SPIFS: A Short Project Instructional File System

Author Information will go here

ABSTRACT

Understanding file systems is a key objective of both operating systems and systems programming courses. In order to enhance student understanding of file systems, it is common to ask students to implement parts of a file system using a high-level programming language. In this paper, we introduce a self-contained file system that can serve as the basis for such projects.

The system includes a set of C++ classes that form the base of a very simple file system. The classes provide only the data structures and interface needed to implement the file system, but not implementations of the key file system operations. It also includes automated test cases for each of ten file system operations that students should implement.

INTRODUCTION

One of the key topics in many operating systems courses is the organization and design of file systems. For example, the CS2013 ACM curriculum recommendation[1] suggests that students master the following learning outcomes related to file systems:

> *1. Describe the choices to be made in designing file systems.*
> *2. Compare and contrast different approaches to file organization, recognizing the strengths and weaknesses of each.*
> *3. Summarize how hardware developments have led to changes in the priorities for the design and the management of file systems.*
> *4. Summarize the use of journaling and how log-structured file systems enhance fault tolerance.*

File system operations can also be an important topic in Systems Programming classes, which typically focus on understanding and using system calls and low-level libraries to accomplish system tasks.

However, file systems contain many abstract data structures and concepts that can be difficult for students to grasp. Students who are used to searching for everything in the age of Google may not even have much exposure to the use of a hierarchical directory structure. Diesburg and Berns[8] point out that one of the objectives of file system design is to hide implementation details from the user. While that creates a much better user experience, it also prevents students from mastering file system concepts through normal use. This is especially true on mobile devices, which make the file system nearly invisible. To address this, they provide a tool (Fileshark[8]) that allows students to visualize file system operations.

One excellent way to introduce students to file systems is to have them implement parts of a file system. Many operating systems courses include a project where students implement or extend parts of an operating system. For example, Andrew Tanenbaum introduced the Minix operating system[5] in 1987. Minix provides students with a fully functional clone of Unix that they can modify in clearly defined ways to explore operating systems concepts. The Nachos operating system[2] was released in the early nineties and provides a similar experience in which students implement synchronization constructs, file operations, and a virtual memory system. Nachos is based on a simulated MIPS system and, while still usable, has become a bit dated.

A team at Stanford produced a more modern version of Nachos called Pintos[3] that is written in C rather than C++. Rather than rely on a MIPS simulation, Pintos produces executable x86 code. It also comes with a set of test cases that students can use to check their implementations as they develop the software. Similarly, a team at Harvard produced the OS/161 project[4]. It has been suggested that OS/161 supports more realistic projects than Pintos, but also that it takes more time for students to complete the projects[6]. Another project of this type is the xv6 project[9] at MIT, which provides a modern version of Unix System VI.

Each of these instructional operating systems includes a component in which students design all or part of a file system. However, the file system is heavily tied to the overall framework and the overhead that entails. As such, the file system project requires a significant investment of class time and student effort. Because implementing even the simplest operating system involves solving complex and abstract concurrency problems, these projects can be some of the most challenging assignments students tackle in an undergraduate program. In fact, as Freedman points out[12], this difficulty is one reason operating systems projects are so valuable – in addition to learning course content, students learn important lessons about programming, software design and project management. However, the time investment required to complete these projects means that they are suitable only for courses where the instructor can dedicate a major component of class time to supporting the project. Furthermore, these projects tend to be cumulative – an error in one of the first projects can affect student success in subsequent projects.

Another drawback to these projects is that they often require extensive setup. For example, Nachos requires a MIPS cross-compiler and tools for converting files to a specialized executable format[2]. Other systems require the use of virtual machines or simulators such as Bochs or Qemu[4,9]. This makes them even more time intensive for students and faculty – and also increases the number of places where something can go wrong.

