Reorganizing UNIX for Reliability

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

Computer Science Dept., Vrije Universiteit Amsterdam
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

Abstract

In this paper, we discuss the architecture of a modular UNIX-compatible operating system, MINIX 3, that provides reliability beyond that of most other systems. With nearly the entire operating system running as a set of user-mode servers and drivers atop a minimal kernel, the system is fully compartmentalized.

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 new design to develop a highly reliable, open-source, POSIX-conformant member of the UNIX family. The resulting system is freely available and has been downloaded over 75,000 times since its release.

1 INTRODUCTION

Operating systems are expected to function flawlessly, but, unfortunately, most of today's operating systems fail all too often. As discussed in Sec. 2, many problems stem from the monolithic design that underlies commodity 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 [1–4], we believe that reducing the operating system kernel and 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 position in the design space of UNIX-like systems, with almost the entire operating system running 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, multiserver operating system that still looks and feels like UNIX.

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 multiserver concept to its logical conclusion 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, a local failure should be detected and the failing component should be automatically and transparently replaced by a fresh copy without affecting user processes.

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 driver 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, POSIX-conformant operating system that is designed to be highly reliable. The system has been released (with all the source code available under the Berkeley license) and over 75,000 people have downloaded it so far, as discussed later.

1.2 Paper Outline

We first survey related work and show how operating system structures have evolved over time (Sec. 2). Then we proceed with an architectural discussion of the kernel and the user-mode servers and drivers on top of it (Sec. 3). We review the system's main reliability features (Sec. 4) and briefly discuss its performance (Sec. 5). Finally, we draw conclusions (Sec. 6) and mention how the system can be obtained (Sec. 7).

2 RELATED WORK

This section gives an overview of the design space that has monolithic systems at one extreme and ours at the other. We briefly discuss starting with the shortcomings of monolithic systems and ways to retrofit reliability. Then we survey increasingly modular designs that we believe will help to make future operating systems more reliable.

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, exokernel, or the bare hardware can have any of the structures discussed below.

2.1 Retrofitting Reliability in Legacy Systems

Monolithic kernels provide rich and powerful abstractions. All operating system services are provided by a single program that runs in kernel mode. A simplified example, vanilla Linux, is given in Fig. 1(a). While the kernel may be partitioned into domains, there are no protection barriers enforced between the components.

Monolithic designs have some inherent reliability problems. 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 executable code (LOC) and reported error rates up to sixteen or 75 bugs per 1000 LOC [5, 6], monolithic systems are prone to bugs. Running untrusted, third-party code in the kernel also diminishes the system's reliability, as evidenced by the fact that the error rate in device drivers is 3 to 7 times higher than in other code [7] and 85% of all Windows crashes are caused by drivers [8].

An important project to improve the reliability of commodity systems such as Linux is Nooks [8, 9]. Nooks keeps device drivers in the kernel but transparently encloses them in a kind of lightweight *protective wrapper* so that driver bugs cannot propagate to other parts of the operating system. All traffic between the driver and the rest of the kernel is inspected by the reliability layer.

Another project uses *virtual machines* to isolate device drivers from the rest of the system [2]. When a driver is called, it is run on a different virtual machine than the main system so that a crash or other fault does not pollute the main system. In addition to isolation, this technique enables unmodified reuse of device drivers when experimenting with new operating systems.

A recent project ran Linux device drivers in user mode with small changes to the Linux kernel [3]. This work shows that drivers can be isolated in separate user-mode processes without significant performance degradation.

While isolating device drivers helps to improve the reliability of legacy operating systems, we believe a proper, fully modular design from scratch gives better results. This includes encapsulating *all* operating system components (e.g., file system, memory manager) in independent, user-mode processes.

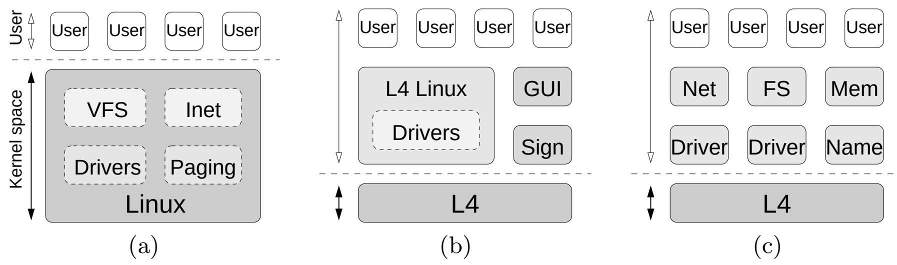


Fig. 1. Three increasingly modular designs of the Linux operating system: (a) Vanilla Linux (widely deployed); (b) L^4Linux and specialized components (working prototype); and (c) envisioned structure of SawMill Linux (abandoned project).

