SYSTEM DOCUMENTATION

for

Arboreal

Version 1.0

Prepared by:
Adrian Barberis, *Wyatt Emery*, *Danny Radosevich*

April 30, 2018

Contents

1 Introduction and Description

1.1 Introduction

Welcome to Arboreal, a next generation file system solution. This document functions as a general overview of key features offered by Arboreal,a brief discussion about its components and restrictions, and a discussion of its efficiency illustrated via statistical analyses and time complexities. For Arboreal's User Guide please see, the document titled *Arboreal User Guide*. For complete source code documentation see the document titled *Arboreal Reference Manual*. For your convenience, a copy of both*Arboreal User Guide* and *Arboreal Reference Manual* are reproduced here under Sections 6 and 7 respectively.

1.2 Purpose

The purpose of this product is to design and implement a file-system with modern computing technology in mind. Current standard file-system implementations are becoming ever more archaic and lack innovations which take advantage of the new computing power available due to advancements in technology. *Arboreal* is a tag based, tree structured file-system which will utilize the full potential of today's computers in order to create a more efficient, flexible, and powerful file-system.

Current file-systems limit users to a set of linearly iterable directories reducing flexibility in file grouping to, effectively, a set of rigidly defined bins (similar to a filing cabinet). *Arboreal*'s free-form binary tree design allows fluid file grouping letting you search for a category and then categories within that category and so on by foregoing the traditional directory structure in favor of file tags. Additionally, due to its use of binary search trees as the backbone data structures (in contrast to the lists of todays file-systems) we are able to enormously speed up the process of finding files. This creates a far more efficient system. All of this is in service of creating a system that frees modern computers and users from the rigid boundaries of the past.

1.3 Intended Audience and Reading Suggestions

This document is intended for those who have some knowledge regarding computer file-systems as well as those interested in a new generation of file-system design. However, many concepts are targeted for readers of all experience levels. Technical jargon and complexity will be kept to a minimum in order to give any reader a solid understanding

our product but, the authors suggest that a cursory understanding of binary trees (especially binary-search trees) and the functions and components of standard file-systems such as FAT, EXT or NTFS would be beneficial.

1.4 Project Scope

Arboreal will use tags to change the way files are stored on a computer. Static directories will be eliminated in favor of *Dynamic Directories*. The use of a balanced binary search tree to store this data will help *Arboreal* outperform other file-systems in terms of search speed. As it stands, *Arboreal* is a virtual file-system, to display its ability and usefulness. However, *Arboreal* **Version 2.0** will be integrated with *FUSE* to allow better interoperability with existing operating systems. There is a Command Line Interface (CLI) and a Graphical User Interface (GUI) by which the user can interact with the file-system. Later versions of *Arboreal* may implement features such as "on the fly" encryption, and use of AI to improve performance and increase the file-system's available feature set.

1.5 Product Perspective

Arboreal is a file-system designed with modern computer architecture in mind. The traditional directory structure of file-systems invented at the beginning of the computer era is still being used and has been greatly improved. However, the underlying design has remained unchanged for over 50 years. *Arboreal* is aims to replace the dominant file-system directory based structure. Directory based file-systems were modeled after the real world. In reality, you may have a filing cabinet; That cabinet has drawers; Those drawers have folders and files inside. You may even have folders within folders. Computer file-systems need not model reality in this way. Indeed, modern computers should be striving to create more efficient digital versions of real-world concepts. For file systems this should take the shape of dynamic directories which allow you to modify, reselect, and filter file groupings on the fly.

Arboreal aims to change to the traditional directory based file system, while still providing the functionality consumers have come to expect. It includes a standard GUI for convenient use by the general user (student, businessman, employee, etc.) as well as a CLI for more use by technicians, information technology specialists, computer scientists, engineers, etc. Users will be able to perform all of the functions normally available in a file-system: creating, deleting, renaming, and opening files, changing groupings, creating new groupings, deleting groupings, moving individual files to new or existing groupings, searching for specific files, searching for files based on one or more tags, adding tags to files post file creation, sorting files based on name or date, and copy/pasting files.

1.6 Product Design Specifications

Arboreal is based around two balanced binary search tree data structures. One will hold references to *tags* (let this be the *root tree*). Each *tag* will itself be a balanced binary

search tree and will contain references to *file information nodes*. These nodes will remain unchanged from the standard structure in use today and are tasked with keeping track of essential file information such as: size, tags, number of tags, location of data in storage, and so on. *Arboreal* **Version 1.0** loads all tag trees into memory on startup. This is not the most efficient way to handle as results in a slow startup time.

However, *Arboreal* **Version 2.0** will work as follows. When a certain *tag* is in use its tree is loaded into working memory and functions as the *current working directory* providing the user with fast and efficient file operations. If a user has worked with multiple tags and therefore multiple tag trees are in working memory, the newest will be considered as being the *current working directory*. All trees that have been pulled into memory will remain in memory until a request for memory exceeds a certain cap. At this point, a number of trees equal to the memory requested will be deleted from memory starting from the oldest tree and moving up. The *root tree* will remain in memory as long as the file-system is in use and will periodically be saved to disk. All together this will provide for an efficient and flexible file system. *Arboreal* implements a journaled approach to *tag tree* and *root tree* edits. This allows for changes in tag trees to be written out to disk with minimal overhead.

1.7 Operating Environment

Arboreal **Version 1.0**, is full file system running on top of currently existing file systems. This means it can only be considered a full filesystem in the case of formatting an external disk. It functions as a virtual filesystem from the Operating System's perspective. *Arboreal* **Version 2.0** will be implemented using FUSE, allowing full operating system integration.

1.8 Design and Implementation Constraints

Arboreal **Version 1.0** was begun and completed over the course of two semesters, about 7 months. The filesystem itself is implemented in C++ using STL functionality. The front-end GUI is implemented in Electron. *Arboreal* **Version 2.0** will be implemented in *Rust* for better memory management and threading capabilities.

1.9 User Documentation

The User Documentation includes this document as well as *Arboreal User Guide* and *Arboreal Reference Manual* contained at the end of this document.

1.10 User Interfaces

Arboreal **Version 1.0** consists of two distinct user interfaces. The first of these is the traditional GUI or Graphical User Interface. The GUI is designed to provide multiple

different user classes with a quick and intuitive way of interacting with the filesystem. The GUI will act as a mediator between the CLI, or Command Line Interface, and the user. Its primary focus will be to translate a user's action into a function call via the CLI and subsequently display the information returned. It will feature a fraction of the commands available via a CLI; that is, only those with which a general user need concern themselves with such as moving files, copying/pasting files, opening files, creating/editing/deleting file groupings, and renaming/re-tagging/deleting files. Since *Arboreal* **Version 2.0** will be implemented with *FUSE*, already existing desktop environments will be available to use. Therefore, neither a standalone CLI or GUI will be included with **Version 2.0**.

The other available interface (for **Version 1.0**) consists of a Command Line Interface. This interface is designed to work directly with the operating system's *system calls* and will be primarily used by the more technically skilled user base. It will have far more commands and functionality available but will present this through a text only interface. These commands will have to be inputed manually and must be known before hand by the user.

2 Features

2.1 Emulated Directory Structure & File System Flexibility

Arboreal does not use traditional directories as do today's file systems. The archaic concept of folders and hierarchical directories does not accurately mirror the fluidity of file categorization that many users need and want. While this is not necessarily bad it is slow and as user data and customization grows, it will be harder and harder to maintain an accurate representation of data associations as well as fast search/operation times by using the current system. Arboreal uses user defined tags to organize files. The tags are structured as binary search trees with each node pointing to a file on disk and the root node storing important information about the tag. These tags give the appearance of a structured and hierarchical directory system but in reality they are the complete opposite. For example, if one were to tag a file with *tag1,tag2,tag3* this would be the exact same as tagging a file with *tag3,tag1,tag2* or any other combination of those three tags. The fact that the tags are still structured as "folders" however, has the advantage of allowing users to maintain common syntax standards such as file paths as well as making integration with the Kernel more natural. It is important to note that if a file is tagged with multiple tags, that file is not duplicated in each tag, rather each tag will point to the same file on disk. References to the tags themselves, are stored in a n overarching binary search tree known as the *Root* tree. This tree makes tag lookup much faster as well as releasing the tags themselves from any further conformity to rigid hierarchical structures.

Due to this tree structure, Arboreal is a very flexible file system allowing not only creation of tags but searching by tags, tag combinations, and logical operations on tags. The user is never required to memorize long pathnames or hierarchies and can retrieve any file by simply remembering its name or at least some of the tags associated with it.

2.2 Disk Block Structure

Disk Super Block
Number of Partitions
Partition Info Repeating
(Partition Name, Partition Size, Start Block, Max File Name Size) ...
Partition Super Block
Free List Start Block
Free List End Block
Root Tree Super Block
Root Name
Root Entry Start Block Number
Root Tree Entries Block 1
(Default, Blocknumber),
(Tag Name, Blocknumber)...
Default Tag Tree Super Block
Default Tag Tree Entries Block 1
(File Name, Finode Block Number)...

Tag Tree Super Block Structure
Size
LastEntry(Blocknumber, Offset)
Start Block

Finode Block Structure
File Name
Attributes Block Number
12 Direct Blocks
Level 1 Indirect Block
Level 2 Indirect Block
Level 3 Indirect Block
(Tags)...
Continuation Block for Tags

3 Functionality

Arboreal currently supports a handful of core file system commands, namely:

- Creating Files
- Creating Tags
- Tagging Files
- Un-tagging Files
- Deleting Files
- Deleting Tags
- Opening Files
- Closing Files
- Reading From Files
- Writing To Files
- Seeking In Files
- Getting File Attributes
- Renaming Tags
- Renaming Files
- Searching For Files By Tag
- Searching For Files By Name

It is important to note that although the arboreal backend supports the above actions, the Arboreal command line interface may not as of yet support some of these commands. At present, the Arboreal command line interface supports the following commands:

- Creating Files
- Creating Tags
- Deleting Files
- Deleting Tags
- Renaming Files
- Renaming Tags
- Opening Files
- Closing Files
- Searching For Files By Tag
- Searching For Files By Name
- Changing Working Directory
- Listing All Tags
- Retrieving File Attributes
- Command Syntax Help

•Tagging Files
•Un-tagging Files

(A lot of these commands have a separate version for when a user is within a working directory acting on a files versus when the user wishes to act on a file from anywhere in the system. For a more comprehensive discussion of the commands available via the command line interface please see the Arboreal User Guide in Section 7).

3.1 Future Functionality

We hope to implement the following functionalities in future versions of Arboreal:

•Merging Tags
•Regular Expression File Search
•Partial Match File Search (In Real Time)
•AI Auto Tagging
•Native Encryption
•Color Coded Tags
•AI Filtering Of Files Based On Content
•Searching For Files By Content
•Copying And Pasting Files
•Automatic Renaming Of Duplicate Files
•Copying And Pasting Tags
•Expanded Tag Search Options
•Ghost Directories/File
•Hidden Directories/Files
•Integration With The Kernel
•Operating System Independence
•Creating Files
•And More

4 UI and Filesystem Communication

This section gives a general overview of the communication protocols between the various processes used by Arboreal.

4.1 Command Line Interface to File System

The command line communicates with the file system back-end via an interim program known as the *Liaison*. In order to maintain a relatively lightweight and encapsulated environment, each process has its own specific purview that it operates on and as a general rule, no one process interferes with the purview of another process.

The command line process' (*Cli.cpp*) purview is to capture user input, check to make sure it is valid and then send the user input to the "file system". This is achieved through the use of regular expressions (specifications for which can be viewed in the *Source Code Documentation* (Section 8)) for validation and a direct pipe to the *Liaison* process. However, the commands as the user entered them are not clean enough to be sent directly to the file system back-end. Until a message containing some form of "success" or "failure" message is received, the command line will not allow the user to continue inputting commands and will block until something is returned by the file system (since the *Liaison* process is the only process that communicates directly with the command line, any data will come from here rather than directly from the file system backend). A further important note is that if the command line invalidates a command, that command will never be sent to the *Liaison*.

This *Liaison* process' sole job is to intercept communications between the command line and the file system and convert them into actionable data for the file system back-end's (*daemon.cpp*) use. It does this by filtering the raw input it received from the command line through a *Parser* class which analyzes the input and restructures it according to which command the user requested. Some of these restructures can be similar in execution but many are unique to the command that the user wished to execute. On the following page is a table of currently implemented raw commands and the parser translations that will eventually be sent to the file system (4.1).

The parsed commands are not sent as a single long string but rather as separate communiques. After sending the parsed data, the *Liaison* will await a response from the bakend. The first of these backend responses will be the total number of bytes that the *Liaison* will have to read. If this is not sent, the *Liaison* has no idea how much data each command is generating and may either read too little or too much data (more likely the former). If this is allowed to happen, all manner of errors may manifest themselves. Finally, when the *Liaison* receives data back from the backend daemon, it will **not** pass the data back up to the command line. This is an attempt to increase

Table 1

		(These are NOT sent as a single string but individually)
“New” Commands		**(Spaces are for Clarity ONLY)**
	new -t [tag1,tag2…etc]	301 tag1 tag2 etc $
	new -f [tag1,tag2…etc]	302 file1- file2- etc $
	new -f /tag1/tag2/etc/file	300 /tag1/tag2/etc/file2 $
“Tag” Commands		
	tag [file1,file2…etc] +> [tag1,tag2…etc]	901 current directory/file1>tag1-tag2-etc $ 901 current directory/file2>tag1-tag2-etc $ etc..
	tag /tag1/tag2/…etc/file1 +> [tag1,tag2…etc]	/tag1/tag2/etc/file1>tag1-tag2-etc $
“Untag” Commands		
	untag [file1,file2…etc] -> [tag1,tag2…etc]	
	untag /tag1/tag2/…etc/file1 -> [tag1,tag2…etc]	
“Find” Commands		
	find -t [tag1,tag2…etc]	400 tag1 tag2 etc $
	find -f [file1,file2…etc]	401 file1 file2 etc $
“Delete” Commands		
	delete -t [tag1,tag2…etc]	501 tag1 tag2 etc $
	delete -f [file1,file2…etc]	502 file1 file2 etc $
“Open” Commands		
		(Open command requires “open flags” -r or -w or -x)
	open (-r|-w|-x) tag1/tag2/…etc/file1	200 /r|w|x/tag1/tag2/etc/file1 $
“Close” Commands		
	close tag1/tag2/…etc/file1	600 /tag1/tag2/etc/file2 $
“Rename” Commands		
	rename -t [tag1,tag2…etc] => [tagX,tagY…etc]	101 tag1-tagX tag2-tagY $
	rename /tag1/tag2/…etc/file1 => fileX	100 /tag1/tag2/etc/file1/fileX $
“Attributes” Commands		
	attire /tag1/tag2/…etc/file1	700 /tag1/tag2/etc/file1 $

Figure 4.1: Raw user input and its equivalent parser translation. Any program wishing to communicate directly to the file system via the daemon process will need to send commands formulated exactly as in column 3 of the table. These are generalized inputs and outputs please do not include any ellipses or ”etc” within the command that is sent to the daemon process.

performance as reading and writing to pipes is rather slow. Both the command line and *Liaison* operate on the same bash/shell window so the data printed to the outstream will seem to come from the command line process itself but it does in fact originate from the *Liaison* process.

The final link in the chain is the backend driver *(daemon.cpp)*. This driver (or more accurately *daemon*) functions as a server accepting any number of incoming connections, taking in the parsed data that the client has sent, and using it to execute the specified file system command. Once the command is executed, the data is passed back up to the client that requested it. Rather than using pipes for its communication, the backend daemon uses TCP sockets and a file descriptor array to keep track of all of its connections (The daemon's port number os **42069**). Each client must connect to the file system on a specific partition and that partition **must** exist. It is also important that only one file system "object" be created regardless of the number of connection attempting to access the file system. In order to keep track of all these moving parts as efficiently as possible, the daemon employs several "file-descriptor-to-X" maps. These maps not only allow better look up times and a more organized communication system/resource handling, but they also ensure that the information destined for *Client B* is never sent to *Client A*. It is important to note that due to this, only one client can operate on the file system at a time (even if the client is operating on a different partition than other clients) however, the daemon allows a backlog of requests and is sufficiently optimized as to seemingly allow concurrent communication among different clients. Finally, the daemon is open to any process which needs to connect to the file system and is not limited to the command line only, in fact the graphical user interface also connects to the file system via this daemon.

