```
code Main

  -- OS Class: Project 2
  --
  -- <PUT YOUR NAME HERE>
  --
  -- This package contains the following:
  --     SimpleThreadExample
  --     MoreThreadExamples
  --     ProducerConsumer
  --     TestMutex
  --     Dining Philospohers



-----------------------------  SynchTest  ---------------------------------

  function main ()
      print ("Example Thread-based Programs...\n")

      InitializeScheduler ()

      -----  Uncomment any one of the following to perform the desired test  -----

      --SimpleThreadExample ()
      --MoreThreadExamples ()
      --TestMutex ()
      --ProducerConsumer ()
      DiningPhilosophers ()

      ThreadFinish ()

    endFunction



-----------------------------  SimpleThreadExample  ---------------------------------

  var aThread: Thread     -- Don't put Thread objects on the stack, since the
                          -- routine that contains them may return!

  function SimpleThreadExample ()
    -- This code illustrates the basics of thread usage.
    --
    -- This code uses 2 threads.  Each thread loops a few times.
    -- Each loop iteration prints a message and executes a "Yield".
    -- This code illustrates the following operations:
    --     Thread creation
    --     Fork
    --     Yield
    --     Thread termination
    -- This code creates only one new thread; the currrent ("main") thread, which
    -- already exists, is the other thread.  Both the main thread and the newly
    -- created thread will call function "SimpleThreadFunction" to perform the looping.
    --
    -- Notice that timer interrupts will also cause "Yields" to be inserted at
    -- unpredictable places.  Thus, the threads will not simply alternate.
    --
    -- Things to experiment with:
    --    In TimerInterruptHandler (in Thread.c), uncomment "print ('_')".
    --    In TimerInterruptHandler (in Thread.c), comment out the call to
    --            Yield, which will suspend timeslicing.
    --    Edit .blitzrc (see "sim" command) and change TIME_SLICE value.
```

```
    --    In this function, comment out the call to "Yield".
    --

      print ("Simple Thread Example...\n")
      aThread = new Thread
      aThread.Init ("Second-Thread")   -- Initialize, giving thread a name
      aThread.Fork (SimpleThreadFunction, 4)  -- Pass "4" as argument to the thread
      SimpleThreadFunction (7)                -- The main thread will loop 7 times
    endFunction

  function SimpleThreadFunction (cnt: int)
    -- This function will loop "cnt" times.  Each iteration will print a
    -- message and execute a "Yield", which will give the other thread a
    -- chance to run.
      var i: int
      for i = 1 to cnt
        print (currentThread.name)
        nl ()
        currentThread.Yield ()
      endFor
      ThreadFinish ()
    endFunction



-----------------------------  MoreThreadExamples  ---------------------------------

  var th1, th2, th3, th4, th5, th6: Thread

  function MoreThreadExamples ()
      var j: int
          oldStatus: int

      print ("Thread Example...\n")

      -- Create some thread objects (not on the heap).

      th1 = new Thread
      th2 = new Thread
      th3 = new Thread
      th4 = new Thread
      th5 = new Thread
      th6 = new Thread

      -- Initialize them.
      th1.Init ("thread-a")
      th2.Init ("thread-b")
      th3.Init ("thread-c")
      th4.Init ("thread-d")
      th5.Init ("thread-e")
      th6.Init ("thread-f")

      -- Start all threads running.  Each thread will execute the "foo"
      -- function, but each will be passed a different argument.
      th1.Fork (foo, 1)
      th2.Fork (foo, 2)
      th3.Fork (foo, 3)
      th4.Fork (foo, 4)
      th5.Fork (foo, 5)
      th6.Fork (foo, 6)

      -- Print this thread's name.  Note that we temporarily disable
      -- interrupts so that all printing will happen together.  Without
```

```
      -- this, the other threads might print in the middle, causing a mess.
      oldStatus = SetInterruptsTo (DISABLED)
        print ("\nThe currently running thread is ")
        print (currentThread.name)
        print ("\n")
        PrintReadyList ()
      oldStatus = SetInterruptsTo (oldStatus)

      for j = 1 to 10
        currentThread.Yield ()
        print ("\n..Main..\n")
      endFor

      -- Print the readyList at this point...
      print ("\nReadyList\n")
      PrintReadyList ()
      currentThread.Print()

/*
      -- Put this thread to sleep...
      oldStatus = SetInterruptsTo (DISABLED)
      print ("About to Sleep main thread...\n")
      currentThread.Sleep ()
      FatalError ("BACK FROM SLEEP !?!?!")
