Project 6

Due Date: 5/3/21 11:59 pm CT - early bird bonus deadline: 4/30/21

Upcoming dates of interest

Item	Deadline	Comment
P5	4/23/21	5 points if submitted by 4/21/21
P6 - disk component	5/3/21	10 points if submitted by 4/30/21
Final	5/5/21	

Be aware that work on P6, you need to have completed your P5 scheduler. With your P5 working, you will find P6 quite simple to tackle.

Contents

1 Introduction

In this project you will work with **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. This means that the CPU will be able to carry out other work while the I/O request is being completed. 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.

[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

1.1 Blocking Drive

In particular, 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 BlockingDisk device with the given size connected to the
      MASTER or SLAVE slot of the primary ATA controller.
      NOTE: We are passing the _size argument out of laziness.
      In a real system, we would infer this information from the
      disk controller. */

   /* DISK OPERATIONS */

   virtual 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! */

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

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 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 in Bochs: Be aware that Bochs 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.

1.2 Opportunities for Bonus Points

OPTION 1: Support for Disk Mirroring. (This option carries 6 bonus points.) Your machine is configured to have two 10MB disks connected to the ATA-0 controller (one is

[2]Not a good solution if you have a high-priority thread that is expected to run as soon as it is created.

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: Using Interrupts for Concurrency. (This option carries 8 bonus points.) Instead of periodically polling the disk to check if an operation has been completed, you leverage interrupts generated by the disk to detect when an operation has been concluded. (**Note:** You may want to go with caution when choosing to implement this option; it may be wiser to give preference to other bonus opportunities. This one may expose you to all kinds of race conditions.) Once you start interacting with the disk, 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.

OPTION 3: Design of a thread-safe disk system. (This option carries 5 bonus points.) In P6, we simplify the implementation by assuming that a disk is to be accessed by at most one thread at a time. In this bonus option, we remove this assumption. With multiple threads possibly accessing the disks 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. You do not need to implement your design.

OPTION 4: 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 3. **Note: Work on this option only after you have addressed Option 3.**

1.3 Testing your solution: the kernel.C file

In order for you to make sure that your environment is working in terms of writing to the simulated disks, we provide a minimalist main file: `kernel-simple-example.C` and the make file `makefile-simple-test` that builds this simplified kernel.

For an example of how the provided SimpleDisk class is used, you can look at `kernel-simple-example.C`. In order to test your system, you can build with the makefile that uses this simplified main() function:

```
make -f makefile-simple-test clean; make -f makefile-simple-test all
```

Regarding the provided `kernel.C` that you will be using to test your blocking disk implementation, the main function for this project is very similar in nature to the one for P5. We have modified the code for some of the threads to access a disk while others continue to do the same type of (useless) computation that we had in P5.

Be aware of the changes you need to make to `kernel.C` in order to exercise your code:

- 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 using the disk object don't need to be changed, since `BlockingDisk` is publicly derived from `SimpleDisk`.
- The `kernel.C` file is very similar to the one handed out in P5 to exercise threads and scheduling. It is still scheduler-free. You will have to modify it (in the same way you did for that project) to bring in your scheduler. Since your code (the *BlockingDisk* implementation) will give up the CPU when issuing the read/write disk operation, this project will make no sense unless you have a schedule to make some other thread to run when the thread doing I/O gives up the CPU.

1.4 The bochs disk 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 to book your kernel. If you look at the file `bochsrc.bxrc`, you will notice the following lines:

```
# hard disks
ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14
ata0-master: type=disk, path="c.img", cylinders=306, heads=4, spt=17
ata0-slave: type=disk, path="d.img", cylinders=306, heads=4, spt=17
```

This portion of the configuration file defines an 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 projects. **Note:** There is no need to modify the `bochsrc.bxrc` file.

The provided version of `bochsrc.bxrc` also has enabled the functionality to direct output to files. We added to `kernel.C` statements that echo the Console output to the file output, in case you need to look at them before they fly away from the visible part of the console.

If you are using VirtualBox, the setup looks for the simulated disk image files (`c.img` and `d.img`) in the current directory where you have the bochs configuration file. We tested the provided code from a local directory, i.e., we did not test the situation where your disk image file is on a "vbshared" directory, i.e., on a file system being shared with the operating system hosting the Virtual Box application.

If you use a different emulator (not bochs) or a different virtual machine monitor, you may need a different mechanism to mount the hard drive image as a hard drive on your simulated environment. If so, follow the documentation for your emulator or virtual machine monitor.

1.5 The Assignment

1. Bring your P5 `scheduler.C` to your P6 project. If you made changes in other files, you will have to apply them on the P6 project source code tree.

2. Implement the Blocking Disk as described above. The main challenge (and it is not hard at all, compared to all you have done so far in the semester) is to make sure that the thread invoking the disk I/O operation passes the CPU to another thread instead of keeping the CPU busy waiting for the disk operation to be completed.
   Again: make sure that the disk does not use busy waiting to wait until the disk comes back from an I/O operation. It needs to yield the CPU to another thread.

3. For this purpose, use the provided code in file `blocking_disk.H` and `blocking_disk.C`, which defines and implements class `BlockingDisk`. This class is publicly derived from `SimpleDisk`, and the current implementation of `BlockingDisk::read` and `BlockingDisk::write` simply call the same functions of class `SimpleDisk`.

   You need to change that to implement non-looping versions of these functions.

4. If you have time and interest, pick one or more bonus options and improve your Blocking Disk.

1.6 What to Hand In

You are to hand in a ZIP file, called `p6.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. A pair of files, called `blocking_disk.H` and `blocking_disk.C`, which contain the definition and implementation of class `BlockingDisk`. Any modification to the provided .H file must be well motivated and documented.

3. Any new or modified file. Clearly identify and comment the portions of code that you have modified.

Grading of these projects 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.

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

Naturally, expect a significant loss of points if you submit incomplete code or code that does not compile for any reason!

2 Document Release Log

- Version 2021-A.0: initial release on 4/22/21