EEL 5737 PoCSD
UFID: 12543599
Student Name: Jingwei Ji

I. Introduction

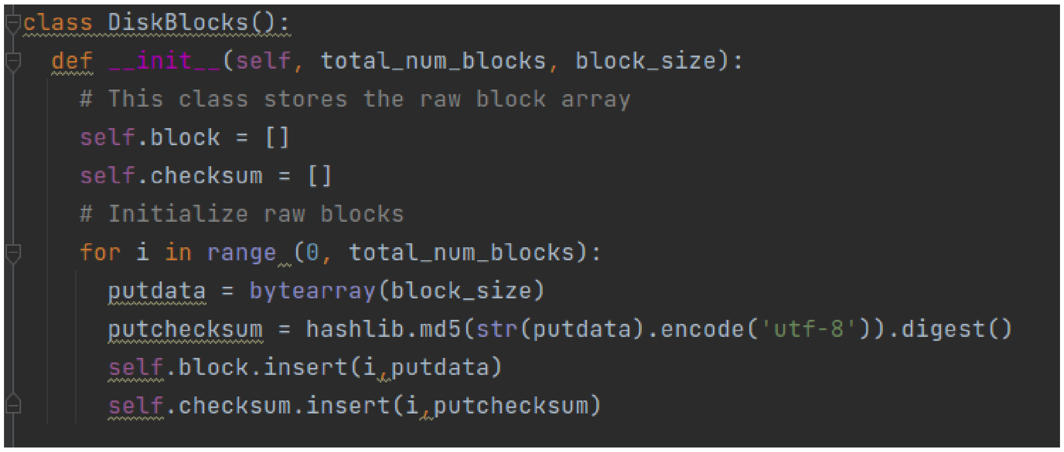
In this project, I built a distributed file system with one client and multiple servers. The project is implemented in Python and imitating the command of Unix system. The client file is built with several layers to distribute load among various servers. The function is mainly achieved in the DiskBlock() function which provide the core function Get() and Put() which communicated with the server directly.

II. Design

The memory is built in the server storage with an rpc connection package. Therefore, the client could access to the server to modify or get from one single block directly. The block of each server is distributed by applying a virtual table onto the physical block number and provided more blocks for storage. From each server's perspective, it provides the server block for storage only and is independent from other servers. From the client's perspective, it utilized method of RAID-5 to divide the parity server and data server, then utilized a virtual block number table to access to each specific server's physical block.

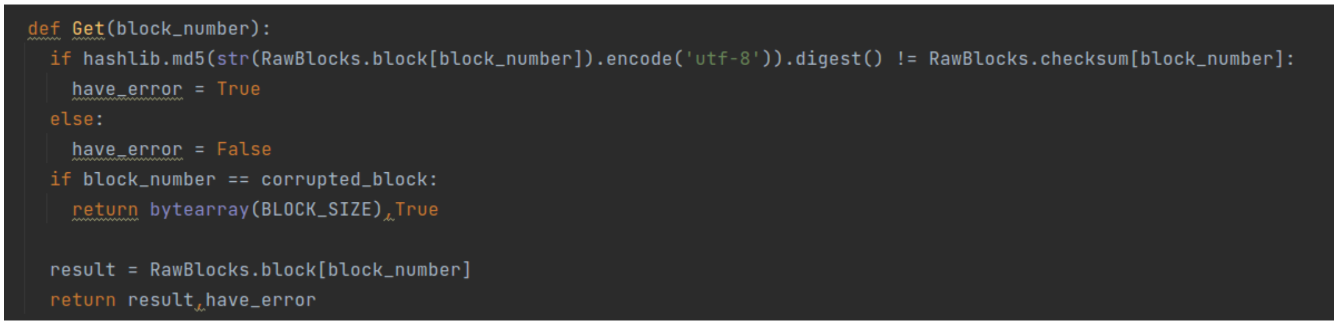
III. Design and Implementation of project milestones

