MEMFiS:

A Fast Memory-Backed File System

By Tamara Stern

6.033 Design Project 1
Professor Sam Madden TR10
TA Hongyi Hu
March 24, 2006

1. Introduction

Many file systems are designed to be fast, while maintaining reliability in the face of crashes. This paper presents an alternate design, MEMFiS: A Fast Memory-Backed File System, which optimizes only for speed and not for reliability. To maximize efficiency and simplicity, MEMFiS employs three unusual design decisions:

1. The File Table is stored in memory, which sacrifices a high maximum number of files for a reduced number of disk seeks per system call.
2. The disk is segmented into pre-sized blocks, which sacrifices optimal disk utilization for a non-fragmented disk with performance that degrades minimally over time.
3. The read-ahead caching scheme, which sacrifices throughput for an initial read() request for data availability in the cache for future requests.

MEMFiS is optimized for a file distribution similar to that of a typical desktop computer, as presented in Table 1.

File size (kilobytes)	% of Total Files Stored	% of Total Disk Space Used
0-1	25%	1%
1-10	45%	2%
10-100	25%	7%
100-1000	3%	20%
>1000	2%	70%

Table 1: The table demonstrates the file distribution of my laptop. For example, 25% of files in a typical file distribution are 0-1 kilobyte in size. However, these files only consume 1% of the total disk space. According to this data, the average file size is 200 kilobytes.

Because MEMFiS stores files in contiguous pre-sized chunks, the typical read() or write() operation will incur one disk access for every megabyte of data read or written. An open(), close(), or unlink() operation incur no disk accesses.

2. Design Description and Analysis

This section presents an overview of the design. Section 2.1 presents the file system interface, 2.2 presents the file system configuration, 2.3 presents the implementation details of system calls, and 2.4 presents a performance analysis of typical workloads.

2.1 File System Interface

Clients use a web server running on the machine to upload, download, and delete files. Multiple clients can read and write files simultaneously. To support these operations, MEMFiS provides an API that is based on the UNIX file system API [1], with the exception of the open() procedure. The open() procedure will require an additional input parameter *n*, the total size of the file, which is available to the web server during uploads and downloads through the http protocol [2]. This parameter allows the file system to allocate the appropriate blocks on disk to store the file. Because *n* is not needed for read operations, the client may pass a value of -1 for *n* when reading files.

2.2 File System Configuration

This section contains a description of the file system components as well as an analysis of how much memory and disk space will be devoted to the different parts of the system.

2.2.1 Hard Disk

Figure 1 presents the division of the hard disk into the four following block types, based on the assumed average workload presented in Table 1: 720,000 4-kilobyte blocks, 300,000 10-kilobyte blocks, 240,000 100-kilobyte blocks, and 90,000 1-megabyte blocks. The disk blocks are specified using 3 byte block numbers ranging from 1 to 1,350,000. The last 120 megabytes of disk are left empty for storage of the tables in memory, as well as other system parameters (e.g., the operation used to convert physical disk addresses to block numbers), during a clean system startup or shutdown.

