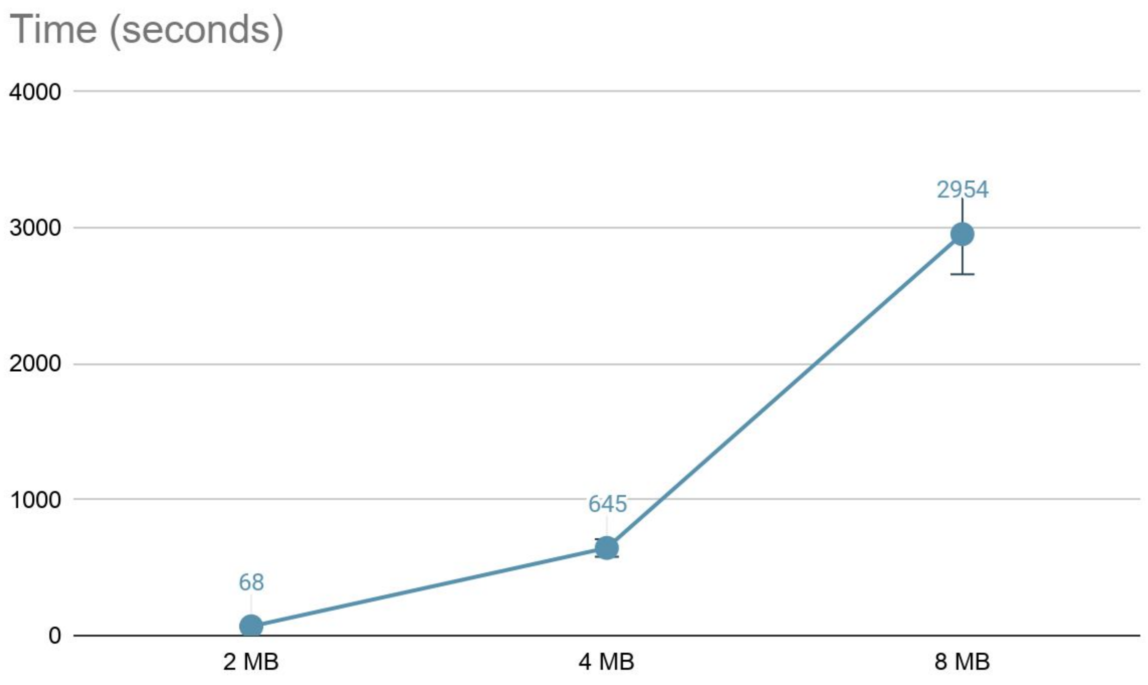
Hakan Sivuk - 21601899
Cevat Aykan Sevinç - 21703201

