Accepting Less(1):
What does it really mean to be UNIX?

A dialogue of computing philosophy as composed by HackerSmacker

This paper is intended to be, as are most papers I compose, orated aloud to an audience.

I often ask myself “why does UNIX exist?” Better yet, I ask myself “why did UNIX need to exist?” These are both questions that come from a common goal I have of understanding the operating system that has totally shaped computing, but I must also wonder if its influence was purely a net positive. I would like to cover the so-called “UNIX philosophy” alongside its pitfalls, as well as compare and contrast modern UNIX systems with what was “UNIX proper” in the elder days of computing -- by understanding the UNIX philosophy, I hope the audience is able to then answer the two questions I have posed.

The “UNIX philosophy” is actually not one singular philosophy, but a collection of various ideals that people lump into a common label; as such, it is necessary that we break it down into its *many* subcomponents. Often times, when we seek to define this so-called philosophy, we consult a work known as *The UNIX Programming Environment*. Keep the title in mind, as UNIX (from its inception) was described as an environment for *programmers*, not users (this will become important later). Rob Pike and Brian Kernighan write that “...although [the UNIX] philosophy can't be written down in a single sentence, at its heart is the idea that the power of a system comes more from the relationships among programs than from the programs themselves.” Despite being a direct contradiction, where the authors said it could not be described in a single sentence then proceed to describe it in a single sentence, this statement will be the root of our analysis. Essentially, the authors are describing a type of emergent property. What is an emergent property? Consider human consciousness: it cannot exist on its own, it must emerge on top of an already-functioning and properly-connected set of neurons — this, in turn, depends on a working body, a livable planet, a functioning solar system, and the universe. In other words, the authors described the UNIX philosophy as the idea that UNIX is an emergent system, comprised of many parts that each serve their individual purpose and only that purpose. This would be expanded on a few months later (in 1984) in a paper called “Program Design in the UNIX Environment,” where it was stated that these emergent-system building blocks should be simple to use on their own, but very easy to combine

with other programs. This too is where a pivotal feature of UNIX, perhaps its most important feature, comes to rise: shell pipelines. In order to form a spell checker, one could write a dedicated spell-checking program, or adapt a series of smaller programs to achieve the same outcome but in a much faster fashion. This is similar to the “tool store analogy” — say I wish to build cabinets. I could build a dedicated cabinet-building machine that will take 2 months to build and construct 1000 cabinets a day, or I can use hand tools that I buy at a tool store to make 10 cabinets a day (but I can start work on those cabinets as soon as I get home). This is the core of the UNIX philosophy aspect as we just discussed: use the tools that are already there, because they were perfected. The screwdriver, the saw, the hammer, the chisel — all perfected tools you can go buy, but your cabinet maker machine will require much effort to become perfected.

UNIX, throughout its history, has always contrasted itself with other systems. The most notorious example is the UNIX `cat` command, which that same paper contrasts with the so-called “file system commands” found on other operating systems of that time. The `cat` command combines one or more input files into a single output file; if the output is not redirected, the output goes straight to your terminal screen. Other operating systems used different commands to combine (concatenate) and display (type) files, but the authors are specifically referring to the PIP command found on RSX-11 (an operating system found contemporaneously on the same hardware, the PDP-11). They proceed to object to their own statement, by stating that neither approach is better or worse, but that one of them (that is, PIP) goes directly against the UNIX philosophy. Doug McIlroy famously stated that text streams are the universal interface, but even this is not entirely true; so begins our criticism of the UNIX program design model. Say, for instance, that I wish to provide an application with network facilities. One such way to do this is to use a program known as `netcat`. Like `cat`, `netcat` provides a simple stream interface in and out of, rather than file, a network socket. Data can be entered into the program by another program to be transmitted to the remote host via the network connection, and can be received from the remote host and deposited back into another program. This can make for a very useful tool, especially when you wish to test network applications or write a very simple shell-scripted interface to a network service, but there are plenty of downsides.

Let’s say that you were writing a C program on a UNIX system, and you wished to give it TCP/IP capability. You have two options: you can use a set of system calls and C library functions, or you can be a UNIX philosopher and open a bidirectional pipe for the `netcat` program. There is nothing stopping you from

achieving network capability purely through `netcat` at all, but you must mind both the ergonomics and performance factors at play here. The read and write system calls will be just as easy to use with a socket file descriptor as they would be with a pipe file descriptor (as they are the exact same thing), but catching any errors from `netcat` will be quite cumbersome compared to checking the `errno` value that is set after a failed attempt to connect to a remote host. Yet and still, you are attempting to write your own network socket facility, something that `netcat` (a "perfected program" as I often call it) does with more features than you could ever possibly need; your actions, seemingly, go directly against the UNIX philosophy. Oh, you seek to use the wonderful EMACS editor solution? Fear not, but do fear — that too is a monolith that exhibits similar UNIX-style design goals.

