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). 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 disk device to become ready.

Simple Disk Drive

You are given the implementation of a simple disk driver, which we call `SimpleDisk`. This disk driver offers the following interface:

```
class SimpleDisk {
        public:
        SimpleDisk(unsigned int _size);
        /* Creates a SimpleDisk device with the given size connected to
             the MASTER 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. */

        /* HANDLING ACCESS DURING BUSY PERIOD OF DEVICE */
        virtual bool is_busy();
        /* Is the disk currently busy? If not, disk is ready to
            transfer data from/to disk. */

        virtual void wait_while_busy() {
                while (is_busy()) { /* busy loop */; }
        }
        /* Is called during each read/write operation to wait until
            disk becomes ready. In SimpleDisk, this function simply
            loops while is_busy() returns true. In more sophisticated
            disk implementations, the thread may give up the CPU and
            return to check later. */
};
```

The detailed implementation of `SimpleDisk` contains additional, slightly tricky, code to communicate with the IDE disk controller of the disk drive using the so-called ATA protocol. Feel free to ignore the details of this implementation[1]. What is important is that the implementation of the `read` and `write` functions, after issuing the actual commands to the disk controller, need to **wait**

[1] If you are interested, however, you can find a brief, very simplified, overview at `http://www.osdever.net/tutorials/view/lba-hdd-access-via-pio`

until the disk is ready before data can be transferred between disk and memory. Before the disk is ready, the head has to be moved to the correct track, and the data transfer must be set up. This waiting for the disk to become ready is implemented in the `wait_while_busy()` function in `SimpleDisk`. Unfortunately, our `SimpleDisk` implementation uses a silly implementation of this waiting function, which simply busy loops until the disk becomes ready. During the time this function is waiting, the CPU is tied up busy looping. Since the disk is a mechanical device, it can take a long time before the disk becomes ready. During this period of time, the CPU will be tied up for a very long time by this busy looping.

Nonblocking Disk

In this machine problem, you will improve the device driver so as to not tie up the CPU while waiting for the disk to become ready. You will do this by implementing a device called `NonBlockingDisk`, which you derive from the existing low-level device `SimpleDisk`. The device `NonBlockingDisk` shall be a `SimpleDisk` with a better implementation of the `wait_while_busy()` function. As a result, the thread that calls the `read` and `write` operations should **not tie up** 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 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 thread safety: At the base level, you can assume that there is only one thread accessing the disk at any time. You therefore don't need to worry about race conditions caused by multiple threads trying to access the disk.

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.) As we stated above, at the base level you only have one thread accessing the disk. In this way, you don't need to worry about race conditions. If multiple threads can access the disk concurrently, there are plenty of opportunities for race conditions. You are to **describe** (in the design document) what kind of race conditions could occur and how you would handle concurrent operations to disk in a safe fashion. This may or may not require a change to the disk interface or to other parts of the OS. Clearly motivate any changes that you make in your design.

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.**

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

OPTION 3: Using Interrupts for Concurrency. (This option carries 8 bonus points. **Note:** Work on this option only after you have addressed Option 2. This is not an easy problem, and you may need to deal with all kinds of race conditions.) In class we discussed device drivers with synchronous top-ends (where the disk operation is initiated) and bottom-ends that are triggered by interrupts issued by the device. In other works, instead of waiting for the disk to become ready by repeatedly checking its status, the disk will indicate that it is ready by triggering Interrupt 14, which in turn triggers the execution of the bottom half. 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. Note that there may be more than one request waiting to be processed. The bottom end therefore checks the device queue for waiting requests and handles one request and releases the thread that is waiting on this request; it also issues the next request in the queue.

A Note about the new Main File

The main file for this MP is very similar in nature to the previous MP. There are a few changes that you may want to know about:

- We have modified the code so that one thread (using thread function `fun2()`) accesses the disk.
- The system components are now accessed as static members of the new class `System`. This class contains pointers to the memory pool, the system disk, and the system scheduler, which in this way can be accessed globally.

```
class System {
public:
    static const unsigned int DISK_SIZE = (10 MB);
    static MemPool* MEMORY_POOL;
    static SimpleDisk* DISK;
    static Scheduler* SCHEDULER;
};
```

- The code in `kernel.C`, instantiates a copy of a `SimpleDisk`. You will have to change this to a `NonBlockingDisk`:

```
System::DISK = new SimpleDisk(System::DISK_SIZE);
// Replace this with commented code below when you are ready!
// #define _USES_SCHEDULER_
// // The NonBlockingDisk uses a scheduler.
// System::DISK = new NonBlockingDisk(System::DISK_SIZE);
```

  The disk can now be accessed as the following example in `kernel.C` illustrates:

```
System::DISK->read(read_block, buf);
```

- `NonBlockingDisk` will need a scheduler. Everything is set up to use a scheduler (including makefile and class `System`), but you will need to turn on the creation of the scheduler and its use. You do this by defining `_USES_SCHEDULER_` as described above. Note that the provided scheduler is empty. Fill in your basic FIFO scheduler that you implemented for MP5. You also may need to update the threads to enable interrupts if you go for Option 3.

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 has an IDE controller with 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`. **You are to identify where the busy looping happens (easy) and how to eliminate – or at least significantly reduce – the busy looping.**

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.