**Pocsd Final Project Report**

Group member names & UFID:

Sihao Lyu, 1888 8483

Zicheng Jiang, 8839 4919

1. ***Introduction***

In this report, we will explain the design of a RAID filesystem based on NFS and UNIX filesystem. Our system is able to run on a number of 4 to 16 servers and run in either RAID-1 or RAID-5 mode decided by the user. For a system with 4 servers, the structure will be like that in Figure 1-1.

|  |
| --- |
|  |
| ***Figure 1-1 Structure of RAID-5 and RAID-1 system*** |

Checksum is also applied to the system to deal with corruption in blocks. The structure of a block with checksum will be like that in Figure 1-2.

|  |
| --- |
|  |
| ***Figure 1-2 Structure of a Data Block with Checksum*** |

The overall structure of our filesystem is shown in Figure 1-3.

The remainder of this report is organized as follows: the design of our filesystem, the tests conducted on our filesystem and the conclusion drawn from the results of the tests.

|  |
| --- |
|  |
| ***Figure 1-3 Layering Structure of the Filesystem*** |

1. ***Design***
2. *Layers that remain unchanged*

For the overall design of the new network-based filesystem, we decide that the methods for transmitting data and command between layers to function in the same way as those in previous homework. Thus, the layers above MemoryInterface(included) can remain unchanged. But there are still some minor modifications that were made for them to fit the API of upper layers. For example, the initialization method in FileSystem now takes two parameters instead of none in order to pass the parameters “server\_id” and “raid\_mode” through the interface to Memory. We will discuss the usage of these two parameters later.

1. *Layers modified or added*
2. *Config.py*

In our design, we applied checksum of md5 algorithm to deal with corruption in blocks. A problem arises from this because the size of space used for storing data is reduced inside a block. Since md5 checksums take up 16 bytes of storage, for a 512-byte block there will be only 496 bytes left. So, we modified the config file to make it support our design. We add another parameter “WHOLE\_BLOCK\_SIZE” as the physical block size and the original “BLOCK\_SIZE” parameter now represents the data field in the block. Any changes in the config file will make some other parameters in the file to be recalculated.

1. *Disk.py and DiskLayout.py*

In order to make our design to suit the function of RAID system, the first modification we made is to the Disk itself. We only keep the read and write methods in the previous Memory.py who serves in the role of a “virtual” disk. To distinguish between them, we create a new file to play the role of the physical disk we need for the system. And the read and write methods in the new Disk.py can only write and read bit code (combination of 0 and 1) which works just like physical hard disks. And it will make implementing checksum and parity blocks that is required in RAID-5 system much easier.

1. *Client\_stub.py and server\_stub.py*

A typical RAID system is usually made up of more than one disk. So unlike the client\_stub and server\_stub we had in HW4, we need them to work under the circumstance of single client and multiple servers. The method in the stubs is also heavily modified for them to work on top of the new physical disks. The first method in client\_stub is the simple raid\_mode method who force the system to work in a chosen mode of raid disks. The second is add\_proxy which create a list to contain server information.

And besides that, there are the read and write methods whose role is to pass data between client and servers, we also add two more method in client\_stub for test. The CorruptData method is designed to corrupt a random data block on a certain server and can be used in the test for checksum. The server\_request\_count method is used to count the number of requests handled by each server and the output for that method can be used in the performance comparison test.

1. *Memory.py*

The previous Memory.py is now actually replace by Disk.py. And the current version of Memory.py in our design plays the most important role as the central controller of the filesystem. In this layer, the translation between string and ASCII code is done in order to write to or read from physical disks. We also implement a mapping rule for read and write methods to follow so that the system will be able to distinguish between parity blocks and data blocks and update parity blocks along with writes to data blocks. Checksum inside blocks is also done in this layer, we apply the MD5 algorithm which will generate a 16-byte checksum for every block.

With the method above, we are able to implement the read and write methods specially designed for RAID disks. The methods will first determine in which mode will they run. In RAID-1 mode, the write method will write a replica of the data to each of the servers. And the read will try to read the data that is required from available servers. The checksum is generated from the ASCII code of data and added to the block automatically upon a write. For each read attempt, the checksum in the fetch block is checked to examine the ingenuity of data. If the checksum does not match, client will discard the block and mark the server as a bad one then try to acquire the block from another server. Both methods are able to find broken servers and record the server’s ID so that the following commands will only be performed on good servers. And if all of the servers turned on at first is broken, they will return with an error. While in RAID-5 mode, things are much more complicated because parity blocks are added to the filesystem. In RAID-5 mode, the mapping rule mentioned above is introduced. The rule enables the system to figure out whether a block is a data block or a parity block and read and write methods will work in the way required by RAID-5 approach. Beside the checksum and server detection function in RAID-1, the write method in RAID-5 mode will update parity block in the same strip when a data block updates and the read method will only read data block if all the servers work fine. When one of the servers is down, the read method will try to recover data from the three remaining servers. The write method will try to write to the parity block in the same strip. When more than one server is down, the program will quit.

Then comes the Initialize method which is based on all the method above. The function of this method is allocating boot block, super block, bitmap blocks, inode blocks and data blocks when the user boot the filesystem. It will also fill all the blocks with empty value so that checksum and parity blocks will work as expected even though no data has been written.

In class Operations are the methods designed to support the interfaces of upper layers. These operations are modified to work much the same as those in the original Memory.py but on the physical disk needed for RAID system. The format of data in layers above MemoryInterface is the string input by the user. So, translation between string to ASCII is needed in these operations. There is also a method that is used to pass the number of requests handled by each server to upper layers.

1. *UI\_window.py*

The top most layer of our design is now the interactive window instead of FileSystem.py. The user only needs to run the UI\_window.py to boot the filesystem. An argument indicating the number of servers needs to be provided when the program starts. After the program automatically create terminals for servers, the user needs to provide the mode in which the system is supposed to run. The system will then start to initialize after given the mode.

