CATONMAT

Bash One-Liners Explained, Part III: All about redirections

Last updated 3 weeks ago

This is the third part of the **Bash One-Liners Explained** article series. In this part I'll teach you all about input/output redirection. I'll use only the best bash practices, various bash idioms and tricks. I want to illustrate how to get various tasks done with just bash built-in commands and bash programming language constructs.

See the first part of the series for introduction. After I'm done with the series I'll release an ebook (similar to my ebooks on awk, sed, and perl), and also bash1line.txt (similar to my perl1line.txt).

Also see my other articles about working fast in bash from 2007 and 2008:

- Working Productively in Bash's Emacs Command Line Editing Mode (comes with a cheat sheet)
- Working Productively in Bash's Vi Command Line Editing Mode (comes with a cheat sheet)
- The Definitive Guide to Bash Command Line History (comes with a cheat sheet)

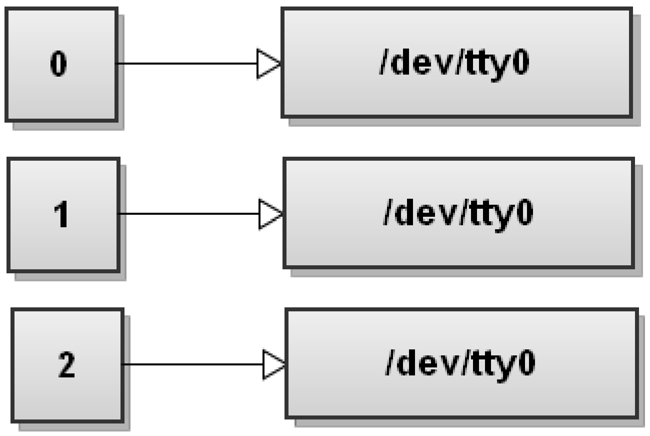
Let's start.

Part III: Redirections

Working with redirections in bash is really easy once you realize that it's all about manipulating file descriptors. When bash starts it opens the three standard file descriptors: stdin (file descriptor 0), stdout (file descriptor 1), and stderr (file descriptor 2). You can open more file descriptors (such as 3, 4, 5, ...), and you can close them. You can also copy file descriptors. And you can write to them and read from them.

File descriptors always point to some file (unless they're closed). Usually when bash starts all three file descriptors, stdin, stdout, and stderr, point to your terminal. The input is read from what you type in the terminal and both outputs are sent to the terminal.

Assuming your terminal is `/dev/tty0`, here is how the file descriptor table looks like when bash starts:



When bash runs a command it forks a child process (see `man 2 fork`) that inherits all the file descriptors from the parent process, then it sets up the redirections that you specified, and execs the command (see `man 3 exec`).

To be a pro at bash redirections all you need to do is visualize how the file descriptors get changed when redirections happen. The graphics illustrations will help you.

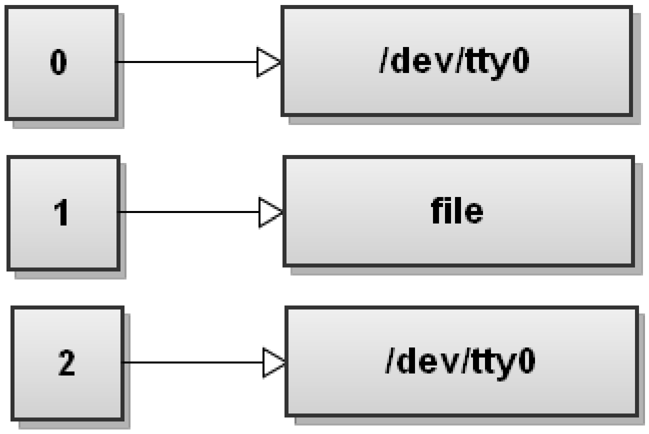
1. Redirect the standard output of a command to a file

```
$ command >file
```

Operator `>` is the output redirection operator. Bash first tries to open the file for writing and if it succeeds it sends the stdout of `command` to the newly opened file. If it fails opening the file, the whole command fails.

Writing `command >file` is the same as writing `command 1>file`. The number `1` stands for stdout, which is the file descriptor number for standard output.

Here is how the file descriptor table changes. Bash opens `file` and replaces file descriptor 1 with the file descriptor that points to `file`. So all the output that gets written to file descriptor 1 from now on ends up being written to `file`:



In general you can write `command n>file`, which will redirect the file descriptor `n` to `file`.

