SFS: A Simple File System for Teaching Parallelism in Computer Systems

Brian P. Railing
Computer Science Dept.
Carnegie Mellon University
Pittsburgh, USA
bpr@cs.cmu.edu

Lukas Kebuladze
Computer Science Dept.
Carnegie Mellon University
Pittsburgh, USA
lkebulad@andrew.cmu.edu

Nathan Deyak
Computer Science Dept.
Carnegie Mellon University
Pittsburgh, USA
ndeyak@andrew.cmu.edu

Zachary Weinberg
Million Concepts LLC
(formerly Carnegie Mellon University)
Pittsburgh, USA
zack@owlfolio.org

***Abstract*—With the growing importance of parallelism to all aspects of computing, we believe it is important to cover the subject in some depth throughout the curriculum, instead of reserving it for advanced courses dedicated to the topic. This work presents an assignment focused on parallelism, appropriate for students with one or two years of college-level programming experience. Students are asked to extend a single-threaded implementation of a simple file system (SFS) to support fine-grained concurrent access to files. We use scripted unit testing, offline consistency checks, and static analysis, to verify both the correctness and the achieved parallelism of students' work. This assignment is currently used in a second-year "introduction to computer systems" course. Most students are completing the assignment and are thereby gaining greater understanding of parallel programming within the context of computer systems.**

***Index Terms*—undergraduate, parallelism, file systems, autograding**

I. Introduction

In Carnegie Mellon University's "Introduction to Computer Systems" course [1], second-year CS students learn about the basic components of a computer system, and how these components support program execution. This course culminates with an introduction to thread-level parallelism. However, prior to our development of SFS, the only programming assignment that involved multithreading was "proxy lab," in which students would develop a caching proxy for HTTP clients.

Proxy lab's primary educational objective is to introduce network programming. Threads are the recommended way to handle concurrent clients, and the cache was a shared data structure, but it was possible to get full marks on the assignment with an unsophisticated approach to thread safety, such as one big lock around the entire cache. Students often would conclude (incorrectly) that parallelism is only useful for network servers, or that coarse locking is always sufficient. The exercise left them poorly prepared for the real concurrency-related challenges that they would encounter later in the curriculum, especially in upper-level algorithms and systems courses [2].

Our new simple file system (SFS) exercise, by contrast, is intended to focus entirely on parallelism. Students receive a complete, but not concurrent, implementation of a simple FAT-like filesystem and are asked to make it permit concurrent access. There are several ways to do this, but "one big lock" is not good enough for full marks. Students must reason about concurrency across three dimensions: data types (e.g. concurrent access to a file and the directory can be safe), data instances (e.g. concurrently writing to two different files can be safe), and code segments (e.g. calls to the "allocate a disk block" function must be serialized).

The main contributions of this work are:

- An assignment suitable for second-year students, involving complex reasoning about concurrency
- Reuse of existing filesystem tests for automatic grading
- Instrumentation and static analysis for repeatable evaluation of the parallelism achieved by student code

The remainder of this paper is organized as follows: Section II covers related work. Section III describes the course context of the SFS assignment. Section IV describes the SFS file system itself, and Section V how it is used in our assignment. Section VI describes how we evaluate student work on SFS, Section VII analyzes the impact of SFS on student learning, Section VIII discusses possible extensions of the assignment, and finally Section IX concludes this work.

II. Related Work

Previous work on the use of file systems in an educational context is limited. In the past, educators focused on tools for learning how file systems work (e.g. [3]–[5]). More recently, two papers discuss file systems designed for students to implement or extend. SPiFS [6] and ezFS [7] both aim to simplify a file system to its essentials, producing something that students can be asked to implement as a short (one- or two-week) programming assignment. These assignments also teach students about file systems, rather than using a file system

as a vehicle for teaching something else. SFS has roughly the same feature set as SPiFS, but has somewhat higher internal complexity, tailored to our goal of teaching parallelism.

