CMPSC 311 - Introduction to Systems Programming

PennState

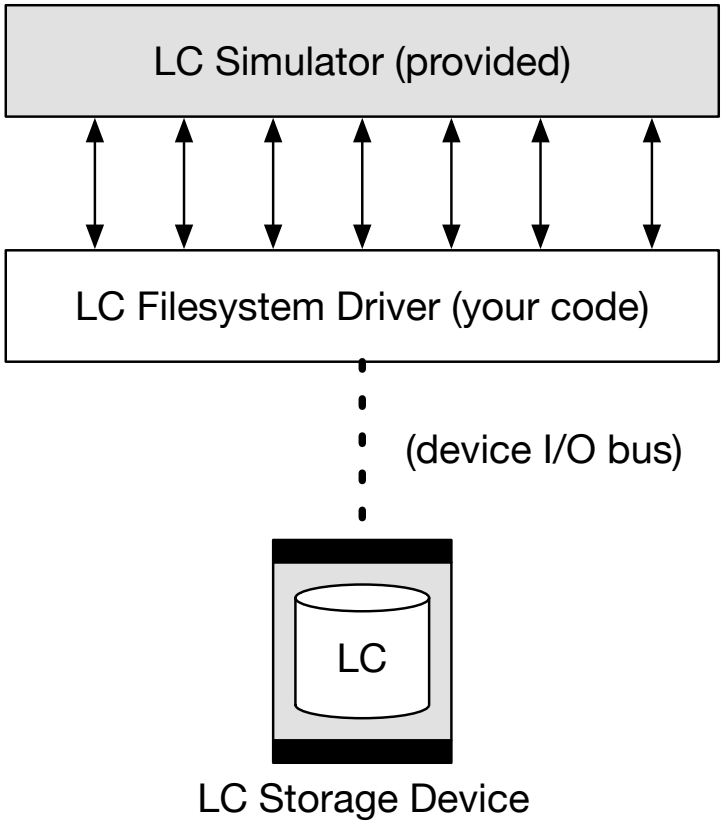
Systems and Internet Infrastructure Security Labratory

Assignment #2 – The Lion Cloud Driver v1.0

Professor Patrick McDaniel

Overview

- Idea you are to maintain the correct file contents in a Cloud Based virtual storage device.
- You will write code to communicate with a devices over a a virtualized bus that will store file-system data.
- 3 things to know
  - How file I/O works
  - How the LoinCloud bus works
  - How to make the memory device stuff look like files



Assignment Basics

- You are to write the file operation functions defined in the `lcloud_filesys.c`
  - Translate then file operations into device IO operations
  - Maintain, in the memory device, the file contents
    - You must insert data, read data, and maintain a file handle and a file allocation data.
    - Your code: open, read, write, close, seek …

- Your code will be exercised by the simulator given to you (it will call your functions and verify the data you are returning is correct).

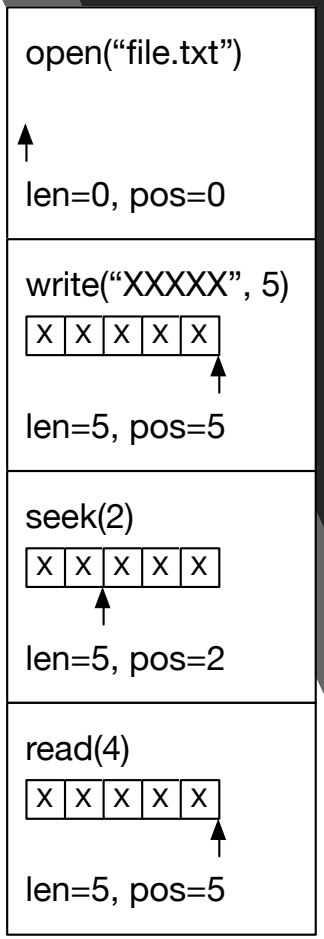
Your driver

- `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. 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.
- `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.

