Construction of a Highly Dependable Operating System

Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum
Vrije Universiteit, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
{jnherder,herbertb,beng,philip,ast}@cs.vu.nl

Abstract

It has been well established that most operating system crashes are due to bugs in device drivers. Because drivers are normally linked into the kernel address space, a buggy driver can wipe out kernel tables and bring the system crashing to a grinding halt.

We have greatly mitigated this problem by reducing the kernel to an absolute minimum and running each driver as a separate, unprivileged user-mode process. In addition, we implemented a POSIX-conformant operating system, MINIX 3, as multiple user-mode servers. In this design, a server or driver failure no longer is fatal and does not require rebooting the computer.

This paper discusses how we designed and implemented the system, which problems we encountered, and how we solved these problems. We also discuss the performance effects of our changes and evaluate how our multiserver design improves operating system dependability over monolithic designs.

’Perfection is not achieved when there is nothing left to add, but when there is nothing left to take away.’

– Antoine de Saint-Exupéry [2]

1. Introduction

For the vast majority of computer users, the biggest perceived problem with using PCs is that they are not dependable and frequently fail. For example, think of the scenario where you buy a new peripheral device, say, an Ethernet card, plug it in, and install the driver. The network connection seems to work fine, but not much later your computer suddenly crashes while downloading a file. Crashes reoccur at seemingly random times, and you start to wonder whether your computer upgrade has anything to do with it.

In our research, we try to address this situation and prevent serious bugs in the operating system, such as a device driver dereferencing an invalid pointer, winding up in an infinite loop, or executing an illegal instruction, from crashing or hanging the computer.

1.1. Why do Systems Crash?

Today’s commodity operating systems use a standard monolithic design, in which the kernel contains the entire operating system linked in a single address space and running in privileged mode. The kernel may consists of different kernel modules, but nothing prevents one module from corrupting another. The lack of proper fault containment allows local problems to spread throughout the kernel and take down the entire system. This property is not caused by a bad implementation, but is inherent to the use of a monolithic design.

As it turns out, many dependability problems in monolithic operating systems are due to simple programming bugs causing fatal exceptions. Fault distribution studies [18, 19, 20] show that code contains 6-16 bugs per 1000 lines of executable code (LoC). Translated to multimillion-line monolithic kernels and using a conservative estimate of 10 bugs per 1000 LoC, the 2.5-million-line Linux kernel probably has at least 25,000 bugs; Windows has as at least double that.

A related problem is the frequent use of untrusted, third-party code, such as device drivers and other extensions, in the kernel—the most sensitive part of the operating system. Still, device drivers typically comprise about 70% of the operating system code, while they have a reported error rate of 3 to 7 times higher than ordinary code [1]. Not surprisingly, the majority of crashes are caused by drivers. For example, in Windows XP they are responsible for 85% of all crashes [15].

1.2. The Solution: Proper Fault Isolation

One of the key observations of the research reported in this paper is that a powerful technique for increasing system reliability is to run each device driver as a separate user-mode process, encapsulated in a private address space protected by the MMU hardware—just like for ordinary application programs. In this way, faulty code is isolated, so a bug in say, the printer driver, may cause printing to cease, but it cannot write garbage all over key kernel data structures and bring the system

Proceedings of the Sixth European Dependable Computing Conference (EDCC'06)
0-7695-2648-9/06 $20.00 © 2006 IEEE

down. In some cases, a faulty user-mode driver can even be killed and replaced without restarting other parts of the operating system.

We do not believe that bug-free code is likely to appear soon, certainly not in operating systems, which are usually written in C or C++. Unfortunately, programs written in these languages make heavy use of pointers, a rich source of bugs. Our approach is therefore based on the ideas of modularity and fault isolation. To keep faults from spreading we have compartmentalized the system by running all servers and drivers as isolated user-mode processes and reducing the part that runs in kernel mode to a bare minimum. Making the kernel small, in our case, under 4000 lines of code, greatly reduces the number of bugs it is likely to contain, since the small size also reduces its complexity and makes it easier to understand. In other words, our approach to reliability is to follow Saint-Exupéry's dictum and make the kernel as small as humanly possible. How that was done and what problems were encountered is described later in this paper.

1.3. Our Contribution

The concrete contribution of the research report here is the design and implementation of a fully compartmentalized operating system, MINIX 3. To properly isolate faults, we have removed all drivers from the kernel and run them as separate, unprivileged user-mode processes, protected by the MMU hardware. Since all servers also run in user mode in our design, only a tiny microkernel that does not contain any foreign, untrusted code is left in kernel mode. Each component has only the minimum privileges it needs in order to prevent failures from spreading. In our design, driver failures are no longer fatal and do not require rebooting the computer.

