STRIPEFS: STRIPING FILE SYSTEM

A Project Report

presented to the Faculty of

Sam Higginbottom Institute of Agriculture, Technology & Sciences

Allahabad, India

In Partial Fulfilment

of the Requirements for the Diploma

Post Graduate Diploma in Computer Applications

by

Mrinal Dhillon

Enrolment No. I-1151070052

June 2016

ABSTRACT

StripeFS: Striping File System

by

Mrinal Dhillon

This project report presents StripeFS file system, an approach to unify local and remote storage options available on unix platforms. StripeFS file system provides RAID 0 [16] [17] type striping function at file level through a stackable design on locally mounted file systems. It provides data security and optimised space by splitting and distributing file across multiple local directories as storage backends. This approach extends advantage to systems with network-attached storage(NAS) such as NFS, SMB, AFP through aggregation of available space by striping onto distinct NAS clients mounted as local directories. StripeFS can also unify multiple virtual file system options for public cloud based storage such as google, apple, amazon, dropbox on unix platforms such as Ubuntu, openSUSE, CentOS, macOS.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

1. Background and Introduction

1.1. Linux Filesystems

Linux VFS [1] [2] provides standardised set of interfaces for applications to perform file level operations over a diverse set of file systems as seen in Figure 1.1. It allows wide range of distinct file systems supporting diverse features and storage medium to co-exist through a common interface to interact with applications in the user space. Systems strive cross platform portability through compliance with Portable Operating System Interface (POSIX). POSIX file system application programming interfaces(API) provide applications access to standard set of file system functions to iterate file system hierarchy; access and manipulate files; without the need of distinct interactions per file system type.

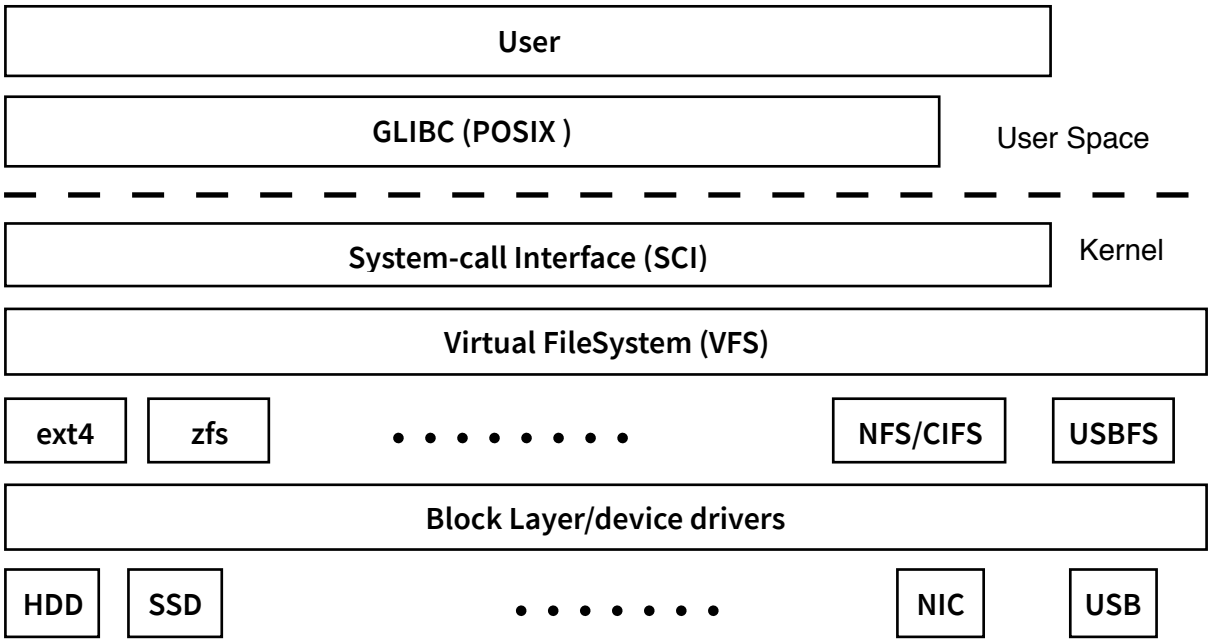


Figure 1.1: Linux FileSystem High Level Architecture [1][2]

1.1.1. Stackable File System in Kernel

Linux VFS through Virtual Node or vnode provides ability to stack custom file systems atop low level file systems in kernel space. This allows extended features without changing code of mature native file systems [2]. UnionFS, WrapFS [3] are examples of such kernel level layered/stackable file systems as seen in Figure 1.1.1 developed using FiSTGen tools [4].

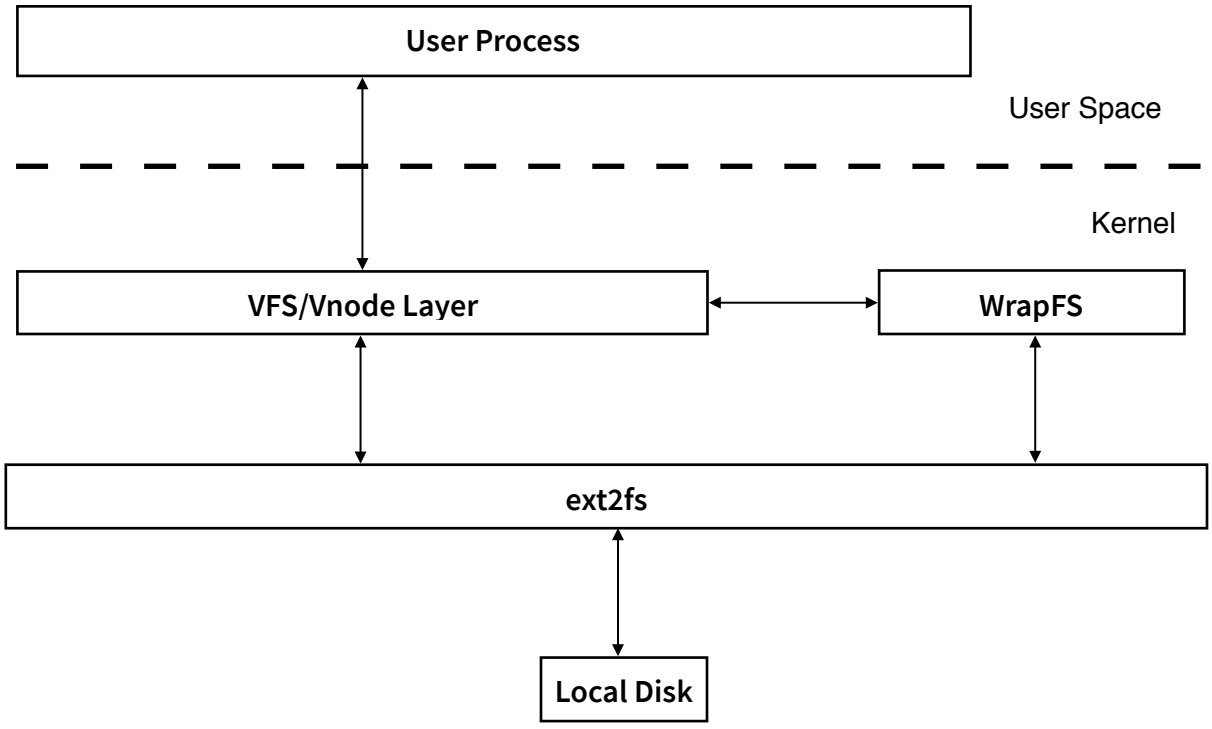


Figure 1.1.1: Stackable File Systems in Kernel [3]

1.1.2. Fuse

Fuse as seen in Figure 1.1.2 is VFS compliant in-kernel file system module with user space component that allows implementation of file system api in the user space. It is very useful to implement complex file functions using extensive systems and application level features [5]. For example GCSFuse [6], S3FS [7] are fuse based file systems that enable applications to access cloud based object store as locally mounted file system. These filesystem use systems level libraries like curl for restful access to objects in the cloud; schema handling using json, xml; caching for object manipulation and local databases like postgresql to maintain metadata in user space. Such functionalities would be cumbersome to implement in kernel.

