A project Report on

File System Simulation

Project Report submitted in partial fulfillment of the requirement for the degree of

Bachelor of Technology.

in

Computer Science & Engineering

under the Supervision of

Ms. Ruchi Verma

By

Parth Gupta (Roll No. 111431)

to

Jaypee University of Information and Technology

Waknaghat, Solan – 173234, Himachal Pradesh

Certificate

This is to certify that project report entitled “File System Simulation”, submitted by Parth Gupta in partial fulfillment for the award of degree of Bachelor of Technology in Computer Science & Engineering to Jaypee University of Information Technology, Waknaghat, Solan has been carried out under my supervision.

This work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.

Date: 15 May, 2015

Mrs. Ruchi Verma
Assistant Professor

Acknowledgement

I have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals and organizations. I would like to extend my sincere thanks to all of them.

I am highly indebted to Jaypee University Of Information and Technology for their guidance and constant supervision as well as for providing necessary information regarding the project & also for their support in completing the project.

I would like to express my gratitude towards Mrs. Ruchi Verma for her kind co-operation and encouragement, which help me in completion of this project.

My thanks and appreciations also go to my colleagues in developing the project and people who have willingly helped me out with their abilities.

Table of Contents

List Of Figures

List Of tables

Abstract

The project involves creation of a virtual file system. The project would require creation and manipulation of various data structures to store the contents of the file system. There should be a programmer-level library of functions (API) like my_create, my_delete, my_open, my_close, my_read, my_write, etc to simulate file system operations. The APIs will work on this simulated file system. The file system can model an existing system such as Unix/Windows or you can invent your own. Provide a programmer-level library of functions (API) like my_create, my_delete, my_open, my_close, my_read, my_write, etc to simulate file system operations. The APIs will work on a simulated file system. You could either model your file system on an existing system (e.g. Unix, Dos, etc) or invent your own. Your system should provide support for directory hierarchies.

You should allocate a large file on the actual file system, and treat it as a virtual disk for your file system simulation. In order to implement your API, you would create and manipulate various data structures on your virtual disk to create and manage your file system. For manipulating your file system's data on the virtual disk, you can use C file functions such as fopen, fread, fwrite or corresponding java functions.

For demonstration of the use of your APIs, you will need to write simple user level commands or small programs that use your APIs. The commands should allow navigation of this file system and creation/removal/editing of entries in the file system. A command for listing the contents of a particular node in the file system is also required at a minimum.

You need to implement only one mechanism each for free-space management, data access, and some simple security mechanism, but you must do the analysis for how it compares with other alternative strategies for the same.

The file should support a hierarchical organization of data.

1. INTRODUCTION

1.1 Overview

This report discusses the result of the work done in development of a "File System Simulator" on Java Platform. It is a final year project going in Computer Science Department, Jaypee University Of Information and Technology and aims at the development of an application framework for providing a common platform for facilitating the use of methodological approach, integration of various tools developed during the execution of the project.

1.2 Background and Motivation

As processors, memories, and networks continue to speed up relative to secondary storage, file and disk systems have increasingly become the focus of attention. The Berkeley Log-structured File System (LFS),[1] Redundant Arrays of Independent Disks (RAID),[2] and log-based fault tolerant systems[3,4] are some well-known examples of the newer innovative designs. Analysis of these systems has exposed many subtleties that affect performance. Current technology trends lead us to believe that file system and disk system design and analysis will continue to be one of *the* key areas in computer system design.

Typical performance studies of file systems involve the control of three distinct but related aspects: the disk, the file system, and the workload. In each of these areas, simple models trade off accuracy for modeling ease or tractability. Although useful early results can come from less detailed models with modest effort, these are no longer sufficient when more careful comparisons are desired. Indeed, back-of-the-envelope calculations or simple modeling of the software and/or the disk hardware can yield results that are contrary to real-life performance. We cite below some cases in point, where lack of detail or accuracy in the models led to predictions that turned out to be at variance with actual performance.

1.3 Objective

The final goal of the project was twofold.

1. An Integrated Framework was required for interaction with the various tools (like Software/Hardware Estimation, Partitioning, Synthesis tools etc.) with the platform specification being done in the application itself.

2. Based on the final platform configuration and bindings, an Analysis and Visualization framework was required for getting performance metrics of the system and for visualization of the analysis results and the target platform.

Along with above main goals , capability to design the target platform manually was also desire

1.4 Methodology

To implement the above goals , the following methodology needs to be followed :

1. Specifying the Application and various components of the Architecture.
2. Specifying the bindings between the tasks and the resources either manually or by the design tools.
3. Specifying the port interconnections between the resources.
4. Analysis : Extracting the data required for analysis and the doing the analysis.

1.5 Requirements

Hardware requirements

Number	Description	Alternatives (If available)
1	PC with 2 GB hard-disk and 256 MB RAM	Not-Applicable

Software requirements

Number	Description	Alternatives (If available)
1	Unix/Linux	Windows
2	C/C++/Java compiler	Not Applicable

2. File System

2.1 Introductions

A **file system** (or **filesystem**) is used to control how data is stored and retrieved. Without a file system, information placed in a storage area would be one large body of data with no way to tell where one piece of information stops and the next begins. By separating the data into individual pieces, and giving each piece a name, the information is easily separated and identified. Taking its name from the way paper-based information systems are named, each group of data is called a "file". The structure and logic rules used to manage the groups of information and their names is called a "file system".

There are many different kinds of file systems. Each one has different structure and logic, properties of speed, flexibility, security, size and more. Some file systems have been designed to be used for specific applications. For example, the ISO 9660 file system is designed specifically for optical discs.

File systems can be used on many different kinds of storage devices. Each storage device uses a different kind of media. The most common storage device in use today is a hard drive whose media is a disc that has been coated with a magnetic film. The film has ones and zeros 'written' on it sending electrical pulses to a magnetic "read-write" head. Other media that are used are magnetic tape, optical disc, and flash memory. In some cases, the computer's main memory (RAM) is used to create a temporary file system for short term use.

Some file systems are used on local data storage devices;[1] others provide file access via a network protocol (for example, NFS,[2] SMB, or 9P clients). Some file systems are "virtual", in that the "files" supplied are computed on request (e.g. procfs) or are merely a mapping into a different file system used as a backing store. The file system manages access to both the content of files and the metadata about those files. It is responsible for arranging storage space; reliability, efficiency, and tuning with regard to the physical storage medium are important design considerations.

2.2 Types Of File System

Disk file systems

A *disk file system* takes advantages of the ability of disk storage media to randomly address data in a short amount of time. Additional considerations include the speed of accessing data following that initially requested and the anticipation that the following data may also be requested. This permits multiple users (or processes) access to various data on the disk without regard to the sequential location of the data. Examples include FAT (FAT12, FAT16, FAT32), exFAT, NTFS, HFS and HFS+, HPFS,UFS, ext2, ext3, ext4, XFS, btrfs, ISO 9660, Files-11, Veritas File System, VMFS, ZFS, ReiserFS and UDF. Some disk file systems are journaling file systems orversioning file systems.

Optical discs

ISO 9660 and Universal Disk Format (UDF) are two common formats that target Compact Discs, DVDs and Blu-ray discs. Mount Rainier is an extension to UDF supported since 2.6 series of the Linux kernel and since Windows Vista that facilitates rewriting to DVDs.

Flash file systems

A *flash file system* considers the special abilities, performance and restrictions of flash memory devices. Frequently a disk file system can use a flash memory device as the underlying storage media but it is much better to use a file system specifically designed for a flash device.

Tape file systems

A *tape file system* is a file system and tape format designed to store files on tape in a self-describing form. Magnetic tapes are sequential storage media with significantly longer random data access times than disks, posing challenges to the creation and efficient management of a general-purpose file system.

Tape formatting

Writing data to a tape is often a significantly time-consuming process that may take several hours. Similarly, completely erasing or formatting a tape can also take several hours. With many data tape technologies it is not necessary to format the tape before over-writing new data to the tape. This is due to the inherently destructive nature of overwriting data on sequential media.

Database file systems

Another concept for file management is the idea of a database-based file system. Instead of, or in addition to, hierarchical structured management, files are identified by their characteristics, like type of file, topic, author, or similar rich metadata.

Transactional file systems

Some programs need to update multiple files "all at once". For example, a software installation may write program binaries, libraries, and configuration files. If the software installation fails, the program may be unusable. If the installation is upgrading a key system utility, such as the command shell, the entire system may be left in an unusable state.

Network file systems

A *network file system* is a file system that acts as a client for a remote file access protocol, providing access to files on a server. Examples of network file systems include clients for the NFS, AFS, SMB protocols, and file-system-like clients for FTP and WebDAV.

Shared disk file systems

A *shared disk file system* is one in which a number of machines (usually servers) all have access to the same external disk subsystem (usually a SAN). The file system arbitrates access to that subsystem, preventing write collisions. Examples include GFS2 from Red

Hat, GPFS from IBM, SFS from DataPlow, CXFS from SGI and StorNext from Quantum Corporation.

3. File systems and operating systems

3.1 Introduction

Many operating systems include support for more than one file system. Sometimes the OS and the file system are so tightly interwoven it is difficult to separate out file system functions.

There needs to be an interface provided by the operating system software between the user and the file system. This interface can be textual (such as provided by acommand line interface, such as the Unix shell, or OpenVMS DCL) or graphical (such as provided by a graphical user interface, such as file browsers). If graphical, the metaphor of the *folder*, containing documents, other files, and nested folders is often used (see also: directory and folder).

3.2 Unix-like operating systems

Unix-like operating systems create a virtual file system, which makes all the files on all the devices appear to exist in a single hierarchy. This means, in those systems, there is one root directory, and every file existing on the system is located under it somewhere. Unix-like systems can use a RAM disk or network shared resource as its root directory.

Unix-like systems assign a device name to each device, but this is not how the files on that device are accessed. Instead, to gain access to files on another device, the operating system must first be informed where in the directory tree those files should appear. This process is called mounting a file system. For example, to access the files on a CD-ROM, one must tell the operating system "Take the file system from this CD-ROM and make it appear under such-and-such directory". The directory given to the operating system is called the *mount point* – it might, for example, be /media. The /media directory exists on many Unix systems (as specified in theFilesystem Hierarchy Standard) and is intended specifically for use

as a mount point for removable media such as CDs, DVDs, USB drives or floppy disks. It may be empty, or it may contain subdirectories for mounting individual devices. Generally, only the administrator (i.e. root user) may authorize the mounting of file systems.

Unix-like operating systems often include software and tools that assist in the mounting process and provide it new functionality. Some of these strategies have been coined "auto-mounting" as a reflection of their purpose.

