CS 342 Operating Systems

Spring 2021

Project 4 Final Report

Simple File System with Indexed Allocation

Zeynep Cankara

21703381

Section: 1

Instructor: İbrahim Körpeoğlu

Introduction

The indexed allocation strategy helps us prevent external fragmentation caused by allocating the contagious set of blocks [1]. In the indexed allocation scheme with a file system only consisting of a root directory. We need to access the root directory first, get the index block, and then use the corresponding file's index block to get the address of any disk block. However, this method of allocating block comes with the downside of the wasted space used for allocating the index blocks. Each index block limits the size of the files allocated using the system since we can only allocate (BLOCKSIZE / DISK_POINTER_SIZE) number of blocks for a single file bounded by the block size.

Implementation Details

The following figure describes implementation details of my simple file system that I implemented for the project:

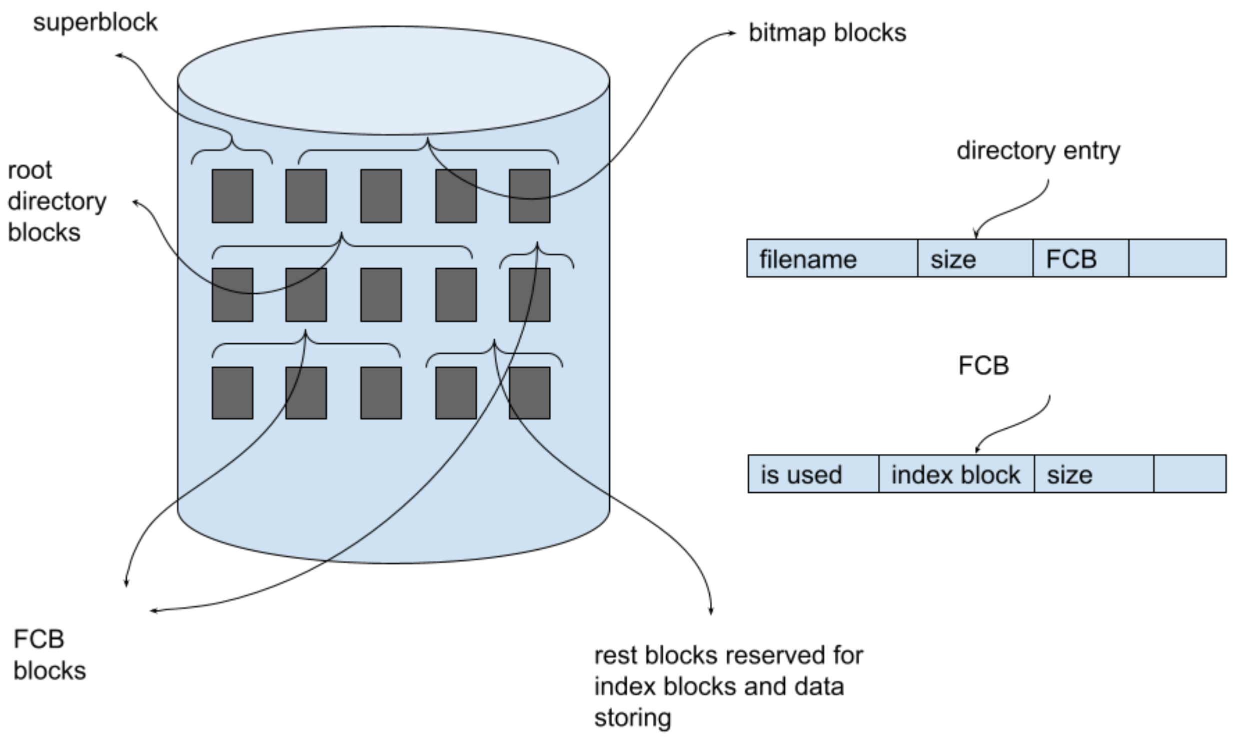

Figure 1. Visualisation of the implementation details of the disk structure

As a summary of the figure, Block 0 reserved to store the volume information. In Block 0, I stored the superblock information: the number of files using the system and the number of free blocks available. Blocks {1, 2, 3, 4} reserved for storing bitmap, which helped me access blocks' availability. Blocks {5, 6, 7, 8} reserved for the root directory which contains the directory entries. Each directory entry stores filename and FCB block information: the FCB block index, FCB block no, and offset for that specific directory entry. Additionally contains the size of the file associated with the file directory entry. Blocks {9, 10, 11, 12} reserved for the File Controller Blocks (FCB). They contain the necessary information for the index block number reserved for the file and has a bit indicating whether the FCB is in use.

Furthermore, I implemented an *open file table* for tracking the open files in the system. Within the open file table, I stored the struct *File* within indices. The struct *File* contains file related information such as directory entry associated with the file, the disk no and disk offset the directory entry stored in the system and a pointer keeping track of the read operation position upon reading from the file consecutively.

Experiments and Results

Experiment 1)

The testing time it takes to *create_format_vdisk* for various sizes.

