DOI:10.1145/2795228

MINIX shows even an operating system can be made to be self-healing.

BY ANDREW S. TANENBAUM

Lessons Learned from 30 Years of MINIX

WHILE LINUX IS well known, its direct ancestor, MINIX, is now 30 and still quite spry for such aged software. Its story and how it and Linux got started is not well known, and there are perhaps some lessons to be learned from MINIX's development. Some of these lessons are specific to operating systems, some to software engineering, and some to other areas (such as project management). Neither MINIX nor Linux was developed in a vacuum. There was quite a bit of relevant history before either got started, so a brief introduction may put this material in perspective.

In 1960, the Massachusetts Institute of Technology, where I later studied, had a room-size vacuum-tube-based scientific computer called the IBM 709. Although a modern Apple iPad is 70,000x faster and has 7,300x more RAM, the IBM 709 was the most powerful computer in the world when introduced. Users wrote programs, generally in FORTRAN, on 80-column punched cards and brought them to the human operator, who read them in. Several hours later the results appeared, printed on 132-column fan-fold paper. A single misplaced comma in a FORTRAN statement could cause a compilation failure, resulting in the programmer wasting hours of time.

To give users better service, MIT developed the Compatible Time-Sharing System (CTSS), which allowed users to work at interactive terminals and reduce the turnaround time from hours to seconds while at the same time using spare cycles to run old-style batch jobs in the background. In 1964, MIT, Bell Labs, and GE (then a computer vendor) partnered to build a successor that could handle hundreds of users all over the Boston area. Think of it as cloud computing V.0.0. It was called MULTiplexed Information and Computing Service, or MULTICS. To make a long and complicated story very short, MULTICS had a troubled youth; the first version required more RAM than the GE 645's entire 288kB memory. Eventually, its PL/1 compiler was improved, and MULTICS booted and ran. Nevertheless, Bell Labs soon tired of the project and pulled out, leaving one of its programmers on the project, Ken Thompson, with a burning desire to reproduce a scaled-down MULTICS on cheap hardware. MULTICS itself was released commercially in 1973 and ran at a number of installations worldwide until the last one was shut down on Oct. 30, 2000, a run of 27 years.

Back at Bell Labs, Thompson found a discarded Digital Equipment Corp. PDP-7 minicomputer and wrote a stripped down version of MULTICS in PDP-7 assembly code. Since it could handle only one user at a time, Thompson's col-

» key insights

- **Each device driver should run as an independent, user-mode process.**
- **Software can last a long time and should be designed accordingly.**
- **It is very difficult to get people to accept new and disruptive ideas.**

MINIX's longtime mascot is a raccoon, chosen because it is agile, smart, usually friendly, and eats bugs.

league Brian Kernighan dubbed it the UNIplexed Information and Computing Service, or UNICS. Despite puns about EUNUCHS being a castrated MULTICS, the name UNICS stuck, but the spelling was later changed to UNIX. It is sometimes now written as Unix since it is not really an acronym anymore.

In 1972, Thompson teamed up with his Bell Labs colleague Dennis Ritchie, who designed the C language and wrote a compiler for it. Together they reimplemented UNIX in C on the PDP-11 minicomputer. UNIX went through several internal versions until Bell Labs decided to license UNIX V6 to universities in 1975 for a $300 fee. Since the PDP-11 was enormously popular, UNIX spread fast worldwide.

In 1977, John Lions of the University of New South Wales in Sydney, Australia, wrote a commentary on the V6 source code, explaining line by line what it meant, a technological version of a line-by-line commentary on the Bible. Hundreds of universities worldwide began teaching UNIX V6 courses using Lions's book as the text.

The lawyers at AT&T, which owned Bell Labs, were aghast that thousands of students were learning all about their product. This had to stop. So the next release, V7 (1979), came equipped with a license that explicitly forbade anyone from writing a book about it or teaching it to students. Operating systems courses went back to theory-only mode or had to use toy simulators, much to the dismay of professors worldwide. The early history of UNIX has been documented in Peter Salus's 1994 book.[14]

MINIX Is Created

There matters rested until 1984, when I decided to rewrite V7 in my spare time while teaching at the Vrije Universiteit (VU) in Amsterdam in order to provide a UNIX-compatible operating system my students could study in a course or on their own. My idea was to write the system, called MIni-uNIX, or MINIX, for the new IBM PC, which was cheap enough (starting at $1,565) a student could own one. Because early PCs did not have a hard disk, I designed MINIX to be V7 compatible yet run on an IBM PC with 256kB RAM and a single 360kB 5¼-inch floppy disk—a far smaller configuration than the PDP-11 V7 ran on. Although the system was supposed to run on this configuration (and did), I realized from the start that to actually compile and build the whole system on a PC, I would need a larger system, namely one with the maximum possible RAM (640kB) and two 360kB 5¼-inch floppy disks.

My design goals for MINIX were as follows:

▸ Build a V7 clone that ran on an IBM PC with only a single 360kB floppy disk;

▸ Build and maintain the system using itself, or "self-hosting";

▸ Make the full source code available to everyone;

▸ Have a clean design students could easily understand;

▸ Make the (micro) kernel as small as possible, since kernel failures are fatal;

▸ Break the rest of the operating system into independent user-mode processes;

▸ Hide interrupts at a very low level;

▸ Communicate only by synchronous message passing with clear protocols; and

▸ Try to make the system port easily to future hardware.

