Bilkent University

Bilkent University

Computer Science

CS 342 - Operating Systems

Project 4 - Report

Arda Akça Büyük	21802835	Section: 1
Elif Özer	21602495	Section: 1

File System Design:

Content of Virtual Disk

Block 0 (Superblock)
Block 1 (Bitmap)
Block 2 (Bitmap)
Block 3 (Bitmap)
Block 4 (Bitmap)
Block 5 (Directory Entry)
Block 6 (Directory Entry)
Block 7 (Directory Entry)
Block 8 (Directory Entry)
Block 9 (FCB)
Block 10 (FCB)
Block 11 (FCB)
Block 12 (FCB)
-Content & Index Tables- Block 13 . . . . Block (Disk Size / 4KB)

In our file system design, files are stored in a virtual disk and the operations for creating/using files are executed through the virtual disk, as stated in the requirements. Virtual disk is divided into blocks with size 4KB and structures of the file system are stored in the first 13 blocks.

In the first block, Block 0, the superblock structure is stored to contain volume information.

Block 1, Block 2, Block 3 and Block 4 contain bitmap. In order to implement the bitmap free space management, we used an unsigned integer array of size 1024. Each entry in the array is 32-bit and therefore initialized to value INT_MAX (which is basically the maximum value of an unsigned int: 11111111111111111111111111111111). In our implementation, 1's denote free space where 0's denote used space. In the create_format_vdisk() function, we switch the first 13 bits to 0 as we allocate the blocks for the disk structures. Finding and freeing data blocks are done with arithmetic and logarithmic operations besides linear array search.

Block 5, Block 6, Block 7 and Block 8 contain root directory. In order to implement the root directory, we used a structure that corresponds to directory entry and then we created an array that contains 128 directory entries. Each block contains 32 entries of the directory structure and we search each directory when needed. Directory entry structure has file name, FCB index and available attributes that will be used for the operations for using(read/write) and creating files. We completed the size of the struct to 128 bytes with an empty array 'foo' since each dirEntry is 128 bytes. The implementation is as follows:

```
typedef struct dirEntry {
  char fileName[MAX_FILE_NAME];
  char foo[DIR_SIZE-(sizeof(char)*MAX_FILE_NAME)-(sizeof(int)*2)];
  int FCB_index; //FCB index
  int available; //1 if available
}dirEntry

//array that corresponding root directory, DIR_SIZE = 128
struct dirEntry dirStructure[DIR_SIZE];
```

Block 9, Block 10, Block 11 and Block 12 contain the list of FCBs for files. In order to satisfy this requirement, we used a structure that corresponds to FCB table entry and then we created an array that contains 128 FCB entries. We put the FCB table in Block 9, Block 10, Block 11 and Block 12 such that each block contains 32 FCB structures. FCB structure has isUsed, index_table_block, index and fileSize attributes to perform file operations. We completed the size of each FCB to 128 bytes with an empty array 'foo' since each possible FCB is 128 bytes. The implementation of FCB structure is as follows:

```
typedef struct FCB {
  char foo[DIR_SIZE - 4*sizeof(int)];
  int isUsed; //is the FCB is used at the moment (1 & 0)
  int index_table_block; //the block in the disk that contains the index table
  int index; //FCB index
  int fileSize; //Current size of the file
} FCB;

struct FCB fcb_table[FCB_SIZE];
```

Implementation of open file table:

For operations requiring an open file table we used a structure called *openFileTableEntry* corresponding to the entry of the open file table. Then, in order to implement an open file table, we created a global array that contains 16 *openFileTableEntry.* The attributes of open file table entry are name(file name), accessMode (READ or APPEND), file_offset (to trace the last location used in the file), available(1 if that entry is free) and openNum. Each entry in the open file table is 128 bytes. The implementation of open file table entry structure is as follows:

```
typedef struct openFileTableEntry {
  char name[MAX_FILE_NAME];
  int accessMode; //MODE_APPEND or MODE_READ
  int file_offset; //where the current read/write position is
  int available; //if the table entry is available (i.e. no open files on the entry)
  int openNum; //how many processes opened the file
} openFileTableEntry;

struct openFileTableEntry openFileTable[MAX_FILE_OPEN];
```