There is more prior work on teaching parallelism, and we point to two recent surveys reviews on the topic [8], [9]. Our approach to the SFS assignment was guided by earlier experiments with presenting parallelism close to the beginning of a CS undergraduate curriculum [10], [11]. However, we have not seen any discussion of teaching parallelism specifically in the context of a computer systems overview course. We can devote only a short time to the topic (see Section III), so we cannot ask students to carry out the same exercise several times using different high-level frameworks for parallelism, as was done by Czarnul, Matuszek and Krzywaniak [12]. Adams, Brown, and Shoop [13] and Brown et al. [14] recommend an initial focus on high-level design patterns for parallelism. Our course's "bottom-up" presentation, beginning with fundamental building blocks (threads, mutexes, semaphores, etc), is complementary to this approach.

As parallelism is rendered ineffective by poor design choices, and as it can introduce entire new classes of bugs, teaching the *evaluation* of parallel code is just as important as teaching the techniques for writing it in the first place. Other educators have presented exercises specifically in modeling parallel execution [15] and in evaluating parallel code for performance [16]. The SFS exercise does not emphasize this aspect of parallel programming, but we evaluate student work using a task-graph model (see Section VI) which can find both serial bottlenecks and race bugs in their code.

III. Course Background

SFS was developed for use in CMU's version of "Introduction to Computer Systems" [1], typically taken by undergraduates in their third or fourth semester. It is a requirement for many students in the School of Computer Science and a prerequisite for many upper-level CS courses. Students are expected to have some familiarity with the C programming language.

Over the 14-week semester, students learn the basics of how the CPU, compiler, operating system, etc. work together to support program execution. Concurrency appears relatively late in the course, alongside Unix processes, asynchronous signals, and networking. Three of the final lectures focus specifically on parallel execution, covering its value as a tool for performance and responsiveness, the new problems it introduces (deadlocks, livelocks, races), how to identify shared data and critical sections, and tools for synchronization (mutexes, reader-writer locks, etc). The SFS exercise provides students with practical experience in the topics covered during this part of the course, with emphasis on how to parallelize access to complex data structures without introducing serial bottlenecks. Because concurrency appears late in the course, the SFS exercise has to be kept short: students only have one week to complete it. To make room for it, we shortened the "proxy lab" exercise described in the Introduction, by removing the cache component. That exercise still demonstrates the use of threading to serve concurrent clients, but no longer involves concurrent updates of a data structure.

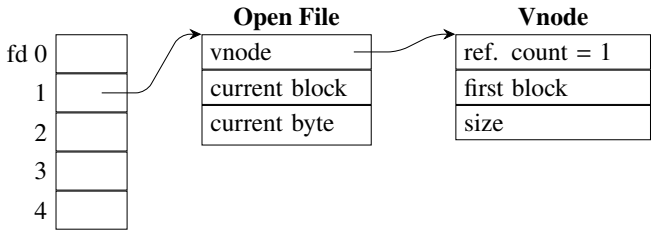

Fig. 1. SFS's data structure for open files

```
typedef struct sfs_mem_filedesc_t
{
    sfs_mem_file_t *fileEntry;
    block_id startBlock;
    block_id currBlock;
    size_t currPos;
} sfs_mem_filedesc_t;
```

Fig. 2. SFS's open-file `struct`

IV. SFS

The Simple File System (SFS)[1] is a FAT-type file system which supports a minimal set of file operations: `open`, `close`, `read`, `write`, `seek`, `rename`, and `unlink`. The semantics of these functions are taken from POSIX, simplified to avoid distracting students from the main educational goals of the assignment. For example, `open` always opens a file for both reading and writing, creating it if it does not exist. There is no access control and no subdirectories (but see Section VIII-A).

SFS is implemented as a user-mode library, running within a single user-mode process, which may or may not be multithreaded. This architecture allows students to debug their SFS code using the same techniques that worked for earlier assignments. However, it also means that test programs must be explicitly written to use the SFS library. We simplified this task by providing a custom, multithreaded Lua [17] interpreter that exposes the SFS API (see Section V-C).

SFS uses Unix-like file descriptors (small positive integers) to identify open files. The in-memory data structure for open files is illustrated in Figure 1. On the left is an array of file descriptors, each of which may (or may not) point to an open-file structure. An open-file structure is in the middle; it primarily tracks the file position, plus, as an optimization, the disk block containing the file position. (The full C `struct` definition is shown in Figure 2.) Open file structures point to v-nodes that identify the actual file on disk; an example v-node is on the right. SFS omits all the Unix features that involve "duplication" of file descriptors (e.g. `fork` and `dup`), so open-file objects do not need to carry a reference count. However, each file on disk might be opened several times at once, so the v-nodes include reference counts and students need to reason about concurrent access to a single file.

