Issues in On-line Backup

Steve Shumway – SunSoft, Inc.

ABSTRACT

Administrators of UNIX systems have long been cautioned to run their backups only while in single-user mode. The specific reasons for this practice have often been shrouded in mystery. Administrators have been forced to make trade-offs between system availability and backup reliability without knowing all the facts. Nevertheless, the majority of administrators opt for the high-availability solution of running their backups on-line, on ''live'' file systems. We present a catalog of many of the dangers of this practice and a method for constructing backup programs that work reliably when run on active file systems.

Backup program functionality can be broken into two phases: a **scan phase**, where the files to be backed up are identified, and a **copy phase** (or **dump phase**), in which the files are copied to output media (e.g., dump tapes). Backup programs are most vulnerable to data loss while executing their scan phases, during which they expect to see a consistent file system image. Other problems can occur if files are deleted, moved, truncated or modified during the dump phase.

Construction of a backup program that runs safely on active file systems requires kernel (i.e., file system) support to ''lock'' the file system at critical points during the program's execution. Programs that use on-media data for synchronization (such as *dump*(8) and *restore*(8)) must also take defensive measures to ensure that file system modifications during the dump phase do not result in loss of synchronization that might render all or part of the output media unusable.

Introduction

It is the unfortunate reality of systems administration that files do not always hang around as long as we would like. Whether due to hardware failure or user error, files sometimes disappear when not intended. It is for this reason that we do regular backups (you do run regular backups, right?).

Another fact of life is that it is really hard for administrators to keep their users happy. Users want their systems up all the time (or at least when they want to use them!) and want all their files kept around (and recoverable) forever. Unfortunately, most backup programs do not produce reliable results when run on active file systems, dictating some sort of compromise. This paper is going to explore some of the trade-offs between system availability and backup reliability for different backup schemes and present some suggestions as to how to maximize reliability and availability at the same time.

First, we will explore a taxonomy of backup solutions, including standard approaches based on *dump*(8), *tar*(1), and *cpio*(1). Next, we will look at some of the problems associated with performing backups on live file systems using each approach. Finally, we will explore modifications to those approaches that allow one to perform more reliable backups with minimal impact on system availability.

Backup Taxonomy

In order to understand what can go wrong when backup programs are run on active file systems, we must first understand how these programs work. We can categorize backup programs according to the way in which they access the data to be copied and by their execution profile, i.e., the number of passes they make across the file system during execution.

Access Method

The first classification system is based on the way in which a backup program accesses the target file system. Backup programs can be categorized as using either a **device based** (or **inode based**) or **file system based** access method.

In a device based paradigm, the program accesses file system data by reading directly from the underlying device. This method has the advantage of speed but the disadvantages of coding complexity, limited portability, and data inconsistency. The *dump* program supplied with Berkeley-derived UNIX systems uses this access method.

A device based method requires complete knowledge of how data is organized on disk, i.e., what an inode looks like, how inodes are distributed, how data blocks are allocated, etc. Such programs use very low-level system calls, primarily `lseek` and `read`. Potentially, this makes them very fast but restricts their operation to a particular file system type.

Programs using this method are also prone to various data inconsistencies caused by the operating system's buffering of I/O operations. Since the kernel maximizes I/O performance by scheduling operations asynchronously, the underlying device does not necessarily reflect the true state of the file system at any given point in time. This can produce stale data in the output file, leading to anything from corrupt to missing files.

On the other hand, programs that use the file system interface are generally simpler but slower than those using the underlying device. The advantages of this method include reduced complexity, enhanced portability, and better data consistency. The *tar* and *cpio* programs access files through the file system interface.

Rather than carry extensive knowledge of file system structure, such programs are implemented using loops consisting of calls to `opendir`, `readdir`, `stat` (or `lstat`), `open`, and `read`. Programs using this interface are often recursive.

Because file system semantics are fairly well-defined and consistent across multiple platforms, programs that use this interface are more portable than those using the underlying device. Theoretically, one could write a program that was equally adept at backing up both local and remote (via NFS, say) files.

Programs using the file system interface also ''see'' a more consistent representation of the data because this interface presents a view using the kernel's buffer pool. Changes made to the file system are immediately visible to programs accessing data through the file system interface.

