CS 519 – Operating Systems Theory

Fall 2015 – Homework 2

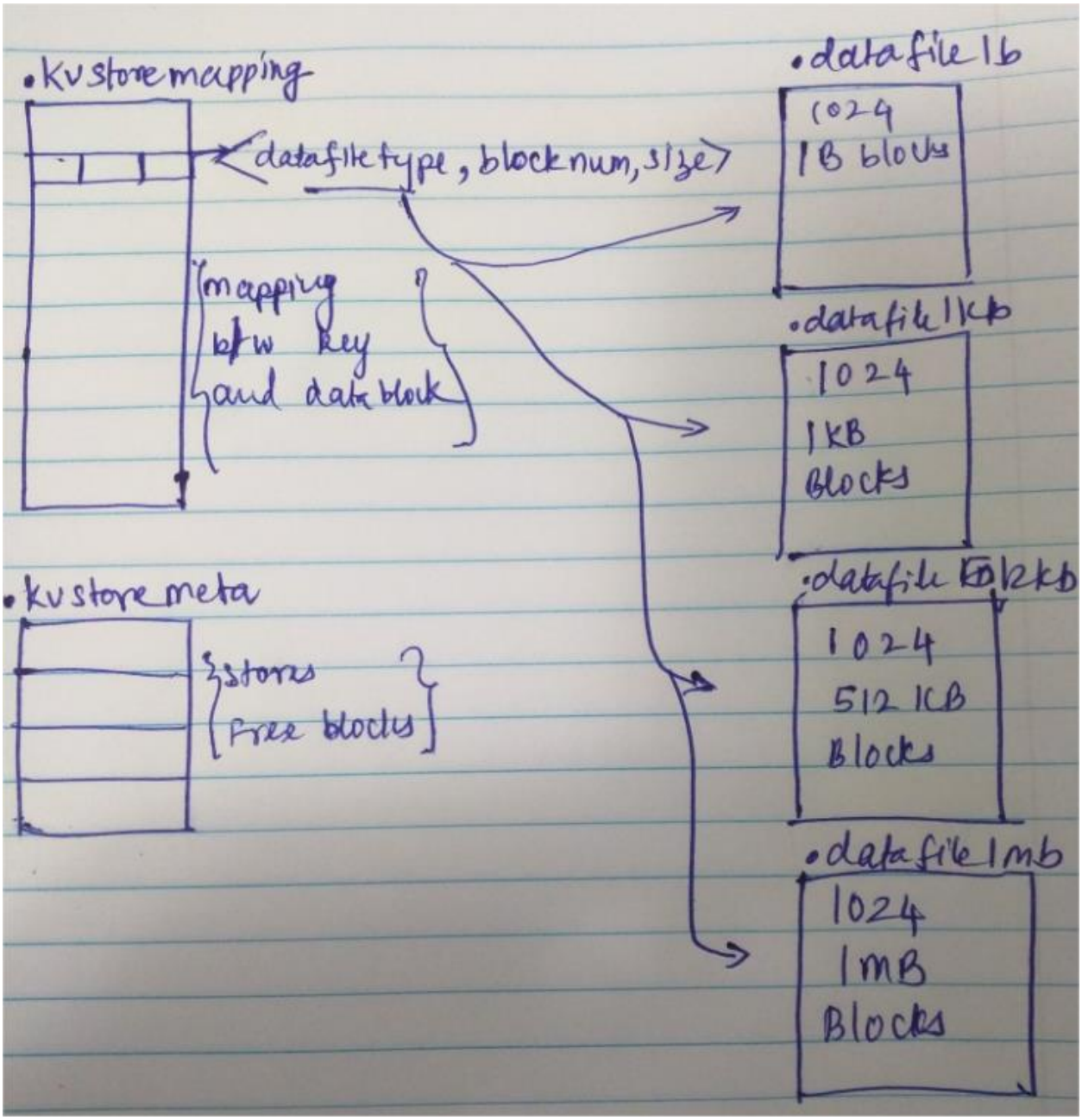
Abhijit Shanbhag (as2249)
Priyanka Dhingra (pd374)

I. Key value store

A. Assumptions:

1. The key value store that we implemented needs a persistent hash table for optimized key insert, access and delete operations. For the version that we are submitting in checkpoint 1, we have implemented using a naïve hash table (i.e. an array of size 2^{32} elements). We have couple of ideas on how to implement this better, however we will implement this before our final implementation deadline after approval from Bill.

B. High level Design:



1. We implemented key value store library in checkpoint 1 that we intend to use to build the file system later for the final submission.
2. In this key value store, we support arbitrary sized blocks of size: 1B, 1KB, 512KB and 1MB as required in the assignment.
3. As we need to store the mapping of a key to a variable size block size, we do not store it directly, but using a mapping of the key to data blocks.
4. We store the key value store as 3 type of files as below:
    i. **Key value Store Mapping file: (.kvstoremapping)**
        - This file stores the mapping between the key and the data block.
        - The mapping is stored using a struct valuetuple_t data type which stores the data block type (i.e. 1B, 1KB, 512KB or 1MB), the block number and the actual size of data stored in the block.

```
typedef struct valuetuple {
    int data_fd;
    int n_block;
    int size;
} valuetuple_t;
```

        - Hence, each key takes up 12B of storage.
        - As we have implemented a naïve hash table, the max file of this mapping would be 2^{32}*12B.