Initially, I did software development on my home IBM PC running Mark Williams Coherent, a V7 clone written by alumni of the University of Waterloo. Its source code was not publicly available. Using Coherent was initially necessary because at first I did not have a C compiler. When my programmer, Ceriel Jacobs, was able to port a C compiler based on the Amsterdam Compiler Kit,[18] written at the VU as part of my research, the system became self-hosting. Because I was now using MINIX to compile and build MINIX, I was extremely sensitive to any bugs or flaws that turned up. All developers should try to use their own systems as early as feasible so they can see what users will experience.

Lesson. Eat your own dog food.

The microkernel was indeed small. Only the scheduler, low-level process management, interprocess communication, and the device drivers were in it. Although the device drivers were compiled into the microkernel's executable program, they were actually scheduled independently as normal processes. This was a compromise because I felt having to do a full address space switch to run a device driver would be too painful on a 4.77MHz 8088, the CPU in the IBM PC. The microkernel was compiled as a standalone executable program. Each of the other operating system components, including the file system and memory manager, was compiled as a separate program and run as a separate process. Because the 8088 did not have a memory management unit (MMU), I could have taken shortcuts and put everything into one executable but decided against it because I wanted the design to work on future CPUs with an MMU.

Be careful what you put out on the Internet; it might come back to haunt you decades later.

It took me approximately two years to get it running, working on it only evenings and weekends. After the system was basically working, it tended to crash after an hour of operation for no reason at all and in no discernible pattern. Debugging the operating system on the bare metal was well nigh impossible and I came within a hair of abandoning the project.

I then made one final effort. I wrote an 8088 simulator on which to run MINIX, so when it crashed I could get a proper dump and stack trace. To my horror, MINIX would run flawlessly for days, even weeks, at a time on the simulator. It never once crashed. I was totally flummoxed. I mentioned this peculiar situation of MINIX running on the simulator but not on the hardware to my student, Robbert van Renesse, who said he heard somewhere that the 8088 generated interrupt 15 when it got hot. I told him there was nothing in the 8088 documentation about that, but he insisted he heard it somewhere. So I inserted code to catch interrupt 15. Within an hour I saw this message on the screen: "Hi. I am interrupt 15. You will never see this message." I immediately made the required patch to catch interrupt 15. After that MINIX worked flawlessly and was ready for release.

Lesson. Do not trust documentation blindly; it could be wrong.

Thirty years later the consequences of Van Renesse's offhand remark are enormous. If he had not mentioned interrupt 15, I would probably have eventually given up in despair. Without MINIX, it is inconceivable there would have been a Linux since Linus Torvalds learned about operating systems by studying the MINIX source code in minute detail and using it as a base to write Linux. Without Linux, there would not have been an Android since it is built on top of Linux. Without Android, the relative stock prices of Apple and Samsung might be quite different today.

Lesson. Listen to your students; they may know more than you.

I wrote most of the basic utilities myself. MINIX 1.1 included 60 of them, from `ar` to `wc`. A typical one was approximately 4kB. A boot loader today can be 100x bigger. All of MINIX, including the binaries and sources, fit nicely on eight 360kB floppy disks. Four of them were the boot disk, the root file system, `/usr`, and `/user` (see Figure 1). The other four contained the full operating system sources and the sources to the 60 utilities. Only the compiler source was left out, as it was quite large.

Lesson. Nathan Myhrvold's Law is true: Software is a gas. It expands to fill its container.

With some discipline, developers can try to break this "law" but have to try really hard. The default is "more bloat."

Figuring out how to distribute the code was a big problem. In those days (1987) almost nobody had a proper Internet connection (though newsgroups on USENET via the UUCP program and email existed at some universities). I decided to write a book[15] describing the code, like Lions did before me, and have my publisher, Prentice Hall, distribute the system, including all source code, as an adjunct to the book. After some negotiation, Prentice Hall agreed to sell a nicely packaged box containing eight 5¼-inch floppy disks and a 500-page manual for $69. This was essentially the manufacturing cost. Prentice Hall had no understanding of what software was but saw selling the software at cost as a way to sell more books. When high-capacity 1.44MB 3½-inch floppies became available later, I also made a version using them.

Lesson. No matter how desirable your product is, you need a way to market or distribute it.

Within a few days of its release, a USENET newsgroup, comp.os.minix, was started. Before a month had gone by, it had 40,000 readers, a huge number considering how few people even had access to USENET. MINIX became an instant cult item.

I soon received an email message from Dan Doernberg, co-founder of the now-defunct Computer Literacy bookstore in Silicon Valley inviting me to speak about MINIX if I was ever there. As it turned out, I was going to the Bay Area in a few weeks to attend a conference, so I accepted. I was expecting him to set up a table and chair in his store for me to sign books. Little did I know he would rent the main auditorium at the Santa Clara Convention Center and do enough publicity to nearly fill it. After my talk, the questions went on until close to midnight.