4.2 Graphical User Interface to FileSystem

For the majority of users a command line can be intimidating, and not always accessible despite its greater functionality. Because of this, a GUI was also implemented for the files system. The GUI seeks to emulate a traditional file explorer that most people are used to, in an aid to ease the transition for typical users. The GUI conencts to the daemon with the same port as the command line liason does, but the one caviot being the daemon must be strarted prior the the start up of the GUI. Once the daemon is started via the command line, the GUI can be statred from a command line as well, but an install package can be generated to allow launch from an icon like any other appication. Upon start up the GUI establishes a TCP connection with the deamon and polls it initially to get all the current tags that have been genreated. It then uses the TCP connection to get any of the informtation needed from the file system. It follows the same communication style as the liason, but the GUI is able to read in all the data from the file system at once. At this stage the GUI allows one to see a display of the tags that have been added to the filesystem, and upon clicking one being able to see all the files associated with the tag selected. It also allows for one to search for a file based upon its name. The GUI as of now doesn't match the functionality of the of the

command line, but with some future work many of the functions would be able to be implemented. As of now, it has basic browsing and no more. The GUI was developed using Electron a tool to develope portable cross platform desktop applications that can also be used in the web. It utilizes HTML for the layout and javascript to preform the any of the actions needed.

5 Performance Analysis

5.1 Testing Procedures/Methodology

All tests were performed on the same computer, concurrently. We used a program to create and populate a filesytem to generate realistic conditions and gather timing data along the way. A virtual disk of size 5 GB and formatted with *Arboreal* **Version 1.0** was used. The distribution of tags per file was a Chi-Squared distribution with 4 degrees of freedom. This created a situation where most files had 1-5 tags, but allowing a small percentage of files to have a larger amount of tags. We felt this adequately modeled real usage scenarios, since most files are only a couple directories deep, some are 6 or 7 directories and just a few are more than that. This distribution of files per tag was a lognormal distribution. This created a situation where most tags had a small number of files associated with them. However, a few select tags would have a large amount of files associated with them. Again this models reality as typically a user would have many files in Documents, Root, Pictures, etc.. but only a few files in something like Homework 2018. The operations were timed as the disk was populated. Files goes up to 300,000 and tags goes up to 5000.

5.2 Start Up

This is where most of our improvement needs to be done. *Arboreal* **Version 2.0** will address this issue and significantly improve results. While actual data collection of startup times at various file system states proved too time consuming, the expected time complexity is TODO. *Arboreal* **Version 1.0** also reads in every tag tree at startup which is an unnecessary overhead. Due to time constraints on the project, as well as the need to develop two User interfaces for interaction with *Arboreal*, we did not have time to refine this feature. *Arboreal* **Version 2.0** will selectively import tag trees at startup based on that user's habits and probable searches. Startup time is never expected to be fantastic, but we believe sacrificing some startup time in order to improve productivity for the duration of use is well worth it.

5.3 File Creation

Figure 5.1a shows Time~Files. There is no correlation here as expected. Figure 5.1b shows Time~Files. There is no correlation here as expected. Figure 5.1c shows Time~NumTags, where NumTags is the number of tags to be associated with the created file. Time

complexity is $nlog(n)$ with NumTags which is seen in the graph. NumTags was statistically significant. This is not a a great time complexity but, we always expect NumTags to be small.

Create File Statistical Data	
Variable	P-Value
NumTags	$2 * 10^{-16}$
log(NumTags)	$4.62 * 10^{-5}$
Residual Standard Error .01473	
$R^2 = 0.6098$	

5.4 Tag Creation

Figure 5.2a shows Time~Files. There is no correlation here as expected. Figure 5.2b shows Time~Files. There is no correlation here as expected. The line at the far right of the graph is caused by the fact that when doing the tests, I had the tags cap at 5000, but the files continue to increase, causing more data to be concentrated at this point.On both graphs, it appears a log on time would be in order. However, the graphs appear no different when that is done.

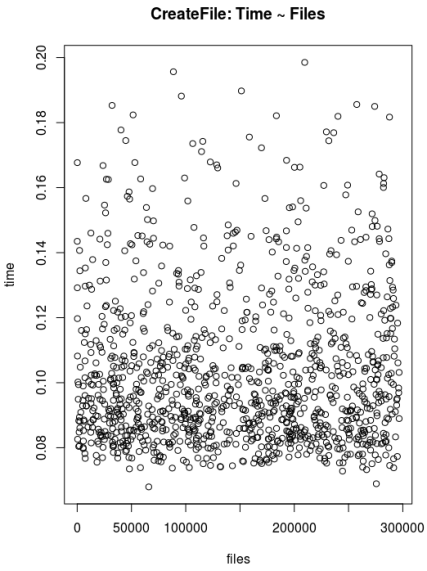
5.5 File Search

Speeding up file searching was one of the primary goals of Arboreal. We met that goal. Figure 5.3a shows Time~Files. There is no correlation here as expected. Figure 5.3b shows Time~Files. There is no correlation here as expected. However, these two graphs look a bit strange. This is due entirely to the way the dataset was created. As you can see in Figure 5.3c, the line pattern from both of these graphs comes from that. These graphs are also colored by the number of files found. Black, associated with only one file found is concentrated at the bottom. This demonstrates again that these line like patterns stem only from the number of files found. Figure 5.3c shows Time~NumFiles, where NumFiles is the number of files found in the search. This is linear as expected, simply because it takes longer to return 600 files that it does 1. NumFiles was the only statistically significant variable, shown below. We have very good predictability in this model, with a low Standard error and a very high R^2.

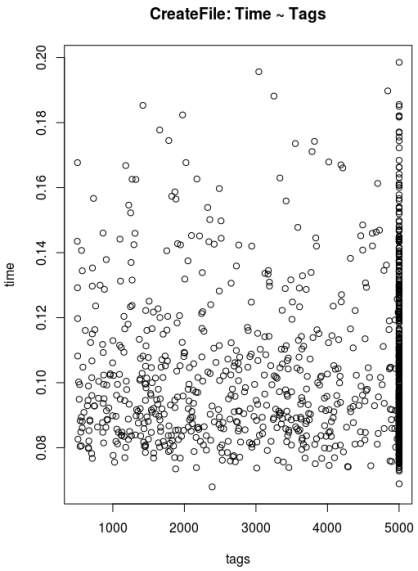
File Search Statistical Data	
Variable	P-Value
NumFiles	$2 * 10^{-16}$
Residual Standard Error .004936	
$R^2 = 0.982$	

5.6 Tag Search

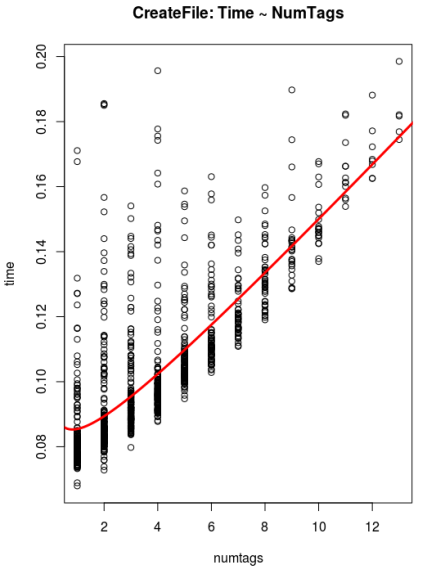
TagSearch is arguably the most versatile operation in *Arboreal*. It is also used very often. Having this be fast is a requirement for Arboreal to be a high functioning filesytem.



(a)

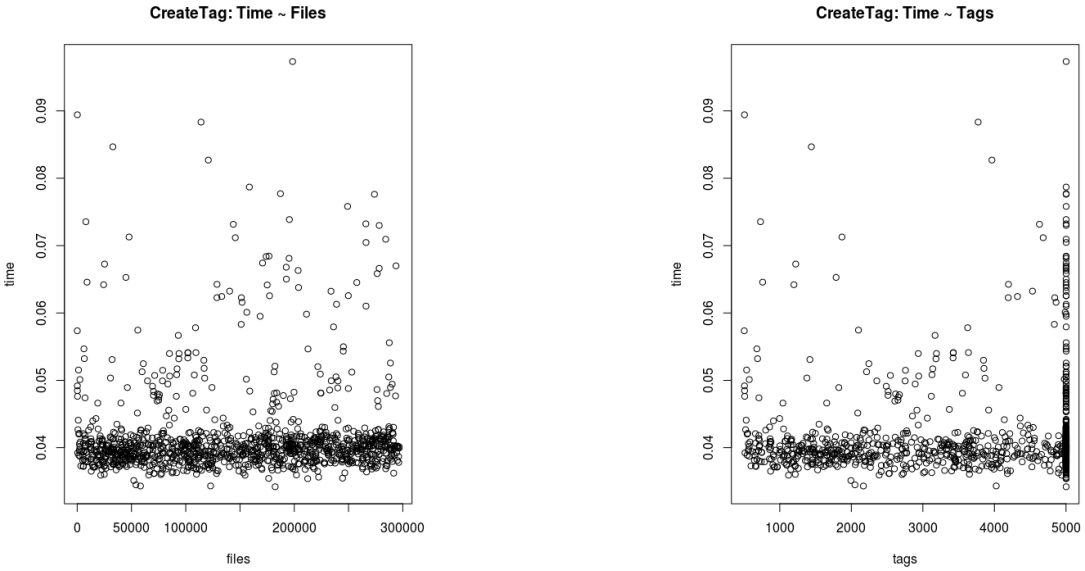


(b)



(c)

Figure 5.1: Graphs for CreateFile operation.

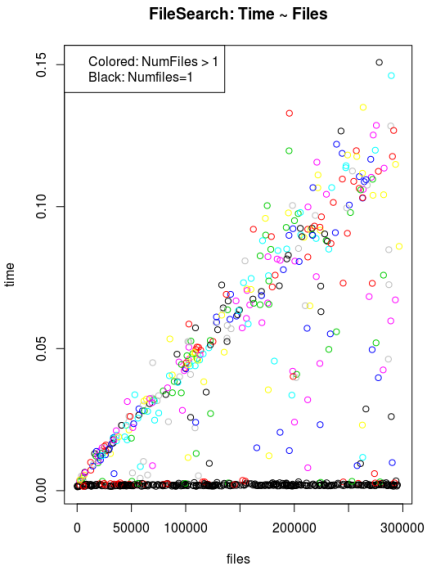


(a) (b)

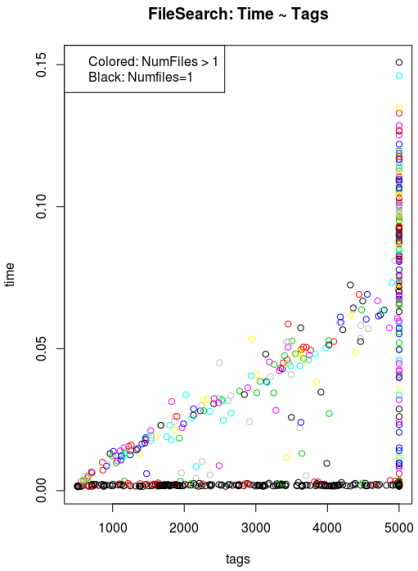
Figure 5.2: Graphs for CreateTags operation.

Figure 5.4a shows Time˜Files. There is no correlation here as expected. Figure 5.4b shows Time˜Files. There is no correlation here as expected. However, these two graphs look a bit strange in the same way the graphs from File Search looked strange. This is due to the same reasons. Figure 5.4c shows Time˜NumFiles, where NumFiles is the number of files found in the search. This is linear as expected, simply because it takes longer to return 3000 files that it does 1. Figure 5.4d shows Time˜NumTags The reason for such a sharp decline in time is two-fold. First, the more tags there are, the more likely one will be small and we can use that to narrow our search space for overlap. Second, The more randomly chosen tags you have, the less chance there will be any overlap. This is a flaw in the testing methodolgy. Unfortunately, it is not and easy one to combat. Due to the fact creating a large dataset that also tags files in such a way that you later can easily find different size overlaps for a large number of tags is not a trivial task. Future versions of Arboreal may use real data to better assess the capabilities. NumFiles was the only statistically significant variable, shown below. Our R^2 is not super high, but overall the linear model for NumFiles works pretty well. I was unable to find a correlation with NumTags.

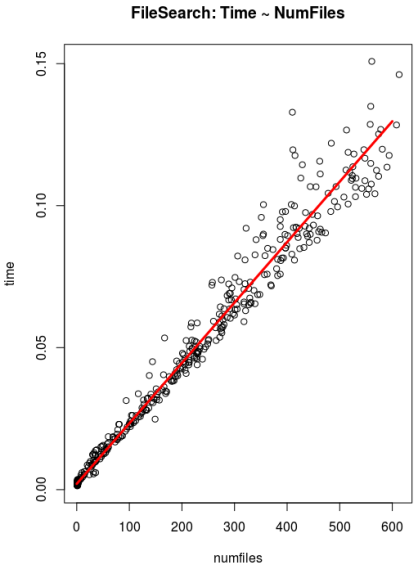
Tag Search Statistical Data	
Variable	P-Value
NumFiles	$2 * 10^{-16}$
Residual Standard Error .03415 $R^2 = 0.7366$	



(a)

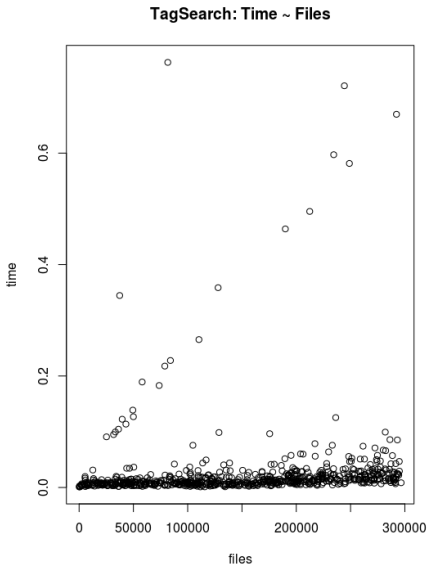


(b)

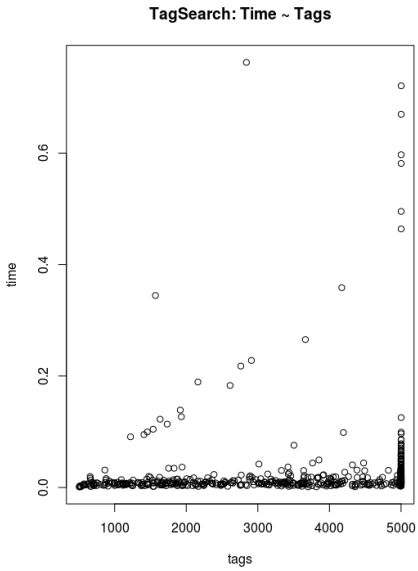


(c)

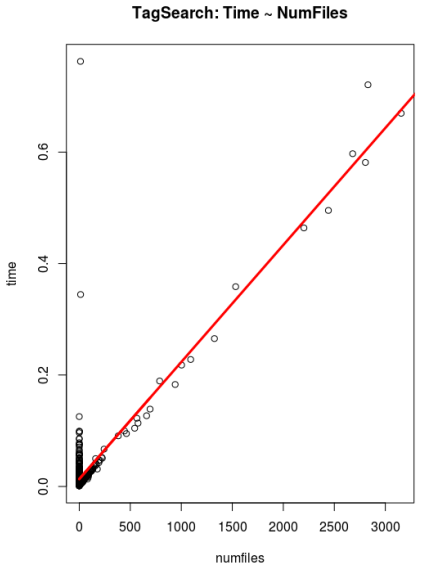
Figure 5.3: Graphs for FileSearch operation.



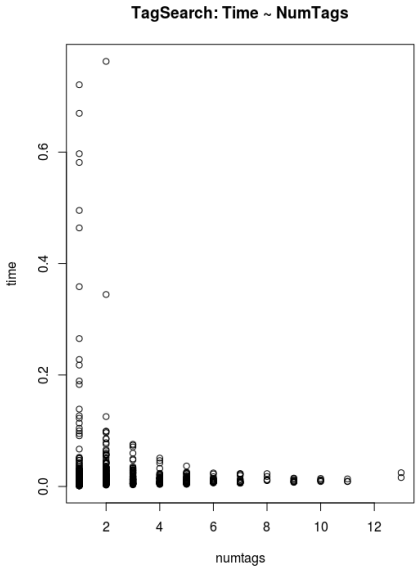
(a)



(b)



(c)



(d)

Figure 5.4: Graphs for TagSearch operation.

