JORRIT N. HERDER, HERBERT BOS,
BEN GRAS, PHILIP HOMBURG,
AND ANDREW S. TANENBAUM

modular system programming in MINIX 3

WHEN THE FIRST MODERN OPERAT- ing systems were being developed in the early 1960s, the designers were so worried about performance that these systems were written in assembly language, even though high-level languages such as FORTRAN, MAD, and Algol were well established. Reliability and security were not even on the radar. Times have changed and we now need to reexamine the need for reliability in operating systems.

Jorrit Herder holds a M.Sc. degree in computer science from the Vrije Universiteit in Amsterdam and is currently a Ph.D. student there. His research focuses on operating system reliability and security, and he is closely involved in the design and implementation of MINIX 3.

jnherder@cs.vu.nl

Herbert Bos obtained his M.Sc. from the University of Twente in the Netherlands and his Ph.D. from the Cambridge University Computer Laboratory. He is currently an assistant professor at the Vrije Universiteit Amsterdam, with a keen research interest in operating systems, high-speed networks, and security.

herbertb@cs.vu.nl

Ben Gras has a M.Sc. in computer science from the Vrije Universiteit in Amsterdam and has previously worked as a sysadmin and a programmer. He is now employed by the VU in the Computer Systems Section as a programmer working on the MINIX 3 project.

bjgras@cs.vu.nl

Philip Homburg received a Ph.D. from the Vrije Universiteit in the field of wide-area distributed systems. Before joining this project, he experimented with virtual memory, networking, and X Windows in Minix-vmd and worked on advanced file systems in the Logical Disk project.

philip@cs.vu.nl

Andrew S. Tanenbaum is a professor of computer science at the Vrije Universiteit in Amsterdam. He has written 16 books and 125 papers and is a Fellow of the ACM and a Fellow of the IEEE. He firmly believes that we need to radically change the structure of operating systems to make them more reliable and secure and that MINIX 3 is a small step in this direction.

ast@cs.vu.nl

If you ask ordinary computer users what they like least about their current operating system, few people will mention speed. Instead, it will probably be a neck-and-neck race among mind-numbing complexity, lack of reliability, and security in a broad sense (viruses, worms, etc.). We believe that many of these problems can be traced back to design decisions made 40 or 50 years ago. In particular, the early designers' goal of putting speed above all else led to monolithic designs with the entire operating system running as a single binary program in kernel mode. When the maximum memory available to the operating system was only 32K words, as was the case with MIT's first timesharing system, CTSS, multi-million-line operating systems were not possible and the complexity was manageable.

As memories got larger, so did the operating systems, until we got to the current situation of operating systems with hundreds of functions interacting in such complex patterns that nobody really understands how they work anymore. While Windows XP, with 5 million LoC (Lines of Code) in the kernel, is the worst offender in this regard, Linux, with 3 million LoC, is rapidly heading down the same path. We think this path leads to a dead end.

Various studies have shown the number of bugs in programs to be in the range 1–20 bugs per 1000 LoC [1]. Furthermore, operating systems tend to be trickier than application programs, and device drivers have an order of magnitude more bugs per thousand LoC than the rest of the operating system [2, 3]. Given millions of lines of poorly understood code interacting in unconstrained ways within a single address space, it is not surprising that we have reliability and security problems.

Operating System Reliability

In our view, the only way to improve operating system reliability is to get rid of the model of the operating system as one gigantic program running in kernel mode, with every line of code capable of compromising or bringing down the system. Nearly all the operating system functionality, and especially all the device drivers, have to be moved to user-mode processes, leaving only a tiny microkernel running in kernel mode. Moving the entire operating system to a single user-mode process as in L^{4}Linux [4] makes rebooting the operating system after a crash faster, but does not address the fundamental problem of every line of code being critical. What is required is splitting the core of the operating system functionality—including the file system, process management, and graphics—into multiple processes, putting each device driver in a separate process, and very tightly controlling what each component can do. Only with such an architecture do we have a chance to improve system reliability.