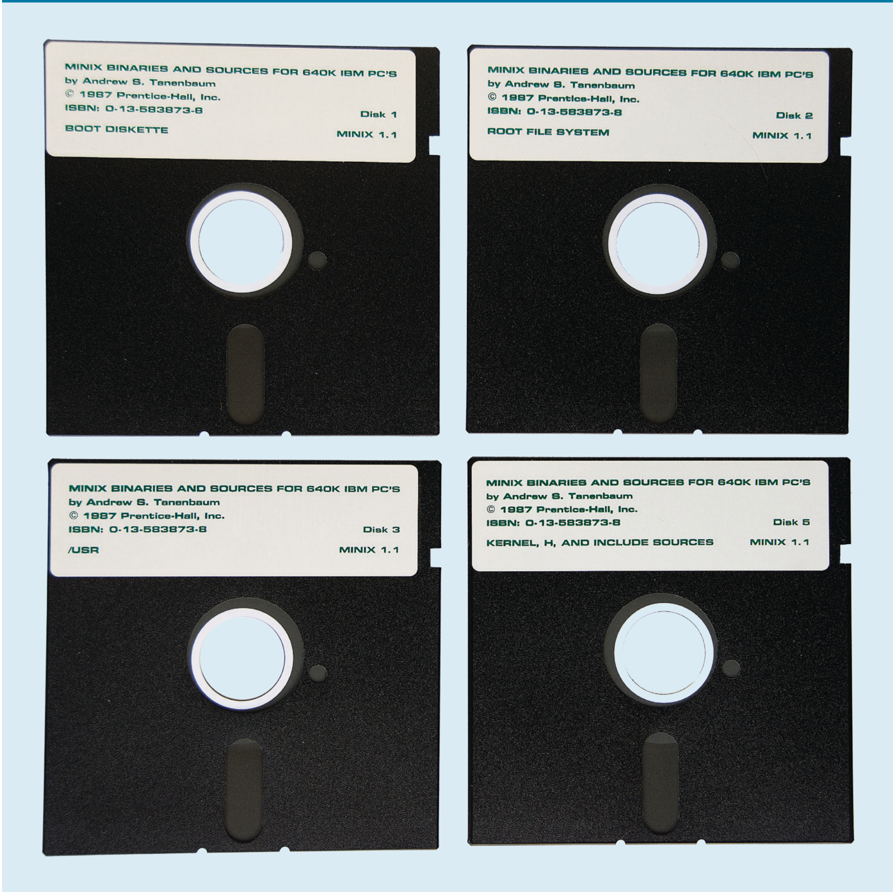
I began getting hundreds of email messages asking for (no, demanding) this feature or that feature. I resisted some (but not all) demands because I was concerned about the possibility the system would become so big it would require expensive hardware students could not afford, and many people, including me, expected either GNU/Hurd or Berkeley Software Distribution (BSD) to take over the niche of full-blown open-source production system, so I kept my focus on education.

People also began contributing software, some very useful. One of the many contributors was Jan-Mark Wams, who wrote a hugely useful test suite that helped debug the system. He also wrote a new compression program that was better than all the existing ones at the time. This reduced the number of floppy disks in the distribution by two disks. Even when the distribution later went online this was important because not many people had a super-speed 56kbps modem.

Lesson. Size matters.

In 1985, Intel released its 386 processor with a full protected-mode 32-bit architecture. With the help of many users, notably Bruce Evans of Australia, I was able to release a 32-bit protected mode version of MINIX. Since I was always thinking about future hardware, from day 1, the code clearly distinguished what code ran in "kernel mode" and what code ran as separate processes in "user mode," even though the 8088 had only one mode. This helped a lot when these modes finally appeared in the 386. Also, the original code clearly distinguished virtual addresses from physical addresses, which did not matter on the 8088 but did matter (a lot) on the 386, making porting to it much easier. Also around this time two people at the VU, Kees Bot and Philip Homburg, produced an excellent 32-bit version with virtual memory, but I

Figure 1. Four of the original 5¼-inch MINIX 1 floppy disks.


MINIX BINARIES AND SOURCES FOR 640K IBM PC'S
by Andrew S. Tanenbaum
© 1987 Prentice-Hall, Inc.
ISBN: 0-13-583873-8
Disk 1
BOOT DISKETTE
MINIX 1.1
MINIX BINARIES AND SOURCES FOR 640K IBM PC'S
by Andrew S. Tanenbaum
© 1987 Prentice-Hall, Inc.
ISBN: 0-13-583873-8
Disk 2
ROOT FILE SYSTEM
MINIX 1.1
MINIX BINARIES AND SOURCES FOR 640K IBM PC'S
by Andrew S. Tanenbaum
© 1987 Prentice-Hall, Inc.
ISBN: 0-13-583873-8
Disk 3
/USR
MINIX 1.1
MINIX BINARIES AND SOURCES FOR 640K IBM PC'S
by Andrew S. Tanenbaum
© 1987 Prentice-Hall, Inc.
ISBN: 0-13-583873-8
Disk 5
KERNEL, H, AND INCLUDE SOURCES
MINIX 1.1

decided to stick with Evans's work since it was closer to the original design.

Lesson, Try to make your design be appropriate for hardware likely to appear in the future.

By 1991, MINIX 1.5, had been ported to the Apple Macintosh, Amiga, Atari, and Sun SPARCstation, among other platforms (see Figure 2).

Lesson. By not relying on idiosyncratic features of the hardware, one makes porting to new platforms much easier.

As the system developed, problems cropped up in unexpected places. A particularly annoying one involved a network card driver that could not be debugged. Someone eventually discovered the card did not honor its own specifications.

Lesson. As with software, hardware can contain bugs.

