The Conceptual Architecture of the FreeBSD Kernel

EECS 4314: Advanced Software Engineering
Dr. Zhen Ming (Jack) Jiang
Assignment 1 Report
February 9th, 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 seeks to explore the conceptual structure of the FreeBSD kernel. By examining the official FreeBSD documentation and referring to related architectural designs, this essay evaluates FreeBSD kernel architecture as a monolithic design with a modular framework. This analysis reveals that the FreeBSD architecture is composed of two main components: the machine-independent section, and the machine-dependent part.

The machine-independent element includes system amenities such as basic kernel features, memory management assistance, widespread system interfaces, the filing system, terminal-management aid, interprocess-communication services, and system communication assistance.

On the other hand, the machine-dependent portion consists of platform-specific modules including low-level system start-up, trap and fault handling, execution-time support for I/O devices, low-level manipulation of the run-time context of a process, and hardware device configuration and initialization.

The FreeBSD kernel is a complex system that requires a thorough understanding of its architecture in order to fully utilize its capabilities. By analyzing the two main components of the FreeBSD kernel, this essay has provided an in-depth look into the structure of the FreeBSD kernel and its modular framework. This understanding of the FreeBSD kernel will enable developers to better utilize its features and create more efficient and effective applications.

Introduction and Overview

The 4.4BSD Operating System is a major advancement in operating system design and engineering. Developed by the University of California, Berkeley, it was first released in 1994 and has since become a foundational part of the BSD family of Unix-based operating systems. It is well-known for its streamlined architecture that makes it simple for developers to comprehend and customize. The system consists of two primary components: user-level programs providing the graphical user interface and other services, and the kernel, which oversees resources like memory, processes, and devices.

The 4.4BSD Operating System was designed with a number of goals in mind, most notably a well-defined separation of concerns to facilitate easy comprehension and alteration. By incorporating fast algorithms, effective data structures, caching, and buffering strategies, it was also crafted to be performance-oriented and scalable. Furthermore, it was constructed to be portable, allowing it to function on a variety of hardware platforms and permitting its broad distribution.

The Conceptual Architecture of the FreeBSD Kernel report is a critical source of information for those seeking a greater understanding of the FreeBSD operating system. This scholarly examination breaks down the system into its components and explores how they interact with one another. In this way, it is able to provide an insightful analysis of the system's functionality, architecture, control and data flow, concurrency, and the responsibilities of the participating developers.

At the foundation of this study are the kernel, system calls, file systems, and networking parts, which are thoroughly investigated with regard to their data processing and alteration. Furthermore, state and sequence diagrams are employed to illustrate the various use cases and how they operate within the system.

Ultimately, the comprehensive examination of the FreeBSD operating system's architecture presented in The Conceptual Architecture of the FreeBSD Kernel report makes it an indispensable resource for those interested in this open-source project.

Our process for deriving the conceptual architecture of the FreeBSD kernel included:

- Reading the official FreeBSD design and implementation document.
- Reading the conceptual design document of similar systems such as Linux.
- Studying the architecture and design of monolithic operating systems.

Architecture

In the field of Operating Systems, the kernel is a minute segment of the system that furnishes only the elemental amenities requisite for the execution of supplemental operating system characteristics. There are distinct schools of thought with regard to the design of an operating system's kernel such as the monolithic kernel architecture, microkernel architecture, hybrid kernel architecture, and various others contingent on the usage scenario and functional prerequisite of the operating system. The organization of the FreeBSD kernel is monolithic with a modular design. Parts such as drivers are designed as modules which can be loaded and unloaded at any time by the user. In addition, an operating system constructed with a monolithic kernel as its base has the complete operating system functioning in the kernel space. The monolithic architecture was chosen for its simplicity and configuration.

The FreeBSD kernel altogether provides four services:

1. Processes: approaches for generating, concluding, and governing processes.
2. A filesystem: a set of named files, organized in a tree-structure hierarchy of directories, and operations to modify them.
3. Communications: secure single direction streams between affiliated processes (pipes), notification of irregular events (signals), a broad interprocess-communication facility, and a general networking framework
4. System startup: activation and operational issues.

The FreeBSD kernel can be analyzed by its dynamic functioning and viewed as a service purveyor to user processes grouped as per the amenities provided to its users. These services are accessed by user processes through system calls.

Further still, the FreeBSD kernel can be divided into two parts: the first, a more comprehensive portion, consists of machine-agnostic elements which provide system services which applications can access by means of system calls. The majority of the software within this portion is transferable across various hardware architectures. This segment is structured into:

- Basic kernel functions
- Memory management support
- Generic system interfaces
- The file system
- Terminal-handling support
- Interprocess-communication facilities
- Network communication support

The second, a smaller section, is split from the main code and holds platform-specific components. When an action which is dependent on the architecture is called for, the platform-independent part of the kernel summons an architecture-particular function that resides in the machine-dependent part of the kernel. This part is organized as:

- low-level system-startup actions
- trap and fault handling

- low-level manipulation of the run-time context of a process
- configuration and initiation of hardware devices
- run-time support for I/O devices

The Larger Part of the FreeBSD Kernel (Machine Independent)

1. **Basic Kernel Facilities**
   The kernel furnishes basic capabilities such as timer and system lock management. Timers are employed to arrange the execution of a function at a pre-determined time in the future. System lock handling delivers multiple different synchronization primitives, granting programmers to safely access and manipulate numerous data types.