[1]Also known internally as the Shark File System.

```
typedef uint32_t block_id;
typedef struct sfs_block_t
{
    unsigned char type[4];
    block_id prev_block;
    block_id next_block;
    unsigned char data[BLOCK_DATA_SIZE];
} sfs_block_t;
```

Fig. 3. SFS Disk Blocks

```
typedef struct sfs_dir_entry_t
{
    block_id first_block;
    uint32_t size;
    char name[SFS_FILE_NAME_SIZE_LIMIT];
} sfs_dir_entry_t;
```

Fig. 4. SFS Directory Entries

SFS filesystems are stored as disk images within ordinary files on the host OS's file system. The library also provides an ancillary API to format and mount SFS disk images (only one image at a time). An SFS disk image is split into 512-byte disk blocks, regardless of the host OS's disk block size. The first block in the file (block 0) is the filesystem's 'super block,' containing metadata about the entire filesystem, such as its overall size and references to the beginning of the (only) directory and the list of free blocks.

Like the real FAT file system family, files are represented as linked lists of blocks, using block numbers as pointers. (Block number 0 is reused as a null pointer/end-of-list marker, since no list ever needs to point back to the super block.) Unlike FAT, these linked lists are stored intrusively, occupying the first 12 bytes of each disk block, as shown in Figure 3. Intrusive linked lists are familiar to our students from an earlier assignment.

Also like the real FAT family, all metadata about a file is stored in its directory entry. Per-file metadata is minimal, as shown in Figure 4.

V. Programming Assignment

The educational goal of the SFS assignment is to teach parallelism, not file system implementation. To minimize the secondary challenge of working with the complex data structures of a file system, and also to make it feasible to complete the assignment in only one week, we provide students with a "starter" implementation. It is not quite feature-complete and lacks any code to support concurrent access. Students also receive a test harness, a set of test "traces" (see Section V-C), and a consistency checker for SFS's on-disk data structures.

A. Missing Features

The SFS starter code already implements most of the basic file system APIs: `open`, `close`, `read`, `write`, and `unlink`. As a warm-up, students are asked to implement `getPos`[2], `seek`, and `rename`.

In order to implement these functions, students must familiarize themselves with the open-file structure (for `getPos`), the chain of blocks that hold the contents of a file (for `seek`), and the super block with its embedded directory (for `rename`). This covers most of the starter code, and all of the parts that will need to be updated to make the file system thread-safe.

[2]get file position; separated from `seek` for pedagogical clarity.

B. Thread Safety

Having completed the missing features, students are then asked to augment SFS with support for concurrent access using multiple threads. This part of the exercise challenges students to identify all of the shared data structures involved in SFS, identify *how* each shared data structure can be accessed concurrently, and then develop an approach for making all of these concurrent accesses safe.

Students are given guidance suggesting an incremental approach to the problem, with three levels of increasingly fine-grained locking: first a single lock over the whole file system, then moving to locks on individual files, and finally to locks on individual file descriptors. (A common mistake is to think that locking on individual files completely supersedes locking of the overall file system; see Section VII.) We do not suggest this, but some students also think to use reader-writer locks, which can enable greater parallelism between read accesses to the same file or file descriptor.

C. Test Traces

Several assignments in our class are tested using "trace files" which describe a sequence of operations to be performed by student-written code. For example, the "malloc lab" assignment asks students to implement a basic heap allocator. [18] Trace files for malloc lab describe a series of calls to `malloc`, `realloc`, and `free`. Each trace file is designed to challenge the students' work in some way—continuing the example, malloc lab includes traces that will cause a first-fit allocator to waste a lot of memory on heap fragmentation. Along with the assignment, students receive a set of trace files and a test driver that interprets them. They are encouraged to write additional trace files for experimentation or debugging.