Your driver (cont.)

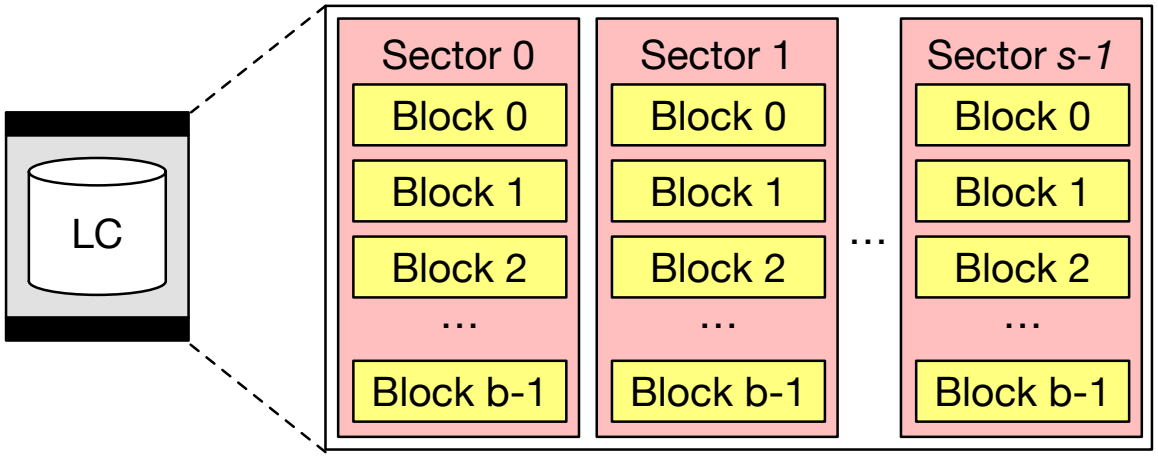
- `int lcwrite( LcFHandle fh, char *buf, size_t len );` - The function should write count bytes into the file referenced by the file handle. If the write goes beyond the end of the file the size should be increased. The function should always return the number of bytes written, e.g., count. The function 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. 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.

Maintaining file contents



- Open prepares an empty file for reading (zero length)
- Write writes bytes into the file
- Seek seeks to a position in the file (end if pos > length)
- Read copies up to length number of bytes from the file
- Close deletes the contents

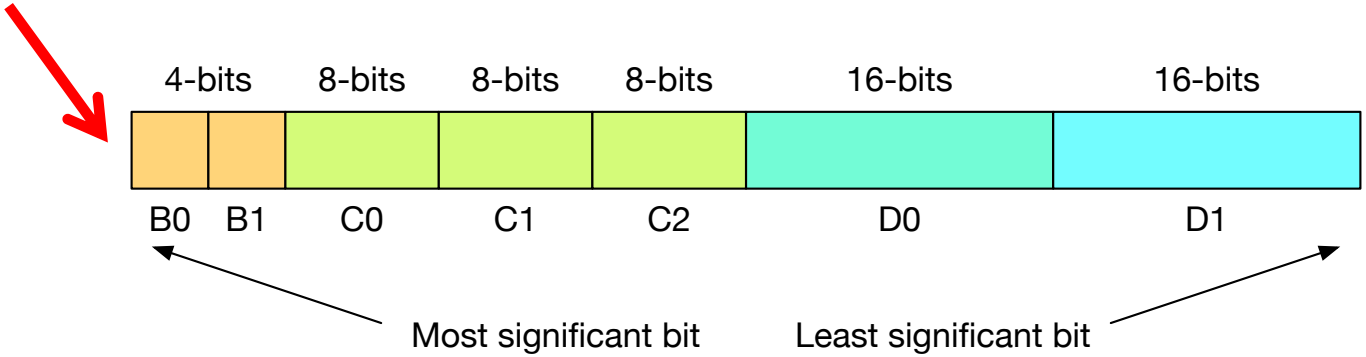
Lion Cloud Devices



- Each device has `LC_DEVICE_NUMBER_SECTORS` sectors and `LC_DEVICE_NUMBER_BLOCKS` blocks. The blocks and sectors are zero indexed.
- A frame is memory block of `LC_DEVICE_BLOCK_SIZE` bytes.
- You can assume that there is only one device in the system.

