6.033 Design Project 1

Report

A Fast but Potentially Unreliable File System

Nada Amin

Introduction

This report presents a fast but potentially unreliable file system, which will be used by a webserver to allow clients to upload, download, and delete files.

The design takes advantage of two properties. First, the file system doesn't need to be reliable in face of crashes, which allows the use of memory for storage. Second, files are never modified once uploaded, so their layout on the disk may be considered fixed.

The design tries to minimize the number of non-consecutive blocks in the layout of a file on disk. When a file is laid out consecutively, the file system achieves excellent performance. For example, it reads a 1-megabyte file with 96.6% efficiency. However, the performance might decrease with time, because of disk fragmentation. To limit disk fragmentation, the file system separates small files from large files on disk.

File System Behavior

Request for Storing a File

A request for storing a file, if successful, copies the content of the file on the disk and adds entry in the files map. If enough space is not available on the disk, the request fails with a "disk full" error. If a file already exists with the chosen name, it fails with a "name used" error.

The content of the file is usually laid out consecutively, by using memory as temporary storage. If the system runs out of memory, it "flushes" the memory, transferring all the in-process writings

to disk, and resumes. If a large enough chunk of consecutive blocks is not available on disk, the system uses the largest chunk available and iterates. To limit fragmentation, small files are stored starting at the beginning of the disk, and larger files are stored starting at 25% of the disk.

Request for Retrieving a File

A request for retrieving a file copies the content of the file from the disk to the caller's memory. If the file name is not in the files map or if the file is still being written, the request fails with a "no such file" error.

Request for Deleting a File

A request for deleting a file, if successful, frees the disk blocks associated with the file, and removes its entry in the files map. If the file is being read, the request fails with a "file in use" error message. If it is being written, the request fails with a "no such file" error.

File System Interface

The file system implements the standard `read()`, `write()`, `open()` and `close()` interface.

