

UNIX-FU:

A FORKING PRESENTATION FOR DEVELOPERS

https://github.com/gregmalcolm/unix_for_programmers_demo

[git://gist.github.com/1156222.git](https://gist.github.com/1156222.git)

@gregmalcolm



Most rubyists develop for a unix platform. Unix as a platform is designed with programmers in mind. Today I'll cover a little bit about how to get the most from it.



1969 - UNIX is Born
1972 - Rewritten in C
1977 - BSD UNIX
1981 - IBM PC
1983 - Richard Stallman founds FSF
1991 - Linus Tovalds starts Linux

TIMELINE

1969 – Unix was invented at Bell Labs starting with Ken Thompson and shortly followed by Dennis Ritchie who invented C. Over 40 years ago!
1983 – US Government breaks up Bell System, FSF founded

PHILOSOPHY

Doug McIlroy (inventer of unix pipeline):

“This is the Unix philosophy:

Write programs that do one thing and do it well.

Write programs to work together.

*Write programs to handle text streams,
because that is a universal interface.”*

Doug McIlroy's take on the Unix Philosophy sums it up very nicely

PHILOSOPHY

Doug McIlroy (inventor of unix pipeline):

“This is the Unix philosophy:

Write programs that do one thing and do it well.

Write programs to work together.

*Write programs to handle text streams,
because that is a universal interface.”*

Doug McIlroy's take on the Unix Philosophy sums it up very nicely

PHILOSOPHY

Mike Gancarz (part of X Windows team):

- “
1. *Small is beautiful.*
 2. *Make each program do one thing well.*
 3. *Build a prototype as soon as possible.*
 4. *Choose portability over efficiency.*
 5. *Store data in flat text files.*
 6. *Use software leverage to your advantage.*
 7. *Use shell scripts to increase leverage and portability.*
 8. *Avoid captive user interfaces.*
 9. *Make every program a filter.*“

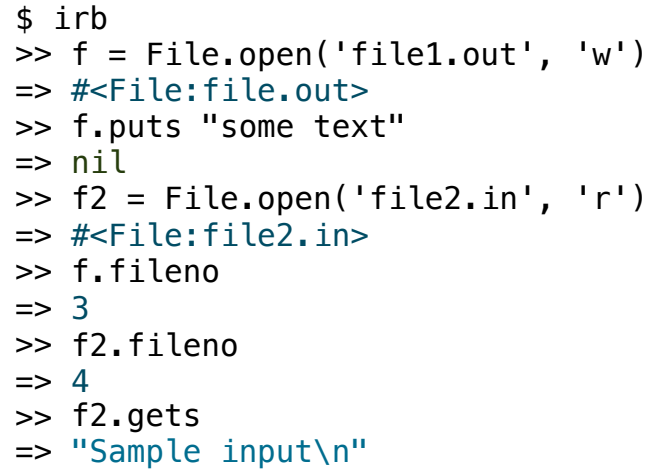
I also like Mike Gancarz's version.

Don't cross the streams!



STREAMS

File Streams



```
Terminal — bash — 84x24

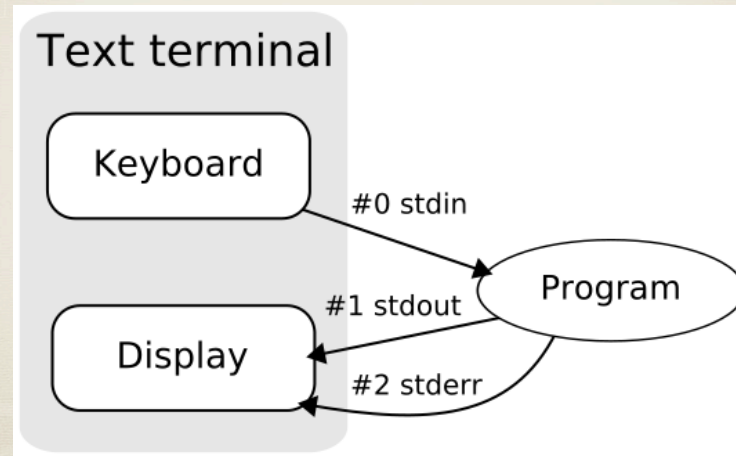
$ irb
>> f = File.open('file1.out', 'w')
=> #<File:file.out>
>> f.puts "some text"
=> nil
>> f2 = File.open('file2.in', 'r')
=> #<File:file2.in>
>> f.fileno
=> 3
>> f2.fileno
=> 4
>> f2.gets
=> "Sample input\n"
```