   Furthermore, the kernel provides descriptor oversight. Most processes anticipate 3 descriptors to be present once they launch execution. These descriptors are 0, 1, and 2 and are frequently referred to as standard input, standard output, and standard error, respectively. These 3 descriptors are connected with a user's terminal by the login procedure and are bequeathed through fork and exec by processes triggered by the user. An application can then gather what the user inputs by scrutinizing standard input, and the application can direct output to the user's display by writing to standard output or utilize the standard error for writing error output. Utilizing the pipe and filter architectural style, pipes allow the result of one program to be input into another program without rewriting or even relinking either program. To illustrate, in place of descriptor 1 of the source program being set up to write to the terminal, it is set up to be the input descriptor of a pipe. Comparatively, descriptor 0 of the sink program is set up to refer to the output of the pipe instead of the terminal keyboard. The combined set of two processes and the correlating pipe is known as a pipeline. Pipelines can be of an arbitrary length series of processes connected by pipes.

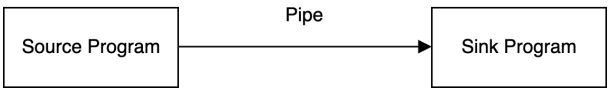

Figure 1: Pipes and Filter Architecture Style

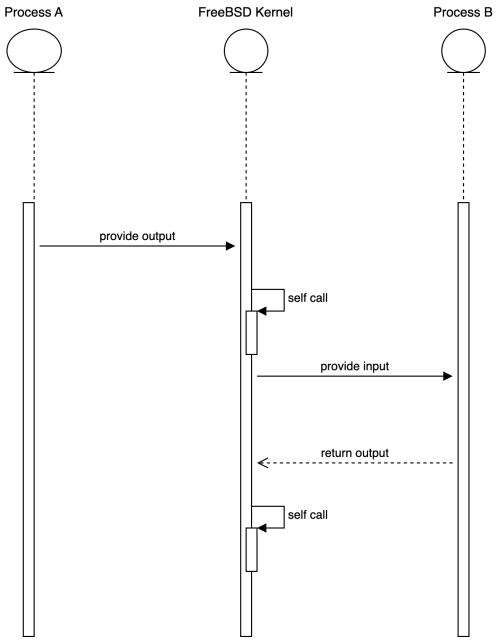

Figure 2: Sequence diagram for inter-process communication (pipes and filter architectural style)

Process governance is a fundamental feature which the kernel supplies. End users are able to initiate processes, manage the processes' operation, and be apprised when the process's operational status modifies. Every process is given a process identifier (PID) which is used to inform a user of status changes, and by a user when addressing a process in a system call. Processes summon the fork system call to form new processes by causing the kernel to replicate the environment of the original process. A process may likewise substitute itself with the data set of another program providing to the freshly formed image a set of arguments, using the system call execve. A process may terminate using the exit system call.

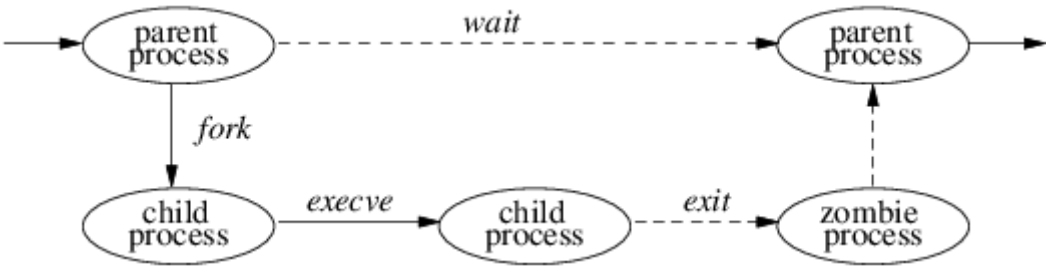

Figure 2: Process life cycle as illustrated in the FreeBSD design document

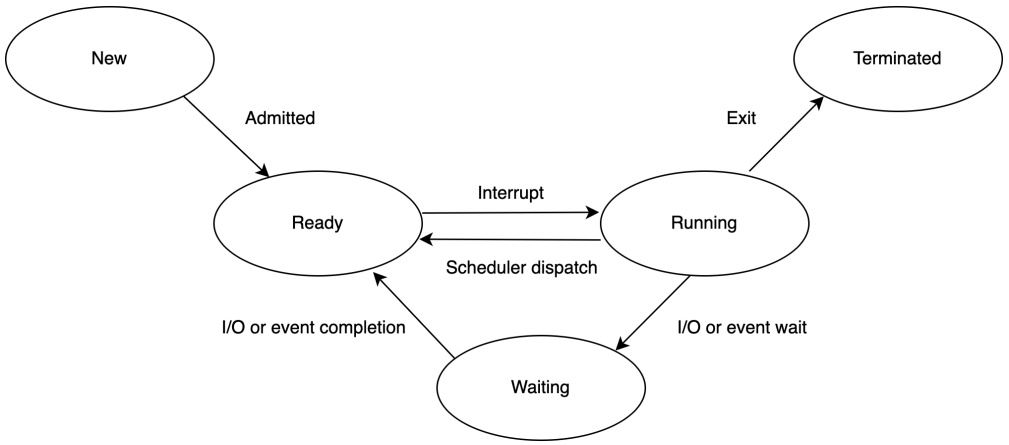

Figure 3: State diagram for context switching by the process scheduler

2. Memory Management Support

In a typical operating system kernel, the allocation of memory for the duration of a single system call often presents a performance issue. This is due to the fact that such memory must be allocated on the run-time stack of a user process, yet this stack is of limited size, making it impractical to allot moderate-sized memory blocks on it. To address this, the FreeBSD kernel incorporates a general memory allocator, a dynamic device that can be utilized by any element of the system to alleviate the intricacies of coding within the kernel.

