SHAWN POWERS

Pipes and STDs

Standard input, output and error are confusing—until now.

Punny title aside, the concepts of STDIN (standard input), STDOUT (standard output) and STDERR (standard error) can be very confusing, especially to folks new to Linux. Once you understand how data gets into and out of applications, however, Linux allows you to string commands together in awesome and powerful ways. In this article, I want to clear up how things work, so you can make the command line work much more efficiently.

Processes and Their Data

At a basic level, when a process is run on the command line, it has three "data ports" where it can send and/or receive data. Figure 1 shows my depiction of an application's I/O design.

Here are some definitions:

- STDIN: this is where an application receives input. If you run a program that asks you to enter your age, it receives that info via its STDIN mechanism, using the keyboard as the input device.
- STDOUT: this is where the results come out of the program. If you type `ls`, the file listings are sent to STDOUT, which by default displays on the screen.
- STDERR: if something goes wrong, this is the error message. It can be a little confusing, because like STDOUT, STDERR is displayed on the screen by default as well. If you type `ls mycooldoc`, but there's no such file as "mycooldoc", you'll get an error message on the screen. Even though it appears on the screen in the same place STDOUT appears, it's important to understand that it came out of a different place. That's important, because STDOUT and STDERR can be directed separately and to different places.

It's also important to realize that I/O is different from command-line

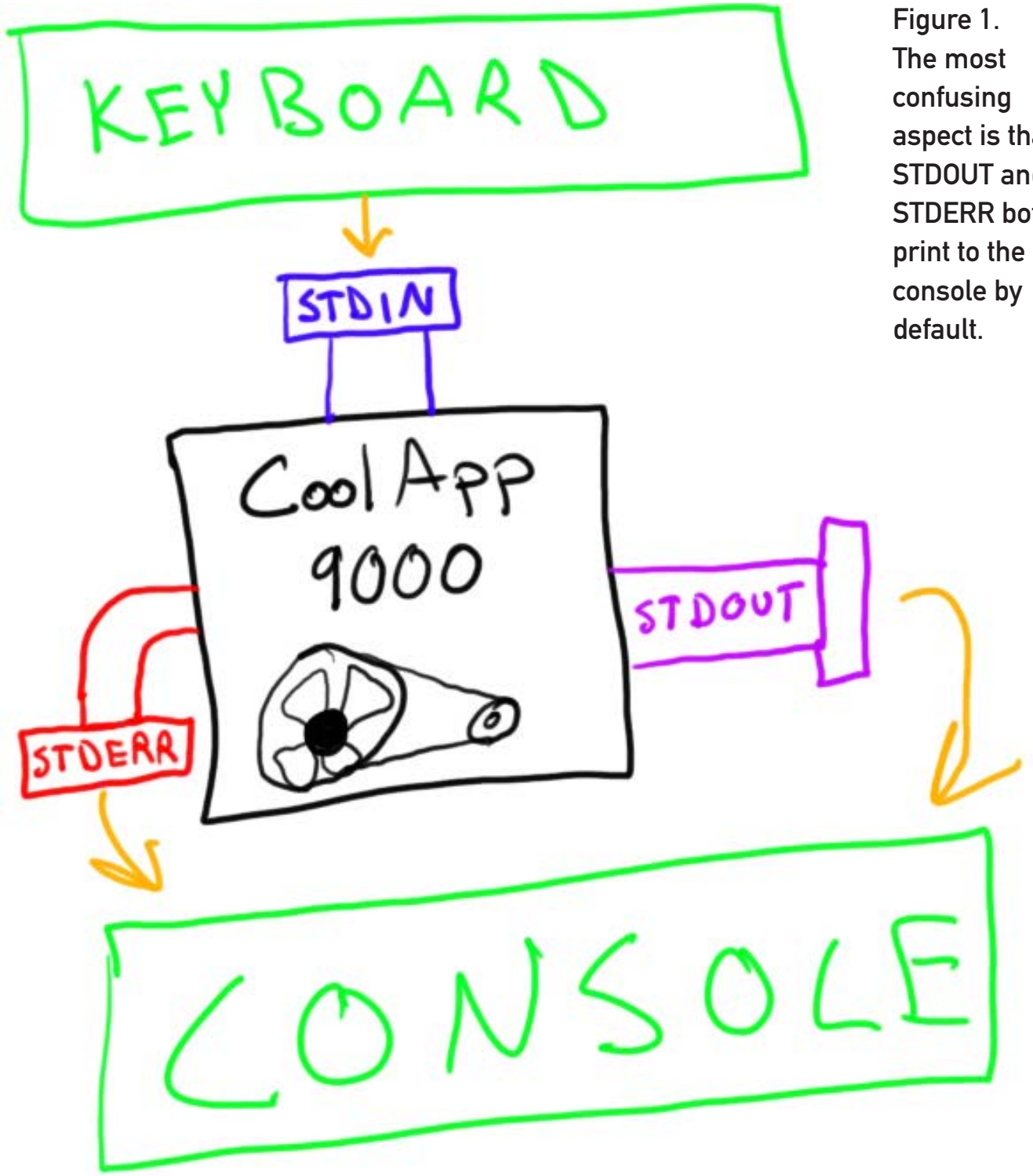

Figure 1. The most confusing aspect is that STDOUT and STDERR both print to the console by default.