File System Configuration

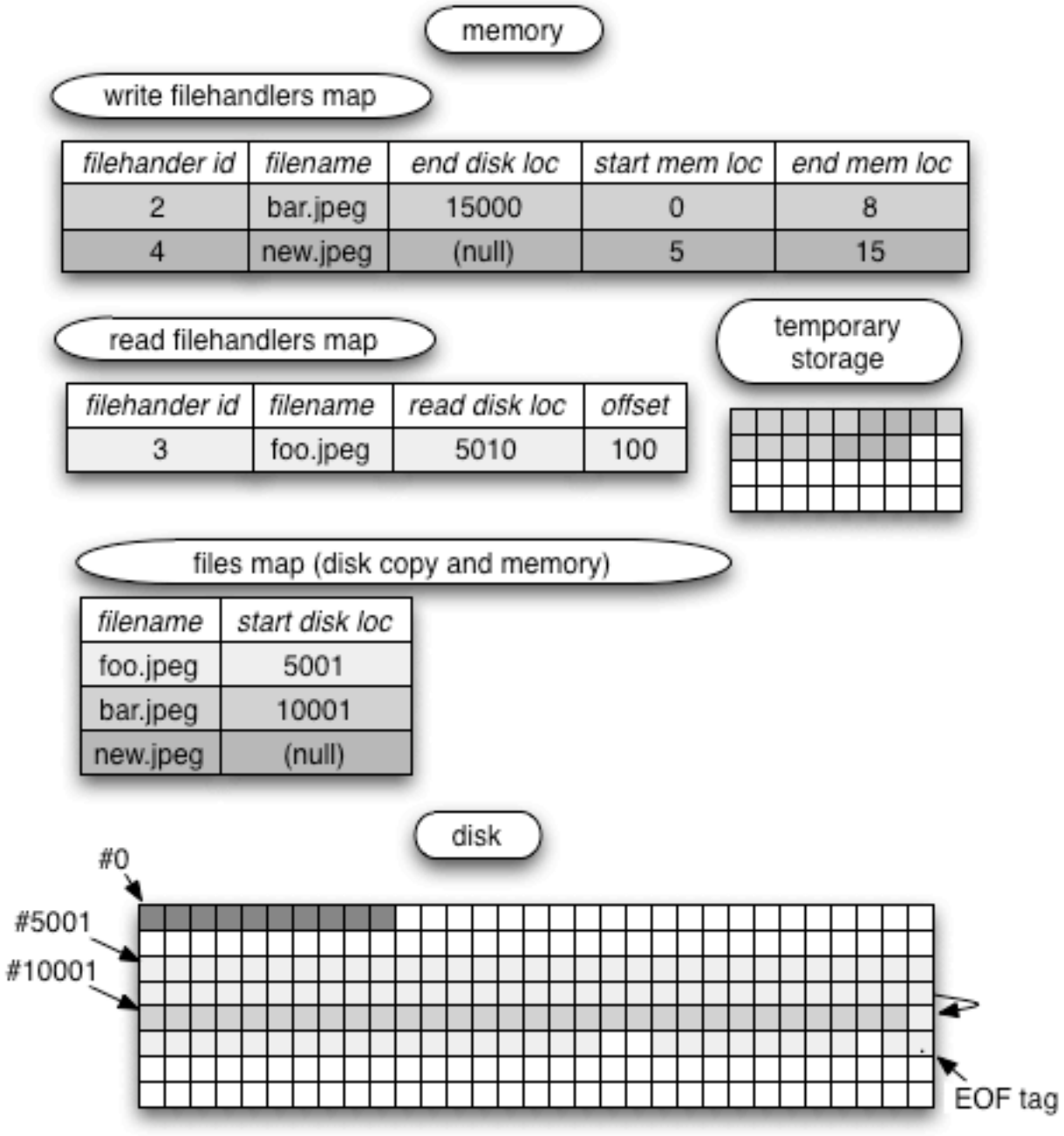

***Figure 1**: File System Configuration*

First, two files “foo.jpeg” and “bar.jpeg” are being written. When “foo.jpeg” fills about 40MB and “bar.jpeg” about 20MB, the system runs out of temporary storage in memory, so it “flushes” the content to disk. Then, “foo.jpeg” and “bar.jpeg” keep being written. Eventually, “foo.jpeg” is closed and re-opened for reading, and a new file “new.jpeg” starts to be written. The layout of “foo.jpeg” is broken into two contiguous pieces.

Figure 1 Explanation

The file system maintains three maps:

(1) the **files map** (**FM**),

(2) the **write files handler map** (**WM**),

(3) the **read files handler map** (**RM**).

Every block on disk used for storing file content has an end pointer to the next (usually consecutive) block for this file. Similarly, the content of a file in memory may be split into chunks, in which case each chunk has an end pointer to the next.

When a file is opened for writing, the system sets up an entry in the WF map with all storage locations null. When a file is being written, the system copies the chunk of content into memory. When it writes the first chunk, it sets the start memory location to point to the beginning of the chunk. When it writes a subsequent chunk, it updates the end pointer of the previous chunk to the beginning of the current chunk. Regardless, it updates the end memory location to the end pointer of the current chunk. When a file is closed, the system transfers the content to disk and removes the entry in the WM map. The end of the file is marked with a special EOF tag.

During the first disk transfer of a file (the end disk location is null), the system adds an entry in the FM map for the file. During a subsequent transfer, the system updates the end pointer of the block addressed by the end disk location to the first block just being written. Regardless, in the WM map, the system sets the end disk location to the end pointer of the last written block and the memory locations to null.

For performance, the FM map might be kept in memory, and the disk copy, stored in the first few blocks, only updated when the system shutdowns cleanly. When the machine crashes, the disk might be in an inconsistent state. Without the performance enhancement, only the temporary storage would be lost during a crash, and on reboot, the system would read the FM map to memory.

