EECS 4314 - Bit Theory Architecture Report

Amir Mohamad
amohamad@my.yorku.ca

Arian Mohamad Hosaini
mohama23@my.yorku.ca

Dante Laviolette
dantelav@my.yorku.ca

Diego Santosuosso Salerno
nicodemo@my.yorku.ca

Isaiah Linares
isaiah88@my.yorku.ca

Joel Fagen
joefagan@my.yorku.ca

Misato Shimizu
misato1@my.yorku.ca

Muhammad Hassan
furquanh@my.yorku.ca

Yi Qin
aidenqin@my.yorku.ca

Zhilong Lin
lzl1114@my.yorku.ca

York University

February 10, 2023

Abstract

Software architecture refers to the high-level view of a software system, including all software components, their respective relationships (connectors), and the principles and guidelines of their structural and behavioral design. It provides a broad and abstract view of the architectural design, and encompasses the key concepts and structures of the system. Each of these elements, or abstractions, collaborate and interact with each other and give rise to a collaborative environment that makes a system more than a simple conglomeration of individual parts. The arrangement of these architectural elements and their interactions are critical to the overall success of the system.

In the context of FreeBSD, a Unix-based operating system, its conceptual architecture defines the fundamental design principles, architectural styles, components and connectors that make up such a powerful computing environment. Some of its components are the kernel, the system libraries and user applications.

FreeBSD is based on the BSD model, which started its development in 1993 with one single goal in mind: to provide software that can be used for any purpose. A series of evolutionary upgrades have been made by the thousands of contributors, committers and core members of the project.

The structure of the operating system is composed of 8 main interacting components, which are: the Kernel Facilities, the Security Features, Memory Management, Generic System Interfaces, File System, Terminal-handling, Inter-process communication, and Network communication.

The data and control flows in the subsystems of FreeBSD are as follows: (1) initialization starts with the BIOS and is taken over by the operating system through various boot programs, eventually leading to the user process; (2) inter-process communication is managed by BSD sockets; and (3) file system management is handled by the Zettabyte File System, which is under the control of the operating system and communicates with devices through ISA drivers. Memory management is handled by the system's memory manager, which uses virtual memory and handles kernel memory allocation through various system calls.

In FreeBSD, concurrency relates to `SMPng` Architecture, which includes mutexes, shared/exclusive locks, semaphores, and condition variables such as those found in many other systems.

The two main use cases of FreeBSD are the Shell, which allows for system interaction, and the Networking environment, which has robust capabilities over several protocols, such as `HTTP` and `FTP`.

The modular, efficient and highly configurable software architecture of FreeBSD makes it such a popular choice for various systems, servers, embedded environments and desktop applications.

Keywords— FreeBSD, software architecture, networking, concurrency, design pattern, open source

Contents

1 Introduction

An Operating System is computer software that manages computer hardware. FreeBSD is an open-source operating system based on Unix and widely known for its speed, stability, reliability, performance and security. However, whereas other operating systems combine features from Unix systems, FreeBSD is based on its own BSD model. The FreeBSD Project started its development in 1993 with the goal to provide software that can be used for any purpose with no strings attached. The idea is that the code will get the widest possible use and provide the most benefit. As a result, the software architecture of FreeBSD is designed to be modular, allowing users to easily add or remove features as needed. On the other hand, and similarly to other Unix-based operating systems, the core components of the architecture include (1) the kernel, (2) system libraries, and (3) user programs. In an operating system, the kernel is responsible for managing system resources, such as memory and CPU time. FreeBSD uses a monolithic kernel design, meaning that all device drivers and system calls are integrated directly into it. This allows for efficient communication between the different components of the operating system, and therefore results in faster system performance. The highly configurable and modular kernel supports over 300 system calls, which provide the means for a user program to ask the operating system to perform tasks on their behalf. As a mechanism to support memory management and CPU scheduling, FreeBSD uses demand paging, which is a technique that only loads pages as they are needed, and is very common in virtual memory systems. One of the key features of FreeBSD's software architecture is its use of a hierarchical file system organized in a tree-like structure. In this sense, the root directory is located at the top of the tree and all other directories and files branch off from it. This organization makes it easy to navigate and manage the file system, and also allows for efficient use of disk space. FreeBDS has over 33,000 ported libraries and applications for desktop, server, appliance and embedded environments, and they provide a wide range of functions to be utilized by user applications. Some of these libraries facilitate networking and database management, and others are just standard C libraries to communicate with user programs. Thousands of user programs and applications have been written for FreeBDS. Since it is based on 4.4BSD, an industry-standard version of Unix, it is easy to compile and run programs for all user needs, whether it is internet services, networking, software development, net surfing, or any other user needs. Overall, the conceptual software architecture of FreeBSD is designed to be modular, efficient and configurable, making it a very powerful, flexible and popular operating system for the execution of a wide range of user tasks.

1.1 Overview

This report will analyze the architecture of FreeBSD, a well-known Unix-based operating system. We will look closely at the core components that make it up - the kernel, security features, file system, memory management, and inter-process communication - and see how they all come together to create a robust and efficient computing system. We will also examine the flow of data and control within the system and touch on how the architecture deals with high levels of concurrency with `SMPng` architecture. In addition, we will discuss how users interact with FreeBSD by exploring the system and networking use cases. We will also examine why its modular and efficient design has made it popular for different systems, servers, embedded devices, and desktop applications. This report will comprehensively analyze the critical high-level components of the FreeBSD architecture and how they contribute to its overall success.

