USENIX Association

Proceedings of the 17th Large Installation Systems Administration Conference

San Diego, CA, USA
October 26–31, 2003

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Further Torture: More Testing of Backup and Archive Programs

Elizabeth D. Zwicky

ABSTRACT

Every system administrator depends on some set of programs for making secondary copies of data, as backups for disaster recovery or as archives for long term storage. These programs, like all other programs, have bugs. The important thing for system administrators to know is where those bugs are.

Some years ago I did testing on backup and archive programs, and found that failures were extremely common and that the documented limitations of programs did not match their actual limitations. Curious to see how the situation had changed, I put together a new set of tests, and found that programs had improved significantly, but that undocumented failures were still common.

In the previous tests, almost every program crashed outright at some point, and `dump` significantly out-performed all other contenders. In addition, many programs other than the backup and archive programs failed catastrophically (most notably `fsck` on many platforms, and the kernel on one platform). Few programs other than `dump` backed up devices, making them incapable of backing up and restoring a functional operating system. In the new tests, crashes are rare, and more programs function as backups of entire systems, but failures in backup and archive programs are still widespread, and `dump` is not inherently more reliable than other programs.

Why Is Copying Files Hard?

You would think that the problem of writing backup and archive programs would have been solved some time ago. After all, the basic task has been performed for almost as long as computers have existed. But there are a number of problems.

Flexibility and Inflexibility

Filesystem design changes relatively fast, and human behavior changes even faster. Archive formats, on the other hand, mutate so slowly that many archive programs will attempt to not only read but also write tar archives in a format not significantly changed in the last 20 years. This is a format that allowed 100 characters for a filename (because it was invented in a world that had only recently given up 14 character filenames, and that had not yet reached auto-completing shells or graphical user interfaces).

The result is what amounts to a war between the filesystem people and the archive people. Just when you think you've invented an archive format that's satisfactorily flexible – it doesn't arbitrarily limit filenames, it can deal with any number of hard links, it can take any character you can dish out – somebody comes along and increases the size of inode numbers, device numbers, user names (which you may be storing as part of the ownership data), and then adds access control lists.

Combinatorics and the Contrariness of Users

My current set of test conditions involves tens of thousands of files, and it tests a fairly small number of conditions. It's quite possible that it misses a number of cases; for instance, what about bugs that are triggered by files with very long names and unusual permissions? Testing backup and archive programs is not an easy process, and it's not surprising that program authors have problems with it.

All this wouldn't matter much if people used a consistent set of filesystem features, but they don't. However improbable a combination may appear, somebody somewhere is going to use it for something. It may be pure accident; it may be the result of file sharing, where perfectly reasonable behavior on one system often becomes deeply weird behavior on another; it may be an automated system that ends up doing things a human wouldn't have thought of; or it may be human perversity. But one way or another, over time, the oddest corners of the filesystem will be explored.

Active Filesystems

Filesystems that are in use are not guaranteed to be in a consistent state. Worse yet, even if they are in a consistent state at the moment when you start to back them up, they will change during the process. Files will be created, deleted, or modified; whole directories may come and go. These problems are discussed in detail in Shumway [1].

The Tests

File Size

Historically, some archive programs have chosen to ignore empty files or directories. This is extremely annoying; empty files are often being used as indicator flags, and empty directories may be mount points. The test program therefore creates an empty file and an empty directory to ensure that they can be backed up. It also creates a test file over four gigabytes to ensure that the program handles large files well.

Devices

In order to restore a system to a functional state, you need to be able to back up and restore devices. The testing program therefore creates a named pipe, a block device, and a character device. (I know of no reason why a program would back up block devices but not character devices, or vice versa; it does both purely for the sake of completeness.) It also creates a character device with major and minor device numbers 65025, if possible, in order to test handling of new longer device numbers.

Strange Names

Different filesystems impose different limitations on the character sets that can be used in filenames. Because "/" is used as a directory separator, it is illegal in filenames; because null is used as a string terminator, although it is theoretically legal it is not in general practical to insert it in a filename. Beyond that, some filesystems allow only 7-bit characters (standard ASCII, including annoyances like escape and bell, which makes your terminal beep), while others allow larger character sets that use 8-bit or larger characters.