Essentially, what I am discussing here is a major component of UNIX, the network stack, being totally independent and exempt from the UNIX philosophy. On all major UNIX systems that have TCP/IP support, the network stack is a monolith that is built into the kernel. On UNIX, as I am sure you are already aware, nearly every device (from the system main memory, to every terminal, to every disk, to nearly everything else) can be accessed through special files in the `dev` directory. The glaring exclusion to this "everything is a file" rule is, of course, network adapters. Despite looking, effectively no UNIX system has a `/dev/net0` device node! Why is this? What would its interface look like? Would I ever need this? To answer those in reverse order, yes, you would ever need this — particularly if you are writing a VPN program (for example). Nearly all modern UNIX systems (Linux, FreeBSD, OS X, and Solaris among them) have what's known as "the tun/tap device" that permits the creation of fake Ethernet cards and loopback devices. This exposes a device you may (on some systems, mind you) be able to open and read/write packets to and from, but those packets are merely sent to the host to be processed. If you wish to use a tap device, for example, to receive arbitrary Ethernet datagrams sent on a LAN, you must use a bridge device to connect the fake Ethernet card to a real Ethernet card (just as you could use a bridge device to connect two real Ethernet cards together, cross-connecting two LANs into one). The network devices that are listed using `ifconfig` are not devices in the `/dev` directory at all, and you cannot UNIX-philosophize your way to a working network stack. For as long as UNIX has existed, save for some notable exceptions (NCP Network UNIX among them), the Internet Protocol was always implemented as something entirely in the kernel, accessible only to the user through special system calls.

So, then, if this is such a clear divergence from the standard UNIX design

methodology, why has it become so pervasive amongst nearly every OS ever, not just UNIX? Well, it comes from a historical perspective that was commonly applied in the era in which Berkeley UNIX gained its now famous TCP/IP stack. UNIX was, and still is, the programmer's system — it has an inbuilt C compiler (or, well, at least, did; now only the BSDs contain one in any "common base distribution"), an online debugger, online pages to remind you of the syntax to various C library functions and system calls, and a somewhat useful filesystem conducive to programs: all things pivotally useful to upholding the UNIX philosophy. The C compiler is subdivided into parts (usually), programs can be manipulated using a variety of command-line stream editor utilities, so on and so forth. As we discussed earlier, however, networking is implemented on UNIX systems in a way that is directly antithetical to how the rest of the operating system is architectured (not to mention what the rest of the consistent user experience might command). There exists this monolith of code, this unapproachable mighty box of system functionality, contained in a field exempt from the UNIX philosophy. Why was this done? Well, put simply, it was done for performance reasons. The UNIX philosophy, as many have noted over the decades it has existed, tends to suffer somewhat when high-performance systems are considered. It is the reason that, for many years, serious enterprise computing has been done on monolithic-kernel systems with, usually, a strong service model. Networking stacks are not implemented with pipes because it is simply not performant, and introduces too much overhead and latency -- this is part of the reason that we have seen the UNIX-dephilosophizing of key parts of the Linux environment (a good example would be `pppd`'s adaption from a pipe-based solution with VTYs to having kernel-mode code that process PPP over Ethernet for certain ISP connections).

With that in mind, is the UNIX philosophy even being held up in this day and age? In the general direction of Linux systems, it would not seem that there is much fuel left to burn the UNIX fire. Major Linux components, such as systemd and Wayland, are functional monoliths in their dominant implementations. X11 was thrown away for being "outdated", all whilst being the most modular graphics system created to date. Was any of this warranted? Yes, of course, but, at the same time, not really. The UNIX philosophy meant that you could mix and match your way to victory, build complex emergent behaviors from simple components, and program with ease. Is UNIX easy for end users to use? No, not at all, it is actually extremely user-hostile! Is UNIX easy for programmers to use? Sure, but, compared to some other systems it competed against (primarily TOPS-10 and VMS), it is no better or worse than any other system. Ultimately, UNIX stands on

its own — as another brick in the wall of a diverse computing infrastructure, another knife in the drawer of cutting solutions.