2 Architecture

2.1 Evolution

2.1.1 Origins

As you may or may not know, BSD stands for Berkeley Software Distribution. “The University of California at Berkeley obtained a copy of Unix from Bell Labs in 1974. Over the following four years, Bell Labs and Berkeley enjoyed a strong collaborative relationship which helped UNIX to flourish”[1]. This collaboration branched out into different systems built on the Unix Architecture, the one of interest here being BSD. This system was shared around the world while also carrying the requirement of paying AT&T for a license

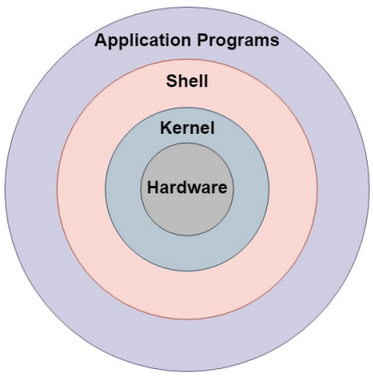

Figure 1: UNIX Layered Architecture Style

first. Due to the commercialization of Unix researchers at Bell Laboratories were no longer able to foster community research. Although community research still continued, just through a different avenue; the Berkeley Computer Research Group, which could further coordinate and produce Unix releases [1]. CSRG was able to make big contributions to UNIX, the major one being implementation of ARPANet protocols (`TCP/IP`). This implementation has been used long after as the basis for `TCP/IP` implementation which shows its significance. Although this was under the licensing fee of AT&T ($50000), this caused many to request for the networking code and utilities to be provided under free licensing terms. Which led to the Networking Release 1. Keith Bostic brought up the idea of another release with even more BSD code, which would take a huge workload to rewrite many lines of code. This pushed him to make a very innovative move which set the standard for open source software, he enrolled developers across the world to contribute to this release, and in return they would be commended with name for their work on the project. This work resulted in Networking Release 2. After this release different groups of development split into different branches all prioritizing different things. One group “initially focused continued development exclusively on the `Intel x86` platform” [1] as well as inexpensive CD-ROM distribution and ease of installation became the most widely used BSD, this is FreeBSD.

2.1.2 Releases

Being an open-source platform, FreeBSD develops as a result of contributions from both its users and its developers. Collaboration is critical to its evolution, which focuses on creating new features, optimizing performance, and introducing additional layers of protection. Developers and users must first identify new system requirements and develop solutions for them in order to start the evolution process. The best suggestions are then chosen for execution after being discussed and assessed by the community.

Significant releases over the years have advanced the development of the FreeBSD system. Some of the most significant releases that demonstrate the system’s evolution through time are the topic of this study.

The release of FreeBSD 4.0 was a pivotal moment in the system’s development. A new package management system was added to the FreeBSD operating system with this version, greatly simplifying the process for users to install and manage packages. Additionally, FreeBSD 4.0 provided significant performance improvements and improved support for modern hardware, making it a more efficient and dependable platform for both residential and commercial use.

FreeBSD 5.0, published in 2004, was the first stable release that represented the system’s growth. With the introduction of this release, the system started adding support for advanced multiprocessors, thread support, and the sparc64 and ia64 platforms. The most important new features were support for hardware virtualization, a new security architecture, and support for more sophisticated file systems like ZFS. These features were among the numerous new features that were added. Additionally, by streamlining the network stack and enhancing the operating system’s virtual memory capabilities, this new version enhanced operating system performance.

Another key release in the development of the system was FreeBSD 7.0. This update improved the system by enhancing data protection features. This edition also saw the introduction of the ZFS file system,

which helped users manage and safeguard their data. It is now much simpler for users to run many virtual machines on a single host thanks to better support for multi-core CPUs and hardware virtualization.

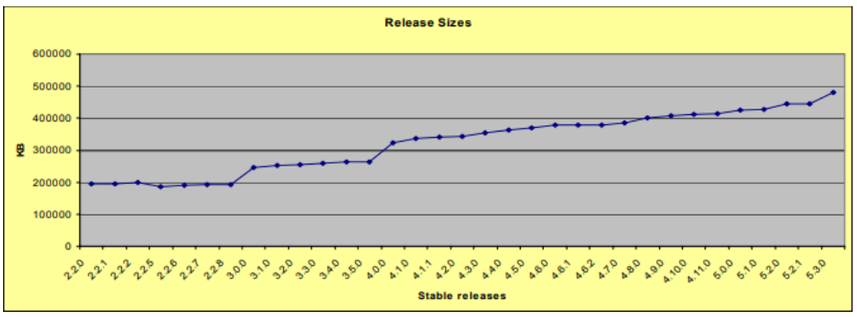

Figure 2: FreeBSD stable release growth [9]

FreeBSD 11.0 brought changes to the systems networked capabilities an further evolved the system into one that could cater to most of its users needs. This release brought major updates to the system network stack. This greatly increased the performance and scalability of the system. In addition `VIMAGE` virtual networking system was introduced and more improvements were made to ZFS support bringing further security enhancements to its users for data protection. To summarize, the evolution of FreeBSD over the years has been shaped by a series of key releases that have brought new features, improved performance, and increased dependability to the operating system. With the help of these updates, FreeBSD has evolved from a simple Unix-like operating system into a strong platform that can accommodate a variety of uses and applications.

2.2 Functionality & Structure

As we all know, the filesystem is an integral part of all kinds of operating systems, and it might be the part that users feel most familiar with. The primary function of a filesystem is to store, input and output the files. Because of that, several parts of FreeBSD need to interact with it. First, the filesystem needs basic system facilities to help control it. After that, memory management and the filesystem cannot work without each other. Then the network communication part needs the support of the filesystem since we often need to upload files. Last but not least, the filesystem is connected with inter-process communication since file changes might happen from processes.