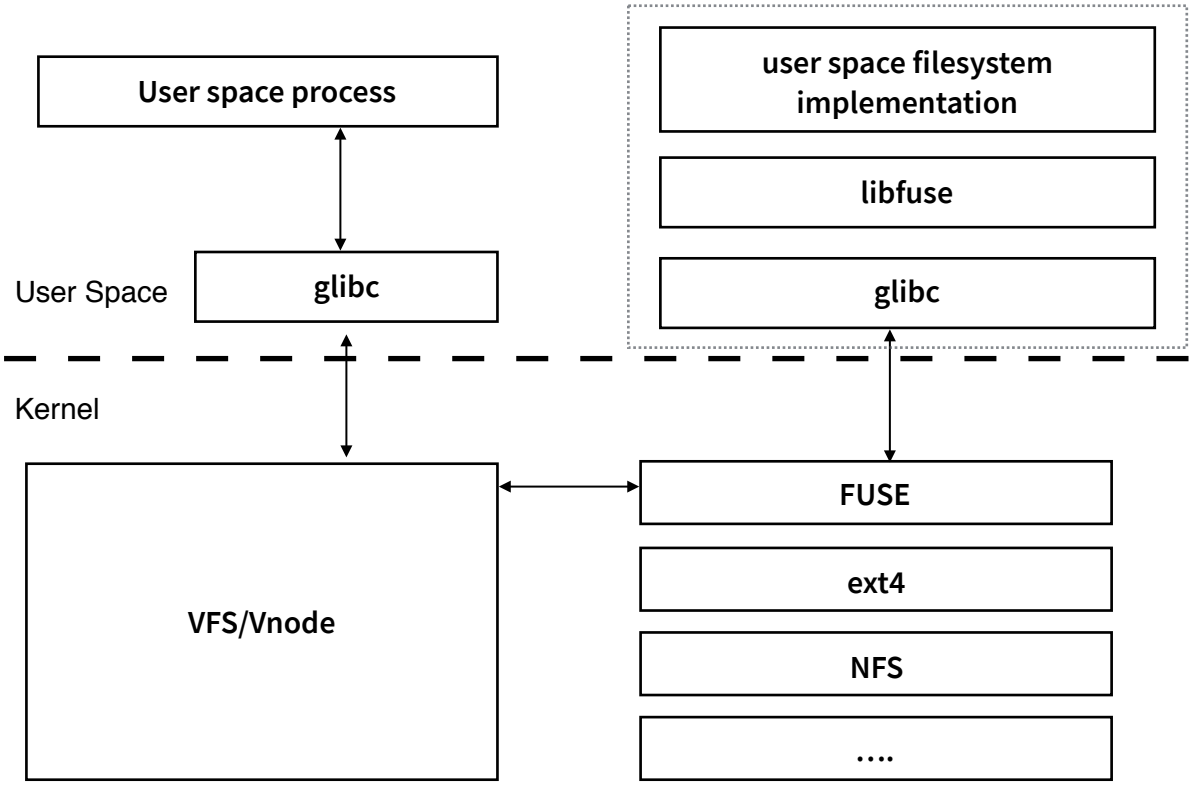


Figure 1.1.2: FUSE Filesystem in User Space [5][15]

1.1.3. Stackable File System in User Space

As discussed in previous section Fuse enables implementation of file system api in user space. Just like an application in user space, fuse file system may also access locally mounted file systems through POSIX file system api. This allows a stackable file system design such as seen in Figure 1.1.3 where the file api may be manipulated by fuse based file system implementation in user space and then forwarded to a native file system path. StripeFS striping and stacking ability as seen in Figure 3.2 in an example of such file systems hierarchy.

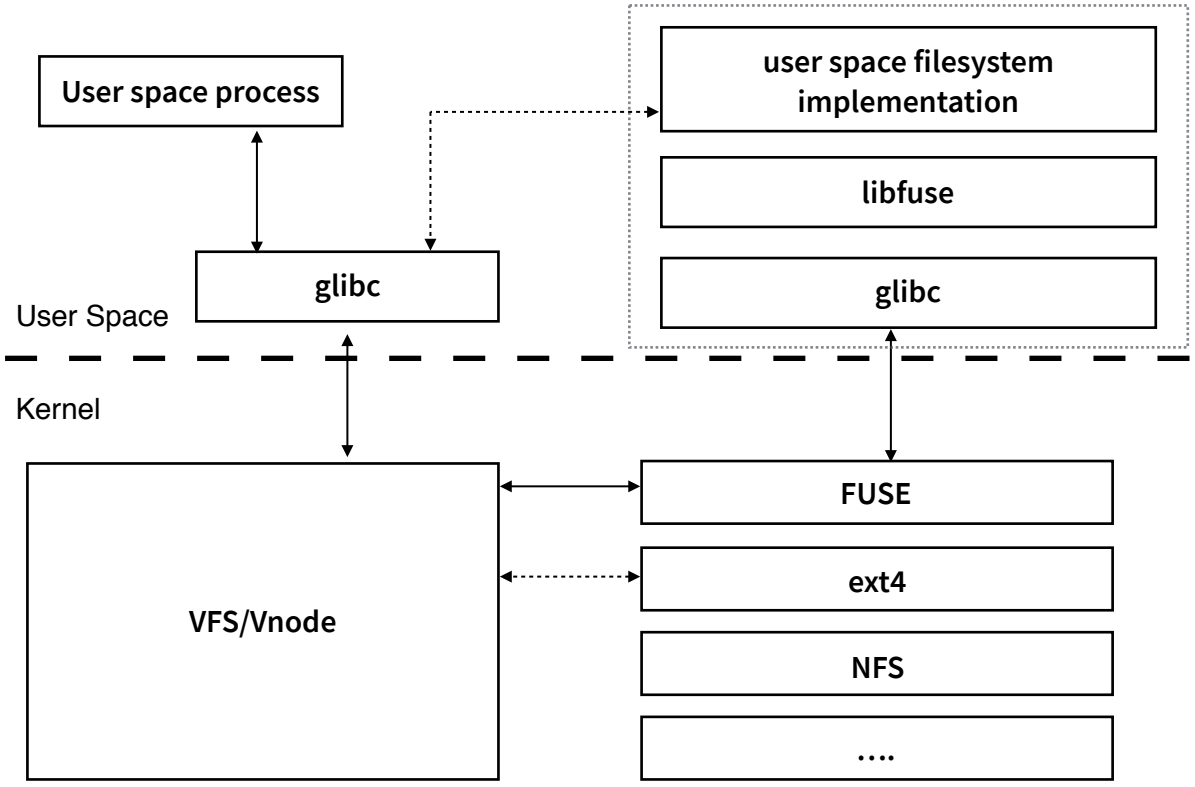


Figure 1.1.3: Fuse based Stacking File System Design

1.2. File Striping

Striping as a term has been used abundantly across multiple platform from Linux to Windows at multiple levels in storage and networking stacks. Data striping means logically segmenting sequential data across multiple storage backends. One of most common use cases of data striping is “redundant array of independent disks” RAID [16] [17] level 0 for disk drives. File striping adhering to RAID 0 is the technique to split and distribute contents of a file across multiple disk sets.

1.3. Networked Storage

Networked storage is remote storage accessible though network medium. Common correlation between network storage and enabling technologies are NAS and SAN based storage. NAS exposes storage as file server accessed using network file sharing protocols such as NFS [8], SMB [9], AFP [10] whereas SAN exposes storage as disks accessed though protocols such as Fibre, iSCSI et al. Public cloud based object storage is another popular cheap remote storage option accessible through Restful/Soap based apis.

1.3.1. Networked File Systems

NFS and Samba [11] are most widely used NAS implementations on unix systems. NFSv4 [12] is natively supported on linux, free bsd; is well supported on macOS and also on Windows. It follow client-server architecture where locally mounted file system is the

client to remote NFS server. It allows supported linux based OS to share its local directories and files with clients over network. NFS is mostly deployed in educational institutes and enterprises where cheap shared shared storage access is required.

File systems to access cloud object stores abstract the disconnect between properties and protocols to access an object and the way a file is handled. Google Drive, GCSFuse, Dropbox, AWS S3FS provide access to public cloud based object containers as local directories providing file abstraction to object and object path-prefix translating to directories.

1.4. Our Contribution

StripeFS is stackable striping file system in user space based on design seen in Figure 1.1.3. It exposes a virtual file system hierarchy to applications while maintaining corresponding file system hierarchy(stripe namespace) and file data across stripe directories. Moreover it adheres to POSIX to achieve seamless integration with applications' file i/o path. File striping is implemented by distributing and splitting file, data operations onto corresponding file stripes in stripe namespace. The stripe directories may be combination of ext4, NFS, S3FS directories. This way users may customise data storage strategies by unifying distinct available storage mediums. It is implemented in Ruby. Detailed architecture is discussed in Chapter 3.

