CSE 451, Introduction to Operating Systems, Autumn 2013

Project 3 - Undelete

Assigned: Sunday November 17
Due: Wednesday December 4 at 11:59 p.m. (electronic submission)

Project introduction

This assignment aims to recover from the following common situation: you've just finished writing an essay/report/paper due in the next hour, but before you print it you accidentally manage to delete it. It would be very convenient to have a program for such situations capable of undeleting files, and that, in fact, is the goal of this project.

Implementations of undelete are very file system-type specific. We're going to write undelete for the ext2 file system, which was released about 20 years ago. The reason is that the file system layout in ext2 is very similar to that of its successors, ext3 and ext4, so it's pertinent to today's systems even if no one is using ext2 any longer. Undelete is not possible for ext3 and ext4 without looking through the filesystem journal, which is a more complicated task than we would like to assign to you.

This project leaves most everything up to you. The coding portion is relatively small; figuring out what to do is about half the assignment and getting it implemented is the other half.

First steps

Before you start, do some reading! Look at the bottom of this page and get up to speed on EXT2 and how kernel modules work. A few hours of reading will save headache later on and the overall organization of this project will make more sense.

Project files

To start the project, you'll need to update your Git repository to acquire our skeleton code. This is a fairly simple process; navigate to your local repository and then execute:

```
local$ git fetch course
From attu.cs.washington.edu:/cse/courses/cse451/13au/451repo
 * [new branch]      master     -> course/master

# Merge in the code
local$ git merge course/master

# Now push your locally added changes into the shared repository
local$ git push origin master
```

The provided code includes:

- undelete.c: This is the core kernel module code. It does all the set up work for you to register a character device and get the super blocks of mounted file systems. This is where you will likely put the bulk of your undelete code, depending on whether you decide to modify the kernel.
- tar_utils.c/h: A routine for writing the tar header block for a file to a buffer.
- Makefile: The file for building the kernel module.
- filesysFile-simple and filesysFile-easy: These are two sample ext2 filesystems on which you can start testing your code. You will definitely need to test more complex filesystems than these.

- mkFilesysFile.sh: This is a script for creating and mounting ext2 filesystems. It takes two parameters: a file name and the number of 1kiB blocks to use. You'll need to specify at least 60 blocks for it to succeed. Furthermore, you will need root access to mount a filesystem, so you'll need to use either a personal machine, the course virtual machine image, or preferrably Qemu.

Your task

You will be creating a kernel module to recover deleted files in ext2 filesystems. In ext2, when a file is deleted, the file's inode and data blocks are marked as free, its directory entry is invalidated, the deleted time of the file's inode is set, and the number of hardlinks in the inode is set to 0. The key thing here, though, is that the file's inode and data blocks are still preserved, at least until the filesystem ends up reclaiming them. You will write a kernel module in C called `undelete` that is capable of recovering these files as follows:

- The `undelete` kernel module will register a character device for each mounted ext2 filesystem.
- These character devices appear as read-only files under `/dev/undelete/XXX`, and when read, they return a GNU tape archive (tar) format archive containing all undeleted files in the filesystem represented by the `/dev/undelete/XXX` file.
- Each recovered file inside the archive should be named `file-nnnnn`, where `nnnnn` is the inode number of the deleted file.
- Restore each file's mode, uid (user ID), gid (group ID), modified time, and contents. You won't have information on the user name and group name for recovered files, so it's fine to set them to any values you like, such as "root". As an example, suppose that a filesystem named `sample-filesystem` has had two files named `sample1.txt` and `sample2.txt` with inode numbers 12 and 117 deleted from it. From a shell in userspace, executing `cat /dev/undelete/sample-filesystem > sample-filesystem.tar` should produce an archive that contains files named `file-00012` and `file-00117` with the same mode, uid, gid, modified time, and contents of `sample1.txt` and `sample2.txt`, respectively.
- For deleted inodes that do not correspond to files, take no action.
- Restore only files that have been deleted. You *will* need to make use of the inode bitmap for this.
- There is a possibility that part of a deleted file has been reclaimed for use by something else. The goal of this project is not to build an extremely robust file recovery solution, so you don't need to worry about handling this gracefully.
- You will need to handle multiple level of data block indirection as well as multiple block groups. The provided filesystems do not exercise these features of ext2 at all, so it is up to you to create sample filesystems that do.
- Do not seek unnecessarily through the filesystem file. In particular, you should not need to visit every single inode if there are many non-deleted files and only one deleted file.
- You may not keep an unbounded number of inodes and data blocks in memory at once (or any structure for that matter). In particular, you should be able to restore a 2 GiB filesystem with about 2 GiB worth of deleted files while using the same (small) amount of memory as if you were restoring a 10 MiB filesystem. You have some flexibility here; just make sure that the number of inodes and data blocks kept in memory is small relative to the size of the filesystems.
- You may not steal code from existing file recovery tools. Most build on existing frameworks, which isn't going to help you anyway, so just don't do it.

