by Peter Collinson, **Hillside Systems**

DAVID FAUST

Copying Disk Contents

Some interesting e-mail arrived a couple of weeks ago. A reader (and I regret I forget your name, sorry) asked me about the wisdom of attempting to clone disk contents using the `dd` command. The `dd` command is one of those primeval UNIX commands that has been kicking around since the dawn of time. The command was written originally to transfer the contents of magnetic tapes between UNIX and big iron machines (read IBM), and I seem to remember takes its name and argument style from a statement in the IBM Job Control Language.

Data is written onto tapes in files, each created from several blocks of information whose size is chosen by the programmer. UNIX "raw" magnetic tape interfaces have always enabled the user process to set the block size from the `write` system call. If the process tells the kernel to write 10,240 bytes in each `write` system call, then the hardware is instructed to write blocks of 10,240 bytes to tape. When reading the data, the process has to ask for at least 10,240 bytes and is handed data from the tape, a block at a time. Data is lost if the read request doesn't specify enough space for the incoming block.

One function of the `dd` command is that it can be given arbitrary block sizes for both its input and output buffers so data can be re-blocked. The big iron machines also often wrote information as records, so `dd` is able to convert from fixed length records to variable length ones (and vice versa). The command is able to convert to and from EBCDIC, the character representation used by the IBM machines, and ASCII, used by UNIX machines.

It may seem that the `dd` command should have died some years ago, having passed out of usefulness. However, in traditional UNIX style, the input and output devices the command uses are passed in from the command line and are actually file names. On UNIX, devices are addressed as files in the file system, so `/dev/rmt/0` is a tape drive on my current Solaris system. Commands don't distinguish between regular files in the file system and special files unless they wish to.

This device independence supported by UNIX means that `dd` can be used to copy from any file to any other. Disks can be addressed as files too, and you can use `dd` to take a copy of the contents of a disk block by block into a file by reading the disk using the "raw" device interface, or copy one disk to another, again using the low level "raw" disk devices. Of course, in each read, we would gulp as much of the disk as we could fit into memory. This technique is often used to create copies of boot floppies. Because we've now got large virtual memories on UNIX systems, you can copy a floppy in one buffer, read the whole floppy image into memory and write it out again.

However, the technique only works if the media you are using is free from defects. If some portion of the floppy is bad, then `dd` is stymied. In recent times, hard disks have effectively become defect free. Their supporting hardware deals with the reality that there are actually bad spots on the disk surface. However is the case, traditionally it's not been a terribly good idea to perform low level copies of disks. My queasiness about doing this persists to this day.

Also, it's rare to want to clone an entire disk partition at the disk block level. You can obtain the same effect by copying the files in the file system that resides in one partition to another. You don't really care where the files are on the disk, you just need to copy their contents. If you are copying a complete file system hierarchy, then it's usually better to create a new empty file system on the target partition and copy the files and directories, as they are needed. Copying at the file level also allows you to copy files between differently sized partitions.

Copying at the file level will also be beneficial. Few UNIX variants regain the disk space allocated to directories when files are removed from them. If you create a huge number of files in a directory then delete all but one, you will not get the blocks back allocated to hold the contents of the huge directory. Of course, with modern immense disk sizes, it is no longer an issue to worry about wasting the odd block here and there.

However, directory search time can be an issue for some applications. What happens when you create a directory and install several thousand files, and then delete all but the last file created? Well, directory contents are generally not compacted when files are deleted. The entry for this last file will probably be located at the end of all the blocks used to store the directory. When you access the file, the kernel will have to search past a great many empty directory entries to find the file that matches the name you are seeking.

If you are worried about search time for directories, then you may need to remake them from time to time. Note that this only applies to directories that have been expanded to contain a huge number of files that are then deleted and won't really be a huge problem for directories housing really dynamic content.