The reasons that such a modular, multiserver design is better than a monolithic one are threefold. First, by moving most of the code from kernel mode to user mode, we are not reducing the number of bugs but we are reducing the power of each bug to cause damage. Bugs in user-mode processes have much less opportunity to trash critical kernel data structures and cannot touch hardware devices they have no business touching. The crash of a user-mode process is rarely fatal, whereas a crash of the kernel always is. By moving most of the code out of the kernel, we are moving most of the bugs out as well.

Second, by breaking the operating system into many processes, each in its own address space, we greatly restrict the propagation of faults. A bug in the audio driver may turn the sound off, but it cannot wipe out the file system by accident. In a monolithic system, in contrast, bugs in any function can destroy code and data structures in unrelated and much more critical functions.

Third, by constructing the system as a collection of user-mode processes, the functionality of each module can be clearly determined, making the entire system much easier to understand and simpler to implement. In addition, the operating system's maintainability will improve, because the modules can be maintained independently from each other, as long as interfaces and shared data structures are respected.

While this article does not focus on security directly, it is important to mention that operating system reliability and security are closely related. Security has usually been designed with the model of the multi-user system in mind, not a single-user system where that user will run hostile code. However, many security problems are caused by malicious code injected by viruses and worms exploiting bugs such as buffer overruns. By moving most of the code out of the kernel, exploits of operating system components are rendered far less powerful. Overwriting the audio driver's stack may allow the intruder to cause the computer to make weird noises, but it does not compromise system security, since the audio driver does not have superuser privileges. Thus, while we will not discuss security much hereafter, our design has great potential to improve security as well.

The observation that microkernels are good for reliability is not new. In the 1980s and 1990s numerous microkernels were constructed, including L4 [5], Mach [6], V [7], Chorus [8], and Amoeba [9]. None of these succeeded in displacing monolithic operating systems with microkernel-based ones, but we have learned a lot since then and the time is right to try

again. Even Microsoft understands this. The next version of Windows (Vista) will feature many user-mode drivers, and Microsoft's Singularity research project is also based on a microkernel.

The MINIX 3 Architecture

To test out our ideas, we have constructed a POSIX-conformant prototype system. As a base for the prototype, we used the MINIX operating system due to its very small size and long history. MINIX is a free microkernel-based operating system that comes with complete source code, mostly written in C. The initial version was written by one of the authors (AST) in 1987, and has been studied by many tens of thousands of students at hundreds of universities for a period of 19 years; over the past 10 years there have been almost no bug reports concerning the kernel, presumably due to its small size.

We started with MINIX 2 and then modified it very heavily, moving the drivers out of the kernel and much more, but we decided to keep the name and call the new system MINIX 3. It is based on a microkernel now containing under 4000 LoC, with numerous user-mode servers and drivers that together constitute the operating system, as illustrated in Figure 1. Despite this unconventional structure, to the user the system appears to be just another UNIX variant. It runs two C compilers (ACK and gcc), as well as many popular utilities—Emacs, vi, Perl, Python, Telnet, FTP, and 300 others. Recently, X Windows has also been ported to it. MINIX 3 is available at http://www.minix3.org with all the source code under the BSD license.

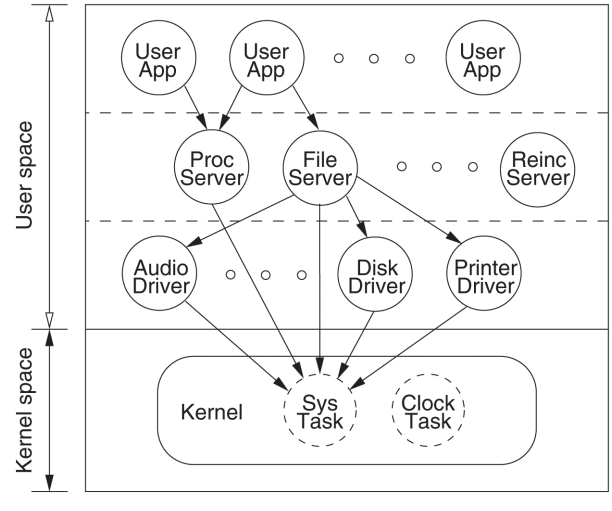

FIG. 1. SKETCH OF THE LAYERED ARCHITECTURE OF MINIX 3

All applications, servers, and drivers run as isolated, user-mode processes. A tiny, trusted kernel is the only part that runs in kernel mode. The layering is a logical one, as all user processes are treated equally by the kernel.

