Assignment #2 - LionCloud (v 1.0)

Submit Assignment

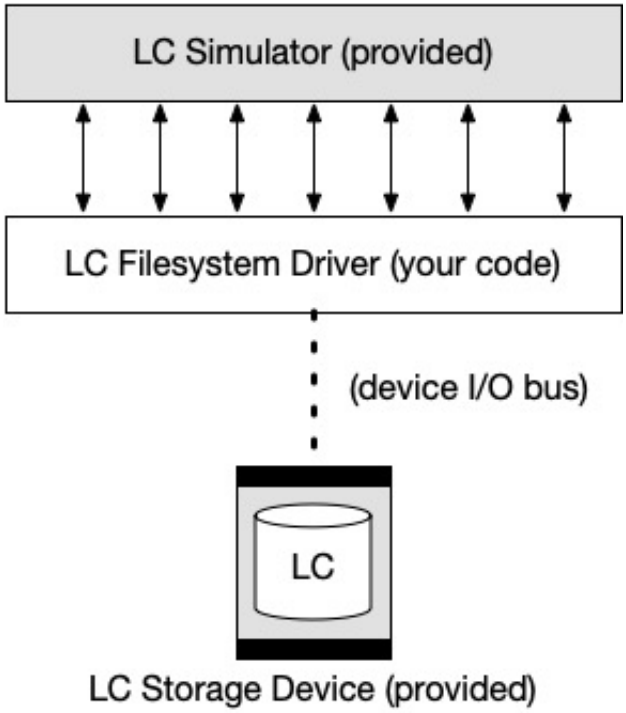
Due Wednesday by 11:59pm **Points** 125 **Submitting** a file upload
File Types tar.gz and tgz **Available** Feb 29 at 12am - Mar 7 at 11:59pm 8 days

All remaining assignments for this class are based on the creation and extension of a user-space device driver for a remotely connected cloud device (called a Lion Cloud device). The vast majority of code you will write and debug will be to provide a filesystem interface on top of the virtual device (which is provided). At the highest level, you will translate file system commands into device operations (see LC device interface specification below). The file system commands include `open`, `read`, `write`, `seek`, `close` and `shutdown` for files that are written to your file system driver. These operations perform the same function with essentially the same arguments as the normal UNIX I/O operations, with the caveat that they direct file contents to the LC storage device instead of the local host/OS filesystem.

Note: This assignment description provides an overview of the assignment. The associated slides (and in class presentation) provide more information, hints, and guidance for the project. Please refer to those materials as well in completing this assignment.

System and Project Overview

You are to write a basic device driver that will sit between a virtual application and virtualized hardware devices. The application makes use of the abstraction you will provide called the LionCloud driver. The design of the system is shown in the figure here:



The LionCloud device is a software simulation of a storage device. It has a basic protocol by which you communicate with it through a series of registers and data transfer region. You will communicate with this device and place and get data in form of blocks. You will provide all the code to talk to these devices and implement a simple flat filesystem (no directories).

Described in detail below, we can see three layers. The `lcloud_sim` application is provided to you and will call your device driver functions with the basic UNIX file operations (open, read, ...). You are to write the device driver code to implement the file OS operations. Your code will communicate with a virtual devices by setting registers and passing data through a virtual bus that connects to one more more virtual cloud devices.

All of the code for application (called the simulator) and the virtual LionCloud devices is given to you. Numerous utility functions are also provided to help debug and test your program, as well as create readable output. A sample workload has 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 LionCloud Filesystem (lcloud)

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).

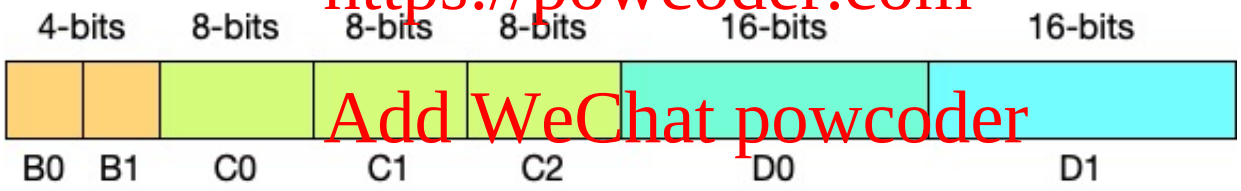
The functions you will implement for this assignment are:

