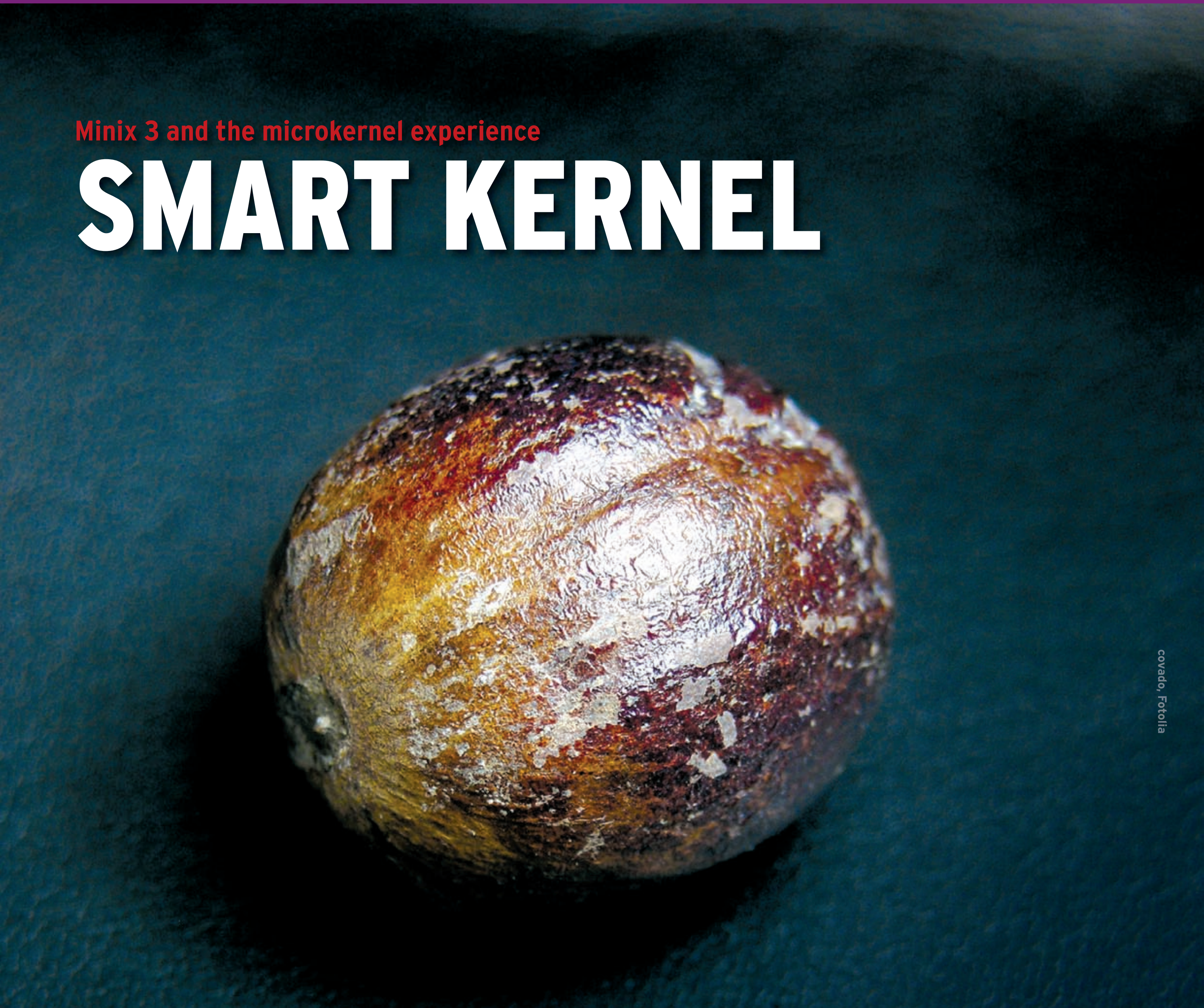
Minix 3 and the microkernel experience

SMART KERNEL



Minix is often viewed as the spiritual predecessor of Linux, but these two Unix cousins could never agree on the kernel design. Now a new Minix with a BSD-style free license is poised to attract a new generation of users. **BY RÜDIGER WEIS**

Linux has a long and stormy relationship [1] with another Unix-like operating system known as Minix [2]. Noted author and computer scientist Andrew S. Tanenbaum released the first version of Minix in 1987 as a tool for teaching students about operating systems, and this small and well-documented system soon became popular with OS enthusiasts. In a post to the Minix newsgroup, upstart Finnish undergraduate Linus Torvalds announced his own experimental system in 1991, and many early Linux contributors came from the ranks of the Minix community.

