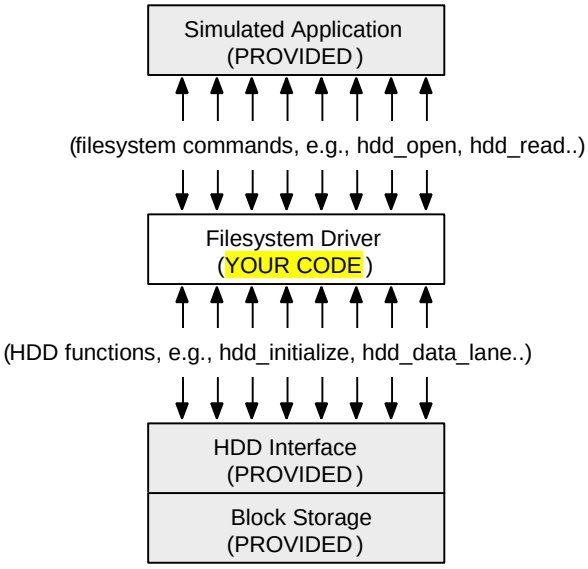
Assignment #2 - CRUD Device Driver (Ver 1.2)

CMPSC311 - Introduction to Systems Programming Fall 2017- Prof. McDaniel

Due date: October 11th, 2017 (11:59pm)

Overview

All remaining assignments for this class are based upon you providing an easily used set of functions so that an application that uses your code can easily talk to an external block storage device like a hard drive (HDD). This device already has its own pre-defined set of functions that allows communication with it. However, they are tedious to use (a common problem with hardware) and not as abstracted as programmers would prefer. Thus, we are going to translate them to mimic the standard C file commands (open, close, read, write, and seek) so that communicating to the device is easier for others. In other words, your application is acting like a driver for the HDD device. ***Our functions are called hdd_open, hdd_write, hdd_read, hdd_seek, and hdd_close***.



What is the block storage device?

The block storage device does not actually exist. It is a virtual device (modeled in code). Pretend the virtual device is some external HDD for your own understanding. The block storage device stores variable sized blocks of data (termed blocks), which your application will write to in this project. A common misconception is that you can just write data directly to a storage device. This is not true. For memory fragmentation and other reasons, writing to a normal disk drive must be done by writing to blocks. Initially, there are no available blocks. Thus, there is nowhere to write data to on the device. However, you can specify the creation of a block and its size. Once a block exists, you can write data to it and then read the data it holds (or update the data if desired). A ***unique*** integer value references each block and is automatically assigned to it during its creation.

How you create blocks, and read/write to them will be discussed shortly, but first understand the following: the point of this project is so that **your coded functions** (hdd_open, hdd_read, hdd_write, etc.) **abstract away the complexity of having to deal with the device's block storage directly**. Within your functions you will have to communicate to the blocks *in a very specific way*, but whoever uses your functions

will have an easier time using the HDD device, because your functions resemble the standard open, read, write, etc. C functions. (i.e. you are creating an Application Programming Interface). Note, a common misconception is that you are supposed to use standard C functions, but you are **NOT meant to use the standard open, read, write, C functions**. Your functions just ***RESEMBLE*** them. It is **highly recommended** you know how those C functions are used so you are familiar with what we are trying to emulate.

The functions you NEED to write (i.e., the filesystem driver)

In this assignment, another application (a.k.a., the unit-tests provided to you) will call your code to evaluate your application by trying to save, read, and write files to the block storage and validate your results. Unit-testing will use your functions to do so, hence do not change function names (you will be penalized). For this assignment (HINT: but not future ones, so plan ahead), there will only ever be one file open at a time, and furthermore, each file that the user wants to write to the device will always fit within the maximum possible size of a block (HDD_MAX_BLOCK_SIZE). Thus, in this assignment, there is a one-to-one mapping of files to blocks.

When your code is called, ***hdd_open*** will be the first function called and will be given a filename. Your function code must return a number, also called a file handle, which will uniquely refer to that particular file across all your other functions. For instance, when the user wants to write something to a file on the device, your ***hdd_write*** function will be called with the first parameter being the corresponding file handle you previously returned in ***hdd_open***. If the user tries to call any of your functions (besides ***hdd_open***) with a file that is not open, you must return -1 (i.e. unsuccessful) and handle appropriately. You must keep track of whether or not a file is open using your own data structure.

