COL 733 Assignment 3
Disk Virtualization

Shubham Rawat (2013CS10258)
Sahil Jindal (2013CS10253)
Prakhar Gupta (2013CS10245)
Anupam Khandelwal (2013CS10212)

13th September 2016

1 Consolidation and Partitioning

1.1 Random Read/Write to Blocks

In this part, a class of BlockInfo in maintained storing the size of the block info object. Two arrays are maintained in the main class (VFS) - one of size 200 and the other of size 300 which are collectively used to store the data. Another array containg the blockInfo of all the 500 blocks in also maintained in VFS class.
While writing to a block, its id is confirmed to be valid (1 <= id <= 500), length of block info is checked to be less than 100 bytes. Block is retrieved from one of the arrays based on the lock number (from array1 if id<200 else from array2), the blockinfo is written at the block and the "free" variable in its metadata is set to false.
While reading from a block, id of the block is confirmed valid and checked that the "free" variable in its metadata should be false. Then block number is resolved as to from which array we need to retrieve the data. Then the block information is stored in the object passed as parameter of the readBlock function.
For testing purposes, multiple read/write of blocks covering most of the corner cases like, invalid block number, length of block info >100 while writing, rewriting over blocks etc.

1.2 Create and Delete Disk

Various classes maintained in this subsection are:

- Blockinfo:
  - size: Size of block
  - free: if the block does not have data written to it
  - unallocated: whether block is allocated to a disk created by user or not
  - disk_id: id of disk to which block is allocated
- DiskInfo:
  - size: Size of the disk
  - start: mapping of first block of the disk to the position in total block space
  - disk_blocks: This function returns the range of blocks allocated to this disk
- VFS:
  - array of size 200 blocks
  - array of size 300 blocks
  - Array storing metadata for each block
  - Array containing metadata for each disk

To create disk, the API request specifies the id and size of disk required. The program checks if any disk with given id already exists or if the disk can be allocated space. If the tests pass, diskInfo object is stored at position id in the disk_metadata array containing the starting block number and size of the disk. Also the block metadata for every block in the range is changed by setting the unallocated variable to false and setting the disk_id variable to the given id.
To delete a disk the API request specifies only the id of the disk. If a disk is found with the given id, its metadata is removed from the metadata array, and block metadata for every block in the range is reset to default.
To write to a block, we are given disk id, block number w.r.t. the disk, and info to be written at the block. The program checks if the disk id given has actually been created and block number specified lies within size of the disk. Then the address of the block is converted to global address and is written to as in previous subsection.
Similarly reading of block takes place.
For testing purposes various create/delete/read/write operations were run along with some invalid operations such as specifying disk id which does not exist, while reading block/deleting disk. Here the problem of disk fragmentation is not yet handled and hence if the program does not find a continuous chunk of free memory available it return out of memory and is unable to create specified disk. This will be handled next.

1.3 Handling disk fragmentation

Here, the disk info object contains an array of blocks which are allocated to it as compared to start and size in previous case as the blocks allocated now may or may not be continuous. In VFS class an additional array is maintained containing a linked list of free blocks.
To create a disk, id and size are checked to be valid. Then size number of blocks are popped from the free blocks list and put to the blocks array in disk info metadata.
To delete a disk, the blocks are popped from the disk info metadata and added to the free blocks list.
Reading/Writing to blocks is similar to the previous cases.
For testing purposes various create/delete/read/write were performed. Also create with not enough contiguous space available was performed which created a disk successfully thus solving the problem of fragmentation.

2 Block Replication

Here block info class is modified to contain variables "error" and "replication". "error" stores if the block has had a read error and is corrupted, and "replication" variable stores the block number of the replica of the block (null if no replica stored yet).
In case of block replication, if a user asks to create a disk of size s, then internally we create a disk of size 2*s to store the replicas(Assuming replication factor to be 2 for every block). The blocks 1 to s are the original blocks accessible to user, and the blocks s+1 to 2*s are used to store replicas. Replica of block i is stored in the ith free block in section s+1 to 2*s.
Delete disk is same as in previous section.
While writing to a block, the program performs the validity checks for block id, size etc. and also checks the error bit of the corresponding block. Writing data to a block is same as previous section. Now a free replica block is obtained for the block from replicas section and same information is written to it. The "replication" variable of the given block is set to point to the replica.
While reading from a block, validity checks are performed initially. Then data is read from the original block initially same as reading block in previous section. If this read operation fails, the information is read from the replica of this block and the original block is marked corrupted.
If the replica read is a success, then the mapping of virtual block number to physical block number is updated to point to its replica i.e. its replica is made the new original block. So that initial read/write whenever a user asked is performed on this block. Now a new free block is obtained from replica sections and replica of current block is maintained at that block.
If replica read fails, we have no way to obtain the information and both the blocks are marked corrupted and user is notified that information is lost.
For testing purposes, multiple read writes are performed so that we get enough read errors. Also blocks are overwritten after reading once to veriy if user can write to blocks for which the original is corrupted but replica is intact.
Following are the figures obtained on running the tests:

- # read errors for original block = 7
- # read errors for both original and replica blocks = 1

After overwriting:

- # read errors for original block = 13
- # read errors for both original and replica blocks = 2

3 Snapshoting

In this part, an array of snapshots is maintained in the disk info metadata class. Create/Delete/Read/Write operations are same as in section 1.
To take a snapshot, create_checkpoint API is used taking disk id as its parameter. For each block in the disk, its metadata and information are stored in the snapshot in form of an array of pair objects. This snapshot is then added to the array of snapshots maintained in the disk info object and the snapshot id (size of snapshots array - 1) is returned to user for further use of this checkpoint.
To rollback to a particular snapshot, disk id and snapshot id are verified to be valid. The snapshot corresponding to the given snapshot id is loaded from the array. For each block in the disk, the metadata and information stored in the block is replaced by the metadata and information of the block stored in the snapshot.
For testing purposes, the disk contents were overwritten and then it was rolled back to the checkpoint before the overwriting and hence previous information was successfully retrieved.