POLYFS VISUALISER

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Paul Fallon

June 2016

© 2016
Paul Fallon
ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE: PolyFS Visualiser

AUTHOR: Paul Fallon

DATE SUBMITTED: June 2016

COMMITTEE CHAIR: Foaad Khosmood, Ph.D.

Professor of Computer Science

COMMITTEE MEMBER: Zachary Peterson, Ph.D.

Associate Professor of Computer Science

COMMITTEE MEMBER: Phil Nico, Ph.D.

Professor of Computer Science

ABSTRACT

PolyFS Visualiser

Paul Fallon

One of the most important operating system topics, file systems, control how we store and access data and form a key point in a computer scientists understanding of the underlying mechanisms of a computer. However, file systems, with their abstract concepts and lack of concrete learning aids, is a confusing subjects for students. Historically at Cal Poly, the CPE 453 Introduction to Operating Systems has been on of the most failed classes in the computing majors [13], leading to the need for better teaching and learning tools. Tools allowing students to gain concrete examples of abstract concepts could be used to better prepare students for industry.

The PolyFS Visualizer is a block level file system visualization service built for the PolyFS and TinyFS file systems design specifications currently used by some of professors teaching CPE 453. The service allows students to easily view the blocks of their file system and see metadata, the blocks binary content and the interlinked structure. Students can either compile their file system code with a provided block emulation library to build their disk on a remote server and make use of a visualization website or place the file mounted as their file system directly into the visualization service to view it locally. This allows students to easily view, debug and explore their implementation of a file system to understand how different design decisions affect its operation.

The implementation includes three main components: a disk emulation library in C for compilation with students code, a node JS back-end to handle students file systems and block operations and a read only visualization service. We have conducted two surveys of students in order to determine the usefulness of the PolyFS Visualizer. Students responded that the use of the PolyFS visualizer helps with

the PolyFS file system design project and has several ideas for future features and expansions.

ACKNOWLEDGMENTS

Thanks to:

- My parents for supporting me throughout my college career.
- Professor Foaad for his guidance and good advice.

TABLE OF CONTENTS

Page

LIST OF TABLES x

LIST OF FIGURES xi

CHAPTER

1 Introduction 1

2 Related Works 3

2.1 DECAFS 3

2.2 Tree-Maps and StepTree 4

2.3 SOsim 6

2.4 PortOS 7

2.5 WADEin 8

3 Background 10

3.1 The Class 10

3.2 PolyFS 10

3.2.1 PolyFS History 11

3.2.2 Poly FS Phase I: Disk Emulator 11

3.2.3 Poly FS Phase II: File System 12

3.3 Difficulties of Teaching Operating Systems 14

4 Design 15

4.1 Goals 15

4.2 Requirements 15

4.2.1 Accessibility 16

4.2.2 Server Deployment and Maintenance 17

4.2.3 Visualizer Deployment 18

4.2.4 Disk Emulator 19

4.2.5 Visualizer Elements 21

4.2.6 Visualizer Configurability 22

5 Implementation 24

5.1 Overview 24

5.2 Server 24
5.2.1 Edit Requests 27
5.2.2 Database Architecture 31
5.2.3 Module Overview 31
5.3 Visualizer 33
5.3.1 Config 36
5.3.2 Disk Metadata 38
5.3.3 Modules 39
5.4 Client Disk Library 43
5.4.1 Modules 45
6 Software Validation 47
6.1 Test Sets 47
6.1.1 External Tests 48
6.1.2 Internal Tests 50
6.1.3 Server Tests 52
6.1.4 Disk Tests 54
6.1.5 Meta Tests 55
6.1.6 Config Tests 56
6.2 Deployment 57
6.2.1 Server Operating System 57
6.2.2 Resolution 58
6.2.3 Browser 59
7 Experiment 61
7.1 Hypothesis 61
7.2 Experimental Design 61
7.3 Concerns with Experimental Design 62
7.4 Survey One Results 63
7.5 Survey Two Results 68
7.6 Survey Conclusions 73
8 Future Work 76
9 Conclusion 77
BIBLIOGRAPHY 78

APPENDICES

A TinyFS Specification . . . 81
A.1 TinyFS and disk emulator . . . 81
A.1.1 Objective . . . 82
A.2 Phase I: Disk Emulator . . . 82
A.2.1 LibDisk Interface Functions . . . 82
A.3 Phase II: TinyFS file system implementation . . . 84
A.3.1 Block Types . . . 84
A.3.2 Block format . . . 86
A.3.3 TinyFS interface functions: . . . 86
A.3.4 Error Codes . . . 88
A.4 Assignment & Additional Features . . . 89
A.5 Deliverables . . . 92
B Before Survey . . . 94
B.1 TinyFS File System and Emulator Project . . . 94
B.1.1 Do you agree with the above statement? . . . 96
B.1.2 Your name . . . 96
B.2 Short Answer Questions . . . 96
B.3 Grid Questions . . . 96
C After Survey . . . 98
C.1 TinyFS File System and Emulator Project . . . 98
C.1.1 Do you agree with the above statement? . . . 100
C.1.2 Your name . . . 100
C.2 Short Answer Questions . . . 100
C.3 Grid Questions . . . 100
C.4 Visualizer Responses . . . 101
D Sample Code . . . 103
D.1 Block Modal Links . . . 103
D.2 Block Metadata Per Type Handlers . . . 105
D.3 Proper Name-spacing . . . 107
E Test code . . . 108
F Test code . . . 117

LIST OF TABLES

Table Page

6.1 The test sets used to verify the PolyFS Visualizer 48
6.2 Table of the external test suite and results 50
6.3 Table of the internal test suite and results 52
6.4 Table of the server test suite and results 53
6.5 Table of the disk test suite and results 54
6.6 Table of the meta test suite and results 55
6.7 Table of the config test suite and results 57
6.8 Table of the resolutions tested 59
6.9 Table of the browsers tested 60

LIST OF FIGURES

Figure		Page
2.1	Tree-Maps used to visualize a file system	4
2.2	A StepTree file system visualization	5
2.3	The OSsim Memory Management module display	7
2.4	The GUI for WADEin	8
5.1	An overview of the PolyFS Visualizer architecture	25
5.2	An overview of the server architecture	27
5.3	Open() flow for num_blocks = 0	29
5.4	Open() flow for num_blocks != 0	29
5.5	Read() flow	30
5.6	Write() flow	30
5.7	An overview of the visualizer architecture	34
5.8	The visualizer before a disk is loaded	35
5.9	The visualizer after disk "demo2" has been loaded	36
5.10	The visualizer with mouse over the first block	37
5.11	The visualizer with block modal shown	38
5.12	The visualizer with disk loading modal shown	39
5.13	An example configuration file	40
5.14	An example metadata file	40
5.15	An overview of the disk emulator architecture	44
5.16	Open request to the server	45
7.1	Survey 1: Aggregation of "Name some of the challenges you faced with this project"	64
7.2	Survey 1: Aggregation of "How did you verify your file system was storing data correctly?"	64
7.3	Survey 1: Aggregation of "What tools did you use to help you with this project?"	65
7.4	Survey 1: Aggregation of "How could this project be improved?"	66
7.5	Survey 1: agree/disagree grid questions	67

7.6 Survey 2: Aggregation of “Name some of the challenges you faced with this project” . . . 69
7.7 Survey 2: Aggregation of “How did you verify your file system was storing data correctly?” . . . 70
7.8 Survey 2: Aggregation of “How could this project be improved?” . 71
7.9 Survey 2: agree/disagree grid questions . . . 72
7.10 Survey 2: Aggregation of all comments on the PolyFS Visualizer . . 73

F.1 Disk Test 1: perfect disk . . . 117
F.2 Disk Test 2: empty disk . . . 117
F.3 Disk Test 3: disk with last block incomplete . . . 118
F.4 Disk Test 4: One incomplete block . . . 118
F.5 Disk Test 2: disk missing . . . 118
F.6 Disk Test 6: 100 blocks x 256 bytes . . . 119
F.7 Disk Test 7: 100 blocks x 1 byte . . . 119
F.8 Disk Test 8: 1 block x 256 bytes . . . 119
F.9 Disk Test 9: 100 blocks x 128 bytes . . . 120
F.10 Disk Test 10: 100 blocks x 512 bytes . . . 120

Chapter 1

Introduction

The PolyFS Visualizer is a visualizer for student made file systems for CPE453: Introduction to Operating Systems. The last project of CPE453 is to build a file system following the sparse PolyFS specification. The PolyFS specification purposefully leaves out most internal design details in order to allow students to create their own designs, making the project very much focused on conceptual file system architecture. As a result, many students struggle to implement their designs, either because they can not translate their design into code or because their design does not completely delineate the file system or programming process. In order to help students implement and understand their designs at a deeper level, the PolyFS Visualizer will allow them to easily examine their disk data at a binary level.

Most previous work in the area of education of operating systems has focused on simulation or portability of learning platforms. Works like SOsim and PortOS exemplify this. However, Cal Poly wishes to focus more on larger design based projects where students are asked to both design and implement a program. This gives need to tools students can use to help them understand how their design and implementation connect and how their implementation reflects their design.

Research in education on operating systems has focused on the canned simulation or, at best, simulation of user inputs. Cal Poly hopes to focus on a more hands on method of allowing students to design and build their own file system while having the resources to visually represent the disk. To close the gap between design and understanding and implementation, which leaves many students confused, we developed PolyFS.

The contribution of this paper is threefold:

1. Design and build a visualization service for PolyFS that views the disk at the block and binary levels, allowing student's to view their disks easily and completely.

2. Design and build a server and client that transparently replaces the student's disk library and builds their disk on a remote server in a format for visualization

3. Test file system visualization in a classroom environment and analyze the effectiveness of visualization tools as they are applied to file system understanding and design.

This thesis describes how these three contributions were achieved.

Background information and related works are chapters 1 and chapter 2 respectively. The goals, requirements and design of the system are explained in chapter 3. The implementation is explained in detail in chapter 4. Validation is in two sections with software validation in chapter 5 and the student experiment in chapter 6. Finally, future work and the conclusion are related in chapter 7 and chapter 8 respectively.

Chapter 2

Related Works

Educational tools have been an area of study and development for some time. The two areas of programming platforms and visualization tools, both of which are incorporated into the PolyFS Visualizer have several prior projects of interest. These project are outlined below with a short critique on the design, implementation and verification of each. Particular concern is placed on the verification due to the direct application of those concepts to this paper.

2.1 DECAFS

DECAFS is a distributed file system (DFS) intended to help teach students how distributed file systems works. Developed by Cal Poly student Halli Meth and advised by Professor Chris Lupo, DECAFS provides a modular, expansive DFS and a set of labs for students to work with.

DFSs tend to be much larger and more complex than typical quarter long projects making it much more difficult for classes to follow Cal Poly's learn by doing ideology. Therefore, DECAFS is intended to be fully working and with a modular design allowing for students to replace specific sections of the DFS. This allows for in depth labs, projects and activities with a real DFS to be accomplished within a quarter class. This project was ambitious with a feature set very close to full industrial DFSs.

DECAFS was tested in grad level distributed programming courses, relying on the knowledge of graduate students to aid in fixing any problems found in either the DFS or the assignments. Unfortunately, it was found the DECAFS was not fully read for deployment. The project was very ambitious and sacrificed some software validation

for additional features. Thus, the graduate class found DECAFS had bugs and other problems making completing the associated labs difficult.

This project has several similarities to the PolyFS Visualizer. Both are intended to allows classes to better follow Cal Poly's Learn by Doing philosophy. Therefore, as a result of DECAFS outcome and usage, it was decided to sacrifice an expansive features set for more rigorous software validation in the building the PolyFS Visualizer. Additionally, user testing was conducted during the course of the project rather than left for future work. [17]

2.2 Tree-Maps and StepTree

StepTree is a visualization tool for hierarchical data structures with an easy application for file systems. The paper uses file systems as its primary example, including using it for an analysis of the effectiveness of its tool.

StepTree was developed in several parts, each building off of the last and providing further features or user studies. In 1991 Brian Johnson and Ben Shneiderman of University of Maryland developed a new technique for the visualizing of hierarchical data structures call Tree-Maps. The technique was based off of Venn diagrams but reformatted to make use of 100% of the provided space.

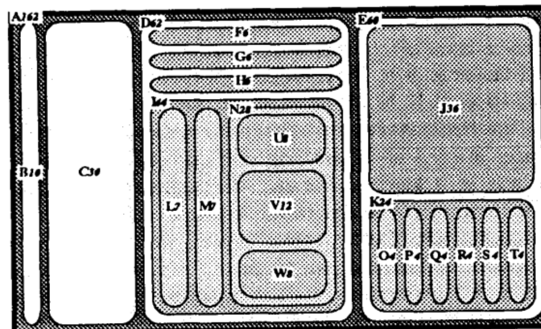


Figure 2.1: Tree-Maps used to visualize a file system

Tree-Maps was extended to create StepTree in 2004 by Thomas Bladh, David A. Carr, and Jeremiah Scholl from the Lule University of Technology. Tree-Maps was extended to 3 dimensions and a user study was performed on the effectiveness of 3D Tree Maps on communicating file system data. The user study included 20 participants (one had to be replaced due to color blindness) and focused on a comparison between StepTree and Tree-Maps. The user study showed the simple data retrieval tasks could be done at least as fast with StepTree despite the 3D user interface.

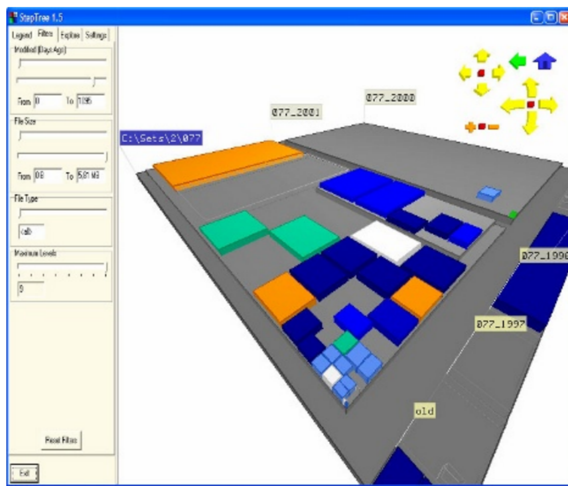


Figure 2.2: A StepTree file system visualization

A final paper was done as an extension of StepTree, again by Thomas Bladh, David A. Carr and Matja Kljun to analyze the affects of adding animations to the navigation through the StepTree. In this paper selection of a sub-hierarchy was animated to try and aid users in understanding the affects of the actions and ease the strain of navigating through the hierarchy. They found that in general use the addition of animation did make it easier for the user to navigate through the hierarchy. However, the general improvement came at the price of much greater difficulty recovering from mistakes. In this case users kept attempting to make use of the shortcuts introduced with the animations rather than going back to a more measured step by step approach.

The core similarity between StepTree, Tree-Maps and the PolyFS Visualizer is the

use of visualization to easily show users file system information. However, StepTree and Tree-Map intend to show directory structure rather than data structure. This difference completely changes the use cases of the system and makes it very difficult to adapt StepTree or Tree-Maps to the problem of visualizing binary disk data for CPE 453 students. [12, 8, 7]

2.3 SOsim

SOsim is a program meant to provide a simulated operating system environment in order to visualize operating system functionality and concepts. The program includes visualization for virtual memory, processes and scheduling algorithms.

To illustrate one of these systems, the memory management system is described by the paper: "main memory is divided in 100 page frames and each process can allocate up to five frames. Processes have their own page table that can be observed while changes take place."

Two things stand out about SOsim. Firstly, it has a global perspective to operating systems and visualizes the system as a whole. Users select what they want to view at any one time but a good portion of the topics in an operating systems course are included. Secondly, the verification of the system is much better documented. A class of 30 students was given a project using SOsim and upon completion of the project filled out a seven question survey (questions documented in the paper). The exact results are in the paper but they showed an agreement that the system helped the students understand operating systems concepts. [14, 15]

Figure 2.3: The OSsim Memory Management module display

2.4 PortOS

PortOS has the goal of providing a portable, standardized operating system for student populations with a large percentage of remote students. Designed off of 3 main principles “provide a realistic but sanitized environment”, “support common platforms” and “support familiar development environments”.

PortOS is used to provide a common operating system for Internet classes and thus focuses on provided portability above all other concerns. Additionally, the features are geared specifically toward students, meaning features are planned more as examples and conceptual explanations rather than rigorous use. [4]

2.5 WADEin

With the goal of providing an adaptable visualization of expression evaluation in the c programming language, WADEin has a GUI interface that allows the input of an expression and shows a step by step processing of the expression. The current expression at the current phase of processing is displayed in a "shrinking copy" on the GUI as seen in 2.4.

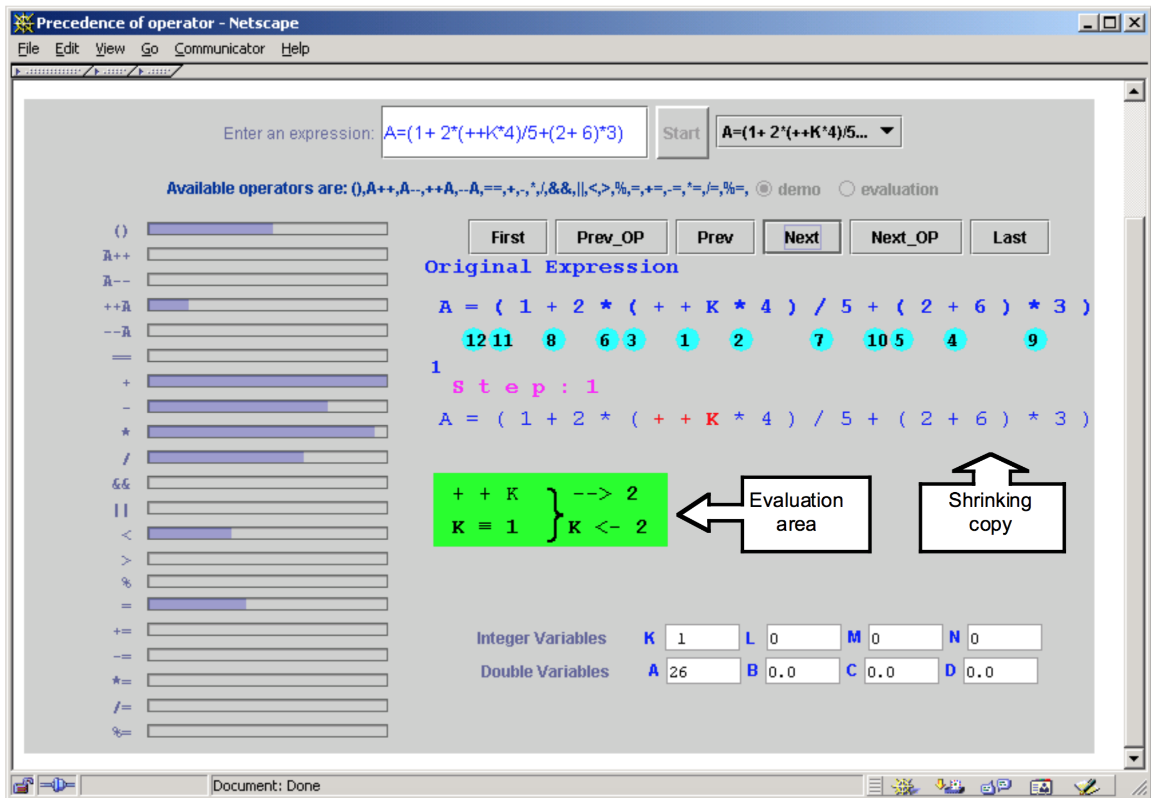


Figure 2.4: The GUI for WADEin

Thought WADEin is geared toward a different topic of computer science (beginning C programmers rather than introduction to operating systems), the concepts used in teaching the students are the same and the project resembles this one in terms of application and methodology of visualization.

The primary contributions of WADEin is the adaptable system meant to show the

correct level of detail to students of differing skill levels. It has 5 skill levels, the first of which shows each step in slow motion with animations and the last processes the expression in one step. The skill of the student is monitored by what they have already viewed through the program as well as through self-evaluation interfaces provided to the student during the animations.

The paper reports positive feedback from students stating “Many students considered WADEin as the most useful tool among all tools that are currently connected to the KnowledgeTree portal”, which is a grouping of server based educational tools meant to aide professors in teaching programming classes. However, the paper only notes pilot testing in a single practical c course adding that “Currently we are running a larger and more formal evaluation of the system in an introductory programming class”. From this comment we can conclude that they are still drawing their conclusions off of small groups of students and their lack of documented experimental design or empirical analysis leads me to believe their evaluation was informal at best. [9]

Chapter 3

Background

3.1 The Class

The PolyFS Visualizer is aimed at helping students in introductory operating systems classes. This specific tool is a proof of concept for file system visualization geared toward classroom learning and will be developed for students taking CPE 453 Introduction to Operating Systems at California Polytechnic State University at San Luis Obispo. These students are typically Junior or Senior level and the course description of CPE 453 according to the Cal Poly Computer Science catalog is:

> Introduction to sequential and multiprogramming operating systems; kernel calls, interrupt service mechanisms, scheduling, files and protection mechanisms, conventional machine attributes that apply to operating system implementation, virtual memory management, and I/O control systems.