In our other assignments, the trace files are not *programs*. The syntax does not provide for conditionals, looping, variables, or concurrency. SFS tests need all four of those features in order to challenge student code with complex patterns of concurrent requests while remaining reasonably easy to develop. Therefore, for SFS, traces are actual programs in the Lua scripting language [17]. This language's interpreter is designed to be easy to embed in a larger program and extend with library routines that call back into the larger program.

Lua proper includes coroutines, but not true concurrent execution. There are several third-party extensions that add some form of concurrency. We selected the "Lanes" extension [19], which runs a separate Lua interpreter in each of several OS-level threads. Coordination between threads is accomplished via message passing and Linda objects [20]. Lanes' architecture allows us to challenge student code with truly concurrent requests, not just interleaved requests.

Our test driver's embedded Lua interpreter exposes the SFS API directly to trace scripts. They also have access to the

```
local fd = check(disk.open("small"))
assert(check(disk.getPos(fd)) == 0)

local data = "hello␣world"
local written = check(disk.write(fd, data))
assert(written == #data)
assert(disk.getPos(fd) == #data)
```

Fig. 5. Example of an **A** trace

```
local laneproc = lanes.gen("string,disk",
    function(tid)
        local fds = {}
        for i = 1, N_FILES do
            local fname = string.format(
                "f-%d-%d", tid, i
            )
            fds[i] = check(disk.open(fname))
        end
        for i = 1, N_FILES do
            assert(check(
                disk.write(fds[i], CONTENTS)
            ) == #CONTENTS)
            disk.close(fds[i])
        end
        return true
    end
)

local lanes = {}
for i = 1, N_THREADS do
    lanes[i] = laneproc(i)
end

for i = 1, N_THREADS do
    check(lanes[i]:join())
end
```

Fig. 6. Example of a **C** trace to **create** files

bulk of the Lua standard library, but not to modules that we felt would confuse students (e.g. the coroutine library—one form of concurrency is enough) or that could disrupt the test driver itself (e.g. most of the built-in I/O functions). After each test trace completes, the test driver automatically runs the consistency checker on the SFS disk image produced by that trace.

We provide three sets of traces to the students. Set "**A**" tests the features of the starter code, as well as the missing features that students are expected to implement, without any use of concurrency. For example, Figure 5 shows the core of an **A** trace to test whether a file can be opened and written to.

Sets "**B**" and "**C**" test concurrent operations. They are paired: each **C** trace performs some sequence of concurrent operations, and the matching **B** trace does the same overall operations but in a sequential fashion. Thus, students can be confident that if a **C** trace fails but the matching **B** trace succeeds, there is a problem with their handling of concurrency rather than a problem with their implementation of the operations themselves.

Figure 6 shows the core of a **C** trace that creates many files concurrently. The library function `lanes.gen` converts a thread procedure into a *lane factory*, here given the name `laneproc`. Each subsequent call to `laneproc` starts a new thread that calls the function that was passed to `lanes.gen`. Thus, this trace runs `N_THREADS` parallel threads each of which creates `N_FILES` files. The complete trace continues with code to verify that all the files were created with the expected contents. The corresponding **B** trace creates all the same files, but sequentially.

The point of the **C** traces is to challenge student code with a stream of operations that should be completed concurrently, so they do not do any serialization themselves. Often, this means there is more than one final on-disk state that qualifies as "correct." For example, when two threads write to a file using the same file descriptor, we can verify that the data from both writes appears in the file, but we cannot require either one or the other to come first.

Many of the test traces are based on tests written for production-grade filesystems as part of the "xfstests" test suite [21]. Since the POSIX filesystem API is a superset of the SFS API, tests that are correct for a production Unix filesystem will also be correct for SFS, as long as they use only the features of SFS. In particular, tests from xfstests already anticipate multiple correct results. xfstests is written in a mixture of Unix shell and C, using the full capabilities of both languages freely. Use of a complete scripting language for SFS traces made it much easier to port tests from xfstests to SFS.

D. Facilitating Debugging

Because SFS is implemented as an ordinary user-mode process, students can use standard debugging tools such as `gdb` and `valgrind`, both of which they are encouraged to use throughout the course. They can also instrument their code with logging and assertions. Students can run each of the test traces in isolation (with or without a debugger attached), write their own test traces, or write their own C programs that are linked against the SFS library. (Students not experienced with Lua may find writing C programs easier.)

