Dekker's Algorithm

Last updated: January 16th 2019, at 11.19am

Nondeterministic results from concurrent processes

Consider the following code, explained in more detail below:

```
/**
   * Define a process with two shared variables.
   * This class is abstract because it does not implement
   * run.
   */

public abstract class Process extends Thread {
   protected static int x = 0;
   protected static int y = 2;
}

/**
   * One version of the process — assign y+1 to x.
   */

public class Process1 extends Process {
   public void run() {
      x = y + 1;
   }
}

/**
   * One version of the process — assign x+1 to y.
   */
```

```
public class Process2 extends Process {
   public void run() {
      y = x + 1;
   }
}


/**
   * Run both processes in parallel
   */


public class Tutorial14 {
   public static void main(String[] args) {
      Process1 process1 = new Process1();
      Process2 process2 = new Process2();
      process1.start(); process2.start();
      try {
         process1.join(); process2.join();
      }  catch (InterruptedException ie) {}
   }
}
```

Two processes are defined (in `Process1` and `Process2`) which share the two variables declared in the **abstract** `Process` class. These two processes each calculate new values for the shared variables. Note that these two assignments do *not* satisfy Reynold's criterium, as there are two critical references in each assignment (e.g. in `Process1`'s assignment there is a read reference of the shared variable `y` *and* a write reference of the shared variable `x`). Two instances of these processes (one of each) are created and run in parallel in the `main` method of the `Tutorial14` class. This question is about the possible values of the shared variables once both processes have terminated.

1. Rewrite the two parallel processes so that there is at most one critical reference in each line of code. You may introduce new local variables if necessary. E.g. in `Process1` you might define a local temporary variable, $\texttt{p}_1$, say, and assign `y+1` to $\texttt{p}_1$, before assigning this to `x`. Alternatively you could assign `y` to $\texttt{p}_1$, and then assign $\texttt{p}_1 + 1$ to `x`. Does it make any difference which way you do it?

2. Using your code as a model for the implementation of the processes and assuming that Reynold's criterium is met, what are the possible values for `x` and `y` upon termination of this programme? I.e. assuming that each line of your code can be considered as an atomic action (cannot be interrupted), consider all the possible interleavings of the lines of code in

the rewritten processes, and trace the values of the variables through these interleavings.

3. Reynold's criterium may not be met. Assume that this programme is implemented on some one address machine, with `x = y + 1` implemented as

```
LDA :y:   ;load y into the accumulator
ADD &1    ;add one to the value in the accumulator
STA :x:   ;store the result from the accumulator back into x
```

   and `y = x + 1` implemented similarly, with the rôles of `x` and `y` reversed.

   Note: If you are not familiar with one address machines, the *accumulator* is a special location in the machine in which all arithmetic calculations are done. In the programme above, to add one to the value of `y` and store the result in `x` we have to

   - copy the current value of `y` to the accumulator (`LDA :y:`),
   - add one to the value in the accumulator (no other value changes),
   - copy the current value in the accumulator back to `x` (`STA :x:`).

   Assume that a single machine instruction (e.g. `ADD &1`) is an indivisible action.

   (a) Show how this programme can terminate with `x` having the value 0 and `y` having the value 1.

   (b) Why does this implementation not satisfy Reynold's criterium?

Ferrocarriles de América del sur

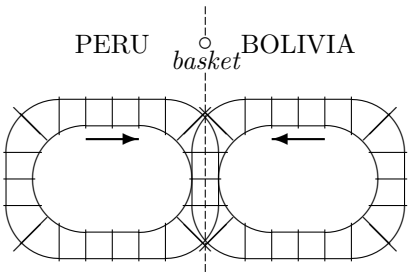

Figure 1: The Peru-Bolivia Railway

High in the Andes there are two oval railways. As shown in figure 1 on page 3 one of the railways is in Peru and the other in Bolivia. They share one section

```
public abstract class Railway extends Thread {
   private String name;
   private static Basket sharedBasket = new Basket("shared basket");
   private Basket basket;

   ...

   public void run() {
      errorFlag = false;
      try {
         runTrain();
      }  catch (RailwaySystemError error) {
         errorFlag = true;
         errorMessage = error.getMessage();
         System.out.println("!!! Something went wrong with the railway. " + errorMessage);
      }
      if (errorFlag) {
         System.out.println(name() + " shut down because of an error. " + errorMessage);
      }  else {
         System.out.println(name() + " shut down because time limit was reached.");
      }
   }

   public abstract void runTrain() throws RailwaySystemError;

}
```

Figure 2: Code for an abstract railway system

of track where the railways cross a mountain pass that lies on the international border. There is exactly one train on each railway. The arrows indicate the direction of travel.

Unfortunately the Peruvian and Bolivian trains crash into each other every now and then, when they both enter the critical section (the mountain pass) of the railway. The cause is that, strange as it may seem, both engine drivers are blind and deaf and can neither see nor hear each other.

Basic code[1] for this railway system is shown in figure 2 on page 4. This code defines a `Railway` superclass. The behaviour of a specific railway will be determined by its implementation of the `runTrain()` method.