Using the Standard Utilities

I often forget that these days many of the standard UNIX utilities are supplied with a `-r` (or sometimes `-R`) flag enabling the command to be actioned recursively down a file hierarchy. The `cp` command is one of these. If you want to copy a file system tree from one place to another you can say

```
$ cp -r olddir newdir
```

The command will traverse the `olddir` tree and copy its contents to `newdir`, preserving the directory hierarchy. However, remember that when you copy a single file, the new file is created with your ownership and three times stored with the file. Creation time, modification time and last inode access time are all set to "now." The recursive copy preserves the semantics and it may not be exactly what you want. Sometimes the metadata that's stored with the file is as important as the file contents itself. I make considerable use of the dates on files, not only with automatic programs like `make`, but I often want to find the last file I have dealt with and will casually type `ls -ltr` to list a directory in reverse modified order. You can add `-p` into the arguments to `cp` to make the copy operation preserve the file metadata. You'll need to be superuser or own all the files to ensure a completely faithful copy.

The `mv` command can also be used to relocate whole directory hierarchies. Typing

```
$ mv olddir newdir
```

just renames the entry for `olddir` in the current directory to `newdir`. The tree of files under `newdir` will not be touched and will remain in place. The system will also use the renaming system call when you do something like

```
$ mv olddir /samepartition/newdir
```

assuming (as the name implies) that `olddir` is in the same disk partition as `samepartition`. You can check this by typing

```
$ df .
```

in the source and destination directories. Incidentally, if `newdir` exists in the target directory when you type the `mv` command, then the directory `/samepartition/newdir/olddir` will be created. If you are moving around files, it pays to investigate the destination first.

If you type something like

```
$ mv olddir /otherpartition/newdir
```

where `otherpartition` is on a different partition, then the `mv` command attempts to "simulate" the action of renaming. It will copy all the files and recreate all the directories, before deleting the source hierarchy.

Actually, I rarely use `mv` to move whole trees. One problem is the command isn't really restartable. If something bad happens then you will be unsure of the state of your files. If I'm copying precious files, I'll first want to copy them and then delete the source files, having made sure the copy is complete. Also, there are some efficiency considerations. The file system code attempts to allocate groups of files (i.e. directories and the files they contain) in a small area of the disk called a "cylinder group." The idea is that all the files in one directory and the inodes that point to them are fairly close to each other on the surface of the disk, minimizing head movement. Using `mv` can destroy this efficiency when `mv`ing within the same partition. Again this is not a huge worry, but you do need to know that it might happen.

Using *tar*

My preferred method for copying trees around the file system is to use the `tar` command. The name derives from "Tape archive." The archive or `ar` command is used to create a single file that contains several other files, and is mostly used these days for storing pre-compiled parts of programs. The `tar` command was designed to write many UNIX files out to tape as a single file, while preserving the metadata and pathname for each file. If you look at the man page for `tar`, you will find you can influence the size of the block `tar` writes or reads, which allows tape blocking factors to be controlled.

Of course, as with the dd command, the program can write to any UNIX file because it is not constrained to write only to tape. The file format used by tar is simple. Each file is written to the output file preceded by a header. The earliest forms of tar wrote a binary record as the header, but this was soon changed to a free form text line. The problem with the binary header was byte order. A 16-bit integer value is placed into two bytes and when we read those two bytes, we can reconstitute the integer in two different ways. How we do that depends on the native byte ordering rules on the machine. Writing the binary number as an ASCII string leads to unambiguous interpretation of the value. The ASCII header made the tar format portable between machines and helped when we went from 16-bit to 32-bit machines in the '80s. Also, the ASCII format in the header allowed the same basic format to transition seamlessly from the original 14-character file names of the early UNIX versions to the current 255-character "infinite length" file names.

Creating a tar file on disk from a file hierarchy is simple:

```
$ tar cf file.tar dirname
```

Notice that I've intentionally not created the tar file in the same directory I am copying. If you do this, the command will attempt to read a file it is creating. It will not complain and will copy the file out with unpredictable results.