CHAPTER 2

2. StripeFS Requirements

2.1. System Requirement

1. Operating System
    1. Linux based distributions.
    2. fuse, ruby >= 1.9

2.2. Filesystem Requirement

2. POSIX Compliant API

   Applications should seamlessly access files without any consideration towards nature of StripeFS.

3. Underlay File systems

   Stripe targets should be on POSIX compliant file systems such as ext4, NFS, Fuse file systems like GCSFuse, S3FS, Google Drive, Dropbox.

2.3. Limitations and Configuration

4. Command Line Options

```
$ stripefs mountpoint –o fuse_options  –s stripe#1 –s stripe#2 … –s
stripe#n –c chunksize
```

5. Number of Stripes

Number of stripes is critical to performance; should be carefully chosen considering type of applications doing bulk of operation on StripeFS and expected average size of files.

6. Stripe Chunk Size

Chunk size for stripes should be multiple of block size of target stripes i.e. 4kB, 8kB, 16kb, 32kb. Since writes are based on file offset, stripe chunk size and size of data buffer, very large chunk size may result in large number files smaller than one chunk being written completely on first stripe which may result in disproportional space consumption among stripes. However very small chunk size may result in large number of file operations on stripe paths for each read, write operation that may result in poor performance.

Linux utilities cp, cat, tar use 32 kb buffers for write operations. If bulk of operations are copy, combinations of following options could be considered.

- Fuse option big_writes enabled.
- 4 <= Stripes Count <= 8.
- Chunk Size = 32kB/(Stripes Count)

7. Parallel Striping

Parallel striping should in theory result in higher throughput. Ideal read and write transfer rates for ***n*** stripes should be upto ***n*** times higher than individual stripe rate [17].

CHAPTER 3

3. StripeFS Design

3.1. Overview

StripeFS is a fuse based filesystem that implements striping function at file level. It exposes a virtual file system hierarchy, POSIX interface to applications under its mount point directory. As shown in Figure 3.1 it imposes a file system abstraction layer encapsulating stripe target directories. File system internals and workflows are explained further in the chapter based on Figure 3.3.

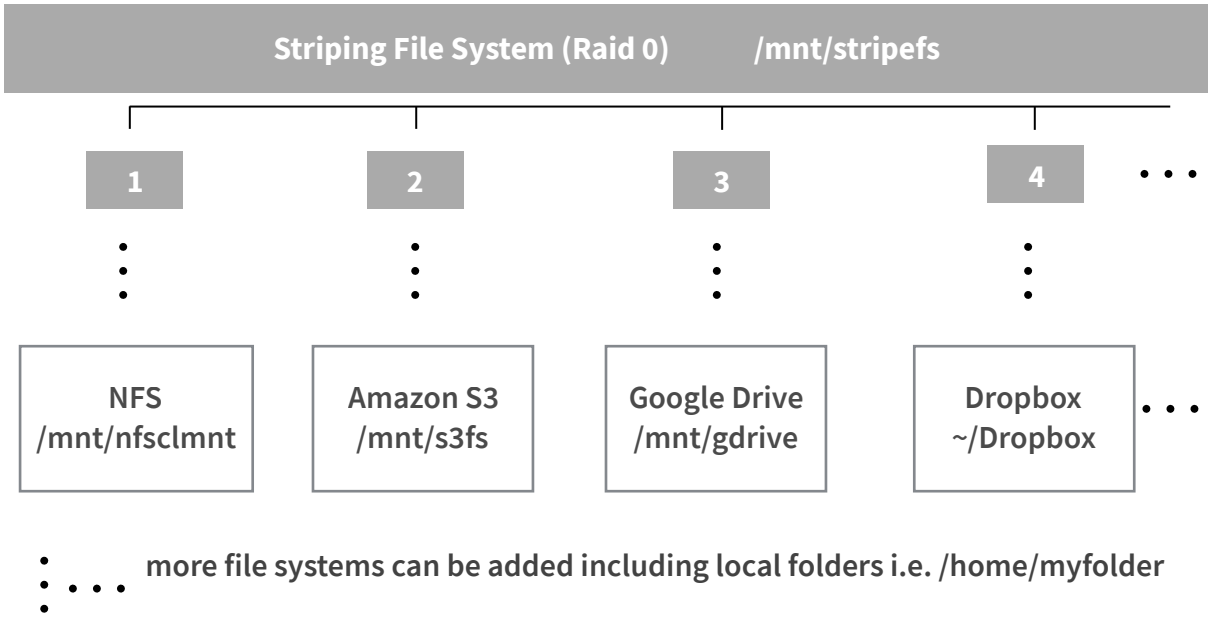


Figure 3.1: High level design of StripeFS

3.2. File System Hierarchy

StripeFS exposes virtual file system hierarchy under its mount path while managing the corresponding distinct hierarchy of files and directories under stripe paths'. Stripe paths and stripe sequence are declared when invoking command line utility to mount the file system. The resultant underlay file system hierarchy created and maintained by StripeFS is tightly coupled with sequence of stripes at the time of mount and thus require initial sequence to be maintained between system reboots and remounts. The corresponding path elements including basename under each stripe is suffixed with ".stripe_number".

```
/mnt/StripeFS/folder/file –> ~/stripe#1/folder.1/file.1, ~/stripe#2/
folder.2/file.2, …, ~/stripe#n/folder.n/file.n
```

3.3. File System Architecture

Figure 3.3 shows the File IO path and internal architecture of linux system that supports StripeFS. StripeFS fuse daemon runs in the application layer/user space. Linux OS provides the supporting infrastructure in kernel and user space such as libc, VFS/Vnode, fuse kernel module, libfuse to forward file io to StripeFS daemon that serves the implementation of these apis.

