A PThread-based Disk Driver

Due at 11:59pm on Monday, 20 November 2017

1 General Overview

A piece of code is required which will function as a driver for a simple disk device. Calls will be made to this software component to pass sectors[1] of data to/from the disk. Sectors read from the disk will be passed to appropriate waiting callers. The emphasis is on the concurrency, not on disk scheduling issues.

The core of your solution will be one or more threads, which move sectors of data between queues or buffers (amongst other things). You will be programming in C, and using the PThread library. PThreads enable you to dynamically create threads and provides support for both mutual exclusion locks and for condition variables (which support a simple wait/signal mechanism and allow a timeout on waits).

A tutorial on Pthreads, together with sample code for a generic BoundedBuffer, is available on Canvas. If you have not already read through the Pthreads tutorial, you should do so immediately.

2 Detailed Problem

You are to write a piece of code which provides the central functionality of a disk device driver. The ADT you are producing will be called DiskDriver, and you will make calls on to an instance of the DiskDevice ADT (provided by the test harness). Calls will be made to your code from the application level, as well.

Your code will pass around pointers to SectorDescriptors; this is an ADT about which you need to know almost nothing. One of the components of the system is a SectorDescriptor constructor; this will be handed a region of memory (actually a void * pointer and a length) and it divides that into pieces, each the right size to hold a SectorDescriptor. The SectorDescriptors are then passed to another component, the FreeSectorDescriptorStore. The FreeSectorDescriptorStore is an unbounded container, which will be used by your code and the test harness as a place from which to acquire SectorDescriptors. When they are no longer used, the SectorDescriptors must be returned to the FreeSectorDescriptorStore.

You are allowed to use an existing piece of code in the test harness for the FreeSectorDescriptorStore, along with other components of the test harness. Similarly you may use the BoundedBuffer provided on Canvas if that is helpful within your solution. *All other code must be your own work; plagiarism detection will be undertaken to check this is the case.*

The test harness includes fake applications, which will acquire SectorDescriptors from the store, initialize their contents and then pass them to your code: the fake applications make two kinds of requests: queue the write of a sector to disk, or queue up a read for a sector from the disk; after making each kind of request, the application makes a subsequent call to either 1) determine that the write was successful, or 2) retrieve the sector after it has been read.

[1] This handout refers to sectors and blocks; they are simply two terms for the same thing, a fixed-size chunk of space on the disk device.

The test harness also includes a fake DiskDevice to which you can pass SectorDescriptors for dispatch. The complication is that the DiskDevice can only write one sector at a time, and is quite slow in doing so; therefore, you may have to queue up other requests. An application thread does not have to wait for its sector to be sent to the disk. Once it has handed it to your code, it is allowed to engage in other activity, returning to your driver to determine that the write has been successful when convenient for the application. This means you are likely to need dedicated threads within your code to pass sectors to the disk device.

Once a sector has been sent to the disk, your code should pass the descriptor back to the FreeSectorDescriptorStore.

The DiskDevice can simultaneously read one sector at a time from the disk. This means that you should always have a read outstanding (if one is available) on the DiskDevice. To maximize the throughput of the DiskDevice, your processing of a recently read sector must be minimal, handing it over to another (buffered?) part of your code for retrieval by the application before initiating another read (if one is available). Again, the application does not wait for its sector to be retrieved from the disk; it will eventually return to the driver to determine that the read has been successful and to access the data read from the disk.

Sectors that have been read will be passed to the fake application layer, and eventually the sector descriptors will be returned to the free sector descriptor store by the applications.

Because this code is supposed to live inside the OS, you are required to use bounded sized data structures within the parts of your code that handle the passing of sector descriptors between the applications and the disk device. It's possible that application threads will block, either because they are awaiting incoming sectors, because they are waiting for a write to complete, because the buffers for outgoing/incoming sectors are full (as those data structures are supposed to be bounded), or because there are no free sector descriptors available.

In summary, you have applications that run at their own pace, and a disk device that runs at a probably different pace and for which the requests must be serialized. Your disk driver has to bridge between the two, despite the speed mismatch. You are to use pthreads and bounded data structures to provide this bridging function to implement the spec in diskdriver.h.

The starting files for this exercise are available from Canvas.

