Integrating a Flexible File Abstraction into the Linux Kernel

BY

POLLY PLANINSEK
BS, University of Illinois at Chicago, Chicago 2021

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2021

Chicago, Illinois

Defense Committee:

Prof. Xingbo Wu, Chair and Advisor
Prof. Jakob Eriksson, Computer Science
Prof. William Mansky, Computer Science

ACKNOWLEDGMENTS

I am deeply grateful to my husband, Theodore Planinsek. Without your constant support I would not be where I am today.

I would like to express my sincere gratitude to Professor Xingbo Wu, working with you throughout this Master's Thesis I have learned a great deal. I will forever be grateful for their unwavering support and belief in me.

A special thanks to Professor Jakob Eriksson and Professor William Mansky for their insightful comments and critiques. I am grateful to have had both of you in my committee.

PP

TABLE OF CONTENTS

LIST OF FIGURES

SUMMARY

Performance of data systems are critical in today's world due to the large amount of data collection being done. Database management systems (DBMS) employ structured files and large amounts of high cost in-place insertions and removals. Traditionally, applications use extra layers of indirection to offset these high-cost operations, but this only adds more complexity and higher access costs. In the face of this, we utilize a flexible address space located in user space called FlexFile. FlexFile utilizes in-place updates for arbitrary-sized data, which creates a more efficient way for insertions and removal operations to be performed.

FlexFile was originally implemented in user space. The user space mock-up of the flexible file abstraction, FlexFile, was successful in showing that the flexible file system was possible. FlexFile in kernel space is the next step in the research process. This allows researchers to visualize the possibilities in the Linux kernel, if it can be compatible with the traditional file system calls.

This research re-implements FlexFile in the Linux kernel and evaluates performance comparisons between both. First, the implementation overview is presented to efficiently realize the user space needs of a traditional file system interface. Next, the current and future implementation possibilities are discussed to perfect and expand this development. Last, both implementation results are compared and contrasted to show how specific locations within a system can be leveraged to produce different results within the FlexFile implementation.

CHAPTER 1

INTRODUCTION

Whether it is customer information, world wide web usage, bank account data, or product SKUs, data is prevalent in the modern world. With all this data, data collection is highly sought after by researchers, businesses, government agencies, etc, [1]. Database systems store, sort and provide access to this data. Databases are traditionally managed by database management systems. The idea of a one all-purpose database is impossible. Therefore, each database may serve a different purpose or different preferred feature, [2]. Oracle, MongoDB [3], RocksDB [4] [5], and LevelDB [5] are some of the many database systems used today. The commonality between all database management systems is their attempt at providing efficient ways to access data by retrieving, inserting, deleting, and updating data, [4].

Database management systems operate on files to store persistent data. The data is typically sorted into files to provide ease of access to database management systems, [2]. To maintain the organization of the file, some file systems will perform rewrites of existing data when in-place updates are committed to a sorted file, [6]. Inserting and deleting in sorted files is time consuming and difficult, [7]. RocksDB utilizes an LSM-tree. LSM-trees, [8], update their sorted-string table by adding new insertions into segment files. LSM-trees also perform a compaction operation. Compaction is important to prevent excessive amounts of segment files, as well as clean up and maintain order of these segment files. LSM-tree's compaction operations will combine and maintain order of the content in segment files. The downside is that it will in return lead to

many copies and rewrites of data, leading to a high write amplification, [4]. Structured files are maintained to where they are easily searched. Expensive in-place insertions are performed on structured files to maintain order, [9]. Applications utilize extra indirections to keep their data sorted, leading to them paying a high cost. When the flexible file abstraction, FlexFile, was originally introduced the goal was to delegate the data organization jobs to the storage layer and removing it from the application layer, [10].

This paper will take the flexible file abstraction, FlexFile which was introduced into user space [10], and reimplement it now in the Linux kernel. Moving FlexFile into the Linux kernel the overall structure of FlexFile as well as FlexTree will be maintained. This paper will address the compatibility issues the user space implementation faced and the resolutions to convert FlexFile into a file system that can utilize the Linux kernels system calls. This paper will briefly explain the purposes of FlexTree and FlexFile but will mainly focus on the conversion of moving the user space implementation into the Linux kernel space.

CHAPTER 2

FLEXTREE

FlexTree implementation contains a modification of a B+-Tree. B+-Trees consist of a root node, internal nodes and leaf nodes. B+-Trees are a modification of B-Trees [11]. Extent data is contained in leaf nodes. Extents contain information on its offset, length and physical address. Internal parent nodes will contain the pivot points to these leaf nodes. B+-Trees are efficient in retrieving and storing content, traditionally block-oriented. For example, file systems, [10].

FlexTree re imagines the B+-tree but with additional data members to handle addressing with byte granularity. The new address metadata representation will allow shifting of extents without high cost penalty. Pivot addresses previously seen in the B+-Tree structure are now considered partial offsets. Partial offsets allow calculating the physical addresses with O(log(N)) time complexity, where N is the number of extents. Detailed outline of the structure, as well as all possible operations, are addressed in later chapters. [10].

CHAPTER 3

FLEXFILE

FlexFile is the storage engine that will provide user processes with persistent data storage. FlexFile introduces users with the possibilities of having in-place inserts and removals with byte granularity. The in-place inserts and removals will also support the feature of not leaving holes in the file, without all the needed in-directions found throughout other file systems. FlexFile maintians the organization of the system with the use of persistent storage files. These files are regarded as a data file and a logical log file. FlexFile also allows a pathway to the FlexTree structure. The FlexFile user space implementation directed all locking responsibilities to the user programs, for multiple processes. Detailed outline of the structure, as well as all possible operations, are addressed in later chapters. [10].

CHAPTER 4

FLEXFILE IN USER SPACE

4.1 Purpose of flexible file abstraction in user space

A flexible file abstraction is needed for applications to have a light-weight option for in-place insertions and removals in sorted files. Linux provides the use of system call fallocate, an operation that will perform inserts and removals on files with a supported filesystem. The downside of this operation is the condition of block alignment, [12]. These Linux file system operations come at a high cost. This flexible file abstraction will support byte granularity in-place insertions and removals at lower cost, [10]. FlexFile is introduced into the Linux user space as a persistent storage engine solution to this ongoing high-cost problem, while keeping other typical file operations at comparable performance, [10].

User space programs operate on the flexible file abstractions, FlexFile, with a slight variation of the traditional Linux file system calls. A custom application program interface of FlexFile is provided for all interactions, [10]. The user space implementation is not yet compatible with traditional Linux kernel system calls, [13]. While this may be a downside, it is one that can be updated in future work. The development and testing of this flexible file abstractions provided freedom to engineer an implementation without all the complications found in the Linux kernel.

