CS-342 Project 4 Report

Cankat Anday Kadim – Rüzgar Ayan
21802988 - 21801984

File System Details

Superblock

In the superblock, the size of the file system and a list of the open files are stored as follows:

```
struct openFileInfo {
    int used; //1 if used, 0 if not used
    int directoryEntryIndex;
    int lastPositionByte;
    int mode;
    pid_t processId;
};

struct superblockData {
    int size;
    struct openFileInfo openFiles[MAX_OPEN_FILES];
};
```

In the entries of the openFiles array, we store:

- Whether or not that entry is used
- Which directory entry that open file corresponds to
- If opened with MODE_READ, a pointer to the last position of reading
- The mode that the files is opened with, MODE_READ or MODE_APPEND
- Pid of the process that opened the file

When a file is opened, an index of the openFiles array is returned as the file descriptor value.

Bitmap

Each bitmap block just stores 1024 int values:

```
struct bitmapBlock {
    int bitmap[BITMAP_ROWS_PER_BLOCK];
};
```

To access and modify the actual bits in these ints, bitwise operations AND and OR are used.

Root Directory

Each root directory block consists of an array of 32 directory entries.

```
struct directoryEntry {
    char fileName[FILENAME_LENGTH];
    int index; //Will be -1 if the directoryEntry is not used
    char filler[10];
};
```

```
struct rootDirectoryBlock {
    struct directoryEntry directoryEntries[DIRECTORY_ENTRIES_PER_BLOCK];
};
```

Each directory entry holds the name of the file, the index of the corresponding FCB entry and a filler of 10 bytes to complete the struct into 128 bytes. Index value -1 is used to show that the directory entry is empty.

FCB Table

Each FCB table block consists of an array of 32 FCBs.

```
struct FCB {
    int used;
    int size;
    int indexBlock;
    char filler[116];
};

struct FCBTableBlock {
    struct FCB FCBList[FCB_PER_BLOCK];
};
```

Each FCB table entry holds whether or not the entry is used, the size of the corresponding file in bytes, the block that stores the file's index table and a filler to complete the struct into 128 bytes.

Index Table

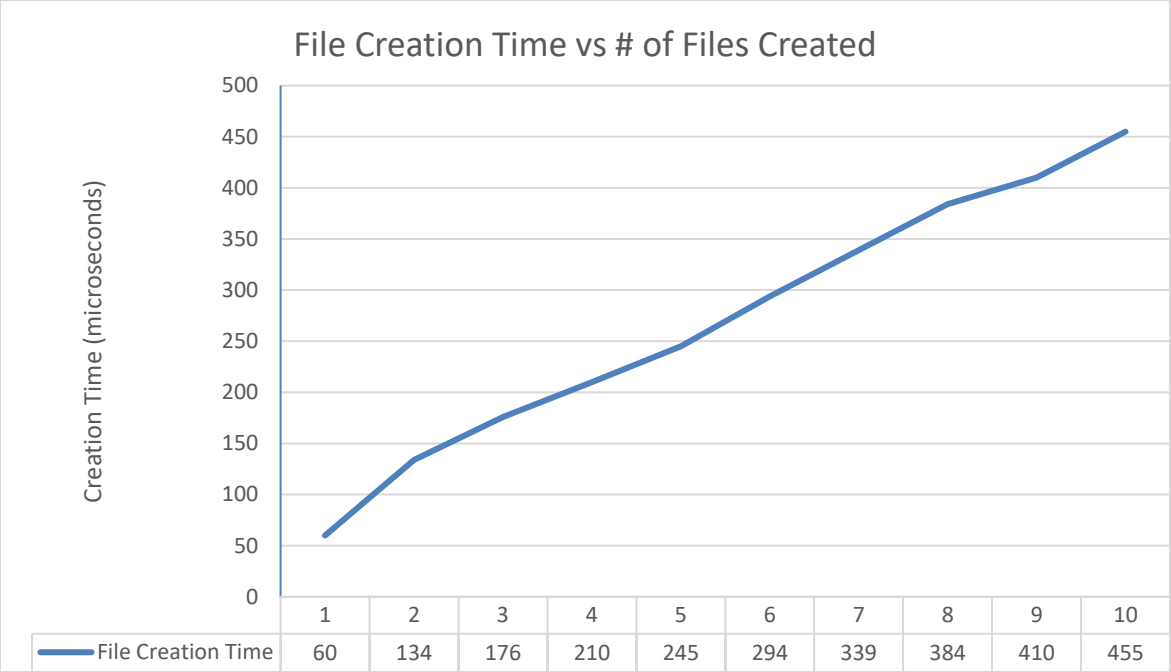
Index tables just hold an array of 1024 int values, the entries of the array that are not -1 show the blocks that the corresponding file is stored at.

```
struct indexTable {
    int indexTable[ROWS_PER_INDEX_TABLE];
};
```