The output file can be compressed using either gzip or compress. Compressed tar files have become the standard way of moving files around the Internet between UNIX systems, which is called a "tarball." It's being slowly superseded by ZIP format files, due to the influence of the PC, although many of the available PC-based ZIP programs can read and understand tar format.

For copying files from one part of the system to another, I'll typically use tar in a pipe:

```
$ mkdir /dest
$ cd srcdir
$ tar cf - . | ( cd /dest; tar xfp - )
```

I make the target directory where the new file will live, and change into the source directory. I run tar and ask it to copy everything under the source directory ("."). The "c" option tells tar to create a new output file; the "f" in the options is followed by a minus sign which tells the command to write this new file to the standard output channel.

The output of the first tar command above is piped into a new shell signified by the brackets around the commands after the pipe symbol. The first command executed by the new shell changes the working directory to the target directory. It then runs tar in extract mode (the "x" option) reading its data from standard input (the "f" and the minus again). The p in this option string tells tar to recreate the ownership and file permissions from source files in the tar archive. It can only change ownership if superuser is running the command.

The result is a reasonably faithful clone of the source tree, and since the tar format doesn't contain directories, directories are made from scratch whenever their presence is indicated by the pathname. The one danger is the cd command can fail if you mistype it or if it doesn't exist. You can end up copying the source files over themselves, and sometimes this can be bad.

The alternative to tar is cpio. I don't use this much, and mostly find it useful when I wish to be selective about what is copied. The cpio command originated in the development part of AT&T when UNIX was being "productized." It replaced tar on UNIX System III that was the basis of "commercial" UNIX systems. Its file format suffered from having binary headers for many years. When the POSIX committees deliberated about whether they should choose tar or cpio for the POSIX standard, someone pointed out that the file formats were similar and the only real difference was in the arguments to each command. It was felt that it should be possible to produce a single program that could use either argument set. This proved to be doable. The resulting program was called pax, signaling the "peace" that occurred at the end of the "tar wars." The pax program was made freely available and is the basis for most of the versions of tar and cpio on free systems today.

Using *dump* and *restore*

If you wish to copy a whole file system from one partition to a new one, one option is to use the dump command (called ufsdump on Solaris in deference to a standard System V utility called dump that was picked up when merging SunOS with System V). I'll call it dump here. The partner to dump is restore, called ufsrestore on Solaris, in deference to the newly renamed ufsdump program. What a tangled web we weave.

The dump program is another original UNIX utility, but one that has been hacked greatly over the years. It's designed to allow complete backups to be made of a file system onto tape. It knows quite a bit about tape and can handle the need to write more than one tape when saving a file system. Once you have created a complete (or epoch) dump you can create incremental backups that only hold the files that have changed since the last run of the program. The man page goes into this in some detail.

The dump command reads a raw disk partition and understands the file system format. It scans the file system and writes a single file containing a complex binary format starting with the inodes for the file system being written, then writes the directory contents, then the file contents. On a disk that's been used for several years, it can take time to scan the disk to place all the files in the required order and will certainly exercise the disk head while doing so.

Using a raw disk will cause problems if the file system living on the disk is active. The command will operate without the knowledge of the kernel, which may be messing with the disk contents or file structure. If the file system changes while dump is running you can end up with an incomplete dump. The man page has always said "run on a quiescent file system" and means it. Most sysadmins take the stance that they will run dump when the system is quiet and users aren't present because the file system image will be mostly intact. The things that change during the dump will be picked up at the next run or are transitory files that no one really cares about anyway.

The restore command has picked up the terminating "e" during its lifetime, which was originally known as restor. One reason for the change was that Kirk McKusick, the author of the new version, was bothered by the misspelling. Another reason was a fundamental change in functionality, and both commands were available on the system for some period. The original command recovered files to a raw disk, but the current command recovers files into a file system using the normal user-level system calls to do the work. As part of the reworking, `restore` gained the immensely useful `i` option allowing users to walk the file system hierarchy on the dump tape to just recover that one file accidentally deleted.