Terminal handling is another essential part of the operating system. Shells are a common way to interface with the underlying operating system through an external interface given input from the user. As a result, it needs to interact with all other parts controlled directly by users, including the generic system interface, basic system facilities and inter-process communication.

Inter-process communication allows different processes to communicate and interact. As mentioned, it gets a command from the terminal handling part and helps kernel facilities, memory management, and the filesystem. At the same time, network communication is also a process, so it needs to interact with inter-process communication. As a result some processes need to be able to keep track of the state of other processes. FreeBSD's process management system uses the observer pattern to manage notify other process of a processes state.

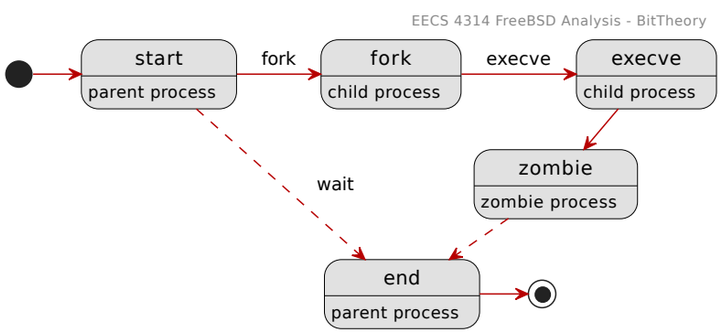

Figure 3: FreeBSD process creation state

Network communication is part of FreeBSD, which is responsible for the internet, which is why we can communicate with each other through the internet. This part gets input from inter-process communication, basic system facilities and the filesystem. After that, it posts output to the security part first to ensure the data received are harmless.

FreeBSD offers advanced networking, performance, security and compatibility features as a multi-user operating system for various platforms. These features break into eight interacting components: basic

kernel facilities (focusing on Process Management), security features (focusing on access control), memory-management support, generic system interfaces (I/O, control and multiplexing operations), file systems, terminal handling, inter-process communication facilities and network communication.

The first is the kernel facilities which include timer and system-clock handling, descriptor management, and process management. Since process management is essential for most kernel layer activities, it will interact with many other parts. The first part it will interact with is security which is also the access control component. Access control is the process where the user's identity is authenticated. The kernel is a crucial component and as a result, FreeBSD is designed to make architecture of this component simpler. FreeBSD uses `Kobj` (Kernel Objects) to provide and Object Oriented Programming (OOP) support with the C language for the kernel.

The second part is memory management. The process will be saved in the memory, but the memory is also the process of controlling a computer's main memory. The third and fourth parts are inter-process communication and file systems. Both parts will depend on the process management to start and end their operations.

The third one is security. It is the access control component of the system, and it provides the functionality for the system to authenticate the user's identity and give access according to the user's role. It is related to the kernel facilities and network communication because both components need to identify the user's access. Security is also related to the generic system interface since it will provide a method for the user to enter their user information and check it with the memory to give access to the user.

The generic system interface is the I/O component of the system. Whenever the user needs an I/O service, the system will connect with other components within the generic system interface. It will also store the information and save them to the main memory.

Memory management is the main component that stores the process, user identification and much other information in the main memory. It gathers the data when the process management calls for the data. It will allocate the memory before and after the process within the system. It has a strong connection with the file system. The file system depends on the memory management system because the information of the files is located inside the memory.

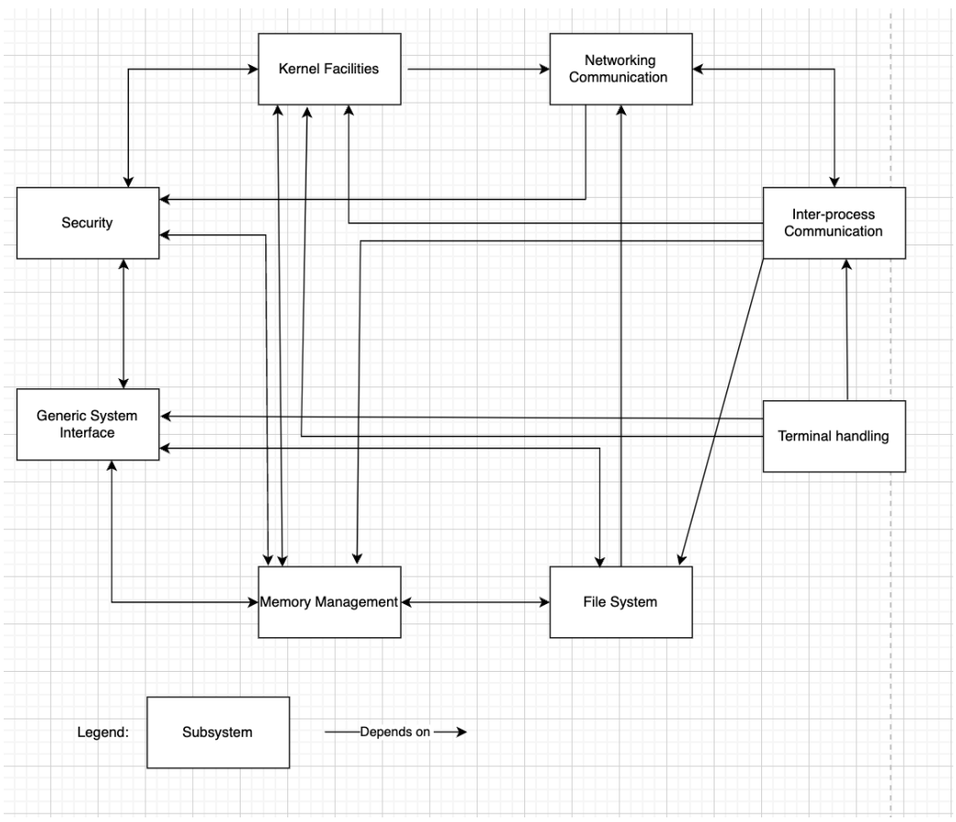

Figure 4: FreeBSD dependency diagram

2.3 Control & Data Flow

What is the control and data flow among parts?

