```
$ make
kpl Kernel -unsafe
asm Kernel.s
lddd Runtime.o Switch.o System.o List.o BitMap.o Kernel.o Main.o -o os
$ blitz -g os
Beginning execution...
====================  KPL PROGRAM STARTING  ====================
Initializing Thread Scheduler...
Initializing Thread Manager...
have threads.
Initializing Frame Manager...


*****  THREAD-MANAGER TEST  *****

123.4.56.78192.10....11..412.313651489.21151617..18.71920.10....114....126.3.514139.8115......+
7.10..17411..18.16122193..5.....15820..7.101413.916......1816.12.5.4.1711158.2319.10....9....2+
0718.16.14512.11317......2.6.910.154.11.8720..16..14..1.1718.3219.5..69......1112208.101413151+
.4....16....5976.1811.1712.3...19..4...20162.8510..14913..1..3..7.196.4..18152011...8..9.14.11+
617.27.125.......10.183861949.14.13...2..12..1015.181201716..3...14...138112.126.19.10.....181+
5..3.14471.813.5...17..19..11.18152203.12.14....9..16.519.7176114..10....1314.9.18.15.1625..1.+
8....19.102013.714.1718..3.12..1...9819..1516210...56.....32013..111798....1715....54196..1432+
0161112.....17..18.13215..45.1..149.6..1017..18.19.13.83..1647.12...14..2..10.11151.181396.17.+
.12..8....142.3.16..41015.18.171112...1.8...7.1914.220139.1018.17....64..18..57.1519.2.10.....+
17.14.164620.1135..9.2.10...12.17.7..1516.18.11.693...4..1920..12.7.17.81518....5.41920....126+
...13...7..3..515.12166.11.1913.20.7..3.16....11.20...19..16.11.13.20...11.20...

***** THREAD-MANAGER TEST COMPLETED SUCCESSFULLY *****



*****  PROCESS-MANAGER TEST  *****

123.4.56.78192.310...114.12.13..14..81565161729103.18.119720....11....4..1314..1215..168.1765.+
2...19.18.1.711..9.410.133..12..20..81617...14.21519181.7....9.10413..12...6112085..314...1521+
6....917.13410.12..1.76..11...3.14152..16820.917.19......1125711.18.4.13.1410...169...81915.21+
..20.126....18..1314435.168711.........1712620.21019181315..16......14.17.911243620.7......16.+
.8.1421011179.181.....7...12.16191384.15314...18......7510212.11161917...9......613187.5208210+
..3....4..9.611512.131814....16....37.411201959.6......13..161.3158741712......6..11..13201611+
02318.....6..19..15115.141382016.....1919..156...10112714.....5...199.413.31817..6...14..15..1+
95.10124119.2....6..13.20.1415.5319...71..210..6...13122016.8.14.11..9.7...12.418.6317.13..10.+
.5..11..12209.1916.14..6..13..10..15581811.1420..19...1..3.6.131012.95.15.14.17..1916..18..18.+
3.2....920..17.14.1911.4.185.3.2...10...169.717.819.4.18..311.2010..12..516...15.8..1841720...+
12711...515..191410..2..7.17.5.20.15.18.....8.20...712..18.17.15....18.17....17.

***** PROCESS-MANAGER TEST COMPLETED SUCCESSFULLY *****



*****  FRAME-MANAGER TEST  *****

1
FATAL ERROR in TestFrameManager: "addrSpace.numberOfPages is wrong" -- TERMINATING!

(To find out where execution was when the problem arose, type 'st' at the emulator prompt.)

====================  KPL PROGRAM TERMINATION  ====================

****  A 'debug' instruction was encountered  *****
Done!  The next instruction to execute will be:
001078: C0100000       sethi   0x0000,r1        ! 0x00001088 = 4232 (noGoMessage)

Entering machine-level debugger...
```

```
=======================================================
=====                                             =====
=====         The BLITZ Machine Emulator          =====
=====                                             =====
=====  Copyright 2001-2007, Harry H. Porter III  =====
=====                                             =====
=======================================================

Enter a command at the prompt.  Type 'quit' to exit or 'help' for
info about commands.
> quit
Number of Disk Reads    = 0
Number of Disk Writes   = 0
Instructions Executed   = 26383495
Time Spent Sleeping     = 0
    Total Elapsed Time  = 26383495
```

```
code Kernel

  -- Ted Timmons, tedt@pdx.edu

-----------------------------  InitializeScheduler  ---------------------------------

  function InitializeScheduler ()
    --
    -- This routine assumes that we are in System mode.  It sets up the
    -- thread scheduler and turns the executing program into "main-thread".
    -- After exit, we can execute "Yield", "Fork", etc.  Upon return, the
    -- main-thread will be executing with interrupts enabled.
    --
      Cleari ()
      print ("Initializing Thread Scheduler...\n")
      readyList = new List [Thread]
      threadsToBeDestroyed = new List [Thread]
      mainThread = new Thread
      mainThread.Init ("main-thread")
      mainThread.status = RUNNING
      idleThread = new Thread
      idleThread.Init ("idle-thread")
      idleThread.Fork (IdleFunction, 0)
      currentThread = & mainThread
      FatalError = FatalError_ThreadVersion       -- Use a routine which prints threadname
      currentInterruptStatus = ENABLED
      Seti ()
    endFunction

-----------------------------  IdleFunction  ---------------------------------

  function IdleFunction (arg: int)
    --
    -- This is the "idle thread", a kernel thread which ensures that the ready
    -- list is never empty.  The idle thread constantly yields to other threads
    -- in an infinite loop.  However, before yielding, it first checks to see if
    -- there are other threads.  If there are no other threads, the idle thread
    -- will execute the "wait" instruction.  The "wait" instruction will enable
    -- interrupts and halt CPU execution until the next interrupt arrives.
    --
      var junk: int
      while true
        junk = SetInterruptsTo (DISABLED)
        if readyList.IsEmpty ()
          Wait ()
        else
          currentThread.Yield ()
        endIf
      endWhile
    endFunction

-----------------------------  Run  ---------------------------------

  function Run (nextThread: ptr to Thread)
    --
    -- Begin executing the thread "nextThread", which has already
    -- been removed from the readyList.  The current thread will
    -- be suspended; we assume that its status has already been
    -- changed to READY or BLOCKED.  We assume that interrupts are
    -- DISABLED when called.
    --
    -- This routine is called only from "Thread.Yield" and "Thread.Sleep".
    --
    -- It is allowable for nextThread to be currentThread.
    --
      var prevThread, th: ptr to Thread
      prevThread = currentThread
      prevThread.CheckOverflow ()
      -- If the previous thread was using the USER registers, save them.
      if prevThread.isUserThread
        SaveUserRegs (&prevThread.userRegs[0])
      endIf
      currentThread = nextThread
      nextThread.status = RUNNING
      --print ("SWITCHING from ")
      --print (prevThread.name)
      --print (" to ")
      --print (nextThread.name)
      --print ("\n")
      Switch (prevThread, nextThread)
      --print ("After SWITCH, back in thread ")
      --print (currentThread.name)
      --print ("\n")
      while ! threadsToBeDestroyed.IsEmpty ()
        th = threadsToBeDestroyed.Remove()
        threadManager.FreeThread (th)
      endWhile
      -- If the new thread uses the USER registers, restore them.
      if currentThread.isUserThread
        RestoreUserRegs (&currentThread.userRegs[0])
        currentThread.myProcess.addrSpace.SetToThisPageTable ()
      endIf
    endFunction

-----------------------------  PrintReadyList  ---------------------------------

  function PrintReadyList ()
    --
    -- This routine prints the readyList.  It disables interrupts during the
    -- printing to guarantee that the readyList won’t change while it is
    -- being printed, which could cause disaster in this routine!
    --
    var oldStatus: int
      oldStatus = SetInterruptsTo (DISABLED)
      print ("Here is the ready list:\n")
      readyList.ApplyToEach (ThreadPrintShort)
      oldStatus = SetInterruptsTo (oldStatus)
    endFunction

-----------------------------  ThreadStartMain  ---------------------------------

  function ThreadStartMain ()
    --
    -- This function is called from the assembly language routine "ThreadStart".