Briefly, the microkernel handles hardware interrupts, low-level memory management, process scheduling, and interprocess communication. The latter is accomplished by primitives that allow processes to send fixed-length messages to other processes they are authorized to send to. Most communication is synchronous, with a sender or receiver blocking if the other party is not ready. Sending a message takes about 500 nsec on a 2.2GHz Athlon. Although a system call usually takes two messages (a request and a reply), even 10,000 system calls/sec would use only 1% of the CPU, so message-passing overhead hardly affects performance at all. In

addition, there is a nonblocking event notification mechanism. Pending notifications are stored in a compact bitmap that is statically declared as part of the process table. This message-passing scheme eliminates all kernel buffer management and kernel buffer overruns, as well as many deadlocks.

The next level up contains the device drivers, one per major device. Each driver is a user process protected by the MMU the same way ordinary user processes are protected. They are special only in the sense that they are allowed to make a small number of kernel calls to obtain kernel services. Typical kernel calls are writing a set of values to hardware I/O ports or requesting that data be copied to or from a user process. A bitmap in the kernel's process table controls which calls each driver (and server) can make. Also, the kernel knows which I/O ports the driver is allowed to use, and copying is possible only with explicit permission.

The operating system interface is formed by a set of servers. The main ones are the *file server*, the *process manager*, and the *reincarnation server*. User processes make POSIX system calls by sending a message to one of these servers, which then carries out the call. The reincarnation server is especially interesting, since it is the parent process of all the servers and drivers. It is different from init, which is the root of ordinary user processes, as it manages and guards the operating system. If a server or driver crashes or otherwise exits, it becomes a zombie until the reincarnation server collects it, at which time the reincarnation server looks in its tables to determine what to do. The usual action is to create a new driver or server and to inform the other processes that it is doing so.

Finally, we have the ordinary user processes, which have the ability to send fixed-length messages to some of the servers requesting service, but basically have no other power. While message passing is used under the hood, the system libraries offer the normal POSIX API to the programmer.

Living with Programming Restrictions

Having explained why microkernels are needed and how MINIX 3 is structured, it is now time to get to the heart of this article: the MINIX 3 programming model and its implications. We will point out some of the properties, strengths, and weaknesses of the programming model in the text below, but before we start, it is useful to recall that, historically, restricting what programmers can do has often led to more reliable code. Let us consider several examples.

First, when the first MMUs appeared, user programs were forced to make system calls to perform I/O, rather than just start I/O devices themselves. Of course, some of them complained that making kernel calls was slower than talking to the I/O devices directly (and they were right), but a consensus eventually emerged saying that this restriction on what a programmer could do was worth the small performance penalty, since bugs in user code could no longer crash the computer.

Second, when E.W. Dijkstra wrote his now-famous letter "Goto Statement Considered Harmful" [10], a massive hue and cry was raised by many programmers who felt their style of writing spaghetti-like code was being threatened. Despite these initial objections, the idea caught on, and programmers learned to write well-structured programs.

Third, when object-oriented programming was introduced, many programmers balked at the idea, since they could no longer count on reading or tweaking data structures internal to other objects, a previously common practice in the name of efficiency. For example, when Java was introduced to C programmers, many of them saw it as a straitjacket, since they could no longer freely manipulate pointers. Nevertheless, object-oriented programming is now common and has led to better-quality code.

The MINIX 3 Restrictions

In a similar vein, the MINIX 3 programming model is also more restrictive for operating system developers than what came before it, but we believe these restrictions will ultimately lead to a more reliable system. For the time being, MINIX 3 is written in C, but gradually rewriting some of the modules in a type-safe language, such as Cyclone, might be possible someday. Let us start our overview of the model by looking at some of these restrictions.

Restricted kernel access. The MINIX 3 kernel exports various kernel calls to support the user-mode servers and drivers of the operating system. Each driver and server has a bitmap in the process table that restricts which of the kernel calls it may use. This protection is quite fine-grained, so, for example, a device driver may have permission to perform I/O or make copies to and from user processes, but not to shut down the system, create new processes, or (re)set restriction policies.