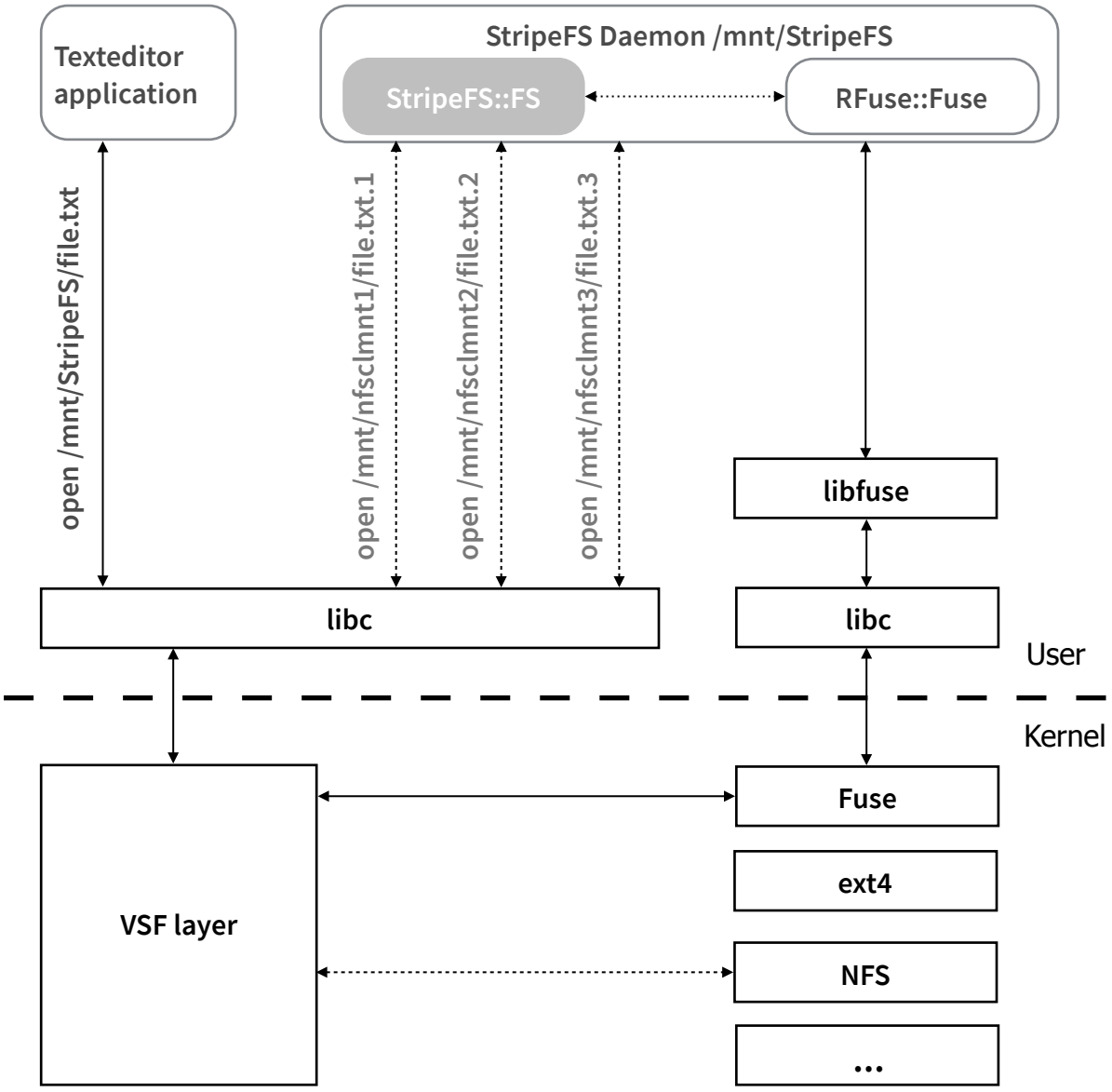


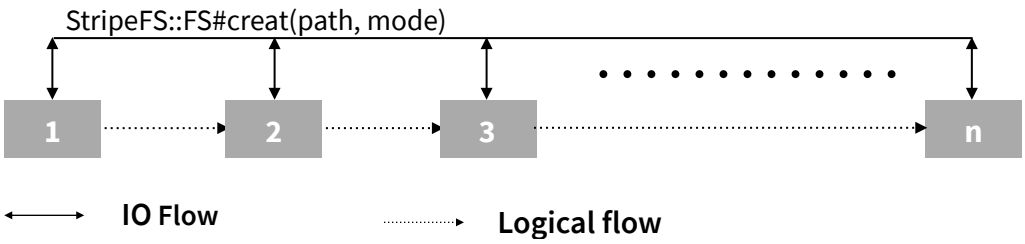
Figure 3.3: StripeFS FileSystem Architecture and Data/Control Flow

3.4. File System I/O Workflow

StripeFS supports standard apis for file and directory operations that are necessary for a functional file system i.e. open, creat, read, write, close, unlink, getattr, fgetattr, truncate, ftrunctate, chown, chmod, utimens, mkdir, rmdir, readdir, statfs. An application performs file manipulation under StripeFS mount point using POSIX apis. The file i/o is sent to VFS in kernel space. VFS forwards call to the corresponding fuse kernel interface that forwards it to fuse daemon running in user space that serves the file operation. Further workflow is discussed from perspective of the applications and StripeFS since file i/o path from applications upto StripeFS is handled by OS components as shown in Figure 3.3 and out of scope of this project. StripeFS fuse daemon on receiving file system call performs corresponding set of operations onto the stripe paths like a system level application confirming to its stackable design.

3.4.1. File Operations

File I/O such as open, create, release, mkdir, rmdir, chown, chmod, getattr, require corresponding operations on each stripe. POSIX system call mkdir(/mnt/stripefs/dir, 644) results in mkdir operations with path elements suffixed with corresponding stripe number under each stripe path. As shown in Figure 3.4.1 for libc function creat("/mnt/stripefs/dir/testfile.txt", 655) StripeFS creates the file if not exists under each stripe path with file permission bits set to 655 suffixed with corresponding stripe number i.e. ~/stripe#1/dir.1/testfile.txt.1, ~/stripe#2/dir.2/testfile.txt.2, …, ~/stripe#*n*/dir.n/testfile.txt.n .



Finger 3.4.1: File IO Striping

3.4.2. Data Operations

Data operations as shown in Figure 3.4.2 such as read, write, truncate, ftruncate, follow round robin algorithm to map the file offset, size of data read/write to corresponding stripe, offset and size in stripe targets. If data offset falls within the boundary of a chunk in stripe#***x*** and length of bytes exceed the end offset of the chunk in the stripe then next stripe and updated offset in the stripe is determined (round robin) till size of data buffer is reached.

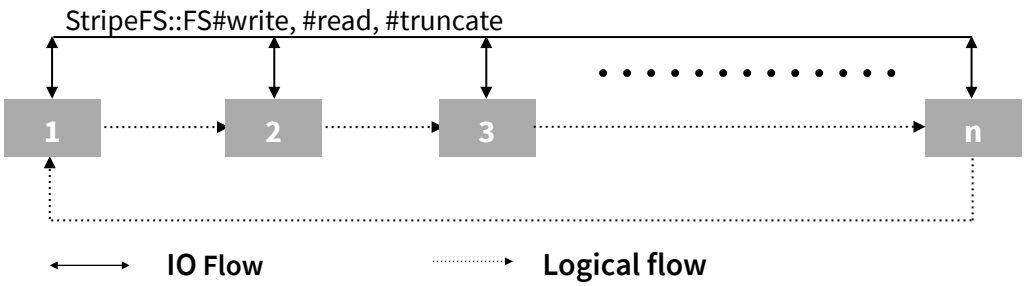


Figure 3.4.2: Data Striping

3.5. Sequence of Fuse Operations

Fuse handles sequence of operations. A typical file access call will result in getattr(stat) file path to determine existence, permissions, properties; accordingly next method is called. Following is sequence of operations for a cat system utility to print file contents on standard output.

```
$ cat /mnt/StripeFS/dir/file
```

Fuse calls following sequence of StripeFS methods

- Check file existence and access, read, write permissions
  - StripeFS::FS#getattr(ctx, “/”) —> Stat
  - StripeFS::FS#getattr(ctx, “/dir”) —> Stat
  - StripeFS::FS#getattr(ctx, “/dir/file”) —> Stat
- Open File /mnt/StripeFS/dir/file
  - StripeFS::FS#open(ctx, path, ffi) —> void
- Until read buffer size equals file size
  - StripeFS::FS#read(ctx, path, offset, size, offset, ffi) —> buffer
- Flush file
  - Not implemented since not required for StripeFS implementation
- Close file
  - StripeFS::FS#release(ctx, path, ffi)

CHAPTER 4

4. StripeFS Implementation

StripeFS is implemented in two layers i.e. StripeFS::CMD and StripeFS::FS.

4.1. StripeFS::CMD

CMD module exposes an intuitive set of options for user to mount the file system. Users can set the mount path, stripe paths and chunk size along with standard fuse options from terminal and from applications.

```
StripeFS::CMD#mount(ARGV)
StripeFS::CMD#mount([“mountpath", "-o", "[optional fuse options]", "-s", "stripe1", "-s", "stripe2", …, "-s", "stripe N", "-c", “chunksize”])
```

4.2. StripeFS::FS

StripeFS::FS layer provides the implementation of RFuse::Fuse [14] api that is the ruby port of libfuse file system development library. It implements the striping logic.

As shown in Table 4.2 StripeFS::FS defines most RFuse::Fuse apis except link, symlink, readlink, mknod, access, getxattr, setxattr, listxattr apis which are not required for a fully functional filesystem.

StripeFS::FS Methods	Action
#chmod(ctx, path, mode) -> void	change permissions of object
#chown(ctx, path, uid, gid) -> void	change ownership of object
#create(ctx, path, mode, ffi) -> void	create a file if not exists
#fgetattr(ctx, path, ffi) -> Stat	get attributes/stat information of object
#ftruncate(ctx, path, size, ffi)	change size of open file
#getattr(ctx, path) -> Stat	stat on path
#mkdir(ctx, path, mode) -> void	create a directory
#open(ctx, path, ffi) -> void	file open operation
#read(ctx, path, size, offset, ffi) -> String	read string of size at offset in file
#readdir(ctx, path, filler, offset, ffi) -> void	returns contents of a directory
#release(ctx, path, ffi) -> void	close a file
#rename(ctx. from, to) -> void	rename path
#statfs(ctx, path) -> StatVfs	get filesystem statistics
#truncate(ctx, path, size) -> void	change size of file
#unlink(ctx, path) -> void	remove a file
#unmount -> Object	unmount filesystem
#utimens(ctx, path, actime, mod time) -> void	change access and modification times of file
#write(ctx, path, data, offset, ffi) -> Integer	write data of size at offset in file

Table 4.2: StripeFS::FS methods

4.2.1. #create(ctx, path, mode) —> void

Unless path exists create file at each stripe path with mode

4.2.2. #fgetattr(ctx, path, ffi) —> Stat

stat = getattr(ctx, path)

return stat

4.2.3. #ftruncate(ctx, path, offset, ffi) —> void

Calculate striped_path[0].offset, …., striped_path[n-1].offset

Truncate striped_path[0].offset, …, striped_path[n-1].offset.

