

Content Addressable Storage Provider in Linux

Vesselin Dimitrov

Project Advisor: Jessen Havill
Department of Mathematics and Computer Science

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the work, and its date appear, and notice is given that copying is by permission of the author. To copy otherwise, or republish, to post on a server, or to redistribute to lists, requires prior specific permission of the author and/or fee. (Opinions expressed by the author do not necessarily reflect the official policy of Denison University.)

Copyright, Vesselin Dimitrov, 2003

1 Introduction

1.1 Operating Systems

An operating system is a computer program that acts as intermediary between a user of the computer and the computer hardware. The computer hardware provides the primitive computing resources such as memory and input/output (via keyboard, monitor, printer, mouse, etc.). Computer users run application programs such as word processor, computer games, web browsers and many others on top of the operating system. All of these application programs compete for the computer's resources (refer to Figure 1). The operating system manages the use of these various resources by the running user application programs. For example saving a file results in system call, which invokes the operating system. Special operating system routines called exception handlers are executed in order the file to be saved before control is returned to the user application. There are many different operating systems in use today, such as Windows, Mac OS, UNIX and Linux. They differ in structure and implementation, but they all perform the same function – to provide a convenient environment for user applications.

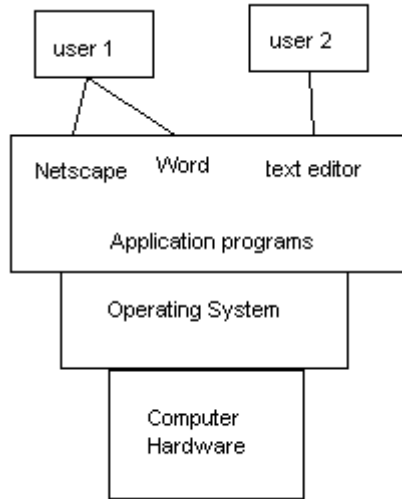


Figure 1. Abstract view of the components of a computer system

Computer programs are written in a high-level language. The high-level language program is then compiled into low-level machine language code that the computer can execute. Modern operating systems are also written in high-level languages.

Linux is an example of an open source operating system. Open source means that the high-level source is freely available to everyone to view and modify. This paper is concerned with kernel development for the Linux operating system. By kernel development we mean the addition of new features to the operating system by writing new source code and incorporating it within the already existing code of the operating system. Specifically we develop a new file system for Linux.

A file system is a key ingredient of an operating system such as Linux. Computers can store information on different devices such as magnetic disks,

optical disks and magnetic tapes. The operating system hides the difference between these different devices by providing a uniform view of storage devices. This is done by providing a logical storage unit – the file. The file system is responsible for mapping files onto physical devices. The user can retrieve a file by a unique user-defined name.

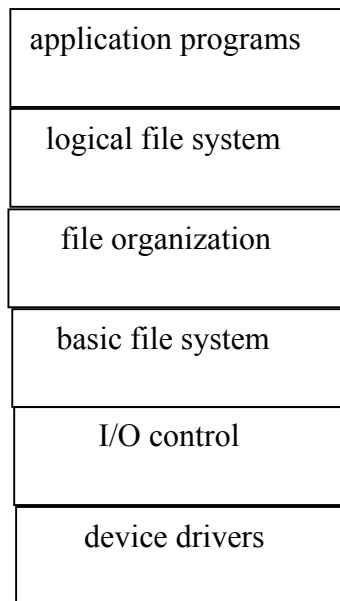


Figure 2. Layered file system

The file system itself consists of many different layers. Figure 2 shows an example of a layered file system design. Each layer uses features of lower layers to create features for use by higher layers. These features are referred to as interfaces and this kind of design architecture is called a layered architecture.

The lowest layer of a file system consists of device drivers. As we mentioned before, computers can store information on different physical devices. The main purpose of the device drivers is to hide the differences between different

devices so the layer above the device driver can use a uniform interface for completely different types of devices.

Device drivers can also be layered. This kind of device driver layering is called layered driver architecture. In layered driver architecture a device driver constitutes a layer above another device driver (Figure 3). The principle of layering device drivers is no different than the one used in layered file systems. Each layer uses the interface exported by the layer directly below it and exports an interface to the layer directly above it. The device at the lowest layer uses the interface exported by the physical device. It exports an interface to the driver on the layer above it.

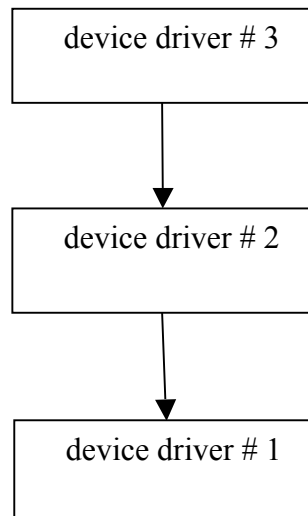


Figure 3. Layered driver architecture

1.2 The Internet

The Internet (Figure 4) is a huge collection of computers connected via telephone cables, radio waves, microwaves and infrared beams. The computers can use the links between them to send messages to each other. These messages

contain information that computers can understand and interpret. Each computer on the internet is called a host. From the user's point of view, a host can be either local or remote, depending upon whether the user is physically at the host running user applications on it, or using network messages to run user applications on the host at another location. The hosts in the Internet are also called peers, because there is no centralized control (server) through which the hosts communicate.

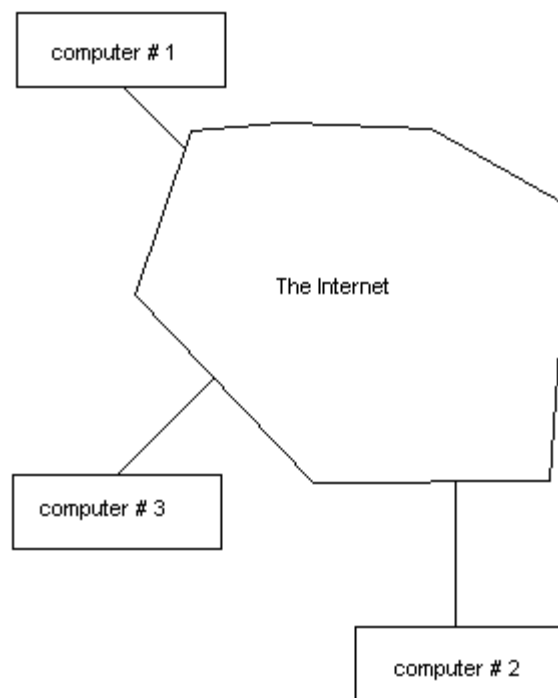


Figure 4. A schematic view of the Internet

The Internet facilitates communication between remote computers and sharing of resources over wide distances. Sharing files can be facilitated by a distributed file system (DFS). In a distributed file system, remote files and directories (collections of files) are visible from the local computer. Most

distributed file systems send files or parts of files over the network to a local machine that requests them. The remote files are kept on the disk of a special machine called a network file server. The local machine using the remote resources on the file server is called a client. File transfer protocol (ftp) and peer-to-peer systems are other methods by which hosts can access remote files.