There are circumstances in which files can end up having illegal filenames. In particular, some programs that share filesystems with other operating systems do not go through the local UNIX filesystem. Since "/" is not the directory separator in either traditional MacOS or Windows, while it is traditionally used to write dates with, these machines may write filenames with "/" in them. These problems are rare these days; programs which share filesystems now routinely translate directory separators so that a Macintosh file which contains a "/" is written on UNIX as ":", and a UNIX file with ":" in it is shared to a Macintosh with a "/", for instance. However, you should be aware that any program that writes to the disk device directly, which may include file sharing programs, is capable of writing files that the filesystem doesn't properly understand. Such files may be possible to back up (for instance, with "dump"), but they are unlikely to be possible to restore. These tests don't test truly illegal filenames.

Strange characters cause problems different ways depending on where they're found. For instance, some programs have problems with whitespace at the end or beginning of file names, but not in the middle; others handle directory names, symbolic link names, or symbolic link targets differently from file names. Therefore, the program runs through all 127 characters of 7-bit ASCII, creating the following files, directories, and links (<charnumber> is the decimal number corresponding to the character, and is included so that when things fail, you know which character caused the failure even if it is non-printable):

- Directories, each containing a file named "<charnumber>":
  - funnydirnames/a<char><charnumber>
  - funnydirnames/<char>
- Files, each containing a line of text indicating which file it is:
  - funnyfilenames/a<char><charnumber>
  - funnyfilenames/<char>
  - plainfilename/<charnumber>
- Symbolic links:
  - named "funnysym/a<charnumber>" to "funnyfilenames/a<char><charnumber>"
  - named "funnysym/a<char><charnumber>" to "plainfilenames/<charnumber>"
  - named "funnysym/<charnumber>" to "funnyfilenames/<char>"
  - named "funnysym/<char>" to "plain filenames/ <charnumber>"
- Hard links:
  - between "funnyfilenames/a<char> <char number>" and "funnyhard/a<charnumber>"
  - between "funnyfilenames/<char>" and "funnyhard/<charnumber>"

Since hard links are symmetrical (the two names are equally applicable to the file), there is no particular reason to believe that strange characters will have a different effect in a hard linked file than in a normal file. Stranger things have happened, but even if there is an effect it's liable to be dependent on which name the program encounters first, which is difficult to control for. The main reason for the hard links is simply to ensure that the test directory has a very large number of hard linked files in it, since that presents problems for some archive programs.

Because Unicode characters are pairs (or triples) of 8-bit characters, I test 8-bit characters in pairs. This test actually runs through both valid Unicode characters and meaningless pairs, since it tests every pair where both halves have the eighth bit set (values from 128-255). It creates the same sorts of directories, files, and links as for the 7-bit characters, but sorts them into subdirectories by the value of the first character in the pair (otherwise, the directories get large enough to be annoyingly slow).

Long Names

Traditionally, UNIX-based operating systems have two kernel parameters that control the maximum length of a filename, and they are "MAXCOMPLEN" and "MAXPATHLEN". MAXPATHLEN is known as PATH_MAX in some kernels, and MAXPATHLEN as a concept has been removed from the Hurd. MAXCOMPLEN is the maximum length of a component in a path (a directory name, or a file name). Traditionally, it's 255 characters in Berkeley-based systems, but your kernel may vary; unless you are running an eccentric system, it is unlikely to be lower than 128 or higher than 512, although some traditionalists may still be running systems with 15 character limits.

MAXPATHLEN is **not** the maximum length of a path. Instead, it is the maximum length of a **relative** path which can be passed to the kernel in a single call. There is no absolute limit on the length of a path in standard UNIX-based operating systems; in any given filesystem, there is a practical limit, which is MAXCOMPLEN times the number of free inodes you have to devote to building a directory tree. In order to build a path over MAXPATHLEN, all that's necessary is to do something like:

```
while (1) {
     mkdir directorynamethatis\
somewherenearmaxcomplenjustfor\
convenience
cd directorynamethatissome\
wherenearmaxcomplenjustforconvenience
}
```

Unfortunately, many programmers have had the understandable but unfortunate impression that MAXPATHLEN is the maximum path length. This leads to any number of undesirable effects (most notably, for a long time fsck simply exited upon encountering such a path, leaving an unclean and unmountable filesystem). Since the first time I ran these tests, this situation has dramatically improved (possibly because a number of evildoers independently discovered this as a user-level denial of service attack against systems). Nonetheless, no backup program I tested will successfully back up and restore paths over MAXPATHLEN.