Interactions with FlexFile provides a path to the FlexTree. FlexFile and FlexTree will then update their persistent files accordingly to maintain the organization of its filesystem, [10].

These files allow users not only to store persistent data files but to be able to load existing FlexTrees efficiently. Figure 1 shows the layers of the system as it sits in user space.

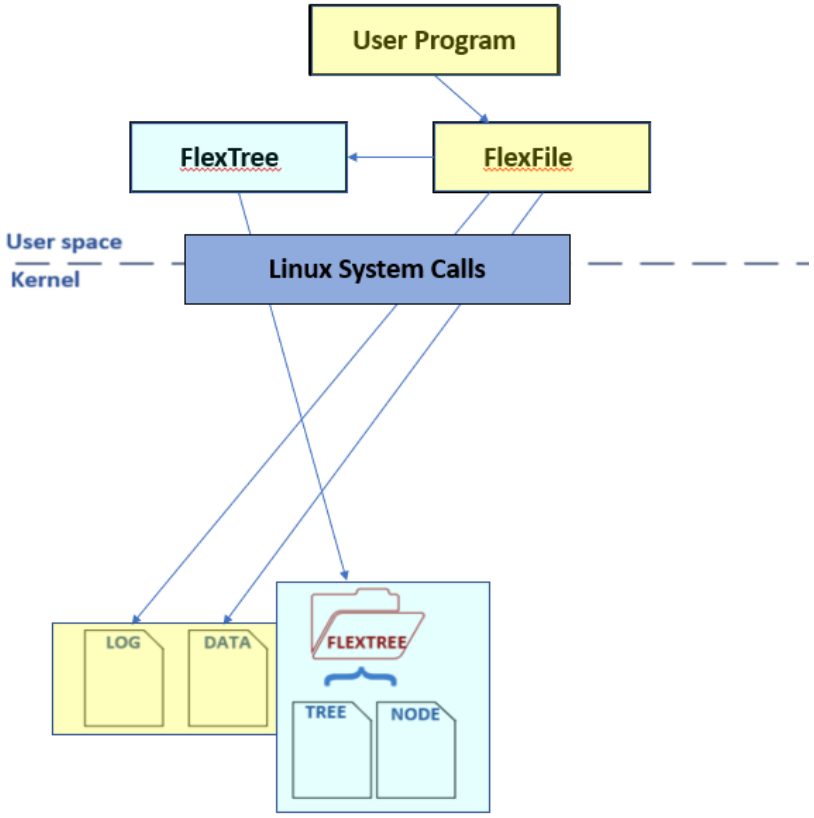

Figure 1. FlexFile User space Layout

4.2 FlexFile mockup

Implementation of FlexFile in user space faced many benefits. User space provides the ideal space to implement mockups. Mockups provide a replica of the preferred machine or structure under development, [14]. Unit testing will be provided early in the development process allowing researchers to focus on content and functionality rather than interfacing compatibility details, [15]. The FlexFile mockup provides rapid performance results to benchmark tests against other Linux file systems. Proof of concept can then be developed. Outlining needed performance enhancements, modifications, and future research advancements.

A kernel space mockup was intentionally avoided in the initial development of FlexFile. Kernel programming comes with many hurdles. Error messages are less detailed, debugging is complicated, and following control flows entail custom kernels to be built with strategic prints. Custom kernels are often avoided due to their compilation time, not to mention the endless amounts of configuration options. Professional kernel documentation is often outdated; therefore, it is best just to read the kernel code yourself. Due to these things kernel programming overall just takes more time.

More library functions are implemented in user space, [16]. Therefore, user space mockups can lead to a faster overall development time. Libraries provide functionality that has been rigorously tested and optimized, [17]. This allows researchers access to maximum performance without all the development time. User space mockups utilizing these libraries not only for faster development but to also have a more compact code base.

User space debugging tools are more convenient compared to those used in the Linux kernel. Error messages in user space are easier to read than in the Linux kernel. Monitoring, controlling, and making corrections on the code base is essential when debugging. Print statements, querying, tracing, gdb and valgrind are some techniques and tools used in user space. Print statements help target precise code segments, allowing a look into the control flow and critical variable's values. When debugging multiple processes, querying can be done to extract running processes state. Querying grants access to the kernel's command line, and each running process command line arguments, memory, and status. User space has tracing tools, strace and ltrace. Strace monitors and tampers with interactions between processes and the Linux kernel. This encompasses system calls, process state updates and signal deliveries. Ltrace displays library calls made by processes. Examining and updating data during manual execution can be done with GDB. During the execution if an error is generated, GDB supports core dumping for better analysis of the crash. Valgrind provides detailed analysis on memory usage as well as memory leaks.

The Linux virtual file system provides an interface for user space programs to communicate through. The Linux kernel provides file operations such as open, close, llseek, read, write, fsync, fallocate, etc, [13]. This allows many different filesystem implementations to coexist. When transitioning FlexFile into kernel space, FlexFile will need to be compatible with this interface. This provides users with a fast transition to employing this new flexible file abstraction. Creation of the mockup did not need to fit this interface due to FlexFile existing first in user space.

4.3 FlexFile's Application Program Interface

FlexFile in user space has the current application program interface of flexfile_open, flexfile_close, flexfile_write, flexfile_read, flexfile_insert, flexfile_collapse, etc. Each operation serves a different purpose and functionality. FlexFile will take these operations and communicate appropriately to FlexTree. FlexTree will then operate on its B+-tree structure making the appropriate actions requested by FlexFile. FlexFile and FlexTree will then make the suitable updates to their persistent files, [10].

Elaborating in greater detail, flexfile_open takes the file path, creates a new FlexFile or opens an existing one, creates a new FlexTree or loads an existing one, and then returns the struct FlexFile for users to operate on. When a user is done operating on a file, the user will use flexfile_close to close the file. Flexfile_close will then take the struct FlexFile, sync the file's data, frees up used memory and returns to the user. Users have read and write access through flexfile_write and flexfile_read. These operations will take the open FlexFile struct, user buffer, length and file offset. FlexFile will then traverse and make appropriate updates to FlexTree ultimately returning to the user with bytes written or read. Flexfile_insert and flexfile_collapse will now provide users with inserts and removals with byte-granularity without leaving holes in files, [10].

4.4 Problems and Resolutions

Examining user space implementation is not compatible with Linux traditional system calls. FlexFile's custom application program interface implementation serves as a good mockup, provides valuable feedback on performance, usage, and future research developments. Transitioning

to a kernel space implementation will be the next step for development. Modifications to the application program interface will be necessary to meet compatibility requirements for Linux kernel system calls.

Other challenges that could be encountered when porting will be removing any library calls found in the user space implementation. These calls will either be replaced, removed completely, or replicated. Memory allocations will need to be adjusted. File operations to FlexFile's and FlexTree's persistent files will no longer be able to utilize system calls. Instead, these file operations need to directly communicate through the virtual file system level functions, with some desired functionality not easily supported. Strong knowledge of the Linux kernel space will be needed for navigating these hurdles.