Internet peer-to-peer systems (e.g., “Napster” and “Gnutella”) allow fast download of files from remote hosts (peers), using the resources of the Internet. Each peer is able to fetch files directly from other peers. Peer-to-peer systems do not use a central file server as a DFS does. In peer-to-peer systems each peer acts as both a client and a server at the same time. These systems have been controversial because they have been used by Internet users to acquire and distribute copyrighted materials such as songs, movies and books for free. In addition, peer-to-peer systems consume large amounts of network resources (bandwidth) because they greatly increase network traffic. However, the concept of peer-to-peer systems can be used to create other useful Internet applications, such as efficient WAN (Wide Area Network) file systems. In this paper, we discuss the implementation of such a system based on a peer-to-peer architecture that uses the concept of Content Addressable Storage (CAS) [26].

Section two discusses some of the background information needed in order to understand how the CAS facility works. Section three presents the CAS driver design. Section four discusses some implementation details of the driver. Section five consists of the performance analysis of the CAS driver.

2 Content Addressable Storage and Related Research

Content Addressable Storage (CAS) is a model of a storage facility where files are retrieved not by user-defined names but by their content. A fixed length digest computed over the content of the file replaces the traditional file name. There exist a couple of cryptographic hash algorithms that, given a variable data chunk, compute a small digest of that data. Examples of such algorithms are MD5 [20] and SHA1 [17]. The hash algorithm can be viewed as a pipe as shown on Figure 5. The file content is fed from one end of the pipe and the file digest is produced on the other end. Actually, that pipe is an abstraction of a computer program. The computer program reads the file and returns the file digest to the user. The digest of the file is also called a hash. Once the hash of the data has been computed it acts as a filename in the CAS facility. We can assume that the hashes for different files are unique since it has been proven that it is a statistically improbable for two different data chunks to have the same hash. It is also impossible to derive the content of the data block solely from its hash and data length.

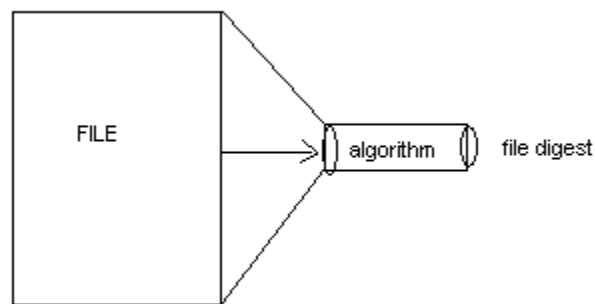


Figure 5. An abstract view of a hash algorithm producing a file digest

This paper focuses on a particular CAS provider that we developed and integrated as a driver in the Linux kernel using the concept of layered-driver architecture. We present the design and implementation of this CAS provider, as well as its performance characteristics. We implement a CAS provider software that responds to requests for files issued by a CAS client (Figure 6). A CAS provider is just a piece of software that functions as a basic file system and exports a uniform CAS interface through which it is given commands. A CAS provider could even be a traditional file system with filenames replaced by the file content digests. CAS providers could also be remote. If that is the case then a set of remote procedure calls (RPC) and a communication protocol should be devised to enable network communication in the form of requests and replies between the CAS client and the CAS provider.

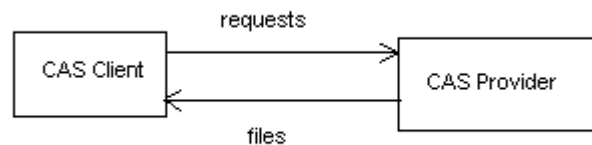


Figure 6. Communication between a CAS client and a CAS provider

Inside a computer system, the CAS facility is layered like a file system or device driver, in the architecture discussed earlier. In Figure 7 a CAS client sits above a CAS library, and the library sits above one or more CAS providers. This library enables the client to communicate with different CAS providers through a uniform library interface. A channel is an abstract connection between a particular

client and provider. There is one-to-one mapping between channels and providers. Thus, in order to use the services of a particular provider, the client has to open a channel associated with that provider. That channel can be closed at any time by either the client or provider, allowing for mobile providers. In other words, providers can arrive and depart at any time. Special library routines exist for determining what providers are available and advertising them to the client.

We wrote a CAS device driver that plays the role of one of the CAS providers at the lowest layer of the system (Figure 7). This system can be used not only as a peer-to-peer system but also for single instance distributed storage [9, 18]. In single instance storage multiple files with the same content are not stored multiple times on the disk. Their content is stored on the disk only once and all the files share the same storage location. Another possible application of the CAS system is reliable backup of files [11]. It also can be incorporated into a DFS (Distributed File System) to increase efficiency over a WAN such as the Internet.

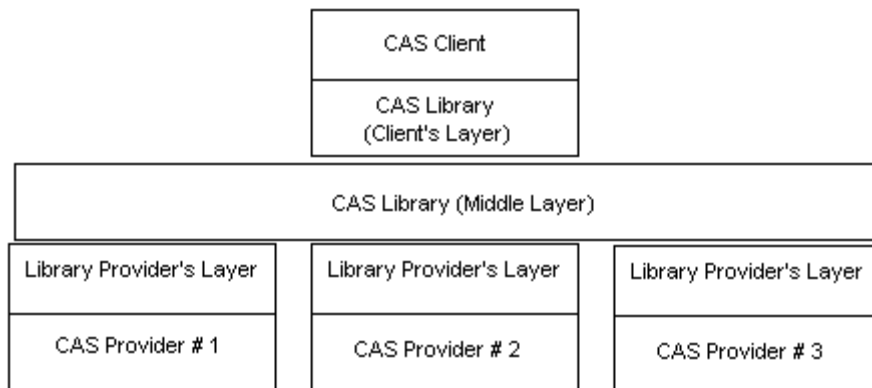


Figure 7. A view of the entire CAS system

CASPER [26] significantly improves the performance of a traditional distributed file system (DFS) when used over a high-delay network. Two key characteristics of CAS responsible for its superior performance are the use of file recipes and the closeby CAS providers that can be either local or remote. A recipe is just a list of digests of file blocks. Retrieving each of the blocks will retrieve the whole file. There is a recipe corresponding to each file stored on the file server. If there is a low-bandwidth, high-delay network separating the client and server in a DFS, it is better if the client queries the file server only for the recipe of the file instead of the entire file content. Since the recipe is a relatively small amount of information it will not add additional strain on the network bandwidth. Once the client has the recipe it can query its local and/or remote CAS providers for the blocks of the file. Ideally, the client will be able to retrieve some of the blocks, if not the whole file, from closeby providers rather than the remote file server. Clearly, this will be faster than retrieving the entire file from the remote file server over the high-latency network between the client and the server. Furthermore, in the former scenario the client can fetch different blocks from different providers concurrently, which will also improve performance, since concurrency will increase throughput. If the providers that the client queries do not have all the blocks, then the client will have to fetch the blocks it does not have from the file

server. However, this scheme will still yield better performance than fetching the whole file content from the server.