1) Checksum.
First, import the md5 checksum from package hashlib. Then initialize a checksum list which consists of the same number of block numbers of each server. Whenever read or write a block from the server, the Get() function will use the initialized checksum lists to return a flag which marked whether the block content read is correct or not. The Put() function will write new content to the server block and then update the related block information in the checksum list.
Initializing the server block and checksum list



```
class DiskBlocks():
  def __init__(self, total_num_blocks, block_size):
    # This class stores the raw block array
    self.block = []
    self.checksum = []
    # Initialize raw blocks
    for i in range (0, total_num_blocks):
      putdata = bytearray(block_size)
      putchecksum = hashlib.md5(str(putdata).encode('utf-8')).digest()
      self.block.insert(i,putdata)
      self.checksum.insert(i,putchecksum)
```

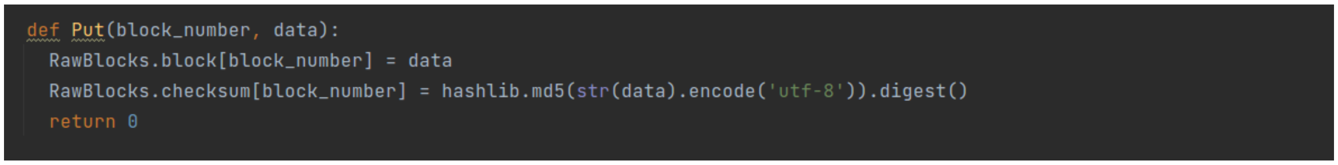
Server.Get() function and return the correct result



```
def Get(block_number):
  if hashlib.md5(str(RawBlocks.block[block_number]).encode('utf-8')).digest() != RawBlocks.checksum[block_number]:
    have_error = True
  else:
    have_error = False
  if block_number == corrupted_block:
    return bytearray(BLOCK_SIZE),True

  result = RawBlocks.block[block_number]
  return result,have_error
```

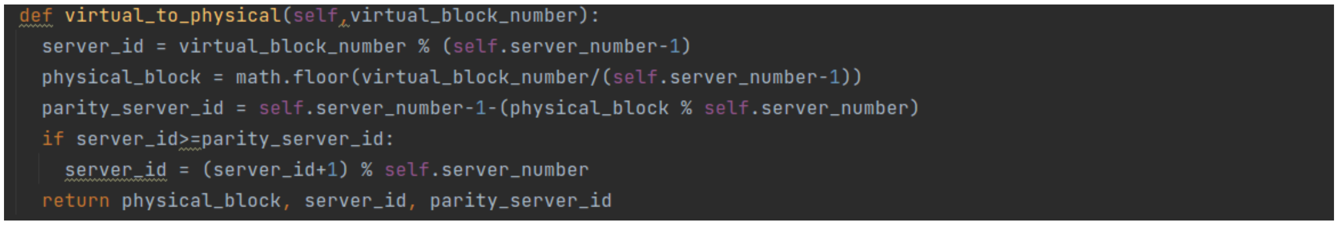
Server.Put() function to update checksum list



```python
def Put(block_number, data):
  RawBlocks.block[block_number] = data
  RawBlocks.checksum[block_number] = hashlib.md5(str(data).encode('utf-8')).digest()
  return 0
```

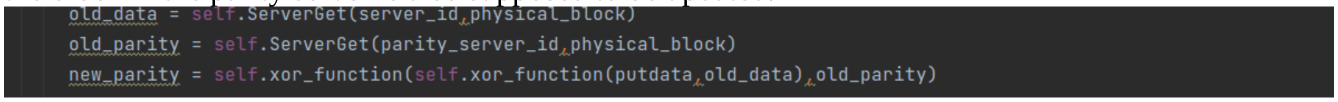
2) Building virtual table based on method RAID-5
Similar to project 3, I built a list to store multiple server block connections through xmlrpc. Therefore, the total block number the client could access is the block number that it could connected from the servers exclude the block for parity. I made a chart to indicate the distribution of various servers' blocks   from the virtual block number. Consider a one-client-four-server distributed file system.

