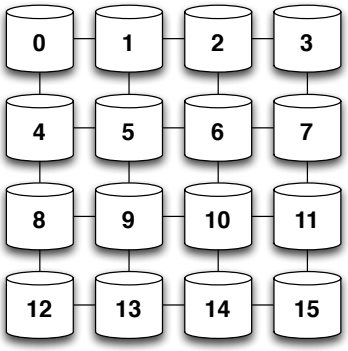
CIS 415 - Operating Systems
Project #3 - Application Space Device Driver
Spring 2014 - Prof. Butler[1]

Due date: June 10, 2014

In this project, you will develop an implementation of a device driver for a virtual device, a large-scale storage array. All of this work will be in user space, so no kernel recompilation will be necessary. Please read the following instructions carefully and perform the requested steps as directed.

1 SMSA - Serially Massive Storage Array

You are to write an application-space device driver (operating system interface) code for a new storage device called a Serially Massive Storage Array (SMSA). The device consists of a 4 by 4 grid of storage drums with 2^{16} bytes of storage each, as seen here:



The SMSA has a single read instrument (head) that physically moves from drum to drum to write and read data from the individual drives.[2] The drum contains a fixed number of blocks of equal size (`SMSA_BLOCK_SIZE`). All reads and writes are done in single blocks, i.e., every read or right is for the same size.

The implementation of the device driver will receive a series of commands form the layer above it. These commands will require you to receive and interpret the commands and call the physical layer interfaces based on the required operation. It is entirely up to you to determine how best to implement these functions within the constraints laid out below.

Your device driver will provide a virtual address space spanning the entire drum array. There are 16 drums, each with $SMSA\_DISK\_SIZE$ bytes. So your driver should support reading and writing to addresses 0 to $(16 * SMSA\_DISK\_SIZE) - 1$. Thus, your driver should translate the SMSA virtual addresses (supported by your code) to the physical addresses of the drums. Each drum is evenly partitioned into blocks of `SMSA_BLOCK_SIZE` bytes. You are to implement the address space in the order of drums, one after the other as follows: if each drum has n bytes, the first drum should map the blocks into addresses $(0 : n-1)$, the second $(n : 2n-1)$, the third $(2n : 3n-1)$, ..., and the 16th drum $(15n : 16n-1)$. In essence, you will be translating the virtual address reads and writes into reads and writes in the drum array.

To talk to the drum array, commands are sent through a single SMSA operation interface. This interface is defined through a single function call: `int smsa_operation( uint32_t op, unsigned char *block );` This function accepts an SMSA instruction `op` and a pointer to `SMSA_BLOCK_SIZE` byte buffer (NULL if not used). The structure of the `op` parameter is given in Table 1. The commands it accepts are detailed in Table 2.

[1] Special thanks to Patrick McDaniel and Eric Kilmer for help with this project.

[2] Although not relevant to the current assignment, drum/block seeks slow down the read time proportional to the distance the head has to move, e.g., a read head moving from drum 0 will take twice as long to move to drum 8 as it does to drum 4. Also, the head can only move sideways or up and down at any given time (no diagonal).

Bits	Width	Field	Description
0-5	6	Opcode	This is the command to be implemented by the drum array. (see below)
6-9	4	Drum ID	This is the ID of the drum to perform operation on
10-23	14	Reserved	Unused bits (for now)
24-31	8	Block ID	Block address within the drum

Table 1: SMSA Instruction Layout

Instruction	Description
0	
SMSA_UNMOUNT	Unmount the drum array. This should be the last thing you do.
SMSA_SEEK_DRUM	Seek to a new drum at **drum id**. Note that the drum read head will be reset to the first block (block 0).
SMSA_SEEK_BLOCK	Seek to a block at **block id**.
SMSA_DISK_READ	Read the block at current drum and block head position. The block head will be at the next block after the read.
SMSA_DISK_WRITE	Write to the block at current drum and block head position. The block head will be at the next block after the read.

