# PolyFS: An Extensible, Underspecified, Pedagogical File System and Disk Emulator

**Foaad Khosmood and Phillip Nico**
**California Polytechnic State University**
foaad@calpoly.edu / pnico@calpoly.edu

## Abstract

In recent years, teaching file systems at the undergraduate level has become increasingly challenging. File systems, while essential to most computer systems, are almost never offered as an exclusive required course for a computer science curriculum. The topic is usually taught as part of a course on operating systems (OS), along with other introductory topics such as process management, scheduling, concurrency, deadlocks, distributed processing and multiprocessing. Introductory OS courses are typically required in computer science programs but the subject matter has grown tremendously in depth and case studies, making it difficult to spend any significant time on any individual topic. In this environment, professors can barely afford to cover the basics, let alone in-depth implementation of OS issues.

PolyFS is proposed as a solution to provide class assignments meant to exercise many of the established OS principles, while offering some level of design and implementation experience to students. Specifically, we stress three advantages for using PolyFS in an instructional setting: Variety, scalability and modularity.

We are developing PolyFS, a polymorphic file system assignment and corresponding storage device emulator compatible with a variety of operating systems. PolyFS specification includes a very basic block-device emulator making it easy to use regular Unix files, or even web-based services, as emulated disks. The file system itself is intentionally under-specified to allow instructors to focus on particular aspect of file systems in their assignments and students to actually design and implement important sub-systems using algorithms covered during lecture.

## Introduction and motivation

Introductory OS courses are challenging to teach partially due to the proliferation of operating system products, interfaces, and standards. To gain a good mastery of the concepts, most laboratory-based courses must involve significant low-level programming. Although there are exceptions such as DLXOS[1] where students implement an entire operating system, most concentrate on a few important subsystems out of necessity. There may be enough time in one term to cover all theory and concepts, but not enough to have programming assignments for each of them. Instructors could therefore be more efficient if they can find assignments that exercise a wide variety of OS concepts.

We believe file system implementation offers a good balance between a project that can realistically be done in a fraction of a college term, but also involve a wide variety of OS concepts and algorithms. Common file system principles overlap with those of OS and even broader computing systems[3]. Of the five major topics in OS courses (Processes, Scheduling, Memory management, Synchronization and I/O systems) all are present to some degree in file system implementation. Two popular undergraduate textbooks, *Tanenbaum & Woodhull*[4] and *Silberschatz, et al.*[5,] each dedicate several chapters to file systems. Recent OS courses at Stanford University[6] and University of California Berkeley[7], dedicate, two weeks and one week to file systems respectively.

Perhaps the most influential teaching-oriented file system is the MINIX file system[4], developed by Andrew Tanenbaum for educational purposes. It was adopted for early versions of Linux before the Extended file system became the Linux standard.

Exercising the students' skills is not the only thing a good assignment can do, however. A good assignment provides opportunities to assess achievement of student learning outcomes, and repeat offerings of the same assignment can form a basis for comparing the accomplishments of different cohorts of students. Genci[2] reports on experiences using a FAT file system assignment to assess student achievement.

In addition to the benefits of repeated use above, there is another, often unstated, benefit to assignment re-use: developing a good assignment is a lot of work. On the other hand, we have observed the phenomenon that over time assignments go stale and lose their assessment value; as more of the student population has done a particular assignment, that assignment becomes more a measure of population achievement than individual accomplishment[a].

We are developing PolyFS as a meta-specification for implementing many similar file systems that exercise the students' skills with respect to major OS topics.

Specifically as an assignment generation system, PolyFS offers variety, scalability, and modularity.

We define variety as the degree of change the assignment can undergo from term to term. We believe, much like midterms and finals, the same exact projects shouldn't be offered every term where they will be inevitably well known in the student community and may become somewhat routine for the instructors. At the same time developing new course material every term is not realistic. But if we can produce a set of reasonably divergent variations of the same assignment, perhaps we can mitigate some of the undesirable affects of repetition in assignments.

By "scalability", we refer to the scope of the deliverables. It's possible to have almost an entire file system already created with only a few minor features left to be implemented by students. This may be suitable for a lab or a small assignment. If the instructor chooses to, however, he or she can offer a much bigger project involving design of major components such as the

1 a Indeed, in Genci's report[2], it was found that 90% of the submitted programs had been plagiarized to some degree.

superblock or the entire file system API. This was the approach taken with TinyFS (Appendix A).

Lastly, modularity is an important feature that offers variation targeted by functionality. For example, an instructor may wish to concentrate on directory support, disk access modeling or caching subsystems. Offering a modular approach means specific features can be exercised and tested for without having to build the support architecture for them.

Modularity also addresses the tension between the assessment value of repeated assignments and the reality of assignments shelf-lives. It is is possible for an instructor to maintain certain modules from term to term while changing others. By doing this, he or she can create a different assignment---a new variant of the file system---each time the course is offered, changing enough components to keep it fresh while maintaining enough components to allow for comparison from term to term.