arguments or flags. Input, for example, is data the process gets from some external source. When you run a command with arguments, those arguments just tell the process how to run. Typing `ls -l`, for instance, just tells the ls program how to execute. The STDIN/OUT/ERR are used only

once the program is running as a way to send or receive data.

STDIN Example

By default, STDIN is read from the keyboard. So, this little script prompts for input via the keyboard. When you enter the information and press enter, it's fed into the application's STDIN. Then that information is processed, and the result is dumped out of STDOUT, which by default is displayed on the command line:

```
#!/bin/bash
echo "What is your favorite number?"
read favnum
echo "My favorite number is $favnum too!"
```

If you look closely, the initial "What's your favorite number?" text is also sent out STDOUT, and since it defaults to the screen, you see it as a prompt before the script uses the read command to get data into STDIN.

Redirecting STDOUT and STDERR

You've seen that STDOUT and STDERR both default to displaying on the screen. It's often more desirable to have one or both get saved to a file instead of displayed. To redirect the output, use the "greater-than" symbol. For example, typing:

```
ls > results.txt
```

will save the directory listing to a file called results.txt instead of displaying it on the screen. That example, however, redirects only the STDOUT, not the STDERR. So if something goes wrong, the error message displays on the screen instead of getting saved to a file. So in this example:

```
ls filename > results.txt
```

if there is not file called "filename", you'll see an error on the screen even though you redirected STDOUT into a file. You'll see something like:

```
ls filename > results.txt
ls: cannot access filename: No such
file or directory
```

There is a way to redirect the STDERR, which is similar to redirecting STDOUT, and without first understanding the difference between the two output "ports", it can be confusing. But to redirect STDERR instead of STDOUT, you'd type this:

```
ls 2> errors.txt
```

Which, when typed, simply would print the file listing on the screen. Using the 2> structure, you are only redirecting errors to the file. So in this case:

```
ls filename 2> errors.txt
```

As you can imagine, redirecting output is very useful when running scripts or programs that are executed in the background; otherwise, you'd never see the output!

if there isn't a file named "filename", the error message would get saved to the file errors.txt, and nothing would display on the screen. It's possible to do both at once too. So you could type:

```
ls > results.txt 2> errors.txt
```

and you'd see the file listing in results.txt, while any error messages would go into errors.txt. You've probably seen something similar in crontab, where the desire is to have both STDOUT and STDERR go into a file. Usually, the desire is to have them both get redirected into the same file, so you'll see something like this:

```
ls > stuff.txt 2>&1
```

That looks really confusing, but it's not as bad as it seems. The first part should make sense. Redirecting STDOUT into a file called stuff.txt is clear. The second part, however, is just redirecting STDERR *into* STDOUT. The reason you can't just type 2>1 is because Bash would interpret that as "I want to save the STDERR into a file named 1", so the ampersand preceding the 1 tells Bash you want to redirect the STDERR into STDOUT.

One last trick regarding the redirection of STDOUT and STDIN is the double greater-than symbol. When you redirect output into a file using the > symbol, it overwrites whatever is in the file. So if you have an errors.txt file, it will overwrite what's already in there and just show the single error. With a >> symbol, it will append the results instead of overwriting. This is really useful if you're trying to make a log file. For example, typing:

```
ls >> files.txt
ls -l >> files.txt
```

will create a file called "files.txt" that has a list of the files, then a long directory listing of the same files. Thankfully, if the file doesn't exist, using a double greater-than symbol will create

the file just like a single greater-than symbol will do. As you can imagine, redirecting output is very useful when running scripts or programs that are executed in the background; otherwise, you'd never see the output!