    -- It is the first KPL code each thread will execute, and it will
    -- invoke the thread’s "main" function, with interrupts enabled.  If the "main"
    -- function ever returns, this function will terminate this thread.  This
    -- function will never return.
    --
      var
        junk: int
        mainFun: ptr to function (int)
      -- print ("ThreadStartMain...\n")
      junk = SetInterruptsTo (ENABLED)
      mainFun = currentThread.initialFunction
      mainFun (currentThread.initialArgument)
      ThreadFinish ()
      FatalError ("ThreadFinish should never return")
    endFunction

-----------------------------  ThreadFinish  ---------------------------------

  function ThreadFinish ()
    --
    -- As the last thing to do in this thread, we want to clean up
    -- and reclaim the Thread object.  This method is called as the
    -- last thing the thread does; this is the normal way for a thread
    -- to die.  However, since the thread is still running in this,
    -- we can’t actually do the clean up.  So we just make a note
    -- that it is pending.  After the next thread starts (in method "Run")
    -- we’ll finish the job.
    --
      var junk: int
      junk = SetInterruptsTo (DISABLED)
      -- print ("Finishing ")
      -- print (currentThread.name)
      -- print ("\n")
      threadsToBeDestroyed.AddToEnd (currentThread)
      currentThread.Sleep ()
      -- Execution will never reach the next instruction
      FatalError ("This thread will never run again")
    endFunction

-----------------------------  FatalError_ThreadVersion  -----------------------

  function FatalError_ThreadVersion (errorMessage: ptr to array of char)
    --
    -- This function will print out the name of the current thread and
    -- the given error message.  Then it will call "RuntimeExit" to
    -- shutdown the system.
    --
      var
        junk: int
      junk = SetInterruptsTo (DISABLED)
      print ("\nFATAL ERROR")
      if currentThread    -- In case errors occur before thread initialization
        print (" in ")
        print (currentThread.name)
      endIf
      print (": \"")
      print (errorMessage)
      print ("\" -- TERMINATING!\n\n")
      print ("(To find out where execution was when the problem arose, type ’st’ at the +
emulator prompt.)\n")
      RuntimeExit ()
    endFunction

-----------------------------  SetInterruptsTo  ---------------------------------

  function SetInterruptsTo (newStatus: int) returns int
    --
    -- This routine is passed a status (DISABLED or ENABLED).  It
    -- returns the previous interrupt status and sets the interrupt
    -- status to "newStatus".
    --
    -- Since this routine reads and modifies a shared variable
    -- (currentInterruptStatus), there is a danger of this routine
    -- being re-entered.  Therefore, it momentarily will disable
    -- interrupts, to ensure a valid update to this variable.
    --
      var
        oldStat: int
      Cleari ()
      oldStat = currentInterruptStatus
      if newStatus == ENABLED
        currentInterruptStatus = ENABLED
        Seti ()
      else
        currentInterruptStatus = DISABLED
        Cleari ()
      endIf
      return oldStat
    endFunction

-----------------------------  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  -----------

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

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

       method Lock ()
           var
             oldIntStat: int
           if heldBy == currentThread
             FatalError ("Attempt to lock a mutex by a thread already holding it")
           endIf
           oldIntStat = SetInterruptsTo (DISABLED)
           if !heldBy
             heldBy = currentThread
           else
             waitingThreads.AddToEnd (currentThread)
             currentThread.Sleep ()
           endIf
           oldIntStat = SetInterruptsTo (oldIntStat)
         endMethod
       -----------  Mutex . Unlock  -----------

       method Unlock ()
           var
             oldIntStat: int
             t: ptr to Thread
           if heldBy != currentThread
             FatalError ("Attempt to unlock a mutex by a thread not holding it")
           endIf
           oldIntStat = SetInterruptsTo (DISABLED)
           t = waitingThreads.Remove ()
           if t
             t.status = READY
             readyList.AddToEnd (t)
             heldBy = t
           else
             heldBy = null
           endIf
           oldIntStat = SetInterruptsTo (oldIntStat)
         endMethod

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

       method IsHeldByCurrentThread () returns bool
           return heldBy == currentThread
         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
```

```
-----------------------------  Thread  -----------------------------------

  behavior Thread

      ----------  Thread . Init  ----------

      method Init (n: String)
        --
        -- Initialize this Thread object, but do not schedule it for
        -- execution yet.
        --
          name = n
          status = JUST_CREATED
          -- The next line initializes the systemStack array, without filling it in.
          *((& systemStack) asPtrTo int) = SYSTEM_STACK_SIZE
          systemStack [0] = STACK_SENTINEL
          systemStack [SYSTEM_STACK_SIZE-1] = STACK_SENTINEL
          stackTop = & (systemStack[SYSTEM_STACK_SIZE-1])
          regs = new array of int { 13 of 0 }
          isUserThread = false
          userRegs = new array of int { 15 of 0 }
        endMethod

      ----------  Thread . Fork  ----------

      method Fork (fun: ptr to function (int), arg: int)
        --
        -- This method will schedule this thread for execution; in other words
        -- it will make it ready to run by adding it to the "ready queue."  This
        -- method is passed a function and a single integer argument.  When the
        -- thread runs, the thread will execute this function on that argument
        -- and then termiante.  This method will return after scheduling this
        -- thread.
        --
          var
            oldIntStat, junk: int
          oldIntStat = SetInterruptsTo (DISABLED)
          -- print ("Forking thread...\n")
          initialFunction = fun
          initialArgument = arg
          stackTop = stackTop - 4
          *(stackTop asPtrTo int) = ThreadStartUp asInteger
          status = READY
          readyList.AddToEnd (self)
          junk = SetInterruptsTo (oldIntStat)
        endMethod

      ----------  Thread . Yield  ----------

      method Yield ()
        --
        -- This method should only be invoked on the current thread.  The
        -- current thread may yield the processor to other threads by
        -- executing:
        --       currentThread.Yield ()
        -- This method may be invoked with or without interrupts enabled.
        -- Upon return, the interrupts will be in the same state; however
        -- since other threads are given a chance to run and they may allow
        -- interrupts, interrupts handlers may have been invoked before
        -- this method returns.
        --
          var
            nextTh: ptr to Thread
            oldIntStat, junk: int
          -- ASSERT:
              if self != currentThread
                FatalError ("In Yield, self != currentThread")
              endIf
          oldIntStat = SetInterruptsTo (DISABLED)
          -- print ("Yielding ")
          -- print (name)
          -- print ("\n")
          nextTh = readyList.Remove ()
          if nextTh
            -- print ("About to run ")
            -- print (nextTh.name)
            -- print ("\n")
            if status == BLOCKED
              FatalError ("Status of current thread should be READY or RUNNING")
            endIf
            status = READY
            readyList.AddToEnd (self)
            Run (nextTh)
          endIf
          junk = SetInterruptsTo (oldIntStat)
        endMethod

      ----------  Thread . Sleep  ----------

      method Sleep ()
        --
        -- This method should only be invoked on the current thread.  It
        -- will set the status of the current thread to BLCOKED and will
        -- will switch to executing another thread.  It is assumed that
        --     (1) Interrupts are disabled before calling this routine, and
        --     (2) The current thread has been placed on some other wait
        --         list (e.g., for a Semaphore) or else the thread will
        --         never get scheduled again.
        --
          var nextTh: ptr to Thread
          -- ASSERT:
              if currentInterruptStatus != DISABLED
                FatalError ("In Sleep, currentInterruptStatus != DISABLED")
              endIf
          -- ASSERT:
              if self != currentThread
                FatalError ("In Sleep, self != currentThread")
              endIf
          -- print ("Sleeping ")
          -- print (name)
          -- print ("\n")
          status = BLOCKED
          nextTh = readyList.Remove ()
          if nextTh == null
            FatalError ("Ready list should always contain the idle thread")
          endIf
          Run (nextTh)
        endMethod

      ----------  Thread . CheckOverflow  ----------

      method CheckOverflow ()
        --
        -- This method checks to see if this thread has overflowed its
        -- pre-alloted stack space.  WARNING: the approach taken here is only
        -- guaranteed to work "with high probability".
