Linux kernel: sysctl

… configure kernel parameters at runtime.
The parameters available are those listed under /proc/sys/. **Procfs** is required for **sysctl** support in Linux. You can use **sysctl** to both read and write **sysctl** data.

sysctl [*options*] [variable[=value]] [...]
sysctl -p [file or regexp] [...]

Linux kernel: Table of sysctl interfaces

Class	Subsystem
abi	Execution domains and personalities
crypto	Cryptographic interfaces
debug	Kernel debugging interfaces
dev	Device specific information
fs	Global and specific filesystem tunables
kernel	Global kernel tunables
net	Network tunables
sunrpc	Sun Remote Procedure Call (NFS)
user	User Namespace limits
vm	Tuning and management of memory, buffer, and cache

Linux folder hierarchy: /etc

/etc	Host-specific system-wide configuration files There has been controversy over the meaning of the name itself. In early versions of the UNIX Implementation Document from Bell labs, /etc is referred to as the *etcetera directory*,[3] as this directory historically held everything that did not belong elsewhere (however, the FHS restricts /etc to static configuration files and may not contain binaries).[4] Since the publication of early documentation, the directory name has been re-explained in various ways. Recent interpretations include backronyms such as "Editable Text Configuration" or "Extended Tool Chest".[5]
/etc/opt	Configuration files for add-on packages that are stored in /opt/.
/etc/sgml	Configuration files, such as catalogs, for software that processes SGML.
/etc/X11	Configuration files for the X Window System, version 11.
/etc/xml	Configuration files, such as catalogs, for software that processes XML.

Linux folder hierarchy: /usr

/usr	*Secondary hierarchy* for read-only user data; contains the majority of (multi-)user utilities and applications.
/usr/bin	Non-essential command binaries (not needed in single user mode); for all users.
/usr/include	Standard include files.
/usr/lib	Libraries for the binaries in /usr/bin/ and /usr/sbin/.
/usr/lib<qual>	Alternate format libraries (optional).
/usr/local	*Tertiary hierarchy* for local data, specific to this host. Typically has further subdirectories, *e.g.*, bin/, lib/, share/.[8]
/usr/sbin	Non-essential system binaries, *e.g.*, daemons for various network-services.
/usr/share	Architecture-independent (shared) data.
/usr/src	Source code, *e.g.*, the kernel source code with its header files.
/usr/X11R6	X Window System, Version 11, Release 6 (up to FHS-2.3, optional).

Linux folder hierarchy: /var

/var	Variable files—files whose content is expected to continually change during normal operation of the system—such as logs, spool files, and temporary e-mail files.
/var/cache	Application cache data. Such data are locally generated as a result of time-consuming I/O or calculation. The application must be able to regenerate or restore the data. The cached files can be deleted without loss of data.
/var/lib	State information. Persistent data modified by programs as they run, *e.g.*, databases, packaging system metadata, etc.
/var/lock	Lock files. Files keeping track of resources currently in use.
/var/log	Log files. Various logs.
/var/mail	This is the location defined by the FHS (File System Hierarchy Standard) to store users' mailbox files. Depending on the extent to which your distribution complies with the FHS, these files may be located in /var/spool/mail.
/var/opt	Variable data from add-on packages that are stored in /opt/.
/var/run	Run-time variable data. This directory contains system information data describing the system since it was booted.[9] In FHS 3.0, /var/run is replaced by /run; a system should either continue to provide a /var/run directory, or provide a symbolic link from /var/run to /run, for backwards compatibility.
/var/spool	Spool for tasks waiting to be processed, *e.g.*, print queues and outgoing mail queue.
/var/spool/mail	Deprecated location for users' mailboxes
/var/tmp	Temporary files to be preserved between reboots.