A hardware "feature" can sometimes be viewed as a hardware bug. The port of MINIX to a PC clone made by Olivetti, a major Italian computer manufacturer at the time, was causing problems until I realized, for inexplicable reasons, a handful of keys on the Olivetti keyboard returned different scan codes from those returned by genuine IBM keyboards. This led me to realize that many countries have their own standardized keyboards, so I changed MINIX to support multiple keyboards, selectable when the system is installed. This is useful for people with Italian, French, German, and other national keyboards. So, my initial annoyance at Olivetti was tempered when I saw a way to make MINIX better for people in countries other than the U.S. Likewise, in several future cases, what were initially seen as bugs motivated me to generalize the system to improve it.

Lesson. When someone hands you a lemon, make lemonade.

Linus Torvalds Buys a PC

On January 5, 1991, Linus Torvalds, a hitherto-unknown Finnish student at the University of Helsinki, made a critical decision. He bought a fast (33MHz) large (4MB RAM, 40MB hard disk) PC largely for the purpose of running MINIX and studying it. On March 29, 1991, Torvalds posted his first message to the USENET newsgroup, comp.os.minix:

"Hello everybody, I've had minix for a week now, and have upgraded to 386-minix (nice), and duly downloaded `gcc` for minix ... "

His second posting to comp.os.minix was on April 1, 1991, in response to a simple question from someone else:

"RTFSC (Read the F***ing Source Code :-)—It is heavily commented and the solution should be obvious ... "

This posting shows that in 10 days, Torvalds had studied the MINIX source code well enough to be somewhat disdainful of people who had not studied it as well as he had. The goal of MINIX at the time was, of course, to be easy for students to learn; in Torvalds' case, it was wildly successful.

Then on August 25, 1991, Torvalds made another post to comp.os.minix:

"Hello everybody out there using minix—I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since April, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things)."

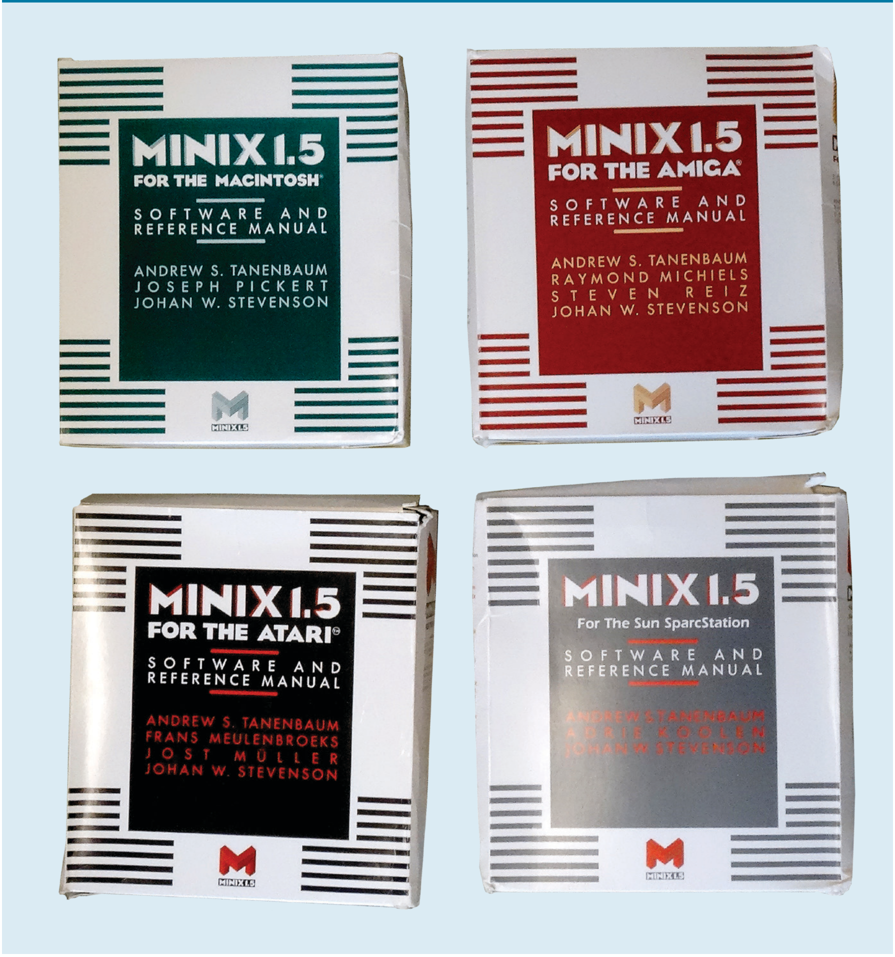
During the next year, Torvalds continued studying MINIX and using it to develop his new system. This became the first version of the Linux kernel. Fossilized remains of its connection to MINIX were later visible to software archaeologists in things like the Linux kernel using the MINIX file system and source-tree layout.

On January 29, 1992, I posted a message to comp.os.minix saying microkernels were better than monolithic designs, except for performance. This posting unleashed a flamewar that still, even today, 24 years later, inspires many students worldwide to write and tell me their position on this "debate."

Lesson. The Internet is like an elephant; it never forgets.

That is, be careful what you put out on the Internet; it might come back to haunt you decades later.

Figure 2. MINIX 1.5 for four different platforms.