2.2 Architecting New Modular Designs

In modular designs, the operating system is split into a set of cooperating servers. Untrusted code such as third-party device drivers can be run in independent user-mode modules to prevent faults from spreading. In principle, modular designs have great potential to increase reliability as each module can be tightly confined according to the principle of least authority [10].

One approach is running the operating system in a single user-mode server on top of a microkernel, for example, L^4Linux on top of L4 [11]. This structure can be combined with specialized components as in DROPS [1] and Perseus [12], which is illustrated in Fig. 1(b). While the specialized components run in isolation, a single bug can still crash Linux and take down all legacy applications.

Some commercial systems like Symbian OS and QNX [13] are also based on multiserver designs, but do not use such an extreme decomposition of the operating system as we do. In Symbian OS, for example, only the file server and the networking and telephony stacks are hosted in user-mode servers, while the QNX kernel still contains process management and other functions which could have been isolated in separate user-mode processes.

SawMill Linux [14] would have been a more sophisticated approach to split the operating system into pieces and run each one in its own protection domain, as illustrated in Fig. 1(c). Unfortunately, the project was abruptly terminated in 2001 when many of the principals left IBM Research, and the only outcome was a rudimentary, unfinished prototype.

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

A recent multiserver system developed by Microsoft Research is Singularity [4]. In contrast to other systems, Singularity uses language protection and bypasses the hardware protection offered by the MMU. The system can be characterized as a microkernel running a set of verifiably safe, software-isolated servers. While language safety might be a viable approach to build reliable systems, Singularity means a paradigm shift for the programmer and is not backwards compatible with any existing applications.

2.3 What's Next?

Although several multiserver systems exist, either in design or in prototype implementation, none of them provides the highly reliable, UNIX-like environment that we strive for. Our approach to operating system reliability is practical for real-world adoption, as we reorganize only the internals of the operating system and do not change the interface offered to applications. Users can still run their favorite software, but now without rebooting their computer every now and then.

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.

3 OUR MULTISERVER ARCHITECTURE

In our design, called MINIX 3, the operating system runs as a set of user-mode servers and drivers on top of a tiny kernel, as illustrated in Fig. 2. The kernel is responsible for low-level and privileged operations such as programming the CPU and MMU, interrupt handling, and IPC, and contains two tasks (SYS and CLOCK) to support the user-mode parts of the operating system.

The simplest servers provide file system (FS), process management (PM), and memory management (MM) functionality. The data store (DS) is a small database server with publish-subscribe functionality. Finally, the reincarnation server (RS) keeps track of all servers and drivers and can transparently repair the system when certain failures occur.

Each component in our design is a small, well-defined entity with limited responsibility and power, as in the original UNIX philosophy. The kernel consists of under 4000 lines of executable code (LoC) and the sizes of the servers approximately range from 1000 to 3000 LoC per server, which makes them easy to understand and maintain. The small size also might make it practical to verify the code either manually or using formal verification tools.

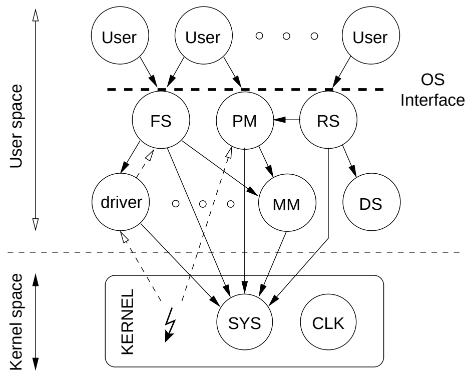


Fig. 2. The core components of the full multiserver operating system, and some typical IPC paths. Top-down IPC is blocking, whereas bottom-up IPC is nonblocking.