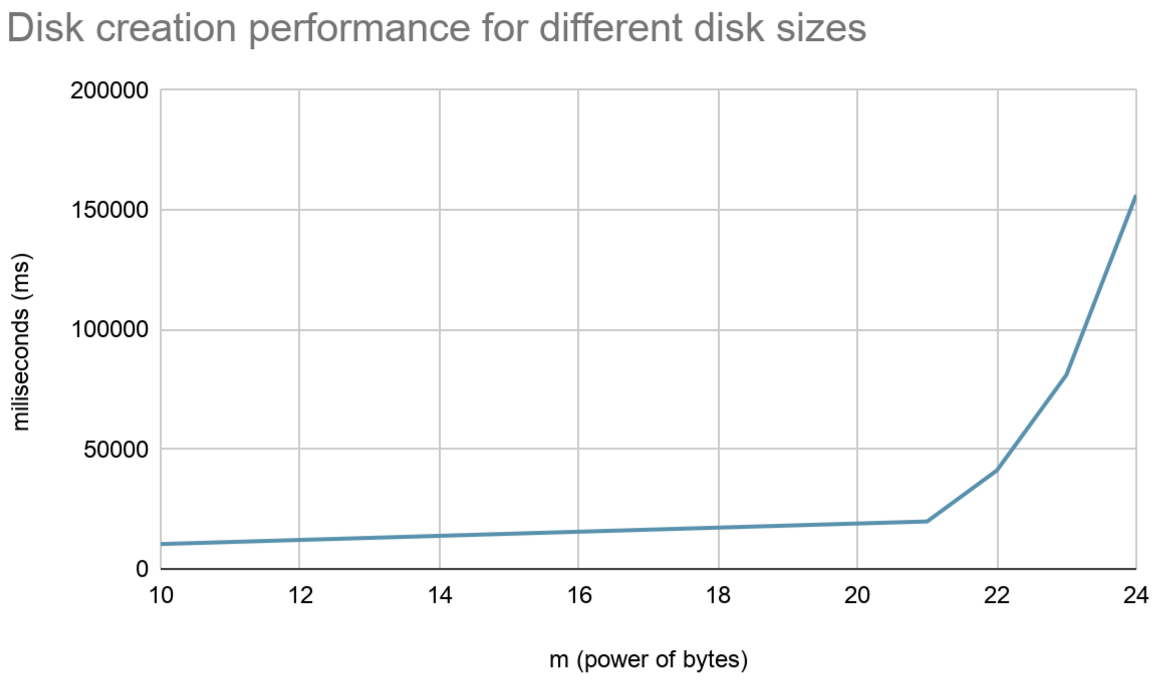

Figure 2. Change in disk performance for different disk sizes of 2^m

Time (ms)	10412	19853	41163	81149	156261
Disk space (2^m)	20	21	22	23	24

I observed that the time it takes to create and format the disk increases exponentially with the file size which is in accordance with my expectations since the size is also increasing exponentially.

Experiment 2)

The testing time it takes for consecutive *sfs_append* calls until all blocks allocated.