Virtual Block Number	Server ID	Parity Server ID	Physical Block
0	0	3	0
1	1		
2	2		
3	0	2	1
4	1		
5	3		
6	0	1	2
7	2		
8	3		
9	1	0	3
10	2		
11	3		
12	0	3	4
13	1		
14	2		
15	0	2	5
16	1		
17	3		
18	0	1	6
19	2		
20	3		
21	1	0	7
22	2		
23	3		



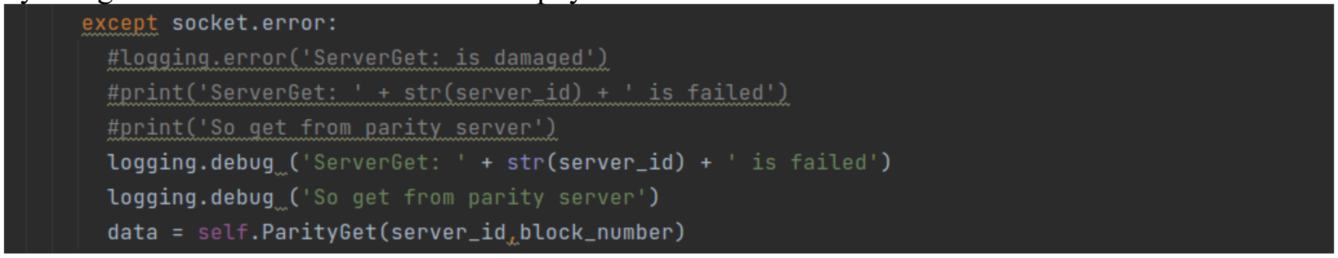
```
def virtual_to_physical(self,virtual_block_number):
  server_id = virtual_block_number % (self.server_number-1)
  physical_block = math.floor(virtual_block_number/(self.server_number-1))
  parity_server_id = self.server_number-1-(physical_block % self.server_number)
  if server_id>=parity_server_id:
    server_id = (server_id+1) % self.server_number
  return physical_block, server_id, parity_server_id
```

3) Solve error condition 1: single server fail
Use try except to check the working status of each server and use ParityGet() to get block content from failed server. The parity server is utilized to store the content of Xor calculation result of the same physical block belonging to the other servers. Whenever a Put() operation is implemented to a block, the block in the parity server is also supposed to be updated.

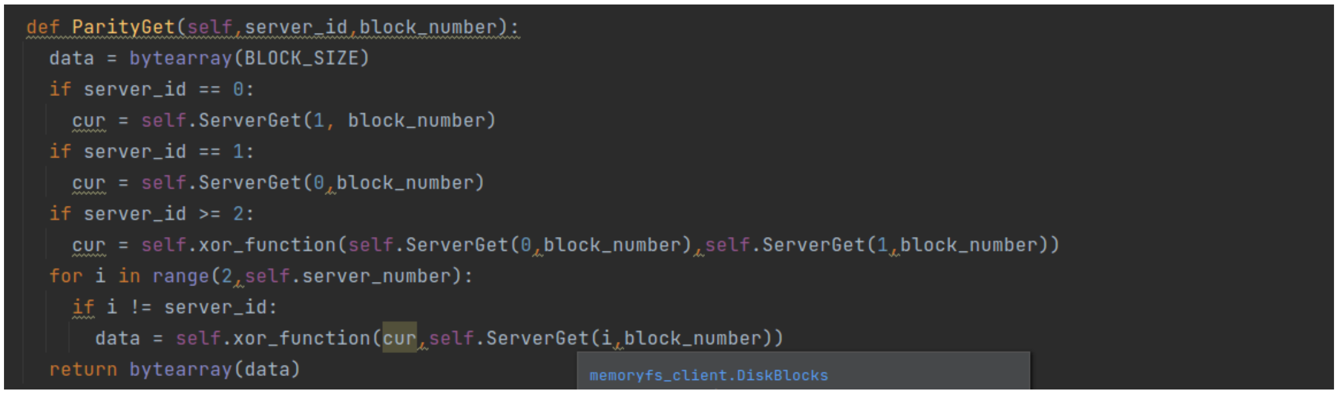


```
      old_data = self.ServerGet(server_id,physical_block)
      old_parity = self.ServerGet(parity_server_id,physical_block)
      new_parity = self.xor_function(self.xor_function(putdata,old_data),old_parity)
```

Therefore, when a single server is failed, the program goes into the condition “except” (when socket error occurred) if the block required to read or write is exactly from this failed server. Then it recovers by using the other server’s with the same physical block number to calculate a result.

