Why Unix?

Steven Zeil

Last modified: May 11, 2017

Contents:

We can actually interpret the title question in a number of different ways. Do we mean, “why does the CS department use Unix?”, or “why is this course about Unix?”, or “why was Unix invented?”, or even “why does Unix behave the way that it does?”

It’s actually the last of these interpretations that I want to address, although understanding the answer to that question will go long ways towards explaining why the CS department uses Unix in most of its courses and, therefore, the very reason for the existence of this course.

I think that, to really understand a number of the fundamental behaviors of Unix, it helps to consider how Unix is different from what is probably a much more familiar operating system to most of you, Microsoft Windows. Furthermore, to understand the differences between these operating systems, you need to look at the history of computer hardware and system software evolution in effect at the time when each of these operating systems was designed. In particular, I want to focus on three ideas: evolution of CPU’s and process support, evolution in display technology, and evolution in network and technology.

Computer historians are fond of pointing out that mainframe computers were huge behemoths, occupying massive rooms, drawing large amounts of electrical power for their operation, and often requiring cooling systems fully as large as the processor itself. For some time, processors continued to be physically large, although the processing power squeezed into that space grew tremendously.

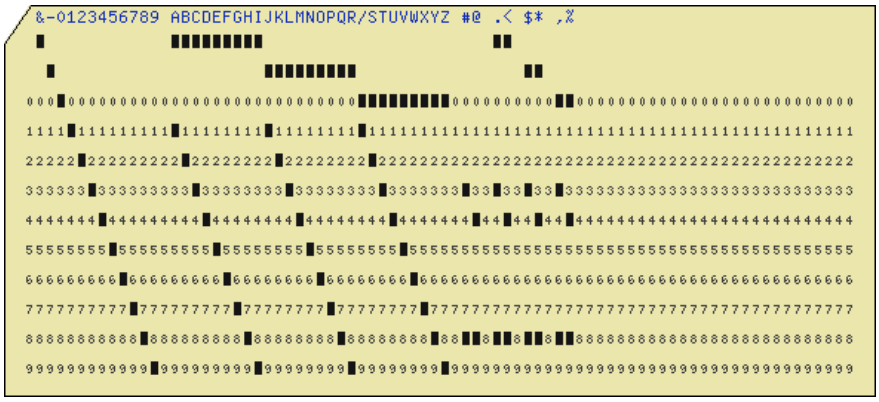
On the early machines, only a single program could be run at any given time. As processors became more powerful, both hardware and system software evolved to permit more than one program to run simultaneously on the same processor. This is called multiprocessing. The initial reason for doing multiprocessing was to allow programs from many different users (programmers) to run at once. This, in turn, is called multiprogramming. At first, it was assumed that a single user had no need for more than one process at a time.

Interactive programs are characterized by long periods of idleness, in which they are awaiting the next input from the user. In an interactive environment, it becomes natural for users to switch attention from one process

that is awaiting input or, in some cases, conducting a lengthy calculation, to another process that has become more interesting. For example, someone using a word processor might want to switch over to their calendar to look up an important date before returning to the word processor and typing that date into their document. Fortunately, once you have support for multiprogramming, you have most of which you need for combined multiprogramming and multiprocessing.

In fact, there is a definite advantage to having started with multiprogramming. In a multiprogramming environment, there is a great danger that one programmer's buggy software could crash and, by rewriting portions of memory or resetting machine parameters, take down not only that programmer's program but other programs that happened to be running on the machine at the same time. Consequently, multiprogramming systems place a heavy emphasis on security, erecting hardware and software barriers between processes that make it very difficult for one process to affect others in any way. Adding that kind of protection on to an operating system that didn't design for it in the first place is much harder.

The trend toward multiprogramming and multiprocessing persisted, not only across families of mainframe computers, but also across the increasing number of desk-size "minicomputers". It is in this context that Unix was developed. From the very beginning, Unix was therefore envisioned as an operating system that would provide support for both multiprocessing and multiprogramming.



During the heyday of the mainframe, most data was entered on punch cards and most output went directly to a printer. Most of these systems had a "console" where commands could be entered directly from a keyboard, and output received directly on an electric typewriter-like printer, but such input was slow and inexact. Prior to multiprocessing, it would have been economic folly to tie up an expensive CPU waiting for someone to type commands and read output at merely human speeds. So the system console generally saw use only for booting up the system, running diagnostics when something was going wrong, or issuing commands to the computer center staff (e.g., "Please mount magnetic tape #4107 on drive 2.").

The advent of multiprocessing and the subsequent rise of interactive computing applications meant that the single system console, hidden away in the computer room where only the computer center staff ever touched it, was replaced with a number of computer terminals accessible to the programmers and data entry staff. An early computer terminal was, basically, a keyboard for input and an electric typewriter for output.

