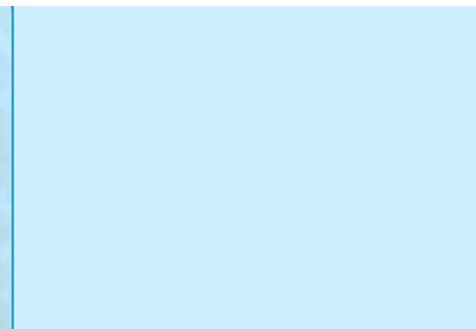
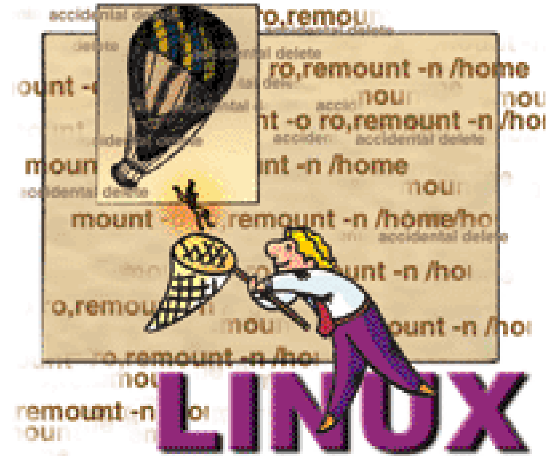


Recovering Deleted Files in Linux



Brian Buckeye and Kevin Liston

Most systems administrators have experienced a situation where a vital file has accidentally been deleted without a recent backup. In this article, we'll explore how to recover files in Linux. To begin, however, there are several caveats:

1. The methods described are emergency measures. They do not replace a working backup process to protect your data. You should also consider version control methods to protect your data from accidents.

2. File recovery is usually a time-consuming process, and often is not completely successful. Once a file is deleted, the space it occupied on the hard drive is marked as "available" and can be overwritten. DO NOT install any file recovery software on the drive that houses the file you want to recover.

3. These data recovery techniques involve elements of luck and timing, in addition to technique. If you've suffered an accidental deletion in the first place, luck isn't necessarily on your side.

4. Even if you do recover the file, there is no guarantee that it will have the same information that was contained in the original. Inspect anything you retrieve and verify the information

before you use it in production.

5. There are several factors acting against a successful recovery, including: time, file size, congestion of the disk partition, and the system activity:

• The more time that passes between the deletion of the file and the initiation of the recovery process, the less likely the process will succeed.

• The larger the size of the deleted file, the more likely damage has occurred.

• The more active the system, the more likely the blocks freed by the deletion will be overwritten by new data.

• If there is little free space on the disk partition, the smaller pool of available blocks increases the chance that the deleted data blocks will be re-used.

With those caveats in mind, we'll examine some options.

Linux and ext2

The default file system used by Linux is the Second Extended File system, referred to as ext2. (Ext3 with its use of journaling has also recently become common, but we will not cover it in this article.) The role of the file system itself is to abstract the physical structure of the storage media. On a physical level, a drive is a series of 512-byte sectors, addressable from 0 to *n* - 1. The file system is responsible for organizing these sectors into files and directories eventually used by applications via the operating system.

Blocks

The Linux file system, ext2, collects sectors into blocks. Ext2 supports block sizes of 1024, 2048, and 4096 bytes. Blocks are organized into block groups. Blocks are either data blocks or superblocks. Data blocks are general-purpose blocks used to store files and directories. Superblocks reside on the border of block groups and contain settings and status of the file system (e.g., formatting and cleanliness state). Block groups consist of a superblock, block allocation bitmap, inode allocation bitmap, inode table, and data blocks. Block groups are usually organized into 8*block-size blocks (e.g., 8192 blocks in a 1024-byte block-sized system). The block allocation bitmap keeps track of which blocks in the block group are in use (allocated vs. free). Our 1024-byte block size example has 1024 bytes responsible for tracking 8192 blocks. Thus, each block is mapped to one bit in the bitmap. (A "1" denotes allocated and a "0" denotes the block to be free.) The make-up of a block group includes a superblock, block allocation bitmap, inode bitmap, inode table, and data blocks.