Redirecting STDIN

This concept is a little bit harder to understand, but once you "get" the whole concept of I/O, it's not too bad. It's important to know that not all applications listen on their STDIN port, so for some programs, redirecting STDIN does nothing. One common command that does listen on STDIN, however, is grep. If you type:

```
grep chicken menu.txt
```

the grep command will search through the menu.txt file for any lines containing the string "chicken", and print those lines on the screen (via STDOUT, which should make sense now). grep also will accept input via STDIN instead of via filename, however, so you could do this:

```
cat menu.txt | grep chicken
```

and the exact same results will be shown. If that seems confusing, just walk through the process with me. When you type `cat menu.txt`, the cat program displays the contents of menu.txt to the screen, via STDOUT. If you used a > symbol, you could redirect that STDOUT into a new file, but if you use the pipe symbol (|), you can redirect the STDOUT data into another program's STDIN. That's what's happening in this example. It's as if the cat program's purple STDOUT tube in Figure 1 is bolted directly onto grep's blue STDIN tube. Since grep is listening on its STDIN port for data, it then executes its search for the word chicken on that data that is coming into STDIN rather than reading from a file.

This example above might seem frivolous, and honestly it is. Where redirecting with a pipe symbol comes in handy is when you string together multiple actions. So this, for example:

```
grep chicken menu.txt | grep pasta
```

will return a list of all of the lines in menu.txt that have the word "chicken" in them *and* have the world "pasta" in them. You could do the same thing by typing this:

```
grep chicken menu.txt > chickenlist.txt
grep pasta chickenlist.txt
```

But, that takes two lines, and then you have a fairly useless file on your system called chickenlist.txt, when all

Once you get used to piping STDOUT from one command into STDIN for another, you'll find yourself becoming a grep ninja in no time.

you wanted was a list of items that contain both chicken and pasta.

Once you get used to piping STDOUT from one command into STDIN for another, you'll find yourself becoming a grep ninja in no time. Granted, there are many other applications that listen on STDIN for information, but grep is one that is very commonly used. For example:

```
ls -l /etc | grep apache
```

is a way to look for any files or directories in the /etc folder that contain the string "apache" in their name. Or:

```
cat /var/log/syslog | grep USB
```

is a great way to look for any log entries in the syslog that mention USB devices. You even could go further and type:

```
cat /var/log/syslog | grep USB > usbresults.txt
```

and you'd have a text file containing any lines in /var/log/syslog that mention USB. Perhaps you're troubleshooting an issue, and you need to send those lines to a tech support person.

Redirecting STDOUT and STDERR into a file, or piping them into another process' STDIN, is an important concept to understand. It's important to know the difference between what a >, >> and | do so that you get the results you want. Sometimes redirecting STDOUT, STDERR and STDIN aren't enough, however, because not all applications listen for data on STDIN. That's where xargs comes into play.

xargs: Making Apps Play Nice

Sometimes you want to use the STDOUT from one command to interact with an application that doesn't support getting data piped into STDIN. In this case, you can use the simple and powerful xargs command. Here's a scenario: your hard drive is filling up, so you want to delete all the .mp3 files in all the folders in the entire system. You can get a list of all of those files by typing:

```
find / -name "*.mp3"
```

and you'll get a list via STDOUT of all the files and their full paths. So you might get something like:

```
/home/spowers/music/re_your_brains.mp3
/home/spowers/music/mysong.mp3
/tmp/coolsong.mp3
/home/donna/.secretfolder/bieber.mp3
/home/donna/.secretfolder/rundmc.mp3
```

You could go through and find all those files and delete them one by one, but it would be far more useful if you could just `rm` them all at once. Unfortunately, `rm` doesn't accept file listings via STDIN, so in order to accomplish this goal, you have to use xargs. It would work like this:

```
find / -name "*.mp3" | xargs rm -rf
```

What xargs does is listen on its STDIN, and then execute whatever command you tell it to while pasting its own STDIN onto the end of the command. So running that above command effectively is executing:

```
rm -rf /home/spowers/music/re_your_brains.mp3 \
/home/spowers/music/mysong.mp3 /tmp/coolsong.mp3 \
/home/donna/.secretfolder/bieber.mp3 \
/home/donna/.secretfolder/rundmc.mp3
```

And since `rm` will delete all the files you list, it deletes all the files with a single command. Without xargs to feed the list of files from the find command, it turns out to be surprisingly difficult to accomplish the task.

STDs and Pipes: Thinking Tools

For some people, the concepts of STDIN, STDOUT and STDERR are second nature. But without that foundational understanding of how processes do I/O, redirection and piping are pretty much incomprehensible. Also, once you truly understand how it all works, the xargs program really starts to shine. I urge you to play around with redirection and piping. At the very least, your grep kung-fu will benefit from the practice!■

Shawn Powers is the Associate Editor for *Linux Journal*. He's also the Gadget Guy for LinuxJournal.com, and he has an interesting collection of vintage Garfield coffee mugs. Don't let his silly hairdo fool you, he's a pretty ordinary guy and can be reached via e-mail at shawn@linuxjournal.com. Or, swing by the #linuxjournal IRC channel on Freenode.net.

Send comments or feedback via http://www.linuxjournal.com/contact or to ljeditor@linuxjournal.com.