FreeBSD:
Conceptual Architecture

EECS 4314: Advanced Software Engineering | Assignment 1

February 8th, 2023

Authors: **Pink Fluffy Unicorns**

Benedict Miguel	216237364	benedictmiguel106@gmail.com
Jiachen Wang	215453988	reimsw98@my.yorku.ca
Keshav Gainda	216891319	gaindakeshav@gmail.com
Ketan Bhandari	217550765	ketan11@my.yorku.ca
Mohaimen Hassan	216187809	hassan08@my.yorku.ca
Pegah Fallah	216262081	fpegah@my.yorku.ca
Shami Sharma	217262510	shami012@my.yorku.ca
Sidharth Sudarsan	216697120	lensman@my.yorku.ca
Suryam Thaker	217281031	suryam@my.yorku.ca
Xiaochuan Wang	214107106	wangx997@my.yorku.ca

Table of Contents

Abstract **2**

Introduction and Overview **3**

Architecture **3**

- Reference Architecture for an Operating System 3
- Overall Structure 4
- Kernel Subsystems 5
  - Process Management 5
  - File Management 6
  - I/O Device Management 6
  - Memory Management 7
  - Security 7
  - Network Management 8
  - Interprocess Communication (IPC) 9
- Interfaces provided by the OS 9
- Concurrency 10

External Interfaces **10**

- Graphical User Interface 10
- Command Line Interface 10

Design Patterns **11**

- Interactions 12

Use Cases **13**

Data Dictionary **14**

Conclusions **15**

Lessons Learned **15**

References **15**

Abstract

This report offers a high-level view of the architecture of FreeBSD, its components, and interactions among them. The architecture of the system is described in terms of its goals, requirements, testability, and support for future changes. FreeBSD has a monolithic kernel with dynamically loadable modules, but implements a layered approach for communication between the user space and the kernel. The kernel provides basic system facilities, memory management, generic system interfaces, file system, terminal-handling support, interprocess communication facilities, and network communication support. The goal of FreeBSD is to provide a stable, secure, and high-performance operating system that is compatible with multiple hardware platforms, while maintaining compatibility with other Unix-based systems. FreeBSD's process management subsystem is crucial for the functioning of the operating system. Various tools such as command line utilities and system calls allow for process management. Interprocess communication within FreeBSD can be established using pipes, sockets, or files. For the file management subsystem, FreeBSD uses Unix File System (UFS), which supplies a graphical method for file management. I/O device management is handled by the kernel, and it includes advanced features like the Complete Fair Queueing (CFQ) scheduler. Memory management is ordered hierarchically with different architectures being used for different types of memory storage. The security subsystem implements process credentials, a privilege model, Mandatory Access Control, cryptography, and a capability model to ensure only the permitted user can access the assigned data. Next, this report finds that processes, threads, and asynchronous I/O are the main units of concurrency within the system, and that global control flow is achieved through a combination of system calls, interrupts, and context switching. In terms of system interactions, the memory management subsystem allocates processes to access the main memory, the I/O subsystem handles read requests and transfers data from the disk to a buffer in memory, and the security subsystem plays a critical role in system events such as file changes. The report also includes a discussion of the external interfaces used by the system. Finally, the report includes diagrams of essential use cases; file reading and firewall trigger and concludes with the lessons learned.

Introduction and Overview

FreeBSD is an open source Unix-like operating system initially developed using the Berkeley Software Distribution and initially released in November 1993. It is widely used today as an operating system for servers, desktops and embedded platforms.

This documentation describes the high level architecture of the FreeBSD operating system, from the layered architecture of the operating system, to the monolithic architecture of its kernel. This documentation also provides insights into the kernel's subsystems ranging but not limited to its process management, file management, I/O device management, memory management, etc. The documentation also includes any design patterns within the operating system, the interfaces provided by the OS, the interactions of the different subsystems, and the control flow of the operating system. Finally, the documentation highlights the use cases of some operations and subsystems and some of the external interfaces provided by FreeBSD.

Architecture

Reference Architecture for an Operating System

An operating system is a low-level software system that implements an interface between the user of a computer and its underlying hardware to provide an environment where programs may be executed. The kernel is the central and most crucial component of the operating system. Its responsibilities include managing the system's resources, i.e. providing necessary I/O, processing, memory, file-system and network capabilities to the applications through inter-process communication mechanisms and system calls. A shell is software that provides an interface for users to an OS, typically falling into two categories: command-line (CLI) and graphical (GUI), to launch programs. OS architectures can be categorized into Layered, Monolithic, Microkernel, Distributed and Hybrid classified on the basis of the structure of components. In the context of FreeBSD in this report, the Layered approach for the OS and the Monolithic kernel architecture will be discussed as reference architectures.

Layered Architecture
Components in a layered architecture are divided into layers grouping similar components, with the lowest layer being the hardware and the highest layer being the applications. Each layer only interacts with its immediate neighbors; the top layer to answer requests and the bottom layer to request services. Layering provides a modular and well-defined structure. Each layer is concerned with its own functions and other layers are abstracted from it, simplifying testing and debugging. However, layering is not easy to implement and can be slow due to its nature. Microsoft Windows NT is an example of the layered architecture.

