The Architecture of a Reliable Operating System

Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum

Dept. of Computer Science, Vrije Universiteit Amsterdam, The Netherlands
{jnherder, herbertb, beng, philip, ast}@cs.vu.nl

Keywords: Operating System Reliability, Design and Implementation, Multiserver Architecture, MINIX 3

Abstract

In this paper, we discuss the architecture of a fully modular, self-healing operating system, which exploits the principle of least authority to provide reliability beyond that of most other operating systems. The system can be characterized as a minimal kernel with the entire operating system running as a set of compartmentalized user-mode servers and drivers.

By moving most of the code 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, the system has been designed to survive and automatically recover from failures in critical modules, such as device drivers, transparent to applications and without user intervention.

We used this design to develop a highly reliable, open-source, POSIX-conformant member of the UNIX family, which is freely available and has been downloaded 50,000 times in the past 3 months.

1 INTRODUCTION

Operating systems are expected to function flawlessly, but, unfortunately, most of today's operating systems frequently fail. As discussed in Sec. 2, many problems stem from the monolithic design that underlies most common systems. All operating system functionality, for example, runs in kernel mode without proper fault isolation, so that any bug can potentially trash the entire system.

Like other groups, we believe that reducing the operating system kernel is a first important step in the direction of designing for reliability. In particular, running drivers and other core components in user mode helps to minimize the damage that may be caused by bugs in such code. However, our system explores an extreme in the design space of UNIX-like operating systems where the entire operating system is run as a collection of independent, tightly restricted, user-mode processes. This structure, combined with several explicit mechanisms for transparent recovery from crashes and other failures, results in a highly reliable, completely multiserver operating system that still looks and feels like UNIX.

While some of the mechanisms are well-known, and multiserver operating systems have been first proposed years ago, to the best of our knowledge, we are the first to explore such an extreme decomposition of the operating system that is designed for reliability, while providing reasonable performance. Quite a few ideas and technologies have been around for a long time, but were often abandoned for performance reasons. We believe that the time has come to reconsider the choices that were made in common operating system design.

1.1 Contribution

The contribution of this work is the design and implementation of an operating system that takes the concept of multiserver to an extreme in order to provide a dependable computing platform. The concrete goal of this research is to build a UNIX-like operating system that can transparently survive crashes of critical components, such as device drivers.

As we mentioned earlier, the answer that we came up with is to break the system into manageable units and rigidly control the power of each unit. The ultimate goal is that a fatal bug in, say, a device driver should not crash the operating system; instead, the failed component should be automatically and transparently replaced by a fresh copy, and running user processes should not be affected.

To achieve this goal, our system provides: simple, yet efficient and reliable IPC; disentangling of interrupt handling from user-mode device drivers; separation of policies and mechanisms; flexible, run-time operating system configuration; decoupling of servers and drivers through a publish-subscribe system; and error detection and transparent recovery for common drivers failures. We will discuss these features in more detail in the rest of the paper.

While microkernels, user-mode device drivers, multiserver operating systems, fault tolerance, etc.

are not new, no one has put all pieces together. We believe that we are the first to realize a fully modular, open-source, POSIX-conformant operating system that is designed to be highly reliable. The system is called MINIX 3. It has been released (with all the source code) and 50,000 people have downloaded it so far, as discussed later.

1.2 Paper Outline

We first introduce how operating system structures have evolved over time (Sec. 2). Then we proceed with a discussion of the kernel and the organization of the user-mode servers on top of it (Sec. 3). We review some implementation issues (Sec. 4) and briefly discuss the system's reliability (Sec. 5) and performance (Sec. 6). Finally, we draw conclusions (Sec. 7).

2 RELATED WORK

This section illustrates the operating system design space with three typical structures and some variants thereof. While most structures are probably familiar to the reader, we introduce them explicitly to show an overview of the design space that has monolithic systems at one extreme and ours at the other.

It is sometimes said that virtual machines and exokernels provide sufficient isolation and modularity for making a system safe. However, these technologies provide an interface to an operating system, but do not represent a complete system by themselves. The operating system on top of a virtual machine or exokernel can have any of the following structures.

2.1 Monolithic Systems

Monolithic kernels provide rich and powerful abstractions. All operating system services are provided by a single, monolithic program that runs in kernel mode; applications run in user mode and can request services directly from the kernel.

