```
########################################################################

----------------------------  Handle_Sys_Fork  --------------------------------
  function Handle_Sys_Fork () returns int
    -- Allocate and set up new Thread and ProcessControlBlock objects
    -- Make a copy of the address space
    -- Invoke Thread.Fork to start up the new processs thread
    -- return the childs pid
     var
        newPCB: ptr to ProcessControlBlock
        oldPCB: ptr to ProcessControlBlock
        newThread: ptr to Thread
        ignore: int
        i: int
        oldUserPC: int

      --print("Handle_Sys_Fork invoked! \n")
      -- Disable Interrupts
      ignore = SetInterruptsTo(DISABLED)


      -- Get new thread and PCB and initialize them
      newPCB = processManager.GetANewProcess()
      oldPCB = currentThread.myProcess
      newThread = threadManager.GetANewThread()
      -- Initialize PCB

      newPCB.parentsPid = oldPCB.pid
      -- Initialize thread (threadStatus set in GetANewThread)
      newThread.name = currentThread.name
      newThread.myProcess = newPCB
      newPCB.myThread = newThread

      -- Grab the values in the user register and store a copy
      -- in the new Thread
      SaveUserRegs(&newThread.userRegs[0])

      -- Re-enable inturrupts
      ignore = SetInterruptsTo(ENABLED)

      -- Share open files with parent
      -- ############# NEW CODE #################
      fileManager.fileManagerLock.Lock()
      for i = 0 to MAX_NUMBER_OF_OPEN_FILES-1
          newPCB.fileDescriptor[i] = oldPCB.fileDescriptor[i]
          if newPCB.fileDescriptor[i] != null
              newPCB.fileDescriptor[i].numberOfUsers = newPCB.fileDescriptor[i].numberOfUsers + 1
            endIf
        endFor
      fileManager.fileManagerLock.Unlock()
      -- ############# NEW CODE #################

      -- We then need to reset the system stack top and
      --ensure that no other threads will touch our user/new stack.
      newThread.stackTop = &(newThread.systemStack[SYSTEM_STACK_SIZE-1])

      -- Next we need to allocate the new frames for this address space
      frameManager.GetNewFrames(& newPCB.addrSpace, oldPCB.addrSpace.numberOfPages)

      -- Copy all the pages!
      for i = 0 to oldPCB.addrSpace.numberOfPages-1
```

```
          if oldPCB.addrSpace.IsWritable(i)
              newPCB.addrSpace.SetWritable(i)
          else
              newPCB.addrSpace.ClearWritable(i)
            endIf
          MemoryCopy( newPCB.addrSpace.ExtractFrameAddr(i),
                      oldPCB.addrSpace.ExtractFrameAddr(i),
                      PAGE_SIZE)
        endFor

      -- Get the User PC (That is buried in the system stack of the current Process)
      -- This value should point to the instruction following the syscall
      oldUserPC = GetOldUserPCFromSystemStack()

      --Fork a new thread and have it 'resume execution in user-land'
      newThread.Fork(ResumeChildAfterFork, oldUserPC)
      return newPCB.pid
    endFunction

  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.
      --
      var
        proc: ptr to ProcessControlBlock
        ignore: int
        i: int
        open: ptr to OpenFile

      -- Save exitStatus
      currentThread.myProcess.exitStatus = exitStatus

      -- Disable Interrupts
      ignore = SetInterruptsTo(DISABLED)

      -- Disconnect the PCB from the Thread
      proc = currentThread.myProcess
      currentThread.myProcess = null
      proc.myThread = null
      currentThread.isUserThread = false

      -- Close any open files
      -- ############# NEW CODE #################
      for i = 0 to MAX_FILES_PER_PROCESS-1
          open = proc.fileDescriptor[i]
          if open != null
              fileManager.Close(open)
            endIf
        endFor
      -- ############# NEW CODE #################

      --Re-enable interrupts
      ignore = SetInterruptsTo(ENABLED)

      -- Return all frames to the Free Pool and turn process into ZOMBIE
      frameManager.ReturnAllFrames( &proc.addrSpace)
      processManager.TurnIntoZombie(proc)

      --Terminate thread (Parent will deal with the Zombie)
      ThreadFinish()
    endFunction

----------------------------  Handle_Sys_Open  --------------------------------
  function Handle_Sys_Open (filename: ptr to array of char) returns int
```

```
      --  Gets the file name, does verification and sets the
      --  file in an empty position in the fileDescriptor array.