Redirections

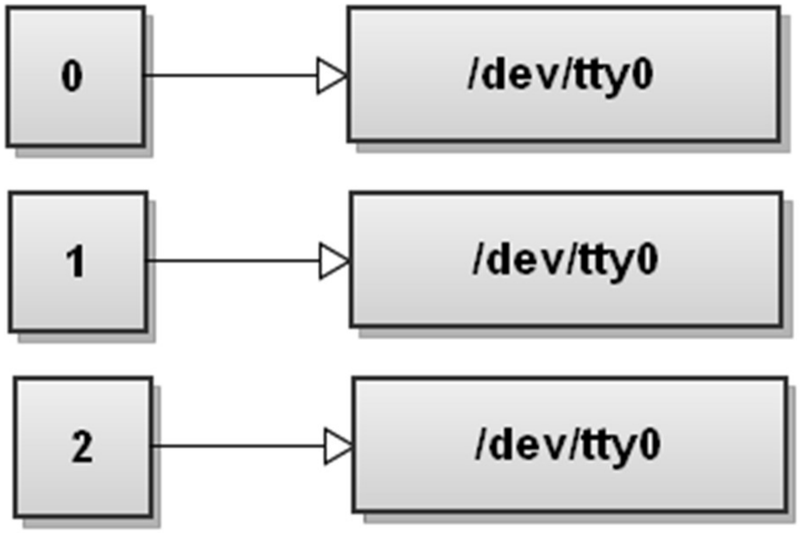
When bash starts it opens the three standard file descriptors:

- stdin (file descriptor **0**)
- stdout (file descriptor **1**)
- 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.

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).

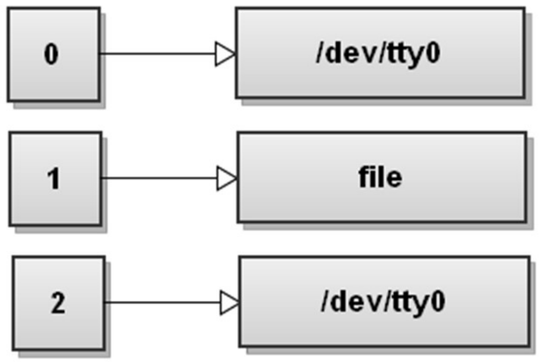
Redirect the standard output of a command to a file

$**command** >*file*

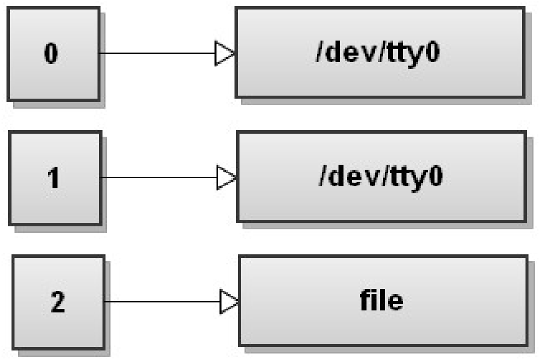
or

$**command 1**>*file*

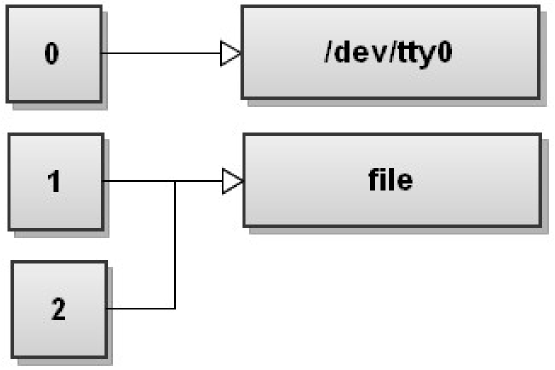
In general you can write **command** n>*file*, which will redirect the file descriptor n to *file*.



$ **command** 2>*file*
Bash redirects the ***stderr*** to *file*.
The number 2 stands for ***stderr***.



$ **command** &>*file*
Bash redirects the ***stderr*** to *file*.
The number 2 stands for ***stderr***.

