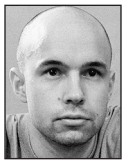
OPINION

The Rise and Fall of the Operating System

ANTTI KANTEE



Antti has been an open source OS committer for over 15 years and believes that code which works in the real world is not born, it is made. He is a fan of simplest possible solutions. Antti lives in Munich and can often be seen down by the Isar River when serious thinking is required. pooka@rumpkernel.org

An operating system is an arbitrary black box of overhead that enables well-behaving application programs to perform tasks that users are interested in. Why is there so much fuss about black boxes, and could we get things done with less?

Historical Perspective

Computers were expensive in the '50s and '60s. For example, the cost of the UNIVAC I in 1951 was just short of a million dollars [1]. Accounting for inflation, that is approximately nine million dollars in today's money. It is no wonder that personal computing had not been invented back then. Since it was desirable to keep millions of dollars of kit doing something besides idling, batch scheduling was used to feed new computations and keep idle time to a minimum.

As most of us intuitively know, reaching the solution of a problem is easier if you are allowed to stumble around with constant feedback, as compared to a situation where you must have holistic clairvoyance over the entire scenario before you even start. The lack of near-instant feedback was a problem with batch systems. You submitted a job, context switched to something else, came back the next day, context switched back to your computation, and discovered the proverbial missing comma in your program.

To address the feedback problem, time-sharing was invented. Users logged into a machine via a teletype and got the illusion of having the whole system to themselves. The time-sharing operating system juggled between users and programs. Thereby, poetic justice was administered: the computer was now the one context-switching, not the human. Going from running one program at a time to running multiple at the "same" time required more complex control infrastructure. The system had to deal with issues such as hauling programs in and out of memory depending on if they were running or not (swapping), scheduling the tasks according to some notion of fairness, and providing users with private, permanent storage (file systems). In other words, 50 years ago they had the key concepts of current operating systems figured out. What has happened since?

It's Called Hardware Because It Makes Everything Hard

When discussing operating systems, it is all but mandatory to digress to hardware, another black box. After all, presenting applications with a useful interface to hardware is one of the main tasks of an operating system, time-sharing or otherwise. So let's get that discussion out of the way first. The question is: why does hardware not inherently present a useful interface to itself? We have to peer into history.

I/O devices used to be simple, very simple. The intelligent bit of the system was the software running on the CPU. It is unlikely that manufacturers of yore desired to make I/O devices simpler than what they should be. The back-then available semiconductor technologies simply did not feasibly allow building complex I/O devices. An example of just how hopeless hardware used to be is the *rotational delay* parameter in old versions of the Berkeley Fast File System. That parameter controlled how far apart, rotationally speaking, blocks had to be written so that contiguous I/O could match the spinning of the disk. Over the years, adding

more processing power to storage devices became feasible, and we saw many changes: fictional disk geometry, I/O buffering, non-spinning disks, automated bad block tracking, etc. As a result of the added processing power, approaches where the systems software pretends it still knows the internal details of devices, e.g., rotational delay, are obsolete or at least faltering.

As a result of added I/O device processing power, what else is obsolete in the software/hardware stack? One is tempted to argue that everything is obsolete. The whole hardware/software stack is bifurcated at a seemingly arbitrary position which made sense 30 years ago, but no longer. Your average modern I/O device has more computing power than most continents had 30 years ago. Pretending that it is the same dumb device that needs to be programmed by flipping registers with a sharpened toothpick results in sad programmers and, if not broken, at least suboptimal drivers. Does doing 802.11 really require 30k+ lines of driver code (including comments), 80k+ lines of generic 802.11 support, and a 1 MB firmware to be loaded onto the NIC? For comparison, the entire 4.3BSD kernel from 1986 including all device drivers, TCP/IP, the file system, system calls, and so forth is roughly 100k lines of code. How difficult can it be to join a network and send and receive packets? Could we make do with 1k lines of system-side code and 1.01 MB of firmware?

The solution for hardware device drivers is to push the complexity where it belongs in 2015, not where it belonged in 1965. Some say they would not trust hardware vendors to get complex software right, and therefore the complexity should remain in software running on the CPU. As long as systems software authors cannot get software right either, there is no huge difference in correctness. It is true that having most of the logic in an operating system does carry an advantage due to open source systems software actually being open source. Everyone who wants to review and adjust the 100k+ lines of code along their open source OS storage stack can actually do so, at least provided they have some years of spare time. In contrast, when hardware vendors claim to support "open source," the open source drivers communicate with an obfuscated representation of the hardware, sometimes through a standard interface such as SATA AHCI or HD audio, so in reality the drivers reveal little of what is going on in the hardware.

The trustworthiness of complex I/O devices would be improved if hardware vendors truly understood what "open source" means: publishing the most understandable representation, not just any scraps that can be run through a compiler. Vendors might prefer to not understand, especially if we keep buying their hardware anyway. Would smart but non-open hardware be a disaster? We can draw some inspiration from the automobile industry. Over the previous 30 years, we lost the ability to fix our cars and tinker with them. People like to complain about the loss of that ability. Nobody remembers to complain about how much better modern cars perform when they are working as expected.