- In many situations, file systems other than the root need to be available as soon as the operating system has booted. All Unix-like systems therefore provide a facility for mounting file systems at boot time. System administrators define these file systems in the configuration file fstab (*vfstab* in Solaris), which also indicates options and mount points.
- In some situations, there is no need to mount certain file systems at boot time, although their use may be desired thereafter. There are some utilities for Unix-like systems that allow the mounting of predefined file systems upon demand.
- Removable media have become very common with microcomputer platforms. They allow programs and data to be transferred between machines without a physical connection. Common examples include USB flash drives, CD-ROMs, and DVDs. Utilities have therefore been developed to detect the presence and availability of a medium and then mount that medium without any user intervention.
- Progressive Unix-like systems have also introduced a concept called **supermounting**; see, for example, the Linux supermount-ng project. For example, a floppy disk that has been supermounted can be physically removed from the system. Under normal circumstances, the disk should have been synchronized and then unmounted before its removal. Provided synchronization has occurred, a different disk can be inserted into the drive. The system automatically notices that the disk has changed and updates the mount point contents to reflect the new medium.
- An automounter will automatically mount a file system when a reference is made to the directory atop which it should be mounted. This is usually used for file systems on network servers, rather than relying on events such as the

insertion of media, as would be appropriate for removable media.

Unix-like operating systems create a virtual file system, which makes all the files on all the devices appear to exist in a single hierarchy. This means, in those systems, there is one root directory, and every file existing on the system is located under it somewhere. Unix-like systems can use a RAM disk or network shared resource as its root directory.

Unix-like systems assign a device name to each device, but this is not how the files on that device are accessed. Instead, to gain access to files on another device, the operating system must first be informed where in the directory tree those files should appear. This process is called mounting a file system. For example, to access the files on a CD-ROM, one must tell the operating system "Take the file system from this CD-ROM and make it appear under such-and-such directory". The directory given to the operating system is called the *mount point* – it might, for example, be /media. The /media directory exists on many Unix systems (as specified in theFilesystem Hierarchy Standard) and is intended specifically for use as a mount point for removable media such as CDs, DVDs, USB drives or floppy disks. It may be empty, or it may contain subdirectories for mounting individual devices. Generally, only the administrator (i.e. root user) may authorize the mounting of file systems.

Unix-like operating systems often include software and tools that assist in the mounting process and provide it new functionality. Some of these strategies have been coined "auto-mounting" as a reflection of their purpose.

- In many situations, file systems other than the root need to be available as soon as the operating system has booted. All Unix-like systems therefore provide a facility for mounting file systems at boot time. System administrators define these file systems in the configuration file fstab (*vfstab* in Solaris), which also indicates options and mount points.
- In some situations, there is no need to mount certain file systems at boot time, although their use may be desired thereafter. There are some utilities for Unix-like systems that allow the mounting of predefined file systems upon demand.
- Removable media have become very common with microcomputer platforms. They allow programs and data to be transferred between machines without a

physical connection. Common examples include USB flash drives, CD-ROMs, and DVDs. Utilities have therefore been developed to detect the presence and availability of a medium and then mount that medium without any user intervention.

- Progressive Unix-like systems have also introduced a concept called **supermounting**; see, for example, the Linux supermount-ng project. For example, a floppy disk that has been supermounted can be physically removed from the system. Under normal circumstances, the disk should have been synchronized and then unmounted before its removal. Provided synchronization has occurred, a different disk can be inserted into the drive. The system automatically notices that the disk has changed and updates the mount point contents to reflect the new medium.
- An automounter will automatically mount a file system when a reference is made to the directory atop which it should be mounted. This is usually used for file systems on network servers, rather than relying on events such as the insertion of media, as would be appropriate for removable media.

3.3 Linux

Linux supports many different file systems, but common choices for the system disk on a block device include the ext* family (such as ext2, ext3 and ext4), XFS, JFS,ReiserFS and btrfs. For raw flash without a flash translation layer (FTL) or Memory Technology Device (MTD), there is UBIFS, JFFS2, and YAFFS, among others.SquashFS is a common compressed read-only file system.

3.4 Solaris

The Sun Microsystems Solaris operating system in earlier releases defaulted to (non-journaled or non-logging) UFS for bootable and supplementary file systems. Solaris defaulted to, supported, and extended UFS.

Support for other file systems and significant enhancements were added over time, including Veritas Software Corp. (Journaling) VxFS, Sun Microsystems (Clustering)QFS, Sun Microsystems (Journaling) UFS, and Sun Microsystems (open source, poolable, 128 bit compressible, and error-correcting) ZFS.

Kernel extensions were added to Solaris to allow for bootable Veritas VxFS operation. Logging or Journaling was added to UFS in Sun's Solaris 7. Releases of Solaris 10, Solaris Express, OpenSolaris, and other open source variants of the Solaris operating system later supported bootable ZFS.

Logical Volume Management allows for spanning a file system across multiple devices for the purpose of adding redundancy, capacity, and/or throughput. Legacy environments in Solaris may use Solaris Volume Manager (formerly known as Solstice DiskSuite). Multiple operating systems (including Solaris) may use Veritas Volume Manager. Modern Solaris based operating systems eclipse the need for Volume Management through leveraging virtual storage pools in ZFS.

3.4 OS X

OS X uses a file system inherited from classic Mac OS called HFS Plus. Apple also uses the term "Mac OS Extended" HFS Plus is a metadata-rich and case-preserving but (usually) case-insensitive file system. Due to the Unix roots of OS X, Unix permissions were added to HFS Plus. Later versions of HFS Plus added journaling to prevent corruption of the file system structure and introduced a number of optimizations to the allocation algorithms in an attempt to defragment files automatically without requiring an external defragmenter.

Filenames can be up to 255 characters. HFS Plus uses Unicode to store filenames. On OS X, the file type can come from the type code, stored in file's metadata, or the filename extension.

HFS Plus has three kinds of links: Unix-style hard links, Unix-style symbolic links and aliases. Aliases are designed to maintain a link to their original file even if they are moved or renamed; they are not interpreted by the file system itself, but by the File Manager code in user land.

OS X also supported the UFS file system, derived from the BSD Unix Fast File System via Next STEP. However, as of Mac OS X Leopard, OS X could no longer be installed on a UFS volume, nor can a pre-Leopard system installed on a UFS volume be upgraded to Leopard. As of Mac OS X Lion UFS support was completely dropped. Newer versions of OS X are capable of reading and writing to the legacy FAT file systems (16 & 32) common on Windows. They are also capable of *reading* the newerNTFS file systems for Windows. In order to *write* to NTFS file systems on OS X versions prior to 10.6 (Snow Leopard) third party software is necessary. Mac OS X 10.6 (Snow Leopard) and later allows writing to NTFS file systems, but only after a non-trivial system setting change (third party software exists that automates this).

3.5 PC-BSD

PC-BSD is a desktop version of FreeBSD, which inherits FreeBSD's ZFS support, similarly to FreeNAS. The new graphical installer of PC-BSD can handle */ (root) on ZFS* and RAID-Z pool installs and disk encryption using Geli right from the start in an easy convenient (GUI) way. The current PC-BSD 9.0+ 'Isotope Edition' has ZFS filesystem version 5 and ZFS storage pool version 28.

3.6 Plan 9

Plan 9 from Bell Labs treats *everything* as a file, and accessed as a file would be (i.e., no ioctl or mmap): networking, graphics, debugging, authentication, capabilities, encryption, and other services are accessed via I-O operations on file descriptors. The 9P protocol removes the difference between local and remote files.

These file systems are organized with the help of private, per-process namespaces, allowing each process to have a different view of the many file systems that provide resources in a distributed system.

The Inferno operating system shares these concepts with Plan 9.

3.7 Microsoft Windows

Windows makes use of the FAT, NTFS, exFAT and ReFS file systems (the last of these is only supported and usable in Windows Server 2012; Windows cannot boot from it).

Windows uses a *drive letter* abstraction at the user level to distinguish one disk or partition from another. For example, the path `C:\WINDOWS` represents a directory `WINDOWS` on the partition represented by the letter C. Drive C: is most commonly used for the primary hard disk partition, on which Windows is usually installed and from which it boots. This "tradition" has become so firmly ingrained that bugs exist in many applications which make assumptions that the drive that the operating system is installed on is C. The use of drive letters, and the tradition of using "C" as the drive letter for the primary hard disk partition, can be traced to MS-DOS, where the letters A and B were reserved for up to two floppy disk drives. This in turn derived from CP/M in the 1970s, and ultimately from IBM's CP/CMS of 1967.

3.8 FAT

The family of FAT file systems is supported by almost all operating systems for personal computers, including all versions of Windows and MS-DOS/PC DOS and DR-DOS. (PC DOS is an OEM version of MS-DOS, MS-DOS was originally based on SCP's 86-DOS. DR-DOS was based on Digital Research's Concurrent DOS, a successor of CP/M-86.) The FAT file systems are therefore well suited as a universal exchange format between computers and devices of most any type and age.

The FAT file system traces its roots back to an (incompatible) 8-bit FAT precursor in Standalone Disk BASIC and the short-lived MDOS/MIDAS project.

Over the years, the file system has been expanded from FAT12 to FAT16 and FAT32. Various features have been added to the file system including subdirectories, code page support, extended attributes, and long filenames. Third parties such as Digital Research have incorporated optional support for deletion tracking, and volume/directory/file-based multi-user security schemes to support file and directory

passwords and permissions such as read/write/execute/delete access rights. Windows does not support most of these extensions.

The FAT12 and FAT16 file systems had a limit on the number of entries in the root directory of the file system and had restrictions on the maximum size of FAT-formatted disks or partitions.

FAT32 addresses the limitations in FAT12 and FAT16, except for the file size limit of close to 4 GB, but it remains limited compared to NTFS.

FAT12, FAT16 and FAT32 also have a limit of eight characters for the file name, and three characters for the extension (such as .exe). This is commonly referred to as the 8.3 filename limit. VFAT, an optional extension to FAT12, FAT16 and FAT32, introduced in Windows 95 and Windows NT 3.5, allowed long file names (LFN) to be stored in the FAT file system in a backwards-compatible fashion.

3.9 NTFS

NTFS, introduced with the Windows NT operating system in 1993, allowed ACL-based permission control. Other features also supported by NTFS include hard links, multiple file streams, attribute indexing, quota tracking, sparse files, encryption, compression, and reparse points (directories working as mount-points for other file systems, symlinks, junctions, remote storage links).

3.10 exFAT

exFAT is a proprietary and patent-protected file system with certain advantages over NTFS with regard to file system overhead.

exFAT is not backward compatible with FAT file systems such as FAT12, FAT16 or FAT32. The file system is supported with newer Windows systems, such as Windows Server 2003, Windows Vista, Windows 2008, Windows 7, Windows 8, and more recently, support has been added for Windows XP.[20]

exFAT is supported in Mac OS X starting with version 10.6.5 (Snow Leopard).[19] Support in other operating systems is sparse since Microsoft has not published the specifications of the file system and implementing support for exFAT requires a license.

