University of Kentucky
UKnowledge

University of Kentucky Master's Theses | Graduate School

2007

RLINKS: A MECHANISM FOR NAVIGATING TO RELATED FILES

Naveen Akarapu
University of Kentucky, naveenakarapu@yahoo.com

Right click to open a feedback form in a new tab to let us know how this document benefits you.

Recommended Citation

Akarapu, Naveen, "RLINKS: A MECHANISM FOR NAVIGATING TO RELATED FILES" (2007). *University of Kentucky Master's Theses*. 467.
https://uknowledge.uky.edu/gradschool_theses/467

This Thesis is brought to you for free and open access by the Graduate School at UKnowledge. It has been accepted for inclusion in University of Kentucky Master's Theses by an authorized administrator of UKnowledge. For more information, please contact UKnowledge@lsv.uky.edu.

ABSTRACT OF THESIS

RLINKS: A MECHANISM FOR NAVIGATING TO RELATED FILES

This thesis introduces *Relative links* or *rlinks*, which are directed labeled links from one file to another in a file system. Rlinks provide a clean way to build and share related-file information without creating additional files and directories. Rlinks form overlay graphs between files of a file system, thus providing useful alternate views of the file system. This thesis implements rlinks for the Linux kernel and modifies the storage structure of the Ext2 file system to store the rlinks.

KEYWORDS: Rlinks, Linux kernel, Ext2, related files, file system

Naveen Akarapu

08/03/2007

RLINKS: A MECHANISM FOR NAVIGATING TO RELATED FILES

By

Naveen Akarapu

Dr. Raphael A. Finkel

(Director of Thesis)

Dr. Raphael A. Finkel

(Director of Graduate Studies)

August 3, 2007

RULES FOR THE USE OF THESES

Unpublished theses submitted for the Master s degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgments.

Extensive copying or publication of the thesis in whole or in part also requires the consent of the Dean of the Graduate School of the University of Kentucky.

A library that borrows this thesis for use by its patrons is expected to secure the signature of each user.

Name Date

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

THESIS

Naveen Akarapu

The Graduate School

University of Kentucky

2007

RLINKS: A MECHANISM FOR NAVIGATING TO RELATED FILES

THESIS

A thesis submitted in partial fulfillment of the
requirements of the degree of Master of Science in the
College of Engineering at the University of Kentucky

By

Naveen Akarapu

Lexington, Kentucky

Director: Dr. Raphael A. Finkel, Professor of Computer Science

Lexington, Kentucky

2007

MASTER'S THESIS RELEASE

I authorize University of Kentucky Libraries to reproduce
this thesis in whole or in part for purposes of research.

Signed: Naveen Akarapu

Date: August 3, 2007

ACKNOWLEDGMENTS

I am greatly thankful to Dr. Raphael Finkel for his continuous support and guidance. Without his encouragement I would have abandoned the thesis. Working under his guidance brought the best out of me. Many ideas implemented in this thesis originated in my meetings with Dr. Finkel.

I thank my parents for their support and confidence in me throughout the process. I thank my two brothers for their constant encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

1. Introduction

This thesis tackles the problem of organizing related information on a filesystem. Specifically, by related information we mean related files. Being able to easily find related files helps users maintain their filesystem and enhances their experience accessing files. As discussed later in Section 3, current mechanisms like directories and symbolic links are ineffective for this purpose. This thesis introduces a new type of links called *Relative Links* or *rlinks*. Rlinks are directed labeled links from one file to another. By placing a file inside a directory that points to another file, symbolic links and hard links semantically create a link from that directory to the pointed file. Instead, rlinks create links between two files. This thesis implements rlinks for the Ext2 filesystem by changing Ext2's directory storage structure.

2. Related Work

Semantic File systems

In one of the earliest papers in this field, Gifford et al [1] define a *semantic file system* as "an information storage system that provides associative access to the system's contents by automatically extracting attributes from files with file type specific *transducers* [sometimes called *importers*]". Traditional directories are replaced by v*irtual directories,* whose names are interpreted as queries to a query engine. A semantic file system is typically an abstraction layer on top of a traditional filesystem like ext2 or reiserfs.

GLScube [2] is a recently developed semantic file system. GLScube collects rich meta data for files. Importers provided for each file type understand files and store relevant information about them in a database. In addition, users can assign tags to files and also create relations with other files. Users access data through *virtual collections,* whose contents are created dynamically through their associated queries. GLScube is implemented as a user-space file system that augments the features of the underlying file system.

The Be File System [3] provides extensive support for storing a file's attributes. Database functionalities like indexing and querying these attributes are also built into the kernel. The Be File System was designed specifically for Be OS.

Other semantic file systems are DBFS [7] and The Placeless documents [4].

Extended Attributes

Extended attributes were introduced in Linux kernel 2.6 to store meta-data to a file (or directory) in addition to its common attributes like permissions. Each extended attribute is a name-value pair associated with a file. The name and the value can be any string.

The most common application of Extended Attributes is to implement POSIX Access Control Lists (ACL). They can also be used for things like storing the character encoding of a file, song information of a music/mp3 file, or tags of a file. A search application, Beagle, uses extended attributes for indexing a file.

3. Problem Explanation

The goal of this thesis is to create a mechanism by which one can easily see and reach related information of a filesystem object (which on Unix systems is a file). Another goal is to build this feature into a filesystem and inside the kernel, both for learning and performance reasons.

Significance of accessible related information

Many successful websites on the Internet provide related information on most of their web pages. Successful social-networking sites like MySpace and Orkut provide links to 'Friends' (related people) in every profile. Almost every news website now assigns a portion of each of its article pages for links to related news. A game-report page of an NFL playoff game, for example, has links to game reports or live scorecards of other playoff games. Such information helps the user explore the area, in this case the NFL playoff games, better.

Many parallels can be drawn between a desktop and a website like that of the BBC. People looking for a particular news item have to traverse the hierarchical organization of information. For example, someone trying to know about a US Open Tennis match would have to traverse through Home Page → Sport → Tennis → US Open. This "US Open" page, which contains all the news (that is, links to news pages) about the tournament, is similar to a directory that contains files pertaining to a project. From this page, when a user opens the news page of one particular match he is interested in, probably because it features his favorite player, he finds other news about the match and the players involved in the "related news" section of the page.

Sometimes the "related news" section contains links to news that belong to a category different from that of the current page, like a link to a page reporting the day's weather from a match report page. Typically, websites organize information hierarchically. If a web page (like the weather page) has information that crosses domains, it first is placed in the domain it predominantly belongs to; then a link to the page is provided from the other domains related news section. This strategy is very similar to placing a file in a directory and creating a symbolic link to that file in a related directory.

Significance of finding related files on a filesystem

A person working on a filesystem object, be it a music file, a video file, a project document, or a program, would be greatly assisted by having related information that's easily accessible.

Just as related information is useful in web pages (or website objects), providing related information is also useful for filesystem objects. For example, installing software usually involves placing its components in different directories. The package file is downloaded into one place, the source code is extracted to another directory, and after compilation the programs are installed in (often copied to) another directory. When there is a need to maintain the software or uninstall it, it is difficult to learn the whereabouts of all its components. If one could instead know the install directory and the source code of the program he is interested in just by requesting *related files* of the program, the task would be a lot easier.

The ever increasing multimedia contents on a filesystem can be organized by storing the related multimedia associated with a multimedia object. For example, one could create a relation between an audio mp3 file, which is usually stored in the audio part of directory hierarchy, and its corresponding video file, which is usually stored among videos. Similarly, a relation could be created from the video file to a movie in which the artist acted. By following such relations, a user can reach all the information he is interested in.

Now that the significance of accessibility of related information in general and on computer systems is established, let's take a look at existing mechanisms that help us achieve that.

The present techniques in Unix-like systems for making related files easily accessible from each other are directories and links. All the files belonging to a project, for example, are placed in a directory (with possible sub-directories). This organization implies that each file in the directory is equally strongly related to every other file in the directory, which is not always the case, especially in a directory containing many files. Files within a directory tend to have subtle relationships between them, in addition to the general *directory* relation.

Furthermore, keeping track of related files is difficult when a file semantically belongs to more than one directory and its related files exist in multiple directories. One solution is placing symbolic links in all the directories containing files related to the file. In addition to increasing the complexity of the directory tree by adding to its entries, such links are tedious to create and manage. Studies [4] have shown that symbolic links are ineffective in these situations.

Some of the feature-rich semantic file systems mentioned above do have mechanisms to reach related files. But they come with a cost of performance. They are typically implemented in user space. A huge amount of information is collected for each file in a database management system (DBMS) and is maintained through daemons that keep track of all changes to the filesystem. This setup is often a significant overhead for the majority of simple tasks. Moreover, the goal of the semantic file systems is more extensive than that of this thesis. It includes changing the hierarchical file-access methodology to something more intuitive and easy. This thesis only augments the hierarchical file-access method with a new feature.

Extended attributes are built into the operating system and the filesystem, which makes accessing meta- data using extended attributes faster than that using the semantic file

systems. Using the attributes associated with a file, a search can retrieve files with similar attributes. A limitation of such an attribute-based search is that it returns files that are connected only indirectly and loosely: Two completely unrelated files can independently have the same tag.

Rlinks, which we introduce in this thesis to solve the problem, provide a way to store related-file information of a file. They are built into the kernel, which makes them very fast. They do not increase the complexity of a directory by adding more entries to it. Rlinks are directed links, which helps in defining finer relationships between files. Rlinks distribute the related-file information throughout the filesystem, instead of storing it in a centralized file or database. Hence in the event of an accidental deletion, only the related-file information of the deleted files is lost.

Because rlinks are directed labeled links from one file to another, they create a logical graph in which files of the filesystem are the vertices and rlinks are the arcs. This graph can be traversed by search and other applications.

Section 5 describes the semantics chosen for rlinks, Section 6 explains an implementation of rlinks and Section 7 contains some applications that use rlinks.

4. Background

Linux is a free and open-source operating system, originally developed by Linus Torvalds in 1991. It implements Unix's API but has completely independent source code and design. The Linux kernel version used for this project is 2.4.33.3.

The Linux kernel is licensed under GNU General Public License (GPL) version 2.0. Consequently, its source code can be freely downloaded and modified. The modified kernel can be distributed, but only under the same license. The 2.4 kernel aims to be compliant with IEEE POSIX standard API.

Most of the content in this section has been summarized from Bovet & Cesati [5] and Love [6].

4.1 System calls

System calls are the interface through which processes running in user space interact with the kernel.

System calls are software interrupts or *exceptions*. Every system call is assigned a number, which is passed to the interrupt handler when a system call is invoked. The interrupt handler (named `system_call()`) looks up the system-call dispatch table stored in the `sys_call_table` array, and calls the system-call service routine indexed by the system call number. The service routine of a system call is named by the system call name prefixed by `sys_`. This invocation chain is illustrated in Figure 4.1 (redrawn from [6]).

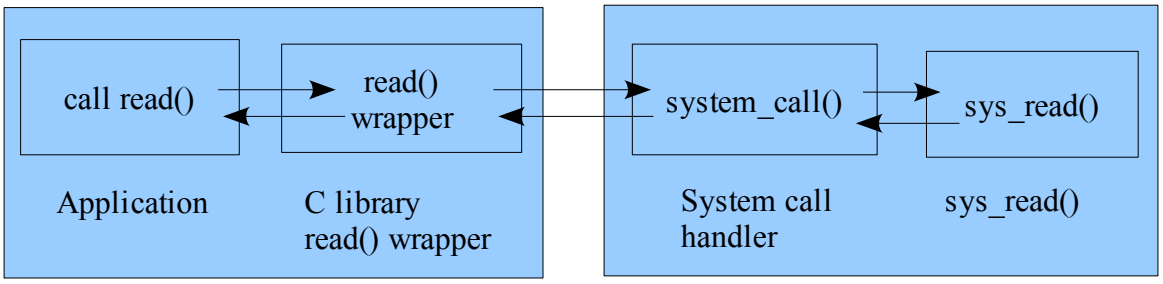

Figure 4.1: Invoking a system call.

4.2 The Virtual Filesystem

The Linux kernel has various subsystems like memory management, networking, device drivers, process management and process scheduling. The subsystem this thesis deals primarily with is the filesystem.