CHAPTER 5

FLEXFILE IN THE LINUX KERNEL

5.1 Purpose of flexible file abstraction in the Linux kernel

FlexFile's user space mockup shows a promising concept that deserves further exploration. The next step is to transition FlexFile into the Linux kernel. This paper will address the possible overhead found when transitioning into the Linux kernel, options to port and all the complications faced. Evaluations will be done on performance differences and the reasoning behind these differences.

Moving file systems into the Linux kernel can be a very long and laborious process. Great determination and extensive understanding of the Linux kernel code is needed to produce bug free and secure code. The term kernel space indicates anything that is happening within the kernel code or the "space" of the kernel code. Resources accessed or modified are typically used by a privileged user, [18]. If the file system is in user space when it crashes it does not necessarily crash the entire operating system. Moving into kernel space, if the file system crashes a greater possibility exists that the kernel could throw an exception and crash, ie. kernel panics, [19]. Overall, anything implemented in the kernel space code must meet a much higher security and safety standard. The development of FlexFile in kernel space was built and tested on a virtual machine, [20]. This provided safety to be able to reset the system easier than testing on bare

metal. Once the kernel mock-up was completed and aggressively tested, it was then moved to bare metal to gather performance results.

The benchmark test machine utilizes a 980 PRO PCIe 4.0 NVMe® SSD.

5.2 Linux kernel file system porting option - Background

A kernel module will be utilized when porting into the kernel space. Kernel modules are not independent executables. Kernel modules are object files to be linked into the kernel during run-time, [21]. The Linux kernel defines a virtual file system interface. This interface provided interacts with user space programs. Providing an abstraction for many different file systems to coexist. System calls are then shared between the different file systems. The virtual file system contains calls such as open, close, read, write, llseek, etc, [22]. Moving the flexible file abstraction into the Linux kernel has several porting design decisions to make. The Linux kernel provides options of a block device, character device, or pseudo file system.

5.3 Linux kernel file system porting option - Block Device

From the kernel's perspective, the smallest logical unit of addressing is a block. Physical devices can be addressed on a sector level, and the kernel utilizes blocks for all disk operations. Block size must meet the following criteria, must be a multiple of sectors, cannot be larger than a page and finally must be a multiple of 2. Block devices are categorized by random access to the data organized on this fixed sized block. Block devices have a higher speed performance than what is seen in character devices, [23] [24].

The Linux kernel provides a specialized application program interface specifically outlined to be used with block devices. These specialized application program interfaces are called

block device operations. Block device operations allow custom open, ioctl, etc. There is no standard read and write operation like found in character devices. If we were to claim that we would just develop our own operations with custom ioctl functions then this could lead to potential increased code bases, a more complicated call from the user, etc. Block devices do offer bdev_read_page and bdev_write_page operations. The operation bdev_read_page starts reading a page from a block device, and bdev_write_page starts writing a page to a block device. These read and write calls also communicate through the cache buffer, which is tied to the page cache, [23] [24].

With this background a major design decision can be made. Block devices communicate by sending entire blocks of data. This is bad for FlexFile. FlexFile users must be able to send arbitrary sized data to the kernel. Another thing is that block devices communicate through the cache buffer. When we first create the kernel space mockup not being heavily tied to the page cache could be beneficial.

5.4 Linux kernel file system porting option - Character Device

Character devices are known to be slower devices when compared to block devices. Character's devices, in contrast to block devices, handle data transfers of small sizes from system to device and device to system. Operations for character devices are done sequentially which is byte by byte, in contrast to block devices which are random access block by block, [25] [26].

The Linux kernel provides customised driver operations. Character devices implement system calls specific to files: open, close, llseek. write, read, etc. System calls specific for files are defined in the structure struct file_operations. When looking into the functions signature,

differences can be found. Parameters passed by user process are different from the parameters received in the character device. This is due to the operating system abstracting away some complications in the character device. One obvious change is that the file descriptor is not passed to the character device, instead a file structure is passed. This file structure is provided to the character device as another driver operation. Found inside this structure, Linux provides a data member called private data. Private data designed to be used for storing user processes specific data, [25] [26].

A design decision can now be made for FlexFile's kernel space mockup. A lot of features found in character devices that are used to interface with the user processes could be viewed as very useful in the implementation of the FlexFile mockup. Character devices can communicate byte by byte. This is essential for FlexFile due to user processes being able to send arbitrary sized data. The resolution of possibly solving many of the compatibility conflicts seen in FlexFile's user space mockup. In particular the structure file has a private data location that can pin the struct flexfile to the file descriptor, reducing the user to implement calls only with that provided file descriptor. A Character device kernel module will be the first attempt at the FlexFile kernel space mockup.

Figure 2 shows the current layout of FlexFile file system using a character device. As seen in the figure user programs will be able to utilize system calls to now communicate to FlexFile. Open was not ideal for this character device mockup, to create functionality for the open functionality, the creation of a custom ioctl can be done. FlexLib is a wrapper for custom

ioctls. FlexLib provides the user a clean interface. Other things that are useful inside FlexLib is the use of error message handling. Users are provided with detailed messages.