Saving and restoring files using relative rather than absolute paths would allow backup programs to avoid problems with MAXPATHLEN, at the expense of drastically complicating the programs and the archive formats. It's not clear whether this would be of any benefit; situations where valid data ends up with absolute paths longer than MAXPATHLEN are rare, because people find path names that long inconvenient. However, it's easy to construct scenarios where people using the filesystem as a database could end up with such path names.

First, the test program finds MAXCOMPLEN by experimentation, creating files with names of increasing length until it gets an error. It creates the following:

- Files:
  - "longfilenames/<length>" padded to the length with "a" (as in "longfilenames/10aaaaaaaa")
  - "longfilenames/<length>" padded to the length with newlines
  - "longfilenames/<length>" padded to the length with a valid Unicode character (and an extra "a" if needed, since the Unicode character requires two bytes)
- Symbolic links:
  - named "longsymlinks/<length>" to "longfilenames/<length>a..."
  - named "longsymlinks/q<length>" to "longfilenames/<length>\n..."
  - named "longsymlinks/u<length>" to "longfilenames/<length><Unicode>..."

Then, it builds directories of every length from 2 to MAXCOMPLEN, each of which contains files with names of every length from 2 to MAXCOMPLEN. This may seem silly, but it finds some strange problems that are sensitive to exact path lengths. Each of these files is linked to by a symbolic link with a short name.

Next, it finds MAXPATHLEN by creating directories with names of MAXCOMPLEN, filling each one with names of every length from 2 to MAXCOMPLEN, and descending until it gets an error.

Finally, it ensures that there are paths from the root of the test directory that are over MAXPATHLEN by changing working directories down a level and creating another directory tree out to MAXPATHLEN, this time without filling all the directories out with files. In order to ensure that there are files in the bottom directory, the directory names are slightly shorter than MAXCOMPLEN (due to people's fondness for powers of 2, MAXPATHLEN is generally an even multiple of MAXCOMPLEN). The bottom directory is filled out with filenames until it gets an error.

Access Permissions

Some archive programs have had problems saving or restoring files with tricky permissions. For instance, they may skip files that don't have read permission (even when running as root). Or, they may do straightforwardly silly things like restoring a directory with its permissions before restoring the files in the directory, making it impossible to restore files in a write-protected directory.

The test program creates a file and a directory with each possible combination of standard UNIX permissions (whether or not it makes sense), by simply iterating through each possible value for each position in octal. Each directory contains a file with standard permissions.

Holes

Some filesystems support a spacesaving optimization where blocks that would not contain any data are not actually written to the disk, and instead there is simply a record of the number of empty blocks. This optimization is intended to be transparent to processes that read (or write) the file. On writing, these are blocks that the writing process has skipped with "seek" or its equivalent. A process that reads the file will receive nulls, and has no way of knowing whether those nulls were actually present on the disk, or are in a skipped disk block. Skipped disk blocks are known as "holes," leading to any number of jokes about "holey" files. Files with holes in them are also known as "sparse" files.

This feature is actually used with some frequency; core dumps usually contain holes, as do some database files. These can be extremely large holes, and core dumps often occur on relatively crowded root file systems. Therefore, filling in holes can make it impossible to restore a filesystem, because the restored filesystem is larger than when it was backed up and may not fit on the same disk. On the other hand, swap files and other database files commonly reserve disk space by genuinely writing nulls to the disk. If these are replaced by holes, you can expect performance problems, at a minimum, and if the space that should have been reserved is used by other files, you may see crashes.

Because a reading process simply sees a stream of nulls, regardless of whether they are on the disk or represented by a hole, it is difficult for backup programs that read files through the filesystem to determine where there are holes. On most modern filesystems, the available information for the file does include not only length (which includes holes), but number of blocks (which does not) and size of a block. By multiplying the number of blocks times the size of a block and comparing it to the length of the file, it's possible to tell whether or not there are holes in the file. If the file also contains blocks that were actually written to disk containing nulls, there's no good way to tell which blocks are holes and which are genuinely there. Fortunately, such files are rare.