You should start the design of your solution now, sketching out what you are trying to achieve and the rough code needed to do it. Your sketch will most likely take the form of an interaction and/or collaboration diagram. Your graduate TA and I will be happy to review your design after you have completed it.

3 Background information (taken from the .h files)

The test harness application threads are distinguished by their PIDs:

```
typedef unsigned long PID;
#define MAX_PID 10
/*
 * A PID is used to distinguish between the different applications
 * It is implemented as an unsigned int, but is guaranteed not to
 * exceed MAX_PID.
 */
```

The actual block number to read or write on the disk is indicated by its Block:

```
/*
```

```
 * Define the data type for a block number on the disk
 */
typedef unsigned long Block;
```

The disk device is "full duplex" – i.e., it can write a sector and read a sector at the same time. Writing a sector entails the calling thread simply invoking a method in the device, with that thread blocking until the sector has been successfully written or until an error is detected. Reading a sector entails making a different call to the device, with that thread blocking until the sector has been successfully read or until an error is detected.

A `SectorDescripter` contains the `PID` of the requesting application, the `Block` of the sector of interest, and the buffer containing the data to write or a buffer into which the sector data is read.

The disk device supports the following calls – your code may invoke these:

```
/*
 * write sector to disk, returns 1 if successful, 0 if unsuccessful
 * this may take a substantial amount of time to return
 * if unsuccessful, you should return an indication of this lack of
 * success to the application
 */
int write_sector(DiskDevice *dd, SectorDescriptor *sd);

/*
 * read sector from disk, returns 1 if successful, 0 if unsuccessful
 * this may take a substantial amount of time to return
 * if unsuccessful, you should return an indication of this lack of
 * success to the application
 */
int read_sector(DiskDevice *dd, SectorDescriptor *sd);
```

The fake application threads may make the following calls to your code – you must implement them.

```
/*
 * the following calls are used to write a sector to the disk;
 * the nonblocking call must return promptly, returning 1 if successful at
 * queueing up the write, 0 if not (in case internal buffers are full)
 * the blocking call will usually return promptly, but there may be
 * a delay while it waits for space in your buffers.
 * neither call should delay until the sector is actually written to the disk
 * for a successful nonblocking call and for the blocking call, a voucher is
 * returned that is required to determine the success/failure of the write
 */
void blocking_write_sector(SectorDescriptor *sd, Voucher **v);
int nonblocking_write_sector(SectorDescriptor *sd, Voucher **v);

/*
 * the following calls are used to initiate the read of a sector from the disk
 * the nonblocking call must return promptly, returning 1 if successful at
 * queueing up the read, 0 if not (in case internal buffers are full)
 * the blocking call will usually return promptly, but there may be
 * a delay while it waits for space in your buffers.
 * neither call should delay until the sector is actually read from the disk
 * for a successful nonblocking call and for the blocking call, a voucher is
 * returned that is required to collect the sector after the read completes.
 */
void blocking_read_sector(SectorDescriptor *sd, Voucher **v);
int nonblocking_read_sector(SectorDescriptor *sd, Voucher **v);

/*
 * the following call is used to retrieve the status of the read or write
 * the return value is 1 if successful, 0 if not
 * the calling application is blocked until the read/write has completed
 * if a successful read, the associated SectorDescriptor is returned in *sd
 */
int redeem_voucher(Voucher *v, SectorDescriptor **sd);
```

There are also some initialization calls, the first is for you to implement, the second is one you can use:

```
void init_disk_driver( DiskDevice *dd, void * mem_start,
        unsigned long mem_length, FreeSectorDescriptorStore **fsds);
/*
 * Called before any other methods[2], to allow you to initialize
 * data structures and start any internal threads.
 * Arguments:
 *      dd: the DiskDevice that you must drive,
 *      mem_start, mem_length: some memory for SectorDescriptors
 *      fpds: You hand back a FreeSectorDescriptorStore into
 *      which SectorDescriptors built from the memory
 *      described in mem_start and mem_length have been placed
 */

void init_sector_descriptor(SectorDescriptor *sd);
/*
 * Resets the sector descriptor to be empty.  Used by fake applications
*/
```