Monolithic Architecture
Monolithic is one of the oldest architectures of operating systems. In monolithic, each component of the operating system is contained within the kernel. All the basic services of the OS like process management, file management, memory management, exception handling, process communication etc. are all present inside the kernel only and are provided through system calls. Monoliths run in a single address space and are easier to implement. However, security and extensibility are limited by the simple structure. Linux is a popular example of a monolithic kernel.

Overall Structure

FreeBSD has a monolithic kernel with dynamically loadable kernel modules (LKM). Although the kernel is monolithic, the OS implements a layered approach for communication between applications in the user space and the kernel via system calls. The FreeBSD kernel provides the basic system facilities; it creates and manages processes and provides functions to access the filesystem and communication facilities. This monolithic kernel provides simplicity and performance to the design and implementation.

The largest part of the kernel implements the system services that applications access through system calls. System calls are the functions that appear to user processes as library subroutines, to provide an interface for the user or the application programs to call upon the services of the kernel. These are generally written in C or C++, while the performance-optimized ones are written in assembly. For developers, the system exposes APIs instead of direct low-level system calls to support program portability between different systems. The kernel services are summarized as follows:

Basic kernel facilities, Memory-management support, Generic system interfaces, The filesystem, Terminal-handling support, Interprocess-communication facilities, Support for network communication