      --  Returns the index position in the fileDescriptor array

      --  Implementation:
      --  1. Copy filename string from virtual space to a small buffer
      --  2. Make sure the legnth of the name doesnt exceed the max size
      --  3. Locate an empty slot in fileDescriptor (if none return -1)
      --  4. Allocate OpenFile obj (return -1 if this fails)
      --  5. set the entry to point at the open File
      --  6. return index of the fileDescriptor array
      var
        numOfBytes: int
        stringStorage: array[MAX_STRING_SIZE] of char
        i: int
        pcb: ptr to ProcessControlBlock
        open: ptr to OpenFile
        holdI: int

      -- 0. Init variables
      pcb = currentThread.myProcess

      -- 1. Copy filename into a small buffer
      numOfBytes = pcb.addrSpace.GetStringFromVirtual(&stringStorage, filename asInteger,
MAX_STRING_SIZE)

      -- 2. make sure the lenth of the name doesnt exceed max (return -1)
      if stringStorage arraySize > MAX_STRING_SIZE
          return -1
        endIf

      -- 3a. locatean empty slot in fileDescriptor
      -- 4a. Allocate OpenFile obj
      open = null
      holdI = -1
      for i = 0 to MAX_FILES_PER_PROCESS - 1
        if pcb.fileDescriptor[i] == null
          holdI = i
          break
        endIf
      endFor

      open = fileManager.Open(&stringStorage)

      -- 3b. Return -1 if an empty slot is not found
      -- 4b. Return -1 if it fails opening a file
      if open == null || holdI == -1
          return -1
        endIf

      -- 5. Set the entry point at the open file
      pcb.fileDescriptor[holdI] = open

      -- 6. Return index of the file descriptr array
      return holdI

    endFunction

----------------------------  Handle_Sys_Close  ---------------------------------
  function Handle_Sys_Close (fileDesc: int)
      -- Check the argument (is it a legal array index/ point to an open file)
      --print("Handle_Sys_Close invoked!\n")
      --print("fileDes = ")
      --printInt(fileDesc)
      --print(".\n")
      var
```

```
        open: ptr to OpenFile

      -- Check to see if the index passed in is valid.
      -- Can't be greater than or equal to MAX OR less than 0
      if fileDesc >= MAX_NUMBER_OF_OPEN_FILES || fileDesc < 0
          return
        endIf

      open = currentThread.myProcess.fileDescriptor[fileDesc]
      currentThread.myProcess.fileDescriptor[fileDesc] = null

      --Make sure the file was really open. Return if can't find file
      if open == null
          return
        endIf

      fileManager.Close(open)
    endFunction


-----------------------------  Handle_Sys_Read  ---------------------------------
  function Handle_Sys_Read (fileDesc: int, buffer: ptr to char, sizeInBytes: int) returns int
      -- The idea behind sys_read, is we want to break the addresses into chunksize
      -- and read by the chunk size until we reach the sizeOfFileInBytes.