Other implementation details:

In our file system design, we used an index table as an integer array of size 1024. Each file has an index table in order to store in which block the content of the file is. With this implementation, we can perform read and write operations. Initially, the index table of the file has -1 in each index. When we write to file, the index table stores the block number that contains written data. For example, when we create a file named “test.txt”, there are 1024 number of -1 in the index table of “test.txt”. When we try to write some data in test.txt, first we search for an empty block in the disk and then we write to the free block found. If the found free block is 65 and we write to the block 65, the first element of the index table becomes 65 indicating that content of the test.txt is stored in that block. If we want to

perform a read operation on test.txt, first we look at the index table of this file and get the block number that the content is stored in so that we can read the data from the block. We simply implemented the index allocation type.

We put the file content and index tables into blocks starting from 13 to (Disk Size / 4KB). We put them into blocks by finding empty blocks. For example, if we first create a file, we put its index table into Block 13. Then, if we create another file, we put its index table into Block 14. Also, if we want to write the first file we write into Block 15 and the implementation will be done in this way.

Experiments:

1-Experiment on read time

We first created and opened a file named “file1.bin” in MODE_APPEND, then wrote 111.100 bytes on it. After writing 111.100 bytes, we closed the “file1.bin” and then we opened it in MODE_READ so that we can read bytes from this file. In this experiment, we calculated read time on different amounts of bytes. We read 100 bytes, 1.000 bytes, 10.000 bytes and 100.000 bytes by calling sfs_read() function, reading 1 byte at each read. Thus, we call sfs_read() for n times to read total n bytes.

Additional info

Create time: 185 ms

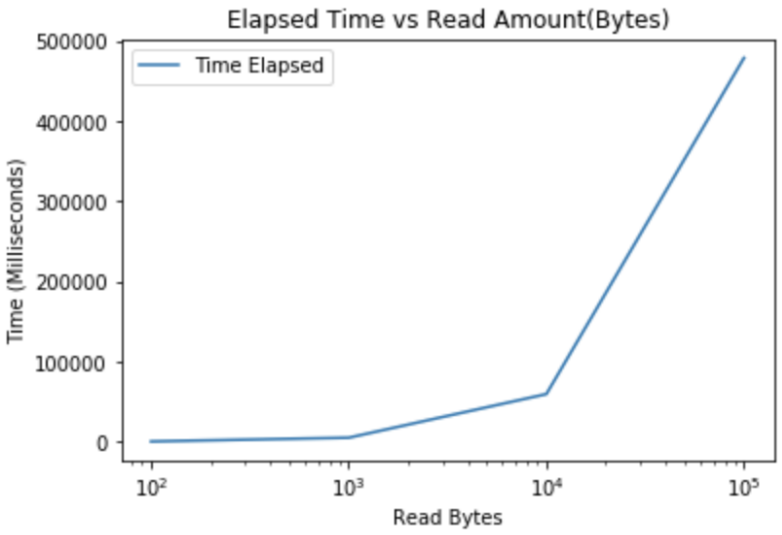
Open file time: 19 ms

Close file time: 38 ms

Total write time: 2868934 ms

Read Amount	Elapsed Time (to read 1 byte per read) (ms)
100 bytes	460
1.000 bytes	5116
10.000 bytes	59512
100.000 bytes	478802

Table 1. Total Read Amount(one per each call) vs Time



Graph 1. Total Read Amount(one per each call) vs Time

Conclusion of Experiment 1:

As can be seen from the results, the total number of bytes read has an effect on the elapsed time. As the total number of bytes read increases, elapsed time also increases, as expected.