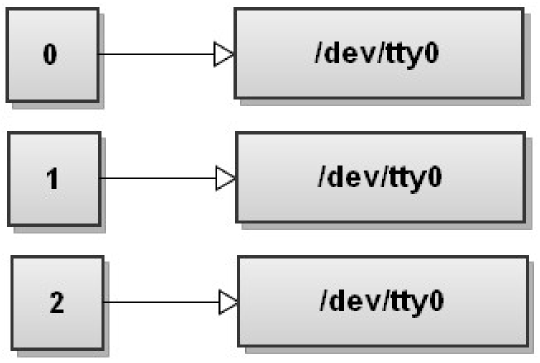


To redirect both streams to the same destination (1/2)

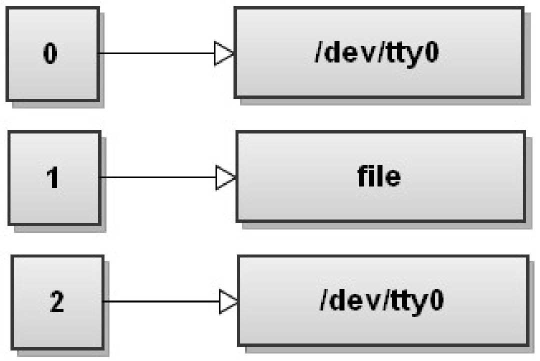
Redirect each stream one after another

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

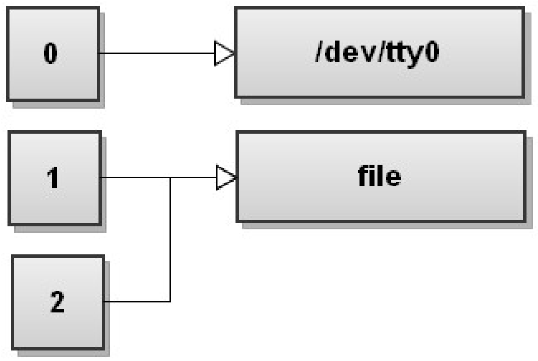
Before running any commands
bash's file descriptor table



processes the first redirection *>file*



second redirection 2>&1

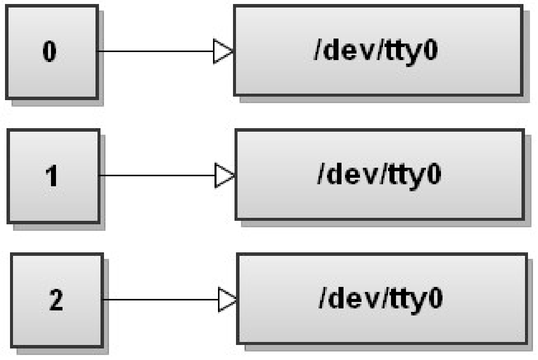


Several ways to redirect both streams to the same destination (2/2)

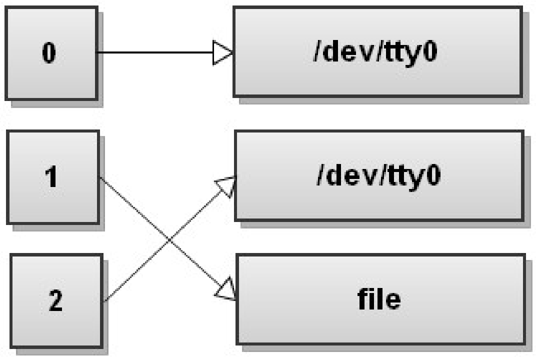
Redirect each stream one after another:

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

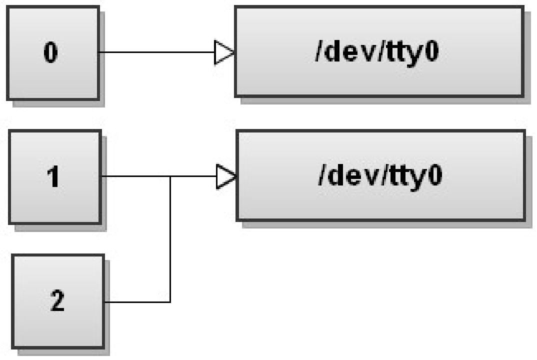
Before running any commands bash's file descriptor table



processes the first redirection >*file*



second redirection 2>&1



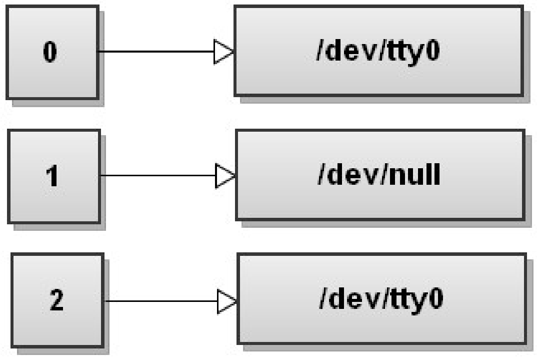
Stdout now points to **file** but the <u>**stderr** still points to the terminal!</u>

Discard the standard output of a command

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

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

The special file */dev/null* discards all data written to it.

