Exercise 1: Barrier Objects

Create a barrier object in Java, Active Oberon or the language of your choice that is able to synchronize a set of predefined threads or active objects respectively.

The barrier object should provide two methods: initialize (e.g., \texttt{init}) and synchronize (e.g., \texttt{sync}). The initialize method is to register a number of threads of a certain group. Calling the synchronize method should lead to a synchronization of the given threads, that means that the statement next to this call in a thread should not be called until all other threads have called synchronize as well.

- Java:

```java
public class Barrier {

/** Number of threads in the group */
private static int N;

/** Current number of waiting threads */
private static int count;

/** Constructor (barrier initialization */
public Barrier(int n) {
    N = n;
    count = 0;
}

/** Called to synchronize with the other blocked threads */
public void synchronized sync() {
    count++;
    if (count == N) {
        /* we reached the barrier: release the lock */
        notifyAll();
        count = 0;
    } else {
        wait();
    }
}
}
```
Active Oberon:

MODULE Barriers;

TYPE
  Barrier = OBJECT
    VAR
      n: LONGINT; (* Number of threads in the group *)
      N: LONGINT; (* Current number of waiting threads *)

    (* Called to synchronize with the other blocked threads *)
    PROCEDURE Sync*;
    VAR i: LONGINT;
    BEGIN {EXCLUSIVE}
      i := n DIV N;
      INC(n);
      AWAIT (i < n DIV N)
    END Enter;

    (* Constructor (barrier initialization) *)
    PROCEDURE & Init (nofProcs: LONGINT);
    BEGIN
      N := nofProcs; n := 0
    END Init;

END Barrier;

END Barriers.
Exercise 2: Deadlocks

Assume an electronic bank transfer system with hundreds of identical processes, each of them working as follows:

Read an input line specifying the amount of money, the from-account and the to-account.
Try to lock both accounts. Transfer the money.

Due to the huge amount of processes it might happen that some process having a lock on an account \(x\) is not able to get the lock on an account \(y\) because the account \(y\) is locked by a process waiting for \(x\). Develop a scheme that prevents deadlocks so that no lock has to be released when a second lock cannot be assigned.

There are four conditions that can generate a lock:

- mutual exclusion: two or more processes access a common resource one at a time.
- hold and wait: processes can get access to resources and wait for some other resources.
- no resource preemption: a resource cannot be removed from a process.
- circular wait: there is a cyclic chain of locks.

Since we cannot introduce resource preemption we try to make cyclic locks impossible by defining a locking order. One option is to define a global ordering of the account locks (which results in a graph without cycles).
Exercise 3: Deadlocks

Assume a gorge with a rope spanning from one side to the other so that baboons can pass. Several baboons can cross the gorge at the same time as long as they move in the same direction. If, at some point, there are baboons moving both to the west and some baboons moving to the east, a deadlock occurs (the baboons get stuck). Every baboon willing to cross the gorge has to be sure that no other baboon is crossing the gorge in the opposite direction at that point in time.

1. Describe a solution which prevents deadlocks, using the concept of semaphores (in this first part don’t consider starvation)

2. Solve the problem avoiding starvation
Solution using semaphores (in pseudocode):

Semaphore rope = 1, mutex_east = 1, mutex_west = 1, turnstyle = 1;

int east = 0; west = 0;

BaboonFromEast:

    turnstyle.P();
    mutex_east.P();
    east++;
    if (east == 1)
        rope.P();
    mutex_east.V();
    turnstyle.V();

    cross();

    mutex_east.P();
    east--;
    if (east == 0)
        rope.V();
    mutex_east.V();

BaboonFromWest:

    turnstyle.P();
    mutex_west.P();
    west++;
    if (west == 1)
        rope.P();
    mutex_west.V();
    turnstyle.V();

    cross();

    mutex_west.P();
    west--;
    if (west == 0)
        rope.V();
    mutex_west.V();

For this example to avoid confusion with the wait Java keyword we use the original Dijkstra notation:

V Verhogen: signals the resource is free. The value of a semaphore is the number of units of
the resource which are free).

P Proberen: waits for a resource (busy-wait).
A ticketing system could also solve the problem (something similar to the tickets you get at the post office ...)