The Concrete Architecture of the FreeBSD Kernel with a focus on the File System

EECS 4314: Advanced Software Engineering
Dr. Zhen Ming (Jack) Jiang
Assignment 2 Report
March 8th, 2023

Group Sudo:
Michael Koiku
Swapnilkumar Parmar
Henry Gu
Amin Mohammadi
Ayub Osman Ali
Omar BaGunaid
Michelangelo Granato
Hieu (Aaron) Le
Vikramjeet Gill
Jai Ramotar

Abstract

This report examines the concrete architecture of the FreeBSD kernel with a focus on the file system. It provides an overview of the FreeBSD kernel, including its design principles, components, and features.

It then delves into the details of the file system, including its structure, organization, and implementation. Additionally, the report evaluates the FreeBSD kernel as a monolithic design with a modular framework, composed of two main components: the machine-independent section, and the machine-dependent part.

Finally, the report concludes with a discussion of the advantages and disadvantages of the FreeBSD kernel and its file system. The report provides an in-depth look at the FreeBSD kernel and its file system.

Introduction and Overview

Popular open-source Unix-like operating system FreeBSD is known for its robustness, security, and reliability. FreeBSD's kernel, which manages system resources, schedules processes, and acts as an interface between hardware and software, is the foundation of the operating system. The FreeBSD kernel is a complex piece of software with several subsystems, each responsible for a certain set of functionalities. The File System, which maintains data storage files and directories on disc, is one of the most important subsystems of the FreeBSD kernel.

FreeBSD utilizes a hierarchical file system concept in this subsystem, in which all files and directories are arranged in a tree-like layout, with the root directory serving as the top-level directory. Furthermore, FreeBSD supports a variety of filesystem types, including the widely used UFS (Unix File System), ZFS (Zettabyte File System), and NFS (Network File System). Its versatility enables users and system administrators to choose the most suitable filesystem for their unique requirements, taking into account elements like performance, scalability, and compatibility.

Understanding the architecture of the FreeBSD kernel, especially its File System subsystem, is critical for anybody interested in operating system design and implementation. In this report, we will explore at the concrete architecture of the FreeBSD kernel, with a particular emphasis on the File System subsystem. We examine the file system's concrete architecture, which is the system's as-built architecture as retrieved by the Understand software analysis tool. We will divide the system's top-level entities into subsystems and identify any relevant design patterns and architectural styles utilised in the file system.

In addition, we will compare the file system's concrete architecture to its conceptual architecture to discover any differences and analyse the reasons for them. Lastly, we will exhibit redrawn diagrams of the concrete design and use sequence diagrams to highlight important parts of the architecture and its subsystems.

Derivation Process

To begin our derivation process to create our containment file, we began by reading the official FreeBSD documentation, man pages, and the official FreeBSD repository. We were able to group some directories from the repository into their appropriate systems based on our understanding of the documentation and the README.md files found in the repository.

Not all directories could be ground into subsystems just by reading the documentation and README.md files. For the directories that could be grouped into the identified subsystems from our conceptual architecture, we wrote a python script to extract those directories into subsystems and produce the corresponding contain statements. We then updated the .raw.ta file to include the new substems.

We then leveraged LSEdit to visualize the instances of the systems to observe the hierarchical decomposition. We then performed manual inspection of the code to be able to classify what subsystem they belonged to. We repeated the overall process multiple times until we achieved a desired result.

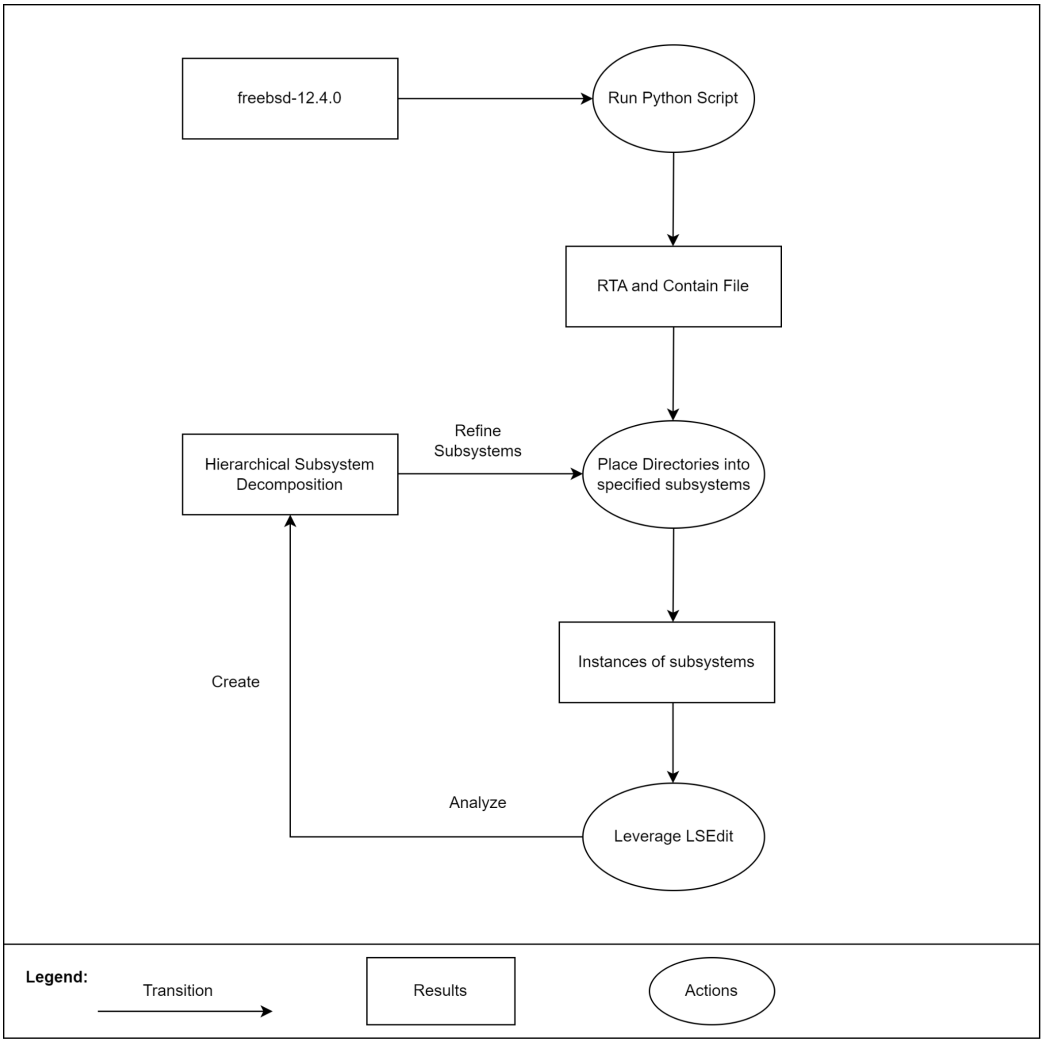


Figure 1: The Derivation Process

Architecture

In our first report we proposed that the architecture of the FreeBSD **Kernel** is that of a layered architectural style and can be divided into two sub-layers, A Machine Dependent layer and a Machine Independent layer.