After the initialization, the user will be able to input commands to the file system. There are 7 types of commands based on the FileSystem layer: mkdir, create, write, read, mv, rm, status. And another 3 types of commands for test: CorruptData, exit and count.

The format of the command line are as follows:

|  |  |
| --- | --- |
| Command Type | Format of Command Lines |
| make directory | $ mkdir path |
| create file | $ create path |
| move | $ mv old\_path new\_path |
| remove | $ rm path |
| read data | $ read path offset length |
| write data | $ write path data offset |
| view status | $ status |
| exit program | $ exit |
| count request | $ count |
| corrupt data block | $ corruptdata server id |

***Chart 3-1 Format for Command Lines***

The exit method can stop the program and close all server terminals. The count command can print the number of requests handled by each server.

1. ***Test and results***
2. *Test conducted*
3. *Test for RAID-1*

Firstly, we ran a test based on the test case of HW3 to examine the basic function of the system (e.g. read, write, mv, rm, create, mkdir, status). The size of data and the offset is calculated for the data to be truncated and distributed into several blocks in order to simulate the real situation.

Then the same group of tests is carried out as shut down the servers one by one to examine whether the redundancy added in RAID-1 approach increase robustness of the filesystem.

1. *Test for RAID-5*

Like the tests conducted in the RAID-1 test, we first ran a test based on the test case of HW3 to examine the basic function of the system. Still, the size of data and the offset is calculated for the data to be truncated and distributed into several blocks in order to simulate the real situation.

As for the system robustness test for RAID-5 system, we shut down just one of the servers and perform the same test again. There is also an additional test for checksum to test the function of checksum and the recover function in raid\_read method.

1. *Performance Comparison Test*

In the last test, we aim to compare the performance of multi-server system (our design) and the single server system (HW4) in terms of the number of requests handle per server. We run the same test on both systems and use the count method to get the number of requests handled by different servers.

1. *Test Results*
2. *Test for RAID-1*

In RAID-1 mode, our system passed all the test for basic functions successfully. In the robustness test, each time we shut down a server, the first command after that action will return with some error to inform the user there is something wrong with the server. And the command will still finish executing eventually after printing the errors. However, if all the servers are shut down, the program will return a fatal error and quit.

The screenshot for this test is in Appendix A.

1. *Test for RAID-5*

In RAID-5 mode, our system also passed all the test for basic functions successfully. And in the robustness test, the first command after shutting down one of the servers will return several errors. And the command as well as the following commands will still finish executing eventually after printing the errors. When more than one server is shut down, the program will return a fatal error and quit.

As for the checksum test, if there is only one corrupted block in a single strip, the read method will execute the recovery operation and return with the data requested and the number of the block that is recovered.

The screenshot for this test is in Appendix B.

1. *Performance Comparison Test*

In the performance comparison test, we separately compare the RAID-1 and RAID-5 with the system in HW4.

When we compare RAID-1 with HW4, the request counter is as follows:

|  |  |  |
| --- | --- | --- |
|  | HW4 | The 4 servers of RAID-1 |
| After Initialization | 3 | 995, 993, 993, 993 |
| After Write | 24 | 1035, 1008, 1008, 1008 |
| Request per server | 21 | 40 for server 0, 15 for others,  21 on average |
| After status | 24 | 1221, 1008, 1008, 1008 |
| After Read | 30 | 1228, 1009, 1009, 1009 |
| Request per server  (on read) | 6 | 7 for server 0, 1 for others,  2.5 on average |

***Chart 3-2 Comparison of RAID-1 and HW4***

When we compare RAID-1 with HW4, the request counter is as

follows:

|  |  |  |
| --- | --- | --- |
|  | HW4 | The 4 servers of RAID-5 |
| After Initialization | 3 | 4464, 4464, 4465, 4469 |
| After Write | 24 | 4488, 4488, 4492, 4509 |
| Request per server  (on write) | 21 | 24 for server 0 and 1, 27 for server 2, 40 for server 3, 29 on average |
| After status | 24 | 4534, 4534, 4539, 4556 |
| After Read | 30 | 4536, 4535, 4542, 4562 |
| Request per server  (on read) | 6 | 2 for server 0, 1 for server 1,  3 for server 2, 6 for server 3,  3 on average |

***Chart 3-3 Comparison of RAID-5 and HW4***

We can see from the two charts that when user issued a write, even though requests are distributed among servers, the number of requests handle by each server in RAID systems is equal to or even more than that of the system in HW4. In our assumption, this is due to the redundancy added in RAID systems which greatly increase the total number of requests issued by the client.

However, when we try to read data from the file, the average number of requests in both RAID cases is lower than that of the system in HW4. This is because the redundancy added in RAID approach will not affect read performance when all the servers are intact. The requests will be distributed evenly among the servers.

