FreeBSD:
Concrete Architecture

EECS 4314: Advanced Software Engineering | Assignment 2
March 10th, 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

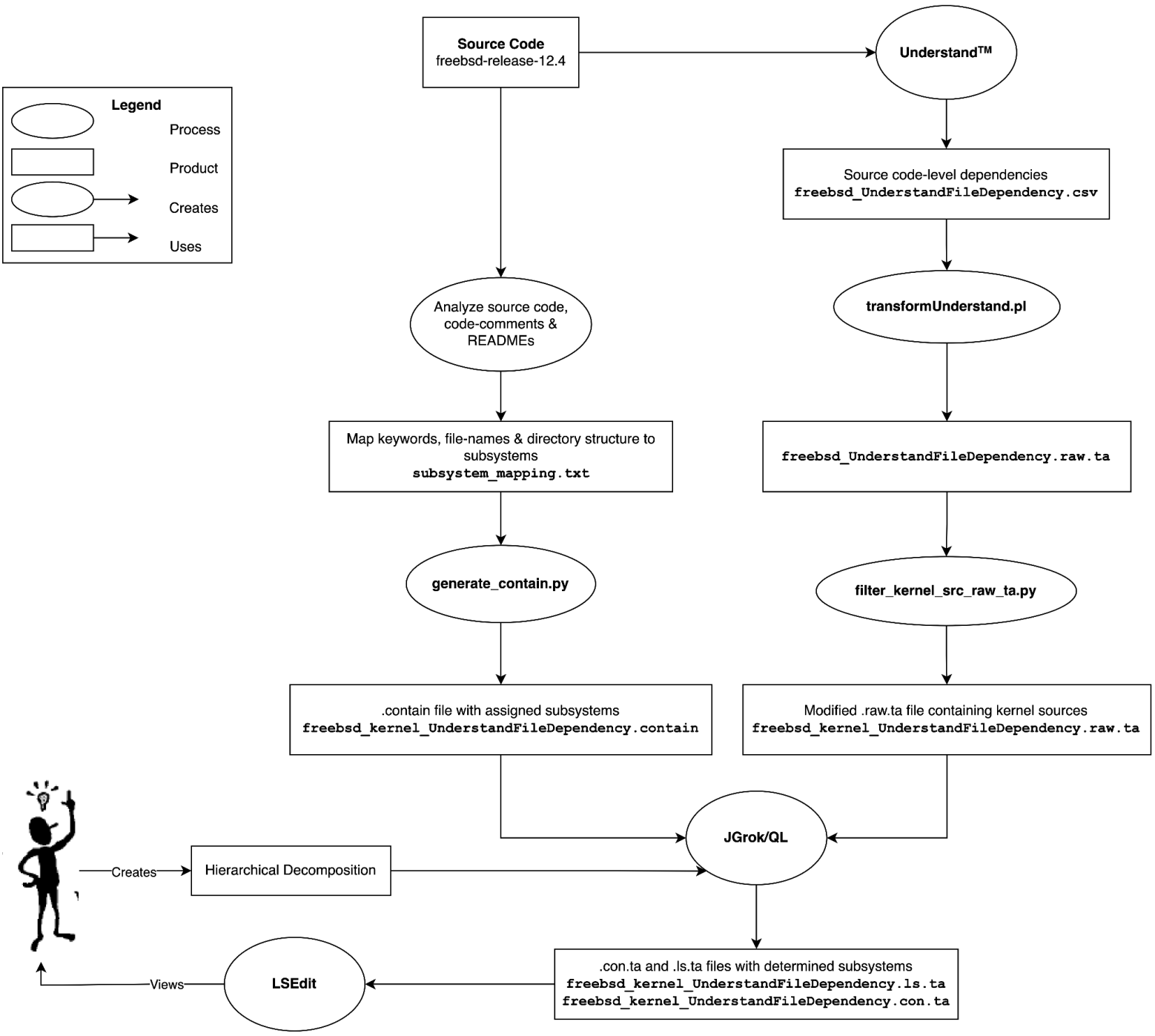
Abstract

This paper investigates and extracts the concrete architecture of FreeBSD. The derivation process for creating the overall container files was divided into three subprocesses: identifying the subsystems, modifying the .raw.ta file, and creating the container files. To identify the subsystems, a hierarchical decomposition was generated by assigning source files to subsystems. The .csv file was reduced to remove files outside of the kernel so a modified and workable .raw.ta file could be created. LSEdit was used to visualize the extracted system architecture. The concrete architecture of the kernel has more subsystems and dependencies than the conceptual architecture, so the conceptual architecture was adjusted accordingly. FreeBSD utilizes locks, mutexes, and other synchronization primitives to achieve concurrency. In terms of the high-level architecture, FreeBSD uses both layered and monolithic architectural styles: communication to and from the kernel uses layered architecture, and the kernel uses a monolithic architecture . The subsystems identified are Core.ss, Memory.ss, IPC.ss, Network.ss, Terminal.ss, Security.ss, DevicIO.ss, Compatibility.ss, Process.ss, Utility.ss, and FileManagement.ss, and Architecture.ss. 'Core.ss' is a central subsystem that all other subsystems have a dependency with. The Memory Management subsystem has dependencies to and from other subsystems. It is responsible for the virtual memory abstraction, handling memory pressure, and kernel memory allocation (Johnston, 2017). Use cases including the Slab Allocator and Page Fault Handling are used to further describe the top level functionality of this subsystem. Process Management is performed by the kernel, and depends on all the subsystems except the Terminal, and IPC subsystem. Next, FreeBSD uses UFS and ZFS for file management. The File Management System has several dependencies, including the Networking system which it needs to access files over the network. The networking subsystem exchanges data with users through files such as core networking code, graph-based networking subsystem, wireless networking, and protocols. FreeBSD's device I/O subsystem depends on the memory management subsystem, which allocates virtual and physical memory. The IPC subsystem uses common routines that other subsystems also use, and the interaction between IPC and Security subsystems implies that the security subsystem may be in charge of ensuring the socket is behaving correctly. This paper also performs a Reflexion analysis which identifies the gaps between concrete and conceptual views with a focus on the unexpected dependencies between subsystems, and the number of them (The FreeBSD Documentation Project, 2012).