1) Machine Independent: This part of the kernel is responsible for managing system resources that are independent of the hardware platform. It includes the following components:
    - Process Management: This subsystem is responsible for creating, managing, and scheduling processes and threads within the system.
    - Memory Management: This subsystem is responsible for managing the allocation and deallocation of system memory and virtual memory.
    - File System: This subsystem provides a uniform interface for accessing different types of file systems and managing file and directory operations.
    - Network Communication: This subsystem provides the infrastructure for network communication within the system, including network protocols, sockets, and device drivers.
    - Inter-Process Communication: This subsystem provides mechanisms for inter-process communication, including pipes, message queues, and **semaphores**.
    - Terminal Handling: This subsystem provides the infrastructure for handling terminal input and output.

2) Machine Dependent: This part of the kernel is responsible for managing system resources that are specific to the hardware platform. It includes the following components:
    - Interrupt Handling: This subsystem manages interrupts from hardware devices and coordinates their handling by the kernel.
    - Low-level System Startup Actions: This subsystem handles low-level system startup actions, including **firmware** initialization, kernel initialization, and system bootstrap.
    - Device Drivers: This subsystem provides interfaces for interacting with hardware devices and managing I/O operations.
    - Hardware Configuration and Initialization: This subsystem configures and initializes hardware devices.

After many iterations of our extraction process, we were able to successfully settle on a concrete architecture of the FreeBSD kernel that matched our conceptual architecture, although there were some additional subsystems we recovered that were not part of our conceptual architecture in Report 1.

From our concrete architecture, we were able to specify the following subsystems and their functions in their respective layers:

- Machine Independent Layer - In this layer, all subsystems were recovered in addition to new subsystems. We shall only discuss the functionality of the newer subsystems discovered.
    - CDDL: This subsystem contains various commands and libraries under the Common Development and Distribution License.
    - Release: This subsystem contains release building makefile & associated tools.
    - Regression Tests: This subsystem is responsible for running automated tests to ensure that the kernel is functioning correctly
    - Common Access Method: This subsystem is responsible for providing a unified interface for accessing various types of hardware devices
    - Security: This subsystem is responsible for providing security features, such as authentication, authorization, and encryption
    - Compatibility: This subsystem is responsible for providing compatibility with other operating systems, such as Linux.
    - GNU Debugger: This subsystem is responsible for providing support for the GNU Debugger (GDB).
- Machine Dependent Layer: In this layer, we discovered new subsystems, and realized that some of the subsystems in our conceptual architecture were combined into a single subsystem in the concrete architecture.
    - CPU: This subsystem manages interrupts from hardware devices and coordinates their handling by the kernel and low-level system startup actions, including firmware initialization, kernel initialization, and system bootstrap.
    - Device: This subsystem provides interfaces for interacting with hardware devices, managing I/O operations, and configuring and initializing hardware devices.
    - Sys: This subsystem is responsible for providing support for system calls, such as open(), close(), read(), and write().
    - Mips: This subsystem is responsible for providing support for the MIPS processor architecture.

