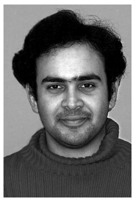
ANDREW TANENBAUM, RAJA APPUSWAMY, HERBERT BOS, LORENZO CAVALLARO, CRISTIANO GIUFFRIDA, TOMÁŠ HRUBÝ, JORRIT HERDER, ERIK VAN DER KOUWE, AND DAVID VAN MOOLENBROEK

MINIX 3: status report and current research

Despite the fact that Andrew Tanenbaum has been producing open source code for 30 years as a professor at the Vrije Universiteit, somehow he found the time to (co)author 18 books and 150 papers and become a Fellow of the ACM and of the IEEE. He was awarded the USENIX Flame Award in 2008. He believes that computers should be like TV sets: you plug them in and they work perfectly for the next 10 years.

ast@cs.vu.nl



Raja Appuswamy is a PhD student at Vrije Universiteit. His research interests include file and storage systems, and operating system reliability. He received his BE from Anna Univeristy, India, and his MS from the University of Florida, Gainesville.

rappusw@few.vu.nl

Herbert Bos obtained his MSc from the University of Twente in the Netherlands and his PhD from the Cambridge University Computer Laboratory (UK). He is currently an associate professor at the Vrije Universiteit in Amsterdam, with a keen research interest in operating systems, high-speed networks, and security.

herbertb@cs.vu.nl

Lorenzo is a post-doctorate researcher at Vrije Universiteit Amsterdam, where he joined Prof. Tanenbaum and his team working on systems dependability and security. Lorenzo's passion for systems security was further inspired by work at UC Santa Barbara and Stony Brook University. Lorenzo received an MSc and a PhD in computer science from the University of Milan, Italy.

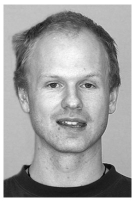
l.cavallaro@few.vu.nl

Cristiano Giuffrida is a PhD student at Vrije Universiteit, Amsterdam. His research interests include self-healing systems and secure and reliable operating systems. He received his BE and ME from University of Rome "Tor Vergata," Italy.

c.giuffrida@few.vu.nl

Jorrit Herder holds an MSc degree in computer science (cum laude) from the Vrije Universiteit in Amsterdam and will get his PhD there in Sept. 2010. His research focused on operating system reliability and security, and he was closely involved in the design and implementation of MINIX 3. He is now at Google in Sydney.

jnherder@gmail.com

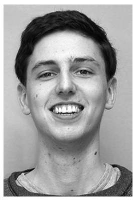


Tomáš Hrubý has master's degrees from both the Charles University in Prague and the Vrije Universiteit. After graduating, he decided to go down under and spent some time at the University of Otago in New Zealand and NICTA in Australia. He is currently a PhD student at the Vrije Universiteit, working on how to match multiserver operating systems to multicore chips.

thruby@few.vu.nl

Erik van der Kouwe got his master's degree at the Vrije Universiteit and is now a PhD student in computer science there. He works on virtualization and legacy driver support.

vdkouwe@cs.vu.nl



David van Moolenbroek has an MSc degree in computer science from the Vrije Universiteit in Amsterdam, and is currently working as a PhD student there. His research interests include file and storage systems and operating system reliability.

dcvmoole@few.vu.nl

MOST PEOPLE WANT THEIR COMPUTER to be like their TV set: you buy it, plug it in, and it works perfectly for the next 10 years. Suffice it to say that current computers—and especially their operating systems—are not even close. We will consider the job done when the average user has never experienced a system crash in his or her lifetime and no computer has a RESET button. In the MINIX project, we are trying to get closer to that goal by improving the reliability, availability, and security of operating systems.

What started in 1987 as MINIX 1, a tool to teach students about operating systems, has become MINIX 3, a more mature operating system whose internal structure promotes high availability while preserving the well-established POSIX interface to application programs and users. Although the name has been kept, the two systems are very different, just as Windows 3 and Windows 7 are both called Windows but are also very different. In this article, we will briefly describe the architecture of MINIX 3 and what it is like now—as an update to the February 2007 *;login:* article [1]—and the work currently in progress to develop it further.

The impetus for much of this work was a grant to one of us (Tanenbaum) from the Netherlands Royal Academy of Arts and Sciences for 1 million euros to develop a highly reliable operating system, followed four years later by a 2.5 million euro grant from the European Research Council to continue this work. This funding has primarily supported PhD students, postdocs, and a couple of programmers to work on the project, which has led to a series of releases, of which 3.1.7 is the latest one.

The MINIX 3 vision has been guided by a number of core principles:

- **Separation of concerns:** Split the OS into components that are well isolated from each other.
- **Least authority:** Grant each component only the powers it needs to do its job and no more.
- **Fault tolerance:** Admit that bugs exist and plan to recover from them while continuing to run.
- **Dynamic update:** Plan on staying up all the time, even in the face of major software updates.
- **Standards compliance:** Be POSIX-compliant on the outside but don't fear change on the inside.