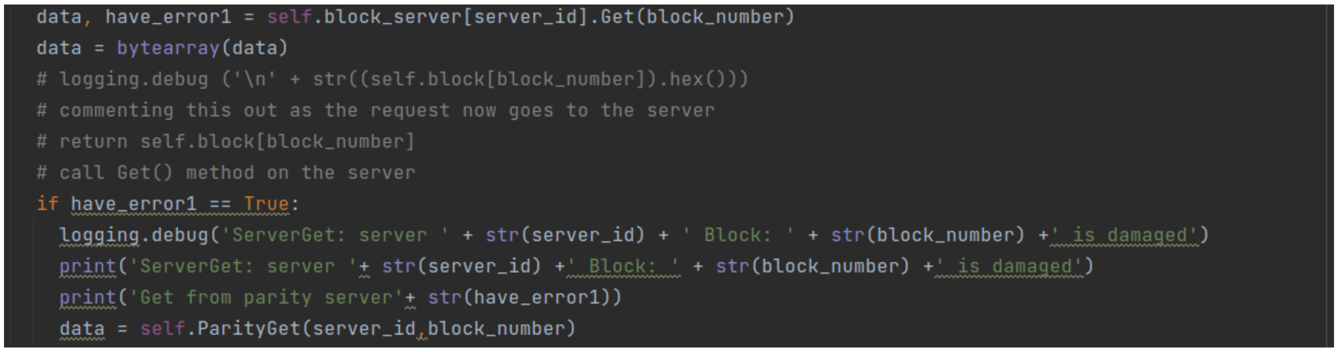


```
      except socket.error:
        #logging.error('ServerGet: is damaged')
        #print('ServerGet: ' + str(server_id) + ' is failed')
        #print('So get from parity server')
        logging.debug ('ServerGet: ' + str(server_id) + ' is failed')
        logging.debug ('So get from parity server')
        data = self.ParityGet(server_id,block_number)
```



```
def ParityGet(self,server_id,block_number):
  data = bytearray(BLOCK_SIZE)
  if server_id == 0:
    cur = self.ServerGet(1, block_number)
  if server_id == 1:
    cur = self.ServerGet(0,block_number)
  if server_id >= 2:
    cur = self.xor_function(self.ServerGet(0,block_number),self.ServerGet(1,block_number))
  for i in range(2,self.server_number):
    if i != server_id:
      data = self.xor_function(cur,self.ServerGet(i,block_number))
  return bytearray(data)
```