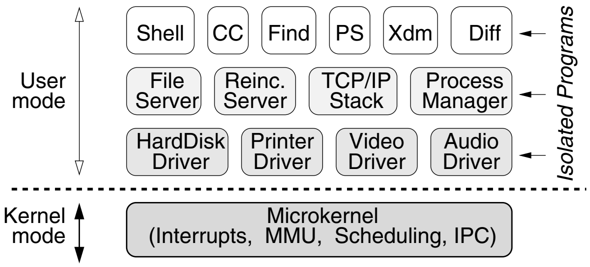

Figure 1: The design of our operating system is fully compartmentalized to properly isolate faults and enable recovery.

The loosely layered structure of our revitalized operating system is illustrated in Fig. 1. All applications, servers, and drivers run as isolated, user-mode programs on top of a tiny microkernel. The microkernel is responsible for low-level operations such as interrupt handling, scheduling, and programming the MMU and CPU. On top of the microkernel, we implemented user-mode device drivers to manage the hardware, including drivers for hard disk, printer, video, and audio. The user-mode drivers are managed by the reincarnation server, whereas the file server, network server and process manager provide operating system services to the application layer.

To the best of our knowledge no one before has put all the pieces together to build a fully modular, open-source, POSIX-conformant UNIX clone that is far more fault tolerant than normal UNIX systems, with a performance loss of only 5% to 10% compared to our base system with drivers in the kernel. In addition, our approach differs from related efforts as we do *not* focus on commodity operating systems. Rather than patching legacy systems, we use a new, lightweight design that can make future operating systems more dependable.

1.4. Paper Outline

We start out by surveying related work in improving operating system dependability (Sec. 2). Then we introduce the base system with in-kernel drivers and analyze the kernel-driver dependencies that we encountered (Sec. 3); we discuss the solutions we came up with and how we operationally moved all device drivers out of the kernel (Sec. 4); and we mention some other improvements that we made to the system (Sec. 5). We also give some performance considerations (Sec. 6) and briefly evaluate the system's main dependability features (Sec. 7). Finally, we present our conclusions (Sec. 8).

2. Overview of Related Work

This project is about building a more dependable operating system by removing device drivers from the kernel. Before describing our design in detail, we briefly discuss what others have done to improve operating system dependability.

2.1. Patching Commodity Systems

An important project to improve the dependability of commodity systems such as Linux is Nooks [15, 16]. Nooks keeps device drivers in the kernel but encloses them in a lightweight *protective wrapper* so that driver bugs cannot propagate to other parts of the operating system. All traffic between the driver and the rest of the kernel is inspected by the reliability layer.

Another project uses a kind of *virtual machine* to isolate device drivers from the rest of the system [10].

When a driver is called, it is run on a different virtual machine than the main system so that a crash or other fault does not pollute the main system. In addition to isolation, this technique enables unmodified reuse of drivers when experimenting with new operating systems.

A recent project ran Linux device drivers in user mode with small changes to the Linux kernel [12]. This work shows that drivers can be isolated in user-mode processes without significant performance degradation.

While isolating device drivers helps to improve the dependability of legacy operating systems, we believe that a proper, fully modular design from scratch gives more dependability. This includes encapsulating *all* operating system servers and drivers in independent, user-mode processes.

2.2. Architecting New Modular Designs

The opposite of a monolithic kernel is the microkernel, which contains only the barest mechanisms, but no policies. A microkernel provides interrupt handlers, a mechanism for starting and stopping processes, a scheduler, and interprocess communication, but ideally nothing else. Standard operating system functionality that is present in a monolithic kernel is moved to user space, and no longer runs at the highest privilege level.

Different operating system organizations are possible on top of a microkernel. A first step is running the operating system in a single user-mode server on top of a microkernel, for example, L^{4}Linux on top of the L4 microkernel [4]. This structure is often combined with specialized components as in DROPS [5] and Perseus [13]. In terms of dependability, this setup adds little over monolithic systems, since a single driver bug can still crash Linux.

In more modular designs, the operating system is split into a set of cooperating servers. Untrusted code such as third-party device drivers can be run in independent, user-mode modules to prevent faults from spreading. In principle, modular designs have great potential to increase dependability as each module can be tightly controlled.

SawMill Linux [3] would have been a more sophisticated approach to split the operating system into pieces and run each one in its own protection domain. However, the project was abruptly terminated in 2001 when many of the principals left IBM, and the only outcome was a rudimentary, unfinished prototype.

The GNU Hurd is a collection of servers that serves as a replacement for the UNIX kernel. Although the project's goal is similar to our, the distribution of functionality over the various servers is different. The current status seems to be that the multiserver system did not work as intended on top of either Mach or L4, and the project currently seeks another microkernel.