A lot of research efforts are currently directed toward this relatively new idea of creating models of CAS, measuring their performance, and comparing it with the performance of traditional storage models. Currently, the results derived by researching distributed hash tables (DHTs) [10, 14, 19, 21, 22, 27] and specialized file systems on top of DHTs [12, 13, 16, 26] show that CAS will probably become an invaluable storage facility and thus an indelible part of any computing system. In systems based on distributed hash tables (DHTs), each file is associated with a key (produced, for instance, by hashing the file) and each host in the system is responsible for storing a certain range of keys. The basic operation in a DHT is `lookup(key)`. This operation returns the identifier of the host that has the file associated with the key. Single instance storage systems [9] use hashes to avoid storing multiple file twice. The content addressable network [19] uses DHTs to provide a scalable distributed hash table lookup system. Chord [22] is a DHT system with a similar functionality. The file systems built on top of DHTs include CFS [12], PAST [13], Ivy [16]. CASPER [26] is a file system that exploits the DHT functionality combined with CAS architecture.

3 Detailed Technical Design

The CAS provider that we developed is actually designed and implemented as character device driver [1, 4, 5] for a virtual CAS device in Linux. The CAS driver constitutes a layer directly above the disk driver layer (Figure 8). The CAS

driver uses the interface exported by the disk driver to read and write to the physical disk. The CAS driver itself exports an interface to the layer directly above it. This type of design is an example of layered driver architecture discussed earlier. Each CAS device has an associated disk partition (virtual disk) mapped to it. This partition is raw (not logically formatted) and the CAS driver invokes the disk driver to read and write to that partition. The CAS driver supports services such as free-space management and file allocation that are typically part of a file system. Since the CAS provider does not need to implement all the functionality modern file systems have, we decided that a character driver was the best design choice for the CAS provider.

For our purposes a file is defined as a variable-size data block. It is important to distinguish between a file at the CAS driver level and at the level above it. A high-level file may consist of many CAS-level files. In other words, the granularities of the two layers differ. As discussed previously, a file can be retrieved by its recipe and is a collection of variable size data blocks, each with its own content digest. When a file has to be written to disk its data blocks are fed one by one to the CAS driver. The CAS driver interprets each of those blocks as a separate file, since the driver stores and retrieves blocks by their digests. In the following subsections, we discuss the memory and disk data structures the CAS driver will use, how to lay those disk data structures on the disk and the interface the driver exports.

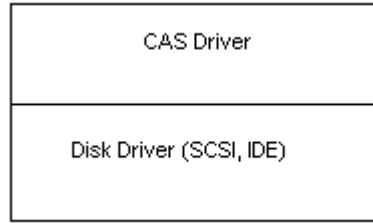


Figure 8. The layered-driver architecture

3.1 Hash Algorithms

As discussed earlier, there exist two hash algorithms for computing the file digest. The SHA1 algorithm computes a 20-byte hash. The MD5 algorithm computes a 16-byte hash. Since the design of the CAS driver strives for maximum flexibility both algorithms can be used to hash a file. The level above the driver decides which algorithm should be used. In addition, if a new cryptographic hash algorithm becomes available it will be relatively easy to modify the current implementation of the driver to support the new algorithm.

3.2 Hash-FCB map

The CAS driver, similarly to a file system, associates a file control block (FCB) with each file on the disk. The file control block contains necessary and sufficient information to retrieve the file content from the disk. Since each hash (digest) corresponds to a file, the driver needs to keep a hash-FCB pair for each

file. Assuming that each FCB is stored on a separate disk sector, one of the main issues in designing the CAS driver is how to create an efficient and flexible map from hashes to physical blocks. We decided to call this map a hash-FCB map. Each entry in the map is 64 bytes long (Figure 9). A 64-byte field is convenient because then the disk sector size, which is 512 bytes, is multiple of the map size. The first 32 bytes are used to store the hash. The current cryptographic hash standards, MD5 and SHA1, have 16 and 20-byte hashes respectively. We use 32 bytes to store a hash to support variable size hashes and gain the flexibility to support other hash standards in the future. The next 8 bytes in the hash-FCB entry hold the pointer to the FCB associated with the file. Using 64-bit pointers allows the implementation to account for the 64-bit disk addressing schemes that are coming into being due to the rapidly increasing size of secondary storage devices. Since current versions of the Linux kernel use 32-bit disk-sector pointers the entire 64-bit field that we have reserved for the FCB pointer is not currently utilized. The next 8 bytes of the hash-FCB pair contain the size of the block in bytes. The last two 8-byte fields in a hash-FCB entry can be used to implement an on-disk index for the hash-FCB map file. (Since the hash-FCB will be stored on disk it can be viewed as a file itself.) For the time being these fields are not used, and the hash-FCB map will be assumed to be small enough that caching it in sorted order in memory is feasible. Given the current trend in disk sizes and file sizes, this assumption is reasonable.

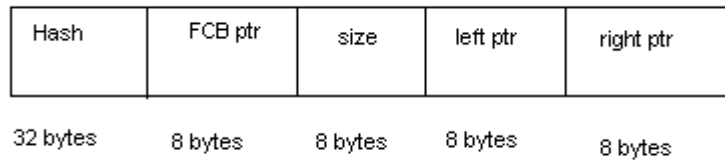


Figure 9. An entry in the hash-FCB map

The CAS driver is a module in the operating system. A module is a component of the operating system with a separate logical function. To use the functionality of a module, that module has to be loaded first. Linux allows modules to be loaded at any time (dynamic loading). When the CAS driver module is loaded, the hash-FCB map is cached in memory and an in-memory index is built to speed up searches in the hash-FCB map. This index is implemented by a red-black tree [29]. In order to increase reliability the hash-FCB map cache content is periodically written back to disk in case it has changed (new files have been inserted or old files have been deleted). Since the hash-FCB map can be interpreted as a file itself, it has an associated FCB. Sectors 1-15 on the disk are reserved for these hash-FCB map FCBs (Figure 10). This design allows maximal flexibility since it allows the driver to support up to 15 different cryptographic hash algorithms. Since currently only two cryptographic hash algorithms are available, only sectors 1 and 2 are utilized. In order to increase reliability, the FCB for each hash-FCB map is duplicated at the end of the disk.

Sector #	
0	Free-List FCB
1	MD5 hash-FCB map FCB
2	SHA1 hash-FCB map FCB
3	Reserved for a hash-FCB map FCB
4	Reserved for a hash-FCB map FCB
.	.
.	.
.	.
16	The beginning of the available data sectors
	.
	.
	The end of the available sectors
	Reserved for a hash-FCB map FCB
	.
	.
	Reserved for a hash-FCB map FCB
	SHA1 hash-FCB map FCB
	MD5 hash-FCB map FCB
	Free-List FCB

Figure 10. A logical view of the disk

3.3 File Allocation Methods

To make disk I/O operations as fast as possible, the data sectors of a file should be contiguously allocated on disk. A classic contiguous allocation scheme will inevitably result in external fragmentation, and so a different type of approach is required. We decided to use a direct indexing allocation method [28]. However,

the sectors associated with a file should be kept contiguous whenever possible. In the current implementation, the FCB of a file contains 10 64-bit direct pointers to data blocks (Figure 11). Since the file size does not change dynamically, all blocks needed for a particular file are allocated in advance. A memory cache of FCBs can be maintained in order to speed-up I/O operations. There is a length field associated with each sector pointer. The length field indicates how many contiguous sectors after the first sector (pointed by the direct pointer) have been allocated to this file. The FCB of the file also contains the size of the file in bytes.