Firstly, there's file streams. I'll open a couple here, one for output, one another for input. Fairly basic stuff. Each file has a file descriptor. Here is how I can print them from ruby.

Now, why does the number start from 3? Answer is on the next page

STREAMS

Input/Output Streams

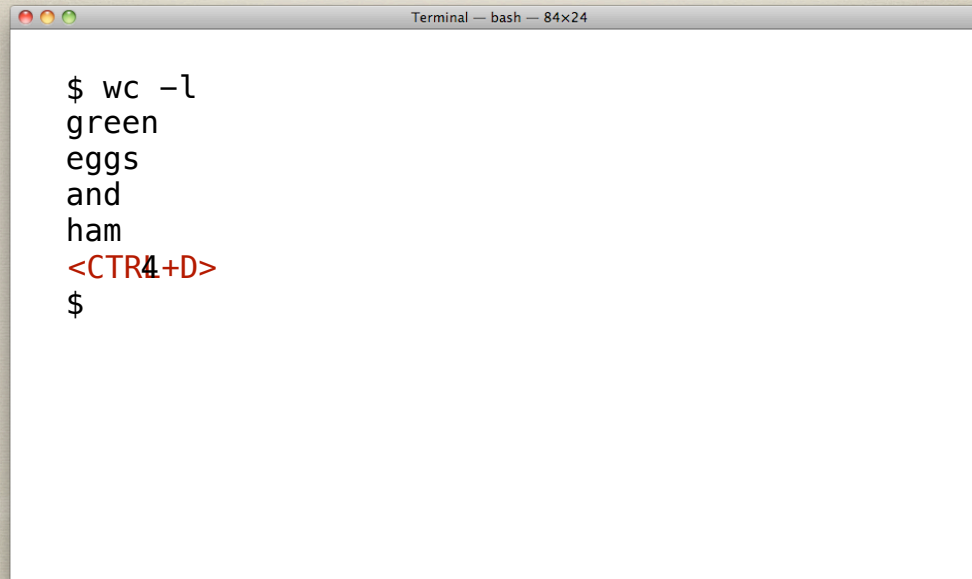


There are 3 special streams available in unix, STDIN, STDOUT and STDERR. These streams are never closed and use up File Descriptors 0, 1 and 2.

STDIN by default is for input (keyboard). STDOUT is for program output, usually going to the console. STDERR also defaults to outputting to the console, but it is easy to redirect to an error log.

STREAMS

STDIN

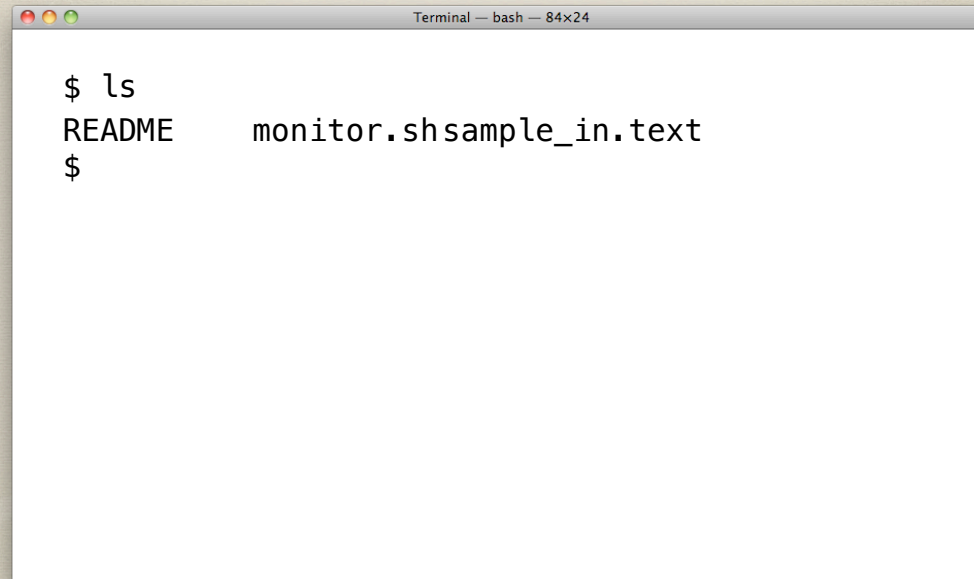
A terminal window titled "Terminal — bash — 84x24" is shown. It contains the following text:

```
$ wc -l  
green  
eggs  
and  
ham  
<CTRL+D>  
$
```

