ezFS: A Pedagogical Linux File System

Emma Nieh
c25en@dalton.org
The Dalton School
New York, NY, USA

Zijian Zhang
zz2795@columbia.edu
Columbia University
New York, NY, USA

Jason Nieh
nieh@cs.columbia.edu
Columbia University
New York, NY, USA

ABSTRACT

Hands-on programming experience is crucial for students to learn about operating systems, but implementing key concepts such as file systems is perceived as being too hard to do for a real operating system in an introductory course on operating systems. To overcome these barriers, we introduce ezFS, a Linux file system that supports standard file system operations to persistent disk storage, yet is simple enough for students in an introductory operating systems course to implement in a couple weeks. ezFS takes advantage of file system and block storage interfaces in Linux that simplify file system implementation, such that its implementation requires only a few hundred lines of C code. We leverage standard file system interfaces to also develop an ezFS grader that can automatically grade ezFS implementations so that it is easy to scale its use for teaching a large course. We have successfully used ezFS as a programming assignment in an introductory operating systems course for hundreds of college students. ezFS significantly enhanced students' understanding of how file systems work in real operating systems, was simpler to implement than even pseudo Linux file systems, and was less difficult to complete than other programming assignments typically assigned for the course.

CCS CONCEPTS

• **Social and professional topics** → **Computer science education**; • **Software and its engineering** → **File systems management**; **Operating systems**.

KEYWORDS

file systems, operating systems, computer science education, Linux

ACM Reference Format:
Emma Nieh, Zijian Zhang, and Jason Nieh. 2025. ezFS: A Pedagogical Linux File System. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE TS 2025), February 26-March 1, 2025, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3641554.3701884

1 INTRODUCTION

Hands-on learning through programming projects is crucial in computer science education, especially for systems-oriented courses such as operating systems (OS). OS courses often involve kernel programming projects to teach students about real-world OS design and implementation. An important topic for any OS course is file systems (FS). A useful FS programming project would teach students about how common file operations work, including how an FS stores data persistently to a storage device, creates and deletes files and directories, and runs programs from the FS. The FS should support reasonable size files so that students can use it to run real programs and store real files such as images. The FS should work with a real OS so students can see how a real FS works in practice. Although such an assignment would provide students with invaluable hands-on learning, conventional wisdom is that this is too difficult for students to do in an introductory OS course [3, 15, 17, 27].

A common approach is to use Linux for programming projects, but make suboptimal tradeoffs to mitigate the complexity of a real OS. Some OS courses do kernel programming projects for other OS topics, but forgo implementing an FS [5, 11, 24] or only do user-level FS projects [29, 30] or systems programming [23]. Students end up having a lesser understanding of how FSes work because of the lack of hands-on FS kernel programming experience. Alternatively, other OS courses do FS kernel programming, but only develop pseudo FSes that provide no persistent storage, or make minor modifications to an existing disk-based FS that do not involve core FS functionality [17, 22]. Students end up with only a limited understanding of the core persistent storage functionality of FSes. Another approach is to build an FS in the context of a pedagogical OS, which may have missing functionality for students to complete. This may avoid the complexity of real OSes, but does not expose students to many of the real-world issues that arise in practice in implementing FSes. Furthermore, since they are not used for running real systems, pedagogical OSes tend to become obsolete, especially as hardware evolves [22].

To address these educational challenges, we introduce ezFS[1], an easy-to-implement Linux FS for students to develop so they can learn how file operations really work. It is simple enough for students in an introductory OS course to implement in a short period of time. ezFS provides students hands-on experience with Linux, not just a toy pedagogical OS. It teaches students how to implement file operations involving files stored on persistent storage, as opposed to just learning about pseudo FSes or tangentially related functions unrelated to core FS functionality. ezFS teaches students the internals of how standard FS operations work, including creating and deleting files and directories and running programs. It can support files of arbitrary size so that it can be used with the same binaries and images that are stored on widely used Linux FSes.

Contrary to conventional wisdom, our insight is that modern commodity OSes such as Linux provide FS and block storage interfaces that can simplify FS implementation while providing students with hands-on experience building a disk-based FS in a real OS. ezFS leverages Linux's Virtual File System (VFS) infrastructure and its generic functions to simplify implementing FS operations. ezFS also

[1] ezFS is short for both easy FS and the Emma Zijian FS.

SIGCSE TS 2025, February 26-March 1, 2025, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-0531-1/25/02
https://doi.org/10.1145/3641554.3701884

takes advantage of Linux's functions that make it easy to access disk storage and interface with its in-memory page cache. ezFS further enables students to learn by example from the available source code of many existing Linux FSes.

