COL733: Assignment 3 Disk Virtualisation

Group 1

Avaljot Singh (2016CS50389)
Mayank Singh Chauhan 2016CS50394
Atishya Jain (2016CS50393)
Mankaran Singh (2016CS50391)

September 11, 2019

1 Consolidation and partitioning

1.1 Design philosophy

In this assignment, we create a virtual storage software, that intercepts I/O requests from the programmer and send those to appropriate physical location of the storage devices that are a part of the overall pool of storage in the virtualised environment. To the user, the virtual storage appears like a standard read/write to a physical drive.
We follow a block-based virtual storage where we essentially abstract the logical storage , such as the disk creation of arbitrary sizes, and also disk deletions, from the actual physical memory blocks in a storage device. This enables our virtualisation management software to collect the capacity of the available blocks of memory space and pool them into a shared resource to be assigned to any number of disks, depending on the requirement.
In lieu of the above formalisation, we simulate two disks of size 300 and 200 by arrays of blocks which are data storing objects each of size 100 bytes. As we discussed above, for the programmer, it appears as if he has an access to a disk of size 500, which is not the case actually.

1.2 Implementation

We have the 3 classes each of a block, fragment and Disk. Block object contains the data, Fragment object contains the starting virtual block number (0-500) and the number of blocks in that fragment. At last, Disk object contains all the metadata related to disk like the list of segments it has been allocated. It maintains this state to allow for applying mapping from user_disk virtual space to global virtual space.

Here we describe the methods and functions defined to have disk virtualisation. Following are the API calls that are exposed to the programmer. These methods help us easily implement the high level functions that will be finally exposed the user for use.

```
virtual_to_physical()
read_physical_block(blockNum)
write_physical_block(blockNum)
```

Here are the implementation details of the above methods:

1. *virtual_to_physical* : We maintain a mapping from virtual block number (0-499) to the physical block number within $diskA$ or $diskB$. This dictionary helps the conversion from virtual space to physical space.

2. *read_physical_block*(*blockNum*): The input *blockNum* is in virtual space (0-500) which gets converted to physical space using *virtual_to_physical* mapping after which we read the desired content stored within the block object.

3. *write_physical_block*(*blockNum*): The input *blockNum* is in virtual space (0-500) which gets converted to physical space using *virtual_to_physical* mapping after which we write the desired content within the block object.

The following APIs facilitate creating, deleting, reading and writing over user allocated disks.

```
createDisk(id, size)
deleteDisk (id)
read_block (id, blockNum)
write_block (id, blockNum)
```

Here are the implementation details of the above methods:

1. *createDisk*(*id*, *size*) : We maintain a list of free segments of memory. Initially its a single large fragment of size 500 starting from block number 0. Now, to allocate free memory, we first check in segmented memory if there is a continuous chunk of memory available to satisfy the disk requirements. If yes, then we allocate that segment of memory and remove that chunk from the free memory list. If no, then we go from the end of the disk virtual space, and go on allocating the free segments until the requirements is satisfied. Along with this, there is proper exception handling in case of disk ID already existing or not enough space available.

2. *deleteDisk*(*id*): The most important concern in this is to merge the free memory segments after deleting the disk. We unallocate the fragments and merge to make a list of free memory blocks.

3. *read_block*(*id*, *blockNum*): The input *blockNum* is in virtual space of the user disk allocated. This will be first converted to the virtual space of global disk (0-500) after which we use the API *read_physical_block* which converts to physical space reads the desired content within the block object.

4. *write_block*(*id*, *blockNum*): The input *blockNum* is in virtual space of the user disk allocated. This will be first converted to the virtual space of global disk (0-500) after which we use the API *write_physical_block* which converts to physical space writes the desired content within the block object.

1.3 Testing

1. We tested the APIs by random reads and writes using the read_physical_block and write_physical_block.

2. We also tried to read and write outside the address space on which our code returned an error. Figure 1 shows the logs obtained after testing these operations.

3. Then we tested Disk Creation and Deletion by creating and deleting disks and also keeping a check on unallotted space to the validity of disk operations.

4. Our code also handles fragmentation by storing a list of unallotted disk blocks. Figure 2 shows the logs obtained after testing Disk Creation/Deletion.

5. After that we tested the virtual disk read and write by writing in many blocks on the disk and then reading them.

6. We also re-tested disk creation and deletion operations along with virtual disk read and write. Figure 3 shows the logs obtained after testing these operations.