**History**

PolyFS is to a large extent a more generalized form of an existing assignment called TinyFS (see Appendix A). TinyFS was created to meet some of the same goals as PolyFS and has been offered for 3 terms already with small improvements made after each term. In general students appreciate being given an opportunity to design aspects of the file system themselves. Creating one's own free block allocation system, or superblock format requires significant understanding of file system principles. Many students have anecdotally cited this assignment as something they discussed during interviews.

While TinyFS offers some design opportunities, its overall structure is fairly static with only specified "gaps" to be filled by students. TinyFS is therefore limited in offering variety and scalability. A comparison of TinyFS and PolyFS is presented below.

**PolyFS and disk emulator overview**

At the architecture level, shown in Figure 1, PolyFS is a system that can describe a specific file system variant (*PolyFS-n*) which in turn uses an emulator or is installed on the host file system.

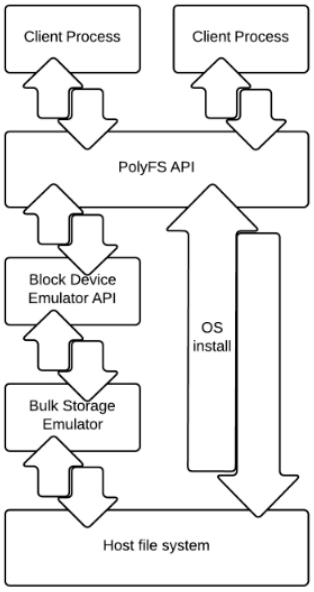


**Figure 1. PolyFS high level architecture**

One or more client programs link to the PolyFS library and interact with the file system using its published API. A header file specifying the PolyFS-n details such as the block formatting and API allows both clients and test programs to read and write to the disk. This means that almost all the functionality of *PolyFS-n* can be tested by writing different client processes. A "black box" testing approach uses the API to interact with the system and assess its features and performance.

Separate tests can also be generated based on current state of the emulated disk. Figure 2 shows a typical block file system storage space allocation. While the status of blocks remains hidden from the client programs by design, testing can be done directly on the emulated disk device to check for consistency and efficiency of use.

| | | |
|---|---|---|
| Super Block | inode Block | file extent |
| file extent | Free Block | Free Block |
| Free Block | file extent | inode Block |
| file extent | Free Block | Free Block |
| Free Block | Free Block | Free Block |

**Figure 2. Example block allocation**

## PolyFS specification and API

PolyFS was conceived with the goals of making it easier to teach OS concepts. It is purposefully underspecified to allow for filling in of gaps by the instructor or students in a design exercise. The final outcome of the exercise, however, depends on a full specification (we call PolyFS-n) and implementation.

In general two categories of specification details can be varied: emulator and PolyFS. Emulator is used to interact with a target device. Thus only block level operations should be specified. The basic API is given in four functions –openDisk(), readBlock(), writeBlock(), and closeDisk()– which will be used by students as a foundation on which to implement the PolyFS interface.

Table 1 is a list of basic PolyFS features, the bare minimum that we consider necessary for an assignment. Using these features a very basic single-directory block file system can be created with both read and write operations and tested. We recommend instructors begin with this and then move into advanced features or alternatively pick and choose which of the advanced features each team should implement.

## Advanced features

Building on the basic features, the instructors now have the opportunity to expand the assignment in one or more directions as desired. Several of the advanced features are shown in Table 2, but more are possible. We elaborate on selected advanced features.

Byte-level updates: To make the problem somewhat more tractable, we specify pfs_writeFile() to accept the entire file to be written, in form of a terminated character buffer. A student can design this function by first calculating the number of blocks necessary to store the buffer, then to create the inode block and file extents. No file pointer implementation is necessary. An advanced feature, pfs_writeByte() is capable of writing just one byte to the location indicated by the file pointer.

Disk status and defragmentation: Fragmentation is a factor in many storage systems. To familiarize the student with fragmentation issues, we extend PolyFS to include functions pfs_fragStatus() and pfs_defrag().

Directory support: Small toy file systems can be implemented with no directory support. That's the case with the base PolyFS. However, directory support including two level or tree-based directory structures can be supported through the advanced feature.

File locks: File-level synchronization support can be added in form of a pair of lock/unlock functions. Implementation of synchronization algorithms is left up to the students as an exercise.

