A Lightweight Method for Building Reliable Operating Systems Despite Unreliable Device Drivers

Technical Report IR-CS-018, January 2006

Jorrit N. Herder
Dept. of Computer Science
Vrije Universiteit, Amsterdam
The Netherlands
<jnherder@cs.vu.nl>

Herbert Bos
Dept. of Computer Science
Vrije Universiteit, Amsterdam
The Netherlands
<herbertb@cs.vu.nl>

Andrew S. Tanenbaum
Dept. of Computer Science
Vrije Universiteit, Amsterdam
The Netherlands
<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 halt. We have greatly mitigated this problem by reducing the kernel to an absolute minimum and running each driver as a separate, unprivileged process in user space. In addition, we implemented a POSIX-conformant operating system as multiple user-mode processes. In this design, all that is left in kernel mode is a tiny kernel of under 3800 lines of executable code for catching interrupts, starting and stopping processes, and doing IPC. By moving nearly the entire operating system to multiple, protected user-mode processes we reduce the consequences of faults, since a driver failure no longer is fatal and does not require rebooting the computer. In fact, our system incorporates a *reincarnation server* that is designed to deal with such errors and often allows for full recovery, transparent to the application and without loss of data. To achieve maximum reliability, our design was guided by simplicity, modularity, least authorization, and fault tolerance. This paper discusses our lightweight approach and reports on its performance and reliability. It also compares our design to other proposals for protecting drivers using kernel wrapping and virtual machines.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design;
D.4.5 [**Operating Systems**]: Reliability;
D.4.8 [**Operating Systems**]: Performance

General Terms

Design, Experimentation, Reliability, Performance

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

– Antoine de Saint-Exupéry [9]

1. INTRODUCTION

For many computer users, the biggest perceived problem with using computers is that they are so unreliable. Computer science researchers are used to computers crashing regularly and installing software patches every few months, and may find this normal. The vast majority of users, however, consider this lack of reliability unacceptable. Their mental model of how an electronic device should work is based on their experience with TV sets and video recorders: you buy it, plug it in, and it works perfectly for the next 10 years. No crashes, no semiannual software updates, no newspaper stories about the most recent in an endless string of viruses lurking out there somewhere. To make computers more like TV sets, the goal of our research is to improve the reliability of computer systems, starting with operating systems.

1.1 Why do Systems Crash?

The underlying reason that operating systems crash can be traced back to two fundamental design flaws they all share: too many privileges and lack of adequate fault isolation. Virtually all operating systems consist of many modules linked together in one address space to form a single binary program that runs in kernel mode. A bug in any module can easily trash critical data structures in an unrelated module and bring the system down in an instant. The reason for linking all the modules together in a single address space, with no protection between the modules, is that designers have made a Faustian bargain: better performance at the cost of more system crashes. We will quantitatively examine the price of this trade-off below.

A closely related issue is why crashes occur in the first place. After all, if every module were perfect, there would be no need for fault isolation between the modules because there would be no faults to propagate. We assert that most faults are due to programming errors (bugs) and are largely the consequence of too much complexity and the use of foreign code. Studies have shown that software averages 1-16 bugs per 1000 lines of code [27, 22, 2] and this range is surely an underestimate because only bugs that were ultimately found were counted. The obvious conclusion of these studies is: *more code means more bugs.* As software develops, each new release tends to acquire more features (and thus more code) and is often less reliable than its predecessor. Studies [22] have shown that the number of bugs per thousand lines of code tends to stabilize after many releases and does not go asymptotically to zero.

Some of these bugs are exploited by attackers to allow viruses and worms to infect and damage systems. Thus some alleged 'security' problems have nothing to do with the security measures in principle (e.g., flawed cryptography or broken authorization protocols), but are simply due to code bugs (e.g., buffer overruns that allow execution of injected code). In this paper, when we talk about 'reliability,' we also mean what is often referred to as 'security'—unauthorized access as a result of a bug in the code.

The second problem is introducing foreign code into the operating system. Most sophisticated users would never allow a third party to insert unknown code into the heart of their operating system, yet when they buy a new peripheral device and install the driver, this is precisely what they are doing. Device drivers are usually written by programmers working for the peripheral manufacturer, which typically has less quality control than the operating system vendor. When the driver is an open-source effort, it is often authored by a well-meaning but not necessarily experienced volunteer and with even less quality control. In Linux, for example, the error rate on device drivers is three to seven times higher than in the rest of the kernel [7]. Even Microsoft, which has the motivation and resources to apply tighter quality controls, does not do any better: 85% of all Windows XP crashes are due to code bugs in device drivers [25].

Recently, related efforts have been reported on isolating device drivers using the MMU hardware [26] and virtual machines [19]. These techniques are focused on handling problems in legacy operating systems; we will discuss them in Sec. 6. Our approach, in contrast, achieves reliability through a new, lightweight operating system design.

1.2 The Solution: Proper Fault Isolation

For decades, the proven technique for handling untrusted code has been to put it in a separate process and run it in user mode. One of the key observations of the research reported in this paper is that a powerful technique for increasing operating system reliability is to run each device driver as a *separate* user-mode process with only the minimal privileges required. 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 down.

In this paper, we will carefully distinguish between an operating system collapse, which requires rebooting the computer, and a server or driver failure or crash, which in our system, does not. In many cases, a faulty user-mode driver can be killed and replaced without restarting other parts of the (user-mode) 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. By breaking the system into many self-contained modules, each running in a separate user-space process, we were able to reduce the part that runs in kernel mode to a bare minimum and to keep faults in other modules from spreading. Making the kernel small greatly reduces the number of bugs it is likely to contain. The small size also reduces its complexity and makes it easier to understand, which also helps the reliability. Hence, we followed Saint-Exupéry's dictum and made the kernel as small as humanly possible: under 3800 lines of code.

One objection that has always been raised about such minimal kernel designs is that they are slow due to the additional context switches and data copies required when different modules in user space need to communicate with one another. This fear is mostly due to historical reasons and we argue that these reasons to a large extent no longer hold. First, new research insights has proven that minimal kernel design do not necessarily cripple the performance [3, 23, 15]. Smaller kernels and clever multiserver protocols help to limit the performance penalty. Second, the vast increase in computer power in the past decade has greatly reduced the absolute performance penalty incurred by a modular design. Third, we believe the time has come when most users would gladly sacrifice some performance for better reliability.

We provide a detailed discussion of the performance of our system in Sec. 5. However, we briefly mention three preliminary performance measurements to support our case that minimal kernel systems need not be slow. First, the measured time for the simplest system call, getpid, is 1.01 μs on a 2.2 GHz Athlon. This means that a program executing 10,000 system calls/sec wastes only 1% of the CPU on context switching overhead, and few programs make 10,000 system calls/sec. Second, our system is able to do a build of itself, including the kernel and the required user-mode parts, compiling 123 files and doing 11 links, within 4 seconds. Third, the boot time, as measured between exiting the multiboot monitor and getting the login prompt, is less than 5 seconds. At that point, a full POSIX-conformant operating system is ready to use.

1.3 The Contribution of This Paper

The research reported in this paper is trying to answer the question: How do you prevent a serious bug (e.g., a bad pointer or infinite loop) in a device driver such as a printer driver from crashing or hanging the entire operating system?