It turns out performance is more important to some people than I had expected. Windows NT was designed as a microkernel, but Microsoft later switched to a hybrid design when the performance was not good enough. In NT, as well as in Windows 2000, XP, 7, 8, and 10, there is a hardware abstraction layer at the very bottom (to hide differences between motherboards). Above it is a microkernel for handling interrupts, thread scheduling, low-level interprocess communication, and thread synchronization. Above the microkernel is the Windows Executive, a group of separate components for process management, memory management, I/O management, security, and more that together comprise the core of the operating system. They communicate through well-defined protocols, just like on MINIX, except on MINIX they are user processes. NT (and its successors) were something of a hybrid because all these parts ran in kernel mode for performance reasons, meaning fewer context switches. So, from a software engineering standpoint, it was a microkernel design, but from a reliability standpoint, it was monolithic, because a single bug in any component could crash the whole system. Apple's OS X has a similar hybrid design, with the bottom layer being the Mach 3.0 microkernel and the upper layer (Darwin) derived from FreeBSD, a descendant of the BSD system developed at the University of California at Berkeley.

Also worth noting is in the world of embedded computing, where reliability often trumps performance, microkernels dominate. QNX, a commercial UNIX-like real-time operating system, is widely used in automobiles, factory automation, power plants, and medical equipment. The L4 microkernel[11] runs on the radio chip inside more than one billion cellphones worldwide and also on the security processor inside recent iOS devices like the iPhone 6. L4 is so small, a version of it consisting of approximately 9,000 lines of C was formally proven correct against its specification,[9] something unthinkable for multimillion-line monolithic systems. Nevertheless, microkernels remain controversial for historical reasons and to some extent due to somewhat lower performance.[16]

What was new about MINIX research was the attempt to build a fault-tolerant multi-server POSIX-compliant operating system on top of the microkernel.

On the newsgroup comp.os.minix in 1992 I also made the point that tying Linux tightly to the 386 architecture was not a good idea because RISC machines would eventually dominate the market. To a considerable extent this is happening, with more than 50 billion (RISC) ARM chips shipped. Most smartphones and tablets use an ARM CPU, including variants like Qualcomm's Snapdragon, Apple's A8, and Samsung's Exynos. Furthermore, 64-bit ARM servers and notebooks are beginning to appear. Linux was eventually ported to the ARM, but it would have been much easier had it not been tied so closely to the x86 architecture from the start.

Lesson. Do not assume today's hardware will be dominant forever.

Also in this vein, Linux is so tightly tied to the `gcc` compiler that compiling it with newer (arguably, better) compilers like clang/LLVM requires major patches to the Linux code.

Lesson. When standards exist (such as ANSI Standard C) stick to them.

In addition to the real start of Linux, another major development occurred in 1992. AT&T sued BSDI (a company created by the developers of Berkeley UNIX to sell and support the BSD software) and the University of California. AT&T claimed BSD contained pieces of AT&T code and also BSDI's telephone number, 1-800-ITS-UNIX, violated AT&T's intellectual property rights. The case was settled out of court in 1994, until which time BSD was handcuffed, giving the new Linux system critical time to develop. If AT&T had been more sensible and just bought BSDI as its marketing arm, Linux might never have caught on against such a mature and stable competitor with a very large installed base.

Lesson. If you are running one of the biggest corporations in the world and a tiny startup appears in an area you care about but know almost nothing about, ask the owners how much they want for the company and write them a check.

In 1997, MINIX 2, now changed to be POSIX-compatible rather than UNIX V7-compatible, was released, along with a second edition of my book *Operating Systems Design and Implementation*, now co-authored with Albert Woodhull, a professor at Hampshire College in Massachusetts.

In 2000, I finally convinced Prentice

Hall to release MINIX 2 under the BSD license and make it (including all source code) freely available on the Internet. I should have tried to do this much earlier, especially since the original license allowed unlimited copying at universities, and it was being sold at essentially the publisher's cost price anyway.

Lesson. Even after you have adopted a strategy, you should nevertheless reexamine it from time to time.

MINIX as Research Project

MINIX 2 continued to develop slowly for a few more years, but the direction changed sharply in 2004 when I received a grant from the Netherlands Organisation for Scientific Research (http://www.nwo.nl) to turn what had been an educational hobby into a serious, funded research project on building a highly reliable system; until 2004, there was no external funding. Shortly thereafter, I received an Academy Professorship from the Royal Netherlands Academy of Arts and Sciences in Amsterdam. Together, these grants provided almost $3 million for research into reliable operating systems based on MINIX.

Lesson. Working on something important can get you research funding, even if it is outside the mainstream.

MINIX was not, of course, the only research project looking at microkernels. Early systems from as far back as 1970 included Amoeba,[17] Chorus,[12] L3,[10] L4,[11] Mach,[1] RC 4000 Nucleus,[3] and V.[4] What was new about MINIX research was the attempt to build a fault-tolerant multiserver POSIX-compliant operating system on top of the microkernel.

Together with my students and programmers in 2004, I began to develop MINIX 3. Our first step was to move the device drivers entirely out of the microkernel. In the MINIX 1 and MINIX 2 designs, device drivers were treated and scheduled as independent processes but lived in the microkernel's (virtual) address space. My student Jorrit Herder's master's thesis consisted of making each driver a full-blown user-mode process. This change made MINIX far more reliable and robust. During his subsequent Ph.D. research at the VU under my supervision, Herder showed failed drivers could be replaced on the fly, while the system was running, with no adverse effects at all.[7] Even a failed disk driver could be replaced on the fly, since a copy was always kept in RAM; the other drivers could always be fetched from disk. This was a first step toward a self-healing system. The fact that MINIX could now do something—replace (some) key operating system components that had crashed without rebooting and without running application processes even noticing it—no other system could do this, which gave my group confidence we were really onto something.