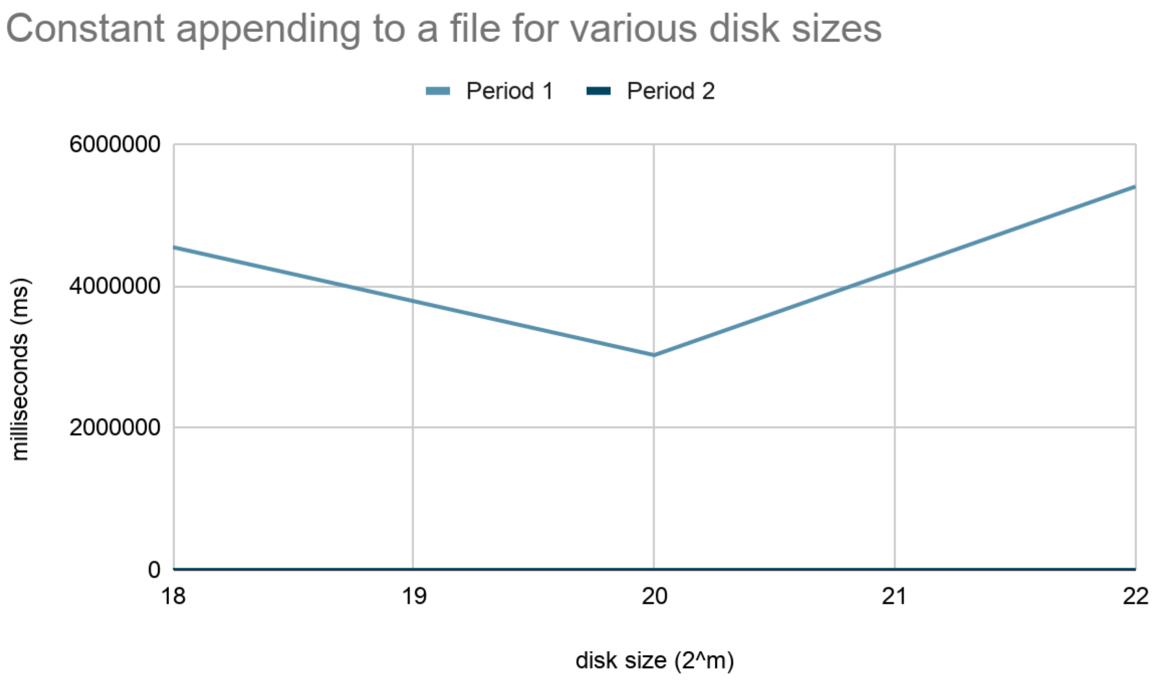

Figure 3. Timing experiment with constant appending to a file

Time (ms)/(s)	4544921/0	3024577/3	4777285/5	5399216/6
Append (bytes)	18	20	21	22

I observed that writing to a file is a costly operation in general after obtaining results from the reading tests. I worked with a small sample size but in general, the time it takes for writing to a file increases linearly as the disk size increase exponentially. Additionally, the dynamic access scheme in the indexed allocation could be worsening these results further given that writing to file already a costly operation.

Experiment 3)

The testing time it takes for consecutive *sfs_read* calls.

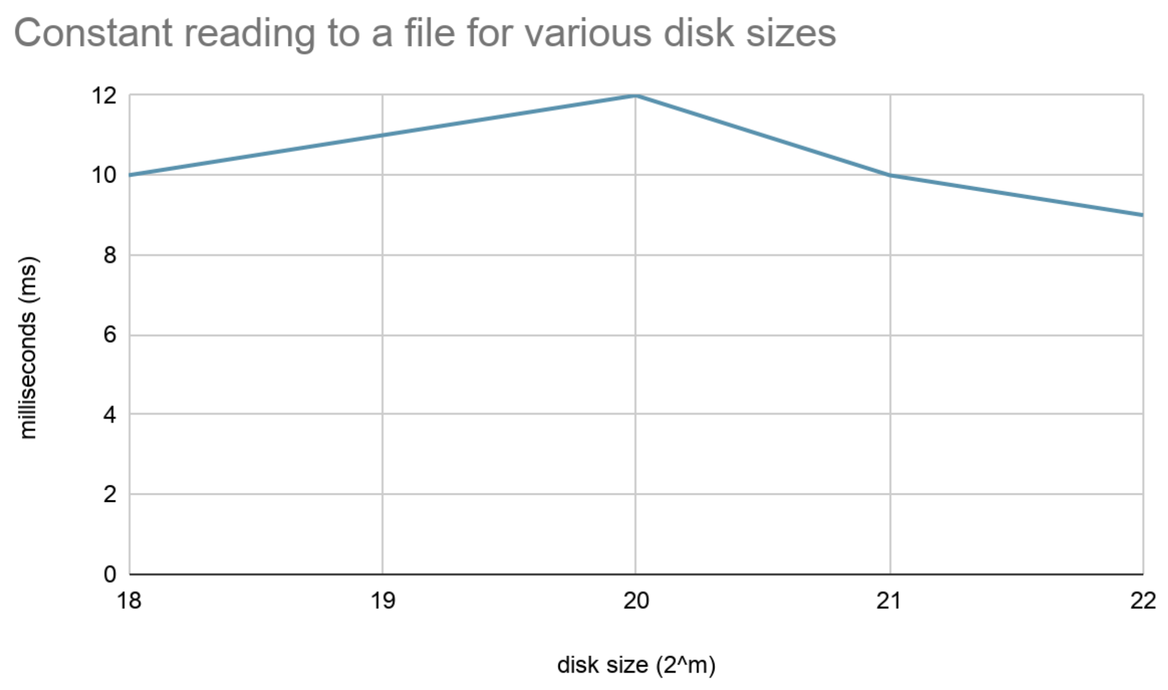

Figure 4. Timing experiment with reading from a file

Time (ms)	10	12	10	9
Read (bytes)	18	20	21	22

I observed a significant time decrease when compared with writing to a file. A possible reason might be that writing to a disk is an I/O costly operation when reading is not so expensive I/O operation. Overall reading from a file did not change rapidly with the disk size but one reason might be that it is not a costly operation in the first place.

Experiment 4)

Continuous read and write operations test and testing the whole interface of the simple file system. This test can be run from the "test.c" file.

- For 1MB disk space, it took 7704 ms to create the file system

- Then tested creating 120 files and opened 16 of them which was the upper limit in the project specification.

- Then append until the end of a file which took 2076226 milliseconds which is 2 seconds.

- Then read until the end of the file which took 9 milliseconds.

- Then tried deleting and re-creating every file which took 846 milliseconds.
- Afterwards, successfully unmounted the system

Conclusion

This project helped me to understand the various disk allocation schemes, especially the indexed allocation, better. Implementing a simple file system interface with the specified disk allocation scheme was both a fun and educative experience. I understood downsides of the using some allocation scheme better.

I realised that the dynamic access without the external fragmentation comes with the cost of index block overhead from the experiments. Thus, indexed allocation causes a performance decrease for the file systems that are performing continuous write and read operations. Using the indexed allocation, we cannot use the direct access schemes in contiguous space allocation and result in increased time in writing to files and closing, deleting them.

References

[1] Disk Allocation Methods.
http://www2.cs.uregina.ca/~hamilton/courses/330/notes/allocate/allocate.html