Execution Profile

In the most simplistic sense, backup programs operate in two phases, scanning the file system identifying files to back up then copying those files to backup media. The first task we call the **scan phase** and the second we call the **copy phase** or **dump phase**. In addition to identifying files to copy, the scan phase is usually responsible for obtaining a snapshot of the file system name space. Potentially, the program uses this information during the dump phase; it is certainly required during file recovery.

Backup programs can be categorized by the number of passes they take to implement these phases. Programs such as *tar* or *cpio* are intrinsically single-pass in nature and exhibit **simultaneous scan and dump** behavior. Programs implemented in this manner make a single pass through the file system and immediately copy files that successfully match the criteria for dumping. The obvious benefit of this method is a potential reduction in the amount of I/O. This paradigm is not without its disadvantages, some of which will be explained in a moment.

Programs such as *dump* make multiple passes through the target file system and exhibit **sequential scan and dump** behavior. In its scan phase, *dump* makes a pass across all files in the file system, accomplishing two things: identifying files for dumping (based on whether their modification time is more recent than the time of a previous level's dump) and taking a snapshot of the directory structure. After completing the scan phase, *dump* copies all appropriate files to the output file in inode order, directories first. *Dump* exhibits slightly hybrid behavior in that the file system name space is both collected and dumped in a single step (during Pass three, in which the directories are dumped).

Note that the execution profile is independent of the method used to access the file system. For instance, *dump*, a multiple-pass program, reads from the raw disk but it might just as easily have been written to access data through the file system interface.

Pitfalls of On-line Backup

The primary requirement of any backup program is that it copy files such that they can be retrieved later in the event of loss. Unfortunately, most current offerings do not satisfy this requirement when run on mounted, active file systems. The problems one can encounter depend somewhat on the implementation of the program in question.

File Movement

By far, the most common and serious threat to backup reliability is file movement during the backup. In particular, directory movement at inopportune times can lead to the complete omission of large sections of the file system hierarchy from the backup. Of course, the converse is also possible, in which portions of the file system tree already scanned or copied are revisited. Programs using combined scan and dump algorithms and file system access (such as *tar* or *cpio*) are most vulnerable, but all backup programs face this problem to some extent.

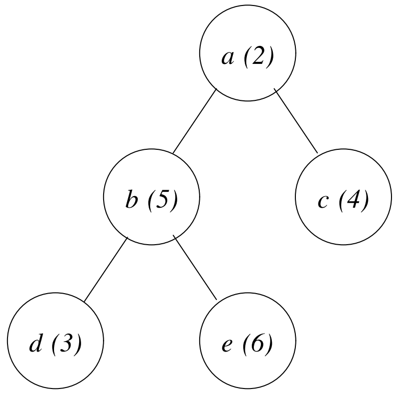

Figure 1: Initial file system state

Figure 1 presents a simple file system tree, with inode numbers shown in parenthesis (we will use them later). For the moment, let us assume that *a*'s directory entries are ordered such that *b* precedes *c*. A program such as *tar* using the file system access method would then traverse this file system in the following order: *a*, *b*, *d*, *e*, then *c*.

Suppose we execute the command

```
% mv /a/b /a/c
```

(the result of which is shown in Figure 2) before our backup program gets a chance to traverse directory *c*. The hierarchy rooted at *b* will be revisited when directory *c*'s sub-directories are traversed. If this occurs during the scan phase, the file will appear to exist in two places in the file system. If this happens during the dump phase, the program's output may very well contain a second copy of the files.

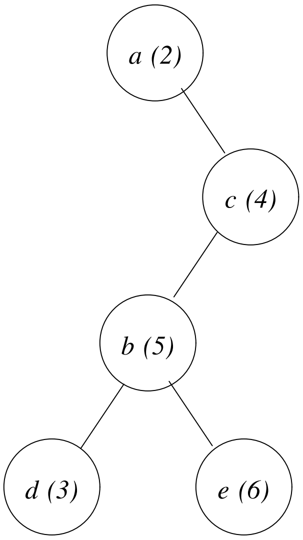

Figure 2: After renaming */a/b* to */a/c/b*

Although, this is relatively benign, it can lead to more serious errors depending on how the program is written. In particular, programs using depth-first file system traversal and returning to the parent directory via the ``..'' entry can lose their place in the file system if traversing a section of the hierarchy in the process of being moved. Fortunately, *tar* is recursive and uses the stack to avoid this particular situation.