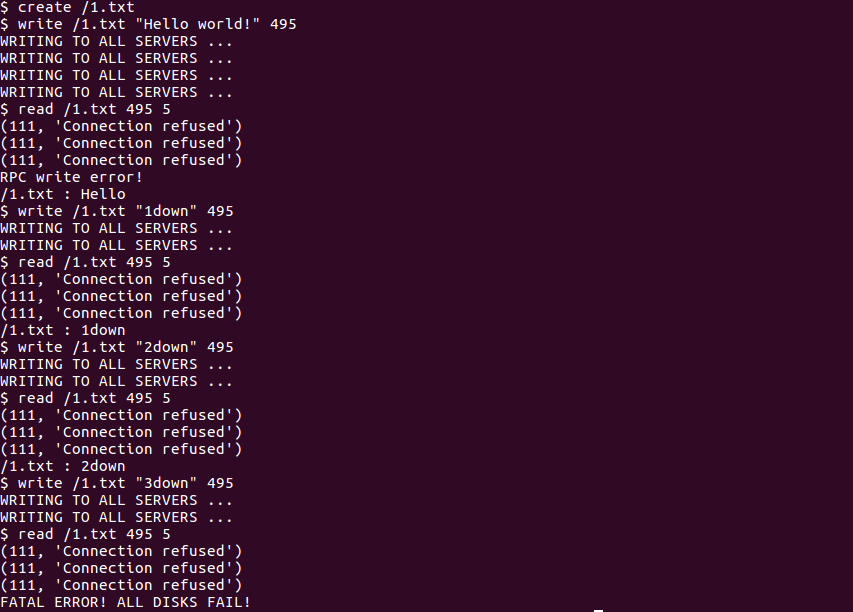
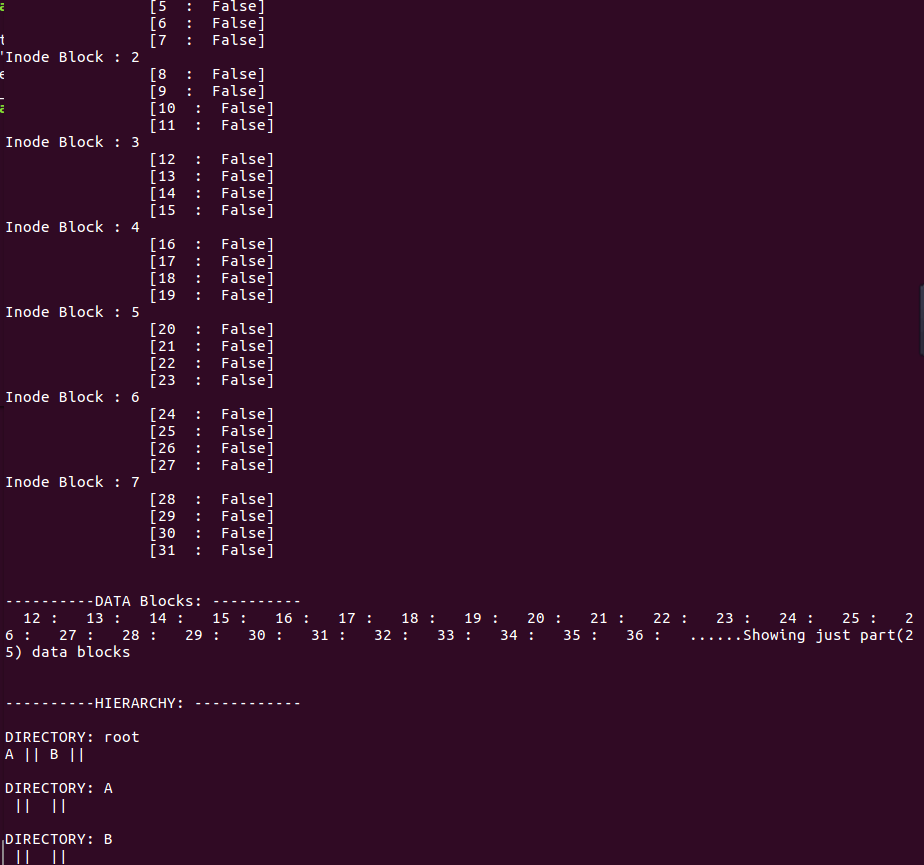
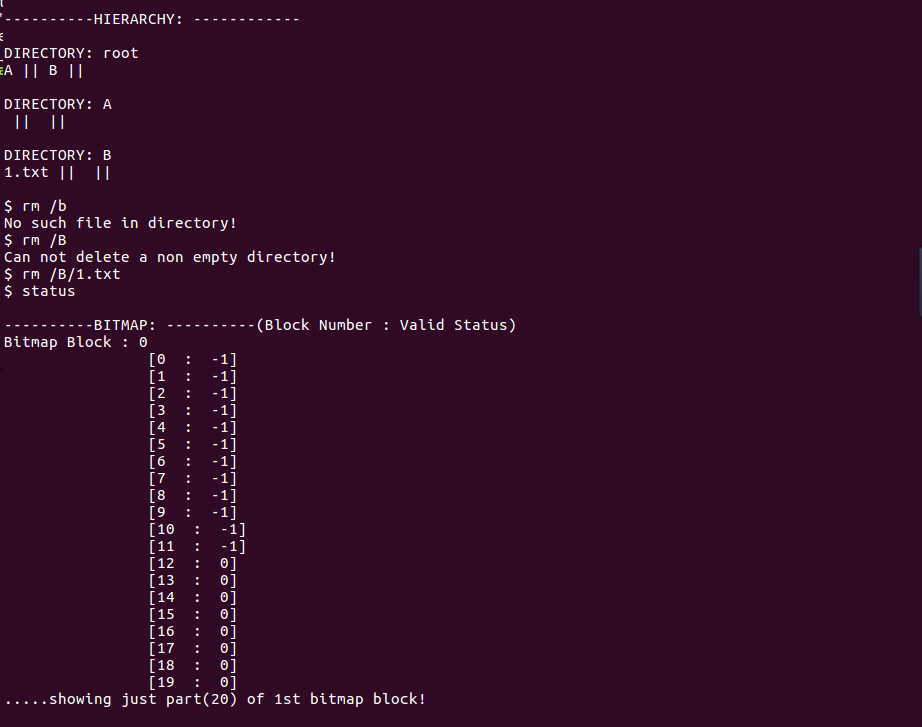
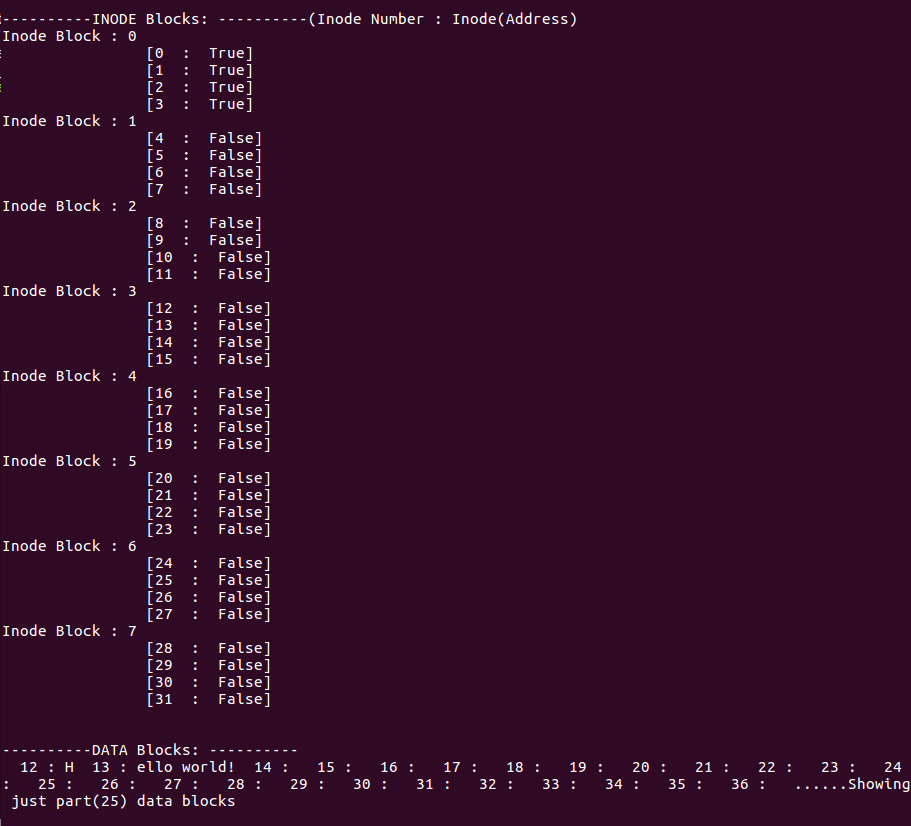