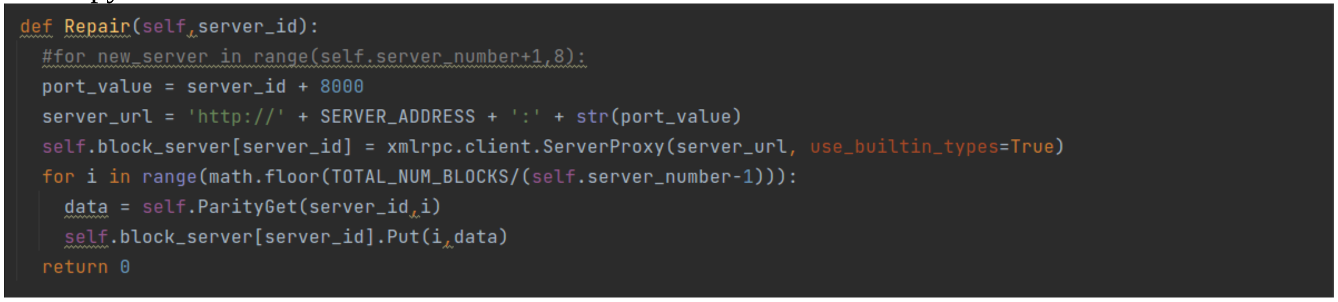
4) Solve error condition 2: Single block content damaged
Recall that the checksum is applied to every single Get() of each server and will return a bool mark as True or false. Whenever it is marked as False, the program will go to get the content from the parity server and ParityGet() function. We can also settled this flag for a specific block directly when initializing the server. No matter what the result we obtained for this block, it will always return a False to utilize the ParityGet().



```
  data, have_error1 = self.block_server[server_id].Get(block_number)
  data = bytearray(data)
  # logging.debug ('\n' + str((self.block[block_number]).hex()))
  # commenting this out as the request now goes to the server
  # return self.block[block_number]
  # call Get() method on the server
  if have_error1 == True:
    logging.debug('ServerGet: server ' + str(server_id) + ' Block: ' + str(block_number) +' is damaged')
    print('ServerGet: server '+ str(server_id) +' Block: ' + str(block_number) +' is damaged')
    print('Get from parity server'+ str(have_error1))
    data = self.ParityGet(server_id,block_number)
```

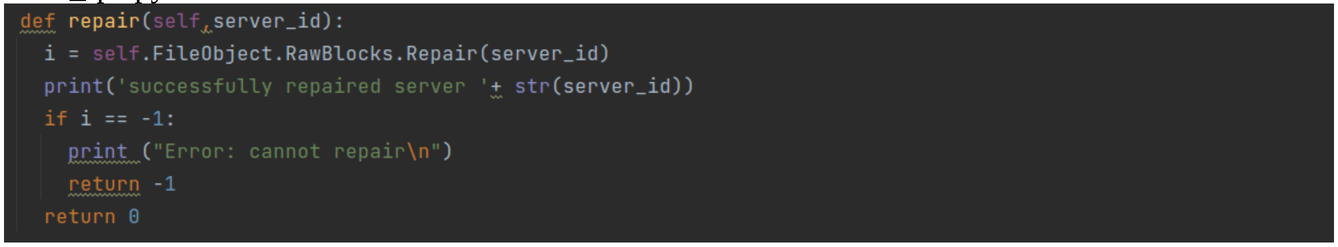
5) Repair function for error condition 1: server fail
A repair function is called to recover a crashed server. When a crashed server is reopened (notice in my program the server id must be the same as the original crashed server or it cannot be repaired), all the block content will be padded with 0. Therefore, the repair process is actually a process of calculating all the block content with ParityGet() function and then put these blocks back to the blank reopened server. The Repair() function is called directly in the memoryfs_shell_rpc.py's repair function.

Client.py



```
def Repair(self,server_id):
  #for new_server in range(self.server_number+1,8):
  port_value = server_id + 8000
  server_url = 'http://' + SERVER_ADDRESS + ':' + str(port_value)
  self.block_server[server_id] = xmlrpc.client.ServerProxy(server_url, use_builtin_types=True)
  for i in range(math.floor(TOTAL_NUM_BLOCKS/(self.server_number-1))):
    data = self.ParityGet(server_id,i)
    self.block_server[server_id].Put(i,data)
  return 0
```

Shell_rpc.py

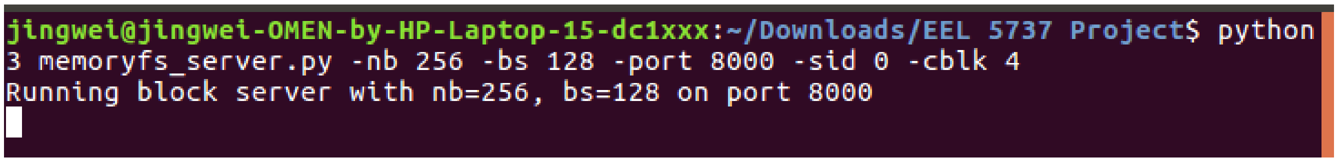


```
def repair(self,server_id):
  i = self.FileObject.RawBlocks.Repair(server_id)
  print('successfully repaired server '+ str(server_id))
  if i == -1:
    print ("Error: cannot repair\n")
    return -1
  return 0
```

