CS 342
Project 4

Ata Deniz Aydın
21502637
CS 342-01

The file system implemented stores metadata in a superblock, containing the name, size, number of blocks etc. of the disk, a directory table, implemented as a sorted dynamic array of maximum size 128, an array of 128 FCBs with each directory entry storing the index of the file in the FCB array, and a file allocation table storing the next block for each of the 2^{15} blocks on disk. Each FCB only needs to store the size and starting block of the file, and hence can be stored contiguously instead of in separate blocks. Representing block numbers using 16-bit integers, the superblock can be mapped to the first block of the disk, the directory entry and the FCB array to the next two blocks, and the FAT to the next 16 blocks.

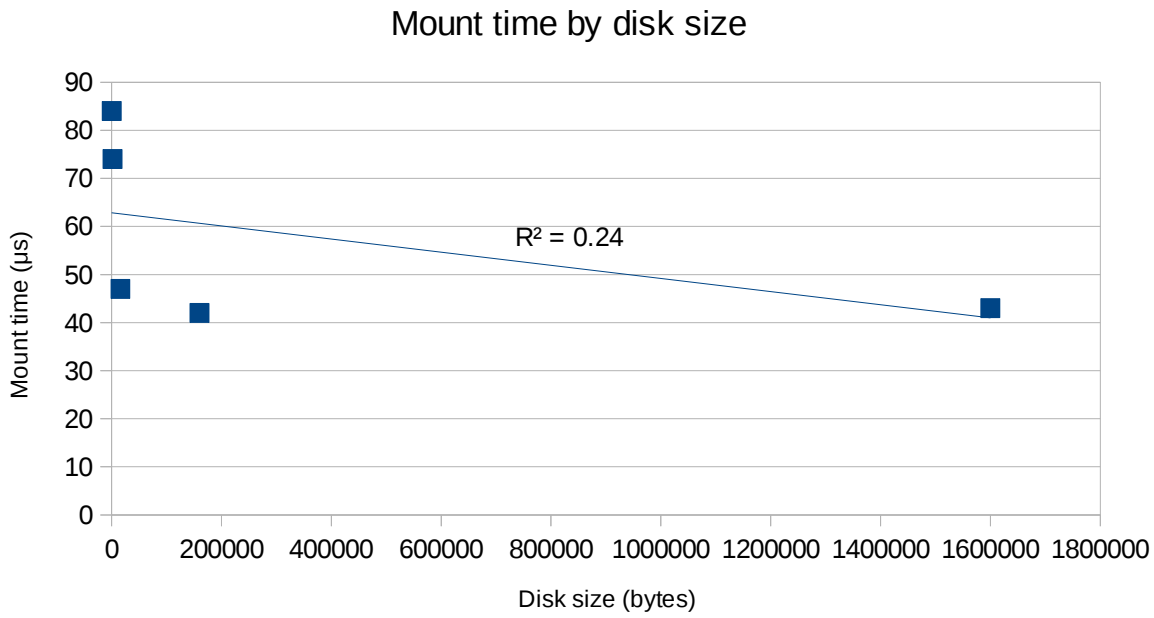
In addition, an in-memory open file table stores a dynamic array of 64 open file entries, each keeping the filename and a pointer to the FCB of the file, as well as the file position pointer and the block it is in. In order to support multiple processes accessing the file system, the directory table and open file table were attempted to be kept in shared memory, but libmyfs.a being a static library it was not possible to link libmyfs.a with the POSIX shared memory library, at least in the given time frame. An alternative would be to utilize the disk itself as shared memory and record in the superblock or some other reserved block the PID of the last process that has modified disk metadata, so that processes would write metadata onto disk after updating and pull them from disk if not up-to-date. This would however have prohibitive disk I/O requirements as it would involve at least one block read or write for each metadata access. Assuming different processes will not access the file system concurrently but only sequentially, in particular one process will mount the disk after the other has unmounted, the current in-memory structures should suffice in accessing and modifying the disk correctly.