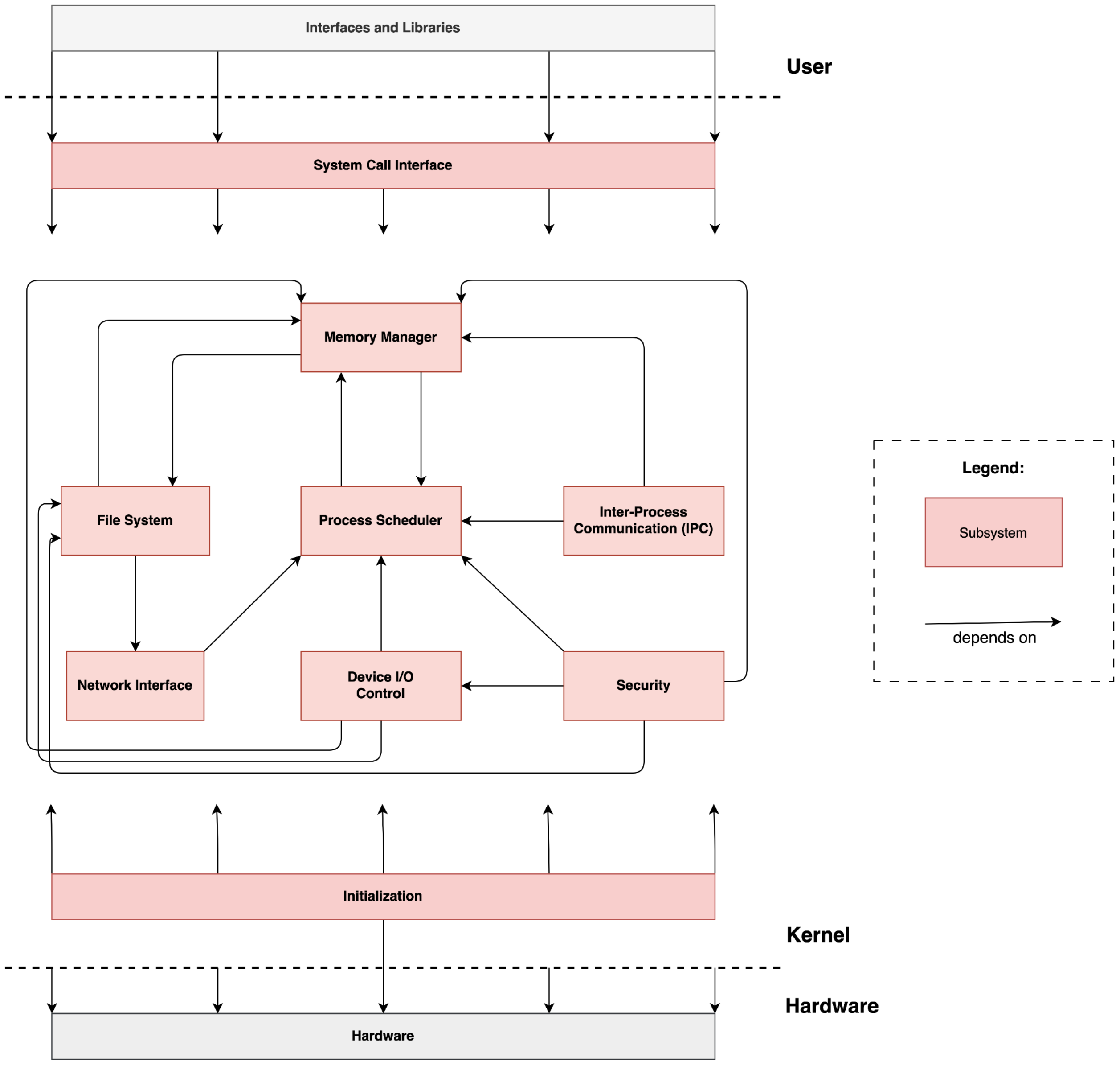


Fig. 1. Overall Conceptual Architecture and Dependency Graph

Kernel Subsystems

Process Management

In FreeBSD, process management plays a crucial role in ensuring the smooth functioning of the operating system. The kernel is responsible for managing processes in FreeBSD, including creating, scheduling, and destroying processes. To determine which processes should be executed, the kernel employs a priority-based algorithm. Processes can be managed in FreeBSD using various tools, such as command line utilities such as ps, top, and kill, as well as system calls provided by the operating system. Additionally, the jail system in FreeBSD provides a process management framework that allows administrators to isolate processes and limit their access to system resources. Some of the benefits of using FreeBSD for process management include stability, security, and advanced networking features.

A process can create a new process that is a copy of the original using the fork system call. The fork call returns twice: once in the parent process, where the return value is the process identifier of the child, and once in the child process, where the return value is 0. The parent-child relationship creates a hierarchical structure among the processes in the system. The new process shares all of its parent's resources, such as file descriptors, signal-handling status, and memory layout. Typically, the new process overlays itself with the memory image of another program using the *execve()* system call and passing parameters, including the name of a file in a recognizable format. A process can terminate by executing an exit system call and sending an 8-bit exit status to its parent. If more than a single byte of information is needed for communication, an interprocess-communication channel can be set up using pipes, sockets, or an intermediate file.

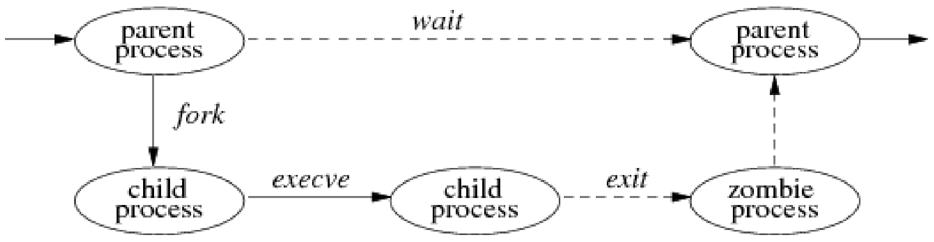


Fig. 4. Process-management system calls

File Management

FreeBSD uses the Unix File System (UFS), which is a family of file systems that is supported by Unix and other Unix-like operating systems, like FreeBSD. The Unix File System stores files in a hierarchical fashion in order to effectively manage files and directories. In order to have a file management system that is both efficient and organized, the data is managed in a tree-like structure, with the root directory being "/" and then the subdirectories branching out from that root directory. Since it uses UFS, the files are able to be manipulated through the standard Unix commands. FreeBSD also provides a graphical method of file management which allows users to manipulate the files in an intuitive and graphical manner. This operating system also includes an advanced file management system, ZFS (Zettabyte File Systems) which comes with a number of advanced file management such as snapshots, check summing and data compressions. On top of this, FreeBSD also supports a variety of other file management systems allowing users to easily access data stored in other devices making it very flexible and an ideal operating system to work with in a multi-device environment.

I/O Device Management

FreeBSD's I/O device management system is one of the most crucial aspects of the operating system as it affects the systems overall performance and reliability. The core of FreeBSD's I/O device management system is the kernel, which acts as an intermediary between the computer's hardware and the operating system. It is responsible for providing an abstract view of the hardware and also handling all the I/O requests. Since it has such a vital responsibility, to make sure this whole process is efficient, the kernel utilizes a variety of data structures and algorithms in order to manage I/O.

The operating system also provides a couple of advanced I/O management features in order to increase the overall performance some of them being the Complete Fair Queueing (CFQ) scheduler and the Deadline scheduler. These two schedulers allow the user to fine-tune the I/O performance based on their specific needs. The Zettabyte File System and the Network file system provide features such as snapshots, data compression etc. Most importantly, FreeBSD's I/O management system is very scalable and is able to handle large amounts of I/O making it very well suited to be used in an enterprise environment with very demanding I/O requirements.

Memory Management

Memory management in BSD is ordered in a hierarchical order. With the main memory residing closest to the CPU and the secondary memory being the next closest. This two-level hierarchy is common, however, for work environments, this two-level hierarchy can be upgraded into a three-level hierarchy with the addition of extra memory components such a network-attached machine or file-server machines. Seeing as BSD is an operating system, page tables are used for swapping between virtual memory addresses and physical memory addresses in addition to a mix of different page swapping algorithms in use.

