Illustrated Redirection Tutorial

This tutorial is not a complete guide to redirection, it will not cover here docs, here strings, name pipes etc... I just hope it'll help you to understand what things like `3>&2`, `2>&1` or `1>&3-` do.

stdin, stdout, stderr

When Bash starts, normally, 3 file descriptors are opened, `0`, `1` and `2` also known as standard input (`stdin`), standard output (`stdout`) and standard error (`stderr`).

For example, with Bash running in a Linux terminal emulator, you'll see:

```
# lsof +f g -ap $BASHPID -d 0,1,2
COMMAND   PID USER   FD   TYPE FILE-FLAG DEVICE SIZE/OFF NODE NAME
bash    12135 root    0u   CHR     RW,LG 136,13      0t0   16 /dev/pts/5
bash    12135 root    1u   CHR     RW,LG 136,13      0t0   16 /dev/pts/5
bash    12135 root    2u   CHR     RW,LG 136,13      0t0   16 /dev/pts/5
```

This `/dev/pts/5` is a pseudo terminal used to emulate a real terminal. Bash reads (`stdin`) from this terminal and prints via `stdout` and `stderr` to this terminal.

```
                  ---       +-----------------------+
standard input   ( 0 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard output  ( 1 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard error   ( 2 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+
```

When a command, a compound command, a subshell etc. is executed, it inherits these file descriptors. For instance `echo foo` will send the text `foo` to the file descriptor `1` inherited from the shell, which is connected to `/dev/pts/5`.

Simple Redirections

Output Redirection "n> file"

`>` is probably the simplest redirection.

```
echo foo > file
```

the `> file` after the command alters the file descriptors belonging to the command `echo`. It changes the file descriptor `1` (`> file` is the same as `1>file`) so that it points to the file `file`. They will look like:

```
                  ---       +-----------------------+
standard input   ( 0 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard output  ( 1 ) ---->| file                  |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard error   ( 2 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+
```

Now characters written by our command, `echo`, that are sent to the standard output, i.e., the file descriptor `1`, end up in the file named `file`.

In the same way, command `2> file` will change the standard error and will make it point to `file`. Standard error is used by applications to print errors.

What will `command 3> file` do? It will open a new file descriptor pointing to `file`. The command will then start with:

```
                  ---       +-----------------------+
standard input   ( 0 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard output  ( 1 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard error   ( 2 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+

                  ---       +-----------------------+
new descriptor   ( 3 ) ---->| file                  |
                  ---       +-----------------------+
```

What will the command do with this descriptor? It depends. Often nothing. We will see later why we might want other file descriptors.

Input Redirection "n< file"

When you run a commandusing `command < file`, it changes the file descriptor `0` so that it looks like:

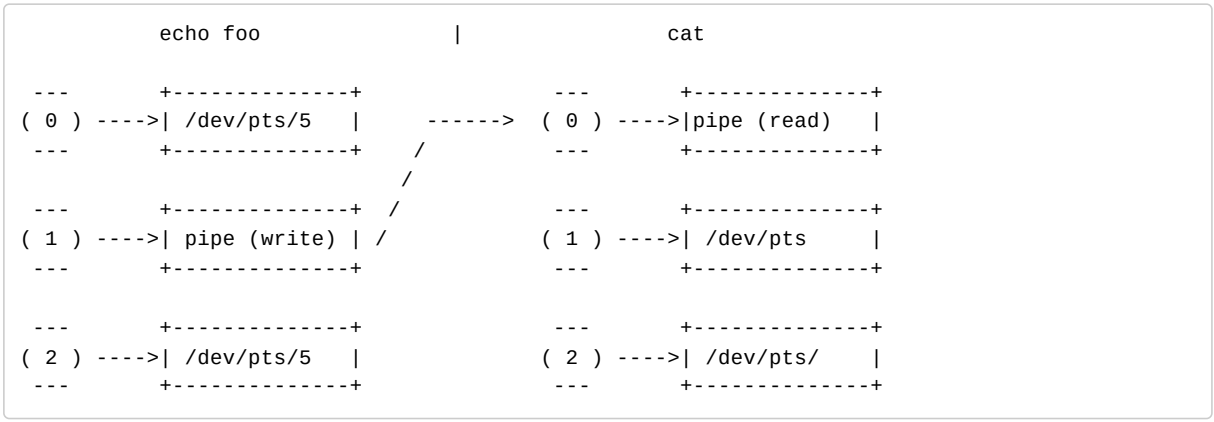
```
                  ---       +-----------------------+
standard input   ( 0 ) <----| file                  |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard output  ( 1 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard error   ( 2 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+
```

If the command reads from `stdin`, it now will read from `file` and not from the console.

As with `>`, `<` can be used to open a new file descriptor for reading, `command 3<file`. Later we will see how this can be useful.

