The Design and Implementation of a Recovery Mechanism of Deleted Files for the Linux Ext2 File System

Master of Science Degree Project

Advised by:
Dr. Raphael A. Finkel
Professor
Department of Computer Science
University of Kentucky
Lexington

Samir Raizada
Department of Computer Science
University of Kentucky
Lexington

Abstract

This work studies possibilities of recovering user files on the Linux Operating System. The goal is to design a user level utility to undelete files with support from the Linux kernel. One possible approach for supporting the undelete facility in the linux kernel is implemented along with a user level undelete utility, for the ext2 file system.

Problem Statement

It is common for users to accidentally delete files (sometimes even recursively) which can be disastrous for the users as there are no ways to recover them on their own. The objective of this work is to provide a way for a user to recover deleted files. The recovery of deleted files requires changes to the Linux kernel and the addition of a utility to access the undelete support implemented in the kernel. The design and implementation of these changes involves study of the Linux file system structure, the Virtual File System layer, and the ext2 file system for which the undelete support is implemented.

Scope

The scope of the project includes the Linux operating system and the Second Extended File System. The study was limited to the undelete operation and does not consider archiving. This undelete operation may be able to recover deleted files upto a predefined period of time. The implementation is on Linux kernel 2.6.13 which may or may not work directly on other versions as a patch.

Introduction

This project modifies the ext2 file system to provide the facility to undelete a file deleted by a user and implements a user level undelete utility.

There are incidences of accidental deletion of files. If this happens, the user will need to get the super-user (root) to recover the files and this may take a long time and at times it is a complex process even for the super-user. Most of such accidental deletes happen due to the use to rm command with the -f or -rf option, with incorrect filename (and the consequences can be much worse with

wildcards). The user mostly realizes the blunder almost instantly, but it is often too late to do anything and depending on the speed and load, all or most of the files may be deleted.

File System Framework

The file system implementation in the traditional Unices was monolithic and they only supported one file system, namely, s5fs. As different file systems were designed, there was a need to develop a framework to support multiple file systems. There were several alternatives proposed and Sun Microsystems' Virtual File System architecture was later accepted as the de facto standard.