4. Aspects of file systems

4.1 Space management

Example of slack space, demonstrated with 4,096-byte NTFS clusters: 100,000 files, each 5 bytes per file, equals 500,000 bytes of actual data, but requires 409,600,000 bytes of disk space to store

File systems allocate space in a granular manner, usually multiple physical units on the device. The file system is responsible for organizing files and directories, and keeping track of which areas of the media belong to which file and which are not being used. For example, in Apple DOS of the early 1980s, 256-byte sectors on 140 kilobyte floppy disk used a *track/sector map*.

This results in unused space when a file is not an exact multiple of the allocation unit, sometimes referred to as *slack space*. For a 512-byte allocation, the average unused space is 256 bytes. For 64 KB clusters, the average unused space is 32 KB. The size of the allocation unit is chosen when the file system is created. Choosing the allocation size based on the average size of the files expected to be in the file system can minimize the amount of unusable space. Frequently the default allocation may provide reasonable usage. Choosing an allocation size that is too small results in excessive overhead if the file system will contain mostly very large files.

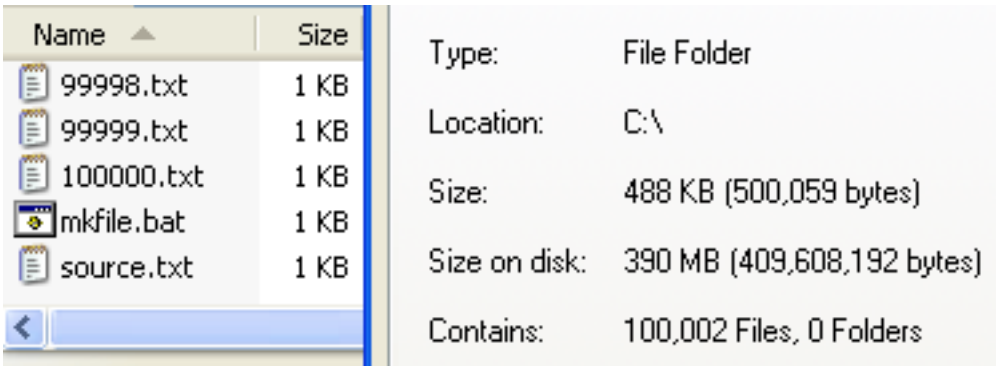


Figure 1: Example of slack space

4.2 Filenames

A **filename** (or **file name**) is used to identify a storage location in the file system. Most file systems have restrictions on the length of filenames. In some file systems, filenames are not case sensitive (i.e., filenames such as `FOO` and `foo` refer to the same file); in others, filenames are case sensitive (i.e., the names `FOO`, `Foo`and `foo` refer to three separate files).

Most modern file systems allow filenames to contain a wide range of characters from the Unicode character set. Most file system interface utilities, however, have restrictions on the use of certain special characters, disallowing them within filenames (the file system may use these special characters to indicate a device, device type, directory prefix, or file type). However, these special characters might be allowed by, for example, enclosing the filename with double quotes ("). For simplicity, special characters are generally discouraged within filenames.

4.3 Directories

File systems typically have **directories** (also called **folders**) which allow the user to group files into separate collections. This may be implemented by associating the file name with an index in a table of contents or an inode in a Unix-like file system. Directory structures may be flat (i.e. linear), or allow hierarchies where directories may contain subdirectories. The first file system to support arbitrary hierarchies of

directories was used in the Multics operating system.[3] The native file systems of Unix-like systems also support arbitrary directory hierarchies, as do, for example, Apple's Hierarchical File System, and its successor HFS+ in classic Mac OS (HFS+ is still used in Mac OS X), the FAT file system in MS-DOS 2.0 and later and Microsoft Windows, the NTFS file system in the Windows NT family of operating systems, and the ODS-2 (On-Disk Structure-2) and higher levels of the Files-11 file system in OpenVMS.

File systems typically have **directories** (also called **folders**) which allow the user to group files into separate collections. This may be implemented by associating the file name with an index in a table of contents or an inode in a Unix-like file system. Directory structures may be flat (i.e. linear), or allow hierarchies where directories may contain subdirectories. The first file system to support arbitrary hierarchies of directories was used in the Multics operating system.[3] The native file systems of Unix-like systems also support arbitrary directory hierarchies, as do, for example, Apple's Hierarchical File System, and its successor HFS+ in classic Mac OS (HFS+ is still used in Mac OS X), the FAT file system in MS-DOS 2.0 and later and Microsoft Windows, the NTFS file system in the Windows NT family of operating systems, and the ODS-2 (On-Disk Structure-2) and higher levels of the Files-11 file system in OpenVMS.

4.4 Metadata

Other bookkeeping information is typically associated with each file within a file system. The length of the data contained in a file may be stored as the number of blocks allocated for the file or as a byte count. The time that the file was last modified may be stored as the file's timestamp. File systems might store the file creation time, the time it was last accessed, the time the file's metadata was changed, or the time the file was last backed up. Other information can include the file's device type (e.g. block, character, socket, subdirectory, etc.), its owner user ID and group ID, its access permissions and other file attributes (e.g. whether the file is read-only,executable, etc.).

A file system stores all the metadata associated with the file—including the file name, the length of the contents of a file, and the location of the file in the folder hierarchy—separate from the contents of the file.

Most file systems store the names of all the files in one directory in one place—the directory table for that directory—which is often stored like any other file. Many file systems put only some of the metadata for a file in the directory table, and the rest of the metadata for that file in a completely separate structure, such as the inode.

Most file systems also store metadata not associated with any one particular file. Such metadata includes information about unused regions -- free space bitmap, block availability map—and information about bad sectors. Often such information about an allocation group is stored inside the allocation group itself.

Additional attributes can be associated on file systems, such as NTFS, XFS, ext2, ext3, some versions of UFS, and HFS+, using extended file attributes. Some file systems provide for user defined attributes such as the author of the document, the character encoding of a document or the size of an image.

Some file systems allow for different data collections to be associated with one file name. These separate collections may be referred to as *streams* or *forks*. Apple has long used a forked file system on the Macintosh, and Microsoft supports streams in NTFS. Some file systems maintain multiple past revisions of a file under a single file name; the filename by itself retrieves the most recent version, while prior saved version can be accessed using a special naming convention such as "filename;4" or "filename(-4)" to access the version four saves ago.

See comparison of file systems#Metadata for details on which file systems support which kinds of metadata.

Other bookkeeping information is typically associated with each file within a file system. The length of the data contained in a file may be stored as the number of blocks allocated for the file or as a byte count. The time that the file was last modified may be stored as the file's timestamp. File systems might store the file creation time, the time it was last accessed, the time the file's metadata was changed, or the time the file was last backed up. Other information can include the file's device type (e.g. block, character, socket, subdirectory, etc.), its owner user ID and group ID, its access permissions and other file attributes (e.g. whether the file is read-only,executable, etc.).

A file system stores all the metadata associated with the file—including the file name, the length of the contents of a file, and the location of the file in the folder hierarchy—separate from the contents of the file.

Most file systems store the names of all the files in one directory in one place—the directory table for that directory—which is often stored like any other file. Many file systems put only some of the metadata for a file in the directory table, and the rest of the metadata for that file in a completely separate structure, such as the inode.

Most file systems also store metadata not associated with any one particular file. Such metadata includes information about unused regions -- free space bitmap, block availability map—and information about bad sectors. Often such information about an allocation group is stored inside the allocation group itself.

Additional attributes can be associated on file systems, such as NTFS, XFS, ext2, ext3, some versions of UFS, and HFS+, using extended file attributes. Some file systems provide for user defined attributes such as the author of the document, the character encoding of a document or the size of an image.

Some file systems allow for different data collections to be associated with one file name. These separate collections may be referred to as *streams* or *forks*. Apple has long used a forked file system on the Macintosh, and Microsoft supports streams in NTFS. Some file systems maintain multiple past revisions of a file under a single file name; the filename by itself retrieves the most recent version, while prior saved version can be accessed using a special naming convention such as "filename;4" or "filename(-4)" to access the version four saves ago.

See comparison of file systems#Metadata for details on which file systems support which kinds of metadata.

4.5 File system as an abstract user interface

In some cases, a file system may not make use of a storage device but can be used to organize and represent access to any data, whether it is stored or dynamically generated (e.g. procfs).

4.6 Utilities

The difference between a utility and a built-in core command function is arbitrary, depending on the design of the operating system, and on the memory and space limitations of the hardware. For example, Microsoft MS-DOS uses a utility for formatting and a built-in command for simple file copying, while in the Apple DOSformatting is a built-in command and simple file copying is performed by using a utility.

File systems include utilities to initialize, alter parameters of and remove an instance of the file system. Some include the ability to extend or truncate the space allocated to the file system.

Directory utilities may be used to create, rename and delete *directory entries*, which are also known as *dentries* (singular: *dentry*), and to alter metadata associated with a directory. Directory utilities may also include capabilities to create additional links to a directory (hard links in Unix), to rename parent links (".." in Unix-likeoperating systems),[*clarification needed*] and to create bidirectional links to files.

File utilities create, list, copy, move and delete files, and alter metadata. They may be able to truncate data, truncate or extend space allocation, append to, move, and modify files in-place. Depending on the underlying structure of the file system, they may provide a mechanism to prepend to, or truncate from, the beginning of a file, insert entries into the middle of a file or delete entries from a file.

Utilities to free space for deleted files, if the file system provides an undelete function, also belong to this category.

Some file systems defer operations such as reorganization of free space, secure erasing of free space, and rebuilding of hierarchical structures by providing utilities to perform these functions at times of minimal activity. Included in this category is the infamous defragmentation utility.

Some of the most important features of file system utilities involve supervisory activities which may involve bypassing ownership or direct access to the underlying device. These include high-performance backup and recovery, data replication and reorganization of various data structures and allocation tables within the file system.

4.7 Restricting and permitting access

There are several mechanisms used by file systems to control access to data. Usually the intent is to prevent reading or modifying files by a user or group of users. Another reason is to ensure data is modified in a controlled way so access may be restricted to a specific program. Examples include passwords stored in the metadata of the file or elsewhere and file permissions in the form of permission bits, access control lists, or capabilities. The need for file system utilities to be able to access the data at the media level to reorganize the structures and provide efficient backup usually means that these are only effective for polite users but are not effective against intruders.

Methods for encrypting file data are sometimes included in the file system. This is very effective since there is no need for file system utilities to know the encryption seed to effectively manage the data. The risks of relying on encryption include the fact that an attacker can copy the data and use brute force to decrypt the data. Losing the seed means losing the data.

4.8 Maintaining integrity

One significant responsibility of a file system is to ensure that, regardless of the actions by programs accessing the data, the structure remains consistent. This includes actions taken if a program modifying data terminates abnormally or neglects to inform the file system that it has completed its activities. This may include updating the metadata, the directory entry and handling any data that was buffered but not yet updated on the physical storage media.