But Tanenbaum and Torvalds clashed early over issues of design. Tanenbaum has always favored the microkernel architecture, a distinguishing feature of Minix to this day (see the box titled "Why Can't Computers Just Work All the Time?"). Linus, on the other hand, built Linux with a monolithic kernel, with filesystems, drivers, and other components incorporated into the kernel. In a famous post to the Minix group, the Minix creator referred to Linux as "… a giant step back to the 1970s," and a confident reply from young Torvalds to this leading expert in the field of operating systems is early evidence of his now-legendary directness. Still, Linus has acknowledged the importance of Tanenbaum's work to the formation of his own ideas. In his autobiography *Just for Fun* [3], Linus refers to Tanenbaum's *Operating Systems: Design and Implementation* as the book that changed his life.

The debate about micro- versus monolithic kernels goes on to this day, and

THE AUTHOR

Rüdiger Weis is a professor of system programming at TFH Berlin. Between 2002 and 2005, he was involved in post-doctorate research on secure operating systems under Professor Andrew S. Tanenbaum at the Vrije Universiteit, Amsterdam.

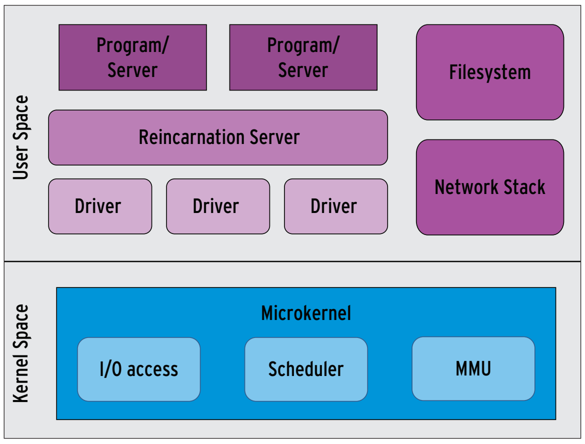


Figure 1: The Minix microkernel encapsulates many subsystems in user space, including drivers, the filesystem, and the network stack. The kernel just runs critical functions, such as underlying I/O, schedulers, and memory management.

just as Linux didn't fade away, neither did Minix. Version 3 of the Minix operating system is designed with the objective of creating a system that is more secure and reliable than comparable POSIX systems, and a BSD-style open source license makes the latest Minix a strong candidate for production as well as educational uses.

Minix is even attracting the attention of some major sponsors. The EU is now sponsoring the project with several million Euros of funding, and Google has a number of Minix projects in its "Summer of Code" program.

Minix 3 runs on 32-bit x86 CPUs, as well as on a number of virtual machines including Qemu, Xen, and VMware. The operating system includes an X Window System (X11), a number of editors (Emacs and Vi), shells (including bash and Zsh), GCC, script languages such as Python and Perl, and network tools such as SSH. A small footprint and crash-resistant design make Minix a good candidate for embedded systems, and its superior stability has led to a promising new role as a firewall system.

Insecure by Design

The security problems facing the current crop of operating systems, including Windows, but also including Linux, are the result of design errors. The errors were inherited for the most part from their predecessors of the 1960s. Most of these problems can be attributed to the fact that developers aren't perfect. Humans make mistakes. Of course, it would be nice to reduce the numbers and mitigate the effects; however, designers have frequently been far too willing to compromise security and a clean design for speed. Tanenbaum refers to this as a "Faustian pact."

In addition to the issues related to sheer size, monolithic designs are also prone to inherent structural problems: Any error is capable of endangering the whole system. A fundamental design error is that current operating systems do

The Question of Extension

Many developers and users disagree with Tanenbaum's doctrine, which he has maintained for over a decade, of being very cautious about introducing extensions to the kernel. Tanenbaum's measure of reasonable operating system complexity is a system that can be taught in a single term. Modularity makes it possible to complete the development of a practically deployable solution in the scope of a thesis. Examples of this are ports for various processor architectures, modifications to Minix for Xen virtualization, and security applications.