Each FCB is stored in a disk sector by itself. Since a FCB is smaller than a disk sector, some disk space, insignificant amount compared to the overall disk size, is wasted.

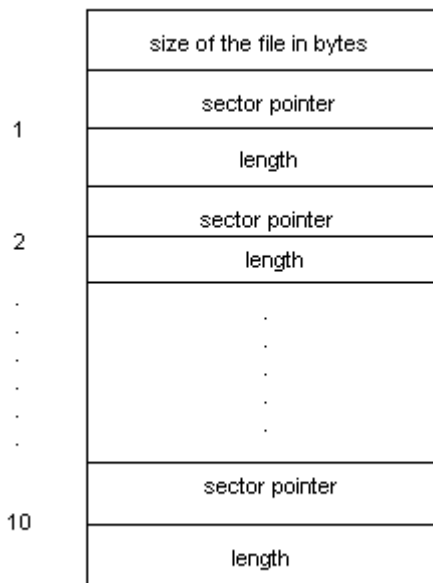


Figure 11. A logical view of a file control block

3.4 Free-Space Management

There are four commonly used methods to keep track of free space on a disk [28]: bit vector, linked list, grouping, and counting. In order for a bit vector scheme to be time efficient, the entire bit vector must be kept in memory. Since the current technology trend is towards larger disks, this is becoming impossible, even with clustering. Linked list and grouping schemes are also not appropriate as the CAS driver must keep track of contiguous sectors. Thus, the only choice left is counting, which maintains a list of contiguous free blocks. These blocks can have variable size and each entry in the list contains the address of the first sector and the number of free contiguous sectors. As blocks are freed, adjacent blocks are combined into a single block. We call this list a free-space list or a free list. The free-space list must be stored on disk. The FCB associated with the free-space list on disk is stored in both the first and last sectors on the disk to increase reliability (Figure 10). Like the hash-FCB map, the free-space list is cached in memory when the module is loaded and periodically written back to the disk if changed.

3.5 Exported Interface

The CAS driver exports four logical operations to the layer above it: CREATE, DELETE, QUERY and FETCH.

CREATE creates a new CAS file on the disk. CREATE is parameterized by the data to write. It computes the hash for that file, and checks the hash-FCB map for that hash. If the hash is not in the hash-FCB map, then a new hash-FCB map entry for that file is created and inserted in the hash-FCB map, and the file is stored on the disk. The hash for that file is returned to the user. If the file is already on the disk (the hash for that file was found in the hash-FCB map) then CREATE just returns file's hash.

DELETE is parameterized by a hash and deletes the file that corresponds to that hash. The disk space occupied by the file is inserted back into the free list. If the file specified by the hash does not exist, then DELETE returns without doing anything. The current version of the CAS driver does not implement the DELETE operation.

QUERY is parameterized by a hash and returns a boolean value indicating whether the hash is in the hash-FCB map. The purpose of QUERY is to check quickly if the CAS provider has a particular file or not. Sometimes a special flag will be passed to QUERY. That flag indicates that the QUERY will be followed by a FETCH if the return value of the QUERY operation indicates that the provider has the file specified by the hash. Since a future FETCH is expected, the QUERY operation may commence the process of copying the file from disk to memory. Thus, when the FETCH is invoked, a large part of the file will likely be already in memory, increasing the efficiency of FETCH.

FETCH is parameterized by a hash and returns the file associated with that hash, if it exists. FETCH is very similar to QUERY. The only difference is that FETCH will read the file from the disk if the file is available.

Both the FETCH and QUERY operations accept a value indicating the length of the file in addition to a hash. Unless the user sets this length to a special value, the CAS driver assumes that the desired file has to be precisely the same size as the length value. If the hashes match but the lengths of the files do not match, then both FETCH and QUERY return an error indicating that the CAS provider does not have the requested file.

Since, from the user's point of view, a file is a collection of variable size CAS data blocks, MULTI_FETCH and MULTI_QUERY operations may improve efficiency. These two operations are the same as FETCH and QUERY except that they are parameterized by an array of hashes and lengths, and return an array of blocks (in the case of MULTI_FETCH) or block-availability indicators (in the case of MULTI_QUERY). When a client needs to fetch a whole file from one hash provider (in a situation where that is the only available provider that has the file) the client can pass the array of file block hashes and their respective data lengths to MULTI_FETCH and/or MULTI_QUERY instead of invoking a FETCH and/or QUERY operation for every file block (and thus incurring the overhead of invoking a system call in the operating system for each single operation).

3.6 The need to logically format the partition

Similar to a file system, the CAS driver sits on top of a raw disk driver. Therefore, the CAS driver is responsible for file allocation and free disk-space management, and needs to lay its own data structures on the disk to facilitate these functions before it can be used for the first time. This is called a logical format of a disk partition. The logical formatting consists of creating empty hash-FCB maps for each hash algorithm the driver supports, creating a free-space list indicating that currently there are no files written to the disk, and writing those data structures to the logical partition. The utility that logically formats the partition is called a format utility. Once the formatting is done, the CAS driver can be safely loaded into memory and used.

When the CAS driver is loaded, it reads the hash-FCB maps and the free-space list from the disk. If the latter two data structures have never been written to the disk (the disk has not been formatted) things can go awry. Therefore, before using the CAS driver for the first time, the system administrator must invoke the CAS format utility on the disk partition that is to be used by the CAS driver. If the system administrator makes a mistake by giving the CAS format utility a partition that has already been formatted and in use by some other file system, all the information stored on that disk partition will be lost. This situation can be avoided by enhancing the CAS driver format utility to check if a file system already exists on the partition and, if so, displaying an alert message asking whether to continue.

4 Implementation of the CAS Driver

The CAS driver is written in the C programming language and follows the standards for dynamic kernel drivers [1]. In the Linux kernel [2, 3, 6, 7] one can load and unload modules, which are pieces of kernel code, dynamically. This feature allows kernel code that is not being used to stay on the disk. The kernel code loaded in memory is then smaller than the actual kernel, which leaves more memory for user programs. The system administrator is responsible for loading and unloading modules. Dynamic loading of the CAS driver works well because the CAS providers are written to be “mobile”. When the CAS driver is loaded it becomes one of the available CAS providers. The driver can be unloaded at any time by the system administrator and this removal does not leave the system in an inconsistent state.