Other failures which the file system must deal with include media failures or loss of connection to remote systems.

In the event of an operating system failure or "soft" power failure, special routines in the file system must be invoked similar to when an individual program fails.

The file system must also be able to correct damaged structures. These may occur as a result of an operating system failure for which the OS was unable to notify the file system, power failure or reset.

The file system must also record events to allow analysis of systemic issues as well as problems with specific files or directories.

4.9 User data

The most important purpose of a file system is to manage user data. This includes storing, retrieving and updating data.

Some file systems accept data for storage as a stream of bytes which are collected and stored in a manner efficient for the media. When a program retrieves the data it specifies the size of a memory buffer and the file system transfers data from the media to the buffer. Sometimes a runtime library routine may allow the user program to define a *record* based on a library call specifying a length. When the user program reads the data the library retrieves data via the file system and returns a *record*.

Some file systems allow the specification of a fixed record length which is used for all write and reads. This facilitates updating records.

An identification for each record, also known as a key, makes for a more sophisticated file system. The user program can read, write and update records without regard to their location. This requires complicated management of blocks of media usually separating key blocks and data blocks. Very efficient algorithms can be developed with pyramid structure for locating records.

4.10 Using a file system

Utilities, language specific run-time libraries and user programs use file system APIs to make requests of the file system. These include data transfer, positioning, updating metadata, managing directories, managing access specifications, and removal.

4.11 Multiple file systems within a single system

Frequently retail systems are configured with a single file system occupying the entire hard disk.

Another approach is to partition the disk so that several file systems with different attributes can be used. One file system, for use as browser cache, might be configured

with a small allocation size. This has the additional advantage of keeping the frantic activity of creating and deleting files typical of browser activity in a narrow area of the disk and not interfering with allocations of other files. A similar partition might be created for email. Another partition, and file system might be created for the storage of audio or video files with a relatively large allocation. One of the file systems may normally be set *read-only* and only periodically be set writable.

A third approach, which is mostly used in cloud systems, is to use "disk images" to house additional file systems, with the same attributes or not, within another (host) file system as a file. A common example is virtualization: one user can run an experimental Linux distribution (using the ext4 file system) in a virtual machine under his/her production Windows environment (using NTFS). The ext4 file system resides in a disk image, which is treated as a file (or multiple files, depending on the hypervisor and settings) in the NTFS host file system.

Having multiple file systems on a single system has the additional benefit that in the event of a corruption of a single partition, the remaining file systems will frequently still be intact. This includes virus destruction of the *system* partition or even a system that will not boot. File system utilities which require dedicated access can be effectively completed piecemeal. In addition, defragmentation may be more effective. Several system maintenance utilities, such as virus scans and backups, can also be processed in segments. For example it is not necessary to backup the file system containing videos along with all the other files if none have been added since the last backup. As for the image files, one can easily "spin off" differential images which contain only "new" data written to the master (original) image. Differential images can be used for both safety concerns (as a "disposable" system - can be quickly restored if destroyed or contaminated by a virus, as the old image can be removed and a new image can be created in matter of seconds, even without automated procedures) and quick virtual machine deployment (since the differential images can be quickly spawned using a script in batches).

4.12 Design limitations

All file systems have some functional limit that defines the maximum storable data capacity within that system. These functional limits are a best-guess effort by the

designer based on how large the storage systems are right now and how large storage systems are likely to become in the future. Disk storage has continued to increase at near exponential rates (see Moore's law), so after a few years, file systems have kept reaching design limitations that require computer users to repeatedly move to a newer system with ever-greater capacity.

File system complexity typically varies proportionally with the available storage capacity. The file systems of early 1980s home computers with 50 KB to 512 KB of storage would not be a reasonable choice for modern storage systems with hundreds of gigabytes of capacity. Likewise, modern file systems would not be a reasonable choice for these early systems, since the complexity of modern file system structures would quickly consume or even exceed the very limited capacity of the early storage systems.

5. Techniques for File System Simulation

5.1 INTRODUCTION

Because file and disk systems are such critical components of modern computer systems, understanding and improving their performance is of great importance. Several techniques can be used to assess the performance of these systems, including abstract performance models, functional simulations, and measurements of a complete implementation. All have their place; we concentrate here on the use of simulations for detailed 'What if?' performance studies. The main advantage of using abstract models as opposed to detailed simulation is their ability to provide adequate answers to many performance questions without the need to represent a great deal of system detail. However, abstract models make simplifying assumptions about aspects of the system that may be important, particularly in the later stages of a study when the design space has been narrowed and subtle issues are being considered. The work we describe here came about when we were doing some design studies of the interactions between current file system designs and new disk systems. In this environment, direct measurement was of course not possible, and we felt that an analytic model would be inadequate for our needs, at least partly because it would have difficulty representing the interactions between performance non-linearity's in the disk system and the file system layout and request-sequencing policies.

5.2 SIMULATOR OVERVIEW

This section introduces the components that comprise our simulation environment for file and disk systems. Our approach is to derive workloads from I/O traces gathered from real systems, and feed these into real file system code sitting on top of a detailed disk model that has been calibrated against real disks. Additional software, which we refer to as *scaffolding*, holds all this together. Figure below shows how these components interact.

We use the term ‘simulator’ in this paper to refer collectively to all the components shown on the right side of Figure below, while we use the term ‘file system simulator’ to refer to the component that mimics just the behavior of the file system code.

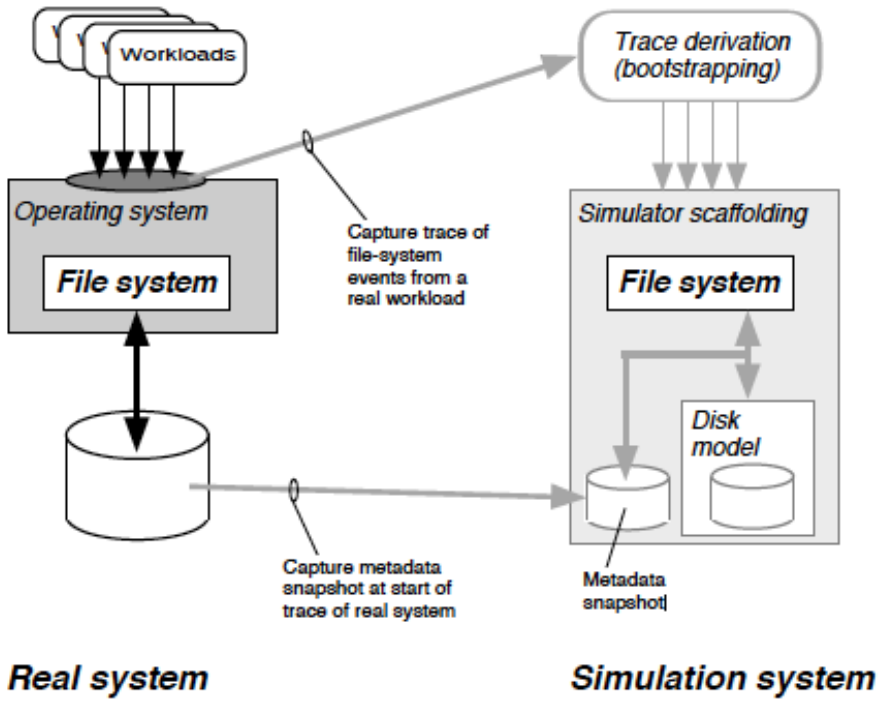


Figure 2: Simulator Framework

As Figure above suggests, we first gather traces from the real system running a real workload. At beginning of the trace period, we take a snapshot of the file system metadata on the disk or disks being studied. This is the information kept by the file system to map <file, offset> pairs into disk block addresses. The trace is optionally used to derive a set of additional traces. Trace requests are supplied to the simulator. Within the simulator, the scaffolding component replays these to create a workload for the file system simulator. The file system simulator, in turn, emits requests to the disk model, which usually just performs timing calculations. In addition, if the request is a read to the metadata, the scaffolding intercepts the request and satisfies

it from the previously-recorded snapshot; a metadata write is used to update the snapshot.
The remainder of this section provides an overview of each simulator component; they are discussed in greater detail in the subsequent three sections.

5.3 Workload Traces

The accuracy of any performance study depends on both the quality of the model and on the quality of the workload representation that is used. The two usual sources for simulation studies are traces and synthetic workload models.
Synthetic workloads are considered more flexible than traces, and do not require significant storage because they are generated on the fly. However, in order to be realistic, synthetic workload models tend to be elaborate, difficult to parameterize, and specific to a single environment. One sample of a synthetic NFS-workload generator[13] uses 24 parameters to describe the workload—a wealth of detail that is not easy to gather.

Instead, our approach is to use trace-driven workloads, but to extend their utility through a technique known as *bootstrapping*, which is described further in the next section. This allows us to collect a single set of traces, and to generate additional sets while retaining certain statistical guarantees with respect to the original.

For investigating file systems that do caching, the most useful results are obtained by tracing requests at the system-call level: byte-aligned reads and writes, plus various control calls, such as file open and close, change directory, and so on.

We did our work on the HP-UX operating system, a POSIX-compliant Unix® system that runs on HP 9000 PA-RISC series 800 and series 700 systems.[14] The HP-UX system has a built-in measurement facility that can be used selectively to trace system events; several other operating systems have similar facilities, or one can be added relatively easily given access to the system's source code.

For our case study, we asked the kernel measurement system to gather information about all file-system related system calls, *fork* and *exit* system calls, and context switches. Together, these allowed our simulation scaffold to replay essentially exactly the sequence of events that took place in the original system.

There are a few important attributes of such trace-gathering systems for work of this kind: the traces must be complete (no records must be missed), they must be accurate (not contain invalid data), they must have precise timestamps (resolution of a few microseconds is acceptable), and gathering them must not disturb the system under test very much. The HP-UX trace facility met all these needs well: its timestamp resolution is 1 μsecond, and the running time for the tests we conducted increased by less than 5%—at least partly because we did trace compaction and analysis off-line.

The next section describes one of the contributions of this paper: a method called 'boot- strapping' for on-the-fly generation of additional traces from a previously collected set of traces. However, from the point of view of the simulator itself, each derived trace is handled the same way, so we will defer further discussion of this aspect for now.

5.4 Metadata Snapshot

Before tracing is begun, a snapshot is made of the metadata on each of the file systems used by the workload being traced. This snapshot includes a copy of the file systems' naming hierarchy, i.e., the directories, overall size information, and a copy of the layout information. In our case, the HP-UX file system uses a slightly modified version of the original 4.2BSD Fast File System,[15] so this data included inode and cylinder-group maps.