Before we continue with the discussion of the core components, we illustrate how our multiserver operating system actually works. Although the POSIX API is implemented by multiple servers, system calls are transparently targeted to the right server by the system libraries. Four examples are given below:

(1) An 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 (SYS) 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 is sent to FS. If the requested block is in the buffer cache, FS asks the kernel (SYS) to copy it to the user. Otherwise it 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 an I/O request to the kernel (SYS), and awaits the hardware interrupt or timeout notification.

(3) Additional servers and drivers can be started on the fly by requesting them from the reincarnation server (RS). RS then forks a new process, sets the process' privileges at the kernel (SYS), 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 a fatal 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 failures in servers and drivers is handled by RS and is discussed below.

3.1 The Kernel

The kernel can be characterized as a true microkernel and provides low-level operations that cannot be done by unprivileged user-mode processes. First, the kernel is responsible for low-level resource management and interaction with the hardware. For example, this includes interrupt handling, programming the CPU and MMU, device I/O, and process scheduling.

Second, the kernel provides a reliable set of interprocess communication (IPC) primitives. Our IPC design eliminates the need for dynamic resource allocation, both in the kernel and in user space. The standard request-reply sequence uses a rendezvous. If the destination is not waiting, the IPC_REQUEST blocks the sender until the IPC_REPLY has been sent. Similarly, a receiver is blocked on IPC_SELECT when no IPC is available. Messages are never buffered in the kernel, but always directly copied from sender to receiver, speeding up IPC and eliminating the possibility of running out of buffers. For special events, the IPC_NOTIFY primitive can be used to send nonblocking notification messages. Notifications are not susceptible to resource exhaustion either, since at most one bit per event is saved in a bitmap that is statically declared as part of the process table.

Third, the kernel maintains several lists and bitmaps to restrict the powers of all system processes. As discussed in Sec. 4, the restriction include IPC primitives that can be used, possible IPC destinations, kernel calls available, I/O ports, IRQ lines, and memory regions. The policies are set by a trusted user-space server (RS), and enforced by the kernel at run time. Each process has its own policy, allowing for fine-grained control of privileges in the system.

Fourth, two independently scheduled processes, SYS and CLOCK, are part of the kernel to support the rest of the operating system. These processes are 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.

System Task (SYS) SYS is the interface to the kernel for all user-mode servers and drivers that require low-level kernel-mode operations. All kernel calls in the system library are transformed into a request message that is sent to SYS, which handles the request if the caller is authorized and sends a reply. SYS never takes initiative by itself, but is always blocked waiting for a new work.

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. Some typical examples of kernel calls were already mentioned in the scenarios above: SYS_DEVIO to do device I/O, SYS_VIRCOPY to copy data using virtual addressing, SYS_SETALARM to schedule an alarm, and SYS_PRIVCTL to set a process' privileges.

Clock Task (CLOCK) CLOCK is responsible for accounting for CPU usage, scheduling another process when a process' quantum expires, managing watchdog timers, and interacting with the hardware clock. 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 interrupt handler only increments a process' CPU usage and decrements the scheduling quantum. If 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_SETALARM that allows system processes to schedule a *synchronous alarm* that causes a 'timeout' notification upon expiration. Since both tasks are in the kernel, SYS can directly call CLOCK's functions.

3.2 The User-Space Servers

On top of the kernel, we have implemented a POSIX-conformant multiserver operating system. All servers and drivers run as independent user-mode processes and are highly restricted in what they can do, just like ordinary user applications. The servers and drivers can cooperate using the kernel's IPC primitives to provide the functionality of an ordinary UNIX operating system. Below we will discuss the core operating system servers shown in Fig. 2 in detail.

Process Manager (PM) Together with FS, PM implements the POSIX interface that is available to application programs. 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 signaling the parent of exiting processes and accounting of CPU time.

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.

Although the kernel provides the low-level mechanisms, for example, to set up the CPU registers, PM implements all process management policies. As far as the kernel is concerned all processes are similar; all it does is schedule the highest-priority ready process. The higher-level process management provided by PM is responsible for the UNIX look and feel of our system.

Memory Manager (MM) To facilitate ports to different architectures, we use a hardware-independent, segmented memory model. Each process has a text segment, which can be shared with processes that execute the same program, and a stack and data segment. System processes can be granted access to additional memory segments, such as the video memory or the RAM disk memory. In addition, they are allowed to request chunks of free memory.