Returning to Figure 1, let us now assume that *a*'s entries are ordered *c*, then *b* and execute the same command. The file system looks the same to the user but the results are disastrously different. If the command is executed just after completing the scan of directory *c*, the backup program will never see directory *b* and the hierarchy beneath it.

If this occurs during the scan phase, the backup program will at best not be able to obtain pathname information for the transported files (it may have a chance to recover during the dump phase, however). At worst, this will lead to the inability to find and copy the file during the dump phase. If the movement occurs during the dump phase, the files will not be present in the program's output. Even worse, subsequent ``incremental'' backups will believe the hierarchy to have been backed up because the missing files' modification dates will be older than the date of the previous backup. In all likelihood, this insidious error would only be detected during an attempt to restore the files, at which point it is probably too late.

Device based access methods are less prone to problems created by file movement simply because files do not really move. In a UNIX file system, moving a file involves modifying only directory entries, not the mapping between a file and its inode, or the mapping between a file and its data blocks. Thus, a program accessing files through the raw device is immune to file movement once it enters its dump phase. Note that *dump* must complete the pass in which it copies directory blocks to tape (pass three) before reaching this stage, because it combines the functions of scanning and dumping during this pass.

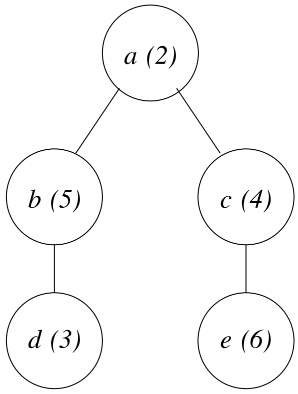

Figure 3: After moving */a/b/e* to */a/c*

Programs like *dump* are still vulnerable to file system name space inconsistencies which can lead to unrestorable files (even though they might be on the tape). In the sample file system shown in Figure 1, a program like *dump* that scans the device in inode order will examine directory *c* (inode 4) before directory *b* (inode 5). If we execute the command

```
% mv /a/b/e /a/c
```

(yielding the result illustrated in Figure 3) after *dump* has scanned directory *c* but before it gets a chance to scan directory *b*, it will still copy *e* during its dump phase. However, the backup tape will contain no path information for *e*, making it unrestorable[1]. The extent of the loss depends on whether *e*

[1] *restore* assumes files without path information were created after *dump* began and discards them with the message ``expected next file <x>, got <y>''.

is a file or directory and, if a directory, the size of the hierarchy below it.

Conversely, the program might also dump the source directory before the destination directory. In this case, *restore* creates an extraneous hard link (linking the file into both directories) when it restores the file. This is benign but nevertheless incorrect.

The Compress Syndrome

A special case of file movement is what we have nicknamed the ‘‘*compress* syndrome’’. This condition occurs when a program such as *compress*(1) that converts a file from one format to another (removing the original) is run. If the output file is placed in a portion of the file system that has already been scanned or dumped and the input file is removed before the program gets a chance to visit it, or if a file's name is changed (e.g., *file* to *file.Z*) after the scan phase but before the dump phase encounters the file, then neither version of the file will appear in the dumpset (with obvious consequences). This scenario can occur in all programs, regardless of access method or execution profile.

File Creation

The ramifications of creating a file in a file system being dumped are relatively innocuous. If the file is created at a point in the file system not yet traversed by the backup program, the effect is the same as if the file had existed when the program was invoked. If the file is added to the file system at a point already traversed, the file will be skipped if the program is using combined scan and dump, or if the program is executing its dump phase. As with movement, a program using sequential scan and dump has the opportunity to detect files created during its scan phase and copy them during its dump phase. We say this case is relatively innocuous because even if the file is not copied during this backup, it will be copied during the next incremental backup, unlike some of the failure scenarios associated with file movement.

