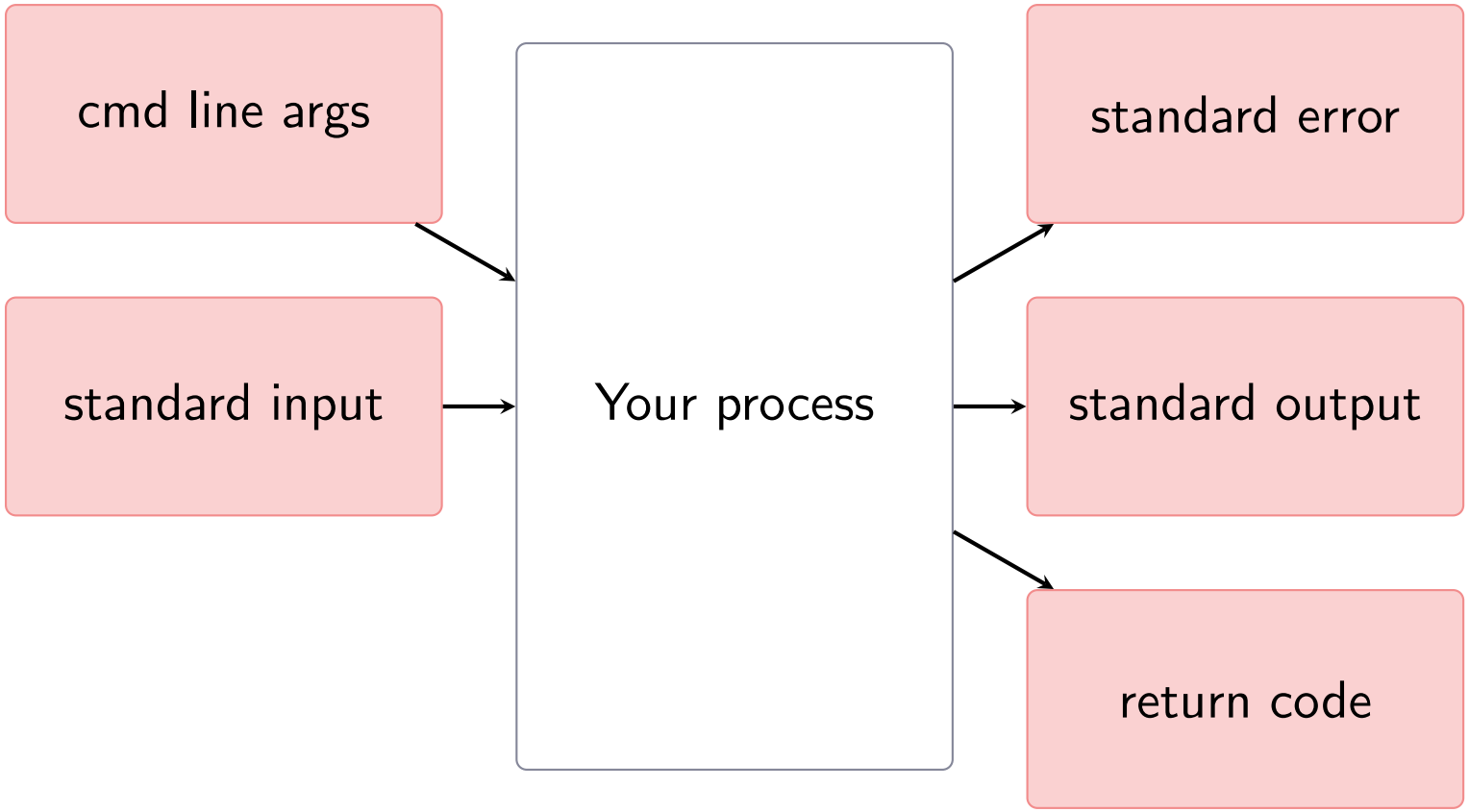