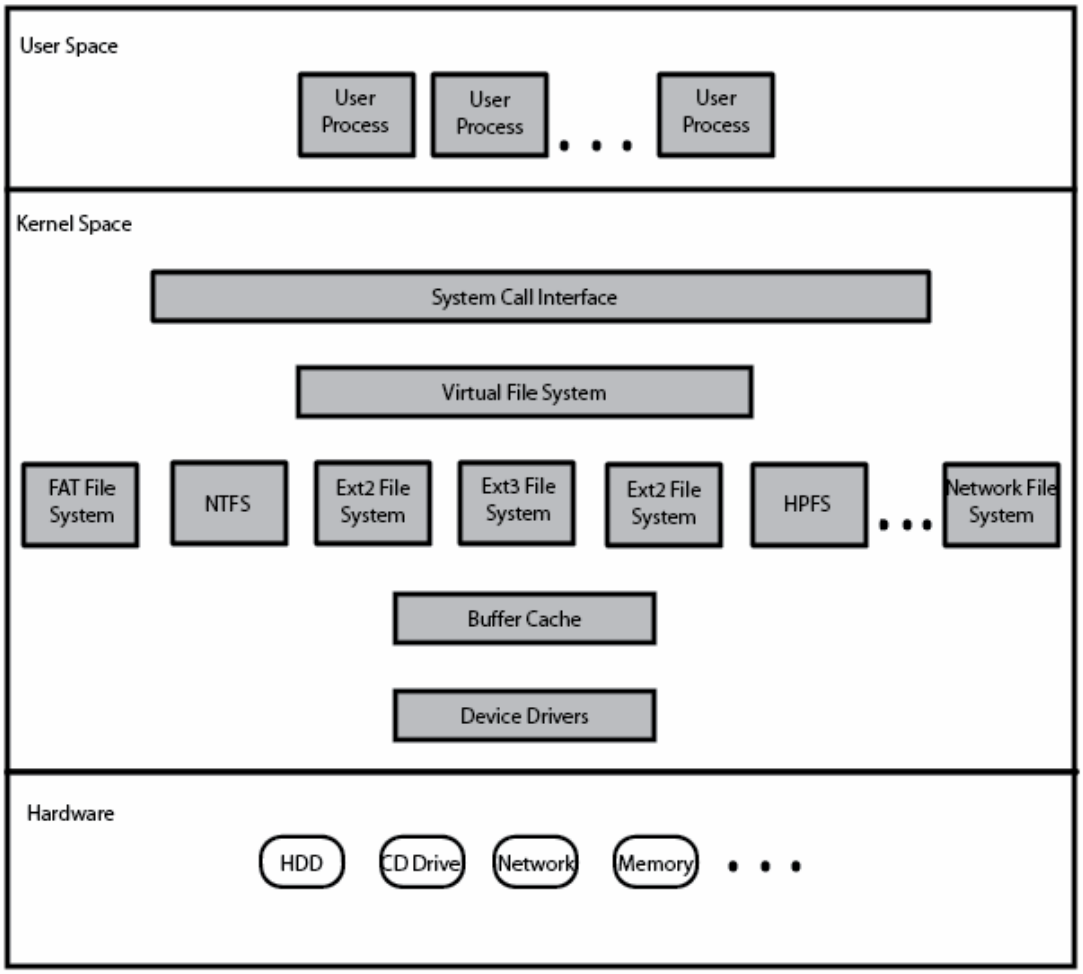

Fig 1. Organization of Linux File System

VFS provides a unified interface, called the Common File Model, for different file systems. It specifies a set of functions that each file system may implement, and these functions translate the physical organization of a specific file system to the VFS interface. These functions are revealed to the kernel when the file system is registered.

Whenever an operation on a file is initiated, the kernel calls the VFS function. This function then calls the corresponding function for the appropriate file system. To this end, VFS maintains the following objects:

Superblock object: Maintains information about the mounted file system.
Inode object: Maintains information about specific files in the file system.
File object: Maintains information regarding interaction between an open file and a process.
Dentry object: Links the directory entry with the corresponding file.

Ext2 File System
Linux was developed on the Minix Operating System and had support for the Minix file system. But it had a few limitations including limits on the file system size and file name size. The Ext2 file system was designed by Remy Card et al to overcome these limitations.

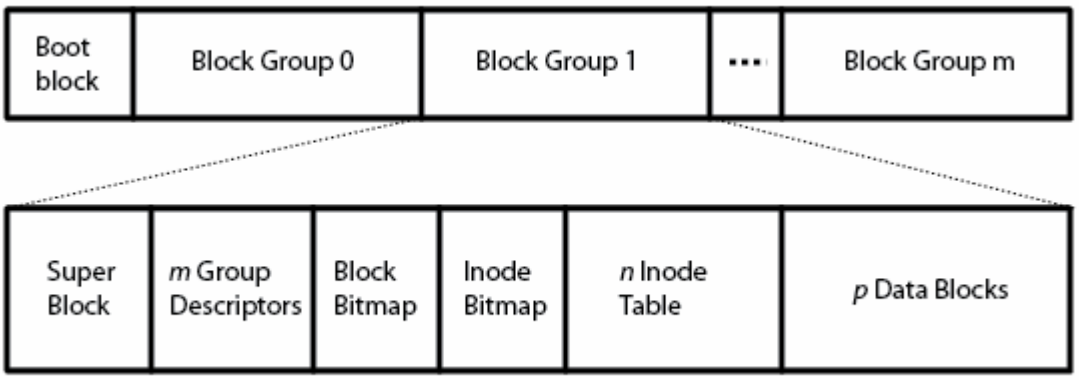

Fig 1. Layout of an Ext2 partition

Relevant File Operations
This section discusses the unlink system call and rm command.

Unlink
The unlink system call deletes a file name from a directory. If this file object was the last one pointing to the inode, and no processes have the file open, then the inodes' reference count is decremented. So the inode and the data blocks for this file are freed and made available. But if the file is open, and if its reference count is 0, then the file will remain in the system until all the file descriptors referring to it are closed.

When the unlink call is made, the filename is read into the kernel memory. Then the directory containing this file to be unlinked is resolved and the nameidata structure is filled with the corresponding information. Then the dentry corresponding to the last field of this nameidata structure (which is the last component of the file name) is searched for and returned. If it's not found in the cache, then a new dentry is created (otherwise, the reference count of the dentry is increased and no further action is taken). Finally, the permissions of the user for the file are checked. If the user has permission to unlink, the file systems unlink implementation is called.

The ext2 file systems' unlink function first finds the directory entry with the file name corresponding to the dentry's name field. Then the page that has this directory entry is also mapped and obtained. Finally the directory entry on the page is deleted and the inode change time is set.

rm
The rm command is used to delete one or more files and it provides different options. It is part of the coreutils package. Following outlines the operation of the rm command:

- First loads different libraries (shared object files).
- Uses lstat to get the file's detailed data structure.
- Verifies the permissions to the file.
- Uses the unlink system call to delete the file.