The test program creates a large number of files with holes; first, there is a file with a straightforward hole, then a file full of nulls that does not contain holes, and then files with 2 through 512 holes, then a file with a four gigabyte hole, and finally (if the filesystem will allow it, which is rare) a file with a four terabyte hole.

Running the Tests

The basic procedure for running the tests was to use the test program to create a directory, to run that through a backup and restore pair (if possible, by piping the backup straight to the restore), and then to compare the resulting directory with the original, using `diff -r` to compare content and a small Perl program to compare just the metadata. The backup and restore were both run as root, to give the program a maximum ability to read and write all the files. If special options were needed to preserve permissions and holes, I used them. If there was an available option to do the read and write in the same process, I tested that separately. Usually I tested both the default archive format and the archive format with the support for the largest number of features; I did not test old archive formats that are known not to support the features I was testing except where the program defaulted to those formats. (In at least one case, I wasn't able to test the program default because it simply wasn't capable of backing up any files on the filesystem I was using – I had accidentally tested large inode numbers!)

I ran the tests on the archive programs installed by default on the operating systems I was testing, plus some archive programs I found on the network. The operating systems were chosen relatively unscientifically; I was aiming for availability and popularity, not necessarily quality. Your mileage will almost certainly vary, running programs with the same names, depending on the program version, the operating system version, the type of filesystem you are using, and even the size of the filesystem (large disks may result in inode numbers too large for some archive formats). I tested on RedHat Linux 8.0 with ext2 file systems, FreeBSD 4.8 with ufs file systems, and Solaris 5.9.

The testing program's approach is brute force at its most brutal, and as such it treats "." and "/" as normal characters, even when attempting to make hard links. This results in tests that attempt to hard link directories together, which is flagrantly illegal. Solaris actually allows this operation to succeed, creating an extra challenge for the programs being tested.

In a few cases, I tested other features; incrementals, exclusion of file names, tables of contents, comparison of archives to directories. When people are actually using backup and archive programs, they tend to use these features, which may change results. In particular, if you use an archive program within a wrapper, the wrapper may rely on tables of contents to determine what was archived; if the table of contents doesn't match the archive, that's a problem. Note that if the table of contents is fine, but the wrapper program has difficulties of its own parsing strange characters or long file names, you still have problems; I didn't test any wrapper programs, but given that many tables of contents use newline as a delimiter, and also put unquoted newlines in filenames, I can only assume that problems will result.

The Results

Tar

Classic tar has very clear limits, imposed by the archive format. These limits include:

- No support for holes
- 100 character filename limit
- No support for devices and named pipes

There is a newer POSIX standard for an extended format that fixes these problems, and most versions of "tar" are now capable of using it.

In my original testing, most tested systems were using classic tar; the POSIX standard was sufficiently new so that only a few operating systems were using it, and as a result tar tested very badly. Even the systems that were using newer tar formats tended to have short filename limits. GNU tar did much better.

Both RedHat and FreeBSD ship GNU tar as `tar`, which defaults to using the extended format.

RedHat: GNU tar 1.13.25

```
tar -cSf - . |
    (cd ../testrestored; tar -xpf -)
```

Removed every symbolic link where the target's last directory component + filename = 490, every file where the last directory component + filename = 494.

Note that longer filenames were handled perfectly, up to MAXPATHLEN. Converted blocks of nulls to holes.

I also tested incrementals (touching every file in the test directory and using tar's list feature), which worked as expected, providing the same results as fulls. Simple exclusion of a test filename worked correctly, both with a normal character and with 8-bit characters. Compressing with -z or -Z provided the same result as testing without compression.

Comparing the archive to the test directory did not work. Tar complained about header problems, complained that the files with holes in them were too large, complained that the archive was incorrect, and finally dumped core.

FreeBSD: GNU tar 1.13.25

I tested only straightforward tar on FreeBSD; the results were identical to the results under RedHat.