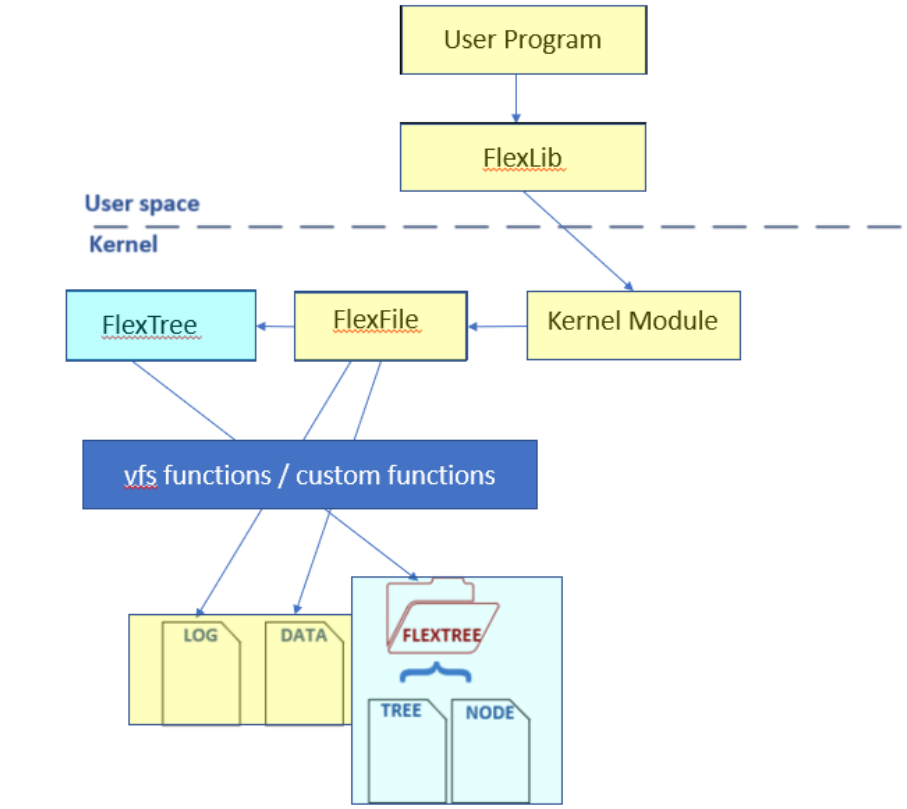

Figure 2. FlexFile Kernel Module Character Device Layout

5.4.1 Difficulties

In the development of FlexFile kernel space mockup, there were many difficulties that were faced. Debugging in the Linux kernel is difficult when comparing to what is accustomed to in user space. The gdb equivalent kgdb faces many challenges. The kernel often is compiled with optimization flags. These optimization flags cause difficulties when trying to trace code and variables in kgdb. Kgdb also comes into difficulty when tracing through context switches. This can often lead to confusion on why the code is jumping to different areas, [27] [28].

Debugging in the Linux kernel can also be done with custom print statements, [29]. Doing this inside a kernel module is quite simple, tracing code here can be done easily. Problems arise when putting prints inside the base Linux kernel code. This is a problem because it calls for recompiling a custom kernel. Recompiling the Linux kernel means that one must configure the desired modules, then compiling these modules and the kernel, then after that completes, installing the kernel image. This sounds straight forward but it is not, there are endless configuration settings. Depending on the hardware of the system, recompiling can also take a very long time. This time could be spent doing other things within the code base. When the custom kernel is finally ready for use, you may need to add more prints once you understand the path the kernel is taking, [30].

Kernel functionality that is required is not exported. This can end up with two solutions, the first one is to recompile the kernel with these now exported. That sounds like a good plan, but it is not that easy, often there is a reason it is not exported. Understanding the code is essential, many hours of studying the kernel and its functions is needed to get a good idea of

what it is trying to do. Another point is the time to recompile the kernel as previously stated. Most importantly, from that point on, with every new kernel version a custom kernel must now be maintained as well. If users want use of FlexFile on their local machine, having them create or utilize a custom kernel can lead to much difficulty. Not all users will be experienced. The best solution is to study the Linux kernel code and re-implement the desired functionality.

Recreating kernel functionality can face its own difficulties. Like previously stated, time spent studying the Linux kernel code is necessary. Let's say we run into a function not exported, we study, recreate, test, trace bugs, and confirm the correct results. For example, without studying the vfs_open [31] we see the use of exported filp_open, [31]. Seems straight forward but looking closely one can easy miss the need to assign the file descriptors back to the corresponding user processes. Vfs_close [31] has the same issue. Now let's show a function that is exported, without studying the kernel code one might miss the need of utilizing supporting calls before and after the use of the exported function. One example being vfs_mkdir [32] first need to take the path passed by the user process, convert to a kern path then passed to vfs_mkdir. This is just a small subset of what actually needs to be done with these functions. Finally let's say there is a function that is exported. Without paying close attention to the kernel code, the kernel versioning gets updated, the implantation gets altered now creating a crash in your code development. Without paying attention and understanding the code base can lead to hours to days of debugging kernel code.

Understanding proper Linux kernel space allocations is essential. Let's use kmalloc and vmalloc as examples, kmalloc is used for small allocation sizes. Vmalloc is used for large allo-

cations. Diving a little deeper, kmalloc should only be used when allocating smaller than 128K bytes. Allocations for kmalloc are contiguous physical and virtual memory. This leads to a disadvantage because in special cases if there is no longer enough contiguous physical memory it will return an error. Vmalloc handles larger allocations, it should only be used when allocating larger or equal to a page worth of bytes. This is only contiguous in virtual memory, not in physical. This leads to an advantage, if there is not enough contiguous physical memory then vmalloc will allocate memory virtually in chunks and links to the physical memory space. Understanding the implementation will show that this is slower than kmalloc due to the need of the pages obtained will then need to be remapped to their original pages when freed, [33].

FlexFile specific difficulties are found when needing to associate multiple objects to the private data, as well as the need for multiple processes accessing the same flexfile structure. This is a design decision that must be made. FlexFile will now support multiple user processes, therefore each flexfile struct must also have an associated lock. Each flexfile struct must also be accessible to multiple users, these will be the shared files not unique. First a data structure is created to hold all opened flexfiles, when a user accesses the kernel module, this structure will be searched for the file wanting to be accessed. If the file is already opened the user will get a pointer to this flexfile structure. With these unique locks are mapped in some fashion to the corresponding flexfile structure, this will used to block multiple processes from making edits to the structure or the FlexTree at the same time, leading to possible race conditions and incorrect files. All of this is then needing to be pinned to the private data, [26].

Difficulties can be encountered when picking the correct type of lock. In the Linux kernel the use of spinlocks as well as mutexlocks can be found. Upon study we see that spinlocks will crash if there is a call that sleeps while holding the lock. Mutexlocks can allow sleeping while a process is holding the lock but should not in practice. This is relevant because upon further study we see kmalloc with the flag GFP_KERNEL can sleep. FlexFile utilizes this call therefore the appropriate lock would be to use mutexlocks, [34].

Difficulties can be found in many areas of kernel programming. Debugging takes a lot of the development time. Refactoring code is necessary, so the code base does not get too large. The option to support multiple kernel versions needs to be addressed. The list could really go on further, but these highlights point out some of the major problems encountered when developing the FlexFile kernel space mockup.

5.4.2 Performance

When the mockup of FlexFile was done, extensive testing was completed to analyze its performance against other file systems, such as XFS, Ext, F2, and Btr. Results were broken down by different categories. Inserts were seen to have 180X higher throughput than Ext4,, [10]. Random insert showed high write amplification for Ext4 and XFS. Random writes and sequential writes showed equivalent performance, [10].

Read operations on all tested systems showed similar speeds. When run with multiple processes, FlexFile showed 2.8 times to 4.8 times lower throughput. The full detailed analysis can be found in the user space implementation of FlexFile, [10]. This paper will take this knowledge and reevaluate based on the new implementation done in the Linux kernel.