**Table 1. PolyFS basic features**

| category | Feature / API | specification status | notes |
|---|---|---|---|
| emulator | using emulator or direct OS installation? | instructor decides this | if emulator, specify Unix file(s) to use as emulated disk |
| emulator | using storage driver or emulating disk operations? | instructor decides this | |
| storage emulator | basic device interface | Student design exercise | additional ioctl() function call to be called inside disk emulator functions |
| emulator | openDisk(), readBlock(), writeBlock(), closeDisk() | specified by instructor for target block device, or use default for Unix files | Instructor may choose to forego using an emulator, and require installation of PolyFS on the target OS |
| emulator | formatDisk(), sync() | specified by instructor or students | formatDisk() requires access to PolyFS general block spec. |
| PolyFS | general block size (default: 256 bytes) and format | specified by PolyFS / modifiable by instructor | magic number, in particular could be set by the instructor each term |
| PolyFS | inode block spec. | student design exercise | |
| PolyFS | file extent block spec. | student design exercise | |
| PolyFS | file block allocation | student design exercise | algorithm to recover all blocks of a file |
| PolyFS | free block allocation | student design exercise | algorithm to manage free blocks |
| PolyFS | superblock spec. | student design exercise | |
| PolyFS | directory inodes | student design exercise | |
| PolyFS | symbolic links | student design exercise | |
| PolyFS | consistency checks and defragmentation | student design exercise | |
| PolyFS | file naming convention | specified by PolyFS, modifiable by instructor | could be altered with directory support |
| PolyFS | basic API | specified by PolyFS | extensible by instructor |
| PolyFS | pfs_openFile(), pfs_renameFile() | student design exercise | returns a file descriptor |
| PolyFS | pfs_writeFile() | student design exercise | writes an entire terminated buffer as single PolyFS file to disk |
| PolyFS | pfs_readByte() | student design exercise | reads one byte from a pfs file at the file pointer location |
| PolyFS | pfs_seek() | student design exercise | moves the file pointer |
| PolyFS | pfs_closeFile() | student design exercise | closes file and de-allocates memory resident resources |
| PolyFS | pfs_deleteFile() | student design exercise | deletes file from disk |

Disk scheduling: Storage devices are covered in most OS courses. We have designed PolyFS with a separate module dedicated to storage systems. Rotating media physical subsystems, for example, can be modeled inside the module allowing students to implement disk scheduling algorithm covered in lecture.

**Assignment**

A typical PolyFS assignment will consist of providing all the instructor-specified information, as well as a number of features to be implemented. The basic deliverable source files are the emulator library, PolyFS library and a demo program that shows the instructor the functionalities implemented. The instructors will have multiple test client programs of their own that can be linked to the relevant libraries and make use of the system. Figure 3 shows a sample Makefile for a Unix based PolyFS assignment.

**Assignment evaluation**

We recognize that evaluation of assignments is a significant part of the teaching effort. Any assignment that is unusually difficult to evaluate for classes ranging from 20 to 200 in size would probably not be adopted by educators. We have had evaluation in mind when designing PolyFS. Automated test case evaluation has two distinct benefits. First, it eases the burden on the educator, allowing more focus on code reading, style and performance assessment. Second, it can provide a level of self-assessment to the student. Making some of the elementary test cases public, with a public and reliable evolution system will result in higher quality assignment submissions.

In the case of PolyFS, the nature of the interface greatly helps in automated evaluation. Using a test program accessing the disk through the established API in the assignment allows for the instructor scripts to easily verify many of the basic functions: reading and writing to blocks, superblock structure, file operations, time stamps and access rights can easily be tested within a single instructor test program.

Two test programs can be used within a script to evaluate file locks and concurrency features.

**Table 2. Advanced Features**

| feature | additional API | notes |
|---|---|---|
| storage crypto emulator | *store_encrtypt(), store_decrypt()* | encryption algorithm needed |
| storage compression emulator | *store_compress(), store_decompress()* | compression algorithm needed |
| byte-level update | *pfs_writeByte()* writes one byte to the current offset of an open file | basic API only supports *pfs_writeFile()* where the entire file content must be passed in buffer |
| support file creation time and modification time | *pfs_readFileInfo()* returns an array of two time stamps | involves modifying open() and writeFile() API calls |
| disk status check and defragmentation | *pfs_fragStatus()* *pfs_defrag()* moves blocks to place all free blocks together at the last portion of the disk | returns a char vector for block fragmentation status, plus an additional char with an overall status |
| directory support | *pfs_makeDir()* *pfs_deleteDir()* *pfs_copy()* works on both directories and files *pfs_listDir()* returns information for all files in the directory | create and delete directories. Open() API all will have to change to accommodate a longer string being passed in, rename() can be modified to achieve a "move" from one directory to another |
| file locks | *pfs_lock(), pfs_unlock()* | allows synchronization at file level, testable with multiple client programs |
| access rights, and mode | *pfs_chmod(), pfs_chown()* | support read-only, write-only, and basic user-level ownership, access-rights scheme to be specified in header files |
| block rotation | *pfs_engageBlockRotation()* | supports rotation of blocks to simulate all parts of the device being equally affected by degradation, used in sold-state disk technology |
| disk scheduling | *pfs_applyDiskScheduling()* | simulates rotating media buffer/cache systems and sector-level updating |
| visualization | | Write a separate process to periodically read the PolyFS disk and provide a graphical live representation of each block which can be shown to an observer as an application or web page while testing is in progress. |

```
CC = gcc
FLAGS = -Wall -g
PROG = PolyFsDemo
OBJS = PolyFsDemo.o libPolyFS.o libDisk.o

$(PROG): $(OBJS)
$(CC) $(CFLAGS) -o $(PROG) $(OBJS)

PolyFsDemo.o: PolyFsDemo.c libPolyFS.h
$(CC) $(CFLAGS) -c -o $@ $<

libPolyFS.o: libPolyFS.c libPolyFS.h libDisk.h libDisk.o
$(CC) $(CFLAGS) -c -o $@ $<

libDisk.o: libDisk.c libDisk.h
$(CC) $(CFLAGS) -c -o $@ $<
```