Lesson. Try for an early success of some kind; it builds up everyone's morale.

This change made it possible to implement the Principle of Least Authority, also called Principle of Least Privilege,[13] much better. To touch device registers, even for its own device, a driver now had to make a call to the microkernel, which could check if that driver had permission to access the device, greatly improving robustness. In a monolithic system like Windows or Linux, a rogue or malfunctioning audio driver has the power to erase the disk; in MINIX, the microkernel will not let it. If an I/O memory-management unit is present, mediation by the microkernel is not needed to achieve the same effect.

In addition, components could communicate with other components only if the microkernel approved, and components could make only approved microkernel calls, all of this controlled by tables and bitmaps within the microkernel. This new design with tighter restrictions on the operating system components (and other improvements) was called MINIX 3 and coincided with the third edition of my and Woodhull's book *Operating Systems Design and Implementation, Third Edition*.

Lesson. Each device driver should run as an unprivileged, independent user-mode process.

Microsoft clearly understood and still understands this and introduced the User-Mode Driver Framework for Windows XP and later systems, intending to encourage device-driver writers to make their drivers run as user-mode processes, just as in MINIX.

In 2005, I was invited to be the keynote speaker at ACM's Symposium on Operating System Principles (http://www.sosp.org), the top venue for operating systems research. It was held in October at the Grand Hotel in Brighton, U.K., that year. I decided in my talk I would formally announce MINIX 3 to the assembled operating system experts. Partway through my talk I removed my dress shirt on stage to reveal a MINIX 3 T-shirt. The MINIX website was set up to allow downloading starting that day. Needless to say, I wanted to be online during the conference to see if the server could handle the load. Since I was the honored guest of the conference, I was put in the Royal Suite, where the Queen of England would stay should she choose to visit Brighton. It is a massive room, with a magnificent view of the sea. Unfortunately, it was the only room in the hotel lacking an Internet connection, since apparently the Queen is not a big Internet user. To make it worse, the hotel did not have Wi-Fi. Fortunately, one of the conference organizers took pity on me and was willing to swap rooms so I could have a standard room but with that oh-so-important Ethernet port.

Lesson. Keep focused on your real goal.

That is, do not be distracted when something seemingly nice (like a beautiful hotel room) pops up but is actually a hindrance.

By 2005, MINIX 3 was a much more serious system, but so many people had read the *Operating Systems Design and Implementation* book and studied MINIX in college it was very difficult to convince anyone it was not a toy system anymore. So I had the irony of a very well-known system but had to struggle to get people to take it seriously due to its history. Microsoft was smarter; early versions of Windows, including Windows 95 and Windows 98, were just MS-DOS with a graphical shell. But if they had been marketed as "Graphical MS-DOS" Microsoft might not have done as well as renaming them "Windows," which Microsoft indeed did.

Lesson. If V3 of your product differs from V2 in a really major way, give it a totally new name.

In 2008, the MINIX project received another piece of good luck. For some years, the European Union had been toying with the idea of revising product liability laws to apply to software. If one in 10 million tires explode, killing people, the manufacturer cannot get off the hook by saying, "Tire explosions happen." With software, that argument works. Since a country or other jurisdiction cannot legislate something that is technically impossible, the European

Research Council, which is funded by the E.U., decided to give me a European Research Council Advanced Grant of roughly $3.5 million to see if I could make a highly reliable, self-healing operating system based on MINIX.

While I was enormously grateful for the opportunity, this immense good fortune also created a major problem. I was able to hire four expert professional programmers to develop "MINIX 3, the product" while also funding six Ph.D. students and several postdocs to push the envelope on research. Before long, each Ph.D. student had copied the MINIX 3 source tree and began modifying it in major ways to use in his research. Meanwhile, the programmers were busy improving and "productizing" the code. After two or three years, we were unable to put Humpty Dumpty back together again. The carefully developed prototype and the students' versions had diverged so much we could not put their changes back in, despite our using `git` and other state-of-the-art tools. The versions were simply too incompatible. For example, if two people completely rewrite the scheduler using totally different algorithms, they cannot be automatically merged later.

Also, despite my stated desire to put the results of the research into the product, the programmers strongly resisted, since they had been extremely meticulous about their code and were not enthusiastic (to put it mildly) about injecting a lot of barely tested student-quality code into what had become a well-tested production system. Only with a lot of effort would my group possibly succeed with getting one of the research results into the product. But we did publish a lot of papers; see, for example Appuswamy et al.,[2] Giuffrida et al.,[5] Giuffrida et al.,[6] and Hruby et al.[8]

Lesson. Doing Ph.D. research and developing a software product at the same time are very difficult to combine.

Sometimes both researchers and programmers would run into the same problem. One such problem involved the use of synchronous communication. Synchronous communication was there from the start and is very simple. It also conflicts with the goal of reliability. If a client process, C, sends a message to a server process, S, and C crashes or gets stuck in an infinite loop without listening for the response, the server hangs because it is unable to send its reply. This problem is inherent in synchronous communication. To avoid it, we were forced to introduce virtual endpoints, asynchronous communication, and other things far less elegant than the original design.