STDIN is usually inputted through the keyboard.
In this example I'm asking running the word count command. The `-l` tells it to report how many lines. `Ctrl+D` acts as an end of stream indication.

STREAMS

STDOUT

A screenshot of a macOS Terminal window. The title bar at the top reads "Terminal — bash — 84x24". The terminal content shows a prompt "\$" followed by the command "ls". The output of the command is displayed on the next line as "README" and "monitor.shsample_in.text". A second prompt "\$" is visible on the line following the output.

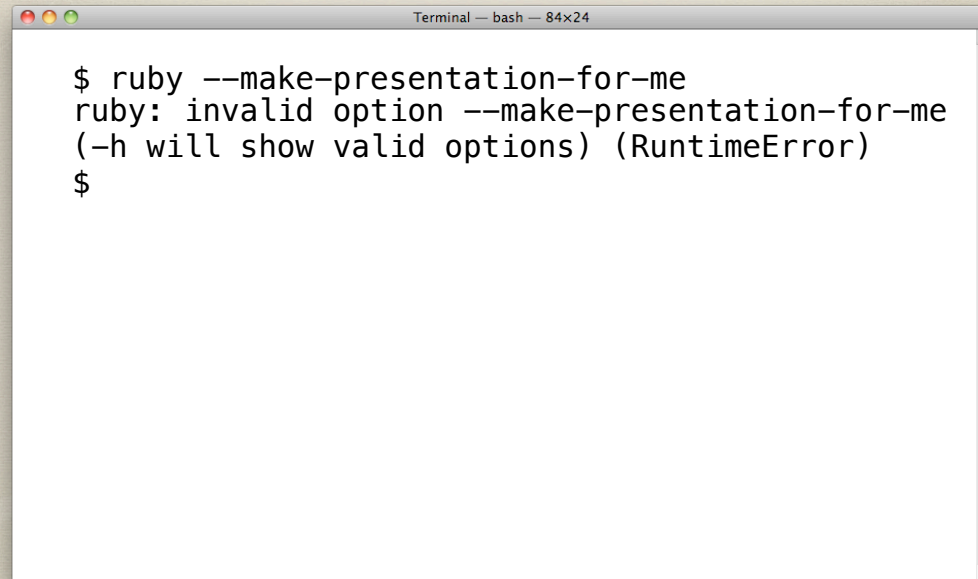
```
Terminal — bash — 84x24

$ ls
README      monitor.shsample_in.text
$
```

STDOUT by default outputs to the console. For example, if I run `ls` I'll get output to the console.