4.2.4. #getattr(ctx, path) —> Stat

Get attributes of object at striped_path[0]

Add size of objects at striped_path[0] .. striped_path[n-1]

Return attributes

4.2.5. #mkdir(ctx, path, mode) —> void

Unless path exists create directory at each stripe path with mode

4.2.6. #open(ctx, path, ffi) —> void

striped_paths.each {|stripe| stripe.open(ffi.flags)}

4.2.7. #read(ctx, path, size, offset, ffi) —> String

Map offset and size in file to offsets and corresponding stripes_read_size in each stripe in round robin fashion till stripes_read_size is less that size

1. Find which stripe to begin readiing
2. Find offset_in_stripe to read at.
3. Calculate size to read based on chunk boundary at the offset_in_stripe
4. Size to read size_in_stripe = chunk boundary - offset_in_stripe
5. Read buffer of size_in_stripe at offset_in_stripe
6. Update file offset
7. Update size -= read_length
8. Repeat steps 1 to 8 if size != 0 or buffer.empty?

4.2.8. #readdir(ctx, path, filler, offset, ffi) —> void

Get immediate children of striped_paths[0]

children.each do |child|

    remove trailing index string ".1"

    filler.push(child)

end

4.2.9. #release(ctx, path, ffi) —> void

Since we are not returning open file pointers in ffi during #open, #create this function is just a placeholder. In future version this function will be implemented since opening and closing files for read, write operations are expensive.

4.2.10. #rename(ctx, from, to) —> void

striped_paths_**from**.each_with_index do |stripe, index|

stripe.rename(striped_paths_**to**[index])

end

4.2.11. #truncate(ctx, path, size) —> void

Calculate striped_path[0].offset, …., striped_path[n-1].offset

Truncate striped_path[0].offset, …, striped_path[n-1].offset.

4.2.12. #utimens(ctx, path, actime, mtime) —> void

striped_paths.each { |stripe| stripe.utime(atime, mtime) }

4.2.13. #write(ctx, path, data, offset, ffi) —> Integer

Map offset and data size in file to offsets and corresponding write_size in each stripe in round robin fashion till write_length is less that data.size.

1. Find which stripe to begin writing
2. Find offset_in_stripe to write at.
3. Calculate size to write based on chunk boundary at the offset_in_stripe
4. Size to write size_in_stripe = chunk boundary - offset_in_stripe
5. Write data of size_in_stripe at offset_in_stripe
6. Data seek current plus length_written.
7. Update offset += length_written
8. Repeat steps 1 to 8 if data.tell < data.size

CHAPTER 5

5. Testing

5.1. Test Setup

Testing is done on Ubuntu 14.10 LTS server virtual machine with 2 processors, 4096 MB RAM and 50GB virtual disk formatted with ext4, Fuse version 2.9.2, Ruby 1.9.3 installed. File system is mounted with three local stripe targets under users home directory. Benchmarking is done with stripe chunksize 4096 and fuse big_writes enabled.

5.2. System Utilities

Standard linux command line utilities such ls, stat, find, cat, touch, truncate, cp, mv, rm, chmod, chown, tar, unzip are used to test filesystem compatibility. File Explorer desktop utility is used test common operation in Desktop environment. All operations have maintained file integrity. Large file copy of 1.1 GB image file took 12.03 seconds as against 1.35 seconds taken by same operation on local directory.

5.3. Software

Text file editing using Vim, LibreOffice. Libfuse source compiled on file system root using make utility involves most file system apis and is a good test for robustness of StripeFS.

5.4. Benchmarking

Bonnie [13] benchmark has passed the file system with low performance numbers.

Version 1.97		Sequential Output						Sequential Input				Random Seeks			Sequential Create						Random Create					
	Size	Per Char		Block		Rewrite		Per Char		Block				Num Files	Create		Read		Delete		Create		Read		Delete	
		K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	/sec	% CPU		/sec	% CPU	/sec	% CPU	/sec	% CPU	/sec	% CPU	/sec	% CPU	/sec	% CPU
dev	8G	7	8	35222	9	32401	5	4868	98	142992	5	1593	15	16	1536	3	4193	9	1907	6	1664	3	5144	4	2108	4
	Latency	1728ms		832ms		818ms		10237us		25627us		13167us		Latency	7254us		3886us		43043us		7279us		4718us		9685us	

Figure 5.4: Benchmarking using Bonnie [13]

CHAPTER 6

6. Use Cases

6.1. NFS

Most enterprise and educational institutes deploy on-premise NFS servers to enable cheap storage and sharing of files. NFS servers export local directories over the network to client computers. Client computers are required to locally mount the NFS client with required access permissions to access the NFS exports. StripeFS can unify multiple NFS Clients mounts and stripe files onto corresponding NFS exports. Various configurations are possible in such scenario with NFS exports being distributed across multiple NFS servers to enable load balancing and data security through striping. Figure 3.3 illustrates StripeFS striping onto district NFS Client mounts.

6.2. Cloud Storage

With advent of public cloud, object store based solutions have become common on personal computers. Applications like Google Drive, S3FS, Dropbox, iCloud Drive provide file system abstraction to cloud based object storage. Most users have multiple free or payed subscriptions to cloud storage. Solutions like Google Drive provide universal compatibility while iCloud Drive provides privacy and exclusiveness to apple users. StripeFS has the ability to stripe files across the distinct locally mounted cloud storage

containers while providing users and applications an abstraction of unified file system hierarchy. This way users can more efficiently use space and ensure data security. This also allows for efficient bandwidth consumption and faster sync since the stripes will be synced in parallel rather than sequential file sync to the corresponding cloud backends.

CHAPTER 7

7. Conclusion and Further Work

7.1. Conclusion

StripeFS enables files striping across distinct storage options such as NAS, cloud storage, local disk space while providing a unified file system view to application in user space. It takes advantage of matured file systems that connect networked storage to users desktop. It allows users to efficiently aggregate local and networked storage across personal computers, enterprise nas servers, public and private object stores.

7.2. Next Versions

- File system apis symlink, link, readlink, setxattr, getxattr, listxattr, mknod, access.
- In-memory file handles will be maintained by StripeFS methods open, creat, read, write, release, ftruncate.
- Disk space constraints should adhere to stripes' aggregated capacity and individual space limits.

7.3. Desired Features

- Stripes should be accessed in parallel to achieve higher performance.
- Concurrent file access is not handled. StripeFS relies on properties of underlay file systems. Further research is needed to support concurrent access.
- Ruby as the choice of language is used for fast prototyping and potability. C++ is being considered for further development to achieve higher performance.
- Features such mirroring (RAID 1) [16][17], encryption are required to provide added security and integrity of data through failures.
- macOS support require resolution of RFuse::Fuse incompatibility with osxfuse.

7.4. Testing

Further testing is needed using various configuration of distinct file system backend stripes such as NFSv4 clients, Samba, AFP, S3FS, Google Drive, Dropbox and benchmarking tools such as filebench, iozone.

Bibliography

(1) M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and H. Bohme, Linux Kernel Internals. Addison-Wesley Publishing Company, 1997.

(2) M. Time Jones, Anatomy of the Linux virtual file system switch, http://www.ibm.com/developerworks/library/l-virtual-filesystem-switch/

(3) Erez Zodac, Ion Badulescu, A Stackable Filesystem Interface For Linux, https://www.filesystems.org/docs/linux-stacking/linux.pdf

(4) FiST: Stackable Filesystem Language and Templates, https://www.filesystems.org

(5) M. Szeredi. Filesystem in userspace, http://fuse.sourceforge.net/

(6) Google Cloud Storage Fuse, https://cloud.google.com/storage/docs/gcs-fuse

(7) S3FS: Fuse Over Amazon, https://github.com/s3fs-fuse/s3fs-fuse

(8) NFS: Network File System Protocol Specification, https://tools.ietf.org/html/rfc1094

(9) Microsoft SMB Protocol and CIFS Protocol Overview, https://msdn.microsoft.com/en-us/library/windows/desktop/aa365233(v=vs.85).aspx

(10) AFP: Apple Filing Protocol, https://developer.apple.com/library/mac/documentation/Networking/Reference/AFP_Reference/index.html

(11) Samba, https://www.samba.org

(12) Network File System (NFS) version 4 Protocol, https://www.ietf.org/rfc/rfc3530.txt

(13) T. Bray, Bonnie, http://www.textuality.com/bonnie

(14) RFuse 1.1.3, https://rubygems.org/gems/rfuse/versions/1.1.2

(15) File System in User space, https://en.wikipedia.org/wiki/Filesystem_in_Userspace visited 4 April 2016.