Lesson. Einstein was right: Things should be as simple as possible but not simpler.

What Einstein meant is everyone should strive for simplicity and make sure their solution is comprehensive enough to do the job but no more. This has been a guiding principle for MINIX from the start. It is unfortunately absent in far too much modern bloated software.

Around 2011, the direction we were going to take with the product began to come into better focus, and we made two important decisions. First, we came to realize that to get anyone to use the system it had to have applications, so we adopted the headers, libraries, package manager, and a lot more from BSD (specifically, NetBSD). In effect, we had reimplemented the NetBSD user environment on a much more fault-tolerant substructure. The big gain here was 6,000 NetBSD packages were suddenly available.

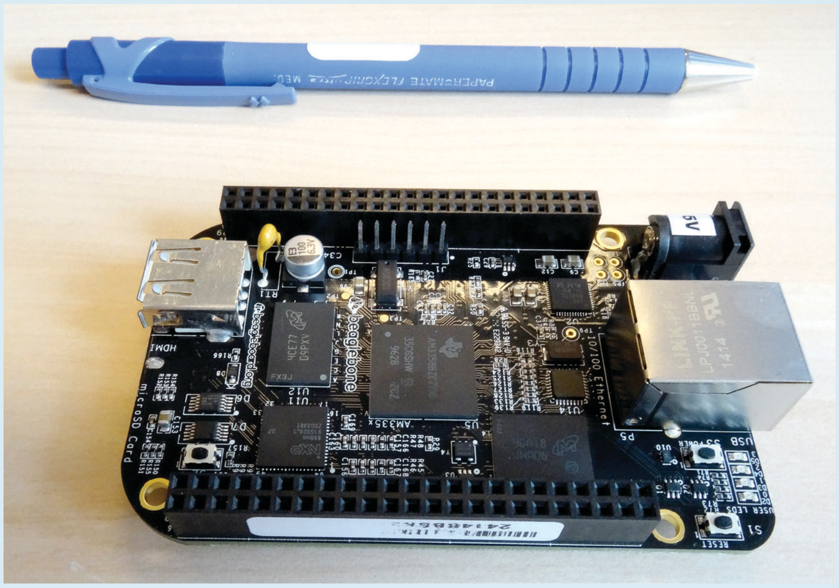
Lesson. If you want people to use your product, it has to do something useful.

Second, we realized winning the desktop war against Windows, Linux, OS X, and half a dozen BSDs was a tall order, although MINIX 3 could well be used in universities as a nice base for research on fault-tolerant computing. So we ported MINIX 3 to the ARM processor and began to focus on embedded systems, where high reliability is often crucial. Also, when engineers are looking for an operating system to embed in a new camera, television set, digital video recorder, router, or other product, they do not have to contend with millions of screaming users who demand the product be backward compatible to 1981 and run all their MS-DOS games as fast as their previous product did. All the users see is the outside, not the inside. In particular, we got MINIX 3 running on the BeagleBone series of single-board computers that use the ARM Cortex-A8 processor (see Figure 3). These boards are essentially complete PCs and retail for about $50. They are often used to prototype embedded systems. All of them are open source hardware, which made figuring out how they work easy.

Lesson. If marketing the product according to plan A does not work, invent plan B.

Retrospective. With 20-20 hindsight, some things stand out now. First, the idea of a small microkernel with user-level system components protected from each other by the hardware MMU is probably still the best way to aim for

Figure 3. A BeagleBone Black Board.



highly reliable, self-healing systems because this design keeps problems in one component from spreading to others. It is perhaps surprising that in 30 years, almost no code was moved into the MINIX microkernel. In fact, some major software components, including all the drivers and much of the scheduler, were moved out of it. The world is also moving (slowly) in this direction (such as Windows User-mode drivers and embedded systems). Nevertheless, having most of the operating system run as user-mode processes is disruptive, and it takes time for disruptive ideas to take hold; for example, FORTRAN, Windows XP, mainframes, QWERTY keyboards, the x86 architecture, fax machines, magnetic-stripe credit cards, and the interlaced NTSC color television standard made sense when they were invented but not so much anymore. However, they are not about to exit gracefully. For example, according to Microsoft, as of March 2016, the obsolete Windows XP still runs on 250 million computers.

Lesson. It is very difficult to change entrenched ways of doing things.

Furthermore, in due course, computers will have so much computing power, efficiency will not matter so much. For example, Android is written in Java, which is far slower than C, but nobody seems to care.

My initial decision back in 1984 to have fixed-size messages throughout the system and avoid dynamic memory allocation (such as `malloc`) and a heap in the kernel has not been a problem and avoids problems that occur with dynamic storage management (such as memory leaks and buffer overruns).

Another thing that worked well in MINIX is the event-driven model. Each driver and server has a loop consisting of

```
{ get_request();
  process_request();
  send_reply();
}
```

This design makes them easy to test and debug in isolation.

On the other hand, the simplicity of MINIX 1 limited its usability. Lack of features like kernel multithreading and full-demand paging were not a realistic option on a 256kB IBM PC with one floppy disk. We could have added them (and all their complexity) at some point, but we did not (although we have some workarounds) and are paying a price today, as porting some software is more difficult than it would otherwise be.