3. Generic system interfaces

The kernel furnishes a generic interface for communication with an assortment of file systems and apparatus utilizing file identifiers. File identifiers are integers that are supplied to user-space programs and utilized to refer to the related kernel-mode structure. File identifiers can have assorted types, they can allude to files on a file system, the input/output flows to/from a terminal, conduits to different processes, printers, tape drives, network sockets, and so forth.

There are numerous ways to acquire a file identifier, regularly when a program starts, it has two yield file identifiers, stdout and stderr to the terminal, and one input, stdin, which peruses from the terminal. At the point when the parent process begins the child process, it can change these identifiers to allude to different things, for example, a pipe to another process or to a file. Processes can likewise utilize various system calls, for example, open-to-open file identifiers that allude to files on the file system or exceptional files that allude to things, for example, printers or terminals. Processes can likewise utilize the socket system call to make a file identifier that is utilized to communicate both over the network or between processes.

Contingent upon the type of file identifier, programs can peruse and compose different sorts of information. For example, in the event of a typical file, the program can peruse or compose a stream of bytes to the file identifier. Be that as it may, for specific kinds of network sockets (such as UDP sockets), processes would compose individual bundles at once, instead of a continuous stream of bytes.

The program additionally exhibits the capacity to manipulate the file descriptor in diverse manners; for instance, a file may be traversed by seeking to its central or terminal regions, for the purpose of reading or writing information present there. Conversely, for a network socket, this is impossible, as attempting to hop forward or backward in a real-time data stream would be illogical. Various other kinds of objects have their own type of control, including the ability to utilize ioctl on a terminal file descriptor in order to adjust options, such as the modem's settings (for archaic terminals), echoing of input, and the dealing with of exceptional key sequences, etc. The potential for a given file descriptor to be adjusted depends on the capabilities of the file descriptor itself.

4. **The filesystem**

The filesystem is a core component of the kernel, providing processes with space to store data. Regular files on the filesystem are simply sequences of bytes, how they are used is determined by the programs that run on the operating system. The filesystem organizes these files into a tree structure, starting at the root directory, which has the path /. Each directory contains various entries, which are a name and some associated content, such as a pointer to a regular file or another directory. Directories can be recursively nested, creating the tree structure of the file system.

Absolute paths to files and directories start with / and are followed by the name of the directory entries separated by /'s, then finally the name of the file. Relative paths are similar, however, do not begin with the '/' character. These paths represent the directory entries to follow in order to reach the target file starting from the process' *working directory* rather than the root directory.

For example, in the root directory, there could be a file named "test" which would have the path /test. There could also be a subdirectory named "**foo**", which then could have a file named "bar", and it would have a path **/foo/bar**. If a process had its working directory in **/foo**, it could refer to the file as simply **bar**. If a process had its working directory in **/**, it could refer to the file as either **foo/bar** or **/foo/bar**.

Every directory entry also has an associated owner and group, as well as permissions which determine which users are allowed to interact with the directory entry and in what ways they are allowed to interact with the directory entry. Permissions are set up to control the permissions of the user who owns the file, permissions of the group who owns the file, and permissions of all other users. The permissions for each group control whether they can read, write, and execute the file. In the case of directories, to read a directory is to list the files within the directory, to write a directory is to create a file within the directory, and to execute the directory is to set the process' working directory to that directory. To access files within a directory also requires the execute permission.

The file system available to processes is not a “real” file system on a disk, but rather a virtual file system. This virtual file system allows multiple file systems to be accessible as if it were a single file system. There is a real file system mounted as the root of the filesystem, then additional file systems can be mounted at various locations in the virtual file system. For example, if the user inserted a second disk, the user could then instruct the kernel to mount the second disk at a specific path, such as **/mnt**, then all files within the second disk would appear to all processes on the system as if they were in **/mnt**. The virtual file system also abstracts away the underlying details of how the file system works and allows processes to access any type of underlying file system with the same interface.

5. **Terminal-handling support**
   The FreeBSD kernel's terminal handling feature provides support for terminal devices, which are used to communicate with the operating system. These **terminal device drivers** that are at the lowest level are controlled using input and output processing by terminal handling. The terminal driver processes the input from terminal devices into formats (formatting services can be disabled individually) that can be utilized by applications. The terminal driver then processes the output to terminal devices in order for it to be displayed on the terminal. Some Unix-like systems including FreeBSD contain a layer above the terminal device drivers called **line disciplines** that provide various degrees of character processing and to implement behaviour common to terminal devices. The terminal can also be configured in numerous ways to meet specific needs.

6. **Interprocess-communication facilities**
   Interprocess communication can be organized into distinct domains in which communication takes place between endpoints known as sockets, generated when a system call to the socket is executed and a file descriptor is subsequently returned. Each socket is associated with a communication protocol and is characterized by a type which defines the communicative semantics, including reliability, sequence, and the prevention of message duplication. The primary intention of the socket interface was for programs to operate without alteration on stream-type links. These applications can function only if the read-and-write system calls remain unaltered. Thus, the foundational interfaces were maintained to work on stream-type sockets, with a separate interface developed for more complex sockets, necessitating the inclusion of a destination address in every send call. Furthermore, the novel interface is portable across a variety of platforms. In addition, other local IPC techniques unrelated to networking, including semaphores, message queues, and shared memory are also available.