**Figure 3. Sample Makefile for a PolyFS assignment**

Table 3 gives a brief explanation of all the source files involved.

**Table 3. Example assignment deliverable source files**

| source file | supplied by | notes |
|---|---|---|
| libDisk.h | instructor | provides emulator API, and disk information |
| libDisk.c | student | implementation of the block device emulator |
| libPolyFS.h | instructor / student | PolyFS API functions, PolyFS block format specifications |
| libPolyFS.c | student | implementation of PolyFS API |
| PolyFSDemo.c | student | a client program interacting with a PolyFS disk |

## Pedagogical experience

A precursor to PolyFS called TinyFS has been already implemented and used for three terms as the final assignment in OS courses of the California Polytechnic State University. TinyFS features 256 byte blocks only. Student feedback indicates that the design opportunity is much appreciated. Students report that they find themselves reviewing textbook chapters on block allocation and superblock functionality in order to design an efficient file system.

## Work in progress

We are currently working to support an automated PolyFS spec generator based on instructor input. Such a generator could produce a *PolyFS-n* where n is a unique identifier reflecting assignment choices made by the instructor. Producing these unique specs would allow for automated testing tools to be developed as well. The theory is that unit-level testing routines for particular features could be automatically combined based on the particular specification.

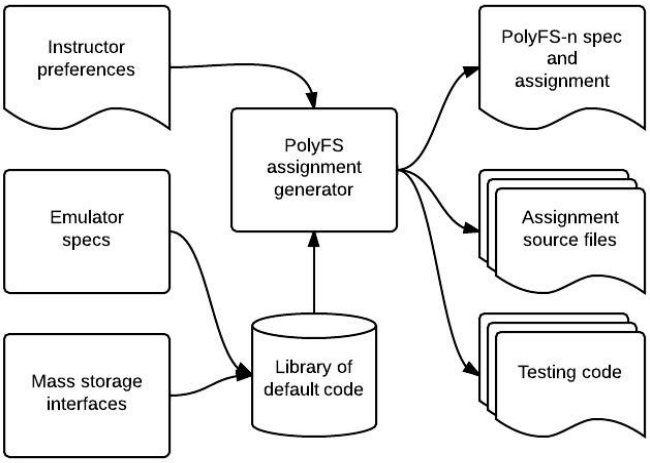


**Figure 4. PolyFS assignment generator**

## Student feedback

Our development of PolyFS takes into account our experience from TinyFS, including student feedback. For Winter term 2013, we surveyed one class that was given TinyFS as its final assignment, representing one out of 4 large programming assignments, and 10% of total grade in the operating systems class. We asked mostly for comparisons of the TinyFS assignment against the other three assignments in that same course.

29 students responded (out of 33). The majority of the students are seniors in their last four quarters of the B.S. program in Computer Science, Computer Engineering or Software Engineering at California Polytechnic State University.

We find that in general, students support and are open to design based assignments and prefer more of them. They feel that this particular assignment taught them much about file systems. They feel that, for the TinyFS assignments, concepts are relatively easy, but testing is the most difficult aspect. They also value group work highly for this assignment.

The students were also asked to respond in paragraph form to the question "What was the most difficult aspect of this assignment for you?" 29 students participated.