5.7 Tag File

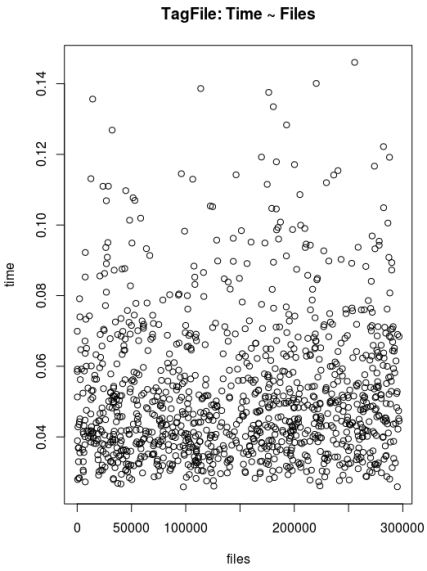
Tag File is expected to be a very fast operation, and it is. Figure 5.5a shows Time~Files. There is no correlation here as expected. Figure 5.5b shows Time~Tags. There is no correlation here as expected. Figure 5.4d shows Time~NumTags, where NumTags is the number of tags to be associated with the file. It has the same shape as the graph of Figure 5.1c. It was an $nlog(n)$ graph as well. NumTags was the only stastically significant variable as shown below. Again, not a super high ^{2}. This is due mainly to the variance in time, especially at lower numTag values. Overall, a good model for NumTags. We also always expect NumTags to be small.

Tag File statistical Data	
Variable	P-Value
NumTags	$2 * 10^{-16}$
log(NumTags)	$3.05 * 10^{-9}$
Residual Standard Error .009616	
$R^2 = 0.767$	

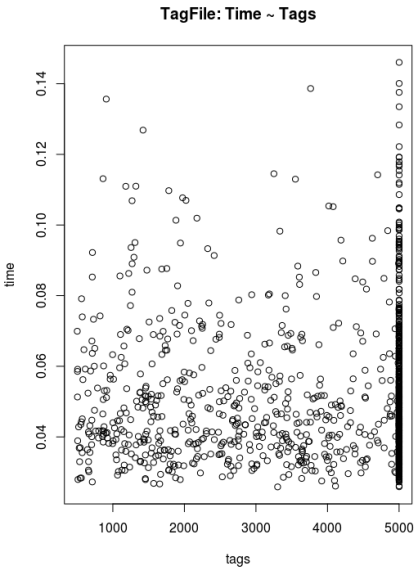
5.8 Rename Tag

RenameTag is a slower operation relative to everything else. This is one place where Arboreal is at a disadvantage to other filesystems. Renaming a directory with standard filesystems is a constant operation. We must touch every file in memory with associated with that tag. We do have ideas of how to improve this in the future and hope to eventually get it down to constant time. Figure 5.6a shows Time~Files. There is no correlation here as expected. Figure 5.6b shows Time~Tags. There is no correlation here as expected. Figure 5.6c shows Time~Associated Files, where Associated Files is the number of files associated with the tag to be renamed. RenameTag again shows the linear correlation in Figure 5.6a and Figure 5.6b that FileSearch and TagSearch showed. However, the only statistically significant variable is AssociatedFiles, which you can see is a very good model. With a high R^2, we also have very good predictability. AssociatedFiles may grow very large and this could end up being an expensive operation. It is currently the "worst" operation, and we have plans to improve it for *Arboreal* **Version 2.0**.

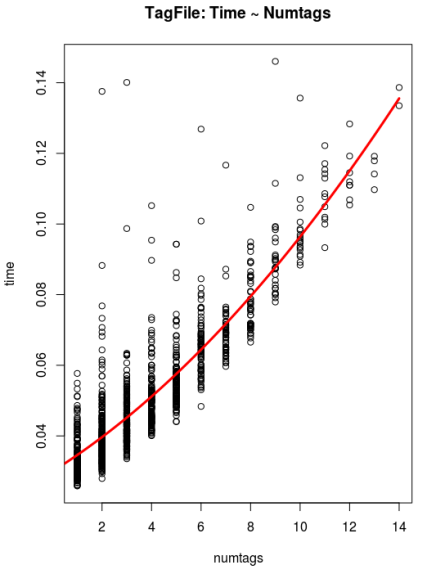
Rename Tag Statistical Data	
Variable	P-Value
AssociatedFiles	$2 * 10^{-16}$
Residual Standard Error .5441	
$R^2 = 0.9936$	



(a)

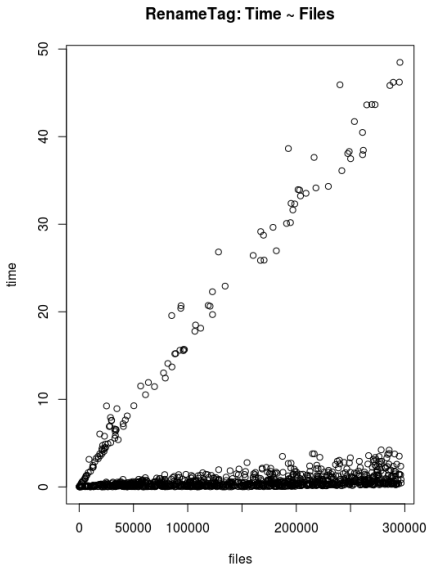


(b)

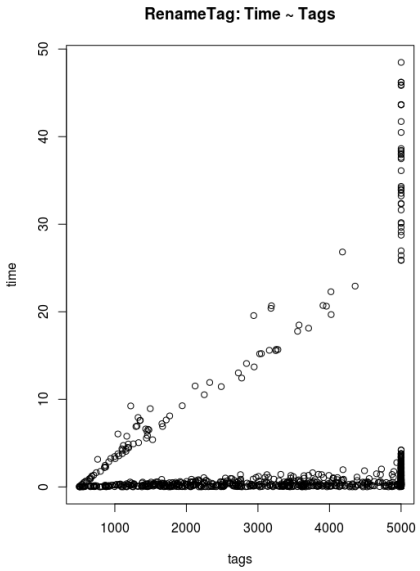


(c)

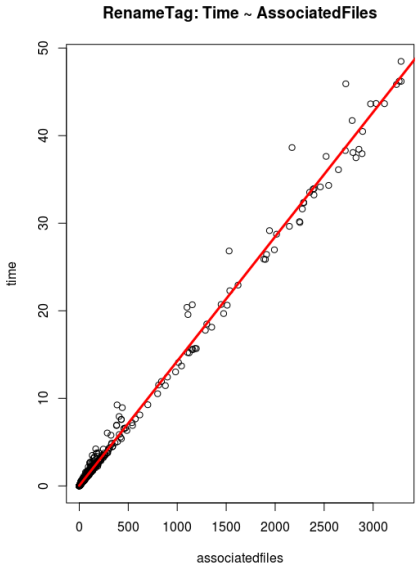
Figure 5.5: Graphs for TagFile operation.



(a)



(b)



(c)

Figure 5.6: Graphs for RenameTag operation.

6 Restrictions

Disk formatting is done via a program called *format.cpp* and a file called *diskInfo.d*. When editing the disk information file please keep the following in mind:

•Line 1 of the disk information file **must** have the following pattern: *diskfile name number of blocks on disk disk block size in bytes total number of partitions* (include spaces).

•Subsequent lines in the disk file will detail partition information and **must** be as follows: *partition name number of blocks within the partition maximum filename size in bytes*

•Filename sizes shall not exceed $blockSize/2$

•Block size **must** be a power of 2

•Hardcap on the number of tags associated with a file is equal to $((blockSize - filenameSize - 136)/sizeof(BlkNumType)) + (blockSize/sizeof(BlkNumType))$

•Maximum command size (sent from command line or liaison to daemon) is **4096 bytes**. Please pad all commands of size less than 4096 bytes to 4096 bytes with nullbytes (\0)

•Any commands that include comma separated lists should **not** include spaces in between each item within the list

•Currently requires about 300 MB of free RAM per 300,000 files.

7 User Guide

(See following pages)

Arboreal User Guide

Table of Contents

Installing Arboreal

Arboreal is currently not integrated with the kernel and as such runs similarly to a virtual file system albeit with a more experimental structure. Future work will be focused on direct integration with the kernal in order to provide more traditional usability. In the meantime playing around with and testing the file system can be achived through a few easy steps:

1. **Download the project**
2. **Change the directory (within the shell) to the folder within the project hierarchy named `Source`**
3. **Type `make` into your shell**
4. **You will now need to first run the daemon process. This process intercepts all communication attempts with the File System and will execute functions accordingly. There are a number of command line arguments that can be passed to the daemon:**
    - `-d` **This flag is used to tell the daemon to enable debugging**
    - `-v` **This flag is used to tell the daemon to return file information (such as that returned by a call to find) with as much information as possible. Omitting this flag will cause the daemon to return a reduced version of the file information**
    - **You may enable either of these options or both (input order does not matter, that is `-d -v` will work the same as `-v -d`)**
5. **Finally, Simply Type `./daemon` followed by your chosen flag or flags (be sure to include a space in between). For example, if I wanted to run the daemon with verbose file information and debugging enabled the command would look like: `./daemon -v -d`**

At this point you'll be ready to move on to the next step, starting the command line or GUI interface. Notice that the daemon does not output anything to the screen as it is running. **This is OK!** Its whole purpose is to be a background process that aids communication with the file system. **If you decided to enable debugging, the output will be located in a file called `Arboreal.log` .**

A final note:
By typing make, the "disk" will be formatted for you with the default values and partition names and counts. It is possible to change these to better suit your needs however it is a little bit more involved.

1. **Open and edit a file called `diskInfo.d` (It is located in the `Source` folder)**
2. **Using the following syntax, edit the file as you see fit:**
    - **Line 1 needs to always be:**
      `Diskfile name, number of blocks on disk, size of each block in bytes, number of partitions` (Omit the commas in favor of spaces)
    - **Lines 2 to X need to always be:** `Partition name, number of blocks in the partition, maximum filename size` (Again omit commas in favor of spaces)
    - **There are restrictions on values for `diskInfo` :**
        - **Filename sizes** are restricted to **no more than** `block size / 2`
        - **Block size must be a power of 2**
        - Bear in mind that there exists a **hardcap on the number of tags that can be associated with a file** and it is equal to:

          `( ( ( block size - filename size - 136 ) / sizeof( BlkNumType ) ) + ( block size / sizeof( BlkNumType ) )`
        - The **maximum possible blocksize** is `16 Kb`
3. **Next you will have to open `daemon.cpp` (it is located under `Source/Filesystem/`) and edit this line of code:**
    - `d = new Disk(#1, #,2 const_cast<char *>("diskfile_name"));`

- **Change `#1` to whatever value you picked for `number of blocks` in `diskInfo.d` .**
    - So if I decided that I wanted 4000 blocks I would type `4000` in for `#1` (**The number here and in `diskInfo.d` MUST MATCH**).
- **Change `#2` to whatever value you picked for `block size in bytes` in `diskInfo.d`**
    - So if I decided that I wanted blocks to be 4096 bytes large I would type `4096` in for `#2` (**Once again I stress that the number here and in `diskInfo.d` MUST MATCH**)
- **Finally, change `diskfile_name` to the name of the disk file you chose in `diskInfo.d`** (make sure they match!!)

4. **You are now almost ready, the final step is to type `make clean` followed by `make` in the shell and run the through the same steps as above for starting the daemon**
5. **You are now good to go!**

Starting The Command Line

Before beginning anything below, make sure that a daemon process (and ONLY ONE daemon process) is running, if the command line cannot connect to the daemon process it will quit on startup with an appropriate error message

The command line utility has multiple optional arguments but it does contain a single mandatory argument. This is the **Partition Name** that the command line will be working on. If no partition name is given the command line will fail on startup with an appropriate error message. Additionally it is important to note that **the partition that is given to the command line must already exist**. If it does not, an appropriate error will be thrown. Finally, **it does not matter if the partition is already in use by another command line and it does not matter how many command lines are currently active**, in both cases you will still be able to work with the file system (provided that the partition you gave exists). After providing the partitoon name you are free to run the command line. However, should you wish to, there are some optional command line arguments:

- `-d` **This argument will enable debugging for both the command line and the liaison process**
- `-s` **This argument will alert the command line that input will be coming from a file rather than a user**
- `-s -d` **This will enable debugging for the command line AND alert it to the fact that input will be coming from a file rather than a user**

For example, if i wished to pipe input from a file to the command line and enable debugging, I would run the command line process like so:
`./commandline PartitionName -s -d < some_random_file.ext`

(*Note that* `./commandline -d -s < some_random_file.ext` *will not work, that is, make sure the debug flag comes last!*)

But if I wanted to just enable debugging and read from user input, I would run the command line process like so:
`./commandline PartitionName -d`

At this point you should see the arboreal header and `Arboreal >>` indicating that the command line is ready to accept input. **To send input to the command line simply type the command you wish to execute** (see *Valid Commands And Their Syntax* section for commands or type `help` or `h`) **and press enter.**

Note
If you chose to enable debugging for the command line, all debug output will be written to a file named `Arboreal.log`. Do not worry if this file does not yet exist, it will be created for you on startup.

Valid Commands And Their Syntax

Before we begin please make note of a few things:

1. **There is a maximum command size of `4096 bytes`** Although a lot of these commands have support for defining multiple operations within a single command (such as search for multiple files by name without having to enter a seperate `find -f` command for each file), if the total size of the command exceeds the current default of `4096 bytes`, the command will fail to send and be ignored.
2. **DO NOT include spaces between `list` and/or `set` items or between items and `{}` or `[]` or `/`.**
3. Commands of the form `../tag/tag/file` or `../tag/tag` only **emulate** directory structures. In fact, **Arboreal does not have the concept of a directory.** However, this style of syntax is universally understood and easy to type and so was maintained.

"Help" Commands

```
Arboreal >> help
Arboreal >> h
```

These two commands will bring up a helper subprocess which will display a list of the command archetypes and show the user the specific commands (and their syntax) that are housed under each archetype. The helper subprocess continues running until the user decides to quit it.

```
Arboreal >> -h --command_archetype

e.g.
Arboreal >> -h --find
```

This version of the help command will show the usage for a single command archetype. (Unlike the `help` or `h` commands it will not start a "helper" subprocess but will simply display the usage for the particular archetype and await the next file system command)

"Quit" Commands

```
Arboreal >> quit
Arboreal >> q
Arboreal >> Q
```

All of these will attempt to terminate the current command line process. This command does not affect other concurrently running command lines it will only quit the currently active command line process. The user must confirm the quit before the command will actually be executed. this is to prevent accidental quits. The quit commands are built with proper cleanup in mind and should not leave any junk behind.

Changing The Current Working Directory

```
Arboreal >> cd tagnameX/.../tagnameXn
Arboreal >> cd ./tagname1/tagname2
```

Changing "directories" is quite easy and supports relative paths and absolute paths.

- Version 1 will change to the "directory" which includes files tagged with all of the tags included in the command. That is, if you were to `cd Documents/December/Papers` all files "within" the "directory" would be tagged with `Documents, December, Papers`.
- Version 2 will append whatever you typed to the current working directory. **You MUST include the period**. For example if I'm currently in the "directory" `Documents/December/Papers` and I `cd ./CreativeWriting/BeerBrawls`, my resultant "directory" will be `Documents/December/Papers/CreativeWriting/BeerBrawls`.

"Find" Commands

Find Files By Tag

```
Arboreal >> find -t [tagnameX,...,tagnameXn]
Arboreal >> find -t {tagnameX,...,tagnameXn}
Arboreal >> find -t [tagnameZ,{tagnameX,...,tagnameXn},...,tagnameZn]
Arboreal >> find -t {tagnameZ,tagnamZ1,[tagnameX,...,tagnameXn],...,tagnameZn}
```

This command searches for files by tag. It is quite powerful and allows you to search for any combination of tags. Commands that use `{}` are called `sets` and will tell the file system to **search for ALL files which are tagged with ALL of the specified tags.** You can think of this as a bunch of `&&` operations, that is, you want a file tagged with :

`{ this tag, and this tag, and this tag, ... etc}`

Commands that use `[]` are called `lists` and will tell the system to **search for ANY file which is tagged with ANY of the tags specified.** You can think of this as a bunch of `||` operations, that is, you want a file tagged with:

`[this tag, or this tag, or this tag, ... etc]`

What's great is that you can actually nest any of these within one another! Although nesting a bunch of `sets` or `lists` won't be any diffferent from simply using one big list or set (i.e. `[t1,[t2,t3,t4]]` is the exact same as `[t1,t2,t3,t4]` this goes for `sets` as well). However, things get interesting when you pass a command such as:

`find -t [tag1,tag2,{tag45,tag78,[tag9,tag10],tag5},tag100]`

This particular command will search for any file with:

```
    ( tag1 )
    -or- ( tag2 )
    -or- ( tag100 )
    -or- ( tag45 & tag78 & tag9 & tag5 )
    -or- ( tag45 & tag78 & tag10 & tag5 )
```

(*Of course you can accomplish similar things with a command that is a `list` nested within a `set` rather than this example which is a `set` nested within a `list`*)

As you can see, nesting these operations creates some really powerful search options!

