Assignment 2 - Device Driver

Due Jul 1, 2019 by 11:59pm **Points** 100 **Submitting** a file upload
File Types tar.gz and tgz
Available Jun 4, 2019 at 12am - Jul 4, 2019 at 11:59pm about 1 month

This assignment was locked Jul 4, 2019 at 11:59pm.

Assignment #2 - Block Storage Filesystem

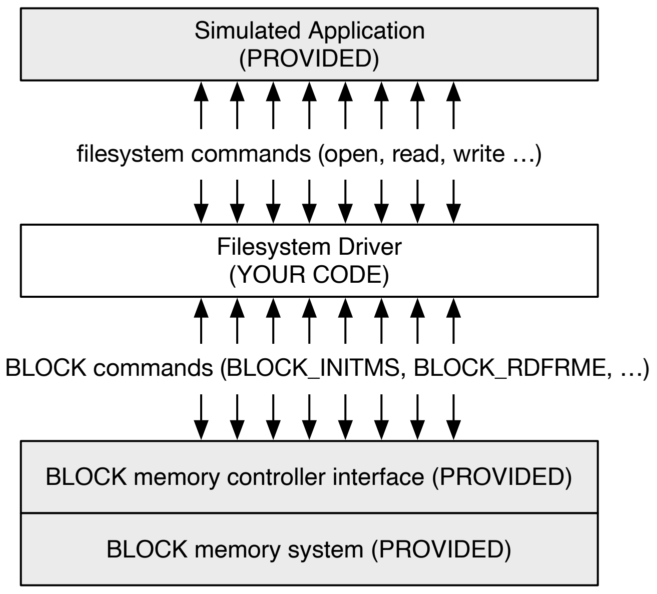
CMPSC311 - Introduction to Systems Programming

Summer 2019 - Prof. McDaniel

Due date: June 24, 2019 (11:59pm UTC-4)

All remaining assignments for this class are based on the creation and extension of a user-space device driver for a in memory filesystem that is built on top of a block storage device. At the highest level, you will translate file system commands into memory frame level memory operations (see memory interface specification below). The file system commands include `open`, `read`, `write`, `seek` and `close` for files that are written to your file system driver. These operations perform the same as the normal UNIX I/O operations, with the caveat that they direct file contents to the block storage device instead of the host filesystem. The arrangement of software is as follows:

System and Project Overview


Simulated Application
(PROVIDED)
filesystem commands (open, read, write ...)
Filesystem Driver
(YOUR CODE)
BLOCK commands (BLOCK_INITMS, BLOCK_RDFRME, ...)
BLOCK memory controller interface (PROVIDED)
BLOCK memory system (PROVIDED)

You are to write a basic device driver that will sit between a virtual application and virtualized hardware. The application makes use of the abstraction you will provide called Block Memory System (BLOCK). The design of the system is shown in the figure to the right:

Described in detail below, we can see three layers. The BLOCK application is provided to you and will call your device driver functions with the basic UNIX file operations (`open`, ...). You are to write the device driver code to implement the file operations. Your code will communicate with a virtual controller by sending opcodes and data over an I/O bus.

All of the code for application (called the simulator) and BLOCK memory system is given to you. Numerous utility functions are also provided to help debug and test your program, as well as create readable output. Sample workloads have been generated and will be used to extensively test the program. Students that make use of all of these tools (which take a bit of time up front to figure out) will find that the later assignments will become much easier to complete.

The Block Memory System Driver (BLOCK)

You are to write the driver that will implement the the basic UNIX file interface using the memory system. You will write code to implement the filesystem, and make several design decisions that will determine the structure and performance of the driver. In essence you will write code for the read, write, open, close and other high level filesystem functions. Each function will translate the call into low-level operations on the device (see below).

The functions you will implement are:

