Machine Problem 6: Primitive Disk Device Driver

Introduction

In this machine problem you will investigate **kernel-level device drivers** on top of a simple programmed-I/O block device (programmed-I/O LBA disk controller)[1]. The given block device uses **busy waiting** to wait for I/O operations to complete. 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.

A Note about Terminology

We will follow the recommendations of the Open Source Hardware Association (OSHWA) and use the terms **master** and **dependent** to identify the two slots of the ATA disk controller. In some documentation and APIs you may find the deprecated terms **master** and **slave**.

Blocking Drive

In particular, you will implement a device called `NonBlockingDisk`, which is derived from the existing low-level device `SimpleDisk`. The Device `NonBlockingDisk` shall implement (at least) the following interface:

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

    /* DISK OPERATIONS */
    void read(unsigned long _block_no, unsigned 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, unsigned char * _buf);
    /* Writes 512 Bytes from the buffer to the given block on the disk. */
};
```

Note: The thread that calls 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 simple disk 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 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

[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`

regular intervals (for example each time a thread resumes execution[2]) we check the status of the disk queue and of the disk, and complete the I/O operations if possible.
Note about inaccurate emulation: Be aware that our emulator does not very accurately emulate disk behavior. Since the disks are emulated, and there is no actual heads moving, the IO requests may come back much sooner than expected. Do not be surprised by this. Instead, reason your way through what would have to be done if the requests actually were to take time on the disk.

Opportunities for Bonus Points

OPTION 1: Design of a thread-safe disk system. (This option carries 4 bonus points.) For implementation purposes, you can assume that a disk is to be accessed by at most one thread at a time. If multiple threads can access these items concurrently, there are plenty of opportunity for race conditions. You are to **describe** (in the design document) how you would handle concurrent operations to disk in a safe fashion. This may require a change to the interface.
OPTION 2: Implementation of a thread-safe disk system. (This option carries 6 bonus points.) For this option you are to **implement** the approach proposed in Option 1. **Note: Work on this option only after you have addressed Option 1.**
OPTION 3: Using Interrupts for Concurrency. (This option carries 8 bonus points. **Note:** You may want to go slow with this option and give preference to other options. This one may expose you to all kinds of race conditions.) In class we discussed device drivers with synchronous top-ends (where the disk operation is initiated) and bottom-ends, which are triggered by interrupts issued by the device. In this option, you design and implement a device driver that issues the disk operation in a synchronous top-end and handles the copying of data to/from buffers in a bottom end. The bottom end will be triggered by Interrupt 14. You will have to define an interrupt handler as part of your device driver, which checks the device queue for waiting requests and handles one request and releases the thread that is waiting on this rquest. The hooks for masking/enabling interrupts are provided in the code for `SimpleDisk`.

A Note about the new Main File

The main file for this MP is very similar in nature to the one for the previous MP. We have modified the code for some of the threads to access a disk.

- The code in `kernel.C` instantiates a copy of a `SimpleDisk`. You will have to change this to a `NonBlockingDisk`. Other parts of the code don't need to be changed, since `NonBlockingDisk` is publicly derived from `SimpleDisk`.
- The `kernel.C` file is very similar to the one handed out in the machine problem on threads and scheduling. It is still scheduler-free. You will have to modify it (in the same way you did for that MP) to bring in your scheduler. Since blocking threads give up the CPU, this MP will make no sense unless you have a scheduler!

A Note about the Configuration

In this MP the underlying machine will have access to a hard drive, which is represented by the disk file image `c.img`. In the `makefile` we set up qemu such that it it has an IDE controller with

[2]Not a good solution if you have urgent threads!

one disk connected to it.

The Assignment

1. Implement the Non Blocking Disk as described above. Make sure that the disk does not use busy waiting to wait until the disk comes back from an I/O operation.
2. For this, use the provided code in file `nonblocking_disk.H` and `nonblocking_disk.C`, which defines and implements class `NonBlockingDisk`. This class is publicly derived from `SimpleDisk`, and the current implementation of `NonBlockingDisk::read` and `NonBlockingDisk::write` simply call the same functions of class `SimpleDisk`. **You need to change that, and implement non-looping versions of these functions.**
3. If you have time and interest, pick one or more options and improve your Blocking Disk.

What to Hand In

You are to hand in a ZIP file, called `mp6.zip`, containing the following files:

1. A design document called `design.pdf` (in PDF format) that describes your design and the implementation of your Blocking Disk. **If you have selected any 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. All the source file and the `makefile` needed to compile the code.
3. Any modification to the provided .H file must be well motivated and documented.
4. Clearly identify and comment the portions of code that you have modified.
5. Grading of these MPs is a very tedious chore. These handin instructions are meant to mitigate the difficulty of grading, and to ensure that the grader does not overlook any of your efforts.

Note: Pay attention to the capitalization in file names. For example, if we request a file called `file.H`, we want the file name to end with a capital `H`, not a lower-case one. While Windows does not care about capitalization in file names, other operating systems do. This then causes all kinds of problems when the TA grades the submission.

Failure to follow the handin instructions will result in lost points.