Students in CPE 453 are expected to have completed CSC/CPE 357, and CSC/CPE 225 or CPE/EE 229 or CPE/EE 233. CPE 357 is a course in C systems programming covering advanced C including memory management. CPE 225, EE 229 and EE 233 are courses that cover low level organization of computing systems and include topics such as assembly languages, basic computer architecture, instruction set architectures. Students are expected to be comfortable with these topics at the start of CPE 453.

3.2 PolyFS

PolyFS is a two phase project. The first phase is a disk emulator mounted onto a single Unix file system in order to abstract out hardware interaction. The second phase

builds a complete, if basic, file system on top the disk emulator. The PolyFS Visualizer is designed to be integrated with PolyFS and its predecessor TinyFS. The visualizer will provide a framework for the second part of the assignment and allow students to make use of several additional features without removing any of the current learning objectives or assignment deliverables. PolyFS and TinyFS are specifications for a small file system with a basic feature set and milestones that can be implemented by the students within a 3 week period. While the entire specification of both projects are included in the appendices, important features and considerations for PolyFS are highlighted here. Both phases have a defined API that must be implemented by the students

3.2.1 PolyFS History

TinyFS was created by Foaad Khosmood in 2012 in order to provide a more in depth assignment for students in CPE 453. TinyFS provides students the opportunity to design portions of their file system implementation which requires significant understanding of the concepts. TinyFS has been offered as an assignment for several years with incremental improvements after each term.

PolyFS is a more dynamic successor, introduced originally in 2013 and will soon replace TinyFS. PolyFS is intended to accomplish the same goals as TinyFS but provide students more opportunities to implement their own designs [13].

3.2.2 Poly FS Phase I: Disk Emulator

Phase I is a disk emulator implemented as a single Unix file and supporting basic block level operations. All operations for Phase I will be based around blocks, with all data buffers being of BLOCK_SIZE size and undefined behavior for any input buffers not of the correct size. Phase I has 3 functions to implement at the disk

emulator level, for a more in depth discussion of these functions view the complete specification in the appendix:

```
1    int openDisk(char *filename, int nBytes);
2    int readBlock(int disk, int bNum, void *block);
3    int writeBlock(int disk, int bNum, void *block);
```

3.2.3 Poly FS Phase II: File System

PolyFS Phase II is a file system build on top of the disk emulator. The specifications for the part of the assignment are purposefully vague allowing for students to apply the knowledge they learned in the class including a wide variety of file system design methodologies. The specifications have three main goals: provide the students a direction and place to start, ensure the projects are uniform enough to be graded fairly, and give the professor and TAs enough information on the project to be able to aid students their specific implementations. The specifications require the implementation of several basic file system functions, listed below. Again, for more specifics regarding each function view the full specification in the appendices.

```
1    int tfs_mkfs(char *filename, int nBytes);
2    int tfs_mount(char *filename);
3    int tfs_unmount(void);
4    fileDescriptor tfs_openFile(char *name);
5    int tfs_closeFile(fileDescriptor FD);
6    int tfs_writeFile(fileDescriptor FD, char *buffer, int size);
7    int tfs_deleteFile(fileDescriptor FD);
8    int tfs_readByte(fileDescriptor FD, char *buffer);
9    int tfs_seek(fileDescriptor FD, int offset);
```

PolyFS does not support hierarchical files and having no directories and a completely flat structure. Thus, the file system can have four types of blocks:

Block name	Block code	Description	Number Possible	Size (bytes)
superblock	1	must contain the magic number, pointer to root inode, and the free block-list implementation	1	256
inode	2	must contain the name of the file, the file size and a data block indexing implementation	many	256
file extent	3	contains block# of the inode block	many	256
free	4	is ready for future writes	many	256

Each block is required to have a couple bytes at the beginning of the block in a specific format. The format is meant to provide a structure to the blocks and includes a preset location for linking blocks (free lists, file linked lists, etc), a magic number to easily find corrupt or empty blocks and the block type. The exact layout is shown below:

Byte	first byte offset	second byte offset
0	[block type = 1,2,3,4,....]	0x44
2	[address of another block]	[empty]
4	[data starts]	...
6	...	...

It should be noted that these specifications do not require any design decisions in

terms on how to track free blocks, group data blocks into files, or manage data on the inodes or superblock. All these design decisions are up to the student.

3.3 Difficulties of Teaching Operating Systems

Operating Systems is one of the harder classes for professors to effectively teach and for students to understand. "Unlike other disciplines in the computer area, OS is a subject that does not exhibit a linear structure that allows the lecturer to progress through the topics in a sequential order."[14]. The lack of a linear progression means students may have a hard time connecting individual topics into a single framework of understanding. Prior works have noted that "The problem is due to both the teaching model and the lack of appropriate tools capable of translating the theory being presented into a more practical reality. And without a practical vision the student tends to lose touch and just 'float' around the introduced concepts and mechanisms without gaining a realistic view of what is really going on."[14]. Thus, there is a well established need for more structure and teaching tools for introductory Operating Systems classes.

Chapter 4

Design

This section discusses the general goals of the projects and then formalize these goals into a set of requirements. The requirements are not a mix of technical specifications and usability concerns, focusing on how users will interact with the product rather than how it will be built.

4.1 Goals

First and foremost, the visualizer is meant to help students understand file systems. Logistically this means students must have easy access to an intuitive, easy-to-use interface with the ability to test their code. Additionally the visualizer must not draw conclusions for the students but merely provides students with the tools to draw their own conclusions. For example, the visualizer should not directly identify errors, but merely display the content that was found. Students will be able to identify errors from the content. Furthermore, the visualizer must not become a crutch for students to complete assignments easily and quickly but rather a way for them to expand their knowledge and build a deeper understanding of the material through concrete and visual examination of their file system.

4.2 Requirements

In this section, the requirements for the PolyFS Visualizer are discussed and laid out in a more formal format. Since the goals of the system are so generic, we must distill out a more stringent set of requirements for what is displayed and how as well as the ability of the students to access and use the system. The requirements are discussed

for each of six elements: accessibility, server deployment and maintenance, visualizer deployment, disk emulator, visualizer elements and visualizer configurability.

Accessibility is the ability of the students to access the system as well as the equipment necessary for them to do so. Server maintenance and deployment is the ease of setting up new systems as well as maintaining old ones. Visualizer deployment is the ease of setting up the visualizer. Note that this section is necessary because the visualizer can be deployed separately from the server. Disk emulator is the part of the server as well as the associated client side library that handles disk access and manipulation. Visualizer elements is concerned with what is actually displayed in the visualizer while visualizer configurability is the ease of configuring the visualizer elements to fit different disk formats and visual needs.

4.2.1 Accessibility

The visualizer must be easily accessible for students. In order to help students it was decided that the visualizer must be accessible in two formats, a server framework with associated website and a read only website front end that can be deployed locally quickly and easily. This allows students more options on how they want to use the system as well as ensuring they have the ability to make use of it no matter where, when or how they work.

The formalized requirements are noted below:

- Visualizer can be deployed on any web server (such as apache) without any custom back-end
- Visualizer must be a read only web application
- Visualizer must work on laptops with a resolution greater than 1024x640 pixels

- Visualizer must work on the three main browsers use by students browsers (Chrome, Firefox, Safari)
- Visualizer must not require the installation of any plug-ins
- When the server is unavailable (possibly due to maintenance or network outage) another means to access the visualizer must be possible

Several things are noted to be outside the specification for this project:

- Visualizer is not required to work on mobile devices

4.2.2 Server Deployment and Maintenance

The server must be easy to deploy and maintain since there will be no dedicated resources for the long term maintenance of the system. It is assumed that professors of CPE 453 will be deploying and maintaining the visualization service and associated infrastructure after research and development has been concluded. It is therefore required that maintenance take very limited effort during the quarter but some potentially with some required setup or cleanup between quarters. Additionally, it is assumed that the professors using this tool have a solid understanding of how to deploy and manage websites but no knowledge of this specific system. Deployment must still be easy to accomplish for the professors and all dependencies and requirements must be clearly established.

The formalized requirements are noted below:

- Server must not require maintenance during school quarters
- Server must provide a script to manage deletion of disks by the server admin
- Server must run on a standard Ubuntu Linux box

- Server must support up to 100 students over the course of a quarter
- Server must not generate unreasonable amounts of network traffic
- Complete documentation of all server requirements and dependencies must exist

Several things are noted to be outside the specification for this project:

- Server is not required to include automated student content management such as deletion of old disks
- It is not required to have a scripted deployment

4.2.3 Visualizer Deployment

The deployment of the visualizer is concerned with two main goals: the accessibility of the site for the students and the ease of deploying the visualizer with the server.

For the students, the visualizer must be deployable as a separate component easily and quickly so that students can set up their own local visualizers. The visualizer must also be compatible with Cal Poly's CSC student websites, which are hosted through the students CSC Unix server accounts and are read-only. The visualizer must not have any external dependencies in order to focus on an easy installation.

For deployment with the server, the visualizer must be easily associated with the server back-end. This means the server back-end must send data to the visualizer in the exact same format as the students deploying the visualizer locally.

The formalized requirements are noted below:

- As noted in the accessibility section, the visualizer can be deployed on any web server (such as apache) without any custom back-end

- As noted in the accessibility section, the visualizer must be a read only web application
- Visualizer must be compatible with the student websites hosted on cal poly CSC servers
- Visualizer must be able to easily disassociate from the back-end server
- Visualizer must accept only file inputs both as a separate entity and when deployed with the server.
- A space within the Visualizer must be provided for students to place Disk files along with associated meta and settings files

Several things are noted to be outside the specification for this project:

- Visualizer is not required to update when the disk is updated
- Visualizer is not required to provide front end interface for uploading disks
- Visualizer is not required to provide front end interface for changing configurations
- Visualizer is not required to provide security

4.2.4 Disk Emulator

The Disk emulator will replace the students version after they complete the first part of PolyFS. Therefore, the disk emulator must have the exact same user facing API as the PolyFS specification as well as the exact same operation:

```
1    int openDisk(char *filename, int nBytes);
2    int readBlock(int disk, int bNum, void *block);
3    int writeBlock(int disk, int bNum, void *block);
```

Furthermore, the disk API will be given to students when they complete the first part of the PolyFS specification (their own disk API). Thus, the disk API must not provide students an easy way to obtain the solution to building their own disk API. It also must easily compile with students' code for their file system in the place of their own disk API, without any modifications to the existing code.

Finally, the disk API will connect to the visualizer through a server. It is not required to handle the case where the Internet is not available or deal with any issues involved with the Internet. If the Internet is available it will open a socket to the server and send the appropriate requests.

The formalized requirements are noted below:

- The disk emulator must provide openDisk, readBlock and writeBlock functions as noted above
- The disk emulator must be able to compile with students c or c++ programs using gcc or g++
- The disk emulator must meet the specification for PolyFS exactly
- The disk emulator must not have any dependencies
- The disk emulator must be provided to students in a compiled format

Several things are noted to be outside the specification for this project:

- The disk emulator is not required to store a copy of the disk locally
- The disk emulator is not required to function without Internet

4.2.5 Visualizer Elements

The purpose of the visualizer is to provide students with the ability to examine their file system easily and give them the means to dig deeper into the design decisions required in building a file system as well as how they affect the final product.

The visualizer is meant to have all the basic information ready at a glance. Therefore the visualizer must provide three main sections on a single page: disk metadata, file system block overview, and block metadata for a selected block. The disk metadata is all global metadata relevant to the disk such as the size and number of blocks. The block metadata is all the values stated in the PolyFS specification and a block's metadata should be displayed when that block is selected from the block overview.

Additionally, greater depth should be available to the students, most importantly in the form of complete data representations of the blocks. Thus, it must be easy to access both hex and ascii representations of the block data as well as examine block links.

Finally, the interface for loading disks must be transparent and easy to use. However, it is not required to provide the ability to load multiple disks at the same time, merely that switching between disks should be fast and easy.

The formalized requirements are noted below:

- The disk metadata, block overview and block metadata for a selected block must be available from a single page
- The disk metadata must include the disk name, users name, number of blocks and the block size
- The block metadata must show the values of all metadata in the PolyFS specification and the associated meanings if provided in specification (i.e. type field

must display value and the string type name)

- The block overview must easily show block type
- The block overview must be expandable to show block details
- Block details must include the data of the block in both hex and ascii representations
- An intuitive interface must be provided for loading disks

Several things are noted to be outside the specification for this project:

- It is not required to allow multiple file systems to be loaded simultaneously
- Visualizer is not required to update when the disk is updated

4.2.6 Visualizer Configurability

The application of the visualizer should not be limited solely to the current PolyFS specification. This is both because new applications for the software could arise other than showing PolyFS data and because the PolyFS specification is prone to change.

Therefore, it is required to allow the visualizer to allow the configuration of minor design decisions in the PolyFS specification. The most important of these design decisions is the size of a block. Additionally, the metadata of a block should be completely configurable with the sole assumption that it will be a contiguous component at the beginning of every block.

Since these changes may apply to only a subset of the disks, settings must be manageable at the disk level. This will allow for the addition of disks with minor changes to the visualization. However, it is not a requirement to add the ability to change the settings through the front end visualizer.

The formalized requirements are noted below:

- The visualizer block size should be modifiable
- The visualizer should make no assumptions about number of blocks
- The visualizer should have completely configurable block and byte colorings
- The visualizer metadata bytes should be configurable and of variable length but always a contiguous chunk at the beginning of the block
- The block types should be configurable
- Visualizer is configurable for each disk independently

Several things are noted to be outside the specification for this project:

- The visualizer does not need to provide a method for changing the configurations through the front end

Chapter 5

Implementation

This section explains how the PolyFS Visualizer was implemented. The goal of the section is to make clear the structure and program flow both. Additionally, for anyone choosing to add or work with the PolyFS Visualizer code base, this section provides a general overview of the code base and its organization.

5.1 Overview

The architecture has three main components. The disk emulator is formed by a server side block driver and a client side block API. The block API is compiled with the students code to transparently replace the students disk emulator. The block driver fulfills requests from the block API and edits the disk hosted on the server. The last component is the visualizer which is separate, but also relying on the students disk. The overview is shown in Figure 5.1.

5.2 Server

The server is written in Node.js with express. The server is structured with server javascript files at the base level and three directories of resources. The disks directory stores all the disks and their associated data and metadata. The static directory stores all javascipt files needed for the visualizer with the exception of libraries which are stored in the vendor directory.

The server handles several types of requests. A “visualizer request” is a request for the main page of the visualizer or any of the associated files in the static or vendor directories. An “edit request” is a request to modify a disk stored on the server. The

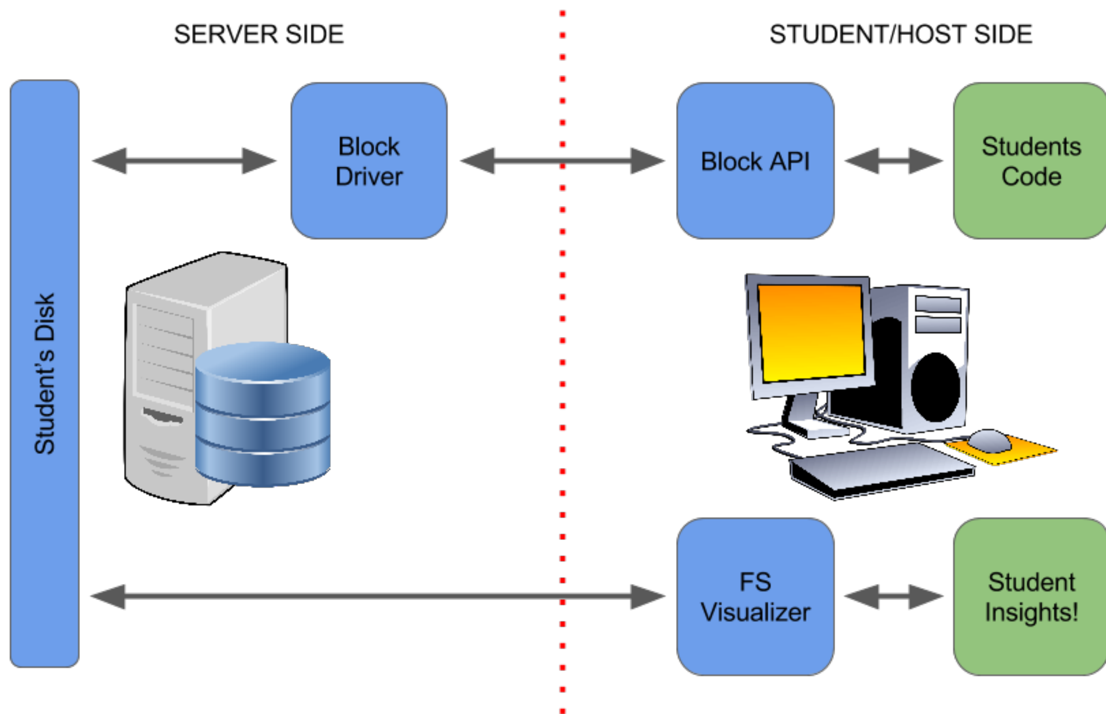


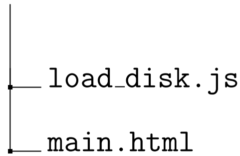
Figure 5.1: An overview of the PolyFS Visualizer architecture

last type of request is a "disk request" which is a request for the raw data of a disk within the disk directory.

Disks stored within the disks directory are viewed as static from the perspective of the visualizer and user media from the perspective of the disk emulator. This allows the visualizer to be deployed on any web server without any custom back-end. A disk includes a basic binary data file with the disk's data, a metadata file with a JSON structure of all the disks metadata and a config file with a JSON structure of the configuration of the visualizer. The binary disk data is named with the disk name and no extension while the metadata and config files are named with the disk name and .meta and .config extensions respectively.

The directory structure of the entire server and visualizer is shown below for reference.

```
visualizer
├── server.js
├── edit.js
├── disks.js
├── handles.js
├── meta.js
├── error.js
├── disks
│   ├── demo1
│   ├── demo1.meta
│   └── demo1.config
├── vendor
│   ├── bootstrap
│   ├── jquery
│   └── jcanvas
└── static
    ├── binaryTransprot.js
    ├── block_meta.js
    ├── block_modal.js
    ├── config.js
    ├── custom.css
    ├── data.js
    ├── disk_config.js
    ├── disk_meta.js
    ├── draw.js
    ├── draw_utils.js
    └── history.js
```



The architecture is design to try to isolate specific features into different files to create a more modular and extensible design. The final design is show in figure 5.2. Server.js receives all requests and passes off some of the requests on to edit.js for processing. Each file shown is given a short description at the end of this section.

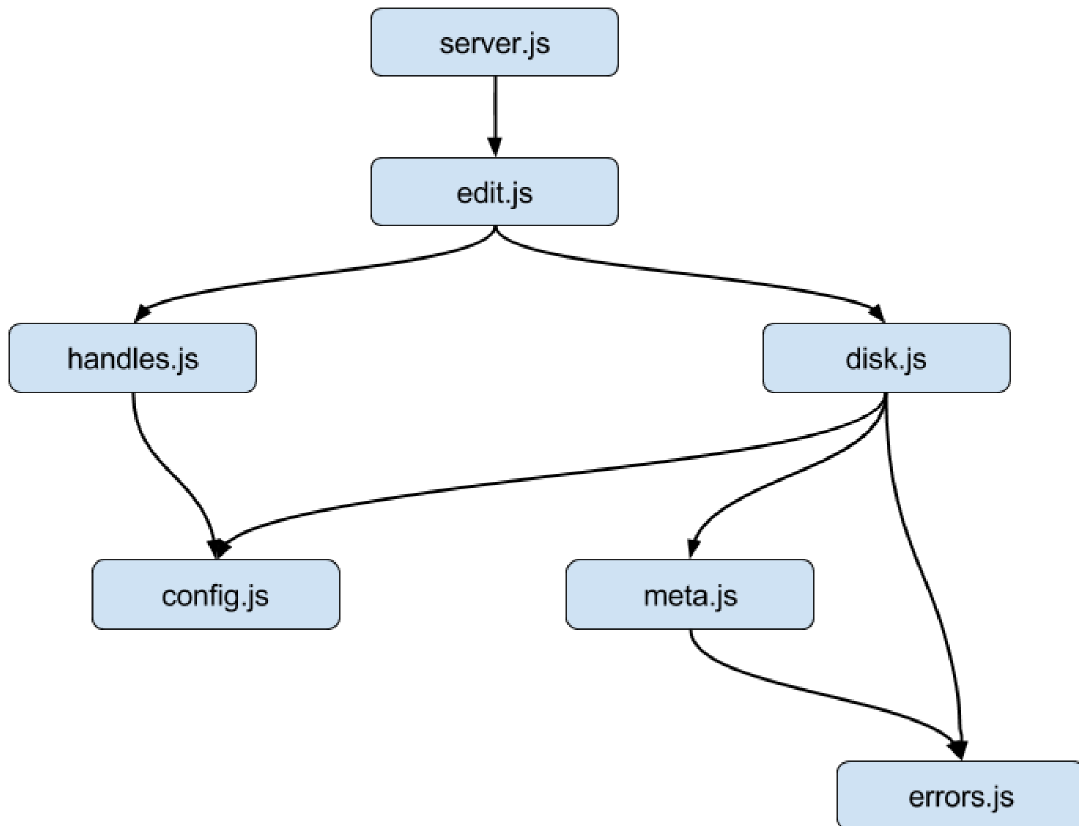


Figure 5.2: An overview of the server architecture

5.2.1 Edit Requests

The edit requests are responsible for all requests to manipulate the data of the disks on the server. These requests are handled through a separate router on the server so

they can be easily decoupled if only the visualizer is wanted.