STREAMS

STDERR

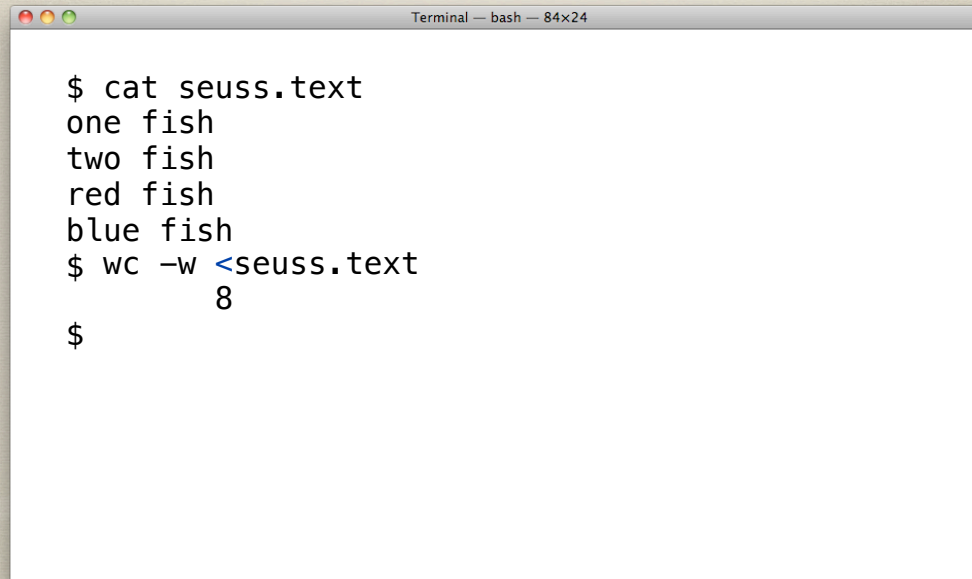
A screenshot of a terminal window titled "Terminal — bash — 84x24". The terminal shows a command prompt "\$" followed by the command "ruby --make-presentation-for-me". The output is an error message: "ruby: invalid option --make-presentation-for-me (-h will show valid options) (RuntimeError)". The prompt "\$" is shown again on the next line.

```
$ ruby --make-presentation-for-me
ruby: invalid option --make-presentation-for-me
(-h will show valid options) (RuntimeError)
$
```

Standard Error also outputs to the console by default. For example, if I run ruby with a bad option the message is sent via STDERR.

STREAMS

Redirecting STDIN

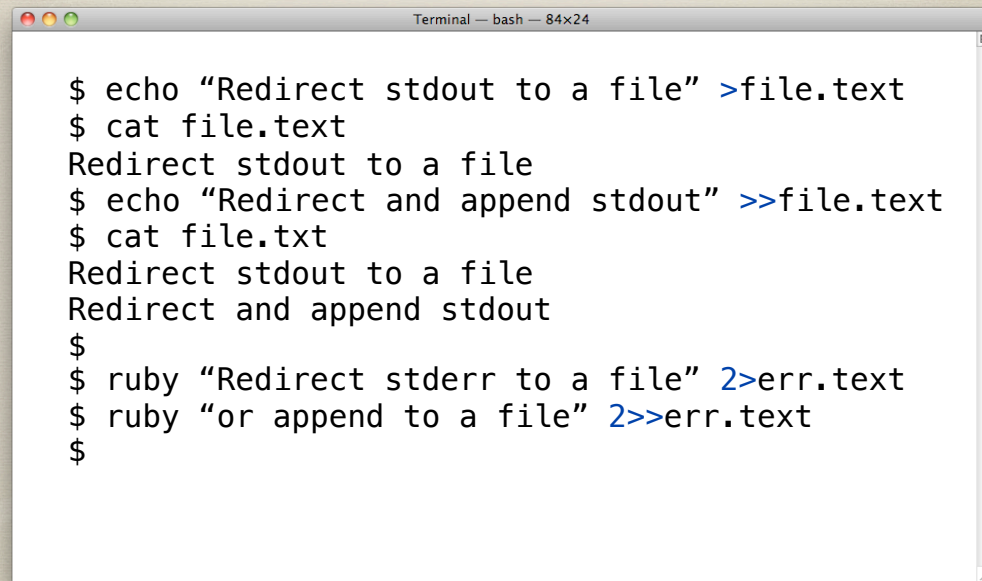
A terminal window titled "Terminal — bash — 84x24" is shown. It contains the following text:

```
$ cat seuss.text
one fish
two fish
red fish
blue fish
$ wc -w <seuss.text
      8
$
```

We can override the default streams through use of redirections. In this example we redirect to the Dr Seusse text into STDIN. By the way, this time we're using the `-w` option with `wc` to count words.

