



Erik Engheim

Follow

Jul 13 · 23 min read · ✨ · 🎧 Listen

Save



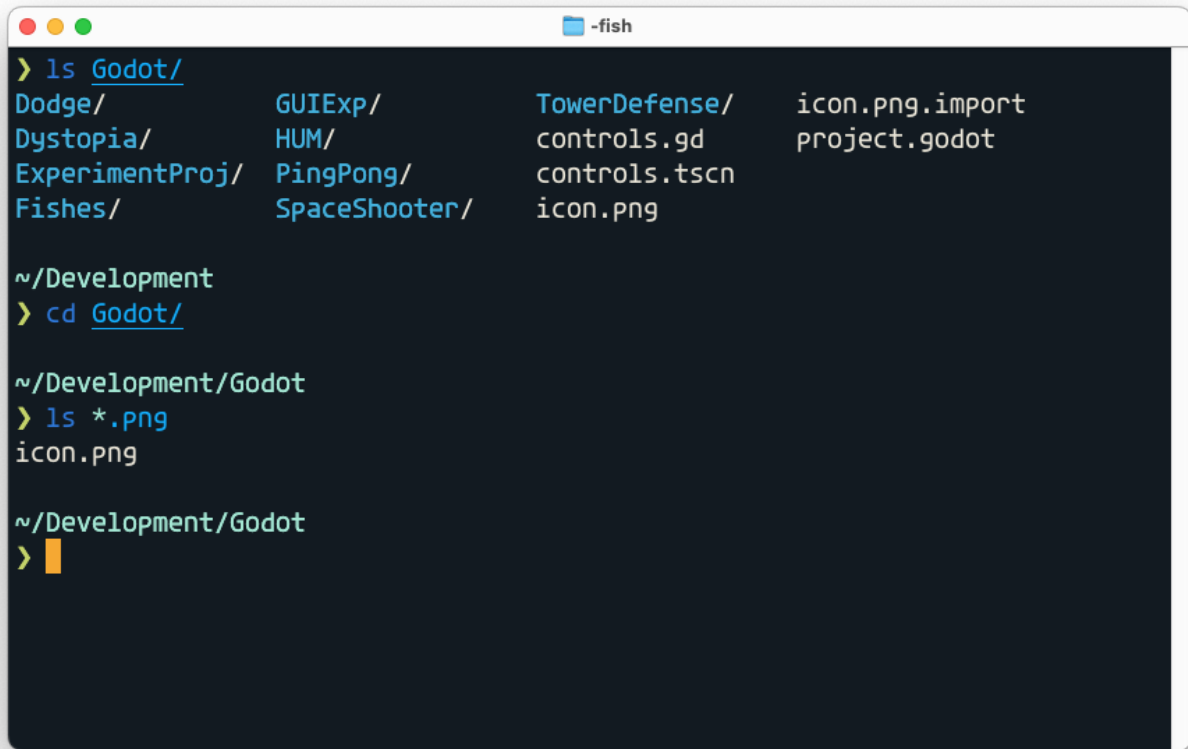
Unix Shells and Terminals

Is a Unix shell and a Unix terminal the same thing?



Unix did not start life on a personal computer. It was programmed through a teletype terminal.

You have probably seen a Unix terminal emulator before. The image below is from the macOS Terminal application. There are many of these. My favorite is [iTerm2](#). Linux's users may use the GNOME Terminal, or the KDE application Konsole.

A macOS Terminal window with a dark background and light blue text. The window title is "-fish". The user has entered the command "ls Godot/" and the output shows a list of directories and files: Dodge/, GUIExp/, TowerDefense/, icon.png.import, Dystopia/, HUM/, controls.gd, project.godot, ExperimentProj/, PingPong/, controls.tscn, and Fishes/, SpaceShooter/, icon.png. The user then enters "cd Godot/" and the prompt changes to "~/Development/Godot". Finally, the user enters "ls *.png" and the output is "icon.png".

```
> ls Godot/
Dodge/          GUIExp/        TowerDefense/   icon.png.import
Dystopia/       HUM/           controls.gd     project.godot
ExperimentProj/ PingPong/       controls.tscn
Fishes/        SpaceShooter/   icon.png

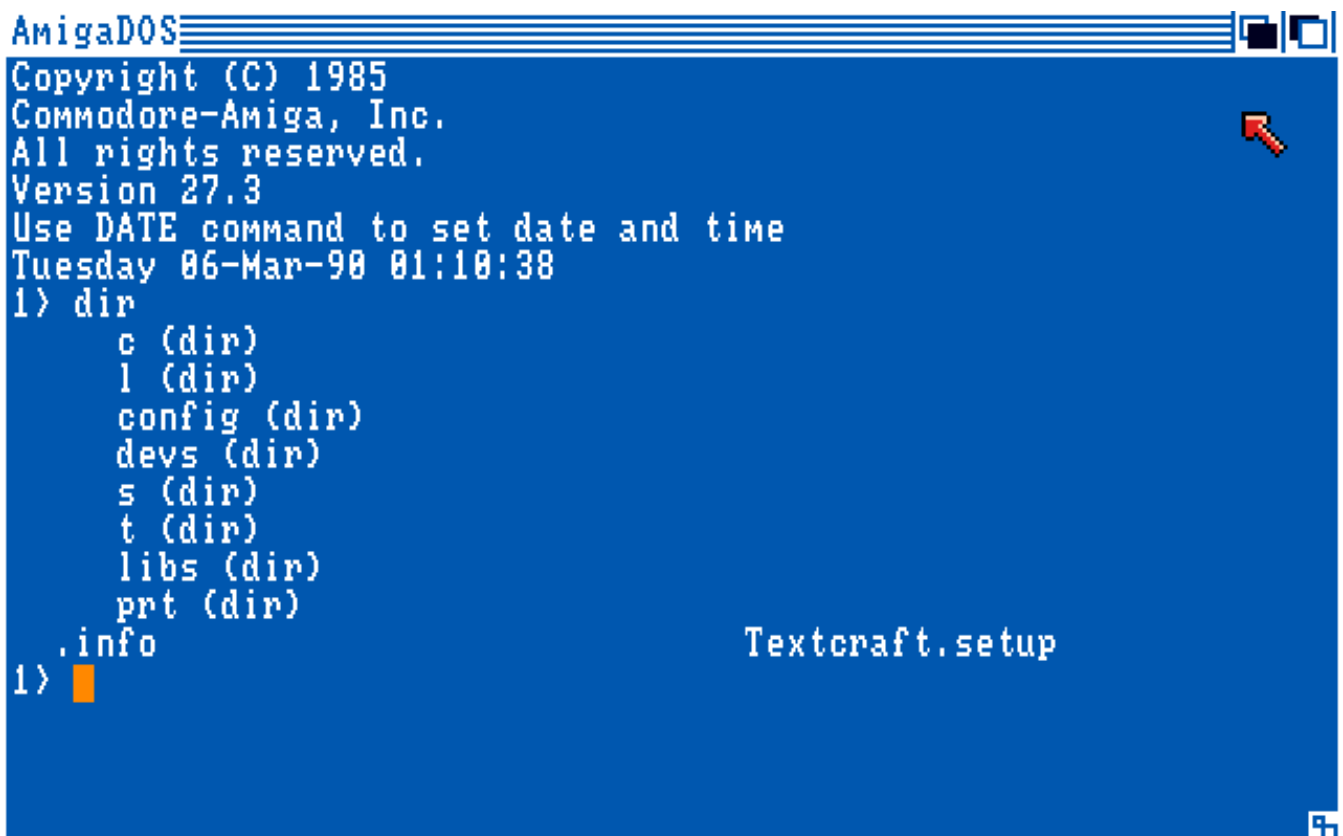
~/Development
> cd Godot/

~/Development/Godot
> ls *.png
icon.png

~/Development/Godot
> 
```

Listing contents of Godot directory using Terminal application

A more generic term for these interfaces is CLI (Command Line Interface). My first exposure to a CLI was not actually a Unix system or Microsoft DOS, but on an Amiga 1000 computer running AmigaDOS in a blue CLI window.