Data flow is how data flows among system components during processes. Control flow is the order of executions among components. The explanations below show how both flows are structured in subsystems of FreeBSD.

2.3.1 Initialization

In the initialization process, Basic input/output system (BIOS) with read-only memory (ROM) will be the first program appears in the flow. Under the control of BIOS, the processor finds the address `0xffffff0` which contains the how-to of POST routines for low-level initializations. Then, as the last POST routine, BIOS follows an instruction, called `INT 0x19`, to reach out the address `0x7c00` and find a boot0 program. At the point when a boot0 program executes, the control will be taken over by the FreeBSD operating system. After taking over its control, the system will jump to multiple addresses to go through other boot programs and BTX server (boot1, BTX server, and boot2), and eventually reach a loader, which boots the system's kernel. After the kernel completes its boot process, the control would be taken over by the user process, `init`, and it begins user-level configurations so that users can login and access to a shell. In multi-user operation, `init` calls a get tty (`getty`) program to call `login` out, and in single-user operation, `init` will be return to a root shell instead.

2.3.2 Inter-Process Communication

The major component for IPC in FreeBSD is a descriptor file, called BSD sockets. For example, to obtain an object on a web page, the system recognizes the object, which is encoded as a file, and utilizes a HTTP protocol to obtain more information on the object. These processes can be completed in the system's kernel. Then, BSD sockets take care of the process of utilizing Transmission Control Protocol (TCP) and Internet Protocol (IP) to obtain the object from a server.

2.3.3 File System

FreeBSD supports many different types of file systems and this is possible due to the usage of the adaptor design pattern. For the processes of communicating with devices, Zettabyte file system (ZFS) takes a significant role in the system. ZFS is a system that handles general file system jobs as well as the volume management jobs, and it is under the control of the operating system. By combining multiple devices into a pool, it generates a large file system. Based on this file system, data can be exchanged between a hardware device and programs. For programs to obtain data, stored in a device, an ISA driver controls the communication between them, and it copies the necessary data from the device memory into the main memory. Moreover, if data needs to be stored in a device, the driver takes the data address and sends it to the device. Then, the hardware's Direct Memory Access (DMA) mechanism allows the device to access the data. Another type of file system is the Network File System (NFS). It uses a server-client pattern along with the master-slave design pattern to allow remote connections to the host through the networking stack.

2.3.4 Memory Manager

We can find a lot of components that the system's memory manager handles. One of the responsibilities is kernel memory allocation, which the system utilizes virtual memory (VM) to handle page tables with addresses. For instance, system calls such as `brk` can be made by library functions (e.g., `malloc`) to control the allocation volume of a process, and the system's kernel in the OS reach out to the main memory if necessary. The whole process here can be completed by the kernel in the system. FreeBSD uses the singleton pattern in its memory manager, more specifically the kernel has a single memory allocator. This crucial design choice allows the system to be less complex.

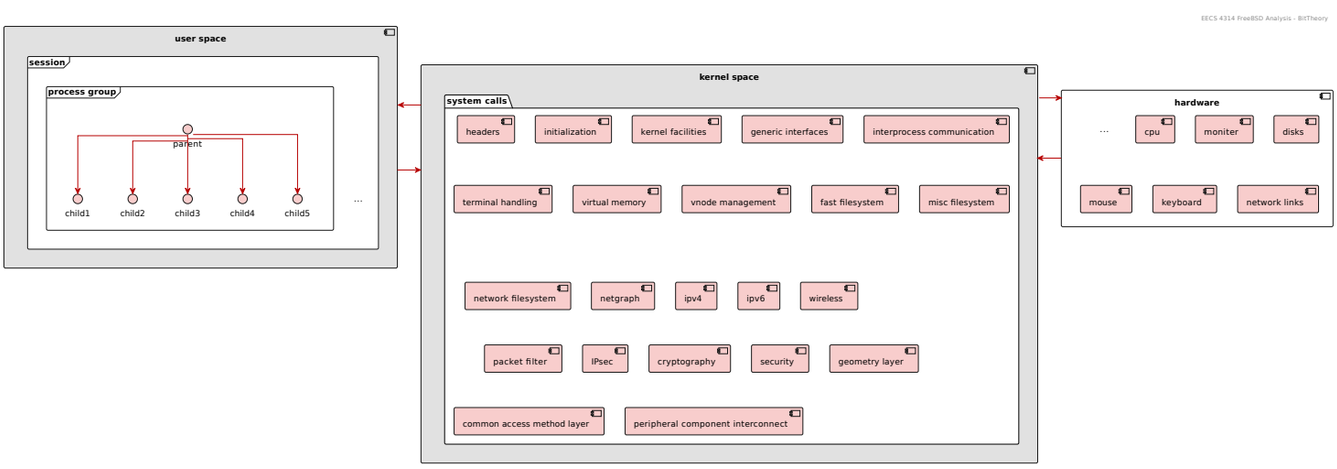

Figure 5: FreeBSD control & data flow between major layers

2.4 Concurrency

Concurrency and parallelism in FreeBSD are primarily driven by the `SMPng` Architecture. The `SMPng` Architecture shares a lot of the same primitives as other architectures, such as mutexes, shared/exclusive locks, semaphores, and condition variables. The `SMPng` design was introduced in FreeBSD Version 5, to better accommodate multiprocessing systems and improve scalability.