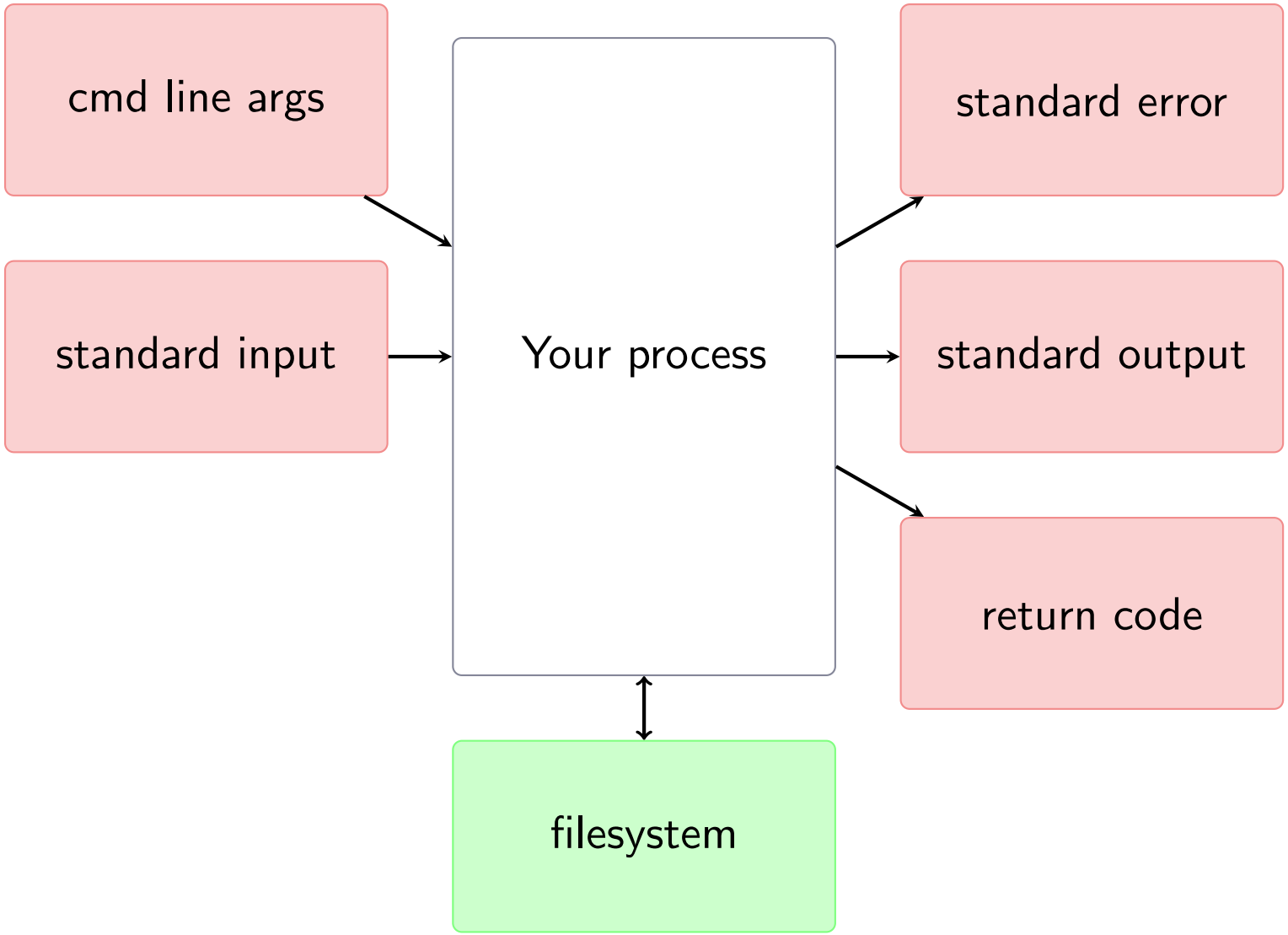
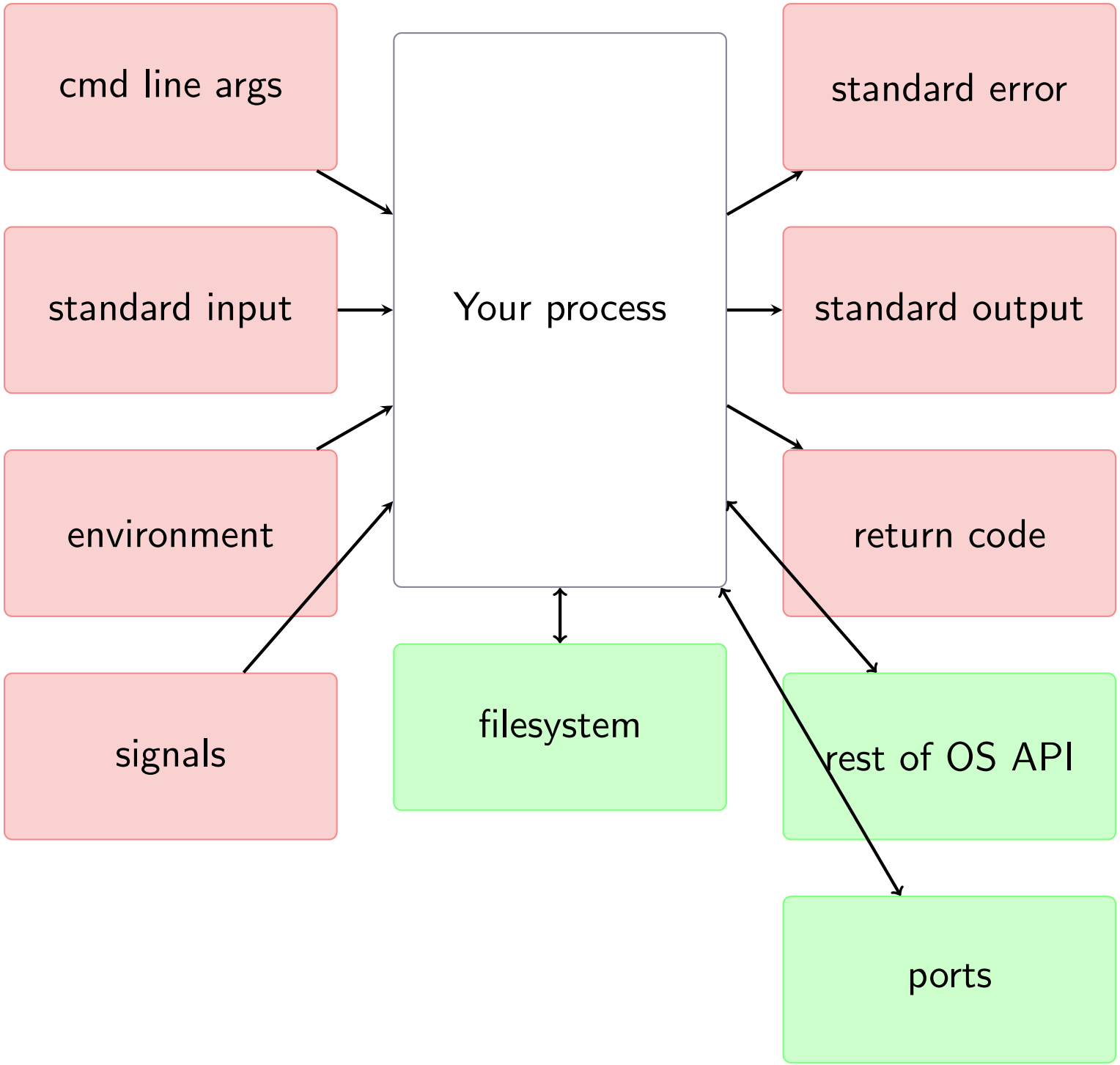