VI. Evaluation

We rely primarily on "autograding"—automatic evaluation of student code against a test suite—for SFS, as we do for the other assignments in our course. Instructors also review student code by hand and provide individualized feedback to the students; however, students' grades for each assignment are mostly determined by its test suite.

To autograde the SFS assignment, we use the same test traces that students are given along with the starter code. The overall grading scheme is shown in Table I. Points are split equally between correctness and concurrency, but concurrency is only evaluated if the student gets full marks for correctness.

The autograding system we use restricts each grading task to a single processor. Therefore, the ratio of **C** trace to **B** trace execution time does not effectively measure how much concurrency is achieved. Instead, we evaluate performance for SFS deterministically, using an approach similar to Tareador [22]. We use a custom LLVM pass [23] that instruments the student-modified SFS library to produce a *task graph* [24] of the execution of each **C** trace. From these task graphs, we can

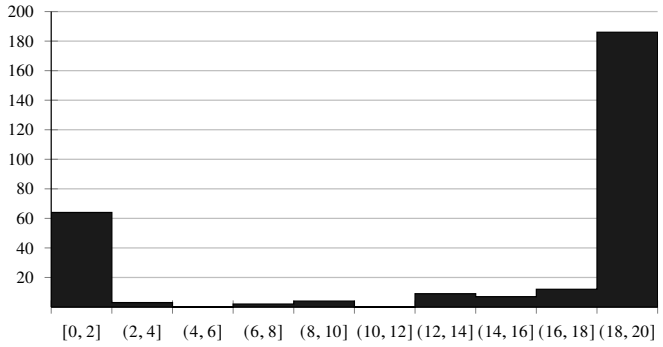

Fig. 7. Concurrency Scores for Student Submissions (N=287)

TABLE I
GRADE BREAKDOWN FOR THE SFS ASSIGNMENT

	Points possible
A trace correctness	5
B trace correctness	8
C trace correctness	7
Concurrency score	20
Total	40

$$\text{Concurrency score (capped at 20)} = 100 \times \sum_{i \in \text{C traces}} \left(1 - \frac{s_i}{w_i}\right) \quad (1)$$

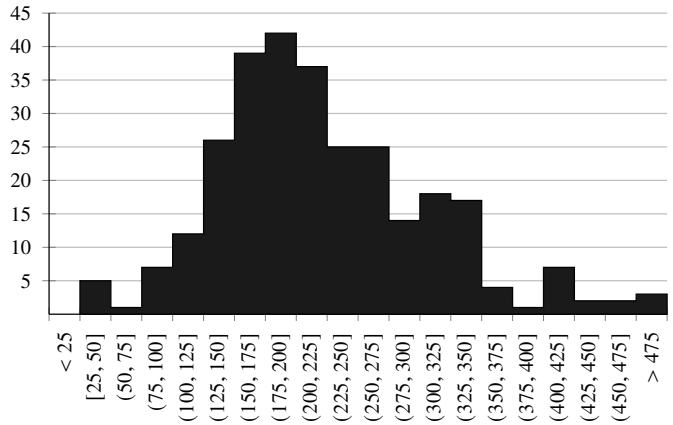

Fig. 8. Additional Lines of Code in Student Submissions (N=287) (min 27, max 681, average 223)

TABLE II
STUDENT RESPONSES TO POST-ASSIGNMENT SURVEY

Prompt	Mean	Out of
Total time spent	13 h	
Concurrency score	15.8	20
Self-assessed preparation for:		
Implementing APIs	3.64	5
Code analysis	3.52	5
Refactoring for Parallelism	2.81	5

estimate the total computational work performed and the length of the critical serial path for each trace. From these, we calculate a *concurrency score* for the whole assignment according to Equation (1), where s_i is the critical path length and w_i the total computational work for **C** trace i. This score estimates the concurrency that the student's work could achieve if executed on an ideal parallel processor.