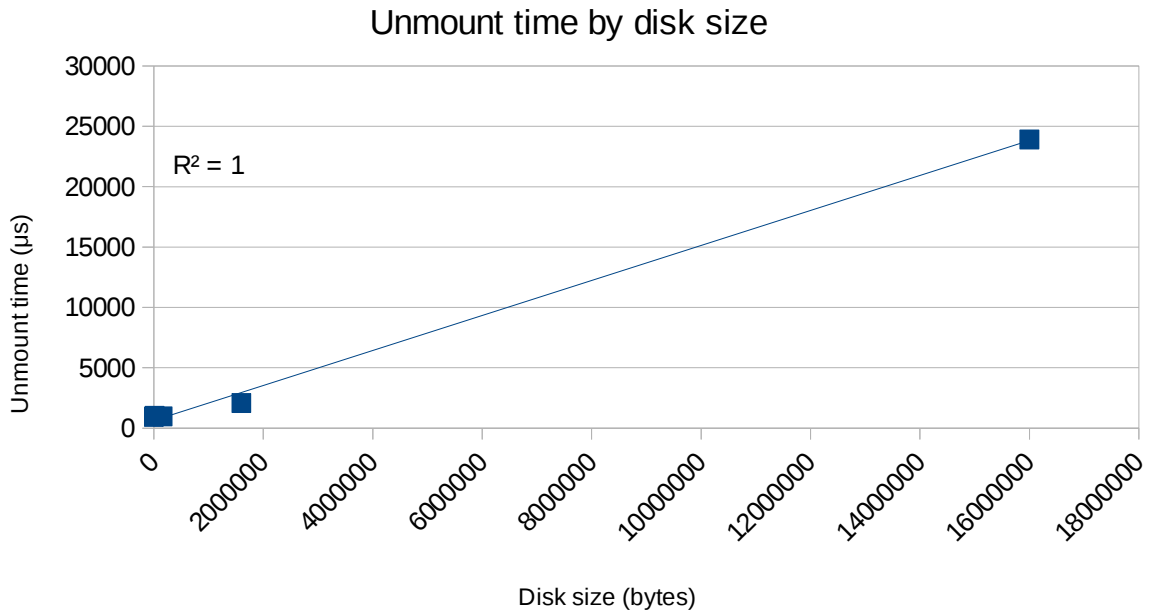
At mount-time, the directory table and the array of FCBs are loaded into memory as an instantiation of struct dir, defined in dir.h/c, and the open file table, defined in opentable.h/c, is similarly allocated in memory and initialized. Unmounting simply saves these on-memory structures into the disk and deallocates them in memory. Formatting fills the blocks corresponding to these structures with zeros. File creation involves adding a new directory entry and FCB entry, opening adding a new open file entry, closing removing the open file entry and deletion removing the directory entry, marking the FCB entry as invalid, and traversing and deallocating each block used by the file. Reading copies the current block into a buffer, writes from the buffer to the array provided, and continues to the next block until the desired read size or the end of the file is reached. Writing works similarly, except it keeps writing after EOF, increasing file size, and allocates new blocks if necessary.

The test program implemented in app.c created 16 files, wrote and read 100, 1000, 10,000, 100,000, and 1,000,000 bytes to each file, opened and closed each file after writing and after reading, and mounted and unmounted the disk after writing to and reading from all 16 files. At the last iteration, before unmounting the disk, each file was deleted. Hence, the program executed 10 mounts and unmounts, 16 file creations and deletions, 160 opens and closes, and 80 reads and writes, and printed the execute time in microseconds of each operation to stderr. The following tables and graphs were collected from one run of the test program, compiled using GCC 5.4.0 and executed on Ubuntu 16.04.4, running on a VM, with a 64-bit 1.8 GHz Intel Core i5 processor and 4 GB RAM.

