Assignment #3 - HDD Extension

CMPSC311 - Introduction to Systems Programming
Spring 2018- Prof. McDaniel
Due date: April 6th, 2018 (before 11:59pm)

Overview

You must start this assignment using the hdd_file_io.c code that you submitted in assignment #2.

This assignment will expand upon the previous one by saving the state of the HDD device in a file called **hdd_content.svd** (you create this file from your tool). To be more precise, you'll be able to create hdd_content.svd, which will hold all the device's blocks. In this project, you'll write three functions that allow for the creation, loading, and deletion of hdd_content.svd. You won't actually be dealing with file IO directly (so no fopen or the like), but you will be using the same (and improved) functions provided last time (*i.e. compare differences and look at your code comments to get started*) to handle everything to do with hdd_content.svd. Continue reading for more clarity.

Multiple Files

Unlike the previous assignment, you must be able to handle multiple open files at the same time. Additionally, files are no longer deleted when closed so change your **hdd_close** function to no longer delete a file. Thus, now **hdd_close** should "close" the file—meaning, once the user calls **hdd_close**, the seek position is reset and the user must re-open the file via **hdd_open** in order to then use **hdd_write** and **hdd_read** again.

Note, that in order to manage multiple files, it's recommended you use an array of structs for your global data structure as all bytes are laid contiguously in memory (*which you may find useful for other parts of this assignment*).

The Meta Block

The block storage now has a block that you can write to without needing a block ID: the meta block. You can create, read, and write to it just like you would any other block using **hdd_data_lane** except with the following changes:

- The BID within HddBitCmd should be set to 0
- The Flags within HddBitCmd should be set to HDD_META_BLOCK (i.e., a new enum for 1 defined in hdd_driver.h)

The meta block should hold all the metadata about all of your files (*i.e., in your global data structure*). Thus, when the entire block storage is saved to **hdd_content.svd**, the meta block contents is saved so that when hdd_content.svd is loaded back into the device, you can retrieve the previous file system state.

New Functions

Below details **three new functions that YOU must implement**. Note that in the previous assignment, the first function called was always **hdd_open** (*at least, that's the intended use of the functions as calling anything else first doesn't make sense*). However, in this assignment either

hdd_format or **hdd_mount** will be called first. Thus, make sure you have an initialization check in both functions.

- **[1] uint16_t hdd_format(void);**
    - Returns 0 on success and -1 on failure
    - Initializes the device if not already done—this is done the same way as the previous assignment
        - If initializing, then the following will happen automatically: the **hdd_content.svd** file will be read and the device's block storage will now match what was in **hdd_content.svd**
    - Sends a **format** request to the device which will delete all blocks (*how you do so is detailed later*)
    - **<u>Creates</u>** the **meta block** and saves your global data structure to it
- **[2] uint16_t hdd_mount(void);**
    - Returns 0 on success and -1 on failure
    - Initializes the device if not already done—this is done the same way as the previous assignment
        - If initializing, then the following will happen automatically: the **hdd_content.svd** file will be read and the device's block storage will now match what was in **hdd_content.svd.**
    - **<u>Reads</u>** from the **meta block** to populate your global data structure with previously saved values
- **[3] uint16_t hdd_unmount(void);**
    - Returns 0 on success and -1 on failure
    - **Saves** the current state of the global data structure to the meta block
    - **Sends** a save and close request to the device, which will create/update the **hdd_content.svd** file (how you do so is detailed later)
    - ***Note: consider the device uninitialized after this function call***

New Flags and Op Type

In order to communicate with the meta block, there's a new flag called **HDD_META_BLOCK** that should be set in the **HddBitCmd** parameter of **hdd_data_lane**. In order to format the device, **hdd_data_lane** should also be used in which the **HddBitCmd** uses a new op type, **HDD_DEVICE**, and flag, **HDD_FORMAT**. Similarly, in order to save the blocks in the device to **hdd_content.svd**, the **HDD_DEVICE** op type is used and there's a new flag, **HDD_SAVE_AND_CLOSE**. Below are more details:

- **HDD_META_BLOCK**
    - As mentioned previously, this flag value in **HddBitCmd** signifies a read or write to the meta block
- Both the **HDD_FORMAT** and **HDD_SAVE_AND_CLOSE** flags need to be used in conjunction with the new op type, **HDD_DEVICE**. When the op type is **HDD_DEVICE**, every field besides the op field itself and the flags field should be 0. Additionally, the void data pointer (*the second parameter to hdd_data_lane*) can be set to NULL. Below are the details of the two flags to be used with the **HDD_DEVICE** op type:
    - **HDD_FORMAT**

    - This will delete the **hdd_content.svd** file (*if it exists*) and clear the block storage of all blocks
- **HDD_SAVE_AND_CLOSE**
    - This will create the **hdd_content.svd** file (*or overwrite it if it exists*) and will uninitialize the device

<u>To accommodate the new flags hdd_read_block size is now defined as</u>:

```
int32_t hdd_read_block_size(HddBlockID bid, uint8_t flags);
```

This function works exactly the same way as before in assignment #2, but if you'd like to get the size of the meta block, then you provide a BID of 0 and set the flags to **HDD_META_BLOCK**. Getting the size of normal blocks is same except just set the flags parameter to **HDD_NULL_FLAG** (i.e., 0).

General Compilation and Setup

1. To get started, copy over your previous assignment's functions (hdd_open, hdd_read,…) and any global data structures into this assignment's **hdd_file_io.c** file.
2. 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. On CANVAS, the Makefile book link is provided to you as a reference.

Testing (3 phases)

The ***<u>first</u>*** phase of testing the program is performed by using the unit test function similarly to last time (but updated to include calling the new functions). Run it by typing:

```
./hdd_sim -u -v
```

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

```
HDD unit tests completed successfully.
```

The ***second*** phase of testing will run two workloads on your filesystem implementation. To do this, run the following commands

```
./hdd_sim -v workload-one.txt
./hdd_sim -v workload-two.txt
```

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

```
HDD simulation completed successfully.
```

Be aware that running the first workload will produce a **hdd_content.svd** file that the second workload depends on. **<u>Thus, when testing, make sure you do so in the order specified here.</u>** Additionally, the next phase will depend on the **hdd_content.svd** file produced by workload two. In this ***last phase of testing***, certain files loaded in the previous workloads will be extracted. Run by typing:

```
./hdd_sim -v -x simple.txt
```

This should extract the file simple.txt from the device and write it to the local filesystem—meaning you'll have a new file called simple.txt in your directory (if you want to re-run this command, delete the newly created .txt file). Next, use the diff command to compare the contents of the new file with the original version simple.txt.orig distributed with the starter code:

```
diff simple.txt simple.txt.orig
```

If they are identical, diff will give no output (i.e., no differences). **Repeat these commands** to extract and compare the content for the files raven.txt, hamlet.txt, penn-state-alma-mater.txt, firecracker.txt, and solitude.txt.

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.