Synchronization

Surfacing a tar file containing the recovered files to userspace is unfortunately not as simple as writing them to a file in a particular format. After `open()`ing a character device, a userspace process will proceed to invoke `read()` repeatedly on it until reaching the end of the file, at which point it will `close()` the device.

One potential approach is to run the undelete operation as part of `open()` and buffer the contents of the tar file so that they can later be returned via calls to `read()`, but the problem with this is that a filesystem with, say, a 2GB deleted file would require more than 2GB of memory for the undelete process, which is far too much. Instead, we model the character device as an iterator; a call to `open()` sets up some state, and calls to `read()` consume data, advancing a state machine forward that tracks where in the file system the undelete operation is.

For each recoverable file, the undelete operation should:

- Write a 512-byte tar file header to the shared buffer. A function is provided in the skeleton code that does this for you given some basic file attributes.
- Write the file data to the shared data buffer in sequential 512-byte data blocks. The final block should be padded to fit all 512 bytes.
- When the header and data for all recoverable files has been written, the undelete operation should write two empty 512-byte data blocks to the shared data buffer to indicate the end of the archive, then exit.

Traversing the ext2 file system

The kernel module skeleton code that is provided to you acquires a reference to the `super_block` struct (see `include/linux/fs.h`) for each ext2 file system. These structs provide a high-level description of the file system that they represent, though nothing in them is specific to ext2. The Linux kernel uses a virtual file system abstraction in which each file system surfaces a common API for opening, reading, writing, creating inodes, etc. even though the implementations differs between them. In ext2, the `super_operations` struct in `include/linux/fs.h` related to modifications to the entire file system are implemented in `fs/ext2/super.c`, for instance, which registers handlers for each function in its `ext2_sops` struct.

To traverse the ext2 file system to look for deleted files, you have a couple options. One is to add an "`undelete`" super operation to the kernel and then implement it within ext2. The benefit of this is that you will have full access to all of the ext2-specific functions in `fs/ext2/ext2.h`, while the downside is that you'll actually have to modify the kernel to make any changes in functionality. In the Linux kernel, only functions that have been exposed using the `EXPORT_SYMBOL` macro can be called from kernel modules like the one that you are writing, which means that ext2-specific functions such as the ones defined in `fs/ext2/ext2.h` are not accessible; you can only directly invoke the non ext2-specific ones declared in `include/linux/fs.h` from the kernel module.

The other (preferred) option is to figure out a way to interact with the ext2 file system as if it were a file. The exact layout of the ext2 file system on disk is well-documented, so assuming that you are able to figure out how to open and read from this file system file, you can actually implement all of the undelete tool inside the kernel module. The upside is that you won't need to make any changes to the kernel (so design iterations will be much faster), but the downside is that you won't have access to all of the nice functions in `fs/ext2/ext2.h`, just the definitions of the structs. As an example of how this sort of IO works, take a look at the `ext2_get_inode` function, which uses `sb_bread` to read from a particular offset in the file system file.

Using the kernel module

After building the kernel module `ext2undelete.ko`, you can insert it into the kernel of a running kernel (such as the course VM or Qemu) via:

```
$ sudo insmod ext2undelete.ko
```

This will automatically create entries for all mounted file systems under `/dev/undelete/` such as `/dev/undelete/loop0`, for instance. To unload the kernel module, use:

```
$ sudo rmmod ext2undelete.ko
```

This will remove the `/dev/undelete` entries and deactivate the kernel module.