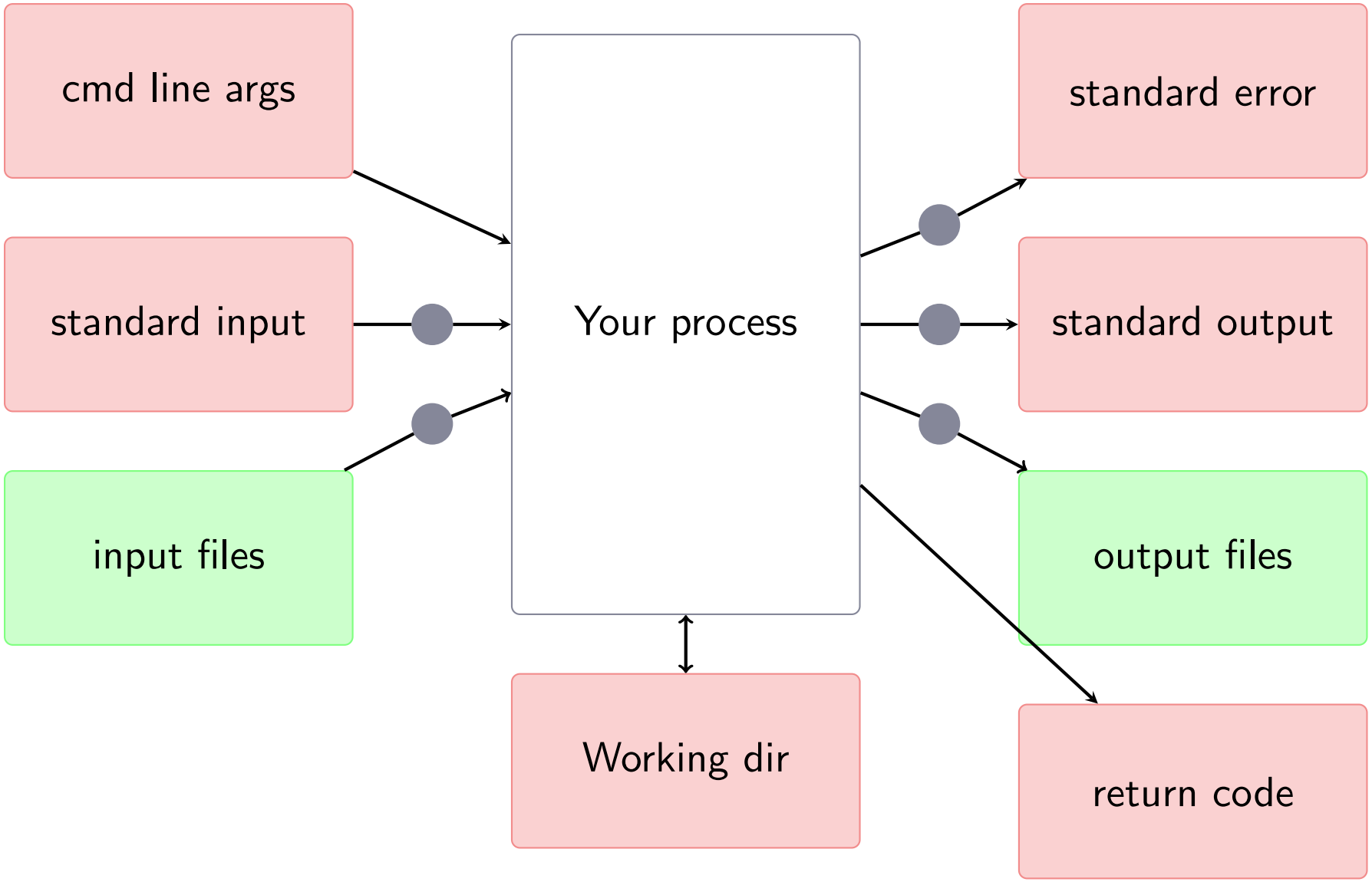

