ASSIGNMENT-4 Amandeep Kaur

Kernel.c

```
code Kernel

  --AMANDEEP KAUR
----------------------------  ThreadManager  ---------------------------------

  behavior ThreadManager

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

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

          threadManagerLock=new Mutex
          threadManagerLock.Init()                       --Initilailize mutex lock
          aThreadBecameFree=new Condition
          aThreadBecameFree.Init()                  --Initilaize condition variable
          threadTable=new array[MAX_NUMBER_OF_PROCESSES] of Thread
{MAX_NUMBER_OF_PROCESSES of new Thread}
                                                     --initialize array of threads
          freeList=new List[T                                 --create a free list
          for i=0 to MAX_NUMBER_OF_PROCESSES-1
          threadTable[i].status = UNUSED
          threadTable[i].Init("UNUSED")      --set status of all threads to UNUSED
          freeList.AddToEnd(&threadTable[i])       --place all threads to freelist
         endFor
        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
        --
        -- This method returns a new Thread; it will wait
        -- until one is available.
        --
         var
            p:ptr to Thread
          threadManagerLock.Lock()                          --acquire lock
          while freeList.IsEmpty()                        --check if list is empty
          aThreadBecameFree.Wait(& threadManagerLock)    --wait while list is empty
          endWhile
          p=freeList.Remove()            --if not empty remove thread from freelist
          p.status = JUST_CREATED       --change status of t thread to just created
          threadManagerLock.Unlock()                        --unlock the lock
          return p                                          --return thread
        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.
        --
          threadManagerLock.Lock()                              --acquire lock
          freeList.AddToEnd(th)                       --add thread to the free list
          th.status= UNUSED                               --change status to unused
          aThreadBecameFree.Signal(& threadManagerLock) --signal any waiting thread
          threadManagerLock.Unlock()                            --unlock the lock
        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.
        --
        -- NOT IMPLEMENTED

        var
           i:int
          print("Initializing Process Manager...\n")
          processManagerLock=new Mutex
          processManagerLock.Init()                           --initialize lock
          aProcessBecameFree=new Condition
          aProcessBecameFree.Init()                     --initialize condition var
          aProcessDied=new Condition
          aProcessDied.Init()
          processTable=new array[MAX_NUMBER_OF_PROCESSES] of
ProcessControlBlock{MAX_NUMBER_OF_PROCESSES of new ProcessControlBlock}
                                                   --create new array pf processes
          freeList=new List[ProcessControlBlock]               --create freelist
          for i=0 to MAX_NUMBER_OF_PROCESSES-1
             processTable[i].status = FREE
             processTable[i].Init()
             freeList.AddToEnd(& processTable[i])
                                         --add all processes from table to freelist
         endFor
        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
            p:ptr to ProcessControlBlock
          processManagerLock.Lock()                              --acquire lock
          while freeList.IsEmpty()                     --check while list is empty
              aProcessBecameFree.Wait(& processManagerLock)      --if yes then wait
          endWhile
          p=freeList.Remove()                      --if no remove thread from head
          p.status = ACTIVE                               --change status to active
          p.pid=p.pid+1                                          --update pid
          processManagerLock.Unlock()                            --unlock lock
          return p                                               --return
        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.