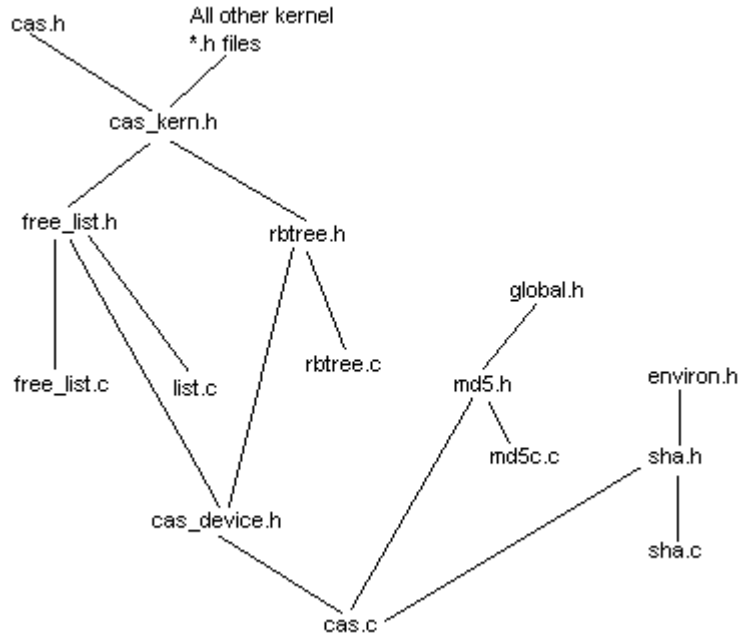


Figure 12. The file hierarchy that constitutes the CAS driver implementation

The CAS driver itself consists of multiple source (.c) and header (.h) files. Header files contain data structures definitions while source files contain the functions that manipulate those data structures. Figure 12 shows the dependencies between the files that constitute the CAS driver. The main part of the driver containing the system call implementations and the load/unload routines is contained in the file cas.c. This is why cas.c is located at the lowest layer of the file hierarchy shown in Figure 12. At the highest level of the hierarchy is cas.h. This file must be included by the layer above the CAS driver, which in our system is the CAS library. The cas.h file contains structures used by the CAS library to communicate with the CAS driver. The cas.h file, as well as the kernel *.h files are included from cas_kern.h, which contains structures used only by the files that

constitute the CAS driver. The `cas_kern.h` file is included from both `free_list.h` and `rbtree.h`. The `free_list.h` file contains structure definitions used by the free-space list module. Both `list.c` and `free_list.c` include `free_list.h`. All of the important list-manipulation routines are in `list.c`. The `free_list.c` file contains list routines used mainly for debugging. The `rbtree.h` and `rbtree.c` files contain the red-black tree implementation used to implement the hash-FCB maps. The red-black component is an example of a full experience component. This means that the better part of the code was already written for us and we just had to do slight modifications to adjust the component to fit our needs. We used acquired the code for the red-black tree component from OOPWeb.com [25]. The components implementing the MD5 and SHA1 cryptographic hash algorithms are also full experience components. The MD5 component consists of the following files: `global.h`, `md5.h` and `md5c.c`. `Environ.h`, `sha.h` and `sha.c` constitute the SHA1 component. We acquired the source code for these algorithms from “John Halleck's SHA implemented in C” [23] and “MD5 Homepage” [24]. If the reader is interested in the specifics of these algorithms, we recommend sources “Secure Hash Standard (SHS)” [17] and “The MD5 Message-Digest Algorithm”, RFC 1321 [20]. The `cas_device.h` file includes `free_list.h` and `rbtree.h` and is directly included by `cas.c`. The `cas_device.h` file contains the data structure that is used to implement the abstraction of a CAS device. The contents of all these files are included in Appendix A.

The CAS driver can be used by multiple processes concurrently. In order to ensure consistency of the kernel data structures used by the driver, we implement a

simple locking mechanism. Processes that invoke CREATE and FETCH are enqueued on a common process wait queue and only one processes is dequeued at a time. When that process has completed its operation the next process in the queue is dequeued and so on until the queue is empty. This ensures that only one process is in the middle of a CREATE or FETCH operation at any time. Processes that invoke QUERY are not queued on this wait queue and thus multiple processes can be performing a QUERY operation concurrently. To ensure consistency of the hash-FCB maps a kernel lock is implemented for every map. Before any operation on a hash-FCB map the process trying to perform that operation has to acquire the lock corresponding to that map. After the operation is complete the process has to release the lock so other process can acquire it.

5 Performance

In this section we present some performance results derived by testing the CAS driver. To test the driver we created a user program that reads all the files in a directory and invokes either a CREATE or FETCH operation on each one of the files. We created another user program that generates binary files with a mean size according to a normal distribution. The input to this program is n – the number of files to generate, and the mean size of the files. The variance of the normal distribution is set to be 10% of the mean. Using this program we created 3 directories with test files: one containing 300 files with mean size 2 KB, another

containing 100 files with mean size 100 KB, and third containing 20 files with mean size 4000 KB (4 MB). For each directory we applied 5 different test conditions: condition 1 invokes CREATE on non-existent file, condition 2 invokes CREATE on already created files with a cold cache, condition 3 is the same as condition 2 except the cache is warm, condition 4 invokes FETCH on existent files with a cold cache, condition 5 is the same as condition 4 except that the cache is warm. We expect the three major bottlenecks to be disk I/O, searches through the hash-FCB maps and to some extent computing the file digest in CREATE. In condition 1 both of these bottlenecks exist. In conditions 2 and 3 the bottleneck is only incurred by searches through the hash-FCB and the digest computation. Conditions 4 and 5 incur the cost of disk read and searches through the hash-FCB map.

As expected for all file sizes the driver shows the worst performance under test condition 1. It is clear from Figures 13-18 that no matter what the file sizes are the warm cache conditions give significantly better performance than the cold cache conditions. Figures 19 – 22 show that the size of the file is the main influence on the performance of the CAS driver. However, if we increase the file size by a factor of 50 the performance decreases only by a factor in the range 5-20. Therefore, the file size and performance variables are not proportional.

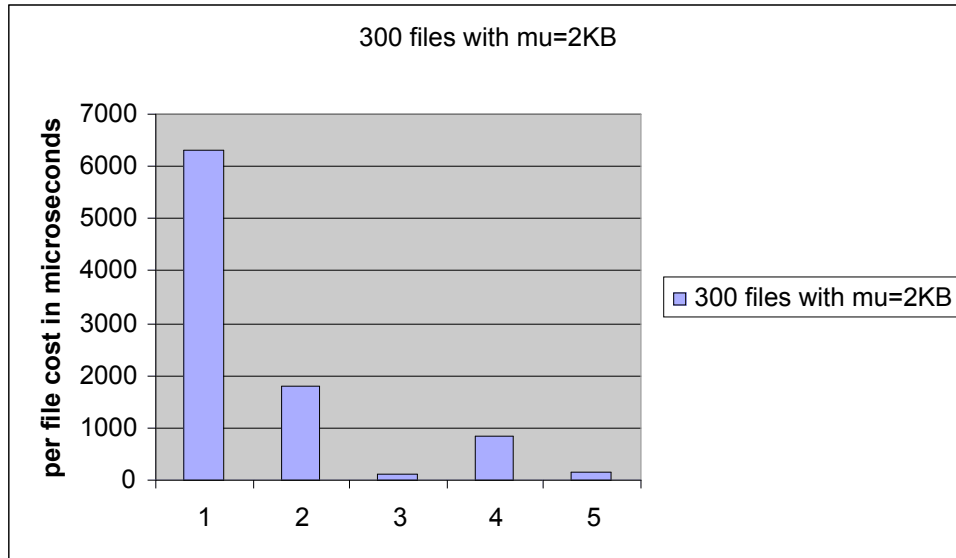


Figure 13. The cost per file for the 2 KB files under the five different conditions

