MINIX 3: A Highly Reliable, Self-Repairing Operating System

Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum
Dept. of Computer Science, Vrije Universiteit Amsterdam, The Netherlands
{jnherder, herbertb, beng, philip, ast}@cs.vu.nl

1. INTRODUCTION

Different kinds of people use computers now than several decades ago, but operating systems have not fully kept pace with this change. It is true that we have point-and-click GUIs now instead of command line interfaces, but the expectation of the average user is different from what it used to be, because the user is different. Thirty or 40 years ago, when operating systems began to solidify into their current form, almost all computer users were programmers, scientists, engineers, or similar professionals doing heavy-duty computation, and they cared a great deal about speed. Few teenagers and even fewer grandmothers spent hours a day behind their terminal. Early users expected the computer to crash often; reboots came as naturally as waiting for the neighborhood TV repairman to come replace the picture tube on their home TVs. All that has changed and operating systems need to change with the times.

Modern computer users are from a broad cross-section of society. Most are not very technical and do not have a clue how to program their video recorder. They do use electronic devices, however, such as televisions, digital still cameras, camcorders, cell phones, MP3 players, stereos, and DVD players. Most of them have a set of mental expectations that we call *The TV model*. It goes like this:

1. You buy the device.
2. You turn it on.
3. It works perfectly for the next 10 years.

Most electronic devices fit this model well, the one exception being the personal computer. In addition to mind-numbing complexity (e.g., even networking experts have trouble configuring a wireless base station, despite the 500-page manual), they are prone to crashes and blue screens of death, issues unheard of with other electronic devices. Other problems include viruses, worms, spyware, and spam, but in this article we focus on operating system problems.

Most modern computer users want their systems to work all the time and never crash, ever. In engineering terms, this requires a mean time to failure (MTTF) appreciably longer than the expected lifetime of the computer. The average user virtually never complains that the computer itself is too slow (e.g., it cannot update a spreadsheet fast enough), although complaints about the speed of the Web are common. Over time, the relationship between speed and reliability has reversed. Most users now consider the reliability of the computer to be far more important than its speed, the reverse of 40 years ago.

Yet operating system reliability is still poor. To make the research challenge more explicit, consider a device driver that contains a fatal bug such as a store through an invalid pointer or an infinite loop. In commodity operating systems, when this bug is triggered, it crashes or hangs the entire system because the buggy code is running in kernel mode. All user programs that were running at the time the bug struck are killed, all user work is lost, and all FTP, Web, and e-mail transfers are abruptly aborted. This is the situation we are trying to address.

Since faults in software are a fact of life, our approach to reliability is to anticipate failures and design a self-repairing operating system. Studies have shown that software contains about 6–16 bugs per 1000 lines of code [3, 29, 30]; it is simply infeasible to get all code to be correct. Therefore, we have designed MINIX 3 in such a way that certain major faults are properly isolated, defects are detected, and failing components can be replaced on the fly, often transparent to applications and without user intervention or loss of data or work. For example, in our view, a bug in a minor driver, such as a printer driver, should not crash the system.

To avoid any confusion, we are focused on consumer machines (desktop, notebook, PDA, etc.) and possibly embedded systems, but not on high-end servers, multicore chips, multiprocessors, and clusters. And within that category, we are somewhat more interested in ordinary computers used by nonprofessionals and also low-end machines (such as found in Third World countries), rather than large state-of-the-art powerhouses.

2. THE WHEEL OF REINCARNATION

It should be obvious to anyone well versed in computer technology that the problem of system crashes is due to the software, not the hardware. While considering possible solutions, we were quickly confronted with the issue of how operating systems came to be unreliable. We believe the answer goes back 40 years, when computers were slow and performance was king. By building the operating system as a single monolithic program that ran in kernel mode, it could be made faster, and that was all that counted back then.

Before getting into our revitalized operating system design, we wish to point out that there are cycles in computer science, just as there are in other areas. Sometimes designs that are popular fade out and come back decades later, when the times (technology, applications, users, etc.) have changed. Let us now look briefly at some of these cycles as motivation for our research.

Decades ago, there was a battle between people who believed programs should be compiled versus those who believed they should be interpreted. Compiled programs ran faster, but used more memory and were harder to debug, since run-time errors were not caught by the interpreter. In the 1960s, compilation was the dominant mode. For example, programs in FORTRAN, COBOL and PL/I were generally compiled. In the 1970s, especially in academia, the wheel turned, and UCSD Pascal became popular. The P-compiler compiled programs from Pascal to the assembly language of an imaginary stack machine called P-code and then interpreted the P-code programs. Although interpretation cost a factor of 10 on machines 1000x slower than current ones, few people complained; the ease of porting programs (by just writing a new P-code interpreter) and the interpreter's help in debugging them were worth the cost. Interpretation was in. During the 1980s, interpretation largely fell out of favor with the rise of C, but then in the 1990s, Java brought back the 1970s model of compilation to an intermediate language, JVM, followed by interpretation of the JVM program. Interpretation was in again, at least for some applications.

As a second example, consider virtual machines [33]. In the mid-1960s, a group of programmers at IBM's Cambridge (MA) research lab who were not happy with the operating system on the IBM 360, wrote their own, wholly unlike IBM's. It consisted of two layers: a lower layer called CP-67, which ran in kernel mode and created the illusion of multiple, virtual, IBM 360 model 67s, and an upper layer, which ran in user mode and consisted of a simple single-user operating system, CMS, that ran on each virtual machine. The idea of a virtual machine system can thus be traced back 40 years. Although IBM has supported VM/370 and its successors for decades, outside the world of IBM mainframes, it was largely forgotten until the advent of VMware, which debuted in 1999, more than 30 years after CP-67 was up and running. Later, the development of Xen [10] caused this old warhorse to suddenly become popular again.