- `LcFHandle lcopen( const char *path );` - This function will open a file (named path, e.g.,) 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 (position 0). Note that there are no subdirectories in the filesystem, just files (so you can treat the path as a unique filename). The function should return a unique file handle used for subsequent operations or -1 if a failure occurs.
- `int lcclose( LcFHandle fh );` - 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.
- `int lcread( LcFHandle fh, char *buf, size_t len );` - 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.
- `int lcwrite( LcFHandle fh, char *buf, size_t len );` - 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.
- `int lcseek( LcFHandle fh, size_t off );` - The function should set the current position into the file to `loc`, where 0 is the first byte in the file (`loc` is an *absolute* position from the front of 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.
- `int lcshutdown( void );` - The function should shut down the system, including powering off the devices and closing all files.

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, (b) allocate parts if the memory system to place data, (c) copy data into and out of the devices 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 Lion Cloud Device Cluster (LionCloud)

You will implement your driver on top of the Lion Cloud device cluster, which is a collection of virtual storage devices that communicate over a kind of virtual bus/network connection. You will begin by powering on the bus, probing to find out how many devices are available (there are a maximum of 16), then allocating space on the devices as the application creates and writes file data to the devices. Each device is uniquely identified by is device Id (whose type is LcDeviceId) which is discovered during the probe process. Each device has a two-level addressing scheme, where it has a number of sectors (s) containing blocks (b); therefore each device has s*b total blocks. A block is a region of memory of `LC_DEVICE_BLOCK_SIZE` bytes, as declared in the (see `lcloud_controller.h` header file.



- You communicate with the devices system by sending requests and receiving responses 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
------ ----------------------------------------------
   0-3 B0 - first 4 bit register
   4-7 B1 - second 4 bit register
  8-15 C0 - first 8 bit register
 16-23 C1 - second 8 bit register
 24-31 C2 - third 8 bit register
 32-47 D0 - first 16 bit register
 48-63 D1 - second 16 bit register
```

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

```
LCloudRegisterFrame lcloud_io_bus( LCloudRegisterFrame frm, void *xfer );
```

All messages use the following registers:

Register	Request Value	Response Value
b0	0 when message sent to devices	1 when message sent from devices
b1	0 when sending to devices	1 is acknowledge/success from device, any other value represents failure code from device
c0	always the operation code (see LcOperationCode type in `lcloud_controller.h`)	The same code as sent.

The operation code (`LcOperationCode`) indicates what operation is being requested (when sending) or completed (when receiving from the device). The following operations are available:

Operation	Description	Remaining register use
LC_POWER_ON	Initialize interface to the device cluster	All other registers (c1, c2, d0, d1) should be 0 on both send and receive
LC_DEVPROBE	Probe the bus to see what devices are	All other registers should be 0 on both send. On receive, d0 contains a bit mask of present devices, where device there are a possible 16 devices, where the bit 2^x=1 indicates that device ID x is present. For example if the 2^4 (16) is present, then you know a device with ID 4 is on the bus.
LC_BLOCK_XFER	Transfer a block to the device	◦ c1 - the device ID for the device to read from ◦ c2 - LC_XFER_WRITE for write, LC_XFER_READ for read ◦ d0 - sector to read/write from

		◦ d1 - block to read/write from
LC_POWER_OFF	Power off the device	All other registers should be 0 on both send and receive

---

Honors Option

When you receive the indication to shut down the device cluster, you must go back and erase all of the data you wrote to the array. That is you must write all zeros you previously used in any write operation, regardless of what file it may have been mapped to.

Instructions

1. Login to your virtual machine. From your virtual machine, download the starter source code provided for this assignment. To do this, go to the canvas website and download the file assign2-starter.tgz (you should be able to click the link here).

2. 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
% cd assign2
[ START EDITING ]
```


   Once unpacked, you will have the starter files in the assign2, including the source code, libraries, some workload and manifest files and a Makefile.

3. Install some libraries you will need for the assignment. Run the following command in your terminal window.

```
sudo apt-get install libcurl4-gnutls-dev libgcrypt-dev
```

4. You are to complete the `lcloud_filesys.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 (after running `make`) with these provided workload files, run the code specifying a workload file as follows:

   ```
   ./lcloud_sim -v cmpsc311-assign2-manifest.txt cmpsc311-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:

   LionCloud simulation completed successfully!!!

To turn in:

1. Create a tarball file containing the `assign2` directory, source code and build files as completed above. Submit the final tar file through canvas by the assignment deadline. 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 ~ see above)
   ```
