24-Feb-2024 Update: This little article is itself now 20 years old, so it's talking about UNIX 40 years ago! My website is now mrochkind.com. Enjoy!

Has UNIX Programming Changed in 20 Years?

Date: May 28, 2004

With the publication of Advanced UNIX Programming, Second Edition, Marc Rochkind takes a look at what has changed in the 20 years since he wrote the first edition.

When I wrote the first edition of *Advanced UNIX Programming* in 1984, UNIX was already 15 years old. I've just finished the second edition, and another 20 years have passed. It's interesting to ask, how has UNIX Programming changed since 1984? Well, some things are pretty much the same, and, of course, a lot is different.

The Fundamentals Are Mostly Unchanged

What's the same is that UNIX still works in essentially the same way. Processes are created with fork, the program the process runs is chosen by one of the exec calls, you open a file with open, and so on. Many—perhaps most—serious UNIX programs access the Internet nowadays, and those system calls (socket, connect, and so on) are also nearly unchanged since they were introduced in BSD UNIX in the 1980s. Even the primary implementation language, C, is largely unchanged, although it has grown up: Its standard has been revised a few times, and everybody now codes with function prototypes and without assuming anything about the size of integers and pointers.

In fact, as I prepared for the second edition of the book, I discovered that the book's 1984 example code still worked, although the GCC compiler complained a lot about the primitive form of C in which those programs were written.

Some Things Are Very Different

If all the basics are the same, what has changed? Well, these things:

- Number of system calls
- Languages we use
- Subsystems we program
- Need for portability
- Relevance of UNIX standards

More System Calls

The number of system calls has quadrupled, more or less, depending on what you mean by "system call." The first edition of *Advanced UNIX Programming* focused on only about 70 genuine kernel system calls—for example, open, read, and write; but not library calls like fopen, fread, and fwrite. The second edition includes about 300. (There are about 1,100 standard function calls in all, but many of those are part of the Standard C Library or are obviously not kernel facilities.) Today's UNIX has threads, real-time signals, asynchronous I/O, and new interprocess-communication features (POSIX IPC), none of which existed 20 years ago. This has caused, or been caused by, the evolution of UNIX from an educational and research system to a universal operating system. It shows up in embedded systems (parking meters, digital video recorders); inside Macintoshes; on a few million web servers; and is even becoming a desktop system for the masses. All of these uses were unanticipated in 1984.

More Languages

In 1984, UNIX applications were usually programmed in C, occasionally mixed with shell scripts, Awk, and Fortran. C++ was just emerging; it was implemented as a front end to the C compiler. Today, C is no longer the principal UNIX application language, although it's still important for low-level programming and as a reference language. (All the examples in both books are written in C.) C++ is efficient enough to have replaced C when the application requirements justify the extra effort, but many projects use Java instead, and I've never met a programmer who didn't prefer it over C++. Computers are fast enough so that interpretive scripting languages have become important, too, led by Perl and Python. Then there are the web languages: HTML, JavaScript, and the various XML languages, such as XSLT.

Even if you're working in one of these modern languages, though, you still need to know what going on "down below," because UNIX still defines—and, to a degree, limits—what the higher-level languages can do. This is a challenge for many students who want to learn UNIX, but don't want to learn C. And for their teachers, who tire of debugging memory problems and explaining the distinction between declarations and definitions.

TIP

To enable students to learn UNIX without first learning C, I developed a Java-to-UNIX system-call interface that I call *Jtux*. It allows almost all of the UNIX system calls to be executed from Java, using the same arguments and datatypes as the official C calls. You can find out more about Jtux and download its source code from http://basepath.com/aup/. *[now mrochkind.com/aup/index.html]*

More Subsystems

The third area of change is that UNIX is both more visible than ever (sold by Wal-Mart!) and more hidden, underneath subsystems like J2EE and web servers, Apache, Oracle, and desktops such as KDE or GNOME. Many application programmers are programming for these subsystems, rather than for UNIX directly. What's more, the subsystems themselves are usually insulated from UNIX by a thin portability layer that has different implementations for different operating systems. Thus, many UNIX system programmers these days are working on middleware, rather than on the end-user applications that are several layers higher up.

More Portability

The fourth change is the requirement for portability between UNIX systems, including Linux and the BSD-derivatives, one of which is the Macintosh OS X kernel (Darwin). Portability was of some interest in 1984, but today it's essential. No developer wants to be locked into a commercial version of UNIX without the possibility of moving to Linux or BSD, and no Linux developer wants to be locked into only one distribution. Platforms like Java help a lot, but only serious attention to the kernel APIs, along with careful testing, will ensure that the code is really portable. Indeed, you almost never hear a developer say that he or she is writing for XYZ's UNIX. It's much more common to hear "UNIX and Linux," implying that the vendor choice will be made later. (The three biggest proprietary UNIX hardware companies—Sun, HP, and IBM—are all strong supporters of Linux.)

More Complete Standards

The requirement for portability is connected with the fifth area of change, the role of standards. In 1984, a UNIX standards effort was just starting. The IEEE's POSIX group hadn't yet been formed. Its first standard, which emerged in 1988, was a tremendous effort of exceptional quality and rigor, but it was of very little use to real-world developers because it left out too many APIs, such as those for interprocess communication and networking. That minimalist approach to standards changed dramatically when The Open Group was formed from the merger of X/Open and the Open Software Foundation in 1996. Its objective was to include all the APIs that the important applications were using, and to specify them as well as time allowed—which meant less precisely than POSIX did. They even named one of their standards *Spec 1170*, the number being the total of 926 APIs, 70 headers, and 174 commands. Quantity over quality, maybe, but the result meant that for the first time programmers would find in the standard the APIs they really needed. Today, The Open Group's Single UNIX Specification is the best guide for UNIX programmers who need to write portably.

Conclusion

So, yes, UNIX programming has changed a lot, because UNIX and the subsystems that run on it are much more complicated and the language technologies have evolved. But it's still recognizably the same UNIX; and, while many of us no longer program in C, we still acknowledge C as the official UNIX reference and implementation language. This is true even if the UNIX is really Linux, the language is Python, and the application is a web site that's running Apache and MySQL.