7. **Support for network communication**
   Network communication in FreeBSD provides access to **network protocols** necessary for communication over a network. These protocols are implemented in the kernel as a separate layer of software that sits logically below the socket software. The kernel offers numerous auxiliary services for using a variety of different network protocols. This flexibility in the architecture is a crucial part of the overall system design because it enables the system to interact with a wide range of other systems and to support a broad range of network-based functionality.

The Smaller Part of the FreeBSD Kernel (Machine Dependent)

1. **Low-level system-startup actions**
   Upon initialization, the kernel carries out an array of primary processes to establish the rudimentary conditions necessary for the machine to function, including hardware initiation, memory management organization, and the loading of applicable device drivers and system utilities

2. **Trap and fault handling**

Trap and fault handling is a very low-level aspect of the kernel and its implementation varies wildly between architectures. Although they remain similar, the exact definition of terms such as “trap”, “fault”, “exception”, and “interrupt”, vary between architectures. At a high level, when certain events related to the current code occur, known as synchronous interrupts, such as accessing illegal memory addresses or arithmetic errors (such as division by zero), the CPU can interrupt the current code execution and call a handler function in the kernel to deal with the event. Additionally, there are also asynchronous interrupts where the events unrelated from the current code occur, such as external hardware like a keyboard or incoming network packets, it also interrupts the current code and calls a handler function. The kernel has various handler functions that decide what to do, for errors such as arithmetic errors, the kernel will then signal the userspace process and let the process decide what to do. In the case of hardware events, such as network packets, the relevant hardware driver will be called to handle the interrupt. This mechanism is also used to implement system calls. Since processes run in unprivileged user mode, the process can intentionally trigger an interrupt (most architectures have dedicated interrupts for exactly this purpose), which switches execution back into kernel mode so that the kernel can process the system call.

3. **Low-level manipulation of the run-time context of a process**
   The FreeBSD kernel is equipped with the ability to provide the appearance of simultaneous execution of multiple programs or processes via context switching, an operation depending on the underlying hardware facilities. To accomplish this, certain hardware architectures feature instructions designed to save and restore the hardware execution environment, including the virtual address space. Others must rely on software to retrieve the hardware state from registers and store it, then load these with the new state. Ultimately, regardless of the architecture, the software context of the kernel must be conserved and renewed. As context switching is a frequent occurrence, expediting it allows for less time to be spent in the kernel and more for the execution of user-level applications. Thus, reducing the amount of information needed for the context switch provides a feasible means of achieving quicker context switches.

4. **Configuration and initialization of hardware devices**
   This necessitates the identification of hardware, the arrangement of hardware resources, the initiation of hardware, the incorporation of device drivers, and the configuration of interrupt handling.

5. **Run-time support for I/O devices**
   An infrastructure for the management of system devices known as newbus is employed.