5.4.2.1 FlexFile kernel module vs FlexFile user space – writes

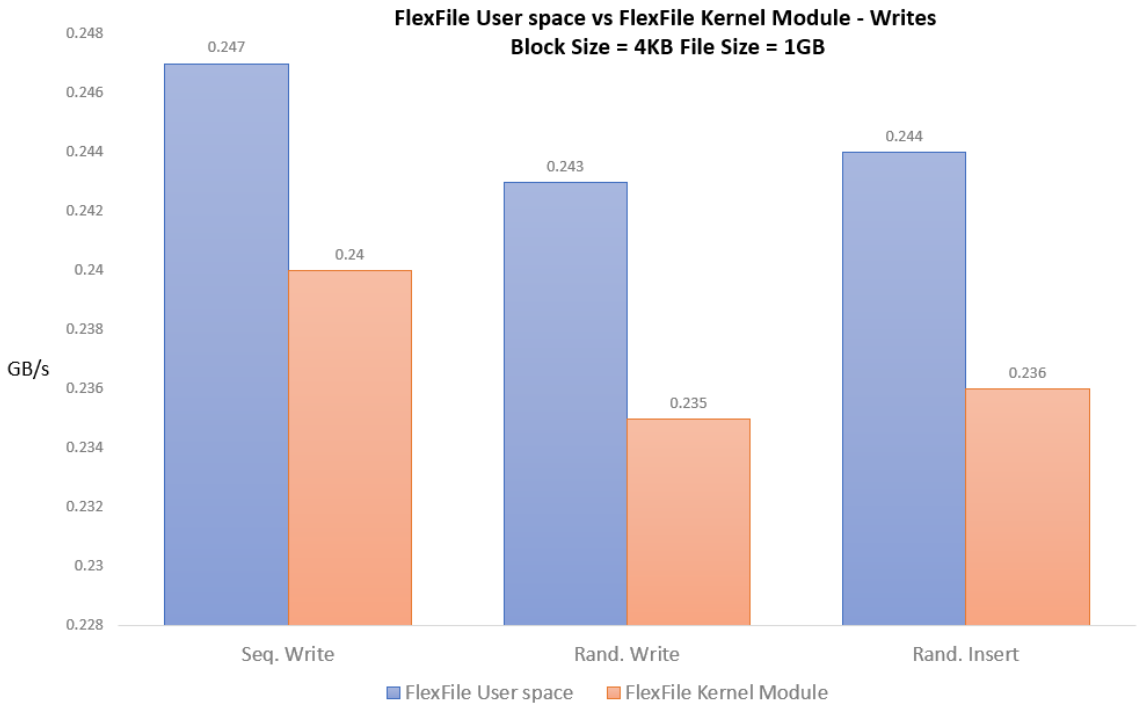

Figure 3. Performance Results for Write Operations, FlexFile User Space vs FlexFile Character Device Kernel Module

Figure 3 shows that all write operations between FlexFile in user space as well as FlexFile in kernel space are equivalent, showing the highest difference at 0.008 GB/s.

5.4.2.2 FlexFile kernel module vs FlexFile user space - reads

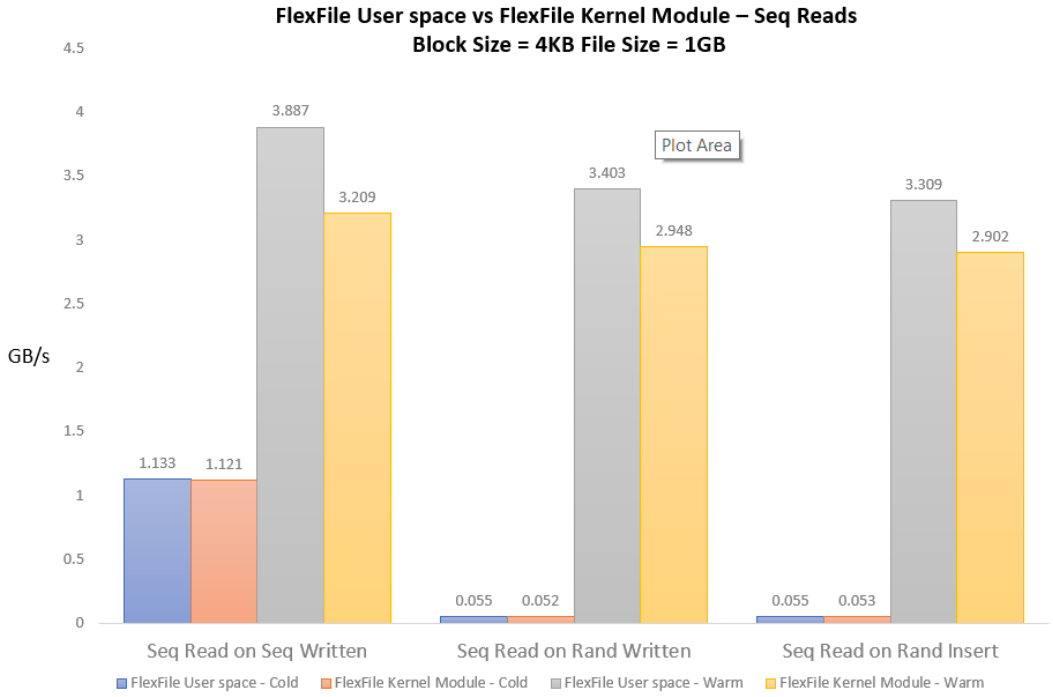

Figure 4. Performance Results for Sequential Reads, FlexFile User Space vs FlexFile Character Device Kernel Module