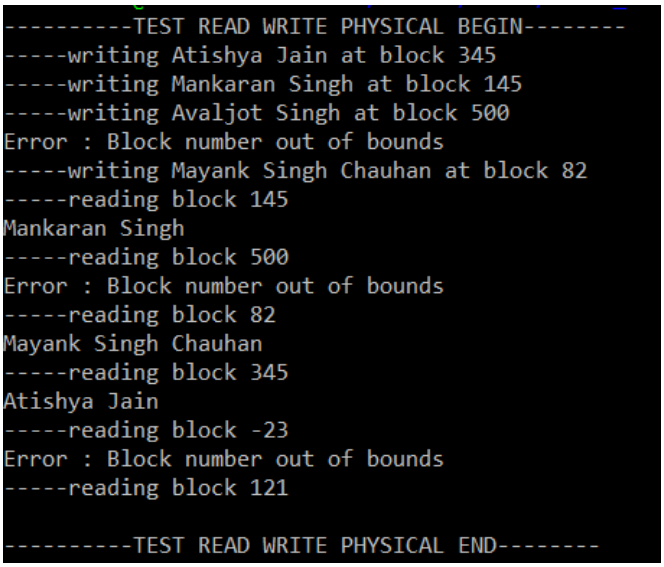

Figure 1: Testing physical Block Read and Write

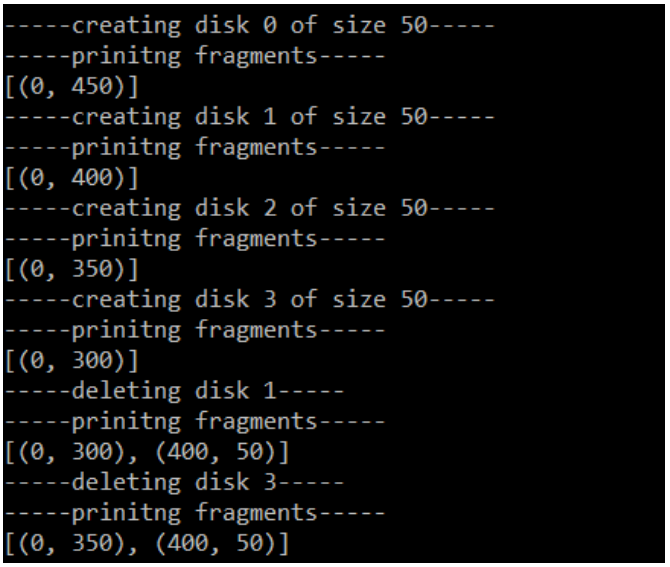

Figure 2: Testing Disk Creation and Deletion

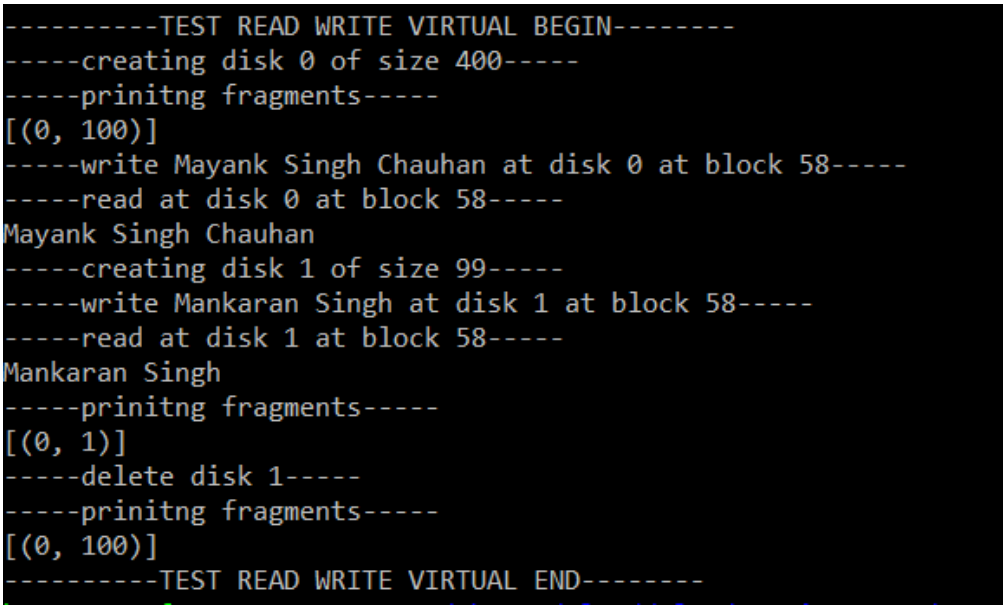

Figure 3: Testing physical Block Read and Write

2 Block replication

2.1 Design philosophy

In this section, we introduce block replication strategy in order to make our system fault tolerant. While allocating blocks for the disk space demanded by the user, we allocate double the no. of blocks that are actually demanded by the user. The user is exposed however, to exactly those no. of blocks that were demanded. The additional blocks are used to store the duplicate of the data stored in the disk. Formalising the above formulation, we allocate two disks, namely, *actualDisk* and *duplicateDisk* of the sizes that were demanded by the user. Both the disks actually store the same data. In all the normal cases, *actualDisk*