Possible Implementations of undelete

Undelete may be supported under Linux Operating System in different ways. Following are some of the possible implementations considered:

1. **Trash folder:** This implementation works by moving/transferring the deleted file to a different folder. There may just be one common folder for

all the users (with access permissions to view only one's own files) or one folder per user. These files may be recovered or moved back to the original folder or may be purged later. This approach is used by Microsoft's Windows operating system. The disadvantage with this is a lack of transparency to the user and the need for a manual deletion (this may be overcome by running a daemon process to periodically empty the trash folder.

2. **Delayed deletion:** This approach involves deletion of the directory entry, but keeping the inode and data blocks around for some time. This can be done in two ways:
    a. **Initial Reference Count:** Each new inode may start with a reference count of 1 (instead of 0) and so at any time an inode's reference count is always 1 more than the number of files referencing it. And deleting the files referencing an inode will decrement the reference count until it finally reaches 1.
    b. **Delayed deletion of inodes' reference count:** This approach delays decrementing of the inode's reference count for a specified amount of time.

3. **Recovering inodes**: This approach doesn't involve any modification to the unlink system call. The dentry corresponding to the file along with the inode block and the data blocks may be deleted. This implementation may keep some data corresponding to the deleted file, including the file name, owner's user ID, inode number (and may be the address of the block) and addresses of the data blocks (All this information may not be necessary but may be helpful in fast recovery).

    There are a few advantages to this approach: it's completely transparent to the user, and the files are deleted as they are, in the present system. The space that was used by these files may not be counted against the user.

    The allocation of data blocks and inode blocks is done in such a way that they are in the same block group and the kernel looks for the blocks in the same block group first and then looks for blocks in the other block group. So the data and inode blocks of a deleted file may be used sooner depending on the usage of that block group. This makes it seem impossible to be able to guarantee the recovery of files for any specified amount of time as there may be other users/sessions which may create files and may reuse the deleted inodes and data blocks.

4. **Support at the VFS layer**: An implementation at the VFS layer (without modifying the implementation of individual file systems) has the advantage of being available to all the file systems implementations. This can be accomplished by keeping track of the files that have to be deleted and then waiting for a certain amount of time before deleting them. But the problem with this implementation is that directory entries are maintained by specific file system implementations (the Ext2 file system manages each directory entry and so if a directory entry has to be deleted, a routine has to be implemented in the Ext2 file system and hence, it will involve modifying a specific file system) and the directory entry corresponding to the file has to be deleted to make it transparent to the user.

   One way to avoid changing the specific file systems is to maintain a list of files deleted and modify the dir system call to filter out these files and just expose the files which are still present. These files may later be undeleted or purged using a different interface or system call.

Implemented Approach

This section describes the delayed decrementing of the inode reference count, which is implemented in this project. In this approach, when a file is deleted, the directory entry is removed from the directory, but the inode's reference count is not decremented immediately, and it is delayed for a period of time as specified by a configuration option (the default is 30 minutes). As the inode still doesn't have all the information related to the file (specifically, the file name), some information about the deleted file is recorded. So the file system maintains information about the deleted files. This data is discarded and the inode's reference count is decremented at the end of a pre-defined recoverable time.

Deleted files which are still available for recovery (in the recoverable time) may be recovered by looking for the file in the data structure maintained by the file system. A directory entry is then created and the inode corresponding to the file being recovered is linked to this directory entry.

Since there are limits on the amount of disk space a user can use (implemented by quota system) and since these deleted files still count in the quota for the user who owns them, a user may want to get rid of the files in case that user is getting close to the limit. So this implementation provides for a mechanism to purge a deleted file (and also to purge all the deleted files.)

So, this provides a transparent way of deleting a file and still being able to recover the file at user level.

Implementation Details

This project implements 5 system calls, one of these a modification of the unlink system call to enable file recovery for the Ext2 file system. This section describes these system calls and the data structures.

The modified Ext2 file system maintains the user id of the owner, the name of the file and inode information for each file that has been deleted by the user but awaits purging by the system. These are stored in a linked list data structure, with head and tail pointers. A deleted file is always linked at the tail and a search for a deleted file is always started at the head of this linked list.

A timer is used to periodically purge the deleted files from the file system. This timer expires in the defined amount of time and each time it goes through the list and marks all the files in the list as ready for deletion.

An optimization to this system would be to not mark all structures corresponding to deleted files, instead just mark the first one (or store a pointer to the marked file). Anytime a file is purged manually, the mark can be passed on to the next file on the list. In the next pass, when the timer expires, all the files after the first marker are deleted. This approach will be implemented as time permits.