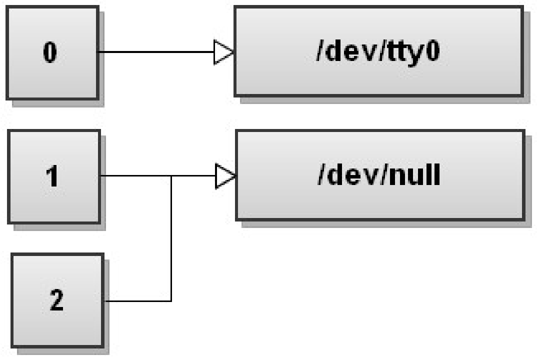


Discard both stdout and stderr by doing:

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

or

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



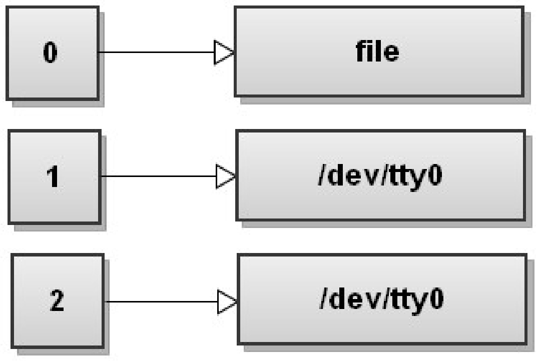
Redirect the contents of a file to the stdin of a command

$**command** < *file*

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.



$ **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**.

Redirect a single line of text to the stdin of a command

$ **command** <<< *"foo bar baz“*

let's say you quickly want to pass the text in your clipboard as the ***stdin*** to a command

$ **echo** "clipboard contents" **| command**

or

$ **command** <<< "clipboard contents"

Redirect stderr of all commands to a file forever

```
$ exec 2>file            # the built-in exec bash command
$ command1
$ command2
$ ...
```

If you specify redirects after it, then they will last forever, meaning until you change them or exit the script/shell.

In this case the 2>file redirect is setup that redirects the **stderr** of the current shell to the file. Running commands after setting up this redirect will have the **stderr** of all of them redirected to file.

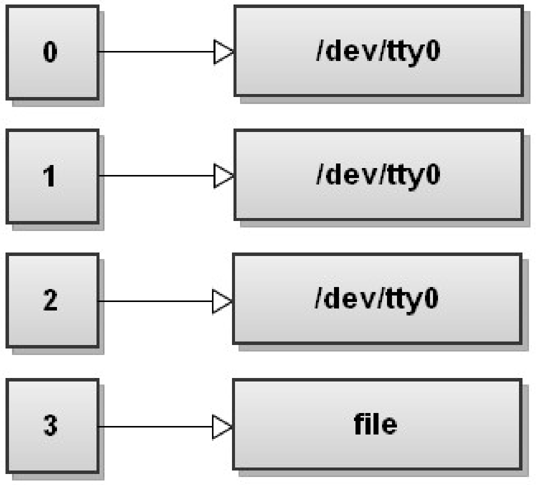
It's really useful in situations when you want to have a complete log of all errors that happened in the script, but you don't want to specify **2>***file* after every single command!

In general exec can take an optional argument of a command. If it's specified, bash replaces itself with the command. So what you get is only that command running, and there is no more shell.

Open a file for reading using a custom file descriptor

exec 3<*file*

What this does is opens the file for reading and assigns the opened file-descriptor to the shell's file descriptor number 3.