We believe we have made a good start toward producing a general-purpose, POSIX-compliant operating system with excellent fault tolerance. As hardware speeds have shot up over the past two or three decades, we do not believe that most users really care about squeezing the last drop of performance out of the hardware. For example, in MINIX 3, a build of the entire operating system (about 120 compilations and a dozen links) takes under 10 seconds on a modern PC. Good enough.

If you get the MINIX 3.1.7 CD-ROM from www.minix3.org and install it, to the user it looks like other UNIX systems, albeit with fewer ported application programs (so far), since that has not been our focus. But on the inside it is completely different. It is a multiserver operating system based on a small microkernel that handles interrupts, low-level process management, and IPC, and not much more. The bulk of the operating system runs as a collection of user-mode processes. The key concept here is "multiserver operating system"—the design of the system as a collection of user-mode drivers and servers with independent failure modes. While the term "microkernel" gets a lot of attention—and microkernels are widely used in cell phones, avionics, automotive, and other embedded systems where reliability is crucial [2]—it is the multiserver aspect of the system that concerns us here.

The microkernel runs in kernel mode, but nearly all the other OS components run in user mode. The lowest layer of user-mode processes consists of the I/O device drivers, each driver completely isolated in a separate process protected by the MMU and communicating with the kernel via a simple API and with other processes by message passing. The next layer up consists of servers, including the virtual file server, the MINIX file server, the process manager, the virtual memory manager, and the reincarnation server. Above this layer are the normal user processes, such as X11, shells, and application programs (see Figure 1).

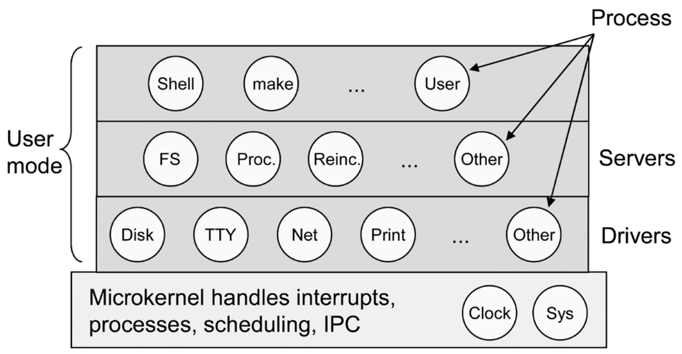


FIGURE 1: THE STRUCTURE OF MINIX 3

The reincarnation server is the most unusual part of the design. Its job is to monitor the other servers and drivers, and when it detects a problem, it replaces the faulty component (driver or server) on the fly with a clean version taken from the disk (or, in the case of the disk driver, from RAM). Since most errors are transient, even after a driver crashes (e.g., due to a segmentation fault after dereferencing a bad pointer), the system can, in many cases, continue without user processes even knowing part of the system has been replaced. We ran a fault-injection test in which 2.4 million faults were intentionally injected into drivers, and although we got thousands of driver crashes, the system continued to run correctly in all trials [3]. It didn't crash even once.

Current Status of MINIX 3

MINIX 3 is not standing still. The MINIX 3 Web site has been visited 1.7 million times and the CD-ROM image has been downloaded over 300,000 times. We have been selected to participate in the Google Summer of Code in 2008, 2009, and 2010. There is a wiki, a twitter feed, an RSS feed, and an active Google newsgroup.

Since the 2007 paper in *;login:*, there have been numerous improvements to MINIX 3, large and small. Here is a brief summary of where the system stands now.

- POSIX-compliant operating system with virtual memory and TCP/IP networking
- User interface is typically X11, although a simple GUI (EDE) is also available
- Various device drivers (e.g., Gigabit Ethernet, OSS audio framework)
- Virtual file system with support for various file systems (e.g., MFS, ISO, HGFS)
- Three C compilers (ACK, gcc, LLVM), as well as C++, Perl, Python, PHP, and more
- Various shells (bash, pdksh, sh)
- Choice of BSD, GNU, or V7 utilities (awk, grep, ls, make, sed, and all the others)
- Many packages (e.g., Apache, Emacs, Ghostscript, mplayer, PostgreSQL, QEMU, vi)
- Software RAID-like layer that protects integrity even from faulty disk drivers
- Numerous correctness, conformance, code coverage, and performance test suites

In addition, other changes to the system are planned for the near future, including:

- Porting of the DDEkit [4], which will give us many new Linux device drivers
- Asynchronous messaging (which means a faulty client cannot hang a server)
- Kernel threads

In short, while the system is not nearly as complete as Linux or FreeBSD, neither is it a toy kernel. It is a full-blown UNIX system but with a completely different and highly modular, reliable structure internally. This also makes it a good research vehicle for testing out new OS ideas easily.

