```
code Synch

  -- OS Class: Project 2
  --
  -- Ted Timmons, tedt@pdx.edu / ted@perljam.net

-----------------------------  Semaphore  ---------------------------------

  behavior Semaphore
    -- This class provides the following methods:
    --    Up()  ...also known as "V" or "Signal"...
    --         Increment the semaphore count.  Wake up a thread if
    --         there are any waiting.  This operation always executes
    --         quickly and will not suspend the thread.
    --    Down()   ...also known as "P" or "Wait"...
    --         Decrement the semaphore count.  If the count would go
    --         negative, wait for some other thread to do an Up()
    --         first.  Conceptually, the count will never go negative.
    --    Init(initialCount)
    --         Each semaphore must be initialized.  Normally, you should
    --         invoke this method, providing an 'initialCount' of zero.
    --         If the semaphore is initialized with 0, then a Down()
    --         operation before any Up() will wait for the first
    --         Up().  If initialized with i, then it is as if i Up()
    --         operations have been performed already.
    --
    -- NOTE: The user should never look at a semaphore's count since the value
    -- retrieved may be out-of-date, due to other threads performing Up() or
    -- Down() operations since the retrieval of the count.

      ----------  Semaphore . Init  ----------

      method Init (initialCount: int)
          if initialCount < 0
            FatalError ("Semaphore created with initialCount < 0")
          endIf
          count = initialCount
          waitingThreads = new List [Thread]
        endMethod

      ----------  Semaphore . Up  ----------

      method Up ()
          var
            oldIntStat: int
            t: ptr to Thread
          oldIntStat = SetInterruptsTo (DISABLED)
          if count == 0x7fffffff
            FatalError ("Semaphore count overflowed during 'Up' operation")
          endIf
          count = count + 1
          if count <= 0
            t = waitingThreads.Remove ()
            t.status = READY
            readyList.AddToEnd (t)
          endIf
          oldIntStat = SetInterruptsTo (oldIntStat)
        endMethod

      ----------  Semaphore . Down  ----------

      method Down ()
          var
```

```
          oldIntStat: int
        oldIntStat = SetInterruptsTo (DISABLED)
        if count == 0x80000000
          FatalError ("Semaphore count underflowed during 'Down' operation")
        endIf
        count = count - 1
        if count < 0
          waitingThreads.AddToEnd (currentThread)
          currentThread.Sleep ()
        endIf
        oldIntStat = SetInterruptsTo (oldIntStat)
      endMethod

  endBehavior

-----------------------------  Mutex  --------------------------------

  behavior Mutex
    -- This class provides the following methods:
    --    Lock()
    --         Acquire the mutex if free, otherwise wait until the mutex is
    --         free and then get it.
    --    Unlock()
    --         Release the mutex.  If other threads are waiting, then
    --         wake up the oldest one and give it the lock.
    --    Init()
    --         Each mutex must be initialized.
    --    IsHeldByCurrentThread()
    --         Return TRUE iff the current (invoking) thread holds a lock
    --         on the mutex.

      ----------  Mutex . Init  ----------


      -- Takes initial state of the mutex (LOCKED, UNLOCKED).
      --    Init()
      --         Each mutex must be initialized.
      method Init ()

          if waitCount < 0
            FatalError ("Mutex created with waitCount < 0")
          endIf

          -- set up our variables:
          -- heldBy: the Thread that is holding the lock
          heldBy = null
          -- state: the lock itself
          state = UNLOCKED
          -- waitingThreads: FIFO queue of threads that are asleep, waiting for lock
          waitingThreads = new List [Thread]
          -- waitCount: the number of items on the list/queue.
          waitCount = 0

        endMethod

      ----------  Mutex . Lock  ----------

      --    Lock()
      --         Acquire the mutex if free, otherwise wait until the mutex is
      --         free and then get it.
      method Lock ()
          var oldIntStat: int
          -- var oldState: int
```

```
    -- critical section, disable interrupts.
    oldIntStat = SetInterruptsTo (DISABLED)

    -- if an "if" is used here instead of "while", that will potentially cause
    -- the code to wake up while the lock is held elsewhere. The "while" makes
    -- sure that we loop until the lock is actually available, not simply until
    -- we wake up.
    while state == LOCKED
      -- print (" sleeping on lock, we don't have it (")
      -- print (currentThread.name)
      -- print (").\n")
      waitingThreads.AddToEnd (currentThread)
      waitCount = waitCount + 1
      currentThread.Sleep ()
    endWhile

    -- We are guaranteed to have state=UNLOCKED at this point.