A recent multiserver system developed by Microsoft Research is Singularity [9]. In contrast to other systems, Singularity uses language protection and can run without the hardware protection offered by the MMU. The system can be characterized as a microkernel running a set of verifiably-safe, software-isolated servers. While language safety might be a viable approach to build reliable systems, Singularity means a paradigm shift for the programmer and is not backwards compatible with any existing applications.

3. Analysis of Drivers in the Base System

As the base for our work we started with an existing hybrid microkernel-based operating system, MINIX 2, which we then heavily modified. This approach allowed us to focus on mechanisms for improving dependability without having to write large amounts of code not relevant to this project. In MINIX 2, the file system, memory manager, and networking were already running as separate user-mode processes called servers. Drivers ran in kernel address space but were scheduled as separate processes, each with its own process table entry and stack. Nevertheless, a bug in a driver could wipe out the kernel and take down the entire system. Our goal was to ensure that a driver bug could not damage any code except the driver itself.

To determine what needed to be done, we first analyzed how drivers and the kernel depend on each other. A dependency means that a symbol, that is, a variable or function, can no longer be directly referenced by a device driver that is compiled as a separate program. The dependencies were found by copying all files of a given driver to a separate directory, trying to compile the driver isolated from the kernel, and inspecting the compiler and linker's errors. In the remainder of this section, we will discuss the dependencies that we found.

3.1. Who Depends on What?

We analyzed the interdependencies and were able to group them into roughly five categories based on who depends on whom or what. Each of the categories required a different approach to remove the dependencies:

A. Driver depends on kernel symbol.
B. One driver calls another.
C. Kernel task depends on driver.
D. Interrupt handler uses driver data.
E. Driver uses I/O hardware.

For each category, a different approach in removing the dependencies was needed.

(A) Many drivers directly call kernel functions or touch kernel variables, for example, to copy data to and from user-mode processes. The obvious solution is to add new kernel calls to support the drivers in user space.

(B) Sometimes one driver calls another. For example, drivers require services from the console driver to output a message. Like above, new message types can be defined to request services from each other.

(C) The kernel can depend on a driver, for example, when a timer expires and the watchdog function of a driver is called by the clock task. Events that originate in the kernel are now communicated to user space using nonblocking messages.

(D) Some interrupt handlers directly touched data structures of the in-kernel device drivers. The solution is to transform the hardware interrupt into a notification which is processed local to the driver in user space.

(E) All drivers interact with the I/O hardware, which they cannot not do in user space; attempts to read or write I/O ports result in an exception. Since only the kernel is allowed to do I/O, several kernel calls relating to I/O were added.

3.2. Functional Classification

In addition to looking at who depends on what, it is possible to ask: Why? What kinds of procedures are drivers calling, etc.? We analyzed the reasons that drivers were interacting with other parts of the system and found the main classes of dependencies as follows:

1. Actual device I/O.
2. Copying data.
3. Access to kernel information.
4. Enabling and disabling interrupts.
5. System shutdown.
6. Using the clock.
7. Debug dumps.
8. Assertions and panics.
9. Bad design.

This classification was helpful to find general approaches to resolve entire classes of dependencies, instead of ad hoc solutions for individual dependencies.

Class 1 dependencies occur because device drivers do I/O. With the exception of the RAM disk driver, all drivers do actual I/O, which requires reading and writing I/O device registers. On most machines, doing I/O is not possible in user mode. On some machines there may be a way to map a page containing I/O registers to user space or map some of the I/O ports to user space, but this is not always possible and should not be relied on.

Class 2 dependencies are due to drivers needing a way to copy data to or from user processes. When a driver is in the kernel, it can copy data anywhere it wants to, but when the driver is moved to user mode, it can no longer directly access other processes' address spaces.

Class 3 dependencies relate to some drivers' need to access the process table and other kernel data structures, from absolute memory to environment variables. None of these are available in user mode.

Class 4 dependencies exist because drivers often need to enable, disable, and manage interrupts on the devices they control. From inside the kernel, doing this is easy; from user mode it is impossible without help.

Class 5 dependencies arise because system shutdown is a peculiar business, with various idiosyncracies. Drivers play a role here shutting down their respective devices and cooperating when the servers shut down.

Class 6 dependencies relate to managing timers. Many drivers need to deal with time, for example, setting watchdog timers. In-kernel drivers have easy access to the clock and its functions. Outside the kernel, an explicit interface to the clock driver is needed.

Class 7 dependencies have to do with the debugging displays produced when a function key is struck. This used to be handled by the keyboard driver on behalf of other drivers, but with all the drivers in different user-mode processes, some other solution is needed.

