8893 Concurrent Programming
Assignment #1 — Sample Solutions

Due: Tuesday, February 10 at 0900

1. 2.7 on p. 83.

   a) [10 points]

```
/**************************************************************
 * Memorial University of Newfoundland<br>
 * 8893 Concurrent Programming<br>
 * Assignment 1 Q1 (a) -- Andrews00 q 2.7(a)<br>
 * Sample solution.
 *
 * @version $Revision$ $Date$
 * @author Dennis Peters ($Author$)
 *
 * $RCSfile$
 * $State$
 *
 **************************************************************/
public class assign1_q7a
{
  public static final int n = 20; /** Size of arry */
  public static final int pr = 4;
  /** number of processes. Assumed to be a divisor of n */

/**************************************************************
 * @param args the command line arguments (not used)
 **************************************************************/
  public static void main(String[] args)
  {
    int[] a = new int[n]; // shared array
    Summer[] worker_impl = new Summer[pr];
    Thread[] worker = new Thread[pr]; // worker processes
    int total = 0; // total of all sums

    for (int i = 0; i < n; i++) {
      a[i] = (int)Math.round(Math.random()*2*n);
      // Fill the array with random values
      System.out.print(a[i] + ", ");
    }
    System.out.println();
```

```
    int stripSize = n/pr;
    for (int p = 0; p < pr; p++) {
      worker_impl[p] = new Summer(a, p*stripSize, stripSize);
      worker[p] = new Thread(worker_impl[p]);
    }
    for (int p = 0; p < pr; p++) {
      worker[p].start();
    }
    try {
      for (int p = 0; p < pr; p++) {
        while (worker[p].isAlive()) Thread.sleep(50);
        total += worker_impl[p].getSum();
      }
    }
    catch (InterruptedException e) {}

    System.out.print("Sum = " + total);
    if (Oracle(a, total)) {
      System.out.println(" correct!");
    } else {
      System.out.println(" false!");
    }
    System.exit(0);
  }

  /**
   * Test to see if the results are correct.
   * @return true if the result is correct, false otherwise
   */
  private static boolean Oracle(int[] a, int sum)
  {
    int actual = 0;
    for (int i = 0; i < n; i++) {
      actual += a[i];
    }
    return (actual == sum);
  }
}

class Summer
  implements Runnable
{
  private int[] a;  /** shared array */
  private int first; /** start index for my stripe */
```

```
  private int num; /** number of elements to sum */
  private int sum; /** result */

/**********************************************************************
 * @param a_ the array to be shared.
 * @param first_ the index for this instance to start at
 * @param num_ the number of elements to sum
 **********************************************************************/
  Summer(int[] a_, int first_, int num_)
  {
    a = a_;
    first = first_;
    num = num_;
    sum = 0;
  }

/**********************************************************************
 * Method invoked by start.
 **********************************************************************/
  public void run()
  {
    int indx = first;
    for (int i = 0; i < num; i++) {
      sum += a[indx];
      indx++;
    }
  }

/**********************************************************************
 * Return the sum computed by this instance.
 **********************************************************************/
  public int getSum()
  {
    return sum;
  }

}

/**********************************************************************
 *                      REVISION HISTORY
 *
 * $Log$
 *
 **********************************************************************/
```