We have successfully used ezFS as a programming assignment in an introductory OS course for hundreds of college students at Columbia University. We also developed an ezFS autograder that leverages standard FS interfaces to automatically grade assignments so that it is easy to scale its use for teaching a large course. We surveyed over two hundred students to gather their experiences with ezFS. ezFS was no more difficult to complete than most other OS programming assignments, including previous FS assignments that did not provide students with the opportunity to implement a disk-based FS. It is simpler to implement than almost all existing FSes in mainline Linux, including pseudo FSes. Despite its simplicity, the assignment significantly enhanced students' understanding of how FSes work in real OSes.

2 DESIGN

Any disk-based FS requires an on-disk format to represent its FS metadata and data in persistent storage, which is then translated to an in-memory representation while the FS is in use. Linux FSes make use of its VFS interface, which is a software layer within the Linux kernel that provides a unified interface to interact with FSes. Linux FSes have an on-disk representation that is composed of an array of disk blocks storing metadata and data, and must in turn support three in-memory VFS data structures listed in Table 1, the superblock, inode, and dentry. The basic operation of an FS will read the on-disk FS-specific representations into FS-specific data structures in memory, populate the respective in-memory VFS data structures, update the in-memory data structures as needed, then eventually write them back to disk for persistent storage. Linux provides a buffer abstraction which is used by block I/O interfaces to read and write persistent storage as a linear array of blocks, also listed in Table 1.

ezFS uses an FS structure to make it as simple as possible to interact with the VFS and the Linux memory subsystem. It simplifies the use of buffers with 4KB disk blocks that match the default granularity of memory pages for the most popular CPU architectures, including both x86 and Arm, enabling a one-to-one mapping of disk blocks to memory pages. It makes it easy to manage the superblock and inodes by having on-disk FS structures that mirror these in-memory objects. It has a simple representation of the entries in a directory, which makes it easy to populate the in-memory dentry.

ezFS's on-disk FS layout consists of two blocks of FS metadata and an arbitrary number of blocks of FS data. The first block is its superblock, which includes two free space bit vectors, one to track which inodes are free and the other to track which data blocks are free. An inode or data block is in use if its corresponding location in the respective bit vector is set to one. The second block consists of an array of inodes. Since all inodes are stored in a single 4KB disk block, this limits the number of inodes in the FS to roughly 40 if we assume each inode is roughly 100B, which is sufficient for a small FS. The rest of the FS consists of data blocks. Each data block can contain either regular file data or the contents of a directory. In the case of the latter, the block is simply an array of directory entries, each of which is 128B in size.

Abstraction	Description
superblock	Object representing metadata of the overall FS
inode	Object representing metadata of an individual file or directory
dentry	Object mapping filename to its inode, cached by dentry cache
buffer	Object representing disk block in memory, data in page cache

Table 1: Linux VFS and block I/O abstractions.

Each inode contains an index of the data blocks that are part of the file. For simplicity, ezFS has a small two-entry index in each inode, which can be used to flexibly support different schemes to track multiple data blocks per file. For example, ezFS can support directories of up to two blocks in size if each index entry references a single data block containing directory entries. Directories can be simplified to just use one index entry if we restrict directories to a single data block. This is still sufficient for a directory to have up to 32 files if each ezFS directory entry is 128B and 4KB data blocks are used. To support files with more than two data blocks, ezFS can use an indexed allocation scheme in which the two entries refer to a direct and indirect block. The former references a single data block for the file while the latter references a block that contains references to other data blocks. File size is limited by the number of block addresses that can be stored in an indirect block; a file can be up to 2MB in size with 4KB data blocks and 64-bit block addresses. To support even larger files, ezFS can use a contiguous allocation scheme, which allows an arbitrary number of data blocks per file. Since the data blocks for a file are assumed to be contiguous, the two index entries only need to store the block number of the first data block and the number of data blocks for the file.

To help students understand how ezFS operates, we provide a header file with the ezFS-specific data structures for FS metadata, namely the superblock, inode, and directory entry, and structure the ezFS programming project in multiple parts. The first part teaches students how to format a disk to use ezFS and provides them with the source code for a simple ezFS disk formatting utility program that lays out the ezFS FS structures on a disk. Subsequent parts teach students how to implement ezFS, with each part requiring the implementation of some FS operation. The order of parts build on one another such that students can leverage the functionality of previous parts to check the correctness of subsequent parts. ezFS is also implemented as a kernel module to make the assignment debug-friendly. This enables students to quickly update the FS implementation by unloading then loading the ezFS module again without needing to recompile or reboot the kernel.