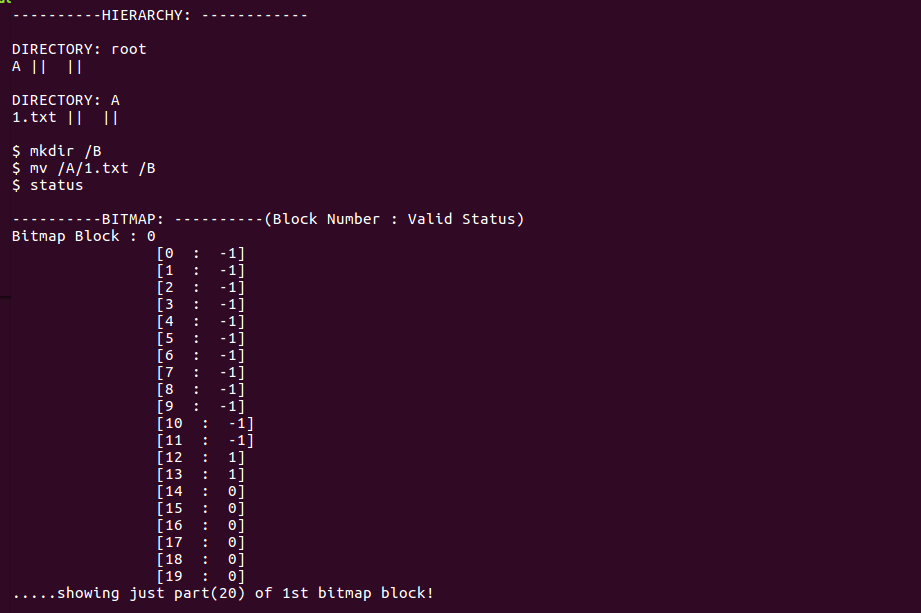
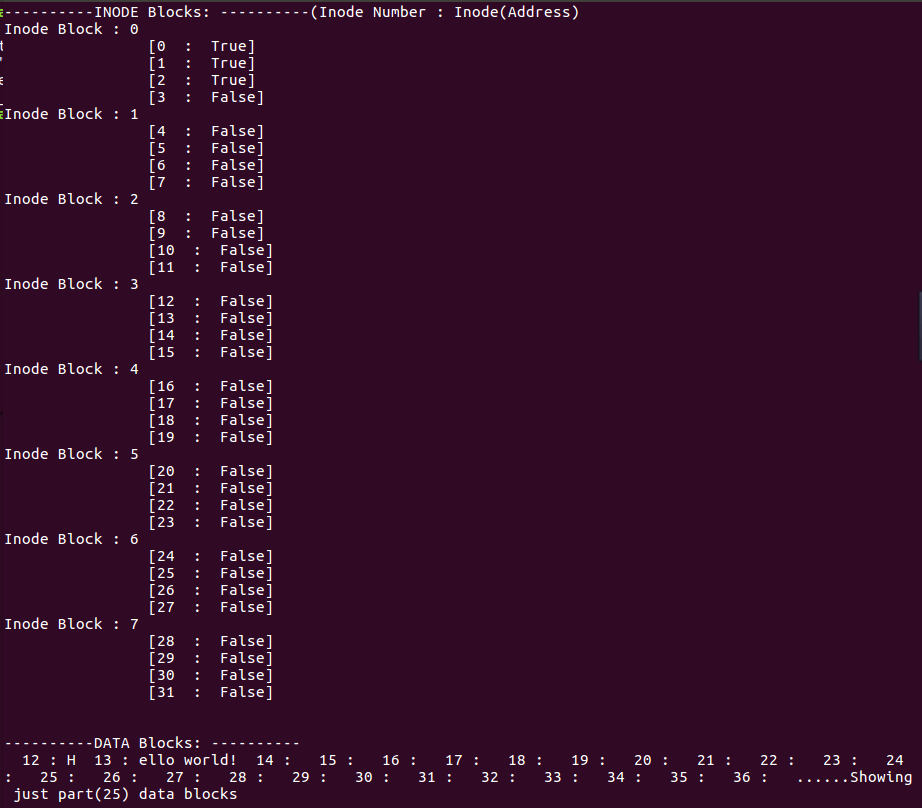
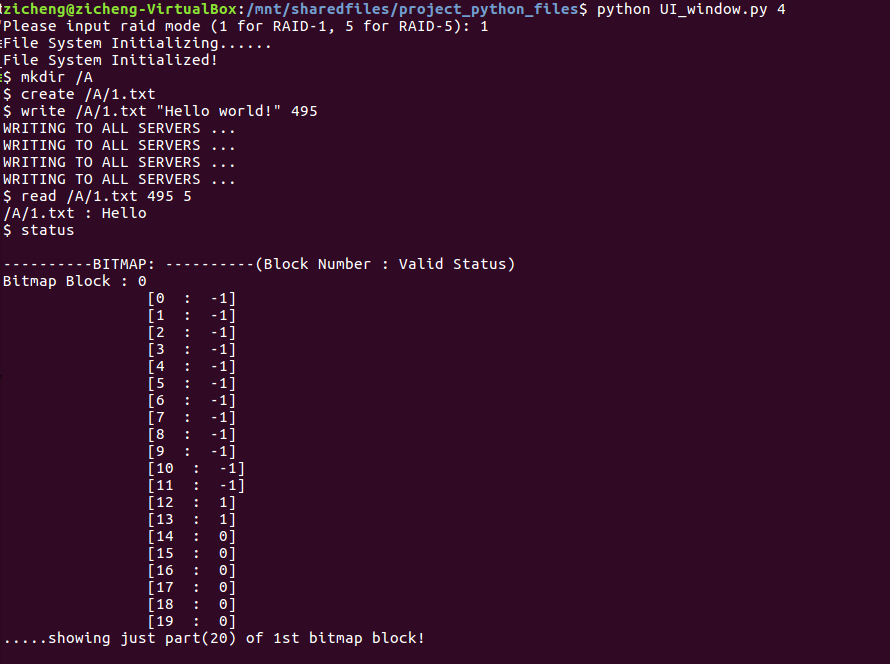
The screenshot for this test is in Appendix C.