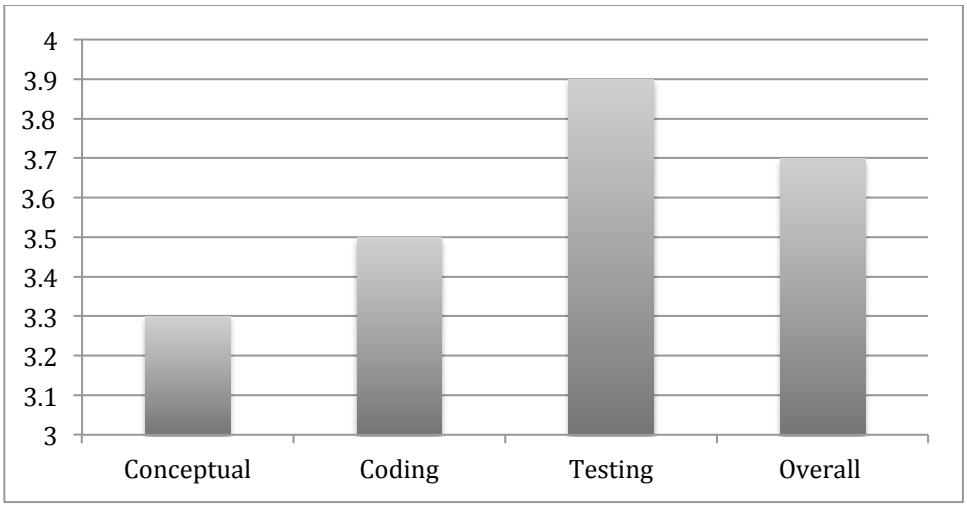


**Figure 5. Student self-reported assignment difficulty (1-5 where 5 is "very difficult").**

Although a few miscellaneous responses pointing to difficulty of working with partners and not being familiar enough with the C programming language were mentioned, the majority of the respondents cited the spec as having been vague forcing the students to have to come up with their own design and further failure scenarios. Some of the more interesting responses in this category are:

- Visualizing the system as a whole.
- The spec is very open ended so it is very difficult to figure out the best way to implement something.
- Testing has definitely been the most difficult aspect of this project, as there are many edge cases to account for. I also feel unsure of what will be tested, which makes the process more frustrating. Maybe a test driver would make the process less stressful.
- The most difficult aspect of the assignment for me was keeping track of all of the bits in the bit vector. I was not the most familiar with bit operations and so I learned a lot along the way.
- I think the hardest part of the assignment is figuring out how to implement things since it is a design assignment. There is a lot of freedom to implement ideas in different ways, so careful thought and planning must be used to avoid problems arising. A big point in this is planning ahead for the extra features, which can alter the layout of an inode block and add in more complication to the functionality.
- The most difficult aspect of the assignment was wrapping my head around what I was supposed to do. It took a long time to create the correct picture of how everything fit together. I also had a hard time distinguishing a file descriptor for a regular file and a file descriptor for the disk.

Responses to other quantitative questions on the survey are illustrated in Figure 6.

Lastly, two "Yes/No" questions were included: "Would you prefer more design?" and "Was group work important in this assignment?" The responses to both, shown in Figure 7, were overwhelmingly "Yes".

