ECE 329 HW #2

In this assignment you will implement a simulated hard disk, which will be stored in a single actual file on the real hard disk. Write a class with the following interface:

```
class SimulatedHardDisk
{
public:

  // number of bytes in a sector
  enum { SHD_SECTOR_SIZE = 256 };

  // parameter indicates whether write (true) or
  //    read (false) has finished
  typedef void (*InterruptHandler)(bool write);

  // create a file on the real hard disk to
  //    contain a simulated hard disk
  // 'real_filename':  name of the real file
  // 'total_size':  size of the file in bytes
  static void FormatHardDisk(const char* real_filename,
                             int total_size);

  SimulatedHardDisk();
  ~SimulatedHardDisk();

  // tell this class which simulated hard disk to use
  bool SetDevice(const char* real_filename);

  // set the function to be called whenever a
  //    read or write is complete
  void SetInterruptHandler(InterruptHandler interrupt_handler);

public:
  // to read or write bytes, follow these steps:
  //   1.  set 'sector' to the sector number
  //   2.  set 'read' or 'write' to true
  //   3.  the sector read or write will begin
  //   4.  when the read or write is complete,
  //       the interrupt handler will be called (if set),
  //       then 'write' and 'read' will be set to false
  int sector;
  bool write, read;
  unsigned char buffer[ SHD_SECTOR_SIZE ];
};
```

The constructor should create a thread that continually monitors the ‘read’ and ‘write’ variables. When either is set, the thread initiates the read or write, transferring a single byte at a time with a sleep of 1 millisecond between transfers. The class should reside in two files: SimulatedHardDisk.h and SimulatedHardDisk.cpp.

Using your class, augment your UNIX shell with the following commands:

- *mkfs devicefile*
  creates a file system called *devicefile*. In UNIX, the command formats an existing device, but your command will simply create a file on the hard disk with the name *devicefile*. In UNIX, the raw device would normally be named something like */dev/dsk0*, so you should use a name like *devdsk0*.
- *mount devicefile dir*
  mounts the raw device *devicefile* to the directory *dir*. Since you will not be implementing a hierarchical tree structure, the only value allowed for dir is '.' (the current directory). When a device is mounted, any other device that happens to already be mounted is hidden. If *mount* is called with no arguments, then the name of the current mounted device (i.e., the name of *devicefile*) is printed. (Note: On some systems this is invoked with the *–p* option.)
- *write sec val*
  writes the value *val* to all the bytes in the sector *sec*. val should be between 0 and 255, inclusive. (No, this is not a UNIX command.)
- *read sec*
  reads the values of all the bytes in the sector *sec* and prints them to stderr. (No, this is not a UNIX command.)

The *write* and *read* commands should not return until the write or read is complete. To get comfortable with both approaches, use a spinlock for *write* and a sleep/wakeup (using a semaphore) for *read*. Be sure to put a `Sleep(1)` inside your spinlock, so that the main thread does not hog the CPU.

Hint: A clean way to implement a class that runs a thread uses two additional methods:

```
protected:
  static DWORD WINAPI ThreadProc(LPVOID lpParameter);
private:
  void MainLoop();
```

In this case, `ThreadProc` and `this` are passed to `CreateThread`. `ThreadProc` casts `lpParameter` to `SimulatedHardDisk*` which is used to call `MainLoop`.

The following library routines may be helpful:

- `WaitForSingleObject` (MSDN) -- locks a mutex; releases a semaphore
- `ReleaseMutex` (MSDN) -- unlocks a mutex
- `ReleaseSemaphore` (MSDN) -- signals a semaphore

Separately, answer the following problems in Chapter 2 of the textbook (Tanenbaum, *Modern Operating Systems*, 3rd ed.): 1, 3, 4, 5, 7, 8, 9, 10, 12, 19.