Our approach was to design a reliable, multiserver operating system on top of a tiny kernel that does not contain any foreign, untrusted code. To isolate faults properly each server and driver runs as a separate, unprivileged user-mode process. In addition, we added mechanisms to recover from common failures. We describe the reliability features in detail and explain why they are absent in traditional monolithic operating systems. We also discuss measurements we made of its performance and show that the reliability features slow the system down by about 5-10%, but make it able to withstand bad pointers, infinite loops, and other bugs that would crash or hang traditional operating systems.

While none of these individual aspects, such as small kernels, user-mode device drivers, or multiserver systems is new, no one before has put all the pieces together to build a small, flexible, modular, UNIX clone that is far more fault-tolerant than normal UNIX systems, while the performance loss is only 5% to 10% compared to our base system with drivers in the kernel.

Furthermore, our approach is fundamentally different from related efforts as we do *not* focus on commodity operating systems. Instead, we obtain reliability through a new, lightweight design. Rather than adding additional reliability code to patch unreliable systems, we split the operating system into small components and achieve reliability by exploiting the system's modularity. While our techniques cannot be applied to legacy operating systems, we believe they make future operating systems more reliable.

We start out by describing how our design compares to other operating system structures (Sec. 2) followed by an extensive discussion of the reliability features of our system (Sec. 3). Then we analyze the system's reliability (Sec. 4) and performance (Sec. 5) based on actual measurements. Finally, we examine related work (Sec. 6) and present our conclusions (Sec. 7).

2. OPERATING SYSTEM DESIGN

This project is about building a more reliable operating system. Before describing our design in detail we will briefly discuss how the choice of an operating system structure can immediately affect its reliability. For our purposes, we distinguish between two operating system structures: *monolithic systems* and *minimal kernel systems*. In addition there are other types of operating systems, such as exokernels [10] and virtual machines [24]. They are not directly relevant to this paper, but we will revisit them in Sec. 6.

2.1 Problems with Monolithic Systems

In a standard monolithic system, the kernel contains the entire operating system linked in a single address space and running in kernel mode, as shown in Fig. 1. The kernel may be structured into components or modules, as indicated by the dashed rectangular boxes, but there are no protection boundaries around the components. In contrast, the solid rounded rectangles indicate separate user-mode processes, each of which runs in a separate address space protected by the MMU hardware.

Monolithic operating systems have a number of problems that are inherent to their design. While some of these problems were already mentioned in the introduction, we summarize them here:

1. No proper isolation of faults.
2. All code runs at the highest privilege level.
3. Huge amount of code implying many bugs.
4. Untrusted, third-party code in the kernel.
5. Hard to maintain due to complexity.

This list of properties questions the reliability of monolithic systems. It is important to realize that these properties are not due to a bad implementation, but are fundamental problems that stem from the operating system design.

The kernel is assumed to be correct, while its size alone means that it must contain numerous bugs [27, 22, 2]. Moreover, with all operating system code running at the highest privilege level and no proper fault containment, any bug might be fatal. A malfunctioning third-party device driver, for example, can easily destroy key data structures and take down the entire system. That this scenario is a serious threat

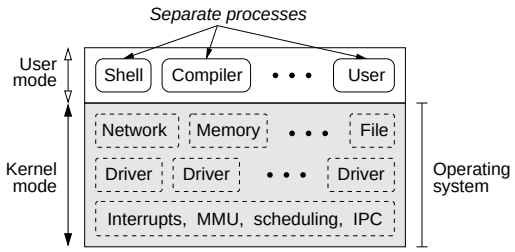

Figure 1: Structure of a monolithic system. The entire operating system runs in kernel mode without proper fault isolation.

follows from the observation that the majority of operating system crashes are caused by device drivers [7, 25]. Yet another problem is that the immense size of monolithic kernels makes them very complex and hard to fully understand. Without a global understanding of the kernel even the most experienced programmers can easily introduce faults by not being aware of some peculiar side-effect of their actions.

2.2 Minimal Kernel Systems

At the other extreme is the minimal kernel, which contains only the barest mechanism, but no policy. A minimal kernel provides interrupt handlers, a mechanism for starting and stopping processes (by loading the MMU and CPU registers), 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 minimal kernel. One option is to run the entire operating system in a single user-mode server, but in such a design the same problems as in a monolithic system exist, and bugs can still crash the entire user-mode operating system. In Sec. 6, we will discuss some work in this area.

A better design is to run each untrusted module as a separate user-mode process that is isolated from the others. We took this idea to the extreme and fully compartmentalized our system, as shown in Fig. 2. All operating system functionality, such as device drivers, the file system, the network server and high-level memory management, runs as a separate user process that is encapsulated in a private address space. This model can be characterized as a *multiserver operating system*.

Logically, our user processes can be structured into three layers, although from the kernel's point of view, they are all just processes. The lowest level of user-mode processes are the device drivers, each one controlling some device. We have implemented drivers for IDE, floppy, and RAM disks, keyboards, displays, audio, printers, and various Ethernet cards. Above the driver layer are the server processes. These include the file server, process server, network server, information server, reincarnation server, and others. On top of the servers come the ordinary user processes, including shells, compilers, utilities, and application programs. With a small number of minor exceptions, the servers and drivers are normal user processes.

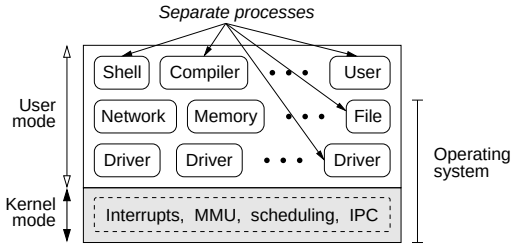

Figure 2: Structure of our system. The operating system runs as a collection of isolated user-mode processes on top of a tiny kernel.

To avoid any ambiguity, we note again that each server and driver runs as a *separate* user process with its own address space completely disjoint from the address spaces of the kernel and other servers, drivers, and user processes. In our design, processes do not share any virtual address space and can only communicate with each other using the IPC mechanisms that are provided by the kernel. This point is crucial to the reliability 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.

In user mode, the operating system processes are restricted in what they can do. Therefore, to support the servers and drivers in performing their tasks, the kernel exports a number of kernel calls that authorized processes can make. Device drivers, for example, no longer have privileges to perform I/O directly, but can request the kernel to do the work on their behalf. In addition, servers and drivers can request services from each other. All such IPC is done by exchanging small, fixed-size messages. This message passing is implemented as traps to the kernel, which checks the call being requested to see if the caller is authorized to make it.

Now let us consider a typical kernel call. An operating system component running as a user-mode process may need to copy data to/from another address space, but cannot be trusted to be given access to raw memory. Instead, kernel calls are provided to copy to/from valid virtual addresses within the data segment of a target process. This is a far weaker call than giving it the ability to write to any word in memory, but it still has considerable power, so its use is restricted to operating system processes that need to do interaddress space block copies. Ordinary user processes are prohibited from using it.

Given this structure, we can now explain how user processes obtain the operating system services defined by the POSIX standard. A user process wanting to make, say, a READ call builds a message giving the system call number and (pointers to) the parameters and executes the kernel trap to send the little request message to the file server, another user process. The kernel makes sure that the caller is blocked until the request has been served by the file server. By default all communication between processes is denied for security reasons, but this request succeeds because communication with the file server is explicitly allowed to ordinary user processes.

