10 Things I Hate About (U)NIX

Date: Nov 4, 2005

Return to the article

UNIX was a terrific workhorse for its time, but eventually the old nag needs to be put out to pasture. David Chisnall argues that it's time to retire UNIX in favor of modern systems with a lot more horsepower.

For more information on *nix-based operating systems, visit our Linux Reference Guide or sign up for our Linux Newsletter

In 1971, the *UNIX Time Sharing system, First Edition* was released. This simple operating system allowed multiple users to use a single, low-end minicomputer. It became popular, largely due to the fact that the source code was available for free to universities, which produced a generation of graduates who grew up learning UNIX.

Rather than developing their own operating systems, a lot of companies licensed the UNIX source code and produced their own derivatives to run on their hardware. Eventually, UNIX replaced most commercial operating systems.

In the last decade, free clones and derivatives of UNIX have started to take over from the old-guard UNIX systems. In terms of source code, these versions share very little, if anything, with their predecessors, but in terms of design and philosophy a lot can be traced back to the original roots.

UNIX has a lot of strengths, but like any other design it's starting to show its age. Some of the points listed in this article apply less to some UNIX-like systems, some apply more.

Everything Is a File (Unless It Isn't)

Everything in a UNIX system is a file. Well, except things that aren't files, such as sockets. This is widely regarded as one of the defining points of UNIX. What is a file in UNIX? It's a collection of bytes. No type information is encoded, so the only way of understanding the contents of a file is to already know about it.

The file metaphor is becoming increasingly strained in UNIX:

- Physical disks are files—these have a fixed size, but you can seek to any point on the disk.
- Serial ports are files—these can be read from and written to, but seeking in a serial port has no meaning.
- Normal files are also files, and these can be read sequentially or randomly and even increased or reduced in size. Of course, a program that's given a filename has no way of knowing what kind of operations are possible on a file, other than to try an operation and see if it fails. In good UNIX tradition, some of these operations, such as locking an NFS-shared file, will appear to work but silently fail.

The often stated advantage of this paradigm is that you can connect programs to devices and they'll just work, without being specially designed to interface with the device. This was almost true at one point. Writing to a text-only line printer was exactly the same as writing to a text-only terminal or writing to a text file. Now, however, most people tend to deal in a little more than just plain text. If I have a program that outputs an image, can I just send that image to a terminal and have it display? Can I send it to the printer in the same way? Well, if I happen to have an X server that supports the XPrint extension, the answer is maybe. If I don't, then I have to send it in X drawing calls to the screen, in PostScript to the printer, and in a serialized byte stream to a file. Having the same interface for these devices does no good at all when I have to understand the device on the far end—far less good, in fact, than a higher-level abstraction layer would do me.

Everything Is Text

Everything inside a file is a stream of bytes. That's fine for text. Well, it *would* be fine for text if everyone lived in the USA and only ever needed to use 7-bit ASCII. Unfortunately, some people live elsewhere or have different character-set requirements. All those folks need some sort of mechanism for describing what character set they're using.

The assumption that everything is—and should be—text permeates the UNIX design. Except in a very few cases, commands produce text and expect to produce text. This is rather a strange design choice, considering the other UNIX philosophy of putting everything into the shell, even if it belongs in a shared library. A more intelligent approach would be for the commands to produce typed binary data and have the shell display it, if it were intended for display.

Consider the `ls` command, which lists the contents of a directory. If you wanted the contents sorted in a case-insensitive way, you would pipe the output into `sort`. Now imagine that you want the output sorted by file size. You can make `ls` display the file size, and then tell `sort` to sort it by that column. This is fine, except that then the file sizes are all in bytes (or sometimes allocation units, usually of 512 bytes, depending on your UNIX variant). This is not very human-readable, so you tell `ls` to output the size in human-readable format—in bytes, kilobytes, megabytes, etc. Unfortunately, `sort` doesn't understand that 1MB is bigger than 6KB, so it sorts everything into a silly order. On the other hand, if ls would output a header defining its output as a set of columns with names and types, then you could tell `sort` to sort by the column called *size*, and tell your shell to translate the size into a human-readable form.

No Introspection

The UNIX command line is considered one of its greatest assets. While I like the concept of a command line in principle, I consider the UNIX shell to be one of the worst possible implementations of the ideal.

Anyone who has used `zsh` will be familiar with the incredible auto-completion features it provides. For a great many programs, it can auto-complete command-line options. When you use the `secure copy` program, it can even create a remote shell on the other machine and auto-complete filenames for you. Looking at how this works, however, is likely to give anyone nightmares. For each program (or group of programs, if you're really lucky), a parser is built that scans the help output from that program and creates the auto-completion list. When you add a new program that doesn't produce a help screen, or produces one in a different format, `zsh` is stuck.