The metadata snapshot is a copy of the data needed by the file system itself. The snapshot allows the scaffolding to provide the file system code under test the same data that it would have had access to, had it been running on the real system. In

particular, the file system reads the directory data to do name lookups, and uses the layout information to turn user-level reads and writes into disk operations. As the simulation progresses, the file system under test modifies the metadata as a result of trace-driven user-level requests, and the scaffolding faithfully performs the requested updates so that future requests to the metadata will return the correct data.
Since the snapshot contains no data files, it is of modest size: a few percent of the total disk
space being simulated. This is possible because we do not simulate the contents of user-data blocks, just their movement. Accesses to the metadata snapshot by the simulator scaffolding go though the real file system of the machine used to run the simulation, and so are subject to caching, which improves the elapsed simulation time.

5.5 Scaffolding

The scaffolding is the glue that binds together the entire simulation. It provides the following facilities:

1. Lightweight threads, i.e., execution contexts, which are used to simulate processes making file system calls and the concurrent execution of activities inside our disk simulator.
2. Time-advance and other mechanisms needed for the discrete-event simulation being performed.
3. Emulation of the kernel procedures that are accessed by the real file system. For example,
sleep and *wakeup* calls are mapped onto synchronization primitives derived from the underlying lightweight thread library.
4. Software to access the metadata snapshot when requested by the file system code.
5. Software to manage the correct replay of an input trace.

At a high level, the working of the scaffolding is quite straightforward. The scaffolding uses one lightweight thread to simulate each independent process encountered in the trace. It then reads trace records that are fed to threads simulating

user processes until the trace is exhausted, or the simulation has reached a sufficiently stable state that the desired confidence intervals have been achieved.

Each trace record is handed to the thread that is emulating the appropriate process. Most of these requests are read or write operations, which turn into calls on the file system code, and perhaps generate one or more simulated disk requests. Whenever the real process would have spent real time—e.g., while waiting for a disk access to complete—the thread is blocked, and waits for simulated time to advance to the appropriate point before it is allowed to proceed again. Once the request has been completed the thread goes back to wait for the next request for it to simulate.

If the operation being simulated is a *fork*, a new thread is created and associated with the child process, which then proceeds to accept and process requests, while the parent continues. Asynchronous disk requests do not cause the thread to delay.

Occasionally, the metadata snapshot needs to be consulted, and real data needs to be transferred between the snapshot and the buffer cache used by the file system code being run in the simulation. This is usually the result of a directory lookup or inode update. Note that the file system only invokes this mechanism when the needed metadata is not already in the simulated buffer cache. As a result, most reads and writes only do simulated data movement.

5.6 Disk simulator

Since we were interested in exploring the interaction between file systems and future disk designs, we chose to construct a disk model that could easily be tuned to reflect design changes extrapolated from current performance characteristics. We took some pains to calibrate this model against current real disks, and in particular, to include the effects of caching in the disk drive, which prior work had shown to be an important part of getting good agreement between a model and reality.[8,16] As a result, we were able to achieve differences between the real disk and the modeled one, i.e., the model demerit figure, of only 5%.[8]

The first component of our disk model is a buffer cache, which is used to keep track of data that has been read, read-ahead, or written. Appropriate replacement policies allow us to alter the behavior of this cache for different experiments. In addition, we modeled the physical disk mechanism—the rotating media and the moving disk head and arm—and the DMA engine used to transfer data from the disk cache to and from the disk-to-host bus. By making each of these lightweight tasks, we were able to model the overlap between disk accesses and data transfers to and from the host system that occurs in real disks.

Our model uses replaceable modules for each of these components, so it is relatively easy

to make changes to explore different design choices. For example, enhancing the disk model to predict the effects of making its buffer cache non-volatile, to be described in the case study, took less than a day.

5.7 File system simulator

The other important component we were interested in modeling was the file system. One approach to modeling a file system is to construct a simplified simulation of the file system code. By making suitable assumptions, the resulting complexity and development time could be kept within reasonable bounds. However, this is a process fraught with difficulties, as our examination of the LFS development process suggests. Ensuring that the right simplifications are made is difficult, doubly so because the expectations of the experimenter can often bias the choices in favor of the set of assumptions made during the design of the file system that is being investigated.

We believe that it is possible to do better. In fact, our approach is to use the real file system code instead of an imperfect model of it, thereby eliminating any possibility of incorrect assumptions. To do this, we bring the file system out of its normal execution environment, which is the operating system kernel or a trusted address space. We do this by providing a set of scaffolding that looks—as far as the file system is concerned—just like the kernel environment in which it normally runs, down to and including the synchronization primitives the code is written to invoke. The entire ensemble runs as a regular, untrusted, user-level application.

File system designers have used this technique before to run kernel-level code at user-level to simplify debugging during program development, but not, to our knowledge, explicitly for performance studies.

Our approach allows the file system implementation and an understanding of its performance to develop together. For example, in addition to providing functionality stubs for incomplete

portions of the code, we can provide performance stubs as well. Running the new design in a user-space scaffolding, as in our approach, combines the advantages of easier development, faster turnaround time, and more flexible debugging with early access to performance data.

In the case study we conducted, we used the production HP-UX file system code as the file system simulator.

The other important component we were interested in modeling was the file system. One approach to modeling a file system is to construct a simplified simulation of the file system code. By making suitable assumptions, the resulting complexity and development time could be kept within reasonable bounds. However, this is a process fraught with difficulties, as our examination of the LFS development process suggests. Ensuring that the right simplifications are made is difficult, doubly so because the expectations of the experimenter can often bias the choices in favor of the set of assumptions made during the design of the file system that is being investigated.

We believe that it is possible to do better. In fact, our approach is to use the real file system code instead of an imperfect model of it, thereby eliminating any possibility of incorrect assumptions. To do this, we bring the file system out of its normal execution environment, which is the operating system kernel or a trusted address space. We do this by providing a set of scaffolding that looks—as far as the file system is concerned—just like the kernel environment in which it normally runs, down to and including the synchronization primitives the code is written to invoke. The entire ensemble runs as a regular, untrusted, user-level application.

File system designers have used this technique before to run kernel-level code at user-level to simplify debugging during program development, but not, to our knowledge, explicitly for performance studies.

Our approach allows the file system implementation and an understanding of its performance to develop together. For example, in addition to providing functionality stubs for incomplete

portions of the code, we can provide performance stubs as well. Running the new design in a user-space scaffolding, as in our approach, combines the advantages of easier development, faster turnaround time, and more flexible debugging with early access to performance data.

In the case study we conducted, we used the production HP-UX file system code as the file system simulator.

5.8 Implementation and validation

In our implementation, bootstrap generation is done as a three stage software pipeline. The first stage rolls a die multiple times to choose a set of processes that are to be included in the bootstrap. The second stage deletes the traces of the processes that are not part of the bootstrap. The final stage duplicates the traces of processes as necessary; new process identifiers and sequence numbers are created at this stage. Recall that creating a bootstrap involves selection with replacement. This process allows the individual elements to execute independently of one another.

Duplicated records have the same time-stamps as the original records they are derived from. This could lead to increased contention for file and disk resources. In our experiments, this has not been a significant issue for the average case behavior because of the filtering performed by the user-level file cache in the file system simulation.

For a given trace, we generate multiple bootstraps and run the simulator on each bootstrap,

and then aggregate the performance data that results across these runs. Bootstrapping theory tells us that bootstrap distribution of a particular statistic closely approximates the true, but not directly measurable, distribution of the statistic in the real population. Thus on the average, the performance of the simulator on the bootstraps will be similar to what would have been seen if it had been run on real traces, i.e., samples from the real population.

The entire process of generating bootstraps can be done on-the-fly. It is also repeatable, if
the same pseudo-random number generator is used for the selection process. This means that exactly the same bootstraps can be generated several times if so desired, e.g., for runs with different simulation parameters.
The main value of using bootstrapping in simulation studies is to extend existing trace data on-the-fly while retaining certain statistical guarantees. As long as the original trace data.

5.9 Analysis: detail and complexity versus efficiency

We argue here for the use of detailed models for file system and disk system components. Our contention is that such models lead to increased confidence and increased accuracy, without an excessive increase in execution time or complexity.
Consider first the degree of detail that is *desired* in modeling the system. Obviously, if the real system is available to test, it is usually best to measure that system, since this minimizes the uncertainty. One of the strengths of our approach is the use of real file system code as the file system simulator, which removes one major cause of uncertainty; another is the use of real traces rather than synthetic ones; a third is the calibration of the disk system against real disks.
Consider next the degree of detail that is *required* to model the system. Work in disk drive modeling has shown that detailed models are a necessity there: ignoring caching effects, which in turn depends on modeling rotation position in the disk, can result in mean simulated times as much as a factor of two larger than they should be.[8] So, sufficient detail is essential if useful results are to be acquired.
Of course, it is not always possible to determine which features of the model will prove to be the most important—indeed, these may change as a function of what is being modeled. For example, a file system that did not make rotational-position layout optimizations or use the disk's aggressive write caching would be much less sensitive to caching effects in the disk. Thus, we believe it prudent to err on the side of caution.
Finally, consider the *cost* of detailed models. We believe that the approach we advocate is not particularly costly: our disk simulator is able to process about 2000

requests per second on a 100 MHz PA-RISC processor; the file system code runs at full processor speed; and—just as in real life—the metadata snapshot information is frequently cached by the underlying real file system that the simulator is hosted on. The result is that the elapsed time for executing the simulations is much less than that required to execute the real system executing the traced workload. Furthermore, as processors speed up relative to I/O, this disparity in performance is likely to increase.

A significant benefit of our approach is *confidence* in the results. In the final stages of design, omitting a crucial detail may be potentially dangerous. Our approach makes it easy to construct a detailed model that avoids this pitfall.

We feel the accuracy and confidence offered by our approach far outweigh the small investment in time to build the scaffolding. This is a one-time cost that can be amortized over many studies. In our experience, the code to implement a detailed disk model and the bootstrap generator proved fairly straightforward. By simply dropping the real file system code into the simulator, our development time for this portion of our model was zero.

On the other hand, a potential drawback to our scheme is that it assumes the availability of the file system code. Sometimes this might not be case, e.g., when designing a new file system from scratch. However, even in these cases, many aspects of the our system, e.g., the disk model, workload characterization, and parts of the scaffolding, may be used independently.

5.10 MODELING THE DISK

Although there are some exceptions, much of the prior work on disk modeling has not accurately reflected the considerable concurrency that occurs in modern disk drives, nor the actual operational characteristics of the disk itself, including non-linear seek versus distance times, bus transfer effects, and caching. We endeavored to address these issues in our model. Its calibration has been described elsewhere; here, we concentrate on a description of the elements that go to make up the model.

