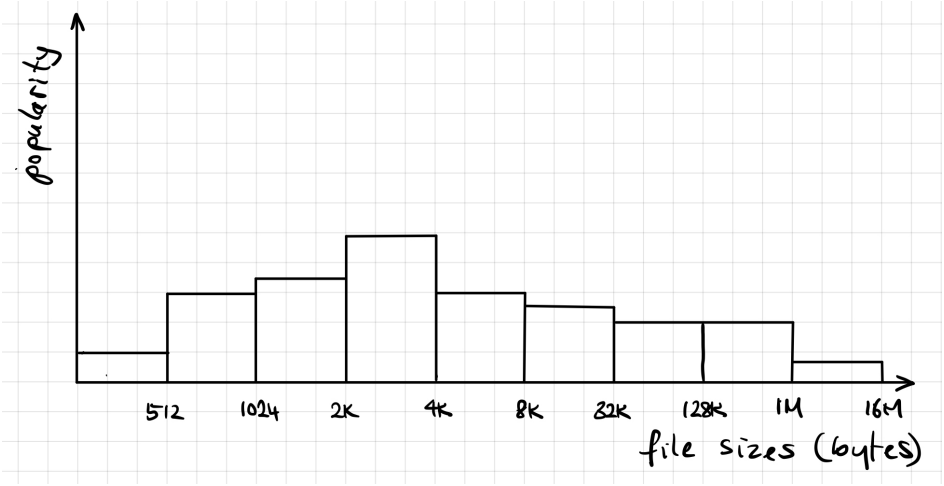
Design and Analysis of a File System

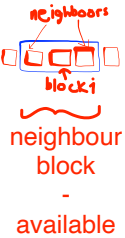
Check My Courses for Due Date

High-Level Problem Description

You have a very large rotational disk drive ($2\ TB$). You have very large number of files that need to be stored in the disk. The total number of bytes required to host all the files is Q bytes – that is the sum of the sizes of all the files. In this assignment, you are required to design a file system that goes on top of the disk drive to host all the files. The total number of bytes in the files in TB is constrained as: $1 < Q < 2$. The file size distribution is given by the following histogram.



The main memory of the computer is 16 GB. You can set aside about 5% (at most) for caching purposes. That means you cannot assume that you can cache large number of disk blocks in memory. Still, you need caching to gain performance. In particular, meta information such as File Allocation Tables, i-nodes, directory blocks need to be cached. Due to memory limitations, you cannot bring in a whole lot of these blocks into the memory. From the file system side, we are accessing the disk as blocks. The disk itself is viewed as a large array of blocks. The blocks are consecutively numbered and the numbers that are close to each other point to disk sectors that are close to each other. This means when we are at block i it is cheap (in terms of access time) to access nearby blocks (i.e., blocks that are close to i and costly to access far away blocks). To keep things simple, we split the available blocks with respect to a given block i into two sets: its neighboring blocks and outside its neighboring blocks. Ideally, from a performance perspective, we want to keep the information that is going to be accessed next (i.e., related to the file connected to block i) in its neighborhood.



Given the disk size, we can expect most of the key data structures to be in multiple data blocks. For instance, the FAT table (in a FAT table-based file system design) will be in multiple data blocks. Same way in an i-node-based file system design the i-node table will be across multiple data blocks. Depending on the size of a file, the file could be in any number of blocks. With multiple blocks, allocation of the blocks on the disk becomes an important issue. Also, the location

of the data blocks with respect to the locations of the directory blocks and i-node tables also becomes an important issue.

Let's assume the following regarding the files. We have almost all the files at the file system creation time. We also know the access (both reading and writing) patterns of the files. Using the access patterns, we can group the files into clusters of files that are going to be accessed close to each other. That is, if a file in a cluster is currently being accessed, it is highly likely that the next access will be to a file in the same cluster. While most of the file accesses are reads, we also have writes. However, the write accesses don't grow the files by much. Over a long period of time (e.g., for 12 months), the total file size (the total number of bytes held in storage) grows by less than 10%. That means the writes are mostly updates that are overwriting existing information that is not growing the file sizes. The number of new files created, or old files deleted is very smal – almost negligible. As a result, we can assume that the directory sizes are staying nearly the same over long period of time.

