Accessing Files on Unmounted File Systems

Willem A. (Vlakkies) Schreüder – University of Colorado, Boulder

ABSTRACT

This paper describes a utility named `ruf` that reads files from an unmounted file system. The files are accessed by reading disk structures directly so the program is peculiar to the specific file system employed. The current implementation supports the *BSD FFS, SunOS/Solaris UFS, HP-UX HFS, and Linux ext2fs file systems. All these file systems derive from the original FFS, but have peculiar differences in their specific implementations.

The utility can read files from a damaged file system. Since the utility attempts to read only those structures it requires, damaged areas of the disk can be avoided. Files can be accessed by their inode number alone, bypassing damage to structures above it in the directory hierarchy.

The functions of the utility is available in a library named `libruf`. The utility and library is available under the BSD license.

Introduction

There are many important reasons for being able to access unmounted file systems, the prime example being a damaged disk. This paper describes a utility that can be used to read a disk file without mounting the file system. The utility behaves similar to the regular `cat` utility, and was originally named `dog`, but was renamed to `ruf` for **r**eading **u**nmounted **f**ilesystems to avoid a name conflict with an older utility.

In order to access an unmounted file system, the utility must read the disk structures directly and perform all the tasks normally performed by the operating system; this requires a detailed understanding of how the file system is implemented. Implementing this utility for a particular file system is an interesting academic exercise and a good way to learn about the file system. The original work on this utility was in fact done in Evi Nemeth's system administration class.

Boot Block	Cylinder Group 1	Cylinder Group 2	. . .	Cylinder Group *m*
1	*k blocks*	*k blocks*		*k blocks*

Figure 1: File system layout.

Most Unix systems – including HP-UX, Solaris, Linux and the BSD family – use a general purpose file system derived from the Fast File System (FFS). [McKusick, et al., 1984] While the Linux ext2fs implementation differs considerably from the FFS, the concepts it uses are basically the same.

Figure 1 shows the layout of the file system. The file system, which may be all or part a physical disk, consists of a boot block followed by a number of equal sized entities called cylinder groups. The physical data blocks that make up a cylinder group are organized such that they can be read with minimal movement of the disk heads, thus improving performance. In fact, the FFS is itself derived from the original System V File System (S5FS); the major difference between the FFS and S5FS is that the S5FS has a single cylinder group. Cylinder groups are relatively small, often tens of megabytes in size. Contemporary file systems may therefore contain hundreds or thousands of cylinder groups.

All disk drives organize data into fixed sized physical blocks. The file system uses blocks that are one or more physical disk blocks. The terminology of the FFS is a bit confusing, in that a block consists of several fragments or frags. A physical disk block may therefore really correspond to a fragment and not to a block. For example, a 4 kB block may consists of four 1 kB fragments. The block address actually refers to the fragment address, so that in this example with four fragments per block, the addresses of the blocks are 0, 4, 8, etc., so that each address readily decodes to the correct block or fragment.

Super Block	Accounting Information	Inode Table	Data Blocks
1	*k blocks*	*l blocks*	*n blocks*

Figure 2: Cylinder group layout.

The layout of a cylinder group is shown in Figure 2. The superblock contains many critical file system parameters such as the block size, number of fragments per block, and other information needed to access the file system. Since these parameters are critical to the integrity of the data on the disk, several backup copies of the superblock are spread over the disk, potentially one per cylinder group.

Most information about a file is saved in a structure called an inode. This information includes the file owner, permissions, times, size and so on. Inodes are of fixed size and are numbered sequentially. Given the inode number, a somewhat complicated algorithm gives the exact location of the inode on the disk.

The file contents are stored in data blocks. A list of the data blocks associated with a file is stored in the inode. Reading a file involves reading the inode, extracting the list of data blocks, then reading the actual data in the blocks.

The typical tree structured directory system is implemented as a sequence of files. A directory is nothing more than a special file that associates file names with inode numbers. To read a file by its file name, the root file that appears at a fixed inode number is read. Sub-directories appear as records in the file with associated inodes. Marching the directory tree requires reading sub-directories until the specific file and the associated inode number are found.

Program Organization

The `ruf` program is organized into three sets of functions. At the lowest level, there are the device functions: These functions read bytes from the disk at boundaries and in sizes appropriate for the device being accessed. They also buffer data to improve performance. While important to the operation of the program, this paper simply asserts the ability to read arbitrary byte offsets on the disk using a 64 bit address.