CART Opcode Execution

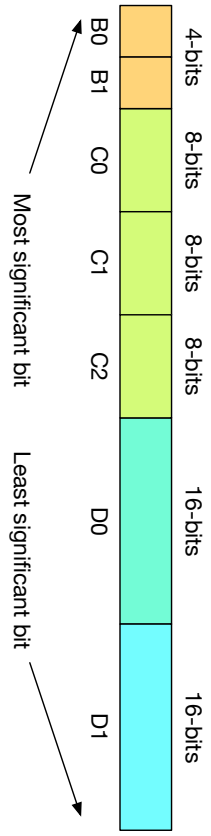
```
LCloudRegisterFrame lcloud_io_bus( LCloudRegisterFrame frm, void *xfer );
```



You communicate with devices through a set of registers of different sizes that encode the opcode and arguments for the operation. The “data” associated with the opcode (where needed) is communicated through the fixed sized transfer buffer (LC_DEVICE_BLOCK_SIZE).

Using the Lion Cloud Registers

- There are 7 registers: B0/B1 (4 bits), C0/C1/C2 (8 bits), and D0/D1 (16 bits)
  - B0 is always used to indicate who is the sender of the message. It is always 0 when you are sending, 1 when it is the devices responding.
  - B1 always contains a return/status code, should always be 0 when sending to devices. 1 is acknowledge/success from device, anything else is failure.
  - C0 is always the operation code (see LcOperationCode type in lcloud_controller.h)
  - The rest of the registers are dependent on the type of operation being performed (see next)



Operation types / the IO bus

- LC_POWER_ON - Initialize interface to a given device
  - All other registers should be 0 on both send and receive

- LC_POWER_OFF - Power off the device
  - All other registers should be 0 on both send and receive

- LC_DEVPROBE - Probe the bus to see what devices are present
  - 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 device with ID 4 is on the bus.

Operation types / the IO bus (cont.)

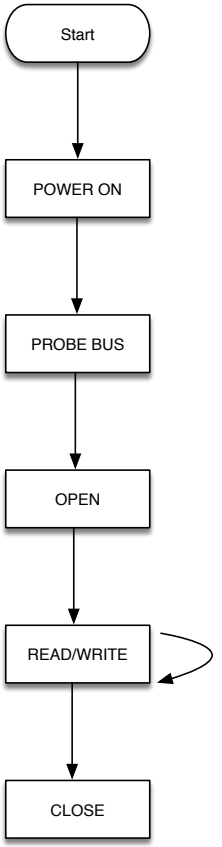
- LC_BLOCK_XFER - Transfer a block to the device
  - `c1` - the device ID for the device to read from
  - `c2` - LC_XFER_WRITE for write, LC_XFER_READ for read
  - `d0` - sector to read/write from
  - `d1` - block to read/write from

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

`xfer` is `NULL` for every command except `LC_BLOCK_XFER`, in which you put a pointer to a buffer to read or write from.

Summary

- To summarize, your code will receive commands from the virtual application (simulator). It will do the following functions to implement the device driver:
  a. Power on the devices using the power on
  b. Probe the bus to see what devices are available (one)
  c. On writes, you have to figure out where to place the data (and remember it)
  d. On reads, return the previously stored data in those blocks
  e. Power off the controller when asked to



Honors option

- 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. The program on exit will indicate via the log whether you have successfully completed the honors option:

Testing your program

- The testing of the program is performed by using the simulated workloads. The main function provided to you simply calls the simulator. To test the program, you execute the simulator using the -v option, as:

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

- If you have implemented everything correctly, the log should display the following message:

```
LionCloud simulation completed successfully!!!
```

Getting started ...