      -- Execution will never reach this point, since the current thread
      -- was not placed on any list of waiting threads.  Nothing in this
      -- code could ever move this thread back to the ready list.
*/

      PrintReadyList ()
      ThreadFinish ()
      PrintReadyList ()

    endFunction


  function foo (i: int)
      var j: int

      for j = 1 to 30
        printInt (i)

        if j == 20
          -- Next is an example of aborting all threads and shutting down...
          --   FatalError ("Whoops...(SAMPLE ERROR MESSAGE)")

          -- Next is an example of just quietly shutting down...
          --   RuntimeExit ()

          -- Next is an example of what happens if execution errors occur...
          --   i = j / 0         -- Generate an error
        endIf

        -- Call Yield so other threads can run.  This is not necessary,
        -- but it will cause more interleaving of the various threads,
        -- making this program’s output more interesting.
        currentThread.Yield ()
      endFor
    endFunction


----------------------------  Test Mutex  --------------------------------
```

```
  -- This code illustrates the ideas behind "critical sections" and "mutual
  -- exclusion".  This code creates several threads.  Each thread accesses
  -- some shared data (an integer) in a critical section.  A single lock
  -- is used to control access to the shared variable.  Each thread locks
  -- the mutex, computes a while, increments the integer, prints the new value,
  -- updates the shared copy, and unlocks the mutex.  Then it does some
  -- non-critical computation and repeats.

  var
    myLock: Mutex = new Mutex      -- Could also use "Mutex2" instead
    sharedInt: int = 0
    thArr: array [7] of Thread = new array of Thread {7 of new Thread }

  function TestMutex ()
      myLock.Init ()

      print ("\n-- You should see 70 lines, each consecutively numbered. --\n\n")

      thArr[0].Init ("LockTester-A")
      thArr[0].Fork (LockTester, 100)

      thArr[1].Init ("LockTester-B")
      thArr[1].Fork (LockTester, 200)

      thArr[2].Init ("LockTester-C")
      thArr[2].Fork (LockTester, 1)

      thArr[3].Init ("LockTester-D")
      thArr[3].Fork (LockTester, 50)

      thArr[4].Init ("LockTester-E")
      thArr[4].Fork (LockTester, 300)

      thArr[5].Init ("LockTester-F")
      thArr[5].Fork (LockTester, 1)

      thArr[6].Init ("LockTester-G")
      thArr[6].Fork (LockTester, 1)

      ThreadFinish ()
    endFunction

  function LockTester (waitTime: int)
    -- This function will do the following actions, several times in a loop:
    --     Lock the mutex
    --     Get the current value of the "sharedInt" variable
    --     Compute a new value by adding 1
    --     Wait a while (determined by parameter "waitTime") to simulate
    --        actions done within the critical section
    --     Print the thread's name and the new value
    --     Update the "sharedInt" variable
    --     Unlock the mutex
    --     Wait a while (determined by parameter "waitTime") to simulate
    --        actions done outside of the critical section
      var
        i, j, k: int
      for i = 1 to 10

        -- Enter
--print ("locktester on ")
--print (currentThread.name)
--print (", locking.\n")
```

```
        myLock.Lock()
--print (" have lock for ")
--print (currentThread.name)
--print ("\n")

        -- Critical Section
        j = sharedInt + 1                   -- read shared data
        for k = 1 to waitTime               -- do some computation
        endFor                              --
        printIntVar (currentThread.name, j)  -- print new data value
        sharedInt = j                       -- update shared data

        -- Leave
--print ("locktester on ")
--print (currentThread.name)
--print (", unlocking.\n")
        myLock.Unlock()

        -- Perform non-critical work
        for k = 1 to waitTime
        endFor

      endFor
    endFunction



-----------------------------  ProducerConsumer  ---------------------------------

  -- This code implements the consumer-producer task.  There are several
  -- "producers", several "consumers", and a single shared buffer.