Another example that is relevant to our work is the use of software-based versus hardware-based protection in operating systems. In the early 1960s, Burroughs produced a popular computer called the B5000 that was based on having all software written in a type-safe language, ALGOL. Hardware protection to distinguish kernel-mode and user-mode components was not needed because the language provided complete protection. Later, CPU protection rings and MMU hardware were developed and unsafe languages, such as C, could be used, because the hardware isolated processes. Recently though, Microsoft Research has developed a new operating system, called Singularity [23], which does not rely on an MMU for isolation. Instead, it once again largely uses a type-safe language (Sing#) to achieve a dependable computing platform.

Our point in presenting these examples is to demonstrate that ideas come and go in computer science. What is hot in one decade may become cold in the next, but hot again one or two decades later. One should not be afraid to pick up old ideas that are quite sound but have fallen out of favor for one reason or another and reexamine them in light of the current situation.

3. MICROKERNELS

Another idea that has come and gone is that of the microkernel [38]. In the mid-1980s, various research groups produced operating system kernels that were very small, including Amoeba [28], Chorus [5], Mach [1], and V [7]. Unlike the virtual machine systems such as CP-67, the programming interface offered by the microkernel was not a clone of the hardware, but featured processes and an interprocess communication facility, similar to, but much simpler than, modern operating systems. In most cases, a complete operating system ran as a single process in user mode on top of the microkernel. User processes obtained operating system services by using the microkernel to send messages to the user-mode operating system process. One popular example was Berkeley UNIX running on top of the CMU Mach microkernel.

To make a long story short, the performance of these microkernels left something to be desired so they never became mainstream, although Mac OS X is basically a modified version of Mach with Berkeley UNIX running on top of it, albeit in kernel mode [39].

After this historical digression, we get back to our main story. When we set out to produce a new operating system that could be self-organizing, (in the sense of automatically detecting and repairing its own faults) we started with the premise that kernel bugs are usually fatal, whereas user-mode bugs often are not, and began wondering if microkernels might be the answer. If we could reduce the size of the kernel from 5 million lines of C/C++ (the approximate size of the Windows XP kernel) or 2.5 million lines of C (the size of the Linux kernel) to, say, 5000 lines of code, we reasoned that we could reduce the number of kernel bugs by roughly three orders of magnitude, clearly a vast improvement.

Of course moving code to user space does not eliminate any bugs, but we thought by no longer running the operating system as a giant user process and instead splitting it into many small, tightly restricted processes, each with limited functionality, we could perhaps build a self-repairing system in which (1) bug-induced damage could not propagate and affect the entire system, and (2) faulty components could be replaced on the fly, during system operation, thus greatly improving reliability.

While we had to consider the performance, work in the 1990s had already shown that the carefully engineered L4 microkernel [27] running in kernel mode plus L^4Linux running as a giant user process got the performance loss due to the extra context switches down to under 5–10% [18]. The research challenge was thus not improving the performance; we already knew that could be done with careful design. The hard part was to split the operating system into many user-space processes to prevent problems from spreading and make the system self-repairing, something that had not been done before.

The result, MINIX 3, is a microkernel-based multiserver operating system that provides fault isolation and can automatically recover from many failures. The architecture of MINIX 3, its self-repairing property, and the most important reliability features are discussed in the following sections.

4. THE ARCHITECTURE OF MINIX 3

The goal of our design was to produce an operating system far more reliable than current ones. Our approach is to run all servers and drivers as independent user-mode processes on top of a tiny, trusted microkernel. Each component is tightly restricted to prevent problems from spreading. Furthermore, the operating system's well-being is constantly monitored by a special server, called the reincarnation server. If a defect is detected, the system automatically attempts to (transparently) replace the malfunctioning component, or if that is not possible, at least to shut down gracefully. In other words, we have made our system self-repairing, a key characteristic of a self-organizing system. The architecture of MINIX 3 is shown in Fig. 1.

To avoid having to rewrite large amounts of straightforward but otherwise boring code, such as interrupt handlers and keyboard drivers, we started with MINIX 2, which, although the device drivers ran inside the kernel, was a reasonable starting place. This system was then very heavily modified to become an essentially new system. For lack of a better name, we just incremented the counter and called it MINIX 3. However, just as Windows 3.11 and Windows XP bear a common first name but are very different systems, so are MINIX 2 and MINIX 3.

We have intentionally kept the design simple by using straightforward techniques rather than complex techniques aimed at optimizing resource usage. We would rather waste a small amount of memory with a simple algorithm than save memory and introduce bugs with a complex algorithm. Of course, this strategy is a trade-off between reliability and other demands such as performance and scalability, but the problem, after all, is not a shortage of resources; the problem is weak reliability due to too many bugs in the code.

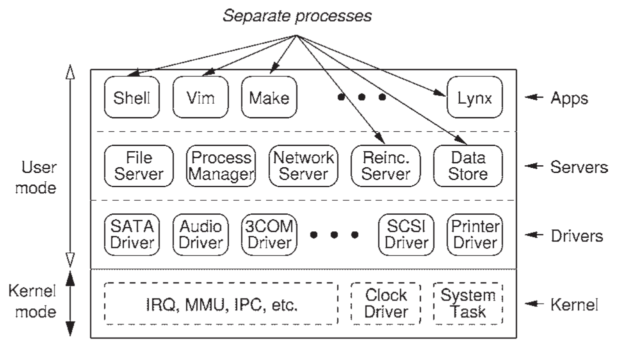

Figure 1: The architecture of the MINIX 3 multiserver operating system. Although a logical layering of processes can be seen, all processes are treated equally by the kernel.