Processes in memory

Processes in BSD are stored in a data structure that's composed of the parts text, data and stack. The text portion of the data structure is a read-only shareable section that contains the machine instructions of the process. The data and stack sections are readable and writable where the data portion stores the necessary data for the process and the stack stores the current run-time stack. The data portion can be extended via system calls, and the stack growth is controlled by the kernel, while the text can extend only when it is replaced or when the process is being debugged.

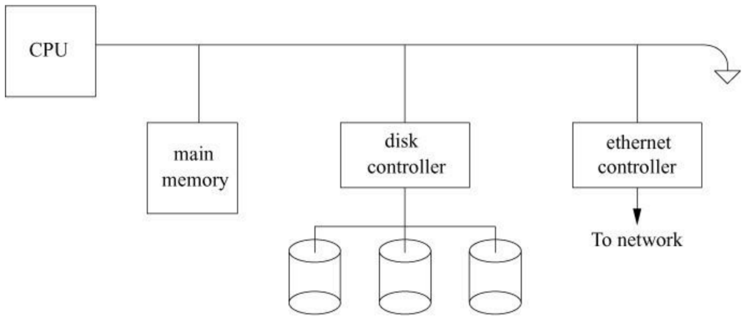


Fig. 2. Memory hierarchy in FreeBSD

Files in memory

The sharing of memory is possible in BSD by allowing pages to be mapped in memory. These pages have protection, privacy, and sharing options available. Processes are able to access these options, allowing control on when it should be shared, when it is private or control the behavior of these pages. Synchronization is done through semaphores that require protection levels prot_read and prot_write to be present.

Conceptual Architecture

Memory management in BSD uses different architectures. The way BSD stores memory depending on what it needs to store can be implied that it uses the Data Abstraction and Object-Oriented Organization. As an example, BSD stores Processes by dividing the memory page based on text, data and stack. While BSD stores shared memory with additional options such as protection, privacy and sharing. The way that memory is allocated can also imply a repository like architecture, where when the system allocates memory it is an independent component that acts on a central data structure, which is the entire memory. Pipe and filter can be implied as a conceptual architecture regarding the design of physical and virtual memory addresses, where the Page Table is the filter and input and output are the physical or virtual memory addresses.

Security

The FreeBSD security model contains a lot of different functions and variables, and all these functions and variables are designed to make sure only the permitted user can access the assigned data.

Process Credentials

Process credential is a bunch of variables associated with the kernel, and these variables contain UNIX user identifiers (UID), group identifiers (GID) ETC. Every time when the kernel receives a process, the kernel will check these credentials before allowing the operation to proceed.

Privilege Model

This model can allow the root user to modify the kernel data or the system-level resources or debug the system process. In order to make sure it is the root user who is using this model, the system will check if the current thread has the root credentials.

Capability Model

This model can make sure the system runs those untrustworthy code to be run in a safe way. By limiting the process that can only access a set of files of its creation time or the delegated. For example, limit the operation of a file.

Mandatory Access Control

Mandatory access control is a set of security policies that help the administrator to control the system, such as information flow. All of these policies will compile in the kernel to help the kernel to make security decisions.

Cryptography

In order to protect the security of the user, FreeBSD has a very strong cryptographic random number generator, these numbers can protect the users data safe, for example use the random number to generate a hash to protect the users password. And the generator oscillator loops to generate difficult-predicted numbers. And this process is based on many cups, so the security of the random number can be guaranteed. Cryptography mainly contains symmetric encryption and asymmetric encryption, and also uses hash during the encrypt.

Network Management

FreeBSD uses the TCP/IP and UDP/IP stack using the layer design, which are physical, data-link, network, transport layer and application layer. However, as a service FreeBSD is using IPV4 for the subnetwork.

IPv4 is at the network layer, which is for the host-to-host connection. When the hosts in the server are communicating with each other, three layers may be involved, first application layer, the host will use I/O device to send information to the kernel, and the internet layer is for the hop-by-hop communication, because all the users are using the same server, third the transport layer, since the connection need to use the router, so the transport layer should be used. After using these three layers, the internet working will be done. There are also some utility tools such as “ifconfig”, “route”, “ipfw”, etc. These tools allow the root users to view and modify the network information.

Interprocess Communication (IPC)

Communication within the Domains

Communication within the domains takes place via sockets. Sockets are an abstract data type that allow endpoint communication by sending and receiving messages, and unlike a file, once it is no longer referenced, the socket becomes free. Each realization of sockets have their own semantics regarding, in-order delivery of data, unduplicated delivery of data, reliable delivery of data, connection-oriented communication, preservation of message boundaries, and support for out-of-bound messages. Since sockets are an abstract data-type, other sockets can be implemented with their own semantics and protocols, it is important to know however, that not all sockets are supported by a domain.

Protocols

To use a socket within a domain, the semantics of the socket is implemented using the appropriate protocol required for communication within the domain. This means that a socket may use a protocol to adhere to communication protocols and that the semantics of a socket is realized through the implementation of the protocol within the socket.