Find Files By Name

```
Arboreal >> find -f [filenameX(.ext),...,filenameXn(.ext)]
```

This command will (as the title suggests) find a file based on its name. In order to make searching for multiple files easier, you have the option of defining several filenames to search for. For example:

```
Arboreal >> find -f [BookReport.pdf,ExperimentData.numbers,Makefile]
```

Will search for a file named `BookReport.pdf`, a file named `ExperimentData.numbers`, and a file named `Makefile`. **This command does not support nested `[]` or `{}` as these have no meaning when searching for files by name.**

It is also important to note that, while file extensions are optional for this command, a search for `BookReport` will NOT return `BookReport.pdf` and a search for `BookReport.pdf` will NOT return `BookReport`.

Both finding files by tag and by name will return file information and the toal number of files found. If the file could not be found or the tag exists but is completely empty, a message stating so will be returned instead. If the results are a mixure of non existant and existant files, the appropriate message will be printed for each file (either the file information or "not found" message) and a count of the total number fo files found will still be displayed.

"New" Commands

Creating New Tags

```
Arboreal >> new -t [tagnameX,...,tagnameXn]
```

You may specify any number of new tags to be created provided that the resultant command length does not exceed the maximum of `4096 bytes`. You also need not worry if a tag already exists but you did not know it at the time. **The file system will skip the creation of tags that already exist and tell you if that happend.** You can call this command from anywhere, that is, it does not care about the current working directory.

Creating New Files

```
Arboreal >> new -f [filenameX(.ext),...,filenameX2(.ext)]

Arboreal >> new -f tagnameX/.../tagnameXn/filename(.ext)
```

There are two ways of creating a new file and they have a very important distinction between them:

- **#1:** allows you to create any number of files **within the current working directory ONLY**
- **#3:** allows you to create a **SINGLE file regardless of the current working directory however, you MUST specify the absolute "path" of the file.** *(In a future update we may implement relative paths for this as well)*

If a file could not be created (possibly because it already exists), an appropriate message will be returned instead of the newly created file's information.

"Delete" Commands

Deleting Tags

```
Arboreal >> delete -t [tagnameX,...,tagnameXn]
```

Delete any number of tags. **Tags will be deleted ONLY if they are empty.**

Deleting Files

```
Arboreal >> delete -f [filenameX(.ext),...,filenameXn(.ext)]

Arboreal >> delete -f tagnameX/.../tagnameXn/filename(.ext)
```

As with the `new` commands, there are two ways to delete a file:

- **#1:** will delete any number of files **provided that they exist within the current working directory.**
- **#3:** will delete a **single file regardless of current working directory, however, the file's absolute "path" MUST be provided.** *(In a future update we may implement relative paths for this as well)*

Although extensions on files are optional for these commands, you should be aware that:
`delete -f [BookReport.pdf]` will NOT delete `BookReport` `delete -f [BookReport]` will NOT delete `BookReport.pdf`

"Tag" Commands

```
Arboreal >> tag [filenameX(.ext),...,filenameXn(.ext)] +> [tagnameX,...,tagnameXn]

Arboreal >> tag tagnameX/.../tagnameXn/filename(.ext) +> [tagnameX,...,tagnameXn]
```

You may start noticing a pattern here of commands coming in pairs. This is completely natural and you are not going crazy. As with previous commands there are two ways to tag a file:

- **#1:** will **tag any number of files within the current working directory with any number of tags**. If the file or files have already been associated with a given tag then that tag is skipped.
- **#3:** will **tag a file regardless of the current working directory but the file's absolute path MUST be provided.**

(As always extensions on file names are optional but not providing them (if the file has one) will result in the expected file not being tagged or else an error stating that the file does not exist)

"Untag" Commands

```
Arboreal >> untag [filenameX(.ext),...,filenameXn(.ext)] -> [tagname1,...,tagnameXn]

Arboreal >> untag tagnameX/.../tagnameXn/filename(.ext) -> [tagnameX,...,tagnameXn]
```

- **#1:** will untag any number of files **provided that they are within the current working directory** and that they **are in fact tagged with the tags that you wish to be removed**.
- **#3:** will untag a **single file regardless of the current working directory,** provided that the file's **absolute path** is given and that the file is tagged with the tags you want to remove

(As always extensions on file names are optional but not providing them (if the file has one) will result in the expected file not being untagged or else an error stating that the file does not exist)

"Merge" Commands

Please note that these are still in production and not currently available

```
Arboreal >> merge tagnameX => tagnameY

Arboreal >> merge [tagnameX1,...,tagnameXn] => tagnameY
```

- **#1:** will merge `tagnameX` into `tagnameY`
- **#3:** will merge `tagnameX - tagnameXn` into `tagnameY`

"Open" Commands

```
Arboreal >> open filename(.ext)
Arboreal >> open tagnameX/.../tagnameXn/filename(.ext)
```

- **#1:** will open a file **provided that it exists within the current directory**
- **Version 2** will open a file regardless of the current working directory **provided that the file exists and the file's absolute path is provided**

"Close" Commands

```
Arboreal >> close filename(.ext)

Arboreal >> close tagnameX/.../tagnameXn/filename(.ext)
```

- **#1:** will close a file **provided that it is currently open**
- **#3:** will close a file regardless of the current working directory **provided that the file is open and the file's absolute path is provided**

"Read" Commands

Please note that these are still in production and not currently available

```
Arboreal >> read filename(.ext)

Arboreal >> read filename(.ext) -b #-of-bytes-to-read

Arboreal >> read tagnameX/.../tagnameXn/filename(.ext)

Arboreal >> read tagnameX/.../tagnameXn/filename(.ext) -b #-of-bytes-to-read
```

- **#1:** reads the entire contents of a file and outputs them to the screen. Will automatically open the fie named (assuming it exists) **provided that the file is within the current working directory.**
- **#3: reads X bytes** of a file and outputs them to the screen. Will automatically open the file named, (assuming it exists) **provided that the file is within the current working directory.**
- **#5:** reads the entire contents of a file and outputs them to the screen. Will automatically open the file named (assuming it exists) **regardless of the current directory.**
- **#7: reads X bytes** of a file and outputs them to the screen. Will automatically open the file named (assuming it exists) **regardless of the current directory**.

"Write" Commands

Please note that these are still in production and not currently available

```
Arboreal >> write tagnameX/.../tagnameXn/filename(.ext)

Arboreal >> write -a tagnameX/.../tagnameXn/filename(.ext)

Arboreal >> write filename(.ext)

Arboreal >> write -a filename(.ext)
```

- **#1:** Writes whatever comes after the command is executed to the specified file. **Full path to file MUST be provided. Previous file contents WILL be OVERWRITTEN**
- **#3:** Appends whatever comes after the command is executed to the specified file. **Full path to file MUST be provided**
- **#5:** Writes whatever comes after the command is executed to the specified file. **File MUST be within the current directory. Previous file contents WILL be OVERWRITTEN**
- **#7:** Appends whatever comes after the command is executed to the specified file. **File MUST be within the current directory.**

"Copy" Commands

Please note that these are still in production and not currently available

```
Arboreal >> copy tagnameX/.../tagnameXn/filename(.ext) -> tagnameX/.../tagnameXn/filename2(.ext)

Arboreal >> copy filename1(.ext) -> filename2(.ext)

Arboreal >> copy -b Z tagnameX/.../tagnameXn/filename(.ext) -> tagnameX/.../tagnameXn/filename2(.ext)

Arboreal >> copy -b Z filename(.ext) -> filename2(.ext)

Arboreal >> copy -a -b Z tagnameX/.../tagnameXn/filename(.ext) -> tagnameX/.../tagnameXn/filename2(.ext)

Arboreal >> copy -a -b Z  filename(.ext) -> filename2(.ext)
```

- **#1**: Copies the contents of "filename" to "filename2", **Overwrites "filename2" in the process; Absolute paths for both files MUST be provided.**
- **#3:** Copies the contents of "filename" to "filename2", **Overwrites "filename2" in the process; BOTH files MUST be within the current directory.**
- **#5:** Copies the first "Z" bytes from "filename" to "filename2", **Will overwrite the first "Z" bytes of "filename2" with the first "Z" bytes of "filename"; Absolute paths for both files MUST be provided.**
- **#7:** Copies the first "Z" bytes from "filename" to "filename2", **Will overwrite the first "Z" bytes of "filename2" with the first "Z" bytes of "filename"; BOTH files MUST be within the current directory.**
- **#9:** Appends the first "Z" bytes of "filename" to "filename2", **Does NOT overwrite the contents of "filename2"; Absolute paths for both files MUST be provided.**
- **#11:** Appends the first "Z" bytes of "filename" to "filename2", **Does NOT overwrite the contents of "filename2"; BOTH files MUST be within the current directory.**

"Rename" Commands

Version #3 Not Yet Available

```
Arboreal >> rename -t [tagname1old,...,tagnameXold] => [tagname1new,...,tagnameXnew]

Arboreal >> rename -f [filename1old(.ext),...,filenameXold(.ext)] => [filename1new(.ext),...,filenameXnew(.ext)]

Arboreal >> rename tagname1/.../tagnameX/filenameOld(.ext) => filenameNew(.ext)
```

- **#1:** Renames any number of tags. **Tags are renamed in order, that is, the tag at index == 1 in the first list will be renamed to the name at index == 1 in the second list.** If a tag will be renamed to another tag which already exist, or the tag to be renamed does not exist, the operation will fail.
- **#3:** (Not yet available) Renames any number of files. **All files to be renamed MUST exist within the current working directory; Files are renamed in order, that is, the file at index == 1 in the first list will be renamed to the name at index == 1 in the second list**. If a file will be renamed to another file which already exists, or the file to be renamed does not exists, the operation will fail.
- **#5:** Rename a single file from anywhere. **Absolute path to file MUST be provided** If a file will be renamed to another file which already exists, or the file to be renamed does not exists, the operation will fail.

"File Attributes" Commands

Version #3 Not Yet Available

```
Arboreal >> attr tagname1/.../tagnameX/filename(.ext)

Arboreal >> attr [filename1(.ext),...,filenameX(.ext)]
```

- **#1:** Gets the file attributes for a single file. **Absolute path to file MUST be provided**. If the file does not exist, the operation will fail.
- **#3:** (Not yet available) Get the attributes of any number of files provided that **those files exist AND are within the current directory.**

The Graphical User Interface

Troubleshooting

Help My List Command Keeps Coming Up As Invalid: Please remeber that there should be no spaces inbetween list elements.

Help My Open Command Keeps Coming Up As Invalid: Please remember that open comands require a flag such as -r or -w or -x infront of the file to be opened.

Help The Help Process Is Not Recognizing My Commands: Please remember that the help process syntax is `--command-from-table` no spaces.

Help I Can't Tag My File: Make sure that the tag you are trying to use actually exists / the file you want to tag exists

Help My List Commands Won't Work: In general any list command that is not a creation command requires that you be in the right tag space in order to work. Thus if you are trying to say tag files tagged with for example `tag1` from the root directory the command will fail unless you use the "complete path version". Simply `cd /tag1` in order to be able to use the list command again.

Help `lt` command shows only some of the tags: This is a known bug and has not yet been fixed. Just keep typing `lt` eventually all the tags will show up.

Help My File Extensions Aren't Working: File extensions can be only alphabetic charcters, no numbers please. If this is entered: `new -f [file.1]` the commandline will treat `1` as a file name and the `.` as an improper comma and the command will fail.

Help My Absolute Path Commands Aren't Working: Make sure all tags and files exist and make sure the path starts with `/`.

Help How Do I Display All Files: `find -t [default]`

Help I'm Sure I'm Correct But the Command Still Won't Work: Please open a github issue with the command you are trying to execute, what you typed in, and the Arboreal.log file generated (this will give us information as too what else was done previous to the bad command) we will get too it as soon as we can!

8 Source Code Documentation

(See following pages) *Note that table of contents page number hyperlinks are not accurate; the table of contents is only included in order to provide an overview of the Source code documentation contents.*

Arboreal

Generated by Doxygen 1.8.14

Contents

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

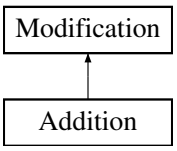
Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

Class Documentation

4.1 Addition Class Reference

Inheritance diagram for Addition:



Public Member Functions

- **Addition** (TreeObject ∗obj, TreeObject ∗parent)
- void **write_out** (PartitionManager ∗pm)

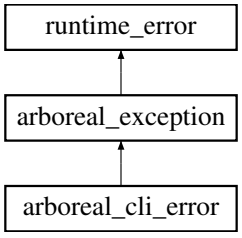
Additional Inherited Members

The documentation for this class was generated from the following files:

- Filesystem/DaemonDependancies/Trees/Trees.h
- Filesystem/DaemonDependancies/Trees/Trees.cpp

4.2 arboreal_cli_error Class Reference

Inheritance diagram for arboreal_cli_error:



Public Member Functions

- **arboreal_cli_error** (const string &where, const string &what, const int ecode=99)
- **arboreal_cli_error** (const char ∗what, const char ∗where, const int ecode=99)
- **arboreal_cli_error** (const char ∗what, const string &where, const int ecode=99)
- **arboreal_cli_error** (const string &what, const char ∗where, const int ecode=99)

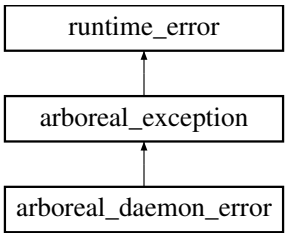
Additional Inherited Members

The documentation for this class was generated from the following files:

- SharedHeaders/Arboreal_Exceptions.h
- SharedCPPFiles/Arboreal_Exceptions.cpp

4.3 arboreal_daemon_error Class Reference

Inheritance diagram for arboreal_daemon_error:



Public Member Functions

- **arboreal_daemon_error** (const string &where, const string &what, const int ecode=99)
- **arboreal_daemon_error** (const char ∗what, const char ∗where, const int ecode=99)
- **arboreal_daemon_error** (const char ∗what, const string &where, const int ecode=99)
- **arboreal_daemon_error** (const string &what, const char ∗where, const int ecode=99)

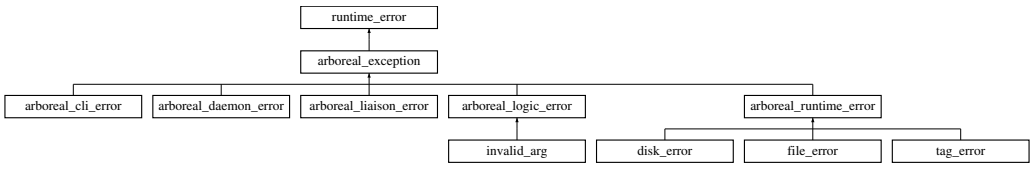
Additional Inherited Members

The documentation for this class was generated from the following files:

- SharedHeaders/Arboreal_Exceptions.h
- SharedCPPFiles/Arboreal_Exceptions.cpp

4.4 arboreal_exception Class Reference

Inheritance diagram for arboreal_exception:



Public Member Functions

- **arboreal_exception** (const char ∗what, const char ∗where, const int ecode=99)
- **arboreal_exception** (const char ∗what, const string &where, const int ecode=99)
- **arboreal_exception** (const string &what, const string &where, const int ecode=99)
- **arboreal_exception** (const string &what, const char ∗where, const int ecode=99)
- virtual const char ∗ **where** () const
- virtual const int **ecode** () const

Protected Attributes

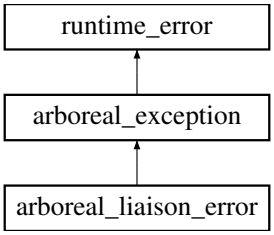
- string **_where**
- int **_ecode**

The documentation for this class was generated from the following files:

- SharedHeaders/Arboreal_Exceptions.h
- SharedCPPFiles/Arboreal_Exceptions.cpp

4.5 arboreal_liaison_error Class Reference

Inheritance diagram for arboreal_liaison_error:



Public Member Functions

- **arboreal_liaison_error** (const string &where, const string &what, const int ecode=99)
- **arboreal_liaison_error** (const char ∗what, const char ∗where, const int ecode=99)
- **arboreal_liaison_error** (const char ∗what, const string &where, const int ecode=99)
- **arboreal_liaison_error** (const string &what, const char ∗where, const int ecode=99)

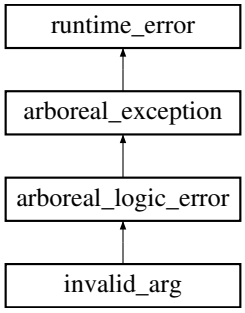
Additional Inherited Members

The documentation for this class was generated from the following files:

- SharedHeaders/Arboreal_Exceptions.h
- SharedCPPFiles/Arboreal_Exceptions.cpp

4.6 arboreal_logic_error Class Reference

Inheritance diagram for arboreal_logic_error:



Public Member Functions

- **arboreal_logic_error** (const char ∗what, const char ∗where, const int ecode=99)
- **arboreal_logic_error** (const char ∗what, const string &where, const int ecode=99)
- **arboreal_logic_error** (const string &what, const string &where, const int ecode=99)
- **arboreal_logic_error** (const string &what, const char ∗where, const int ecode=99)

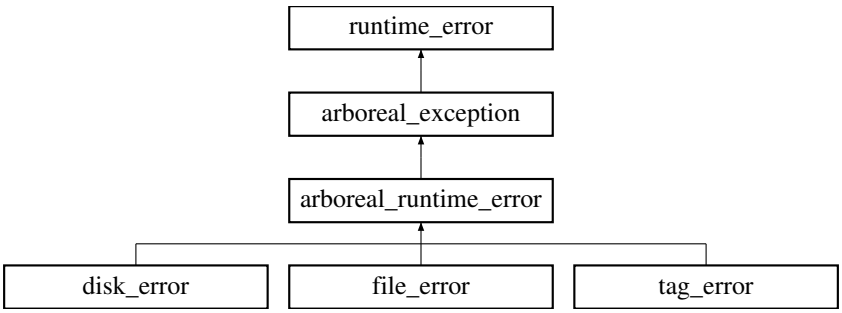
Additional Inherited Members

The documentation for this class was generated from the following files:

- SharedHeaders/Arboreal_Exceptions.h
- SharedCPPFiles/Arboreal_Exceptions.cpp

4.7 arboreal_runtime_error Class Reference

Inheritance diagram for arboreal_runtime_error:



Public Member Functions

- **arboreal_runtime_error** (const char ∗what, const char ∗where, const int ecode=99)
- **arboreal_runtime_error** (const char ∗what, const string &where, const int ecode=99)
- **arboreal_runtime_error** (const string &what, const string &where, const int ecode=99)
- **arboreal_runtime_error** (const string &what, const char ∗where, const int ecode=99)

Protected Attributes

- string **_where**
- int **_ecode**

The documentation for this class was generated from the following files:

- SharedHeaders/Arboreal_Exceptions.h
- SharedCPPFiles/Arboreal_Exceptions.cpp

4.8 Attributes Class Reference

Public Member Functions

- **Attributes** (BlkNumType blknum, PartitionManager *pm)

Modifier Functions

- void write_out ()
- void read_in ()
- void del ()
- void set_creation_time ()
- void set_owner (int owner)
- void set_permissions (char *perms)
- void set_access ()
- void set_edit ()
- void update_size (size_t size)

Accessor Functions

- time_t get_creation_time ()
- int get_owner ()
- char * get_permissions ()
- time_t get_access ()
- time_t get_edit ()
- size_t get_size ()
- FileAttributes get_file_attributes ()

4.8.1 Member Function Documentation

4.8.1.1 del()

```
void Attributes::del ( )
```

Removes the Attributes presence on disk

4.8.1.2 get_access()

```
time_t Attributes::get_access ( )
```

Returns

the UNIX time the file was last accessed

4.8.1.3 get_creation_time()

```
time_t Attributes::get_creation_time ( )
```

Returns

the UNIX time the file was created

4.8.1.4 get_edit()

```
time_t Attributes::get_edit ( )
```

Returns

the UNIX time the file was last edited

4.8.1.5 get_file_attributes()

```
FileAttributes Attributes::get_file_attributes ( )
```

Returns

the entire FileAttributes struct

4.8.1.6 get_owner()

```
int Attributes::get_owner ( )
```

Returns

the UID of the owner of the file

4.8.1.7 get_permissions()

```
char * Attributes::get_permissions ( )
```

Returns

the permisssions

See also

FileInfo::get_permissions(char*)

4.8.1.8 get_size()

```
size_t Attributes::get_size ( )
```

Returns

the size of the file in bytes

4.8.1.9 read_in()

```
void Attributes::read_in ( )
```

Reads in the Attributes from disk

4.8.1.10 set_access()

```
void Attributes::set_access ( )
```

Marks down the time as accessed time as UNIX timestamp

4.8.1.11 set_creation_time()

```
void Attributes::set_creation_time ( )
```

Marks down the creation time of the associated FileInfo as UNIX timestamp

4.8.1.12 set_edit()

```
void Attributes::set_edit ( )
```

Marks down the time as modified time as UNIX timestamp

4.8.1.13 set_owner()

```
void Attributes::set_owner (
            int owner )
```

Marks the owner as their UID

4.8.1.14 set_permissions()

```
void Attributes::set_permissions (
            char * perms )
```

sets the permisssions of the file

See also

FileInfo::set_permissions(char*)

4.8.1.15 update_size()

```
void Attributes::update_size (
            size_t size )
```

sets the size to the specified size

4.8.1.16 write_out()

```
void Attributes::write_out ( )
```

Writes out the Attributes to disk

The documentation for this class was generated from the following files:

- Filesystem/DaemonDependancies/Trees/Trees.h
- Filesystem/DaemonDependancies/Trees/Trees.cpp

4.9 CLI Class Reference

Public Member Functions

- CLI (char **partition)
- CLI (char **partition, bool debug)
- CLI (char **partition, char *isScript)
- CLI (char **partition, char *isScript, bool debug)
- ~CLI ()
- void start ()
- void run (std::string input)
- void run ()
- char * build (const int id, const std::string input)
- void send_cmnd (const char *command)
- void await_response ()

  Block while waiting for response from filesystem.

4.9.1 Constructor & Destructor Documentation

4.9.1.1 CLI() [1/4]

```
CLI::CLI (
            char ** partition )
```

Parameters

partition	A pointer to a charachter array containing the partition name that this particular command line interface will operate in

Constructor for use in Mode 1 of the Command Line Interface Reads from explicit user input Does NOT print debug data to log

4.9.1.2 CLI() [2/4]

```
CLI::CLI (
            char ** partition,
            bool debug )
```

Parameters

partition	A pointer to a charachter array containing the partition name that this particular command line interface will operate in
debug	Wether or not debug messages should be turned on for this interface

Constructor for use in Mode 2 of the Command Line Interface Reads from explicit user input Does PRINTS DEBUG data to log

4.9.1.3 CLI() [3/4]

```
CLI::CLI (
            char ** partition,
            char * isScript )
```

Parameters

partition	A pointer to a charachter array containing the partition name that this particular command line interface will operate in
isScript	Flag telling whether or not the input for this interface will be coming from a file (The flag value is '-s')

Constructor for use in Mode 3 of the Command Line Interface Reads from file Does NOT print debug data to log

4.9.1.4 CLI() [4/4]

```
CLI::CLI (
            char ** partition,
            char * isScript,
            bool debug )
```

Parameters

partition	A pointer to a charachter array containing the partition name that this particular command line interface will operate in
debug	Wether or not debug messages should be turned on for this interface
isScript	Flag telling whether or not the input for this interface will be coming from a file (The flag value is '-s')

Constructor for use in Mode 3 of the Command Line Interface Reads from file Does PRINTS DEBUG data to log

4.9.1.5 ∼CLI()

```
CLI::∼CLI ( )
```

Default Destructor

4.9.2 Member Function Documentation

4.9.2.1 await_response()

```
void CLI::await_response ( )
```

Block while waiting for response from filesystem.

Receive data from the liaison process The data is X number of charachters The data can be anything from a list of files returned by the 'find' operation To an error message. This function blocks until it receives data.

Most filesystem commands operate on a 1:1 ratio, that is, sending one command will generate one response. However, some commands (most notably 'find' & 'read') may have a ratio of 1:Many (For example 'find -t [tag1]' may return any number of files but it is only a single command). In situations such as these it is necssary to tell the Command Line to wait until the filesystem has sent all data. Thus, if the Command Line receives "WAIT" it will know to continue to block on a call to receive until the Liaison has gathered all of the nescessary data. However this is still not enough and it is also nescessary to tell the Command Line how much data it must read, for this reason, the first piece of data that the Liaison will send, will be the number of bytes the Command Line needs to read. After this value is received the actual data is sent.

4.9.2.2 build()

```
char * CLI::build (
            const int id,
            const std::string input )
```

Converts a std::string to a C-Style String, embeds the command id into the C-String, and pads it to length = Max↩ BufferSize

Parameters

id	File System Command ID
input	File System Command

Returns

A C-Style String of length = MaxBufferSize containing the command ID in the first X Bytes where X is the size of an integer type followed by the command itself followed by as many nullbytes as nescesarry in order to have a length = MaxBufferSize

Format user input for use by Liaison process:

1) Prepend a byte representation of the command ID to the array 2) Copy the user input into the the array (skip the first X indecies were X is the size of an integer (we don't want to overwrite the command ID))

Parameters

id	Comand ID
input	User input string

Returns

A pointer to a charachter array

4.9.2.3 run() [1/2]

```
void CLI::run (
            std::string input )
```

This function operates the same as run() but takes its input from a filestream rather than a user. Reads in the input data (A File System Command) and sends it down to the file system.
Some commands that do not need to interact with the File System code are handled in this function. For example, displaying the 'help' messages is executed from this function since the File System does not have or need and 'help' command. This function will block until it receives a response from the File System (provided that the command inputted is intended to go to the File System) this function will continue reading from the input file until an error occurs or 'end' is read in.

Parameters

input	A std::string value representing a File System command. This value is generally handed to the function by reading an input file. But may also be sent to it from another process such as a UI

4.9.2.4 run() [2/2]

```
void CLI::run ( )
```

Reads in the input data (A File System Command) and sends it down to the file system.
Some commands that do not need to interact with the File System code are handled in this function. For example, displaying the 'help' messages is executed from this function since the File System does not have or need and 'help' command. This function will block until it receives a response from the File System (provided that the command inputted is intended to go to the File System) this function will continue reading from user input until an error occurs or the user quits the application.

Reads input from user and sends it to the Liaison Process. Waits for corresponding data from the File System.

4.9.2.5 send_cmnd()

```
void CLI::send_cmnd (
            const char * cmnd )
```

Sends a command converted to a C-Style String to the Liaison Process for parsing and execution.

Parameters

command	A C-Style String of length = MaxBufferSize containing the command ID in the first X Bytes where X is the size of an integer type followed by the command itself followed by as many nullbytes as nescesarry in order to have a length = MaxBufferSize

Send user input (A filesystem command) to the Liaison Process

Parameters

cmnd	The input to send

4.9.2.6 start()

```
void CLI::start ( )
```

Performs initial set-up activities such as initiating connections and sending handshakes. Upon the completion of a successful handshake, run() is called and the interface is ready to use. If the handshake was not successful, the interface notifies the user and quits.

Run initial Command Line Interface setup operations:

1) Generate Shared Memory Segment For Process Synchronization 2) Fork And Run A Liaison Process 3) Create Sockets For Connection To Liaison 4) Send Handshake Command To File System 5) Run The Command Line

The documentation for this class was generated from the following files:

- CommandLineInterface/CLHeaders/Cli.h
- CommandLineInterface/Cli.cpp

4.10 DebugMessages Class Reference

Public Member Functions

- DebugMessages ()
- DebugMessages (std::string logfile_name)
- ~DebugMessages ()
- void ON (void)
- void OFF (void)
- template<typename T >
  void display (const T data, bool force=false)
- template<typename T >
  void log (const T data, bool force=false)
- template<typename T >
  void debug (const T data, bool force=false)
- void **lock** ()
- void **unlock** ()

4.10.1 Constructor & Destructor Documentation

4.10.1.1 DebugMessages() [1/2]

```
DebugMessages::DebugMessages ( )  [inline]
```

Create a new DebugMessage object using default logfile name: 'Arboreal.log' Automatically creates the log if it does not exist and if it does exist it will overwrite all the data in the log with the empty string. Sets the debug flag _DEBUG to FALSE on startup.

4.10.1.2 DebugMessages() [2/2]

```
DebugMessages::DebugMessages (
            std::string logfile_name )  [inline]
```

Create a new DebugMessage object using a user defined logfile name. Automatically creates the log if it does not exist and if it does exist it will overwrite all the data in the log with the empty string. Sets the debug flag _DEBUG to FALSE on startup.

4.10.1.3 ~DebugMessages()

```
DebugMessages::~DebugMessages ( )  [inline]
```

Default Destructor

4.10.2 Member Function Documentation

4.10.2.1 debug()

```
template<typename T >
void DebugMessages::debug (
            const T data,
            bool force = false )  [inline]
```

Template function for writing debug information to std::cout AND std::fstream.

Parameters

data	The data to be written to std::cout and a file. If the type of data passed is not supported by std::cout or outstream operators, behavior is undefined.
force	If data needs to be written before debugging offically starts this flag should be set to TRUE. Default value is FALSE.

4.10.2.2 display()

```
template<typename T >
void DebugMessages::display (
            const T data,
            bool force = false )  [inline]
```

Template function for writing debug information to std::cout ONLY.

Parameters

data	The data to be written to std::cout. If the type of data passed is not supported by std::cout, behavior is undefined.
force	If data needs to be written before debugging offically starts this flag should be set to TRUE. Default value is FALSE.

4.10.2.3 log()

```
template<typename T >
void DebugMessages::log (
            const T data,
            bool force = false )  [inline]
```

Template function for writing debug information to std::fstream ONLY.

Parameters

data	The data to be written to a file. If the type of data passed is not supported by outstream operators, behavior is undefined.
force	If data needs to be written before debugging offically starts this flag should be set to TRUE. Default value is FALSE.

4.10.2.4 OFF()

```
void DebugMessages::OFF (
            void  )  [inline]
```

Turns Debugging OFF Sets _DEBUG to FALSE

4.10.2.5 ON()

```
void DebugMessages::ON (
            void  )  [inline]
```

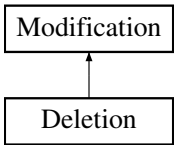
Turns Debugging ON Sets _DEBUG to TRUE

The documentation for this class was generated from the following file:

- SharedHeaders/DebugMessages.hpp

4.11 Deletion Class Reference

Inheritance diagram for Deletion:



Public Member Functions

- **Deletion** (TreeObject ∗obj, TreeObject ∗parent)
- void **write_out** (PartitionManager ∗pm)

Additional Inherited Members

The documentation for this class was generated from the following files:

- Filesystem/DaemonDependancies/Trees/Trees.h
- Filesystem/DaemonDependancies/Trees/Trees.cpp

4.12 Disk Class Reference

Public Member Functions

- Disk (BlkNumType numblocks, size_t blockSize, char ∗location)

Modifier Functions

- void writeDiskBlock (BlkNumType blknum, char ∗blkdata)

Accessor Functions

- void readDiskBlock (BlkNumType blknum, char ∗blkdata)
- size_t getBlockSize ()
- int getBlockCount ()

4.12.1 Constructor & Destructor Documentation

4.12.1.1 Disk()

```
Disk::Disk (
            BlkNumType numblocks,
            size_t blockSize,
            char * location )
```

Parameters

numblocks	the number of blocks on the Disk
blocksize	the block size for Disk blocks
location	the location of the Disk

4.12.2 Member Function Documentation

4.12.2.1 getBlockCount()

```
int Disk::getBlockCount ( )
```

Returns

the number of blocks on the entire Disk

4.12.2.2 getBlockSize()

```
size_t Disk::getBlockSize ( )
```

Returns

the blocksize of the Disk

4.12.2.3 readDiskBlock()

```
void Disk::readDiskBlock (
            BlkNumType blknum,
            char * blkdata )
```

Reads a block from the Disk.

Parameters

blknum	the blocknumber to be read
blkdata	the buffer to put the read data. must be large enough to contain an entire block of data

See also

PartitionManger::readDiskBlock() ParitionManager::readDiskBlock()

4.12.2.4 writeDiskBlock()

```
void Disk::writeDiskBlock (
            BlkNumType blknum,
            char * blkdata )
```

Writes a block to the Disk.

Parameters

blknum	the blocknumber to be written
blkdata	the buffer to write the data from. It Will write an entire block size of data.