Pipes |

What does this `|` do? Among other things, it connects the standard output of the command on the left to the standard input of the command on the right. That is, it creates a special file, a pipe, which is opened as a write destinaton for the left command, and as a read source for the right command.

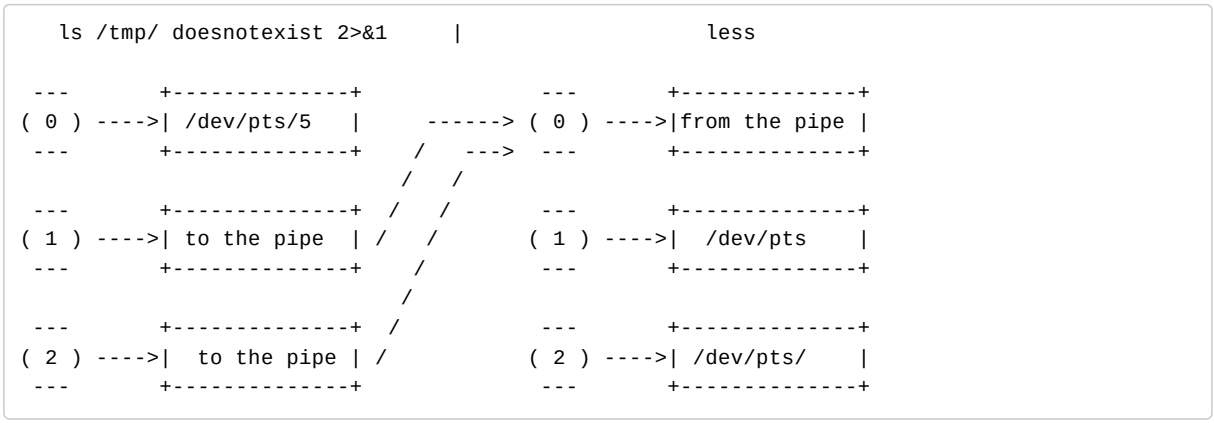


This is possible because the redirections are set up by the shell **before** the commands are executed, and the commands inherit the file descriptors.

More On File Descriptors

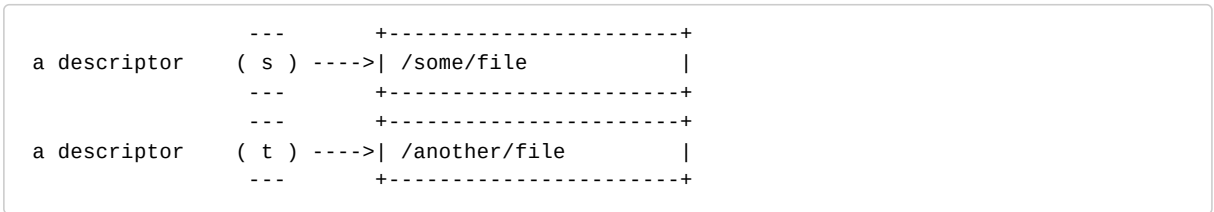
Duplicating File Descriptor 2>&1

We have seen how to open (or redirect) file descriptors. Let us see how to duplicate them, starting with the classic `2>&1` . What does this mean? That something written on the file descriptor `2` will go where file descriptor `1` goes. In a shell `command 2>&1` is not a very interesting example so we will use `ls /tmp/ doesnotexist 2>&1 | less`



Why is it called *duplicating*? Because after `2>&1` , we have 2 file descriptors pointing to the same file. Take care not to call this "File Descriptor Aliasing"; if we redirect `stdout` after `2>&1` to a file `B` , file descriptor `2` will still be opened on the file `A` where it was. This is often misunderstood by people wanting to redirect both standard input and standard output to the file. Continue reading for more on this.

So if you have two file descriptors `s` and `t` like:



Using a `t>&s` (where `t` and `s` are numbers) it means:

> Copy whatever file descriptor `s` contains into file descriptor `t`

So you got a copy of this descriptor:

```
                  ---       +-----------------------+
 a descriptor    ( s ) ---->| /some/file            |
                  ---       +-----------------------+
                  ---       +-----------------------+
 a descriptor    ( t ) ---->| /some/file            |
                  ---       +-----------------------+
```

Internally each of these is represented by a file descriptor opened by the operating system's `fopen` calls, and is likely just a pointer to the file which has been opened for reading (`stdin` or file descriptor `0`) or writing (`stdout` / `stderr`).

Note that the file reading or writing positions are also duplicated. If you have already read a line of `s` , then after `t>&s` if you read a line from `t` , you will get the second line of the file.

Similarly for output file descriptors, writing a line to file descriptor `s` will append a line to a file as will writing a line to file descriptor `t` .