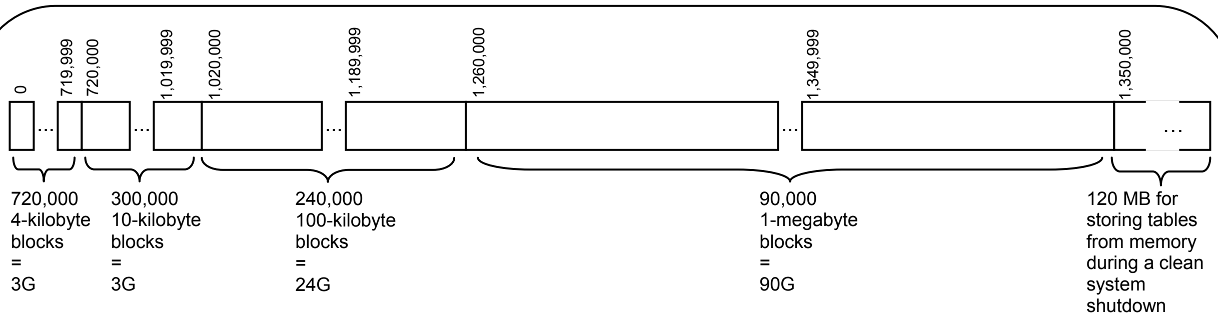

0
719,999
720,000
1,019,999
1,020,000
1,189,999
1,260,000
1,349,999
1,350,000
720,000 4-kilobyte blocks = 3G
300,000 10-kilobyte blocks = 3G
240,000 100-kilobyte blocks = 24G
90,000 1-megabyte blocks = 90G
120 MB for storing tables from memory during a clean system shutdown

Figure 1: The hard disk is split into blocks as follows: 750,000 4-kilobyte blocks, 300,000 10-kilobyte blocks, 240,000 100-kilobyte blocks, and 90,000 1-megabyte blocks. The blocks are specified using 3 byte block numbers ranging from 1 to 1,350,000. The last 120 megabytes of disk are left empty for storage of the tables in memory, as well as other system parameters (e.g., the operation used to convert physical disk addresses to block numbers), during a clean shutdown.

The disk will store files in one or more disk blocks (see Block-Choosing algorithm in Figure 7). Pointers to these blocks are stored in memory.

The hard disk will provide the following procedures to the file system:

- HD_READ takes a block number and offset, and reads the data stored at that location.
- HD_WRITE takes a block number, an offset, and data, and writes the data to the specified location.

2.2.2 Memory

The memory consists of the Free Space Bit Map, the File Table, the File Descriptor Map, the Current Reads/Writes Table, the Caching Table, and space reserved for the web server.

- *Free Space Bit Map (170 kilobytes)*: The map, as shown in Figure 2, keeps track of which disk blocks are currently storing data.

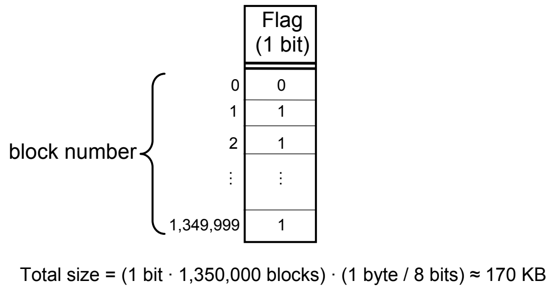
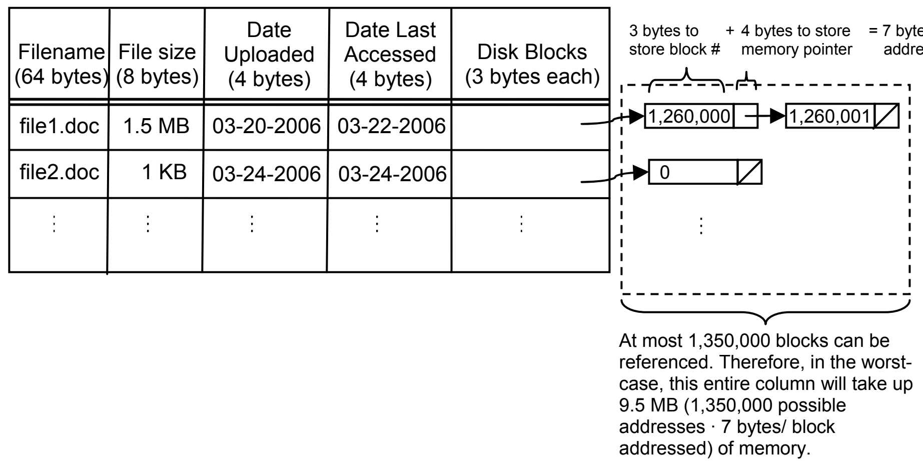

Flag (1 bit)
block number
0 0
1 1
2 1
⋮ ⋮
1,349,999 1
Total size = (1 bit · 1,350,000 blocks) · (1 byte / 8 bits) ≈ 170 KB

Figure 2: The Free Space Bit Map is stored in memory. Each allocation block is represented by a bit in the map: if the bit is set to 1, then the block is in use; if it is set to 0 then the block is free.