There are three edit requests: open, read and write. Each request is described in detail below.

The open request is a JSON post request that accepts in JSON format the "name", "block_size", "num_blocks", and "user" of a disk. Name and user are string identifiers of the disk and the user respectively. If a user tries to open a disk that a different user created the request fails and a fail message is returned to the client. The block_size is a positive integer for the number of bytes in a block and num_blocks is a positive integer for the number of blocks in a disk. If any of these fields are missing from the request the request fails and a fail message is returned to the client. If num_blocks has a value of 0 then the disk will be created if it does not already exists. In this case, if it does exist an error will be returned to the user. If num_blocks does not have a value of 0 then the disk will be found and its handle returned. Again, in this case, if the disk does not exist in this case an error is returned.

The read request is another JSON post request. It accepts in JSON format the "handle" and "block". The handle is the numeric id of the disk to be read and the block is the numeric index of the block to be read from the disk. If any of these fields is missing the request fails and an error message is returned to the client. The handle is then used to fetch the disk information. If the handle is invalid an error message is returned to the user. After the disk information is fetched, the block is read from the disk and base 64 encoded before being returned to the user.

The write request is also a JSON post request. It accepts in JSON format the "handle", "block" and the "buffer". The handle is a numeric identifier of the disk to be written to and block is the index of the block to be written. Buffer is a base 64 encoding of a binary block. The disk is fetched using the handle and the buffer is written to the correct block of the disk. If the block is invalid or if the buffer is not

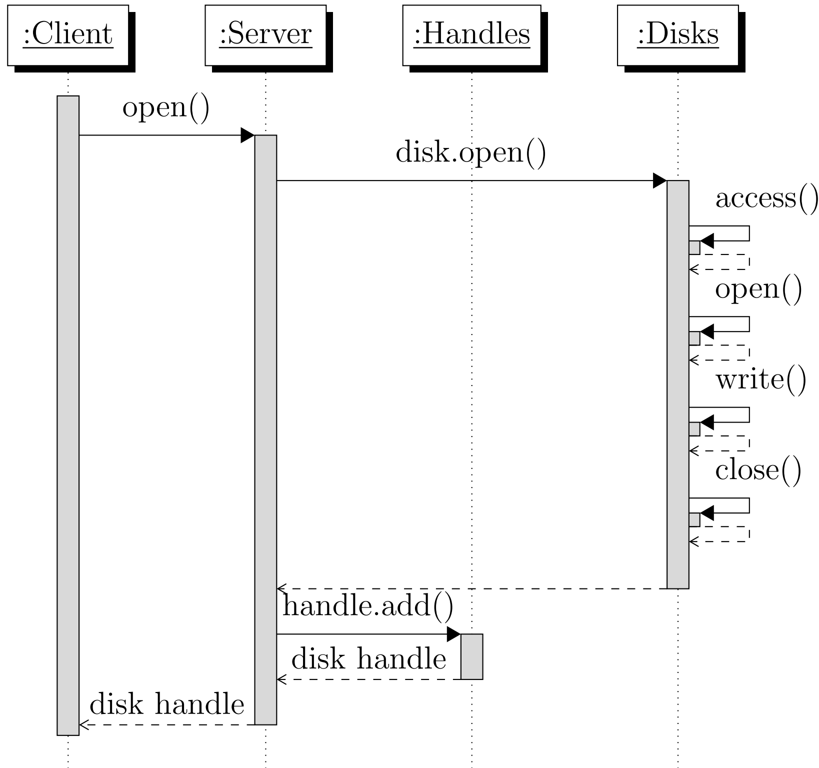


Figure 5.3: Open() flow for num_blocks = 0

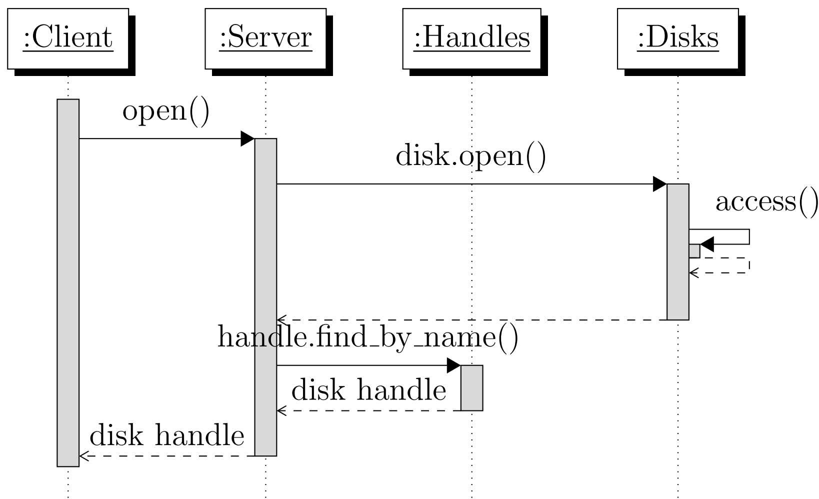


Figure 5.4: Open() flow for num_blocks != 0

the same size as a block an error is returned to the user.

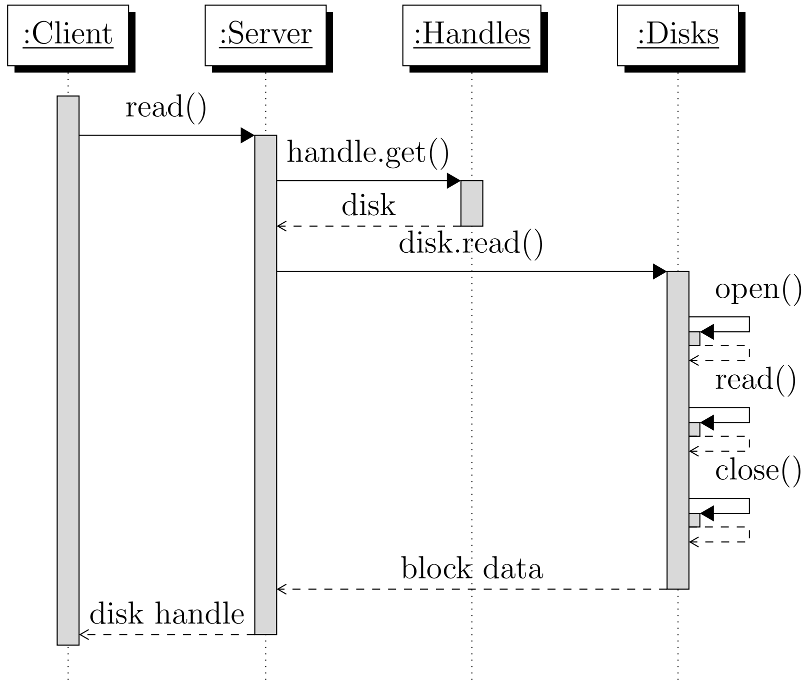


Figure 5.5: Read() flow

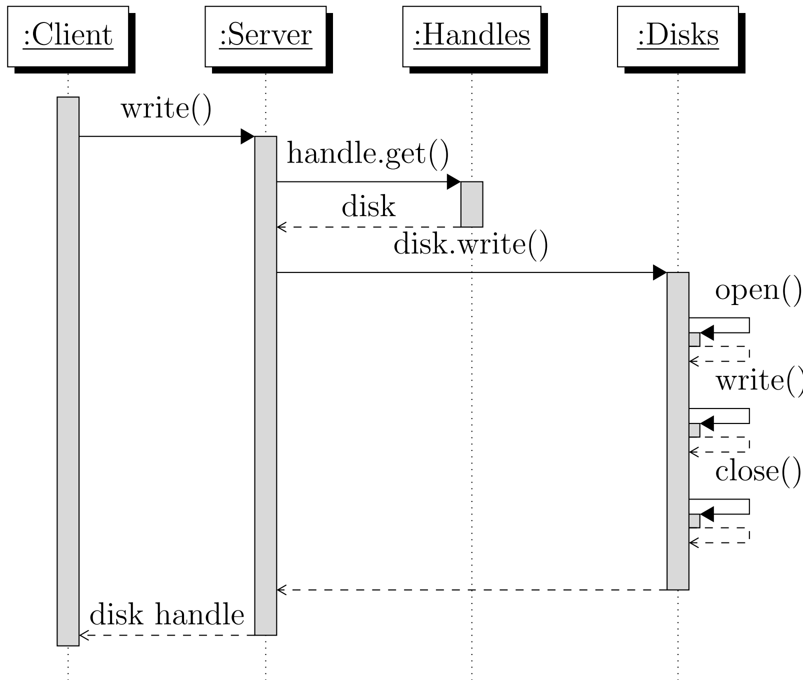


Figure 5.6: Write() flow

5.2.2 Database Architecture

The database is meant to keep track of all the disks stored by the server. The visualizer does not make any use of the database and does not track disks available to it. Instead, it just fetching disks by name and returns an error to the user if the name does not exist.

The database consists of a sole table that is a mapping from handles to the disk data. The tables columns are:

- block_size INT
- num_blocks INT
- id INT NOT NULL AUTO_INCREMENT
- name VARCHAR(30)

The block_size is the size of the block in bytes. Num_blocks is the number of blocks on the disk. The id is an auto incrementing numeric handle used to reference a disk. Finally, the name is simply the string name of the disk and can be up to 30 characters.

There is one entry per disk in the database and multiple users access a disk through the same handle in the database. The entry is added when a disk is created and removed when the disk is removed.

5.2.3 Module Overview

This section is intended to aid anyone attempting to understand the code base for the server. Each of the modules is given a short description of its goal as well as role within the server. The server is designed along block box principles with a hierarchy of modules, each dedicated to a specific aspect of the overall function.

server - The module server.js is the main for the node.js server. It starts listening on a port for incoming requests and handles disk requests and visualizer requests. It passes off all edit requests onto the edit.js module. If the 2 lines initializing the edit module and setting up the routes to it are removed then the entire disk emulation back-end is removed leaving just the visualizer.

edit - The module edit.js is the router for all edit requests. The types of requests include open, read and write. Edit requests are described in detail in 5.2.1. edit.js makes use of 3 modules: handles for database transactions, disks for disk transactions and meta to generate and fetch disk metadata.

handles - The module handles.js is responsible for all interactions with the mysql database. The database is used to store mappings of handles to all essential data for disk operations. Handles.js provides four functions: add_entry, remove_entry, get and find_by_name. Add_entry adds a new disk to the database, remove_entry removes the entry from the database. The get and find_by_name simply fetches and returns the entry of a given handle or the name respectively.

disks - The module disk.js is responsible for all interactions with the disk binary files. The binary files consist of a series of blocks each a determined number of bytes. Disk.js provides three methods for working with the disk on a block level: open_disk, read_block and write_block. Open_disk handles ensuring the disk exists and if instructed (num_blocks is 0) creates the disk. Read_block accepts a disk handle and block number and returns the binary data of the given block. Finally, write_block accepts a disk handle, block number and buffer and writes the buffer to the given block on the given disk.

meta - The module meta.js handles the disk metadata. It provides to methods: create and get_user. When a disk is created its create method is called to create the associated .meta file. Also when a disk is opened its get_user method is called to

check the user.

config - The module config.js contains the global configuration that may have to be changed for different deployments of the server. Currently that includes a max_block_size and max_num_blocks in order to limit the size of the disks stored on the server.

error - The module error.js is simply a mapping of errors to error codes and is used to return error codes to the client.

5.3 Visualizer

The visualizer is primarily Javascript formed around HTML. Bootstrap is used for the majority of the formatting though CSS makes up small changes on specific elements. Additionally, jquery is used for the interactions with the DOM from Javascript. Finally, Jcanvas is used to add a level of layering and functionality to the HTML canvas elements. Jcanvas allows for mouse events to be made on specific objects on the canvas making it possible to draw an interactive diagram.

Like the server architecture, the visualizer architectures is intended to isolate specific features into different files to create a more modular and extensible design. The final design is show in figure 5.7. In addition to this architecture is html in main.html and a config in config.js. These two files are left off of the diagram as they are above and below the Javascript code shown. All of the files can use the config.js without affecting any other files since config.js is completely static. The exact opposite is true with main.html, which uses all of Javascript files and can be edited by them. Since these edits are independent (independence is guaranteed since each file edits separate parts of the DOM with no overlap), main.html remains outside the hierarchy.

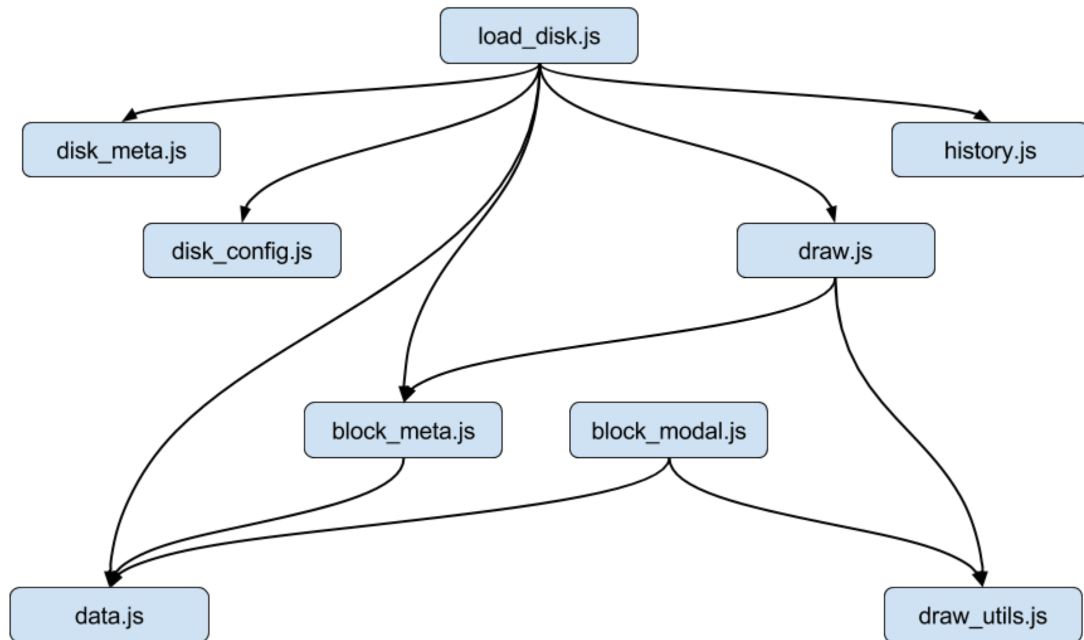


Figure 5.7: An overview of the visualizer architecture

The site has one page and three tabs linking to modals. The first tab opens up the load disk modal allowing the user to specify the name of the disk they wish to load. The second tab is for instructions and the third is a simple about-this-project modal. Most of the information is on the main page which has five main elements: disk metadata, history, block diagram, block metadata and legend. These elements get updated when a disk is loaded.

At the top of the left margin, the disk metadata panel displays statistics about the disk included the block size and number of blocks. Below the disk metadata panel the history panel tracks the disks the user has loaded and provides quick links to each one. The center panel has an array of blocks representing a disk, each block is colored to represent the block type. Blocks in the array can be inspected by mousing over them, populating the block metadata panel on the top of the right column. Also clicking on a block brings up a detailed representation of the data in the block modal.

The coloring of the blocks is explained by the last element on the main page, the legend at the bottom of the right column.

Before a disk is loaded most of the panels are not populated with any data and remain either blank or with a shell for the data that will be loaded as seen in Figure 5.8.

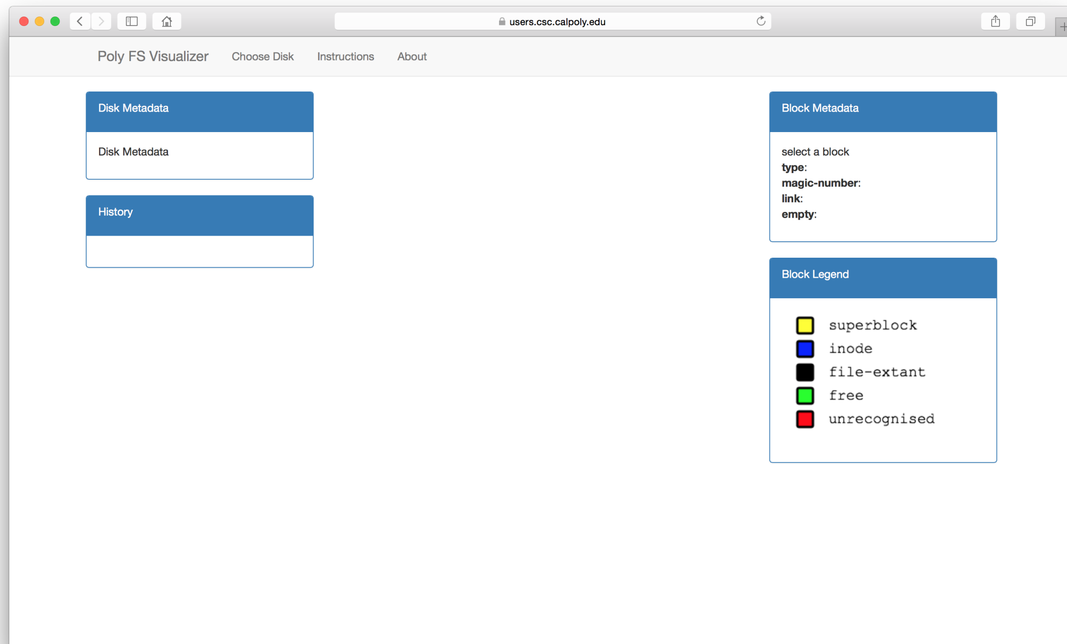


Figure 5.8: The visualizer before a disk is loaded

After disks are loaded then most of the panels populate with content. Figure 5.9 shows a picture of the main page with the disk “demo2” loaded. Note that “demo1” was loaded previously and therefore is accessible by a link in the history panel.

If a student chooses to look at the metadata for a given block, they can easily do so by mousing over the block to inspect it. In Figure 5.10 the block metadata panel is populated with the highlighted blocks information.

Then if the metadata for that block was not specific enough the block modal

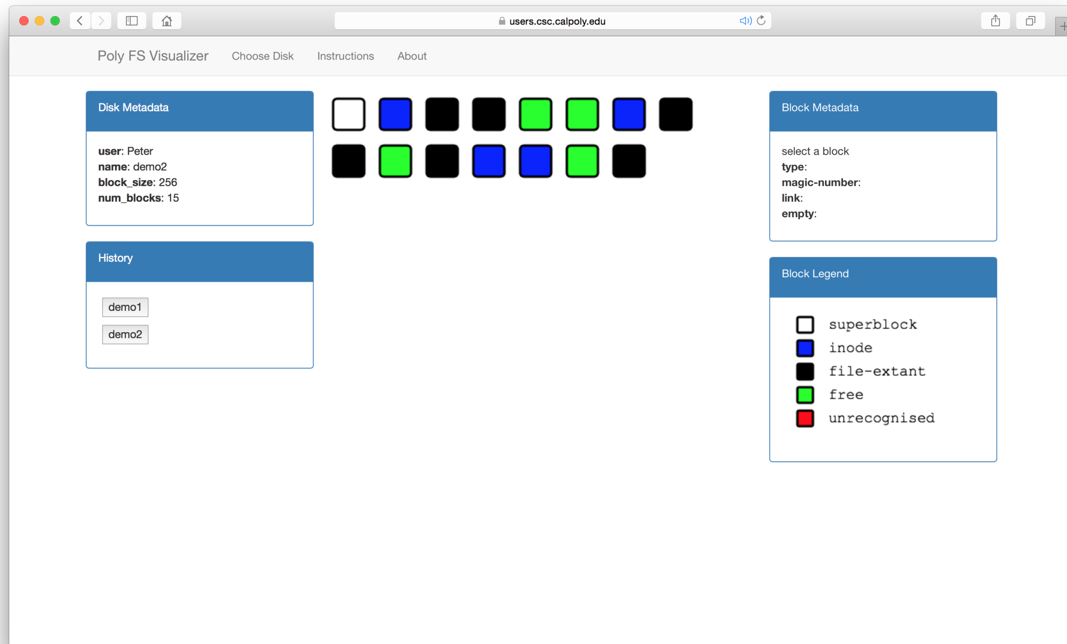


Figure 5.9: The visualizer after disk "demo2" has been loaded

could be brought up by simply clicking on the block. As seen in Figure 5.11, the block modal includes representations of the disks data in both hexadecimal and ascii as well as color coded metadata.

A new disk can always be loaded through the "Choose Disk" located in the menu bar. Figure 5.12 shows the modal.