Derivation Process

A high level overview of the derivation process is represented in the given figure. As for more detailed information on the derivation process, the derivation process was divided into three separated subprocesses. Each subprocess created as an output the necessary files to be used to create the overall container files. These subprocesses are as follows

- Identifying the Subsystems
- Modifying the .raw.ta
- Creating the .contain files



Identifying Subsystems

As opposed to the conceptual architecture where understanding is derived from the documentation, understanding of the concrete architecture requires understanding of the source code, in particular the folder structure of the source code. When deriving the different subsystems the kernel is made of, we first had to identify the actual source tree of folders. From the source tree, we determined that the /sys/ subfolder is the kernel folder and once it was found, we looked at the different subfolders that

the kernel folder was made of. These subfolders would then be analyzed to see if it is a potential subsystem. During this process, the folder /sys/ contained a readme that gave us clues to the potential different subsystems the /sys/, presumably the kernel, uses. This source roadmap was the basis of identifying whether the subfolders of /sys/ is a subsystem or belong to a subsystem. Each of these subfolders were individually analyzed based on different criterias such as what their ReadMe documents (if it exists), the naming convention of these subfolders and other factors. The idea was that each of these subfolders must be in some way connected to a subsystem, is the subsystem itself, or that the combination of these subfolders results in a subsystem. And as such our structure of a subsystem was essentially made up of folders that create the subsystem. For example, the Network subsystem consists of the following subfolders "/net", "/net80211", "/netgraph", "/netinet", "/netinet6", "/netipsec", "/netpfil" as determined by looking at the source tree and the source code. Towards the end of this process, we determined the different potential subsystems the kernel was made of and the different directories each subsystem uses or contains.

```
Source Roadmap:
---------------
| Directory | Description |
| --------- | ----------- |
| amd64 | AMD64 (64-bit x86) architecture support |
| arm | 32-bit ARM architecture support |
| arm64 | 64-bit ARM (AArch64) architecture support |
| cam | Common Access Method storage subsystem - `cam(4)` and `ctl(4)` |
| cddl | CDDL-licensed optional sources such as DTrace |
| conf | kernel build glue |
| compat | Linux compatibility layer, FreeBSD 32-bit compatibility |
| contrib | 3rd-party imported software such as OpenZFS |
| crypto | crypto drivers |
| ddb | interactive kernel debugger - `ddb(4)` |
| fs | most filesystems, excluding UFS, NFS, and ZFS |
| dev | device drivers and other arch independent code |
| gdb | kernel remote GDB stub - `gdb(4)` |
| geom | GEOM framework - `geom(4)` |
| i386 | i386 (32-bit x86) architecture support |
| kern | main part of the kernel |
| libkern | libc-like and other support functions for kernel use |
| modules | kernel module infrastructure |
| net | core networking code |
| net80211 | wireless networking (IEEE 802.11) - `net80211(4)` |
| netgraph | graph-based networking subsystem - `netgraph(4)` |
| netinet | IPv4 protocol implementation - `inet(4)` |
| netinet6 | IPv6 protocol implementation - `inet6(4)` |
| netipsec | IPsec protocol implementation - `ipsec(4)` |
| netpfil | packet filters - `ipfw(4)`, `pf(4)`, and `ipfilter(4)` |
| opencrypto | OpenCrypto framework - `crypto(7)` |
| powerpc | PowerPC/POWER (32 and 64-bit) architecture support |
| riscv | 64-bit RISC-V architecture support |
| security | security facilities - `audit(4)` and `mac(4)` |
| sys | kernel headers |
| tests | kernel unit tests |
| ufs | Unix File System - `ffs(7)` |
| vm | virtual memory system |
| x86 | code shared by AMD64 and i386 architectures |
```

Figure 1 - The subfolder /sys/'s roadmap used for determining the subsystems

Modifying the csv and the generate raw.ta file

Initially the .csv file provided included subfolders that were not inside the /sys/ subfolder. This means that the .csv also included other systems outside of the operating system's kernel, adding unnecessary lines within the .csv files. This caused the size of the .raw.ta file generated by the Perl script to be increased unnecessarily as well. This large .raw.ta file almost makes it impossible to map into LSEdit, and thus the .raw.ta file required modification to decrease its file size while increasing its usability. From the previous process, we determined that the kernel of the operating system is located under the folder /sys/, this means that any files outside of the /sys/ may not be necessary. A script was created to remove rows that contained references outside the /sys/ directory. This script reduced the number of lines from the initial ~1.2 million lines to a more workable ~190,000 lines. We were also careful to not reduce the number of lines such that certain subsystems that we identified would reference empty folders. The result of this process was therefore, a modified and workable .raw.ta file that LSEdit is able to use and we are able to use when creating the container files.

Creating the container files

With the different container files created, we created a script to combine the different container files to create the overarching container file that reflects that whole kernel of the operating system. However,

before using LSEdit we first needed to modify the parameters for LSEdit to optimize performance and launch the editor with enough resources. The following flags were added in the .runLSEdit.sh script.

- `XMS1024M`
- `XMX1024M`
- `Dsun.java.2d.noddraw=true`
- `Dsun.java2d.ddscale=true`