```

```
        --
          if systemStack[0] != STACK_SENTINEL
            FatalError ("System stack overflow detected!")
          elseIf systemStack[SYSTEM_STACK_SIZE-1] != STACK_SENTINEL
            FatalError ("System stack underflow detected!")
          endIf
        endMethod

      ----------  Thread . Print  ----------

      method Print ()
        --
        -- Print this object.
        --
          var i: int
              oldStatus: int
          oldStatus = SetInterruptsTo (DISABLED)
          print ("  Thread \"")
          print (name)
          print ("\"    (addr of Thread object: ")
          printHex (self asInteger)
          print (")\n")
          print ("    machine state:\n")
          for i = 0 to 12
            print ("      r")
            printInt (i+2)
            print (": ")
            printHex (regs[i])
            print ("   ")
            printInt (regs[i])
            print ("\n")
          endFor
          printHexVar ("    stackTop", stackTop asInteger)
          printHexVar ("    stack starting addr", (& systemStack[0]) asInteger)
          switch status
            case JUST_CREATED:
              print ("    status = JUST_CREATED\n")
              break
            case READY:
              print ("    status = READY\n")
              break
            case RUNNING:
              print ("    status = RUNNING\n")
              break
            case BLOCKED:
              print ("    status = BLOCKED\n")
              break
            case UNUSED:
              print ("    status = UNUSED\n")
              break
            default:
              FatalError ("Bad status in Thread")
          endSwitch
          print ("    is user thread: ")
          printBool (isUserThread)
          nl ()
          print ("    user registers:\n")
          for i = 0 to 14
            print ("      r")
            printInt (i+1)
            print (": ")
            printHex (userRegs[i])
            print ("   ")
            printInt (userRegs[i])
            print ("\n")
          endFor
          oldStatus = SetInterruptsTo (oldStatus)
        endMethod

  endBehavior

-----------------------------  ThreadPrintShort  ---------------------------------

  function ThreadPrintShort (t: ptr to Thread)
    --
    -- This function prints a single line giving the name of thread "t",
    -- its status, and the address of the Thread object itself (which may be
    -- helpful in distinguishing Threads when the name is not helpful).
    --
      var
        oldStatus: int = SetInterruptsTo (DISABLED)
      if !t
        print ("NULL\n")
        return
      endIf
      print ("  Thread \"")
      print (t.name)
      print ("\"    status=")
      switch t.status
        case JUST_CREATED:
          print ("JUST_CREATED")
          break
        case READY:
          print ("READY")
          break
        case RUNNING:
          print ("RUNNING")
          break
        case BLOCKED:
          print ("BLOCKED")
          break
        case UNUSED:
          print ("UNUSED")
          break
        default:
          FatalError ("Bad status in Thread")
      endSwitch
      print ("    (addr of Thread object: ")
      printHex (t asInteger)
      print (")")
      nl ()
      -- t.Print ()
      oldStatus = SetInterruptsTo (oldStatus)
    endFunction

-----------------------------  ThreadManager  ---------------------------------

  behavior ThreadManager

      ----------  ThreadManager . Init  ----------

      method Init ()
        var i: int = 0
        --
        -- This method is called once at kernel startup time to initialize
        -- the one and only "ThreadManager" object.
        --
          print ("Initializing Thread Manager...\n")

          -- init array of Threads
          threadTable = new array of Thread
                      { MAX_NUMBER_OF_PROCESSES of new Thread }
          -- init the individual threads. Unrolling the loop so we can
          -- more easily name the threads. (if only KPL supported easy
          -- concatenation..)
          threadTable[0].Init("thread 0")
          threadTable[1].Init("thread 1")
          threadTable[2].Init("thread 2")
          threadTable[3].Init("thread 3")
          threadTable[4].Init("thread 4")
          threadTable[5].Init("thread 5")
          threadTable[6].Init("thread 6")
          threadTable[7].Init("thread 7")
          threadTable[8].Init("thread 8")
          threadTable[9].Init("thread 9")

          --print("thread zero status: ")
          --printInt(threadTable[0].status)
          --print ("\n")
         print ("have threads.\n")

          -- init freeList
          -- Not worrying about our mutex, because we are in the init routine,
          -- so we are pretty much guaranteed to be single-threaded.
          freeList = new List[Thread]

          -- set each thread to UNUSED
          -- put all Threads into the freeList
          for i = 0 to MAX_NUMBER_OF_PROCESSES-1
            threadTable[i].status = UNUSED
            freeList.AddToEnd(&threadTable[i])
          endFor

          -- set up our mutex
          threadCheck = new Mutex
          threadCheck.Init()

          -- init our condition (monitor)
          threadFree = new Condition
          threadFree.Init()
        endMethod

      ----------  ThreadManager . Print  ----------

      method Print ()
        --
        -- Print each thread.  Since we look at the freeList, this
        -- routine disables interrupts so the printout will be a
        -- consistent snapshot of things.
        --
        var i, oldStatus: int
          oldStatus = SetInterruptsTo (DISABLED)
          print ("Here is the thread table...\n")
          for i = 0 to MAX_NUMBER_OF_PROCESSES-1
            print ("  ")
            printInt (i)
            print (":")
            ThreadPrintShort (&threadTable[i])
          endFor
          print ("Here is the FREE list of Threads:\n   ")
          freeList.ApplyToEach (PrintObjectAddr)
          nl ()
          oldStatus = SetInterruptsTo (oldStatus)
        endMethod

      ----------  ThreadManager . GetANewThread  ----------

      method GetANewThread () returns ptr to Thread
        var
          --i: int = 0
          currThread: ptr to Thread
        --
        -- This method returns a new Thread; it will wait
        -- until one is available.
        --

          threadCheck.Lock()
          -- check free status, then wait for a thread to be available
          if (freeList.IsEmpty())
            while( ! currThread )
              if (threadCheck.IsHeldByCurrentThread())
                currThread = freeList.Remove()
              endIf

              if (! currThread)
                --print ("in loop, no currThread\n")
                threadFree.Wait(&threadCheck)
              endIf
            endWhile
          else
            -- simple case: freeList isn’t empty
            currThread = freeList.Remove()
          endIf

          -- lock the mutex so we can safely remove a thread.
          --threadCheck.Lock()

          --print ("got thread from freeList: ")
          --print (currThread.name)
          --print ("\n")
          currThread.status = JUST_CREATED

          -- We’re done with the mutex, unlock it.
          threadCheck.Unlock()
          return currThread
        endMethod

      ----------  ThreadManager . FreeThread  ----------

      method FreeThread (th: ptr to Thread)
        --
        -- This method is passed a ptr to a Thread;  It moves it
        -- to the FREE list.
        --

          -- lock the mutex so we can safely re-add the thread.
          threadCheck.Lock()

          --print ("giving thread back to freeList: ")
          --print (th.name)
          --print ("\n")
          th.status = UNUSED
          freeList.AddToEnd(th)

          -- done with the mutex.
          threadFree.Signal(&threadCheck)
          threadCheck.Unlock()
        endMethod

    endBehavior

--------------------------  ProcessControlBlock  ------------------------------

  behavior ProcessControlBlock

      ----------  ProcessControlBlock . Init  ----------
      --
      -- This method is called once for every PCB at startup time.
      --
      method Init ()
          pid = -1
          status = FREE
          addrSpace = new AddrSpace
          addrSpace.Init ()
-- Uncomment this code later...
/*
          fileDescriptor = new array of ptr to OpenFile
                      { MAX_FILES_PER_PROCESS of null }
*/
        endMethod

      ----------  ProcessControlBlock . Print  ----------

      method Print ()
        --
        -- Print this ProcessControlBlock using several lines.
        --
        -- var i: int
          self.PrintShort ()
          addrSpace.Print ()
          print ("    myThread = ")
          ThreadPrintShort (myThread)
-- Uncomment this code later...
/*
          print ("    File Descriptors:\n")
          for i = 0 to MAX_FILES_PER_PROCESS-1
            if fileDescriptor[i]
              fileDescriptor[i].Print ()
            endIf
          endFor
*/
          nl ()
        endMethod

      ----------  ProcessControlBlock . PrintShort  ----------

      method PrintShort ()
        --
        -- Print this ProcessControlBlock on one line.