Diagrams

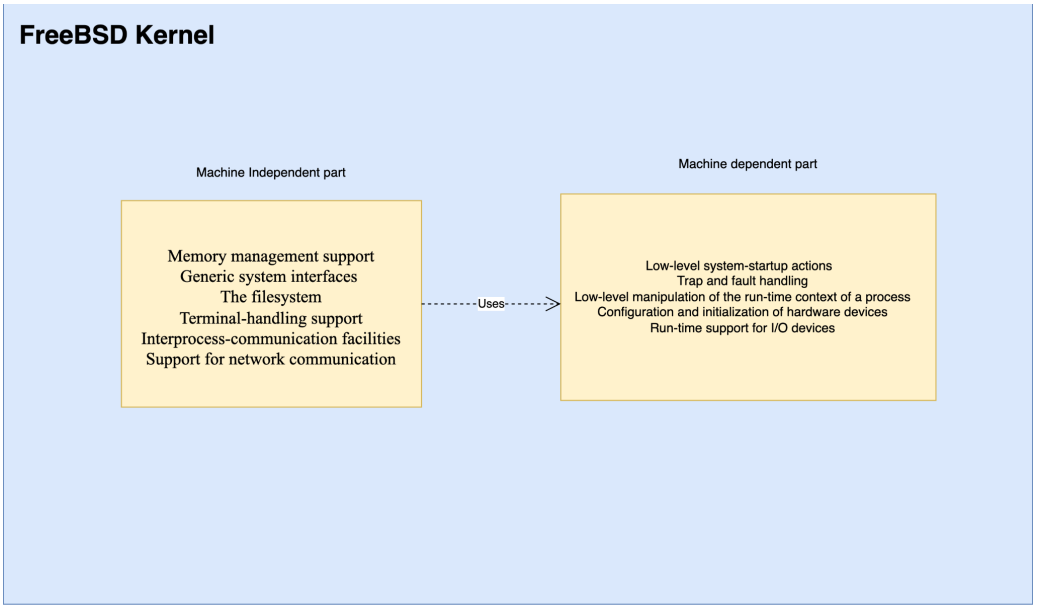

Figure 4: The 2 parts of the FreeBSD kernel

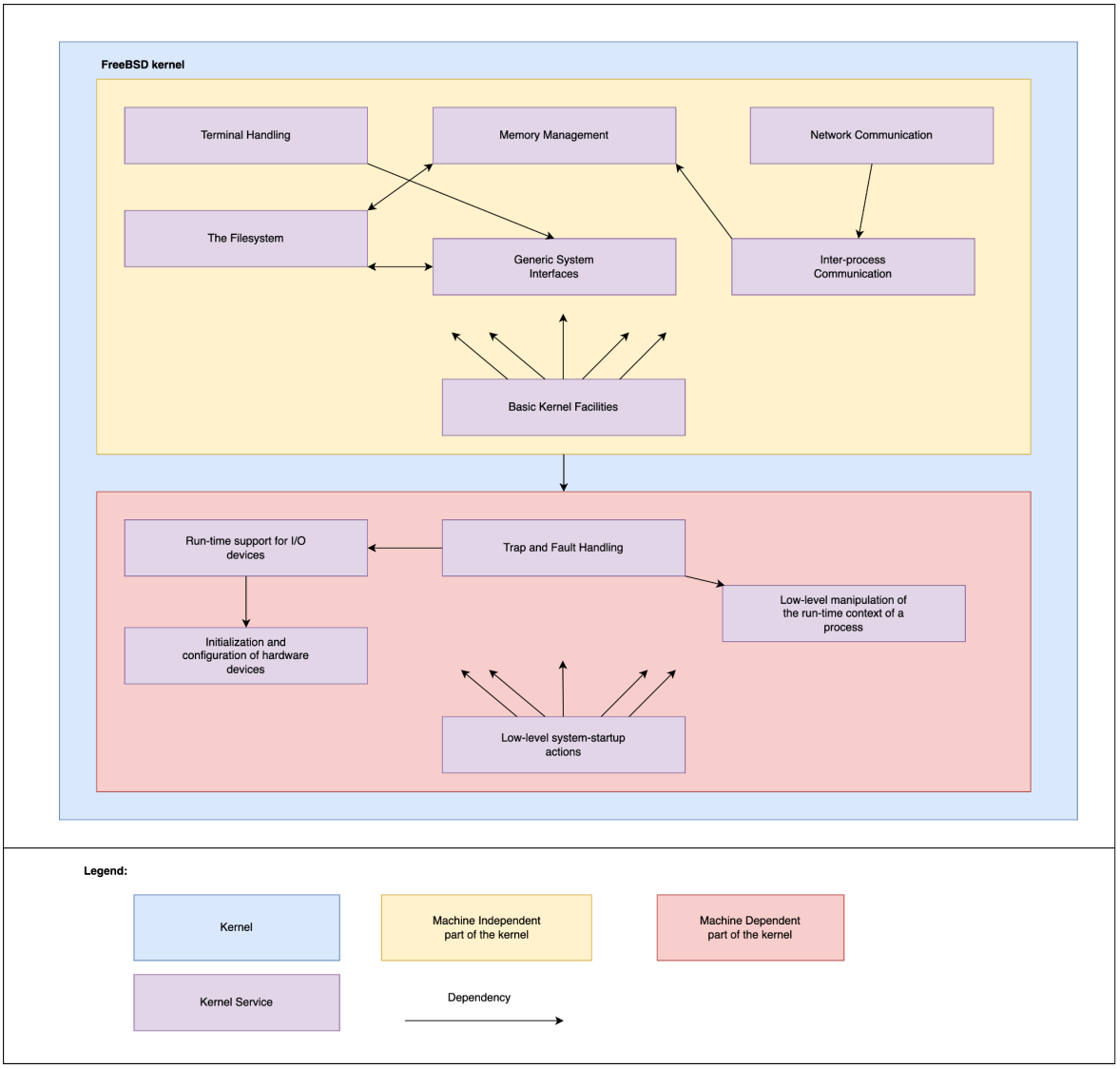

Figure 5: FreeBSD Kernel Conceptual Architecture

External Interfaces

A machine wouldn't be able to function if there's no proper way for a user to control or input commands and thus it needs a controller. In this sense, the external interfaces serve as a bridge between the users and the system, a means for users to properly feed new inputs into the machine. Since the system has to perform various tasks ranging in complexity, the external interfaces are also designed as a combination of different physical and virtual components to serve their purposes.

System Call Interface

An important layer between the user-level and the kernel-level code as all commands or services requested via applications go through system calls to reach the kernel. System calls don't appear as physical commands or applications to the users but rather as hidden signifiers known as hardware "traps" for the system to recognize new inputs, validate them and execute at a lower level. Since its function is associated as the bridge between users and all kernel-level services, its capability ranges from handling simple tasks such as returning the current time of the day or complicated operations such as writing data to secondary storage. This interface is the primary means for applications to access resources and services provided by the kernel, which includes file system access, process management, and device driver services.

Device Driver Interface

FreeBSD provides an interface for accessing and controlling hardware devices through device drivers. Each of these said devices has its own character-device interfaces, which are accessible through the filesystem namespace. Within the filesystem, they are treated as special files and thus are granted their own section beside the virtual filesystem. A device driver is a bridge between the kernel and hardware devices such as network interface cards, keyboard, mouse and display devices. By transferring and even helping categorize input data received from users, it is an important means of communication for the kernel to gain access and control of a wide range of hardware devices.

Network Interface

One of the most commonly used services at the user level is file sharing among computers and it could only be done with the aid of networking devices. Like other external devices, networking devices also have their own device drivers. Constructed in combination with the socket interface and network protocols, the network interface is responsible for converting data into packets and either transmitting or receiving them on some devices such as the Ethernet hardware, a commonly used driver with the FreeBSD kernel.

Virtual Memory

An essential part of a system that enables effective memory resource management is virtual memory. Virtual memory is used to map and temporarily store dormant or unused software on a hard drive while the real *RAM* is occupied by running applications. This helps to free up physical *RAM* so that space can be made for a few additional new apps and makes sure the system keeps running properly. The system can manage its memory resources dynamically by employing virtual memory, ensuring that each application has access to the memory it requires to function properly. The virtual memory interface guarantees that applications have access to the memory they require to function properly and plays a critical role in the stability and performance of the system.