High-level Concrete Architecture

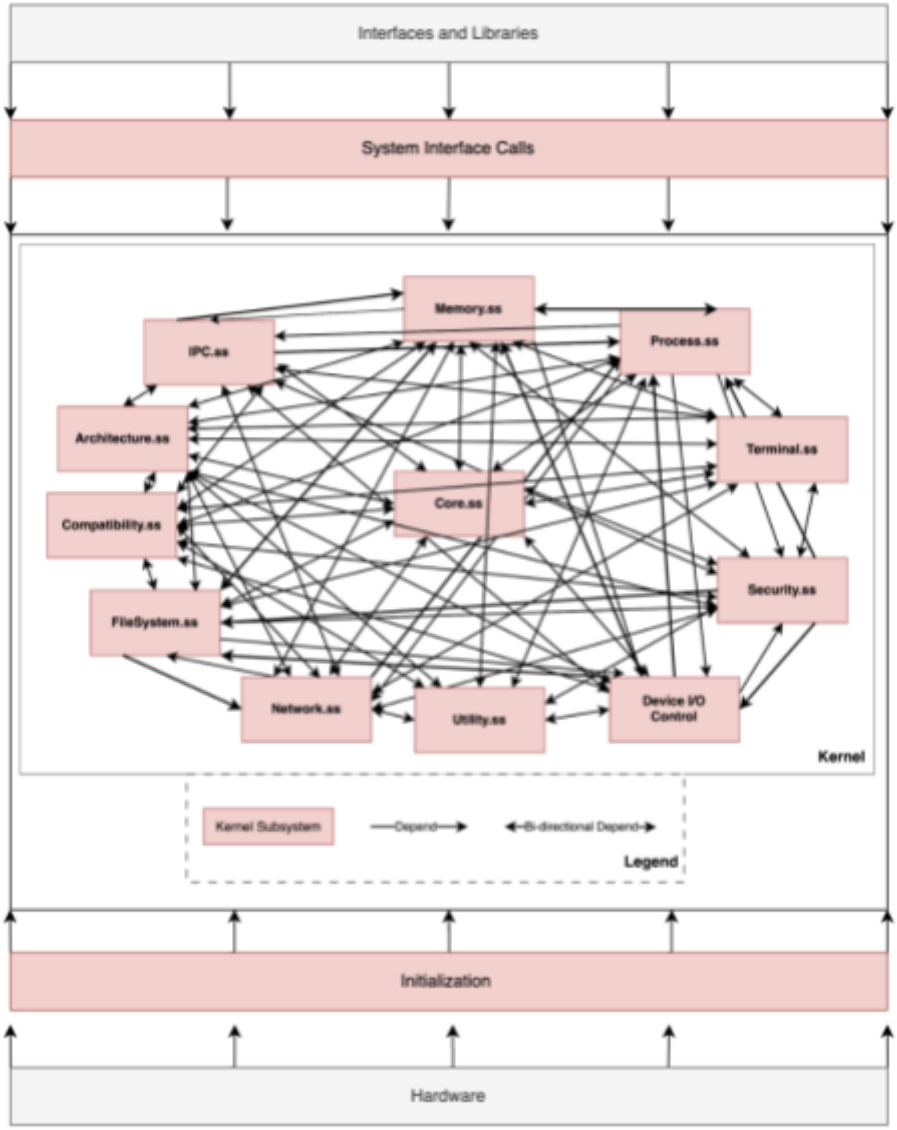


Figure 2 - The layered architecture of FreeBSD and the Monolithic Architecture of FreeBSD's kernel

As revealed in the conceptual architecture, the overall structure of FreeBSD's kernel still follows the monolithic architecture. Communication to the kernel is done through the System Call interface that sits between the users/interfaces/libraries and the kernel itself which follows the layered architecture style (McKusick et al., 2015, 45). In comparing the differences between the conceptual and concrete architecture, their monolithic architecture is the same. However what differentiates them is that the number of subsystems and the number of dependencies between these subsystems is increased in comparison to the conceptual architecture.

Further analysis of the kernel's concrete architecture revealed a central subsystem, 'core.ss', in which all the other subsystems have a dependency on. This revelation gives us potential clues that it might be the case that the kernel itself uses the repository style architecture. However, further inspection reveals this can't be the case because the subsystems themselves have dependencies on each other, and that it is likely the case that 'core.ss' is a subsystem that contains the most commonly used processes and routines the subsystems use. This leads then to two architecture styles that FreeBSD uses. Communication to and from the kernel uses the layered architecture style, and that the kernel itself uses the monolithic architecture style (The FreeBSD Documentation Project, 2012). Services of the kernel are as follows: Memory, IPC, Process,

Architecture, Compatibility, FileSystem, Network, Utility, DeviceIOControl, Security, Terminal, Process and the all-important Core.

Kernel Subsystems

Memory Management

FreeBSD's memory management subsystem is responsible for the virtual memory abstraction, handling memory pressure, and kernel memory allocation. The system implements this virtual memory abstraction through a set of system calls, such as mmap, msync, mlock, and munlockl, which allow processes to manipulate virtual memory mappings. The system also handles page faults by enforcing memory protection, copy-on-write, tracking page dirty state, and page-in from swap/filesystem/device memory. Furthermore, it handles memory pressure by reclaiming pages, approximating LRUs, and maintaining the page queue. Also, it holds miscellaneous tightly coupled subsystems like FS buffer cache, and POSIX shared memory. The key characteristics of the FreeBSD Virtual-Memory System are that it offers a division between machine dependent and non machine dependent components, it supports the sharing of memory between processes, it supports multiprocessor systems, and is based on object oriented design principles (Johnston, 2017) (*Chapter 7. Virtual Memory System*, 2021).

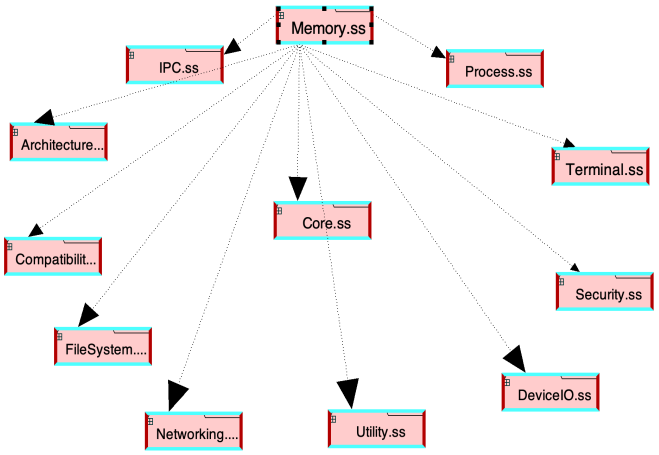


Fig. 3. ***Memory.ss forward query on LSEdit***

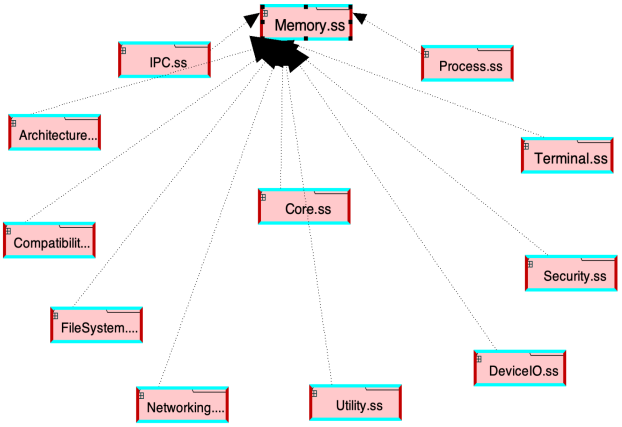


Fig. 4. ***Memory.ss backward query on LSEdit***

To function effectively, the Memory subsystem depends on other subsystems including IPC, Network, Terminal, Security, Device IO, Compatibility, Process, Utility, and File Management. Figure 3 demonstrates the forward query for Memory.ss, which indicates the subsystems it depends on. Figure 4 is the backward query, which depicts the subsystems that depend on the Memory subsystem.