The Virtual Filesystem Switch (VFS) is the subsystem of the kernel that implements the filesystem-related interfaces provided to user-space programs. VFS allows multiple filesystems to coexist and interoperate. The VFS enables system calls like `open()`, `read()`

and `write()` to work regardless of the filesystem of the underlying physical medium. Such a generic interface for any type of filesystem is feasible because the kernel implements an abstraction layer around its low-level filesystem interface. VFS provides a common file model that is capable of representing any filesystem's general features and behavior.

For example, consider a user-space program that invokes
`write(f, &buf, len);`

This call writes `len` bytes pointed to by `&buf` into the current position in the file represented by the file descriptor `f`. This system call is first handled by a generic `sys_write()` system call service routine that determines the actual file writing method for the filesystem on which `f` resides. The generic write system call then invokes this method, which is part of the filesystem implementation, to write data into the media. Figure 4.2 (redrawn from Love [6]) below shows the control flow.

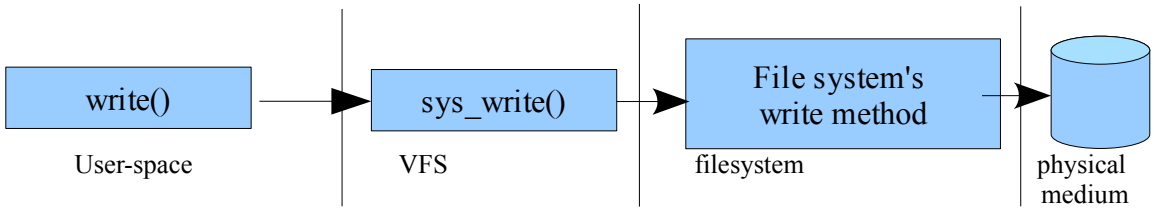

Figure 4.2: Control flow through VFS

VFS Objects

The VFS is object-oriented. A family of data structures represents the common file model. Similar to objects in an object-oriented language, the structures contain both data and pointers to filesystem-implemented functions that operate on the data. The rlink implementation involves manipulating many of these data structures.

The Superblock Object

The superblock object is implemented by each filesystem and is used to store information describing that specific filesystem. The object is represented by `struct super_block`.

The `superblock` structure contains a field `s_op` of type `struct super_operations` which holds a table of super block operations. Each item in the `s_op` structure is a pointer to a function that operates on a superblock object.

A complete listing of the table is provided in Appendix B.

The Inode Object

The inode object represents all the information needed by the kernel to manipulate a file or directory. The inode object is represented by `struct inode`. The complete definition of `struct inode` is provided in Appendix B.

The inode structure contains an inode operations field `i_op`, of type `struct inode_operations`, which holds a table of pointers to functions implemented by a filesystem. These functions are invoked by VFS on an inode. When a filesystem needs to perform an operation on an inode of its file, it follows the pointers from the file's inode object to the desired method. For example if a filesystem wants to truncate a file, it invokes the method as follows:

```
i->i_op->truncate(i)
```

where `i` is a pointer to an inode. In this case, the `truncate()` operation defined by the filesystem on which `i` exists is called on the given inode. The definition of the `inode_operations` structure is provided in Appendix B.

The dentry object

The `dentry` object represents a directory entry, a single component of a path. For example, in the path `/bin/vi, vi, bin` and `/` are all represented by dentries. Dentry objects are all components of a path, not differentiating between files and directories. Dentries help in resolving a path. Unlike inode and superblock objects, the dentry object does not correspond to any on-disk structure; VFS creates dentries on the fly. To speed up the lookup process, previouslycreated dentries are cached in the dentry cache. The dentry structure and its dentry operations table are given in Appendix B.

The file object

The file object represents an open file as associated with a process. A file object exists for each file opened by a process. The object is created in response to the `open()` system call and destroyed in response to the `close()` system call. The object stores the interaction between the process and a file. It stores information like access mode and current offset. Whereas a file has unique inode and dentry objects in the memory, it can have many file objects. The definitions of the file object and its associated operations table are provided in Appendix B.

The operations objects for these primary VFS objects are implemented as a structure of pointers to functions that operate on the parent object. For many methods, the objects can inherit a generic function when the basic functionality is sufficient. Otherwise, the specific instance of the particular filesystem fills in the pointers with its own filesystem-specific methods.

VFS Objects associated with filesystem

VFS keeps track of all filesystem types whose code is currently included in the kernel by performing filesystem type registration. Filesystems are registered during system initialization and also when a module implementing a filesystem is loaded. Each registered filesystem is represented as a `file_system_type` object (whose definition is provided in Appendix B). This structure provides information that helps VFS read the superblock of a filesystem into the VFS superblock object when an instance of the filesystem type is mounted. This information is subsequently used in creating other VFS objects like inode and file.

When a filesystem is mounted, a `vfsmount` structure is created. This structure represents the specific instance of a filesystem. This structure contains information about the mount point, such as its location, mount flags and the relationship between the filesystem and other mounted filesystems. The definition of `vfsmount` structure is provided in Appendix B.

4.3 Buffer cache

The smallest addressable unit on a block device is called a *sector*. The smallest addressable unit of the filesystem is called a ***block***. The block is a software abstraction of the filesystem; filesystems can only be accessed in multiples of blocks. Because the device's smallest addressable unit is a sector, the block size must be a multiple of the sector size. Furthermore, the kernel requires that a block be no larger than the memory's page size. Common block sizes are 512 bytes, 1 KB and 4 KB.

When a block is stored in memory, it is stored in a *buffer*. Each buffer is associated with exactly one block. In order to maintain control information about the data in buffer such as the device identifier and block number, the kernel associates each buffer with a descriptor, called the ***buffer head*** (`struct buffer_head`). The buffer head holds all the information the kernel needs to manipulate the buffer. The buffer head structure is defined in Appendix B.

4.4 Page I/O and block I/O

The rlink implementation involves both block I/O and page I/O. As explained later in section 6.2.2, block I/O operations are used to read and write rlink blocks to disk, whereas page I/O operations are used to read and write a file's directory-entry page.

Block I/O operations

Block I/O operations transfer a single block of data in to a single buffer. The buffer is associated with a specific block, which is identified by the major and minor numbers of the block device and by the logical block number.

Block I/O operations are used when kernel reads or writes single blocks in a filesystem, like a block containing an inode or a superblock.

For reasons of efficiency, buffers are stored in special pages called *buffer pages* instead of as independent memory objects. All the buffers in a buffer page have the same size, and they must be adjacent disk blocks.

Block devices transfer information one block at a time, while process address spaces (i.e., memory regions allocated to the process) are defined as sets of pages. This mismatch is hidden by page I/O operations.

Page I/O operations

Page I/O operations transfer as many blocks of data as needed to fill a single page frame (the exact number depends both on the block size and on the page frame size). Each page frame contains data belonging to a file. Because the data is not necessarily in adjacent disk blocks, it is identified by the file's inode and by an offset within the file. Page I/O operations are used mainly for reading and writing files.

Both kinds of I/O operations rely on the same functions to access a block device (because the requests need to go through the block device driver), but the kernel uses different algorithms and buffering techniques with them.

4.5 Kernel Synchronization

Data shared by multiple processes needs to be protected against race conditions. Many shared kernel data structures like dentry objects, mount structures and memory pages are accessed during the rlink operations. The kernel provides the following primitive operations to synchronize access to such data:

Atomic operations on `atomic_t` variables increment, decrement, and test the 24-bit `atomic_t` type counter variables atomically.

Atomic bit operations test and change state of a bit atomically.

Spin locks synchronize access to a data structure in a multiprocessor environment. Control paths waiting on a spin lock 'spin', repeatedly executing a tight instruction loop.

Semaphores allow the waiting processes to sleep, unlike spin locks.

The Big Kernel Lock (BKL) allows only process to be executing in the system. This lock is specially designed for multiprocessor environments.

4.6 The Ext2 Filesystem

Ext2 is Linux's native and the most used filesystem type. This thesis adds rlink functionality to Ext2 and calls the new filesystem type *RelExt2*. In order to understand how rlinks are implemented in RelExt2, it is necessary to understand Ext2's storage structure.

To differentiate the filesystem type from a filesystem of that type, in this section I use *Ext2* to mean the filesystem type and *partition* to mean an instance of a filesystem type.

Ext2 Disk Data Structures

Except the first block, which is a boot block used for system startup, the rest of the Ext2 partition is split into *block groups*, each of which has a layout shown in Figure 4.3 (taken from [5], Copyright © 2002 O'Reilly Media, Inc. All rights reserved. Used with permission).

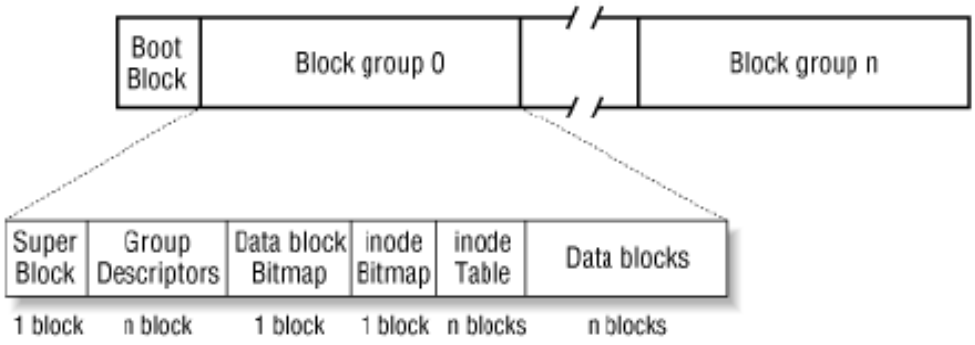

Figure 4.3: Layout of an Ext2 partition

Each block in a block group contains one of the following pieces of information:

- A copy of the filesystem's superblock
- A copy of all block group descriptors
- A data-block bitmap
- A group of inodes
- An inode bitmap
- A chunk of data that belongs to a file, that is, a data block

The superblock contains information pertaining to the entire partition, like the number of inodes and the number of blocks. The complete definition of the Ext2 superblock structure is given in Appendix B. Similarly, a group descriptor stores information pertaining to the block group like the number of free blocks in the group and the number of free inodes in the group. A data-block bitmap is a sequence of bits, where a value of 1 indicates that the corresponding data block is in use, and a value of 0 indicates that the corresponding data block is free.

The inode region that follows the data-block bitmap consists of several blocks, each containing a fixed number of inodes. Each inode structure is of 128 bytes. An inode-bitmap block stores a bitmap indicating the free and allocated inodes in the group.

The rest of the blocks in the block group are data blocks storing data belonging to files.

Ext2 Directories

Ext2's directories are special files that store filenames together with the corresponding inode numbers. Such files contain structures of type `ext2_dir_entry_2.` The fields of the structure are shown below.

```
struct ext2_dir_entry_2 {
        __u32   inode;                  /* Inode number */
        __u16   rec_len;                /* Directory entry length */
        __u8    name_len;               /* Name length */
        __u8    file_type;              /* File type */
        char    name[EXT2_NAME_LEN];    /* File name */
};
```

The `inode` field stores the inode number of the file specified by the directory entry. The `name` field, which stores the final component of a file's path, is a variable length array of up to `EXT2_NAME_LEN` characters. The `name_len` field stores the length of the string stored in `name`. The `file_type` field stores an integer value that indicates the file type of the entry.12

The `rec_len` field stores the length of the directory entry. The value in this field can be added to the address of the directory entry to obtain the starting address of the next directory entry. An example of a Ext2 directory is shown in Figure 4.4 (taken from [5], Copyright © 2002 O'Reilly Media, Inc. All rights reserved. Used with permission).

Figure 4.4: An example Ext2 directory

5. Relative Links (rlinks)

Introduction

This thesis introduces rlinks into the Linux kernel. An rlink is a labeled directed link from one file to another. The head of an rlink is called a *from-file* and the tail is called a *to-file*. An rlink is illustrated in figure 5.1

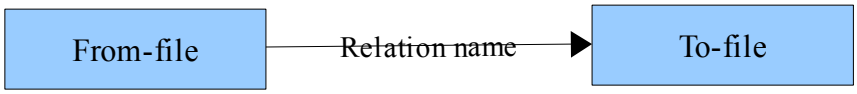

Figure 5.1: An rlink

Rlinks are based on pathnames. In Linux, a file can be addressed in two ways – itsinode number and its (path)name. Symbolic links address their targets by name. Hard links, on the other hand, address their targets by inode. Similarly, rlinks address their targets (to-files) by name. Although rlinks' addressing mechanism is similar to that of symbolic links, rlinks require that their targets exist during creation, whereas symbolic links don't.