cmd line args
standard input
Your process
standard error
standard output
return code


cmd line args
standard input
Your process
standard error
standard output
return code
filesystem


cmd line args
standard input
environment
signals
Your process
filesystem
standard error
standard output
return code
rest of OS API
ports


cmd line args
standard input
input files
Your process
Working dir
standard error
standard output
output files
return code

What happened? (more or less)

- I clicked on the Terminal.app icon in the Dock
- Some device driver noted that a click had occurred.
- The OS noticed
- The OS checked where the cursor was
- The OS did some calculations and determined that it was in the "Dock"
- The OS told the `Dock` process that a click had occurred.
- By asking the OS and doing calculations, the `Dock` determined that it was the Terminal.app icon that was clicked.
- The `Dock` process told the OS to tell the `Finder` to launch Terminal.app
- `Terminal` was launched and it negotiated with the OS and `WindowServer` to display a window.
- The `Terminal` launched a process called `login`
- `login` read a history file. Based on this information it wrote a message to its standard output stream.
- `Terminal` is reading `login`'s output, and `Terminal` makes the window display the message.
- `login` launched a process called `bash`

- `bash` initialized itself (by reading files such as `/Users/mholder/.bash_profile`)
- `bash` wrote its prompt (the characters '~ 500 $ ') to its standard error stream.
- `Terminal` has wrapped up bash's standard streams (input, output and error stream). When it detects that bash wrote something `Terminal` does some OS calls to draw the characters in the window.
- `bash` told the OS through some function we'll call "readNextLineOfInput" that it wanted to read the next line from standard input. Because there was no input, the execution of the `bash`'s process hangs.
- I typed 'l'
  1. a keyboard device drive noticed the key was hit and told the OS
  2. the OS asked the window manager what application had "focus" – the answer was `Terminal`
  3. The OS told the `Terminal` that there was a key-down event
  4. The `Terminal` (with help of OS) the letter 'l'
  5. The `Terminal` wrote 'l' to a stream that (via OS functions) was connected to `bash` standard input
  6. The `Terminal` displayed the letter 'l' in the Window
  7. Because it was not a carriage return, the 'readNextLineOfInput'

function stored the character, but did not return anything to `bash`. So `bash` still has not heard anything yet.