The inode allocation bitmap work similarly, but typically uses less space than allocated, unless you have defined the system to have one inode per data block (which would be the case in a system optimized to handle a large amount of small files such as a news server). Inodes are special data-structures, each 128 bytes in length, which represent a file. By default, **mke2fs** (used to format an ext2 partition) reserves an inode for every 4096 bytes of file system space. The first ten inodes in a file system are special purpose:

1 — Bad blocks inode

2 — Root inode

3 — acl index inode (not supported)

4 — acl data inode (not supported)

5 — Boot loader inode

6 — Undelete directory inode (not implemented)

7-10 — Reserved

The bad blocks inode lists all of the data blocks on the file system that have detected unrecoverable errors. The root inode points to the directory file of **/**. The acl-related and undelete directory inodes are currently not implemented.

Pointers

Inodes contain information about a file, such as modification, access, creation (and deletion) time, ownership, permissions, size, and pointers to the actual data blocks of the file. There are 15 pointers to data blocks; the first 12 are references to direct blocks (actual file-data). The 13th pointer references the indirect block, which is a data block containing a list of 4-byte pointers to direct blocks (i.e., another 256 direct blocks in a 1024-byte block-sized system, 1024 direct blocks in a 4096-byte block-sized inode). The 14th pointer references the doubly indirect block, which is a block containing pointers to 256 (in the case of a 1024-byte block-sized file system) indirect blocks. In other words, the 14th pointer serves as the root of a tree that references 65536 data blocks in a 1024-byte block-sized file-system. The 15th pointer points to the triply indirect block, or a block full of references to doubly indirect blocks. In other words, this forms an asymmetrical tree-structure, where the inode references 15 children, the first 12 are terminal, the 13th has 1 level, the 14th has 2 levels, and the 15th has 3 levels. This causes the 1.6-GB file-size limit on 1024-byte block-sized systems.

Everything is a File

In Linux, directories are simply special files. The second inode in the file system points to **/**. This directory links to other subdirectories (which are other directory files). Directories are simply lists of four-tuples, consisting of an inode number, entry length, name length, and filename. The entry length denotes the length of the directory entry itself. This structure allows the use of long filenames without wasting disk space, but there is some waste from directories due to block size. This is why you see a size such as 1024 for **.** and **..** in the output of **ls -la**.

Also implemented with Linux is the **/proc** pseudo filesystem. Staying consistent with the UNIX everything-is-a-file metaphor, the **/proc** directory allows access to kernel data structures. The process structures are handy for data recovery. As root, change directory to **/proc/<pid>**, where **<pid>** is the process ID you're interested in. You will see a number of directories, links, and files (note that they take up no space). Two of these directories are useful for recovering files: **/proc/<pid>/exe**, and **/proc/<pid>/fd**.

The **exe** link is an actual pointer to the file that is being executed. The **fs** link is a directory of file descriptors currently opened or in-use by the process. Every process will have at least three, which are listed first and denote **STDIN**, **STDOUT**, and **STDERR**, respectively. Other possible entries are network sockets (e.g., **20 -> \socket:[450]**, or port 450) and files (e.g., **4 ->/home/kliston/.list.swp**).

In Linux, each inode keeps track of a file's link count, which is the number of times that a directory lists the inode. When a file is deleted, its entry is removed from the directory file and the inode's link count is decremented. If this link is reduced to 0, then the inode is marked as "free" in the inode bitmap, and all of the blocks referenced by that inode are marked as "free" in the block bitmap. The deletion time field is set in the inode. The OS also keeps track of the processes linked to an inode. This can be used to your advantage if you are notified of the accidental deletion in time.

Getting Your Files Back

This all may be interesting, but you still need to know how to get your files back. The first step is determining how important the information is, and how vital it is to get it back intact. In Linux, there are a few things you can try before mounting the affected partition in read-only mode.

If you need to recover an executable that happens to be currently running (such as in a forensics case where an intruder has a backdoor running, but has deleted it to cover his tracks), you can recover simply with:

```
cp /proc/415/exe /tmp/backdoor
```

If you have a process running that references a recently deleted file, you can try similar tricks with the **/proc/<pid>/fd** directory. In the example above, we had:

```
/proc/415/fd/4 -> /home/kliston/.list.swp
```

This happened to be the swap file from a vi session. Performing **strings 4** returned the contents of **/home/kliston/list** with some garbage as the header. Using the **/proc/<pid>/fd** technique will require some understanding of the applications to be fully successful. To list the files currently open on a system, use **lsof**, or for a quick and dirty method to generate a list of candidates for this technique:

```
ls -l [0-9]*/fd|grep <deleted_file>
```

If you're not lucky enough to have a case that can be solved by using the **/proc** recovery techniques, you need to cease write activity to the affected partition. Our examples will be recovering data from **/home** or **/dev/hdc6**.