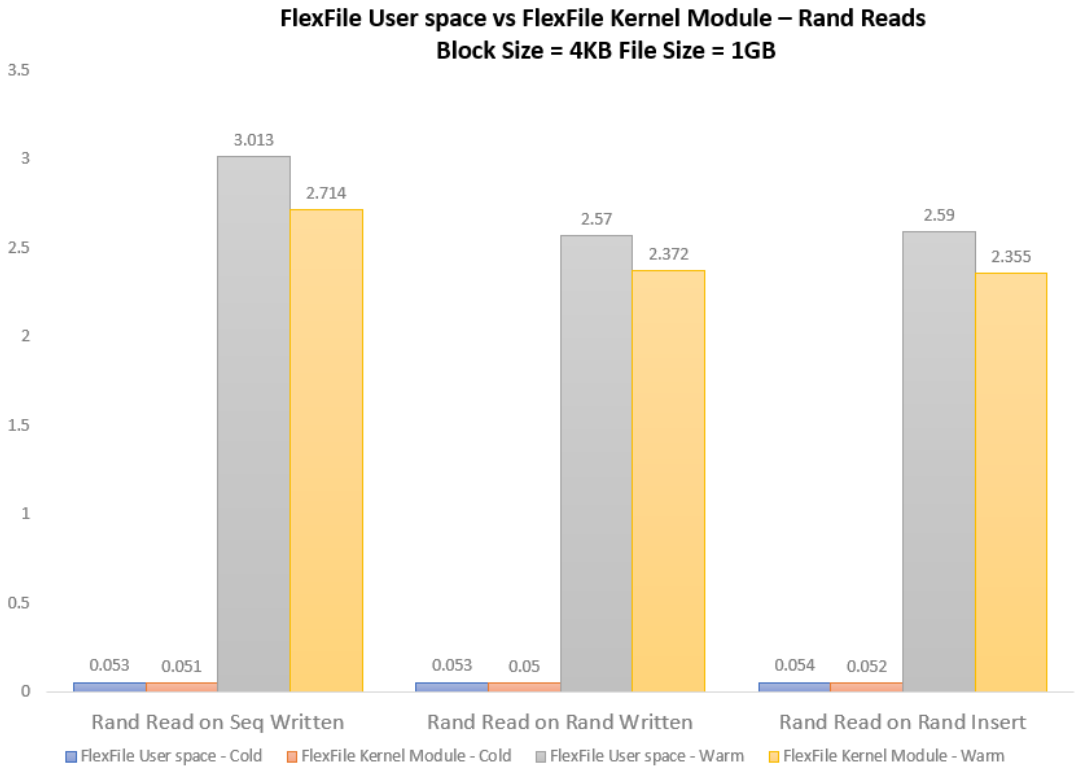

Figure 5. Performance Results for Random Reads, FlexFile User Space vs FlexFile Character Device Kernel Module

Figure 4 shows sequential read performance evaluations. All cold cache sequential reads between both implementations of FlexFile are equivalent, with the highest difference at 0.012 GB/s. Things get interesting when evaluating the warm cache operations. Performance of the FlexFile implementation have a decrease in performance areas. Sequentially written files will contain the least number of extents compared to the number is extents seen in random written and random inserted FlexFiles. This is due to the way FlexTree handles extents and logical to physical addressing. The files with the "easiest" reads will show the highest difference. This is not because of any updated to FlexTree but due to the fact of overhead from the kernel module. Figure 5 shows random read performance evaluations. The performance gets closer to FlexFile in user space due to the same fact as previously stated. As the operations get "harder", ie. more extents to filter through, the less overhead you will see from the kernel module.

The kernel modules overhead come from many things. First doing the amount of context switches being done. Previously FlexFile user space utilized the operation of pread, the character device kernel module does not support this but instead supports llseek and read. This will lead to double the context switches for read operations. Another overhead can be found with the retrieval of the FlexFile struct, this leads to a few more instructions per system call, ie. memory access, casting etc. In addition, locking is now being handled by the kernel space FlexFile. Previously locking was not considered when run due to it being the user processes responsibility, now the kernel module will handle all locking simplifying the operations for the user processes. This also leads to a few extra instructions per system call. Lastly, FlexFile user space utilized

io_uring, this is an asynchronous I/O. The kernel implementation does not utilize this because it was not available in kernel space.

5.4.2.3 FlexFile kernel module vs Other filesystems – writes

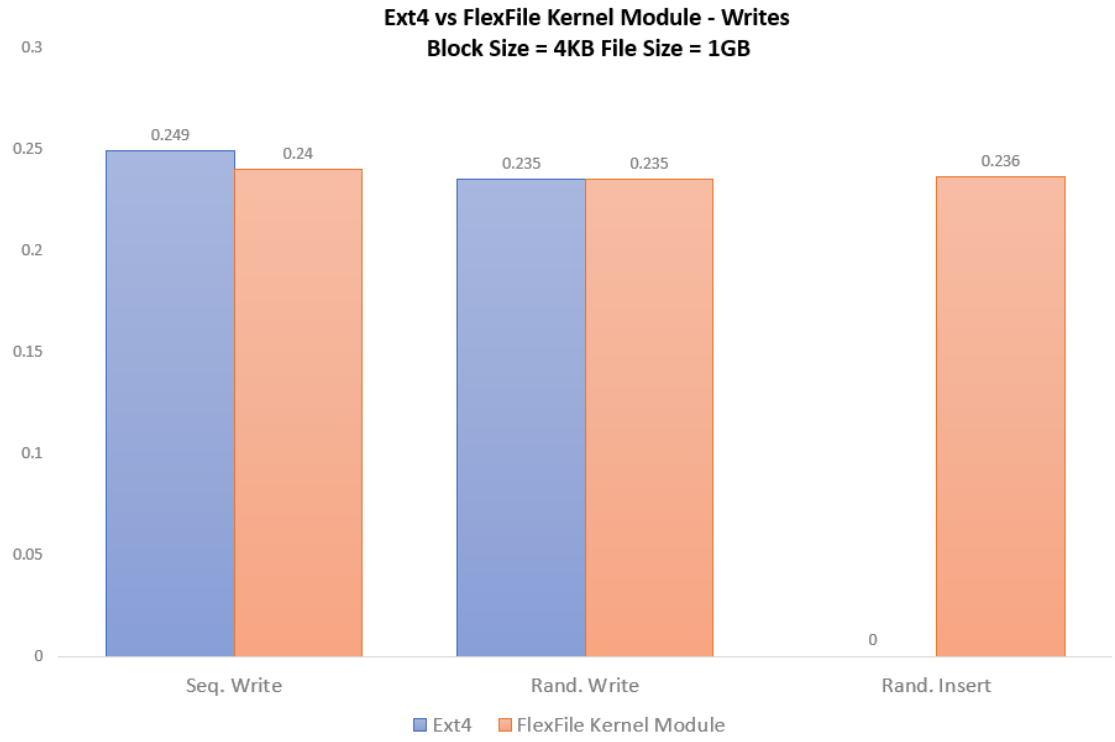

Figure 6. Performance Results for Write Operations, Ext vs FlexFile Character Device Kernel Module

Figure 6 shows comparisons between Ext4 and the FlexFile character device kernel module. Here we can see the same statements as was previously stated in the FlexFile user space implementation [1]. Since me maintained at the same performance level as the user space implementations, we remain equivalent to all other file systems tested, XFS, Ext, F2, Btr, [10].