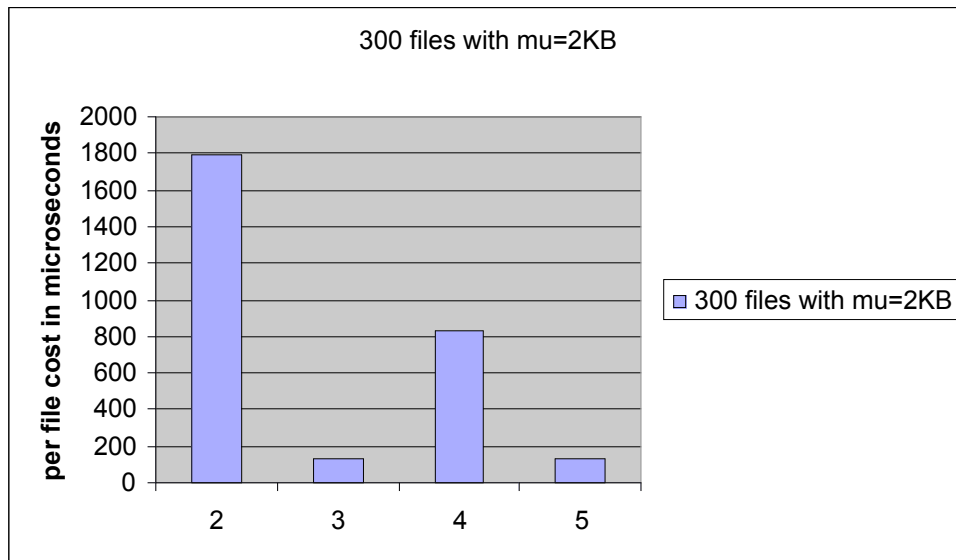


Figure 14. The cost per file for the 2 KB files under conditions 2-5

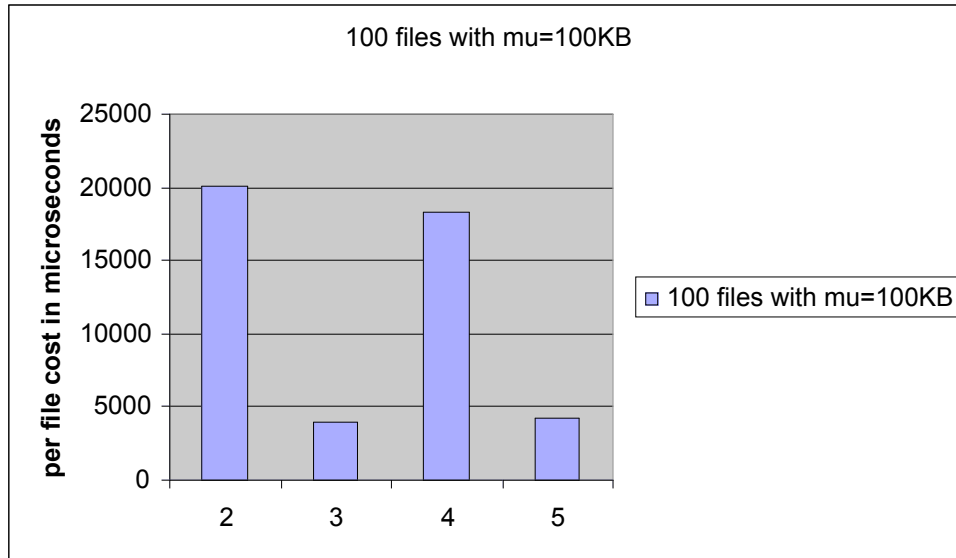


Figure 15. The cost per file for the 100 KB files under conditions 2-5

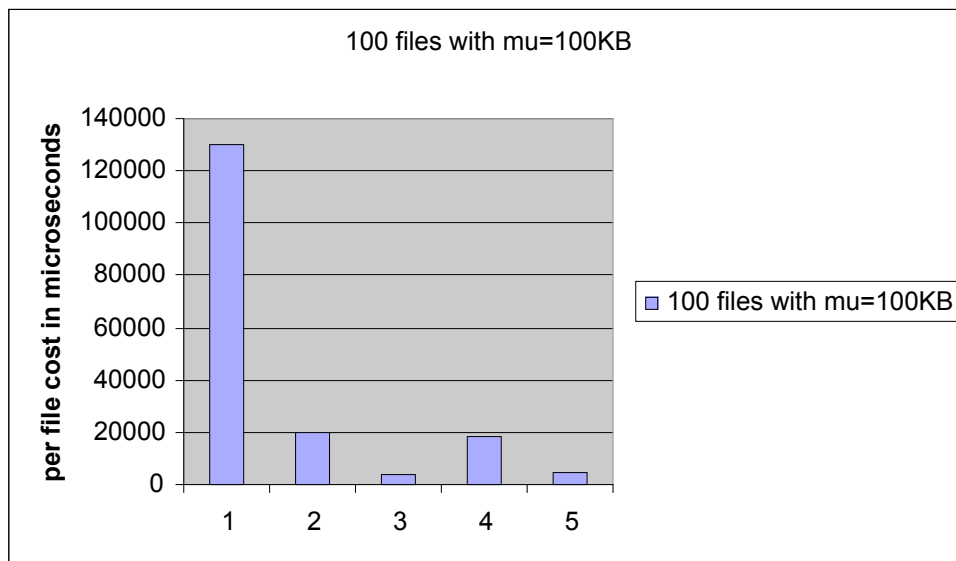


Figure 16. The cost per file for the 100 KB files under the five different conditions