(16) RAID https://en.wikipedia.org/wiki/RAID visited 4 April 2016

(17) Standard RAID levels https://en.wikipedia.org/wiki/Standard_RAID_levels visited 4 April 2016

Appendix

1. System Manual

StripeFS fuse based file system is developed using RFuse ruby port of libfuse. You may email at mrinaldhillon@gmail.com to request the code.

1.1. Development Dependencies:

- Ubuntu, openSUSE, CentOS
- fuse 2.9.2 Filesystem in Userspace [5]
- Ruby 1.9
- rfuse (1.1.2) ruby gem [14]

```
# ubuntu LTS 14.04
$ sudo apt-get install ruby
$ gem install rfuse
```

1.2. Using Interactive Ruby Shell

```
$ irb
irb(main):001:0> require "stripefs"
=> true
irb(main):002:0> StripeFS::CMD.mount(["/mnt/stripefs", "-s", "~/stripes/
1", "-s", "~/stripes/2", "-s", "~/stripes/3", "-c" "4096"])
```

1.3. Mount from Application

```
require "stripefs"
StripeFS::CMD.mount(["/mnt/stripefs", "-s", "~/stripes/1", "-s",
"~/stripes/2", "-s", "~/stripes/3", "-c" “4096"])
```

2. User Manual

StripeFS is fuse based file system application. You may email at mrinaldhillon@gmail.com to request the application bundle.

2.1. Command Line

- Mount

```
$ stripefs /mnt/stripefs -s ~/stripes/1 -s ~/stripes/2 -s ~/stripes/3, -c
4096 &
$ mount | grep /mnt/stripefs
/dev/fuse on /mnt/stripefs type fuse (rw,nosuid,nodev,user=dev)
$ df | grep /mnt/stripefs
Filesystem           1K-blocks Used    Available Use% Mounted on
/dev/fuse             1000000  500000    990000  34% /mnt/stripefs
$ cd /mnt/stripefs
```

- Unmount

```
$ fusermount -u /mnt/stripefs
```

3. Code Listing

stripefs/bin/stripefs

stripefs/lib/stripefs.rb

stripefs/lib/stripefs

stripefs/lib/stripefs/utils.rb

stripefs/lib/stripefs/fs.rb

3.1. StripeFS Binary

```
    # bin/stripefs
1.  #!/usr/bin/env ruby
2.  require_relative '../lib/stripefs'
3.
4.  # Command line utility to mount StripeFS FileSystem
5.  #
6.  # @author Mrinal Dhillon
7.  # @example
8.  #   bin/stripefs /mnt/stripefs -s ~/stripes/1 -s ~/stripes/2
9.  #                                  -s ~/stripes/3 -c 4096
10.
11. StripeFS::CMD.mount(ARGV)
```

3.2. StripeFS::CMD Parse and Mount Api

```
# lib/stripefs.rb
1.  require 'optparse'
2.  require_relative 'stripefs/utils'
```

```
3.   require_relative 'stripefs/fs'
4.   # StripeFS is stackable striping file system
5.   #
6.   # @author Mrinal Dhillon
7.   # @email mrinaldhillon@gmail.com
8.   module StripeFS
9.       # Command line utility to mount StripeFS FileSystem
10.      #
11.      #  @example
12.      #        StripeFS::CMD.mount(["/mnt/stripefs, "-s", "~/stripes/1",
13.      #        "-s", "~/stripes/2", "-s", "~/stripes/3", "-c" "4096"])
14.      module CMD
15.          extend self
16.          # Parse options
17.          # @param [Array<String>] command line arguments
18.          #
19.          # @return [Fixnum, Array<String>]  chunksize, array of stripe
     paths
20.          # @note method is written to handle command line arguments
     ARGV.
21.          def parse(args)
22.              chunksize = 1024   #default chunksize is 1024
23.  # lib/stripefs.rb
24.              stripes = []
25.
26.              opt_parser = OptionParser.new do |opts|
27.                  opts.banner = "Usage: stripefs [options]"
28.                  opts.separator ""
29.                  opts.separator "Specific options:"
30.                  opts.on("-s", "--stripe String", String, "Stripe
     path"){
```

```
31.                                             |str| stripes << str }
32.                 opts.on("-c", "--chunksize N", Integer, "Chunksize") {
33.                                             |chk| chunksize = chk }
34.                 opts.separator ""
35.                 opts.separator "Common options:"
36.
37.                 opts.on_tail("-h", "--help", "Show this message") do
38.                     $stdout << opts.to_s
39.                     exit
40.                 end
41.             end
42.
43.             opt_parser.parse!(args)
44.             return chunksize, stripes
45.         end
46.
47.         # Mount StripeFS file system
48.         #    @param argv [Array<String>] command line options
49.         def mount(argv)
50.             begin
51.                 chunksize,stripes = parse(argv)
52.
53.                 if Utils.is_blank?(stripes) || stripes.count <= 1
54.                     parse(["-h"])
55.             # lib/stripefs.rb
56.                     fail ArgumentError, "Number of Stripes should be
                                                          greater than 1"
57.                 end
58.
59.                 RFuse.main(argv) do |options, args|
60.                     FS.new(options[:mountpoint], stripes, chunksize)
```

```
61.                 end
62.
63.                 rescue => error
64.                   $stderr << error
65.                 fail error
66.             end
67.         end
68.     end # CMD
69. end # StripeFS
```