$ **exec 3>&-** # the file descriptor 3 is duped to -, which is bash's special way to say "close this fd"

Open a file for writing using a custom file descriptor

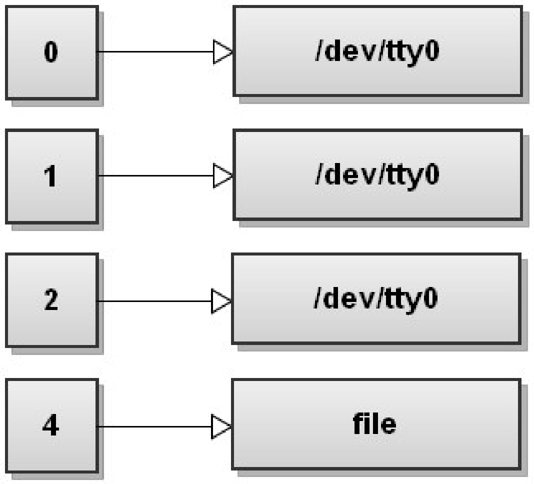
$exec 4>*file*

Tell bash to open file for writing and assign it number 4

File descriptor number from 0 to 255.

$echo "foo" >&4

$exec 4>&-   # close the file descriptor 4



11. OPEN A FILE BOTH FOR WRITING AND READING

Background image

```
$ exec 3<>file
```

The diamond operator <> opens a file descriptor for both reading and writing.

```
$ echo "foo bar" > file        # write string "foo bar" to file "file".
$ exec 5<> file                # open "file" for rw and assign it fd 5.
$ read -n 3 var <&5            # read the first 3 characters from fd 5.
$ echo $var
```

This will output foo as we just read the first 3 chars from the file.

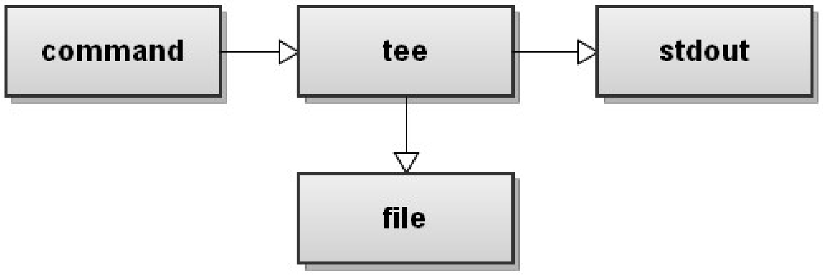
```
$ echo -n + >&5                # write "+" at 4th position.
$ exec 5>&-                    # close fd 5.
$ cat file
```

This will output foo+bar as we wrote the + char at 4th position in the file.

16. Redirect standard input to a file and print it to standard output

$ **command | tee** *file*

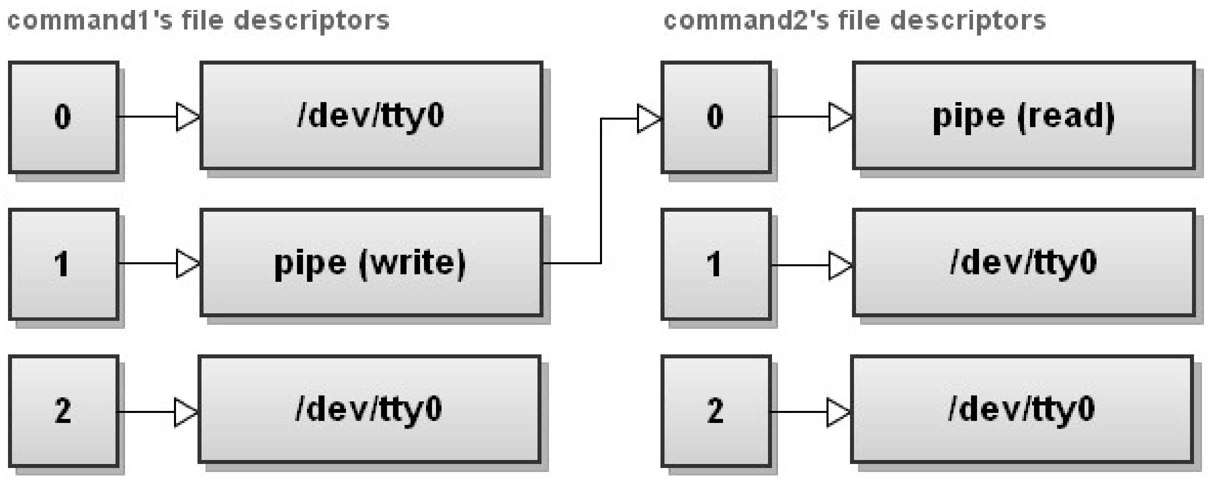
It takes an input stream and prints it both to standard output and to a file.



