4

  // This tests a multiplier m on two n by n matrices.
  // It returns the number of nanoseconds used.
  def test_multiplier(m: Multiplier): Long = {
    val a = Array.fill(n)(Array.fill(n)(scala.util.Random.nextInt(123)))
    val b = Array.fill(n)(Array.fill(n)(scala.util.Random.nextInt(123)))


    val t0 = System.nanoTime()
    val c = m.multiply(n, a, b)
    val t1 = System.nanoTime()
    val cc = Sequential_multiplier.multiply(n, a, b)

    for(i <- 0 until n)
      for(j <- 0 until n)
        assert(c(i)(j) == cc(i)(j), "Multiplier faulty")

    t1 - t0
  }

  if(args.size < 2){
    println("Need at least two arguments")
    println("Give argument help for help")
  }
  else{
    nChunks = args(1).toInt
    val time = args(0) match{
      case "help" =>{
        println("To use, pass two arguments. The first is one of: ")
        println("   sequential: sequential multiplier")
        println("   indirect: indirect parallel multiplier")
        println("   direct: direct parallel multiplier")
        println("The second is: number of chunks to split the lines / columns into")
        0
      }
      case "sequential" =>
        test_multiplier(Sequential_multiplier)
      case "direct" =>
        test_multiplier(new Direct_par_mult(nChunks, nWorkers))
      case "indirect" =>
        test_multiplier(new Indirect_par_mult(nChunks, nWorkers))
      case _ => {
        println("Bad argument")
        0
      }
    }
    println("Total time: " + time)
  }
  exit()
}
```

7.1 Experimental data

I tested the code on computer with a 2-core 2.3 GHz Intel Core i5, with 8 GB of RAM, and 4 logical threads. Each test consisted of multiplying two random 2000 by 2000 matrices. I tested using various numbers of subtasks; the time it took is in the table below, rounded to two decimal places, in seconds:

Multiplier variety / Number of subtasks	Direct	Indirect	Sequential
4	32.16	97.48	52.78
9	34.86	100.77	
16	32.20	97.45	
100	31.30	97.51	
400	31.40	98.39	
2500	29.91	97.00	
10000	29.18	96.57	
40000	29.31	95.54	
250000	33.70	96.40	
1000000	48.64	95.45	

From here we can deduce that:

- The indirect multiplier is extremely slow, probably due to the `ManyOne` channel used.
- The direct multiplier works best with around 10000 tasks, having worse performance for both fewer and more tasks.

8 Question 8

To achieve this:

1. Sort a.
2. Instantiate some number of workers.
3. Split b into a number of chunks equal to the number of available workers.
4. Send each worker a chunk of b.
5. Each worker now finds, for each x in its chunk of b, the number of times x appears in a, using binary search.
6. Each worker sends it's result back to the controller, which sums these to find the overall result.