b) [10 points]

```
/**********************************************************************
 * Memorial University of Newfoundland<br>
 * 8893 Concurrent Programming<br>
 * Assignment 1 Q1 (b) -- Andrews00 q 2.7(b)<br>
 * Sample solution.
 *
 * @version $Revision$ $Date$
 * @author Dennis Peters ($Author$)
 *
 * $RCSfile$
 * $State$
 *
 **********************************************************************/
public class assign1_q7b
{
  public static final int n = 40; /** Size of arry */
  public static final int pr = 5;
  /** number of processes. Assumed to be a divisor of n */

/**********************************************************************
 * @param args the command line arguments (not used)
 **********************************************************************/
  public static void main(String[] args)
  {
    int[] a = new int[n]; // shared array
    Summer worker_impl = new Summer(a, 0, n);
    Thread worker = new Thread(worker_impl); // worker process

    for (int i = 0; i < n; i++) {
      a[i] = (int)Math.round(Math.random()*2*n);
      // Fill the array with random values
      System.out.print(a[i] + ", ");
    }
    System.out.println();

    worker.start();
    try {
      while (worker.isAlive()) Thread.sleep(50);
    }
    catch (InterruptedException e) {}

    System.out.print("Sum = " + worker_impl.getSum());
    if (Oracle(a, worker_impl.getSum())) {
      System.out.println(" correct!");
```

```
    } else {
      System.out.println(" false!");
    }
    System.exit(0);
  }

  /**
   * Test to see if the results are correct.
   * @return true if the result is correct, false otherwise
   */
  private static boolean Oracle(int[] a, int sum)
  {
    int actual = 0;
    for (int i = 0; i < n; i++) {
      actual += a[i];
    }
    return (actual == sum);
  }
}

class Summer
  implements Runnable
{
  private int[] a;  /** shared array */
  private int first; /** start index for my stripe */
  private int num; /** number of elements to sum */
  private int sum; /** result */

/*****************************************************************
 * @param a_ the array to be shared.
 * @param first_ the index for this instance to start at
 * @param num_ the number of elements to sum
 *****************************************************************/
  Summer(int[] a_, int first_, int num_)
  {
    a = a_;
    first = first_;
    num = num_;
    sum = 0;
  }

/*****************************************************************
 * Method invoked by start.
 *****************************************************************/
  public void run()
```

```
  {
    if (num <= assign1_q7b.n/assign1_q7b.pr) {
      // base case, num < threshold
      int indx = first;
      for (int i = 0; i < num; i++) {
        sum += a[indx];
        indx++;
      }
    } else {
      // recursive case, split list in half.
      Summer left = new Summer(a, first, num/2);
      Summer right = new Summer(a, first+num/2, num-num/2);
      Thread left_th = new Thread(left);
      Thread right_th = new Thread(right);
      left_th.start();
      right_th.start();
      try {
        while (left_th.isAlive()) Thread.sleep(50);
        while (right_th.isAlive()) Thread.sleep(50);
      }
      catch (InterruptedException e) {}
      sum = left.getSum() + right.getSum();
    }
  }

/**********************************************************************
 * Return the sum computed by this instance.
 **********************************************************************/
  public int getSum()
  {
    return sum;
  }

}

/**********************************************************************
 *                      REVISION HISTORY
 *
 * $Log$
 *
 **********************************************************************/
```

2. 2.13 on p. 85. For this question it was sufficient to simply work through each of the possibilities, but here are the proof outlines.

a) [5 points]

$\#\{\mathtt{x} = 2 \wedge \mathtt{y} = 5\}$
`x = x + y;`
$\#\{\mathtt{x} = 7 \wedge \mathtt{y} = 5\}$
`y = x - y;`
$\#\{\mathtt{x} = 7 \wedge \mathtt{y} = 2\}$
`x = x - y;`
$\#\{\mathtt{x} = 5 \wedge \mathtt{y} = 2\}$

b) [5 points]

$\#\{\mathtt{x} = 2 \wedge \mathtt{y} = 5\}$
`co`

$$\#\left\{\begin{array}{l}(\mathtt{x} = 2 \wedge \mathtt{y} = 5) \vee (\mathtt{x} = 2 \wedge \mathtt{y} = -3) \vee (\mathtt{x} = -3 \wedge \mathtt{y} = 5)\\ \vee(\mathtt{x} = 5 \wedge \mathtt{y} = -3) \vee (\mathtt{x} = -3 \wedge \mathtt{y} = -8)\end{array}\right\}$$