In his memoir [3], Linus Torvalds states his reason for rejecting the microkernel architecture for Linux. "The theory behind the microkernel was that you split the kernel into fifty independent parts, and each of the parts is a fiftieth of the complexity. But everybody ignores the fact that the communication among the parts is actually more complicated than the original system was – never mind the fact that the parts are still not trivial." A messy monolithic system can thus offer some performance and scalability benefits, even if it lacks the stability of a microkernel.

not follow the Principle Of Least Authority (POLA). To put this simply, POLA states that developers should distribute systems over a number of modules so an error in one module will not compromise the security and stability of other modules. They should also make sure that each module only has the rights that it actually needs to complete its assigned tasks.

Continued operating system growth comes with the integration of new drivers. Monolithic systems build device drivers into the kernel, which means that a driver error can compromise the stability of the whole system. Closed source drivers in particular endanger system security. According to Tanenbaum, building a closed source driver into the kernel is like accepting a sealed package from a stranger and bringing it into the cockpit of a plane.

Transparent Architecture

Minix is probably the most fully documented operating system around. *The Minix Book* by Tanenbaum and Woodhull [4] is the primary reference. Numerous publications on new features and ongoing research are found on the Minix

Why Can't Computers Just Work All the Time? *By Andrew S. Tanenbaum*

Computer users are changing. Ten years ago, most computer users were young people or professionals with lots of technical expertise. When things went wrong – which they often did – they knew how to fix things. Nowadays, the average user is far less sophisticated, perhaps a 12-year-old girl or a grandfather. Most of them know about as much about fixing computer problems as the average computer nerd knows about repairing his car. What they want more than anything else is a computer that works all the time, with no glitches and no failures.

Many users automatically compare their computer to their television set. Both are full of magical electronics and have big screens. Most users have an implicit model of a television set: (1) you buy the set; (2) you plug it in; (3) it works perfectly without any failures of any kind for the next 10 years. They expect that from the computer, and when they do not get it, they get frustrated. When computer experts tell them: "If God had wanted computers to work all the time, He wouldn't have invented RESET buttons" they are not impressed.

For lack of a better definition of dependability, let us adopt this one: A device is said to be dependable if 99% of the users never experience any failures during the entire period they own the device. By this definition, virtually no computers are dependable, whereas most TVs, iPods, digital cameras, camcorders, etc. are. Techies are willing to forgive a computer that crashes once or twice a year; ordinary users are not.

Home users aren't the only ones annoyed by the poor dependability of computers. Even in highly technical settings, the low dependability of computers is a problem. Companies like Google and Amazon, with hundreds of thousands of servers, experience many failures every day. They have learned to live with this, but they would really prefer systems that just worked all the time. Unfortunately, current software fails them.

The basic problem is that software contains bugs, and the more software there is, the more bugs there are. Various studies have shown that the number of bugs per thousand lines of code (KLoC) varies from 1 to 10 in large production systems. A really well-written piece of software might have 2 bugs per KLoC over time, but not fewer. An operating system with, say, 4 million lines of code is thus likely to have at least 8000 bugs. Not all are fatal, but some will be. A study at Stanford University showed that device drivers – which make up 70% of the code base of a typical operating system – have bug rates 3x to 7x higher than the rest of the system. Device drivers have higher bug rates because (1) they are more complicated and (2) they are inspected less. While many people study the scheduler, few look at printer drivers.

The Solution: Smaller Kernels

The solution to this problem is to move code out of the kernel, where it can do maximal damage, and put it into user-space processes, where bugs cannot cause system crashes. This is how Minix 3 is designed. The current Minix system is the (second) successor to the original Minix, which was originally launched in 1987 as an educational operating system but has since been radically revised into a highly dependable, self-healing system. What follows is a brief description of the Minix architecture; you can find more at *http://www.minix3.org*.

Minix 3 is designed to run as little code as possible in kernel mode, where bugs can easily be fatal. Instead of 3-4 million lines of kernel code, Minix 3 has about 5000 lines of kernel code. Sometimes kernels this small are called microkernels. They handle low-level process management, scheduling, interrupts, and the clock, and they provide some low-level services to user-space components.