The next set of functions are the file system functions, where the majority of work is performed. At this level, peculiarities such as disk, inode and directory structures are resolved. The functions `fsmount` and `fsumount` read the superblock and perform memory management functions, analogous to the `mount` and `umount` system calls. Alternate superblocks may be read with appropriate command-line parameters. The functions `fsopen`, `fsread`, and `fsclose` are used to open, read and close a file based on an inode number passed to `fsopen`. The function `fsscandir` is implemented to read directory files using `fsopen`, `fsread` and `fsclose`.

The last set of functions take care of the walking the directory structure, resolving symbolic links and listing the file or directory. The function `walkpath` takes a path and finds the corresponding inode number by reading each of the directories in the path. Symbolic links causes `walkpath` to be called recursively. The function `stream` lists the contents of a regular file to standard out, or performs a directory listing looking like `ls -il` to standard out. Finally, a function `ruf` provides a convenient interface to all the lower level functions.

The functions are implemented in a library named `libruf`. A simple main program calls functions in `libruf`, but the functionality may be readily incorporated into other programs.

Porting `libruf` to a new file system derived from the FFS should only require specifying which header files to include and definition of a few macros in the files fs.h and fs.c.

The Superblock

The first step in accessing the file system is reading the superblock. This is done through the function `fsmount`, which returns a private structure that is used in subsequent operations.

The primary superblock is stored near the beginning of the file system. At the very beginning of the file system is typically 1 kB of boot information. The superblock is at a fixed offset from the beginning of the file system, typically 1 kB. When using 1 kB blocks, the boot information is contained in the first block, and the superblock appears in the second block. Larger blocks contain both the boot information and superblock. Reading the primary superblock simply requires seeking to the superblock offset and reading the required number of bytes.

Reading an alternate superblock is somewhat of a chicken and egg problem. The alternate superblocks appear at the beginning of some subsequent cylinder groups. However, in order to determine the cylinder group size, the information in the superblock is required. Fortunately, cylinder groups are not of arbitrary size. One of the important components of the cylinder group is a set of bits indicating used blocks. This set is stored as a single block. Since a 1 kB block contains 8192 bits, the cylinder group for this block size will often contain 8192 blocks.

In order to read an alternate superblock, it may be necessary to try a few of the common block sizes in order to find the superblock. The function `sbfind` reads all the blocks on the disk and prints fragment addresses where the superblock magic appears at the correct offset. If the root superblock is readable, the function `sblist` can be used to read it and then test all the appropriate locations and print the fragment addresses where superblocks occur.

In older implementations, every cylinder group contained a copy of the superblock. With the explosive growth in disk sizes, the number of cylinder groups have grown so large that newer implementations store only a dozen or so copies of the superblock.

Much of the superblock information is intended to specify parameters that optimize disk performance and facilitating writing to the file system. The number of parameters required to read the file system is actually a very small subset of the values in the superblock. The number of bytes per block, number of bytes per fragment and number of fragments per block are critical parameters. Furthermore, the number of inodes per cylinder group is required to determine which cylinder group contains the inode. Finally the number of blocks per cylinder group determines the cylinder group boundaries. These parameters are saved as part of the private structure returned by `fsmount`. This structure must be passed to all functions that read the file system. When done, the program should call `fsumount`. This function frees allocated memory and closes the device file used to read the file system.

Locating the Inode

The most difficult part of reading a file is locating the appropriate inode. File systems directly derived from the original FFS define convenient macros to aid in this process. Others such as the ext2fs do not.

All cylinder groups contain the same number of inodes as defined in the superblock. Call this *ipg* for inodes per group. The cylinder group of an inode for the ext2fs is then $(inode - 1)/ipg$, while for the others it is $inode/ipg$. The offset of the inode index within the cylinder group is $(inode - 1)\%ipg$ for the ext2fs, and $inode\%ipg$ for the others.

The inodes are stored sequentially in the cylinder group, as shown in Figure 2, in the block labeled "inode table." In file systems other than the ext2fs, the inode table starts at an offset defined by the cgimin macro. This offset is actually stored in the superblock.

For the ext2fs, there is no convenient macro or superblock parameter. Figure 3 illustrates the layout of an ext2fs cylinder group in detail. In order to determine the offset of the inode table, the number of blocks consumed by the other structures must be calculated. The superblock is offset *sboff* bytes from the start of each cylinder group, so if the offset superblock does not fit into a single block, it consumes more than one block. For a block size of *bs*, the superblock structure *sb* will therefore consume:
$(sboff + sizeof(sb) - 1)/bs + 1$ blocks.