The microkernel of MINIX 3 is responsible for interrupt handling, programming the CPU and MMU, scheduling, interprocess communication, and kernel calls. Clearly these are the things normal operating systems do, but only the basic mechanisms. Policies, in contrast, are part of the user-mode operating system servers. There is no attempt to create the illusion of multiple copies or distinct partitions of the hardware as in VM/370, VMware, Xen, or Exokernel [11].

Processes are essentially the same as in UNIX systems. They have private address spaces protected by the MMU hardware, registers, and can hold many resources—but a substantial part of the resources, including file descriptors and signal masks, are now managed by user-space servers. Although servers and drivers require more privileges than ordinary applications (such as the ability to perform I/O), each process is treated equally by the kernel. The novelty here is the fact that *all* servers and drivers run as independent user-mode processes.

Processes can communicate with each other and with the kernel by sending typed, fixed-length messages. Message passing is synchronous using the rendezvous principle. If a process tries to receive a message and none has been sent, it blocks. Likewise, a sender with no receiver also blocks. When both sender and receiver are ready to dance, the kernel copies the message from the sender's address space to the receiver's. This scheme eliminates all the complexity associated with buffer management, as well as buffer overruns due to variable-sized messages.

When synchronous message passing is not adequate, a notification mechanism is used. This scheme allows a sender to notify a process that is not currently receiving without blocking itself. The notification is stored as a bit in a bitmap and the sender is allowed to continue. When the target is ready to receive a message, the kernel creates a message of type NOTIFICATION and passes it the notification bitmap. Interrupt handlers use this mechanism to avoid blocking themselves when notifying device drivers that are busy at the time of the interrupt.

The kernel contains two processes to support the user-mode parts of the operating system. One is the clock driver, which is so intimately tied to scheduling that it would be difficult and inefficient to put it in user space. The other is the system task, which handles the approximately 35 kernel calls that authorized processes can make to the kernel. These calls allow user-mode operating system processes to perform actual I/O, copy blocks of data from one address space to another, and similar low-level functions. These kernel calls should not be confused with POSIX system calls, which user processes can make and which are discussed below. Although part of the kernel space, both the clock driver and system tasks are scheduled like any other processes. Their memory maps just happen to coincide with the kernel's address space.

The next layer up consists of all device drivers except the clock. These include drivers for the keyboard, mouse, screen, disk, Ethernet, sound, etc. Each driver is a separate process and is scheduled when its turn has come. It can send messages to other processes as well as make kernel calls, for example, to read or write I/O ports. With only a few exceptions, driver processes are normal user processes prevented from touching anything outside their own address space by the MMU hardware.

Above the driver layer is the server layer. These, too, are ordinary processes. The file server is basically a 4500-line C program that accepts requests from user programs to open, read, write, and close files, and perform other POSIX calls relating to files. A typical system call, say, read, starts when a user process sends a fixed-length message to the file server asking it to read from a previously opened file some number of bytes. If the file server does not have the requested data in its cache, it sends a fixed-length message to the disk driver process asking the latter to read the data into the file server's buffer cache. Once the data are in place, the file server makes a kernel call asking the system task to copy it to the user. Data is copied the same number of times as in all UNIX systems. The only extra overhead here is four messages and two extra context switches. As we will show in Sec. 7, sending a message and doing a context switch takes about 500 nsec, so this overhead is really negligible. Even 10,000 system calls/sec would eat up less than 1% of the CPU.

Other servers that provide POSIX functionality include the process manager and the network server. The process manager handles POSIX system calls such as fork and generally manages memory allocation policy. It also handles signals and some of the simple POSIX system calls, such as getpid. While the kernel is responsible for low-level process management mechanisms, it is the process manager that implements all policies.

The network server provides a complete TCP/IP stack that supports BSD sockets. Performance measurements have shown that fast Ethernet easily runs at full speed, and initial tests show that we can also drive gigabit Ethernet at full speed in user space, as discussed in Sec. 7.

5. SELF-REPAIRING PROPERTY

MINIX 3 includes two special servers, the reincarnation server and the data store, that support automatic repair of common failures. The reincarnation server manages all servers and drivers and monitors the system's well-being. The data store provides a place to backup state and is used by the reincarnation server to publish information about the system's configuration.

While a human user detects a driver crash when the system freezes, the operating system needs different techniques. The reincarnation server can detect component crashes because it is the parent process of all the drivers and servers. If one crashes, it is collected by the reincarnation server, the same way all UNIX systems allow parent processes to collect children that have exited. In addition, the reincarnation server can periodically send a status request to selected drivers. If no reply is received within the timeout interval, further action is taken.

Whenever a problem is detected, the reincarnation server looks up the malfunctioning component's policy script that governs the recovery procedure. The script normally causes the failed component to be replaced with a fresh copy, but it can also record the event in a log file, move the core dump of the dead process to a special directory for subsequent debugging, or even send e-mail to a remote system administrator. It is important to note that while all this is happening, the system continues to run normally and no processes are terminated.

The underlying assumption of MINIX 3's self-repairing property is that problems can be detected and that restarting a component makes it possible to repair a defect. The defects that MINIX 3 can repair include, for example, transient failures and aging bugs. Transient failures are problems caused by specific configuration, such as deadlocks or timing issues that are unlikely to happen. Aging bugs are implementation problems that cause a component to fail over time, for example, when it runs out of buffers due to memory leaks.

MINIX 3 is not designed to deal with malicious processes that do not violate any rules nor with logical errors of components that adhere to the specified system behavior but fail to perform a request correctly. For example, consider a printer driver that accepts a print job and replies that it successfully handled the request, but in fact prints garbage. Such bugs are virtually impossible to catch in any system. It is up to the user to replace a malfunctioning component in this case.