Rlinks create a directed labeled graph in which files are the vertices and rlinks are the arcs. Because each from-file maintains a list of its to-files, in terms of graph theory, each vertex stores the list of vertices that are adjacent to it in the graph. Thus the graph is stored in the form of an adjacency list that is distributed among its vertices. Graph algorithms can be applied on such a graph, as illustrated in section 7.1, where we describe a program that computes the shortest path in the graph.

The rest of this section explains the functionality of rlinks with the help of three primary rlink operations. Section 6.1 explains the system calls underlying these operations, which applications use for manipulating rlinks.

An rlink can be created with a command such as the following:

```
rlink <from-file> <to-file> <relation-name>
```

This command creates an rlink of name `relation-name` with `from-file` as its head and `to-file` as its tail.

The semantics for creating rlinks are

- The from-file and to-file must belong to the same file system (i.e., partition). However, this limitation can be overcome by creating a symbolic link in the desired filesystem pointing to the file stored on a different filesystem and creating rlinks using this symbolic-link file.

- Multiple rlinks with the same relation-name between the same two files are not allowed. However, one can create multiple rlinks between two files using a different relation-name. Such links make the graph a multigraph.

- When symbolic-link files are given as input files, the `rlink` command creates rlinks between the symbolic-link files rather than the targets of the symbolic-link files.

Rlinks of a file can be read from command line with a command such as the following

```
readrlink filename [relation-name]
```

`readrlink` prints rlinks in which `filename` is the from-file. If a relation-name is specified, rlinks of only that relation type are returned; otherwise, all relation types along with their to-files are returned.

An example of readrlink output is shown below:

```
readrlink casino.avi
  scorsese:
        /mnt/gsfs/movies/goodfellas.avi
        /mnt/gsfs/movies/raging_bull.avi
        /mnt/gsfs/movies/taxi_driver.avi
        /mnt/gsfs/movies/Gangs_of_New_York.avi
  deniro:
        /mnt/gsfs/movies/taxi_driver.avi
        /mnt/gsfs/movies/goodfellas.avi
        /mnt/gsfs/movies/raging_bull.avi
  joepesci:
        /mnt/gsfs/movies/raging_bull.avi
```

In the above readrlink output, `scorsese, deniro and joepesci` are relation-names, and the files listed below each of these names are the to-files of `casino.avi`.

Rlinks of a file can be deleted with a command such as the following:

```
unrlink from-file [to-file [relation-name]]
```

unrlink deletes an rlink from `from-file` to `to-file` of type `relation-name` when all three values are given. Both `to-file` and `relation-name` are optional. When `relation-name` is not specified, unrlink deletes all relations from `from-file` to `to-file`. When neither `relation-name` nor `to-file` are specified, it deletes all rlinks of `from-file`.

The semantics of unrlink are

- If a to-file does not exist (that is, the rlink is “broken”), the user must have execute permission on all the existing ancestors of the to-file in the to-file's path. For example, if a to-file is missing but its parent directory exists, the user may delete the rlink only if the user has execute permission on all the components of to-file's absolute path until the to-file's parent directory. These semantics ensure that the to-file would have been accessible to the user if it existed. However, the path specified for the missing to-file must not contain any broken symbolic links other than the to-file.

Characteristics of rlinks

- Because rlinks join files on same partition, the links persist even when the partition is unmounted and mounted on another system, or when the partition is mounted at a different mount point on the same system.

- Rlink semantics lie between that of symbolic links and hard links on a strictness spectrum. Symbolic links do not care if the file they are pointing to exists or if the user creating them has any permission on the target file. In contrast, the semantics of rlinks require that to-file be not only reachable but also readable to the user. Hard links require that the target file be reachable and be on the same partition *and the same mount point* as the new file, whereas rlinks are more lenient in that they require the to-file's path be from any of multiple simultaneous mounts of the partition. Rlinks do not restrict deletion of the to-file, whereas the presence of at least one hard link prevents a file from being deleted.

- The number of rlinks from the same from-file depends on the individual file system implementation. RelExt2 allocates one block of the filesystem to store rlink information belonging to a from-file. The size of the block is the only constraint on the number of rlinks that can be created for a file. Typical block sizes are 1KB, 2KB and 4KB.

6. Implementation of Rlinks

6.1 System calls

This thesis introduces three system calls to enable rlink operations:: `rlink()`, `readrlink()` and `unrlink()`.

Rlink:

```
#include <rlink.h>
int rlink(const char *from_file, const char *to_file, const char
*relation);
```

The `rlink()` system call creates an rlink from `from_file` to `to_file` of `relation-name` relation. `From_file` and `to_file` are respectively null-terminated pathnames of the from-file and the to-file of the rlink. Relation is a null-terminated unicode string that defines the relationship. The maximum length of a relation string is 255.

The process must have read permission on both the files. In addition, a process must have write permission on the `from_file`'s parent directory. Appendix A presents a detailed discussion of permission issues.

On success, `rlink()` returns zero. On failure, `rlink()` returns -1 and sets `errno` appropriately.

Errors:

- ENOENT -- Either from_file or to_file does not exist.
- EXDEV – From_file and to_file do not belong to the same filesystem.
- EACCES – The process does not have the necessary permissions.
- ENOSYS – The system call is not implemented by the filesystem to which the from_file belongs.
- EEXIST – The rlink already exists.
- ENOSPC – There is inadequate space on the filesystem for this link.

Readrlink:

```
      #include "rlink.h"
      int readrlink(const char *from_file, const char *rel, char *buf,
size_t *bufsize, char *mntdir, size_t *mntdir_size)
```

From_file is the file whose rlink information is being read. Rel is a null-terminated relation-name string. Buf is the buffer used to store output rlink information. Bufsize indicates the size of input buffer, buf, and on return from the call indicates the size of the data in buf.

On return from the system call, mntdir contains the mount directory of the from_file's filesystem. mntdir_size holds the size of the buffer mntdir; on return from the call it holds the size of the pathname in mntdir. This pathname is necessary because the to-file paths returned in buf are filesystem-absolute paths with respect to the from-file's filesystem. If mntdir is NULL or the value of integer at mntdir_size is zero, the mount directory is not computed; the contents of mntdir and mntdir_size remain unchanged.

If the parameter rel is NULL, then readrlink() returns the files corresponding to all the relations of from_file; otherwise, it only returns the files corresponding to the relation type mentioned in rel.

The process must have read and execute permission on the from_file's parent directory.

On successful return, buf contains one record for each relation-type returned. Each record is of type struct rlink_blk_entry, which is defined as follows:

```
struct rlink_blk_entry {
      unsigned short reclen;   //length in bytes of the relation entry
      char relstr[0]; // a string containing both relation-name and the
set of to-files
}
```

The field relstr contains the address of the null-terminated relation name string. After the relation name, the rest of the record contains a list of to-files delimited by null. Reclen contains the length of the entire record, including its own length (two bytes usually). When multiple relations are returned by readrlink(), reclen gives the offset of the next relation record from the beginning of current record.

When no rlinks matching the input criteria exist, readrlink sets `bufsize` zero, places a null string in `buf`, and returns zero (success). If the input `bufsize` is smaller than the size of the output rlink data, `readrlink()` fills the buffer `buf` with `bufsize` bytes and returns zero.

On success, `readrlink()` returns zero. On failure, `readrlink()` returns `-1` and sets `errno` appropriately.

Errors:

- `ENOENT` – `from_file` does not exist.
- `EACCES` – The process does not have the necessary access permissions.
- `ENOSYS` – The system call is not implemented by the filesystem to which the `from_file` belongs.
- `ENOMEM` – There is not enough memory to perform this operation.
- `ENAMETOOLONG` – The length of the buffer given to store the mount directory's pathname (`mntdir_size`) is too short.

Unrlink:

```
      #include "rlink.h"
      int unrlink(const char *from_file, const char *to_file, const char
*relation);
```

The system call `unrlink()` deletes an rlink of `relation` type from `from_file` to `to_file`. Both `from_file` and `to_file` are null-terminated strings indicating paths to the files. `Relation` is a null-terminated string of maximum length 255.

If no relation-name is specified, that is, when `relation` is `NULL`, `unrlink()` deletes all rlinks from `from_file` to `to_file`. Similarly, if `to_file` is `NULL`, it deletes the relation along with all its to-files. In this case, when the `to_file` is `NULL`, it is not necessary to have the required permissions on the to-files being deleted. If both `to_file` and `relation` are `NULL`, it deletes the entire rlink information of `from_file`.

On success , `unrlink() returns` zero. On failure, `unrlink()` returns -1 and sets `errno` appropriately.

To perform `unrlink()`, the process must have write and execute permission on the `from_file`'s parent directory, read permission on the `from_file`, and read permission on `to_file`. These permission rules are identical to those of the `rlink()` system call.

Errors:

- `ENOENT` – `from_file` does not exist; if `to_file` and/or `relation` is not null, no rlink defined for the given `to_file` and/or `relation` exists.
- `EXDEV` – `From_file` and `to_file` (or its deepest existing ancestor) do not belong to the same filesystem.
- `EACCES` – The process does not have necessary permissions.
- `ENOSYS` – The system call is not implemented by the filesystem to which `from_file` belongs.

Appendix B presents a detailed discussion of the permissions required to successfully perform these operations.

6.2 Implementation design

I implemented rlinks in a three-layer framework. The first layer is the *VFS layer,* to which the control arrives immediately after an application invokes an rlink-related system call. This layer performs tasks like checking the validity of system-call parameters and permissions of files. For the rest of the tasks, the VFS layer invokes the methods of the second layer, *the RelExt2 layer*. RelExt2 is Ext2 with the additional rlink functionality. This layer understands RelExt2 filesystem's data structures. Therefore, it performs tasks that manipulate these data structures, like allocating a disk block (to store rlink information) and retrieving data from such a block. Although the RelExt2 layer understands the filesystem's data structures, it does not understand the storage format of data inside an rlink block. A third layer, *the rlink-block layer*, manipulates the data inside an rlink block. The following sections explain the functionalities of the three layers in greater detail.

This layered framework provides flexibility and modularity to the software. The first pair of layered modules – VFS and RelExt2 – is already present in the design of Linux kernel. This modularization helps Linux access filesystems of many different types, as discussed in Section 4.2. The second pair of modules – RelExt2 and rlink-block – detaches rlink block's storage format from the functionality of RelExt2. Owing to this modularization, the RelExt2 code remains the same even if the rlink-block format changes. Conversely, any other filesystem type can use the rlink-block layer implementation to implement rlinks if it uses the same interface to the layer. This modularization also aids in development and maintenance of the kernel because one can modify and test each of the modules independently.

6.2.1 VFS Layer

As mentioned in section 4.2, the VFS layer handles filesystem-related system calls. This project adds three service routines in the VFS layer, `sys_rlink()`, `sys_readrlink()` and `sys_unrlink()`, to handle the system calls `rlink`, `readrlink` and `unrlink` respectively.

The three VFS-layer service routines check for the existence of from-file and for appropriate permissions on the from-file and its parent directory. The service routines also translate input paths to the VFS objects required to invoke respective inode operations. `sys_rlink()` and `sys_unrlink()`, which receive the to-file path, also check if both the input files belong to the same filesystem. After performing all the tasks at this layer, the routines invoke the filesystem's methods to accomplish filesystem-specific tasks.

The interface to filesystem's methods is provided through inode operations of the filesystem's inodes. Three new inode operations are added to VFS's inode operations table, as shown below:

```
struct inode_operations {
       ...
       int (*rlink) (struct inode *parent_dir, struct dentry *d_from, struct
nameidata *nd_tofile, const char *relation_name);
       int (*readrlink) (struct inode *parent_dir, struct dentry *d_from, const
char *relation_name, char *relbuf, size_t *bufsize);
       int (*unrlink) (struct inode *parent_dir, struct dentry *d_from, struct
nameidata *nd_tofile, const char *relation_name);
};
```

To enable rlinks, a filesystem must define the methods corresponding to these three inode operations and connect them to the VFS inode operations table for files belonging to that filesystem. The connection occurs when the kernel creates a VFS inode for a file of the filesystem; it places the addresses of the corresponding filesystem's methods in the inode operations table of the inodes. If a filesystem does not define (and connect) methods for these inode operations, system call service routines return with error code `-ENOSYS` ("`Function not implemented`"). Section 6.3 details the functionality of each of the three service routines.