- *File Table (70 megabytes)*: Figure 3 presents the constant-time lookup table, which maintains each filename, file size, date uploaded, date last accessed, and the block blocks in which the file is stored.

  Keeping the File Table small enough to fit in memory imposes minor restrictions. First, the filename size limited to 64 bytes. However, other file systems support as little as 12 bytes per filename [3]. Additionally, the maximum number of files is limited to 750,000. This limit permits an average file size of 160 kilobytes (120GB total disk space / 750,000 files), which is less than the assumed average of 200 kilobytes, as presented in Table 1.



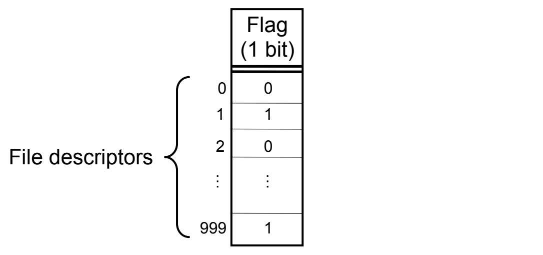
Filename (64 bytes)	File size (8 bytes)	Date Uploaded (4 bytes)	Date Last Accessed (4 bytes)	Disk Blocks (3 bytes each)
file1.doc	1.5 MB	03-20-2006	03-22-2006	
file2.doc	1 KB	03-24-2006	03-24-2006	
⋮	⋮	⋮	⋮	⋮

3 bytes to store block #
+ 4 bytes to store memory pointer
= 7 bytes/block addressed
1,260,000
1,260,001
0
At most 1,350,000 blocks can be referenced. Therefore, in the worst-case, this entire column will take up 9.5 MB (1,350,000 possible addresses · 7 bytes/ block addressed) of memory.

Total size = 78 bytes/entry · 750,000 entries + 9.5 MB for entire Disk Blocks column ≈ 70 MB

Figure 3: The File Table is stored in memory. It maintains a maximum of 750,000 files and the metadata associated with them.

- *File Descriptor Map (125 bytes)*: The bit map presented in Figure 4 maintains file descriptors that are currently in use. Unique file descriptors permit concurrent readers and writers to access the same file. MEMFiS limits the total number of concurrent reads and writes to 1000, which is on par with other systems [4].



Total size = (1 bit · 1,000 file descriptors) · (1 byte/8 bits) ≈ 125 bytes

Figure 4: The File Descriptor Map is stored in memory. It keeps track of the file descriptors that are currently in use; if the bit is set to 1, then the file descriptor is in use. If it is zero, then it is free to use.

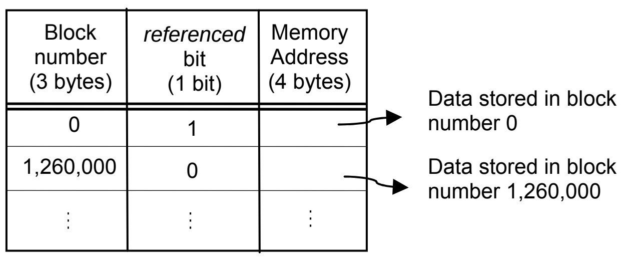
- *Current Reads/Writes Table (72 kilobytes)*: The constant-time lookup table maintains the files currently being read from or written to. As shown in Figure 5, the table is indexed by a file descriptor associated with a filename and the next block number (plus an offset) to access for subsequent read or write operations. There will be at most 1000 open files in the table, which is enforced by the limited number of file descriptors.

File Descriptor (2 bytes)	Filename (64 bytes)	Next Read/Write Block (3 bytes)	Next Read/Write Block Offset (3 bytes)
2	file1.doc	0	256
999	file2.doc	1,260,001	250,000
⋮	⋮	⋮	⋮

Total size = 72 bytes/entry · 1000 entries ≈ 72 KB

Figure 5: The Current Reads/Writes Table is kept in memory. It maintains the next block number to access for at most 1000 open files, which is enforced by the limited number of file descriptors.