See also

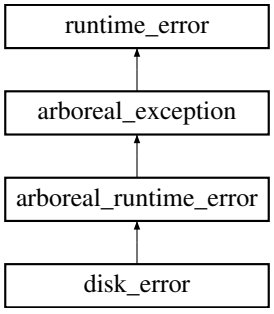
PartitionManger::writeDiskBlock() ParitionManager::writeDiskBlock()

The documentation for this class was generated from the following files:

- Filesystem/DaemonDependancies/Disk/Disk.h
- Filesystem/DaemonDependancies/Disk/Disk.cpp

4.13 disk_error Class Reference

Inheritance diagram for disk_error:



Public Member Functions

- **disk_error** (const char *what, const char *where, const int ecode=99)
- **disk_error** (const char *what, const string &where, const int ecode=99)
- **disk_error** (const string &what, const string &where, const int ecode=99)
- **disk_error** (const string &what, const char *where, const int ecode=99)

Additional Inherited Members

The documentation for this class was generated from the following files:

- SharedHeaders/Arboreal_Exceptions.h
- SharedCPPFiles/Arboreal_Exceptions.cpp

4.14 DiskManager Class Reference

Public Member Functions

- DiskManager (Disk *d)

Accessor Functions

- void readDiskBlock (string partitionName, BlkNumType blknum, char *blkdata)
- size_t getBlockSize ()
- BlkNumType getPartitionSize (string partitionName)
- DiskPartition * findPart (string partitionName)

Modifier Functions

- void writeDiskBlock (string partitionName, BlkNumType blknum, char *blkdata)

4.14.1 Constructor & Destructor Documentation

4.14.1.1 DiskManager()

```
DiskManager::DiskManager (
            Disk * d )
```

Parameters

d	Pointer to the Disk this will manage

4.14.2 Member Function Documentation

4.14.2.1 findPart()

```
DiskPartition * DiskManager::findPart (
            string partitionName )
```

Parameters

partitionName	the name of the partition

Returns

the size of a partition in blocks

4.14.2.2 getBlockSize()

```
size_t DiskManager::getBlockSize ( )
```

Returns

the blocksize of the Disk

4.14.2.3 getPartitionSize()

```
BlkNumType DiskManager::getPartitionSize (
            string partitionName )
```

Parameters

partitionName	the name of the partition

Returns

the size of a partition in blocks

4.14.2.4 readDiskBlock()

```
void DiskManager::readDiskBlock (
            string partitionName,
            BlkNumType blknum,
            char * blkdata )
```

Reads a block from the Disk.

Parameters

partitionName	the name of the partition to write the block to
blknum	the blocknumber to be read
blkdata	the buffer to put the read data. must be large enough to contain an entire block of data

See also

PartitionManger::readDiskBlock() ParitionManager::readDiskBlock()

4.14.2.5 writeDiskBlock()

```
void DiskManager::writeDiskBlock (
            string partitionName,
            BlkNumType blknum,
            char * blkdata )
```

Writes a block to the Disk.

Parameters

partitionName	the name of the partition to write the block to
blknum	the blocknumber to be written
blkdata	the buffer to write the data from. It Will write an entire block size of data.

See also

PartitionManger::writeDiskBlock() ParitionManager::writeDiskBlock()

The documentation for this class was generated from the following files:

- Filesystem/DaemonDependancies/DiskManager/DiskManager.h
- Filesystem/DaemonDependancies/DiskManager/DiskManager.cpp

4.15 DiskPartition Struct Reference

Public Attributes

- string **partitionName**

- BlkNumType **partitionSize**
- BlkNumType **partitionBlkStart**
- int **fileNameSize**

The documentation for this struct was generated from the following file:

- Filesystem/DaemonDependancies/DiskManager/DiskManager.h

4.16 File Class Reference

Public Member Functions

- File (string name, const vector< string > &tags, FileAttributes attributes)

Accessor Functions

- string get_name ()
- vector< string > & get_tags ()
- FileAttributes get_attributes ()

Static Public Member Functions

- static File * read_buff (const char *serializedFile)

4.16.1 Constructor & Destructor Documentation

4.16.1.1 File()

```
File::File (
            string name,
            const vector< string > & tags,
            FileAttributes attributes )
```

Parameters

name	the name of the File
tags	the tags to be associated with the File
attributes	the File attributes

4.16.2 Member Function Documentation

4.16.2.1 get_attributes()

```
FileAttributes File::get_attributes ( )
```

Returns

the attributes associated with this File

4.16.2.2 get_name()

```
string File::get_name ( )
```

Returns

The name of the File

4.16.2.3 get_tags()

```
vector< string > & File::get_tags ( )
```

Returns

The tags associated with this File

4.16.2.4 read_buff()

```
File * File::read_buff (
            const char * serializedFile )  [static]
```

Will take a char* buffer and create a File object from it. The buffer must have been serialized in the correct format

Parameters

serializedFile	the serializedFile object

Returns

a File* to the created File

See also

FileInfo::serialize()

The documentation for this class was generated from the following files:

- Filesystem/DaemonDependancies/File/File.h
- Filesystem/DaemonDependancies/File/File.cpp

4.17 file_attributes Struct Reference

Public Attributes

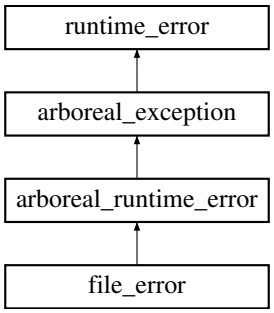
- time_t **creationTime**
- time_t **lastAccess**
- time_t **lastEdit**
- size_t **size**
- char **permissions** [12]
- int **owner**

The documentation for this struct was generated from the following file:

- Filesystem/DaemonDependancies/Types/types.h

4.18 file_error Class Reference

Inheritance diagram for file_error:



Public Member Functions

- **file_error** (const char ∗what, const char ∗where, const int ecode=99)
- **file_error** (const char ∗what, const string &where, const int ecode=99)
- **file_error** (const string &what, const string &where, const int ecode=99)
- **file_error** (const string &what, const char ∗where, const int ecode=99)

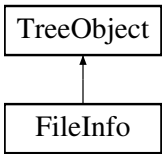
Additional Inherited Members

The documentation for this class was generated from the following files:

- SharedHeaders/Arboreal_Exceptions.h
- SharedCPPFiles/Arboreal_Exceptions.cpp

4.19 FileInfo Class Reference

Inheritance diagram for FileInfo:



Public Member Functions

- FileInfo (string filename, BlkNumType blknum, PartitionManager ∗pm)
- void write_out ()
- void read_in (unordered_multimap< string, FileInfo ∗> ∗allFiles, RootTree ∗rootTree)
- void erase (string name)
- void insert (string name, TreeObject ∗ptr)
- void del ()
- void delete_cont_blocks (BlkNumType blknum)
- void insert_addition (TreeObject ∗add)
- void insert_deletion (TreeObject ∗del)

Accessor Functions

- string mangle ()
  mangles the filename with its tags
- string mangle (vector< string > &tags)
  mangles the filename with the specified tags
- string mangle (unordered_set< string > &tags)
  mangles the filename with the specified tags
- Finode get_finode ()
- size_t get_file_size ()
- Attributes ∗ get_attributes ()
- FileAttributes **get_file_attributes** ()
- unordered_set< string > get_tags ()
- vector< string > **get_vec_tags** ()

Modifier Functions

- void add_direct_block (BlkNumType blknum, int index)
- void add_indirect_block (BlkNumType blknum, short level)
- void update_file_size (size_t bytes)
- void set_access ()
- void set_edit ()
- void set_permissions (char ∗perms)
  sets the permisssions for this file

Static Public Member Functions

- static string * serialize (FileInfo *file)

Additional Inherited Members

4.19.1 Constructor & Destructor Documentation

4.19.1.1 FileInfo()

```
FileInfo::FileInfo (
            string filename,
            BlkNumType blknum,
            PartitionManager * pm )
```

Parameters

filename	Name of the File
blknum	the blocknumber of the associated Finode on disk

4.19.2 Member Function Documentation

4.19.2.1 add_direct_block()

```
void FileInfo::add_direct_block (
            BlkNumType blknum,
            int index )
```

adds the specified blocknumber to the array of direct blocks in this file's Finode

Parameters

blknum	the block number of the direct block that has already been allocated
index	the index of the blknum in the array, must be less than 12 and at least 0.

Exceptions

arboreal_logic_error	index out of bounds

See also

add_indirect_block

4.19.2.2 add_indirect_block()

```
void FileInfo::add_indirect_block (
            BlkNumType blknum,
            short level )
```

adds the specified blocknumber to the Finode as the start of the specified level of indirect blocks

Parameters

blknum	the block number of the indirect block that has already been allocated
level	the level that the block number is associated with. must be 1, 2 or 3.

Exceptions

arboreal_logic_error	Invalid level

See also

add_direct_block

4.19.2.3 del()

```
void FileInfo::del ( )  [virtual]
```

Will completely remove the TreeObject's presence on disk

Implements TreeObject.

4.19.2.4 delete_cont_blocks()

```
void FileInfo::delete_cont_blocks (
            BlkNumType blknum )  [virtual]
```

Will follow the chain of continuation blocks and free all of them

Parameters

blknum	will free the blknum and use it to follow the chain of continuation blocks

Reimplemented from TreeObject.

4.19.2.5 erase()

```
void FileInfo::erase (
            string name )  [virtual]
```

Disassociate the given name from this object

Parameters

name	the name of the object to be erased.

Exceptions

arboreal_logic_error	

Reimplemented from TreeObject.

4.19.2.6 get_attributes()

```
Attributes * FileInfo::get_attributes ( )
```

Returns

the Attributes accociated with this file

4.19.2.7 get_file_size()

```
size_t FileInfo::get_file_size ( )
```

Returns

the size of this file in bytes

4.19.2.8 get_finode()

```
Finode FileInfo::get_finode ( )
```

Returns

the Finode associated with this file

4.19.2.9 get_tags()

```
unordered_set< string > FileInfo::get_tags ( )
```

Returns

The tags associated with this file

4.19.2.10 insert()

```
void FileInfo::insert (
            string name,
            TreeObject * obj )  [virtual]
```

Associate a TreeObject with this object

Parameters

name	name of the object, mangled if inserting a FileInfo
obj	the object to be inserted

Exceptions

tag_error	

See also

FileInfo::insert()

Reimplemented from TreeObject.

4.19.2.11 insert_addition()

```
void FileInfo::insert_addition (
            TreeObject * add )  [virtual]
```

Do not call on FileInfo

Reimplemented from TreeObject.

4.19.2.12 insert_deletion()

```
void FileInfo::insert_deletion (
            TreeObject * del )  [virtual]
```

Do not call on FileInfo

Reimplemented from TreeObject.

4.19.2.13 mangle() [1/3]

```
string FileInfo::mangle ( )
```

mangles the filename with its tags

The name is mangled as follows: Each tag is placed in alphabetical order and appended to the filename using '_' as the seperator.

Returns

the mangled name of this file.

See also

mangle(vector<string>&) mangle(unordered_set<string>& tags)

4.19.2.14 mangle() [2/3]

```
string FileInfo::mangle (
            vector< string > & tags )
```

mangles the filename with the specified tags

The name is mangled as follows: Each tag is placed in alphabetical order and appended to the filename using '_' as the seperator.

Returns

the mangled name of this file.

Parameters

tags	the tags you wish to mangle the filename with

See also

mangle() mangle(unordered_set<string>& tags)

4.19.2.15 mangle() [3/3]

```
string FileInfo::mangle (
            unordered_set< string > & tags )
```

mangles the filename with the specified tags

) The name is mangled as follows: Each tag is placed in alphabetical order and appended to the filename using '_' as the seperator.

Returns

the mangled name of this file.

Parameters

tags	the tags you wish to mangle the filename with

See also

mangle() mangle(unordered_set<string>& tags

4.19.2.16 read_in()

```
void FileInfo::read_in (
            unordered_multimap< string, FileInfo *> * allFiles,
            RootTree * rootTree )  [virtual]
```

Will read in all object data from disk

Parameters

allFiles	a pointer to the map of all files
rootTree	a pointer to the root tree

Implements TreeObject.

4.19.2.17 serialize()

```
string * FileInfo::serialize (
            FileInfo * file )  [static]
```

Will serialize a FileInfo object such that it can be read in as a File object

Parameters

file	the FileInfo object to be serialized

Returns

The serialized object in string form

See also

File::read_buff()

4.19.2.18 set_access()

```
void FileInfo::set_access ( )
```

marks the file as accessed at the current UNIX time

4.19.2.19 set_edit()

```
void FileInfo::set_edit ( )
```

marks the file as edited at the current UNIX time

4.19.2.20 set_permissions()

```
void FileInfo::set_permissions (
            char * perms )
```

sets the permisssions for this file

The permisssions format is as follows. a 1 for true 0 false Byte 0, 1, 2 : reserved, for now Byte 3 - 5 : read write and execute permisssions for the user Byte 6 - 8 : read write and execute permisssions for the group Byte 9 - 11 : read write and execute permisssions for the world Currently there is no differentiation between user group and world

Parameters

perms	the permisssions in the correct format

4.19.2.21 update_file_size()

```
void FileInfo::update_file_size (
            size_t bytes )
```

Sets the file size to the specified bytes. Only the filesystem should call.

Parameters

bytes	the new file size

4.19.2.22 write_out()

```
void FileInfo::write_out ( )  [virtual]
```

Intended to write out the object to disk

Implements TreeObject.

The documentation for this class was generated from the following files:

- Filesystem/DaemonDependancies/Trees/Trees.h
- Filesystem/DaemonDependancies/Trees/Trees.cpp

4.20 FileOpen Class Reference

Public Member Functions

- **FileOpen** (FileInfo *file, char mode, PartitionManager *pm)
- FileInfo * **get_file** ()
- size_t **get_seek** ()
- char **get_mode** ()
- void **increment_seek** (size_t bytes, bool write=false)
- void **decrement_seek** (size_t bytes)
- Index **byte_to_index** (short offset)
- Index **increment_index** ()
- void **set_EOF** ()
- void **reset_seek** ()
- bool **get_EOF** ()
- void **go_past_last_byte** ()
- void **refresh** ()

The documentation for this class was generated from the following files:

- Filesystem/DaemonDependancies/FileSystem/FileSystem.h
- Filesystem/DaemonDependancies/FileSystem/FileSystem.cpp

4.21 FileSystem Class Reference

Public Member Functions

- FileSystem (DiskManager *dm, string partitionName)
- void write_changes ()
- FileInfo * path_to_file (vector< string > &path)
- int get_file_name_size ()
- size_t **num_of_files** ()
- size_t **num_of_tags** ()

Tag Operations

- vector< FileInfo * > * tag_search (unordered_set< string > &tags)
- void create_tag (string tagName)
- void delete_tag (string tagName)
- void merge_tags (string tag1, string tag2)
- void tag_file (FileInfo *file, unordered_set< string > tagsToAdd)
- void tag_file (vector< string > &filePath, unordered_set< string > tags)
- void untag_file (FileInfo *file, unordered_set< string > tagsToRemove, bool deleting=false)
- void untag_file (vector< string > &filePath, unordered_set< string > tags)
- void rename_tag (string originalTagName, string newTagName)

File Operations

- vector< FileInfo * > * file_search (string name)
- FileInfo * create_file (string filename, unordered_set< string > &tags)
- int open_file (vector< string > &filePath, char mode)
- void close_file (unsigned int fileDesc)
- size_t read_file (unsigned int fileDesc, char *data, size_t len)
- size_t write_file (unsigned int fileDesc, const char *data, size_t len)
- size_t append_file (unsigned int fileDesc, const char *data, size_t len)
- void seek_file_absolute (unsigned int fileDesc, size_t offset)
- void seek_file_relative (unsigned int fileDesc, long int offset)
- void rename_file (vector< string > &originalFilePath, string newFileName)
- Attributes * get_attributes (vector< string > &filePath)
- void set_permissions (vector< string > &filePath, char *perms)
- void delete_file (FileInfo *file)
- void delete_file (vector< string > &filePath)

Debug Functions

- void print_root ()
- void print_tags ()
- void print_files ()

4.21.1 Constructor & Destructor Documentation

4.21.1.1 FileSystem()

```
FileSystem::FileSystem (
            DiskManager * dm,
            string partitionName )
```

Parameters

dm	the Disk manager for the disk that this Filesystem will be accessing
partitionName	the name of the partition that this FileSystem will be associated with

4.21.2 Member Function Documentation

4.21.2.1 append_file()

```
size_t FileSystem::append_file (
            unsigned int fileDesc,
            const char * data,
            size_t len )
```

Will Append a number of bytes to an open file. The file must have been opened with write permissions.

Parameters

fileDesc	the file descriptor returned from open_file
data	a buffer to be read from to write to the file. must be at least of len size
len	the number of bytes to write from data.

4.21.2.2 close_file()

```
void FileSystem::close_file (
            unsigned int fileDesc )
```

Will close a file. the File must have been opened.

Parameters

fileDesc	the file descriptor returned from open_file

4.21.2.3 create_file()

```
FileInfo * FileSystem::create_file (
            string filename,
            unordered_set< string > & tags )
```

Will create a new file with the specified name and tags. The new file must not already exist.

Parameters

filename	the name of the new file
tags	the tag set to tag the file with. If empty, will be tagged with default.

Returns

a FileInfo to the created file, in case the calling code needs it

4.21.2.4 create_tag()

```
void FileSystem::create_tag (
            string tagName )
```

Will create a new tag if that tag name does not already exist

Parameters

tagName	the name of the Tag to create

4.21.2.5 delete_file() [1/2]

```
void FileSystem::delete_file (
            FileInfo * file )
```

Delete a particular file. The file must exist.