The text segment of all processes has read-only protection and the stack and data segments are not executable, which makes buffer overrun vulnerabilities harder to exploit by viruses and worms, since injected code cannot be executed directly. Other memory protection mechanisms are discussed in Sec. 4.

Although the kernel is responsible for hiding the hardware-dependent details such as programming the MMU, 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, but work is in progress to split it out and offer virtual memory capabilities.

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 supported includes symbolic links and the select() system call. 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.

Currently, our system offers only one file system—our own native file system—but work is in progress to transform FS into a virtual file system server (VFS) that supports multiple, different file system servers. Both VFS and each file server will be run as an isolated, user-mode process. The file system underneath each mount point will be served by a separate file server so that a file server failure can affect only a subtree of the virtual file system.

Since device drivers can be dynamically loaded in our system, each file server maintains a mapping of major numbers onto specific drivers. As discussed below, changes in the configuration are broadcast through a publish-subscribe system. This mechanism decouples the file servers and the drivers they depend on.

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: servers and drivers. It acts as a guardian and ensures liveness of the operating system. Administration of system processes also is done 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 and (re)set the policy script that is run on certain events, including driver crashes.

Fault detection and recovery works as follows. 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 to 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, but the precise actions taken can be different for each server and each driver. The associated policy (shell) script could restart the failed component, enter the failure in a system log, backup the core image of the failed component for later inspection, send an e-mail to a remote system administrator, or other things. If crashes reoccur, a binary exponential backoff protocol could be used to prevent bogging down the system with repeated recoveries. More details about the recovery process are given in Sec. 4.

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, but only to the process that stored it.

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.

Among other things, DS is used as a *naming service*. Because every process has a unique IPC endpoint that is automatically generated by the kernel, 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.

Device Drivers All operating systems hide the raw hardware under a layer of device drivers. To get started and prove that our principles work in practice, 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 that each of ours runs as an independent user-mode process to prevent faults from spreading and make it easy to replace failing driver without a reboot.

We are aware that not all bugs can be cured 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. Moreover, our system can take other measures as well, such as pinpointing the driver that is responsible for the failure and notifying a remote administrator.

4 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. This strong compartmentalization improves the system's reliability in various ways [15, 16]. Faults are properly isolated and the system can often gracefully recover by restarting the failed component rather than rebooting the entire computer.

4.1 Fault Isolation

The kernel and MMU hardware ensure that processes are fully isolated. Each server and driver is encapsulated in a private address space that is protected by the MMU hardware. Illegal access attempts are caught, just like for user applications. Processes can exchange data in a controlled way by using the kernel's virtual copy construct. Copying is possible only with a capability-like descriptor—created by the other party—that grants access to a precisely specified memory region. This prevents memory corruption, even in the light of malicious processes.

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 policies are set by RS and the kernel enforces them 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 will 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. A temporary solution that is possible in our system is to deny access to the DMA controller and, instead, have a trusted server to mediate any DMA attempts.

Furthermore, we tightly restrict the IPC and kernel call capabilities of each process. For each user, server, and driver process we specify which IPC primitives it may use, which IPC endpoints are allowed, and which kernel calls it can make, depending on their needs. Ordinary applications, for example, cannot request kernel services at all, but need to contact the POSIX servers instead.

4.2 Fault Resilience

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. As discussed in Sec. 3, the RS server executes a policy script when it detects a failure and can automatically replace a failed system process with a fresh copy.

Next to RS, DS is an integral part of our design for fault tolerance. Its publish-subscribe mechanism makes it very suitable to inform other processes of changes in the operating system. For example, FS subscribes to the identifier for the disk drivers. When a driver crashes and RS registers a new one, DS notifies FS about the event; FS then can take further action to recover from the failure.

Different recovery strategies are used depending on the kind of driver that fails [15]. When a block device driver failure is detected, the file server can recover transparently by retrying the I/O operation that failed. For character devices, the file server pushes errors to user space, but transparent recovery sometimes is also possible. A print job, for example, can be reissued by the print spooler system. Finally, for Ethernet drivers, transparent recovery is possible when a reliable transport protocol, such as TCP, is used. In this case, the network server (not discussed here) can retransmit lost packets.

