Performing Data Surgery

By Jerry Peek

A year ago, the November 2003 "Power Tools" column (available online at http://www.linux-mag.com/2003-11/power_01.html) looked into some lesser-known tools for editing text: the line editors *ex* and *ed*, and the stream editor *sed*.

This month, let's dig deeper and see some uses for the almost-unknown utility *dd* — which has many more uses than just reading data from magnetic tapes (one of its most common uses in years past). On the way, we'll touch upon the better-known editing utilities *head* and *tail*, the "octal dump" utility *od*, the */dev/random* device, and more.

Ready?

Data Is Just Data

Let's start by looking at a common way that Linux handles data.

In general, no matter where Linux data comes from — a disk, a network, a tape, or whatever — it's just a sequence of bytes. That means that you usually don't need to be concerned about how to read from a disk or how to write to a tape. You simply read or write a sequence of bytes, and the device driver does the translation to the device's internal format — disk blocks, network packets, and so on. (However, there are times when it helps to know the block size of a device, and we'll see some of those.)

There's just one data format issue that most everyone needs to understand: the difference between line-structured data and so-called binary data.

In line-structured data, a "line" is a sequence of characters (bytes) followed by a newline character. (Many programming languages represent a newline character as the two-character sequence `\n`, but it's actually stored in a data stream as a single character.)

So, for example, when the *head* utility gives you the first ten lines of a file, it reads a series of bytes from the file until it's seen ten newline characters, then it stops reading. It may do this, for instance, by making ten calls to a library routine that knows how to read a single line of text from a stream of bytes. You can find a text file in the sidebar "What's In That File?"

Binary data is basically anything that's not line-structured. It's simply a series of bytes. It may have internal structure — for instance, many types of data files start with a special sequence of characters to identify it — but there's no Linux-imposed "file type" to tell you how the data is structured.

As mentioned above, data can come from a network connection, from a pipe, from a keyboard, and so on. If the data is in a disk file, someone may have given the file a name ending with a dot and some more characters that identify what's in the file. For instance, *.txt* for text files and *.jpg* for JPEG image files. And some applications may require that you follow that convention. But, in general, Linux doesn't care: it just handles a sequence of bytes.

Just a Little Data, Please

If you read a file with a utility like *cat* or you use the shell's redirection operators (`|`, `<` and `>`) to send data from one device or application to another, you generally get all of the data at once unless you limit it somehow.

If you want to see just the first few lines of a text file or files — to find out what's in them, for instance — the *head* utility can do the job. By default, *head* shows the first ten lines of each file (with a brief header before each file if there's more than one). You can adjust this, say, to get a certain number of bytes instead of lines, with command-line options. The syntax for *head* varies, system to system, so see `man head` for details. A similar utility named *tail* shows the last lines of a file, and most of its versions also handle multiple files.

Like well-behaved Linux utilities, both *head* and *tail* read *standard input* if you don't specify a filename. This lets you use them in a setup like the next example.

```
% prog | tee prog.out | tail
...last ten lines of prog output...
```

The *prog* output goes to *tee*, which writes all of the data into a file named *prog.out* and also sends the data down a pipe to *tail*, which shows the last ten lines of data from *prog* just after it finishes:

head and *tail* work well with text files, but using the utilities with arbitrary files can be problematic. If *head* or *tail* output non-text data to a terminal, you may see garbage, and the data could also corrupt your display. To avoid this problem, you can filter the output by piping it through a pager utility like *less*, which displays non-printable characters safely. You can also pipe the data to *cat* with its `-v` option. That displays control characters (except linefeed and tab) as a two-character string like `^X`, and other non-ASCII characters (with their high bit set) as `M-` followed by the value of the other seven bits.

Chunking Data, Part I

Let's see how to write a long stream of data from (for instance) a pipe or a single file into a series of shorter files.