6.2.2 RelExt2 Layer

The RelExt2 filesystem is based on the Ext2 filesystem. In order to store rlinks, RelExt2 has a slightly different storage structure from Ext2. . I chose Ext2 as an example of a filesystem for implementing rlinks because it is Linux's native and the most frequently used filesystem.

Section 4.6 shows Ext2's directory entry structure, `ext2_dir_entry_2`. RelExt2 has a slightly different directory entry structure:

```
struct gsfs_dir_entry_2 {
        __u32   inode;          /* Inode number */
        __u16   rec_len;        /* Directory entry length */
        __u8    name_len;       /* Name length */
        __u8    file_type;
        __u32   rlink_blk;      /* Number of block storing rlink info*/
        char    name[EXT2_NAME_LEN];    /* File name */
};
```

An additional field, `rlink_blk`, appears between `file_type` and `name`. The field `rlink_blk` stores the logical block number of the block containing rlink information of the from-file to which the directory entry belongs.

According to this storage structure, a from-file's rlink information is stored in its directory entry (which is always present in the parent directory's file). Consequently, if a from-file has more than one directory entry (which happens if a file has hard links), each of those entries has its own rlink block.

The `rlink()` system call verifies the value in `rlink_blk` field of from-file's directory entry. A value of zero means the from-file does not have an rlink block (RelExt2 initializes `rlink_blk` to zero whenever it creates a new directory entry); RelExt2 allocates a new disk

block in proximity to the directory's data blocks and writes the logical block number of that block in the `rlink_blk` field. If the value is not zero, it reads the block corresponding to the value to memory and manipulates it there. When reading or deleting rlink(s) of a from-file, RelExt2 uses the value in the `rlink_blk` field in the from-file's directory entry to read its rlink block from the disk.

RelExt2 frees an allocated rlink block only when the from-file is deleted. Renaming a file, which creates a new directory entry for the file, retains the file's `rlink_blk` value .

RelExt2 translates the to-file's path to its *filesystem-absolute path*. A **filesystem-absolute path** of a file is the path of the file with respect to the filesystem's root directory (instead of the system's root directory). This filesystem-absolute path represents the to-file inside an rlink block. Storing filesystem-absolute paths makes rlinks mount-point independent. The to-file names of rlinks remain valid even if the filesystem is mounted at a different mount point or on a different system, because RelExt2 can always derive the complete path by concatenating the filesystem-absolute path with the latest mount-directory path.

RelExt2 adds three new methods to Ext2: `gsfs_rlink()`, `gsfs_readrlink()` and `gsfs_unrlink()`.

```
int gsfs_rlink(struct inode *dir, struct dentry *d_fromfile,
                        struct nameidata *nd_tofile, const char *rel_name);
int gsfs_readrlink(struct inode *dir, struct dentry *d_fromfile,
                    const char *rel, char *buf, size_t *size)
int gsfs_unrlink(struct inode *dir, struct dentry *d_fromfile,
                        struct nameidata *nd_tofile, const char *rel_name)
```

When a RelExt2 filesystem is mounted, the kernel assigns the addresses of these three methods to the `rlink()`, `readrlink()` and `unrlink()` inode operations of VFS inode objects respectively of files belonging to the RelExt2 filesystem. The first parameter of all the three methods is the inode of the from-file's parent directory. Therefore, these methods are defined only for VFS's directory inode objects. Section 6.2.4 details these methods.

RelExt2 layer does not know the format of data storage inside an rlink block. It uses the interface provided by the rlink-block layer to manipulate the data inside an rlink block. Section 6.2.3 explains the rlink-block layer.

Reading and writing rlink blocks

The RelExt2 layer performs block I/O (see section 4.4) operations when reading or writing rlink blocks. The function `ext2_alloc_block()` allocates a new block when RelExt2 creates rlinks for the first time for a from-file. This function returns the logical block number of a free block. The function `sb_getblk()` creates a buffer and the associated buffer head for the new block. The function `sb_bread()` reads a block containing rlink data to a buffer.

If any change occurs to data in the buffer (which happens during `rlink` and `unrlink`) that needs to be written to the disk, RelExt2 marks the buffer as dirty. It then adds the buffer to the list of dirty data buffers belonging to the from-file's parent directory's inode.

Synchronization

Rlink block synchronization

Before RelExt2 accesses the data contained in an rlink block's buffer it must obtain a lock on the buffer. It does so by invoking `lock_buffer()`, passing as a parameter the address of buffer's associated buffer head. `Lock_buffer()` uses atomic bit operations to set the `BH_Lock` flag in the buffer head's `b_state` field. The kernel places the processes waiting on this lock in the `b_wait` queue in the buffer head structure and puts them to sleep. Upon completion of a read or write operation, the process that acquired the lock releases the lock on the buffer by invoking `unlock_buffer()`, passing to it the address of the buffer head of the buffer.

This synchronization is necessary to prevent multiple processes from simultaneously executing the critical regions. The critical regions for rlink operations are the rlink-block layer functions. The resource shared by the processes is the buffer representing an rlink block. Synchronization ensures that no two processes simultaneously execute an rlink-block layer function on the same buffer.

Rlink block number synchronization

Another data structure requiring synchronization is the directory file's page that contains the from-file's directory entry; multiple processes could simultaneously try to access the `rlink_blk` field of the entry. The value in this field is changes only once in the lifetime of the entry – when a file's first rlink is created.

To illustrate race conditions occurring on `rlink_blk` field, consider two processes simultaneously attempting to create the first rlink of a file. Without any synchronization, they both read the `rlink_blk` value in the file's directory entry as zero. Then they independently allocate blocks and write each block number in this field. Only one of the two block numbers is written. The data written in the other block is lost and the block itself is left dangling, because its block number is not accessible anywhere to free it.

To avoid this race condition, before reading the `rlink_blk` field of a file's directory entry is read, RelExt2 must acquire an exclusive lock on the page containing the entry. If the value in `rlink_blk` is non-zero, meaning an rlink block already exists, the kernel releases the lock on the page and proceeds with other operations on the block. Otherwise, if the value is zero, the process releases the lock only after assigning a new value to the `rlink_blk` field. With this synchronization in place, two (or more) processes simultaneously trying to create the first rlink attempt to acquire a lock on the page;only one of them succeeds, finds that `rlink_blk` value is zero, proceeds to allocate an rlink block and writes the block number in the field. It then releases the lock. Then the second process reads the `rlink_blk` field and finds it to be non-zero. It proceeds to read the block and perform operations on it. RelExt2 layer synchronizes the access to data inside the block by a buffer lock.

6.2.3 Rlink block layer

An rlink block is a RelExt2 filesystem block allocated to store a from-file's rlink information. The rlink block layer handles all manipulation of data inside an rlink block. This layer provides an interface to the RelExt2 layer consisting of the following four functions.

```
int reset_rlink_blk(char *blkbuf, unsigned short blksize)
int insert_rlink(char *blkbuf, unsigned short bufsize, const char *relname,
char *tofile)
int read_rlink_blk(char *blkbuf, const char *rel, char *buf, int bufsize)
int delete_rlink_blk(char *blkbuf, char *tofile, char* relname)
```

All the three functions accept as parameters the address of the beginning of buffer representing an rlink block,its size, and task-specific parameters. Because these functions always manipulate a buffer whose address and size are given as input parameters, the rlink-block layer is independent of varying disk-block sizes.

Structure of an rlink block

An rlink block has a header and a list of relation entries. The header of a rlink block is of type `struct rlink_blk_header`, which is defined as follows:

```
struct rlink_blk_header {
      __u16 free_space; //offset to beginning of free-space area
      __u16 reserved[2];
}
```

The field `free_space` stores the offset inside an rlink block where the free space starts. Four bytes following this field are reserved for future use.

The rlink block header is followed by a set of relation entries. Each relation entry consists of a relation-name followed by a set of to-files that are linked by that relation-name to the from-file. The structure of a relation entry is defined as follows:

```
struct rlink_blk_entry {
      __u16 reclen;   //length of relation entry
      char relstr[0]; // a string containing both relation-name and
            // the set of to-files
}
```

The field `relstr` stores a null-terminated relation-name followed by a set of to-file names, each of which is delimited by a null character. The field `reclen` stores the length of the entire relation entry, which includes two bytes of `reclen`, the length of relation-name and that of all the to-files. Every relation entry starts at a even address. If the value of reclen is odd, the relation entry is null-padded so that the next relation entry starts at a even address. Reading the two-byte integer field `reclen` from a even address is more efficient than reading it from an odd address. As a result, an odd value in the `reclen` field indicates that the last to-file of the relation entry has an additional null character. The storage structure of relation entries inside a rlink block is illustrated in Figure 6.1.

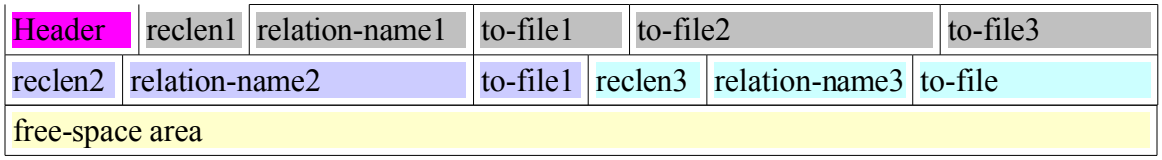

Figure 6.1: Layout of an rlink block

The storage format of rlink block data is designed to pack as much information as possible because there is only one block to store all the rlink data of a from-file. This format conserves space by placing a relation-name and all its to-files together. If the to-files were instead scattered at different places in the block, an integer value of two bytes would have been required to point to each to-file.

Maintaining this storage format requires lot of shuffling of data during rlink insertion and deletion operations. But because a block is read at once into memory, only one I/O is required for any operation. Due to the speed in processor and memory technologies, the cost of operations on a memory buffer is negligible.

Insert, delete, resize relation entries

When the rlink layer creates a new rlink for a from-file, the relation-name of the rlink might already exist in the rlink block. If so, the rlink layer appends the new to-file at the end of the relation-name's entry (and updates its `reclen` field)Otherwise the rlink layer creates a new relation entry for the relation-name. In both cases, the relation entry is placed at the beginning of the free space area inside the block. When modifying an existing relation entry, the rlink layer removes the relation entry from its place by moving entries following it to the begininning of the relation entry, overwriting it. It then places the relation entry at the end of used area in the block. However, if the relation entry is newly created it is simply placed at the end of used area in the block.

For example, in Figure 6.1, the rlink layer adds to-file2 to relation entry of relation-name2 by moving relation-name3 entry after relation-name1 entry. It then inserts the new relation-name2 entry after the relation-name3 entry.

The rlink-block layer deletes a to-file in a relation entry by moving all the rlink block's data following the to-file's name to its beginning, overwriting the to-file's name. It then updates the `Reclen` of the corresponding relation entry and the value in `free_space` header field. If the to-file to be deleted is the only to-file of the relation, it deletes the entire relation entry by moving the data following the relation entry to the beginning of the relation entry, overwriting it. It then updates the free space value accordingly. For example, to delete relation-name2 entry in Figure 6.1, the rlink-block layer moves the relation-name3 entry after the relation-name1 entry and then sets the `free_space` field in the header appropriately.

Section 6.3 has detailed description of the sequence of steps performed by rlink block functions in accomplishing these tasks.