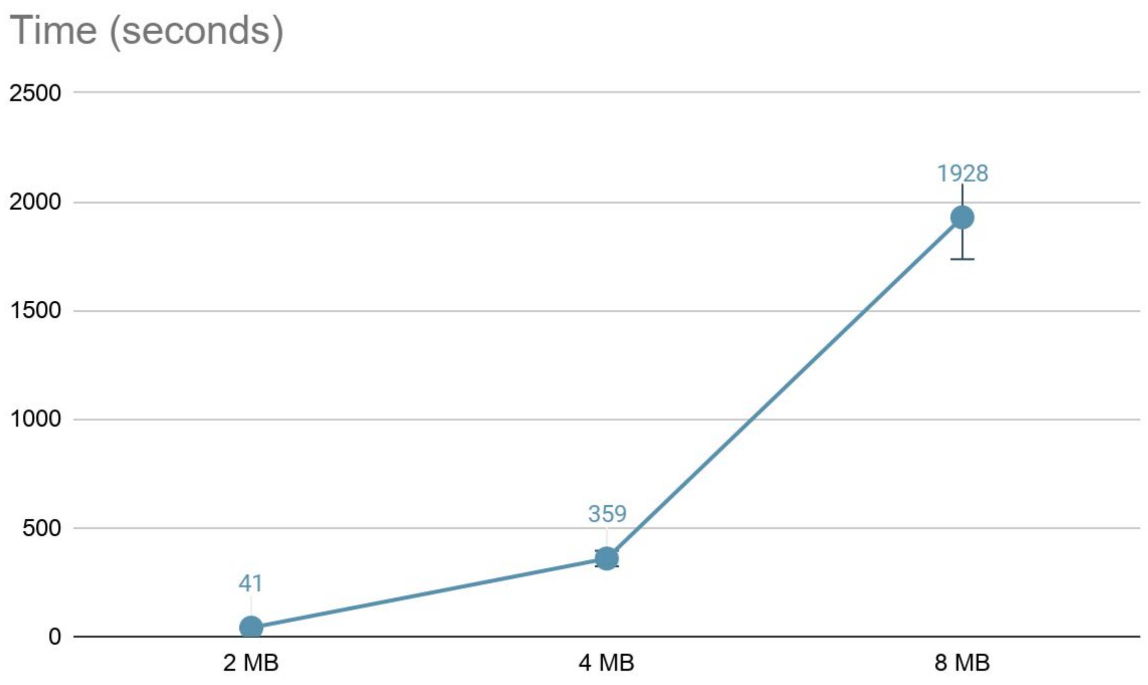
CS342-1 Project 4 Report

Experiments

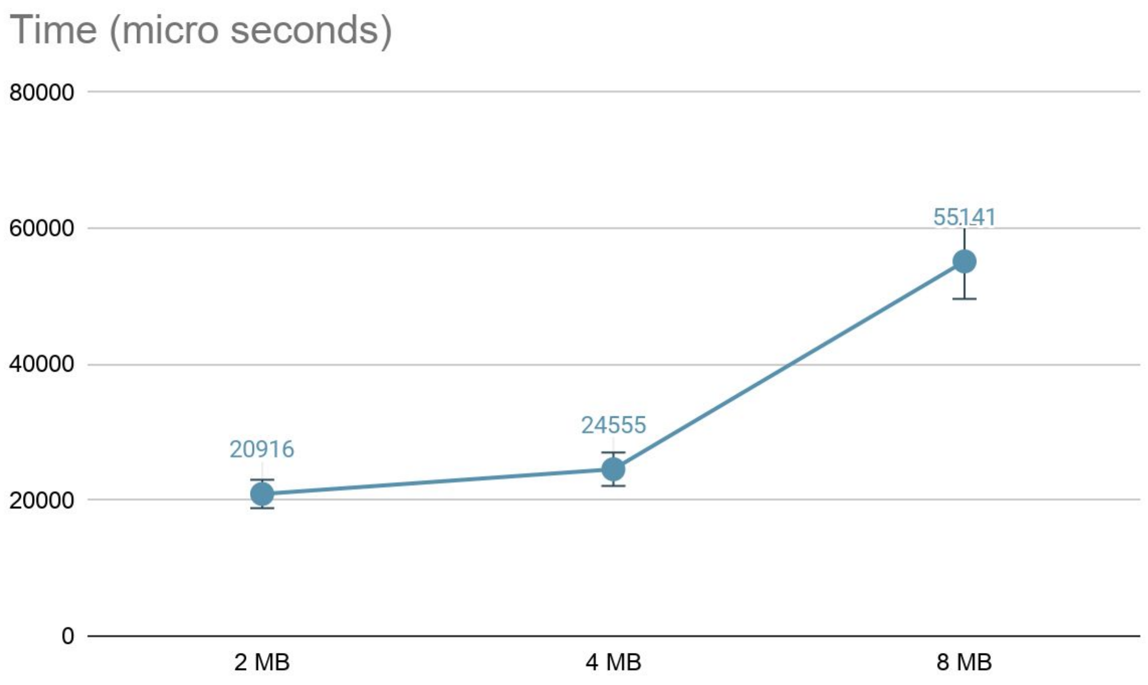
Our experiments are as follows. First, we write a file until it is full. This means we use every available memory in the system. Out max file size consists of 2 MBs, 4 MBs and 8 MBs due to time complexity. Next, we read from the file that is fully written. Ultimately, we delete this file and continue to repeatedly delete and open new files.



We repeatedly wrote 4 bytes to a file until there was not any memory left in the system. As the maximum file size increases, the time to write to a file until it is full increases. This is because the time complexity to write to a file is O(Number of FAT entries * (Available Memory Size at The Start / 4)). We need to traverse the FAT table. Moreover, we need to repeat this process until the memory is full. As a result, time to write to a file increases as we input more and more byte. This can be improved by caching.



Here, we repeatedly read the information that was written in the previous experiment. The time complexity is the same with the previous experiment. However, there is a major difference. The time it takes is almost half of the previous experiment. This is due to the fact that writing to a file is more costly than reading from a file. As a result, this experiment took almost half the time of the first experiment set.



Another interesting observation is related to file system creation. As the file system size increases, the time it takes to set up the system increases. This is due to the FAT set up. There is a linear relationship in the FAT set up regarding time.

Creating and deleting files repeatedly is very easy. Compared to reading and writing, these operations are almost constant time. It took 3949, 3137, 4674 microseconds to delete and create files for every directory. Compared to writing, reading and FAT set up, file creation and deletion is very fast.

File System Design

As it is stated in the project document, we used the first block as superblock. We used four bytes for the number of data blocks, four bytes for the index of the FAT entry that points to the head of the free block list, four bytes for the number of free data blocks and finally four bytes for the number of files created before. These numbers are written to the superblock in this order by starting from the beginning of the block. We initialized superblock as (number of data blocks, 0, number of data blocks, 0)

The directory entry is the following: file name (64 bytes), size of the file (4 byte), fat index of the file (4 byte), int indicating whether this entry is used for a file (0, 1) (1 byte). These information are written to each directory entry in that order from the beginning. We initialized directory structure by setting the used byte of each entry as 0.

The FAT entry is the following: fat index for the next block or -1 if there is no next block (4 byte). Each FAT entry is pointing to a data block with the same index. For example n-th fat entry points to the n-th data block. Therefore, we don't keep the data block number in the FAT entry. We keep data blocks of a file as well as free block list as linked lists. For a file, we can start with the fat index of the file and go until see -1 (last block for that file) as FAT entry. For the free block list, we can start with the index of the FAT entry that points to the head of the free block list, and go until see -1 (end of the free block list) as FAT entry. We initialized the FAT structure by setting each entry keeps the index of the next entry (entry i keeps i+1) and the last entry keeps -1.

We make these formatting operations in the create_format_vdisk function.

When a process uses the vsfs_mount function and mounts the filesystem, we get superblock information from the disk. When it uses the create function, we set one of the empty direct structure entries for this file. When the process wants to open a file, we search the directory structure entries and find the related entry. If it is not already opened, we get this directory entry and keep it in the open table. We keep directory entry of the file in addition to read pointer and open mode of the file. When the process closes a file, we update its directory structure entry in the disk by using the related element of the open table. When the process unmounts the file system, we update the superblock information according to the changes. When the process deletes a file, if the file is still open, we first close the file. Then we search the directory structure and set the used byte of the related directory structure entry to 0. When reading a file we update the read pointer in the open table accordingly. Finally, when the process uses the append function, if a new block is necessary we allocate blocks to this file from the empty block list. At the end, the index of the head of the empty block list is updated accordingly.