Solaris: GNU tar 1.13.19

I tested only straightforward GNU tar on Solaris. Aside from the linking problems mentioned earlier, the only problem was converting blocks of nulls to holes.

Standard Solaris tar

```
tar -cf - . |
    (cd ../testtar; tar -xpf - )
```

Filled in holes. Omitted all files with names over 100 characters long. Omitted all directories with names over 98 characters long. Omitted all symbolic links with targets over 100 characters long.

Dealt with erroneous hardlinked directories by linking the individual files within the directories.

```
tar -cEf - . |
    (cd ../testtare; tar -xpf - )
```

This uses an extended tar format that supports longer names. Unfortunately, it does not appear to support 8-bit characters. It did not succeed in backing up files with 8-bit characters in names or symbolic link targets, and crashed when it processed too many of them, making it difficult to complete the tests correctly. Filled in holes.

Cpio

Cpio relies on find to provide lists of filenames for it, so some of its performance with filenames depends on find's abilities. In the original testing, this presented problems, but since RedHat and FreeBSD both ship GNU cpio and GNU find, which are capable of using nulls instead of newlines to terminate filenames, this problem has been significantly reduced.

Cpio's manual page suggests using the -depth option to find to avoid problems with directory permissions. It does not explain that if you actually do this, you must also use special cpio options to get it to create the directories on restore. I tested both depth-first and breadth-first.

I did not test cpio under Solaris, because of time constraints.

RedHat: GNU cpio 2.4.2

```
find . -true -print0 |
    cpio -0 -o --sparse -H newc |
    (cd ../testcpio; cpio -i)
```

"-H newc" uses newcpio format; this is necessary because as it happens the test filesystem is on a large disk and has large inode numbers, and the default format could not dump any data from it.

Cpio could not handle the larger files with holes in them, claiming they were too large for the archive format, and converted blocks of nulls to holes. (This test run was missing the large file without holes in it.) Otherwise, handled all tests correctly up to MAXPATHLEN.

```
find . -depth -true -print0 |
    cpio -0 -o --sparse -H newc |
    (cd ../testcpio; cpio -i -d)
```

Exactly the same results as without -depth, suggesting that -depth is not in fact important in current versions of cpio if you're running as root.

```
find . -depth -true -print0 |
     cpio -0 --sparse \
          -d -p ../testcpioio
```

Exactly the same results as running a separate cpio to archive and to restore.

FreeBSD: GNU cpio 2.4.2

```
find . -print0 |
    cpio -0 -o --sparse -H newc |
    (cd ../testcpio; cpio -i)
```

The underlying filesystem would not allow creation of the largest files with holes, so those weren't tested. The four gigabyte file was truncated to four blocks, and all holes were filled in. Otherwise, handled all tests correctly up to MAXPATHLEN. Note that although this is the same cpio version tested on RedHat, it did not return the same results! On RedHat, holes were handled correctly.

```
find . -depth -print0 |
    cpio -0 -o --sparse -H newc |
    (cd ../testcpiodepth; cpio -i -d)
```

The same as without the -depth option.

```
find . -depth -print0 |
    cpio -0 --sparse -d -p ~zwicky/testcpioio
```

Omitted paths over 990 characters (MAXPATHLEN is 1024). Correctly handled holes, aside from converting the block of nulls to a hole. Truncated the large file from four gigabytes to four blocks.

Star

Star is an archiver which uses the new extended tar format. It isn't particularly well-known, but it ships with RedHat. It was not tested in my original tests.

RedHat: star 1.5a04

```
star -c -sparse . |
    (cd ../teststar; star -x )
```

Reports 14 files with names too long, one file it cannot stat. Passes all tests up to MAXPATHLEN except that blocks of nulls are converted to holes.

Pax

Pax is an archiver which will use tar or cpio formats; it was one of the first implementations of the POSIX extended tar formats. It doesn't appear to be particularly well-known, despite the fact that it's widely available. Unfortunately, it defaults to the old tar format.

In my original testing, pax did relatively well, correctly handling devices and named pipes, handling holes (but converting blocks of nulls to holes), and correctly handling all file names. It had unfortunate length problems, at different places depending on the format, and bad permissions on directories made it core dump (regardless of format).