Current Research

We have various ongoing research areas, all focused on the goal of producing a highly reliable, modular system according to the principles given above on modern hardware. We are also committed to producing a usable prototype that clearly demonstrates that you can build a real system using our ideas. Here is a brief rundown of five of the projects.

LIVE UPDATE

Due to its modular structure, we would like to be able to update large parts of the system on the fly, without a reboot. We believe it will be possible to replace, for example, the main file system module with a later version while the system is running, without a reboot, and without affecting running processes. While Ksplice [5] can make small patches to Linux on the fly, it cannot update to a whole new version without a reboot (and thus downtime).

Our starting point for the live update is that the writer of the component knows that it will be updated some day and takes that into account. In particular, the old and new components actively cooperate to make the update process go smoothly. Nearly all other work on live update assumes that the update comes in as a bolt out of the blue and has to be done instantly, no matter how complicated the state the old component is in. We believe this is the wrong approach and that by delaying the update for a few seconds we can often make it much easier and more reliable.

Live-updating MINIX 3 is much easier than live-updating monolithic kernels, because each component runs as a separate process. To update a component, the reincarnation server sends the component an *update* message. The component then finishes its current work and queues, but does not start, any new requests that come in while it is finishing up. It then carefully saves its state in the data store so the new component can find it later. After the new version has been started, it goes to the data store to fetch the saved state, reformats and converts it as necessary, and starts processing the queued requests. The other components should not even notice this upgrade—certainly, not those running user programs.

It is entirely possible that some data structures have been reorganized in the new version. For example, information previously stored as a list may now be in a hash table, so it is the job of the new version to first convert the stored data to the new format before running. In this way, the system may be able to run for months or years, surviving many major upgrades, without ever needing a reboot.

CRASH RECOVERY OF STATEFUL SERVERS

The current system can handle recovery from crashes of stateless drivers and servers but cannot transparently recover components that have a lot of internal state, which is lost in a crash. We need to make sure that the internal state can be recovered after a crash. Checkpointing is not a good approach for components such as the file system, which have a huge amount of state that changes thousands of times every second. Instead, we are experimenting with techniques to replicate and checksum the state internal to each process in real time, as it changes. To do this we need to change the compiler to insert code to do this, which is why we switched to LLVM, which has hooks for this (which ACK and gcc do not).

The idea is that after the crash of a stateful component, a scavenger process comes in and inspects the core image of the now-dead component. Using the replicated data and checksums, it can determine which items are valid and which are not and so recover the good ones. For example, if we can tell which entries in the inode table are corrupted (if any) and which are correct, we can do a much better job of recovering files that are fully recoverable. To some extent, journaling can also achieve this goal, but at greater expense.

SUPPORT FOR MULTICORE CHIPS

We are working on supporting multicore chips in a scalable way. Intel has already demonstrated an 80-core CPU, and larger ones may be on the way. We do not believe the current approach of treating these chips as multiprocessors is the right way to go, since future chips may not be cache-coherent or may waste too much time fighting for cache lines and software locks. Instead, we intend to treat each core as a separate computer and not share memory among them. In effect, our approach is to treat a multicore chip more or less like a rack full of independent PCs connected by Ethernet, only smaller. But we are not sure yet quite how this will work out, so we may allow some restricted sharing. We do believe (along with the designers of Barrelfish [6]) that not sharing kernel data structures across cores will scale better to the large chips expected in the future.

While some people are looking at how to parallelize applications, fewer are looking at how to parallelize the operating system. In the case of the multiserver MINIX 3, in which the processor has many cores and the operating system is made up of many processes, it seems a natural fit to put each process on its own dedicated core and not have it compete for resources with any other processes. These resources include L1 and L2 cache lines, TLB entries, entries in the CPU's branch prediction table, and so on, depending on hardware constraints. In other words, when work comes in, the component is all set up and ready to go, with no process-switching time. If cores are essentially free and the multiserver/multicore design speeds up the OS by, say, 20% or 50% or 100%, even if it uses, say, 3, 5, or 10 cores to do so, that is a gain that you would not otherwise have. Having a lot of cores sitting around idle is worth nothing (except slightly lower energy costs).

NEW FILE SYSTEM

Pretty much all current file systems are recognizably derived from the 1965 MULTICS file system [7], but a lot has changed in 45 years. Modern disk drives exhibit partial failures, such as not writing a data block or writing it to a different location from the intended one, but may still report back success. New classes of devices such as SSDs and OSD (Object-based Storage Devices) are being introduced. These devices differ from disk drives in several ways, with different price/performance/reliability trade-offs. Volume managers and other tools have broken the “one file system per disk” bond but have also complicated storage administration significantly.