Experiments

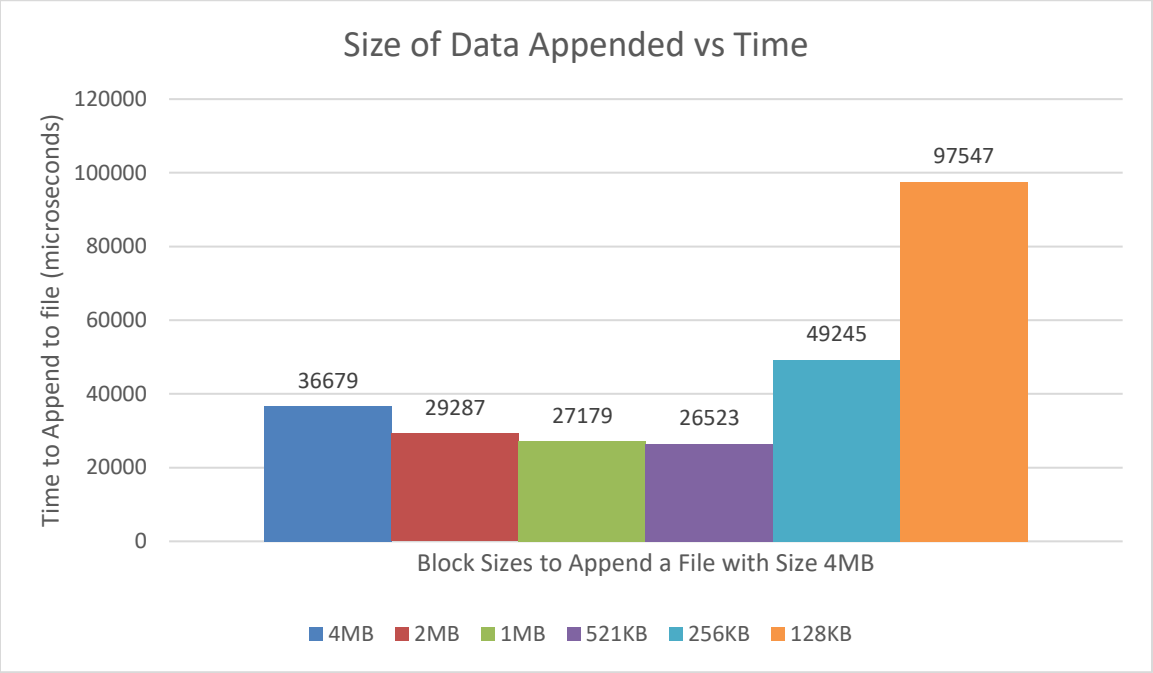
For our experiments we started with clocking the file creation time.

Since the file creation doesn't start with an file size at the begining only the file creation time against # of created files was plotted.



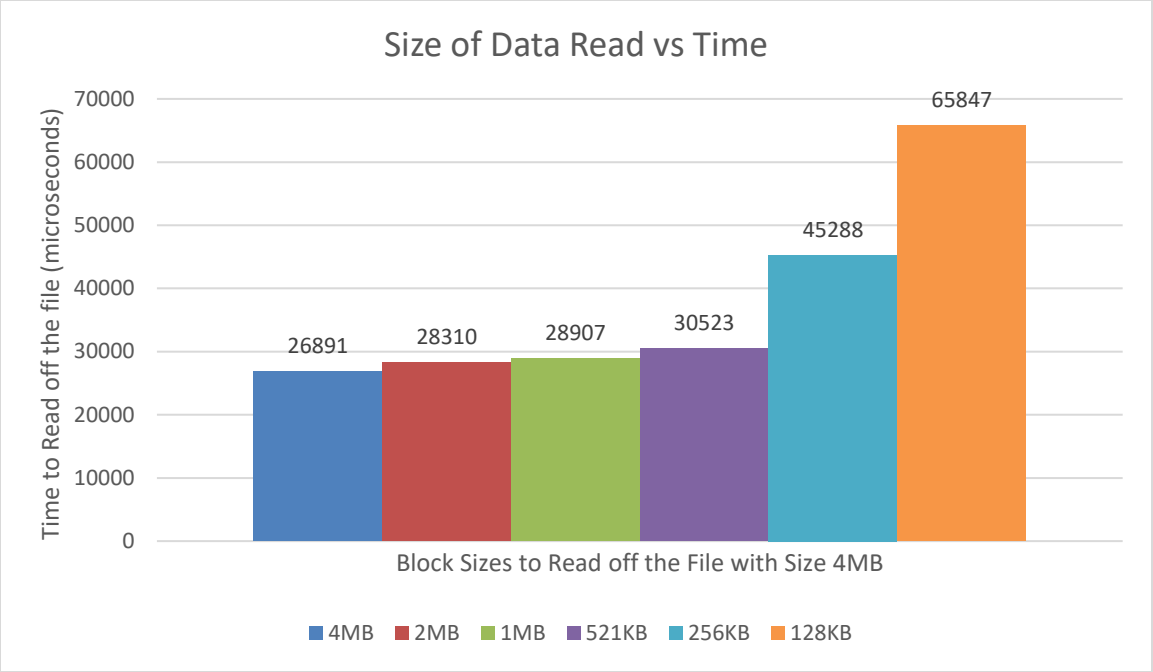
We can see that, disregarding small variations, the file creation is quite linear. As each time we have to create a file we have to access the library from strach and create indexes for the file all the same.