You will need to use the FreeSectorDescriptorStore facilities. You can create a FreeSectorDescriptorStore with:

```
FreeSectorDescriptorStore *create_fsds(void);
```

populate it with SectorDescriptors created from a memory range using:

```
void create_free_sectors_descriptors(FreeSectorDescriptorStore *fsds,
                void *mem_start, unsigned long mem_length);
```

and then acquire and release SectorDescriptors using:

[2] When an operating system boots up, it typically invokes an initialization routine in each kernel module to enable that module to prepare for operation. It invokes these initialization routines before it enables any applications to be launched.

```
void blocking_get_sd(FreeSectorDescriptorStore *fsds, SectorDescriptor **sd);
int nonblocking_get_sd(FreeSectorDescriptorStore *fsds, SectorDescriptor **sd);

void blocking_put_sd(FreeSectorDescriptorStore *fsds, SectorDescriptor *sd);
int nonblocking_put_sd(FreeSectorDescriptorStore*fsds, SectorDescriptor*sd);

/*
 * As usual, the blocking versions only return when they succeed.
 * The nonblocking versions return 1 if they worked, 0 otherwise.
 * The _get_ functions set their final arg if they succeed.
 */
```

Finally, given a SectorDescriptor, you can access the embedded PID and Block fields using:

```
PID sector_descriptor_get_pid(SectorDescriptor *sd);
Block sector_descriptor_get_block(SectorDescriptor *sd);
```

4 Pseudocode for Your Driver

```
/* any global variables required for use by your threads and your driver routines */

/* definition[s] of function[s] required for your thread[s] */

void init_disk_driver(DiskDevice *dd, void *mem_start,
                      unsigned long mem_length, FreeSectorDescriptorStore **fsds)  {
   /* create Free Sector Descriptor Store */
   /* load FSDS with packet descriptors constructed from mem_start/mem_length */
   /* create any buffers required by your thread[s] */
   /* create any threads you require for your implementation */
   /* return the FSDS to the code that called you */
}

void blocking_write_sector(SectorDescriptor *sd, Voucher **v) {
   /* queue up sector descriptor for writing */
   /* return a Voucher through *v for eventual synchronization by application */
   /* do not return until it has been successfully queued */
}

int nonblocking_write_sector(SectorDescriptor *sd, Voucher **v) {
   /* if you are able to queue up sector descriptor immediately
   /*     return a Voucher through *v and return 1 */
   /* otherwise, return 0 */
}

void blocking_read_sector(SectorDescriptor *sd, Voucher **v) {
   /* queue up sector descriptor for reading */
   /* return a Voucher through *v for eventual synchronization by application */
   /* do not return until it has been successfully queued */
}
```

```
int nonblocking_read_sector(SectorDescriptor *sd, Voucher **v) {
   /* if you are able to queue up sector descriptor immediately
   /*     return a Voucher through *v and return 1 */
   /* otherwise, return 0 */
}

/*
 * the following call is used to retrieve the status of the read or write
 * the return value is 1 if successful, 0 if not
 * the calling application is blocked until the read/write has completed
 * if a successful read, the associated SectorDescriptor is returned in *sd
 */
int redeem_voucher(Voucher *v, SectorDescriptor **sd);
```

5 Developing Your Code

The must develop your code in Linux running inside the virtual machine image provided to you; all of the object files for the testharness supplied to you have been compiled in this way. This also gives you the benefit of taking snapshots of system state right before you do something potentially risky or hazardous, so that if something goes horribly awry you can easily roll back to a safe state.

You should use your Bitbucket GIT repositories for keeping track of your programming work. As a reference, you can perform the command line steps below to create a new project directory and upload it to your uoregon-cis415 repository.

```
% cd /path/to/your/uoregon-cis415
% mkdir project2
% echo "This is a test file." >project2/testFile.txt
% git add project2
% git commit –m "Initial commit of project2"
% git push –u origin master
```

Any subsequent changes or additions can be saved via the add, commit, and push commands.

6 Helping your Classmate

This is an individual assignment. You should be reading the manuals, hunting for information, and learning those things that enable you to do the project. However, it is important for everyone to make progress and hopefully obtain the same level of knowledge by the project's end. If you get stuck, seek out help to get unstuck. Sometimes just having another pair of eyes looking at your code is all you need. If you cannot obtain help from the TA or the lecturer, it is possible that a classmate can be of assistance.