Class 8 dependencies are caused by the presence of assertion statements in the code that are verified at run time. They previously caused panics, forcing a system shutdown. When they occur in user-mode processes, they need to be treated differently, for example, causing just the faulty component to be killed and restarted.

Class 9 dependencies were caused by bad design, for example, when variables that were really local were declared global. This is yet another example why in-kernel drivers are a bad idea. Kernel development is complex and does not enforce a proper coding style, easily leading to mistakes. Fortunately, these dependencies were easily fixed by moving them to the right file and declaring them static.

4. Moving Drivers out of the Kernel

The goal of the new system, MINIX 3, is to remove all device driver from the kernel and produce a true microkernel-based operating system, as shown in Fig. 1. Each driver will run in a separate user-mode process with its own private address space. One exception is the clock driver, which is very simple and remains in the kernel to facilitate process scheduling. Since each server also runs in user mode (in its own address space), only a tiny microkernel is left in kernel mode. All that this microkernel does is catch interrupts and convert each one to a message, select the next process to run, load the selected process registers and MMU entries, and handle the transport of fixed-size message between processes using the rendezvous principle. Everything else is done by user-space processes.

In this section, we describe what was needed to resolve the driver dependencies discussed in Sec. 3. We also discuss how we operationally removed all drivers from the kernel and how we transformed them into isolated, user-mode processes.

4.1. New System Calls

Some of the problems were solved by adding new kernel calls that drivers can make. The major ones are listed in Fig. 2. Below, we will briefly discuss the most important calls of interest to drivers.

Kernel Call	Purpose
SYS_VDEVIO	Read or write a vector of I/O ports
SYS_VIRCOPY	Safe copy between address spaces
SYS_IRQCTL	Set or reset an interrupt policy
SYS_GETINFO	Get a copy of kernel information
SYS_SETALARM	Set or reset a synchronous alarm
SYS_PRIVCTL	Restrict a process' privileges

Figure 2: A selection of common kernel calls. All calls require privileged operations and are handled by SYS.

The SYS_VDEVIO call is used to read or write (a set of) I/O ports. By specifying a vector of ports, a single call can return the values of multiple I/O ports. Similarly, a single call can write values to multiple I/O ports to start an I/O operation.

The SYS_IRQCTL call allows drivers to enable, disable, and manage interrupt handlers in several ways. Interrupts are still caught by the kernel, but are immediately converted into a nonblocking notification message that is sent to the associated user-mode driver, as discussed below.

The SYS_VIRCOPY call is used to copy data between two address spaces. Copying is only possible when the other party explicitly granted access to a region of its memory. The memory grant contains the process that is allowed to use it, read-write permission bits, the virtual base address, and the number of bytes to transfer. The kernel is responsible for validating the memory grant and performing the actual copy.

The SYS_GETINFO call provides a way for drivers to acquire kernel information they once could access directly by just reading memory. For example, drivers need access to environment variables that contain information about the system's hardware configuration or user-specific settings passed through the boot monitor.

The SYS_SETALARM call replaces the old method of handling watchdog timers—having each driver call internal clock routines to schedule a future call of one of its own internal functions. Instead, they now call the clock driver asking for a notification to be sent when the timer goes off. In the old system, the watchdog procedure ran as part of the clock process. Now it runs as part of the caller's process, a much cleaner design.

The SYS_PRIVCTL call can be used to set the privileges of each process in the system. The call can only be used by a privileged user-mode server, and is used, for example, to restrict the I/O ports that can be used by individual drivers. Other resources that can be restricted are discussed in Sec. 5.

In addition to the new calls listed above, another dozen low-level kernel calls exist, some carried over unmodified from the base system and some modified for the new system. They include process management, memory management, copying data between processes, device I/O and interrupt management, access to kernel data structures, and clock services.

4.2. Disentangling Interrupt Handlers

When the drivers and the interrupt handlers were in kernel mode, programmers were sometimes sloppy about which functionality went where. There was a tendency to put functionality that logically belongs in the driver in the interrupt handler. For the printer driver, for example, practically the entire logic of printing was in the interrupt handler.

When the drivers were moved out of the kernel, the issue of where the boundary between the driver and the handler should be became acute, since it was no longer possible to blithely slosh functionality back and forth. In a much more consistent way than was previously the case, interrupt handlers were stripped to their bare essentials—catching interrupts and sending messages to their respective drivers to unblock them. The logic of what to do next was put back in the drivers because we believe that interrupt handlers are so critical with respect to time and reliability that they should perform only those functions that simply cannot be done anywhere else.