The autograder also analyzes each task graph to detect data races in student code [25]. When a race is detected, the tool reports the memory address being accessed without sufficient synchronization, along with the threads and functions involved. Typical output is shown below.

```
Conflicting access address:
    55a428f04230(Idx:3) in (Contech:Task) --
    (4:14) and (3:14)
 BB#133
145, sfs_read, sfs-disk.c:486
133, sfs_read, sfs-disk.c:458
```

We are continuing to improve these reports, as well as investigating how our race detector compares to other race detectors (e.g. valgrind and LLVM's built-in TSan instrumentation).

VII. Learning Impact

To assess the effectiveness of the new SFS assignment, we first consider student grades from the Fall 2024 session of Introduction to Computer Systems. Figure 7 shows a histogram of all 287 students' concurrency scores. A large majority of the students got the maximum possible score, thus demonstrating a deeper understanding of parallelism than what was required for the old proxy lab. However, roughly one-fifth of the students got the *lowest* possible score, indicating that they were unable to pass the correctness tests.

Figure 8 shows a histogram of how much new code students wrote for the assignment. By way of comparison, the starter implementation of SFS was 615 lines long, or 945 lines counting header files. Most students could complete the assignment using only 150–300 lines of additional code. Longer student submissions may have more complex locking strategies or may simply include more comments.

After the Fall 2024 students completed the SFS assignment, we surveyed them informally about it. We asked them to report how long they spent on the assignment, what their final concurrency score was, and, on a scale of 1 to 5, how well they thought the course had prepared them for each of three aspects of the assignment. 60 out of 287 students responded to our questionnaire; the results are shown in Table II.

Students reported spending an average of thirteen hours on SFS, over the course of one week, which is on par with the workload of other assignments in this course. The mean concurrency score for students responding to the survey is somewhat above the mean concurrency score for the whole class (15.8 versus 14.6); this suggests the survey results are slightly biased toward those students who successfully completed the first half of the assignment (i.e. full marks for correctness).

Students report themselves reasonably well prepared for the first half of the assignment, i.e. the warm-up exercise of implementing APIs, and identifying what will need to change to implement safe concurrent operation. They find actually

carrying out the necessary changes to be more difficult.

Finally, we reviewed a sample of student submissions for common mistakes. These consistently revolve around introducing gaps in the locking strategy when passing from one stage to the next of the incremental approach suggested in assignment guidance (see Section V-B). For example, students may achieve correct (but not concurrent) operation using one big lock, and then replace that with a lock for each open file, not realizing that they still need separate locks for updates to data structures which are not files, such as the root directory and the list of free blocks.

Overall, we consider the new SFS assignment an improvement to our course's presentation of parallel programming. Most students are able to progress through the full assignment and develop a well-performing parallel implementation of SFS. In the near term, a priority is to understand why one-fifth of students fail to complete the first half of the assignment, and improve our teaching materials to cover any gaps we discover. We are also reviewing how the starter code might be refactored to reduce inessential cognitive load on students.

VIII. Possible Extensions

The SFS assignment concentrates on one aspect of parallel programming: safe concurrent access to shared data. While there are other aspects that would also be valuable to address early in the undergraduate curriculum, such as understanding the difference between inherently parallel and inherently serial algorithms, and how to efficiently divide up work among parallel tasks to minimize communication overhead, we feel that these are complex enough to deserve exercises devoted to them.

The SFS assignment is still quite new and we have kept it minimal for the time being, but we have considered a few possibilities for future additions. Sticking to the theme of safe concurrent access to shared data, these might add interesting complications to the core concurrency challenge, or give students a glimpse of what awaits them in upper-level systems courses.

A. Directories

As provided to students, SFS does not have any support for subdirectories. The "root directory" of the file system is directly embedded in the super block; it cannot even grow into additional blocks, as files can. However, we left room for a growable root directory and for the addition of subdirectories. We see this primarily as "teaser" material for upper-level systems courses, to be offered as an optional exercise for students doing well in the class. Students would need to modify the SFS `open` API, add the `mkdir` API, and define a new on-disk data structure for the contents of a directory, but they would not need to change existing on-disk data structures. Path walking concurrent with directory creation and deletion is a challenging algorithm to implement, but it is a natural progression from what students already accomplish in the lab. We tested and prototyped the necessary changes, and left comments in the code pointing out the possibility, even though it is not required in the current version of the assignment.