For example,

```
$ ls > file_list
```

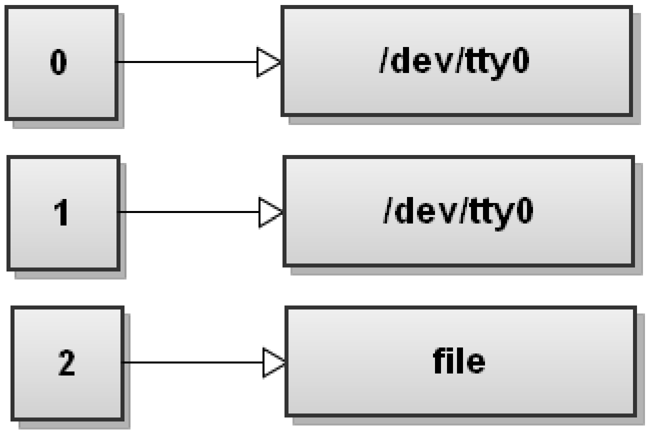
Redirects the output of the `ls` command to the `file_list` file.

2. Redirect the standard error of a command to a file

```
$ command 2> file
```

Here bash redirects the stderr to file. The number `2` stands for stderr.

Here is how the file descriptor table changes:



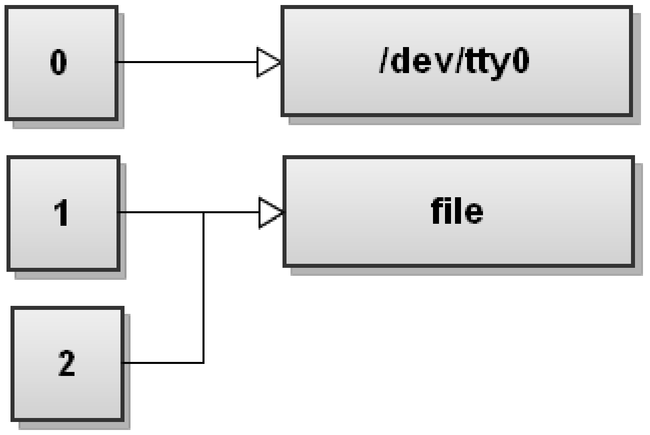
Bash opens `file` for writing, gets the file descriptor for this file, and it replaces file descriptor 2 with the file descriptor of this file. So now anything written to stderr gets written to file.

3. Redirect both standard output and standard error to a file

```
$ command &>file
```

This one-liner uses the `&>` operator to redirect both output streams - stdout and stderr - from `command` to `file`. This is bash's shortcut for quickly redirecting both streams to the same destination.

Here is how the file descriptor table looks like after bash has redirected both streams:



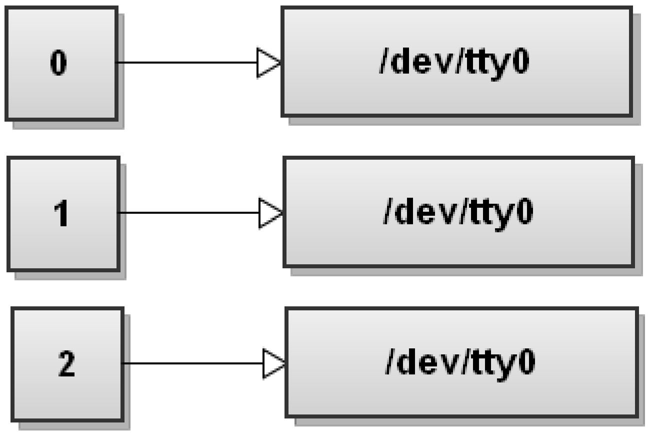
As you can see both stdout and stderr now point to `file`. So anything written to stdout and stderr gets written to `file`.

There are several ways to redirect both streams to the same destination. You can redirect each stream one after another:

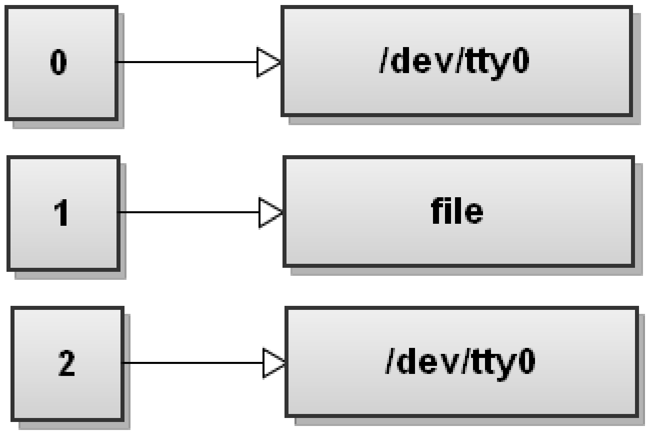
```
$ command >file 2>&1
```

This is a much more common way to redirect both streams to a file. First stdout is redirected to `file`, and then stderr is duplicated to be the same as stdout. So both streams end up pointing to `file`.