Recovery of some lost state after a component restart and informing dependent processes of changes in the system configuration is done through the data store. The data store is a tiny database server that allows system components to store and retrieve strings by name. For example, after a driver crash and recovery, the file server can ask it for the internal name of the new driver to allow the system to transparently recover from driver failures. As another example, the RAM disk driver stores the base address and size of the physical memory taken by the RAM disk at the data store. In the future, it might be possible to log incoming driver IPC to allow transparent recovery of individual driver calls, as in Nooks [36].

To provide transparent recovery to the user, an integral approach is required whereby the driver, server, and applications may be involved. For example, after a driver failure, the reincarnation server replaces the driver and informs the file server or network server about this event to initiate further recovery. If the server cannot recover, it pushes the error up to the application level, which in turn might retry the failed operation or warn the user.

Although in principle MINIX 3 can handle both server and driver failures, our system is currently mainly designed to reincarnate dead device drivers, since such failures are more common [8, 36].

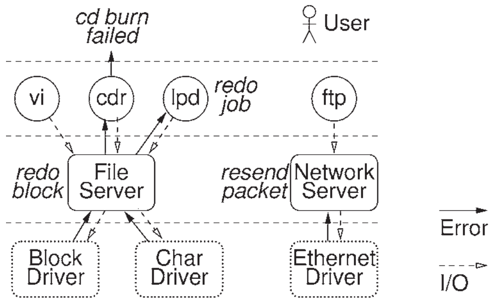

Figure 2: Recovery scenarios for device driver for block, character, and Ethernet devices. Errors are pushed up when they cannot be handled at the current level.

Different recovery scenarios for different kinds of device drivers are shown in Fig. 2. Block device driver failures and Ethernet driver failures can be recovered transparently by retrying the failed block or reinserting missing network packets, respectively. Character driver failures are pushed to the application level, because it is impossible to tell where the stream of data was interrupted. Depending on the application and type of I/O application-level recovery may or may not be possible. For example, a print job can be restarted by the printer daemon, but a failure while burning a CD-R is not recoverable.

As an example of how self-repairing works, consider the crash of an Ethernet driver. The reincarnation server will notice the failure and restart the dead driver. The reincarnation server publishes the new system configuration at the data store, which notifies the network server of this event. Then the network server starts its recovery procedure. It reinitializes the driver and when the TCP stream is interrupted, it retransmits lost packets. In this case, the use of a reliable transport protocol enables full recovery transparent to the application level and without user intervention. Sec. 7 shows how the recovery overhead is limited to only a few percent even in the light of frequent Ethernet driver crashes. For example, one simulated crash every 4 seconds leads to performance degradation of just 8%, but the transfer successfully completes without even bothering the user.

6. THE TOP 10 RELIABILITY FEATURES

MINIX 3 has many features that enhance its reliability above that of conventional monolithic operating systems where all the code is contained in a single binary program running in kernel mode. Below we will briefly mention the top 10 reliability features.

First, the amount of code that runs in kernel mode has been kept to an absolute minimum, in our case, about 4000 lines of executable code—including the clock and system task. When all the headers and comments are included, the full kernel can be printed in under 100 pages. A single programmer can easily understand all of it, whereas no one can fully understand all of the 2.5 million lines of the Linux kernel or much larger Windows XP kernel. It is tough to get a program bug free when no one understands the whole thing, as changes in one part can affect other parts.

Second, although moving code to user space does not eliminate bugs, it greatly reduces the amount of damage a bug can do. A bug in, say, a sound driver in the kernel can easily crash the operating system; the same bug in a user-mode sound driver will crash only the driver, since it is encapsulated in a private address space protected by the MMU hardware. Faults in one process thus cannot corrupt other processes' memory. Using 10 bugs per 1000 lines of code as a plausible number [3, 29, 30], the MINIX 3 kernel probably has fewer than 50 bugs, whereas the kernels of Linux and Windows XP may have over 25,000 and 50,000 bugs, respectively. Getting large numbers of bugs out of the kernel and into tightly restricted user processes certainly is a reliability enhancement.

Third, since drivers run in user-mode, they cannot use privileged CPU instructions, such as disabling interrupts and performing I/O, which we see as a reliability feature. Instead, the driver has to make a kernel call giving a list of I/O ports to read or write. The kernel checks each call to make sure the printer driver cannot touch the I/O ports for the disk, and so on. Similarly, kernel calls to manage interrupt lines are available. The overhead here is negligible (about 1 μsec for the kernel call) but reliability is enhanced by preventing buggy drivers from touching devices that they do not own.

Fourth, the operating system is self-repairing as described in Sec. 5. All drivers and servers are children of the reincarnation server. When a server or driver crashes or misbehaves, the reincarnation server notices and can often transparently start a new one, as described above. Currently we can restart most device drivers and a few servers that maintain no or little state. In theory, we could also restart some of the core operating system servers, but we have not yet implemented that.

Fifth, our interprocess communication uses synchronous rendezvous communication using fixed-length messages. The message size is not a parameter of the IPC calls; it is a system-wide compile-time constant, allowing all message buffers to be easily set to the correct (static) size. This scheme eliminates many bugs associated with buffer management as well as stack overruns caused by unexpectedly large messages. In the occasional case that a variable-length data structure has to be passed, such as a file name, a pointer to it is passed instead.

Sixth, interrupts and message passing are unified in a very simple way. When an interrupt happens, the corresponding driver is notified. If it is waiting for a message, it gets the interrupt immediately as a message. If it is busy, it gets the interrupt notification message as soon as it is able to handle it without any complex locking schemes. Converting interrupts to notifications at a very low level eliminates many of the problems associated with kernel reentries and makes system programming much easier.