B. Tree-Structured File Allocation

In SFS, a file is identified by its directory entry, which directly holds the file's size and the number of its first block (as shown in Figure 4). The rest of the file can be located by walking a doubly-linked list embedded in each block (as shown in Figure 3). This is simple and straightforward for students to understand. Sequential reading and writing is made efficient by caching the current block pointer in the open-file structure. However, `seek` operations take time and disk accesses proportional to the size of the change in the current file position.

File systems that offer $O(1)$ random access to files use completely different on-disk data structures. For example, the classic BSD Fast File System [26] has each directory entry point to a skewed tree, consisting of the inode and the indirect blocks, which in turn points to the file's data blocks. Replacing SFS's intrusive linked lists with FFS-style skewed trees would alter almost every aspect of the on-disk data structure and require extensive changes to its logic as well. We have prototyped this change ourselves, but we currently think it is not a feasible challenge for students at the level of our course. It might be an interesting exercise for more advanced students, perhaps in an upper-level data structures or operating systems course.

IX. Conclusion and Future Work

This work presented a programming assignment which trains and assesses students' understanding of concurrent access to shared data, using a simple file system as its vehicle. We also showed a technique based on compile-time instrumentation for deterministically scoring the performance of parallel code without requiring a multi-core autograding system. This technique is also capable of identifying data races in student code. Finally, we experiment with a widely used scripting language (Lua) as a means to express concurrency in test cases for autograding.

We plan on continuing to improve both the SFS assignment and the associated course material, particularly our lectures on parallel execution, as we better identify where students are now struggling and where they are left with misconceptions. We are also working on improving the autograder to ensure consistency in the performance metrics and to provide more helpful output when races are detected.

Acknowledgments

We want to thank Michael Melville and CMU's Eberly Center for their support through IRB approvals and educational analysis. We are also grateful to the teaching assistants who have managed the assignment and its testing, especially Parth Sangani and Caleb Oh. Finally, we want to thank all our students for their work, perseverance, and feedback as we continue to improve the course and this assignment.

References

[1] R. E. Bryant and D. R. O'Hallaron, "Introducing computer systems from a programmer's perspective," in *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '01. New York, NY, USA: ACM, 2001, pp. 90–94. [Online]. Available: http://doi.acm.org/10.1145/364447.364549

[2] R. K. Raj, C. J. Romanowski, J. Impagliazzo, S. G. Aly, B. A. Becker, J. Chen, S. Ghafoor, N. Giacaman, S. I. Gordon, C. Izu, S. Rahimi, M. P. Robson, and N. Thota, "High performance computing education: Current challenges and future directions," in *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, ser. ITiCSE-WGR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 51–74. [Online]. Available: https://doi.org/10.1145/3437800.3439203

[3] S. Diesburg and A. Berns, "Fileshark: A graphical file system visualization tool," in *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1359. [Online]. Available: https://doi.org/10.1145/3328778.3372648

[4] B. Mechtly, F. Helbert, D. Cox, and Z. Hastings, "The visible file system: an application for teaching file system internals," *J. Comput. Sci. Coll.*, vol. 34, no. 4, p. 24–31, Apr. 2019.

[5] L. Thompson, J. Clarke, and R. Sheehan, "edufuse a visualizer for user-space file systems," in *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 549–550. [Online]. Available: https://doi.org/10.1145/3341525.3393989

[6] R. Marmorstein, "Spifs: short project instructional file system," *J. Comput. Sci. Coll.*, vol. 36, no. 3, p. 111–120, Oct. 2020.

[7] E. Nieh, Z. Zhang, and J. Nieh, "ezfs: A pedagogical linux file system," in *Proceedings of the 56th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '25. New York, NY, USA: Association for Computing Machinery, 2025.

[8] J. A. Carneiro Neto, A. J. Alves Neto, and E. D. Moreno, "A systematic review on teaching parallel programming," in *Proceedings of the 11th Euro American Conference on Telematics and Information Systems*, ser. EATIS '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3544538.3544659

[9] T. de Jesus Oliveira Duraes, P. Sergio Lopes de Souza, G. Martins, D. Jose Conte, N. Garcia Bachiega, and S. Mazzini Bruschi, "Research on parallel computing teaching: state of the art and future directions," in *2020 IEEE Frontiers in Education Conference (FIE)*, 2020, pp. 1–9.