Hierarchical Decomposition

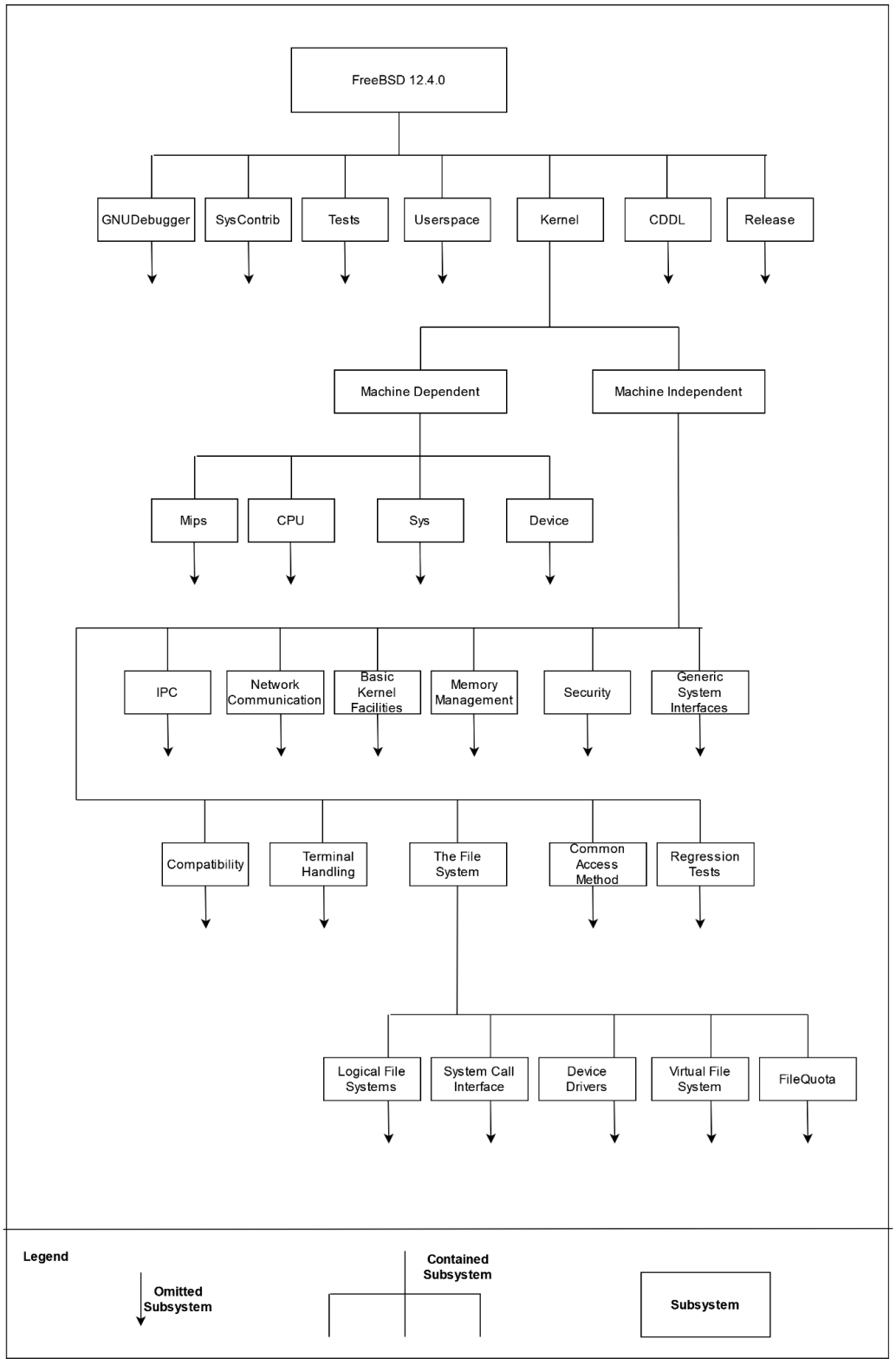


Figure 2: The Hierarchical Decomposition of FreeBSD 12.4.0

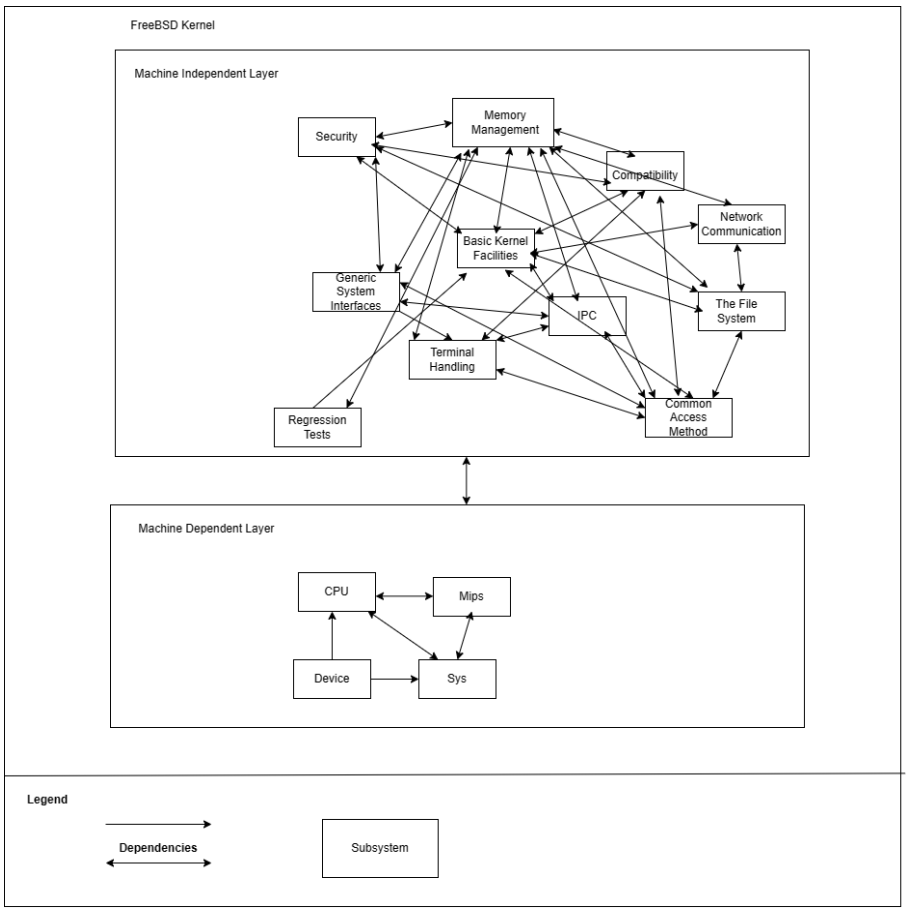


Figure 3: Concrete Architecture of the FreeBSD Kernel

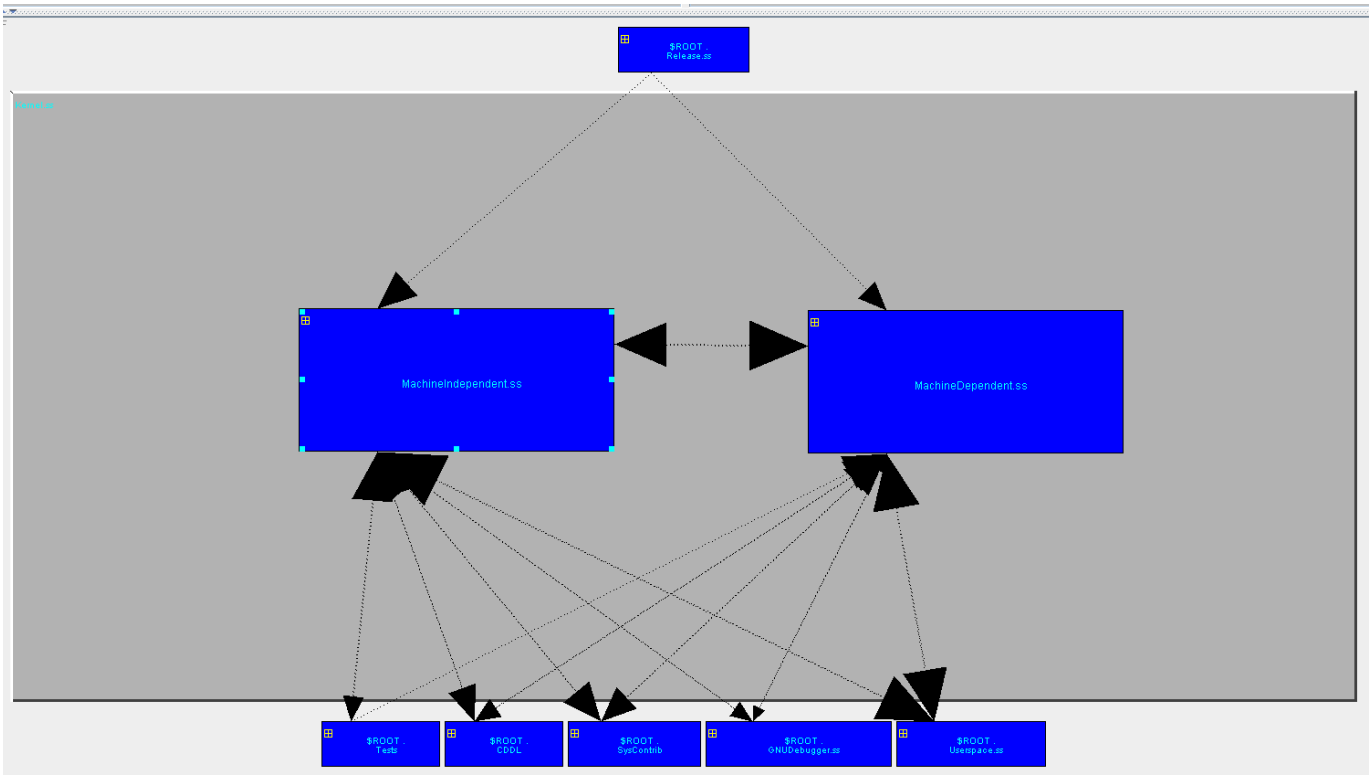


Figure 4: LSEditor Top Level View of the Kernel

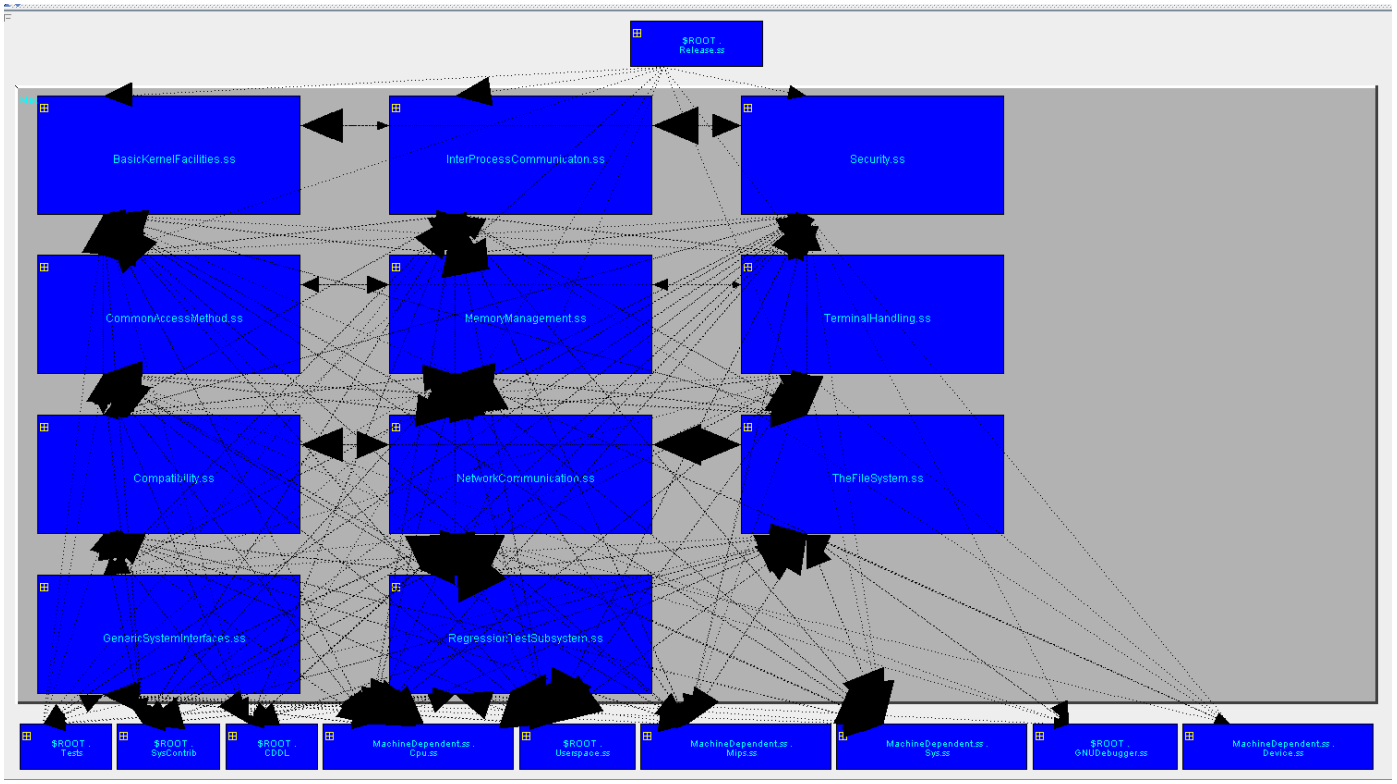


Figure 5: LSEditor Top Level View of the Machine Independent part of the Kenel

Conceptual Architecture of the File System

The conceptual architecture of FreeBSD's filesystem is a layered architecture. At the top is the *virtual filesystem*, known as the VFS. The VFS uses the strategy pattern to select the relevant logical filesystem implementation to handle the low level details of the filesystem, and combines multiple filesystems by *mount*ing them at different locations. For example, the root of the virtual filesystem (path */*) could be using a local file system, such as the Unix File System (UFS). Then a Network File System (NFS) could be mounted at the home directory (path */home*) so that all users have access to the same personal files, regardless of the system they login to. The VFS is responsible for translating operations to files in the root directory to the UFS implementation, except if they are under the home directory, then they are sent to the NFS implementation. In an actual system, there will be many mounts for all the various uses. Various pseudo-filesytems used to interact with other kernel interfaces, such as procfs, devfs, as well as pseudo-filesystems such as tmpfs for volatile temporary files stored in memory, different mounts for the root of the filesystem, for the home directory, log directory, and even more needed for specific application requirements. All of these mounts are easily configured from userspace by the system administrator. The VFS implements a system call interface that provides user space applications with many system calls used to manipulate files, such as open, close, read, write, stat, etc. (and all their variants). The VFS also handles other aspects that are shared between the other various logical filesystems. The VFS is responsible for enforcing unix permissions as well ACLs on files and directories. The VFS also provides file-level caching, so repeatedly reading a file does not need to wait for the lower levels to reread every time. Additionally, VFS also caches writes, so programs do not need to wait for the filesystem to commit data before they can work on something else, as well as buffering writes such that repeated writes result in only the final file contents being written.