5.3.1 Config

The visualizer is configurable both in terms of data formatting and data display. The configuration is done through the back end and read as a static file by the visualizer. This has the downside of making it impossible for students to edit the configuration of their disks on the server but the much greater upside of keeping the visualizer itself read only.

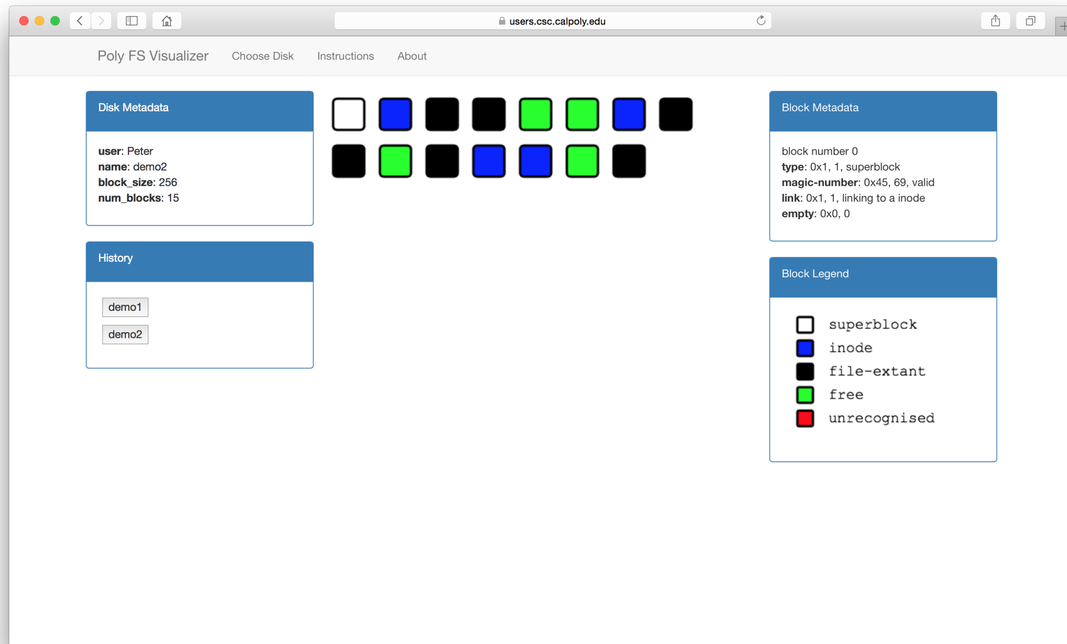


Figure 5.10: The visualizer with mouse over the first block

The config file is stored with the disks and named with the name of the disk it is associated with and the extension ".config". It is a JSON file with a specific format shown in Figure 5.13.

The data_config object tells the visualizer what different values mean for each metadata type. Each type makes up an object with its potential values as entries. For each entry is an English interpretation of that value. The English interpretation is displayed in the block metadata panel of the visualizer.

The display config object is for coloring. The block object is for configuring for block display on the main page while the byte object is for configuring the block modal's byte representations. Currently fill style is the only color configuration available but the configuration structure allows for others to easily be added. The block fill style is based off of the data_config.types English representations. Each representation has a key and the value is the RGB string color. The byte fill style is based

Figure 5.11: The visualizer with block modal shown

off of the block_config with each entry in the block_config being a key and the value is the RGB string color.

The last object, the block_config, is for block metadata. Each byte of metadata can be named and the name will be used in the block modal. The list of metadata bytes starts at the first byte of the block and continues in the order defined by the block_config list until the last entry, then the regular block data starts. It is not possible to define metadata that is not a contiguous chunk at the start of the block.

5.3.2 Disk Metadata

When a disk is created by the server, a .meta file is generated with it. If the user is looking at the visualizer without the sever then they will need to create the associated meta file themselves. This holds basic static information about the disk and is never updated after disk creation. The meta file is loaded by the visualizer in order to

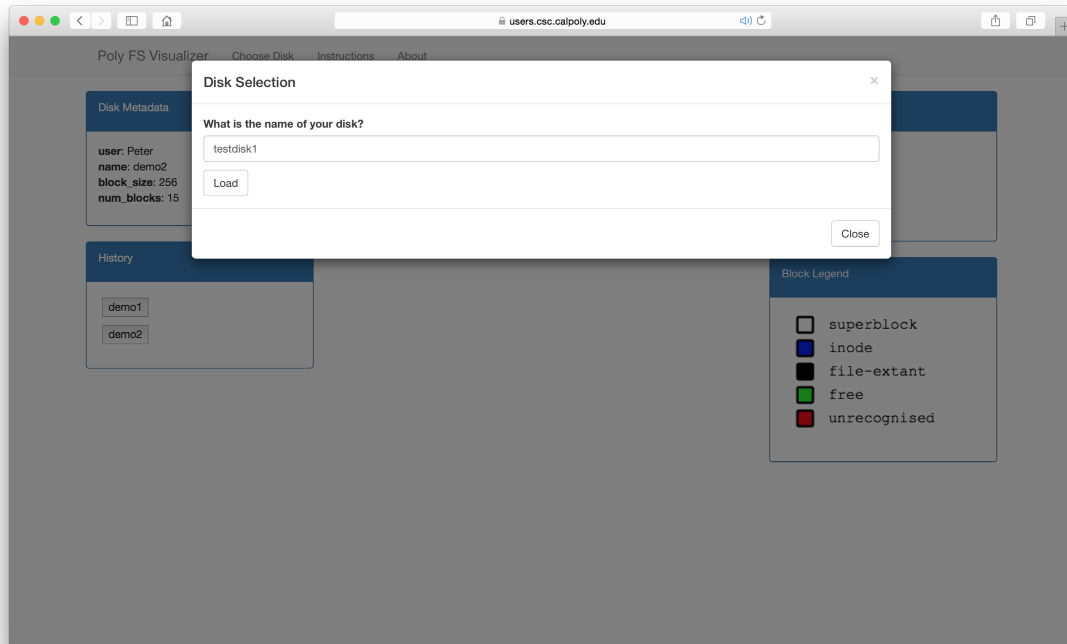


Figure 5.12: The visualizer with disk loading modal shown

understand the format of the disk well enough to parse it.

The meta file, an example of which is shown in Figure 5.14, holds only a few values. User and name are string identifier of the user and disk respectively. Block_size and num_blocks are positive integers for the size of a block and the number of blocks on the disk respectively.

5.3.3 Modules

This section is intended to aid anyone attempting to understand the code base for the visualizer. Most of the modules are given a short description of its goal as well as its role within the visualizer. The modules that were left out are made up of helper functions to be used by larger modules. The visualizer's design centers around main.html and all the modules are name-spaced so that the code is more readable.

demo1.config

```
{
    "data_config": {
        "type": {
            "0x01": "superblock",
            "0x02": "inode",
            "0x03": "file-extant",
            "0x04": "free",
            "default": "unknown"
        },
        "link": {

        },
        "magic-number": {
            "0x45": "valid",
            "default": "invalid"
        },
        "empty": {

        }
    },

    "display_config": {

        "block": {
            "fillstyle": {
                "mapping": {"superblock": "#FFFFFF", "inode": "#0000FF", "file-extant": "#000000", "free": "#00FF00"},
                "default": "#FF0000"
            }
        },

        "byte": {
            "fillstyle": {
                "mapping": {"type": "#FF0000", "magic-number": "#00FF00", "link": "#0000FF", "empty": "#FFFF00"},
                "default": "#000000"
            }
        }
    },

    "block_config": {
        "bytes": [
            "type",
            "magic-number",
            "link",
            "empty"
        ]
    }

}
```

Figure 5.13: An example configuration file

demo1.meta

```
{
        "user": "Paul",
        "name": "demo1",
        "block_size": 256,
        "num_blocks": 10
}
```

Figure 5.14: An example metadata file

The modules are all loaded through one HTML page, main.html. They also all access config.js for any configuration information. There are five modules to populate

visual panels of the web page: block_meta.js, block_modal.js, draw.js, disk_meta.js and history.js. Supporting these modules are three that load and process resources: load_disk.js, data.js, disk_config.js.

main - This is the main html for the page and the starting point for all the Javascript modules. Main.html provides the structure for the page and the modals but does not populate most of the content.

config - This is the config for the visualizer and includes all configurations needed to display a disk along with access methods. The default config and the actual running config are separated with a provided method, reset(), to set the running config back to the default. Several access methods are also supplied for to simplify config access.

load disk - This handles the loading of a disk. It directly handles the loading of the main disk binary file and calls the data module for processing. It also calls the disk config and disk meta modules for loading of their respective resources. Finally, it updates the history using the history module with the loaded disk. It should also be noted that in order to load a binary file, a custom ajax transport (binaryTransport.js) had to be used.

data - This is the processing modules that accepts a disk as binary data array and processes it into blocks. A block is an object with a block type and separate data representations as hex and char. The binary disk is broken up into blocks and processed into both binary and hexidecimal formats to create an array of blocks. This array of blocks is available to other modules to actual display the data.

disk config - This loads the config file for the disk and handles the configuration updates. The user define configuration is loaded at the root level meaning every element of the root of the JSON config object can be set by the user independently. Any root element not defined by the user is set to the default values. This module is called for every disk, so every disk has separate user configurations. If the config

file for a disk does not exist an alert is shown on the visualizer and the default configuration is used.

disk meta - This loads the metadata for the disk and handles the disk metadata panel updates. No assumption is made about the contents of the metadata except that it is a single level JSON structure. The contents are displayed in the disk metadata panel in a “key: value\n” format. If the metadata fails to load then defaults are used. However, since the disk metadata is key to correct representation of the blocks of a disk this is not recommended.

block meta - This accepts a block name and updates the block metadata panel with all of that blocks data. The block name is the numeric index of the block, and is retrieved from the name of the block element drawn on the canvas that the user has selected. Block_meta.js uses the block config from the config module to identify the metadata for the block and displays both the hex and binary representations of the data in the panel. It also calls a per-type custom function that allows for interpretation of the metadata into meaningful results. For example the type 0x01 can be represented as the string “super block”.

block modal - This accepts a block and updates the block modal with all of that blocks data. It has one externally used method, update_block, which accepts the block object of the block to be displayed in the block modal and sets up the display. It displays both the hex and char as well as a legend. Additionally it sets up click-able link meta-data so that links bold when you move your mouse over them and when clicked go to the linked block.

draw - This handles drawing the main canvas with the blocks and the legend that goes with it. It has two methods: draw and update_legend. Draw accepts a list of blocks and displays the list of blocks on the main canvas of the main page. Update_legend fills in the legend associated with the main canvas.

history - This accepts a disk name and updates the history panel with a quick link to that block. When the button is clicked the block is loaded without having to go through the load modal.

5.4 Client Disk Library

In order to communicate with the server and provide a transparent disk emulator for the students, the disk library replaces the student's disk emulator. The disk library formats and sends JSON POST requests to the server and then receives and parses the response. It provides the exact same API interface as the disk emulator:

```
1    int openDisk(char *filename, int nBytes);
2    int readBlock(int disk, int bNum, void *block);
3    int writeBlock(int disk, int bNum, void *block);
```

Each of the API calls is essentially a wrapper for the associated server edit request and will handle the inputs and network for each request. Also, in order to allow for easy modification, the disk emulator contains several layers of APIs in addition to the external API. These allows for different functionality to be modularized and extensible. It also allows for additional features to be hidden from the students but available to professors for use or future addition. Figure 5.15 shows the organization of the APIs in relation to each other with api.c being the external API and networks.c being in direct communication with the server.

The first API below the external API is called the "internal" API. The internal API allows greater flexibility for requests with the external API being a simplified wrapper for the internal API. The internal API has more options and features that can easily be hidden from the students but still available for professors. These features include support for user spaces and specific disk sizes. The exact declaration of the API is below:

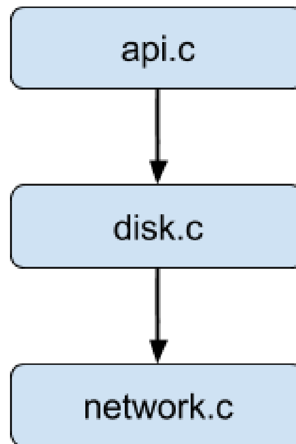


Figure 5.15: An overview of the disk emulator architecture

```
1   int send_open(char *name, int num_blocks,
2                 int block_size, char *user);
3   int send_read(int handle, int block, char *buffer);
4   int send_write(int handle, int block,
5                 char *buffer, int size);
```

Finally, the external API sends all requests using a network API that accepts the url and body and returns the response text.

```
1   void setup();
2   char *send_post_request(char *url, char *body);
3   void cleanup();
```

The end result of all the nested APIs is a separation of tasks. The network API sends data to the server. The Internal API formats the data to send the server and then formats the response to send to the user. Finally, the external API forms a simplified interface that exactly matches the requirements of the disk library in PolyFS. An example of the call stack for a request to the server is shown in Figure 5.16.

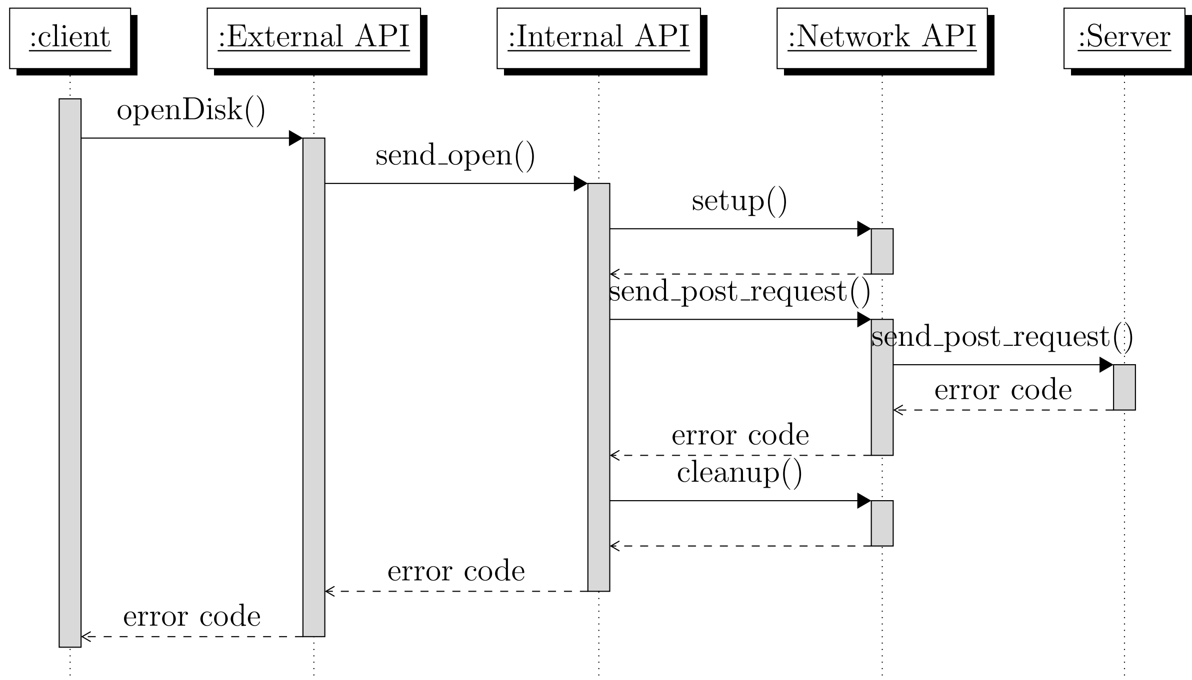


Figure 5.16: Open request to the server

5.4.1 Modules

This section is intended to aid anyone attempting to understand the code base for the disk library. All of the modules are given a short description of its goal as well as its role within the disk library. Generally, each file or module forms an API layer and works with the API below it while being called by the API above it.

api - The module api.c forms the external API and provides a thin wrapper on the internal API, simplifying the interface for the students.

disk - The module disk.c forms the internal API and provides most of the functionality for the disk emulator. It formats the JSON for the JSON requests before passing off both the url and JSON body to the network module. It also does limited error checking on the parameters. However, most of the error checking is left to the server. If an error is found, disk.c uses the same error codes as the server.

network - The module network.c forms the network API. For every request it

opens a socket to the server and sends the request. It the cleans up the socket after the request has been made. If an error happens in sending or receiving the packet then an error message is printed and the program is exited. However, if an error happens when the response is being parsed then an error is printed and NULL is returned by send_post_request().

base64 - The module base64.c provides a library for converting to and from base64 data. This is required to base64 encode the binary data blocks in order to send them in the JSON POST requests.

Chapter 6

Software Validation

This section provides an explanation of how the code base was tested to ensure it was working properly. In this case, working properly means all the components function according to design with no unintended results even for invalid inputs. However, this section does not intend to analyze the validity of the design itself, merely that the implementation is complete and correct in following the design.

6.1 Test Sets

The software was validated through the use of test cases with a combination of manual verification of test case results and automated error code checks. Test cases were meant to test both valid operations or valid inputs and invalid operations or invalid inputs.

The entire test suite is composed of five sets of tests, each targeting a different interface of the PolyFS Visualizer. All of the tests were conducted as block box tests, with know knowledge of the internal workings of the code. The five interfaces identified as potential risks were the external API, internal API, server requests, disk file, .meta file, and the .config file. Though the internal API and server requests are not user facing, they are more flexible allowing for in depth tests not possible with the user facing APIs. The other three interfaces are user facing and thus must be tested to be able to handle user actions and mistakes. It should be noted now that though limitted testing has been done to address basic security concerns, security was not a part of the software validation and has not been addressed in depth in any part of this project.

Table 6.1 shows all the test sets as well as the interface they are intended to test and the a generalized testing goal for the types of test and their coverage.

Table 6.1: The test sets used to verify the PolyFS Visualizer

Test Name	Interface tested	Test Type
externalTest	external API	general usage
internalTest	internal API	general usage
networkTest	server API	malformed server requests
configTest	Configuration file	malformed .config files
metaTest	Metadata file	malformed .meta files
diskTest	disk file	malformed disk files

The following sections explain each of the test sets in turn. The test performed and the test coverage they provide is examined as well as the results of the tests.

6.1.1 External Tests

The external API is the api the students use to replace their disk emulator and exactly matches the disk emulator. Therefore, testing was required to ensure it maintained the same level of functionality according to the PolyFS specification.

The test cases are meant to cover the possible error cases. First is simply no error, with several tests for the correct operation of each API. Each input is then tested independently, with the error codes checked against expected values to ensure the visualizer caught the input being tested and not some other problem. Finally, null inputs and similar values are tested (i.e. NULL, 0 and ""). These test cases combine to give good coverage of the possible errors the students could make.

Since the external API is written in C, a strictly typed language, the required

test space is greatly decreased as each parameter has limited inputs. Thus these test cases, which test a wide range of values for each parameter including 0, above the upper bound, below the lower bound, and a valid test case are adequate to have reasonable assurance of proper execution. The external tests test the disk emulator and the server back-end, ensuring the student facing API is functioning. A summary as well as the results of the test can be seen in Table 6.2.

Table 6.2: Table of the external test suite and results

Test Name	Test Description
open()	
1	Test standard usage, disk exists
2	Test standard usage, disk does not exist
3	Invalid disk name
4	Null inputs
5	Invalid number of bytes
6	Disk does not exist
7	Disk exists, should not create
read()	
8	Test of standard usage
9	Invalid disk
10	Null inputs
11	Invalid block number
write()	
12	Test of standard usage
13	Invalid disk
14	Null inputs
15	Invalid block number
16	Invalid buffer size

6.1.2 Internal Tests

The internal API allows additional flexibility than the external API. Thus several additional tests can be done at this level that are impossible from the external API.

Furthermore, future use of the PolyFS visualizer and potential future changes of the PolyFS spec could easily cause changes in the external API to take greater advantage of the internal APIs flexibility. In this case, the internal API would have to function outside the bounds of the external API tests.

The internal API tests focus on the ability of the internal API to specify block size and number of blocks independently as well as the size of the write buffer. These features provide the ability to easily change the block size of the disk emulator. For each function in the API several tests of the standard use cases are performed. In addition to these tests, tests of various block sizes and numbers of blocks were done with the send_open(). These parameters were tested independently, keeping the other constant and the error codes were checked to ensure the correct error was caught and reported. Similar tests were performed for the write() testing the functionality that was not visible in the external API: the ability to specify buffer size. Again, attention was paid to which error code was reported to ensure the test was successfully reaching the correct case.

Tests of the read() API were unnecessary because the external API read simply calls the internal API equivalent with no manipulation of parameters.

Code coverage was primarily achieved through the combined external and internal tests. The tests listed here ignore all the cases already covered by the external tests and focus purely on the additional functionality presented by the internal API. This test focuses on the disk emulator and the server back-end, without testing the visualization front-end at all. A table of the tests as well as the results can be seen in Table 6.3.

Table 6.3: Table of the internal test suite and results

Test Name	Test Description
send_open()	
1	Test of standard usage
2	Various block sizes
3	Various numbers of blocks
4	Null inputs
send_read()	
tests unnecessary, completely covered by external tests of read()	
send_write()	
5	Test of standard usage
6	Invalid disk
7	Null inputs
8	Various block numbers
9	Various block sizes
10	Various buffer sizes
11	Large disk writes

6.1.3 Server Tests

The server itself can be used without the disk emulator, providing another interface to test. Since the server can be used independently of an provided client side code, it should be tested to ensure it can handle inputs in formats other than those provided by the disk emulator.