Although funding has now ended, the MINIX project is not ending. It is instead transitioning to an open source project, like so many others. Various improvements are in progress now, including some very interesting ones (such as being able to update nearly all of the operating system drivers, file system, memory manager, and process manager) on the fly to major new versions (potentially with different data structures) while the system is running.[5,6] These updates require no down time and have no effect on running processes, except for the system freezing very briefly before continuing. The structure of the system as a collection of servers makes live update much simpler than in traditional designs, since it is possible to do a live update on, say, the memory manager, without affecting the other (isolated) components because they are in different address spaces. In systems that pass pointers between subsystems within the kernel, live updating one piece without updating all of them is very difficult. This area is one of the few where the research may make it into the product, but it is an important one that few, if any, other systems have.

MINIX 3 can be downloaded for free at http://www.minix3.org.

Acknowledgments

I would like to thank the hundreds of people who have contributed to the MINIX project over the course of 30 years. Unfortunately there are far too many to name here. Nevertheless, some stand out for a special mention: Kees Bot, Ben Gras, Philip Homburg, Kees Jongenburger, Lionel Sambuc, Arun Thomas, Thomas Veerman, and Jan-Mark Wams. This work was supported in part by the Netherlands Organisation for Scientific Research, as well as by European Research Council Advanced Grant 227874 and ERC Proof of Concept Grant 297420.

References

1. Accetta, M., Baron, R., Golub, D., Rashid, R., Tevian, A., and Young, M. Mach 1986: A new kernel foundation for Unix development. In *Proceedings of the USENIX Summer Conference* (Atlanta, GA, June 9–13). USENIX Association, Berkeley, CA, 1986, 93–112.
2. Appuswamy, R., van Moolenbroek, D.C., and Tanenbaum, A.S. Loris: A dependable, modular file-based storage stack. In *Proceedings of the 16th Pacific Rim International Symposium of Dependable Computing* (Tokyo, Dec. 13–15). IEEE Computer Society, Washington, D.C., 2010, 165–174.
3. Brinch Hansen, P. The nucleus of a multiprogramming system. *Commun. ACM 13*, 4 (Apr. 1970), 238–241.
4. Cheriton, D.R. The V kernel, a software base for distributed systems. *IEEE Software 1*, 4 (Apr. 1984), 19–42.
5. Giuffrida, C., Iorgulescu, C., Kuijsten, A., and Tanenbaum, A.S. Back to the future: Fault-tolerant live update with time-traveling state transfer. In *Proceedings of the 27th Large Installation System Administration Conference* (Washington D.C., Nov. 3–8). USENIX Association, Berkeley, CA, 2013, 89–104.
6. Giuffrida, C., Kuijsten, A., and Tanenbaum, A.S. Safe and automatic live update for operating systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, TX, Mar. 16–20). ACM Press, New York, 2013, 279–292.
7. Herder, J. *Building a Dependable Operating System, Fault Tolerance in MINIX 3.* Ph.D. Thesis, Vrije Universiteit, Amsterdam, the Netherlands, 2010; http://www.cs.vu.nl/~ast/Theses/herder-thesis.pdf
8. Hruby, T., Bos, H., and Tanenbaum, A.S. When slower is faster: On heterogeneous multicores for reliable systems. In *Proceedings of the Annual Technical Conference* (San Jose, CA, June 26–28). USENIX Association, Berkeley, CA, 2013, 255–266.
9. Klein G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Swell, T., Tuch, H., and Winwood, S. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd Symposium on Operating Systems Principles* (Big Sky, MT, Oct. 11–14). ACM Press, New York, 2009, 207–220.
10. Liedtke, J. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (Asheville, NC, Dec. 5–8). ACM Press, New York, 1993, 174–188.
11. Liedtke, J. On microkernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain Resort, CO, Dec. 3–6). ACM Press, New York, 1995, 237–250.
12. Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M. Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Leonard, P., and Neuhauser, W. The CHORUS distributed operating system. *Computing Systems Journal 1*, 4 (Dec. 1988), 305–370.
13. Saltzer, J.H. and Schroeder, M.D. The protection of information in computer systems. *Proceedings of the IEEE 63*, 9 (Sept. 1975), 1278–1308.
14. Salus, P.H. *A Quarter Century of UNIX.* Addison-Wesley, Reading, MA, 1994.
15. Tanenbaum, A.S. *Operating Systems Design and Implementation, First Edition.* Prentice Hall, Upper Saddle River, NJ, 1987.
16. Tanenbaum, A.S., Herder, J., and Bos, H.J. Can we make operating systems reliable and secure? *Computer 39*, 5 (May 2006), 44–51.
17. Tanenbaum, A.S. and Mullender, S.J. A capability-based distributed operating system. In *Proceedings of the Conference on Local Networks & Distributed Office Systems* (London, U.K., May 1981), 363–377.
18. Tanenbaum, A.S, van Staveren, H., Keizer, E.G., and Stevenson, J.W. A practical toolkit for making portable compilers. *Commun. ACM 26*, 9 (Sept. 1983), 654–660.

Andrew S. Tanenbaum (ast@cs.vu.nl) is a professor emeritus of computer science in the Department of Computer Science in the Faculty of Sciences at the Vrije Universiteit, Amsterdam, the Netherlands and an ACM Fellow.

Watch the author discuss his work in this exclusive *Communications* video. http://cacm.acm.org/videos/lessons-learned-from-30-years-of-minix