3.3. StripeFS::FS Fuse api

```
# lib/stripefs/fs.rb
1.  require 'rfuse'
2.  require 'pathname'
3.  require "stringio"
4.  require "fileutils"
5.  require_relative “utils”
6.
7.  module StripeFS
8.    # Defines RFuse::Fuse file system apis to implement StripeFS
9.    #
10.   # @author Mrinal Dhillon
11.   # @todo link, symlink, readlink, mknod, access, getxattr,
12.   #     setxattr, listxattr, flush apis
13.   class FS
14.         #     Creates StripeFS::FS instance
15.         #
16.         #     @param mountpath [String] mountpoint for fuse filesystem
17.         #     @param stripes [Array<String>] stripe paths
```

```
18.         #     @param chunksize [Fixnum] chunksize for stripes
19.         #
20.         #     @raise [ArgumentError]
21.         def initialize(mountpath, stripes, chunksize)
22.               fail ArgumentError, "Invalid Inputs" if Utils.is_blank?
    (stripes) || Utils.is_blank?(mountpath)
23.
24.               @rootstat = Pathname.new(mountpath).lstat
25.               @root = Pathname.new(mountpath)
26.               fail ArgumentError,
27.                     "Mountpoint #{mountpath} is not a directory" unless
    @root.directory?
28.
29.               @chunksize = chunksize
30.               @uid = Process.uid
31.               @gid = Process.gid
32.
33.               @stripes = []
34.               Array(stripes).each_with_index do |stripe, index|
35.                     @stripes[index] = Pathname.new(stripe)
36.                     fail ArgumentError,
37.                     "Stripe target #{stripe} is not a directory" unless
    @stripes[index].directory?
38.               end
39.         end   #     initialize
40.
41.         INIT_TIMES = Array.new(3,0)
42.
43.         # Return stat info of object
44.         # @param path [::Pathname] path of object
45.         #
```

```
46.         # @return [::Stat] stat of object
47.         def lstat(path)
48.               return @rootstat if path.to_s == @root.to_s
49.               path.lstat
50.         end   # lstat
51.
52.         def method_missing(method, *args)
53.               $stderr << "#{method} is not implemented"
54.         end   # method_missing
55.
56.         # Fuse statfs method. {RFuse::Fuse#statfs}
57.         # @param ctx [RFuse::Context]
58.         # @param path [String] object path
59.         #
60.         #     @return [RFuse::StatVfs]
61.         #     @todo calculate based on stripe paths
62.         def statfs(ctx, path)
63.               s = RFuse::StatVfs.new()
64.               s.f_bsize    = 1024
65.               s.f_frsize   = 1024
66.               s.f_blocks   = 1000000
67.               s.f_bfree    = 500000
68.               s.f_bavail   = 990000
69.               s.f_files    = 10000
70.               s.f_ffree    = 9900
71.               s.f_favail   = 9900
72.               s.f_fsid     = 23423
73.               s.f_flag     = 0
74.               s.f_namemax  = 10000
75.               return s
76.         end   # statfs
```

```
77.
78.         # Convert string to RFuse::Stat object type
79.         #     @param type [String]
80.         #
81.         #     @return [RFuse::Stat::Constant]
82.         def typeofstat(type)
83.               case type
84.               when "directory"
85.               return RFuse::Stat::S_IFDIR
86.               when "file"
87.               return RFuse::Stat::S_IFREG
88.               when "link"
89.               return RFuse::Stat::S_IFLNK
90.               when "characterSpecial"
91.               return RFuse::Stat::IFCHR
92.               when "blockSpecial"
93.               return RFuse::Stat::IFBLK
94.               when "fifo"
95.               return RFuse::Stat::IFIFO
96.               when "socket"
97.               return RFuse::Stat::IFSOCK
98.               else
99.               return RFuse::Stat::S_IFMT
100.              end
101.        end   # typeofstat
102.
103.        # Utility function to convert StripeFS path to striped paths
104.        #     @param path [String] path to object
105.        #
106.        # @return [Array<String>] object path on stripes
107.        #     @todo write overload with stripe index
```

```
108.        #     @example
109.        #           StripeFS path /1/2/3 for n stripes corresponds to /
    1.1/2.1/3.1,
110.        #           /1.2/2.2/2.3, ..., /1.n/2.n/3.n
111.        def get_striped_paths(path)
112.              striped_paths = @stripes.dup
113.              return striped_paths if path == "/"
114.
115.              pathname = Pathname.new(path)
116.              pathname.each_filename do |filename|
117.                    striped_paths.each_with_index do |stripe, index|
118.                          striped_paths[index] = striped_paths[index] +
    "#{filename}.#{index+1}"
119.                    end
120.              end
121.
122.              striped_paths
123.        end   # get_striped_paths
124.
125.        # Fuse getattr method. {RFuse::Fuse#getattr}
126.        # @param ctx [RFuse::Context]
127.        # @param path [String] object path
128.        #
129.        # @return [RFuse::Stat] stat info of object
130.        def getattr(ctx, path)
131.              if path == "/"
132.                    stat = @rootstat.dup
133.              else
134.                    striped_paths = get_striped_paths(path)
135.                    stat = striped_paths[0].lstat
136.              end
```

```
137.
138.             values = {:uid => @uid, :gid => @gid, :size => stat.size,
139.                   :atime => stat.atime, :mtime => stat.mtime, :ctime
    => stat.ctime,
140.                   :dev => stat.dev, :ino => stat.ino, :nlink =>
    stat.nlink,
141.                   :rdev => stat.rdev, :blksize =>
    stat.blksize, :blocks => stat.blocks}
142.             type = typeofstat(stat.ftype)
143.             if type == RFuse::Stat::S_IFDIR || type ==
    RFuse::Stat::S_IFLNK
144.                   return RFuse::Stat.new(type, stat.mode, values)
145.             end
146.
147.             striped_paths.each_with_index do |stripe, index|
148.                   next if index == 0
149.                   values[:size] += stripe.size
150.             end
151.             return RFuse::Stat.new(type, stat.mode, values)
152.       end   # getattr
153.
154.       # Fuse getattr method. {RFuse::Fuse#fgetattr}
155.       # @param ctx [RFuse::Context]
156.       # @param path [String] object path
157.       #     @param ffi [RFuse::FileInfo] object to store information
    of an
158.       #
    open file such file handle
159.       #
160.       #     @return [RFuse::Stat] stat info of object
161.       def fgetattr(ctx, path, ffi)
```

```
162.              getattr(ctx, path)
163.        end   #     fgetattr
164.
165.        # Fuse readdir method to list contents of directory.
    {RFuse::Fuse#readdir}
166.        # @param ctx [RFuse::Context]
167.        # @param path [String] object path
168.        #     @param filler [RFuse::Filler] structure to collect
    directory entries
169.        def readdir(ctx, path, filler, offset, ffi)
170.              striped_paths = get_striped_paths(path)
171.
172.               filler.push(".",nil,0)
173.               filler.push("..",nil,0)
174.
175.               striped_paths[0].each_child(false) do |child|
176.                    filler.push(child.to_s.chomp(".1"), nil, 0)
177.              end
178.        end   #     readdir
179.
180.        # Fuse mkdir method to make directory. {RFuse::Fuse#mkdir}
181.        # @param ctx [RFuse::Context]
182.        # @param path [String] object path
183.        #     @mode mode [Fixnum] to obtain correct directory
    permissions
184.        def mkdir(ctx, path, mode)
185.              striped_paths = get_striped_paths(path)
186.              striped_paths.each { |stripe| stripe.mkdir(mode) }
187.        end   #     mkdir
188.
```

```
189.        # Fuse rmdir method to remove empty directory.
    {RFuse::Fuse#rmdir}
190.        # @param ctx [RFuse::Context]
191.        # @param path [String] object path
192.        def rmdir(ctx, path)
193.              striped_paths = get_striped_paths(path)
194.              striped_paths.each { |stripe| stripe.rmdir }
195.        end
196.
197.        # Fuse chmod method to change object permissions.
    {RFuse::Fuse#chmod}
198.        # @param ctx [RFuse::Context]
199.        # @param path [String] object path
200.        #     @mode mode [Fixnum] object permissions
201.        def chmod(ctx, path, mode)
202.              get_striped_paths(path).each { |stripe|
    stripe.chmod(mode) }
203.        end   #     chmod
204.
205.        # Fuse chown method to change object ownenership.
    {RFuse::Fuse#chown}
206.        # @param ctx [RFuse::Context]
207.        # @param path [String] object path
208.        #     @mode uid [Fixnum] user id
209.        #     @mode gid [Fixnum] group id
210.        def chown(ctx, path, uid, gid)
211.              get_striped_paths(path).each { |stripe| stripe.chown(uid,
    gid) }
212.        end   #     chown
213.
214.        # Fuse rename method to rename object. {RFuse::Fuse#rename}
```

```
215.        # @param ctx [RFuse::Context]
216.        # @param from [String] object path
217.        # @param to [String] object path
218.        def rename(ctx, from, to)
219.              striped_topaths = get_striped_paths(to)
220.              get_striped_paths(from).each_with_index do |stripe,
    index|
221.                    stripe.rename(striped_topaths[index].to_s)
222.              end
223.        end   #rename
224.
225.        # Fuse open file method. {RFuse::Fuse#open}
226.        # @param ctx [RFuse::Context]
227.        # @param path [String] object path
228.        #     @param ffi [RFuse::FileInfo] stores file information such
    as file handle
229.        def open(ctx, path, ffi)
230.              get_striped_paths(path).each do |stripe|
231.                    stripe.open(ffi.flags) { |f| } # open and close
    file
232.              end
233.        end   #     open
234.
235.        # Fuse creat file method. {RFuse::Fuse#creat}
236.        # @param ctx [RFuse::Context]
237.        # @param path [String] object path
238.        #     @mode mode [Fixnum] object permissions
239.        #     @param ffi [RFuse::FileInfo] stores file information such
    as file handle
240.        def create(ctx, path, mode, ffi)