The tests focus on malformed JSON and packets. Malformed requests are tested

with each parameter missing as well as the wrong type. Additionally, tests of JSON syntax errors and empty JSON objects were made. These tests were conducted by leveraging the network API in order to completely customize the body of the requests, allowing for malformed requests.

Again, like the internal tests, test coverage is achieved here by focusing solely on the cases missed by the internal and external tests. These tests focus on the server back-end and has only a minimum of client side code to run the test. The tests as well as their results can be viewed in Table 6.4.

Table 6.4: Table of the server test suite and results

Test Name	Test Description
editopen	
1	Test of standard usage
2	Invalid JSON
3	Wrong parameter types
4	Missing parameters
editread	
5	Test of standard usage
6	Invalid JSON
7	Wrong parameter types
8	Missing parameters
editwrite	
9	Test of standard usage
10	Invalid JSON
11	Wrong parameter types
12	Missing parameters

6.1.4 Disk Tests

The disk tests focus on the visualizer as an entity separate from the server. When deployed on its own, the user will provide the disk, .mete and .config files, introducing the possibility for additional errors not found in the system with the server. These tests are focused solely on the visualizer and do not test the server back-end or disk emulator.

The main tests is with the disks used by CPE 453 students with TinyFS. However, these disks are rather specific with set block size and number of blocks. Therefore, other tests were run with a variety of disk configurations and full disk writes in order to ensure the disk could be configured and written. In addition to valid disks, different varieties on invalid disks including disks with incomplete blocks, missing disks and simply empty disks.

Since the disk is binary data and only structured by both the server and the visualizer in terms of blocks, testing differences in data on the disk has no value. Instead, tests must evaluate the amount of binary data in comparison to the block size as well as the size and number of the blocks. Therefore, these tests, while not providing complete code coverage, do exercise all main visualizer elements and features.

Table 6.5: Table of the disk test suite and results

Test Name	Test Description
1	Test of standard usage
2	Disk missing
3	Incomplete blocks
4	Empty disk
5	Various valid disk sizes

6.1.5 Meta Tests

Similar to the disk tests, the .meta file can be provided by the user when the visualizer is deployed separate from the server. Thus, testing of the .meta file in the context of the visualizer is necessary to ensure users inputs will not cause unintended results and provide proper error handling and user messages to fix invalid files.

Since the .meta file is a JSON file the tests revolve around detecting invalid JSON and then missing required data. The first test is simply of a standard, correct .meta file while the rest of the tests focus on categories of errors.

The goal of these tests is to show that the visualizer will detect and throw out invalid metadata files, always managing to load the disk. Since the format is JSON and validated JSON parsers are used, the tests can ensure code coverage even with limited testing of invalid JSON formats. Completeness of the testing of valid JSON formats is easier, with tests for of missing and extra keys being enough because the .meta file is ambivalent to the existence of most keys.

Table 6.6: Table of the meta test suite and results

Test Name	Test Description
1	Test of standard usage
2	Meta file missing
3	Invalid JSON
4	Empty meta file
5	Extra keys
6	Missing keys

6.1.6 Config Tests

The last of the files editable by users when the visualizer is used as a separate unit is the config file. The .config is most complex than the .meta file and has several more failing modes. Where possible, the user needs to be notified of the errors so they can be corrected and where notification is not possible the visualizer needs to contain failures.

The tests of the .config file are similar to the .meta file, with simply more individual tests for each grouping. However, given the wide variety of possible valid config files, a limited selection of the possibilities is tested to verify performance while the most of the tests are conducted to ensure failures are found, contained and reported. These tests include missing config files, empty config files and invalid config formats. The possible invalid configuration formats are broken down into invalid or missing keys, missing or empty objects or values and wrong value types. A grouping of tests covers each of these cases.

Testing all three of the types of invalid configurations allows for reasonable confidence of the code coverage of the tests in terms of handling errors. The ability of the visualizer to correctly execute the config in the visualizer is shown through examples but incomplete simply due to the large possible combinations of configurations.

Table 6.7: Table of the config test suite and results

Test Name	Test Description
1	Test of standard usage
2	Config file missing
3	Empty Config file
4	Invalid/Missing Keys
5	Missing/Empty objects and values
6	Wrong value types

6.2 Deployment

This section discusses the testing used to determine the portability of the PolyFS Visualiser to different platforms. Since the visualizer is intended for students, the requirements for computer and browser are as few as possible. Thus, the visualizer must be tested to confirm its functionality on a variety of browsers, operating systems and screens.

The visualiser was never intended for mobile or tablet applications, so these are outside the scope of the tests.

6.2.1 Server Operating System

Though web applications and servers are supposed to be far removed from the operating system, minor changes in how API calls act on their parameters can easily break servers. Thus full deployment on the supported operating systems was necessary in order to validate the server.

It was decided to support two operating systems OSX and Ubuntu Linux. De-

ployment was completed on both operating systems and all of the server oriented tests on both. These tests include the external test, internal test and server tests. It was not necessary to run the visualizer tests on each operating system since the visualizer is isolated from the operating system by the web server.

6.2.2 Resolution

Another major concern with deployment is the diversity of screen sizes in computers used by students. Testing is required to ensure the visualizer UI is usable for all reasonable resolutions. The visualizer was exercised on a selection of different resolutions.

Importantly, since the project is intended for students in Cal Poly CSC classes, the resolutions of the Cal Poly computer science lab was tested specifically. The tests performed as well as the computer that was used to perform the tests are listed below. Since all errors that were found have been corrected and are no longer present in the final project, the outcome of the tests is invariably success and thus has been left out of the table.

The tests conducted are shown in table 6.8.

Table 6.8: Table of the resolutions tested

Test	Resolution	Computer	Operating System
1	1920x1200	Macbook Pro 2016	OSX (Yosemite)
2	1680x1050	Macbook Pro 2016	OSX (Yosemite)
3	1440x900	Macbook Pro 2016	OSX (Yosemite)
4	1280x800	Macbook Pro 2016	OSX (Yosemite)
5	1024x640	Macbook Pro 2016	OSX (Yosemite)
6	1680x1050	Cal Poly Computer Science Lab	Centos
7	2560x1440	Cal Poly Computer Science Lab	OSX

6.2.3 Browser

Browsers must also be explicitly tested as many create slightly different environments.For example it was found that the PolyFS Visualizer conflicted with some environment variables in the chrome browser. Since, from personal experience, students use primarily Firefox, Chrome and Safari it is sufficient to support those three browsers.

Additionally, portability between operating systems in addition to browsers was found to be an issue. For example, Linux and OSX calculate the height of canvas elements differently. Thus, testing was done for each of the browsers for each of OSX, Linux (Centos) and Windows 10. However, safari was not tested on browsers other than OSX since it is so rarely used on any other operating system.

The tests conducted are shown in table 6.9.

Table 6.9: Table of the browsers tested

Test	Browser	Operating System
1	Firefox	OSX (Yosemite)
2	Safari	OSX (Yosemite)
3	Chrome	OSX (Yosemite)
4	Firefox	Linux (Centos)
5	Chrome	Linux (Centos)
4	Firefox	Windows 10
5	Chrome	Windows 10

Chapter 7

Experiment

Since this thesis is an educational tool, its purpose is to aid students to gain a better and deeper understanding of the material. Thus, in order to measure the effectiveness of the PolyFS Visualizer the abilities of the students must be assessed in comparison with students with similar curriculum's minus the visualization software.

7.1 Hypothesis

With this basic concept a hypothesis can be formed: The PolyFS Visualizer increases students ability to understand file system concepts.

7.2 Experimental Design

In an ideal experiment, two identical groups of students would be placed into identical classrooms with identical professors and curriculum. One group would also be provided the PolyFS visualizer as a platform with which to do their file system project while the other group would complete the same project with the currently used tools instead of the visualizer. Both groups would receive a test at the end of the project meant to test understanding of file systems. The test would have a very wide range of questions not given in the class, allowing for a good scale of the students ability to be developed. The comparison of the two groups could then be used to determine if the PolyFS Visualizer aided in a significant increase in student understanding.

Thus the independent factors in the experiment would be the students, the learning environment, curriculum and professor. The dependent variable would be the usage of the PolyFS Visualizer. The measured variable would the the test scores of

both groups of students from after the file system project was completed.

This experiment attempts to get as close to the ideal as possible. Since we must work with live classes, it is inappropriate to split the classes and provide some with the visualizer and provide no tools to the other. Thus the experiment was be split over two quarters with a the PolyFS Visualizer provided in the second quarter. Survey 1 will be given to students in the first quarter to analyze their needs, techniques and tools and then survey 2 will be given to students in the second quarter with the exact same questions plus a few specific to the PolyFS Visualizer.

Between these two surveys we hope to draw conclusions about how students complete the TinyFS and PolyFS project and how they use the PolyFS Visualizer.

7.3 Concerns with Experimental Design

There are several problems and concerns with the current experimental setup. Firstly, no two groups of students are identical. Different pools of students will not provide the same data even in the exact same conditions unless they are of very large size. The performance of individual classes (the most basic pool of students available) often varies greatly over quarters or even sections during the same quarter. Thus it is harder to draw conclusions based on comparing two classes. Secondly, the Cal Poly class schedule does not allow for the exact same teaching environment and professor for both groups of students in the experiment. Given the option a preference should be made for having the same professor over having the same classroom and environment due to the overwhelming effects of different professors on students understanding (i.e. professors apply emphasis to different topics within the same curriculum).

In this study we recognise the bias applied by the lack of constant independent variables. As a result the data collected in the two surveys is suspect and can not be trusted entirely. However, the efforts made to create a fair test allow us to draw some

conclusions from the results in an effort to get closer to understanding how students learn file systems and apply what they learned.

7.4 Survey One Results

Survey 1 was conducted in Professor Foaad Khosmood's Winter 2016 CPE 453 class and 28 students filled out the survey. The survey was given to students after they had completed the TinyFS assignment for the class. All students completed the survey in its entirety, although several responses were clearly jokes or filler and were excluded.

For the open response questions, it was necessary to break down students responses into specific items. This was done by going through each item and placing them in a bin, if they did not fit any current bin a new one was created for it. The responses therefore have been broken down into the core concepts the student was trying to get across.

The first prompt for the students was "Name some of the challenges you faced with this project". Figure 7.1 shows the a chart of how each students answered. It is interesting to not that the largest problem concerning students was implementation, followed by design and testing. The root cause of struggle with the implementation can be several things: the underlying design is inherently flawed leading to difficulty implementing it, the programming itself is difficult, or the sheer size of the program is daunting. The PolyFS Visualizer would be able to aid students understand what their implementation is doing by displaying their disk. This would leave the programming and debugging to the students while providing them a bit of help to bridge the concepts with concrete code.

The second question, shown in figure 7.2, for students was "How did you verify your file system was storing data correctly?". This question is in reference to the PolyFS Visualizer as a testing tool that would easily allow students to see if their

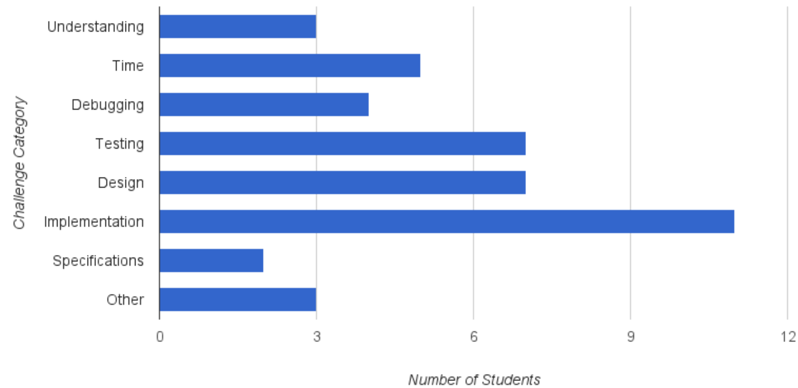


Figure 7.1: Survey 1: Aggregation of "Name some of the challenges you faced with this project"

data was being written correctly. In the absence of the PolyFS Visualizer, over half the students stated they analyzed the data manually. Many stated they used hexdump to read the data. This methodology lends itself very well to the PolyFS Visualizer which basically acts as a more intelligent hexdump by delineating the blocks and breaking out the metadata.

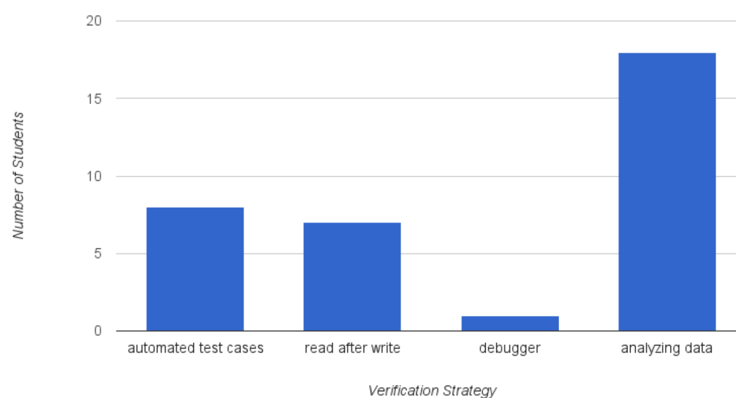


Figure 7.2: Survey 1: Aggregation of "How did you verify your file system was storing data correctly?"

The third question is “What tools did you use to help you with this project?”. This question was intended to help determine what students used to aid them with their projects in order to determine what tools the PolyFS Visualizer could replace and how it would be better than those tools. The PolyFS Visualizer was intended to replace hexdump which was theorized to be a common tool for this project. The graph is split up into categories with students answers for what they used within a category show. The categories are version control, documentation sources, editors, debuggers and visualizers. Hexdump and hexedit were considered visualizers because their main purpose for this project is to edit the disks not the code directly. Interestingly, 25% of the class mentioned they used hexdump or hexedit, despite the extremely open ended question. The rest of the categories clearly reflect how Cal Poly students are taught, favoring git repositories, vim and GDB which are all taught and supported better than their peers.

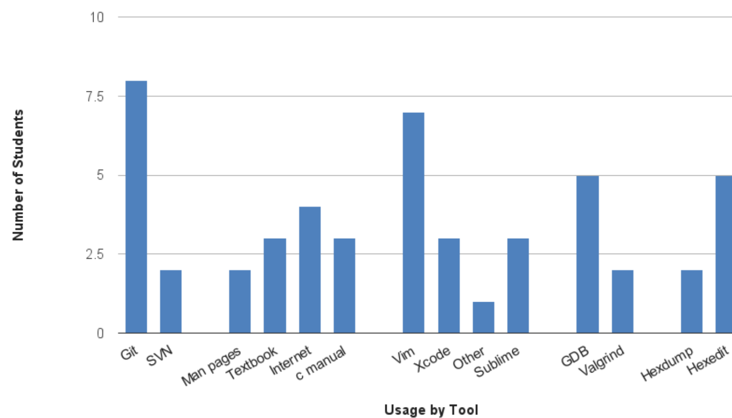


Figure 7.3: Survey 1: Aggregation of “What tools did you use to help you with this project?”

The fourth question, shown in figure 7.4, is “How could this project be improved”. Asking for improvements is intended to examine what students think needs to be changed for the project in order to figure out if a tool along the lines of the PolyFS

Visualizer would be a good addition to the class. Students may have mentioned more than one category and thus the total of the results will not add up to the number of students. It is very interesting to note that the most popular category is asking for more functionality and options to implement within their file system. It seems students are looking to expand the project. Tied for second is testing help and specifications. The PolyFS Visualizer is clearly intended to aid students in testing their code and should help the students who said the former. The later on the other hand results from either poorly written specs or a poor understanding of the concepts on the part of the students. The ability of the PolyFS Visualizer to show students demo disks and let them examine an actual disk implementing PolyFS will probably greatly help them understand what the spec is expecting and the concepts it covers.

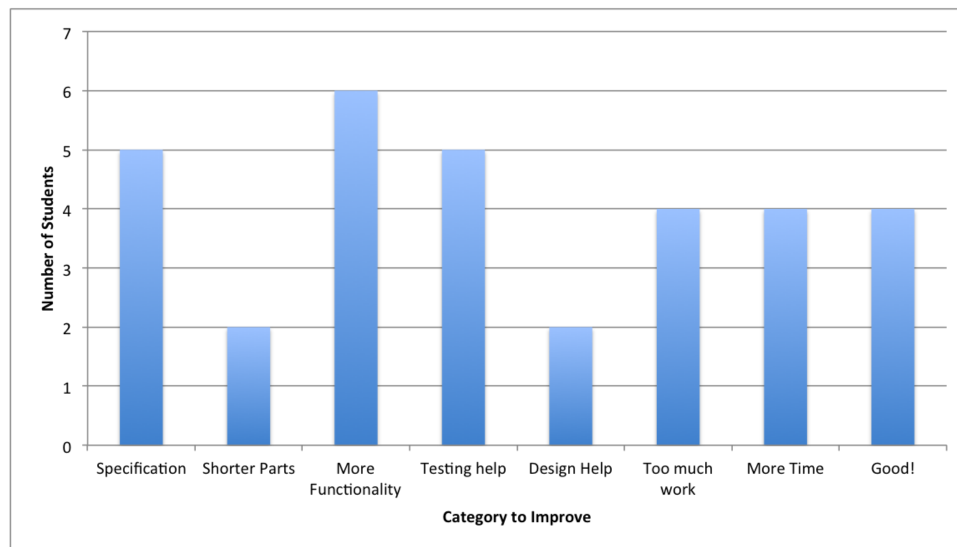


Figure 7.4: Survey 1: Aggregation of "How could this project be improved?"

The last set of questions are all agree/disagree questions, with the results shown in figure 7.5. Students were given a statement and asked to what degree did they agree with the statement. The options were strongly agree, agree, neutral, disagree, strongly disagree. The first and second statements were "The project" is difficult and

"The project was helpful in understanding file systems". Students overwhelmingly agreed or strongly agreed with these two statements showing an excellent mix of difficulty, file system concepts and time.

However, when asked the next two questions, "I had difficulty visualizing the content of the disk" and "It was hard to design the superblock", about half the students disagreed or strongly disagreed, possibly showing they did not need help from a visualization service. Of the remaining students less than a quarter stated they agreed to either statement. However, give the number of students that cited design and testing to be difficult portions of the project in the previous questions, it seems likely that the original meaning for the question was not clear to students. It is likely that the word "visualize" was ambiguous to the students and if they answered based off of their ability to read the disk through hexdump then many of them would have disagreed.

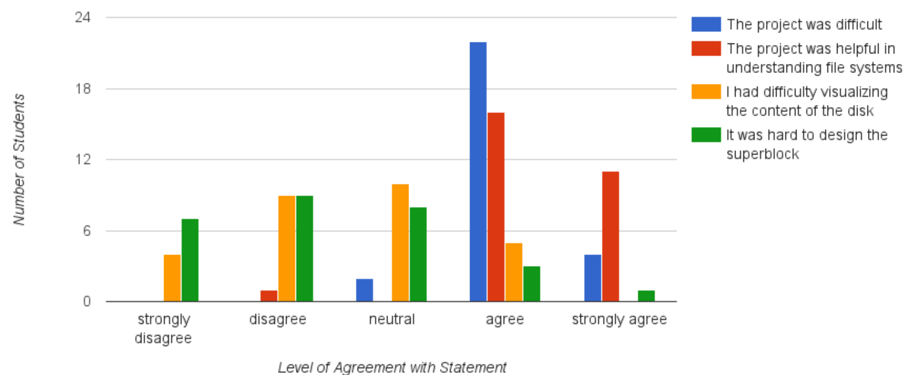


Figure 7.5: Survey 1: agree/disagree grid questions

Finally, one students actually stated in their answer that having a program like the PolyFS Visualizer would be really helpful. The student said:

> "Something which would have been *really* cool would have been a real-time hexdumping libDisk. Imagine if we had been provided with a lib-

> Disk.o which used ncurses to demonstrate the contents of the disk, as hexdump would, as our TinyFS is interacting with it."

This program would be a very close rendition of the PolyFS Visualizer. It would be completely command line based and focus on just hexdump rather than providing a lot of the other features however, the main concept is definitely the same. It is interesting that in an open ended question a student would state a "really cool" idea that is so close to the project we are trying to validate.

7.5 Survey Two Results

Survey 2 was conducted the quarter after survey 1 with Professor Foaad and Professor Peterson's Spring 2016 CPE 453 classes. During this quarter students were provided the PolyFS Visualizer for use with their project. Survey 2 includes all the questions in survey 1 but also has some questions specific to the PolyFS Visualizer.

The results were analyzed in the same manner as survey 1. However, the results are skewed by the necessity of taking the survey earlier in the quarter. Students had not completed the TinyFS project before taking the survey and were just finishing the first phase of the project. Thus, many were focused more on the disk emulator and had not finished the main focus of the project.