A screenshot of an AmigaDOS Command Line Interface (CLI) window. The window has a blue background and white text. At the top, it says "AmigaDOS" in a stylized font. Below that, it displays copyright information: "Copyright (C) 1985 Commodore-Amiga, Inc. All rights reserved. Version 27.3". It also shows the date and time: "Tuesday 06-Mar-90 01:10:38". The user has entered the command "dir" and the output shows a list of directories: "c (dir)", "l (dir)", "config (dir)", "devs (dir)", "s (dir)", "t (dir)", "libs (dir)", and "prt (dir)". The user has also entered ".info" and the output is "Textcraft.setup". The prompt "1>" is visible at the bottom left.

```
AmigaDOS
Copyright (C) 1985
Commodore-Amiga, Inc.
All rights reserved.
Version 27.3
Use DATE command to set date and time
Tuesday 06-Mar-90 01:10:38
1> dir
  c (dir)
  l (dir)
 config (dir)
  devs (dir)
  s (dir)
  t (dir)
  libs (dir)
  prt (dir)
.info
1> 
```

The Unix command-line can be confusing to get for beginners. People get confused by the difference between a shell, such as the Bourne Again Shell `bash`, a computer terminal, such as VT100, and a terminal emulator such as Konsole or GNOME Terminal. To add more confusion, we can talk about Pseudoterminals.

A couple of questions naturally pop up from these examples:

1. Why on earth does it have to be so complicated to have a command-line-interface?
2. Do I even need to know any of this? Can I just live in ignorant bliss?

For a long time, I did not know the nuts and bolts of the Unix command-line and was still using it without too many problems. Yet, without understanding the underlying technology, you will frequently struggle with understanding important things.

I will take an example from the Git version control system: For a long time, I tried to stick with only knowing the commands to issue. I tried to only relate to Git as a user. As someone who only knew what commands to use. That didn't work. I got extremely frustrated and was ready to give up on Git entirely. A friend of mine implored me to not give up. So, I made a last ditch effort by reading a book on Git under the hood. Suddenly, everything clicked. I actually made a talk based on this experience which helped many people who struggled with Git: [Making Sense of the Git Version Control System](#).

The Unix command-line is much the same. By understanding the underlying concepts, you will be able to do more with the command-line.

Why is the Unix CLI So Complicated?

If you built a command-line interface from scratch today, you could make something much simpler. It would be designed for a mouse, window, and keyboard from the start. Copy-paste would on Windows and Linux work with Ctrl-C and Ctrl-V just like any other application. Hotkeys for jumping one word or line at the time would work like any other text editor. You would be able to place the cursor easily with the mouse between letters. Yet, you normally cannot do any of these things in a Unix terminal, and it is probably not obvious why these seemingly arbitrary restrictions exist.

The problem is that large amounts of Unix tools such as `ls`, `telnet`, `ftp`, `ed`, `awk`, `sed`, `grep`, `tar`, `gzip` and many others got made for computer hardware which no longer exists. People would rather not abandon all the software they had grown accustomed to using. They would rather not rewrite all this software from scratch. Thus, as Unix hardware and software evolved, they created various forms of emulation on top to keep existing tools such as `grep` and `tar` thinking they are still running on an old Unix system.

It is the same reason why we use Intel x86 processors today. The instruction-set architecture is quite outdated. It does not look like what we would have made if we could have designed a microprocessor from scratch. Yet, the design has not stood still. It has evolved and expanded with a lot more features to stay relevant. A modern instruction-set architecture would be closer to Arm used in M1 and M2 Macs today, or the RISC-V instruction-set.

We see the same with programming languages. C++ has turned into a monstrosity. Nobody would design anything like C++ today if given a clean slate. The language gained ground initially because developers could reuse their old C code. Later, it has remained relevant by adding features while keeping old C++ code running.

Thus, understanding any widely used technology today is equally about archaeological digging as it is about understanding engineering and program design. I used to work on a 3D modeling tools written in C++ from 1992. Frequently understanding the architecture and design could not be done by considering engineering trade-offs but by asking the old timers to give you a history lesson about the good-old-days.

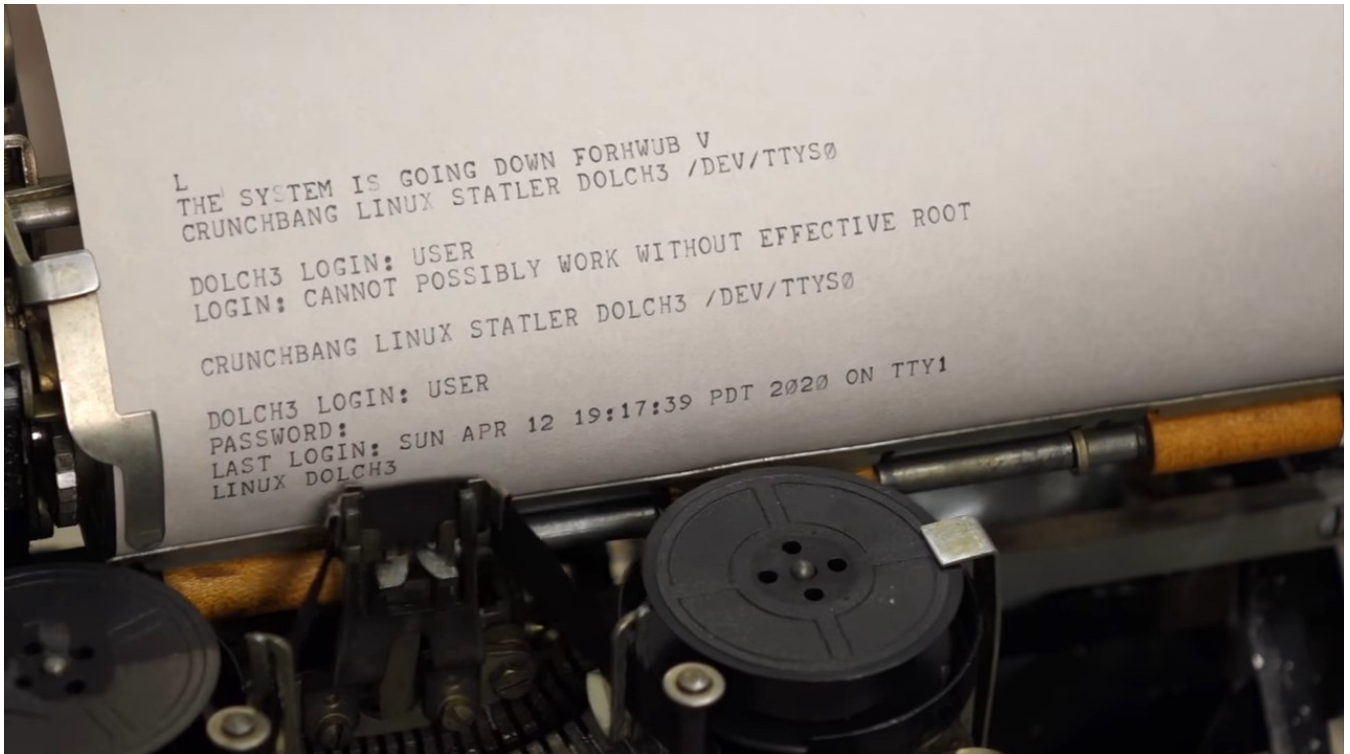
It may seem like a waste of time and energy to learn, but I can promise you that understanding the history of how Unix terminals came to be, will greatly help you understand why the system works the way it does.