  --
  -- The producers are named "A", "B", "C", etc.  Each producer is a thread which
  -- will loop 5 times.  For each iteration, the producer thread will add its
  -- character to a shared buffer.  For example, "Producer-B" will add 5 "B"s to
  -- the shared buffer.  Since the 5 producer threads will run concurrently, the
  -- characters will be added in an unpredictable order.  Regardless of the order,
  -- however, there will be five "A"s, five "B"s, five "C"s, etc.
  --
  -- There are several consumers.  Each consumer is a thread which executes an
  -- inifinite loop.  During each iteration of its loop, a consumer will remove
  -- whatever character is next in the buffer and will print it.
  --
  -- The shared buffer is a FIFO queue of characters.  The producers put characters
  -- in one end and the consumers take characters out the other end.  Think of a
  -- section of steel pipe.  The capacity of the buffer is limited to BUFFER_SIZE
  -- characters.
  --
  -- This code illustrates the mechanisms required to synchronize the producers,
  -- consumers, and the shared buffer.  Consumers must wait if the buffer is empty.
  -- Producers must wait if the buffer is full.  Furthermore, the buffer is a shared
  -- data structure.  (The buffer is implemented as an array with pointers to the
  -- next position to add or remove characters.)  No two threads are allowed to
  -- access these pointers simultaneously, or else errors may result.
  --
  -- To document what is happening, each producer will print a line when it adds
  -- a character to the buffer.  The line printed will include the buffer contents
  -- along with the name of the poducer.  Also, each time a consumer removes a
  -- character from the buffer, it will print a line, showing the buffer contents
  -- after the removal, along with the name of the consumer thread.  Each line of
  -- output is formated so that you can see the buffer growing and shrinking.  By
  -- reading the output vertically, you can also see what each thread does.
  --
```

```
  const
    BUFFER_SIZE = 5

  var
    buffer: array [BUFFER_SIZE] of char = new array of char {BUFFER_SIZE of '?'}
    bufferSize: int = 0
    bufferNextIn: int = 0
    bufferNextOut: int = 0
    thArray: array [8] of Thread = new array of Thread { 8 of new Thread }
    semEmpty: Semaphore = new Semaphore
    semFull: Semaphore = new Semaphore


  function ProducerConsumer ()

      semEmpty.Init(BUFFER_SIZE)
      semFull.Init(0)

      print ("     ")

      thArray[0].Init ("Consumer-1                               |      ")
      thArray[0].Fork (Consumer, 1)

      thArray[1].Init ("Consumer-2                               |          ")
      thArray[1].Fork (Consumer, 2)

      thArray[2].Init ("Consumer-3                               |              ")
      thArray[2].Fork (Consumer, 3)

      thArray[3].Init ("Producer-A         ")
      thArray[3].Fork (Producer, 1)

      thArray[4].Init ("Producer-B             ")
      thArray[4].Fork (Producer, 2)

      thArray[5].Init ("Producer-C                 ")
      thArray[5].Fork (Producer, 3)

      thArray[6].Init ("Producer-D                     ")
      thArray[6].Fork (Producer, 4)

      thArray[7].Init ("Producer-E                         ")
      thArray[7].Fork (Producer, 5)

      ThreadFinish ()
    endFunction

  function Producer (myId: int)
      var
        i: int
        c: char = intToChar ('A' + myId - 1)
      for i = 1 to 5
        -- Perform synchroniztion...
        semEmpty.Down()

        -- Add c to the buffer
        buffer [bufferNextIn] = c
        bufferNextIn = (bufferNextIn + 1) % BUFFER_SIZE
        bufferSize = bufferSize + 1

        -- Print a line showing the state
        PrintBuffer (c)
```

```
        -- Perform synchronization...
        semFull.Up()

      endFor
    endFunction

  function Consumer (myId: int)
      var
        c: char
      while true
        -- Perform synchroniztion...
        semFull.Down()

        -- Remove next character from the buffer
        c = buffer [bufferNextOut]
        bufferNextOut = (bufferNextOut + 1) % BUFFER_SIZE
        bufferSize = bufferSize - 1

        -- Print a line showing the state
        PrintBuffer (c)

        -- Perform synchronization...
        semEmpty.Up()

      endWhile
    endFunction

  function PrintBuffer (c: char)
    --
    -- This method prints the buffer and what we are doing to it.  Each
    -- line should have
    --        <buffer>  <threadname> <character involved>
    -- We want to print the buffer as it was *before* the operation;
    -- however, this method is called *after* the buffer has been modified.