One way uses *split*. This simple utility reads data and writes it as many fixed-size files as necessary. It fills one file after another until all input data has been read (and written). It's handy, for instance, when you have a long file that you want to transmit over an unreliable network connection — where (unlike modern FTP) the transfer protocol can't resume an interrupted transfer — so any transmission problem means re-transmitting the whole file from the beginning. In that case, it's best to break the huge file into smaller chunks, send the chunks one-by-one (retransmitting any that fail), then re-assemble the transmitted chunks into the complete file.

Here's an example. At your office, you have the 100 MB *bigfile.tar.gz*. You want to split it into one hundred 1 MB files for downloading, via dialup modem, from your home computer. You give the command:

```
office$ split -b 1m bigfile.tar.gz
```

Now you have one hundred 1 MB files named (by default) *xaa*, *xab*, *xac*, ... *xdu*, and *xdv*. You transmit them to your home system. At home, you reassemble the one hundred files, using shell wildcard operators to match all of the three-character filenames in alphabetical order, as in:

```
home% cat x[a-d]? > bigfile.tar.gz
```

It's best to use a utility like *md5sum* or *sum* to be sure that the reassembled *bigfile.tar.gz* is identical to the original.

Data Dumping with dd

dd does low-level data transfer, byte-by-byte or block-by-block, with adjustable block sizes. It can also skip specified numbers of blocks in the input and/or output files, as well as converting data formats. All of those are handy for working with magnetic tape and disks. But it's also useful for many types of data transfers.

By default, *dd* reads the *standard input* and writes to the *standard output*. Input and output filenames, and other options too, are given in an unusual syntax without leading dash (`-`) characters.

For instance, to read a floppy disk and write its image to a file, you could type:

```
$ dd if=/dev/fd0 of=dosboot.img
2880+0 records in
2880+0 records out
$ ls -l dosboot.img
-rw-rw-r- ... 1474560 Nov 2 12:59 dosboot.img
```

The *dd* command line says, "Reading from the input file */dev/fd0*, write all of the data to the file *dosboot.img*." *dd* doesn't try to find lines of data or individual files on the disk; it does a binary copy of the bytes from first to last. *dd* always tells you (on the *standard error*) how many times it read and wrote data. Above, it read 2,880 512-byte blocks. If you don't want to see this information — or any error messages, either — you can redirect *dd*'s standard error to the Linux "bit bucket," */dev/null*, by adding the Bourne shell operator `2>/dev/null` to the command line.

It's more efficient to specify a larger block size so the device drivers do a single read and write. There are lots of other options, and many of them start with `conv=`, like `conv=unblock` to replace trailing spaces in a block with a newline, and `conv=swap` to swap pairs of input bytes (which is needed with some tapes written on other types of hardware). But we'll leave that sort of optimization to you and the *dd man* page. Let's look at some less-obvious uses of this handy utility.

WHAT'S IN THAT FILE?

The *od* ("octal dump") utility displays a file in various formats. It's great for looking at a non-text file without messing up your terminal, or for seeing exactly what's in a text file.

For instance, let's use the Bourne shell's quoting operators to store three lines of text in a file named *textfile*. Then we'll dump it with `od -c`, which shows data in a character format:

```
$ echo "test
> 1
> 2" > textfile
>
$ od -c textfile
0000000   t   e   s   t  \n   1  \n   2  \n
0000011
```

Notice the three newlines in the file? A text file from a DOS-type system would have pairs of `\r \n` at the end of each line. The ending number `0000011` shows, in octal, how many bytes *od* displayed. (11 octal is 9 decimal, so we saw 9 bytes.)

Next let's look at the 50 bytes we read from */dev/urandom.* We'll use an octal byte format (the first byte is 31 octal, the second is 62 octal, and so on):

```
$ od -b myrand
0000000 031 062 132 063 153 015 075 364 061 070
  375 013 365 372 316 270
0000020 256 307 144 345 016 121 162 074 260 151
  022 361 116 257 324 251
0000040 263 056 233 123 171 277 274 121 102 221
  305 111 332 330 327 213
0000060 053 233
0000062
```

Using `od -c` instead shows that some of those random bytes are legal as characters, and others are not. This technique is useful for seeing what's in a file that doesn't seem to "look right" on your terminal.