Using the filesystem creation tool

The filesystem creation tool provided with the starter code above is fairly simply, and you are free to customize it as you like. Here is what it contains:

```
#!/bin/bash

if [ $# -ne 2 ]; then
  echo "Usage: $0 [filename] [size-in-1kB-blocks]"
  exit 1
fi

dd if=/dev/zero bs=1024 count=$2 of=$1 && \
  mkfs.ext2 -F -b 1024 -c $1 && tune2fs -c0 -i0 $1 \
  && mkdir -p mnt && sudo mount -o loop $1 mnt

echo

if [ $? -eq 0 ]; then
  echo "Mount completed successfully! Use 'sudo umount mnt' to unmount"
else
  echo "Mount failed...please try again."
fi
```

- The `dd` command copies data from a device (in this case `/dev/zero`, which outputs 0s) to an output file using a

given number of blocks of a certain size.
- The `mkfs.ext2` command creates an ext2 filesystem in the given file with blocks of the given size (1024 bytes).
- The `tune2fs` command prevents the OS from running `fsck` on the filesystem automatically, since that could modify the filesystem in ways you don't expect.
- The `mkdir` command creates a mount point for the filesystem.
- The `mount` command mounts the filesystem using `mnt` as a mount point. Use this command by itself to mount an already-created filesystem.

`mount` requires root privilege, so you will only be able to run this on your own machine, on a VM image, or using Qemu.

When you create a filesystem and mount it, you will not necessarily see the changes to the filesystem file until you unmount the filesystem. For example, let's say that you create and mount a filesystem file named `test`. You then create a file on it through `echo "Some file contents" > mnt/sample.txt` and delete the file with `rm mnt/sample.txt`. Before you run your undelete tool to try to recover the file, first run `sudo umount mnt` to flush the changes to disk. To mount the filesystem again, use `sudo mount -o loop test mnt`. If you don't unmount the filesystem, you won't necessarily see the changes reflected in it.

Debugging

An important part of this project is debugging. You're going to run into off-by-one errors--it's pretty much guaranteed, given that block and inode numbering is kind of funny. If it doesn't seem like you're reading the right data from the filesystem file for the block or inode number you are using, there are a couple different tools to help you. First of all, make liberal use of debug printing. To make a switch for enabling or disabling printing of debug statements, add some code like this near the top of your source file:

```
#define CSE451_DEBUG
#ifdef CSE451_DEBUG
#define LOG_PRINTK(...) printk(__VA_ARGS__)
#else
#define
LOG_PRINTK(...)
#endif
```

Any macro name is fine, really, as long as you don't override any existing or commonly-used macros. Use `LOG_PRINTK` just as you would `printk`. To disable debug output, simply remove the `#define CSE451_DEBUG` line.

After setting up debug printing through something like the above, when you go to read something from the filesystem, print out its absolute offset within the file as a hexadecimal value (e.g. 0x4000). If what you read at that point doesn't match your expectations, open the filesystem file using a hexeditor such as `hexedit`. Look at the data around that location. Was the data close by? Try to identify whether you were off by one index or whether you were looking in the wrong place entirely.

There is a tool called `dumpe2fs` that will be very useful to you as well (see `man dumpe2fs`. It prints a variety of information about offsets, which inodes are free, etc. and you can (and should) use this for figuring out what the expected values at various points in your undelete utility are.

When you want to test recovery of large files, try using images. If you are able to recover a deleted image file, it should appear as before, and you should quickly be able to tell whether any of the data was corrupted.

Tips and Resources

Tips

- If working in the kernel and undeleting at the same time is intimidating, try getting undelete to work in userspace using a file system image file. If your core undelete functions are architected well, switching to kernel space shouldn't require many changes.
- `cmp`, `sum`, `md5sum`, and `shasum` can be used to quickly check if a recovered file is a binary match to its original.
- Consider creating a workflow where a single command automatically runs undelete and checks the output is as expected using one of the above programs. This can be done e.g. in make or a bash script and can save quite a bit of time.
- Don't forget that when working in kernel space you need to verify user space buffer pointers before writing to them.

tar format

USTAR. We are using the POSIX tar format, you can read about it there.

ext2 references

There is a wealth of information about the layout of ext2 filesystems online. In order of helpfulness, I would recommend reading the following pages:

- http://www.nongnu.org/ext2-doc/ext2.html: If you were going to use only one reference for this project, it would be this one. It covers all attributes of ext2 that you need to know.
- http://www.tldp.org/LDP/tlk/fs/filesystem.html: A description of ext2 from a book about the Linux kernel from quite some time ago.
- http://homepage.smc.edu/morgan_david/cs40/analyze-ext2.htm: Somewhat less technical than the above two sources and walks through some helpful examples.

To learn more about device drivers, take a look at the first three chapters of the Linux Device Drivers book.

Submission instructions

Submit an archive named `netid1-netid2.tar.gz` to the Catalyst DropBox for the class that contains:

- All of your source files, including any from the Linux kernel that you modified (depending on your undelete implementation).
- A file named README with a high-level description of how your undelete implementation works. If you made changes to the kernel, list each file that you modified and why made the changes that you did.