        --
          print ("  ProcessControlBlock   (addr=")
          printHex (self asInteger)
          print (")   pid=")
          printInt (pid)
          print (", status=")
          if status == ACTIVE
            print ("ACTIVE")
          elseIf status == ZOMBIE
            print ("ZOMBIE")
          elseIf status == FREE
            print ("FREE")
          else
            FatalError ("Bad status in ProcessControlBlock")
          endIf
          print (", parentsPid=")
          printInt (parentsPid)
          print (", exitStatus=")
          printInt (exitStatus)
          nl ()
        endMethod

    endBehavior

-----------------------------  ProcessManager  ---------------------------------

  behavior ProcessManager

      ----------  ProcessManager . Init  ----------

      method Init ()
        --
        -- This method is called once at kernel startup time to initialize
        -- the one and only "processManager" object.
        --
        var i: int = 0

        -- init freeList
        freeList = new List[ProcessControlBlock]
        -- no Init() method, actually.
        --  freeList.Init()

        -- init the processTable array
        processTable = new array of ProcessControlBlock { MAX_NUMBER_OF_PROCESSES of new +
ProcessControlBlock }

        -- init all ProcessControlBlocks in processTable array
        -- place PCBs on freeList
        for i = 0 to MAX_NUMBER_OF_PROCESSES - 1
          processTable[i].Init()
          freeList.AddToEnd(&processTable[i])
        endFor

        -- init processManagerLock
        processManagerLock = new Mutex
        processManagerLock.Init()

        -- init aProcessBecameFree, aProcessDied vars
        aProcessBecameFree = new Condition
        aProcessBecameFree.Init()
        aProcessDied = new Condition
        aProcessDied.Init()

        endMethod

      ----------  ProcessManager . Print  ----------

      method Print ()
        --
        -- Print all processes.  Since we look at the freeList, this
        -- routine disables interrupts so the printout will be a
        -- consistent snapshot of things.
        --
        var i, oldStatus: int
          oldStatus = SetInterruptsTo (DISABLED)
          print ("Here is the process table...\n")
          for i = 0 to MAX_NUMBER_OF_PROCESSES-1
            print ("  ")
            printInt (i)
            print (":")
            processTable[i].Print ()
          endFor
          print ("Here is the FREE list of ProcessControlBlocks:\n   ")
          freeList.ApplyToEach (PrintObjectAddr)
          nl ()
          oldStatus = SetInterruptsTo (oldStatus)
        endMethod

      ----------  ProcessManager . PrintShort  ----------

      method PrintShort ()
        --
        -- Print all processes.  Since we look at the freeList, this
        -- routine disables interrupts so the printout will be a
        -- consistent snapshot of things.
        --
        var i, oldStatus: int
          oldStatus = SetInterruptsTo (DISABLED)
          print ("Here is the process table...\n")
          for i = 0 to MAX_NUMBER_OF_PROCESSES-1
            print ("  ")
            printInt (i)
            processTable[i].PrintShort ()
          endFor
          print ("Here is the FREE list of ProcessControlBlocks:\n   ")
          freeList.ApplyToEach (PrintObjectAddr)
          nl ()
          oldStatus = SetInterruptsTo (oldStatus)
        endMethod

      ----------  ProcessManager . GetANewProcess  ----------

      method GetANewProcess () returns ptr to ProcessControlBlock
        --
        -- This method returns a new ProcessControlBlock; it will wait
        -- until one is available.
        --
          var currProcess: ptr to ProcessControlBlock
          --print ("\nHello from GANP\n")

          -- Safely pull the next item off the freeList.
          processManagerLock.Lock()
          if (freeList.IsEmpty())
            while (! currProcess)
              if (processManagerLock.IsHeldByCurrentThread())
                currProcess = freeList.Remove()
              endIf
              -- print ("got thread\n")

              -- we don’t have/didn’t get a currProcess, so we need to
              -- patiently Wait.
```

```
              if (! currProcess)
                aProcessDied.Wait(&processManagerLock)
              endIf
            endWhile
          else
            currProcess = freeList.Remove()
          endIf

          -- release our lock. We’re done!
          processManagerLock.Unlock()
          return currProcess
        endMethod

      ----------  ProcessManager . FreeProcess  ----------

      method FreeProcess (p: ptr to ProcessControlBlock)
        --
        -- This method is passed a ptr to a Process;  It moves it
        -- to the FREE list.
        --
        processManagerLock.Lock()

        -- mark process as ready
        p.status = FREE
        -- add to the free list
        freeList.AddToEnd(p)

        -- finish signaling
        aProcessDied.Signal(&processManagerLock)
        processManagerLock.Unlock()


        endMethod


    endBehavior

-----------------------------  PrintObjectAddr  ---------------------------------

  function PrintObjectAddr (p: ptr to Object)
    --
    -- Print the address of the given object.
    --
      printHex (p asInteger)
      printChar (’ ’)
    endFunction

-----------------------------  ProcessFinish  --------------------------

  function ProcessFinish (exitStatus: int)
    --
    -- This routine is called when a process is to be terminated.  It will
    -- free the resources held by this process and will terminate the
    -- current thread.
    --
      FatalError ("ProcessFinish is not implemented")
    endFunction

-----------------------------  FrameManager  ---------------------------------

  behavior FrameManager

      ----------  FrameManager . Init  ----------
      method Init ()
        --
        -- This method is called once at kernel startup time to initialize
        -- the one and only "frameManager" object.
        --
        var i: int
          print ("Initializing Frame Manager...\n")
          framesInUse = new BitMap
          framesInUse.Init (NUMBER_OF_PHYSICAL_PAGE_FRAMES)
          numberFreeFrames = NUMBER_OF_PHYSICAL_PAGE_FRAMES
          frameManagerLock = new Mutex
          frameManagerLock.Init ()
          newFramesAvailable = new Condition
          newFramesAvailable.Init ()
          -- Check that the area to be used for paging contains zeros.
          -- The BLITZ emulator will initialize physical memory to zero, so
          -- if by chance the size of the kernel has gotten so large that
          -- it runs into the area reserved for pages, we will detect it.
          -- Note: this test is not 100%, but is included nonetheless.
          for i = PHYSICAL_ADDRESS_OF_FIRST_PAGE_FRAME
               to PHYSICAL_ADDRESS_OF_FIRST_PAGE_FRAME+300
               by 4
            if 0 != *(i asPtrTo int)
              FatalError ("Kernel code size appears to have grown too large and is +
overflowing into the frame region")
            endIf
          endFor
        endMethod

      ----------  FrameManager . Print  ----------

      method Print ()
        --
        -- Print which frames are allocated and how many are free.
        --
          frameManagerLock.Lock ()
          print ("FRAME MANAGER:\n")
          printIntVar ("  numberFreeFrames", numberFreeFrames)
          print ("  Here are the frames in use: \n    ")
          framesInUse.Print ()
          frameManagerLock.Unlock ()
        endMethod

      ----------  FrameManager . GetAFrame  ----------

      method GetAFrame () returns int
        --
        -- Allocate a single frame and return its physical address.  If no frames
        -- are currently available, wait until the request can be completed.
        --
          var f, frameAddr: int

          -- Acquire exclusive access to the frameManager data structure...
          frameManagerLock.Lock ()

          -- Wait until we have enough free frames to entirely satisfy the request...
          while numberFreeFrames < 1
            newFramesAvailable.Wait (&frameManagerLock)
          endWhile

          -- Find a free frame and allocate it...
          f = framesInUse.FindZeroAndSet ()
          numberFreeFrames = numberFreeFrames - 1

          -- Unlock...
          frameManagerLock.Unlock ()

          -- Compute and return the physical address of the frame...
          frameAddr = PHYSICAL_ADDRESS_OF_FIRST_PAGE_FRAME + (f * PAGE_SIZE)
          -- printHexVar ("GetAFrame returning frameAddr", frameAddr)
          return frameAddr
        endMethod

      ----------  FrameManager . GetNewFrames  ----------

      method GetNewFrames (aPageTable: ptr to AddrSpace, numFramesNeeded: int)
          -- GetAFrame allocates one frame (waits until oen is available)
          -- NOT IMPLEMENTED
          -- store addr of each frame into AddrSpace object
          -- acquire frame manager lock
          -- wait on newFramesAvailable until numFramesNeeded frames available
          -- for each frame:
            -- determine which frame is free
            -- figure out address of free frame
            -- store address of frame
            -- aPageTable.SetFrameAddr (i, frameAddr)
         -- adjust number of free frames
         -- Set aPageTable.numberOfPages to the number of frames allocated
         -- Unlock the frame manager
        endMethod

      ----------  FrameManager . ReturnAllFrames  ----------

      method ReturnAllFrames (aPageTable: ptr to AddrSpace)
          -- NOT IMPLEMENTED
        endMethod

    endBehavior

-----------------------------  AddrSpace  ---------------------------------

  behavior AddrSpace

      ----------  AddrSpace . Init  ----------

      method Init ()
        --
        -- Initialize this object.