You can assume that when a user opens a file for the first time via ***hdd_open***, that there is no pre-existing data in the block storage about that file. The application will need to call ***hdd_write*** in order to write anything about this file for the first time to a block. Then the application will need to call ***hdd_read*** to get back any data that has been written to that file. When the application has finished using a file (i.e. read/write), it calls ***hdd_close***. For this assignment, calling ***hdd_close*** will delete all the contents of the file in the block storage (HINT: future assignments will keep contents). Lastly, where the application starts reading from and writing to in a file is determined by the ***current seek position***. The seek position is automatically placed at the end of the last read/write operation. The ***hdd_seek*** function places the seek position wherever the user would like within the file (again, it is recommended to understand the standard seek C function).

As a programmer, it is up to you to decide how to implement these functions. However, the functions must maintain the file contents in exactly the same way as a normal filesystem would (meaning do not re-order the bytes of data). The functions that you are to implement are declared in **hdd_file_io.h** and should be implemented in **hdd_file_io.c** as follows:

Function	Description
hdd_open	[1] This call opens a file (i.e., sets any initially needed metadata in your data structure) and returns a UNIQUE integer file handle (to be assigned by you). [2] For this assignment, the file can be assumed to be non-existent on the device. [3] You should initialize the device here (**but only the first time this function is called i.e. singleton pattern**). [4] The function returns -1 on failure and UNIQUE integer on success.

hdd_close	[1] This call closes the file referenced by the file handle. [2] For this assignment, you are to delete all contents stored in the device's blocks associated with this file when it is closed. [3] The function returns -1 on failure and 0 on success.
hdd_read	[1] This call reads a *count* number of bytes from the current position in the file and places them into the **buffer called *data***. [2] The function returns -1 on failure or the number of bytes read if successful. [3] If there are not enough bytes to fulfill the read request, it should only read as many bytes that are available and return the number of bytes read.
hdd_write	[1] This call writes a *count* number of bytes at the current position in the file associated with the file handle *fh* from the buffer *data*. [2] The function returns -1 on failure, or the number of written read if successful. [3] When number of bytes to written extends beyond the size of the block, a new block of a larger size should be created to hold the file.
crud_seek	[1] This call changes the current seek position of the file associated with the file handle *fh* to the position *loc*. [2] The function returns -1 on failure (like seeking out range of the file) and 0 on success.

A central constraint to be enforced on your code is that **you cannot maintain any file content or length information in any data structures after your functions have returned**–all such data must be stored by the device and its blocks. **Assumptions** you can make for this assignment:

- No file will become larger than the maximum block size (HDD_MAX_BLOCK_SIZE).
- Your program will never have more than one file open at a time for this assignment.

How to communicate with the device?

In order to communicate with the device, there are four functions:

```
int32_t hdd_initialize();
int32_t hdd_read_block_size(HddBlockID bid);
int32_t hdd_delete_block(HddBlockID bid);
HddBitResp hdd_data_lane(HddBitCmd command, void * data);
```

<u>You will not be able to see the internals of these functions</u> (they are stored in a static library (.a) provided to you), but you can see the function declarations in the **hdd_driver.h** file. The first three (relatively simpler) functions are described below:

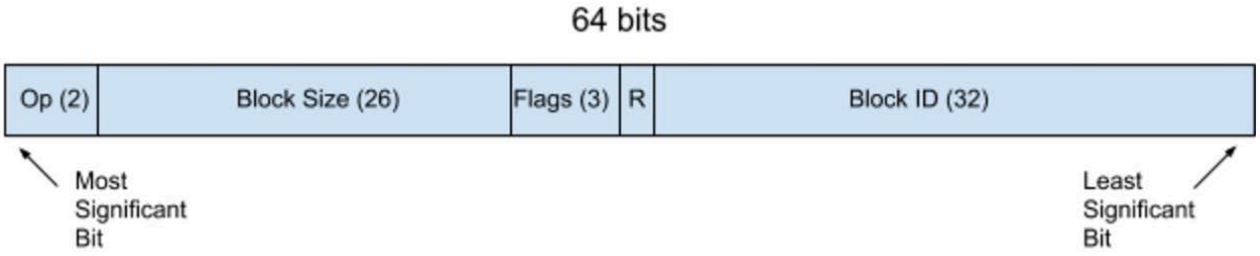
hdd_initialize	This must be called **only once throughout the entire program execution** called (i.e. singleton pattern) and be called before any of the other three functions. This function initializes the device for communication. It returns 1 on success and -1 on failure.
hdd_read_block_size	This function expects a block ID and returns its block size (in bytes). You must read the size of a block **that exist**s (i.e., one you have already created) or it will return an error if the block does not exist. This function returns the block length on success and -1 on failure.