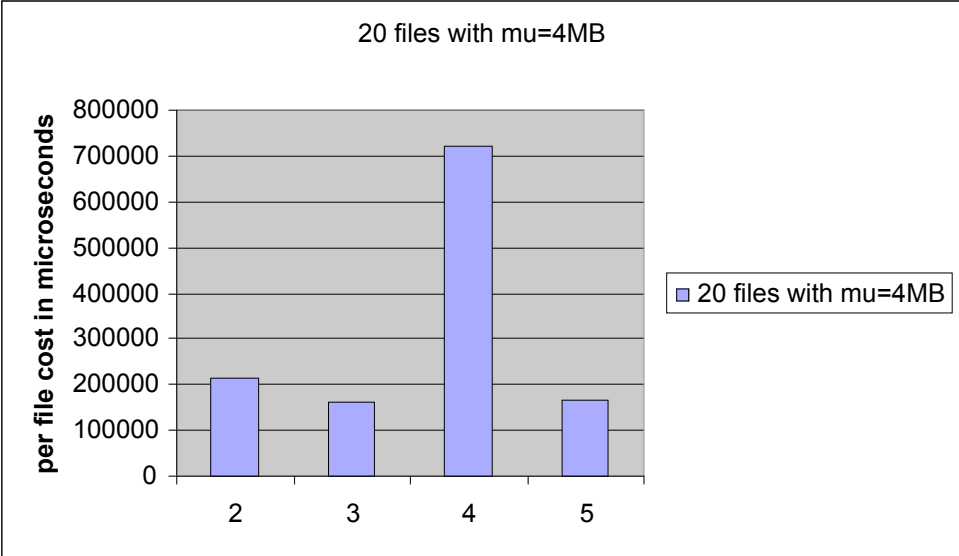


Figure 17. The cost per file for the 4 MB files under conditions 2-5

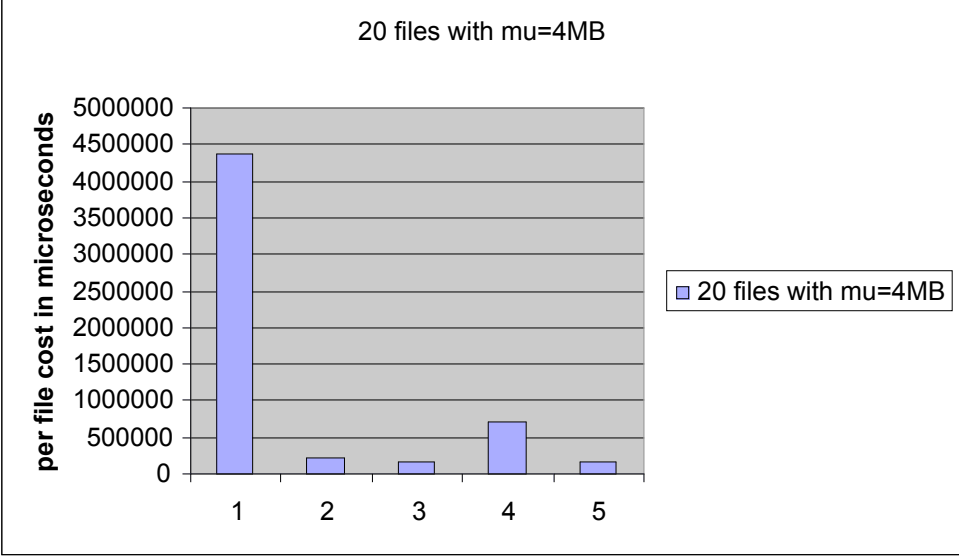


Figure 18. The cost per file for the 4 MB files under the five different conditions

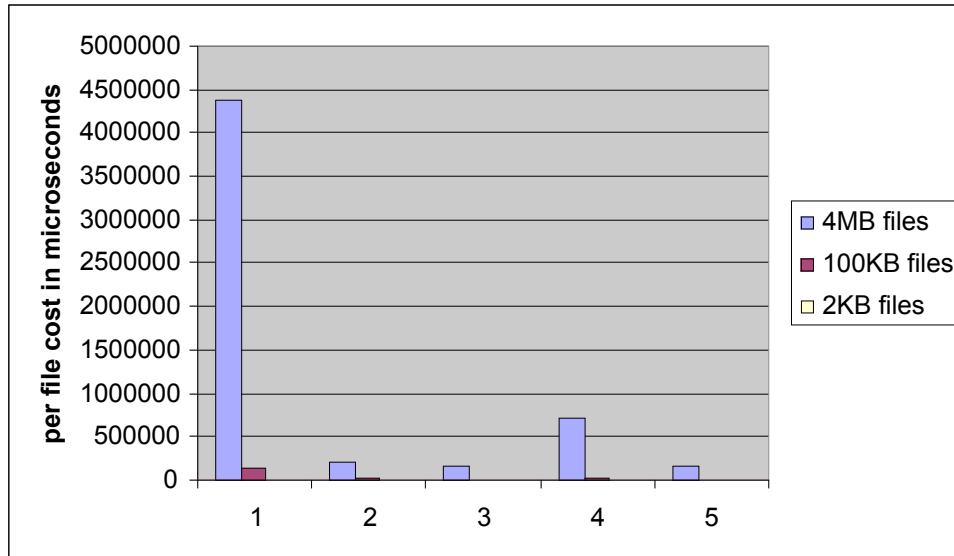


Figure 19. The cost per file for the 3 file sizes under the five different conditions

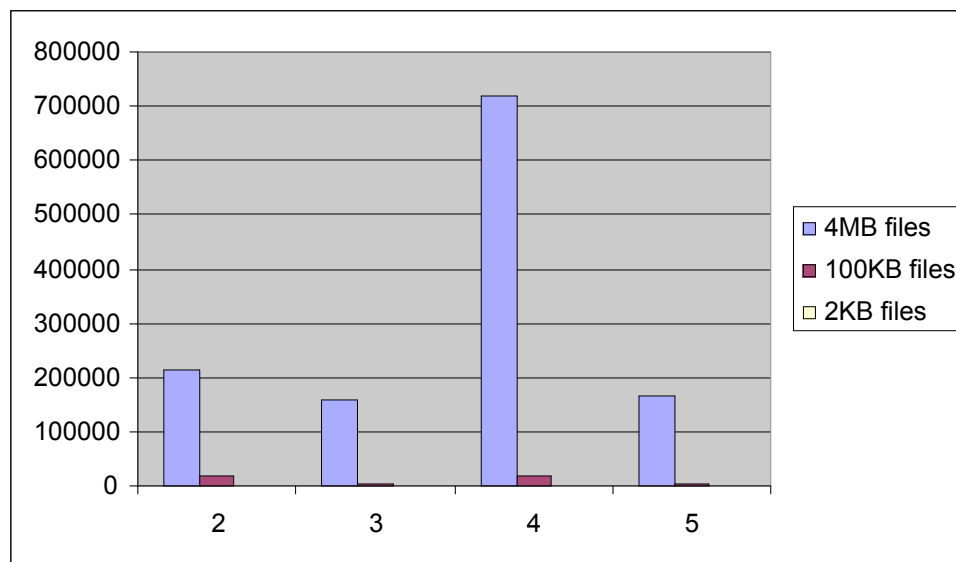


Figure 20. The cost per file for the 3 file sizes under conditions 2-5

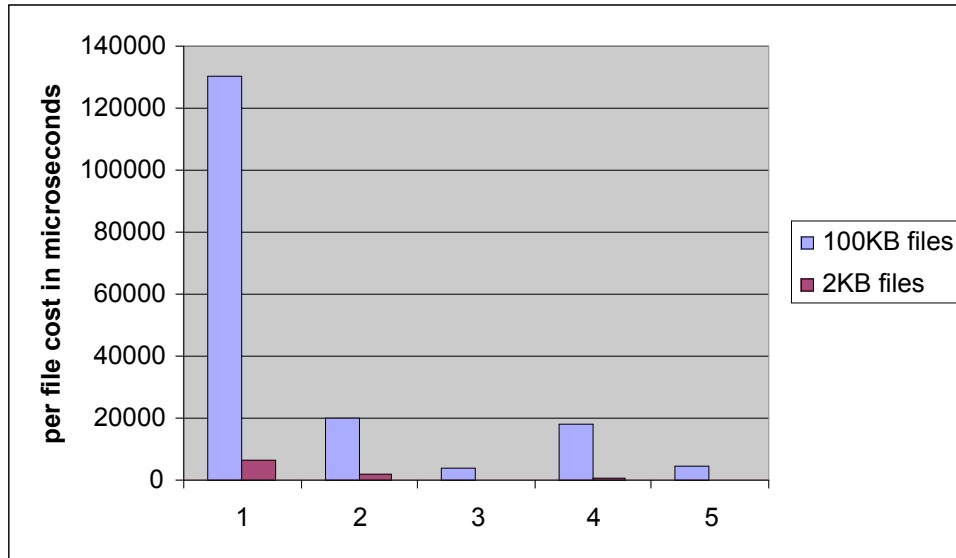


Figure 21. The cost per file for the 100 KB and 2KB file sizes under the five different conditions