An alternative approach is to have students extend real world operating systems. Hess and Paulson[13]; Laadan, Nieh, and Viennot[14]; and Nutt[15] describe different projects that use Linux kernel extensions that are appropriate for undergraduate coursework. Schmidt, Polze, and Probert[16] describe projects that use the Windows kernel. Andrus and Nieh[20] instead use the Android mobile operating system. These projects have the advantage of realism, but the disadvantage of the additional complexity such realism requires.

A simpler approach to providing hands-on experience for operating system concepts is to assign projects that focus on a single concept or set of learning objectives. For example, Bynum and Camp[7] describe the use of the BACI concurrent language to teach synchronization principles such as deadlocks and mutual exclusion. Goldweber, Davoli, and Jonjic[10]; Buck and Perugini[11]; and Robbins and Robbins[[19] describe different approaches to teaching scheduling using simulations. Sibai, Ma, and Lill[17] used the MOSS simulator to introduce students to memory management algorithms.

However, there seem to be few projects focused on solely on file system design. Instead, research has focused on ways to allow students to visualize file systems such as Fileshark[8] and the visualization tool described in [18].

A BASIC FILE SYSTEM

To address this gap, we present SPIFS (the Short Project Instructional File System). SPIFS provides a simple (less than 300 lines of code) self-contained base file system which students extend to implement ten basic file system operations: format, create, list, remove, rename, copy, open, close, read, and write. The file system is loosely inspired by the disk format used by Commodore DOS [21] to organize files on a floppy disk into a single flat directory, but without the complexity of disk geometry and block allocation. Spifs is implemented as a set of C++ classes that can be treated as a software library. Test cases consist of programs compiled against the library that create a file system object and then call the ten basic file system operations on that object.

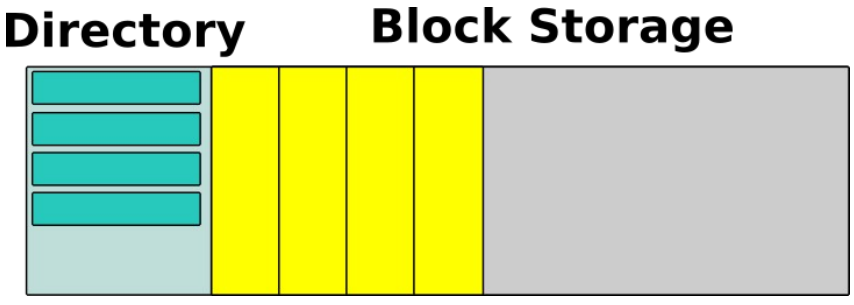

Figure 1: File System Layout

In SPIFS, the “disk” consists of a single dynamically allocated array of bytes. As shown in figure 1, we divide this array into two logical sections: a directory listing and a block storage.

We make several simplifying assumptions about the file system to reduce the complexity and make the project as accessible to students as possible. First, we assume that all files fit into a single 1KiB block. Second, we assume that the file system supports a single directory located at the beginning of the “disk”. Third, we assume all operations are single-threaded.

These assumptions can be weakened to extend the project and adjust the level of difficulty to suit the needs of a particular course.

Conceptually, the directory structure consists of a sequential sequence of fixed-size entries. The structure of a single directory entry is illustrated in figure 2. Each entry contains a 16-byte name consisting of 15 alphanumeric symbols plus a terminating null byte, a four byte integer representing the size of the file, and a four byte integer representing the start of the file block within the array.

We restrict filenames to alphanumeric characters and limit the size of the disk to 4GiB. For a 4MiB disk image, this gives a file system with about 4002 twenty-four byte directory entries and 4002 1Kib file blocks.