        --
          numberOfPages = 0
          pageTable = new array of int { MAX_PAGES_PER_VIRT_SPACE of 0x00000003 }
        endMethod

      ----------  AddrSpace . Print  ----------

      method Print ()
        --
        -- Print this object.
        --
          var i: int
          print ("        addr        entry          Logical    Physical   Undefined Bits  +
Dirty  Referenced  Writeable  Valid\n")
          print ("     ==========   ==========     ==========  ==========  =============  +
=====  ==========  =========  =====\n")
          for i = 0 to numberOfPages-1
            print ("     ")
            printHex ((&pageTable[i]) asInteger)
            print (":  ")
            printHex (pageTable[i])
            print ("     ")
            printHex (i * PAGE_SIZE)   -- Logical address
            print ("  ")
            printHex (self.ExtractFrameAddr (i))       -- Physical address
            print ("    ")
            if self.ExtractUndefinedBits (i) != 0
              printHex (self.ExtractUndefinedBits (i))
            else
              print ("          ")
            endIf
            print ("     ")
            if self.IsDirty (i)
              print ("YES")
            else
              print ("   ")
            endIf
            print ("      ")
            if self.IsReferenced (i)
              print ("YES")
            else
              print ("   ")
            endIf
            print ("         ")
            if self.IsWritable (i)
              print ("YES")
            else
              print ("   ")
            endIf
            print ("      ")
            if self.IsValid (i)
              print ("YES")
            else
              print ("   ")
            endIf
            nl ()
          endFor
        endMethod

      ----------  AddrSpace . ExtractFrameAddr  ----------

      method ExtractFrameAddr (entry: int) returns int
        --
        -- Return the physical address of the frame in the selected page
        -- table entry.
        --
          return (pageTable[entry] & 0xffffe000)
        endMethod

      ----------  AddrSpace . ExtractUndefinedBits  ----------

      method ExtractUndefinedBits (entry: int) returns int
        --
        -- Return the undefined bits in the selected page table entry.
        --
          return (pageTable[entry] & 0x00001ff0)
        endMethod

      ----------  AddrSpace . SetFrameAddr  ----------
```

```
      method SetFrameAddr (entry: int, frameAddr: int)
        --
        -- Set the physical address of the frame in the selected page
        -- table entry to the value of the argument "frameAddr".
        --
          pageTable[entry] = (pageTable[entry] & 0x00001fff) | frameAddr
        endMethod

      ----------  AddrSpace . IsDirty  ----------

      method IsDirty (entry: int) returns bool
        --
        -- Return true if the selected page table entry is marked "dirty".
        --
          return (pageTable[entry] & 0x00000008) != 0
        endMethod

      ----------  AddrSpace . IsReferenced  ----------

      method IsReferenced (entry: int) returns bool
        --
        -- Return true if the selected page table entry is marked "referenced".
        --
          return (pageTable[entry] & 0x00000004) != 0
        endMethod

      ----------  AddrSpace . IsWritable  ----------

      method IsWritable (entry: int) returns bool
        --
        -- Return true if the selected page table entry is marked "writable".
        --
          return (pageTable[entry] & 0x00000002) != 0
        endMethod

      ----------  AddrSpace . IsValid  ----------

      method IsValid (entry: int) returns bool
        --
        -- Return true if the selected page table entry is marked "valid".
        --
          return (pageTable[entry] & 0x00000001) != 0
        endMethod

      ----------  AddrSpace . SetDirty  ----------

      method SetDirty (entry: int)
        --
        -- Set the selected page table entry’s "dirty" bit to 1.
        --
          pageTable[entry] = pageTable[entry] | 0x00000008
        endMethod

      ----------  AddrSpace . SetReferenced  ----------

      method SetReferenced (entry: int)
        --
        -- Set the selected page table entry’s "referenced" bit to 1.
        --
          pageTable[entry] = pageTable[entry] | 0x00000004
        endMethod

      ----------  AddrSpace . SetWritable  ----------
```

```
  method SetWritable (entry: int)
    --
    -- Set the selected page table entry’s "writable" bit to 1.
    --
      pageTable[entry] = pageTable[entry] | 0x00000002
    endMethod

  ----------  AddrSpace . SetValid  ----------

  method SetValid (entry: int)
    --
    -- Set the selected page table entry’s "valid" bit to 1.
    --
      pageTable[entry] = pageTable[entry] | 0x00000001
    endMethod

  ----------  AddrSpace . ClearDirty  ----------

  method ClearDirty (entry: int)
    --
    -- Clear the selected page table entry’s "dirty" bit.
    --
      pageTable[entry] = pageTable[entry] & ! 0x00000008
    endMethod

  ----------  AddrSpace . ClearReferenced  ----------

  method ClearReferenced (entry: int)
    --
    -- Clear the selected page table entry’s "referenced" bit.
    --
      pageTable[entry] = pageTable[entry] & ! 0x00000004
    endMethod

  ----------  AddrSpace . ClearWritable  ----------

  method ClearWritable (entry: int)
    --
    -- Clear the selected page table entry’s "writable" bit.
    --
      pageTable[entry] = pageTable[entry] & ! 0x00000002
    endMethod

  ----------  AddrSpace . ClearValid  ----------

  method ClearValid (entry: int)
    --
    -- Clear the selected page table entry’s "valid" bit.
    --
      pageTable[entry] = pageTable[entry] & ! 0x00000001
    endMethod

  ----------  AddrSpace . SetToThisPageTable  ----------

  method SetToThisPageTable ()
    --
    -- This method sets the page table registers in the CPU to
    -- point to this page table.  Later, when paging is enabled,
    -- this will become the active virtual address space.
    --
      LoadPageTableRegs ((& pageTable[0]) asInteger, numberOfPages*4)
    endMethod
  ----------  AddrSpace . CopyBytesFromVirtual  ----------

  method CopyBytesFromVirtual (kernelAddr, virtAddr, numBytes: int)
                returns int
    --
    -- This method copies data from a user's virtual address space
    -- to somewhere in the kernel space.  We assume that the
    -- pages of the virtual address space are resident in
    -- physical page frames.  This routine returns the number of bytes
    -- that were copied; if there was any problem with the virtual
    -- addressed data, it returns -1.
    --
      var copiedSoFar, virtPage, offset, fromAddr: int
      -- print ("CopyBytesFromVirtual called...\n")
      -- printHexVar ("  kernelAddr", kernelAddr)
      -- printHexVar ("  virtAddr", virtAddr)
      -- printIntVar ("  numBytes", numBytes)
      if numBytes == 0
        return 0
      elseIf numBytes < 0
        return -1
      endIf
      virtPage = virtAddr / PAGE_SIZE
      offset = virtAddr % PAGE_SIZE
      -- printHexVar ("  virtPage", virtPage)
      -- printHexVar ("  offset", offset)
      while true
        if virtPage >= numberOfPages
          print ("  Virtual page number is too large!!!\n")
          return -1
        endIf
        if ! self.IsValid (virtPage)
          print ("  Virtual page is not marked VALID!!!\n")
          return -1
        endIf
        fromAddr = self.ExtractFrameAddr (virtPage) + offset
        -- printHexVar ("  Copying bytes from physcial addr", fromAddr)
        while offset < PAGE_SIZE
          -- printHexVar ("  Copying a byte to physcial addr", kernelAddr)
          -- printChar (* (fromAddr asPtrTo char))
          * (kernelAddr asPtrTo char) = * (fromAddr asPtrTo char)
          offset = offset + 1
          kernelAddr = kernelAddr + 1
          fromAddr = fromAddr + 1
          copiedSoFar = copiedSoFar + 1
          if copiedSoFar == numBytes
            return copiedSoFar
          endIf
        endWhile
        virtPage = virtPage + 1
        offset = 0
      endWhile
    endMethod

  ----------  AddrSpace . CopyBytesToVirtual  ----------

  method CopyBytesToVirtual (virtAddr, kernelAddr, numBytes: int)
                returns int
    --
    -- This method copies data from the kernel's address space to
    -- somewhere in the virtual address space.  We assume that the
    -- pages of the virtual address space are resident in physical
    -- page frames.  This routine returns the number of bytes
    -- that were copied; if there was any problem with the virtual
    -- addressed data, it returns -1.