Monolithic designs have some inherent problems that affect their reliability. All operating system code, for example, runs at the highest privilege level without proper fault isolation, so that any bug can potentially trash the entire system. With millions of lines of code (LoC) and 1-16 bugs per 1000 LOC [13, 14], monolithic systems are likely to contain many bugs. Running untrusted, third-party code in the kernel also diminishes the system's reliability, as evidenced by the fact that 70% to 85% of all operating system crashes are caused by device drivers [2, 11]

From a high-level reliability perspective, a monolithic kernel is unstructured. The kernel may be partitioned into domains but there are no protection barriers enforced between the components. Two simplified examples, Linux and MacOS X, are given in Fig. 1.

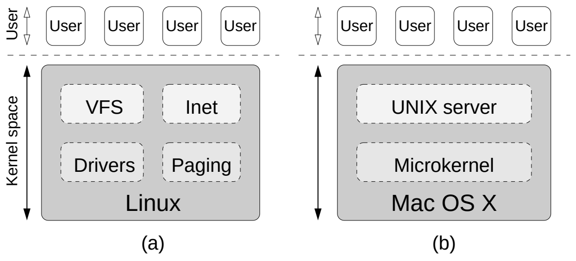

Figure 1: Two typical monolithic systems discussed in Sec. 2.1: (a) Vanilla Linux and (b) Mac OS X.

2.2 Single-Server Systems

A single-server system has a reduced kernel, and runs a large fraction of the operating system as a single, monolithic user-mode server. In terms of reliability, this setup adds little over monolithic systems, as there still is a single point of failure. The only gain in case of an operating system crash is a faster reboot.

An advantage of this setup is that it preserves a UNIX environment while one may experiment with a microkernel approach. The combination of legacy applications and real-time or secure modules allows for a smooth transition to a new computing environment.

Mach-UX [1] was one of the first systems to run BSD UNIX in user-mode on top of the Mach 3 microkernel, as shown in Fig. 2(a). Another example, shown in Fig. 2(b), is Perseus [9], running Linux and some specialized components for secure digital signatures on top of the L4 microkernel.

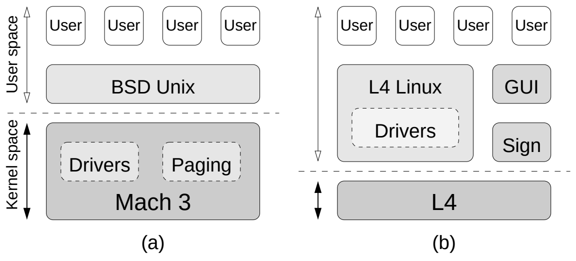

Figure 2: Two typical single-server systems discussed in Sec. 2.2: (a) Mach-UX and (b) Perseus.

2.3 Multiserver Systems

In a multiserver design, the operating system environment is formed by a set of cooperating servers. Untrusted, third-party code such as device drivers can be run in separate, user-mode modules to prevent faults from spreading. High reliability can be achieved by applying the principle of least authority [10], and tightly controlling the powers of each.

A multiserver design also has other advantages. The modular structure, for example, makes system administration easier and provides a convenient programming environment. A detailed discussion is out of the scope of this paper.

Several multiserver operating systems exist. An early system is MINIX [12], which distributed operating system functionality over two user-mode servers, but still ran the device drivers in the kernel, as shown in Fig. 3(a). More recently, IBM Research designed SawMill Linux [3], a multiserver environment on top of the L4 microkernel, as illustrated in Fig. 3(b). While the goal was a full multiserver variant of Linux, the project never passed the stage of a rudimentary prototype, and was then abandoned when the people working on it left IBM.

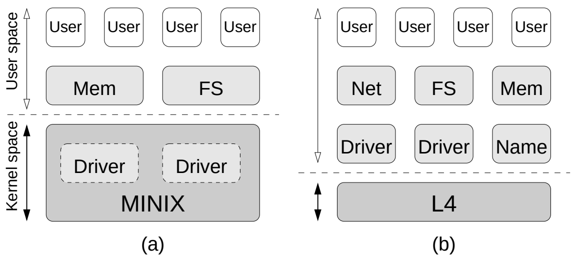

Figure 3: Two typical multiserver systems discussed in Sec. 2.3: (a) MINIX and (b) SawMill Linux.