A set of group descriptors, one for each cylinder group, occupies the next *k* data blocks. The number of cylinder groups is calculated from the total number of inodes on the disk *inod_count* as $inod\_count/ipg$. The number of blocks *k* is therefore given by:

$$\frac{((inod\_count/ipg - 1) * sizeof(ext2\_group\_desc))}{bs} + 1$$

where $ext2\_group\_desc$ is the group descriptor structure.

The next two data blocks are bitmaps for the data blocks and inodes, respectively. Therefore, the block offset at which the inode table is

$$\frac{((inod\_count/ipg - 1) * sizeof(ext2\_group\_desc))}{bs} + \frac{(sboff + sizeof(sb) - 1)}{bs} + 4.$$

Once the offset of the inode table is found, the offset of the inode of interest is simply the size of an inode times the number of intervening inodes. The inode is read by seeking to the appropriate location and reading the appropriate number of bytes, usually 128.

Reading the file

The inode contains all information about the file except the file name and actual file data. This includes the file attributes such as user and group identifiers, permissions, times, file type and size. The `fsopen` function locates and reads the inode and allocates a private data structure for the file. This structure is passed to subsequent calls which read the file. When done with the file the `fsclose` function is used to deallocate the structure.

Sequentially reading the file with `fsread` requires reading the data blocks of the file in order. The inode contains a list of (typically) the first twelve data blocks known as direct blocks. For small files, all data blocks can be addressed in this way.

There are typically three slots for indirect blocks. These slots are used for single, double and triple indirection. The single indirect block is a data block containing a list of pointers to data blocks. The double indirect block points to a data block containing pointers to indirect blocks. Each of these indirect blocks in turn points to data blocks. For even bigger files, triple indirection is used. Table 1 shows the maximum file size that can be addressed using different block sizes.

Block Size	Direct	Single Indirect	Double Indirect	Triple Indirect
512	6 kB	70 kB	8 MB	1 GB
1024	12 kB	268 KB	64 MB	16 GB
2048	24 kB	1 MB	513 MB	256 GB
4096	48 kB	4 MB	4 GB	4 TB
8192	96 kB	16 MB	32 GB	64 TB

Table 1: Maximum file sizes by block size.

The purpose of using both fragments and blocks is to optimize disk usage. For example, on HFS systems, the block size is typically 8 kB, consisting of eight 1 kB fragments. The block addresses are therefore multiples of 8. Hence, for large files consisting of mostly blocks, eight times fewer addresses are required than if fragments were addressed, thus decreasing the indirect block overhead. The very last block of the data typically does not fill a data block. Therefore the last data block address typically addresses only a fragment or sequence of fragments. The fragments are contiguous, so it is simply

Super Block	Group Descriptors	Data Block Bitmap	inode Bitmap	inode Table	Data Blocks
1+ blocks	*k blocks*	*1 block*	*1 block*	*n blocks*	*m blocks*

Figure 3: ext2fs cylinder group layout.

necessary to read the number of bytes remaining in the file from that fragment address in order to finish reading the file.

Data block addresses are absolute, so that a file may use blocks from more than one cylinder group if necessary. A block address of zero has special meaning. When this address appears in a direct or indirect block, it implies a block with all zeroes. This mechanism conveniently handles files with holes. In the read functions, this address is conveniently handled by a simple test that returns a data block with all zeroes instead of performing an actual disk read.

In current implementations, the ext2fs has only one fragment per block. Therefore, although it does store fragment parameters in the superblock, fragments and blocks are in fact the same.

Finding the File

A directory is simply a file with a defined structure. The original S5FS allowed only file names of 14 characters, so that each directory entry could be stored as a fixed-length record; file systems with this limitation are rare today. A typical directory is made up of a sequence of variable-length records; each record consists of an inode number, record length, name length and the name itself. Some file systems such as FFS and later versions of the ext2fs also store the file type in the directory record, duplicating the information stored in the inode. A record with an inode of zero is a placeholder and is skipped. The record length will often be greater than necessary, as deleted directory entries are subsumed by extending the previous directory entry.

The function `fsscandir` is used to read the directory. The directory is identified by its inode number; when a directory entry to find is specified, the inode number of that entry is returned. If no entry is specified, a directory listing is produced instead.