The bulk of the operating system runs as a collection of device drivers and servers, each running as an ordinary user-space process with restricted privileges. None of these drivers and servers run as superuser or equivalent. They cannot even access I/O devices or the MMU hardware directly. They have to use kernel services to read and write to the hardware. The layer of processes running in user-mode directly above the kernel consists of device drivers, with the disk driver, the Ethernet driver, and all the other drivers running as separate processes protected by the MMU hardware so they cannot execute any privileged instructions and cannot read or write any memory except their own.

Above the driver layer comes the server layer, with a file server, a process server, and other servers. The servers make use of the drivers as well as kernel services. For example, to read from a file, a user process sends a message to the file server, which then sends a message to the disk driver to fetch the blocks needed. When the file system has them in its buffer cache, it calls the kernel to move them to the user's address space.

In addition to these servers, there is another server called the reincarnation server. The reincarnation server is the parent of all the driver and server processes and monitors their behavior. If it discovers one process that is not responding to its pings, it starts a fresh copy from disk (except for the disk driver, which is shadowed in RAM). The system has been designed so that many (but not all) of the critical drivers and servers can be replaced automatically, while the system is operating, without disturbing running user processes or even notifying the user. In this way, the system is self healing.

To test whether these ideas work in practice, we ran the following experiment. We started a fault-injection process that overwrote 100 machine instructions in the running binary of the Ethernet driver to see what would happen if one of them were executed. If nothing happened for a few seconds, another 100 were injected, and so on. In all, we injected 800,000 faults into each of three different Ethernet drivers and caused 18,000 driver crashes. In all cases, the driver was replaced automatically by the reincarnation server. Despite injecting 2.4 million faults into the system, not once

3 homepage [2]. Minix is compliant with the POSIX standard IEEE 1003.2-1996, and developers have already ported many Unix programs to Minix.

The Minix Difference

Minix 3 is on the syllabus of many universities, and many generations of students have scrutinized the few thousand lines of Minix code and fixed most errors. The microkernel architecture implements drivers as separate user mode processes that are not permitted to execute privileged commands or I/O operations or write directly to memory. Instead, these operations are performed by auditable kernel calls (see Figure 1).

Minix 3 uses fixed-length messages for process communications. This design simplifies the code structure and helps mitigate the danger of buffer overflows. The Minix filesystem runs as a simple user process. Because it is made up of around 8,200 lines of userspace code, but no kernel code, it is easy to debug.

An innovative component, the reincarnation server, enhances the reliability of the Minix system by serving as the parent of all servers and drivers. It detects crashes very quickly and continually monitors the function of critical processes, re-starting fallen processes as necessary to keep the system running.

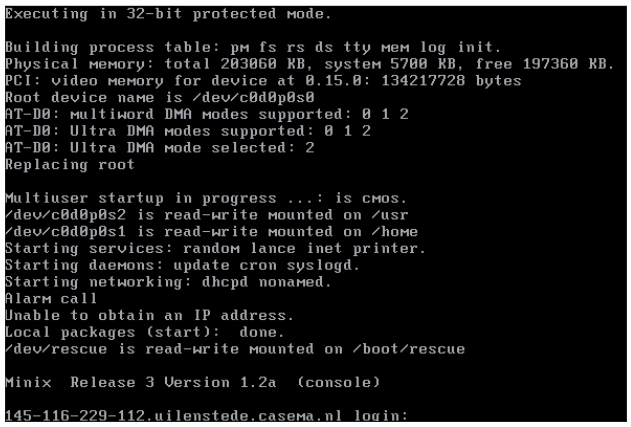
Minix Firewall Project

Packet filters are an endangered system component. Despite the excellent quality of the Linux Netfilter implementation, a number of security issues have surfaced in the past. If a subsystem of this kind is running on the Linux kernel, it will endanger system security. Building on work by the Tanenbaum group, the Technical University of Applied Science Berlin ported the widespread Netfilter framework to Minix 3 [5].

Here again, the stability of the microkernel architecture delivers additional benefits. In Linux, an attacker who succeeds in provoking a crash – for example, by exploiting a buffer overflow in the *do_replace()* function – can bring a Linux firewall to its knees. In Minix 3, a single user process could crash without compromising system security. The reincarnation server would simply restart the process.