Technology should encapsulate complexity and be optimized for the common case, not for the worst case, even if it means we, the software folk, give up the illusion of being in control of hardware.

If It Is Broken, Don't Not Fix It

The operating system is an old concept, but is it an outdated one? The early time-sharing systems isolated users from other users. The average general purpose operating system still does a decent job at isolating users from each other. However, that type of isolation does little good in a world that does not revolve around people logging into a time-sharing system from a teletype. The increasing problem is isolating the user from herself or himself.

Ages ago, when those who ran programs also wrote them, or at least had a physical interaction possibility with the people who did, you could be reasonably certain that a program you ran did not try to steal your credit card numbers. Also, back then your credit card information was not on the machine where you ran code, which may just as well be the root cause as to why nobody was able to steal it. These days, when you download a million lines of so-so trusted application code from the Internet, you have no idea of what happens when you run it on a traditional operating system.

The time-sharing system also isolates the system and hardware components from the unprivileged user. In this age when everyone has their own hardware—virtual if not physical—that isolation vector is of questionable value. It is no longer a catastrophe if an unprivileged process binds to transport layer ports less than 1024. Everyone should consider reading and writing the network medium as unlimited due to hardware no longer costing a million dollars, regardless of what an operating system does. The case for separate system and user software components is therefore no longer universal. Furthermore, the abstract interfaces that hide underlying power, especially that of modern I/O hardware, are insufficient for high-performance computing. If the interfaces were sufficient, projects looking at unleashing the hidden I/O power [3, 4] would not exist.

In other words, since the operating system does not protect the user from evil or provide powerful abstractions, it fails its mission in the modern world. Why do we keep on using such systems? Let us imagine the world of computing as a shape sorter. In the beginning, all holes were square: all computation was done on a million-dollar machine sitting inside of a mountain. Square pegs were devised to fit the holes. The advent of time-sharing brought better square pegs, but it did so in the confines of the old scenario of the mountain-machine. Then the world of computing diversified. We got personal computing, we got mobile devices, we got IoT, we got the cloud. Suddenly, we

had round holes, triangular holes, and the occasional trapezoid and rhombus. Yet, we are still fascinated by square-shaped pegs, and desperately try to cram them into every hole, regardless of whether they fit.

Why are we so fascinated with square-shaped pegs? What happens if we throw away the entire operating system? The first problem with that approach is, and it is a literal show-stopper, that applications will fail to run. Already in the late 1940s computations used subroutine libraries [2]. The use of subroutine libraries has not diminished in the past 70 years, quite to the contrary. An incredible amount of application software keeping the Internet and the world running has been written against the POSIX-y interfaces offered by a selection of operating systems. No matter how much you do not need the obsolete features provided by the square peg operating system, you do want the applications to work. From-scratch implementations of the services provided by operating systems are far from trivial undertakings. Just implementing the 20-or-so flags for the `open()` call in a real-world-bug-compatible way is far from trivial.

Assuming you want to run an existing libc/application stack, you have to keep in mind that you still have roughly 199 system calls to go after `open()`. After you are done with the system calls, you then have to implement the actual components that the system calls act as an interface to: networking, file systems, device drivers, etc. After all that, you are finally able to get to the most time-consuming bit: testing your implementation in the real world and fixing it to work there. In essence, we are fascinated by square-shaped pegs because our applications rest on the support provided by those pegs. That is why we are stuck in a rut and few remember to look at the map.

There Is No Such Thing as Number One

The guitarist Roy Buchanan was confronted with a yell from the audience titling him as number one. Buchanan's response was: "There is no such thing as number one ... but I love you for thinking about it, thank you very much." The response contains humble wisdom: no matter how good you are at some style(s), you can never be the arch master of all the arts. Similarly, in the ages past the mountain-machine, there is no one all-encompassing operating system because there are so many styles to computing. We need multiple solutions for multiple styles. The set presented below is not exhaustive but presents some variations from the mountain-machine style.

Starting from the simplest case, there is the embedded style case where you run one trust-domain on one piece of hardware. There, you simply need a set of subroutines (drivers) to enable your application to run. You do not need any code that allows the single-user, single-application system to act like a time-sharing system with multiple users. Notably, the single-application system is even simpler and more flexible than the single-user system [5], which, in turn, is simpler and more flexible than the multi-user system.

Second, we have the cloud. Running entire time-sharing systems as the provisioning unit on the cloud was not the ticket. As a bootstrap mechanism it was brilliant: everything worked like it worked without virtualization, so the learning curve could be approximated as having a zero-incline. In other aspects, the phrase "every problem in operating systems can be solved by *removing* layers of indirection" was appropriate. The backlash to the resource wastage of running full operating systems was containers, i.e., namespace virtualization provided by a single time-sharing kernel.

While containers are cheaper, the downside is the difficulty in making guarantees about security and isolation between guests. The current cloud trend is gearing towards *unikernels*, a term coined and popularized by the MirageOS project [6], where the idea is that you look at cloud guests just like you would look at single-application hardware. The hypervisor provides the necessary isolation and controls guest resource use. Since the hypervisor exposes only a simple hardware-like interface to the guest, it is much easier to reason about what can and should happen than it is to do so with containers. Also, the unikernel can be optimized for each application separately, so the model does not impose limiting abstractions either. Furthermore, if you can reasonably partition your computations so that one application instance requires at most one full-time core, most of the multi-core programming performance problems simply disappear.