    -- mutex is free, so we'll acquire it.
    -- print (" getting the lock for ")
    -- print (currentThread.name)
    -- print ("\n")

    -- sanity-check/assert that we aren't locking an already-held lock
    if heldBy != null
      -- print ("holding a held lock. state: ")
      -- if (state == LOCKED)
      --   print ("locked")
      -- endIf
      -- print ("\n")
      FatalError ("about to hold a held lock, eep!")
    endIf

    -- actually lock the state and indicate who it is held by
    state = LOCKED
    heldBy = currentThread

    -- success!
    oldIntStat = SetInterruptsTo (oldIntStat)
  endMethod

----------  Mutex . Unlock  ----------

method Unlock ()
    var
      oldIntStat: int
      nextThread: ptr to Thread

    oldIntStat = SetInterruptsTo (DISABLED)

    if state == UNLOCKED
      FatalError ("asked for lock to be released, but nothing was locked!")
    endIf

    -- Make sure we are releasing a lock that we hold, not someone else.
    if heldBy != currentThread
      -- print ("heldby: ")
      -- print (heldBy.name)
      -- print (" .. currentThread: ")
      -- print (currentThread.name)
      -- print ("\n")
      FatalError ("thread was not locked by currentThread.")
    endIf
```

```
          -- print (" unlocking for ")
          -- print (currentThread.name)
          -- print ("\n")
          -- Actually release the lock, now that we've verified everything.
          state = UNLOCKED
          heldBy = null

          -- pull our next thread from the (lock) waiting list.
          -- Don't start it, but mark it ready.
          if waitCount > 0
            waitCount = waitCount - 1
            nextThread = waitingThreads.Remove()
            nextThread.status = READY
            readyList.AddToEnd (nextThread)
          endIf

          oldIntStat = SetInterruptsTo (oldIntStat)

        endMethod

      ----------  Mutex . IsHeldByCurrentThread  ----------

      method IsHeldByCurrentThread () returns bool

          -- is it locked? Are we holding it? Great!
          if (state == LOCKED && heldBy == currentThread)
            return true
          endIf

          -- Not held, or at least not held by us.
          return false
        endMethod

  endBehavior

-----------------------------  Condition  --------------------------------

  behavior Condition
    -- This class is used to implement monitors.  Each monitor will have a
    -- mutex lock and one or more condition variables.  The lock ensures that
    -- only one process at a time may execute code in the monitor.  Within the
    -- monitor code, a thread can execute Wait() and Signal() operations
    -- on the condition variables to make sure certain condions are met.
    --
    -- The condition variables here implement "Mesa-style" semantics, which
    -- means that in the time between a Signal() operation and the awakening
    -- and execution of the corrsponding waiting thread, other threads may
    -- have snuck in and run.  The waiting thread should always re-check the
    -- data to ensure that the condition which was signalled is still true.
    --
    -- This class provides the following methods:
    --    Wait(mutex)
    --         This method assumes the mutex has alreasy been locked.
    --         It unlocks it, and goes to sleep waiting for a signal on
    --         this condition.  When the signal is received, this method
    --         re-awakens, re-locks the mutex, and returns.
    --    Signal(mutex)
    --         If there are any threads waiting on this condition, this
    --         method will wake up the oldest and schedule it to run.
    --         However, since this thread holds the mutex and never unlocks
    --         it, the newly awakened thread will be forced to wait before
    --         it can re-acquire the mutex and resume execution.
```

```
--    Broadcast(mutex)
--         This method is like Signal() except that it wakes up all
--         threads waiting on this condition, not just the next one.
--    Init()
--         Each condition must be initialized.

