Overview

Introduction

Design & implement a virtual hard disk with variable block and disk sizes (specified at creation time). Drive must be persistent (use a data file); drive should be able to store other data files persistently; data files should be read from drive and still remain intact.

The virtual hard disk code is written in C and is contained in one file (vhd.c).

General Disk/Block Behavior & Layout

Minimum disk size is 1 MB; maximum disk size is 128 MB.

Minimum block size is 512 bytes; maximum block size is 8192 bytes.

User defined disk and block sizes could possibly be adjusted to prevent wasted space (if last block created does not fit on disk then it would have been truncated otherwise); see ***sanitize***.

It helps to visualize, at a high level, interpreting the disk from left (beginning) to right (end). At a lower level, big endian coding is the interpretation when accessing disk sections such as the super-block, i-nodes, and free space management; these sections will also be in the indicated order on the disk (left to right).

Super-block

Guaranteed to be located and contained within the first block (block 0), to the exclusion of all other data. Contains four data fields of 8 bytes each indicating (in order):

1. number of blocks
2. block size (in bytes)
3. block number indicating start of i-node section
4. block number indicating start of free space management section

The super-block is accessed by the functions ***vhd_create*** and ***vhd_mount***.

I-nodes

Begins at block 1 of virtual drive. Since the number of blocks is determined at run time, on the fly quantity is required of the i-nodes. Each i-node will contain:

1. file size in bytes
2. 8 direct pointers
3. 1 single indirect pointer
4. 1 double indirect pointer
5. 1 triple indirect pointer

Each field in the i-node is 8 bytes, totaling to a size of 96 bytes. Indirection blocks are generalized to 8 fields of 8 bytes each (64 bytes in total), no matter where in the indirection chain the

block is located. Each i-node is assigned an equal amount of indirection blocks and this amount remains static. Thus, an i-node can track a file of up to 592 blocks; the aggregate size of an i-node (including all associated indirection blocks) is 4768 bytes.

I-nodes and their corresponding indirection blocks are accessed by the following functions: ***vhd_create***, ***vhd_open_file***, ***vhd_write_file***, ***vhd_read_file***, and ***vhd_remove_file***. Those functions may in turn use the static functions ***get_inode***, ***set_inode***, ***get_indirect_block***, and ***set_indirect_block***.

Since disk and block sizes will vary, so too will maximum file size and maximum file quantity; different files will also not share blocks. Thus, it is guaranteed that a valid disk will have at least two i-nodes, one for the root directory and the other for a regular file.

Free Space Management

Free space management is implemented as a bit-map, mapping blocks to bits. A 0 indicates a block is in use, a 1 indicates that a block is free for use. Accessed by the functions ***vhd_write_file*** and ***vhd_remove_file*** which in turn call the static functions ***set_block_use*** and ***get_block_use*** which finally call ***byte_pack***.

File Storage & Directories

The root directory will always be tracked by the first i-node. The root directory is limited to one block in size and is laid out sequentially in ordered pairs of i-node number (8 bytes) and file name (16 bytes).

Functionality for additional directories is not implemented at present. Potential implementations could include adding a flag to the i-nodes specifying whether an i-node is a file or a directory. This entails that constants for i-node size will have to be adjusted as well as implementing generalized accessor functions for directories (see ***vhd_open_file*** for an idea of how to do that).

The root directory is accessed by the functions ***vhd_open_file*** and ***vhd_remove_file***.

Public API

vhd_create

Specify new drive name, disk size, and block size. Fail if drive of same name already exists. Returns maximum disk_size (which may be less than specified disk size), 0 if fail.

Calls ***sanitize*** to parse and clean up user input. Writes super-block data, a properly initialized free space management bit map, and sets up each i-node's indirection pointer chain. Each actual block address field found at the ends of the indirection chains are set to a sentinel value of -1 cast to *size_t*. Also writes starting root directory structure. Calls on ***get_indirect_block***, ***set_indirect_block***, ***get_inode***, ***set_inode***, ***set_block***, and ***set_block_use***.

vhd_mount

Specify name of existing drive. Fail if drive does not exist. Returns a ***vhd*** pointer to a virtual hard disk, NULL if fail.

Processes super-block to create a ***vhd*** opaque (to the user) data structure. All necessary data for accessing disk can be calculated from the four data fields in the super-block.

vhd_open_file

Specify virtual drive and file name. If file does not exist a new one will be created. Returns a file descriptor (which is essentially the number of the i-node that tracks the file).