The syntax is somewhat confusing in that you would think that the arrow would point in the direction of the copy, but it's reversed. So it's `target>&source` effectively.
So, as a simple example (albeit slightly contrived), is the following:

```
exec 3>&1         # Copy 1 into 3
exec 1> logfile   # Make 1 opened to write to logfile
lotsa_stdout      # Outputs to fd 1, which writes to logfile
exec 1>&3         # Copy 3 back into 1
echo Done         # Output to original stdout
```

Order Of Redirection, i.e., "> file 2>&1" vs. "2>&1 >file"

While it doesn't matter where the redirections appears on the command line, their order does matter. They are set up from left to right.

- `2>&1 >file`

A common error, is to do `command 2>&1 > file` to redirect both `stderr` and `stdout` to `file` . Let's see what's going on. First we type the command in our terminal, the descriptors look like this:

```
                  ---       +-----------------------+
standard input   ( 0 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard output  ( 1 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard error   ( 2 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+
```

Then our shell, Bash sees `2>&1` so it duplicates 1, and the file descriptor look like this:

```
                  ---       +-----------------------+
standard input   ( 0 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard output  ( 1 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard error   ( 2 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+
```

That's right, nothing has changed, 2 was already pointing to the same place as 1. Now Bash sees `> file` and thus changes `stdout` :

```
                  ---       +-----------------------+
standard input   ( 0 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard output  ( 1 ) ---->| file                  |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard error   ( 2 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+
```

And that's not what we want.

- `>file 2>&1`

Now let's look at the correct `command >file 2>&1` . We start as in the previous example, and Bash sees `> file` :

```
                  ---       +-----------------------+
standard input   ( 0 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard output  ( 1 ) ---->| file                  |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard error   ( 2 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+
```

Then it sees our duplication `2>&1` :

```
                  ---       +-----------------------+
standard input   ( 0 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard output  ( 1 ) ---->| file                  |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard error   ( 2 ) ---->| file                  |
                  ---       +-----------------------+
```

And voila, both `1` and `2` are redirected to file.

Why sed 's/foo/bar/' file >file Doesn't Work

This is a common error, we want to modify a file using something that reads from a file and writes the result to `stdout` . To do this, we redirect stdout to the file we want to modify. The problem here is that, as we have seen, the redirections are setup before the command is actually executed.

So **BEFORE** sed starts, standard output has already been redirected, with the additional side effect that, because we used >, "file" gets truncated. When `sed` starts to read the file, it contains nothing.

exec

In Bash the `exec` built-in replaces the shell with the specified program. So what does this have to do with redirection? `exec` also allow us to manipulate the file descriptors. If you don't specify a program, the redirection after `exec` modifies the file descriptors of the current shell.

For example, all the commands after `exec 2>file` will have file descriptors like:

```
                  ---       +-----------------------+
standard input   ( 0 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard output  ( 1 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard error   ( 2 ) ---->| file                  |
                  ---       +-----------------------+
```

All the the errors sent to `stderr` by the commands after the `exec 2>file` will go to the file, just as if you had the command in a script and ran `myscript 2>file`.

`exec` can be used, if, for instance, you want to log the errors the commands in your script produce, just add `exec 2>myscript.errors` at the beginning of your script.

Let's see another use case. We want to read a file line by line, this is easy, we just do:

```
while read -r line;do echo "$line";done < file
```

Now, we want, after printing each line, to do a pause, waiting for the user to press a key:

```
while read -r line;do echo "$line"; read -p "Press any key" -n 1;done < file
```

And, surprise this doesn't work. Why? because the shell descriptor of the while loop looks like:

```
                  ---       +-----------------------+
standard input   ( 0 ) ---->| file                  |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard output  ( 1 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard error   ( 2 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+
```

and our read inherits these descriptors, and our command (`read -p "Press any key" -n 1`) inherits them, and thus reads from file and not from our terminal.

A quick look at `help read` tells us that we can specify a file descriptor from which `read` should read. Cool. Now let's use `exec` to get another descriptor:

```
exec 3<file
while read -u 3 line;do echo "$line"; read -p "Press any key" -n 1;done
```

Now the file descriptors look like:

```
                  ---       +-----------------------+
standard input   ( 0 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard output  ( 1 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+

                  ---       +-----------------------+
standard error   ( 2 ) ---->| /dev/pts/5            |
                  ---       +-----------------------+

                  ---       +-----------------------+
new descriptor   ( 3 ) ---->| file                  |
                  ---       +-----------------------+
```

and it works.

Closing The File Descriptors

Closing a file through a file descriptor is easy, just make it a duplicate of -. For instance, let's close `stdin <&-` and `stderr 2>&-`:

```
bash -c '{ lsof -a -p $$ -d0,1,2 ;} <&- 2>&-'
COMMAND   PID USER   FD   TYPE DEVICE SIZE NODE NAME
bash    10668 pgas    1u   CHR  136,2         4 /dev/pts/2
```

we see that inside the `{}` that only `1` is still here.