`< x = x + y; >`

$$\#\left\{\begin{array}{l}(\mathtt{x} = 7 \wedge \mathtt{y} = 5) \vee (\mathtt{x} = -1 \wedge \mathtt{y} = -3) \vee (\mathtt{x} = 2 \wedge \mathtt{y} = 5)\\ \vee(\mathtt{x} = 2 \wedge \mathtt{y} = -3) \vee (\mathtt{x} = -11 \wedge \mathtt{y} = -8)\\ \vee(\mathtt{x} = 7 \wedge \mathtt{y} = 2) \vee (\mathtt{x} = 2 \wedge \mathtt{y} = -3) \vee (\mathtt{x} = 2 \wedge \mathtt{y} = -3)\\ \vee(\mathtt{x} = 2 \wedge \mathtt{y} = 5) \vee (\mathtt{x} = 2 \wedge \mathtt{y} = -3) \vee (\mathtt{x} = 5 \wedge \mathtt{y} = 2)\end{array}\right\}$$

`//`

$$\#\left\{\begin{array}{l}(\mathtt{x} = 2 \wedge \mathtt{y} = 5) \vee (\mathtt{x} = 7 \wedge \mathtt{y} = 5) \vee (\mathtt{x} = -3 \wedge \mathtt{y} = 5)\\ \vee(\mathtt{x} = 2 \wedge \mathtt{y} = 5) \vee (\mathtt{x} = 2 \wedge \mathtt{y} = 5)\end{array}\right\}$$

`< y = x - y; >`

$$\#\left\{\begin{array}{l}(\mathtt{x} = 2 \wedge \mathtt{y} = -3) \vee (\mathtt{x} = 7 \wedge \mathtt{y} = 2) \vee (\mathtt{x} = -3 \wedge \mathtt{y} = -8)\\ \vee(\mathtt{x} = 2 \wedge \mathtt{y} = -3) \vee (\mathtt{x} = 2 \wedge \mathtt{y} = -3)\\ \vee(\mathtt{x} = -1 \wedge \mathtt{y} = -3) \vee (\mathtt{x} = -11 \wedge \mathtt{y} = -8) \vee (\mathtt{x} = 2 \wedge \mathtt{y} = -3)\\ \vee(\mathtt{x} = 5 \wedge \mathtt{y} = -3) \vee (\mathtt{x} = 5 \wedge \mathtt{y} = 2) \vee (\mathtt{x} = 2 \wedge \mathtt{y} = -3)\end{array}\right\}$$

`//`

$$\#\left\{\begin{array}{l}(\mathtt{x} = 2 \wedge \mathtt{y} = 5) \vee (\mathtt{x} = 7 \wedge \mathtt{y} = 5) \vee (\mathtt{x} = 2 \wedge \mathtt{y} = -3)\\ \vee(\mathtt{x} = 7 \wedge \mathtt{y} = 2) \vee (\mathtt{x} = -1 \wedge \mathtt{y} = -3)\end{array}\right\}$$

`< x = x - y; >`

$$\#\left\{\begin{array}{l}(\mathtt{x} = -3 \wedge \mathtt{y} = 5) \vee (\mathtt{x} = 2 \wedge \mathtt{y} = 5) \vee (\mathtt{x} = 5 \wedge \mathtt{y} = -3)\\ \vee(\mathtt{x} = 5 \wedge \mathtt{y} = 2) \vee (\mathtt{x} = 2 \wedge \mathtt{y} = -3)\\ \vee(\mathtt{x} = 2 \wedge \mathtt{y} = 5) \vee (\mathtt{x} = 2 \wedge \mathtt{y} = -3) \vee (\mathtt{x} = -11 \wedge \mathtt{y} = -8)\\ \vee(\mathtt{x} = -3 \wedge \mathtt{y} = -8) \vee (\mathtt{x} = 2 \wedge \mathtt{y} = -3) \vee (\mathtt{x} = 2 \wedge \mathtt{y} = -3)\end{array}\right\}$$