History of the Unix Terminal

My aim in this section is to get you to grasp how a terminal, shell and terminal emulator relate to each other.

The original Unix mainframe computers did not have electronic displays like you are used to today. They were not individual personal computers, but huge boxes meant to be used by multiple users. Computer users instead had something called a teletype on their desk. You may also have heard them referred to as a teletypewriter, teleprinter, or

TTY. These machines worked like a glorified typewriter and telegraph all rolled into one. It had no memory, microprocessors, or anything like that. It was primarily an electro-mechanical device. You typed letters on it that showed up on paper like a real typewriter, so you could see what you were typing. The key difference is that the letters you typed got sent over a serial cable (RS-232) to a Unix computer.

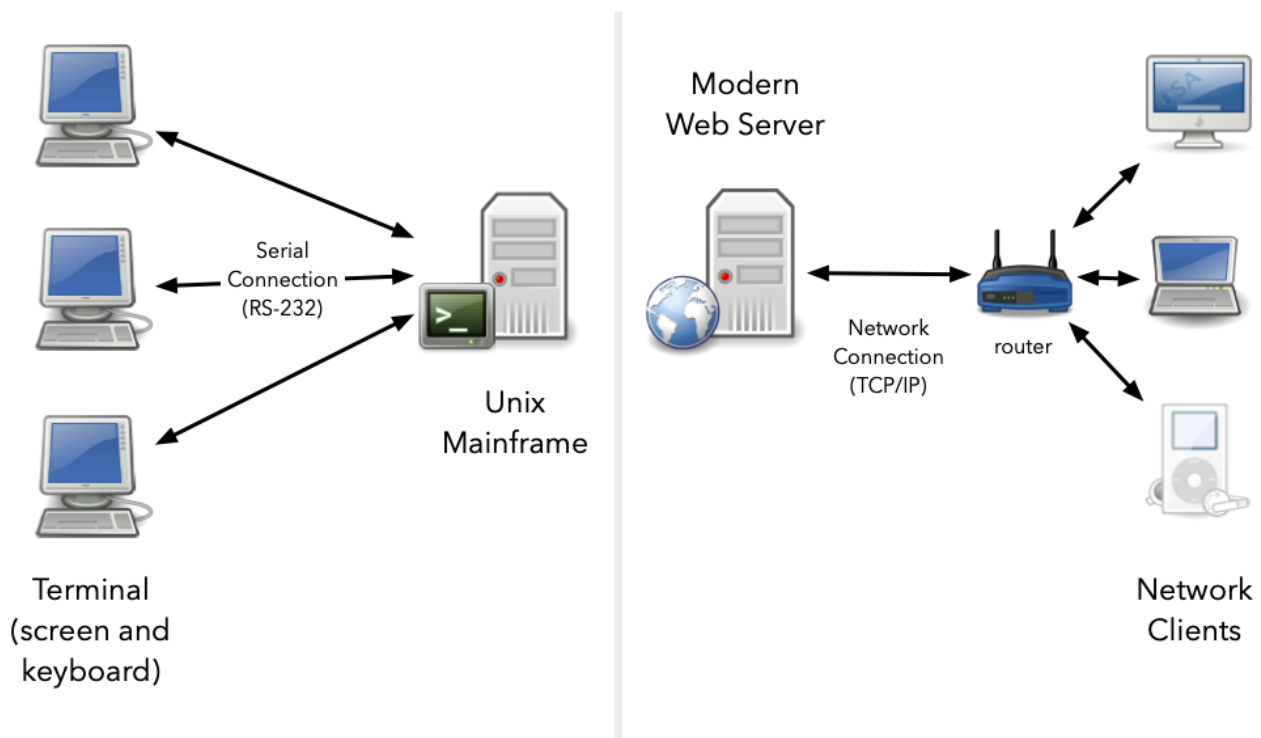


Linux running on 1930s teletype. Results from shell commands gets printed on actual paper.

To understand the relation between Unix computers and teletypes, it helps to make an analogy with our modern web-driven world. Google, Amazon, and others have numerous servers running web server software. Users can connect to these computers through the internet from their smartphones, tablets, laptops, and desktop computers running web browsers such as Firefox, Chrome, or Safari. Ultimately, you have multiple users connecting to the same computer over network cables. The server responds with requested web pages.

In the Unix mainframe setup, everything is more primitive by our modern standards. Instead of a smartphone, you connect with an electro-mechanical device, the teletype, which isn't even a real computer. A teletype cannot perform any calculations or run any software. All it can do is register what letters you press and send them over a serial cable to the Unix mainframe. The Unix mainframe will respond by sending letters back across the serial cable. These letters will get punched out on paper on your type-writer-like machine as they arrive.

There are several notable differences from our modern web and internet world. Networks send data in packets, such as TCP/IP packets. Even a humble USB cable actually sends packets. A packet is a self-contained little chunk of data. You can think of a packet like a letter: It says where it is from, where it is going to and contains some data. That way, data from multiple different clients can move along the same network cable to a server. When received the server can separate out the different packages so that it can handle communication between itself and different smartphones, tables and laptops all with just one physical cable.



Comparing relation between Unix terminals and modern Web clients

A Unix computer did not work like that. Every teletype needs a separate serial cable. It doesn't send packets of data. It just sends binary digits representing the keys you press as soon as you press them. It is a basic system. It was not actually made for computers in the first place. Teletype machines got made as a replacement for the telegraph. Instead of tapping a telegraph and hearing beeps on the other end, you had the luxury of being able to type whole texts. No need to learn morse code.