Mounting and Unmounting

Disk size (bytes)	Mount time (µs)	Disk size (bytes)	Unmount time (µs)
0	84	1,600	931
1,600	74	16,000	1,026
16,000	47	160,000	968
160,000	42	1,600,000	2,076
1,600,000	43	16,000,000	23,915



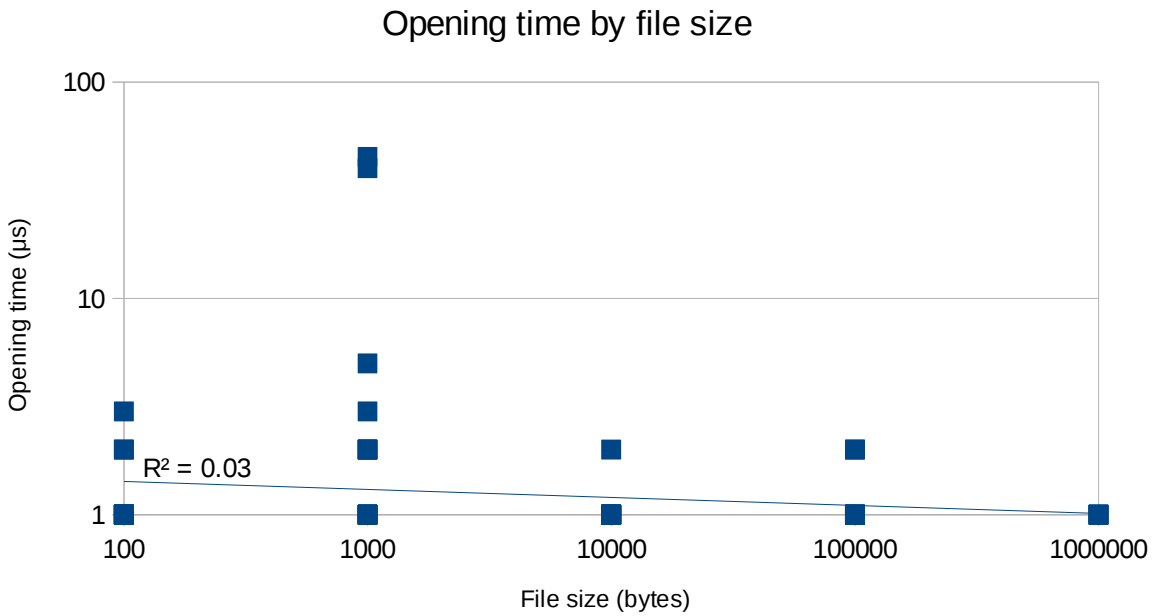


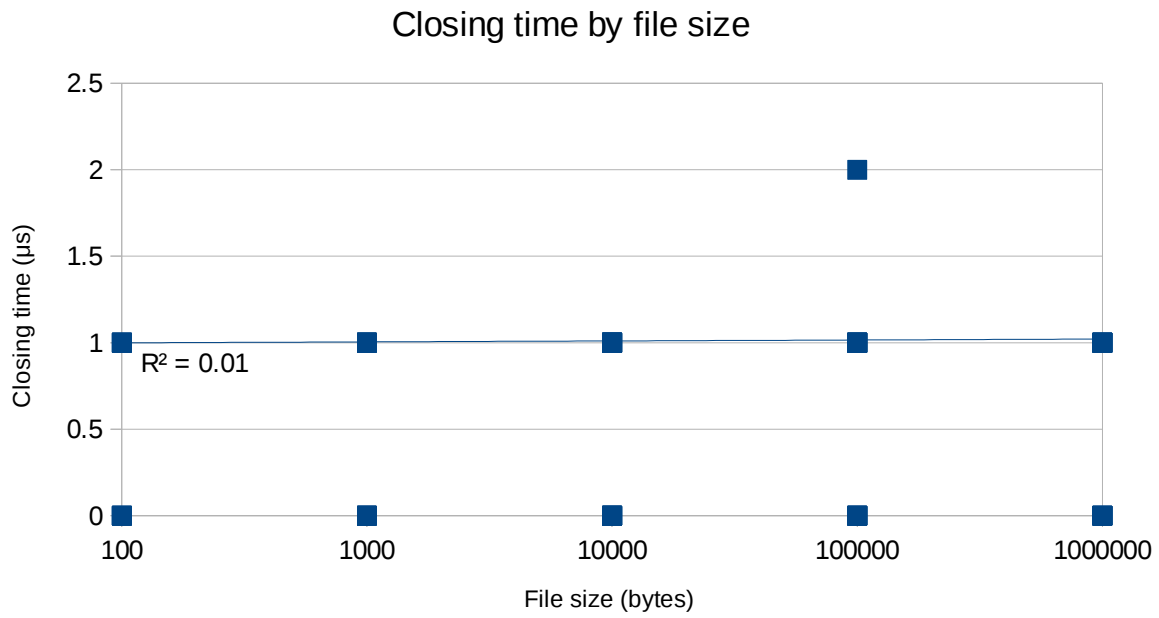
From these figures, we may observe that while the disk was mounted in constant time, unmounting required time linearly proportional to the disk size (not including metadata). This might be attributed to the fact that the blocks written to were kept on a buffer cache and then written to file at unmount time, by way of the fsync system call. All operations for mounting and unmounting other than the template code already provided concerned loading and storing metadata structures, which are of constant size. Note also that the first two mounts and unmounts took longer relative to disk size than the remaining three. This might be due to initial cache or page misses, changes in CPU scheduling or some other initial overhead that is diminished later in the execution of the program.