5.4.2.4 FlexFile kernel module vs Other filesystems – reads

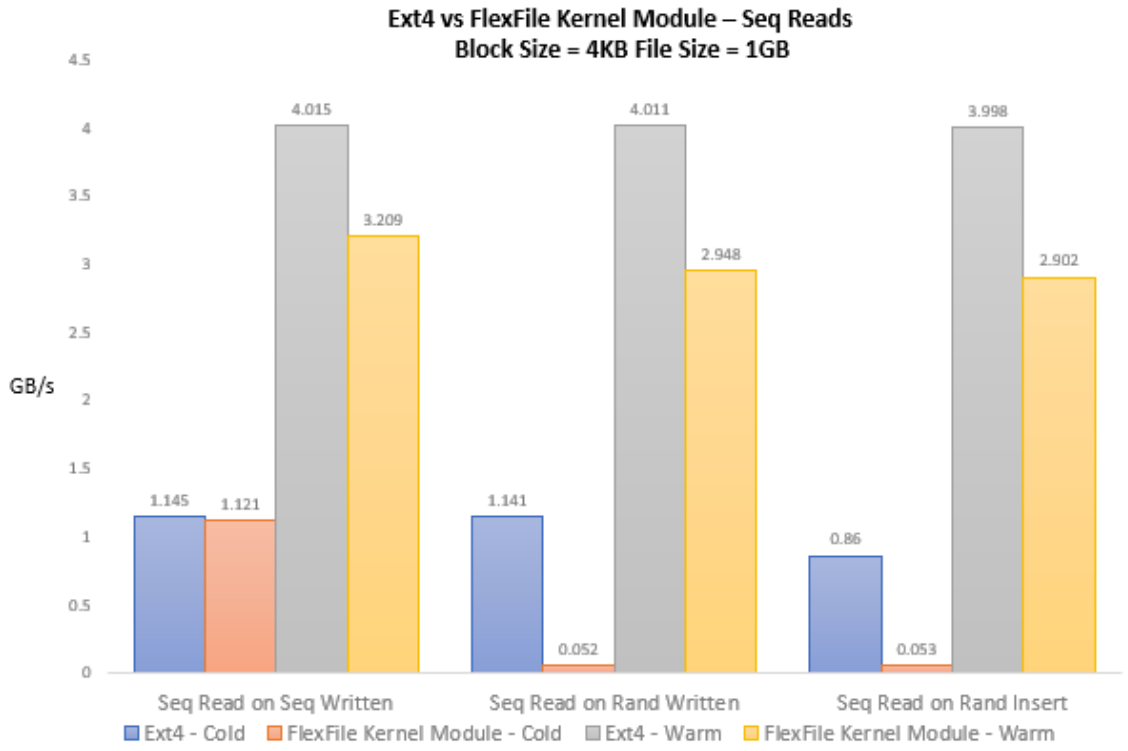

Figure 7. Performance Results for Sequential Reads, Ext vs FlexFile Character Device Kernel Module

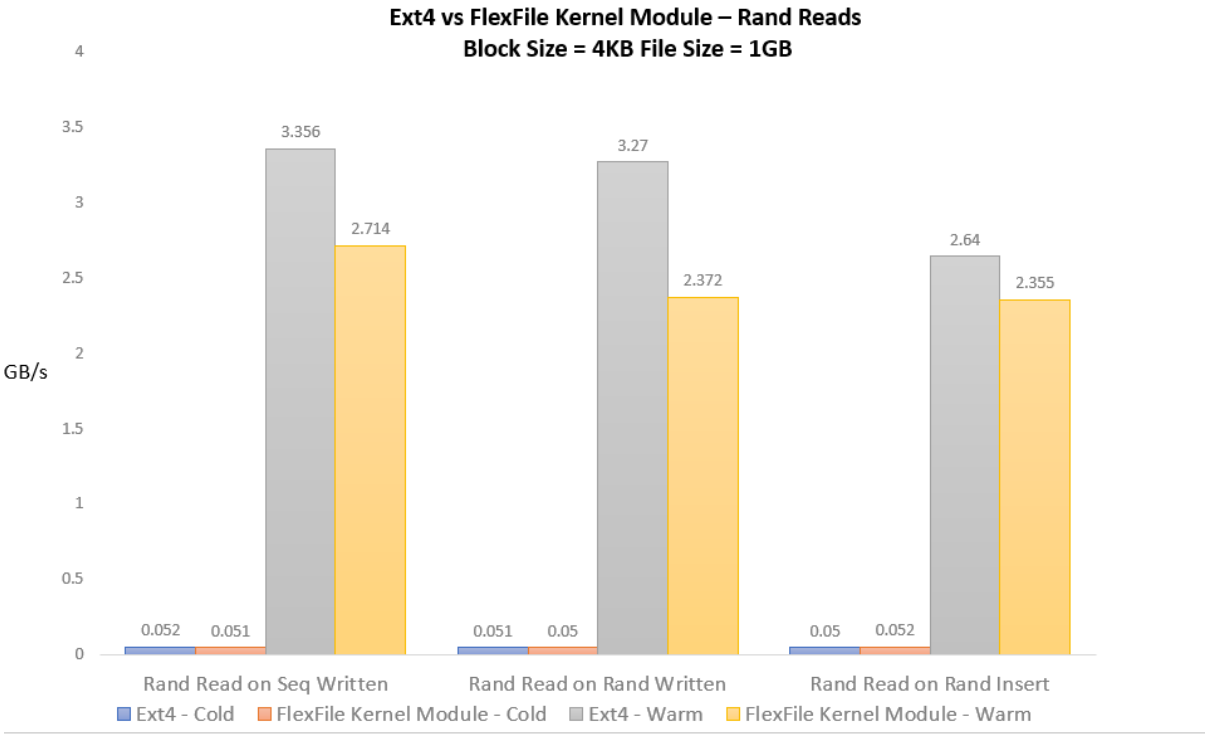

Figure 8. Performance Results for Random Reads, Ext vs FlexFile Character Device Kernel Module

Figure 7 shows comparisons between Ext4 and FlexFile character device kernel module. For sequential reads we see a slightly higher difference than the FlexFile user space programs performance. With higher differences between "easier" workloads. This is due to the kernel modules overhead being more prevalent in the data. For "slower" operations this overhead will

be hidden from the complex computation being done. Differences of 1GB/s from sequential reads on random written and random inserts are prevalent due to Ext4 not having fragmented files where FlexFile have a better chance at being highly fragmented. Also, look-ahead in the warm cache plays a role. Ext4 has a high chance of having valid look-ahead due to not being fragmented, FlexFile will have a lesser chance of having valid look ahead because of the fragmented file.

Figure 8 evaluated random read operations on the three different file types. Cold cache tests show that FlexFile kernel module as well as Ext4 performance is equivalent. This is due to the page cache not having valid look ahead for both cases. Warm cache shows differences <1GB/s but a higher difference than what is seen with FlexFile user space. This is due to the overhead being prevalent but not as sever for more complex workloads.

5.4.3 Problems and Resolutions

To summarize the findings, we see a slight drop in performance, when comparing to the user space implementation of FlexFile, but only for some workloads as outlined. This is due to the overhead of the character device kernel module overhead. The overhead is found in increased context switches, ie pwrite/pread used in the FlexFile user space vs llseek and write/read used in FlexFile kernel space. Overhead of increased instructions are found when preforming locking and flexfile struct retrieval. Lastly, the inability of io_uring will also decrease performance slightly. This overhead will be seen more drastically throughout “easier” workloads and in a sense hidden in more complex workloads.