6) **Load distribution check**
To check the load distribution, I utilized a list to record each server's calling times. The list is initialized at the beginning of DiskBlock() class and counted when a block is called to write data in Put() function.
When the client called load_distri in the terminal, the list will be shown to indicate the load distribution of each server. The distribution is not equally distributed because the bitmap and inode block may be called frequently than the data blocks. However, it can still distribute the load with multiple servers utilized.

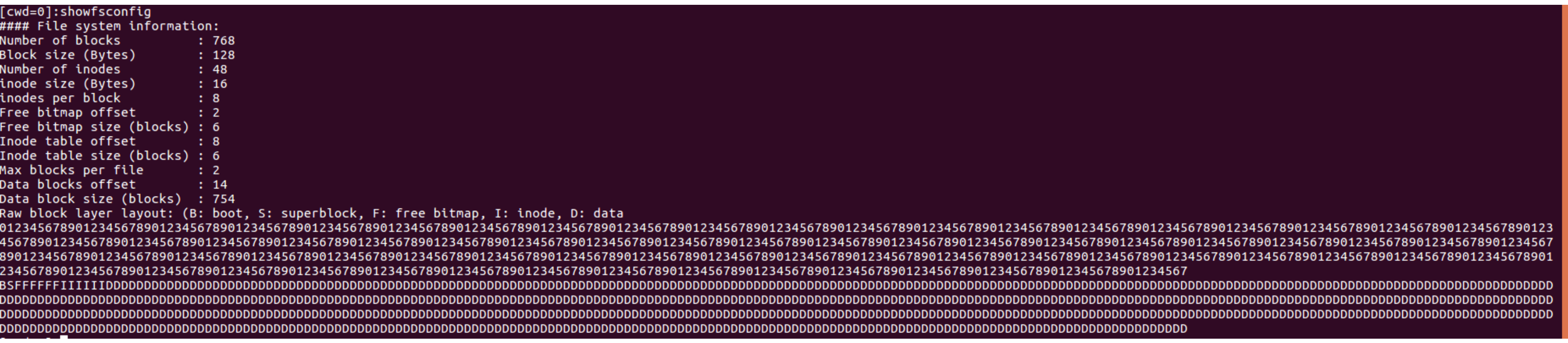
IV. Evaluation in each file
1) memoryfs_server.py
Comparing to assignment 3, a checksum list is added to check whether the block is damaged or not, we can also settle one of the block to be corrupted or not. The command for argument() is shown as below.

For example, the corrupted block is settled to 4 and the server id is 0, then the virtual block number is 12 according to the chart. When we called a function to get data from block 12 or put data to block 12, it will call the ParityGet() to get the data from the other servers.
The command in server terminal



```
jingwei@jingwei-OMEN-by-HP-Laptop-15-dc1xxx:~/Downloads/EEL 5737 Project$ python
3 memoryfs_server.py -nb 256 -bs 128 -port 8000 -sid 0 -cblk 4
Running block server with nb=256, bs=128 on port 8000
```