Though the OS () will probably clean up the mess, it is perhaps a good idea to close the file descriptors you open. For instance, if you open a file descriptor with `exec 3>file`, all the commands afterwards will inherit it. It's probably better to do something like:

```
exec 3>file
.....
#commands that uses 3
.....
exec 3>&-

#we don't need 3 any more
```

I've seen some people using this as a way to discard, say stderr, using something like: command 2>&-. Though it might work, I'm not sure if you can expect all applications to behave correctly with a closed stderr.

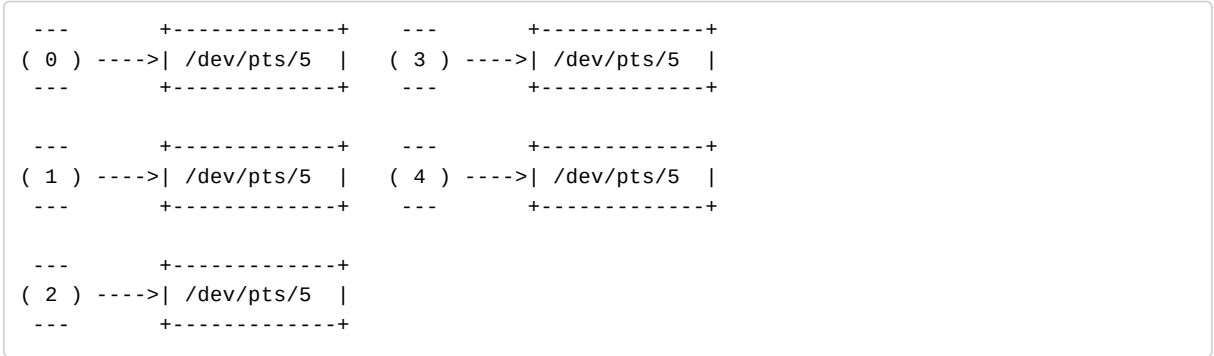
When in doubt, I use 2>/dev/null.

An Example

This example comes from this post (ffe4c2e382034ed9) (http://groups.google.com/group/comp.unix.shell/browse_thread/thread/64206d154894a4ef/ffe4c2e382034ed9#ffe4c2e382034ed9) on the comp.unix.shell group:

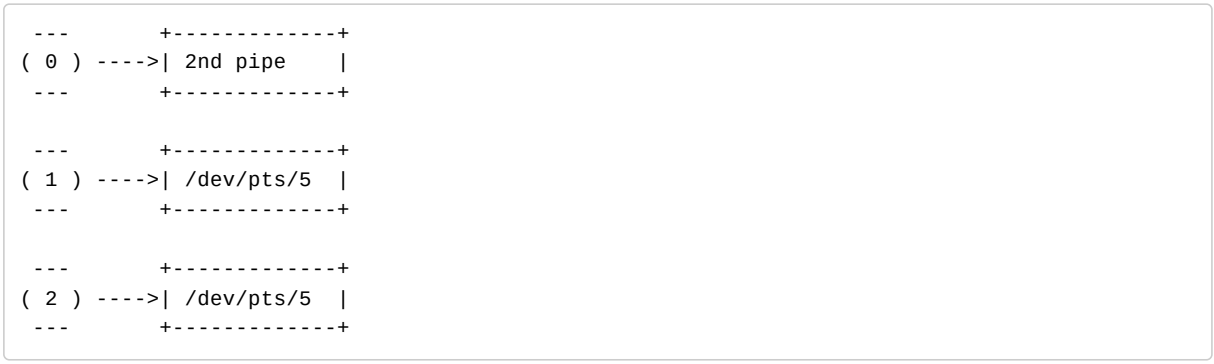
```
{
  {
    cmd1 3>&- |
      cmd2 2>&3 3>&-
  } 2>&1 >&4 4>&- |
    cmd3 3>&- 4>&-

} 3>&2 4>&1
```

The redirections are processed from left to right, but as the file descriptors are inherited we will also have to work from the outer to the inner contexts. We will assume that we run this command in a terminal. Let's start with the outer `{ } 3>&2 4>&1`.



