acmqueue Can More Code Mean Fewer Bugs?

The bytes you save today may bite you tomorrow

George V. Neville-Neil

Dear KV,

One of the coders I work with keeps removing my calls to `system()` from my code, insisting that it's better to write code that does the work that I'm doing via the shell. He keeps saying that it's far safer to code using the language we're using than to call out to the shell to get this work done. I would believe that if he didn't add 10 to 20 lines of code just to do what I do in one line with `system()`. How can increasing the number of lines of code decrease the number of bugs?

Happy with the One Liner

Dear One,

You almost had me with your appeal to simplicity, that having a single line with `system()` on it reduces the potential for bugs. Almost, but not quite.

When you call out to the shell from any language, you're not using a single line of code, but thousands. Calling a shell at this point is like using a nuke to kill a flea. That flea will be very dead when you're done, but you've also wasted a lot of energy in killing it, and it may result in collateral damage. Each and every call to `system()` is trusting all that underlying code, and the issue is not only that there are a lot of lines under there, but also that the things the shell can do are extremely powerful—it is probably the most powerful program on any system.

A command shell on any system—Unix, Windows, or otherwise—is there to command the system, and it has accreted to itself, over time, the ability to do things to the system in a single line that really should require a bit more thought. The obvious example is moving and removing files. I would actually like to think that most programmers know better than to do something like calling `system()` with an `rm` command, in particular one in which they supply unchecked user input to the call to `system()`. While I say that “I’d like to think that,” a voice—a very loud voice—in the back of my head is screaming, “It’s not so! It’s not so! They’ll do it! Stop them now!” I hate that voice, but no matter how I try to drown it out—and I’ve tried—there it remains.

A worse offense than invoking the `rm` command in a `system` command is calling a shell script via `system()`. Why? Because the poor sap reading your code later will probably have no idea what the script does. I’m sure that it will have a descriptive name such as `update_sales_2.sh`, and that it will be checked in to the source repository rather than being stored in the `/home/bob/update_test2/` directory, but perhaps it won’t, and the fragile setup I describe here will ring true: when Alice comes to read the code, she will then have to go read `update_sales_2.sh`, so long as she has read access to Bob’s home directory. She will be reading it at 3 a.m. when something has broken, and this will all go well and everyone will live happily ever after.

Perhaps my very favorite abuse of the `system()` routine in code is when it’s used to build up a

complex pipeline of commands. Using pipes in a call to `system()` in a program is a fine way to launch a fork bomb into your system, especially if the pipes are built up on the fly.

Running a complex piped command from a shell by hand has less risk of hammering the machine because humans are slow—and supposedly they’re paying attention to what they’re doing. Once you put a set of pipes into `system()` and then let your code run unattended, you run the risk, should there be a bug in your pipeline, of repeatedly forking subprocesses and overwhelming whatever machine you’re running on.

In the shell’s defense, it does handle pipelines better than most programming languages, as anyone who has tried to use pipes and signals in C would readily acknowledge, but forking processes automatically has to be done with great care. Too many times I’ve seen coders work out a nice pipeline incrementally in their terminal windows, and then cut and paste that into a program that will be executed not by hand, but by another program. The look on their faces when the system running their pipeline in a loop destroys their system is somewhat amusing, but hardly makes up for the fact that they, or their whole work group, are about to lose work because a system reset is necessary.

Finally, and this is probably the most important and most subtle argument against using `system()`, doing so requires the programmer reading the code to mentally context-switch from one language to another. Unless you’re calling `system()` in a shell script, the language that the programmer is reading when reaching the call to `system()` is very much not shell; it’s C, C++, Python, Perl, Ruby, or something else. That means all of the mental context that you have built up while working on the code is about to be lost as you bring in the shell-scripting context, or you’re simply going to gloss over it and make a mistake because you aren’t thinking in shell when you get to the call to `system()`.

It’s not that this cannot be done, but it definitely increases the cognitive load on the reader, so you had better have a very good reason for switching into the shell—something better than not wanting to figure out how the `unlink` system call works.

KV

Dear KV,

Why do some modern network protocols not have sequence numbers? I would think that by now all protocol designers would have realized that having a simple sequence number in each packet helps people in debugging their network setups.

Out of Sequence

Dear OoS,

You might as well ask why people insist on not wearing seatbelts after all of the years that particular technology has been proven to save lives.

People will, it seems, persist in the optimistic belief that everything will be OK so long as they are otherwise careful. They think that bad things happen only to other people’s protocols, or packets, but not to theirs. Hope springs eternal and dies in the cold, cold winter of experience.

I want to make two points in response to your plea for sanity in network-protocol design. The first is that it’s not just having a sequence number that is important, but how the sequence number is used is important as well. Consider the sequence number in TCP, which counts the bytes that have been communicated between two endpoints. When TCP was designed, the fastest network

in common use was a 10-Mbps Ethernet LAN. Pay attention, that's an M, not a G—10 *megabits* per second. At 10 Mbps, transmitting 2^{32} bytes of data takes approximately 3,400 seconds, or just less than an hour, which is an eternity to a computer. On commodity 10-Gbps hardware available today, it takes 3.4 seconds to transmit the same data, meaning that the sequence space rolls over about every four seconds. If a packet is lost for more than four seconds, there is a nonzero probability that data on the connection will get munged. With hardware that will be available quite soon, the time will drop to 0.3 seconds for the sequence space to roll over.

None of this is to say that TCP was poorly designed (heck, at least it had a sequence number), but it is important for designers of modern protocols to understand the future proofing-vs.-space tradeoffs when selecting a sequence number. If at some point TCP is extended, then the sequence number could be increased to 64 bits, which even at 100 Gbps would require 46 years to roll the number over. Any packet lost in the network that long will be quite lost indeed. When you choose a sequence number, consider what you're protecting. With TCP it is protecting all the bytes transmitted, so that none is lost or reordered on delivery. With other protocols it might be necessary to count whole messages only, so that the receiver can say that packet A arrived before packet B rather than worrying about every byte in the message.

The second point I would like to make is that timestamps are not good sequence numbers. While it is common to believe that time always moves forward, this is often not the case in computing. Many bugs crop up in dealing with time on computers, not the least of which is that different clocks, on different computers, often proceed at different paces. This is why we have protocols such as NTP (Network Time Protocol) and PTP (Precision Time Protocol) to discipline our computer clocks. Alas, computers don't like to be disciplined, and even when running a time protocol, the clocks on two computers are always somewhat offset from each other, so running a time protocol does not solve this problem. Leaving aside the mind-bending relativity problems of computer timekeeping—and trust me, you really want to leave those aside—the fact remains that using the time on a computer as a packet sequence number is problematic. Incrementing a counter is easier, faster, and less error-prone than making sure that the timestamp you received is monotonically increasing. For the case of packet sequencing, simpler is better—and simpler is a counter.

To those who design or hope to design network protocols, please, I beg of you, do not skimp on the sequencing numbers. The bytes you save today will bite you on the ass tomorrow.

KV

KODE VICIOUS, known to mere mortals as George V. Neville-Neil, works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler who currently lives in New York City.

© 2012 ACM 1542-7730/11/0800 \$10.00