MODULE *mutex*



This is a demonstration of how one could model an implementation
of *mutex_lock*() and *mutex_unlock*() in *PlusCal*.
An alternative is to just be careful in applications that
use mutexes to create a critical section. In that case,
you don't need this implementation. Just make sure not to
place any labels in the critical section of the
application that you are modelling.

Finally, you can easily add macros for *mutex_lock*() and *mutex_unlock*()
in other programs. After the "–algorithm" statement, in the
"variables" statement for global variables, include "$lock = 0$".
Then, after the "variables" statement, include:
macro *mutex_lock*(){*await* $lock = 0$; $lock := 1$;}
macro *mutex_unlock*(){$lock := 0$;}

EXTENDS *Naturals*, *Sequences*, *TLC* Sequences required for "procedure" stmt
CONSTANT $N$ $N$ is number of iterations. Assign to it in model overview.

```
--algorithm mutex{
  variables total = 0, lock = 0,
            iterations = [i ∈ {"thr1", "thr2"} ↦ N];

  procedure mutex_lock( )
  {
    l0: while ( TRUE ) {
    l1:   if ( lock = 0 ) {   Test if someone released the lock, or if lock = 0 before
            lock := 1;        We atomically test and acquire the lock, and return
    l_end:  return;
          }
        }
   }

  procedure mutex_unlock( )
  {
    u0: assert lock = 1;
        lock := 0;   Release the lock, atomically
    u_end:    return;
   }

  process ( thread ∈ {"thr1", "thr2"} )
    variable register;
  { start: while ( iterations[self] > 0 ) {
      p1: call mutex_lock();
```

```
      p2: register := total    ;
      p3: register := register + 1 ;
          total := register ;
      p4: call mutex_unlock() ;
      p5: iterations[self] := iterations[self] − 1 ;
      } ;
    assert iterations[self] = 0 ;

    if ( iterations["thr1"] = 0 ∧ iterations["thr2"] = 0 ) {
      assert total = 2 ∗ N ;
     }
  }  end process block

}   \ ∗ end algorithm
```