The IPC subsystem provides mechanisms like shared memory, and message passing that allow processes to communicate memory usage, which illustrates this dependency relationship (Hameed, n.d.). Similarly, the Network subsystem is responsible for providing access to remote memory, and memory-mapped network devices. This enables processes to utilize memory outside of the local machine. Furthermore, the File Management subsystem depends on the Memory Management subsystem to provide virtual memory mappings and efficient memory management as it requires

memory allocation and deallocation (McKusick et al., 2015). Figure 3 demonstrates the Matrix view of the memory subsystem, which includes the files of each subsystem that interact with the Memory.ss.

Use Cases

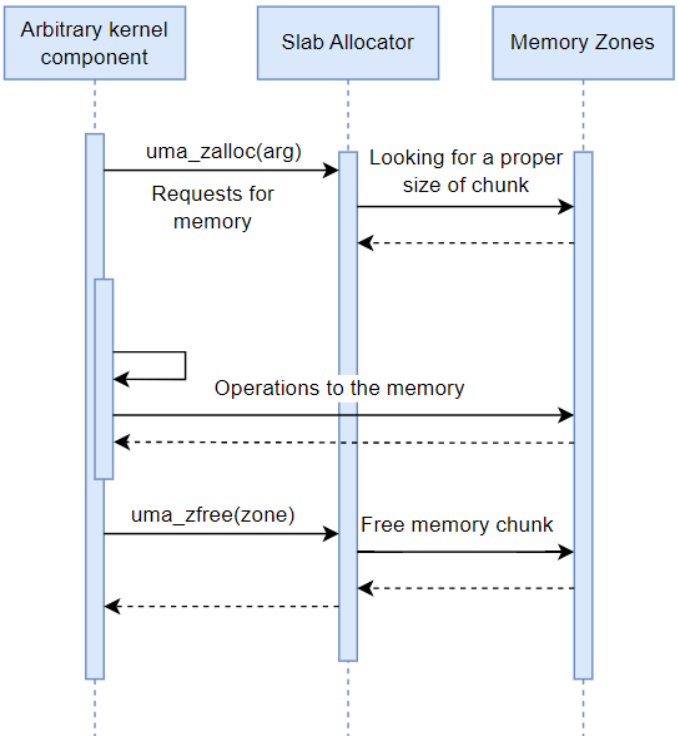


Fig. 6. ***Memory Allocation Through Slab Allocator***

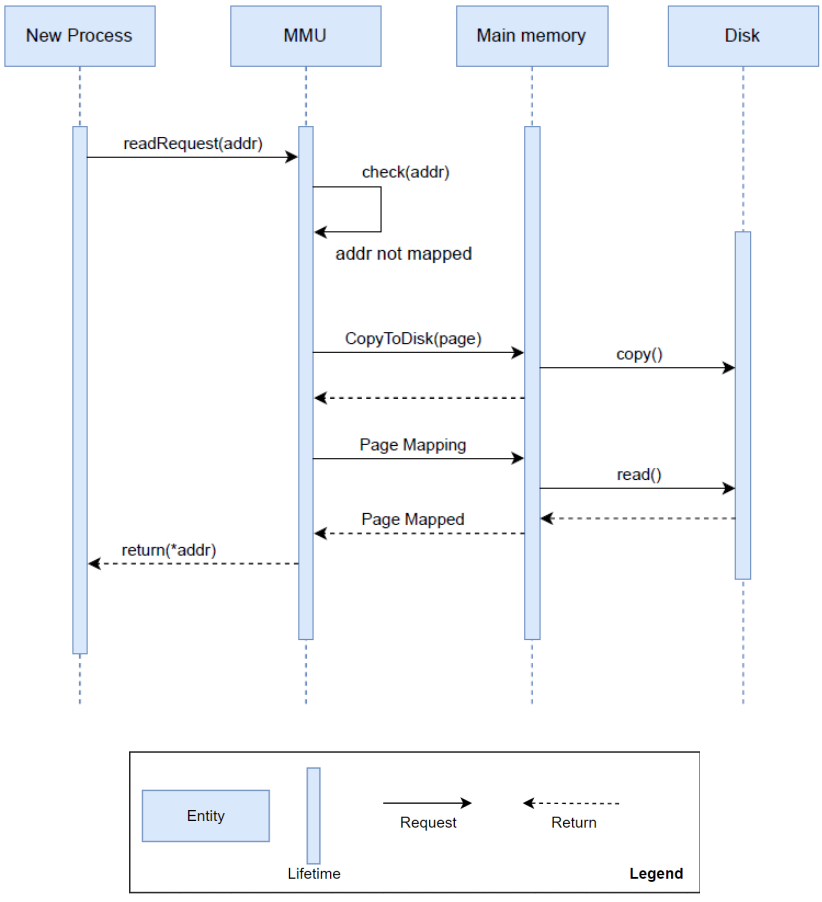
Slab Allocator

In FreeBSD, kernel data structures such as process control blocks can be inefficiently managed by the malloc() function as these structures tend to be large. FreeBSD uses a Kernel Zone Allocator (Slab Allocator) to allocate fixed-size blocks of memory from a reserved memory pool, which is created for every type of structure that will use the allocator. Furthermore, the Slab Allocator improves cache performance by ensuring that allocation of each structure begins on a page boundary, which keeps linkage pointers in the blocks on the same line in the hardware cache. This reduces cache misses during list traversals. Also, the dedicated pool of memory for each structure means that substructures only need to be initialized once, improving initialization performance (Rosenberg, 2015) (*Chapter 7. Virtual Memory System*, 2021).