2.1 Manually creating an FS

The first part of the assignment gives students an understanding of the on-disk ezFS layout and how it works. We provide students with an ezFS formatting utility program and its source code which they can use to format a disk to use ezFS. Students execute `dd` and `losetup` commands to make a file accessible as a block device by creating a disk image file and assigning it to a loop device. This makes it easy to use a file on the default FS as a storage device on which to install ezFS. Students format the loop device using the formatting utility, which sets up the superblock, inodes, and a hello world text file on the FS. We provide students a compiled ezFS kernel module, sans its source code, so they can then mount and use the formatted FS to see its resulting behavior, which is to show its root directory with a single hello world text file. Figure 1 shows some of the code

```
int main(int argc, char *argv[])
{
 ssize_t ret, len;
 struct ezfs_inode inode;
 struct ezfs_dir_entry de;
 char *hello_contents = "Hello world!\n";
 int fd = open(argv[1], O_RDWR);
...
 inode.nlink = 1;
 inode.mode = S_IFREG | 0666;
 inode.data_block_number = EZFS_ROOT_DATABLOCK_NUM + 1;
 inode.file_size = strlen(hello_contents);
 inode.nblocks = 1;
 ret = write(fd, &inode, sizeof(inode));
...
 strncpy(de.filename, "hello.txt", sizeof(de.filename));
 de.active = 1;
 de.inode_no = EZFS_ROOT_INODE_NUM + 1;
 ret = write(fd, &de, sizeof(de));
...
 ret = write(fd, hello_contents, strlen(hello_contents));
...
}
```

Figure 1: Adding hello world text file in formatting utility.

provided to the students that adds the hello world text file to the FS, showing them how to fill in the metadata for an inode and directory entry and write it to disk along with the file contents.

Students can review the formatting utility source code to see how it generates an FS with the resulting behavior, including the exact values that are written to the fields of the superblock, the fields of the inodes for the root directory and the hello world text file, the directory entry in the root directory for the hello world text file, and the file contents of the hello world text file. We ask the students to extend the formatting utility to add other files to the FS. Students use the provided ezFS data structures to add small files, big files, and subdirectories. They then check that all required ezFS information was written properly by mounting the FS and using the reference ezFS kernel module to see if the mounted FS has the correct hierarchy of directories, file metadata, and file contents. Students gain an understanding of how to manually create files and directories in the FS via the formatting utility before attempting to implement the functionality in the FS.

2.2 Initializing and mounting the FS

Once students understand how to manually add files to their FS via the formatting utility, they create their own module implementing the FS operations so it can use the FS created by the formatting utility. The first step is to initialize and mount the FS, which involves writing the functions that are called to load and unload the kernel module, to register and unregister the FS, and to mount and unmount the FS. We provide suggestions to the students to reference existing Linux FS implementations, including ramFS [18] and BFS [12].

Initialization is trivial to implement, but `mount` and `umount` require students to gain some understanding of the basic VFS generic objects, including the data structures for the in-memory representation of superblocks and inodes and the current Linux VFS interface functions including `init_fs_context` and `get_tree`. Students also need to learn the basic block storage interface functions `sb_bread` and `brelse`, which provide simple ways to read disk blocks by block number into the page cache and reference them as buffers, then release them when they are no longer needed. These functions make it straightforward to read data from disk into the cache and flush data in the cache to disk. They are essential for accessing FS metadata from disk. Students then use `sb_bread` when the FS is mounted to read the on-disk superblock and root directory inode metadata , and

```
const struct file_operations ezfs_file_ops = {
 .read_iter = generic_file_read_iter, ...
};

const struct address_space_operations ezfs_aops = {
 .read_folio = ezfs_read_folio, ...
};

int ezfs_read_folio(struct file *f, struct folio *folio)
{
 return block_read_full_folio(folio, ezfs_get_block);
}

int ezfs_get_block(struct inode *inode, sector_t block,
    struct buffer_head *bh_result, int create)
{
 struct ezfs_inode *ezfs_inode = inode->i_private;
 int ez_n_blk = inode->i_blocks / 8;
 int phys = ezfs_inode->data_block_number + block;
 if (ez_n_blk && block < ez_n_blk) {
  map_bh(bh_result, inode->i_sb, phys);
  return 0;
 }
...
}
```

Figure 2: Implementation to read regular files.

`brelse` to release resources when the FS is unmounted. This part is complete once the FS can be mounted and unmounted successfully without errors.

2.3 Listing the contents of directories

The next step is to list the contents of directories so students can see the files in the FS. All of the root directory's in-memory data structures have already been initialized so all that is required to list its contents is to implement the function `iterate_shared`. This involves reading the inode of the directory to identify the block number of the data block containing the directory entries, using `sb_bread` to get the data block itself, reading each directory entry from the block and invoking `dir_emit` to output its filename. The required functionality is complete once `ls -l` correctly lists the contents of the root directory.

