Programming Assignment #1: Disk emulator

COMP310/ECSE 427: Fall 2012

1. Basic disk emulator

In this assignment, you are required to implement a disk emulator that provides basic interfaces to access blocks on the emulated disk. Your emulator is supposed to use a standard Linux file to store all the blocks of the emulated disk. More specifically, the emulated disk consists of max_blocks blocks. Each block has a pre-defined size BLOCK_SIZE. The required interfaces are as follows:

- int init_disk(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. Once initialized, the valid disk address ranges from 0 to max_blocks * BLOCK_SIZE-1. *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.

NOTE: pls be careful when you pass values to this function, the value of *max_blocks* and disk_type must be meaningful

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

start_address is the beginning disk address of the operations; *nbytes* is the number to bytes you read/write; *buffer* is the source/destination memory buffer of your reading/writing operation. Before performing actual read/write, you are required to check the parameter, i.e. 0<=start_address <= start_address + nbytes < max_blocks*BLOCK_SIZE. The buffer is guaranteed and requires no validation. These two functions return 0 on success and returns 1 if parameters are invalid.

- void close_disk(char const *file_name);

Close the file that stores the blocks. Do the cleanup if necessary.

Note: Operations contained between an init_disk and close_disk pair is called a "session". A session is associated with the configuration given in its init_disk. At any time, your emulator should only have one session. But you are supposed to support multiple sessions (i.e., after calling close_disk, user can call init_disk to start a new session).

Please use standard C library functions such as fread(), fwrite(), fseek() in your implementation.

2. Latency emulation

In addition to basic functionality, you are also required to simulate the access latency. In this assignment, two types of disk are simulated and the below table summarizes the calculation of latency (*nblocks* are the number of blocks being accessed):

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

From the above table, you can find out that HDD operations require a base seek time (HDD_SEEK). It means that when you read/write from/to a random range of the disk, the HDD_SEEK overhead is imposed for each time you move between sectors. That is why sequential read/write on HDD is much faster than random operations. On the contrary, SSD performance only depends on the data size being accessed (ideally).

Constants in the above table are given. In your read/write functions, nblocks should be computed first and then the latency. Please update the variable HDD_latency and SSD_latency for each time you read/write from/to disk or cache. CACHE_READ_LATENCY and CACHE_WRITE_LATENCY are also given. For simplification, you don't need to consider how the sequence of writing/reading operations affect the performance of HDD disk, HDD_SEEK is imposed for each mydisk_write and mydisk_read.

Note: Be careful when calculating *nblocks*. For example, accessing 450 bytes starting from address 100 actually touches two blocks.

3. Caching

Caching is a common way to boost access performance. Cache contains a given number of blocks (usually fewer than all the blocks on the disk). When the cache is enabled, read/write operations will first check if the request block is in the cache. If the requested blocks are in the cache (cache hit), then the content is read/write from/to the cache. Otherwise (cache miss), the requested blocks are fetched from the disk and occupy one of the cache entries. On cache miss, the new block either takes a free cache entry or replace an old one. Especially, different implementations of cache have varying policy for write miss. In our cache, we implement write-allocate and fetch-on-write policy, which means that when there is a write-miss, we write the data to disk, and then allocate a block in the cache for missed block and fetch the written disk from the disk. You are also required to implement the Least Recently Used (LRU) replacement scheme (see more in the reading list). When one block is evicted and another takes its place, you also need to check if the evicted block is modified. If the evicted block is dirty (modified), you need to write back to the disk block. In summary, you are required to implement a standard write-back LRU cache. The interfaces exposed in this part are 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();

Free the cache buffer and do cleanup. After calling this function, the cache is disabled.

For the codes of querying and updating the content of cache, you should add them to your basic implementation of read/write operations in the first task.

Note: Be aware to check if cache is enabled before using the cache.

Your caching code should also track each cache access (block-wise). At last, when calling *close_cache()*, you should report the miss rate (the ratio of the number of cache miss against the number of total access) using given function *report_cache_perf()*.

4. Testing and Grading

There are seven other files being delivered with this document: mydisk.h, mydisk.c, mycache.h, mycache.c, myqueue.h, myqueue.c test.c, and Makefile. You should only change mydisk.c, mycache.c and myqueue.c. You can 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 Linux terminal to generate the executable, and type "make test" to run the program. If any compile/runtime errors happen, the error message will be displayed on screen. Otherwise you should see a bunch of "PASS"/"FAILED" showing your results on test cases.

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

DO NOT handle in code that contains compile errors!

5. Reading List:
   Cache: Computer Organization and Design, Revised Fourth Edition, Fourth Edition: The Hardware/Software Interface, http://www.amazon.com/Computer-Organization-Design-Revised-Edition/dp/0123747503/ref=sr_1_1?ie=UTF8&qid=1346817070&sr=8-1&keywords=computer+organization+and+design
   C Language: C in a Nutshell, http://www.amazon.com/C-Nutshell-In-OReilly/dp/0596006977
   LRU: http://www.cis.upenn.edu/~palsetia/cit595s07/FIFOLRUExample.pdf

NOTICE: the detailed requirements for each function can be found in source files

And you need to use handin script to submit your programs(http://socsinfo.cs.mcgill.ca/wiki/Handin), the grading script will be run in the linux workstation of our department