2.4.1 Read and Write Locks

The `SMPng` Architecture supports both read locks and write locks. Multiple threads can safely read a portion of data at the same time without it becoming corrupted. However, once write operations are introduced, locks are required to protect data. One method used to circumvent this issue is by simply using an exclusive lock for writes and a shared lock for reads. Another method is less commonly used. It involves having multiple locks for a portion of data, with a read lock for only one of the locks and a write lock for all of them.

2.4.2 Giant Lock vs Fine-Grained Locking

Older versions of the FreeBSD system would use one Giant lock for all of the kernel, but in later versions the Giant lock was removed for various parts of the system such as the device drivers, UFS file system, and network stack. Overtime, the Giant lock has been replaced with smaller, fine-grained locks which improves parallel performance and scalability with multi-processor systems. Another reason the Giant lock was overhauled was because of Giant lock contention. This occurs most frequently during kernel-heavy workloads, such as networking or storage, resulting in more delays as the CPU waits for the lock to be unlocked.

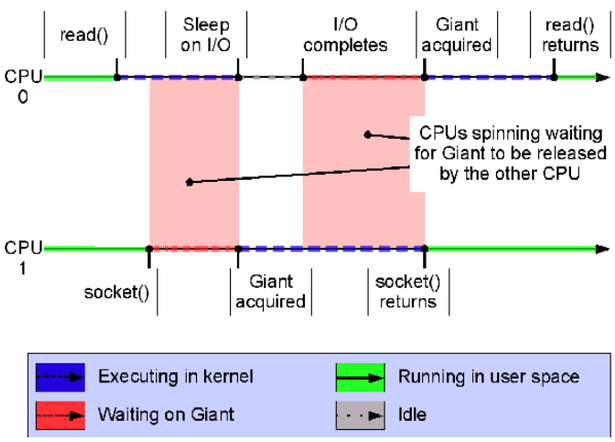

Figure 6: Giant Locking

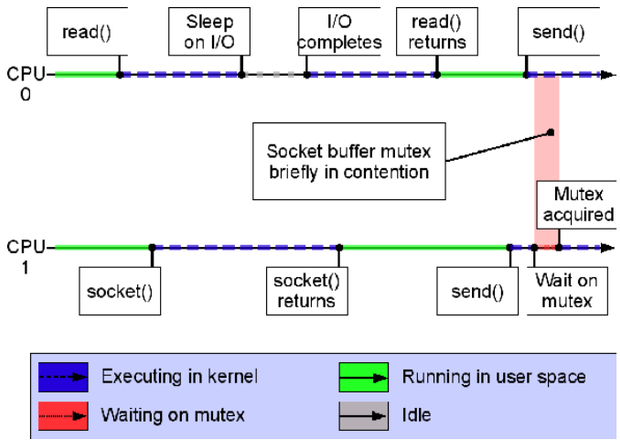

Figure 7: Fine-Grained Locking

Figure 8: Comparison of different kinds of contention between Giant Lock and Fine-Grained Lock (Source: Watson) [11]

2.4.3 Atomic operations

The `SMPng` Architecture also supports atomic operations, i.e. a set of instructions that are protected together by a lock, without releasing the lock in between the execution of operations by a CPU. Atomic operations guarantee that after execution, all changes to the data made by a CPU are made visible at the same time to other CPUs. Atomic operations only allow one item to be read or written at a time.

2.4.4 Interrupt Handling

FreeBSD provides interrupt handlers with interrupt thread context, allowing them to block processes during execution, even with locks. The interrupt threads are often called heavyweight interrupt threads, because switching to them requires a full context switch. In order to mitigate latency, the kernel was made preemptive. Interrupt threads run at real-time kernel priority (ie the highest priority) and therefore, they should not run very long to avoid starvation. There are some interrupt handlers that does not execute in a thread context, instead opting for primary interrupt context, but these handlers are only used in clock interrupts and serial I/O interrupts. In general, the CPU should be doing the highest priority work whenever possible.

3 Use cases

3.1 Shell

The highest level use case of FreeBSD is using a shell to interact with the system. Figure 9 shows how FreeBSD handles this use-case for a single user, from starting up the system, to shutting it down. It is important to note that this diagram showcases the flow in multi-user mode. If the system were in single-user mode, `init` would return a root shell instead of going through the `getty` and `login` process[4][6]. FreeBSD uses the command design pattern to allow users to issue commands to the OS along with the pipe and filter architecture to process complex commands.