Virtual-Filesystem interface

At the user level, we may find it easy to look for files through simple search operations in the *filesystem*, however, since a system would consist of numerous files of different types, a data structure that categorizes each of these different files and constructs paths to them are necessary to have. Below this generalized interface, within the kernel layer is a subsystem called the virtual-node layer. Designed as an object-oriented interface that contains all info of files such as reference points, attribute flags, type of underlying objects, associated field names, etc, this layer is responsible for forming and returning all possible *pathnames* to certain files once the kernel receives lookup requests from users.

Use Cases

Process Management and Inter-Process Communication

Process management capabilities provided by the kernel are essential to use of the system. One use case is in a multi-tasking scenario, where multiple processes are running simultaneously. In this use case, the process management subsystem is responsible for ensuring each process receives an appropriate share of CPU time and memory. To achieve this, the process management subsystem communicates with the memory management subsystem to allocate and de-allocate memory, based on priority. FreeBSD's process management subsystem also gives Inter-Process Communication support, allowing each running process to communicate. Additionally, processes that require access to files and/or the network, in which the process management subsystem will communicate with the file system and networking subsystem respectively, to handle process permissions and access.

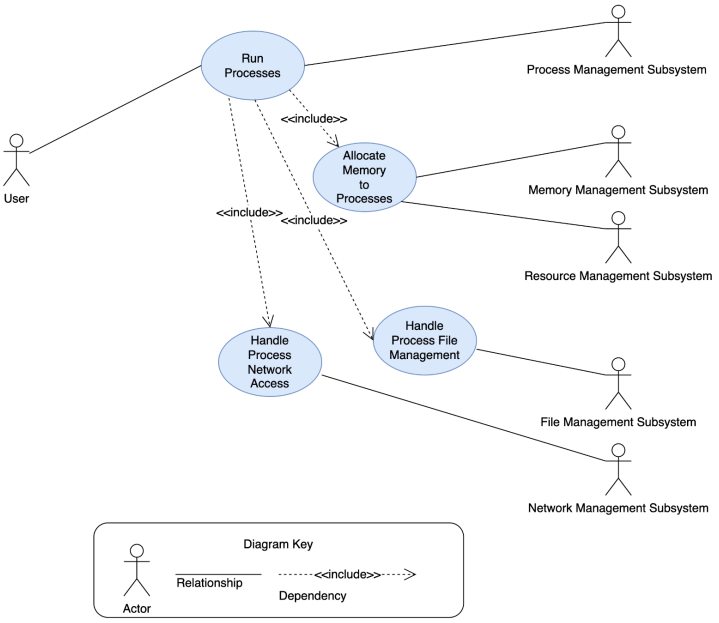

Figure 6: Use Case Diagram for process management in the FreeBSD Kernel

File Storage

FreeBSD is often used as a storage server because of its stability, security, scalability, and performance on very large filesystems. The FreeBSD kernel ships with the Zettabyte Filesystem (ZFS), a non-overwriting filesystem. Its design was intended for the use case of very large filesystems, as well as strong data integrity, with cheap snapshots and clones, storage pooling (including across multiple disks), RAID, and zettabyte scale storage pool support. There is also support for other file systems such as UFS and the fast filesystem. Paired with the network file systems NFS and SMB, and FreeBSD's support for various hardware platforms, FreeBSD can be set up as a scalable storage system.

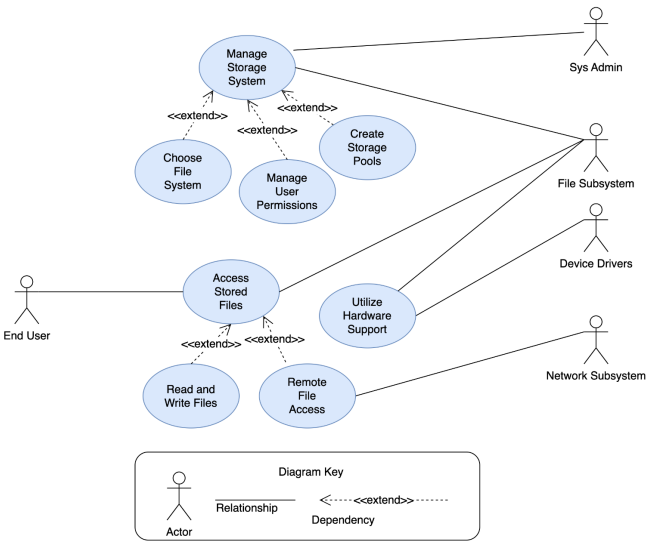

Figure 7: Use Case Diagram for File Storage

Security

FreeBSD's security is built into its design and architecture, and it provides a number of features and mechanisms to help secure systems running the operating system. Some important use cases from the perspective of a system administrator and user are shown in the diagram on the right. Other use cases include the extensibility for users and developers to implement their own security solutions on the OS, such as VPNs, storage encryption, and other security features, built on top of FreeBSD's designed cryptography framework.