Calls on ***get_inode*** and ***set_inode***.

vhd_file_size

Specify virtual drive and file name. If file does not exist 0 is returned, otherwise the size of the file in bytes is returned.

Calls on ***get_inode*** and ***set_inode***.

vhd_write_file

Specify virtual drive, file descriptor, buffer containing bytes to be written, number of bytes to be written, and the byte offset (where to start writing in the file). Returns number of bytes written.

This function is longer than it should be; subdividing it into further functions is suggest. Any ugliness stems from tracking through an i-node's various indirection pointers; pay close attention to how the starting block and starting byte are determined. Designed to fall through to deeper levels of indirection as necessary, up to the requested number of bytes written or maximum file size. If a new block is needed it will be selected at random from the available free blocks.

This function calls on ***get_indirect_block***, ***set_indirect_block***, ***get_inode***, ***set_inode***, ***get_block***, ***set_block***, ***get_block_use***, and ***set_block_use***.

vhd_read_file

Specify virtual drive, file descriptor, buffer for bytes to be read into, number of bytes to be read, and the byte offset (where to start writing in the file). Returns number of bytes read.

This function is longer than it should be; subdividing it into further functions is suggest. Any ugliness stems from tracking through an i-node's various indirection pointers; pay close attention to how the starting block and starting byte are determined. Designed to fall through to deeper levels of indirection as necessary, up to the requested number of bytes read or end of file.

This function calls on ***get_indirect_block***, ***get_inode***, and ***get_block***.

vhd_close_file

Specify virtual drive and file descriptor. A file must be closed before it can be removed from disk..

vhd_remove_file

Specify virtual drive and file name. Removes file entry from root directory, shifting all entries that occur after it over by one. Also not the prettiest of functions as it must traverse the indirection pointer chains and free up corresponding blocks found at the end of each chain.

Calls on ***get_indirect_block***, ***set_indirect_block***, ***set_block_use***, and ***set_inode***.

vhd_unmount

Specify address of pointer to virtual drive. Sets pointer to virtual drive to NULL. Essentially frees memory allocated from the heap.

vhd_state

Specify point to virtual drive. Prints drive data to standard out (debugging purposes).

Static Functions

byte_pack

Specify byte to write to, which bit to write to (least significant bit that is altered), width of bits altered, and the value of bits to write. Returns altered byte.

get_block

Specify file descriptor of virtual disk, buffer to write to, block number, and block size in bytes. Buffer should be block size in bytes.

set_block

Specify file descriptor of virtual disk, buffer to write from, block number, and block size in bytes. Buffer should be block size in bytes.

set_block_use

Specify file descriptor of virtual disk, value (0 for in use, 1 for free), block number, block number of start of free space management, and block size in bytes.

get_block_use

Specify file descriptor of virtual disk, block number, block number of start of free space management, and block size in bytes. Returns 0 if block in use, 1 if block is free.

get_inode

Specify file descriptor of virtual disk, buffer for i-node, i-node number, and block size in bytes.

set_inode

Specify file descriptor of virtual disk, buffer for i-node, i-node number, and block size in bytes.

get_indirect_block

Specify file descriptor of virtual disk, buffer for indirection block, indirection block number, block size in bytes, and total number of i-nodes.

set_indirect_block

Specify file descriptor of virtual disk, buffer for indirection block, indirection block number, block size in bytes, and total number of i-nodes.

Sanitize

Specify buffer for disk size in bytes and buffer for block size in bytes. These parameters must have some value specified. If values would lead to wasted space they are altered to prevent this. Further specified buffered parameters include number of blocks, number of bitmap blocks, and number of i-node blocks. These values are derived from (sanitized) disk size and block size and stored in their corresponding buffers for further use.

Conclusion

Designing this virtual disk module has given some insight into what it may take to write a virtual hard drive for some virtual machine (mainly examining how to implement variable sized disk and block sizes). There is heavy use of the *size_t* data type as it provides for truly large drive sizes if desired; for our purposes it seems like overkill but it does provide for maximum sizes allowed by what ever machine is running this module.

If given more time I would further refactor some specific functions such as ***vhd_write_file*** and ***vhd_read_file*** into a collection of smaller functions for readability and maintainability. Having said that I have worked out some of the more serious bugs encountered in previous versions; coupled with an interactive test program it is perfectly demonstrable for files below about 30,000 bytes or so (there is an unresolved bug that produces corrupted files beyond this size; no known solutions as of yet).