The underlying assumption of our approach is that failures are transient and that restarting a component allows to fix the problem. The *fault set* that RS deals with are internal errors, timing failures, aging bugs, and attack failures. Internal errors mean that a system process encounters an exception and panics or gets killed, for example, because it dereferences an invalid pointer. Timing failures are caused by specific configuration or unexpected hardware timing issues. 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. Finally, attack failures are caused by malicious code, such as variations on the 'ping-of-death.'

Byzantine failures and logical errors where a server or driver perfectly adheres to the specified system behavior but fails to perform the actual request are excluded. For example, consider 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.

In principle, RS guards both servers and drivers, but our system currently is mainly designed to deal with device driver failures. If one of the core servers discussed in the previous section crashes, recovery is not (yet) possible and the system will be hampered. For example, a crash of PM or FS that together implement the POSIX interface will directly affect application programs. However, given that typically about 70% of the operating system consists of device drivers and that they have error rates 3 to 7 times higher than ordinary code [7], we have tackled an important class of problems with our design.

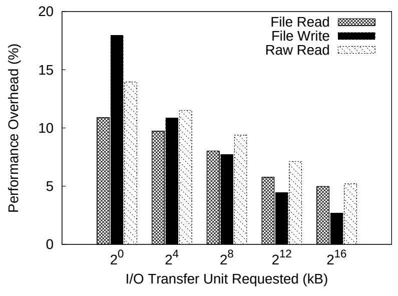
5 PERFORMANCE

Modular systems have been criticized for decades because of alleged performance problems. Modern multiserver systems, however, have proven that competitive performance can be realized [3, 11]. We have done extensive measurements of our system (on a 2.2 GHz Athlon), showing that the performance overhead compared to the base system with in-kernel drivers is limited to 5–10% [17].

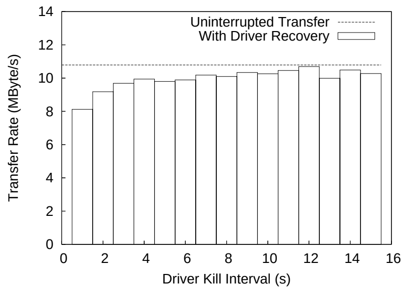
The simplest system call, `getpid()`, takes 1.011 microseconds, which includes passing two messages and two context switches. Rebuilding the full system, which is a heavily disk bound job, has an overhead of 7% compared to the base system with in-kernel device drivers. Jobs with mixed computing and I/O, 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 build 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). The overhead on disk transfer times of user-mode disk drivers is shown in Fig 3(a). Fast Ethernet easily runs at full speed, and initial tests show that we can also drive gigabit Ethernet at full speed from a user-mode driver. 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 of the Ethernet driver 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. 3(b). 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. This recovery time is due to the TCP retransmission timeout; restarting the failed driver takes only a few milliseconds.

It has to be noted that the overhead of the many new security checks is not related to the multiserver design per se and will be visible in any system. Furthermore, we did not yet do any performance optimizations. Careful analysis and removal of bottlenecks may bring the performance.



(a)



(b)

Fig. 3. Performance measurements: (a) Overhead of our user-mode disk driver compared to the in-kernel driver of the base system. (b) Throughput while repeatedly killing the Ethernet driver during a 512-MB transfer with various time intervals.

6 CONCLUSIONS

Our main contribution in the research presented in this paper is that we have actually built a highly reliable, UNIX-compatible multiserver operating system with a performance loss of only 5% to 10%. We have discussed the design and implementation of a useful and stable prototype that currently runs over 400 standard UNIX applications, including the X Window System, two C compilers, many editors, a complete TCP/IP stack that supports BSD sockets, and all the standard shell, file, text manipulation, and other UNIX utilities.

To achieve high reliability we have reorganized the monolithic design that is common to many UNIX operating systems. Our design consists of a small kernel running the entire operating system as a collection of independent, isolated, user-mode processes. The kernel implements only the minimal mechanisms, such as interrupt handling, IPC, policy enforcement, 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). The size of the kernel and the core servers ranges from about 1000 to 4000 lines of code.

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 powers to an absolute minimum. By fully compartmentalizing the operating system's device drivers, we were able to reduce the size of the TCB by over two orders of magnitude. We do not claim we have removed all the bugs, but the system is fault tolerant, and can withstand and often recover from common failures.