Drivers can instruct the kernel to transform specific interrupts into notification messages using the new SYS_IRQCTL kernel call. After registration, drivers can tell the kernel to enable and disable hardware interrupts. The kernel catches all hardware interrupts with a *generic interrupt handler* that looks up which drivers are associated with the IRQ line, and sends a nonblocking notification message to each of them. Drivers that control multiple IRQ lines, such as our serial line driver, can specify an identifier for each IRQ line that is returned in the notification message. This way the driver can directly tell different interrupt requests apart.

4.3. Shared Code

In the old system, code was sometimes shared among drivers, for example, the main loop. In the new system this sharing is difficult because each driver is in a private address space. We considered inventing a complex mechanism for allowing shared code, but decided this violated the entire spirit of keeping the code simple and such a complex addition would have no doubt introduced new bugs. Furthermore, given the limited amount of sharing present (about 4300 bytes), it is entirely possible that the code to allow sharing would have cost more memory than the redundant code itself.

While a technically small issue, it is an important philosophical difference with existing systems. Performance can be grafted on later but dependability cannot. We strongly believe that what is needed is making systems more reliable and where simplicity and code reduction can be bought by wasting CPU time, memory, or disk space, within reason, designers should do so. For most programmers, wasting any resource just to have simpler code is anathema. This attitude needs to be upgraded.

Emphasizing simplicity and reliability is even true for consumer appliances that are under tight price constraints because the cost of a product recall to fix buggy code will dwarf the pennies saved by shaving off a few kilobytes of ROM.

4.4. Testing the Drivers

Operationally, after the dependencies were duly noted by compiling each driver in its own directory, the drivers were put back into the kernel. Then the problems were solved one by one as described above, for example, by replacing dependencies with kernel calls one at a time. By keeping each driver in the kernel until the last of its problems was solved, it was possible to maintain a working system during the process of removing the dependencies. When a driver was finally fully independent of the kernel, only then was it moved to user mode and the process repeated with the next driver. In theory, any driver could be easily reinserted into the kernel.

With these changes, it was possible to successfully move drivers for the hard disk, floppy disk, printer, keyboard, display, serial line, RAM disk, and various fast Ethernet boards to user mode. Work is currently in progress to develop new user-space drivers, including a driver for gigabit Ethernet, from scratch. All in all, we believe MINIX 3 provides a complete set of user-mode drivers, demonstrating that it is possible to build a fully compartmentalized system that is useful for real tasks.

It is worth noting that it is much easier to develop, test, and debug user-mode drivers than kernel-mode drivers as they are just ordinary user programs [6]. For example, drivers can be modified and tested without a reboot, which speeds up the debug cycle considerably.

5. Other System Improvements

Moving the drivers to user mode is the most important change we made, but there were also a few other changes to the operating system that are worth mentioning and may be applicable to other systems.

5.1. Improved IPC

While synchronous message passing using fixed-size messages is simple, fast, and requires very little code,it does have a downside—one that is increasingly apparent as more servers and drivers migrate to user mode. The problem is that if a user-mode process does a send but no subsequent receive, the receiver (which in some cases can be the kernel itself or the clock driver) will block when it tries to send a reply. This is unacceptable. Also, the proliferation of parts of the system in user mode makes it necessary to protect these processes from unexpected messages from other processes that have no business communicating with them at all.

To solve these problems, three additions were made to the IPC mechanism:

1. Possible IPC target were highly restricted.
2. In most situations, a rendezvous is enforced.
3. Nonblocking, asynchronous IPC was added.

First, each process has been given a bitmap telling which processes it may send to. The allowed receivers may include the drivers, the servers, the kernel, and users, all of whom are lumped together so a process can either send to users or it cannot. No distinction is made between user process i and user process j.

Second, with few exceptions, drivers and servers are required to use rendezvous message passing, which combines sending and receiving in one call. The caller will be blocked until the entire operation completes. Enforcing this rule eliminates the situation of a driver sending the kernel a message and not doing a receive to accept the reply, thus hanging the kernel.

Third, for those few cases where a server or driver cannot afford to block if the receiver is busy, a nonblocking option has been added. For example, a driver that cannot directly deliver the requested data to the file server can send a preliminary reply, and later wake up the file server using a nonblocking notification.

5.2. Restriction Policies

While putting drivers in user space prevents them from using privileged CPU instructions, such as I/O, the use of other resources, such as kernel calls, is not automatically restricted. Without further measures, a malfunctioning user-mode driver could still cause substantial damage. Therefore, we have carefully restricted the privileges of each server and driver according to the principle of least authority [14].

Examples of privileges that can be restricted for each server and driver in our system include the user ID and group ID to control the use of POSIX system calls, IPC primitives that can be used, who can request services from whom, individual kernel calls, memory access, IRQ lines controlled, and more. In addition, each driver can be restricted to a range of I/O ports, with attempts to access other ports refused.