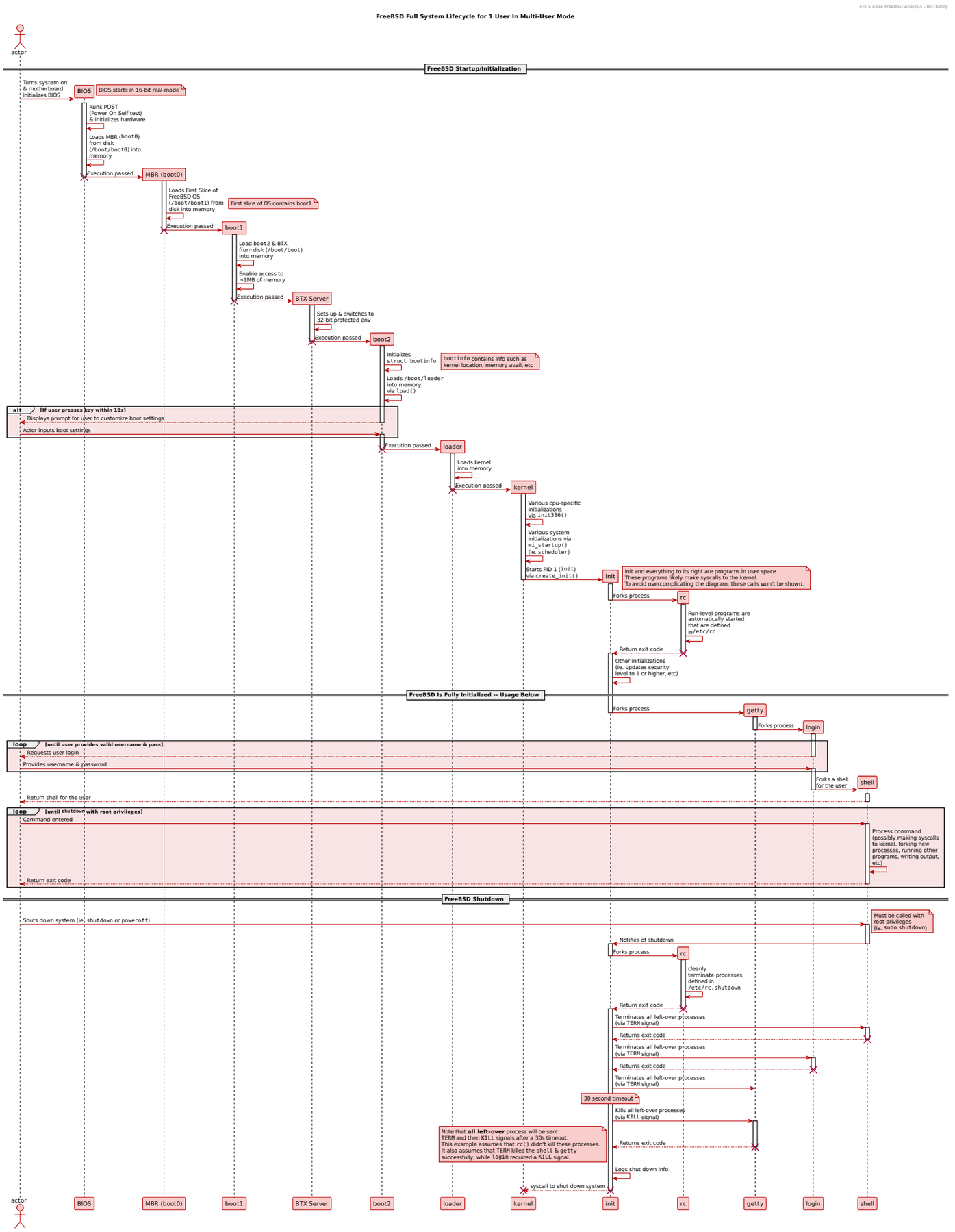

Figure 9: Sequence diagram of FreeBSD system-flow from startup to shutdown (In multi-user mode) [4][6][7][8]

3.2 Networking

One of the most important use cases of FreeBSD is the ability to network with other servers and systems. Because FreeBSD is widely used as a server, as result it must have robust networking capabilities over several protocols such as `HTTP`, `FTP`, `NFS` and many more. Figure 10 shows how FreeBSD handles a network request using the `HTTP` protocol using a browser. Please note that the default FreeBSD system does not come with applications such as browsers or networking tools such as `curl` and they must be installed after initialization.