The differences become even more apparent if an attacker succeeds in executing code. In Minix, a hijacked user process is still a problem, but the effect is far less serious thanks to isolation.

Even Microsoft is exploring their own microkernel system, named Singularity [6]. Although Minix has played the microkernel game for many years now, its biggest obstacle to becoming more widespread has always been its non-free license. Now that Minix 3 is released under the BSD open source license and the firewall extensions are available under the GPL [7]. Researchers at the TFH Berlin are also working on exploring Minix's potential as a virtualized firewall. Stability, a small footprint, and a new licensing model give Minix 3 a strong potential for growth, especially in embedded systems. ■



```
Executing in 32-bit protected mode.

Building process table: pm fs rs ds tty mem log init.
Physical memory: total 203060 KB, system 5700 KB, free 197360 KB.
PCI: video memory for device at 0.15.0: 134217728 bytes
Root device name is /dev/c0d0p0s0
AT-D0: multiword DMA modes supported: 0 1 2
AT-D0: Ultra DMA modes supported: 0 1 2
AT-D0: Ultra DMA mode selected: 2
Replacing root

Multiuser startup in progress ...: is cmos.
/dev/c0d0p0s2 is read-write mounted on /usr
/dev/c0d0p0s1 is read-write mounted on /home
Starting services: random lance inet printer.
Starting daemons: update cron syslogd.
Starting networking: dhcpd nonamed.
Alarm call
Unable to obtain an IP address.
Local packages (start):  done.
/dev/rescue is read-write mounted on /boot/rescue

Minix  Release 3 Version 1.2a  (console)

145-116-229-112.uilenstede.casema.nl login:
```

Figure 2: Minix is very spartan on launching. This said, version 3.1.2a does include an X11 interface and developer tools.

did the operating system crash. Needless to say, if a fatal error occurs in a Linux or Windows driver running in the kernel, the entire operating system will crash instantly.

Is there a downside to this approach? Yes. There is a performance hit. We have not measured it extensively, but the research group at Karlsruhe [University of Karlsruhe, Germany], which has developed its own microkernel, L4, and then run Linux on top of it as a single user process, has been able to get the performance hit down to 5%. We believe that, if we put some effort into it, we could get the overhead down to the 5–10% range, too. Performance has not been a priority for us, as most users reading their e-mail or looking at Facebook pages are not limited by CPU performance. What they do want, however, is a system that just works all the time.

If Microkernels Are So Dependable, Why Doesn't Anyone Use Them?

Actually, they do. Probably you run many of them. Your mobile phone, for example, is a small but otherwise normal computer, and there is a good chance it runs L4 or Symbian, another microkernel. Cisco's high-performance router uses one. In the military and aerospace markets, where dependability is paramount, Green Hills Integrity, another microkernel, is widely used. PikeOS and QNX are also microkernels widely used in industrial and embedded systems. In other words, when it really matters that the system "just works all the time" people turn to microkernels. For more on this topic, see *www.cs.vu.nl/~ast/reliable-os/*.

In conclusion, it is our belief, based on many conversations with nontechnical users, that what they want above all else is a system that works perfectly all the time. They have a very low tolerance for unreliable systems but currently have no choice. We believe that microkernel-based systems can lead the way to more dependable systems.

INFO

[1] Tanenbaum, Andrew S., "Some Notes on the 'Who wrote Linux' Kerfuffle, Release 1.5," 2004, *http://www.cs.vu.nl/~ast/brown/*

[2] Minix Project: *http://www.minix3.org*

[3] Torvalds, Linus, and David Diamond. *Just for Fun*. HarperBusiness, 2001

[4] Tanenbaum, Andrew S., and Albert S. Woodhull. *The Minix Book: Operating System Design and Implementation*. Prentice-Hall, 2006

[5] Weis, Rüdiger, "Linux is obsolete 2.0," presented at Chaos Communication Camp 2007, *http://public.tfh-berlin.de/~rweis/vorlesungen/ComputerSicherheit/WeisLinuxIsObsolete2.pdf*

[6] Microsoft Singularity: *http://www.codeplex.com/singularity*

[7] Minixwall: *http://wiki.tfh-berlin.de/~minixwall*