We also need to address the complex general purpose desktop/mobile case, which essentially means striking a balance between usability and limiting what untrusted applications can do. Virtualization would provide us with isolation between applications, but would it provide too much isolation?

Notably, when you virtualize, it is more difficult to optimize resource usage, since applications do not know how to play along in the grand ecosystem. For the cloud, that level of general ignorance is not a huge problem, since you can just add another datacenter to your cloud.

You cannot add another datacenter into your pocket in case your phone uses the local hardware resources in an exceedingly slack manner. Time will tell if virtualization adapted for the desktop [7] is a good enough solution, or if more fine-grained and precise methods [8] are required, or if they both are the correct answer given more specific preconditions. Even on the desktop, the square peg is not the correct shape: we know that the system will be used by a single person and that the system does not need to protect the user from non-existent other users. Instead, the system should protect the user from malware, spyware, trojans, and anything else that can crawl up the network pipe.

What We Are Doing to Improve Things

We can call them drivers, we can call them components, we can call them subroutines, we can call them libraries, but we need the pegs at the bottom of the computing stack for our applications to work. In fact, everything apart from the topmost layer of the software stack is a library. These days, with virtually unlimited hardware, it is mostly a matter of taste whether something is a "system driver" or "application library."

Rolling your own drivers is a hopeless battle. To address that market, we are providing componentized, reusable drivers at **http://rumpkernel.org/**. Those drivers come unmodified from a reputable kernel. Any approach requiring modification (aka porting) and maintenance induces an unbearable load for anything short of the largest projects with vast amounts of developer resources.

Treating the software stack as a ground-up construction of driver components gives the freedom to address each problem separately, instead of trying to invent ways to make the problem isomorphic to a mountain-machine. Drivers lifted from a time-sharing system will, of course, still exhibit time-sharing characteristics—there is no such thing as number one with drivers either. For example, the TCP/IP driver will still prevent non-root from binding to ports less than 1024. For example, in a unikernel, you are free to define what root or non-root means or simply compile the port check out of the driver. You can perform those modifications individually to suit the needs of each application. As a benefit, applications written for time-sharing-y, POSIX-y systems will not know what hit them. They will simply work because the drivers provide most everything that the applications expect.

We ended up building a unikernel based on the drivers offered by rump kernels via rumpkernel.org: *Rumprun*. We were not trying to build an OS-like layer but one day simply realized that we could build one which would just work, with minimal effort. The noteworthiness of the Rumprun unikernel does not come from the fact that existing software such as Nginx, PHP, and mpg123 can be cross-compiled in the normal fashion and then run directly on the cloud or on bare metal. The noteworthiness comes from the fact that the implementation is a few thousand lines of code ... plus drivers. The ratio of drivers to "operating system" is on the order of 100:1, so there is very little *operating* system in there. The Rumprun implementation is that of an *orchestrating* system, which conducts the drivers.

Conclusion

Time-sharing systems were born over 50 years ago, a period from which we draw our concept of the operating system. Back then, hardware was simple, scarce, and sacred, and those attributes drove the development of the concepts of the system and the users. In the modern world, computing is done in a multitude of ways, and the case for the all-encompassing operating system has been watered down. Advances in semiconductor technology have enabled hardware to be smart, but hardware still exposes dumb interfaces, partially because we are afraid of smart hardware.

The most revered feature of the modern operating system is support for running existing applications. Minimally implemented application support is a few thousand lines of code plus the drivers, as we demonstrated with the Rumprun unikernel. Therefore, there is no reason to port and cram an operating system into every problem space. Instead, we can split the operating system into the "orchestrating system" (which also has the catchy OS acronym going for it) and the drivers. Both have separate roles. The drivers define what is possible. The orchestrating system defines how the drivers should work and, especially, how they are not allowed to work. The two paths should be investigated relatively independently as opposed to classic systems development where they are deeply intertwined.

References

[1] http://www.computerhistory.org/timeline/?category=cmptr.

[2] M. Campbell-Kelly, "Programming the EDSAC: Early Programming Activity at the University of Cambridge," *IEEE Annals of the History of Computing,* vol. 2, no. 1 (January–March 1980), pp. 7–36.

[3] S. Peter, J. Li, Irene Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, T. Roscoe, "Arrakis: The Operating System Is the Control Plane," *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation,* (2014), pp. 1–16.

[4] L. Rizzo, "netmap: A Novel Framework for Fast Packet I/O," *Proceedings of the USENIX Annual Technical Conference* (2012), pp. 101–112.

[5] B. Lampson and R. Sproull, "An Open Operating System for a Single-User Machine," *ACM Operating Systems Rev.,* vol. 11, no. 5 (Dec. 1979), pp. 98–105.

[6] MirageOS: https://mirage.io/.

[7] Qubes OS: https://www.qubes-os.org/.

[8] Genode Operating System Framework: http://genode.org/.