Concurrent Programming Practical 1: Monitors and Semaphores

The aim of this practical is to model the following scenario using monitors and/or semaphores.

A tutor is sleeping in his room, waiting for two students to arrive for a tutorial.[1] The first student to arrive sleeps until the tutorial partner arrives. The second to arrive wakes the first, and one of them wakes the tutor. The two students then sleep while the tutor gives the tutorial. At the end of the tutorial, the two students wake up and leave. The tutor sleeps until the students arrive for their next tutorial. For simplicity, we will assume that there is just a single pair of students, and that they repeatedly return for tutorials.

An outline of code to capture this scenario is defined in Figure 1 (page 3) as the trait `SleepingTutor`. Your solution is likely to be an object that extends the trait. To be precise, your task is to implement the four procedures:

`TutorWait` The tutor waits for the students to arrive;

`Arrive` A student arrives;

`ReceiveTute` A student receives a tutorial;

`EndTeach` The tutor finished the tutorial.

The requirements to be enforced are:

- The tutor starts to teach only after both students have arrived;
- The students leave only after the tutor ends the tutorial.

You should implement these procedures either

- using a controller built in the monitor style, or
- using boolean semaphores.

[1] We suspect that many students believe their tutor to be like the light inside a refrigerator – inasmuch as it turns on only when the door is opened. We regret that the scenario outlined here may add verisimilitude to this otherwise unfounded belief.

Optional: do both. Comment on the effect of having more than two students in your system.

Your report should be in the form of a well commented program, describing any design decisions you have made.[2]

Here is a solution in which there is no enforcement of the requirements:

```
object ChaoticTutor extends SleepingTutor
{ def TutorWait         = {}
  def Arrive            = {}
  def ReceiveTute       = {}
  def EndTeach          = {}
}
```

Here's part of a transcript of a run.[3]

```
553 $ xso ChaoticTutor
0·01657 Tutor waiting
0·01915 Tutorial starts
0·01946 Tutorial ends
1·02007 Tutor waiting
1·02070 Tutorial starts
1·02108 Tutorial ends
2·01720 Ada   arrives
2·01720 Bob   arrives
2·01782 Ada   ready
2·01792 Bob   ready
2·01824 Ada   leaves
2·01830 Bob   leaves
3·01959 Bob   arrives
3·01959 Ada   arrives
3·02048 Bob   ready
3·02057 Ada   ready
3·02085 Bob   leaves
3·02101 Ada   leaves
3·02146 Tutor waiting
3·02188 Tutorial starts
```

Bernard Sufrin
1st October, 2017(4.1553)

[2]You may well find, as I do, that the sequence of log messages for a correct solution is not output in order of their timestamps: this is not incorrect, but you should explain why it happens.

[3]Quite unlike *my* college tutorials, as my students will aver.

```
trait SleepingTutor
{ def TutorWait:        Unit
  def Arrive:           Unit
  def ReceiveTute:      Unit
  def EndTeach:         Unit

  val start = nanoTime
  def now   = (nanoTime-start)/1E9 // in seconds
  def log(me: String, item: String) =
      println(f"$now%6.5f␣$me%-5s␣$item%s")

  val random = new scala.util.Random

  def Student(me: String) = proc(s"Student␣$me") {
    while (true) {
      sleep(random.nextInt(3)*Sec)
      log(me, "arrives"); Arrive
      log(me, "ready");   ReceiveTute
      log(me, "leaves")
    }
  }

  def Tutor = proc("Tutor") {
    while (true) {
      log("Tutor", "waiting");   TutorWait
      log("Tutorial", "starts")
      sleep(random.nextInt(3)*Sec)
      log("Tutorial", "ends");   EndTeach
      sleep(random.nextInt(3)*Sec)
    }
  }

  def main(args: Array[String]) =
  {    if (args.contains("-d")) println(debugger)
       run(Tutor||Student("Ada")||Student("Bob"))
  }
}
```

Figure 1: Sleeping Tutor Test-Rig