You can use `dump` and `restore` to move whole file systems from partition to partition with a pipeline:

```
$ cd destination
$ dump 0f - /dev/sd0a | restore rf -
```

Here `destination` is assumed to be a mounted file system that has space to receive the source files. The `0` (zero) option to the `dump` command tells it to take an epoch dump, the `f` and minus forces output to the pipe. I've given a file system name from my BSD/OS system, the command is clever enough to know that it should use the "raw" device and will change the device to `/dev/rsd0a` when it runs. The `restore` command is given the `r` option to tell it to rebuild the file system, and of course is told to read from standard input.

Caveat: I have rarely done this type of file copying and I haven't had the opportunity to test the sequence above for the purposes of this article. Read your man pages carefully when attempting to use the pipeline above. Ideally, the source file system should be unmounted or the system be running in single user mode with no background daemons to ensure you get an unmodified source disk.

Using *rdist*

If you want to maintain a clone of a file system that changes from day to day, then `rdist` may be the command you need. I use this on my Web server to back up the whole active file system from one disk to another, so I always have yesterday's files online and available. The ability to retrieve an old copy of a file has saved me from lengthy tape operations to recover accidentally deleted files on several occasions. My intention also is to have a hot standby. If the main disk dies, I can retreat to using the standby, and perhaps plug this into another machine should I need to.

The `rdist` command was written in the early days of networked workstations. The idea was that it could automatically maintain the system files on a set of workstations from one master copy. Every night `rdist` is woken up and compares the files on the reference machine with those on each of the target machines. If a file on a target machine is out of date, then a new file is copied into place. Optionally, if a file is deleted on the reference machine, then it can also be deleted on the target machine. The machines remain in step running the same code and supporting the same set of system files.

The entire file system on my Web server machine consists of two partitions, the root of the file system and a `/usr` partition. I copy the whole tree onto a matching disk holding two partitions. The root of the clone disk is mounted on `/bck`. This lets me find the copy for any file by taking its pathname: for example, take `/usr/bin/ls` and insert `/bck`, and the clone file is `/bck/usr/bin/ls`.

The `rdist` system is controlled by a small file that tells it what to do. Here's the one I am using:

```
( / ) -> ( localhost )
install -R -w /bck;
except //amd ;
except //bck ;
except //cdrom ;
except //dev ;
except //tmp ;
except //var/tmp ;
except //var/run ;
except_pat ( \\.lock\$ lock\$ FIFO.*\$ );
```

The first two lines tell `rdist` to copy all files from `/` to the machine, in this case `localhost`. The second line tells it to install all files recursively (`-R`) but rename them to start in `/bck` (the `-w` option). There are then a number of `except` statements telling it to ignore various paths in the root file system. Notice that this includes the all-important injunction that it shouldn't copy things in `/bck`, otherwise we would get problems. It also ignores files that are locks.

The above lines are written to a file called `clone_dist`, and the `rdist` command is called every night using:

```
rdist -R -f clone_dist > Log 2>&1
```

The `-R` option here tells `rdist` to delete any files that shouldn't be there. The final piece of magic on the line causes `rdist` to log its actions into a log file so I can see what has happened.

The `rdist` system consists of two parts: a client running on the reference machine and a server running on the target machine. The reference machine uses the `rsh` command to start the server on the client machine and is subject to the usual rules of access for `rsh`. Notice the control file above is actually writing back to the same machine using the `localhost` loopback function, which works as long as you can use `rsh` to `localhost`.

The version of `rdist` on the BSD/OS system on which the above code runs allows you to specify an alternate command to be run instead of `rsh`, and I use a small shell script that just runs `rdist` in server mode. On my machine I am communicating between client and server using a pipe. Sadly this option does not appear to be available on Solaris.

Here's a final caveat: use the debugging facilities when you are setting up `rdist`, and run `rdist` in "tell me what you would do" mode. Do this; it's all too easy to zap files that are important.

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever ... He writes, teaches, consults and programs using Solaris running on an UltraSPARC/10. Email:* `pc@cpg.com`.