    -- To achieve the right order, we print the operation first, skip to
    -- the next line, and then print the buffer.  Assuming we start by
    -- printing an empty buffer first, and we are willing to end the output
    -- in the middle of a line, this prints things in the desired order.
    --
      var
        i, j: int
      -- Print the thread name, which tells what we are doing.
      print ("   ")
      print (currentThread.name)  -- Will include right number of spaces after name
      printChar (c)
      nl ()
      -- Print the contents of the buffer.
      j = bufferNextOut
      for i = 1 to bufferSize
        printChar (buffer[j])
        j = (j + 1) % BUFFER_SIZE
      endFor
      -- Pad out with blanks to make things line up.
      for i = 1 to BUFFER_SIZE-bufferSize
        printChar (' ')
      endFor
    endFunction


-----------------------------  Dining Philosophers  ---------------------------------

  -- This code is an implementation of the Dining Philosophers problem.  Each
```

```
-- philosopher is simulated with a thread.  Each philosopher thinks for a while
-- and then wants to eat.  Before eating, he must pick up both his forks.
-- After eating, he puts down his forks.  Each fork is shared between
-- two philosophers and there are 5 philosophers and 5 forks arranged in a
-- circle.
--
-- Since the forks are shared, access to them is controlled by a monitor
-- called "ForkMonitor".  The monitor is an object with two "entry" methods:
--     PickupForks (phil)
--     PutDownForks (phil)
-- The philsophers are numbered 0 to 4 and each of these methods is passed an integer
-- indicating which philospher wants to pickup (or put down) the forks.
-- The call to "PickUpForks" will wait until both of his forks are
-- available.  The call to "PutDownForks" will never wait and may also
-- wake up threads (i.e., philosophers) who are waiting.
--
-- Each philospher is in exactly one state: HUNGRY, EATING, or THINKING.  Each time
-- a philosopher's state changes, a line of output is printed.  The output is organized
-- so that each philosopher has column of output with the following code letters:
--           E    --  eating
--           .    --  thinking
--         blank  --  hungry (i.e., waiting for forks)
-- By reading down a column, you can see the history of a philosopher.
--
-- The forks are not modeled explicitly.  A fork is only picked up
-- by a philospher if he can pick up both forks at the same time and begin
-- eating.  To know whether a fork is available, it is sufficient to simply
-- look at the status's of the two adjacent philosophers.  (Another way to state
-- the problem is to forget about the forks altogether and stipulate that a
-- philosopher may only eat when his two neighbors are not eating.)

enum HUNGRY, EATING, THINKING
var
  mon: ForkMonitor
  philospher: array [5] of Thread = new array of Thread {5 of new Thread }

function DiningPhilosophers ()

    print ("Plato\n")
    print ("    Sartre\n")
    print ("        Kant\n")
    print ("            Nietzsche\n")
    print ("                Aristotle\n")

    mon = new ForkMonitor
    mon.Init ()
    mon.PrintAllStatus ()

    philospher[0].Init ("Plato")
    philospher[0].Fork (PhilosphizeAndEat, 0)

    philospher[1].Init ("Sartre")
    philospher[1].Fork (PhilosphizeAndEat, 1)

    philospher[2].Init ("Kant")
    philospher[2].Fork (PhilosphizeAndEat, 2)

    philospher[3].Init ("Nietzsche")
    philospher[3].Fork (PhilosphizeAndEat, 3)

    philospher[4].Init ("Aristotle")
    philospher[4].Fork (PhilosphizeAndEat, 4)
```

```
     endFunction

  function PhilosphizeAndEat (p: int)
    -- The parameter "p" identifies which philosopher this is.