Conceptual Architecture

The conceptual architecture for this subsystem follows the Data Abstraction and Object-Oriented Organization. This is because of the way BSD uses sockets. Sockets in BSD are abstract and can be extended or implemented in different ways, particularly the way the socket object is agnostic to any existing protocols. The sockets themselves can be extended to include protocols, but they themselves are not defined by a protocol.

Interfaces provided by the OS

FreeBSD offers interfaces for file and terminal I/O operations. These operations are performed on descriptors and involve tasks like reading data from a file, or writing data to a socket (read, write functions). FreeBSD allows for programs to manage processes by providing the Process Management interface. Functions such as fork(), exec(), and exit() are implemented as system calls which allow access to services from the operating system.

The Virtual-File System(VFS) provides an abstract interface for accessing multiple file systems. It receives requests for file system operations and translates them into the specific operations that the underlying file system requires. It also manages the caching of file system data, the maintenance of consistency across different file systems, and the handling of errors and exceptions.

The memory management interface provides APIs for organizing, allocating, and deallocating memory. The interface is implemented by the Virtual-Memory System. The pager interface is part of the virtual memory interface. It allows data to be moved between the backing store and physical memory which helps to manage available RAM, and to make efficient use of disk storage.

The network communication interface in FreeBSD provides the infrastructure for communication between different systems over a network. It implements the Socket-to-Protocol interface, Protocol-to-Protocol Interface, and Protocol-to-Network Interface, which enable the system to transmit and receive data between layers.

Concurrency

In FreeBSD, processes and threads are the two main units of concurrency. FreeBSD implements concurrency through multi-tasking, multi-threading, and asynchronous I/O. The kernel is extremely concurrent and supports more than 128 hardware threads.

Multi-tasking: FreeBSD implements multi-tasking by allowing multiple processes to run concurrently on a single CPU. Each process has its own memory space and appears to execute independently from other processes. The process scheduler is responsible for selecting which process should run next and allocating CPU time to each process based on priority.

Multi-threading: Within each process, FreeBSD supports multi-threading, allowing multiple threads of execution to run concurrently within a single process. Each thread has its own stack and set of register values, but shares the memory space of the process. This facilitates communication between threads and simplifies inter-process communication.

Global control flow is achieved through a combination of system calls, interrupts, and context switching. The operating system uses scheduling algorithms to determine which unit of concurrency should run next, and it uses context switching to switch between units of concurrency smoothly and efficiently. Context switching ensures that the state of a process can be restored and execution can be resumed at a later time.

External Interfaces

FreeBSD has external interfaces which allows the operating system to communicate with other entities, such as users, devices, and other systems. This could range from controlling other devices, to receiving data from other systems - it is crucial that external interfaces exist so that the operating system can be interacted with. Without a wide range of external interfaces, the operating system would not be useful as we would not be able to control many aspects of our machine, if any at all.

Graphical User Interface

A graphical user interface, often referred to as GUI, offers users the ability to visually control and issue commands to the system. The key components here are the display server, window manager and graphical interface. The display server communicates with the hardware, through the kernel, and issues low-level instructions, through the external interfaces offered by FreeBSD, for displaying things on the screen. The window manager runs on top of the server, allowing the management of windows - such as their sizes, position and order. This is done by calling APIs through the display server which issues the instructions to the hardware. This models the client-server architecture as the display **server** is in charge of managing the visual output depending on instructions sent by the **client(s)** to the server. The client is the intermediary between the user and the server. This creates modularity as the entity issuing instructions is not coupled with the entity that is controlled by the user. This also allows clients to live on a separate machine, as the requests to the server can be made over a network.

Command Line Interface

In FreeBSD, the command line interface is known as a shell. Users can invoke commands through this shell and interact with the computer. These shells have built in functions which help users do routine tasks, and manage their system. There is a set of shells included as part of FreeBSD, such as Bourne shell, C-Shell and sh. The shell works by making system calls, which is provided through an API exposed by the kernel. This API is the external interface which the CLI calls upon to execute actions given by the user.

Design Patterns

Upon conducting a thorough analysis of the FreeBSD documentation and the services provided by its kernel, it was observed that the following design patterns were implemented to attain the objectives of the design. It is worth noting, however, that the kernel is written in the C programming language, which does not possess intrinsic support for Object-Oriented Programming (OOP). Despite this limitation, it is believed that the design patterns were implemented manually utilizing the versatility of C, in a manner similar to the implementation of the Linux kernel.

Factory

Factory pattern is used in the implementation of device drivers in FreeBSD, where drivers are implemented as objects that can be instantiated as needed. For example, a driver for a network interface can be created as an instance of a factory class, with its own unique configuration and settings.

Singleton

Singleton pattern is used in the implementation of some kernel data structures in FreeBSD, which are designed to have a single instance in the system. For example, the global process table, which holds information about all processes in the system, is a singleton data structure.

Observer

Observer pattern is used in the implementation of some event-driven systems in FreeBSD, where objects can register to be notified of changes to other objects in the system. For example, the network stack in FreeBSD uses the observer pattern to notify the network devices of changes to the network configuration.