Terminals were not cheap, but their lifetime cost was actually dominated by the amount of paper they consumed. In fairly short order, the typewriter output was replaced by a CRT screen. This opened up new possibilities in output. A CRT screen can be cleared, output to it can be written at different positions on the screen, and portions of the screen can be rewritten without rewriting the entire thing. These things can't be done when you are printing directly onto a roll of paper. Terminal manufacturers began to add control sequences, combinations of character codes that, instead of printing directly, would instruct the terminal to shift the location where the next characters would appear, to change to bold-face or underlined characters, to clear the screen, etc. All of this wizardry was hard-wired – there were no integrated-circuit CPUs that could be embedded into the box and programmed to produce the desired results. Consequently, terminals were quite expensive (the fancier ones costing as much as a typical new car). Different manufacturers selected their control sequences as much based upon what they could wire in easily as upon any desire for uniformity or comparability. Consequently, there were eventually hundreds of models of CRT-based computer terminals, *all of which used incompatible sets of control sequences*.

Embedded microprocessors eventually simplified the design of computer terminals considerably (sinking a number of companies along the way that had made their money leasing the older expensive models), and the capabilities of computer terminal began to grow, including adding graphics and color capabilities. Eventually, PCs became cheap enough that the whole idea of a dedicated box serving merely as a terminal came into question, and the computer terminal now exists as a separate entity only in very special circumstances, although there are periodic attempts to revive the idea (e.g., so-called Internet appliances).

Before there was a World-Wide Web, there was an Internet. The Internet grew out of a deliberate attempt to allow researchers all around the country access to the limited number of highly expensive mainframe CPU's. Internet traffic originally was dominated by telnet, a protocol for issuing text commands to a computer via the Internet, and FTP, a protocol for transferring files from machine to machine via the Internet. Email came along later.

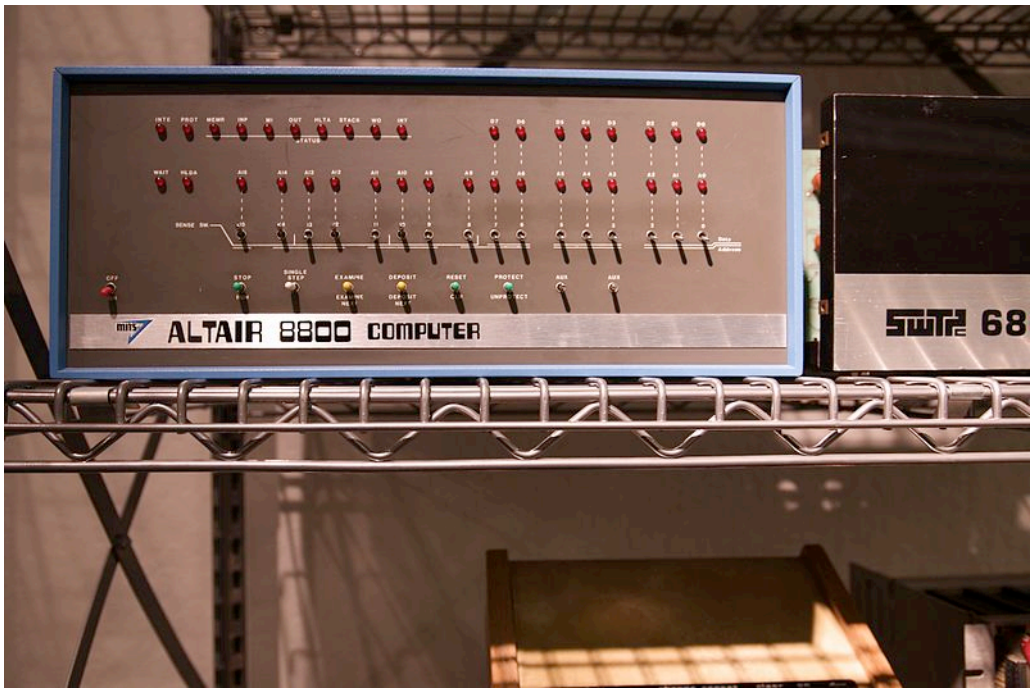
In imitation of (and perhaps in jealousy of) the Internet, UseNet evolved as an anarchic collection of mainframe and minicomputers that each knew a handful of telephone numbers of other UseNet computers and could pass email and news (a.k.a. bulletin board) entries along those connections.

As the idea of long range networking took hold, more and more sites began installing local area networks to enable communication among their own machines.