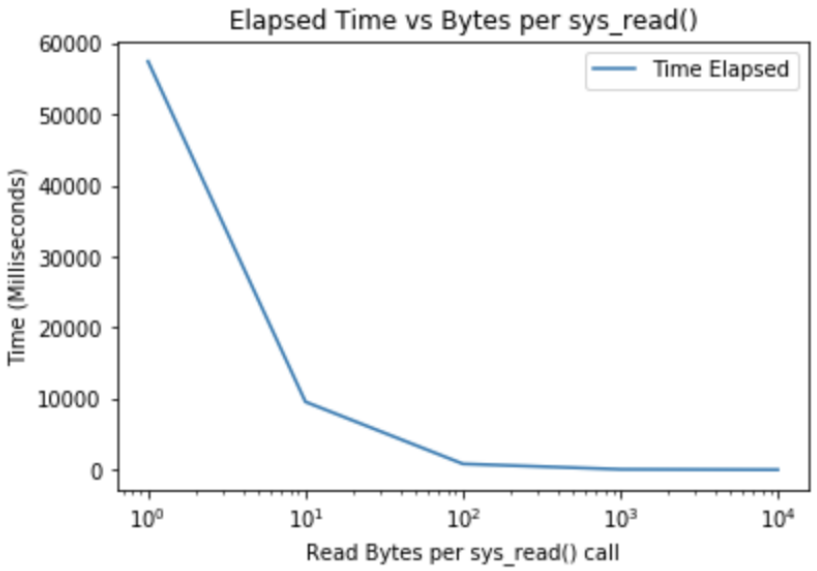
2- Experiment on read time with different read amount of bytes

In this experiment, we first created and opened a file named “file2.bin” in MODE_APPEND. Then, we wrote 50.000 bytes on it. After writing 50.000 bytes, we closed the “file2.bin” and then we opened it in MODE_READ so that we can read bytes from this file. The aim of this experiment is to observe the effect of the number of read bytes per each sys_read() call on time. Thus, we call the sys_read() function (for 10.000/n times) to read 10.000 bytes by reading n bytes for each call.

Additional info
Create time: 2 ms
Open file time: 21 ms
Close file time: 9 ms
Total write time: 1353525 ms

Amount of read bytes per sys_read()	**The number of sys_read() calls**	**Time(ms)**
1 byte	10.000	57495
10 bytes	1.000	9558
100 bytes	100	825
1.000 bytes	10	55
10.000 bytes	1	0

Table 2. Amount of read bytes per call vs Time



Graph 2. Amount of read bytes per call vs Time

Conclusion of Experiment 2:

As can be seen from Table 2 and Graph 2, the amount of bytes read per sys_read() call has an effect on the elapsed time such that they are inversely proportional. As the amount of bytes read increases per each sys_read(), elapsed time to read total 10.000 bytes (per each experiment) decreases. We associate this inverse proportion to the various operations (checkings, searches) in the sfs_read function. The total number of bytes read stays the same, however, the number of checks and searches done in the sfs_read function decreases, therefore, the elapsed time decreases as a result.