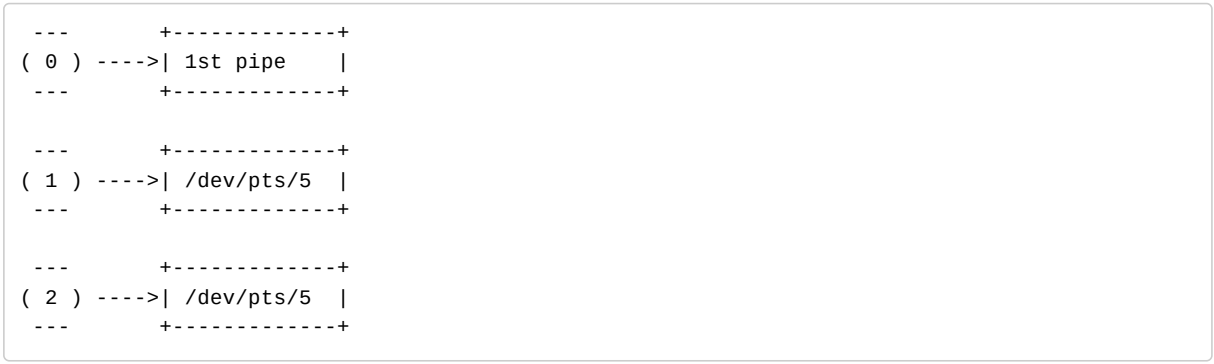
We only made 2 copies of `stderr` and `stdout`. `3>&1 4>&1` would have produced the same result here because we ran the command in a terminal and thus `1` and `2` go to the terminal. As an exercise, you can start with `1` pointing to `file.stdout` and 2 pointing to `file.stderr`, you will see why these redirections are very nice.

Let's continue with the right part of the second pipe: `| cmd3 3>&- 4>&-`

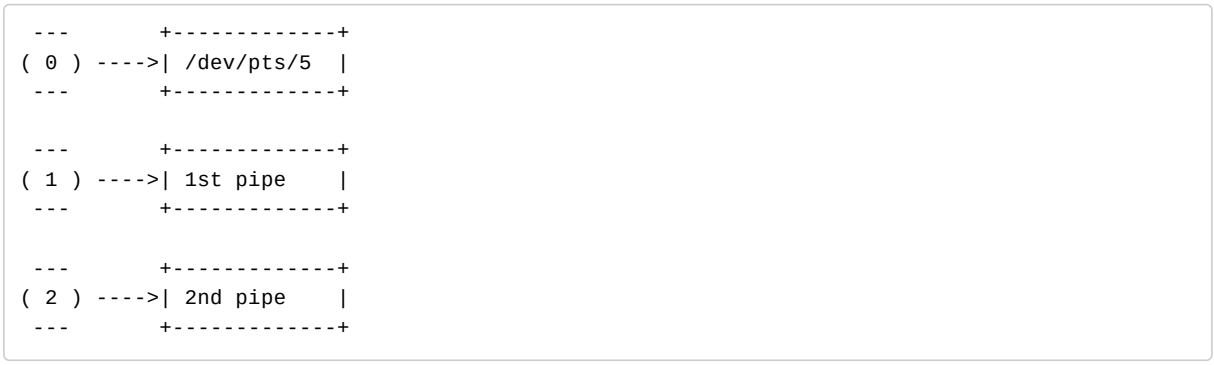


It inherits the previous file descriptors, closes 3 and 4 and sets up a pipe for reading. Now for the left part of the second pipe `{…} 2>&1 >&4 4>&- |`

First, The file descriptor `1` is connected to the pipe (`|`), then `2` is made a copy of `1` and thus is made an fd to the pipe (`2>&1`), then `1` is made a copy of `4` (`>&4`), then `4` is closed. These are the file descriptors of the inner `{}`. Lcet's go inside and have a look at the right part of the first pipe: `| cmd2 2>&3 3>&-`