Seventh, wild stores due to bad pointers in a driver cannot corrupt any memory outside the driver's own address space. All servers and drivers are encapsulated in a private address space that is protected by the MMU hardware, just as with ordinary user applications. Most commonly, a wild store will ultimately bring down the driver, in which case it will automatically be restarted. Some work may be lost, for example, a job may not be printed, but the system will not crash.

Eighth, damage from buffer overrun vulnerabilities that are commonly exploited by viruses and worms to execute injected code is limited. Many exploits work by overrunning a buffer to trick the program into returning from a function call using an overwritten stacked return address pointing into the overrun buffer. In MINIX 3, this attack does not work because instruction and data space are split and only code in (read-only) instruction space can be executed.

Ninth, infinite loops in drivers no longer hang the system. The scheduling algorithm will notice that the looping driver is using large amounts of CPU time and gradually lower its priority. Meanwhile, the reincarnation server will notice that the driver is not answering its status requests, and will eventually kill it and start a fresh copy. Since a copy of the disk driver is kept in RAM all the time, it is even possible to restart a looping disk driver with a fresh version.

Tenth, the system can be configured to restrict drivers and servers to a subset of the available kernel calls. Which kernel calls a process can make is controlled by a bitmap in its process table entry. User processes can only send messages (and then only to servers). An audio driver, for example, is allowed to request I/O, but cannot change the process privileges, which is the job of the reincarnation server. Likewise, the process manager is the only process that can instruct the kernel to start or stop processes.

There are various other reliability features available as well [20], but these give an idea. Basically, running drivers and servers as user processes rather than as kernel functions in monolithic operating systems makes it possible to encapsulate them, restrict what they can do, monitor their behavior, and replace them on the fly if need be. No other operating system to date has all these properties.

7. PERFORMANCE

In the 1980s, microkernels were plagued with performance problems, but since then we have learned a great deal about how to solve them. The current version of L^4Linux has a performance about 2–3% worse than Linux running on the bare metal. Our performance is not quite so good yet because we have not even attempted to tune it (remember, our research interest is in reliability, not performance), but we could probably achieve something close to this if we tried hard.

Measuring performance is very tricky. What we really are interested in is how much the microkernel design costs. Running some benchmark on MINIX 3 and on Linux or FreeBSD would tell us that the various systems have different compilers, different file systems, different memory management strategies, etc. There would be no way to see how much of the difference is due to, for example, the different file caching algorithms. Also, the nature of the benchmark matters a lot. For example, in a networking test, the speed of the Ethernet might easily be the bottleneck. Two systems that were capable of driving the Ethernet at full speed would be declared equal, while one of them might be able to go faster had the network been able to handle it.

Thus we did what scientists normally do: run controlled experiments with two systems that differ in only one parameter. Then we know for sure that any observed differences are due to that parameter. In our case, the experiments compared MINIX 2, which has in-kernel drivers, against MINIX 3, which has user-space drivers. Furthermore, we measured the recovery overhead incurred by the self-repairing mechanisms of MINIX 3. All tests were done on a 2.2 GHz Athlon.

Our first test measured individual system calls. The results are shown in Fig. 3. The first table entry shows that the getpid call—a simple message from a user process to the process manager and an immediate answer—takes 180 nsec more in MINIX 3 than in

MINIX 2. The difference is due to the extra checking done since the number of messages and context switches is the same. Since little real work is done, getpid shows that a typical request-reply IPC interaction costs about 1 μsec. Reading and writing is about 8% slower in MINIX 3 due to the extra two messages and extra two context switches required for user-mode drivers.

Call	MINIX 2	MINIX 3	Δ	Ratio
getpid	0.831	1.011	0.180	1.22
lseek	0.721	0.797	0.076	1.11
open+close	3.048	3.315	0.267	1.09
read 64k+lseek	81.207	87.999	6.792	1.08
write 64k+lseek	80.165	86.832	6.667	1.08
creat+wr+del	12.465	13.465	1.000	1.08
fork	10.499	12.399	1.900	1.18
fork+exec	38.832	43.365	4.533	1.12
mkdir+rmdir	13.357	14.265	0.908	1.07
rename	5.852	6.812	0.960	1.16
Average				**1.12**

Figure 3: System call times for kernel-mode vs. user-mode drivers and their differences and ratios. All times are in μsec.

We also ran some macroscopic tests, as shown in Fig. 4. These are actual programs rather than just system calls. The first two are large *make* jobs, one to build the kernel and the user-mode servers and drivers; and one to build the POSIX test suite. Again we see an overhead of about 8%. These tests are heavily I/O bound. The other five tests consisted of sorting, sedding, grepping, prepping, and uuencoding a 64-MB file, respectively. These tests were run only once, reducing the file server's cache hit ratio to 0, so every block came from the disk.

Program	MINIX 2	MINIX 3	Δ	Ratio
Build image	3.630	3.878	0.248	1.07
Build POSIX tests	1.455	1.577	0.122	1.08
Sort	99.2	103.4	4.2	1.04
Sed	17.7	18.8	1.1	1.06
Grep	13.7	13.9	0.2	1.01
Prep	145.6	159.3	13.7	1.09
Uuencode	19.6	21.2	1.6	1.08
Average				**1.06**

Figure 4: Run times in sec for various I/O-bound programs.