The second survey had 26 participants, although one student took the survey twice, saying they wanted to correct their answers in light of continued use of the PolyFS Visualizer. In addition, students taking survey 2 were told the survey concerned the PolyFS Visualizer tool they were testing versus students taking survey 1 who were told simply that the survey concerned PolyFS. This had a very large affect on the results as students interpreted "this project" to mean the PolyFS Visualizer when it was intended to refer to the PolyFS project they were completing. As a result, the first set of questions have a category in the aggregation of results for students that

answered the questions from the perspective of the PolyFS Visualizer rather than the PolyFS assignment. The students are still included in the results to show the bias.

As a result of these changes in the survey set up, comparisons of survey 1 and survey 2 are not valid and will not be attempted. Both surveys will be analyzed entirely separately.

Just like in survey 1, the first prompt for the students was "Name some of the challenges you faced with this project". Figure 7.6 shows a chart of how each students answered. It is interesting to note that students taking survey 2 stated they had difficulties with design and concepts. Several cited the PolyFS Visualizer as a tool that helped them understand how the disk was supposed to be set up and more specifically how their disk implementation worked. It is likely that the increase in difficulty with understanding and design stems from the timing of the survey. In the beginning of the project, students typically are going through the design and understanding portion of the work and therefore a survey given at that time would greatly skew the results toward those categories. In fact one student even stated in the survey that they had not gotten far enough in the project to begin testing and could not answer a question as a result.

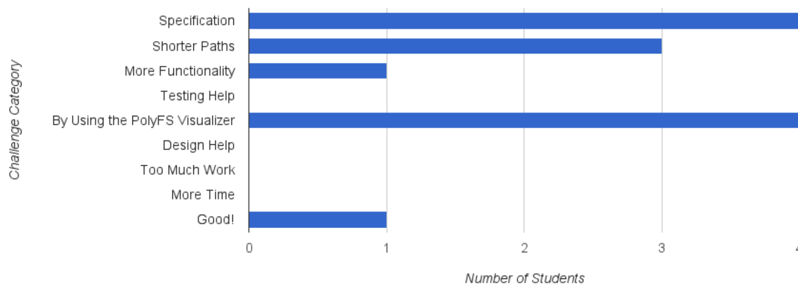


Figure 7.6: Survey 2: Aggregation of "Name some of the challenges you faced with this project"

Again, like the first survey, the second question for students was "How did you

verify your file system was storing data correctly?". The results, shown in figure 7.7, shows students often relying on the PolyFS and hexdump. This is also an artifact of the time the survey was given. At this stage of the project students are primarily working with the disk emulator which has both a small code base and a basic output, lending itself to more visual and manual methods of debugging such as examining the data and printing out debug messages.

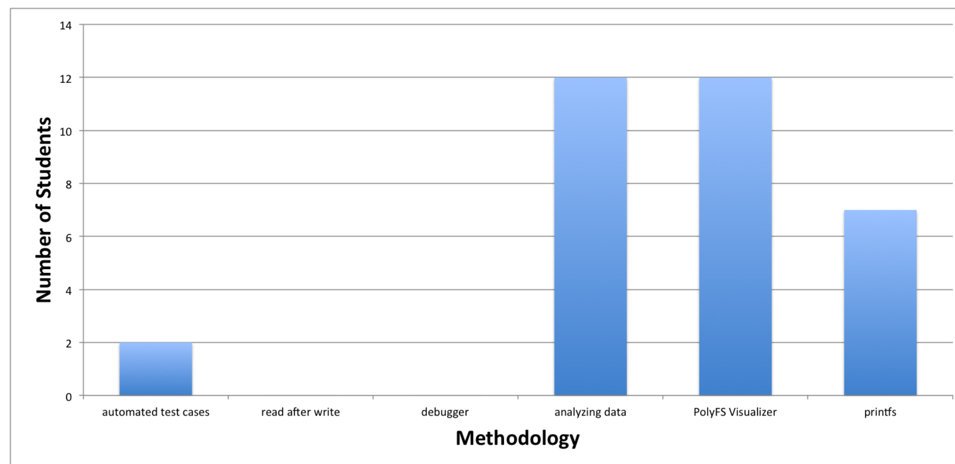


Figure 7.7: Survey 2: Aggregation of "How did you verify your file system was storing data correctly?"

The third question is "How could this project be improved?" shown in figure 7.8. This question was largely answered with problems with the ambiguities in the specification, saying the using the PolyFS Visualizer or a tool with similar features would make it better, or saying they needed more deadlines and shorter parts to help their time management. The large number of students mentioning specification is likely a result of students being earlier in the project and in the phase of trying to understand the specification and concepts. Interestingly, a couple of the students stated they needed help visualizing the content of the disk but then stated they did not use the visualizer in the survey. For example one students said in answer to this question that they needed "An easier way to see what I had written to my TinyFS file system" and yet this student did not make use of the PolyFS Visualizer. One possibility is

installing the visualizer was daunting or that the student did not even attempt to figure out what the project was and therefore did not know it would help them.

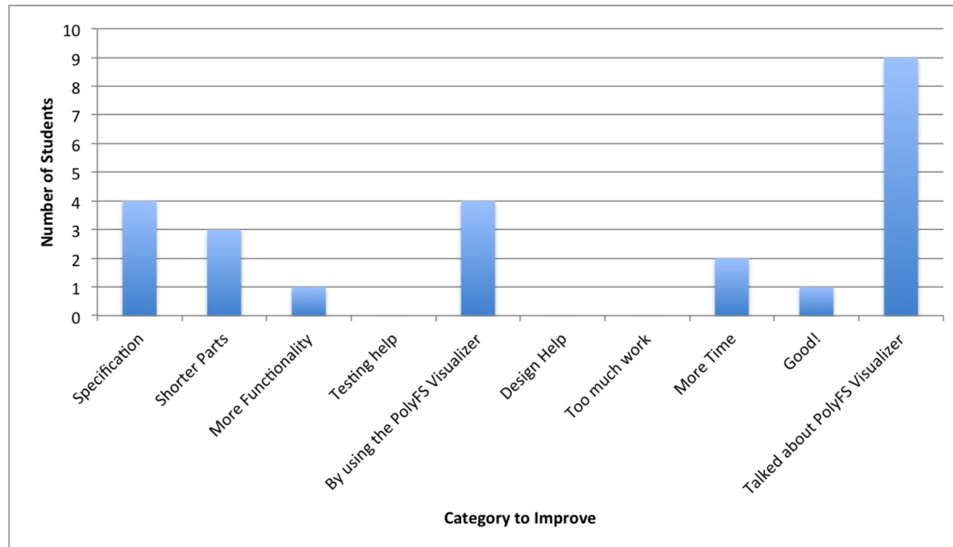


Figure 7.8: Survey 2: Aggregation of "How could this project be improved?"

The grid questions, exactly the same as the first survey, simply asked to what degree did students agree with a statement. The first two questions shown in figure 7.9 have very unsurprising results and are very congratulatory of the PolyFS project as a whole. However, the last two questions in figures 7.9 show an even split in the students. This reflects the general trend of the second survey toward greater difficulty with concepts, specifications and other items early in the development cycle.

The last figure of the after survey is not in survey 1 and consists of an aggregation of all the students comments about the PolyFS Visualizer. Since the students got confused and answered many of the initial questions in terms of the PolyFS Visualizer, this graph does not represent any one question. The results concerning the visualizer were combined per person from all the questions they answered. This allowed the general sentiment of the student to be analyzed manually, creating a more accurate conclusion over studying each question individually.

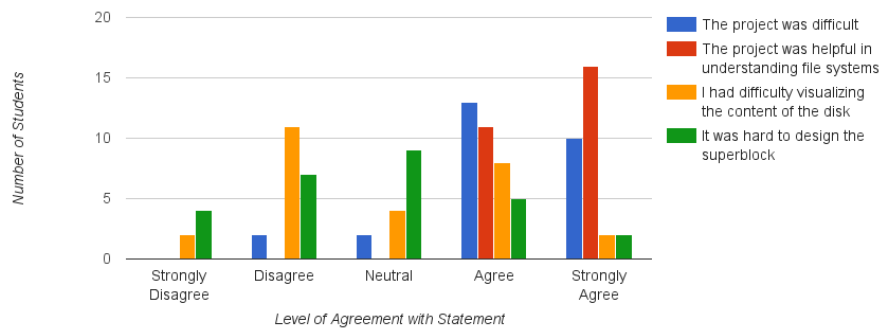


Figure 7.9: Survey 2: agree/disagree grid questions

For purposes of the survey, the students were provided with the standalone visualizer. This visualizer must be installed into the students websites hosted on the CSC servers. Additionally, students were not taught or informed of the possibility of configuring their disks or editing the meta file. The test was intended to be simple and specific to PolyFS. The graph below categorizes students comments on the standalone visualizer. It is promising to see that the two top comment categories are that it was helpful and suggestions for new features. The other categories featured in prominence deal with the installation, which turned out to be a very divisive topic. Students stated in nearly equal numbers that installation was challenging and that installation was easy.

Students stated several areas for improvement for the PolyFS Visualizer. One student stated an item that was already featured prominently in the product they tested. Since other students seemed to have little trouble finding this feature, it is likely that the student was not very thorough in their evaluation. Another student asked for a feature that is included in the extended feature set, and was not explained for the purposes on the survey. The remaining students asked for features that were not present in the visualizer and would suggest possible future work or revisions to the project. These suggested revisions are:

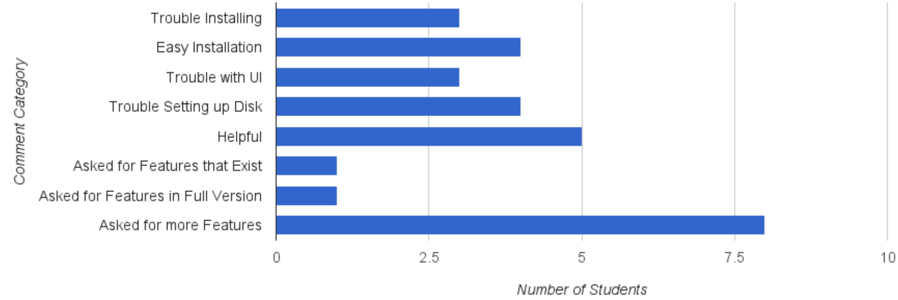


Figure 7.10: Survey 2: Aggregation of all comments on the PolyFS Visualizer

- add disk drop-down selector (requested by 3 students)
- make visualizer work locally
- change demo disk to have more clear data examples
- create an easier way to make config and meta files
- add a way to select and see two blocks at a time (the student wanted look for data overflow between blocks)

These revisions are left for future work. However, it should be noted that a drop-down selector was considered, but a method of scanning for available disks was not found creating custom back-end calls and breaking the requirement that the visualizer be deploy-able with no custom back-end. The other items are left for future work, with a potential emphasis placed on the 4 item as both the most complex and potentially the most useful item.

7.6 Survey Conclusions

The two surveys were meant to be compared directly in order to establish what difference the PolyFS Visualizer provided. However, due to limited time and experience

with the visualizer and a great difference in the timing of the two surveys, the results are not comparable. As a result any analysis of our original hypothesis is impractical and the experiment remains inconclusive. Most of the difference in results can easily be explained by the phase of the project the students were in rather than any affect of the PolyFS Visualizer. Additionally, most students did not work in depth with the visualizer and opened it primarily for the purpose of the survey rather than for actual use with their project. Therefore, it will be more important to take note of students continued use of the system now that they have been motivated to make use of it.

While the surveys could not be compared, the items on individual surveys do give us a lot of data on how students go about working on their projects and what they struggle with. Since the first survey was taken after the project was concluded and without the introduction of the PolyFS Visualizer, its results can be trusted. It shows that students generally struggle with design, implementation and specifications. Thus a tool to help with this project should target those areas. Anecdotal evidence from the second survey does show that the PolyFS Visualizer successfully targets these areas with students saying: “Using the PolyFS Visualizer from the beginning would have helped, especially seeing a working one [disk] when first starting to help understand and visualize the structure” and “CS needs more tools like this. Students that struggle always think the solution is some sort of magic, but if they could just see it’s all logical I think that would be a huge help. I am a big fan of tools like these.”. To back up their point, the later student had built a visualizer of their own with a similar purpose and feature set but not GUI or server based.

Additionally, the PolyFS Specific comments on the after survey are quite helpful. Despite the confusion causing students to put their comments on the PolyFS Visualizer under the wrong questions, the comments themselves are still valid. Thus we can conclude that the visualizer needs the most work in its installation process in order to get students to use it. In response to this survey, an installation script has

been provided that downloads, installs the standalone visualizer and starts it. Simply running the script is enough to set up the visualizer and if the student turns it off they can turn it back on by simply running the script again, it will not only restart the visualizer if it detects an existing installation.

The two other most common comments were that the visualizer is helpful and they thought it was a good addition to the project and that they would like to add additional features to the project. Both of these comments are very good signs, showing that students think the project would help them better complete the project and that they see the possibility for future improvements.

Overall, though the surveys allow for conclusion on whether or not the PolyFS Visualizer improved students understanding of file systems. However, the statistics of student's troubles with the project and anecdotal evidence from the students shows that the PolyFS Visualizer targets the right areas for students to gain the most. Hopefully, future work can help show more conclusively whether or not this is in fact the case.

Chapter 8

Future Work

This thesis provides a robust basis for file system visualization. It provides a slightly limited feature set with a high degree of confidence, allowing for future extensions to be easily built.

It was noted than many students use hex edit in order to create known disks that they could use to test their writes. Thus adding the ability to edit disks and save their contents back to the server would be a good extension of the PolyFS Visualizer. However, this would break the requirements that the visualizer be read only.

Finally additional work could be done with the visualization. Additional features of the file system could be detected and visualized. For example, since certain data types such as dates have known formats, detection of these data types is possible and could be very useful for recognising other parts of the file system.

Overall, this thesis provides a strong, dependable base visualization service that allows for a great deal of expansion and innovation in the future.

Chapter 9

Conclusion

In this thesis we presented the PolyFS Visualizer a three part system that seamlessly replaces the disk emulator in the TinyFS and PolyFS specifications in order to build the disk on a remote server and provide visualization of the disks data. In order to complete this project, we designed a set of requirements for a system that would aid students in their project without encroaching on the design portions of the PolyFS project.

The implementation of these requirements, the PolyFS Visualizer, has undergone thorough software validation on every API interface as well as user provided files. Additionally, the PolyFS Visualizer has also been tested on a wide range of machines and deployments allowing it to be easily accessible to students.

Finally, the visualizer has been tested in Cal Poly's Introduction to Operating Systems classes, where students used the system to help with their TinyFS project and responded to surveys. Though the surveys were not conclusive in showing that students gained a greater understanding of file systems through the use of the PolyFS Visualizer, anecdotes from students show that the system is helpful and a good addition to the project. One student even said "Using the PolyFS Visualizer from the beginning would have helped, especially seeing a working one [disk] when first starting to help understand and visualize the structure".

Overall, this project successfully built and tested a file system visualization service. Perhaps most importantly, the PolyFS Visualizer provides a strong base and thorough software validation for other projects that can build and improve upon what was accomplished here.

BIBLIOGRAPHY

[1] Cal poly catalog. `http://catalog.calpoly.edu/collegesandprograms/collegeofengineering/computerscience/#courseinventory`.

[2] Cal Poly Github. `http://www.github.com/CalPoly`.

[3] Tinyfs specification.

[4] B. Atkin and E. G. Sirer. Portos: an educational operating system for the post-pc environment. In *ACM SIGCSE Bulletin*, volume 34, pages 116–120. ACM, 2002.

[5] M. J. Becker. CUDA Web API Remote Execution of CUDA Kernels using Web Services. Master's thesis, California Polytechnic State University, San Luis Obispo, 2012.

[6] M. Bedy, S. Carr, X. Huang, and C.-K. Shene. A visualization system for multithreaded programming. In *ACM SIGCSE Bulletin*, volume 32, pages 1–5. ACM, 2000.

[7] T. Bladh, D. A. Carr, and M. Kljun. The effect of animated transitions on user navigation in 3d tree-maps. In *Ninth International Conference on Information Visualisation (IV'05)*, pages 297–305, July 2005.

[8] T. Bladh, D. A. Carr, and J. Scholl. *Computer Human Interaction: 6th Asia Pacific Conference, APCHI 2004, Rotorua, New Zealand, June 29-July 2, 2004. Proceedings*, chapter Extending Tree-Maps to Three Dimensions: A Comparative Study, pages 50–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[9] P. Brusilovsky and H.-D. Su. Adaptive visualization component of a distributed

web-based adaptive educational system. In *Intelligent Tutoring Systems*, pages 229–238. Springer, 2002.

[10] D. Dobrilovi and Z. Stojanov. Using Virtualization Software in Operating Systems Course. Master's thesis, 2006.

[11] B. M. English and S. B. Rainwater. THE EFFECTIVENESS OF ANIMATIONS IN AN UNDERGRADUATE OPERATING SYSTEMS COURSE. Master's thesis, Henderson State University and The University of Texas at Tyler, May 2006.

[12] B. Johnson and B. Shneiderman. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In *Visualization, 1991. Visualization '91, Proceedings., IEEE Conference on*, pages 284–291, Oct 1991.

[13] F. Khosmood and P. Nico. Polyfs: An extensible, underspecified, pedagogical file system and disk emulator. In *2013 ASEE PSW Conference Proceedings*, pages 251–268. ASEE, 2013.

[14] L. P. Maia, F. B. Machado, and A. C. Pacheco Jr. A constructivist framework for operating systems education: a pedagogic proposal using the sosim. In *ACM SIGCSE Bulletin*, volume 37, pages 218–222. ACM, 2005.

[15] L. P. Maia and A. C. Pacheco Jr. A simulator supporting lectures on operating systems. In *Frontiers in Education, 2003. FIE 2003 33rd Annual*, volume 2, pages F2C–13. IEEE, 2003.

[16] J. I. Messner and M. J. Horman. Using Advanced Visualization Tools to Improve Construction Education. Master's thesis, 2003.

[17] H. E. Meth. DecaFS: A Modular Distributed File System to Facilitate Distributed Systems Education. Master's thesis, California Polytechnic State University, San Luis Obispo, 2014.

[18] P. C. L. X.-p. NIU Zhen-zhou, YANG Xing-qiag. Research on Windows Operating System Visualization. Master's thesis, Shandong University and Jinan Vocational Collage.

[19] T. M. Peters. DEFY: A Deniable File System for Flash Memory. Master's thesis, California Polytechnic State University, San Luis Obispo, 2014.

[20] S. C. Pungdumri. An Interactive Visualization Model for Analyzing Data Storage System Workloads. Master's thesis, California Polytechnic State University, San Luis Obispo, 2012.

APPENDICES

Appendix A

TinyFS Specification

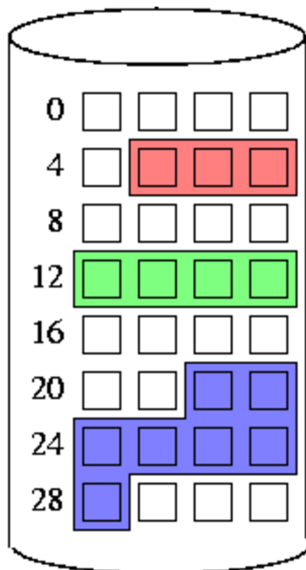
Program 4 — CPE 453

Last Modified: May 27, 2015

You may work in groups of three for this assignment

A.1 TinyFS and disk emulator

For this assignment, youll be implementing TinyFS (the tiny file system), mounted on a single Unix file that emulates a block device.



Directory:

file	start	length
moo	5	3
snow	22	7
fall	12	4

A.1.1 Objective

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

A.2 Phase I: Disk Emulator

The first part of the assignment is to build a disk emulator. You will implement an emulator that will accomplish basic block operations, like the kind supported by block devices (e.g. hard disk drives), on a regular Unix file.

A.2.1 LibDisk Interface Functions

The emulator is a library of three functions that operate on a regular UNIX file. The necessary functions are: **openDisk()**, **readBlock()**, **writeBlock()**. There is also a single piece of data that is required: `BLOCKSIZE`, the size of a disk block in bytes. This should be statically defined to 256 bytes using a macro (see below). All IO done to the emulated disk must be block aligned to `BLOCKSIZE`, meaning that the disk assumes the buffers passed in **readBlock()** and **writeBlock()** are exactly `BLOCKSIZE` bytes large. If they are not, the behavior is undefined.

```
/* This functions opens a regular UNIX file and designates the first nBytes
of it as space for the emulated disk.  If nBytes is not exactly a multiple
of BLOCKSIZE then the disk size will be the closest multiple of BLOCKSIZE
that is lower than nByte (but greater than 0) If nBytes is less than BLOCKSIZE
failure should be returned.  If nBytes > BLOCKSIZE and there is already
a file by the given filename, that files content may be overwritten.  If
```

```
nBytes is 0, an existing disk is opened, and should not be overwritten.
There is no requirement to maintain integrity of any file content beyond
nBytes.  The return value is -1 on failure or a disk number on success.
*/
```

int openDisk(char *filename, int nBytes);

```
/* readBlock() reads an entire block of BLOCKSIZE bytes from the open disk
(identified by disk) and copies the result into a local buffer (must be
at least of BLOCKSIZE bytes).  The bNum is a logical block number, which
must be translated into a byte offset within the disk.  The translation
from logical to physical block is straightforward:  bNum=0 is the very first
byte of the file.  bNum=1 is BLOCKSIZE bytes into the disk, bNum=n is n*BLOCKSIZE
bytes into the disk.  On success, it returns 0.  -1 or smaller is returned
if disk is not available (hasnt been opened) or any other failures.  You
must define your own error code system.  */
```

int readBlock(int disk, int bNum, void *block);

```
/* writeBlock() takes disk number disk and logical block number bNum and
writes the content of the buffer block to that location.  block must be
integral with BLOCKSIZE. Just as in readBlock(), writeBlock() must translate
the logical block bNum to the correct byte position in the file.  On success,
it returns 0.  -1 or smaller is returned if disk is not available (i.e.
hasnt been opened) or any other failures.  You must define your own error
code system.  */
```

int writeBlock(int disk, int bNum, void *block);

A.3 Phase II: TinyFS file system implementation

TinyFS is a very simple file system. It is purposefully under-specified, giving you the freedom to implement it using many of the algorithms and primitives youve learned throughout this course. TinyFS does not support a hierarchical namespace, i.e. there are no directories beyond the root directory, and all the files are in a flat namespace.