When RAID was introduced, it was made to look like “just another disk” to be backward compatible with existing systems. Placing RAID at the bottom of the storage stack has caused several reliability, heterogeneity, and flexibility problems. In the presence of partial failures, block-level RAID algorithms may propagate corruption, leading to unrecoverable data loss. Block-level volume management is incompatible with new device access granularities (such as byte-oriented flash interfaces). Even a simple task such as adding a new device to an existing installation involves a series of complicated, error-prone steps.

In order to solve these problems, we are working on a clean-slate design for a new storage stack. Similar to the network stack, the new storage stack has layers with well-defined functionalities, namely:

- The naming layer (handles name and directory handling)
- The cache layer (handles data caching)
- The logical layer (provides RAID and volume management)
- The physical layer (provides device-specific layout schemes)

The interface between these layers is a standardized file interface. Since the new stack breaks compatibility with the traditional stack, we run it under the virtual file system, so the OS can mount both a disk partition containing a legacy file system and a partition containing the new one. Thus old and new programs can run at the same time.

The new storage stack solves all the aforementioned problems. By performing checksumming in the physical layer, all requests undergo verification, thus providing end-to-end data integrity. By having device-specific layout schemes isolated to the physical layer, RAID and volume management algorithms can be used across different types of devices. By presenting a device management model similar to ZFS’s storage pools, the new stack automates several aspects of device administration. In addition, since the logical layer is file-aware, RAID algorithms can be provided on a per-file basis. For instance, a user could have crucial files automatically replicated on his own PC, on a departmental or master home PC, and on a remote cloud, but not have compiler temporary files even written to the disk.

Virtualization

The original work on virtual machine monitors, which goes back over 35 years [8], focused on producing multiple copies of the underlying IBM 360 hardware. Modern virtualization work is based on hypervisors to which guest operating systems can make numerous calls to access services and get work done. In effect, they are more like microkernels than virtual machine monitors. We believe the boundary between microkernels and virtual machines is far from settled and are exploring the space of what exactly the hypervisor should do.

One of the ideas we are looking at is having a dual kernel. In addition to the regular microkernel to handle interrupts, service requests from drivers and servers, and pass messages, there is a component running in kernel mode to handle VM exits. However, the two parts—microkernel and hypervisor—run in different address spaces (but both in kernel mode) to avoid interfering with one another. Taking this idea to its logical conclusion, we may have one microkernel and as many hypervisors as there are virtual machines, all protected from one another. This arrangement will, hopefully, give us the best of both worlds and allow us to explore the advantages and trade-offs of putting functionality in different places. In addition, we will look at moving as much of the hypervisor functionality to user mode as possible.

We also have some novel ideas on ways to employ this technology to reuse some legacy software, such as device drivers.

Conclusion

MINIX 3 is an ongoing research and development project that seeks to produce a highly dependable open source operating system with a flexible and modular structure. While the grants pay for the PhD students and postdocs and a small number of programmers, we are dependent, like most open source projects, on volunteers for much of the work. If you would like to help out, please go to www.minix3.org to look at the wish list on the wiki, and read the Google MINIX 3 newsgroup. But even if you don't have time to volunteer, go get the CD-ROM image and give it a try. You'll be pleasantly surprised.

ACKNOWLEDGMENTS

We would like to thank Ben Gras, Philip Homburg, Kees van Reeuwijk, Arun Thomas, Thomas Veerman, and Dirk Vogt for their great work in programming various parts of the system. Without their efforts, we would have a paper design but not an actual working system that people can use. They also gave feedback on this paper.

REFERENCES

[1] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Roadmap to a Failure-Resilient Operating System," *;login:*, vol. 32, no. 1, Feb. 2007, pp. 14–20.

[2] G. Heiser, "Secure Embedded Systems Need Microkernels," *;login:,* vol. 30, no. 6, Dec. 2005, pp. 9–13.

[3] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Fault Isolation for Device Drivers," *Proceedings of the 39th International Conference on Dependable Systems and Networks*, 2009, pp. 33–42.

[4] http://wiki.tudos.org/DDE/DDEKit.

[5] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic Rebootless Kernel Updates," *Proceedings of the 2009 ACM SIGOPS EuroSys Conference on Computer Systems*, 2009, pp. 187–198.

[6] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The Multikernel: A New OS Architecture for Scalable Multicore Systems," *Proceedings of the ACM SIGOPS 22nd Symposium on OS Principles*, 2009, pp. 29–43.

[7] R. C. Daley. and P. G. Neumann, "A General-Purpose File System for Secondary Storage," *Proceedings of the AFIPS Fall Joint Computer Conference*, 1965, pp. 213–229.

[8] R. P. Parmelee, T. I. Peterson, C. C. Tillman, and D. J. Hatfield, "Virtual Storage and Virtual Machine Concepts," *IBM Systems Journal,* vol. 11, 1972, pp. 99–130.