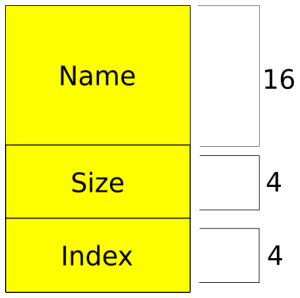

Figure 2: SPIFS Directory Entry

TESTING

As students work on each of the ten operations, they can check their work by running twelve test cases provided with the base system. All twelve tests can be run using the "make test" command or test can be compiled and run individually from the tests/ folder. In addition, we encourage students to write their own tests and add them to the test suite by extending a "BaseTest" class. Providing test cases gives students immediate feedback about their progress. While the test cases aren't exhaustive, they do catch many of the ways students are likely to fail in each task.

OPERATIONS

The first operation students implement is a "format" operation. Since the file system consists of a single directory, the formatting task is trivial. We instruct the students to write "*" characters over all the filenames of the directory. The easiest way to do this is to simply copy asterisks over the whole disk, but the test case only checks the filenames, so other solutions are also acceptable.

The second task is to implement the "create" operation. This creates a new empty file by finding a free directory entry, copying the correct filename into that entry, finding a free file block, and initializing the size and index properties to correct values. Due to the simplicity of the file system, students do not need to create any additional data structures in order to allocate file blocks. They can simply map each directory entry to a corresponding file block. However, they must also properly handle error conditions, such as the existence of an existing file with the same name, an invalid filename, or a full directory.

The third task, implementing the "list" operation, requires students to iterate through the directory, printing the name and size of valid directory entries. In order to allow the testing framework to check that the list is correct, the function returns a string, rather than printing to the console.

The fourth task requires students to implement the "rename" operation. To do this, student must locate the provided filename within the directory and change it to the new name. They must properly handle error conditions, such as renaming a file to an existing name, and attempting to rename a file that has been deleted or does not yet exist.

The fifth task requires students to delete a file. This can be handled by replacing the filename with "*" characters. As with real file systems, deleting the file does not necessarily delete the data associated with that file. The test cases verify that students check whether the file exists and has not already been deleted.

The remaining tasks require students to implement a data structure for storing "file handle" objects represent an open file on the system. Each file handle keeps track of the position at which subsequent reads or writes should begin. Implementing the "Open" task requires students to create a file handle, initialize it to point to the beginning of a file block, and return a file descriptor representing that object. Students must verify that the file exists and that there is room in the data structure for storing the file handle object.

Implementing the "Close" task frees the resources allocated for the file handle. The test cases verify that students properly handle closing an already closed file or using an invalid file descriptor.

The "Read" task asks students to implement a function that reads a fixed number of bytes from an open file handle into a

buffer while the “Write” task asks them to write to the file from the buffer. In both cases, students must verify that the file descriptor is valid and that the number of bytes to be transferred does not exceed the size of the file.

EXTENDING THE PROJECT

The base project can be assigned either to individuals or as a group project and is simple enough that most students can complete it within a week. It can be extended in many ways to make the project more challenging and to cover more file system concepts. Here are some relatively simple extensions that can make the content richer and more interesting:

1. Use a file allocation table to add support for multi-block files

A file allocation table represents files as “chains” where each entry in the table consists of a single integer value representing either a free block, or the position of the next block in the chain. Blocks in the free storage of the disk are mapped one-to-one to FAT entries. Students can reserve a section of the disk array to hold the FAT and treat the directory index as the location of the first block. Most changes to the file system operations are trivial except for the create function, which must now search the FAT table for a free block. The delete, read, and write operations also require modification to correctly iterate through the linked list of blocks represented by the chain.

2. Implement multiple directories using inodes

Instead of treating file blocks as indistinguishable blocks of data, students can treat them as structures containing a “type” field which can either be “free”, “file”, or “directory”. In this manner, the system can represent a hierarchical directory structure with a “master directory” at the top level and subdirectories stored in file blocks. This requires the addition of new operations for creating and removing directories. It also requires extensive modification to the “List”, “Open” and “Delete” operations to allow them to support paths containing multiple directories. The “Copy” operation must also be modified to allow students to move files from one directory to another. Optionally, the “Rename” function can be made to double as a “Move” function – a choice made by many modern operating systems, such as Linux.