- Get the file from the canvas page.
- Change into your development directory and unpack the file:

```
% cd ~/cmpsc311
% cp assign2-starter.tgz cmpsc311
% cd cmpsc311
% tar xvfz assign2-starter.tgz
% cd assign2
% make
```

Install libraries

- You will have to 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
```

Hints

- Use the `logMessage` interface to log information about how your program is running. (see next page)
- Carefully read and understand the error messages that are being written to the log.
- Review the simulator code to see how the interfaces in the program should operate.

!!!!

logMessage()

- Available in the cmpsc311_log.h – provides you a way to print out information from your code, and works like printf()
  - logMessage(LEVEL, “Stuff … %d”, parameters);
  - Use the “LcDriverLevel” level for your code (global value)

```
logMessage(lcDriverLevel, “This is my code %d [%s]”, val, str );
```

Open() pseudo-code

```
Open (path) {

        1) check if file already open
              (fail if already open)

        2) pick unique file handle

        3) save filename and file information locally

        4) set file pointer to first byte

        5) if file not exists
              set length to 0

        return file handle

}
```

Read() pseudo-code

```
Read ( file handle, length, buffer ) {

        1) check if file handle valid (is associated with open file)

        2) check length to if it is valid

        3) figure out what device/sector/block data for the read is

        4) get the block

        5) copy the data from the xfer block to buffer

        6) update the buffer length

        return the number of read bytes

}
```

First functions ...

- The first functions you should write should create a register structure and extract a register structure:

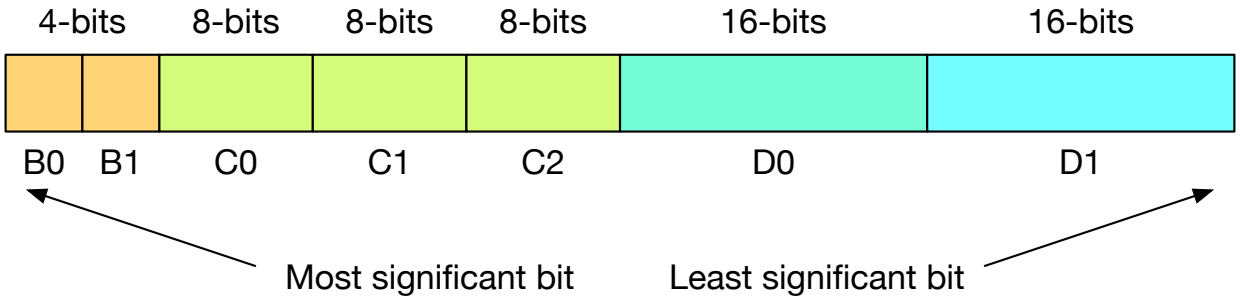
```
LCloudRegisterFrame create_lcloud_registers(…?) {
      ???
}

??? extract_lcloud_registers(LCloudRegisterFrame resp, …) {
      ???
}
```

Hint: use the bit operations to build packed registers ....

Packing

- To pack the values (variables b0, b1 ….), you need to combine some set of shift operations and bitwise (&)



Hint:

