CSCE 613 – Fall 2016
Project 6: Due on 5/2/16 11:59pm CT
(bonus points if Part I is turned in by 4/18/15 11:59pm CT)

Simple Disk Device Driver and a Simple File System

In the Part I of this project, you will investigate **kernel-level device drivers** on top of a simple programmed-I/O block device (programmed-I/O LBA disk controller)[1]. You will add a layer on top of this device to support the same blocking read and write operations as the basic implementation, but **without busy waiting** in the device driver code. The user should be able to call read and write operations without worrying that the call may either return prematurely (i.e. before the disk is ready to transfer the data) or tie up the entire system waiting for the device to return.

In Part II, you write a simple file system on top of your disk support.

You need to have your scheduler (from P5) working in order to work on P6.

Part I: Blocking Drive

You will implement a device called *BlockingDisk*, which is derived from the existing low-level device *SimpleDisk*. The Device *BlockingDisk* shall implement (at least) the following interface:

```
class BlockingDisk : public SimpleDisk {
public:
    BlockingDisk(DISK_ID _disk_id, unsigned int _size);
    /* Creates a SimpleDisk device with the given size connected
     * to the MASTER or SLAVE slot of the primary ATA controller.
     */

    /* DISK OPERATIONS */
    void read(unsigned long _block_no, char * _buf);
    /* Reads 512 Bytes from the given block of the disk and copies
     * them to the given buffer. No error check!
     */
    void write(unsigned long _block_no, char * _buf);
    /* Writes 512 Bytes from the buffer to the given block
     * on the disk.
     */
};
```

Note: The thread calling the read and write operations should not block the CPU while the disk drive positions the head and reads or writes the data. Rather, the thread should give up the CPU until the operation is complete. This cannot be done completely because the read and write operations of the *SimpleDisk* use programmed I/O. The CPU keeps polling the device until the data can be read or written. You will have to find a solution that trades off quick return time from these operations with low waste of

[1] There is no need to get into the gory details of the implementation of SimpleDisk. If you are interested, however, you can find a brief overview at http://www.osdever.net/tutorials/view/lba-hdd-access-via-pio

CPU resources.

One possible approach would be to have a blocked-thread queue associated with each disk. Whenever a thread issues a read operation, it queues up on the disk queue and yields the CPU. At regular intervals (for example each time a thread resumes execution) we check the status of the disk queue and of the disk, and complete the I/O operations if possible.

Part II: Simple File System

On top of the disk device you are to implement a simple file system. In this file system, files support sequential access only, and the file name space is very simple (files are identified by unsigned integers; no multilevel directories are supported). The file system is to be implemented in classes File and FileSystem, respectively. Class File implements sequential read/write operations on an individual file:

```
class File {
    /* -- your file data structures here ... */
public:
    File(/* you may need arguments here */);
    /* Constructor for the file handle. Set the 'current position'
     * to be at the beginning of the file.
     */
    void Read(unsigned int _n, char * _buf);
    /* Read _n characters from the file starting at the current
     * location and copy them in _buf. Updates 'current position'
     * Return the number of characters read.
     */
    void Write(unsigned int _n, char * _buf);
    /* Write _n characters to the file starting at the current
     * location; if we run past the end of file, we increase the
     * size of the file as needed. Updates 'current position'.
     */
    void Reset();
    /* Set the 'current position' at the beginning of the file.
     */
    void Rewrite();
    /* Erase the content of the file. Return any freed blocks.
     * Note: This function does not delete the file!
     * It just erases its content.
     */
    BOOLEAN EoF();
    /* Is the current location for the file at the end of the file? */
};
```

[2] Not a good solution if you have threads needing to run soon.

Notice that we don't do much error checking. This is no industrial-strength code[2]. I know we all have high standards and we would like to work with a more robust API, but the end of the semester is soon, so let's focus on getting the system to run correctly in the absence of errors.

Class FileSystem controls the mapping from file name space to files and handles file allocation, free-block management on the disk, and other issues. Its interface is defined as follows:

```
class FileSystem {
    /* your file system data structures here ... */
public:
    FileSystem();
    /* Just initializes local data structures.
     * Does not connect to disk yet.
     */
    BOOLEAN Mount(SimpleDisk * _disk);
    /* Associates the file system with a disk.
     * Limit to at most one file system per disk.
     * Returns TRUE if operation successful (i.e. there is indeed
     * a file system on the disk.
     */
    static BOOLEAN Format(SimpleDisk * _disk, unsigned int _size);
    /* Wipes any file system from the given disk and installs an
     * empty file system of given size.
     */
    BOOLEAN LookupFile(int _file_id, File * _file);
    /* Find file with given id in file system.
     * If found, initialize the file object and return TRUE.
     * Otherwise, return FALSE.
     */
    BOOLEAN CreateFile(int _file_id);
    /* Create file with given id in the file system.
     * If file exists already, abort and return FALSE.
     * Otherwise, return TRUE.
    */
    BOOLEAN DeleteFile(int _file_id);
    /* Delete file with given id in the file system and free any
     * disk block occupied by the file.
     */
};
```

[2] I know you are not surprised. The interfaces for projects P2-P5 were not specified well, to put it gently.