These protection mechanisms are implemented by means of bitmaps that are statically declared as part of the process table. Each driver and server was given its own bitmap with possibly distinct privileges. This is space efficient, prevents resource exhaustion, and allows for fast permission checks since only simple bit operations are required.

5.3. Run-Time Configuration

Since our servers and drivers are normal user processes, we can control and manage them like ordinary applications. For example, we can start and stop device drivers on the fly. A special server, called the reincarnation server, is used to coordinate this procedure; it manages all servers and drivers in the system. The reincarnation server is also the only process that can use the SYS_PRIVCTL kernel call to control process privileges. Whenever a new server or driver is started, the reincarnation server assigns its restriction policy.

Another important benefit is that failures in servers and drivers can be easier detected, much like we can detect a crashing user process, and potentially recovered. Since drivers are now properly isolated, failures can no longer affect the entire system. The reincarnation server can detect defects and in some cases can recover the system on the fly [7].

6. Performance Optimizations

Performance measurements comparing MINIX 2 and MINIX 3 show that the average overhead introduced by our changes is limited to 5–10% [8]. While user-mode drivers introduce a performance hit, the overhead we measured is also due to other changes, such as stricter checks on all IPC. However, if performance is crucial, there are other areas where gains are possible. In the following subsections we will discuss some of them.

6.1. File System Block Size

One optimization is increasing the block size. It is well known that large blocks amortize the costs of a (slow) disk seek over more bytes. The disadvantage of large blocks is the disk space wasted by internal fragmentation in the last block of a file. However, the real loss depends on the file size distribution [17]. If most files are small, this effect is deadly; if most are large, the waste is acceptable.

We have experimented with different block sizes in MINIX 3 [8]. In one experiment, we changed the disk block size from 1 KB to 8 KB, reduced the number of blocks in the file system buffer cache by 8 (so it contains the same number of MB) and called the old and new system MINIX 3.0.0 and MINIX 3.0.1, respectively. We then ran tests writing a file in different size units ranging from 1 KB to 64 MB. The results are shown in Fig. 3.

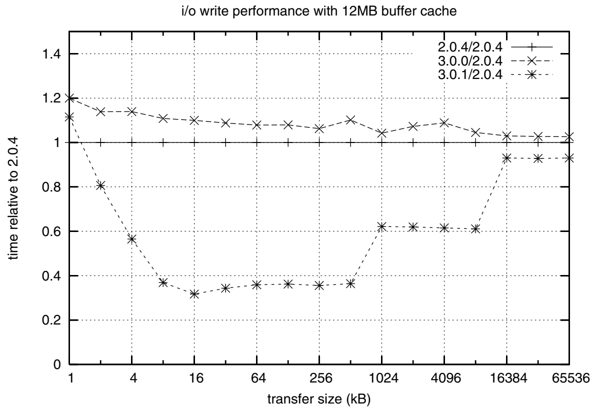

Figure 3: Ratio of write times for an 8-KB file system to those on a 1-KB file system.

The solid line at y = 1 is the baseline MINIX 2.0.4 system. The line with crosses (×) above the baseline shows the write times for MINIX 3.0.0 divided by the write times for MINIX 2.0.4. They are all above 1.0, indicating that MINIX 3.0.0 is slower than the original system. The curve with stars (∗) shows the performance of MINIX 3.0.1 divided by that of MINIX 2.0.4.

The curve for MINIX 3.0.1 exhibits four performance regimes. For sizes from 1 KB to 8 KB, write times decrease as the transfer size increases since there are fewer calls to the kernel for copying data and each kernel call adds overhead. For sizes from 8 KB to 512 KB, the performance is constant and almost three times as good as MINIX 2.0.4 with its 1-KB blocks. At 1 MB, the data no longer fits in the CPU's level 2 cache, so there is a substantial increase in write time because the file server is no longer being fed from the L2 cache. This effect is present in all versions of MINIX , but is more noticeable here due to the higher throughput of MINIX 3.0.1, which makes the slower memory accesses a larger percentage of the total execution time. At 16 MB, the 12-MB file server cache is exceeded and actual disk I/O occurs, with MINIX 3.0.1 outperforming MINIX 2.0.4 by about 7%, despite its use of user-mode drivers. Between the effects of the CPU's L2 cache, the file system cache, and the disk controller's track-at-a-time cache, it is difficult to interpret the measurements, but we believe Fig. 3 gives an impression of what is happening. In any case, the point here is a hit of 8% due to the user-mode drivers can be won back by adjusting other parameters, such as the block size.