Parameters

file	the FileInfo object to be deleted.

See also

delete_file(vector<string>&)

4.21.2.6 delete_file() [2/2]

```
void FileSystem::delete_file (
            vector< string > & filePath )
```

Delete a particular file. The file must exist.

Parameters

filePath	the full path to the file to you wish to delete

See also

delete_file(FileInfo*)

4.21.2.7 delete_tag()

```
void FileSystem::delete_tag (
            string tagName )
```

Will delete the specified tag only if it has no files associated with it(it is empty) and it does in fact exist.

Parameters

tagName	the name of the tag to be deleted

4.21.2.8 file_search()

```
vector< FileInfo * > * FileSystem::file_search (
            string name )
```

Will search for a specified file name. Searches the entire FileSystem

Parameters

name	the name of the file to search for.

Returns

a pointer to a vector of FileInfo objects that have the specified name. This should be freed by the calling code

4.21.2.9 get_attributes()

```
Attributes * FileSystem::get_attributes (
            vector< string > & filePath )
```

Will search for a file and return its Attributes

Parameters

filePath	the full path to the file to you wish to get the Attributes of

Returns

the Attributes object associated with a particular file.

4.21.2.10 get_file_name_size()

```
int FileSystem::get_file_name_size ( )
```

Returns

the Maximum file name size for the partition associated with this FileSystem object

4.21.2.11 merge_tags()

```
void FileSystem::merge_tags (
            string tag1,
            string tag2 )
```

TODO: description and Function

Parameters

tag1	
tag	2

4.21.2.12 open_file()

```
int FileSystem::open_file (
            vector< string > & filePath,
            char mode )
```

Will open a file. The file must exist. There is a cap on the Maximum number of open files. You can open the same file as many times as you want.

Parameters

filePath	the full path including the file name as the last entry
mode	the mode to open the file with. r, w, or x. x is read and write ability.

Returns

a file descriptor that can later be used to reference the opened file

4.21.2.13 path_to_file()

```
FileInfo * FileSystem::path_to_file (
            vector< string > & path )
```

Will find a FileInfo object if it exists, given the full path

Parameters

path	The full path to the file. The filename must be the last entry in the vector. an file name with no path is considered to be in the default path

Returns

the found FileInfo object

4.21.2.14 print_files()

```
void FileSystem::print_files ( )
```

Print out all files and their blocknumbers

4.21.2.15 print_root()

```
void FileSystem::print_root ( )
```

Print out the root Tree

4.21.2.16 print_tags()

```
void FileSystem::print_tags ( )
```

Print out all the tag trees and their contents

4.21.2.17 read_file()

```
size_t FileSystem::read_file (
            unsigned int fileDesc,
            char * data,
            size_t len )
```

Will read a number of bytes from an open file. The file must have been opened with read permissions. If you read past the end of the file, EOF will be tripped and you cannot continue reading. will return all the data up to that point

Parameters

fileDesc	the file descriptor returned from open_file
data	a buffer to store the read data must be at least len size
len	the number of bytes to read.

4.21.2.18 rename_file()

```
void FileSystem::rename_file (
            vector< string > & originalFilePath,
            string newFileName )
```

Will rename a file. The new file must not already exist in the emulated directory

Parameters

originalFilePath	the full path to the file to be renamed
newFileName	the name that the file will be renamed to.

4.21.2.19 rename_tag()

```
void FileSystem::rename_tag (
            string originalTagName,
            string newTagName )
```

Will rename the tag. The tag must exist. The new tag name must already exist. This is a slow operation.

Parameters

originalTagName	the name of the tag to be renamed
newTagName	the new tag name for that tag

4.21.2.20 seek_file_absolute()

```
void FileSystem::seek_file_absolute (
            unsigned int fileDesc,
            size_t offset )
```

Seek to an absolute position in the file. Will trip EOF if the offset is larger than the file size. The posistion in the file is indexed at 1.

Parameters

fileDesc	the file descriptor returned from open_file
offset	the absolute position in the file to seek to.

4.21.2.21 seek_file_relative()

```
void FileSystem::seek_file_relative (
            unsigned int fileDesc,
            long int offset )
```

Seek to a relative position in the file. Will trip EOF if you try to seek too far in a direction. The posistion in the file is indexed at 1.

Parameters

fileDesc	the file descriptor returned from open_file
offset	the relative position in the file to seek to. may be a negative number.

4.21.2.22 set_permissions()

```
void FileSystem::set_permissions (
            vector< string > & filePath,
            char * perms )
```

Set the permissions for a file. The format is defined in FileInfo.

Parameters

filePath	the full path to the file to you wish to get the Attributes of
perms	the permissions following the correct format to set to this file

See also

FileInfo::set_permissions()

4.21.2.23 tag_file() [1/2]

```
void FileSystem::tag_file (
            FileInfo * file,
            unordered_set< string > tagsToAdd )
```

Will tag a file with the specified tags. If some or all of the tags do not exist, a warning is printed and the operation continues. The file must exist. The file that would be created by adding tags must not already exist.

Parameters

file	the FileInfo* that will be tagged with the specified tags
tagsToAdd	the tags that will be added to the file's tag set

See also

tag_file(vector<string>&, unordered_set<string>)

4.21.2.24 tag_file() [2/2]

```
void FileSystem::tag_file (
            vector< string > & filePath,
            unordered_set< string > tags )
```

An alternate way to tag a file using a file path instead.Will tag a file with the specified tags. If some or all of the tags do not exist, a warning is printed and the operation continues. The file must exist. The file that would be created by adding tags must not already exist.

Parameters

filePath	the FileInfo* that will be tagged with the specified tags
tagsToAdd	the tags that will be added to the file's tag set

See also

tag_file(FileInfo*, unordered_set<string>)

4.21.2.25 tag_search()

```
vector< FileInfo * > * FileSystem::tag_search (
            unordered_set< string > & tags )
```

Search for files by tags. The tag search is an "and" operation, meaning the files returned will have at least all the specified tags.

Parameters

tags	the tags that the files will be tagged with in the return vector

Returns

a pointer to a vector of the FileInfo objects which then can be serialized. The returned vector should be freed by the calling code

4.21.2.26 untag_file() [1/2]

```
void FileSystem::untag_file (
            FileInfo * file,
            unordered_set< string > tagsToRemove,
            bool deleting = false )
```

Will remove tags associated with the specified file. The tags must exist. The file must exist. The file that would be created by removing tags must not already exist.

Parameters

file	the FileInfo* that will be untagged with the specified tags
tagsToRemove	the tags that will be removed from the file's tag set
deleting	this is a tag only used by the FileSystem itself for deleting a file

See also

tag_file(FileInfo*, unordered_set<string>)

4.21.2.27 untag_file() [2/2]

```
void FileSystem::untag_file (
            vector< string > & filePath,
            unordered_set< string > tags )
```

Will remove tags associated with the specified file. The tags must exist. The file must exist. The file that would be created by removing tags must not already exist.

Parameters

file	the FileInfo* that will be untagged with the specified tags
tagsToRemove	the tags that will be removed from the file's tag set
deleting	this is a tag only used by the FileSystem itself for deleting a file

See also

tag_file(FileInfo*, unordered_set<string>)

4.21.2.28 write_changes()

```
void FileSystem::write_changes ( )
```

Since the FileSystem is journaling. The changes to tag trees and the Root tree are only written out when this is called. File Operations are not journaled.

4.21.2.29 write_file()

```
size_t FileSystem::write_file (
            unsigned int fileDesc,
            const char * data,
            size_t len )
```

Will write a number of bytes to an open file. The file must have been opened with write permissions. You can write past the EOF with no problems.

Parameters

fileDesc	the file descriptor returned from open_file
data	a buffer to be read from to write to the file. must be at least of len size
len	the number of bytes to write from data.

The documentation for this class was generated from the following files:

- Filesystem/DaemonDependancies/FileSystem/FileSystem.h
- Filesystem/DaemonDependancies/FileSystem/FileSystem.cpp

4.22 finode Struct Reference

Public Attributes

- BlkNumType **attributes**
- BlkNumType **directBlocks** [12]
- BlkNumType **level1Indirect**
- BlkNumType **level2Indirect**
- BlkNumType **level3Indirect**

The documentation for this struct was generated from the following file:

- Filesystem/DaemonDependancies/Types/types.h

4.23 index Struct Reference

Public Attributes

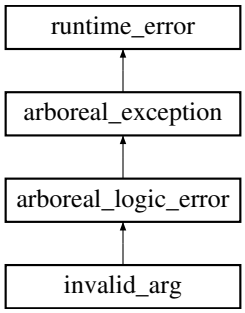
- BlkNumType **blknum**
- size_t **offset**

The documentation for this struct was generated from the following file:

- Filesystem/DaemonDependancies/Types/types.h

4.24 invalid_arg Class Reference

Inheritance diagram for invalid_arg:



Public Member Functions

- **invalid_arg** (const char ∗what, const char ∗where, const int ecode=99)
- **invalid_arg** (const char ∗what, const string &where, const int ecode=99)
- **invalid_arg** (const string &what, const string &where, const int ecode=99)
- **invalid_arg** (const string &what, const char ∗where, const int ecode=99)

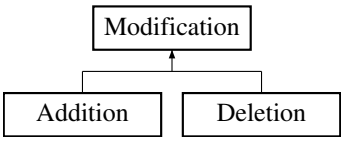
Additional Inherited Members

The documentation for this class was generated from the following files:

- SharedHeaders/Arboreal_Exceptions.h
- SharedCPPFiles/Arboreal_Exceptions.cpp

4.25 Modification Class Reference

Inheritance diagram for Modification:



Public Member Functions

- virtual void **write_out** (PartitionManager ∗pm)=0

Protected Member Functions

- **Modification** (TreeObject ∗obj, TreeObject ∗parent)

Protected Attributes

- TreeObject * **_mod**
- TreeObject * **_parent**

The documentation for this class was generated from the following files:

- Filesystem/DaemonDependancies/Trees/Trees.h
- Filesystem/DaemonDependancies/Trees/Trees.cpp

4.26 ParseError Class Reference

Public Member Functions

- ParseError (const char *where, const char *what)
- std::string where ()
- std::string what ()

4.26.1 Constructor & Destructor Documentation

4.26.1.1 ParseError()

```
ParseError::ParseError (
            const char * where,
            const char * what )  [inline]
```

Parameters

where	Where the parse error took place
what	What the parse error consisted of

4.26.2 Member Function Documentation

4.26.2.1 what()

```
std::string ParseError::what ( )  [inline]
```

Returns

A std::string detailing what the parse error consisted of

4.26.2.2 where()

```
std::string ParseError::where ( )  [inline]
```

Returns

A std::string detailing where the parse error occured

The documentation for this class was generated from the following file:

- SharedHeaders/Parser.h

4.27 Parser Class Reference

Public Member Functions

- Parser (char *buffer, char *cwd, int max_name_size)
- Parser (std::string string, std::string cwd, int max_name_size)
- Parser (const char *string_lit, const char *cwd, int max_name_size)
- Parser ()
- ~Parser ()
- void reset (std::string string, std::string cwd="")

  Changes the member _string of the parser class to whatever is passed.
- void reset (char *buffer, char *cwd=NULL)

  Changes the member _string of the parser class to whatever is passed.
- void reset (const char *string_lit, const char *cwd="")

  Changes the member _string of the parser class to whatever is passed.
- void set_max_name_size (int size)

  Sets the maximum allowed file and tagname size that the Parser will use.
- void set_cwd (std::string cwd)

  Sets the Current Working Directory that the Parser will use.
- std::vector< std::string > parse (int type)

  Parse a string based on a certain rule.
- std::vector< std::string > get_cwd_tags ()

  Returns a vector representation of the current working directory.

Static Public Member Functions

- static std::vector< std::string > split_on_delim (std::string string, char delim)

  Splits a string at each instance of a particualar char (the delimeter)

4.27.1 Constructor & Destructor Documentation

4.27.1.1 Parser() [1/4]

```
Parser::Parser (
            char * buffer,
            char * cwd,
            int max_name_size )
```

Parameters

buffer	A C-Style String representation of the string to be parsed
cwd	A C-Style String representation of the current working directory; (This value is typically provided by the Liaison process). The directory string is used to parse commands which act within directories only thus providing commands such as 'tag' a "path" to the file(s) which will be tagged without the user having to explicitly enter those file's entire paths themselves.
max_name_size	The maximum length that a file or tagname is allowed to have; (This value is typically provided by the Liaison process)

4.27.1.2 Parser() [2/4]

```
Parser::Parser (
            std::string string,
            std::string cwd,
            int max_name_size )
```

Parameters

buffer	A std::string representation of the string to be parsed
cwd	A std::string representation of the current working directory; (This value is typically provided by the Liaison process). The directory string is used to parse commands which act within directories only thus, providing commands such as 'tag' a "path" to the file(s) which will be tagged without the user having to explicitly enter those file's entire paths themselves.
max_name_size	The maximum length that a file or tagname is allowed to have; (This value is typically provided by the Liaison process)

4.27.1.3 Parser() [3/4]

```
Parser::Parser (
            const char * string_lit,
            const char * cwd,
            int max_name_size )
```

Parameters

buffer	A String Literal representation of the string to be parsed
cwd	A String Literal representation of the current working directory; (This value is typically provided by the Liaison process). The directory string is used to parse commands which act within directories only thus, providing commands such as 'tag' a "path" to the file(s) which will be tagged without the user having to explicitly enter those file's entire paths themselves.
max_name_size	The maximum length that a file or tagname is allowed to have; (This value is typically provided by the Liaison process)

4.27.1.4 Parser() [4/4]

```
Parser::Parser ( )
```

Default Constructor to be used in case initialization of values needs to be done elsewhere

4.27.1.5 ~Parser()

```
Parser::~Parser ( )
```

Default Destructor; Does nothing

4.27.2 Member Function Documentation

4.27.2.1 get_cwd_tags()

```
std::vector< std::string > Parser::get_cwd_tags ( )
```

Returns a vector representation of the current working directory.

That is, it will decompose '/string1/string2' into a vector containing [string1, string2]. This is useful when the calling code requires the current working directory as a vector of strings rather than as a standard string representation.

Returns

A std::vector of std::string comprised of the non-'/' parts of the Parser member value _cwd

4.27.2.2 parse()

```
std::vector< std::string > Parser::parse (
            int type )
```

Parse a string based on a certain rule.

The rule generally corresponds to how a CLI command should be decomposed.
For example the CLI command for finding files takes a list of files, hower the filesystem itself does not support batch commands, therefore, the Parser will decompose the command into its constituent parts (i.e. a single file).
This particular behavior is access by passing '8' as the "type" of decomposition that needs to take place (Note that this corresponds to the command's ID).
However the Parser can be extended to support any rule whatsoever, so long as it is added to the Parser's parse() function switch statement.

Parameters

type	The integer identification of the parse rule that will be executed

Returns

A std::vector of std::string comprised of the result after the chosen parse rule is executed.

4.27.2.3 reset() [1/3]

```
void Parser::reset (
            std::string string,
            std::string cwd = "" )
```

Changes the member _string of the parser class to whatever is passed.

The Parser class conducts all operations on its member _string rather than requiring that a string value be passed to its parse() method. This was done in order to make use of the class as streamlined as possible.

Parameters

string	A std::string representation of the string to be parsed
cwd	A std::string representation of the current working directory; Note that this argument is optional and allows the user to both reset the string the Parser will work with as well as the directory string the Parser will use. The directory string is used to parse commands which act within directories only thus providing commands such as 'tag' a "path" to the file(s) which will be tagged without the user having to explicitly enter those file's entire paths themselves.

4.27.2.4 reset() [2/3]

```
void Parser::reset (
            char * buffer,
            char * cwd = NULL )
```

Changes the member _string of the parser class to whatever is passed.

The Parser class conducts all operations on its member _string rather than requiring that a string value be passed to its parse() method. This was done in order to make use of the class as streamlined as possible.

Parameters

string	A C-Style String representation of the string to be parsed
cwd	A C-Style String representation of the current working directory; Note that this argument is optional and allows the user to both reset the string the Parser will work with as well as the directory string the Parser will use. The directory string is used to parse commands which act within directories only thus providing commands such as 'tag' a "path" to the file(s) which will be tagged without the user having to explicitly enter those file's entire paths themselves.

Returns

Void

4.27.2.5 reset() [3/3]

```
void Parser::reset (
            const char * string_lit,
            const char * cwd = "" )
```

Changes the member _string of the parser class to whatever is passed.

The Parser class conducts all operations on its member _string rather than requiring that a string value be passed to its parse() method. This was done in order to make use of the class as streamlined as possible.