RedHat

```
pax -w . |
     (cd ../testrestored; pax -r -p -e)
```

Removed every file with filename at or over 100 characters, every symbolic link with target at or over 100 characters, every directory with a name at or over 139 characters. Turned blocks of nulls into holes.

```
pax -w -x cpio . |
     (cd ../testpaxcpio; pax -r -p -e)
```

Truncated all symbolic link targets to 3072 characters (this is 3 * 1024, which is probably not a coincidence). Returned error messages saying that filenames with length 4095 and 4096 are too long for the format, but in fact it also fails to archive all paths over 3072 characters. Blocks of nulls become holes. The largest files are missing, with a correct error message.

```
pax -r -w -p e . ../testpax3
```

The same results as with cpio format, except that large files are now present.

FreeBSD

MAXPATHLEN on ufs is only 1024 characters, and the large files with holes are not created, so there are no problems with symbolic link targets, long paths, or large holes in files. Otherwise, the results are identical to the results on RedHat; the net effect is that using read-write or cpio mode, all tests are passed up to MAXPATHLEN, except that blocks of nulls are converted to holes.

Solaris

```
pax -w . |
     (cd ../testpax2; pax -r -p e)
```

Removed every file with a filename over 100 characters. Pax then exited after the long filename tests; long hard links, pathnames, and symbolic links were not tested. Changed blocks of nulls to holes; shrank holes (all files with holes larger than one block in them still had holes but were larger on disk than in the original).

Dealt with the erroneous hard-linked directories by unlinking them. However, the second copy of the file encountered became 0 length.

```
pax -w -x cpio . |
     (cd ../testpaxcpio; pax -r -p e)
```

Changed blocks of nulls to holes; shrank holes. Otherwise, passed all tests up to MAXPATHLEN; however, MAXPATHLEN was 1024, not long enough to reveal the length problems shown on Linux.

Dealt with the erroneous hard-linked directories by unlinking them. However, the second copy of the file encountered became 0 length.

```
pax -r -w -p e . ../testpaxrw
```

Pax again crashed, this time during the long pathname tests. This time, I repeated the test on the remaining directories with success. Permissions were not preserved.

Dealt with the erroneous hard-linked directories by linking the individual files within the directories. Blocks of nulls were changed to holes, but all holes were transferred correctly.

Dump/Restore

Dump and restore performed almost perfectly in the original testing, only showing problems with filenames at or near MAXPATHLEN.

RedHat: dump and restore 0.4b28

```
/sbin/dump -0 -f ./dump -A \
     ./dumptoc /dev/hda2
/sbin/restore -if ../dump
```

The symbolic link tests were dramatic failures; several long symbolic links (3 of the 15 links where the filename component was 236 characters long) had content from other files appended to the symbolic link target. The most boring of these changed "236aaa...aaa" to "236aaaa...aaaaized", while a more dramatic one finishes up its a's and continues on with "bleXML\verbatimXML readfile{\truefilename{#1}{}{}\endgraf" and so on for another screenfull. This was so peculiar that I repeated the test, with the test directory on a different filesystem; on the second try, only two of the links where the filename component was 236 characters long were affected, and the filesystem had fewer files with interesting content, so the additional text was boring test content from other files in the test directory.

Aside from that, dump/restore passed all tests up to MAXPATHLEN and appeared to produce a correct table of contents. Running its test of the archive produced the unedifying and incorrect result "Some files were modified!" Even if this had been correct, there is no way that it could ever be useful, since it does not tell you which files it believes were modified.

FreeBSD

Passed all tests up to MAXPATHLEN.

Solaris

Restore crashed and dumped core when attempting to add the files that were over MAXPATHLEN to the restore list. Aside from that, passed all tests up to MAXPATHLEN.

Dealt with the erroneous hardlinked directories by restoring the second link encountered as a file.

Dar and Rat

Dar and rat are Linux archive programs I picked up to see whether the improvements in test results were fundamental changes, or just reflected the increased age and stability of the programs under test. It would appear that the latter is the case.

In general, I test archive programs by running them in the test directory, and archiving the current directory ".". (This gives the most possible pathlength.) Neither dar nor rat would archive "." successfully, and neither one would run with an archive piped to a restore.