Next, we timed the time it takes to write to a file. Since in this stage we could utilize the size of the file we plotted for different sizes of data to write to our file. What we did was we tried to create a file that was 4 MB big, however we tried a few different methods. So first, we just appended the file the entire 4MB, then we did 2MB appendations 2 times so on and so forth.



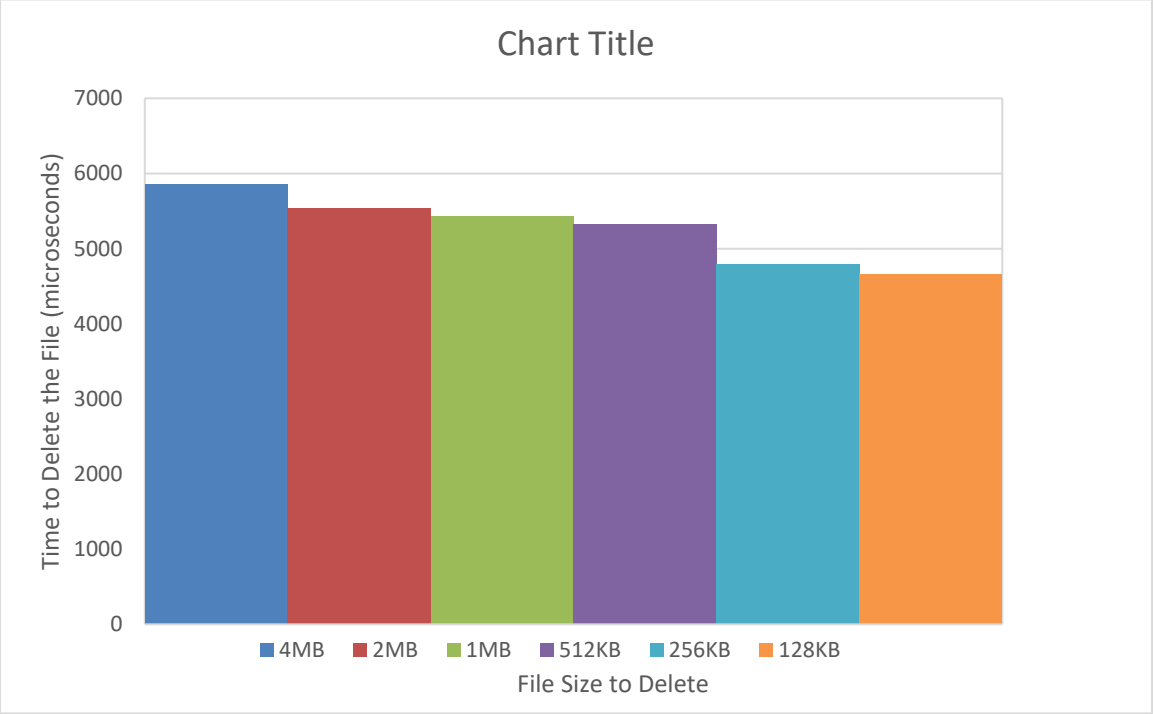
After running these tests numerous times, the results kinda surprised us. We expected the time to expoentioally rise if we shrunk the size of the data we were appending to the file, however the fact that 4MB was somewhat slower than 2 and 1MB data sizes is somewhat unexpected. We assumed the constant index and block creation may have played a role however we are not certain.

After, appending to the file, we timed the read times for files of different sizes. Again utilizing a similar technique we used while appending to the file.



However this time unlike our unexpected results in data written, in our experiments for the read time everything went as we expected. It was clear to us that as the size of the buffer that we were reading to from the file shrunk the time to read grew exponentially. This can be seen in our plots regarding the experiments.

And lastly, we decided to do our last experiments on file deletion and the time it would take to delete a file of varying sizes as we did with the experiments before this one.



And again we got results inline with our expectations. As we have assumed, as the index table and with it the size the file takes up increases the more time it takes to delete the file. The correlation between the deletion time and the size of the file seems to be somewhat linear however our tests where all powers of 2, any non power of 2 file may infact take the same amount as the smallest 2's power would.