Below the VFS are the actual logical filesystems, these include the aforementioned UFS and NFS, but also others such as ZFS, EXT2/3/4, and more. The details of these file systems vary greatly, and some are old and rarely used outside of legacy situations, other are more modern and selected based on the usecase, and even used simultaneously for different parts of the system. However, something that all of these logical file systems share in common is that they provide the ability to store files and organize them in a directory hierarchy, which is used by the VFS to provide it's higher level services.

The layer below the logical file systems depends on the specific implementation. For example, NFS being a networked file system would depend on the networking subsystem and connect to other hosts on the network. Other local file systems, such as UFS, EXT2/3/4, etc. read from and write to character devices, which simply store a contiguous sequence of bytes, and the file system (UFS, EXT2/3/4) organize files and directories into these devices.

Concrete Architecture of the File System

The file system subsystem of FreeBSD 12.4.0 kernel is a complex architecture consisting of several layers and components. The main components are:

1) File System Interface Layer: This layer provides a standard interface for all file systems in the system. It is responsible for managing file system mount points, maintaining the file system namespace, and performing file system operations such as file and directory creation, deletion, and modification.
2) Virtual File System Layer: This layer provides an abstraction of the file system that allows different file systems to be mounted at different points in the file system namespace. It provides a uniform interface for accessing files and directories across different file systems and provides support for file system quotas and other file system-related features.
3) Buffer Cache Layer: This layer provides a cache for frequently accessed blocks of data in file systems. It helps to improve performance by reducing the number of disk accesses required for file I/O operations.
4) Block I/O Layer: It provides a layer of abstraction between the file system and the underlying storage devices. It allows the file system to interact with the storage devices in terms of blocks, rather than individual sectors, and handles buffering and caching of data to optimize I/O performance.
5) Device Driver Layer: It provides the interface between the file system and the physical devices that store the data. In addition, the device driver layer provides a mechanism for caching data in memory, which can improve performance by reducing the number of I/O operations required to read and write data to the storage devices.
6) File System Implementations: This layer provides the implementation of file systems that are mounted on the system.