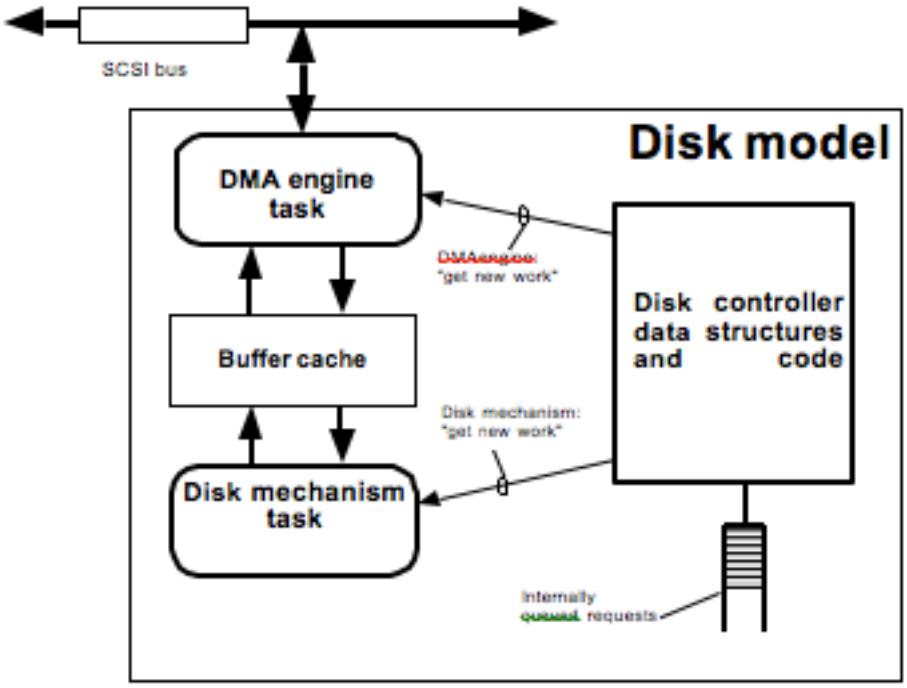


Figure 3: Major Componenets of the disk model.

Figure above shows the components of our model. The major components are as follows; in our implementation, each is a C++ object:

1. The *disk controller* is a data structure that binds together the other elements, and provides a placeholder for them. It also provides the management code for the disk's actions, and a number of parameters used to control modeling of aspects like controller overhead.
2. The *disk cache* represents the on-board cache memory in the disk. It can be managed as a simple speed-matching buffer, or segmented and used to cache data before or after it is explicitly referred to by the host. Here are two examples of the caching policies that

our model supports. If the disk is idle, and the last request was a read, the controller may choose to continue doing a speculative read-ahead into the cache in case the host is making sequential transfers. If the last request is a write, the controller may allow data transfer across the bus into the disk in parallel with the execution of the last request; this is known as immediate reporting, and allows efficient writes to consecutive disk addresses.

3. The *disk mechanism* task models the rotating media and the disk heads attached to a moving arm. In practice, most of the code is concerned with translating logical addresses into physical ones, taking into account details of the disk drive geometry such as zoning, which allows more sectors on the outer tracks than the inner ones spare sectors, and head- and track-sector skew, which minimize rotation delays on head and track-switches.
4. The *DMA engine* task models the transfer of data across the interface between the disk and the bus connecting it to the host system. The bus is acquired and released according to policies determined by the design of the disk controller, parameters that can be set by the host system, and the availability of data or space in the disk cache. This allows contention between multiple disks on the same bus to be modeled correctly.
5. The *request-scheduling policy* determines, in combination with the cache-management policies, which request will be executed next if the disk drive has been passed more than one. For example, this allows the command queueing of SCSI-2 to be modeled.

We found it convenient to have each task call into the disk controller code to request work for it to do, blocking if there was none. This allowed each task to be a simple get-work— execute-it loop, and let us concentrate the complexities of handling the interactions between the cache management and the request scheduling in one place.
While this model might appear complex, it is in fact quite easy to implement. Our scaf- folding provides lightweight threads, synchronization objects such as semaphores, and queue abstractions. The disk elements are implemented as independent threads that send messages to each other through queues and synchronize as needed using semaphores. The model has been parameterized for several different disks using a combination of manufacturer-supplied data and direct measurements. The simulation is tuned to minimize error in the transfer size range, typically 4–8 kbytes, commonly used by current file system designs. Calibration against real disk performance under a range of workloads yields excellent agreement, within 5%. The total code required to achieve this level of accuracy is modest—a little over 3000 lines of C++. The particular disk model that we describe here has been extensively used in other studies.

A separate paper[8] contains quantitative information of how different portions of the model contribute to its accuracy and how it compares with typical simple models. Undoubtedly, an accurate model like ours is more complicated than a simpler, less accurate, model. On the other hand, we can quantify and bound its deviations from the behavior of real disks, and we know that it does a good job of modeling components of disk behavior that are growing in importance as file system designs attempt to adapt to, and take advantage, of exactly these performance non-linearities.

5.11 MODELING THE FILE SYSTEM

Since we were particularly interested in exploring the effects of changing disk technology on file system behavior and performance, we developed techniques that allowed us to use the actual file system code rather than an imperfect abstract model of it. By comparison with an abstract model, our approach increased our confidence in the results, and also ensured that we did not have to continually adjust the parameters of the file system model as a result of different workloads or disk behaviors.

We found it straightforward to adapt the file system code running in the kernel to run as an untrusted user application within the simulator. The infrastructure requirements of a file system are typically straightforward: some multitasking, simple memory management, and access to physical devices and user memory space—usually through a very stylized, well-controlled interface. The multitasking support usually has to include some form of threads and a set of synchronization primitives. All these are relatively easy to emulate in a user-space scaffolding. For example, the device-driver routines can easily be provided by a set of procedures that invoke the interface provided by the disk simulator. Processes in the original system can be treated as independent threads each with per address space structures imitating those of the original system. Though we happened to use the HP-UX file system as a base for our case study, these techniques are applicable in exporting code from other systems to run at user level.

As a specific example, for the case study to be described later, the entire HP-UX file system,[14] which is derived from the 4.2BSD Fast File System,[15] was run at user level without modification. In this case study, almost all the code in the file system simulator

was taken from a copy of the HP-UX product source code. Additional code that was needed to make it execute correctly at user level was quite minimal—about 3000 lines of C. This represents code that is implemented once; the actual code for the various file systems under test runs unchanged. This represents a huge saving in work, because typical file system implementations are quite large. Most of the code we added is required to provide the right kernel-level abstractions and the correct device interface at user level and can be reused without any change to simulate other file systems.

To validate our file system simulator implementation against a real kernel, we compared the block requests issued by the real file system running inside the operating system kernel and the simulator. There were no significant differences between the two systems on a set of several different programs. This is not too surprising: we were executing the same code in both cases, but we found that it inspired our confidence in our results.

6. CODE

6.1 Main Functioality

```
package filesystem.core;

import java.io.*;
import java.util.Vector;

public class FileSystem
{
 protected static final String FILE_SYSTEM_PATH = "c:\\filesystem.fs";
 private RandomAccessFile raf;

 public static String ERROR = "";

 public FileSystem()
 {
 }

 protected RandomAccessFile OpenFile() throws Exception
 {
  if(raf != null)
  {
   throw new Exception("File is already Opened");
  }
  else
  {
   raf = new RandomAccessFile(FILE_SYSTEM_PATH, "rw");
   return raf;
  }
 }

 public void closeFile() throws Exception
```

```
{
 raf.close();
 raf = null;
}
public RandomAccessFile getRAF()
{
 return raf;
}

protected void writeRootNode(Node n, long offset) throws Exception
{
 raf.seek(offset);
 raf.write(n.getNodeNamebytes());
 raf.writeByte(n.getNodeType());
 raf.writeLong(n.getOffset());
 raf.writeLong(n.getNodeLink());
 raf.writeLong(n.getSizeOnDisk());
 raf.writeLong(n.getFreeSpace());
 raf.writeLong(n.getLastNodeWritten());
}

protected Node readRootNode(long offset) throws Exception
{
 raf.seek(offset);
 Node rootNode = new Node();
 raf.read(rootNode.nodeName);
 rootNode.setNodeType(raf.readByte());
 rootNode.setOffset(raf.readLong());
 rootNode.setNodeLink(raf.readLong());
 rootNode.setSizeOnDisk(raf.readLong());
 rootNode.setFreeSpace(raf.readLong());
 rootNode.setLastNodeWritten(raf.readLong());
 return rootNode;
}
```

```
protected Object[] readRoots() throws Exception
{
  Vector v = new Vector();
  Node n = new Node();
  n = readRootNode(0);
  while(n.getNodeLink() != 0)
  {
    v.add(n);
    n = readRootNode(n.getNodeLink());
  }
  if(n.getNodeName() != null || n.getNodeName().length() != 0)
    v.add(n);
  return v.toArray();
}

protected void writeDirectory(Node dNode, long offset)throws Exception
{
  raf.seek(offset);
  raf.write(dNode.getNodeNamebytes());
  raf.writeByte(dNode.getNodeType());
  raf.writeLong(dNode.getOffset());
  raf.writeLong(dNode.getNodeLink());
  raf.writeByte(dNode.getStatus());
  raf.writeLong(dNode.getChildNode());
}
protected byte getNodeType(long offset)throws Exception
{

  raf.seek(offset);
  raf.skipBytes(Node.MAX_NODE_LENGTH);
  byte b = raf.readByte();

  return b;
```

```
  }
//this method will either return directory or file depends upon when it finds
  protected Node readNode(long offset) throws Exception
  {

    Node n = new Node();
    if (getNodeType(offset)==Kernel.DIRECTORY)
    {
      n = readDirectory(offset);
    }
    else if(getNodeType(offset)==Kernel.FILE)
    {
      //read file code goes here
      n = readFile(offset);
    }

    return n;
  }
  //return true if there is no file or directory exists in the root
  protected boolean isRootEmpty(Node root) throws Exception
  {
      raf.seek(root.getOffset()+Kernel.ROOT_SIZE+1);
      raf.skipBytes(Node.MAX_NODE_LENGTH);
      byte b = raf.readByte();
      if(b > 0)
        return false;
      else
        return true;
  }
  protected boolean isFileSystemEmpty()
  {
    try
    {
      OpenFile();
```

```
    raf.seek(0);
    raf.skipBytes(Node.MAX_NODE_LENGTH);
    byte b = raf.readByte();
    closeFile();
    if(b <= 0)
      return true;
   }
   catch(Exception ex){
    try{closeFile();}catch(Exception e){return true;} return true;}
   return false;
}
protected Object[] readRoot(Node root)throws Exception
{
   if(!isRootEmpty(root))
   {
    Vector v = new Vector();
    long offset = root.getOffset() + Kernel.ROOT_SIZE + 1;
    Node tmp = readNode(offset);
    //set the parent node of this node
    tmp.setParentNode(root);
    tmp.setPreviousNode(root);
    tmp.setParentDirectory(root);
    //if(tmp.getStatus()!= Kernel.IS_DELETED)
      v.add(tmp);
    while(tmp.getNodeLink() >0)
    {
      //OpenFile();
      Node previousNode = tmp;
      tmp = readNode(tmp.getNodeLink());
      tmp.setPreviousNode(previousNode);
      tmp.setParentNode(root);
      tmp.setParentDirectory(root);
      //if(tmp.getStatus()!= Kernel.IS_DELETED)
        v.add(tmp);
```

```
      //closeFile();
    }
    return v.toArray();
  }
  else
    return null;
}
protected Node readDirectory(long offset) throws Exception
{
  raf.seek(offset);
  Node tmp = new Node();
  raf.read(tmp.nodeName);
  tmp.setNodeType(raf.readByte());
  tmp.setOffset(raf.readLong());
  tmp.setNodeLink(raf.readLong());
  tmp.setStatus(raf.readByte());
  tmp.setChildNode(raf.readLong());
  return tmp;
}
protected boolean deleteNode(Node n)
{
  try
  {
    n.setStatus(Kernel.IS_DELETED);
    this.writeDirectory(n,n.getOffset());

    //Node previousNode = n.getPreviousNode();
    //make connection b
    //previousNode.setNodeLink(n.getNodeLink());
  }
  catch(Exception ex)
  {
    System.out.println("Unable to delete file: " + ex.getMessage());
    ex.printStackTrace();
```