`oc`

$$\#\left\{\ (\mathtt{x} = 2 \wedge \mathtt{y} = -3) \vee (\mathtt{x} = -11 \wedge \mathtt{y} = -8) \vee (\mathtt{x} = 5 \wedge \mathtt{y} = 2)\ \right\}$$

The pre-conditions for each branch of the co-oc are formed by first considering the case where that branch executes first ($\mathtt{x} = 2 \wedge \mathtt{y} = 5$), and then considering each of the other possibile sequences. The first two lines of each post-condition are calculated

simply by applying the statement to the pre-conditions. The third and fourth lines of the post-conditions are added to weaken the post-condition so that it is not interfered with by either or both of the other two statements in either order. When these post-conditions are conjoined ("anded") they reduce to give the three possible final states given.

c) [5 points]

$\#\{\mathtt{x} = 2 \wedge \mathtt{y} = 5\}$
`co`
  $\#\{(\mathtt{x} = 2 \wedge \mathtt{y} = 5) \vee (\mathtt{x} = -3 \wedge \mathtt{y} = 5)\}$
  `< await (x > y)`
    `x = x + y;`
    `y = x - y; >`
  $\#\{\underline{false}\}$
`//`
  $\#\{\mathtt{x} = 2 \wedge \mathtt{y} = 5\}$
  `< x = x - y; >`
  $\#\{\mathtt{x} = -3 \wedge \mathtt{y} = 5\}$
`oc`
$\#\{\underline{false}\}$

This program fails to terminate since the condition $\mathtt{x} > \mathtt{y}$ is never true.

3. [10 points] 2.16 on p. 86.
For this proof we introduce a thought variable `t` to keep track of if the increment in the last `await` statement has happened yet.

```
int x = 0;
int t = 0; # thought variable
## { x == 0 /\ t == 0 }
co
  ## { x == 0 \/ t == 1 /\ (x == 5 \/ x == 2) }
  < await (x != 0) x = x - 2; >
  ## { t == 1 /\ (x == 3 \/ x == 0) }
//
  ## { x == 0 \/ t == 1 /\ (x == 5 \/ x == 3) }
  < await (x != 0) x = x - 3; >
  ## { t == 1 /\ (x == 2 \/ x == 0) }
//
  ## { x == 0 /\ t == 0 }
  < await (x == 0) x = x + 5; t = 1; >
  ## { t == 1 /\ (x == 0 \/ x == 2 \/ x == 3 \/ x == 5) }
oc
## { x == 0 }
```

First convince yourself that each of the triples above is valid. To do this we apply the `await` statement rule together with the assignment axiom. For example, for the first triple we need to show that
$\{(\mathtt{x} == 0 \vee \mathtt{t} == 1 \wedge (\mathtt{x} == 5 \vee \mathtt{x} == 3)) \wedge \mathtt{x} \neq 0\}$
$\langle \mathtt{x} = \mathtt{x} - 2 \rangle$
$\{\mathtt{t} == 1 \wedge (\mathtt{x} == 3 \vee \mathtt{x} == 0)\}$
is valid. The pre-condition reduces to $\mathtt{t} == 1 \wedge (\mathtt{x} == 5 \vee \mathtt{x} == 3)$, and by the assignment axiom we get the post condition. Note that in the case of the third branch the post-condition is much weaker than what we know (i.e., we could show $\mathtt{t} == 1 \wedge \mathtt{x} == 5$) for the purposes of proving non-interference.

Non-interference:
Consider each of the assignment actions above, using the definition of noninterference (text 2.5 p. 64).