Parameters

string	A String Literal representation of the string to be parsed
cwd	A String Literal representation of the current working directory; Note that this argument is optional and allows the user to both reset the string the Parser will work with as well as the directory string the Parser will use. The directory string is used to parse commands which act within directories only thus providing commands such as 'tag' a "path" to the file(s) which will be tagged without the user having to explicitly enter those file's entire paths themselves.

Returns

Void

4.27.2.6 set_cwd()

```
void Parser::set_cwd (
            std::string cwd )
```

Sets the Current Working Directory that the Parser will use.

The directory string is used to parse commands which act within directories only thus providing commands such as 'tag' a "path" to the file(s) which will be tagged without the user having to explicitly enter those file's entire paths themselves. This function does not have counterparts which tahe C-Style Strings or String Literals. This is because, in all situations, if the current working directory must be set using this method, it is highly likely that the calling code has a std::string representation of the current working directory rather than a representation in one of the other formats. If such functionality (C-Style Strings and others) is desired, extensibility is easy enough. Regardless the Parser's _cwd member will always be a std::string.

Parameters

cwd	A std::string representation of the current working directory

Returns

Void

4.27.2.7 set_max_name_size()

```
void Parser::set_max_name_size (
            int size )
```

Sets the maximum allowed file and tagname size that the Parser will use.

If this size is exceeded an error is thrown and the Parser will stop its current activities. This value is dictated by the CLI and is generally provided to the Parser by the Liaison Process.

Parameters

size	The maximum file/tag name length

4.27.2.8 split_on_delim()

```
std::vector< std::string > Parser::split_on_delim (
            std::string string,
            char delim )  [static]
```

Splits a string at each instance of a particualar char (the delimeter)

The delimeters are NOT included anywhere in the resulting vector. This function is static and is mainly used outside the Parser in order to split values that the parser returned. This can happen because the complexity of certain commands does not allow the parser to fully decompose the string and instead it can only reorganize the command into a form which can be easily split later. It is important to note that this function does not differentiate between the number of delimeter charcters the string contains. That is, it will read the whole string and split it at any point where the delimeter is seen whether it is seen in 1 or 100 places.

Parameters

string	A std::string representation of whatever string needs to be split
delim	A char value representing where the string should be split

The documentation for this class was generated from the following files:

- SharedHeaders/Parser.h
- SharedCPPFiles/Parser.cpp

4.28 PartitionManager Class Reference

Public Member Functions

- PartitionManager (DiskManager *dm, string partitionName)

Accessor Functions

- void readDiskBlock (BlkNumType blknum, char *blkdata)
- size_t getBlockSize ()
- string getPartitionName ()
- int get_file_name_size ()

Modifier Functions

- void writeDiskBlock (BlkNumType blknum, char *blkdata)
- BlkNumType getFreeDiskBlock ()
- void returnDiskBlock (BlkNumType blknum)

4.28.1 Constructor & Destructor Documentation

4.28.1.1 PartitionManager()

```
PartitionManager::PartitionManager (
            DiskManager * dm,
            string partitionName )
```

Parameters

dm	the DiskManager associated with this object
partitionName	the name of the partition that this will be managing

4.28.2 Member Function Documentation

4.28.2.1 get_file_name_size()

```
int PartitionManager::get_file_name_size ( )
```

Returns

The maximum file name size for this partition in bytes

4.28.2.2 getBlockSize()

```
size_t PartitionManager::getBlockSize ( )
```

Returns

the blocksize of the Disk

4.28.2.3 getFreeDiskBlock()

```
BlkNumType PartitionManager::getFreeDiskBlock ( )
```

Allocates a block on disk if there is a free one. The Disk free list is updated accordingly

Returns

the block number of the newly allocated block

4.28.2.4 getPartitionName()

```
string PartitionManager::getPartitionName ( )
```

Returns

The name of the partition this PartitionManager is associated with

4.28.2.5 readDiskBlock()

```
void PartitionManager::readDiskBlock (
            BlkNumType blknum,
            char * blkdata )
```

Reads a block from the Disk.

Parameters

blknum	the blocknumber to be read
blkdata	the buffer to put the read data. must be large enough to contain an entire block of data

4.28.2.6 returnDiskBlock()

```
void PartitionManager::returnDiskBlock (
            BlkNumType blknum )
```

returns a block to the Disk free list and zeros it out before writing.

Parameters

blknum	the blocknumber of the block to be freed

4.28.2.7 writeDiskBlock()

```
void PartitionManager::writeDiskBlock (
            BlkNumType blknum,
            char * blkdata )
```

Writes a block to the Disk.

Parameters

blknum	the blocknumber to be written
blkdata	the buffer to write the data from. It Will write an entire block size of data.

The documentation for this class was generated from the following files:

- Filesystem/DaemonDependancies/PartitionManager/PartitionManager.h
- Filesystem/DaemonDependancies/PartitionManager/PartitionManager.cpp

4.29 rootSuperBlock Struct Reference

Public Attributes

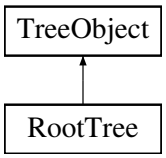
- size_t **size**
- Index **lastEntry**
- BlkNumType **startBlock**

The documentation for this struct was generated from the following file:

- Filesystem/DaemonDependancies/Types/types.h

4.30 RootTree Class Reference

Inheritance diagram for RootTree:



Public Member Functions

- RootTree (PartitionManager *pm)
- void write_out ()
- void read_in (unordered_multimap< string, FileInfo *> *allFiles, RootTree *rootTree)
- void del ()

Additional Inherited Members

4.30.1 Constructor & Destructor Documentation

4.30.1.1 RootTree()

```
RootTree::RootTree (
            PartitionManager * pm )
```

Parameters

pm	the PartitionManager to be associated with the RootTree

4.30.2 Member Function Documentation

4.30.2.1 del()

```
void RootTree::del ( )  [virtual]
```

Will completely remove the TreeObject's presence on disk

Implements TreeObject.

4.30.2.2 read_in()

```
void RootTree::read_in (
            unordered_multimap< string, FileInfo *> * allFiles,
            RootTree * rootTree )  [virtual]
```

Will read in all object data from disk

Parameters

allFiles	a pointer to the map of all files
rootTree	a pointer to the root tree

Implements TreeObject.

4.30.2.3 write_out()

```
void RootTree::write_out ( )  [virtual]
```

Intended to write out the object to disk

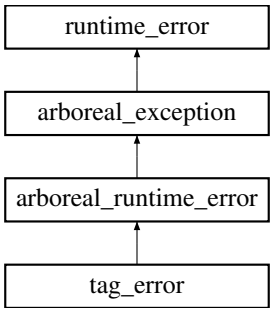
Implements TreeObject.

The documentation for this class was generated from the following files:

- Filesystem/DaemonDependancies/Trees/Trees.h
- Filesystem/DaemonDependancies/Trees/Trees.cpp

4.31 tag_error Class Reference

Inheritance diagram for tag_error:



Public Member Functions

- **tag_error** (const char *what, const char *where, const int ecode=99)
- **tag_error** (const char *what, const string &where, const int ecode=99)
- **tag_error** (const string &what, const string &where, const int ecode=99)
- **tag_error** (const string &what, const char *where, const int ecode=99)

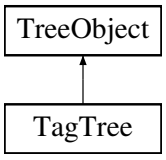
Additional Inherited Members

The documentation for this class was generated from the following files:

- SharedHeaders/Arboreal_Exceptions.h
- SharedCPPFiles/Arboreal_Exceptions.cpp

4.32 TagTree Class Reference

Inheritance diagram for TagTree:



Public Member Functions

- TagTree (string tagName, BlkNumType blknum, PartitionManager *pm)
- void write_out ()
- void read_in (unordered_multimap< string, FileInfo *> *allFiles, RootTree *rootTree)
- void del ()

Additional Inherited Members

4.32.1 Constructor & Destructor Documentation

4.32.1.1 TagTree()

```
TagTree::TagTree (
            string tagName,
            BlkNumType blknum,
            PartitionManager * pm )
```

Parameters

tagName	the name of this tag
blknum	the blocknumber for the superblock of this tagTree

4.32.2 Member Function Documentation

4.32.2.1 del()

```
void TagTree::del ( )  [virtual]
```

Will completely remove the TreeObject's presence on disk

Implements TreeObject.

4.32.2.2 read_in()

```
void TagTree::read_in (
            unordered_multimap< string, FileInfo *> * allFiles,
            RootTree * rootTree )  [virtual]
```

Will read in all object data from disk

Parameters

allFiles	a pointer to the map of all files
rootTree	a pointer to the root tree

Implements TreeObject.

4.32.2.3 write_out()

```
void TagTree::write_out ( )  [virtual]
```

Intended to write out the object to disk

Implements TreeObject.

The documentation for this class was generated from the following files:

- Filesystem/DaemonDependancies/Trees/Trees.h
- Filesystem/DaemonDependancies/Trees/Trees.cpp

4.33 tagTreeSuperBlock Struct Reference

Public Attributes

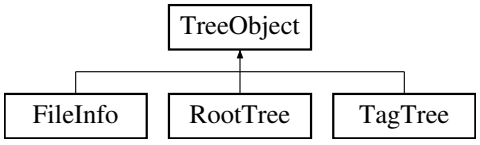
- size_t **size**
- Index **lastEntry**
- BlkNumType **startBlock**

The documentation for this struct was generated from the following file:

- Filesystem/DaemonDependancies/Types/types.h

4.34 TreeObject Class Reference

Inheritance diagram for TreeObject:



Public Member Functions

- TreeObject (string name, BlkNumType blknum, PartitionManager *pm)

Accessor Functions

- string get_name () const
- BlkNumType get_block_number () const
- Index get_index (TreeObject *obj) const
- Index get_last_entry () const
- BlkNumType get_start_block () const
- size_t size () const
- unordered_map< string, TreeObject * >::iterator begin ()
- unordered_map< string, TreeObject * >::iterator end ()
- TreeObject * find (string name) const
- queue< Index > * get_free_spots ()

Modifier Functions

- void set_name (string name)
- void add_index (TreeObject *obj, Index index)
- void set_last_entry (Index index)
- virtual void insert (string name, TreeObject *obj)
- virtual void erase (string name)
- virtual void insert_addition (TreeObject *add)
- virtual void insert_deletion (TreeObject *del)

Disk Functions

- virtual void write_out ()=0
- virtual void read_in (unordered_multimap< string, FileInfo *> *allFiles, RootTree *rootTree)=0
- virtual void del ()=0
- void increment_allocate (Index *index)
- void increment_follow (Index *index)

Protected Member Functions

- virtual void delete_cont_blocks (BlkNumType blknum)

Protected Attributes

- queue< Modification * > _modifications

  A collection of associated Modifications.
- unordered_map< string, TreeObject * > _myTree

  A collection of contained TreeObjects.
- string _name

  name or value
- BlkNumType _blockNumber

  Blocknumber of the superblock on disk.
- unordered_map< TreeObject *, Index > _indeces

  location(s) of the superblock entry(ies) on disk
- Index _lastEntry

  Index of the last entry of data on disk.
- BlkNumType _startBlock

  blocknumber of the start of this data on disk
- PartitionManager * _myPartitionManager

  Associated PartitionManager.
- queue< Index > **_freeSpots**

4.34.1 Constructor & Destructor Documentation

4.34.1.1 TreeObject()

```
TreeObject::TreeObject (
            string name,
            BlkNumType blknum,
            PartitionManager * pm )
```

Parameters

name	name of this object
blknum	blocknumber of the superblock
pm	PartitionManager object to be associated with this object

4.34.2 Member Function Documentation

4.34.2.1 add_index()

```
void TreeObject::add_index (
            TreeObject * obj,
            Index index )
```

Add an index to _indeces for the specified TreeObject. If the index already existed. nothing happpens

Parameters

obj	the object that the Index references to
index	the Index of obj

4.34.2.2 begin()

```
unordered_map< string, TreeObject * >::iterator TreeObject::begin ( )
```

Returns

An iterator to the beginning of the TreeObjects associated with this object

4.34.2.3 del()

```
virtual void TreeObject::del ( )  [pure virtual]
```

Will completely remove the TreeObject's presence on disk

Implemented in RootTree, TagTree, and FileInfo.

4.34.2.4 delete_cont_blocks()

```
void TreeObject::delete_cont_blocks (
            BlkNumType blknum )  [protected], [virtual]
```

Will follow the chain of continuation blocks and free all of them

Parameters

blknum	will free the blknum and use it to follow the chain of continuation blocks

Reimplemented in FileInfo.

4.34.2.5 end()

```
unordered_map< string, TreeObject * >::iterator TreeObject::end ( )
```

Returns

An iterator to the end of the TreeObjects associated with this object

4.34.2.6 erase()

```
void TreeObject::erase (
            string name )  [virtual]
```

Disassociate the given name from this object

Parameters

name	the name of the object to be erased.

Exceptions

arboreal_logic_error	

Reimplemented in FileInfo.

4.34.2.7 find()

```
TreeObject * TreeObject::find (
            string name ) const
```

Search _myTree for the specified name

Parameters

name	the name of the desired object

Returns

a pointer to the object if found, 0 otherwise

4.34.2.8 get_block_number()

```
BlkNumType TreeObject::get_block_number ( ) const
```

Returns

The blocknumber of the superblock

4.34.2.9 get_free_spots()

```
queue< Index > * TreeObject::get_free_spots ( )
```

Returns

a pointer to the queue of empty spaces where new entries can be added

4.34.2.10 get_index()

```
Index TreeObject::get_index (
            TreeObject * obj ) const
```

Searches for obj and returns the Index of obj on disk, if found

Parameters

obj	object whose position is desired

Returns

The Index of obj on disk,

Exceptions

arboreal_logic_error	

4.34.2.11 get_last_entry()

```
Index TreeObject::get_last_entry ( ) const
```

Find the Index of the last entry for this object on disk

Returns

Index of the last entry on disk

4.34.2.12 get_name()

```
string TreeObject::get_name ( ) const
```

Returns

The name

4.34.2.13 get_start_block()

```
BlkNumType TreeObject::get_start_block ( ) const
```

Returns

The start block of data for this object

4.34.2.14 increment_allocate()

```
void TreeObject::increment_allocate (
            Index * index )
```

Will increment the Index passed and allocate blocks if necessary to do so

Parameters

index	the Index to be incremented

4.34.2.15 increment_follow()

```
void TreeObject::increment_follow (
            Index * index )
```

Will increment the Index passed but only follow the chain of already allocated blocks

Parameters

index	the Index to be incremented

4.34.2.16 insert()

```
void TreeObject::insert (
            string name,
            TreeObject * obj )  [virtual]
```

Associate a TreeObject with this object

Parameters

name	name of the object, mangled if inserting a FileInfo
obj	the object to be inserted

Exceptions

tag_error	

See also

FileInfo::insert()

Reimplemented in FileInfo.

4.34.2.17 insert_addition()

```
void TreeObject::insert_addition (
            TreeObject * add )  [virtual]
```

Add an Addition to the list of Modifications so that it can be written out later. Note: Do not call this on a FileInfo.

Parameters

add	the object that was previously inserted to this object which will be added to the list of Modifications

See also

FileSystem::write_out() TreeObject::insert()

Reimplemented in FileInfo.

4.34.2.18 insert_deletion()

```
void TreeObject::insert_deletion (
            TreeObject * del )  [virtual]
```

Add a Deletion to the list of Modifications so that it can be written out later. Note: Do not call this on a FileInfo.

Parameters

del	the object that was previously erased from this object which will be added to the list of Modifications

See also

FileSystem::write_out() TreeObject::erase()

Reimplemented in FileInfo.

4.34.2.19 read_in()

```
virtual void TreeObject::read_in (
            unordered_multimap< string, FileInfo *> * allFiles,
            RootTree * rootTree )  [pure virtual]
```

Will read in all object data from disk

Parameters

allFiles	a pointer to the map of all files
rootTree	a pointer to the root tree

Implemented in RootTree, TagTree, and FileInfo.

4.34.2.20 set_last_entry()

```
void TreeObject::set_last_entry (
            Index index )
```

Set the last Index for the last entry belonging to this object on disk

Parameters

index	The last Index

4.34.2.21 set_name()

```
void TreeObject::set_name (
            string name )
```

Set the name

Parameters

name	The new name

4.34.2.22 size()

```
size_t TreeObject::size ( ) const
```

Returns

The size of _myTree

4.34.2.23 write_out()

```
virtual void TreeObject::write_out ( )  [pure virtual]
```

Intended to write out the object to disk

Implemented in RootTree, TagTree, and FileInfo.

The documentation for this class was generated from the following files:

- Filesystem/DaemonDependancies/Trees/Trees.h
- Filesystem/DaemonDependancies/Trees/Trees.cpp

Index