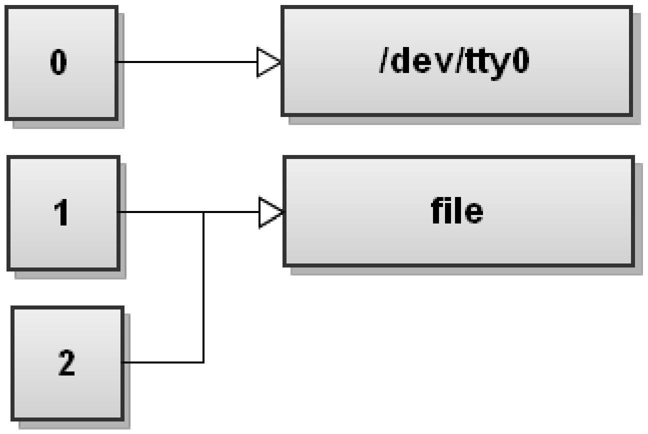
When bash sees several redirections it processes them from left to right. Let's go through the steps and see how that happens. Before running any commands bash's file descriptor table looks like this:



Now bash processes the first redirection `>file`. We've seen this before and it makes stdout point to `file`:



Next bash sees the second redirection `2>&1`. We haven't seen this redirection before. This one duplicates file descriptor 2 to be a copy of file descriptor 1 and we get:



Both streams have been redirected to `file`.

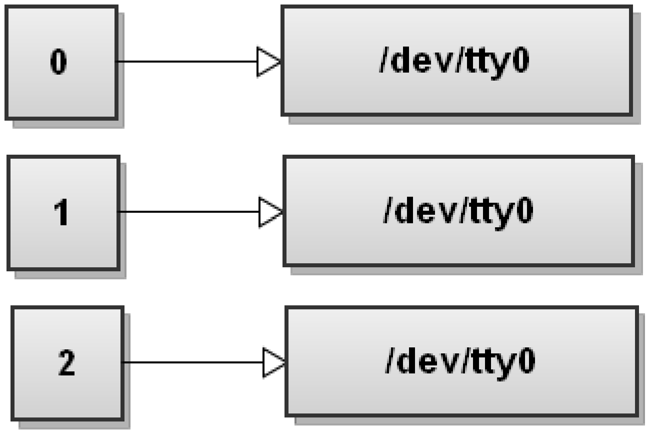
However be careful here! Writing:

```
command >file 2>&1
```

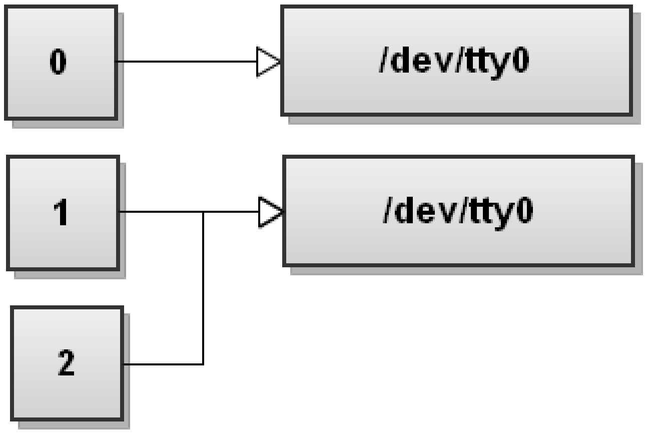
Is not the same as writing:

```
$ command 2>&1 >file
```

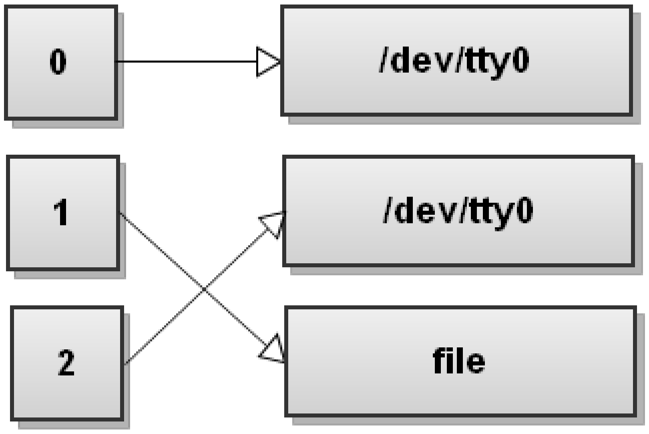
The order of redirects matters in bash! This command redirects only the standard output to the file. The stderr will still print to the terminal. To understand why that happens, let's go through the steps again. So before running the command the file descriptor table looks like this:



Now bash processes redirections left to right. It first sees `2>&1` so it duplicates stderr to stdout. The file descriptor table becomes:



Now bash sees the second redirect `>file` and it redirects stdout to `file`:



Do you see what happens here? Stdout now points to `file` but the stderr still points to the terminal! Everything that gets written to stderr still gets printed out to the screen! So be very, very careful with the order of redirects!