To measure disk I/O performance, we read and wrote a file in units of 1 KB to 64 MB. The tests were run many times, so the file being read was in the file server's 12-MB cache except for the 64-MB case, when it did not fit. The disk controller's internal cache was not disabled. Another test read the raw block device corresponding to the hard disk. This test bypasses the file server's buffer cache and just tests moving bits from the disk. Writing to the raw device would destroy the contents of the file system on it, so that test was not performed. The overhead of user-mode drivers is shown in Fig. 5. We measured a performance hit ranging from 3% to 18% with an average of 8.4%. The worst performance is for 1-KB writes, but the absolute time increase is only 457 nsec, so writing 1000 1-KB blocks per second would waste less than half a millisecond. The ratio decreases when more I/O is done since the relative overhead decreases. For the 64-MB tests the overhead is only 3% to 5%.

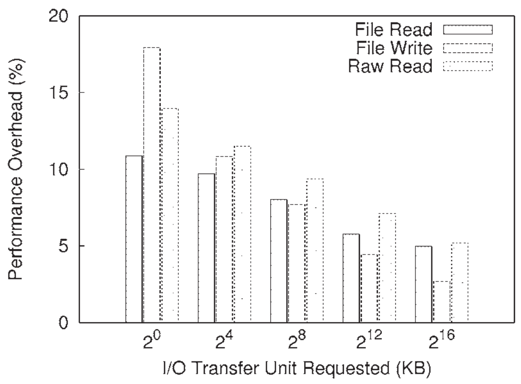

Figure 5: Performance overhead of user-mode drivers compared to kernel-mode drivers when reading and writing increasingly large chunks of a large file and reading the raw device.

We also tested the networking performance of user-mode drivers. The test was done with the Intel Pro/100 card as we did not have a driver for the Intel Pro/1000 card. We were able to drive the Ethernet at full speed. In addition we ran loopback tests, with the sender and receiver on the same machine and observed a throughput of 1.7 Gbps. Since this is roughly equivalent to using a network connection to send at 1.7 Gbps and receive at 1.7 Gbps at the same time, we are confident that handling gigabit Ethernet with a single unidirectional stream at 1 Gbps should pose no problem with a user-mode driver.

To measure the costs of the protection and recovery mechanisms of our system, we initiated a TCP transfer using the *wget* utility to retrieve a large (512-MB) file from the Internet. For this test we used the user-mode RealTek 8139 Ethernet driver, and repeatedly killed it during the transfer using a SIGKILL signal (to simulate a crash) with varying intervals between the simulated crashes. As described in Sec. 5, the reincarnation server detects such a failure and restarts the Ethernet driver. Because lost data was automatically resent due to the use of the TCP protocol, full recovery transparent to the application and without user intervention was possible.

Even though the Ethernet driver repeatedly failed, *wget* successfully completed in all cases; the only difference is a little performance degradation as shown in Fig. 6. The mean recovery time for the RealTek 8139 driver failures is 0.36 sec, which is mainly due to the TCP retransmission timeout. The transfer times range from

47.46 sec without interruptions to 63.03 sec in the worst case, when the driver is killed once per second. The loss in throughput due to driver failures ranges from 25% to just 1% in the best case. With no failures there is no performance overhead.

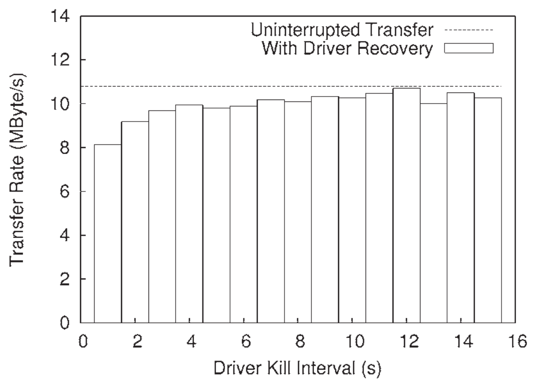

Figure 6: Networking throughput when using *wget* retrieve a 512-MB file from the Internet while repeatedly killing an Ethernet driver with various time intervals.

All in all, the performance loss due to our design is in the 5–10% range, and could no doubt be improved with careful tuning. The self-repairing functionality of our system adds a little overhead only when actual failures occur. We consider this performance loss completely acceptable in trade for the better reliability.

8. THE USER VIEW OF MINIX 3

Using MINIX 3 is like using a normal multiuser UNIX system. Of course, since the system has only been under development for about 18 months, with a small core of developers, it is not as complete as Linux or FreeBSD yet. Nevertheless, over 400 UNIX programs have been written or ported. A few of them include:

Shells: *ash, bash, pdksh, rsh, zsh*
Editors: *ed, emacs, ex, nvi, vim, elvis, elle, mined, sed*
Languages: *awk, C, bison, flex, perl, python, yacc*
Programming tools: *cdiff, make, patch, tar, touch*
Networking: *ssh, pine, ftp, lynx, mail, rlogin, wget*
File utilities: *cat, cp, bzip2, compress, mv, dd, uue*
Text utilities: *grep, head, paste, prep, sort, spell, tail*
Administration: *adduser, cron, fdisk, mknod, mount*

In addition, nearly all of the other standard UNIX shell and file utilities the GNU utilities and the X Window System are available. Large GUI applications, such as the Firefox web browser and the Thunderbird e-mail client, have not yet been ported, but can be used on MINIX 3 by launching X clients over the network. MINIX 3 also supports multiple virtual terminals, which is more-or-less the command-line interface equivalent of running multiple Xterms on X. This feature is important on low-end machines with very limited memory (e.g., 8 MB) that cannot support X.

While MINIX 3 is not yet competitive with much more mature systems in terms of software available, there is clearly enough already to eliminate any doubt that it could be done given some programming effort. Furthermore, the overall performance is surprisingly good. For example, a system build of the kernel plus all standard user-mode drivers and servers (125 compilations), takes under 4 sec, as mentioned above. Furthermore, the time interval between leaving the multiboot monitor and getting the login prompt is only 5 sec. At that time a POSIX-conformant operating system is ready to use.