File names, as stored in directories, do not contain path information; instead, only the base file name is stored. For example, in the /etc directory the file /etc/passwd will simply be named passwd.

All files are accessed through the root directory, which is always at a known inode number. The function `walkpath` parses the path into its components, and reads each directory in turn with `fsscandir` to find the inode of the next entry in the path. This entry is associated with an inode number, denoting the subdirectory, symbolic link or the final file to be read.

When a file system is not mounted, the path names are relative to the root of the file system. For example, when the /usr tree is in a separate file system, the file /usr/bin/gcc will appear as /bin/gcc in this file system.

Links and Devices

On UNIX systems, everything is a file. Devices are represented as files with a special file type. The kernel addresses these files using a major and minor number stored in the inode, and there are no associated data blocks. The `ruf` program simply reports these special files as device files.

Hard links exist when more than one entry to an inode exists in the directory files. Hard links require no special treatment when reading a file. Symbolic links, also known as soft links, however, are special files. Instead of containing actual file data, the symbolic link references another file or directory. A symbolic link may point to a file not on the disk in question. When reading a directory tree, encountering a symbolic link requires that the current leaf in the file path be replaced by a new, arbitrary path. This is achieved by recursively calling `walkpath`. The explicit occurrence of the .. file name in each directory allows resolution of the inodes in linear progression.

Symbolic links are a significant overhead in resolving path names; to improve performance, short symbolic links can be stored in the inode itself, where the data blocks would be stored (since no data blocks are needed). Typical file system implementations allow symbolic links of up to 60 bytes to be stored this way; symbolic links that exceed this length is stored in a data block. Strangely, while both UFS and HFS suggests that this can be done, all symbolic link names are stored in a data block, regardless of the length of the symbolic link.

Usage

The `ruf` utility behaves much like the regular `cat` utility, except that it takes the device name as the first argument before the file names. For example, on an HP-UX system with the root system on c0t6d0, the file /etc/fstab can be read using either

```
ruf /dev/dsk/c0t6d0 /etc/fstab
```

or

```
ruf /dev/rdsk/c0t6d0 /etc/fstab
```

Note that both the character and block devices can be used. On a Linux system with /dev/hda8 containing /var, the file /var/log/messages is read using

```
ruf /dev/hda8 /log/messages
```

Alternately a directory listing of the /var/log directory can be obtained using

```
ruf /dev/hda8 /log
```

Assuming this listing shows messages to have inode number 13294, the file can be read using

```
ruf /dev/hda8 13294
```

An integer file name is assumed to be an inode number.

This particular file systems uses 4 kB blocks. The same file can be read using the alternate superblock at fragment 98304 using

```
ruf -s98304:4096 /dev/hda8 /log/messages
```

or

```
ruf -s98304:4096 /dev/hda8 13294
```

The values of important file system parameters such as the block and frag size, blocks, fragments and inodes per group and number of cylinder groups can be read by omitting the file name, for example

```
ruf /dev/hda8
```

When the root superblock is readable, the fragments of all the alternate superblock locations can be listed using

```
ruf -l /dev/hda8
```

If the root superblock is not readable, the entire disk can be searched for potential superblocks using

```
ruf -f4096 /dev/hda8
```

Conclusions

The `ruf` utility is useful for accessing unmounted file systems. It is also very instructive in learning implementation details of various file systems. The functionality can be embedded in other programs through the `libruf` library. All the software is available under the BSD license at http://www.netperls.com/ruf.

About the Author

Vlakkies Schreüder holds a Ph.D in computational fluid dynamics and is currently a senior engineer with Principia Mathematica where he works on solving practical problems in fluid flows. He is also working on a Ph.D in parallel systems at the University of Colorado, Boulder. Reach him at vlakkies@colorado.edu.

Acknowledgments

I want to thank Evi Nemeth for putting me up to this in the first place, and Adam Moskowitz for suggesting many improvements to the paper.

References

Bach, M. J., *The Design of the Unix Operating System,* Prentice-Hall Software Series, Englewood Cliffs, NJ, 1986.

Bovet, D. P., and M. Cesati, *Understanding the Linux Kernel,* O'Reilly, Sebastopol, CA, 2001,

McKusick, M., W. Joy, S. Leffler, and R. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, Vol. 2, Num. 3, pp. 181-197, August, 1984.