File Deletion

There is very little a backup program can do if a file is deleted before the program reaches that point in the file system. One can only hope the file was saved during the previous backup. Programs using the file system interface could conceivably become ‘‘lost’’ should their current or some set of parent directories be removed out from under them. Again, properly written programs can defend against this condition by avoiding the use of ‘‘..’’.

File Modification

Appending to a file being backed up will probably produce an incomplete representation of the file being updated for several reasons. First, programs such as *dump* obtain the size of each file before copying its data and will only copy this amount of data. Second, reading is almost always faster than writing. This means that a process reading a file that is being updated will eventually catch up with the process writing the file and will see the end of the file. Third, the writing process may be driven interactively, in which case the associated ‘‘think time’’ will accelerate the process of ‘‘catching up’’ just described. If the file update is occurring in place the results are somewhat unpredictable; the backed up file may contain a mixture of old and new data.

File Truncation

In general, if a file is truncated as it is being backed up, the backup program will only be able to copy data up to the point of truncation. Backup programs using *mmap*(2) to copy data must be careful not to attempt to read past this point or be prepared to catch `SIGSEGV` and react accordingly.

A device-based program will not react as quickly to truncation as a program using file system access, which may lead to inclusion of spurious data blocks in the backed-up image of a file. This could occur if data blocks originally allocated to the truncated file are re-used after the program has collected the file's block allocation list but before it actually copies the data in the blocks. This scenario can potentially lead to data corruption and security violations. For example, a programmer might truncate his source file and on restoration discover it contains the corporation's complete salary information!

Another quirk (really of *restore*) can lead to the unusability of large portions of a dump tape. If a file is truncated such that *restore* is presented less data than expected, the truncation boundary must be such that *dump* includes at least 8 Kbytes of data (actually, the maximum file system block size) in the dumpset. Should a file be truncated such that this condition cannot be satisfied, *restore* will immediately abort, rendering all data on the dumpset from that point on unusable. This loss could be catastrophic if this error occurs near the beginning of a full dump.

Solutions

We now examine some of the techniques one can use to build a backup program capable of producing reliable results when run on active file systems. Note that what will be discussed in this section is only necessary on mounted, active file systems. The tasks described here are unnecessary when backing up unmounted, quiescent file system.

Defending Against Movement

File System Locking

One way of ensuring a consistent file system name space during a backup is to employ kernel support to block such operations. Unfortunately, although this solution is extremely reliable, it is also very difficult to implement. It requires kernel source and very sophisticated knowledge of how the UNIX file system works. How long the file system would remained locked depends on the access method used by the program – device based or file system based.

In a device based strategy, one would block `rename` and `unlink`[2] system calls during the scan phase, until the program obtains a consistent copy of the file system's name space. In the *dump* program, this would mean locking the file system during passes one, two, and three. After this point, the program can release the locks as subsequent file movement would not affect the process of locating and copying the remaining data blocks.

A file system based strategy dictates that the locks be held for the duration of the backup, as movement while files are being copied can lead to file loss (as discussed above). A hybrid strategy or position-independent (i.e., inode-based) way of opening files during the dump phase would allow the program to release the locks during its dump phase.

Since a file system is arguably less available while it is locked, the amount of time so spent should be minimized as much as possible. This requirement makes programs based on simultaneous scan and dump algorithms less attractive because they are required to lock the file system for the entire backup. One important point to bear in mind is that although the availability of one file system may suffer slightly, overall system availability is maximized.

Detecting Moved Files

Without kernel support, it is difficult to obtain a consistent backup from an active file system. One possible approach is to detect file movement on the next incremental backup and ensure the moved files are copied then. For instance, a *tar* based scheme might miss an entire hierarchy during a full backup. If the backup program kept a list or database of the files it backed up, the next incremental run could check each file it encountered against the list from the previous backup. If the program discovered a file with a modification time older than the date of the full backup but not in the database, it could assume that it missed the file due to movement and back the file up anyway.

[2]The combination of `link` followed by `unlink` can be equivalent to `rename`, whereas `link` by itself is rather benign.

Defending Against Modification

File System Locking

Again, if changes are to be disallowed during the backup, the best strategy is one based on locking the file system against modifications. System calls such as `write`, `truncate`, `mkdir`, `link`, and `mknod` all cause modifications to files or directories. In addition, files may be modified without the use of explicit system calls if they are mapped into the process' virtual memory (with `mmap`).