Some resolutions can be found when moving onto the last possible solution for porting into the Linux kernel, a pseudo filesystem. With this we will be able to remove the overhead of increased context switching. After implementation, we will be able to see if this played a large role in the difference or a slight role. Other possible resolutions is to build a cache that has more valid lookahead that what can be found between the page cache and FlexFile.

5.5 Linux kernel file system porting option - Pseudo File system

A pseudo filesystem will be helpful to provide connections to not only the Linux virtual file system but also to the file system basics, ie. directory contents. The Linux kernel handles most of the work while handlers are used to handle file specific tasks that are accessible to each file system. The kernel uses several operation tables, and super block operations. The operation tables contain an assortment of handlers for each individual operation. Once the inode is opened different operation tables are configured to handle inodes and files. The super block operations are set up during mounting of the file system.

This option will be helpful after the development of the character device kernel module. After an analysis is finished, improvements made, and an extensive testing suite set up, then the transition to a pseudo filesystem can be completed. The pseudo filesystem will add more features to truly get the kernel mockup behaving like a real file system passed just the basic file system calls. The development of this pseudo file system may be able to resolve some of FlexFile kernel mockup's overhead cost by reducing context switches. The possible use of pread and pwrite will be able to provide the user with an option that is not preforming a seek then read or a seek then write.

CHAPTER 6

FUTURE DEVELOPMENT

With these performance results we can see open areas for further research. The use of a custom cache could be implemented. This cache can utilize the new type of addressing found in FlexTree. Another possible further development can be to finish implementing the pseudo filesystems. A pseudo file system can possibly resolve some of FlexFile character device kernel module's overhead. These future developments could lead to closing the gap between FlexFile kernel implementation and FlexFile user space implementation.

CITED LITERATURE

1. MARY MADDEN AND LEE RAINIE: Americans' views about data collection and security. https://www.pewresearch.org/internet/2015/05/20/americans-views-about-data-collection-and-security/, May 2015.

2. What is a database. https://www.oracle.com/database/what-is-database/, 2021.

3. Daniel Beßler Sascha Jongebloed Michael Beetz: Prolog as a querying language for mongodb. https://arxiv.org/pdf/2110.01284.pdf, October 2021.

4. Yifan Qiao, Xubin Chen, Ning Zheng, Jiangpeng Li, Yang Liu, Tong Zhang: Closing the b-tree vs. lsm-treewrite amplification gap on modern storage hardware with built-in transparent compression. https://arxiv.org/pdf/2107.13987.pdf, 2021.

5. Haoyu Huang and Shahram Ghandeharizadeh: Nova-lsm: A distributed, component-based lsm-tree key-value store*. https://arxiv.org/pdf/2104.01305.pdf, May 2021.

6. Sorted sequential files. http://www.cs.bilkent.edu.tr/~kdincer/teaching/spring1999/bu-bil212-fo/lectures/pdf-files/bil212-chp3-2.pdf.

7. Md. Rafiqul Islam, S. M. Raquib Uddin and Chinmoy Roy: Computational complexities of the external sorting algorithms with no additional disk space. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.524.5876&rep=rep1&type=pdf, December 2005.

8. Patrick O'Neil Edward Cheng Dieter Gawlick and Elizabeth O'Neil: The log-structured merge-tree (lsm-tree). https://www.cs.umb.edu/~poneil/lsmtree.pdf, 1996.

9. Patrick O'Neil, Edward Cheng, Dieter Gawlick, Elizabeth O'Neil: The log-structured merge-tree (lsm-tree). https://www.cs.umb.edu/~poneil/lsmtree.pdf, 1996.

10. Chen Chen, Wenshao Zhong, and Xingbo Wu: Efficient data management with a flexible address space. https://arxiv.org/pdf/2011.01024.pdf, August 2021.

11. Goetz Graefe and Harumi Kuno: Modern b-tree techniques. In IEEE International Conference on Data Engineering. IEEE, 2011.

12. fallocate(2) — linux manual page. https://man7.org/linux/man-pages/man2/fallocate.2.html.

13. McCarty, S.: Architecting containers part 1: Why understanding user space vs. kernel space matters. https://www.redhat.com/en/blog/architecting-containers-part-1-why-understanding-user-space-vs-kernel-space-mat 2015.

14. Horton, T.: Hci in software development. http://www.cs.virginia.edu/~horton/cs3205/cs3205-9-prototyping3-s17.pdf.

15. Unit testing software testing. https://www.geeksforgeeks.org/unit-testing-software-testing/, 2019.

16. How kernel, compiler, and c library work together. https://wiki.osdev.org/How_kernel,_compiler,_and_C_library_work_together.

17. Is fast code or small code preferred? https://www.gnu.org/software/libc/manual/html_node/FP-Function-Optimizations.html.

18. Protection and the kernel. https://courses.cs.duke.edu/cps110/spring00/slides/kernel.pdf, 2000.

19. Resolving kernel panics. http://www.thexlab.com/faqs/kernelpanics.html.

20. Virtual box. https://www.virtualbox.org/.

21. What is a kernel module? https://linux.die.net/lkmpg/x40.html.

22. Overview of the linux virtual file system. https://www.kernel.org/doc/html/latest/filesystems/vfs.html.

23. Block device drivers. https://linux-kernel-labs.github.io/refs/heads/master/labs/block_device_drivers.html.

24. Linux kernel blkdev.h. https://elixir.bootlin.com/linux/latest/source/include/linux/blkdev.h#L1855.

25. Character device drivers. https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html.

26. Linux kernel fs.h. https://elixir.bootlin.com/linux/latest/source/include/linux/fs.h#L2022.

27. Debugging kernel and modules via gdb. https://www.kernel.org/doc/html/latest/dev-tools/gdb-kernel-debugging.html.

28. Debugging the linux kernel. http://www.embeddedlinux.org.cn/EmbeddedLinuxPrimer/0136130550/ch14lev1sec3.html.

29. Message logging with printk. https://www.kernel.org/doc/html/latest/core-api/printk-basics.html.

30. Kernel/traditional compilation. https://wiki.archlinux.org/title/Kernel/Traditional_compilation.

31. Linux kernel open.c. https://elixir.bootlin.com/linux/v4.7/source/fs/open.c#L840.

32. Linux kernel namei.c. https://elixir.bootlin.com/linux/latest/source/fs/namei.c#L3865.

33. Difference between kmalloc() and vmalloc(). https://www.emblogic.com/blog/10/difference-between-kmalloc-and-vmalloc/.

34. Chapter 3. locking in the linux kernel. https://www.kernel.org/doc/htmldocs/kernel-locking/locks.html.

VITA

NAME:	Polly Planinsek
EDUCATION:	BS, Computer Science, University of Illinois at Chicago, Chicago, Illinois, 2021
	MS, Computer Science, University of Illinois at Chicago, Chicago, Illinois, 2021