Stupid dd Tricks

Need a file with 100 arbitrary bytes — for testing, for instance? The Linux device */dev/urandom* (available since *Linux 1.3.30)* can supply as many pseudo-random bytes as you can read from it. To get just 100 bytes, set a block size of 1 byte with `bs=1` and tell *dd* to stop after copying 100 "blocks" (here, that's 100 bytes):

```
$ dd if=/dev/urandom of=myrand bs=1 count=100
```

What's in that *myrand* file? The *od* utility can show you. (See the sidebar "What's In That File?")

If you need more-random data, try */dev/random* instead. Reading data from */dev/random* can take some time, though, as the *random(4) man* page explains. When you read from */dev /random*, set a block size of 1.

No matter where Linux data comes from — a disk, a network, a tape, or whatever — it's just a sequence of bytes

Another use for *dd* is for "wiping" a text file before you delete it. Simply removing a Linux file (with *rm*, for instance) only deletes the inode that points to the data. A cracker with *root* access might read the raw disk (with *dd*!) and find the "deleted" file. We can use *dd* to write random data over the file before deleting it. Normally *dd* truncates a file before writing, so use `conv=notrunc` to make it write over the existing data. Set `bs` to the file size and `count` to `1`. For example:

```
% ls -l afile
-rw——- ... 3769 Nov  2 13:41 afile
% dd if=/dev/urandom of=afile \
  bs=3769 count=1 conv=notrunc
1+0 records in
1+0 records out
% rm afile
```

If you want to, you can repeat the "wiping" command several times with the C shell *repeat* command, the Z shell *repeat* loop, or simply use the history operator `!!`.

Chunking Data, Part II

If you need to emit chunks of data one-by-one, pausing to do some operation between each chunk, *split* can't do the job. It only writes data to files, not stopping until all data has been written. Let's use *dd* instead. We'll also need our knowledge of how Linux handles a stream of data.

POWER TIP: Archive Your Configuration Files

Configuration and setup files can be easy to ignore — and hard to restore after an error or an application crash. So, consider making a directory full of symlinks pointing to important configuration files. Run a script from *cron* that makes local copies (into a sibling directory, for instance) of the symlink targets.

You can also archive all changes to system configuration files with a system like RCS. However, be sure that your system doesn't change the last-modified timestamp of the configuration files. (You might make a copy of the file and archive the copy.) A setup like the *ci_crontab* script, described in the February 2003 column "Running Jobs Unattended" (available online at http://www.linux-mag.com/2003-02/power_01.html) could do the trick.

Listing Two in the May 2004 article "Great Command-line Combinations" (available online at http://www.linux-mag.com/2004-05/power_01.html) showed a shell loop reading data from a stream, line by line, until the data had been exhausted. In that example, we used the *read* command to read lines of *find* output. Now we'd like to use the same technique to read the output of a program *someprog* in fixed-size chunks. The basic *bash* loop looks like this:

```
someprog |
while :
do
  chunk=$(dd count=1 bs=10 2>/dev/null)
  test -z "$chunk" && break
  ...process $chunk...
done
```

The shell operator `:` (a colon), which simply returns a zero ("true") exit status, makes an endless *while* loop that iterates until it's stopped by a *break* command within the loop body.

The *bash* command substitution operator `$(command)` captures the 10 bytes from *dd*'s standard output, which is then saved into the shell variable named *chunk*. We're throwing away *dd*'s standard error, which has messages like `1+0 records in` (and any actual errors, too).

Next, the code tests to see whether `$chunk` is empty and breaks the loop if it is. It can't test *dd*'s exit status (as was done in the May 2004 article) because *dd* returns an exit status of zero whether it has read data or not. A technique like we use here, though — where *dd* breaks data into chunks — can be handy in a lot of cases. Remember to set `count=1` to keep *dd* from reading all of the data during a single pass of the loop.

Jerry Peek is a freelance writer and instructor who has used Unix and Linux for over 20 years. He's happy to hear from readers; see http://www.jpeek.com/contact.html.