- I typed 's' – same steps as for when I typed 'l'
- I typed the 'return' key – steps 1-6 occurred as before.
- The 'readNextLineOfInput' function that `bash` called returned the string 'ls<newline>'
- `bash` (through rules we'll talk about later) determined that 'ls' means that I wanted to run the program called `ls` with no command line arguments.
- Via negotiations with the OS, `bash` launched `ls`
- Any standard input of `bash` will now be redirected to `ls`, and the standard output of `ls` will be written to the stream, but it will be redirected to `bash`'s stdandard output.
- `ls` checked to see if it got any command line arguments (it did not).
- `ls` asked the OS what it should use as its working directory (The OS said '/Users/mholder').
- `ls` asked the OS for the contents of the file '/Users/mholder'
- The OS dealt with some filesystem device driver software to get the contents and return the answer.
- `ls` filtered out any entries in that file that started '.'

- `ls` queried the OS to find out how many characters wide its standard output was.
- `ls` formatted the entries such that they fit nicely in the width.
- `ls` wrote the formatted strings to its standard output.
- That standard out from `ls` is passed to `bash`'s standard output
- `Terminal` is reading `bash`'s output, and it makes sure that the output is displayed in the window that we see.
- The `ls` process is exits
- `bash` detects that `ls` exited. It writes it's prompt again.
- `Terminal` displays the latest output from `bash`, which is the new prompt.

`ls` never deals with `bash` (and certainly not with processes further up stream such as `login` or `Terminal`). `ls` just:

- checks its command-line arguments,
- figures out its working directory (asks the the operating system some context information),
- composes an answer to its assigned task,
- writes the answer to its standard output stream,
- exits

`bash` never deals with `login` or `Terminal`. It just:

1. checks its command-line arguments,
2. asks the the operating system some context information,
3. reads a line of input from standard input,
4. takes the action requested,
5. writes "the answer" to through its output and error streams,
6. goes back to step 3

`bash` is a shell that is often used to launch other processes. So the "action" in step 4 is often: "launch a process and redirect your streams to the new process until it exits."

Shells

The interface to the OS and kernel is a huge number of functions – it easy to invoke these functions in the wrong way, and their "raw" response is often cryptic.

Shells:

1. protect the kernel – rather than pounding the kernel with our typos, the shell composes valid requests.
2. make the raw output of system functions human-readable.

The kernel does not understand text like 'ls'

Shells are simple programming language interpreters that take text that humans write, and convert it into instructions for the machine.

1. The shell is a way of creating a process from the executable and controlling the process' initial context:
   (a) Specifies command line argument for the process
   (b) You can control the environment of the process easily through a shell (we won't tweak the environment much in this course)
   (c) The shell's working directory becomes the process's working directory
   (d) The shell controls what happens to the process' standard streams (in, out, and err)
   (e) The shell captures the return code of the executable as the shell variable '?'
2. Failures of commands/executables are denoted by any return code other than 0
3. >
   will redirect the standard output of a process to a file.
4. 2>
   will redirect the standard error stream of a process to a file.
5. <
   can be used to redirect a file as standard input
6. |
   redirects standard output of one process to standard input of another.

`Bash` line processing:

1: Read a line of input (using an OS file-reading function)
2: Tokenize and expand variables
3: The first token is the COMMAND
4: **if** COMMAND is "exit" **then**
5: the `bash` shell exits
6: **else if** COMMAND is a special `bash` keyword **then**
7: take the required action
8: **else**
9: COMMAND should be an executable
10: **if** An executable called COMMAND is found on `bash`'s search path **then**
11: Launch the executable; give it the other tokens as cmd-line args.
12: Store the return code of the executable in the variable '?'
13: **else**
14: Give an error message
15: Store 127 in the variable '?'
16: **end if**
17: **end if**
18: Go back to step 1