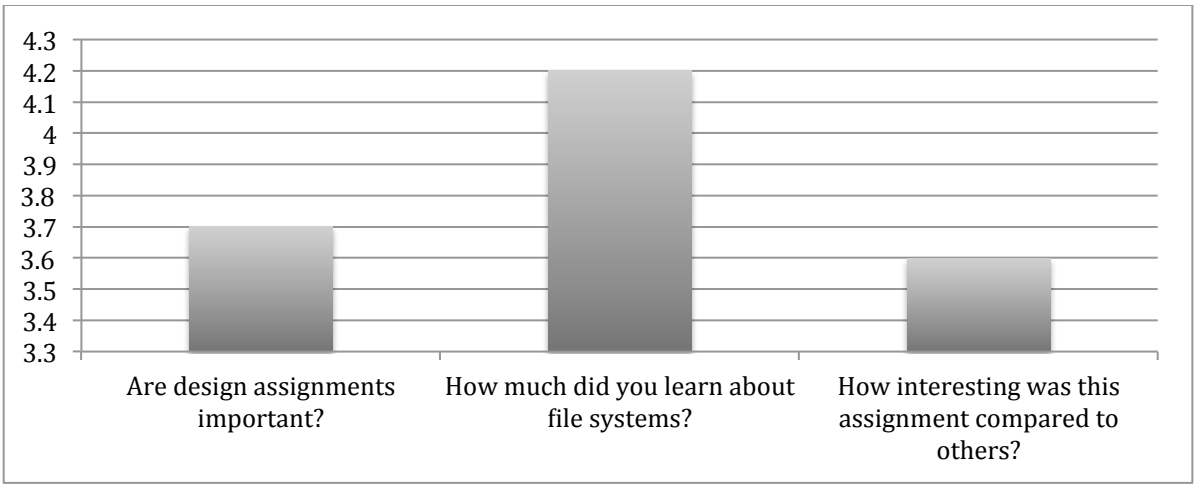


**Figure 6. Quantitative student feedback (1-5 scale).**

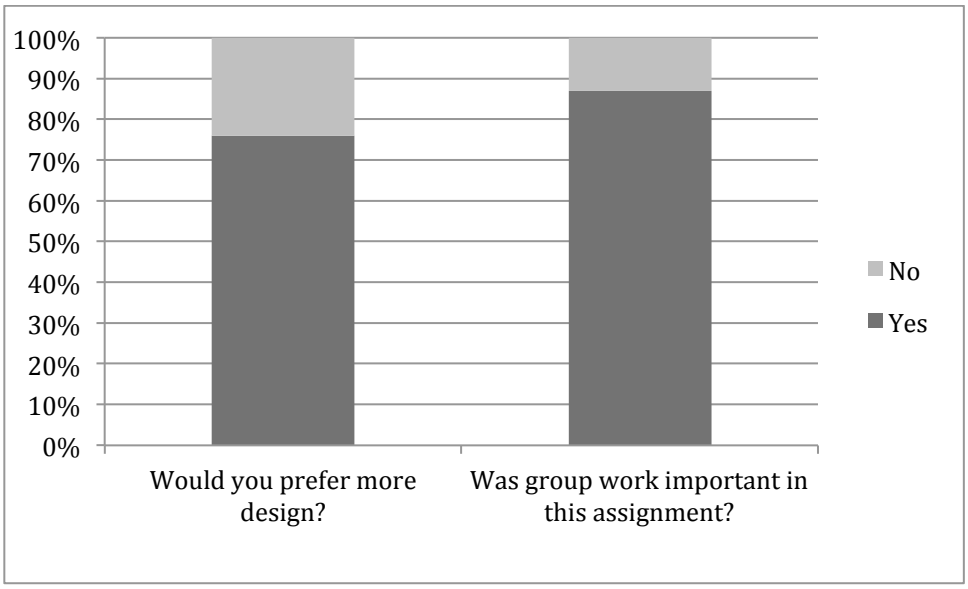


**Figure 7. Yes/No questions on student survey.**

## Conclusion

PolyFS is a file system standard capable of generating individual course assignments to test particular areas of focus for file system education. PolyFS is a work in progress, drawing from the lessons and feedback of TinyFS. We are encouraged that design and group work are valued by students, and that the general approach leads to substantial retention of information in comparison to straight implementation assignment or lecture material on file systems. In the coming terms, we will implement more and more of the overall system and use the results in real OS classes as we have been doing with the precursor, TinyFS.

Evaluation work-load can be reduced by using the same API to test PolyFS functionality. Work continues toward a suite of tools integrating specification generation with test-case generation for a particular PolyFS-n variant.

**Appendix A. TinyFS Assignment**

This is the existing TinyFS assigbnment from CPE 453 Operating Systems course, California Polytechnic State University, Winter 2013.

**Program 4** | CPE 453 | Professor Foaad Khosmood
This assignment can be done in groups of up to 3.

# TinyFS file system and disk emulator

For this assignment, you'll be implementing the TinyFS file system and emulating it on a single Unix file.

## Objective

The goal for this assignment is to gain experience with the fundamental operations of a file system. File systems are not only themselves an integral part of every operating system, but they incorporate aspects of fault tolerance, scheduling, resource management and concurrency.

## Phase I: disk emulator

The first part of the assignment is to build a disk emulator. At the lowest level of operation, an input/output control (ioctl) system call interacts directly with the device to accomplish an operation requested by the user. For disk drives (called block devices) this is usually just reading or writing a block. We will implement an emulator that will accomplish basic block operations on a regular Unix file.

The emulator is just a library of functions that interacts with a file. Three functions are necessary: openDisk(), readBlock() and writeBlock(). There are a couple of pieces of static data that are required. These can be #defined in header files. Two important ones are: block size (blockSize) in bytes, and default name of the disk file (diskName), which should be set to "TinyFSDisk".

```
/* this functions opens a regular Unix file and designates the first nBytes of it as
space for the TinyFS Disk. If nBytes > 0 and there is already a file by that name,
that file's content will be overwritten. There is no requirement to maintain integrity
of any file content beyond nBytes. That means, you can always open a new file and
write nBytes to it. To open an existing disk (assuming the filename is valid), call
openDisk() with nBytes = 0. The return value is -1 on failure or a disk# on success.
*/
int openDisk(char *filename, int nBytes);

/* readBlock() reads an entire block of blockSize size from the open disk (identified
by the disk#) and copies the result into a local buffer (must be at least of
blockSize). The bNum is a block number, which must be translated into a byte offset to
be seek()ed in the Unix file. That translation is simple: bNum=0 is the very first
byte of the file. bNum=1 is blockSize offset from the beginning of the file. bNum=X is
X*blockSize bytes into the file. On success, it returns 0. -1 or smaller is returned
if Disk is not available (hasn't been opened) or any other failures. You may define
your own error code system. */
int readBlock(int disk, int bNum, void *block);
```

```
/* writeBlock() takes a disk# and a block number and writes the content of the
argument block to that location. Just as readBlock(), it must seek() to the correct
position in the file and then write to it. On success, it returns 0. -1 or smaller is
returned if Disk is not available (hasn't been opened) or any other failures. You may
define your own error code system. */
int writeBlock(int disk, int bNum, void *block);
```