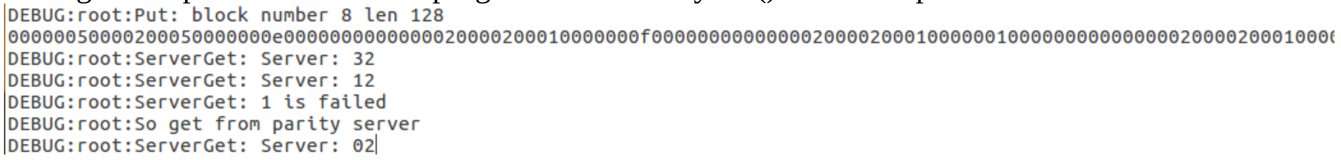
2) memoryfs_client.py
Suppose we have already started four servers. We are expected to calculate the correct total block numbers in the client terminal. The answer is 256*3 if the server block number is 256. These blocks are distributed based on RAID-5 method among these four servers.
The command will be:
python3 memoryfs_shell_rpc.py -ns 4 -port0 8000 -port1 8001 -port2 8002 -port3 8003 -nb 768 -bs 128 -is 16 -ni 48 -cid 0
The structure of the block is structured as:



```
[cwd=0]:showfsconfig
#### File system information:
Number of blocks          : 768
Block size (Bytes)        : 128
Number of inodes          : 48
inode size (Bytes)        : 16
inodes per block          : 8
Free bitmap offset        : 2
Free bitmap size (blocks) : 6
Inode table offset        : 8
Inode table size (blocks) : 6
Max blocks per file       : 2
Data blocks offset        : 14
Data block size (blocks)  : 754
Raw block layer layout: (B: boot, S: superblock, F: free bitmap, I: inode, D: data
012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123
456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567
890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901
234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567
BSFFFFFFIIIIIIDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
```

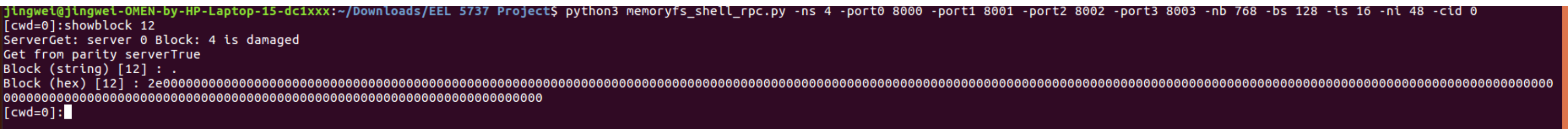
The function is mainly achieved by Put() and Get() function, and both of them are based on ServerGet() function which read and write server block directly given server id and physical block number.

1. Implementation for server fail.
The log file contains the status of failed server. I didn't print them in the terminal because I don't want the terminal to report frequently whenever a block is read from this server. When I shut down server 1, the log will report it and then the program called ParityGet() to fix this problem.



```
DEBUG:root:Put: block number 8 len 128
0000050000200050000000e00000000000000020000200010000000f00000000000000020000200010000000100000000000000002000020001000
DEBUG:root:ServerGet: Server: 32
DEBUG:root:ServerGet: Server: 12
DEBUG:root:ServerGet: 1 is failed
DEBUG:root:So get from parity server
DEBUG:root:ServerGet: Server: 02
```

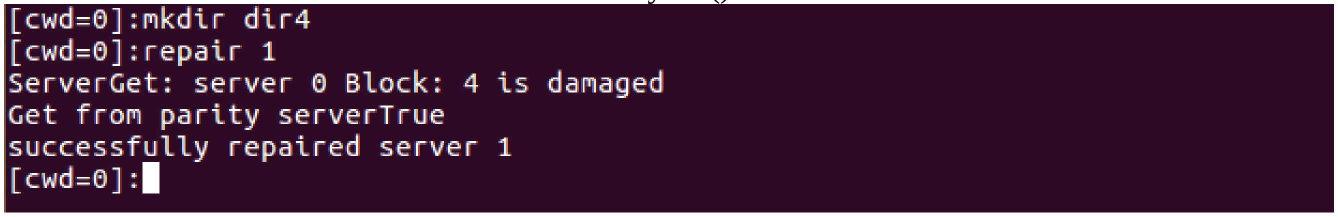
2. Implementation for corrupted block
Initializing the server 0 started with corrupted physical block 4 (virtual block is 12). The showblock 12 operation in the client terminal. It will always returned False when calling this block and then utilizing ParityGet()



```
jingwei@jingwei-OMEN-by-HP-Laptop-15-dc1xxx:~/Downloads/EEL 5737 Project$ python3 memoryfs_shell_rpc.py -ns 4 -port0 8000 -port1 8001 -port2 8002 -port3 8003 -nb 768 -bs 128 -is 16 -ni 48 -cid 0
[cwd=0]:showblock 12
ServerGet: server 0 Block: 4 is damaged
Get from parity serverTrue
Block (string) [12] : .
Block (hex) [12] : 2e000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000
[cwd=0]:
```