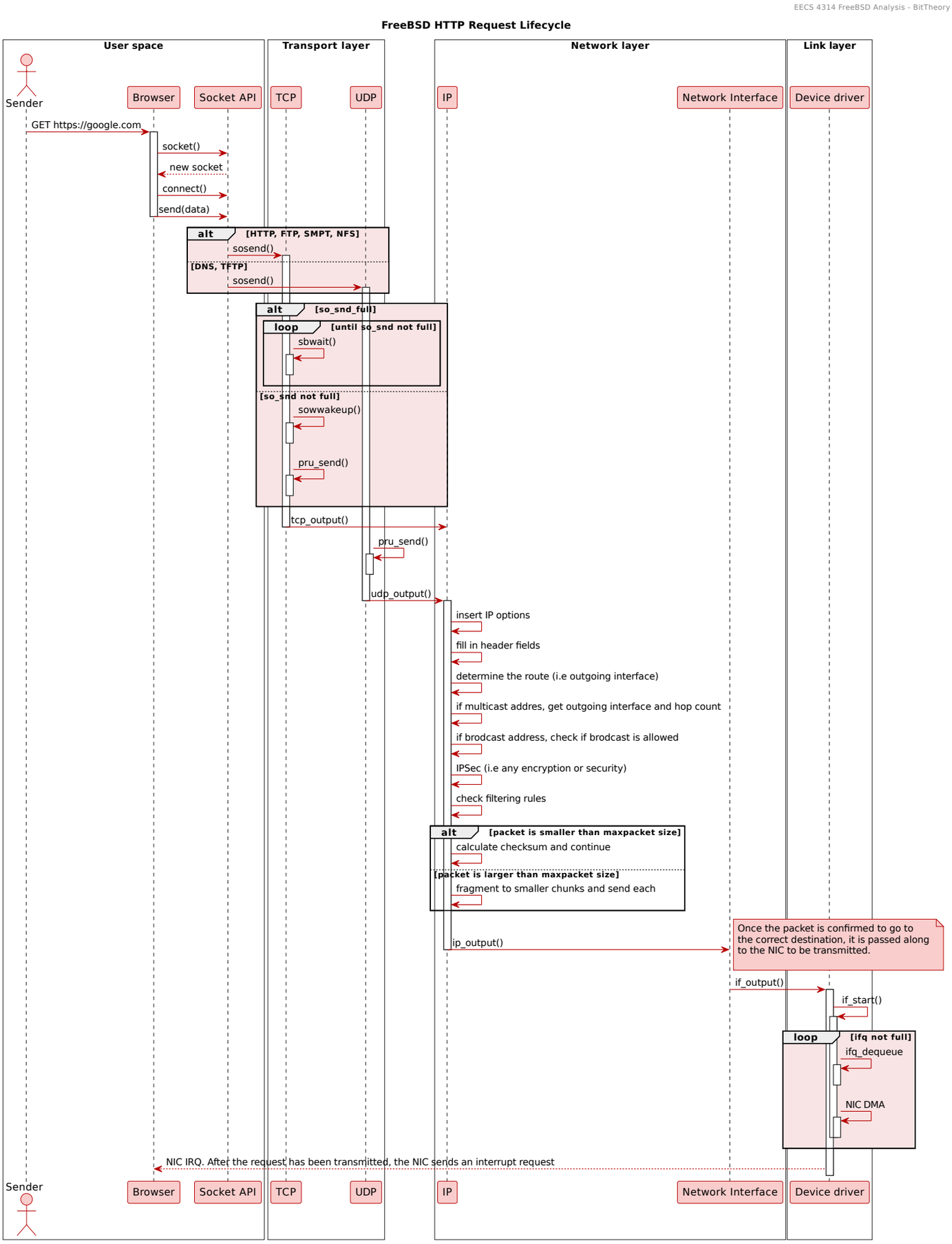

Figure 10: Sequence diagram of FreeBSD network-flow when a user makes an `HTTP` request [10]

4 The engineering process behind FreeBSD development

The FreeBSD project is a unique collaborative venture that sees developers from around the world come together to contribute to and maintain an open-source operating system. It relies heavily on the dedication and commitment of its developers, requiring them to collaborate, compromise and work harmoniously to produce the best results. [2] shows that increasing the number of participants increases the communication in the project exponentially. The creators of FreeBSD inherited the project model approach to reduce the communications overhead [5]. The project model is not meant to create impositions for participating developers, but its purpose is to act as a tool to facilitate interaction and coordination. This section will explore the implications of responsibilities among participating developers in the FreeBSD project and how these responsibilities can vary based on individual roles.

4.1 Standards and guidelines

The FreeBSD project requires developers to adhere to certain standards and guidelines for successful outcomes. To ensure that the entire team works together effectively, each developer must take ownership of their own tasks and be confident in their abilities. This means ensuring that any code they contribute complies with the coding style guide set out by the team, any tests written are accurate and complete, as well as making sure that any issues raised on GitHub or elsewhere regarding their work are addressed promptly. Moreover, communication between contributors is a must; this includes responding to queries in a timely manner and engaging positively in discussions relating to the project. Furthermore, it is important for developers involved in the project to stay up to date with changes made by other contributors so they can adjust their own work accordingly if necessary. This responsibility allows them to understand how each element has been incorporated into the wider system, as well as how it affects other parts of it. By taking responsibility for their individual roles within the FreeBSD project and adhering to established guidelines and standards, developers can help ensure its success.

4.2 Collaboration tools

There are various tools that enable developers to collaborate with one another and share their work on an online platform. These tools include issue tracking systems which allow developers to track tasks and defects in order to ensure progress towards a successful completion; access control systems which provide users with secure logins and permissions; versioning tools, so that different versions of code can be managed effectively; bug tracking systems, so any issues or bugs can be quickly identified and addressed; documentation repositories, where contributors can provide explanations for how certain features work; test suites enabling automated testing of code before it is released into production environments; and code review processes which help ensure quality standards are met. All these mechanisms combined mean that anyone looking to contribute something valuable has the resources they need at their fingertips while still being accountable for every line they write.

4.3 Project Management Hierarchy

FreeBSD project has a well-defined project management hierarchy. There is a Core Team that leads the project, a group of experienced developers who have commit rights to the source repo of FreeBSD, and finally there are contributors who despite not having commit right, help in advancement of the project.

4.3.1 Core Team

The core team is a group of 9 individuals who are responsible for overseeing the overall direction of the project. These are experienced developers who have a long history of contributing to the project.

4.3.2 Project Lead and Committers

Beneath the Core Team, we have the Project Leads. These are individuals who are responsible for specific areas of the project, such as the kernel, userland utilities, or documentation. They are the primary point of contact for questions and problems related to their area of expertise. Next up, we have the Committers. These are individuals who have been given the ability to make changes to the codebase. They review and merge contributions from others, and they work closely with the Project Leads to ensure that the code they commit is high quality and meets the standards set by the Core Team.

4.3.3 Contributors

At the bottom of the hierarchy, we have the Contributors. These are individuals who have contributed to the project, such as submitting a bug report, writing documentation, or writing code. They play an important role in the development of the operating system, but they do not have the same level of decision-making authority as the other members of the hierarchy.