In the 1970s, teletypes evolved into more modern electronic variants such as the VT100. You can consider it to be a monitor and keyboard joined together as one device. Hence, you can think of old Unix mainframes as a computer which allowed you to plug in multiple screens and keyboards. Each keyboard and screen combo served a different user.



VT100 terminal. Not a PC. Only communicates with computer.

These terminals did not have to be connected to Unix mainframes. They could be connected to many other systems, such as VAX; thus they were quite generic devices. When Unix computers shrunk and became desktop computers for individual users such as a PC, the relationship between terminals and the computer changed. Unix got a windowing system called X Window System, where you run different software. One type of applications you could run were terminal emulators. An early variant was simply called `xterm`.

Terminal emulators pretend to be old physical terminals like the VT100. The underlying Unix system still thinks you are typing in a physical terminal connected by a serial cable. But how exactly do you trick Unix into thinking that the window of a terminal emulator is a “real” terminal? That is the next question we will explore.

Unix Device Files, TTY and PTY

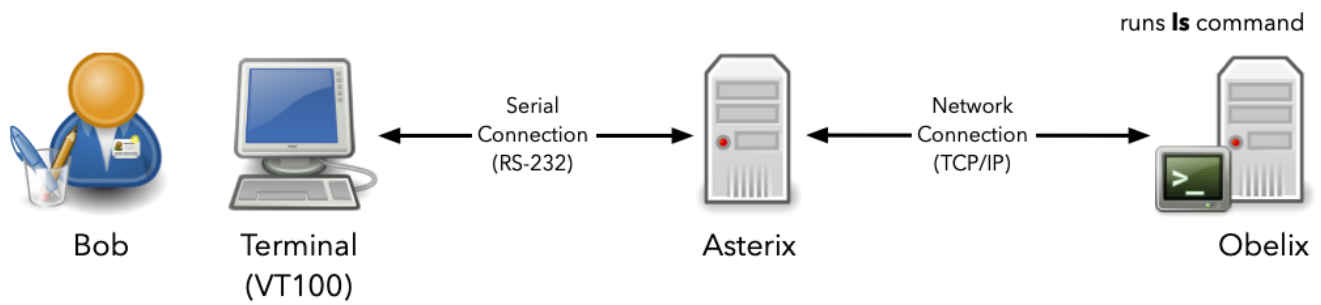
On Unix, the filesystem is more like a namespace for different kinds of operating system resources, rather than simply physical files on disk. Most files map to files on your hard drive. However, on Unix files can also represent physical devices like your keyboard, mouse, audio card, hard drive, floppy disk drive, serial communication to a modem or terminal.

You find files representing these devices in the `/dev` directory. These files provide a way to communicate with the device drivers handling communications with the underlying physical device. The device driver is software which makes the underlying hardware look like a file in the `/dev` directory. In the old days, Unix would communicate with each serial interface through `/dev/ttyS0`, `/dev/ttyS1` and `/dev/ttyS2` files (The `s` stands for serial port).

A program could open one of these files and read from it to get what was being typed on a teletype machine. The programs could write to the file to send letters to the paper on the teletype machine.

Representing this connection to the outside world as just a file proved to be a powerful abstraction. Unix vendors could write new drivers to make communication with a terminal emulator look the same. Unix programs would read and write to the `/dev/tty` files without knowing that they now represented one of many windows in a graphical user interface. Unix has lived on to the modern era in large part because it created powerful abstraction early on.

While Unix users with physical terminals connected through a TTY device, they would often run commands such as `telnet`, `rsh`, `rcp` and `rlogin` (see [r-commands](#)). All these commands imply connecting to another Unix machine. Several Unix machines could be connected through a TCP/IP network. Say user Bob is sitting by an old teletype terminal and connects through a serial port to a Unix mainframe called Asterix. On this computer, he runs the `telnet` program to connect to another Unix computer called Obelix, which is connected to Asterix through a TCP/IP network connection.



Running the `ls` command remotely on another Unix mainframe than the one Bob has connected his terminal to.

While on the Obelix machine remotely, Bob issues commands such as `ls`. How does `ls` know where to send its output? None of the TTY devices would work as they relate to physical ports, but Bob is connected over a TCP/IP connection.

The solution is pseudo-terminals written PTY for short. These are represented as files under `/dev` which get created on the fly by the network applications. In fact, a terminal emulator is handled as a pseudo-terminal since it does not represent an actual physical port and you can make as many terminal windows as you like. On macOS, which I use they have names such as `/dev/ttys000`, `/dev/ttys001` and `ttys002`. They get created in order as I make new terminal windows and tabs.

On macOS, you can check what TTY device your terminal emulator window is communicating with by issuing the `tty` command.

```
> tty
/dev/ttys000
```

You can use the process info command with the `-a` switch to get an overview over all the TTY devices currently used and what programs are being run on them. You can see from this overview that the first thing done after `ttys000` was created was to run the `login -fp erikengheim` command.

```
> ps -a
  PID TTY          TIME CMD
 25797 ttys000    0:00.02 login -fp erikengheim
 25798 ttys000    0:00.10 -fish
 25898 ttys000    0:00.00 ps -a
 25849 ttys001    0:00.02 login -fp erikengheim
```

```
25850 ttys001      0:00.09 -fish
25897 ttys001      0:00.00 nc -l 1234
```

Unix programs when run create what we call *processes*. A process is the representation of a running program in an operating system. A process can create a child process, or *spawn* a child process, as we would normally call it. The `login` process spawns the `fish` shell process, which finally spawns the `ps -a` process, which gives this overview.

Meanwhile, on the second window I opened, `/dev/ttys001` we got a number of other processes spawned. The beginning is the same, but I chose to start the NetCat program to listen for connections on port 1234.

That means I can send message and receive message from NetCat by doing a regular network socket connection to port 1234 on localhost. However, all it will end up doing is forwarding data to the `/dev/ttys001` pseudo-terminal in which NetCat is running and listening on port 1234. We can test all this to see how it works. Create two separate terminal windows:

- Server — Launch `nc -l 1234` in this window.
- Client — Connect using `telnet localhost 1234`.

You can write a message in the client window to see if it pops up in the server window:

```
> telnet localhost 1234
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hello world
```

On the server side, you should receive the following:

```
> nc -l 1234
hello world
```

If you are connecting from another computer you need to connect using `telnet`, but since we are local, we can cheat and write and read directly from the `/dev/ttys001`

pseudo-terminal. To get out of `telnet` press `Ctrl-]`. This gives you the telnet prompt, allowing you to issue different commands. You will just type `quit`:

```
> telnet localhost 1234
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
^]
telnet> quit
Connection closed.

>
```

Let us restart NetCat to do our little cheat. First, we use the `tty` command to make sure we got the correct TTY device. On my macOS I would get `/dev/ttys001`, but if you run on Linux you would get something different.

```
> tty
/dev/ttys001

> nc -l 1234
```