    --
      var copiedSoFar, virtPage, offset, destAddr: int
      if numBytes == 0
        return 0
      elseIf numBytes < 0
        return -1
      endIf
      virtPage = virtAddr / PAGE_SIZE
      offset = virtAddr % PAGE_SIZE
      while true
        if (virtPage >= numberOfPages) ||
           (! self.IsValid (virtPage)) ||
           (! self.IsWritable (virtPage))
          return -1
        endIf
        destAddr = self.ExtractFrameAddr (virtPage) + offset
        while offset < PAGE_SIZE
          * (destAddr asPtrTo char) = * (kernelAddr asPtrTo char)
          offset = offset + 1
          kernelAddr = kernelAddr + 1
          destAddr = destAddr + 1
          copiedSoFar = copiedSoFar + 1
          if copiedSoFar == numBytes
            return copiedSoFar
          endIf
        endWhile
        virtPage = virtPage + 1
        offset = 0
      endWhile
    endMethod

  ----------  AddrSpace . GetStringFromVirtual  ----------

  method GetStringFromVirtual (kernelAddr: String, virtAddr, maxSize: int) returns int
    --
    -- This method is used to copy a String from virtual space into
    -- a given physical address in the kernel.  The "kernelAddr" should be
    -- a pointer to an "array of char" in the kernel's code.  This method
    -- copies up to "maxSize" characters from approriate page frame to this
    -- to the target array in the kernel.
    --
    -- Note: This method resets the "arraySize" word in the target.  It is
    -- assumed that the target array has enough space; no checking is done.
    -- The caller should supply a "maxSize" telling how many characters may
    -- be safely copied.
    --
    -- If there are problems, then -1 is returned.  Possible problems:
    --       The source array has more than "maxSize" elements
    --       The source page is invalid or out of range
    -- If all okay, then the number of characters copied is returned.
    --
      var sourceSize: int
      -- print ("GetStringFromVirtual called...\n")
      -- printHexVar ("  kernelAddr", kernelAddr asInteger)
      -- printHexVar ("  virtAddr", virtAddr)
      -- printIntVar ("  maxSize", maxSize)
      -- Begin by fetching the source size
      if self.CopyBytesFromVirtual ((&sourceSize) asInteger,
                                    virtAddr,
                                    4) < 4
            return -1
          endIf
          -- printIntVar ("  sourceSize", sourceSize)
          -- Make sure the source size is okay
          if sourceSize > maxSize
            return -1
          endIf
          -- Change the size of the destination array
          * (kernelAddr asPtrTo int) = sourceSize
          -- Next, get the characters
          return self.CopyBytesFromVirtual (kernelAddr asInteger + 4,
                                            virtAddr + 4,
                                            sourceSize)
        endMethod

    endBehavior

-----------------------------  TimerInterruptHandler  ---------------------------------

  function TimerInterruptHandler ()
    --
    -- This routine is called when a timer interrupt occurs.  Upon entry,
    -- interrupts are DISABLED.  Upon return, execution will return to
    -- the interrupted process, which necessarily had interrupts ENABLED.
    --
    -- (If you wish to turn time-slicing off, simply disable the call
    -- to "Yield" in the code below.  Threads will then execute until they
    -- call "Yield" explicitly, or until they call "Sleep".)
    --
      currentInterruptStatus = DISABLED
      -- printChar ('_')
      currentThread.Yield ()
      currentInterruptStatus = ENABLED
    endFunction

-----------------------------  DiskInterruptHandler  --------------------------

  function DiskInterruptHandler ()
    --
    -- This routine is called when a disk interrupt occurs.  It will
    -- signal the "semToSignalOnCompletion" Semaphore and return to
    -- the interrupted thread.
    --
    -- This is an interrupt handler.  As such, interrupts will be DISABLED
    -- for the duration of its execution.
    --
-- Uncomment this code later...
      FatalError ("DISK INTERRUPTS NOT EXPECTED IN PROJECT 4")
/*
      currentInterruptStatus = DISABLED
      -- print ("DiskInterruptHandler invoked!\n")
      if diskDriver.semToSignalOnCompletion
        diskDriver.semToSignalOnCompletion.Up()
      endIf
*/
    endFunction

-----------------------------  SerialInterruptHandler  --------------------------

  function SerialInterruptHandler ()
    --
    -- This routine is called when a serial interrupt occurs.  It will
    -- signal the "semToSignalOnCompletion" Semaphore and return to
    -- the interrupted thread.
    --
    -- This is an interrupt handler.  As such, interrupts will be DISABLED
    -- for the duration of its execution.
    --
      currentInterruptStatus = DISABLED
      -- NOT IMPLEMENTED
    endFunction
-----------------------------  IllegalInstructionHandler  --------------------------

  function IllegalInstructionHandler ()
    --
    -- This routine is called when an IllegalInstruction exception occurs.  Upon entry,
    -- interrupts are DISABLED.  We should not return to the code that had
    -- the exception.
    --
      currentInterruptStatus = DISABLED
      ErrorInUserProcess ("An IllegalInstruction exception has occured while in user mode")
    endFunction

-----------------------------  ArithmeticExceptionHandler  --------------------------

  function ArithmeticExceptionHandler ()
    --
    -- This routine is called when an ArithmeticException occurs.  Upon entry,
    -- interrupts are DISABLED.  We should not return to the code that had
    -- the exception.
    --
      currentInterruptStatus = DISABLED
      ErrorInUserProcess ("An ArithmeticException exception has occured while in user mode")
    endFunction

-----------------------------  AddressExceptionHandler  --------------------------

  function AddressExceptionHandler ()
    --
    -- This routine is called when an AddressException occurs.  Upon entry,
    -- interrupts are DISABLED.  We should not return to the code that had
    -- the exception.
    --
      currentInterruptStatus = DISABLED
      ErrorInUserProcess ("An AddressException exception has occured while in user mode")
    endFunction

-----------------------------  PageInvalidExceptionHandler  --------------------------

  function PageInvalidExceptionHandler ()
    --
    -- This routine is called when a PageInvalidException occurs.  Upon entry,
    -- interrupts are DISABLED.  For now, we simply print a message and abort
    -- the thread.
    --
      currentInterruptStatus = DISABLED
      ErrorInUserProcess ("A PageInvalidException exception has occured while in user mode")
    endFunction

-----------------------------  PageReadonlyExceptionHandler  --------------------------

  function PageReadonlyExceptionHandler ()
    --
    -- This routine is called when a PageReadonlyException occurs.  Upon entry,
    -- interrupts are DISABLED.  For now, we simply print a message and abort
    -- the thread.
    --
      currentInterruptStatus = DISABLED
      ErrorInUserProcess ("A PageReadonlyException exception has occured while in user mode")
    endFunction

-----------------------------  PrivilegedInstructionHandler  --------------------------

  function PrivilegedInstructionHandler ()
    --
    -- This routine is called when a PrivilegedInstruction exception occurs.  Upon entry,
    -- interrupts are DISABLED.  We should not return to the code that had
    -- the exception.
    --
      currentInterruptStatus = DISABLED
      ErrorInUserProcess ("A PrivilegedInstruction exception has occured while in user mode")
    endFunction

-----------------------------  AlignmentExceptionHandler  --------------------------

  function AlignmentExceptionHandler ()
    --
    -- This routine is called when an AlignmentException occurs.  Upon entry,
    -- interrupts are DISABLED.  We should not return to the code that had
    -- the exception.