Memory protection. In the multiserver design of MINIX 3, all servers and drivers of the operating system run as isolated user-mode processes. Each is encapsulated in a private address space that is protected by the MMU hardware. An illegal access attempt to another process's memory raises an MMU exception and causes the offender to be killed by the process manager. Of course, the file system and device drivers need to interact with user processes to perform I/O, but this is done using safe virtual copies mediated by the kernel. A copy to another process is possible only when permission is explicitly given by that process or a trusted process such as the file system. This design takes away the trust from drivers and prevents memory corruption.

Restricted I/O port access. Each driver has a limited range of I/O ports that it may access. Since user processes do not have I/O privileges, the kernel has to mediate and can check whether the I/O request is permitted. The allowed port ranges are set when a driver is started. For ISA devices this is done with the help of configuration files; for PCI devices the port ranges are automatically determined by the PCI bus server. The valid port ranges for each driver are stored in the driver's process table entry in the kernel. This protection ensures that a printer driver cannot accidentally write garbage to the disk, because any attempt to write to the disk's I/O ports will result in a failed kernel call. Servers and ordinary user processes have no access to any I/O ports.

Restricted interprocess communication. Processes may not send messages to arbitrary processes. Again, the kernel keeps track of who may send to whom, and violations are prevented. The allowed IPC primitives and destinations are set by the reincarnation server when a new system process is started. For example, a driver may be allowed to communicate with just the file server and no other process. This feature eliminates some bugs where a process tries to send a message to another process that is not expecting it.

Operating System Development in User Space

Now let us look at some other aspects of the MINIX 3 programming model. While there are some restrictions, as pointed out above, we believe that programming in a multiserver operating system environment has many benefits and may lead to higher productivity and better code quality.

Short development cycle. The huge difference between a monolithic and a multiserver operating system immediately becomes clear when looking at the development cycle of operating system components. System programming on a monolithic system generally involves editing, compiling, rebuilding the kernel, and rebooting to test the new component. A subsequent crash will require another reboot, and tedious, low-level debugging usually follows, frequently without even a core dump. In contrast, the development cycle on a multiserver system like MINIX 3 is much shorter. Typically, the steps are limited to editing, compiling, testing, and debugging. We will elaborate on these steps below.

Normal programming model. Because drivers and servers are just ordinary user processes, they can use any libraries that are needed. In some cases, even POSIX system calls can be used. The ability to do these things can be contrasted with the more rigid environment available to programmers writing code for monolithic kernels. In essence, working in user mode makes programming easier.

No system downtime. The required reboots for a monolithic operating system effectively kick off all users, meaning that a separate development system is to be preferred. In MINIX 3, no reboots are required to test new components, so other users are not affected. Furthermore, bugs or other problems are isolated in the new components and cannot affect the entire system, because the new component is run as an independent process in a restricted execution environment. Problems thus cannot propagate as in a monolithic system.

Easy debugging. Debugging a device driver in a monolithic kernel is a real challenge. Often the system just halts and the programmer does not have a clue what went wrong. Using a simulator or emulator usually is of no use because typically the device being driven is something new and not supported by the simulator or emulator. In contrast, in the MINIX 3 model, a device driver is just a user process, so if it crashes, it leaves behind a core dump that can be inspected using all the normal debugging tools. In addition, the output of all printf() statements in drivers and servers automatically goes to a log server, which writes it to a file. After a failed run with the new driver, the programmer can examine the log to see what the driver was doing just before it died.

Low barrier to entry. Because writing drivers and servers is much easier than in conventional systems, researchers and others can try out new ones easily. Ease of experimentation can advance the field by allowing people with good ideas but little experience in kernel programming to try out their ideas and build prototypes they would not be able to construct with monolithic kernels. Although the hardest part of writing a new device driver may be understanding the actual hardware, other operating system components can be easy to realize. For example, the case study at the end of this article illustrates how semaphore functionality can be added to MINIX 3.

High productivity. Because operating system development in user space is easier, the programmer can get the job done faster. Also, since no lengthy

system build is needed once the bug has been removed, time is saved. Finally, since the system need not be rebooted after a driver crash, as soon as the programmer has inspected the core dump and the log and has updated the code, it is possible to test the new driver without a reboot. With a monolithic kernel, two reboots are often needed: one to restart the system after the crash and one to boot the newly built kernel.