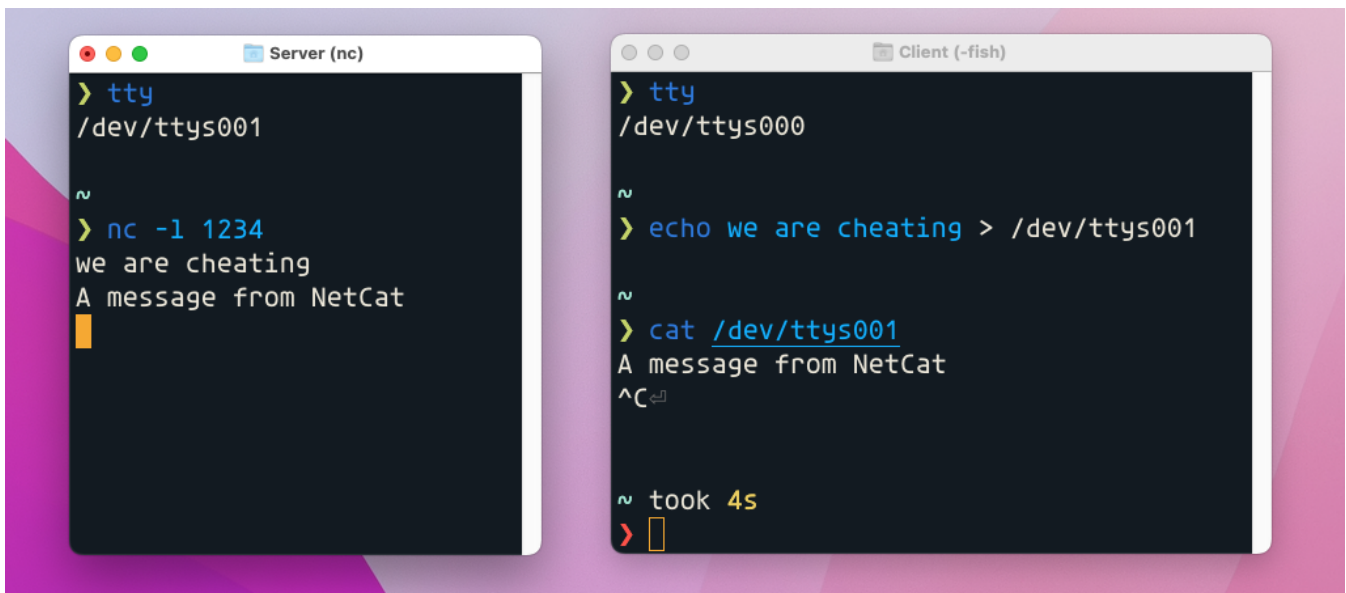
We switch to another terminal window and try sending text to the `/dev/ttys001` device or whatever TTY you use.

```
> echo we are cheating > /dev/ttys001
```

You should see “we are cheating” pop up in the other window.

```
> nc -l 1234
we are cheating
```

You can equally well read from the TTY as if it were a file using `cat`. In this case, `cat` will block until you write a message.



Using NetCat to demonstrate TTY communication

You can signal end of communication by sending a `Ctrl-D` from NetCat. The `cat` command at the other end will interpret that as a hangup and exit. Technically, it maps to an EOF (End-of-File).

```
> nc -l 1234
we are cheating
A message from NetCat
```

At the other end, we will then see:

```
> cat /dev/ttys001
A message from NetCat
```

Any program allowing you to read or write to a file would work. For instance, editors such as `vim`, `kak` and `emacs` would work as well. When you save the file, it will write to the TTY device, and you will see what you edited in your NetCat window.

Serial Communication over USB Cables

Faking a serial communication to retain backwards compatibility can be beneficial in many contexts. We make it look as if a Terminal window is having a serial communication over a RS-232 cable to Unix shell commands using pseudo-terminals. On the Mac, they are represented by `/dev/ttys` files. That is an example of where

everything happens within software. However, we can also use the same trick to make hardware connections which are not serial connections look like one.

Why is that useful? Very few if any computers today come equipped with RS-232 serial ports, instead they come with USB ports. USB is a far more complex connector, working more like a network connection sending data packages rather than individual characters like a good old Unix TTY connection.



RS-232 and USB connector compared

By emulating the software interface of a serial connection, we can write code to communicate with a microcontroller such as an Arduino, as if we connected to it through an old fashion RS-232 port. On an Arduino board there is a USB connector which hardware that translates to a serial connection. Thus the AVR microcontroller on the Arduino board thinks it is communicating across a serial cable. It allows us to write code for serial communication on both sides of the communication link.



Arduino UNO board as a USB-B port. The board uses a USB-to-serial hardware translator.

If you plug an Arduino into your Mac with a USB cable, then a device file such as `/dev/cu.usbserial-10` will pop into existence. The exact name will vary. You can find them by just writing `ls /dev/cu.*`. Why the `/dev/cu*` name instead of `/dev/tty*`? The former was used for model dialup. You could sit at home and dialup to a Unix mainframe through a modem with your terminal. The `/dev/tty*` devices were for people in the building of the Unix mainframe connected directly.

The key difference is the use of the DTR (Data Terminal Ready) control signal on the RS-232 port. A program talking to DTR would block until DTR signals. A teletyper would send a signal to DTR to indicate it is ready to communicate.

For modems it is a bit different because initially you are giving commands to the modem about whom to connect to and how. So in this case DTR is used to signal that a connection has been made. So modems would use the CU device.

I hope this was not too big of a detour. I elaborated on this to hammer home the whole idea of the many ways you can present a different facade to what is happening at the

hardware level and why that is often useful.

Despite all we have gone through I have still not explained anything that helps understand why Terminal emulators are quite complex and can be configured to emulate numerous terminals such as VT100, xterm, ANSI, rxvt and many others.

Control Codes and Terminal Emulators

If all a terminal had to send and receive was visible characters, then everything would be very simple. The problem is that we need a lot more than visible characters. When sending text to a terminal, you need a way to tell it to begin a new line or start a new line. If it has different color cartridges, you need a way to tell it to change what color it is printing on paper.

The user may want to edit or modify text. Thus, we need to be able to move the cursor forward, backwards, up and down, next page, erase a line and so on. To represent all these “invisible” characters, we developed control codes. Different terminals would use different control codes, which is why they would not be all compatible with each other.

Fortunately, an ANSI standard emerged, which is what most people working with terminals will use today. Many of these may be known to you if you have programmed in C or a language inspired by C:

- **Line feed** `0x0A` `\n` - Move to next line. Written as `\n` in C code. On a normal Unix terminal, you send it to your TTY by pressing Ctrl-J.
- **Carriage Return** `0x0D` `\r` - Move to beginning of line. Send to TTY with Ctrl-M.
- **Backspace** `0x08` `\b` - Move one character back (left), typically deleting a letter that was there. Send to TTY with Ctrl-H.
- **Tab** `0x09` `\t` - Move right 8 spaces. Sent with Ctrl-I.
- **Vertical Tab** `0x0B` `\v` - Vertical tab. Move down.
- **Escape** `0x1B` `\e` or `\x` - Start an escape sequence or exit some current mode. Sent with `Ctrl-[`
- **EOT** — `0x04` - End of transmission. When you press `Ctrl-D` you tell the terminal that you are done typing letters. You will often see this referred to as `EOF` because

it can be used to cause an End-of-File condition at the other end. But `EOF` is more of a condition. The end of files don't store character value `0x04`, they just end.