[10] D. J. Conte, P. S. L. de Souza, G. Martins, and S. M. Bruschi, "Teaching parallel programming for beginners in computer science," in *2020 IEEE Frontiers in Education Conference (FIE)*, 2020, pp. 1–9.

[11] L. B. A. Vasconcelos, F. A. L. Soares, P. H. M. M. Penna, M. V. Machado, L. F. W. Góes, C. A. P. S. Martins, and H. C. Freitas, "Teaching parallel programming to freshmen in an undergraduate computer science program," in *2019 IEEE Frontiers in Education Conference (FIE)*, 2019, pp. 1–8.

[12] P. Czarnul, M. Matuszek, and A. Krzywaniak, "Teaching high–performance computing systems – a case study with parallel programming apis: Mpi, openmp and cuda," in *Computational Science – ICCS 2024: 24th International Conference, Malaga, Spain, July 2–4, 2024, Proceedings, Part VII*. Berlin, Heidelberg: Springer-Verlag, 2024, p. 398–412. [Online]. Available: https://doi.org/10.1007/978-3-031-63783-4_29

[13] J. Adams, R. Brown, and E. Shoop, "Patterns and exemplars: Compelling strategies for teaching parallel and distributed computing to cs undergraduates," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, ser. IPDPSW '13. USA: IEEE Computer Society, 2013, p. 1244–1251. [Online]. Available: https://doi.org/10.1109/IPDPSW.2013.275

[14] R. A. Brown, J. C. Adams, C. Ferner, E. Shoop, and A. B. Wilkinson, "Teaching parallel design patterns to undergraduates in computer science," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 547–548. [Online]. Available: https://doi.org/10.1145/2538862.2538875

[15] G. E. Blelloch, Y. Gu, and Y. Sun, "Teaching parallel algorithms using the binary-forking model," in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2024, pp. 346–351.

[16] S. Vargas-Pérez, "Teaching performance metrics in parallel computing courses," in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2024, pp. 385–390.

[17] L. H. de Figueiredo, R. Ierusalimschy, and W. Celes Filho, "The design and implementation of a language for extending applications," in *Proceedings of XXI Brazilian Seminar on Software and Hardware*, 1994, pp. 273–283. [Online]. Available: https://www.lua.org/semish94.html

[18] B. P. Railing and R. E. Bryant, "Implementing malloc: Students and systems programming," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 104–109. [Online]. Available: https://doi.org/10.1145/3159450.3159597

[19] A. Kauppi and B. Germain, *Lua Lanes—multithreading in Lua*, 2007. [Online]. Available: https://lualanes.github.io/lanes/

[20] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and friends," *Computer*, vol. 19, no. 8, pp. 26–34, 1986.

[21] Z. Lang, D. J. Wong, C. Brauner *et al.*, "The fs qa suite," 2001–present. [Online]. Available: https://git.kernel.org/pub/scm/fs/xfs/xfstests-dev.git/about/

[22] E. Ayguadé, R. M. Badia, D. Jiménez, J. R. Herrero, J. Labarta, V. Subotic, and G. Utrera, "Tareador: a tool to unveil parallelization strategies at undergraduate level," in *Proceedings of the Workshop on Computer Architecture Education*, ser. WCAE '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2795122.2795123

[23] B. P. Railing, E. R. Hein, and T. M. Conte, "Contech: Efficiently generating dynamic task graphs for arbitrary parallel programs," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 2, pp. 25:1–25:24, Jul. 2015. [Online]. Available: http://doi.acm.org/10.1145/2776893

[24] V. S. Adve and R. Sakellariou, "Compiler synthesis of task graphs for parallel program performance prediction," in *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers*, ser. LCPC '00. London, UK, UK: Springer-Verlag, 2001, pp. 208–226. [Online]. Available: http://dl.acm.org/citation.cfm?id=645678.663959

[25] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, p. 558–565, Jul. 1978. [Online]. Available: https://doi.org/10.1145/359545.359563

[26] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A fast file system for unix," *Transactions on Computer Systems*, vol. 2, no. 3, pp. 181–197, 1984.