2.4 Designing for Reliability

Although several multiservers systems exist, either in design or in prototype implementation, none of them has high reliability as an explicit design goal. In the rest of this paper, we present a new, highly reliable, open-source, POSIX-conformant multiserver operating system that is freely available for download, and has been widely tested.

Some commercial systems like Symbian OS and QNX [7] are also based on multiserver designs. However, they are proprietary and distributed without source code, so it is difficult to verify the claims. Still, the fact that innovating companies use multiserver designs, demonstrates the viability of the approach.

We will now discuss our multiserver architecture in detail, and show why it is a reliable system and how it can automatically recover from common failures.

3 THE ARCHITECTURE OF MINIX 3

The MINIX 3 operating system runs as a set a user-mode servers on top of a small kernel of under 4000 lines of code (LoC). Most of the servers are relatively small and simple. Their sizes approximately range from 1000 to 3000 LoC per server, which makes them easy to understand and maintain. The core components of MINIX 3 are shown in Fig. 4.

The general design principle that led to the above set of servers is that each process should be limited to its core business. Having small, well-defined services helps to keep the implementation simple and understandable. As in the original UNIX philosophy, each server has limited responsibility and power.

Before we continue with the discussion of the components of MINIX 3, we give some examples to illustrate how our multiserver operating system actually works. Fig. 4 also shows some typical IPC interactions initiated by user processes. Although the POSIX operating system interface is implemented by multiple servers, system calls are transparently targeted to the right server by the system libraries. Four examples are given below:

(1) The application that wants to create a child process calls the fork() library function, which sends a request message to the process manager (PM). PM verifies that a process slot is available, asks the memory manager (MM) to allocate memory, and instructs the kernel to create a copy of the process. Finally, PM sends the reply and the library function returns. All message passing is hidden to the application.

(2) A read() or write() call to do disk I/O, in contrast, is sent to FS. If the requested block is available in the buffer cache, FS asks the kernel to copy it to the user. Otherwise it first sends a message to the disk driver asking it to retrieve the block from disk. The driver sets an alarm, commands the disk controller through a device I/O request to the kernel, and awaits the hardware interrupt or timeout notification.

(3) Additional servers and drivers can be started on the fly by requesting the reincarnation server (RS). RS then forks a new process, assigns all needed privileges, and, finally, executes the given path in the child process (not shown in the figure). Information about the new system process is published in the data store (DS), which allows parts of the operating system to subscribe to updates in the system's configuration.

(4) Although not a system call, it is interesting to see what happens if a user or system process causes an exception, for example, due to an invalid pointer. In this event, the kernel's exception handler notifies PM, which transforms the exception into a signal or kills the process when no handler is registered. Recovery from operating system failures is discussed below.

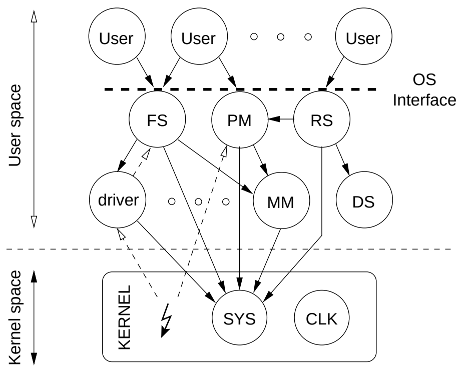

Figure 4: The core components of the full multiserver operating system, and some typical IPC paths. Top-down IPC is blocking, whereas bottom-up IPC is nonblocking.

3.1 The MINIX 3 Kernel

The MINIX 3 kernel provides privileged operations that cannot be done in user space in a portable way to support the rest of the operating system. The kernel is responsible for low-level interrupt handling, programming the MMU, interprocess communication (IPC), device I/O, and starting and stopping processes.

In addition, the kernel maintains several lists and bitmaps to restrict the powers of all system processes, for example, IPC primitives, IPC endpoints, I/O ports, IRQ lines, and memory regions. The policies are set by a trusted user-space server when a new system process is started, and enforced by the kernel at run time.

The kernel runs two independently scheduled processes called tasks (to distinguish them from the user-mode servers). Although the tasks are in kernel address space and run in kernel mode, they are treated in the same manner as any other user processes. The kernel tasks, for example, can be preempted, which helps to achieve a low interrupt latency.

3.1.1 System Task (SYS)

SYS is the interface to the kernel for all user-mode servers and drivers. All kernel calls in the system library are transformed into request messages that are sent to SYS, which processes the requests, and sends reply messages. SYS never takes initiative by itself, but it is always blocked waiting for a new request.