```
a = 0011
b = 0011
c = (b<<2) & a
```

Now : `c = 1111`

Doing an operation

- For each operation, say LC_POWER_ON

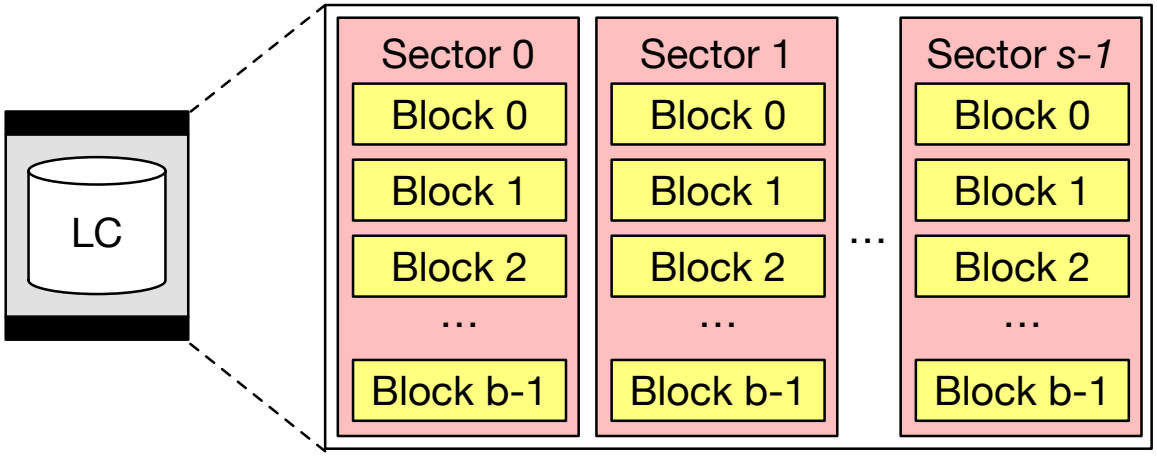
1. Pack the registers using your `create_lcloud_registers` function
2. Set the `xfer` parameter (NULL for power on)
3. Call the `lcloud_io_bus` and get the return registers
4. Unpack the registers using your `extract_lcloud_registers` function
5. Check for the return values in the registers for failures, etc.

Doing an operation

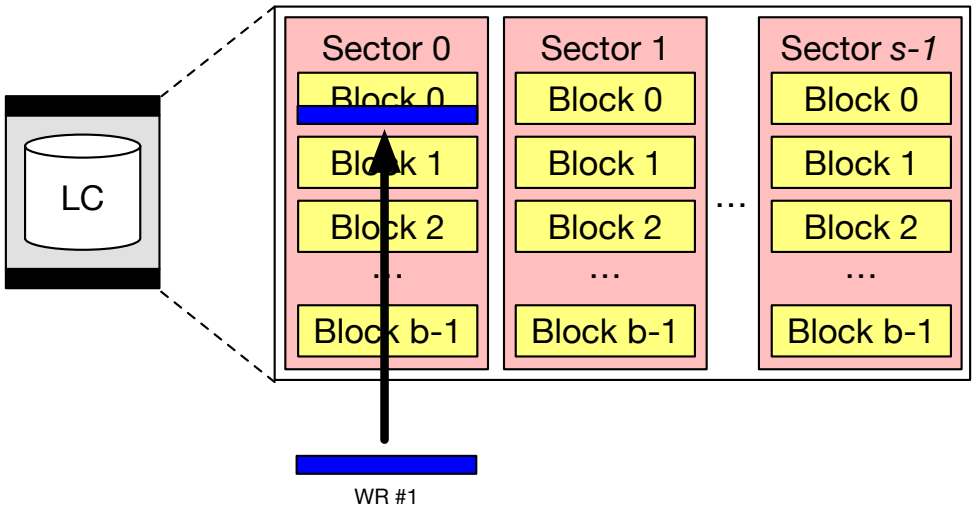
- Example from PDM's implementation of the filesystem
  - the variable names should tell you something about what is going one)
  - This particular code contains an operation into WRITE into the device …
  - buf is a 256 byte character array (see last lecture)