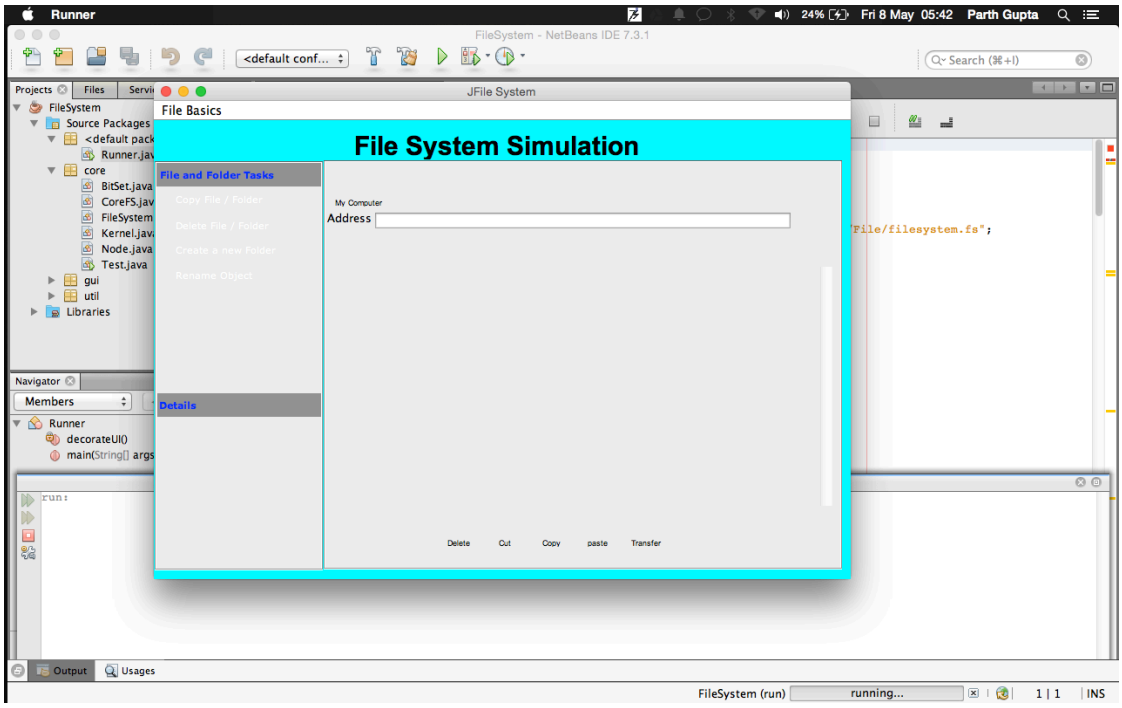
```
    return false;
   }
   return true;
 }
 //return the last node in the root
 protected Node getLastRootNode(Node root)
 {
   try
   {
     long offset = root.getOffset() + Kernel.ROOT_SIZE + 1;
     Node n = this.readNode(offset);
     if(n.getOffset() == 0)
       return null;
     while(n.getNodeLink() >0)
       n = this.readNode(n.getNodeLink());
     return n;
   }
   catch(Exception ex)
   {
     System.out.println("Unable to get last root directory"+ex.getMessage());
     return null;
   }
 }
 //write the file
 protected void writeFile(Node aFile)throws Exception
 {
   try
   {
     raf.seek(aFile.getOffset());
     raf.write(aFile.getNodeNamebytes());
     raf.writeByte(aFile.getNodeType());
     raf.writeByte(aFile.getStatus());
     raf.writeByte(aFile.getFileAttributes());
     raf.writeLong(aFile.getOffset());
```

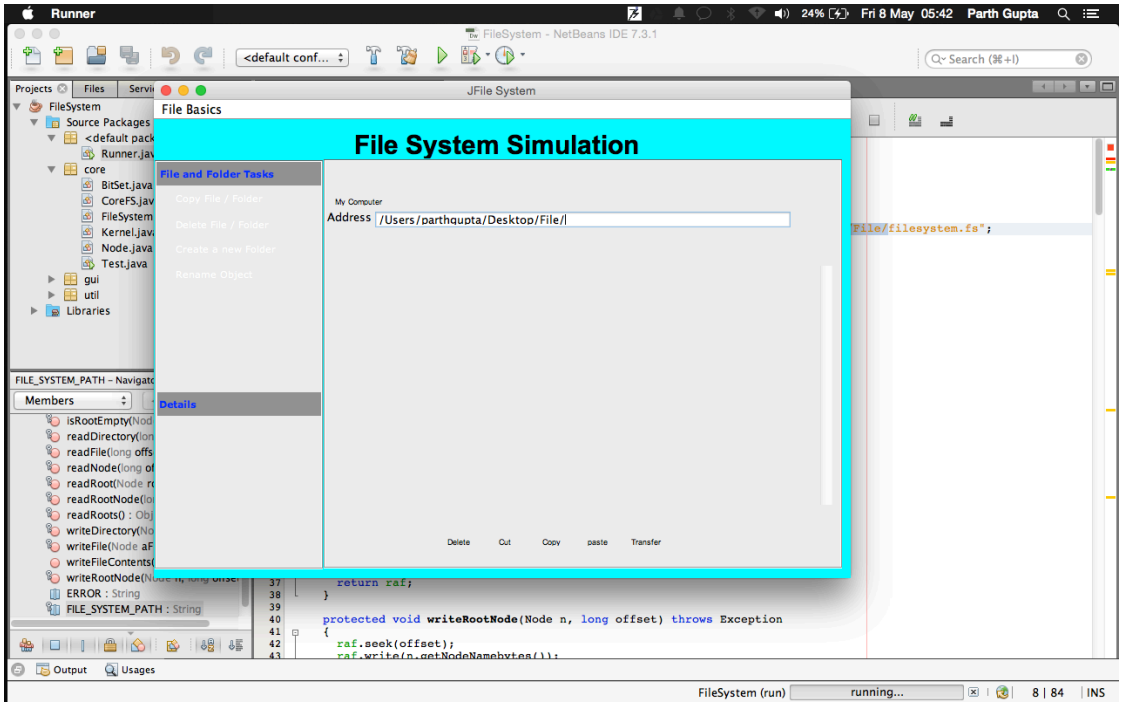
```
    raf.writeLong(aFile.getNodeLink());
    raf.writeLong(aFile.getSizeOnDisk());
    raf.writeLong(aFile.getLastModified());
   }
   catch(Exception ex)
   {
    throw new Exception(ex.getMessage(),ex);
   }
  }
  public void writeFileContents(byte b)throws Exception
  {
   try
   {
    raf.write(b);
   }catch(Exception ex)
   {
    throw new Exception ("Unable to write file contents: ",ex);
   }
  }
  protected Node readFile(long offset) throws Exception
  {
   raf.seek(offset);
   Node tmp = new Node();
   raf.read(tmp.nodeName);
   tmp.setNodeType(raf.readByte());
   tmp.setStatus(raf.readByte());
   tmp.setFileAttributes(raf.readByte());
   tmp.setOffset(raf.readLong());
   tmp.setNodeLink(raf.readLong());
   tmp.setSizeOnDisk(raf.readLong());
   tmp.setLastModified(raf.readLong());
   return tmp;
  }
}
```