Good accountability. When a driver or server crashes, it is completely obvious which one it is (because its parent, the reincarnation server, knows which process exited). As a consequence, it is much easier than in monolithic systems to pin down whose fault the crash was and possibly who is legally liable for the damage done. Holding the producers of commercial software liable for their errors, in precisely the same way as the producers of tires, medicines, and other products are held accountable, may improve software quality.

Great flexibility. Our modular model offers great flexibility and makes system administration much easier. Since operating system modules are just processes, it is relatively easy to replace one. It becomes easier to configure the operating system by mixing and matching modules. Furthermore, if a device driver needs to be patched, this can usually be done on the fly, without loss of service or downtime. Module substitution is much harder in monolithic kernels and often requires a reboot. Finally, maintenance also becomes easier, because all modules are small, independent, and well understood.

Case Study: Message-Driven Programming in MINIX 3

We will now evaluate the MINIX 3 programming model aided by a little case study that shows how semaphore functionality can be added to MINIX 3. Although this is easier than implementing a new file server or device driver, it illustrates some important aspects of MINIX 3.

Semaphores are positive integers, equal to or greater than zero, and support two operations, UP and DOWN, to synchronize multiple processes trying to access a shared resource. A DOWN operation on semaphore *S* decrements *S* unless *S* is zero, in which case it blocks the caller until some other process increments *S* through an UP operation. Such functionality is typically part of the kernel in a monolithic system, but can be realized as a separate user-space server in MINIX 3.

The structure of the MINIX 3 semaphore server is shown in Fig. 2. After initialization, the server enters a main loop that continues forever. In each iteration the server blocks and waits until a request message arrives. Once a message has been received, the server inspects the request. If the type is known, the associated handler function is called to process the request, and the result is returned unless the caller must be blocked. Illegal request types directly result in an erroneous reply.

As mentioned above, ordinary user processes in MINIX 3 are restricted to synchronous message passing. A request will block the caller until the response has arrived. We will use this to our advantage when constructing the semaphore server. For UP operations, the server simply increments the semaphore and directly sends a reply to let the caller continue. For DOWN operations, in contrast, the reply is withheld until the semaphore can be decremented, effectively blocking the caller until it is properly synchronized. The semaphore has an associated (FIFO) queue of processes to keep track of processes that are blocked. After an UP operation, the queue is checked to see whether a waiting process can be unblocked.

```
void semaphore_server( ) {
   message m;
   int result;
   /* Initialize the semaphore server. */
   initialize( );
   /* Main loop of server. Get work and process it. */
   while(TRUE) {

      /* Block and wait until a request message arrives. */
      ipc_receive(&m);

      /* Caller is now blocked. Dispatch based on message type. */
      switch(m.m_type) {
        case UP:     result = do_up(&m);     break;
        case DOWN: result = do_down(&m);   break;
        default:     result = EINVAL;
      }
      /* Send the reply, unless the caller must be blocked. */
      if (result != EDONTREPLY) {
          m.m_type = result;
          ipc_reply(m.m_source, &m);
      }
   }
}
```

FIG. 2. THE MAIN LOOP OF A SERVER IMPLEMENTING ONE SEMAPHORE, *S*

All servers and drivers have a similar main loop. The function initialize() *is called once before entering the main loop, but is not shown here. The handler functions do_up() and do_down() are given in Fig. 3.*

With the structure of the semaphore server in place, we need to arrange that user processes can communicate with it. Once the server has been started it is ready to serve requests. In principle, the programmer can construct request messages and send them to the new server using ipc_request(), but such details usually are conveniently hidden in the system libraries, along with the other POSIX functions. Typically, new library calls sem_up() and sem_down() would be added to libc to handle these calls. Although this case study covers a very simplified semaphore server, it can easily be extended to conform to the POSIX semaphore specification, handle multiple semaphores, etc.