STREAMS

Redirecting STDOUT and STDERR



```
Terminal — bash — 84x24

$ echo "Redirect stdout to a file" >file.text
$ cat file.text
Redirect stdout to a file
$ echo "Redirect and append stdout" >>file.text
$ cat file.txt
Redirect stdout to a file
Redirect and append stdout
$
$ ruby "Redirect stderr to a file" 2>err.text
$ ruby "or append to a file" 2>>err.text
$
```

For stdout we can use `>` to redirect to a file. Or `>>` if we just to append to an existing file.

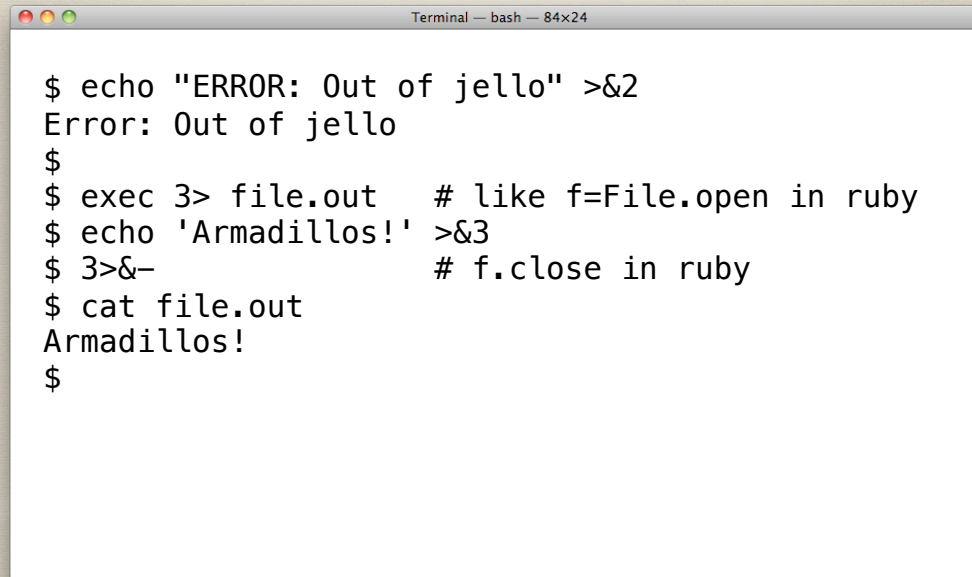
Similarly we can redirect to STDERR with `2>` or `2>>` (ruby doesn't know how to parse the argument)

Note you can stick a 1 in front of STDERR redirections and a 0 in front of STDIN, but its only when it comes to STDERR that you really have a need.

Btw, if you used `<<` for STDIN it actually does something a bit different. `cat << ENDWORD` will accept text until the word "ENDWORD" is encountered.

STREAMS

Redirecting to File Descriptors



```
Terminal — bash — 84x24

$ echo "ERROR: Out of jello" >&2
Error: Out of jello
$
$ exec 3> file.out      # like f=File.open in ruby
$ echo 'Armadillos!' >&3
$ 3>&-                  # f.close in ruby
$ cat file.out
Armadillos!
$
```

You can also redirect to open file descriptors.
To redirect an STDOUT to STDERR use `>&2` (ie redirect from File Descriptor #1 to File Descriptor #2)
On line 4 we open `File.out` into File Descriptor no 3. In this instance, using the word `exec` is like saying "For output in this current process...", so "For output in this current process redirect FD3 to fileout".
`3>&-` is kind of saying "Don't redirect FD3 to anything." In other words, close FD3.