Memory and Disk Usage

The beginning of the disk is devoted to storing the files map (FM). Each file on disk has an entry in the FM map with its filename and the starting block of its content on disk. Initially, 10Mb of storage is allocated for the FM map, which allows roughly 800'000 file entries, assuming an average of 8 characters per filename. The FM map may use more space if needed, because, as usual, each block has an end pointer to the next block.

Apart from the FM map, the only other metadata are the end pointer of each block. Assuming an end pointer uses 28 bits, this metadata occupies less than 1% of the disk space, which is very reasonable.

In memory, the file system stores and maintains the write & read file handler maps (WM & RM). For performance reasons, it also keeps the FM map, and updates the disk copy only during a clean shutdown. The file system also uses memory for temporary storage. Ideally, the temporary storage would handle *max(number of simultaneous writes)*max(size of file),* in order to avoid flushes that would cause files to be laid out in non-contiguous chunks.

Performance Analysis

Assumptions

The throughput between disk and memory is 10 megabytes/second. A block being 4 kilobytes, the disk can read 2'560 adjacent blocks/seconds into memory. I am assuming that the time to read a block is (1/2560) seconds and the time to seek to a non-adjacent block and read it is (1/256) seconds, i.e. 10 times more.

Small File Sequential Workload

For the small file sequential workload (creating many small files, followed by reading the files in the order in which they were created), the file system stores each file on a block. Because the files are most likely stored adjacently, the file system only needs to seek to the first file, when reading them in order. The file system takes *(1/256 + (N-1)/2560)* seconds to read the files, where *N* is the number of files. For *N=247*, the reading time is *1/10* seconds, and, if each file is 1

kilobyte, the throughput is *2470* kilobytes/seconds. The throughput is only about *25%* of the maximum throughput, because the files are so small that *3/4* of each block is wasted.

Large File Random Workload

For the large file random workload (creating several large files, followed by reading the files in some unpredictable order), I am assuming the file system succeeds in storing each file on consecutive blocks. This assumption is reasonable, if the files are created sequentially and the disk is not significantly fragmented. To read the files in some unpredictable order, the file system takes *(1/256 + (m-1)/2560)*N* seconds, where *m* is the number of blocks per file and *N* is the number of files. For 1 megabyte files (*m=256*), the reading time is *3.3125* seconds for *N=32*, the throughput is *9.66* megabytes / seconds, or *96.6%* of the maximum throughput.

Now, I assume the disk is fragmented into chunks of *d* blocks. So a file of *m* blocks would be laid out in *m/d* chunks. The reading time would be *(m/d)(1/256) +(m – (m/d))(1/2560)) = (m/256) * ((1/d) + (1-(1/d))(1/10))*. The throughput would be *(10 d)/(10 + d -1)*. **Figure 2** shows a plot of *d* versus the throughput. For example, to achieve 80% efficiency, *d* needs only to be *36*. Since the file system separates files larger than 1 megabyte from smaller files, *d* is likely to remain reasonable.