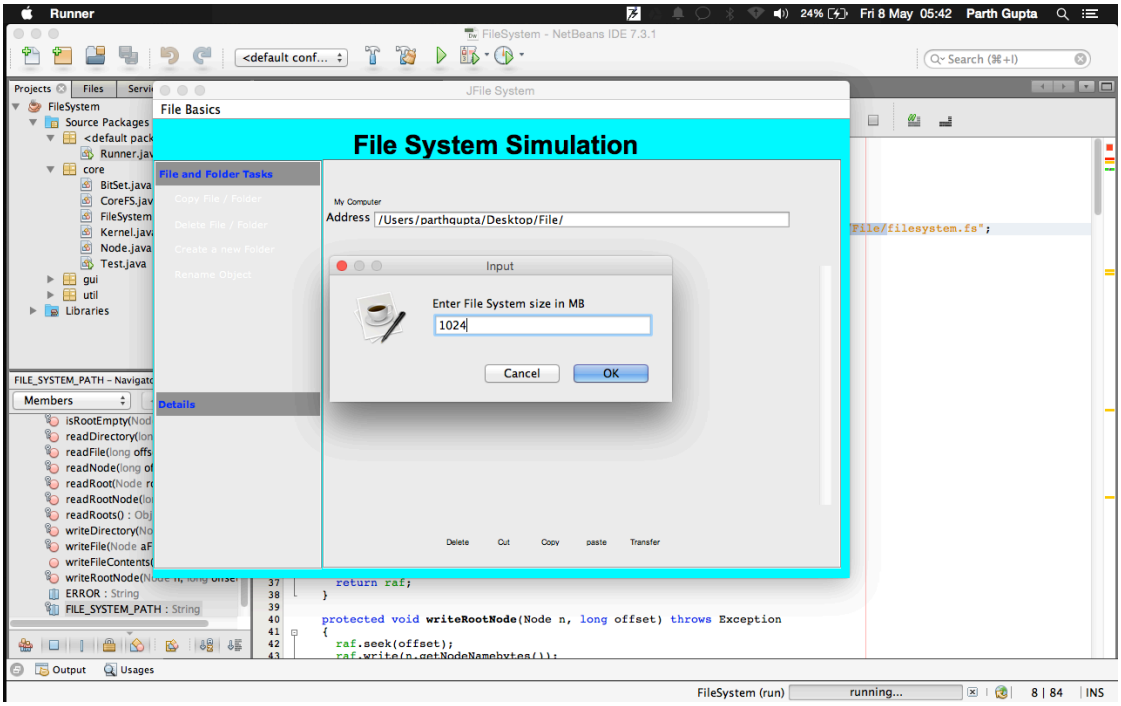
6.2 Kernal Functionality

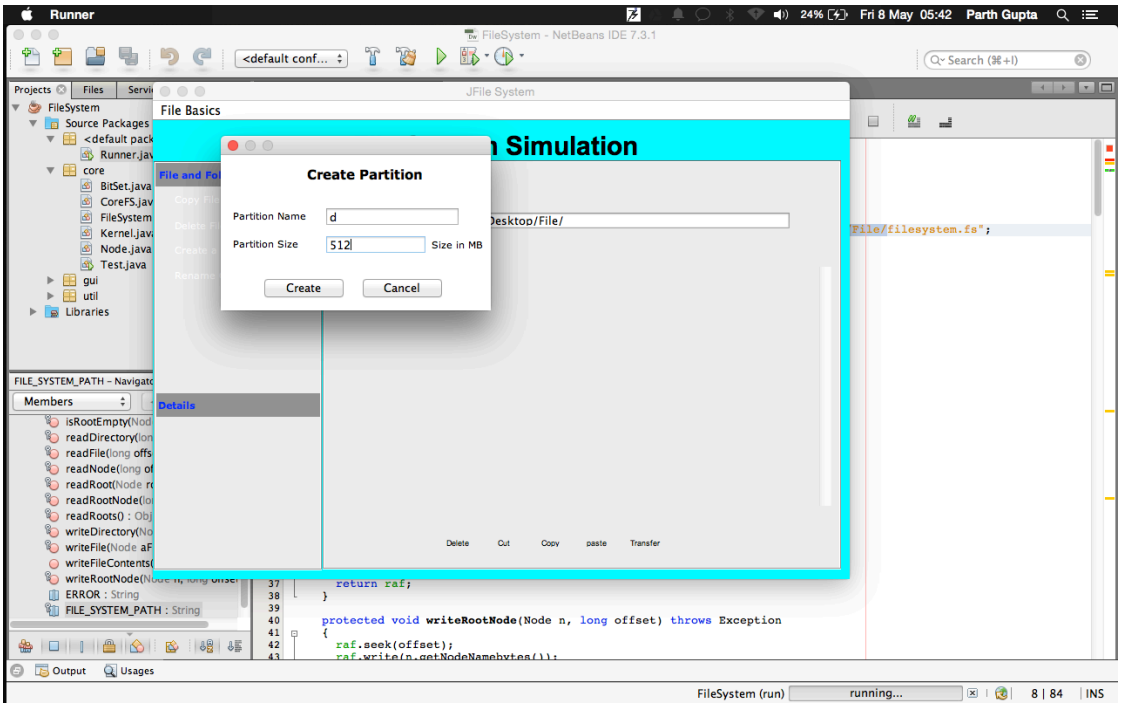
```
package filesystem.core;

public class Kernel
{
  public static final byte ROOT = 1;
  public static final byte DIRECTORY = 2;
  public static final byte FILE = 3;


  public static final int ROOT_SIZE = 49; //in bytes
  public static final int DIRECTORY_SIZE = 34; //in bytes
  public static final int FILE_SIZE = 35; //in bytes

  public static final byte IS_DELETED = 1;

  public Kernel()
  {
  }

}
```

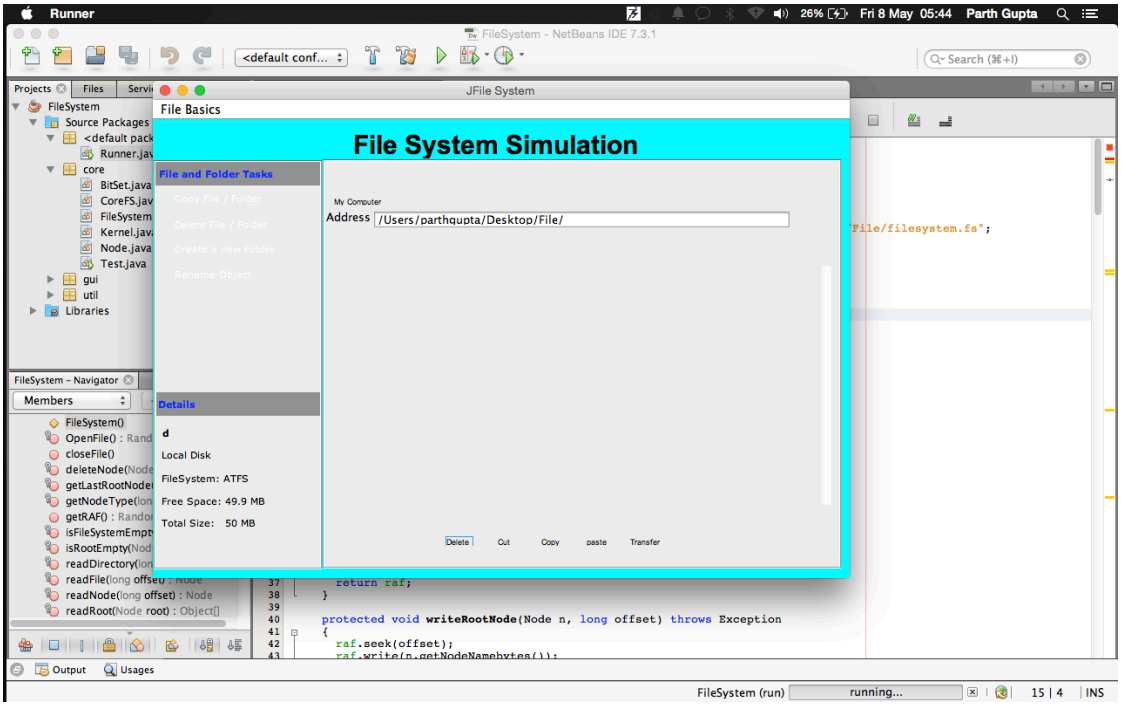
7. RUNNING PROGRAM SNIPPETS

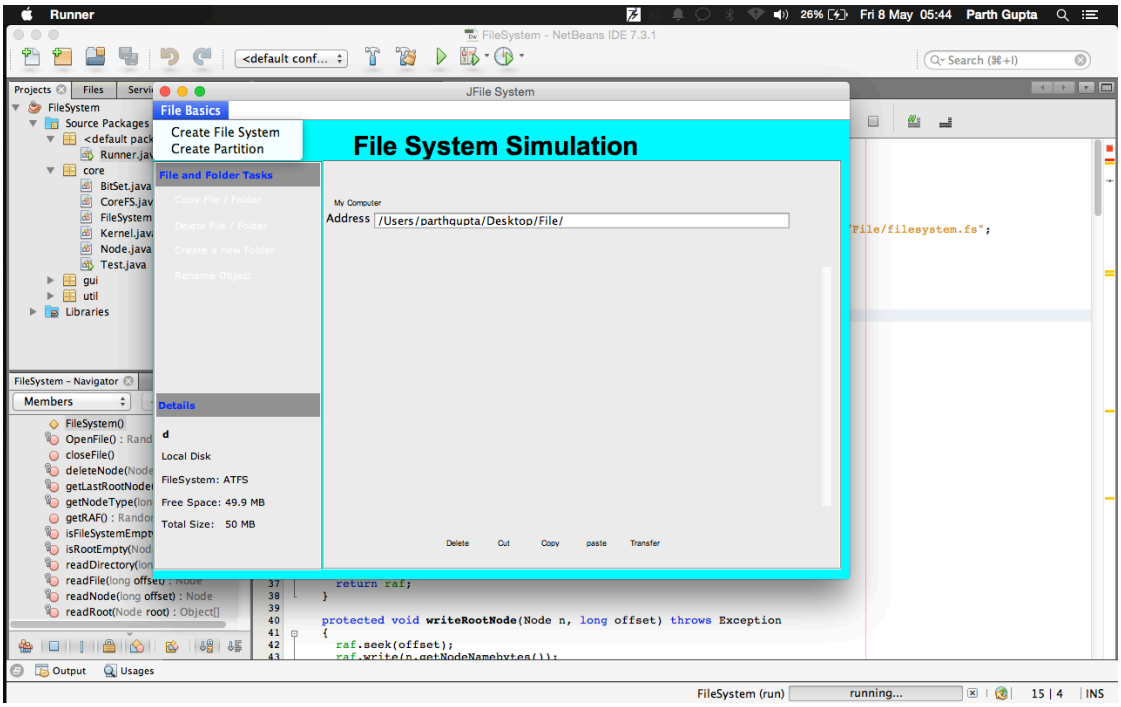





File Basics
File System Simulation
File and Folder Tasks
Address /Users/parthgupta/Desktop/File/
Input
Enter File System size in MB
1024
Cancel
OK
Details
Delete
Cut
Copy
paste
Transfer


File Basics
Create Partition
Partition Name
d
Partition Size
512
Size in MB
Create
Cancel
Details
Delete
Cut
Copy
paste
Transfer


JFile System
File Basics
File System Simulation
File and Folder Tasks
Address /Users/parthgupta/Desktop/File/
Details
d
Local Disk
FileSystem: ATFS
Free Space: 49.9 MB
Total Size: 50 MB
Delete
Cut
Copy
paste
Transfer


JFile System
File Basics
Create File System
Create Partition
File System Simulation
File and Folder Tasks
Address /Users/parthgupta/Desktop/File/
Details
d
Local Disk
FileSystem: ATFS
Free Space: 49.9 MB
Total Size: 50 MB
Delete
Cut
Copy
paste
Transfer

8. BIBLIOGRAPHY

1. Mendel Rosenblum and John K. Ousterhout, 'The design and implementation of a log-structured file system', *ACM Transactions on Computer Systems*, **10**, (1), 26–52, (February 1992).

2. David Patterson, Garth Gibson, and Randy Katz, 'A case for redundant arrays of inexpensive disks (RAID)', *ACM SIGMOD 88*, 109–116, (June 1988).

3. David J. DeWitt, Randy H. Katz, Frank Olken, L.D. Shapiro, Mike R. Stonebraker, and David Wood, 'Implementation techniques for main memory database systems', *Proceedings of SIGMOD 1984*, June 1984, pp. 1–8.

4. Robert B. Hagmann, 'A crash recovery scheme for a memory-resident database system', *IEEE Transactions on Computers*, **35**, (9), 839–843, (September 1986).

5. Mark Holland and Garth A. Gibson, 'Parity declustering for continuous operation in redundant disk arrays', *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992, pp. 23–35.