6.2. Use of the RAM Disk

As another example, MINIX 3 can be configured to operate with the root device (which holds /bin, /lib, and other top-level directories) on the RAM disk. With 16-MB devoted to the RAM disk, the system is extremely responsive, since all the standard programs (shell utilities, compilers, etc.) load instantly. MINIX 3 also has a standard LRU file system buffer cache, but this tends to be full of data rather than programs. As mentioned above, the normal configuration of MINIX 3 in a development environment is to keep the entire source tree on the RAM disk, in which case a full system build takes about 5 seconds. Despite the use of a 16-MB RAM disk that must be copied from the hard disk at boot time, MINIX 3 boots in less than 10 seconds.

6.3. Future Optimizations

One thing that we have learned from our research is that with nearly all of the system outside the kernel, the precise interface between the user-mode servers and the kernel is very important. The aim has to be to reduce the number of kernel calls made.

For example, one area where we did it right was having a kernel call to read or write a vector of I/O ports in a single kernel call, thus eliminating unnecessary context switches. An area where we did it wrong was having the call from the file server to the kernel to move data from its cache to the caller's address space not use the same principle. If a process asks for n blocks worth of data, the file server should give the kernel a list of blocks and say: "Move all of these." Currently, the file server makes a separate kernel call for each block.

Although we have not looked carefully yet, we suspect that there are other trade-offs to be made that win back some performance. For example, the base system, MINIX 2, was never optimized for performance. Instead, the source code was setup to be well-structured and readable, even if better—but more complex—algorithms were available. The same holds for MINIX 3. While the system probably can be optimized in various ways to boost its performance, we are careful not to introduce unnecessary complexity.

7. Qualitative Dependability Evaluation

The evaluation criterion that we used is that in our system, no bug, no matter how bad, in a driver, should crash the computer, in exactly the same way that no bug in, say, a browser should crash the computer. The worst a bug in, say, the printer driver should be able to do is stop printing, print garbage, or hang the printer. It also should be possible to bring up a new printer driver without rebooting the operating system.

We believe that we have succeeded in designing and implementing a system that fullfills these goals, and that this system improves dependability over other operating systems in three important ways:

1. It reduces the number of fatal errors.
2. It limits the consequences of bugs.
3. It can recover from common failures.

We will now explain why. We will also compare how certain bugs affect MINIX 3 versus how they affect monolithic systems such as Windows, Linux, and FreeBSD.

7.1. Eliminating Fatal Errors

Our first line of defense is a very small kernel. We have been very strict and moved every driver (except for the clock driver) to user space. We could have made all kinds of exceptions for performance reasons, but have chosen for the best design from a dependability perspective, which is an important philosophical difference with existing systems.

It is well understood that more code means more bugs, so having a small kernel means fewer (potentially fatal) kernel bugs. Using an estimate of 10 bugs per

1000 LoC, (also see Sec. 1.1), there are less than 50 bugs in the kernel. With under 4000 lines of executable code (LoC) the size of our kernel is much smaller than most other microkernels. For example, L4 [11] contains over 10,000 LoC. While this difference is partly due to differences in functionality, we explicitly strive for simplicity. A small and simple kernel means the people can actually understand its full working, which also helps to get the implementation correct over time.

In contrast, a monolithic system such as Linux with 2.5 million lines of executable code in the kernel is likely to have at least 10 x 2500 = 25,000 bugs. For Windows XP, which has about 5 million lines of kernel code, the problem is twice as bad. Moreover, with multimillion-line systems, no person will ever read the complete code and fully understand its working, which can lead to servers and drivers interacting in poorly understood ways.

7.2. Reducing Bug Power

Another dependability improvement comes from the fact that we have placed the majority of the operating system code in isolated user-mode processes. While the total amount of code—and thus the number of bugs—has not changed, bugs are tamed, because when a bug is triggered, the effects will be less devastating by converting it from a kernel-mode bug to a user-mode bug.

All servers and drivers are normal user processes, each with its own address space protected by the MMU hardware and completely disjoint from the address spaces of the kernel and other servers, drivers, and user processes. In addition, we have tightly restricted the powers of each server and driver to an absolute minimum, as discussed in Sec. 5.2.

These points are crucial to the dependability as it prevents faults in one server or driver from spreading to a different one, in exactly the same way that a bug in a compilation going on in one process cannot affect what a browser in a different process is doing. For example, a user-mode sound driver that tries to dereference a bad pointer is killed by the process server, causing the sound to stop, but leaving the rest of the system unaffected. An infinite loop in a driver is also detected and simply causes the offending process' priority to be lowered, again affecting only a single I/O device.