# Phase II: TinyFS file system implementation

TinyFS is a very simple file system. In fact it is under-specified to give you the freedom to implement it using many algorithms that you learned about. There are no directories or mount points, which means all the files are under a single directory. The disk blocks of TinyFS can be any of these types:

| Block name | Block code | Description | number possible | size (bytes) |
|---|---|---|---|---|
| Superblock | 1 | contains the magic number, free-list implementation and other info | 1 | 256 |
| inode | 2 | contains name of the file, file block list implementation | many | 256 |
| file extents | 3 | contains block# of the inode block | many | 256 |
| free | 4 | is ready for future writes | many | 256 |

# Block format

These bytes are defined. The rest are up to you to implement however you see fit. For example to keep track of the free blocks, you may want to use a bit vector or a forwarding link on the superblock. Same with file name information for an inode.

| Byte | first byte offset | second byte offset |
|---|---|---|
| 0 | [block type = 1,2,3,4] | 0x45 |
| 2 | [address of another block] | [empty] |
| 4 | [data] | ... |
| 6 | ... | ... |

# Block Types

## Super Block

The Super Block stores meta-information about the file system and is always block 0. The block contains three different pieces of information. First, it provides a mechanism to detect when the disk is not of the correct format. Second, it contains the block number of the root inode (for directory-based file systems). Third, it handles the list of free blocks. This can be done by having a link to the first free block in a chain of free blocks, or implementing a bit vector and storing the vector right there in the super block.

The mechanism the Super Block uses to detect a disk that isn't formatted properly is the magic number mechanism. That means a number not likely to be found in a block by accident. For us that number will be 0x45 and it is to be found exactly on the second byte of every block.

## inode

The inode block keeps tracks of meta-data for the file object. Typically ownership (user, group), file type, creation time, access time, etc. are included. For TinyFS only the name is required. The name can be just 8 alphanumeric characters an no more. Examples: “file1234”, “khosmood” or “my2ndLog”.

For inode blocks, you have to design where and how to store the file meta-data like the file name and/or time stamps and ownership. You also have to pick an offset (for example 32 bytes) at which the actual data part of the file starts.

### file extent

A file extent block contains file content data. It may be just a part of the content in which case it should contain a link to the next block of the same file content. It may contain the last of the file content (file ending) in which case, the rest of the bytes should be zero'ed out and the link byte should also be set to 0.

### free block

Free blocks are empty and available to be written to. But just as any other block, they have to have the required bytes 0,1 and 2. You may choose to use the link (byte #2) to form a chain of free blocks, otherwise you can set it to 0.

## TinyFS interface functions:

Only 6 API functions are needed to implement the TinyFS interface.

```
/* Opens a file for reading or writing. Create a dynamic resource table entry for the
file, and returns a file descriptor (integer) that can be used to reference this file
from now on. */
fileDescriptor tfs_openFile(char *name);

/* Closes the file, de-allocates all system/disk resources, and removes table entry */
int tfs_closeFile(fileDescriptor FD);

/* writes an entire buffer, representing the entire file content, to a file. Sets the
file pointer to 0 (the very beginning) when done. Returns success/error codes.
(content terminated by null "\0") */
int tfs_writeFile(fileDescriptor FD,char *buffer);

/* deletes a file and marks its blocks available on disk. */
int tfs_deleteFile(fileDescriptor FD);

/* reads one byte from the file and copies it to buffer, uses the current file pointer
location, and increments it by one after. If the file pointer is already at the end of
the file (where the terminating NULL character is) then tfs_readByte() should return
an error (- value) and not increment the file pointer. */
int tfs_readByte(fileDescriptor FD, char *buffer);

/* change the file pointer location to offset (absolute) */
int tfs_seek(fileDescriptor FD, int offset);
```

## Assignment

- Implement the 6 interface functions above (80%)
- Add two additional areas of functionality from the list (a-d) below. You are free to implement them any way you wish with any number of parameters / return type. (20%)
    - a Fragmentation info and defragmentation
        - implement tfs_displayFragments() /* this function allows the user to see a map of all blocks with the non-free blocks clearly designated. You can return this as a linked list or a bit map which you can use to display the map with */

        - implement tfs_defrag() /* moves blocks such that all free blocks are contiguous at the end of the disk. This should be verifiable with the tfs_displayFraments() function */
    b. Directory and renaming
        - tfs_rename() /* renames a file. New name should be passed in. */
        - tfs_dir() /* lists all the files on the disk */
    c. Read-only and writeByte support
        - implement the ability to designate a file as “read only”. By default all files are “read write” (RW).
        - tfs_makeRO(char *name) /* makes the file read only. If a file is RO, all tfs_write() and tfs_deleteFile() functions that try to use it fail. */
        - tfs_makeRW(char *name) /* makes the file read-write */
        - tfs_writeByte(fileDescriptor FD, int offset, unsigned int data), a function that can write one byte to an exact position inside the file.
            - tfs_writeByte(fileDescriptor FD, unsigned int data) is also acceptable. (uses current file pointer instead of offset).
    d. Time stamps
        - implement creation time stamps for each file to be stored in the inode block
        - tfs_readFileInfo(fileDescriptor FD) /* returns the file’s creation time */
- Write a demo program that includes your TinyFS interface to demonstrate the basic functionality of the 6 required functions and your chosen additional functionality. You can display informative messages to the screen for the user to see how you demonstrate these.

