Programming Assignment #1: Disk emulator

COMP310/ECSE 427: Fall 2013

Basic disk emulator

In this assignment, you are required to implement a program that provides access to blocks of a virtual disk. A disk contains consecutive blocks, each with the same size of BLOCK_SIZE bytes. You are supposed to use a standard Linux file to store all the blocks of the emulated disk. Here is the specification for the functions you need to implement:

- int mydisk_init (char const *file_name, int max_blocks, int disk_type);

Initialize an emulated disk with *max_blocks* blocks in a Linux file named *file_name*. If the file already exists, override it. Otherwise, create a new one. The file has to be filled with zeros (real zero not the zero character, see reference) after initialization. *disk_type* indicates if the emulated disk is a HDD(0) or a SSD(1). This function returns 0 on success and returns 1 if any error happens.

- int mydisk_read_block(int block_id, void *buffer);
- int mydisk_write_block(int block_id, void *buffer);

These two are the fundamental block access functions. *block_id* should be no larger than *max_blocks*. Buffer is guaranteed to have exactly BLOCK_SIZE bytes. The virtual disk is a flat space where the first BLOCK_SIZE bytes is the first block (with block_id == 0). These two functions perform read/write for the given block id and read the block into the buffer (or store the buffer content to the disk). Note that the mydisk_write_block() will overwrite the content store at the particular block. These two functions return 0 on success and return 1 if parameters are invalid. You must use standard C library functions such as *fread(), fwrite(), fseek()* in your implementation.

- int mydisk_read (int start_address, int nbytes, void *buffer);
- int mydisk_write (int start_address, int nbytes, void *buffer);

These two functions are built on top of previous two and read/write multiple blocks in one shot. Once initialized, the valid disk address ranges from 0 to max_blocks * BLOCK_SIZE-1. These two functions read/write *nbytes* bytes starting at *start_address* into memory. *buffer* is the source/destination memory buffer, which is guaranteed to have exactly nbytes. You must use mydisk_read_block() and mydisk_write_block() in your implementation. DO NOT directly use *fread(), fwrite(), fseek()* for convenience. Note that the *start_address* and the end address (start_address + nbytes) may appear in the middle of a block. So you need to truncate the block being read or fill the block being written. Before performing any actual action, you are required to check the parameters, i.e. 0<=start_address <= start_address + nbytes < max_blocks*BLOCK_SIZE. Return 0 on success and return 1 if parameters are invalid.

- void mydisk_close();

Close the virtual disk (paired with mydisk_init()). Do the cleanup if necessary.

Caching

Caching helps to reduce access latency. The in-memory cache buffer keeps a fixed number of blocks (usually fewer than all the blocks on the disk). If a request block is in the cache, read/write operations will retrieve the block from the cache (cache hit). Otherwise (cache miss), the request block will be fetched from the disk as usual. Caching helps to reduce the number of real disk reads/writes.

On cache miss, the newly fetched block being either takes a free cache entry or replaces an old one if the cache is ready full. You are required to implement the First-In-First-Out (FIFO) replacement scheme (see more in the reference).

The write operation is slightly different. When the write hits in cache, the block will not be written to the disk immediately. Instead, it stays in the cache and is marked as a "dirty" block. When a "dirty" block is to be evicted from the cache, it will be written back to the disk. This kind of cache is named as a "write-back" cache.

In summary, you are required to implement a standard write-back FIFO cache. The specification is as follows:

- int init_cache(int nblocks);

Allocate the cache buffer with *nblocks* blocks. After calling this function, the cache is enabled.

- void close_cache();

Flush all the dirty blocks to disk and free the cache buffer. After calling this function, the cache is disabled.

- void *get_cached_block(int block_id);

Find the cache entry that stores the given block id. Return the cached block. If the block is not in the cache, return NULL.

- void* create_cached_block(int block_id);

Create a cache entry for the block id. This function should insert a new entry into the cache (possibly by kicking another one). The memory for the new block should be obtained by malloc() and returned to the caller.