A typical example of a kernel call is SYS_FORK that is called when a new process must be created. The kernel calls handled by SYS can be grouped into several categories, including process management, memory management, copying data between processes, device I/O and interrupt management, access to kernel data structures, and clock services.

3.1.2 Clock Task (CLOCK)

CLOCK is responsible for accounting of CPU usage, scheduling another process when a process' quantum expires, managing watchdog timers, and interacting with the hardware clock. It does not have a publicly accessible user interface like SYS.

When the system starts up, CLOCK programs the hardware clock's frequency and registers an interrupt handler that is run on every clock tick. The handler only increments a process' CPU usage and decrements the scheduling quantum. If the a new process must be scheduled or an alarm is due, a notification is sent to CLOCK to do the real work at the task level.

Although CLOCK has no direct interface from user space, its services can be accessed through the kernel calls handled by SYS. The most important call is SYS_ALARM that allows system processes to schedule a *synchronous alarm* that causes a 'timeout' notification upon expiration.

3.2 The User-Space Servers

On top of the kernel, we have implemented a multiserver operating system. The servers run in user mode and are restricted in what they can do, just like ordinary user applications. They can use the kernel's IPC primitives, however, to request services from each other and the kernel. Below we will discuss the core operating system servers shown in Fig. 4.

3.2.1 Process Manager (PM)

PM is responsible for process management such as creating and removing processes, assigning process IDs and priorities, and controlling the flow of execution. Furthermore, PM maintains relations between processes, such as process groups and parent-child blood lines. The latter, for example, has consequences for exiting processes and accounting of CPU time.

Although the kernel provides mechanisms, for example, to set up the CPU registers, PM implements the process management policies. As far as the kernel is concerned all processes are similar; all it does is schedule the highest-priority ready process.

Signal Handling PM is also responsible for POSIX signal handling. When a signal is to be delivered, by default, PM either ignores it or kills the process. Ordinary user processes can register a signal handler to catch signals. In this case, PM interrupts pending system calls, and puts a signal frame on the stack of the process to run the handler. This approach is not suitable for system processes, however, as it interferes with IPC. Therefore, we implemented an extension to the POSIX sigaction() call so that system processes can request PM to transform signals into notification messages. Since event notification messages have the highest priority of all message types, signals are delivered promptly.

3.2.2 File Server (FS)

FS manages the file system. It is an ordinary file server that handles standard POSIX calls such as open(), read(), and write(). More advanced functionality includes support for symbolic links and the select() system call. FS is also the interface to the network server.

For performance reasons, file system blocks are buffered in FS' buffer cache. To maintain file system consistency, however, crucial file system data structures use write-through semantics, and the cache is periodically written to disk.

Since the file server runs as an isolated process that is fully IPC driven, it can be replaced with a different one to serve other file systems, such as FAT. Moreover, it should be straightforward to transform FS into a network file server that runs on a remote host.

Device Driver Handling Because device drivers can be dynamically configured, FS maintains a table with the mapping of major numbers onto specific drivers. As discussed below, FS is automatically notified of changes in the operating system configuration through a publish-subscribe system. This decouples the file server and the drivers it depends on.

A goal of our research is to automatically recover from common driver failures without interrupting processes and without human intervention. When a disk driver failure is detected, the system can recover transparently by replacing the driver and rewriting the blocks from FS' buffer cache. For character devices, transparent recovery sometimes is also possible. Such failures are pushed to user space, but may be dealt with by the application if the I/O request can be reissued. A print job, for example, can be reissued by the print spooler system.

3.2.3 Memory Manager (MM)

To facilitate ports to different architectures, we use a hardware-independent, segmented memory model. Memory segments are contiguous, physical memory areas. Each process has a text, stack, and data segment. System processes can be granted access to additional memory segments, such as the video memory or the RAM disk memory. Although the kernel is responsible for hiding the hardware-dependent details, MM does the actual memory management.

MM maintains a list of free memory regions, and can allocate or release memory segments for other system services. Currently MM is integrated into PM and provides support for Intel's segmented memory model, but work is in progress to split it out and offer limited virtual memory capabilities.

We will not support demand paging, however, because we believe physical memory is no longer a limited resource in most domains. We strive to keep the code simple and eliminate complexity whenever possible. Likewise we did not implement swapping segments to disk in the interest of simplicity.