You know that you are not hitting Ctrl-J to type commands in the terminal. Instead, you are hitting the actual Enter key. What is going on here? That is part of the complexity of terminal emulators. Your terminal emulator translates you hitting the return key, tab, and escape to key combinations such as Ctrl-J and Ctrl-I. In some cases, you may want to change this mapping even. You can actually query your TTY to figure out how various control characters (cchars) are configured:

```
> stty -a
speed 38400 baud; 15 rows; 69 columns;
lflags: icanon isig iexten echo echoe echok echoke -echonl echoctl
        -echoprt -altwerase -noflsh -tostop -flusho pendin -
nokerninfo
        -extproc
iflags: -istrip icrnl -inlcr -igncr -ixon -ixoff ixany imaxbel iutf8
        -ignbrk brkint -inpck -ignpar -parmrk
oflags: opost onlcr -oxtabs -onocr -onlret
cflags: cread cs8 -parenb -parodd hupcl -clocal -cstopb -crttscts -
dsrflow
        -dtrflow -mdmbuf
cchars: discard = ^O; dsusp = ^Y; eof = ^D; eol = <undef>;
        eol2 = <undef>; erase = ^?; intr = ^C; kill = ^U; lnext =
^V;
        min = 1; quit = ^\; reprint = ^R; start = ^Q; status = ^T;
        stop = ^S; susp = ^Z; time = 0; werase = ^W;
```

There are so many escape codes to do different things that we would quickly run out of characters when we only have 7 or 8 bits at our disposal. For this reason, one has made *escape codes*. They start with the escape key (ASCII code `0x1B`) and allow us to use multiple characters to specify the operation we want. The common standard being used today is called the ANSI escape codes. Here is a useful [cheatsheet](#).

Escape codes allow us to do lots of things such as moving the cursor to a specific position, changing color of characters, clear the screen or even toggle on and off line wrapping. Text-mode editors such as Emacs, Vim, Pico and [Kakoune](#) (user friendly Vim) use these escape codes extensively to create their text-based interfaces. Some examples:

- `ESC[H` - Move to home position (0, 0).

- `ESC[nA` - Move cursor up `n` lines.
- `ESC[n;mH` - Move to position (n, m) on screen.
- `ESC[0m` - Reset all modes and colors.
- `ESC[31m` - Make text red.

You can fire up a programming language such as Julia or Python and try out these escape codes. Here is a some Julia code which changes to red text, writes some letters and then does a reset.

```
> julia

julia> println("\e[31m Hello world \e[0m")
Hello world
```

You can do it in Python as well, it is just a bit more clunky as you cannot use the `\e` escape code:

```
> python

>>> print("\x1b[31m Hello world \x1b[0m")
Hello world
```

We can also achieve this at the Unix terminal itself. It varies which shell support it. I could not get it working on Z Shell `zsh` and Bourne Again Shell `bash`. However, I get it to work in Fish shell `fish`. One just has to remember to escape the square bracket.

```
> echo \e\[31mhello\e\[0m world
hello world
```

Terminals, File Descriptors and Pipes

We are getting closer to getting a more complete picture of how Terminals work. There is a crucial missing point: How does Unix commands and programs direct their output

to the correct terminal? Occasionally, a program runs remotely and needs to send their output across a pseudo-terminal over the network.

In fact, Unix programs are far more flexible than that. Not only can send output to different terminals, but it can also be sent to entirely different files. For instance, if I type `ls` I get a directory listing sent to my default TTY terminal. However, if write `ls > foo.txt` the listing is sent to a file named `foo.txt` instead. The `rev` command will normally reverse everything I write in (exit with Ctrl-D)

```
> rev
hello
olleh
world
dlrow
```

But it is perfectly possible to get the input to `rev` from a file instead by writing `rev < foo.txt`. Here is an example where we create a file which we later feed the contents of to `rev` using the redirect symbol `<`.

```
> cat > hello.txt
hello
world # Press Ctrl-D

> cat hello.txt
hello
world

> rev < hello.txt
olleh
dlrow
```

When you run a program such as `rev` you create a *process* (representation of a running program in memory). A Unix process use numbered *file descriptors* to communicate with files. Under the hood, when you open a file, a file description is created to reference that file.

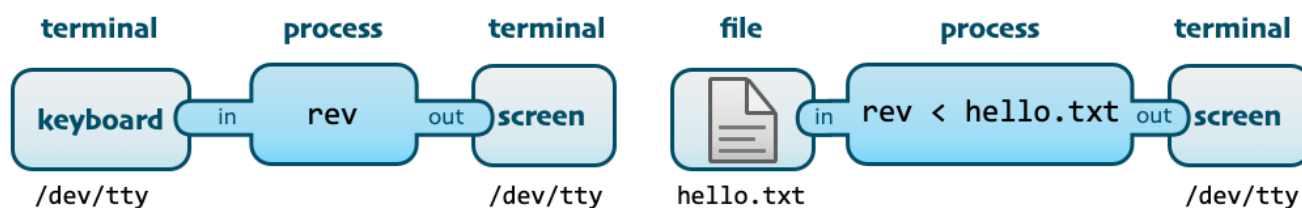
A Unix process has 3 file descriptors numbered 0, 1 and 2 which are always open. These represent:

- standard input — stdin
- standard output — stdout
- standard error — stderr

In fact, you can see them as files under `/dev`. Here you will likely find differences between different Unix operating systems. I am describing how this works on macOS. Each of these files are links to the numbered file descriptors (fd).

- `/dev/stdin` - links to `/dev/fd/0`
- `/dev/stdout` - links to `/dev/fd/1`
- `/dev/stderr` - links to `/dev/fd/2`

In these examples, we will mostly ignore `stderr` and focus on `stdin` and `stdout`. You can think of a process as having 3 sockets labeled `stdin`, `stdout` and `stderr` where we can plug in different files or file-like objects. In the illustration below I am just showing the connectors and I have shortened them to `in` and `out`.



Redirecting input to come from a file

By default, `/dev/stdin` and `/dev/stdout` points to our default terminal `/dev/tty` which again points to a specific pseudo terminal such as `/dev/ttys001`. With the `<` and `>` redirect symbols, we can change what file descriptor our process is actually plugged into for its inputs and outputs.