If the file server has the requested data in its buffer cache, it makes a kernel call asking the kernel to copy it to the user's buffer. If the file server does not have the data, it sends a message to the disk driver requesting the necessary block. The disk driver then commands the disk to read the block directly to the address within the file server's buffer cache. When the disk transfer is complete, the disk driver sends a reply message back to the file server giving it the status of the request (success or the reason for failure). The file server then makes a kernel call asking the kernel to copy the block to the user's address space.

This scheme is simple and elegant and separates the servers and drivers from the kernel so they can be replaced easily, making for a modular system. Although up to four messages are required here, these are very fast (under 500 ns each, depending on the CPU). When both the sender and receiver are ready, the kernel copies the message directly from the sender's buffer to the receiver without first putting it into its own address space. Furthermore, the number of times the data are copied is precisely the same as in a monolithic system: the disk puts the data directly in the file server's buffer cache and there is one copy from there to the user process.

2.3 Design Principles

Before we move on to a detailed treatment of the reliability features of our system we will now briefly discuss the design principles that guided our quest for reliability:

1. Simplicity.
2. Modularity.
3. Least authorization.
4. Fault tolerance.

First, we have kept our system as simple as possible so that it is easy to understand and thus more likely to be correct. This concerns both the high-level design and implementation. Our design structurally avoids known problems such as resource exhaustion. If needed we explicitly trade resources and performance for reliability. The kernel, for example, statically declares all data structures instead of dynamically allocating memory when needed. While this may waste some memory, it is much simpler to manage and can never fail. As another example, we have deliberately not implemented multithreading. This may (or may not) cost some performance, but not having to care about potential race conditions and thread synchronization makes life much easier for the programmer.

Second, we have split our system into a collection of small, independent modules. Exploiting modularity properties, such as fault containment, is a crucial element of the design of our system. By fully compartmentalizing the operating system we can establish 'firewalls' across which errors cannot propagate, thus resulting in a more robust system. To prevent failures in one module from *indirectly* affecting another module, we structurally reduced interdependencies as much as possible. When this was not possible due to the nature of the modules we employed additional safety measures. The file system, for example, depends on the device drivers, but is designed in such a way that it is prepared to handle a driver failure.

Third, we enforce the principle of least authorization. While isolation of faults helps to keep faults from spreading, a fault in a powerful module still might cause substantial damage. Therefore, we have reduced the privileges of all user processes, including the servers and drivers, as far as we could. The kernel maintains several bit maps and lists that govern who can do what. These include, for example, the allowed kernel call map and the list of permitted message destinations. This information is centralized in each process' process table entry, so it can be tightly controlled and managed easily. It is initialized at boot time largely from configuration tables created by the system administrator.

Fourth, we have explicitly designed our system to withstand certain failures. All servers and drivers are managed and monitored by a special server known as the *reincarnation server* that can handle two kinds of problems. If a system process unexpectedly exits this is immediately detected and the process will be restarted directly. In addition, the status of each system process is periodically checked to see if it is still functioning properly. If not, the malfunctioning server or driver is killed and restarted. This is how fault tolerance works: a fault is detected, the faulty component is replaced, but the system continues running the entire time.

3. RELIABILITY FEATURES

We believe that our design improves system reliability over all current operating systems in three important ways:

1. It reduces the number of critical faults.
2. It limits the damage each bug can do.
3. It can recover from common failures.

In the following subsections we will explain why. We will also compare how certain classes of bugs affect our system versus how they affect monolithic systems such as Windows, Linux, and BSD. In Sec. 6, we will compare our approach to reliability to other ideas proposed in the literature.

3.1 Reducing the Number of Kernel Bugs

Our first line of defense is a very small kernel. It is well understood that more code means more bugs, so having a small kernel means fewer kernel bugs. Using an estimate of 6 bugs per 1000 lines of executable code as the lower limit [27], with 3800 lines of executable code are probably at least 22 bugs in the kernel. Furthermore, 3800 lines of code (under 100 pages of listing, including headers and comments) is sufficiently small that a single person can understand all of it, greatly enhancing the chance that in the course of time all the bugs can be found.

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 6 x 2500 = 15,000 bugs. Moreover, with multimillion-line systems, no person will ever read the complete source code and fully understand it working, thus reducing the chance that all the bugs will ever be found.

3.2 Reducing Bug Power

Of course, reducing the size of the kernel does not reduce the amount of code present. It just shifts most of it to user mode. However, this change itself has a profound effect on reliability. Kernel code has complete access to everything the machine can do. Kernel bugs can accidentally trigger I/O or do the wrong I/O or interfere with the memory map or many other things that unprivileged user-mode programs cannot do.

Thus we are not arguing that moving most of the operating system to user mode reduces the total number of bugs present. We *are* arguing that when a bug is triggered, the effects will be less devastating by converting it from a kernel-mode bug to a user-mode bug. 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.

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 in a monolithic design. It might land on the memory management code and start corrupting key data structures such as page tables and memory-hole lists. Monolithic systems are very brittle in this respect and tend to collapse when a bug is triggered.

3.3 Recovering from Failures

Servers and drivers are started and guarded by a system process called the *reincarnation server*. If a guarded process unexpectedly exits or crashes this is immediately detected—because the process server notifies the reincarnation server whenever a server or driver terminates—and the process is automatically restarted. Furthermore, the reincarnation server periodically polls all servers and drivers for their status. If one does not respond correctly within a specified time interval, the reincarnation server kills and restarts the misbehaving server or driver. Since most I/O bugs tend to be transient, due to rare timing, deadlocks, and the like, in many cases just restarting the driver will cure the problem.

A driver failure also has consequences for the file system. Outstanding I/O requests might be lost, and in some cases an I/O error must be reported back to the application. In many cases, full recovery is possible, though. A more detailed discussion of the reincarnation server and application-level reliability is given in Sec. 4.

Monolithic systems do not usually have have the ability to detect faulty drivers on the fly like this, although some recent work in this area has been reported [25]. Nevertheless, replacing a kernel driver on the fly is tricky since it can hold kernel locks or be in a critical section at the time of replacement.

3.4 Limiting Buffer Overruns

Buffer overruns are a famously rich source of errors that are heavily exploited by viruses and worms. While our design is intended to cope with bugs rather than malicious code, some features in our system offer protection against certain types of exploits. Since our kernel is minimal and uses only static allocation of data, the problem is unlikely to occur in the most sensitive part of the system. If a buffer overrun occurs in one of the user processes, the problem is not as severe because the user-mode servers and drivers are limited in what they can do.

Furthermore, our system only allows execution of code that is located in a read-only text segment. While this does not prevent buffer overruns from occurring, it makes it harder to exploit them because excess data that ends up on the stack or heap cannot be executed. This defense mechanism is extremely important since it prevents viruses and worms from injecting and executing their own code. The worst case scenario changes from giving direct control to overwriting the return address on the stack and executing an existing library procedure of choice. The most well-known example is often referred to as a 'return-to-libc' attack and is considered significantly more complex than execution of code on the stack or heap.

On monolithic systems, in contrast, root privileges are gained if the buffer overrun occurs in any part of the operating system. Moreover, many monolithic systems allow execution of code on the stack or heap, making it much easier to exploit buffers overruns.

3.5 Ensuring Reliable IPC

A well-known problem with message passing is buffer management, but we avoid this problem altogether in our design of our communication primitives. Our synchronous message passing mechanism uses rendezvous, which eliminates the need for buffering and buffer management, and cannot suffer from resource exhaustion. If the receiver is not waiting, a SEND blocks the caller. Similarly, a RECEIVE blocks if there is no message pending for it. This means that for a given process only a single pointer to a message buffer has to be stored at any time in the process table.