3.2.4 Reincarnation Server (RS)

RS is the central component responsible for managing all operating system servers and drivers. While PM is responsible for general process management, RS deals with only privileged processes. It acts as a guardian and ensures liveness of the operating system.

Administration of system processes also goes through RS. A utility program, service, provides the user with a convenient interface to RS. It allows the administrator to start and stop system services, (re)set their policies, or gather statistics. For optimal flexibility in specifying policies a shell script can be set to run on certain events, including device driver crashes.

Fault Set The fault set that RS deals with are protocol errors, transient failures, and aging bugs. Protocol errors mean that a system process does not adhere to the multiserver protocol, for example, by failing to respond to a request. Transient failures are problems caused by specific configuration or timing issues that are unlikely to happen. Aging bugs are implementation problems that cause a component to fail over time, for example, when it runs out of buffers due to memory leaks.

Logical errors where a server or driver perfectly adheres to the specified system behavior but fails to perform the actual request are excluded. An example of a logical error is a printer driver that accepts a print job and confirms that the printout was successfully done, but, in fact, prints garbage. Such bugs are virtually impossible to catch in any system.

Fault Detection and Recovery During system initialization RS adopts all processes in the boot image as its children. System processes that are started later, also become children of RS. This ensures immediate *crash detection*, because PM raises a SIGCHLD signal that is delivered at RS when a system process exits.

In addition, RS can check the liveness of the system. If the policy says so, RS does a periodic *status check*, and expects a reply in the next period. Failure to respond will cause the process to be killed. The status requests and the consequent replies are sent using a nonblocking event notification.

Whenever a problem is detected, RS can replace the malfunctioning component with a fresh copy from disk. The associated policy script, however, might not restart the component, which is useful, for example, for development purposes. Another policy might use a binary exponential backoff protocol when restarting components to prevent clogging the system due to repeated failures. In any event, the problems are logged so that the system administrator can always find out what happened. Optionally, an e-mail can be sent to a remote administrator.

3.2.5 Data Store (DS)

DS is a small database server with publish-subscribe functionality. It serves two purposes. First, system processes can use it to store some data privately. This redundancy is useful in the light of fault tolerance. A restarting system service, for example, can request state that it lost when it crashed. Such data is not publicly accessible.

Second, the publish-subscribe mechanism is the glue between operating system components. It provides a flexible interaction mechanism and elegantly reduces dependencies by decoupling producers and consumers. A producer can publish data with an associated identifier. A consumer can subscribe to selected events by specifying the identifiers or regular expressions it is interested in. Whenever a piece of data is updated DS automatically broadcasts notifications to all dependent components.

Naming Service IPC endpoints are formed by the process and generation numbers, which are controlled and managed by the kernel. Because every process has a unique IPC endpoint, system processes cannot easily find each other. Therefore, we introduced stable identifiers that consist of a natural language name plus an optional number. The identifiers are globally known. Whenever a system process is (re)started RS publishes its identifier and the associated IPC endpoint at DS for future lookup by other system services.

In contrast to earlier systems, such as Mach [1], our naming service is a higher-level construction that is realized in user space. Mach's naming service required bookkeeping in the kernel and did not solve the problems introduced by exiting and reappearing system services. We have intentionally pushed all this complexity to user space.

Error Handling Since fault tolerance is an explicit design goal, DS is an integral part of the design. Its publish-subscribe mechanism makes it very suitable to inform other processes of changes in the operating system. Moreover, recovery of, say, a driver is made explicit to the services that depend on it.

For example, FS subscribes to the identifier for the disk drivers. If a disk driver crashes and RS registers a new one, DS notifies FS about the event. FS then calls back to find out what happened, and takes further action to recover from the failure.

3.2.6 Device Drivers

All operating systems hide the raw hardware under a layer of device drivers. Consequently, we have implemented drivers for ATA, S-ATA, floppy, and RAM disks, keyboards, displays, audio, printers, serial line, various Ethernet cards, etc.

Although device drivers can be very challenging, technically, they are not very interesting in the operating system design space. What is important, though, is each of ours runs as an independent user-mode process to prevent faults from spreading outside its address space and make it easy to replace a crashed or looping driver without a reboot. While other people have measured the performance of user-mode drivers [8], no currently-available system is self healing like this.