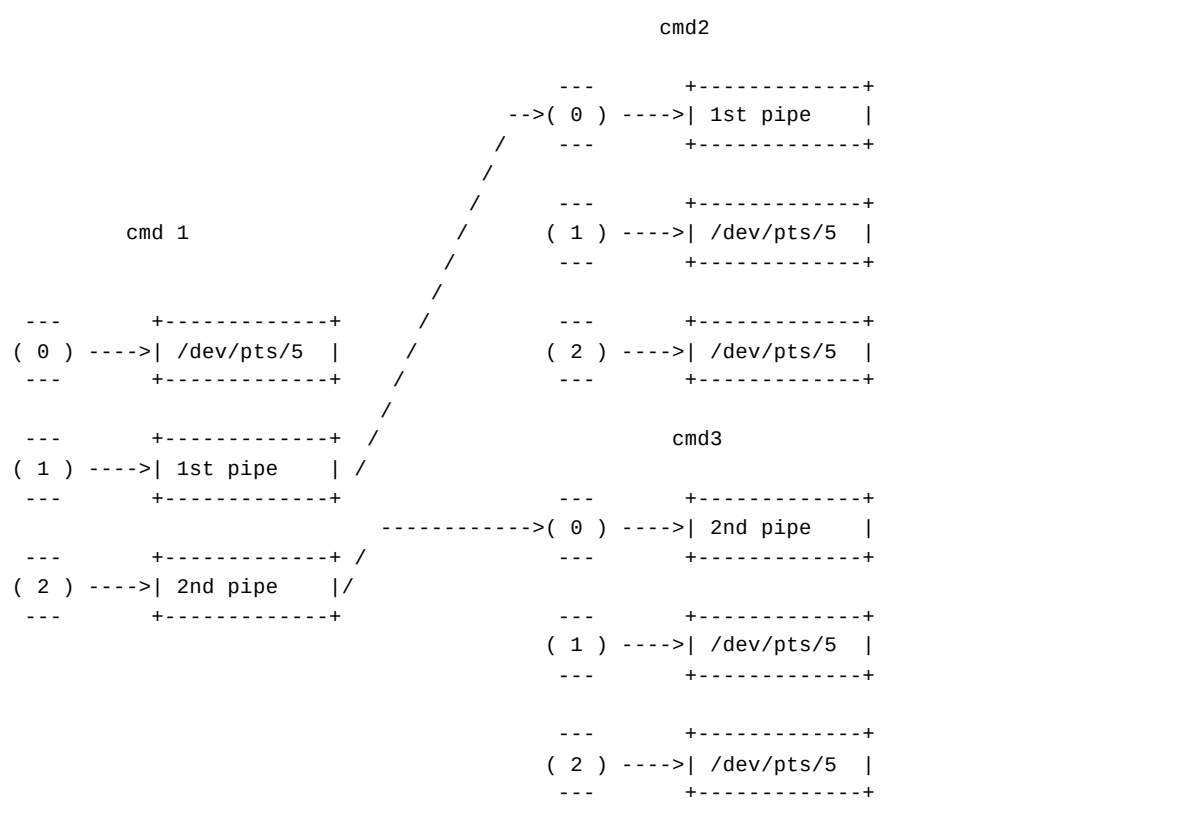


It inherits the previous file descriptors, connects 0 to the 1st pipe, the file descriptor 2 is made a copy of 3, and 3 is closed. Finally, for the left part of the pipe:



It also inherits the file descriptor of the left part of the 2nd pipe, file descriptor `1` is connected to the first pipe, `3` is closed.

The purpose of all this becomes clear if we take only the commands:



As said previously, as an exercise, you can start with `1` open on a file and `2` open on another file to see how the `stdin` from `cmd2` and `cmd3` goes to the original `stdin` and how the `stderr` goes to the original `stderr`.

Syntax

I used to have trouble choosing between `0&<3 3&>1 3>&1 ->2 -<&0 &-<0 0<&-` etc... (I think probably because the syntax is more representative of the result, i.e., the redirection, than what is done, i.e., opening, closing, or duplicating file descriptors).

If this fits your situation, then maybe the following "rules" will help you, a redirection is always like the following:

```
lhs op rhs
```

- `lhs` is always a file description, i.e., a number:
  - Either we want to open, duplicate, move or we want to close. If the op is `<` then there is an implicit 0, if it's `>` or `>>`, there is an implicit 1.
- `op` is `<`, `>`, `>>`, `>|`, or `<>`:
  - `<` if the file decriptor in `lhs` will be read, `>` if it will be written, `>>` if data is to be appended to the file, `>|` to overwrite an existing file or `<>` if it will be both read and written.
- `rhs` is the thing that the file descriptor will describe:
  - It can be the name of a file, the place where another descriptor goes (`&1`), or, `&-`, which will close the file descriptor.

You might not like this description, and find it a bit incomplete or inexact, but I think it really helps to easily find that, say `&->0` is incorrect.

A note on style

The shell is pretty loose about what it considers a valid redirect. While opinions probably differ, this author has some (strong) recommendations:

- **Always** keep redirections "tightly grouped" – that is, **do not** include whitespace anywhere within the redirection syntax except within quotes if required on the RHS (e.g. a filename that contains a space). Since shells fundamentally use whitespace to delimit fields in general, it is visually much clearer for each redirection to be separated by whitespace, but grouped in chunks that contain no unnecessary whitespace.

- **Do** always put a space between each redirection, and between the argument list and the first redirect.
- **Always** place redirections together at the very end of a command after all arguments. Never precede a command with a redirect. Never put a redirect in the middle of the arguments.
- **Never** use the Csh `&>foo` and `>&foo` shorthand redirects. Use the long form `>foo 2>&1` . (see: obsolete)

```
# Good! This is clearly a simple commmand with two arguments and 4 redirections
cmd arg1 arg2 <myFile 3<&1 2>/dev/null >&2

# Good!
{ cmd1 <<<'my input'; cmd2; } >someFile

# Bad. Is the "1" a file descriptor or an argument to cmd? (answer: it's the FD). Is the spa
ce after the herestring part of the input data? (answer: No).
# The redirects are also not delimited in any obvious way.
cmd 2>& 1 <<< stuff

# Hideously Bad. It's difficult to tell where the redirects are and whether they're even val
id redirects.
# This is in fact one command with one argument, an assignment, and three redirects.
foo=bar<baz bork<<< blarg>bleh
```

Conclusion

I hope this tutorial worked for you.