In addition, we have an asynchronous message passing mechanism, NOTIFY, that also is not susceptible to resource exhaustion. Notification messages are typed and for each process only one bit per type is saved. All pending notifications thus can be stored in a compact bit map that is statically declared as part of the process table. Although the amount of information that can be passed this way is limited, this design was chosen for its reliability.

As an aside, we avoid buffer overruns in our IPC by restricting all communication to short fixed-length messages. The message is a union of several typed message formats, so the size is automatically chosen by the compiler as the largest of the valid message types, which depends on the size of integers and pointers. All requests and replies use this message passing mechanism.

3.6 Restricting IPC

IPC is a powerful mechanism that must be tightly controlled. Since our rendezvous message passing mechanism is synchronous, a process performing IPC is blocked until both parties are ready. User process could easily misuse this property to hang system processes by sending a request and not waiting for the response. Therefore, another IPC primitive, SENDREC, exists, which combines a SEND and a RECEIVE in a single call. It blocks the caller until the reply to a request has been received. To protect the operating system, this is the only primitive that can be used by ordinary users. In fact, the kernel maintains a bit map per process to restrict the IPC primitives each is allowed to use.

Furthermore, the kernel maintains a bit map telling with which drivers and servers a process may communicate. This send mask is the mechanism by which user processes are prevented from sending messages to drivers directly. Instead, they are restricted to only communicating with the servers that provide the POSIX calls. However, the send mask mechanism is also used to prevent, say, the keyboard driver from sending an (unexpected) message to the sound driver. Again, by tightly encapsulating what each process can do, we can largely prevent the inevitable bugs in drivers from spreading and affecting other parts of the system.

In contrast, in a monolithic system, any driver can call any piece of code in the kernel using the machine's subroutine call instruction (or worse yet, the subroutine return instruction when the stack has been overwritten by a buffer overrun), letting problems within one subsystem propagate to other subsystems.

3.7 Avoiding Deadlocks

Because the default mode of IPC are synchronous SEND and RECEIVE calls, deadlocks can occur when two or more processes simultaneously try to communicate and all processes are blocked waiting for one another. Therefore, we carefully devised a *deadlock avoidance* protocol that prescribes a partial, top-down message ordering.

The message ordering roughly follows the layering that is described in Sec. 2.2. Ordinary user processes, for example, are only allowed to SENDREC to the servers that implement the POSIX interface, which can request services from the drivers, which in turn can call upon the kernel. However, for asynchronous events such as interrupts and timers, messages are required in the opposite direction, from the kernel to a server or a driver. Using synchronous SEND calls to communicate these events can easily lead to a deadlock. We avoid this problem by using the NOTIFY mechanism, which never blocks the caller, for asynchronous events. If a notification message cannot be delivered, it is stored in the destination's process table entry until it does a RECEIVE.

Although the deadlock avoidance protocol is enforced by the send mask mechanism discussed above, we also implemented *deadlock detection* in the kernel. If a process unexpectedly were to cause a deadlock, the offending is denied and an error message is returned to the caller.

3.8 Unifying Interrupts and Messages

The basic IPC mechanism is rendezvous message passing, but asynchronous messages are also needed, for example, for reporting interrupts, a potential source of bugs in operating systems. We have greatly reduced the chance of errors here by unifying asynchronous signaling and messages. Normally when a process sends a message to another process and the receiver is not ready, the sender is blocked. This scheme does not work for interrupts because the interrupt handler cannot afford to block. Instead the asynchronous notification mechanism is used, in which a handler issues a NOTIFY call to a driver. If the driver is waiting for a message, the notification is directly delivered. If it is not waiting, the notification is saved in a bitmap until the driver subsequently blocks on a RECEIVE call.

3.9 Restricting Driver Functionality

The kernel exports a limited set of functions that can be called. This kernel API is the only way a driver can interact with the kernel. However, not every driver is allowed to use every kernel call. The kernel maintains a bit map per driver (in the process table), telling which kernel calls that driver may make. The granularity of kernel calls is quite fine. There is no multiplexing of calls onto the same function number. Each call is individually protected with its own bit in the bit map. Internally, several calls may be handled by the same kernel function, though. This technique allows a fine grain of control.

For example, some drivers need to read and write user address spaces to move data in and out, but the calls for reading and writing user address spaces are different. Thus we do not multiplex read and write onto the same call with a 'direction' parameter. Accordingly, it is possible to give a printer driver, for example, the ability to make the kernel call to read from user processes but not the ability to write to them. As a consequence, a bug in a driver with read-only permission cannot accidentally corrupt user address spaces.

Contrast this situation with a device driver in a monolithic kernel. A bug in the code could cause it to write to the address space in a user process instead of reading from it, destroying the process. In addition, a kernel driver can call any function in the entire kernel, including functions no driver should ever call. Since there is no intrakernel protection, it is virtually impossible to prevent this. In our design, no driver can ever call a kernel function that has not been explicitly exported as part of the kernel-driver interface.

3.10 Denying Access to I/O Ports

For each driver, the kernel maintains a list of which I/O ports it may read and which ones it may write. Read and write access are protected separately, so a process that has read-only permission to some I/O port cannot write to it. Any attempt to violate these rules will result in an error code returned to the caller. In this way, a printer driver can be restricted to the I/O ports for the printer, a sound driver can be restricted to the I/O ports for the sound card, etc.

In contrast, there is no way in monolithic systems to restrict an in-kernel driver to only a handful of I/O ports. A kernel driver can accidentally write to any I/O port and cause substantial damage

In some cases the actual I/O device registers can be mapped into the driver's address space to avoid any interaction with the kernel when doing I/O. However, since not all architectures allow to map I/O registers into user processes in a fine-grained protected way, we have chosen for a model in which only the kernel performs actual I/O operations. This design decision is yet another example of how we have opted for reliability over performance.

While the tables with allowed I/O ports are currently initialized from a configuration file, we plan to implement a PCI bus server to do this automatically. The PCI bus server can obtain the I/O ports required by each driver from the BIOS and use this information to initialize the kernel tables.

3.11 Parameter Checking

Since all kernel calls are traps to the kernel, it is possible for the kernel to do a limited amount of parameter validation before dispatching the call. This validation includes both *sanity* and *permission checks*. For example, if a driver asks the kernel to write a block of data using physical addressing, the call may be denied because not all drivers have permission to do so. Using virtual addressing, the kernel can probably not tell if this is the right address to write, but it can at least check that the address is, in fact, a valid address within the user's data or stack segments, and not in the text segment or some random invalid address.

While such sanity checks are coarse, they are better than nothing. In a monolithic system, nothing prevents a driver from writing to addresses that should not be written to under any conditions, such as the kernel's text segment.

3.12 Catching Bad Pointers

C and C++ programs use pointers a great deal and tend to suffer from bad pointer errors all the time. Dereferencing a bad pointer often leads to a segmentation fault detected by the hardware. In our design, a server or driver dereferencing a bad pointer will be killed and given a core dump for future debugging, just like any other user process. If a bad pointer is caught in a part of the user-mode operating system, the reincarnation server will immediately notice the failure and replace the killed process with a fresh copy.