Consider now a hypothetical system in which each object, including programs, supported simple introspection methods. In this system, the shell could query the applications with a well-defined interface to get a list of available options. It could also get restrictions on the types of input files and other useful data. In an ideal system, this information would be locale-specific, so the user could enter command-line options in his or her own language.

Introspection is useful in a large number of cases. On Mac OS, the underlying file system supports the most basic form of introspection on files—you can get a 32-bit file type. This isn't ideal, but it at least allows some kind of intelligent decision to be made about which application should be used to open the file. Taking this capability a step further, each file type could have a machine-parsable description of the file structure associated with it. A word processing program could then scan a file from another system and translate it easily into its native format using this information—at the very least, it could extract the plain text and maybe some basic formatting.

X11: Almost a GUI

X11 is a mechanism for displaying pixels on a screen. Beyond that, it supports some very basic windowing abilities. If you want to display things such as buttons and scrollbars, you need a higher-level toolkit.

In itself, this isn't a bad idea. Adding extra layers of abstraction incurs a slight performance penalty, but makes code a lot more maintainable. The first problem is that there is no standard toolkit—or rather, there are several, so it's very difficult for applications to have the same basic look and feel. Of course, the lack of a set of published human interface guidelines for X11 doesn't really help, either.

The main problem with X11 is network transparency. One of the design requirements for X11 was that it would allow remote display. Unfortunately, this is at much too low a level. All X11 does is remote display—all of the logic remains at the opposite end of the connection from the display. If you're on a fast local connection, this distinction doesn't matter much. If you're on the other end of a slow or high-latency connection, however, it really does matter.

At the same time X11 was becoming popular, another system was developed. NeWS was based on PostScript. The most obvious advantage of this origin for developers was that you could send the same PostScript output to the screen as to a printer. The second advantage was that it ran UI components on the machine connected to the screen. If you pressed a button, enough logic would be available locally to respond to the button press and send an event to the server.

On the X11 model, the click is sent to the machine running the application, which then must send each frame in the sequence of actions to the screen. If this seems bad, consider a text field—every time you enter a character, it requires a round trip to the server to display it. Now imagine that you're on an Internet link with a 200ms latency to the server.

The idea behind NeWS was so good that it keeps being reinvented. The latest Internet buzzword, AJAX, is another implementation of the same concept. User interface elements are rendered by a browser, and the back-end server asynchronously provides the application logic. Even before that, Java applets were doing the same thing—drawing the UI in Java and communicating with a remote server running the rest of the application.

Standard Input, Standard Output

On the UNIX command line, you can join programs by using pipes. This is nice for very simple workflows. It lacks flexibility, however, due to the mechanism used to implement it. Originally, programs read input and wrote output. On interactive computers, it became beneficial to have two output streams—one for expected output and one for error messages. Now, on a UNIX system, each program has a single input stream and two output streams.

This limitation means that some things simply cannot be done with a UNIX command line. Consider a filter that de-multiplexes a movie clip into video and audio streams. At best, you could send the video stream to the standard output and the audio to the standard error, but then you would have no means of reporting errors.

To make things worse, each of these streams is unidirectional. This means that there can be no negotiation between two filters. Ideally, the source filter should be able to advertise a selection of output formats, and the destination filter should be able to select one. On a UNIX command line, this must be done in advance by the user. Once the filters are running, there's no way of changing the output format—for example, to reduce the bit-rate if less network bandwidth is available.

Synchronous System Calls

Every time you do a system call in a UNIX system, you have to swap into kernel space—a relatively expensive operation. You have to wait until the kernel completes the operation, and then continue. In some cases, the kernel just copies your parameters and processes later. Consider the case of writing some bytes to a stream. In a UNIX system, you would use the `write` system call. This would switch to kernel space, copy the buffer, and then transmit it later. If you're lucky, later you'll get a notification that the write succeeded (or failed).

If you're writing a lot of data, you end up making a lot of system calls and spending a significant amount of your time switching between privileged and unprivileged modes on your CPU. If you're on an SMP system, as is increasingly common these days, you'll have to wait for all sorts of kernel synchronization code to run first.

On a microkernel system such as QNX, system calls are asynchronous. This means that you can queue them up in a user space buffer and have them all swapped into kernel space when your quantum expires and the kernel interrupts your process. This approach is a lot more scalable on SMP systems and provides a performance boost on single-processor systems by reducing the amount of mode-switching required.

One-Way System Calls

Although partially resolved in Linux by the NetLink API, it's very difficult to get data out of the kernel. There's no standard mechanism for a UNIX kernel to call a function in a UNIX program. In fact, it's very difficult to send more than one bit of information to a user space program from the kernel. The one bit is a signal, a flag saying that something has happened. Consider the example of an asynchronous I/O request. Ideally, you would like some notification when your I/O operation has completed. On UNIX, you receive a signal. This tells you that an asynchronous I/O operation has happened, but not which one—you then have to poll all of them to try to find out.