hdd_delete_block	This function requires a block ID and deletes it and all data associated with it on the device. You must delete a block **that exists** or it will return an error if the block does not exist. This function returns 1 on success and -1 on failure. Note that a deleted block's ID may be recycled again for use.

The ***hdd_data_lane*** function, the most complicated function of the four has been designed for your application to transfer data to and from the device. The function allows you to create a block (and give it data), read data from a block, and overwrite a block with new data. First examine the ***HddBitCmd*** parameter and the ***HddBitResp*** return value. They are defined in the hdd_driver.h file as:

```
typedef uint64_t HddBitCmd;
typedef uint64_t HddBitResp;
```

They are both just 64-bits of information. **DO NOT THINK THEY ARE MEANT TO BE INTERPRETED AS 64-BIT INTEGERS EVEN THOUGHT THEIR TYPES ARE 64-BIT INTEGERS**. Yes, these parameters are of 64-bit integer type, but in the hdd_data_lane function, those 64-bits are taken in as a word and divided into sections (using bitwise operators) that each have their own meanings (figure below). The lower 32 bits represent a parameter called Block ID, the next bit represents a parameter called R, and so on. See the below figure:



These subdivisions can be thought of as a compact way for one function to have many parameters within just one parameter. Essentially, the ***HddBitCmd*** parameter allows you to specify a block to transfer data to or from, and that data is pointed to by the *data* parameter, which points to either the actual data to transmit or to where the received data should be stored. How the *data* parameter is used depends on the HddBitCmd's fields. **The exact meanings of the parameter, HddBitCmd, and the returned value, HddBitResp, are listed below**:

- Block ID
  - In **HddBitCmd**: This is the block identifier of the block you are executing a command on. If the block does not yet exist, when trying to create one, leave this field as all 0s.
  - In **HddBitResp**: hdd_data_lane returns the created block's id if HddBitCmd's Op field was HDD_BLOCK_CREATE (otherwise, it returns the same block ID you gave in the HddBitCmd).
- Op - Opcode
  - In **HddBitCmd**: This is the request type of the command you are trying to execute. The value can be one of HDD_BLOCK_CREATE, HDD_BLOCK_READ, or HDD_BLOCK_OVERWRITE (see next section for meaning of these).
  - In **HddBitResp**: It will be the same Op as what you sent in HddBitCmd (thus, not useful).
- Block Size
  - In **HddBitCmd**: This is the length of the block you request to read from, overwrite, or create. This is always the number of bytes in the *data* parameter that are read from and written to.

    - In **HddBitResp**: It will be the same block size as what you sent in HddBitCmd (thus, not useful).
- Flags - These are unused for this assignment (set to 0 in HddBitCmd).
- R- Result code:
    - In **HddBitCmd**: Not used (set to 0).
    - In **HddBitResp**: This is the success status of the command execution, **where 0 (zero) signifies success, and 1 signifies failure.** You must check the success value for each bus operation even though nothing should be failing.

The Op's values can be found in hdd_driver.h, and below summarizes what they mean to help you understand how to use them in your application:

- HDD_BLOCK_CREATE - This command creates a block whose size is defined in the Block Size field of the HddBitCmd. The *data* buffer passed to hdd_data_lane should point to the start location of the data bytes to be transferred. After completion, the data has now been saved to the newly created block on the device. If successful, the operation will return the new block ID in the HddBitResp's Block ID field.
- HDD_BLOCK_READ - This command reads a block (in its entirety) from the device and copies its contents into the passed *data* buffer. The Block Size field should be set to the exact size of the block you're trying to read from (thus, you must read the entire block, not just parts of it). The *data* buffer should have enough allocated memory to store the entire block.
- HDD_BLOCK_OVERWRITE - This command will overwrite the contents of a block. **Note that the block size CAN NEVER change**. Thus, the call will fail unless the *data* buffer sent in is the same size as the original block created. Just like in HDD_BLOCK_CREATE, the *data* buffer should point to the start location of the data bytes to be transferred.

