Linux Fu: Named Pipe Dreams

Originalartikel

Backup

<html> If you use just about any modern command line, you probably understand the idea of pipes. Pipes are the ability to connect the output from one program to the input of another. For example, you can more easily review contents of a large directory on a Linux machine by connecting two simple commands using a pipe: <pre>ls | less</pre> This command runs

```
ls
```

and sends its output to the input of the

```
less
```

program. In Linux, both commands run at once and output from ls immediately appears as the input of less. From the user’s point of view it’s a single operation. In contrast, under regular old MSDOS, two steps would be necessary to run these commands: <pre>ls > SOME_TEMP_FILE less < SOME_TEMP_FILE</pre> The big difference is that

```
ls
```

will run to completion, saving its output a file. Then the

```
less
```

command runs and reads the file. The result is the same, but the timing isn’t. You may be wondering why I’m explaining such a simple concept. There’s another type of pipe that isn’t as often used: a named pipe. The normal pipes are attached to a pair of commands. However, a named pipe has a life of its own. Any number of processes can write to it and read from it. Learn the ways of named pipes will certainly up your Linux-Fu, so let’s jump in! <h2>Quick Example: Building a Logging Script</h2> Suppose you want to create a simple logging facility. Of course, making a daemon that runs all the time is an entirely different subject, but I’m just going to create a simple and non-robust script. A named pipe can accept the input lines from other programs and the daemon can timestamp each line and write it to a file. Here’s the daemon: <pre>#!/bin/bash mkfifo /tmp/nplogpipe while true do

```
read LINE &lt;/tmp/nplogpipe
echo $(date): "$LINE" &gt;&gt;/tmp/nplog.txt
```

done</pre> The

```
mkfifo
```

command creates the named pipe (a first in, first out or FIFO). Older scripts might use

```
mknod
```

for this purpose and that will work, too. If the pipe already exists, the command will fail, but it won’t matter. After that, a read waits for input from the pipe. When it arrives, the script writes out a date and the line to a log file and goes back for more. To test out your quick and dirty logging system, run the script in the background or in one terminal window. Then in the foreground or in another terminal try this: <pre>echo The first log entry >/tmp/nplogpipe echo Read Hackaday every day >/tmp/nplogpipe</pre> You can add a few more lines and then examine the /tmp/nlog.txt file. It should look something like this: <pre>Sat Jun 29 07:37:44 CDT 2019: The first log entry Sat Jun 29 07:39:57 CDT 2019: Read Hackaday every day</pre> <h2>The Tricks are in the Details</h2> One small point: in this case, the read command executes repeatedly in the loop, but in general when a sender process exits, it will cause the receiving program to also exit. That isn’t always the behavior you want. The usual way to deal with this is to open the pipe in a way that will hold the pipe open until it is closed. To observe this effect, try this (with the daemon running): <pre>ls / >/tmp/nplogpipe</pre> The log will only get the first line of the ls. That’s because by the time it finishes processing the first line, the ls command exited and that clears the pipe. Now try this: <pre>exec 3>/tmp/nplogpipe ls / >&3 free >/tmp/nplogpipe exec 3>&- # close pipe</pre> The first exec line holds the pipe open until the last line closes it. Once open you can refer to the pipe as &3 or by its full name. Now all the output will appear in the log file. Another nuance is that the pipe sort of looks like a file. That means that programs that expect files can usually use pipes. It also means you can control access to a pipe using the same security mechanisms that work with files (e.g., chmod to set permissions for a specific user or group). <h2>Why Use Named Pipes?</h2> You might wonder what advantage these have compared to a regular pipe or a file. Unlike a regular file, the pipe doesn’t fill up. That also means it has more chance of staying in memory although, of course, it could get swapped out just like any other memory. In addition, it is easy to have multiple writers to a named pipe. Of course, there are other issues to worry about. For example, if multiple programs are writing more than one line of data to the pipe at a time, you’d have to work out your own scheme to sort them all out. Still, for a quick way to push and collect data between possibly unrelated processes, named pipes are an easy way to go. By the way, this is the twentieth installment of Linux Fu! The links below will take to the earlier postings and stay tuned for even more to come. </html>