17. Send stdout of one process to stdin of another process

$ command1 | command2

A pipe connects **stdout** of **command1** with the **stdin** of **command2.**



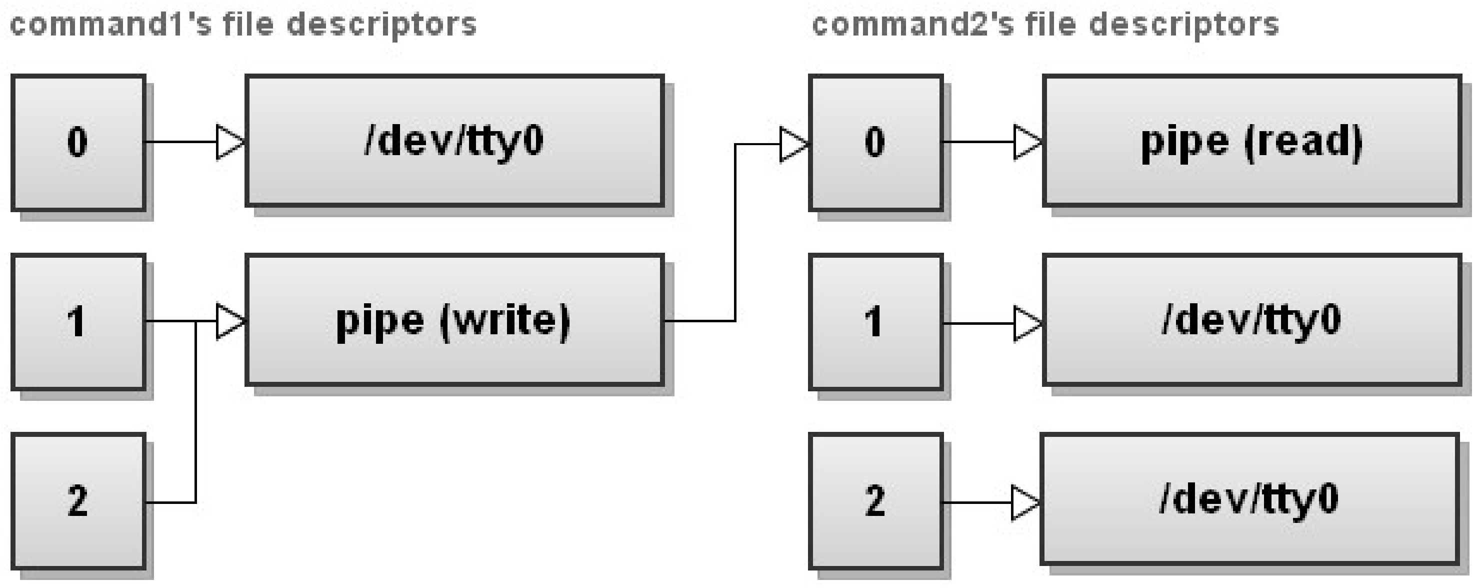
Everything sent to file descriptor **1** (**stdout**) of **command1** gets redirected through a pipe to file descriptor **0** (**stdin**) of **command2**.

man 2 pipe

Send stdout and stderr of one process to stdin of another process

```
$ command1 |& command2
```

*The |& redirection operator sends both **stdout** and **stderr** of **command1** over a pipe to **stdin** of **command2***



First command1's **stderr** is redirected to **stdout**, and then a pipe is setup between **command1**'s **stdout** and **command2**'s **stdin**.

19. Give file descriptors names

$ **exec** {*filew*}>*output_file*

Named file descriptors look like {***varname***}.