In contrast, consider a bug in a kernel-mode sound driver that inadvertently overwrites the stacked return address of its procedure and then makes a wild jump when it returns. It might land on the memory management code and start corrupting key data structures such as the page tables. In general, monolithic systems tend to collapse when a bug is triggered.

7.3. Recovering from Failures

As described in Sec. 5.3, our new operating system design has potential to recover from many driver failures by simply restarting the driver. We can currently deal with transient failures, such as exceptions caused by rare hardware timing or unexpected boundary cases. Idempotent I/O can simply be retried by the file server without affecting applications. Some of the simpler servers can be recovered as well, but a crash of a core server is usually fatal because too much state is lost.

The defect detection mechanisms and recovery procedure are outside the scope of this paper, but are published elsewhere [7]. What is important, however, is that failures can often be repaired by restarting the failed component rather than rebooting the computer.

8. Conclusions

The primary achievement of the work reported here is that we have actually built a highly dependable, open-source, POSIX-conformant, multiserver operating system, MINIX 3, that can be freely downloaded for inspection. The fully modular structure of our system is shown in Fig. 1. We have discussed the implementation process and the resulting design in detail and evaluated its dependability and performance.

Our system is based on a microkernel whose complete source is under 4000 lines of executable code. This code represents the total amount of code that runs in kernel mode. To the best of our knowledge, this is by far the smallest microkernel in existence that supports a POSIX-conformant multiserver operating system in user mode. It is also the only one that has each server and each device driver running as a separate user-mode process, with many encapsulation facilities, and the ability to withstand and often recover from failures that normally would be fatal.

We make no claim that we can catch every bug, but we greatly improve the operating system's dependability by structurally eliminating many different types of failures. A small and understandable kernel means fewer fatal bugs and thus fewer crashes. Furthermore, while shifting code to user space does not eliminate bugs, it makes the consequences of failures less devastating since they are encapsulated in highly restricted user-mode processes. Moreover, most servers and all device drivers of the operating system are monitored and often can be automatically revived if a problem is detected.

Concluding, we have shown how we moved device drivers to user space and minimized the amount of code that runs in kernel mode in order to improve operating system dependability.

9. Availability

The system is called MINIX 3 because we started with MINIX 2 as a base and then modified it very heavily. It is free, open-source software, available via the Internet. You can download MINIX 3 from the official homepage at: http://www.minix3.org/, which also contains the source code, documentation, news, contributed software packages, and more. Over 75,000 people have downloaded the CD-ROM image in the past few months, resulting in a large and growing user community that communicates using the USENET newgroup *comp.os.minix*. MINIX 3 is actively being developed, and your help and feedback are much appreciated.

10. Acknowledgements

We would like to thank Mischa Geldermans, Jan Looyen, Ruedi Weis, Bruno Crispo, Chandana Gamage, and Carol Conti for their help and feedback.

Supported by the Netherlands Organization for Scientific Research (NWO) under grant 612-060-420.

References

[1] A. Chou and J. Yang and B. Chelf and 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.

[2] A.-M. de Saint-Exupéry. *Wind, Sand, and Stars*. Harcourt, Brace & Co, NY, 1940.

[3] 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.

[4] H. Härtig and M. Hohmuth and J. Liedtke and S. Schonberg and J. Wolter. The Performance of μ-Kernel-Based Systems. In *Proc. 16th ACM Symp. on Oper. Syst. Prin.*, pages 66–77, Oct. 1997.

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

[6] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Modular System Programming in MINIX 3. *;login: The USENIX Magazine*, 31(1):19–28, Apr. 2006.

[7] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Reorganizing UNIX for Reliability. In *Proc. 11th Asia-Pacific Computer Systems Architecture Conference*, Sept. 2006.

[8] J. N. Herder, H. Bos, and A. S. Tanenbaum. A Lightweight Method for Building Reliable Operating Systems Despite Unreliable Device Drivers. In *Technical Report IR-CS-018 [www.cs.vu.nl/∼jnherder/ir-cs-018.pdf]*, Vrije Universiteit, Jan. 2006.

[9] 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, Oct. 2005.

[10] J. LeVasseur and V. Uhlig and 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.

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

[12] 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.

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

[14] J. Saltzer and M. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), Sept. 1975.

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

[16] M. Swift, B. Bershad, and H. Levy. Improving the Reliability of Commodity Operating Systems. 23(1):77–110, 2005.

[17] A. S. Tanenbaum, J. N. Herder, and H. Bos. File Size Distribution on UNIX Systems: Then and Now. *ACM SIGOPS Operating Systems Review*.

[18] T.J. Ostrand and E.J. 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.

[19] T.J. Ostrand and E.J. Weyuker and and R.M. 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.

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