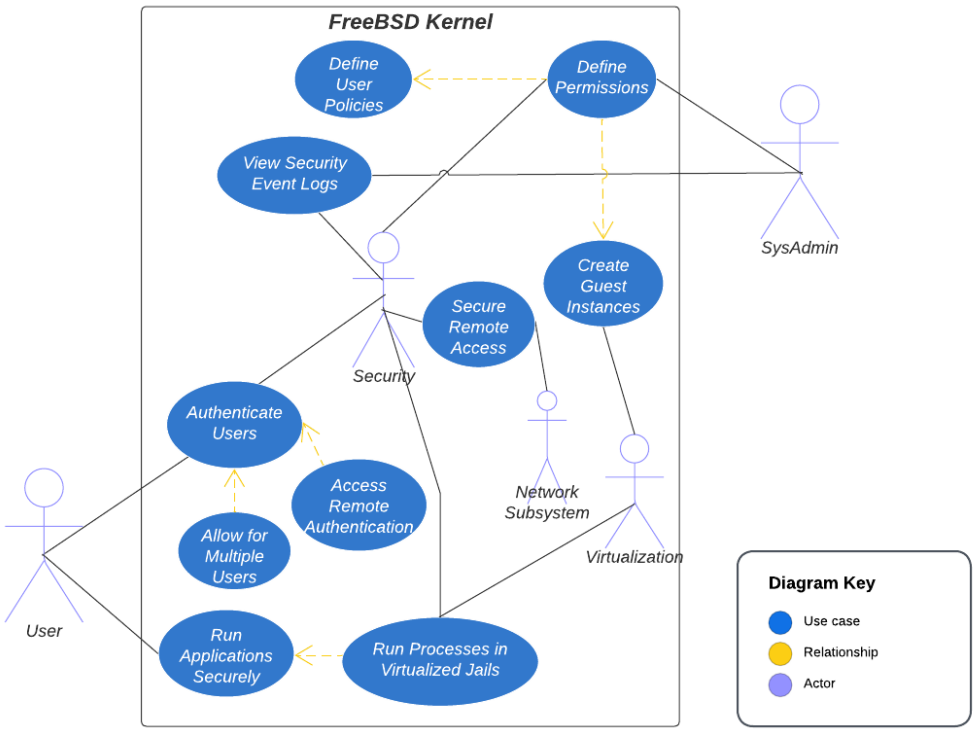

Figure 8: Use case diagram for Security

Data Dictionary

Term	Definition	Type
Monolithic	A monolithic is a single, large application that performs all essential operating system operations, including memory management, device driver management, and process management.	
Microkernel	A microkernel is a type of operating system architecture that uses a small, minimalistic kernel to provide only the basic services required to run an operating system.	
Hybrid Kernel	A hybrid kernel is an operating system architecture that combines features of both monolithic kernels and microkernels.	
Modular	Modular is often used to describe a design philosophy for the operating system's components.	
Signals	Signals are a mechanism used to stop a process from running and inform it of an event.	
Processes	A process is a unit of execution or a program that has its own memory space, file descriptors, and system resources.	

Pipes	Pipes allow two or more processes to communicate with each other by passing data through a common channel.	
system calls	System calls are the primary way for processes to communicate with the operating system and request resources such as memory allocation, file I/O, and inter-process communication.	
Pipeline	A pipeline is a shell feature that allows the output of one command to be sent as the input to another.	
Standard error	It's a predefined file descriptor that handles error messages and sends them to the user. *stderr*.	
Standard input	It's a predefined file descriptor associated with the keyboard, *stdin* it reads input from the user.	
Standard output	It's a predefined file descriptor associated with the terminal screen, *stdout* it sends output to the user.	
Terminal	Terminal provides a text-based interface for a user to enter commands and interact with the system.	
Exec	Is used to replace the current process image with a new process image.	
Fork	It creates a new process by duplicating the calling process. The new process is called the child process, and the calling process is called the parent process.	
Execve	It is used to replace the current process image with a new process image.	
Process identifier (PID)	Is a unique numerical identifier assigned to each process in an operating system.	int
Exit	Is a system call that is used to terminate a process.	
Socket	A socket is an endpoint for communication between processes. Processes can communicate with one another through sockets, either inside the same system or among various systems connected by a network.	
Descriptor	A descriptor is a data structure that is used to represent an open file, a network socket, or other similar objects.	
IPC	Inter-Process Communication, and it refers to the mechanisms that allow processes to communicate with each other within an operating system.	
Concurrent execution	The ability of an operating system to run multiple processes or threads at the same time.	
Network protocols	Network protocols are sets of rules that control communication between network devices.	

Naming Conventions

The part of the system that runs in protected mode and mediates access by all user programs to the underlying hardware and software constructs is called the kernel. Each task/thread being executed is called a process. Child-process is the name given to a process the kernel created by duplicating the context of another process. Every process is assigned a unique value used by the kernel to identify a process when reporting status changes to a user,