serves the read requests. However, in cases where the read operation fails because of some corruption in the $actualDisk$, the real use of $duplicateDisk$ comes into picture. Now, the block in the $actualDisk$ acts like a bad sector of the disk and should never be used again. We, therefore mark it with a special flag such that it will not be allocated to any disk ever again. Also, we need to bring back the replication factor of the data that was stored in the $actualDisk$. We allocate a new block of the same size from our $reserveblocks$ to replicate the data.

2.2 Implementation

To achieve replication, we made some changes in the previous section code.

1. $createDisk(id, size)$: Whenever create disk is called, we will create 2 disks with same size. One disk is used for storing data and the other disk will be used for data replication.

2. $write\_block(id, block\_num, info)$: Whenever the user writes something in a disk, same content will be written in the replica disk as well.

3. We now have an additional disk(id = -1) of size 50 blocks(10 percent of the whole address space) which is reserved for storing replicas of corrupted disk blocks.

4. Every disk will now store 2 additional attributes. One is corrupted block list. It will store the block numbers of the disk that are corrupted. The second additional attribute will store the mapping from original block number of the corrupted block to the block number of disk -1 where the corrupted data has been replicated.

5. $read\_block(id, block\_num)$: While reading, if the block number is already corrupted, we read from the second disk. If not, then we randomly check whether the block is corrupted or not. If the block is corrupted, then it is flagged as corrupted by storing it in the corrupted block list and copying its data into disk -1 to maintain 2 replicas.

6. $delete\_disk(id)$: Whenever a disk is deleted, its replica disk also gets deleted. Then the block numbers which are corrupted are translated to actual physical block numbers. These physical block numbers are then marked as corrupted so that they can never be allotted again.

7. Also, we have a design decision that if 10% of the disk gets corrupted, the disk is deemed to be unusable.

2.3 Testing

1. First we create a disk with 200 blocks and write "Atishya Jain" at block 0.

2. Then we write "Mankaran Singh" at block 99.

3. Then we write "Mayank Singh Chauhan" at block 199.

4. Then while reading all 3, we randomly got an error while reading block 99 i.e. "Mankaran Singh". Indicating block 99 is corrupt.

5. In our address space, block 99 of this disk corresponds to physical address 349. Hence, the physical address 349 is corrupted.

6. After that we deleted the disk and checked the unallotted/available physical address list.

7. We found that the physical address 349 is shown to be allotted even after we deleted the disk. This means that the physical address 349 is un-accessible and hence will never be used.

8. Also, throughout our testing, we can see that physical address 450-499 are already allotted to disk -1 which stores the data of blocks which has been corrupted.