Facade

Facade pattern is used for the system calls exposed via APIs to developers. This pattern hides the complexities of the larger low-level system and provides a simpler interface. For e.g. the file system system calls are exposed via read(), write() functions to abstract the complex file system underlying it.

Interactions

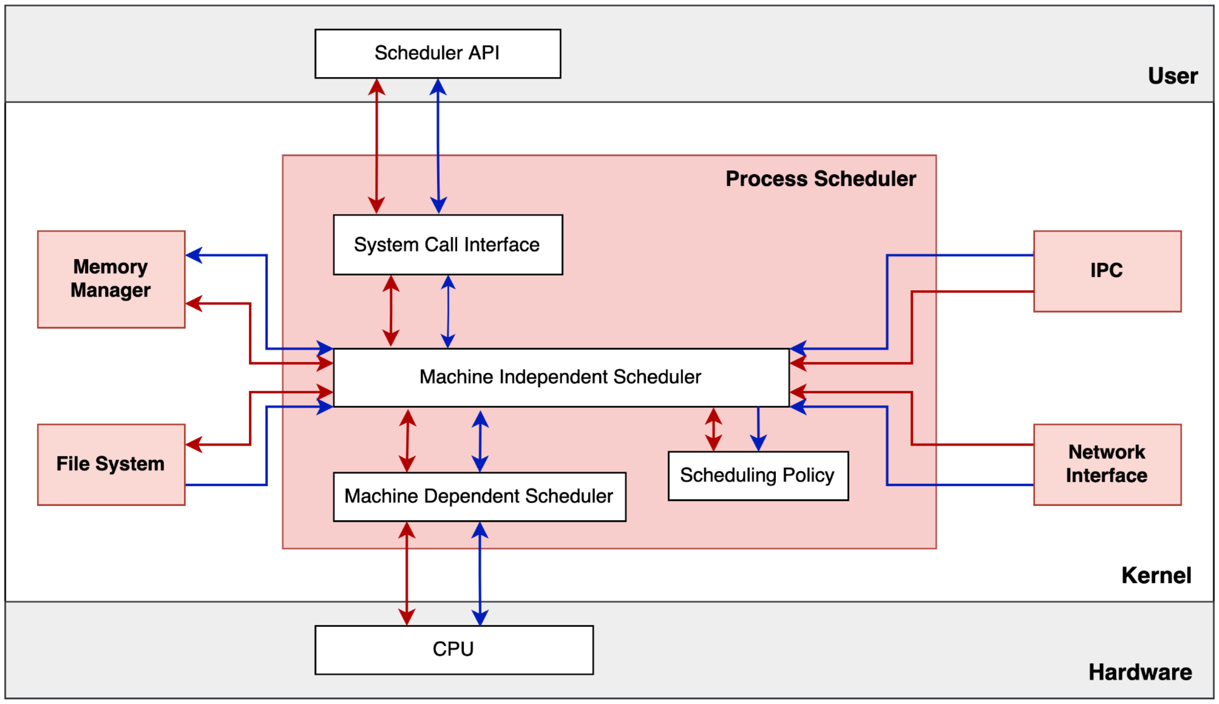


Fig. 3. Data Flow & Control Flow through the Process Scheduler

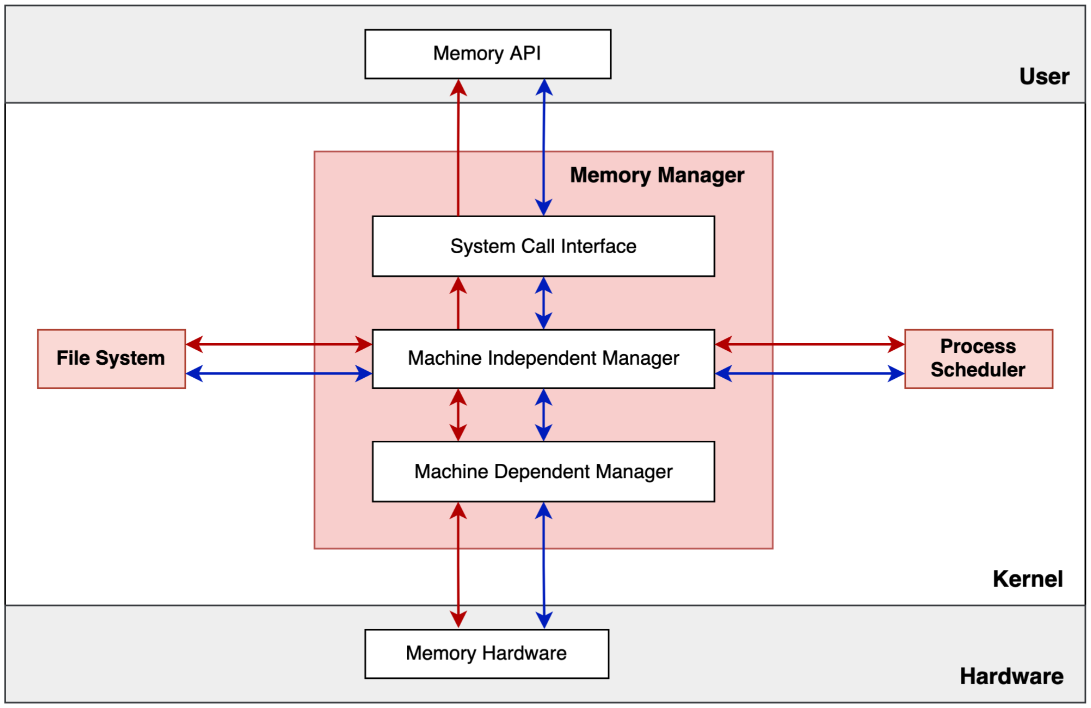
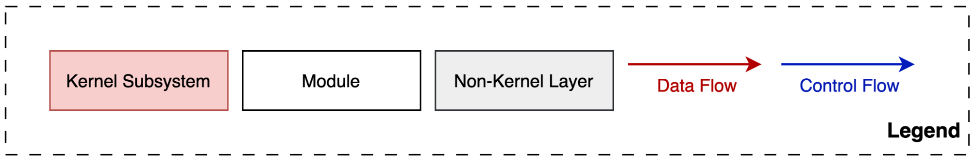


