CMPSC 311 Fall 2019, Project 4
Cart Lab : Hierarchical Ram Backed Filesystem
Assigned: Monday, Nov. 13th, Due: Wednesday, Dec. 4th, 11PM

Yanling Wang (yuw17@psu.edu) is the lead person for this assignment.

Introduction

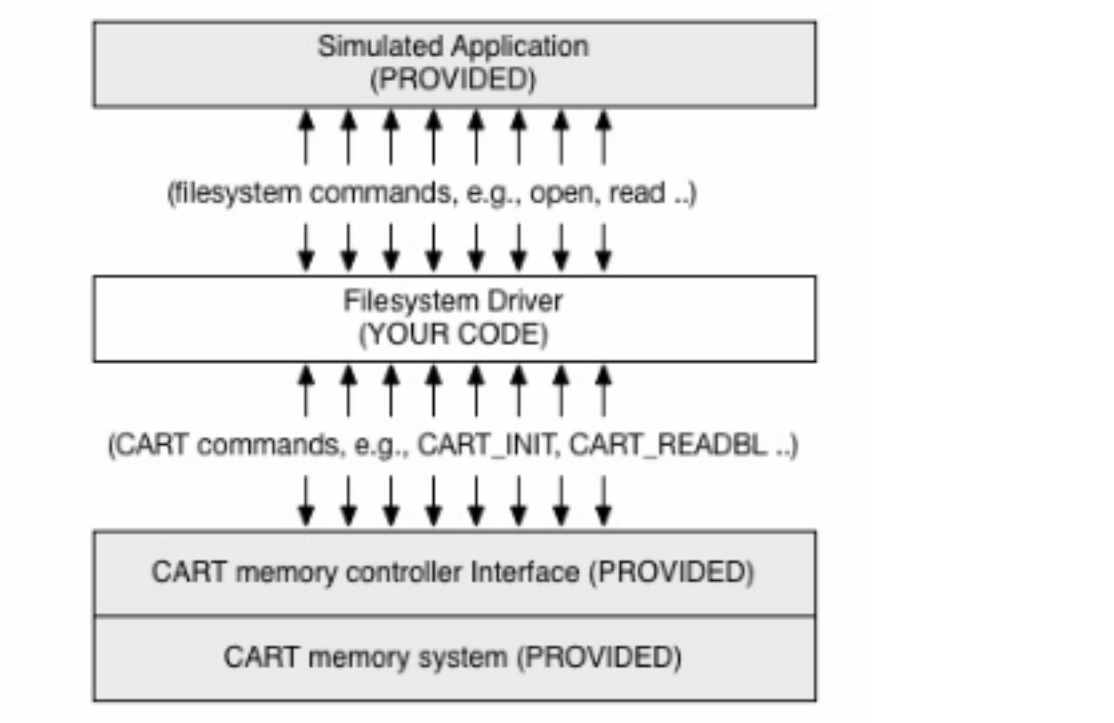
The purpose of this lab is to create a user-space device driver for a in memory filesystem that is built on top of a hierarchical random access memory system (HRAM). At the highest level, you will translate file system commands into memory frame level memory operations (see memory interface specification below). The file system commands include open, close, read, write and seek for files that are written to your file system driver. These operations perform the same as the normal unix I/O operations, with the caveat that they direct file contents to the HRAM storage device instead of the host filesystem. The arrangement of software is described below.

System and Project Overview

You are to write a basic device driver that will sit between a virtual application and virtualized hardware. The application makes use of the abstraction you will provide called HRAM "cartridge" memory system (CART). The design of the system is shown in the figure below.

Described in details below, we can see three layers. The CART application is provided to you and will call your device driver functions with the basic UNIX file operations (open, close, read, write, seek). You are to write the device driver code to implement these file operations. Your code will communicate with a virtual controller by sending opcodes and data over an I/O bus.

All of the code for application (called the simulator) and compiled binary for CART memory system is given to you. Numerous utility functions are also provided to help debug and test your program, as well as create readable output. Sample workloads have been generated and will be used to extensively test the program.



The Cartridge Memory System Driver (CART)

You are to write the driver that will implement the basic UNIX file interface 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/seek filesystem functions. You will also need to implement the poweron and poweroff function that initializes your filesystem's internal data structure and clean up when the system is shutting down while also make calls into low-level operation on the device.