and by a user when referencing a process in a system call that is called process identifier (PID). The set of named files, organised in a tree-structured hierarchy of directories, and of operations to manipulate them is appropriately named the filesystem. System calls is the name given to functions to access the filesystem and communication facilities and that allow applications to request services from the kernel. The command-language interpreter to enter system calls to interact with the kernel is called the shell. System calls are called synchronous because the application does not run while the kernel does the actions associated with a system call. The name of the kernel's memory-management support processes is paging (when a reference to a part of a process's address space that is not resident in main memory is brought into memory) and swapping (moving the entire context of a process to secondary storage should there be a severe resource shortage). The kernel's interprocess-communication facilities are called sockets (endpoint of communication referred to by a descriptor). The system can create requests called signals for the processor to interrupt currently executing code, so that the event can be processed in a timely manner. A process may specify a user-level subroutine called a handler to specify where a signal should be delivered. When processes are terminated, a file called a core file is created that contains the current memory image of the process for use in postmortem debugging. To control access to terminals and to provide a means of distributing signals to collections of related processes, processes are organised into sections called process groups or jobs which can be further grouped together into subsections called sessions. By changing the process-group identifier of the terminal, a shell can arbitrate a terminal among several different jobs which is called job control. Each process has its own private address space which is divided into three logical segments called text (read-only and contains the machine instructions of a program), data (contains the initialized and uninitialized data portions of a program), and stack (holds the application's run-time stack). A page that is protected against being written by being made read-only is called a copy-on-write page. The basic model of the UNIX I/O system is a sequence of bytes called I/O stream that can be accessed either randomly or sequentially. Processes use small unsigned integers obtained from the open and socket system calls named descriptors to reference I/O streams. A linear array of bytes with at least one name is called a file and a linear array of bytes used solely as an I/O stream and is unidirectional is called a pipe. Pipes allow the output of one program to be input to another program without rewriting or even relinking of either program. The resulting set of two processes and the connecting pipe is known as a pipeline. The kernel uses a table called a descriptor table to translate the external representation of a descriptor into an internal representation. Most processes expect three descriptors to be open already when they start running which are numbered 0, 1, 2 and are called standard input, standard output, and standard error respectively. Mapping descriptors to objects other than the terminal is called I/O redirection. Structured hardware devices (have a block structure) are called block devices while unstructured hardware devices (do not have a block structure) are called character devices. Unstructured files are called character devices. Virtual nodes in the kernel are called vnodes. A directory is a type of file, but, in contrast to regular files, a directory has a structure imposed on it by the system. A process can read a directory as it would an ordinary file, but only the kernel is permitted to modify a directory. The information in a directory about a file is called a directory entry and includes, in addition to the filename, a pointer to the file itself. A process's root directory is the topmost point in the filesystem that the process can access; it is ordinarily set to the root directory of the entire filesystem. 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. Files may have void areas in the linear extent of the file where data have never been written called holes. The remote machine that exports one or more of its filesystems is called the server and the local machine that imports those filesystems is called the client. Screen editors and programs that communicate with other computers generally run with input being passed through to the reading process immediately and without interpretation. All special-character input processing is disabled, no erase or other line editing processing is done, and all characters are passed to the program that is reading from the terminal. This is called non canonical mode (also commonly referred to as raw mode or character-at-a-time mode).

Conclusions

In conclusion, the FreeBSD kernel is a powerful and versatile system that provides a wide range of features and capabilities. It is highly modular and extensible, allowing for customizations and additions to be made to the kernel. Additionally, it is well-documented and supported, making it an ideal choice for developers and users alike. The

FreeBSD kernel is a great example of how an open-source system can be used to create a powerful and reliable operating system. Going forward, it is important to continue to improve the FreeBSD kernel by adding new features and capabilities, as well as improving the existing ones. Additionally, it is important to ensure that the kernel remains secure and stable so that users can continue to rely on it for their computing needs. Our proposal for the future would be to explore in great detail one of the services the kernel provides, namely, the filesystem and its concrete implementation.

Lessons Learned

If given the chance to design our own kernel, We would have opted for the microkernel architecture as microkernels are **more secure** than monolithic kernels because the operating system is unchanged if a service fails in a microkernel. On the other hand, if a service fails in a monolithic kernel, the entire system fails.

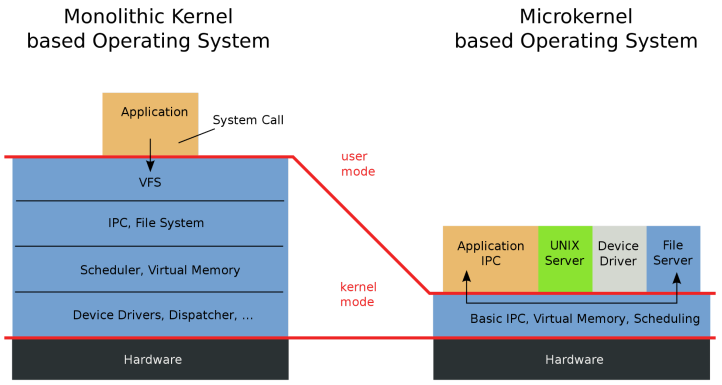

Figure 6: Monolithic kernel architecture vs Microkernel architecture

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.

McKusick, M. K., Neville-Neil, G. V., & M., W. R. N. (2015). Chapter 2. Design Overview of FreeBSD. In *The design and implementation of the freebsd operating system* (pp. 44–83). essay, Addison-Wesley

Wikipedia contributors. (2020, August 30). FreeBSD. In Wikipedia, The Free Encyclopedia. Retrieved 22:17, August 30, 2020, from https://en.wikipedia.org/w/index.php?title=FreeBSD&oldid=974477862

Wikipedia contributors. (2020, August 25). Kernel (operating system). In Wikipedia, The Free Encyclopedia. Retrieved 22:17, August 30, 2020, from https://en.wikipedia.org/w/index.php?title=Kernel_(operating_system)&oldid=974437862

Wikipedia contributors. (2020, August 30). Monolithic kernel. In Wikipedia, The Free Encyclopedia. Retrieved 22:17, August 30, 2020, from https://en.wikipedia.org/w/index.php?title=Monolithic_kernel&oldid=974478556

Bowman, I. (1998). Conceptual architecture of the linux kernel. *URL: http://plg. uwaterloo. ca/itbowman/CS746G/a1*.