3.13 Taming Infinite Loops

When a driver gets stuck in an infinite loop, it threatens to consume endless CPU time. The scheduler notices this behavior and gradually drops the offending process' priority until it becomes the idle process. Other processes can continue to run normally however. After a predetermined time the reincarnation server will notice that the driver is not responding and will kill and restart it.

In contrast, when a kernel driver gets into an infinite loop, it consumes all CPU time and effectively hangs the entire system.

3.14 DMA Checking

One thing that we cannot do is prevent bad DMA from wreaking havoc on the system. Hardware protection is required to prevent a driver from overwriting any part of real memory. However, we can catch some DMA errors as follows. DMA is normally started by writing the DMA address to some I/O port. We can give the library procedure that is called to write to an I/O port a (device-specific) way to decode I/O port writes to find the DMA addresses that are used and check them for validity. A malicious driver could bypass this checking, but for catching programming errors (as opposed to hostile drivers), it is a cost-efficient way to weed out at least some bugs.

Depending on the hardware we can do even better. If the peripheral bus has an I/O MMU we might be able to precisely restrict memory access on a per-driver basis [16]. For systems with a PCI-X bus we intend to make our PCI bus server responsible for initializing the tables of the I/O MMU. This is part of our future work.

4. RELIABILITY ANALYSIS

To test the reliability of the system, we manually injected some carefully selected faults into some of our servers and drivers to see what would happen. As described in Sec. 3.3, our system is designed to detect and correct many failures, and this is precisely what we observed. If a component failed for whichever reason this was detected by the reincarnation server, which took all needed measures to revive the failing component. This is described in more detail below.

To appreciate the working of our design, two classes of errors have to be distinguished. First, *logical errors* mean that a server or driver adheres to the intermodule protocol and normally responds to requests as if it successfully did the work, while, in fact, it did not. An example is a printer driver that prints garbage but returns normally. It is very hard if not impossible for *any* system to catch this kind of errors. Logical errors are outside the scope of this research.

A second class of errors are *protocol errors* where the rules determining how the servers and drivers should behave are violated. In our system, servers and drivers, for example, are required to respond to the periodic status requests from the reincarnation server. If they do not obey this rule, corrective action will be taken. Our system is designed to deal with protocol errors.

4.1 The Reincarnation Server

The reincarnation server is the central server that manages all servers and drivers of the operating system. It greatly enhances the reliability by offering:

1. Immediate crash detection.
2. Periodic status monitoring.

It thus helps to catch two common failures: dead or misbehaving system processes, and immediately tackles the worst problem. Whenever a system process exits, the reincarnation server is directly notified and checks its tables to see if the service should be restarted. This mechanism, for example, ensures that a driver that is killed because of a bad pointer is replaced instantaneously. In addition, periodic status monitoring helps to discipline misbehaving system services. A driver that, for example, winds up in an infinite loop and fails to respond to a status request from the reincarnation server will be killed and restarted.

Replacing a device driver is a tightly controlled sequence of events. First, the reincarnation server spawns a new process, which is inhibited from running because its privileges are not yet assigned. The reincarnation server then tells the file system about the new driver and, finally, assigns the required privileges. When all of these steps have succeeded, the new process starts running and executes the driver's binary from the file system. As an extra precaution, the binary of certain drivers can be shadowed in RAM, so that, for example, the driver for the root file system disk can be reloaded without requiring disk I/O.

4.2 Application-Level Reliability

A failing driver might have implications for the file system and applications doing I/O. If the file system had an outstanding I/O request, an error will be returned telling that the driver failed. At this point different actions may be taken. A distinction must be made between block devices and character devices, because I/O for the former is buffered in the file systems's buffer cache. Fig. 3 gives an overview of different recovery scenarios at the application level.

Driver	Type	Recovery	How
Hard disk	Block	Full	Flush FS cache
RAM disk	Block	Full	Flush FS cache
Floppy	Block	Full	Flush FS cache
Printer	Character	Partial	Reissue print job
Ethernet	Character	Full	Transport layer
Sound	Character	Partial	Jitter

Figure 3: Different application-level recovery scenarios for various types of failing device drivers.

A crash of a block device driver allows for full recovery transparent to the application and without loss of data. When the failure is detected, the reincarnation server starts a fresh copy of the sick driver and flushes the file system's cache to synchronize. The buffer cache thus not only improves performance, but is also crucial for reliability.

When a character device driver fails transparent recovery is sometimes possible. Since the I/O request is not buffered in the file system's block cache, an I/O error must be reported to the application. If the application is not able to recover, the user will be notified about the problem. Effectively, driver failures are pushed up, which leads to different recovery scenarios. For example, if an Ethernet driver fails the networking server will notice the missing packets and recover transparently when the application uses a reliable transport protocol, such as TCP. On the other hand, if a printer driver fails, the user will no doubt notice that his printout failed and reissue the print command.

In many cases, our system thus can provide *full* recovery at the application level. In the remaining cases, the I/O failure is pushed back to the user. It might be possible to alleviate this nuisance by using a shadow driver to recover applications that were using a faulty driver when it crashed, by applying the same techniques as demonstrated in [25]. Lack of manpower prevented us from doing this.

4.3 Reliability Test Results

To verify the reliability of our system we manually injected faults in some of our drivers to test specific kinds of errors and observed what happened. In the simplest case, we killed a driver with a SIGKILL signal. More severe test cases caused drivers to dereference a bad pointer or loop forever. In all cases, the reincarnation server detected the problem and replaced the malfunctioning driver with a fresh copy. The test results are shown in Fig. 4.

While testing the reliability we learned several lessons that are important for the design of our system. First, since the reincarnation server restarts bad servers and drivers, it is required that they remain stateless so that they can properly reinitialize when they are brought up again. Stateful components, such as the file system and the process server, cannot be cured this way because they loose too much state on a restart. There are limits to what we can do.

Cause	Effect	Action	Recovery
Bad pointer	Killed	Restart	OK
Infinite loop	Hung	Kill & Restart	OK
Panic	Exit	Restart	OK
Kill signal	Killed	Restart	OK

Figure 4: Test results for serious failures in device drivers. If the reincarnation server detects a problem it automatically takes corrective measures.

Another insight is that some drivers were implemented such that initialization only happened on the first OPEN call. However, for transparent application-level recovery from a driver failure an application should not be required to call OPEN again. Instead, issuing a READ or WRITE call to a revived driver should trigger the driver to reinitialize.

Furthermore, while we anticipated dependencies between the file system and the drivers, our tests revealed some other interdependencies. Our information server that displays debugging dumps when a function key is pressed, for example, lost its key mappings after a restart. As a general rule, dependencies must be prevented and all components should be prepared to deal with unexpected failures.

Finally, user-space applications should be changed as well to further enhance the reliability. For historical reasons, most applications assume that a driver failure is fatal and immediately give up, while recovery sometimes is possible. Printing is example where application-level recovery is possible. If a line printer daemon is notified about a temporary driver failure, it can automatically reissue the print command without user intervention. Further experimentation with application-level recovery is part of our future work.

5. PERFORMANCE MEASUREMENTS

The issue that has haunted minimal kernels for decades is performance. Therefore, the question immediately arises: how much do the changes discussed above cost? To find out, we built a prototype consisting of a small kernel beneath a collection of user-mode device drivers and servers. As a base for the prototype, we started with the MINIX 2 system due to its very small size and long history. The code has been studied by many tens of thousands of students at hundreds of universities for a period of 18 years with almost no bug reports concerning the kernel in the last 10 years, presumably due to its small size and thus lack of bugs. We then heavily modified the code, removing the device drivers from the kernel and adding the reliability features discussed in Sec. 3. In this way, we arrived at what is effectively a new system, called MINIX 3, without having to write large amounts of code not relevant to this project such as drivers and the file system.