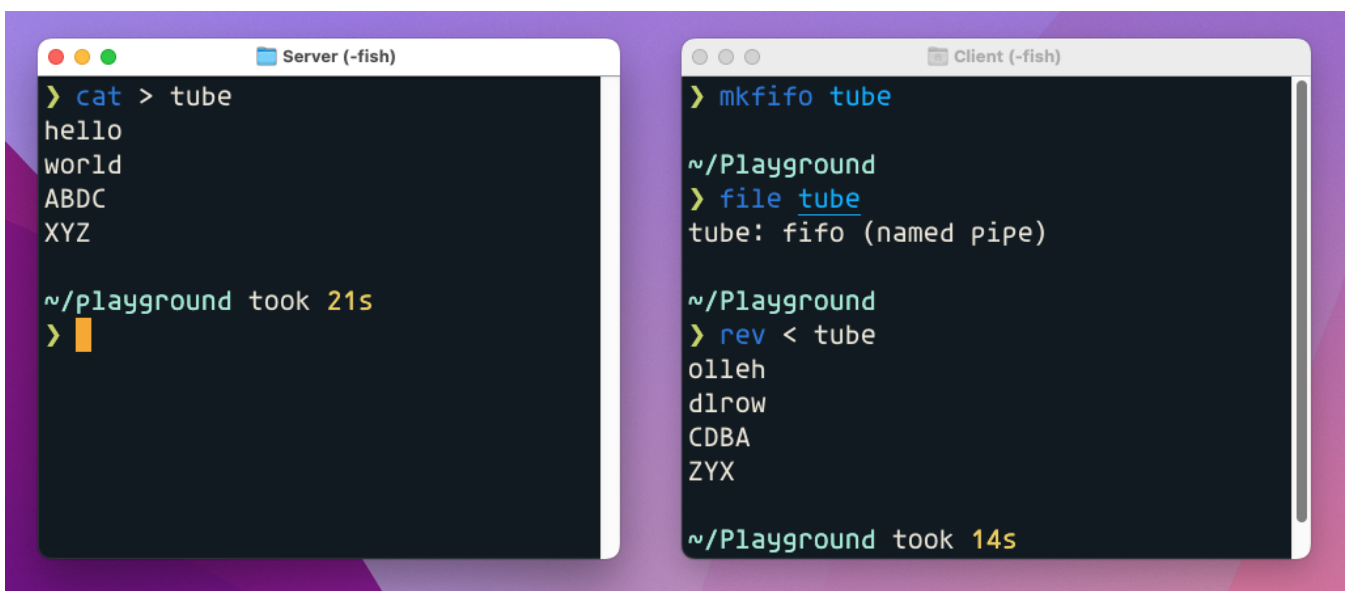
One clever innovation on this abstraction is the concept of a pipe. A pipe is a bit like a file, except two processes can keep it open at the same time. One process writes to it and another reads to it.

We can make such a pipe with the `mkfifo` command. We are making a pipe named `tube`.

```
> mkfifo tube

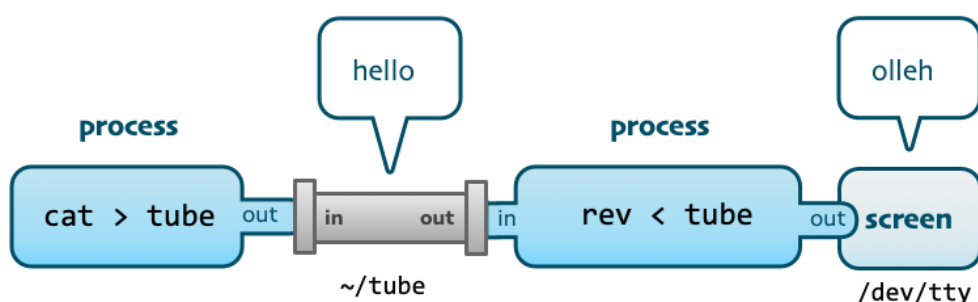
> file tube
tube: fifo (named pipe)
```

This creates an entry in the filesystem called `tube` which will remain there even after you reboot. However, data written and read from a named pipe is never stored on disk. It is entirely handled in memory. Let us look at how this is achieved in practice.



Setting up a named pipe communication

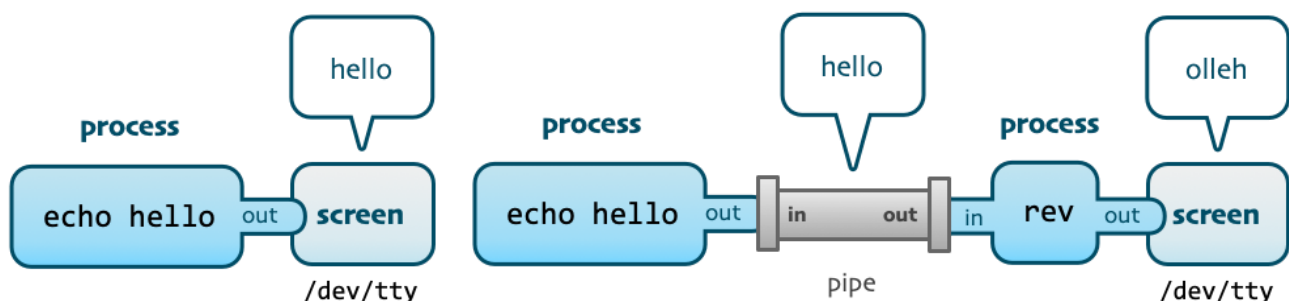
You need two Terminal windows to make this work. In one window you write `rev < tube` which means you try to read from the named pipe. In the other window you write `cat > tube` which means anything you write on your keyboard will be sent into `tube` and the `rev` process will realize it can read from the tube.



Most of the time we don't need named pipes. We are not interested in the name of the pipe or keeping it around permanently. We are quite happy to create a pipe temporarily so we can easily funnel data from one command to another. We can use the pipe `|` character to create such pipes.

```
> echo hello | rev
olleh
```

Conceptually, we have still created a pipe with two ends: one for writing and one for reading, but we haven't given it a name.



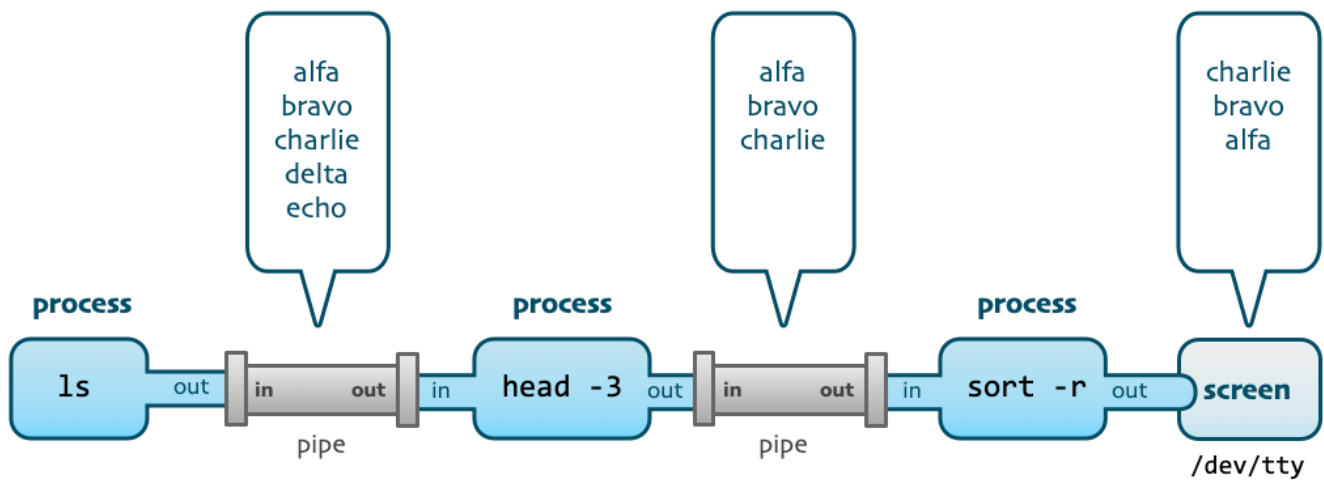
Pipes can be combined in complex manners, allowing us to chain together the functionality provided by many different commands. Let us look at an example of chaining together `ls`, `head` and `sort`:

```
> touch hotel golf foxtrot echo

> ls | head -3
echo
foxtrot
golf

> ls | head -3 | sort -r
golf
foxtrot
echo
```

The `head -3` command picks the first three lines, while `sort -r` will sort inputs in reverse. Conceptually, the stdout and stdin of the processes get connected through pipes as shown below:



Example of chaining together three commands using pipes.