```
/* Do the write operation, check result */
frm = create_lcloud_registers(0, 0, LC_BLOCK_XFER, did, LC_XFER_WRITE, sec, blk );
if ( (frm == -1) || ((rfrm = lcloud_io_bus(frm, buf)) == -1) ||
    (extract_lcloud_registers(rfrm, &b0, &b1, &c0, &c1, &c2, &d0, &d1)) ||
    (b0 != 1) || (b1 != 1) || (c0 != LC_BLOCK_XFER) ) {
    logMessage( LOG_ERROR_LEVEL, "LC failure writing blkc [%d/%d/%d].", did, sec, blk );
    return( -1 );
}
```

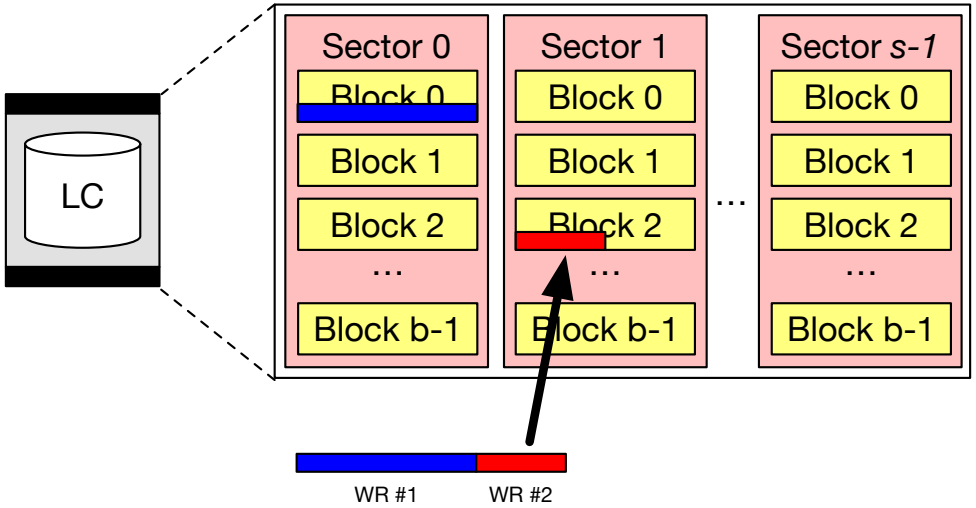
What you are doing (visually) ....



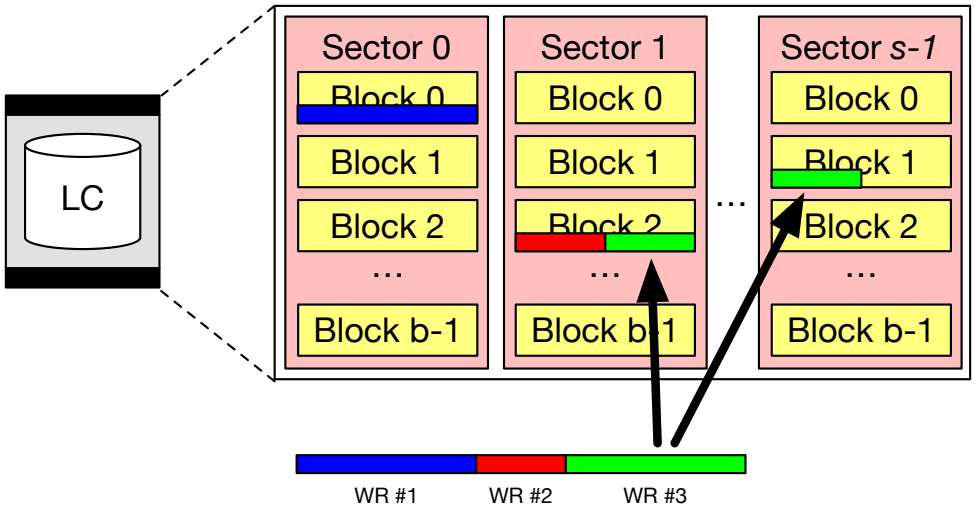
- Initially you will start with empty devices, all blocks are free.
- Once you power things on and process the open, you will see writes
- For this example, we are ignoring reads but it should be clear how to do reads once you understand how to do writes.



- On the first write (#1) you will pick a block (any block, it is up to you).
- For the sake of this example, the write is exactly one block in size.
- You pick sector 0, block 0 (for one strategy), and xfer the block to it …

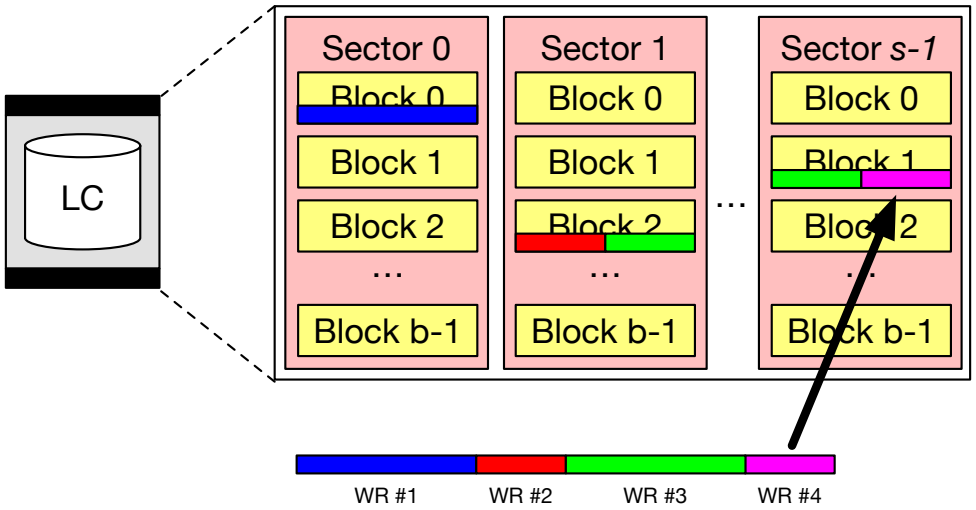


- On the second write (#2) you will pick a block (any block, it is up to you, but you can't pick 0,0 because it already has file data)
- For the sake of this example, the write is exactly one half block in size.
- You pick sector 1, block 2 (for one strategy), and write the block to it … BUT you have to remember you have only used the first half of that block …



- On the third write (#3) you start by filling up what is unused by the previous write in the block 1,2
- This write is one block long, so you split it into TWO block xfers the first half to 1,2 and the second half to a new block s-1, 1 (again up to you)

What you are doing (visually) ....



- And so on … i.e., as you add to the file you just allocate more blocks.