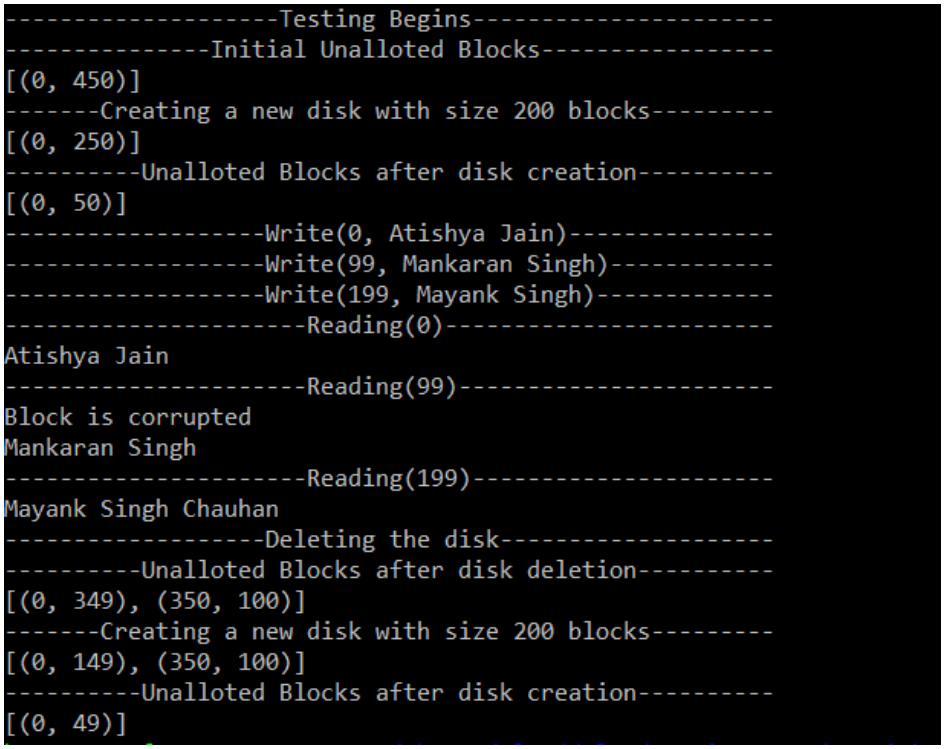

Figure 4: Replication Testing

3 SnapShotting

3.1 Design philosophy

In this section we try to implement the idea of checkpointing. We maintain an addition *log* file which is used to store all the write operations issued by the user. During *checkpoint creation*, we store a copy of this *log* file into *checkpoints* array, so that during *checkpoint restore* we can follow all the write operations mentioned in this file to get back to current state. We need not store the complete copy of the file system which would have taken a lot of memory. This idea leverages the fact that we only need to store the changes that were made in the state of the file system.

3.2 Implementation

```
create_checkpoint()
restore_checkpoint()
create_checkpoint(disk_id)
restore_checkpoint(disk_id, cpt_id)
```

1. *create_checkpoint()*: This is a member function of class Disk. It creates a copy of the current log file and stores it inside the checkpoints array. Returns the index number of the latest added checkpoint.
2. *restore_checkpoint()*: First it clears the entire content of the disk. Then it follows the write logs in checkpoint file and sets this checkpoint log file as the current disk log file.
3. *create_checkpoint(disk_id)*: Calls the create_checkpoint() function of the disk with id disk_id.
4. *restore_checkpoint(disk_id,cpt_id)*: Calls the restore_checkpoint(cpt_id) function of the disk with id disk_id.

3.3 Testing

1. First we create a disk C with 100 blocks and write 1 at block 1.
2. Then we write 2 at block 2.
3. Create checkpoint *cpt1*.
4. Write 3 at block 2.
5. Create checkpoint *cpt2*.
6. Restore state at *cpt1*, read block 2, it says 2.
7. Restore state at *cpt2*, read block 2, it says 3.
8. Write 4 at block 2.
9. Create checkpoint *cpt3*.
10. Restore state at *cpt1*, read block 2, it says 2.
11. Restore state at *cpt3*, read block 2, it says 4.

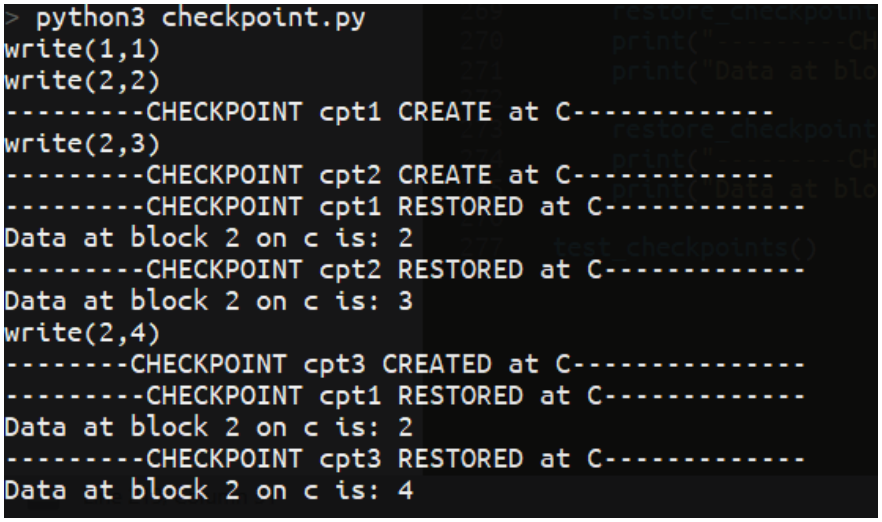

Figure 5: Checkpoint test logs