We are obviously aware that not all bugs can be eliminated by restarting a failed driver, but since the bugs that make it past driver testing tend to be timing bugs or memory leaks rather than algorithmic bugs, a restart often does the job.

4 IMPLEMENTATION ISSUES

As a base for the prototype, we started with MINIX 2 [12] due to its very small size and long history. We fully revised the system—moving all drivers out of the kernel and adding various new servers such as RS and DS—to arrive at MINIX 3. This approach allowed us to implement a new, highly reliable multiserver operating system, without having to write large amounts of code not relevant to this project.

A detailed discussion of the problems that were encountered [5] is outside the scope of this paper. However, a brief discussion of the dependencies we found when we moved device driver out of the kernel—and the solution we chose to resolve them—might benefit other operating systems developers.

Types of Interdependencies We analyzed the interdependencies and were able to group them into roughly five categories. These categories show who depends on whom or what, and each the dependencies in each category can be tackled in a different way.

(A) Many drivers directly call kernel functions or touch kernel variables, for example, to copy data to and from user-mode processes or set a watchdog timer at CLOCK. 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. Again, new new message types can be defined to request services from each other.

(C) The kernel can depend on driver, for example, when a timer expires and the watchdog function of a driver is called. Such asynchronous events that originate in the kernel are communicated to higher-level processes with nonblocking notification 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 and process it local to the driver in user space.

(E) Bad design, for example, local variables that are globally declared. This is yet another example why in-kernel drivers are not a good idea. Kernel development is complex and does not enforce a proper coding style, easily leading to mistakes. Fortunately, these dependencies were easily fixed.

Functional Classification Several functional classes could be formed by grouping similar dependencies. This classification was helpful to find general approaches to resolve entire classes of dependencies, instead of ad hoc solutions for individual dependencies.

The functional classes that we distinguished are: device I/O, copying data, access to kernel information, interrupt handling, system shutdown, clock services, debug dumps, and assertions and panics.

5 RELIABILITY

One of the strengths of our system is that it moves device drivers and other operating system functionality out of the kernel into unprivileged user-mode processes and introduces protection barriers between all modules. Therefore, it is no longer required to trust the entire operating system, but only the kernel and the servers and drivers that are needed for a given task. The small size of these parts—about 1000 to 4000 lines of code—makes it practical to verify the code either manually or using formal verification tools.

Each process is encapsulated in a private address space that is protected by the MMU hardware. Invalid pointers or illegal access attempts are caught by MMU, just like for ordinary applications. The use of separate memory segments even protects against buffer overrun exploits that are commonly used by viruses and worms. Execution of injected code is no longer possible because the text segment is read-only and the stack and data segment are not executable. While other types of attacks exist, for example, the return-to-libc attack, they are harder to exploit.

Minimizing Privileges The user-mode operating system components do not run with superuser privileges. Instead, they are given an unprivileged user and group ID to restrict file system access and POSIX calls. In addition, each user, server, and driver process has a restriction policy, according to the principle of least authority [10]. The kernel maintains various lists and bitmaps that specify what each process can do and enforces this policy at run time.

Driver access to I/O ports and IRQ lines are assigned when they are started. In this way, if, say, the printer driver tries to write to the disk's I/O ports, the kernel can deny the access. Stopping rogue DMA is not possible with current hardware, but as soon as an I/O MMU is added, we can prevent that, too.

Furthermore, we tightly restrict the IPC capabilities of each process. For each process we specify what IPC primitives it may use and what IPC endpoints are allowed. This mechanism is used to enforce a multiserver protocol that specifies who can talk to whom.

We also restrict the kernel calls that are available to each server and driver, depending on their needs. Ordinary applications cannot request kernel services at all, but need to contact the POSIX servers instead.

Protecting the System While we do not claim that our system is free of bugs, in many cases we can recover from crashes due to programming errors or attempts to exploit vulnerabilities, transparent to applications and without user intervention. The RS server automatically replaces a system process that unexpectedly crashes, exits or otherwise misbehaves with a fresh copy, as discussed in Sec. 3.

6 PERFORMANCE

Multiserver systems based on microkernels have been criticized for decades because of alleged performance problems. To illustrate the case, BSD UNIX on top of the early Mach microkernel was well over 50% slower than the normal version of BSD UNIX, and led to the impression of microkernels being slow.