Now there is a possibility that a user may delete a lot of files and cause this data structure to swell. This may lead to a memory problems and a system crash. One way to solve this is to have a limit on the amount of memory this data structure can occupy. But, this may limit other users' ability to delete files. So, the approach this implementation takes is to have a per user limit. It maintains the number of files a user has deleted which are not purged yet in the *user_struct*. The pointers to the first and last files deleted by the user are also stored in the *user_struct* along with pointers to the previous deleted file structures for the same user in the ext2 file system data structure. This helps in finding the file to be purged (it will be the last file in the list for the user) efficiently in case the limit of the user who is trying to delete, is reached.

The system calls implemented are described below.

1. *safeunlink:*

This system call is a modified version of the unlink system call. This, like unlink system call, deletes the directory entry from the directory, but leaves the inode reference count unchanged. In addition it also adds this deleted file's information to the data structure being maintained. It also starts the timer which is used to periodically purge the deleted files from the file system, if it has not already been started. This implementation thus allows the older programs or software to behave the way they did on the original file system and provides a new system call which provides the facility to undelete. One of the design choices would have been to modify the existing unlink system call, which provides the undelete capability to all the files which use the old unlink and implement(s) a new system call (purge) which will delete the directory entry as well as decrement the inode's reference count.

2. *undelete_list*
   This system call can be used to retrieve the list of files available for a user to undelete. Implementation involves going through the linked list of deleted files and returning the names of files that may be undeleted for a particular user (all recoverable files for the root). These file names are absolute. If the size of the buffer passed is 0, then the size of the buffer required to hold the list is returned.

3. *undelete*
   The undelete system call recovers a deleted file, as specified by the file name parameter which is the absolute path, that is not purged from the file system. It goes through the list of deleted files and tries to find the file and recovers it.

4. *purge*
   This system call purges a deleted file, as specified by the absolute file name parameter, from the data structure, if the user is the owner of the file. This involves going through the linked list and finding the file to purge, fixing the markings/data structures and finally decrementing the inode's reference count, if the user is the owner of the inode.

5. *purgeall*
   The Purgeall system call purges all the deleted files owned by the user. The linked list is traversed and all the deleted files owned by the user are purged.

This kernel implementation for this project involved modifying the following files:

- arch/i386/kernel/entry.S
- include/asm-i386/unistd.h
- include/linux/fs.h
- include/linux/syscalls.h
- fs/Kconfig
- fs/namei.c
- fs/ext2/namei.c

The number of lines of code modified in the kernel is somewhere around 1000 (it may be 1500).