Given the low costs for this increase in operating system reliability and the fact that we were able to maintain the look and feel of an ordinary UNIX system, we believe that our reorganization of UNIX is practical for real-life adoption.

7 AVAILABILITY

The system is called MINIX 3 because we started with MINIX 2 as a base and then modified it very heavily. It is free, open-source software, available via the Internet. You can download MINIX 3 from the official homepage at: http://www.minix3.org/, which also contains the source code, documentation, news, contributed software packages, and more. Over 75,000 people have downloaded the CD-ROM image since the release (October 2005) 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

This work was supported by Netherlands Organization for Scientific Research (NWO) under grant 612-060-420.

REFERENCES

1. Härtig, H., Baumgartl, R., Borriss, M., Hamann, C.J., Hohmuth, M., Mehnert, F., Reuther, L., Schonberg, S., Wolter, J.: DROPS–OS Support for Distributed Multimedia Applications. In: Proc. 8th ACM SIGOPS Eur. Workshop. (1998) 203–209
2. LeVasseur, J., Uhlig, V., Stoess, J., Gotz, S.: Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In: Proc. 6th Symp. on Operating Systems Design and Implementation. (2004) 17–30
3. Leslie, B., Chubb, P., Fitzroy-Dale, N., Gotz, S., Gray, C., Macpherson, L., Daniel Potts, Y.T.S., Elphinstone, K., Heiser, G.: User-Level Device Drivers: Achieved Performance. Journal of Computer Science and Technology **20**(5) (2005)
4. Hunt, G.C., Larus, J.R., Abadi, M., Aiken, M., Barham, P., Fahndrich, M., Hawblitzel, C., Hodson, O., Levi, S., Murphy, N., Steensgaard, B., Tarditi, D., Wobber, T., Zill, B.: An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research (2005)
5. Basili, V., Perricone, B.: Software Errors and Complexity: An Empirical Investigation. Comm. of the ACM (1984) 42–52
6. 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, ACM (2002) 55–64
7. Chou, A., Yang, J., Chelf, B., Hallem, S., Engler, D.: An Empirical Study of Operating System Errors. In: Proc. 18th ACM Symp. on Operating System Principles. (2001) 73–88
8. Swift, M., Bershad, B., Levy, H.: Improving the Reliability of Commodity Operating Systems. ACM Trans. on Computer Systems **23**(1) (2005) 77–110
9. Swift, M., Annamalai, M., Bershad, B., Levy, H.: Recovering Device Drivers. In: Proc. 6th Symp. on Operating Systems Design and Implementation. (2004) 1–15
10. Saltzer, J., Schroeder, M.: The Protection of Information in Computer Systems. Proceedings of the IEEE **63**(9) (1975)
11. Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S., Wolter, J.: The Performance of -Kernel-Based Systems. In: Proc. 6th Symp. on Operating Systems Design and Implementation. (1997) 66–77
12. Pfitzmann, B., Stüble, C.: Perseus: A Quick Open-source Path to Secure Signatures. In: 2nd Workshop on Microkernel-based Systems. (2001)
13. Hildebrand, D.: An Architectural Overview of QNX. In: Proc. USENIX Workshop in Microkernels and Other Kernel Architectures. (1992) 113–126
14. Gefflaut, A., Jaeger, T., Park, Y., Liedtke, J., Elphinstone, K., Uhlig, V., Tidswell, J., Deller, L., Reuther, L.: The SawMill Multiserver Approach. In: ACM SIGOPS European Workshop. (2000) 109–114
15. Herder, J.N., Bos, H., Gras, B., Homburg, P., Tanenbaum, A.S.: MINIX 3: A Highly Reliable, Self-Repairing Operating System. ACM SIGOPS Operating System Review **40**(3) (2006)
16. Tanenbaum, A.S., Herder, J.N., Bos, H.: Can We Make Operating Systems Reliable and Secure? IEEE Computer **39**(5) (2006) 44–51
17. Herder, J.N., Bos, H., Tanenbaum, A.S.: A Lightweight Method for Building Reliable Operating Systems Despite Unreliable Device Drivers. In: Technical Report IR-CS-018 [www.cs.vu.nl/∼jnherder/ir-cs-018.pdf], Vrije Universiteit (2006)