6.3 System call algorithms

6.3.1 The `rlink()` system call

VFS Layer

The `rlink()` system call is serviced by `sys_rlink()` function, which receives as parameters the path of the from-file, the path of the to-file, and the relation name. The function returns zero on success and an appropriate negative error code on failure. `sys_rlink()` performs the following operations:

1. Invokes `__user_walk()` passing as parameters, the from-file path, `LOOKUP_PARENT` lookup flag, and the address of a local `nameidata` structure. This function performs a lookup for from-file's parent directory and returns the results in the `nameidata` parameter (`nd_dir`).
2. Invokes `__user_walk()` again passing the same parameters as above except the lookup flag is `LOOKUP_POSITIVE`. This call returns with `nameidata` structure for from-file.
3. Invokes `__user_walk()` as in step 2, this time passing to-file path as the parameter.
4. Using `nameidata` structures of from-file (`nd_from`) and to-file (`nd_tofile`) checks if they belong to same file system. It compares addresses of their super blocks to do the check: `nd_from.dentry->d_sb == nd_tofile.dentry->d_sb`. If they are not equal `sys_rlink()` returns with the error code `-EXDEV`.
5. Invokes `permission()` passing as parameter the inode of the from-file's parent directory, and setting `MAY_WRITE` and `MAY_EXEC` bits in its flags parameter. This function checks the inode's `i_mode` field with the userid of the current process (`current->fsuid`) to check if the process has write and execute privileges on the from-file's parent directory. If the check fails, with `permission()` returning a non-zero value, sys_rlink returns with the error code `-EPERM`.
6. Invokes `permission()` once each for from-file and to-file passing as parameters their inodes and the flag parameter with `MAY_READ` flag set.
7. If the `rlink()` inode operation is defined for the from-file's parent directory's inode, invokes it passing as parameters the parent directory's inode, dentry of the from-file, nameidata structure of the to-file and the relation name. This inode operation reads contents of from-file's parent directory and modifies it to create the rlink.
8. Invokes `path_release()` once for each of the nameidata structures obtained above – `nd_dir`, `nd_from` and `nd_tofile` – to decrement the usage counter for respective dentry and vfsmount structures.

RelExt2 layer

`sys_rlink()` invokes `gsfs_rlink()` when it invokes the `rlink()` inode operation of a directory file belonging to the RelExt2 filesystem. This function receives as parameters the inode of from-file's parent directory, dentry of from-file, nameidata structure of to-file, and a relation name. This function creates an rlink from from-file to to-file with relation name. On success it returns zero, whereas on failure it returns an appropriate negative error code. It performs the following tasks:

1. Invokes `get_path()` to obtain file-system-absolute path of to-file. The parameters it passes are the nameidata structure of to-file, pointer to the beginning of a free page, and size of the page. `get_path()` returns an address which marks beginning of a null-terminated pathname. `get_path()` performs the following tasks:
   a) Invokes `dget()` to increment its usage counter of to-file's dentry.
   b) Invokes `mntget()` to increment usage counter of to-file's vfsmount structure.
   c) Invokes `dget()` passing the dentry of root directory of to-file's file system. The dentry is obtained from `mnt->mnt_root` field of input nameidata structure.
   d) Acquires a spin lock on dentry cache using the `dcache_lock` global spinlock variable.
   e) Invokes `__d_path()` passing as parameters the dentry objects obtained in steps a and c, vfsmount structure from b, the input page buffer parameter and the buffer's size. `__d_path` uses a dentry's `d_parent` field to traverse from to-file's dentry to its root dentry, while collecting intermediate path components on its way using dentry structure's `d_name` field. The path returned by `__d_path()` is terminated by null and stored at the end of `__d_path`'s input buffer. The address of the start of the path is then returned on success, null on failure.
   f) Releases the spin lock acquired in step d.
   g) Invokes `dput()` twice, passing the dentry objects obtained from steps a and c each time, to decrement their usage counters.
   h) Invokes `mntput()` to decrement usage counter of vfsmount structure obtained in step b.
   i) returns the value of returned from `__d_path()`.
2. The path obtained from `get_path()` has a leading slash;but because the path is always absolute, the leading slash is redundant unless it is the only component. Therefore, except when to-file's path is root of the file system, `gsfs_rlink()` removes the leading slash by incrementing path's starting address.
3. Invokes `ext2_find_entry()`, passing as parameters the inode of the directory, the dentry of from-file and the address of the pointer variable to a page structure. `ext2_find_dentry()` populates the pointer variable with the address of the page in directory file that contains the from-file's entry; it returns the address of the directory entry (of type `gsfs_dir_entry_2)` within the page.
4. Invokes `lock_page()` to acquire a lock on the page obtained in previous step (3).
5. Checks if an rlink block already exists by checking the value of the `rlink_blk` field in from-file's directory entry. If this value is not zero, which means an rlink block already exists, it goes to step 6. On the other hand if the value is zero meaning an rlink block does not exist, it performs the following tasks.

a) Invokes `ext2_alloc_block()` passing as parameters the directory inode, an integer value goal and address of an integer variable to store error values. Goal is a value Ext2 uses to allocate blocks as close as possible to the value. The value assigned to goal here the block number of first buffer page of the directory page obtained in step 3. On success, `ext2_alloc_block` returns a non-zero number of a free block; on failure, it returns a zero.
b) Assigns the disk block number obtained above to `rlink_blk` field of from-file's directory entry.
c) Invokes `ext2_commit_chunk()` to write the directory page containing from-file's directory entry to disk.
d) Invokes `sb_getblk()`, passing as parameters the super block of the directory and the newly obtained block number, to allocate a buffer page for the block. `sb_getblk()` creates a buffer head for the buffer page and returns its address.
e) Locks the buffer page obtained in previous step (d) by invoking `lock_buffer()`, passing to it the buffer head.
f) Invokes `unlock_page()` to unlock the directory page locked in step 4.
g) Initializes the new block by invoking the `reset_rlink_blk()` rlink-block layer function, passing as parameters the address of buffer page obtained in step e and the size of the page.
h) Unlocks the buffer page locked in step e.
i) Control goes to step 8.

6. Invokes `unlock_page` to unlock the directory page locked in step 4.
7. Invokes `sb_bread()`, passing as parameters the directory's inode and the rlink block number contained in the from-file's directory entry, to read the rlink block from the disk into a buffer page. `sb_bread()` returns the buffer head of this block's buffer.
8. Locks the buffer (obtained either in step 7 or 5d) using the `lock_buffer()` function.
9. Invokes the `insert_rlink()` rlink-layer function passing as parameters the buffer page obtained previously, the size of the buffer page, the relation name and the to-file's path obtained in step 2. `Insert_rlink()` updates the buffer page with the new rlink information. It returns zero on success and an appropriate negative error code on failure.
10. Unlocks the buffer page locked in step 7
11. Marks the buffer page as dirty using the `mark_buffer_dirty()` function.
12. Inserts the buffer page in the directory inode's list of dirty data buffers using `buffer_insert_inode_data_queue()`.
13. Releases the buffer page using `__brelse()`.
14. Decrements the usage counter of directory's page (obtained in step 3) using `ext2_put_page()`.

Rlink block layer

The RelExt2 layer function `gsfs_rlink()` invokes the rlink-block layer functions `reset_rlink_blk()` and `insert_rlink()`.

`reset_rlink_blk()` receives as parameters a character buffer and size of that buffer. It assigns the `free_space` field of input buffer's header the length of the header indicating that everything after the header is free for use.

`Insert_rlink()` receives as parameters the address of a character buffer representing an rlink block, the size of that buffer, the relation-name string, and the path of the to-file. It returns zero on success and a negative error code on failure. It performs the following actions:

1. Checks if an entry for the relation-name already exists in the block buffer. If the relation-name's entry does not exist, it performs actions in step 2; otherwise it performs actions in step 3.
2. Creates a new record for the relation by performing the following tasks:
   a) Allocates a `rlink_blk_entry` type buffer to temporarily store the new record using `kmalloc()`.
   b) Estimates the length of the new record. If the block buffer does not have enough free space to store the record, function returns the error code `-ENOSPC`.
   c) Copies relation-name into the `relstr` field of the record (terminated by null).
   d) Copies to-file path following the relation-name string (terminated by null).
   e) Stores the length of the record in its `reclen` field. If the length is an odd number, increments it by one and appends an extra null character at the end of the record.
3. Creates a new record by updating the old one and updates the rlink block as follows:
   a) Checks if the to-file already exists. If it does, the function returns with error code `-EEXIST`.
   b) Estimates the additional storage required to add the new to-file path to the existing relation record. If there is not enough free space, the function returns with error code `-ENOSPC`.
   c) Allocates an `rlink_blk_entry` type buffer to temporarily store the new record.
   d) Copies the current relation entry from the block buffer in to a temporary buffer.
   e) Appends the new to-file at the end of the record in the temporary buffer.
   f) Updates the `reclen` field of the record in the temporary buffer. If the value is an odd number, increments it by one and appends an extra null character at the end of record.
   g) Calls `shuffle_blk()` to move the contents in the block buffer following the relation entry to the beginning of the relation entry, effectively overwriting it. `shuffle_blk()` also updates the `free_space` block header field.
4. Writes the new record obtained from either step 2 or step 3 into the block buffer at the beginning of its free space.
5. Updates `free_space` field of block header to include the newly appended record.
6. Frees the temporary buffer allocated in step 2a or 3c by invoking `kfree()`.

6.3.2 The `readrlink()` system call

VFS Layer

The `readrlink()` system call is serviced by the `sys_readrlink()` function, which receives as parameters the from-file name, a relation-name string, a buffer to hold output relation entries, the size of the buffer available, a buffer to hold the mount directory path and the size of this buffer. The function returns zero on success and an appropriate negative value on error. The function performs the following operations:

1. Invokes `__user_walk()` passing as parameters the from-file path, the `LOOKUP_PARENT` lookup flag, and the address of a local `nameidata` structure. This function performs a lookup for from-file's parent directory and returns the results in the `nameidata` parameter (`nd_dir`).
2. Invokes `__user_walk()` again passing the same parameters as above except the lookup flag is `LOOKUP_POSITIVE`. This call returns with `nameidata` structure for from-file (`nd_from`).
3. Checks whether the current process has necessary privileges by invoking `permission()`, passing as parameters the inode of the from-file's parent directory and the flags parameter with `MAY_WRITE` and `MAY_EXEC` bits set. This function checks the inode's `i_mode` field with the userid of the current process (`current->fsuid`) to check if the process has write and execute privileges on the from-file's parent directory. If the check fails, with `permission()` returning a non-zero value, `sys_readrlink()` returns with the error code `-EACCES`.
4. If the `readrlink()` inode operation is defined for the from-file's parent directory's inode, invokes it, passing as parameters the parent directory's inode, dentry of the from-file, the relation-name, the relation-entry buffer and the variable containing its size. This inode operation reads the contents of the from-file's parent directory and fills the relation-entry buffer and its size variable with relation entries and their total size respectively. The next subsection details all the operations done by this function in the RelExt2 filesystem. The BKL is acquired just before invoking the `readrlink()` inode operation and released immediately after the function returns.
5. If the mount directory buffer is not null and the corresponding buffer size is not zero, invokes `get_mnt_path()` to obtain the mount directory of from-file's filesystem. `get_mnt_path()` receives as parameters the nameidata structure of the from-file, a buffer to hold the output path and the size of this buffer. `get_mnt_path()` performs the following tasks to get the path:
   a) Obtains the dentry and vfsmount structures of the mount point of from-file's filesystem through `mnt->mnt_mountpoint` and `mnt->mnt_parent` fields of the input nameidata structure respectively.
   b) Obtains the vfsmount structure of the root filesystem by looping through the `mnt_parent` field of the vfsmount structures starting from the one obtained in step a, until the value in the `mnt_parent` field of a vfsmount structure is the same as the address of the structure. A spin lock is obtained on the dentry cache (using the `dcache_lock` variable) just before this operation.
   c) Obtains the dentry of the system's root by accessing `mnt_root` field of the vfsmount structure obtained in step 5b.

d) Invokes `__d_path()`, passing as parameters the dentry and vfsmount structures of the mount point (obtained in step a), the root directory's dentry (step c), the root filesystem's vfsmount structure (step b) and the buffer and buffer's size passed to `get_mnt_path()` as parameters. `__d_path()` returns the path of the mount point by returning an address inside the input buffer where the null-terminated path is located.
e) Returns the return value of `__d_path()`. Before returning, releases the spin lock acquired in step b and decrements all the counters on the dentry and vfsmount structures sent to `__d_path()`, which were incremented before invoking `__d_path()`.

6. Copies the mount directory obtained in step 5 to the user space buffer given as input parameter and assigns the length of the path to the buffer size parameter. If the size is too small, returns with the error code `-ENAMETOOLONG` after performing step 7.
7. Invokes `path_release()` once for each of the nameidata structures obtained above — `nd_dir`, `nd_from` — to decrement the usage counter for the respective dentry and vfsmount structures.