Also note that in bash, writing this:

```
$ command &>file
```

Is exactly the same as:

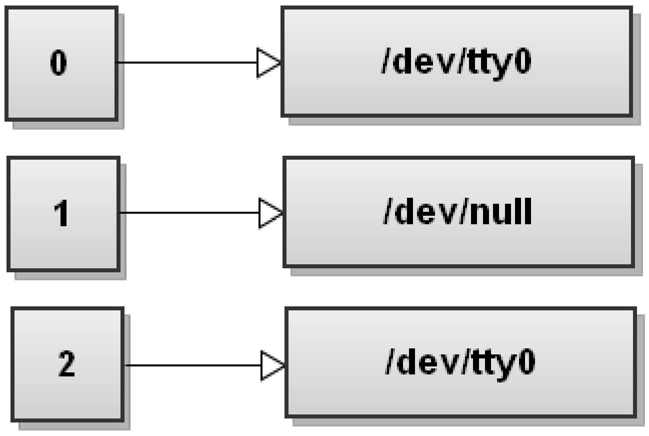
```
$ command >&file
```

The first form is preferred however.

4. Discard the standard output of a command

```
$ command > /dev/null
```

The special file `/dev/null` discards all data written to it. So what we're doing here is redirecting stdout to this special file and it gets discarded. Here is how it looks from the file descriptor table's perspective:



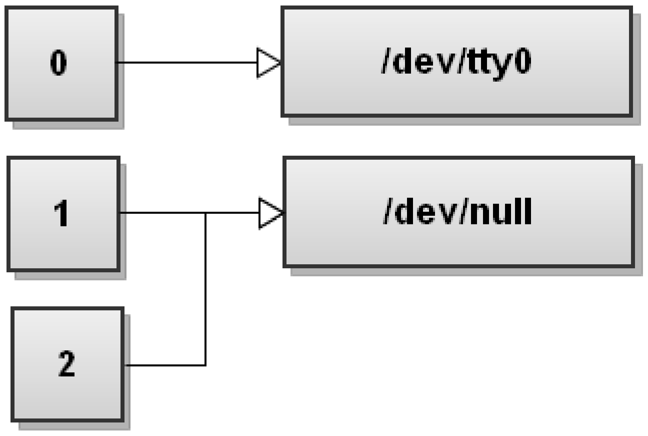
Similarly, by combining the previous one-liners, we can discard both stdout and stderr by doing:

```
$ command >/dev/null 2>&1
```

Or just simply:

```
$ command &>/dev/null
```

File descriptor table for this feat looks like this:

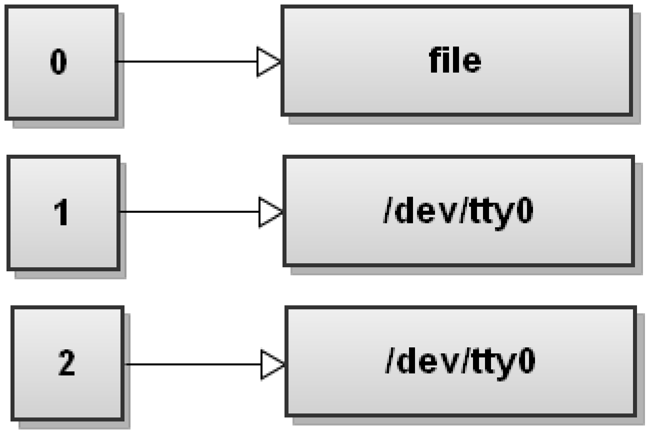


5. Redirect the contents of a file to the stdin of a command

```
$ command <file
```

Here bash tries to open the file for reading before running any commands. If opening the file fails, bash quits with error and doesn't run the command. If opening the file succeeds, bash uses the file descriptor of the opened file as the stdin file descriptor for the command.

After doing that the file descriptor table looks like this:



Here is an example. Suppose you want to read the first line of the file in a variable. You can simply do this:

```
$ read -r line < file
```

Bash's built-in `read` command reads a single line from standard input. By using the input redirection operator `<` we set it up to read the line from the file.