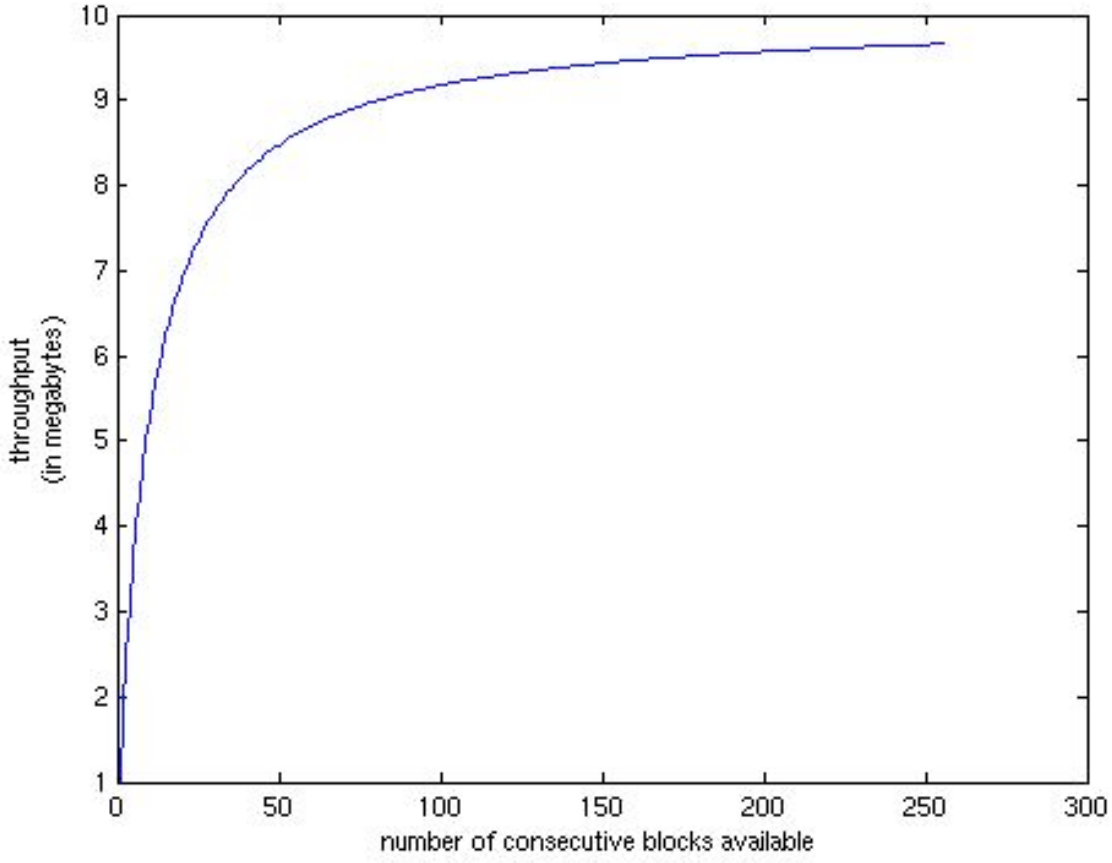

***Figure 2**:* d *versus throughput*

Large File Workload with Deletes

The performance for the large file workload with deletes (creating and deleting many large files followed by reading the remaining files in the order in which they were created), is similar to the performance for the preceding workload. The deletions might cause some disk fragmentation, which might degrade future performance.

Rationale

Assuming that accessing a non-adjacent block is an order of magnitude slower than accessing an adjacent block, the design does well to focus on limiting non-contiguous blocks in the layout of a file on disk.

My design attempts to achieve good performance without sacrificing simplicity. In fact, I discarded alternative solutions that would have brought too much complexity. For example, I considered keeping a file in memory after its upload completed. This way, the file system wouldn't need to access the disk during a request to read the file. In particular, the small file sequential workload would achieve excellent performance because the file system would never touch the disk. However, eventually, the memory storage would need to be flushed to disk. Now, flushing would be much more complicated, because some read file handlers might be pointing to memory cells to be transferred to disk. The file system might attempt to implement a more traditional form of caching, but an end-to-end argument suggests that the application could do it better.

The performance on the large file workloads is very good, but it relies on the assumption that the files can be laid out consecutively. This assumption becomes weakened after the file system has been running for a long time, because numerous creations and deletions might cause disk fragmentation. Nevertheless, as the analysis shows, by simply separating large files from smaller ones, the file system maintains a satisfying performance. If the performance really suffers from disk fragmentation, it might be useful to apply a de-fragmentation routine.

Conclusion

Despite its simple design, this file system performs well. On large files, the file system reads with high efficiency as long as the disk fragmentation is limited. To limit disk fragmentation, the file system stores small files and large files separately. Because this measure does not absolutely guarantee a low disk fragmentation, it is recommended to periodically run a de-fragmentation procedure, if performance becomes an issue. Reading smaller files is not as efficient, because disk block space might be wasted when files are significantly smaller than a block.

[Word count: 1832]