RelExt2 Layer

The RelExt2 layer function `gsfs_readrlink()` handles the call to `readrlink()` inode operation when the inode belongs to a RelExt2 filesystem file. The function accepts as parameters the inode of from-file's parent directory, the dentry of from-file, the relation-name, relation-entry buffer and the size of the relation-entry buffer. On successful return, the relation-entry buffer contains the requested relation entries, and the size variable contains the length of the content in the buffer. The `gsfs_readrlink()` function returns zero on success and an appropriate error code on failure. The function performs the following tasks:

1. Obtains the from-file's directory entry (of type `struct gsfs_dir_entry_2`) by invoking `ext2_find_entry()`
2. If the directory entry's `rlink_blk` field is zero, which means no rlinks were ever created for the from-file, sets the size parameter to zero and assigns null to the first character in the buffer (indicating a zero-length string). The function returns zero.
3. If the `rlink_blk` field is not zero, invokes `sb_bread()`, passing `rlink_blk` value as one of the parameters, to read the corresponding block in to a buffer. `sb_bread()` returns the address of the buffer head of the buffer.
4. Invokes `read_rlink_blk()`, an rlink-block layer function, to read the buffer obtained in step 3 to retrieve the requested relation entries. The parameters passed to this function are the address of the buffer (value of `b_data` field of buffer head obtained above), the size of the buffer, the relation-name whose entries we are trying to retrieve and a buffer to hold the resultant entries. On success, the function returns the size of the contents in the buffer; on failure, it returns a negative value indicating an error code.

   Before invoking `read_rlink_blk()`, locks the buffer using the `lock_buffer()` function passing the buffer head as its parameter; releases the lock immediately after the function returns.

5. If the return value from step 4 is negative, sets the relation-entry buffer size to zero and returns that negative value.
6. If the return value from step 4 is positive or zero, sets the relation-entry buffer size value to that value and copies the relation entries to user space.
7. Releases the buffer head of the buffer by invoking `__brelse()` and decrements the usage counter of the page that contained the from-file's directory entry.

Rlink-block layer

The rlink-block layer function `read_rlink_blk()` is invoked by `gsfs_readrlink()` (step 4 above) to read the contents of the buffer corresponding to a rlink block to retrieve the requested relation entries. The parameters passed to the function are a character buffer representing an rlink block, the relation-name whose to-files are requested, a buffer to hold the output relation entries, and the size of this buffer. `read_rlink_blk()` performs the following tasks:

1. If the input relation-name parameter is not null,
   a) Invokes `rel_name_exists()` to check if an entry for the relation-name exists in the block. If an entry exists, `rel_name_exists()` returns the address inside the block where the relation-name's entry begins. The function starts from the relation entry following the block's header and iterates through the relation entries in the block using each entry's `reclen` field. During each iteration it checks the `relstr` field of an entry for a match with the input relation-name; when a match occurs, it returns the address of the entry; otherwise, it returns NULL.
   b) If `rel_name_exists()` returns NULL, sets the first character of the relation-entry buffer to null and returns zero; otherwise, copies the relation entry into the buffer using the return value and returns the length of the relation entry.
2. If the input relation-name is null, iterates through the relation entries, copying each of them into the input buffer. Each relation entry is of type `struct rlink_blk_entry` (see section 6.2.3 for its definition). It copies the consecutive relation entries into the buffer in the same format as they are stored in the block (see section 6.2.3), except without padding at the end of relation entries. The function then returns the combined size of all the relation entries.

6.3.3 The `unrlink()` system call

VFS layer

The `unrlink()` system call is serviced by the VFS layer function `sys_unrlink()`, which receives as parameters the path of the from-file, the path of the to-file and a relation-name string. The function returns zero on success and an appropriate negative error code on failure. `sys_unrlink()` performs the following tasks:

1. If the relation-name parameter is not `NULL`, invokes `copy_from_user()` to copy the relation-name string from user space into a kernel-space variable.
2. Invokes `__user_walk()` passing as parameters the from-file path, `LOOKUP_PARENT` lookup flag, and the address of a local nameidata structure. This function performs a lookup for from-file's parent directory and returns the results in the nameidata parameter.
3. Invokes `__user_walk()` again passing the same parameters as above, except the lookup flag is `LOOKUP_POSITIVE`. This call fills the nameidata structure's fields with from-file's information.
4. Checks if the process that invoked the `unrlink()` system call has the required permissions on the from-file and its parent directory (see Appendix A for a discussion of the permission semantics). It does so by invoking `permission()` once each for the from-file and its parent directory passing each time their respective inodes and the permission flags.
5. If the input to-file parameter is `NULL` (which means the process wants to delete rlinks irrespective of the to-file) assigns `LAST_NO_FILE` (defined as -1) to the to-file's nameidata structure's `last_type` field.
6. If the input to-file parameter is not NULL, invokes `to_file_lookup()` passing as parameters the to-file (which it copies to a kernel-space variable using `getname()` before this invocation) and the address of a local nameidata structure. The function `to_file_lookup()` performs the following tasks:
   a) Invokes `path_lookup()` passing as parameters the to-file path, the `LOOKUP_POSITIVE` flag and the nameidata structure received as parameter. This function fills the dentry field of the nameidata structure and returns zero indicating success if the to-file exists. If the to-file does not exist or any other failure occurs, the function returns a negative number indicating the type of failure.
   b) If the `path_lookup()` invocation in the previous step returns a negative value not equal to `-ENOENT`, returns the value to the caller.
   c) If the invocation in step 6a returns successfully and the returned nameidata structure contains a valid inode (of the to-file), checks if the process has the required permissions on the to-file by invoking `permission()`.
   d) If the return value from the invocation in step 6a is `-ENOENT`, iterates through the components of to-file's path until an existing ancestor of the to-file is found. On each iteration it removes the current last component of the path, using `chop_last_component()`, and checks for the existence of the resulting filename (which is always a directory, because it was an inner component of the to-file path before the last component was chopped) by invoking `path_lookup()`. When it finds such a file, checks if the process has the

required permissions (MAY_EXEC) on the file by invoking permission(). While iterating it stores the chopped components of the to-file's path in the nameidata structure's last.name field.

7. If the to_file_lookup() invocation in step 6 returns an error, releases the dentry and nameidata structures obtained in steps 2 and 3 and returns the error.
8. If the to_file_lookup() invocation in step 6 returns successfully, checks if the inode returned by the function belongs to the same filesystem as the from-file. If this check fails, releases the structures obtained in steps 2, 3 and 6 and returns the error code -EXDEV.
9. If the unrlink() inode operation is defined for the from-file's parent directory's inode, invokes it passing as parameters the parent directory's inode, the dentry of the from-file, the nameidata structure of the to-file (or its ancestor returned by to_file_lookup()) and the relation-name.
10. Releases all the structures allocated earlier and returns the return value of the invocation in step 9.

RelExt2 Layer

The RelExt2 layer function gsfs_unrlink() executes when sys_unrlink() invokes the unrlink() inode operation of an inode that belongs to RelExt2. gsfs_unrlink() receives as parameters the from-file's parent directory's inode, the dentry of the from-file, the nameidata structure of the to-file (or its ancestor) (say, nd_tofile) and the relation-name. It performs the following tasks:

1. If the last_type field of the input nameidata structure (nd_tofile->last_type) equals LAST_NO_FILE, assigns NULL to a local variable, tofile_path, that stores the to-file's filesystem absolute path.
2. If the last_type field checked in step 1 is not equal to LAST_NO_FILE, performs the following tasks to obtain the to-file's filesystem-absolute path into the local variable, tofile_path.
   a) Invokes get_path(), passing as parameters the to-file's nameidata structure (nd_tofile), a page-size buffer and the size of the buffer, to obtain the to-file's filesystem-absolute path (see the RelExt2 Layer section of section 6.3.1 for a detailed description of get_path()).
   b) Removes the beginning slash from the path returned by get_path(), because every filesystem-absolute path has the starting slash, making it redundant to store. It now matches the path stored in the rlink block.
   c) Checks nd_tofile's last.name field for any chopped components of the to-file. Appends all such components to the end of the path obtained in step 2b.
3. Invokes ext2_find_entry() passing as parameters the inode of the directory, the dentry of from-file and the address of a pointer variable to a page structure. ext2_find_dentry() populates the pointer variable with the address of the page in directory file that contains the from-file's entry and returns the address of the directory entry (of type gsfs_dir_entry_2) within the page.
4. If the rlink_blk field of the directory entry obtained in step 3 is zero, releases the page allocated in step 3 and returns the error code -ENOENT.

5. If the `rlink_blk` field is not zero, invokes `sb_bread()`, passing the `rlink_blk` value as one of the parameters, to read the corresponding block in to a buffer. `sb_bread()` returns the address of the buffer head of the buffer.
6. Invokes the rlink-block layer function `delete_rlink_blk()` passing as parameters the buffer head obtained in step 5, the to-file's path obtained in either step 1 or 2 and the relation-name input parameter. This function compares its parameters to the rlink block's data and deletes the matching rlinks. This invocation is surrounded by calls to `lock_buffer()` and `unlock_buffer()`, passing the buffer head obtained in step 5 as parameter, to prevent operations on the buffer by other processes.
7. Marks the buffer as dirty using the `mark_buffer_dirty()` function.
8. Inserts the buffer in the directory inode's list of dirty data buffers using `buffer_insert_inode_data_queue()`.
9. Releases the buffer using `__brelse()`.
10. Decrements the usage counter of directory's page (obtained in step 3) using `ext2_put_page()`.

Rlink-block layer

The RelExt2 layer function `gsfs_unrlink()` invokes the rlink-block layer function `delete_rlink_blk()`, passing as parameters the buffer head of the rlink block, the to-file's path and the relation-name, to delete the contents of the rlink block's buffer. It performs the following tasks:

1. If the relation-name string is of non-zero length,
   a) Invokes `rel_name_exists()` passing as parameters the address of the buffer and the relation-name. `rel_name_exists()` returns the address of the relation-entry within the buffer if a relation-entry exists for the input relation-name.
   b) If a matching relation-entry does not exist, returns the error code -ENOENT.
   c) If a matching relation-entry exists and the to-file parameter is NULL, overwrites the relation-entry with the buffer's contents following the relation-entry, thus deleting the relation-entry. It also updates the `free_space_area` header field of the buffer.
   d) If the relation-entry exists and the to-file parameter is not `NULL`,
      1. Checks if the to-file exists in the relation-entry obtained in step 1a by invoking `tofile_exists()`.
      2. If the to-file doesn't exist, returns the error code `-ENOENT`.
      3. If the to-file exists, invokes `delete_to_file()` to delete the to-file. `delete_to_file()` deletes the to-file by moving the contents of the buffer following the to-file's path to the address of the to-file's path, thus deleting it. If the to-file was the last to-file of the relation-entry, deletes the relation-entry by moving the contents following the entry to the beginning of the entry. Finally, it updates the `free_space_area` header field accordingly.
2. If the relation-name string is of zero-length, meaning no relation-name was specified by the process,
   a) If the to-file input parameter is `NULL`, invokes `reset_rlink_blk()` (see section 6.3.1 for an explanation of `reset_rlink_blk()`) to delete all the rlinks in the buffer.

b) If the to-file parameter is not `NULL`, iterates through each relation-entry, using the `reclen` field of the entry, to delete the to-file in it. During each iteration it invokes `delete_to_file()` (see step 1d3) passing as parameters the buffer, the relation-entry pertaining to that iteration, and the to-file.

7. Applications

Applications using rlinks can be classified into those that view rlinks as connecting related files and those that view rlinks as creating a graph of the filesystem and apply graph algorithms on the resulting graph.

7.1 Rlinks as connecting related files

`ls` with rlinks

`ls` is a Unix utility that lists directory contents. Most Linux distributions include the Free Software Foundation's version of the implementation of this utility; the source code is governed by GNU's General Public License(GPL) Version 2 and is distributed as part of GNU's Coreutils package. The version of Coreutils used in this thesis is 6.7.

The complete syntax for using `ls`, along with a complete listing of its options, can be found in its man page by typing `"man ls"` at the command line. In this thesis, I introduce three new options — `j`, `y`, `Y` — that dictate the type of rlink information printed for the listed files. Each of these options is explained below.

`j` print an additional column at the beginning of a file's information, displaying "`r`" if the file has at least one rlink and "-" if the file has none.
`y` print the relation-names associated with each file.
`Y` print the relation-names associated with a file, along with the to-files associated with each relation-name. Each to-file is displayed in a separate line and is grouped under a relation-name.