The modular structure of MINIX 3 helps to speed up the development of the semaphore server in several ways. First, it can be implemented independently from the rest of the operating system, just like ordinary user applications. When it is finished, it can be compiled as a stand-alone application and be dynamically started to become part of the operating system. It is not necessary to build a new kernel or to reboot the system, which prevents downtime, other users from being kicked off, disruption of Web, mail, and FTP servers, etc. When the server is started, its privileges are restricted according to the principle of least authority, so that testing and debugging of the new semaphore server can be done without affecting the rest of the system. Once it is ready, the startup scripts can be configured to load the semaphore server automatically during operating system initialization.

```
int do_down(message *m_ptr) {
    /* Resource available. Decrement semaphore and reply. */
    if (s > 0) {
        s = s – 1;                  /* take a resource */
        return(OK);                 /* let the caller continue */
    }
    /* Resource taken. Enqueue and block the caller. */
    enqueue(m_ptr->m_source);  /* add process to queue */
    return(EDONTREPLY);        /* do not reply in order to block the caller */
}

int  do_up(message *m_ptr) {
message m;                      /* place to construct reply message */
    /* Add resource, and return OK to let caller continue. */
    s = s + 1;                  /* add a resource */

    /* Check if there are processes blocked on the semaphore. */
    if (queue_size() > 0) {     /* are any processes blocked? */
        m.m_type = OK;
        m.m_source = dequeue(); /* remove process from queue */
        s = s – 1;              /* process takes a resource */
        ipc_reply(m.m_source, m); /* reply to unblock the process */
    }
    return(OK);                 /* let the caller continue */
}
```

FIG. 3. up AND down OPERATIONS OF THE SEMAPHORE SERVER

The functions enqueue(), dequeue(), and queue_size() do list management and are not shown.

Conclusion

MINIX 3 is a new, fully modular operating system designed to be highly reliable. Like other innovations, our quest for reliability imposes certain restrictions upon the execution environment, but the multiserver environment of MINIX 3 makes life much easier for the OS programmer. The development cycle is shorter, system downtime is no longer required, the programming interface is more POSIX-like, and testing and debugging become easier. Programmer productivity is likely to increase, and code quality might improve because of better accountability. The system administrator also benefits, since MINIX 3 improves configurability and maintainability of the operating system. Finally, we have illustrated the message-driven programming model of MINIX 3 with the construction of a simple semaphore server and discussed how its development benefits from the modularity of MINIX 3. Interested readers can download MINIX 3 (including all the source code) from http://www.minix3.org. Over 50,000 people have already downloaded it; try it yourself.

REFERENCES

[1] T.J. Ostrand and E.J. Weyuker, "The Distribution of Faults in a Large Industrial Software System," *Proceedings of the SIGSOFT International Symposium on Software Testing and Analysis,* ACM, 2002, pp. 55–64.

[2] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An Empirical Study of Operating System Errors," *Proceedings of the 18th ACM Symposium on Operating System Principles,* 2001, pp. 73–88.

[3] M.M. Swift, M. Annamalai, B.N. Bershad, and H.M. Levy, "Recovering Device Drivers," *Proceedings of the 6th Symposium on Operating System Design and Implementation*, 2004, pp. 1–15.

[4] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter, "The Performance of µ-Kernel–Based Systems," *Proceedings of the 16th Symposium on Operating System Principles*, 1997, pp. 66–77.

[5] J. Liedtke, "On µ-Kernel Construction," *Proceedings of the 15th ACM Symposium on Operating System Principles*, 1995, pp. 237–250.

[6] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the USENIX 1986 Summer Conference*, 1986, pp. 93–112.

[7] D.R. Cheriton, "The V Kernel: A Software Base for Distributed Systems," *IEEE Software,* vol. 1, no. 2, 1984, pp. 19–42.

[8] 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," *Proceedings of the EurOpen Spring 1991 Conference*, 1991, pp. 13–32.

[9] S. Mullender, G. Van Rossum, A.S. Tanenbaum, R. Van Renesse, and H. Van Staveren, "Amoeba: A Distributed Operating System for the 1990s," *IEEE Computer Magazine,* vol. 23, no. 5, 1990, pp. 44–54.

[10] E.W. Dijkstra, "Goto Statement Considered Harmful," *Communications of the ACM*, vol. 11, no. 3, 1968, pp. 147–148.