I lied, I did not explain `1>&3-` , go check the manual 😉

Thanks to Stéphane Chazelas from whom I stole both the intro and the example….

The intro is inspired by this introduction, you'll find a nice exercise there too:

- A Detailed Introduction to I/O and I/O Redirection (http://tldp.org/LDP/abs/html/ioredirintro.html)

The last example comes from this post:

- comp.unix.shell: piping stdout and stderr to different processes (http://groups.google.com/group/comp.unix.shell/browse_thread/thread/64206d154894a4ef/ffe4c2e382034ed9#ffe

See also

- Internal: Redirection syntax overview

Discussion

Armin, 2010/04/28 10:24 (), 2010/12/16 00:25 ()

Hello,

thank you for that comprehensive explanation.

I was looking for a solution for the following problem: I want to execute a shell script (both remotely via RSH and locally). The output from stdout and stderr should go to a file, to see the scripts progress at the terminal I wanted to redirect the output of some echo commands to the same file and to the terminal. Based on this tutorial I implemented the following solution ~~(I don't know how to produce an ampersand, therefore I use "amp;" instead)~~:

```
# save stdout, redirect stdout and stderr to a file
exec 3>&1 1>logfile 2>&1

# do something which creates output to stdout and/or stderr
# "restore" stdout, issue a message both to the terminal and to the file
exec 1>&3
echo "my message" | tee -a logfile
exec 1>>logfile

# do something which creates output to stdout and/or stderr

exit
```

As soon as output to stderr happens it doesn't work as I expected. E.g.

```
exec 3>&1 1>logfile 2>&1
echo "Hello World"
ls filedoesnotexist
exec 1>&3
echo "my message" | tee -a logfile
ls filedoesnotexistyet
exec 1>>logfile
echo "Hello again"
ls filestilldoesnotexist
exit
```

results in the following content of file logfile:

```
Hello World
ls: filedoesnotexist: No such file or directory
ls: filedoesnotexistyet: No such file or directory
ls: filestilldoesnotexist: No such file or directory
```

I.e. the texts "my message" and "Hello again" have been overwritten by the stderr output of the ls commands.

If I change in the 1st exec to append stdout to logfile (`exec 3>&1 1>>logfile 2>&1`) the result is correct:

```
Hello World
ls: filedoesnotexist: No such file or directory
my message
ls: filedoesnotexistyet: No such file or directory
Hello again
ls: filestilldoesnotexist: No such file or directory
```

If I change the 7th line to `exec 1>>logfile` **`2>&1`** the result looks better, but is not correct (still the line with "my message" is missing):

```
Hello World
ls: filedoesnotexist: No such file or directory
ls: filedoesnotexistyet: No such file or directory
Hello again
ls: filestilldoesnotexist: No such file or directory
```

I finally implemented the following solution, which is maybe not so elegant but works:

```
exec 3>&1 4>&2 1>logfile 2>&1
echo "Hello World"
ls filedoesnotexist
exec 1>&3 2>&4
echo "my message" | tee -a logfile
ls filedoesnotexistyet
exec 1>>logfile 2>&1
echo "Hello again"
ls filestilldoesnotexist
exit
```

This has the effect that the error output of the 2nd ls goes to the terminal and not to the file which is absolutely ok for my case.

Can anybody explain what exactly happens? I assume it has something to with file pointers. What is the preferred solution of my problem?

Regards

Armin

P.S.: I have some problems with formatting, esp. with line feeds and empty lines.

Jan Schampera, 2010/04/28 20:02 (), 2010/12/16 00:26 ()

Try this. In short:

- no subsequent set/reset of filedescriptors
- `tee` gets a process substitution as output file, inside a `cat` and a redirection to FD1 (logfile)
- `tee` s standard output is redirected to FD3 (terminal/original stdout)

```
echo "my message" | tee >(cat >&1) >&3
```

This is a Bash specific thing.