Creation, Opening, Closing and Deletion

The mean and standard deviation for file creation and deletion time, file opening time by file size, and file closing time by file size are tabulated below.

File size (bytes)	File creation/deletion time (µs)		File opening time (µs)		File closing time (µs)	
	Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
0	0.50	0.63				
100			0.77	0.73	0.27	0.45
1,000			3.70	10.63	0.33	0.48
10,000			0.63	0.56	0.33	0.48
100,000			0.83	0.53	0.77	0.50
1,000,000	513.19	31.98	0.50	0.52	0.28	0.46





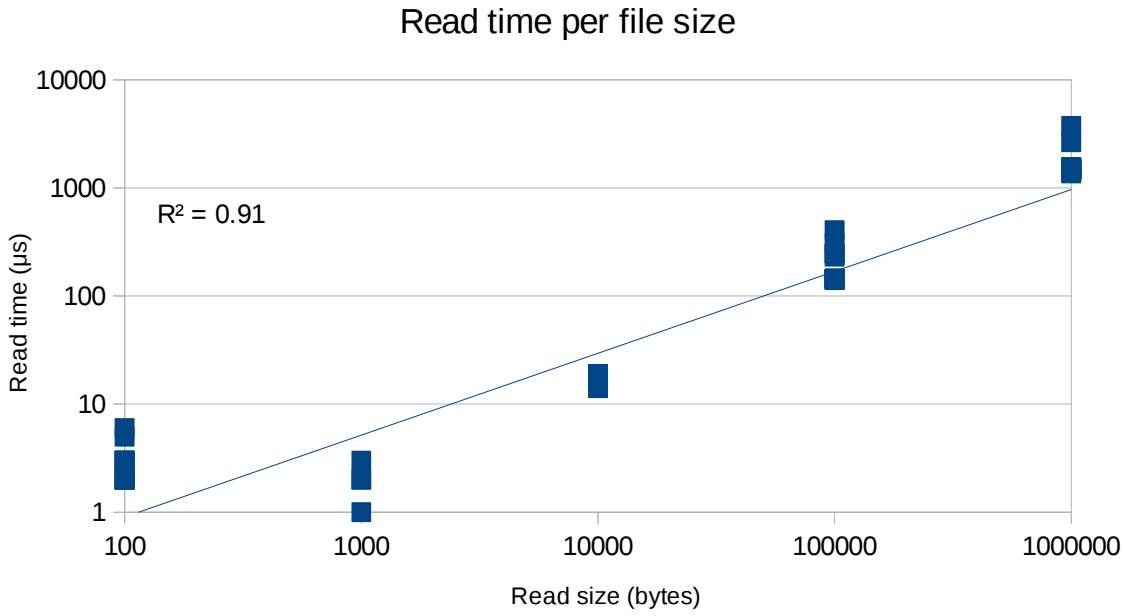
From these charts it can be seen that opening and closing files take the same time irrespective of the file size. This is because exactly the same operations are executed for each call: while opening, the FCB for the file is found from the directory entry and the open file entry is added, and while closing, the open file entry is removed from the table. All operations traversing the file that would take linear time take place during reads and writes, and the data written is saved directly to disk during calls to write, instead of being stored in a buffer and saved during close. Further, the runtime of these three operations are much smaller in magnitude and rate of growth than unmounting and reading/writing, both in mean and in standard deviation, as they only operate on in-memory structures and do not require disk I/O at all.