- *Caching Table (5 megabytes)*: Figure 6 shows the table used to cache data that is brought into memory during read() requests. In this table, block numbers are mapped to the location in memory storing the data. Additionally, there is a *referenced* bit, used by the clock page-removal policy described in the 6.033 lecture notes [5], which is an approximation of the least recently used (LRU) removal policy.



Block number (3 bytes)	*referenced* bit (1 bit)	Memory Address (4 bytes)
0	1	
1,260,000	0	
⋮	⋮	⋮

Data stored in block number 0
Data stored in block number 1,260,000

Cache size = 5 megabytes (This value is configurable upon system installation)

Figure 6: The Caching Table is kept in memory. It keeps track of the block number mapped to the memory address of any cached data. Additionally there is a *referenced* bit that is used by the clock page-removal policy, as described in 6.033 lecture notes [5].

- *Web Server (45 megabytes)*: The remaining space is reserved for the web server.

2.3 Implementation of System Calls

This section describes the process for system calls to open(), read(), write(), close(), and unlink(), as well as the system behavior during a shutdown and crash recovery.

2.3.1 Implementation of *open(name, flags, mode, size)*

1. Check the File Descriptor Map.
   a. If there is no free file descriptor, return an error.
   b. Otherwise, choose a free file descriptor and update the File Descriptor Map.
2. Check the value of size.

   a. If size is equal to -1, a read() request will follow (see Section 2.1 File System Interface). In this case, if name is not in the File Table, return an error and skip to step 7.
   b. Otherwise, a write() request will follow. In this case, if name is already in the File Table, return an error. Otherwise, continue to the next step.
3. Check the Free Space Bit Map.
   a. If there is not enough free space on the disk, return an error.
   b. Otherwise, continue to the next step.
4. Add an entry to the File Table with the filename, size, and date uploaded. Assign disk blocks according to Block-Choosing Algorithm described in Figure 7.
5. Update the Free Space Map based on the blocks chosen.
6. Add an entry to the Current Reads/Writes Table with the file descriptor chosen, filename, and the first block in which to write the file.
7. Return the file descriptor.

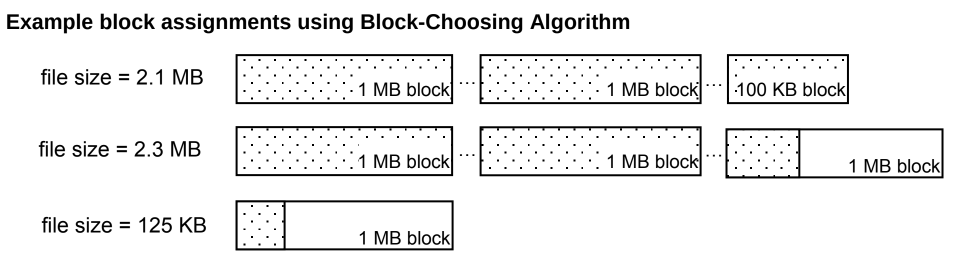


Figure 7: The Block-Choosing Algorithm has three goals. The first priority is to assign the fewest number of disk blocks to store the file. The second priority is to waste as little space as possible by assigning the appropriately-sized disk blocks. The third priority is to choose blocks that are contiguous, if possible. As a result, this algorithm does not maximize disk utilization. For example, a 125 kilobyte file will be stored in one 1-megabyte block instead of one 100-kilobyte block, two 10-kilobyte blocks, and 4-kilobyte block (wasting 899 kilobytes v. 1 kilobyte). However, this algorithm does optimize for speed, minimizing the number of disk seeks necessary for disk transfers (1 disk seek v. 4 disk seeks).

2.3.2 Implementation of *read(fd, buf, n)*

1. Check the Current Reads/Writes Table for next block to read.
2. Check the File Table to retrieve any subsequent blocks to read.
3. Check the Caching Table.
   a. If the data is in the table, extract the requested regions and set the *referenced* bit to 1.
   b. Otherwise, read the blocks from disk with a look-ahead policy that reads a minimum of one megabyte (this number is configurable) of data for each read request. The data read is stored in the Caching Table, using the clock page-removal policy [5] if necessary.
4. Update the Current Reads/Writes Table with the next read location.
5. Return the data requested.