3. Add file permissions to the directory entry object

Since SPIFS does not have a concept of users or processes, it doesn’t make much sense to implement access control lists or traditional Unix group and user permissions, but students can still modify the directory structure to allow files to be marked as “hidden”, “read-only”, or “append-only”. This requires a minor change to the directory structure and a slight alteration of the list, read, and write operations.

4. Add support for multi-threading

In the base project, all operations and test cases are single threaded, but it is fairly straightforward to use POSIX threads to enable parallel or asynchronous I/O operations. More complex tasks (including requiring students to implement a complete Readers/Writers system) are also possible extensions.

5. Ensure file system integrity with checksums

One of the main concerns of a real file system is ensuring that data retains integrity and that storage errors are detected quickly. A simple way to extend the project is to replace the simple undifferentiated file blocks with a structure that adds a length field and a checksum or CRC code. Writing realistic test cases for this extension is somewhat tricky, but flipping random bits in the block storage area and verifying changes to the checksum can give a rough simulation of media errors.

6. Add routines that transfer data from the host operating system

It is relatively trivial to add functions that copy files from the host operating system into Spifs and back out again, but adding these functions allows students to perform more interesting experiments with their file system such as running a performance evaluation using different block sizes.

CONCLUSIONS

In our undergraduate operating systems course, we require students to implement projects from an instructional operating system at the beginning of the semester and use SPIFS as the final project of the semester. In our offering of the project, we

split the project into two parts. In part one, students complete the ten base operations. In part two, students select an extension of their choice to implement. Both parts are graded for credit. In our experience, students can reasonably complete the project in a single week. Student feedback indicates that students view SPIFS as a much easier and more enjoyable project than the instructional operating system projects. Since it is independent of other projects it gives them a chance to "reset" and earn points toward their final grade that do not depend on previous work. In addition to giving students hands-on practical experience with file systems, SPIFS reinforces important programming concepts such as the importance of testing, data structure design, and source code management.

Grading and administering the project is also easy, since the source code requires no tools other than a C++ compiler and the score can be calculated largely from the number of test cases successfully passed. For better test coverage, the tests can be repeated using different disk image sizes.

OBTAINING SPIFS

The latest version of SPIFS can be cloned from gitlab.com using the following command:

git clone AUTHOR INFORMATION REMOVED

A sample handout for the project can be found in OpenDocument format at AUTHOR INFORMATION REMOVED

WORK CITED

[1] Joint Task Force on Computing Curricula. 2013. Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science. Association for Computing Machinery (ACM) and IEEE Computer Society. ISBN: 9781450323093.

[2] Wayne A. Christopher, Steven J. Procter, and Thomas E. Anderson. 1993. The Nachos instructional operating system. In Proceedings of the USENIX Winter 1993 Conference (USENIX'93). USENIX Association, USA, 4.

[3] Ben Pfaff, Anthony Romano, and Godmar Back. 2009. The pintos instructional operating system kernel. In Proceedings of the 40th ACM technical symposium on Computer science education (SIGCSE '09). Association for Computing Machinery, New York, NY, USA, 453–457. DOI:https://doi.org/10.1145/1508865.1509023

[4] David A. Holland, Ada T. Lim, and Margo I. Seltzer. 2002. A new instructional operating system. SIGCSE Bull. 34, 1 (March 2002), 111–115. DOI:https://doi.org/10.1145/563517.563383

[5] Andrew S Tanenbaum. 1987. A UNIX clone with source code for operating systems courses. SIGOPS Oper. Syst. Rev. 21, 1 (Jan. 1987), 20–29. DOI:https://doi.org/10.1145/24592.24596

[6] Thomas Anderson and Michael Dahlin, Operating Systems: Principles and Practice Web Site. "Labs". Retrieved 9 July, 2020. http://ospp.cs.washington.edu/labs.html.