Unix evolved for minicomputers in an historical context where

- Multiprocessing was expected, and the hardware provided safeguards for protecting one running process from affecting or being affected by other processes on the same CPU.
- The most common displays were computer terminals, which came in many different models, all of which used mutually incompatible control sequences. Most of these could display text only, or text with simple vertical & horizontal line graphics. "True" graphics terminals were not unknown, and were clearly on their way, but were so expensive as to be comparatively rare.
- Networking was common. In fact, it was normal, perhaps even the rule, for users to be controlling, via the network, machines that were remote from the users' actual location.

When personal-computer (PC's) came on the scene, they represented a revolution in terms of both decreased size and decreased cost, but they represented a step backwards in terms of total computing power and in terms of the sophistication of the hardware support for many systems programming activities.



Oddly enough, PC systems seemed to recap the entire history of computing up till that time, though at a somewhat faster pace:

- "One user – One CPU" was a rallying cry of the early PC proponents. They argued that, although an individual PC presented limited CPU power compared to mainframe or mini machines, the individual PC could still provide a single user with more CPU power than that person would receive as their *share* of a mainframe when split over a large number of simultaneous users. So early PC operating systems returned to the single-user, single-process model that had gone out of fashion decades before in the world of larger computers. (Because, after all, that single user didn't really need to split those precious CPU cycles among more than one application at a time, right?)

- Display technology reverted initially to the electric-typewriter style system console. This was quickly supplanted by a "dumb terminal" CRT-and-keyboard, though early printers were still electric typewriter based. (In fact, I recall seeing ads for a device consisting of a panel of solenoids and control circuits that could be placed over the keyboard of an electric typewriter. Send the panel the ASCII code for an "a", and a solenoid "finger" would punch down right where the "a" key would be on a typewriter. Send it the ASCII code for "Z", and a pair of solenoids would strike the typewriter's "shift" and "z" keys.)

  The very existence of integrated circuit CPUs, however, lowered the cost of CRT displays to the point where more elaborate, graphics-capable displays were soon available.

- Network technology was initially spurned. "One user – One CPU", remember? Why would anyone *need* access to other computers. Email and net news could be handled by modem connection without full-fledged networking. It took a surprisingly long time before PC operating systems and applications began to acknowledge that not every bit of information and not every hardware/software resource could economically be replicated on every PC system.

MSDOS was developed in a context where

- "One user – One CPU" was the rule. Multiple processes for a single user were not deemed necessary.
- Most PCs had a CRT display with limited character and graphics capabilities.
- Networking was deemed unnecessary.

As MSDOS evolved into Windows, it did so in response to changes in the HW/SW context:

- "One user – One CPU" remained the rule, but a single user might have multiple processes.
- PC displays could show characters in a variety of fonts, and graphics capabilities were more common.

- Some people might want local networking, but it was supplied by third-party add-ons with minimal support from the operating system itself. As for the Internet, why would anyone with a PC want to communicate with all those mainframe dinosaurs?

Of course, both MS Windows and Unix continued to evolve past their earliest forms, but the contexts in which they have evolved helped establish their fundamental philosophy and continues to influence how they work today.

- Unix users tend to do a lot more typing than Windows users. Graphics capabilities were rare when Unix was developed, so commands had to be typed out rather than always working through a window GUI, and that practice of typing commands continues to influence the "look and feel" of Unix. Unix does have a windowing GUI, but the Unix approach is to launch an application via a typed command, then let that application open up windows if it needs them. Compare to MS Windows, where most users never type a command, and may not even realize that many common Windows applications offer a variety of command line options. (Try, for example, creating shortcuts on your Windows desktop with the Target:

  ```
  C:\WINDOWS\EXPLORER.EXE
  ```

  and another with the Target:

  ```
  C:\WINDOWS\EXPLORER.EXE /n,/e,c:\
  ```

  Try them each and see what they do. The difference is potentially useful, but this possibility is unknown to most Windows users.

- Because graphics capabilities were rare when Unix was developed, many Unix applications are text based. The canonical Unix application reads a stream of text in (from "standard input") and produces a stream of text as output (on "standard output"). Because the output is often considered a slightly modified version of the input, such programs are called filters. Unix users are more likely to string together a bunch of filters that each do something simple than to hunt for a massive dedicated application that does the whole job at once. For example, if you wanted to know how many statements occurred in some file of C++ code, an MS Windows programmer would load that file into a visual C++ programming environment, then search through the menu bars for a "properties" or "number of statements" item that might convey the required information. A Unix programmer would, in a single line of text commands, feed that file into a program that extracted each line containing a ";" (because most C++ statements end with semicolons) and then feed that set of ";"-containing lines through a line-counting filter.

- "One user – One CPU" thinking dominates Windows applications even though networking is widely available. MS Windows has never offered the same level of protection against one process affecting others on the same machine. In Windows, it's still all too common for a single crashing application to lock up the machine to the point where a reboot is required. In Unix, such reboots are quite rare.