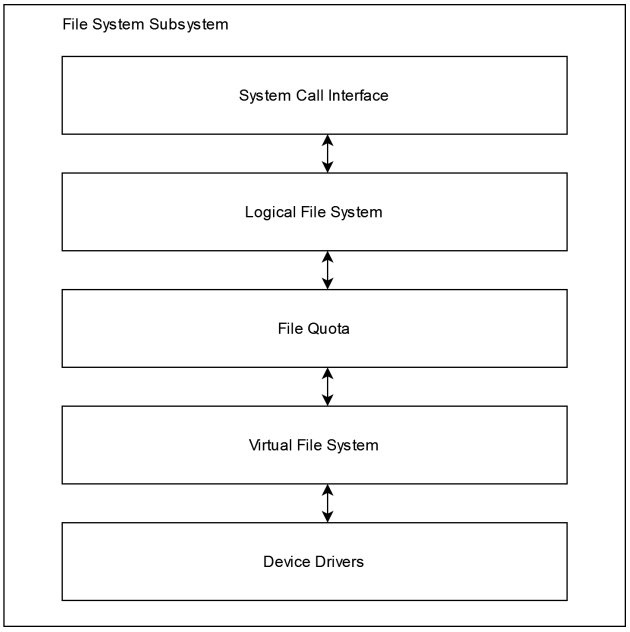


Figure 6: File System Concrete Architecture

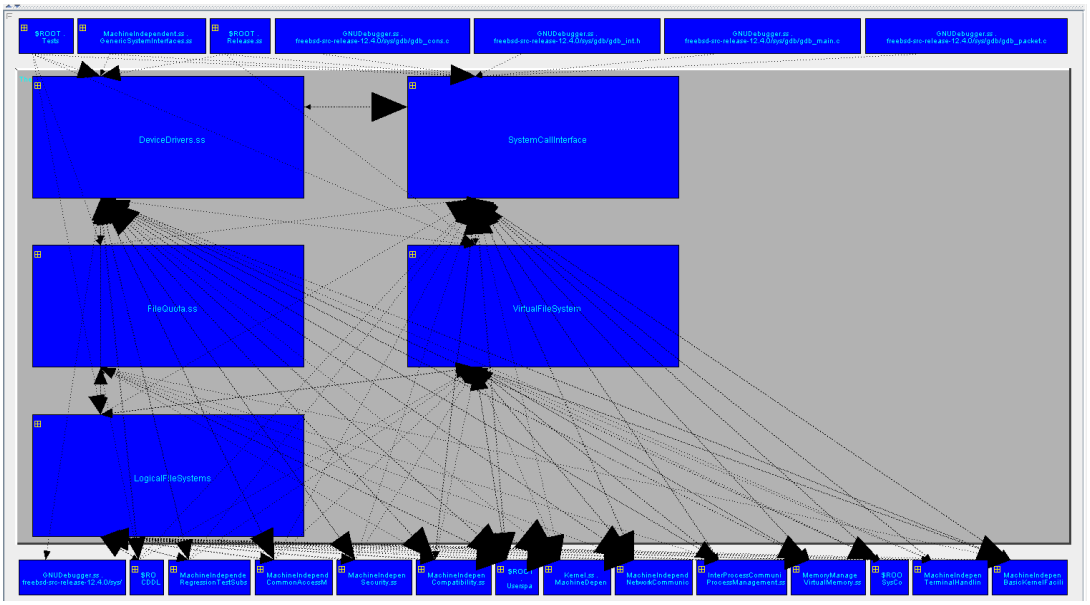


Figure 7: LSEditor Top Level View of the File System subsystem

The different file systems supported by FreeBSD are:

1) UFS (Unix File System): This is the default file system for FreeBSD, and it supports features like soft updates and journaling.
2) ZFS (Zettabyte File System): This is a modern file system that provides advanced features like snapshots, compression, and RAID.
3) NFS (Network File System): This is a distributed file system that allows files to be shared over a network.
4) CD9660 (ISO 9660 File System): This is a read-only file system commonly used for CDs and DVDs.
5) MSDOSFS (Microsoft DOS File System): This is a file system that will permit the FreeBSD kernel to read and write MS-DOS-based file systems. It is commonly used for removable media like USB drives.
6) ext2/ext3/ext4 (Extended File System 2/3/4): These are extended file systems commonly used in Linux.
7) NTFS (New Technology File System): This is a proprietary journaling file system developed by Microsoft.

Architectural Styles and Design Patterns in File System

Similar to the architectural style used throughout the FreeBSD kernel, the file system also uses a **layered architecture**, with each layer responsible for a specific aspect of file system management. The **vnode/vfs pattern** has been used in FreeBSD and other operating systems to provide a layered architecture that allows for flexibility in supporting different types of file systems and file operations. We can see evidence of the layered architecture in the way that the various layers are initialized and connected to each other during the system startup process. For example, the Virtual File System layer is initialized early in the boot process and is responsible for loading the appropriate File System Implementation implementations for the file systems that are mounted on the system.

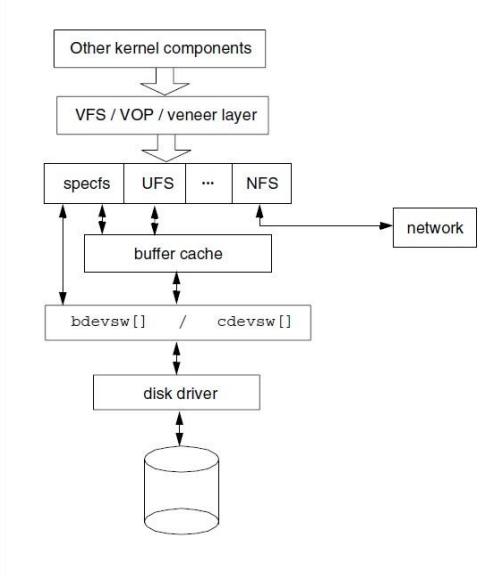


Figure 8: VFS Architecture

In FreeBSD, the file system code uses the **iterator pattern** to traverse directories and read the contents of files, and the interface for it is provided by the VFS subsystem. The ***vop_readdir()*** VFS operation returns directory entries one by one through the use of a directory iterator object. The iterator pattern allows the file system to provide a consistent and efficient way of iterating over directory entries, regardless of the specific file system implementation.

The FreeBSD file system uses the **strategy pattern** to allow for different file system implementations to use different algorithms for performing various operations such as reading or writing data to disk. The strategy design pattern is a behavioral software design pattern that enables selecting an algorithm at runtime. In FreeBSD, the strategy pattern is used in the VFS layer, where the buffer cache subsystem provides a common interface for file systems to read and write data to disk (figure 5). Each file system provides its own strategy routine, which is responsible for performing the actual read or write operation using the appropriate algorithm. The buffer cache subsystem can be found in the file ***sys/kern/vfs_bio.c***, and the strategy routines for different file systems can be found in their respective source files (e.g., ***sys/ufs/ffs/ffs_vnops.c*** for the UFS file system).

FreeBSD also uses the **Observer pattern** in the file system to allow different components of the system to be notified when a file system event occurs. This pattern is implemented through the use of the ***vnode*** system in FreeBSD, which allows for the creation of *vnode* objects that represent files, directories, and other entities within the file system. When a file system event occurs, such as a file being opened, the appropriate *vnode* is located and a notification is sent to any registered observers.