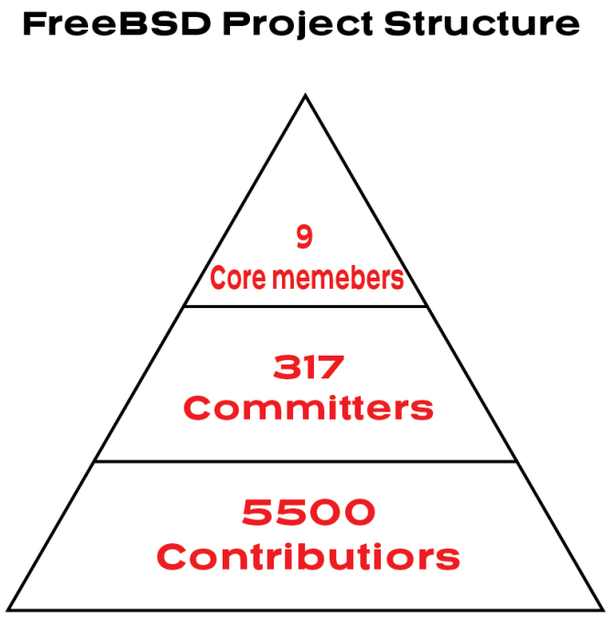

Figure 11: Project Hierarchy of FreeBSD [5]

5 Conclusion

In conclusion, FreeBSD is an Open Source Operating System that can be used effectively in many diverse situations without any licensing fees and liberal copyright terms, which is what it was intended to do when it was first created. This community build distribution shows the strength of open source development because of its features, performance, and wide use in the industry. Commonly used for networking because of its reliability leading to less time fixing problems and rebooting servers (widely used for `TCP/IP` stack). Its wide hardware support also helps to make FreeBSD usable in more situations. We have completed our investigation into the conceptual architectural architecture of FreeBSD and found how the components interact with and depend on each other. What we have found has shown us why this distribution is actually used widely in the industry even though it is an open source operating system. We first looked at the top -down structure and functionality of the entire operating system, showing the main dependencies of the components of the system. Then found some notable advantages in the architecture compared to other operating systems in the Control and Concurrency such as the Zettabyte File System and the `SMPng` Architecture.

6 Lessons Learned

The architecture of FreeBSD, a Unix-based operating system, is a prime example of a well-designed and flexible software architecture. With its main interacting components, including the kernel, memory management, and inter-process communication, FreeBSD offers a robust environment for a wide range of applications. Different components, such as the memory manager and the Zettabyte File System, smoothly manage the data and control flows in the system. The `SMPng` Architecture ensures that the system operates efficiently and securely, even in high-concurrency situations. Whether using the Shell or the Networking environment, users can count on the configurable and highly functional architecture of FreeBSD to meet their needs. Overall, the architecture of FreeBSD is a critical factor in its popularity and success as an open-source operating system.

7 Data Dictionary

- `brk`: FreeBSD program for handling the volume of memory allocation
- `malloc`: A function for allocating uninitialized memory in the kernel space
- `init`: FreeBSD program for handling system (de)initialization
- `rc`: FreeBSD program for (de)initializing programs on startup and shutdown
- `getty`: FreeBSD program to set terminal mode
- `Kobj`: kernel objects that provide an OOP system or the kernel
- User space: The upper layer where user processes (non-kernel applications) run
- Kernel space: The area of memory restricted for privileged kernel mode processes

8 Naming Conventions

- BIOS: Basic input/output system
- ROM: Read-only memory
- IPC: Inter-process communication
- TCP: Transmission control protocol
- IP: Internet protocol
- ZFS: Zettabyte file system
- DMA: Direct Memory Access
- VM: Virtual Memory
- POST: Power on Self Test
- MBR: Master Boot Record
- BTX: Boot Extender
- PID: Process ID
- CSRG: Berkeley Computer Research Group
- OS: Operating System
- I/O: Input/Output
- NIC: Network interface card
- IRQ: Interrupt request
- NFS: Network file system

References

[1] Bretthauer, David. “Open Source Software: A History.” OpenCommons@UConn, Published Works, 2001, https://opencommons.uconn.edu/librpubs/7/.

[2] Brooks, Frederick P. The Mythical Man-Month: Essays on Software Engineering. Anniversary Edition (2nd ed.). Addison-Wesley Pub Co, 1995.

[3] “Chapter 8. SMPNG Design Document.” FreeBSD Documentation Portal, https://-docs.freebsd.org/en/books/arch-handbook/smp/.

[4] “Chapter 14. the Freebsd Booting Process.” FreeBSD Documentation Portal, https://-docs.freebsd.org/en/books/handbook/boot.

[5] “FreeBSD Books.” dev-model, https://docs.freebsd.org/en/books/dev-model/

[6] “FreeBSD Manual Pages.” Init, https://man.freebsd.org/cgi/man.cgi?init.

[7] “FreeBSD Manual Pages.” Getty(8), https://man.freebsd.org/cgi/man.cgi?query=getty

[8] “FreeBSD Manual Pages.” Login(1), https://man.freebsd.org/cgi/man.cgi?login%281%29.

[9] Izurieta, Clemente, and James Bieman. “The Evolution of Freebsd and Linux: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering.” ACM Conferences, Association for Computing Machinery, 21 Sept. 2006, https://dl.acm.org/doi/10.1145/1159733.1159765.

[10] McKusick, Marshall Kirk, et al. The Design and Implementation of the Freebsd Operating System, 2nd Edition = FreeBSD Cao Zuo Xi Tong She Ji Yu Shi Xian, Di 2 Ban. Ren Min You Dian Chu Ban She, 2016.

[11] Watson, R. “[PDF] Introduction to Multithreading and Multiprocessing in the Freebsd SMPNG Network Stack: Semantic Scholar.” [PDF] Introduction to Multithreading and Multiprocessing in the FreeBSD SMPng Network Stack | Semantic Scholar, 1 Jan. 1970, https://www.semanticscholar.org/paper/Introduction-to-Multithreading-and-Multiprocessing-Watson/088630aae6b4622e4175f58045cde69159d9eff5.