        --
          -- NOT IMPLEMENTED
          processManagerLock.Lock()                                  --acquire lock
          freeList.AddToEnd(p)                                   --add to free list
          p.status= FREE                                       --set status to free
          aProcessBecameFree.Signal(& processManagerLock)
                                                       --signal any waiting process
          processManagerLock.Unlock()                             --unlock the lock
        endMethod
    endBehavior
-----------------------------  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 ()
          waitThread=new Condition
          waitThread.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)
        --
        --
        --
          var  i,f, nwaitframe,frameAddr: int
           frameManagerLock.Lock ()                              --acquire lock
           nwaitframe=nwaitframe+1        --increment number of waiting frames by 1
           if nwaitframe>1   --check if number of waiting frames are greater than 1
               waitThread.Wait(&frameManagerLock)
                                          --if yes thn add it to waiting thread list
           endIf

           while numberFreeFrames < numFramesNeeded
                                --if number of free frames are less than needed than
               newFramesAvailable.Wait(& frameManagerLock)
                                             --than thread on front list should wait
           endWhile
          for i=0 to numFramesNeeded -1
             f=framesInUse.FindZeroAndSet()            --find which frames are free
             frameAddr=PHYSICAL_ADDRESS_OF_FIRST_PAGE_FRAME + (f * PAGE_SIZE)
                                                --figure out address of first frame
             aPageTable.SetFrameAddr(i,frameAddr)
                                  --store address of frame which has been allocated
           endFor

           numberFreeFrames=numberFreeFrames-numFramesNeeded
                                                     --update number of free frames
           aPageTable.numberOfPages=numFramesNeeded
                                      --set number of pages to no. of frames needed
           nwaitframe=nwaitframe-1                                   --decrement waitframes
           waitThread.Signal(&frameManagerLock)       --signal  any thread waiting
           frameManagerLock.Unlock ()                       --unlock frame manager
        endMethod

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

      method ReturnAllFrames (aPageTable: ptr to AddrSpace)
           var i,bitNumber,frameAddr,numFramesReturned :int
           frameManagerLock.Lock ()                                 --acquire lock
           numFramesReturned=aPageTable.numberOfPages
                                                    --check how many pages returned
           for i=0 to numFramesReturned-1
           frameAddr=aPageTable.ExtractFrameAddr(i)
                                                           --find out frame address
           bitNumber= (frameAddr-PHYSICAL_ADDRESS_OF_FIRST_PAGE_FRAME)/PAGE_SIZE
           framesInUse.ClearBit(bitNumber)
                                                     --clear bit and set it to free
           endFor
           numberFreeFrames=numberFreeFrames+numFramesReturned
                                                     --adjust number of free frames
           newFramesAvailable.Signal (& frameManagerLock)
                                                           --notify waiting threads
           frameManagerLock.Unlock ()
        endMethod

    endBehavior

-----------------------------HoareCondition----------------------------

 -- 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 "Hoare" semantics, which
    -- means that in the time between a Signal() operation and the awakening
    -- and execution of the corresponding waiting thread,no other threads can
    -- run.
    --
    --
    -- This class provides the following methods:
    --    Wait(mutex)
    --         This method assumes the mutex has already been locked.
    --         It unlocks it, and goes to sleep waiting for a signal on
    --         this condition.
    --    Signal(mutex)
    --         If there are any thread waiting on this condition, this
    --         method will wake up the one and schedule it to run.
    --         The thread that is signalled will immediatly acquire the mutex and
    --         resume execution.
    --    Init()
    --         Each condition must be initialized.

behavior HoareCondition

     ----------  HoareCondition . Init  ----------

      method Init ()
          waitingThreads = new List [Thread]              --initilaize waiting list
        endMethod
      ----------  HoareCondition . Wait  ----------

      method Wait (mutex: ptr to Mutex)
          var
            oldIntStat: int
          oldIntStat = SetInterruptsTo (DISABLED)           --disable interupts
           if ! mutex.IsHeldByCurrentThread ()
            FatalError ("Attempt to wait on condition when mutex is not held")
          endIf
          mutex.Unlock ()         --unlock the lock so that it can be pass to other
          waitingThreads.AddToEnd (currentThread)      -add itself to waiting list
          currentThread.Sleep ()                        --and sleeps
          oldIntStat = SetInterruptsTo (oldIntStat)
      endMethod

      ----------  HoareCondition . Signal  ----------

      method Signal (mutex: ptr to Mutex)
          var
            oldIntStat: int
            t: ptr to Thread

          oldIntStat = SetInterruptsTo (DISABLED)               --disable interupts
            if ! mutex.IsHeldByCurrentThread ()
            FatalError ("Attempt to signal a condition when mutex is not held")
          endIf

          t = waitingThreads.Remove ()            --remove thread from waiting list
          if t
            t.status = READY                        --change status to ready
            readyList.AddToFront (t)              --add thread to front of list so
that it is the next one to run
            mutex.heldBy=t      --make sure lock to be held by this one thread only
          endIf
          mutex.Lock()                    --acquire lock
          oldIntStat = SetInterruptsTo (oldIntStat)
          endMethod

       endBehavior
```