2.3.3 Implementation of *write(fd, buf, n)*

1. Check the Current Reads/Writes Table for the next location to write to.
2. Check the File Table to retrieve additional blocks to write to.
3. Check if the block number being written to is currently stored in the Caching Table. If so, write the data to the block in memory specified by the Caching Table.
4. Write the data to the blocks on hard disk.
5. Update the Current Reads/Writes Table with the next location.

2.3.4 Implementation of *close(fd)*

1. Remove the file from the Current Reads/Writes Table.
2. Set the bit in File Descriptor Map to 0.

2.3.5 Implementation of *unlink(name)*

1. Check the Current Reads/Writes Table.
   a. If the file is currently being read or written, return an error. This behavior is similar that of the Windows file system [6].
   b. Otherwise, continue to next step.
2. Update the Free Space Bit Map to reflect free space.
3. Remove data from the Caching Table.
4. Remove file from File Table.

2.3.6 System Shutdown Behavior

When MEMFiS is signaled to shut down, it writes the tables in memory to the last 120 MB of the disk. Upon system startup, reload the data into the appropriate tables.

2.3.7 System Crash Recovery

When the machine crashes, all data is lost. When the machine reboots, all table entries are set to null and all bit map values are set to zero.

2.4 Performance Analysis

This section presents an analysis of the worst-case and best-case performance of MEMFiS on various workloads. The best and worst cases are differentiated by the following parameters:

1. The amount of free space on disk, which determines the degree to which files are stored contiguously on disk, which ultimately reduces the number of disk seeks per read() or write() call.
2. The presence of interleaved requests because, which interrupts potentially contiguous disk accesses with non-adjacent disk accesses to service other clients requests.

The calculations for the analyses are demonstrated in Figure 8, as described in 6.033 lecture notes [7]. The metrics analyzed include the time spent on disk transfers, time spent on disk seeks, average time elapsed between a client request and response reception (latency), the rate at which data can be transferred (throughput) and the achieved throughput compared to what the disk hardware allows (efficiency).

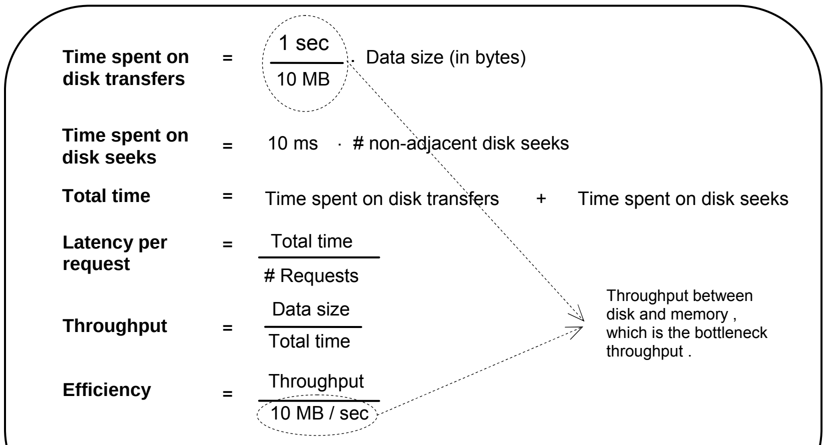

Time spent on disk transfers = 1 sec / 10 MB · Data size (in bytes)
Time spent on disk seeks = 10 ms · # non-adjacent disk seeks
Total time = Time spent on disk transfers + Time spent on disk seeks
Latency per request = Total time / # Requests
Throughput = Data size / Total time
Efficiency = Throughput / 10 MB / sec
Throughput between disk and memory, which is the bottleneck throughput.

Figure 8: Performance analysis metrics defines, as described in 6.033 lecture notes [7].