For subdirectories which have not yet been accessed, students need to further implement a lookup function to identify and initialize the inode for a directory when it is accessed. When a directory is accessed, the generic VFS lookup function is called, which in turn calls the ezFS-specific lookup function that students need to implement. This lookup function takes as its arguments the inode of the parent directory and the semi-initialized dentry of the directory. Students need to implement the code to find the corresponding directory entry in the parent directory's data block to identify the inode number of interest, retrieve the inode from disk using `sb_bread`, allocate and initialize an in-memory VFS inode using the metadata from the on-disk inode, and finally link the VFS inode to the dentry. At this point, the dentry is complete and the workflow to list the contents of the directory flows back to `iterate_shared` again. This part is complete once `ls -l` can correctly list the contents of any subdirectories.

2.4 Reading regular files

The next step is to support reading regular files. The simplest and recommended approach is to leverage existing generic functions to implement the `read_iter` function, which is called on the `read` system call, specifically the `generic_file_read_iter` function as shown in Figure 2. This function already supports complex read ahead logic so that file blocks can be cached in memory by the time they are needed to avoid blocking on I/O.

Generic FS functions are unaware of FS-specific functionality to determine what data blocks are associated with a file, so they invoke address space functions which implement FS-specific functionality to map a logical block of a file to a page of memory caching the contents of an actual disk block. `generic_file_read_iter` will invoke `read_folio` to read a file. Students need to implement an ezFS-specific `read_folio`, which simply involves calling the Linux function `block_read_full_folio`. This function requires implementing an ezFS-specific function `ezfs_get_block` with the core functionality needed. This function needs to create a mapping from the logical block number of the file being requested to the actual disk block number based on the metadata in the respective inode. This mapping is straightfoward to compute based on the information in the inode. For example, with a contiguous allocation scheme, the mapping is just the logical block number plus the starting disk block number in the inode. ezFS's use of disk blocks with the same size as memory pages also simplifies this mapping. After the mapping is returned by `read_folio`, the VFS will then read the actual disk block and cache it in memory in the page cache. Figure 2 shows that reading existing files only requires students to implement a few lines of code overall, and students can easily learn much of this basic structure from looking at the source code for other FSes that come with Linux, such as BFS. Once this part is complete, students will have a fully readable FS. Students can use `cat` to check if the contents of files are the same as what they wrote into the disk using the formatting utility.

2.5 Writing existing files

The next step is to support writing existing files. The recommended approach is to use the existing function `generic_file_write_iter` to implement the `write_iter` function which is called on the `write` system call. `generic_file_write_iter` takes care of most of the logic, including determining the logical block number of interest, copying the data to be written from userspace, and updating metadata such as the size of the file.

Writing files requires implementing `write_begin`, an FS-specific function which needs to provide the mapping from a logical file block number to an actual disk block number, similar to `read_folio`. The VFS will then cache the contents of the disk block if it is not already in the page cache and perform the actual write. Depending on the allocation scheme used, a challenge that students need to address when writing as opposed to reading a file is when writing increases the file size and requires a new data block. For example, assuming a contiguous allocation scheme, if the data block following the last existing data block of the file is already in use, the implementation has to move the existing blocks to another position that can accommodate the new file size to maintain contiguous allocation of data blocks. This part is completed when students can write arbitrary content of various sizes to existing files and truncate existing files, resulting in an FS that is both readable and writable.

2.6 Create and delete files and directories

The next step is to support creating and deleting files and directories, which involves implementing `create` and `unlink` file operations for files and `mkdir` and `rmdir` for directories. Creating a file involves getting the data block number of the parent directory from its inode, iterating through its array of directory entries in the data block to find an unused entry, iterating through the free space vector for inodes to find an unused inode, creating and initializing a new in-memory VFS inode with the respective inode number and connecting the inode to the dentry, getting the filename from the dentry, filling in the unused directory entry with the filename and inode number, and updating the metadata in the superblock. The logic of `mkdir` is similar to creating a file. The only difference is that, although the directory is empty, it still needs one data block. Therefore, besides finding an available inode, it also needs to find an empty data block.

Deleting a file involves getting the data block number of the parent directory from its inode, iterating through its array of directory entries in the data block to find the file to be deleted, invalidating the directory entry, and decreasing the link count for the inode by one. The logic of rmdir is similar, but students must implement checks to make sure that the directory is empty and also update the link count of the parent directory. When an inode's link count drops to zero, the VFS will call `evict_inode` to reclaim the inode.