The `ls` output for each of the three options is illustrated below.

```
% ls -j
- ls                    r catch_me_if_you_can.avi  r raging_bull.avi  r
titanic.avi
r Gangs_of_New_York.avi  - goodfellas.avi           - taxi_driver.avi
r casino.avi             r minority_report.avi      r the_aviator.avi


% ls -jy
- ls                                          r minority_report.avi
(spielberg)
r Gangs_of_New_York.avi (scorsese,dicaprio)   r raging_bull.avi
(joepesci,scorsese)
r casino.avi (scorsese,deniro,joepesci)       - taxi_driver.avi
r catch_me_if_you_can.avi (dicaprio,spielberg)  r the_aviator.avi
(scorsese,dicaprio)
-  goodfellas.avi                               r titanic.avi (dicaprio)
```

```
% ls -Yl casino.avi
-rw-r--r-- 1 naveen users 0 2007-06-19 05:29 casino.avi
scorsese:
  --> /mnt/gsfs/movies/goodfellas.avi
  --> /mnt/gsfs/movies/raging_bull.avi
  --> /mnt/gsfs/movies/taxi_driver.avi
  --> /mnt/gsfs/movies/Gangs_of_New_York.avi
deniro:
  --> /mnt/gsfs/movies/taxi_driver.avi
  --> /mnt/gsfs/movies/goodfellas.avi
  --> /mnt/gsfs/movies/raging_bull.avi
joepesci:
  --> /mnt/gsfs/movies/raging_bull.avi
```

`ls` accepts an additional option `--color` to display relation-names and to-files in distinguishing colors.

Developers can integrate rlinks into other Unix utilities like `cp` (copy files; see `man cp`) and `ln` (link files; see `man ln`) by similarly introducing such new options. For example "`cp -y`" could create rlinks (with relation-names, say, "copiedfrom" and "copiedto") between the files involved in copying; these links help a user in keeping track of transitions of files in the filesystem. In the same way "`ln -y`" could create rlinks (with relation-names, say, "linkfrom" and "linkto") between the two files involved in linking. Rlinks created between hard links keep track of hard links of a file.

Related-file search

A related-file search (rfs) program receives as input a set of files. It scans the rlink information of these files to return files that are related to the input files in the order of their strength of relation.

7.2 Rlinks to represent a graph

Dijkstra's shortest path algorithm

Rlinks give a graphical view of the filesystem. The dijk application illustrates applying graph algorithms to such a view. Given two filenames, dijk finds the shortest path to reach one file from the other applying Dijkstra's shortest path algorithm.

The distance from file A to file B is the integer equivalent of the relation-name of rlink from file A to the file B. That is, if there exists an rlink with relation-name "5" between file A and file B, then the distance from A to B is 5. Rlinks with non-integer-equivalent relation-names are not considered part of the graph.

dijk starts at the first of the two input files (say, A and Z), scans its rlinks, adds the to-file with the lowest 'cost' to the tree (implemented by a linked list), and updates the cost of to-files that are reachable from files in the tree. In the next iteration, dijk adds the reachable file with the lowest cost to the tree and accordingly updates the cost of other reachable files. This process continues until the destination file (Z) is added to the tree, at which time the cost and the minimal path are returned.

To illustrate, consider the graph in Figure 7.1.

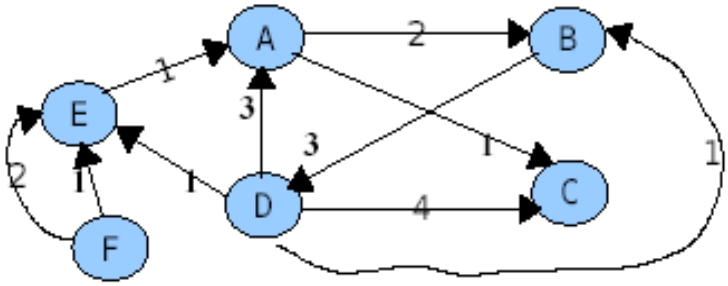

Figure 7.1: An rlink graph

Each node in the graph is a file in the filesystem; each edge is an rlink of relation-name representing the label of the edge. The output of `readrlink` command of the file `A`, for example, is as follows:

```
% readrlink A
A:
  1:
    -->   /mnt/gsfs/graph/C
  2:
    -->   /mnt/gsfs/graph/B
```

The following command displays the shortest distance from file B to file A.

```
% dijk B A
The minimum total cost is 5
graph/A
  ^
  | 1
graph/E
  ^
  | 1
graph/D
  ^
  | 3
graph/B
  ^
  | 0
```

The output shows that the shortest path from file `B` to file `A` is `B` → `D` → `E` → `A`, and the shortest distance is 5.

8 Conclusions

In this thesis I have introduced a new type of link, rlink, to store related-file information of a file. I defined a set of semantics for the links, made changes to Linux's VFS layer to accommodate the new links, modified a popular filesystem, Ext2, to implement and support rlinks. I implemented a few applications that illustrate the use of rlinks. From this work I draw the following conclusions:

- Rlinks provide a new way to link files.
- Rlinks are effective in providing related-file information of any chosen file.
- Because related-file information is spread across the filesystem instead of in a centralized location, rlinks are robust during storage system failures.
- Rlinks form overlay graphs between files of a filesystem, thus providing useful alternative views of the filesystem.
- Rlinks provide a fast alternative to using a full-fledged semantic file system for storing related-file information.
- Rlinks can be implemented in existing filesystems with only slight modifications.

9 Future Work

Several ideas arose during development that I did not implement either due to time constraints or the lack of empirical knowledge of use of rlinks. Here is a list of some of those ideas.

1. Write a program to convert an Ext2 filesystem to a RelExt2 filesystem. This program would add an extra field to all the directory entries in the directory files of the filesystem.
2. If the rlink block size is too restrictive for applications, the single block can be replaced by a linked list of blocks connected by block numbers stored in the currently reserved fields of each rlink block header.
3. Even a change to a filesystem as minor as adding a field to a directory entry could be too drastic for mass adoption. Hence the existing Extended Attributes feature in the Linux 2.6 kernel can be enhanced to store *rlink* attributes. The attribute name-value pairs, for example, could be,

   ```
   user.rlink.relationname: /the/related/to-file
   ```

   This enhancement requires implementing the rlink semantics and resolving the pathnames before setting and retrieving the rlink extended attributes.
4. More utilities can be added for operations like renaming a relation-name, copying rlink information between files, and retrieving only the relation-names of a file.

Appendix A: Rlink Permission semantics

While designing rlinks, we considered a few sets of permission semantics for rlinks. In this section I describe three important sets of semantics along with advantages and disadvantages of each of them.

Set 1:
The table A.1 shows the set of permissions required by each rlink operation on the files involved in the operation. The permission "*access*" for a file means that the file should be accessible to the user. A file is accessible to a user when the file exists and the user has *execute* permission on all directories in the file's path name. The access permission allows an application to determine the existence of a file.

Table A.1

Parameters / System call	From-File	To-file	From-file's parent directory
Rlink	Read	Read	Execute+Write
Readrlink	Access	None	Execute+Read
Unrlink	Read	Read/Ancestor Access	Execute+Write

The semantics of requiring read access on both from-file and to-file to rlink or unrlink stems from the idea that a user should know the content of both the files in order to decide whether a relation exists between them. The execute permission on the from-file's parent directory ensures that the from-file is accessible.

The implementation of rlinks in this thesis stores rlink information of a file in the file's parent directory. Hence, according to Unix's file-access semantics, the user needs write permission on the from-file's parent directory to perform rlink or unrlink operations. However, the rlink implementation doesn't require read permission for from-file and to-file. It is a constraint imposed by our chosen rlink semantics in addition to Unix's file-access semantics.

If a to-file is missing when an unrlink operation is attempted, the accessibility of the to-file to the process, had the to-file existed, is checked. That is, to successfully perform such an operation, the process must have execute permission on all the to-file's existing ancestors (starting from its parent directory).

Advantages:

1. Because users creating rlinks have read access to both from-file and to-file, rlinks when created are more meaningful than rlinks created by users who do not have access to contents of the file.
2. Because rlinks are created between files that exist, it is possible to enforce a constraint that both the files belong to the same filesystem. Such a constraint ensures that the links are valid when the partition is mounted on different systems or different directories of the same system.

3. Because the existence of the to-file is checked when an rlink is created, rlinks are more likely to be valid than rlinks that were created without such a check on to-file (as in Set 3 below), because the to-file existed at some time in the past.

Disadvantages:

1. The semantics are not consistent. Even though the existence of both files is checked during rlink creation, the constraint is not maintained throughout the lifetime of rlinks.
2. The semantics are very restrictive, because they enforce more constraints than required by Unix's file-access semantics to perform the operations.

Set 2:

Table A.2

Parameters / System call	From-File	To-file	From-file parent directory
Rlink	Access	Access	Execute+Write
Readrlink	Access	None	Execute+Read
Unrlink	Access	Access/ Ancestor Access	Execute+Write

This set of semantics differs from Set 1 in that read permission on the from-file and to-file is not required to perform rlink and unrlink operations.

Advantages:

1. Advantages 2 and 3 from Set 1.
2. Less restrictive semantics than Set 1, allowing greater freedom for users to create and delete rlinks.

Disadvantages:

1. Disadvantage 1 of Set 1.
2. Rlinks are less meaningful than in Set 1. Even though this set of semantics ensures the two files involved in rlink operations exist, it doesn't ensure that the content in files is related because the user may not have read permission to check the content.

Set 3:

Table A.3

Parameters / System call	From-File	To-file	From-file parent directory
Rlink	Access	None (Treated as text)	Execute+Write
Readrlink	Access	None	Execute+Read
Unrlink	Access	None (Treated as text)	Execute+Write

This set of semantics checks for the existence of the from-file for all three rlink operations but considers the to-file name given as a string. The rlink and unrlink operations are successful even if the to-file does not exist, or exists on a different filesystem or is not even a valid filename. These to-file semantics are very similar to the source-filename semantics for creating symbolic links. This set of semantics represents the minimal permission requirements to perform rlink operations that conform with the Unix's file-access semantics for creating rlinks.

Advantages

1. Semantics are consistent (a disadvantage of Sets 1 and 2). Rlinks are never considered as created with valid to-files.
2. Because this set of semantics is very similar to that of familiar symbolic links, users can easily understand and use rlinks.
3. The to-file field of an rlink can contain informative text. This information can be shared by other users and applications.
4. Because to-files can exist on any filesystem, rlinks can be created between files across filesystems.

Disadvantages

1. Links are less likely to be valid, that is, both components of an rlink — the from-file and the to-file — exist, compared to Sets 1 and 2. Links are less likely to be valid; firstly because rlinks are potentially created between files that never existed, and secondly, if a non-root filesystem is mounted at different place than it was when rlinks were created for its files, the entire rlink information of the partition becomes invalid. This loss of information leads to users losing trust in rlink information.
2. Because the to-file can be an arbitrary character string, rlinks are no longer links between files, but are a way of storing meta-data of from-files.

For this thesis I chose to implement the semantics in Set 1, because the goal of this thesis is to provide users with an effective mechanism to navigate through related files of a file. Sets 2 and 3 do not enforce the relatedness of files' content. The loss of information between mounts of a filesystem at different mount points makes Set 3 ineffective.