1. $\{(\mathtt{x} == 0 \vee \mathtt{t} == 1 \wedge (\mathtt{x} == 5 \vee \mathtt{x} == 3)) \wedge (\mathtt{x} == 0 \vee \mathtt{t} == 1 \wedge (\mathtt{x} == 5 \vee \mathtt{x} == 2))\}$
   $\langle \mathtt{await}(\mathtt{x} \neq 0)\mathtt{x} = \mathtt{x} - 2; \rangle$
   $\{\mathtt{x} == 0 \vee \mathtt{t} == 1 \wedge (\mathtt{x} == 5 \vee \mathtt{x} == 3)\}$
   The pre-condition reduces to $\mathtt{x} == 0 \vee \mathtt{t} == 1 \wedge \mathtt{x} == 5$ and the action gives $\mathtt{t} == 1 \wedge \mathtt{x} == 3$, which implies the post-condition.
2. $\{(\mathtt{t} == 1 \wedge (\mathtt{x} == 2 \vee \mathtt{x} == 0)) \wedge (\mathtt{x} == 0 \vee \mathtt{t} == 1 \wedge (\mathtt{x} == 5 \vee \mathtt{x} == 2))\}$
   $\langle \mathtt{await}(\mathtt{x} \neq 0)\mathtt{x} = \mathtt{x} - 2; \rangle$
   $\{\mathtt{t} == 1 \wedge (\mathtt{x} == 2 \vee \mathtt{x} == 0)\}$

In a similar manner, to above.

3. $\{(\mathtt{x} == 0 \wedge \mathtt{t} == 0) \wedge (\mathtt{x} == 0 \vee \mathtt{t} == 1 \wedge (\mathtt{x} == 5 \vee \mathtt{x} == 2))\}$
   $\langle \mathtt{await}(\mathtt{x} \neq 0)\mathtt{x} = \mathtt{x} - 2; \rangle$
   $\{\mathtt{x} == 0 \wedge \mathtt{t} == 0\}$

   In this case the pre-condition reduces to $\mathtt{x} == 0 \wedge \mathtt{t} == 0$, which, when we apply the `await` statement rule reduces to *false*, meaning that the triple is valid for any `await` statement body and post-condition (i.e., it's impossible for the `await` statement to execute from that state, so the code is correct no matter what it does).

4. $\left\{\begin{array}{l}(\mathtt{t} == 1 \wedge (\mathtt{x} == 0 \vee \mathtt{x} == 2 \vee \mathtt{x} == 3 \vee \mathtt{x} == 5)) \wedge \\ (\mathtt{x} == 0 \vee \mathtt{t} == 1 \wedge (\mathtt{x} == 5 \vee \mathtt{x} == 2))\end{array}\right\}$
   $\langle \mathtt{await}(\mathtt{x} \neq 0)\mathtt{x} = \mathtt{x} - 2; \rangle$
   $\{\mathtt{t} == 1 \wedge (\mathtt{x} == 0 \vee \mathtt{x} == 2 \vee \mathtt{x} == 3 \vee \mathtt{x} == 5)\}$

   This is similar to the first two triples, above. The pre-condition reduces to $\mathtt{t} == 1 \wedge (\mathtt{x} == 5 \vee \mathtt{x} == 2)$, giving the post condition of $\mathtt{t} == 1 \wedge (\mathtt{x} == 3 \vee \mathtt{x} == 0)$.

5. $\{(\mathtt{x} == 0 \vee \mathtt{t} == 1 \wedge (\mathtt{x} == 5 \vee \mathtt{x} == 2)) \wedge (\mathtt{x} == 0 \vee \mathtt{t} == 1 \wedge (\mathtt{x} == 5 \vee \mathtt{x} == 3))\}$
   $\langle \mathtt{await}(\mathtt{x} \neq 0)\mathtt{x} = \mathtt{x} - 3; \rangle$
   $\{\mathtt{x} == 0 \vee \mathtt{t} == 1 \wedge (\mathtt{x} == 5 \vee \mathtt{x} == 2)\}$

   The pre-condition reduces to $\mathtt{x} == 0 \vee \mathtt{t} == 1 \wedge \mathtt{x} == 5$, and using the `await` rule as above we can show that the post-condition is $\mathtt{t} == 1 \wedge \mathtt{x} == 2$, which implies the desired condition.