The MINIX 3 website is *www.minix3.org*. It contains all the source code, documentation, and news about MINIX 3. It also has a CD-ROM image containing the binaries and sources that can be downloaded, burned onto a CD-ROM and then booted as a live CD. Partitioning the hard disk is not necessary to test MINIX 3, but of course, to do anything useful a free hard disk partition is required. Installation is simple. Just log in as root and type *setup*. After you have answered a few simple questions such as which keyboard you have (US, French, German, etc.), MINIX 3 will install itself in about 10 minutes. In the first few months after it became available, 50,000 people downloaded the CD-ROM image.

A USENET newsgroup, *comp.os.minix*, hosts a lively discussion forum where people can ask questions, exchange code, and get and give help.

9. RELATED WORK

Below we will briefly discuss some related work, focusing on complete operating systems, not just microkernels. Writing a small kernel is one thing; putting a complete (POSIX-conformant) operating system on top of it and then porting hundreds of programs to it and being able to use it as a UNIX clone is something quite different.

Our work differs significantly from other approaches that isolate device drivers in that we combine full compartmentalization of the operating system in user mode with explicit mechanisms to recover from critical failures. We rely on hardware protection offered by the MMU and take the process model to its logical conclusion.

Nooks proposed and implemented in-kernel wrapping of drivers to isolate faults in commodity operating systems using manually written wrappers for each component to be isolated [36, 37]. In contrast, we do not use wrappers, but put all servers and drivers in unprivileged user-mode processes, and deal with faults differently.

Virtual machines-like approaches like VM/370, Exokernel, VMware, Denali, and Xen [33, 11, 35, 40, 2] are powerful tools for running multiple services, but cannot prevent a bug in a device driver from crashing the hosted operating system. Fault isolation can be achieved by means of para-virtualization and running each driver in a dedicated operating system instance [13, 25, 26]. In contrast, MINIX 3 isolates drivers by running each as an independent user-mode process, encapsulated in a private address space that is protected by the MMU hardware.

Various microkernels have been built, such as Mach, V, and Amoeba, [1, 41, 7, 28], but they all had unwrapped device drivers inside the kernel. User-mode drivers have been used before [12, 9, 24], but only as a partial approach where lots of extensions and other code remain in the kernel. More sophisticed multiserver designs that move more functionality out of the kernel also have been proposed. For example, Perseus, DROPS, and NIZZA are multiserver systems on top of the L4 microkernel [31, 16, 17, 27]. Other efforts include QNX, GNU Hurd, Nemesis, Mungi, and SawMill Linux [21, 6, 32, 19, 15, 14]. These approaches compartmentalize operating system, but do not have explicit mechanisms to recover from failures in critical components, such as device drivers, like we do.

Language-based protection and formal verification can also be used to isolate drivers, as proposed by SPIN, Coyotos, VFiasco, and Singularity [4, 34, 22, 23]. These approaches are complementary to ours, since the user-mode servers and drivers of our operating system can be implemented in a type-safe language and the small size of each component might make code verification feasible. For example, we have experimented with languages like Cyclone and Objective Caml.

Various other experimental systems have been written over the years, but few, if any, of these have gotten beyond the stage of a paper design and a primitive lab prototype, usually with an ad hoc API and little user-level software. Until the system is really built and can be used in a serious way by a real user community, it is hard to say whether it is a suitable alternative.

10. CONCLUSIONS

We have presented a design that can improve operating system reliability. It consists of a very small kernel and a large number of modular, tightly controlled user-mode processes, each performing a specific task, such as being a device driver or server. The design has many reliability features not present in current systems, including the ability to withstand wild stores, infinite loops, and other (transient) errors in drivers, the ability to strictly limit what kernel calls, I/O ports, message targets, etc. operating system components can use.

Moreover, MINIX 3 has the ability to restart failing or failed operating system components on the fly without rebooting the system. If a critical component, such as a device driver, crashes, the reincarnation server notices and initiates the recovery process. Character device driver failures have to be dealt with at the application level, whereas full recovery transparent to applications and without user intervention is possible for block device drivers and Ethernet drivers.

While other systems have had some of these features, no previous research team has put them all together, built a serious open-source prototype that is POSIX-conformant, ported hundreds of programs to it, and gotten 50,000 people to download it.

All in all, we believe this is a small step in the direction of an operating system with a mean time to failure approaching that of a television set.

REFERENCES

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. of USENIX'86*, pages 93–113, 1986.

[2] P. T. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proc. 19th ACM Symp. on Oper. Syst. Prin.*, pages 164–177, 2003.

[3] V. Basili and B. Perricone. Software Errors and Complexity: An Empirical Investigation. *Commun. of the ACM*, 21(1):42–52, Jan. 1984.

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. J. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proc. 15th ACM Symp. on Oper. Syst. Prin.*, pages 267–284, 1995.

[5] A. Bricker, M. Gien, M. Guillemont, J. Lipkis, D. Orr, and M. Rozier. A New Look at Microkernel-Based UNIX Operating Systems: Lessons in Performance and Compatibility. In *Proc. EurOpen Spring 1991 Conf.*, pages 13–32, May 1991.

[6] M. I. Bushnell. The HURD: Towards a New Strategy of OS Design. *GNU's Bulletin.*

[7] D. Cheriton. The V Kernel: A Software Base for Distributed Systems. *IEEE Software*, 1(2):19–42, Apr. 1984.

[8] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating System Errors. In *Proc. 18th ACM Symp. on Oper. Syst. Prin.*, pages 73–88, 2001.

[9] P. Chubb. Get More Device Drivers Out of the Kernel! In *Proc. Linux Symp.*, pages 149–162, July 2004.