## Deliverables

- as usual, submit a tar.gz archive via polylearn with the following:
    - all source files: .c, .cpp, .h
        - You must have at least three separate source files
            1. emulator file (libDisk)
            2. tinyFS interface file (libTinyFS). This file will access libDisk for disk emulator functionality.
            3. demoTfs driver file that contains a main(), and includes libTinyFS headers (but not libDisk).
    - a makefile (called Makefile) that compiles all the libraries and makes the following executable:
        - demoTfs
    - a README with:
        - Names of all partners
        - An explanation of how well your TinyFS implementation works
        - An explanation of which additional functionality areas you have chosen and how you have shown that it works.

## Appendix B. A TinyFS demo program

This is a TinyFS demo program listing and a subsequent look at the emulated disk.

```
foaad@unix3:~/453/workspace $ ./demoTfs
No TinyFSDisk found, formatting...
Checking input in file abcd1234:
The quick brown fox jumped over the lazy dog

Reading after seek exepct 5:  t

Testing tfs_readFileInfo output:
File created on: Wed Jan 30 11:50:11 2013

Closed tfs_closeFile then called tfs_write, this should fail:
failed to write file

Closed tfs_closeFile then called tfs_readByte, this should fail:
failed to read

Reading input from abcd1234, file descriptor fd1, output:
some sentences some sentences some sentences some sentences

Reading after seek exepct 5:  5

Reading input from abcd1234, file descriptor fd, output:
6789

List all files in directory with tfs_dir:
new3
new2
newName

Reading input from abcd1234, file descriptor fd1:
success

Attempt to write to deleted file should fail:
it is a failure


foaad@unix3:~/453/workspace $ hexdump -C TinyFSDisk
00000000  01 45 09 00 00 00 00 00  00 00 00 00 00 00 00 00  |.E..............|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000100  02 45 00 00 66 64 33 00  6e 65 77 32 17 79 09 51  |.E..fd3.new2.y.Q|
00000110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000200  02 45 00 00 61 62 63 64  31 32 33 34 17 79 09 51  |.E..abcd1234.y.Q|
00000210  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000300  04 45 07 00 00 00 00 00  00 00 00 00 00 00 00 00  |.E..............|
00000310  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000400  04 45 0a 00 00 00 00 00  00 00 00 00 00 00 00 00  |.E..............|
00000410  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000500  02 45 00 00 6e 65 77 33  00 6e 65 77 0d 79 09 51  |.E..new3.new.y.Q|
00000510  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000600  02 45 00 00 6e 65 77 32  00 6e 65 77 0d 79 09 51  |.E..new2.new.y.Q|
00000610  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000700  04 45 04 00 00 00 00 00  00 00 00 00 00 00 00 00  |.E..............|
00000710  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000800  02 45 00 00 66 64 32 00  66 64 33 00 17 79 09 51  |.E..fd2.fd3..y.Q|
00000810  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000900  04 45 03 00 00 00 00 00  00 00 00 00 00 00 00 00  |.E..............|
00000910  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

**Bibliography**

[1] Miller, E. L., "The DLX Operating System (DLXOS)" http://users.soe.ucsc.edu/~elm/Software/Dlxos/dlxos.shtml

[2] Genci, J. Knowledge assessment — practical example in testing. In *Technological Developments in Education and Automation,* M. Iskander, V. Kapila, and M. A. Karim, Eds. Springer Netherlands, 2010, pp. 409–412.

[3] Holliday, M.A. "Teaching Computer Systems through Common Principles", 41st ASEE/IEEE Frontiers In Education Conference, October 2011, Rapid City, SD, pp. S2G-1 to S2G-6.

[4] Tanenbaum, Andrew S; Albert S. Woodhull , *Operating Systems: Design and Implementation* (3rd ed.). Prentice Hall, 2006.

[5] Silberschatz, Galvin, and Gagne, *Operating System Concepts*, 8th Edition. Wiley*, 2009.*

[6] Joseph, Anthony and Stoica, Ion. Course "CS 160: Operating Systems and Systems Programming", University of California Berkeley, Spring 2012. http://inst.eecs.berkeley.edu/~cs162/sp12/

[7] Ousterhout, John. Course "CS 140: Operating Systems", Stanford University, Winter 2013. http://www.stanford.edu/~ouster/cgi-bin/cs140-winter13/