    --
      currentInterruptStatus = DISABLED
      ErrorInUserProcess ("An AlignmentException exception has occured while in user mode")
    endFunction

-----------------------------  ErrorInUserProcess  --------------------------

  function ErrorInUserProcess (errorMessage: String)
    --
    -- This routine is called when an error has occurred in a user-level
    -- process.  It prints the error message and terminates the process.
    --
      print ("\n**********  ")
      print (errorMessage)
      print ("  **********\n\n")

      -- Print some information about the offending process...
      if currentThread.myProcess
        currentThread.myProcess.Print ()
      else
        print ("  ERROR: currentThread.myProcess is null\n\n")
      endIf
      currentThread.Print ()

      -- Uncomment the following for even more information...
      -- threadManager.Print ()
      -- processManager.Print ()

      ProcessFinish (-1)
    endFunction

-----------------------------  SyscallTrapHandler  --------------------------

  function SyscallTrapHandler (syscallCodeNum, arg1, arg2, arg3, arg4: int) returns int
    --
    -- This routine is called when a syscall trap occurs.  Upon entry, interrupts
    -- will be DISABLED, paging is disabled, and we will be running in System mode.
    -- Upon return, execution will return to the user mode portion of this
    -- thread, which will have had interrupts ENABLED.
    --
      currentInterruptStatus = DISABLED
      /*****
      print ("Within SyscallTrapHandler: syscallCodeNum=")
      printInt (syscallCodeNum)
      print (", arg1=")
      printInt (arg1)
      print (", arg2=")
      printInt (arg2)
      print (", arg3=")
      printInt (arg3)
      print (", arg4=")
      printInt (arg4)
      nl ()
      *****/
      switch syscallCodeNum
        case SYSCALL_FORK:
          return Handle_Sys_Fork ()
        case SYSCALL_YIELD:
          Handle_Sys_Yield ()
          return 0
        case SYSCALL_EXEC:
          return Handle_Sys_Exec (arg1 asPtrTo array of char)
        case SYSCALL_JOIN:
          return Handle_Sys_Join (arg1)
        case SYSCALL_EXIT:
          Handle_Sys_Exit (arg1)
          return 0
        case SYSCALL_CREATE:
          return Handle_Sys_Create (arg1 asPtrTo array of char)
        case SYSCALL_OPEN:
          return Handle_Sys_Open (arg1 asPtrTo array of char)
        case SYSCALL_READ:
          return Handle_Sys_Read (arg1, arg2 asPtrTo char, arg3)
        case SYSCALL_WRITE:
          return Handle_Sys_Write (arg1, arg2 asPtrTo char, arg3)
        case SYSCALL_SEEK:
          return Handle_Sys_Seek (arg1, arg2)
        case SYSCALL_CLOSE:
          Handle_Sys_Close (arg1)
          return 0
        case SYSCALL_SHUTDOWN:
          Handle_Sys_Shutdown ()
          return 0
        default:
          print ("Syscall code = ")
          printInt (syscallCodeNum)
          nl ()
          FatalError ("Unknown syscall code from user thread")
      endSwitch
      return 0
    endFunction

-----------------------------  Handle_Sys_Exit  ---------------------------------

  function Handle_Sys_Exit (returnStatus: int)
      -- NOT IMPLEMENTED
    endFunction

-----------------------------  Handle_Sys_Shutdown  ---------------------------------

  function Handle_Sys_Shutdown ()
      -- NOT IMPLEMENTED
    endFunction

-----------------------------  Handle_Sys_Yield  ---------------------------------

  function Handle_Sys_Yield ()
      -- NOT IMPLEMENTED
    endFunction

-----------------------------  Handle_Sys_Fork  ---------------------------------

  function Handle_Sys_Fork () returns int
      -- NOT IMPLEMENTED
      return 0
    endFunction

-----------------------------  Handle_Sys_Join  ---------------------------------

  function Handle_Sys_Join (processID: int) returns int
      -- NOT IMPLEMENTED
      return 0
    endFunction

-----------------------------  Handle_Sys_Exec  ---------------------------------

  function Handle_Sys_Exec (filename: ptr to array of char) returns int
      -- NOT IMPLEMENTED
      return 0
    endFunction

-----------------------------  Handle_Sys_Create  ---------------------------------

  function Handle_Sys_Create (filename: ptr to array of char) returns int
      -- NOT IMPLEMENTED
      return 0
    endFunction

-----------------------------  Handle_Sys_Open  ---------------------------------

  function Handle_Sys_Open (filename: ptr to array of char) returns int
      -- NOT IMPLEMENTED
      return 0
    endFunction

-----------------------------  Handle_Sys_Read  ---------------------------------

  function Handle_Sys_Read (fileDesc: int, buffer: ptr to char, sizeInBytes: int) returns int
      -- NOT IMPLEMENTED
      return 0
    endFunction

-----------------------------  Handle_Sys_Write  ---------------------------------

  function Handle_Sys_Write (fileDesc: int, buffer: ptr to char, sizeInBytes: int) returns int
      -- NOT IMPLEMENTED
      return 0
    endFunction

-----------------------------  Handle_Sys_Seek  ---------------------------------

  function Handle_Sys_Seek (fileDesc: int, newCurrentPos: int) returns int
      -- NOT IMPLEMENTED
      return 0
    endFunction
```

```
-----------------------------  Handle_Sys_Close  ---------------------------------

  function Handle_Sys_Close (fileDesc: int)
      -- NOT IMPLEMENTED
    endFunction

endCode
```

```
header Kernel

  uses System, List, BitMap

  const

    SYSTEM_STACK_SIZE = 1000       -- in words
    STACK_SENTINEL = 0x24242424    -- in ASCII, this is "$$$$"

    -- The kernel code will load into the first megabyte of physical memory.  This
    -- should be more than enough.  We will use the second megabyte for page frames.
    -- Thus, the frame region is 128 page frames of 8K each.