Opportunities for Bonus Points

Recall that P6 represents 11% of your final grade. So leveraging bonus point opportunities below can have a significant impact in your grade.

Option 0 (10 bonus points): early completion of Part I.
Complete and submit working code for Part I of your project by Monday, April 18 (11:59pm CT).

OPTION 1 (10 bonus points): Support for Disk Mirroring.
Your machine is configured to have two 10MB disks connected to the ATA-0 controller (one is the MASTER, the other the SLAVE). As part of this option, you are to implement a class *MirroredDisk*, which is derived from *BlockingDisk* (easy) or from *SimpleDisk* (harder, but maybe higher-performance). Write operations are issued to both disks. Read operations are supposed to return to the caller as soon as the first of the two disks is ready to return the data.

OPTION 2 (10 bonus points): Design of a thread-safe disk and file system.
For implementation purposes, in the two parts of P5 you can assume both disks and your file system are accessed by at most one thread at a time. If multiple threads can access these items concurrently, there are plenty of opportunities for race conditions. This bonus option asks you to describe (in a design document) how you would handle concurrent operations to disk and to the file system in a safe fashion. This may require changes to the interfaces. Separate your design in three portions: (a) Disk access, (b) File System access, and (c) File access.

OPTION 3 (10 bonus points): Implementation of a thread-safe disk and file system.
For this option, you are to implement the approach proposed in Option 2. Feel free to focus on disk access only (for partial credit). Identify clearly in the design document what part you address! It is a good idea to complete Option 2 before starting Option 3.

Option 4 (10 bonus points): Using Interrupts for Concurrency.
Note: This option may expose you to all kinds of race conditions, be prepared so that your debugging goes better. You may notice all kinds of unexpected (and therefore unhandled) interrupts. At least some of these come from the disk controller. The disk may indicate through an interrupt that it needs attention. You can make use of this to cut down on the amount of polling that you do. Rather than checking the state of the disk at regular intervals, you register an interrupt handler, which wakes up the appropriate waiting thread and has it complete the I/O request.

About the new Main File (kernel.C)

The main file for this problem is very similar in nature to the one for P5. We have modified the code for some of the threads to access the blocking disk and the file system, respectively. Things you should be aware of:

• Use the macro definitions USES_DISK and USES_FILESYSTEM to control which part of the code gets exercised.
• The code in kernel.C instantiates a copy of a *SimpleDisk*. You will have to change this to a *BlockingDisk*. Other parts of the code don't need to be changed, since *BlockingDisk* is publicly derived from *SimpleDisk*.

• The portion of the code that exercises your file system code should be included in the function exercise_file_system(). In the version we provide you, there is no code there yet. Feel free to add code to convince yourself that your implementation is working. (Note: At the beginning of the execution, the disks are unformatted. You will need to format them and only then mount the file system.)
• Stating again: the kernel.C file is very similar to the one handed out in P5. It is still scheduler-free! You will have to modify it (in the same way you did for P5) to bring in your scheduler. Part I of this project will make no sense unless you have a scheduler!

About the bochs configuration

In this project, the underlying machine will have access to two hard drives, in addition to the floppy drive that you have been using until now. The configuration of these hard drives is defined in file bochsrc.bxrc (available in P5-provided-code.zip). You will notice that the file contains lines defining ata0, ata0-master, and ata0-slave. These lines defines an the ATA controller to respond to interrupt 14, and connects two hard disks, one as master and the other as slave to the controller. The disk images are given in files ”c.img” and ”d.img”, respectively, similarly to the floppy drive image in all the previous machine problems.

There is no need for you to modify further the bochsrc.bxrc file.
If you use a different emulator or a virtual machine monitor you will be using a different mechanism to mount the hard drive image as a hard drive on the virtual machine. If so, follow the documentation for your emulator or virtual machine monitor.

What to Hand In

You are to hand in a ZIP file containing the following files:

1. A design document (in PDF format) that describes your design and the implementation of Part 1 and Part 2. If you have selected any bonus options, likewise describe design and implementation for each option. **Clearly identify in your design document and in your submitted code what options you have selected, if any.**
2. Any new or modified file. Clearly identify and comment the portions of code that you have modified.

Failure to follow the handing instructions will result in lost points. In particular, expect a significant loss of points if you submit incomplete code or code that does not compile for any reason!