The functions you will implement are:

- `int32_t cart_poweron(void)`: This function will initialize the CART interface basic filesystem. To do this you will execute the init CART opcode and zero out all cartridges of the CART filesystem. You will also need to initialize your internal data structures that track the state of the filesystem.
- `int32_t cart_poweroff(void)`: This function will shut down the CART interface basic filesystem. To do this you will execute the shutdown CART opcode and close all the files. You will also need to cleanup your internal data structures that track the state of the filesystem.
- `int16_t cart_open(char *path)`: This function will open a file (named path) in the filesystem. If the file does not already exist, it should be created and set to zero length with read/write position also pointing at zero. If it does exist, it should be opened and its read/write position should

be set to zero to the beginning of the file. 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.

- `int16_t cart_close(int16_t fd)`: 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.
- `int32_t cart_read(int16_t fd, void *buf, int32_t count)`: This function shold read count bytes from the file referenced by the file handle starting at the current position. If there are not enough bytes left in the file, the function should read to the end of the file and return the actual 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 fild handle is bad or the file was not previously open.
- `int32_t cart_write(int16_t fd, void *buf, int32_t count)`: This 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.
- `int32_t cart_seek(int16_t fd, uint32_t loc)`: This function should set the current position into the file to loc, where 0 is the beginning 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.

The key is to figure out what datastructure you need to keep track of all necessary information about the files. You will need to

- Maintain information about current files in the system. For each file you will need to keep track of if it is open, file size, current position, as well as info about all the frames we used to store this file in the CART memory system.
- You need to figure out a way to allocate frames in the CART memory system whenever a file needs it to place data.
- You need to copy data into and out of the CART memory system as needed to serve reads and writes. You have to be aware that the low-level CART memory system only supports read and write of a complete frame of fixed size while your read and write function needs to support read and write of various sizes that might only involve partial frame or multiple frames.

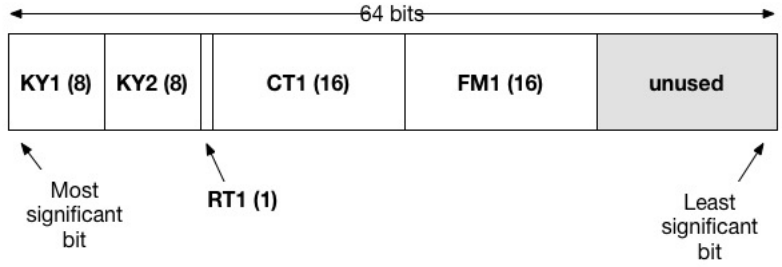
The Cartridge Memory System (CART)

You will implement your driver on top of the CART memory system (which is referred to throughout simply as the CART). The CART is a hierarchical memory system which means that there are multiple levels of memory organization. In the case of the CART, there are three levels: the system as a while consists of multiple cartridges, each of which contain multiple frames, and each frame is a fixed byte sized memory block. Some key facts of the system include (see `cart_controller.h` for definitions):

- There are `CART_MAX_CARTRIDGES` cartridges in the system. They are numbered starting from 0.
- Each cartridge contains `CART_CARTRIDGE_SIZE` frames. They are numbered starting from 0.
- A frame is a memory block of `CART_FRAME_SIZE` bytes.

You communicate with the memory system by sending code 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 (note that top is bit 0)
------   -------------------------------
0-7     - KY1 (Key Register 1, 8bits)
8-15    - KY2 (Key Register 2, 8bits)
16      - RT1 (Return code register 1, 1bit)
17-32   - CT1 (Cartridge register 1, 16bits)
33-48   - FM1 (Frame register 1, 16bits)
49-63   - UNUSE 15bits
```



The following opcodes define how you interact with the controller.

Register	Request Value	Response Value
CART_OP_INITMS - Initialize the memory system		
Register: **KY1**	CART_OP_INITMS	CART_OP_INITMS
Register: **RT**	**N/A**	0 if successful, -1 if failure
CART_OP_BZERO - zero the currently loaded cartridge		
Register: **KY1**	CART_OP_BZERO	CART_OP_BZERO
Register: **RT**	**N/A**	0 if successful, -1 if failure
CART_OP_LDCART - load a cartridge		
Register: **KY1**	CART_OP_LDCART	CART_OP_LDCART
Register: **RT**	**N/A**	0 if successful, -1 if failure
Register: **CT1**	cartridge number to load	**N/A**
CART_OP_RDFRME - read a frame from the current cartridge		
Register: **KY1**	CART_OP_RDFRME	CART_OP_RDFRME
Register: **RT**	**N/A**	0 if successful, -1 if failure
Register: **FM1**	frame number to read from	**N/A**
CART_OP_WRFRME - write a frame to the current cartridge		
Register: **KY1**	CART_OP_WRFRME	CART_OP_WRFRME
Register: **RT**	**N/A**	0 if successful, -1 if failure
Register: **FM1**	frame number to write to	**N/A**
CART_OP_POWOFF - power off the memory system		
Register: **KY1**	CART_OP_POWOFF	CART_OP_POWOFF
Register: **RT**	**N/A**	0 if successful, -1 if failure

Note that the UNUSED parts of the opcode should always be zero, as should any register not explicitly listed in the specifications below. If the frame argument is not specified, then it should be passed as NULL.

To execute an opcode, create a 64 bit value (uint64_t) and pass it any needed buffer (for read/write) to the bus function defined in `cart_controller.h`:

```
CartXferRegister cart_io_bus(CartXferRegister regstate, void *buf);
```

The function returns packed register values that is as listed in the ”Response Value” above.

Hand Out Instructions

Start by copying the file `cartlab-handout.tar` from Canvas to the protected directory (the *311 directory*) in which you plan to do your work. Then do the following:

- Type the command `tar xvf cartlab-handout.tar` to expand the tarfile.
- Type your name and PSU ID in the header comment at the top of `cart_driver.c`.
- Start editing your `cart_driver.c` file.

- Type the command `make` to compile and link some test routines.
- Test your code similar as below, preferrably in the order given.

```
$./cart_sim workload/test1.txt
$./cart_sim workload/test2.txt
$./cart_sim workload/test3.txt
$./cart_sim workload/test4.txt
$./cart_sim workload/test5.txt
$./cart_sim workload/test6.txt
```

Evaluation

The evaluation for this assignment is different from in the past. Each test is worth certain points that you can earn to a maximum of 50 points with distributions described as below. However, we will manually grade by reading the implementation of each component, and bad design/code style will result in the loss of the correctness point up to a deduction of up to 25 points. As usual, code that compiles with any error or warning messages will result an automatic zero for the project.

4 test1.txt:You will need to impement poweron/poweroff/open/close/write mostly correctly to be able to pass test1.

1 test2.txt:After passing test1, you will have to implement seek properly to pass test2.

2 test3.txt:You will need to impement multiple writes/overlapping writes correctly to pass test3.txt.

2 test4.txt:You will need to impement read properly to pass test4.txt.

6 test5.txt:You will need to implement support for simultaneously multiple opened files and read/write operation too these files.

35 test6.txt:You will need to implement support for writing and reading larger files that need more than one frame.

-25 manual grading: proper indentation; detailed block comments, in particular for any internal data structures and helper functions; responsible use of memory (free memory if you use malloc or any other utilities to allocate memory for any internal data structures).

Hint and assumptions

You may assume that the following is true for your program:

- Maximum number of files is 1024
- Maximum length of file name (including '
  0') is 128

- Maximum size of any file is 102400 (fit in 100 frames)
- Once a file is created on your system, you can not delete it or shrink its size, though you could overwrite old content by seeking to the beginning and write new content.

Carefully design your internal data structure. You will need internal data to keep track of these info about each file created, indexed by a unique file handle:

- file name (consider an array)
- file size
- file current position
- information about the frames we used from CART memory system to store the content of this file. Keep in mind that each frame in CART is identified by Cartridge id and Frame id.
- file open status

Carefully design internal data and helper function that will allocate a frame for a file. You may consider the most straightforward linear allocation algorithm where we allocate from cartridge 0 frame 0, and cartridge 0 frame 1, ... and move on to the next cartridge when we use up frames in cartridge 0. Since our filesystem does not support deleting a file or shrinking a file, so you don't have to worry about recycling used frames.

For read and write, keep in mind that CART file system only supports reading and writing a whole frame, so for a write that only write to part of the frame, you will need to read out the frame first, write your partial data into this frame, and then write the whole frame back. If you don't do this, then some write will overwrite existing data with unitialized garbage data when writing a partial frame.

Hand In Instructions

- Make sure you have included your name and PSU ID in the header comment of `cart_driver.c`.
- Submit your `cart_driver.c` file to Gradescope.

Good luck!