In FreeBSD, the **Factory pattern** is used to create various objects in the file system. For example, the *vnode* creation function, ***vn_create***, is responsible for creating *vnodes* (virtual nodes) in the file system. This function uses a *vnode* factory, ***vnode_pager_alloc***, to create the *vnode* object and it can be found in the file ***sys/vm/vnode_pager.c***. This is an example of the Factory pattern because the ***vnode_pager_alloc*** function is responsible for creating *vnode* objects without the caller knowing the details of the allocation process.

Reflexion Analysis and Rationale for Differences

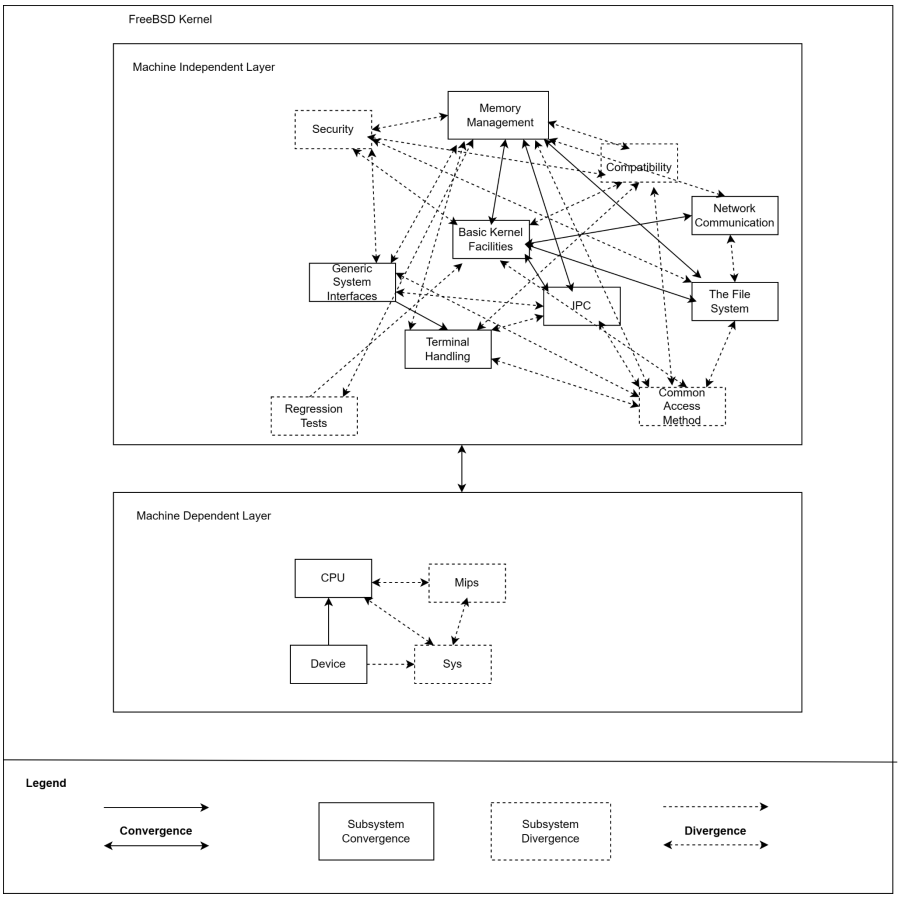


Figure 9: Reflexion Diagram

In our concrete architecture we discovered more subsystems that provided additional functionality to the oversystem that our conceptual architecture did not take into consideration. The subsystems missing in our conceptual architecture include the Security, Compatibility, Common Access Method, Regression Tests, Mips, and Sys subsystems.

We also discovered that the 4 services outlined in the machine dependent part of our conceptual architecture were provided by two subsystems in the concrete architecture which are the CPU and Device subsystem.

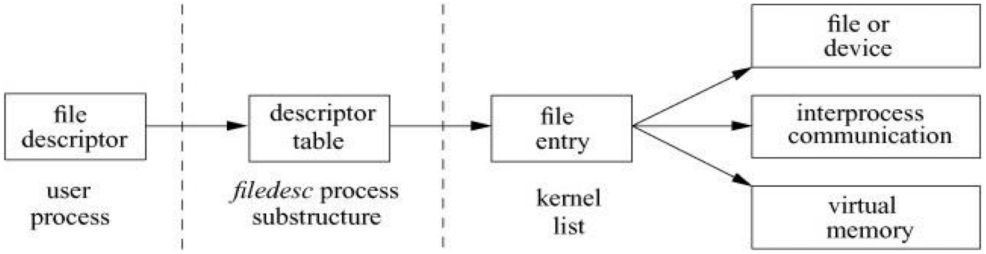
Finally, we also discovered more dependencies between existing subsystems that did not exist in our conceptual architecture.

Interfaces

Within the operating system, files are organized in different directories. These directories contain info and pathnames to other files and directories and this would form a tree-structured hierarchy called the File System. The File System is a critical layer in FreeBSD that functions as a library of different types of files and device drivers. This layer helps organize and store data, provide access and control to users so that they can alter and manage their files, directories to their liking as well as provide protection for the files from unauthorized access if needed. Since the File System is an internal layer of the operating system, it doesn't have any external interfaces as parts of it. For that reason, in order to perform said functions, FreeBSD is built so that this complex internal layer could work with various external interfaces and each of them is responsible for a different purpose:

- **File Descriptor Interface**
    - The **File descriptors** are used to access files, sockets, pipes, and other I/O resources such as character devices and blocked devices. The file descriptors are represented as positive integers starting from 0, they are used in system calls such as **open**(), **read**(), **write**(), and **close**().
    - The open **file descriptors** of a process in FreeBSD are controlled by an array of pointers to file entries, which is indexed by the file descriptor value. Each element of the array holds information about some open file.

- The list or array of file descriptors is saved in the kernel data structure known as the **file *descriptor table*** *of* the process. The table is constructed when some process starts and can be changed as the process opens or closes files. The descriptor table is also used to keep track of open files and perform some I/O operations on those files. When a process performs a read or a write operation, the kernel will look for a matching **file entry** in the table and conduct the operation on that file using the file descriptor.



- ***File descriptor*** *table* can point to different types of resources depending on their purpose. For example, it may reference a socket if it is used for interprocess communication. Alternatively, if the file entry is used for such an unnamed high-speed local communication, it will point to a pipe, while named high-speed local communication will reference a fifo.