Page-Fault Dispatch

When a process accesses a page that is not currently in memory, a page fault occurs. The kernel's page-fault handler is triggered and it performs four steps (*Chapter 7. Virtual Memory System*, 2021):

1. Finds the process's virtual memory space.
2. Locates the relevant segment within the memory map.
3. Converts the virtual address to an offset within the object.
4. Requests the object to allocate a memory page

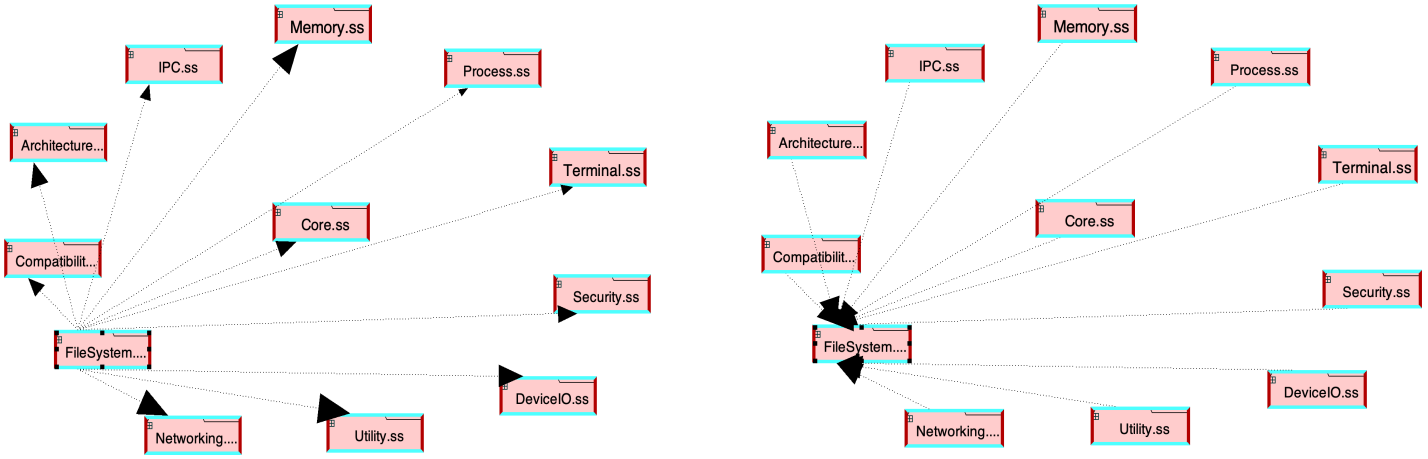


Process Management

Process is managed by the kernel, but the user can still access it by the user mode, and when the user is trying to access the process management, their operation will be non privileged. There is a jail system that can be used for isolation and accessibility, it will create an isolated environment for different processes which can make sure that they will not impact each other and for safety reasons (The FreeBSD Documentation Project, 2012).

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. 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 (McKusick et al., 2015).

File Management



The File Management System in FreeBSD depends on IPC, Networking, Terminal, Memory and other subsystems in order for it to function as an operating system successfully. For example, the IPC in FreeBSD is responsible for and provides mechanisms such as pipes, message passing. For the file management system to be working effectively, it needs to be able to utilize these features in order to coordinate file access and sharing between multiple processes. Another important example would be the NFS (Networking File system) utilizing the Networking system system in order to access files over the network. Similarly to how the File Management subsystem depends on other subsystems for its functionalities, other subsystems are also relying on File System as most subsystems are heavily reliant on files to store very important information (McKusick et al., 2015).

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 (McKusick et al., 2015).

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. Similar to other subsystems in the operating system, the Device I/O has other subsystems that it is dependent on as well as other subsystems that are dependent on it for many functionalities. For example, the Device I/O subsystem in FreeBSD is dependent on the Memory Management subsystem because this subsystem is responsible for the allocation of the system's virtual and physical memory. When you need to install different device drivers and buffers in the system, it has to depend on the memory management subsystem for the allocation of this memory (McKusick et al., 2015).

Security

The security subsystem in FreeBSD is a critical aspect of the operating system's design. Its primary goal is to ensure that only authorized users have access to specific data and functions. To achieve this, the security subsystem utilizes a variety of different functions and variables that work together to enforce multilevel security policies such as Mandatory Access Control (MAC) and Discretionary Access Control (DAC) (McKusick et al., 2015).

Within the security subsystem, there is a cryptography subsystem. The cryptography subsystem provides an OpenCrypto framework that allows cryptographic hardware drivers to register with the kernel and utilize its capabilities .

Utility

The utility subsystem in FreeBSD is a crucial component of the operating system that provides various utility functions to the kernel. It includes a variety of tools and files that are essential to the system's operation. Within the utility subsystem, there is a *debug* and *external* subsystem. The debug subsystem includes tools and utilities for debugging kernel code and identifying and fixing issues that may arise during system operation. The debug subsystem can be especially useful for developers working on the FreeBSD Kernel, as it allows them to diagnose and resolve issues with the system. The external subsystem contains third-party code used by the FreeBSD system, such as drivers and libraries. By incorporating third-party code into the operating system, FreeBSD can leverage the work of other developers and ensure that the system has access to a wide range of tools and capabilities (The FreeBSD Documentation Project, 2012).

Networking

The networking subsystem is the connection part in FreeBSD,it's job is to connect to the ethernet and exchange data with the users, in the networking subsystem, there are different files which are responde for different usage, for example the core networking coe, the graph-based networking subsystem, the wireless networking , and the protocols like ipv4, ipv6, Netipsec, etc (FreeBSD, 2022).

In order to allocate the packet into the right memory space, the mbuf will be used at this time, due to the packet size is changing - headers are added and removed, it may be necessary to split users data into segments, collect them back when reading into a larger user process buffer etc. mbuf is a data

structure with a small fixed size, and it combines the packet data and the linklist. And as the figure shows, the mbuf is in the kernel which is the core subsystem, and can be used for networking and memory subsystems (FreeBSD, 2022).