    PAGE_SIZE = 8192                                  -- in hex: 0x0000 2000
    PHYSICAL_ADDRESS_OF_FIRST_PAGE_FRAME = 1048576    -- in hex: 0x0010 0000
    --NUMBER_OF_PHYSICAL_PAGE_FRAMES = 512              -- in hex: 0x0000 0200
    NUMBER_OF_PHYSICAL_PAGE_FRAMES = 27               -- for testing only

    MAX_NUMBER_OF_PROCESSES = 10
    MAX_STRING_SIZE = 20
    MAX_PAGES_PER_VIRT_SPACE = 20
    MAX_FILES_PER_PROCESS = 10
    MAX_NUMBER_OF_FILE_CONTROL_BLOCKS = 10
    MAX_NUMBER_OF_OPEN_FILES = 10
    USER_STACK_SIZE_IN_PAGES = 1
    NUMBER_OF_ENVIRONMENT_PAGES = 0

    SERIAL_GET_BUFFER_SIZE = 10
    SERIAL_PUT_BUFFER_SIZE = 10

  enum JUST_CREATED, READY, RUNNING, BLOCKED, UNUSED      -- Thread status
  enum ENABLED, DISABLED                                  -- Interrupt status
  enum FILE, TERMINAL, PIPE                               -- Kinds of OpenFile

  -- Syscall code numbers for kernel interface routines
  -- NOTE: These codes must exactly match an identical enum in UserSystem.h
  enum SYSCALL_EXIT = 1,
       SYSCALL_SHUTDOWN,
       SYSCALL_YIELD,
       SYSCALL_FORK,
       SYSCALL_JOIN,
       SYSCALL_EXEC,
       SYSCALL_CREATE,
       SYSCALL_OPEN,
       SYSCALL_READ,
       SYSCALL_WRITE,
       SYSCALL_SEEK,
       SYSCALL_CLOSE
  enum
    ACTIVE, ZOMBIE, FREE     -- Status of a ProcessControlBlock

  var
    readyList: List [Thread]
    currentThread: ptr to Thread
    mainThread: Thread
    idleThread: Thread
    threadsToBeDestroyed:  List [Thread]
    currentInterruptStatus: int
    processManager: ProcessManager
    threadManager: ThreadManager
    frameManager: FrameManager
    --diskDriver: DiskDriver
    --serialDriver: SerialDriver
    --fileManager: FileManager

  functions

    -- These routines are called from the Runtime.s assembly code when
    -- the corresponding interrupt/syscall occurs:

    TimerInterruptHandler ()
    DiskInterruptHandler ()
    SerialInterruptHandler ()
    IllegalInstructionHandler ()
    ArithmeticExceptionHandler ()
    AddressExceptionHandler ()
    PageInvalidExceptionHandler ()
    PageReadonlyExceptionHandler ()
    PrivilegedInstructionHandler ()
    AlignmentExceptionHandler ()
    SyscallTrapHandler (syscallCodeNum, arg1, arg2, arg3, arg4: int) returns int

    -- These routines are invoked when a kernel call is made:

    Handle_Sys_Fork () returns int
    Handle_Sys_Yield ()
    Handle_Sys_Exec (filename: ptr to array of char) returns int
    Handle_Sys_Join (processID: int) returns int
    Handle_Sys_Exit (returnStatus: int)
    Handle_Sys_Create (filename: String) returns int
    Handle_Sys_Open (filename: String) returns int
    Handle_Sys_Read (fileDesc: int, buffer: ptr to char, sizeInBytes: int) returns int
    Handle_Sys_Write (fileDesc: int, buffer: ptr to char, sizeInBytes: int) returns int
    Handle_Sys_Seek (fileDesc: int, newCurrentPos: int) returns int
    Handle_Sys_Close (fileDesc: int)
    Handle_Sys_Shutdown ()

    InitializeScheduler ()
    Run (nextThread: ptr to Thread)
    PrintReadyList ()
    ThreadStartMain ()
    ThreadFinish ()
    FatalError_ThreadVersion (errorMessage: ptr to array of char)
    SetInterruptsTo (newStatus: int) returns int
    ProcessFinish (exitStatus: int)

    -- Routines from Switch.s:

    external Switch (prevThread, nextThread: ptr to Thread)
    external ThreadStartUp ()
    external GetOldUserPCFromSystemStack () returns int
    external LoadPageTableRegs (ptbr, ptlr: int)  -- Execute "LDPTBR" and "LDPTLR"
    external SaveUserRegs (p: ptr to int)         -- Execute "readu" instructions
    external RestoreUserRegs (p: ptr to int)      -- Execute "writeu" instructions

    -- The following routine sets the "InterruptsEnabled" bit, sets the
    -- "PagingEnabled" bit, clears the "SystemMode" bit, and jumps to the
    -- address given by "initPC".
    external BecomeUserThread (initStack, initPC, initSystemStack: int)


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

  class Semaphore
    superclass Object
    fields
```

```
      count: int
      waitingThreads: List [Thread]
    methods
      Init (initialCount: int)
      Down ()
      Up ()
  endClass

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

  class Mutex
    superclass Object
    fields
      heldBy: ptr to Thread           -- Null means this mutex is unlocked.
      waitingThreads: List [Thread]
    methods
      Init ()
      Lock ()
      Unlock ()
      IsHeldByCurrentThread () returns bool
  endClass

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

  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

  ---------------  Thread  ---------------

  class Thread
    superclass Listable
    fields
      -- The first two fields are at fixed offsets, hardwired into Switch!
      regs: array [13] of int        -- Space for r2..r14
      stackTop: ptr to void          -- Space for r15 (system stack top ptr)
      name: ptr to array of char
      status: int                    -- JUST_CREATED, READY, RUNNING, BLOCKED, UNUSED
      initialFunction: ptr to function (int)    -- The thread’s "main" function
      initialArgument: int                      -- The argument to that function
      systemStack: array [SYSTEM_STACK_SIZE] of int
      isUserThread: bool
      userRegs: array [15] of int    -- Space for r1..r15
      myProcess: ptr to ProcessControlBlock
    methods
      Init (n: ptr to array of char)
      Fork (fun: ptr to function (int), arg: int)
      Yield ()
      Sleep ()
      CheckOverflow ()
      Print ()
  endClass

  -----------------------------  ThreadManager  --------------------------------
  --
  --  There is only one instance of this class, created at startup time.
  --
  class ThreadManager
    superclass Object
    fields
      threadTable: array [MAX_NUMBER_OF_PROCESSES] of Thread
      freeList: List [Thread]
      threadCheck: Mutex
      threadFree: Condition
    methods
      Init ()
      Print ()
      GetANewThread () returns ptr to Thread
      FreeThread (th: ptr to Thread)
  endClass

  -----------------------------  ProcessControlBlock  ---------------------------------
  --
  --  There are a fixed, preset number of these objects, which are created at
  --  startup and are kept in the array "ProcessManager.processTable".  When
  --  a process is started, a ProcessControlBlock is allocated from this
  --  array and the state of the process is kept in this object.
  --
  class ProcessControlBlock
    superclass Listable
    fields
      pid: int                      -- The process ID
      parentsPid: int               -- The pid of the parent of this process
      status: int                   -- ACTIVE, ZOMBIE, or FREE
      myThread: ptr to Thread       -- Each process has one thread
      exitStatus: int               -- The value passed to Sys_Exit
      addrSpace: AddrSpace          -- The logical address space
      -- fileDescriptor: array [MAX_FILES_PER_PROCESS] of ptr to OpenFile
    methods
      Init ()
      Print ()
      PrintShort ()
  endClass

  -----------------------------  ProcessManager  ---------------------------------
  --
  --  There is only one instance of this class, created at startup time.
  --
  class ProcessManager
    superclass Object
    fields
      processTable: array [MAX_NUMBER_OF_PROCESSES] of ProcessControlBlock
      processManagerLock: Mutex               -- These synchronization objects
      aProcessBecameFree: Condition           --     apply to the "freeList"
      freeList: List [ProcessControlBlock]
      aProcessDied: Condition                 -- Signalled for new ZOMBIEs
      nextPid: int
    methods
      Init ()
      Print ()
      PrintShort ()
      GetANewProcess () returns ptr to ProcessControlBlock
      FreeProcess (p: ptr to ProcessControlBlock)
      --TurnIntoZombie (p: ptr to ProcessControlBlock)
      --WaitForZombie (proc: ptr to ProcessControlBlock) returns int
  endClass

  -----------------------------  FrameManager  ---------------------------------
  --
  --  There is only one instance of this class.
  --
  class FrameManager
    superclass Object
    fields
      framesInUse: BitMap
      numberFreeFrames: int
      frameManagerLock: Mutex
      newFramesAvailable: Condition
    methods
      Init ()
      Print ()
      GetAFrame () returns int                         -- returns addr of frame
      GetNewFrames (aPageTable: ptr to AddrSpace, numFramesNeeded: int)
      ReturnAllFrames (aPageTable: ptr to AddrSpace)
  endClass

  -----------------------------  AddrSpace  ---------------------------------
  --
  --  There is one instance for every virtual address space.
  --
  class AddrSpace
    superclass Object
    fields
      numberOfPages: int
      pageTable: array [MAX_PAGES_PER_VIRT_SPACE] of int
    methods
      Init ()
      Print ()
      ExtractFrameAddr (entry: int) returns int
      ExtractUndefinedBits (entry: int) returns int
      SetFrameAddr (entry: int, frameAddr: int)
      IsDirty (entry: int) returns bool
      IsReferenced (entry: int) returns bool
      IsWritable (entry: int) returns bool
      IsValid (entry: int) returns bool
      SetDirty (entry: int)
      SetReferenced (entry: int)
      SetWritable (entry: int)
      SetValid (entry: int)
      ClearDirty (entry: int)
      ClearReferenced (entry: int)
      ClearWritable (entry: int)
      ClearValid (entry: int)
      SetToThisPageTable ()
      CopyBytesFromVirtual (kernelAddr, virtAddr, numBytes: int) returns int
      CopyBytesToVirtual (virtAddr, kernelAddr, numBytes: int) returns int
      GetStringFromVirtual (kernelAddr: String, virtAddr, maxSize: int) returns int
  endClass

endHeader
```