The applications using the files are accessing them as follows. The applications specify the filename and open a file. Read portions of the existing data from the file. In some cases, the applications would write new data into the file and then close the file. The applications are keeping the association with a file for a short period of time. That is, a file is not kept open (or active) for a long period of time. A file could be opened for read only or read and write. In most cases, the applications are opting for read and write permissions. In some cases, the applications are opting for read only access.

Candidate File Systems

In this assignment, you are expected to create a design for the file system and then analyze its performance. In particular, you need to provide a reasoning for your design choices that shows why your design is going to perform very well for reads and writes. In particular, given the way an application is accessing the files (as described above), you need to reason why the performance delivered to the application would be high (i.e., why the number of application operations processed per unit time – would be high).

Candidate #1: *FAT File System*. In this file system, we use a FAT table for file space allocation. You can use a similar data structure for free space or a bitmap for the free space management. It is important to keep the restriction in terms of memory usage in mind. For instance, the bitmap caching or FAT caching can be consuming large amount of memory. With a limitation in memory use, you cannot bring the whole of FAT table or free bitmap into the memory. You could bring in portions of the disk data structure. If you want to use this architecture, you need to improve the base architecture to maximize locality. In a naïve FAT table-based file system, the FAT table will be in portion of the file system and free bitmap will be in another portion. The data blocks will be the rest. Although it is a simple structure, you can end up doing many disk accesses (i.e., disk head movements).

Candidate #2: *i-node File System*. The i-node File System is similar to the FAT table-based file system. The major difference is the replacement FAT with the i-node table. In this candidate architecture, you have a single i-node table (like shown in the lecture slides). As explained above, the file operations can create disk head movements across many tracks – lot of time seeking on the disk. This means the disk throughput can be reduced.

Candidate #3: *Clustered i-node File System*. The key idea of clustered i-nodes is to split the i-node table into different clusters. Instead of having a single i-node table at the beginning of the disk, we have multiple clusters distributed in the disk. The free bitmap (or any structure you may use for tracking free blocks) is also distributed into the clusters. The directories are also distributed. You need to maintain global information in the superblock (or something similar) about the distribution of the clusters. This way you can access the different clusters very efficiently. If you select this architecture, you design needs to clearly present all the necessary details.

What You Do?

By default, you should not use more than 100 words for answering any question. When there is a different word count required/allowed it will be specified (check the number at the end of the question).

What is the block size you want to use? Why did you select the block size value? (**50 words**)

What is the file system candidate you want to use? Why? (**250 words**)

Give a diagram to illustrate your file system design. Mark all important components there.

Give the pseudo code for the following operations: create, open, read, write, close, seek (**should not exceed 300 lines in total**). Pseudo code *must be self explanatory*. If you add comments to explain concepts, they count towards the pseudo code lines.

Optionally give the pseudo code for the performance improvement helper functions that could be used in the above implementation (**100 lines in total**).

Other explanations (**500 words**).

Some Additional Information

Application operation is the following: open a file, read and/or write certain amount of data from or to the file and close the file. The application would specify the name of the file on which it wants to perform the operation. An application can work on many files during its lifetime. There can be many applications in the system. The performance of the file system is measured in terms of number of application operations that can be completed in a unit time. We prefer to have a file system design that will allow the applications in the system to perform as much activity as possible.

Evaluation

1. Reasoning for block size selection (**10 points**)

For block size selection, you need to think about internal fragmentation. The space wasted in the internal fragmentation balloons the space requirements. For instance, if you waste 50% of the space, you double the space that is required to store the files. Depending on how much space you have and how much data you have to store, you need to control the internal fragmentation. Also, you need to be concerned about future growth for about a year. The block size also impacts the throughput you get from the file system. A small block size means there will more disk movement because you need to access different blocks.

2. File system architecture selection and reasoning (**5 points**)

Here you are given 3 candidate architectures. You can select one of them. You can optionally select another one from the textbook (e.g., WAFL). You need to rationalize your selection. The way to rationalize is to present the advantages your selection presents against other possible selections. In this assignment, we are concerned about access performance for the application. That is the number of application operations we can complete per unit time is the major consideration. You need to explain why your architecture is better than other candidates for obtaining higher performance.