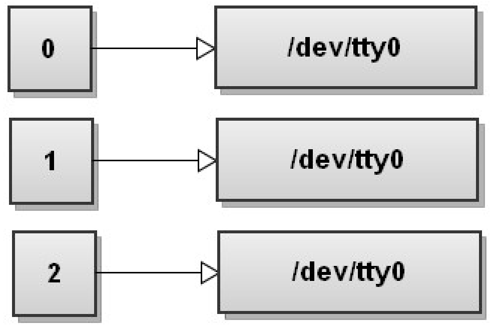
Bash internally chooses a free file descriptor and assigns it a name.

21. Swap stdout and stderr

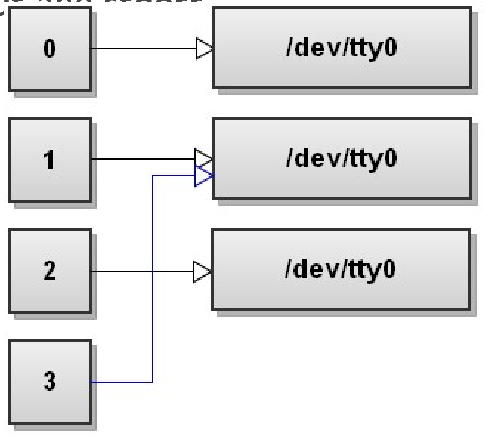
$ command 3>&1 1>&2 2>&3

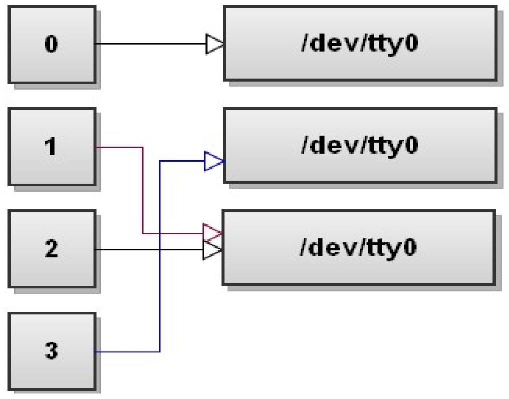
First duplicate file descriptor **3** to be a copy of **stdout**. Then duplicate **stdout** to be a copy of **stderr**
Finally duplicate **stderr** to be a copy of file descriptor **3**, which is **stdout.** In result swapped **stdout and stderr**

file descriptors pointing to the terminal:

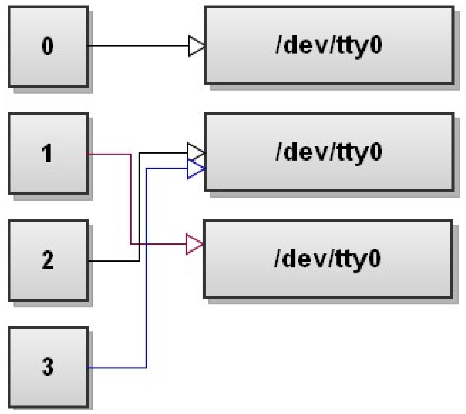


bash setups 3>&1 redirection





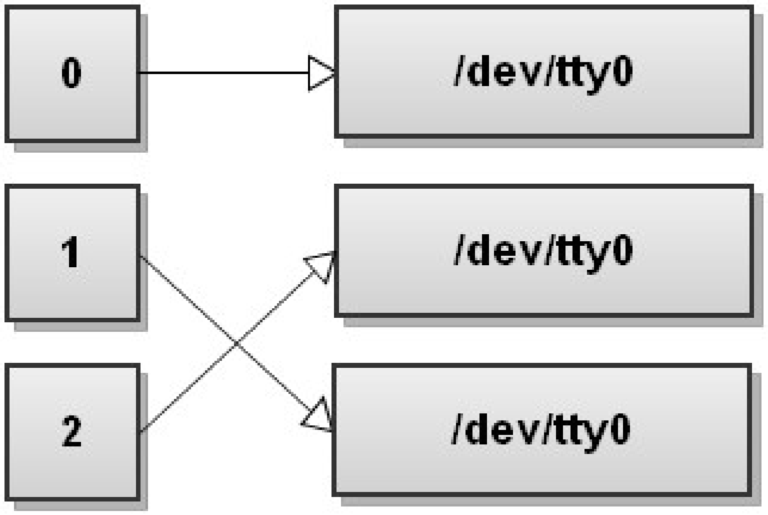
bash setups 1>&2 redirection



bash setups 2>&3 redirection

Swap stdout and stderr

```
$ command 3>&1 1>&2 2>&3 3>&-
```



22. Send stdout to one process and stderr to another process

```
$ command > >(stdout_cmd) 2> >(stderr_cmd)
```

The **>(...)** operator runs the commands in **...** with **stdin** connected to the read part of an anonymous named pipe.
Bash replaces the operator with the filename of the anonymous pipe

For example:
The first substitution **>(stdout_cmd)** might return **/dev/fd/60**, and the second substitution might return **/dev/fd/61.**
Both of these files are named pipes that bash created on the fly
Both named pipes have the commands as readers. The commands wait for someone to write to the pipes so they can read the data.

23. Find the exit codes of all piped commands

```
$ cmd1 | cmd2 | cmd3 | cmd4
```

Special **PIPESTATUS** array that saves the exit codes of all the commands in the pipe stream

```
$ pp
$ echo ${PIPESTATUS[@]}
0 1 0 0
```

Filters

… is class of programs you can use with pipes.

- Filters take standard input and perform an operation upon it and send the results to standard output. In this way, they can be used to process information in powerful ways. Here are some of the common programs that can act as filters:

Program	What it does
sort	Sorts standard input then outputs the sorted result on standard output.
uniq	Given a sorted stream of data from standard input, it removes duplicate lines of data (i.e., it makes sure that every line is unique).
grep	Examines each line of data it receives from standard input and outputs every line that contains a specified pattern of characters.
fmt	Reads text from standard input, then outputs formatted text on standard output.
pr	Takes text input from standard input and splits the data into pages with page breaks, headers and footers in preparation for printing.
head	Outputs the first few lines of its input. Useful for getting the header of a file.
tail	Outputs the last few lines of its input. Useful for things like getting the most recent entries from a log file.
tr	Translates characters. Can be used to perform tasks such as upper/lowercase conversions or changing line termination characters from one type to another (for example, converting DOS text files into Unix style text files).
sed	Stream editor. Can perform more sophisticated text translations than tr.
awk	An entire programming language designed for constructing filters. Extremely powerful.

Watch or Monitor Log Files in Real Time

tail [OPTION]... [*FILE*]...

Print the last 10 lines of each *FILE* to standard output. With more than one *FILE*, precede each with a header giving the file name. With no *FILE*, or when *FILE* is -, read standard input.

OPTION	DSCRIPTION
-c, --bytes=[+]*NUM*	output the last NUM bytes; or use -c +NUM to output starting with byte NUM of each file
-f, --follow[={*name*|*descriptor*}]	output appended data as the file grows; an absent option argument means '*descriptor*'
-F	same as --follow=name --retry
-n, --lines=[+]*NUM*	output the last NUM lines, instead of the last 10; or use -n +NUM to output starting with line NUM
--max-unchanged-stats=*N*	with --follow=name, reopen a FILE which has not changed size after N (default 5) iterations to see if it has been unlinked or renamed (this is the usual case of rotated log files); with inotify, this option is rarely useful
--pid=*PID*	with -f, terminate after process *ID*, *PID* dies
-q, --quiet, --silent	never output headers giving file names
--retry	keep trying to open a file if it is inaccessible
-s, --sleep-interval=*N*	with -f, sleep for approximately *N* seconds (default 1.0) between iterations; with inotify and --pid=*P*, check process P at least once every N seconds
-v, --verbose	always output headers giving file names
-z, --zero-terminated	line delimiter is NUL, not newline

Watch or Monitor Log Files in Real Time

The command tail needs the -f argument to follow the content of a file.

```
$ sudo tail -f /var/log/apache2/access.log
```

less Command – Display Real Time Output of Log Files

… can display the live output of a file if you type `Shift+F.`

1. 1. An opened file in less will start following the ***end of the file***.
2. For break display the live output of a file, you can type `Ctrl+C.`

Alternatively, you can also start less with less `+F` flag to enter to live watching of the file.