Active File Detection and Response

Another approach depends on detecting modifications rather than preventing them. This strategy is based on comparing the file's modification time prior to and after dumping and potentially some period of time after dumping. If the times prior to and immediately after dumping are different, the file was modified as it was being dumped. Comparison of file modification times some time after the file has been completely dumped will indicate whether the file was still in the process of being updated after its dump ''completeed''. This is the so-called ''slow writer problem'' and is common on file systems exported to NFS.

Should subsequent activity be detected in this manner, the same file could be ''re-dumped'' in an attempt to collect a complete version. In order to prevent the output media from being filled by copies of this same file, this is something that cannot be done repeatedly or if the file is very large.

Defending Against Truncation

It should come as no surprise that file system locking is the only way to prevent files from being truncated as they are being backed up. In fact, this problem is really an offshoot of the previous one (modifications) and is solved concurrently. Should locking not be available as a solution, the implementer of the backup program must be careful to code defensively, assuming that a file could be truncated out from under the program at any time.

In particular, if the copy mechanism involves the use of *mmap*, truncation could lead to an attempt to read past the end of the file, which generates a segmentation violation (`SIGSEGV`). This signal is easily caught, but the programmer must be aware of this possibility.

Programs such as *restore* that use on-media data for synchronization can easily lose their place in the stream if *dump* provides them with less data than they expect. The programmer must take care to ensure than under no circumstances does the restoration program lose synchronization, as this effectively results in the loss of data.

A Working Dump Paradigm

Goals

In the development of our on-line solution, we defined the following criteria for what constitutes a ''good'' dump:

- The file system being dumped remains on-line – most file operations proceed.
- All operations performed by NFS clients on file systems being dumped proceed unless explicitly locked.
- All operations performed by NFS clients on file systems not currently being dumped proceed as usual.
- All files present at the start of the dump are backed up and can later be restored (this is optional).
- If a file changes as it is being backed up, *dump* attempts to copy the most recent version of that file (optional, subject to site-definable restrictions on file size and number of retries).
- *dump* provides notification of changing files if a consistent (and complete) version cannot be dumped (again, optional).
- Files created after the dump begins are dumped no later than the next incremental backup.

Our new *dump* program addresses these criteria through a combination of kernel support (including file system locking and file hole detection), an undocumented position-independent access method, active file detection, and various defensive measures. While designing the system, we recognized that there are inherent trade-offs between some of the goals outlined above and file system availability; therefore, some of the features that ensure completeness are optional. However, our primary goal was that under no circumstances would we produce a dumpset that could not be restored. We make the guarantee that any file on the dumpset can be recovered.

Name Space Consistency

To meet this goal, we added file system locking to the standard UNIX file system. Our implementation includes five lock levels:

- **none** – all operations are allowed to proceed.
- **write** – any attempt to create or modify a file in the target file system is blocked.
- **name** – any attempt to move a file in the target file system is blocked. This blocks `rename`, `rmdir`, and `unlink`.
- **delete** – any attempt to remove a file from the target file system is blocked.
- **hard** – any attempt to access the target file system fails.

A locked operation blocks until that particular lock is released (with the exception of the hard lock, which returns an error). NFS clients attempting locked operations block and behave as if the server is down (with respect to the locked file system, only) generating the familiar ''server not responding'' message. Operations pending against unlocked file systems are not affected by locks on other file systems.

In our solution, *dump* installs a **name lock** (blocks all `rename`, `rmdir`, and `unlink` operations) during its scan phase until it has dumped all directories in pass three – this takes on the order of 5 minutes for a 1 Gbyte file system. At this point, by default we relax the lock by installing a **delete lock**, preventing removal of files for the duration or the backup.

We recognize that this might be too intrusive for some environments, so we optionally allow the administrator to specify that all locks are to be removed during the remainder of the dump. By the same token, we also realize that some file systems may require complete locking, so we allow the administrator to specify that a file system should be **write locked** for the entire backup.