Interprocess Communication (IPC)

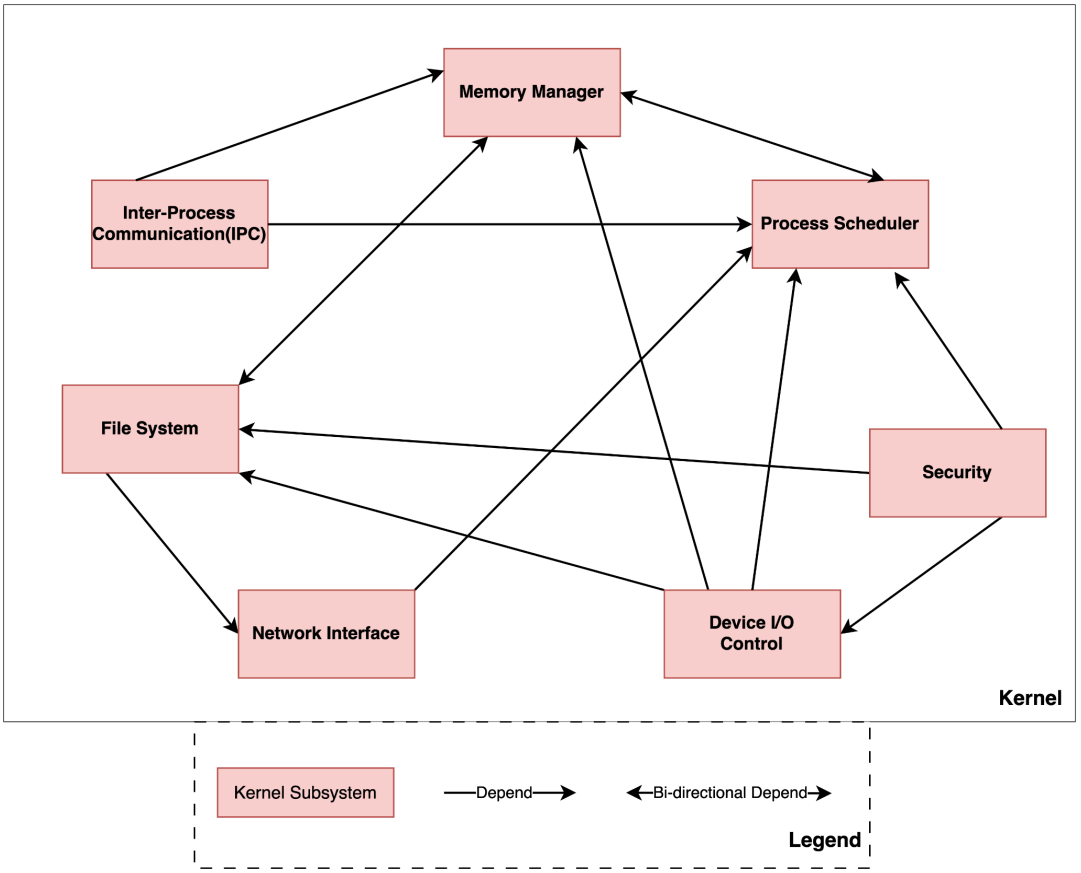
In looking at the interactions the IPC subsystem creates with other subsystems, it allows us to infer how the IPC subsystem functions with respect to the other subsystems. The IPC subsystem, like all subsystems, are connected to the central core.ss, implying that the IPC subsystem uses common routines and processes that other subsystems also use. The interaction between IPC subsystem and Security subsystem implies that the security subsystem may be incharge of making sure the socket(s) is behaving correctly as it is implemented based on a protocol. The interactions between the IPC and Memory subsystems are also important as it demonstrates how the socket is an abstract data- type that is implemented and requires the allocation and deallocation of memory. The non-existent interaction between the IPC and the Terminal subsystem could be an indication that the Terminal interface is not allowed to directly enter the IPC subsystem and create connections within it, which also applies to the Process subsystem. This of course does not mean that they do not at all interact, there likely is an indirect interaction between these subsystems by using the other subsystems as a path towards the IPC subsystem. For example, because the arrows indicated are by bi-directional, and that the core subsystem connects to all subsystems, the Process subsystem may have an indirect interaction with the IPC subsystem through the core subsystem and vice-versa (*System Call*, n.d.) (McKusick et al., 2015, #).

Concurrency

In order to make sure that concurrency can function properly, which means when two users are trying to access the same file at the same time, or when two processes are trying to access the same memory area, no error will occur. So there will be a “lock” to lock the file or the memory area, and unlock them in sequence when the previous process finishes or the interrupt happens.

For this process it’s mainly about the interactions between the core and the memory subsystem, there two files in the core subsystem or the kernel which are the lockf and mutex, these two functions are two of the most important functions that the kernel has, mutex is one for making sure that the cups can process the multiple threads at the same time, and make sure that when the cpus reach their upper limit, the new threads will not influence the current thread. For lockf, it is a protection mechanism to make sure that when a user or process is using the memory, and this part of using memory will not be accessed by another user or process, so the conflict situation will not happen. There is one example of lockf, when too many users are trying to connect to the server and when the server is overloaded, only the first few users can actually connect to the server and the later user will get an error lockf. And by comparison of these two functions on the LSEdit, lockf has a wider field compared to mutex, mutex is only for arranging threads, which can only be used in core subsystem and the forward query only contains the memory and utility subsystem. However, for lockf, it can be used both in core and memory subsystem, and the forward query contains networking, filesystem, device I/O, utility and also external subsystem (The FreeBSD Documentation Project, 2012).