- **Virtual File System Interface**
  - The **Virtual FileSystem** (**VFS**) interface layer stands between the file system implementation and the rest of the kernel. It provides an interface for working with different file systems, regardless of their implementation or physical location. The **VFS** layer provides an abstraction layer for file systems and allows them to be implemented in different ways, for example, network file systems, in-memory file systems, and disk-based file systems.
  - It provides a subsystem called the Vnode layer that manages multiple files through their reference points, attribute flags, field names, etc and forms a pathway to them so users can easily access these files via system calls.
  - This layer lets applications access and manage files using system calls, regardless of the file system implementation. The **Virtual FileSystem** interface also maintains the file system abstraction layer, including file system mounts, which is one of the essential elements of file systems managed by the VFS layer, by attaching a file system to a mount point in the file system hierarchy.
  - Read, write, and directory, are simply a few of the common file system operations that the VFS layer makes available to file systems. The VFS layer is expandable, meaning it is accepting a variety of new file systems to the system without any need to change the kernel. This can be done by creating a standard set of interfaces and callbacks that file systems operate to make the communication with the VFS layer.
  - **Vnode Interface:** The **vnode** interface is a core part of the VFS the Vnode interface is a layer that provides a consistent interface making it easier to access different types of file systems such as local or network file systems. These file systems are represented as a tree of **vnode** objects that are dynamically created. There are operations that the **vnode** interface can perform on **vnode** objects such as opening, closing, reading and writing to a file as well as going through directories.
- **System call interface:** a critical component between the user-level applications and the kernel. This layer is responsible for receiving service requests from the users and transfer to the lower level for execution. Therefore, it is one of the most vital kernel components that works with the file system as many tasks such as files sharing or accessing, establishing connection between computers wouldn't be executed without it.
- **Network interface:** Comprising the network device drivers, the socket interface, network protocols and the Network File System, this layer is responsible for enabling communication between the system and others through a network. With this interface, the File System is able to share and receive files between machines with ease.
  - **Network File System (NFS)**: The NFS is not an actual interface on its own, but rather a part of the combination that form the network interface described above. Designed as a client-server application, NFS enables users access to files remotely on another machine through **remote procedure call** (RPC) requests and data marshaling. The said marshaling process breaks down files, bundles them up as network byte order then sends them to the clients based on order of the RPC requests and reconstructs the data into the original file again. As a result, the data accessed via NFS could be processed and used as if it is locally stored. In addition to the ease of

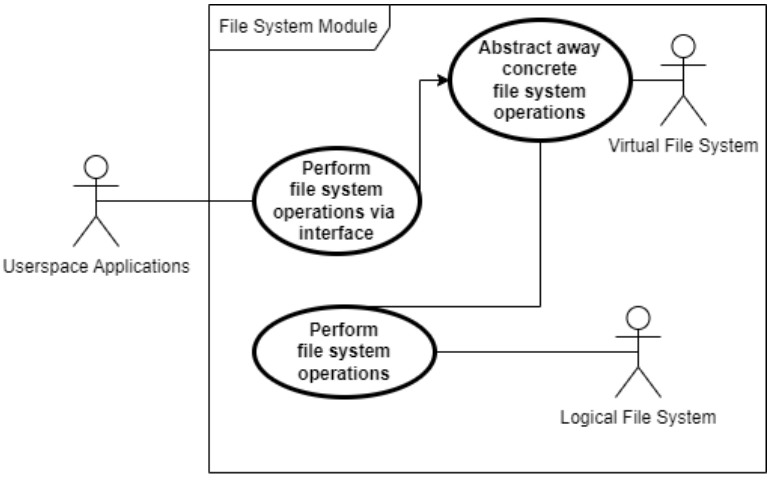
file accessing, NFS also makes file sharing easier as this protocol is designed to run on different filesystems, even the one with less-rich semantics such as MS-DOS filesystem.

- **Network Drivers:** These are the character-device interface of many devices that are responsible for establishing connection between the operating system and the network interface and aiding in the file sharing process. Their main tasks include managing and stabilizing network connections, configuring network devices setting, working as a medium to receive and transfer data packets.

Use Cases

Virtual File System (VFS)

In many cases, user-space applications need to perform file operations. Since FreeBSD supports multiple file systems, there is a need for an abstraction layer to avoid coupling of the logical file systems and user-space applications. This is where the Virtual File System (VFS) comes in, providing an abstraction layer and interface for applications to use.

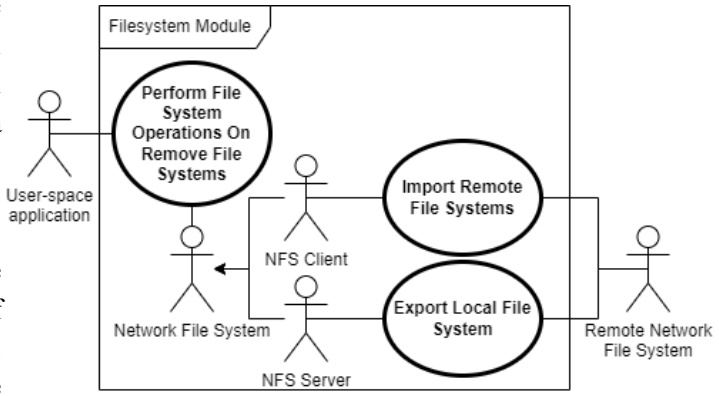


The VFS acts as a mediator between user-space applications and the actual file system, translating the operations that the application requests into the specific commands that the file system understands. This allows user-space applications to work with files in a consistent and predictable way, regardless of the underlying file system.

By abstracting away the details of the file system, the VFS makes it easier for developers to create applications that work with files. This is particularly important in multi-platform environments, where different operating systems may have different file systems with varying characteristics and features. By using the VFS, applications can be developed in a way that is platform-independent, making them more portable and easier to maintain.

Network File System (NFS)

It is common for user-level applications to require access to remote file systems and conversely, it is desirable to be able to make local files available to share to remote systems. The Network File System (NFS) is FreeBSD's solution for these use cases, as it provides a reliable and compatible architecture.



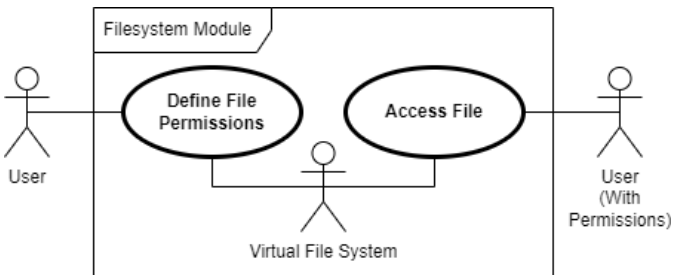
The NFS protocol allows users to access files on remote machines as if they were stored locally, without needing to worry about the details of the remote file system. The NFS supports a range of operations for file access, including read, write, delete, as well as permissions handling, making it a versatile tool for file management in distributed environments.

Overall, the NFS protocol is a powerful solution for managing file systems in distributed environments. Its Client-Server architecture allows for , and its support for a range of file operations makes it a versatile tool for managing data across multiple machines.