  In part, this is because Unix allows processes to interact in only two ways:

  - By having one process write out files that the other reads
  - By having one process write characters into a "pipeline" that is read by another process.

  By contrast, MS Windows has a variety of communication mechanisms, some of which allow one process to directly manipulate the data of another. This allows different applications to work together in interesting and valuable ways (e.g., embedding charts from a spreadsheet directly inside a word-processor document), but at an inevitable cost to overall system security and stability.

- Unix applications are often designed to be run remotely, via a network. Even the Unix windowing GUI, called X, is based on the idea that an application can be run on whatever machine it is installed upon, but will actually display its windows on and accept its mouse clicks and other inputs from any machine on the network. MS Windows, on the other hand, assumes that the application is running on the machine where its outputs should be displayed. If you want to use, say, a paint program that isn't installed on the PC you are sitting at but is installed on another PC in the same room, you can't run that program without actually getting up and moving to the other PC.

- Another instance of this bias, one that is, I suspect, much closer to the hearts and pocketbooks of Windows applications developers, can be seen in the deliberate ignorance of many applications to the realities of the networking world. Although Windows will provide support for different users logging in (one at a time!) to given PC, the majority of application software that a user might install on that PC will assume that a single data area and a single set of option and preference settings are enough for that PC. Therefore every time you change a setting, you affect the behavior of that application not only for yourself, but for all other people who use that same machine. Unix applications, on the other hand, are far more likely to be distributed under site licenses permitting all users at a site equal access.

    - Indeed, Unix applications are far more likely to be distributed with source code. It's worth noting that both the Free Software movement and the Open Source movement originated in the Unix community.

  If you have PCs arranged on a network, so that they can access a common pool of disk drives, when you purchase a new windows application, it is likely to try to install itself onto a single PC, not allowing itself to be run from the other PCs that can access the drive where you install the software. And even if you could run it by from a different PC, you will often find that all of your preferences or options settings and all of your accumulated data is squirreled away in the registry or systems area of the PC on which the software was installed, making it unusable from other PCs.

All this may help to explain why the CS Dept. makes such heavy use of Unix. It's not that we dislike MS Windows. But if we require a particular software package (say, a compiler) in our classes, under Unix we install it on a few Unix machines and let students run it from remote locations via the Internet. Under Windows we have to install it on *every* CS Dept machine, ask that it be installed on every machine in the ODU laboratories (and at the distance learning sites). And when an updated version of that package comes out, we have to go through the entire process all over again. It's just far, far easier to maintain a consistent working environment for all students under Unix.

1 All in the *nix Family

Is Unix still relevant? Will you really ever need to use or work in a Unix environment?

Unix is an *old* operating system. Sometimes its age shows. Some Unix applications seem to use odd choices for interpreting special keys; some interpret mouse clicks in unexpected (to Windows users) ways; some have odd mechanisms for copying and pasting and sometimes the windows look funny. Take something as simple as as copy-and paste. Windows users know that the keyboard shortcut to paste information is Control-v. This is pretty much universal among Windows programs. But in the Unix `emacs` editor, instead of "pasting", you "yank" information with Control-y. Ask an `emacs` fan why `emacs` didn't adopt the same "standard" key stroke as the rest of the world was already using, and you will be given a condescending smile while the fan explains that `emacs` adopted the Control-y yank long before Microsoft even existed, and that the real question is why the rest of the world agreed to the idea that letter "v" was a logical choice to stand for "paste".

Unix is a *new* operating system. It has been continually changed, updated, evolved, and used as the basis for new "Unix-like" operating systems. In fact, it really makes more sense to think of Unix as a family of operating systems (sometimes denoted as "*nix"). There is an actual standards process that allows an operating system to

certify that is it a “real” Unix. But there may be even more machines out there running Unix-like operating systems, in other words, something in the *nix family.

You might well be using a Unix or Unix-like operating system without realizing it:

- The Apple OS/X operating system is a certified, true Unix.
- Linux systems come in many flavors (called “distributions”), most of them available for free. Linux is not certified as Unix, but is so close that it makes no difference to most users.
- The Apple iOS used on iPhones and iPads is based on OS/X and so is in the Unix family.
- Android, the chief competitor to iOS for portable devices, is based on a modified Linux kernel.
- There are a whole host of consumer electronic devices (e.g., DVRs, smart TVs, automobile entertainment systems) that contain CPUs and that are running some version of Linux inside.

So there’s actually a pretty good chance that any computerized system that you lay your hands on that isn’t running Windows will actually be running something in the Unix family. Although some of these hide the operating system deeply enough that you won’t notice it when you use them, if your future profession should involve programming for any of them, you may find yourself working on a *nix system as the closet thing to the platform where your code will eventually run.