This does not mean pair programming or any other form of collusion. We will be checking for collusion; better to turn in an incomplete solution that is your own

than a copy of someone else's work. We have very good tools for detecting collusion.

In your status report on the project, you should provide the names of classmates that you have assisted, with an indication of the type of help you provided. You should also indicate the names of classmates from whom you have received help, and the nature of that assistance.

7 Testing your driver

The `Makefile` provided builds a program named `my_demo` from your diskdriver.o and the rest of the object files provided in the starting archive. A working version of the testharness built from a correct disk driver is also in the archive, named `demo`. Both `demo` and `my_demo` take a single, optional argument – the number of seconds to run the simulation. If you do not specify the optional argument, then the simulation will run for 2 minutes (120 seconds). I suggest that while testing your diskdriver implementation, you use the following command line to bash:

```
$ ./mydemo 30 2>&1 | tee log
```

This will cause your simulation to run for 30 seconds, and cause everything written to `stdout` or `stderr` to be piped to `tee`, which will save all such output to a file named `log`, as well as display it on your screen while it is working.

The previous sections have described constraints on the workings of your driver. Here I list diagnostics that will be displayed by the simulation that indicate that you have violated one or more of these constraints.

```
BUG: Two or more parallel calls to write a sector
```
The diskdevice can only handle one write at a time; you have asked it to do 2 or more.

```
BUG: Two or more parallel calls to read a sector
```
The diskdevice can only handle one read at a time; you have asked it to do 2 or more.

8 Submission[3]

You will submit your solutions electronically by uploading a gzipped tar archive via Canvas.

Your TGZ archive should be named "<duckid>-project2.tgz", where "<duckid>" is your duckid. It should contain your "diskdriver.c" and a document named "report.pdf" or "report.txt", describing the state of your solution, and documenting anything of which we should be aware when marking your submission. If your solution depends upon any

[3] ***100% penalty will be assessed (i.e. you will receive 0 points) if you do not follow these submission instructions. See the handout for Project 0 if you do not remember how to create the gzipped tar archive for submission.***

other files ***that you have created***, these must also be provided in the archive, along with a modified Makefile that builds your "mydemo" from the testharness files and your .o files. Note that ease of marking dictates that you ***not*** create additional files – i.e. if you have other helper routines needed in your diskdriver, these should be static functions declared in diskdriver.c. Additionally, I do not want to see ***any*** other files in your archive.

Section 2 above indicates that you should prepare a sketch of your design; the most useful form of such a sketch is as an interaction and/or a collaboration diagram. If you submit a PDF file for your report, you should include the diagram in your report; if you submit a TXT file for your report, you should include design.jpg in your submitted archive. In either case, it can be hand-drawn diagram that is then scanned to a JPEG file.

These files should be contained in a folder named "<duckid>". Thus, if you upload "jsventek-project2.tgz", then we should see the following when we execute the following command:

```
% tar –ztvf jsventek-project2.tgz
-rw-rw-r-- jsventek/group  5125 2015-03-30 16:37 jsventek/diskdriver.c
-rw-rw-r-- jsventek/group   527 2015-03-30 16:30 jsventek/report.txt
-rw-rw-r-- jsventek/group 12345 2015-03-30 16:24 jsventek/design.jpg
```

Each of your source files must start with an "authorship statement", contained in C comments, as follows:

- state your name, your login, and the title of the assignment (CIS 415 Project 2)
- state either "This is my own work." or "This is my own work except that …", as appropriate.

Marking Scheme for CIS415 Project 2

Design (30)

- design diagram showing dataflows and interactions in your system (16)
- appropriate sizing of bounded buffers/other data structures (7)
- explanation of blocking behaviour matching the requirements (7)

Implementation (70)

- honest statement of the state of the solution (10)
- ***workable*** implementation (30)
    - appropriate synchronization (6)
    - appropriate number and initialization of threads (4)
    - appropriate return of SectorDescriptors (4)
    - low complexity search for next SectorDescriptor associated with the redeeming thread (6)
    - appropriate initialization of data structures (5)
    - sufficient commentary in the code (5)
- ***working*** implementation (20)
- excellent commentary (10)

TOTAL (100)