- void mark_dirty(int block_id);

Mark the entry for the given block id in the cache as dirty.

- int mydisk_read_block(int block_id, void *buffer);
- int mydisk_write_block(int block_id, void *buffer);

Caching is on a block basis. So you need to extend mydisk_read_block() and mydisk_write_block() to enable caching. The idea is to check if the request block is in the cache buffer. If so, return the block from the cache buffer. Otherwise, perform the normal disk read/write. These two functions return 0 on cache miss, 1 if parameters are invalid, -1 if cache hit.

Note: Remember to check if cache is enabled before using the cache. By doing so, your code should still work if the cache is turned off (as your baseline test).

Latency calculation

Due to different disk types and caching policies, read/write operations incur different latencies. In this part, you will calculate and report access latency. As mentioned, caching can reduce latency. If a request block is found in the cache, it incurs MEMORY_LATENCY. For cache miss, the below table summarizes the calculation of latency for two disk types. For a detailed explanation of the composition, please refer to the reference:

	Read	Write
HDD	HDD_SEEK+nblocks*HDD_READ_LATENCY	HDD_SEEK +nblocks*HDD_WRITE_LATENCY
SSD	nblocks*SSD_READ_LATENCY	nblocks*SSD_WRITE_LATENCY

For example, assume BLOCK_SIZE=512. Reading 450 bytes from address 100 actually touches two blocks (0 to 511 and 512 to 1023). If one hits in the memory and one goes to the HDD, the total latency for this operation is 1*MEMORY_LATENCY + HDD_SEEK + 1*HDD_READ_LATENCY.

You are required to extend mydisk_read() and mydisk_write() to report their latency. At the end of these two functions, do the calculation and call report_latency() to report the latency for this operation.

Note:

- You do not need to implement report_latency() in test.c. But you can print out the latency when debugging.
- You can tell if cache hit/miss from the return value of mydisk_read_block() or mydisk_write_block().
- You can find the disk type when the mydisk_init() is called.

Testing

There are five other files being delivered with this document: mydisk.h, mydisk.c, caching.c, test.c, and Makefile. Your code should appear mydisk.c and caching.c. Please search for the "TODO" marks in the source code for more instructions. Please leave other files unchanged.

To test your implementation, type "make" command in the Linux terminal. If the code contains compile errors, the error message will be displayed and no executable will be generated. Otherwise you should get an executable file named "mydisk". Run this program and you will get a bunch of "PASS"/"FAILED" showing your results on test cases.

Be careful about common programming errors such as segmentation faults and memory leak.

Grading

- You are required to use the handin script to submit your source code (http://socsinfo.cs.mcgill.ca/wiki/Handin). Our grading script runs on the Linux workstation and minimizes TAs' intervention. No handin, no score.
- Code with compile errors will automatically get a score of ZERO if you do not make any confession.
- Code that does not pass all test cases or has infinite loop or crashes (due to segmentation fault, free exception, etc.) will get partial score.
- Code that passes all test cases DOES NOT GURANTEE a full mark. You will be tested against more in-house test cases.
- Coding style: although style is subjective, every programmer should write code in a readable manner. We recommend you to use standard style before you start to develop your own one. Coding style worth one point. Although small, it counts as the last point towards a full mark. The judgment is subjective, too. However, we ensure every grader agrees on K&R style. See the reference for details.

Reference:

- **C Language:** C in a Nutshell, by Peter Prinz. If you are not familiar with a language, always keep a language reference book at hand and keep Google and stackoverflow accessible.
- **Zero and zero character:** http://stackoverflow.com/questions/7578632/what-is-the-difference-between-char0-and-0-in-c
- **Coding style (complete, not too long):** http://www.cas.mcmaster.ca/~carette/SE3M04/2004/slides/CCodingStyle.html
- **Coding style (brief):** http://en.wikipedia.org/wiki/Indent_style#K.26R_style
- **FIFO:** http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/fifopagereplacement.htm
- **Disk access latency:** http://en.wikipedia.org/wiki/Hard_disk_drive_performance_characteristics