[7] Bill Bynum and Tracy Camp. 1996. After you, Alfonse: a mutual exclusion toolkit. In Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education (SIGCSE '96). Association for Computing Machinery, New York, NY, USA, 170–174. DOI:https://doi.org/10.1145/236452.236533

[8] Sarah Diesburg and Andrew Berns. 2020. Fileshark: A Graphical File System Visualization Tool. In Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20). Association for Computing Machinery, New York, NY, USA, 1359. DOI:https://doi.org/10.1145/3328778.3372648

[9] Russ Cox, Frans Kaashoek, and Robert Morris. October 27, 2019. "Xv6: a simple, Unix-like teaching operating system." https://pdos.csail.mit.edu/6.828/2019/xv6/book-riscv-rev0.pdf

[10] Michael Goldweber, Renzo Davoli, and Tomislav Jonjic. 2012. Supporting operating systems projects using the μMPS2 hardware simulator. In Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education (ITiCSE '12). Association for Computing Machinery, New York, NY, USA, 63–68. DOI:https://doi.org/10.1145/2325296.2325315

[11] Joshua W. Buck and Saverio Perugini. 2019. An interactive, graphical CPU scheduling simulator for teaching operating systems. J. Comput. Sci. Coll. 35, 5 (October 2019), 78–90.

[12] Reva Freedman. 2020. Using an Operating Systems Class to Strengthen Students' Knowledge of C++. In Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20). Association for Computing Machinery, New York, NY, USA, 947–953. DOI:https://doi.org/10.1145/3328778.3366936

[13] Rob 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 '10). Association for Computing Machinery, New York, NY, USA, 485–489. DOI:https://doi.org/10.1145/1734263.1734428

[14] Oren Laadan, Jason Nieh, and Nicolas Viennot. 2011. Structured Linux kernel projects for teaching operating systems concepts. In Proceedings of the 42nd ACM technical symposium on Computer science education (SIGCSE '11). Association for Computing Machinery, New York, NY, USA, 287–292. DOI:https://doi.org/10.1145/1953163.1953250

[15] Gary J. Nutt. 2000. Kernel Projects for Linux (1st. ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[16] Alexander Schmidt, Andreas Polze, and Dave Probert. 2010. Teaching operating systems: windows kernel projects. In Proceedings of the 41st ACM technical symposium on Computer science education (SIGCSE '10). Association for Computing Machinery, New York, NY, USA, 490–494. DOI:https://doi.org/10.1145/1734263.1734429

[17] Fadi N. Sibai, Maria Ma, and David A. Lill. 2009. Teaching page replacement algorithms with a Java-based VM simulator. In Proceedings of the 14th Western Canadian Conference on Computing Education (WCCCE '09). Association for Computing Machinery, New York, NY, USA, 22–28. DOI:https://doi.org/10.1145/1536274.1536284

[18] Bruce Mechtly, Fritz Helbert, Dylan Cox, and Zachary Hastings. 2019. The visible file system: an application for teaching file system internals. J. Comput. Sci. Coll. 34, 4 (April 2019), 24–31.

[19] Steven Robbins and Kay A. Robbins. 1999. Empirical exploration in undergraduate operating systems. In The proceedings of the thirtieth SIGCSE technical symposium on Computer science education (SIGCSE '99). Association for Computing Machinery, New York, NY, USA, 311–315. DOI:https://doi.org/10.1145/299649.299795

[20] Jeremy Andrus and Jason Nieh. 2012. Teaching operating systems using android. In Proceedings of the 43rd ACM technical symposium on Computer Science Education (SIGCSE '12). Association for Computing Machinery, New York, NY, USA, 613–618. DOI:https://doi.org/10.1145/2157136.2157312

[21] Richard Immers and Gerald G. Neufeld. 1984. Inside Commodore DOS. Reston Publishing Company. Reston, VA, USA, ISBN: 0-8359-3091-2