The lack of this mechanism dramatically increases the number of things that must be done in the kernel. It's a UNIX axiom that policy and mechanism must be separated, and yet it's very difficult to provide mechanism in a UNIX kernel and policy outside, since the kernel has no way of informing a user space delegate that some kind of policy decision is required. The standard UNIX workaround to this problem is to have a program initiate a system call that then blocks until the kernel has something to say and returns at some later time.

Note that this doesn't apply to microkernel UNIX-like systems, such as GNU HURD or Mach. These systems use a very simple kernel that provides a simple interface to the hardware and message-passing facilities, and run everything else in user space. Since the kernel in these systems delegates a lot to user space servers, it needs a mechanism for sending messages, and this mechanism is available to all user space programs.

C: Cross-Platform PDP Assembler

The C language was written to enable UNIX to be portable. It's designed to produce good code for the PDP-11, and very closely maps to that machine's capabilities. There's no support for concurrency in C, for example. In a modern language such as Erlang, primitives exist in the language for creating different threads of execution and sending messages between them. This is very important today, when it's a lot cheaper to buy two computers than one that's twice as fast.

C also lacks a number of other features present in modern languages. The most obvious is lack of support for strings. The lack of bounds-testing on arrays is another example—one responsible for a large number of security holes in UNIX software. Another aspect of C that's responsible for several security holes is the fact that integers in C have a fixed size—if you try to store something that doesn't fit, you get an

overflow. Unfortunately, this overflow isn't handled nicely. In Smalltalk, the overflow would be caught transparently to the developer and the integer increased in size to fit it. In other low-level languages, the assignment would generate an error that could be handled by the program. In C, it's silently ignored. And how big is the smallest value that won't fit in a C integer? Well, that's up to the implementation.

Next, we get to the woefully inadequate C preprocessor. The preprocessor in C works by very simple token substitution—it has no concept of the underlying structure of the code. One obvious example of the limitations of this setup is when you try adding control structures to the language. With Smalltalk, this is trivial—blocks of code in Smalltalk can be passed as arguments, so any message call can be a control statement. In LISP, the preprocessor can be used to encode design patterns, greatly reducing the amount of code needed. C can just about handle simple inline-function equivalents.

The real problem with C, however, is that it's the standard language for UNIX systems. All system calls and common libraries expose C functions, because C is the lowest common denominator—and C is very low. C was designed when the procedural paradigm was only just gaining acceptance, when Real Programmers used assembly languages and structured programming was something only people in universities cared about. If you want to create an object-oriented library on UNIX, you either expose it in the language in which it was written—forcing other developers to choose the same language as you—or you write a cumbersome wrapper in C. Hardly an ideal solution.

Small Tools, Not Small Libraries

The C philosophy is one of small tools doing one thing well. The difference between a small tool and a small library is twofold:

- A small tool has a larger overhead; it needs its own process with all the associated costs.
- A small tool is less flexible. Another program making use of it can invoke it (possibly with some command-line arguments) and then it can only be controlled by the standard input. This is not very flexible, and leads to problems.

Most often cited is the UNIX `mv` command. The designers of UNIX decided that the shell, rather than the application, should be responsible for expanding wildcards. This means that the program being invoked would have no way of knowing whether the arguments were originally a list of files or a wildcard. Imagine the following command line:

```
mv *.exe *.bin
```

If UNIX provided a library for wildcard expansion rather than making the shell do it, the `mv` command would know that you had given it two arguments and that you wanted to rename everything that ends with `.exe` to the same thing with a `.bin` extension. Under UNIX, however, this isn't the case. The `mv` command receives a list of files and tries to rename them all with the name of the last file.

In-Band Signaling as Standard

Any communications engineer will tell you that in-band signaling is a bad idea. In UNIX, it's often the *only* way of doing things. UNIX seems to have a strong philosophical objection to the idea of metadata. The UNIX file system doesn't even include a single bit to mark a file as hidden. Instead, UNIX uses a dot (.) at the start of a filename to mark it as hidden. When Mac OS was using 32-bit file type codes, and DOS-like systems were using a three-character file extension to identify file types, UNIX was embedding the file type in the filename.

In-band signaling is everywhere in UNIX. Everything is a file, so everything is a stream of bytes. While other systems allow other metadata to be associated with files, UNIX just treats them as a long stream. This leads to control structures and data being intermingled because the way applications are connected to each other or to devices is by one pretending to be a file.

Time for U(NIX) 2 Retire

UNIX was a good operating system for its time. It performed well on relatively low-end minicomputers, and provided some of the features that had previously only been available on mainframes. For a PDP-11, it's an ideal choice. Modern operating systems based on the same philosophical design, however, are seriously hampered by compromises made to fit the system onto a machine less powerful than a modern pocket calculator.

For more information on *nix-based operating systems, visit our Linux Reference Guide or sign up for our Linux Newsletter

---