Students are expected to do lots of create and delete operations to test robustness. ezFS's design of one block for all inodes and use of potentially one-block directories requires students to properly check FS limits, as the number of files in any directory and in ezFS overall are both limited to maximums that are easily testable.

2.7 Compile and run executable files

The last part of the assignment involves supporting compiling and running executable files. Students can look at the source code for the `exec` system call to see that this requires support for the `mmap` operation, which is trivial as it just involves calling the generic function `generic_file_mmap`. All FS-specific functions that the generic function calls are already implemented in earlier parts of the assignment. The FS can now be used to compile and execute programs.

2.8 Autograder

To simplify grading, we designed an ezFS autograder, leveraging the ezFS kernel module design and the well-defined functionality of different parts of the assignment that are straightforward to check. Students only need to ensure the name attribute of their FS is `myezfs` for identification purposes and use a provided template Makefile to generate the kernel module so that its name is well known. The autograder takes a few minutes to grade each submission. It loads the module, sets up a loop device with a preformatted disk image, then runs various commands and checks the result. For example, the autograder tests the correctness of creating a regular file as follows:

(1) Run `touch [newfile]` then `ls` to check if `[newfile]` is listed.
(2) Run `touch` to create multiple files in a directory, check if it fails when the number of files exceeds the limit per directory.
(3) Run `touch` to create multiple files in multiple directories, check if it fails when the number of inodes exceeds the global limit.

3 EXPERIENCES

We have used ezFS for three years as the FS programming project for a one semester introductory OS course at Columbia University. Course enrollments range from seventy to over a hundred students in a given semester, a mix of advanced undergraduate and graduate students. Six programming projects were assigned for the course, with the first being an individual non-kernel programming assignment

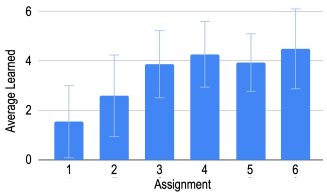

Figure 3: Student learning.

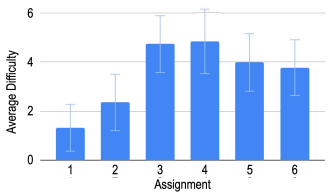

Figure 4: Assignment difficulty.

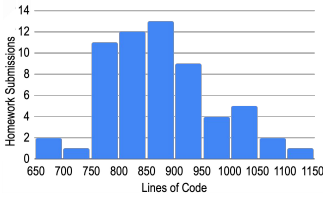

Figure 5: Student submissions.

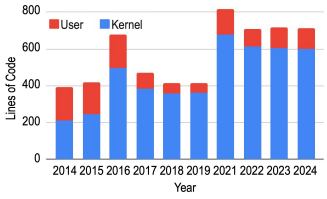

Figure 6: FS project solutions.

and the remaining five being kernel programming assignments. Kernel programming assignments were done in teams of mostly three students and sometimes two students. Student teams had two weeks to complete each assignment, including the FS assignment. Students ran Linux virtual machines [4] using VMware [21] on their own computers to complete the assignments, then uploaded their work to GitHub for grading. We used ezFS across courses that used different Linux kernel versions, upgrading to a newer kernel version each semester. Despite major changes to the Linux VFS infrastructure in between updates, including the introduction of folios in the Linux 5.16 kernel, only minor updates were required to ezFS to work across kernel versions, a testament to its clean and simple design.

During three separate semester offerings of the course taught by different faculty members, we asked students to complete a brief survey at the end of the course to help answer various research questions (RQs). Each semester survey asked students how difficult each programming assignment was including the ezFS assignment and how many hours they spent on the ezFS assignment. The latter two semester surveys also asked students how much they learned from each programming assignment including the ezFS assignment and a Likert scale question on whether "the ezFS assignment significantly improved my understanding of FSs." Learning was rated on a scale of 1 to 6, with larger numbers indicating students learned more. Similarly, difficulty was rated on a scale of 1 to 6, with larger numbers being more difficult. 216 out of 265 students completed the surveys in total. The Likert scale was 1 to 5, with 5 indicating strong agreement.

RQ1: How much do students learn about FSes from ezFS? Students strongly agreed that the ezFS assignment significantly improved their understanding of FSes, with an average Likert scale rating of 4.6 out of 5. This was also reflected in student comments in the survey: "best hw among all assignments," "one of the most fun programming projects I've completed," "very enjoyable," "appropriate level of challenge," "super cool to learn how to implement our own FS," "enjoyed this aspect of working with an FS and learning in depth how they work," "learned a lot about FSes through this assignment," and "assignment has truly given me a comprehensive understanding of FS operations and their implementations." Students think the ezFS assignment is a bit like stages of a game and are excited as their baby FS "grows up" after each part. There is a sense of accomplishment when they pass the final part, making the FS able to run executables.