Appendix B: Data Structures Reference

2.1 VFS Super block object

```
struct super_block {
        struct list_head        s_list;         /* Keep this first */
        kdev_t                  s_dev;          /* Device identifier */
        unsigned long           s_blocksize;    /* Block size in bytes */
        unsigned char           s_blocksize_bits;/* Block size in # of bits */
        unsigned char           s_dirt;         /* Modified (dirty) flag */
        unsigned long long      s_maxbytes;     /* Max file size */
        struct file_system_type *s_type;        /* Filesystem type */
        struct super_operations *s_op;          /* Super block methods */
        struct dquot_operations *dq_op;        /* Disk quota methods */
        struct quotactl_ops     *s_qcop;
        unsigned long           s_flags;       /* Mount flags */
        unsigned long           s_magic;       /* Filesystem magic number */
        struct dentry           *s_root;       /* Dentry of mount directory */
        struct rw_semaphore     s_umount;      /* Semaphore used for unmounting
*/
        struct semaphore        s_lock;        /* Super block semaphore */
        int                     s_count;       /* Reference counter */
        atomic_t                s_active;      /* Secondary reference counter
*/
        struct list_head        s_dirty;        /* dirty inodes */
        struct list_head        s_locked_inodes;/* inodes being synced */
        struct list_head        s_files;        /* List of file objects
assigned to super block */

        struct block_device     *s_bdev;        /* Pointer to the block device
descriptor */
        struct list_head        s_instances;    /* Pointers for a list of
superblock objects of a given filesystem type */
        struct quota_info       s_dquot;        /* Diskquota specific options
*/
        union u;
        struct semaphore s_vfs_rename_sem;      /* Kludge */
        struct semaphore s_nfsd_free_path_sem;
}


struct super_operations {
        struct inode *(*alloc_inode)(struct super_block *sb);
        void (*destroy_inode)(struct inode *);
        void (*read_inode) (struct inode *);
        void (*read_inode2) (struct inode *, void *) ;
        void (*dirty_inode) (struct inode *);
        void (*write_inode) (struct inode *, int);
        void (*put_inode) (struct inode *);
        void (*delete_inode) (struct inode *);
        void (*put_super) (struct super_block *);
        void (*write_super) (struct super_block *);
        int (*sync_fs) (struct super_block *);
        void (*write_super_lockfs) (struct super_block *);
        void (*unlockfs) (struct super_block *);
        int (*statfs) (struct super_block *, struct statfs *);
        int (*remount_fs) (struct super_block *, int *, char *);
        void (*clear_inode) (struct inode *);
```

```
        void (*umount_begin) (struct super_block *);
        struct dentry * (*fh_to_dentry)(struct super_block *sb, __u32 *fh, int
len, int fhtype, int parent);
        int (*dentry_to_fh)(struct dentry *, __u32 *fh, int *lenp, int
need_parent);
        int (*show_options)(struct seq_file *, struct vfsmount *);

}
```

VFS Inode Object

```
struct inode {
        struct list_head        i_hash;
        struct list_head        i_list;
        struct list_head        i_dentry;

        struct list_head        i_dirty_buffers;
        struct list_head        i_dirty_data_buffers;

        unsigned long           i_ino;
        atomic_t                i_count;
        kdev_t                  i_dev;
        umode_t                 i_mode;
        unsigned int            i_nlink;
        uid_t                   i_uid;
        gid_t                   i_gid;
        kdev_t                  i_rdev;
        loff_t                  i_size;
        time_t                  i_atime;
        time_t                  i_mtime;
        time_t                  i_ctime;
        unsigned int            i_blkbits;
        unsigned long           i_blksize;
        unsigned long           i_blocks;
        unsigned long           i_version;
        unsigned short          i_bytes;
        struct semaphore        i_sem;
        struct rw_semaphore     i_alloc_sem;
        struct semaphore        i_zombie;
        struct inode_operations *i_op;
        struct file_operations  *i_fop; /* former ->i_op->default_file_ops */
        struct super_block      *i_sb;
        wait_queue_head_t       i_wait;
        struct file_lock        *i_flock;
        struct address_space    *i_mapping;
        struct address_space    i_data;
        struct dquot            *i_dquot[MAXQUOTAS];
        /* These three should probably be a union */
        struct list_head        i_devices;
        struct pipe_inode_info  *i_pipe;
        struct block_device     *i_bdev;
        struct char_device      *i_cdev;

        unsigned long           i_dnotify_mask; /* Directory notify events */
        struct dnotify_struct   *i_dnotify; /* for directory notifications */

        unsigned long           i_state;

        unsigned int            i_flags;
        unsigned char           i_sock;
```

```
        atomic_t                i_writecount;
        unsigned int            i_attr_flags;
        __u32                   i_generation;
          union                           u;
}

struct inode_operations {
        int (*create) (struct inode *,struct dentry *,int);
        struct dentry * (*lookup) (struct inode *,struct dentry *);
        int (*link) (struct dentry *,struct inode *,struct dentry *);
        int (*unlink) (struct inode *,struct dentry *);
        int (*symlink) (struct inode *,struct dentry *,const char *);
        int (*mkdir) (struct inode *,struct dentry *,int);
        int (*rmdir) (struct inode *,struct dentry *);
        int (*mknod) (struct inode *,struct dentry *,int,int);
        int (*rename) (struct inode *, struct dentry *,
                        struct inode *, struct dentry *);
        int (*readlink) (struct dentry *, char *,int);
        int (*follow_link) (struct dentry *, struct nameidata *);
        void (*truncate) (struct inode *);
        int (*permission) (struct inode *, int);
        int (*revalidate) (struct dentry *);
}
```

VFS dentry object

```
struct dentry {
        atomic_t d_count;
        unsigned int d_flags;
        struct inode  * d_inode;        /* Where the name belongs to - NULL is
negative */
        struct dentry * d_parent;       /* parent directory */
        struct list_head d_hash;        /* lookup hash list */
        struct list_head d_lru;         /* d_count = 0 LRU list */
        struct list_head d_child;       /* child of parent list */
        struct list_head d_subdirs;     /* our children */
        struct list_head d_alias;       /* inode alias list */
        int d_mounted;
        struct qstr d_name;
        unsigned long d_time;           /* used by d_revalidate */
        struct dentry_operations  *d_op;
        struct super_block * d_sb;      /* The root of the dentry tree */
        unsigned long d_vfs_flags;
        void * d_fsdata;                /* fs-specific data */
        unsigned char d_iname[DNAME_INLINE_LEN]; /* small names */
};

struct dentry_operations {
        int (*d_revalidate)(struct dentry *, int);
        int (*d_hash) (struct dentry *, struct qstr *);
        int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
        int (*d_delete)(struct dentry *);
        void (*d_release)(struct dentry *);
        void (*d_iput)(struct dentry *, struct inode *);
};
```

VFS File Object

```
struct file {
        struct list_head        f_list;
        struct dentry           *f_dentry;
        struct vfsmount         *f_vfsmnt;
        struct file_operations  *f_op;
        atomic_t                f_count;
        unsigned int            f_flags;
        mode_t                  f_mode;
        loff_t                  f_pos;
        unsigned long           f_reada, f_ramax, f_raend, f_ralen, f_rawin;
        struct fown_struct      f_owner;
        unsigned int            f_uid, f_gid;
        int                     f_error;

        size_t                  f_maxcount;
        unsigned long           f_version;

        /* needed for tty driver, and maybe others */
        void                    *private_data;

        /* preallocated helper kiobuf to speedup O_DIRECT */
        struct kiobuf           *f_iobuf;
        long                    f_iobuf_lock;
};

struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned
long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*readv) (struct file *, const struct iovec *, unsigned long,
loff_t *);
        ssize_t (*writev) (struct file *, const struct iovec *, unsigned long,
loff_t *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t
*, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long,
unsigned long, unsigned long, unsigned long);
};
```

file_system_type structure

```
struct file_system_type {
        const char *name;
        int fs_flags;
        struct super_block *(*read_super) (struct super_block *, void *, int);
        struct module *owner;
```

```
        struct file_system_type * next;
        struct list_head fs_supers;
};
```

vfsmount structure

```
struct vfsmount
{
        struct list_head mnt_hash;
        struct vfsmount *mnt_parent;    /* fs we are mounted on */
        struct dentry *mnt_mountpoint;  /* dentry of mountpoint */
        struct dentry *mnt_root;        /* root of the mounted tree */
        struct super_block *mnt_sb;     /* pointer to superblock */
        struct list_head mnt_mounts;    /* list of children, anchored here */
        struct list_head mnt_child;     /* and going through their mnt_child */
        atomic_t mnt_count;
        int mnt_flags;
        char *mnt_devname;              /* Name of device e.g. /dev/dsk/hda1 */
        struct list_head mnt_list;
};
```

nameidata structure

```
struct nameidata {
        struct dentry *dentry;
        struct vfsmount *mnt;
        struct qstr last;
        unsigned int flags;
        int last_type;
};
```

Buffer head structure

```
struct buffer_head {
        /* First cache line: */
        struct buffer_head *b_next;     /* Hash queue list */
        unsigned long b_blocknr;        /* block number */
        unsigned short b_size;          /* block size */
        unsigned short b_list;          /* List that this buffer appears */
        kdev_t b_dev;                   /* device (B_FREE = free) */

        atomic_t b_count;               /* users using this block */
        kdev_t b_rdev;                  /* Real device */
        unsigned long b_state;          /* buffer state bitmap (see above) */
        unsigned long b_flushtime;
        struct buffer_head *b_next_free;/* lru/free list linkage */
        struct buffer_head *b_prev_free;/* doubly linked list of buffers */
        struct buffer_head *b_this_page;/* circular list of buffers in one page
*/
        struct buffer_head *b_reqnext;  /* request queue */
        struct buffer_head **b_pprev;   /* doubly linked list of hash-queue */
        char * b_data;                  /* pointer to data block */
        struct page *b_page;            /* the page this bh is mapped to */
        void (*b_end_io)(struct buffer_head *bh, int uptodate);
        void *b_private;                /* reserved for b_end_io */
        unsigned long b_rsector;        /* Real buffer location on disk */
        wait_queue_head_t b_wait;
        struct list_head     b_inode_buffers;   /* doubly linked list of inode
dirty buffers */
}
```

Ext2 Super Block

```
struct ext2_super_block {
        __u32   s_inodes_count;         /* Inodes count */
        __u32   s_blocks_count;         /* Blocks count */
        __u32   s_r_blocks_count;       /* Reserved blocks count */
        __u32   s_free_blocks_count;    /* Free blocks count */
        __u32   s_free_inodes_count;    /* Free inodes count */
        __u32   s_first_data_block;     /* First Data Block */
        __u32   s_log_block_size;       /* Block size */
        __s32   s_log_frag_size;        /* Fragment size */
        __u32   s_blocks_per_group;     /* # Blocks per group */
        __u32   s_frags_per_group;      /* # Fragments per group */
        __u32   s_inodes_per_group;     /* # Inodes per group */
        __u32   s_mtime;                /* Mount time */
        __u32   s_wtime;                /* Write time */
        __u16   s_mnt_count;            /* Mount count */
        __s16   s_max_mnt_count;        /* Maximal mount count */
        __u16   s_magic;                /* Magic signature */
        __u16   s_state;                /* File system state */
        __u16   s_errors;               /* Behaviour when detecting errors */
        __u16   s_minor_rev_level;      /* minor revision level */
        __u32   s_lastcheck;            /* time of last check */
        __u32   s_checkinterval;        /* max. time between checks */
        __u32   s_creator_os;           /* OS */
        __u32   s_rev_level;            /* Revision level */
        __u16   s_def_resuid;           /* Default uid for reserved blocks */
        __u16   s_def_resgid;           /* Default gid for reserved blocks */
        __u32   s_first_ino;            /* First non-reserved inode */
        __u16   s_inode_size;           /* size of inode structure */
        __u16   s_block_group_nr;       /* block group # of this superblock */
        __u32   s_feature_compat;       /* compatible feature set */
        __u32   s_feature_incompat;     /* incompatible feature set */
        __u32   s_feature_ro_compat;    /* readonly-compatible feature set */
        __u8    s_uuid[16];             /* 128-bit uuid for volume */
        char    s_volume_name[16];      /* volume name */
        char    s_last_mounted[64];     /* directory where last mounted */
        __u32   s_algorithm_usage_bitmap; /* For compression */
        __u8    s_prealloc_blocks;      /* Nr of blocks to try to preallocate*/
        __u8    s_prealloc_dir_blocks;  /* Nr to preallocate for dirs */
        __u16   s_padding1;
        __u32   s_reserved[204];        /* Padding to the end of the block */
};
```

References

[1] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O Toole. "*Semantic file systems". Proceedings of the Symposium on Operating Systems Principles*, pages 16-25, 1991.

[2] Ahmed Salama, Ahmed Samih Amr Ramadan, Karim M. Yousef. "GNU/Linux Semantic Storage System".

[3] Dominic Giampaolo. *Practical File System Design with the Be File System*. Morgan Kaufmann Publishers, Inc, San Francisco, California, 1999.

[4] Paul Dourish et al. "Extending Document Management Systems with User-Specific Active Properties". *ACM Transactions on Information Systems (TOIS)*, **18**(2):140-170, 2000.

[5] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2nd edition, 2002.

[6] Robert Love. *Linux Kernel Development*. Novell Press, 2nd edition, 2005.

[7] Nick Murphy, Mark Tonkelowitz, and Mike Vernal. *The design and implementation of the database file system. 2002.*

Vita

Full Name: Naveen Akarapu

Date of Birth: August 30, 1980.

Place of Birth: Hyderabad, Andhra Pradesh, India.

Education: Bachelor of Technology in Computer Science and Engineering
Jawaharlal Nehru Technological University, Hyderabad, India, 2001.