A.3.1 Block Types

The disk blocks of TinyFS may be any of these types:

Block name	Block code	Description	Number Possible	Size (bytes)
superblock	1	must contain the magic number, pointer to root inode, and the free block-list implementation	1	256
inode	2	must contain the name of the file, the file size and a data block indexing implementation	many	256
file extent	3	contains block# of the inode block	many	256
free	4	is ready for future writes	many	256

1. superblock

The superblock stores metadata about the file system and is always stored at logical block 0. The block contains three different pieces of information. 1) It specifies the magic number, used for detecting when the disk is not of the correct format. For TinyFS, that number is 0x44, and it is to be found exactly on the second byte of every block. 2) It contains the block number of the root inode (for directory-based

file systems). 3) It contains a pointer to the list of free blocks, or some other way to manage free blocks. How you implement the free block list is up to you, but it might be done, for example, by having a pointer to the first free block in a chain of free blocks, or by implementing a bit vector and storing it directly within the superblock.

2. inode

An inode block keeps tracks of metadata for each file within TinyFS. In real file systems, this is typically ownership (user, group), file type, creation time, access time, etc. For TinyFS core only the files name and size is required. You must support names up to 8 alphanumeric characters (not including a NULL terminator), and no longer. For example: file1234, file1 or f.

For inode blocks, you must design where and how to store the file metadata. This includes how you index the data blocks (file extents) that correspond to the inode. Here again, there are many possible implementations, including a linked list of blocks, direct indexing, multi-level indexing, or even content-based addressing (see additional features below).

3. file extent

A file extent block is a fixed sized block that contains file data and (optionally) a pointer to the next data block. If the file extent is the last (or only) block, the remaining bytes and the pointer should be set to 0x00.

4. free block

Free blocks are empty and available for writing. But just as any other block, they have to have the required bytes 0 and 1 (see below). Once again, you have many options for managing your free block list. For example, you may choose to use the link field (found at byte 2) to form a chain of free blocks.

A.3.2 Block format

The following bytes are defined for all blocks: Bytes 0 and 1 must be formatted as specified. A suggestion for bytes 2 and 3 has been made for you, but is optional. I.e. the bytes beyond byte 1 are up to you to implement however you see fit.

Byte	first byte offset	second byte offset
0	[block type = 1,2,3,4,....]	0x44
2	[address of another block]	[empty]
4	[data starts]	...
6	...	...

A.3.3 TinyFS interface functions:

Nine API functions are needed to implement the TinyFS interface.

```
/* Makes a blank TinyFS file system of size nBytes on the unix file specified
by ‘filename. This function should use the emulated disk library to open
the specified unix file, and upon success, format the file to be mountable
disk.  This includes initializing all data to 0x00, setting magic numbers,
initializing and writing the superblock and inodes, etc.  Must return a
specified success/error code.  */
```

int tfs_mkfs(char *filename, int nBytes);

```
/* tfs_mount(char *diskname) ‘‘mounts a TinyFS file system located within
diskname unix file.  tfs_unmount(void) ‘‘unmounts the currently mounted
file system.  As part of the mount operation, tfs_mount should verify the
file system is the correct type.  Only one file system may be mounted at
```

a time. Use tfs_unmount to cleanly unmount the currently mounted file system. Must return a specified success/error code. */

int tfs_mount(char *diskname); int tfs_unmount(void);

/* Creates or Opens an existing file for reading and writing on the currently mounted file system. Creates a dynamic resource table entry for the file, and returns a file descriptor (integer) that can be used to reference this file while the filesystem is mounted. */

fileDescriptor tfs_openFile(char *name);

/* Closes the file, de-allocates all system/disk resources, and removes table entry */

int tfs_closeFile(fileDescriptor FD);

/* Writes buffer ‘buffer of size ‘size, which represents an entire files content, to the file system. Previous content (if any) will be completely lost. Sets the file pointer to 0 (the start of file) when done. Returns success/error codes. */

int tfs_writeFile(fileDescriptor FD,char *buffer, int size);

/* deletes a file and marks its blocks as free on disk. */

int tfs_deleteFile(fileDescriptor FD);

/* reads one byte from the file and copies it to buffer, using the current file pointer location and incrementing it by one upon success. If the file pointer is already at the end of the file then tfs_readByte() should return an error and not increment the file pointer. */

int tfs_readByte(fileDescriptor FD, char *buffer);

`/* change the file pointer location to offset (absolute). Returns success/error codes.*/`

int tfs_seek(fileDescriptor FD, int offset);

In your tinyFS.h file, you must also include the following definitions:

`/* The default size of the disk and file system block */`

#define BLOCKSIZE 256

`/* Your program should use a 10240 Byte disk size giving you 40 blocks total. This is a default size. You must be able to support different possible values */`

#define DEFAULT_DISK_SIZE 10240

`/* use this name for a default disk file name */`

#define DEFAULT_DISK_NAME "tinyFSDisk

typedef int fileDescriptor;

A.3.4 Error Codes

You must specify a set of unified error codes returned by your TinyFS interfaces. textbfAll error codes must be negative integers (-1 or lower), but it is up to you to assign specific meaning to each. Error codes must be informational only, and not used as status in subsequent conditionals. Create a file called `tinyFS_errno.h` and implement the codes as a set of statically defined macros. Take a look at `man 3`

`errno` on the UNIX* machines for examples of the types of errors you might catch and report.

A.4 Assignment & Additional Features

- Implement the core interface functions above (80%).
- Add two additional areas of functionality from the list (a-h) below (20%). Note that some features count as two. You are free to implement the features in your own way, so be creative, but feel free to do a little research, and base your design decisions on existing solutions.

  1. Fragmentation info and defragmentation (10%)
     - 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 */
  2. Directory listing and renaming (10%)
     - `tfs_rename(fileDescriptor FD, char* newName)` /* renames a file. New name should be passed in. File has to be open. */
     - `tfs_readdir()` /* lists all the files and directories on the disk, print the list to stdout */
  3. Hierarchical directories (20%)
     - Support hierarchical directories by creating a root directory link in the superblock and re-designing the inode block, so that in can indicate a

directory (same name requirements as a file). Use absolute paths for all files and directories.

  * discuss your design and implementation

- `tfs_createDir(char *dirName)` /* creates a directory, name could contain a / delimited path) */
- `tfs_removeDir(char *dirName)` /* deletes empty directory */
- `tfs_removeAll(char *dirName)` /* recursively remove dirName and any file and directories under it. Special / token may be used to indicate root dir. */
- `tfs_openFile()`: modify so that it supports directories as part of the file name. Return an error if any directory in the path does not exist. It must maintain backwards compatibility.

4. Read-only and writeByte support (10%)

   - 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).

5. Timestamps (10%)

   - implement creation, modification and access timestamps for each file to be stored in the inode block

- `tfs_readFileInfo(fileDescriptor FD)` /* returns the files creation time or all info (up to you if you want to make multiple functions) */
- return format is up to you

6. Implement content-based address (20%)
   - Instead of addressing data blocks by their offsets, address them by their content
   - In this way, identical blocks will be shared between files, reducing the total number of data block necessary

7. Implement full-disk encryption (20%)
   - All data and metadata should be encrypted using a semantically secure block cipher in a sensible mode of operation (e.g. CTR, CBC, or better XTS). This page on disk encryption theory may be helpful.
   - You may (and should) use an pre-existing cryptography library, such as OpenSSL for your core cryptographic operations.
   - The key and data in memory may be unencrypted, but at rest data (data on disk) should always be encrypted.
   - The key used to encrypt/decrypt data should be derived from a password given when you format the file system; you must use a secure key derivation function (e.g. PBKDFv2, scrypt or bcrypt).
   - Modify `tfs_mkfs()` to accept a password, and format and encrypt an initial TinyFS image.
   - `tfs_mount()` should also be modified to take in the users password, unlocking the file system; all other TinyFS interfaces should remain unaltered.
   - Discuss your design decisions and threat model (i.e. what attacks your system is strong and weak against.)

8. Implement file system consistency checks (10%)
    - Upon mount, verify the entire file system is in a consistent state, and fail to mount and report an error if it is not.
    - An inconsistent file system might include:
        * blocks on both the free list and allocated to an inode
        * data blocks that are not on the free list of allocated to an inode
        * blocks that have been corrupted due to latent disk failures
        * Other scenarios.
    - See this set of lecture notes for more information on file system consistency
    - Discuss the types of inconsistency your file system detects.

- Write a demo program that includes your TinyFS interface to demonstrate the basic functionality of the required functions and your chosen additional functionality. You can display informative messages to the screen for the user to see how you demonstrate these.
- Implementation: This program must be implemented in C.
- EXTRA CREDIT: Extra credit points, which will count towards your total programming assignment score, may be awarded for completing up to two additional features. Each single feature will contribute up to 10 points, with those features that count as 2, up to 20 points. No more than 20 points, in total, will be awarded for extra credit.

A.5 Deliverables

As usual, submit a tar.gz archive via PolyLearn with the following:

- all source files: .c, .h (at least three separate source files)
  - emulator file (libDisk)
  - tinyFS interface file (libTinyFS). This file will access libDisk for disk emulator functionality
  - tinyFsDemo driver file that contains a main(), and includes libTinyFS headers (but not libDisk)
- A Makefile that compiles all the libraries and makes the following executable:
  - tinyFsDemo
- a README with:
  - Names of all partners
  - 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.
  - Any limitations or bugs your file system has.

Appendix B

Before Survey

This survey was given to Professor Foaad Khosmood's Winter 2016 CPE 453 class. It is intended for students who have already completed the TinyFS assignment but did not used the PolyFS Visualizer.

B.1 TinyFS File System and Emulator Project

Introduction. You are invited to participate in a research study entitled "Tiny FS. The purpose of this study is to investigate how TinyFS help teach file system concepts.

This research project is being conducted by the following investigators:

* Paul Fallon, MS student, Computer Science, Cal Poly * Foaad Khosmood, Assistant Professor of Computer Science, Cal Poly

Activity. You are being asked play a game related to the study and then provide feedback through an anonymous survey. The surveys will ask questions about your background and opinions related to the game, and to the subject matter. Participation in the surveys will likely involve between 2-5 minutes. None of the activities are strenuous; indeed, they are intended to be engaging and fun. Nevertheless, you may withdraw at any point in the survey, or only answer those questions that interest you, or withdraw from any other portion of the research activity, without penalty.

Location. The activity will occur online. Cost. There is no cost to participate in this study. Compensation. No compensation will be provided for your participation.

Voluntary Nature of Study. Your participation in this study is strictly voluntary. Your grades will not be affected by your participation or lack thereof. If you choose

to participate, you can still change your mind at any time and withdraw from the study. If you choose not to participate in this study or to withdraw, you will not be penalized in any way or lose any other entitled benefits. You do not have to answer any question you choose not to answer.

Potential Risks or Discomforts. There are no risks anticipated with your participation in this study. Only limited identifying data will be collected during the study, so even in the unlikely event of data mismanagement (i.e., unintended disclosure of study data) there is no clear harm anticipated.

Anticipated Benefits. Anticipated benefits from this study are improvements to educational programs here at Cal Poly and beyond.

Confidentiality & Privacy Act. Any information that is obtained during this study will be kept confidential to the full extent permitted by law. Any collected materials that carry your name (like this one) will be held in an offline, physically secure archive (access to which is strictly controlled). Research results will use only summary and anonymized data. Quoted responses will only ever be anonymous (i.e., "one student observed...). You will not be mentioned by name by this study. The results of your participation will be confidential.

Points of Contact. If you have any questions or comments about the research, or have questions about any discomforts that you experience while taking part in this study please contact the Principal Investigator, Foaad Khosmood, foaad@calpoly.edu. If you have concerns regarding the manner in which the study is conducted, you may contact Dr. Steve Davis, Chair of the Cal Poly Human Subjects Committee, at (805) 756-2754, sdavis@calpoly.edu, or Dr. Dean Wendt, Dean of Research, at (805) 756-1508, dwendt@calpoly.edu.

Statement of Consent. If you agree to voluntarily participate in this research project as described, please indicate your agreement by selecting I volunteer and

completing the online survey. Please print a copy of this document now and retain for your reference.

Online Consent. You may be shown this informed consent form in an online form prior to answering survey questions. You can indicate your acceptance by continuing on to the survey. If you do not agree, we ask that you stop immediately and not further continue with the survey.

B.1.1 Do you agree with the above statement?

Yes or no

B.1.2 Your name

short answer

B.2 Short Answer Questions

These questions are about your final assignment: TinyFS file system and emulator

Name some of the challenges you faced with this project.

How did you verify your file system was storing data correctly?

What tools did you use to help you with this project?

How could this project be improved?

B.3 Grid Questions

Indicate your level of agreement

	strongly disagree	disagree	neutral	agree	strongly agree
This project was difficult					
This project was helpful in understanding file systems					
I had difficulty visualizing the content of the disk					
It was hard to design the superblock.					

Appendix C

After Survey

This survey was given to Professor Foaad Khosmood and Professor Zachary Peterson's Spring 2016 CPE 453 classes. It is intended for students who have started the TinyFS assignment and are using the PolyFS Visualizer.

In order to allow for comparison this survey is exactly the same as the before survey with the exception of the last two questions.

C.1 TinyFS File System and Emulator Project

Introduction. You are invited to participate in a research study entitled "Tiny FS. The purpose of this study is to investigate how TinyFS help teach file system concepts.

This research project is being conducted by the following investigators:

* Paul Fallon, MS student, Computer Science, Cal Poly * Foaad Khosmood, Assistant Professor of Computer Science, Cal Poly

Activity. You are being asked play a game related to the study and then provide feedback through an anonymous survey. The surveys will ask questions about your background and opinions related to the game, and to the subject matter. Participation in the surveys will likely involve between 2-5 minutes. None of the activities are strenuous; indeed, they are intended to be engaging and fun. Nevertheless, you may withdraw at any point in the survey, or only answer those questions that interest you, or withdraw from any other portion of the research activity, without penalty.

Location. The activity will occur online. Cost. There is no cost to participate in this study. No compensation will be provided for your participation.

Voluntary Nature of Study. Your participation in this study is strictly voluntary. Your grades will not be affected by your participation or lack thereof. If you choose to participate, you can still change your mind at any time and withdraw from the study. If you choose not to participate in this study or to withdraw, you will not be penalized in any way or lose any other entitled benefits. You do not have to answer any question you choose not to answer.

Potential Risks or Discomforts. There are no risks anticipated with your participation in this study. Only limited identifying data will be collected during the study, so even in the unlikely event of data mismanagement (i.e., unintended disclosure of study data) there is no clear harm anticipated.

Anticipated Benefits. Anticipated benefits from this study are improvements to educational programs here at Cal Poly and beyond.

Confidentiality & Privacy Act. Any information that is obtained during this study will be kept confidential to the full extent permitted by law. Any collected materials that carry your name (like this one) will be held in an offline, physically secure archive (access to which is strictly controlled). Research results will use only summary and anonymized data. Quoted responses will only ever be anonymous (i.e., “one student observed...). You will not be mentioned by name by this study. The results of your participation will be confidential.

Points of Contact. If you have any questions or comments about the research, or have questions about any discomforts that you experience while taking part in this study please contact the Principal Investigator, Foaad Khosmood, foaad@calpoly.edu. If you have concerns regarding the manner in which the study is conducted, you may contact Dr. Steve Davis, Chair of the Cal Poly Human Subjects Committee, at (805) 756-2754, sdavis@calpoly.edu, or Dr. Dean Wendt, Dean of Research, at (805) 756-1508, dwendt@calpoly.edu.

Statement of Consent. If you agree to voluntarily participate in this research project as described, please indicate your agreement by selecting I volunteer and completing the online survey. Please print a copy of this document now and retain for your reference.

Online Consent. You may be shown this informed consent form in an online form prior to answering survey questions. You can indicate your acceptance by continuing on to the survey. If you do not agree, we ask that you stop immediately and not further continue with the survey.

C.1.1 Do you agree with the above statement?

Yes or no

C.1.2 Your name

short answer

C.2 Short Answer Questions

These questions are about your final assignment: TinyFS file system and emulator

Name some of the challenges you faced with this project.

How did you verify your file system was storing data correctly?

What tools did you use to help you with this project?

How could this project be improved?

C.3 Grid Questions

Indicate your level of agreement

	strongly disagree	disagree	neutral	agree	strongly agree
This project was difficult					
This project was helpful in understanding file systems					
I had difficulty visualizing the content of the disk					
It was hard to design the superblock.					

C.4 Visualizer Responses

These questions are about your usage of the PolyFS Visualizer

Did you use the PolyFS Visualizer for this project?

yes or no

Is there anything you would like to change about the PolyFS Visualizer?

short answer

Did you use the PolyFS Visualizer for this project?

yes or no

Did you have any difficulty using the PolyFS Visualizer?

short answer

Is there anything you would like to change about the PolyFS Visualizer?

short answer

Any comments or suggestion concerning the PolyFS Visualizer?

short answer

Appendix D

Sample Code

A couple sections of the code have been highlighted here to highlight some of the techniques used in this project as well as some of the challenges faced and problems solved.

D.1 Block Modal Links

The requirements of the block modal were to include click-able links for the "link" block metadata type. This required retrieving the block from the canvas click, closing the current modal and opening a new block modal for the correct block.

The first step was getting the value of the link from a click. The below code segment is the click call back function for then the canvas receives a click event. Jcanvas passes into all events the name of the specific object that was clicked as an identifier. Therefore, the hex and char representations of the bytes within the block modal were drawn onto Jcanvas named according to their index and a pre-pended "h" for hex or "c" for char. This made it possible to identify the exact byte clicked and its formate (2 ascii symbols per hex byte versus 1 ascii symbol per char byte). From the name the value of the link that was clicked could be retrieved (see lines 3 - 5 in the code below).

The second step was acting on a click by closing the current block modal and populating a new one with the correct data. Hiding the current modal is easy. However, telling the modal to reopen with the correct data as more difficult. The final tactic used was to define a hidden field within the modal to store the block to show. Thus, the callback could simply set the hidden field to the value of the link and tell open

the modal (see line 11). The draw handlers for the modal would read the hidden field and load the correct data.

The final problem present in following links within the block modal is showing the user that the link is being followed. If the new modal pops up immediately then it is not immediately apparent that the data has changed and can get very confusing. Using the hidden field method of selecting data for display it is possible to completely separate the what and when properties of the modal. Therefore, a timeout function could be used to open the new modal (line 12) with a suitable delay after the previous one had been closed (line 6).

```
1   show_block_callback: function(block, byte_num){
2      return function(layer){
3         var number = layer.name.substring(1);
4         var byte_num = layer.name.substring(0, 1) == "h" ? Math.
             floor(number/2) : number;
5         var link = parseInt(block.get_byte(byte_num));
6         $('#blockModal').modal('hide');
7         if(link < 0 || link >= polyFS_visualizer.data.blocks.length
             ){
8            warning("invalid link");
9            return;
10        }
11        $("#block-number").val(link);
12        setTimeout(function(){ $('#blockModal').modal('show'); },
             600);
13     };
14  }
```

D.2 Block Metadata Per Type Handlers

The requirement for the block meta was to display a meaningful message for each metadata element based on the data in the element. Thus for the "type" byte of the metadata the string name of the block type should be displayed next to the hex and char values in the block metadata panel.

However, this requires a different, custom action per metadata type. Additionally, since the metadata is configurable it could easily be the case that the metadata type did not exist when the PolyFS Visualizer was created and a default action should be taken instead. This represented a challenge of managing a wide range of handlers.