Fig. 4. Data Flow & Control Flow through the Memory Manager



Use Cases

Fig. 5. ***File Reading***

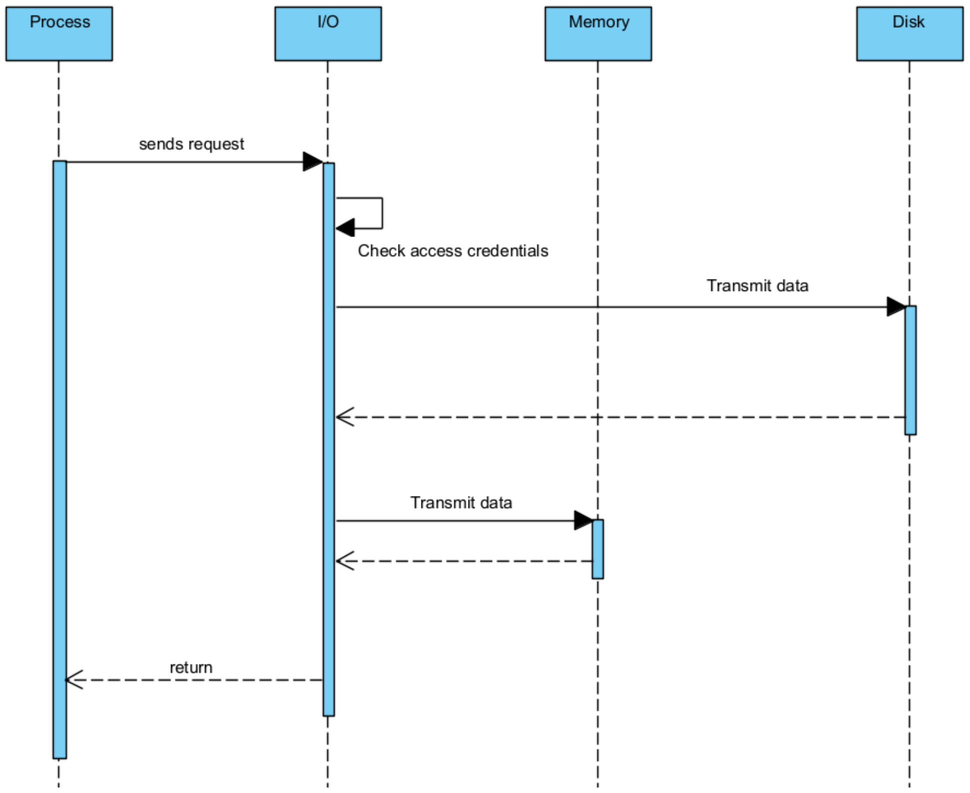
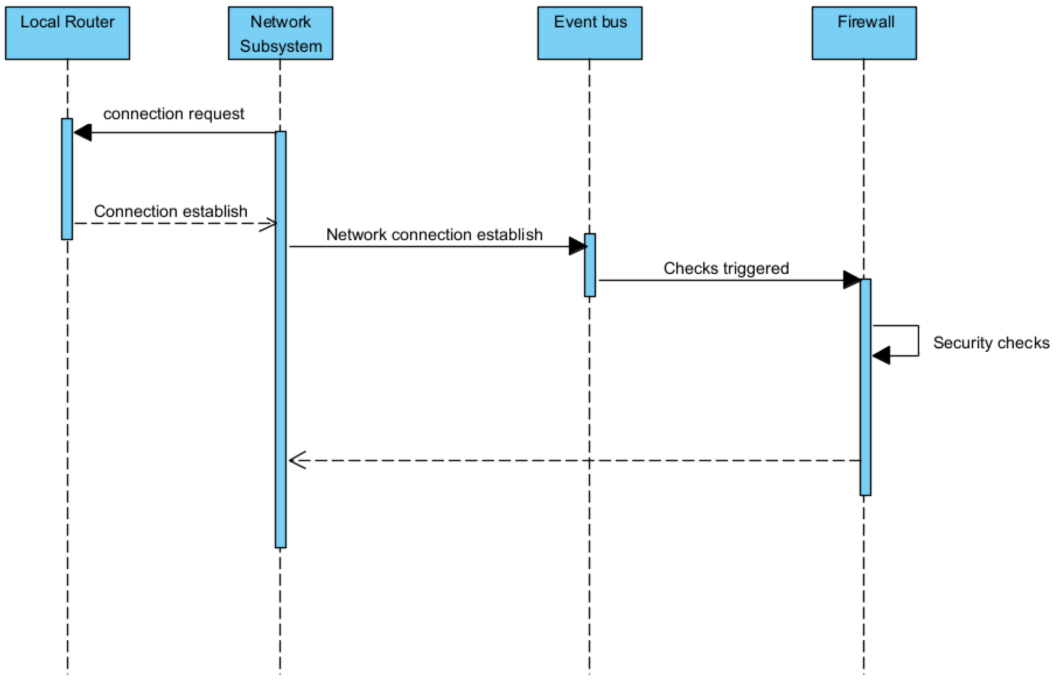