We also used the learning rating from the surveys to compare ezFS against other programming assignments. Figure 3 shows the average learning rating of all the programming assignments for the OS course, including ezFS (assignment 6), with variance indicated by error bars. Figure 3 shows that students rated ezFS the highest among all the programming assignments in terms of how much they learned from the assignment. Except for the ezFS assignment, the other assignments closely resemble those given in previous semesters. The first two assignments mostly get students comfortable with the basics of systems and kernel programming by writing a shell and baremetal hello world program, and a system call to dump process tree information. The latter four assignments each required students to implement some kernel functionality to understand a key component of the OS kernel, including CPU scheduling, memory management, and the FS. Given the more basic nature of the first two assignments, it is not surprising that students learned more from the latter four assignments. Students learned the most from the two assignments that leveraged Linux infrastructure to build an entire OS subsystem themselves, namely an FS (ezFS) and a scheduler (assignment 4).

RQ2: Can ezFS be implemented with reasonable effort? Figure 4 shows the average difficulty rating for each programming assignment, including the ezFS assignment (assignment 6), with variance indicated by error bars. Except for the first two more basic assignments, ezFS was the easiest of the programming assignments. Together, Figures 3 and 4 show that ezFS provides significantly more learning for students than all other assignments while being easier to implement than most other assignments. ezFS is easier than most other assignments for several reasons, including that it is structured in stages, leverages existing Linux VFS functions, has simple Linux FSes to learn from, and can be more self-contained as a kernel module that does not keep the system from booting even if it is not working. The results indicate that ezFS achieves its goal of providing hands-on kernel development experience for learning about FSes in a manner that is no more difficult, and in many cases easier than, what is required to learn about other OS kernel components. With students spending an average of 15 hours per week on the two-week ezFS assignment plus attending lectures for 2.5 hours per week, the total hours required was in line with what was expected for a 3 unit course.

Figure 5 quantifies the difficulty of ezFS in terms of how many lines of code (LoC) students implemented for the programming assignment as measured using `cloc` [9] on their submitted code. Code that was provided to students as part of each assignment was excluded. Although a LoC metric may not directly correlate with difficulty, it provides a quantitative measure of how much code the students had to write for the assignment. We only included student submissions that received a grade above the median grade for the assignment, to avoid skewing the LoC statistics with submissions of lower quality FS functionality. The student submissions on average contained less than 900 LoC, with roughly 140 LoC for the formatting utility and 750 LoC for ezFS kernel module itself. Substantial portions of the code are easy to write. For example, the formatting utility involves relatively simple user-level programming, and approximately 150 LoC for the kernel module are for just registering and mounting the

FS, much of which can be copied from existing Linux FSes. Considering the fact that the assignment was completed in teams of three students, on average each student had to write less than 300 LoC, which is quite manageable for a two-week assignment. For comparison, the ezFS assignment solutions only involve implementing about 600 LoC for ezFS itself and another 130 LoC for the formatting utility. The solutions encapsulate common code in helper functions called by multiple functions for creating and deleting files and directories, whereas most student submissions have more redundant code, resulting from copying the same code to multiple functions.

RQ3: How difficult is implementing ezFS compared to other FSes? Figure 6 quantifies the difficulty of ezFS versus FS assignments assigned in seven previous semesters of the OS course in terms of LoC in the solutions for each assignment. ezFS was assigned with contiguous allocation in 2022, and indexed allocation in 2023 and 2024. Its code complexity is within the range of previous FS project assignments, yet none of the previous projects enabled students to implement a disk-based FS. Previous projects were either implementing pseudo FSes, or adding inode metadata to the default Linux ext4 FS such as GPS location data, the former requiring more complex functionality. For example, the most complex assignment in terms of LoC was from 2021, which involved implementing a pseudo FS that exports kernel information about physical page usage for currently running processes. Unlike ezFS, these previous assignments did not involve any disk-based functionality and omitted key FS operations such as creating files and directories. In contrast, ezFS teaches students the fundamentals of how an FS really works to manage and access data on persistent storage, the main purpose of most FSes.

We also measured the code complexity of ezFS versus existing FSes in Linux. Of the 79 FSes in the mainline Linux kernel version 6.1, only six FSes have less code than ezFS, all of which are either pseudo FSes (Ramfs) or lack generic FS functionality (Devpts, Tracefs, Exportfs, Openpromfs, and QNX4fs). Linux FSes of similar size to ezFS lack all of ezFS's functionality for creating, writing, and executing files or directories. For example, QNX4fs [14] only supports reading QNX4 FSes; it cannot create or write files or directories. Other Linux disk-based FSes range from twice as much code, in the case of BFS, to orders of magnitude more code than ezFS for FSes such as ext4, the default Linux FS. Other Linux pseudo FSes are also substantially larger than ezFS. ezFS provides disk-based FS functionality with minimal complexity compared to other Linux FSes.

