RIK FARROW

musings

rik@spirit.com

PROGRAMMING IS AN ART. WHEN different people attempt to accomplish the same tasks, their programs will generally be quite different. And when examined carefully, some programs will stand out as beautiful code, while others could quite easily be described as ugly.

I learned programming in college, working in the painful paradigm of punchcards and mainframes. A single typo meant waiting hours, or even until the next day, to see the results of the correction. While I believe that using punchcards encouraged a certain discipline, in coding and in exactness, it did nothing to instill in me a real understanding of how to write beautiful programs. I could, and did, write robust, functioning code, but it certainly lacked the elegance that I could sometimes discern in other people's code.

Inelegant code, however, has effects that go beyond aesthetics. I once was tasked with writing a text formatter, with requirements similar to the ones found in Kernighan and Plauger's *Software Tools*. I didn't know of that book at the time (1979), but plowed in with vigor. When finished, I had written a program that worked correctly, taking marked-up text, formatting it, and producing a table of contents, all on a computer that used two 8-inch floppy disks for file storage. Compiling the 16-page program took about 15 minutes, and using the finished program, written in PL/Z (Zilog's version of PL/I) took a long time too.

After I left that company, I found out that my replacement had been given the same task, but used BASIC instead. His version of the code ran *three times faster* because BASIC had much better string handling routines than PL/Z. I knew my code would be inefficient in places, and had I rewritten those places in assembler, my code would (likely) have been as fast. But I sure was embarrassed.

Looking Deeper

Today's computers make the computers I learned on look like electro-mechanical calculators. The mainframe I used in college filled a room, required massive cooling, and actually used other computers for its input and output (reading punchcards, writing them to tape, and printing). The noise of the cooling fans was incredible, but so were the blinking lights, and the computers ran

slowly enough that you could actually watch patterns emerge in the display of memory address accesses. With systems like these, every instruction counted.

When we write programs today, we can easily be misled into believing that elegance and efficiency don't matter. Instead, our fast computers can fool us into thinking that everything is running fine. Problems often don't emerge until a program goes into production and fails under real loads, or turns out to include a security flaw that converts code into a back door.

For this issue, I sought out programmers who were willing to write about their art. I was fortunate that Brian Kernighan was willing to share his experience in teaching advanced programming. Brian's article explains how he uses testing to maintain AWK and uses that same testing in his classes. I found myself wondering if I would have been a better programmer had I learned the testing discipline that Brian instills in his students today.

David Blank-Edelman's Perl column also begins by discussing testing in Perl. Various Perl modules provide a framework that can be properly (or poorly) used to aid in building packages that can be tested before installation.

Diomidis Spinellis has written about the effects of the many levels of performance found in modern computer memory. The amount of memory available to run programs at full speed on modern processors is tiny, and each additional level offers lower performance. Diomidis explains how the different levels function, provides hints for improving performance in critical areas, and concludes with an analysis of price/performance of memory that is sure to arouse some discussion.

You will also discover other programming-focused articles. Chaos Golubitsky writes about cflow, a tool she used when analyzing the security of IMAP servers in her LISA '05 paper. Luke Kanies explains why he chose Ruby for his implementation of Puppet. If you have heard about Ruby and are wondering if you should learn it, you should read Luke's article.

Nick Stoughton reports on his work on several standards committees, work that has real impact upon both programming and open source. If you care about these issues, you need to read Nick's report.

Fond Dreams

While I have been busy ranting about the need for new operating system design, Andrew Tanenbaum and his students have been busy writing MINIX 3. I don't know how many times I have written about the need for a small kernel that can be trusted and running services without privileges, in their own protected memory domains, but MINIX 3 actually does this.

Andy wrote MINIX as a tool for teaching operating systems back when the next best thing was UNIX, an operating system that was growing far beyond an easy-to-understand size and was encumbered by copyrights and AT&T lawyers. While we now have open source operating systems, such as Linux and the BSDs, they too have grown in size and complexity over the years. MINIX 3 manages the feat of being a next-generation operating system with an actually comprehensible size. The kernel is only 4000 LoC (almost equally split between C and assembly), and the process management server is 3600 lines of C. The file containing the implementation of execve() is 594 LoC in MINIX 3 (servers/pm/exec.c) and 1496 LoC in Linux (2.6.10/fs/exec.c).

By "next-generation," I mean that MINIX is a microkernel in design and philosophy. Only the kernel runs as privileged, and all other services, including process management, file systems, networking, and all device drivers, run in their own private address spaces. Just moving device drivers out of the kernel and into their own address spaces means that they can crash without crashing the kernel. It also means that system code, including device drivers, can be tested without rebooting, and failed drivers (or servers) can be restarted.

While MINIX 3 is not going to replace your desktop today, it is already a good candidate for embedded systems where robustness, reliability, and a small memory footprint are crucial. Perhaps your cell phone will be running MINIX 3 some day.

What, No Security?

For a change, there is no Security section in this issue of *;login:*. There are two Sysadmin articles, with David Malone writing a detective story about a mysterious flood of HTTP requests and Randolph Langley telling us about software he has created to provide better logging for sudo.

We have two new columns this month. Heison Chak will be writing about VoIP, providing background in this column for later articles that will help system administrators charged with supporting (and implementing) VoIP in their networks. Robert Ferrell has taken charge of the humor department, entertaining us with /dev/random.

The summaries of LISA '05, WORLDS '05, and FAST '05 appear in the back. You might wonder why summaries from December don't appear until April, but if you look at the publishing schedule of *;login:*, you can see that none of these conferences finished before the articles for the February issue were due. I have, of course, read all of these summaries more than once, and I encourage you to see what is being presented in conferences you don't attend.

Finally, we have an Opinion piece from Tom Haynes. Tom writes that he got so excited about OpenSolaris that he just had to do something about it. And he did.