Remount the partition in read-only mode:

```
mount -o ro,remount -n /home
```

This will allow you to access the system and stop processes from overwriting your to-be-recovered data blocks. The **-n** flag instructs **mount** to not write to **/etc/mtab**, enabling you to recover data from partitions that contain **/etc**, such as **/**.

There are a few factors that can be used to gauge your chances for success. Before kernel 2.2.x, the indirect inode pointers (pointers 13 and above) were also zeroed out when a file was deleted. If you are working with a kernel older than 2.2.0 (use **uname -r** to find out), you're limited to the file size that you can recover using a direct inode reference technique. This recoverable limit is 12*block size. You can pull the system's block size from the superblock by doing the following (where **/dev/hdc6** is an example file system):

```
echo stats|debugfs /dev/hdc6
```

These examples were performed on a system running kernel version 2.2.19-6. The file system had a block size of 4096 and 10 block groups. Files were recovered from the **/dev/hdc6** partition using **/home** as a mount point. The server saw low-to-moderate activity as a general-purpose server in a home/lab environment.

Using the **debugfs** utility, you can generate a list of deleted inodes, or inodes that have a non-zero time in their "Deleted Time" field. Generate a list of deleted inodes:

```
echo lsdel | /sbin/debugfs /dev/hdc6 > /tmp/lsdel.out
```

This generates an output similar to:

```
debugfs:   7344 deleted inodes found.

Inode   Owner Mode      Size   Blocks   Time deleted
62984   511   100600   12288   3/   3   Thu Dec 27  10:38:44 2001
62980   511   100644     693   1/   1   Thu Dec 27  10:39:09 2001
110212  511   100644 2123710   520/ 520 Thu Dec 27  10:54:35 2001
```

Needless to say, a lot of entries were omitted, and we've only shown the last three that belong to our user id since that's what we're interested in. To examine these files a bit more, use the **stat** command in **debugfs** to pull additional information about the file referenced by the inode:

```
debugfs /dev/hdc6
> stat <110212>
```

This will return the link count (probably 0), the creation, access, modify, and deletion times, and a list of all of the blocks that make up the file. This information will determine whether this inode is your candidate. To actually recover the data, use **debugfs** to dump the data to which the inode is pointing to a new file:

```
debugfs /dev/hdc6
dump <110212> /tmp/recovered
```

To recover all three of these files, edit **/tmp/lsdel.out** down to the desired files as **/tmp/lsdel.edited** and do something like this:

```
     awk '{print $1}' /tmp/lsdel.edited > /tmp/inodes
     for i in $(cat /tmp/inodes); do echo <$i> \
        -p /tmp/recovered.$u\i" | debugfs
/dev/hdc6; done
```

This creates a series of files in **/tmp**, but there is still the task of discovering their names and where to place them.

An alternative method (which is more risky but can work when you don't have another partition to restore to, and this is rarely the case) involves directly editing the inode itself. Zero-out the deletion date and create a link to the inode (both raising the link count to one, and

providing an access point in a directory):

```
debugfs -w /dev/hdc6
> mi <110212>
```

This action will walk us through the settings of the inode. It will show the current setting and offer to change it. Press "Return" to accept the current (or default) setting. When you arrive at the "Deleted Time" field, enter "0" and then continue accepting the rest of the settings. Then, change directory to where you want to link the file. Note that the top directory in **debugfs** will be the mount point, **/home** in our example:

```
> cd kliston/
> link <110212> recovered_file
```

It is important to **unmount** the altered partition and run **fsck** upon it. It will discover that there are blocks that are marked as free in the block allocation table, yet linked to an active inode. Let **fsck** make the required fixes. Now your file will remain safe, otherwise the data blocks will still be marked as available and eventually other files will reuse them and corrupt data.

It is simply a matter of chance should these techniques work. In test recoveries, we were able to help successfully recover log files on December 27th that had been deleted on October 11th. This was from a low-to-mid-use home/lab server, so these results are probably atypical.

Known-Text Recovery

What if the file wasn't **rm**ed? What if your unfortunate user typed:

```
cat /dev/null > important_file
```

In this case, the inode isn't deleted but all of the data block pointers are zeroed and the data blocks are freed up in the block allocation bitmap. The odds of recovery have just decreased by an order of magnitude, but there are some other options.

The "known-text recovery method" is more of an art than a science and is less likely to succeed, but it has the advantage of working on file systems other than ext2 (such as Solaris's **ufs**). This technique involves searching for a known pattern through an image of the affected file system. The pattern should be unique to the file that needs restoration. Crafting the search pattern is the artistic part of the process. A poorly written pattern can return too many hits, or no hits at all.