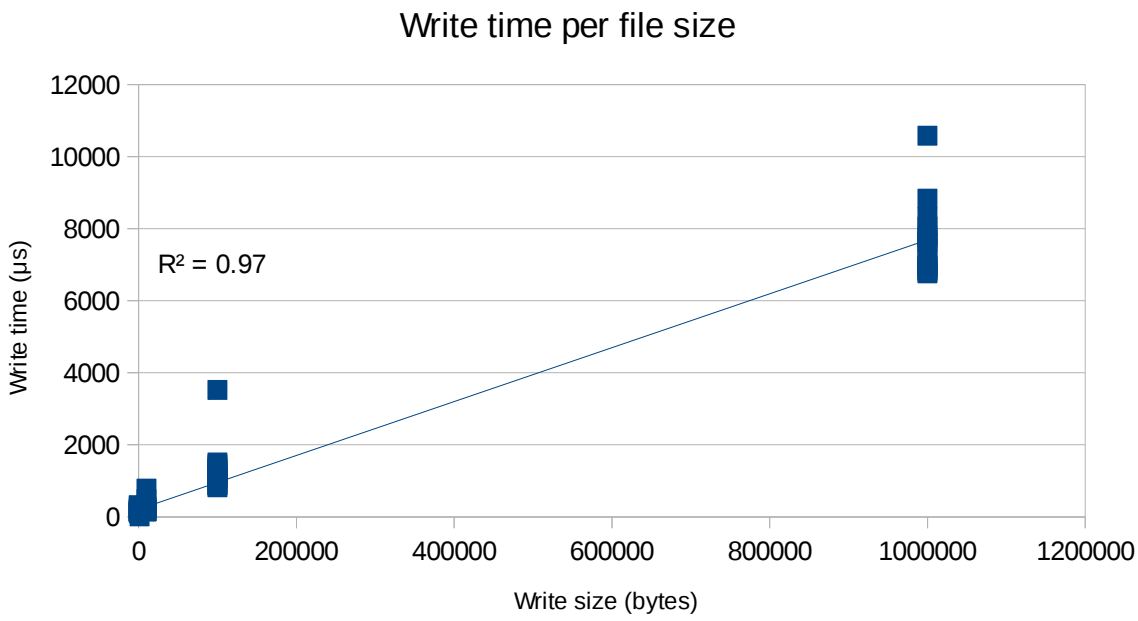
However, because each deletion first traversed the FAT to find and mark as invalid each allocated block for the file in order, reading and writing to each FAT block containing the blocks of the file, deletions took much longer than opening and closing files. And these consecutive disk accesses might have also contributed to the variance in deletion times, due to the arbitrary locations of disk cylinders accessed as well as other external randomness.

Reading and Writing

Since the maximum size for a single read or write is 1 KB, the last three reads and writes were executed in a for loop reading/writing 1024 bytes and a final read/write for the remainder.

Read/write size (bytes)	File reading time (µs)		File writing time (µs)	
	Mean	Std. Dev.	Mean	Std. Dev.
100	2.69	1.20	183.81	61.97
1,000	2.00	0.37	6.06	2.43
10,000	16.06	1.61	243.63	169.57
100,000	188.56	78.98	1266.25	640.72
1,000,000	1689.38	622.64	7658.06	984.93





Here we can observe a strong linear correlation between read/write times and data size, as one might expect. The first write operations also took longer than the subsequent read and write operations in the first two iterations, presumably due to the first blocks of each file being cached in memory after being accessed, as well as the first blocks of each file being allocated the first time they are written to. Both of these may also have caused the fact that writes took more time in general than reads, and also exhibited larger variance. Each write operation dynamically allocates new blocks for data, which requires further disk access as the FAT is stored on disk, and both the FAT blocks and the data blocks accessed during writes and reads may be stored in the buffer cache after writes, further decreasing the runtime of reads compared to writes. The last reads still took longer than file deletions, as reads also involved copying from the data block into memory, not only the FAT block.