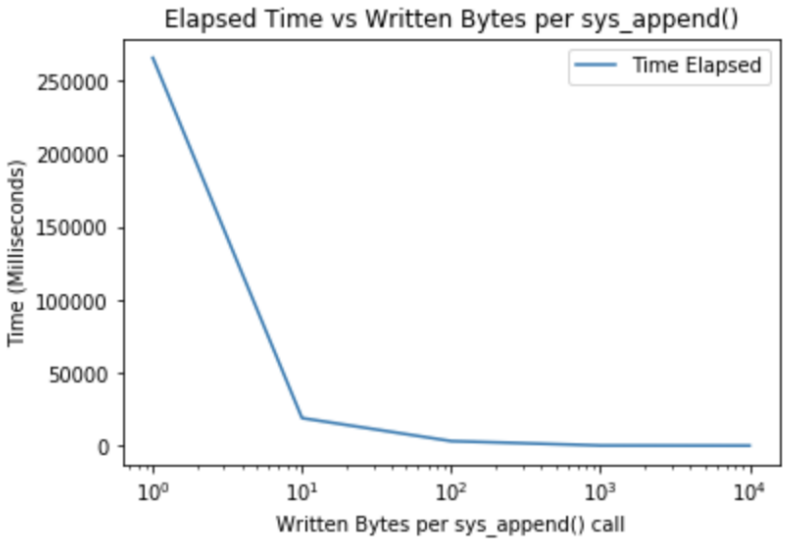
3-Experiment on write time

In this experiment, we created and opened a file named “file3.bin” in MODE_APPEND, then we wrote 10.000 bytes to it. We repeated this process 5 times (in total, 50.000 bytes) to compare bytes written per sys_append() vs time. The aim of this experiment is to observe the effect of the number of written bytes per sys_append() call on the time.

Additional info
Create time: 148 ms
Open file time: 18 ms
Close file time: 8 ms

Written bytes per write()	The number of write() call	Elapsed Time to Write 10.000 bytes
1 byte	10.000	265986
10 bytes	1.000	19224
100 bytes	100	3436
1.000 bytes	10	435
10.000 bytes	1	338

Table 3. Written bytes per sys_append() vs Elapsed Time



Graph 3. Written bytes per sys_append() vs Elapsed Time

Conclusion of Experiment 3:

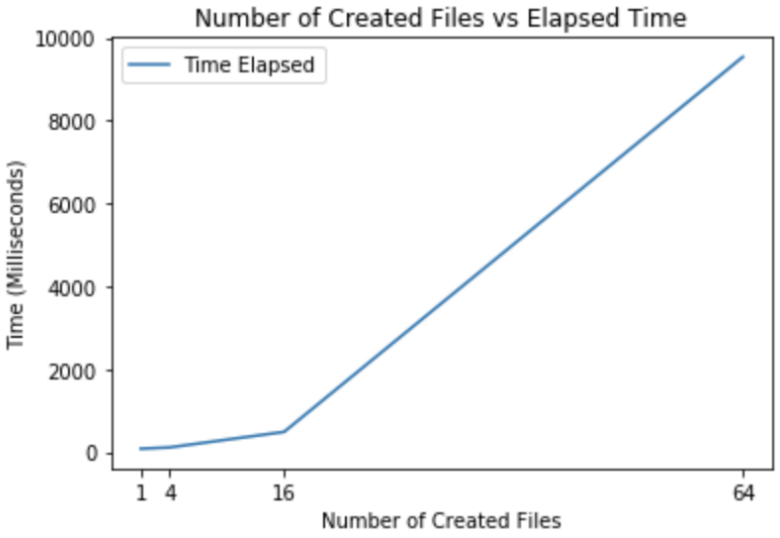
As can be seen from Table 3 and Graph 3, the amount of bytes written per sys_append() call has an effect on the elapsed time such that they are inversely proportional. As the amount of bytes written increases per each sys_append(), elapsed time to write a total 10.000 bytes (per each experiment) decreases. We associate this inverse proportion to the various operations (checkings, searches) in the sfs_append function. The total number of bytes appended stays the same, however, the number of checks and searches done in the sfs_append function decreases, therefore, the elapsed time decreases as a result.

4-Experiments on Creating Files

In this experiment, we measure the time for creating files. We create different numbers of files to observe the number of created files on time. We create 1, 4 ,16 and 64 files respectively and then observe the elapsed time to create 1, 4, 16 and 64 files.

Number of Created Files	Elapsed Time to Create Files
1	79
4	110
16	485
64	9538

Table 4. Number of Created Files vs Elapsed Time



Graph 4. Number of Created Files vs Elapsed Time

Conclusion for Experiment 4:

As can be seen from Table 4 and Graph 4, the number of created files has an effect on the elapsed time such that there is a positive correlation between them. As we expected, the elapsed time increased as the number of created files increased.

GitHub Link:

https://github.com/elifozerr/CS342_Project4_FileSystem