- `int32_t block_poweron(void);` - This function will initialize the BLOCK interface basic filesystem. To do this you will execute the init BLOCK opcode and zero all of the memory locations. You will also need to setup your internal data structures that track the state of the filesystem.
- `int32_t block_poweroff(void);` - This function will show down the BLOCK interface basic filesystem. To do this you will execute the shutdown BLOCK opcode and close all of the files. You will also need to cleanup your internal data structures that track the state of the filesystem
- `int16_t block_open(char *path);` - This function will open a file (named path) in the filesystem. If the file does not exist, it should be created and set to zero length. If it does exist, it should be opened and its read/write postion should be set to the first byte. Note that there are no subdirectories in the filesystem, just files (so you can treat the path as a filename). The function should return a unique file handle used for subsequent operations or -1 if a failure occurs.
- `int16_t block_close(int16_t fd);` - This function closes the file referenced by the file handle that was previously open. The function should fail (and return -1) if the file handle is bad or the file was not previously open.
- `int32_t block_read(int16_t fd, void *buf, int32_t count);` - This function should read `count` bytes from the file referenced by the file handle at the current position. Note that if there are not enough bytes left in the file, the function should read to the end of the file and return the number of bytes read. If there are enough bytes to fulfill the read, the function should return `count`. The function should fail (and return -1) if the file handle is bad or the file was not previously open.
- `int32_t block_write(int16_t fd, void *buf, int32_t count);` - The function should write `count` bytes into the file referenced by the file handle. If the write goes beyond the end of the file the size should be increased. The function should always return the number of bytes written, e.g., `count`. The function should fail (and return -1) if the file handle is bad or the file was not previously open.
- `int32_t block_seek(int16_t fd, uint32_t loc);` - The function should set the current position into the file to `loc`, where 0 is the first byte in the file. The function should fail (and return -1) if the loc is beyond the end of the file, the file handle is bad or the file was not previously open.

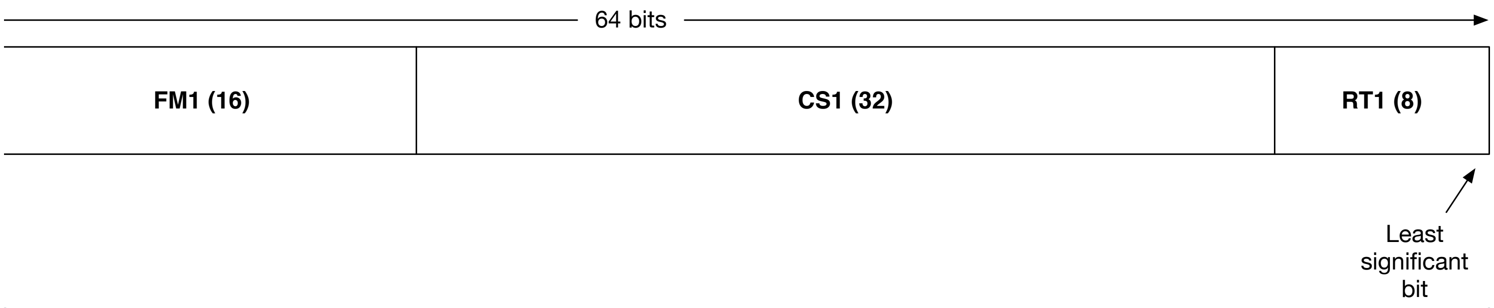
The key to this assignment if figuring out what you need to do to implement these functions. You are specifically not given guidance on how to do it. You need to (a) maintain information about current files in the file system in memory, (b) allocate parts of the memory system to place data, (c) copy data into and out of the memory system as needed to serve reads and writes. How you do this is up to you, but think carefull about it before beginning to code. What is important is that the code you write will be built upon the whole semester.

The Block Memory System (BLOCK)

You will implement your driver on top of the BLOCK memory system (which is referred to throughout simply as the BLOCK). The BLOCK consists of one continuous disk that contains many frames. Each frame is a fixed byte sized memory block. Some key facts of the system include (see `block_controller.h` for definitions):

- The disk contains `BLOCK_BLOCK_SIZE` frames, each of which is numbered from `0` to `BLOCK_BLOCK_SIZE-1`.
- A frame is `BLOCK_FRAME_SIZE` bytes.

You communicate with the memory system by sending code through a set of *packed registers*. These registers are set within a 64-bit value to encode the opcode and arguments to the hardware device. The opcode is laid out as follows:



```
 Bits    Register (note that top is bit 0)
------   -----------------------------------
  0-7 - KY1 (Key Register 1)
 8-23 - FM1 (Frame Register 2)
24-55 - CS1 (Checksum register 1)
56-63 - RT1 (Return code register 1)
```

The following opcodes define how you interact with the controller. Note that any register not explicitly listed in the specifications below should always be zero. If the `frame` argument is not specified, then it should be passed as NULL.