Since we are interested in isolating the cost of the changes discussed in this paper, we compare our system with the base system in which the device drivers are part of the kernel by running the same tests on both of them. This is a much purer test than comparing the system to Linux or Windows, which would be like comparing apples to pineapples. Such comparisons would be plagued by differences in compiler quality, memory management strategy, file systems, amount of optimization performed, system maturity, and many other factors that would completely overshadow everything else.

The test system was a 2.2 GHz Athlon (specifically, an AMD64 3200+) with 1 GB of RAM and a 40-GB IDE disk. None of the drivers have been optimized for user-mode operation yet. For example, on the Pentium, we expect to be able to give drivers direct access to the I/O ports they need in a protected way, thus eliminating many kernel calls. To maintain portability the interface will not be changed, though. Also, the drivers currently use programmed I/O, which is much slower than using DMA. We expect substantial improvement when these optimizations are implemented. Nevertheless, even with the current system, the performance penalty turns out to be very reasonable.

5.1 System Call Test Results

The first batch of tests contained pure POSIX system call tests. A user program was programmed to record the real time in units of clock ticks (at 60 Hz), then make a system call millions of times, then record the real time again. The system call time was computed as the difference between the end and start time divided by the number of calls, minus the loop overhead, which was measured separately. The number of loop iterations was different for each test because testing getpid 100 million times was reasonable but reading a 64-MB file 100 million times would have taken too long. All tests were made on an idle system. For these tests, both the CPU cache hit rate and the file server buffer cache hit rate were presumably 100%. The results are shown in Fig. 5.

Call	Kernel	User	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 5: System call times for kernel-mode versus user-mode drivers. All times are in microseconds.

Let us briefly examine the results of these tests. The getpid system call takes 0.831 μs with kernel mode drivers and 1.011 μs with user-mode drivers. This call consists of a simple message from the user process to the memory manager and an immediate answer. It is slower with user drivers due to the increased checking of who is allowed to send to whom. With such a simple call, even a few additional lines of code slows it down measureably. While the percent difference is 22%, we are only talking about 180 ns per call, so even with 10,000 calls/sec the loss is only 2.2 ms/second, well under 1%. The lseek call has more work to do, so the relative overhead drops to 11%. For opening and closing a file, the user-driver version is 9% worse.

Reading and writing of 64-KB data chunks takes just under 90 μs and has a performance penalty of 8%. Creating

a file, writing 1 KB to it, and then deleting it takes 13.465 μs with user-mode drivers. Due to the file server's buffer cache, none of these tests involve driver calls, so we can conclude that the other nondriver-related changes slow the system down by about 12%.

5.2 Disk I/O Test Results

For the second batch of tests, 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. That is why there is such a large jump from 4 MB (reading from and writing to the cache) and 64 MB, where the file did not fit. The disk controller's internal cache was not disabled. The results are shown in Fig. 6.

File reads	2.0.4	3.0.1	Δ	Ratio
1 KB	2.619	2.904	0.285	1.11
16 KB	18.891	20.728	1.837	1.10
256 KB	325.507	351.636	26.129	1.08
4 MB	6962.240	7363.498	401.258	1.06
64 MB	16.907	17.749	0.841	1.05
Average				**1.08**

File writes	2.0.4	3.0.1	Δ	Ratio
1 KB	2.547	3.004	0.457	1.18
16 KB	18.593	20.609	2.016	1.11
256 KB	320.960	345.696	24.736	1.08
4 MB	8376.329	8747.723	371.394	1.04
64 MB	18.789	19.294	0.505	1.03
Average				**1.09**

Figure 6: Times to read and write chunks of a large file. Times are in microseconds except for the 64 MB operations, where they are in seconds.

The result is that we see a performance hit ranging from 3% to 18% with an average of 8.4%. However, note that the worst performance is for 1-KB writes, but the absolute time increase is only 457 ns. The ratio decreases when more I/O is done because the relative overhead decreases. On the three 64-MB tests in Figs. 6 and 7, it is only 3% to 5%.

Another test reads the raw block device corresponding to the hard disk. Writing to the raw device would destroy its contents, so that test was not performed. The results are shown in Fig. 7. These tests bypass the file system's buffer cache and just test moving bits from the disk. Here we see an average overhead of just 9%.

Raw reads	2.0.4	3.0.1	Δ	Ratio
1 KB	2.602	2.965	0.363	1.14
16 KB	17.907	19.968	2.061	1.12
256 KB	303.749	332.246	28.497	1.09
4 MB	6184.568	6625.107	440.539	1.07
64 MB	16.729	17.599	0.870	1.05
Average				**1.09**

Figure 7: Times to read the raw disk block device. Times are in microseconds except for the 64 MB operations, where they are in seconds.

5.3 Application Test Results

The next set of tests were actual programs rather than pure measures of system call times. The results are given in Fig. 8. The first test consisted of building a boot image, in a loop containing system("make image") to run the build many times. The C compiler was called 123 times, the assembler 4 times, and the linker 11 times. Building the kernel, drivers, servers and the *init* program, and assembling the boot image took 3.878 seconds. The mean compilation time was under 32 ms per file.

Program	2.0.4	3.0.1	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 8: Run times in seconds for various test programs. The first two test were repeatedly run a loop, while the others were run only once to exclude effects from the file system's cache.

The second test consisted of a loop compiling the POSIX-conformance test suite repeatedly. The suite of 42 test programs compiled in 1.577 seconds, or about 37 ms per test file. The third through seventh tests consisted of sorting, sedding, grepping, prepping, and uuencoding a 64-MB file, respectively. These tests mix computation with disk I/O in varying amounts. Each test was run only once, so the file server's cache was effectively useless; every block came from the disk. The average performance hit here was 6%, similar to the final lines in Figs. 6 and 7.

If we average the last column of the 22 tests reported in Figs. 6 through 8, we get 1.08. In other words the version with user-mode drivers is about 8% slower than with kernel-mode drivers for operations involving disk I/O.

5.4 Networking Performance

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 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.

5.5 Code Size

Speed is not the only metric of interest; the number of bugs is also very important. Unfortunately, we were not able to enumerate all the bugs directly, but the number of lines of code is probably a reasonable proxy for the number of bugs. Remember: the more code, the more bugs.

Counting lines of code is not as straightforward as it might at first appear. First, blank lines and comments do not add to the code complexity so we omitted them. Second, #define and other definitions in header files do not add to the code complexity, so we omitted the header files as well. Line counting was done by the *sclc.pl* Perl script available on the Internet. The results for the kernel, four servers (file system, process server, reincarnation server, information server), five drivers (hard disk, floppy disk, RAM disk, terminal, log device), and the *init* program are given in Fig. 9.

Part	#	C	Asm	;	Binary
Init	1	327	0	193	7088
File	25	4648	0	2698	43,056
Process	13	2242	0	1308	20,208
Reinc.	28	519	0	278	6368
Info	6	783	0	457	13,808
Hard disk	1	1192	0	653	24,384
Floppy	1	770	0	435	10,448
RAM disk	1	237	0	116	4992
Terminal	19	5161	120	2120	26,352
Log device	4	430	0	235	6048
Kernel	45	2947	778	1729	21,312
Total	**127**	**18,009**	**898**	**10,363**	**173,844**

Figure 9: MINIX 3 code size statistics. For each part the number of files, the number of lines of C and assembler code, the number of semicolons, and the binary size of the text segment in bytes is given.