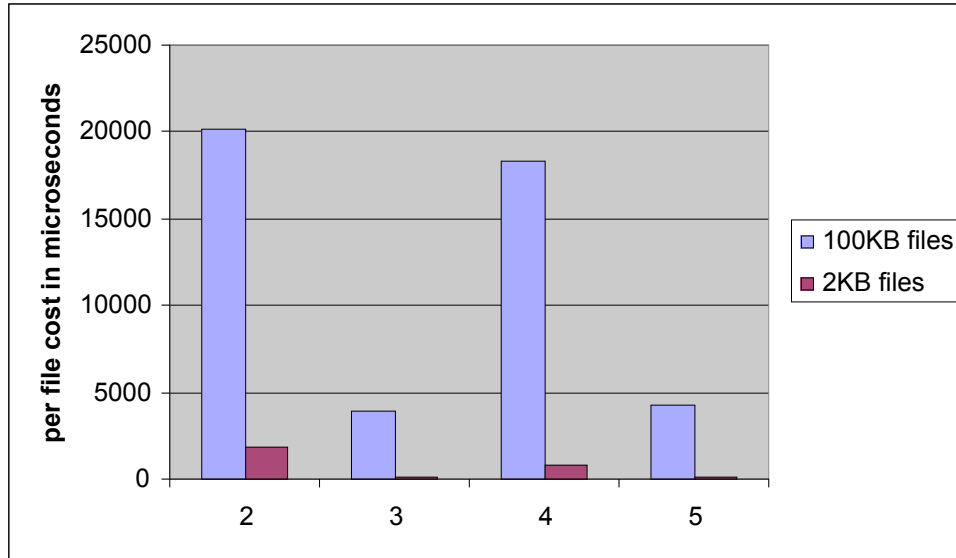


Figure 22. The cost per file for the 100 KB and 2KB files under conditions 2-5

6 Conclusion

In this thesis we discussed a CAS provider implemented as a character driver in the Linux kernel. This driver constitutes a layer on top of the disk driver layer and functions as a basic file system. In the future we plan on enhancing the driver by implementing additional features, as well as re-implementing some of the current features for better efficiency. In addition, we plan to perform a more sophisticated performance analysis by comparing the CAS driver to some popular file systems such as NFS and ext3. This driver will become a part of the larger CAS system shown in Figure 7. The code for the CAS driver will be made open source by posting it on the web.

7 Acknowledgements

I would like to thank Dr. Thomas Bressoud and Dr. Jessen Havill for all their help and feedback.

References

- [1] A. Rubini and J. Corbet, Linux Device Drivers, O'Reilly & Associates Inc., 2001
- [2] D. Bovet and M. Cesati, Understanding the Linux Kernel, O'Reilly & Associates Inc., 2001
- [3] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schroter and D. Verworner, Linux Kernel Programming, Pierson Education Limited 2002
- [4] J. Egan and T. Teixeira, Writing a Unix Device Driver, John Wiley & Sons, Inc., 1988
- [5] G. Pajari, Writing Unix Device Drivers, Addison-Wesley Publishing Company, Inc., 1992
- [6] G. Nutt, Kernel Projects for Linux, Addison-Wesley Longman, Inc., 2001
- [7] D. Mosberger and S. Eranian, IA-64 Linux Kernel, Hewlett-Packard Company, 2002
- [8] S. M. Sarwar, R. Koretsksky and S. A. Sarwar, Linux: The Textbook, Addison-Wesley Longman, Inc., 2002
- [9] W. Bolosky, S. Corbin, D. Goebel and J. Doucer, "Single instance storage in windows 2000", In proceedings of the 4th USENIX Windows Systems Symposium (Seattle, WA, August 2000), pp. 13-24
- [10] I. Clarke, O. Sandberg, B. Wiley and T. Hong, "Freenet: A distributed anonymous information storage and retrieval system", Lecture Notes in Computer Science 2009 (2001)
- [11] P. Cox, C. Murray and B. Noble, "Pastiche: Making backup cheap and easy", In proceedings of the Symposium on Operating Systems Design and Implementation (2002)
- [12] F. Dabek, M. Kaashoek, D. Karger, R. Morris and I. Stoica, "Wide-area cooperative storage with CFS", In Proceedings of 18th ACM Symposium on Operating Systems Principles (SOSP '01) (Chateau Lake Louise, Banff, Canada, Oct. 2001)
- [13] P. Druschel and A. Rowstron, "PAST: A large-scale, persistent peer-to-peer storage utility", pp. 75-80
- [14] S. Ratnasamy, S. Shenker and I. Stoica, "Routing Algorithms for DHTs: Some Open Questions", Technical Report, University of California, Berkeley, 2002
- [15] A. Muthitacharoen, B. Chen and D. Mazieres, "A Low-Bandwidth Network File System", In Proceedings of the 18th ACM Symposium on Operating Systems Principles (Chateau Lake Louise, Banff, Canada, Oct. 2001)
- [16] A. Muthitacharoen, R. Morris, T. Gil and B. Chen, "Ivy: A read/write peer-to-peer file system", In Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02) (Boston, Massachusetts, December 2002)
- [17] NIST, "Secure Hash Standard (SHS)", In FIPS Publication 180-1 (1995)
- [18] S. Quinlan and S. Dorward, "Venti: a new approach to archival storage", In Proceedings of the FAST 2002 Conference on File and Storage Technologies (2002)
- [19] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker, "A Scalable Content addressable network", In Proceedings of ACM SIGCOMM 2001
- [20] R. Rivest, "The MD5 Message-Digest Algorithm", RFC 1321 (1992)
- [21] A. Rowstron and P. Druschel, "Pastry: Scalable, Distributed Object Location and

- Routing for Large-Scale Peer-to-Peer Systems”, In Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware) (Heidelberg, Germany, 2001)
- [22] I. Stoica, R. Morris, D. Karger, M. Kaashoek and H. Balakrishnan, “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications” In Proceedings of the ACM SIGCOMM 2001 (San Diego, CA, 2001)
- [23] “John Halleck's SHA implemented in C”, <http://www.cc.utah.edu/~nahaj/c/sha/>
- [24] “MD5 Homepage”<http://userpages.umbc.edu/~mabzug1/cs/md5/md5.html>
- [25] “OOPWeb.com”,<http://oopweb.com/Algorithms>
- [26] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, T. Bressoud and A. Perrig, “Opportunistic Use of Content Addressable Storage for Distributed File Systems”
- [27] I. Clarke, “A Distributed Decentralized Information Storage and Retrieval System”, Report of the Division of Informatics at the University of Edinburgh, 1999
- [28] A. Silberschatz, P. Galvin and G. Gagne, Operating System Concepts, John Wiley & Sons, Inc., 2002
- [29] T. Cormen, C. Leiserson and R. Rivest, Introduction to Algorithms, McGraw-Hill Book Company, 1999