These pipe examples cement the power of Unix to redirect input and output to a variety of places while the programs involved has not idea what is going on. The `ls` command does not need to know whether the output it is sending is going to a pipe, another file, or perhaps a pseudo terminal created by an telnet client. It could even be sending data across a USB cable pretending to be a serial connection.

Thus, when you create a simple program with the code `print("hello world")` a lot will happen behind the scenes:

- The text is written to the file `/dev/stdout` but this is just a link to `/dev/fd/1`
- The `/dev/fd/1` will again point to `/dev/tty` which is your current TTY device.
- `/dev/tty` will point to an actual pseudo terminal such as `/dev/ttys001` which refers to an actual terminal emulator window.

This is how “hello world” get sent to the correct window in your user interface. Of course `/dev/stdout` could be redirected, in which case the text goes somewhere else.

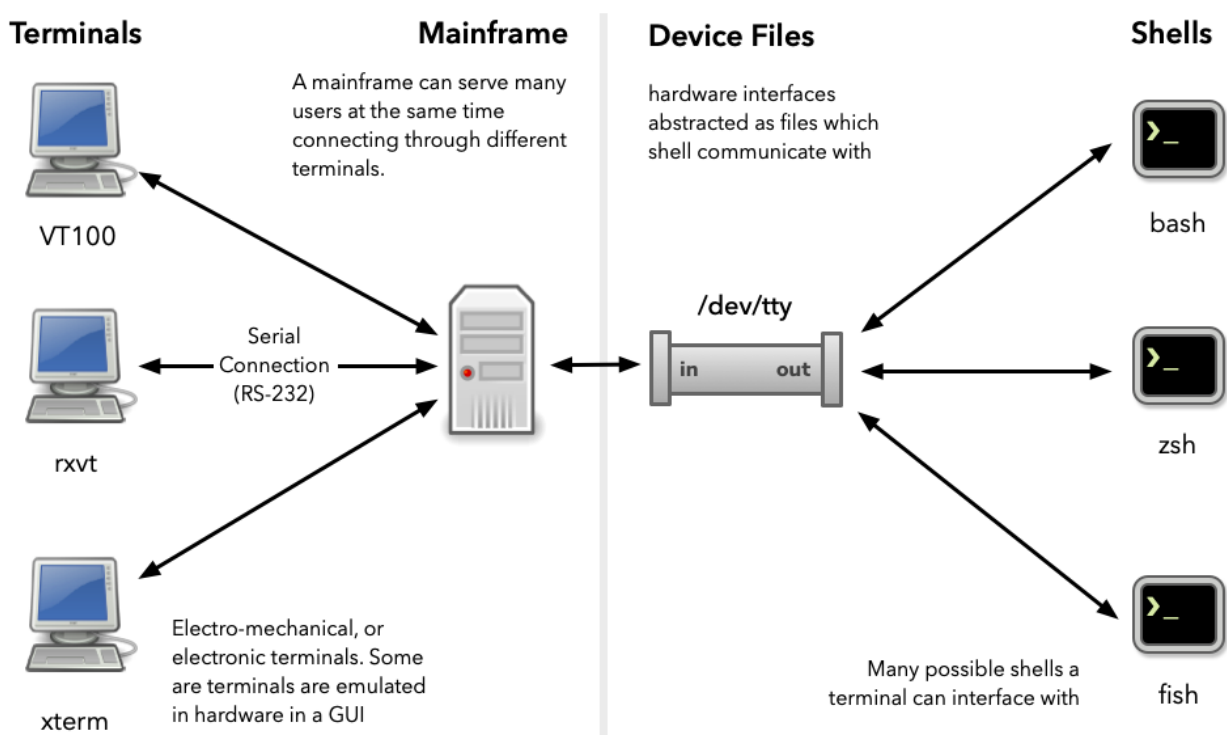
Terminal vs. Shell

A Unix Terminal is really just a way to pump characters into a `/dev/tty` device and read characters and control characters from the same device. A terminal by itself does not offer much functionality.

To be able to parse user input, interpret that input as commands to run and display results, we need a Unix Shell. The Shell manages things like prompt, where you write your commands.

The shell does jobs management, such as letting you switch between different running processes in case one process takes long time to finish. Not every shell will have the same syntax. In many ways, you can view programming languages such as Python, Ruby and Julia as shells onto themselves. However, these programming environments are not well suited as Unix shells since files, processes, programs, directories, and pipes are not first-class objects.

Unix shell like `bash` are not good as generic programming languages but they are good at the things like file-management and jobs control. The diagram below is to summarize the relations between terminals and shells.



In the diagram above I have drawn the terminals as physical devices. In the modern world that is of course rare. They will usually exist in the form of terminal emulators living inside some windowing system.

When you open a terminal emulator window, it will run the `login` program which will spawn your shell. It will start one of the shells listed in the `/etc/shells` file if you are on macOS:

```
> cat /etc/shells
/bin/bash
/bin/csh
```



```
/bin/ksh
/bin/sh
/bin/tcsh
/bin/zsh
/usr/local/bin/fish
```

You can see there are a lot of shells which have been made over the years. On modern macOS the Z Shell `zsh` has become the default although both Linux and macOS long used Bourn Again Shell `bash` as the default.

In the old days the `login` program looked at the `/etc/passwd` file to login a user. There the system could check if the given user was present and the correct password was given. Afterwards `login` could launch the configured shell. Each line in the `/etc/passwd` file look something like this:

```
tommy:x:1000:1000:./home/tommy:/bin/bash
```

Each line gives the name of the user, home directory and what shell to run. For security reasons this file has lost much of its importance. You cannot directly access a password file anymore. It was too easy for hackers to use it to perform dictionary attacks. Instead we use commands to modify user data, such as changing what shell to use. To change default shell we use the `chsh` (CHange SHell) command. This is how I change to use the `fish` shell:

```
> chsh -s /usr/local/bin/fish
```

There is no way to cover everything about Unix terminals and shells in a single article, so I will be adding related stories as they get done in links.

Related Stories

Stores related to Unix, shells and terminals.

- [Unix command line crash course](#) — Focused on very hands on how to use Unix commands. Intended for absolute beginners. Avoid getting into the technical details.