From the figure it can be seen that the kernel consists of 2947 lines of C and 778 lines of assembler (for low-level functionality, such as catching interrupts and saving the CPU registers on a process switch). The total is 3725 lines of code. This is the *only* code that runs in kernel mode. Another way to measure code size for C programs is to count the number of semicolons, since many C statements are terminated by a semicolon. The number of semicolons present in the code for the kernel is 1729. Finally, the compiled size of the kernel is 21,312 bytes. This number is just the code (i.e., text segment) size. Initialized data (3800 bytes) and stack are not included in this number.

Interestingly, the code size statistics shown in Fig. 9 represent a minimal yet functional operating system. The total size for both the kernel part and the user-mode part, is just over 18,000 lines of code, remarkably small for a POSIX-conformant operating system. We will compare these numbers to other systems in Sec. 6.

6. RELATED WORK

We are not the first researchers trying to prevent buggy device drivers from causing system crashes. Nor are we the first to examine minimal kernels as a possible solution. We are not even the first to implement user-space device drivers. Nevertheless, we believe we are the first to build a fully POSIX-conformant multiserver operating system with excellent fault isolation properties on top of a 3800-line minimal kernel with each driver running as a separate user-mode process and the OS itself running as multiple separate user-mode processes. In this section we will discuss work by other research groups that is similar in part to what we have done.

6.1 Wrapping Drivers in Software

One important research project that attempts to build a reliable system in the presence of unreliable device drivers is Nooks [26]. The goal of Nooks is to improve the reliability of *current* operating systems. In the words of the authors: "we target existing extensions for commodity operating systems rather than propose a new extention architecture. We want today's extensions to execute on today's platforms without change if possible." The idea is to be backward compatible with existing systems, but small changes are permitted.

The Nooks approach is to keep device drivers in the kernel but to enclose them in a kind of lightweight protective wrapper so that driver bugs cannot propagate to other parts of the operating system. Nooks works by transparently interposing a reliability layer between the device driver being wrapped and the rest of the operating system. All control and data traffic between the driver and the rest of the kernel is inspected by the reliability layer. When the driver is started, the reliability layer modifies the kernel's page map to turn off write access to pages that are not part of the driver, thus preventing it from directly modifying them. To support legitimate write access to kernel data structures, Nooks copies needed data into the driver, and copies them back after modification.

Our goal is completely different than that of Nooks. We are not attempting to make legacy operating systems more reliable. As researchers, we ask the question: How should *future* operating systems be designed to prevent the problem in the first place? We do believe the right design for future systems is to build a multiserver operating system and run untrustworthy code in independent user-mode processes, where it can do far less damage, as discussed in Sec. 3.

Despite different goals there are also technical points on which the systems can be compared. Consider just a few examples. Nooks cannot handle byzantine faults such as a driver inadvertently changing the page map; in our system, drivers have no access to the page map. Nooks cannot handle infinite loops; we can because when a driver fails to respond correctly to the reincarnation server, it is killed and a fresh copy started. While in practice Nooks can handle wild stores into kernel data structures most of the time, in our design, such stores are structurally impossible. Nooks cannot handle a printer driver that accidentally tries to write to the I/O ports that control the disk; we catch 100% of such attempts. Also worth mentioning is the code size. Nooks is 22,000 lines of code, almost six times the size of our entire kernel and larger than a minimal configuration of our complete operating system. It is difficult to get away from this fundamental truth: more code means more bugs. Thus, statistically, Nooks itself probably contains five times as many bugs as our entire kernel.

6.2 Wrapping Drivers with Virtual Machines

Another project that encapsulates drivers does this using the concept of a virtual machine to isolate them from the rest of the system [19, 18]. 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. Like Nooks, this approach is entirely focused on running legacy drivers for legacy operating systems. The authors make no

claim that for new designs it is a good idea to put untrusted code in the kernel and then protect each driver by running it on a separate virtual machine.

While this approach does accomplish what it was designed for, it does have some problems. First, there are issues with how much the main system and the driver's virtual machine trust each other. Second, running a driver on a virtual machine raises timing and locking issues because all the virtual machines are timeshared, and a kernel driver that was designed to run to completion without interruption may be unexpectedly timesliced with unintended consequences. Third, some resources, such as the PCI bus configuration space, may need to be shared among multiple virtual machines. Fourth, the virtual machine machinery consumes extra resources, although the amount is comparable to what our scheme costs: 3% to 8%. While solutions have been proposed to these problems, the approach is at best cumbersome and mainly suited for protecting legacy drivers in legacy operating systems rather than being used in new designs, which is the problem we are addressing.

6.3 Language-Based Safety Measures

In previous work, one of the authors also addressed the problem of safe execution of foreign code into the kernel. The Open Kernel Environment (OKE) provides a safe, resource-controlled environment which allows fully optimized native code to be loaded in the kernel of the Linux operating system [4]. The code is compiled with a customized Cyclone compiler that adds instrumentation to the object code according to a policy that corresponds to the user's privileges. Cyclone, like Java, is a type-safe language, in which most pointer errors are prevented by the language definition. Explicit trust management and authorization control make sure that administrators are able to exercise strict control over which parties are given which privileges, and this control is automatically enforced on their code. Besides the authorization, a central role is played by the compiler which checks that code conforms to the agreed policy by both static checks and dynamic instrumentation.

The OKE allows the foreign module to interact intensively with the rest of the kernel, e.g., by sharing kernel memory. The runtime makes sure of crucial safety measures, such as that data is always garbage collected and no dereferencing of pointers to freed memory can ever happen. Moreover, the OKE is able to enforce strict control over all of the foreign kernel modules resource's: CPU time, heap, stack, entry points, etc.

The OKE was developed with an eye on writing device drivers and kernel extensions. However, because of the strict access control procedures and the complex measures required to make programming in the Linux kernel safe, it was fairly difficult to use. According to the authors, the main cause is the organization of Linux that is simply not designed to allow for safe extension.

6.4 Virtual Machines and Exokernels

Classical virtual machines [24] are a powerful tool for running multiple operating systems at the same time. Exokernels [10] are similar to virtual machines, but partition the resources rather than replicating them, leading to greater efficiency. However, neither approach solves the problem posed at the beginning of Sec. 1.3: how do you prevent a bug in a driver from crashing the operating system.

6.5 User-Mode Drivers on Monolithic Kernel

An early project with user-mode drivers was Mach 3.0 [11]. It consisted of the Mach microkernel on top of which ran Berkeley UNIX as a user-mode process along with device drivers, also as user-mode processes. Unfortunately, if a driver crashed, Berkeley UNIX had to be restarted, so little was gained by isolating the drivers. A multiserver operating system that would run on Mach was planned but never fully implemented.

A similar project at the University of New South Wales implemented Linux drivers in user mode for the hard disk and gigabit Ethernet [8]. Disk performance was noticeably better for the kernel driver for block sizes under 32 KB, but identical for block sizes of 32 KB or more. The Ethernet test exhibited so many anomalies, probably related to buffer management, that it was difficult to draw any conclusions.