`bash` tokenization and variable expansion

1. tokens are usually delimited by whitespace
2. to make whitespace a part of a token you have to use quotes or the backslash (escape) mechanism
3. the $ in a bash command triggers replacement of the following variable name with the variable's value
4. single quotes prevent variable substitution
5. double quotes allow variable substitution

`Bash` COMMAND dispatching (How the shell finds executables):

1: **if** COMMAND is a path with a directory **then**
2: **if** the path COMMAND does not exist **then**
3: print `-bash: COMMAND: No such file or directory` and return 127
4: **else if** COMMAND is not flagged as an executable **then**
5: print `-bash: COMMAND: Permission denied` and return 126
6: **else**
7: run the executable COMMAND and return its return code
8: **end if**
9: **else**
10: **for** each DIRECTORY in `$PATH` **do**
11: **if** the DIRECTORY has an executable called COMMAND **then**
12: run the executable DIRECTORY/COMMAND and return its return code
13: **end if**
14: **end for**
15: print `-bash: COMMAND: command not found` and return 127
16: **end if**

Directories and the filesystem

1. hierarchical (like a phylogenetic classification where the higher level groups correspond to directories)
2. directories are separated by /
3. you can specify paths by:
   (a) their absolute location (always starts with / for the root of the file system), or
   (b) the path relative to the current working directory.
4. The current directory is denoted with a dot `.`
5. The parent directory is denoted with a two dots `..`
6. The grandparent directory is `../..`
7. These rules for finding file paths apply to the OS library calls for opening files as well as `bash` (In python we'll see the same rules).
8. filename that start with a dot are "hidden"

`ls`	list the contents of a directory ls -l .Trash
`echo`	writes the command line arguments (separating them with a space) to standard error echo hi there
`cd`	change the shell's working directory cd Desktop
`mkdir`	create a new directory mkdir tmp
`rm`	Remove a file (or directory if the -r is used). Be Careful!! – their is no UNDO ! rm a.out
`cp`	Copy a file (or directory if the `-r` is used) to a new location cp src dest

`mv`	Move a file (or directory if the `-r` is used) to a new location mv orig new
`man`	Uses the PAGER interface to look at a help page for a command (it may be easier to use google to find the man page) man rm
`pwd`	writes the path to the current working directory to standard out pwd
`env`	writes all of the shell variables to standard out env
`ssh`	starts a `login` on another machine ssh phylo.bio.ku.edu
`scp`	like `cp` but can copy to a remote machine scp src 129.237.138.127:dest

`cat, tail, head`	Write contents of a file to standard out cat x.txt
`wc`	Counts words, characters, or lines in a file wc -l x.txt
`which`	Writes the full path to an executable to standard out which ls
`ps`	Lists the running processes ps auxww
`chmod`	Change the mode of a file. This is how we grant read/write/executable permissions. chmod +x echo.py would make `echo.py` executable

Efficiency tricks

1. arrow up and arrow down to move through your command history
2. `Control-a` moves the cursor to the front of a line; `Control-e` to the end.
3. `Meta-f` moves the cursor forward one word; `Meta-b` move back one word.
4. history displays your history. The command !34 would repeat the 34th command.
5. Hitting the `Tab`-key when you are typing in `bash` will autocomplete paths
6. cd - takes you back to your previous working directory.
7. use a alias shortname='long command here' in your home directory's `.bash_profile` file to make shortcuts to commonly used, but cumbersome commands (one to open a file in your preferred text editor is nice).
8. On Mac, open filename opens the path 'filename' in whatever the Finder thinks is the appropriate application.

Software – instructions for a computer.

A program – text that conforms to the definition of a programming language. The program is not written in the binary operations that the hardware can operate on. It has to be translated into the language of the machine.

Compiled languages:

1. Your program → (compiler) → executable

Then you have to launch the executable:

2. executable → (kernel) → running process → Your results

Interpreted languages:

1. interpreter executable → (kernel) → running interpreter process
2. Your script → (interpreter process) → Your results

Python

1. an interpreted procedural language with good support for lots of programming tasks
2. free
3. terse, but clear syntax
4. platform-independent (and your python programs will be if you don't do weird things)
5. execution speed is slow (but that rarely matters)
6. mature and stable (lots of libraries, runs reliably on any modern computer)
7. Comes with standard `sys`, `os`, and `subprocess` libraries for interacting with the Operating System.

Memory

1. internally all memory is binary – bytes on the hard disk
2. a semantic (interpretation) step is needed to express complex data