  ----------  Condition . Init  ----------

  method Init ()
      waitingThreads = new List [Thread]
    endMethod

  ----------  Condition . Wait  ----------

  method Wait (mutex: ptr to Mutex)
      var
        oldIntStat: int
      if ! mutex.IsHeldByCurrentThread ()
        FatalError ("Attempt to wait on condition when mutex is not held")
      endIf
      oldIntStat = SetInterruptsTo (DISABLED)
      mutex.Unlock ()
      waitingThreads.AddToEnd (currentThread)
      currentThread.Sleep ()
      mutex.Lock ()
      oldIntStat = SetInterruptsTo (oldIntStat)
    endMethod

  ----------  Condition . Signal  ----------

  method Signal (mutex: ptr to Mutex)
      var
        oldIntStat: int
        t: ptr to Thread
      if ! mutex.IsHeldByCurrentThread ()
        FatalError ("Attempt to signal a condition when mutex is not held")
      endIf
      oldIntStat = SetInterruptsTo (DISABLED)
      t = waitingThreads.Remove ()
      if t
        t.status = READY
        readyList.AddToEnd (t)
      endIf
      oldIntStat = SetInterruptsTo (oldIntStat)
    endMethod

  ----------  Condition . Broadcast  ----------

  method Broadcast (mutex: ptr to Mutex)
      var
        oldIntStat: int
        t: ptr to Thread
      if ! mutex.IsHeldByCurrentThread ()
        FatalError ("Attempt to broadcast a condition when lock is not held")
      endIf
      oldIntStat = SetInterruptsTo (DISABLED)
      while true
        t = waitingThreads.Remove ()
        if t == null
          break
        endIf
        t.status = READY
        readyList.AddToEnd (t)
```

```
          endWhile
          oldIntStat = SetInterruptsTo (oldIntStat)
        endMethod

  endBehavior

endCode
```

```
header Synch

  uses Thread

  class Semaphore
    superclass Object
    fields
      count: int
      waitingThreads: List [Thread]
    methods
      Init (initialCount: int)
      Down ()
      Up ()
  endClass

  enum UNLOCKED, LOCKED   -- Mutex status
  class Mutex
    superclass Object
    fields
      state: int
      heldBy: ptr to Thread
      waitingThreads: List [Thread]
      waitCount: int
    methods
      Init ()
      Lock ()
      Unlock ()
      IsHeldByCurrentThread () returns bool
  endClass

  class Condition
    superclass Object
    fields
      waitingThreads: List [Thread]
    methods
      Init ()
      Wait (mutex: ptr to Mutex)
      Signal (mutex: ptr to Mutex)
      Broadcast (mutex: ptr to Mutex)
  endClass

endHeader
```

```
Script started on Sun 18 Oct 2009 07:16:54 PM PDT
$ make && blitz -g os
kpl Main -unsafe
asm Main.s
lddd System.o List.o Thread.o Switch.o Synch.o Main.o Runtime.o -o os
Beginning execution...
====================  KPL PROGRAM STARTING  ====================
Example Thread-based Programs...
Initializing Thread Scheduler...

-- You should see 70 lines, each consecutively numbered. --

LockTester-A = 1
LockTester-A = 2
LockTester-A = 3
LockTester-A = 4
LockTester-A = 5
LockTester-A = 6
LockTester-F = 7
LockTester-F = 8
LockTester-F = 9
LockTester-F = 10
LockTester-F = 11
LockTester-A = 12
LockTester-A = 13
LockTester-A = 14
LockTester-A = 15
LockTester-C = 16
LockTester-C = 17
LockTester-C = 18
LockTester-C = 19
LockTester-D = 20
LockTester-D = 21
LockTester-E = 22
LockTester-F = 23
LockTester-F = 24
LockTester-F = 25
LockTester-F = 26
LockTester-F = 27
LockTester-G = 28
LockTester-G = 29
LockTester-G = 30
LockTester-G = 31
LockTester-G = 32
LockTester-G = 33
LockTester-G = 34
LockTester-G = 35
LockTester-E = 36
LockTester-C = 37
LockTester-C = 38
LockTester-C = 39
LockTester-C = 40
LockTester-C = 41
LockTester-C = 42
LockTester-E = 43
LockTester-B = 44
LockTester-E = 45
LockTester-G = 46
LockTester-G = 47
LockTester-E = 48
LockTester-D = 49
LockTester-D = 50
LockTester-E = 51
```

```
LockTester-B = 52
LockTester-E = 53
LockTester-D = 54
LockTester-D = 55
LockTester-E = 56
LockTester-B = 57
LockTester-E = 58
LockTester-D = 59
LockTester-D = 60
LockTester-E = 61
LockTester-B = 62
LockTester-D = 63
LockTester-B = 64
LockTester-D = 65
LockTester-B = 66
LockTester-B = 67
LockTester-B = 68
LockTester-B = 69
LockTester-B = 70

*****  A 'wait' instruction was executed and no more interrupts are scheduled... halting +
emulation!  *****

Done!  The next instruction to execute will be:
000EC8: 09000000       ret
Number of Disk Reads    = 0
Number of Disk Writes   = 0
Instructions Executed   = 353623
Time Spent Sleeping     = 0
    Total Elapsed Time  = 353623
$ exit

Script done on Sun 18 Oct 2009 07:17:07 PM PDT
```