6.6 Minimal Kernel Designs

While getting the drivers out of the kernel is a big step forward, getting the operating system out of there as well is even better. This is where minimal kernels come in: vastly reducing the number the number of abstractions they implement. Arguably the first minimal kernel was Brinch Hansen's RC4000 system, which dates to the early 1970s [13]. Starting in the mid 1980s, a number of minimal kernels were written, including Amoeba [21], Chorus [5], Mach [1], and V [6]. None of these practiced safe software, however: they all had unwrapped device drivers inside the kernel.

QNX is a closed-source, commercial UNIX-like real-time system [17]. Although it has a minimal kernel, called Neutrino, little has been published about the system and precise details are unknown to us. However, from recent information sheets we conclude that Neutrino is a hybrid kernel as the process manager shares its address space with the kernel.

Starting in the early 1990s, the late Jochen Liedtke wrote a minimal kernel called L4 in assembly code for the x86 architecture. It quickly became clear that it was not portable and hard to maintain, so he rewrote it in C [20]. It has continued to evolve since then. There are currently two main branches: L4/Fiasco, maintained at the Technical University of Dresden and L4Ka::Pistachio, maintained at the University of Karlsruhe and the University of New South Wales. These are written in C++.

The key ideas in L4 are address spaces, threads, and IPC between threads in different address spaces. A user-mode resource manager brought up when the system is booted controls system resources and distributes them among user processes. L4 is one of the few other true minimal kernels in existence, with device drivers running in user mode. However, there is no implementation with each driver in a separate address space and its API is quite different from ours, so we have not run any tests on it.

However, it was straightforward to run the line count script on the current L4Ka::Pistachio kernel. The results are

shown in Fig. 10 and can be compared to the 'Kernel' line in Fig. 9. The source code is about twice the size of our kernel and the memory footprint of the code is six times larger, but its functionality is different, so it is hard to conclude much except that it is considerably bigger.

Part	Files	C++	Asm	;	Binary
Kernel	57	6881	420	3053	114 KB

Figure 10: Code size statistics for L4Ka::Pistachio.

6.7 Single-Server Operating Systems

One of the ways that minimal kernels have been used is to provide a platform on top of which an entire operating system is run as a single server, possibly in user mode. To obtain system services, user programs request them from the operating system process. This design has similar properties to monolithic systems, as discussed in 2. A bug in a driver can still bring down the entire operating system and, as a result, all the application programs. Thus in terms of fault isolation, running the entire operating system in a single user process is no better than running it in kernel mode. The only real gain is that rebooting the user-mode operating system server and all the applications after a crash is faster than rebooting the computer.

One example of this technology is Berkeley UNIX on top of Mach (renamed Darwin by Apple), which is the core of the Apple Mac OS X system [28]. However in this system, UNIX runs in the kernel, which makes it simply a differently-structured monolithic kernel. A second example is MkLinux, in which Linux is run as a single user process on top of Mach. A third example is L4-Linux, in which full Linux is run on top of L4 [15]. In the latter system, user processes obtain operating system services by making remote procedure calls to the Linux server using L4's IPC mechanism. Measurements show the performance penalty over native Linux to be 5% to 10%, quite similar to what we have observed. However, a single line of faulty code in a Linux driver can crash the entire operating system, so the only gain of this design from a reliability point of view is a faster reboot.

6.8 Multiserver Operating Systems

A more sophisticated approach is to split the operating system into pieces and run each one in its own protection domain. One such project was SawMill Linux [12]. However, this project was abruptly terminated in 2001 when many of the principals left IBM.

Another multiserver system is DROPS, which is also built on top of the L4/Fiasco minimal kernel [14]. It is targeted toward multimedia applications. However, most of the device drivers still run as part of a big L4-Linux server process, with only the multimedia subsystem running separately. After some tuning, the performance penalty dropped to the 2% to 4% range.

Nemesis [23] is yet another multiserver operating system with user-mode device drivers. This system had a single address space shared by all processes, but with hardware protection between processes (called domains). Like DROPS, it was aimed at multimedia applications, but it was not POSIX conformant or even UNIX like.

7. CONCLUSIONS

The primary achievement of the work reported here is that we have built a POSIX-conformant operating system based on a minimal kernel whose complete source is under 3800 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 minimal kernel in existence that supports a fully POSIX-conformant multiserver operating system in user mode. It is also the only one that has each device driver running as a separate user-mode process, with many encapsulation facilities, and the ability to reincarnate dead or misbehaving drivers on the fly without rebooting the operating system. We make no claim that we can catch every bug, but we greatly improve the operating system's reliability by structurally eliminating many different classes of bugs.

To achieve maximum reliability our design was guided by simplicity, modularity, the principle of least authorization, and fault tolerance. An understandable and minimal kernel means fewer kernel bugs and thus fewer crashes. Our kernel code, for example, is not subject to the buffer overruns that plague other software, because it statically declares all data instead of using dynamic memory allocation. Furthermore, by moving most of the code (and thus most of the bugs) to unprivileged user-mode processes and restricting the powers of each one, we gain proper fault isolation and limit the damage bugs can do. Moreover, most servers and all device drivers of the operating system are monitored and automatically revived if a problem is detected. For this reduction in operating system crashes, we pay a performance penalty of 5% to 10%. We consider this price well worth paying.

Of course, drivers, file systems, and other components are not magically rendered bug-free by our design. However, with a stable minimal kernel in place, the *worst case* scenario changes from requiring a computer reboot to restarting the operating system in user mode. At the very least, this recovery is much faster. In the best case, if, say, a wild store in the printer driver causes it to crash, the reincarnation server automatically starts a fresh copy of the printer driver. The current print job will have to be redone, but other programs running at the time of the driver crash will not be affected. For block devices the situation is even better. If a disk driver failure is detected, the system can fully recover by transparently replacing the driver and rewriting the blocks from the file system's buffer cache.

Concluding, we have demonstrated how operating system reliability can be increased with an elegant, lightweight approach. Our system can currently withstand most malfunctions caused by bugs. Malicious or hostile servers and drivers pose new challenges, though. Therefore, our research in this area is continuing.

8. ACKNOWLEDGEMENTS

We would like to thank Ben Gras and Philip Homburg for running the test programs, doing some of the programming, and for critically reading the paper. We would also like to thank Jan Looyen, Ruedi Weis, Bruno Crispo, and Carol Conti for their help and feedback.

9. REFERENCES

[1] M. Acceta, R. Baron, W. Bolosky, D. Holub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. 1986 USENIX Summer Tech. Conf.*, pages 93–112, June 1986.

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

[3] B. Bershad. The Increasing Irrelevance of IPC Performance for Microkernel-Based Operating Systems. In *Proc. Usenix Microkernels Workshop*, pages 205–211, Apr. 1992.

[4] H. Bos and B. Samwel. Safe Kernel Programming in the OKE. In *Proc. of the 5th IEEE Conference on Open Architectures and Network Programming*, pages 141–152, June 2002.

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

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] P. B. Hansen. *Operating System Principles.* Prentice Hall, 1973.

[14] 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.

[15] H. Härtig, M. Hohmuth, J. Liedtke, 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.

[16] H. Härtig, J. Löser, F. Mehnert, L. Reuther, M. Pohlack, and A. Warg. An I/O Architecture for Microkernel-Based Operating Systems, July 2003. Technical Report. TU Dresden.

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

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

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

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

[25] 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.

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

[27] 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.

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