I did not end up testing rat, due to its habit of exiting the writing process before actually writing any data when it hit files over MAXPATHLEN. Although this is certainly one approach to error handling (it ensures that you are never surprised by an archive that appears good but is missing files), I didn't have time to weed the testing directory down until I could actually get some testing done.

Dar did somewhat better. On the first attempt, when it reached the largest file with holes, it increased memory usage until the operating system killed the process for lack of memory (unfortunately, at this point my entire login session was also dead, but that could be considered RedHat's fault.) Most annoyingly, although this happens to be one of the last files to be written, so that there was quite a large archive file at that point, no data was recoverable from the archive file (apparently it has to finish in order to be willing to read any data from the archive). In this case, since there was one clearly identifiable offending file, I was able to remove it and re-run the tests, and while dar filled in holes, it otherwise passed all tests up to MAXPATHLEN.

Cross-compatibility

Given that a large number of programs support the same formats, it's reasonable to be curious about compatibility between them. There are far too many possible combinations to test exhaustively, but I tested a few to see what would happen. All of these tests were on RedHat.

Star to Tar

```
star -c -sparse . |
    (cd ../teststar; tar -xp)
```

Removed every symbolic link where the target's last directory component + filename = 490, every file where the last directory component + filename = 494. Note that longer filenames were handled perfectly, up to MAXPATHLEN. Many of the files with holes were deleted; in fact, every other file in the archive. There were error messages complaining about header problems. It is impossible to tell whether blocks of nulls were converted to holes, since that file is one of the missing ones, but it seems safe to assume that they would have been.

Tar to Star

```
tar -cSf - . |
    (cd ../testtartostar; star -xp)
```

Correctly backed up all files up to MAXPATHLEN, except that the block of nulls was converted to a hole.

Conclusions

Archive and backup programs have considerably improved in the last 10 years, apparently due to increased maturity in the programs being distributed. There are still unfortunate and peculiar problems, but compared to the frequent core dumps in the previous testing, they are relatively mild.

It is important to note that you cannot draw conclusions about programs based on their names. As the results show, it's simply inappropriate to make generalizations like "dump is better than tar." This is true on Solaris (where the dump is a reasonably mature version, but tar is merely elderly) but false on RedHat (where the dump has not yet reached version 1, but the tar is the reasonably solid GNU tar). In fact, there were significantly different results for GNU cpio running exactly the same version on exactly the same hardware, using different operating systems and filesystem types. It is simply not safe to generalize about the behavior of programs without looking at how the particular version you are using works with the particular operating system, filesystem, and usage pattern at your site.

For the best results in backups and archiving:

- Use a mature program. Treat people who write their own archive programs like people who write their own encryption algorithms; they might have come up with an exciting new advance, but it's extremely unlikely.
- Test the program you are using in exactly the configuration you need it to run in. Apparently minor changes in configuration may cause large changes in behavior.
- Avoid directories at or near MAXPATHLEN.
- Be sure to use options to handle sparse (holey) files, but be aware that this may create extra holes.

Availability

The test programs were written in a spirit of experimentation, rather than with the intention of producing software for other people to use. I strongly encourage people who are interested in testing backup and archive programs to produce their own tests that cover the cases they are most interested in. However, if you insist on using my programs, or just want to snicker at my programming, they are available from http://www.greatcircle.com/˜zwicky

Biography

Elizabeth Zwicky is a consultant with Great Circle Associates, and one of the authors of ‘‘Building Internet Firewalls,’’ published by O’Reilly and Associates. She has been involved in system administration for an embarrassingly long time, and is finding it increasingly difficult to avoid saying horrible things like ‘‘Why, I remember when we could run an entire university department on a machine a tenth the size of that laptop.’’ Reach her electronically at zwicky@greatcircle.com.

References

[1] Shumway, Steve, ‘‘Issues in On-Line Backup,’’ *Proceedings of the Fifth LISA Conference*, USENIX, 1991,.

[2] Zwicky, Elizabeth, ‘‘Torture-testing Backup and Archive Programs: Things You Ought to Know But Probably Would Rather Not,’’ *Proceedings of the Fifth LISA Conference*, USENIX, 1991.