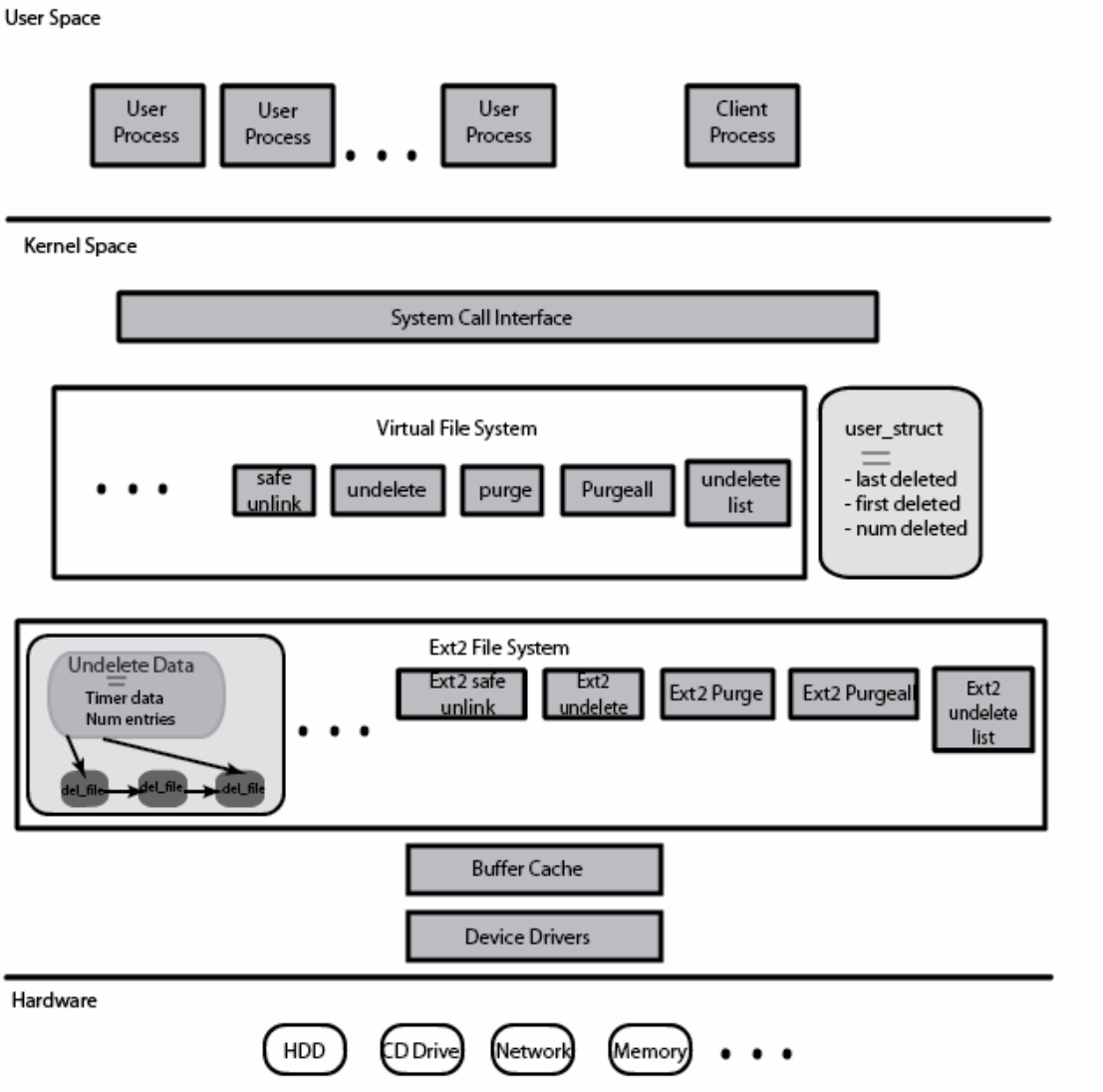

Fig 2. Modifications to the VFS layer and Ext2 File System

Non-kernel Modifications

To utilize the facilities implemented in the kernel, a few user level programs need to be implemented or modified. These changes are described in this section.

Undelete utility**:**
The undelete utility makes use of the system calls implemented to undelete the recoverable files. A user can invoke this utility to list the recoverable files, and then recover the files. It also enables a user to purge the deleted files from the file system.

rm command:
The rm utility needs to be modified or a new rm has to be written to be able to recover the deleted files. This utility should use the new safeunlink system call instead of the old unlink system call. The programs which create temporary files, for example, compilers, media editing programs, may still use the unlink. This helps user to recover space immediately after the temporary files that are created/used by these programs are deleted.

rm is part of coreutils which are a set of basic tools for file, text and shell manipulation. These are based on the glibc library and use the API provided by glibc

Performance

Performance measurements are done on a 300 MHz Pentium II machine with 192MB of RAM running 2.6.5-1 linux kernel. All the measurements are done at the user level (which may be affected by the scheduling).

Number of Operations	*delete*	*Safe delete*	*undelete*
1			
10			
100			

Future Work

As discussed earlier, there are different possibilities; this leaves scope for a lot of work. This section will cover some of this.

- An implementation may use the EXT2_UNRM_FL file attribute (of the inode) to decide if the file's undelete information should be stored. This attribute may be set and checked by chattr and lsattr shell commands.
- This implementation doesn't take into account the directory structure and an implementation which extends this to consider directory structure may be desirable.
- This implementation uses in-memory data structures to store the deleted file data. So this can be stored on persistent storage. Then the fsck may be modified to take this deleted file data into consideration so that the data and inode blocks are not deleted and are still recoverable.
- One can extend the approach to other file system implementations.
- One can implement the VFS file system implementation discussed in this document.

References

1. Linux Source Code (Kernel 2.6.13).
2. Daniel Bovet and Marco Caseti, Understanding the Linux Kernel, Second and Third Edition, O'Reilly.
3. Uresh Vahalia, Unix Internals: The new frontiers, Pearson Education.2001.
4. Michael Beck et. al., Linux Kernel Internals, Second Edition, Addison Wesley 2000.
5. Design and Implementation of the Second Extended Filesystem,
6. Jonathan Corbet and Alessandro Rubini, Linux Device Drivers, Third Edition, O'Reilly.
7. Gerlof Langeveld, Linux Kernel Internals: The File Subsystem, AT computing, Apr 2003.