4 RELATED WORK

Using kernel-level programming projects to teach OS courses has become common place at universities [11]. Linux is widely used in this context given its open-source nature, available development tools, learning opportunities in a real production OS, ability to leverage real code examples from real developers for students to learn from by example, and ease of maintenance [1, 8, 16, 17, 21]. Many previous Linux-based FS projects do not enable students to implement a disk-based FS due to complexity [17, 22], resulting in students missing out on hands-on learning about key concepts and abstractions for FSes [20]. ezFS solves this problem and can be easily adopted, especially in existing Linux-based kernel programming courses. Its approach of having students first manually create an FS by modifying a formatting utility program was inspired by PantryFS [10], an FS assignment with single block files which was previously used at Columbia for a different section of the OS course.

Linux provides example FSes, although they cannot be directly used for hands-on implementation experience. We encourage students to reference Ramfs and BFS. Ramfs is "most useful not as a real filesystem, but as an example of how virtual filesystems can be written" [18]. It is similar to ezFS in code size, but operates entirely in memory and provides no persistent storage. BFS [12] is a disk-based FS for a boot loader that uses contiguous allocation, but does not support directories, shrinking files, or sparse files, uses outdated VFS interfaces, and yet requires twice as much code as ezFS.

Pedagogical OSes have also been developed [2, 6, 7, 13, 19, 25]. Programming projects in this context often involve completing missing functionality in the OS. These pedagogical OSes typically already have an FS, but lack the well-developed FS and block storage interfaces of real production OSes [26] for developing other FSes. As a result, projects typically focus on enhance existing FS functionality, such as increasing the maximum file size or adding symbolic links [7]. In contrast, ezFS affords students the opportunity to build an entire FS. Keeping pedagogical OSes from becoming outdated as hardware and production OSes continue to evolve remains difficult; for example, MINIX was last updated more than five years ago [13]. The MINIX FS has been ported to Linux, but has more than three times as much code as ezFS. ezFS avoids MINIX's complexity and maintenance challenges to make it easier to learn about FSes.

Filesystem in Userspace (FUSE) [28] makes it possible to create FSes without writing kernel code. It is primarily used for implementing virtual FSes as opposed to disk-based FSes. FiST [31, 32] uses FS stacking to simplify kernel programming, but is only designed for implementing virtual FSes. While these approaches can simplify implementing FSes, they are limited in providing hands-on kernel-level programming experience and teaching students about FS-related kernel internals that are a crucial part of learning about OSes.

5 CONCLUSIONS

ezFS provides hands-on learning for students to learn about FSes in a real OS, yet, at about 600 LoC, is simple enough for students to implement in an introductory OS course. Its simple design stems in part from its two-block FS metadata representation, yet supports flexibly-sized files to store realistic file content. Students gain experience and understanding implementing common file operations involving persistent storage, and can do so with modest effort by leveraging Linux FS and block storage interfaces and generic VFS functions. The ezFS assignment is structured to guide students in building a Linux FS step by step, enabling files to be readable, writable, then executable. We have taught hundreds of students using ezFS in an introductory OS course. Students found the assignment helpful to learn about FSes in the context of a real OS with reasonable effort, and learned more with less difficulty than other other existing kernel programming assignments for the course. It also taught students more about how FSes really work than FS assignments used in prior versions of the course, without introducing greater code complexity.

6 ACKNOWLEDGMENTS

This work was supported in part by NSF grants CNS-2052947, CCF-2124080, and CNS-2247370.

REFERENCES

[1] Jeremy Andrus and Jason Nieh. 2012. Teaching Operating Systems Using Android. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE 2012)*. Raleigh, NC, 613–618.

[2] Benjamin Atkin and Emin Gün Sirer. 2002. PortOS: An Educational Operating System for the Post-PC Environment. In *Proceedings of the 33rd ACM Technical Symposium on Computer Science Education (SIGCSE 2002)*. New York, NY, 116–120.

[3] Angela Demke Brown. 2022. Personal Communication (Teaching OS at University of Toronto).

[4] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. 2017. *Hardware and Software Support for Virtualization.* Morgan and Claypool Publishers.

[5] Mark Claypool, David Finkel, and Craig Wills. 2001. An Open Source Laboratory for Operating Systems Projects. In *Proceedings of the 6th Conference on Innovation and Technology in Computer Science Education (ITiCSE 2001)*. Canterbury, UK, 145–148.