Reflexion Analysis



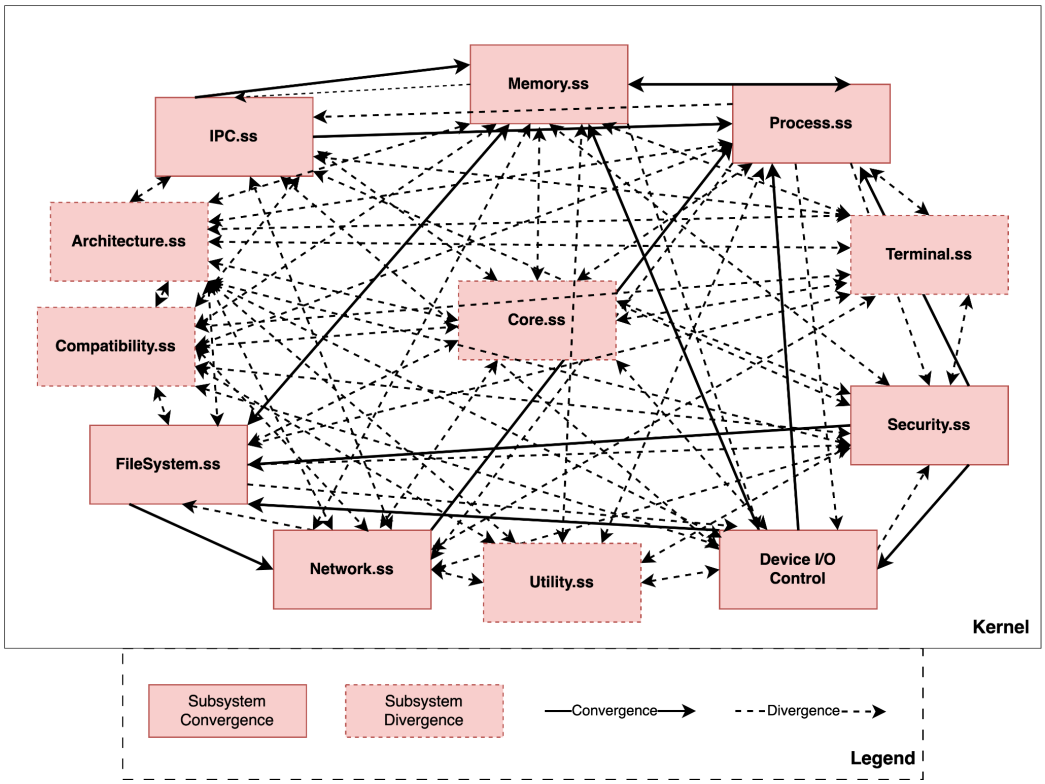


Fig. 11. ***Conceptual view of Memory.ss***

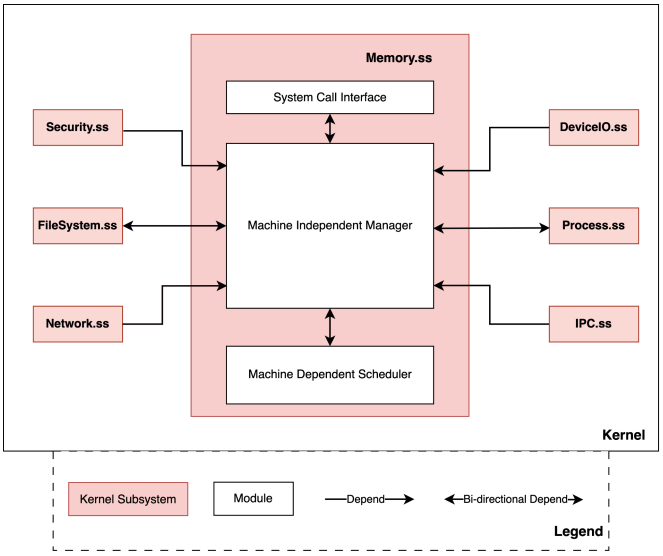
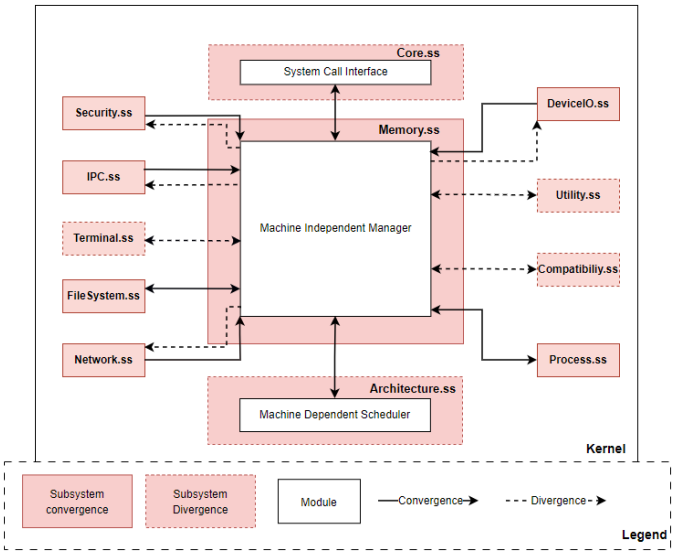


Fig. 12. ***Reflexion Model of Memory.ss***



The source code allowed the identification of the different Subsystems FreeBSD's kernel is made of. Using a tool called Understand to map the file dependencies in addition to using LSEdit, we were able to have a complete map of the subsystems. The mapping of the subsystems also gave us the ability to see the number of interactions a subsystem creates with the other subsystems. This stemmed into a new diagram using this graph editor to create a graph of interactions between subsystems. The table below shows the number of interactions each subsystem has with the other subsystems.

Memory to IPC	25	IPC to Memory	21
Memory to Architecture	49	IPC to Architecture	36
Memory to Compatibility	9	IPC to Compatibility	6
Memory to FileSystem	70	IPC to FileSystem	11
Memory to Networking	329	IPC to Core	8
Memory to Core	125	IPC to Networking	60
Memory to Utility	459	IPC to Utility	298
Memory to DeviceIO	161	IPC to DeviceIO	17
Memory to Security	3	IPC to Security	19
Memory to Terminal	4		
Memory to Process	10		
Architecture to Memory	1499	Compatibility to Architecture	123
Architecture to IPC	27	Compatibility to IPC	3
Architecture to Compatibility	217	Compatibility to Memory	146
Architecture to FileSystem	145	Compatibility to Core	15
Architecture to Networking	3300	Compatibility to FileSystem	54
Architecture to Utility	8363	Compatibility to Networking	249
Architecture to DeviceIO	3419	Compatibility to Utility	1171
Architecture to Security	46	Compatibility to DeviceIO	254
Architecture to Terminal	3	Compatibility to Security	8
Architecture to Process	1	Compatibility to Terminal	1