      -- We want to utilize SynchRead that will access the Read in Disk
      --print("Handle_Sys_Read invoked! \n fileDesc = ")
      --printInt(fileDesc)
      --print("\nvirt addr of buffer = ")
      --printHex(buffer asInteger)
      --print("\nsizeInBytes = ")
      --printInt(sizeInBytes)
      --print("\n")
      var
        open: ptr to OpenFile
        virtAddr: int
        virtPage: int
        offset: int
        copiedSoFar: int
        nextPosInFile: int
        thisChunksize: int
        sizeOfFile: int
        hold: bool
        destAddr: int

      -- Begin by checking fileDesc
      if fileDesc >= MAX_NUMBER_OF_OPEN_FILES || fileDesc < 0
          return -1
        endIf

      -- Check to see if sizeInBytes is negative
      if sizeInBytes < 0
          return -1
        endIf

      --Get the OpenFile
      open = currentThread.myProcess.fileDescriptor[fileDesc]
      if open == null
          return -1
        endIf

      virtAddr = buffer asInteger
      virtPage = virtAddr / PAGE_SIZE
      offset = virtAddr % PAGE_SIZE
      copiedSoFar = 0
      nextPosInFile = open.currentPos
      sizeOfFile = open.fcb.sizeOfFileInBytes
```

```
      -- Each iteration will compute the size of the next chunk and process it
      while true
          --compute size of chunk
          thisChunksize = PAGE_SIZE - offset

          if nextPosInFile + thisChunksize > sizeOfFile
              thisChunksize = sizeOfFile - nextPosInFile
            endIf

          if copiedSoFar + thisChunksize > sizeInBytes
              thisChunksize = sizeInBytes - copiedSoFar
            endIf


          -- Check to see if we're done
          if thisChunksize <= 0
              break
            endIf

          -- check for various errors
          if virtPage < 0 ||  virtPage > NUMBER_OF_PHYSICAL_PAGE_FRAMES ||
!currentThread.myProcess.addrSpace.IsValid(virtPage) ||
!currentThread.myProcess.addrSpace.IsWritable(virtPage)
            return -1
            endIf

          --Do the read:
          --Set dirtyBit for this page
          currentThread.myProcess.addrSpace.SetDirty(virtPage)
          --set referencedBit for this page
          currentThread.myProcess.addrSpace.SetReferenced(virtPage)

          destAddr = currentThread.myProcess.addrSpace.ExtractFrameAddr(virtPage) + offset
          if destAddr == 0
              return copiedSoFar
            endIf

          -- Perform read into destAddr(with next postion in file and chunksize)
          hold = fileManager.SynchRead(open, destAddr, nextPosInFile,thisChunksize)

          -- Increment
          nextPosInFile = nextPosInFile + thisChunksize
          open.currentPos = nextPosInFile
          copiedSoFar = copiedSoFar + thisChunksize
          virtPage = virtPage + 1
          offset = 0

          -- Check to see if we're done
          if copiedSoFar == sizeInBytes
              break
            endIf

        endWhile

      return copiedSoFar
    endFunction


------------------------------  Handle_Sys_Write  ---------------------------------
  function Handle_Sys_Write (fileDesc: int, buffer: ptr to char, sizeInBytes: int) returns int
      -- The idea behind sys_write, is we want to break the addresses into chunksize
      -- and write by the chunk size until we reach the sizeOfFileInBytes.
      -- We want to utilize SynchWrite that will access the Write in Disk
      --print("Handle_Sys_Write invoked!\n")
      --print("fileDesc = ")
```

```
      --printInt(fileDesc)
      --print("\nvirt addr of buffer = ")
      --printHex(buffer asInteger)
      --print("\nsizeInBytes = ")
      --printInt(sizeInBytes)
      --print("\n")
        var
        open: ptr to OpenFile
        virtAddr: int
        virtPage: int
        offset: int
        copiedSoFar: int
        nextPosInFile: int
        thisChunksize: int
        sizeOfFile: int
        hold: bool
        destAddr: int

      -- Begin by checking fileDesc
      if fileDesc >= MAX_NUMBER_OF_OPEN_FILES || fileDesc < 0
          return -1
        endIf

      -- Check to see if sizeInBytes is negative
      if sizeInBytes < 0
          return -1
        endIf