1. ***Conclusion***

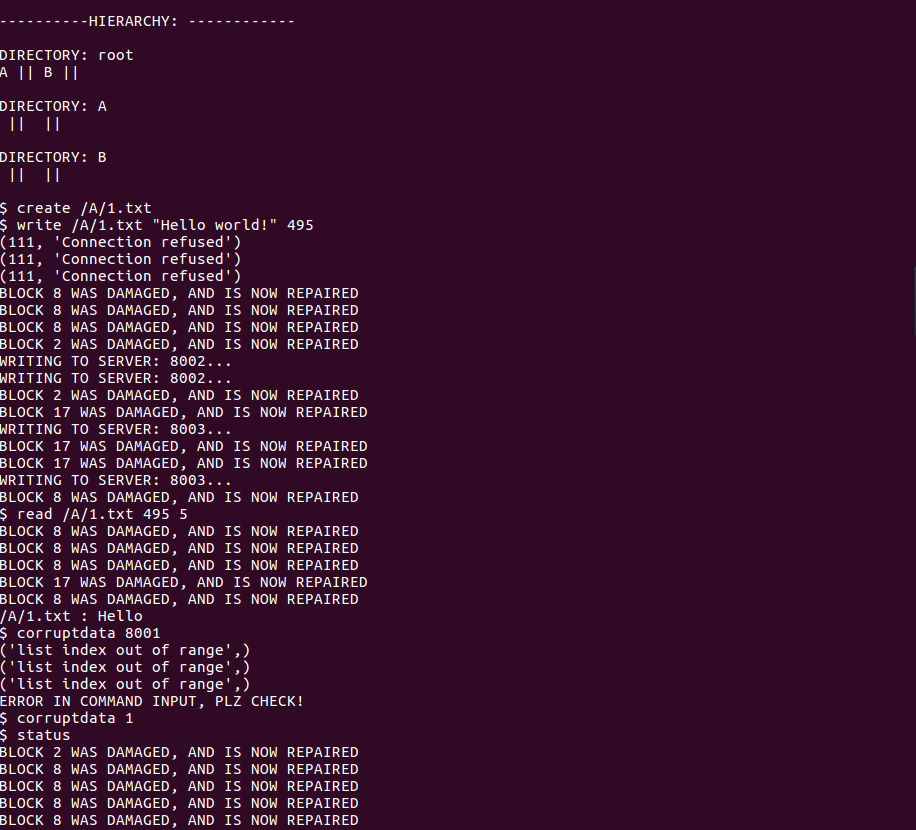
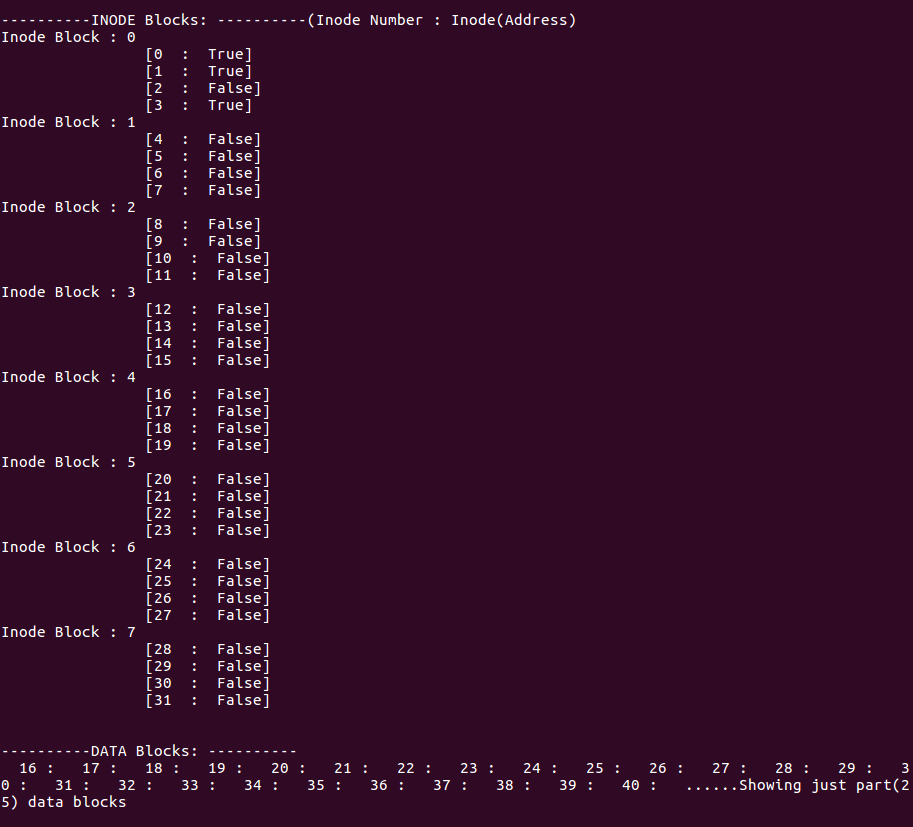
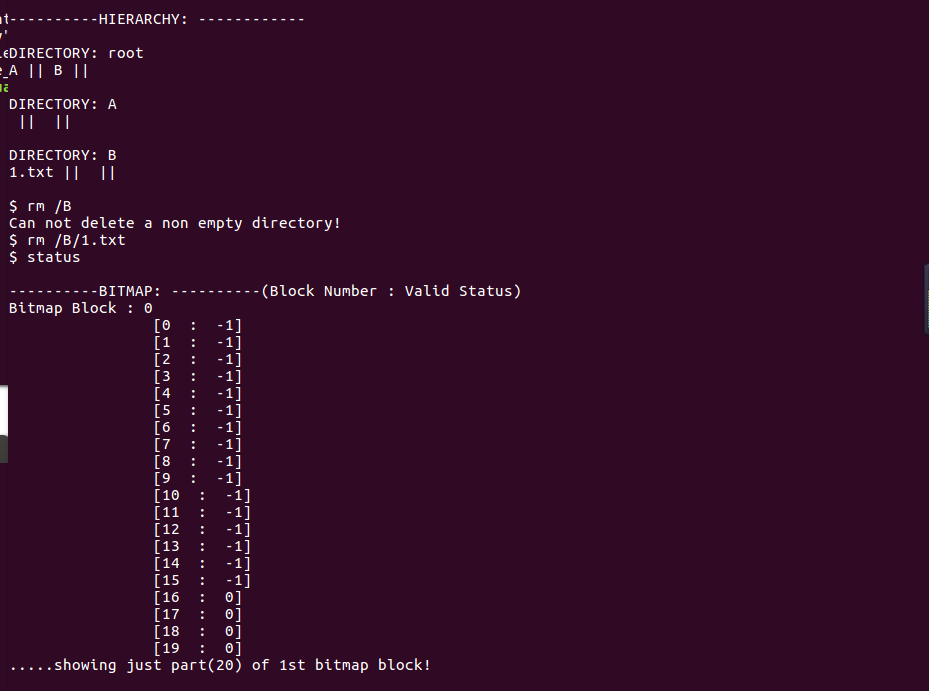
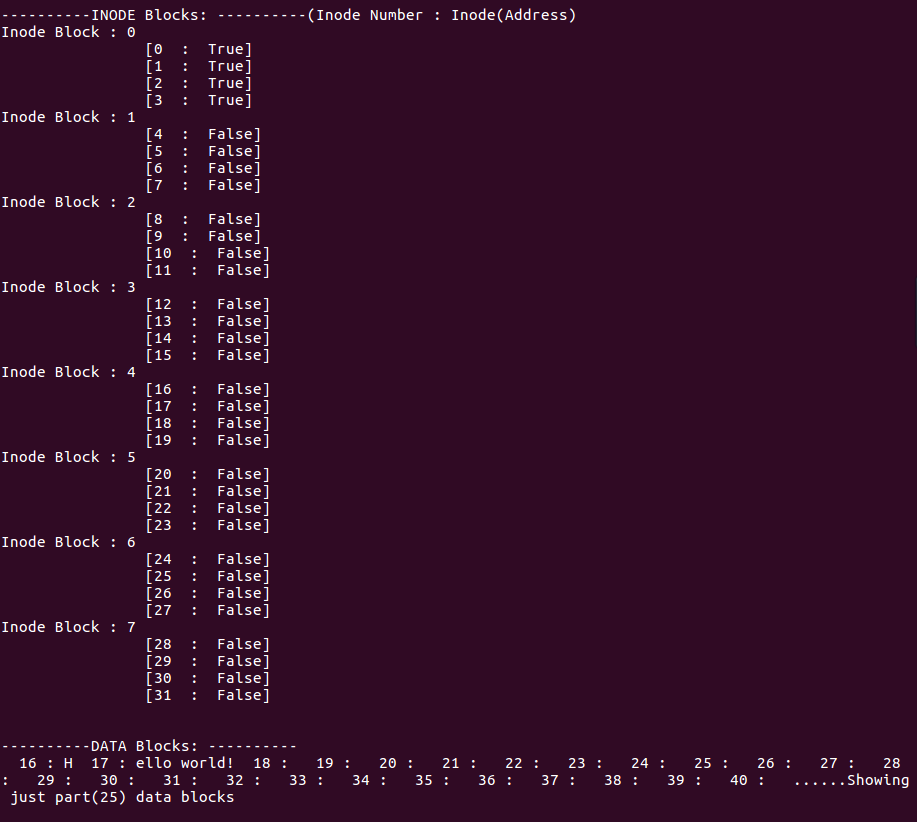
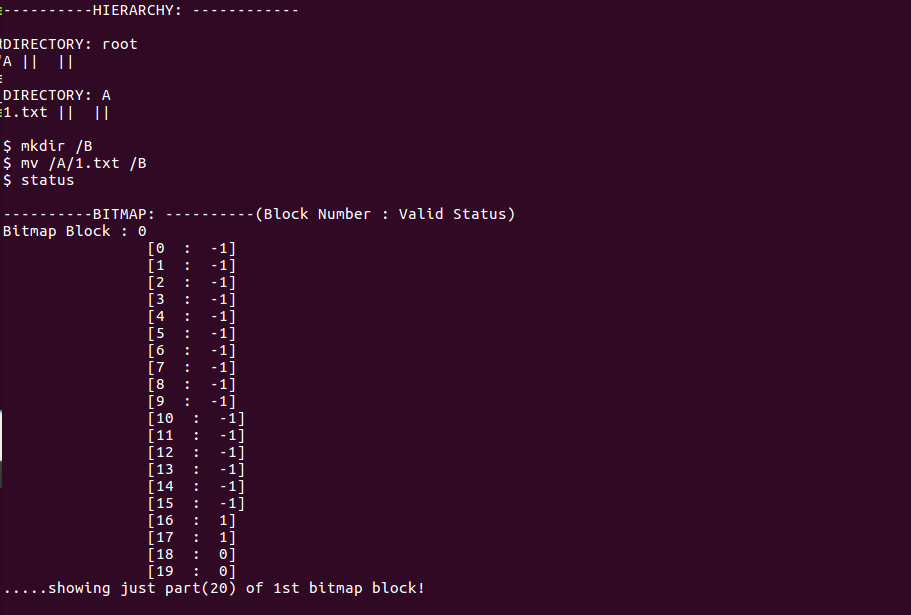