3. Details of the selected architecture (**15 points**)

For the selected architecture you need to provide the details. For instance, what information you would store in the super block and how that information is going to be used by the file operations (i.e., open, read, write, etc). In the case of clustered i-nodes, you need to specify you would put the clusters, what you would include in the clusters (i-nodes, directories, data blocks, etc). This section need not include reasoning, instead it is focused on providing the details of your design. You need to provide the details at the level of what is required for an implementation. A programmer must be able to take your detailed architecture and convert it to an implementation without asking any further questions! ✳

4. C-like pseudo code for create (**5 points**)

The create call is going to place the file in the file system. Because we need performance in this file system, you can do some enhancements that are typically not done like pre-allocating a block at creation. That is, at creation time you create a directory entry, *i*-node (or FAT entry if you select that organization), and an empty data block. This way you can influence the rest of the data block allocations (this is just a suggestion!). Where a create call would place a file would depend on the locality you guess for the file. The locality is the clustering of the files. That is, you want to place a file close to other files because the reads and writes to this collection of files are going to happen at the nearly the same time (i.e., temporal locality). By placing the files close to each other, you can keep the disk head movements localized to certain tracks. If you are not careful in placing the files (i.e., did not care of the locality of access), the disk head would still move large distances and result in poor disk throughput. The actual create call structure is not restricted. You can design it close to what Linux offers, but modify it to handle locality.

5. C-like pseudo code for open (**10 points**)

This call associates the on-disk file structures with the in-memory tables. Also, it could be pre-loading data blocks from the file. However, pre-loading does not always results in improving the performance. For instance, pre-loading can bring in the first one or two blocks into the memory cache anticipating their access by the read() and write() operations. However, the read() and write() operations could be seeking to access different blocks. This makes the pre-loaded data useless.

Once the open call is successful, you get an index known as the file descriptor. This descriptor is useful in subsequent operations (e.g., read, etc).

6. C-like pseudo code for read (**15 points**)

This call reads disk data into a memory buffer. You can follow the Linux read() system call as an example. Depending on the number of bytes you put in the request, this call can access multiple blocks in the disk. When the read call is issued, the file control block (i.e., the i-node or the FAT table entries) are in memory. However, in some cases additional block accesses would be needed. For instance, with i-nodes you can have reads to far away locations (i.e., locations linked by

indirect blocks) creating additional disk accesses. The locations where these disk blocks are written would impact the read performance. That is, if the indirect blocks are written far away from the current disk head location, then time will be lost in moving the head to the location where the disk block is located.

7. C-like pseudo code for write (**15 points**)

This call writes a memory buffer to disk. You can follow the Linux write() system call as an example. Depending on the size of the buffer, this call can write multiple disk blocks. The disk block locations on the disk are important for locality. The locations on disk are mostly decided by the file system architecture you selected in a previous step of the design. While the number of blocks you write depends on the number of bytes you have in the buffer, in some cases, the offset in the file also becomes a factor. In particular, if you do a seek to a distant offset from the beginning of the file there is a question of whether you would fill up the intermediate bytes or not. In a dense file, there will be blanks written (effectively) when you seek to a distant offset. On the other hand, a sparse file would not write blank blocks. You need to keep the files sparse.

8. C-like pseudo code for close (**5 points**)

The call disassociates the file from the in-memory structures. Also, any blocks cached in the memory are flushed to the disk.

9. C-like pseudo code for seek (**5 points**)

The seek call operates with the in-memory data structures. It does not write or read any data from the disk. This call sets up state in the in-memory data structures so that subsequent calls to the disk are done appropriately.

10. Discussion of performance optimization (**10 points**).

Explain why your file system design is going to minimize the time required for application operations. What special features of the file system architecture are allowing you to get the best performance? In your explanation, show how pseudo-code's execution time can be affected by the disk actions.

Missing Requirements

This document does not spell out all the requirements. For instance, what should the create call do when a file is already there. You need to think through the file system actions and design the pseudo-code to handle the different situations.

The design document you present should have a MISSING REQUIREMENTS section that states the missing requirements you have considered. (**5 points**)