56 PE UP KS is. *	1-operable, 2-fixed, low E class, no grids, \$750. pair, call 901-218- XXXX	WASHER clean, su more, wh make offe
SS ★ g mat- Wraps olesale 0- XXXX	DRYER, KENMORE, \$75 Heavy Duty, white, lg. capac- ity. Runs great. (662) 449- XXXX	WASHER - (cost \$1,0 Call ★★★★★
59	FORK , mangled, \$0.50. Also selling garbage disposal, used once, needs repair. 901-529- XXXX	WICKER F white, nig bookcase, ing, \$325. E
	FREEZER (7.5 cubic feet) - \$60; Dinnerware - Harvest Moon pattern, all white with fruit, place setting for 12- \$35; Lamps - \$5-\$25; Side Tables-	WINE STO Cedar int., prox 200 b 55" H x 38" W

Whenever you create a new process in UNIX you actually fork off from the current process. Meaning create a clone of the current process as a child process.

\$ wait 2012

PID = 1998

#<Targets FD=34>

#<Oracle FD=39>

#<TeddyBear FD=41>

\$ exec csi_duty.rb

PID = 2012

#<InappropriateCommentsAboutTheBody FD=43>

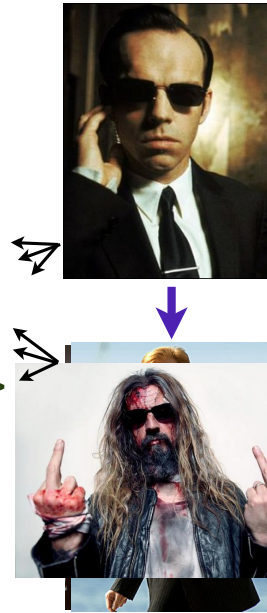


Illustration of how forking works: We start off with the Agent Smith program which we want to fork off from. We do so and end up with a child process which is an exact clone. Because its a clone it even has access to the same open File Descriptors. This can prove very useful! On forking we load a new program into memory over the top of the old Agent Smith program. So Agent Smith becomes Horace from CSI Miami by running "exec csi_duty.rb". Remember exec from the previous example? running a command through exec causes it to replace the current program with a new one. Eventually Horace will finish his work and exit. The process will actually close down, but there is still an entry in the process table. When in this state it is referred to as a Zombie Process (or a Rob Zombie process in this case?). Meanwhile while Horace has been doing his thing Agent Smith has been completing his work and will eventually call "wait" to wait for the event of a child process exiting. When he finds that Horace has exited and retrieved the exit status the zombie process is cleared away.

```
▶ puts "Parent pid is #{$$}"
  unless fork
    puts "In child process. Pid is now #{$$}"
    exit 42
  end
  child_pid = Process.wait
  puts "Child (#{child_pid}) terminated with status #
    {${$.exitstatus}"}"
```

Parent PID 14724

```
puts "Parent pid is #{$$}"
▶ unless fork
  puts "In child process. Pid is now #{$$}"
  exit 42
end
child_pid = Process.wait
puts "Child (#{child_pid}) terminated with status #
  {${$.exitstatus}"}"
```

Child PID 14725

```
$ ./fork_if.rb
Parent pid is 14724
In child process. Pid is now 14725
Child (pid 14725) terminated with status 42
```

This is a simulation of running fork_if.rb (code available in https://github.com/gregmalcolm/unix_for_programmers_demo)

SOURCES

<http://mij.oltrelinux.com/devel/unixprg/>

<http://www.faqs.org/docs/artu/>

http://en.wikipedia.org/wiki/Pipeline_%28Unix%29

http://whynotwiki.com/Ruby/_/_Process_management

<http://www.unix.com/unix-dummies-questions-answers/100737-how-do-you-create-zombie-process.html>

http://ruby-doc.org/docs/ProgrammingRuby/html/ref_m_kernel.html#Kernel.fork

<http://cheezburger.com/>

http://en.wikipedia.org/wiki/Unix_philosophy

<http://vimeo.com/11202537>

<http://cheezburger.com/>

https://github.com/gregmalcolm/unix_for_programmers_demo

<git://gist.github.com/1156222.git>

[@gregmalcolm](#)