The following code shows the solution employed by the PolyFS Visualizer to solve the problem. Essentially it is a map of metadata byte types to the function handler for that type. Additionally, a default type exists that can be used if the more specific handler is missing. This has the advantage that all the handlers are in one place, clearly labeled and easily understood. Changing the handlers or adding new ones is very easy and can be done without changing or affecting the current code.

```
1   metadata_config: {
2       "type": function (block){
3               return ", " + polyFS_visualizer.config.get_data_config("
                    type", block.get_byte(polyFS_visualizer.config.
                    block_config.bytes.indexOf("type")));
4           },
5       "magic-number": function (block){
6               return ", " + polyFS_visualizer.config.get_data_config("
                    magic-number", block.get_byte(polyFS_visualizer.
                    config.block_config.bytes.indexOf("magic-number")));
7           },
8       "link": function (block){
9               var linked_block = polyFS_visualizer.data.blocks[block.
```

```
                get_byte(polyFS_visualizer.config.block_config.bytes.
                indexOf("link"))];
10          if(!linked_block){
11              return "";
12          }
13          return ", linking to a " + polyFS_visualizer.config.
                get_data_config("type", linked_block.get_byte(
                polyFS_visualizer.config.block_config.bytes.indexOf("
                type")));
14      },
15    "default": function (block){
16          return "";
17      },
18  }
```

In order to make use of the function map, the polyFS Visualizer attempts to get the handler for the type. If the handler does not exist then it uses the default.

The following code example shows a call to fetch the text metadata message for the byte from the correct handler. If the specific handler exists then the or statement will be short-circuited, the default handler will never be fetched and the specific handler will be used. On the other hand if the specific handler does not exist then the default handler will be called and used in its place.

```
1   Var custom_message = (polyFS_visualizer.block_meta.
        metadata_config[polyFS_visualizer.config.block_config.bytes[
        index]] || polyFS_visualizer.block_meta.metadata_config["
        default"])(current_block);
```

D.3 Proper Name-spacing

When testing the project with Google Chrome it was found that some of the visualizers variables conflicted with Google Chromes environment leading to the page loading improperly. In order to fix this the entire project was name-spaced under object polyFS_visualizer. All objects were included into the polyFS_visualizer namespace in order to guarantee browsers would not conflict with the visualizer.

Appendix E

Test code

This is the external test code included as an example of the test suites used. The internal tests and server tests were handled in very similar fashions with very similar executables.

```
1  #include <string.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  #include "libDisk.h"
6
7  #define BLOCK_SIZE 256
8  #define NUM_BLOCKS 50
9
10 // disk0: exists, normal operation
11 // disk1: does not exist
12 // disk2: exists, normal operation, writes
13 // disk3: exists, normal operation, reads
14
15 int run_open(char *filename, int nBytes, int expectedError);
16 void test_open();
17 void run_read(int disk, int bNum, void *block, int expectedError,
       char *expectedBlock);
18 void run_write(int disk, int bNum, void *block, int expectedError);
19 void test_write();
20 void test_read();
21
22 int main(){
23    test_open();
```

```
24     test_write();
25     test_read();
26 
27     printf("\nTest success!\n");
28 }
29 
30 int run_open(char *filename, int nBytes, int expectedError){
31     printf("running open disk ... ");
32     int fd = openDisk(filename, nBytes);
33     printf("%d\n", fd);
34 
35     // make sure error is as expected
36     if( (!expectedError && fd < 0) || (expectedError && fd !=
           expectedError) ){
37         printf("ERROR: expected %d but got %d\n", expectedError, fd);
38         exit(0);
39     }
40 
41     return fd;
42 }
43 
44 // int openDisk(char *filename, int nBytes){
45 void test_open(){
46 
47     printf("\n\nRUNNING OPEN TESTS\n");
48 
49     // create disk
50     printf("test 1:\n");
51     run_open("disk0", BLOCK_SIZE * NUM_BLOCKS, 0);
52     run_open("disk2", BLOCK_SIZE * NUM_BLOCKS, 0);
53     run_open("disk3", BLOCK_SIZE * NUM_BLOCKS, 0);
54     run_open("disk4", BLOCK_SIZE * NUM_BLOCKS, 0);
```

```
55
56     // open disk that does not exist, no create
57     printf("test 2:\n");
58     run_open("disk1", 0, -203);
59
60     // create disk that already exists
61     printf("test 3:\n");
62     run_open("disk0", BLOCK_SIZE * NUM_BLOCKS, -202);
63
64     // open disk that exists
65     printf("test 4:\n");
66     run_open("disk0", 0, 0);
67     run_open("disk2", 0, 0);
68     run_open("disk3", 0, 0);
69     run_open("disk4", 0, 0);
70
71     // disk with name too long
72     printf("test 5:\n");
73     run_open("
         disk1_this_name_is_much_too_long_as_you_can_tell_by_looking",
         BLOCK_SIZE * NUM_BLOCKS, -300);
74     run_open("
         disk1_this_name_is_much_too_long_as_you_can_tell_by_looking",
         0, -300);
75
76     // open disk with null inputs
77     printf("test 6:\n");
78     run_open(NULL, 0, -300);
79     run_open(NULL, BLOCK_SIZE * NUM_BLOCKS, -300);
80
81     // disk with empty string name
82     printf("test 7:\n");
```

```
83      run_open("", 0, -300);
84      run_open("", BLOCK_SIZE * NUM_BLOCKS, -300);
85
86      // open disk with negative nBytes
87      printf("test 8:\n");
88      run_open("disk1", -1, -205);
89
90      // open disk with less than a block of bytes
91      printf("test 9:\n");
92      run_open("disk1", 1, -203);
93
94      // try insecure disk names
95      printf("test 10:\n");
96      run_open("../disk0", BLOCK_SIZE * NUM_BLOCKS, -300);
97      run_open("<body onload=console.log('XSS ATTACK SUCCESSFUL')>",
            BLOCK_SIZE * NUM_BLOCKS, -300);
98      run_open("hi, 1, 1); drop table open_disks;", BLOCK_SIZE *
            NUM_BLOCKS, -300);
99
100
101 }
102
103 void run_read(int disk, int bNum, void *block, int expectedError,
        char *expectedBlock){
104
105     printf("running read disk ... ");
106     int error = readBlock(disk, bNum, block);
107     printf("%d\n", error);
108
109     // make sure error is as expected
110     if( (expectedError >= 0 && error < 0) || (expectedError && error
            != expectedError) ){
```

```
111         printf("ERROR: expected %d but got %d\n", expectedError, error
                );
112         exit(0);
113     }
114 
115     if (expectedError >= 0) {
116 
117         // check buffer against expected result
118         if(memcmp(block, expectedBlock, BLOCK_SIZE) != 0){
119             printf("ERROR: read wrong data!\n Expected:\n\n%s\n\n
                    Received:\n\n%s\n\n", (char *)block, expectedBlock);
120             exit(0);
121         }
122     }
123 }
124 
125 void run_write(int disk, int bNum, void *block, int expectedError){
126     char *test_block[BLOCK_SIZE];
127 
128     printf("running write disk ... ");
129     int error = writeBlock(disk, bNum, block);
130     printf("%d\n", error);
131 
132     // make sure error is as expected
133     if( (expectedError >= 0 && error < 0) || (expectedError && error
            != expectedError) ){
134         printf("ERROR: expected %d but got %d\n", expectedError, error
                );
135         exit(0);
136     }
137 
138     // do a test read on the block to make sure it is good
```

```
139     if(expectedError >= 0){
140         run_read(disk, bNum, &test_block, 0, block);
141     }
142 }
143 
144 // int readBlock(int disk, int bNum, void *block){
145 void test_read(){
146     char block0[BLOCK_SIZE];
147     char block1[BLOCK_SIZE];
148     char block2[BLOCK_SIZE];
149     char read1[BLOCK_SIZE];
150 
151     printf("\n\nRUNNING READ TESTS\n");
152 
153     // open up a disk for the test
154     int fd3 = run_open("disk3", 0, 0);
155     printf("fd3: %d\n", fd3);
156 
157     // create a couple blocks for the test
158     memset(block0, '$', BLOCK_SIZE);
159     memset(block1, 0x01, BLOCK_SIZE);
160     memset(block2, 0, BLOCK_SIZE);
161 
162     // write a block for correct tests
163     // since run_write checks write correctness with reads this also
            test standard read cases
164     printf("test 1:\n");
165     run_write(fd3, 0, block0, 0);
166     run_write(fd3, 5, block0, 0);
167     run_write(fd3, 40, block1, 0);
168     run_write(fd3, 49, block1, 0);
169 
```

```
170     // read block before it is ever written
171     printf("test 2:\n");
172     run_read(fd3, 1, read1, 0, block2);
173 
174     // read block with negative block number
175     printf("test 3:\n");
176     run_read(fd3, -1, read1, -200, block0);
177 
178     // read block with too large block number
179     printf("test 4:\n");
180     run_read(fd3, 100, read1, -200, block0);
181 
182     // read block with invalid disk number
183     printf("test 5:\n");
184     run_read(-100, 5, read1, -206, block0);
185 
186     // read block with null values
187     printf("test 6:\n");
188     run_read(fd3, 5, NULL, -207, block0);
189 }
190 
191 // int writeBlock(int disk, int bNum, void *block){
192 void test_write(){
193     char block0[BLOCK_SIZE];
194     char block1[BLOCK_SIZE];
195     char block2[BLOCK_SIZE/2];
196     char block3[BLOCK_SIZE*2];
197 
198     printf("\n\nRUNNING WRITE TESTS\n");
199 
200     // open up a disk for the test
201     int fd2 = run_open("disk2", 0, 0);
```

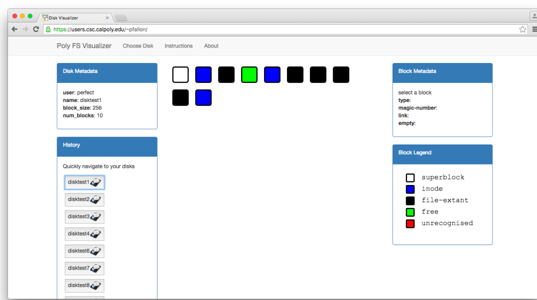
```
202    printf("fd2: %d\n", fd2);
203
204    // create a couple blocks for the test
205    memset(block0, '$', BLOCK_SIZE);
206    memset(block1, 0x01, BLOCK_SIZE);
207    memset(block2, 'a', BLOCK_SIZE/2);
208    memset(block3, 'a', BLOCK_SIZE*2);
209
210    // write block
211    printf("test 1:\n");
212    printf("HERE 1\n");
213    run_write(fd2, 0, block0, 0);
214    printf("HERE 2\n");
215    run_write(fd2, 5, block0, 0);
216    printf("HERE 3\n");
217    run_write(fd2, 40, block1, 0);
218    printf("HERE 4\n");
219    run_write(fd2, 49, block1, 0);
220    printf("HERE 5\n");
221
222    // write block with negative block number
223    printf("test 2:\n");
224    run_write(fd2, -1, block1, -200);
225
226    // write block with too large block number
227    printf("test 3:\n");
228    run_write(fd2, 100, block1, -200);
229
230    // write block with invalid disk number
231    printf("test 4:\n");
232    run_write(-100, 5, block1, -206);
233
```

```
234     // write block with null values
235     printf("test 5:\n");
236     run_write(fd2, 5, NULL, -207);
237 
238     // write block with block size too large
239     printf("test 6:\n");
240     run_write(fd2, 7, block3, 0);
241 
242     // write block with block size too small
243     printf("test 7:\n");
244     run_write(fd2, 15, block2, 0);
245 
246 }
```

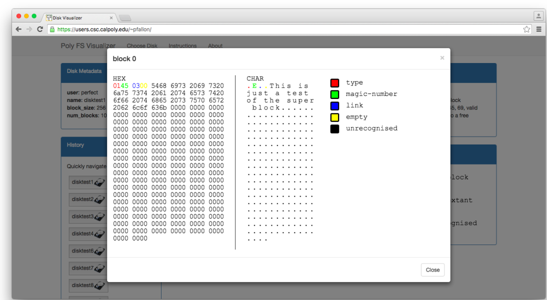
Appendix F

Test code

These images show the outcome of the disktests test suite. The suite is described in detail in the validation section.



(a) main page



(b) first block

Figure F.1: Disk Test 1: perfect disk

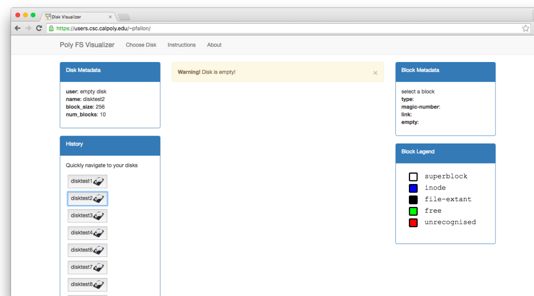
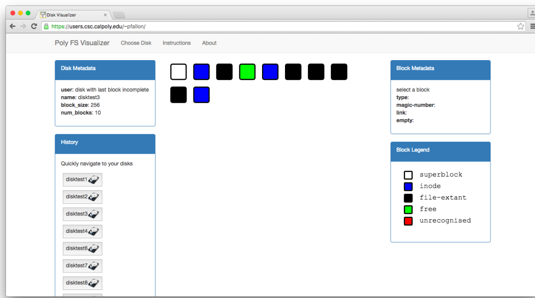
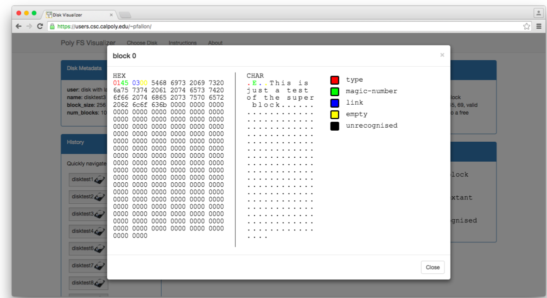


Figure F.2: Disk Test 2: empty disk

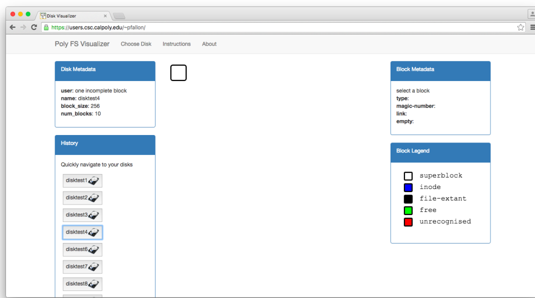


(a) main page

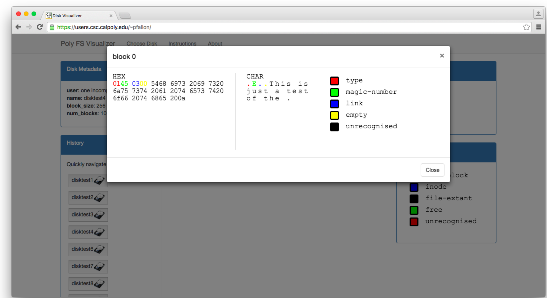


(b) first block

Figure F.3: Disk Test 3: disk with last block incomplete



(a) main page



(b) first block

Figure F.4: Disk Test 4: One incomplete block

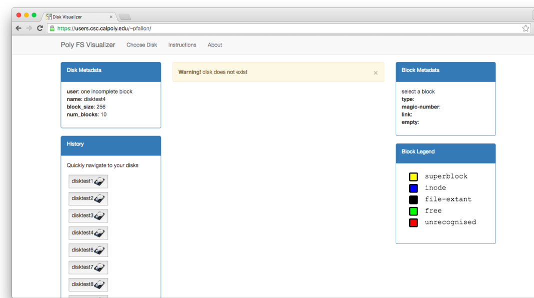
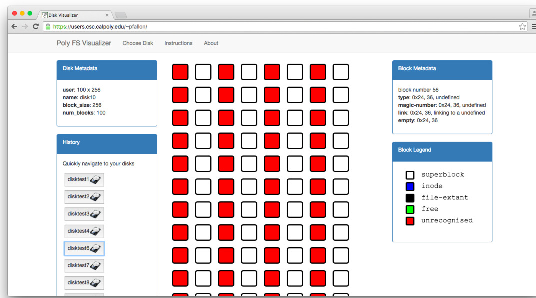
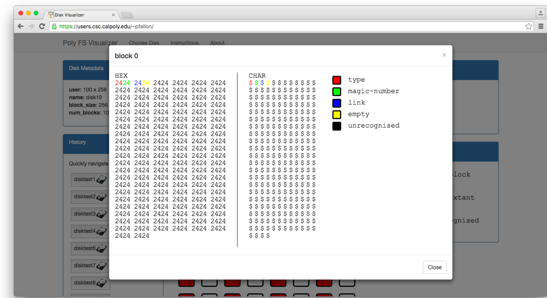


Figure F.5: Disk Test 2: disk missing

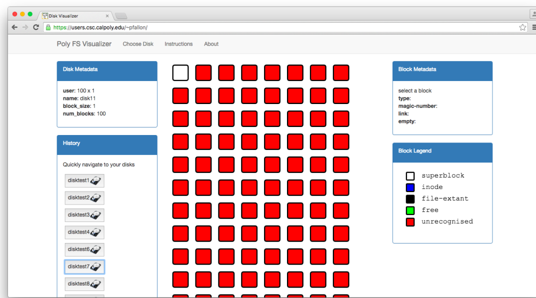


(a) main page

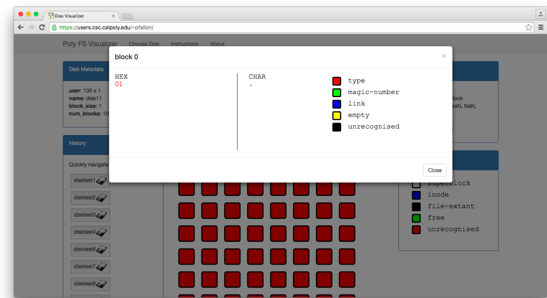


(b) first block

Figure F.6: Disk Test 6: 100 blocks x 256 bytes

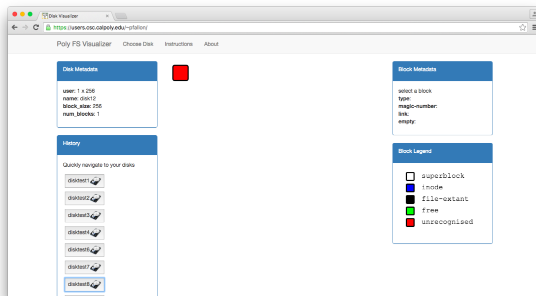


(a) main page

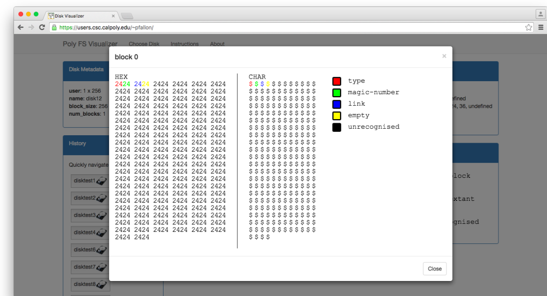


(b) first block

Figure F.7: Disk Test 7: 100 blocks x 1 byte

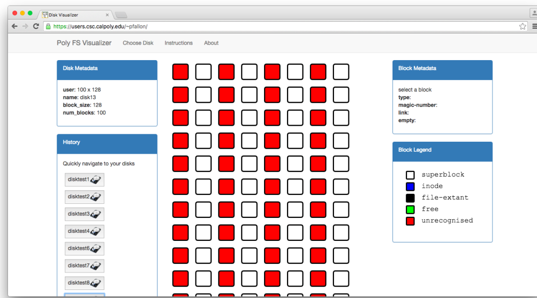


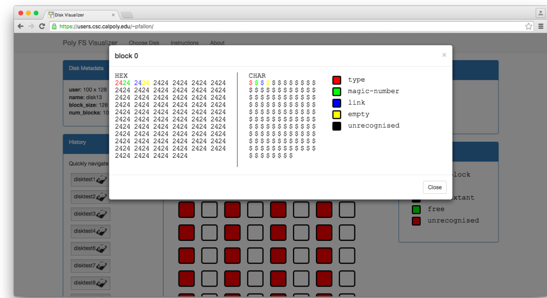
(a) main page



(b) first block

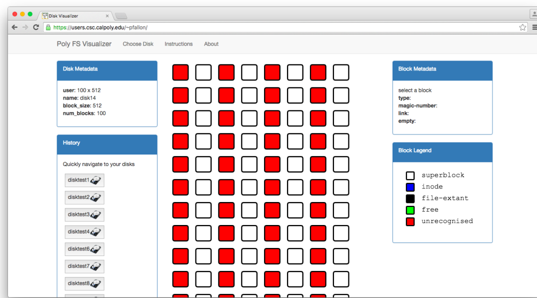
Figure F.8: Disk Test 8: 1 block x 256 bytes

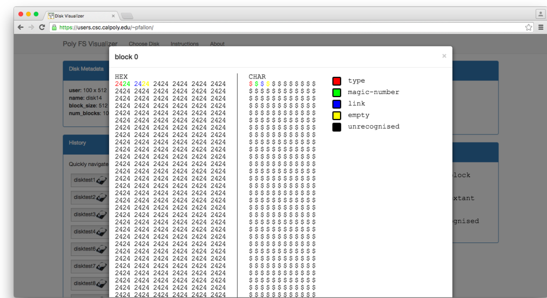




(a) main page (b) first block

Figure F.9: Disk Test 9: 100 blocks x 128 bytes





(a) main page (b) first block

Figure F.10: Disk Test 10: 100 blocks x 512 bytes