For the wiki quirks: I surrounded your code with `<code>...</code>` tags. For the ampersand issue I have no solution, sorry. Seems to be a bug in this plugin. It only happens on "preview", but it works for the real view.

typedeaF, 2011/08/15 15:35 ()

I am looking to implement the features of Expect, with bash. That is, to design a wrapper program that will assign the called program to redirect its 0-2 to named pipes. The wrapper will then open the other end of the named pipes. Something like this:

```
exec 3<>pipe.out
exec 4<>pipe.in

( PS3="enter choice:"; select choice in one two three; do echo "you choose \"$choice\"";
done )0<&4 1>&3 2>&1

while read -u pipe.out line
do
    case $line in
    *choice:)
        echo "$line"
        read i; echo i >&4
        ;;
    EOF)
        echo "caught \"EOF\", exiting";
        break;;
    *)
        echo "$line"
        ;;
    esac
done
```

Of course, this doesn't work at all. The first problem is, when using a pipe, the process hangs until both ends of the pipe are established. The second part of the problem is that the bash built-in "read" returns on a newline or the option of N chars or delimiter X –neither of which would be useful in this case. So the input of the while loop never "sees" the "enter choice:" prompt, since there is no newline. There are other problems as well.

Is it possible to get Bash to do this? Any suggestions? Here is something that does work.

terminal 1:

```
(exec 3<pipe.out; while read -u 3 line; do case $line in *choice:) echo $line; echo "i s
hould do a read here";; EOF) echo "caught EOF"; break;; *) echo "$line"; esac; done)
```

terminal 2:

```
(PS3="enter choice:"; select choice in one two three; do echo "you choose $choice"; don
e)1>pipe.out 2>&1
```

In this case, the program continues when both ends connect to the pipe, but since we arent redirecting stdin from a pipe for the select, you have to enter the choice in terminal 2. You will also notice that even in this scenario, terminal 1 does not see the PS3 prompt since it does not return a newline.

Tony, 2012/02/10 00:41 (), 2012/02/10 05:35 ()

Hello,

Many thanks for the comprehensive tutorial. I have learned a great deal about redirection.

In my script, I want to redirect stderr to a file and both stderr and stdout to another file. I found this construction works but I don't quite understand how. Could you explain ?

```
((./cmd 2>&1 1>&3 | tee /tmp/stderr.log) 3>&1 1>&2) > /tmp/both.log 2>&1
```

Also, if I want to do the same in the script using exec to avoid this kind of redirection in every command in the script, what should I do ?

thanks

Tony

Jan Schampera, 2012/02/10 05:46 ()

You pump `STDERR` of the command to descriptor 1, so that it can be transported by the pipe and seen as input by the `tee` command. At the same time you redirect the original `STDOUT` to descriptor 3. The `tee` command writes your original standard error output to the file plus outputs it to its `STDOUT` .

Outside the whole construct you collect your original standard output (descriptor 3) and your original standard error output (descriptor 1 - through `tee`) to the normal descriptors (1 and 2), the rest is a simple file redirection for both descriptors.

In short, you use a third descriptor to switch a bypass throuch `tee` .

I don't know a global method (`exec` or thelike) off my head. I can imagine that you can hack something with process substitution, but I'm not sure.

jack, 2012/03/02 16:41 ()

Many thanks for these explanations! Just one point which confused me. In the example from comp.unix.shell, you wrote: "Now for the left part of the second pipe…" The illustration for the result confused me because I was assuming the fds where coming from the previous illustration, but I then understood that they come from their parent process and hence from the previous previous illustration. I think it would be a little bit clearer if you would put a label on each of your illustrations and make more explicit the transition from one illustration to another. Anyway, many thanks again.)jack(

R.W. Emerson II, 2012/12/09 16:30 ()

Pipes seem to introduce an extraneous line at EOF. Is this true? Try this:

```
declare tT="A\nB\nC\n"                    # Should have three lines here
echo -e "tT($tT)"                         # Three lines, confirmed
echo -e "sort($(sort <<< $tT))"           # Sort outputs three lines
echo -e "$tT" | sort                      # Sort outputs four lines
```

When three lines go into the pipe, four lines come out. The problem is not present in the here-string facility.

What a helpful and badly needed site! Thank you!

Jan Schampera, 2012/12/16 13:13 (), 2012/12/16 13:14 ()

I see those additional line coming from the previous echo:

```
bonsai@core:~$ echo -e "$tT"
A
B
C

bonsai@core:~$
```

It is the additional newline echo adds itself to finalize the output. In your first echo, this is the newline after the closing bracket.

You can verify it when you use `echo -n` (suppresses the newline echo itself generates)

Hans Ginzel, 2015/10/02 09:03 ()

Thank you for comprehensive manual.

Plase add this example, http://stackoverflow.com/questions/3141738/duplicating-stdout-to-stderr (http://stackoverflow.com/questions/3141738/duplicating-stdout-to-stderr).

echo foo |tee /dev/stderr

Are there better/cleaner solutions? It seems that /dev/stderr can have problem in cron.

Jan Schampera, 2015/10/21 04:51 ()

It's a functionality of the shell itself, the shell duplicates the relevant file descriptors when it sees those filenames. So it may depend on the shell (or shell compatibility level) you use in cron.