241.              get_striped_paths(path).each do |stripe|
```

```
242.                     stripe.open("a+") do |f|
243.                           f.chmod(mode)
244.                     end #open and close the file by calling open with
    block
245.               end
246.         end   #     creat
247.
248.         #     Write at offset in stripe
249.         #     @param striped_paths [Array<String>]      array of
    striped paths
250.         #     @param io [StringIO] write buffer
251.         #     @param offset [Fixnum] offset in file
252.         #     @param chunksize [Fixnum] chunksize
253.         #     @param ffi [RFuse::FileInfo] stores file information such
    as file handle
254.         #
255.         #     @return [Fixnum] update offset
256.         def stripe_write(striped_paths, io, offset, chunksize, ffi)
257.               stripes_count = striped_paths.count
258.
259.               # Find stripe number to begin writing
260.               chunks_till_offset_in_file = offset/chunksize
261.               stripe_at_offset = chunks_till_offset_in_file %
    stripes_count
262.
263.               size = io.length - io.tell
264.
265.               # Calculate Offset in Stripe to write at
266.               offset_in_chunk_in_stripe = offset % chunksize  #offset
    relative to a chunk
```

```
267.             chunk_till_offset_in_stripe =
    chunks_till_offset_in_file / stripes_count
268.             offset_in_stripe = (chunk_till_offset_in_stripe *
    chunksize) + offset_in_chunk_in_stripe
269.
270.             # Calculate Size of data to write at offset in stripe
271.             size_available_in_chunk = chunksize -
    offset_in_chunk_in_stripe
272.             size_to_write_in_stripe = size >
    size_available_in_chunk ? size_available_in_chunk : size
273.
274.             # Write data of calcalated size at offset in stripe
275.             ::File.open(striped_paths[stripe_at_offset], "r+") do |
    file|
276.                   file.seek(offset_in_stripe, 0)
277.                   offset +=
    file.write(io.read(size_to_write_in_stripe))
278.             end
279.             # Return updated offset
280.             offset
281.         end   #     stripe_write
282.
283.         # Fuse write file method. {RFuse::Fuse#write}
284.         # @param ctx [RFuse::Context]
285.         # @param path [String] object path
286.         #     @param data [String] data buffer to write
287.         #     @param offset [Fixnum] offset in file to write at
288.         #     @param ffi [RFuse::FileInfo] stores file information such
    as file handle
289.         #
290.         # @return [Fixnum] number of bytes written
```

```
291.        def write(ctx, path, data, offset, ffi)
292.              chunksize = @chunksize
293.              striped_paths = get_striped_paths(path)
294.              offset_in_file = offset
295.              io = StringIO.new(data)
296.              size_of_buffer = io.length
297.              return 0 if 0 == io.length
298.              begin
299.                    offset_in_file = stripe_write(striped_paths, io,
    offset_in_file, chunksize, ffi)
300.              end while io.tell != size_of_buffer
301.
302.              length_written = offset_in_file - offset
303.              length_written
304.        end   #     write
305.
306.        #     Read at offset in stripe
307.        #     @param striped_paths [Array<String>] array of striped
    paths
308.        #     @param size [Fixnum] read size
309.        #     @param offset [Fixnum] offset in file
310.        #     @param chunksize [Fixnum] chunksize
311.        #     @param read_length [Fixnum] number of bytes read
312.        #     @param ffi [RFuse::FileInfo] stores file information such
    as file handle
313.        #
314.        # @return [String] read buffer
315.        def stripe_read(striped_paths, size, offset, chunksize,
    read_length, ffi)
316.              stripes_count = striped_paths.count
317.              # Calculate Stripe number corresponding to offset in file
```

```
318.             chunks_till_offset_in_file = offset/chunksize
319.             stripe_at_offset = chunks_till_offset_in_file %
    stripes_count
320.
321.             size -= read_length
322.             read_buffer = ""
323.             # Calculate Offset in Stripe to read at
324.             offset_in_chunk_in_stripe = offset % chunksize  #offset
    relative to a chunk
325.             chunk_till_offset_in_stripe =
    chunks_till_offset_in_file / stripes_count
326.             offset_in_stripe = (chunk_till_offset_in_stripe *
    chunksize) + offset_in_chunk_in_stripe
327.
328.             # Calculate Size of data to read at offset in stripe
329.             size_to_read_in_chunk = chunksize -
    offset_in_chunk_in_stripe
330.             size_to_read_in_stripe = size > size_to_read_in_chunk ?
    size_to_read_in_chunk : size
331.
332.             # Read data of calcalated size at offset in stripe
333.             ::File.open(striped_paths[stripe_at_offset], "r+") do |
    file|
334.                   file.seek(offset_in_stripe, 0)
335.                   read_buffer = file.read(size_to_read_in_stripe)
336.             end
337.             read_buffer
338.         end   #     stripe_read
339.
340.         # Fuse read file method. {RFuse::Fuse#read}
341.         # @param ctx [RFuse::Context]
```

```
342.        # @param path [String] object path
343.        #     @param offset [Fixnum] offset in file to read from
344.        #     @param ffi [RFuse::FileInfo] stores file information such
    as file handle
345.        #
346.        # @return [String] containing read bytes
347.        def read(ctx, path, size, offset, ffi)
348.              chunksize = @chunksize
349.              striped_paths = get_striped_paths(path)
350.              read_length = 0
351.              buffer = ""
352.              read_buffer = ""
353.
354.              begin
355.                    read_buffer = stripe_read(striped_paths, size,
    offset, chunksize, read_length, ffi)
356.                    if !read_buffer.nil?
357.                          buffer << read_buffer
358.                          len = read_buffer.length
359.                          read_length += len
360.                          offset += len
361.                    end
362.              end while (!read_buffer.nil? && read_length < size)
363.              buffer
364.        end   #     read
365.
366.        # Fuse release method called once after #open.
    {RFuse::Fuse#release}
367.        # @param ctx [RFuse::Context]
368.        # @param path [String] object path
```

```
369.        #     @param ffi [RFuse::FileInfo] stores file information such
    as file handle
370.        def release(ctx, path, ffi)
371.        end   #     release
372.
373.        # Fuse unlink method to delete file. {RFuse::Fuse#release}
374.        # @param ctx [RFuse::Context]
375.        # @param path [String] object path
376.        def unlink(ctx, path)
377.              get_striped_paths(path).each { |path| path.unlink }
378.        end   #     unlink
379.
380.        # Fuse utimens method to set file access and modification
    time. {RFuse::Fuse#utimens}
381.        # @param ctx [RFuse::Context]
382.        # @param path [String] object path
383.        # @param atime [Fixnum] access time
384.        # @param mtime [Fixnum] modified time
385.        def utimens(ctx, path, atime, mtime)
386.              get_striped_paths(path).each do |path|
387.                    path.utime(atime/1000000000, mtime/1000000000)
388.              end
389.        end   #     utimens
390.
391.        # Fuse truncate method. {RFuse::Fuse#truncate}
392.        # @param ctx [RFuse::Context]
393.        # @param path [String] object path
394.        #     @param length [Fixnum] length to truncate
395.        def truncate(ctx, path, length)
396.              striped_paths = get_striped_paths(path)
397.              stripes_count = striped_paths.count
```

```
398.              chunksize = @chunksize
399.              remaining_length = 0
400.              bytes_to_skip = 0
401.              stripe_final_size = []
402.
403.              # Initialize truncate size for each stripe path to
404.              #     multiple of chunksize for complete iteration
405.              (0..stripes_count-1).each_with_index do |index|
406.                    break if length > (chunksize * stripes_count)
407.                    stripe_final_size[index] = (length / (chunksize *
    stripes_count)) * chunksize
408.              end
409.
410.              # Calculate final truncate size for last incomplete
411.              #                         iteration through stripes.
412.              remaining_length = length % (chunksize * stripes_count)
413.
414.              (0..stripes_count-1).each_with_index do |index|
415.                    break if remaining_length < 0
416.                    bytes_to_skip = (remaining_length - chunksize) >
    0 ? chunksize : remaining_length
417.                    remaining_length -= bytes_to_skip;
418.                    stripe_final_size[index] += bytes_to_skip;
419.              end
420.
421.              striped_paths.each_with_index do |stripe, index|
422.                    stripe.truncate(stripe_final_size[index])
423.              end
424.        end   # truncate
425.
426.        # Fuse ftruncate method. {RFuse::Fuse#ftruncate}
```

```
427.        # @param ctx [RFuse::Context]
428.        # @param path [String] object path
429.        # @param length [Fixnum] length to truncate
430.        # @param ffi [RFuse::FileInfo] stores file information such as
    file handle
431.        def ftruncate(ctx, path, length, ffi)
432.              truncate(ctx, path, length)
433.        end   # ftruncate
434.
435.  end   # FS
436. end    # StripeFS
```

3.4. StripeFS::Utils

```
# lib/stripefs/utils.rb
1.   module StripeFS
2.    # Module provides common utility functions
3.    module Utils
4.          extend self
5.          #     Check if variable is blank i.e. empty, nil
6.          #     @param var [Object]
7.          #
8.          #     @return [Boolean] true or false
9.          def is_blank?(var)
10.               var.respond_to?(:empty?) ? var.empty? : !var
11.         end
12.
13.   end   # Utils
14.  end    # StripeFS
```