Assignment 3 - Persistent Block Storage

Re-submit Assignment

Due Jul 29, 2019 by 11:59pm **Points** 150 **Submitting** a file upload
File Types tar.gz and tgz **Available** after Jul 4, 2019 at 11:59pm

Assignment #3 - Persistent Block Storage

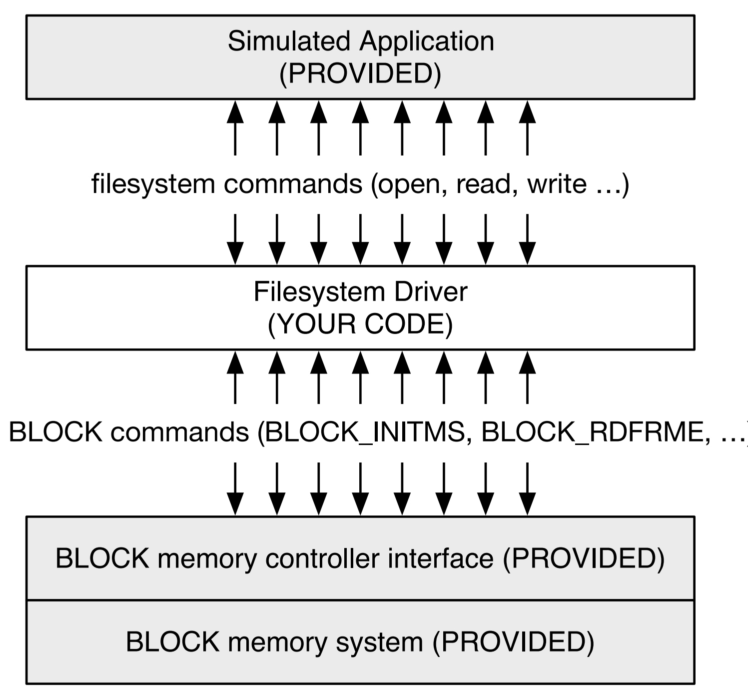
CMPSC311 - Introduction to Systems Programming

Summer 2019 - Prof. McDaniel

Due date: July 15, 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



In the previous assignment, you wrote a basic device driver that sits between a virtual application and virtualized hardware. The application makes use of the abstraction you will provide called Block Memory System (BLOCK). As a reminder, the design of the system is shown in the figure to the right:

In this assignment, you will have to implement a new functionality for the driver you previoulsy wrote: persistence across sequential runs.

What this means in practice is that your code should now be able to "remember" the state of the BLOCK device after it was powered off and on again. The next section will go into more details on how to implement this.

All the code you need for this assignement is already in your possession. The only new file you will need is a new workload that is provided below. This workload whould be run after the first workload has been successfully run. If both workloads pass successfully, you most likey implemented the persistence correctly.

How to implement the persistence

In the code that was given to you in the previous assignment, the BLOCK device is already configured to write its contents to a file (called `block_memsys.bck`) when it is powered off (i.e. it receives the `BLOCK_OP_POWOFF` opcode), and read it back when powered on (i.e. when it receives the `BLOCK_OP_INITMS` opcode. This means that when you initialize the device, its memory state (the contents of its frames) is the same as when you last powered it off.

The steps you then have to take to implement persistence are:

- Disable the zero-ing of the memory in `block_poweron()`.
- Determine how you want to store the files metadata in the device, and make sure the frames you want to use for this purpose cannot be used to store file data by you other functions.
- In your `block_poweroff()` function, add a step that will write the metadata of your files to these frames on the device.
- Add an extra step in your `block_poweron()` function that will read the frames that contain the files metadata you have stored, and use it to populate your data structures you use to keep track of your files. Keep in mind that all files start closed (i.e. you should not have to create any file handle at this point).

The key to this assignment is figuring out how you want to store your metadata to the device, in a way that can be reliably read later on. How you do this is up to you, but think carefull about it before beginning to code. Keep in mind that the next assignment will once again use your code.

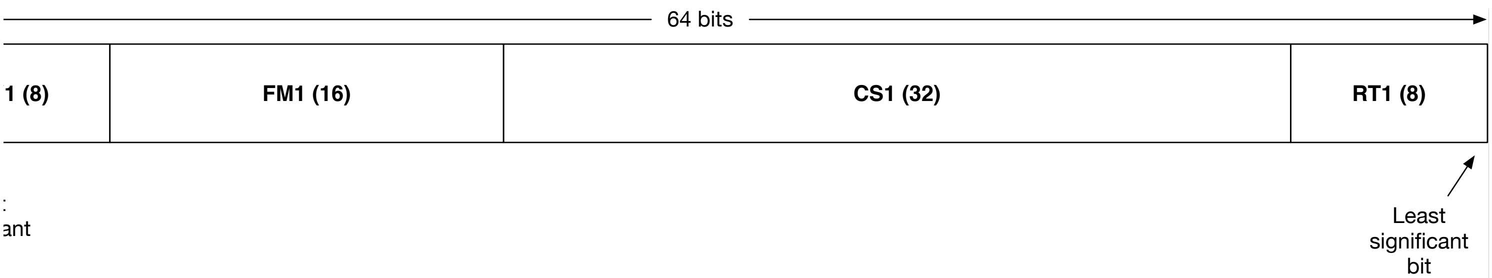
The Block Memory System (reminder)

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.
- The connection to the BLOCK routinely corrupts frames, and thus a checksum field is used to ensure frame integrity.

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)
7-23   - FM1 (Key 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 the `UNUSED` parts of the opcode should always be zero, as should any register not explicitly listed in the specifications below. If the `frame` argument is not specified, then it should be passed as NULL.

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
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

Register: **FM1**	frame number to write to	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.

Instructions

1. Login to your virtual machine. From your virtual machine, download the new workload provided for this assignment. To do this, use the `wget` utility to download the file off the main course website **here**.

2. Copy your assignment 2 directory and copy the new workload to it. Change into that directory.

```
% cd cmpsc311
% cp -r assign2 assign3
% cp ~/cmpsc311-sum19-assign3-workload.txt assign3/workload
% cd assign3
```

3. You are to modify the `block_driver.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.

4. 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.

5. To test your program with these provided workload files, run the code specifying a workload file as follows:

```
% rm block_memsys.bck
% ./block_sim -v workload/cmpsc311-sum19-assign2-workload.txt
% ./block_sim -v workload/cmpsc311-sum19-assign3-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 for both workloads: **BLOCK simulation: all tests successful!!!.**

To turn in:

1. Create a tarball file containing the `assign3` 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-assign3.tgz
% cd assign3
% 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.