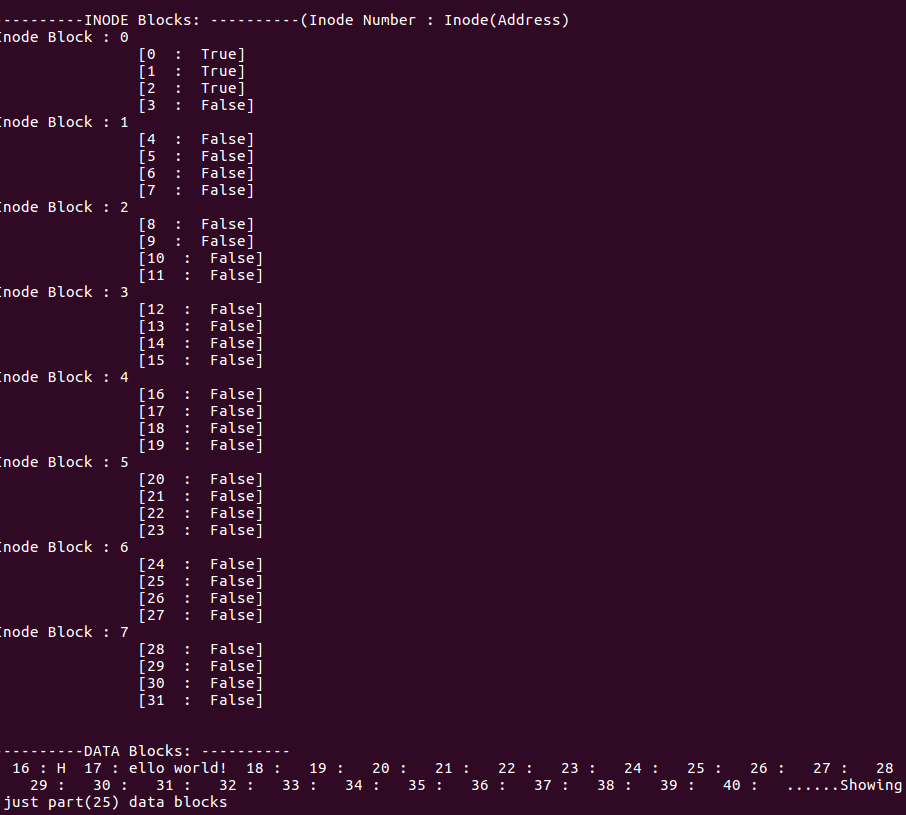
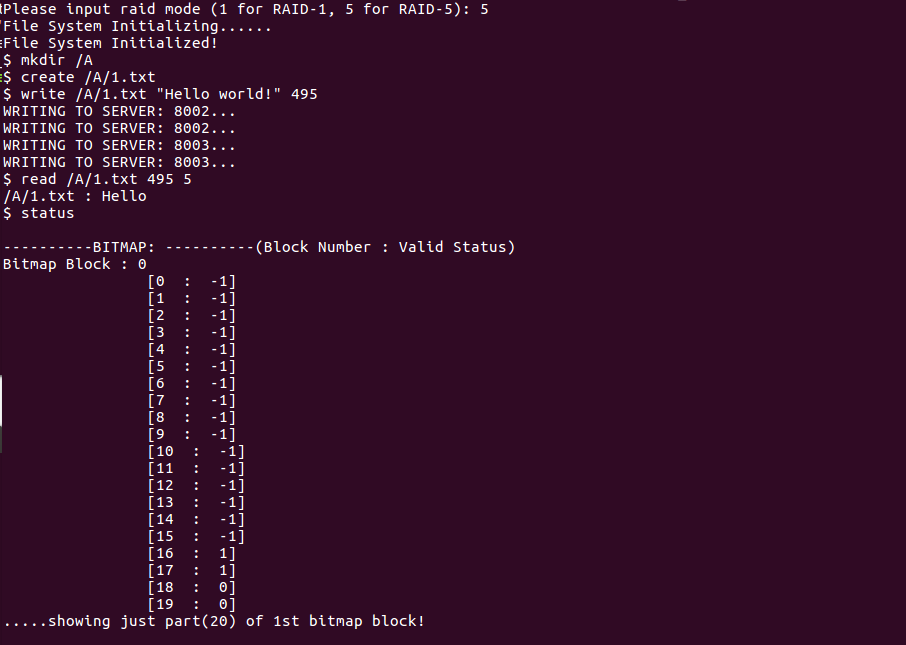
According to the result of the tests mentioned in part III, the system we designed based on the design of the system in homework can achieve the basic functions of a filesystem (e.g. create, mkdir, mv, rm, read, write). And by adding redundancy as described in RAID approach, we can deal with problems in the filesystem like bad server or corruption in data blocks.