3) memoryfs_shell_rpc.py

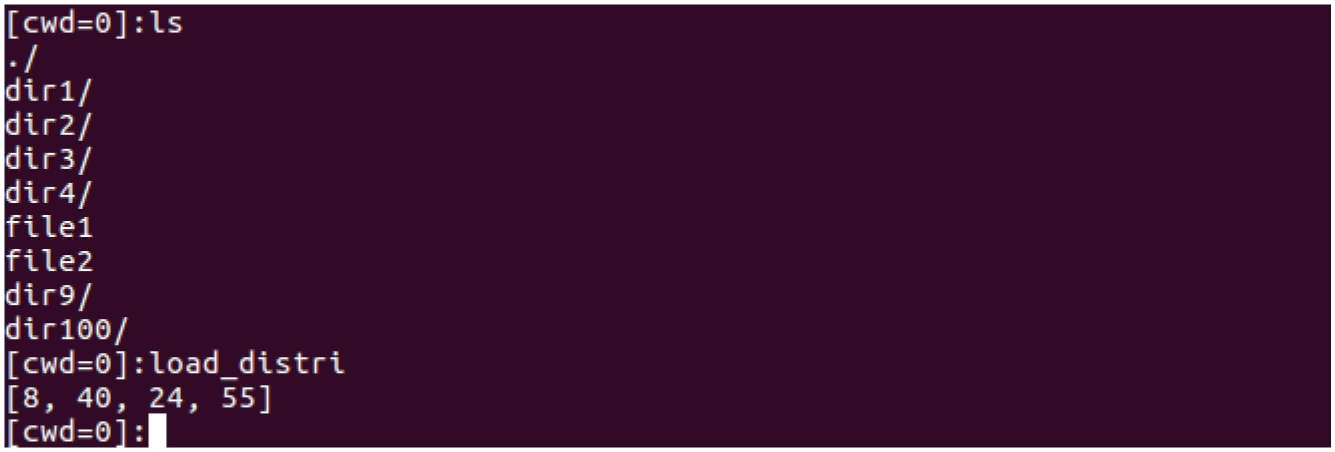
1. repair function of crashed server
I shut down server 1 and reopened the same server with the same port(Notice that my program cannot repair the server if the reopened server is not the original port), then utilized repair function to recover the block content from the other servers with ParityGet().



```
[cwd=0]:mkdir dir4
[cwd=0]:repair 1
ServerGet: server 0 Block: 4 is damaged
Get from parity serverTrue
successfully repaired server 1
[cwd=0]:
```

Since the block 4 of server 0 is called to recover the server 1's block 4 once, so the sentence is printed again in the client's terminal.

2. print load distribution of variable servers
To distinguish the difference between load file function and load distribution, I called this function as load_distri(). In this distributed file system, it has four servers with the load printed in list of the terminal.



```
[cwd=0]:ls
./
dir1/
dir2/
dir3/
dir4/
file1
file2
dir9/
dir100/
[cwd=0]:load_distri
[8, 40, 24, 55]
[cwd=0]:
```

4) Problem occurred when:
When the same physical block of two servers is damaged, or no less than two servers are failed together. The program will go into an infinite loop between ServerGet() and ParityGet(), and finally quit:
RecursionError: maximum recursion depth exceeded while calling a Python object

V. Conclusion
This file system can only solve problems like a single block damage and a single server failed. The method to quit the program when above problems occurred is violently cause a maximum recursion depth exceeded which is supposed to be optimized. Among these four assignments, I thought the most difficult assignment is the first one which first introduced each layer and the practical concept of

modularity in a file system which required me to go through the whole inode layer, and understand how the block is allocated in a simple file system.