The time spent between the network and memory is not considered because it is negligible as compared with the disk transfer time (100 times faster), as long as the network does not overload the system. Additionally, the time spent on adjacent track seeks is negligible as compared with non-adjacent track seeks (10 times faster) so it is not considered in the analysis.

Finally, only read() and write() operations are considered in the analysis because open(), close(), and unlink() require no disk seeks. This performance surpasses that of the UNIX file system, which requires at least two disk seeks to perform open() or unlink() [8].

2.4.1 Small File Sequential Workload:

The analysis for the Small File Sequential Workload, which consists of creating many 1-kilobyte files followed by reading the files in the order in which they were created, is presented in Table 2.

	Time spent on disk transfers	Time spent on disk seeks	% Time spent seeking	Latency per request	Throughput	Efficiency
Worst write() case (small files written to a partially filled disk, with interleaved requests by other clients)	0.1 sec	10 sec	99%	0.0101 sec	99 KB/sec	1%
Best write() case (small files written to an empty disk, with no interleaved requests)	0.1 sec	0.01 sec	9.1%	0.00011 sec	9.1 MB/sec	91%
Worst read() case (small files read from disk, with interleaved requests by other clients)	100 sec	10 sec	10%	0.11 sec	9 KB/sec	0.09%
Best read() case (small files read from disk, with no interleaved requests)	0.1 sec	0.01 sec	9.1%	0.00011 sec	9.1 MB/sec	91%

Table 2: The Small File Workload Performance Analysis is performed for 1000 requests to read or write files that are 1 kilobyte in size.

This workload performs poorly for the worst read() case because the file system will read and cache 1 megabyte even if only 1 kilobyte is being requested. However, the cache optimizes performance in the average case, in which chunks of data are often read sequentially (Locality of Reference principle [9]). This optimization is particularly useful when there are interleaved requests that would cause the system to seek to a position away from the file being read. The cache is also useful for keeping popular files in memory, which, if kept in the cache, may be returned immediately upon request.

The disk utilization for this workload could potentially be as low as 0.625% (750,000 maximum # of files · 1 kilobyte/small file / 120 GB total disk storage). However, the expected workload has a greater file size distribution.

For a small file workload in general, MEMFiS will more likely have best-case performance for read() requests, because of the read-ahead caching scheme, and worst case performance for write() requests, because the system will not guarantee the usage of adjacent blocks when writing many small files (see Section 3 Conclusion and Future Work for suggestions for batching write() requests).

2.4.2 Large File Random Workload & Large File Workload with Deletes

The Large File Random Workload consists of creating several large files, followed by reading the files in some unpredictable order. The Large File Workload with Deletes consists of creating and deleting many large files followed by reading the remaining files in the order in which they were created. The analysis for these workloads, as presented in Table 3, are identical because delete() operations require no additional disk seeks and reading sequentially after any arbitrary number of delete() operations is analyzed in the same way as reading in an unpredictable order because there is no guarantee of the relative location of the remaining files.

	Time spent on disk transfers	Time spent on disk seeks	% Time spent seeking	Latency per request	Throughput	Efficiency
Worst write() case (large files written to the 4-kilobyte blocks, with interleaved requests by other clients)	0.1 sec	2.5 sec	96%	2.6 sec	385 KB/sec	3.9%
Best write() case (large files written to the 1-megabyte blocks, with no interleaved requests)	0.1 sec	0.01 sec	9.1%	0.11 sec	9.1 MB/sec	91%
Worst read() case (large files read from the 4-kilobyte blocks, with interleaved requests by other clients)	0.1 sec	2.5 sec	96%	2.6 sec	385 KB/sec	3.9%
Best read() case (large files read from 1-megabyte blocks, with no interleaved requests)	0.1 sec	0.01 sec	9.1%	0.11 sec	9.1 MB/sec	91%

Table 3: The Large File Random Workload & Large File Workload with Deletes Performance Analysis is performed for 1 request to read or write a file that is 1 megabyte in size.

The worst case for read() requests is not as poor as in the Small File Sequential Workload because the read() request is already 1 megabyte in size so MEMFiS will not read any additional data.