    -- In a loop, he will think, acquire his forks, eat, and
    -- put down his forks.
      var
        i: int
      for i = 1 to 7
        -- Now he is thinking
        mon. PickupForks (p)
        -- Now he is eating
        mon. PutDownForks (p)
      endFor
    endFunction

  class ForkMonitor
    superclass Object
    fields
      status: array [5] of int             -- For each philosopher: HUNGRY, EATING, or +
THINKING
      sem: array [5] of Semaphore
    methods
      Init ()
      PickupForks (p: int)
      PutDownForks (p: int)
      PrintAllStatus ()
  endClass

  behavior ForkMonitor

    method Init ()
        var
          i: int
        for i = 0 to 4
          -- Initialize so that all philosophers are THINKING.
          philosopher[i].status = THINKING
          philosopher[p].sem.Init(1)
        endFor
      endMethod

    method PickupForks (p: int)
        -- This method is called when philosopher ’p’ is wants to eat.
        var
          prev: int
          next: int

        prev = (p-1) % 5
        next = (p+1) % 5
        if (philosopher[prev].Status == EATING)
  print ("-1 is eating, so we’re hungry, down/wait.\n")
          philosopher[p].Status == HUNGRY
          philosopher[prev].sem.Down()
        endIf

        if (philosopher[next].Status == EATING)
  print ("+1 is eating, so we’re hungry, down/wait.\n")
          philosopher[p].Status == HUNGRY
          philosopher[next].sem.Down()
        endIf

        -- we should be able to get both forks now.
        philosopher[p].sem.Up()
```

```
        philosopher[p].Status = EATING
print ("yum, we (")
print (p)
print (") are eating!\n")


      endMethod

    method PutDownForks (p: int)
      -- This method is called when the philosopher 'p' is done eating.
      philosopher[p].sem.Up()
      philosopher[p].status = THINKING
      endMethod

    method PrintAllStatus ()
      -- Print a single line showing the status of all philosophers.
      --      '.' means thinking
      --      ' ' means hungry
      --      'E' means eating
      -- Note that this method is internal to the monitor.  Thus, when
      -- it is called, the monitor lock will already have been acquired
      -- by the thread.  Therefore, this method can never be re-entered,
      -- since only one thread at a time may execute within the monitor.
      -- Consequently, printing is safe.  This method calls the "print"
      -- routine several times to print a single line, but these will all
      -- happen without interuption.
        var
          p: int
        for p = 0 to 4
          switch status [p]
            case HUNGRY:
              print ("    ")
              break
            case EATING:
              print ("E   ")
              break
            case THINKING:
              print (".   ")
              break
          endSwitch
        endFor
        nl ()
      endMethod

  endBehavior

endCode
```

```
Script started on Sun 18 Oct 2009 07:52:07 PM PDT
$ make && blitz -g os
make: Nothing to be done for ‘all’.
Beginning execution...
====================  KPL PROGRAM STARTING  ====================
Example Thread-based Programs...
Initializing Thread Scheduler...
        Producer-A        A
A       Producer-A        A
AA      Producer-A        A
AAA     Producer-A        A
AAAA    Producer-A        A
AAAAA   Consumer-1                              |      A
AAAA    Consumer-2                              |          A
AAA     Consumer-2                              |          A
AA      Consumer-2                              |          A
A       Consumer-3                              |              A
        Producer-C                C
C       Producer-C                C
CC      Producer-C                C
CCC     Producer-D                    D
CCCD    Producer-E                        E
CCCDE   Consumer-3                              |              C
CCDE    Consumer-3                              |              C
CDE     Consumer-3                              |              C
DE      Consumer-3                              |              D
E       Consumer-2                              |          E
        Producer-C                C
C       Producer-C                C   Producer-D                     D
CCD     Producer-E                        E
CCDE    Producer-B            B
CCDEB   Consumer-1                              |      C
CDEB    Consumer-1                              |      C
DEB     Consumer-1                              |      D
EB      Consumer-2                              |          E
B       Producer-D                    D
BD      Producer-E                        E
BDE     Producer-B            B
BDEB
BDEB    Producer-D                    D
BDDBE   Consumer-2                              |          B
DDBE    Consumer-2                              |          D
DBE     Consumer-3                              |              D
BE      Consumer-3                              |              B
E       Consumer-1                              |      E
        Producer-B            B
B       Producer-B            B
BB      Producer-B            B
BBB     Consumer-2                              |          B
BB    Producer-D                    D
BBD     Producer-E                        E
BBDE    Consumer-3                              |              B
BDE     Consumer-1                              |      B
DE      Consumer-1                              |      D
E       Consumer-1                              |      E
          Producer-E                        E
E       Consumer-1                              |      E

*****  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   = 118433
Time Spent Sleeping     = 0
    Total Elapsed Time  = 118433
$ exit

Script done on Sun 18 Oct 2009 07:52:11 PM PDT
```