    ii. **Key value Store Meta file: (.kvstoremeta)**
        - This file stores information about free blocks in 4 char arrays for each data block.
        - Size = 1024 * 4 B
        - Currently in version 1 we are using a char array for storing the storing the status, we would be using a bitvector in the final version.
    iii. **Key Value Store Data file: (.datafile1b .datafile1kb .datafile1mb .datafile512kb)**
        - Corresponding data file for each type of block.
        - Currently supports 1024 blocks of each type and is configurable.
5. Implemented 4 APIs for key value store as below:
    i. **int size(void* keyPtr) :**
    ii. **int put(void* keyPtr, void* valPtr, int size) :**
    iii. **int get(void* keyPtr, void* valPtr, int size) :**
    iv. **int delete(void* keyPtr) :**
6. We gave special importance on how the key value store is implemented with regards to transactions. For example in the PUT operation the order of updates were as below:
    i. **Update datablock.** If something fails after this, this block is overwritten and we are ok with that.
    ii. **Update the kvstoremeta file.** If something fails after this, in the worst case scenario we have a block which could be never used.

iii. **Update the kvstoremapping file.** This is the final step and the mapping is persisted after this step.

7. Similar care was given to the delete operation.
   i. **Delete the mapping in kvstoremapping file**
   ii. **Free block in kvstoremeta file.** Actual data block was not deleted as the data could never be found using the key value store and will be overwritten next time.

C. Compiling and Testing:

I. Compiling:

```
-bash-4.1$ make
make all - to build library and create all object files
make clean - to clean everything
make kvstore_test - to build the test case
make checkpoint1_test - to build the test case
make checkpoint_userinput1_test - to build the test case
make library - to create library libkvstorelib.a
-bash-4.1$
```

II. Checkpoint1_test:

We wanted to write the test case as Bill suggested, using dev/random system call. However since this is a sudo call, we were not able to work on development ILAB machine. We however used the same essence and build a random generated key and value program to test the functionality of our key value store.

Our test program does the following things **1000 times**:

- Randomly generates a key and val integers up to 1024*1024.
- Create a input string comprised of repeated ‘x’s val number of times. Eg. If val was 10, the input string would be ‘xxxxxxxxx’ of size 10B.
- Perform SIZE operation on this key. They GET should return 0.
- Perform PUT operation on the key and input string. The PUT operation should be successful.
- Perform SIZE operation and the size of the stored block should match the size of the input block.
- Perform GET operation and store in result string. The input and result string should match.
- Perform DELETE operation and status should be successful.

```
**********Performing experiment 994**********
Key generated: 193536 with value- A string with 179200 x's of size 179200 bytes
SIZE operation before inserting key: 193536 return size: 0
PUT operation on key: 193536 returns status: 1
SIZE operation on key: 193536 return size: 179200
GET operation on key: 193536 returns status: 1
DELETE operation on key: 193536 returns status: 1
Experiment: 994 SUCCESS as all operations succeeded
**********Performing experiment 995**********
Key generated: 444416 with value- A string with 215040 x's of size 215040 bytes
SIZE operation before inserting key: 444416 return size: 0
PUT operation on key: 444416 returns status: 1
SIZE operation on key: 444416 return size: 215040
GET operation on key: 444416 returns status: 1
DELETE operation on key: 444416 returns status: 1
Experiment: 995 SUCCESS as all operations succeeded
**********Performing experiment 996**********
Key generated: 132096 with value- A string with 33792 x's of size 33792 bytes
SIZE operation before inserting key: 132096 return size: 0
PUT operation on key: 132096 returns status: 1
SIZE operation on key: 132096 return size: 33792
GET operation on key: 132096 returns status: 1
DELETE operation on key: 132096 returns status: 1
Experiment: 996 SUCCESS as all operations succeeded
**********Performing experiment 997**********
Key generated: 169984 with value- A string with 409600 x's of size 409600 bytes
SIZE operation before inserting key: 169984 return size: 0
PUT operation on key: 169984 returns status: 1
SIZE operation on key: 169984 return size: 409600
GET operation on key: 169984 returns status: 1
DELETE operation on key: 169984 returns status: 1
Experiment: 997 SUCCESS as all operations succeeded
**********Performing experiment 998**********
Key generated: 365568 with value- A string with 590848 x's of size 590848 bytes
SIZE operation before inserting key: 365568 return size: 0
PUT operation on key: 365568 returns status: 1
SIZE operation on key: 365568 return size: 590848
GET operation on key: 365568 returns status: 1
DELETE operation on key: 365568 returns status: 1
Experiment: 998 SUCCESS as all operations succeeded
**********Performing experiment 999**********
Key generated: 784384 with value- A string with 361472 x's of size 361472 bytes
SIZE operation before inserting key: 784384 return size: 0
PUT operation on key: 784384 returns status: 1
SIZE operation on key: 784384 return size: 361472
GET operation on key: 784384 returns status: 1
DELETE operation on key: 784384 returns status: 1
Experiment: 999 SUCCESS as all operations succeeded
**********Performing experiment 1000**********
Key generated: 434176 with value- A string with 867328 x's of size 867328 bytes
SIZE operation before inserting key: 434176 return size: 0
PUT operation on key: 434176 returns status: 1
SIZE operation on key: 434176 return size: 867328
GET operation on key: 434176 returns status: 1
DELETE operation on key: 434176 returns status: 1
Experiment: 1000 SUCCESS as all operations succeeded
-bash-4.1$
```

III. **Checkpoint_userinput1_test:**

Same test as above but with user input key and value.

IV. **Kvstore_test:**

This test case is to check for basic functionality and to check with the edge cases for the key value store.

D. Difficulties and Challenges:

1. We had difficulties in implementing persistent hash table. We could not find a ready-made library to implement this.
2. Also we faced difficulties in writing the gnome hash table to file as we couldn't find reliable reference to do it.
3. We have an idea where we use gnome hash table in memory and also update the files accordingly, but will implement this after discussing with Bill for the final submission.