The worst case for write() requests could arise if 90,000 medium-sized (greater than 101 kilobyte) files are utilizing all of the 1-megabyte disk blocks. However, only 3% of expected files fall into this range (see Table 1). Nonetheless, MEMFiS optimizes performance in this situation by attempting to assign contiguous blocks when storing larger files.

For the large file workload in general, because of the expected file distribution, MEMFiS will more likely have best-case performance in which large files are stored in 1-megabyte blocks.

3. Conclusion and Future Work

MEMFiS is a file system optimized for speed rather than reliability. The key aspects of the MEMFiS design include maintenance of the File Table in memory and pre-sized disk blocks for file storage. These design decisions offer a solution that reduces the number of disk seeks significantly and avoids disk fragmentation, increasing overall efficiency.

MEMFiS is designed to perform optimally on a distributed workload with recurring requests for popular cached files, and frequent calls to open(), close(), and unlink(), which require no disk seeks.

MEMFiS does not optimize for highly interleaved workloads of very small, unordered read() and write() requests. However, the read-ahead scheme handles this situation by speculating which blocks will potentially be read and copying them to the cache.

There are four key areas in which future versions of MEMFiS could be optimized. First, buffering write() requests in memory would increase the performance when multiple clients are making interleaved write() requests to write to non-adjacent parts of the disk. Second, incorporating a queue of requests that could be batched and reordered would increase performance by reducing the overhead for each individual request. Third, future versions of MEMFiS could provide hierarchical directories for file organization. Finally, the system would benefit from increased security by maintaining and enforcing permissions. This improvement would also allow users to upload files that have the same name (e.g., index.html) because filenames could be tagged with user identification numbers.

4. References and Acknowledgements

4.1 References

[1] Saltzer, Jerome H. and Kaashoek, M. Frans. "Case study of UNIX file system layering and naming," Principles of Computer System Design, Appendix 2-A, pp. 2-52, 2006.
[2] "Hypertext Transfer Protocol," [Online document], 1999, [cited 2006 March 5]. Available HTTP: http://www.w3.org/Protocols/rfc2616/rfc2616.html.
[3] "Comparison of file systems," [Online document], 2006 March, [cited 2006 March 20]. Available HTTP: http://en.wikipedia.org/wiki/Comparison_of_file_systems.
[4] "Web Server Comparisons," [Online document], 2006 March, [cited 2006 March 21]. Available HTTP: http://www.acme.com/software/thttpd/benchmarks.html.
[5] Saltzer, Jerome H. and Kaashoek, M. Frans. "Chapter 6: Performance," Principles of Computer System Design, Section B-7, pp. 46, 2006.
[6] "File Utilities," [Online document], 2006 March, [cited 2006 March 21]. Available HTTP: http://developer.gimp.org/api/2.0/glib/glib-File-Utilities.html.
[7] Saltzer, Jerome H. and Kaashoek, M. Frans. "Chapter 6: Performance," Principles of Computer System Design, Section B-7, pp. 9-11, 2006.
[8] Ritchie, D.M, and Thompson, K. "The UNIX Time-Sharing System." BSTJ version of C.ACM Unix paper (1974): 6-7.
[9] "Operating Systems: Glossary of Common Terms," [Online document], 2006 March, [cited 2006 March 21]. Available HTTP: http://www.321site.com/greg/courses/os/glossary.htm.

4.2 Acknowledgements

I would like to thank Hongyi Hu, my 6.033 TA, for his tutorials and office hours, which were very helpful in clarifying the assignment. Michael Lin, my 6.033 tutor, spent significant time helping me to understand the assignment and drawing out possible file system configurations. I would like to thank Keith Winstein, another 6.033 TA, for his suggestion to eliminate disk fragmentation from my design. I would also like to thank Jane Connor and Robert Doherty, my 6.033 writing tutors, who helped me learn to write better papers, proposals, and projects. Finally, I would like to thank Professor Madden, my 6.033 recitation instructor, for his patience when answering so many questions during and outside of his office hours.

---

word count: 2,465
(not including figures, tables, title page, references, or acknowledgements)