The performance of the system in term of request handled by each server is different on read and write. The RAID system writes a little bit slower than the original single server system due to the redundancy added. However, the raid reads much faster than the original system because requests are evenly distributed among servers.

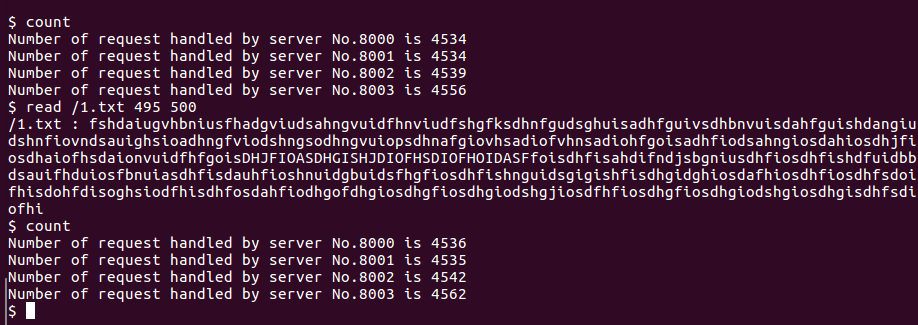
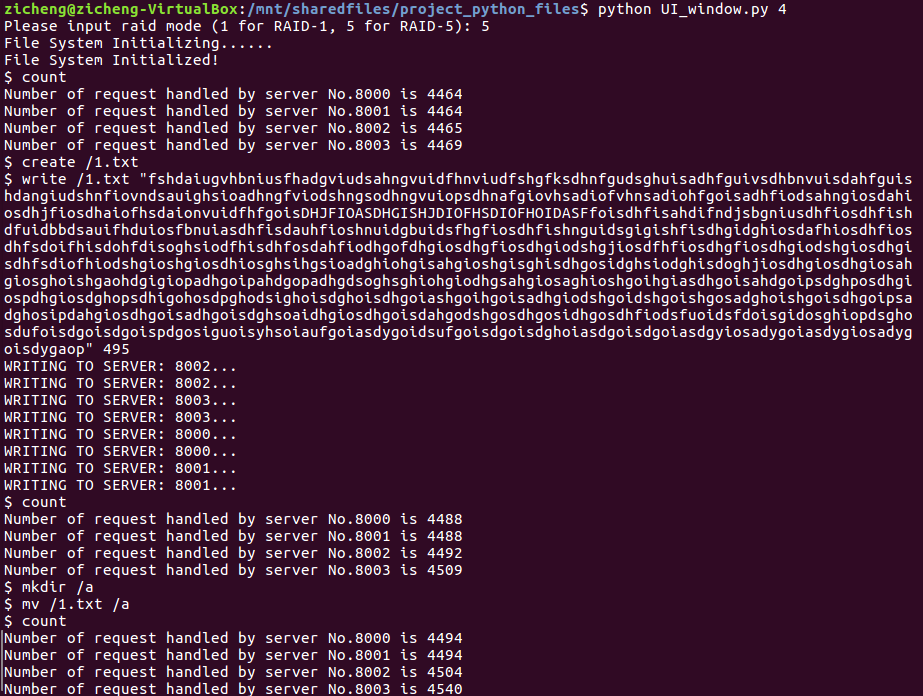
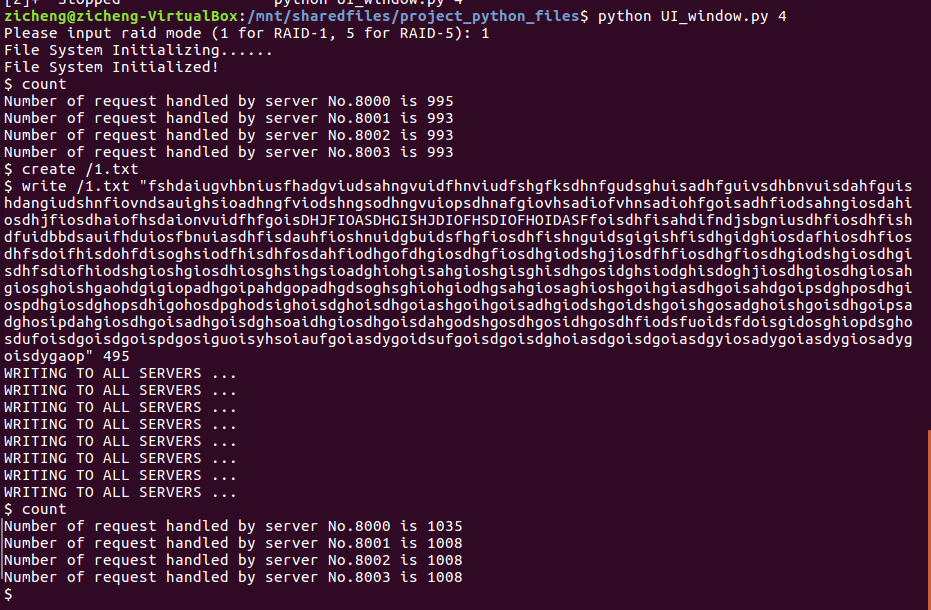
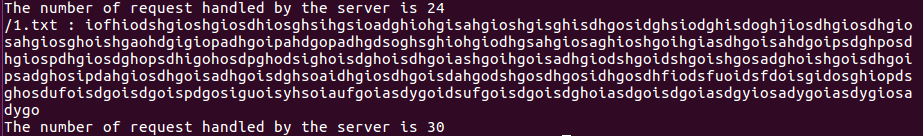
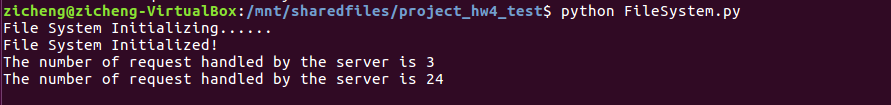
***Appendix A***

******

***Appendix B***

******

***Appendix C***

******