Popping Kernels

CHOOSING BETWEEN PROGRAMMING IN THE KERNEL OR IN USER SPACE

Dear KV,

I've been working at the same company for more than a decade now, and we build what you can think of as an appliance—basically a powerful server that's meant to do a single job, instead of operating as a general-purpose system. When we first started building this system, nearly all the functionality we implemented was added to the operating-system kernel as extensions and kernel modules. We were a small team and capable C programmers, and we felt that structuring the system this way gave us more control over the system generally, as well as significant performance gains since we didn't have to copy memory between the kernel and user space to get work done.

As the system expanded and more developers joined the project, management started to ask questions about why we were building software in such a difficult-to-program environment and with an antiquated language. HR complained that they could not find sufficient, qualified engineers to meet the demands of management for more hands to make more features. Eventually, the decision was made to move a lot of functions out of the kernel and into user space. This resulted in a split system, where nearly everything had to go through the kernel to get to any other part of the system, which resulted in lower performance as well as a large number of systemic errors. I have to admit that those errors, if they occurred in the kernel, would

who is
KV?

click for video

have caused the system to panic and reboot, but even in user space, they caused functions to restart, losing state and causing service interruptions.

For our next product, management wants to move nearly all the functions into user space, believing that by having a safer programming environment, the team can create more features more quickly and with fewer errors. You talk about kernel programming from time to time; do you also think that the kernel is not for “mere mortals” and that most programmers should stick to working in the safer environment of user space?

Safety First

Dear Safety,

The wheel of karma goes round and round and spares no one, including programmers, kernel, user space, or otherwise.

Programming in user space is safer for a very small number of reasons, not the least of which is the virtual memory system, which tricks programs into believing they have full control over system memory and catches a small number of common C-language programming errors, such as touching a piece of memory that the program has no right to touch. Other reasons include the tried-and-true programming APIs that operating systems have now provided to programs for the past 30 years. All of which means that programmers can possibly catch more errors before their code ships, which is great news—old news—but great news. What building code in user space does not do is solve the age-old problems of isolation, composition, and efficiency.

If what you’re trying to build is a single program that takes some input and transforms it into another form, think of common tools such as `sed`, `diff`, and `awk`, and then, yes,

Coordination is coordination, whether you do it in the kernel, user space, or with pigeons passing messages, though the first two places have fewer droppings to clean up.

those programs are perfectly suited to user space. What you describe is a system that likely has more interactions with the outside world than it has with a typical end user.

Once we move into the world of high-throughput and/or low-latency systems for the processing of data, such as a router, high-end storage device, or even some of the current crop of devices in the Internet of things (see “IoT: The Internet of Terror”; <http://queue.acm.org/detail.cfm?id=3121440>), then your system has a completely different set of constraints, and most programmers are not taught how to write code for this environment; instead, they learn it through very painful experience. Of course, trying to explain that to HR, or management, is a lot like beating your head on your desk—it only feels good when you stop.

You say you’ve been at this for a while, so surely you have already seen that things that are hard to do right in the kernel are nearly as hard to get right in user space and rarely perform as well. If your problem must be decomposed into a set of cooperating processes, then programming in user space is the exact same problem as programming in the kernel, only with more overhead to pay for whatever bastard form of interprocess communication you use. My personal favorite form of this stupidity is when programmers build systems in user space, using shared memory, and then reproduce every possible contortion of the locking problem seen in kernel programming. Coordination is coordination, whether you do it in the kernel, user space, or with pigeons passing messages, though the first two places have fewer droppings to clean up.

The tension in any of these systems is between performance and isolation. Virtual memory—which gives

us the user/kernel space split and the process model of programming whereby programs are protected from each other—is just the most pervasive form of isolation. If programmers were really trusting, then they would have all their code blended into a single executable where every piece of code could touch every piece of memory, but we know how that goes. It goes terribly. What is to be done?

Over the past few years, there have been a few technological innovations that might help with this problem, including new systems programming languages such as Rust and Go, which have more built-in safety, but they have yet to prove their worth in a systems environment such as an operating system. No one is replacing a Unix-like operating system with something written in Go or Rust just yet. Novel computer architectures such as the work on Capabilities carried out in the CHERI project, developed at SRI International and the University of Cambridge, might also make it possible to decompose software for safety and retain a high level of performance in the overall system, but again, that has yet to be proven in a real deployment of the technology.

For the moment, we are stuck with the false security of user space, where we consider it a blessing that the whole system doesn't reboot when a program crashes, and we know how hard it is to program in the wide open, single address space of an operating-system kernel.

In a world in which high-performance code continues to be written in a fancy assembler, aka C, with no memory safety and plenty of other risks, the only recourse is to stick to software-engineering basics. Reduce the amount of code in harm's way (also known as the attack surface),

Related articles

A Nice Piece of Code

George V. Neville-Neil

Colorful metaphors and properly reusing functions

<https://queue.acm.org/detail.cfm?id=2246038>

The Cost of Virtualization

Ulrich Drepper

Software developers need to be aware of the compromises they face when using virtualization technology.

<https://queue.acm.org/detail.cfm?id=1348591>

Unikernels: Rise of the

Virtual Library Operating System

Anil Madhavapeddy and David J. Scott

What if all the software layers in a virtual appliance were compiled within the same safe, high-level language framework?

<https://queue.acm.org/detail.cfm?id=2566628>

keep coupling between subsystems efficient and explicit, and work to provide better tools for the job, such as static code checkers and large suites of runtime tests.

Or, you know, just take all that carefully crafted kernel code, chuck it into user space, and hope for the best. Because, as we all know, hope is definitely a programming best practice.

KV

Code 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. Neville-Neil is the co-author with Marshall Kirk McKusick and Robert N. M. Watson of The Design and Implementation of the FreeBSD Operating System (second edition). He is an avid bicyclist and traveler who currently lives in New York City.

Copyright © 2017 held by owner/author. Publication rights licensed to ACM.