6. $\{(\mathtt{t} == 1 \wedge (\mathtt{x} == 3 \vee \mathtt{x} == 0)) \wedge (\mathtt{x} == 0 \vee \mathtt{t} == 1 \wedge (\mathtt{x} == 5 \vee \mathtt{x} == 3))\}$
   $\langle \mathtt{await}(\mathtt{x} \neq 0)\mathtt{x} = \mathtt{x} - 3; \rangle$
   $\{\mathtt{t} == 1 \wedge (\mathtt{x} == 3 \vee \mathtt{x} == 0)\}$

   As for the previous case, here we have the post condition of $\mathtt{t} == 1 \wedge \mathtt{x} == 0$.

7. $\{(\mathtt{x} == 0 \wedge \mathtt{t} == 0) \wedge (\mathtt{x} == 0 \vee \mathtt{t} == 1 \wedge (\mathtt{x} == 5 \vee \mathtt{x} == 3))\}$
   $\langle \mathtt{await}(\mathtt{x} \neq 0)\mathtt{x} = \mathtt{x} - 3; \rangle$
   $\{\mathtt{x} == 0 \wedge \mathtt{t} == 0\}$

   As for 3 above.

8. $\left\{\begin{array}{l}(\mathtt{t} == 1 \wedge (\mathtt{x} == 0 \vee \mathtt{x} == 2 \vee \mathtt{x} == 3 \vee \mathtt{x} == 5)) \wedge \\ (\mathtt{x} == 0 \vee \mathtt{t} == 1 \wedge (\mathtt{x} == 5 \vee \mathtt{x} == 3))\end{array}\right\}$
   $\langle \mathtt{await}(\mathtt{x} \neq 0)\mathtt{x} = \mathtt{x} - 3; \rangle$
   $\{\mathtt{t} == 1 \wedge (\mathtt{x} == 0 \vee \mathtt{x} == 2 \vee \mathtt{x} == 3 \vee \mathtt{x} == 5)\}$

   Similar to 4 above, the pre-condition reduces to $\mathtt{t} == 1 \wedge (\mathtt{x} == 5 \vee \mathtt{x} == 3)$ giving a post-condition of $\mathtt{t} == 1 \wedge (\mathtt{x} == 2 \vee \mathtt{x} == 3)$.

9. $\{(\mathtt{x} == 0 \vee \mathtt{t} == 1 \wedge (\mathtt{x} == 5 \vee \mathtt{x} == 2)) \wedge (\mathtt{x} == 0 \wedge \mathtt{t} == 0)\}$
   $\langle \mathtt{await}(\mathtt{x} == 0)\mathtt{x} = \mathtt{x} + 5; \mathtt{t} = 1; \rangle$
   $\{\mathtt{x} == 0 \vee \mathtt{t} == 1 \wedge (\mathtt{x} == 5 \vee \mathtt{x} == 2)\}$

   The pre-condition reduces to $\mathtt{x} == 0 \wedge \mathtt{t} == 0$ which by the `await` rule and assignment axiom gives $\mathtt{t} == 1 \wedge \mathtt{x} == 5$ which implies the desired post-condition.

10. $\{(\texttt{t} == 1 \wedge (\texttt{x} == 3 \vee \texttt{x} == 0)) \wedge (\texttt{x} == 0 \wedge \texttt{t} == 0)\}$
$\langle \texttt{await}(\texttt{x} == 0)\texttt{x} = \texttt{x} + 5; \texttt{t} = 1; \rangle$
$\{\texttt{t} == 1 \wedge (\texttt{x} == 3 \vee \texttt{x} == 0)\}$
The pre-condition reduces to *false*, so any code and post-condition makes a valid triple (i.e., the pre-condition can't happen, so the code is not expected to do anything about it).