Table 2: SMSA Instruction Details

The simulator program code provided to you will read a *workload* file of drum commands and call your implementation to serve then. The program is called as follows:

```
USAGE: smsa [-h] [-u] [-v] [-l <logfile>] <workload-file>

where:
    -h - help mode (display this message)
    -u - run the SMSA unit test
    -v - verbose output
    -l - write log messages to the filename <logfile>

    <workload-file> - file contain the workload to simulate
```

There are three workload files (in increasing workload complexity) included with the assignment, `simple.dat`, `linear.dat`, and `random.dat`. Your program will be run against these workloads and validated for correctness when being graded.

The program will print out a good deal of output if you use the `-v` verbose flag. Use this to debug your program. It is also encouraged to use the logging facility provided by the `logMessage` function listed in the `cmpsc311_log.h` file. See its use in the simulator code for examples. To redirect the output into a utility you can use to walk through the data, redirect the output to `less`:

```
./smsasim -v simple.dat 2>&1 | less
```

For more information on using `less`, see the manpage.

2 SMSA Specification

1. From your virtual machine, download the starter source code provided for this assignment.
2. Install the gcrypt library using the package interface:

```
% sudo apt-get install libgcrypt-dev
```

3. You are to implement 4 functions defined in the interface header file `smsa_driver.h` whose implementation is in `smsa_driver.c`:

   - `smsa_vmount()` - this will open the drum array for reading on the virtual address space. This should initialize the drum array by either retreiving the previously stored contents or formatting the drums using the SMSA interface functions (see below).
   - `smsa_unmount()` - this will close the interface to the storage array virtual address space. It should store all of the values to an external file for reloading later.
   - `smsa_vread(addr, len, buf)` - this will read a set of **len** bytes into the buffer **buf** from a specific address **addr** within the drum array. Note that the read may span multiple drum blocks and possibly drums. If any part of the read is out of range (beyond the end of the last drum), then nothing should be read and an error be returned.
   - `smsa_vwrite(addr, len, buf)` - this will write this a set of **len** bytes from the buffer **buf** to a specific address **addr** within the drum array. Note that the write may span multiple drum blocks and possibly drums. If any part of the write is out of range (beyond the end of the last drum), then nothing should be written and an error be returned.

   Note that you may need to define a number of other support functions to make clean code here. All of these functions should check the correctness of every parameter and log an error and return -1 if any value is illegal.

4. Edit the Makefile by adding all of the dependencies to the area (commented at the bottom). Make any other changes to the Makefile you feel are necessary.

5. Run the program with the example workload files and confirm that they run to completion. Sample output will be provided to compare against your run.

Bonus Points: You are to implement a function that saves the entire contents of the drum array to drum when unmount is called, and loads the contents back from drum into the array when mount is called. In essence, making the SMSA persistent across workload runs.

3 Submission Instructions

You **must** turn in the following:

1. A `README` file. The `README` will contain the following:

   - Your name
   - A list of submitted source files
   - Compilation instructions
   - General comments and anything that can help us grade your code
   - Number of hours spent on the project and level of effort required.

2. All of the source files neceesary to make the project, and the corresponding Makefile.

3. Any build files that you have created.

You can use the same directory structure as the unpacked tarball. Re-archive this directory using tar and gzip the file.

cd up a directory so that `hw1` is a subdirectory of your working directory, and run the following command to create your submission tarball, but replacing "UOEMAIL" with your cs.uoregon.edu email account name.

[frame=single,fillcolor=] tar cvzf hw3$_submission_UOEMAIL.tar.gzproject3$

Use the turnin script, located at `http://systems.cs.uoregon.edu/apps/turnin.php`, to submit your tarball.

4 Grading Guidelines

This programming project will be graded as follows:

- 10% General information about the program, and code design
- 15% Proper implementation of `smsa_vmount`
- 15% Proper implementation of `smsa_unmount`
- 25% Proper implementation of `smsa_vread`
- 25% Proper implementation of `smsa_vwrite`
- 10% Testing to be free of bugs and memory errors.

The assignment is due at 11:59 PM Samoan Standard Time about this time zone on June 10, 2014. You will have three grace days for late assignments for this class with the ability to use a maximum of two on any given assignment. You must specify that you intend to use these in your README file. If you exceed your allotment of grace days, points will be deducted at 25% per day.

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.