Architecture to Core	98		
FileSystem to Memory FileSystem to IPC 10 FileSystem to Architecture 123 FileSystem to Compatibility 26 FileSystem to Networking 798 FileSystem to Utility 1673 FileSystem to DeviceIO 144 FileSystem to Security 34 FileSystem to Terminal 6 FileSystem to Process 3 FileSystem to Core 34	264	Networking to Memory Networking to IPC 100 Networking to Architecture 1326 Networking to Compatibility 872 Networking to FileSystem 251 Networking to Utility 4598 Networking to DeviceIO 998 Networking to Security 111 Networking to Terminal 24 Networking to Process 35 Networking to Core 371	176
Utility to Memory 97 Utility to IPC 4 Utility to Architecture 359 Utility to Compatibility 115 Utility to FileSystem 211 Utility to Networking 6791 Utility to DeviceIO 1466 Utility to Security 24 Utility to Terminal 2 Utility to Process 3 Utility to Core 115		DeviceIO to Memory 662 DeviceIO to IPC 8 DeviceIO to Architecture 2522 DeviceIO to Compatibility 1999 DeviceIO to FileSystem 93 DeviceIO to Networking 6636 DeviceIO to Utility 19953 DeviceIO to Security 218 DeviceIO to Terminal 2 DeviceIO to Core 97	
Security to Memory 25 Security to IPC 2 Security to Architecture 35 Security to Compatibility 16 Security to FileSystem 19 Security to Networking 277 Security to Utility 578 Security to DeviceIO 54 Security to Terminal 1 Security to Core 7		Terminal to Memory 7 Terminal to Architecture 28 Terminal to FileSystem 7 Terminal to Networking 33 Terminal to Utility 124 Terminal to DeviceIO 15 Terminal to Core 2	
Process to Memory 12 Process to to Architecture 4 Process to Compatibility 4 Process to FileSystem 2 Process to Networking 13 Process to Utility 178 Process to DeviceIO 4 Process to Security 4 Process to Core 2		Core to Memory 196 Core to IPC 20 Core to Architecture 63 Core to Compatibility 54 Core to FileSystem 60 Core to Networking 247 Core to Utility 1717 Core to DeviceIO 192 Core to Security 33 Core to Terminal 5 Core to Process 6	

Data Dictionary

Command Line Interface (CLI) - A text based interface used to communicate with the system[1]

Conceptual Architecture - The software architecture derived from reading the documentation, reference architecture, developer experience with other software and from questions answered by domain experts

Concrete Architecture - The software architecture derived from creating a dependency graph based on the source code of the given software

Concurrency - The ability to do multiple computations at the same time[1]

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

Graphical User Interface (GUI) - An interface that uses graphical icons, menus etc. to communicate with the system[1]

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.[1]

Transmission Control Protocol (TCP) - One of the protocols in the Internet Protocol suite used for its high reliability of transmission.[1]

User Datagram Protocol (UDP) - One of the protocols in the Internet Protocol suite used for time dependent transmissions such as video streaming.[1]

Universal Memory Allocator (UMA) – provides an efficient interface for managing dynamically-sized collections of items of identical size, referred to as zones.

Memory management unit (MMU) – Keeps allocated references of memory, and maps virtual memory pages with physical memory pages.

Conclusions

In conclusion, in this project the first step is to determine the subsystem , modify raw.ta, and then get the container, and use LSEdit to show the dependency of different subsystems by graph. Then use the graph to analyze the dependency of different subsystems, and mainly focus on the memory management. By analyzing all of the subsystems and trying to find the connection between the these subsystems and memory management or other subsystems, then we analyze these relation and

[1] Silberschatz, A., Gagne, G., & Galvin, P. B. (2018). *Operating System Concepts, 10e EPUB Reg Card Abridged Print Companion Set*. Wiley.

understand how does these subsystem interprocess with each other, at last we use the cases which are memory allocation and page fault handling to show how the memory management work.

Lessons Learned

The process of generating a hierarchical decomposition of a system requires significant human involvement, and is an iterative process. It was essential to adjust our conceptual model based on our understanding of the concrete architecture in order to accurately compare the two. Furthermore, we gained a concrete understanding of the implementation of FreeBSD components and how they interact with each other. We employed the Software Reflexion Framework to identify convergence, divergence, and absences between our concrete architecture and our conceptual one, in which we found that concrete implementations almost always use different mechanisms from the conceptual. As such, our understanding from the concrete architecture should be utilized to refine the conceptual model.

References

Chapter 7. Virtual Memory System. (2021, December 11). FreeBSD Documentation. Retrieved March 10, 2023, from https://docs.freebsd.org/en/books/arch-handbook/vm/

FreeBSD. (1993, 01 01). *Tags*. FreeBSD Source Code. Retrieved March 10, 2023, from https://github.com/freebsd/freebsd-src/tags

FreeBSD. (2022, April 12). *Memory management FreeBSD kernel*. The FreeBSD Forums. Retrieved March 10, 2023, from https://forums.freebsd.org/threads/memory-management-freebsd-kernel.84787/

The FreeBSD Documentation Project. (2012, 01 01). *FreeBSD Architecture Handbook*. FreeBSD Documentation. Retrieved March 10, 2023, from https://docs.freebsd.org/en/books/arch-handbook/book/

Garlan, D., & Shaw, M. (1994, 01 01). *An Introduction to Software Architecture*. Computer Science. Retrieved March 10, 2023, from https://userweb.cs.txstate.edu/~rp31/papers/intro_softarch.pdf

Hameed, S. (n.d.). *TCP/IP layers and their functions*. Linux Hint. Retrieved March 10, 2023, from https://linuxhint.com/tcp-ip-layers-and-functions/

Johnston, M. (2017, 01 01). *Memory Management in FreeBSD 12.0*. FreeBSD Presentations and Papers. Retrieved March 02, 2023, from https://papers.freebsd.org/2017/bsdtw/johnston-memory_management_in_freebsd_12.0/

McKusick, M. K., Neville-Neil, G. V., & Watson, R. N. M. (2015). *The Design and Implementation of the FreeBSD Operating System*. Addison Wesley.

Rosenberg, B. (2015, October 01). *Introduction to Virtual Memory*. Introduction to Virtual Memory. Retrieved March 10, 2023, from https://www.cs.miami.edu/home/burt/learning/Csc521.101/notes/virtual-memory-notes.html

System call. (n.d.). Wikipedia. Retrieved March 10, 2023, from https://en.wikipedia.org/wiki/System_call