The checksum of the frame as described below

Register	Request Value	Response Value
BLOCK_OP_INITMS - Initialize the memory system		

Register: **KY1**	BLOCK_OP_INITMS	BLOCK_OP_INITMS
Register: **RT**	N/A	0 if successful, -1 if failure

BLOCK_OP_BZERO - zero the entire block system

Register: **KY1**	BLOCK_OP_BZERO	BLOCK_OP_BZERO
Register: **RT**	N/A	0 if successful, -1 if failure

BLOCK_OP_RDFRME - read a frame from the block system

Register: **KY1**	BLOCK_OP_RDFRME	BLOCK_OP_RDFRME
Register: **RT**	N/A	0 if successful, -1 if failure
Register: **FM1**	frame number to read from	N/A
Register: **CS1**	N/A	The checksum of the frame as described below

BLOCK_OP_WRFRME - write a frame to the block device

Register: **KY1**	BLOCK_OP_WRFRME	BLOCK_OP_WRFRME
Register: **RT**	N/A	0 if successful, -1 if failure, 2 on incorrect checksum
Register: **FM1**	frame number to write to	N/A
Register: **CS1**	The checksum of the frame as described below	N/A

BLOCK_OP_POWOFF - power off the memory system

Register: **KY1**	BLOCK_OP_POWOFF	BLOCK_OP_POWOFF
Register: **RT**	N/A	0 if successful, -1 if failure

To execute an opcode, create a 64 bit value (uint64_t) and pass it any needed buffers to the bus function defined in `block_controller.h`:

`BlockXferRegister block_io_bus(BlockXferRegister regstate, void *buf)`

The function returns packed register values with as listed in the "Response Value" above.

Block Checksums

The `BLOCK_OP_RDFRME` and `BLOCK_OP_WRFRME` operations make use of the `CS1` checksum register. When writing, the checksum must be populated in the register, and when reading the checksum must be checked to ensure frame integrity. Checksums are computed by the `generate_md5_signature` method in `cmpsc311_util.h`, with `sigsz` of 4. These 4 bytes are then copied or compared directly against the 4 bytes in `CS1`.

Frames are corrupted with random probability 1/128. When a frame checksum does not validate or the driver returns BLOCK_RET_CHECKSUM_ERROR, the command should be retried indefinitely until it succeeds.

Instructions

1. Login to your virtual machine. From your virtual machine, download the starter source code provided for this assignment **here**.
2. You'll need several packages to compile and run the source code. Run the following command in the terminal to install these:

   ```
   sudo apt update && sudo apt install -y build-essential libgcrypt20-dev libcurl4-gnutls-dev
   ```

3. Create a directory for your assignments and copy the file into it. Change into that directory.

   ```
   % mkdir cmpsc311
   % cp assign2-starter.tgz cmpsc311
   % cd cmpsc311
   % tar xvzf assign2-starter.tgz
   ```

   Once unpacked, you will have the starter files in the assign2, including the source code, libraries, and a Makefile. There is also a subdirectory containing workload files.

4. You are to complete the `block_drver.c` functions defined above. Note that you may need to create additional supporting functions within the same file. Include functional prototypes for these functions at the top of the file.
5. Add comments to all of your files stating what the code is doing. Fill out the comment function header for each function you are defining in the code. A sample header you can copy for this purpose is provided for the main function in the code.
6. To test your program with these provided workload files, run the code specifying a workload file as follows:

   ```
   ./block_sim -v workload/cmpsc311-sum19-assign2-workload.txt
   ```

Note that you don't necessarily have to use the -v option, but it provides a lot of debugging information that is helpful.

If the program completes successfully, the following should be displayed as the last log entry:**BLOCK simulation: all tests successful!!!.**

To turn in:

1. Create a tarball file containing the `assign2` directory, source code and build files as completed above. Submit the tarball through Canvas by the assignment deadline (11:59pm of the day of the assignment). The tarball should be named `LASTNAME-PSUEMAILID-assign2.tgz`, where LASTNAME is your last name in all capital letters and PSUEMAILID is your PSU email address without the "@psu.edu". For example, the professor was submitting a homework, he would call the file `MCDANIEL-pdm12-assign2.tgz`. Any file that is incorrectly named, has the incorrect directory structure, or has misnamed files, will be assessed a one day late penalty.
2. Before sending the tarball, test it using the following commands (in a temporary directory -- NOT the directory you used to develop the code):

   >

   ```
   % tar xvzf LASTNAME-PSUEMAILID-assign2.tgz
   % cd assign2
   % make
   ... (TEST THE PROGRAM)
   ```

Note: Like all assignments in this class you are prohibited from copying any content from the Internet or discussing, sharing ideas, code, configuration, text or anything else or getting help from anyone in or outside of the class. Consulting online sources is acceptable, but under no circumstances should *anything* be copied. Failure to abide by this requirement will result dismissal from the class as described in our course syllabus. The assignment environment may be logged for debugging and other course uses.