[6] Douglas Comer. 2015. *Operating System Design: The XINU Approach.* Chapman and Hall/CRC.

[7] Russ Cox, Frans Kaashoek, and Robert Morris. 2020. xv6: a simple, Unix-like teaching operating system. https://pdos.csail.mit.edu/6.S081/2020/xv6/book-riscv-rev1.pdf.

[8] Christoffer Dall and Jason Nieh. 2014. Teaching Operating Systems Using Code Review. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE 2014)*. Atlanta, GA, 549–554.

[9] Albert Danial. 2022. cloc. https://github.com/AlDanial/cloc.

[10] Mitchell Gouzenko, Kevin Chen, Xijiao Li, Hans Montero, and Tal Zussman. 2023. PantryFS. https://cs4118.github.io/pantryfs/.

[11] Rob K Hess and Paul Paulson. 2010. Linux Kernel Projects for an Undergraduate Operating Systems Course. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE 2010)*. Milwaukee, WI, 485–489.

[12] Martin Hinner. 1999. Boot File System. http://martin.hinner.info/fs/bfs/.

[13] Thom Holwerda. 2023. MINIX is dead. https://www.osnews.com/story/136174/minix-is-dead/. *OSnews* (June 2023).

[14] Black Berry Inc. 2023. QNX 4 filesystem. https://www.qnx.com/developers/docs/6.6.0.update/index.html#com.qnx.doc.neutrino.sys_arch/topic/fsys_QNXFSYS.html.

[15] Orran Krieger. 2022. Personal Communication (Teaching OS at Boston University).

[16] Oren Laadan, Jason Nieh, and Nicolas Viennot. 2010. Teaching Operating Systems Using Virtual Appliances and Distributed Version Control. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE 2010)*. Milwaukee, WI, 480–484.

[17] Oren Laadan, Jason Nieh, and Nicolas Viennot. 2011. A Structured Approach to Linux Kernel Projects for Teaching Operating Systems. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE 2011)*. Dallas, TX, 287–292.

[18] Rob Landley. 2023. Linux ramfs Documentation. https://www.kernel.org/doc/Documentation/filesystems/ramfs-rootfs-initramfs.txt.

[19] Haifeng Liu, Xianglan Chen, and Yuchang Gong. 2007. BabyOS: A Fresh Start. In *Proceedings of the 38th ACM Technical Symposium on Computer Science Education (SIGCSE 2007)*. New York, NY, 566–570.

[20] Robert Love. 2010. *Linux Kernel Development.* Addison-Wesley Professional.

[21] Jason Nieh and Ozgur Can Leonard. 2000. Examining VMware. *Dr. Dobb's Journal* 315 (Aug. 2000), 70–76.

[22] Jason Nieh and Chris Vaill. 2005. Experiences Teaching Operating Systems Using Virtual Platforms and Linux. In *Proceedings of the 36th ACM Technical Symposium on Computer Science Education (SIGCSE 2005)*. St. Louis, MO, 520–524.

[23] Gary Nutt. 2001. *Kernel Projects for Linux.* Addision Wesley Longman.

[24] University of Tennessee at Chattanooga. 2018. CPSC 2800 Introduction to Operating Systems.

[25] Ben Pfaff, Anthony Romano, and G. Back. 2009. The Pintos Instructional Operating System Kernel. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE 2009)*. New York, NY, 453–457.

[26] Vijayan Prabhakaran. 2000. Linux File Systems: An Overview. *Linux Gazette* 48 (2000).

[27] Ryan Stutsman. 2022. Personal Communication (Teaching OS at University of Utah).

[28] Miklos Szeredi. 2023. libfuse. https://github.com/libfuse/libfuse.

[29] James Wolfer. 2014. Linux Experience in the General Operating Systems Class. In *XIII International Conference on Engineering and Technology Education.*

[30] Wei Xu, Xiaoyang Wang, Haoyu Mao, and Yongkun Li. 2022. The Design and Practice of Linux Kernel Based Experiments for Operating System Course. In *Proceedings of the 4th International Conference on Advanced Information Science and System (AISS 2022).*

[31] Erez Zadok, Johan M. Andersen, Ion Badulescu, and Jason Nieh. 2001. Fast Indexing: Support for Size-Changing Algorithms in Stackable File Systems. In *Proceedings of the 2001 USENIX Annual Technical Conference.* Boston, MA, 289–304.

[32] Erez Zadok and Jason Nieh. 2000. FiST: A Language for Stackable File Systems. In *Proceedings of the 2000 USENIX Annual Technical Conference.* San Diego, CA, 55–70.