11. $\{(\texttt{x} == 0 \vee \texttt{t} == 1 \wedge (\texttt{x} == 5 \vee \texttt{x} == 3)) \wedge (\texttt{x} == 0 \wedge \texttt{t} == 0)\}$
$\langle \texttt{await}(\texttt{x} == 0)\texttt{x} = \texttt{x} + 5; \texttt{t} = 1; \rangle$
$\{\texttt{x} == 0 \vee \texttt{t} == 1 \wedge (\texttt{x} == 5 \vee \texttt{x} == 3)\}$
As for 9 above.

12. $\{(\texttt{t} == 1 \wedge (\texttt{x} == 2 \vee \texttt{x} == 0)) \wedge (\texttt{x} == 0 \wedge \texttt{t} == 0)\}$
$\langle \texttt{await}(\texttt{x} == 0)\texttt{x} = \texttt{x} + 5; \texttt{t} = 1; \rangle$
$\{\texttt{t} == 1 \wedge (\texttt{x} == 2 \vee \texttt{x} == 0)\}$
As for 10 above.

Finally, having shown that the conditions are interference free, the co-begin rule allows us to conjoin (i.e., "and") the pre-conditions to get the pre-condition for the co-begin and similarly for the post-conditions to get the post-condition of $\texttt{x} == 0$.

4. [5 points] 3.7 on p. 144.

```
1   int lock = 0;
2   process CS[i = 1 to n] {
3     while (true) {
4       < await (lock == 0) >; lock = i; Delay;
5       while (lock != i) {
6         < await (lock == 0) >; lock = i; Delay;
7       }
8       critical section;
9       lock = 0;
10      noncritical section;
11    }
12  }
```

a) Without the Delay code this protocol does **not** ensure mutual exclusion. Consider the case where the await statement in line 4 is enabled in two processes, say i = 1 and 2, at the same time, and both processes get to execute it (in either order). Now if one, say 1, completes line 4 and goes on to enter its critical section, then 2 completes line 4 it too will get to enter its critical section, so mutual exclusion is violated.

This protocol **does** avoid deadlock. For deadlock to occur requires that `lock` $\neq 0$ and all processes are waiting on one of the await statements. If a process is waiting at the await on line 4, then either it has last set `lock` to 0, or at least one process is in the critical section, in which case `lock` $= i$, and that process will eventually leave the critical section and set `lock` $= 0$, so the process will get to proceed. If all but one processes are waiting at the await on line 6, then the last process will not enter the `while` loop, and thus will evenutally exit the critical section and release another process.

The protocol **does** avoid unnecessary delay. If no processes are in their critical section then `lock` $= 0$, so any process wishing to enter will be allowed to do so immediately.

Eventual entry is **not** assured by this protocol, even if the scheduler is strongly fair (although I accepted it if you said it was). The condition in the await will not stay continuously true if any process gets to enter its critical section, so under weakly fair scheduling some process may be starved at the `await`. With strongly fair scheduling no process will be starved at the `await`, but there is no assurance that a particular process will ever 'win' the race to be the last to assign to `lock`, so a process may starve anyway.

b) With the given assumptions about Delay, this protocal **does** ensure mutual exclusion. If more than one process is waiting before line 4 when `lock` becomes 0, then one or more will be released, and the last one to execute `lock = i` will, after Delay, get to enter its critical section. All other waiting processes will wait in the loop (lines 5 – 7) until they 'win' by being the last to execute `lock = i`.

As above, deadlock is avoided by ensuring that there is always one 'winner' who will get to enter the critical section.

The protocol does introduce a small amount of unecessary delay (or is it necessary?), in the Delay statement, which will be executed even if there are no other processes waiting. As above, eventual entry is not assured even with a strongly fair scheduler.