      --Get the OpenFile
      open = currentThread.myProcess.fileDescriptor[fileDesc]
      if open == null
          return -1
        endIf

      virtAddr = buffer asInteger
      virtPage = virtAddr / PAGE_SIZE
      offset = virtAddr % PAGE_SIZE
      copiedSoFar = 0
      nextPosInFile = open.currentPos
      sizeOfFile = open.fcb.sizeOfFileInBytes

      -- Each iteration will compute the size of the next chunk and process it
      while true
          --compute size of chunk
          thisChunksize = PAGE_SIZE - offset

          if nextPosInFile + thisChunksize > sizeOfFile
              thisChunksize = sizeOfFile - nextPosInFile
            endIf

          if copiedSoFar + thisChunksize > sizeInBytes
              thisChunksize = sizeInBytes - copiedSoFar
            endIf


          -- Check to see if we're done
          if thisChunksize <= 0
              break
            endIf

          -- check for various errors
          if virtPage < 0 ||  virtPage > NUMBER_OF_PHYSICAL_PAGE_FRAMES ||
!currentThread.myProcess.addrSpace.IsValid(virtPage) ||
!currentThread.myProcess.addrSpace.IsWritable(virtPage)
            return -1
            endIf
```

```
            --Do the write:
            --set referencedBit for this page
            currentThread.myProcess.addrSpace.SetReferenced(virtPage)

            destAddr = currentThread.myProcess.addrSpace.ExtractFrameAddr(virtPage) + offset
            if destAddr == 0
                return copiedSoFar
              endIf

            -- Perform read into destAddr(with next postion in file and chunksize)
            --fileManager.fileManagerLock.Unlock()
            hold = fileManager.SynchWrite(open, destAddr, nextPosInFile,thisChunksize)  --I

            -- Increment
            nextPosInFile = nextPosInFile + thisChunksize
            open.currentPos = nextPosInFile
            copiedSoFar = copiedSoFar + thisChunksize
            virtPage = virtPage + 1
            offset = 0

            -- Check to see if we're done
            if copiedSoFar == sizeInBytes
                break
              endIf

          endWhile
        return copiedSoFar
      endFunction


-----------------------------  Handle_Sys_Seek  ---------------------------------
  function Handle_Sys_Seek (fileDesc: int, newCurrentPos: int) returns int
      -- NOT IMPLEMENTED
      --print("Handle_Sys_Seek invoked!\n")
      --print("fileDesc = ")
      --printInt(fileDesc)
     -- print("\nnewCurrentPos = ")
      --printInt(newCurrentPos)
      --print("\n")
      -- Implementation:
      --    1. Lock the FileManager
      --    2. Check fileDesc and get a pointer to the Open File
      --    3. Make sure the file is open (null entry == not open)
      --    4. Deal with new curPos == -1
      --    5. Deal with new curPos < -1 (Zero is okay)
      --    6. Deal with new curPos > filesize
      --    7. Update currentPos
      --    8. return new curPos

      var
        pcb: ptr to ProcessControlBlock
        open: ptr to OpenFile


      -- 0. Initilize
      pcb = currentThread.myProcess

      -- 1. Lock the FileManager
      fileManager.fileManagerLock.Lock()

      -- 2. Check fileDesc and get a pointer to the open File
      if fileDesc > MAX_FILES_PER_PROCESS || fileDesc < 0
          fileManager.fileManagerLock.Unlock()
          return -1
        endIf
```

```
  open = pcb.fileDescriptor[fileDesc]

  if open == null
      fileManager.fileManagerLock.Unlock()
      return -1
    endIf

  -- 3. Make sure the file is open
  if open.fcb == null
      fileManager.fileManagerLock.Unlock()
      return -1
    endIf

  -- 4.  Deal with new Current Position being -1
  if newCurrentPos == -1
      newCurrentPos = open.fcb.sizeOfFileInBytes
    endIf

  --5. Deal with new current Position being < -1
  --6. Deal with new current Position being > filesize
  if newCurrentPos < -1 || newCurrentPos > open.fcb.sizeOfFileInBytes
      fileManager.fileManagerLock.Unlock()
      return -1
    endIf

  --7. update currentPos
  open.currentPos = newCurrentPos

  --8. return new curPos
  fileManager.fileManagerLock.Unlock()
  return newCurrentPos
endFunction
```