Possible code for this method for the Bolivian and Peruvian railways is shown in figure 3 on page 5.[2] This implementation makes no attempt to protect the

[1] An implementation of this code is available in this week's packages.

[2] Note: this code does not make best use of OO inheritance, in that the `runTrain()` methods

```
public class Bolivia extends Railway {

   ...

   public void runTrain() throws RailwaySystemError {
      Clock clock = getRailwaySystem().getClock();
      while (!clock.timeOut() && !errorFlag) {
         choochoo();
         crossPass();
      }
   }
}

public class Peru extends Railway {

   ...

   public void runTrain() throws RailwaySystemError {
      Clock clock = getRailwaySystem().getClock();
      while (!clock.timeOut() && !errorFlag) {
         choochoo();
         crossPass();
      }
   }
}
```

Figure 3: Possible implementations of the Peruvian and Bolivian railways

pass.

The two engine drivers agree that this is not acceptable, and decide on the following method for avoiding crashes. They put a large basket at the entrance to the pass. Before entering the the pass the engine driver must stop, walk to the basket and feel if there is a stone in the basket. If the basket is empty the engine driver finds a stone and puts it in the basket in order to indicate that he is about to enter the pass. As soon as he has crossed the pass he walks back and retrieves his stone in order to indicate that the pass is free. Finally he walks back to his train and continues his journey.

If one of the engine drivers finds a stone already in the basket he leaves it in there, and has a siesta. When he wakes up again he repeats the actions he took when he arrived. The `runTrain()` method used by both `Bolivia` and `Peru` is

for `Peru` and `Bolivia` are identical. This has been done here because some later implementations of `Bolivia` and `Peru` *will* have differing `runTrain()` methods, and the differences between the implementations are clearer when maintaining a symmetry between the two railways.

```
public void runTrain() throws RailwaySystemError {
   Clock clock = getRailwaySystem().getClock();
   while (!clock.timeOut() && !errorFlag) {
      choochoo();
      while (getSharedBasket().hasStone(this)) {
         siesta();
      }
      getSharedBasket().putStone(this);
      crossPass();
      getSharedBasket().takeStone(this);
   }
}
```

Figure 4: A `runTrain()` method shared between `Peru` and `Bolivia` that uses stones in a shared basket as a message system. A stone in the basket indicates that there is a train in the pass.

shown in figure 4 on page 6.

1. There were soon problems with this system.

   (a) The Bolivians complained that subversive timetables put together by the Peruvian railways could block the Bolivian trains for ever.

   Give a scenario, using the code from figure 4 on page 6, that shows how this could happen.

   (b) More seriously, one day the trains crashed.

   Give a scenario, using the code from figure 4 on page 6, that shows how this could happen.

2. After the crash the train drivers agreed on a different system. The Peruvian engine driver would wait by the entrance to the pass until the basket is empty, cross the pass, and then walk back to put a stone in the basket. The Bolivian engine driver would wait until there is a stone in the basket, then cross the pass, and then walk back to take the stone out of the basket.

   (a) Provide code for the `Peru` and `Bolivia` classes for this solution.

   (b) Does this method prevent crashes?

   (c) There was immediately a dispute between Peru and Bolivia about timetabling.

   Explain why.

3. The train companies agreed that they needed expert help, and hired a consultant, a graduate of Russell University. He proposed using two baskets,

```
public void runTrain() throws RailwaySystemError {
   Clock clock = getRailwaySystem().getClock();
   Railway nextRailway = getRailwaySystem().getNextRailway(this);
   while (!clock.timeOut() && !errorFlag) {
      choochoo();
      while (nextRailway.getBasket().hasStone(this)) {
         siesta();
      }
      getBasket().putStone(this);
      crossPass();
      getBasket().takeStone(this);
   }
}
```

Figure 5: The `runTrain()` method for both `Peru` amd `Bolivia` (it is the same for both railways) as used in question 3.

one for each engine driver. If an engine driver came to the entrance to the pass he must first check the other driver's basket to see if it is empty. If it is he puts a stone in his own basket, crosses the pass and then walks back to remove the stone from his basket.

If there is a stone in the other driver's basket, he has a siesta, then checks the other driver's basket again, repeating this process until the other driver's basket is empty. when it is he puts a stone in his own basket, crosses the pass and then walks back to remove the stone from his basket.

Code for this proposal is shown in figure 5 on page 7.

Unfortunately, the trains now crashed again. Explain how.

4. They went back to the consultant, who told them to put stones in their own baskets *before* checking the other driver's basket.

   (a) Give code for this attempted soloution.

   (b) One day, the trains stopped running. Explain how this can happen.

5. The consultant was asked for advice again which, for a fat fee, he provided. He told the drivers that, of course, before having a siesta they should remove the stone from their basket, and put it back in again as soon as they wake up.

   (a) **Model exercise.** Provide code for this "solution".

   (b) This was an improvement on the previous attempts but, after some experience with it, the railways were still not happy. Why not?

   (c) Provide a scenario to demonstrate your answer to question 5b.

6. **Logbook exercise.** The train companies sacked the consultant, and hired a recent Huddersfield graduate. She immediately saw the solution, having paid careful attention to Hugh's lecture on Dekker's algorithm.

   Implement her solution.

End of Dekker's algorithm tutorial