Modern multiserver systems, however, have proven that competitive performance actually can be realized. L^{4}Linux on top of L4, for example, has a performance loss of about 5% [4]. Another project recently demonstrated that a user-mode gigabit Ethernet can perform within 7% of a kernel-mode driver [8].

Performance Measurements We have done extensive measurements of our system and presented the results in a technical report [6]. We can summarize these results (done on a 2.2 GHz Athlon) as follows. The simplest system call, getpid, takes 1.011 microseconds, which includes two messages and four context switches. Rebuilding the full system, which is heavily disk bound, has an overhead of 7%. Jobs with mixed computing and I/O, such as sorting, sedding, grepping, prepping, and uuencoding a 64-MB file have overheads of 4%, 6%, 1%, 9%, and 8%, respectively. The system can do a build of the kernel and all user-mode servers and drivers in the boot image within 6 sec. In that time it performs 112 compilations and 11 links (about 50 msec per compilation). Fast Ethernet easily runs at full speed, and initial tests show that we can also drive gigabit Ethernet at full speed. Finally, the time from exiting the multiboot monitor to the login prompt is under 5 sec.

We have also measured the performance overhead of our recovery mechanisms by simulating repeated crashes during a transfer of a 512-MB file from the Internet with crash intervals ranging from 1 to 15 sec. The results are shown in Fig. 5. The transfer successfully completed in all cases, with a throughput degradation ranging from 25% to only 1%. The mean recovery time was 0.36 sec.

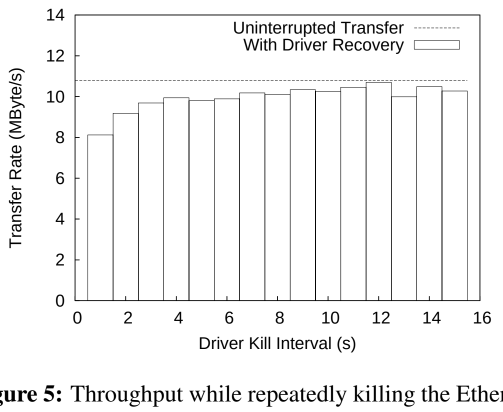

Figure 5: Throughput while repeatedly killing the Ethernet driver during a 512-MB transfer with various time intervals.

7 CONCLUSIONS

We have demonstrated that it is feasible to build a highly reliable, multiserver operating system with a performance loss of only 5% to 10%. We have discussed the design and implementation of a serious, stable, prototype that currently runs hundreds of standard UNIX applications, including two C compilers, language processors, many editors, networking, and all the standard shell, file, text manipulation, and other UNIX utilities.

Our system represents a new data point in the spectrum from monolithic to fully modular structure. The design of consists of a small kernel running the entire operating system as a collection of independent, isolated, user-mode processes. While people have tried to produce a fully modular microkernel-based UNIX clone with decent performance for years (such as GNU Hurd), we have actually done it, tested it heavily, and released it.

The kernel implements only the minimal mechanisms required to build an operating system upon. It provides interrupt handling, IPC, and scheduling, and contains two kernel tasks (SYS and CLOCK) to support the user-mode operating system parts. The core servers are the process manager (PM), memory manager (MM), file server (FS), reincarnation server (RS), and data store (DS). Since the size of the operating system components ranges from about 1000 to 4000 lines of code, it may be practical to verify them either manually or using formal verification tools.

Our multiserver architecture realizes a highly reliable operating system. We moved most operating system code to unprivileged user-mode processes that are encapsulated in a private address space protected by the MMU hardware. Each user, server, and driver process has a restriction policy to limit their capabilities according to the principle of least authority. We do not claim we have removed all the bugs, but the system is robust and self healing, so that it can withstand and often recover from common failures, transparent to applications and without user intervention.

8 AVAILABILITY

MINIX 3 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 50,000 people have downloaded the CD-ROM image in the past 3 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.

ACKNOWLEDGMENTS

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

REFERENCES

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

[2] 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.

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

[5] J. N. Herder. Towards a True Microkernel Operating System. Master's thesis, The Netherlands, Feb. 2005.

[6] J. N. Herder, H. Bos, and A. S. Tanenbaum. A Lightweight Method for Building Reliable Operating Systems Despite Unreliable Device Drivers. In *Technical Report*, Jan. 2006.

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

[8] 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.

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

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

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

[12] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems Design and Implementation*. Prentice-Hall, 2nd edition, 1997.

[13] 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.

[14] 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.