The example here involves recovering a DNS database file from the catastrophic **cat /dev/**

null > important_domain.com.db. Because we're looking for a bind data file, we could search on a pattern containing "IN SOA", or for a known host of the missing domain.

The first active step involved in this technique is the creation of the recovery copy of the partition. By this time, the partition should have already been unmounted, or mounted read-only (see above techniques). Copy the partition to another file system (which must be large enough to hold the affected partition) with a command such as:

```
dd if=/dev/hdc6 of=/opt/hdc6.image
```

Apply an **fgrep** filter to locate the pattern (a unique hostname, in this case) in the recovery image:

```
fgrep "elmenop" /opt/hdc6.image
```

Here, we're looking for the domain record that defined **elmenop.important_domain.com**. In the test case, this returned most of the domain record surrounded by nulls. It probably recovered unused space from a temporary file that referenced the file, rather than the file itself. If you need to search or use regular expressions, you can use **egrep** in lieu of **fgrep**, which will output all instances of your search pattern. Then, based on either knowledge, or trial and error, use **fgrep**'s **-A** and **-B** flags to pull a slice out of your recovery copy into (hopefully) an editable file that can be cleaned up for use.

The **-A** flag denotes how many lines after the match to print, and **-B** instructs how many lines before the match that **grep** will print. In the example, **elmenop** is a hostname that appears in the domain file. Using some guesswork (based on inspecting other domain data files that were not deleted), there is a window size of seven lines before, and ten lines after. There is added buffer room to our estimates to increase the odds of grabbing all of the usable data in one pass. In this special case, we lacked physical access to the server, and we didn't have enough space to create a recovery copy, so the action was performed on a live pattern (not recommended unless you're intentionally pushing your luck as we were):

```
dd if=/dev/hdc6 | fgrep -B 7 -A 10 --text "elmenop" > \
  /tmp/pattern_match.1
```

This approach created an editable output, capable of rebuilding the original file. This was successful after **cat /dev/null > important_domain.com.db** was used to "destroy" the file. The recovery attempt was made less than 24 hours later only to find that the data blocks had been overwritten. Once again, we find that time is not your friend when it comes to data recovery.

Recovery Tools

Are there programs out there to make this any easier? Absolutely. But, as sys admins, we know that you need at least three ways to fix a problem — none of them will work, but they'll give you an idea for a fourth way that probably will. Taking time to work through the abstraction of the operating system and understand what is happening at a lower level may help you see the problem differently. Tools tend to hide what is going on and may blind you to another answer. Realistically, working through the problem yourself is not always the most expeditious path. These tools may make administration a little easier for you:

The Coroner's Toolkit

(http://www.fish.com/tct) — A collection of tools originally created for computer forensics work. It includes the data recovery tools **unrm** and **lazarus**, both of which can be used to recover accidentally deleted data.

The Recover Tool

(http://recover.sourceforge.net) — Automates the direct inode recovery technique described above. It's good to use if you have a large number of files to recover.

Conclusion

In the end, retrieving a file on Linux comes down to luck, timing, luck, technique, and luck. Most file recovery tools are fairly inexpensive and easily available and should be a standard part of any systems administrator's toolbox. So, the next time a user accidentally deletes that vital file, you can say, "Relax, it's probably already too late. But maybe, just maybe, there's something I can do."

References

Ferlito, John and Widdowson, Liam. "Tales from the Abyss: UNIX File Recovery," *Sys Admin* magazine, November 2001:

Mandia, Kevin and Prossise, Chris. *Incident Response: Investigating Computer Crime*. Osborne/McGraw Hill, New York 2001.

Wall, Kurt, Watson, Mark, and Whitis, Mark. *Linux Programming Unleashed*. Sams, 1999.

Ext2 file Undeletion: http://www.billjonas.com/papers/undeletion.html

Crane, Aaron. Linux Ext2fs Undeletion mini-HOWTO: http://www.praeclarus.demon.co.uk/tech/e2-undel/howto.txt

Card, Remy, Ts'o,Theodore, and Tweedie, Stephen. Design and Implementation of the Second Extended Filesystem: http://e2fsprogs.sourceforge.net/ext2intro.htm

Oxman, Gadi. The extended-2 filesystem overview: http://www.nondot.org/sabre/os/files/FileSystems/Ext2fs-overview-0.1.pdf

Brian Buckeye is the Director of IT for a medium sized Ohio business. He can be reached at: **brian@blindpanic.com**.

Kevin Liston is a consulting security engineer. He can be reached at: **kliston@infornographic.com**.