[10] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J. Matthews. Xen and the Art of Repeated Research. In *Proc. Usenix Tech. Conf. (Freenix Track)*, pages 135–144, 2004.

[11] D. Engler, M. Kaashoek, and J. J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *Proc. 15th ACM Symp. on Oper. Syst. Prin.*, pages 251–266, 1995.

[12] A. Forin, D. Golub, and B. Bershad. An I/O System for Mach 3.0. In *Proc. Second USENIX Mach Symp.*, pages 163–176, 1991.

[13] K. Fraser, S. Hand, I. Pratt, A. Warfield, R. Neugebauer, and M. Williamson. Safe Hardware Access with the Xen

Virtual Machine Monitor. In *Proc. 1st Workshop on Oper. Sys. and Arch. Supp. for the on demand IT Infrastructure (OASIS-2004)*, Oct. 2004.

[14] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J. Tidswell, L. Deller, and L. Reuther. The SawMill Multiserver Approach. In *ACM SIGOPS European Workshop*, pages 109–114, Sept. 2000.

[15] S. M. Hand. Self-Paging in the Nemesis Operating System. In *Proc. 3rd Symp. on Oper. Syst. Design and Impl.*, pages 73–86, 1999.

[16] H. Härtig, R. Baumgartl, M. Borriss, C.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schonberg, and J. Wolter. DROPS OS Support for Distributed Multimedia Applications. In *Proc. 8th ACM SIGOPS European Workshop*, pages 203–209, Sept. 1998.

[17] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter.

[18] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The Performance of -Kernel-Based Systems. In *Proc. 6th Symp. on Oper. Syst. Prin.*, pages 66–77, Oct. 1997.

[19] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke. The Mungi Single-Address-Space Operating System. *Software: Practice and Experience*, 28(9):901–928, July 1998.

[20] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Modular System Programming in MINIX 3. *USENIX ;login:*, 31(2):19–28, Apr. 2006.

[21] D. Hildebrand. An Architectural Overview of QNX. In *Proc. USENIX Workshop in Microkernels and Other Kernel Architectures*, pages 113–126, Apr. 1992.

[22] M. Hohmuth and H. Tews. The VFiasco Approach for a Verified Operating System. In *Proc. 2nd ECOOP Workshop on Prog. Lang. and Oper. Sys.*, July 2005.

[23] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, Redmond, WA, USA, Oct. 2005.

[24] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Gotz, C. Gray, L. Macpherson, Y.-T. S. Daniel Potts, K. Elphinstone, and G. Heiser. User-Level Device Drivers: Achieved Performance. *Journal of Computer Science and Technology*, 20(5), Sept. 2005.

[25] J. LeVasseur and V. Uhlig. A Sledgehammer Approach to Reuse of Legacy Device Drivers. In *Proc. 11th ACM SIGOPS European Workshop*, pages 131–136, Sept. 2004.

[26] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proc. 6th Symp. on Oper. Syst. Design and Impl.*, pages 17–30, Dec. 2004.

[27] J. Liedtke. On μ-Kernel Construction. In *Proc. 15th ACM Symp. on Oper. Syst. Prin.*, pages 237–250, Dec. 1995.

[28] S. Mullender, G. V. Rossum, A. Tanenbaum, R. V. Renesse, and H. V. Staveren. Amoeba: A Distributed Operating System for the 1990s. In *IEEE Computer Magazine 23(5)*, pages 44–54, May 1990.

[29] T. Ostrand, E. Weyuker, , and R. Bell. Where the Bugs Are. In *Proc. of the 2004 ACM SIGSOFT Int'l Symp. on Software Testing and Analysis*, pages 86–96. ACM, 2004.

[30] T. Ostrand and E. Weyuker. The Distribution of Faults in a Large Industrial Software System. In *Proc. of the 2002 ACM SIGSOFT Int'l Symp. on Software Testing and Analysis*, pages 55–64. ACM, 2002.

[31] B. Pfitzmann and C. Stüble. Perseus: A Quick Open-source Path to Secure Signatures. In *2nd Workshop on Microkernel-based Systems*, 2001.

[32] T. Roscoe. The Structure of a MultiService Operating System. Ph.D. Dissertation, Cambridge University.

[33] L. Seawright and R. MacKinnon. VM/370—A Study of Multiplicity and Usefulness. *IBM Systems Journal*, 18(1):4–17, 1979.

[34] J. Shapiro, M. S. Doerrie, E. Northup, S. Sridhar, and M. Miller. Towards a Verified, General-Purpose Operating System Kernel. In *1st NICTA Workshop on Operating System Verification*, Oct. 2004.

[35] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proc. 2001 USENIX Ann. Tech. Conf.*, pages 1–14, 2001.

[36] M. Swift, M. Annamalai, B. Bershad, and H. Levy. Recovering Device Drivers. In *Proc. 6th Symp. on Oper. Syst. Design and Impl.*, pages 1–15, 2004.

[37] M. Swift, B. Bershad, and H. Levy. Improving the Reliability of Commodity Operating Systems. *ACM Trans. on Comp. Syst.*, 23(1):77–110, 2005.

[38] A. S. Tanenbaum, J. N. Herder, and H. Bos. Can We Make Operating Systems Reliable and Secure? *IEEE Computer*, 39(5):44–51, May 2006.

[39] A. Weiss. Strange Bedfellows. *NetWorker*, 5(2):19–25, June 2001.

[40] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proc. 5th Symp. on Oper. Syst. Design and Impl.*, 2002.

[41] M. Young, M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, and A. Tevanian. Mach: A New Kernel Foundation for UNIX Development. In *Proc. 1986 Summer USENIX conference*, pages 93–113, June 1986.