Security

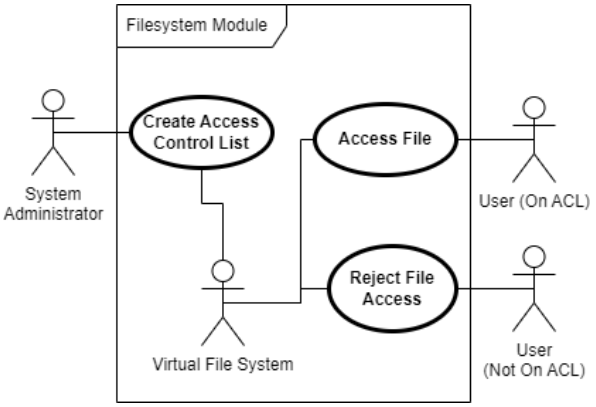
Security and privacy control are critical aspects of file system management, particularly in multi-user environments. In the FreeBSD operating system, the Virtual File System (VFS) provides several solutions to enable users to control file privacy effectively.

One of the most commonly used solutions for access control in FreeBSD is Discretionary Access Control (DAC). With DAC, users



can set permissions on files and directories to determine who can access them and what actions they can perform on them. This simple form of access control allows users to have complete control over their files, and is easy to use and implement.

In addition to DAC, FreeBSD's VFS also supports Access Control Lists (ACLs), which provide a more granular form of access control. ACLs allow users to set permissions for specific users or groups of users, enabling more fine-grained control over file access. This makes it easier to manage large groups of users and to ensure that each user has access only to the files they need.



Overall, the combination of DACs and ACLs in the FreeBSD file system provides powerful tools for managing file security and privacy. By giving users control over their files and enabling fine-grained access control, FreeBSD allows users to work effectively and safely in multi-user environments. This makes it an ideal choice for organizations that require strong file security and privacy controls.

Data Dictionary

Term	Definition	Type
Process	A process is a unit of execution or a program that has its own memory space, file descriptors, and system resources.	
Kernel	kernel refers to the core component of the operating system that manages system resources and provides interfaces for applications to interact with the hardware.	
Strategy Pattern	Strategy Pattern is a design pattern that is used in software development to allow for the dynamic selection of algorithms or strategies at runtime.	
Factory Pattern	Factory Pattern is a design pattern commonly used in software development to encapsulate the creation of objects and provide a means for creating objects without specifying their concrete types.	
Iterator Pattern	The Iterator Pattern is a design pattern used in software development to provide a way to access the elements of a collection or container without showing their actual implementation.	
Observer Pattern	Observer Pattern is a design pattern used to establish a one-to-many relationship between objects. It defines a subscription mechanism where multiple observers can register to receive notifications from a subject.	
vnode/vfs pattern	The vnode/vfs pattern is a design pattern used in FreeBSD to provide a uniform interface for accessing different file systems.	
Vnode	Vnode objects represent files, directories and other file system objects.	
File Descriptor	A file descriptor is a non-negative integer used by an operating system to uniquely identify an open file or input/output resource.	Int
File Entry	A file entry is a data structure that represents an open file in the	

	kernel. It contains information about the file and is used by the kernel to manage the file and its associated resources.	
Vnode_Pager_Alloc	is a function used to allocate and initialize a vnode-backed memory object for use with the VM system.	
Layered Architecture	In layered architecture, components that perform similar or related roles are grouped together into horizontal layers. Each layer is responsible for 1 specific role ex: VFS layer is responsible for abstraction of the file systems and nothing else.	
vop_readdir()	Is an operation that returns directory entries one by one through the use of a directory iterator object.	
vn_create	Is the function that creates Vnodes.	
Remote Procedure Call (RPC)	Is a request that users can make via the Network File System to access files remotely.	
Semaphores	A variable used to control access to some common resource (ex: a file).	
Firmware	Software that handles low level control of hardware components for machines.	

Naming Conventions

A linear array of bytes that can be read and written starting at any byte is called a **regular file**. These can be named with a string of up to 255 characters called a **filename**. The file that filenames are stored in are called **directories**, which is a type of file that can be read by a process but only the kernel can modify. The information in a directory about a file is called a **directory entry** and includes the filename and a pointer to the file itself. Since directory entries can refer to other directories and plain files, a hierarchy of directories and files called a **filesystem** is formed. Filesystems have a tree-like structure, the beginning of which is the **root directory** or **slash (/)** that contains files. A string composed of zero or more filenames separated by slash (/) characters that a process identifies a file by is called the file's **pathname**. A pathname beginning with a slash is called an **absolute pathname**, and is interpreted by the kernel starting with the process's root directory. A pathname that does not begin with a slash is called a **relative pathname**, and is interpreted relative to the current working directory of the process. The system call for a process to set its root directory is called **chroot** and the system call for a process to set its current directory is called **chdir**. The filesystem that anchors all absolute pathnames is called the **root filesystem**. The system call that takes the name of an existing file and another name to create for that file is called **link** and the system call that removes filenames is called **unlink**. The system call to create directories is called **mkdir** and the system call to remove directories is called **rmdir**.The system call that sets the owner and group of a file is called **chown**. The system call that changes protection attributes is called **chmod**.The system call to read back the owner/group of a file and the protection attributes of a file is called **stat**. The system call to give a file a new name in the filesystem is called **rename**. The system call to shorten a file to an arbitrary offset is called **truncate**.Void areas in the linear extent of the file where data have never been written are called **holes**.The **filestore** is what is responsible for the organization and management of the data on the storage media. The three different filestore layouts 4.4BSD supports are called **Berkley Fast Filesystem, log-structured filesystem**, and **memory-based filesystem.**

Conclusions & Lessons Learned

From our extensive study of the file system subsystem in FreeBSD, we have learned several lessons for architecting an efficient file system. Initially, the file system needs to be scalable to manage an expanding number of concurrent requests. It is crucial that the system can process these requests without slowing down as more users and applications utilize the file system.

Moreover, other services that rely on the file system should be designed as an abstraction. Design patterns, such as the facade design pattern, can be used to accomplish this and assist the system in becoming less complex. The file system can be easily handled by being abstracted so that other services can access it with a more straightforward interface.

The file system should be built for performance as it is an essential component of the entire system. This covers optimizing disc I/O, cache management, and file access. Moreover, features like soft updates and snapshots, which enable point-in-time copies of the file system and offer a more practical approach to changing file system metadata, can enhance file system efficiency.

Another crucial aspect to take into account while designing a file system is security. Encryption can also be used in FreeBSD to secure user information and avoid unauthorized access to the file system, although access control measures ensure that only permitted users and applications have access to the file system.

References

Https://docs.freebsd.org/en/. https://docs.freebsd.org/en/. (n.d.). Retrieved February 10, 2023, from https://docs.freebsd.org/

McKusick, M. K. (2006). *The design and implementation of the 4.4BSD operating system*. Addison-Wesley.

FreeBSD. (n.d.). Retrieved from https://github.com/freebsd/freebsd-src

Kernel.org. (n.d.). Retrieved from https://www.kernel.org/doc/html/next/filesystems/vfs.html

FreeBSD Man. (n.d.). Retrieved from https://man.freebsd.org/cgi/man.cgi?query=filesystem&&manpath=SunOS+5.9