Fig. 6. ***Firewall Trigger***



Data Dictionary

Abstract - A creational design pattern that focuses on providing an interface for creating objects without a concrete class.

Command Line Interface (CLI) - A text based interface used to communicate with the system

Concurrency - The ability to do multiple computations at the same time

Device Driver - The program responsible for how the operating system interacts with an external device connected to the system

Facade - A structural design pattern focused on providing a unified interface to a set of interfaces to reduce the complexity of how to use the subsystem.

Graphical User Interface (GUI) - An interface that uses graphical icons, menus etc. to communicate with the system

Layered - An architecture style that focuses on the separation and the ordering of layers such that a program in one layer may access services below its layer.

Monolithic - An architecture style that focuses on having individual components under a single unified system.

Observer - A behavioral design pattern that defines a dependency between objects such that objects are notified and/or updated when another object changes its state.

Singleton - A creational design pattern that ensures a class only contains one instance of itself.

Transmission Control Protocol (TCP) - One of the protocols in the Internet Protocol suite used for its high reliability of transmission.

User Datagram Protocol (UDP) - One of the protocols in the Internet Protocol suite used for time dependent transmissions such as video streaming.

Conclusions

In conclusion, FreeBSD is using the layered architecture for the structure, and each layer can only interact with its neighbors. FreeBSD is also using monolithic architecture. The only way for the user to give commands to the kernel is system call. In the kernel, there are different kernel subsystems, such as process management, file management, I/O device management, or file management etc. All these subsystems in the kernel help the kernel to handle different kinds of processes. In the kernel, interprocess communication always depends on the socket, socket's main job is for delivering data, and in BSD, socket can be extended or implemented in different ways, because socket can use any existing protocols. For the interface, FreeBSD is using VFS, VFS can receive commands and translate it into operation. There are also some external interfaces such as command line interface, which is for collecting users' commands. In the future FreeBSD may be working on how to extend a new function, because it is using monolithic architecture, all the subsystems are in the kernel, it is hard to add new subsystems to the kernel.

Lessons Learned

Throughout the completion of this assignment, a lot of the focus was placed on getting a deep understanding of FreeBSD components and how they interact with each other. This required the group to extensively research on how operating systems function as well as how they are able to efficiently handle different aspects of the computing system. The group not only studied about the general functionality, but also gained familiarity with the design and implementation of an operating system and the architectures. One of the most critical aspects of succeeding in this assignment depended on how well the team communicated with each other and the organization of the documentation.

References

1. FreeBSD Architecture Handbook
2. *Marshall Kirk McKusick, George V. Neville-Neil, Robert N.M. Watson (2015)* The Design and Implementation of the FreeBSD® Operating System, 2 e.d.
3. *William Joy, Robert Fabry, Samuel Leffler, M. Kirk McKusick, Michael Karels,* Berkeley Software Architecture Manual 4.4BSD Edition https://docs.freebsd.org/44doc/psd/05.sysman/paper.html
4. *David Garlan, Mary Shaw* (1994) An Introduction to Software Architecture
5. TCP/IP layers and their functions (linuxhint.com)
6. *Bowman, I.* (2003) Conceptual Architecture of the Linux Kernel
7. https://matheustavares.gitlab.io/assets/oop_git_and_kernel.pdf
8. https://github.com/jmarkowski/design-patterns
9. https://en.wikipedia.org/wiki/Windowing_system#Display_server
10. https://docs.freebsd.org/doc/3.5-RELEASE/usr/share/doc/handbook/shells.html
11. https://gs.statcounter.com/os-market-share/desktop/worldwide
12. https://en.wikipedia.org/wiki/System_call