We take care to avoid deadlocking by refusing to write output files on locked file systems. We also track whether the directory data has been completely committed to the dumpset. This last point is important in case a device error necessitates restarting from a volume checkpoint. If the error occurs on a volume containing any of this dump's directory data, we must restart the dump from scratch because once *dump*begins regular files it relaxes the name lock. Without restarting the dump, we cannot make the guarantee about the dumpset's restorability.

Data Consistency

We wanted to attempt to obtain the most consistent copy of each file's data as possible. To attain this goal, we needed a method of accessing files through the file system in order to use data in the kernel's buffer pool. Traditional access routines such as `open` or `fopen` were not acceptable, as *dump* does not maintain the requisite path information.

We chose to use an undocumented option to `fcntl` that allows the programmer to specify the file to be opened by inode and inode-based generation number. This routine returns a file descriptor as if the file had been opened with `open`, but does so in a completely name-independent way. In conjunction with this mechanism we invented a special `ioctl` call that returns a representation of the file's block allocation list, so we could build the map of holes required by *dump*.

Active Files

A file is deemed ''active'' if its modification time changes between the time the file is opened for dumping and the time its last byte is copied. Should *dump*detect it adds the active file to an internal **active file list** and the file become a candidate for redumping.

Whether the file is dumped again depends on site definable criteria such as the file's size, type, and number of previous retries. *Dump* maintains a table of such criteria that specifies under what circumstances files should be recopied. In order to prevent the output media from being filled by copies of a small number of files, this table typically limits the number of retries to some small number (possibly zero) for files that cross certain size thresholds.

Once *dump* completes its dump phase (copying all files to the dumpset), it closes the dumpset (writing two tape marks if the dump volume is a tape) and makes a pass through the active file list. Should an active file match the criteria for redumping, its retry count is updated and it is added to an internal list of files to be recopied. *Dump* generates a status message for each file on the active list informing interested parties of the file's status (either incomplete or attempting to recopy).

If any files are present on the recopy list after completing the scan through the active list, *dump* appends a complete dumpset (it looks a little like an incremental dump) containing the files on the recopy list. *Dump* repeats this process using the previous iteration's recopy list as the basis for the new active list until it produces an empty recopy list (either all files are successfully copied, or the number of retries is exhausted). The dumpsets produced by this method are fully compatible with the current version of *restore*.

Defensive Measures

We take great care to ensure that *restore* never loses synchronization, regardless of what might have been happening to the file system during the dump. In particular, we sense file truncation and generate the amount of data that will keep *restore* in sync. If locking has been released during the dump phase, files can be deleted at inopportune times, such as during volume switches. We also ensure that *dump* generates the appropriate amount of data after a volume switch, even if the file being continued was deleted during the time it took to mount the next tape.

Conclusions

Much can go wrong during an on-line backup if the program in use does not contain appropriate safeguards. File movement is probably the greatest threat to reliability as it can cause large sections of the target file system to be omitted from the backup, render sections of the file system on the backup media unusable, and prevent sections of the hierarchy from being copied during subsequent incremental backups. Unfortunately, taking systems down to perform backups severely impacts availability, so a large proportion of administrators elect to take the risk and perform their backups on-line.

Our new backup scheme attempts to mitigate the problems of system availability and backup reliability through the use of file system locking to ensure a consistent file system name space. Within this constraint, we designed the system to be configurable, leaving some of the trade-offs between availability and completeness to the discretion of the administrator. A solution such as this is not for the faint of heart; it is easy to deadlock either the program or the system if not implemented carefully.

Author Information

Steve Shumway is currently a Software Engineer with SunSoft, Sun Microsystems' newly-created software subsidiary. He works at the Rocky Mountain Technology Center in Colorado Springs developing advanced administrative applications. Before joining Sun, he spent a year with Prisma, an ill-fated start-up (also in Colorado Springs) where he hoped to make big bucks. Prior to entering the industrial ranks, he spent several years at Duke University as the Senior Systems Programmer for the Computer Science Department. Steve has a B.S. in Zoology from Duke and completed the course-work for a Master's degree while working there. Steve is married and enjoys skiing, mountain biking, windsurfing, and volleyball, among other things. He can be reached via electronic mail as shumway@central.sun.com.