- Reads are the same, only you don't do allocation, you just figure out how to read the bytes from the blocks in one or more reads from the device.
- The whole assignment revolves around piecing together the reads/writes from blocks.

How to keep track of everything ...

A sample data structure for data for the file from previous example:

Filename	sample.txt									
Handle	11	Block 0		Block 1		Block 2		....	Block n	
Pos	768	Sec	0	Sec	1	Sec	S-1		Sec	-1
Len	768	Blk	0	Blk	1	Blk	1		Blk	-1

A sample data structure for data for the device from previous example:

	Block				
Sector	0	1	2	....	B-1
0	1	0	0		0
1	0	0	0		1
2	0	1	0		0
....					
B-1	0	0	0		0

Other notes

- You cannot call _any_ of the C I/I functions such as open(), fopen(), seek(), lseek(), read(), write(), close() or any of the f-varients of those functions (e.g., fopen()). We discovered that it is possible to fake out the simulator using these functions to make it look like it completed successfully but never actually do any of the device I/O correctly.
- File handles - this is just a number YOU select (any number), that allows you code to understand which file it is referring to when doing a read or write. For the 2nd assignment you can just return any integer number of your choosing (e.g., 11), since there is only one fie.

Questions asked:

- Q: In the lcseek function, I am still unable to understand the phrase in the slides 'The function should set the current position into the file to loc, where 0 is the first byte in the file'. Is loc referring to 'off' in size_t off? If so, does this mean that off has to be equal to something?
- A: off is the “offset from zero”, so it is setting the absolute location. So,
  - lcseek(10) - sets the file position to the 10th byte
  - Lcseek(1000) - sets the file position to the 1000th byte