General Compilation and Running Instructions *[See class notes to help with some of these steps]*

1. From your virtual machine, download the starter source code provided for this assignment from CANVAS class website. Move the tgz file to your assignment directory.
2. Install this dependency you will need via the terminal:
   % sudo apt-get install libgcrypt11-dev
3. Have the terminal open to the directory where the .tgz starter code file is located. Now unpackage the contents of the file:
   % tar xvfz assign2-starter.tgz
4. You should be able to use the Makefile provided to build the program without modification (thus just compile by typing:
   % make clean; make
5. To get started, focus on **hdd_file_io.c. This file** contains code templates for the functions you **MUST** fill in as described above.
6. Add appropriate English comments to your functions stating what the code is doing. All code must be correctly (and consistently) indented. Use your favorite text editor to help with this process! **You will lose points if you do not comment and indent appropriately**. Graders and TAs will need to understand your comments to follow your code and grade. Do not forget to add comments to the Makefile. The Makefile structure has been explained in class.
7. Built into hdd_file_io.c file is a function called **hddIOUnitTest**, which is automatically called when you run the program to check for its correctness. The main() function simply calls the function hddIOUnitTest.

If you have implemented your code correctly, it should run to completion successfully. To test the program, you execute the simulated filesystem using the -u and -v options, as:

./hddsim -u -v

If the program completes successfully, the following should be displayed as the last log entry:

HDD unit tests completed successfully.

Getting Started: First Steps in writing your code

1. Start by writing code for hdd_open first. Lessons learnt from previous years, writing hdd_write, hdd_read, hdd_seek, and then hdd_close in that order will help you complete the assignment with less difficulty. For example, code hdd_write before you write hdd_read, because the unit tests will call hdd_write before hdd_read making it easier for you to debug your code. Additionally, this makes sense as nothing can be read without some data there that's already written.
2. You will need a global data structure placed at the top of your hdd_file_io.c file to help you preserve metadata about what files are open, their associated object, and other details defined by you. Write a global data structure to store file metadata and remember you cannot store length information or the actual contents of the file itself. You can assume for this assignment that there is only one file ever in existence at a time.
3. Make sure you call hdd_initialize in hdd_open before calling any of the other three hdd_ functions. Recall hdd_initialize can only be called once throughout the entire program's execution.
4. Note: when using printf always end with newline ("\n"). For instance, if doing printf("My message"); instead do printf("My message\n") as the print statement may not get executed in case of error otherwise. **Additionally, use printf messages to help debug your code.** For example, provide function name and location to trace where your application fails (i.e. core dumps), but you can also run the code with the gdb debugger by just typing "gdb hddsim"—a full tutorial on the gdb is out of the scope of these instructions. This will also help the TA's trace and debug your code. See the TA's for help!

Turn-in Instructions

1. Create a tarball file containing the assign2 directory, source code and build files. Upload the program to 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 given 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.

Honors Option [2 extra points for other students Warning: not trivial]:

Additional Coding: Place the additional restriction that each object in the object store can be no longer than 1024 bytes. This requires the system to track multiple ordered objects on the storage device.

Questions below, answer using word document.

Question 1: This code snippet is noncompliant. Diagnose and explain why?

```c
#include <errno.h>
#include <stdio.h>
void func(const char *filename)
{
    FILE *fileptr;
    errno = 0;
    fileptr = fopen(filename, "rb");

    if (errno != 0)
    { /* Handle error */ }

}
```

Question 2: This code snippet is noncompliant. Diagnose and explain why?

```c
enum { TABLESIZE = 1000 };
static int table[TABLESIZE];
int *fcmpsc311(int index)
{
    if (index < TABLESIZE)
    {
        return table + index;
    }

    return NULL;
}
```

Question 3: This code snippet is noncompliant. Diagnose and explain why?

```c
void func(void)
{
    for (float x = 0.1f; x <= 1.0f; x += 0.1f)
    {
        /* Loop may iterate ???? times */
    }
}
```